A Fully Pipelined FPGA Architecture of a Factored Restricted Boltzmann Machine Artificial Neural Network

LOK-WON KIM, Cisco Systems
SAMEH ASAAD and RALPH LINSKER, IBM T. J. Watson Research Center

Artificial neural networks (ANNs) are a natural target for hardware acceleration by FPGAs and GPGPUs because commercial-scale applications can require days to weeks to train using CPUs, and the algorithms are highly parallelizable. Previous work on FPGAs has shown how hardware parallelism can be used to accelerate a "Restricted Boltzmann Machine" (RBM) ANN algorithm, and how to distribute computation across multiple FPGAs.

Here we describe a fully pipelined parallel architecture that exploits "mini-batch" training (combining many input cases to compute each set of weight updates) to further accelerate ANN training. We implement on an FPGA, for the first time to our knowledge, a more powerful variant of the basic RBM, the "Factored RBM" (fRBM). The fRBM has proved valuable in learning transformations and in discovering features that are present across multiple types of input. We obtain (in simulation) a 100-fold acceleration (vs. CPU software) for an fRBM having N = 256 units in each of its four groups (two input, one output, one intermediate group of units) running on a Virtex-6 LX760 FPGA. Many of the architectural features we implement are applicable not only to fRBMs, but to basic RBMs and other ANN algorithms more broadly.

Categories and Subject Descriptors: B.5.1 [**Register-Transfer-Level Implementation**]: Design

General Terms: Design

Additional Key Words and Phrases: Restricted Boltzmann Machine, FPGA based system design, hardware acceleration of Neural Network, pipelined and parallel hardware architecture

ACM Reference Format:
Kim, L.-w., Asaad, S., and Linsker, R. 2014. A fully-pipelined FPGA architecture of a factored restricted Boltzmann machine. *ACM Trans. Reconfig. Technol. Syst.* 7, 1, Article 5 (February 2014), 23 pages.
DOI: http://dx.doi.org/10.1145/2539125

1. INTRODUCTION

In recent years, there has been growing interest in Deep Belief Networks (DBN), introduced by Hinton et al. [2006], for their potential to solve difficult machine learning problems. DBNs are multilayer artificial neural networks, in which each layer of connections can be initially trained by a Restricted Boltzmann Machine (RBM) training algorithm. More recently, a more powerful variant of the RBM, the "Factored RBM" (fRBM), has been developed [Memisevic and Hinton 2010]. The fRBM is particularly well-suited for learning transformations between pairs of input patterns, learning

L.-w. Kim performed this work while an intern at IBM. He was at that time a graduate student in the Electrical Engineering Department, University of California.
Authors' addresses: L.-w. Kim, Cisco Systems, Inc., San Jose, CA 95134; email: knoublok@ucla.edu; S. Asaad and R. Linsker, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598; email: {asaad, linsker}@us.ibm.com.

DOI: http://dx.doi.org/10.1145/2539125

features that are invariant across multiple classes of input, and applying a set of learned transformations to novel input domains (a type of "learning by analogy").

The fRBM algorithm is computationally more expensive than the basic RBM, and, as we shall see, this is particularly so when each is implemented in special-purpose hardware and the nodes ("neurons") have binary-valued activities.

The need for hardware acceleration of such algorithms is many-fold. For modern machine learning problems of interest, network sizes may be of the order of thousands of nodes or more, rendering software-only solutions inadequate due to long training times (days to weeks). Parallel software implementations are not very efficient on multicore architectures due to the all-to-all communication nature of the algorithm when no locality is imposed. In many commercial systems, there is a need for embedding such networks in other systems, placing tight constraints on area, power, and physical size, and therefore requiring efficient hardware implementation of such algorithms. Finally, research in the ANN field itself will be greatly served by flexible, accelerated hardware that enables more rapid algorithm testing and development.

Accelerating the basic RBM, or portions of it, using reconfigurable hardware or general purpose GPUs has been the subject of recent articles. However, there has been to the best of our knowledge no reported FPGA implementation of an fRBM.

This article makes the following contributions.

(1) We present a reconfigurable hardware architecture that implements the fRBM algorithm.
(2) We present a design that exploits the use of "mini-batch" ANN training (common in software implementations) by using a fully pipelined parallel architecture that allows input data to be streamed while keeping the pipeline nearly full at all times. This design significantly increases performance and decreases intermediate-result memory requirements, by comparison with alternative ANN architectures that either operate on a single input case at a time [Ly and Chow 2010] or duplicate hardware to operate independently on multiple cases in parallel [Kim et al. 2010].
(3) We describe an efficient virtualized architecture that can simulate an fRBM that is too large to be handled by a given FPGA without multiplexing.

Many of the pipelining, parallelization, scaling, streaming, and other features of our design are applicable as well to basic RBMs and are, more broadly, useful for mapping ANN algorithms into reconfigurable hardware.

