Automated Software Design for FPGAs on an Example of Developing a Genetic Algorithm

Anatoliy Doroshenko [1,2], Volodymyr Shymkovych [2], Olena Yatsenko [1], and Tural Mamedov [1]

[1] *Institute of Software Systems of National Academy of Sciences of Ukraine, Glushkov prosp. 40, Kyiv, 03187, Ukraine*
[2] *National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Peremohy prosp. 37, Kyiv, 03056, Ukraine*

Abstract

The paper proposes the method and software tools for automated design and synthesis of parallel programs for field-programmable gate arrays (FPGAs) based on the algebra-algorithmic approach. The developed facilities provide the construction of parallel algorithm schemes by superposition of language constructs of Glushkov's system of algorithmic algebra. Based on schemes, the corresponding source code in VHDL is automatically generated, which is further executed on an FPGA. The flexibility of reconfigurable FPGA architecture is very attractive for the realization of computationally complex algorithms and allows synthesizing high-efficiency solutions that differ from other architectures by substantially less energy consumption at similar performance rates. The approach to the automated design of parallel programs for FPGA is illustrated with an example of developing a genetic algorithm utilized at the training of multilayer neural networks. The results of the experiment consisting in executing the generated program code on an FPGA are given.

Keywords

Algorithmic algebra, automated algorithm design, FPGA, genetic algorithm, parallel computation, software synthesis, neural network

1. Introduction

The rapid growth of integration degree and functional complexity of modern electronic devices results in the necessity of improving and developing methods of designing and programming integrated circuits, in particular, field-programmable gate arrays (FPGAs). FPGAs contain an array of programmable logic blocks, and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together, like many logic gates that can be inter-wired in different configurations. Logic blocks can be configured to perform complex combinational functions or merely simple logic gates. In most FPGAs, logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory. Many FPGAs can be reprogrammed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software. To define the behavior of the FPGA, the user provides a design in a hardware description language (HDL) or as a schematic design. The most common HDLs are VHDL [1] and Verilog [2] as well as extensions such as SystemVerilog [3]. VHDL provides a high-level abstraction for describing hardware facilities owing to the availability of a set of predefined data types and a possibility to create user-defined hierarchically organized data types based on the basic ones built into the language. Designing in HDLs is rather a complex process and has been compared to the equivalent of programming in assembly languages. Therefore, there is a need to raise the abstraction level of design.

ICTERI-2021, Vol I: Main Conference, PhD Symposium, Posters and Demonstrations, September 28 – October 2, 2021, Kherson, Ukraine
EMAIL: doroshenkoanatoliy2@gmail.com (A. Doroshenko); v.shymkovych@kpi.ua (V. Shymkovych); oayat@ukr.net (O. Yatsenko); tural.mamedov@outlook.com (T. Mamedov)
ORCID: 0000-0002-8435-1451 (A. Doroshenko); 0000-0003-4014-2786 (V. Shymkovych); 0000-0002-4700-6704 (O. Yatsenko); 0000-0003-3029-5834 (T. Mamedov)

In the previous works [4–7], we had been developing a theory, methodology, and tools for automated program construction, based on Glushkov's system of algorithmic algebra (SAA) [4, 5]. The specific feature of the developed methodology consists in the formalization of processes of design and synthesis of algorithms and programs. The algorithms are designed in terms of high-level schemes represented in SAA. The formal facilities and software automation tools for designing parallel programs for multicore CPUs [6], Nvidia GPUs using CUDA [5], and heterogeneous platforms using OpenCL [7] were developed. The purpose of this paper is to apply our tools for the automated design of parallel programs for FPGAs in VHDL language and illustrate the approach with an example of designing a genetic algorithm used at the training of multilayer neural networks. The results of the experiment consisting in executing the developed program on an FPGA are also given.

2. Algebra of Algorithms and Designing of Parallel Programs for FPGA

The design of parallel programs for FPGA being proposed is based on the system of algorithmic algebra [4] focused on high-level construction and transformation of algorithms represented in the form of schemes. SAA is the two-sorted algebra $GA = <\{Pr, Op\}; \Omega_{GA}>$, where Pr and Op are the sets of predicates and operators defined on an information set; Ω_{GA} is the signature consisting of logic operations (disjunction, conjunction, negation) and operator constructs, in particular:

- serial execution of operators: "*operator*1"; "*operator*2" ;
- branching: IF '*condition*' THEN "*operator*1" ELSE "*operator*2" END IF ;
- for loop: FOR (*counter* FROM *start* TO *fin*) "*operator*" END OF LOOP .

