Artificial Neural Network Engine: Parallel and Parameterized Architecture Implemented in FPGA

Milene Barbosa Carvalho¹, Alexandre Marques Amaral¹,
Luiz Eduardo da Silva Ramos^{1,2}, Carlos Augusto Paiva da Silva Martins¹,
and Petr Ekel¹

¹ Pontifical Catholic University of Minas Gerais (BRAZIL), ² Rutgers University (USA),
Av. Dom José Gaspar, 500, Prédio 3, 30535-610, Belo Horizonte, MG – Brazil
{milene, alexmarques, luizesramos}@ieee.org,
{capsm, ekel}@pucminas.br

Abstract. In this paper we present and analyze an artificial neural network hardware engine, its architecture and implementation. The engine was designed to solve performance problems of the serial software implementations. It is based on a hierarchical parallel and parameterized architecture. Taking into account verification results, we conclude that this engine improves the computational performance, producing speedups from 52.3 to 204.5 and its architectural parameterization provides more flexibility.

1 Introduction

Artificial neural networks (ANN) implemented in digital computers normally generate a high demand of computational performance. Its serial software implementations executed in programmable hardware, e.g. microprocessor, normally produce relative high response time and unsatisfactory performance [1]. This performance problem is a critical factor for most ANN based applications and it is our motivator problem for the present work. In many situations, first of all, in real-time systems, a high response time can invalidate the responses and solutions. Our main goals in this work are to propose, design and implement an artificial neural network hardware engine in order to improve the computational performance.

2 Proposed Engine

Among different types of artificial neural networks (ANN), we initially propose and design an engine to implement multilayer perceptron (MLP) networks [2,3]. This choice is based on the high utilization of MLP on ANN applications [4].

MLP networks are composed of at least three layers: one input, one or more hidden and one output layers. Input layer does not perform processing, but represents an input data set for neurons of the first hidden layer. Hidden and output layers perform processing of inputs and weights and are composed of perceptron neurons, shown in fig.1a. MLPs are feedforward networks (fig.1b). This means that the inputs of the neurons of any layer (except the input layer) are the output values from neurons of the previous layer [3] [4].

As presented in fig.1a, the output of a perceptron neuron is calculated by the function $f(S)$, where f is the transfer function and S represents the summation of all input-weight products. The transfer function can be any function, but the more popular are threshold and sigmoid functions [4].

Fig. 1. a) Perceptron neuron. b) MLP topology example. c) MLP execution pseudo-code.

Analyzing fig.1a, we are able to state that there is inherent spatial parallelism in the execution of the neuron's products, called intra-neural parallelism. In fig.1b we notice that a neuron inside a layer is independent from the others within the same layer, (intra-layer parallelism). However, there is dependency among the neurons from a layer and those from the previous layer. It happens because the outputs from a layer are the inputs of the next layer. Nevertheless, the computation of different layers can be done simultaneously, since each neuron has all inputs (temporal parallelism or pipeline). It means that if the layers process different data sets, they can execute simultaneously (inter-layer parallelism).

Fig.1c presents a MLP pseudo-code. The first (outer) loop executes the entire network for all data sets. The second loop executes all hidden and output layers. The third loop executes all neurons of the layer specified in the second loop. The fourth loop executes the products and sums of each neuron, where weights and inputs are determined by previous loops. After that, the transfer function is applied to the total sum of a neuron, generating the neuron's output. Serial code implemented like this and executed in general purpose processors (GPPs) fails to explore the several different levels of inherent parallelisms inside an MLP, as previously indicated. In this implementation, the operations of each neuron are sequentially processed and also the operations within each layer and all over the network. Since this implementation fails to explore parallelism, the overall performance cannot reach the ideal high performance.

Some works implement ANN in parallel computers [1], e.g., clusters and multi-processors [5], which yield great speedup over the sequential monoprocessed one. However, since MLP network present fine-grained parallelism, their implementation in parallel computers not always is efficient, due to speedup, scalability and cost.

Our solution hypothesis is to design and implement MLP networks using hierarchical parallel and parameterized dedicated hardware architectures, to improve the computational performance.

