Proceedings of the Estonian Academy of Sciences,
2017, **66**, 3, 323–335
https://doi.org/10.3176/proc.2017.3.07
Available online at www.eap.ee/proceedings

COMPUTER SCIENCE

Fast iterative circuits and RAM-based mergers to accelerate data sort in software/hardware systems

Valery Sklyarov[a], Iouliia Skliarova[a], Artjom Rjabov[b]*, and Alexander Sudnitson[b]

[a] Department of Electronics, Telecommunications and Informatics/IEETA, University of Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; {skl, iouliia}@pb.ua.pt

[b] Department of Computer Systems, School of Information Technologies, Tallinn University of Technology, Akadeemia tee 15A, 12618 Tallinn, Estonia; aleksander.sudnitson@ttu.ee

Received 3 February 2017, accepted 14 March 2017, available online 6 July 2017

Abstract. The paper suggests and describes two architectures for parallel data sort. The first architecture is applicable to large data sets and it combines three stages of data processing: data sorting in hardware (in a Field-Programmable Gate Arrays – FPGA), merging preliminary sorted blocks in hardware (in the FPGA), and merging large subsets received from the FPGA in general-purpose software. Data exchange between the FPGA and a general-purpose computer is organized through a fast Peripheral Component Interconnect (PCI) express bus. The second architecture is applicable to small data sets and it enables sorting to be done at the time of data acquisition, i.e. as soon as the last data item is received, the sorted items can be transferred immediately. The results of experiments clearly demonstrate the advantages of the proposed architectures that permit the reduction of the required hardware resources and increasing throughput compared to the results reported in publications and software functions targeted to data sorting.

Key words: parallel data processing, merging, iterative networks, communication-time processing, Field-Programmable Gate Array (FPGA), Peripheral Component Interconnect (PCI) express bus.

1. INTRODUCTION

Sorting is a procedure that is needed in numerous computing systems [1,2]. For many practical applications, sorting throughput is very important. To better satisfy performance requirements, fast accelerators based on Field-Programmable Gate Arrays (FPGAs) (e.g. [3–11]), Central Processing Units (CPUs) (e.g. [7,12–16]), and multi-core CPUs (e.g. [17,18]) have been researched in depth. Two of the most frequently explored parallel sorters are based on sorting [1–3,19] and linear [4] networks. A sorting network is a set of vertical lines composed of comparators that can swap data to change their positions in the input multi-item vector. The data propagate through the lines from left to right to produce the sorted multi-item vector on the outputs of the rightmost vertical line. Three types of such networks have been studied: pure combinational (e.g. [3,9]), pipelined (e.g. [2,3,9]), and combined (partially combinational and partially sequential) [2,5,20]. The linear networks, which are often referred to as linear sorters [4], take a sorted list and insert new incoming items in the proper positions. The method is the same as the insertion sort [1] that compares a new item with all items in parallel, then inserts the

[*] Corresponding author, artjom.rjabov@ttu.ee

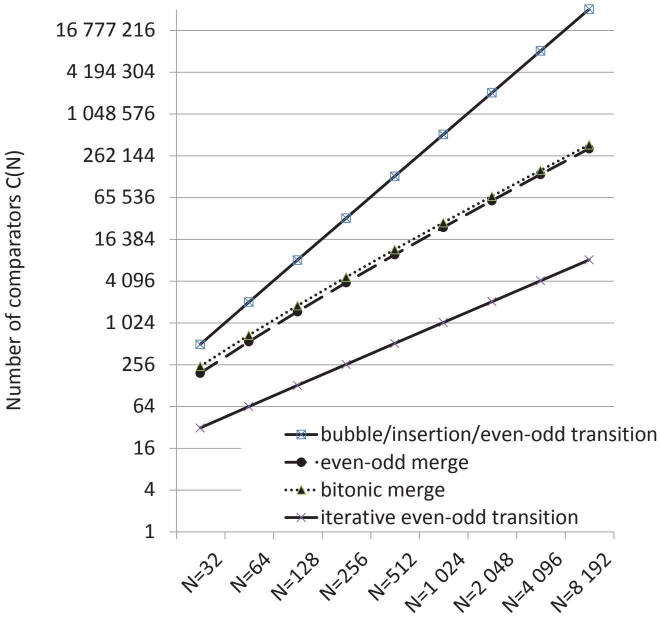

Fig. 1. The number of comparators for different values *N* of data items.

new item at the appropriate position, and shifts the existing elements in the entire multi-item vector. The main problem with this method is that it is applicable only to small data sets (see, for example, the designs discussed in [4], which accommodate only tens of items).

The majority of sorting networks implemented in hardware use Batcher even-odd and bitonic mergers [21]. Other types are rarer (see, for example, the comb sort [22] in [8], the bubble and insertion sort in [3,9], and the even-odd transition (transposition) sort in [12]). Research efforts are concentrated mainly on the following three directions: (1) networks with a minimal depth or number of comparators (e.g. [3,13]); (2) co-design, rationally splitting the problem between software and hardware (e.g. [3,9]), and (3) the regularity of the circuits and interconnections (e.g. [2,5]).

We target our results towards FPGAs because these devices are regarded more and more as a universal platform that enables computational algorithms to be significantly accelerated. The FPGAs still operate on a lower clock frequency than non-configurable Application-Specific Integrated Circuits (ASICs) and Application-Specific Standard Products (ASSPs) and broad parallelism is evidently required to compete with potential alternatives. Thus, sorting and linear networks can be seen as very adequate models. Unfortunately, they have many limitations. Suppose N data items, each of size M bits, need to be sorted. The results of [3,13] show that the most widely used sorting networks [19,21] cannot be built for $N > 128$ ($M = 32$), even in a relatively advanced FPGA because the hardware resources are not sufficient. Iterative networks from [2] enable the number of comparators $C(N)$ to be notably decreased but even after that we cannot sort more than 4096 items in the most advanced FPGAs, such as that from the Virtex-7 family of Xilinx. When N is increased, the complexity of the networks (the number of comparators/swappers $C(N4)$) grows rapidly [1–3,9,19] (see Fig. 1).

