POLITECNICO DI TORINO
Repository ISTITUZIONALE

A High-level Implementation Framework for Non-Recurrent Artificial Neural Networks on FPGA

(Article begins on next page)

19 September 2024

A High-level Implementation Framework for Non-Recurrent Artificial Neural Networks on FPGA

Luciano Prono*, Alex Marchioni[†,‡], Mauro Mangia[†,‡], Fabio Pareschi*[,‡], Riccardo Rovatti[†,‡], Gianluca Setti*[,‡]

* DET – Politecnico di Torino, corso Duca degli Abruzzi 24, 10129 Torino, Italy.
email: luciano.prono@studenti.polito.it, {fabio.pareschi, gianluca.setti}@polito.it
† DEI – University of Bologna, viale Risorgimento 2, 40136 Bologna, Italy.
email: {alex.marchioni, mauro.mangia2, riccardo.rovatti}@unibo.it
‡ ARCES – University of Bologna, via Toffano 2/2, 40125 Bologna, Italy.

Abstract—**This paper presents a fully parametrized framework, entirely described in VHDL, to simplify the FPGA implementation of non-recurrent Artificial Neural Networks (ANNs), which works independently of the complexity of the networks in terms of number of neurons, layers and, to some extent, overall topology. More specifically, the network may consist of fully-connected, max-pooling or convolutional layers which can be arbitrarily combined. The ANN is used only for inference, while back-propagation is performed off-line during the ANN learning phase. Target of this work is to achieve fast-prototyping, small, low-power and cost-effective implementation of ANNs to be employed directly on the sensing nodes of IOT (i.e. Edge Computing). The performance of so-implemented ANNs is assessed for two real applications, namely hand movement recognition based on electromyographic signals and handwritten character recognition. Energy per operation is measured in the FPGA realization and compared with the corresponding ANN implemented on a microcontroller (μC) to demonstrate the advantage of the FPGA based solution.**

I. Introduction

Nowadays many sophisticated signal processing operations are becoming more and more pervasive in everyday life, ranging from voice, language and image recognition, to extraction of bio-information from vital signals, to arrive to more complicated tasks such as robot control and assisted driving. In several of these operations, the complexity of the task to perform is so high that machine learning techniques exploiting Artificial Neural Networks (ANNs) are often the only viable solution [1], [2].

There are two main types of ANNs: feed-forward and recurrent. While recurrent ANNs, being composed by the interconnection of (first- or second-order) dynamical systems, can exhibit very reach forms of dynamic behavior and therefore (in principle) are better suited for mimicking superior brain-like tasks, their learning algorithms are today not as accurate as for the feed-forward networks [3]. Conversely, feed-forward ANNs are algebraic systems and their possible behaviors are theoretically much more simple, since their outputs depends only on the inputs given at the same time. Yet, for the latter, learning algorithms have recently proven to be able to train the ANN to perform very complex tasks [1].

By building networks of increasing dimensions today we can achieve a high level of accuracy for feed-forward ANNs.

Approaches based on the use of a large number of processors such as GPUs are quite effective but at the same time expensive and energy-inefficient. For example, Edge Computing on the node sensors of a IOT network would require low-power, cost-effective and compact processing units for signal elaboration. These requirements can be met with the implementation of small ANNs mainly in two ways: i) by programming the ANN structure using microcontrollers (μC) or ii) by describing the architecture in a hardware description language (VHDL) to be mapped on a Field Programmable Gate Array (FPGA). While the performance on μC is expected to be lower compared to FPGA, the implementation of the ANN on FPGA is much more time-consuming. In order to let the designer experiment quickly with many different implementation of the ANN at the hardware level it is therefore necessary to devise a way to quickly and easily generate high performing structures on the basis of a given set of parameters. The availability of such a framework would allow to measure not only the accuracy of many different nets but also their performance in terms of the actual implementation. Aim of this paper is to compare μC vs FPGA implementations, highlighting their pros and con towards fast-prototyping and cost-effective ANN implementation[1]. This paper is organized as follows. Section II introduces the basic mathematical framework for fully-connected and convolutional feed-forward ANNs. Section III describes how the feed-forward ANNs are being implemented in VHDL and the level of parametrization of the net. Section IV shows the performance of the architecture of Intel Cyclone 10 LP FPGA in comparison with the equivalent implemented on an ARM Cortex M4 μCs. Conclusions are finally drawn.

