# A High-Performance, Reconfigurable Architecture for Restricted Boltzmann Machines

by

Daniel Le Ly

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

# Abstract

**A High-Performance, Reconfigurable Architecture for Restricted Boltzmann Machines**

Daniel Le Ly
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

Despite the popularity and success of neural networks in research, the number of resulting commercial or industrial applications have been limited. A primary cause of this lack of adoption is due to the fact that neural networks are usually implemented as software running on general-purpose processors. Hence, a hardware implementation that can take advantage of the inherent parallelism in neural networks is desired.

This thesis investigates how the Restricted Boltzmann machine, a popular type of neural network, can be effectively mapped to a high-performance hardware architecture on FPGA platforms. The proposed, modular framework is designed to reduce the time complexity of the computations through heavily customized hardware engines. The framework is tested on a platform of four Xilinx Virtex II-Pro XC2VP70 FPGAs running at 100MHz through a variety of different configurations. The maximum performance was obtained by instantiating a Restricted Boltzmann Machine of $256 \times 256$ nodes distributed across four FPGAs, which results in a computational speed of 3.13 billion connection-updates-per-second and a speed-up of 145-fold over an optimized C program running on a 2.8GHz Intel processor.

# Acknowledgements

First and foremost, I would like to thank Professor Paul Chow for his invaluable guidance and advice throughout the years. Thank you for supporting this project and making it as fun and exciting as it was stimulating and enlightening. I'm grateful for the opportunity to learn so much from you – not only from your extensive experience on technical matters, but on matters of professionalism and conduct as well.

Furthermore, I would like to thank Arun Patel and Manuel Saldaña for their support and feedback; thanks for putting up with my countless questions and being there when things inevitably went wrong. I would also like to thank everyone in the Chow group over the years: Jeff Goeders, Danny Gupta, Alireza Heidarbarghi, Andrew House, Professor Jiang Jiang, Alex Kaganov, Geng Liu, Chris Madill, Vince Mirian, Nihad Nasim, Daniel Nunes, Chu Pang, Emanuel Ramalho, Keith Redmond, Kam Pui Tang, Danyao Wang, David Woods, Mike Yan, and Matt Zeiler. It was great working with all of you and thank you for making this experience so enjoyable. Next, I would like to thank Professor Geoffrey E. Hinton and Graham Taylor of the Machine Learning group for their comments and conversations.

I would like to acknowledge CMC Microsystems, NSERC and Xilinx for their funding and support of this project. Without their support, we wouldn't have the resources and technology that made this research possible.

And, of course, I would like to thank my family for their love and support – none of this would be possible without you. Thank you for being there to make the bad times bearable, and the good times so memorable.

# Contents

# List of Tables

# List of Figures

# List of Acronyms

| | |
|---|---|
| **AGS** | Alternating Gibbs Sampling |
| **ASIC** | Application Specific Integrated Circuit |
| **BRAM** | Block Random Access Memory |
| **BEE2** | Berkeley Emulation Engine 2 |
| **CUPS** | Connection Updates Per Second |
| **DBN** | Deep Belief Nets |
| **DSP** | Digital Signal Processing |
| **FIFO** | First-In-First-Out |
| **FF** | Flip-Flops |
| **FPGA** | Field-Programmable Gate Array |
| **GPU** | Graphics Processing Unit |
| **I/O** | Input/Output |
| **LI** | Linear Interpolator |
| **LUT** | Look-Up Table |
| **MPI** | Message Passing Interface |
| **PLI**$^k$ | Piecewise Linear Interpolator of the $k$th order |
| **RBM** | Restricted Boltzmann Machine |
| **XST** | Xilinx Synthesis Tool |

# Chapter 1

# Introduction

## 1.1 Motivation

Neural networks have captured the interest of researchers for decades due to their superior ability over traditional approaches for solving machine learning problems. They are able to extract complex, underlying structure from the statistical distribution of data by using networks of simple, parallel processing elements. Of the many neural network varieties, the Restricted Boltzmann Machine (RBM) is a popular architecture that is capable of unsupervised learning and data generation through stochastic processes. RBMs form the building blocks for the popular Deep Belief Nets (DBN), which have been successfully applied to a wide variety of research areas including recognizing hand-written digits [1], reducing the dimensionality of data [2] and generating motion capture data [3].

However, there are significant difficulties in adapting current applications to commercial or industrial settings because software implementations on general purpose processors lack the required performance and scalability. Sequential processors iterate through every connection in the network, which increases complexity quadratically with respect to the number of processing elements. Individual RBMs can scale up to sizes of $2000 \times 500$ nodes [1], taking weeks to train on a desktop computer. As a result, software programs of large RBMs are unable to satisfy the real-time constraints required to solve real-world problems. Furthermore, every processing element only utilizes a small fraction of the processor's resources, exacerbating the performance bottleneck and limiting its cost-effectiveness.

To address these issues, a hardware RBM framework is designed for Field Programmable Gate Arrays (FPGAs). By taking advantage of the inherent parallelism in neural networks, a high-performance system capable of applications beyond academic settings can be realized.

There have been many attempts to create hardware implementations to speed up the performance of neural networks [4,5]. Although many approaches, from analog to VLSI systems, have been attempted, they have not resulted in widely used hardware. These systems are typically plagued with a variety of problems including lack of resolution, limited network size, and a difficult to use or non-existent software interface [6].

In addition to difficulties with the hardware platform, another common problem is the choice of neural network architectures – most architectures are not particularly well suited for hardware systems. The most common type of neural network is the multilayer perceptron with back-propagation architecture [7,8]. Although this architecture is popular and has many applications, the processing elements require real number arithmetic as well as resource intensive components such as multipliers and accumulators, and complex transfer functions. As a result, each processing element requires significant resources, which restricts the scalability of implementation. The common solution is to achieve parallelism by creating a customized pipeline similar to the super-scalar design used by processors. The pipeline is then replicated as necessary – unfortunately, this approach does not result in enough parallelism and speed-up to justify the cost and effort of using such systems.

In comparison, RBMs are well suited for hardware implementations. First, RBMs can use data types that map well to hardware. The node states are binary-valued – the properties of binary arithmetic ensures that operations, such as multiplication, can be done with simple gates instead of resource intensive multipliers. Next, RBMs do not require high precision. Thus, fixed-point arithmetic units can be used to reduce resource utilization and increase processing speed. The simplicity in the neural network architecture allows for clever hardware design, providing scalability and parallelism.

In particular, FPGAs have many advantages over other hardware platforms for RBM implementations. FPGAs are growing rapidly, allowing entire systems to be implemented on a single chip. In addition to the raw fabric, FPGAs have numerous hard components, including on-board memory, Digital Signal Processing (DSP) units, Input/Output (I/O) transceivers and even processors, which aid in designing full-scale systems.

However, the most important aspect of FPGAs is reconfigurability. Since the topology of the network defines its application – the arrangement of processing elements will dictate the capabilities and behaviour of the network. Application Specific Integrated Circuit (ASIC) implementations must balance the trade-off between performance and versatility, and since a general solution is highly unoptimized, this trade-off is a serious concern. Being able to design for a reconfigurable system allows hardware to be generated that suits the exact required topology, thus, optimizing performance without sacrificing versatility.

## 1.2 Contributions

This thesis proposes a reconfigurable architecture with modular components for implementing high-performance RBMs in hardware. The primary contributions of this work are as follows:

- Complexity and performance analysis of Restricted Boltzmann Machines
- A method to partition large RBMs into smaller, equivalent networks
- A series of modular computational engines capable of implementing a wide variety of RBM topologies
- A high-performance microarchitecture that provides a versatile interface to the computational engines
- A distributed memory data structure that allows for instantaneous matrix transposes

- Optimized computational engines that accelerate RBM computation through parallelism and pipelining
- A piecewise linear interpolator for Look-Up Table function implementations that increase resolution while maintaining a high throughput
- A method of designing multi-FPGA implementations of the RBM architecture with a significant increase in performance
- A method of virtualizing the RBM architecture to implement large networks with limited hardware
- Performance comparisons between the hardware and software implementations, characterizing the performance of various arrangements of the reconfigurable architecture

## 1.3 Overview

The remainder of the thesis is organized as follows: Chapter 2 provides background and analysis regarding RBMs, as well as discussion of other implementations on accelerated platforms. Chapter 3 describes a novel method of partitioning large RBMs into smaller, equivalent networks. Chapter 4 outlines the hardware architecture, including descriptions of the cores and implementation examples. The evaluation methodology is presented in Chapter 5 and the results are discussed in Chapter 6. The thesis is summarized and concluded in Chapter 7, followed by a discussion of future work.

# Chapter 2

# Background

## 2.1 Restricted Boltzmann Machine Theory

This section briefly the terminology, mathematical background and procedures involved in the mechanics of RBMs. Additional details, including the historical development and statistical motivation, can be found in [9, 10].

### 2.1.1 Terminology

Neural networks form a computational paradigm which is used to model non-linear, statistical data. Inspired by their biological counterparts, artificial neural networks consist of a distribution of simple processing elements arranged in a networked structure to exhibit emergent behaviours. Different neural network architectures define the type of processing elements and how they are interconnected.

A RBM is a generative, stochastic neural network architecture. It is used to model the statistical behaviour of a particular set of data – given a series of input vectors with some shared, underlying properties, the network will build an internal model of the statistical distribution of that data. This internal model can be used to recognize whether an arbitrary datum point belongs to the same group as the original series of input vectors. Furthermore, the internal model allows the network to produce new data which is consistent with the original distribution; this property is called *generative*.

The RBM is *stochastic* because it uses a probabilistic approach to modelling data. To capture statistical properties, the RBM determines the probability distribution of a given data set through the aid of random processes. These two properties, generative and stochastic, makes RBMs a unique neural network architecture.

Like all neural networks, RBMs are interesting because of their capability to learn. RBMs use an internal model which is described mathematically by a multitude of independent parameters. Learning consists of tuning the values of these parameters so that the RBM can model the data in an appropriate fashion. Due to the combinatorial explosion of the parameter space, finding a correct set of parameters is a non-trivial task. To learn the optimal parameters, the RBM processes a set of data vectors, called the *training data*, and *learning rules* are applied iteratively. The RBM repeatedly processes the training data until it is able to reproduce the desired effect. At this stage, the RBM is sufficiently *trained*. For verification, a new set of previously unexposed data vectors, called the *test data*, can be used to confirm its behaviour.

The RBM was conceived as an amalgamation of ideas from neural network and statistical mechanics – as a result, the terminology is derived from both these otherwise distinct fields. In neural networks, processing elements are often referred to as *nodes* or *neurons*. The nodes in a RBM have binary *states*: they can either be on or off. A RBM consists of two layers of nodes, the *visible* and *hidden* layers. The visible layer is used to store the data while the hidden layer acts as an internal representation of the data. There are *connections* between every node in opposite layers, and no connections between any nodes in the same layer. Each of these connections have an associated *weight*, which provides the learning parameters for the RBM.

The following notation system will be used: $v_i$ and $h_j$ are the binary states of the $i$th and $j$th node, where $i = \{0, \ldots, I-1\}$ and $j = \{0, \ldots, J-1\}$, in the visible and hidden layer, respectively; $w_{i,j}$ is the connection weight between the $i$th and $j$th node. The terminology and notation is summarized in a schematic representation in Fig. 2.1.

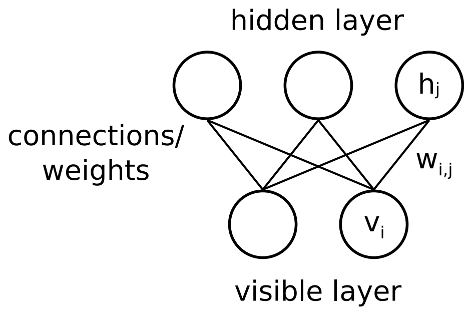

Figure 2.1: A schematic diagram of a RBM with labelled components.

### 2.1.2 Alternating Gibbs Sampling

Alternating Gibbs Sampling (AGS) is the operating process for the RBM. It is the basis for generating node states as well as the learning rules [11]. AGS is divided into two phases, the *generate* and *reconstruct* phases. During the generate phase, the visible layer is *clamped*, which is to set the node states so that they cannot be altered. The clamped visible states are then used to determine the node states of the hidden layer. In the reconstruction phase, the opposite occurs where the hidden layer is clamped to generate the node states of the visible layer. To begin the process, an initial vector from the training data is placed in the visible layer and the phases are utilized in an alternating manner starting with the generate phase. The phases are numbered in counting succession, starting with one for the first generate phase. To differentiate nodes between phases, the node states will be indexed with the phase number as a superscript. A schematic representation of this process is summarized in Fig. 2.2.

To understand how the node states are determined, the concept of *global energy* must first be introduced. Any relation to a physical manifestation of energy is lost during the translation from statistical mechanics: it is best to think of the energy as simply a numeric value that defines the operation and behaviour of a RBM. The goal of the RBM is to tune the weight parameters and node states so that the global energy is minimized for a given data set. The global energy, $E$, is defined in Eq. 2.1.

$$E = -\sum_{i=0}^{I-1}\sum_{j=0}^{J-1} w_{i,j} v_i h_j \tag{2.1}$$

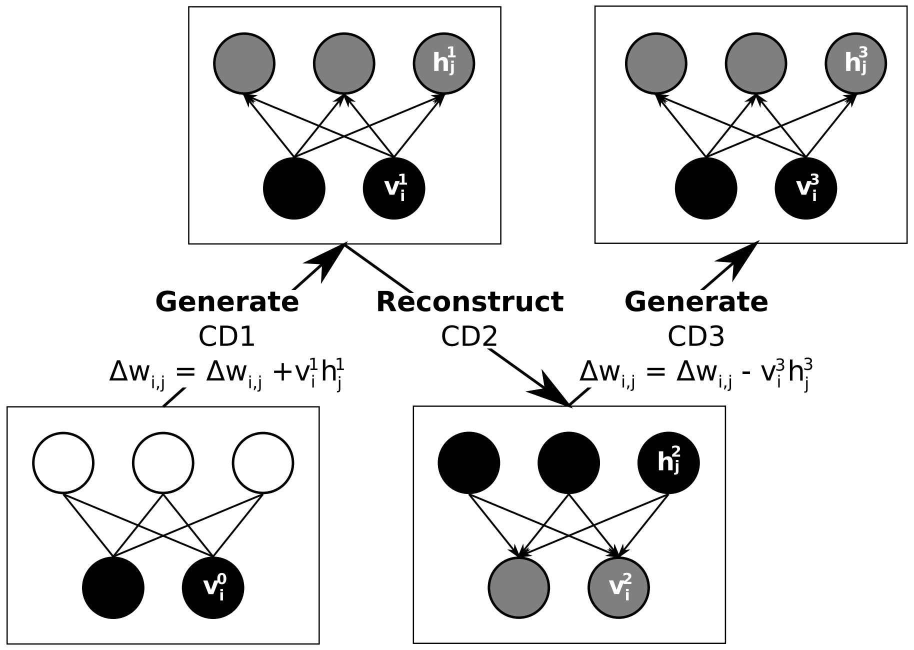

Figure 2.2: A schematic diagram of the Alternating Gibbs Sampling for three phases. Uninitialized nodes are white, clamped nodes are black, and computed nodes are grey.

Because the connections only exist between nodes of opposite layers, the energy can be redefined as a sum of *partial energies*, depending on which AGS phase is being computing. Rather than computing and minimizing the global energy in every phase, the same goal can be achieved by maximizing the local energies since the node states in a single layer is clamped. The generate and reconstruct phase use Eqs. 2.2 and 2.3, respectively.

$$E = -\sum_{i=0}^{I-1} v_i \left( \sum_{j=0}^{J-1} w_{i,j} h_j \right) = -\sum_{i=0}^{I-1} v_i E_i \tag{2.2}$$

$$= -\sum_{j=0}^{J-1} h_j \left( \sum_{i=0}^{I-1} w_{i,j} v_i \right) = -\sum_{j=0}^{J-1} h_j E_j \tag{2.3}$$

The formulation of the partial energies indicates that the global energy can be determined using just the clamped node states and its respective partial energy. Since the partial energies are independent of the related node states, they can be calculated simultaneously allowing for a parallel computation of global energy.

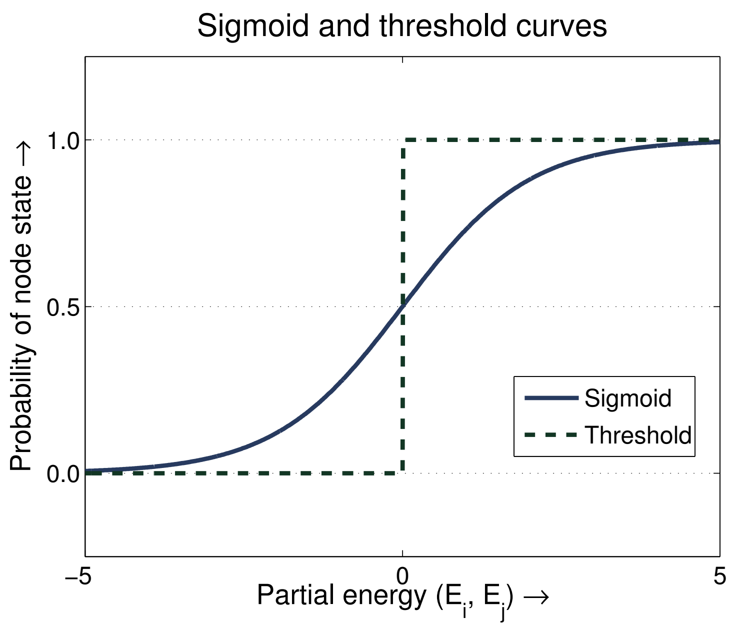

Figure 2.3: A plot of a sigmoid and threshold function.

Using the statistical mechanics approach of defining probabilities with respect to energy functions, the node states have a cumulative distribution function of a logistic or sigmoid function. The probability of a node state turning on for a visible and hidden node is expressed in Eqs. 2.4 and 2.5, respectively. To determine the node state, a uniformly random variable must be sampled against the cumulative distribution function.

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

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

Occasionally, the probabilistic approach might be undesirable due to various reasons including unpredictability as well as computational complexity. Instead, a deterministic, first-order approximation threshold function can be used (Eqs. 2.6 and 2.7). By using the threshold function, the node states can be determined directly from the energies without any random processes. Comparisons of the sigmoid distribution function and the deterministic threshold function is in Fig. 2.3.

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

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

### 2.1.3 Learning

One of the primary reasons for developing neural networks is their capability for machine learning, and as a result the learning rules for RBMs are of great interest [12]. In review, the weight values are parameters that dictate the energies and subsequently the node states. To model a given data set, the weights are modified so the RBM generates the minimum energy across the entire training set.

Since each weight is an individual learning parameter, the global energy is differentiated with respect to the individual weights to find the minimum (Eq. 2.8). In this notation, the $\langle \ldots \rangle^x$ represents the expected values of the entire data set for the $x$th AGS phase, where $x = \{0, \ldots, X-1\}$, and $\epsilon$ is the *learning rate* of the network, a parameter that sets the rate at which the RBM converges on the proper solution.

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

The node states are generated through the iterative process of AGS, and as a result, the energy derivative indicates the direction vector of steepest descent in the weight space to reach a minima. As a result, the weights must be iteratively modified according to this derivative at the end of every training iteration.

This formulation raises several important points. First, to properly descend the gradient, the expected values of the node interactions are required over the entire data set; this is called *batch learning*. However, large batches require a significant amount of computation time for a single weight update. As a result, the batch can be divided into smaller groups, called *mini-batches*, which allows the weight updates to occur with more frequently due to smaller groups of training vectors. If the mini-batch only includes a single data vector, this is referred to as *on-line learning*. This raises a trade-off between the frequency and accuracy of the weight updates – in practice, mini-batches of several hundred training vectors is typical.

Next, the formal definition of the gradient descent requires the node states from the infinite AGS phases. Because this is impractical, researchers have found that the infinite AGS phase can be accurately predicted with a small, finite AGS phase. This process is called *contrastive-divergence learning*, and is labelled CD$x$, where $x$ is the $x$th AGS phase. RBMs have been successfully trained with the lowest possible unique AGS phase, CD3 (Fig. 2.2).

Finally, the learning rate is an independent parameter which describes the factor of weight updates and thus the magnitude of the gradient descent vector. Unfortunately, there is a trade-off in picking learning rates – small learning rates ensure convergence, while large learning rates decrease learning times. Although there are only heuristics to suggest a good learning rate, there are some algorithms that suggest modifying the learning rate in between batches to achieve a fast but convergent solution in a process called *simulated annealing* [11, 13].

