Building a Multi-FPGA Virtualized Restricted Boltzmann Machine Architecture Using Embedded MPI

Charles Lo and Paul Chow
Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada M5S 3G4
{locharl1, pc}@eecg.toronto.edu

ABSTRACT

Several FPGA architectures exist for accelerating Restricted Boltzmann Machines (RBMs). However, the network size for most is limited by the amount of available on-chip memory. Therefore, many FPGAs are required to implement very large networks for use in real-world applications. A virtualized design is able to time-multiplex the hardware resources and handle much larger networks but suffers a performance penalty due to the context switch. In this paper, we present a number of improvements to a virtualized FPGA architecture for RBMs. First, we take advantage of 16-bit arithmetic to pack larger networks onto a chip. Second, a custom DMA engine is designed to reduce the performance impact of the large amount of memory transactions. Finally, the architecture is scaled to multiple FPGAs to gain additional performance through coarse grain parallelism. The design effort required to implement these changes is minimized through the use of an embedded MPI framework. The architecture, tested on a Berkeley Emulation Engine 3 platform running at 100 Mhz, achieves a speed of 12.563 GCUPS on a 8192x8192 network.

Categories and Subject Descriptors

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

General Terms

Design, Performance

Keywords

Restricted Boltzmann Machines, Neural Network Hardware, FPGA, High Performance Computing

FPGA'11, February 27–March 1, 2011, Monterey, California, USA.

1. INTRODUCTION

A Restricted Boltzmann Machine (RBM) is a type of Artificial Neural Network that has garnered interest in the Machine Learning community recently due to its role as a fundamental building block of Deep Belief Networks (DBNs). DBNs have been successfully applied to a number of machine learning problems including semantic hashing of text documents [1] and recognition of handwritten digits [2]. However, a serious impediment to applying RBMs in real world applications is the quadratic increase in computation time with network size. Training of large DBNs can take days on general purpose computers [2].

Fortunately, the structure of the RBM is very amenable to parallel hardware architectures and several FPGA implementations have been proposed to accelerate operations on the network [3, 4, 5]. These architectures are able to achieve a very large speed-up relative to software implementations by utilizing the FPGA multiplier and RAM resources to perform the RBM operations in linear time. However, the size of network is limited by the amount of available on-chip memory. In follow-up works, Ly and Chow developed two methods to handle larger networks. First, one could distribute the network onto multiple FPGAs [6]. This method has the advantage of achieving additional speed-up due to the parallelism afforded by the extra FPGAs, but it does not scale well since the number of required FPGAs increases quadratically with network size. Second, one could virtualize the hardware of a single FPGA to compute the different portions of the network sequentially as they are loaded from external memory [7]. This method is more practically viable, but memory bandwidth and latency become the limiting factors for performance. Kim et al. also proposed loading the network from off-chip [3, 4], but have not yet demonstrated such a system. In comparison to software implementations, a multi-FPGA implementation by Kim et al. [4] achieved a speed-up of 76.67 fold over a MATLAB implementation and a 32 fold speed-up was observed with the virtualized architecture by Ly and Chow [7] relative to an optimized C implementation.

In this paper, we present a number of improvements to the virtualized system in [7]. First, to increase the network size operable on a single FPGA as well as simplify logic, the fixed-point representation of the RBM parameters was reduced from 32-bits to 16-bits. This allowed for a doubling of network size that could fit on-chip. Second, a new Direct Memory Access (DMA) engine was designed to increase the throughput to the external memory as well as minimize processor interaction. Finally, the virtualized design was ex-

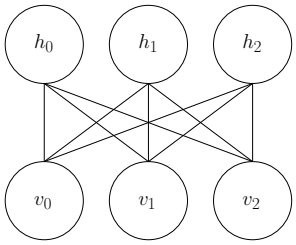

Figure 1: Structure of a 3×3 Restricted Boltzmann Machine. Each line between nodes is a weighted connection.

tended to multiple FPGAs to gain additional performance through coarse grain parallelism.

The architecture in [7] was developed using an embedded Message Passing Interface (MPI) programming model. MPI is popular in the High Performance Computing (HPC) community for implementing parallel distributed memory systems. In embedded systems, MPI allows for the abstraction of hardware engines, streamlining the design flow as well as providing a simple method of interaction between hardware and software components. However, due to the differences between HPC and embedded system environments, MPI implementations in the embedded domain are still maturing. This work presents an opportunity to study the use of embedded MPI. In particular, we examine how the MPI programming model facilitates the modification and extension of embedded FPGA applications.

The rest of the paper is organized as follows: Section 2 provides some background on the operation of Restricted Boltzmann Machines as well as an overview of the basis virtualized architecture. Section 3 describes the modifications made to the hardware due to the 16-bit fixed point representation. Section 4 describes some of the communication challenges involved in implementing the virtualized system as well as the custom DMA engine. In Section 5 the extension to multiple FPGAs is presented. Section 6 examines the use of embedded MPI in this project. Results are presented in Section 7 and conclusions are given in Section 8.

2. BACKGROUND

A Restricted Boltzmann Machine (RBM) consists of two layers of binary-valued neurons or *nodes*; a *visible* layer as well as a *hidden* layer. The network is a bipartite graph, where each node of one layer is connected to all of the nodes of the opposite layer by a weighted connection with no connections between nodes of the same layer. We will denote the weight connecting visible node i to hidden node j by $w_{i,j}$. A graphical representation of this topology is shown in Fig. 1.

