Multilayer Neuronal network hardware implementation and co-simulation platform

Nabil. Chouba, Khaled. Ouali, Mohamed. Moalla

(LIP2, UNIVERSITY OF TUNIS EL MANAR)

***Abstract*— Perceptron multilayer neuronal network is widely used in industrial embedded applications. In this paper, we present a new hardware architecture that is fully generic to allow maximum flexibility by selecting the suitable design depending on available resources on FPGA or ASIC: area, consuming power and time execution. We introduce the use of the co-simulation and co-design to ensure the convergence of the learning step in simulated environment. We choose the robot Khepera obstacle avoider as a validation test case.**

***Index Terms*— neuronal network, Perceptron multilayer, back-propagation, co-simulation, hardware implementation.**

I. INTRODUCTION

The changes in the density of integration into integrated circuits, offer the opportunity to explore new different approach from Von Neumann's model. In fact, researches in cognitive science have developed formal models, bio-inspired, called "Neural Networks". These models outperform the limits of the Von Neumann model on several aspects: noise tolerance, learning from limited examples and inherit parallelism. The most popular and used by the industry is "Perceptron multilayer" using back-propagation as a learning process. This model is adopted in our work.

Since 1980, several neural ASIC networks were designed [1] by large companies such as Intel [2], AT & T, Hitachi, Philips, Siemens and IBM [3]. Each design differs from the other depending on the number of neurons emulated, precision bit, the number of synapses and the speed of the learning process. Many developments were implemented on FPGA [4] [5] [6] due to the evolution of the latter one. Until 1995, the FPGA contains less than 5000 logic gates, allowing just the implementation of an instruction MAC [4] [5]. In the years that followed, many implementations [7] were made using the arithmetic series [8] [9] [10]. The use of arithmetic series (ie bit by bit computing) makes it possible to reduce the space used. Instead, it forces them to perform the calculation by cycle iterations.

Another solution to the problem of density FPGA is to divide the calculation steps by reconfigure the FPGA at each stage. Such a solution must submit a compromise between computing time and reconfiguration of time [11], [12].

In this paper we present a new neural design that exploits the rapid evolution of semiconductor field to surpass the existing solution [1-12]. In fact, our method proposes a generic and a dynamically reconfigurable design allowing the possibility to determine, in the synthesis step, the width of arithmetic to be used in the design. In other advantage of this architecture is the possibility to change on line the neural network topology. The last one is characterized by the number of layers, the number of neurons in each one, the weight of connectivity between neurons and the morphology of the neuron activation function. This design was implemented using new refinement methodology based on systemC. A great importance was allocated to the co-simulation of hardware and embedded software, lake of suitable solution to fix network topology and the arithmetic width, allowing to back-propagation algorithm to converge.

The rest of the paper is organized as follow. In section 2 we briefly introduce the formal model of multilayer neural networks, the impact of reduced precision on the back-propagation. In section 3 we present our generic architecture and dynamically reconfigurable, the synthesize results are shown also in this section. In section 4 we explain the co-simulation tool that simulates the IP to command the robot Khepera [17] obstacle avoider. Section 5 concludes the paper.

II. NEURONAL TECHNIQUES

This part presents the most popular neural algorithms for learning process. We focus on the multilayer perceptron network.

A. PROPAGATION ALGORITHM

Consider the neural network presented in Figure 1, composed of *k* layers (*0* to *k-1*). Each layer *i* is

composed of P_i neurons. The network takes n inputs (E_i), it returns q outputs (S_j)

