A Fast FPGA Implementation of a General Purpose Neuron

Valentina Salapura, Michael Gschwind, Oliver Maischberger
{*vanja,mike,oliver*}*@vlsivie.tuwien.ac.at*

Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3-182-2
A-1040 Wien
AUSTRIA

Abstract. The implementation of larger digital neural networks has not been possible due to the real-estate requirements of single neurons. We present an expandable digital architecture which allows fast and space-efficient computation of the sum of weighted inputs, providing an efficient implementation base for large neural networks. The actual digital circuitry is simple and highly regular, thus allowing very efficient space usage of fine grained FPGAs. We take advantage of the re-programmability of the devices to automatically generate new custom hardware for each topology of the neural network.

1 Introduction

As conventional computer hardware is not optimized for simulating neural networks, several hardware implementations for neural networks have been suggested ([MS88], [MOPU93], [vDJST93]). One of the major constraints on hardware implementations of neural nets is the amount of circuitry required to perform the multiplication of each input by its corresponding weight and their subsequent addition:

$$n_i\left(x_1, \ldots, x_{m_i}\right) = a_i\left(\sum_{1 \leq j \leq m_i} w_{ji} * x_j\right),$$

where x_j are the input signals, w_{ji} the weights and a_i the activation function.

The space efficiency problem is especially acute in digital designs, where parallel multipliers and adders are extremely expensive in terms of circuitry [CB92]. An equivalent bit serial architecture reduces this complexity at the cost of net performance, but still tends to result in large and complex overall designs.

We decided to use field-programmable gate arrays (FPGAs) to develop a prototype of our net [Xil93]. FPGAs can be reprogrammed easily, thus allowing different design choices to be evaluated in a short time. This design methodology also enabled us to keep overall system cost at a minimum. Previous neural network designs using FPGAs have shown how space efficiency can be achieved

[vDJST93], [GSM94], [Sal94]. The neuron design proposed in this paper makes a compromise between space efficiency and performance.

We have developed a set of tools to achieve complete automatization of the design flow, from the network architecture definition phase and training phase to the finished hardware implementation. The network architecture is user-definable, allowing the implementation of any network topology. The chosen network topology is described in an input file. Tools automatically translate the network's description into a corresponding net list which is downloaded into hardware. Network training is performed off-chip, reducing real estate consumption for two reasons:

- No hardware is necessary to conduct the training phase.
- Instead of general purpose operational units, specialized instances can be generated. These require less hardware, as they do not have to handle all cases. This applies especially to the multiplication unit, which is expensive in area consumption terms. Also, smaller ROMs can be used instead of RAMs for storing the weights.

As construction and training of the neural net occurs only once in an application's lifetime, namely at its beginning, this off-chip training scheme does not present a limitation to a net's functionality. Our choice of FPGAs as implementation technology proved beneficial in this respect, as for each application the best matching architecture can be chosen, trained on the workstation and then down-loaded to the FPGA for operational use.

2 Related Work

The digital hardware implementations presented in literature vary from bit-stream implementations, through bit-serial and mixed parallel-serial implementations to fast, fully parallel implementations.

The pulse-stream encoding scheme for representing values is used in an analog implementation by Murray and Smith [MS88]. They perform space-efficient multiplication of the input signal with the synoptic weight by intersecting it with a high-frequency chopping signal.

van Daalen et al. [vDJST93] present a bit-stream stochastic approach. They represent values v in the range $[-1, 1]$ by stochastic bit-streams in which the probability that a bit is set is $(v + 1)/2$. Their input representation and architecture restrict this approach to fully interconnected feed-forward nets. The non-linear behavior of this approach requires that new training methods be developed.

In [GSM94], we propose another bit-stream approach. Digital chopping and encoding values v from the range $[0, 1]$ by a bit stream where the probability that a bit is set is v are used. Using this encoding, an extremely space efficient implementation of the multiplication can be achieved. In this design, only 22 CLBs [Xil93] are required to implement a neuron. This method enables the

construction of any network architecture, but constrains applications to those with binary threshold units.

The approach in [Sal94] is based on the idea to represent the inputs and synaptic weights of a neuron as delta encoded binary sequences. For hardware implementation delta arithmetic units are used which employ only one-bit full adders and D flip-flops. The performance of the design is improved and some real-estate savings are achieved. The design can be used for assembling of feed-forward and recursive nets.

