A MULTI-FPGA ARCHITECTURE FOR STOCHASTIC RESTRICTED BOLTZMANN MACHINES

Daniel L. Ly and Paul Chow

Department of Electrical and Computer Engineering
University of Toronto
Toronto, ON, Canada, M5S 3G4
email: lyd@eecg.toronto.edu, pc@eecg.toronto.edu

ABSTRACT

Although there are many neural network FPGA architectures, there is no framework for designing large, high-performance neural networks suitable for the real world. In this paper, we present two concepts to support a multi-FPGA architecture for stochastic Restricted Boltzmann Machines (RBM), a popular type of neural network. First, a hardware core, called the kth Stage Piecewise Linear Interpolator, is used to implement a high-precision, pipelined function generator. The interpolator increases the resolution of a Look Up Table implementation, guaranteeing an additional bit of precision for every pipeline stage. This function generator is used to implement a sigmoid function required in stochastic node selection. Next, a partitioning algorithm is used to efficiently divide a RBM amongst multiple FPGAs. The partitioning algorithm optimizes performance by minimizing the inter-FPGA communication. The architecture is tested on the Berkeley Emulation Engine 2 running at 100MHz. One board supports a RBM of 256×256 nodes, and 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.

1. INTRODUCTION

There is a growing interest for large, high-performance neural networks. The capabilities of a neural network are highly dependent on its size; this raises a computational barrier since the complexity of software implementations grows quadratically with respect to network size. As a result, training large networks for real-world applications often takes weeks on general-purpose processors. It should be noted that neural networks are composed of an interconnected network of independent processing elements, and thus, are intrinsically parallel. A hardware implementation can achieve superior performance by taking advantage of this parallelism. However, an architecture for large, hardware neural networks has not been proposed due to several key design issues.

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 Saldaña for their advice and feedback.

First, although the majority of computations require few resources, some neural network components do not have straightforward hardware implementations. Most neural networks achieve their unique capabilities through non-linear, transcendental functions. Generating resource-efficient, high-precision transcendental functions in hardware is difficult.

Next, the issue of scaling large networks onto FPGAs is also an open problem. While there have been many attempts to put large networks onto a single FPGA, distributing neural networks across multiple FPGAs has not been as thoroughly investigated. Partitioning neural networks is difficult because there is a significant amount of data dependence and minimizing communication is non-trivial.

This paper will present two novel contributions:

- A piecewise linear interpolator that can be used to increase the precision of Look Up Tables. This hardware can implement transcendental functions with minimal resource utilization. The proposed design is pipelined and can be clocked at high frequencies.
- A method of partitioning Restricted Boltzmann Machines to be distributed across multiple FPGAs. The proposed design will maximize parallel FPGA computation while minimizing communication.

Section 2 will provide some background on Restricted Boltzmann Machines. Section 3 will describe the piecewise linear interpolator while Section 4 will outline the method of multi-FPGA partitioning. Results are presented in Section 5 and the concluding remarks are given in Section 6.

2. BACKGROUND

Of the many neural network varieties, our work focuses on the Restricted Boltzmann Machine (RBM). It is a stochastic and generative network that is capable of capturing and reproducing the underlying statistical properties of a given data set. These unique properties have resulted in a variety of successful applications ranging from recognizing handwritten digits to reducing the dimensionality of data.

A RBM consists of two layers of processing elements, or *nodes*, the *visible layer* and the *hidden layer*. The visible

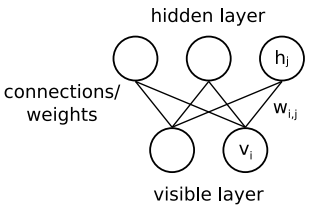

Fig. 1. A schematic diagram of a Restricted Boltzmann Machine with labelled components.

layer is used for input/output access while the hidden layer acts as a latent representation of the data. The nodes have binary states. There are *connections* between every node in opposite layers and no connections between any nodes in the same layer. Every connection has an associated *weight*, which provides the learning parameters for the RBM.

The following notation system will be used: v_i and h_j are the binary states of the ith and jth node in the visible and hidden layer, respectively; $w_{i,j}[k]$ is the weight between the ith and jth node for the kth update. The terminology is summarized in a schematic representation in Fig. 1.

For brevity, matrix expressions are often used to represent the node states and weights, as shown in Eqs. 1-3.