Although these learning algorithm shortcuts deviate from the strict definition of gradient descent, they enhance operating speed and are widely popular. The learning rules are updated in Eqs. 2.9-2.11, where $k$ is the discrete iteration count and the $l$th batch belongs in the range of $l = \{0, \ldots, L-1\}$ vectors:

$$w_{i,j}[k+1] = w_{i,j}[k] + \Delta w_{i,j}[k] \tag{2.9}$$

$$\Delta w_{i,j}[k] = \epsilon \left( \langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^{X-1} \right) \tag{2.10}$$

$$\langle v_i h_j \rangle^x = \frac{1}{L} \sum_{l=0}^{L-1} v_i^x h_j^x \tag{2.11}$$

### 2.1.4 Matrix Notation

For ease of understanding and computation, Eqs. 2.1-2.11 can be reformulated succinctly using matrix expressions that encapsulate the concept of layers and batches instead of operations on individual nodes. The basic notation must be redefined. For a RBM of $I$ visible nodes and $J$ hidden nodes, the visible and hidden layers will be represented respectively as:

$$\mathbf{v}_l^x = [v_0^x \dots v_{I-1}^x] \in \mathbb{B}^{1\times I} \tag{2.12}$$

$$\mathbf{h}_l^x = [h_0^x \dots h_{J-1}^x] \in \mathbb{B}^{1\times J} \tag{2.13}$$

Thus, the visible and hidden layers for an entire batch can be written as:

$$\mathbf{V}^x = \begin{bmatrix} \mathbf{v}_0^x \\ \vdots \\ \mathbf{v}_{L-1}^x \end{bmatrix} \in \mathbb{B}^{L\times I}, \qquad \mathbf{H}^x = \begin{bmatrix} \mathbf{h}_0^x \\ \vdots \\ \mathbf{h}_{L-1}^x \end{bmatrix} \in \mathbb{B}^{L\times J} \tag{2.14}$$

The weights, as a function of discrete count $k$, can also be formulated as:

$$\mathbf{W}[k] = \begin{bmatrix} w_{0,0}[k] & \cdots & w_{0,J-1}[k] \\ \vdots & \ddots & \vdots \\ w_{I-1,0}[k] & \cdots & w_{I-1,J-1}[k] \end{bmatrix} \in \mathbb{R}^{I\times J} \tag{2.15}$$

Then, the Eqs. 2.1-2.11 can be reformulated as:

$$\mathbf{V}^x = \begin{cases} \mathbf{V}^0 & , \quad x = 0 \\ f(\mathbf{E}_{\mathbf{v}}^{x-1}) & , \quad x \text{ is even} \\ \mathbf{V}^{x-1} & , \quad x \text{ is odd} \end{cases} \tag{2.16}$$

$$\mathbf{H}^x = \begin{cases} f(\mathbf{E}_{\mathbf{h}}^{x-1}) & , \quad x \text{ is odd} \\ \mathbf{H}^{x-1} & , \quad x \text{ is even} \end{cases} \tag{2.17}$$

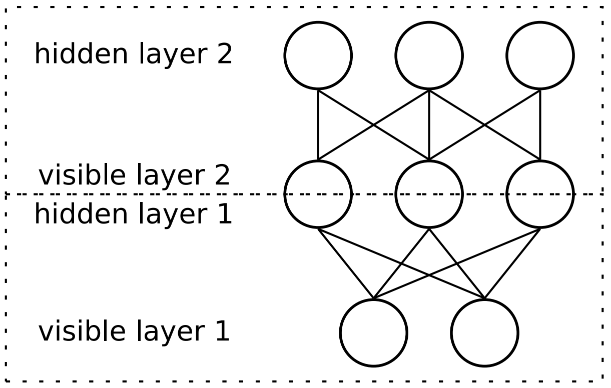

Figure 2.4: A schematic diagram of a dual-layered Restricted Boltzmann Machine.

$$\mathbf{E}_{\mathbf{v}}^{x} = (\mathbf{H}^{x})\mathbf{W}^{\mathrm{T}}[k], \qquad \in \mathbb{R}^{L\times I} \tag{2.18}$$

$$\mathbf{E}_{\mathbf{h}}^{x} = (\mathbf{V}^{x})\mathbf{W}[k], \qquad \in \mathbb{R}^{L\times J} \tag{2.19}$$

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \frac{\epsilon}{L}\left((\mathbf{V}^{1})^{\mathrm{T}}\mathbf{H}^{1} + (\mathbf{V}^{X-1})^{\mathrm{T}}(\mathbf{H}^{X-1})\right) \tag{2.20}$$

Where $f(\cdot)$ is the transfer function applied element-wise to the matrix – it can either the sigmoid function random variable test (Eqs. 2.4-2.5) or the threshold function (Eqs. 2.6-2.7).

## 2.1.5 Deep Belief Nets and Layered Restricted Boltzmann Machines

The most common implementation of Restricted Boltzmann Machines involves their application in Deep Belief Nets (DBN) [1]. DBNs are Bayesian networks that model random variables and their conditional probabilities. Directed DBNs are composed of layers of RBMs, which individually have a single layer of hidden nodes, and as a result, can only capture first order statistics. The underlying structure in a given data set may require higher-order statistics for complete description, and thus, a single layer RBM will be insufficient. However, increasingly complex data sets can be modelled by layering RBMs so that the hidden layer of one is the visible layer of another. This process of stacking RBMs can be done indefinitely; the first RBM models patterns of data, while each subsequent layer models patterns of patterns. This is illustrated schematically in Fig. 2.4.

| Section | Time Complexity | Equation |
|---|---|---|
| Node select | $O(n)$ | 2.16, 2.17 |
| Energy compute | $O(n^2)$ | 2.18, 2.19 |
| Weight update | $O(n^2)$ | 2.20 |

Table 2.1: The time complexity analysis for each section of the RBM algorithm.

The layering of the RBM modifies the global operating and learning algorithms slightly: the individual RBMs operate in the exact same way, but there is a macro-algorithm to organize how the layers operate with respect to each other. However, this section is only meant to introduce the basic concepts of RBMs. The idea of layered RBMs is raised because a hardware architecture must also support this highly popular DBN extension.

## 2.2 Restricted Boltzmann Machine Complexity Analysis

To understand why sequential processors are not well suited for RBM implementations, the algorithm to implement Eqs. 2.16-2.20 must be analyzed. A pseudocode sketch of the algorithm is summarized in Fig. 2.5.

For ease of analysis, the restriction that the RBM must have symmetric layers will be applied ($I = J = n$). By simply tracing the loops in Fig. 2.5, the complexity analysis of the algorithm is straightforward. For a more detailed look, the algorithm is divided into three sections by their computation; node select (Eqs. 2.16-2.17), energy compute (Eqs. 2.18-2.19), and weight update (Eq. 2.20). The complexity of these individual sections is summarized in Table 2.1. The overall time complexity of the RBM algorithm is $O(n^2)$, which illustrates why implementing RBMs in software on sequential processors results in decreased performance as network size increases.

In addition to the time complexity, it is important to understand how memory resources scale with respect to network sizes. Analyzing Eqs. 2.16-2.20, it is clear that only a handful of variables need to be stored: visible and hidden node states, partial energies, learning rate, as well as weight parameters and their corresponding weight updates. Using a single bit to

```
for k in every(time_step) :
    for l in every(batch) :
        visible[] = get_datavector(l)

        for x in every(AGS_phase) :
            if AGS_phase is odd :
                # Energy compute Eq.2.19 - 2 loops -> O(n^2)
                for j in every(hidden_node) :
                    for i in every(visible_node) :
                        energy[j] += visible[i]*weight[i][j]

                # Node select Eq.2.17 - 1 loop -> O(n)
                for j every(hidden_node) :
                    hidden[j] = transfer_function(energy[j])

            else :
                # Energy compute Eq.2.18 - 2 loops -> O(n^2)
                for i in every(visible_node) :
                    for j in every(hidden_node) :
                        energy[i] += hidden[j]*weight[i][j]

                # Node select Eq.2.16 - 1 loop -> O(n)
                for i in every(visible_node) :
                    visible[i] = transfer_function(energy[i])

            # Weight update Eq.2.20 - 2 loops -> O(n^2)
            if (x == 1) :
                for i in every(visible_node) :
                    for j in every(hidden_node) :
                        weight_update[i][j] += visible[i] * hidden[j]
            else if (x == ABS_phase) :
                for i in every(visible_node) :
                    for j in every(hidden_node) :
                        weight_update[i][j] -= visible[i] * hidden[j]

    # Weight update Eq.2.20 - 2 loops -> O(n^2)
    for i in every(visible_node) :
        for j in every(hidden_node) :
            weight[i][j] += learning_rate/batch * weight_update[i][j]
```

Figure 2.5: A pseudocode sketch of the RBM algorithm. All variables are initialized to zero.

store binary values and a word size of $w$-bits to represent real values, the memory resources for each variable are summarized in Table 2.2. It is clear that the four types of variables; node states, energies, learning rate, and weights; require drastically different resources. Despite the varying differences, sequential processors must treat all of the variables in a similar manner and is unable to customize its memory access to the unique requirements of each variable.

| Variable | Symbol | Memory Size[bits] | Memory Complexity | Equation |
|---|---|---|---|---|
| Visible node states | $\mathbf{v}_l^x$ | $n$ | $O(n)$ | 2.16 |
| Hidden node states | $\mathbf{h}_l^x$ | $n$ | $O(n)$ | 2.17 |
| Partial energies | $\mathbf{E}_\mathbf{v}^x$, $\mathbf{E}_\mathbf{h}^x$ | $n \cdot w$ | $O(n)$ | 2.18, 2.19 |
| Learning rate | $\epsilon$ | $w$ | $O(1)$ | 2.20 |
| Weight parameters | $\mathbf{W}[k]$ | $n^2 \cdot w$ | $O(n^2)$ | 2.20 |
| Weight updates | $\Delta\mathbf{W}[k]$ | $n^2 \cdot w$ | $O(n^2)$ | 2.20 |

Table 2.2: The memory utilization analysis for each variable in the RBM algorithm with a word size of $w$-bits.

## 2.3 Related Work

Although there have been many attempts to design hardware implementations of various neural network architectures [4–6], there is a growing interest in hardware-accelerated Restricted Boltzmann Machines due to the popularity of DBN applications. There are two hardware RBM implementations that are of keen interest to this thesis.

The first implementation is by Kim et al. [14], where they designed an RBM architecture for FPGAs. Although their design also targeted FPGAs, they used an alternate architecture that was implemented on an Altera Stratix III EP3SL340 FPGA with a DDR2 SDRAM interface. The Altera Nios II soft-processor is used and operates at 100MHz while the RBM module operates at 200MHz. Their RBM module has an array of weights and neurons that are fed into an array of multipliers and adders to perform matrix multiplication. They are capable of supporting both real-valued and binary valued visible node states.

Their architecture uses 16-bit fixed-point numbers, which is capable of adequately representing the weights. They achieve their performance acceleration by implementing groups of multipliers, adders and embedded RAM. Each group stores a small portion of the weights; the groups are sized to match the 256-bit DDR2 memory bus so that 16 weights are stored in local memory. The computation hardware is then selected on the fly depending 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.

For node selection, a piecewise linear approximation of a non-linear function was used to create a sigmoid function, which only requires a minimal number of addition and shift operations. The random number generator uses a combination of a 43-bit Linear Feedback Shift Register and 37-bit Cellular Automata Shift Register, which provides good statistical properties and a cycle length of $2^{80}$.

They compared their implementation to the MATLAB code provided by Hinton et al [1] using a 2.4GHz 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-ups of 25-fold compared to single precision MATLAB and 30-fold for double precision MATLAB.

The second implementation is by Raina et al. [15], where they designed an RBM architecture for a Graphics Processing Unit (GPU). Their design was programmed using NVIDIA's CUDA language and was tested on an NVIDIA GeForce GTX 280 graphics card with 1GB memory. In addition to the typical graphic processing considerations, such as coalesced memory accesses and use of shared memory, they achieve their performance acceleration by introducing a technique called "overlapping patches", which tiles a 2D representation of the visible nodes. Each overlapping patch is independent of other patches, resulting in very sparse networks. However, deep networks can be built by connecting these sparse overlapping patches. Unlike the work by Kim et al. [14], it is important to note that the large networks achieved by the overlapping patches technique do not have nodes that are fully connected to the opposite layer. This restrictive technique results in a simpler subset of RBM implementations that greatly reduces the effective size of the weight matrix and the amount of necessary communication. Comparing with a Goto BLAS implementation [16] running on a dual-core 3.16GHz CPU, they achieved a maximum speed-up of 72-fold for a network size of $4096 \times 11008$.

# Chapter 3

# Restricted Boltzmann Machine Partitioning

## 3.1 Motivation

A major difficulty that RBM implementations must overcome is the rapid growth of resource utilization required to store the weight parameters and weight updates. As illustrated in Section 2.2, these data structures require $O(n^2)$ memory resources. Since the amount of low-latency memory is limited in most platforms, implementing large RBMs that maintain the performance of smaller networks is extremely difficult with a naive approach of simply scaling the implementation to the size of the large network. To accommodate for the limited resources, a significantly larger off-chip memory is often used, which results in a computational bottleneck due to decreased bandwidth.

Instead, a divide-and-conquer method is proposed that will partition a large RBM into an equivalent collection of smaller but congruent networks. This method of partitioning the RBM into smaller networks allows any implementation to create small networks that do not exhaust the low-latency memory resources. As a result, this novel partitioning approach will provide a method for implementations to better benefit from the data locality of RBMs.

## 3.2 Mathematical Foundation

First, the visible and hidden layers will be partitioned into two disjointed sets. For the visible layer, the original set of $\{0, \dots, I-1\}$ will be divided into the sets $A_0 = \{0, \dots, A-1\}$ and $A_1 = \{A, \dots, I-1\}$. Likewise, the hidden layer will be divided into the sets $B_0 = \{0, \dots, B-1\}$ and $B_1 = \{B, \dots, J-1\}$. As a result, each layer can be represented as a collection of two sets of vectors:

$$\begin{aligned} \mathbf{v} &= [v_0 \dots v_{A-1} \quad v_A \dots v_{I-1}] && , \mathbf{v} \in \mathbb{B}^{1\times I} \\ &= [\mathbf{v}_{A_0} \quad \mathbf{v}_{A_1}] && , \mathbf{v}_{A_0} \in \mathbb{B}^{1\times A}, \mathbf{v}_{A_1} \in \mathbb{B}^{1\times I-A} \end{aligned} \tag{3.1}$$

$$\begin{aligned} \mathbf{h} &= [h_0 \dots h_{B-1} \quad h_A \dots h_{J-1}] && , \mathbf{h} \in \mathbb{B}^{1\times J} \\ &= [\mathbf{h}_{B_0} \quad \mathbf{h}_{B_1}] && , \mathbf{h}_{B_0} \in \mathbb{B}^{1\times B}, \mathbf{h}_{B_1} \in \mathbb{B}^{1\times J-B} \end{aligned} \tag{3.2}$$

Continuing this partitioning to the weight matrix:

$$\begin{aligned} \mathbf{W} &= \left[\begin{array}{ccc|ccc} w_{0,0} & \cdots & w_{0,B-1} & w_{0,B} & \cdots & w_{0,J-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ w_{A-1,0} & \cdots & w_{A-1,B-1} & w_{A-1,B} & \cdots & w_{A-1,J-1} \\ \hline w_{A,0} & \cdots & w_{A,B-1} & w_{A,B} & \cdots & w_{A,J-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ w_{I-1,0} & \cdots & w_{I-1,B-1} & w_{I-1,B} & \cdots & w_{I-1,J-1} \end{array}\right] && , \mathbf{W} \in \mathbb{R}^{I\times J} \\ &= \left[\begin{array}{c|c} \mathbf{W}_{A_0B_0} & \mathbf{W}_{A_0B_1} \\ \hline \mathbf{W}_{A_1B_0} & \mathbf{W}_{A_1B_1} \end{array}\right] && , \mathbf{W}_{A_0B_0} \in \mathbb{R}^{A\times B}, \mathbf{W}_{A_0B_1} \in \mathbb{R}^{A\times J-B} \\ & && , \mathbf{W}_{A_1B_0} \in \mathbb{R}^{I-A\times B}, \mathbf{W}_{A_1B_1} \in \mathbb{R}^{I-A\times J-B} \end{aligned} \tag{3.3}$$

Expanding Eqs. 2.18-2.19 to partitioned form:

$$\mathbf{E_h} = \mathbf{VW}$$

$$\Rightarrow \begin{bmatrix} \mathbf{E_h}_{B_0} \\ \mathbf{E_h}_{B_1} \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{A_0}\mathbf{W}_{A_0B_0} + \mathbf{v}_{A_1}\mathbf{W}_{A_1B_0} \\ \mathbf{v}_{A_0}\mathbf{W}_{A_0B_1} + \mathbf{v}_{A_1}\mathbf{W}_{A_1B_1} \end{bmatrix} \quad , \quad \begin{matrix} \mathbf{E_h}_{B_0} \in \mathbb{B}^{1\times B} \\ \mathbf{E_h}_{B_1} \in \mathbb{B}^{1\times J-B} \end{matrix} \tag{3.4}$$

$$\mathbf{E_v} = \mathbf{HW}^{\mathrm{T}}$$

$$\Rightarrow \begin{bmatrix} \mathbf{E_v}_{A_0} \\ \mathbf{E_v}_{A_1} \end{bmatrix} = \begin{bmatrix} \mathbf{h}_{B_0}\mathbf{W}^{\mathrm{T}}_{A_0B_0} + \mathbf{h}_{B_1}\mathbf{W}^{\mathrm{T}}_{A_0B_1} \\ \mathbf{h}_{B_0}\mathbf{W}^{\mathrm{T}}_{A_1B_0} + \mathbf{h}_{B_1}\mathbf{W}^{\mathrm{T}}_{A_1B_1} \end{bmatrix} \quad , \quad \begin{matrix} \mathbf{E_v}_{A_0} \in \mathbb{B}^{1\times A} \\ \mathbf{E_v}_{A_1} \in \mathbb{B}^{1\times I-A} \end{matrix} \tag{3.5}$$

Thus, for any partition pair $(A_\alpha, B_\beta)$, the matrix form of the AGS equations (Eqs. 2.16-2.20) can be reformulated as a partitioned set:

$$\mathbf{V}^x_{A_\alpha} = \begin{cases} \mathbf{V}^0 & , \quad x = 0 \\ f(\mathbf{E_v}^{x-1}_{A_\alpha}) & , \quad x \text{ is even} \\ \mathbf{V}^{x-1} & , \quad x \text{ is odd} \end{cases} \tag{3.6}$$

$$\mathbf{H}^x_{B_\beta} = \begin{cases} f(\mathbf{E_h}^{x-1}_{B_\beta}) & , \quad x \text{ is odd} \\ \mathbf{H}^{x-1} & , \quad x \text{ is even} \end{cases} \tag{3.7}$$

$$\mathbf{E_v}^x_{A_\alpha B_\beta} = (\mathbf{H}^x_{B_\beta})\mathbf{W}^{\mathrm{T}}_{A_\alpha B_\beta}[k] \tag{3.8}$$

$$\mathbf{E_v}^x_{A_\alpha} = \sum_{\beta=0}^{1} \mathbf{E_v}^x_{A_\alpha B_\beta} \tag{3.9}$$

$$\mathbf{E_h}^x_{A_\alpha B_\beta} = (\mathbf{V}^x_{B_\beta})\mathbf{W}_{A_\alpha B_\beta}[k] \tag{3.10}$$

$$\mathbf{E_h}^x_{B_\beta} = \sum_{\alpha=0}^{1} \mathbf{E_h}^x_{A_\alpha B_\beta} \tag{3.11}$$

$$\mathbf{W}_{A_\alpha B_\beta}[k+1] = \mathbf{W}_{A_\alpha B_\beta}[k] + \frac{\epsilon}{L}\left((\mathbf{V}^1_{A_\alpha})^{\mathrm{T}}\mathbf{H}^1_{B_\beta} + (\mathbf{V}^{X-1}_{A_\alpha})^{\mathrm{T}}(\mathbf{H}^{X-1}_{B_\beta})\right) \tag{3.12}$$

With the exception of Eqs. 3.9 and 3.11, the set of partitioned AGS equations are identical to the equations of a single RBM (Eqs. 2.16-2.20). Thus, this method of partitioning RBMs allows a collection of small RBM implementations to be amalgamated in order to implement a significantly larger RBM for the added requirement that the supplementary computation for Eqs. 3.9 and 3.11 is executed.

It is important to note that there is no restriction on the relative sizes of the partitions – the partition boundaries of $A$ and $B$ can be placed at any arbitrary location of the layers. Furthermore, although this example illustrated the partition as four components, $\{A_0, A_1, B_0, B_1\}$, it can be easily proved by induction that this method of partitioning holds for any number of components as long as the partitions are disjoint sets.

## 3.3 Implications

The ability to partition RBMs into smaller networks has profound effects on how they are implemented. The numerous benefits of network partitioning are a result of how the networks are divided. To understand this significance, the memory analysis described in Section 2.2 must be carefully considered: weight parameters and updates are both plentiful and resource intensive since they require $n^2$ $w$-bit, fixed-point values. Energies require significantly less resources with $n$ $w$-bit, fixed-point values while node states only require $n$ 1-bit values for each layer. Given the disproportionate balance of memory requirements, this method of partitioning RBMs specifically ensures that each partition has a unique set of private weights parameters and weight updates. Thus, there will be no transfer of weight variables allowing for a high degree of data locality. As a consequence, the only the partitioned energies and subsequent node states will be transferred, both of which have a size of $O(n)$. This trade-off is extremely advantageous to RBM implementations since the most resource intensive data can be stored locally while only relatively small amounts of data need to be transferred, ensuring a low communication-to-computation ratio.

The first effect of this partitioning method is that it allows RBMs to be implemented on multiple FPGAs. Although multiple FPGAs provide more resources, implementing effective designs that require data dependencies can be difficult since communication is detrimental to the overall computational throughput. This partitioning scheme resolves this issue by ensuring that the data dependencies require very little transfer of information since only the partial energies and node states, both of which are $O(n)$, need to be transferred. Furthermore, the scheme allows a high degree of coarse-grain parallelism since the FPGAs can compute a significant portion of the AGS equations (Eqs. 3.6-3.8, 3.10, 3.12) independently.

The next effect is that it allows large RBM networks to be virtualized on a much smaller hardware implementation. The results of a small hardware implementation can be stored in memory and the compute engines can be time-multiplexed so that all of the computation is completed. Unfortunately, this requires a complete context switch, which includes the expensive swapping of weight memory resources. However, the frequency of context switches can be significantly reduced by increasing the batch size – rather than swapping after the energies are computed for a single RBM context, the same weight data will be kept locally until the entire batch is computed. In the pseudocode of Fig. 2.5, this would be analogous to looping through the batches before the AGS phases by switching the indices in lines 2 and 5. This would allow the cost of a single context switch to be amortized across the entire batch.

The final effect is that this method allows the design of an architecture with significant limitations on the network topology. These limitations can be imposed so that the resulting designs can be efficiently mapped to hardware implementations, as explored in Section 4.4.1.1. Since the partitioning method is versatile in terms of the number and size of each partition, the original limitations can be easily overcome by integrating groups of these limited networks together. This allows for extremely effective hardware designs that still have the versatility to implement a wide variety of network topologies.

# Chapter 4

# FPGA Restricted Boltzmann Machine Architecture

## 4.1 Design Philosophy

This section discusses the design philosophy that guided the conception and implementation of the FPGA architecture. First, the design goals are outlined and the major criteria for evaluation are described in detail. Next, the benefits of using FPGAs as a design platform are discussed – in particular, how the implementation details of RBMs map to FPGAs is thoroughly analyzed.

### 4.1.1 Design Goals

To properly design the architecture, the goals of the project must be described. In order of importance, the three design goals are as follows:

**Scalable** First, the FPGA implementation must be scalable with respect to both resources and performance – an increase in available resources should result in a proportionate increase in neural network size as well as an increase in performance. Specifically, an $O(n)$ resource utilization is desirable. The resources in an FPGA are rapidly increasing and the proposed architecture must be able to take advantage of the additional resources.

Furthermore, there is an increasing trend for high-performance computing to incorporate multiple FPGAs in a single design. The architecture must be able to scale across multiple FPGAs as well in such a manner that additional performance can be achieved by adding more FPGAs to the implementation. Thus, maintaining an effective communication-to-computation ratio is an important criteria.

**High-Performance** The FPGA implementation must be significantly faster than any software implementation. For hardware systems, the time and effort required for development and end-user application is much higher than software implementations, and thus, the performance must be significantly faster to justify the efforts for a hardware design.

