IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 6, NO. 1, MARCH 2010

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

Daniel L. Ly, Student Member, IEEE, and Paul Chow, Senior Member, IEEE

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 because neural networks are usually implemented as software running on general-purpose processors. Hence, a hardware implementation that can exploit the inherent parallelism in neural networks is desired.

This paper investigates how the Restricted Boltzmann Machine, a popular type of neural network, can be 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.

Index Terms—Restricted Boltzmann Machines, Deep Belief Nets, Field-Programmable Gate Arrays, High Performance Computing, Neural Network Hardware

I. 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 capable of unsupervised learning and stochastic data generation. RBMs form the building blocks for the popular Deep Belief Nets (DBN), which have been applied to a wide variety of research areas including recognizing handwritten 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 since 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. Thus, 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) – a semiconductor device with programmable logic. By taking advantage of the inherent parallelism in neural networks, a high-performance system capable of applications beyond research and development can be realized. There have been numerous attempts to create hardware implementations to speed up the performance of neural networks [4], [5]. Despite the variety of approaches, from analog to VLSI systems, no hardware is widely used. These systems are typically plagued with a range of issues including limited resolution, small network sizes, and cumbersome software interfaces [6].

In addition, the neural network architecture also affects the capabilities of the hardware implementation – most architectures are not well suited for hardware systems. The common neural network architecture is the multilayer perceptron with back-propagation (MLP-BP) [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, accumulators and transcendental functions. As a result, each processing element requires significant resources, which restricts the scalability of the implementation. The common solution is to achieve parallelism by creating a customized pipeline similar to the super-scalar design used by processors. Unfortunately, these systems do not result in sufficient parallelism and performance to justify the cost and effort of using such hardware.

In comparison, RBMs are well-suited for hardware implementations. First, RBMs use data types that map well to hardware. The node states are binary-valued, which allow arithmetic operations, such as multiplication, to be completed with basic logic gates instead of resource intensive multipliers. Next, RBMs do not require high precision. Fixed-point arithmetic units can be used to reduce resource utilization and increase processing speed. Finally, RBMs have a high degree of data locality, which minimizes the overhead of transferring data and maximizes the computational throughput. The simplicity in the neural network architecture allows for clever hardware design, providing scalability and parallelism.

In particular, the reconfigurable aspect of FPGAs provides a distinct advantage over other hardware platforms for RBM
implementations. Since the arrangement of processing elements dictates the capabilities and behaviour of the network, being able to tailor the hardware to each arrangement is highly desirable. In contrast, Application Specific Integrated Circuit (ASIC) implementations must balance the trade-off between performance and versatility. Being able to design for a reconfigurable system allows hardware to be generated that suits the exact required topology.

This paper proposes a reconfigurable architecture with modular components for implementing high-performance RBMs in hardware. This paper builds on previous work ([9], [10]) and the primary contributions are as follows:

- A method to partition RBMs into congruent networks
- A collection of modular computational engines capable of implementing a wide variety of RBM topologies
- A method of virtualizing the RBM architecture to implement large networks with limited hardware

The remainder of the paper is organized as follows: Section II provides background and related work. Section III describes the partitioning method and Section IV outlines the hardware architecture. The evaluation methodology is presented in Section V and the results are discussed in Section VI. The conclusion and future work is discussed in Section VII.

II. BACKGROUND

A. Restricted Boltzmann Machines

A RBM is a generative, stochastic neural network architecture consisting of two layers of nodes representing visible and hidden variables. This work focuses on the family of RBMs where both the visible and hidden variables have binary states. There are weighted connections between every node in opposite layers, and no connections between any nodes in the same layer. Biases are represented by setting the first node.

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 \( i = \{1, \ldots, I\} \) and \( j = \{1, \ldots, J\} \), 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 Fig. 1.

Alternating Gibbs Sampling (AGS) and Contrastive-Divergence (CD) has been found to be an effective process to determine the node states and update the weight parameters [11], respectively. AGS is divided into two phases, the generate and reconstruct phases. During the generate phase, the visible layer is clamped and used to determine the node states of the hidden layer. In the reconstruction phase, the opposite occurs by clamping the hidden layer and reconstructing the visible nodes. 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. 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. This process is summarized in Fig. 2.