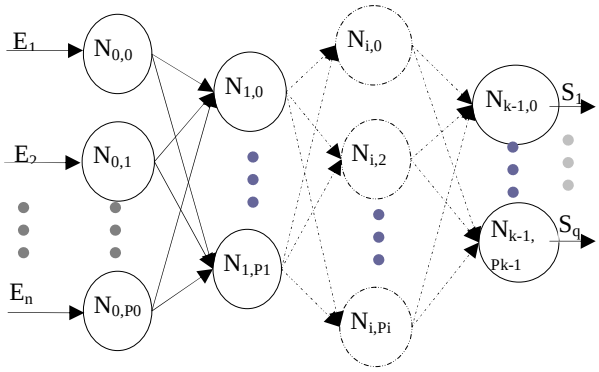


Figure 1: Example of neuronal network

The neuronal processing propagates the neuronal inputs E_i to the outputs S_j. The outputs of the neurons in the first layer are initialized directly by E_i input. The propagation is done through the different layers of neurons; each neuron $N_{m,j}$ of the layer m calculates its output $X_{m,j}$ from a weighted sum of the outputs $X_{m-1,i}$ neurons in the layer *m-1* connected to it. This processing is summarized [16] by the following algorithm:

Begin
Initializing of the output neurons of the first layer: ($X_{0,i} = E_i$)
For each layer m, starting with the layer 1 to k-1,
For each neuron j in the layer m
- Calculate the output $X_{m,j}$ of the neuron j :

$$X_{m,j} = f\left(\left(\sum_{i=1}^{P_{m-1}} X_{m-1,i} \times W_{i,j}\right) + b_j\right)$$

$W_{i,j}$ Is the weight of the synaptic connections $N_{m-1,i}$ $N_{m,j}$ et b_j is the "bias" factor associated to neuron.
The output S_j is confused with $X_{k,j}$.
End

B. BACK-PROPAGATION ALGORITHM

The first stage of learning algorithm [16] is the submission of the neuronal input E and desired output D to the network. Then we calculate the error δ between the desired D value and the effective output value of the network. Next we execute the back-propagation algorithm to find the part error δ_i for the neuron N_i, (the processing is done from the output layer and arriving at the input layer, through the hidden layers) in order to correct the synaptic weight of each neuron.
This algorithm involves five steps:

- *Step 1:* initialization of weights of the network (usually by random)
- *Step 2:* initialization of input and desired output vectors of the network.
- *Step 3:* propagation of the input E to the output (obtaining the propagated output)
- *Step 4:* Calculation of the total error of the back-propagation, to update weight synaptic neurons.
-*Sub-phase 4.1:* Calculation of the total error: **Error** = ½ Σ $(d_i - X_i)^2$ ($i \in [1,...,ns]$ with ns is the number of neurons in the output layer
- *Sub-phase 4.2:* calculating the error of each neuron N_i belonging to the output layer
$\delta_i = X_i (1 - X_i) (d_i - X_i)$
- *Sub-phase 4.3:* For each hidden layer, calculating the error of each neuron N_i belonging to this layer
$\delta_i = X_i (1 - X_i) \Sigma_j \delta_j W_{ij}$ ($j \in [1,...,np]$ with np is the number of neurons (of the next layer), which are connected to the output of neuron N_i).
- *Sub-phase 4.4:* Update of synaptic weights $\Delta W_{ki} = \eta \delta_k X_i$ (W_{ki} is the weight that connects the neurons N_k and N_i, and the η is the epoch learning)
$W_{ki} (t+1) = W_{ki} (t) + \Delta W_{ki}$
- *Step 5:* Convergence analysis to determine the end of learning or iteration from step 2 with the next input.
Steps 1 and 2 are limited only to initialization task while steps 3 and 4 refer to operations of calculation and synaptic weight correction. These two operations are time costly and are hard-coded. Finally, step 5 is a test step; the decision is determined by software.

C. PRECISION LIMITATIONS

The back-propagation explained previously uses continuous functions and real numbers. In hardware implementation, the real numbers are approximated by fixed-point values. The use of floating point is not recommended for hardware applications because of its silicon area cost and execution cycles. Many researches have been developed to study the possibility and effects in using fixed-point or integer approximation.

The analysis of the fixed-point effect made in [15] [13] requires that a minimum synaptic weight precision of 20-22 bits is generally needed to guarantee the convergence of the back-propagation algorithm. This precision can be reduced to 14-16 bits through a judicious selection of learning epoch η. The authors proposed a method of how to choose this η. For the neurons outputs, the precision of 8 to 9 bits is sufficient.

It has been shown in [14] that it is impossible to compensate the error coming from precision by an increase in the number of neurons in the hidden layers. The study also shows that a higher precision is always necessary for the hidden layers.