It is easy to conclude from Fig. 1 that sorting networks can be implemented in an FPGA only for a small number N of items while practical applications require millions of such items to be processed. One possible way is to sort relatively small subsets of larger sets in an FPGA and then to merge the subsets in software of a higher-level system (see Fig. 2). The initial set of data that is to be sorted is divided into Z subsets of N items. Each subset is sorted in an FPGA using the referenced networks. Merging is executed as shown in Fig. 2, in a host system/processor that interacts with the FPGA. Each horizontal level of merging permits the size of

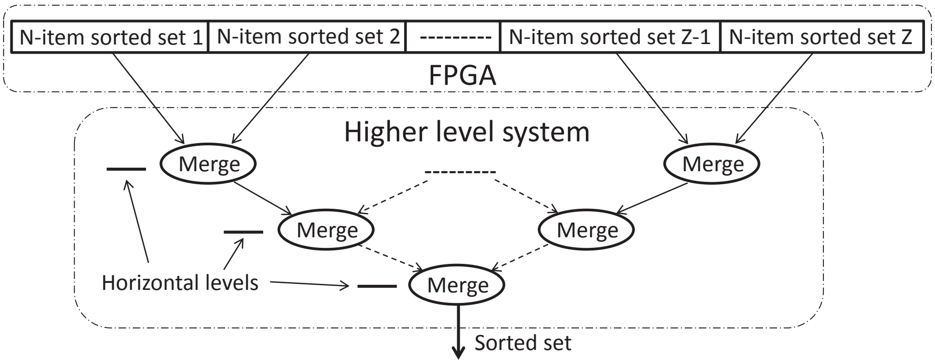

Fig. 2. The merging of sorted subsets in a software of a higher-level system.

blocks to be doubled. Thus, if $N = 2^{10} = 1024$ and $K = 2^{20} = 1\,048\,576$ items are to be sorted, then 10 levels of mergers are required (see Fig. 2). Clearly, the larger are the blocks sorted in FPGAs, the less merging is needed. Thus, we have to sort in hardware as many data items as possible with such throughput that is similar to the throughput of sorting networks. Besides, the networks [19,21] involve significant propagation delays through long combinational paths. Such delays are caused not only by comparators, but also by multiplexers that have to be inserted and by interconnections. Hence, clock signals with high frequency cannot be applied. Pipelining permits the clock frequency for circuits to be increased because delays between registers in a pipeline are reduced. A number of such solutions are described in [3,13]. However, once again, the complexity of the circuits becomes the main limitation. The analysis presented in [2] enables us to conclude the following: (1) the known even-odd merge and bitonic merge circuits [19,21] are the fastest and enable the best throughput to be achieved. However, they are very resource-consuming and can only be used effectively in existing FPGAs for sorting very small data sets; (2) pipelined solutions permit faster circuits than in point (1) to be designed. However, assuming that pipelining can be based on flip-flops in the used slices (so that additional slices are not required), resource consumption is at least the same as in point (1), therefore, in practice, only very small data sets can be sorted; (3) the existing even-odd merge and bitonic merge circuits are not very regular (compared to the even-odd transition network, for example) and, thus, the routing overhead may be significant in FPGAs.

There is also another problem that might arise. As a rule, initial data and final results are stored in conventional memories and each data item is kept at the relevant address of the memory. Suppose we would like to sort a set of data items. Let us look at Fig. 3 where the initial (unsorted) set is saved in the memory and the resulting (sorted) set is also saved in the memory. Parallel operations need to be applied to parallel subsets of data items, thus, in the beginning, initial data need to be unrolled (see Fig. 3) and the sorted items need to be stored in the memory one by one (see Fig. 3). Hence pre- and post-processing operations are involved and they (1) sequentially read unsorted data items and save them in a long-size input register and (2) copy the sorted data items from the long-size output register to conventional memories. These operations undoubtedly involve significant additional time. To reduce or even avoid such time, we have to be able to combine reading/writing data items and their sorting. We will call such type of data sorters communication-time data sorters.

This paper proposes a set of methods and device architectures with the following novel contributions:

1. The less resource-consuming iterative networks from [2] should be combined in hardware with pipelined Random Access Memory (RAM)-based data mergers, which permits
   (a) increasing the number of data items sorted in hardware significantly (more than one hundred times compared to [2]) without performance degradation,
   (b) performing data sorting in parallel with merging in hardware;

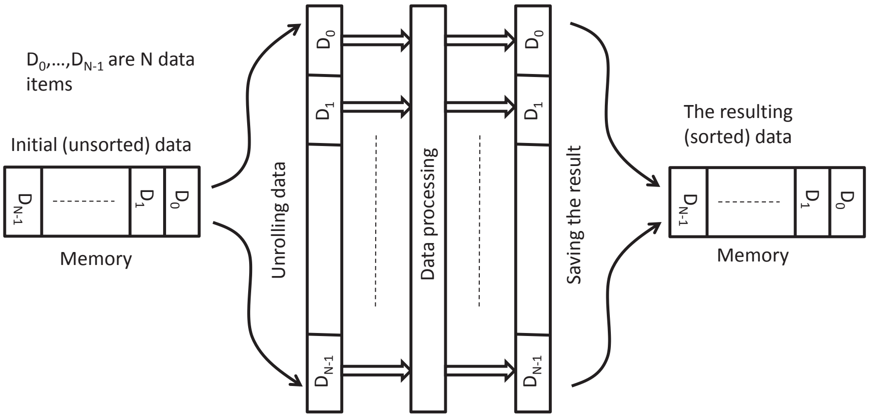

Fig. 3. Pre- and post-processing.

