FPGA IMPLEMENTATION OF A RESTRICTED BOLTZMANN MACHINE FOR HANDWRITING RECOGNITION

BY

TIAN XIA

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Assistant Professor Shobha Vasudevan

ABSTRACT

Despite the recent success of neural network in the research field, the number of resulting applications for non-academic settings is very limited. One setback for its popularity is that neural networks are typically implemented as software running on a general-purpose processor. The time complexity of the software implementation is usually $O(n^2)$. As a result, neural networks are inadequate to meet the scalability and performance requirements for commercial or industrial uses. Several research works have dealt with accelerating neural networks on Field-Programmable Gate Arrays (FPGAs), particularly for Restricted Boltzmann Machines (RBMs) — a very popular and hardware-friendly neural network model. However, when using their implementations for handwriting recognition, there are two major setbacks. First, the implementations assume that the sizes of the neural networks are symmetric, while the size of RBM model for handwriting recognition is in fact highly asymmetric. Second, these implementations cannot fit a model with a visible layer larger than 512 nodes on a single FPGA. Thus, they are highly inefficient when apply to handwriting recognition application.

In this thesis, a new framework was proposed for an RBM with asymmetric weights optimizing for handwriting recognition. The framework is tested on an Altera Stratix IV GX(EP4SGX230KF40C2) FPGA running at 100 MHz. The resources support a complete RBM model of 784 by 10 nodes. The experimental results show the computational speed of 4 billion connection-update-per-second and a speed-up of 134 fold with I/O time and a speed-up of 161 fold without I/O time compared with an optimized MATLAB implementation running on a 2.50 GHz Intel processor. Compared with previous works, our implementation is able to achieve a much higher speed-up while maintaining comparable resources used.

To my family and friends, for their love and support

ACKNOWLEDGMENTS

I am taking this opportunity to express my gratitude to everyone who supported me throughout the course of my master's study. I am thankful for their guidance and advice during the project. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to my study, my research, and my thesis.

Foremost, I would like to express my deepest gratitude to my advisor, Professor Shobha Vasudevan, for her continuous support and encouragement for my graduate study and research. When I started my graduate study, I knew almost nothing about research. Prof. Vasudevan was very encouraging when I was exploring different research project. I am sincerely thankful to her for giving me this opportunity to work on this project, a project that I am truly passionate about. Her inspiring talk during the group meetings motivated all of us not only to study the topics that are directly related to our research, but also to learn a different variety of subjects to widen our knowledge. Throughout my two years of master's study, I learned a great deal from her professionalism and dedication toward her research. Without her, this thesis would would not be possible.

I would like to particularly express my appreciation to Sai Ma for countless inspiring discussions about FPGA implementation. I sincerely thank her for listening to my problems and giving me wise suggestions and encouragement. I would also like to thank Jiayi for his patient explanation of neural networks concepts when I was first introduced to this topic. I would also like to express my heartfelt thankfulness to my wonderful colleagues, friends, and family, who supported me to overcome challenges that I faced. In the end, I would like to thank Alter University Program for donating the DE4 board that I needed for this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Artificial Neural Network

Artificial Neural Networks (ANNs) are computational modeling tools that are used to solve complex various real-world problems. Inspired by biological neural networks, ANNs are massively parallel computing systems that consist of numerous adaptive yet simple processing nodes that are densely interconnected [1]. Although ANN is an abstraction of the biological networks of the human brain, it is not a computational model which can duplicate the operations of biological neural networks. It is only a computational structure that models the known functionality of the biological neural networks for solving complicated problems. Similar to biological networks, ANNs have remarkable data processing and generalization characteristics such as massive parallelism, nonlinearity, robustness, fault tolerance, learning ability, generalization ability, and the ability to handle fuzzy information [2]. These characteristics are very desirable because of the following aspects [3]. First, nonlinearity allows the model to better fit data when it is complicated. Second, noise-insensitivity can provide an accurate prediction when data uncertainty, measurement errors, and outliers are presented in the training sample data. Third, high parallelism in the model can lead to fast data processing since multiple data sets can be processed simultaneously. Massive parallelism can also provide failure-tolerance in the system which can still provide an accurate prediction when part of the system failed. Last but not least, the learning ability and the adaptivity allow the system to update its internal structure in response to environmental changes, while generalizations enable applications to learn the underlying features of the given data

ANNs are abstractions of the biological neural networks and the biological neurons are the basic building blocks of the nervous system. The operation of

neurons will be briefly explained for understanding the operation of artificial neurons and the analogy between ANNs and biological neural networks.

A neuron, as shown in Fig. 1.1, is a special biological cell that acts as a basic information processing unit for the nervous system. It is composed of a cell body and two types of out-reaching tree-shaped branches: dendrites and axons. The dendrites of one neuron is connected to then axons of other neuron. The cell body, or soma, contains information about the heredity traits, plasma, and molecular equipment used for producing the material required for the neuron to function. The dendrites receive signals form the surrounding neurons and pass them to the soma. The soma collects all the signals it receives from the dendrites of its neighboring neurons, and transmits the signals through its axons to its surrounding neurons. This basic mechanism of signal transfer establishes the fundamental step of early neurocomputing and the operation of basic building units of the ANNs.

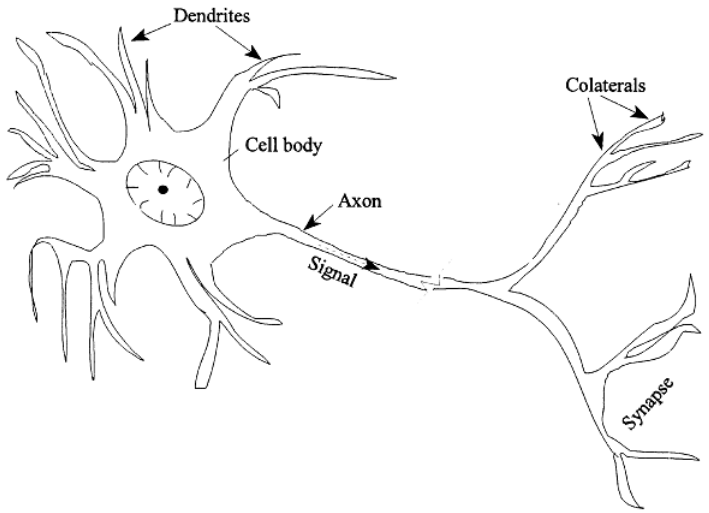

Figure 1.1: Schematic of biological neuron

The analogy between artificial neurons and biological neurons is the connection between the nodes that represents the connection between dendrites and axons. The connection weight represents the strength of the signal received from the dendrites and the threshold function approximates the activity of soma. Figure 1.2 illustrates n biological neurons with various signals strength x with connection strength w feeding into the soma with a threshold of b resulting in a signal y that is transmitted to other neurons through axons, and the equivalent artificial neuron system.

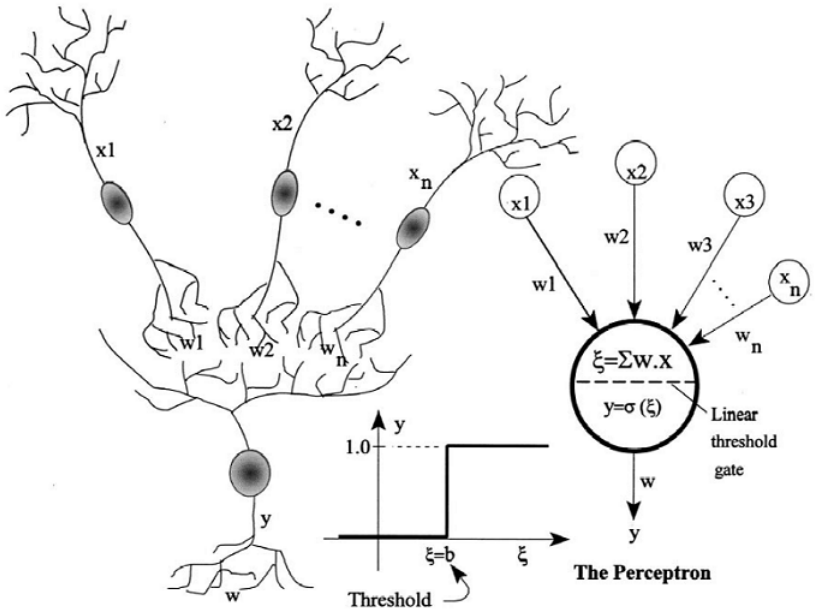

Figure 1.2: Signal interaction from neurons and is analogous to signal summing in ANN

An artificial neuron is a device, often called a node, with multiple inputs and one output. It receives its inputs from other nodes or an external source, and each input has an associated weight that can be adjusted through a learning process. Each neuron has two modes of operations: learning and using. During the learning mode, neurons can be trained to fire according to its firing rule which is often modeled by an activation function. During the using mode, the neurons decide to fire using the trained connection weights and its activation function.

The learning ability is a fundamental feature of intelligence, and the ability to learn automatically from given examples makes ANNs both attractive and impressive. Instead of following a set of rules specified by experts, ANNs appear to learn underlying rules from a given set of training examples. Although the precise definition for learning is difficult to define, the learning process of ANNs can be viewed as the process of updating the internal representation of the network in response to external stimuli so that it can be trained to efficiently perform a specific task. This is done iteratively by modifying the network architecture by adjusting the connection weights according to the input training data. The performance improves over time as the weights adjusts gradually.

1.1.1 Classification of ANNs

According to one or more relevant characteristics of ANNs, they can be classified in many different ways [3]. Generally, classification is based on the function that the ANN is designed to serve (e.g., pattern association, clustering), the degree of connectivity (partial/full) of the neurons in the networks, the direction of flow of the information within the networks (recurrent and nonrecurrent), and the type of learning algorithms. The ANNs can also be classified based on the learning rule of the networks (the driving engine of the learning algorithm) and the degree of learning supervision needed for ANN training. Supervised learning involved training of ANN with the target value for each input data, and using the error between the ANN solution and the corresponding target values to adjust the weights accordingly. Unsupervised learning does not require a correct answer for the training inputs. It learns by exploring the underlying structure in the data and correlates them between various data, organizing the examples into clusters based on their similarity or dissimilarity.

As examples of classification, Lippmann [4] classified ANNs according to their degree of learning supervision needed into two categories (supervised vs. unsupervised). Simpson [5] classified ANNs according to the flow of data in the networks (feedforward vs. feedback). Maren [6] proposed a hierarchical categorization based on structure followed by dynamics, then learning. Jain and Mao [2] introduced a four-level classification based on the degree of learning supervision, the learning rule, data flow in the ANN and the learning algorithm.

1.1.2 Application of ANNs

Generally, ANNs are more robust and often can provide better performance compared to other computational tools in solving a variety of challenging problems from the following seven categories.

1. Pattern classification is used to assign an unknown input pattern to one of several pre-specified classes. ANNs can solve such a classification problem with supervised learning by assigning proper class labels based on one or more properties that characterized a given class, as shown in Fig. 1.3(a). Classification applications that use ANNs range from

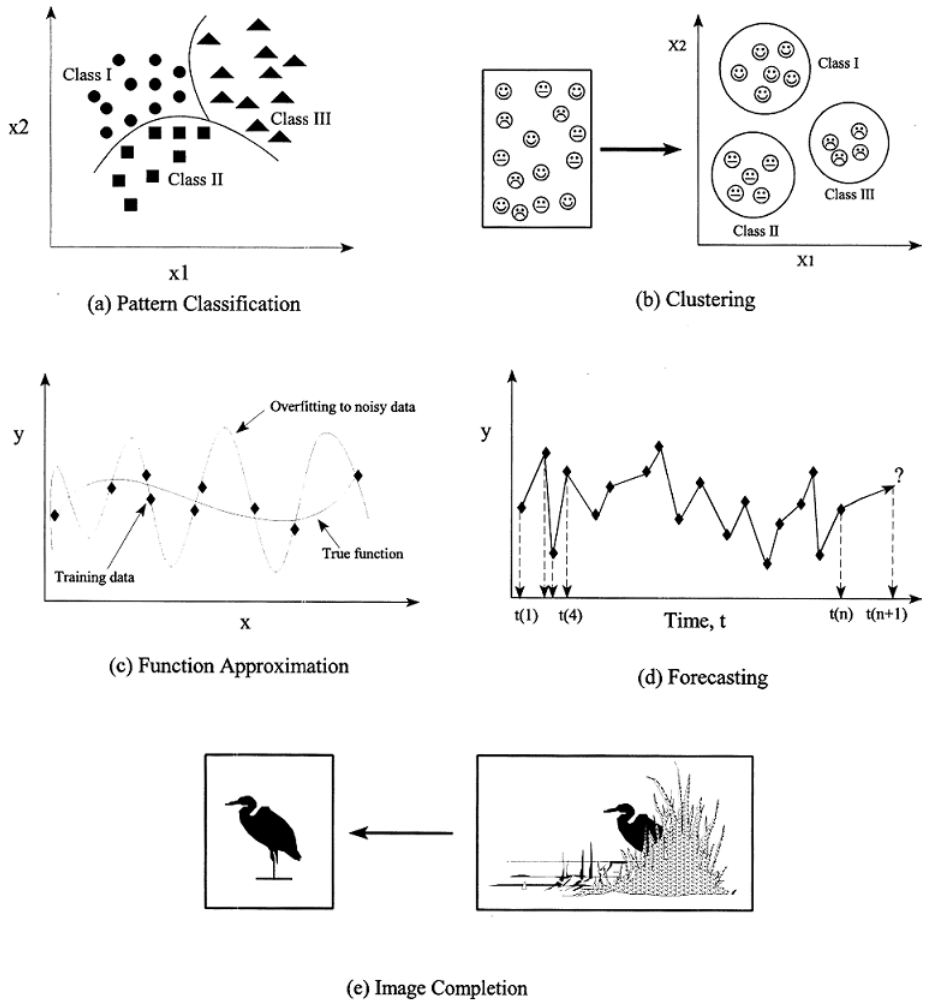

Figure 1.3: Application of ANNs: (a) pattern classification, (b) clustering, (c) function approximation, (d) forecasting, (e) association

microbiology characteristics [7], [8], [9] to areas of computer vision and signal processing such as handwriting and speech recognition [10], [11].

2. Unlike pattern classification, clustering is performed using unsupervised learning. ANNs can be trained with data with unknown class labels by exploring the similarity and dissimilarity between the neighboring data. The network can then assign similar patterns to the same cluster as shown in Fig. 1.3(b).

3. Function approximation includes training an ANN on input-output data so that the ANN can approximate the underlying rules or functions between inputs and outputs, as shown in Fig. 1.3(c). This is

extremely helpful in cases where there is no theoretical model for observed data available. It can also be useful when theoretical models are difficult to compute or analyze. Multilayer ANNs are considered to be the universal approximators that can approximate any arbitrary function to any degree of precision [1].

4. Prediction involves training an ANN on a set of samples representing a certain phenomenon at a given scenario at a certain time. The trained ANN is then used to predict the behavior for other scenarios at subsequent times. For example, as shown in Fig. 1.3(d) the ANN will be trained using data from $t(1)$ and $t(4)$. The ANN is then used to predict behavior of the model form $t(n)$ to $t(n+1)$.

5. Optimization is finding the best solution to maximize or minimize an objective function subject to a set of constraints. Optimization problems are a well-established field in mathematics. However, ANNs such as the Hopfield network [12] were found to be able to solve complicated and nonlinear optimization problems [13] more efficiently.

6. Association involves training a pattern associated ANN using noise-free training data. The well-trained ANN is then used to classify noisy or corrupted data. The associated neural networks should be able to reconstruct the corrupted or incomplete data. As shown in Fig. 1.3(e), the image of the bird was able to be reconstructed from the incomplete input image data. ANNs such as Hopfield and Hamming networks [4] are widely used for this application. A multilayer backpropagation ANN trained with identical input and output patterns can serve similar purposes [14].

1.2 Accelerating Neural Network

Although many existing ANN applications are usually developed as software, there are specific applications that demand high volume adaptive real-time processing, large data-set training done in reasonable time, and usage of energy-efficiency needed. To fulfill these requires, the ANN applications needed to be implemented in hardware since hardware can truly take advan-

tage of inherent parallelism in ANN architecture to achieve those requirements. Hardware devices that are specifically designed to model ANN architecture and associated learning algorithm can especially provide true parallel processing. These hardware devices are referred to as hardware neural networks or HNNs. Overall, HNNs can offer the following three advantages [15].

- Speed: Specialized hardware can offer a large amount of computational power, thus it can obtain several orders of speed-up, especially in the neural network system where parallelism and distributed computing are inherently involved. For instance, very large-scale integration implementation for cellular neural networks can obtain speed-ups to several teraflops [16]. This speed is very high for conventional DSPs, PCs, or even workstations.

- Cost: A hardware implementation of ANN provides the possibility for reducing system cost by lowering the total number of components and decreasing the power usage. This can be extremely crucial in high-volume processing applications, such as ubiquitous consumer-products for real-time image processing, which is price-sensitive.

- Graceful degradation: A fundamental limitation of any sequential uniprocessor based application is its extreme vulnerability to malfunction due to failure in the system. The primary reason for this limitation is lack of redundancy in the system architecture. As recent research indicates [17], even with modern multi-core processor architectures, the demand for effective fault-tolerant mechanisms still exits. Unlike the sequential processors, HNNs have parallel and distrusted architectures which allow the applications to continue to function while a small part of the system has failed.

1.2.1 GPUs

Recently, General Purpose Graphical Processing Units (GP-GPUs) have been identified as an intriguing technology to accelerate numerous data-parallel algorithms. ANN, on the other hand, embraces massive threads and data parallelism, which matches perfectly with GPUs. There are several attempts

to accelerate ANN training with GPUs [18], [19]. Liu and Guo [20] have proposed an approach which used CUDA programming model to train multilayer neural networks with back-propagation algorithm. Their implementation exploits the computing power of GPUs to accelerate the training process. The experimental results have shown that their approach can achieve up to 7 times the speed-up over its CPU counterpart. Similarly, Sierra-Canto et al. [21] proposed an implementation of the back-propagation algorithm on CUDA. They used a CDUA implementation of the Basic Linear Algebra Subprograms (CUBLAS) library to simplify the training process. Their implementation was able to achieve 63 times faster speed than its sequential version. On the other hand, Yan Zhang and Saizheng Zhang [22] introduced an optimized deep learning architecture with flexible layer structures and fast matrix operation kernels on parallel computing platform. Their fast matrix operation kernels are implemented deep in the architecture's propagation process which can save up to 70% of time on average compared with the kernels in CUBLAS library.