The neuron designed to compose our architecture (fig.2a) has three main modules, named: multiplication, addition and transfer function. In the first module, the inputs are multiplied by their respective weights. In the second, all products are summed.

Then, the summation of the products is processed by the transfer function module, which calculates the neuron's output. Among all possible hierarchical parallelism parameters, there are spatial parallelism in the multiplications and temporal parallelism (pipeline) in each neuron among its three modules.

Fig. 2. a) Proposed neuron. b) Proposed MLP architecture. c) Neuron implementation.

The main features of our architecture are its spatial and temporal parallelisms in different hierarchical levels, and their parameterizations. The parameters are divided in two groups, named network and architecture parameters. The first group determines the main features of the network, such as: number of inputs, number of neurons, number of layers, type of transfer function and so on. The second group determines the main features of the architecture, such as: parallelism degree among layers, neurons and modules, implementation of the sub-operations, word length (to represent input, weight and output values), and so on.

The proposed architecture is hierarchically composed of layer, neurons and modules (fig.2b). Observing fig.2, we notice that there are four possible parallelism hierarchical levels in our architecture: (1) H1 is the network, composed of layers (temporal parallelism); (2) H2 is the layer, composed of several neurons (spatial parallelism); (3) H3 is the neuron, with operation modules pipelined execution (temporal parallelism); (4) finally H4 is neuron module with parallel implementation (temporal and spatial parallelism) of each module (fig.2a). Fig.2.c is a possible implementation of a neuron with parallelism in H4 in multiplication and addition modules.

Although there are parallelism levels in our architecture, they can be used or not. Thus, the designer must analyze the tradeoffs between performance and cost. Total parallelism implies in high performance, but higher relative cost. For example, it is possible to design an engine without H1 parallelism. In this case, only one layer would be executed at a time, which does not affect other parallelism levels or their execution.

Using the previously described architecture, we have implemented our artificial neural network engine. To design, verify and synthesize our engine, we have codified it in a Hardware Description Language (HDL). The chosen language was VHDL (VHSIC – Very High Speed Integrate Circuit Hardware Description Language)

because of its design portability and simplicity to describe a design. Thus, it was possible to define an engine with network and architecture's parameters easily modified. Our implementation has the maximum parallelism that the architecture allows: (1) the layers association composes a pipelined structure; (2) the neurons are disposed in parallel inside each layer, (3) the neuron modules are disposed in a pipelined structure and inside them was applied parallelism. All multiplications are executed in parallel and the summation is executed in a binary tree of synchronous pipelined adders.

Besides the internal parallelism of neuron modules, their design is important for the performance. We designed multipliers and adders considering the tradeoffs between cost and performance. The values of their latencies are two and one clock pulses, respectively. Another limitation of the neurons' implementation is the transfer function (e.g. the sigmoid equation is complex to implement in hardware). There are two frequently used solutions: the implementation of a lookup-table containing the function values for a range, or the implementation of a piecewise function. The former consumes large hardware resources and it implies in high cost and the latter provides less precision. But as errors are inherent of neural networks, some implementations using approximation are acceptable. Thus, we implemented a piecewise function known as PLAN function [6].

The performance of the engine is determined by parallelism degree and clock frequency necessary for neuron's modules execution. In order to determine the pipeline latency of our engine, it is necessary to consider the number of network layers. In Section 3 we discuss the latency and global performance for two engine FPGA implementations.

3 Verification Results

Our verification method consists of the following steps: (1) engine verification using VHDL logic simulations; (2) verification of the FPGA implementation using VHDL, post Place and Route (synthesis), simulations; (3) experimental measurements of software implementations; and (4) performance evaluation and comparative analysis of the hardware and software implementations.

Firstly, we synthesized our engine in a Xilinx XC2V3000 FPGA, and used it to solve a simple problem (XOR operation). We chose this operation in order to verify the implementation's behavior and functionality. We also implemented the same ANN in software and executed on top of a Pentium IV 2.66GHz and an Athlon 2.0GHz. The weights were obtained from the training of the neural network software implemented in C++. In both cases the implemented neural network is a three-layer MLP with a number of inputs (i) varying from 2 to 5 and only one output. The input layer has i neurons, the hidden layer has 2i-1 neurons and the output has one neuron.