As outlined in Section 2.2, the overall computational complexity is $O(n^2)$. The proposed architecture must take advantage of the inherent parallelism in RBM to achieve a computational complexity of $O(n)$, resulting in significant acceleration due to the scalability.

**Versatility** As with all neural networks, the network topology dictates the performance of the RBM. For RBMs, the topology is defined by a number of parameters including the number of layers, visible nodes, hidden nodes, as well as batch size and number of AGS phases. The architecture must provide the end-user with enough versatility to accommodate every topology without sacrificing scalability or performance.

### 4.1.2 Benefits of Using FPGA Platforms

In addition to the design goals, understanding which characteristics of RBM networks maps effectively to FPGAs is vital in designing the proposed architecture. The following list of characteristics, in no particular order, describes how certain properties can be efficiently implemented on FPGA technology:

**Parallelism and Replicated Computation** Neural networks have a high degree of inherent parallelism since all of the computations regarding a single node are independent. Furthermore, a single hardware design can be used for every node since the computation is identical for each node. These two properties allow for an effective FPGA implementation since deep pipelines for the data path can be used. The heavy use of pipeline implementations ensures high computational throughput while maintaining a low resource utilization.

**Data Locality** Neural networks have a high degree of data locality since computation only requires repeated access to the same set of weights and node states. By storing all of the data on-chip, low-latency memory, high performance computation can be achieved. When the memory requirements exceed the available local resources, the data locality still provides a computation benefit since a single set of weights can be used repeatedly for a series of computations and is only swapped out of local memory infrequently.

**Customized Memory Structures** As noted in Section 2.2, the data types are wildly varying in terms of resource utilization and access patterns. The benefit of using an FPGA platform is the explicit control over how the hardware accesses memory. This is of particular interest since the high degree of parallelism requires a significant amount of memory bandwidth to ensure that the computational hardware is not idle.

To minimize communication, the proposed architecture will store the $O(n^2)$ data types of weights parameters and weight updates in such a manner that greatly reduces the frequency at which they are swapped. To compensate, the $O(n)$ data types of node states and energies will be replicated and distributed accordingly.

**Binary Data Types** Since RBM node states are binary valued, computations can be effectively mapped to FPGA implementations, which are able to provide efficient bit-wise operations. The finer granularity provided by FPGAs provides a more efficient use of resources compared to other platforms. Furthermore, binary data types reduce the complexity of arithmetic operations: for example, multiplication operations are reduced to logical AND operations, which require significantly less resources.

**Limited Range** Although a significant portion of RBM data types are real-valued; such as energies, weight parameters and weight updates; none of the data types require dynamic range or floating-point representation. Instead, fixed-point representation with a fixed radix is sufficient for most RBM implementations. This maps well to FPGA platforms since fixed-point arithmetic units have efficient resource utilization.

## 4.2 Message Passing Infrastructure

### 4.2.1 Motivation

The proposed architecture is built on top of the message passing ideology – an embedded FPGA implementation of the Message Passing Interface (MPI) is used as an underlying infrastructure for the architecture [17]. This infrastructure provides numerous benefits including a high-performance communication protocol, a straightforward software-hardware interface as well as portability and versatility.

The embedded MPI implementation provides low-latency, high-bandwidth software and hardware implementations. The designs are light-weight and do not require significant resources. Furthermore, there is little overhead in terms of message preparation as well as communication protocol – the hardware implementation incurs a three cycle latency to prepare each message. As a result, this MPI implementation provides a high-performance communication infrastructure for the various hardware components in the RBM architecture.

Furthermore, the MPI implementation provides a straightforward software-hardware interface. MPI is a popular programming model used in high-performance computing and is often applied in large cluster-based systems. This embedded implementation maintains the widely used interface and allows different processes, called *ranks*, to communicate with each other without knowing whether the other party is implemented in hardware or software. By using MPI, end-users of the RBM architecture can simply write MPI software without understanding any of the underlying hardware implementation. Furthermore, the software implementation is not limited to standalone embedded processors – compatible software implementations for

Intel x86 processors and embedded PowerPC processors running Linux kernels have been fully implemented, extending the MPI interface to common platforms.

Finally, the MPI implementation provides extensive portability and versatility. By using MPI, the communication infrastructure becomes independent of the hardware platform, allowing the same architecture to be instantiated on different generations of FPGA chips without any changes to the hardware. Rather than relying on a variety of bus architectures, a common MPI interface drastically increases the portability of the design. Next, MPI allows the architecture to be versatile and reconfigurable. Individual processes can be selectively added to achieve the desired performance and functionality, allowing for a modular design. This concept is further extended for multi-FPGA platforms – the physical location and routing to individual ranks is abstracted from functionality of the hardware core.

### 4.2.2 Implications

The underlying MPI infrastructure has fundamental implications on the RBM architecture. Each compute core will be implemented as an independent MPI rank that is responsible for a particular subset in executing the AGS equations (Eqs. 3.6-3.12). The memory and computational resources required for a single task are entirely encapsulated within the single core providing an equation-to-hardware mapping. Variables will be transferred between cores via messages and the contents of the message will be described by a parameter called the *MPI Tag*, which are fully listed in Appendix A. This implementation provides an effective method of abstracting the functionality and communication protocol of each hardware core.

An important consequence of using the MPI infrastructure is the effect of communication on the heterogeneous environment. There are two types of interactions: software-hardware interactions between embedded processors and compute engines, and hardware-hardware interactions between a pair of compute engines.

In embedded FPGA designs, processors play a vital role as control and memory distribution units, and often perform very little computation. Unfortunately, they are significantly slower than their hardware counterparts. As a result, software-hardware interactions must be used selectively. In the embedded MPI implementation, there are a number of enhancements for this type of interaction – in particular, the design of a hardware engine that provides MPI functionality through direct memory access allows for high-bandwidth burst memory transactions for their respective messages. Although the processor only needs to write four words to the hardware engine to begin data transfer, that creates a relatively high overhead compared to the speed of hardware engines. Thus, for software-hardware interactions, sending frequent messages results in a noticeable overhead.

Hardware-hardware interactions are significantly more homogeneous since it is easy to design hardware that transfers messages at the same rate. Also, initiating a message can be done in a few cycles. In addition, the hardware MPI implementation provides a much finer grain approach to messages since the parallel nature of hardware allows data to be operated on as soon as it is received. Thus, the size and frequency of hardware-hardware interactions is not a primary factor in the design of hardware engines.

## 4.3 Architecture Platforms

This section describes various platforms to illustrate the capabilities and features of the architecture from a top-level perspective. The hardware cores are introduced and their role in implementing RBMs are explained. Furthermore, baseline platforms are presented to show the mapping between a schematic RBM and its hardware implementation. To illustrate how the architecture provides versatility to implement a variety of RBMs, three different types of platforms are discussed: single FPGA, multi-FPGA, and virtualized systems.

### 4.3.1 Hardware Core Overview

The architecture consists of three different cores that implement various components of the partitioned AGS equations (Eqs. 3.6-3.12): the Restricted Boltzmann Machine Core (RBMC),

| Compute Engine | Inputs | Outputs | Equation |
|---|---|---|---|
| **Restricted Boltzmann Machine Core (RBMC)** | | | |
| Energy Compute Engine | $\mathbf{v}, \mathbf{h}, \mathbf{W}$ | $\mathbf{E_v}, \mathbf{E_h}$ | 3.8, 3.10 |
| Weight Update Compute Engine | $\mathbf{v}, \mathbf{h}, \epsilon, \mathbf{W}, \mathbf{\Delta W}$ | $\mathbf{\Delta W}, \mathbf{W}$ | 3.12 |
| **Node Select Core (NSC)** | | | |
| Threshold Node Select | $\mathbf{E_v}, \mathbf{E_h}$ | $\mathbf{v}, \mathbf{h}$ | 2.6, 2.7, 3.6, 3.7 |
| Sigmoid Probabilistic Node Select | $\mathbf{E_v}, \mathbf{E_h}$ | $\mathbf{v}, \mathbf{h}$ | 2.4, 2.5, 3.6, 3.7 |
| **Energy Accumulator Core (EAC)** | | | |
| Streaming Implementation | $\mathbf{v}, \mathbf{h}, \mathbf{E_v}, \mathbf{E_h}$ | $\mathbf{v}, \mathbf{h}, \mathbf{E_v}, \mathbf{E_h}$ | 3.9, 3.11 |
| RAM Implementation | $\mathbf{v}, \mathbf{h}, \mathbf{E_v}, \mathbf{E_h}$ | $\mathbf{v}, \mathbf{h}, \mathbf{E_v}, \mathbf{E_h}$ | 3.9, 3.11 |

Table 4.1: Descriptions of the three hardware cores in the RBM architecture and the mapping of hardware to the AGS equations.

Node Select Core (NSC) and Energy Accumulator Core (EAC). By supporting the complete set of partitioned AGS equations, an arbitrary RBM can be implemented by selecting the appropriate cores and organizing them into a suitable platform.

Table 4.1 provides a detailed description of the hardware cores and their mapping to the AGS equations. It is important to understand that the architecture does not attempt to generate hardware that resembles a schematic diagram of an RBM. For example, there is no hardware that connects a visible and hidden node with a weighted connection, as described in Fig. 2.1. Instead, the architecture achieves its high performance through implementing the less discernible AGS equations in an efficient manner.

By focusing on the equations rather than the schematic diagrams, it can be difficult to identify RBM components and understand how the hardware realizes the required AGS equations. It is more efficient to understand the mapping of the schematic diagram to the pseudocode described in Fig. 2.5, followed by the mapping of the pseudocode to the AGS equations, and finally the mapping of the AGS equations to the hardware cores.

Each of the cores are independent and are responsible for encapsulating their computational and memory resources. The cores uses MPI hardware as a communication infrastructure and transfer messages with the appropriate information. Also, a processor is used as a controller for the entire system and also manages access to main memory.

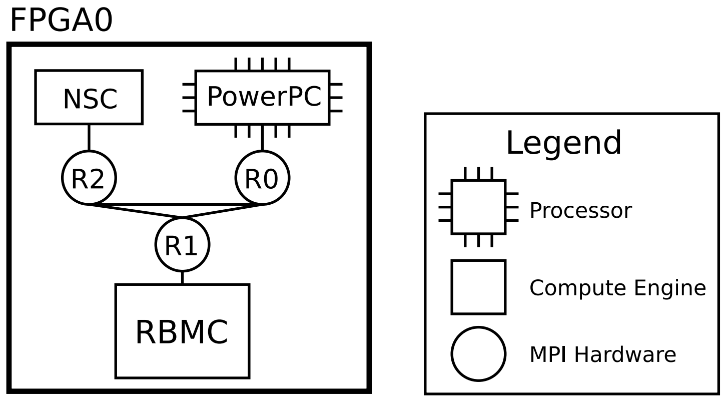

Figure 4.1: The block diagram for the single FPGA system.

### 4.3.2 Single FPGA Platform

The first platform that is described is the single FPGA system. This platform is used to illustrate how the processor, RBMC and NSC cores are assembled to implement a symmetric RBM network. Each core is designated as a rank in the MPI network and transfers information via messages. A schematic diagram of the platform is shown in Fig. 4.1.

The operation of this platform follows a specific order of computation and communication. First, the processor initializes the RBMC with instructions followed by the learning rate and the initial data vector. After the RBMC receives the instruction, it begins to calculate the corresponding AGS equations. As it generates the partial energies, the RBMC sends them directly to the NSC, which computes the node states. The states are then sent back to the RBMC for the next phase of the AGS cycle. By transferring data back and forth, the RBMC and NSC are capable of computing the required AGS equations for a symmetric RBM network. This process is described in further detail in Table 4.2.

| No. | Hardware status | | | Comment |
|---|---|---|---|---|
| | Processor Rank 0 | RBMC Rank 1 | NSC Rank 2 | |
| 1 | S1 | R0 | | Processor sends instructions to RBMC |
| 2 | S1 | R0 | | Processor sends learning rate to RBMC |
| | | | | *Batch = 0* |
| 3 | S1 | R0 | | *AGS phase = 0*<br>Processor sends initial data to RBMC (Eq. 3.6, ln 3) |
| 4 | | C | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 5 | | S2 | R1 | RBMC sends hidden energy to NSC |
| 6 | | | C | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 7 | | R2 | S1 | *AGS phase = 1*<br>NSC sends hidden node states to RBMC |
| 8 | | C | | RBMC computes positive weight update (Eq. 3.12, ln 28-30) |
| 9 | | C | | RBMC computes visible energy (Eq. 3.8, ln 18-20) |
| 10 | | S2 | R1 | RBMC sends visible energy to NSC |
| 11 | | | C | NSC computes visible node states (Eq. 3.6, ln 23-24) |
| 12 | | R2 | S1 | *AGS phase = 2*<br>NSC sends visible node states to RBMC |
| 13 | | C | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 14 | | S2 | R1 | RBMC sends hidden energy to NSC |
| 15 | | | C | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 16 | | R2 | S1 | *AGS phase = 3*<br>NSC sends hidden node states to RBMC |
| 17 | | C | | RBMC computes negative weight update (Eq. 3.12, ln 32-34) |
| | | | | *Batch = 1* |
| 18 | S2 | R1 | | *AGS phase = 0*<br>Processor sends initial data to RBMC (Eq. 3.6, ln 3) |
| 19 | | C | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 20 | | S2 | R1 | RBMC sends hidden energy to NSC |
| 21 | | | C | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 22 | | R2 | S2 | *AGS phase = 1*<br>NSC sends hidden node states to RBMC |
| 23 | | C | | RBMC computes positive weight update (Eq. 3.12, ln 28-30) |
| 24 | | C | | RBMC computes visible energy (Eq. 3.8, ln 18-20) |
| 25 | | S2 | R1 | RBMC sends visible energy to NSC |
| 26 | | | C | NSC computes visible node states (Eq. 3.6, ln 23-24) |
| 27 | | R2 | S1 | *AGS phase = 2*<br>NSC sends visible node states to RBMC |
| 28 | | C | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 29 | | S2 | R1 | RBMC sends hidden energy to NSC |
| 30 | | | C | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 31 | | R2 | S1 | *AGS phase = 3*<br>NSC sends hidden node states to RBMC |
| 32 | | C | | RBMC computes negative weight update (Eq. 3.12, ln 32-34) |
| | | | | *Finish AGS* |
| 33 | | C | | RBMC commits weight updates (Eq. 3.12, ln 37-39) |

Table 4.2: Description of the communication and computation for a single FPGA system. A batch size of two is used and three AGS phases are computed per data vector. The equivalent AGS equations and pseudocode line numbers in Fig. 2.5 are shown in parenthesis, respectively. The following notation is used for the hardware status: *blank cell* is idle, S*n* is sending message to rank *n*, R*n* is receiving message from rank *n*, and C is computing.

### 4.3.3 Multi-FPGA Platform

The second platform presented is the multi-FPGA platform. This platform shows how the streaming EAC hardware core can be used to provide support for larger networks with comparable performance to the single FPGA platform. Only a single processor is used but multiple instances of the hardware cores are now distributed amongst numerous FPGAs. The multi-FPGA platform uses the partitioning method to amalgamate these smaller cores to behave as a single, larger RBM. High-performance is maintained through coarse grain parallelism since the cores can complete their computation independently.

Two designs will be introduced to illustrate the multi-FPGA platform: one design that uses two FPGAs and another that uses four FPGAs. The two FPGA design illustrates how RBM partitioning can be used to overcome the hardware limitations of the RBMC. In particular, the RBMC can only support RBMs with symmetrically sized layers; this is further discussed in detail in Section 4.4.1.1. The two FPGA design shows how the partitioning method extends the capabilities of the architecture by supporting vastly asymmetrical RBMs. The four FPGA design shows how the resource utilization scales for larger networks. For both of these designs, the limited resources will be illustrated by the restriction that a single FPGA can only support an RBMC with a network size of $32 \times 32$ – however, the NSC and streaming EAC have negligible resource utilization and multiple instances of these cores can be instantiated on a single FPGA without concerns.

#### 4.3.3.1 Two-FPGA platform

Using two FPGAs, two different RBMCs with network sizes of $32 \times 32$ can be instantiated with sufficient resources for one RBMC on each FPGA. Through the use of the partitioning method, a larger RBM with a network size of $64 \times 32$ can be achieved. This type of platform illustrates how highly asymmetrical networks can be composed of these smaller symmetrical networks while maximizing resource utilization. A schematic diagram of the two FPGA platform is shown in Fig. 4.2.

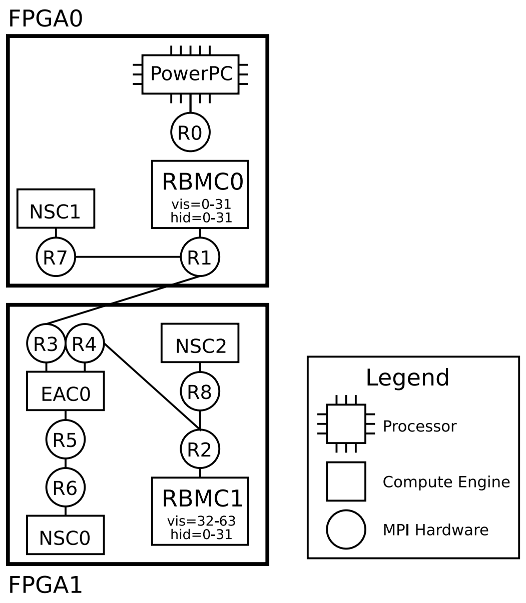

Figure 4.2: The block diagram for the two FPGA system. The node state indices for each RBMC are listed in the core. The processor (rank 0) is connected to all other ranks but the connections are not shown for clarity.

In this example, there are 64 visible nodes and 32 hidden nodes. The first RBMC stores the weights associated with the bottom half of the visible nodes and all of the hidden nodes, while the second RBMC stores the weights associated with the top half of the visible nodes and all of the hidden nodes. Although each RBMC can run independently providing coarse-grain parallelism, determining the node states depends on both sets of hidden energies. This achieved through the inclusion of the streaming EAC core.

As a overview of the processing order, each RBMC receives its instruction and initial node states from the processor. The RBMCs concurrently calculate their partitioned partial energies (Eqs. 3.8, 3.10). To determine the hidden node states, both RBMCs send their respective visible energies to the EAC. These energy vectors are summed to obtain the partial energies required for node selection (Eqs. 3.9, 3.11) and are transferred to the single NSC. The node states are then forwarded to both RBMCs, ensuring consistent hidden nodes. To determine the visible node states, each RBMC can individually generate the required partial energy terms and thus no extra communication is required – the RBMCs send the energies to their corresponding NSC. This process is described in further detail in Table 4.3.

| No. | Hardware status | | | | | | | Comment |
|---|---|---|---|---|---|---|---|---|
| | Proc.<br>Rank 0 | RBMC0<br>Rank 1 | RBMC1<br>Rank 2 | EAC0<br>R3-5 | NSC0<br>Rank 6 | NSC1<br>Rank 7 | NSC2<br>Rank 8 | |
| 1 | S1,S2 | R0 | R0 | | | | | Processor sends instructions to RBMCs |
| 2 | S1,S2 | R0 | R0 | | | | | Processor sends learning rate to RBMCs |
| | | | | | | | | *Batch = 0* |
| 3 | S1,S2 | R0 | R0 | | | | | *AGS phase = 0*<br>Processor sends initial data to RBMCs (Eq. 3.6, ln 3) |
| 4 | | C | C | | | | | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 5 | | S3 | S4 | R1,R2 | | | | RBMCs sends partitioned hidden energy to EAC |
| 6 | | | | C | | | | EAC computes sum of energies (Eq. 3.11, ln 8-10) |
| 7 | | | | S6 | R5 | | | EAC sends hidden energy to NSC |
| 8 | | | | | C | | | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 9 | | | | R6 | S5 | | | NSC sends hidden node states to EAC |
| 10 | | R3 | R4 | S1,S2 | | | | *AGS phase = 1*<br>EAC sends hidden node states to RBMCs |
| 11 | | C | C | | | | | RBMCs computes positive weight update (Eq. 3.12, ln 28-30) |
| 12 | | C | C | | | | | RBMCs computes visible energy (Eq. 3.8, ln 18-20) |
| 13 | | S7 | S8 | | | R1 | R2 | RBMCs sends visible energy to NSCs |
| 14 | | | | | | C | C | NSCs computes visible node states (Eq. 3.6, ln 23-24) |
| 15 | | R7 | S8 | | | S1 | S2 | *AGS phase = 2*<br>NSCs sends visible node states to RBMCs |
| 16 | | C | C | | | | | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 17 | | S3 | S4 | R1,R2 | | | | RBMCs sends partitioned hidden energy to EAC |
| 18 | | | | C | | | | EAC computes sum of energies (Eq. 3.11, ln 8-10) |
| 19 | | | | S6 | R5 | | | EAC sends hidden energy to NSC |
| 18 | | | | | C | | | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 19 | | | | R6 | S5 | | | NSC sends hidden node states to EAC |
| 20 | | R3 | R4 | S1,S2 | | | | *AGS phase = 3*<br>EAC sends hidden node states to RBMCs |
| 21 | | C | C | | | | | RBMCs computes negative weight update (Eq. 3.12, ln 32-34) |
| | | | | | | | | *Batch = 1* |
| 22 | S1,S2 | R0 | R0 | | | | | *AGS phase = 0*<br>Processor sends initial data to RBMCs (Eq. 3.6, ln 3) |
| 23 | | C | C | | | | | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 24 | | S3 | S4 | R1,R2 | | | | RBMCs sends partitioned hidden energy to EAC |
| 25 | | | | C | | | | EAC computes sum of energies (Eq. 3.11, ln 8-10) |
| 26 | | | | S6 | R5 | | | EAC sends hidden energy to NSC |
| 27 | | | | | C | | | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 28 | | | | R6 | S5 | | | NSC sends hidden node states to EAC |
| 29 | | R3 | R4 | S1,S2 | | | | *AGS phase = 1*<br>EAC sends hidden node states to RBMCs |
| 30 | | C | C | | | | | RBMCs computes positive weight update (Eq. 3.12, ln 28-30) |
| 31 | | C | C | | | | | RBMCs computes visible energy (Eq. 3.8, ln 18-20) |
| 32 | | S7 | S8 | | | R1 | R2 | RBMCs sends visible energy to NSCs |
| 33 | | | | | | C | C | NSCs computes visible node states (Eq. 3.6, ln 23-24) |
| 34 | | R7 | S8 | | | S1 | S2 | *AGS phase = 2*<br>NSCs sends visible node states to RBMCs |
| 35 | | C | C | | | | | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 36 | | S3 | S4 | R1,R2 | | | | RBMCs sends partitioned hidden energy to EAC |
| 37 | | | | C | | | | EAC computes sum of energies (Eq. 3.11, ln 8-10) |
| 38 | | | | S6 | R5 | | | EAC sends hidden energy to NSC |
| 39 | | | | | C | | | NSC computes hidden node states (Eq. 3.7, ln 13-14) |
| 40 | | | | R6 | S5 | | | NSC sends hidden node states to EAC |
| 41 | | R3 | R4 | S1,S2 | | | | *AGS phase = 3*<br>EAC sends hidden node states to RBMCs |
| 42 | | C | C | | | | | RBMCs computes negative weight update (Eq. 3.12, ln 32-34) |
| | | | | | | | | *Finish AGS* |
| 43 | | C | C | | | | | RBMCs commits weight updates (Eq. 3.12, ln 37-39) |

Table 4.3: Description of the communication and computation for a two FPGA system. A batch size of two is used and three AGS phases are computed per data vector. The equivalent AGS equations and pseudocode line numbers in Fig. 2.5 are shown in parenthesis, respectively. The following notation is used for the hardware status: *blank cell* is idle, S*n* is sending message to rank *n*, R*n* is receiving message from rank *n*, and C is computing.

