High-Level Descriptions of Hardware Accelerators for Sorting Algorithms

Mauricio Fayula Gonzalez, Alexander Petrovich Antonov
Peter the Great St. Petersburg Polytechnic University
St. Petersburg, Russia
fayula.gm@edu.spbstu.ru, antonov_ap@spbstu.ru

***Abstract*—Sorting algorithms are among the most employed in computer applications. As increasingly large amounts of information need to be processed efficiently, the need for effective methods to sort data remains a relevant problem in applications involving databases, digital signal processing and many others. Hardware accelerators are particularly suitable for processing information with high degree of parallelism. In this article, we present several approaches for sorting data by means of reconfigurable hardware, showing its advantage over microprocessor based solutions. Furthermore, the design of a hardware system written in a high-level programming language is confronted with a traditional description with System Verilog. This article also proposes the restructuring of the code describing the sorting network in HLS and several optimization techniques allowing to achieve higher performances.**

I. Introduction

Finding efficient ways to sort large sets of information is currently one of the main problems in big data applications and others where large databases take part.

The goal of a sorting algorithm is to reorganize a random input set of elements into a monotonically increasing or decreasing sequence. A classical sorting algorithm is merge-sort. The working principle is simple: an input sequence of elements is divided into two halves to be ordered independently. Each of those halves are subsequently divided in two parts, recursively, until each subsequence has an elementary size, typically, of two elements. In this case, the sorting becomes trivial and is solved by means of a simple comparison, and possible swap in the order of the elements. At the return of the recursive function, each couple of left and right parts is then merged in order, by comparing the elements of both subsequences. The final result is the full sequence completely sorted. In terms of performance, sequential sorting algorithms can't surpass the barrier of *O(n)*, while in practice the best achievable case for several such algorithms is

$$O(n \cdot logn) \quad (1)$$

where *n* is a key parameter of the algorithm, typically the size.

While traditional sorting algorithms are aimed at sequential processing, sorting networks [1][11] allow the concurrent comparison of different pairs of elements within the given input data. Therefore, parallel computing systems particularly benefit from sorting networks when ordering sequences of data.

Traditional microprocessor-based computing systems are limited in terms of the number of threads that may execute in parallel, among others. Moreover, the repeated execution of simple operations such as the compare and conditional swap in sorting algorithms, doesn't make full use of their complex architecture. For this reason, a more suitable platform is found in GPGPUs. A significantly larger number of cores may execute simpler instructions with a higher degree of parallelism. However, this comes with a considerable increase in terms of power consumption.

Fine-grained reconfigurable devices such as FPGAs provide a customizable platform that may execute very complex functions in few clock cycles, such as integrated circuits do. But they may also be configured in such a way that a basic operation is solved by a simple processing element, which may be replicated many times across the device, achieving unique massive degrees of parallelism. Such computing potential is often achieved with a fraction of the power consumption required by other processing systems such as microprocessors or GPUs.

These characteristics make reconfigurable devices ideal for implementing massively parallel algorithms, such as sorting networks of large amounts of data.

Moreover, while traditional hardware implementations were described by means of Hardware Description Languages (HDL) such as VHDL or Verilog, modern tools such as High-Level Synthesis (HLS) allow to describe hardware systems employing the syntax of high-level programming languages such as C and pragmas in the style of parallel programming paradigms.

In this work, several sorting networks are described, both by means of HDL and HLS, showing the significant performance improvement of the implemented systems when compared with traditional software running in parallel on microprocessors, with a design methodology that resembles that of software development. The advantages and limitations of HLS are confronted with traditional descriptions in HDL for the systems under study. Moreover, a convenient restructuring of the sorting network and several manual optimizations are proposed, allowing HLS to generate the corresponding

hardware more efficiently. In this article, Section II introduces the algorithms used and related work on the topic, Section III describes the technologies employed, Section IV presents different designed systems, Section V shows the main results obtained, and relevant conclusions are summarized in Section VI.

II. Background and previous work

Many sequential sorting algorithms achieve worst case results within (1). One of such algorithms is merge-sort, which is the base of several sorting networks. A pseudo-code of merge-sort is described in Algorithm 1.