Recently, there was a study done Gu et al. [23] comparing the speed-up when using CPUs, GPUs and APUs. They implemented a multi-layer preceptron and an auto-encoder on various GPUs and APUs from mainstream processor manufactures. Evaluation results have shown that GPUs are faster than APUs at the cost of burning much more power, while APUs can give better performance per watt. Around the same time, there was another study [24] done to compare performance among multi-core CPUs, GP-GPUs and Field-Programmable Gate Arrays (FPGAs) for accelerating ANN. The results have shown that FPGAs can provide highest performance but needed multiple FPGA boards to fit the entire neural network. GP-GPUs on the other hand, are able to provide flexible solution with reasonably high performance.

1.2.2 FPGAs

In FPGA implementations of ANNs, the connection weights can be stored in registers, latches, or memories. Memory storage alternatives include dynamic RAM or static RAM [25]. Adders, subtracters, and multipliers are available on FPGAs for performing matrix multiplications. Nonlinear acti-

vation functions can be implemented using look-up tables or using adders, multipliers, etc. The FPAG implementations of ANNs entail advantages such as simplicity, low noise, flexibility and cheap fabrication [26].

Reconfigurable FPGAs provide an effective programmable hardware resource allowing different design choices to be evaluated very quickly. The cost for modifying the design is very low and it can provide the speed of hardware and the flexibility of software. In contrast to custom VLSI, FPGAs are readily available at a reasonable price and have a reduced hardware development cycle. Moreover, FPGA-based systems can be tailored to specific ANN configuration. However, the resource density on a single FPGA is still low which limits the size of neural networks that can be implemented on a single FPGA to be thousands of neurons.

The first successful FPGA implementation [27] of artificial neural networks was published a little over two decades ago. Since then, there have been many attempts to accelerate different ANN architectures using FPGAs [28], [29], [30], [31], [32], [33], [34], [35] for different applications ranging from speech recognition to simple classification. These proposed approaches try to optimized their design with different objectives such as speed, resource utilization, area etc. For instance, Krips et al. [36] presented an FPGA implementation of a neural network meant for designing a real-time hand detection and tracking system applied to video images. Their approaches tried to achieve reasonable processing time so that it could be useful for real-time application. Similarly, Rice et al. [37] report that a FPGA-based implementation of a neocortex inspired cognitive model can provide an average throughput gain of 75 times more than the software implementation on a full Cray XD1 supercomputer. They used the hierarchical Bayesian network model based on the neocortex developed by George and Hawkins [38].

From all different FPGA implementations, there are three typical aspects that designers will try to explore with different options to optimize their designs [39].

1. Data Representation: There are multiple research works that indicate training ANNs with integer weights is possible. If weights are represented as integers instead of floating points, the multipliers can be implemented more efficiently. There are a few attempts to implement ANNs with floating-points, but no successful implementation has been

reported up to the present. Nichols et al. [40] showed that despite continuing advances in FPGA technology, it is still impractical to implement ANNs on FPGA with weights represented with floating points. Another approach is to use a special learning logarithm [41] which uses powers-of-two integers as connection weights. The advantage of this is to simplify multipliers with series of bit shift operations.

2. Weight Precision: Selecting weight precision is one of the most important choices when implementing ANNs on FPGAs. Implementation with high weight precision will increase the implementation cost and decrease the computational speed. If weight precision is low, then it might compromise the functionality of the ANNs. The trade-off can be resolved if the minimum precision is determined. Holt and Baker [42] studied this problem by simulating using software with a set of benchmark classification problems. Their results indicate that a 16-bit fixed-point is the minimum precision without diminishing the learning ability of ANNs.

3. Transfer Function Implementation: The direct implementation of non-linear sigmoid transfer function can be very costly. There are two practical approaches to approximate the sigmoid function with FPGAs: piece-wise linear approximation or the look-up table. Piece-wise linear approximation is an approximate sigmoid function with a set of straight lines in the form of $y = ax + b$. If the coefficients for the lines are chosen to be power of 2, then the sigmoid function can be implemented using shift and add operations, which decrease the implementation cost further. The second method is to use a look-up table. The data used in the loop-up table are uniformly sampled from the sigmoid function. However, there is a trade-off with sample size and accuracy. A large sample size requires more memory which increases the implementation cost, while a small sample size leads to lower accuracy which might compromise the learning ability of the ANNs.

Besides all the detailed design choices, an important challenge that designers face when implementing ANN on FPGAs is to select an appropriate ANN model for a specific application so that the utilization of hardware resources can be optimal. Simon Jothson and others provide an inspiring insight on

this problem [43] by carrying out a comparative study on different ANN architectures. They implemented four different ANNs on FPGAs and analyzed the hardware requirements for each ANN structure on a benchmark classification problem. Even though their results are limited due to the number of ANN architectures they included in their study, their work provides a insight into HNNs with FPGAs.

1.2.3 Analog

Analog implementations of ANNs are usually more efficient in terms of chip area and processing speed, but this comes at the price of limited accuracy of the network component. The digital implementation, on the other hand, ensures the accuracy of the network component but with higher area cost and power consumption [44].

In analog implementations, the synaptic weights are typically stored using resistors [45], charge-coupled devices [46], capacitors [47], and floating-gate EEPROMs [48]. In VLSI, a variable resistor as a connection weight can be implemented as a circuit with two MOSFETs [49]. However, discrete values of channel length and width of the MOS transistors may cause a quantization effect in the weight value. The scalar product and subsequent nonlinear mapping are performed by a summing amplifier with saturation [50]. Unlike the digital implementation, the characteristics of the nonlinear activation function can be captured directly as a current that operates above saturation levels or as the voltage characteristics of transistors. In analog implementation, signals are usually represented by currents [51] and/or voltages [49]. Current flow is preserved at each junction point by Kirchhoff current law, and during multiplication various resistance values can be used for the matrix. Thus a network of resistors can simulate the necessary network conventions and their resistances are the adaptive weight needed for learning. Overall, analog neuron implementations benefit by exploiting simple physical effects to carry out some of the network functions [52]. For instance, the accumulator can be a common output line to sum currents. Analog elements are usually smaller and simpler than their digital counterparts. However, obtaining a consistently precise analog circuit, especially to compensate for variations in temperate and control voltages, requires sophisticated design

and fabrication.

There has been much work done to use analog circuits to model ANNs. Ortiz and Ocasio [53] presented a discrete analog hardware model for the morphological neural networks. They replaced the classical operations of multiplication and addition with addition and maximum or minimum operations. By doing so, they are able to simplify their hardware implementation. Milev and Hristov [54] presented an analog-signal synapse model using MOSFETs to analyze the effect of the synapse's inherent quadratic nonlinearity on learning convergence and on the optimization of vector direction. The synapse design is then used in a VLSI architecture for a finger-print feature extraction application. Similarity, Brown et al. [55] described an implementation of a signal processing circuits for a continuous-time recurrent neural network using sub-threshold analog VLSI in a mixed mode approach. In their implementation, each state variable is represented as a voltage while the neural signals are represented as currents. The use of currents allows the accuracy of the signals to be maintained over long distances, which made this implementation robust and scalable.

1.2.4 Mixed Signal

Mixed signal implementations of neural networks are designed to combine the digital and analog technologies in an attempt to get the best of both. For instance, analog implementation can be used for internal processing for speed while connection weights are stored digitally. The work done by the Mesa Research Institute at University of Twente [56] used 70 analog inputs, six hidden nodes, and one analog output with 5-bit digital weights to achieve the feed-forward processing rate of 20 GCPS. The final output of the neural network had no transfer function, so that multiple chips could be added to increase the number of hidden units. Similarly, a mixed signal architecture with on-chip learning has been presented in [57]. The overall architecture is divided into two parts, analog and digital. The analog ANN unit executes the neural function processing using a charged-based circuit structure, while the units for error correction, circuit control and clock generation are kept purely digital.

1.3 Motivation

When neural networks are implemented as software running on general-purpose processors, the algorithm complexity is generally $O(n^2)$. As a result, neural networks are unable to provide the performance and scalability required in non-academic settings. There have been many attempts to design hardware implementations to speed up the performance of neural networks [58], [59]. Although a variety of approaches, from analog to VLSI systems, have been pursued, they have not resulted in widely used hardware. These attempts are typically flawed with a lack of resolution, limited neural network size, and an absence of software interfaces.

Additionally to the difficulties with the hardware implementations, another common issue is the choice of the neural network architectures. This is due to the fact that most of the neural networks are not well suited for hardware systems. For instance, one of the most common types of neural networks is the multilayer perception with back-propagation architectures [60], [61]. Although this type of neural network is very popular and used for a variety of applications, the processing elements require massive real number arithmetic as well as great deal of resource intensive components such as multipliers and accumulators. Furthermore, the transfer function for this type of neural network is also very complicated. Consequently, the hardware implementation requires a significantly greater amount of resources, which limits the scalability of the hardware. The typically solution is to introduce a pipeline to obtain parallelism. However, this approach does not result in enough parallelism and speed-up to justify the cost and effort of using such systems [28].

1.3.1 Introduction of Restricted Boltzmann Machines

A Restricted Boltzmann Machines (RBM) is a generative stochastic artificial neural network model. It is able to learn the probability distribution over a given set of inputs. It was originally invented under the name Harmoium by Paul Smolensky [62] in 1986. It did not become popular until Hinton et al. [63] introduced the fact learning algorithm for RBMs. RBMs are widely used in applications such as dimensionality reduction [64], classification [65], and feature abstractions.

In comparison with other ANN models, RBMs have hardware-friendly architectures, well suited for hardware implementation. RBMs can use data types that map well to hardware since the node states are binary-values. As a result, binary arithmetic ensures that operations can be done with simple logic gates instead of resource intensive multipliers. In some cases, the node probability is used instead of the binary-valued node state. When this happens, the value for each node can only takes values from 0 to 1, and RBM does not require high precision, the node can be represented using fixed-point numbers, and the fixed-point arithmetic units can be used to decrease resource utilization and increase processing speed. The simplicity in RBM architecture allows more scalability and parallelism in hardware design.

Implementation of RBMs on FPGAs has several advantages over other hardware implementation methods for normal RBM architecture.

- One big drawback of the software implementation is that the complexity of the matrix multiplication needed for the learning algorithm is $O(n^2)$. If the learning algorithm is implemented with a FPGA, the fine-grain parallelism of the FPGA can be utilized for speed up matrix multiplication.

- RBMs have a hardware-friendly structure since the data can be represented using a fixed-point data type. Several previous research works have shown that only 18, or even a presentation with fewer bits is sufficient enough to represent the training data and the connection weights for the neural network to function correctly [66], [42], [67]. On the other hand, FPGAs have abundant embedded 18-bit by 18-bit multipliers available for speeding up the matrix multiplication process.

- FPGAs are rapidly growing. In addition to the raw fabric, FPGAs have various hardware components, such as on-board RAMs, DSP blocks, I/O transceivers and even processors. This allows the entire system to be implemented on the single board.

- The most important aspect of an FPGA is its ability to reconfigure. The topology of the network defines its application. The organization of processing units will define the capabilities and behavior of the neural network. Being able to implement on a reconfigurable system allows

hardware to be generated to suit the exact required topology, thus optimizing performance without sacrificing adaptability.

1.3.2 Previous Implementations of the RBMs

Recently, there have been work that introduced couple FPGA implementations for training RBMs, and the handwriting recognition was used as a benchmark to compare their results with the software implementations [67], [68], [69]. The first work that tried to implement an RBM on FPAG is done by Ly and Chow [69]. They implemented a high performance RBM for general use, but their implementation did not scale well. Thus, Kim et al. [67] proposed a highly scalable implementation for RBMs. However, there are two major drawbacks in their implementations. First, all of their implementations are based on the assumption that connection weights have a symmetric structure and the network has the same number of visible nodes and hidden nodes. However, if their implementations are used for training the entire RBM network for handwriting recognition, their implementations would simply not work or would be highly inefficient since the visible layer is much larger than the hidden layer for a RBM trained for handwriting recognition application. One possible solution is to zero pad the hidden nodes to be the same size as the visible nodes. However, once the hidden layer is zero padded, the overall size of the neural network is too large for the implementations to fit the entire system on a single FPGA. Although using multiple FPGAs to train one large RBM is possible [70], it is extremely inefficient when the problem can be solved using just one single FPGA. In this thesis, we proposed a new implementation called RAW, which stands for **R**estricted Boltzmann machine with **A**symmetric **W**eight. Compared with previous works, RAW is optimized for handwriting recognition and is able to perform the training process very efficiently.

1.4 Contribution

The new architecture, RAW, that we proposed in this thesis is able to train RBMs on FPGA efficiently. The implementation is also optimized for the handwriting recognition application. In RAW, we introduced a new method

to avoid the weight transpose problem. We stored each row of the weight matrix on a separate on-chip RAM, which allows the matrix multiplications to be processed in parallel. Furthermore, RAW used DSP blocks with four-multiplier adder mode to maximize the number of embedded multipliers available for matrix multiplications. As a result, we reordered multiplication and addition operations used for the matrix. We also introduced a shift register structure to the node selection module to reduce the hardware resources needed for this implementation. As shown in Fig. 1.4, we implemented RAW on Altera Stratix IV GX that ran at 100 MHz. The results were compared with a MATLAB implementation for RBMs. The experimental results indicate that RAW is able to achieve a speed-up of 134 fold with I/O time and a speed-up of 160 fold without I/O time. Compared to previous works, RAW is able to achieve a much higher speed-up while the hardware resources needed are very comparable with previous works. The main reason that it is much faster is that RAW is able to calculate the matrix multiplication with more parallelism due to the structure difference in the network and implementation. We also modified RAW implementation so it can be trained using different input sizes. The experimental results show that our implementation also scales well.

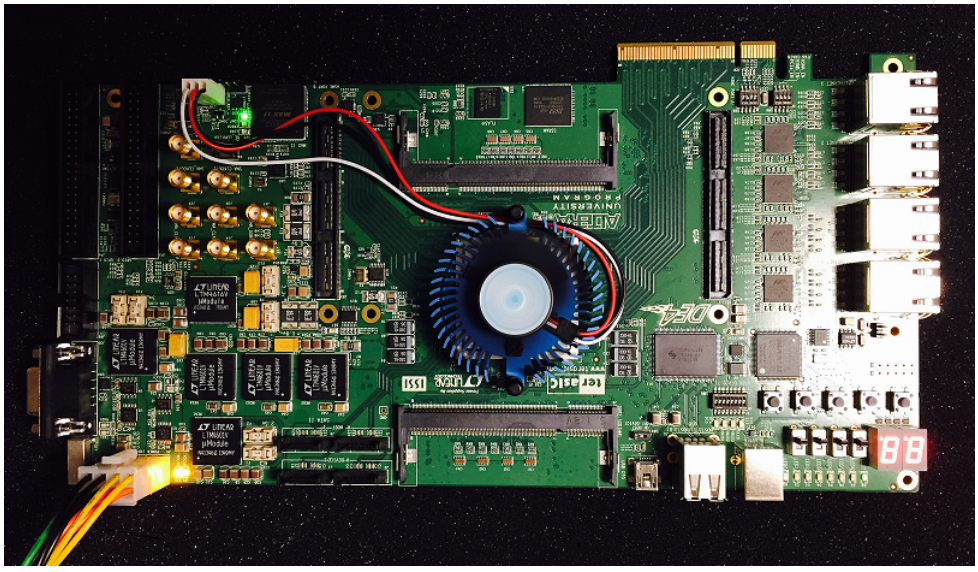

Figure 1.4: A picture of the Altera Stratix IV GX

The rest of the thesis is organized as follows.

- Chapter 2 provides preliminaries of the restricted Boltzmann machine

and two previous FPGA implementations with symmetric weights.

- Chapter 3 describes the FPGA implementation of the restricted Boltzmann machine with asymmetric weights.
- Chapter 4 describes the optimization made for the implementation.
- Chapter 5 presents experimental results with speed-up, area, and scalability comparisons.
- Chapter 6 concludes the thesis with a brief discussion of possible future work.

CHAPTER 2

PRELIMINARIES

2.1 Restricted Boltzmann Machine

This section briefly discuss the terminology, mathematical background and training procedures involved in the mechanics of Restricted Boltzmann Machine (RBMs). Additional details, including the historical development and statistical motivation can be found in [71], [72]. An RBM is a generative, stochastic neural network architecture. It is used to model the statistical behavior of a given set of training data.

The restricted Boltzmann machine is a generative, stochastic, and unsupervised learning neural network architecture. It uses statistical behaviors to model a particular set of data. Given a series of training input vectors, the network will be able to build an internal model based on the statistical distribution of the given data. Based on the training data set, the network is able to abstract the underlying properties of the input vector. The internal architecture can be used to detect whether an arbitrary data point belongs to the original input data.

The RBM is generative because the internal structure allows the network to produce new data which is consistent with the distribution. The RBM is also stochastic because it uses a probabilistic approach to model the input data. To capture statistical properties on the training data, the RBM determines the probability distribution of a given set with the help of random processes. These two properties, generative and stochastic, makes RMBs a unique artificial neural networks architecture.

Like all ANNs, the RBM is capable of learning. The internal structure of the RBM is mathematically defined by numerous independent parameters. Due to the state explosion of the parameter space, finding a correct set of parameters is a non-trivial task. To find the optimal parameters, the RBM

processes a set of training data, and applies the learning rules iteratively. The RBM repeatedly processes training data until it can generate desired output. Once the RBM is well trained, a new set of unexposed data, called the test data, can be used to verify its behavior.