If we partition the visible and hidden nodes into subsequences, the weights can be represented as a block matrix as shown in Eqns. 1-3

$$\mathbf{W} = \begin{bmatrix} W_{0,0} & \cdots & W_{0,M-1} \\ \vdots & \ddots & \vdots \\ W_{N-1,0} & \cdots & W_{N-1,M-1} \end{bmatrix} \quad (1)$$

$$\mathbf{V} = [V_0 \cdots V_{N-1}] \quad (2)$$

$$\mathbf{H} = [H_0 \cdots H_{M-1}] \quad (3)$$

Each weight block and related visible and hidden subsequences have the same structure as the overall weight matrix and node layers. Thus, operations performed on the overall network can be divided into similar operations on the smaller partitions of the network. This method of partitioning is important for time-multiplexing the RBM hardware. Eqns. 4-6 show the individual components of partition (n, m) assuming the sub-sequences are of equal length K.

$$W_{n,m} = \begin{bmatrix} w_{nK,mK} & \cdots & w_{nK,(m+1)K-1} \\ \vdots & \ddots & \vdots \\ w_{(n+1)K-1,mK} & \cdots & w_{(n+1)K-1,(m+1)K-1} \end{bmatrix} \quad (4)$$

$$V_n = [v_{nK} \cdots v_{(n+1)K-1}] \quad (5)$$

$$H_m = [h_{mK} \cdots h_{(m+1)K-1}] \quad (6)$$

The state of each node is a function of the sum of its weighted connections, which can be interpreted as an *energy*. Eqns. 7 and 8 show the energy for the visible node i and hidden node j respectively. For the partitioned block weights and nodes, *partial energies* may be computed that must be accumulated to then form the final energy.

$$E_{V_i} = \sum_{j=0}^{MK-1} h_j w_{i,j} = \sum_{m=0}^{M-1} (\sum_{k=0}^{K-1} h_{mK+k} w_{i,mK+k}) \quad (7)$$

$$E_{H_j} = \sum_{i=0}^{NK-1} v_i w_{i,j} = \sum_{n=0}^{N-1} (\sum_{k=0}^{K-1} v_{nK+k} w_{nK+k,j}) \quad (8)$$

The energy calculation may also be expressed in vector form:

$$\mathbf{E_V} = H \cdot W^T = \begin{bmatrix} \mathbf{E_{V_0}} \\ \vdots \\ \mathbf{E_{V_{N-1}}} \end{bmatrix} = \begin{bmatrix} \sum_{m=0}^{M-1} H_m W_{0,m}^T \\ \vdots \\ \sum_{m=0}^{M-1} H_m W_{N-1,m}^T \end{bmatrix} \quad (9)$$

$$\mathbf{E_H} = V \cdot W = \begin{bmatrix} \mathbf{E_{H_0}} \\ \vdots \\ \mathbf{E_{H_{M-1}}} \end{bmatrix} = \begin{bmatrix} \sum_{n=0}^{N-1} V_n W_{n,0} \\ \vdots \\ \sum_{n=0}^{N-1} V_n W_{n,M-1} \end{bmatrix} \quad (10)$$

Given the energy, node states are determined stochastically using the *sigmoid* function shown in Eqns. 11 and 12.

$$P(v_i = 1) = \frac{1}{1 + e^{-E_{V_i}}} \quad (11)$$

$$P(h_j = 1) = \frac{1}{1 + e^{-E_{H_j}}} \quad (12)$$

RBM training consists of two main stages: a node state calculation stage called Alternating Gibbs Sampling (AGS) and a weight update stage. AGS consists of a number of phases. In the first AGS phase, the visible layer is initialized with a training example or test data and the hidden layer is *generated* using the equations above. In the next

AGS phase, the visible layer is similarly *reconstructed*. This node selection continues in an alternating fashion until some S'th AGS phase. The AGS phase will be denoted by a superscript so that v_i^1 is the state of the i'th visible node in the first AGS phase. The energy calculation during the AGS phases is an $O(n^2)$ operation in a sequential processor and the sigmoid function involves costly division and exponentiation operations.

The weight update stage involves taking the nodes from the first and S'th AGS phases and applying the learning rule shown in Eqn. 13 where ϵ is the *learning rate*. The most precise weight update is calculated when $S = \infty$. However, learning has been shown to perform well when $S = 3$ [8] . Notice that the weight updates may be performed independently for each partition of the network.

$$\Delta w_{i,j} = \epsilon((v_i h_j)^0 - (v_i h_j)^S) \tag{13}$$

To have weight updates that represent the entire set of training data, it would be best to calculate the average weight update for all training examples before committing the change; this is called *batch learning*. However for large training sets, this method of weight update could result in long computation times between updates. To address this problem, we can reduce the number of training examples by splitting them into *mini-batches* and thus increase the update rate at the expense of precision. The number of training examples used per weight update is called the *mini-batch size*. More details on RBM operation can be found in [9].

This work extends upon the single FPGA virtualized architecture in [7]. That system consisted of four major components connected via an embedded MPI network: A Restricted Boltzmann Machine Core (RBMC), an Energy Accumulator Core (EAC), a Node Select Core (NSC) and a PowerPC processor. The RBMC performed Eqns. 7 and 8 in $O(n)$ time by operating on a row or column of the weight matrix in parallel. To facilitate this kind of access pattern, many Block RAMs (BRAMs) were required to store components of the weight matrix, thus the size of RBM was limited by the number of BRAMs available on the FPGA. Partial energy accumulation was handled by the EAC that then passed the energy to the NSC to perform stochastic node selection (Eqns. 11 and 12) using a look up table and piecewise linear interpolator. The PowerPC was used to arbitrate the hardware engines as well as stream data to and from external memory.

3. FIXED-POINT REPRESENTATION

The virtualized Restricted Boltzmann Machine architecture in [7] represented weight and energy values as 32-bit fixed-point numbers. This data width was a convenient design choice since the on-chip network operated with 32-bit data widths and the configurable dual-ported Block RAMs (BRAMs) supported up to 36-bit widths. However, significant performance improvements can be realized with a reduction in bit widths. In particular, by reducing the width to 18-bits or less, two weights can be stored on the same BRAM. This is significant since it doubles the amount of weight data the RBMC can process at once.