1.1. Hardware Acceleration of ANNs

There have been many publications that address methods to accelerate training of artificial neural networks [Dias et al. 2004; Lindsey and Lindblad 1994; Zhu and Sutton 2003]. For example, Boser and Sackinger [2002] exploited highly parallel special-purpose analog circuits to increase the training speed of neural networks. Jung and Kim [2007] created a hybrid hardware system using a DSP processor and customized function blocks on an FPGA to provide online learning and system control. FPGA ANN designs [Maeda and Tada 2003; Zhu and Sutton 2003] have used fine-grain parallelization; the highly parallel processing of graphics processing units (GPUs) has been used in Oh and Jung [2004]; and various commercial hardware solutions including dedicated ASIC chips have been developed [Dias et al. 2004]. Experiments on bit-width optimization and its effect on learning performance, which is a critical issue for cost-effective hardware implementation, have also been reported [Holt and Hwang 1993].

In recent years increasing attention has been directed to the acceleration of deep learning algorithms, in particular the basic RBM algorithm. Raina et al. [2009] applied the fine-grained parallelism of graphics processor units (GPU) to the basic RBM ANN. Ly and Chow [2010] investigated how the basic RBM can be mapped to an FPGA hardware architecture. They created dedicated hardware processing cores which were optimized for certain parts of the algorithm: an embedded processor, memory, and the Message Passing Interface (MPI) manage system controls, storing of intermediate results, and communication among the cores. Their experimental system, using a Xilinx FPGA (XC2VP70), integrated 128 neurons in each of two fully interconnected layers. The architecture in Kim et al. [2010] uses multiple RBM processing modules in parallel, with each module responsible for a relatively small number of ANN nodes. Their experimental system on an Altera FPGA (EP3SL340) integrated 256 neurons in each of two fully-interconnected layers.

The architecture we describe here for the Factored RBM algorithm is suitable for FPGA or ASIC implementation, and is extendable to multi-FPGA systems. Our experimental system, run on a Xilinx Virtex-6 FPGA (XC6VLX760), can integrate up to 256 nodes in each of the four sets of ANN units ("neuron" layers x, y, and h, and the set of "factors" f, as described in this article), with full interconnection between layer f and each of layers x, y, and h.

Our architecture exploits parallel operators and a coarse-grain pipeline to increase system throughput. The weight update operations are divided into twelve processing phases that comprise seven pipeline stages. The design is fully parametrized, so that setting network size parameters in a script automatically generates the required HDL code for FPGA configuration. In addition, using multiple instances of processing blocks or time-division multiplexed processing, the proposed architecture supports super-scalar or weight training operations for virtually increased node sizes.

1.2. Organization of This Article

The remainder of this article is organized as follows: Section 2 introduces the fRBM ANN algorithm. An architectural description of the proposed hardware solution is presented in Section 3. Section 4 describes the design methodology. Section 5 presents experimental results, and Section 6 concludes this article.

2. BASIC AND FACTORED RESTRICTED BOLTZMANN MACHINE ANNS

2.1. The Basic RBM

A basic RBM [Hinton et al. 2006] has a "visible" (v) input layer and a "hidden" (h) layer of nodes with symmetric-weight bidirectional connections between each v and each h node. The goal of training is for the network to learn (approximately) a maximum likelihood model of the input data. In so doing, the h layer comes to represent statistically important features of the data. For computational tractability, [Hinton et al. 2006] uses a "contrastive divergence" algorithm: in its simplest form ("CD-1") the network computes $v \rightarrow h \rightarrow v' \rightarrow h'$ and the weight updates $\Delta w_{ij} \propto (v_i h_j - v'_i h'_j)$.

Note that, for hardware implementation, the matrix "multiplications" required to train a basic RBM having binary-valued node activities involve only binary mask and accumulate operations, not true fixed-point multiplications.

2.2. The Factored RBM

The basic RBM learns statistical dependencies between two layers of units, v and h. However, for various applications it is important for the network to learn three-way dependencies. For example: What is the transformation T that relates two patterns

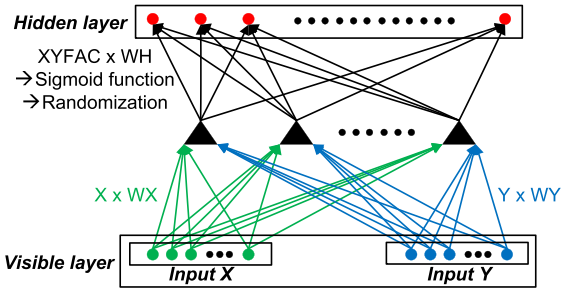

Fig. 1. A conceptual diagram of a fRBM ANN. Arrows show signal flow during an upward pass (they are reversed for a downward pass).

x and y within a pair, and into what y does T transform a previously unseen pattern x (a type of "learning by analogy")? What is the covariance structure that relates pairs of parts within a single image? How does the mapping that relates a pair of patterns x and y depend upon a gating vector h?