Algorithm 1 Merge_sort(*data*,*left*,*right*)

```
0:  if (left<right) begin
1:    center = (left+right)/2
2:    Merge_sort(data,left,center)
3:    center = center+1
4:    Merge_sort(data,center,right)
5:    while (left<right) begin
6:      if(data[left]>data[center]) begin
7:        Swap(data,left,center)
8:        center = center+1
9:      end
10:     left = left+1
11:   end
12: end
```

The working principle of the recursive algorithm consists on subdividing the input data array in halves repeatedly until the most elementary size of two elements; then the calls to the recursive functions will return both halves merged and sorted. In the elementary case, the merging process consists in comparing two values and possibly exchanging them so that the lesser one remains on the left. Merging larger sequences of sorted elements will require iterating through the elements of both halves, each time comparing and possibly swapping if not in order.

Sorting networks [1][11] were proposed after the emergence of the first parallel computing systems as an efficient method for interconnecting their building components, such as microprocessors or memories. A distinctive characteristic of the sorting networks and algorithms is the large number of repetitions of the same operation between different pairs of the input data. This operation is the elementary comparison between two elements, which allows to determine the order between them, and the conditional swap, as seen in Fig. 1. The simplicity of this operation and its repetitive execution makes sorting networks specially suitable for massively parallel architectures, such as fine-grained reconfigurable devices (FPGAs).

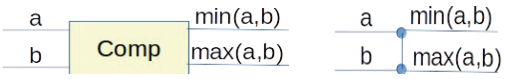

Fig. 1. A comparator and conditional swapper, the elementary operation in sorting networks.

Classical sorting networks [1] such as odd-even and bitonic networks are parallel versions of the merge-sort algorithm. Their working principle is based on merging two previously ordered sequences, so that the whole network is composed of an initial sorter for each of the two input sequences followed by a merger.

Bubblesort [9] is yet a classical sequential sorting algorithm, probably the most trivial of them all. Its working principle is based on repeating the compare-swap operation exclusively among all neighboring elements, repeating this process as many times as elements constitute the input sequence, minus one. In practice, this implies slowly bringing lower values to the front of the sequence, with such displacement taking place only to neighbor positions at each loop iteration, while taking larger ones to the back. Algorithm 2. describes its behavior.

Algorithm 2 Bubblesort(*data*)

```
0:  iterator = 1
1:  while (iterator<size(data)) begin
2:    index = 1
3:    while (index<size(data)-iterator) begin
4:      if(data[index]>data[index+1])
5:        Swap(data,index,index+1)
6:      index = index+1
7:    end
8:    iterator = iterator+1
9:  end
```

Despite its worst case performance is certainly far from ideal at $O(n^2)$, it may achieve optimal completion times of $O(n)$ under certain circumstances, thus outperforming merge-sort in those particular cases. In particular, one property of this simple algorithm is that its execution may be halted as soon as one iteration doesn't find any unordered pair of elements, implying the whole sequence is sorted. Thus, making use of this feature, lineal complexity is achieved for mostly ordered sequences.

Previous works [9] have compared the efficiency of different sorting algorithms implemented on FPGA in terms of execution time, requirements of logical resources and power consumption, pointing to bubble sort as a particularly fast case.

As with merge-sort, a sorting network counterpart of the bubble sort algorithm exists, being named odd-even transposition network [7]. A drawback of this network is its relative large number of processing elements, as shown in expression (2). However, its regular structure may result in a beneficial feature both due to its relative design simplicity and to enabling more advanced features such as folding and reproducibility.

$$S = \frac{N^2 - N}{2} \quad (2)$$

where N is the size of the input sequence.

Several authors have confirmed that massively parallel computing systems benefit from algorithms that naturally allow higher levels of parallelism, such as sorting networks. Several sorting algorithms have been implemented on FPGA and compared [2]. The two sorting networks studied, odd-even and

bitonic sorters, clearly outperformed other traditional algorithms. Typical performance improvements ranged from 2x to 4x, while the area for both networks remained at around a half when compared with other implementations.

In [3], different variants of the merge-sort algorithm were designed for FPGAs, including serial and parallel versions of the traditional merge-sort, the odd-even and bitonic networks and a customized sorting network, based on a combination of both. Compared with the serial approach, the authors noted a modest speedup of around 1.4x for the traditional parallel version, but over 30x for all sorting networks. Furthermore, while the area requirements of both the serial and parallel approaches of the traditional algorithm would remain similarly high, the sorting networks studied resulted in a fraction of resources consumption.

The works mentioned above were aimed at processing small data sequences. The use of a folded Clos network as a streaming permutation network has also been studied [4], allowing larger datasets. Two study cases were presented. A high performance design would allow reaching close to optimal execution time, at a cost of more logical resources. A more resource efficient variant is also studied, leading to a significant increase in computation delay.