Reducing the fixed-point precision can introduce some serious problems that must be weighed against the performance advantages. Depending on the RBM application, there exists the possibility of overflow or underflow. This could lead to problems finding a set of values in weight space to accurately represent the given training set. Based on software simulations, a width of 16-bits for weights and energies was found to be adequate for training images from the MNIST data set of handwritten digits. In addition, the RBM implementation by Kim et al. uses 16-bit weight representation [3] and previous studies support the use of 16-bit weights in neural networks [10].

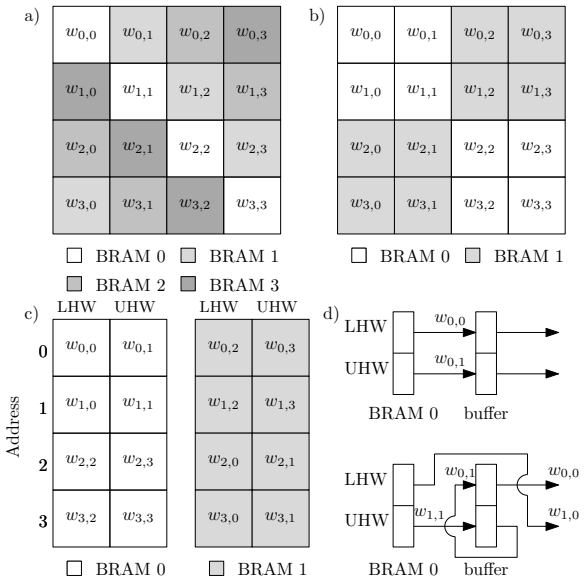

Figure 2: Arrangement of a 4×4 weight matrix in BRAM. a) Original BRAM structure with one weight at each address. b) Block diagonal structure where each BRAM is responsible for two adjacent elements of every row and column. c) Layout of the weights in the BRAMs. One set of weights occupies the Lower Half Word (LHW) at each address while the other occupies the Upper Half World (UHW). d) The two phases of weight re-ordering during a column read. This pipeline stage allows one column to be read out each cycle. The weights shown illustrate reading the first two columns of the weight matrix from BRAM 0.

The RBMC was able to to access rows or columns of the weight matrix in parallel by storing the diagonals of the matrix in separate BRAMs such that each was only responsible for one element in every row and column. A special addressing scheme was used to ensure the correct elements were accessed during the row or column reads.

By halving the fixed-point precision, the same BRAM can store two sets of weights but the access pattern becomes more complicated since two weights, not necessarily at the same address, must be accessed in parallel on the same BRAM. FPGA RAM primitives are typically dual-ported, so a simple way of accessing two weights simultaneously is to take advantage of the independent read ports. However, there are advantages to using a single read port. For instance, the 18 Kbit Xilinx Virtex-5 BRAMs can be configured in simple dual-ported mode in which case the port width doubles from 18-bits to 36-bits, with a corresponding decrease in depth, but with one dedicated read port and one dedicated write port. To take advantage of this type of

RAM, we would like a method of addressing using only one read port. This can be achieved by storing block diagonals of the weight matrix in each RAM as shown in Fig. 2b.

This method of weight distribution in BRAM allows for any two sequential rows or columns of the weight matrix to be read every two cycles. When reading rows of the matrix, the elements of any row may be accessed in parallel by simply setting the address of all of the BRAMs to the appropriate row number. However, when reading columns, the weights must be reordered as they exit the BRAMs before they can enter the energy calculation hardware. This step requires one buffer and can be pipelined so that effectively one column is read per cycle. Fig. 2d shows an example of how this buffer works to re-order the weights during column reads.

4. COMMUNICATIONS

Communications can easily become the bottleneck in hardware systems. This is especially true in virtualized systems where it is important to minimize the time spent moving new data and synchronizing components during a context switch relative to the time spent in computation. In this section we will examine the data flow to and from memory required to virtualize the hardware engines, the capabilities of the MPI network on chip and describe a new DMA Engine that helps reduce the cost of communication and thus increase the relative time spent in computation.

4.1 Context Switch

The memory structure in the RBMC works well when weights are loaded in BRAMs with individual read ports. However, the access patterns are not well suited to streaming portions of weights from external memory since non-contiguous addresses must be read and thus burst reads may not be used. Instead of streaming, the virtualized design in [7] used full context switches; loading entire weight blocks into the BRAMs before any computation. This allowed for the use of bursts, but created a very large amount of idle time for the RBMC while data was being read and written back to external memory.

To examine the data flow in the virtualized system, we can break the computation into three main parts for each network partition. First, in the energy computation phase, weights and node states are sent to the RBMC after which partial energy is calculated and written back for each node layer in the mini-batch. During the node selection phase, the partial energies calculated for a row or column are accumulated in the EAC which forwards the result to the NSC. The NSC then calculates the node states and the result is written back to memory. Note that node selection occurs once for each block row or column of the weight matrix consisting of P partitions and thus only occurs for a fraction of the total partitions. Finally, in the weight update phase, weights are sent once more to the RBMC along with the node states calculated during the AGS phases and the updated weights are written back to memory. Fig. 3 shows the data flow between hardware components and Table 1 summarizes the types and amount of data that must be sent for each phase.

