# A High-Performance FPGA Architecture for Restricted Boltzmann Machines

Daniel L. Ly and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
{lyd, pc}@eecg.toronto.edu

## ABSTRACT

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. Algorithms to implement a neural network in software are typically $O(n^2)$ problems – as a result, neural networks are unable to provide the performance and scalability required in non-academic settings.

In this paper, we investigate how FPGAs can be used to take advantage of the inherent parallelism in neural networks to provide a better implementation in terms of scalability and performance. We will focus on the Restricted Boltzmann machine, a popular type of neural network, because its architecture is particularly well-suited to hardware designs. The proposed, multi-purpose hardware framework is designed to reduce the $O(n^2)$ problem into an $O(n)$ implementation while only requiring $O(n)$ resources. The framework is tested on a Xilinx Virtex II-Pro XC2VP70 FPGA running at 100MHz. The resources support a Restricted Boltzmann machine of $128 \times 128$ nodes, which results in a computational speed of 1.02 billion connection-updates-per-second and a speed-up of 35 fold over an optimized C program running on a 2.8GHz Intel processor.

## Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Base Systems; I.5.5 [**Computing Methodologies**]: Pattern Recognition—*Implementation*

## General Terms

Design, Performance

## Keywords

Restricted Boltzmann machines, neural network hardware, FPGA, high-performance computing, scalable hardware designs, complexity reduction


*FPGA'09,* February 22–24, 2009, Monterey, California, USA.


## 1. INTRODUCTION

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. These unique properties have allowed them to be successfully applied to a wide variety of research areas ranging from recognizing hand-written digits [1] to reducing the dimensionality of data [2].

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. 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 being 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.

The balance of this section provides motivation for using FPGAs and related work. Section 2 gives some background on RBMs and Section 3 describes the RBM architecture being developed. The results and analysis of this work are presented in Section 4 and concluding remarks are given in Section 5.

### 1.1 Motivation for FPGA implementation

There have been many attempts to create hardware implementations to speed up the performance of neural networks [3], [4]. 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 reso-

lution, limited network size, and a difficult to use or non-existent software interface [5].

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 [6], [7]. 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 can be done with simple gates instead of 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 RAM, DSP units, 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 on a reconfigurable system allows hardware to be generated to suit the exact required topology thus, optimizing performance without sacrificing versatility.

# 2. RESTRICTED BOLTZMANN MACHINE THEORY

This section briefly describes the terminology, mathematical background and procedure involved in the mechanics of RBMs. Additional details, including the historical development and statistical motivation, can be found in [8], [9].

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.

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 original series of input vectors. Furthermore, the internal model allows the network to produce new data which is consistent with that distribution; this property is called *generative*.

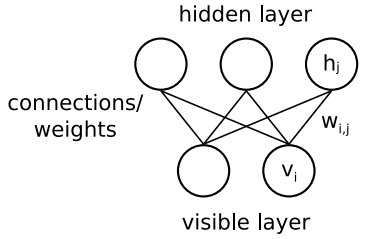


**Figure 1: A schematic diagram of a Restricted Boltzmann machine with labelled components.**

The RBM is *stochastic* because it uses a probabilistic approach to modeling 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, the RBM is capable of learning. The internal model is described mathematically by a multitude of independent parameters. 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*. 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 layer* and a *hidden layer*. The visible layer is used for input/output access while the hidden layer acts as an internal representation of the data for the network. 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 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. 1.

## 2.1 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 [10]. AGS is divided into two phases, the *generate* and *reconstruct* phases. During the generate phase, the visible layer is clamped to determine

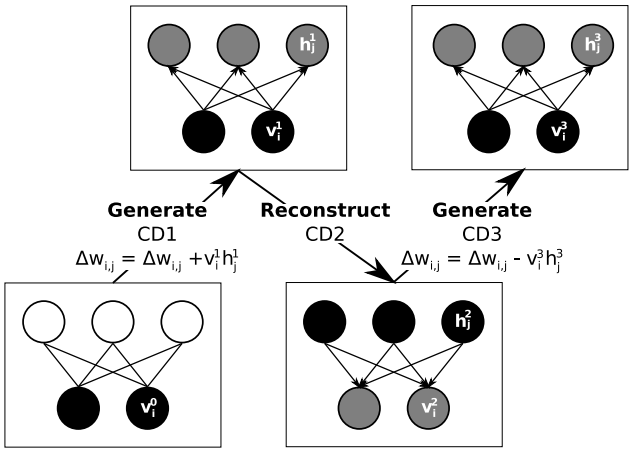


**Figure 2: A schematic diagram of the alternating Gibbs sampling for three phases. Uninitialized nodes are white, clamped nodes are black, and computed nodes are grey.**