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

$$\mathbf{h} = [h_0 \dots h_{j-1}] \in \mathbb{B}^{1\times j} \tag{2}$$

$$\mathbf{W}[k] = \begin{bmatrix} w_{0,0}[k] & \cdots & w_{0,j-1}[k] \\ \vdots & \ddots & \vdots \\ w_{i-1,0}[k] & \cdots & w_{i-1,j-1}[k] \end{bmatrix} \in \mathbb{R}^{i\times j} \tag{3}$$

The RBM operates and learns via the Alternating Gibbs Sampling (AGS) method. AGS determines the node states of one layer given the other. The process starts with an initial data vector in the visible layer, and generates each layer in an alternating fashion. To differentiate between AGS cycles, each state is indexed with a superscript, x. To determine the node states, an intermediate value, called the *partial energy* for each layer must be calculated, $\mathbf{E_v}$ and $\mathbf{E_h}$ respectively. The AGS computations are summarized in Eqs. 4-8.

$$\mathbf{V}^{x+1} = \begin{cases} \mathbf{V}^0 & , \quad x = 0 \\ f(\mathbf{E}_\mathbf{v}^x) & , \quad x \text{ is odd} \\ \mathbf{V}^x & , \quad x \text{ is even} \end{cases} \tag{4}$$

$$\mathbf{H}^{x+1} = \begin{cases} f(\mathbf{E}_\mathbf{h}^x) & , \quad x \text{ is even} \\ \mathbf{H}^x & , \quad x \text{ is odd} \end{cases} \tag{5}$$

$$\mathbf{E}_\mathbf{v}^x = (\mathbf{H}^x)\mathbf{W}^{\mathrm{T}}, \in \mathbb{R}^{1\times i} \tag{6}$$

$$\mathbf{E}_\mathbf{h}^x = (\mathbf{V}^x)\mathbf{W}, \in \mathbb{R}^{1\times j} \tag{7}$$

$$\mathbf{W}[k+1] = \mathbf{W}[k] + \epsilon\left((\mathbf{V}^1)^{\mathrm{T}}\mathbf{H}^1 - (\mathbf{V}^X)^{\mathrm{T}}(\mathbf{H}^X)\right) \tag{8}$$

Where $x = \{0, \dots, X\}$ and $f(\cdot)$ is a sampled sigmoid probability distribution applied to each element in the partial energy vector (Eqs. 9-10).

$$\mathrm{P}(v_i = 1) = \frac{1}{1+e^{-E_i}} \tag{9}$$

$$\mathrm{P}(h_j = 1) = \frac{1}{1+e^{-E_j}} \tag{10}$$

Additional details regarding the operation and learning of RBMs can be found in [1].

This work builds on the framework proposed in [2], which presented a RBM FPGA architecture capable of decreasing the $O(n^2)$ energy computation and weight update to $O(n)$ time complexity through customized compute engines. As a result, the overall AGS computation was reduced to $O(n)$, in hardware, compared to a $O(n^2)$ software implementation. Two hardware cores were introduced: the Restricted Boltzmann Machine Core (RBMC) and the Node Select Core (NSC). The RBMC calculated Eqs. 6-8 while the NSC implemented a simplified, threshold function node selection (Eqs. 4-5). An important design constraint of the RBMC was that the visible and hidden layers have the same node size that is a power of two, $(i = j = n = 2^a)$.

This work adds to the proposed architecture by removing two significant limitations. The NSC used a deterministic threshold function for node selection in Eqs. 4-5. Although this provides an acceptable approximation, a complete RBM implementation should use the sigmoid probability function, Eqs. 9-10. Next, the RBMC was limited to a network size of 128×128 on a Virtex-II Pro FPGA. Real applications require much larger networks and distributing the network across multiple FPGAs will be the first step towards large hardware neural network implementations.

3. STOCHASTIC NODE SELECTION DESIGN

3.1. Background

Finding a method to compute the sigmoid function, required in Eqs. 9-10, has been a source of difficulty in hardware neural network design. The naive approach requires both exponential functions and division, two operations that would require significant hardware resources.

However, the sigmoid function is amenable for hardware implementations. First, the range of the function is bounded in the interval $(0, 1)$ – floating point representation is not required. 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 [3],[4],[5]. However, the implementations were often designed for a different use case: the function was vastly replicated across the FPGA. As a result, it was designed for minimal resource utilization and low latency. Precision and bandwidth was not a priority.

