A Neural Network Implementation Using Reconfigurable Architectures

Steven A. Guccione and Mario J. Gonzalez

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712, USA

Abstract

Several architectures based on Field Programmable Gate Arrays (FPGAs) have recently been introduced. These machines have demonstrated a high level of performance for a variety of problems. Despite this success, software development on these systems is generally limited to hardware description languages. One programming model that has been proposed for use with reconfigurable architectures is the vector based data parallel model. This paper describes the implementation of a multi-layer feed-forward neural network using a vector based data parallel approach. The algorithm is described using a subset of the C programming language. This description is translated into a circuit which may be programmed into the FPGA based processor.

INTRODUCTION

Recently, several FPGA-based machines have been designed and built. These machines have demonstrated supercomputer-level performance for a variety of computationally intensive problems. In spite of these impressive demonstrations, FPGA-based machines have not found widespread use. One limitation of these machines is their programming environment. For the most part, these machines have been programmed using hardware design tools. While this approach permits the most flexibility and highest performance, it requires that the programmer be a skilled hardware designer.

Guccione and Gonzalez (1993) have proposed a more traditional programming model for these machines based on the vector-based data-parallel model of computation. This model takes algorithms described in a high-level C-like language and translates them into high-performance digital circuits. In this paper, this technique is used to implement a multilayer feed-forward neural network.

THE NEURAL NETWORK MODEL

The network model is shown in Figure 1. This model contains three layers of neurons. These layers are referred to as the *input* layer, the *hidden* layer and the *output* layer. Data flows from the input neurons, through interconnections to the hidden neurons, then through interconnections to the output neurons. The path through the network is feed-forward and layered.

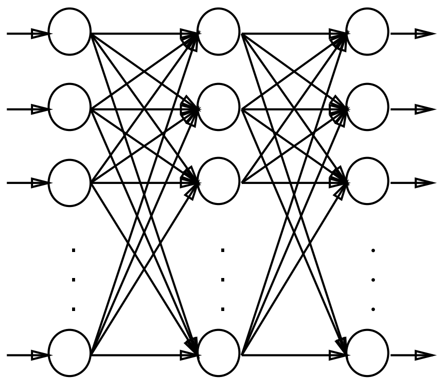

Figure 1 A three layer feed-forward network.

Each neuron broadcasts its output value to all neurons in the next layer. These output values pass to the next layer of neurons via *weighted* connections. These weighted connections amplify or attenuate the output values before they are input to the neurons in the next layer. The weighted values are summed, processed by some limiting function, then output to the next layer.

The values of the weights for the interconnections define the behavior of the network. Several automated techniques exist that allow networks to "learn" the values of these weights. For a set of inputs A, a set of associated weights W and a limiting function $f(x)$ we can describe the behavior of a neuron with the following equation:

$$output = f\left(\sum_i a_i w_i\right) \tag{1}$$

Despite this simple representation, calculating the output of this network is a computationally intensive problem. Each weighted interconnection in the network requires an addition and a multiplication operation. In a fully connected network with I inputs, H hidden units and O outputs, the number of interconnections is $(I \times H) \times (H \times O)$. To calculate the outputs of this network, $(I \times H^2 \times O)$ multiplications and additions must be performed as well as $(I + H + O)$ limiting functions $f(x)$.

Figure 2 shows a direct digital hardware implementation of a neuron. For a neuron with N inputs, N multipliers, $N - 1$ adders and the hardware to implement the limiting function, $f(x)$ are required. For networks containing a large number of neurons, this direct hardware implementation quickly becomes impractical.

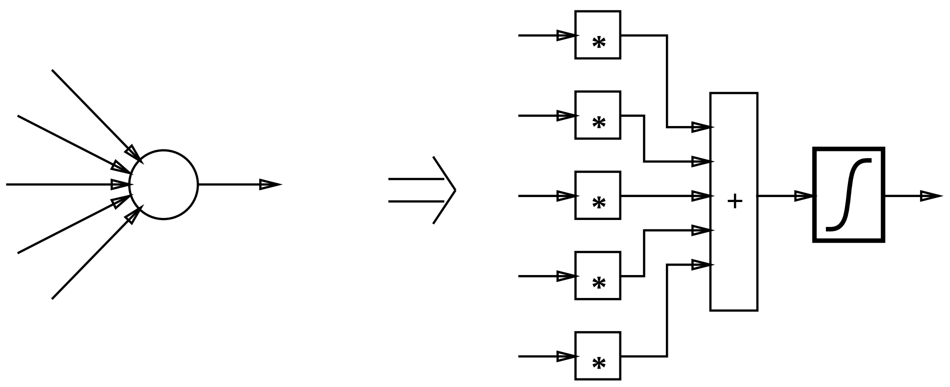

Figure 2 A digital representation of a neuron.