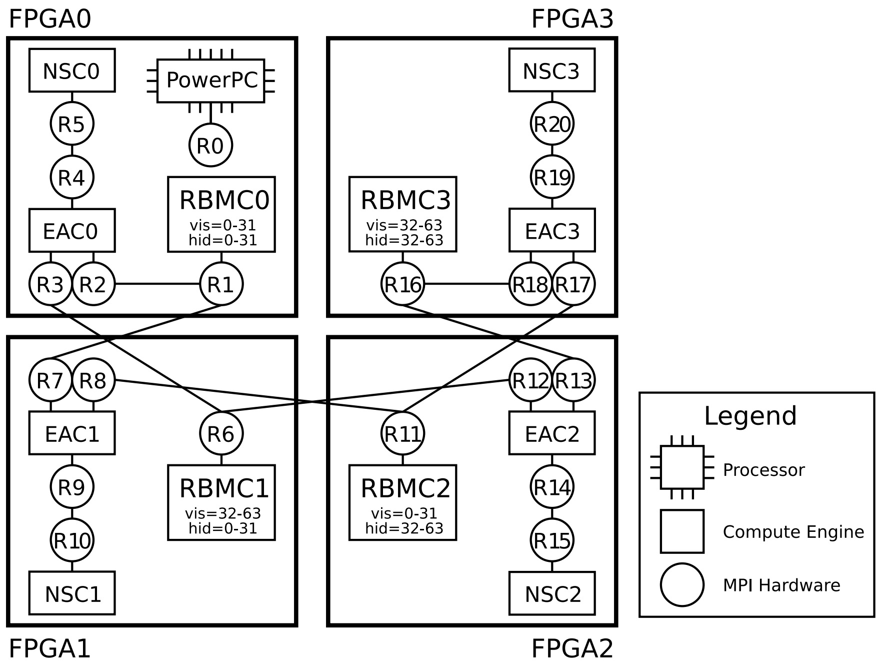

Figure 4.3: The block diagram for the four FPGA system. The node states are listed below the RBMC. Rank 0 is connected to all other ranks but the connections are not shown for clarity.

#### 4.3.3.2 Four FPGA platform

With four FPGAs, a large RBM with a network size of $64 \times 64$ can be composed of four separate $32 \times 32$ RBMCs. This illustrates how adding additional FPGAs provides parallelism and scalability with respect to network sizes. Each RBMC can compute their results independently, providing a coarse grain parallel design. However, the scalability is limited since the cumulative resources required to store the respective weight matrix grows with a rate of $O(n^2)$, while adding another FPGA to the design only provides a constant-size or $O(1)$ increase in resources. Since the scalability is extremely limited, this type of platform should be used to achieve additional performance through coarse grain parallelism by adding FPGAs to the design instead of attempting to implement a very large RBM. A schematic diagram of the four FPGA platform is shown in Fig. 4.3.

In terms of the processing order, the four FPGA design operates in a similar manner to the two FPGA design with the addition that the EAC is used to accumulate the energies for both the visible and hidden nodes. This process is described in further detail in Table 4.4.

| No. | Hardware status |  |  |  |  |  |  |  |  |  |  |  |  | Comment |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | Proc. Rank 0 | RBMC0 Rank 1 | RBMC1 Rank 6 | RBMC2 Rank 11 | RBMC3 Rank 16 | EAC0 R2-4 | EAC1 R7-9 | EAC2 R12-4 | EAC3 R16-19 | NSC0 Rank5 | NSC1 Rank10 | NSC2 Rank15 | NSC 3 Rank20 |  |
| 1 | S1,S6 S16,S11 | R0 | R0 | R0 | R0 |  |  |  |  |  |  |  |  | Processor sends instructions to RBMCs |
| 2 | S1,S6 S16,S11 | R0 | R0 | R0 | R0 |  |  |  |  |  |  |  |  | Processor sends learning rate to RBMCs |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | *Batch = 0* |
| 3 | S1,S6 S16,S11 | R0 | R0 | R0 | R0 |  |  |  |  |  |  |  |  | *AGS phase = 0* Processor sends initial data to RBMCs (Eq. 3.6, ln 3) |
| 4 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 5 |  | S2 | S3 | S17 | S18 | R1,R6 |  |  | R11,R16 |  |  |  |  | RBMCs sends partitioned hidden energy to EACs |
| 6 |  |  |  |  |  | C |  |  | C |  |  |  |  | EACs computes sum of energies (Eq. 3.11, ln 8-10) |
| 7 |  |  |  |  |  | S5 |  |  | S20 | R4 |  |  | R19 | EACs sends hidden energy to NSCs |
| 8 |  |  |  |  |  |  |  |  |  | C |  |  | C | NSCs computes hidden node states (Eq. 3.7, ln 13-14) |
| 9 |  |  |  |  |  | R5 |  |  | R20 | S4 |  |  | S19 | NSCs sends hidden node states to EACs |
| 10 |  | R2 | R3 | R17 | R18 | S1,S6 |  |  | S11,S16 |  |  |  |  | *AGS phase = 1* EAC sends hidden node states to RBMCs |
| 11 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs computes positive weight update (Eq. 3.12, ln 28-30) |
| 12 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs computes visible energy (Eq. 3.8, ln 18-20) |
| 13 |  | S7 | S12 | S8 | S38 |  | R1,R11 | R6,R16 |  |  |  |  |  | RBMCs sends partitioned visible energy to EACs |
| 14 |  |  |  |  |  |  | C | C |  |  |  |  |  | EACs computes sum of energies (Eq. 3.9, ln 18-20) |
| 15 |  |  |  |  |  |  | S10 | S15 |  |  | R9 | R14 |  | EACs sends hidden energy to NSCs |
| 16 |  |  |  |  |  |  |  |  |  |  | C | C |  | NSCs computes visible node states (Eq. 3.6, ln 23-24) |
| 17 |  |  |  |  |  |  | R10 | R15 |  |  | S9 | S14 |  | NSCs sends visible node states to EACs |
| 18 |  | R7 | R12 | R8 | R38 |  | S1,S11 | S6,S16 |  |  |  |  |  | *AGS phase = 2* EACs sends visible node states to RBMCs |
| 19 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs computes hidden energy (Eq. 3.10, ln 8-10) |
| 20 |  | S2 | S3 | S17 | S18 | R1,R6 |  |  | R11,R16 |  |  |  |  | RBMCs sends partitioned hidden energy to EACs |
| 21 |  |  |  |  |  | C |  |  | C |  |  |  |  | EACs computes sum of energies (Eq. 3.11, ln 8-10) |
| 22 |  |  |  |  |  | S5 |  |  | S20 | R4 |  |  | R19 | EACs sends hidden energy to NSCs |
| 23 |  |  |  |  |  |  |  |  |  | C |  |  | C | NSCs computes hidden node states (Eq. 3.7, ln 13-14) |
| 24 |  |  |  |  |  | R5 |  |  | R20 | S4 |  |  | S19 | NSCs sends hidden node states to EACs |
| 25 |  | R2 | R3 | R17 | R18 | S1,S6 |  |  | S11,S16 |  |  |  |  | *AGS phase = 3* EAC sends hidden node states to RBMCs |
| 26 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs computes positive weight update (Eq. 3.12, ln 32-34) |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | *Batch = 1* |
| 27-49 |  |  |  |  |  |  |  |  |  |  |  |  |  | Same as steps 3-26 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  | *Finish AGS* |
| 50 |  | C | C | C | C |  |  |  |  |  |  |  |  | RBMCs commits weight updates (Eq. 3.12, ln 37-39) |

Table 4.4: Description of the communication and computation for a four FPGA system. A batch size of two is used and three AGS phases are computed per data vector. The equivalent AGS equations and pseudocode line numbers in Fig. 2.5 are shown in parenthesis, respectively. The following notation is used for the hardware status: *blank cell* is idle, S$n$ is sending message to rank $n$, R$n$ is receiving message from rank $n$, and C is computing.

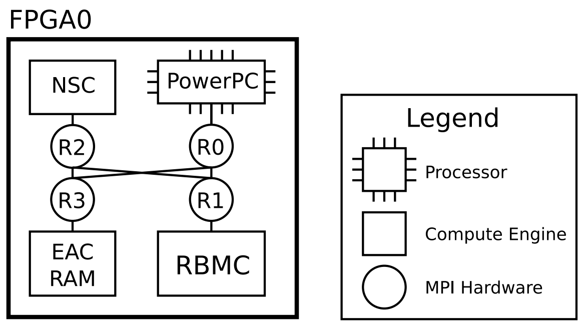

Figure 4.4: The block diagram for the virtualized FPGA system.

### 4.3.4 Virtualized FPGA Platform

The final platform is the virtualized FPGA platform. Although the multi-FPGA approach provides high-performance through coarse grain parallelism, the design has limited scalability that is fundamentally restricted by the size of the respective FPGAs. Thus, it is infeasible to implement networks that are orders of magnitude larger than what can be supported on a single FPGA. For example, the CUDA RBM implementation [15] is capable of supporting $4096 \times 11008$ networks, while a single Xilinx XC2VP70 can only support a network size of $128 \times 128$. Instead, a method of virtualizing the FPGA resources is described that allows a single set of hardware to be time multiplexed. Although this approach does not provide the same performance as the multi-FPGA platform, it can support much larger RBMs while using less resources. A schematic diagram of the virtualized FPGA platform is shown in Fig. 4.4.

This platform introduces the RAM EAC implementation, which unlike its streaming counterpart, stores data required for the context switches. In this example, the four FPGA design is virtualized – it is easy to extrapolate the virtualization construct to two or more FPGAs. Since the virtualization requires a complete context switch, which includes the swapping of the $O(n^2)$ weight matrix, the implementation takes advantage of the data locality for increased performance. This is achieved by computing all possible batches with a single set of weights before requiring a switch. This process is described in further detail in Table 4.5. Although the process in Table 4.5 only illustrates the use case for a batch size of two, larger batch sizes will increase performance since a single weight swap can be amortized across more computations.

| No. | Hardware status | | | | Comment |
|---|---|---|---|---|---|
| | Proc. Rank 0 | RBMC Rank 1 | NSC Rank 2 | EAC Rank 3 | |
| 1 | S1 | R0 | | | Processor sends instructions to RBMC |
| 2 | S1 | R0 | | | Processor sends learning rate to RBMC |
| | | | | | *AGS Phase = 1* |
| 3 | S1 | R0 | | | *RBMC = 0*<br>Processor send weights corresponding to RBMC0 to RBMC |
| 4 | S1 | R0 | | | *Batch = 0*<br>Processor sends initial data to RBMC (Eq. 3.6, ln 3) |
| 5 | | C | | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 6 | R1 | S0 | | | RBMC sends hidden energy to processor |
| 7 | S1 | R0 | | | *Batch = 1*<br>Processor sends initial data to RBMC (Eq. 3.6, ln 3) |
| 8 | | C | | | RBMC computes hidden energy (Eq. 3.10, ln 8-10) |
| 9 | R1 | S0 | | | RBMC sends hidden energy to processor |
| 10-16 | | | | | Same as steps 3-9 with weights corresponding to RBMC1 |
| 17-23 | | | | | Same as steps 3-9 with weights corresponding to RBMC2 |
| 24-30 | | | | | Same as steps 3-9 with weights corresponding to RBMC3 |
| 31 | S3 | | | R0 | Processor all hidden energies to EAC |
| 32 | | | | C | EACs computes sum of energies (Eq. 3.11, ln 8-10) |
| 33 | | | R3 | S2 | EAC sends hidden energy to NSC |
| 34 | | | C | | NSC computes hidden node states (Eq. 3.6, ln 23-24) |
| 35 | | | S3 | R2 | NSC sends hidden node states to EAC |
| 36 | R3 | | | S0 | EAC sends hidden node states to Processor |
| 37-70 | | | | | *AGS Phase = 2*<br>Same as steps 3-36 with visible energies and nodes |
| 70-103 | | | | | *AGS Phase = 3*<br>Same as steps 3-36 |
| | | | | | *Weight Update* |
| 104 | S1 | R0 | | | Processor send weights corresponding to RBMC0 to RBMC |
| 105 | S1 | R0 | | | Processor visible and hidden node positive pair for Batch = 1 to RBMC |
| 106 | | C | | | RBMC computes positive weight update (Eq. 3.12, ln 28-30) |
| 107 | S1 | R0 | | | Processor visible and hidden node positive pair for Batch = 2 to RBMC |
| 108 | | C | | | RBMC computes positive weight update (Eq. 3.12, ln 28-30) |
| 109 | S1 | R0 | | | Processor visible and hidden node negative pair for Batch = 1 to RBMC |
| 110 | | C | | | RBMC computes negative weight update (Eq. 3.12, ln 32-34) |
| 111 | S1 | R0 | | | Processor visible and hidden node negative pair for Batch = 2 to RBMC |
| 112 | | C | | | RBMC computes negative weight update (Eq. 3.12, ln 32-34) |
| 113 | | C | | | RBMC commits weight updates (Eq. 3.12, ln 37-39) |
| 114 | R0 | S1 | | | RBMC send updated weights for RBMC0 to Processor |
| 105-115 | | | | | Same as steps 104-114 with weights corresponding to RBMC1 |
| 116-126 | | | | | Same as steps 104-114 with weights corresponding to RBMC2 |
| 127-137 | | | | | Same as steps 104-114 with weights corresponding to RBMC3 |

Table 4.5: Description of the communication and computation for a virtualized FPGA system computing the equivalent RBM as the four FPGA platform described in Section 4.3.3.2. A batch size of two is used and three AGS phases are computed per data vector. The equivalent AGS equations and pseudocode line numbers in Fig. 2.5 are shown in parenthesis, respectively. The following notation is used for the hardware status: *blank cell* is idle, S*n* is sending message to rank *n*, R*n* is receiving message from rank *n*, and C is computing.

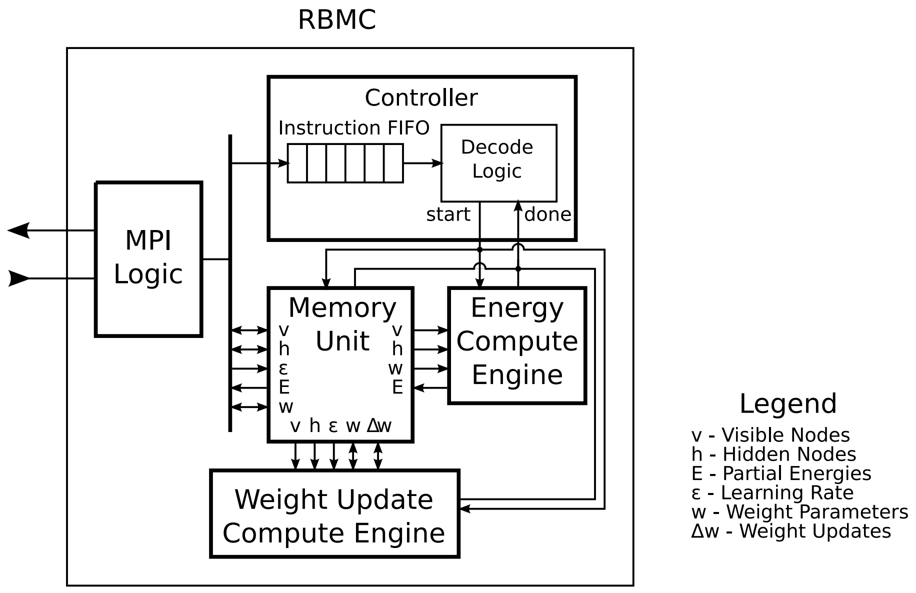

Figure 4.5: A block diagram of the RBMC indicating the interactions and data transfer between the major components; the controller, memory unit, as well as the energy and weight update compute engines.

## 4.4 Hardware Implementation

This section outlines the implementation details of the three types of cores in the RBM FPGA architecture: the Restricted Boltzmann Machine Core (RBMC), Node Select Core (NSC) and Energy Accumulator Core (EAC).

### 4.4.1 Restricted Boltzmann Machine Core

The Restricted Boltzmann Machine Core (RBMC) is the primary computational core of the FPGA RBM architecture. The RBMC is designed specifically to take advantage of the data locality of the weight variables – as a result, it is responsible for calculating partial energies and updating weights (Eqs. 3.8, 3.10 and 3.12). Recalling the time complexity analysis (Table 2.1), these are the two $O(n^2)$ sections that must be reduced to $O(n)$. Furthermore, the transmission of weights requires $O(n^2)$ words, while transmitting the partial energies or the node states are only $O(n)$. Thus, by storing all of the weights locally, the RBMC is able to achieve this performance goal.

This core itself is divided into four components: the microprogrammed controller, the memory organization, the energy compute engine and the weight update compute engine. A block diagram of the core is shown in Fig. 4.5. A functional C implementation of the RBMC is described in Appendix B.

#### 4.4.1.1 Network Restrictions

To achieve an efficient hardware mapping, rather than attempting to create hardware to suit the computational problem, it is far more efficient to restrict the computation problem in a manner that is well-suited for hardware design. Because FPGAs are reconfigurable, this approach is acceptable because the restrictions can be overcome through additional hardware and the RBM partitioning method. The following is a description of the limitations, their desired effect, and how they will be overcome.

- The weights will use a 32-bit fixed-point representation. A fixed-point representation is desirable because the simplified arithmetic units require less resources and are faster. This is acceptable since a 32-bit fixed-point number can provide sufficient resolution [14].
- The layers must be symmetric; the number of nodes in the visible and hidden layers must be the same ($I = J = n$). This maximizes hardware usage since the same hardware can be used to calculate the local energies (Eq. 3.8, 3.10). To understand how this limitation is acceptable, two cases must be analyzed:
  - For nearly symmetric RBMs ($I \sim J$), the larger layer size will be instantiated. The relative cost of wasting resources is negligible compared to the speed-up achieved. To ensure proper functionality, all the associated weights of non-existing nodes should be set to the most negative value.
  - For vastly asymmetric RBMs ($I \gg J$ or $I \ll J$), multiple RBMCs will be instantiated. By partitioning large RBMs into smaller components, significantly better performance can be achieved by taking advantage of the locality of the weights.

- The number of nodes in a layer must be a power of two ($n = 2^a$). Specific hardware, such as binary trees, can take advantage of this constraint for maximum speed-up for a given resource utilization. This is acceptable since layer sizes have approximately this resolution and exact sizes are not mandatory.

- Batch sizes must be a power of two ($L = 2^b$). This is desirable because calculating the weight update accumulation (Eq. 2.11) requires a division that can be implemented by an arithmetic bit shift only if this constraint is satisfied. This is acceptable because batch sizes can be chosen within this resolution.

- The number of AGS phases and learning rates are software inputs. This is a benefit rather than a limitation since it provides the end-user the ability to update the learning rate and AGS phases without hardware updates.

#### 4.4.1.2 RBMC Microarchitecture

Instead of the traditional approach of using finite state machines for logic control, the RBMC uses a microprogrammed controller with instructions retrieved from memory. There are several factors in the decision to use a controller approach. First, the RBMC is a complex core: it is responsible for a variety of compute engines and memories. Since the compute engines require shared access to the range of memories, the RBMC must implement arbitration and control flow to organize the computation. The microprogrammed approach provides an efficient method to control the hardware, and is shown in Fig. 4.5.

More importantly, the RBMC must be versatile since it is responsible for computing the major components of a small RBM. There are a variety of parameters that dictate the order as well as type of computation, and creating a state machine to capture all of the possible combinations would result in a complicated and ineffective implementation. As evidenced by Tables 4.2 and 4.5, the order of execution in a virtualized network is entirely different than in a single FPGA implementation – the RBMC must have the versatility to be able to support both platforms. By using a microprogram, the end-user can program instructions for the RBMC to execute, providing significant reconfigurability.

Finally, the architecture was designed to interface with an MPI software counterpart. As described in Section 4.2.2, the processor is used for control and memory distribution. By sending a message composed of instructions from a processor, the end-user has an effective software interface over the entire hardware implementation. This approach provides additional flexibility since the software can change the execution operation of the platform without resynthesizing the hardware design.

The microarchitecture was designed according to the following protocol. The FPGA processor gathers a series of instructions into a single MPI message and sends this to the RBMC. The instructions are a series of computation and MPI message instructions – there are no internal control flow or branch instructions. Thus, these instructions are stored in a First-In-First-Out (FIFO) data structure. An instruction is dequeued as soon as it is completed and the RBMC continues to process as long as the FIFO is not empty. Because the computation of the AGS equations (Eqs. 3.6-3.12) and the subsequent messages are well defined, the instructions are relatively complex, where a single instruction can describe several low-level operations. This high-level instruction set allows for efficient use of memory and resources without sacrificing versatility or performance. The instructions use a one-hot encoding scheme and are described in Appendix C.

A point of interest is the design of the controller hardware. A typical controller design with constant depth datapath is unsuitable for this architecture due to the wide variety of high-level tasks. There are two types of instructions: the *compute instructions*, which were designed to encapsulate a top-level computation required in the AGS equations; and the *message instructions*, which were designed to initiate the MPI protocol and transfer data using messages. As a result, each instruction has drastically different timing requirements. Thus, an approach inspired by asynchronous circuits was used where each compute engine uses a combination of handshaking signals to indicate their status. Each component has a *start* and *done* signal. When idle, all of the components pull their done signals high and start signals low. Due to the simplicity of the one-hot encoding, the instructions are decoded combinationally and the appropriate start signals are pulsed high. Upon receiving a start signal, each of the active components pull their done signals low and begin computing. As each component completes its task, the done signal is then pulled back high. The controller moves onto the next instruction when all of the done signals are high again. For high-performance hardware accelerators where versatility is required, this microprogrammed controller approach of using a complex instruction set with handshaking signals provides an interesting architecture.