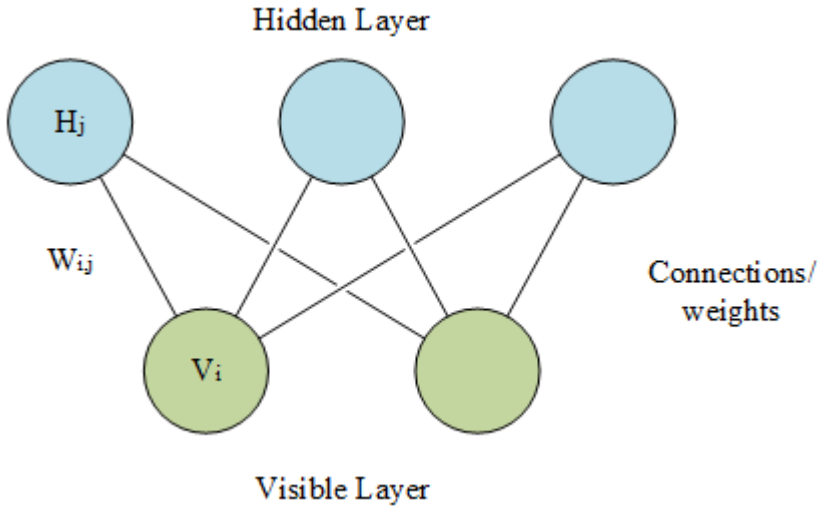

Figure 2.1: A schematic diagram of a restricted Boltzmann machine

In neural networks, the processing units are often called nodes. The nodes in the RBM have binary states: on or off. As shown in the Fig. 2.1, the RBM consists of two layers of nodes, a hidden layer and a visible layer. The visible layer is used for input access while the hidden layer acts as an internal representation of the data for the networks. There are connections between every node in the two different layers, but no connections exist between nodes in the same layer. Each of these connections has an associated weight, which is the parameter that the RBM tries to optimize at each training iteration. As shown in Fig. 2.1, v_i and h_j are the binary states of the ith and jth nodes in the visible and hidden layers respectively. $w_{i,j}$ is the weight for the connection between v_i and h_j.

2.1.1 Alternating Gibbs Sampling

Alternating Gibbs sampling (AGS) is the training operating process for the RBM. It is the fundamental rule for generating node states and learning optimal connection weights [73]. AGS is divided into two phases: *construction* and *reconstruction* phases. During the construction phase, the visible layer

is used to determine the node state and the probability of the hidden layer. During the reconstruction phase, the hidden layer is used to generate the node state and the probability of the visible layer. The change in the weights is calculated in the last AGS phase. To begin the process, an initial data vector is loaded into the visible layer and phases are operated in an alternating manner starting with the construction phase. To differentiate the nodes between different phases, the node state will be label with the phase number as its superscript. Figure 2.2 is a representation of the AGS process.

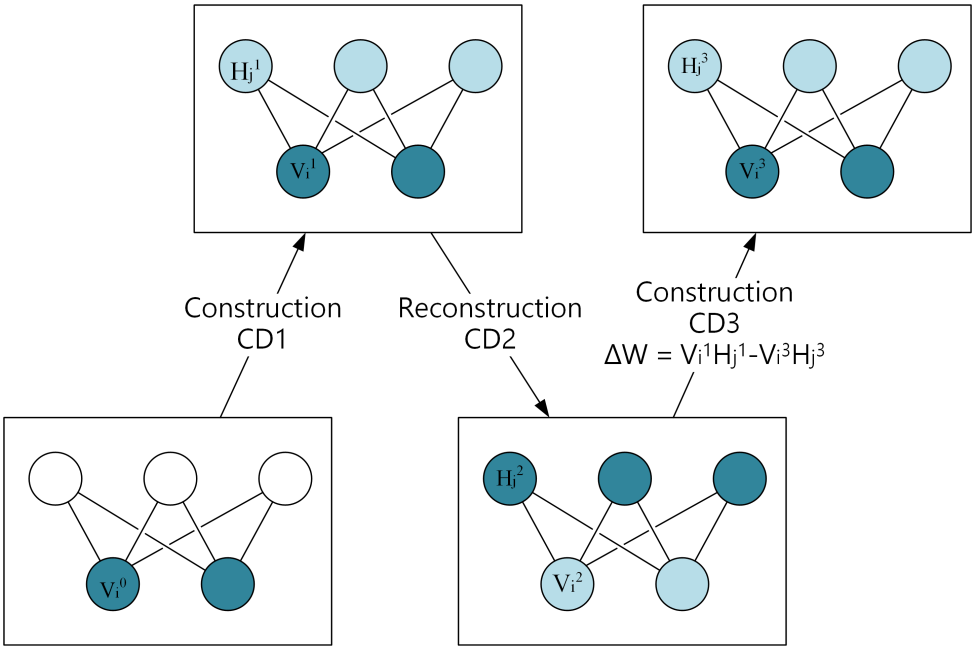

Figure 2.2: A schematic diagram of the alternating Gibbs sampling for three phases

In order to understand how the node states are determined, the concept of *global energy* must be introduced first. The global energy can be simply treated as a numeric value that determines the operation and the behavior of an RBM. The global energy is defined in Eq. (2.1).

$$E = -\sum_{i,j} w_{i,j} v_i, h_j \tag{2.1}$$

Since the weight connections only exits between nodes in different layers, the energy function can be redefined as a sum of two *partial engergies*. Depending on which AGS phase is being computed, the partial energy will be calculated using different equations. The construction phase uses Eq. (2.2),

and the reconstruction phase uses Eq. (2.3).

$$E = -\sum_i v_i(\sum_j w_{i,j}h_j) = -\sum_i v_i E_i \tag{2.2}$$

$$E = -\sum_j h_j(\sum_i w_{i,j}v_i) = -\sum_j h_j E_j \tag{2.3}$$

The equations for calculating the partial energies have show that the global energies can be calculated by using just the node states and the partial energy. Since the partial energy is independent of related node state, they can be calculated concurrently to speed up the calculation of the global energy. Using the statistical approach of defining probability with respect to energy functions, the node states have a cumulative distribution function of a sigmoid function. The probability for a visible or a hidden node to be turned on is expressed in Eq. (2.4) and Eq. (2.5).

$$p(v_i = 1) = \frac{1}{1 + e^{-E_i}} \tag{2.4}$$

$$p(h_j = 1) = \frac{1}{1 + e^{-E_j}} \tag{2.5}$$

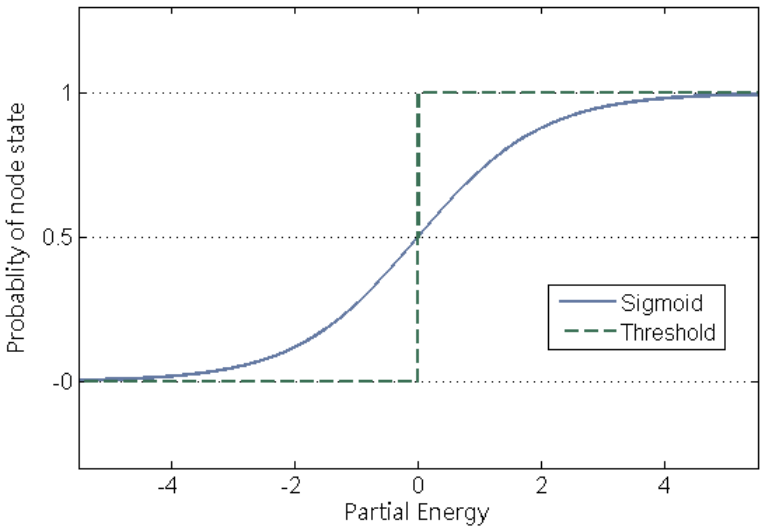

Figure 2.3: A plot of a sigmoid function and a threshold function

To determine the node state from the sigmoid function, a uniformly ran-

dom variable must be sampled. Sometime, when the probabilistic approach is undesired, a deterministic, first-order approximation threshold function expressed in Eq. (2.6) and Eq. (2.7) is used. A comparison plot between the sigmoid function and the threshold function is shown in Fig. 2.3.

$$v_i = \begin{cases} 0, & E_i < 0 \\ 1, & E_i \geq 0 \end{cases} \tag{2.6}$$

$$h_j = \begin{cases} 0, & E_j < 0 \\ 1, & E_j \geq 0 \end{cases} \tag{2.7}$$

2.1.2 Learning

One of the primary reasons for neural networks to be attractive is their ability to learn, and as result, the learning rules of RBMs are generating great interest [74], [75]. In the learning rules of RBMs, the connection weights are parameters used to determine the energies and node state for next AGS phase. To model a given data set, the connection weights have to be adjusted at each iteration so that the energy generated from the RBM for the entire set of training data is minimum. To find the minimum energy, the differential equation of E with respected to the individual connection weight is expressed in Eq. (2.8).

$$\frac{\partial E}{\partial w_{i,j}} = \epsilon(\langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^\infty) \tag{2.8}$$

In this equation, the $\langle \cdots \rangle^X$ represents the expected values of Xth AGS phase. ϵ is the *learning rate* of the network that is defined by the user. The node states are calculated by an iterative process of AGS. As a result, the derivative of the energy function indicates the direction vector of steepest descent in the weight space to reach the minima. Therefore, the weights must be adjusted according to the derivative at the end of every training set.

This formulation raises three important points. First, the expected values of the node interactions are required over the entire set of training data to calculate the gradient descend properly. This is called *batch learning*. However, if the batches are large, the calculations will require a significant amount of time. One way to resolve this is to divide the batches into smaller groups. The weights will be updated with each smaller batch. This is called

mini-batch learning. If the mini-batch is still undesired, the batches can be divided into each individual input vector, and this is called *on-line learning*.

Second, according to Eq. (2.8), the formal definition of the gradient descent requires the node state values from the infinite AGS and that is impractical to implement. Thus researchers have found that the infinite AGS phase can be replace with a small finite number. For RBMs, the lowest possible AGS phase to train the model correctly is 3.

Last, the learning rate is an independent parameter which defines the step size for each weight update. A larger learning rate leads to a faster learning process, while a smaller learning rate ensures convergence. Therefore the designers need to carefully choose the learning rate due to this trade off. Some studies suggest that the learning rate can be modified between batches to achieve a convergent solution quickly. This is called *simulated annealing* [75], [74].

Although these learning algorithm shortcuts deviate from the formal definition of gradient descent, they enhance operational speed and are widely adapted. The learning algorithm for weight update now can be defined as shown in Eq. (2.9) and Eq. (2.10), where k is the number of batches, and L is the number of data vectors in one batch.

$$w_{i,j}[k+1] = w_{i,j}[k] - \epsilon(\langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^X) \tag{2.9}$$

$$\langle v_i h_j \rangle^X = \frac{1}{L} \sum_{l=0}^{l} v_i^X h_j^X \tag{2.10}$$

In order to make the learning algorithm easier to understand and compute, Eq. (2.1) to Eq. (2.10) can be reformulated using matrix expression. For an RBM with i visible nodes and j hidden nodes, the visible and hidden layers can be expressed in Eq. (2.11) and Eq. (2.12) respectively.

$$v_l^X = [v_0^X \cdots v_{i-1}^X] \mathbb{B}^{1\times i} \tag{2.11}$$

$$h_l^X = [h_0^X \cdots h_{j-1}^X] \mathbb{B}^{1\times j} \tag{2.12}$$

The connection weights can be reformulated as Eq. (2.13).

$$W[k] = \begin{bmatrix} w_{0,0}[k] & \cdots & w_{0,j}[k] \\ \vdots & \ddots & \vdots \\ w_{i,0} & \cdots & w_{i,j}[k] \end{bmatrix} \in \mathbb{R}^{i \times j} \tag{2.13}$$

Then, the Eq. (2.1) to Eq. (2.10) can be reformulate as:

$$V^{X+1} = \begin{cases} V^0, & X = 0 \\ f(E_v^X), & X \; is \; odd \\ V^X, & X \; is \; even \end{cases} \tag{2.14}$$

$$H^{X+1} = \begin{cases} f(E_h^X), & X \; is \; even \\ H^X, & X \; is \; odd \end{cases} \tag{2.15}$$

$$E_v^X = (H^X)W^T, \in \mathbb{R}^{l \times i} \tag{2.16}$$

$$E_h^X = (V^X)W, \in \mathbb{R}^{l \times j} \tag{2.17}$$

$$W[k+1] = W[k] + \frac{\epsilon}{l}((V^1)^T H^1 + (V^X)^T H^X) \tag{2.18}$$

Here $f(\cdot)$ is the sigmoid or the threshold transfer function applied element-wise to the matrix.

2.1.3 Complexity Analysis

To understand the reason that a software implementation of the RBM running on a sequential processor is not desired, the algorithm for it needs to be analyzed. The pseudo code for the software implementation of the RBM is presented in Algorithm 1. In order to make the analysis easier, we are going to assume that the RBM will have symmetric layers, where the hidden layer and visible layer have the same size ($i = j = n$). The algorithm for training the RBM is divided into three code block: node select, energy computation, and weight update. A detailed complexity analysis is shown in Table 2.1. As indicated in Table 2.1, the complexity of overall the algorithm is $O(n^2)$.

Algorithm 1 pseudo-code of RBM training algorithm

```
1: for every batch in the training data do
2:     visible []= get_datavector(batch);
3:     for every AGS_phase do
4:         if AGS_phase is odd then
           /* Engery computer Eq. (2.17) - 2 loops ⟶ O(n^2) */
5:             for every hidden_node do
6:                 for every visible_node do
7:                     energy[j]+=visible[i]weight[i][j]
8:                 end for
9:             end for
               /* Node Select Eq. (2.15) - 1 loop ⟶ O(n) */
10:            for every hidden_node do
11:                hidden[j] = transfer_function(energy[j])
12:            end for
13:        else
           /* Energy Compute Eq. (2.16) - 2 loop ⟶ O(n^2) */
14:            for every visible_node do
15:                for very hidden_node do
16:                    energy[i] += hidden[i]*weight[i][j]
17:                end for
18:            end for
               /* Node Select Eq. (2.14) - 1 loop ⟶ O(n) */
19:            for every visible_node do
20:                visible[i] = transfter_function(energy[i])
21:            end for
22:        end if
           /* Weight update Eq. (2.18) - 2 loops ⟶ O(n^2) */
23:        if AGS_phase == 1 then
24:            for every visible_node do
25:                for every hidden_node do
26:                    weight_update[i][j] += visible[i]*hidden[j]
27:                end for
28:            end for
29:        else if AGS_phase == AGS_limit then
30:            for every visible_node do
31:                for every hidden_node do
32:                    weight_update[i][j]-=visible[i]*hidden[j]
33:                end for
34:            end for
35:        end if
36:    end for
37: end for
    /* Weight update Using Eq. (2.18) - 2 loops ⟶ O(n^2) */
38: for every visible_node do
39:    for every hidden_node do
40:        weight[i][j]+=learning_rate/batch*weight_update[i][j]
41:    end for
42: end for
```

Table 2.1: The complexity analysis for each code block of the RBM algorithm

Procedure	Complexity	Equation
Node select	$O(n)$	(2.14), (2.15)
Energy computer	$O(n^2)$	(2.16), (2.17)
Weight update	$O(n^2)$	(2.18)

2.1.4 Layered Networks

A single RBM only has one layer of visible nodes and one layer of hidden nodes. As a result, the RBM can only model first-order statistics. This limits the modeling ability of the RBM to learn when given training data with underlying properties that require higher orders of statistics for complete description. Although a single RBM has limited modeling ability, we can stack multiple RBMs together to increase its modeling ability as long as the number of nodes matches up [63]. As shown in Fig. 2.4, the hidden layer

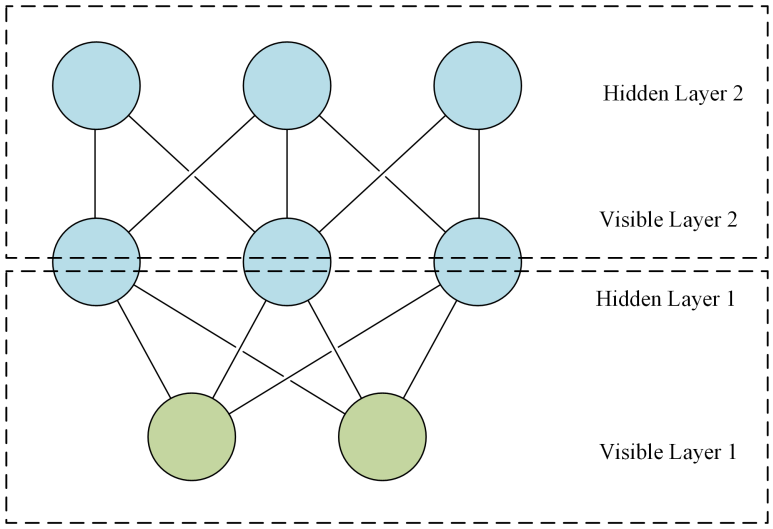

Figure 2.4: A schematic diagram of a double-layered RBM

of one RBM will acts like the visible layer of another RBMs. The stacking is indefinite as long as there are enough resources to support the stacked RBMs. The operations and learning algorithm are changed slightly for the stacked RBMs. The individual RBMs still operate the same way, but there is a macro-algorithm to organize how the layers operate with respected to each other. More detailed description can be found in [63].

2.2 FPGA Implementation of RBM with Symmetric Weight

Although there have been many attempts to design hardware implementation of various neural network architectures, there is a growing interest in hardware-accelerated restricted the Boltzmann machine due to the popularity of deep belief nets applications. When implementing RBMs on FPGAs, one of the major issues is the weight storage. Depending on different AGS phases, Wor W^T will be needed to calculate the partial energy. In order to speed up the matrix multiplication operation, a row or a column needs to be accessed at the same time so that the multiplication can proceed in parallel. Thus the distribution of the weights in a non-trivial problem is due to the transpose operation that occurs during the reconstruction phase. There are two hardware RBM implementations that have interesting ways to solve this issue, and they are the main interest to this thesis.

2.2.1 BRAM-Based Distribution for Memory Core

The first implementation is done by Ly and Chow [68]. They implemented their design on Xilinx Virtex II-Pro XC2VP70 FPGA running at 100 MHz. The resources support an RBM up to 128×128 nodes. Their solution to solve the weight distribution problem is that they distributed the connection weights onto different BRAMs in a way that no embedded RAM will simultaneously read out two or more elements from the same row with the same address, and no embedded RAM will contain two or more elements of the same column. Then by using a carefully designed address scheme, a column or row of the matrix is read out from the memory each cycles and no communication is required for the transpose. This distributed BRAM-based matrix data structure is illustrated in Fig. 2.5 with $n = 4$. Their architecture uses binary-valued visible node states, which reduce the resource utilization greatly without using any multipliers for multiplication. Since the node states are binary values, the matrix multiplication operations are implemented with a series of AND gates and a binary tree adder to calculate the partial energies. To further simplify their implementation, they use a threshold function for node selection.