GANGLION [CB92] is a fast implementation of a simple three layer feed forward net. The implementation is highly parallel achieving performance of 20 million decisions per second. This approach needs 640 to 784 CLBs per neuron, making this implementation extremely real estate intensive.

3 The Neuron

Each processing unit computes a weighted sum of its inputs plus a bias value assigned to that unit, applies an activation function, and takes the result as its current state. The unit performs the multiplication of 8 bit unsigned inputs by 8 bit signed integer weights forming a 16 bit signed product. The eight products and a 16 bit signed unit-specific bias are accumulated into a 20 bit result. The final result is computed by applying an arbitrary activation function. This process scales the 20 bit intermediate result stored in the accumulator to an 8 bit value (see figure 1).

We use the fact that multiplication is commutative, and instead of multiplying the input values with the weight, we multiply the (signed) weight with the (positive) input values. Thus, multiplication is reduced to multiplying a signed value by an unsigned value. This can be implemented using fewer logic gates.

Multiplication is performed by using the well-know shift and add algorithm. The first synapse weight is loaded into the 16 bit shift register from the weight ROM, and the synapse input in the 8 bit shift register. Then, the shift and add multiplication algorithm is performed, using a 20 bit accumulator.

After eight iterations, the first multiplication $w_{ji} * x_j$ has been processed. To process the next neuron input, the input and weight values for the next multiplication are loaded into their respective shift registers and the process starts over. At the same time, the accumulator is used for implementing the accumulation of the multiplication result and adding the results of all eight multiplications.

After the result $\sum_{1 \leq j \leq m_i} w_{ji} * x_j$ has been computed, the activation function is applied to this intermediate result. Depending on the complexity of the activation function, this can take 0 or more cycles. This activation function also scales the intermediate result to an unsigned 8 bit output value. This output value is either the final result or fed to a next layer neuron.

As the constructed unit can have at most eight inputs and as the multiplication of one input requires eight cycles, a new computation cycle is started every 64 cycles (plus the time used for computing the activation function). This
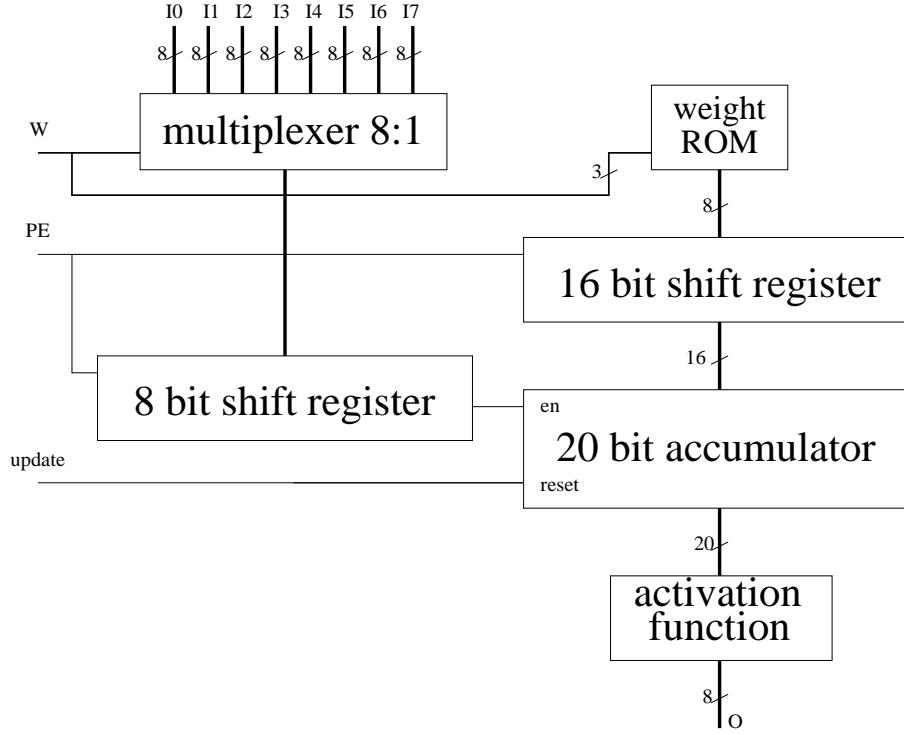
Fig. 1. Schematic diagram of a neuron.

condition is checked by a global counter, and distributed to all neurons. Upon receiving this signal, the neurons will latch their input state into an output register, load the bias into the accumulator and start a new computation.