A VECTOR REPRESENTATION

The neural network model contains a large amount of parallelism. From the model, it is clear that all weighted inputs in a layer can be calculated concurrently. While this is possible, the number of hardware multipliers necessary to perform this task make this approach impractical for all but the smallest networks. This parallelism, however, may also be expressed in vector form. From this vector representation an efficient custom circuit can be extracted.

A vector representation of the network is shown in Figure 3. This figure shows a small network used to implement the Exclusive-OR function. The values of the inputs, outputs and weights are grouped into vectors.

The interconnection weights between the input and hidden layers are stored in the variable $w1$. This variable is not a matrix, but rather a list of vectors. The number of vectors in this list is equal to the number of hidden units. The length of the vectors in $w1$ is equal to the number of input neurons, plus one. The additional value in each vector is for the neuron *offset*. The offset can be viewed as a weight conected to an input whose value is always '1'.

Similarly, the weights between the hidden layer and the output layer are stored in the variable $w2$. The number of vectors in the list is equal to the number of output neurons. Since there is only a single output neuron in this network, there is only a single vector in the $w2$ variable. The length of this vector is equal to the number of neurons in the previous layer, plus one.

Three other vectors store the state of the network. The input vector, In, supplies values to the input neurons. This vector has two elements, one for each input neuron, and is padded with an additional vector element. This additional element is used in the calculation of the neuron offset. Since the offset may be considered a weight which is always connected to an input value of '1', this last element in the input vector is always set to '1', and represents the offset input. This extra vector element permits the offset to be treated in the same manner as the other weighted interconnections.

In a similar fashion, the hidden vector, h, holds the output of the neurons in the

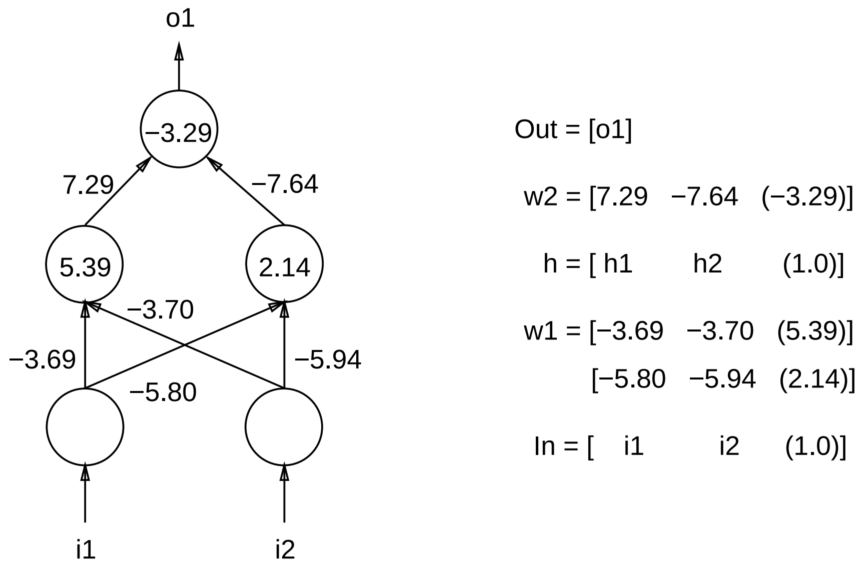

Figure 3 An Exclusive-OR network.

hidden layer. Like the input vector, this vector is also padded with an additional vector element containing a value of '1'. Finally, the output vector, *Out*, holds the value of the output neuron. This vector has a length equal to the number of output neurons.

Using this representation, a pairwise multiplication of the input vector *In* and each of the two weight vectors in *w1* performs all of the multiplications necessary for the calculation of the first layer. The products in each of these vectors are then summed. Finally, these sums are passed through the limiting function $f(x)$. This produces the outputs of the hidden layer neurons.