Their results were compared with an optimized C program implementation

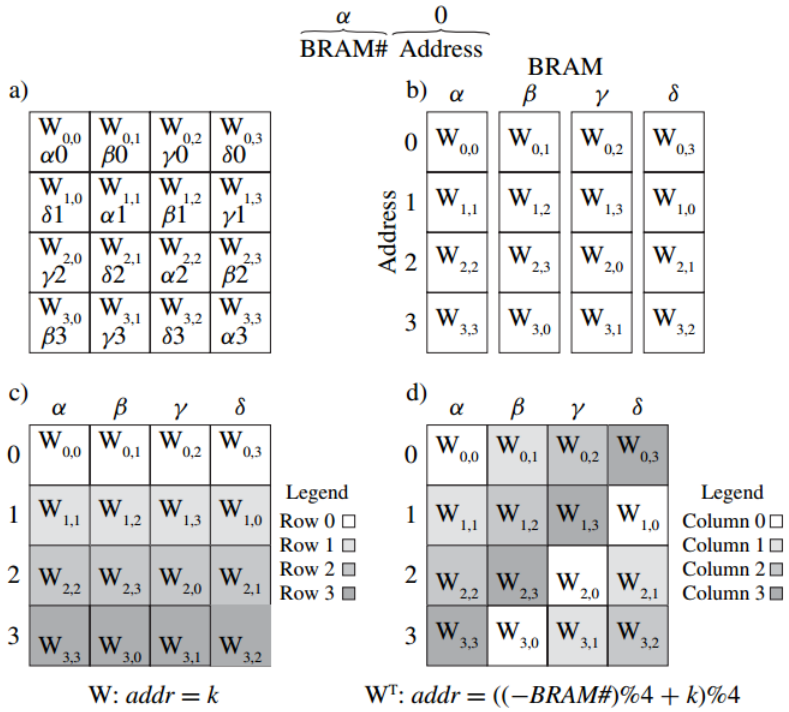

Figure 2.5: (a) Typical weight matrix organization with BRAM addresses added, (b) weight assigned to BRAMs, (c) weights access by row, (d) weights access by column

of an RBM running on a 2.8 GHz Intel processor. Their implementation was able to achieve computational speed of 1.02 billion connection-updates per second and a speed-up of 35 fold compare to a software implementation.

2.2.2 Sub-Row Memory Core

The second implementation is done by Kim et al. [67], where they designed a highly scalable RBM on FPGA. They implemented their RMB architecture on an Altera Stratix III EP3SL340 FPGA with a DDR2 SDRAM interface. The Altera Nios II soft-processor is also used and running at 100 MHz while the RBM module ran at 200 MHz. Unlike the first implementation, their work was capable of supporting both real-valued and binary-valued visible node states. At high level, their RBM module has an array of weights and neurons that are fed into an array of multipliers and adders to perform matrix multiplication. At the lower level, their RBM module is segmented into several groups, with each group consisting of an array of multipliers, adders,

embedded RAM, and logic components. The weights and neurons are also distributed across the network. This implementation makes the design highly scalable since a group can be easily added to the RBM module to increase the size of the RBM model.

Their architecture used 16-bit fixed-point numbers to represent the weight and node probability, which is capable of training RBM without affecting its learning ability. They treated matrix multiplication in several different ways. A matrix multiplication shown in Eq. (2.19) can be considered as multiple linear combinations of vectors, multiple vector inner products, and as a sum of vector outer products as shown in Eq. (2.20), Eq. (2.21), and Eq. (2.22). They achieved performance acceleration by implementing groups of multipliers, adders and embedded RAM. Each group stores a row of the weights stored on separated local memory. The computation hardware is then selected on the type of energy that is being generated using the DSP units. Multiply-and-accumulate logic generates the visible energies while an adder tree is used for the hidden energies. This allows both types of energies to be generated using the identical memory access to avoid the weight transpose problem.

$$C = A \cdot B \quad where\ A\ \epsilon\ \mathbb{R}^{m\times k}, B\ \epsilon\ \mathbb{R}^{k\times n} \tag{2.19}$$

$$\begin{bmatrix} C_{1,j} \\ C_{2,j} \\ \cdots \\ C_{m,j} \end{bmatrix} = \sum_{i=1}^{k} \left(B_{i,j} \begin{bmatrix} A_{1,i} \\ A_{2,i} \\ \cdots \\ A_{m,i} \end{bmatrix} \right) \tag{2.20}$$

$$C_{i,j} = \begin{bmatrix} A_{i,1} & A_{i,2} & \cdots & A_{i,k} \end{bmatrix} \cdot \begin{bmatrix} B_{1,j} \\ B_{2,j} \\ \cdots \\ B_{k,j} \end{bmatrix} \tag{2.21}$$

$$C = \sum_{i=1}^{k} \left(\begin{bmatrix} A_{1,i} \\ A_{2,i} \\ \cdots \\ A_{m,i} \end{bmatrix} \times \begin{bmatrix} B_{i,1} & B_{i,2} & \cdots & B_{i,n} \end{bmatrix} \right) \tag{2.22}$$

For node selection, a piecewise linear approximation of a nonlinear function was used to create a sigmoid function, which only requires a minimal number of addition and shift operations [76]. The random number generator uses a

combination of a 43-bit Linear Feedback Shift Register (LFSR) and 37-bit Cellular Automata Shift Register (CASR), which provides good statistical properties and a cycle length of 2^{80}, which is sufficient for RBM application.

They compared their implementation to the MATLAB code provided by Hinton et al. [1] using a 2.4 GHz Intel Core2 processor implemented in a single thread. Implementing network sizes of 256 $\times$ 256, 512 $\times$ 512, and 256 $\times$ 1024. They achieved speed-up of 25 fold compared to a single-precision MATLAB implementation and 30-fold for a double-precision MATLAB implementation.

CHAPTER 3

OUR IMPLEMENTATION OF RBM WITH ASYMMETRIC WEIGHT

The objective of this thesis is to implement an RBM architecture on FPGA optimized for handwriting recognition. Due to the nature of this application, the neural networks used to train the input data will have very asymmetric connection weights. Each input data vector represents an handwriting image, where each node presents a pixel in the image. The hidden nodes on the other hand represent the 10-bit label of each image, which each hidden node presents 1 bit. As a result, the number of nodes in the visible layer is going to be much larger than the number of nodes in the hidden layer. Initially, the implementation is optimized for the MNIST benchmark where each image is 28 by 28 pixels and the labels for each image is represented using a 10-bit vector. Later in the implementation stage, the architecture is change so the visible layer size can scale to different sizes to train different input image sizes.

Our implementation is capable of supporting both real-valued and binary-valued visible node states. We used 18-bit fixed-point to represent node probabilities and connection weights. There are two reasons for choosing 18-bit fixed-point numbers to be our data type. First, studies have been shown that RBMs can be trained with a minimum of 16-bits. Second, the FPGA that we used for this implementation has abundant 18 $\times$18 bit embedded multipliers.

To make the implementation simpler, we decide to use ϵ that is a negative power of 2 so that the multiplication operations for calculating the delta weight can be implemented by shift operations instead of resource intense multipliers.

Our overall design breaks down into seven big modules: control unit, node selection core, matrix multiplication core, memory core, visible nodes module, hidden nodes module, and I/O interface. This chapter will discuss the design of each module in great detail.

3.1 RBM Core

RBM is the top-level entity of the entire design. It consists of seven modules.

1. Control Unit: This module made up by two state machines, one for the RBM computation, and one for fetching input data from the SRAM. The state machine for RBM computation keeps track of the AGS phases. Depending on the AGS phases, the control unit will generate different control signals to other modules.

2. Node Selection Core: This module computes the probability of a node state to be turned on for a visible and hidden node using a sigmoid function and the partial energy of the hidden layer or visible layer.

3. Matrix Multiplication Core: This core is responsible for any matrix multiplication operation needed for the RBM algorithm. It consists of an array of multipliers and numerous full bit adders.

4. Memory Core: This core has all the connection weights stored on several individual on-chip memories. Each row of the weight matrix is stored on a separated RAM block. The implementation is designed in a way that no more than two connection weights in the same column will be access at the same time.

5. Visible Node: This module contains the node probability values for the visible layer at the first and Xth AGS phases. It also contains an array of shift registers that are used as temporary storage for the next input training vector.

6. Hidden Node: This module contains the node probability values for the hidden layer at the first and Xth AGS phases.

7. I/O Interface: This module is responsible for fetching the next input training data from the SRAM while the RBM core is performing computation on the current input data. It gives the control unit a signal when data is ready.

The overall architecture and data flow of the entire design are shown in Fig. 3.1. The control units and their outputs are highlighted in red. As shown in Fig. 3.1, the control signals control all other modules. The outputs

the of weight block, visible node, and hidden node are all inserted into the *Matrix Mult* module with two multiplexers. The outputs of *Matrix Mult* either feed into the weight block to update the connection weights or feed into the node select module and update the values in visible or hidden layer.

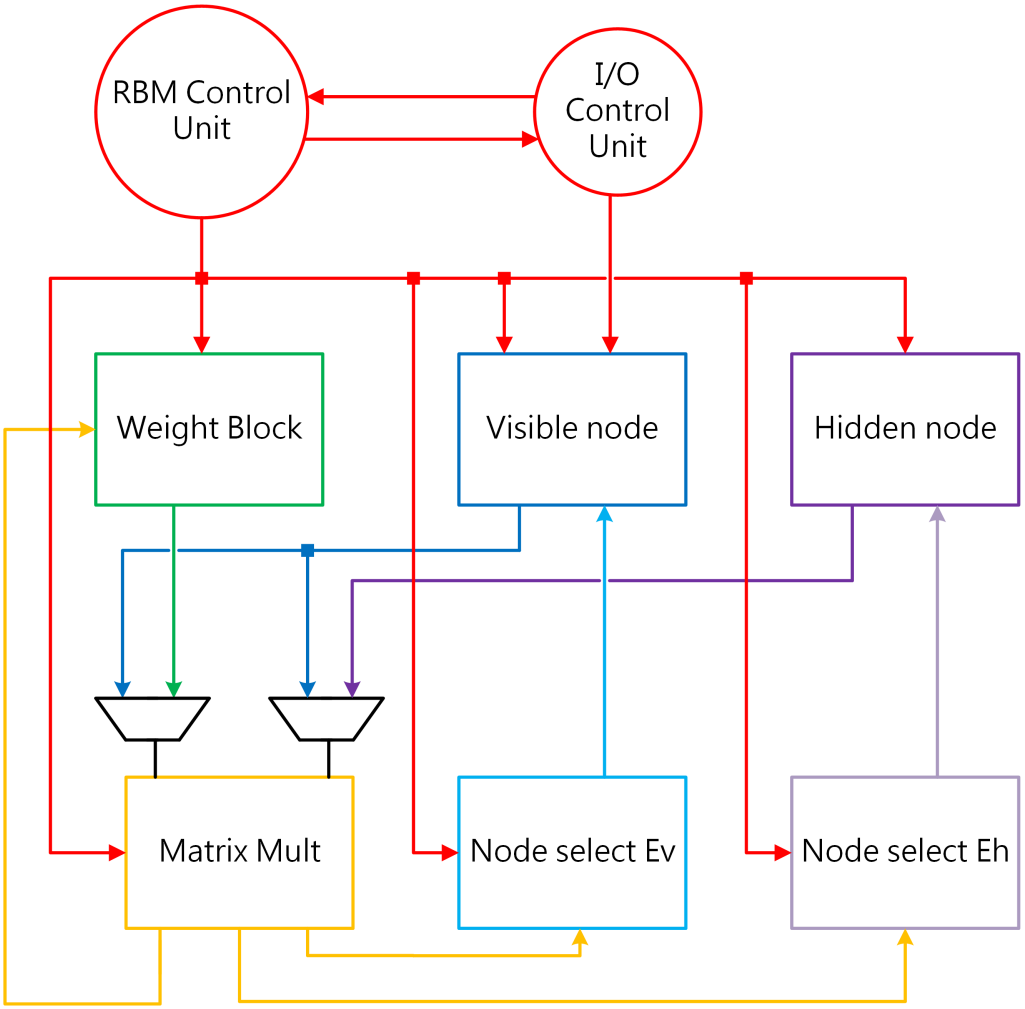

Figure 3.1: A top-level diagram of the RBM core

3.2 Control Units

The *controlunits* module consists of two control units, one for the RBM computation running at 100 MHz and another one for the I/O interface running at 200 MHz. The control unit for RBM computation has seven states as shown in Fig. 3.2.

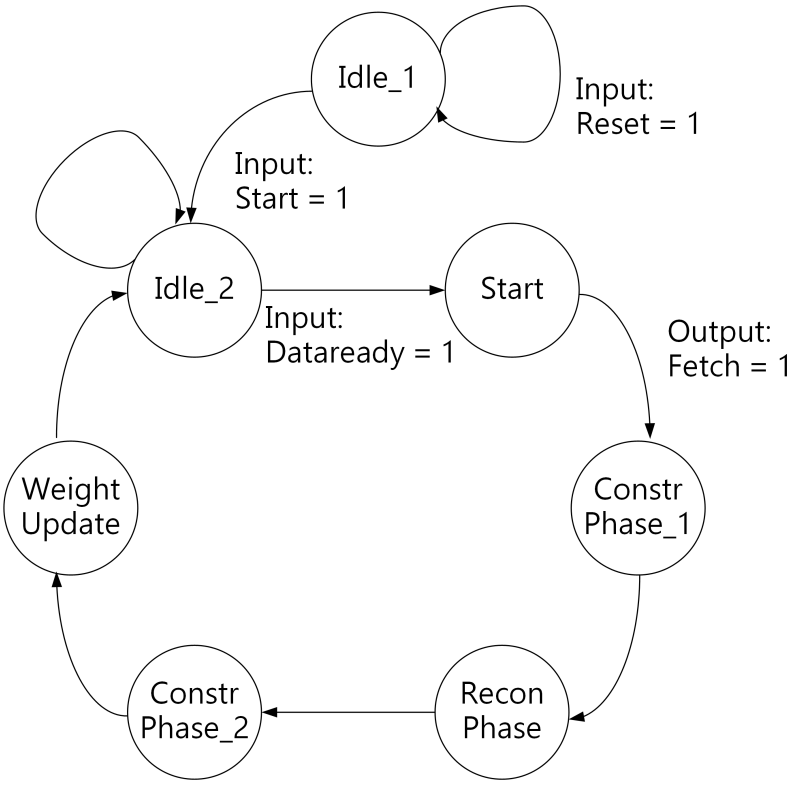

Figure 3.2: The state machine of RBM computation

1. When the state machine starts or whenever the *reset* signal is one, the state machine will go to $Idel\_1$ state. The state machine will stay on this state until the *start* signal is one, and then to move on the $Idle\_2$ state.

2. In $Idle\_2$ state, the RBM computation core is waiting for the I/O interface to finish fetching one input data vector. Once the *dataready* from another control unit is one, the state machine will move on to the *start* state.

3. During the *start* state, the data that stored in the shift registers will be loaded into registers that hold the values for visible nodes. At the same time, the control unit sent the I/O interface control unit a $fetch$ signal, so that the I/O interface will start fetching the next input data vector from the SRAM to the shift registers while the computational core is processing the current input data. Without any additional input signals, the state machine will move on to the next state.

4. During first construction phase, the computational core is calculating

the node values for the first AGS phase using Eq. (2.14), Eq. (2.15), and Eq. (2.17). As shown in Fig. 3.3, the connection weights and visible node values are fed into the matrix multiplication core. At each clock cycle, a partial energy E_h is calculated and its value is fed into the *Node selection Eh* module. Once the *Node selection Eh* module calculates the sigmoid function of the given partial energy, the hidden node will be updated with the new values.

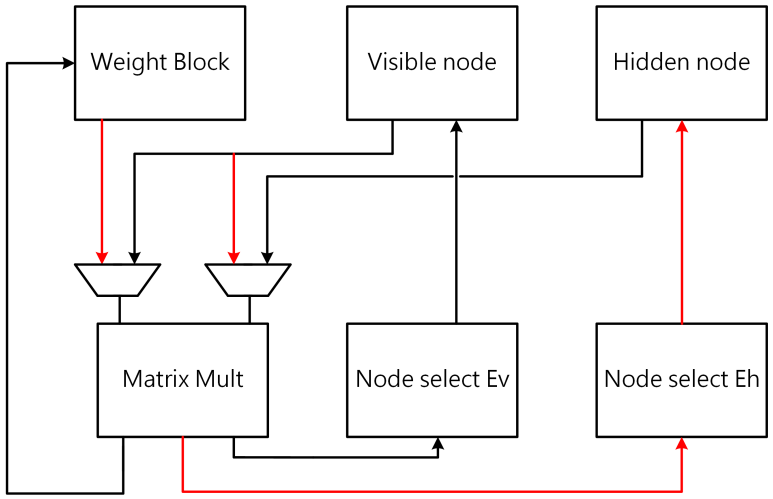

Figure 3.3: A data flow diagram for the construction phase

5. Once the construction phase is finished, the state machine will move to the reconstruction phase. During this state, the computational core is calculating the node value for the second AGS phase using Eq. (2.14), Eq. (2.15), and Eq. (2.16). As shown in Fig. 3.4, the connection weights and the hidden node are fed into the matrix multiplication core. Once all the partial energies E_v are calculated, their values will be fed into the node selection Ev module to compute the sigmoid function. After that, the visible layer will update its node values.

6. Once the reconstruction phase is done, the state machine will move to the second construction phase which calculates the third AGS phase. In this phase, the data flow is exactly the same as the first construction phase.

7. After the second construction phase is finished, the state machine will move to the weight update phase. As shown in Fig. 3.5, the visible

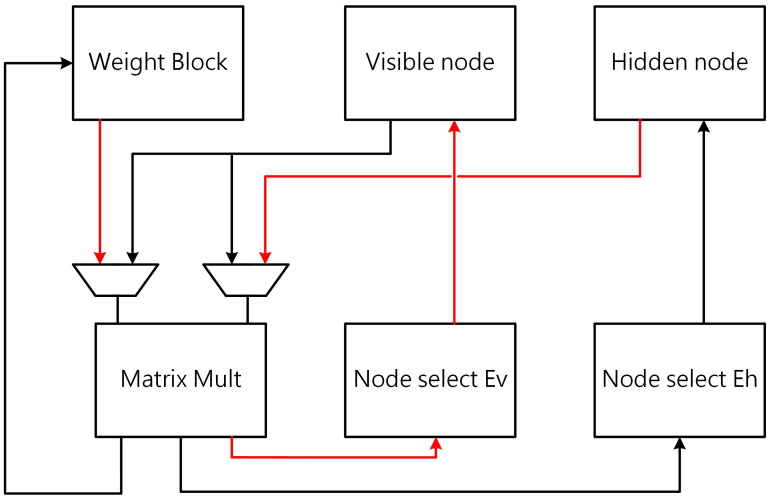

Figure 3.4: A data flow diagram for the reconstruction phase

node and hidden node values from both the first and third AGS phases will be feed into the matrix multiplication core to perform the multiplication operations using Eq. (2.18). The weights are also fed into the matrix multiplication core to calculated the new weights. Once the new weights are calculated, weight block will update its memory contents accordingly.