In SAA, identifiers of basic and compound predicates are enclosed in single quotes and identifiers of operators are written with double ones. The representations of algorithms in SAA are called SAA schemes. The main difference of Glushkov's SAA from other procedural programming languages is that it allows to specify programs in algebraic and natural linguistic form, and contains facilities for formal program transformation. The developed Integrated toolkit for Designing and Synthesis of programs (IDS) [4–6] provides automated construction of algorithm schemes and generation of corresponding code in target programming languages (C, C++, Java). Algorithms are designed using a list of SAA constructs and a tree. The user chooses the constructs from the list and adds them to an algorithm tree. On each step of the construction process, the system allows a user to select only those operations, the insertion of which into a scheme does not break its syntactical correctness. The algorithm tree is then used for the automatic generation of SAA scheme text and program code. The mapping of each SAA construct to a text in a programming language is specified as a code template in the IDS database. In this paper, we add new SAA constructs intended for high-level design of programs for FPGA in VHDL language.

VHDL [1] is a formal notation aimed at the description and logic organization of a digital system. The function of the system is defined as the conversion of values at inputs into values at outputs. The organization of the system is defined by a set of connected components. The language is intended for modeling primarily on a gate level, register-transfer level, and chip frames, and is used at a synthesis of devices. VHDL has facilities for describing asynchronous parallel processes.

An entity and an architecture belong to the main concepts in the VHDL language.

The entity is defined as an interface of a project object. It's a description of a project component having well-defined inputs and outputs and performing a certain function. It can represent the whole system being designed, some subsystem, device, node, chipboard, macrocell, logic unit, etc. The description of the entity in the SAA language is the following:

ENTITY *entity_name* IS
  PORT ("*operator*")
END OF ENTITY,

where *entity_name* is the identifier of the project object; "*operator*" are basic operator(s) declaring input and output ports. The examples of operators declaring input and output ports can be the following:

"Signal (*name*) direction (*dir*) of type (*tp*)";
"Signal (*name*) direction (*dir*) of type (*tp*) and range (*rng*) with initial value (*val*)",

where *dir* can be *in*, *out*, or *inout*.

An example of an entity declaration for a combinatorial circuit implementing a logical function $f = (x1 \text{ and } x2) \text{ or } x3$ (see Figure 1) is the following:

ENTITY *and_or* IS
  PORT (
    "Signals (*x*1, *x*2, *x*3) direction (in) of type (*bit*)";
    "Signal (*f*) direction (out) of type (*bit*)");
END OF ENTITY.

The architecture defines the behavior of the system or its structure on a functional level of its description. The description of the architecture in the SAA language is the following:

ARCHITECTURE *arch_name* of *entity_name* IS
  DECLARATIONS ("*operator*1");

  "*operator*2"
END OF ARCHITECTURE,

where *arch_name* is the identifier of the architecture; *entity_name* is the name of the system (entity) for which the architecture is defined; "*operator*1" defines architecture declarations which may typically be any of the following: type, subtype, signal, constant, file, alias, component, attribute, function, procedure, configuration specification; "*operator*2" are operator(s) describing the system behavior.

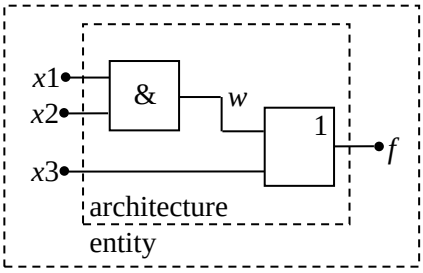

Figure 1: Combinational logic circuit

The operator of a process belongs to parallel operators in VHDL. It defines the independent sequential behavior of some part of a project, described by an ordered set of sequential operators. The construction defining the process in SAA is the following:

PROCESS *name* (*signal*1, *signal*2, …) IS
  DECLARATIONS ("*operator*1");

  "*operator*2"
END OF PROCESS,

where *name* is the identifier of the process followed by an optional list of signals which cause the process to be activated; "*operator*2" defines the process body.

As an example, consider the architectures *a*1 and *a*2 defining the behavior of the above combinatorial circuit:

ARCHITECTURE *a*1 of *and_or* IS

(*f* <= (*x*1 and *x*2) or *x*3);
END OF ARCHITECTURE;

ARCHITECTURE *a*2 of *and_or* IS
DECLARATIONS ("Signal (*w*) of type (*bit*)");

(*w* <= *x*1 and *x*2);
PROCESS *p*1 (*w*, *x*3) IS
(*f* <= *w* or *x*3);
END OF PROCESS;
END OF ARCHITECTURE,

where <= is the operation assigning a value to a signal.

Other examples of application of the above operations are given in Section 3.

3. Designing a Parallel Genetic Algorithm for FPGA

A genetic algorithm is a simple model of evolution in nature, implemented as a computer program [8, 9]. It reflects the processes of genetic inheritance and natural selection. It imitates the evolution of the population as a cyclic process of crossover and generation change. Genetic algorithms in various forms are used for solving scientific and technical problems [10–19]. In machine learning, they are used at designing neural networks and manipulation robotics.