III. HARDWARE IMPLEMENTATION

The designed IP is named NEURONA; it was implemented to be fully generic and dynamic to overcome the problems of precision limitation and the choice of the neuronal network topology. Theses two problems remain the handicap of neuronal hardware implementation. In fact, they cause the divergence of the back-propagation algorithm (See §II-C). The proposed design is illustrated by Figure 2.

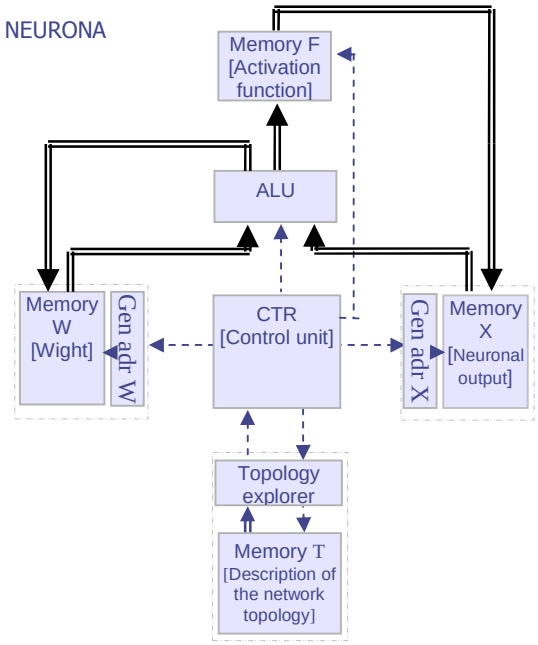
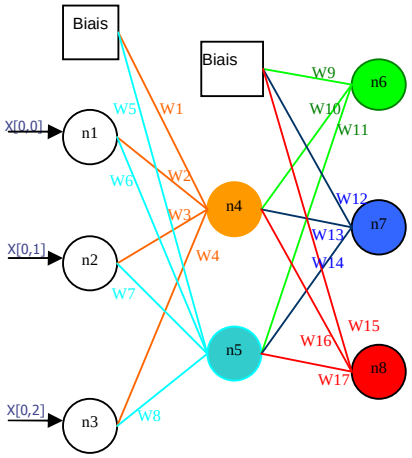


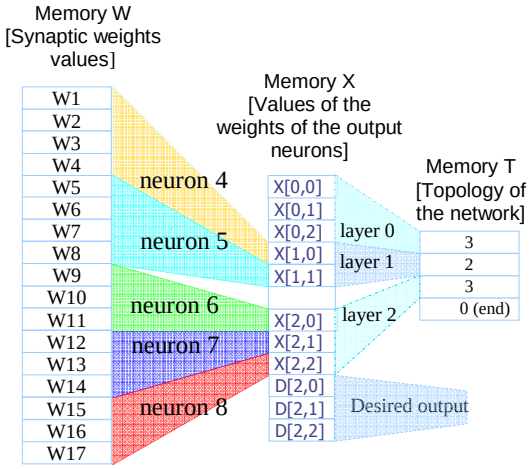
Figure 2: Schema block

Our design is composed of:

- ALU which calculates the output neurons from the inputs values X_i and the synaptic weight W_{ij}.
- Memory F which synthesizes the activation function
- Memory T which contains the topology code of the neuronal network (number of layers and the number of neurons per layer). This memory associates with each layer *i* (*i* ranges from 0 to k-1) the word in the address *i* whose content reflects the number of neurons forming the layer. The last word in the address *k* contains a zero value indicating the end of the network.
- Memory W that contains synaptic weights $W_{i,j}$ of the connections between neurons. The order of $W_{i,j}$ storage in the memory is chosen in order to simplify the generation of address when performing the algorithm.
- Memory X which stores the neuron output value in propagation mode. In back-propagation mode, this memory contains the error made by the neurons. Each Memory T, W and X has its own address generator (Figure 3).
- Topology Explorer which searches the information stored in the Memory T. This exploration is controlled by the Control Unit CTR step by step. At every step, this block informs the CTR whether we move to the next neuron in the same layer or move to the next layer.
- The Control Unit CTR. It uses the information coming from the previous block to command the ALU and the address generator gen_add_x (for the Memory X) and gen_add_w (for the Memory W).



