FPGA-BASED RECONFIGURABLE ARCHITECTURES FOR NEURAL NETWORK

Wee Leng Goh

School of Electrical and Electronic E ngineering,
Nanyany Technological University, S1, Nanyang Avenue, Singapore 639798
E-mail: wlgoh@ntu.edu.sg

Abstract

This paper describes the use of FPGA-based reconfigurable architectures to implement artificial neural networks. The research is focused on investigating the properties of FPGAs (Field programmable gate arrays) to determine whether they are suitable hardware solutions, and to experiment with their reconfigurability as potential implementation approaches for neural network computing. Two implementation approaches were proposed. The first proposal, known as the template-based approach, is aimed at producing a computing architecture that combines high computational power with user-programmable flexibility to handle a wide variety of neural networks. The second approach introduced a partial-RTR (run-time reconfiguration) methodology that formats the native reconfigurable logic resources of an FPGA, allowing its reconfigurable region to be manipulated like a functional entity. The architecture is able to benefit from higher functional density without the complexities associated with conventional partial-RTR methodologies. A prototype system, FRANN, was constructed to test the proposed implementation approaches, and to act as an experimental testbed for future development.

Keywords: FPGA, reconfigurable architecture, artificial neural networks.

1. Introduction

One of the main attraction of artificial neural networks (ANN) information processing system is its ability to learn a task rather than being programmed to do it. Learning is achieved by iteratively adjusting the network's weights such that the neural network will response correctly to the input signals. The computation during the learning phase requires massive parallelism through a dense synaptic network, thus requiring computing systems capable of high computational power.

The widespread use of personal computers has allowed ANN to be implemented in software, where the inherent parallelism is simulated in a sequential manner. For large networks that require huge training data sets, the training time is normally very long. In order to reduce the training time, many implementations had used parallel computers or specialized architectures in an attempt to physically realize as much of the inherent parallelism that exist in neural networks.

Most of these neural computing systems catered to specific applications and implement only one neural network. For general-purpose neurocomputing, the ability to implement a variety of neural networks requires the system to be flexible as well as programmable. Unfortunately, in most computing architecture, customization and parallelism are often achieved at the expense of flexibility. So, an ideal system to carry out general-purpose neural training should be flexible and yet possess high computing power.

This led to the investigation of the use of FPGA devices, which can be used for building specialized digital circuits, and reconfigured repeatedly. In theory, a computing architecture consisting of FPGA devices may provide a promising solution to the customisation-flexibility issue. It is the goal of this research to find out whether the FPGA is really suitable, and if so, how it can be used.

The main contribution of this paper is the proposed implementation using FPGA's reconfigurability for creating neural processing hardware. The first method proposed, called the template-based approach, provides hardware programmability by deriving custom computing architecture from a generic hardware template. The template and its supporting system architecture takes into account the computation structure of neural networks. The advantage of the template approach is its ability to include circuit specialization for higher computational power, and flexibility together in the same system.

The second method proposed is an implementation approach for run-time reconfiguration. This method provides a simple solution to the difficulties encountered in existing partial run-time reconfiguration approaches. In general, it incorporates a reconfigurable framework, which can be manipulated like normal digital logic, into the processing architecture. The framework is generated by formatting the FPGA's reconfigurable logic resources.

A prototype was built to implement neural network training based on the proposed methodologies. The prototype consists of both hardware and software, and can be easily modified for future improvements. This paper provides a discussion on its design, results and observations obtained from building and testing the system.

2. FPGA-based Implementation of ANN: A Literature Review

The main goal of mapping neural network algorithms onto hardware is to achieve higher computational power. There are two ways of doing this: (1) by building high-speed processors, and (2) by employing parallel processing.

The three main arithmetic operations used in neural computations are multiplication, addition (or accumulation), and non-linear functions. For instance, the accumulation of weighted-inputs at the input of every neuron requires the multiplication of the input signal and the weight of the link, followed by the accumulation of these product values for all links going into the neuron. The resulting net input is then passed through a non-linear function, normally implemented in look-up tables, to generate the output. Multiplication is also used extensively in the generation of the $\boldsymbol{\Delta w}$ term in a majority of neural network algorithms.

