Data Sorting Using Graphics Processing Units

Marko J. Mišić and Milo V. Tomašević

***Abstract* — Graphics processing units (GPUs) have been increasingly used for general-purpose computation in recent years. The GPU accelerated applications are found in both scientific and commercial domains. Sorting is considered as one of the very important operations in many applications, so its efficient implementation is essential for the overall application performance. This paper represents an effort to analyze and evaluate the implementations of the representative sorting algorithms on the graphics processing units. Three sorting algorithms (*Quicksort*, *Merge sort,* and *Radix sort*) were evaluated on the Compute Unified Device Architecture (CUDA) platform that is used to execute applications on NVIDIA graphics processing units. Algorithms were tested and evaluated using an automated test environment with input datasets of different characteristics. Finally, the results of this analysis are briefly discussed.**

***Keywords* — CUDA, data sorting, graphics processing units, parallel processing, parallel programming.**

I. Introduction

DATA sorting is a very frequent and compute-intensive operation, so its efficient implementation is of great importance in contemporary applications. It is one of the most studied activities, since the sorting algorithms are used in various domains and could be considered as the building blocks of more complex algorithms. There are many efficient sequential implementations, but with the emergence of contemporary parallel architectures, such as multicore central processing units (CPUs) or manycore graphics processing units (GPUs), the parallel implementations have been increasingly important.

In the past several years, significant improvements in multiprocessor architectures have been made. A very good example of such an evolution is the advance in the GPU field. In the early years, the GPUs have been specialized, fixed-function processors used primarily for 3D graphics rendering. Nowadays, the GPUs are highly parallel, multithreaded processor arrays capable of the execution of general-purpose, compute-intensive computations.

Numerous papers report high speedups obtained in various GPU accelerated applications. Sorting operations are the core parts in many of them, since these operations are used to optimize search and merging activities, to produce human-readable form of data, etc. Sorting operation is not only a part of many parallel applications, but also an important benchmark for parallel systems. It consumes a significant bandwidth for communication among processors since highly rated sorting algorithms access data in irregular patterns [1]. This is very important for the GPUs, since they need many calculations per one memory access to achieve their peak performance. Sorting algorithms usually do not have high computation to global memory access ratio, which puts the emphasis on algorithms that access the memory in favorable ways.

The goal of this paper is to present a short survey and performance analysis of sorting algorithms on graphics processing units. It presents three representative sorting algorithms (quicksort, merge sort, and radix sort) implemented using the CUDA platform to execute on modern GPUs. These algorithms were chosen for evaluation because they are known to a wider research community and their inherent characteristics allow efficient implementation on the GPUs.

The rest of the paper is organized as follows. The second section presents a short history of modern GPUs, and presents an overview of the CUDA platform that exploits the GPU computing power for general purpose computation. The third section covers some general sorting topics and briefly reviews related work in the field of sorting algorithms on the GPUs. The fourth section concentrates on the implementations of chosen sorting algorithms. Automated test environment and testing methodology are described in the fifth section with the emphasis on the input datasets used for performance evaluation. The sixth section presents and discusses experimental results. The final section draws some conclusions and proposes future work.

II. General Purpose Computation on GPUs

From the early days of their existence, the GPUs have been used for specialized, compute-intensive computations in the domain of computer graphics. Over the time, the GPUs offered programmability to some extent, through graphics application programming interfaces (APIs), such as OpenGL. Even at that point, some researchers tried to use an abundant computational power of those processors in general, non-graphics computations, which is described in [2] and [3]. It was clearly proved that the GPUs are especially good at data-parallel processing in certain domains, so that led to rapid development of the GPU architectures. The GPUs evolved to programmable, highly parallel multiprocessors with considerable computational power and high bandwidth, an order of magnitude higher than those of the contemporary CPUs. Unlike the CPUs,

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (III44009 and TR32039). The authors gratefully acknowledge this financial support.

Marko J. Mišić is with the School of Electrical Engineering, University of Belgrade, Serbia (phone: 381-11-3218-392; e-mail: marko.misic@etf.bg.ac.rs).