The RBM tunes the weights to minimize the global energy, \( E \), for a given data set, which is defined in Eq. 1.

\[
E = - \sum_{i=1}^{I} \sum_{j=1}^{J} w_{i,j} v_i h_j
\]  

(1)

Since connections only exist between nodes of opposite layers, the global energy can be redefined as a sum of partial energies, which depends on the AGS. The clamping of node states provides a localized computation. The generate and reconstruct phase use Eqs. 2 and 3, respectively.

\[
E = - \sum_{i=1}^{I} v_i \left( \sum_{j=1}^{J} w_{i,j} h_j \right) = - \sum_{i=1}^{I} v_i E_i
\]  

(2)

\[
E = - \sum_{j=1}^{J} h_j \left( \sum_{i=1}^{I} w_{i,j} v_i \right) = - \sum_{j=1}^{J} h_j E_j
\]  

(3)

The joint probabilities of the node configurations are defined using the Boltzmann distribution. Individual node state probabilities have a cumulative distribution function of a sigmoid function solely depending on the partial energy, expressed in Eqs. 4 and 5, for a visible and hidden node respectively. To determine the node state, a uniform random variable must be sampled against the cumulative distribution function.

\[
P(v_i = 1) = \frac{1}{1 + e^{-E_i}}
\]  

(4)

\[
P(h_j = 1) = \frac{1}{1 + e^{-E_j}}
\]  

(5)

The weight parameters are tuned using a method called Contrastive-Divergence (CD). Two pairs of AGS node states are used in CD learning – the first pair and an arbitrary, odd-numbered AGS phase. The notation CDX is used, where \( X \) is the arbitrary AGS limit (Fig. 2). Large limits provide better approximations to gradient descent but require more processing time. In addition, the training data vectors are often grouped into batches, allowing the weights to be updated over
the average of the inputted data. Large batch sizes provide smoother learning. Using a batch size of \( K \) and learning rate \( \epsilon \), the CD learning rules are described in Eqs. 6-7.

\[
\Delta w_{i,j} = \epsilon \left( \langle v_i h_j \rangle - \langle v_i h_j \rangle^X \right) \tag{6}
\]

\[
\langle v_i h_j \rangle^x = \frac{1}{K} \sum_{k=1}^K v_i^x h_j^x \tag{7}
\]

For simplicity, Eqs. 1-7 can be reformulated succinctly using matrix expressions that encapsulate the concept of layers and batches instead of individual scalar operations. For a RBM of \( I \) visible nodes and \( J \) hidden nodes, the visible layer, hidden layer and weights are represented respectively as:

\[
\begin{align*}
V^x & = \begin{bmatrix} v_1^x & \cdots & v_I^x \end{bmatrix} \in \mathbb{B}^{1 \times I} \\
H^x & = \begin{bmatrix} h_1^x & \cdots & h_J^x \end{bmatrix} \in \mathbb{B}^{1 \times J} \\
W & = \begin{bmatrix} \begin{bmatrix} w_{1,1} & \cdots & w_{1,J} \end{bmatrix} & \cdots & \begin{bmatrix} w_{I,1} & \cdots & w_{I,J} \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{I \times J}
\end{align*}
\]

The layers for the complete batch are represented as:

\[
\begin{align*}
V^x & = \begin{bmatrix} v_1 & \cdots & v_I \end{bmatrix} \in \mathbb{B}^{K \times I} \\
H^x & = \begin{bmatrix} h_1 & \cdots & h_J \end{bmatrix} \in \mathbb{B}^{K \times J}
\end{align*}
\]

Thus, the AGS Eqs. 1-7 can be reformulated as:

\[
\begin{align*}
V^x & = \begin{cases}
V^0, & x = 0 \\
f(\mathbf{E}^{x-1}_v), & x \text{ is even} \\
V^{x-1}, & x \text{ is odd}
\end{cases} \\
H^x & = \begin{cases}
f(\mathbf{E}^{x-1}_h), & x \text{ is odd} \\
H^{x-1}, & x \text{ is even}
\end{cases} \\
\mathbf{E}^v_x & = (\mathbf{H}^x)^T \mathbf{W}^T, \quad \in \mathbb{R}^{L \times I} \\
\mathbf{E}^h_x & = (\mathbf{V}^x)^T \mathbf{W}, \quad \in \mathbb{R}^{L \times J} \\
\Delta \mathbf{W} & = \frac{\epsilon}{K} \left( (\mathbf{V}^T) \mathbf{H}^x + (\mathbf{V}^T)^T (\mathbf{H}^x) \right)
\end{align*}
\]

Where \( f(\cdot) \) is the sigmoid function random variable test applied element-wise to the matrix (Eqs. 4-5).