#### 4.4.1.3 Memory Allocation and Storage

The design of the RBMC revolves around the memory core since the compute engines would be memory bandwidth limited otherwise; for a $128 \times 128$ hardware RBM running at 100MHz, the peak bandwidth usage is 102GB/s since 128 32-bit words are read and written at every clock cycle. As a result, the core takes advantage of the hardware distributed Block RAMs (BRAM) on the FPGA – the BRAMs have low latency and collecting them in parallel provides an aggregate, high-bandwidth port to support the compute engines.

The memory core is organized into storing five variables as described in Table 2.2. There are two major factors in deciding the required hardware data structures: the size of the variables and the required bandwidth. As a result, the memory resources are divided accordingly:

**Visible and Hidden Node States** These variables are very limited in size due to their binary values ($n$ bits) but require moderate bandwidth considerations ($n$ bits/cycle). Due to their limited size and bandwidth requirements, they are stored in two sets of flip-flops that allow for parallel access to all the node states simultaneously.

**Partial Energies** This variable is moderate in size ($32n$ bits) but require low bandwidth since only a single word needs to be read or written each cycle (32 bits/cycle). Due to their size and low bandwidth requirements, they are stored in a single BRAM since serial data transfer is sufficient. Furthermore, the single BRAM will be implemented as a FIFO data structure for implementation simplicity.

**Learning Rate** This variable is stored in a single 32-bit word and does not scale with network size. As a result, it is stored in a single register that provides sufficient bandwidth and effective resource utilization.

**Weight Parameters and Weight Updates** This variable is the largest in size ($32n^2$ bits) and requires high bandwidth to satisfy the computational engines by providing a single row or column of the corresponding matrix in a single cycle ($32n$ bits/cycle). Due to their size and high bandwidth requirements, they are implemented in a unique hardware data structure that is composed of $n$ BRAMs.

Access to the weight BRAMs provides an integrated data structure that is essential to obtaining performance speed-up. To calculate the energies, the weight matrix must be transposed (Eq. 3.8). There are a number of assumptions that can be safely made to allow for the use of this data structure. First, the computation requires a matrix-matrix operation to be done in hardware in a row- or column-wise manner; there will be no random access to an individual row or column in the matrix. The next assumption is that there are sufficient BRAMs; to reduce the $O(n^2)$ single memory accesses to $O(n)$ vector accesses, $n$ BRAMs must be dedicated to implement this system. The final assumption is that a non-standard element order in the vector is acceptable – as long as the non-standard order is deterministic, the compute engine can account for it. Reordering the matrix elements is extremely resource intensive. Instead, it is more efficient to manipulate the binary valued node states to mimic the non-standard element order.

This distributed BRAM-based matrix data structure will be illustrated with a $n = 4$ example (Fig. 4.6). Four BRAMS will be used to appropriately partition the $4 \times 4$ matrix – each element is labelled with $\alpha, \beta, \gamma, \delta$ to indicate the BRAM, followed by an integer to indicate the address within that BRAM. Fig. 4.6.a) illustrates how the standard or-

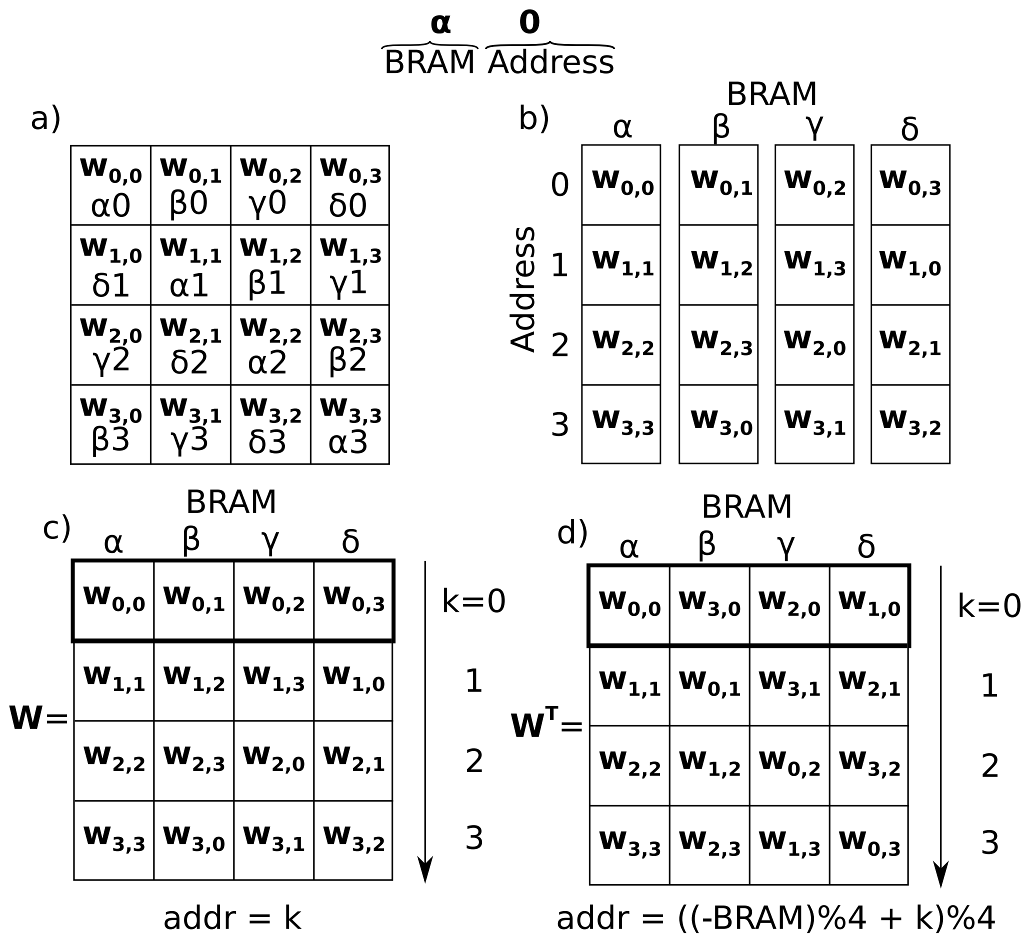

Figure 4.6: The distributed BRAM-based matrix transpose data structure for a $n = 4$ example.

ganization of the matrix is mapped to the various BRAMs. It is important to note that no BRAM has two elements on the same row or column. Fig. 4.6b) illustrates the elements in the matrix reorganized according to BRAM and address. Fig. 4.6c) illustrates the row-wise access to the matrix. To access row $k$, the address for each BRAM should be set to the expression $addr = k$. Fig. 4.6d) illustrates the column-wise (or conversely, the transposed row-wise) access to the matrix. To access column $k$, the address for each BRAM should be set to the expression $addr = ((-\text{BRAM})\%4 + k)\%4$, where $BRAM$ is the numerated label of each BRAM and $\%$ is the modulus operator. Thus, by following a specific distribution of the matrix and addressing scheme, an entire row or column of the matrix can be retrieved immediately with low resource utilization.

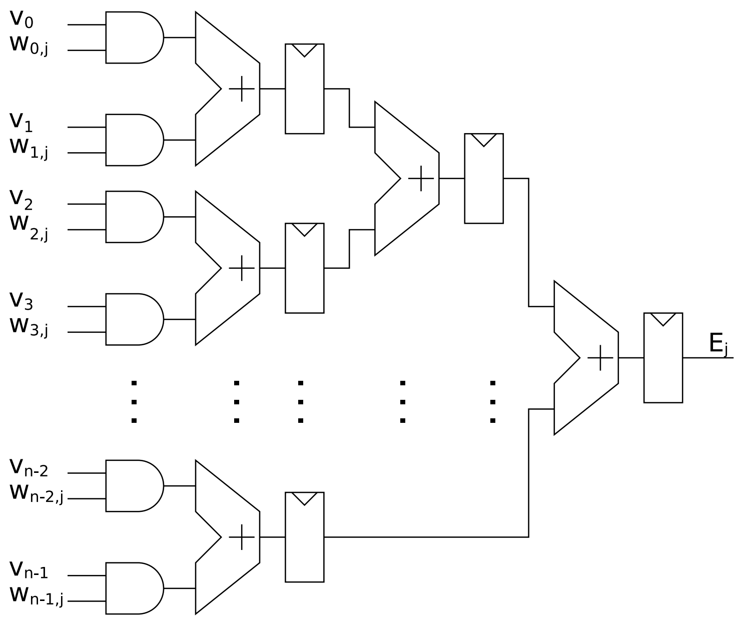

Figure 4.7: A circuit diagram of the binary adder tree set to calculate the hidden partial energies (Eq. 3.8).

This matrix data structure that supports the transfer of an entire row or column simultaneously while providing an instantaneous matrix transpose is crucial to implementing the weight matrix in an advantageous manner for the compute engines. Given that the weight parameters are stored in this fashion, the weight updates are also stored in a corresponding fashion to provide a straightforward implementation.

#### 4.4.1.4 Energy Compute Engine

The energy compute engine is responsible for calculating the energies (Eqs. 3.8, 3.10). To complete the vector-matrix operation, it requires one of the layers and the weights. At every clock cycle, the compute engine multiplies the vector layer with one of the columns or rows in the weight matrix to generate a scalar element in the column of the energy matrix. Because of the restrictions defined in Section 4.4.1.1, the computation can be done with simple hardware components: AND gates, multiplexers and registered, fixed-point adders.

The resulting hardware is a deep but low resource pipeline with short critical paths. The deep pipeline takes advantage of the inherent parallelism and replicated computation in RBMs

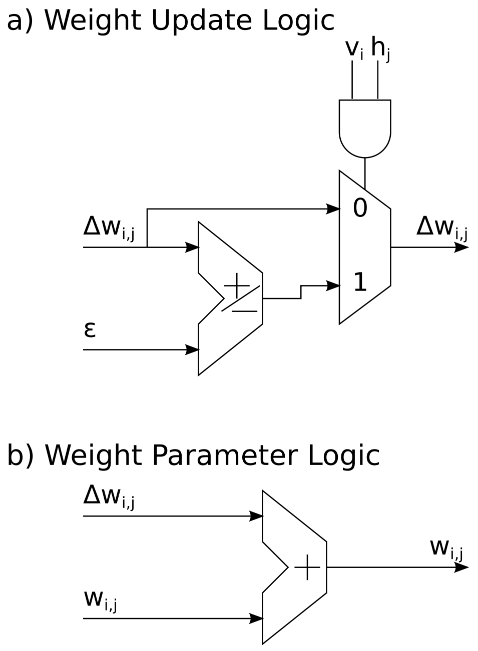

Figure 4.8: A circuit diagram of a) the weight update logic and b) weight parameter logic for an individual memory component (Eq. 3.12).

by time-sharing the arithmetic logic at every clock cycle. Since no flow control is required, this hardware implementation computes a single partial energy every clock cycle regardless of the RBM size while easily reaching and maintaining the peak computational bandwidth of $2(n-1)\cdot 32$-bits/cycle. This binary tree of adders effectively reduces a $O(n^2)$ time complexity to $O(n)$, while only requiring $O(n)$ resources. A circuit diagram of the pipelined, binary adder that is responsible for energy calculation, (Eqs. 3.8, 3.10), is described in Fig. 4.7.

This hardware implementation is effective since the hardware is identical for calculating both visible and hidden energies. By using the same hardware, this results in minimal resource utilization. The energy compute engine is capable of using the same hardware since the weights are stored in a manner that provides an entire row or column of the matrix.

#### 4.4.1.5 Weight Update Compute Engine

The weight update compute engine has two roles: to keep track of the weight update term for the entire batch as well as to commit and clear the weight update terms (Eq. 3.12). During

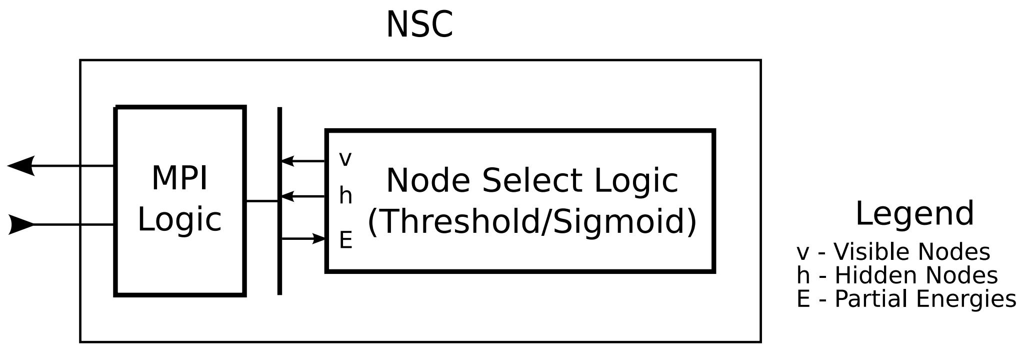

Figure 4.9: A block diagram indicating the data transfer of the NSC. The node select logic depends on whether the threshold or sigmoid probabilistic algorithm is used – their implementation is described in Sections 4.4.2.1 and 4.4.2.2, respectively.

the first and last AGS phases, the weight update compute engine reads both layer registers in parallel and accumulates the learning rates. When the entire batch is complete, the weight update accumulation is then committed to the weights. These operations only require AND-gates, multiplexers and fixed-point adder/subtractor units.

The low-level implementation is straightforward since each element of the weight matrices is independent. As a result, the output of each individual BRAM is sent to the arithmetic units and then immediately stored back into a corresponding BRAM. Since the memory is updated in parallel, the time complexity is reduced from $O(n^2)$ to $O(n)$, while only requiring $O(n)$ resources. Two circuit diagrams that are responsible for calculating the weight updates and committing the changes to the weight parameters, (Eq. 3.12), are described in Fig. 4.8.

### 4.4.2 Node Select Core

The Node Select Core (NSC) is a supporting compute core of the architecture. It is responsible for calculating the node states based on the partial energies (Eqs. 3.6, 3.7). Unlike the RBMC, which uses internal memory to alleviate bandwidth limitations, the NSC is designed to provide the maximum throughput given the limitations of the communication network – the communication network is limited to transferring a single 32-bit word per cycle and cannot be further parallelized. In addition, the software energy compute implementation was reduced to a time complexity of $O(n)$ (Table 2.1). Since the data can only be produced and transferred at at a rate of $O(n)$, it unnecessary to further decrease the complexity of the node selection computation.

However, the goal is to ensure maximum data throughput while minimizing resources. This is achieved by creating a streaming pipeline; this implementation does not require the storage of any data and ensures maximum data throughput.

Since the core has a single purpose and the data path has limited control flow, the NSC uses a finite state machine. A block diagram of core is shown in Fig. 4.9. The state machine will not be described, but a functional C implementation of the NSC is described in Appendix D.

#### 4.4.2.1 Threshold Node Select

The threshold select compute engine is easy to implement while still providing an acceptable transfer function. Because the partial energies are 32-bit fixed-point numbers, the node state is set as the inverse of the sign bit of the corresponding energy – due to the simplicity, a circuit diagram will not be presented. This implementation is able to select a single node state every clock cycle (Eqs. 2.6, 2.7).

#### 4.4.2.2 Sigmoidal Probabilistic Node Select

Finding a method to compute the sigmoid function, required in Eqs. 2.4, 2.5, has been a source of difficulty in hardware neural network design. The naive approach requires both exponential functions and division, two operations that would require significant hardware resources.

However, the sigmoid function is amenable for hardware implementations. First, the range of the function is bounded in the interval $(0, 1)$ – floating point representation is not required since the range is bounded. Also, the function has odd symmetry – a method to compute half of the domain is sufficient to generate the remainder of the domain.

There have been numerous studies on various hardware implementations of sigmoid functions [18–20]. However, the implementations were often designed for a different use case: the function was vastly replicated across the FPGA. As a result, it was designed for minimal resource utilization and low latency. Precision and bandwidth was not a priority.

A significantly different use case is present in the current framework. The RBMC is capable of providing one energy per clock cycle, which serializes the computation. As a result, maximizing bandwidth and the ability to select a node state every cycle is the highest priority. High latency due to deep pipelines is acceptable. Furthermore, since the NSC will not be vastly replicated; using more resources, including using one BRAM as a Look Up Table (LUT), is acceptable. Finally, high precision is desired.

**Piecewise Linear Interpolator** A BRAM LUT implementation is an efficient method to provide a reasonable approximation for bounded, transcendental functions. The results are precomputed and stored in a BRAM, where solutions are obtained in a single read. This is effective for application-specific architectures, which use a pre-defined set of functions. However, a BRAM LUT provides limited resolution. A 2kB BRAM with 32-bit (4-byte) outputs can only have 512 entries, meaning there is only 9-bit resolution for input values.

To increase the resolution, an interpolator was designed to operate on the two boundary outputs of a LUT. The implementation focused on the Linear Interpolator (LI), Eq. 4.1. The following notation will be used: the desired point $(u, v)$ exists between the end points $(x_0, y_0)$ and $(x_1, y_1)$.

$$v = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(u - x_0) + y_0 \tag{4.1}$$

The naive hardware implementation of Eq. 4.1 requires both division and multiplication; two operations which utilize significant resources. Instead, it should be noted that adding, subtracting, shifting, and comparing have efficient hardware implementations on FPGAs. Rather than calculating the interpolation exactly, a recursive piecewise implementation was designed. Knowing that the midpoint is found by adding the endpoints and a right shift by one, the search point is iteratively compared to the midpoints. This creates a piecewise approximation of a linear interpolator with little hardware overhead and is easily pipelined.

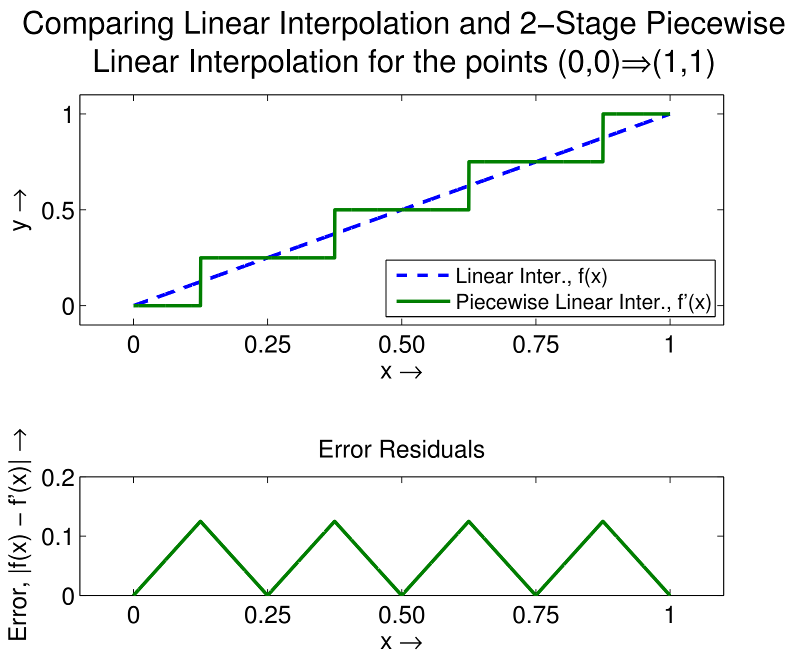

Figure 4.10: Comparison and error residuals of LI and PLI$^2$.

This hardware is called the $k$th Stage Piecewise Linear Interpolator (PLI$^k$), where each successive stage does one iteration of a binary search for the search point for one cycle of latency. A comparison of PLI$^2$ with a LI and the corresponding error is shown in Fig. 4.10, where $f(x)$ is the ideal function and $f'(x)$ is its approximation. A low-level schematic diagram of the PLI$^k$ design is shown in Fig. 4.11.

Comparing PLI$^k$ with LI, the error is a function of the number of stages and decreases geometrically. Thus, each PLI$^k$ will guarantee an additional bit of precision for every stage. The average and peak error are shown in Eqs. 4.2-4.3.

$$|v_{\text{LI}} - v_{\text{PLI}^k}|_{\text{average}} = \frac{y_1 - y_0}{2^{k+2}} \tag{4.2}$$

$$|v_{\text{LI}} - v_{\text{PLI}^k}|_{\text{peak}} = \frac{y_1 - y_0}{2^{k+1}} \tag{4.3}$$

It is important to note that the PLI$^k$ can be used on any LUT function implementation to increase the precision and is not limited to neural network architectures.

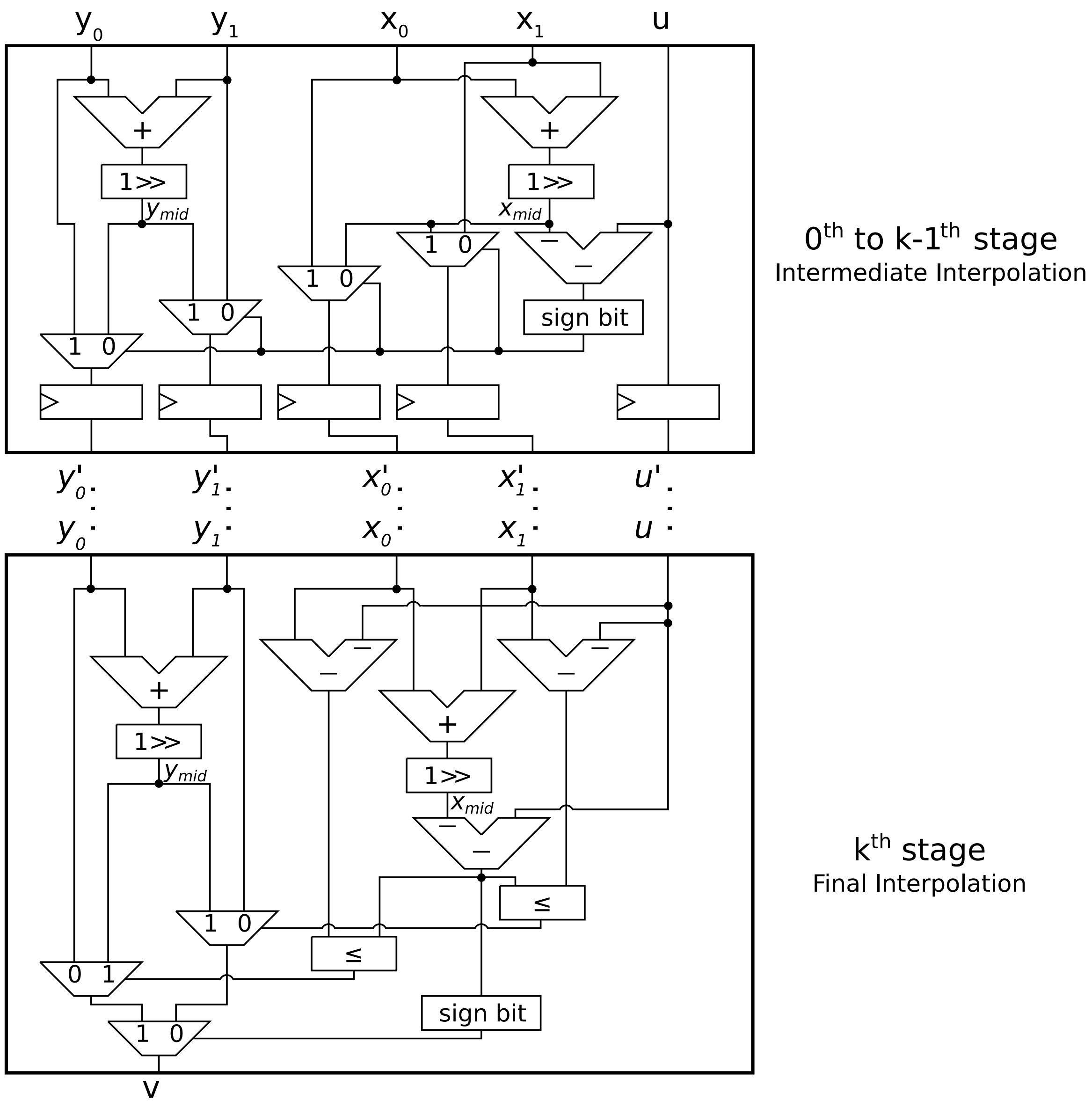

Figure 4.11: A schematic diagram of the PLI$^k$. The *intermediate* stages iteratively selects new end points from the current midpoints and end points based on the search value, $u$. The *final* stage selects an output, $v$, from the values $\{y_0, y_{\text{mid}}, y_1\}$ based on the distance between $u$ and $\{x_0, x_{\text{mid}}, x_1\}$.