Milo V. Tomašević is with the School of Electrical Engineering, University of Belgrade, Serbia (phone: 381-11-3218-392; e-mail: mvt@etf.bg.ac.rs).

the GPUs have more transistors devoted to data processing rather than to data caching and sophisticated flow control. The GPU is a suitable platform for all compute-intensive problems that exhibit high regularity.

CUDA (*Compute Unified Device Architecture*) is a parallel computer architecture developed by NVIDIA. It significantly simplified the use of the GPUs for non-graphics purposes through its programming model and an extension of C language. For non-graphics computations, CUDA exposed a generic parallel programming model in a multithreaded environment, with support for synchronization, atomic operations, and eased memory access. Programmers do not need to use the graphics API anymore to execute non-graphics computation.

Typically, the CUDA programs are executed in coprocessing mode, where the GPU serves as a coprocessor to the CPU and accelerates the most time consuming parts of the application. The sequential parts of the application are executed on the CPU (*host*) and the compute-intensive parts are executed on the GPU (*device*). The applications are programmed through a simple API and C language extensions that target the parts of the code executed on the device.

The CPU initiates the execution on the device, being also responsible for data management and device configuration. The parallel parts of the application are executed on the device as special functions (*kernels*) called from the CPU side. A kernel is run in parallel by batches of lightweight threads executed on the processing units called streaming multiprocessors. To maintain scalability, a kernel execution is organized as a grid of thread blocks, as shown in Fig. 1. Every kernel is called with execution configuration that specifies the number of thread blocks in the grid and the number of threads in every thread block.

CUDA offers the dedicated memory hierarchy to support fast, parallel execution. All threads can access the global memory of the device. Global memory accesses are slow, so CUDA offers other smaller memories in the hierarchy to speed up the execution. Thus, threads have access to registers, local memory, shared memory, constant memory, texture memory, and global memory.

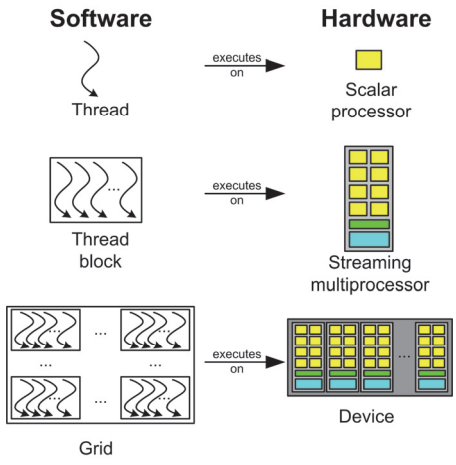

Fig. 1. CUDA execution model.

Threads in the same block can cooperate through a very fast shared memory and use barrier synchronization to coordinate their execution. Threads from different thread blocks cannot cooperate, since they may or may not execute on the same streaming multiprocessor. So, global synchronization is achieved only through repeated kernel calls, and that is one of the significant disadvantages of the CUDA platform that strongly affects the algorithm design. More information about the general-purpose computation on the GPUs and CUDA platform could be found in [4] and [5].

III. Sorting on the GPUs

Sorting is one of the most widely studied areas in the field of computer science. Lots of interesting and diverse solutions to this problem are proposed, but the advances in architectures make this topic still active. The raw processing power of graphics processing units attracted researchers, so some very efficient solutions, such as [6] and [7], were implemented even in the early days of GPU computing, when non-graphics applications were implemented using the graphics API.

Two approaches are commonly used in parallel sorting, as described in [8]. The merging approach sorts equally sized tiles locally in parallel, before proceeding recursively with a merge operation until the entire input array is sorted. On the contrary, the distribution approach reorders the keys globally into the buckets, such that all keys from one bucket are greater than those in the previous bucket and smaller from the ones in the next bucket. The procedure is carried on recursively on the buckets and the buckets are concatenated into the final sorted order. Quicksort and radix sort are the basic examples of the distribution sort algorithms, while merge sort is an example of the merging approach.

A. Sequential Algorithms

Quicksort is a divide-and-conquer algorithm which recursively splits the unsorted array into two partitions separated with the pivot element where the lower partition consists of the elements smaller than the pivot while the upper partition consists of the elements greater than the pivot. Since the partitions are independent, the algorithm could proceed with the same procedure in parallel, recursively, until all partitions collapse to a single element when the array is sorted.