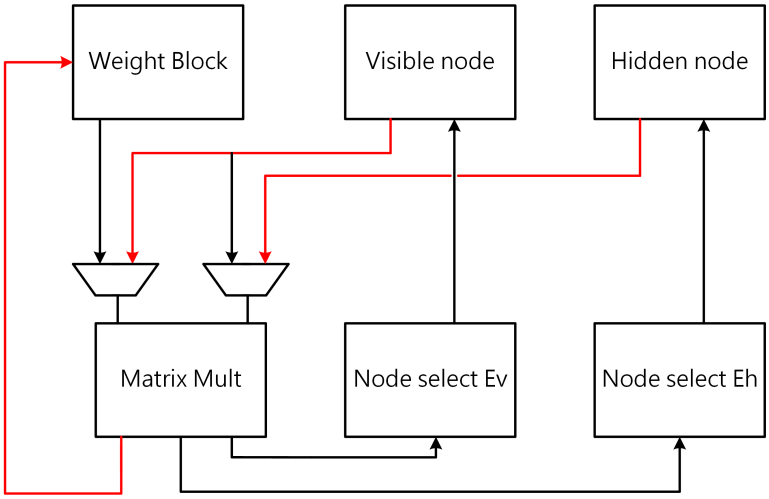

Figure 3.5: A data flow diagram for the weight update phase

Unlike the control unit for computational core, the state machine for I/O interface is much simpler. As shown in Fig. 3.6, the state machine only consists of two states, $Idle$ and $ReadData$. The state machine starts with the $Idle$ state, and spins on that state until the $fetch$ signal is high. During

the *ReadData* state, the control unit will generate the address for the data to be read, and store incoming data into a shift register. Each input vector contains 784 of 8-bit words and the I/O bandwidth is only 16 bits, therefore state machine will be spinning on *Read Data* for at least 392 cycles. Once the input vector is ready, it will generate a *dataready* signal, and move back to the $Idle$ state waiting for another $fetch$ signal.

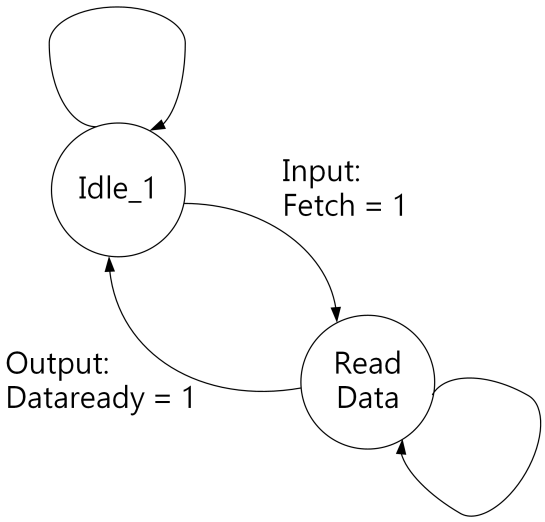

Figure 3.6: The state machine of I/O interface

3.3 Stochastic Node Selection Design

This module calculates the node value and node state using a sigmoid function and a random number generator. To implement the sigmoid function is very difficult in hardware. A naive approach requires both exponential functions and division, and these two operations are very expensive to implement using hardware. However, the sigmoid function is amenable for hardware implementations. First, the range of the function is bounded in the interval (0, 1). As a result, floating-point representation is not required and can be replace with fixed-point representation. Second, the function has odd symmetry. Thus, computing half of the domain is sufficient to generate the remainder of the domain.

3.3.1 Piecewise Linear Interpolator

Originally, a ROM-based look-up table implementation was used. It is an efficient method to provide reasonable approximation for bounded transfer function. The values for the function are precomputed. Then function is then evenly sampled and the sampled data is stored in an on-chip ROM. This is efficient, but it only provides limited resolution. For a 2 kB on-chip ROM with 32-bit output, can only have 512 sampled entries, meaning there is only 9-bit resolution for input values.

To increase the resolution, we implemented the an interpolation that was proposed by Ly and Chow [68] to increase the resolution by operates on the two boundary outputs of a look-up table. The implementation uses linear interpolator as shown in Eq. (3.1), where (u, v) represent the desired point between points (x_0, y_0) and (x_1, y_1).

$$v = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(u - x_0) + y_0 \tag{3.1}$$

The naive hardware implementation of Eq. (3.1) requires both division and multiplication which utilized significant amount of hardware resources. Thus, rather than calculating the interpolation exactly, a recursive piecewise implementation was used. Knowing the midpoint, which can be found by adding the endpoints and right shift by one, the search point is iteratively compared with the midpoints. This implementation gives a good approximation while using little hardware overhead. Furthermore, the design can be easily pipelined.

This hardware implementation is called the kth Stage Piecewise Linear Interpolator (PLI^k), where each successive stage does one iteration of a binary search for the desired point. A comparison of PLI^2 and LI and corresponding error is shown in Fig. 3.7. A detailed schematic diagram of the PLI^k architecture is shown in Fig. 3.8.

Using the ROM-based look-up table and PLI^k, a pipelined, high-precision, and resource efficient sigmoid transfer function was implemented. Using fixed-point input data, the sigmoid function is defined as a piecewise implementation using Eq. (3.2). A comparison between the ideal sigmoid function

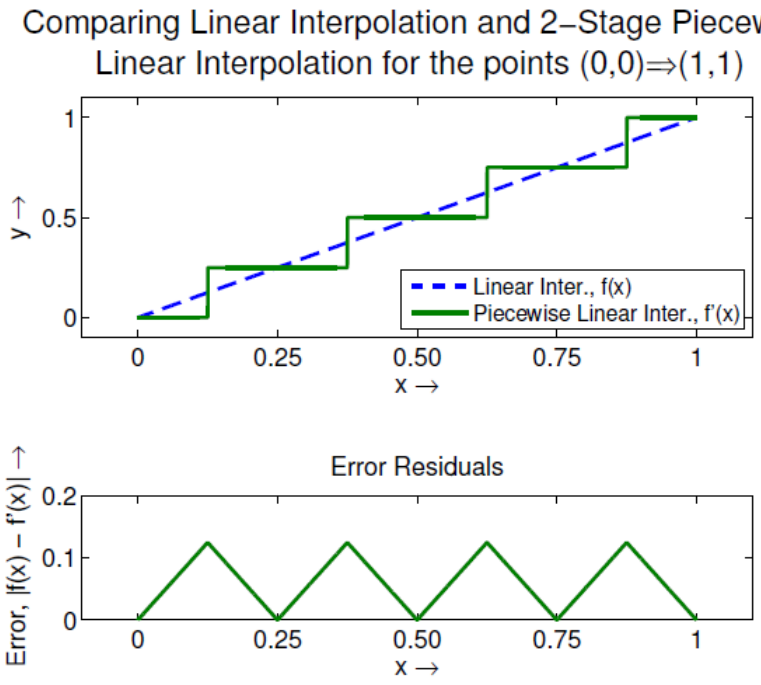

Figure 3.7: Comparison and error residuals of LI and PLI^2 [68]

the piecewise implementation and the error residuals are shown in Fig. 3.9.

$$f'(x) = \begin{cases} 0, & x \leq -8 \\ 1 - PL, I^3(LUT(-x)), & -8 < x \leq 0 \\ PLI^3(LUT(x)), & 0 < x \leq 8 \\ 1, & x > 8 \end{cases} \tag{3.2}$$

Finally, the results of the sigmoid transfer function is compared with a random number to select the final node state. There are many efficient FPGA implementations of uniform random number generators [77], [78]. A Linear Feedback Shift Register (LFSR) is implemented for this RBM architecture due to its simplicity. The block diagram for the node select module for computing the hidden node states is shown in Fig. 3.10.

Although the node select module for computing visible nodes is very similar to the module used for the hidden nodes, it is much larger compared to the node select module for the hidden nodes. This is due to the fact when partial energies for hidden nodes are computed, the RBM computational core outputs one E_h every clock cycle. Since the node select module for E_h is pipelined, thus only one piecewise sigmoid function core is need. When the

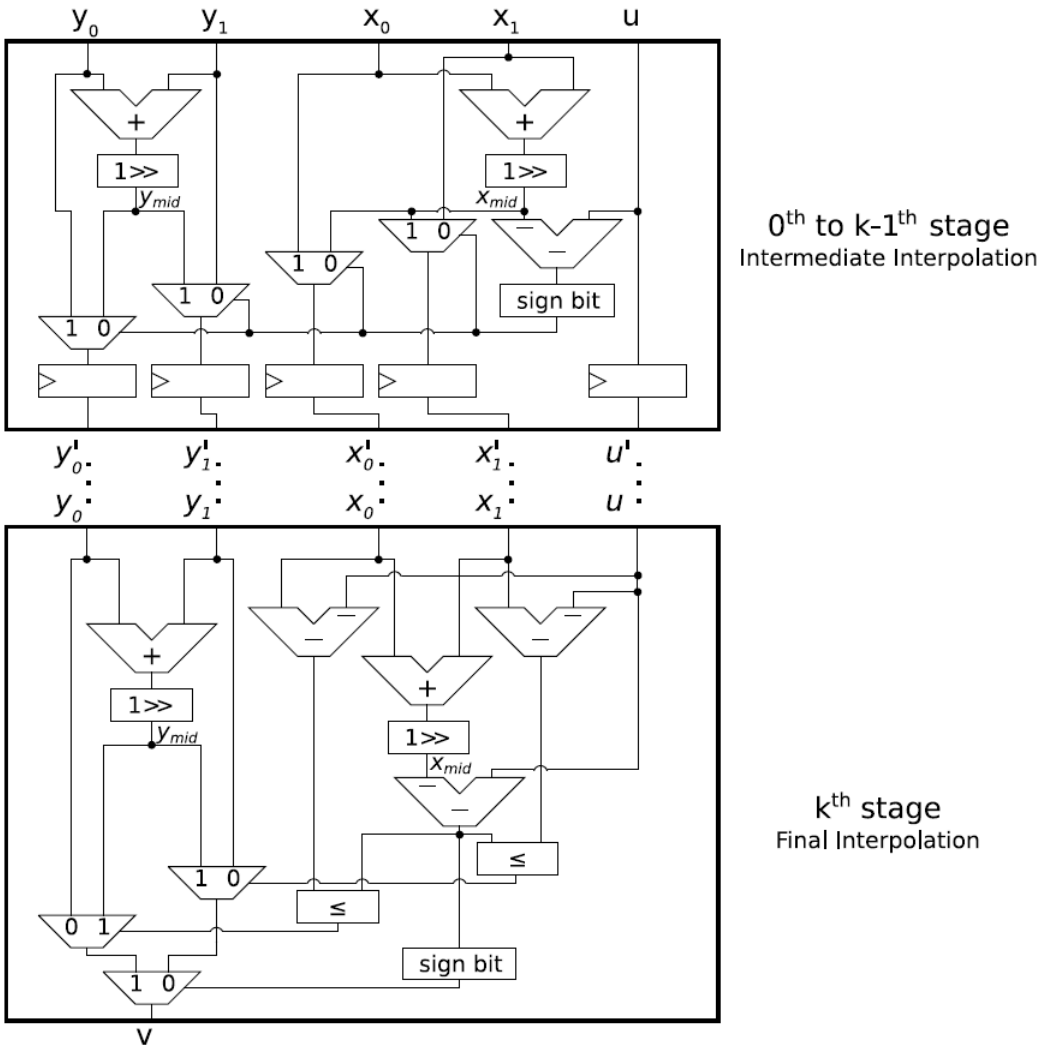

Figure 3.8: A schematic diagram of the PLI^k [68]

partial energies for visible nodes are calculated, 784 of them are computed in parallel. As a result, the node selected module for E_h contains 784 of the piecewise sigmoid function core. This implementation is later optimized, and will be discussed more in detail in Chapter 4.

3.4 Memory Core

To understand how our implementation resolve weight transpose problem, the key observation required is that matrix multiplications can be viewed as multiple linear combinations of vectors, multiple vectors inner products, or as

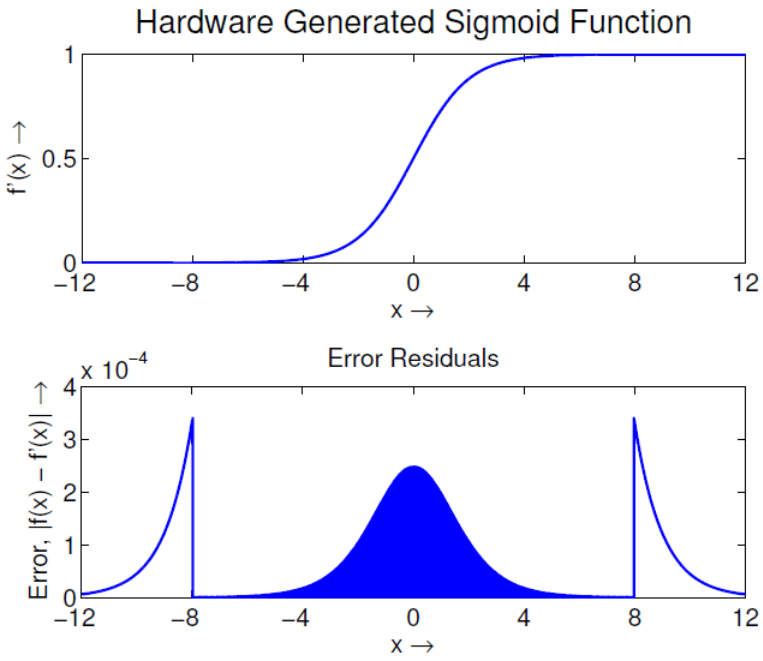

Figure 3.9: A schematic diagram of the PLI^k [68]

a sum of vector outer products. If the construction phase (VW) is viewed as vector inner products, then each row of V and each column of W is multiplied element-wise, followed by a sum reduction. This suggests that each row of W should be placed in separate on-chip RAMs so that all of these elements can be read simultaneously, as shown in Fig. 3.11(a). For the reconstruction phase, HW^T, consider the transposed matrix operation (WH^T), and view the operation as linear summation of vectors. This requiers that the jth column vector of W is multiplied by the jth element in a column vector of H^T. This gives the structure of Fig. 3.11(b), which compute multiple visible neurons in parallel. Since at each cycles we only need to read a column vector of W for both phases, the memory layout for the weights can remain the same, and it requires no communication for a transpose operation. In work done be Kim et al. [67], they have a similar implementation to store the connection weights. They placed each column of W onto separated on-chip RAMs while our implementation placed each row of W onto separated on-chip RAMs. Even though the difference between the implementations seems to be insignificant, it makes a huge difference on performance. When Kim et al. [67] designed their implementation, they mostly consider for neural networks with symmetric size. Thus, storing a row or a column of their

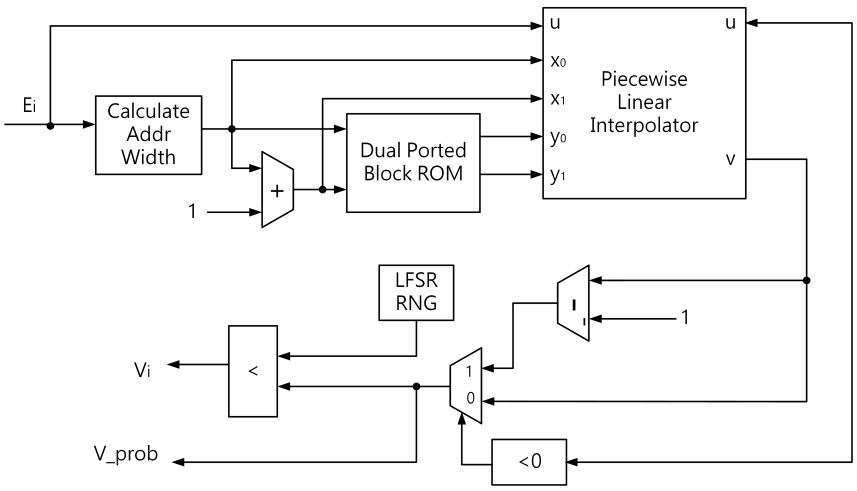

Figure 3.10: A schematic diagram of the node select module for computing hidden node

implementation does not make a difference. However, for our design, we are dealing mostly with asymmetric neural networks, where the visible layer is much larger than the hidden layer. Suppose we are storing each column of W onto separate on-chip RAMs, we are only able to process ten multiplication operations in parallel. But if we are storing each row of W onto different on-chip RAMs, we are able to process 784 multiplication operations in parallel, that is more than 78 times faster.

3.5 Matrix Multiplication Core

The matrix multiplication core consists of two components: array of multipliers and array of adders. Since we are trying to process 784 multiplication operations in parallel, we will need 784 of 18-bit multipliers. Depending on AGS phase, the input multiplexer will select appropriate input from connection weight, hidden layer, and visible layers to perform multiplication operations. Once the multiplication operation is performed, the output from the multiply will be feed into array of adders to either perform sum reduction or linear summation of vectors.