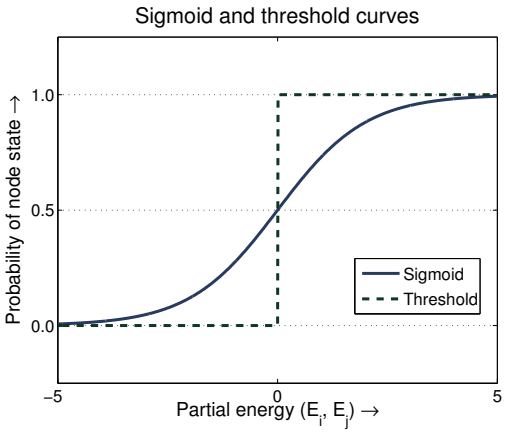


**Figure 3: A plot of a Sigmoid and Threshold function.**

the node states of the hidden layer, while in the reconstruction phase, the hidden layer is clamped to generate the node states of the visible layer. To begin the process, an initial data vector 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.

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 global energy, $E$, is defined in Eq. 1.

$$E = -\sum_{i,j} w_{i,j} v_i h_j \tag{1}$$

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. The generate and reconstruct phase use Eq. 2 and Eq. 3, respectively.

$$E = -\sum_i v_i \left( \sum_j w_{i,j} h_j \right) = -\sum_i v_i E_i \tag{2}$$

$$= -\sum_j h_j \left( \sum_i w_{i,j} v_i \right) = -\sum_j h_j E_j \tag{3}$$

The formulation of the partial energies indicates that the global energy can be determined using just the 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.

Using the statistical mechanics approach of defining probabilities with respect to energy functions, the node states have a cumulative distribution function of a logistic/sigmoid function. The probability of a node state turning on for a visible and hidden node is expressed in Eq. 4 and Eq. 5, respectively.

$$p(v_i = 1) = \frac{1}{1 + e^{-E_i}} \tag{4}$$

$$p(h_j = 1) = \frac{1}{1 + e^{-E_j}} \tag{5}$$

To determine the node state, a uniformly random variable must be sampled against the cumulative distribution function. Occasionally, the probabilistic approach might be undesirable. Instead, a deterministic, first-order approximation threshold function can be used (Eq. 6 and Eq. 7). By using the threshold function, the node states can be determined directly from the energies without any random processes. A comparison of the sigmoid distribution function and the deterministic threshold function are shown in Fig. 3.

$$v_i = \begin{cases} 0 & , \quad E_i < 0 \\ 1 & , \quad E_i \geq 0 \end{cases} \tag{6}$$

$$h_j = \begin{cases} 0 & , \quad E_j < 0 \\ 1 & , \quad E_j \geq 0 \end{cases} \tag{7}$$

## 2.2 Learning

One of the primary reasons for developing neural networks is their capability for machine learning, and as a result the learning rules for RBM are of great interest [11]. In review, the weight values are parameters used to 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. To find the minimum, the energy is differentiated with respect to the individual weights (Eq. 8).

$$\frac{\partial E}{\partial w_{i,j}} = \epsilon \left( \langle v_i h_j \rangle^1 - \langle v_i h_j \rangle^\infty \right) \tag{8}$$

In this notation, the $\langle \dots \rangle^X$ represents the expected values of the $X$th AGS phase for the entire training set and $\epsilon$ is the *learning rate* of the network.

The node states are generated through the iterative process of AGS, and as a result, the energy derivative shows

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 set.

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, for large batches, this will require a significant amount of time. As a result, the batch can be divided into smaller groups, called *mini-batches*, which allows the weight updates to occur with less data vectors. If the mini-batch only includes a single data vector, this is referred to as *on-line learning*.

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 replaced with a small finite number. 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).

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, it is important to note that 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* [12], [10].

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 Eq. 9-10, with $k$ batches of $L$ vectors each:

$$w_{i,j}[k+1] = w_{i,j}[k] - \epsilon\left(\langle v_i h_j\rangle^1 - \langle v_i h_j\rangle^X\right) \tag{9}$$

$$\langle v_i h_j\rangle^X = \frac{1}{L}\sum_{l=0}^{l} v_i^X h_j^X \tag{10}$$

## 2.3 Matrix notation

For ease of understanding and computation, Eq. 1-10 can be reformulated using matrix expressions. 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:

$$\mathbf{v}_l^X = [v_0^X \dots v_{i-1}^X] \in \mathbb{B}^{1\times i}$$
$$\mathbf{h}_l^X = [h_0^X \dots h_{j-1}^X] \in \mathbb{B}^{1\times j}$$