Merge sort takes the advantage of the ease of merging shorter sorted sequences into a longer sorted sequence. Merge sort employs the merge operation to sort a sequence. It starts by comparing every two elements in the sequence and putting them in the correct order. It then merges each of the resulting sequences of two into the sequences of four and repeats the merging operation, and so on, until last two sorted sequences are merged into the final sorted sequence. Merge operation exhibits inherent parallelism, since it could be done through a merge tree.

Radix sort algorithm is based on the representation of keys as b-bit integers. This algorithm sorts the keys by examining groups of r bits in each pass, resulting in b/r

passes in total. In every pass, keys are classified into buckets and the input sequence is reordered. Internally, radix sort often uses the counting operation to sort a sequence in each pass. The counting operation is used to obtain the number of elements in every bucket depending on the group of bits being processed. Efficient implementation of the counting operation is the core part of the parallel radix sort algorithm.

B. Towards GPU Implementations

Quick sort and merge sort are comparison-based algorithms, while radix sort works on the bitwise representation of the keys. The comparison-based algorithms usually switch to some other strategy when a tile or bucket fits into the processor cache line or shared memory on the GPUs [8]. That fact revived some old, inherently parallel ideas, like bitonic networks [9], so the bitonic sorting is a preferable choice for many algorithms in their beginning or final stages.

Many GPU sorting algorithms use some parallel primitives to perform the sorting operation. Two very commonly used primitives are parallel reduction and parallel prefix sum (scan) operation. Parallel reduction operation reduces an array of values to a single value $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_n$, given some binary operator $\oplus$. Scan operation [10] takes an input of n elements $(x_0,..x_{n-1})$ and produces an output $(y_0,..y_{n-1})$, where $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$. Output can include or exclude the element x_i, and $\oplus$ is a binary operator.

C. Related Work

First attempts to implement an efficient sorting algorithm using the innovative CUDA architecture were made by Harris et al. to demonstrate their implementation of the efficient scan primitive [11]. They implemented radix sort and a hybrid merge sort algorithm. Cederman and Tsigas implemented GPU quicksort algorithm [12] with a three-level strategy, described in Section IV.A. Satish et al. demonstrated even faster merge sort and radix sort, published in [13]. They implemented an efficient parallel merge operation, described in Section IV.B. Leischner et al. developed the comparison-based sample sort [14] that outperforms the merge sort of [13] for about 30% for 32-bit integer keys. Dehne and Zaboli implemented the deterministic sample sort with similar results for uniformly distributed data [15].

Researchers paid even more attention to radix sort. Bandyopadhyay and Sahni developed the radix sort algorithm (GRS - GPU radix sort) suitable for sorting records with many fields [16]. Merrill and Grimshaw further improved the scan operations, which resulted in a highly optimized radix sort. At present, it is claimed to be the fastest GPU radix sort for 32-bit integers [17].

IV. Parallel Implementations of Sorting Algorithms

Although numerous papers reported various improvements in the domain of sorting algorithms (as reported in Section II), only several implementations are publicly available, and thus could be used for comparative analysis and evaluation. Quicksort implementation of Cederman and Tsigas [12] is the only publicly available quicksort implementation, although Harris et al. [11] also implemented quicksort to demonstrate the usage of their scan primitives. The only merge sort implementation available is a part of the Thrust library [18]. The Radix sort implementations are available through CUDPP library [19] and NVIDIA GPU Computing SDK, both based on the work of Harris et al. We have chosen CUDPP implementation for testing, because it is better supported for various key types. The main algorithm charcteristics are described in the following sections.

A. Quicksort

The general approach of this algorithm follows the guidelines given in Section III.A, but takes into account specific execution on a manycore processor like the GPU. The algorithm recursively divides the unsorted array into more and more progressively smaller partitions until the entire array is sorted. Every partition operation results in moving all elements less than a pivot to the positions left of the pivot and all elements greater than a pivot to the positions right of it. In every iteration of the partition operation a new pivot element is chosen and two new partitions are created that can be independently sorted. The partition operation is repeated until there are enough partitions to assign to one thread block. Then, a thread block can efficiently sort the assigned partition in the per-block shared memory. Since the thread blocks need to cooperate before they create enough independent partitions, the algorithm consists of two similar phases.

