FPGA Implementations of Feed Forward Neural Network by using Floating Point Hardware Accelerators

Gabriele-Maria LOZITO, Antonino LAUDANI, Francesco RIGANTI-FULGINEI, Alessandro SALVINI

Department of Engineering, Roma Tre University, via Vito Volterra 62, 00146 Roma, Italy

gabrielemaria.lozito@uniroma3.it, alaudani@uniroma3.it, riganti@uniroma3.it,asalvini@uniroma3.it

Abstract. *This paper documents the research towards the analysis of different solutions to implement a Neural Network architecture on a FPGA design by using floating point accelerators. In particular, two different implementations are investigated: a high level solution to create a neural network on a soft processor design, with different strategies for enhancing the performance of the process; a low level solution, achieved by a cascade of floating point arithmetic elements. Comparisons of the achieved performance in terms of both time consumptions and FPGA resources employed for the architectures are presented.*

Keywords

Embedded floating point, FPGA, neural networks, soft-core processor, VHDL.

1. Introduction

Field Programmable Gate Arrays (FPGA) designs are very common in the field of computational electronics [1], [2], [3]. Digital Signal Processing (DSP) models, often analyzed in high level environments, show heavy restraints on performance once implemented on embedded systems whose bottleneck is, despite the ongoing advances in Floating Point Units (FPU) development, the low floating point operations per second (FLOPS), [4]. Compared to a microcontroller implementation (based on the sequential execution of instructions by the CPU) the nature of an FPGA design exploits the concepts of customization and parallelization to enhance the throughput of a computational system [5]. Customization allows the designer to create, through Hardware Description Language (HDL), the internal architecture of the system down to Register Transfer Level (RTL), defining as a matter of fact a flexible Application Specific Integrated Circuit (ASIC). Parallelization spreads modular and sequential algorithms on a parallel interface, improving the throughput of complex algorithms by a multiplicative factor [6].

Neural Networks in embedded systems are frequently implemented on microcontroller units [7], [8]. A neural network implementation on a microcontroller, even when built with simple integer arithmetic, lacks the performance enhancement of a parallel design [9]. The choice of implementing a neural network architecture on FPGA benefits from customization and parallelization in different ways.

Very large Feed Forward Neural Networks (FFNN), especially if designed to work with floating point (FP) precision, performs a large number of elementary products and sums. Moreover, for each neuron of FFNN within the hidden layers, a non-linear function computation is required to determine the activation value of the neuron. Without dedicated FP hardware such computations can hinder the whole performance of the system, hence making the design difficult to be used in critical applications like real-time control systems [10].

In literature different approaches have been followed to reduce the computational cost of this particular activation function, using piecewise linear interpolation [11], polynomial fitting techniques [12], [13], [14], [15], enhanced computational algorithms [16], [17] and Look-Up Tables [18], [19], [20], [20], [21]. In this way, customization allows the designer to implement blocks inside the FPGA to speed up the calculus of FP operations.

The concept of parallelization is implicit in the high performance of the solutions explained above: a RTL-defined LUT can compute an arbitrarily complex operation in few clock cycles, assuming the memory of the system can contain the values. The same can be said for the arithmetic units, which can exploit powerful pipelines to speed up the calculus. The number of interconnection between the neurons, however, grows exponentially with the size (in terms of input and outputs) of the network. It's possible to reduce the complexity of the FFNN by splitting a Multiple Input Multiple Output (MIMO) FFNN into a smaller and simpler Single Input Single Output (SISO) FFNN that can be easily processed in parallel by means of multivariate function decomposition [22], [23].

2. The Feed Forward Neural Network

The Feed Forward Neural Network implemented in this paper is a SISO Feed Forward Neural Network, composed by a single hidden layer of 10 neurons with a non-linear activation function Logsig (Eq. 1) and\or Tansig (Eq. 2):

$$act = \frac{1}{1+e^{-nst}}, \tag{1}$$

$$act = \frac{2}{1+e^{-2nst}} - 1. \tag{2}$$

This architecture was chosen for the easiness of the training process and the modularity of the structure: indeed it is possible to face MIMO problems by using SISO FFNN as described in [22]. The FFNN was created and trained in Matlab® environment. The normalization of the inputs and outputs was disabled and the activation function of the output layer was a pure linear function.

3. Implementation on Nios II/f Soft Processor

The first solution attempted to implement the network on FPGA makes usage of the soft core processor Nios II/f, released by Altera® as a crypted core. This core can be synthesized with as low as 1600 logic elements (LE) and supports a maximum frequency of 140 MHz [24], [25].

After synthesis and programming on the FPGA device, the soft core itself can be programmed and debugged in C using a JTAG tool chain running inside an Eclipse environment. This soft core processor supports hardware integer multiplication and division, and up to 255 custom instructions definable by the designer. These custom instructions can be defined at RTL level using Very High Speed Integrated Circuits (VHSIC) Hardware Description Language (VHDL) or Verilog® code, and are synthesized as parallel blocks of the internal Nios II Arithmetic Logic Unit (ALU) as shown in Fig. 1, when a custom instruction is called from the instruction memory of the Nios II, the operands are transferred in the custom logic and, according to the type of custom instruction (combinatorial or sequential) the result is collected after a definite number of clock cycles [26].

