An FPGA-based accelerator for analog VLSI artificial neural network emulation

Citation for published version (APA):

DOI:
10.1109/DSD.2010.20

Document status and date:
Published: 01/01/2010

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
An FPGA-based Accelerator for Analog VLSI Artificial Neural Network Emulation

Barend van Liempd  
Department of Mixed-signal Microelectronics  
Faculty of Electrical Engineering  
Eindhoven University of Technology  
Eindhoven, The Netherlands  
Email: b.w.m.v.liempd@student.tue.nl

Daniel Herrera and Miguel Figueroa  
Department of Electrical Engineering  
Faculty of Engineering  
University of Concepción  
Concepción, Chile  
Email: {danherrera,mifiguero}@udec.cl

Abstract—Analog VLSI circuits are being used successfully to implement Artificial Neural Networks (ANNs). These analog circuits exhibit nonlinear transfer function characteristics and suffer from device mismatches, degrading network performance. Because of the high cost involved with analog VLSI production, it is beneficial to predict implementation performance during design.

We present an FPGA-based accelerator for the emulation of large (500+ synapses, 10k+ test samples) single-neuron ANNs implemented in analog VLSI. We used hardware time-multiplexing to scale network size and maximize hardware usage. An on-chip CPU controls the data flow through various memory systems to allow for large test sequences.

We show that Block-RAM availability is the main implementation bottleneck and that a trade-off arises between emulation speed and hardware resources. However, we can emulate large amounts of synapses on an FPGA with limited resources. We have obtained a speedup of 30.5 times with respect to an optimized software implementation on a desktop computer.

Keywords—Artificial neural networks, analog VLSI emulation, FPGA-based accelerators, hardware time-multiplexing, embedded systems

I. INTRODUCTION

Artificial neural networks (ANNs) have learning capabilities that are used in a variety of applications, such as face recognition, motor control, automated medical diagnosis, signal decoding and data mining. ANNs simulate biological neural networks in order to model complex relations between inputs and outputs of a network [1]. According to the perceptron neuron model, ANNs consist of simple processing elements called artificial neurons, which in turn consist of artificial synapses. Mathematically, an artificial synapse multiplies a stored weight value with an input value. To use ANNs practically, an adaptive algorithm changes the weight values contained in artificial synapses so that the network output converges over time to a desired value. The desired value can be given to the network as an input (supervised learning) or the algorithm can determine these desired values for itself (non-supervised learning). This convergence of weight values represents the previously mentioned ANN capability to learn.

Various implementation methods for ANNs exist today [2]. CPU implementations are an option, but implementations on platforms that allow for parallel processing of data are more efficient due to the parallel nature of ANNs. Furthermore, the computational-intensive nature of ANNs and their algorithms implies that even custom digital Application-Specific Integrated Circuits (ASICs) solutions become constrained by power and size limitations [3]. Mixed-signal Very-Large-Scale Integration (VLSI) circuits have shown to be a feasible way of implementing ANNs [4].

The problem with the implementation of ANNs in mixed-signal VLSI is that the analog circuits used for the implementation of the neural network suffer nonlinearities in their current-voltage transfer characteristics due to Process/Voltage/Temperature (PVT) spread (device mismatch) and the network learning performance suffers from these variations [5]. We previously compensated for these problems at the cost of chip area [4], which is not always possible. Since design and production of analog VLSI circuits has high costs and performance is degraded by these nonlinearities, it is beneficial to predict implementation performance during design. A performance prediction tool (emulator) is thus required to foresee the influence of nonlinearities and device mismatch when implementing networks on analog VLSI.

CPU implementation of such an emulator proved too slow for large networks and large input data test sets [6]. In order to speed up the implementation, various accelerator options arise due to the parallelism in large ANNs. We chose a Field Programmable Gate Array (FPGA) solution for massive parallelism, adaptivity and flexibility, which are all needed for the emulation of practical algorithms and circuits. Alternatives are Graphics Processing Unit (GPU) and Digital Signal Processor (DSP) solutions, which provide higher processing speeds, but do not offer the parallelism of an FPGA [7].