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

$$\mathbf{V}^X = \begin{bmatrix} \mathbf{v}_0^X \\ \vdots \\ \mathbf{v}_{L-1}^X \end{bmatrix} \in \mathbb{B}^{l\times i}, \mathbf{H}^X = \begin{bmatrix} \mathbf{h}_0^X \\ \vdots \\ \mathbf{h}_{L-1}^X \end{bmatrix} \in \mathbb{B}^{l\times j},$$

The weights can also be formulated as:

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

Then, the Eq. 1-10 can be reformulated as:

$$\mathbf{V}^{X+1} = \begin{cases} \mathbf{V}^0 & , \quad X = 0 \\ f(\mathbf{E}_\mathbf{v}^X) & , \quad X \text{ is odd} \\ \mathbf{V}^X & , \quad X \text{ is even} \end{cases} \tag{11}$$

$$\mathbf{H}^{X+1} = \begin{cases} f(\mathbf{E}_\mathbf{h}^X) & , \quad X \text{ is even} \\ \mathbf{H}^X & , \quad X \text{ is odd} \end{cases} \tag{12}$$

$$\mathbf{E}_\mathbf{v}^X = (\mathbf{H}^X)\mathbf{W}^\mathrm{T}, \in \mathbb{R}^{l\times i} \tag{13}$$

$$\mathbf{E}_\mathbf{h}^X = (\mathbf{V}^X)\mathbf{W}, \in \mathbb{R}^{l\times j} \tag{14}$$

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \frac{\epsilon}{l}\left((\mathbf{V}^1)^\mathrm{T}\mathbf{H}^1 + (\mathbf{V}^X)^\mathrm{T}(\mathbf{H}^X)\right) \tag{15}$$

Where $f(\cdot)$ is the transfer function applied element-wise to the matrix – it can either the sigmoid function random variable test (Eq. 4-5) or the threshold function (Eq. 6-7).

## 2.4 Complexity Analysis

To understand why sequential processors are not well suited for RBM implementations, the algorithm to implement Eq. 11-15 must be analyzed. A pseudocode sketch of the algorithm is summarized in Fig. 4.

```
for every batch :
    visible[] = get_datavector(batch)

    for every AGS_phase :
        if AGS_phase is odd :
            # Energy compute Eq.14 - 2 loops -> O(n^2)
            for every hidden_node :
                for every visible_node :
                    energy[j] += visible[i]*weight[i][j]

            # Node select Eq.12 - 1 loop -> O(n)
            for every hidden_node :
                hidden[j] = transfer_function(energy[j])

        else :
            # Energy compute Eq.13 - 2 loops -> O(n^2)
            for every visible_node :
                for every hidden_node :
                    energy[i] += hidden[j]*weight[i][j]

            # Node select Eq.11 - 1 loop -> O(n)
            for every visible_node :
                visible[i] = transfer_function(energy[i])

        # Weight update Eq.15 - 2 loops -> O(n^2)
        if ABS_phase == 1 :
            for every visible_node :
                for every hidden_node :
                    weight_update[i][j] += visible[i] * hidden[j]
        else if ABS_phase == ABS_limit :
            for every visible_node :
                for every hidden_node :
                    weight_update[i][j] -= visible[i] * hidden[j]

# Weight update Eq.15 - 2 loops -> O(n^2)
for every visible_node :
    for every hidden_node :
        weight[i][j] += learning_rate/batch * weight_update[i][j]
```

**Figure 4: A Python-like pseudocode sketch of the algorithm required to process a RBM**

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. 4, 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 (Eq. 11-12), energy compute (Eq. 13-14),

| Procedure | Complexity | Equation |
|---|---|---|
| Node select | $O(n)$ | 11, 12 |
| Energy compute | $O(n^2)$ | 13, 14 |
| Weight update | $O(n^2)$ | 15 |

**Table 1: The complexity analysis for each section of the RBM algorithm**

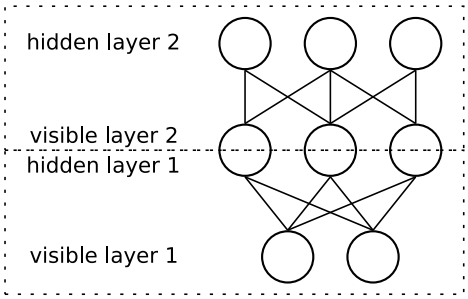


**Figure 5: A schematic diagram of a dual layered Restricted Boltzmann machine.**

and weight update (Eq. 15). The complexity of these individual sections is summarized in Table 1.

## 2.5 Layered networks

RBMs only have one layer of hidden nodes, and as a result, are only able to capture first order statistics. The underlying structure in a given set of data may require higher-order statistics for complete description, and thus, a single layer RBM will be insufficient. However, increasingly complex structures 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 to increase the modelling capabilities as long as the number of nodes match. This is illustrated schematically in Fig. 5.