In the first phase, many thread blocks work together on the unsorted array. The only way to synchronize the threads from different thread blocks is through repeated kernel calls. The kernel calls are not an expensive operation on the GPU, but still not negligible, so they are used a minimal number of times. In the second phase, every thread block works on the given part of the array which consists of the elements greater than in the previous block and smaller than in the next thread block. Since there is no need for thread block synchronization, the second phase is entirely executed in one kernel call.

Both phases use the scan operation to determine the final position of an element in each phase. During the partition operation, every thread block counts the number of elements smaller than pivot and the number of elements greater than pivot for the given subsequence. Then, scan operation is performed and prefix sums are calculated for every element assigned to the thread. Additionally in the first phase, a global prefix sum across all blocks is calculated, in order to determine the final position of every element. At the end of iteration, the elements are reordered to their new positions in the array. In the final stage of the second phase, when subsequences are smaller than 1024 elements, the algorithm changes the strategy and uses bitonic sort to sort the remaining sequence. This is done because the overhead of the partition operation becomes too high.

On conventional processor architectures, it is desirable to perform in-place sorting, since it exploits spatial locality and makes better utilization of the cache memory. Since the GPUs do not have a cache memory in a traditional way, the sorting is not done in-place. Also, the in-place sorting would impose the synchronization of threads during memory access, which is expensive on GPU architectures. Instead, an additional buffer space is used, which enables the performance benefits from coalesced reads and writes. In each iteration, data is read from the primary buffer and the result is written to the auxiliary buffer. Then the two buffers switch places. A similar case is with the following algorithms.

B. Merge Sort

Since the direct manipulation of keys is not always allowed in the sorting operation, one alternative is to use an efficient comparison-based algorithm. A viable solution is to use divide-and-conquer merge sort, which is proved to be efficient on traditional architectures. Also, the merge operation is a frequent parallel primitive, so several implementations of merge operation could be found in the literature, and [13] gives a good overview of those techniques. The GPU execution can exploit fine-grain parallelism, so the implementations should be tuned accordingly.

Merge sort is frequently used as an external sorting algorithm, where the sequence being sorted is stored in a large external memory and the processor has direct access only to a much smaller memory [13]. A similar case is with the GPUs, since they have a large, but slow, global memory (up to 6 GB DRAM) and a small, on-chip shared memory (16 or 48 KB, depending on the configuration). Access time to a global memory is usually around 200 cycles, while shared memory access time is 3 to 4 cycles. This is the reason we need to split the data into blocks that fit into the shared memory, to sort it locally and then to move the data out. The whole process should be repeated as long as it is needed to sort the given sequence.

A merge sort algorithm consists of three phases. In the first phase, the input data are split into p equally sized blocks. In the second phase, all p blocks are sorted using p thread blocks (usually 256 threads each). In the final phase, sorted blocks are merged into the final sequence. On the block level, an alternative sorting method is used. At first, authors experimented with the common bitonic sort algorithm, but then they switched to odd-even merge sort as the authors claim that it is 5-10% faster in practice [13]. The most intensive part of the algorithm is the merge operation. The merge operation is implemented through a pair-wise merge tree, but since the number of pairs to be merged decreases geometrically, it was of great importance to implement the merge operation in such a way that would exploit fine-grain parallelism rather than the coarse-grain parallelism inherent to the merge tree.

The merging process is outlined as follows. If we have two sorted sequences A and B, each less than 256 elements in size, we can merge them using only one thread block. Since A and B sequences are accessed fairly randomly during the merging process, it is very important that sequences should fit into the fast shared memory. Each thread in the block takes an element from the A sequence and then determines the rank (position) of that element in the merged sequence. Since both A and B sequences are sorted, the final rank is easily determined using the rank of the element in the A sequence and parallel binary search to determine the rank of the same element in the B sequence. The elements of B sequence are merged in the same way.

If there are more than 256 elements in A and B sequences, they are split into a set of subsequences, using splitter elements. The splitters are chosen from the two sequences such that the interval between two successive splitters is small enough to be merged by a thread block.