In this paper, we present an implementation of an FPGA-based emulator for analog VLSI large-size single-neuron ANNs to analyze and predict performance of operation for such implementations. Furthermore, we investigate the limits imposed by FPGA resources. We focus on the emulation of a single neuron consisting of a set of artificial synapses as a starting point, because neurons are the basic building blocks.
of larger, more complex neural networks.

The first hardware implementation for the emulator [8] showed that limited FPGA resources are the main bottleneck for the amount of synapses that can be emulated. In the current implementation, we define a new approach involving hardware re-use that allows us to overcome this limitation. We give a detailed explanation on this technique in Subsection III-A. Furthermore, we operate external memory systems to enable emulation of large data sets and use a CPU for data processing.

The difference between this work and other work [9], [10] is that other work is focused on the implementation of ANNs in terms of speed and power usage, while we focus on prediction of the influence of nonlinearities in analog VLSI ANNs. To the best of our knowledge, no other work performing emulation of ANNs implemented on analog VLSI with an FPGA has been published to date.

The rest of this paper is organized as follows. First, we present background information on neural networks implemented on analog VLSI and the emulation of such implementations. Then, we present the proposed hardware time-multiplexing technique and the accelerator implementation in Section III. In Section IV, we verify correct operation and present resource details. Finally, we draw conclusions in Section V.

II. BACKGROUND

Our emulator emulates ANNs implemented in analog VLSI. As already noted, we focus on single-neuron ANNs. Furthermore, we focus on the implementation of the Least Mean Squared (LMS) algorithm as a proof of concept. Before exploring the implementation of the emulator, we present brief mathematical concepts for a single neuron in Subsection II-A. Also, we briefly present the transfer functions of the analog VLSI circuits we used to implement artificial synapses and emulation techniques for these transfer functions in Subsection II-B.

A. Neuron model

A single neuron with $M$ synapses is modeled to have a transfer function as shown in (1). Here, $y_k$ denotes the neuron output value for sample $k$. Furthermore, $w_{i,k}$ denotes the weight value and $x_{i,k}$ denotes the input value for synapse $i$ and sample number $k$.

$$y_k = w_k \cdot x_k = \sum_{i=1}^{M} w_{i,k} \cdot x_{i,k}$$

(1)

(2) shows the supervised LMS weight update rule. The weight updates for sample $k+1$ denoted $\Delta w_{k+1}$ are calculated from synapse inputs and neuron output data at sample $k$. Here, $d_k$ is the desired neuron output, representing the supervisor of the algorithm. $\mu$ is a constant denoting the learning rate which controls learning speed and resolution.

$$\Delta w_{k+1} = \mu \cdot x_k \cdot (d_k - y_k)$$

$$w_{k+1} = w_k + \Delta w_{k+1}$$

(2)

For more detailed background information on neural networks see [1].

B. Emulation of analog VLSI hardware

An artificial synapse in analog VLSI consists of a multiplier circuit and a memory circuit to store the weight value of a synapse. In previous work, Figueroa et al. produced 64 synapses in a $0.35\mu m$ CMOS process [11], using a digital implementation of the LMS algorithm and Pulse-Width Modulator (PWM) [4]. A Gilbert cell topology is used as a current multiplier [12], [13]. A floating-gate pFET transistor is used to implement analog memory in CMOS [14]. See Figure 1 for the nonlinear transfer functions in the produced analog circuits. In the following paragraphs, the emulation of the analog multiplier and memory cells will be detailed.