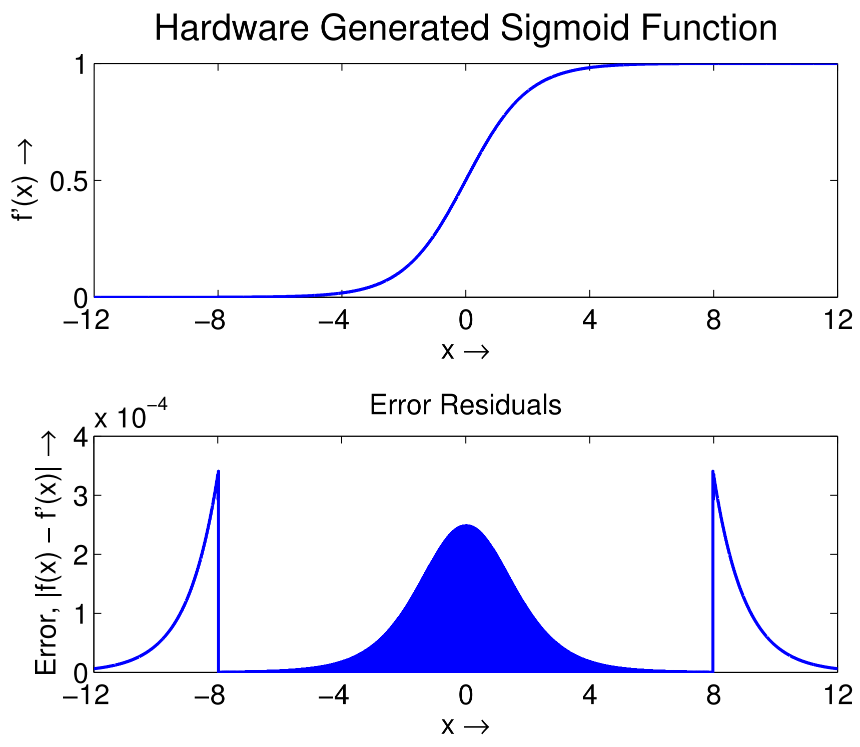

Figure 4.12: The reconstructed signal and error residues generated by the hardware for Eq. 4.4. The error residuals do not have a smooth curve since there are 512 points, which are exact results from the LUT.

**Sigmoid Node Selection** Using the BRAM LUT and PLI$^k$, a high-precision pipelined sigmoid transfer function was generated. Using fixed-point inputs, the sigmoid function is defined as a piecewise implementation (Eq. 4.4). This implementation takes advantage of the various favourable properties including odd symmetry and bounded range.

$$f'(x) = \begin{cases} 0 & , x \leq -8 \\ 1 - \text{PLI}^3(\text{LUT}(-x)) & , -8 < x \leq 0 \\ \text{PLI}^3(\text{LUT}(x)) & , 0 < x \leq 8 \\ 1 & , x > 8 \end{cases} \tag{4.4}$$

This implementation uses the bounded and odd symmetry properties of the sigmoid function to increase the LUT sampling frequency. For the outer limits of the domain, $x > 8$ or $x \leq -8$, the results are sufficiently close to the bounds of 1 and 0, respectively, with a max-

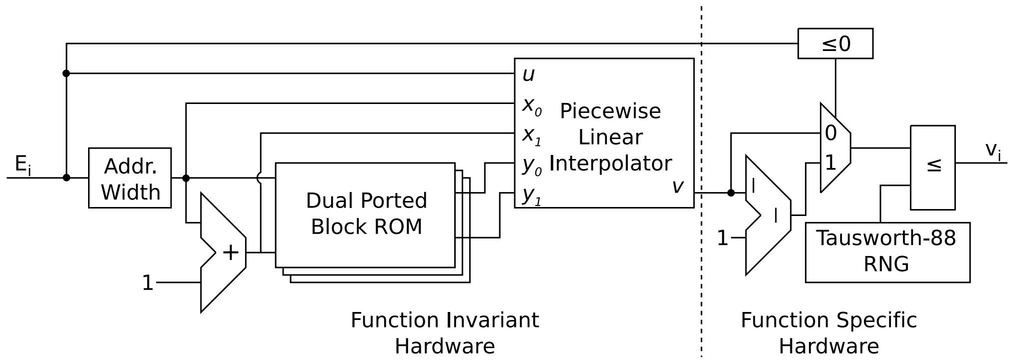
≤0
u
x0
x1
y0
y1
Piecewise Linear Interpolator
v
Ei
Addr. Width
Dual Ported Block ROM
+
1
0
1
≤
vi
Tausworth-88 RNG
Function Invariant Hardware
Function Specific Hardware

Figure 4.13: Block diagram of the stochastic node selection. Note the system is divided into a function invariant section and a sigmoid specific section. The logic for inputs $E_i > 8$ or $E_i \leq -8$ is not included.

imum error of 3.36E-4. Because the sigmoid function has odd symmetry, one dual-ported BRAM is used to store 512 evenly spaced points in the domain $0 < x \leq 8$. The dual-ported BRAM provides simultaneous access to the two nearest points. A PLI[3] is used to reduce the error such that the maximum error occurs at the $x = 8$ boundary. The average and peak error are for the sigmoid function in the domain $[-12, 12)$ are 4.82E-5 and 3.36E-4, respectively, with a precision of 11 bits (Fig. 4.12).

Finally, the result of the sigmoid function must be compared with a uniform random number to select the node state. There are many effective FPGA implementations of uniform random number generators. The Tausworth-88 random number generator was used because it generates high-quality random numbers with a cycle length of $2^{88}$, produces one result every clock cycle and requires little resource overhead [21]. The hardware implementation of the Tausworth-88 random number generator is described in Appendix E.

A complete block diagram of the stochastic node selection is presented in Fig. 4.13. The total latency for the hardware implementation is 8 clock cycles and, due to the pipelined design, is able to select a node every clock cycle.

### 4.4.3 Energy Accumulator Core

The Energy Accumulation Core (EAC) is a supporting compute engine of the architecture that provides the hardware required for RBM partitioning (Eqs. 3.9, 3.11). It receives the partial energies from multiple RBMCs and sums the energy vectors in an element-wise fashion. These energies are then transferred to the NSC. The NSC returns the node states, which are subsequently transferred to the RBMC. Like the NSC, this core is designed to provide maximum throughput given the limitations of the communication network and the sequential transfer of energies. As well, minimum resource utilization is a priority.

There are two distinct implementations of the EAC as a result of the different platforms. There is a *streaming* implementation designed for multi-FPGA architectures, which take advantage of hardware MPI communication to achieve significant throughput with very limited resources. There is also a *BRAM* implementation designed for virtualized architectures, which requires additional memory resources to store information to account for the context switches of the RBMC. Both implementations have a similar MPI communication protocol, providing a modular and reconfigurable architecture.

Since the core has a single purpose and the data path has limited control flow, the EAC uses a finite state machine. The state machine will not be described, but a functional C implementation of the BRAM implementation is described in Appendix F. The streaming implementation cannot be easily described in C due to its hardware design – however, from a top level perspective, the two cores are functionally identical but are able to achieve different levels of performance due to different message architectures.

#### 4.4.3.1 Streaming Implementation

The streaming EAC implementation is used for multi-FPGA platforms. It is able to minimize hardware utilization by taking advantage of the fine grain control provided by hardware MPI designs. At a lower level, the EAC begins by initiating messages with both the RBMCs and NSCs. Once each of the compute engines is ready to transmit energies and node states, the

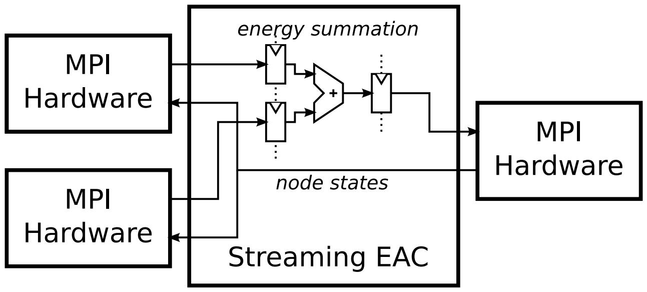

Figure 4.14: A circuit diagram of the arithmetic units in the streaming EAC implementation.

EAC then streams data bidirectionally through its compute engine using a pipelined datapath. The pipelined datapath accumulates one energy from each RBMC, sums the energies, and sends it to the NSC in each clock cycle. For the node states, the EAC retrieves the data from the NSC and forwards the same data to each of the RBMC. This implementation provides a low overhead communication protocol that provides significant performance benefits while also minimizing resources. A circuit diagram of the arithmetic units in the streaming implementation is presented in Fig. 4.14.

#### 4.4.3.2 RAM Implementation

The EAC RAM implementation is used for virtualizing the modules in the RBM architecture. Although the streaming implementation provides high performance and minimal resource utilization, it requires that the corresponding compute cores are active and ready for data transfer.

However, in virtualized platforms, there is only a single physical instantiation of the hardware that is being time multiplexed for use by many virtual instances. Thus, it is impossible to have all the other hardware cores ready. Instead, additional memory must be used to store intermediate results as each context switch occurs.

The EAC RAM utilizes a single FIFO to store both the energies and node states. Since the EAC RAM only needs to store the energy necessary to compute the corresponding node states, large memories are not required and a local BRAM provides sufficient resources. First, the EAC waits for messages containing partial energies. As the energies are received, each incoming value is summed with the next value in the FIFO and pushed to the back of the data

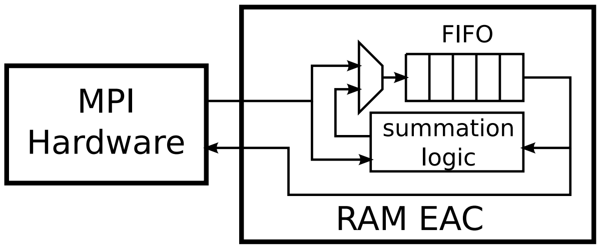

Figure 4.15: A circuit diagram of the RAM-based EAC implementation.

structure. This implementation allows a single hardware instantiation to be used for networks of any size. Since the energies are transferred sequentially, this maintains the same bandwidth. After the EAC has received all the messages, it EAC sends all of the currently summed energies to the corresponding NSC. As the node states are returned, the EAC forwards them back to the initial rank that originally all of the energies. A circuit diagram of the RAM-based EAC implementation is presented in Fig. 4.15.

# Chapter 5

# Methodology

## 5.1 Benchmarks

Unfortunately, there is a lack of a standardized benchmark for comparing FPGA implementations. The majority of hardware accelerated platforms are designed for a specific application in mind. As a result, an in-house application is often used as a point of comparison.

Since there are no widely available benchmarks, a custom software application is used here. Due to the research based nature of development, most neural network implementations are written in MATLAB. The MATLAB RBM algorithm in an available database [2] for a popular handwritten digit recognition RBM is used as the basis for a software benchmark written in C. The results of the benchmark are verified against the MATLAB implementation. Furthermore, since all the results used fixed-point representations, the hardware FPGA implementation produces the exact same results as the C software program.

## 5.2 Test and Verification Platforms

The benchmark is compiled with gcc version 4.3.3 with optimization level 2. An Intel Pentium 4 processor running Debian at 2.8GHz with 2GB of DDR RAM is the baseline machine. Cache optimization is not considered a significant factor since the entire program (data and instructions, combined) uses less than 150kB of memory – which fits in the 512kB L2 cache. In addition, gcc is unable to automatically vectorize the software implementation with SSE/SSE2 instructions using the *-msse2* flag. Hand-optimized vector operations could potentially lead to faster software implementations. However, this did not warrant further investigation since the maximum speed up of four-fold of the 128-bit vectors compared to the 32-bit scalar software implementation is considered insignificant.

The hardware implementation was tested on the Berkeley Emulation Engine 2 (BEE2) [22]. This high-performance system has five Virtex-II Pro XC2VP70 FPGAs connected in a communication mesh with 6-cycle latency and a bandwidth of 1.73GB/s between pairs of computing FPGAs. A hard PowerPC processor is responsible for retrieving the initial data and sending it to the hardware cores. The PowerPC is running at 300MHz while the hardware cores and the remainder of the FPGA logic is running at 100MHz.

Three different platforms were synthesized and tested: single FPGA, multi-FPGA and virtualized designs. RBMCs were synthesized with Xilinx Synthesis Tool (XST) with layer sizes of $n = \{32, 64, 128\}$ – the $32 \times 32$ RBM is considered the limit of efficient implementation and the size is increased in powers of two until the FPGA was resource limited.

The limiting factor in increasing the clock frequency was the routing congestion. XST reported the *fmax* of the RBMC, NSC, and EAC for the $128 \times 128$ RBM to be 143MHz, 110MHz, and 245MHz, respectively, for the XC2VP70 FPGA. However, the $128 \times 128$ RBM had timing closure difficulties. The timing reports indicate that the critical path is a result of the routing congestion due to long wire delays. Consequently, there was no additional effort to increase the clock frequency over the 100MHz goal.

To avoid overflow conditions, the software benchmark was used to determine the operating range for the magnitudes of the weights and energy values. This analysis suggested a 32-bit fixed-point representation with 1 sign bit, 8 integer bits and 23 fractional bits. However, the location of the radix point is parameterizable from the top-level specification and does not affect performance or resource utilization of the implementation.

For the single FPGA, an additional network size of $256 \times 256$ was simulated on a cycle-accurate, full-system behavioural simulation using ModelSim6.0. This was used to provide a theoretical limit to compare with the other platforms. For the multi-FPGA design, only the four FPGA implementation was synthesized with each individual RBMC using the same sizes as the single FPGA. Finally, for the virtualized designs, all three of the four FPGA implementations were virtualized on a single FPGA.

For the software program, the function *gettimeofday()* in the standard C *time.h* library is used to time stamp the software implementation at the beginning and end of every batch. The testbench was measured on an idle machine with no other user processes and the arithmetic mean of 10 runs was reported. For the hardware implementation, the PowerPC used the MPI function *MPI_TIME()* to time stamp every batch.

## 5.3 Metrics

In terms of the goals outlined in Section 4.1.1, scalability and performance are measured quantitatively. For scalability, the number of BRAMs, Flip-Flops (FFs), and Look-up Tables (LUTs) are recorded for each of the hardware cores and various network sizes.

For performance, the lack of a standard neural network metric raises some issues. An absolute measure of performance is desirable; however,there are no metrics that can account for the differences in neural network architectures. An effective metric for computational performance of a single type of neural network is the number of Connections Updates per Seconds (CUPS) – the rate at which a neural network can complete a weight update [5]. For a RBM, CUPS is defined as the number of weights, $n^2$, divided by the period for one complete AGS cycle, $T$ (Eq. 5.1).

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

For comparing two different implementations of the same architecture, the *update period* is a simple and effective metric. The update period is the time it takes for the implementation to complete a single batch of data. The *speed-up* will be measured by the ratio described in Eq. 5.2, where $S$ is the speed-up, and $T_{hw}$ and $T_{sw}$ are the update periods for the hardware and software implementations, respectively.

$$S = \frac{T_{sw}}{T_{hw}} \tag{5.2}$$

# Chapter 6

# Results and Analysis

## 6.1 Resource Utilization

Resource utilization is the primary metric to measure the scalability of the architecture. As outlined in Section 5.3, the amount of FFs, 4-input LUTs and BRAM resources of the three hardware cores are measured across a variety of configurations. It is important to note that only the RBMC's resource utilization is a function of network size; both the NSC and EAC use the same amount of resources regardless of the number of nodes in the network. The resource utilization of all three cores is summarized in Table 6.1.

It is clear that the RBMC requires the vast majority of the FPGA resources, while the NSC and EAC require negligible amounts even when synthesizing the more complex Sigmoid and RAM-based implementations, respectively. Next, the relative rates of growth for each resource for the RBMC indicates an important limitation – the BRAM utilization increases at a much faster rate than both the FFs and LUTs. Thus, the limiting factor for this architecture is the amount of available BRAMs.

With respect to the goals outlined in Section 4.1.1, the architecture achieves the desired $O(n)$ resource utilization. The RBMC's resource utilization scales linearly, while both the NSC's and EAC's utilization remains constant. This trend is shown in Fig. 6.1.

| Component | Comment | FFs | LUTs | BRAMs |
|---|---|---|---|---|
| RBMC | $n = 32$ | 6649 (10%) | 7408 (11%) | 66 (20%) |
| | $n = 64$ | 13005 (16%) | 14130 (21%) | 130 (39%) |
| | $n = 128$ | 25706 (38%) | 27911 (42%) | 258 (78%) |
| NSC | Threshold | 74 (0%) | 136 (0%) | 0 (0%) |
| | Sigmoid | 568 (0%) | 862 (1%) | 1 (0%) |
| EAC | Streaming | 40 (0%) | 140 (0%) | 0 (0%) |
| | RAM | 106 (0%) | 278 (0%) | 1 (0%) |

Table 6.1: The distribution of resources for various configurations of the architecture cores. The results of the synthesis report are listed with the percentage of the total Xilinx XC2VP70 FPGA occupied resources in parenthesis.

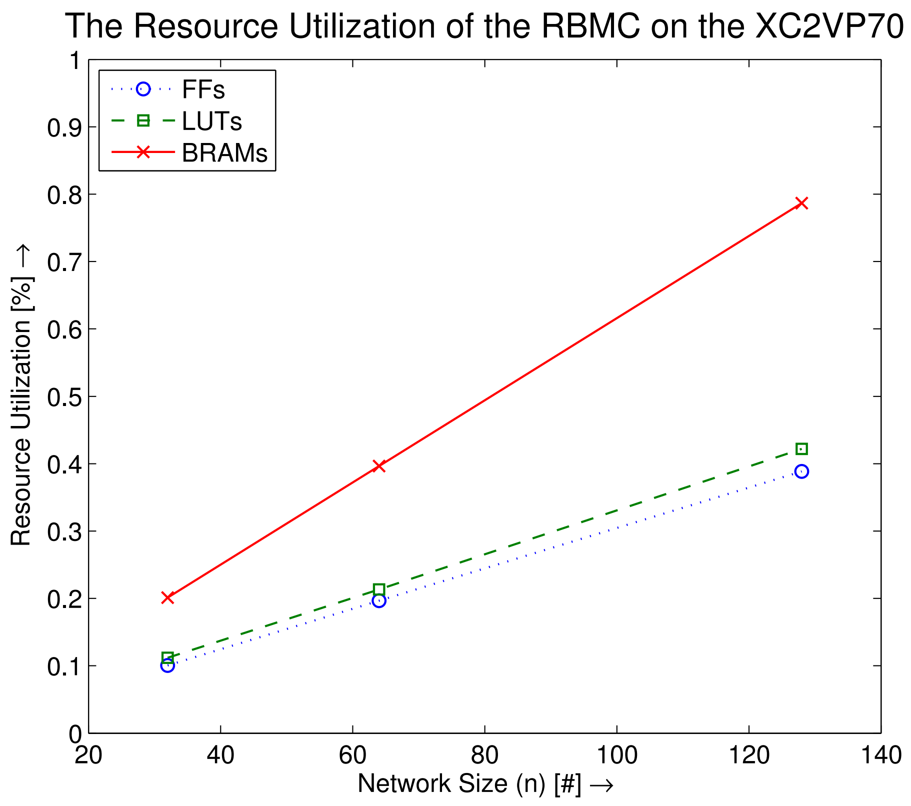

Figure 6.1: The resource utilization of the RBMC on a Xilinx XC2VP70 with respect to network size.

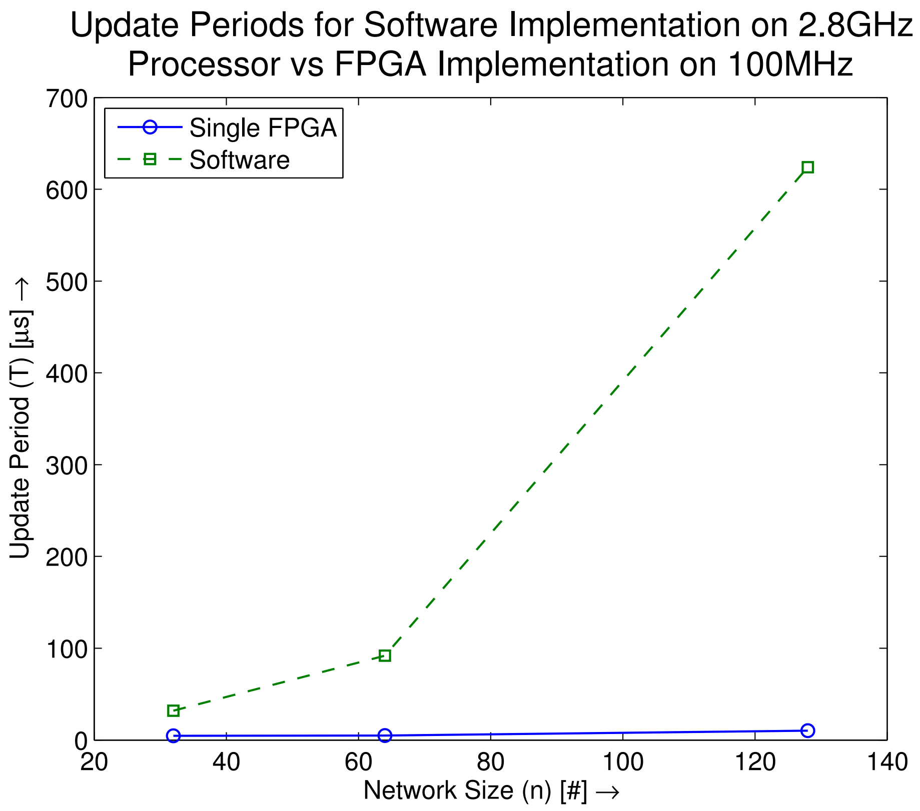

Figure 6.2: The update period for the single FPGA platform compared to the software implementation.

