An Efficient Architecture for Floating Point based
MISO Neural Networks on FPGA

Antonino Laudani, Gabriele Maria Lozito, Francesco Riganti Fulginei, Alessandro Salvini

Department of Engineering
Università degli Studi di Roma Tre
Via Vito Volterra 62 – 00146 – Roma (Italy)
gabrielemaria.lozito@uniroma3.it

Abstract - The present paper documents the research towards the development of an efficient algorithm to compute the result from a multiple-input-single-output Neural Network using floating-point arithmetic on FPGA. The proposed algorithm focus on optimizing pipeline delays by splitting the “Multiply and accumulate” algorithm into separate steps using partial products. It is a revisit of the classical algorithm for NN computation, able to overcome the main computation bottleneck in FPGA environment. The proposed algorithm can be implemented into an architecture that fully exploits the pipeline performance of the floating-point arithmetic blocks, thus allowing a very fast computation for the neural network. The performance of the proposed architecture is presented using as target a Cyclone II FPGA Device.

Keywords: FPGA, Neural Networks, VHDL, embedded floating point.

I. INTRODUCTION

The use of Artificial Neural Networks (NN) is a well-established method to solve interpolation problems, develop control systems and bypass complex calculation for nonlinear optimization [1-3]. Indeed, if seen from a high-level programming point of view, a NN is utterly simple. The process of building the input value for each neuron is just an iterative sum of products, while the output of the neuron (referred as “act”) can be computed as a non-linear function of input of the neuron (referred as “net”). An interpreted high level programming language, like Matlab®, Wolfram Mathematica or Python, makes use of powerful libraries and the oversized computing power of the CPU to elaborate the NN output. The computing time of a simple SISO (Single Input Single Output) Neural Network with 10 neurons in the hidden layers is in the order of the tenth of milliseconds on a Core™ i3 machine running Matlab. Considering the analytic advantages of the tool itself, in high-level environment the computing time of the NN is seldom considered as a determining factor. However, such assumption cannot be made when a real time embedded system is involved. Consider the implementation on an embedded system of a simple SISO Feed Forward Neural Network (FFNN), the performance of the system has two main bottlenecks. The first one lies in the number of floating point multiply-and-accumulate (MAC) operations. The second one in the complexity of the non-linear activation function. The latter problem has been addressed in several papers focused on optimizing the performance of a NN in embedded system by reducing the complexity of the activation function [4-6]. A possible strategy is using a polynomial approximation of the function near the origin and consider the saturation values far from it. Another possibility is to sample the activation function and address it directly in a Look-Up Table (LUT), trading memory for performance [7-8]. Another set of studies revolved around the MAC part of the network, where different architectures and algorithms were proposed [9-12]. On a low-end microcontroller unit, the floating-point operations are carried out in software, leaving the code performance to the capacity of the assembly compiler libraries. Many high-end microcontrollers on the other hand figure a Floating Point Unit with dedicated multipliers that can speed up the MAC part of the neural network computation. Several comparative studies have been made on this topic [13-14]: however, few options are available to the programmer to optimize the code itself. Working on a FPGA embedded system a different approach can be followed to enhance the Neural Network performance. All the proposed solutions (especially the LUT) to optimize the activation function computation can be applied. The MAC algorithm is usually carried out by using a multiplier, an adder and a one-word register to store the partial MAC results [15]. Since floating-point arithmetic is more complex than integer arithmetic, computation blocks usually trade performance for footprint (in terms of logic elements used) by adding a pipeline. Since the process of multiply-and-accumulate is inherently sequential, the adder must wait for a new multiplication result each time it updates the cumulative sum, yielding poor results in terms of throughput. The burden of this problem grows exponentially if the Neural Network has multiple inputs (MISO) or more hidden layers. Although the use of a MISO architecture can be avoided by taking advantage of analytic decomposition techniques [16-17], this approach is of complex implementation in embedded systems.