B. Complexity Analysis

To understand why sequential processors are not well suited for RBM implementations, the algorithm to implement Eqs. 8-12 must be analyzed. A pseudocode sketch of the algorithm is summarized in Fig. 3.

1) Time complexity: Assuming the layers each have approximately \( n \) nodes, the time complexity of the algorithm is determined by simply tracing the loops. The algorithm is divided into three sections; node select (Eqs. 8-9), energy compute (Eqs. 10-11), and weight update (Eq. 12); and is summarized in Table I. The overall time complexity of the RBM algorithm is \( O(n^2) \), which illustrates the limited scalability of implementing RBMs on sequential processors.

2) Memory complexity: Analyzing Fig. 3, it is clear that only a handful of variables need to be stored. Using a bit for node states and a word size of \( w \)-bits for real numbers, the memory resources for each variable are summarized in Table II. It is important to note that each variable requires drastically different sizes and bandwidth.

\[
\text{Table I}
\]

\[
\text{Table II}
\]
provided by Hinton et al. [1] using a 2.4GHz Intel Core2 processor implemented using a single thread. For network sizes of $256 \times 256$, $512 \times 512$, and $256 \times 1024$, the maximum speed-up achieved was 25-fold compared to single precision MATLAB and 30-fold for double precision MATLAB.

Rainea et al. [13] accelerated RBMs using a Graphic Processing Unit (GPU). The implementation was written in CUDA and tested on an NVIDIA GeForce GTX 280. In addition to the typical graphic processing considerations, such as coalesced memory accesses and shared memory, performance acceleration was further advanced by introducing a technique called “overlapping patches”, which tile small localized RBMs. Each overlapping patch is independent, resulting in globally sparse networks with locally dense connections, greatly reducing the memory size and bandwidth requirements while providing scalability. Comparing with a Goto BLAS implementation [14] 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.

We will introduce a new architecture that uses specialized access patterns to the high-bandwidth, local resources on an FPGA to achieve improved performance. Furthermore, we will show a better partitioning of large RBMs that will allow scaling of these local resources.

III. PARTITIONING METHOD

A primary obstacle that RBM implementations must overcome is the rapid growth of memory resources required to store and transfer the weight parameters and weight updates, which grow at rate of $O(n^2)$ (Table II). Large off-chip memories are often used, resulting in a bandwidth-limited implementation and reduced scalability.

Instead, a novel divide-and-conquer method is proposed that partitions a large RBM into an equivalent collection of smaller but congruent networks. This technique allows any implementation to create small networks that do not exhaust low-latency memory resources and are better capable of exploiting the data locality of RBMs.

First, the visible and hidden layers are partitioned into disjoint sets; the divisions are arbitrary and do not require uniform spacing. Using a prefix notation, each layer can be represented as a collection of vectors:

$$
\begin{align*}
\mathbf{v} &= [\alpha_1 \mathbf{v} \quad \alpha_2 \mathbf{v} \quad \ldots \quad \alpha_A \mathbf{v}] \\
\mathbf{h} &= [\beta_1 \mathbf{h} \quad \beta_2 \mathbf{h} \quad \ldots \quad \beta_B \mathbf{h}]
\end{align*}
$$

Continuing this partitioning to the weight matrix:

$$
\mathbf{W} = 
\begin{bmatrix}
\alpha_1 \beta_1 \mathbf{w} & \ldots & \alpha_A \beta_1 \mathbf{w} \\
\vdots & \ddots & \vdots \\
\alpha_1 \beta_B \mathbf{w} & \ldots & \alpha_A \beta_B \mathbf{w}
\end{bmatrix}
$$