1) Multiplier cell: A Gilbert cell multiplier has a transfer function as given in (3) [12]. Here, $I_{out}$ is the output current,
$I_0$ is the multiplier cell saturation current, $U_T$ is the thermal voltage and $V_{w,k}$ are the inputs of the multiplier. Figure 1(a) shows nonlinearities for eight analog multipliers for varying weight voltages, keeping the input voltage constant.

$$I_{out}(t) = I_0 \cdot \tanh \frac{V_w(t)}{2U_T} \cdot \tanh \frac{V_z(t)}{2U_T}$$

(3)

In order to emulate an analog multiplier cell, we measured the transfer functions of the Gilbert cells by increasing $V_w$ with constant step size. We then fitted these measurements with tanh-curves (4). Here, the sampled versions of $I_{out}(t), V_w(t), V_z(t)$ are denoted as $y_k, w_k, x_k$, respectively. Furthermore, $A_{w,k}, B_{w,k}$ and $C_{w,k}$ are the fitting parameters which represent the analog VLSI transfer functions. Finally, the $\tanh$ is approximated with the first order Taylor approximation, which is sufficiently precise for small $x_k$. $X_{x,k}$ and $Y_{x,k}$ are the first order Taylor approximation parameters.

$$y_k(w_k, x_k) \approx A_{w,k} \cdot \tanh(B_{w,k} \cdot x_k) + C_{w,k}$$

$$\approx A_{w,k} \cdot (X_{x,k} \cdot B_{w,k} \cdot x_k + Y_{x,k}) + C_{w,k}$$

(4)

2) Memory cell: The output voltage $V_w$ of an analog VLSI memory cell depends linearly on the amount of electrons stored on the floating gate as can be seen from Figure 1(b). However, each memory cell transfer function has a different slope due to PVT spread [11]. The amount of weight change per electron pulse is denoted $\text{slope}_{TF}$. The amount of electron pulses (denoted $N_{\text{pulses},k}$), needed for the required voltage change $\Delta w_k$, is calculated by the algorithm using (5). The memory cell weight voltage is changed by adding the pulses to the floating gate using the digitally implemented PWM.

$$N_{\text{pulses},k} = \frac{\Delta w_{\text{real},k}}{\text{slope}_{TF}}$$

(5)

To emulate the memory cell, we change the stored weight value relative to the change in the number of pulses in the analog implementation. Mathematically, each memory cell changes the weight value $w_k$ according to the weight change $\Delta w_k$ calculated by the algorithm. In (6), the required weight change is then given by $\Delta w_{\text{real},k}$. The approximated version of this weight change is denoted $\Delta w_{\text{approx},k}$. The remainder $R_k$ represents the difference between the required weight change and the calculated weight change, which arises since a finite amount of steps are used to represent the linear transfer function.

$$\Delta w_{\text{real},k} = \Delta w_k + R_{k-1}$$

$$\Delta w_{\text{approx},k} = N_{\text{pulses},k} \cdot \text{slope}_{TF}$$

$$R_k = \Delta w_{\text{real},k} - \Delta w_{\text{approx},k}$$

(6)

III. IMPLEMENTATION

In this section we show how the mathematical descriptions from the previous section are mapped to an implementation on a Xilinx Virtex 2 Pro (V2P) FPGA. First, we explain our proposed hardware re-use technique in Subsection III-A. Secondly, we describe system data flow and hardware/software interaction in Subsection III-B. Finally, we give an overview of the emulator hardware and describe the hardware implementation in Subsection III-C.

A. Temporal synapse slicing

As noted in the introduction, we propose to solve resource constraints through the re-use of hardware blocks to enable emulation large-size networks. We will from now on refer to this technique as temporal synapse slicing. A temporally sliced synapse consists of an emulated multiplier cell, an emulated memory cell and control hardware, which together emulate the function of one artificial synapse as implemented on analog VLSI. We re-use this single temporally sliced synapse, physically implemented on the FPGA, to emulate multiple artificial synapses over time. A temporal slice refers to all temporally sliced synapses in the system together at a single point in time. For example, the first slice are all physical synapses operating to emulate the first artificial synapse they represent. A combination of a number of slices and a number of temporally sliced synapses creates a neuron. For example, when we operate 5 slices with 5 physically implemented temporally sliced synapses, we emulate a 25-synapse neuron. See figure 2. For each sample, all sliced synapses are operated sequentially: one synapse is operated $M \times K$ times.