(3-a)



(3-b)

Figure 3: Coding memory of the case studied network

In the first step, the CTR Unit initializes the different blocks. Then, it commands the beginning of propagation. So, it orders the Topology Explorer to read the first two words of memory T. Once the blocks are initialized, the calculation of weighted sum $\Sigma_j X_j W_{ij}$ begins. Finally, the activation function output is stored in X memory. The CTR repeats this task until the propagation is done. In the back-propagation step, we choose those strategies:

- Processing Steps 1 and 2 (§II-B): Storage of input $X_{0,i}$ and the desired output d_i values of the network in the X Memory (Table1)

- Next in the Sub-phase 4.2 (§II-B), the CTR controls the address generators (gen_add_x and gen_add_w) to calculate the error $\boldsymbol{\delta_i}$ in the output layer. This error is then propagated though other network to input layer.

Table 1 Memory X

$X_{0,0}$	$X_{1,0}$	…	…	$X_{f,0}$	d_0
$X_{0,1}$	$X_{1,1}$	…	…	$X_{f,1}$	d_1
...	...	$X_{\alpha,i}$	$X_{\alpha+1,i}$	...	…
$X_{0,n}$	…	…	…	$X_{f,n}$	d_n

- Processing of the Sub-phase 4.3 (§II-B): Calculation of error of each neuron *i* in the hidden layer. To optimize memory space, the value $\boldsymbol{\delta_{\alpha,I}}$ will be stored in the boxes $\boldsymbol{X_{\alpha+1,i}}$ in the X memory. (Table 2) This choice allows to progressively replace $\boldsymbol{X_{\alpha,i}}$ by $\boldsymbol{\delta_{\alpha-1,i}}$ once calculated, $\boldsymbol{X_{\alpha,i}}$ are not longer useful for the back-propagation algorithm.

Table 2 - Memory X

$X_{0,0}$	$X_{1,0}$	…	…	$\boldsymbol{\delta_{f-1,0}}$	$\boldsymbol{\delta_{f,0}}$
$X_{0,1}$	$X_{1,1}$	…	…	$\boldsymbol{\delta_{f-1,1}}$	$\boldsymbol{\delta_{f,1}}$
...	...	$X_{\alpha,i}$	$\boldsymbol{\delta_{\alpha,i}}$	...	…
$X_{0,n}$	…	…	…	$\boldsymbol{\delta_{f-1,n}}$	$\boldsymbol{\delta_{f,n}}$

- Processing of Sub-phase 4.4 (§II-B): Update of synaptic weights. To optimize both hardware implementation and time execution, the two sub-phases 4.3 and 4.4 were carried out simultaneously: each time calculation of the error $\boldsymbol{\delta_{\alpha,i}}$ done, this value is stored in the memory replacing $\boldsymbol{X_{\alpha+1,i}}$ and immediately updating synaptic weight W_{ij} connected to this neuron (weight that connects the neuron *i* to the next layer)

IV. Synthesis Results

The neural architecture has been implemented on FPGA. We give below the characteristics and results of this implementation.
- Synthesis Tool: Symplicity, version 7.3, Build 192R
- FPGA: Apex [ep20k100eqc240-2x]
- Synthesis constraint: Frequency 20.0 MHz (fixed by the studied board)

Table 3 - Synthesis Results

W	X	RAM W	RAM X	RAM T	On line Learn	logic	Memory	Frec, MHz
16	16	200	128	4		18%	23%	25.0
16	16	200	128	4	x	56%	30%	23.9
8	8	32	32	4		9%	15%	37.0
8	8	32	32	4	x	23%	15%	36.1

V. Platform and Co-simulation

The proposed platform is a CAD tool for neuronal embedded implementation, the platform gives the possibility to simulate and select the three components of neuronal system: application environment, the hardware architecture design and the embedded software. The Figure 4 shows the architecture of this tool. The development is specifically dedicated to the mobile robot navigation, and can be advantageously transposed to other neuronal application by just redefining the environmental model.

