A Large-scale Architecture for Restricted Boltzmann Machines

Sang Kyun Kim, Peter L. McMahon, Kunle Olukotun
Department of Electrical Engineering
Stanford University
Stanford, California, United States
{*skkim38,pmcmahon,kunle*}*@stanford.edu*

***Abstract*—Deep Belief Nets (DBNs) are an emerging application in the machine learning domain, which use Restricted Boltzmann Machines (RBMs) as their basic building block. Although small scale DBNs have shown great potential, the computational cost of RBM training has been a major challenge in scaling to large networks. In this paper we present a highly scalable architecture for Deep Belief Net processing on hardware systems that can handle hundreds of boards, if not more, of customized logic with near linear performance increase. We elucidate tradeoffs between flexibility in the neuron connections, and the hardware resources, such as memory and communication bandwidth, required to build a custom processor design that has optimal efficiency. We illustrate how our architecture can easily support sparse networks with dense regions of connections between neighboring sets of neurons, which is relevant to applications where there are obvious spatial correlations in the data, such as in image processing. We demonstrate the feasibility of our approach by implementing a multi-FPGA system. We show that a speedup of 46X-112X over an optimized single core CPU implementation can be achieved for a four-FPGA implementation.**

***Keywords*-Accelerators; Neural network hardware; Computer architecture; Large-scale systems; Field programmable gate arrays; Parallel processing; Boltzmann machines;**

I. Introduction

A Deep Belief Network (DBN) is a multilayer generative model that is trained to extract the essential features of the input data by maximizing the likelihood of its training data. DBNs have recently gained great popularity in the machine learning community due to their potential in solving previously difficult learning problems. Introduced in 2006 by Hinton et al. [1], DBNs use Restricted Boltzmann Machines (RBMs) to efficiently train each layer of a deep network. DBNs have been successfully demonstrated in various applications, such as handwritten digit recognition [1] and human motion modeling [2].

Although DBNs appear to be a promising tool, investigations are limited by the significant amount of processing that RBMs require; current state-of-the-art implementations, including those for multi-core CPUs and GPUs, have long running times even for relatively small nets [3]. The primary issue is that conventional processors do not efficiently exploit the fine grain parallelism present in RBM training algorithms, which are dominated by large matrix multiplications. Graphics processors have significant performance benefits over CPUs in matrix operations, but do not scale to very large networks due to I/O bandwidth limitations, which we discuss in Section IV.

We seek to address this problem in both the short and long term by developing a scalable, highly optimized custom computer architecture for DBN processing. In the near term, systems implementing this architecture in field programmable gate arrays may be able to achieve considerable speedups over conventional CPUs. Longer term, we believe that future generations of many-core processors may contain cores that are optimized for specific classes of applications. An architectural exploration in this area using our ideas could lead to future processors that are better suited to DBN processing.

We show the trade-offs of the network configuration and hardware resources, such as memory and IO bandwidth, using which it will be possible to build a custom DBN network that is best suited for a particular application. To validate our claims in this paper, we implemented a multi-FPGA RBM system that can process 256 neurons per FPGA.

II. Related work

There has been considerable interest in accelerating the training of neural networks using customized hardware. In 1992, Cox and Blanz [4] demonstrated an FPGA implementation of a layered neural network for performing classification tasks. In 1994, Lysaght et al. [5] showed that dynamic reconfiguration of FPGAs could be used to train larger layered networks. Zhu and Sutton [6] provide a survey of FPGA implementations of neural networks, trained using backpropagation. Graf et al. [7] introduced a single FPGA design optimized for support vector machine training and convolutional neural network processing. Using systolic array for neural networks have also been explored to exploit the parallelism in neural networks [8].

Since the introduction in 2006 of Hinton et al.'s fast learning algorithm [1] for DBNs (Deep Belief Nets), there has been renewed interest in neural networks. Ly and Chow [9] introduced an FPGA architecture for training DBNs. Our previous work [10] improved the single FPGA architecture by generalizing the data representation and adding runtime flexibility of major learning parameters. Ly and Chow extended their work to multiple FPGAs [11]. However, their

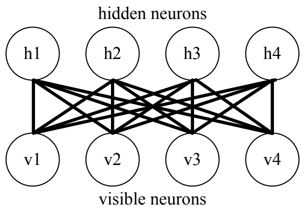

Figure 1. Illustration of an RBM network.

architecture will be challenging to scale to large networks due to their communication resource requirements.

III. Scalable Restricted Boltzmann Machine architecture

The Restricted Boltzmann Machine is the key component of DBN processing, where the vast majority of the computation takes place. As shown in ref. [10], matrix multiplication is responsible for more than 99% of the execution time for large networks. In this section, we briefly explain the RBM training algorithm and describe how previous single FPGA implementations accelerated the algorithm. Then we explore how we can extend the single FPGA implementation to a very large scale architecture for DBN processing. We investigate the trade-offs in designing a large scale DBN processing system, which involves the network structure and the hardware resources. Finally, we demonstrate how this architecture can be used to support sparse networks in a limited way, since exploiting the sparsity can significantly decrease the amount of computation and communication required.