Of primary concern in terms of performance is the $O(n^2)$ amount of weights that must be transferred during energy computation and weight update. The system achieves speedup by performing the RBM operations in $O(n)$ time, thus if

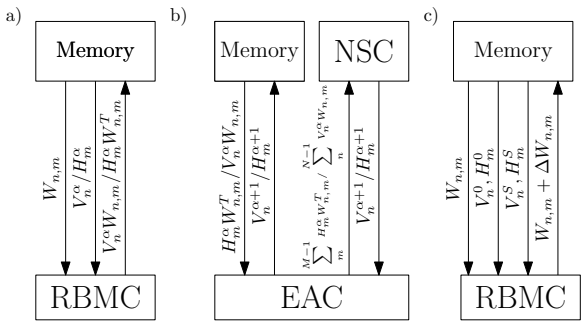

Figure 3: Data flow between memory and the hardware components for a network partition (n, m). a) Energy computation for AGS phase α. b) Node selection for AGS phase α. c) Weight update phase

Energy Computation		
Data Type	Complexity	Size [bits]
weights	$O(n^2)$	$16n^2$
energy	$O(n)$	$16n * L$
nodes	$O(n)$	$n * L$
Node Selection		
Data Type	Complexity	Size [bits]
energy	$O(n)$	$16n * L * P$
nodes	$O(n)$	$n * L$
Weight Update		
Data Type	Complexity	Size [bits]
weights	$O(n^2)$	$2 * 16n^2$
nodes	$O(n)$	$4 * n * L$

Table 1: Summary of major memory transfers for each network partition during a batch update where n is the number of nodes in the layers of the network partition, L is the batch size and P is the number of partitions in a row or column assuming the network is symmetric

the majority of the time is spent in weight transfers, we lose a dramatic amount of performance. One way of reducing the impact of the context switches is by using large batch sizes. Once the weights are loaded into the RBMC BRAMs, computing energies and weight updates only require the transfer of node states and energy. These transfers are much smaller in size and grow linearly with network and batch size. Thus, we can amortize the cost of the weight transfers by using sufficiently large batch sizes such that the operating time becomes dominated by the $O(n)$ operations. Fig. 5 in the results section shows the effect of batch size on performance.

4.2 Network on Chip

The communication layer is provided by ArchES-MPI[1] [11], an embedded implementation of the Message Passing Interface (MPI). Processors and hardware engines are abstracted by the concept of *ranks* and a maximum throughput of 128 bits/cycle is available between each rank via point-to-point links. Hardware engines interact with the network via a message passing engine (MPE) that handles the details of the network protocol. To initiate a transfer, one

[1]ArchES-MPI is derived from TMD-MPI

72-bit command is written to the MPE. Thus, the layer is low overhead while providing high bandwidth. ArchES-MPI supports both rendezvous and ready sends. The former incurs additional latency due to the synchronization of the transmitting and receiving MPEs but is safer and avoids network congestion.

The 128-bit data path allows for the transfer of eight 16-bit weights or energies per cycle or 128 node states per cycle. To take advantage of the wide datapath, the EAC and NSC were modified to perform parallel energy accumulation and node selection on the eight incoming energies per cycle. This required the use of eight parallel uniform random number generators for node selection. During the development of this system, multiple Tausworthe-88 generators [12] were instantiated with arbitrarily chosen seeds to facilitate testing. This is not satisfactory in practice, since generated subsequences could overlap and are not necessarily uncorrelated [13]. The other FPGA RBM architectures [4, 6] also used parallel node selection but also have not addressed the problem of parallel random number generation. This is left as a topic for future work.

4.3 DMA Engine

In [7], the processor was responsible for queuing MPI data transfers between the compute cores and the external memory. A Direct Memory Access (DMA) engine connected via the Processor Local Bus (PLB), called the PLB_MPE, was used to interface the processor with the MPI network. The PLB_MPE allowed for the use of burst transactions from a Multi-Port Memory Controller (MPMC) [14] over the PLB as well as nonblocking sends and recieves. However, the PLB has significant overhead and relatively low throughput. In addition, the PowerPC incurred function call overheads when beginning MPI transactions.

Instead of using a PLB connected interface, a new Memory Access Core (MAC) was designed to use the Native Port Interface of the MPMC. The MAC operates using a simple set of instructions identifying the MPI operation (Ready Send, Synchronous Send or Receive), the memory location to access, the message tag and the size of the message. These instructions are stored on a simple FIFO and can be loaded either from another rank or read directly from external memory. Thus, once loaded in the FIFO, only one cycle is required to fetch the next instruction and begin the next MPI transaction. By operating the core at the memory frequency, twice that of the hardware cores, the MAC is able to saturate its MPI link with 128-bits per cycle during bursts. The MAC handles all memory accesses and essentially replaces the processor during computation. A processor is only required as an interface to load data from a host computer and to send the first instruction to the MAC. In the future, the on-chip processor could be completely replaced with a MPI link directly to an external X86 processor.

5. EXTENSION TO MULTIPLE FPGAS

The goal in extending the virtualized architecture to multiple FPGAs is to use the coarse grain parallelism to provide extra speed-up. Two primary challenges arise when distributing the RBM across chips. First, the amount of work should be balanced such that the available hardware is being fully utilized at all times. Second, since communication costs can be very high in off-chip interconnects, inter-FPGA communication should be minimized.

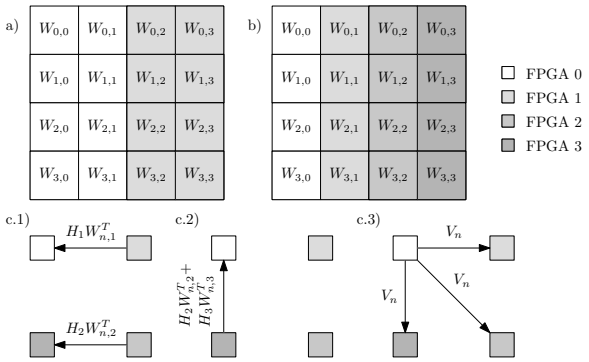

Figure 4: Distribution of 4×4 partitions of weights among multiple FPGAs. a) Two FPGAs: Each FPGA can calculate two blocks of hidden nodes independently but synchronization is required between both FPGAs during selection of visible nodes. b) With four FPGAs, each is responsible for one block of hidden nodes and synchronization is required for visible node selection. c) Visualization of energy summation and node selection for a block of visible nodes V_n in the four-FPGA system shown in (b).