51 CLBs are used for implementing the base neuron. Depending on the complexity of the activation function used, additional CLBs may be necessary to implement look-up tables or other logic. The `ppr` tool [Xil92] reports the following design data for a single neuron:

Packed CLBs	51
FG Function Generators	102
H Function Generators	16
Flip Flops	44
Equivalent "Gate Array" Gates	1458

4　The Overall Network Architecture

The design of the neurons is such that any neural architecture can be assembled from single neurons. Users can choose an optimal interconnection pattern for

their specific application, as these interconnections are performed using FPGA routing. This neuron design can be used to implement a wide range of different models of neural networks whose units have binary or continuous input and unit state, and with various activation functions, from hard-limiter to sigmoid. The implementation of both feed-forward networks and recursive networks [Hop82], [Koh90] is possible.

Any network can be implemented using the proposed units. The design includes a global synchronization unit which generates control signals distributed to the whole network. Figure 2 shows a feed-forward network with four neurons in the input layer, four neurons in the hidden layer and two neurons in the output layer.
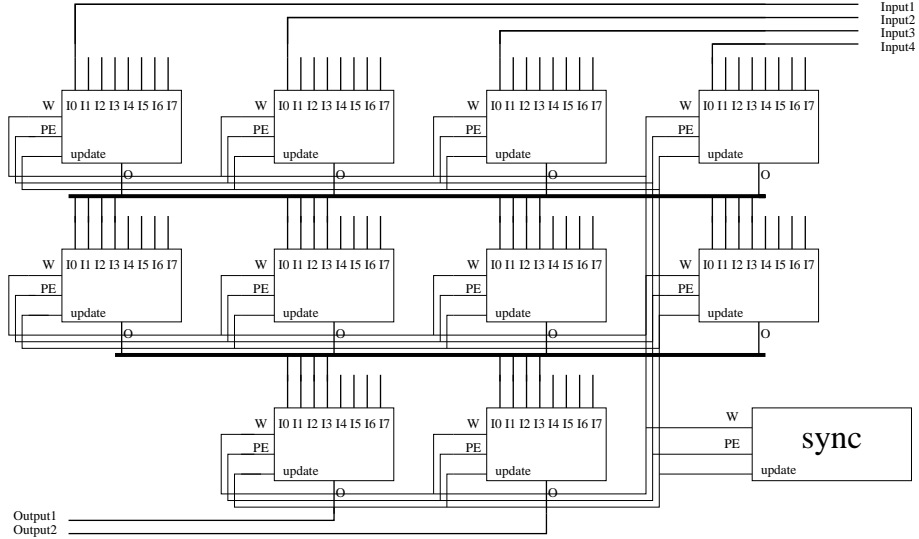

Fig. 2. Example architecture: a feed-forward network with ten neurons.

Several neurons can be placed on one FPGA. The exact number of neurons fitting on one FPGA depends on the exact FPGA type and the complexity of the activation function. By using multiple FPGAs, arbitrarily large, complex neural nets can be designed cheaply and efficiently. Having neurons as indivisible functional units allows absolute freedom in choosing any topology required.

5 Automation of the Design Process

To design a network for a new application, a new network topology is selected. On this network, the training process is performed, yielding a set of new weights and biases. These new connections, weights and biases have to be mapped to the

logic of the LCAs. Embedding these parameters into the LCAs alters the routing within the LCAs. To customize the base LCA design for each new application, we have developed tools that enable the fully automation of the designing process. The arbitrarily network topology with trained weights is described in an input file. Complete translation into LCAs and design optimization is then performed automatically, entirely invisible to the user.

```
I    I1
I    I2
N    S0N0        0
C    I1    GND   GND   GND   GND   GND   GND   GND
W    126   0   0   0   0   0   0   0
N    S0N1        0
C    I2    GND   GND   GND   GND   GND   GND   GND
W    126   0   0   0   0   0   0   0
N    S1N0        192
C    S0N0 S0N1 GND   GND   GND   GND   GND   GND
W    63    63   0   0   0   0   0   0
N    S2N0        64
C    S0N0 S0N1 GND   GND   GND   GND   GND   GND
W    63    63   -126   0   0   0   0   0
O    S2N0
```

Fig. 3. Example input file: a feed-forward network with four neurons.