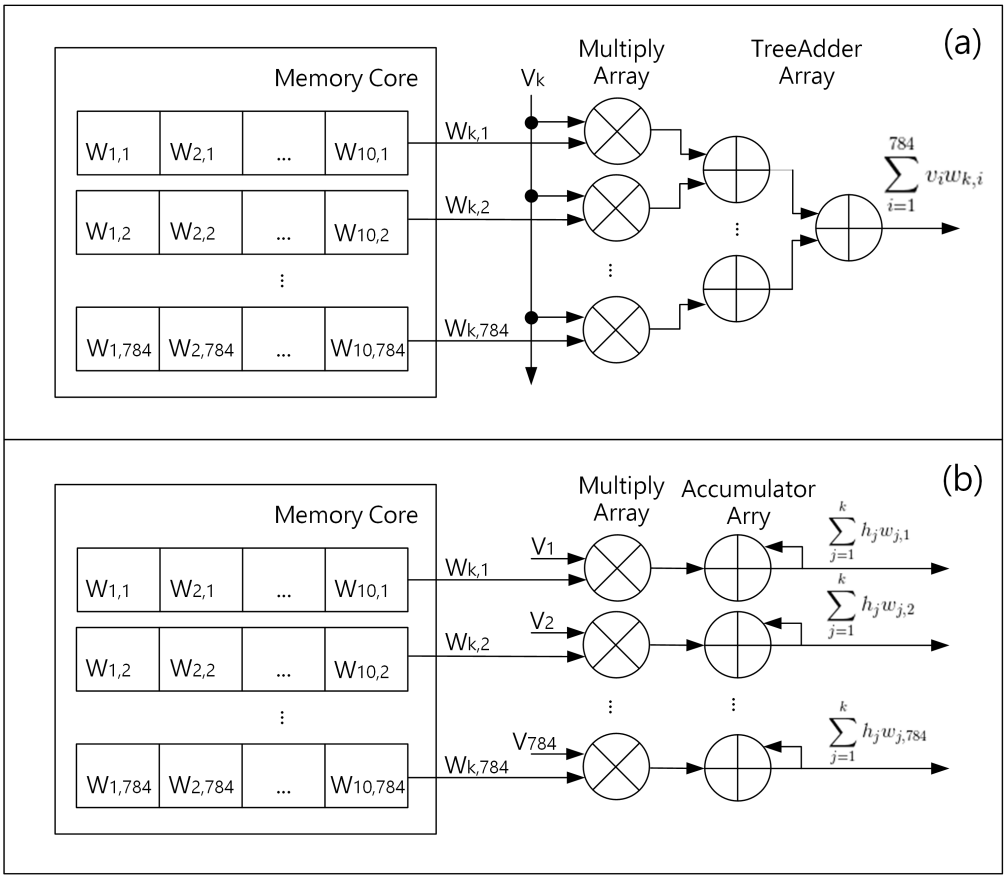

Figure 3.11: (a) Matrix multiplication for computing hidden nodes, (b) matrix multiplication for computing visible nodes

3.5.1 Tree Adder and Accumulator

In order to perform the sum reduction and linear summation of vectors, a tree adder and accumulator are need. To perform a sum reduction of 784 numbers, ten levels of tree adders are needed. On the other hand, to perform a linear summation of vectors with 784 elements, 784 of 18-bit accumulators are needed. To have the tree adders and accumulator adders implemented separately, a significant number of full adders are need from the FPGAs. Since the tree adder and accumulator are needed for different AGS phases, they will never be used in the same cycles. Thus one solution to decrease the number of adders needed is to combine the first level of tree adders with the accumulator adders. The those adders will have multiplexers at one of their inputs. The control signal for the multiplexers will decide which mode the adder will perform. By combining two adders together, we are able to save 392 full adders. The block diagram for tree adders and accumulators is

shown in Fig. 3.12.

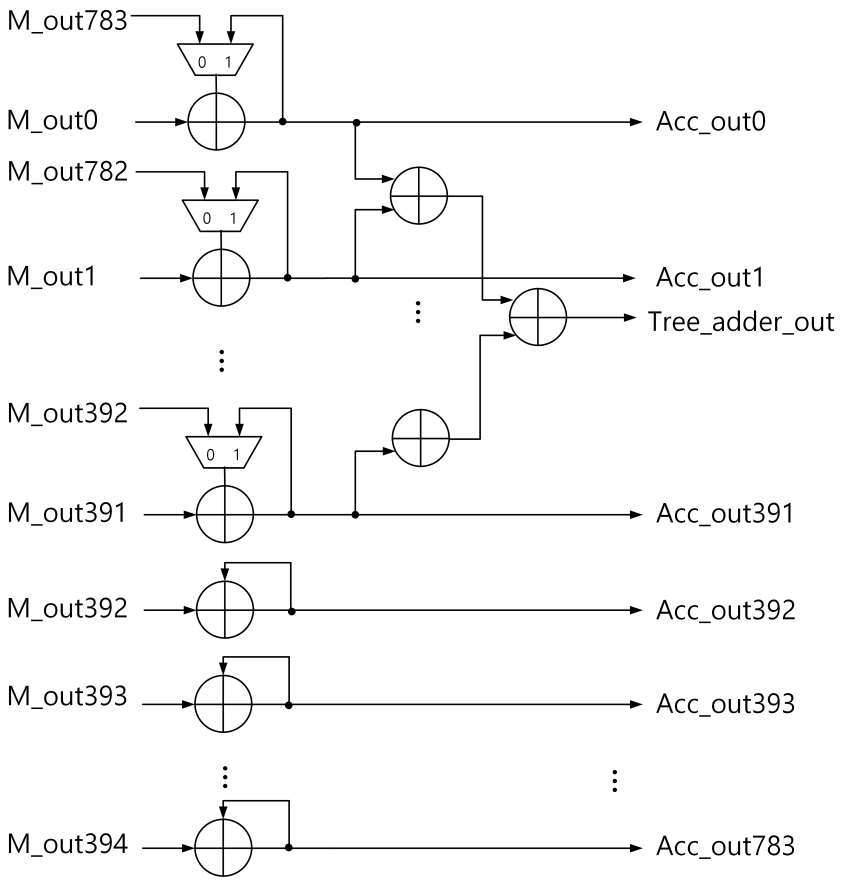

Figure 3.12: A schematic diagram for combined the tree adders and accumulators

3.6 Visible Nodes

The visible node module contain three components, a shift register and two registers. The shift register has a width of 8 bits and depth of 784 operating at 200 MHz. It serves as a temporary storage for reading the next input data from the SRAM. Once the shift register contains one input vector, bit-wise operation will be performed on its value to match the data type in the RBM computational core and then loaded into the other two registers. The other two register are exactly the same size, 18 bits in width and 784 in depth. Unlike the shift register, they are running at 100 MHz. One is used to store the visible node value at the first AGS phase and another is used to store the

visible node value at the Xth AGS phases. Instead of storing the temporary ΔW at first AGS phase, we decided to store V^1 and H^1 so that the ΔW can be calculated at once during the weight update phase. This way, we can save a lot of memory for temporary storage.

3.7 Hidden Nodes

Similar to the visible node module, the hidden node module also contain three components, that is, three registers. The first two registers are similar to the registers used for the visible node. They have 18 bits of widths, and 10 in depth. Since the label for each handwriting image is represented by a 10-bit vector, there are only 10 nodes in the hidden layer. These two registers are used to store the hidden node values at the first AGS phase and the Xth AGS phases. The third register has 1 bit in width and 10 in depth. It is used to store the final state (0 or 1) of each hidden node when RBM is used to testing against untrained data.

3.8 I/O Interface

I/O interface is responsible for reading input data from the SRAM. The control unit for I/O interface running at 200 MHz will generate the address for the SRAM. Since the input data for each pixel image is 8 bits, and the bandwidth of the SRAM is 16 bits, thus we are able to read two pixels of data every clock cycles. At each clock cycle, the 16-bit data from the SRAM will be shifted into a shift register that has 784 8-bit registers connected together. After reading all 784 pixels data from the SRAM, the shift register will load its values into the visible node module. The I/O interface is also responsible to write the final connection weight values from the on-chip memory to the SRAM after the training is done.

CHAPTER 4

OPTIMIZATION

There are two major optimizations done to the implementation to minimize the hardware resource utilization without diminishing the performance.

4.1 Independent Multiplier vs. Two-Multiplier Adder Mode

During the construction phase, the visible layer is multiplied by each column of weight element-wise; during the reconstruction size, the jth element in the hidden layer is multiplied with the jth column in the weight; during the weight update phase each hidden node is multiplied with a vector of visible node. As a result, to achieve the maximum performance we need 784 18-bit by 18-bit multipliers. However, as indicated by the data sheet as shown in Fig. 4.1, we do not have enough independent 18×18 bit embedded multipliers on FPGA to support this implementation. The board is only able to support

Family	Device	DSP Blocks	Independent Input and Output Multiplication Operators					High-Precision Multiplier Adder Mode	Four Multiplier Adder Mode
			9 × 9 Multipliers	12 × 12 Multipliers	18 × 18 Multipliers	18 × 18 Complex	36 × 36 Multipliers	18 × 36 Multipliers	18 × 18 Multipliers
Stratix IV E	EP4SE230	161	1,288	966	644	322	322	644	1288
	EP4SE360	130	1,040	780	520	260	260	520	1040
	EP4SE530	128	1,024	768	512	256	256	512	1024
	EP4SE820	120	960	720	480	240	240	480	960
Stratix IV GX	EP4SGX70	48	384	288	192	96	96	192	384
	EP4SGX110	64	512	384	256	128	128	256	512
	EP4SGX180	115	920	690	460	230	230	460	920
	EP4SGX230	161	1,288	966	644	322	322	644	1288
	EP4SGX290	104	832	624	416	208	208	416	832
	EP4SGX360 *(1)*	130	1,040	780	520	260	260	520	1,040
	EP4SGX360 *(2)*	128	1,024	768	512	256	256	512	1,024
	EP4SGX530	128	1,024	768	512	256	256	512	1,024

Figure 4.1: Data sheet for Altera Stratix IV GX230

644 of independent 18×18 bit embedded multipliers, which is not enough to process 784 nodes at the same time. However, if we switch to four-multiplier-adder mode, it can support 1288 embedded multipliers. The reason for this shown in Fig. 4.2. Both Fig. 4.2(a) and (b) represent half of the DSP block, but with different operation mode. The output of two multipliers connect to an adder, and there is no output port that directly outputs the value from the multipliers. As a result, when the DSP block operates at independent multiplier mode, half of the multipliers on the DSP block are not used in order to propagate the value from the multiplier through the adder. When DSP operates at four-multiplier adder mode, we are able to utilized all the embedded multipliers. But since the multiplier has a extra adder, the order of operation will need to be modified accordingly. Since all independent

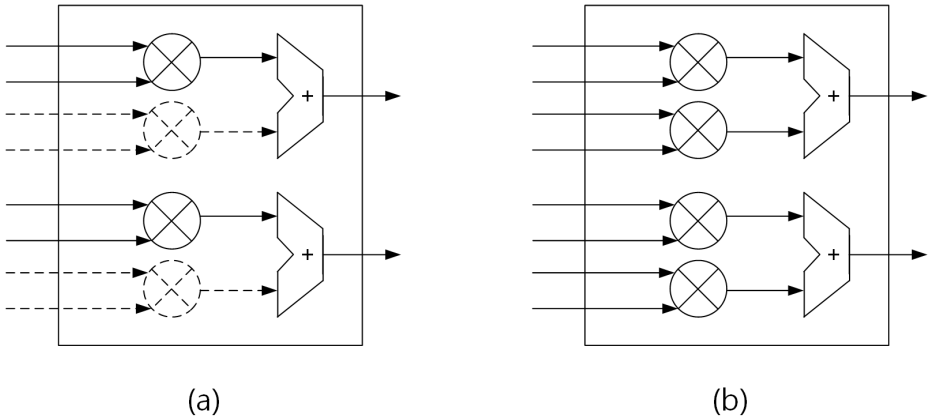

Figure 4.2: (a) Independent multipliers and (b) four-multiplier adder mode

multipliers are replaced with four-multiplier-adders, matrix multiplication for each AGS phase needs to be modified accordingly.

- Construction phase: The original implementation was to have the visible node multiply by each column of weight in a element-wise manner, and then perform a sum reduction with a 10-level of tree adders. Now having an extra adder that sums up the output of two multipliers in pairs, the tree adder only need to be adjust from ten levels to nine levels. Then the construction phase should behavior correctly.

- Reconstruction phase: The original implementation was to have the jth element of the hidden nodes multiply with the jth column of weights. Now with the extra adder, we reorder the matrix multiplication so that

at each clock cycle, the jth and $(j+1)th$ element of the hidden node multiply with the jth and $(j+1)th$ column of the weights. Then sum the products if the weights from both column belong to the same row. Since we only have 784 multipliers, we have only half jth and $(j+1)th$ column multiply and adding with jth and $(j+1)th$ element of the hidden node in parallel. Originally, 784 partial energies for the visible nodes are calculated simultaneously, and after ten clock cycles, the partial energies are ready. Now with a extra adder attached multipliers, 392 of partial energy for visible node is calculated simultaneously, and after 5 clock cycles, 392 of partial energies are ready. Thus overall cycles for reconstruction phase still stays the same. In order to make this implementation works, the RAM for store connection weight need to be change to dual ports since we are reading two connection on the same row.

- Weight update phase: In order to calculate the ΔW, the hidden node and the visible node from the first AGS phase feed into one multiplier, while the corresponding hidden and visible node from the third AGS phase feed into the second multiplier in the same DSP block. Since the ΔW is the difference between the two products instead of the summation, we could simply negate the hidden node value from the third AGS phase, or simply store the negation of the hidden node value into the hidden layer during the second construction phase. This way 392 of ΔW can be calculated every cycles. To further optimized the implementation, the connection weight can be read out at the same time ΔW is calculated. Then the first level of tree adder can be use to find the new weight and write back the next cycle. Since the RAMs for storing connection weights are now modified to have dual ports, one port can be use for read, while another port can be used for writing in the new connection weights.

4.2 Activation Function

Since the partial energy for visible nodes are calculated as linear summation of vectors, the partial energy for each visible node is processed simultaneously.

As a result, 392 of the partial energies are ready at the same cycle. In order to find the sigmoid function of all of them in parallel, a naive approach is to instantiate the same node select module for Eh 392 times. Since the random number generator is not needed for finding the new visible node states, that part of hardware can be removed to save hardware utilization. After removing all other unnecessary hardware, a module called activation function, which calculates only the sigmoid function, is shown in Fig. 4.3. After optimizing

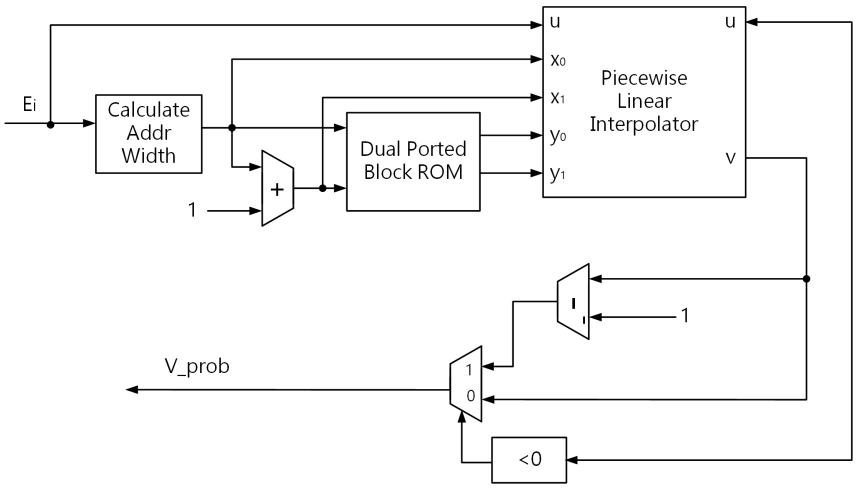

Figure 4.3: A schematic diagram of the activation function

the hardware usage, the activation function module is instantiated it 392 times. The block diagram of node select for Ev with naive approach is shown in Fig. 4.4. However, even after removing all the unnecessary hardware, the node selection for Ev module still requires about 180,000 logic elements in total. This naive approach takes about 81% total logic elements of the entire board, which makes the entire implementation impossible to fit on one FPGA board.

In order to reduce the total resource of the node select module for visible nodes, two shift registers are introduced, one at front and another one at back of the activation modules. A shift register with parallel loads, as shown in Fig. 4.5, is added in front of the activation function module. The shift register is four words in depth, and each word is 18 bits wide. When partial energies are calculated, they will be grouped into 98 groups with four partial energies in each group. The partial energies in each group will be loaded into the shift register using parallel load. Then, each partial energy will be shifted into the activation function one by one at every clock cycles to calculates its

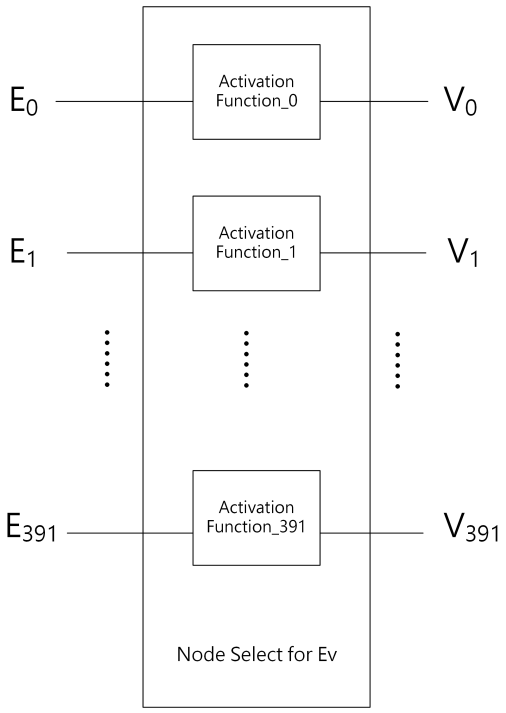

Figure 4.4: A schematic diagram of node select with activation function module

corresponding sigmoid function value. Since the activation function module is pipelined, it is able to handle a new input at every clock cycle. Similarly, a shift register with the same depth and width without any parallel load, as shown in Fig. 4.6, is also added at the end of the activation module. It is used to store the visible node value until all four values are outputted by the activation function modules. Then the values stored in the shift register will be loaded into the visible node module simultaneously.