In this paper, an approach figuring the splitting of MAC procedure by using partial products is presented. As will be discussed, the proposed solution is a revisit of the classical algorithm for NN computation, able to overcome the main computation bottleneck in FPGA environment. This solution has been effectively implemented on a Cyclone II FPGA, showing good performance in comparison to different programming environments.
II. ARITHMETIC BILL OF A NEURAL NETWORK

The basic expressions of the \( u \)-th output of the Feed Forward Neural Network shown in Fig. 1, with \( n \) input neurons, \( m \) neurons in one single hidden layer and \( p \) neurons in the output layer is shown in (1)(2) and (3).

\[
Out_u = b^\text{out}_u + \sum_{k=1}^{m} w^\text{out}_{k,u} \times \text{act}_k \\
\text{act}_k = \frac{1}{1 + e^{-\text{net}_k}} \\
\text{net}_k = b^\text{hidden}_k + \sum_{i=1}^{n} w^\text{hidden}_{k,i} \times \text{input}_i
\]

The matrix representation of the stated relations points out that each neuron \( \text{net} \) can be seen as the scalar product between the input vector and the weight matrix.

\[
\begin{bmatrix}
\text{net}_1 \\
\vdots \\
\text{net}_n
\end{bmatrix} =
\begin{bmatrix}
w_{11} & \cdots & w_{1l} \\
\vdots & \ddots & \vdots \\
w_{m1} & \cdots & w_{ml}
\end{bmatrix}
\times
\begin{bmatrix}
\text{input}_1 \\
\vdots \\
\text{input}_n
\end{bmatrix}
+ \begin{bmatrix}
bias_1 \\
\vdots \\
bias_n
\end{bmatrix}
\]

Then, the resulting vector of \( \text{net} \) values must be processed by the nonlinear activation function to create the output values, \( \text{act} \), of the layer itself. This process can be repeated for multiple consecutive hidden layers, however, it is proved that a single layer FFNN of proper size can interpolate any function [18]. The output layer is a particular case of the hidden layer were the \( \text{act} \) value is propagated directly in output, and the relation between \( \text{act} \) and \( \text{net} \) is usually a linear function. From this, the scalar product is analogous to the one present in the hidden layer. Then, a generalized representation of the Neural Network as a succession of stages, where each \( k \)-th layer has \( n_k \) inputs, \( m_k \) outputs and \( f_k \) relation between \( \text{act} \) and \( \text{net} \) can be formulated. Each stage contributes to the bill of floating point operations independently. For the scalar product the needed operations are: \( m_k \times n_k \) floating point products and \( m_k \times n_k \) floating point sums. Considering a log-sigmoid activation function, the needed operations are: \( m_k \) exponentials, \( m_k \) sums and \( m_k \) divisions. Then, even a modest neural network with 10 inputs, 10 neurons in the hidden layer and 1 output yields a total of 100 products, 110 sums, 10 exponentials and 10 divisions for the hidden layer, plus another 10 products and 10 sums for the output layer.

III. MULTIPLY AND ACCUMULATE BOTTLENECK

In HDL language, the multiplier and adder for floating point numbers are usually implemented with an arithmetic logic block that shows a pipeline delay. The delay of the blocks is the time (expressed in clock cycles) between the first input getting in and the first output going out. After this initial delay has expired, the data will output with the same cadence (even clock-wise) it was input. Suppose a multiplier block with 3 clock cycles delay. If data is presented at \( t = \{0, 1, 2, 3, 4\} \), the result of the multiplication can be collected at \( t = \{3, 4, 5, 6, 7\} \). Now suppose a “multiply-and-accumulate” block as the one shown in Fig. 2, where the multiplier and the adder have both 10 clock cycles delay. The first two factors are loaded in the multiplier at \( t = 0 \). At \( t = 10 \), the output is available at the input of the adder, which can start the computation and complete it after 10 cycles (\( t = 20 \)). Once the computation is completed, the adder can update the 32-Bit accumulator. Since only after the accumulator has been updated the next sum can begin, exploiting the blocks pipeline is impossible: if two data were sent in the multiplier at \( t = \{0, 1\} \) the second product would be ready at \( t = 11 \). At this time, even if the adder could start a pipelined sum, one of the operands (the one from the accumulator) would not be ready for another nine clock cycles. In conclusion, the adder delay limits the throughput of the MAC. The solution to this problem is twofold: first, the multiplier timing must be decoupled from the adder; second, a particular approach for the adder must be followed to allow a synchronization between the end of a sum and the beginning of a new one. The first problem can be easily solved by storing the partial products of the neurons in a buffer memory. Since the results are all stored, the data can be sent into and collected from the multiplier at full speed. The synchronization of the adder is a more complex matter. For each of the \( m \) neurons in the layer, a total of \( n \) partial products must be summed. The total sum of \( n \) numbers requires \( n-1 \) operations, with a variable number of operations requiring the result of a previous sum. For easiness, we will suppose the worst case where all the operations but the first one will require a previous result.