A. RBM training algorithm summary

Here we briefly summarize the algorithm by Hinton et al. [1] for training RBMs, which we seek to accelerate.

RBMs are probabilistic generative models that are able to automatically extract features of their input data using a completely unsupervised learning algorithm. RBMs consist of a layer of hidden and a layer of visible neurons with connection strengths between hidden and visible neurons represented by an array of weights (see Figure 1). To train an RBM, samples from a training set are used as input to the RBM through the visible neurons, and then the network alternatively samples back and forth between the visible and hidden neurons. The goal of training is to learn visible-hidden connection weights and neuron activation biases such that the RBM learns to reconstruct the input data during the phase where it samples the visible neurons from the hidden neurons.

Figure 2 shows the pseudo-code for the RBM training algorithm. Each sampling process is essentially matrix-matrix multiply between a batch of training examples and the weight matrix, followed by a neuron activation function, which in many cases is a sigmoid function $(1/(1+e^{-x}))$.

```
-Visible neurons initially set to a batch of training examples, denoted vis_batch_0
-Repeat until convergence {
   1) Sample hid_batch_0 from P(h|vis_batch_0)
      a) tmp_matrix_1 = vis_batch_0 * weights
      b) tmp_matrix_2 = tmp_matrix_1 + hid_biases
      c) tmp_matrix_3 = sigmoid(tmp_matrix_2)
      d) hid_batch_0 = tmp_matrix_3 > rand()
   2) Sample vis_batch_1 from P(v|hid_batch_0)
   3) Sample hid_batch_1 from P(h|vis_batch_1)
   4) Update parameters:
      a) weights += $\alpha$(vis_batch_0$^T$*hid_batch_0 - vis_batch_1$^T$*hid_batch_1)
      b) vis_biases += $\alpha$(vis_batch_0$^T$*1 – vis_batch_1$^T$*1)
      c) hid_biases += $\alpha$(hid_batch_0$^T$*1 – hid_batch_1$^T$*1)
}
```

Figure 2. RBM training algorithm pseudo-code.

The sampling between the hidden and visible layers is followed by a slight modification in the parameters (controlled by the learning rate α) and repeated for each data batch in the training set, and for as many epochs as is necessary to reach convergence.

B. Previous single FPGA implementation overview

The RBM algorithm computation time is dominated by matrix-matrix multiplication of batches of neuron values by a weight matrix W. In addition, a significant amount of time may also be spent in the matrix-matrix multiplication of the visible neuron batches V and hidden neuron batches H in the weight update phase. To fully exploit the abundant parallelism in matrix multiply operations, a hardware RBM implementation should maximize the number of multipliers that can be supported by the memory bandwidth and logic gate resources, while reserving resources for other computation such as adders or the sigmoid function.

Previous work on accelerating the RBM algorithm using custom hardware includes single FPGA implementations [9][10]. Both approaches utilize the large embedded memory of recent FPGAs to fit the weight matrix on-chip. The main challenge in designing a single FPGA RBM accelerator is deciding how to perform the matrix transpose. The training algorithm requires three matrix-matrix products: VW, HW^T, and V^TH. Suppose we have M multipliers. To fully utilize the M multiplier resources, M weights must be read from the memory blocks each cycle.

We have previously demonstrated [10] a scalable approach that involves changing the ordering of the matrix multiply operations for different computation phases such that only row vectors of the weight matrix need to be accessed. The cost for this is the use of twice as many adders. The matrix multiplication HW^T can be performed as a collection of vector inner products, where the *row* vectors of the weight matrix W are accessed each cycle. To obtain the product VW, W^TV^T is computed instead, which then is viewed as a collection of vectors, each of which is a linear combination of the column vectors of W^T, with V^T as the coefficents. Thus, only columns of W^T (rows of W) need to be read from memory, which are all multiplied by an element of a row vector from V each cycle.

Another benefit of this approach is that the matrix product $V^T H$ in the weight update phase can reuse the structure for multiplication VW. Notice the memory bandwidth requirement for visible neurons is one neuron per cycle to be broadcasted, while the bandwidth requirement for a hidden neuron computation is a row vector of the hidden neuron batch H. If we view $V^T H$ as the sum of the outer products of each row vector of V and H, we can broadcast the visible node each cycle, which is multiplied by a row of hidden neurons to get the outer product. We can then sum the outer products using the existing accumulator. Since the matrix multiply structure for weight updates is basically the same as the hidden neuron computation phase, we mainly focus on the two multiplications VW and HW^T for the remainder of this paper.