2. Communication-time data sorters that enable data acquisition and sorting to be executed in parallel in such a way that data sorting is completed as soon as the last data item has been received;
3. Three-level data sorters, two of which (network-based sorters and RAM-based mergers) are implemented in an FPGA and the last one – in a higher-level computing system that is in our case a general-purpose computer interacting with the FPGA through the Peripheral Component Interconnect (PCI) express bus.

2. SYSTEM ARCHITECTURE

Figure 4 depicts the considered system architecture. There are two basic subsystems that are a general-purpose computer (GPC) and an FPGA interacting through the PCI express bus. Let us assume that the FPGA can sort L blocks and each block contains up to N data items, i.e. such a number of items that can be sorted in the network [2]. The FPGA receives L blocks (containing up to N data items) from the GPC, sorts each block (see the rectangle A in Fig. 4), merges the sorted blocks (see the rectangle B in Fig. 4), and sends $L \times N$ sorted data items back to the GPC. The size M of each item is chosen to be 32 bits and it might be increased easily (FPGA circuits are easily scalable). Four 32-bit data items are packed in 128-bit words for data exchange through the PCI express bus.

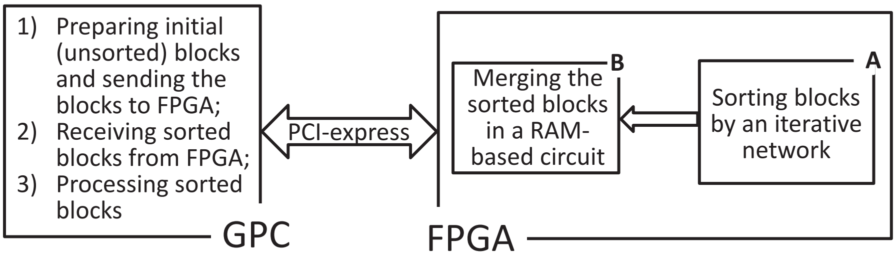
1) Preparing initial (unsorted) blocks and sending the blocks to FPGA;
2) Receiving sorted blocks from FPGA;
3) Processing sorted blocks
GPC
PCI-express
Merging the sorted blocks in a RAM-based circuit
B
Sorting blocks by an iterative network
A
FPGA

Fig. 4. General architecture of the considered system.

The FPGA implements circuits for the two levels referenced above, i.e. for an iterative sorter (see the rectangle A in Fig. 4) and for a merger (see the rectangle B in Fig. 4). In the beginning, we will use the network from [2] extended with some additional registers allowing data acquisition, sorting, and subsequent merging to be partially combined. Such an architecture implemented in the FPGA will be discussed in the following two subsections. The next section suggests some improvements of the network to design communication-time data sorters.

2.1. Iterative network for sorting data

Figure 5 depicts the used iterative network. The core of the network is the circuit proposed in [2]. There are also two additional registers R_i and R_o. The register R_i sequentially receives N data items from the GPC through the PCI express bus. It was explained above that such N items compose one block that can be entirely sorted in the network [2]. In practice, four items are packed and thus, parallel writing to the register R_i of four 32-bit items is actually done. As soon as the first block is received, all data items from this block are sorted in the iterative network from [2], and the maximum number of clock cycles is $N/2$ [2]. At the same time, data items from the next block are received from the GPC through the PCI express bus. As soon as data items from the first block are sorted, they are copied in parallel to the output register R_o. After that the second block is copied to the register R and sorted (see Fig. 5) and the third block is being received from the GPC through the PCI express bus. At the same time, the first sorted block is copied to the embedded block-RAM for subsequent merging. Hence, the first sorted block will be copied to RAM after the acquisition of two blocks from the PCI express bus. Then data acquisition from the GPC, data sorting, and copying data to the merger will be done in parallel. We can see from Fig. 5 that there are just two sequential levels of comparators/swappers in the iterative data sorter [2]. Thus, the delay is very small and we can apply high synchronization frequency. The results of [2] clearly demonstrate that such circuits are very efficient. Additional improvements are done to adjust the speed of data acquisition and sorting. Indeed, one block of N data items is received in $N/4$ clock cycles and the sorting time is up to $N/2$ clock cycles, i.e. it is almost two times longer.

Figure 6 demonstrates how to adjust the speed. There are now two iterative data sorters running in parallel. The first sorter processes data from the first half of the register R_i and the second sorter processes data from the second half of the register R_i. In the beginning, two blocks with $2 \times N$ items are copied to R_i and it involves $2 \times N/4 = N/2$ clock cycles. Then two blocks are sorted in parallel, which also involves up to $N/2$ clock cycles.

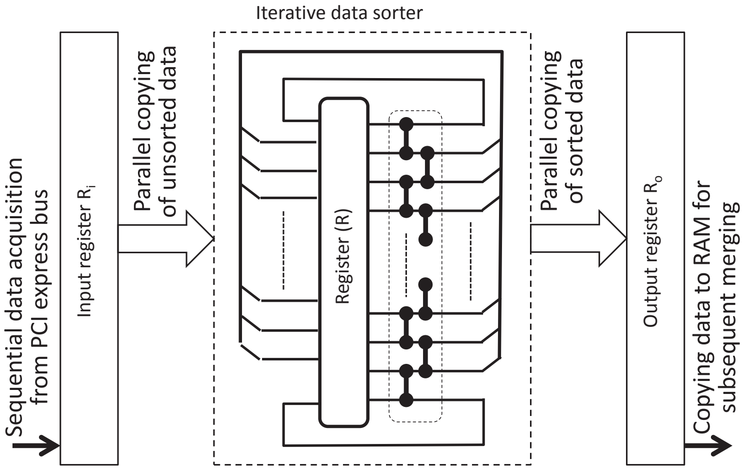
Iterative data sorter
Sequential data acquisition from PCI express bus
Input register R$_i$
Parallel copying of unsorted data
Register (R)
Parallel copying of sorted data
Output register R$_o$
Copying data to RAM for subsequent merging

Fig. 5. The circuit for sorting blocks.