A significantly different use case is present in the current framework. The RBMC is capable of providing one energy per clock cycle, which serializes the computation. As a result, maximizing bandwidth and the ability to select a node

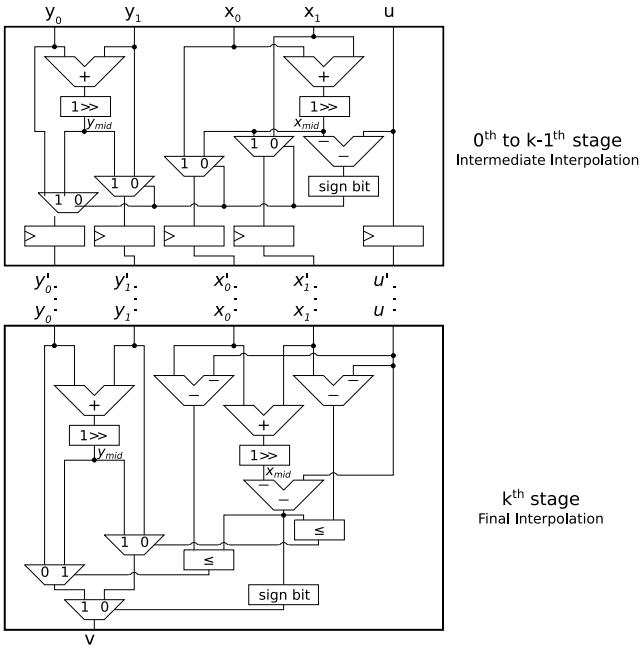

Fig. 2. A schematic diagram of the PLI^k. The *intermediate* stages iteratively selects new end points from the current midpoints and end points based on the search value, u. The *final* stage selects an output, v, from the values $\{y_0, y_{\text{mid}}, y_1\}$ based on the distance between u and $\{x_0, x_{\text{mid}}, x_1\}$.

state every cycle is the highest priority. High latency due to deep pipelines are acceptable. Furthermore, since the NSC will not be vastly replicated; using more resources, including using one Block RAM (BRAM) as a Look Up Table (LUT), is acceptable. Finally, high precision is desired.

3.2. Piecewise Linear Interpolator

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

To increase the resolution, an interpolator was designed to operate on the two outputs of a LUT. The implementation focused on the Linear Interpolator (LI), Eq. 11. The following notation will be used: the search point (u, v) exists between the end points (x_0, y_0) and (x_1, y_1).

$$v = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(u - x_0) + y_0 \tag{11}$$

The naive hardware implementation of Eq. 11 requires both division and multiplication; two operations which utilize significant resources. Instead, it should be noted that adding, subtracting, shifting, and comparing have hardware efficient 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.

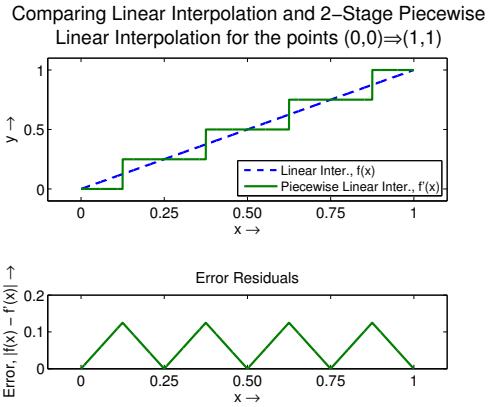

Fig. 3. Comparison and error residuals of LI and PLI^2.

This hardware is called the kth Stage Piecewise Linear Interpolator (PLI^k), where each successive stage does one iteration of a binary search for the search point for one cycle of latency. A low-level schematic diagram of the PLI^k design is shown in Fig. 2. A comparison of PLI^2 with a LI and the corresponding error is shown in Fig. 3, where $f(x)$ is the ideal function and $f'(x)$ is its approximation.

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. 12-13.

$$|v_{\text{LI}} - v_{\text{PLI}^k}|_{\text{average}} = \frac{y_1 - y_0}{2^{k+2}} \tag{12}$$

$$|v_{\text{LI}} - v_{\text{PLI}^k}|_{\text{peak}} = \frac{y_1 - y_0}{2^{k+1}} \tag{13}$$

It is important to note that the PLI^k can be used on any LUT function implementation to increase the precision.