II. The ANN basic behavior

A complete feed-forward ANN is composed of layers of different types. The first one is called *input layer*, which is simply the input port to the ANN. All the following layers are called *hidden layers* and can be of different types as highlighted shortly. Finally, the *output layer* has the same

---

[1]The implementation does not include hardware for the back-propagation algorithm, i.e. the tool used in the training phase. The network training is to be considered an off-line task running on Matlab™ Deep Learning Toolbox™.

structure as an hidden one, but its outputs are fed directly to the output of the ANN. An example of net can be seen in Fig. 1.
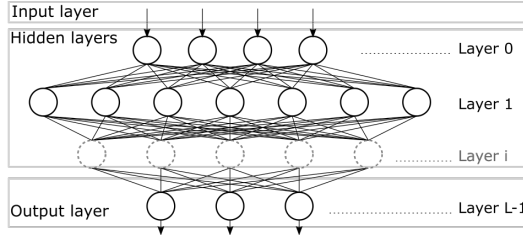

Fig. 1. Disposition of the layers of a fully-connected ANN, where L is the total number of layers minus the input layer.

A. The neuron

The neuron is the basic element of an ANN. It receives N inputs $x_j, j = 0, \dots, N-1$, multiplies each of them by a different weighting coefficient w_j (called just weight for simplicity) and adds up all the products plus a bias value b. The sum z is then given as the argument of a function called *activation function* to get the output value y. In formulas

$$\begin{cases} z = b + \sum_{j=0}^{N-1} x_j \cdot w_j \\ y = \sigma(z) \end{cases} \tag{1}$$

where $\sigma(\cdot)$ is the activation function.

The activation function can have many forms. To maintain the hardware complexity low, we will here use the Rectified Linear Unit (ReLU) $y = \max\{0, z\}$, which can be easily built using a simple comparator.

B. The fully-connected layer

The fully-connected layer has in general S neurons and N inputs: as an example in Fig. 1 layer 1 has $N = 4$ and $S = 7$. All layer inputs are connected to each neuron. As such we have N weights plus a bias for each neuron, which generate an output so that we have S outputs for the whole layer.

While this is the most general type of layer it is also the most expensive in terms of memory necessary to store the values of the weights.

C. The convolutional layer

Convolutional layers are mainly used to work with 3D data structures which are ideal for image representation: a 3D tensor can be seen as a group of matrices where each component corresponds to a pixel while each matrix describes a color between red, green or blue. The input of convolutional layers is a 3D tensor or volume with height H, width W and depth D, where the total number of inputs is $N = H \times W \times D$.

Convolutional layers use the concept of local connectivity: instead of connecting all the inputs to each neuron, only a portion of them is used at once. We define the *receptive field* as the portion of inputs which is connected to the neurons.

A single portion of inputs generates S outputs which is the number of neurons of the layer. The receptive field moves along the input volume and, for each position it reaches, the neurons generate S outputs. We define the receptive field size (F_h and F_w) and the *stride* as its vertical and horizontal shift (St_v and St_h). Additionally we use a *padding* parameter to generate a frame of zeros around the input matrices (P_t, P_b, P_l, P_r for top, bottom, left and right). For each position of the receptive field the neurons receive inputs from all the input matrices. A visual representation of this layer can be seen in Fig. 2. With the parameters given before we can evaluate the dimensions of the output volume as follows

$$\begin{cases} H_{\text{out}} = \lfloor (H - F_h + P_t + P_b)/St_v \rfloor + 1 \\ W_{\text{out}} = \lfloor (W - F_w + P_l + P_r)/St_h \rfloor + 1 \\ D_{\text{out}} = S \end{cases} \tag{2}$$

The big advantage given by this type of layer is that for each position of the receptive field we use the same weights, thus saving resources in terms of memory occupation.
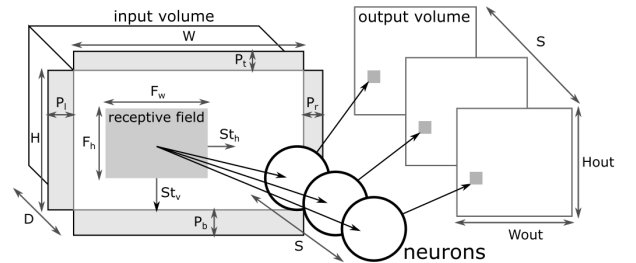

Fig. 2. Visual representation of the convolutional layer.