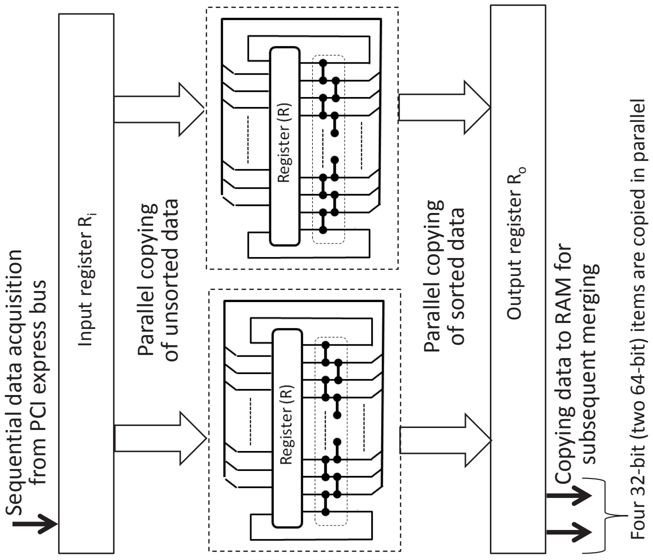
Sequential data acquisition from PCI express bus
Input register R$_i$
Parallel copying of unsorted data
Register (R)
Register (R)
Parallel copying of sorted data
Output register R$_o$
Copying data to RAM for subsequent merging
Four 32-bit (two 64-bit) items are copied in parallel

Fig. 6. Adjusting the number of clock cycles required in different blocks.

Finally, two sorted blocks are copied to two dual-port embedded block-RAMs. The respective write port is configured for data width 64. Thus, pairs of data items are copied in each clock cycle and it involves totally also $N/2$ clock cycles for both blocks. Therefore, everything is completely adjusted.

2.2. Pipelined merging

Merging is done on the basis of embedded block-RAM. Figure 7 shows one level of merging. Input data comes from two embedded block-RAMs, which is merged, and copied to a new embedded block-RAM. There are two address counters for each input RAM. In the beginning they are set to 0. Two data items are read and compared. If the item is selected from the first RAM, the address counter of the first RAM is incremented, otherwise the address counter of the second RAM is incremented. Two N-item blocks are merged in $2 \times N$ clock cycles. Different types of parallel merging have been verified and compared. We found that the best result (i.e. the fastest and the less resource-consuming) is produced in a simple RAM-based circuit depicted in Fig. 8.

There are G levels to merge L sorted blocks and $2^{G-1} < L \leq 2^G$. The first level is composed of L embedded block-RAMs. The second level is composed of $L/2$ embedded block-RAMs, and the last level is composed of one embedded block-RAM. The size of each RAM for the first level is N 32-bit words for reading and $N/2$ 64-bit words for writing. The size of each subsequent level is doubled. Initially, L embedded block-RAMs of the first level are filled in with sorted blocks. Then these blocks are merged at the second level. Afterwards the blocks of the second level are merged at the third level and at the same time the block-RAMs of the first level are being

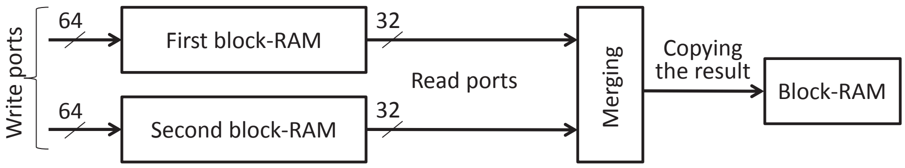

Fig. 7. Simple merging of two sorted blocks.

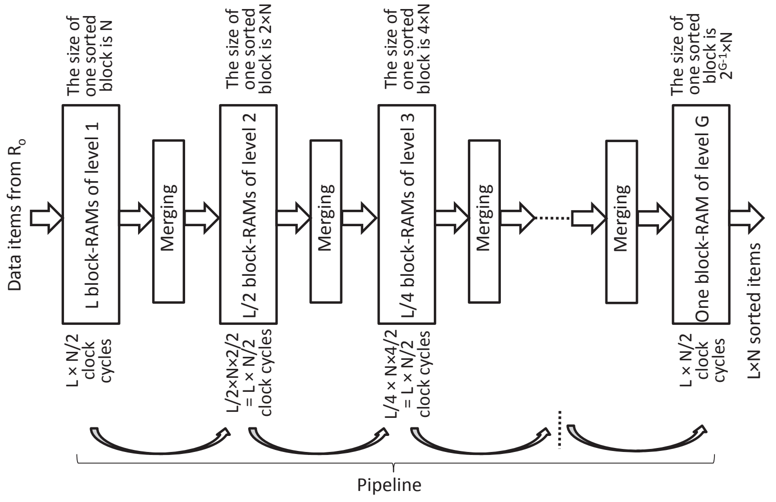
Data items from R$_o$
L block-RAMs of level 1
The size of one sorted block is N
L × N/2 clock cycles
Merging
L/2 block-RAMs of level 2
The size of one sorted block is 2×N
L/2×N×2/2 = L × N/2 clock cycles
Merging
L/4 block-RAMs of level 3
The size of one sorted block is 4×N
L/4 × N×4/2 = L × N/2 clock cycles
Merging
Merging
One block-RAM of level G
The size of one sorted block is 2^{G-1}×N
L × N/2 clock cycles
L×N sorted items
Pipeline

Fig. 8. Pipelined merging with embedded block-RAM.

filled in with a new subset of L sorted blocks. Thus, many subsets of L blocks will be processed in parallel and this is a special type of pipeline organized based on embedded block-RAMs (see Fig. 8).