Hybrid designs have also been considered [5]. The traditional merge-sort algorithm is constrained in terms of achievable parallelism by its last stages. For this reason, the use of a mixed algorithm, computing the first stages of the traditional merge-sort algorithm in parallel, while substituting the last ones with bitonic sorters has also been proposed. The performance attained was higher than previous works [4].

Other authors have also confronted the aforementioned bottleneck in the last stages of the traditional merge-sorter algorithm and employed custom networks [6], achieving higher performances in this way.

Hybrid sorting architectures have also been applied to the case when only a subset of the largest or smallest elements are of interest [10].

Some articles have studied the use of HLS for describing and implementing sorting algorithms, including sorting networks. Certain modifications, including the use of hybrid algorithms, have been put forward. Apart from the comparative study of different sorting networks implemented in hardware using either HDLs or high-level programming languages, this article proposes the restructuring of the code in HLS, together with several manual optimizations for achieving a higher performance through such automatization environment, after conveniently applying suitable directives.

III. Technology and Methods

A. *Sorting networks*

Sorting networks [1] are represented as a fixed set of nodes conveniently interconnected in order to guarantee that a given randomly unordered sequence is processed and transformed such that the elements are output monotonically increasing or decreasing.

The key element is the computing node, which performs a simple comparison between two data and passes them to the output if they were ordered according to the given criteria, or reorders (swaps) them otherwise.

A bitonic network has as main distinguishing characteristic that at the input of the merger is provided a bitonic sequence, defined as a succession of elements monotonically increasing first and monotonically decreasing later. Fig. 2 shows an elementary bitonic sorter of a sequence of four elements.

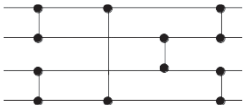

Fig. 2. A bitonic sorter of four elements.

As with odd-even mergers, bitonic sorters of larger sizes are also composed mostly of smaller sorters. For instance, Fig. 3 shows how a bitonic sorter of eight elements uses two four-elements sorters and an initial halving structure.

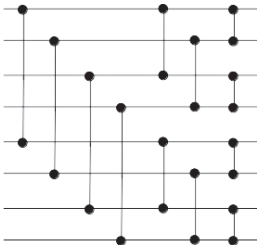

Fig. 3. A bitonic merger of eight elements.

Odd-even transposition networks present the most regular structure, at the expense of a larger number of PEs. Fig. 4 depicts an eight element odd-even transposition network.

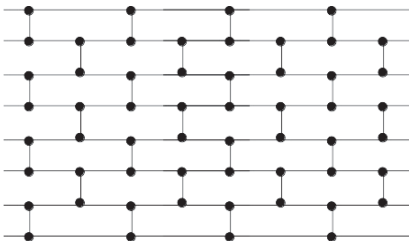

Fig. 4. Odd even transposition network

B. *High-Level Synthesis*

Typically, hardware realizations are designed using a hardware description language (HDL), such as VHDL or Verilog. Despite their syntax may seem similar, HDLs strongly differ from programming languages in their semantics, since their goals are completely different. While programming languages describe sequences of instructions that usually are to be executed by a microprocessor or microcontroller, HDLs describe the structure of digital circuits.

One key difference among both kinds of languages is the sequential nature of the code aimed at being run by a processor, even when several cores may work in parallel, while the statements described in HDL represent circuits that will be implemented by different resources of the

reconfigurable device and compute the corresponding operations in a parallel fashion.

This characteristic may cause HDLs to seem less intuitive. On the other hand, the standardization of computing systems based on microprocessors implies the popularization of the use of programming languages, becoming a natural tool to describe the work to be developed by a computer, in detriment of HDLs.

With this in mind, several approaches have been sought in order to increase the productivity when obtaining hardware implementations.

The application of high-level programming languages to the design flow of reconfigurable hardware has long been studied in the last decades. As a result, High-Level Synthesis (HLS) offers a way to describe parallel hardware systems in a high-level language such as C with the addition of pragmas similar to standard parallel programming paradigms.

IV. Architecture

A. Odd-even transposition network

Two realizations are proposed:

1) HDL: The system has been described by means of a hardware description language. All descriptions are parametrized in order to adapt to the size of the problem. The basic processing element (PE) is the comparator and conditional swapper of two input data, being repeated throughout the structure of the network.

Within the classical design flow based on HDL, three approaches have been studied: fully combinational, fully sequential and the hybrid from both of them, with a significant and also parametrized amount of logic between sequential elements.