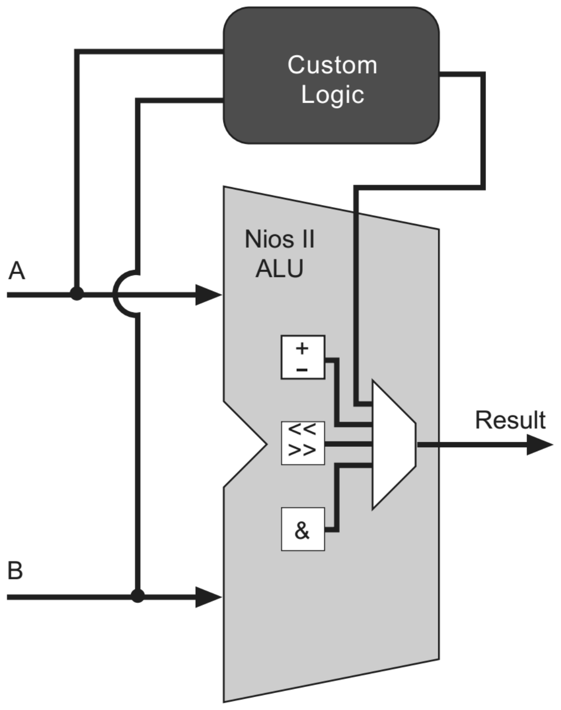

Fig. 1: Implementation of custom logic in the Nios II ALU.

3.1. Overall System Description

The design proposed in this section is based on the Nios II/f core, modified to have a Floating Point ALU and two system works with a 100MHz clock, which is replicated by means of a PLL with a phase shift of -3 ns to control an external 8 Mb SDRAM [27]. As shown in Fig. 2, the processor was equipped with a standard JTAG interface for programming and a Performance Counter to determine the execution time of the implemented code. The Floating Point ALU was the standard block from the library released by Altera® as a part of the Quartus II® environment. Two Activation Function LUT(s) were created in VHDL (one for the Tansig and one for the Logsig) and imported into the design as user-made custom instructions.

3.2. LUT(s) Use for Computing Activation Function

The main performance bottleneck for neural networks using floating point arithmetic lies in the activation function computation for the hidden layer. Computing

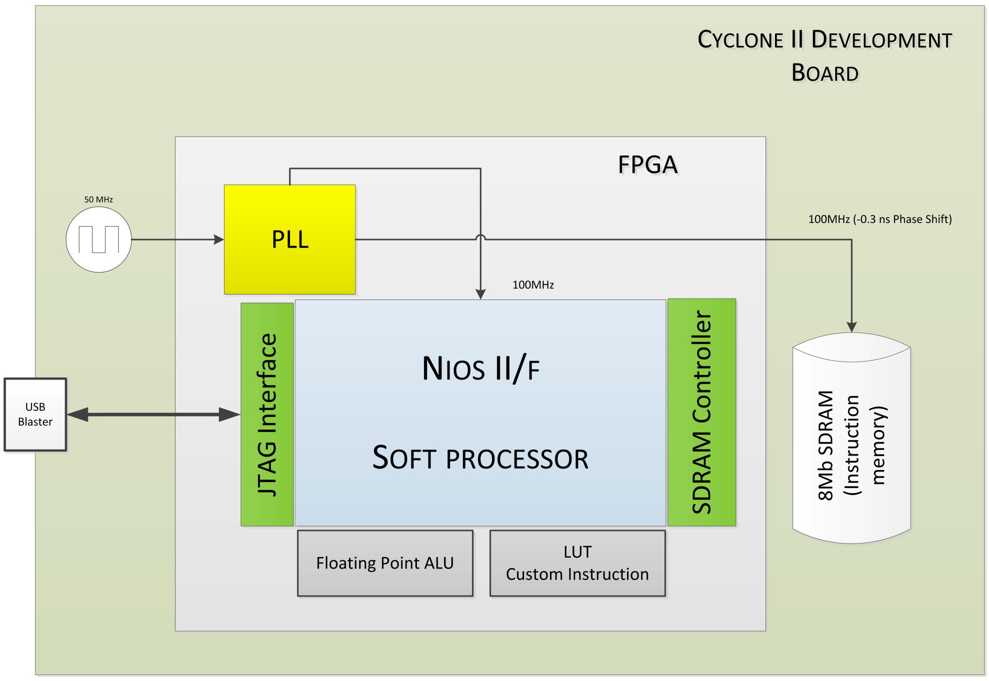

Fig. 2: Soft processor and Peripherals.

this function using "full precision" software functions is often too slow for time critical applications [28]. Instead of calculating the activation function, an alternative solution is to sample it, loading the obtained values in a LUT [18], [19], [20], [21]. In the present paper, the function was not sampled with a uniform and constant spacing between the sampling points. This is because the activation function assumes almost constant values near the saturation points, making it wasteful to choose a fine sampling in their proximity. On the other hand, near the origin, the slope of the function is very high, and a finer sampling may help in reducing quantization error. In [19] only two kinds of spacing are used: a fine one, near the origin, and a wide one, near the saturation branches. In this work, a different approach is proposed: the distance between a sample and the following one is inversely proportional to the slope of the function in the sampled point.