C. Extension to multiple chip architecture

Ly and Chow [11] recently proposed an extension from the single FPGA architecture to a multi-FPGA system, where a partitioning algorithm is used to distribute the work amongst multiple FPGAs while minimizing the communication. However, the inter-chip network requires communication resources that increase quadratically with the number of neurons, which makes it difficult to scale to large networks. Therefore, instead of focusing on minimizing the amount of communication, we localize the communication to allow scalability.

The modular approach in our previous work [10] divides the work into multiple groups, localizing most operations such as matrix multiplication, the sigmoid function, and weight updates. Localization enables the user to easily migrate the same design to a future technology and take advantage of the technology by adding more modules. The few operations that do require global communication are appropriately buffered to avoid long wiring.

Although our modular approach is scalable within a chip, we cannot directly extend it to multiple-chip systems. The main issue is in how to deal with the global communication across the chips, which includes visible neuron broadcast in the hidden neuron computation phase and tree add reduction for the visible reconstruction phase.

Figure 3 illustrates our multi-FPGA system architecture for a network of three FPGAs. Our novel design builds on top of our previous single FPGA architecture. We see no inherent obstacles to extending our design to tens or hundreds of FPGAs (although of course there will be practical challenges that arise when constructing such large systems).

Our key insight is that it is possible to distribute the computation across multiple FPGAs and only require two nearest-neighbour communication links between FPGAs, including a connection from the first FPGA to the last FPGA.

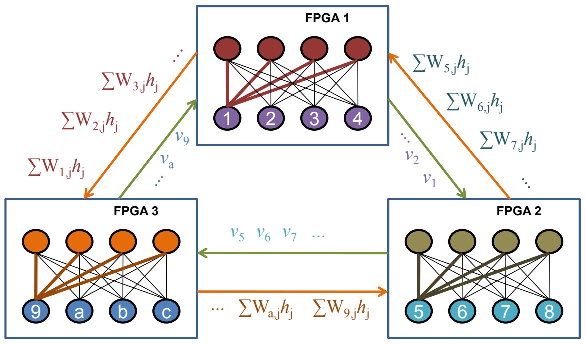

Figure 3. A high-level view of our architecture illustrating the inter-chip communications of a system with three FPGAs. Communications for both the visible and hidden neuron update phases are shown.

Let's first consider the hidden node computation phase. In the single FPGA implementation, visible neurons are broadcasted during the computation of the hidden neurons. However, broadcasting to multiple FPGAs would severely limit the scalability of our system. Thus, broadcasting the neurons is only done within the FPGA (with the appropriate buffering). Instead of broadcasting to all FPGAs, each FPGA passes the visible neurons it has read or received to its neighbor in one direction. To avoid any initial idle cycles waiting for data, the FPGA first reads from the local memory its portion of visible data and multiplies it with the appropriate weights, illustrated as the bold lines in Figure 4(a). Meantime each FPGA passes the visible data it had processed and consumes new incoming visible data as shown in Figure 4(b) until all the visible data has completely traversed the ring.

Reconstruction of the visible neurons on multiple FPGAs is done in a similar manner. Visible neuron computation requires a global add reduction. If we were to implement the global add reduction across multiple FPGAs with a similar method used in the single FPGA implementation, then either the connections between the FPGAs would need to be almost all-to-all, or the global reduction would have to be performed and transferred at a slow rate from shared wire contention, limiting the overall performance. Instead of performing the global add reduction all at once, we have each FPGA calculate the partial reduction for its final destination FPGA and pass this result to the neighboring FPGA. Figure 4(c) and Figure 4(d) illustrates how the partial reductions are passed to neighboring FPGAs. In Figure 4(c), each FPGA starts by computing the partial reduction for the furthest FPGA and passes its result to its neighbor. Then each FPGA in Figure 4(d) computes the partial sum for the next furthest FPGA and adds it to the incoming partial sum. This continues until the partial sums add up to be complete at the final destination FPGA, where the visible node is reconstructed.