\[
\begin{array}{c}
n_1 + n_2 = n_{12} \\
n_{12} + n_3 = n_{123} \\
n_{123} + n_4 = n_{1234} \\
\ldots \text{and so on}.
\end{array}
\]

Figure 2: A “multiply-and-accumulate” block with input, weights and biases.
Supposing the adder has a 10 cycles delay, between each sum, at least 10 cycles must pass. However, during this delay the adder can compute the sums for the other neurons.

\[
\begin{align*}
\text{n}_1 + \text{n}_2 &= \text{n}_{12} & \#1 \\
\text{n}_1 + \text{n}_2 &= \text{n}_{12} & \#2 \\
\text{n}_1 + \text{n}_2 &= \text{n}_{12} & \#3 \\
\vdots \\
\text{n}_1 + \text{n}_2 &= \text{n}_{12} & \#9 \\
\text{n}_1 + \text{n}_2 &= \text{n}_{12} & \#10 \\
\text{n}_{12} + \text{n}_3 &= \text{n}_{123} & \#1 \\
\text{n}_{12} + \text{n}_3 &= \text{n}_{123} & \#2 \\
\text{...and so on.}
\end{align*}
\]

Obviously, this calls in some considerations on the pipeline delay size. If the pipeline delay \( d \) is equal to \( m \), the system will be inherently synchronized. If \( d \) is larger than \( m \), the computation time for the sum will be the same as the one required for a layer of \( d \) neurons. If \( d \) is smaller than \( m \), the system could go out of synchrony, since the results from the previous computation would arrive too early. It is possible however to increase the pipeline depth of any block by adding a cascade of delay registers to the output. In conclusion, the larger between \( d \) and \( m \) determines the computation time for the sum.

**IV. MISO NN Core**

The algorithm proposed in the previous section was implemented in FPGA environment. The architecture, shown in Fig. 3, is composed by a cascade of high performance arithmetic blocks developed by Altera ®, whose dataflow is controlled by a dedicated Control Block running a Finite State Machine coded in VHDL. The proposed architecture can compute a 10 inputs MISO NN with 10 neurons in the hidden layer.

Four main blocks can be identified in the architecture: the MAC block, the Activation Function, the Control Block, and the Memory block. Each of these blocks will be discussed.

**MAC:** The MAC block is responsible for the execution of the enhanced “multiply-and-accumulate” algorithm previously show. Input data and weights are fed to the multiplier, which stores the results in the 256x32-Bit RAM. As it can be seen, whereas the connection between RAM and multiplier is unidirectional, the connection with the adder is bidirectional. The reason lies in the “-and-accumulate” optimization: the adder must be able to read the partial products while writing the results in the RAM itself. Amongst other reasons, this is why the RAM must be of dual access type. As it can be seen, the Adder is isolated from the outside architecture even if the biases for the neuron could be sent directly as input. Instead, when needed, the biases are (pre)loaded in the RAM.

**Activation Function:** This block computes the activation function shown in (1). The first operation needed is a sign change, achieved through a logical not on the MSB of the input, representing the sign of the floating-point value. The result is fed to an exponential arithmetic block whose output is connected to an adder that sums the result to the constant value of one. The result of the sum is then inverted and the result (the activation value of the neuron) is propagated to the output. Since in this case the operations are independent,
the three pipelines were used in simple cascade without the need for a buffer RAM. A note on the choice for the activation function is necessary: this particular function was chosen, instead of the more common \(tansig\) activation function, for the easiness of implementation in FPGA environment: the arithmetic for a \(tansig\) activation would require an additional adder and multiplier. However, a \(tansig\) activation function could also be seen as:

\[
tansig(a) = 2 \times \log\left(2 \times a\right) - 1 \quad (5)
\]

Where the additional subtraction and products are apparent.