One of the main features of neural networks is a parallel processing of signals. Multilayer neural networks are homogenous computing environments. According to the neuroinformatics terminology, they are universal parallel computing structures intended for solving various classes of tasks. In implementing neural networks on FPGA, each network layer is working in parallel with others, which allows using the pipeline principle. Neurons in each layer work in parallel too according to the principle of multiprocessor processing of data. That is, every neuron is a separate process, and processing information in each neuron is carried out simultaneously. Each neuron is presented as a separated block consisting of several parallel processes, and the neural network is a multiprocessor system. Programming language (VHDL) allows to explicitly define signals launching a process. For launching the computing process with a neuron, the input signal of the neuron is used. In our previous works [11–13], a method for implementing nonlinear neural network activation functions on FPGA was developed. Examples of realization of sigmoidal neural network activation functions and Gaussian activation functions for radial-base neural networks are considered. Hardware implementation of activation functions, artificial neurons, and a series of neural networks has been performed. The comparison with existing analogues on parameters of speed, the used hardware resource, and accuracy is executed.

In this section, a parallel genetic algorithm for training neural networks on FPGA is designed in SAA with the further generation of VHDL code. Every neuron corresponds to a process in the VHDL language. Based on this, the neurons (separate processes) are naturally are combined into a network: the outputs of the preceding network layer launch the processes of the next layer. The training of an artificial neural network consists in the tuning of weight coefficients $w_{i,j}$ of its basic elements, as a result of which the network performs certain tasks as recognition, optimization, approximation, and controlling.

In developing the parallel computing system of optimization of weight coefficients of neural networks, it is necessary to explore the characteristics of the algorithm being implemented [10]. Preparation of hardware implementation of neural network training procedures with the genetic algorithm is done based on the graph

$$GDF = (A, D),$$

where A is a set of vertices corresponding to operations; D is a set of edges representing data flows.

Figure 2 shows the graph of computations in a genetic algorithm with tasks highlighted that can be executed in parallel.

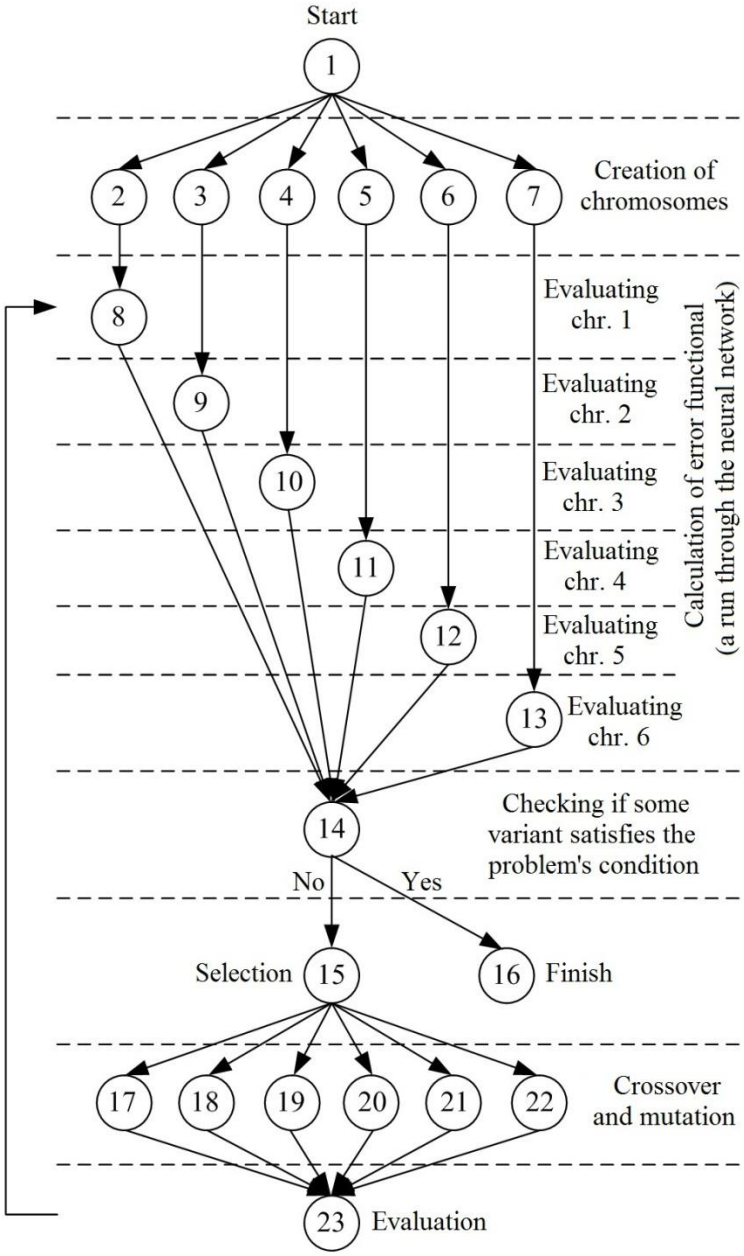

Figure 2: The graph of computations in a parallel genetic algorithm

The implementation of the genetic algorithm on FPGA consists of the following steps.

