Journal of
Low Power Electronics and Applications

MDPI

Article

Analysis and Comparison of Different Approaches to Implementing a Network-Based Parallel Data Processing Algorithm

Iouliia Skliarova

Institute of Electronics and Informatics Engineering of Aveiro (IEETA), Department of Electronics, Telecommunications and Informatics, University of Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal; iouliia@ua.pt

Abstract: It is well known that network-based parallel data processing algorithms are well suited to implementation in reconfigurable hardware recurring to either Field-Programmable Gate Arrays (FPGA) or Programmable Systems-on-Chip (PSoC). The intrinsic parallelism of these devices makes it possible to execute several data-independent network operations in parallel. However, the approaches to designing the respective systems vary significantly with the experience and background of the engineer in charge. In this paper, we analyze and compare the pros and cons of using an embedded processor, high-level synthesis methods, and register-transfer low-level design in terms of design effort, performance, and power consumption for implementing a parallel algorithm to find the two smallest values in a dataset. This problem is easy to formulate, has a number of practical applications (for instance, in low-density parity check decoders), and is very well suited to parallel implementation based on comparator networks.

Keywords: data processing; parallel algorithm; hardware accelerator; high-level synthesis; embedded processor; two smallest values in a dataset

Citation: Skliarova, I. Analysis and Comparison of Different Approaches to Implementing a Network-Based Parallel Data Processing Algorithm. *J. Low Power Electron. Appl.* **2022**, *12*, 38. https://doi.org/10.3390/jlpea12030038

Academic Editors: Xinfei Guo and Andrea Acquaviva

Received: 22 May 2022
Accepted: 6 July 2022
Published: 9 July 2022

1. Introduction

Many data processing algorithms recur to parallel networks, in which several data items are handled concurrently by independent processing units organized in a network level. The input data are fed to the first network level, processed there in parallel, and then propagate through the remaining network levels until the result(s) can be read from the last level output(s). The number of network levels is known as the network's depth, which is directly related to the network's latency. The processing units employed in such networks are usually simple combinational blocks like comparators and half-adders. Therefore, the higher the latency, the lower the network performance (throughput). This is one of the reasons why designers try to increase the throughput by inserting pipeline registers between the network's levels so that different sets of data can be processed in the network at the same time.

Some examples of parallel data processing networks are sorting networks [1], searching networks [1], and counting networks [2]. It has been shown by various studies that parallel data networks are well suited for implementation in reconfigurable hardware, such as Field-Programmable Gate Arrays (FPGA) and Programmable Systems-on-Chip (PSoC) [3–14]. This is because many processing elements can easily be instantiated, synthesized, and implemented according to the required network structure, and modern FPGAs contain plenty of distributed storage elements that can be used for effective pipelining. The principal characteristics, benefits, and limitations of the different approaches to implementing network-based hardware accelerators in FPGA and PSoC are reviewed in [15]. As indicated in [15], the majority of the analyzed implementations recur to low-level hardware designs (usually in VHDL/Verilog or in a specially developed language whose specifications are later translated to standard HDL (Hardware Description Language) RTL

(Register-Transfer Level) descriptions). None of the respective authors realized any study or comparison of using different specification methods (ranging from high-level descriptions to low-level code) when implementing a particular data processing network.

In this work, we study one data network type dealing with the problem of finding the two smallest values in a dataset. Three design specification methods, based on using an embedded processor soft core, low-level VHDL RTL design, and High-Level Synthesis (HLS) from C++ specification, are discussed, analyzed, and compared, and their pros and cons are identified. We believe that the problems pinpointed and the conclusions drawn would be beneficial for future designers, suggesting new parallel data processing architectures in reconfigurable hardware.

The remainder of this paper is organized as follows. Section 2 discusses the considered problem and the respective algorithms, which are suitable for parallel implementation in hardware. Section 3 exploits the three considered specification methods to implement the selected parallel algorithm in FPGA. Section 4 demonstrates the results of experiments and comparisons. Finally, conclusions are given in Section 5.

2. Algorithms for Finding Two Smallest Values

Sorting data and finding a maximum/minimum value within a dataset is a very common task in many data processing algorithms [1,16–18]. The problem of discovering two smallest (or greatest) values looks similar to but is more complex than discovering only the minimum (or maximum). This task, besides being very didactical and therefore actively explored in traditional programming courses, has a number of practical applications. For instance, low-density parity-check (LDPC) decoders may efficiently be implemented using a min-sum scheme, whose operation can be improved by producing only the two smallest values within a check-node function unit [13]. Likewise, a number of largest-weight feature words must be extracted for Web page clustering, such as in [19].

The problem of finding the two smallest values can be solved by firstly sorting the data in ascending order and then outputting the first two values. However, this is an unnecessarily complex approach; therefore, more efficient solutions have been proposed in the literature [1,13,14]. They are mainly based on networks of comparators [1], networks of comparators with memory [13], and bit searching [14]. These solutions are good candidates for implementation in reconfigurable hardware (particularly in FPGA) since a number of the operations that are involved may be executed in parallel, increasing the throughput and reducing the latency of the respective circuits. Graphics processing units are also good platforms that offer valuable support for parallel algorithms [20] but use significantly more power than FPGA. Besides the GPUs' control overhead due to the Single Instruction, Multiple Threads (SIMT) execution model cannot compete with FPGAs fully custom datapaths.

Parallel comparator networks have long been used to find the smallest value (min_1st) in a data set. Such networks are especially suitable for parallel implementations in reconfigurable hardware [10]. To process $N = 2^L$ values, L lines (levels) of comparators are required, as depicted in Figure 1a for N = 8 and integer values. The comparators are represented as pairs of two dots connected by a short vertical line using the widely known Knuth notation ( ) and function so that if a data item in the upper input is smaller than the data item in the bottom input, the input values are swapped (see Figure 1c). Otherwise, the input values pass unchanged through the comparator. This guarantees that the smallest value will gradually move "down" the network and after L levels can be read from the bottom line. All the comparators belonging to the same level can operate in parallel because they do process independent data items. After finding the first smallest value, the search for the second smallest value (min_2nd) could be continued in a similar manner, but by processing N-1 remaining data items (see Figure 1b). For any $N \geq 2$, the number of comparators C(N) is given by Formula (1), and the number of lines (levels) of comparators L is given by Formula (2) below:

$$C(N) = 2 \times N - 3 \quad (1)$$

$$L = \log_2 N \tag{2}$$

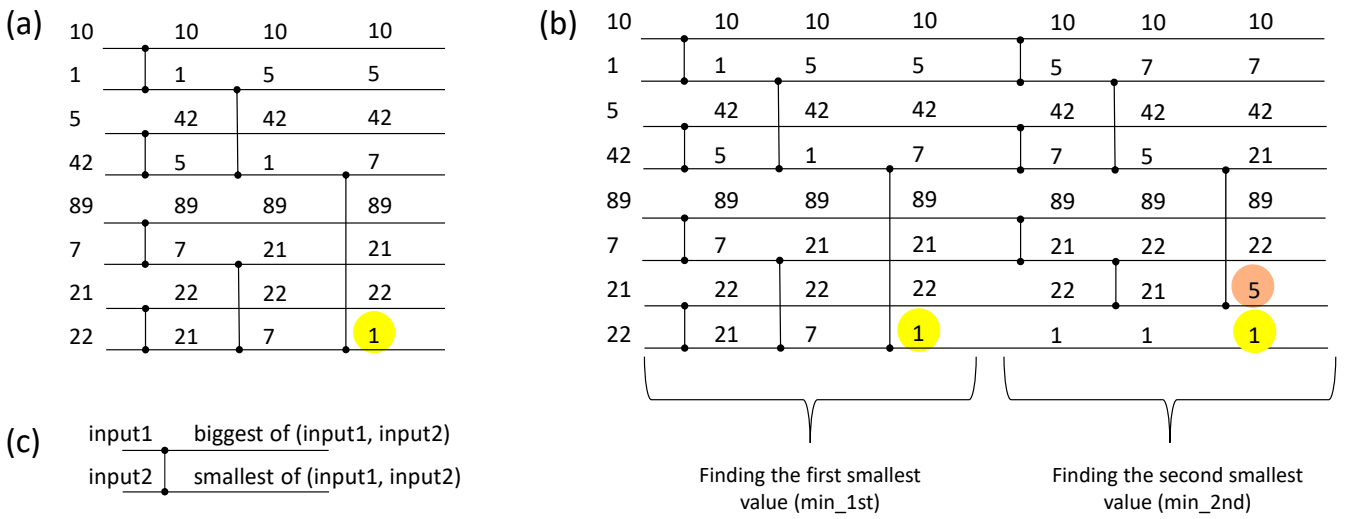

Figure 1. A network of comparators for finding the smallest data item (**a**). A network of comparators for finding two smallest data items (**b**). A single comparator (**c**).

The number of comparators in each level i = 0, 1, . . . , L − 1 is given by Formula (3):

$$\frac{2^L}{2^{i+1}} \tag{3}$$

The number of comparators could be reduced if we keep track of the compared data items at each level. The respective algorithm for N = 8 (and the same data set as in Figure 1) is depicted in Figure 2. Here, each data element requires a small memory that keeps a maximum of L data items. After every comparison operation, these memories need to be updated for the involved data items. For example, on the first level of comparators (the left-hand side of Figure 2), the first comparator analyzes the data items 10 and 1. Consequently, the value 1 is written to item 10's memory and the value 10 is written to item 1's memory. In the second level of comparators, the value 5 is compared to the value 1. Once again, the value 1 is written to item 5's memory and the value 5 is added to item 1's memory. After executing all L comparator levels, the smallest data item (min_1st=1) can be read from the bottom line, as before. However, to find the second smallest value, (min_2nd=5) just L-1=2 additional comparators are required, which would only analyze data values recorded in the memory of min_1st (i.e., data values 10, 5, and 7 in the example of Figure 2). As a result, the required number of comparators is reduced to:

$$C(N) = N + L - 2 \tag{4}$$

For large values of N, the resources (the number of comparators) are diminished considerably, but additional memory requirements and memory management make this approach not very suitable for parallel hardware implementations. For every data item, an independent memory block is required (either an embedded block RAM or a distributed memory constructed from flip-flops or look-up tables). The memory blocks have to be independent to allow for parallel writing operations. In any case, the respective multiplexing scheme tends to be quite complex.

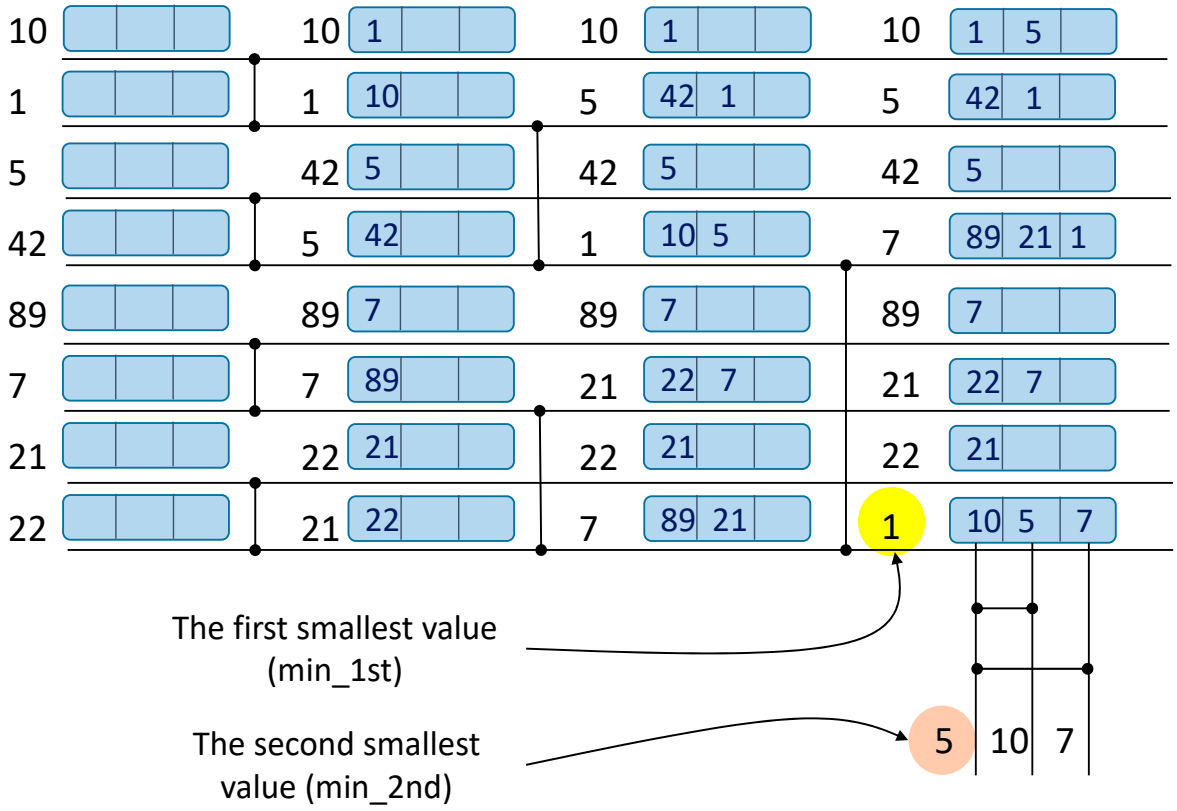

Figure 2. Network of comparators with memory for finding the two smallest data items.