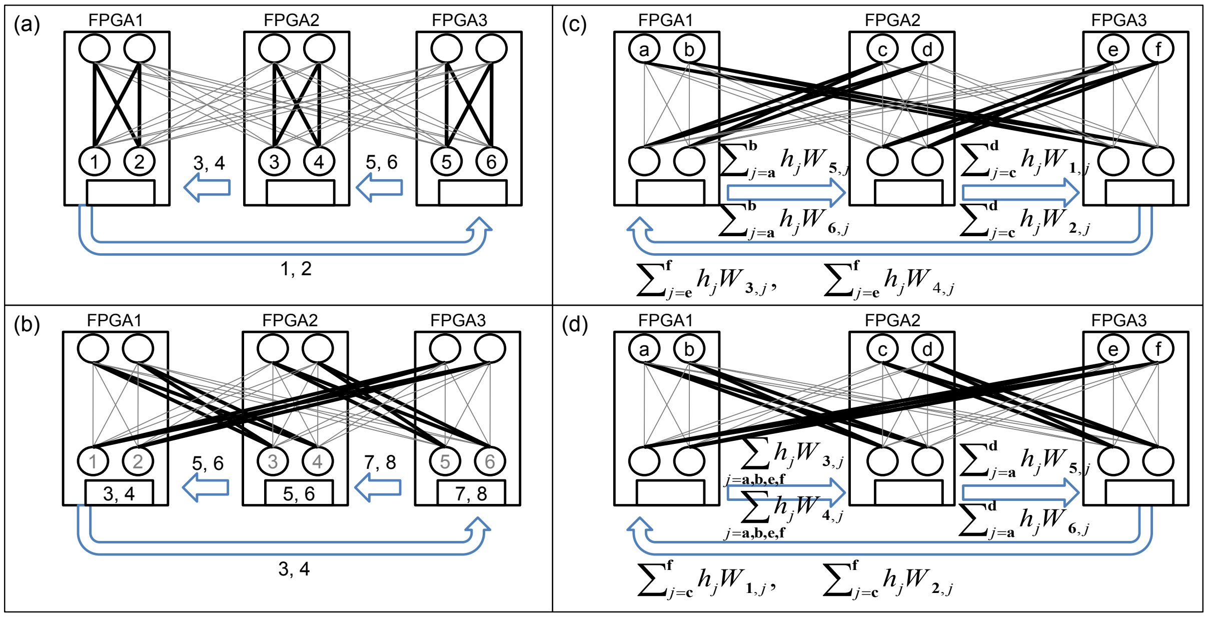

Figure 4. Simplified version of the communication between FPGAs

Since the hidden neuron computation requires one visible neuron broadcast per cycle, the I/O bandwidth requirement between the FPGAs is determined by the fact that each FPGA need only send one visible neuron to a neighbor per cycle, or less if there are more neurons than the number of multipliers per FPGA. The partial sums from visible data reconstruction also only require one partial sum per cycle, which in our system is 24 bits wide. Communications are only in one direction during a particular phase of the computation, although the direction of communication changes periodically (hence a physical interconnect that can only support a single direction is not sufficient, but a full duplex link is not necessary).

The abovementioned proposal does not consider the communication latency. However, off-chip communication does have a certain amount of latency. To tolerate this latency, each FPGA must process its local data first while data is buffered in its input queue. Since the communication is always nearest-neighbor, the latency only has to be less than the batch size to keep the RBM pipeline full.

The design of the parallel computation that only requires a ring topology for connecting FPGAs has several advantages. Modern FPGAs have significant off-chip IO bandwidth by providing many pins that can be clocked at high frequencies. However, the number of pins is limited, so a ring topology, as opposed to one with a higher number of connections from each chip, is one of the few that allows the logical connections to be implemented directly as physical connections. This enables both higher bandwidth, and is cheaper — solutions involving high-bandwidth switches (such as 10GbE) can be prohibitively costly.

D. *Memory bandwidth vs IO bandwidth trade-off*

A major issue that was not addressed in Section III-C is that the weight matrix is assumed to fit on-chip. For a single FPGA implementation, that assumption is reasonable as modern FPGAs provide significant amounts of on-chip memory. However, the weight matrix grows as $O(N^2)$ where N is the number of neurons. As we increase the number of FPGAs, the number of multipliers increases proportionally, thus the number of neurons that can be computed per cycle also grows linearly. However, the embedded memory also only increases linearly, while the weight matrix increases quadratically.

To overcome this issue and scale to large systems, the weight matrix must be streamed in from off-chip memory. Suppose we have 256 multipliers in a chip. To fully utilize the multipliers, we need to stream in 256 weights per cycle. Assuming a 200MHz RBM with 16-bit precision weights, the required bandwidth is around 100GB/sec, which is beyond the limits of current FPGAs.

One way to alleviate the memory bandwidth problem is to exploit the additional parallelism in the matrix multiplication by processing multiple visible training examples at once. This allows each weight to be fed into several multipliers rather than just one. Figure 5 compares the case of processing one training example per cycle versus two training examples per cycle. The two training examples case reduces the number of weights needed per cycle while still utilizing all the multipliers. The trade-off is that two visible neurons instead of one need to be read from the local memory. Given a fixed on-chip memory size, it becomes a trade-off between the number of neurons to be processed each cycle in the