Figure 3 shows (schematically) an example of the neuron structure as it arises through the use of the synapse slicing. In this example we show $M$ slices and 2 temporally sliced synapses per slice to form a 2$M$-synapse neuron. Each synapse contains a memory cell emulator, a multiplier cell emulator and a slice adder. The synapse outputs $y_{1,k}$ and $y_{2,k}$ are intermediate slice results, added by a third adder to form the total network output $y_k$. Furthermore, the input data block shown in the figure contains a set of $K$ input samples for each network input. One sample is fed to the network inputs in parallel and samples follow each other sequentially, until the whole data block has been processed. Finally, the algorithm processes network information to calculate weight updates, using the LMS algorithm as given in (2).
Note that, to explain the principle, only 2 temporally sliced synapses are shown. In practice, as much hardware synapses would be implemented on an FPGA as possible, with the amount of slices required to scale the network to the required size. The amount of synapses contained within the single-neuron ANN is the multiplication of the amount of slices ($M$) with the amount of temporally sliced synapses. In theory, this allows for the emulation of thousands of artificial synapses. However, since execution time increases linearly when more slices are used, a practical limit exists and a trade-off arises between hardware usage, time usage and network size.

**B. Data flow and HW/SW interaction**

To allow for large tests ($\geq 10k$ samples) to be performed with the emulator, sufficiently large memory systems are required. We used a Xilinx University Program (XUP) test board with a Xilinx Virtex 2 Pro (V2P) FPGA for the implementation of the emulator system [15]. It is equipped with a flash memory socket (up to 4GB), an SDRAM socket (up to 512MB) and on-chip Block-RAM (BRAM, 2448Kb) available for data storage. BRAM is too small to store large data sets. To maximize the amount of input test data, we divide data in smaller blocks before copying the data to memories with lower capacity. SDRAM access times of are higher than BRAM access times, but there is a need for large data sets for which BRAM is not suited. A PowerPC (PPC) on-chip CPU executes the data flow, dividing data samples into data blocks. We have used the Xilinx Embedded Development Kit (EDK) HW/SW development platform to implement the system.

Figure 4 shows the data flow as we have implemented it on the CPU. First, the PPC handles samples one-by-one, copying them from the flash memory (denoted ‘FL.’ in the figure) and storing them on SDRAM. Samples are divided in data blocks, which are processed by emulator hardware before the output is finally written to the flash memory. Data stored on flash memory is not divided into smaller blocks which fit SDRAM, first due to simplicity and second because a 512MB SDRAM can already store around 134 million 32-bit samples, which is sufficient for most development or research applications. We thus use the flash system only to provide a practical means to copy data from and to the FPGA.

The PPC is required to interact with the emulator hardware and monitor progress until operation is finished. We used a handshaking protocol using Software-Accessible Registers (SARs) to implement hardware/software interaction. SARs are reserved registers with read/write accessibility for both the CPU and the FPGA hardware. The term SAR has been coined by Xilinx.

We will from now on refer to the handshaking protocol as polling. It operates as follows. First, the custom hardware enables the PPC through a SAR. Then the processor continuously reads out and compares the value contained within a second SAR, while the hardware is operating. Finally, the emulator hardware is disabled by the CPU when the hardware sets a ready flag. There are some disadvantages to the polling approach. One disadvantage is that the processor cannot perform other tasks while waiting for the emulator hardware to finish. A second disadvantage is that the PPC wastes power while the emulator hardware operates. However, no other tasks but data management are required from the processor in this implementation. Advantages of the polling protocol include small resource requirements, omission of a set-up time and ease of implementation. Furthermore, the V2P includes a second PPC processor to perform other tasks if it would be required.