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 extension. Further details are left to a more advanced paper on this matter [1].

# 3. RESTRICTED BOLTZMANN MACHINE FPGA ARCHITECTURE

Before the FPGA architecture is described, the two primary goals of the project will be outlined:

**Scalable** The FPGA implementation must be scalable with respect to both resources and performance - the design should fully utilize any hardware resources and the performance should benefit from additional resources. Specifically, an $O(n)$ resource utilization is desirable. The resources in an FPGA are growing fast and the implementation must be able to take advantage of the additional resources from new technology. Furthermore, there is an increasing trend for high-performance computing to include multiple FPGAs. In preparation for future systems, the implementation must be able to scale across multiple FPGAs as well.

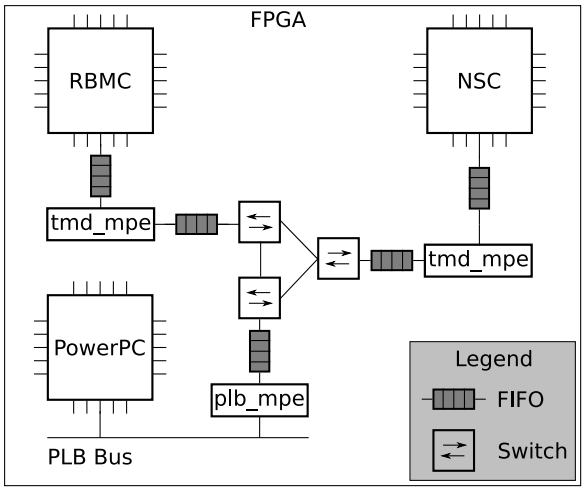


**Figure 6: A block diagram of a single FPGA implementation. The processor and two hardware cores are connected though various interfaces used to support TMD_MPI.**

**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 design.

These two goals drive the design decisions of the RBM architecture on an FPGA. In addition, since the RBM operation is well-defined, the computational hardware can be controlled through a finite state machine, removing the need for general-purpose processors of any complexity.

The FPGA implementation currently consists of two hardware cores: the *Restricted Boltzmann Machine Core* (RBMC) and the *Node Select Core* (NSC). The decision to divide the system into a number of hardware cores is motivated by the desire to increase multi-FPGA scalability – although the work presented here is developed on a single FPGA, the implementation framework is designed for large system scalability. The cores communicate with each other and the software front-end using the Message Passing Interface (MPI). An example of a single FPGA implementing a single RBM engine is shown in Fig. 6. It uses a hardware variant of the Message Passing Interface (MPI) called TMD-MPI [13].

The MPI communication network enables large system integration. MPI is a message-based communication specification that is used in a variety of high-performance applications. Of the many distributed communication protocols, it was decided that the MPI approach would be best suited for this implementation for several reasons. First, its widespread popularity in the scientific computing community means many of the target end-users will have some familiarity with this system. Next, since the data vectors will need to be retrieved from a shared file system or a sensor, a processor with any MPI implementation provides a software abstraction layer that allows for easy integration. Finally, the MPI protocol allows for the implementation to

be scaled across multiple FPGAs. By connecting the appropriate hardware cores, an extremely large RBM can be implemented on a collection of FPGAs.

The implementation uses two different hardware cores from TMD-MPI. The PowerPC processor uses a *plb_mpe*, which resides on the Processor Local Bus (PLB), while the two hardware cores use the *tmd_mpe*. These hardware interfaces allow the cores to communicate through a low-latency and high-bandwidth communication network with little resource and computational overhead.

In addition to understanding the hierarchical structure, it is also important to explain how the RBM operation (Eq. 11-15) can be accelerated at an architectural level. The most obvious approach would be to adapt some existing FPGA matrix accelerated hardware; however, the implementations often use some form of super-scalar pipeline to achieve a basic level of parallelism. Since the RBM operation is well defined and these cores can utilize all the FPGA resources, an architecture that takes more advantage of the parallelism can be designed. Thus, the basic principle behind the performance speed-up is to achieve a massively parallel vector-operator that can reduce the time complexity of the system from $O(n^2)$ to $O(n)$ by operating on an entire row of the weight matrix simultaneously. Furthermore, rather than attempting to operate on the entire batch as defined by Eq. 11-15, the FPGA implementation calculates a single data vector and stores its contribution in memory. As the implementation finishes iterating through the batch, the weight updates are committed to the weights and cleared for the next batch. This redefines the problem as vector-matrix calculations as opposed to the matrix-matrix formulations.