**Memory:** The memory block is a set of 32-Bit RAM units. The blocks hold the input values, the weights and the biases of the NN. These blocks are accessible by JTAG interface for online programming. The Feedback block is used to store the values from layer to layer: it is connected via multiplexer to the multiplier along with the NN input. As the first layer computing is completed, the second layer receives as input the results from the previous layer.

**Control Block:** A control block is required to manage data processing from input to output. The block needs to synchronize the dataflow according to the pipeline delays, thus sending and recovering data at specific times. Along with this, it is responsible for memory addressing. In a high-level programming language, a simple and effective method to create a control system is a finite state machine. A finite state machine (FSM) is a set of code blocks, each responsible for a particular function (state). Each of these functions is executed accordingly to the value of a state variable, controlling a switch/case structure. In embedded system programming, the switch/case structure is often enclosed in an endless loop, usually referred as superloop. This approach is not as straightforward in HDL languages, since the code does not describe a set of instructions executed by a processor, but instead a logic architecture to be synthesized. Then, a hardware emulating a processor sequential behavior must be created. A possible approach, proposed in [19] and successfully applied in [8], revolves around the creation of an instruction counter whose value is increased at every clock edge. The instruction counter is decoded by a complex net of comparators, each triggering specific logic functions. The important advantage obtained from this approach is that the “artificial” processor is inherently synchronous with the rest of the architecture. This allows time tuning of instructions, a fundamental requirement for pipeline alignment. In VHDL this architecture can be defined by the use of two code blocks (processes), one sequential and one combinatorial. The first process increases the instruction counter at every clock edge, and it is synthesized as a simple binary counter. The second process decodes the instructions into logic signals, and it is synthesized as a net of comparators.

**Data Flow:** The data flow of the entire system is organized by the control block. The following routines are executed in overlap to maximize the throughput; an indicative timeline is shown in Fig. 4. The type of operation performed can be of either loading a pipeline with operands, or recovering the results from it. The only exception, beside the initialization, is the READ_PP routine, where the pipeline is loaded and unloaded continuously.

**Hidden Layer:**
- **LOAD_MULT_F:** The input RAM and the weights RAM are connected to the multiplier. All the inputs and weights are sent to the multiplier clock-wise.
- **LOAD_PP:** The output of the multiplier is connected to Port A of the dual access RAM. The partial products are stored in the RAM.
- **READ_PP:** This routine computes the sum of the partial products. Port A of the RAM is connected to the input of the Adder. Port B of the RAM is connected to the output of the Adder. The accumulated result is written in base 10 memory positions: neuron 1 in 0x000, neuron 2 in 0x00A, neuron 3 in 0x014.
- **LOAD_BS:** Port B of the RAM is connected to the biases RAM. The biases are loaded in base 10+1 positions: neuron 1 in 0x001, neuron 2 in 0x00B, neuron 3 in 0x015.
- **SUM_BS:** This routine sums the biases to the accumulated partial products. Note that after this routine is completed, the base 10 positions of the dual access RAM will have the net values of the neurons.
- **LOAD_AF:** Port A of the RAM is connected to the input of the activation function chain. The net values of the neurons are sent through the chain clock-wise.
- **WRITE_AF:** The Feedback RAM is connected to the output of the activation function chain. The results from the activation function are stored in the RAM.

**Output Layer:**
- **LOAD_MULT_F2:** Analogous to LOAD_MULT_F, but with the output layer weights.
- **LOAD_PP:** As above.
- **READ_PP:** As above.
- **LOAD_BS_F2:** Analogous to LOAD_BS, but with the output layer biases.
- **SUM_BS:** As above.