The partitioning method described for virtualizing a RBM on a single FPGA can be easily applied to distribute the work across multiple FPGAs. Partitions of the weight matrix and the corresponding nodes are distributed across the FPGAs by grouping the partitions into larger contiguous blocks as shown in Fig. 4a and 4b. The system operates very similarly to the single FPGA version. The phases of energy computation and weight update only require data local to each partition, so they can be performed independently on each FPGA. During node selection, energies are first summed locally and if required, accumulated across FPGAs before the final node calculation. For instance, when calculating visible node states in the two FPGA system shown in Fig. 4a, two partial energies would be accumulated locally on each FPGA before being summed on one FPGA. The selected nodes are then broadcast to the participating FPGAs. Notice that if the partitions on an FPGA span an entire row or column of the weight matrix, no inter-FPGA communications are required to calculate the corresponding node states.

The synchronization step involved in node selection represents a major bottleneck in terms of the scalability of the system. To facilitate the transfer of partial energies between many FPGAs, the EACs were modified to support a large scale tree-adder style summation. This helps reduce network congestion by limiting the number of energy transmissions at any one time to $log_2(N)$ where N is the number of FPGAs. Fig. 4c shows the steps in node selection for a four-FPGA system. The node states are only n bits, where n is the number of nodes per layer in the RBM partition, so the broadcast is acceptable. However, the number of node transmissions increases linearly with the number of FPGAs, thus the overall cost of synchronization increases as $O(N)$ with the number of FPGAs.

6. MPI PROGRAMMING MODEL

As FPGAs increase in size, they become more and more viable as platforms for reconfigurable hardware accelerators such as the RBM architecture presented in this paper. However, as the systems become more complex, they also become more difficult to design and manage. The Message Passing Interface (MPI) programming model provides one method of partitioning embedded systems and simplifying their design.

In an MPI design flow, the application is first explicitly partitioned into a number of tasks. These tasks are performed by compute engines divided into *ranks*. Through the abstraction provided by the concept of ranks, the implementation details of compute engines are hidden from other components in an MPI system. This abstraction can be provided in a reconfigurable hardware design by embedded MPI implementations such as ArchES-MPI. In ArchES-MPI, hardware engines interact with an on-chip point-to-point network via a Message Passing Engine (MPE). The MPE hides the details of the communication layer and thus isolates the engine from other parts of the system.

The use of ArchES-MPI has several benefits from a design and maintainability standpoint. First, since the hardware components are encapsulated into ranks, the design becomes portable and scalable over platform changes. Designs may take advantage of larger FPGAs by simply instantiating more engines on the same chip. Since each instantiation is an independent rank, the control required to utilize the additional hardware is just the appropriate MPI messages. Likewise, additional FPGAs may be added to the system by connecting them to the MPI network. No significant redesign is required to take advantage of additional resources.

ArchES-MPI also provides an abstraction between hardware and software components. This enables the seamless interaction of software and hardware though a standard interface. Since the implementation details of each rank are hidden, systems may be easily prototyped in software and very intensive tasks can be independently moved to hardware without affecting the operation of the remaining software components. In addition, hardware changes can be made very easily without affecting the operation of the rest of the system. This is imperative for incremental improvements in large designs.

Finally, from a system design perspective, ArchES-MPI allows for a flexible data flow. Since data is routed through a point-to-point network on chip, the data path can be easily reconfigured via messages. This creates flexibility in the way hardware engines are used in different situations.

During the design of the multi-FPGA RBM architecture in this paper, we took advantage of several of the features provided by ArchES-MPI. First, the architecture was ported from the Berkeley Emulation Engine 2 (BEE2) [15] platform to the Berkeley Emulation Engine 3 (BEE3) [16]. The BEE2 consisted of five Virtex-II Pro FPGAs with hard PowerPC processors connected in a mesh whereas the BEE3 contained four Virtex-5 FPGAs connected in a ring without any hard processor. In the absence of the PowerPCs, MicroBlaze soft-processors were instantiated and integrated seamlessly into the MPI network. Since the hardware engines were behind the MPI layer, only an update in the netlist was required during the transition. In addition, the details of the inter-FPGA links were abstracted by the network on chip. Since the data ports between FPGAs on the BEE3 were only 72-bits wide, the width of the MPI network had to be reduced to 32-bits to facilitate synchronous communication bewteen the FPGAs. A packet width converter was used to split the 128-bit MPI data into four 32-bit parts before being sent over the interconnection. This change in data width was also transparent to the hardware components. Second, the MAC took over many of the responsibilities of the processor. Originally, the processor was used to feed the compute engines since the PLB interface was a simple way to access external memory. However, when additional performance was desired, the MAC was designed. No modifications to the rest of the hardware cores were required to accommodate this change. Finally, the programmable nature of the MPI network allowed the EACs to be configured to act as a large scale tree adder across chip boundaries as well as simple summation units.

ArchES-MPI may not be appropriate for all scenarios. First, not all designs may be amenable to an MPI based dataflow. In particular, the design must be carefully partitioned such that portions requiring very high throughput and specific control such as pipelined datapaths do not transfer data through the network. In addition, some overhead is incurred during data transfer, especially during synchronous sends where handshaking is required between the sender and receiver. Finally, to make the most of the on-chip links, the data width of the hardware engines should be matched to the width of the network. Designs should be evaluated to determine whether or not they fit the MPI framework.