This yields a finer sampling near the origin, gradually getting wider near the saturation points. The Logsig function was sampled with 256 values between -16 and $+16$, while the Tansig, being an odd function, was sampled for positive arguments only, with 256 values between 0,2 and. Using these values, a VHDL combinatorial code was written and simulated in Altera ModelSim environment for RTL analysis.

The implemented block has a single floating point input, that is split in sign, exponent and mantissa. Through the use of a suitable IF-THEN-ELSE chain the input value addresses a specific entry in the LUT, that is propagated as output. If the input value magnitude is bigger than the saturation values, a suitable constant value is propagated as output. Since the Tansig, near the origin, can be approximated to the bisector of the first quadrant, values smaller than 0,2 are directly propagated in output (thus approximating the function linearly). The synthesis result of this IF-THEN-ELSE structure is a very long chain of comparators. Propagation of the signal through this chain can be long, so a tunable delay of 4 clock cycles was introduced to ensure result stability (the delay is controlled by a simple counter that can be modified to suit the size of the LUT).

3.3. Polynomial Fitting

The basic operations of floating point math are greatly fastened by the presence of a Floating Point ALU (about 10 times faster [29]). Thus, other than speeding up the Multiplier-Accumulation part of the FFNN, this hardware module can be used to compute a polynomial approximation of the activation function. A group of second-degree polynomials was chosen to fit the activation functions. The coefficients of the polynomials were determined in Matlab® environment through the use of the Curve Fitting Tool. Both the functions were fitted only for positive arguments.

For the Logsig polynomial fitting, a function (denoted as 5PY-L) composed by the superposition of 5 second-degree polynomials, has been implemented. Even if the Logsig function is not odd, a partial symmetry is present. This was exploited for its negative arguments: first, the value of the function is calculated

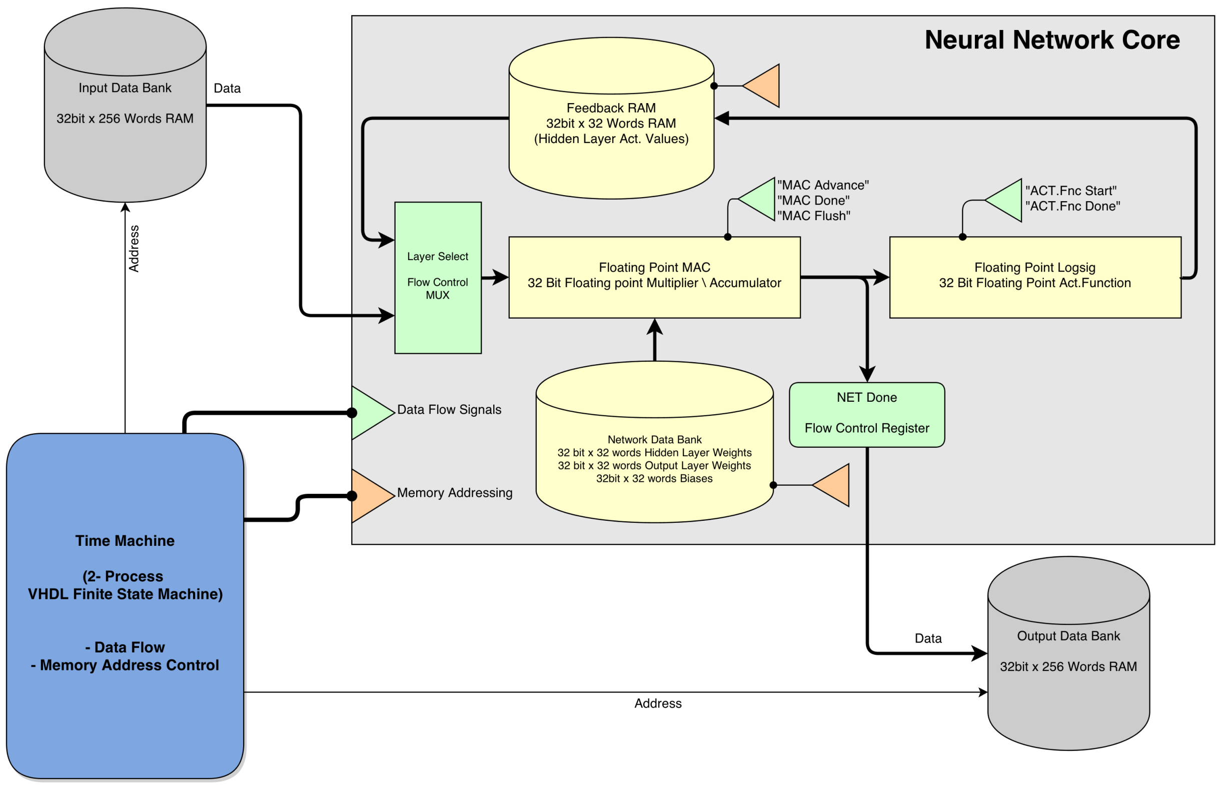

Fig. 3: NN Core schematic diagram.

Tab. 1: Nios II/f test results on FFNN with Logsig activation functions.

Function	MSE	Average time/sample
Floating Point	0.0000 (ref)	650 µs
LUT (Logsig)	0.1598	17.5 µs
5PY-L	0.0075	185 µs

Tab. 2: Nios II/f test results on FFNN with Tansig activation functions.