![Figure 4: Timeline of the finite state machine routines](image)
SISO/MISO/MIMO scalability: It can be seen from the timeline that the output layer routines are as time-consuming as the hidden layer routines. This should be not expected if the network is a MISO architecture. For example, a 10-10-1 network would require 100 products and sums for the hidden layer, but only 10 products and sums for the output layer. In this architecture, both the hidden layer and the output layer perform the same number (100) of operations, but in the output layer, 90 of the 100 weights are null. This may appear as a waste of time resources, but two aspects must be discussed. First, only the multiplier routines throughput could be increased. The adder routines throughput, as explained in section III, are limited by the adder pipeline delay. The delay cannot be reduced because of the hidden layer size. Another adder, with a specific pipeline delay for the output layer could be implemented, but at the cost of incrementing the used resources. Second, by computing the output layer with the same routines used in the hidden layer, the architecture can be used for both a MISO NN and a MIMO NN. In fact, by modifying the weights and biases null elements, this architecture can work as a SISO, MISO or MIMO. In Fig. 5, three of the possible configurations for the network are shown. Each couple of tables represents the RAM containing the weights of the NN. The memory content was rearranged on rows of 10 elements. The red elements are non-null values, the gray elements are null values. The Hidden Layer table is filled analogously as the weight matrix (or column, in case of single input) shown in (4), with the conceptual difference that the matrix size corresponds to the size of the non-zero area in the table. In the Output Layer table, the width of the non-zero area is bound to the height of the Hidden Layer table (i.e. the number of hidden neurons), while its width is the number of outputs desired for the network. The Bias memory arrangement is simpler and does not require a graphic explanation. The column vector of the biases for the Hidden Layer goes from 0x000 to 0x00A, while the one for the Output Layer goes from 0x00B to 0x014.

V. PERFORMANCE ANALYSIS COMPARISON

To evaluate the performance of the proposed architecture, the three configurations (SISO, MISO, and MIMO) were implemented in different environments, profiling the code execution. All the networks had 10 neurons in the hidden layer. The MISO architecture had 10 inputs and 1 output. The MIMO architecture had 10 inputs and 10 outputs. Four different environments were chosen for the validation: Matlab, an x86 architecture, Nios II Soft Processor for FPGA and a Cortex M4A ARM Microcontroller.

Matlab: The NNs were computed by taking advantage of the matrix arithmetic functions implemented in Matlab language. Matlab was running on a Windows 7 64-bit Core i3 machine with 2 GB of RAM.

C Code on x86: The network was implemented using nested for loops and single precision floating point functions from the ANSI C library. The code was compiled using Tiny GCC Compiler and was executed on the same machine as above.

FPGA Nios II Soft Processor: the Nios II Soft Processor is a high performance microprocessor synthesizable on a FPGA platform. The processor has both hardware multipliers/dividers and a dedicated Floating Point Unit. The clock for this processor was 100 MHz. The same C code used for x86 architecture was used for this test.

Cortex M4A ARM Microcontroller: The LM4F120XL microprocessor by Texas Instruments was used as deployment platform. This microcontroller runs at 80MHz and, like the Nios II, has a dedicated Floating Point Unit. As above, the same C code for the x86 architecture was used.

The computation time for the MISO NN Core is 17us on a 50MHz clock. The computation time is the same for both SISO, MISO and MIMO architectures. The resource footprint for the design is 5492 Logic Elements, about 100kbit of memory, and 52 9-bit multipliers. The results are shown as a comparative histogram in Fig. 6.
VI. CONCLUSIONS AND FUTURE DEVELOPMENTS

An efficient algorithm to compute a Feed Forward Neural Network in an FPGA environment was presented. Since the NNs can be also utilized in strong and complex tasks [20-22], their computational load (in terms of arithmetic operations) was examined, with special interest on the performance bottleneck in the “multiply-and-accumulate” algorithm. A possible solution for this problem was proposed in the form of an algorithm. The proposed algorithm can be implemented into an architecture that fully exploits the pipeline performance of the floating-point arithmetic blocks, thus allowing a very fast computation for the neural network. The architecture was implemented on an FPGA board and the performance was tested on a MISO, a MIMO and a SISO network. As a comparison, the same networks were implemented in two high level environments (Matlab and x86 Compiled C) and two embedded environments (Nios II Soft Processor and ARM Cortex). Found results shows that the MISO NN Core largely outperforms other embedded environments even by running at a lower clock rate. The drawback of this performance is, of course, the inherent lack of flexibility of a HDL design: interfacing this neural network with a sensor/actuator array or a more complex system could be challenging. However, as shown in [7-8] [23-24], the Nios II Soft Processor can synthesize custom logic blocks in parallel to the central ALU. Once implemented this fashion, the logic can be called as a “custom instruction” of the processor. As follows, the high performance of the Core can be coupled with the flexibility of a high-level programming environment like to one used by the Nios II processor.

REFERENCES