The bit searching approach proposed in [14] for finding the two smallest values is not based on comparisons between the elements of a dataset; instead, the items' bits are scanned from the most significant bit to the least significant bit with the aid of filters constructed from FPGA look-up tables (LUTs). This scheme works very well for small data items (whose number of bits $M \leq 3$) but is not efficiently scalable for greater values of M.

That is why, for the implementations and experiments, the algorithm from Figure 1b will be used.

3. Specification Methods

Three different specification methods are evaluated in this work:

- VHDL RTL specification;
- High-level C code to be executed on an embedded soft processor;
- High-level C++ code to be synthesized with HLS tools.

All the implementations are targeted towards a low-cost/low-power xc7a100tcsg324-1 FPGA of the Xilinx Artix-7 family [21], available on a Nexys-4 prototyping board [22]. For the design entry, synthesis, simulation, implementation, and tests, Xilinx Vivado/Vitis 2020.2 tools were employed.

First, C++ code that permitted high-level modeling of the architecture in Figure 1b was developed. Such high-level modelling is very helpful for interaction, experiments, and the evaluation of the results. Moreover, the verification and debug may be performed at earlier stages, significantly reducing the verification effort because of the high speed of execution (compared to register-transfer level simulations) and due to the different easy-to-use debug tools being readily available. Therefore, the design errors are much easier to locate and fix at higher abstraction levels; this is especially true for determining the right data indices to be analyzed in different comparators of the network.

After exhaustive tests were executed in software, we advanced to the implementation in hardware, recurring to three different specification methods detailed below.

3.1. VHDL RTL Specification

For the most "traditional" low-level specification method, the VHDL language was used. The main objective was to design a fully parameterizable circuit that can easily be adapted to different values of N (number of data items), M (bit-width of a data item), and L ($N = 2^L$).

The hardware architecture is a combinational circuit, which generally follows the idea from Figure 1b by implementing a network of parallel comparators. The respective specification in VHDL is as follows (Listing 1):

Listing 1: VHDL specification of the algorithm for finding the two smallest data items.

```
entity two_min is
    generic (M : positive := 8;  -- data bit width
             L : positive := 3;  -- network's depth
             N : positive := 8); -- N must be equal to L**2
    port    (inData            : in std_logic_vector(M*N-1 downto 0);
             min_1st, min_2nd  : out std_logic_vector(M-1 downto 0)
             );
end two_min;

architecture Behavioral of two_min is
    signal workData : std_logic_vector(M*N-1 downto 0);
    type LEVELS is array (0 to L) of std_logic_vector(M*N-1 downto 0);
    signal sto_level1, sfrom_level1 : LEVELS;
    signal sto_level2, sfrom_level2 : LEVELS;
begin

process(inData) -- network for finding the smallest value min_1st
    variable to_level1, from_level1 : LEVELS;
begin
  to_level1(0) := inData;
  for i in 0 to L-1 loop
    from_level1(i) := to_level1(i);
    for j in 0 to ((2**L)/(2**(i+1)) - 1) loop
      if (to_level1(i)( ((2**i+j*2**(i+1))*M)-1 downto (2**i-1+j*2**(i+1))*M) <
   to_level1(i)(((2**i+j*2**(i+1)+2**i)*M)-1 downto ((2**i-1+j*2**(i+1)+2**i)*M)))
      then
from_level1(i)(((2**i+j*2**(i+1)+2**i)*M)-1 downto ((2**i-1+j*2**(i+1)+2**i)*M) ) :=
to_level1(i)( ((2**i+j*2**(i+1))*M)-1 downto (2**i-1+j*2**(i+1))*M );
from_level1(i)(((2**i+j*2**(i+1))*M)-1 downto (2**i-1+j*2**(i+1))*M ) :=
to_level1(i)( ((2**i+j*2**(i+1)+2**i)*M)-1 downto ((2**i-1+j*2**(i+1)+2**i)*M) );
      else
from_level1(i)( ((2**i+j*2**(i+1)+2**i)*M)-1 downto ((2**i-1+j*2**(i+1)+2**i)*M)) :=
to_level1(i)( ((2**i+j*2**(i+1)+2**i)*M)-1 downto ((2**i-1+j*2**(i+1)+2**i)*M) );
from_level1(i)( ((2**i+j*2**(i+1))*M)-1 downto (2**i-1+j*2**(i+1))*M ) :=
to_level1(i)( ((2**i+j*2**(i+1))*M)-1 downto (2**i-1+j*2**(i+1))*M );
      end if;
    end loop;
    to_level1(i+1) := from_level1(i);
  end loop;
  min_1st <= from_level1(L-1)(N*M-1 downto (N-1)*M);
  sfrom_level1 <= from_level1;
  sto_level1 <= to_level1;
end process;

-- A similar code specifying the network for finding the second smallest value which
-- gets source data from sfrom_level1(L-1) and produces the result min_2nd

end Behavioral;
```

In the VHDL code above, the **for** ... **loop** VHDL construction permits the required number of levels of comparators to be generated based on the parameter L. The parameter M configures the required comparators' width. The data to be processed must be supplied to the circuit as a long vector of all the concatenated data items (this vector has N × M bits). The results are generated on two outputs: min_1st and min_2nd. An analysis of the elaborated design confirms that the circuit corresponding to Figure 1b is constructed.

The proposed specification has been validated with the aid of VHDL testbench in the integrated Vivado simulator, and the conducted tests confirmed that the code is correct and can be used for synthesis and further integration with higher-level circuits to provide test input data and visualize the results.

3.2. Software Running on an Embedded Soft Processor

The second specification method that was analyzed consists of directly running the high-level C code (derived from the C++ code given in Listing 2) on an embedded soft core processor. For this purpose, a hardware platform, based on the MicroBlaze processor [23], has been developed using Vivado IP (Intellectual Property) Integrator tool. One instance of MicroBlaze core was created and configured to run baremetal code (no operating system) in 32-bit mode. The processor was optimized for performance with the instruction and data caches disabled and the debug module and the peripheral AXI data interface enabled. Two interfaces for memory accesses were used: local memory bus and AXI4 for peripheral access.