Function	MSE	Average time/sample
Floating Point	0.0000 (ref)	715 µs
LUT (Tansig)	0.0053	17.5 µs
4PY-L	0.0039	142 µs
5PY-L	0.0018	174 µs

considering the absolute value of the input; then, if the input is negative, the calculated value is subtracted by the value of 1. For the Tansig polynomial fitting, two functions, composed by 4 and 5 second-degree polynomials have been implemented, respectively denoted as 4PY-T and 5PY-T. This time, since the Tansig is an odd function, the argument is considered in absolute value, and the sign is directly propagated to the output.

3.4. Test Results and Considerations

The design was used to simulate a FFNN trained on the function $y = x^2$, and was tested on a vector of 2048 linearly spaced inputs between -5 and $+5$. The results in Tab. 1 and Tab. 2 show the performance in terms of mean squared error (MSE) and execution time of the different solutions proposed above. As a reference for execution time, the performance of a FFNN featuring a full precision software implementation of the activation function is shown in both tables.

4. NN Core Implementation

In the following part of this paper a solution based on low level architecture is presented. The proposed design was used for the implementation of the same FFNN previously described.

4.1. Overall System Description

The proposed design is an arithmetic core composed (see Fig. 3) by high performance floating point arithmetic blocks developed by Altera®, whose data flow is controlled by a Finite States Machine (FSM) written in VHDL. The arithmetic core is composed by 3 blocks: a multiplier-accumulator (MAC), an activation function,

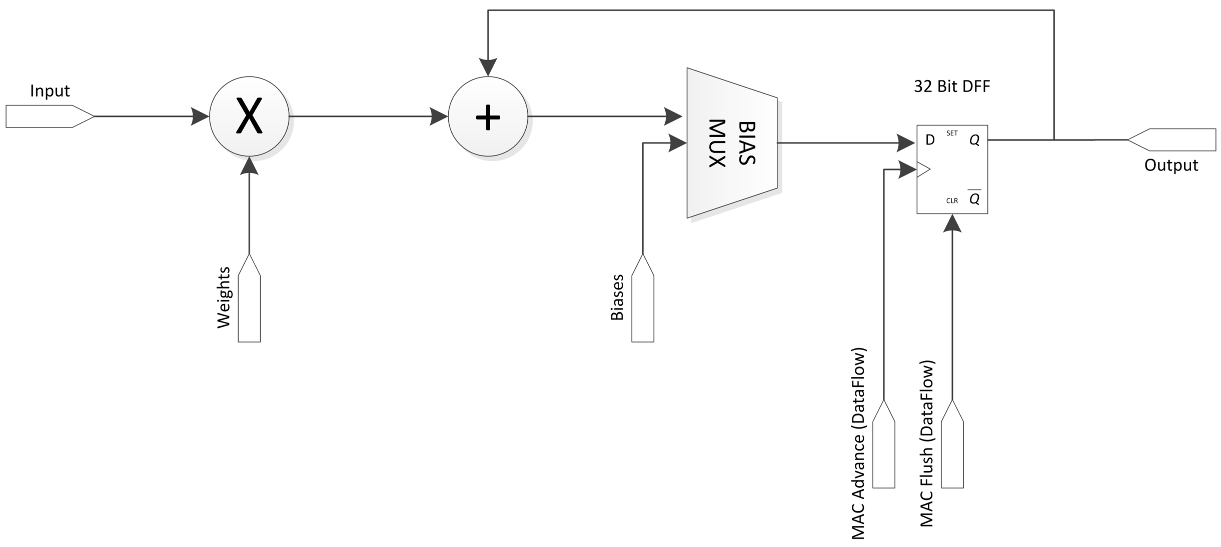

Fig. 4: MAC block diagram.

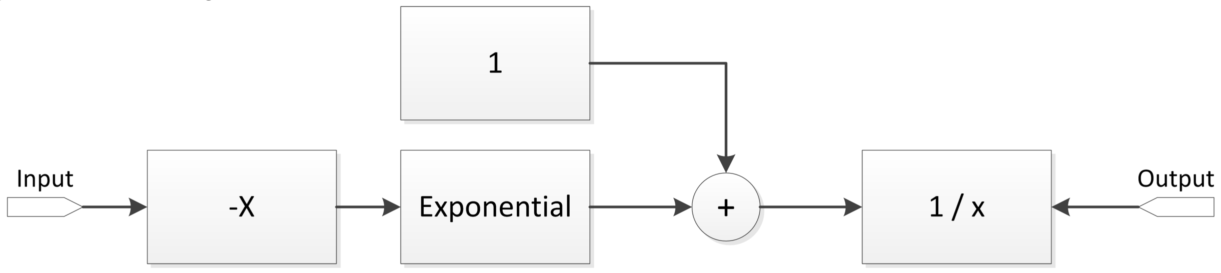

Fig. 5: Logsig block diagram.

and a feedback RAM. These three blocks constitute a suitable base to build a Neural Network [30]. The first block computes, for each neuron, the weighted sum of the inputs.