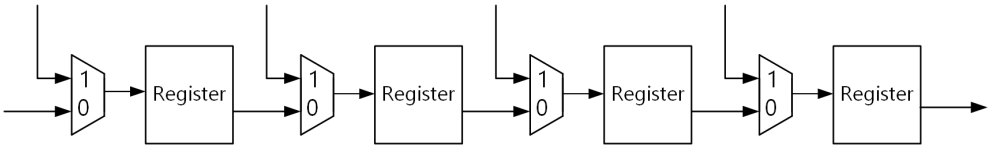

Figure 4.5: Shift register with a parallel load

With these two structures added for the activation function modules, we are able to reuse each activation function module four times. As a result, only 98 activation function modules are needed for the node selection module for calculating the partial energies of the visible layer. With the optimization, we are able to reduce the logic element by 80,000 for this module and only

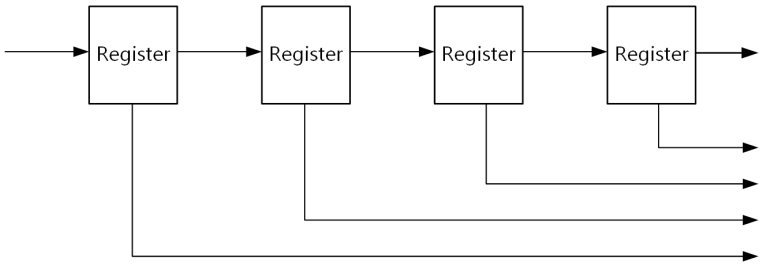

Figure 4.6: Shift register without a parallel load

adding eight clock cycles to the entire AGS phase. The final schematic of the node selection for visible node is shown in Fig. 4.7.

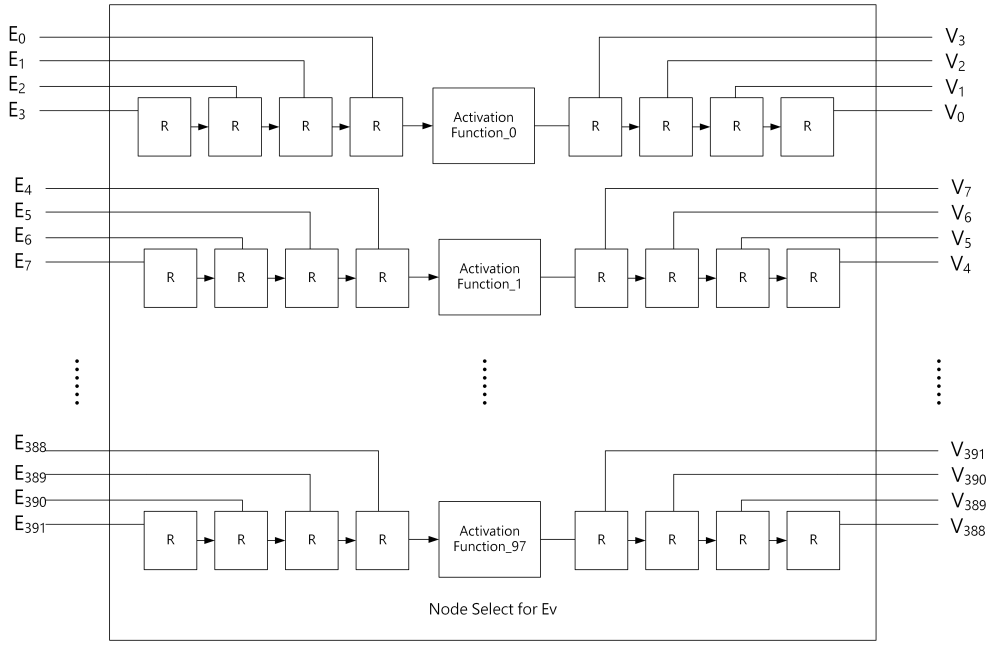

Figure 4.7: A schematic diagram of the optimized node selection module for E_v

CHAPTER 5

EXPERIMENTAL RESULTS

Due to the fact that different FPGA development boards and neural network architectures used across different research studies, there is a lack of a standardized benchmark for comparing FPGA implementations. At the same time, a majority of hardware accelerated platforms are designed for a specific application in mind. As a result, an in-house application is usually used as a point of comparison.

Since the previous works used the MATLAB implementation as a baseline, we used the same MATLAB implementation of the RBM from Hinton et al. [63] to compare the performance of our implementation. The benchmark used in the experiments is a very popular handwritten digit recognition database called MINIST. It has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from the NIST. All the digits in the training set have been size-normalized and centered in a 28 $\times$ 28 pixel image. A few sample images from the MNIST data set are shown in Fig. 5.1.

In this chapter, we tested our implementation with different AGS limits and different network sizes. We record the training performance with and without I/O time. The results are compared with the MATLAB implementation and previous works. The hardware recourse needed for this implementation are also reported in detail and compared with previous works. The rest of the chapter is organized as follows.

- The different metrics that used to measure the performance of RAW will be described.
- The resource utilization of the implementation will be reported.
- The performance comparison with the MATLAB implementation will be presented.

- The performance comparison between different platform will be discussed.
- The result of scalability test of RAW will be provided.

Figure 5.1: Sample training input images from the MNIST dataset

5.1 Metrics

For performance, the lack of standard neural networks metrics raises some issues. Although an absolute measure of performance is desired, there are no metrics that can account for the difference in neural network architectures. An effective metric for comparing computational performance of a single type of neural networks is Connection-Update-Per-Second (CUPS) [59]. It measures the rate of weight changes during the learning process. It also measures how fast a system is able to perform input-output mappings. For an RBM architecture, CUPS is defined as the number of weights divided by the periods for one complete AGS cycle, T. The equation for symmetric networks is in Eq. (5.1), where n is the size of visible and hidden layer. The equation for asymmetric networks size is in Eq. (5.2), where m is the size for the visible layer and k is the size for hidden layer.

$$CUPS = \frac{n^2}{T} \tag{5.1}$$

$$CUPS = \frac{m \times k}{T} \tag{5.2}$$

For comparing two different implementations of the same neural network architecture, the *update period* is also a powerful metric to compare the performance. It measures the time that the implementation takes to complete a single batch of data. Thus, the higher the *update period* value is, the slower the implementation performs.

Another simple and effective metric is the speed-up. This measures the ratio between the times to complete training neural networks using different implementations. The equation for calculating the speed-up between the software implementation and the hardware implementation is given in Eq. (5.3), where S represents the speed-up, T_{sw} represents the update period for the software implementation, and T_{hw} represents the update period for the hardware implementation.

$$S = \frac{T_{sw}}{T_{hw}} \tag{5.3}$$

5.2 Resource Utilization

The entire implementation contains 11 entities. The resources that used for each entity are reported in Table 5.1. Among all the entities, the *matrix_mult* and *node_select_ev* require the most number of combination ALUTs. The is because that complicated logic is used for the PLI inside the *node_select_ev* and a large number of multiplexers are used for selecting signals for multipliers and adders inside the *matrix_mult* module. The *node_select_ev* and *weight_block* use the majority of memory bits needed for this design. This is because that the large number of RAMs and ROMs used for storing connection weights and the sigmoid function look-up table. On the other hand, the *matrix_mult* module used all the DSP blocks needed for the design since all the multiplication is done inside the *matrix_mult* module.

5.3 Performance Comparison

The benchmark implemented on Altera Stratix IV GX(EP4SGX230KF40C2) FPGA with the computational core running at 100 MHz and I/O interface

Table 5.1: The resource utilization for different modules

Entity	Combinational ALUTs	Registers	Memory Bits	DSP blocks
RBM_control	229	48	0	0
hidden_node0	0	10	0	0
hidden_probs0	7	7	0	0
hidden_probsx	0	11	0	0
matrix_mult	52,935	10,638	0	784
node_select_eh	417	322	9,216	0
node_select_ev	36,652	43,806	903,168	0
visible_data0	0	6,272	0	0
visible_node0	0	3,136	0	0
visible_nodex	0	14,125	0	0
weight_block	3	0	225,792	0
Total	94,174	78,375	1,138,176	784

running at 200 MHz. The performance measurements are done in comparison against the MATLAB implementation trained on an Intel Core i5 CPU running at 2.5 GHz with a double-threaded version of the RBM application. When the AGS limit is 3 and the size of the visible layer is 784, the speed-up for pure CPU time is 161, while the speed-up with I/O time considered is 134. The difference between two speed-ups is due to the fact that the time for computing one data vector is faster than the time for reading out one data vector from the SRAM. The computational core has to halt and wait for the I/O interface to finish reading the next input vector. A graph of the speed-up vs. AGS limits to train each data vector is shown in Fig. 5.2. Three data points in the figure correspond to AGS limit 3, 5, and 7. As shown in Fig. 5.2, the marginal gain of speed-up decreases as the number of AGS increases for each input data vector. This is because after certain AGS limit, the I/O interface are able to fetch the next data before the computational core finishing processing the current data.

5.4 Platform Comparison

The resource comparisons between our implementation and previous works are reported in Table 5.2. Ly and Chow used the fewest number of registers and zero DPS block. This is due to the fact that their implementation only

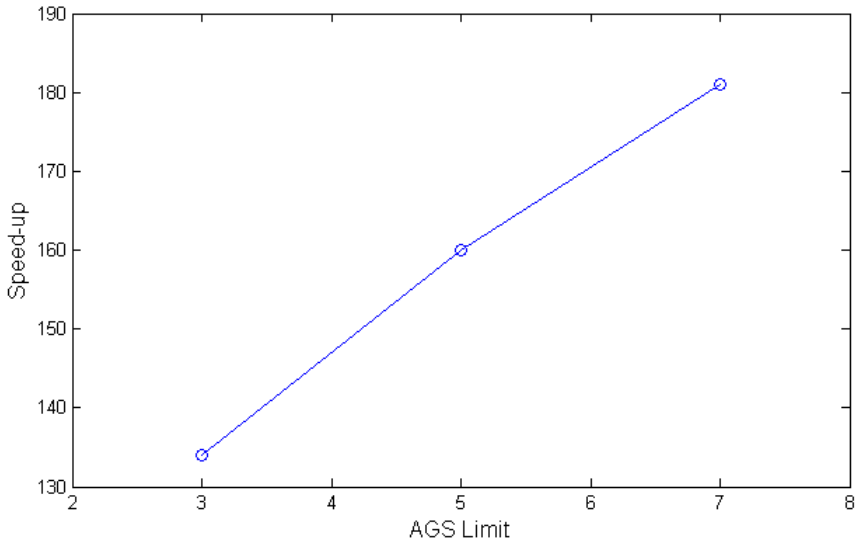

Figure 5.2: The comparison of speed-up between different AGS limits; the marginal gain of speed-up decreases as the number of AGS limits increases

Table 5.2: The resource utilization between different implementations

Design	Register	Combinational ALUTs	Block Memory	DSP blocks
Kim [67]	160,996 (60%)	107,097 (53%)	9,560,288 (57%)	288 (100%)
Ly [68]	29,885 (45%)	30,403 (45%)	4,626,000 (78%)	0 (0%)
RAW	78,374 (43%)	94,174 (52%)	1,138,176 (8%)	784 (61%)

works with binary data type, but does not support real-value data, which allows their implementation to replace multiplication operations with AND and add operations. Our implementation uses the highest number of DSP blocks due to the high level of parallelism that we can achieve. As a result, we are able to obtain a higher speed-up and CUPS compared to their reported results. Our implementation also uses the fewest number of memory blocks. The reason for this is that instead of storing the intermediate values for ΔW, we stored the intermediate values for the visible and hidden nodes, which saves about 90% of the memory blocks. Overall, RAW uses less memory blocks, while using comparable hardware resources in other categories.

The speed-up comparisons between our implementation and previous works are reported in Table 5.3. Kim et al. [67] and our work both use the MAT-

Table 5.3: The comparison of various RBM implementations

Design	Ly [68]	Ly [68]	Ly [68]	Kim [67]	RAW
Platform	1 FPGA	4 FPGAs	Virtualized FPFGA	1 FPGA	1 FPGA
Network Size	128×128	256×256	256×256	256× 256	784×10
Clock Speed	100 MHz	100 MHz	100 MHz	200 MHz	100 MHz
Absolute Performance (CUPS)	1.58 G	3.13 G	725 M	-	4G
Relative Performance	61×	145×	32×	35×	160×
Baseline Platform	C	C	C	MATLAB	MATLAB

LAB implementation as the baseline, and our implementation is able to achieve 3 times more speed-up compared to their result. When comparing our results with Ly and Chow [68], the speed-up of our implementation is still higher. Even though the C implementation might run faster than the MATLAB implementation and comparing the relative performance might not be accurate, our implementation is still able to achieve a much better absolute performance. Our implementation with only one FPGA is able to achieve a better performance compared with the implementation of Ly and Chow [68] with four FPGAs. As a result, our implementation is optimized for handwriting recognition in terms of speed compared to previous works.

5.5 Scalability

The resource utilization is a good metric to measure the scalability of the architecture. To show the scalability of our design in terms, we adjusted our implementation so that it is able to train the inputs with different network sizes. The hardware resources needed for each network size are reported in Table 5.4. The percentages of the resource utilization for combinational ALUTs, registers, memory bits, and DSP blocks are shown in Fig. 5.3. As the size of the visible layer increases, the number of registers needed for the implementation increases linearly, while the combinational ALUTs grow linearly too but at a smaller rate. The number of required DSP blocks leveled off when the size of the visible layer is equal to 784. This is due the fact that

the implementation for the matrix multiplications is pipelined. Thus, the DSP blocks can be easily reused as the size of the visible layer increases. On the other hand, the memory bits that are needed for this implementation increase gradually, only 1 percent increased as the size of the visible layer doubled.

Table 5.4: The resource utilization for different network sizes

Network size	Combinational ALUTs	Register	Memory Bits	DSP Blocks
1568×10	127,103 (69%)	130,154 (71%)	1,363,968 (9%)	784
784×10	94,174 (52%)	78,375 (43%)	1,138,176 (8%)	784
392×10	71,450 (39%)	52,497 (29%)	1,025,280 (7%)	392

To show scalability of our implementation in terms of the update period and the speed-up, we ran the benchmark on different sizes of neural networks and compared the result with the runtime of the MATLAB implementation. Figure 5.4 shows the update period for both software implementation and FPGA implementation. As the size of the visible layer increases, the update period for software implementation increases in an exponential manner, while the update period for FPGA implementation barely increases.

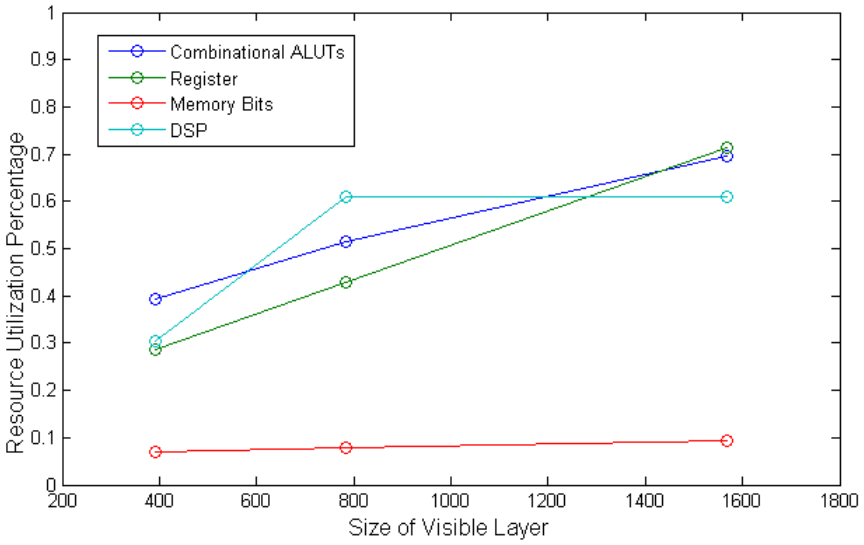

Figure 5.3: The resource utilization for different network sizes; the resource utilization is able to scale well as the network size increases

Figure 5.5 shows the speed-up comparision between the FPGA and the

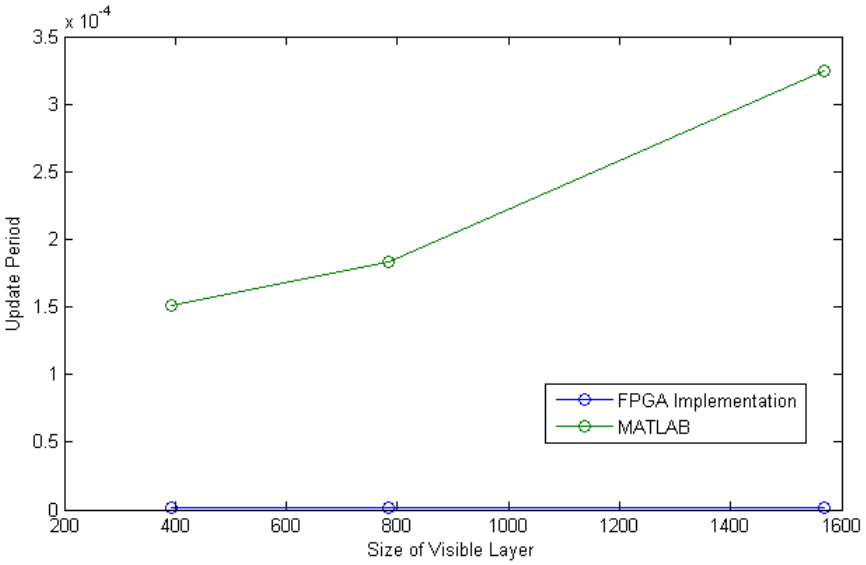

Figure 5.4: The update period comparison for different visible layer sizes; the networks size affects the update period of the software implementation greatly while it barely affects the update period of RAW implementation

MATLAB implementation for different sizes of the visible layers. As the size of the visible layer increases, the speed-up grows almost exponentially.