After the outputs of the hidden neurons have been calculated, a similar process is used to calculate the outputs of the output neurons. The vector of values representing the outputs of the hidden layer, *h*, is multiplied by the vectors in *w2*. These products are summed and passed through the function $f(x)$ to produce the final output vector.

The vector-based data-parallel code for this process is fairly simple. Figure 4 shows the code used to compute the output of the hidden layer. For clarity, the initialization of the weight and input vectors are not shown in this code.

The code in Figure 4 declares the input vector in and two vector lists $t1$ and $w2$. The vector list $w2$ contains the weights used in the calculation. The variable $t1$ is used to store the results. The code follows these declarations. In this case, the code may be written in just one line.

The vector multiplication operation computes the weighted inputs to the hidden units. Since *w1* is a list of vectors and *In* is a vector, each of the vectors in *w1* are multiplied by *In*. The *add_scan()* operation is used to sum the products. This function is a parallel prefix vector operation which sums the values in the vector. The *add_scan()* operation, along with other parallel prefix operations, is used by the *APL* programming language. Parallel prefix operations have more recently been used by the **LISP* data parallel programming

```
VectorList  t1(2,3);
VectorList  w2(2,3);
Vector      in(3);

/* Calculate outputs of hidden neurons */
t1 = f(add_scan(w1 * in));
```

Figure 4 Code to calculate the output of the hidden layer.

language (Hillis 1985).

Once the products have been summed, the output limiting function $f(x)$ is applied. The result is stored in the temporary list of vectors *t1*. This data structure, like *w1* contains two vectors of length three. The last elements in each of the two *t1* vectors contain the output values of the hidden neurons. These values are copied to the vector h and used to calculate the values of the output neurons. The other elements in the *t1* vectors are partial sums which are discarded.

The code for calculating the values of the output neurons is nearly identical to the code used to calculate the outputs of the hidden neurons. If the second layer weight vector list *w2* is substituted for the first layer weight vector list *w1* and the hidden layer output vector h is substituted for the input vector *in*, the code is the same. Given the regular structure of the network model, this similarity is not surprising.

THE SIGMOID FUNCTION

The function $f(x)$ is used to limit the values of the neuron outputs. In this network, we wish to limit the output to values between 0 and 1. McClelland and Rumelhart (1988) use what they term the *logistic function* to perform this limiting. This function is given by the equation:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

This equation was selected because it provides the necessary limiting of the outputs while having some properties which are useful in the learning phase of the algorithm. Unfortunately, this equation contains the transcendental function e^x, which is somewhat difficult to calculate. Nordström and Svensson (1992) list several functions which may be used as an approximation to the function used by McClelland and Rumelhart. These functions all have the same general characteristics. They are continuously increasing, approach 0 at $-\infty$ and 1 at $+\infty$, and have a continuous first derivative. The approximation we will use is given by the function:

$$f(x) = \frac{1}{2}\left(\frac{x}{1+|x|} + 1\right) \tag{3}$$

This function is a simple polynomial which uses no transcendentals. The graph in Figure 5 shows both the logistic function of McClelland and Rumelhart, given by *f1(x)* and the approximation above, given by *f2(x)*. Note that the curves provide a similar limiting function. It is the general characteristics of the sigmoid, not the precise equation which is important in this case.

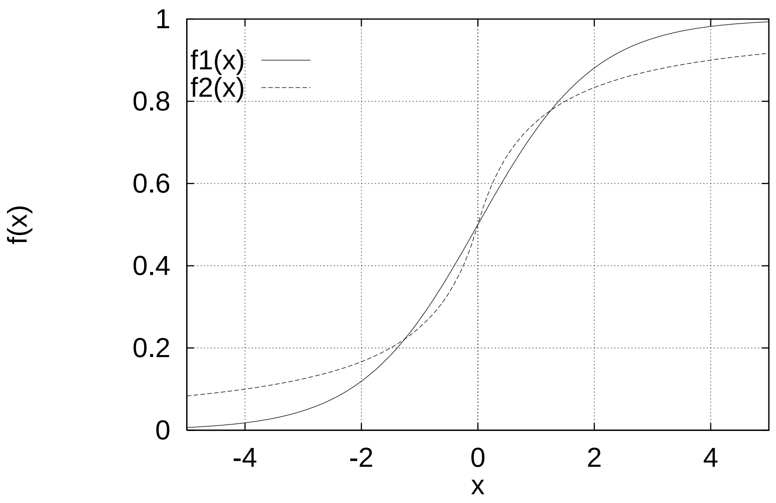

Figure 5 Sigmoid activation functions.