The second block has the results of the first block as inputs, and computes the activation values for the hidden layer. The third block, receiving the output from the activation function block, stores the values from the hidden layer. These values are then sent through a MUX back into the MAC block for the output layer computation. Both input and output data of the FFNN are stored in RAM blocks that are accessible through JTAG interface using the Quartus II® software. The whole Core and the data banks are controlled by a free running 2-Process Finite State Machine "Time Machine" using data flow control signals and address registers. Internal data flow of the core is regulated by a number of 32-bit wide MUXes and D-Type Flip Flops (DFFs). The design was implemented on a EP2C20F484C7 Cyclone II FPGA mounted on a DE2 – Development Board. After synthesis and fitting the full design occupied about 5000 logic elements (LE) and all the 52 hardware multipliers present on the FPGA.

4.2. Data Flow of the Arithmetic Core

The computation of the arithmetic core begins by loading the first sample from the Input Data Bank into the MAC block. The core contains into its internal memory the weights and biases of the FFNN. This memory is addressed directly by the Time Machine control block. Since the MAC is computing the hidden layer, each neuron will have a bias value that must be added to the weighted input. This bias value is preloaded into the 32-bit DFF accumulator using the Bias MUX. Inputs and weights are multiplied and the results are added to the preloaded bias (see Fig. 4). Since the hidden layer has only one input, the MAC is done for the first neuron, and the result is propagated to the next block, where the activation function is computed. In this section, a logical not is operated on the MSB of the input, changing its sign. The result is sent to an exponential arithmetic block whose output is connected to an adder that sums the result to the constant value of 1.

The result is then inverted and the activation value of the first neuron is finally written in the Feedback RAM. This operation is repeated for the 10 neurons, filling the RAM with the activation values of the hidden layer. Then, the Time Machine switches the Layer

Tab. 3: Best performance comparison.

Function	MSE	Average time/sample	Full Time (2048 Samples)
NN Core (50 MHz clock)	0.0000 (ref)	154 µs	315.4 ms
NN Core (100 MHz clock)	0.0000	78 µs	159.8 ms
5PY-L	0.0075	185 µs	378.8 ms
LUT (Tansig)	0.0054	17.5 µs	35.79 ms

Tab. 4: Nios II/f design main resources usage by entity.

Entity	LC Comb.	LC Reg.	DSP Elements
Nios II CPU	2382	1799	4
Floating Point Unit	5125	3783	7
LUT (Tansig)	1815	4	0
LUT (Logsig)	1617	4	0

Select MUX so that the MAC block is now connected to the Feedback RAM. The bias of the output neuron is preloaded in the accumulator, and the MAC computes the weighted sum of all the activation values from the hidden layer. This is the output result of the network, and is saved in the Output Data Bank.

4.3. Time Machine FSM

Data processing from input to output needs to be managed by some sort of control block, responsible for synchronizing the dataflow and, were needed, perform memory addressing. In a traditional programming language, like C, a popular approach to create such controller is to use a finite state machine (FSM). In its simplest form, a FSM is a set of code blocks, each identifying a particular function (e.g. "load data from RAM", "sum input A and input B", "transpose array C"), inside a switch/case structure. If the FSM is the sole controller of the system, the switch/case structure is confined in an endless loop. The variable controlling the switch is updated at the end of each code block, ensuring that every time the switch/case is evaluated the FSM will execute a specific code block (i.e. will be in a known and definite state). This rather simple approach is not as straightforward in HDL languages, since the code is not executed by a processor, thus not inherently sequential.

Hardware, emulating the processor sequential behaviour, must be created. A possible approach, proposed in [31], is to create an instruction counter whose value is increased at every clock edge. By using a net of comparators, when a particular value is assumed by the instruction counter, specific logic functions (states) are executed. Creating the FSM in this way grant an important advantage: since the instruction counter is updated on clock edge, the FSM can work synchronously with the other elements in the design. This is very important when some blocks in the design have definite input-output delays, since the FSM can be programmed to remain in a "wait" state until the output is ready to be propagated to the next block. In VHDL this architecture can be defined by the use of two code blocks (processes), one sequential and one combinatorial.

The first one is responsible for the instruction counter increase at every clock edge, and is synthesized with a counter register. The second one is responsible for decoding the instruction counter into actual logic signals, and is synthesized with a network of comparators. The cycle of operations performed by the FSM is obviously limited, once the last operation is performed (i.e. the last output value has been loaded in the Output Data Bank), the FSM will reset and start over. With a 50 MHz clock, the computation of a single sample takes about 150 µs.

5. Solutions Comparison

In the Tab. 3, a comparison of the best performances among solutions is presented. At full precision, the NN Core design provides a quite lower computation time than the Nios II design. Moreover, by doubling the clock frequency through a PLL (thus using the same frequency used for the Nios II designs, 100 MHz) the computation time drops at 78 µs/sample. However, if full precision is not needed (and the choice of a particular activation function is not mandatory), implementing a FFNN based on a Tansig activation function yields the lowest computation time, using the Nios II design. In particular, implementing a LUT yields the best results in terms of precision over computation time.

In Tab. 4 and Tab. 5 the resources, in terms of dedicated Combinatorial and Register logics (LC Comb. and LC Reg.) are shown. The high level solution is expensive in terms of resources usage, peaking with 15 098 logic elements (LE) if both the LUT(s) are im-

Tab. 5: Best performance comparison.