Listing 2: High-level modelling in C++ of the algorithm for finding the two smallest data items.

```
void swap(int& a, int& b)    // modelling a comparator
{   int aux = a;
    if (a < b)    {    a = b;    b = aux;  }
}
void ProcessData(vector<int>& data, int N)
{   int L = ceil(log2(N)); // number of levels = network's depth
    for(int i = 0; i < L; i++)
       for (int j = 0; j <= (pow(2,L))/(pow(2,(i+1))) - 1; j++)
            swap(data[pow(2, i) - 1 + j * pow(2, i + 1)],
                 data[pow(2, i) - 1 + j * pow(2, i + 1) + pow(2, i)]);
}
int main()      {
    vector<int> data; // … filling data vector with input values
    ProcessData(data, data.size());    // determining min_1st
    cout << "Smallest value: " << data[data.size()-1] << endl;
    data[data.size()-1] = data[data.size()-2]; // "delete" the smallest item
    ProcessData(data, data.size());    // determining min_2nd
    cout << "Second smallest value: " << data[data.size()-1];
    return 0;   }
```

Besides the processor itself, the following IP cores were added to the hardware platform:

- Local memory connected to the MicroBlaze through the local memory bus core;
- Debug module interfacing with the JTAG port of the FPGA to provide support for software debugging tools;
- UARTLite module implementing AXI4-Lite slave interface for interacting with the FPGA through UART from the host PC, which is used to interact with the MicroBlaze through a serial port terminal;
- Programmable AXI timer used for measuring the execution times;
- Fixed-interval timer responsible for synchronizing external displays;

- Several general-purpose input/output modules providing an interface to the board's peripheral devices such as push buttons, switches, LEDs, and 7-segment displays, used for tests and experiments;
- AXI interrupt controller that together with the concat module supports interrupts from the AXI timer, the UARTLite module, and the board's input peripheral devices;
- Clock and reset modules to generate the clock signal for the design and provide customized resets for the system's blocks;
- AXI interconnect with one slave and seven master interfaces. The interconnect core connects AXI memory-mapped master devices to several memory-mapped slave devices. The single slave port of the interconnect is connected to the MicroBlaze. The seven master ports are linked with the interrupt controller, UARTLite module, AXI timer, and four types of peripheral devices.

The final block diagram is illustrated in Figure 3.

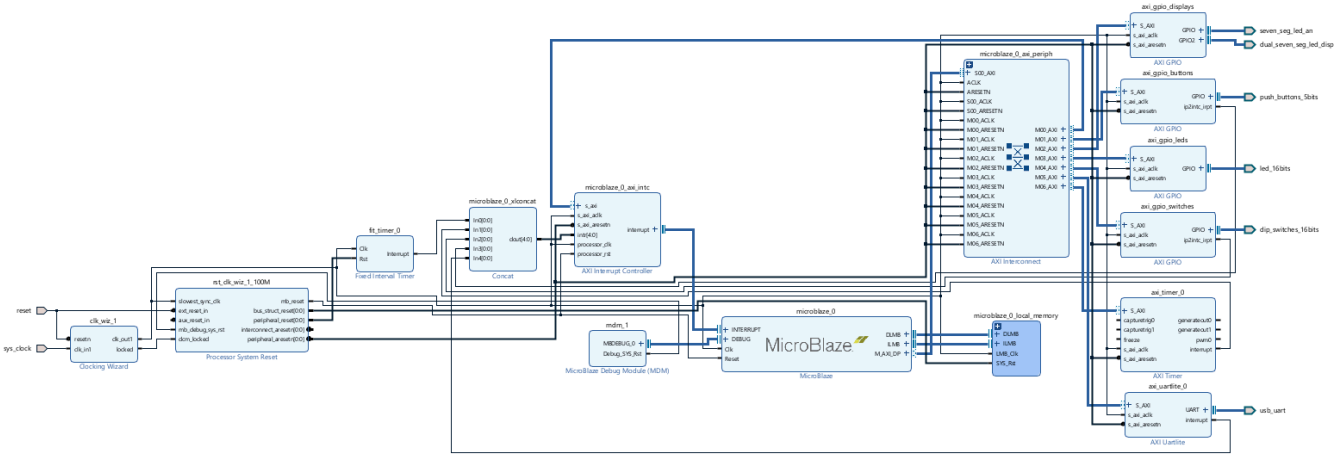

Figure 3. Block diagram of the microprocessor system used for experiments.

After the synthesis, implementation, and bitstream generation steps, the hardware platform was exported to Vitis, where C code was developed executing the following steps:

- Randomly generate input test data according to the given parameter N;
- Reset and start the programmable AXI timer;
- Execute the ProcessData function whose code is almost identical to the one given Listing 2 (the only difference is that instead of obtaining a vector of integer values, the function receives an array of integers (a pointer to the first array's element)).
- Stop the AXI timer and calculate the processing time;
- Print all the results on a serial port terminal interacting with the MicroBlaze though the UART-lite module.

The developed C code, which can be easily adapted to experiments with other problems, is given below in Listing 3:

Listing 3: C code of the algorithm to be executed on MicroBlaze.

```
void RestartPerformanceTimer()
{     XTmrCtr_Disable(XPAR_TMRCTR_0_BASEADDR, 0);
      XTmrCtr_SetLoadReg(XPAR_TMRCTR_0_BASEADDR, 0, 0x00000001);
      XTmrCtr_LoadTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, 0);
      XTmrCtr_SetControlStatusReg(XPAR_TMRCTR_0_BASEADDR, 0, 0x00000000);
      XTmrCtr_Enable(XPAR_TMRCTR_0_BASEADDR, 0);
}

unsigned int StopAndGetPerformanceTimer()
{  XTmrCtr_Disable(XPAR_TMRCTR_0_BASEADDR, 0);
      return XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, 0);
}

void swap(int* a, int* b)   // modelling a comparator
{   int aux = *a;
    if (*a < *b)
    {   *a = *b;    *b = aux;    }
}

void ProcessData(int* data, int N)
{   int L = ceil(log2(N)); // number of levels
    for(int i = 0; i < L; i++)
       for (int j = 0; j <= (pow(2,L))/(pow(2,(i+1))) - 1; j++)
           swap(&data[(int)(pow(2, i) - 1 + j * pow(2, i + 1))],
                &data[(int)(pow(2, i) - 1 + j * pow(2, i + 1) + pow(2, i))]);
}

int main()
{   init_platform();
    // … GPIO tri-state configuration
    unsigned int timeElapsed;
    const unsigned int N = 256;
    int data[N];                 // … filling memory with pseudo-random data
    RestartPerformanceTimer();  // reset and enable the AXI timer
    ProcessData(data, N);       // find the smallest value min_1st
    xil_printf("Smallest value: %04x\n", data[N-1]);
    data[N-1] = data[N-2];      // "delete" the smallest item
    ProcessData(data, N);       // find the second smallest value min_2nd
    xil_printf("Second smallest value: %04x\n", data[N-1]);
    timeElapsed = StopAndGetPerformanceTimer();// stop the AXI timer
    xil_printf("\n\rSoftware only two_min time: %d microseconds",
         timeElapsed / (XPAR_CPU_M_AXI_DP_FREQ_HZ / 1000000));
    cleanup_platform();
    return 0;
}
```

3.3. High-Level Synthesis

Finally, high-level synthesis technology has been explored based on Vitis HLS tools [24]. This approach is particularly interesting because of the relatively small effort required from the designer to produce the hardware circuit. That is, given the C++ code that was used for the high-level algorithm validation, it must be slightly adapted and the remaining steps that are required to run the code in programmable logic are automated by the Vitis HLS tool. Therefore, engineers with little hardware design experience might refer to this specification method to produce reasonable designs.