From a different perspective, the operations can be treated as vector operations [1], as explained below:

1) Multiplication of a vector and a matrix (***Vector x Matrix***), as in the summation of weighted-inputs, and;
2) Multiplication of a matrix with another matrix (***Matrix x Matrix***), as in the generation of the $\boldsymbol{\Delta w}$ in the learning phase.

As can be seen, one of the most frequently used operation in ANN is multiplication. Every synaptic link requires a multiplication between the synapse input and the weight, and for two adjacent layers with just ten neurons each, a total of 100 multiplications are required. Unfortunately, high-speed digital multipliers often require a lot of logic resources to implement.

Various techniques to reduce the amount of hardware used had been proposed: using simpler but slower multiplier like those in bit-stream arithmetic, or modifying the neural algorithm such that it does not require any multiplication. Many bit-stream encoding and arithmetic methods have been used in the implementation of digital ANN hardware because they can simplify the multiplication operation. Examples are the stochastic methods [2], delta encoding [3] or the digital pulse stream techniques [4].

In most neural network hardware, data precision of 8 to 16 bits are sufficient, although intermediate results, such as net-inputs and products of multiplications, may require more bits to represent them. Without any demanding needs for high data precision, it is also sufficient to use fixed-point arithmetic operations rather than the hardware-intensive floating-point operations.

3. Implementation of the Prototype System

This section describes the implementation of a prototype neural computing machine using FPGA-based reconfigurable architecture. The prototype was used as a testbed for neural network hardware based on the proposed implementation approaches. Its design and development caters for a high degree of flexibility in order to accommodate changes during the course of the research project.

The FRANN prototype system consists of three main pieces of hardware:

1) a host computer;
2) an FPGA subsystem; and
3) UCIS bus-interfacing unit between the host computer and the FPGA system.

The host computer is an IBM-compatible PC that takes inputs from users and allows neural networks to be designed in software. The neural network and its parameters are stored in hard drives and retrieved whenever it is needed. The host's software also generates the FPGA's circuitry and configuration data for downloading onto the FPGA chips. All these functionality constitute the software platform of the FRANN system.

The FPGA subsystem is a collection of interconnected printed circuit boards, which house all the FPGA chips. Its main purpose is to provide a hardware platform to support all the FPGAs' functional requirements such as power supply lines, decoupling capacitors, headers for ribbon cables connections, configuration interfaces, and tapping the desired lines from the package pins. The

FPGA boards also contain static RAM memory to store data locally during system operation.

An ISA bus-interfacing unit known as UCIS (Universal Computer Interfacing System) bridges the FPGA subsystem and the host computer. It is based on the general-purpose ISA-interfacing hardware designed by Bruce Chubb [5]. The UCIS was modified to suit the needs of the FRANN prototype system.

Figure 1 shows the setup of the system.

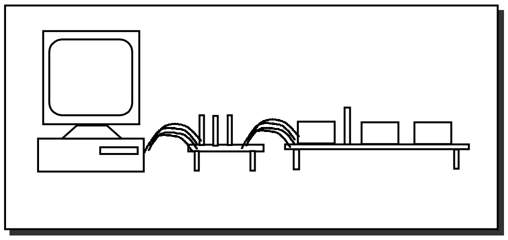

Figure 1: FRANN system setup

3.1 Computational Performance.

In neural network computing machines, the performance of the system is presented in two units. The first one is presented in terms of connections per second (CPS), or the number of weight multiplications in each second. This measurement is used during the operational or recall phase of the neural network, where the input signals are propagated forward. The second performance measurement is used during the training phase and indicates the number of weights updated per second, or connections update per second (CUPS). CPS and CUPS are often quoted in multiples of a million, as in million connections per second (MCPS), and million connection updates per second (MCUPS). These performance values take into consideration the total time required to complete either the operational or the training phase. This means that various delays due to communication, control and functional units are considered.