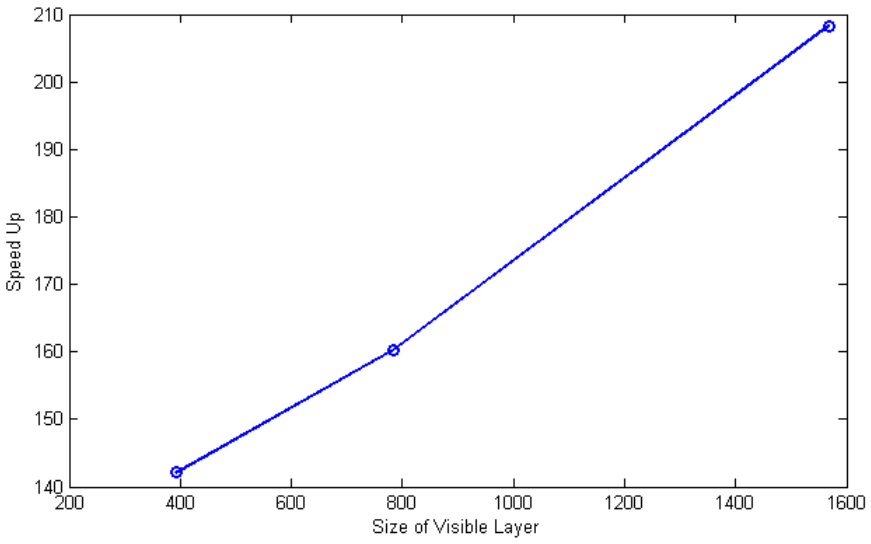

Figure 5.5: Speed-up for single FPGA platform with different visible layer sizes compared to the software implementations; the speed-up increases exponentially as the network size increases

CHAPTER 6

CONCLUSION

6.1 Conclusion

The goal of this thesis was to implement a high-performance restricted Boltzmann machine optimized for handwriting recognition application on FPGA to accelerate the training performance of RBM. Even though there have been many attempts to design hardware implementation of various neural networks architectures, work done by Ly and Chow [68] and Kim et al. [67] are the main interest to this thesis. When implementating RBMs on FPGAs, one of the major issue is the weight storage. Depending on different AGS phases, Wor W^T will be needed to calculated the partial energies. In order to speed up the matrix multiplication operation, a row and a column need to be accessed at the same time so that the multiplication can be done in parallel. Thus the distribution of the weights is a non-trivial problem due to the transpose operation that occurs during the reconstruction phase. Ly [68] and Kim [67] proposed two interesting implementations to resolve the weight transpose problem. However, there are two major setbacks in their implementations used for handwriting application. First, their implementations are based on the assumption that connection weights have a symmetric structure and the network has the same number of visible nodes and hidden nodes. However, in the case of handwriting recognition where the visible layer is much larger than the hidden layer in size. Thus, their implementations would simply not work or would be highly inefficient. Thus in this thesis, we proposed a different solution based on Kim's work to solve the weight transpose problem.

In addition, the hardware implementation of each module was described in great detail. To further improve the resource utilization and computational performance, the four-multiplier-adder mode was used to replace the inde-

pendent 18×18 bit multipliers. The shift register structure is also introduced to the node selection module for the visible layer to reduce the hardware utilization by approximately 80,000 logic elements while only adding eight additional clock cycles to the entire design.

The implementation was compare to the MATLAB implementation running on a 2.5 GHz Intel i5 Processor. The popular database of handwriting digits called MNIST is used as the benchmark to preform the experiments. The experimental results show that our implementations are able to achieve 4 billion CUPS resulting in a speed-up of 134 fold compared to the software implementation when considering I/O time, and it is able to achieve 161 times speed-up without the I/O. These results are much higher when compared with previous works, while the area needed is very comparable with theirs. Therefore, RAW is more optimized for handwriting recognition application.

6.2 Future Work

For future works, we need to further improve the scalability of our design, adding real-time recognition features and extending the implementation to other applications.

6.2.1 Improving Scalability

Right now, the designs are implemented in a way that is very difficult to scale to different sizes. Instead of having a big block of modules, the design should be divided into smaller modules so that the size of the network can be easily modified. The maximum size that can be implemented on a single FPGA is still very limited. If we can extend our current implementation into multiple-FPGAs implementation, that would improve the scalability of the implementation significantly. In order to have the RBM implemented on multiple FPGAs, we will need to come up with a systematic way to break down the networks so that each section can run on different boards simultaneously. The communication between each FPGA board is also needed to be minimized in order to achieve maximum speed up.

Instead of increasing the size of the hidden layer and visible layer to make

the overall network bigger, we could also scale the network work by stacking multiple asymmetric RBM together to increase its learning ability. If we are able to train multiple stacked RBM together on a single FPGA, we are able to train data with more complicated statistic properties

6.2.2 Extension to Other Applications

Right now the application is only for handwriting recognition, since the input image size it can handle is small and it only has one layer of RBM. If our implementation can be of a larger input image size, we might be able to train the network for other application such as face detection or picture classification.

Our current implementation has a fast processing time, thus it is also possible to make it into a real-time handwriting recognition tool. A camera module can be easily added to the FPGA. If we can implement a real-time image processing unit that can parse the digits out into 28×28 pixel images, and feed those images into the RBM network, then our RBM will be able to recognize handwriting digits in real time.

REFERENCES

[1] R. Hechit-Nielsen, *Neurocomputing*. Addison-Wesley Publishing Company, 1990.

[2] A. K. Jain and J. Mao, "Artificial nerual networks: A tutorial," in *IEEE Computer Society*. IEEE, 1996, pp. 31–44.

[3] I. Basheer and M. Hajmeer, "Artificial neural networks: Fundamentals, computing, design, and application," *Journal of Microbiological Methods*, vol. 43, no. 1, pp. 3–31, 2000.

[4] R. Lippmann, "An introduction to computing with neural nets," *ASSP Magazine*, vol. 4, no. 2, pp. 4–22, 1987.

[5] P. Simpson, *Artificial Neural Systems: Foundations, Paradigms, Applications, and Implementations*. Pergamon Press, 1990.

[6] A. Maren, "A logical topology of neural networks," in *Second Workshop on Neural Networks*. WNN-AIND, 1991, p. 91.

[7] A. Garth, D. Rollins, J. Zhu, and V. Chen, "Evalutation of model discrimination techniques in artificial neural networks with application to grain drying," *Artificial Neural Networks in Engineering*, vol. 6, no. 1, pp. 939–950, 1996.

[8] J. Chun, E. Atalana, S.-B. Kima, H.-J. Kima, M. E. Hamida, M. E. Trujilloa, J. G. Mageeb, G. P. Manfioa, A. C. Warda, and M. Goodfellow, "Rapid identification of streptomycetes by artificial neural network analysis of pyrolysis mass spectra," *FEMS Microbiology Letters*, vol. 114, no. 1, pp. 115–119, 1993.

[9] J. Chun, A. C. Warda, and M. Goodfellow, "Artificial neural network analysis of pyrolysis mass spectrometric data in the identification of streptomyces strains," *FEMS Microbiology Letters*, vol. 107, no. 2–3, pp. 321–325, 1993.

[10] S.-B. Cho, "Neural-network classifiers for recognizing totally unconstrained handwritten numerals," *Nerual Networks*, vol. 8, no. 1, pp. 43–53, 2002.

[11] G. Hinton, D. Li, Y. Dong, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[12] J. J. Hopfield and D. W. Tank, "Computing with nerual circuits: A model," *Science*, vol. 233, no. 4764, pp. 625–633, 1986.

[13] P. D.T and P. D.T.N, "Artificial intelligence in engineering," *International Journal of Machine Tools and Manufacture*, vol. 39, no. 6, pp. 937–949, 1999.

[14] L. Fu, *Neural Networks in Computer Intelligence*. Mcgraw-Hill, 1995.

[15] C. S. Lindsey and T. Lindblad, "Review of hardware neural networks: A user's perspective," in *Proceeding of Third Workshop on Neural Networks*. IEEE, 1994, pp. 195–202.

[16] H. Martin, *Cellular Neural Networks: Analysis, Design, and Optimization*. Springer, 2000.

[17] L. Zhang, Y. Han, and X. Li, "Fault tolerance mechanism in chip many-core processors," *Tsinghua Science & Technology*, vol. 12, no. 1, pp. 169–174, 2007.

[18] V. K. Pallipuram, M. Bhuiyan, and M. C. Smith, "A comparative study of GPU programming models and architectures using neural networks," *The Journal of Supercomputing*, vol. 61, no. 3, pp. 673–718, 2011.

[19] K.-S. Oh, "GPU implementation of neural networks," *Pattern Recognition*, vol. 37, no. 4, pp. 1311–1314, 2004.

[20] J. Liu and G. Lei, "Implementation of neural network backpropagation in CUDA," in *Intelligence Computation and Evolutionary Computation*. Springer, 2013, pp. 1021–1027.

[21] X. Sierra-Canto, F. Madera-Ramirez, and V. Uc-Cetina, "Parallel training of a back-propagation neural network using CUDA," in *Machine Learning and Applications*. IEEE, 2010, pp. 307–312.

[22] Y. Zhang and S. Zhang, "Optimized deep learning architectures with fast matrix operation kernels on parallel platform," in *Tools with Artificial Intelligence*. IEEE, 2013, pp. 71–78.

[23] J. Gu, M. Zhu, Z. Zhou, F. Zhang, Z. Lin, Q. Zhang, and M. Breternitz, "Implementation and evaluation of deep neural networks (DNN) on mainstream heterogeneous systems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*. ACM, 2014, pp. 1–7.

[24] B. Essen, M. Macaraeg, C. Gokhale, and R. Prenger, "Accelerating a random forest classifier: multi-core, GP-GPU, or FPGA," in *International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2012, pp. 232–239.

[25] M. Glesner, *Neurocomputers: An Overview of Neural Networks in VLSI*. Chapman & Hal, 1994.

[26] D. Chen, *Design Automation for Microelectronics*. Springer, 2009.

[27] C. Cox and E. Blanz, "GangLion: A fast field programmable gate array implementation of a connectionist classifier," *Journal of Solid-State Circuits*, vol. 28, no. 3, pp. 288–299, 1992.

[28] R. G. Girones, R. C. Palero, J. C. Boluda, and A. S. Cortes, "FPGA implementation of a pipelined on-line backpropagation," *Journal of VLSI Signal Processing*, vol. 40, no. 1, pp. 189–213, 2005.

[29] V. Nambiar, M. Khalil-Hani, R. Sahnoun, and M. Marsono, "Hardware implementation of evolvable block-based neural networks utilizing a cost efficient sigmoid-like activation function," *Neurocomputing*, vol. 140, no. 1, pp. 228–241, 2014.

[30] S.-T. Pan and M.-L. Lan, "An efficient hybrid learning algorithm for neural network-based speech recognition systems on FPGA chip," in *Neural Computing & Application*. Springer, 2012, pp. 1879–1885.

[31] N. Yildiz, K. Cesur, E. Kayaer, V. Tavsanoglu, and M. Alpay, "Architecture of a fully pipelined real-time cellular neural network emulator," in *IEEE Transactions on Circuits and Systems*. IEEE, 2014, pp. 130–138.

[32] E. Ordonez-Cardenas and R. Romero-Troncoso, "MLP neural network and on-line backpropagation learning implementation in a low-cost FPGA," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. ACM, 2008, pp. 333–338.

[33] E. Won, "A hardware implementation of artificial neural networks using field programmable gate arrays," *Nuclear Instruments & Methods in Physics Research*, vol. 581, no. A, pp. 816–820, 2007.

[34] Z. Lin, Y. Dong, Y. Li, and T. Watansbe, "A hybrid architecture for efficient FPGA-based implementation of multilayer neural network," in *Circuits and Systems*. IEEE, 2010, pp. 616–619.

[35] S. Himavathi, D. Anitha, and D. Muthuramalingam, "Feed forward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 880–888, 2007.

[36] M. Krips, T. Lammert, and A. Kummert, "FPGA implementation of a neural network for a real-time hand tracking system," in *First IEEE International Workshop on Electronic Design Test and Applications*. IEEE, 2002, pp. 313–317.

[37] K. L. Rice, T. M. Taha, and C. N. Vutsinas, "Scaling analysis of a neocortex inspired cognitive model on the Cray XD1," *Journal of Supercomputing*, vol. 47, no. 1, pp. 21–43, 2009.

[38] D. George and J. Hawkins, "A hierarchical Bayesian model of invariant pattern recognition in the visual cortex," in *IEEE International Joint Conference on Neural Networks*. IEEE, 2005, pp. 1812–1817.

[39] J. Zhu and P. Sutton, "FPGA implementation of neural networks: A survey of a decade of progress," in *Field Programmable Logic and Application*. Springer, 2003, pp. 1062–1066.

[40] K. R. Nichols, M. A. Moussa, and S. M. Areibi, "Feasibility of floating-point arithmetic in FPGA based artificial neural networks," in *In CAINE*, 2002, pp. 8–13.

[41] M. Marchesi, "Fast neural networks without multipliers," *Transactions on Neural Networks*, vol. 4, no. 1, pp. 53–62, 1993.

[42] J. Holt and T. Baker, "Back propagation simulation using limited precision calculations," in *International Joint Conference on Neural Networks*. IEEE, 1991, pp. 121–126.

[43] S. Johnston, G. Prasad, and L. Maguire, "Comparative investigation into classical and spiking neuron implementations on FPGAs," in *ICANN*. Springer, 2005, pp. 269–274.

[44] J. Misra and I. Saha, "Artificial neural networks in hardware: A survey of two decades of progress," in *Neurocomputing*. Elsevier B.V., 2010, pp. 239–255.

[45] H. Graf, L. Jackel, R. Howard, B. Straughn, J. Denker, W. Hubbard, D. Tennant, and D. Schwartz, "VLSI implementation of a neural network memory with several hundreds of neurons," in *Neural Networks for Computing*. American Institute of Physics, 1987, pp. 182–187.

[46] A. Agranat and C. Neugebauer, "A CCD based neural network integrated circuit with 64K analog programmable synapses," in *International Joint Conference on Neural Networks*. IEEE, 1990, pp. 551–555.

[47] T. Morishita, Y. Tamura, T. Otsuki, and G. Kano, "A BiCMOS analog nerual network with dynamically updated weights," *IEICE Transactions on Electronics*, vol. 75, no. 3, pp. 297–302, 1992.

[48] M. Holler, S. Tam, and H. Castro, "An electrically trainable artificial neural network (ETANN) with 10240 "floating gate" synapses," in *Computer Society Neural Networks Technology series*. IEEE, 1990, pp. 50–55.

[49] B. Wilamowski, J. Binfet, and M. Kaynak, "VLSI implementation of neural networks," *International Journal of Neural Systems*, vol. 10, no. 3, pp. 191–197, 2000.

[50] J. Zurada, "Analog implementation of neural networks," *IEEE Tarnsactions on Nerual Networks*, vol. 7, no. 2, pp. 36–41, 1996.

[51] M. Verleysen and L. Luc Voz, "An analog processor architecture for a neural network classifier," *IEEE Micro*, vol. 14, no. 1, pp. 16–28, 1994.

[52] C. Mead, *VLSI and Neural System*. Addison-Wesley, 1989.

[53] J. Ortiz and C. Ocasio, "Analog hardware model for morphological neural networks," in *International Conference on Neural Networks and Computational Intelligence*. ACTA Press, 2003, pp. 40–44.

[54] M. Milev and M. Hristov, "Analog implementation of ANN with inherent quadratic nonlinearity of the synapses," *Transaction on neural networks*, vol. 14, no. 5, pp. 1187–1200, 2003.

[55] B. Brown, X. Yu, and S. Grverick, "Mixed-mode analog VLSI continuous-time recurrent neural network," in *International Conference on Circuits, Signals, and Systems*. ACTA Press, 2004, pp. 104–108.

[56] P. Masa, K. Hoen, and H. Wallinga, "A high-speed analog neural processor," *Micro, IEEE*, vol. 14, no. 3, pp. 40–50, 2002.

[57] A. Schmid, Y. Leblebici, and D. Mlynek, "A mixed analog digital artificial neural network with on chip learning," in *Circuits, Devices and Systems*. IEEE, 1991, p. 345.

[58] C. Lindsey and T. Lindblad, "Survey of neural network hardware," in *Applications and Science of Artificial Neural Networks*. IEEE, 1995, pp. 1194–1205.

[59] Y. Liao, "Neural networks in hardware: A survey," Santa Cruz, Technical Report, 2001.

[60] P. Ferreira, R. P., A. Antunes, and F. M. Dias, "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function," *Neruocomputing*, vol. 71, no. 1, pp. 71–77, 2007.

[61] D. Shen, L. Jin, and X. Ma, "FPGA implementation of feature extraction and neural network classifier for handwritten digit recognition," *Advance in Neural Networks*, vol. 3173, no. 1, pp. 988–995, 2004.

[62] P. Smolensky, "Information processing in dynamical system: Foundations of harmony theory," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, no. 1, pp. 194–281, 1986.

[63] G. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets," in *Neural Computation*. MIT, 2006, pp. 1527–1554.

[64] G. Hinton and R. Salakhudinov, "Reducing the dimensionality of data with neural networks," in *Science*. AAAS, 2006, pp. 504–507.

[65] H. Larochelle and Y. Bengio, "Classifcation using discriminative restricted Boltzmann machines," in *Inernational Conference on Machine Learning*. ACML, 2008, p. 536.

[66] L. Zadeh, "Fuzzy logic neural networks and soft computing," *Communications of the ACM*, vol. 37, no. 3, pp. 77–84, 1994.

[67] S. K. Kim, L. C. McAfee, P. L. McMahon, and K. Olukotun, "A highly scalable restricted Boltzmann machine FPGA implementation," in *Field Programmable Logic and Applications*. IEEE, 2009, pp. 367–372.

[68] D. L. Ly and P. Chow, "High-performance reconfigurable hardware architecture for restricted Boltzmann machines," *Transaction on neural networks*, vol. 21, no. 11, pp. 1780–1792, 2010.

[69] D. L. Ly and P. Chow, "A high-performance reconfigurable hardware architecture for restricted Boltzmann machines," in *FPGA*. ACM, 2009, pp. 73–82.

[70] D. L. Ly and P. Chow, "A multi-FPGA architecture of stochastic restricted Boltzmann machines," in *Field Programmable Logic and Applications*. IEEE, 2009, pp. 168–173.

[71] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," 1986.

[72] Y. Freund and D. Haussler, *Unsupervised learning of distributions of binary vectors using two layer networks*. Computer Research Laboratory, 1994.

[73] D. Geman and S. Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," *IEEE Transcation onf Pattern Analysis and Machine Intelligence*, vol. 6, no. 6, pp. 721–741, 1984.

[74] D. Ackley, G. Hinton, and T. Sejnowski, "A learning algorithm for Boltzmann machines," *Cognitive Scinece*, vol. 9, no. 1, pp. 147–169, 1985.

[75] G. Hinton and T. Sejnowski, "Learning and relearning in Boltzmann machines," *Parallel Distributed Processing*, vol. 1, no. 1, pp. 282–371, 1986.

[76] H. Amin, K. Curtis, and B. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *Circuits Devices System*, vol. 144, no. 6, pp. 313–317, 1997.

[77] T. Tkacik, "A hardware random number generator," in *Cryptographic Hardware and Embedded Systems*. IEEE, 2003, pp. 450–453.

[78] P. L'Ecuyer, "Maximally equidistributed combined Tausworthe generators," *Math. Comput*, vol. 65, no. 213, pp. 203–213, 1996.