C. Radix Sort

Radix sort assumes that the keys are d-digit numbers and sorts the sequence by processing one digit of the keys at a time, from the least to the most significant digit. To sort the digits within each of the d passes, counting sort or bucket sort are typically used. As described in Section III.A, an efficient radix sort implementation is heavily dependent on the counting operation. The counting operation is easy to parallelize on the GPUs, since it is efficiently supported with scan operations. However, it is of great importance to implement the operation in such a way as to utilize the fast shared memory.

This implementation processes 4-bit digits, so 8 passes are needed to sort 32-bit integers. In every pass of the algorithm, keys are reordered according to a processed digit. In order to do so, it is needed to determine the new rank (position) of each key in the sequence. The rank is determined by counting the number of keys less or equal but occurring earlier in the sequence than a given key. After all ranks are determined, keys are scattered to new positions in the sequence, and the process continues. This counting preserves the relative ordering of keys with equal digits, so sorting each digit from least to most significant is guaranteed to leave the sequence correctly sorted after all d passes are complete [13].

As key reordering is typically the process with an irregular memory access pattern, every pass is implemented in four steps, and utilizes a shared memory in order to avoid expensive global memory accesses. In the first step, a thread block locally reorders a given tile using the bit-split operation according to the digit being processed. Then, a histogram is computed for each tile counting the occurrences of the 2^b possible digits in the given tile. In the third step, prefix sums of the histograms of all tiles are computed, and finally, keys are reordered to their final positions using the computed prefix sums.

This approach has several advantages. It utilizes a global memory bandwidth efficiently, by minimizing the number of scatters to a global memory and maximizing the coherence of scatters. The coherent accesses (coalesced reads and writes) to the global memory are an important factor, since they can improve the bandwidth up to 10 times. The number of scatters to a global memory is

reduced by tiling (blocking of data) and using the higher radix ($b > 1$). Coherent accesses are achieved by locally reordering a given tile of data in the shared memory. Furthermore, this implementation uses 256 threads per thread block. One thread is given four elements to process, so one thread block is in charge of 1024 elements. Although it would be more logical to give every thread only one element, experiments have shown that this way every thread has a bit more sequential work to do, thus hiding the memory latency better.

V. Evaluation Environment

The testing and evaluation of the implemented algorithms was carried on in an automated environment, described in detail in [21]. This environment consists of several components that implement the different aspects needed to test and evaluate sorting algorithms. It includes a pseudorandom generator of different data distributions, the components for time measurment, regression testing to check the correctness of the results, and storing of the results of the experiments.

A. Input Dataset

Input data distribution is the characteristic that must be considered especially, since it can affect the algorithm performance. Sorting performance could heavily depend on the input data distribution, so a careful generation of test arrays is very important. Typically, the performance of algorithms that process a binary representation of keys, such as different variants of radix sort, is dependent on the key length. Some algorithms, such as quicksort, are sensitive to a non-uniform distribution in input arrays, especially when the number of different keys is limited. Also, some algorithms like merge sort can benefit from the presortedness of the input arrays, so all those characteristics were kept in mind when input data sets were chosen. Consequently, the evaluation was conducted using seven different data sets:

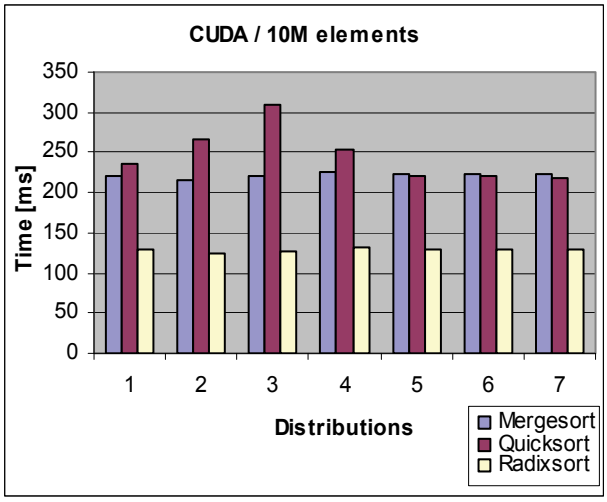

Fig. 2. Performance of the GPU sorting algorithms for different data distributions; 1 – 16-bit uniform distribution, 2 – 32-bit uniform distribution, 3 – Gaussian distribution, 4 – Few unique keys, 5 – Nearly sorted arrays, 6 – Sorted arrays, 7 – Reverse sorted arrays.