The following steps have been applied to test the algorithm with this approach:

- The algorithm C++ code given in Listing 2 was adapted to the HLS requirements. In particular, the ap_uint<> data type was used to define the arbitrary precision integer data type (basically, to be able to variate the M parameter between 8 and 1024 bits), declaring the input data array as follows: ap_uint<M> work_array[N];
- The code was then compiled, simulated, and debugged.
- The algorithm was synthesized to an RTL design and an RTL co-simulation was executed.
- The RTL implementation was packaged and exported as an RTL IP to be used in either of the design approaches explored in the previous two subsections (i.e., RTL design or a hardware platform where the generated IP acts as MicroBlaze co-processor).

Various synthesis directives permit the designer to improve the results of synthesis by driving the optimization engine towards the desired performance goals and RTL architecture. Some examples of the synthesis directives are unrolling loops (by default the loops are kept rolled; when unrolled all or several loop iterations might be executed in parallel, reducing the latency at the cost of occupying more hardware resources), configuring the type of memory used for arrays, and others. The synthesis reports provide information on the generated circuit latency, initiation interval, throughput, and resource utilization, and may direct the designer to the most appropriate synthesis directives to be applied.

The HLS C code of the top-level function is reproduced below in Listing 4:

Listing 4: C code of the algorithm used for HLS for finding the two smallest data items.

```
void swap(ap_uint<M>* a, ap_uint<M>* b)   // modelling a comparator
{   int aux = *a;
    if (*a < *b)    {   *a = *b;       *b = aux;  }       }

void ProcessData(ap_uint<M>* data, int N)
{   int L = ceil(log2(N)); // number of levels
    process_l1: for(int i = 0; i < L; i++)
    {   int comparators = (pow(2,L))/(pow(2,(i+1))) - 1;
            process_l2: for (int j = 0; j <= comparators; j++)
       {   #pragma HLS unroll
          swap(&data[(int)(pow(2, i) - 1 + j * pow(2, i + 1))],
               &data[(int)(pow(2, i) - 1 + j * pow(2, i + 1) + pow(2, i))]);
       }
    }
}

unsigned int two_min (ap_uint<N*M> input_data)
{   ap_uint<M> work_array[N];
   //… filling in the work array from input_data
   ProcessData(work_array, N);
   unsigned int min_1st = work_array[N-1];
   work_array[N-1] = work_array[N-2]; // "delete" the smallest item
   ProcessData(work_array, N);
   unsigned int min_2nd = work_array[N-1];
   //… producing the result from min_1st and min_2nd
   return result;
}
```

4. Experiments

All three considered design methods have been validated through simulations. Afterwards, the experiments were realized in the xc7a100tcsg324-1 FPGA of the Xilinx Artix-7 family. It is very important to underline that different specifications are validated and compared practically as is, i.e., no profound optimization has been executed over any of the methods. For example, in case of a low-level RTL design, it is possible to increase the synthesis and implementation tools' efforts to produce a faster circuit by raising the desired running frequency in the design constraints. Instead of this, the default Nexys-4 clock oscillator frequency (100 MHz) was applied and the reported worst slack analyzed to calculate the shortest supported clock period. In a similar manner, in the case of a MicroBlaze processor, higher GCC compiler optimization flags could be employed. Instead of this, default GCC flags were used. Finally, for HLS, the only synthesis optimization directive that has been applied is the unrolling of the loops to guarantee that all the comparators belonging to the same network level operate in parallel. This is because the objective of the research is not producing the fastest circuit for finding the two smallest values, but instead comparing different specification methods according to criteria such as the ease of use, test, change, and maintenance, portability, and the basic resource requirements and performance. Once a particular design method is selected, additional optimizations must be applied to produce the best circuit according to the target implementation criteria (such as the minimum circuit area, maximum throughput, etc.).

In a similar manner, increasing the N and M parameters as much as possible was not the main objective of this research and the experiments have been executed for relatively small values of M and N. All the designs themselves are scalable to support higher M/N but are subject to two important constraints: limited FPGA resources and a restricted I/O bandwidth. Recurring to a larger FPGA device and pipelining the network might solve the first problem (and this is often what the designers choose to employ). However, the communication overheads (related to supplying the input data to the circuit) will always limit the ideal theoretical throughput. To solve this problem, designs have been proposed that allow the processing to be overlapped in time with data transfers [15].

The results of the experiments with circuits synthesized and implemented from RTL specification in VHDL are summarized in Table 1. The FPGA device that was used has 63 400 LUTs, and the number of LUTs occupied by circuits with different M/N parameters is indicated in the "LUTs" column. A simple calculation will demonstrate that the circuit never exceeds 20% of the FPGA logical resources for $N \leq 128$ and can therefore be employed in embedded applications. The occupied resources are comparable to the sorting-based design XS [13]. When compared to [14], the solution reported here uses fewer FPGA logic resources. It is clear that the required resources are not linearly scalable with the increase of N/M. For instance, if M is doubled, the resulting circuit will not occupy double the amount of the same resources as the circuit synthesized for M/2; instead, the resources will increase more than just two times. This is because some of the FPGA LUTs would be used for routing. The results from Table 1 confirm this observation. Actually, the number of the network's comparators grows as illustrated in the graph in Figure 4. To estimate the scalability, additional experiments were executed for N = 1024 and N = 2048 that revealed that the circuit exceeds the capacity of the low-cost FPGA that was used when N = 2048 and M = 32. So, for large values of N and M, the required number of comparators (and consequently the occupied FPGA logic resources) can be a limiting factor for the respective circuit. One possible improvement to the suggested hardware architecture is to reuse the same group of comparators two times (once to find min_1st and the second time to find min_2nd). The resources can be reduced by constructing a sequential circuit that reuses just one group of comparators but takes two steps to obtain the result. An N×M-bit register is required to store the cascading outputs at the end of each step. The overall throughput of the sequential circuit is slightly lower than that of the combinational circuit (because of additional setup requirements on the flip-flops composing the register) but the resources are reduced in twofold.

Table 1. Hardware resources and performance for implementing the selected algorithm in xc7a100tcsg324-1 FPGA for M = 8/32 and N = 2^L, L = 3, 4, . . . , 8 using RTL specification in VHDL.