## 6.2 Single FPGA Implementation

The single FPGA implementation, as described in Section 4.3.2, is used as a baseline for the other platforms since it is the most rudimentary configuration. Thus, its performance with respect to the software counterpart is of primary interest. The update periods of each implementation for a variety of network sizes is shown in Fig. 6.2, and the relative speed up is shown in Fig 6.3. Fig 6.2 shows that the software implementation has the expected $O(n^2)$ complexity, while the hardware implementation has the desired $O(n)$ scaling. This results in a speed-up with a trend that scales with $O(n)$, seen in Fig. 6.3. The maximum computational throughput achieved with the single FPGA design is 1.58GCUPS for the $128 \times 128$ RBM network, resulting in a relative speed-up of 61-fold.

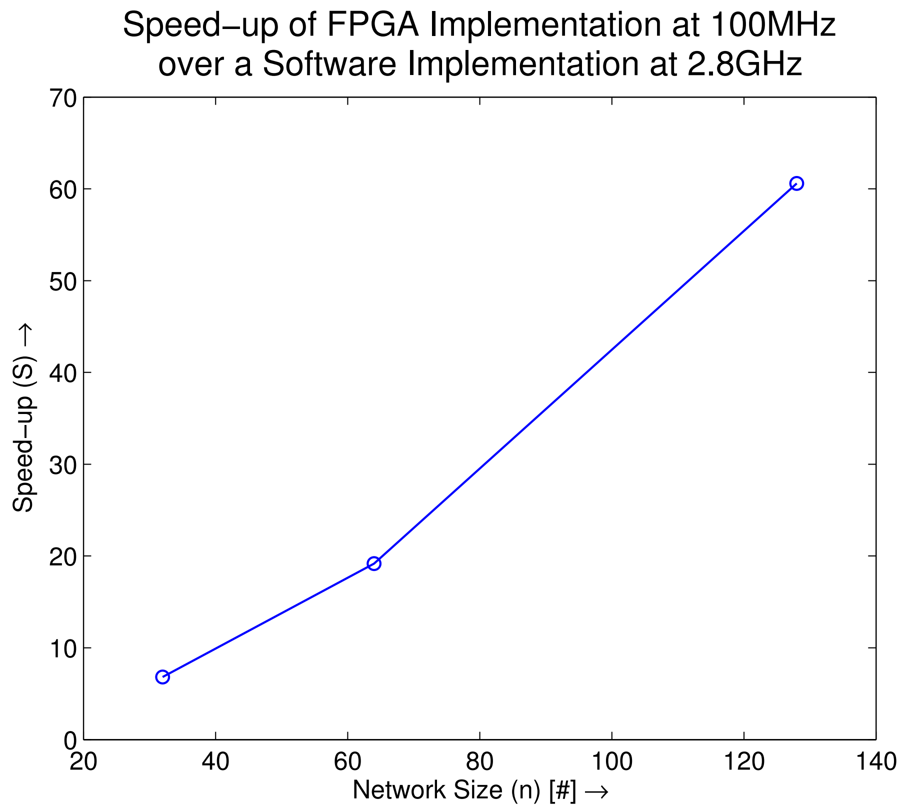

Figure 6.3: The speed-up for the single FPGA platform compared to the software implementation. The speed-up deviates from the expected $O(n)$ at low network sizes since the message communication between the hardware cores and embedded processor creates a non-linear overhead.

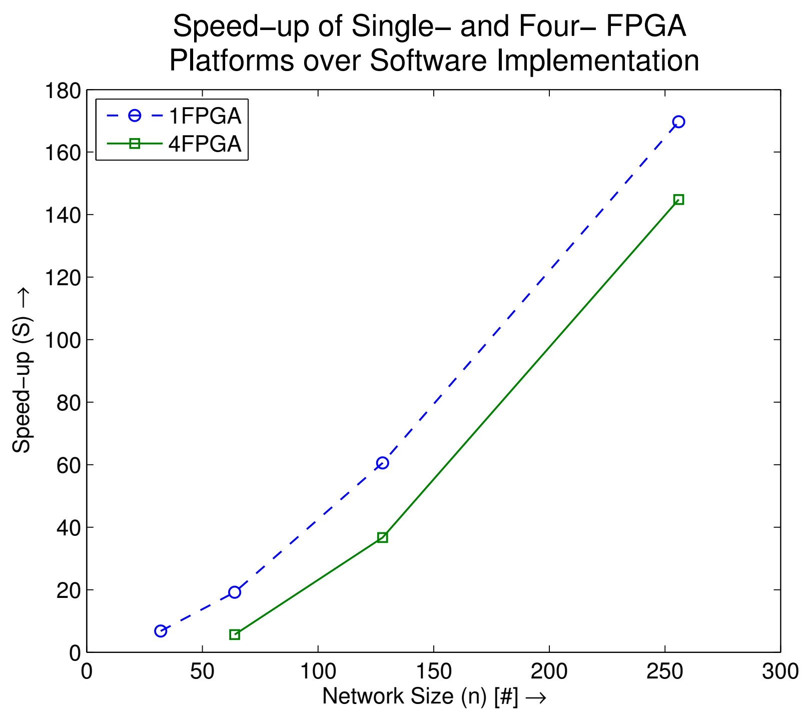

Figure 6.4: The speed-up for the single- and four- FPGA platforms over the software implementation. The result for the $256 \times 256$ single FPGA implementation was achieved through a cycle accurate simulation in ModelSim.

## 6.3 Multi-FPGA Implementation

Only the four FPGA platform, as described in Section 4.3.3.2, is reported since it can be readily compared with its single FPGA counterpart. The respective speed-ups of the single and four FPGA are shown in Fig 6.4. This platform provides sufficient coarse grain parallelism to achieve a maximum computational throughput of 3.13GCUPS using four $128 \times 128$ RBM networks, resulting in a relative speed-up of 145-fold over the software implementation. The architecture takes advantage of the data locality of the weights to obtain a high computation to communication ratio. In addition, the communication only requires the transfer of variables with $O(n)$ size, allowing for the performance of the multi-FPGA system to follow a similar trend to that of the single FPGA baseline. Many factors affect the overall performance – a more detailed discussion and breakdown of the time spent for communication and computation for this platform is outlined in [23].

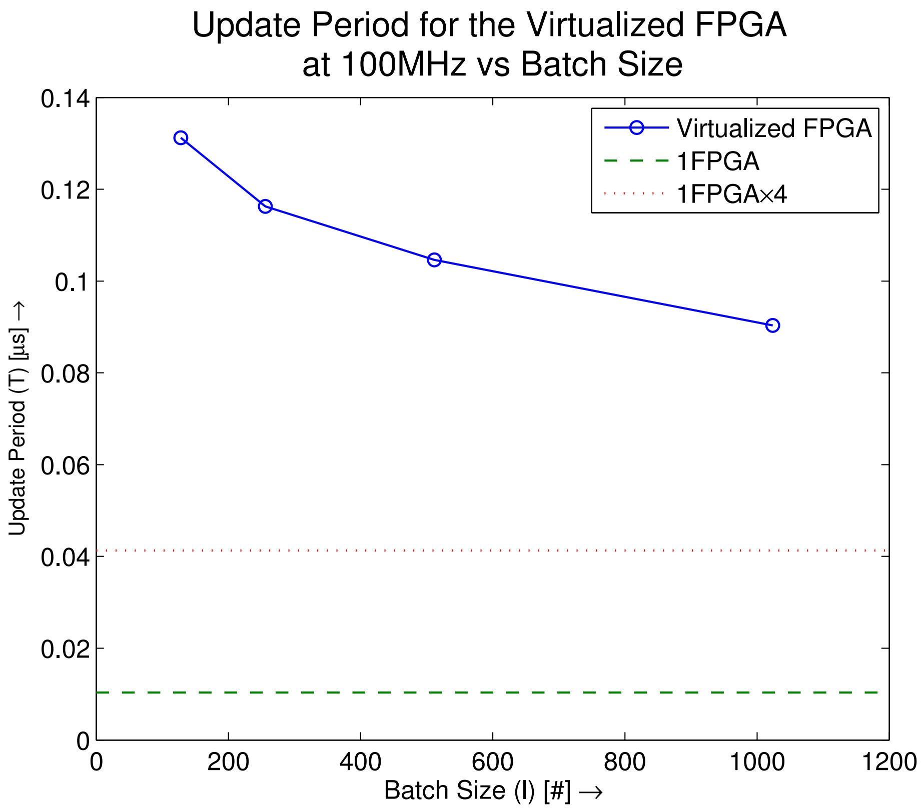

Figure 6.5: The update periods for the virtualized FPGA platform composed of $128 \times 128$ components for a variety of batch sizes. The single FPGA and the idealistic *1FPGA×4* marker performance is overlaid for reference.

## 6.4 Virtualization on a Single FPGA

The virtualized platform of the four FPGA system, as described in Section 4.3.4, is analyzed for its performance with respect to the other platforms. First, the overhead of weight swapping must be carefully quantified to understand the tradeoffs in using the virtualized system. A plot of the update periods with respect to the batch size for the virtualized platform is shown in Fig. 6.5. The *1FPGA×4* marker indicates the update period of a single FPGA multiplied by four, which results in the equivalent computation done by the virtualized platform. The marker will be used as an ideal baseline that only measures the required computation without any communication overhead. Thus, the difference between the virtualized machine and the *1FPGA×4* marker is the overhead due to context switching, the swapping of the weights and partition organization. Although the difference is considerable with over a 10-fold decrease in

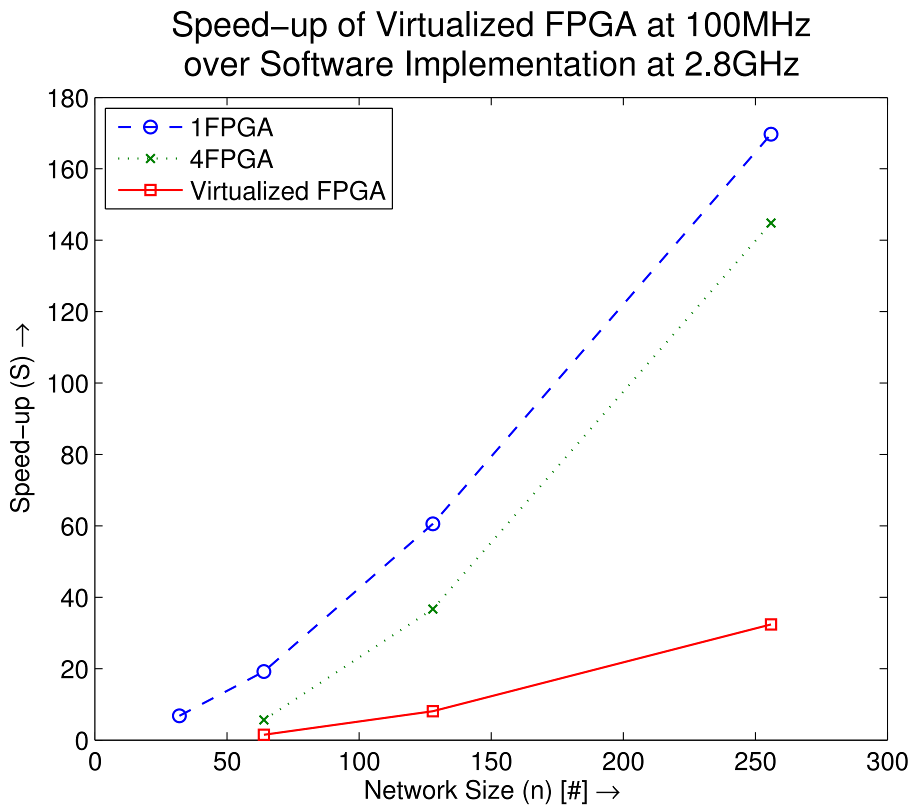

Figure 6.6: The speed-up for the virtualized FPGA platforms over to the software implementation. The single- and multi-FPGA results are overlaid for reference.

performance, it is important to note the effect of batch size on performance. Using the program outlined in Table 4.5, the weight swap can be amortized across the entire batch. As a result, the performance of the virtualized system approaches the ideal *1FPGA*$\times$*4* as the batch size goes to infinity. This is also advantageous in a machine learning aspect since large batch sizes result in better learning for the network.

Next, the performance of the virtualized system must be compared to the software benchmark. Although the virtualized system is considerably slower than its single-FPGA components, the overall performance is still impressive compared to the software implementation. The performance speed-up is a result of the $O(n^2)$ time complexity of the software – by increasing the network size, the software implementation decreases drastically, while the hardware implementation decreases marginally. Even with the weight swapping, by minimizing the memory transfer, a high performance system can still be obtained. The virtualized $256 \times 256$ system using a single $128 \times 128$ hardware core achieves a computational throughput of 725MCUPS, resulting in a speed-up of 32-fold.

Finally, there are several implementation details that restrict the performance of the virtualized platform and must be discussed. These implementation details are independent of the architecture and are a result of the underlying infrastructure. First, the hardware message passing implementation has a few limitations which result in unnecessary and slower transmission of messages. Using MPI implementations that support variable length messages and ready message protocols could drastically increase performance since the bottleneck is often the embedded processor to hardware interactions. These MPI upgrades are currently in progress. Next, the memory controller has a limited capability of maintaining peak bandwidth for a significant period of time. This problem is exacerbated when large memory swaps must occur and the effective memory bandwidth plays a critical role in the network performance. Designing or using a better memory controller that can support the access patterns and maintain a higher bandwidth would result in increased performance. As a result, the performance measurements of the virtualized system are conservative considering the limitations of the implementation and additional performance can be obtained by securing better infrastructure.

## 6.5 Platform Comparison

In the summary, the performance of the three platforms presented in this chapter is compared to the work described in Section 2.3. Platform descriptions and performance metrics are outlined in Table 6.2.

Unfortunately, as mentioned in Section 5.3, the lack of a universal metric for absolute performance makes direct comparisons between systems difficult. To obtain a reference point for comparison in [14], the publicly available MATLAB codebase provided by [1] was compared with the optimized C implementation. For network sizes of $128 \times 128$ and $256 \times 256$, the C implementation outperformed the MATLAB codebase by a factor of 20. It is important to note

| Implementation | Platform | Network Size | Clock Speed | Performance | | Baseline Platform |
|---|---|---|---|---|---|---|
| | | | | Absolute | Relative | |
| Single FPGA | 1 XC2VP70 | 128 × 128 | 100MHz | 1.58GCUPS | 61× | 2.8GHz P4; Optimized C software |
| Multi-FPGA | 4 XC2VP70 | 256 × 256 | 100MHz | 3.13GCUPS | 145× | 2.8GHz P4; Optimized C software |
| Virtualized FPGA | 1 XC2VP70 | 256 × 256 | 100MHz | 725MCUPS | 32× | 2.8GHz P4; Optimized C software |
| Kim FPGA [14] | 1 EP3SL340 | 256 × 256 | 200MHz | – | 25× Baseline | 2.4GHz Core2; MATLAB software |
| Raina GPU [15] | 1 GTX280 | 4096 × 11008 | 1.3GHz | – | 72× Baseline | 3.16GHz "Dual-core"; GOTO Blas software |

Table 6.2: Comparison of various RBM implementations. The relative performance results are measured against their respective baseline platforms and should not be compared directly.

that this MATLAB code was not further optimized and may not be exactly representative of the implementation used by [14]; however, the margin of difference is large enough to suggest that the C implementation significantly outperforms the MATLAB implementation. Determining a reference point for comparison in [15] was more difficult since the GOTO Blas implementation could not be easily obtained.

The proposed RBM architecture outperforms the FPGA implementation by Kim et al. [14] with a significantly smaller and slower FPGA, especially when the MATLAB reference point is taken into consideration. Both implementations support a similar sized network and the difference in performance is significant. Furthermore, their implementation uses the latest generation Altera FPGA; moving the proposed RBM architecture to the latest generation Xilinx Virtex-5 chip is expected to support a single FPGA of $512 \times 512$ with an expected speed-up of 600-fold.

Comparing with the GPU implementation by Raina et al. [15] provides less explicit conclusions. Since the GOTO Blas implementation could not be obtained, there is no direct comparison in performance. In addition, the GPU implementation provides support for significantly larger networks – however, this must be analyzed within the context of the "overlapping patches" technique. The large and deep networks supported by their implementation do not have fully connected layers, and thus, a significant portion of the data dependencies are removed by imposing an artificial simplification on the RBM. A CUPS measurement would be ideal since that would provide an accurate number of the computed connection updates as opposed to the inflated number of connections suggested by the network size.

Finally, the virtualized FPGA platform was designed to provide support for large networks.

However, due to the limited testing of this platform and the complex nature of virtualization, one must be cautious with providing predictions about the scalability and performance of this system. The effect of scaling the virtualized hardware on performance has not been thoroughly investigated. Although it may be safe to assume that the performance degrades when additional networks are virtualized, the rate at which the performance degrades is vital. This is further complicated by the $O(n^2)$ degradation of performance by the software implementation – accurately predicting the relative speed-up requires understanding how the two implementations perform with respect to each other. This study has been left as future work.

# Chapter 7

# Conclusions and Future Work

## 7.1 Conclusions

The goal of this thesis was to develop a high-performance, reconfigurable architecture for Field-Programmable Gate Arrays to drastically speed-up the performance of Restricted Boltzmann Machines. Deviating from the typical hardware neural network approach of using customized, superscalar pipelines, this architecture revolves around a novel method of partitioning large networks into smaller, congruent components. This divide-and-conquer method allowed the design of hardware cores that were able to better take advantage of the inherent parallelism in the neural network.

A series of hardware cores were proposed that implemented a specific subset of the AGS equations. This modular approach provided the versatility and reconfigurability required to implement a high-performance platform for a variety of network topologies. Three different platforms were used to illustrate the versatility of the design: single FPGA, multi-FPGA and virtualized platforms. In addition, the hardware implementation of each core was described in low-level detail. To achieve the desired performance, a number of low-level data structures and compute engines independent of the neural network framework were conceived, including a distributed BRAM matrix data structure and a piecewise linear interpolator.

The architecture was compared to an optimized software C implementation running on a 2.8GHz Intel Pentium 4 Processor. All of the proposed platforms outperformed their software counterpart, with the multi-FPGA design achieving the maximum performance of 3.13GCUPS resulting in a speed-up of 145-fold. These results indicated that a single FPGA obtains the best performance while the multi-FPGA platform provides additional coarse-grain parallelism with limited scalability. In comparison, the virtualized platform lacks the performance of the previous two, but is able to scale to larger networks with limited resources.

As a result, this thesis demonstrates that RBM neural networks can be effectively mapped to hardware designs on FPGA platforms. To achieve the high-performance system, a number of novel techniques and hardware structures were used, suggesting that the mapping from neural networks to hardware is non-trivial. Furthermore, an important aspect of the architecture is its lack of resemblance to the topological layout of the neural network. Instead, performance was obtained by analyzing the required computation, and then building hardware that takes advantage of the inherent parallelism through time multiplexing pipelined hardware. Thus, mapping neural network architectures to hardware platforms may require unintuitive and nontraditional approaches to achieve high-performance systems.

## 7.2 Future Work

There are a number of natural extensions of this research. The following is a short list of research topics that build upon the concepts presented in this thesis:

**Variable bit-width** Currently, the RBM architecture was designed to only use 32-bit energies, weight parameters and weight updates. This bit-width was chosen because it was a common data width size and preliminary research was only conducted on 32-bit fixed-point data. However, some research suggests that RBMs are capable of utilizing even smaller bit-widths while maintaining accurate results [14]. The consequences of this research are critical since using 16- or 8-bit fixed-point representations results in significant benefits: a smaller bit-width will allow an FPGA to instantiate a larger network with the same resources while the performance increases drastically since the larger network size exacerbates the $O(n)$ hardware speed-up.

**Real-valued visible node states** The current implementation uses only binary-valued node states. For the prototyped architecture, designing hardware for binary-valued node states was simple and efficient. Furthermore, the node states in DBNs, which consist of stacked RBMs, are by majority binary valued since only the bottom most visible layer can be real-valued. As a result, the current architecture provides support for the majority of the RBMs in a DBN.

However, using real-valued visible nodes is a primary interest since it provides a wider range of applications. Adapting the current architecture may require significant modifications. One of the main benefits of binary states is that simple flip-flops can be used to provide high-bandwidth parallel access to all of the node states. Real-valued node states require more memory, which may result in different memory data structures. Furthermore, simple hardware implementations such as logical AND operations cannot be used and full multipliers and DSP units will be required.

**Implementing a large RBM** The current virtualized platforms have only been tested for virtualized four-FPGA systems. The scalability of virtualization should be analyzed thoroughly. In particular, understanding the relationship between virtualizing a larger network and the relative speed-up compared to the software implementation is essential.

Implementing large RBMs can be further investigated by virtualizing multiple FPGAs to take advantage of the coarse-grain parallelism available with additional FPGA resources. This adds another degree of complexity to the analysis, but could result in even higher performance systems.

**Mapping hardware to other neural network architectures** Many neural networks share similarities in their architectures: most networks have a large matrix of weights and use complex, non-linear transfer functions. The proposed RBM architecture uses clever hardware designs to resolve these issues through the use of distributed memory and piecewise linear interpolation hardware. Furthermore, the architecture achieves its performance by serializing the computation for each node, allowing for deep pipelined hardware that provides a high computational bandwidth. This approach of focusing on the repeated calculation for each node as opposed to the raw parallelism may provide insight into designing high performance architectures for other neural networks.

# Appendices

# Appendix A

# MPI RBM Message Tags

MPI tags are used to label the contents of messages. For the RBM architecture, only the left-most byte of the possible 32-bits are used; the remainder of the bits are set to zero. A compliment of one-hot encoding is used for the tags, allowing for easy hardware encoding and decoding. The scheme is described in Fig. A.1 and the resulting tags are described in Table A.1.