Entity	LC Comb.	LC Reg.	DSP Elements
MAC Block	1015	620	7
Tansig Block	2784	1874	45
FSM Block	205	130	0

plemented as custom instructions. This is generally not necessary, since only one of the activation functions is used in the network. By excluding the Logsig LUT from the synthesis the LE usage drops to 12 699 LE. The low level solution, although completely saturating the DSP blocks of the FPGA, is contained in 5 037 LE.

6. Conclusions and Future Works

Two possible designs to implement a neural network in a FPGA environment were presented. The first design, taking advantage of the Nios II soft processor, used hardware accelerators to speed up both the calculus of the elementary products of neurons and the computation of the nonlinear activation functions for the hidden layer. By exploiting the soft processor hardware acceleration for floating point operations, an alternative polynomial approximation for the activation functions was implemented and tested for performance.

The second design proposed is composed by a chain of arithmetic units timed and coordinated by a VHDL state machine, which implemented a full precision floating point computation at a fraction of the execution time. The results acquired from this work can advance into a new form of neural network implementation on FPGA. The low level arithmetic chain implemented in the NN Core design could be split and included inside two custom instructions of a soft processor, hence combining the speed of the low level design with the flexibility of a C-programmable environment. This could benefit the design by allowing the inclusion of standard interfaces (like JTAG or I2C) to the system useful for many applications (see for example [32], [33]), while retaining RTL-wise control of the data flow.

In the hypothesis of using the network as a form of DSP for smart sensor or control systems, the floating point precision could be traded for a faster and smaller fixed-point or integer based system [34], [35]. Moreover, an improvement of the whole system can be always obtained if more complex and robust optimization algorithms [36], [37] are used in order to reduce the size of the implemented Neural Networks.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 218086.

References

[1] YAJUAN CH. and Q. WU. Design and implementation of PID controller based on FPGA and genetic algorithm. In: *Proceedings of 2011 International Conference on Electronics and Optoelectronics*. Dalian: IEEE, 2011, pp. 308–311. ISBN 978-1-61284-275-2. DOI: 10.1109/ICEOE.2011.6013491

[2] ZHENBIN G., X. ZENG, J. WANG and J. LIU. FPGA implementation of adaptive IIR filters with particle swarm optimization algorithm. In: *11th IEEE Singapore International Conference on Communication Systems*. Guangzhou: IEEE, 2008, pp. 1364–1367. ISBN 978-1-4244-2424-5. DOI: 10.1109/ICCS.2008.4737406.

[3] OTSUKA, T., T. AOKI, E. HOSOYA and A. ONOZAWA. An Image Recognition System for Multiple Video Inputs over a Multi-FPGA System. In: *IEEE 6th International Symposium on Embedded Multicore SoCs*. Aizu-Wakamatsu: IEEE, 2012, pp. 1–7. ISBN 978-0-7695-4800-5. DOI: 10.1109/MCSoC.2012.33.

[4] RAMAKRISHNAN, A. a J. M. CONRAD. Analysis of floating point operations in microcontrollers. In: *Proceedings of IEEE Southeastcon*. Nashville: IEEE, 2011, pp. 97–100. ISBN: 978-1-61284-739-9. DOI: 10.1109/SECON.2011.5752913.

[5] UNDERWOOD, K. FPGAs vs. CPUs. In: *Proceeding of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. New York: ACM Press, 2004, pp. 171–180. ISBN 1-58113-829-6. DOI: 10.1145/968280.968305.

[6] DELORIMIER, M. *Floating-point sparse matrix-vector multiply for FPGAs*. California, 2005. Master's thesis. California Institute of Technology. Research Advisor Andre DeHon.

[7] ELKATTAN, M., A. SALEM, F. SOLIMAN, A. KAMEL and H. EL-HENNAWY. Microcontroller based neural network for landmine detection using magnetic gradient data. In: *4th International Conference on Intelligent and Advanced Systems*. Kuala Lumpur: IEEE, 2012, pp. 46–50. ISBN 978-1-4577-1968-4. DOI: 10.1109/ICIAS.2012.6306156.

[8] BAYINDIR, R. and A. GORGUN. Hardware Implementation of a Real-Time Neural Network Controller Set for Reactive Power Compensation Systems. In: *Ninth International Conference on Machine Learning and Applications*. Washington: IEEE, 2010, pp. 699–703. ISBN 978-1-4244-9211-4. DOI: 10.1109/ICMLA.2010.107.

[9] PEDRONI, V. A. *Circuit design with VHDL*. Massachusetts: MIT Press, 2004. ISBN 02-621-6224-5.

[10] GHARIANI, M., M. W. KHARRAT, N. MASMOUDI and L. KAMOUN. Electronic implementation of a neural observer in FPGA technology application to the control of electric vehicle. In: *The 16th International Conference on Microelectronics*. Tunis: IEEE, 2004, pp. 450–455. ISBN 0-7803-8656-6. DOI: 10.1109/ICM.2004.1434611.

[11] AYALA, J. L., A. G. LOMENA, M. LOPEZ-VALLEJO and A. FERNANDEZ. Design of a pipelined hardware architecture for real-time neural network computations. In: *45th Midwest Symposium on Circuits and Systems*. MWSCAS-2002. Tulsa: IEEE, 2002. ISBN 0-7803-7523-8. DOI: 10.1109/MWSCAS.2002.1187247.