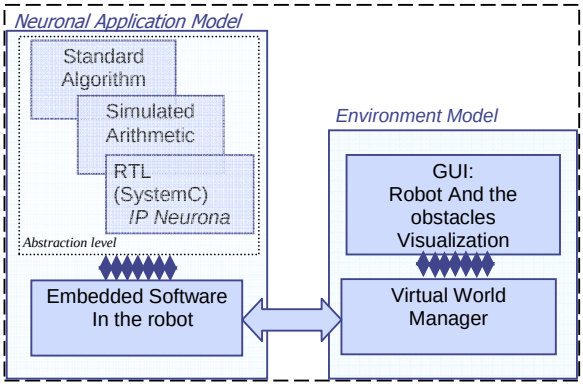


Figure 4: neuronal platform for embedded system

A. Neuronal Application Model

The model consists of neuronal architecture NEURONA coupled to an embedded processor for the supervision of navigation and control learning. The aim of our tool is to achieve optimal architecture NEURONA ensuring the best behavior. The optimization is performed on the number of layers, the number of neuron and the width of the arithmetic. This optimization is achieved by several iterations of refinement based on the simulation at various abstraction levels.

Initially, we perform a standard algorithmic level using floating point of the C++ programming. Then we try to reduce the arithmetic by emulation of the floating-point or fixed-point. Thus, we are sure that we have chosen the adequate parameters of arithmetic. Finally, the use of SystemC RTL level permits to optimize the synthesizable version via generic model of NEURONA. The tool allows validating the learning epoch $\boldsymbol{\eta}$ based on the RTL model. The use of various abstraction levels provides a progressive refinement getting closer to the hardware constraints.

The interaction between NEURONA and the embedded software lets us validate and fix bugs earlier in the design phase.

The fact of using the same language to simulate hardware and software allows us to reduce development and debugging time. The works done in [4] and [5] depend on the emulated hardware model. In fact, to emulate the hardware behaviors and software interactivity, a C++ based hardware model must be developed from its specification.

B. Environment Model

The modeling of the environment aims to create a virtual world representing the robot to serve as a test for embedded neuronal application. This virtual world is modeled by the mathematical equations describing 2D geometry of the navigation space with obstacles. It is also modeled by the equations of emulation behavior of the robot sensors and motor.

The GUI interface allows us to create and modify a virtual environment and to plan and visualize scenarios navigation "used as a test bench".
We used language C++, SystemC and Qt library [18] for the management of graphical interfaces. The migration of embedded environment software is facilitated by the fact that all microcontrollers support the C code.

The implemented platform helps us to validate the overall system behavior, using virtual world. The testbenches are generated by the GUI and allow us to view the evolution of the robot in the universe.

C. Validation of hardware component

The traditional IP validation method is to create a pattern generator based on a C++ or MATLAB algorithm named reference algorithm. The output file is usually a HDL-written testbench.

Under this project, we will adopt a slightly different methodology. Indeed, we compare the reference algorithm results and those of the implementation in SystemC. We perform both algorithms in a single program adding conditional structures which allow this comparison.

The debugging is based mainly on statistical benchmarks to find the source of the error. This method enables us to measure the difference between the output of the algorithm using the standard floating-point and fixed-point and our implementation in SystemC.

D. Simulation results

The tool developed allows us to explore the hardware and the possible neuronal topologies. It also permits to generate graphs (figure 5) which facilitate the simulation analysis.

Several leaning examples confirm the theoretical research already presented on §II.C. Results show the impact of the arithmetic on the leaning speed and on the convergence of the back-propagation algorithm.
The figure 5 clearly shows that 18 bits are not sufficient for the convergence. It proves that the convergence increases by relaxing the arithmetic constraints.