D. The max-pooling layer

Max-pooling layers evaluate the maximum value among the inputs and do not need weights. They work with the same set of parameters of the convolutional layers. For each position of the receptive field, the layer generates a number of output values equal to the number of input matrices. A visual representation of this type of layer can be seen in Fig. 3. The dimensions of the output matrix are evaluated as in Eq. 2, with the difference that $D_{\text{out}} = D$. This type of layer is mainly used to reduce the quantity of data to be elaborated across the ANN.
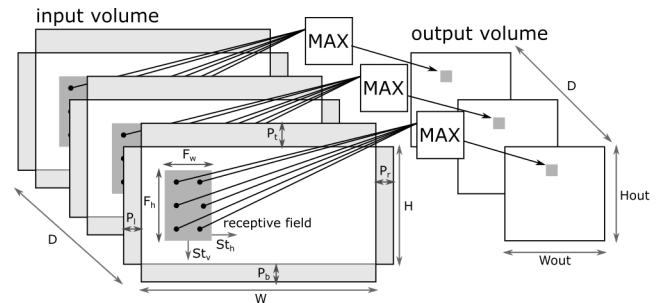

Fig. 3. Visual representation of the max-pooling layer.

III. FPGA IMPLEMENTATION OF THE ANN

The actual framework for the FPGA implementation of an ANN is entirely written in VHDL and it is completely parametrized. The designer can instantiate as many layers of any type as they want. The actual first layer is the first hidden layer of the ANN which receives the inputs from the ports of the ANN entity while the last layer sends the outputs to the output ports of the ANN entity. For the sake of performance and low-power consumption, we use only memories integrated on the FPGA chip, therefore sacrificing the possibility to build large ANNs. A state machine is employed to load the weights into the various memories of the layers.

A. The implementation of the ANN layers

Each layer is composed of an input memory, an elaboration unit and an output memory. We identify three stages:
1) the layer loads the inputs from the previous layer to the input memory, one at a time;
2) the layer elaborates the inputs and obtains the output values;
3) the layer saves the output values into the output memory, once at a time, while they are being evaluated.

Each of the three stages is controlled by its own state machine. Using a series of set-reset signals each stage communicates with the others: when the flag is high, data is ready and is being used; when the flag is low data is not used and is being overwritten.

The elaboration unit for the fully-connected and convolutional layers reproduces the neuron mathematical model by Multiply and Accumulator (MAC) operations. One MAC generates the weighted sum of a single neuron; many iterations are required to evaluate all the neurons of the layer. Multiple parallel MACs can be exploited to evaluate more than one neuron per iteration: the number of MACs employed is one of the parameters to be chosen by the designer. For a fully-connected layer, during one iteration all the N values in the input RAM are being read and used in parallel to evaluate the output of multiple neurons. For a convolutional or max-pooling layer instead only the inputs inside the receptive field are being read and its position is updated at the end of each iteration; the same weights are used until all the positions of the receptive field have been evaluated.

After each iteration, the outputs of the MACs are being sent one at a time through a block which applies the activation function and finally they are memorized in the output memory. The type of activation function employed is also a parameter chosen by the designer. The generic structure of these layers with their elaboration unit is shown in Fig. 4. With this structure it is possible to evaluate the outputs of many neurons in parallel even if the FPGA RAM structures have only one input and one output port.