The fully combinational case represents the fastest structure to obtain an ordered sequence of elements, since the time to complete the task is determined by the delays inherent to the basic logical blocks existing in the reconfigurable device.

The elementary operation of the sorting network is logically represented by a comparator, while a couple of multiplexers model the possible swap between every couple of input elements. This basic processing element is repeated multiple times, following a pattern that resembles the interconnection among nodes depicted in Fig. 4.

A key characteristic of fully combinational designs is the high degree of parallelism achieved, and the minimum delay for providing the results at the output, at the cost of large amounts of logical elements.

A fully sequential design overcomes these limitations by reusing iteratively the same physical elementary components by different pairs of input elements across the sorting network, thus minimizing the logical area required at the cost of a slower output of results.

Therefore, a balanced solution is expected to provide the optimal results in terms of area constraints derived from the target device and high performance when providing the ordered sequences at the outputd.

2) HLS: The hardware implementation of the algorithm was also described in the high-level language HLS. The structure of the code mimics the HDL description, similarly employing constants instead of parameters in order to adapt the resulting system to the given size of the problem.

One approach to modeling an odd-even transposition network with HLS consists on describing each of the odd and even stages by means of separate loops nested within an external loop, being all of them parametrized according to the size of the problem.

B. Bitonic sorter

A bitonic sorter of a given size results from the assembly of bitonic mergers of lesser sizes. Fig. 5 shows a bitonic sorter for a sequence of eight elements, composed of a merger of eight elements, which receives two bitonic sequences of 4 elements through corresponding bitonic mergers of 4 elements, themselves being provided bitonic sequences of 2 elements by pairs of 2-elements mergers.

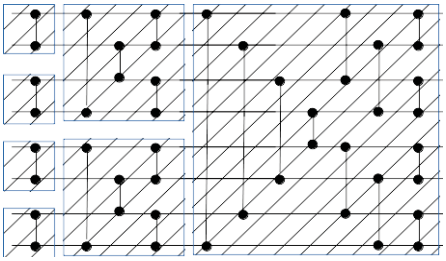

Fig. 5. An 8-elements bitonic sorting network, illustrating each of the bitonic mergers providing bitonic subsequences.

1) HDL: According to the behavior of a bitonic sorter, the system is divided into several levels, each of which is subdivided into sublevels, where PEs are conveniently interconnected according to the position it occupies within the network, which is determined by the parameters of the module.

The system is designed so that the resulting distribution of PEs across the network matches the interconnections of the bitonic sorter shown in Fig. 6. The design flow included the simulation of a model of the system, allowing to visualize the behavior and result of the sorting algorithm.

2) HLS: In this case, a series of nested loops describe the different levels and sublevels mentioned above. However, despite the natural way to describe such system implies the assignment of a loop to each of the mergers depicted in Fig. 6, such an approach doesn't allow to properly exploit the available parallelism, since mergers of the same range could perform their computations in parallel, as long as there are enough logical resources available in the device, without incurring in stalls due to data dependencies between them.

For this reason, we propose adapting the code so that all mergers of the same range, placed vertically in the example of Fig. 6, are joined into single loops, so that the HLS synthesis tool can make use of the available parallelism.

Furthermore, we clearly delimit *horizontally* (as per Fig. 6) the different levels that carry dependencies among them, even within each merger, so that the assignment of separate hardware resources to different levels doesn't bring a higher degree of parallelism, enabling the sharing of hardware resources among different levels, perhaps within the same merger.

Fig. 6 illustrates the proposed rearrangement of the code for the 8-elements bitonic sorter. In this way, all comparisons within each level may be executed in parallel, whereas different levels always carry dependencies among them, preventing their simultaneous computation.

Based on this structure, a code based on a first intuitive approach, assigning a loop to each of the arrangements of the comparators according to mergers of different level, as per Fig. 5, is substituted by an optimized code that allows HLS to exploit the inherent parallelism.

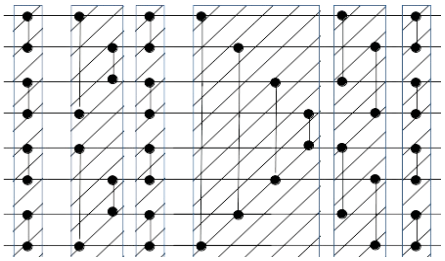

Fig. 6. The conceptual rearrangement of the bitonic sorter.