## 3.1 Restricted Boltzmann machine core

The Restricted Boltzmann Machine core (RBMC) is the primary computational core of the FPGA implementation. The RBMC is designed specifically to take advantage of the data locality of the weight – as a result, it is responsible for calculating partial energies and updating weights (Eq. 11-12 and 15). Recalling the complexity analysis (Table 1), these are the $O(n^2)$ 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 $O(n)$. This core itself is divided into three components: the memory organization, the energy compute engine and the weight update compute engine. A summary of how these cores are connected and what data needs to be transferred is described in Fig. 7.

### 3.1.1 Capabilities and Limitations

To achieve a significant speed-up, 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. 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.

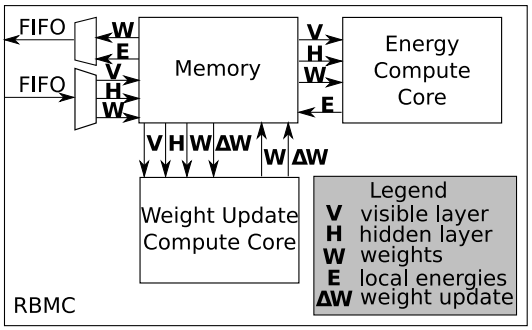


**Figure 7: A block diagram of how the RBMC is organized.**

- 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. 13-14). To understand how this limitation is acceptable, two cases must be analyzed: nearly symmetric RBMs and vastly asymmetric RBMs.
  - 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 zero.
  - For vastly asymmetric RBMs ($i \gg j$ or $i \ll j$), multiple RBMCs will be instantiated. First, it is important to remember that the RBMC achieves it performance by taking advantage of the locality of the weights. Weights are both plentiful and large – as a result, it is undesirable to transmit them since this requires $32n^2$ bits of total bandwidth. Instead, if the weights are kept in local memory, an identical RBM can be achieved by running two RBMCs in parallel and adding additional logic to ensure that the node states remain consistent. This method requires reconfiguration for different network topology and will be investigated in future work.
- 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 because the number of nodes have approximately this resolution and exact numbers are not mandatory.
- Batch sizes must be a power of two ($l = 2^b$). This is desirable because calculating the weight update accumulation (Eq. 10) 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.
- CD values 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 CD values without hardware updates.

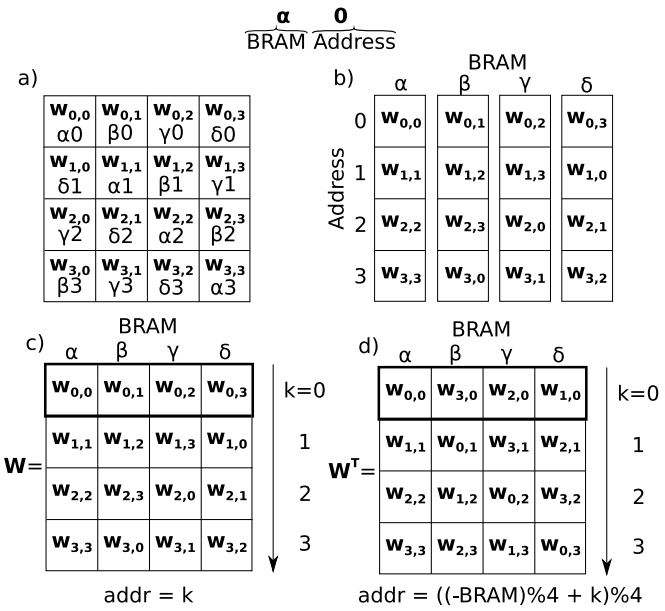


**Figure 8: The BRAM-based matrix transpose data structure for a $n = 4$ example. a) This is the organization of the weights in the standard matrix. The index under the weights corresponds to the BRAM and address of that weight. Notice how no BRAM has two elements on the same row or column. b) This is a reorganization of the weights in BRAM formation. c) A depiction of a row-wise access to the weights organized by the BRAM ports. $k$ is a counter indicating the row access. Notice how every row has the same elements as a row in the weight matrix. d), A depiction of a column-wise or (transpose row-wise access) to the weights organized by the BRAM ports. Notice how every row has the same elements as a column in the weight matrix. The formulation for the proper addresses are included.**

### 3.1.2 Memory core

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 205GB/s. As a result, the system 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. There are two sets of registers for the visible and hidden nodes to allow for parallel access to all the node states simultaneously. There are two sets of BRAMs: one for weight storage and one for weight update accumulation. All of the BRAMs are also dual-ported, allowing them to be written and read simultaneously as long as address conflicts are avoided. There is a single BRAM for storing the partial energies; only a single BRAM is required since the compute engine and the communication can only support serial data transfer. For the majority of the variables, the design and access to the memory hierarchy is straightforward.