The partial energies become sums of the partitioned matrices:

$$
\begin{align*}
\mathbf{E}_h &= \mathbf{VW} = [\beta_1 \mathbf{E}_h \quad \beta_2 \mathbf{E}_h \quad \ldots \quad \beta_B \mathbf{E}_h] \\
\Rightarrow \beta_a \mathbf{E}_h &= \sum_{a=1}^{A} \alpha_a \beta_a \mathbf{E}_h = \sum_{a=1}^{A} \alpha_a \mathbf{v} \mathbf{w} \beta_a \mathbf{h} \\
\mathbf{E}_v &= \mathbf{HW}^T = [\alpha_1 \mathbf{E}_v \quad \alpha_2 \mathbf{E}_v \quad \ldots \quad \alpha_A \mathbf{E}_v] \\
\Rightarrow \alpha_a \mathbf{E}_v &= \sum_{b=1}^{B} \alpha_a \beta_b \mathbf{E}_v = \sum_{b=1}^{B} \beta_b \mathbf{h} \mathbf{w} \alpha_a \beta_B \mathbf{W}^T
\end{align*}
$$

Thus, for any partition pair ($\alpha_a$, $\beta_b$), the AGS equations (Eqs. 8-12) can be reformulated as a partitioned set:

$$
\begin{align*}
\alpha_a \mathbf{V}^x &= \begin{cases} 
\alpha_a \mathbf{V}^0, & x = 0 \\
\mathbf{f}(\alpha_a \mathbf{E}_v^{x-1}), & x \text{ is even} \\
\alpha_a \mathbf{V}^x, & x \text{ is odd}
\end{cases} \\
\beta_b \mathbf{H}^x &= \begin{cases} 
\beta_b \mathbf{H}^{x-1}, & x \text{ is even}
\end{cases}
\end{align*}
$$

$$
\begin{align*}
\alpha_a \beta_b \mathbf{E}_v^x &= \beta_b \mathbf{H}^x \alpha_a \beta_b \mathbf{W}^T \\
\alpha_a \mathbf{V}^x &= \sum_{b=1}^{B} \alpha_a \beta_b \mathbf{E}_v^x \\
\beta_b \mathbf{H}^x &= \sum_{a=1}^{A} \alpha_a \beta_b \mathbf{E}_h^x \\
\Delta \alpha_a \beta_b \mathbf{W} &= \frac{1}{\mathcal{R}} \left( (\alpha_a \mathbf{V}^1)^T \beta_b \mathbf{H}^{1} + (\alpha_a \mathbf{V}^X)^T \beta_b \mathbf{H}^{X} \right)
\end{align*}
$$

With the exception of Eqs. 18 and 20, the partitioned AGS equations are identical to the equations of a single RBM (Eqs. 8-12). This method allows a large RBM to be composed of localized, congruent networks for the cost of a small, single global computation for Eqs. 18 and 20. To illustrate this significance, the disproportionate balance of memory requirements must be considered (Table II). Each partition has a unique set of private weights parameters and updates, allowing the implementation to exploit the high degree of data locality. Only the partitioned energies and subsequent node states are transferred, both of which have a size of $O(n)$. The trade-off achieved by this partitioning method is advantageous to RBM implementations since the most resource intensive data is stored locally which limits the transferring of data, ensuring a low communication-to-computation ratio.

IV. FPGA ARCHITECTURE

A. MPI Infrastructure

The hardware architecture uses message passing as the underlying communication infrastructure between the modular components. An implementation of the Message Passing Interface (MPI) developed specifically for embedded FPGA designs, called TMD-MPI [15], is used to provide numerous features and benefits. The implementation is built directly from the FPGA fabric using minimal resources. Since the communication network is entirely on-chip, messages are delivered with latencies on the order of cycles while supporting data bandwidths of 32-bit words every cycle.

Furthermore, MPI provides a straightforward software-hardware interface. The message passing paradigm is widely used in high-performance computing and TMD-MPI extends this popular protocol to hardware; the hardware RBM implementation is controlled entirely with MPI software code, using messages to abstract the hardware compute engines as computational processes, called ranks. In addition to ease of use, this feature also provides portability and versatility since each compute engine is compartmentalized into message-passing modules, that can be inserted or removed based on available resources and desired functionality.
ware core extends the single FPGA platform to multiple FPGA platform, which shows how the streaming EAC hard-
symmetric RBM network with minimal communication. RBMC and NSC execute the required AGS equations for a
the weights. By only transferring energies and node states, the
has the appropriate node states for CD learning, it updates
RBMC for the next phase of the AGS cycle. Once the RBMC
computes the node states. The states are then sent back to the
the partial energies and sends them directly to the NSC, whic h
vector. After the RBMC receives the data, it begins to comput e
the RBMC with instructions, the learning rate and initial da ta
network (Fig. 4). To begin operation, the processor initializes
NSC cores are assembled to implement a symmetric RBM
FPGA system, which illustrates how the processor, RBMC and
and virtualized single FPGA.