Moreover, the recursive nature of a bitonic sorter, constructed as the interconnection of bitonic mergers of lower orders, leads naturally to coding a hierarchy of nested loops that may result too complex for HLS to provide an optimized hardware, as found in the literature [8]. Algorithm 3 represents the standard structure of the code.

Algorithm 3 Bitonic_sorter(*data*,*left*,*right*)

```
0:  for (...) begin
1:    ...
2:    for (…) begin
3:      for (…) begin
4:        ...
5:      end
6:    end
7:    for (…) begin
8:      for (…) begin
9:        ...
10:     end
11:   end
12: end
```

In this way, a pair of nested loops are embedded inside a general outer loop, which iterates through the bitonic mergers of different magnitude. Each of the pair of nested loops is typically employed to construct the two different structures of comparators that compose a bitonic merger, as represented, for instance in the right part of Fig. 5. For the nested loops, the outer one would iteratively shorten the distance between comparators, while the inner one is used to replicate vertically the bitonic mergers of lower order, as in the left-hand side of Fig. 5. However, this approach prevents the automatic optimization by HLS for parametrized sorters of sequences of random size, given the variable bounds of the loops.

In this work, the nested loops have been joined into a single loop, which allows HLS to process a regular structure.

Furthermore, the modulo operation is very handy in order to process the indexes of the comparators when replicating bitonic mergers. However, this operator is usually very costly in terms of hardware resources, with HLS assigning a *urem* unit, which takes a disproportionate amount of clock cycles to complete, when compared to other operations.

In order to solve this, another manual optimization included in the high-level description of the sorting network has been the substitution of modulo operators by bitwise operators, which are processed very efficiently in hardware and by HLS.

Given all the manual optimizations of the code, the HLS synthesis tool is then instructed to apply pipelining and suitable unroll and array partition factors, achieving the highest performance.

Finally, given the high-level code describing the parametrized system, the process of synthesis is automatized by means of a tcl script, allowing an easy reproduction of experiments with varying parameters.

V. Results

The design environment employed in this work was Xilinx Vivado and Vitis HLS version 2021.2. The study cases were targeted at a Xilinx Alveo U50 device. According to the considerations mentioned above, it was studied the consumption of logical resources for each sorting network by each of the three approaches based on HDL descriptions (fully combinational, fully sequential and hybrid) and by the HLS system. Fig. 7 shows the increase of logical area as a function of the size of the problem for the odd-even transposition network.

This study was aimed at relatively small problems, of up to 256 elements. As expected, the fully combinational approach results the most resource consuming in terms of logical elements. Moreover, as the data shows, the system remains scalable within the problem sizes studied.

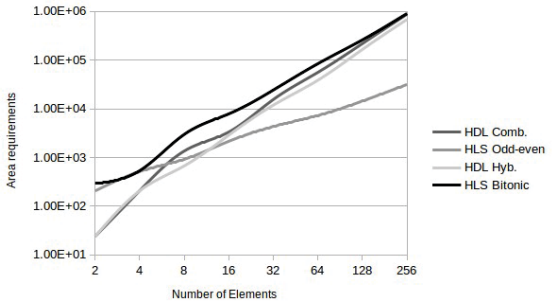

Fig. 7. N*umber of look-up tables required by different implementations of the odd-even transposition network, as a function of the size of the problem.*

For the same set of problems, a comparison of performance, or time to complete the algorithm, is shown in

Fig. 8. The three approaches, fully combinational and hybrid in HDL, and the HLS description, are compared. It can be noticed that, as expected, a fully combinational approach suits problems of small sizes and obtains the ordered sequence of elements in the minimum possible time. An hybrid design, despite slower, may require less logical resources.

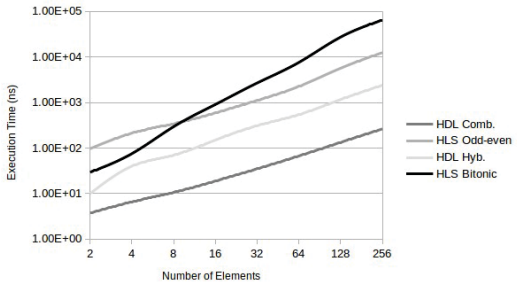

Fig. 8. The execution time of the odd-even transposition network, as a function of the size of the problem, for HDL fully combinational and hybrid designs, and HLS description.

Regarding the implementations in HLS, the odd-even transposition network remains within the same order of magnitude in terms of execution time, when compared with the hybrid approach in HDL. The bitonic sorter designed achieved optimal execution times for small data sizes. Furthermore, the systems remain scalable for the size of the problems studied.