The architecture in Fig. 8 permits many sets with L blocks (each block contains N M-bit data items) to be sorted in the pipeline in the way shown in Fig. 9. Equal numbers enclosed in circles indicate the steps executed in parallel. It was shown in the previous section (2.1) that the first time the level 1 block-RAM will be filled in with sorted data from the first block is after $3 \times N/2$ clock cycles. After that it is updated with the new block in $N/2$ clock cycles. So, an additional delay appears just from the beginning and it is avoided in the subsequent steps. As soon as data are copied to the first-level RAM, merging is started and the sorted data are copied from the first-level to the second-level RAM. This process involves $L \times N/2$ clock cycles. During this period of time the first-level RAM is used for merging and new data items cannot be copied to this RAM. In fact, it is possible to merge and to sort data at the same time. However, we found that such merger requires a complex arbitration which significantly increases hardware resources leading to reducing the size N of blocks. Finally, such more

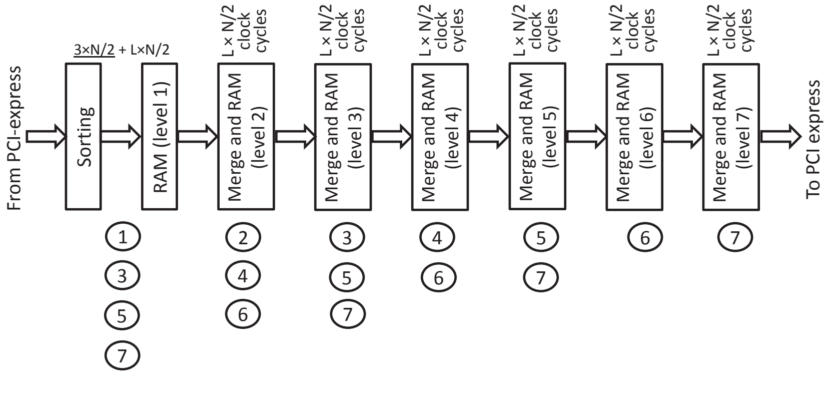

Fig. 9. Parallel operations in the proposed architecture.

complicated circuits do not give any advantage. This means that the resulting throughput cannot be increased. As soon as merging is completed, all data are copied to the second-level RAM and the first-level RAM may be refilled with new L sorted blocks.

Figure 9 explicitly indicates parallel operations. For example, merging at levels 3, 5, 7 is executed in parallel with data sorting. This method can be applied to the sorting of very large sets of data (tens and hundreds of millions of data items). In this case, the GPC (see Fig. 4) divides a very large set into subsets composed of $L \times N$ data items. The subsets are sorted in the pipelined structure shown in Fig. 9 and then merged in the software of the GPC. The experimental section below demonstrates that the data sorter implemented in Virtex-7 FPGA allows sorting data in hardware for $L = 128$ and $N = 512$. Thus, $512 \times 128 = 65536$ 32-bit data items (or 256 KB) are sorted and then 256 KB blocks can be merged in software. It will be shown in the experimental section that sorting in hardware (including data exchange with the GPC) is faster than similar sorting in software. Merging larger blocks permits the time of sorting in software to be considerably reduced.

3. COMMUNICATION-TIME DATA SORTERS

The actual performance of the designed circuits is often limited by the interfacing circuits that supply the initial data and return the results. Indeed, even for the most recent and most advanced on-chip interaction methods, such as those used in the Advanced eXtensible Interface (AXI), the communication overheads do not allow the theoretical throughput to be achieved in practical designs. The method and architecture described above permit only a small delay for data transmission in the beginning. When we sort large sets of data such delay is indeed negligible compared to the total delay. So, the proposed technique is very effective. In many practical cases we would like to sort small sets, such as those composed of N data items. For such a case the delay between the last received item and the final result of sorting becomes up to $N/2$ clock cycles and this may not be acceptable for many practical applications. We consider in this section such a method that enables the sorted results to be sequentially copied immediately after the last data item is received.

We describe below a parallel circuit that enables sorting to be entirely done within the time required for data transfers to and from the circuit; no additional time is required. Further, the design is very economical. The communication-time circuit, which is based on the network for discovering minimum and maximum values from [23], is shown in Fig. 10. It is composed of N M-bit registers R_0,..., R_{N-1}, and $N-1$ comparators/swappers.

At the initialization step, all the registers R_0,..., R_{N-1} are set to the minimum possible value for data items. For the sake of simplicity, this value is assumed to be 0. Any other value may also be chosen. Data items are received sequentially from interfacing circuits through the multiplexer Mux. The value x is set to 0, so all input

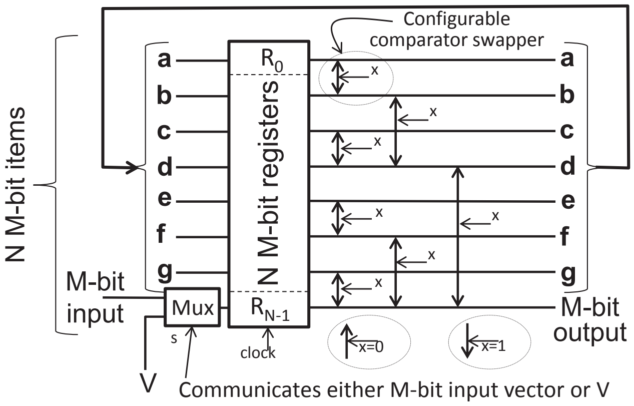

Fig. 10. Communication-time data accumulator/sorter.