1. 16-bit and 32-bit keys of uniform distribution;
2. 32-bit integers of Gaussian distribution;
3. Few unique keys arrays;
4. Nearly sorted arrays;
5. Sorted and reverse sorted arrays.

Since other data distributions could be derived from the uniform distribution, such as normal distribution, it was of great importance to find a good pseudorandom generator. Also, the pseudorandom generator should be fast enough, since test arrays can have millions of elements. Mersenne Twister [20] was used in our experiments. It provides fast generation of very high-quality pseudorandom number, with a period of 2^{19937}-1.

B. Evaluation Methodology

The evaluation was done within our automated test environment. Every algorithm was run four times with different input arrays of different sizes. The array size was varied from 1K to 10M elements. After every sorting operation, a regression test was done, in order to check the correctness of every algorithm. A regression test is done using quicksort implementation (*qsort*) from the standard C library. Test results were stored in the CSV (Comma Separated Values) file, according to the previously defined format and then processed using MS Excel.

Windows XP OS was used to test the algorithms on the machine with processor Core 2 Duo E7600 3.06GHz and 4 GB RAM. CUDA programs were executed on the NVIDA GeForce GTS 250 with 512MB RAM using CUDA driver version 2.3.

VI. Experimental Results

After automated experiments were conducted, results were processed and analyzed. Fig. 2 shows a performance comparison of three algorithms for different data distributions. Generally, radix sort exhibits the best and most stable performance. This was expected, since this algorithm works on the binary representation of keys, and typically outperforms comparison-based algorithms for shorter keys. Quicksort and merge sort show a similar performance, but quicksort is more sensitive to input data distributions, especially for those with a limited number of

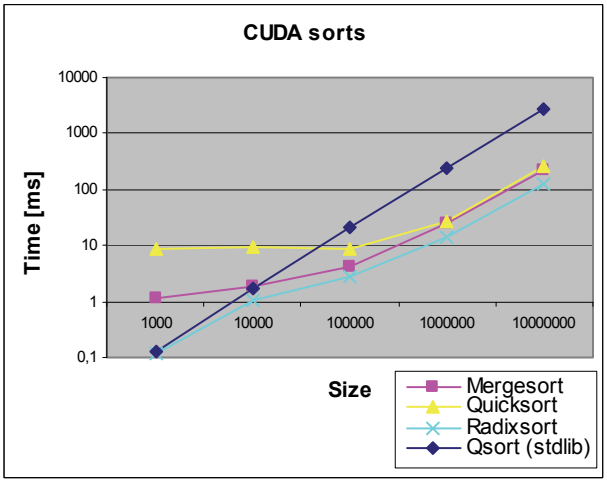

Fig. 3. Performance comparison of the GPU sorting algorithms and sequential (qsort) algorithm.

keys. It is a known drawback of the sequential quicksort algorithm, and it adheres to the GPU implemention to some extent, as well. Despite these performance pitfalls, the GPU algorithms are much less dependent on the input data distribution than their sequential implementations or their counterparts on other parallel platforms, like Message Passing Interface (MPI) or Pthreads [21]. This is mostly due to data-parallel nature and Singe Instruction Multiple Data (SIMD) execution of applications on the GPUs. The GPUs are manycore processors, and thus expose much finer granularity, so the applications could be tuned in a more sophisticated way.

Fig. 2 shows one more interesting fact. Standard C library sort function, *qsort*, is more efficient than the GPU implementations for short arrays, typically under 10K elements. In order to execute the applications on the GPU, we need to take some preparatory steps during the launch of the kernel. Also, memory transfers from CPU to GPU memory and vice-versa are proved to be costly, too. So, there is an amount of parallel overhead involved in the execution of every GPU algorithm. That time dominates in the overall execution time for smaller-sized arrays, while it is amortized for larger arrays.