Three platforms will be presented: a single FPGA, quad-FPGA
and various components of the partitioned AGS equations (Eqs. 15-
1) Single FPGA Platform: The first platform is the single
FPGA system, which illustrates how the processor, RBMC and
NSC cores are assembled to implement a symmetric RBM
network (Fig. 4). To begin operation, the processor initializes
the RBMC with instructions, the learning rate and initial data
vector. After the RBMC receives the data, it begins to compute
the partial energies and 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. Once the RBMC
has the appropriate node states for CD learning, it updates
the weights. By only transferring energies and node states, the
RBMC and NSC execute the required AGS equations for a
symmetric RBM network with minimal communication.

2) Quad-FPGA Platform: The second platform is the quad-
FPGA platform, which shows how the streaming EAC hard-
ware core extends the single FPGA platform to multiple
FPGAs, allowing for larger networks with comparable scal-
ability (Fig. 5). Multiple instances of the hardware cores are
distributed amongst the numerous FPGAs. The partitioning
method is used to amalgamate these smaller cores to behave
as a single, larger RBM with coarse grain parallelism.

The example implements a large, symmetric RBM com-
posed of smaller networks – it should be noted that the
symmetric network was chosen for illustrative purposes and
the partitioning method can implement any arbitrary network.
The process is similar to the single FPGA platform: each
RBMC receives its instruction and initial node states from the
processor. The RBMCs calculate their partitioned partial e n-
ergies concurrently. To determine the node states, the RBMCs
send their respective energies to the EAC, which sums the
energy vectors to obtain the partial energies required for node
selection and is transferred to the NSCs (refer to [10] for
additional details). The NSC determine each partition of node
states, and is sent back to both RBMC via the EAC, ensuring
consistent values.

The scalability of this platform is limited since the cumu-
lative resources required to store the respective weight matrix
grows with a rate of $O(n^2)$, while additional FPGAs only
provides a constant-size, or $O(1)$, increase in resources. This
platform should be used to achieve additional performance
through coarse grain parallelism by adding FPGAs to the
design rather than building very large RBMs.

3) Virtualized FPGA Platform: The final platform is the
virtualized, single FPGA platform, which illustrates how the
partitioning method can be efficiently used to implement large
networks with a single FPGA. The hardware cores are time
multiplexed, allowing multiple RBMs to be computed with a
single set of hardware (Fig. 6).

This platform requires EAC RAM implementation, which
unlike its streaming counterpart, stores data during the context
switches. In this example, the hardware can implement a RBM of any size by partitioning it into sufficiently small networks. Since the virtualization requires context switches, which includes the swapping of the $O(n^2)$ weight matrix, the data locality of the weights is exploited by computing every data vector in a batch corresponding to a single set of weights before switching. Thus, larger batch sizes will increase performance since a single weight swap can be amortized across more computations.

This virtualization technique can be extended to a multi-FPGA system for additional performance. The coarse grain parallelism is maintained as all the FPGAs can switch context independently. Further exploration into virtualized, multi-FPGAs systems is left for future work.

D. Hardware Core Implementation

This section will outline the implementation details for each hardware core, highlighting the architectural features that provide fine-grain parallelism and high performance.

1) Restricted Boltzmann Machine Core: The Restricted Boltzmann Machine Core (RBMC) is the primary computational core of the hardware architecture. The RBMC is designed specifically to exploit the data locality of the weight variables, and thus, is responsible for calculating partial energies and updating weights (Eqs. 17, 19, 21). These two sections have $O(n^2)$ time complexity, and through customized data structures and hardware, reduces the computation to $O(n)$. This core itself is divided into four components: the microprogrammed controller, the memory data structures, the energy compute engine and the weight update compute engine.

The RBMC uses a microprogrammed controller instead of the traditional approach of using finite state machines for logic control. This microprogrammed approach provides an efficient method for flow control and arbitration, which is non-trivial since the compute engines require shared access to the memories. Furthermore, a microprogram allows the end-user to program instructions for the RBMC, providing reconfigurability and reusability without resynthesizing the hardware.

The design of the RBMC focuses on the memory data structures since the compute engines would be memory bandwidth limited otherwise; for a 128 x 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. The core takes advantage of the 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 majority of variables described in Table II can be manipulated to match the corresponding memory words. A non-standard element order in the vector is produced; however, the order is deterministic and the binary valued node states are efficiently accessed. Thus, larger batch sizes will increase performance significantly since the compute engines would be memory bandwidth limited otherwise; for a single BRAM since data is written and read serially.