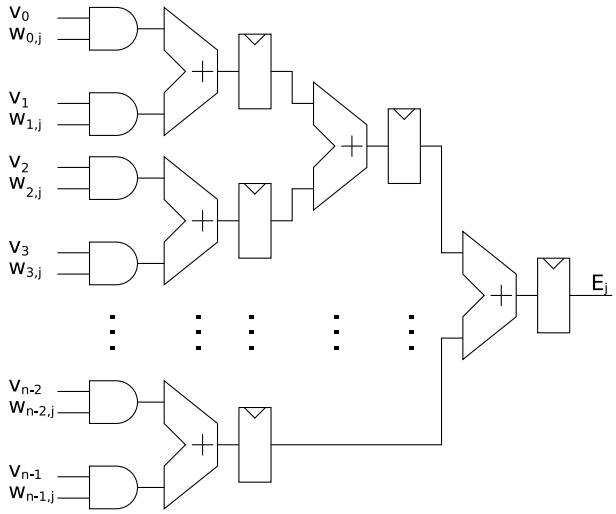


**Figure 9: A circuit diagram of the binary adder tree set to calculate the hidden partial energies (Eq. 13).**

However, 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. 13). 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. The actual operation of the transposing data structure is described in Fig. 8.

### 3.1.3 Energy compute engine

The energy compute engine is responsible for calculating the energies (Eq. 13-14). 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 3.1.1 and the binary states of the nodes, the computation can be done with simple hardware components: AND gates, multiplexers and registered, fixed-point adders. The 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, (Eq. 13-14), is described in Fig. 9.

### 3.1.4 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. 15). During 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 straight forward and a circuit diagram will not be presented. 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.

## 3.2 Node select core

The Node Select Core (NSC) is the secondary compute core of the system. It is responsible for calculating the node states based on the partial energies (Eq. 11-12). 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 $O(n)$. This coupled with the transmission limitations makes it difficult and unnecessary to further decrease the complexity.

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.

At the moment, only the threshold function has been implemented (Eq. 6-7). Although there is nothing prohibiting the design of a hardware sigmoid cumulative distribution function (Eq. 4-5), it would add unnecessary complexity to the prototype. However, there are plans to incorporate the sigmoid in the next phase of the project.

### 3.2.1 *Threshold select compute engine*

The threshold select compute engine is ideal for this prototype system since it 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.

# 4. RESULTS AND ANALYSIS

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. Due to the research based nature of development, most neural network implementations are written in MATLAB. The MATLAB RBM algorithm in an available database for a popular handwritten digit recognition RBM is used as the basis for a software benchmark written in C [2]. 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.

The benchmark is compiled with gcc version 4.3.1 with optimization level 2. An Intel Pentium 4 processor running

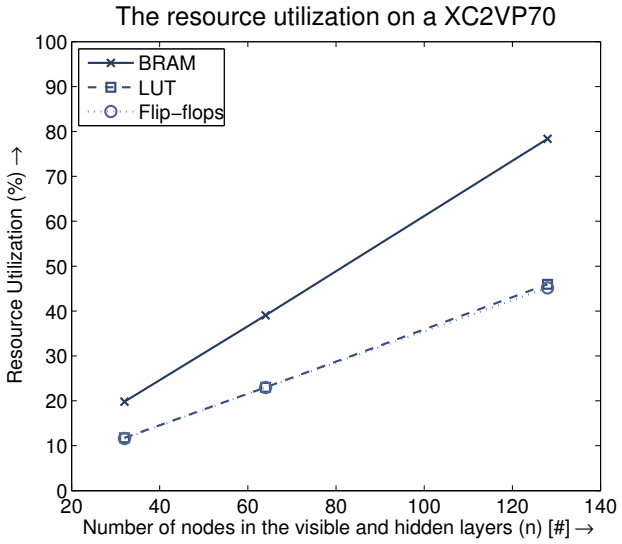


**Figure 10: The resource utilization with respect to the Xilinx Virtex II-Pro XC2VP70 FPGA.**

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 32-bit scalar software implementation is considered insignificant.

The hardware platform is implemented on a single Xilinx Virtex II-Pro XC2VP70 FPGA. The layout used in Fig. 6 is synthesized. The standalone 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 are running at 100MHz. Layer sizes of $n = \{32, 64, 128\}$ were synthesized – the $32 \times 32$ RBM is considered the limit of efficient implementation and the size is increased in powers of two until it is resource limited.