The C-like code for the function $f(x)$ is shown in Figure 6. It is a straightforward translation of the equation into software. Note, however, that the code is vector-oriented. It takes as its input parameter a predefined data type *Vector*. The result is also a *Vector*. This specification indicates that this function performs a vector operation.

```
/* Sigmoid activation function */
Vector
f(Vector  x) {
   return ((1.0 / 2.0) * (x / (1 + abs(x)) + 1));
   }  /* end f() */
```

Figure 6 Code for the sigmoid activation function.

CIRCUIT EXTRACTION

From these code fragments, circuits may be extracted which will implement the neural network algorithm. These circuits are extracted by creating the dataflow graph for the code. This graph is then used to configured the hardware.

Since the final result of this code is a digital circuit, no mechanism for a traditional software function call exists. To create the dataflow graph for the entire algorithm, dataflow graphs for the function calls must be generated, then expanded as macros.

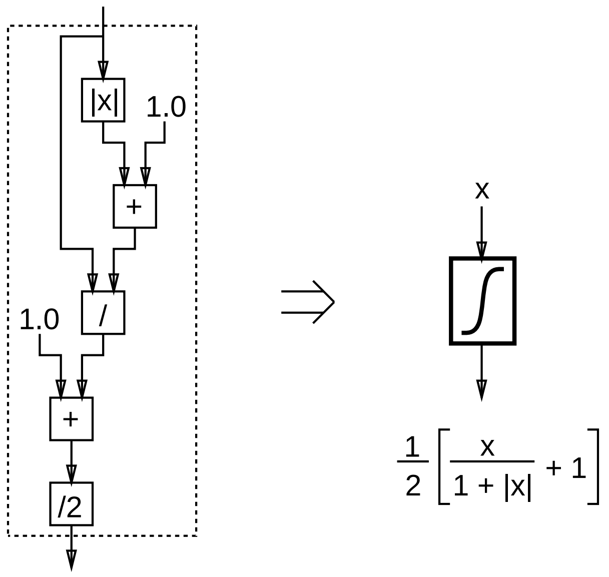

Figure 7 The sigmoid activation function circuit.

Figure 7 shows the dataflow circuit for the function $f(x)$. This circuit takes as its input a value x and returns the output $f(x)$. The functional units used by the circuit are: two adders, a divider, an absolute value and a divide-by-two circuit. Some simple optimizations have been performed on this circuit. A divide-by-two circuit, for instance, has been used rather than a full divider. Once the circuit for this function has been extracted, it may be used as a macrocell, much like the other macrocells in the circuit.

Once the circuit for the sigmoid activation $f(x)$ has been extracted, the code for the neuron output calculation may be converted into a circuit. In this case, the circuit is very simple. As Figure 8 illustrates, the weight and the input vectors are multiplied, with the results being accumulated by an *add_scan()* macrocell. The result is then passed to the sigmoid activation function for output limiting.

This circuit processes one set of vectors for each neuron in the network. Because the *add_scan()* macrocell performs an accumulate function, it will be necessary to clear the output of this macrocell to zero before each vector operation begins. This can be accomplished either with a reset signal, or by writing directly to the accumulate register in the *add_scan()* macrocell. It is expected that a reset signal will be more efficient.

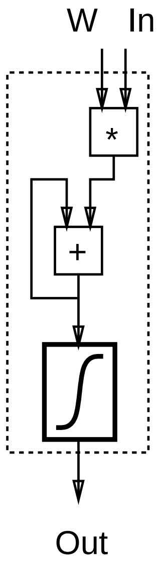

Figure 8 The neural network circuit.

PERFORMANCE

This implementation of this neural network algorithm is well suited to an FPGA-based machine. The uniform nature of the model permits a single circuit to be configured and used to calculate the outputs of the network. Additionally, the bandwidth requirements of the circuit is small. Two vectors are input and a single output vector is produced. Only two input ports and one output port are required.