3.3. Sigmoid Node Selection

Using the BRAM LUT and PLI^k, a high-precision pipelined sigmoid transfer function was generated. Using fixed-point inputs, the sigmoid function is defined as a piecewise implementation (Eq. 14).

$$f'(x) = \begin{cases} 0 & , x \leq -8 \\ 1 - \text{PLI}^3(\text{LUT}(-x)) & , -8 < x \leq 0 \\ \text{PLI}^3(\text{LUT}(x)) & , 0 < x \leq 8 \\ 1 & , x > 8 \end{cases} \tag{14}$$

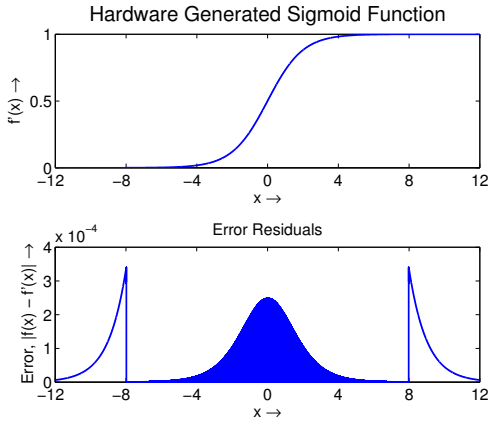

Fig. 4. The reconstructed signal and error residues generated by the hardware for Eq. 14. The error residuals does not have a smooth curve since there are 512 points, which are exact results from the LUT

This implementation uses the bounded and odd symmetry properties of the sigmoid function to increase the LUT sampling frequency. For the outer limits of the domain, $x > 8$ or $x \leq -8$, the results are sufficiently close to the bounds of 1 and 0, respectively, with a 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 are for the sigmoid function in the domain $[-12, 12)$ are 4.82E-5 and 3.36E-4, respectively, with a precision of 11 bits (Fig. 4).

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, produces one result every clock cycle and requires little resource overhead [6].

A complete block diagram of the stochastic node selection is presented in Fig. 5. The total latency for the hardware implementation is 8 clock cycles and, due to the pipelined design, is able to select a node every clock cycle.

4. DISTRIBUTED ARCHITECTURE FOR RESTRICTED BOLTZMANN MACHINES

4.1. Partitioning a Restricted Boltzmann Machine

Large RBMs are difficult to fit onto a single FPGA due to resource limitations. The amount of on-chip memory required to store the weights is the limiting factor. Although using off-chip resources would provide more memory, the bandwidth limitations would create a performance bottleneck.

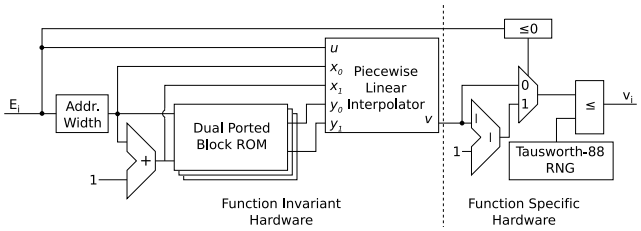

Fig. 5. Block diagram of the stochastic node selection. Note the system is divided into a function invariant section and a sigmoid specific section. The logic for inputs $E_i > 8$ or $E_i \leq -8$ is not included.

As an alternate approach, distributing large RBMs on multiple FPGAs would alleviate that bottleneck. However, the method of partitioning the RBM must be carefully considered since inter-FPGA communication must be minimized to achieve optimal performance.

To determine the optimal partition of RBMs, the memory footprint of RBM variables is analyzed. Weights require n^2 32-bit, fixed-point values, energies require n 32-bit, fixed-point values and the node states only require n 1-bit values for each layer. Given the disproportionate balance of memory requirements, the data locality of the weights provides the solution – there will be no weight transfers if each RBMC contains a unique set of weights. By having private weights but shared node states, an equivalent and distributed RBM is constructed that minimizes communication.

Examples of the partitioning algorithm are shown in Fig. 6. In these examples, the largest RBM that can be instantiated on a single FPGA is 32×32. Three cases will be presented:

Case 0: 1 FPGA, 32 $\times$ 32 – This case is the baseline platform. The 32×32 RBM fits on a single FPGA. The RBMC calculates the partial energies and sends them to the NSC. In return, the RBMC receives the node states from the NSC.

Case 1: 2 FPGAs, 64 $\times$ 32 – A RBM with size 64×32 is divided into two separate 32×32 components.

To determine the hidden node states, both RBMC generate the energies corresponding to their visible nodes. These energy vectors are summed to obtain the partial energies required for node selection. After the node states are generated, they are transferred to both RBMCs, ensuring consistent hidden nodes.

Each RBMC can individually generate the partial energy terms required to determine their respective visible node states. No communication is required.