Step 1. Initialization of the initial population with chromosomes that contain information about the values of the weight coefficients of the network with a given structure. A set of chromosomes is represented as a two-dimensional array (see Table 1). Each row of the table corresponds to a chromosome and contains information about the whole set of weight coefficients of the network.

Step 2. Evaluation of chromosomes of the current population. Each chromosome is decoded to a set of weight coefficients of the neural network. Values of the fitness function are calculated, which takes into account an error and network complexity. This function defines the difference between the obtained network output and the required one.

Step 3. If one of the values satisfies the problem's condition, then go to step 7.

Step 4. Selection of chromosomes for further crossover and mutation, which is done by sorting according to the value of the fitness function.

Steps 5 and 6. Application of the operators of crossover and mutation for chromosomes selected on a previous step.

Table 1
Representation of chromosomes' population with a two-dimensional array

Chromosome's number	Weight coefficients							
	1st neuron			2nd neuron			*n*-th neuron	
	1	2	3	4	5		*n*–1	*n*
1	-1	-1	-1	-1	-1	...	-1	-1
2	-1	-1	-1	0	0	...	0	0
3	1	1	1	1	1	...	1	1
...								
k–1	0	0	0	0.5	0.5	...	-1	-1
k	0.5	0.5	0.5	0.5	0.5	...	0.5	0.5

The following values are calculated:

$$n_2 = \frac{1+\sqrt{1+8N}}{2}, \quad n_1 = n_2 - 0.5 - \sqrt{(n_2 - 0.5)^2 - 2N},$$

$$H' = \sum_{i=1}^{n_1} \sum_{j=i+1}^{n_2} \sum_{m=1}^{mMax} \left(\sum_{k=1}^{b-1} H_{imk} 2^k + H_{jmb} 2^b + \sum_{k=b+1}^{B} H_{imk} 2^k \right), \; n_{1,2} < N,$$

where H' is a new set of chromosomes, H is a previous one. Each chromosome is considered as a set of bits. Then the first sum defines the first chromosome for crossover (its index is *i*), the second sum defines the second chromosome (index *j*). The expression in brackets defines the operations with bits of the new chromosome, where *b* and *k* are indexes of bits; 2^b, 2^k denote the position of a bit in a binary number; H_{ik} is *k*-th bit of *i*-th chromosome; H_{jb} is a bit with number *b* in *j*-th chromosome; *m* is a number of weight coefficient in a chromosome.

According to the formula, all bits of the new chromosome are copied from *i*-th chromosome of the preceding generation, except for the bit with number *b* which is copied from *j*-th chromosome. This operation is implemented using for loops. After performing the operation, the new generation is formed and the transition to step 2 is done.

Step 7. The genetic algorithm is stopped and obtained values are substituted into the neural network.

The genetic algorithm block (see Figure 3) consists of the ports which receive input signals *in*1, *in*2, *in*3, the port which saves the value *zout* to be obtained as a result of training, and the output port *Nout* which shows the current value of the output after each training iteration. The signal *CLC* creates delays in one picosecond.

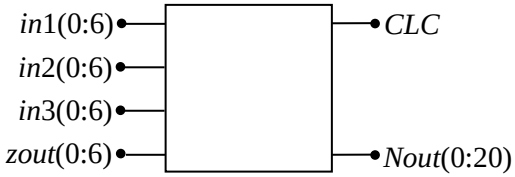

Figure 3: The genetic algorithm block

This block is defined as the following entity in SAA:

ENTITY *genVHDL* is
"Signal (*in*1) direction (*in*) of type (*integer*) and range (-100 to 100)
with initial value (63)";

```
  "Signal (in2) direction (in) of type (integer) and range (-100 to 100)
   with initial value (-82)";
  "Signal (in3) direction (in) of type (integer) and range (-100 to 100)
   with initial value (70)";
  "Signal (zout) direction (in) of type (integer) and range (0 to 100)
   with initial value (77)";
  "Signal (CLC) direction (inout) of type (bit) with initial value ('1')";
  "Signal (Nout) direction (inout) of type (integer) and range
    (-1048575 to 1048576);
END OF ENTITY.
```