However, the weight parameters and weight updates require both significant resources and bandwidth, while sufficient storage and access to this data is essential for obtaining performance speed-up. A fundamental difficulty in calculating the energies is transposing the weight matrix (Eq. 17). The implementation uses $n$ BRAMS to reduce the $O(n^2)$ single memory accesses to $O(n)$ vector accesses. A non-standard element order in the vector is produced; however, the order is deterministic and the binary valued node states are efficiently manipulated to match the corresponding memory words.

This distributed BRAM-based matrix data structure will be illustrated with an $n = 4$ example (Fig. 7). Four BRAMs are 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. 7a) illustrates how the standard organization 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. 7b) illustrates the elements in the matrix reorganized according to BRAM and address. Fig. 7c) 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. 7d) 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 $addr = ((-BRAM)\%4 + k)\%4$, 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.

The energy compute engine is responsible for calculating the energies (Eqs. 17, 19). 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. The computation can be done with simple hardware components: AND gates, multiplexers and registered fixed-point adders (Fig. 8). 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 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 $2(n-1) \times 32$-bits/cycle. This binary tree of adders effectively reduces an $O(n^2)$ time complexity to $O(n)$, while only requiring $O(n)$ resources. The energy compute engine is capable of reusing the same hardware for both visible and hidden energies since the weights are stored in a manner that provides an entire row or column of the matrix.

The weight update compute engine has two roles: to store the weight update term for the entire batch as well as to commit and clear the weight update terms (Eq. 21). 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, and as a result, no circuit diagram is shown. Since memory update is in parallel, the time complexity is reduced from $O(n^2)$ to $O(n)$, while only requiring $O(n)$ resources.

2) Node Select Core: The Node Select Core (NSC) is a supporting compute core that is responsible for calculating the node states given the partial energies (Eqs. 15, 16). The NSC is designed to provide the maximum throughput by converting a single energy to node state every clock cycle.

Finding a method to compute the sigmoid function, required in Eqs. 4, 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 a significant amount of hardware resources. However, the sigmoid function has properties that are amenable for hardware implementations. First, the range of the function is bounded in the interval $(0, 1)$ – floating point representation is not required. 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 [16], [17], [18]. 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 generating one energy per clock cycle, which serializes the computation. As a result, maximizing bandwidth is the highest priority and high latencies 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.

A BRAM Look Up Table 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. 22. The following notation will be used: the desired point $(u, v)$ exists between the end points $(x_0, y_0)$ and $(x_1, y_1)$.

$$v = \left( \frac{y_1 - y_0}{x_1 - x_0} \right) (u - x_0) + y_0 \quad (22)$$

The naive hardware implementation of Eq. 22 requires both division and multiplication; two operations that 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.

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

![Fig. 8. A circuit diagram of the binary adder tree.](image)

![Fig. 9. Comparison and error residuals of LI and PLI$^2$.](image)
latency. A comparison of \( PLI^2 \) with a LI and the corresponding error is shown in Fig. 9, where \( f(x) \) is the linear interpolation and \( f'(x) \) is its piecewise counterpart. A low-level schematic diagram of the \( PLI^k \) design is shown in Fig. 10. 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. 23-24.

\[
|v_{LI} - v_{PLI^k}|_{\text{average}} = \frac{y_1 - y_0}{2^k + 2} \tag{23}
\]

\[
|v_{LI} - v_{PLI^k}|_{\text{peak}} = \frac{y_1 - y_0}{2^k + 1} \tag{24}
\]

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.

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. 25). This implementation takes advantage of the various favorable properties including odd symmetry and bounded range. 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 maximum 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 for the sigmoid function in the domain \([-12, 12] \) are 4.82E-5 and 3.36E-4, respectively, with a resolution of 11 bits (Fig. 11).

\[
f'(x) = \begin{cases} 
0 & x \leq -8 \\
1-PLI^3(\text{LUT}(-x)) & -8 < x \leq 0 \\
PLI^3(\text{LUT}(x)) & 0 < x \leq 8 \\
1 & x > 8 
\end{cases} \tag{25}
\]

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 [19].

A complete block diagram of the stochastic node selection is presented in Fig. 12. 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.