In terms of the goals outlined in Section 3, scalability and performance are measured quantitatively. For scalability, the number of BRAMs, Flip-Flops (FFs), and Look-up Tables (LUTs) are recorded. For performance, the lack of a standard neural network metric raises some issues. 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. 16, where $S$ is the speed-up, and $T_{hw}$ and $T_{sw}$ are the update periods for the hardware and software implementations, respectively.

$$S = \frac{T_{sw}}{T_{hw}} \tag{16}$$

An absolute measure of performance is also desirable. Although it cannot account for the differences in neural network architectures, a common metric for computational per-

| Component | Qty | FFs | LUTs | BRAMs |
|---|---|---|---|---|
| RBMC | 1 | 30403 (45%) | 29885 (45%) | 257 (78%) |
| NSC | 1 | 48 (0%) | 100 (0%) | 0 (0%) |
| FIFOs | 18 | 126 (0%) | 792 (1%) | 0 (0%) |
| switch | 3 | 387 (0%) | 1737 (3%) | 0 (0%) |
| tmd_mpe | 2 | 576 (0%) | 1724 (1%) | 4 (1%) |
| plb_mpe | 1 | 1071 (1%) | 2580 (3%) | 2 (0%) |

**Table 2: The distribution of resources for the $128 \times 128$ RBM on the XC2VP70 with the percentage of total FPGA utilization in brackets. The values represent the combined utilization of every instance. The table is divided into compute cores and network resources.**

formance is the number of Connections Updates per Seconds (CUPS) that can be computed [4] – described by Eq. 17, where $n$ is the node count and $T$ is the update period.

$$\text{CUPS} = \frac{n^2}{T} \tag{17}$$

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. For the hardware implementation, the PowerPC used the MPI function *MPI_TIME()* to time stamp every batch.

## 4.1 Scalability

In the implementation of this prototype, only a single FPGA design is developed and tested. Multi-FPGA scalability will be undertaken as future work.

The resource utilization with respect to the RBM layer size count is summarized in Fig. 10. All three components maintain the desirable $O(n)$ complexity, which is important for ensuring scalability for FPGAs with additional fabric. The distribution of resources for the $128 \times 128$ RBM is shown in Table 2. The majority of resource utilization is due to the RBMC – the NSC and the network infrastructure have a light weight design which adds a small resource overhead.

## 4.2 Communication to computation ratio

As with all parallel applications, the communication to computation ratio plays a critical role in the performance of the design. The distribution of communication and computation components for the $32 \times 32$ and $128 \times 128$ RBMs with an on-line, CD3 learning algorithm are shown in Fig. 11. The PowerPC to hardware communication creates a significant overhead – despite the limited data transfer, there is a large software overhead to prepare the MPI function calls.

Fig. 11 indicates that the MPI software communication overhead for the PowerPC is significant and reducing communication to computation ratio would result in greater speed-ups. However, the communication overhead is relatively constant with respect to node size and there are two methods that can be used to minimize the ratio. First, the number of CD learning cycles can be increased – for every data set, the time spent in hardware computation will increase compared to the software communication overhead. Increasing the batch size is the second method – the majority of the MPI function calls are only required at the beginning of each batch (denoted by the grey boxes in Fig. 11). Thus, larger batches will minimize the effect of the communication overhead required at the beginning of the batch.

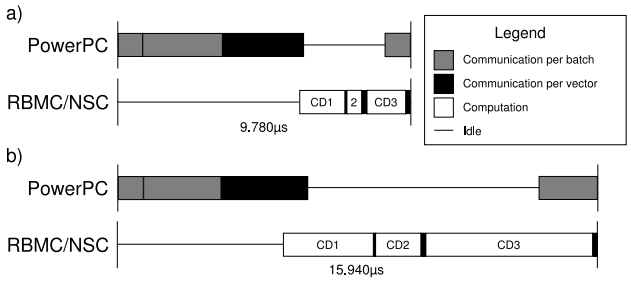


**Figure 11: The distribution of time used by the hardware implementation for communication and computation. Each axis represents the time elapsed during one update period with the total time recorded underneath. The results are obtained through a behavioural, hardware simulation. The a) and b) diagrams are for the $32 \times 32$ and the $128 \times 128$ RBMs, respectively.**

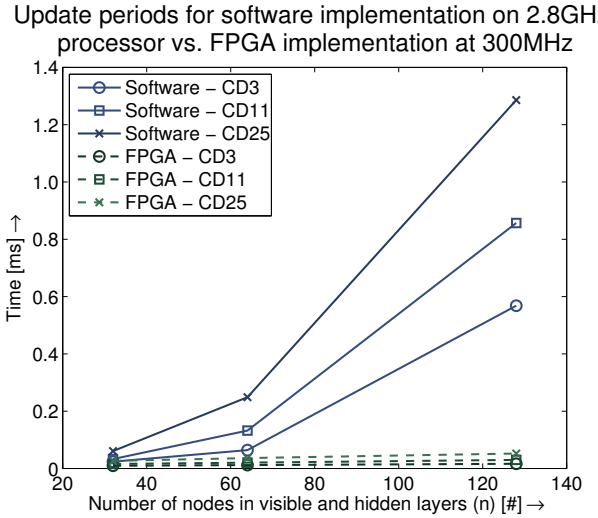


**Figure 12: The update periods for the FPGA and software implementations.**

## 4.3 Performance

The respective update periods and speed-up for a number of CD$X$ parameters are shown in Fig. 12 and 13, respectively. Fig 12 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 of $O(n^2)$, seen in Fig. 13. Specifically, the $128 \times 128$ RBM's performance is measured at 1.02GCUPS, resulting in a speed-up of 35 fold over the software benchmark.