Since one of the goals of parallel computing is to solve demanding problems within a given amount of time [4], a programmer should always be aware of the dimension of the problem to be solved, and choose a suitable algorithm accordingly. Because of the parallel overhead, which is inherent to parallelization process, the dimension of the problem should be large enough in order to obtain the benefit from parallel execution. Hybrid solutions that utilize both multithreading on the CPU and the GPU, depending on the problem size, could be developed and tuned empirically if there is a need to cover a wide range of input data sizes.

VII. Conclusion and Future Work

Modern graphics processing units provide an abundant processing power and the observed speedup could be an order of magnitude higher depending on the problem to be solved. On the other hand, CUDA technology offers the programming of the GPUs through a general API, which in turn considerably eases the use of the GPUs for general-purpose computations.

Sorting algorithms can achieve a good performance and scalability on the GPUs. Observed execution times are 3-5x higher than those of the sequential implementations, which is very important for this bandwidth consuming operation. On the other hand, the GPU implementations are more complex, typically several hundred lines of code, and not easy to understand and maintain. Thus, it is recommended to use available implementations, as they are quite sufficient for the majority of applications. From the platform point of view, the GPU implementations are much better than others as they offer a very good price-performance ratio.

Future work is envisaged in two directions. The first direction is to enhance the evaluation environment by including more diverse test cases. Those would include floating point keys, various key lengths (from 8-bit to 64-bit), and new data distributions, like staggered arrays and distributions with different key entropy levels. The other direction is to include some other implementations of the GPU sorting algorithms, such as sample sort or improved versions of radix sort.

References

[1] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "An Experimental Analysis of Parallel Sorting Algorithms," *Theory Comput. Systems 31*, pp. 135–167, 1998.

[2] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[3] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[4] D. B. Kirk and W. M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.

[5] „*NVIDIA CUDA C Programming Guide*", version 4.0, NVIDIA Corporaton, 2011.

[6] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management," *Proceedings of the 2006 ACM SIGMOD international conference on management of data*, pp. 325–336, 2006.

[7] A. Greba and G. Zachmann, "GPU-ABiSort: optimal parallel sorting on stream architectures," *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, pp. 27, 2006.

[8] N. Leischner, "GPU algorithms for comparison-based sorting and merging based on multiway selection," M.S. thesis, Karlsruhe Institute of Technology, 2010.

[9] K. E. Batcher, "Sorting networks and their applications," *Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968.

[10] G. E. Blelloch, "Prefix Sums and Their Applications," *Synthesis of Parallel Algorithms*, Morgan Kaufmann, pp. 35–60, 1990.

[11] S. Sengupta, M. Harris, Y. Zhang, and J.D. Owens, "Scan primitives for GPU computing," *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware*, pp. 97–106, 2007.

[12] D. Cederman and P. Tsigas, "A Practical Quicksort Algorithm for Graphics Processors," *Technical Report 2008-01*, Chalmers University of Technology, Sweeden, 2008.

[13] N. Satish, M. Harris, and M. Garland, "Designing Efficient Sorting Algorithms for Manycore GPUs," *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–10, 2009.

[14] N. Leischner, V. Osipov, and P. Sanders, "GPU sample sort," *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 1–10, 2010.

[15] F. Dehne and H. Zaboli, "Deterministic Sample Sort For GPUs," CoRR, abs/1002.4464, 2010.

[16] S. Bandyopadhyay and S. Sahni, "GRS - GPU Radix Sort for Large Multifield Records," *International Conference on High Performance Computing (HiPC)*, pp. 1–10, 2010.

[17] D. Merrill and A. Grimshaw, "Revisiting Sorting for GPGPU Stream Architectures," Technical Report CS2010-03, University of Virginia, USA, 2010.

[18] Project „Thrust 1.1", 2010., http://code.google.com/p/thrust/

[19] Project „CUDPP – CUDA Data Parallel Primitives Library 1.1", 2008., http://gpgpu.org/developer/cudpp

[20] M. Matsumoto and T. Nishimura, „Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," *ACM Trans. on Modeling and Computer Simulation Vol. 8*, no. 1, pp. 330, 1998.

[21] M. Mišić, "Comparative analysis of parallel sorting algorithms on different parallel platforms," M.S. thesis, School of Electrical Engineering, University of Belgrade, Belgrade, 2010. (in Serbian)