3) Energy Accumulator Core: The Energy Accumulation Core (EAC) is a supporting compute engine that provides the computation required in RBM partitioning (Eqs. 18, 20). 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.

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 takes advantage of the hardware MPI communication to achieve significant throughput while using limited resources. There is also a BRAM implementation designed for the 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 comm-
The streaming EAC implementation, used for multi-FPGA platforms, 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 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 (Fig. 13).

The EAC RAM implementation is used for virtualizing the modules in the RBM architecture. The streaming implementation cannot be used since it requires the corresponding compute cores to be active and ready for data transfer. In virtualized platforms, there is only a single physical instantiation of the hardware that is being time multiplexed for use by many virtual instances. Instead, additional memory must be used to store intermediate results as each context switch occurs.

The EAC RAM uses a single First-In-First-Out (FIFO) data structure to store both the energies and node states. Large memories are not required and a local BRAM provides sufficient resources (Fig. 14). 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 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 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.

V. Methodology

A. 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 environment, most neural network implementations are written in MATLAB. The MATLAB RBM algorithm in a publicly available database [2] for a 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.

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 relative speed up of four 32-bit word vectors compared to one 32-bit bit scalar computation is considered insignificant.

The hardware implementation was tested on the Berkeley Emulation Engine 2 (BEE2) [20]. 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 the 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 $f_{max}$ 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 made 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.

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 software program, the function $\text{gettimeofday}()$ in the standard $C$ $\text{time.h}$ library was 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 $\text{MPI\_TIME}()$ to time stamp every batch.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Component & Comment & FFs & LUTs & BRAMs \\
\hline
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 & Sigmoid & 568 (0\%) & 862 (1%) & 1 (0\%) \\
EAC & Streaming & 40 (0\%) & 140 (0\%) & 0 (0\%) \\
& RAM & 106 (0\%) & 278 (0\%) & 1 (0\%) \\
\hline
\end{tabular}
\caption{Resource utilization of the architecture cores with the percentage of the total Xilinx XC2VP70 FPGA in parenthesis.}
\end{table}

B. Metrics

The lack of a standard neural network metric raises some issues. An absolute measure of performance is desirable to compare different platforms; however, there is not a single metric that fully quantifies the performance of all neural network architectures. An effective metric for computational performance for any individual type of neural network architecture is 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. 26).

\[
\text{CUPS} = \frac{n^2}{T} \quad (26)
\]

For comparing two different implementations of the same architecture, the speed-up will be measured by the ratio described in Eq. 27, where $S$ is the speed-up, and $T_{hw}$ and $T_{sw}$ are the update periods for the hardware and software implementations, respectively.

\[
S = \frac{T_{sw}}{T_{hw}} \quad (27)
\]

VI. RESULTS

Resource utilization is the primary metric to measure the scalability. The number of Flip-Flops, 4-input LUTs and BRAM resources of the three hardware cores are counted across a variety of configurations (Table IV). 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 number of resources regardless of the number of nodes in the network. The RBMC requires the vast majority of the FPGA resources, while the NSC and EAC are relatively negligible. 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 Flip-Flops and LUTs, resulting in a limiting factor. The architecture achieves the desired $O(n)$ resource utilization as the RBMC’s resource utilization scales linearly, the NSC and EAC utilizations remain constant (Fig. 15).

The single FPGA implementation is used as a baseline for the other platforms since it is the most rudimentary configuration. Its performance with respect to the software counterpart is of primary interest. Since the software implementation has $O(n^2)$ complexity while the hardware implementation is $O(n)$, the speed-up is $O(n)$ (Fig. 16). The maximum computational throughput achieved with a single FPGA design is 1.58GCUPS for the $128 \times 128$ RBM network, resulting in a relative speed-up of 61-fold.

The quad-FPGA platform provides coarse grain parallelism, achieving 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 (Fig. 16). The communication only transfers variables with $O(n)$ size, resulting in a performance trend similar to that of the single FPGA baseline, with the difference indicating the communication overhead. 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 [21].

The virtualized, single FPGA platform achieves a computational throughput of 725MCUPS, resulting in a speed-up of 32-fold for a $256 \times 256$ RBM running on a single $128 \times 128$ hardware core (Fig. 16). Although the virtualized system is considerably slower than its single-FPGA components, the overall performance is still impressive compared to the software implementation. More importantly, the relative speed-up of the virtualized system increases with respect to the network size – by increasing the network size, the performance of the software implementation decreases drastically, while the hardware implementation decreases marginally.