M	N	LUTs	F_{max}
8	8	64	81 MHz
	16	124	63 MHz
	32	168	50 MHz
	64	250	41 MHz
	128	944	31 MHz
	256	1839	28 MHz
32	8	143	67 MHz
	16	228	51 MHz
	32	397	43 MHz
	64	1031	33 MHz
	128	11,614	21 MHz
	256	21,619	18 MHz

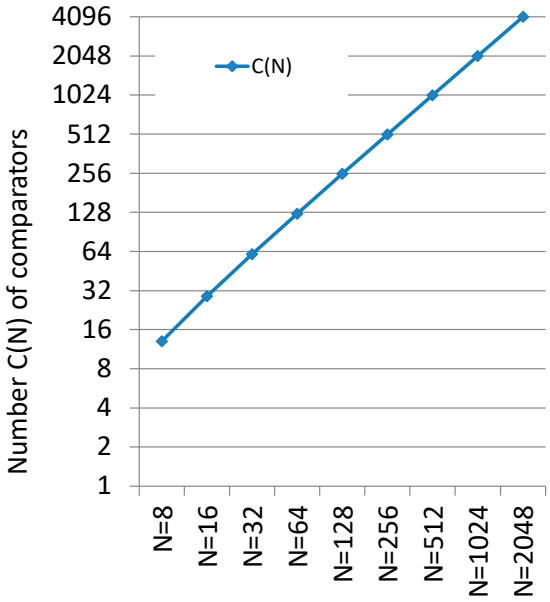

Figure 4. Number of M-bits comparators for N = 2^L, L = 3, 4, . . . , 11 used in the implemented network for finding the two smallest data items.

The column "F_{max}" indicates the maximum operating frequency calculated as described above. As expected, the larger the value of N, the greater the circuit latency and the lower the supported clock frequency. This is because the larger the value N, the larger the number L of comparator levels. This problem can easily be solved with circuit pipelining by inserting registers between the levels of the comparators.

The second tested design is based on the MicroBlaze soft processor core as described in Section 3.2. In this case, the required hardware resources and the maximum operating clock frequency are always the same, regardless of the value of N. The synthesized and implemented design occupies 2223 LUTs (3.5% of the used FPGA resources) and can run at 129 MHz. The hardware resources are comparable with that of the RTL VHDL design for M = 32 and N = 64. The programmable AXI timer is used to measure the circuit performance and the results are summarized in Table 2 (M is always equal to 32 as the C unsigned int data type is used to store and manipulate input data values). All xil_printf statements have been removed to allow for fair time measurements.

Table 2. Execution time of the C code, presented in Section 3.2, on an embedded MicroBlaze processor for M = 32 and N = 2^L, L = 3, 4, . . . , 8 (LUTs = 2223, F_{max} = 129 MHz).

N	Execution Time
8	45.5 ms
16	102.7 ms
32	212.1 ms
64	425.7 ms
128	847.7 ms
256	1686.7 ms

Finally, Vitis HLS synthesis reports were analyzed to assess the performance of the respective implementations described in Section 3.3. The synthesized design requires about 11,250 LUTs (~18%), and a significant portion of these LUTs is due to calculation of different powers of 2 (function pow) when generating the indices of the data to be used for comparators' inputs. It is interesting to observe that both the required LUT count and the operating frequency do not vary significantly with N. The reasonable explanation is that the number of comparators per level varies among networks levels (according to Formula (3) and visually observable in Figure 1b). This means that the inner loops in the code in Section 3.3 have variable bounds. It is known that some of the optimizations that Vitis HLS can apply are prevented when the loop has variable bounds. Since variable bound loops cannot be unrolled, they not only prevent the unroll directive being applied, but they also block pipelining of the levels above the loop [24]. Thus, only the execution time increases with N, but the resource (LUTs) augment is marginable. In contrast to this, the number of required storage elements grows significantly with N (for instance, for N = 128/M = 32, the circuit utilizes 11281 flip-flops, while for N = 32/M = 32, 8047 flip-flops are required). The evaluated performance parameters allowed for an estimate of the synthesized circuit latency and the results are presented in Table 3. These results are only estimates because Vitis HLS is not aware of the routing delays that will be present in the final implemented circuit. As in the previous two designs, additional optimizations must be applied to improve performance. This is the only one from the considered three designs that has just been validated through simulation and no tests in FPGA have been executed. To perform tests on hardware, the RTL design produced by Vitis HLS must be exported to be used by other tools (such as an IP core to be added to Vivado IP catalog).

Table 3. The results of high-level synthesis of the selected algorithm for M = 32 and N = 2^L, L = 3, 4, . . . , 8 (LUTs $\approx$ 12,500, F_{max} = 120 MHz).

N	Execution Time
8	1652 ns
16	2203 ns
32	2837 ns
64	3404 ns
128	4089 ns
256	4673 ns

An analysis of Tables 1–3 permits the conclusion that RTL VHDL-based designs run at a significantly lower frequency compared to the MicroBlaze implementation, but the throughput is much higher. For example, for M = 32/N = 32, the MicroBlaze-based circuit requires ~212 ms to produce the result while the circuit synthesized from VHDL is ready to obtain a new set of inputs every 23 ns. Moreover, many more hardware resources are used for modest values of N (N $\leq$ 64); however, for N $\geq$ 128, the MicroBlaze-based implementation turns out to be more beneficial from the LUT-usage point of view. The circuits generated through HLS require many more resources than any other method. These circuits are significantly faster than executing C code on MicroBlaze but are much slower

than RTL VHDL-based designs. For example, for M = 32/N = 32, the circuit generated through HLS requires 2837 ns to produce the result, which is ~120 times slower than the circuit synthesized from VHDL. Figure 5 summarizes the calculated processing latency for all three analyzed methods for M = 32 and N = 2^L, L = 3, 4, . . . , 8.