VI. Conclusions

Several sorting network algorithms have been implemented in hardware, being presented in this work. In particular, the goal was to study the application of high level languages to the design of hardware realizations providing a solution to currently demanded problems, as sorting of large datasets or digital signal processing.

For this purpose, implementations in System Verilog and HLS for each algorithm are subjects of study. Within the HDL-based designs, several approaches have been studied: fully combinational, fully sequential and a balanced and parametrized combination of both.

It has been proven the suitability of reconfigurable devices as a platform for the implementation of sorting networks. It was shown that implementations of accelerators on reconfigurable hardware allow a significant improvement in the performance of sorting algorithms, in terms of execution time, when compared with parallel software solutions running on multicore microprocessors.

Regarding the methodology applied to the design of hardware systems, it has been studied the suitability of one high-level programming language for implementations on reconfigurable devices: HLS.

Hardware accelerators have been described both in a traditional hardware description language, System Verilog, and in HLS. The HLS approach proved to achieve a performance close to that of the traditional design flow based on System Verilog.

This work also proposes a novel structuring of the algorithm that suits the particularities of reconfigurable devices and the HLS synthesis tool, achieving higher performance. Several manual optimization techniques are also presented.

In terms of efficiency and productivity, it's noted that the effort needed to describe the different systems using high level languages and fulfill the whole design flow may be significantly lower than the traditional HDL based implementation. The set of tools that accompany HLS enable early and easy debugging of the system, while the design environment resemble traditional software development environments.

Nevertheless, it must be noted that the effective use of the HLS set of tools requires the learning of the distinctive characteristics of this particular development environment. Furthermore, a thorough knowledge of the platform that will implement the different algorithms is highly advisable, in order to make an efficient use of the resources available and achieve best performances.

References

[1] Batcher, K. E. (1968). Sorting networks and their applications. Proceedings of the Spring Joint Computer Conference, 307. https://doi.org/10.1145/1468075.1468121

[2] Abdelrasoul, M., Shaban, A. S., & Abdel-Kader, H. (2021). FPGA based hardware accelerator for sorting data. Proceedings of the 9th International Japan-Africa Conference on Electronics, Communications, and Computations (JAC-ECC), 57–60. https://doi.org/10.1109/jac-ecc54461.2021.9691432.

[3] Lobo, J., & Kuwelkar, S. (2020). Performance analysis of merge sort algorithms. 2020 International Conference on Electronics and Sustainable Communication Systems (ICESC), 110–115. https://doi.org/10.1109/icesc48915.2020.9155623.

[4] Chen, R., Siriyal, S., & Prasanna, V. (2015). Energy and memory efficient mapping of bitonic sorting on FPGA. Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 240–249. https://doi.org/10.1145/2684746.2689068.

[5] Srivastava, A., Chen, R., Prasanna, V. K., & Chelmis, C. (2015). A hybrid design for high performance large-scale sorting on FPGA. Proceedings of the 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 1–6. https://doi.org/10.1109/reconfig.2015.7393322.

[6] Mashimo, S., Van Chu, T., & Kise, K. (2017). High-Performance Hardware Merge Sorter. IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), 1–8. https://doi.org/10.1109/fccm.2017.19.

[7] D.A. Dyabilkin, *Sorting Networks*. Kazan: University of Kazan, 2015.

[8] Bozhko, A. B., Antonov, A. P. (2025) Research and development of reconfigurable hardware implementations of sorting networks on FPGA. Proceedings of the Week of the Science of the Institute of Computer Science and Cybersecurity, 228-231.

[9] Anuradha, P., Renuka, G., V.Gurumurthy, Rajkumar, K., Navitha, C., & Krishna, D. S. (2025). Implementation of Sorting Algorithms Using FPGA. International Conference on Recent Innovation in Science Engineering and Technology (ICRISET), 1–5. https://doi.org/10.1109/icriset64803.2025.11251606.

[10] Reddy, B. N. K., K, S., Dandeliya, S., Naidu, S. P. S., & P, N. K. (2023). Accelerating Sorting Performance on FPGA: Combining Quick Sort and Heap Sort through Hybrid Pipelining. IEEE International Symposium on Smart Electronic Systems (iSES), 405–408. https://doi.org/10.1109/ises58672.2023.00090.

[11] Baddar, S. W. A., & Batcher, K. E. (2011). Designing Sorting Networks: a new paradigm. https://openlibrary.org/books/OL27996547M/Designing_Sorting_Networks.