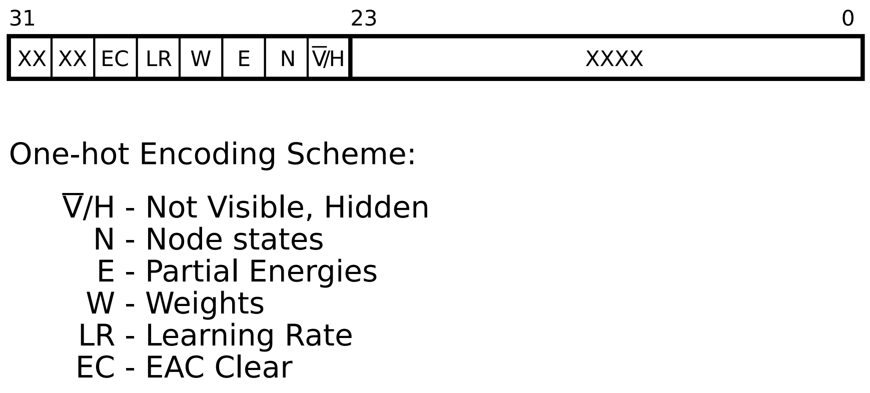

Figure A.1: The encoding scheme for MPI tags. The MPI implementation uses 32-bit tags but only the most significant 8-bits are used.

| Tag | Value | Comments |
|---|---|---|
| T_OPCODE | 0xFF00_0000 | Does not use encoding scheme; payload is a set of instructions |
| T_VIS_NODE | 0x0200_0000 | Payload: packed bit representation of the visible nodes ($\mathbf{v}$) |
| T_HID_NODE | 0x0300_0000 | Payload: packed bit representation of the hidden nodes ($\mathbf{h}$) |
| T_VIS_NRG | 0x0400_0000 | Payload: an ordered array of 32-bit visible energies ($\mathbf{E_v}$) |
| T_HID_NRG | 0x0500_0000 | Payload: an ordered array of 32-bit hidden energies ($\mathbf{E_h}$) |
| T_WEIGHT | 0x0800_0000 | Payload: an array of 32-bit weights using the distributed matrix order ($\mathbf{W}$) |
| T_WEIGHT | 0x1000_0000 | Payload: a single 32-bit learning rate in fixed-point representation ($\epsilon$) |
| T_EAC_VIS_EMPTY | 0x2000_0000 | Command message to notify RAM-based EAC to send partial energies as visible energies; the single word payload describes the number of energies to send |
| T_EAC_HID_EMPTY | 0x2100_0000 | Command message to notify RAM-based EAC to send partial energies as hidden energies; the single word payload describes the number of energies to send |

Table A.1: The list of MPI tags with their corresponding values and descriptions of the message payload.

# Appendix B

# RBMC Functional C Implementation

```
//Define fixed point values as int
#define fixed int

int extern rbmc_size, batch_size;
int opcode, target;
int temp[512];
int instruction_fifo[512];
int vis[rbmc_size], hid[rbmc_size];
int vis_encode[rbmc_size/32], hid_encode[rbmc_size/32];
fixed energy[rbmc_size];
fixed weight[rbmc_size][rbmc_size], weight_update[rbmc_size][rbmc_size];
fixed learning_rate;
MPI_Status s;

while(1)
{
    if (fifo_isempty(instruction_fifo))
    {
        MPI_Recv(temp, 512, MPI_INT, 0, T_OPCODE, MPI_COMM_WORLD, &s);
        push_all(instruction_fifo, temp, s.MPI_SIZE)
    }
    else
    {
        opcode = get_opcode(instruction_fifo);
        target = get_target(instruction_fifo);
        pop(instruction_fifo);

        switch (opcode)
        {
            case (OP_VIS_NODE_SEND) :
                bitwise_encode(vis, vis_encode, rbmc_size);
                MPI_Send(vis_encode, rbmc_size/32, MPI_INT, target, T_VIS_NODE, MPI_COMM_WORLD);
                break;
            case (OP_VIS_NODE_RECV) :
                MPI_Recv(vis_encode, rbmc_size/32, MPI_INT, target, T_VIS_NODE, MPI_COMM_WORLD, &s);
                bitwise_decode(vis, vis_encode, rbmc_size);
                break;
            case (OP_HID_NODE_SEND) :
                bitwise_encode(hid, hid_encode, rbmc_size);
                MPI_Send(hid_encode, rbmc_size/32, MPI_INT, target, T_HID_NODE, MPI_COMM_WORLD);
                break;
            case (OP_HID_NODE_RECV) :
                MPI_Recv(hid_encode, rbmc_size/32, MPI_INT, target, T_HID_NODE, MPI_COMM_WORLD, &s);
                bitwise_decode(hid, hid_encode, rbmc_size);
                break;
            case (OP_VIS_NRG_SEND) :
                MPI_Send(energy, rbmc_size, MPI_INT, target, T_VIS_NRG, MPI_COMM_WORLD);
                break;
            case (OP_HID_NRG_SEND) :
                MPI_Send(energy, rbmc_size, MPI_INT, target, T_HID_NRG, MPI_COMM_WORLD);
                break;
            case (OP_WEIGHT_SEND) :
                MPI_Send(weight, rbmc_size*rbmc_size, MPI_INT, target, T_WEIGHT, MPI_COMM_WORLD);
                break;
            case (OP_WEIGHT_RECV) :
                MPI_Recv(weight, rbmc_size*rbmc_size, MPI_INT, target, T_WEIGHT, MPI_COMM_WORLD, &s);
                break;
            case (OP_LEARNING_RATE_RECV) :
                MPI_Recv(learning_rate, 1, MPI_INT, target, T_LEARNING_RATE, MPI_COMM_WORLD, &s);
                break;
            case (OP_COMP_VIS_NRG) :
                for(int j = 0; j < rbmc_size; j++)
                {
                    energy[j] = 0;
                    for(int i = 0; i < rbmc_size; i++)
                        energy[j] += vis[i]*weight[i][j];
                }
                break;
            case (OP_COMP_HID_NRG) :
                for(int i = 0; i < rbmc_size; i++)
                {
                    energy[i] = 0;
                    for(int j = 0; j < rbmc_size; j++)
                        energy[i] += hid[j]*weight[i][j];
                }
                break;
            case (OP_POS_WUD) :
                for(int i = 0; i < rbmc_size; i++)
                    for(int j = 0; j < rbmc_size; j++)
                        weight_update[i][j] += learning_rate*vis[i]*hid[j];
                break;
            case (OP_NEG_WUD) :
                for(int i = 0; i < rbmc_size; i++)
                    for(int j = 0; j < rbmc_size; j++)
                        weight_update[i][j] -= learning_rate*vis[i]*hid[j];
                break;
            case (OP_COMMIT_WUD) :
                for(int i = 0; i < rbmc_size; i++)
                    for(int j = 0; j < rbmc_size; j++)
                    {
                        weight[i][j] += weight_update[i][j]/batch_size;
                        weight_update[i][j] = 0;
                    }
                break;
        }
    }
}

```

# Appendix C

# RBMC Instruction Set

The instruction set for the RBMC uses a 32-bit word and has a single type of word formatting: the left-most byte is used to represent a one-hot encoded opcode, the second byte is used to represent the target rank, and the remainder of the bits are ignored. The scheme is described in Fig. C.1, the notation is described in Table C.1 and the opcodes are described in Table C.2.

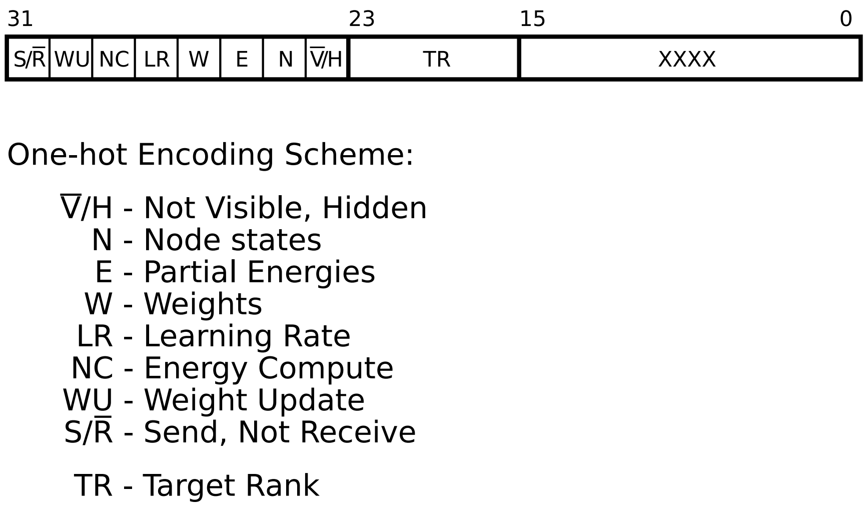

Figure C.1: The word formatting for RBMC instructions. The instructions uses 32-bit words but only the most significant two bytes are used.

| Notation | Description |
|---|---|
| $X \leftarrow Y$ | X is written with Y |
| $X + Y$ | X is added with Y in a scalar or vector format |
| $X - Y$ | Y is subtracted from X in a scalar or vector format |
| $X \cdot Y$ | X is multiplied with Y in a scalar format |
| $X \times Y$ | X is multiplied with Y in a vector format |
| [TR] | Target Rank |
| *array[i]* | The $i$th index of the variable array |
| *array[$*$]* | The variable array accessed in parallel |
| *array[$*$]*$^{\mathrm{T}}$ | The transposed variable array accessed in parallel |
| vis | The memory variable for the visible node states |
| hid | The memory variable for the hidden node states |
| nrg | The memory variable for the partial energies |
| lr | The memory variable for the learning rate |
| weight | The memory variable for the weight parameters |
| wud | The memory variable for the weight updates |
| Send(X, R, T) | MPI Send with message payload X, destination R, tag T |
| Rsend(X, R, T) | MPI Ready Send with message payload X, destination R, tag T |
| Recv(X, R, T) | MPI Recv with message payload X, source R, tag T |
| SIZE | Hardware parameter which describes the layer size |
| BATCH | Hardware parameter which describes the batch size |

Table C.1: A list of the controller notation with their corresponding descriptions.

| Opcode | Instruction Syntax | Command |
|---|---|---|
| **MPI Opcodes** | | |
| 0x82 | `vis_node_send [TR]` | Rsend(vis, TR, T_VIS_NODE) |
| 0x02 | `vis_node_recv [TR]` | Recv(vis, TR, T_VIS_NODE) |
| 0x83 | `hid_node_send [TR]` | Rsend(hid, TR, T_HID_NODE) |
| 0x03 | `hid_node_recv [TR]` | Recv(hid, TR, T_HID_NODE) |
| 0x84 | `vis_nrg_send [TR]` | Rsend(nrg, TR, T_VIS_NRG) |
| 0x85 | `hid_nrg_send [TR]` | Rsend(nrg, TR, T_HID_NRG) |
| 0x88 | `weight_send [TR]` | Send(weight, TR, T_WEIGHT) |
| 0x08 | `weight_recv [TR]` | Recv(weight, TR, T_WEIGHT) |
| 0x10 | `lr_recv [TR]` | Recv(lr, TR, T_LEARNING_RATE) |
| **Compute Opcodes** | | |
| 0x20 | `comp_vis_nrg` | for (i = 0; i < SIZE; i++) <br> $nrg[i] \leftarrow hid[*] \times weight[i][*]^{T}$ |
| 0x21 | `comp_hid_nrg` | for (i = 0; i < SIZE; i++) <br> $nrg[i] \leftarrow vis[*] \times weight[*][i]$ |
| 0x30 | `pos_wud` | for (i = 0; i < SIZE; i++) <br> $wud[*][i] \leftarrow wud[*][i] + lr \cdot vis[*]^{T} \times hid[i]$ |
| 0x31 | `neg_wud` | for (i = 0; i < SIZE; i++) <br> $wud[*][i] \leftarrow wud[*][i] - lr \cdot vis[*]^{T} \times hid[i]$ |
| 0x34 | `commit_wud` | for (i = 0; i < SIZE; i++) <br> $weight[*][i] \leftarrow weight[*][i] + wud[*][i]/BATCH$ |

Table C.2: A list of the controller opcodes, their instruction syntax and operation.

# Appendix D

# NSC Functional C Implementation

```
//Define fixed point values as int
#define fixed int

int extern rbmc_size;
type extern impl_style;
int target;
int node[32];
fixed energy[rbmc_size];
MPI_Status status;

while(1)
{
    MPI_Recv(energy, rbmc_size, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    target = status.MPI_SOURCE;

    for (int i = 0; i < rbmc_size/32; i++)
    {
        if (impl_style == threshold)
        {
            for (int j = 0; j < 32; j++)
                if (energy[i*32+j] > 0)
                    node[j] = 1;
                else
                    node[j] = 0;
        }


        else
        {
            for (int j = 0; j < 32; j++)
                if (rand() < 1/(1+exp(-(float)(energy[i*32+j])))
                    node[j] = 1;
                else
                    node[j] = 0;
        }

        if (status.MPI_TAG == T_VIS_NRG)
            MPI_Send(node, 32, MPI_INT, target, T_VIS_NODE, MPI_COMM_WORLD);
        else
            MPI_Send(node, 32, MPI_INT, target, T_HID_NODE, MPI_COMM_WORLD);
    }
}

```

# Appendix E

# Tausworth-88 Random Number Generator

The Tausworth-88 Random Number Generator is a high-quality uniform random number generator that produces 32-bit random numbers has a periodicity of $2^{88}$. A software C implementation is described in Fig. E.1 and the corresponding circuit diagram is described in Fig. E.2. Further details of the random number generator is explained in [21].

```c
unsigned int reg0, reg1, reg2, temp;

//Generates a floating point number between 0 and 1
float get_rand()
{
    temp = (((reg0 << 13) ^ reg0) >> 19);
    reg0 = (((reg0 & 0xFFFFFFFE) << 12) ^ temp);
    temp = (((reg1 <<  2) ^ reg1) >> 25);
    reg1 = (((reg1 & 0xFFFFFFF8) <<  4) ^ temp);
    temp = (((reg2 <<  3) ^ reg2) >> 11);
    reg2 = (((reg2 & 0xFFFFFFF0) << 17) ^ temp);

    return ((reg0 ^ reg1 ^ reg2) / MAX_INT);
}

```

Figure E.1: A C implementation of the Tausworth-88 random number generator

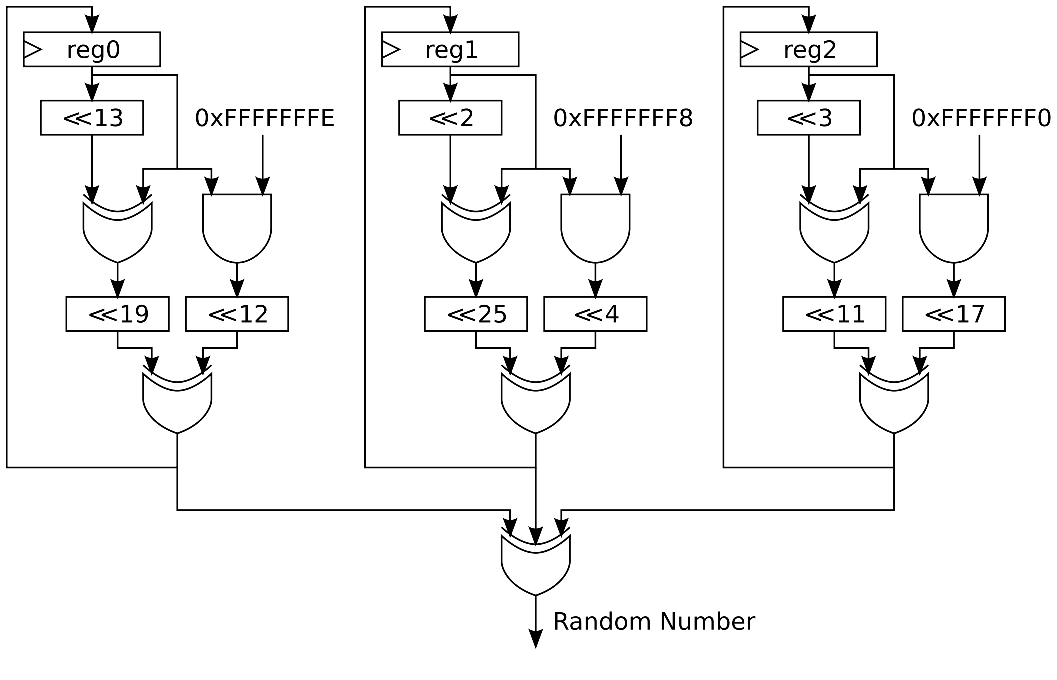

Figure E.2: A circuit diagram of the Tausworth-88 random number generator.

# Appendix F

# EAC BRAM Functional C Implementation

```c
//Define fixed point values as int
#define fixed int

int extern nsc_rank;
int target;
int size;
int temp[512];
int temp_nrg;
fixed fifo[512];
int fifo_isempty = 0;
MPI_Status status;

while(1)
{
    MPI_Recv(temp, 512, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    if ((status.MPI_TAG == T_VIS_NRG) || (status.MPI_TAG == T_HID_NRG))
    {
        if (fifo_isempty == 1)
        {
            push_all(fifo, temp, status.MPI_SIZE);
            fifo_isempty == 0;
        }

        else
            for (int i = 0; i < status.MPI_SIZE; i++)
            {
                temp_nrg = pop(fifo);
                temp_nrg = temp_nrg + temp[i];
                push(fifo, temp_nrg);
            }

    }
    else if (status.MPI_TAG == T_EAC_VIS_EMPTY)
    {
        size = temp[0];
        target = status.MPI_SOURCE;

        MPI_Send(fifo, size, MPI_INT, nsc_rank, T_VIS_NRG, MPI_COMM_WORLD);
        MPI_Recv(fifo, size/32, MPI_INT, nsc_rank, T_VIS_NODE, MPI_COMM_WORLD, &status);
        MPI_Send(fifo, size/32, MPI_INT, target, T_VIS_NODE, MPI_COMM_WORLD);
    }
    else if (status.MPI_TAG == T_EAC_HID_EMPTY)
    {
        size = temp[0];
        target = status.MPI_SOURCE;

        MPI_Send(fifo, size, MPI_INT, nsc_rank, T_HID_NRG, MPI_COMM_WORLD);
        MPI_Recv(fifo, size/32, MPI_INT, nsc_rank, T_HID_NODE, MPI_COMM_WORLD, &status);
        MPI_Send(fifo, size/32, MPI_INT, target, T_HID_NODE, MPI_COMM_WORLD);
    }
}

```

# Bibliography

[1] G. E. Hinton, S. Osindero, and Y. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

[2] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, vol. 313, pp. 504–507, July 2006.

[3] G. W. Taylor, G. E. Hinton, and S. T. Roweis, "Modeling Human Motion Using Binary Latent Variables," *Advances In Neural Information Processing Systems*, no. 19, pp. 1345–1352, 2007.

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

[5] Y. Liao, "Neural Networks in Hardware: A Survey," tech. rep., Santa Cruz, CA, USA, 2001.

[6] J. Zhu and P. Sutton, "FPGA Implementations of Neural Networks - A Survey of a Decade of Progress," *Lecture Notes in Computer Science*, no. 2778, pp. 1062–1066, 2003.

[7] P. Ferreira, P. Ribeiro, A. Antunes, and F. M. Dias, "A high bit resolution FPGA implementation of a FNN with a new algorithm for the activation function," *Neurocomputing*, vol. 71, pp. 71–77, 2007.

[8] D. Shen, L. Jin, and X. Ma, “FPGA Implementation of Feature Extraction and Neural Network Classifier for Handwritten Digit Recognition,” *Lecture notes in computer science*, vol. 3173, pp. 988–995, 2004.

[9] P. Smolensky, *Information processing in dynamical systems: Foundations of harmony theory*. Parallel Distributed Processing: Volume 1: Foundations, MIT Press, Cambridge, MA, 1986.

[10] Y. Freund and D. Haussler, “Unsupervised Learning of Distributions on Binary Vectors Using Two Layer Networks,” *Neural Information Processing Systems Conference (NIPS)*, pp. 912–919, 1992.

[11] D. Geman and S. Geman, “Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 6, no. 6, pp. 721–741, 1984.

[12] D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, “A Learning Algorithm for Boltzmann Machines,” *Cognitive Science*, vol. 9, pp. 147–169, 1985.

[13] G. E. Hinton and T. J. Sejnowski, *Learning and relearning in Boltzmann machines*. Parallel Distributed Processing: Volume 1: Foundations, MIT Press, Cambridge, MA, 1986.

[14] S. K. Kim, L. C. MacAfee, P. L. McMahon, and K. Olukotun, “A Highly Scalable Restricted Boltzmann Machine FPGA Implementation,” *International Conference on Field Programmable Logic and Applications*, 2009.

[15] R. Raina, A. Madhavan, and A. Y. Ng, “Large-Scale Deep Unsupervised Learning using Graphics Processors,” *International Conference on Machine Learning*, 2009.

[16] K. Goto and R. Van De Geijn, “High-performance implementation of the level-3 BLAS,” *ACM Transactions on Mathematical Software*, vol. 35, no. 1, pp. 1–14, 2008.

[17] M. Saldana, A. Patel, C. Madill, D. Nunes, A. Wang, A. Putnam, R. Wittig, and P. Chow, "MPI as an abstraction for software-hardware interaction for HPRCs," in *International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pp. 1–10, Nov. 2008.

[18] M. Tommiska, "Efficient digital implementation of the sigmoid function for reprogrammable logic," *IEE Proceedings – Computers and Digital Techniques*, pp. 403–411, 2003.

[19] A. Savich, M. Moussa, and S. Areibi, "The Impact of Arithemetic Representation on Implementing MLP-BP on FPGAs: A Study," *IEEE Transactions on Neural Networks*, vol. 18, no. 1, pp. 240–252, 2007.

[20] B. Bharkhada, J. Hauser, and C. Purdy, "Efficient FPGA implementation of a generic function approximator and its application to neural net computation," *IEEE International Symposium on Micro-NanoMechatronics and Human Science*, pp. 843–846, 2003.

[21] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203–213, 1996.

[22] C. Chang, J. Wawrzynek, and R. Brodersen, "BEE2: A High-End Reconfigurable Computing System," *IEEE Design & Test of Computers*, pp. 114–125, 2005.

[23] D. L. Ly, M. Saldana, and P. Chow, "The Challenges of Using An Embedded MPI for Hardware-based Processing Nodes," *International Conference on Field-Programmable Technology*, 2009. To Appear.