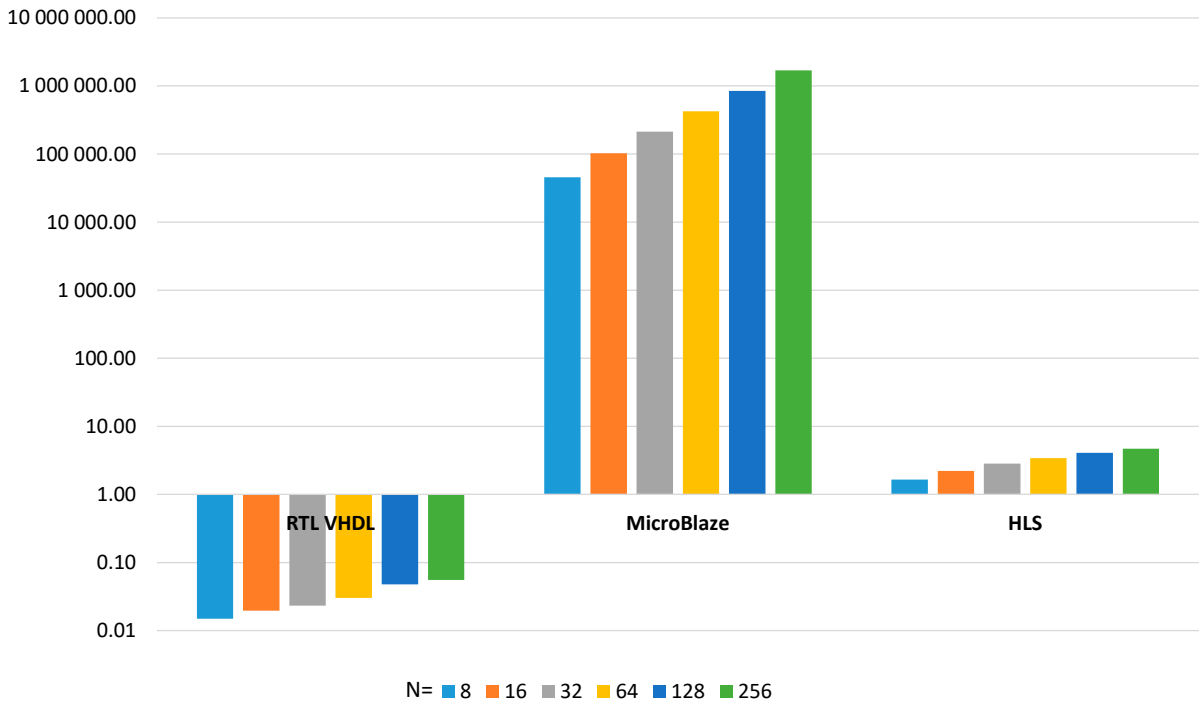

Figure 5. Comparison of the circuit latency (expressed in µs in logarithmic scale) of VHDL RTL design, MicroBlaze-based design, and the circuit generated using HLS for N = 2^L, L = 3, 4, . . . , 8.

5. Conclusions

This paper discusses a network-based parallel data processing algorithm, which permits two minimum values to be found in a set of N data items. The algorithm was modelled using software and then three design approaches were evaluated for implementing it in an FPGA of the Artix-7 family. The results of the experiments revealed that the circuits produced from low-level specifications in a hardware description language (VHDL) are significantly less resource consuming for moderate values of N and exhibit an incomparably better performance. However, the time needed to describe, implement, and validate such a circuit is quite significant, and a profound knowledge of the target hardware architecture, design methods, and HDL syntax is required. The generated circuit specification is fully parameterizable, which is a great benefit since the code may be simulated and debugged for small values of N and then configured through VHDL generic constants to achieve particular application requirements. To be truly competitive, profound optimizations have to be applied to all the considered designs.

Regarding the power consumption, the RTL design tends to consume significantly less power than the MicroBlaze-based hardware platform. This is the expected result, since the MicroBlaze design uses a number of auxiliary components, such as the UART module, which contribute to spending more power. The total on-chip power calculated by the Vivado power analysis tool from the implemented netlist for the RTL-based system with N = 256 and M = 32 is 0.109 W, while the MicroBlaze-based system requires 0.241 W.

When comparing the ease of code development, change, and maintenance, the MicroBlaze-based approach requires almost no design effort as the C code used for modeling can be executed with just slight modifications. The only work consists in creating the base hardware platform, including the MicroBlaze processor itself, the timers used for the performance evaluation, the UART module for communication and debugging, and the

basic support for the input/output. However, the developed hardware platform can easily be reused for experiments with other algorithms, in contrast to the VHDL design, which is completely tailored to the considered task. The price to pay for these facilities is the performance, which is definitely the lowest among the three considered design approaches.

Recurring to HLS proved to be more difficult than was expected because although the source specification code can be almost directly reused from the modeling stage, and the validation step is very simple and straightforward, the generated circuit is not capable of running with an initiation interval equal to 1 and optimization directives must certainly be applied to produce a lower latency design. The HLS tool also prevented the unrolling of the variable bounds loops. Moreover, despite many researchers mentioning that no specific hardware knowledge is required to design with HLS tools, this information seems to be false, because to analyze the synthesis reports and to work efficiently with the profiling tools, a deep understanding of many hardware-related concepts (such as initiation intervals, loop latency, trip count, etc.) is definitely essential. In addition, high-level synthesis takes quite long time, comparable to the synthesis and implementation time of VHDL RTL specification. These conclusions do apply particularly to Vitis 2020.2 HLS tools; no study or analysis of alternative HLS tools has been executed.

The main conclusion is that for quick tests and non-time critical applications, MicroBlaze (or other soft/hard processor)-based designs are much easier to develop, experiment with, and manage. When performance is the critical factor, low-level HDL specifications are the clear winners with the only drawback being that a wide RTL design experience is necessary to produce, test, and optimize high-performance circuits. HLS tools have been introduced as a great opportunity for software developers to enable them to synthesize accelerated hardware from a software algorithm written in a high-level language (usually C/C++). However, many important concepts involved in the design process, although very familiar to people with RTL design experience, are not sufficiently clear to software engineers to enable them to achieve high-performance designs. Moreover, a typical software refers to such programming techniques as recursion and dynamic memory allocation—these techniques are not synthesizable. This means the code must be refactored to make it both synthesizable and as "parallel" as possible (so that the HLS tool could infer parallelism and exploit it to achieve greater performance) [24]. Therefore, HLS tools are more suitable to designers and software programmers who are familiar with concurrent computing concepts and/or special platforms, such as GPUs. Once these topics are dominated, the HLS approach provides quite substantial productivity gains over RTL design. This conclusion is in concordance with related works [25–28], where the authors point out that the HLS method contributes to accelerating the development of data processing algorithms, but also that engineers need to dominate technologies of parallel programming and possess/acquire knowledge about the structure and features of FPGA. Moreover, HLS usually permits individual components to be synthesized that must be further integrated at the RT level, implying that the system-level verification needs to be performed at lower levels of abstraction, which significantly diminishes the benefits of using HLS [29]. So, for complex designs, a trade-off between the design effort and performance should be reached, and various design methods need to be explored for each case [30,31].