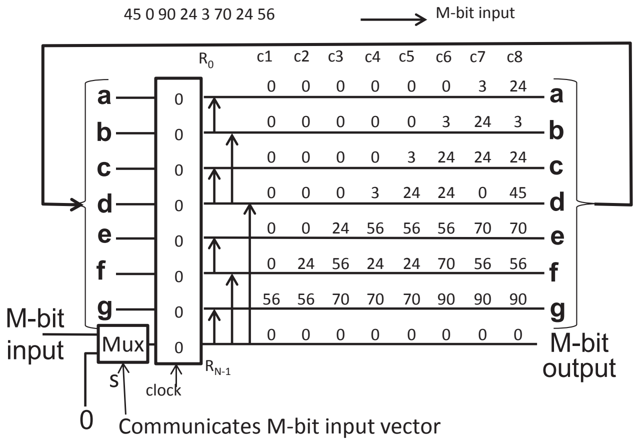

Fig. 11. An example of communication-time accumulation of input data items.

items will be moved up and accommodated somehow in the registers. Indeed, since the bottom line (marked as M-bit output) always contains the smallest value [23], any incoming item is either the smallest, or will be moved up. Figure 11 demonstrates how N M-bit items are accommodated, using an example with $N = 8$ items arriving in the following sequence: 1) 56; 2) 24; 3) 70; 4) 3; 5) 24; 6) 90; 7) 0; 8) 45.

Data may be received from a host system (such as ARM [24]) and accommodated in the registers R_0,..., R_{N-1} during communication time in N clock cycles indicated in Fig. 11 by symbols $c1, ..., c8(N = 8)$. As soon as N sorted data are received, the sorted result can be transferred immediately to the host system as shown in Fig. 12.

Now the multiplexer Mux communicates the maximum possible data value (m) to the register R_{N-1} and x is 0. Since $x = 0$, the maximum value will always be moved up at each clock cycle [23] enabling real-time transmission of the sorted items (through the M-bit output) in ascending order. To transmit the sorted items in descending order, it is necessary to set x to 1 and to replace the maximum possible value for data items (m) that is supplied to the multiplexer M with the minimum possible value.

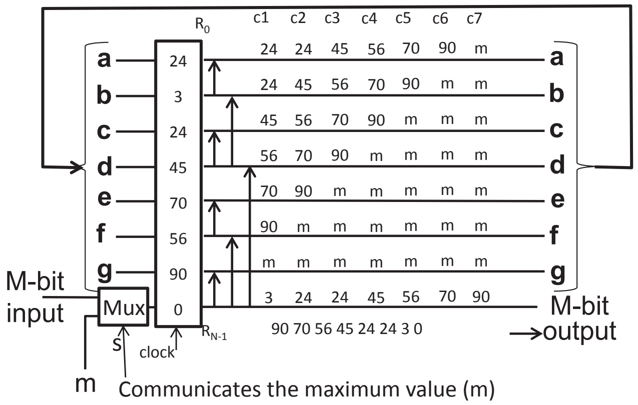

Fig. 12. An example of transmitting sorted data items.

The experiments have demonstrated that the circuit shown in Fig. 10 for $N = 512$, $M = 32$ can be built even for relatively small FPGAs, such as those available in the Nexys-4 prototyping board of Digilent. For advanced FPGAs, such as those from the Xilinx Virtex-7 family, the communication-time data sorter may be built for $N > 4096$. The results of experiments and comparisons will be given in the next section. Note once again that the communication-time circuits described above are advantageous for small autonomous sorters, which need the result to be produced immediately after the last item is received. In particular, they do not give any advantage for the methods and architectures described in Section 2. Therefore, the methods described in Section 2 are beneficial for sorting large data sets and the methods considered here are beneficial for sorting small data sets.

4. EXPERIMENTS AND COMPARISONS

The system for data transfers between a host PC and an FPGA has been designed, implemented, and tested. Experiments were done in the VC707 prototyping board [25] that contains Virtex-7 XC7VX485T FPGA from the Xilinx 7th series with PCI express endpoint connectivity “Gen1 8-lane (x8)”. All circuits were synthesized from the specification in VHDL and implemented in the Xilinx Vivado 2016.1 design suite. Software programs in the host PC run under the Linux operating system and they were developed in C language. The data were transferred from from the host PC to VC707 and back through the PCI express. The host PC is based on Intel core i7 3820 3.60 GHz.

The experiments were done in accordance with Fig. 4. The maximum size of data that are entirely sorted in the FPGA is 256 KB. For a larger size of data additional merging is done in the host PC. The results are presented in Fig. 13. It is clearly seen that the sorting throughput for the proposed systems is significantly better than in the host PC. For example, 1024 KB data can be sorted in the proposed system in 16 ms and in the host PC in 110 ms. The comparison of the time of sorting reported in the referenced papers and the results of Fig. 13 clearly shows that the proposed solutions are faster. Figure 14 demonstrates the organization of the experiments for communication-time data sorters (see Section 3).

Now autonomous circuits applicable to small data sets are synthesized, implemented, and tested. We have used a relatively low-cost Digilent Nexys-4 prototyping board with Xilinx Artix-7 FPGA xc7a100 [26]. N initial unsorted 32-bit data items ($M = 32$) are generated randomly and supplied to the communication-time data accumulator/sorter through the M-bit input (see Fig. 10). The clock frequency for data transfers was chosen to be 100 MHz (that is the default frequency of the on-board oscillator). An initial unsorted set of data is supplied and the sorted set is transmitted back entirely within $2 \times N$ clock cycles, which is just the time for data communication.