We executed the network in hardware and software, and compared the results. Our implementation had a maximum error of 0.02 from the results regarding the software implementation. This architecture's error is insignificant for this problem, considering the required precision and output range from 0 to 1. If a higher precision were required, the word length could be increased.

Fig.3a presents the response time of a single execution of the implemented MLP. In the FPGA implementation, this response time represents the latency of its structure.

Analyzing the results, we notice that the response times of the serial software implementation in both GPP processors increase, as the number of inputs increases, because of its serial processing, as well as the involved overheads (e.g. memory access, operating system etc). Differently, the response time of the FPGA implementation was almost invariable, because of its parallel execution. The FPGA implementation performed better than the software implementations, with a speedup ranging from 7.480 to 19.714.

Fig. 3. Response time of the ANN in FPGA and on top of traditional processors

In Fig.3b we present the response time of a thousand consecutive executions of the MLP in software and hardware. The software implementation behaved similarly as before, with proportional increases in the response times. In the same figure, we notice that the speedup of the non-pipelined FPGA implementation also kept the same proportion, ranging from 7.480 to 19.714. Nevertheless, the pipelined FPGA implementation performed even better, yielding a speedup ranging from 7 to 11 regarding the non-pipelined implementation and from 52.354 to 204.576 regarding the serial software implementations.

Table 1. Resource utilization of the FPGA implementation

	2 inputs	3 inputs	4 inputs	5 inputs	Available
Slices	2448	5256	9179	13453	14336
Flip-Flops	935	2150	3425	5748	28672
4-Input LUTs	4326	9279	16061	23437	28672
Bonded IOBs	48	64	80	96	720
Max. Frequency (MHz)	50.236	59.027	57.091	47.323	-
Clock cycles	7	9	9	11	-

In Table1, we notice that the resource utilization within the FPGA is proportional to the overall number of processing neurons and to the number of inputs of each neuron. We also highlight that the engine implemented in FPGA executes with an average frequency of 46.3 times lower than the processors. Thus, our implementation

also contributes for lower energy consumption, temperature dissipation and the final cost of the product. Even with a much lower frequency, our engine still has a much better performance, as mentioned before.

4 Conclusions

Considering presented and analyzed results, we conclude that the proposed engine correctly implemented the MLP network and yielded better computational performance than software implementations, with speedups from 52.3 to 204.5. Thus, our main goals were totally reached. Also, our engine yield better performance than some related works [1] [5] [7] based on dedicated parallel architectures and FPGA implementations. The main contributions of this work are the originality of our proposed architecture, considering the parallelism and parameterization architectural features, and its higher computational performance improvements. Also, there are the flexibility and scalability provided by the parameterization. Among future works we highlight: study other parameter combinations and their performance impact, implement our engine using different design styles and implementation technologies and design engines to implement other ANN models.

References

1. Misra, M.: Parallel Environments for Implementing Neural Networks. *Neural Computing Surveys*, Vol. 1 (1997) 48-60
2. Rosenblatt, F.: The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, Vol. 65, (1958), 386-408
3. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning Internal Representations by Error Propagation. In: Rumelhart, D.E., McClelland, J.L. (eds.): *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, Cambridge, MA, MIT (1986) 318-364
4. Haykin, S.: *Neural Networks: A Comprehensive Foundation*. 2nd edn. Prentice Hall (1998)
5. Seiffert, U.: Artificial Neural Networks on Massively Parallel Computer Hardware. *Neurocomputing*, Vol. 57 (2004) 135-150
6. Tommiska, M.T.: Efficient Digital Implementation of the Sigmoid Function for Programmable Logic, *IEE Proceedings Computer and Digital Techniques*, Vol. 150, No. 6, (2003), 403-411
7. Linares-Barranco, B., Andreou, A.G., Indiveri, G., Shibata, T. (eds.): Special Issue on Neural on Neural Networks Hardware Implementations, *IEEE Transactions on Neural Networks*, Vol. 14, No. 5, (2003)