Funding: This work was supported by National Funds through the FCT-Foundation for Science and Technology, in the context of the project UIDB/00127/2020.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Knuth, D.E. *The Art of Computer Programming. Sorting and Searching*, 3rd ed.; Addison-Wesley: Boston, MA, USA, 2011.
2. Sklyarov, V.; Skliarova, I. Design and implementation of counting networks. *Comput. J.* **2015**, *97*, 557–577. [CrossRef]
3. Mueller, R.; Teubner, J.; Alonso, G. Sorting Networks on FPGAs. *Int. J. Very Large Data Bases* **2012**, *21*, 1–23. [CrossRef]
4. Zuluaga, M.; Milder, P.; Puschel, M. Computer Generation of Streaming Sorting Networks. In Proceedings of the 49th Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 1245–1253.
5. Sklyarov, V.; Skliarova, I.; Rjabov, A.; Sudnitson, A. Fast Iterative Circuits and RAM-based Mergers to Accelerate Data Sort in Software/Hardware Systems. *Proc. Est. Acad. Sci.* **2017**, *66*, 323–335. [CrossRef]
6. Najafi, M.H.; Lilja, D.J.; Riedel, M.D.; Bazargan, K. Low-Cost Sorting Network Circuits Using Unary Processing. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2018**, *26*, 1471–1480. [CrossRef]
7. Norollah, A.; Derafshi, D.; Beitollahi, H.; Fazeli, M. RTHS: A Low-Cost High-Performance Real-Time Hardware Sorter, Using a Multidimensional Sorting Algorithm. *IEEE Trans. Very Large Scale Integr. VLSI Syst.* **2019**, *27*, 1601–1613. [CrossRef]
8. Skliarova, I. Accelerating Population Count with a Hardware Co-Processor for MicroBlaze. *J. Low Power Electron. Appl.* **2021**, *11*, 20. [CrossRef]
9. Parhami, B. Efficient Hamming weight comparators for binary vectors based on accumulative and up/down parallel counters. *IEEE Trans. Circuits Syst. II Express Briefs* **2009**, *56*, 167–171. [CrossRef]
10. Skliarova, I.; Sklyarov, V. *FPGA-Based Hardware Accelerators*; Springer: Cham, Switzerland, 2019.
11. Pilz, S.; Porrmann, F.; Kaiser, M.; Hagemeyer, J.; Hogan, J.M.; Rückert, U. Accelerating Binary String Comparisons with a Scalable, Streaming-Based System Architecture Based on FPGAs. *Algorithms* **2020**, *13*, 47. [CrossRef]
12. Umuroglu, Y.; Conficconi, D.; Rasnayake, L.K.; Preußer, T.B.; Själander, M. Optimizing Bit-Serial Matrix Multiplication for Reconfigurable Computing. *ACM Trans. Reconfig. Technol. Syst.* **2019**, *12*, 1–24. [CrossRef]
13. Wey, C.L.; Shieh, M.D.; Lin, S.Y. Algorithms of Finding the First Two Minimum Values and Their Hardware Implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2008**, *55*, 3430–3437.
14. Tzimpragos, G.; Kachris, C.; Soudris, D.; Tomkos, I. A Low-Latency Algorithm and FPGA Design for the Min-Search of LDPC Decoders. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshop—IPDPSW'2014, Phoenix, AZ, USA, 19–23 May 2014.
15. Skliarova, I. A Survey of Network-Based Hardware Accelerators. *Electronics* **2022**, *11*, 1029. [CrossRef]
16. Miranda, G.H.S.; Alexandrino, A.O.; Lintzmayer, C.N.; Dias, Z. Approximation Algorithms for Sorting λ-Permutations by λ-Operations. *Algorithms* **2021**, *14*, 175. [CrossRef]
17. Marszałek, Z. Parallelization of Modified Merge Sort Algorithm. *Symmetry* **2017**, *9*, 176. [CrossRef]
18. Teng, J.-H.; Leou, R.-C.; Chang, C.-Y.; Chan, S.-Y. Harmonic Current Predictors for Wind Turbines. *Energies* **2013**, *6*, 1314–1328. [CrossRef]
19. Zhang, Z.; Zhao, J.; Yan, X. A Web Page Clustering Method Based on Formal Concept Analysis. *Information* **2018**, *9*, 228. [CrossRef]
20. Nelson, M.; Sorenson, Z.; Myre, J.M.; Sawin, J.; Chiu, D. Parallel acceleration of CPU and GPU range queries over large data sets. *J. Cloud Comput. Adv. Syst. Appl.* **2020**, *9*, 44. [CrossRef]
21. Xilinx, Inc. 7 Series FPGAs Data Sheet: Overview. 2020. Available online: https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf (accessed on 13 March 2022).
22. Digilent, Nexys 4 Reference Manual. Available online: https://reference.digilentinc.com/reference/programmable-logic/nexys-4/reference-manual (accessed on 13 March 2022).
23. AMD/Xilinx, Inc. MicroBlaze Processor Reference Guide. UG984 (v2019.1). 2019. Available online: https://docs.xilinx.com/v/u/2019.1-English/ug984-vivado-microblaze-ref (accessed on 17 March 2022).
24. AMD/Xilinx, Inc. Vitis High-Level Synthesis User Guide UG1399 (v2021.2) 15 December 2021. Available online: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2021_2/ug1399-vitis-hls.pdf (accessed on 17 March 2022).
25. Farahmand, F.; Nguyen, D.T.; Dang, V.B.; Ferozpuri, A.; Gaj, K. Software/Hardware Codesign of the Post Quantum Cryptography Algorithm NTRUEncrypt Using High-Level Synthesis and Register-Transfer Level Design Methodologies. In Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Barcelona, Spain, 8–12 September 2019; pp. 225–231. [CrossRef]
26. Wang, Z.; Carrion Schafer, B. SSSL: Secure Search Space Locking of Behavioral IPs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2022**, *41*, 1868–1877. [CrossRef]
27. Lavrentiev, M.; Lysakov, K.; Marchuk, A.; Oblaukhov, K.; Shadrin, M. Algorithmic Design of an FPGA-Based Calculator for Fast Evaluation of Tsunami Wave Danger. *Algorithms* **2021**, *14*, 343. [CrossRef]
28. Fingeroff, M. High-Level Synthesis, It's Still Hardware Design, Siemens Digital Industries Software White Paper. 2022. Available online: https://resources.sw.siemens.com/en-US/white-paper-high-level-synthesis-its-still-hardware-design?mid=13094853&PC=L&c=2022_06_30_csd_new_collateral_s1 (accessed on 5 July 2022).
29. Si, Q.; Shetty, S.; Carrion Schaefer, B. Building Complete Heterogeneous Systems-on-Chip in C: From Hardware Accelerators to CPUs. *Electronics* **2021**, *10*, 1746. [CrossRef]

30. Zamiri, E.; Sanchez, A.; Yushkova, M.; Martínez-García, M.S.; de Castro, A. Comparison of Different Design Alternatives for Hardware-in-the-Loop of Power Converters. *Electronics* **2021**, *10*, 926. [CrossRef]
31. Minutoli, M.; Castellana, V.G.; Saporetti, N.; Devecchi, S.; Lattuada, M.; Fezzardi, P.; Tumeo, A.; Ferrandi, F. Svelto: High-Level Synthesis of Multi-Threaded Accelerators for Graph Analytics. *IEEE Trans. Comput.* **2022**, *71*, 520–533. [CrossRef]