7. RESULTS AND ANALYSIS

The design was tested on the Berkeley Emulation Engine 3 (BEE3) hardware platform [16]. The BEE3 contained four Xilinx Virtex-5 XC5VLX155T FPGAs connected in a ring via 72-bit wide interconnects. Each FPGA was also configured with a 2GB RDIMM. The XC5VLX155T has 24,320 logic slices (each slice contains four 6-input LUTS) and 424 18 Kbit BRAMs for 7,632 Kbits of embedded RAM. Based on the number of BRAMs available, a maximum RBMC size of 256x256 was instantiated on each FPGA. A MicroBlaze soft processor was also instantiated on every FPGA to initialize the system. The compute engines, MicroBlaze and MPI network ran at 100 MHz while the DDR2 memory controller and MAC ran at 200 MHz. The clock frequency was limited by long wire delays on this platform. Future work will include the investigation of these critical paths to achieve timing closure at higher clock frequencies. Computation time was measured by calling the MPI function *MPI_TIME()* on the MicroBlaze after each mini-batch.

7.1 Metrics

One popular method of measuring neural network performance is Connection Updates per Second (CUPS). This is defined as the number of weight updates per second or:

$$CUPS = \frac{n^2}{T} \tag{14}$$

Where n is the size of the node layers and T is the amount of time for all of the weights to be updated. One problem with CUPS when measuring hardware performance is it does not take into account mini-batch size; as mini-batch size increases, CUPS will decrease although the number of weight update calculations increases.

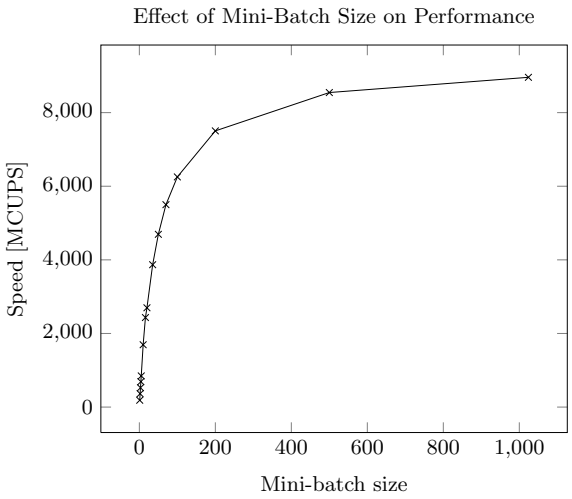

Figure 5: Relationship between mini-batch size and performance for a 1024x1024 network running on a four-FPGA virtualized system.

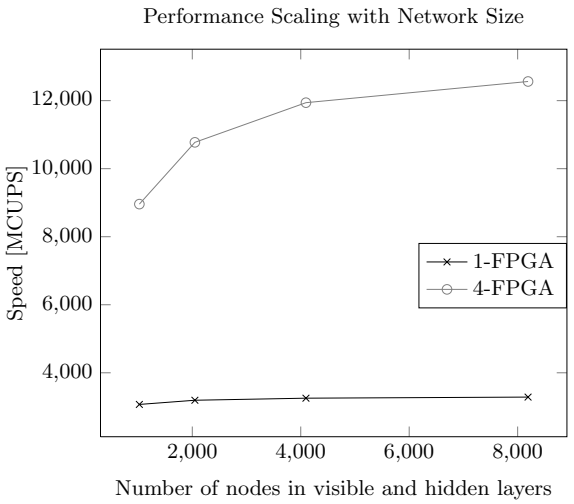

Figure 6: Performance of the single-FPGA and four-FPGA virtualized systems for different network sizes. The mini-batch size was fixed at 1024.

We will use a slightly modified version of CUPS in this paper:

$$CUPS = \frac{n^2}{T} L \tag{15}$$

Where L is the mini-batch size. This definition of CUPS is the effective number of weight changes per second.

In [4], Kim et al. described a performance measure called mult/s which measured the major vector multiplication operations involved in RBM training. For three AGS phases, this results in:

$$1 \text{ CUPS} = 5 \text{ mult/s} \tag{16}$$

We will use CUPS as defined in Eqn. 15 as the metric for the rest of this paper.

7.2 Context Switch Penalty

As mentioned in Section 4, the transfer of weights during context switches represents a very large performance penalty. Fig. 5 shows the speed of the four-FPGA 1024x1024 system for different mini-batch sizes. Performance increases dramatically with mini-batch size. For small mini-batch sizes, the majority of time is spent transferring the weights to and from the RBMC during the energy computation and weight update phases. Once the weights are loaded into the RBMC, the remaining operations of transferring node states and energies as well as the computations take a relatively small percentage of time. Thus, performance scales linearly at low mini-batch sizes with diminishing returns once batch operations begin to dominate at large mini-batch sizes.

It is clear from these results that large mini-batches should be used to maximize the performance of the virtualized system. From an RBM training perspective, this is acceptable since the precision of weight updates increases with mini-batch size for stationary data. The trade-off for mini-batch size has traditionally been computation time. However, since the virtualized system is bottlenecked by the weight transfers for small mini-batches, there is very little penalty in performance for increasing mini-batch size in this system.

7.3 Scalability

The scalability of the system to additional FPGAs is limited by the energy accumulation and node broadcast steps during the node selection phase. Although node selection only takes place for a fraction of the total compute time and involves relatively few operations, the cost of this step is exacerbated by the limited bandwidth available and latency in the FPGA interconnects. In the BEE3 platform, the links are direct PCB connections and thus relatively low latency, but MPI data must be converted to 32-bit words when crossing FPGA boundaries because of the limited number of connections. In addition, during node broadcast, the node states must cross two inter-FPGA links to reach a diagonal FPGA due to the ring topology of the platform. Depending on the distribution of the network partitions, a good topology would have connections between FPGAs which share rows and columns of the weight matrix. In the performance tests, a distribution similar to Fig. 4b was used, so a fully connected network would be desired since all FPGAs must interact to reconstruct visible nodes. These costs cause the scalability of the system to be limited when working with smaller networks.