Table 1 displays the hardware resources that were used, as obtained from the Vivado post-implementation reports (including supplementary circuits, such as random number generation (RND)). Clearly, circuits for

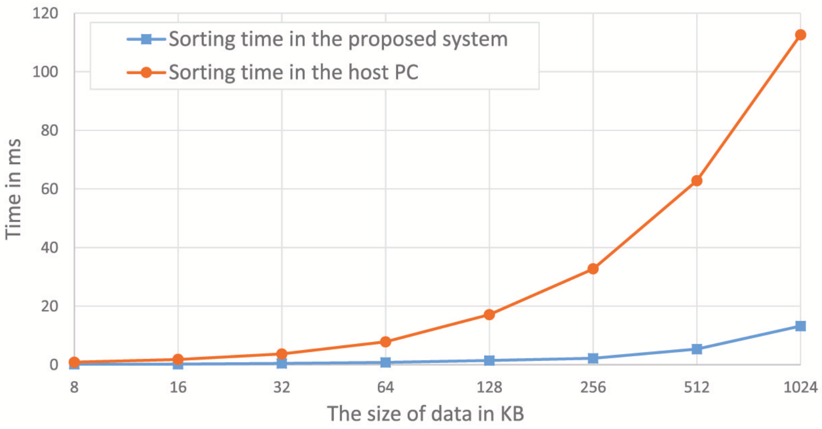

Fig. 13. An example of transmitting sorted data items.

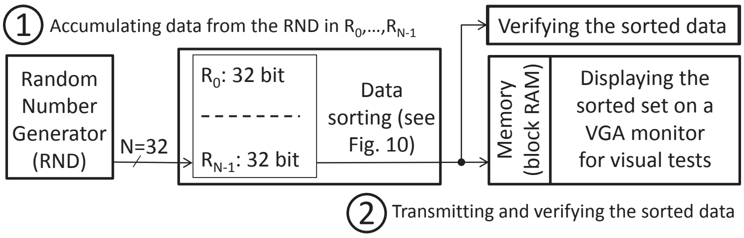
1 Accumulating data from the RND in R0,...,RN-1
Random Number Generator (RND)
N=32
R0: 32 bit
RN-1: 32 bit
Data sorting (see Fig. 10)
Verifying the sorted data
Memory (block RAM)
Displaying the sorted set on a VGA monitor for visual tests
2 Transmitting and verifying the sorted data

Fig. 14. Experimental setup.

Table 1. The hardware resources used for the Nexys-4 prototyping board

	$N=64$	$N=128$	$N=256$	$N=512$
Lookup tables	3760 (6%)	7448 (12%)	20699 (33%)	35645 (56%)
Flip-flops	2213 (2%)	4276 (3%)	8405 (7%)	16643 (13%)

significantly larger values of N than in the known even-odd merge and bitonic networks [19,21] have been built. The design proposed is also faster. Indeed, solving similar problems to those in Table 1 in networks [19,21] requires data to be copied to a long register that provides network inputs. The size S of this register, even for the smallest number of $N = 64$ in Table 1, is equal to $N \times N = 2048$ and if $N = 512$, then $S = 16384$ bits. Commercial FPGAs do not have such a large number of external pins and data items need to be copied sequentially and multiplexed to different sections of the register. Similarly, the sorted items must be segmented and transmitted back sequentially through the relevant interfacing circuits. If we consider on-chip communications (such as those available for Zynq all programmable systems-on-chip – APSoC [25]), we can see that the maximum number of high-performance AXI interfaces is 5 and the maximum number of bits transferred through each interface is 64. Thus, multiplexing is also necessary, which involves additional delays and resources. In the proposed design, the circuit itself receives and transmits data in parallel with sorting and no additional resources are required. The number of combinational levels in the proposed circuit is equal to $\lceil \log 2N \rceil$ and it is less than for the networks [19,21] where it is equal to $\lceil \log_2 N \rceil \times (\lceil \log_2 N \rceil - 1)$.

5. CONCLUSION

The paper proposes two architectures that are applicable to sorting large and small data sets. The distinctive feature of the first architecture is parallelization at several stages with the adjusted time. The first stage is data sorting, which is done in such a way that data acquisition, sorting, and transferring the sorted data are carried out at the same time. The second stage is a pipelined RAM-based merger that enables merging at different levels to be done in parallel and it can also be combined with the first stage. Such a type of processing is efficient for sorting large sets (tens and hundreds of millions of data items). The distinctive feature of the second architecture is communication-time processing, which permits sequential transfer of the results of sorting immediately after the last data items have been received. Such a type of processing is often needed for autonomous sorter operations over a relatively small number of data items (from hundreds to thousands of items). Thus, the proposed architectures complement each other. The experiments were done with an advanced prototyping system (allowing data processing in a general-purpose computer and in a recent FPGA from the Virtex-7 family of Xilinx) and with autonomous circuits implemented in a low-cost FPGA from the Artix-7 family of Xilinx. The results of experiments demonstrate significant acceleration compared to general-purpose software and the results reported in publications.

ACKNOWLEDGEMENTS

This research was supported by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research, the Study IT in Estonia Programme, Estonian Association of Information Technology, and Telecommunications and Portuguese National Funds through FCT – Foundation for Science and Technology, in the context of the project UID/CEC/00127/2013. The publication costs of this article were covered by the Estonian Academy of Sciences.