[12] ZURAIQI, E. A., M. JOLER and C. G. CHRISTODOULOU. Neural networks FPGA controller for reconfigurable antennas. In: *IEEE Antennas and Propagation Society International Symposium*. Toronto: IEEE, 2010, pp. 1–4. ISBN 978-1-4244-4967-5. DOI: 10.1109/APS.2010.5561011.

[13] BAPTISTA, D. a F. MORGADO-DIAS. Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks. *Neural Computing and Applications*. 2013, vol. 23, iss. 3, pp. 601–607. ISSN 0941-0643. DOI: 10.1007/s00521-013-1407-x.

[14] NASCIMENTO, I., R. JARDIM a F. MORGADO-DIAS. A new solution to the hyperbolic tangent implementation in hardware: polynomial modeling of the fractional exponential part. *Neural Computing and Applications*. 2013, vol. 23, iss. 2, pp. 363–369. ISSN 1433-3058 DOI: 10.1007/s00521-012-0919-0.

[15] SOARES, A. M., L. C. LEITE, J. O. P. PINTO, L. E. B. DA SILVA, B. K. BOSE and M. E. ROMERO. Field Programmable Gate Array (FPGA) Based Neural Network Implementation of Stator Flux Oriented Vector Control of Induction Motor Drive. In: *IEEE International Conference on Industrial Technology*. Mumbai: IEEE, 2006, pp. 31–34. ISBN 1-4244-0726-5. DOI: 10.1109/ICIT.2006.372352.

[16] CHEN, X., G. WANG, W. ZHOU, S. CHANG and S. SUN. Efficient Sigmoid Function for Neural Networks Based FPGA Design. In: *International Conference on Intelligent Computing*. Kunming: Springer, 2006, pp. 672–677. ISBN 978-3-540-37271-4. DOI: 10.1007/11816157_80.

[17] FERREIRA, P., P. RIBEIRO, A. ANTUNES and F. M. DIAS. A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function. *Neurocomputing*. 2007, vol. 71, iss. 1-3, pp. 71–77. ISSN 0925-2312. DOI: 10.1016/j.neucom.2006.11.028.

[18] PRADO, R. N. A., J. D. MELO, J. A. N. OLIVEIRA and A. D. DORIA NETO. FPGA based implementation of a Fuzzy Neural Network modular architecture for embedded systems. In: *International Joint Conference on Neural Networks*. Brisbane: IEEE, 2012, pp. 1–7. ISBN 978-1-4673-1489-3. DOI: 10.1109/IJCNN.2012.6252447.

[19] SANTOS, P., D. OUELLET-POULIN, D. SHAPIRO and M. BOLIC. Artificial neural network acceleration on FPGA using custom instruction. In: *24th Canadian Conference on Electrical and Computer Engineering*. Niagara Falls: IEEE, 2011, pp. 000450–000455. ISBN 978-1-4244-9787-4. DOI: 10.1109/CCECE.2011.6030491.

[20] HIMAVATHI, S., D. ANITHA and A. MUTHURAMALINGAM. Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization. *Transactions on Neural Networks*. 2007, vol. 18, iss. 3, pp. 880-.888. ISSN 1045-9227. DOI: 10.1109/TNN.2007.891626.

[21] BAPTISTA, D. and F. MORGADO-DIAS. On the Implementation of Different Hyperbolic Tangent Solutions in FPGA. In: 10^{th} *Portuguese Conference on Automatic Control*. Funchal: Controlo, 2012, pp. 204–209.

[22] RIGANTI-FULGINEI, F., A. SALVINI and M. PARODI. Learning optimization of neural networks used for MIMO applications based on multivariate functions decomposition. *Inverse Problems in Science and Engineering*. 2012,

vol. 20, iss. 1, pp. 29–39. ISSN 1741-5977. DOI: 10.1080/17415977.2011.629047.

[23] RIGANTI-FULGINEI, F., A. LAUDANI, A. SALVINI and M. PARODI. Automatic and Parallel Optimized Learning for Neural Networks performing MIMO Applications. *Advances in Electrical and Computer Engineering*. 2013, vol. 13, iss. 1, pp. 3–12. ISSN 1582-7445. DOI: 10.4316/aece.2013.01001.

[24] ALTERA. *Nios II Processor Reference: Handbook*. 11.0. San Jose, 2011. Availible at: http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[25] ALTERA. *Nios II Software Developer's: Handbook*. 11.0. San Jose, 2011. Availible at: http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf.

[26] ALTERA. *Nios II Custom Instruction: User Guide*. 11.0. San Jose, 2011. Availible at: http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf.

[27] ALTERA. *Using the SDRAM Memory on Altera's DE2 Board with VHDL Design*. 8.0. San Jose, 2011. Availible at: http://www.cs.columbia.edu/~%20sedwards/classes/2013/4840/tut_DE2_sdram_vhdl.pdf.

[28] ORLOWSKA-KOWALSKA, T. and M. KAMINSKI. FPGA Implementation of the Multilayer Neural Network for the Speed Estimation of the Two-Mass Drive System. *IEEE Transactions on Industrial Informatics*. 2011, vol. 7, iss. 3, pp. 436–445. ISSN 1551-3203. DOI: 10.1109/TII.2011.2158843.