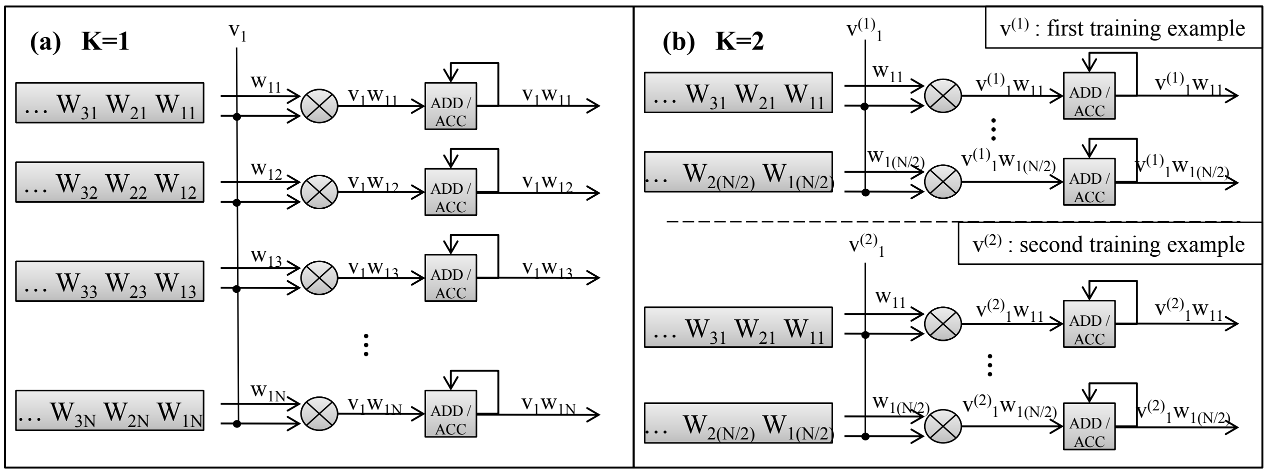
(a) K=1
v_1
W_{11}
v_1W_{11}
ADD / ACC
$\ldots W_{31}\ W_{21}\ W_{11}$
$\ldots W_{32}\ W_{22}\ W_{12}$
$\ldots W_{33}\ W_{23}\ W_{13}$
$\ldots W_{3N}\ W_{2N}\ W_{1N}$
(b) K=2
$v^{(1)}$: first training example
$v^{(2)}$: second training example
$v^{(1)}_1 W_{11}$
$v^{(1)}_1 W_{1(N/2)}$
$v^{(2)}_1 W_{11}$
$v^{(2)}_1 W_{1(N/2)}$
$\ldots W_{2(N/2)}\ W_{1(N/2)}$

Figure 5. Computation of processing (a) one input vector and (b) two input vectors per cycle

FPGA versus a larger batch size. However, as long as the number of neurons per FPGA does not fall below the number of multipliers, the speedup should remain approximately the same regardless of the number of neurons, since the speedup comes from the abundant multiplier resources.

Let us return to the previous example where 100GB/sec bandwidth was required. The bandwidth requirement can be significantly reduced to 6.25GB/sec by using a batch size $K = 16$. This bandwidth is achievable by using a DDR2 400MHz SDRAM which can supply 128 bits at 400MHz, i.e. 6400MB/sec.

By exploiting the parallelism in multiple training examples, we can dramatically decreases the DRAM bandwidth requirement. However, it may also increase the IO bandwidth requirements. Recall that each chip sends to its neighbor a visible neuron each cycle. However, since the RBM is processing multiple visible training examples at once, the IO bandwidth requirement linearly increases with the batch size. The same applies for partial sums in the visible computation phase. Thus, the total memory and IO bandwidth cost for M multipliers per FPGA is

$$M/K + K \qquad (1)$$

where the unit of the cost is 16 bits of data per cycle. To minimize the total bandwidth requirement, we simply set $M/K = K$, which leads to a batch size of $K = \sqrt{M}$. Table I illustrates the memory and IO bandwidth trade-off as K is changed. The number of multipliers were selected to be 256 and 1024 to reflect the latest FPGAs on the market. As can be seen in the table, only the memory bandwidth requirement depends on the number of multipliers. The IO bandwidth remains constant while we change the number of multipliers. However, the memory and IO bandwidth requirements both increase linearly with the clock frequency. Therefore, for a given communication capacity, it is generally more efficient to use FPGAs with more multipliers and reduce the clock frequency than attempting to gain performance by increasing the clock frequency.

Table I
TRADE-OFF OF K, MEMORY BANDWIDTH, AND IO BANDWIDTH AT 75MHZ AND 100MHZ

clock (MHz)	K	Mem (Gbps)	IO (Gbps)	clock (MHz)	K	Mem (Gbps)	IO (Gbps)
# of multipliers = 256							
75	4	76.8	14.4	100	4	102.4	19.2
75	8	38.4	28.8	100	8	51.2	38.4
75	16	19.2	57.6	100	16	25.6	76.8
# of multipliers = 1024							
75	4	307.2	14.4	100	4	409.6	19.2
75	8	153.6	28.8	100	8	204.8	38.4
75	16	76.8	57.6	100	16	102.4	76.8

In conclusion, increasing the batch size helps reduce the memory bandwidth requirement. However, it is limited by the on-chip memory space, and it also increases the IO bandwidth requirement. The right balance for each system depends on the specific values of chip-to-chip and memory bandwidth.