The block implements operators of crossover, mutation, and selection of the next chromosome generation. The architecture of the entity contains four main processes: *neuron*, *ChrsToW*, *main*, and *Genetic*. The *neuron* process starts as soon as the value of the start signal changes its value, i.e. after the training begins. The SAA scheme of this process is the following:

```
PROCESS neuron (start) is
  DECLARATIONS (
    "Variable (LocalOut) of type (integer) and range (0 to 1023)";
    "Variable (lin) of type (integer)");
  (LocalOut := 100);
  (lin := (in1 * W(1) + in2 * W(2) + in3 * W(3)) / 16);
  FOR (a FROM 0 TO 49)
    IF (abs(lin) >= x(a)) AND (abs(lin) < x(a + 1))
    THEN (LocalOut := 50 + a); EXIT LOOP;
    END IF
  END OF LOOP;
  IF (lin < 0) THEN (LocalOut := 100 – LocalOut);
  END IF;
  IF (NOT (finishTeaching)) THEN
    (lout(i) := LocalOut);
    NFinish <= NOT NFinish;
  END IF;
  Nout <= LocalOut;
END OF PROCESS,
```

where *LocalOut* is the current neuron output, which is compared with the expected result of variable *zout*; *W*(1), *W*(2), *W*(3) are neuron weights; *i* is the number of the chromosome on the basis of which weight coefficients were formed before launching the process which forms the output of the neural network.

The process *ChrsToW* converts the value of chromosome object into values of synapse weights. The process *Genetic* performs sorting, crossover, and mutation of the chromosomes. In particular, crossover and mutation is presented by the following scheme:

```
(num3 := 0);
FOR (k FROM 1 TO 3)
  FOR (i FROM k + 1 TO 4)
    (num3 := num3 + 1);
    (chrs(num3) := chrs2(k));
    FOR (j FROM 1 TO 3)
      FOR (m FORM 1 TO 2)
        (num5 := num5 * 29 / 8 rem 1048576);
        (num2 := (num5) rem 8);
        (chrs(num3)(j)(num2) := chrs2(i)(j)(num2));
```

```
      END OF LOOP;
      FOR (m FROM 1 TO 4)
        (num5 := num5 * 29 / 8 rem 1048576);
        (num2 := abs(num5) rem 8);
        (chrs(num3)(j)(num2) := NOT(chrs(num3)(j)(num2));
      END OF LOOP;
    END OF LOOP;
  END OF LOOP;
END OF LOOP.
```

Here *chrs* and *chrs*2 are bit arrays storing weight coefficients of chromosomes.
IDS generated VHDL code based on the designed SAA schemes.

4. Experiment Results

Consider the example of training one neuron with three inputs and three weight coefficients, correspondingly. The hardware implementation of the genetic algorithm according to the proposed methodology was done on Xilinx Spartan 3 XC3S200 FPGA in Xilinx ISE Design Suite 13.2 environment and modeled using ISE Simulator (ISim). The process of training is represented in Figure 4.

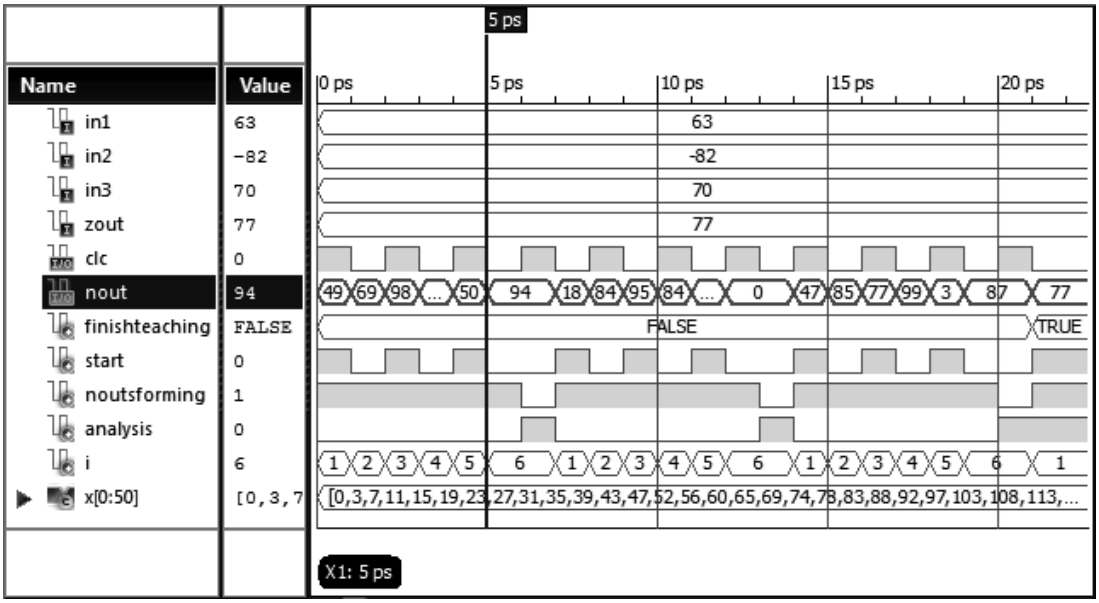

Figure 4: The process of neural network training with the genetic algorithm

Signals *in*1, *in*2, *in*3, *zout*, *clc*, *nout* were described in Section 3. The purpose of the other signals is the following:

- *finishteaching* stores the information that training of the network is finished;
- *start* launches the process, forms the output of the network for each new set of weight coefficients;
- *i* is a number of a chromosome, on the basis of which weight coefficients were formed before the launch of the process forming the network's output (before setting signal *start* to 1);
- *noutsforming* is set to 1 while chromosomes are run through the neural network. It transits from 0 to 1 when all 6 chromosomes are processed, and the process can proceed to results' analysis: finish the algorithm or form new chromosomes using crossover and mutation. The transition of the signal from 0 to 1 means the start of the new iteration of the genetic algorithm;
- *analysis* is set to 1 while the information obtained at running chromosomes through neural networks is analyzed.

The process of adjustment of weight coefficients of the neural network completed in three iterations, which is shown in Figure 4. In the ISE Simulator (ISim) environment, a step lasting 1 picosecond on each operation of the genetic algorithm is defined programmatically for adjustment and work demonstration. The time of training is 0.15 milliseconds.

Processes of tuning of weight coefficients for neural networks with 2 and 3 neurons were also modeled. The first network consisted of two sequentially connected neurons and four synapses. The time of training was 0.2 milliseconds.

The second neural network consisted of two neurons in the first (input) layer and one neuron in the second (output) layer. Neurons are connected with four synapses. The time of training was 0.23 milliseconds.

The comparison of the obtained results with equivalent results found in similar works [14, 15, 17, 18] is presented in Table 2.

Table 2

A comparison of the results obtained by other authors (T_1) and obtained in this work (T_2)

Reference to similar work	N	K	Training time T_1, ms	Training time T_2, ms	Speedup, T_1 / T_2
[14]	64	500	0,941	0,625	1,5
[15]	16	256	0,800	0,230	3,5
[17]	20	380	74,000	0,285	260
[18]	32	200	1,600	0,200	8,0

The comparisons were made with the highest similarity of parameters and implemented on the same chip. The first column shows the references to the works. The next two columns indicate the parameters of the genetic algorithm in corresponding works: N is the size of the chromosome, K is the number of epochs in the genetic algorithm. The following columns show the training time obtained in other papers, the time obtained in this work, and also respective speedup.

The results of the modeling show that the developed method of training of neural networks with a genetic algorithm at hardware implementation on FPGA allows significantly speed up the adaptation of neural network components of control systems and thus increase their efficiency.

5. Related Work

The proposed approach is related to works on the synthesis of programs from specifications [20, 21], automated generation of VHDL code [22–24], and implementation of genetic algorithms on FPGA [14–19, 25–27]. In paper [22], a Java library to read, manipulate, and write (generate) VHDL code is presented. Paper [23] describes an automatic process of converting XSG (Xilinx System Generator) specifications into efficient VHDL code. The process involves customized fixed-point hardware definition, data flow graph extraction, resource-constrained, and latency-constrained scheduling, and VHDL specification of the system. Work [24] presents a generator of high-speed input (parser) and output (deparser) network blocks from the P4 language which is designed for the description of modern packet processing devices. The tool converts a P4 description to a synthesizable VHDL code suitable for the FPGA implementation.

The main difference of our approach from the mentioned works is that it uses algebraic specifications, based on Glushkov's algebra of algorithms. Specifications are represented in a natural linguistic form simplifying the understanding of algorithms and facilitating the achievement of demanded software quality. Another advantage of our tools is the method of automated design of syntactically correct algorithm specifications, which eliminates syntax errors during the construction of algorithm schemes.