The performance results presented in this paper is obtained under several conditions which will be highlighted here. First of all, the prototype machine runs at a global clock frequency of 8 MHz. This clock frequency is chosen to accommodate all the delays incurred in the PCB boards, interfacing logic, interconnecting ribbon cables, and the SRAM read/write access time. The clock frequency is selected to ensure that all the data on the lines are stable before the next rising edge of the clock. From Xilinx timing analyzer tools, the FPGA internal circuitry of the implemented architectures can actually reach a maximum clock frequency of approximately 34 MHz. However, in order to cater for various external delays, it is scaled down to 8 MHz.

Next, the performances are measured at the optimum network condition. Optimum condition means that the percentage of time spent on non-processing activities such as data transfer and reconfiguration is at their minimum. In general, optimum conditions exist when the number of neurons in a neural network layer equals the total number of neural processors that can be physically realized on the FPGAs. Table 1 below shows the maximum number of neural processors that can be accommodated in each processing FPGA, and the total neural processors for the complete set of 4 processing FPGAs. The number of input elements is fixed at 32 for these measurements. Table 2 lists the performance results for all four algorithms and the corresponding network topology.

The performance does not account for:

1) System initialization before training begins.
2) Data transfer between host and local SRAM.
3) The initial programming of all the FPGA chips.

The processing architecture for the ADALINE is simple and straightforward with a group of 24 neural processors operating in SIMD scheme. No activation function unit is required as it uses the identity function. Processing power is increased by employing parallelism through the SIMD architecture, and through circuit customization. For the operational phase, the ADALINE network records a projected performance of 30.4 MCPS for 3072 weight multiplications, when all 4 processing FPGAs are used. During the training phase, the projected performance is 13.2 MCUPS to propagate an input pattern and then train the weights on it.

Notice that this particular ADALINE architecture accepts bipolar inputs (-1 and 1), which is represented as 0 and 1. Thus, the storage of the input data can be customized, i.e. 16 inputs are stored in one memory location. Each time a memory read is performed, 16 input data is loaded onto local registers. The time spent reading inputs from memory is reduced by 16 times.

In addition, with bipolar inputs, the multiplication unit is greatly simplified. Multiplying ***w*** with an input of 1 would result in ***w*** itself, and with 0 the two's complement of ***w*** is produced. Both are achieved using two's complement combinational logic that consumes very little resources and takes only one clock cycle to execute. Customizing the multiplier unit in this way allows us to add more neural processors into each FPGA while reducing its processing latency, both contributing towards increasing performance. Another less obvious advantage is that this multiplier does not double the wordlength of the product, thus no scaling circuitry and its related processing is required, which again leads to a lot of saving in terms of logic resources and computation latency.

Another feature of customization in the proposed architecture is the use of dedicated hardwired controllers for each of the operations performed. At most, only 3 clock cycles are required to set up and initiate each controller. The majority requires only two clock cycles, one to choose the operation, and another one to start the operation. As with all the other algorithms, the controllers are built using the one-hot method, which is fast (can reach 50 MHz or more) but consume more flip-flops.

Unlike ADALINE, the processing architecture for discrete Hopfield network is rather different. First of all, it does not use neuron-oriented mapping, but more closely resemble the synapse-oriented mapping. This is due to its algorithm which requires only one output unit's state to be updated at each forward pass of the feedback signal. In addition, all computation involved are concerned with the operational phase of the network and not the training phase.

Instead of having many simple neural processors executing concurrently, this particular Hopfield architecture uses just one neural processor which performs two weight multiplications, a summation of their two products, and an accumulator. The weights are represented by 8-bit two's complement integer and not the usual 16-bit two's complement fractional numbers. Only two weight multiplication units are employed because each 16-bit memory location can store two weight values. So, each memory read operation will retrieve two weight values per FPGA. The arithmetic units form a two stage pipeline producing a new sum-of-weighted-input value at each cycle after the first initial two clock cycle.