As the network size increases, the proportion of time spent carrying out inter-FPGA synchronization decreases relative to the amount of work performed independently on each FPGA. Therefore, the synchronization penalty decreases as network size increases and almost linear speed-up can be achieved. Fig. 6 shows the speed of the virtualized system running on the single and four-FPGA systems for different network sizes. At a network size of 1024x1024, the four-FPGA system achieves a speed of 9.0 GCUPS or about a three-fold improvement over the single FPGA performance of 3.1 GCUPS. However, at a network size of 8192x8192, the four-FPGA system demonstrates an almost linear speed-up of 12.6 GCUPS relative to the 3.3 GCUPS of the single FPGA.

This architecture performs best when a small number of FPGAs are working on a large network. This type of application is the focus of the virtualized RBM design, and

thus the limits on scalability will not affect the majority of use cases. However, the application should be kept in mind when using this architecture.

7.4 Hardware Architecture Comparison

Table 2 shows a performance comparison of the different FPGA architectures. First, in comparison with the previous virtualized RBM architecture in [7], the improvements made to this system allow it to achieve much greater performance. The architecture in [7] was implemented on a Virtex-II Pro XC2VP70 FPGA with 328 18 Kbit configurable BRAMs. That platform allowed for a maximum 32-bit RBMC of size 128x128 to be instantiated at 100 MHz. On a single FPGA, virtualized systems do not gain much extra performance from increasing RBM size since the number of partitions grows as $O(n^2)$. This can be seen in Fig. 6 where the single-FPGA implementation remains at around 3 GCUPS over a large range of network sizes although the four-FPGA implementation gets extra performance by amortizing the inter-FPGA communication. Therefore, we can roughly compare the raw performance of the single-FPGA virtualized systems and see that the modifications made in this architecture allow for an over four-fold improvement in performance compared to the design in [7].

The virtualized architecture in this paper allows for large networks to be accelerated, but at the cost of performance limitations. First, to achieve maximum performance, the application of this design is limited to large mini-batch sizes and large RBM sizes relative to the number of FPGAs. These conditions are necessary to amortize the cost of the context switch as well as the inter-FPGA communication. Next, even for large mini-batches, the performance of this system is limited by the memory latency and bandwidth. In the best case, the system would achieve maximum performance by keeping the RBMC constantly performing vector and matrix operations. However, since frequent memory accesses are required during node selection, the RBMC must wait for that stage to complete before it can continue processing new data. The latency of external memory access is an additional performance penalty relative to on-chip RAM access.

It is interesting to compare this architecture to the ones proposed by Kim et al. [3, 4]. In their designs, an array of multipliers was used to perform the vector-matrix multiplications. This allowed them to use real valued visible nodes instead of binary valued ones. A key decision in their design was the use of the same memory access pattern for accessing weights from BRAM at the cost of having different sets of accumulator logic for row and column operations. By storing their weights and batch data on-chip, they were able to keep the multiplier array almost constantly busy and thus achieved very high performance. Their architecture also did not require very large batch or RBM sizes to maintain high performance since they did not have external memory access costs to amortize. A disadvantage of this method is that the network and mini-batch sizes are limited by the amount of on-chip resources. However, they may take less of a performance hit if weight streaming is implemented, since the regular weight addressing allows for the use of burst reads and writes from external memory and thus higher bandwidth.

The multi-FPGA architecture presented by Kim et al. [4] also scales better at smaller network sizes. They achieve very high scalability by requiring only nearest neighbour transactions and thus reduce the overhead associated with inter-FPGA communication. In contrast, the architecture presented in this paper only scales well once the cost of inter-FPGA synchronization is accounted for by using large RBM sizes. However, the topology of FPGAs in their architecture is restricted to a ring structure, whereas the multi-FPGA implementation presented in this paper may be used in many different network topologies due to the flexibility provided by ArchES-MPI.

Kim et al. implemented their architecture on four Stratix III EPSL340 FPGAs with the RBM modules running at 150 MHz. The FPGAs were connected in a ring via LVDS pairs with a datarate of 4.8 Gbps in each direction. An EPSL340 FPGA contains 130,000 Adaptive Logic Modules, each acting as two combined 6-input LUTs, for a total of 260,000 6-input LUTs. In addition, the chip contains 288 18x18 multipliers and 16,272 Kbits of embedded RAM. The number of multipliers limits the number of compute elements on each FPGA to 256, thus each FPGA was able to perform 256 multiplications per cycle during the vector operations. This is equivalent to the RBMC size on the BEE3 since a 256x256 RBMC can also operate on 256 vector elements per cycle.

Note that the XC5VLX155T FPGAs in the BEE3, with 97,280 6-input LUTS and 7,632 Kbits of embedded RAM, have about half the logic and RAM capacity as the EPSL340. Given FPGAs with a greater amount of embedded RAM, additional RBMCs could be instantiated to increase computational throughput of this system. The difference in clock frequency also plays an important part in relative performance of the two architectures. The implementation by Kim et at. has inherently 50% greater performance due to the higher clock frequency. Clock frequency was limited by long wire delays on the BEE3 platform; further investigation will be performed to address the critical paths and achieve timing closure at greater frequencies.

As seen in Table 2, performance of the four-FPGA virtualized design is dramatically worse than the multi-FPGA architecture by Kim et al. when compared at the same network size of 1024x1024 and mini-batch size of 16. The low mini-batch size causes the RBMC to be idle for a large portion of the running time while weights are being transferred during the context switch. This is discussed in Section 7.2, and the effect of mini-batch size on performance is shown in Fig. 5. The performance of the architecture by Kim et al. was reported as being invariant to mini-batch size provided the multiplier pipelines were kept fed [4], thus a comparison is still valid at larger mini-batch sizes. Unfortunately, no comparison may be made for larger network sizes, but it can be noted that at 1024x1024 nodes, the inter-FPGA communication costs are not yet amortized and thus the virtualized four-FPGA system performs sub-optimally. If performance is compared at a network size of 8192x8192 and a mini-batch size of 1024, the performance gap partially closes. The remaining difference exists due to the greater clock speed as well as the limited memory bandwidth and latency issues as described earlier. Although the architecture by Kim et al. achieves very high performance in its custom pipeline, the flexibility of MPI makes the design presented in this paper very portable and extensible.