Implementations of genetic algorithms on FPGAs are considered in works [14–19, 25–27]. Paper [14] proposes the modular realization of a genetic algorithm. However, the implementation does not use a parallel processing strategy and uses several loops for each generation. In each generation, it is necessary to read and write the generation from/to the memory. Implementations

considered in [15, 16] are applications of genetic algorithms in systems of digital signal processing and control built into FPGA. Paper [15] presents a real-time genetic algorithm for adaptive filtering program with all modules implemented in hardware, such as fitness function, selection, crossover, mutation, and random number generator functions. The speed of 320 thousand generations per second was reached. Paper [16] proposes a genetic algorithm for dynamic systems based on blocks of filters. Several approaches of high-speed general-purpose hardware for accelerating genetic algorithms are proposed in [17–19]. These approaches improve the configuration of parameters of genetic algorithms in hardware architecture, but decrease the parallelization of hardware and reduce the high-speed performance of a genetic algorithm. Paper [25] proposes a genetic algorithm for sequential and parallel pipeline solutions on FPGA using Verilog HDL, which is applied for solving travelling salesman problem (TSP). Paper [26] presents a hardware implementation of the crossover module in the genetic algorithm for TSP. A combination of pipelining and parallelization with a genetic algorithm processor to improve processing speed is employed. Work [27] describes a parallel implementation of a genetic algorithm on FPGA which can optimize a wide range of functions in a viable time for critical applications that require short time constraints or a large amount of data to be processed in a short interval. Articles [25–27] are aimed at implementing the classical genetic algorithm and solving problems of finding the extremum of a function. Our work differs from them by using the SAA-based toolkit for automated design of such hardware. The hardware is implemented according to the method developed in our previous works [10, 12] to optimize the weights of neural networks using a genetic algorithm.

6. Conclusion

The paper proposes the method and software tools for automated design and synthesis of parallel programs for field-programmable gate arrays based on the algebra-algorithmic approach. The developed facilities provide the construction of parallel algorithm schemes by superposition of language constructs of Glushkov's system of algorithmic algebra. Based on the schemes, the corresponding source code in VHDL is automatically generated, which is further executed on an FPGA. The particular feature of the approach consists in using high-level algebra-algorithmic program specifications represented in a natural linguistic form. The specifications are the basis for the automatic generation of source code in a programming language. The approach is illustrated on developing a genetic algorithm applied at the training of multilayer neural networks. The experiment results showed that with the developed genetic algorithm implemented on FPGA, neural network training is significantly faster than in related works.

References

[1] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability and Scalability, Wiley-Interscience, Hoboken, NJ, 2006.

[2] S. Palnitkar, Verilog HDL: A Guide to Digital Design and Synthesis, Prentice Hall PTR, Upper Saddle River, NJ, 2003.

[3] S. Sutherland, S. Davidmann, P. Flake, SystemVerilog for Design: A Guide to Using Systemverilog for Hardware Design and Modeling, Springer-Verlag, Berlin, 2006.

[4] A. Doroshenko, O. Yatsenko, Formal and Adaptive Methods for Automation of Parallel Programs Construction: Emerging Research and Opportunities, IGI Global, Hershey, 2021.

[5] P. I. Andon, A. Yu. Doroshenko, K. A. Zhereb, O. A. Yatsenko, Algebra-Algorithmic Models and Methods of Parallel Programming, Akademperiodyka, Kyiv, 2018.

[6] A. Doroshenko, K. Zhereb, O. Yatsenko, Developing and optimizing parallel programs with algebra-algorithmic and term rewriting tools, in: V. Ermolayev, H. C. Mayr, M. Nikitchenko, A. Spivakovsky, G. Zholtkevych (Eds.), ICTERI 2013, volume 412 of Communications in Computer and Information Science, Springer-Verlag, Cham, 2013, pp. 70–92. doi:10.1007/978-3-319-03998-5_5.

[7] A. Doroshenko, O. Beketov, M. Bondarenko, O. Yatsenko, Automated design of parallel programs for heterogeneous platforms using algebra-algorithmic tools, in: V. Ermolayev,