**C. Emulator hardware design**

The emulator hardware is shown schematically in Figure 5. First, the process controller is shown, which is a Finite State Machine (FSM) that starts/monitors all FSMs within other hardware blocks and controls two counters.
Secondly, the Synapses block contains all temporally sliced synapses. Thirdly, the Algorithm block consists of a number of pipelined hardware multipliers, calculating all weight update values. Finally, the Addition block is simply an adder for synapse outputs. In subsequent parts of this section, hardware designs for each block will be detailed.

1) Process controller: The process controller is the main FSM for the hardware system. This controller implements the slicing system. The FSM diagram is shown schematically in Figure 6. The FSM starting state is denoted Idle and the FSM ending state is denoted Datablock ready. The flags involved between states are indicated in the figure. For example, in state Await Alg, the ready signal is the ready flag from the algorithm, signaling the process controller it is ready operating.

First, the CPU enables the FSM through a SAR. Then, the multipliers retrieve the value stored in the memory and multiply it with the first input sample for all slices (denoted slice loop 1). Then, outputs w and y are stored in an output BRAM. The algorithm is started to calculate the weight updates. When ready, the memory cells are started, updating their weights to contain the new weight value calculated by the algorithm (denoted slice loop 2). The system executes the described process for all samples contained in the data block (denoted sample loop), after which a ready flag is set in a SAR, notifying the CPU that the hardware is ready for the next data block.

Throughout this process, the sample and slice index values are updated to allow correct data selection from the input data BRAM and control of the amount of loops in the process FSM. The relative simplicity of the process controller FSM shows that our proposed temporal slicing technique can be used in practical systems. By increasing the amount of slice loops, we can linearly scale up the neuron size.

2) Temporally sliced synapse: The hardware block emulating the artificial synapse consists a multiplier cell emulator and a memory cell emulator. Also, it contains a slice BRAM. The slice BRAM contains for each slice in the synapse:

- the memory cell BRAM index \( n_{k-1} \) pointing to the current weight;
- the \( \Delta w \) for the next sample;
- the remainder value \( R_{k} \) arising from weight approximation.

3) Multiplier cell: The multiplier cell is used to calculate the \( y_{k} \) value for a synapse, as seen in (4) (subsection II-B1). The hardware consists of two BRAMs, a single pipelined hardware multiplier and an FSM controller.

The first BRAM contains \( A_{w,k} \), \( B_{w,k} \) and \( C_{w,k} \) for \( N \) measurements for one analog multiplier cell transfer function. Each multiplier cell emulates one of the measured analog VLSI multiplier cells. A disadvantage to this way of implementing the slicing system is that it does not account well for PVT spread if the amount of slices is much higher then the amount of temporally sliced synapses. Therefore, it is essential that as much temporally sliced synapses as possible are implemented. The second BRAM contains \( X_{x,k} \) and \( Y_{x,k} \) for \( M \) approximation points, so that a higher \( M \) gives a higher resolution of the \( \tanh \) approximation.
approximate is an odd, inversely symmetrical function, we calculate the absolute value of \( X_{x,k} \cdot B \cdot x_i + Y_{x,k} \) result to obtain double resolution with the same memory space.

4) **Memory cell:** The memory cell design consists of a Look-Up Table (LUT) for weight values, a divider and an FSM controller. The LUT stores the weight value at each index corresponding to the multiplier cell transfer function BRAM index for each weight value. The divider topology we used for our design uses normalization and shift-registers, producing sufficient accurate results for our application. It has been devised by Kilts et al. [16]. The implementation details for the divider are omitted here. Due to pipelining, no extra hardware multipliers are used by the divider after implementation.