The elaboration unit for the max-pooling layer is much simpler. It simply checks if the input is greater to the value in an accumulator. If it is, it saves it as the new accumulator value. When this is done for a whole pool of values, the maximum is obtained.
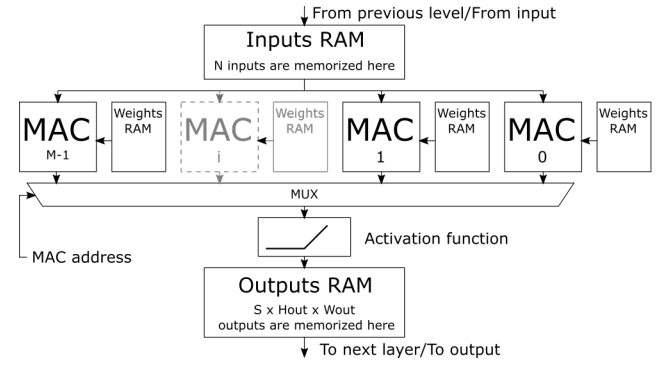

Fig. 4. Generic structure of the fully-connected and convolutional layers. The number of outputs for the fully-connected layers is only S, given that $H_{out} = 1$ and $W_{out=1}$. The MAC address selects the MAC outputs one at a time in order to save them in the output RAM. Max-pooling layers are similar but use a block for the evaluation of the maximum instead of MACs, weights and the activation function.

B. Coding a completely parametrized structure in VHDL

In order to have a completely parametrized structure it is necessary to use the VHDL to its extents. A set of functions is necessary to evaluate the many internal parameters of the net given the values wanted by the designer. For example, given the parameters of a convolutional layer, the compiler must use the Eq. 2 in order to retrieve its number of outputs.

At the same time the code must ensure maximum flexibility to the designer, letting him choose an arbitrary amount of layers, neurons and instantiated MACs. In a file we define for each layer a VHDL record structure which allows us to store different types of constants into an object. All the records are stored in an unconstrained array where every element represent one of the layers of the ANN. From the top level entity to the bottom of the structure these parameters are forwarded through generic mapping. By using the generate VHDL structures, the ANN is built based on these parameters.

The designer decides the number of bits of the data and the weights used in the ANN; these values are fixed point so the number of fractional bits also must be declared. Then they select the dimensions of the input data volume. Finally for each layer the designer chooses the type of layer, the number of neurons and instantiated MACs, the dimensions of the receptive field, the stride, the padding and the activation function type. Many of these parameters do not apply to all the types of layers so when they are not necessary they are being ignored.

IV. PERFORMANCE

We evaluate the performance of an FPGA and μC implementation of the ANN and we compare them. The μC is programmed reducing to the minimum any kind of overhead due to operations which are not strictly necessary. We use Intel Cyclone 10 LP as FPGA, while we choose the stm32l452re of the STM32 family based on ARM Cortex M4 as μC.

Testing needs suitable case studies: we test a fully-connected ANN (FCNN) for the recognition of hand movements which uses forearm electromyographic signals as inputs

TABLE I
RESOURCE ALLOCATION FOR THE ANNS ON FPGA

	FCNN 2 layers	FCNN 3 layers	CNN 3 layers
Memory bits	4%	11%	47%
M9k RAMs	13/30	23/30	16/30
9-bit multipliers	9/30	17/30	10/30
Logic Elements	16%	29%	22%

[2] and then we test a convolutional ANN (CNN) for the recognition of handwritten digits we can find in the MNIST data-set [4]. The weights employed by the ANNs are obtained beforehand using Matlab™ Deep Learning Tool™. Data and weights are 9-bit fixed point values with 5-bit fractional part.

The ANN must fit in the chosen FPGA chip which has a limited amount of resources. In particular, each layer employs one 9-bit multiplier per MAC and a number of M9k RAMs equal to $2 + \text{MACs}_{num}$. In Table I there is a summary of resource allocation for the most significant ANNs employed in the test (8 MACs/layer for FCNN and 5 MACs/layer for CNN) on the smallest chip from the Intel Cyclone 10 LP family.