Case 2: 4 FPGAs, 64 $\times$ 64 – A RBM with size 64×64 is divided into four separate 32×32 components.

Following a protocol similar to Case 1, the RBMCs calculate their respective energies and send them to a summation core. The node states are determined from

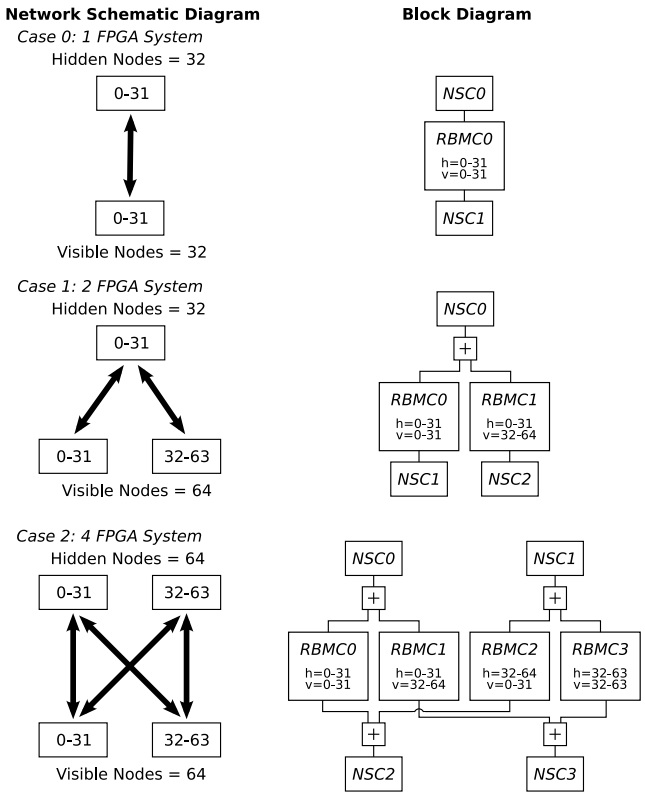

Fig. 6. A graphical representation of the partition algorithm.

the partial energies and are transferred to the RBMCs to ensure consistency. However, two sets of nodes must be calculated: hidden nodes 0-31 and 32-63.

The visible nodes are determined in a similar fashion. However, a different set of RBMC pairs are used to obtain the appropriate partial energies.

It should be noted that this partitioning algorithm provides a method to instantiate highly asymmetrical designs from strictly symmetrical RBMC components (Case 1).

This algorithm can be repeated to construct larger RBMs. Despite the communication overhead, the partitioning still provides a performance benefit if the communication consists of point-to-point links. This methodology maintains the $O(n)$ time complexity of the hardware RBM framework. As a result, the communication overhead of the repeated summation cores is negligible compared to the increase in performance obtained by instantiating a larger network.

4.2. Energy Accumulation Core

The Energy Accumulation Core (EAC), shown as "+" in Fig. 6, is a compute engine required for RBM partitioning. It receives the partial energies from two 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.

5. PERFORMANCE RESULTS

5.1. Metrics

There are two metrics for measuring the performance of a RBM implementation. The first metric is Connection Updates per Second (CUPS) – the rate at which a neural network can complete a weight update [7]. For a RBM, CUPS is defined as the number of weights, n^2, divided by the period for one AGS cycle, T (Eq. 15).

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

Also, the relative speed-up of the various hardware implementations compared to software implementations is required. The speed-up is defined as the ratio of CUPS for a given network size, Eq. 16.

$$S = \frac{\text{CUPS}_{\text{hw}}}{\text{CUPS}_{\text{sw}}} \tag{16}$$

5.2. Software Implementation

The software implementation was based on a modified version of the benchmark in [2]. It is a C benchmark compiled with gcc version 4.3.3 with the flags *-O2 -msse2*. The weights and partial energies used fixed-point representation. For the sigmoid function, the energies were cast as floating-point numbers and the *math.h* standard library was used. A software implementation of the Tausworth-88 random number generator was used since it is faster than the *stdlib.h* library and provided a fair comparison [6]. Other than the precision errors in the sigmoid function, the FPGA and software implementation produce the same results.

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 200kB of memory – which fits in the 512kB L2 cache.

5.3. Hardware Implementation

The hardware implementation was tested on the Berkeley Emulation Engine 2 (BEE2). 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 [8].