[29] ALTERA. *Using Nios II Floating-Point Custom Instructions: Tutorial*. 11.0. San Jose, 2011. Availible at: http://www.altera.com/literature/tt/tt_floating_point_custom_instructions.pdf.

[30] YOUSSEF, A., Karim. MOHAMMED a A. NASAR. A Reconfigurable, Generic and Programmable Feed Forward Neural Network Implementation in FPGA. In: *UKSim 14th International Conference on Computer Modelling and Simulation*. Cambridge: IEEE, 2012, pp. 9–13. ISBN 978-1-4673-1366-7. DOI: 10.1109/UKSim.2012.12.

[31] ZWOLINSKI, M. *Digital system design with VHDL*. Harlow: Prentice Hall, 2004. ISBN 01-303-9985-X.

[32] CARRASCO, M., F. MANCILLA-DAVID, F. RIGANTI-FULGINEI, A. LAUDANI and A. SALVINI. A neural networks-based maximum power point tracker with improved dynamics for variable dc-link grid-connected photovoltaic power plants. *Materials Science, Electromagnetics and Superconductors and Electromagnetics and Mechanics*. 2013, vol. 43, no. 1-2, pp. 127–135. ISSN 1383-5416. DOI: 10.3233/JAE-131716.

[33] MANCILLA-DAVID, F., F. RIGANTI-FULGINEI, A. LAUDANI and A. SALVINI. A Neural Network-Based Low-Cost Solar Irradiance Sensor. *IEEE Transactions on Instrumentation and Measurement*. 2014, vol. 63, iss. 3, pp. 583–591. ISSN 0018-9456. DOI: 10.1109/TIM.2013.2282005.

[34] PLAGIANAKOS, V. P. and M. N. VRAHATIS. Neural network training with constrained integer weights. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99*. Washington: IEEE, 1999, pp. 2007–2013. ISBN 0-7803-5536-9. DOI: 10.1109/CEC.1999.785521.

[35] CHEN, Y. and W. DU PLESSIS. Neural network implementation on a FPGA. In: *6th Africon Conference in Africa*. Nairobi: IEEE, 2002, pp. 337–342. ISBN 0-7803-7570-X. DOI: 10.1109/AFRCON.2002.1146859.

[36] LAUDANI, A., F. RIGANTI-FULGINEI, A. SALVINI, M. SCHMID and S. CONFORTO. CFSO3: A New Supervised Swarm-Based Optimization Algorithm. *Mathematical Problems in Engineering*. 2013, vol. 2013, pp. 1–13. ISSN 1563-5147. DOI: 10.1155/2013/560614.

[37] LAUDANI, A., F. RIGANTI-FULGINEI and A. SALVINI. Closed Forms for the Fully-Connected Continuous Flock of Starlings Optimization Algorithm. In: *UKSim 15th International Conference on Computer Modelling and Simulation*. Cambridge: IEEE, 2013, pp. 45–50. ISBN 978-1-4673-6421-8. DOI: 10.1109/UKSim.2013.25.

About Authors

Gabriele-Maria LOZITO is a Ph.D. student at the Roma Tre University, Department of Engineering, Rome, Italy. He received his master degree in Electronics Engineering in 2011 presenting a thesis on the characterization of an anechoic chamber for microwave equipment calibration. His research field involves the study of numerical computation applied to system modelling and non-linear optimization, with special interest for the implementation of soft computing techniques on embedded systems.

Antonino LAUDANI was born in Catania, Italy, in 1973. He received the Laurea degree Cum Laude from the University of Catania, Italy, in 1999 and the Ph.D. degree from the University of Reggio Calabria, Italy in 2003, both in Electronic Engineering. Currently, he is Assistant Professor of ElectricaL Engineering in the Department of Engineering at the University of Roma Tre. Dr. Laudani is the author of more than 70 international publications. His main research interests include finite element modeling of electromagnetic devices; particle-in-celland numerical methods; neural networks; optimization and inverse problem solutions; photovoltaic system; and the design of embedded systems.

Francesco RIGANTI-FULGINEI is an assistant professor at the University of Roma Tre, Department of Engineering, Rome, Italy, where he teaches and directs research in non–linear optimization and inverse problems as a faculty member of the Department of Applied Electronics. Prof. Riganti–Fulginei received the Ph.D. degree in biomedical electronics, electromagnetism and telecommunications engineering at the University of Roma Tre in 2007. He is the author of several international publications and has been a visiting professor at the University of Colorado Denver, Denver, Colorado, USA and Okayama University, Okayama, Japan. His research interests include non–linear optimization and inverse problems applied to complex systems, in particular power electronics and electromagnetic devices.

Alessandro SALVINI received the Laurea degree in Electrical Engineering Cum Laude from the University of Rome La Sapienza. He was Assistant Professor (1994), Associate Professor (2001) and, at the present, he is Full Professor at the University of Roma Tree, Department of Engineering where he is also the Scientific Coordinator of the Research Unit of Electrical Engineering. He is involved in tutoring Ph.D. students and is responsible for international agreements with foreign universities for the exchange of faculty and students. His research interests include magnetic material modeling, dynamic hysteresis, optimization and inverse problems, soft computing and evolutionary computation.