The input file contains all parameters needed. For illustration, a simple input file is shown in figure 3. It describes a small network with two inputs, two neurons in the first, one neuron in the second and third layers and one output. At the beginning of the file inputs are specified (denoted with I), assigning a name to every input. Then, the neurons are described. The order of neurons in the file is irrelevant. Every neuron is defined with four parameters. Firstly, a name is assigned to every unit. Then, the bias value assigned to the unit is given. After that, the connections are specified: for each of the eight neuron inputs, the name of the input to the network or the name of the unit with which to connect is given. If an input of the unit is unused, it is connected to GND. Finally, the corresponding weights (signed integers) are given. At the end of the file, the list of the outputs is defined, containing the names of the units whose output should be used as outputs of the network.

After the network has been defined and trained, our tool set generates a configuration net list for the FPGA board. The configuration bit-stream is used to initialize the Xilinx FPGAs. Figure 4 shows the phase model for the design of a neural net from training to hardware operation.
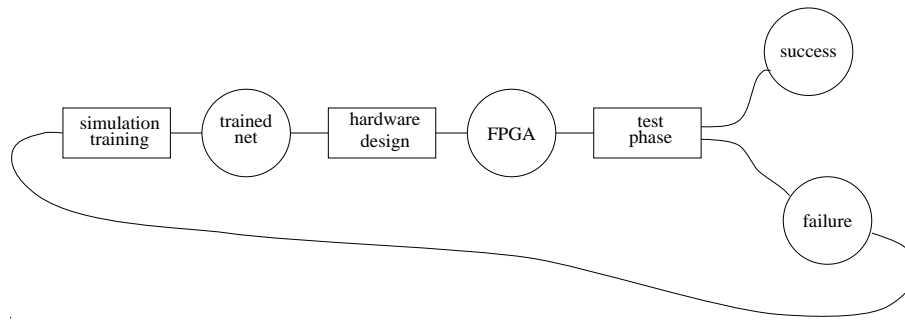
Fig. 4. Phase model of net development

6 Conclusion

We propose a space-efficient, fast neural network design which can support any
network topology. Starting from an optimized, freely interconnectable neuron,
various neural network models can be implemented.

The simplicity of the proposed neuron design allows for the massive repli-
cation of neurons to build complex neural nets. FPGAs are used as hardware
platform, facilitating the implementation of arbitrary network architectures and
the use of an off-chip training scheme.

Tools have been developed to completely automate the design flow from
the network architecture definition phase and training to the final hardware
implementation.

References

[CB92]   Charles E. Cox and W. Ekkehard Blanz. GANGLION – a fast field-
          programmable gate array implementation of a connectionist classifier. *IEEE
          Journal of Solid-State Circuits*, 27(3):288–299, March 1992.
[GSM94]   Michael Gschwind, Valentina Salapura, and Oliver Maischberger. Space
          efficient neural net implementation. In *Proc. of the Second International
          ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, Berkeley,
          CA, February 1994. ACM.
[Hop82]   John J. Hopfield. Neural networks and physical systems with emergent col-
          lective computational abilities. In *Proceedings of the Academy of Sciences
          USA*, volume 79, pages 2554–2558, April 1982.
[Koh90]   Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*,
          78(9):1464–1480, September 1990.
[MOPU93] Michele Marchesi, Gianni Orlando, Francesco Piazza, and Aurelio Uncini.
          Fast neural networks without multipliers. *IEEE Transactions on Neural
          Networks*, 4(1):53–62, January 1993.
[MS88]   Alan F. Murray and Anthony V. W. Smith. Asynchronous VLSI neural
          networks using pulse-stream arithmetic. *IEEE Journal of Solid-State Cir-
          cuits*, 23(3):688–697, March 1988.

[Sal94]   Valentina Salapura. Neural networks using bit stream arithmetic: A space efficient implementation. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, London, UK, June 1994.

[vDJST93] Max van Daalen, Peter Jeavons, and John Shawe-Taylor. A stochastic neural architecture that exploits dynamically reconfigurable FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, April 1993. IEEE CS Press.

[Xil92]   Xilinx. *XACT Reference Guide*. Xilinx, San Jose, CA, October 1992.

[Xil93]   Xilinx. *The Programmable Logic Data Book*. Xilinx, San Jose, CA, 1993.

This article was processed using the LaTeX macro package with LLNCS style