REFERENCES

1. Knuth, D. E. *The Art of Computer Programming. Sorting and Searching*, *Vol. III*. Addison-Wesley, 2011.
2. Sklyarov, V. and Skliarova, I. High-performance implementation of regular and easily scalable sorting networks on an FPGA. *Microprocess. Microsyst.*, 2014, **38**(5), 470–484.
3. Mueller, R., Teubner, J., and Alonso, G. Sorting networks on FPGAs. *Int. J. Very Large Data Bases*, 2012, **21**(1), 1–23.
4. Ortiz, J. and Andrews, D. A configurable high-throughput linear sorter system. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2010, 1–8.
5. Zuluaga, M., Milder, P., and Puschel, M. Computer generation of streaming sorting networks. In *Proceedings of the 49th Design Automation Conference*. ACM, New York, 2012, 1245–1253.
6. Singh, S. and Greaves, D. J. Kiwi: synthesis of FPGA circuits from parallel programs. In *Proceedings of the 16th IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2008, 3–12.
7. Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the 2008 Symposium on Application Specific Processors*. IEEE, 2008, 101–107.
8. Chamberlain, R. D. and Ganesan, N. Sorting on architecturally diverse computer systems. In *Proceedings of the 3rd International Workshop on High-Performance Reconfigurable Computing Technology and Applications*. ACM, New York, 2009, 39–46.
9. Mueller, R. *Data Stream Processing on Embedded Devices*. Ph.D. thesis, ETH, Zurich, 2010.
10. Koch, D. and Torresen, J. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, 2011, 45–54.
11. Sklyarov, V., Skliarova, I., Mihhailov, D., and Sudnitson, A. Implementation in FPGA of address-based data sorting. In *Proceedings of the 21st International Conference on Field-Programmable Logic and Applications*. IEEE, 2011, 405–410.
12. Kipfer, P. and Westermann, R. GPU Gems 2, Improved GPU Sorting. `http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter46.html`, 2005 (accessed 08.06.2016).
13. Gapannini, G., Silvestri, F., and Baraglia, R. Sorting on GPU for large scale datasets: a thorough comparison. *Inf. Process. Manage*, 2012, **48**(5), 903–917.
14. Ye, X., Fan, D., Lin, W., Yuan, N., and Ienne, P. High performance comparison-based sorting algorithm on many-core GPUs. In *Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2010, 1–10.
15. Satish, N., Harris, M., and Garland, M. Designing efficient sorting algorithms for manycore GPUs. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, 1–10.
16. Cederman, D. and Tsigas, P. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th Annual European Symposium on Algorithms*. Springer-Verlag, Berlin, Heidelberg, 2008, 246–258.
17. Grozea, C., Bankovic, Z., and Laskov, P. FPGA vs. multi-core CPUs vs. GPUs: hands-on experience with a sorting application. In *Facing the Multicore-Challenge* (Keller, R., Kramer, D., and Weiss, J. P., eds). Springer-Verlag, Berlin, Heidelberg, 2010, 105–117.
18. Edahiro, M. Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA acceleration. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. IEEE, 2009, 230–233.
19. Aj-Haj Baddar, S. W. and Batcher, K. E. *Designing Sorting Networks. A New Paradigm*. Springer, 2011.
20. Marcelino, R., Neto, H. C., and Cardoso, J. M. P. A comparison of three representative hardware sorting units. In *Proceedings of the 35th Annual IEEE Conference on Industrial Electronics*. IEEE, 2009, 2805–2810.
21. Batcher, K. E. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*. ACM, New York, 1968, 307–314.
22. Lacey, S. and Box, R. A fast, easy sort: a novel enhancement makes a bubble sort into one of the fastest sorting routines. *Byte*, 1991, **16**(4), 315–320.
23. Sklyarov, V. and Skliarova, I. Fast regular circuits for network-based parallel data processing. *Adv. Electr. Comput. Eng.*, 2013, **13**(4), 47–50.
24. Xilinx, Inc. *Zynq-7000 all programmable SoC. Technical Reference Manual*. `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, 2016 (accessed 01.02.2017).

25. Xilinx, Inc. *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*. `https://www.xilinx.com/support/documentation/boards_and_kits/vc707/ug885_VC707_Eval_Bd.pdf`, 2016 (accessed 01.02.2017).
26. Digilent, Inc. *Nexys4 DDR FPGA Board Reference Manual*. `https://reference.digilentinc.com/_media/nexys4-ddr:nexys4ddr_rm.pdf`, 2016 (accessed 08.06.2016).

Kiired iteratiivsed ahelad ja RAM-i baasil ühendajad, kiirendamaks andmete sortimist riist- ning tarkvara süsteemides

Valery Sklyarov, Iouliia Skliarova, Artjom Rjabov ja Alexander Sudnitson

On välja pakutud ja kirjeldatud kaks arhitektuuri paralleelsete andmete sortimiseks. Esimene on mõeldud suuremahuliste andmekogude jaoks, ühendades kolm andmete töötlemise astet: andmete sortimine riistvaras (FPGA-s), eelsorditud andmete ühendamine riistvaras (FPGA-s) ja seejärel nende suurte alamhulkade üldotstarbeline ühendamine tarkvara abil. Andmete vahetamine FPGA ja üldotstarbelise arvuti vahel toimub läbi PCI ekspress-siini. Teine arhitektuur on rakendatav väiksemate andmekogude puhul, võimaldades sortimist andmete samaaegse vastuvõtuga, st kui viimane andmete osa on käes, võib sorditud osad kohe edasi saata. Võrreldes erinevate varem avaldatud tulemustega, kus on kasutatud tarkvaralisi lahendusi, näitavad katsetulemused pakutud arhitektuuride eeliseid, mis lubavad vähendada vajaminevaid riistvararessursse ja suurendada tootlikkust.