For the virtualized platform, the overhead of weight swapping must be carefully quantified to understand the tradeoffs in using the virtualized system (Fig. 17). The Single FPGA$\times$4 line indicates the update period of a single FPGA multiplied by four, which represents an ideal baseline where weight swapping is instantaneous. The difference between the virtualized platform and the Single FPGA$\times$4 update period is the overhead due to context switching and weight swapping. It is important to note the effect of the batch size on performance – by computing the energies for an entire batch at once, the weight swap can be amortized over a longer period. This is also advantageous in a machine learning aspect since large batch sizes result in better learning for the network.

Comparing with the other implementations is difficult due to the lack of a universal metric for absolute performance (Table V). Kim et al. [12] used a significantly larger and faster FPGA, and had comparable results with respect to its MATLAB reference point. Moving the proposed RBM architecture to the latest generation Xilinx Virtex-5 chip (XC5VLX330T, with over twice the resources of the XC2VP70) is expected to result in a single FPGA of $512 \times 512$ with a speed-up of 600-fold over the C software implementation. The GPU implementation by Raina et al. [13] is difficult to compare since the Goto BLAS implementation could not be obtained and there is no direct comparison in performance. Although the GPU implementation supports significantly larger networks, they rely on the “overlapping patches” technique. The large and deep networks supported by their implementation do not have fully connected layers – 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. Furthermore, the “overlapping patches” is extremely amenable for the virtualized system since there are no global computations.

VII. CONCLUSIONS AND FUTURE WORK

The goal of this work was to develop a high-performance, reconfigurable architecture for Field-Programmable Gate Arrays to drastically speed-up the performance of Restricted Boltzmann Machines. 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 developed to implement the AGS equations. This modular approach provided the 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. A number of low-level data structures and compute engines independent of the neural network framework were conceived and built, 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 quad-FPGA design achieving the maximum performance of 3.13GCUPS resulting in a speed-up of 145-fold. The results indicated that a single FPGA obtains the best performance while the multi-FPGA platform provides additional coarse-grain parallelism. The virtualized platform lacks the performance of the previous two, but is able to scale to larger networks with fewer resources.

Future work includes extending the hardware implementation in a variety of ways. The current implementation uses only binary-valued node states, since it resulted in simpler hardware and the majority of node states for RBMs in DBNs are binary valued because only the bottom most visible layer can be real-valued. Extending the implementation to support binary-valued node states would result in a wider range of applications.

ACKNOWLEDGEMENT

We acknowledge the CMC/SOCRN, NSERC and Xilinx for the hardware, tools and funding provided for this project. We
also acknowledge Arun Patel and Manuel Saldana for their insightful conversations and feedback.

REFERENCES


Daniel Le Ly (S’09) received the B.A.Sc. degree with honours in engineering science and the M.A.Sc. degree in computer engineering from the University of Toronto, Toronto, Ont., Canada, in 2008 and 2009, respectively. He is currently a doctorate candidate at Cornell University, Ithaca, NY. His research interests include biologically inspired computing and robotics.

Paul Chow (S’79–SM’83) received the B.A.Sc. degree (with honours) in engineering science, and the M.A.Sc. and Ph.D. degrees in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 1977, 1979, and 1984, respectively. In 1984 he joined the Computer Systems Laboratory at Stanford University, Stanford, CA, as a Research Associate, where he was a major contributor to an early RISC microprocessor design called MIPS-X, one of the first microprocessors with an on-chip instruction cache. Since January 1988, he has been with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada, where he is now a Professor and holds the Dusan and Anne Miklas Chair in Engineering Design. His research interests include high performance computer architectures, architectures and compilers for embedded processors, VLSI systems design, and field-programmable gate array architectures, systems, and applications. From 1998 to 2001, he was the Chairman of the Technical Advisory Committee (TAC) for the Canadian Microelectronics Corporation (CMC). Since 2001, he has been a member of the CMC Board of Directors and still participates as a member of the TAC. In December 1999, he co-founded AcceleLight Networks to build a high-capacity, carrier-grade, optical switching system. He was the Director of ASIC Technology from May 2000 to October 2002 and managed a group of close to 30 designers that developed over 40 large, high-performance field-programmable gate array designs.