The memory FSM controller operates as follows. When the FSM is started, the input signals are ready to be divided, so that we first start the divider. When division is finished, we use the result to calculate the new index and \( \Delta w_{approx} \). Then, we check if the BRAM limits are exceeded. If the limits are exceeded, the \( \tanh \) function reaches the maximum value and we require the last value, so that we set the index to the last value. Otherwise, we maintain the index value which was calculated, we extract the weight, we store the remainder and index for the next operation and set the memory cell to ready/idle. The index value for the weight value is used by the multiplier cell emulator so that the correct transfer function characteristics can be extracted to calculate the output value for the artificial synapse.

5) **Algorithm:** The LMS algorithm block implements the weight update calculation as previously given in (2). Note that we are not required to approximate the algorithm function since the algorithm is implemented in digital hardware in a mixed-signal neural network implementation. The algorithm block is executed for each slice, every time changing the weight value to the data saved in slice memory.

We assume that the learning rate, \( \mu \), changing the weight value to the data saved in slice memory. The algorithm block is executed for each slice, every time hardware in a mixed-signal neural network implementation. Note that we are not required to approximate the algorithm function since the algorithm is implemented in digital hardware in a mixed-signal neural network implementation. The algorithm block is executed for each slice, every time changing the weight value to the data saved in slice memory.

The results produced by both systems are identical. Figure 7 shows weight evolution for the emulated 5-synapse neuron. The weight values converge towards the same fixed values we set when generating the reference data. Furthermore, the simulation results were verified to be identical with our previous CPU implementations of the emulator.

A. **Simulation: emulation of a small test network**

We simulated the systems as follows. First, we generated a random input data set of 1024 samples, which is small enough to fit also on the basic system, where only BRAM is available to store the input data. Then, using the generated data set, we produced a reference output signal \( (d_k) \) by fixing the weights of a 5-synapse neuron. For both systems, we then implemented 5 synapses in the emulator using 5 (randomly selected) measurement data sets from the total set of 64 analog circuits. Finally, we then simulated both systems using the previously generated inputs in the Xilinx ISE Simulator. Note that we configured the slicing system to use 1 slice and (the same selection of) 5 hardware synapses to mirror the basic system.

The results produced by both systems are identical. Figure 7 shows weight evolution for the emulated 5-synapse neuron. The weight values converge towards the same fixed values we set when generating the reference data. Furthermore, the simulation results were verified to be identical with our previous CPU implementations of the emulator. Next to convergence of the weights for a single slice, we also simulated the process controller to verified its operation, also for multiple slices. Multipliers first operate once for all slices. Then, for each slice subsequently, first the algorithm and then the memory cell are operated. This operation order
is in accordance to the intended operation as previously shown in Figure 6.

B. Required hardware resources

Table I shows the required FPGA resources for a single synapse (denoted 1S), a 5-synapse slicing system implementation (denoted 5S+CPU) and for the required resources when only the CPU, data bus and program memory are implemented (denoted CPU). All figures are given as reported after XST optimization.

![Table I](attachment:image.png)

In Table II we compared the available resources for 2 Xilinx FPGAs, the Virtex 2 Pro (V2P-XC2VP30) and the Virtex 6 (V6-XC6VSX475T). A comparison is made regarding the amount of on-chip Multipliers, BRAMs, Logic Cells and the maximum amount of artificial synapses that can be implemented. The V2P is a small FPGA, while the V6 is the currently available Xilinx FPGA equipped with the most BRAM. Note that the Virtex 6 is not equipped with a PowerPC, so that a processor such as the Xilinx MicroBlaze design would be required in order to operate the emulator. We do not take this into account in our comparison. Furthermore, we neglect the increase in BRAM required to store the input data, which becomes larger when more synapses are implemented.

![Table II](attachment:image.png)

We conclude that BRAM availability is the main bottleneck, allowing for the implementation of 12 and 140 artificial synapses on respectively the Virtex 2 Pro and Virtex 6 FPGAs. FPGAs with a larger amount of hardware resources require a lower amount of slices to emulate 500+ synapses, offering higher processing speed compared to entry-level chips.