Two sets of hardware benchmarks were measured. The first test ran on a single FPGA with a standalone PowerPC processor. The standalone processor initialized the data vectors and was running at 300MHz. The remainder of the compute engines were running at 100MHz. This benchmark was used to measure the performance benefit of the hardware sigmoid function and it was also used to provide a baseline to analyze the communication and infrastructure overhead of the multi-FPGA system.

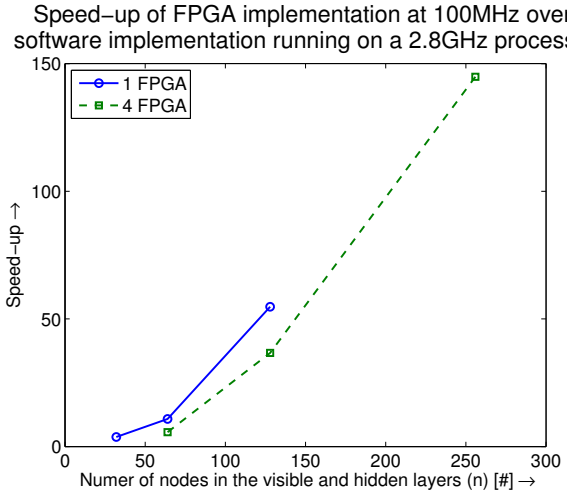

Fig. 7. The speed-up of the FPGA designs over the software implementation.

The second benchmark used four FPGAs. One PowerPC running at 300MHz was used to initialize the data vectors for the compute engines. The compute engines were configured as outlined in Case 2 of Fig. 6, with one of each hardware cores (RBMC, NSC and EAC) on a single FPGA.

For both benchmarks, single FPGA network sizes of $n = \{32, 64, 128\}$ were instantiated, allowing a maximum network size of 256×256 for the four FPGA system.

5.4. Results

The respective speed-up for various network sizes of the two benchmarks are shown in Fig. 7. The 128×128 single FPGA performance is measured at 1.58GCUPS, while the 256×256 four FPGA performance is measured at 3.13GCUPS, with speed-ups of 61-fold and 145-fold, respectively.

Comparing the results of the single FPGA implementation with the results in [2], the hardware implemented stochastic node selection provides a significant speed-up over its software counterpart. The relative performance penalty of the sigmoid cumulative distribution function in software is substantial, while the hardware implementation only results in a few extra cycles of computation.

Both benchmarks show that the implementations are able to maintain the reduction in time complexity achieved by the RBMC. Although, the communication overhead is relatively insignificant, the best performance is still achieved by synthesizing the largest RBM on a single FPGA – distributing RBMs across multiple FPGAs should only be used to extend large RBM designs that do not fit on a single chip.

Although a multi-FPGA distribution is presented, it is not a complete solution for implementing very large RBM designs as an unfeasible number of FPGAs will be required and the interchip communication is not expected to scale. The partitioning of the RBM is designed as a first step towards implementing very large RBM designs. Future work includes virtualizing the FPGA resources to implement very large RBM designs on a few FPGAs.

6. CONCLUSION

This paper shows two different components required in producing large, stochastic Restricted Boltzmann Machine FPGA implementations. First, a general method for creating a high-precision, pipelined function generator is introduced. Using a complement of Look Up Tables and the kth Stage Piecewise Linear Interpolator, a pipelined sigmoid function is generated with 8 cycles of latency and a maximum error of 3.36E-4, resulting in a precision of 11 bits. The sigmoid function is then applied in a stochastic Node Selection Core capable of selecting one node state per cycle. Next, an algorithm for partitioning large Restricted Boltzmann Machines into smaller networks is presented. By taking advantage of weight locality and sharing partial energies and node states, a low communication multi-FPGA system can be achieved. On the Berkeley Emulation Engine 2 running at 100MHz, a RBM of 256×256 nodes is synthesized and a computational speed of 3.13GCUPS was achieved resulting in a speed-up of 145-fold over an optimized C program running on a 2.8GHz Intel Pentium 4 processor. This multi-FPGA implementation demonstrates that large RBMs can be partitioned, resulting in additional performance.

7. REFERENCES

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

[2] D. Ly and P. Chow, "A High-Performance FPGA Architecture for Restricted Boltzmann Machines," *ACM International Symposium on FPGAs*, pp. 73–82, 2009.

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

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

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

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

[7] Y. Liao, "Neural Networks in Hardware: A Survey," Santa Cruz, CA, USA, Tech. Rep., 2001.

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