The processor's raw computing power is two weight multiplications for every clock cycle. If all four processing FPGAs are considered, then 8 weight multiplications occurred in each clock cycle. This is equivalent to a performance of 64 MCPS. However, as mentioned earlier the true performance of each architecture should take into account of other delays such as loading registers, processing by activation unit and so forth. In short, the total time taken to complete the forward phase. Table 3 shows the expected performance of the operational phase with respect to different number of neurons being implemented.

Notice that the performance increases with the number of neurons and slowly saturates when the number of neurons reach 16384 and above, at a value slightly less than 64 MCPS, which is the raw processing power of the processor. At 96 neurons (Table 1), the performance is compatible with that of ADALINE's and Kohonen's. The lower MCPS value is due to the fact that it processed three times more weights during the operational phase.

In general, the training architecture for Kohonen network is quite similar to the ADALINE architecture, but more complicated. Additional circuitry and processing is required to find the winner neuron and calculate how many of its neighbours can learn the current input pattern. Thus, its performance for the operational and training phase is slightly less than that of ADALINE's.

During the training phase, notice that the performance gradually decreases from 12.3 MCUPS to 0.2 MCUPS. This is a result of the gradual reduction of the neighbourhood size. Eventually it will reach zero and only the winner neuron gets to train its weights. The circuit now operates in the same condition as that with the Hopfield architecture except that it is using the ADALINE-like architecture instead of the Hopfield-like architecture. At this point of the processing, the Kohonen architecture is not optimized for the computation at hand and the performance drops tremendously. In fact, the computation becomes sequential in nature with only one active processor.

Overall performance would depend on how long the Kohonen network continues to train with a neighbourhood size of zero before the stopping condition is fulfilled. Assuming that the network spends an equal amount of training cycle with the maximum and minimum values of the neighbourhood size, the average performance of the network can be calculated by adding the maximum and the minimum values and dividing their sum by 2, i.e. approximately 6.25 MCUPS.

If the optimum performance is to be maintained, then one possible solution is to reconfigure the FPGA to change the parallelism from neuron's level to synapse's level when neighbourhood size has become too small, in order to resemble the Hopfield's processing architecture.

The performance of the backpropagation architecture is the lowest of the four algorithms that were implemented. This is a direct consequence of the backpropagation network being more computationally demanding compared to the other three. The differences are as follows:

1) Backpropagation network is multi-layered (the proposed implementation uses two layers). The main cause of delay in multi-layered network is the added data communication to reload the neural network parameters when time-multiplexing the layers.
2) In SRAM memory, the inputs, target and activations are represented by 8 bits two's complement numbers. However, in the neural processors, they are converted to 16-bit representation to maintain uniformity with the 16-bit weights. As a result, 16-bit multiplication operations are required which is very expensive for FPGA-based implementations. To reduce the consumption of logic resources, bit-serial multipliers are used, but at the expense of longer computational latency. The bit-serial multiplier is customized to produce a 32-bit product in 19 clock cycles instead of the usual 32 cycles. But still, when compared to the 1-clock cycle multiplication of the other architectures, it takes around 20 times longer to produce a result.
3) The backpropagation algorithm also uses more operations compared to the others. Thus, its neural processors are more complex and larger in terms of the amount of logic required. As a result, only 8 processors are implemented in each FPGA as compared to 24 in ADALINE and Kohonen's architecture.

Among the four architectures, only the backpropagation implementation uses an RTR region to realize some of its functional units. During the training of one input pattern (one forward and backward pass of the input signal), the RTR region is reconfigured 3 times to implement 3 different functional units. A total of 195 clock cycles are required to reconfigure 64 LUTs and 32 flip-flops. The time spent is only 3% of the total training time for one forward and one backward pass, or 1% for each configuration. The RTR region, including routing, uses approximately 180 CLBs or 11% of the total CLBs per FPGA. In other words, our RTR implementation allows us to reconfigure approximately 11% of the logic resources 3 times using just 3% of the total processing latency to train on one input pattern. The benefits of RTR can be enjoyed without compromising too much on processing latency.