Finally, a number of implementations using Graphics Processing Units (GPUs) have been designed to accelerate RBM training [4], [17]. Since the RBM operations are very heavy

Implementation	Network Size	mBatch Size	Absolute Performance
Virtualized 1-FPGA	1024x1024	16	1051 MCUPS
Virtualized 4-FPGA	1024x1024	16	2433 MCUPS
Virtualized 1-FPGA	1024x1024	1024	3070 MCUPS
Virtualized 4-FPGA	1024x1024	1024	8958 MCUPS
Virtualized 1-FPGA	8192x8192	1024	3286 MCUPS
Virtualized 4-FPGA	8192x8192	1024	12563 MCUPS
Virtualized 1-FPGA [7]	256x256	1024	725 MCUPS
Kim et al. 4-FPGA [4]	1024x1024	16	30666 MCUPS

Table 2: Absolute performance comparison between different FPGA implementations. Here, mBatch size is the mini-batch size. Virtualized FPGA designs were run with a clock speed of 100 MHz and the architecture by Kim et al. used a clock speed of 150 MHz.

in terms of vector and matrix operations, GPUs perform well with respect to general purpose computers and competitively with FPGAs. Comparisons between FPGAs and GPUs may be found in [7] and [4].

8. CONCLUSION

This paper presents several methods of improving the performance of a virtualized Restricted Boltzmann Machine architecture within an embedded MPI framework. Core modifications were made to improve the operable network size and reduce the impact of swapping data to and from external memory. In addition, extra performance was gained through the extension of the architecture to multiple FPGAs. The design effort was greatly reduced through the abstraction provided by ArchES-MPI. Performance of 12.6 GCUPS was achieved on a 8192x8192 RBM with a four-FPGA platform. Very large RBM networks can be realized using this virtualized approach, but due to the inherent performance penalties incurred during memory access, this architecture should be used for applications with large networks and batch sizes to achieve the most speed-up.

Avenues of future work include removing the MicroBlaze processor and replacing it with a PCIe connection to a host processor. This would free up LUTs and BRAMs on the FPGAs as well as improve the speed of initializing the DRAM with test data relative to the JTAG connection currently used. In addition, given more resources, a second MAC could be instantiated to perform node selection in parallel with energy computation. This would reduce the amount of idle time for the RBMC and thus further improve performance. Many neural networks share a similar structure and thus require similar operations. Given the flexibility of the MPI framework, the acceleration of other types of neural networks could also be investigated with this architecture. Finally, the application of this system to Deep Belief Network problems such as the classification of handwritten digits in [2] is currently being investigated.

9. ACKNOWLEDGMENTS

We gratefully acknowledge Daniel Ly for his advice and feedback, NSERC and Xilinx for providing funding and CMC/SOCRN for the hardware and tools used in this project. We also thank the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic Hashing. *International Journal of Approximate Reasoning*, 50(7):969–978, July 2009.

[2] Geoffrey Hinton and Simon Osindero. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, July 2006.

[3] Sang Kyun Kim, Lawrence MacAfee, Peter Leonard McMahon, and Kunle Olukoton. A Highly Scalable Restricted Boltzmann Machine FPGA Implementation. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*, pages 367–372, August 2009.

[4] Sang Kyun Kim, Peter Leonard McMahon, and Kunle Olukotun. A Large-Scale Architecture for Restricted Boltzmann Machines. In *Proceedings of the 2010 18th IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 201–208, May 2010.

[5] Daniel Le Ly and Paul Chow. A High-Performance FPGA Architecture for Restricted Boltzmann Machines. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 73–82, February 2009.

[6] Daniel Le Ly and Paul Chow. A Multi-FPGA Architecture for Stochastic Restricted Boltzmann Machines. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*, pages 168–173, August 2009.

[7] Daniel Le Ly and Paul Chow. High-Performance Reconfigurable Hardware Architecture for Restricted Boltzmann Machines. *IEEE Transactions on Neural Networks*, 21(11):1780–1792, November 2010.

[8] Guy Mayraz and Geoffrey E. Hinton. Recognizing Handwritten Digits Using Hierarchical Products of Experts. *IEEE Transactions on Pattern Analysis and*, 24(2):189–197, February 2002.

[9] Yoav Freund and David Haussler. Unsupervised learning of distributions on binary vectors using 2-layer networks. In *Advances in Neural Information Processing Systems 4*, pages 912–919, 1991.

[10] Perry D. Moerland and Emile Fiesler. Neural Network Adaptations to Hardware Implementations. In *Handbook of Neural Computation*, chapter E1.2. Oxford University Press, 1997.

[11] Manuel Saldaña, Arun Patel, Christopher Madill, Daniel Nunes, Danyao Wang, Henry Styles, Andrew Putnam, Ralph Wittig, and Paul Chow. MPI as an Abstraction for Software-Hardware Interaction for HPRCs. *Proceedings of the Second International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, pages 1–10, November 2008.

[12] Pierre L'Ecuyer. Maximally Equidistributed Combined Tausworthe Generators. *Mathematics of Computation*, 65(213):203–213, January 1996.

[13] Ashok Srinivasan, Michael Mascagni, and David Ceperley. Testing parallel random number generators. *Parallel Computing*, 29(1):69–94, January 2003.

[14] Xilinx. Multi-Port Memory Controller (MPMC) Data Sheet v4.02.a, June 2008.

[15] Chen Chang, John Wawrzynek, and Robert W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design & Test of Computers*, 22(2):114–125, March-April 2005.

[16] John D Davis, Charles P Thacker, and Chen Chang. BEE3: Revitalizing computer architecture research. Technical report, Microsoft Research, April 2009.

[17] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale Deep Unsupervised Learning using Graphics Processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 873–880, 2009.