With Intel Quartus Prime timing analyzer and power analyzer we obtain the maximum operative frequency and the power consumption (P) of the FPGA chip during the elaboration. By running a testbench we also obtain the average time per operation. On μC we measure t_{op} directly on device with an internal counter. The value of P is obtained from the datasheet of the μC. With these informations we estimate the energy consumption $E_{op} = t_{op} \cdot P$.

A. Fully-connected ANN performance

We instantiate an ANN composed of 2, 3 or 4 fully-connected layers and 32 neurons each with the exception of the last which has 3 neurons. There are 16 input values and 3 output values. The ANN best compromise between area and performance is with about 8 MACs per layer. In this working condition the FPGA architecture takes about 100 cycles on average to perform a complete elaboration, while the μC takes 600 cycles at his best. Additionally in Fig. 5 we can see the energy spent by the FPGA and the μC for each complete operation. With about 8 MACs per layer the FPGA energy consumption is basically minimum for each configuration and this is especially evident for a high number of layers.
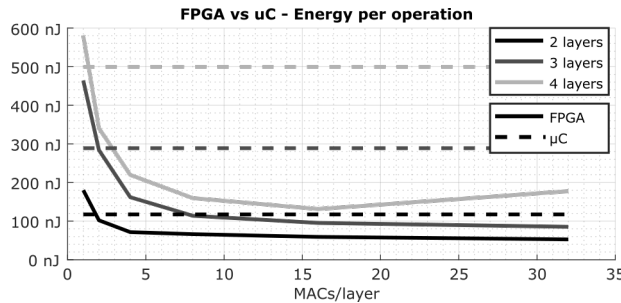

Fig. 5. Comparison between the energy per operation consumed by the FPGA and by the μC for the fully-connected ANN implementation.

B. Convolutional ANN performance

We instantiate two different setups for the convolutional ANN: setup A uses a convolutional, a max-pooling and a fully-connected layer; setup B uses two consecutive convolutional layers, then a max pooling one and finally a fully-connected layer. The input volume size is $20 \times 20 \times 1$ and there are 10 outputs, one for each digit value. All the layers (except for the max-pooling one) use 10 neurons. With 5 MACs per layer, the FPGA takes less than one tenth of the number of cycles taken by the μC to perform one complete operation. In Fig. 6 we can see the energy spent by the FPGA and the μC for each complete operation. FPGA still consumes far less than the μC but only if we keep the level of parallelism high enough.
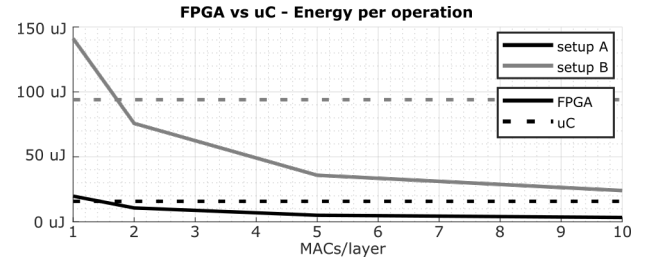

Fig. 6. Comparison between the energy per operation consumed by the FPGA and by the μC for the convolutional ANN implementation.

V. CONCLUSIONS

In this paper we have presented the design of a completely parametrized implementation of a non-recurrent ANN, whose target is fast development on FPGA. Everything is written in VHDL, improving portability of the code. The architecture consists of fully-connected, convolutional and max-pooling layers which can be combined arbitrarily. The structure does not include back-propagation algorithms in order to improve the performance of the net. With this framework a designer can try different ANN setups and measure their performance with little effort in order to find the best solution to a given problem. Simulations show that the implementation on FPGA is far better than the one on μC both from the point of view of speed and from the point of view of energy consumption.

REFERENCES

[1] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, pp. 2295–2329, Dec 2017.
[2] A. Marchioni, M. Mangia, F. Pareschil, R. Rovatti, and G. Setti, "Rakeness-based compressed sensing of surface electromyography for improved hand movement recognition in the compressed domain," in *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, pp. 1–4, Oct 2018.
[3] M. Nielsen, *Neural Networks and Deep Learning.* 2018.
[4] Y. LeCun, C. Cortes, and C. J. Burges, "The mnist database of handwritten digits website," Retrieved November 2018.