Both of the scalability results have important implications about the future of this framework since newer generations of FPGAs will allow for larger RBM instantiations. Moore's Law states that the number of transistors on a chip doubles every 18 months. Current trends in processor design suggest that additional transistors will be used for additional cores – however, increased thread-level parallelism will not reduce the computation complexity. On the other hand, the additional transistors results in increased FPGA fabric: the $O(n)$ scalability can be utilized for each generation to take advan-

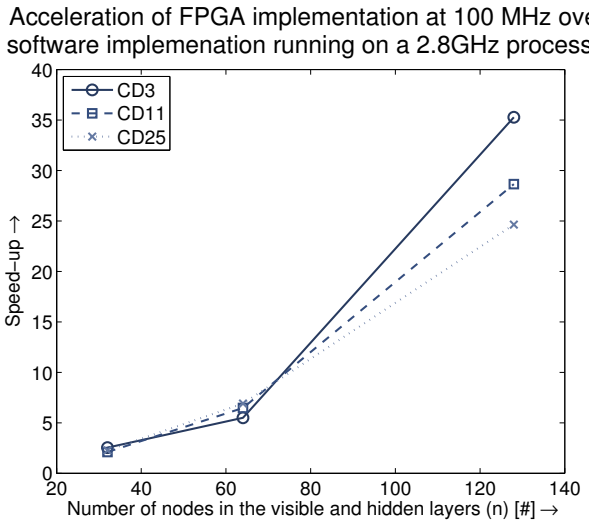


**Figure 13: The speed-up of the FPGA over software implementations.**

tage of the $O(n^2)$ acceleration. By measuring the software benchmark and extrapolating the hardware's linear trend in Fig. 12, a single Virtex 5 XC5VSX240T is expected to support a network of $512 \times 512$ nodes resulting in an estimated speed-up of 250x.

Although the hardware architecture will perform increasingly better with new generations, the current implementation on the two-generation old XC2VP70 can still be used in a number of applications. For example, the CD$X$ parameter has important implications on the efficiency of the learning algorithm. For higher CD$X$ values, the weight converges faster and less training is required. Although CD3 is functional, higher CD values are of special interest to researchers [14]. Typical software applications rarely go above CD25 due to the computation limitations. For the same run time as CD25 running on the software benchmark, the RBMC implementation can achieve CD800.

# 5. CONCLUSIONS

This paper shows that a high-performance, reconfigurable system can be designed to drastically speed-up the performance of Restricted Boltzmann machines. Deviating from the typical approach of most hardware neural network implementations, which consists of duplicating super-scalar, customized pipelines, this project focuses on creating a set of highly scalable compute engines that reduce an $O(n^2)$ problem on general-purpose processors into an $O(n)$ hardware implementation that scales with $O(n)$ resource utilization. On a single Xilinx Virtex II-Pro XC2VP70 running at 100MHz, a maximum performance of 1.02GCUPS is achieved for a network of $128 \times 128$ nodes, resulting in a speed-up of 35 fold over an optimized C benchmark on a 2.8GHz Intel Pentium 4 processor.

Although this work focuses on presenting a prototype on a single XC2VP70, the architecture is designed to allow for larger implementations to be scaled across multiple FPGAs. These initial building block cores are the first step towards building the world's largest and fastest RBM neurocomputer. Additional future avenues of research include how this architecture can be modified to provide the scalability and performance for a range of neural network systems.

# 6. ACKNOWLEDGEMENTS

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project. We also acknowledge Geoffrey Hinton, Graham Taylor, Arun Patel and Manuel Saldaña for their advice and feedback.

# 7. REFERENCES

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] Y. Freund and D. Haussler, "Unsupervised Learning of Distributions on Binary Vectors Using Two Layer Networks," *NIPS*, pp. 912–919, 1992.

[10] 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.

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

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

[13] M. Saldaña and P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors across Multiple FPGAs," *IEEE International Conference on Field-Programmable Logic and Applications (FPL 2006)*, pp. 329–334, 2006.

[14] M. A. Carreira-Perpiñán and G. E. Hinton, "On Contrastive Divergence Learning," *Artificial Intelligence and Statistics*, 2005.