Apart from performance, the other goal of the template-based approach is flexibility. For each of the algorithm above, their topology can have any number of neurons or layer of neurons, as long as all the neural parameters (weights, input, output, etc.) can fit onto the local SRAM memories. The current design of the FRANN system does not allow data transfer between the host computer and local SRAM during the operational or training phase in order to keep the design simple. The memory can accommodate a network with up to 262144 weights to be implemented on the FRANN hardware. This is equivalent to a single-layered network with 500 neurons and 500 inputs, or a two-layered BP network with 280 neurons in each layer. These values are projected figures based on the usage of local SRAM memories. The current application software of the FRANN system allows only 60 to 100 neurons per layer depending on the algorithm implemented.

As for implementing networks using algorithms other than the four which have been included in the architecture, existing templates can be reused, or new ones can be created. A library of templates and neural processors can be built to handle a wide variety of neural algorithms and topology.

3.2 Evaluation of RTR Method

The evaluation of RTR implementation, based on the formatting of reconfigurable logic resources, has shown that most of the design goals have been achieved. For instance, run-time reconfiguration is executed in 8 MHz, which is the system’s clock frequency. Thus, increasing the system’s clock speed would also increase the reconfiguration speed. Operating under the same clock also helps to synchronize the run-time reconfiguration to neighbouring circuitry’s operation. Next, the size (area) of the RTR region is variable allowing the reconfiguration time to be controlled. As is well known, the reconfiguration time is proportional to the amount of logic that needs to be changed. This will provide circuit designers with yet another means of controlling the total computation time. Another important feature of the RTR methodology is that it provides encapsulation of the internal formatted architecture from other circuits. This enables a modular approach to be used for both the static and dynamic circuitry.

The introduction of formatted RTR methodology has demonstrated many advantages. However, there are still several problems and limitations that needs to be considered. Firstly, the current implementation process is still carried out manually, which is limited to small RTR regions only. In order to have larger RTR regions, the process would have to be automated with software programs. Unfortunately, the algorithm required for such a software program turns out to be much more complicated than expected, and goes beyond the scope of this project.

Reconfigurations are only limited to functional elements. Routing elements remain fixed as there are currently no means of manipulating them directly. Data flow control devices such as multiplexers would have to be used to add some flexibility to the interconnection network. However, if too many of them are employed, it could lead to large consumption of logic resources and introduce delays into the signal paths.

Last but not least, if the dynamic circuit is complex, it might become more difficult to implement as compared to the conventional method. This is because the truth tables that are generated might be large and multi-level. In addition, it will become more difficult to design the routing elements so that they can accommodate all the configurations.

In general, circuits implemented using RTR logic consumes slightly more logic resources than the standard method of compiling a design onto FPGA bitstream. This observation is based on the fact that RTR circuitry are not optimized for minimum area during the design cycle. The effect is not obvious in the current implementation because the RTR region is small. However, it is predicted that large RTR regions could degrade the efficient use of resources if no optimization is included; especially when the interconnection network is complex. This is one of the area that will require further study and analysis.

In general, each architecture demands specific requirements. The ADALINE and Hopfield architectures are both very simple but can achieve computational power using different strategies. ADALINE uses a SIMD parallel architecture while Hopfield uses a pipeline processor inserted directly in the data path. Both used a certain degree of circuit customization to boost their performance.

Kohonen’s computation needs are more dynamic and changes from an ADALINE-like processing to a Hopfield-like processing. This dynamism was analyzed from the performance point of view. Finally, the backpropagation architecture was examined to determine how its complexity affects performance.

Observations on the use of formatted RTR logic were also discussed, and some of its advantages or disadvantages were highlighted.