E. Locally dense sparse network

Although RBMs have all-to-all connections between the visible and hidden layers, it is unlikely that all the connections will be actively used. The training of RBMs for several applications where locality is important results in sparse representation of the weight matrix. We can exploit the sparseness of the weight matrix to increase the efficiency of computation.

One simple way to make use of the sparseness is to limit the fanout of each neuron to a constant number C. In addition, for simplicity, we restrict connections of each neuron

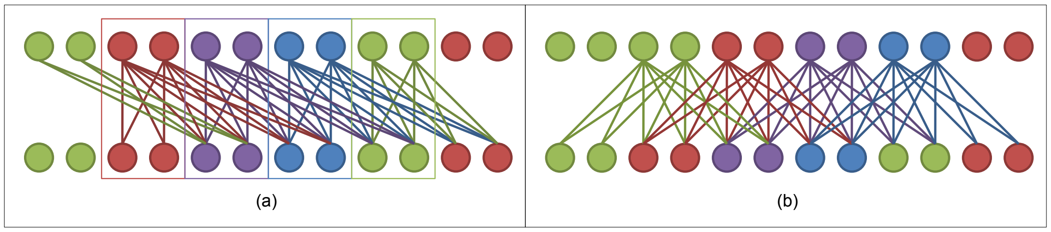

Figure 6. Locally dense sparse network (a) implementation (b) conceptual figure cycle

to be the C nearest neighboring neurons. Then the dense matrix, if any, will occur only at the neighboring nodes. This is overrestictive in the sense that the algorithm does not necessarily converge to such locally dense representation in general. However, we believe that such setting fits well in applications where locality plays an important role such as in image recognition [12], etc.

This restricted sparse configuration also maps well to our architecture. If we set C to be a multiple of the number of neurons per chip, we only need to configure the number of chips the data is being passed down before stopping the computation phase. Figure 6 illustrates this approach. The boxes in Figure 6(a) represent the chip border and the circles are the neurons. The left and right ends of the network are wrapped around such that all nodes have constant fanout. Figure 6(b) is an equivalent diagram of the network in Figure 6(a) leaving out the chip borders. As shown in Figure 6(a), the two layers are not completely connected to each other, but have a constant fanout of $C = 2 \times 3 = 6$. Although the network shown in Figure 6 is not itself "sparse", conceptually it explains how a locally dense sparse network will operate when many chips are connected to the system. By adding this one control, we can easily implement the locally dense sparse network.

Another reason the locally dense sparse network is attractive is that all-to-all dense RBMs are costly to scale due to the $O\left(N^2\right)$ memory requirement for storing the weight matrix. For a locally dense sparse network, the weight matrix can be considerably smaller, since the weight matrix grows as $O\left(N\right)$ with the number of neurons due to constant fanout. This allows us to build a very large scale network, approaching the scale of a human brain.

IV. Experimental results

To demonstrate the validity of our architecture, we conducted experiments on four FPGA boards connected to each other to form a ring network. The following subsections describe the experiment details and discuss the results.

A. Experimental platform and implementation details

We implemented the multi-FPGA system using a development board (Terasic DE3) that has an Altera Stratix III FPGA and a DDR2 SDRAM SODIMM interface. The Stratix III EP3SL340 has 135,000 ALMs (Adaptive Logic Modules; ALMs can be viewed as two 6-input ALUTs combined), 288 18x18 embedded multipliers, and 16,272 kbits of embedded RAM. The RBM module in each FPGA runs at 150MHz and uses all the available multipliers. Our current implementation does not support streaming weights from DRAM, thus we instead utilize the internal memory in each FPGA as was the case in the single FPGA implementation. Since the bandwidth of the embedded memory is extremely large, we adjust the K value to be 1 so that only one visible neuron is sent per cycle. In our development board, we used four LVDS pairs, resulting in a total datarate 4.8 Gbps per direction, to transmit one 24-bit partial-sum or one 16-bit neuron value every cycle for an RBM module running at 150MHz. However, based on the on-chip memory capacity and available hardware resources, only up to four FPGAs may be connected in a ring structure to support the weight matrix. In addition, the on-chip memory on all four FPGAs can store up to a total of 1M elements, which only allows a 256x256 configuration for each FPGA, so the flexibility to adjust the number of neurons as described in [10] is lost. To overcome these issues, future work involves developing the weight streaming logic of RBM modules to better utilize the FPGAs' fast DRAM and IO interfaces.

The four FPGAs are programmed with the Altera's soft processor (Nios II), a DDR2 memory controller, and an RBM module. Nios II receives input from the user, reads data from the SD card to store in DRAM, controls the overall flow of the RBM module, and displays the internal state of the RBM module. Future work may include adding a 1G Ethernet interface to stream in visible data in case the training set is too large to be stored in DRAM.

B. Evaluation

To evaluate our implementation, we used the FPGA boards to train on the MNIST handwritten digits dataset.