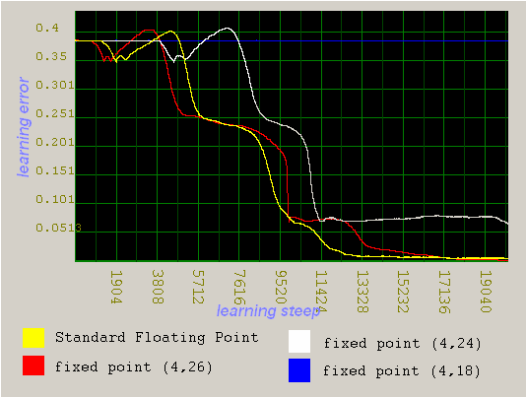


Figure 5: Leaning Path including hardware constraints

VI. Conclusion and perspective

Encouraged synthesis results give us the possibility to go deeply in the exploration of other more interested hardware architecture. We can exploit the inherit parallelism and the growth in FPGA capacity and performance. The used co-simulation has ensured the convergence for used neuronal implementation. This pushes us in future work to continue on the same perspective. In fact, we can use the SystemC TLM library to accelerate the architecture exploration phase and improve the refinement design method and co-simulation.

References

[1] Jan N. H. Heemskerk , Overview of neural hardware ,Unit of Experimental and Theoretical Psychology Leiden University, P.O.Box 9555, 2300 RB Leiden The Netherlands

[2] J-P. LeBouquin, IBM Microelectronics ZISC, Zero Instruction Set Computer, Proc. of the World Congress on Neural Networks, Supplement, San Diego, 1994.

[3] J. Hopfield and D. W. Tank, Computing with neural circuits: A model, science, vol. 233, pp. 625-633, August,1986.

[4] Cox, C.e.; Blanz, w.e. ganglion-a fast field-programmable gate array implementation of a connectionist classifier . solid-state circuits, ieee journal of (march 1992 volume 27 number 3)

[5] Marcelo H. Martine , A Reconfigurable Hardware Accelerator for Back-Propagation Connectionist Classifiers (1994), university of California Santa Cruz

[6] Aaron T. Ferrucci ACME, A Field-Programmable Gate Array Implementation of a Self-Adapting and Scalable Connectionist Network (1994), university of California Santa Cruz

[7] Bernard GIRAU, Du parallélisme des modèles connexionnistes à leur implantation parallèle, L'école normale supérieur de Lyon (1999)
[8] Jean-Luc Beuchat, Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables. Thèse de doctorat, Ecole Polytechnique Fédérale de Lausanne, 2001. Thèse No 2426. BibTeX
[9] Jean-Luc Beuchat, Arnaud Tisserand, Opérateur en-ligne sur FPGA pour l'implantation de quelques fonctions élémentaires. Actes de la conférence Sympa'8 - Symposium en Architectures Nouvelles de Machines, pages 267-274, 2002. BibTeX
[10] K. S. Trivedi and M. D. Ercegovac. On-line Algorithms for Division and Multiplication. IEEE Transactions on Computers, C-26(7):681-687, 1977.
[11] J.G. Elredge and B.L.Hutchings : rrann a hardware implementation of the backpropagation algorithm using reconfigurable fpgas , ieee int cof, neural network june 1994
[12] JL.Beuchat, J-O.Haenni and E. Sanchez, Hardware Reconfigurable Neural Networks
[13] J.L. Holt and J-N Hwang. Finite precision error analysis of neural network hardware implementations. IEEE Transactions on Computers', 42:1380-1389, 1993.
[14] Yun Xie, Marwan Jabri, 91Analysis of the Effects of Quantization in Multi-Layer Neural Networks Using Statistical Model (1992)
[15] Leonardo M. Reyneri and Enrica Filippi. An Analysis on the Performance of Silicon Implementations of Backpropagation Algorithms for Artificial Neural Networks. 1991 IEEE Transactions on Computers, 40(12):1380--1389, December 1991.
[16] B. Widrow and M. A. Lehr. 30 Years of Adaptativ Neural NetWorks: Perceptron, Madaline and Backpropagation. Proc. IEEE, 78(9):1415-1442, 1990
[17] K-TEAM S.A., Preverenges, Switzerland, Khepera User Manual. (http://www.kteam.com).
[18] Trolltech provides cross-platform software development frameworks and application platform (www.trolltech.com)