4. Conclusions

In this paper, the research and analysis of neural network computation and how it could be implemented on hardware were presented. In addition, various digital hardware implementation techniques were examined in an attempt to understand the various computing needs of neural network computing systems. The survey has revealed that there was a need to achieve both high computational power and flexibility if a variety of neural networks were to be implemented on the same hardware resources. The FPGA-based system shows the most potential in achieving these two requirements.

Two implementation approaches for implementing the neural network computing hardware based on FPGA devices were proposed. The first one is known as the template-based approach targeted at achieving both performance and flexibility using the same hardware resources. The basic idea is to construct a complete neural processing architecture by customizing a basic template architecture. Each template can be used for several different neural algorithms. The template’s architecture implements only a layer of neurons and they are time-multiplexed for multi-layered networks. This is based on an earlier analysis which showed that inputs to hidden and output layers are obtained from the outputs of previous layers. Thus, the signal travels from one layer to another. It is not necessary to have all the layers implemented in hardware as only one is active at each instance of time.

The next proposed idea is catered to the implementation of partial-RTR capability onto standard FPGA devices. A method which formats the native reconfigurable logic resources of the FPGA in order to elevate them to the functional level of the system is used. The formatted RTR region can now operate just like a circuit component in the entire architecture eliminating many problems faced by other partial-RTR methodologies. The format consists of various building blocks such as, LUTs for combinational logic, flip-flops and routing resources.

Four neural network architectures - ADALINE, discrete Hopfield, Kohonen and backpropagation - were designed based on the two proposed ideas mentioned above. A prototype system was constructed to test the architectures and to act as a starting platform for future versions of the neural computing system. The prototype system consists of a host computer and a total of 7 Xilinx XC4044XL FPGAs housed on 7 separate printed circuit boards. The PCBs are linked via ribbon cables and communicates with the host computer through a UCIS bus interfacing module. The application software includes functionality for designing neural networks and converting them to FPGA configuration data. The four neural architectures were tested on the constructed prototype. The average performance of the single layer networks were found to be around 27 MCPS and 12 MCUPS, while the two-layered backpropagation computes at 8 MCPS and 2 MCUPS.

.

5. References

[1] Simon C.J. Garth, "*Simulators for Neural Networks*", in Advance Neural Computers, Elsevier Science Publishers B.V., 1990, page(s) 177-183.
[2] Stephen L. Bade, Brad L. Hutchings, "*FPGA-Based Stochastic Neural Networks – Implementation*", IEEE Workshop on FPGAs for Custom Computing Machines, April 1994, page(s)189-198.
[3] Valentina Salapura, "*Neural Networks Using Bit Stream Arithmetic: a Space Efficient Implementation*", 1994 IEEE International Symposium on Circuits and Systems, 1994. ISCAS '94., Volume: 6 , 1994 , page(s) 475 –478.
[4] P. Lysaght, J. Stockwood, J. Law, D. Girma, "*Artificial Neural Network Implementation on a Fine-Grained FPGA*", in Field-Programmable Logic: Architectures,
[5] Bruce Chubb, "*Build Your Own Universal Computer Interface*", Second Edition, McGraw Hill, 1997.

Table 1: Maximum hardware neural processors

Algorithm	No. of NP per FPGA	Total no. of NP for 4 FPGAs	CLB usage per FPGA
ADALINE	24	96	1148 CLBs
Kohonen	24	96	1246 CLBs
Backpropagation	8	32	1154 CLBs

Table 2: Projected performance of the prototype system

Algorithm	Topology	Total weights involved	MCPS	MCUPS
ADALINE	32-96	3072	30.4	13.2
Hopfield	96	9216	26.5	-- *NA* --
Kohonen	32-96	3072	28.6	12.3 - 0.2
Backpropagation	32-32-32	2048	8	2

Table 3: Hopfield operational performance for different network sizes

No. of neurons	MCPS
4	1.7
8	3.6
12	5
16	6.7
32	12.2
64	20.5
96	26.5
128	31
256	41.8
512	50.6
1024	56.5
2048	60
4096	62
8192	63
16384	63.5