F. Mallet, V. Yakovyna, H. Mayr, A. Spivakovsky (Eds.), ICTERI 2019, volume 1175 of Communications in Computer and Information Science, Springer-Verlag, Cham, 2020, pp. 3–23. doi:10.1007/978-3-030-39459-2_1.
[8] E. Dumesnil, P.-O. Beaulieu, M. Boukadoum, Fully parallel FPGA implementation of an artificial neural network tuned by genetic algorithm, in: Proceedings of the 16th. IEEE International New Circuits and Systems Conference, NEWCAS 2018, IEEE, New York, NY, 2018, pp. 365–369. doi:10.1109/ NEWCAS.2018.8585580.
[9] M. F. Torquato, M. A. C. Fernandes, High-performance parallel implementation of genetic algorithm on FPGA 38 (2019) 4014–4039. doi:10.1007/s00034-019-01037-w.
[10] P. I. Kravets, V. M. Shymkovych, A method for optimizing the weighting coefficients of neural networks using a genetic algorithm when implemented on programmable logic integrated circuits, Èlektron. model. 35.3 (2013) 65–74.
[11] P. I. Kravets, V. N. Shimkovich, D. A. Ferens, Method and algorithms of implementation on PLIS the activation function for artificial neuron chains, Èlektron. model. 37.4 (2015) 63–74.
[12] V. Symkovych, P. Kravets, Hardware implementation neural network controller on FPGA for stability ball on the platform, in: Z. Hu, S. Petoukhov, I. Dychka, M. He (Eds.), Proceedings of the 2nd. International Conference on Computer Science, Engineering and Education Applications, volume 938 of ICCSEEA 2019, Springer Nature, Cham, 2020, pp. 247–256. doi:10.1007/978-3-030-16621-2_23.
[13] V. Shymkovych, S. Telenyk, P. Kravets, Hardware implementation of radial-basis neural networks with Gaussian activation functions on FPGA, Neural Comput. & Applic. 33 (2021) 9467–9479. doi:10.1007/s00521-021-05706-3.
[14] F. Mengxu, T. Bin, FPGA implementation of an adaptive genetic algorithm, in: Proceedings of the 12th International Conference on Service Systems and Service Management, ICSSSM 2015, IEEE, New York, NY, 2015, pp. 1–5. doi:10.1109/ICSSSM.2015.7170318.
[15] H. Merabti, D. Massicotte, Hardware implementation of a real-time genetic algorithm for adaptive filtering applications, in: Proceedings of the 27th Canadian Conference on Electrical and Computer Engineering, CCECE 2014, IEEE, New York, NY, 2014, pp. 1–5. doi:10.1109/CCECE.2014.6901026.
[16] N. Sehatbakhsh, M. Aliasgari, S. M. Fakhraie, FPGA implementation of genetic algorithm for dynamic filter-bank-based multicarrier systems, in: Proceedings of the 8th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2013, IEEE, New York, NY, 2013, pp. 72–77. doi:10.1109/DTIS.2013.6527781.
[17] M. S. B. Ameur, A. Sakly, FPGA based hardware implementation of Bat algorithm, Appl. Soft Comput. 58 (2017) 378–387. doi: 10.1016/j.asoc.2017.04.015.
[18] L. Guo, A. I. Funie, Z. Xie, D. Thomas, W. Luk, A general-purpose framework for FPGA-accelerated genetic algorithms, Int. J. Bio-Inspir. Comput. 7.6 (2015) 361–375. doi:10.1504/IJBIC.2015.073183.
[19] M. Peker, A fully customizable hardware implementation for general purpose genetic algorithms, Appl. Soft Comput. 62 (2018) 1066–1076. doi:10.1016/j.asoc.2017.09.044.
[20] P. Flener, Achievements and prospects of program synthesis, in: A. C. Kakas, F. Sadri (Eds.), Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, volume 2407 of Lecture Notes in Artificial Intelligence, Springer-Verlag, London, 2002, pp. 310–346.doi:10.1007/3-540-45628-7_13.
[21] S. Gulwani, Dimensions in program synthesis, in: Proceedings of the 12th. International ACM SIGPLAN symposium on Principles and practice of declarative programming, PPDP ’10, ACM, New York, NY, 2010, pp. 13–24. doi:10.1145/1836089.1836091.
[22] C. Pohl, C. Paiz, M. Porrmann, vMAGIC — automatic code generation for VHDL, International Journal of Reconfigurable Computing 2009 (2009) 1–9. doi:10.1155/2009/205149.
[23] P. Martín, E. Bueno, Fco. J. Rodríguez, O. Machado, B. Vuksanovic, An FPGA-based approach to the automatic generation of VHDL code for industrial control systems applications: a case study of MSOGIs implementation, Mathematics and Computers in Simulation 91 (2013) 178–192. doi:10.1016/j.matcom.2012.07.004.

[24] P. Benáček, V. Puš, H. Kubátová, T. Čejka, P4-To-VHDL: automatic generation of high-speed input and output network blocks, Microprocessors and Microsystems 56 (2018) 22–33. doi:10.1016/j.micpro.2017.10.012.
[25] X. Sun, J. Li, F. Tian, Y. Chen, J. Yang, Design of FPGA hardware based on genetic algorithm, in: Proceedings of the 3rd. International Conference on Computer Engineering, Information Science & Application Technology, ICCIA 2019, volume 90 of Advances in Computer Science Research, Atlantis Press, Dordrecht, 2019, pp. 102–108. doi:https:10.2991/iccia-19.2019.15.
[26] N. Attarmoghaddam, K. F. Li, A. Kanan, FPGA implementation of crossover module of genetic algorithm, Information 10.6 (2019) 1–11. doi:10.3390/info10060184.
[27] M. F. Torquato, M. A. C. Fernandes, High-performance parallel implementation of genetic algorithm on FPGA, Circuits, Systems, and Signal Processing 38 (2019) 4014–4039. doi:10.1007/s00034-019-01037-w.