C. Timing evaluation

After the Place And Route (PAR) process, we analyzed the system timing details for a single-neuron analog VLSI emulator with 2 slices on the Xilinx Virtex 2 Pro. Where possible, we measured in-operation timings to verify the simulated values. All post-implementation system delays are simulated to be <10ns, such that the network can operate using the Virtex 2 Pro maximum clock of 100MHz.

When we implement the algorithm using 5 hardware multipliers (one per synapse), the hardware requires 16, 14 and 7 cycles per sample for respectively the multiplier, memory and algorithm blocks. In total, the process controller requires 100 cycles/sample, which results in a processing speed of 1MSps (Samples per second) for the 5-synapse, 2-slice slicing system (CPU cycles required to load data blocks from flash and SDRAM memory systems to the hardware not included). An average 50 cycles/slice/sample is thus required.

To the best of our knowledge, our work publishes the first accelerator for this application. As such, we were not able to make state of the art comparisons. However, in comparison to our previous CPU implementations, a significant speed-up has been obtained. We compared the operation times (in cycles/sample) for our FPGA accelerator (5 synapses, 1 slice, 100MHz clock speed) with both a Matlab- and C-implementation of the emulator (5 synapses, 2GHz clock speed). To this end we did the same 1024-sample test on all implementations. Our emulator (0.5 µs/sample) shows a speed-up of 5400× (2700 µs/sample) and 30.5× (15.1 µs/sample) in comparison to the Matlab- and C-implementation, respectively. It is expected that a V6 FPGA would perform 2-3 magnitudes better than the optimized C-implementation due to the higher clock speed and the higher maximum number of artificial synapses that can be implemented.

We conclude that temporal synapse slicing allows for a trade-off between time and hardware resources, which was previously not available. To visualize this, we show a number of possible configurations when 100 artificial synapses need to be implemented and their costs in terms of time and hardware resources in Table III.

![Table III](attachment:image.png)
V. Conclusions

The contributions of this work are twofold. We implemented an FPGA-based accelerator for practical emulation of analog VLSI neural networks and investigated the limits that availability of FPGA resources impose on the amount of synapses that we can emulate. First, we conclude that emulation of large analog VLSI neural networks is feasible on an FPGA platform. Secondly, we conclude that availability of on-chip memory limits the amount of test samples, but external memory systems overcome this limitation.

Our emulator allows for emulation of nonlinearities of analog VLSI implementations for artificial neural networks. The emulator enables convergence and performance analysis of large single-neuron ANNs. We show that it is possible to implement 500+ synapses even on an entry-level FPGA with limited resources. We use hardware efficiently through temporally slicing of synapse emulator blocks and show there is a trade-off between resources and emulation speed. Furthermore, we show that external memory systems and a CPU for data flow control together overcome the limitations posed by available on-chip memory regarding the amount of input samples, allowing for test sequences of more than 10K samples. Finally, our Virtex 2 Pro accelerator obtains a speedup of the order of one magnitude compared to a specialized software implementation, while it is expected that a similar implementation on a state of the art FPGA such as the Virtex 6 could obtain a speedup of 2-3 magnitudes.

Future work aims at emulation of multiple layer/neuron networks and the use of more complex algorithms such as Independent Component Analysis (ICA). Also, the current implementation is not user friendly and the user requires knowledge of the inner workings to modify the architecture. For future work, we want to create a user friendly emulator tool which can be used by designers of mixed-signal VLSI and researchers on ANNs in the field. This includes tools for data generation, implementation and network analysis. We will work to enable implementation of different algorithms and circuits by changing equations and measurement data, respectively.

Acknowledgments

This work was partially funded by the Chilean government through grants Fondecyt-1070485 and PFB-0824. Furthermore, this work was partially funded through the Erasmus Mundus External Cooperation Window (EMECW) program from the European Commission.

References