To verify that the results are indeed correct, we modified a reference MATLAB implementation from Hinton et al. [1] to a 16-bit fixed point version so that we may compare the output of the algorithm given the same input. The implementation was considered correct only when both the hardware and software implementations gave the same hidden variables, weights, and biases given the same input.

Table II
PERFORMANCE OF SINGLE CORE CPU, GPU, AND MULTI-FPGA

(a) Batch size K=100		Network size		
		768x768	1024x1024	1024x1024(s)
CPU	runtime (s)	3578.52	5424.8	4332.91
	Gmult/s	2.47	2.90	1.82
GPU	runtime (s)	152.91	236.13	185.10
	speedup	23.40	22.98	23.41
	Gmult/s	57.85	66.61	42.49
FPGAs	runtime (s)	76.97	102.58	51.36
	speedup	46.49	52.88	84.37
	Gmult/s	114.95	153.33	153.13

(b) Batch size K=16		Network size		
		768x768	1024x1024	1024x1024(s)
CPU	runtime (s)	4582.09	7863.75	5755.13
	Gmult/s	1.93	2.00	1.37
GPU	runtime (s)	309.87	569.11	522.46
	speedup	14.79	13.82	11.02
	Gmult/s	28.55	27.64	15.05
FPGAs	runtime (s)	76.97	102.58	51.36
	speedup	59.53	76.67	112.07
	Gmult/s	114.95	153.33	153.13

Performance was compared against a 2.3GHz Intel Xeon E5345 processor and a NVIDIA GeForce GTX 275 GPU, which has 240 CUDA processor cores running at 1.4GHz. The RBM module, written in C++, used the GotoBLAS2[13] library and the NVIDIA CUBLAS library for optimized matrix operations. However, these BLAS libraries support only floating point numbers, so we used the single precision BLAS routines for a fair comparison.

Using the four FPGA boards available to us, we tested our multi-FPGA prototype system with the following combinations of networks: dense 768x768, dense 1024x1024, and "sparse" 1024x1024 with connections between two neighboring boards. Although these are not large networks, they are sufficient to demonstrate our architecture in practice.

The experiments were conducted on a fixed number of 50 epochs, and on two batch sizes, 16 and 100. For the multi-FPGA implementation, a batch size of 100 was not tested since the on-chip memory cannot hold both large batches and the weight matrix at the same time. The sparse RBM was done in CPU and GPU by only computing the required operations. Table II shows the results of the 768x768, 1024x1024 dense network and of the 1024x1024 locally dense sparse network.

Runtime was measured to compare the speedup of the GPU and multi-FPGA systems against a single CPU core. The average number of multiplications per second (mult/s) was used as a universally comparable performance metric that does not depend on the problem size; this is similar to the commonly used metric CUPS (connection update per second), but mult/s provides an independent measure that does not depend on the batch size.

Table III
SCALABILITY: SPEEDUP AGAINST ONE NODE OF EACH PLATFORM

Platform	Number of Nodes		
	2	3	4
CPU (768x768)	1.59	1.80	2.14
GPU (1536x1536)	1.64	0.61	0.62
FPGA	2.01	3.02	4.03

As seen in Table II(b), the CPU and GPU perform poorly with a batch size of 16. This is because small batch sizes tend to perform less well in the SIMD and graphics operations since matrix-vector multiplication does not provide enough parallelism. A batch size of 16 is sufficiently small that this lack of available parallelism becomes apparent in the results. However, the multi-FPGA implementation does not exhibit this problem since the overhead to initiate a matrix multiplication is very small and always uses all the multipliers to achieve maximum performance.

This is a practical advantage for the multi-FPGA architecture since smaller batches may require fewer epochs to converge, so small batch sizes may be favored in practice by end users. Our experiments show that the error level at epoch 50 for batch size 100 can be achieved with only 36 epochs for batch size 16. Thus, if we were to run the algorithm until a certain error rate is reached, then the speedup for the multi-FPGA implementation is even higher.

It is important to notice that the networks we are experimenting with are not large scale. Since graphics processor tend to perform better with larger matrices, the speedup may differ when scaled to larger networks. Therefore, Table II is only a reference to show how our design's performance scales with problem size. In fact, if we increase the network size to 3072x3072, then the graphics processor shows around 52.7X speedup compared to the Intel Xeon processor. A comprehensive large scale comparison is the subject of future work.

Table III illustrates how CPUs, GPUs, and our FPGA architecture scales with the number of nodes. Four CPU machines, using one core per node, and two GPU machines, with two NVIDIA GTX 275 cards each, were fully connected via a Gigabit Ethernet switch to perform the scalability test. OpenMPI was used for communication, and the data was carefully distributed to minimize communication. Network sizes for CPUs and GPUs were chosen such that the matrices were not too small to be inefficient, but not too large to cause overwhelming communication overhead. Our FPGAs, on the other hand, currently have a fixed configuration of 256 neurons per node, so the network size varies with the number of boards (768x768 for 3 FPGAs, 1024x1024 for 4 FPGAs). The speedups in Table III were calculated using the measured Gmult/s performance of each platform.