One unusual feature of this circuit is the calculation performed by the sigmoid function. While it is only necessary to take the sigmoid of the final sum of the weighted inputs, this circuit takes the sigmoid of each of the partial sums. This would be extremely wasteful on an instruction set architecture, but in this case, there is no penalty for performing these extra calculations. In fact, to do otherwise would require that more than one circuit be used in the calculation. This would require an expensive reconfiguration phase in the algorithm. The ability to perform all of the calculations in a single pass with a single circuit is valuable, even though some computed values are never used.

The circuit produced by this code contains seven functional units. These units are cascaded in an essentially linear pipeline. If some of the more complex functional units such as the multiplier or divider are internally pipelined, the number of pipeline stages could be increased. This *superpipelining* of the circuit will increase the maximum clock speed of the circuit.

In estimating the performance of this circuit, two assumptions are made. First, it is assumed that the bandwidth of the memory system is sufficient. Two values must be supplied to the circuit input and one read from the output per clock cycle. Second, it is assumed that the time taken to fill the pipeline is negligible. For larger networks this is a valid assumption. With these assumptions, one weighted interconnection calculation is performed per clock cycle.

Since the metric typically used to measure performance of neural networks in *connections per second* or *CPS*, it is a simple matter to estimate the performance of this circuit. Since one connection is processed per cycle, the performance of the circuit in *CPS* is equal to the circuit clock speed. The clock speed of this circuit will depend on several factors, including the number of bits of accuracy used in the calculation, the type of FPGA devices used and the implementation of the functions in the macrocell library.

As a comparison, the CRAY-2 can simulate this network at approximately 50 MCPS (Nordström and Svensson 1992). This would correspond to a clock speed of 50 MHz for the FPGA-based design. Similarly, a 10-processor Warp system has been benchmarked at 17 MCPS (Pomerleau *et al* 1988). This would correspond to an FPGA circuit clocked at 17 MHz.

Nordström and Svensson (1992) give benchmarks for other architectures, some of which calculate over 1000 MCPS. These machines are not considered for comparison for two reasons. First, many are parallel machines which use several orders of magnitude more hardware than the FPGA approach. Second, many machines are special purpose processors. These machines achieve high performance, but are inflexible.

CONCLUSIONS

Using an FPGA-based reconfigurable architecture and a vector-based data parallel programming model, an efficient neural network can be implemented. This implementation is based on a simple high-level language specification which is translated into a high-performance pipelined dataflow circuit. The performance of this implementation compares favorably to implementations on other larger systems.

REFERENCES

Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs (1977 ACM Turing Award lecture)", *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, August 1978.

Blelloch, G. E., *Vector Models for Data-Parallel Computing*, The MIT Press, Cambridge, MA, 1990.

Cox, C. E. and Blanz, W. E., "GANGLION - a fast field-programmable gate array implementation of a connectionist classifier", *IEEE Journal of Solid State Circuits*, vol. 27, pp. 288–299, March 1992.

Guccione, S. A. and Gonzalez, M. J., "A data-parallel programming model for reconfigurable architectures", in *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 79–87, 1993.

Hillis, W. D., *The Connection Machine*, The MIT Press, Cambridge, MA, 1985.

Hush, D. R. and Horne, B. G., "Progress in supervised neural networks: What's new since Lippmann", *IEEE Signal Processing Magazine*, pp. 8–39, January 1993.

Lippmann, R. P., "Introduction to computing with neural nets", *IEEE Acoustics, Speech and Signal Processing*, pp. 4–22, April 1987.

McClelland, J. L. and Rumelhart, D. E., *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs and Exercises*, MIT Press, Cambridge, Massachusetts, 1988.

Nordström, T. and Svensson, B., "Using and designing massively parallel computers for artificial neural networks", *Journal of Parallel and Distributed Processing*, vol. 14, no. 3, pp. 260–285, March 1992.

Pomerleau, D. A., Gusciora, G. L., Touretzky, D. S., and Kung, H. T., "Neural network simulation at warp speed: How we got 17 million connections per second", in *IEEE International Conference on Neural Networks*, pp. 143–150, 1988.