As shown in Table III, the CPU shows sublinear scalability as we increase the number of nodes. The GPU also showed a sublinear speedup for 2 nodes, but revealed a major performance loss when crossing machine boundaries as the number of nodes increases from two to three. FPGAs, on the other hand, showed good scalability up to four nodes, and are expected to scale well to a very large number. The energy-efficient nature of FPGAs, in addition to the scalability of our design, makes our approach desirable for large-scale DBN implementations.

V. Future Work

Our current prototype places limits on the size of the network due to the limited capacity of the on-chip memory. Future work will include a design that streams weights from DRAM, which also allows flexible network configuration as in our previous work [10]. In addition, our next step to significantly expand the size of networks that can be trained in practice is to implement our architecture with the latest FPGAs which have more multiplier resources and communication capacity. Due to the simplicity of our communication scheme, many existing reconfigurable computing platforms may be appropriate — a ring topology is simple to implement with almost any platform that provides point-to-point communication links.

VI. Conclusion

Research in large scale Deep Belief Nets has been difficult due to the computation intensive and memory intensive nature of the application. However, building on our previous single FPGA implementation, we were able to design an architecture that scales to a very large number of FPGA chips. In addition, we extend the potential scalability even further by exploiting the sparseness of particular applications. A four-FPGA implementation has been demonstrated to show the feasibility of our approach, and showed a 46X-112X speedup with linear scalability. We expect our scalable architecture (which requires communication resources that grow only linearly with the network size) can be used to tackle very large machine learning applications that may have previously been difficult to approach. This is in contrast to previous architectures, such as that in ref. [11], and naïve parallelizations, whose required communication resources scale with the square of the network size, and hence are infeasible to implement for large networks.

Acknowledgments

The authors would like to thank Lawrence McAfee for his assistance in obtaining DBN performance results on GPUs, and Jacob Leverich for his help with the CPU cluster infrastructure. SK gratefully acknowledges funding support from the Kwanjeong Educational Foundation. PLM gratefully acknowledges the financial support of a David Cheriton Stanford Graduate Fellowship. This work is supported by the Stanford Computer Science Department.

References

[1] G. Hinton, S. Osindero, and Y. Teh, "A fast learning algorithm for deep belief nets." *Neural Computation*, vol. 18, pp. 1527–1554, 2006.

[2] G. W. Taylor, G. E. Hinton, and S. T. Roweis, "Modeling Human Motion Using Binary Latent Variables," in *Advances in Neural Information Processing Systems 19*. MIT Press, 2007, pp. 1345–1352.

[3] R. Raina, A. Madhavan, and A. Ng, "Large-Scale Deep Unsupervised Learning using Graphics Processors," in *Proceedings of the 26th International Conference on Machine Learning*, L. Bottou and M. Littman, Eds. Montreal: Omnipress, June 2009, pp. 873–880.

[4] C. Cox and W. Blanz, "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 3, pp. 288–299, Mar 1992.

[5] P. Lysaght, J. Stockwood, J. Law, and D. Girma, "Artificial neural network implementation on a fine-grained FPGA," *Field-Programmable Logic Architectures, Synthesis and Applications*, vol. 849, pp. 421–431, 1994.

[6] J. Zhu and P. Sutton, "FPGA Implementations of Neural Networks: a Survey of a Decade of Progress," in *Proc. 13th International Conference on Field-Programmable Logic and Applications*, Sep. 2003, pp. 1062–1066.

[7] H. P. Graf, S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, and S. Chakradhar, "A Massively Parallel Digital Learning Processor," in *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds., 2009, pp. 529–536.

[8] D. Zhang and S. K. Pal, Eds., *Neural Networks and Systolic Array Design*. Farrer Road, Singapore: World Scientific Publishing Co. Pte. Ltd., 2002.

[9] D. Ly and P. Chow, "A high-performance FPGA architecture for restricted boltzmann machines," in *Proc. of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2009, pp. 73–82.

[10] S. K. Kim, L. C. McAfee, P. L. McMahon, and K. Olukotun, "A Highly Scalable Restricted Boltzmann Machine Implementation," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Sept. 2009.

[11] D. L. Ly and P. Chow, "A Multi-FPGA Architecture for Stochastic Restricted Boltzmann Machine," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, Sept. 2009.

[12] H. Lee, R. Grosse, R. Ranganath, and A. Ng, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations," in *Proceedings of the 26th International Conference on Machine Learning*, L. Bottou and M. Littman, Eds. Montreal: Omnipress, June 2009, pp. 609–616.

[13] K. Goto and R. Van De Geijn, "High-performance implementation of the level-3 BLAS," *ACM Trans. Math. Softw.*, vol. 35, no. 1, pp. 1–14, 2008.