A network in which every triple of nodes (x_i, y_j, h_k) has a three-way connection with weight w_{ijk} was used as the basis of an RBM algorithm [Memisevic and Hinton 2007]. However, the structure of the data, as well as training efficiency, often favors factoring this tensor w into a sum over products of three two-way weight matrices, $\sum_f w^x_{if} \times w^y_{jf} \times w^h_{kf}$. In particular, the factored-weight form enables the learning of parameters that are shared among classes; that is, of features that are invariant across classes (e.g., spatial transformations that act similarly on faces, handwriting, etc.).

A "Factored RBM" learning algorithm was therefore developed [Memisevic and Hinton 2010]. It is more powerful than the basic RBM in that it can be trained on an ensemble of pairs of input vectors (call them x and y), and learn the transformation(s) that relates x and y. This enables more powerful feature learning and generalization to novel input sets. In this formulation, one pattern (x) of the pair is held constant while pattern y and the hidden nodes h are treated as the (visible, hidden) pair in a CD-1 computation: the network dynamics sequentially map $(x, y) \rightarrow h$; $(x, h) \rightarrow y'$; and $(x, y') \rightarrow h'$; and the differences between pairs of terms that are functions of (x, y, h) and (x, y', h'), respectively, are used to calculate updates to the three weight matrices. This is called a "conditional fRBM" to distinguish it from alternative fRBM applications in which, for example, x and y are constrained to be identical (for learning the covariance structure of an image X) and for which the training process is more complex [Ranzato and Hinton 2010]. In this article, we deal with the implementation of a conditional fRBM. The conditional fRBM has been successfully applied to learning styles of human movement [Taylor and Hinton 2009], learning of class-invariant features [Memisevic et al. 2010], and learning of image transformations [Memisevic and Hinton 2010].

In contrast to the basic RBM, each of several processing phases for a binary fRBM (i.e., having binary-valued input and output nodes x, y, h) does require a large number of fixed-point multiplications of two fixed-point values in parallel, contributing a substantial hardware load to the design.[1]

An fRBM network is shown in Figure 1. There are connections from each node of layers x and y to a layer of "factors," denoted by triangles, and connections from each

[1]For example, Eq. (1) is computed serially over p and f, and the products of the fixed-point scalar ($XFAC_{pf} \times YFAC_{pf}$) with the weights w^h_{kf} are computed in parallel for all k.

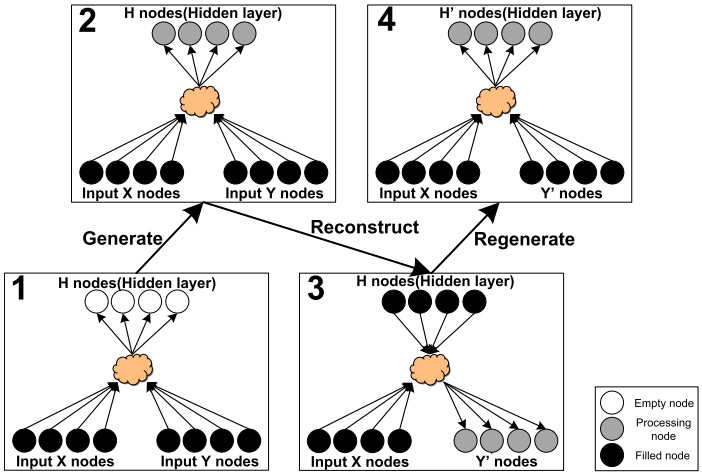

Fig. 2. Three step operations (Generate-Reconstruct-Regenerate) to compute H, Y′ and H′ node values.

factor to each node of layer h. In a binary fRBM (considered here), the activity of each x, y, and h node is either 0 or 1 at any given time. We will label each input pair (and the activities computed using it) by a "presentation" (or "case") index p. The activation rule for an upward pass from (x, y) to h states that each factor f computes the weighted linear combinations of inputs $XFAC_{pf} \equiv \sum_i x_{pi} w^x_{if}$ and $YFAC_{pf} \equiv \sum_j y_{pj} w^y_{jf}$, and outputs the product $(XFAC_{pf} \times YFAC_{pf})$ to each node k of layer h. The activity at the kth h-node is set to 1 with probability

$$P(h_{pk} = 1) = \sigma\left(\sum_f XFAC_{pf} \times YFAC_{pf} \times w^h_{kf} + w^{hb}_k\right), \tag{1}$$

and to 0 otherwise, where w^{hb}_k is the "bias weight" for node h_k and $\sigma(u)$ is a sigmoid-shaped function such as the logistic function $\sigma(u) \equiv 1/(1+e^{-u})$. During a "downward" pass from (x, h) to y (in which x remains unchanged), the activity of the jth y-node, now denoted y'_{pj}, is similarly set to 1 with probability $P(y'_{pj} = 1) = \sigma(\sum_f XFAC_{pf} \times HFAC_{pf} \times w^y_{jf} + w^{yb}_j)$ where $HFAC_{pf} \equiv \sum_k h_{pk} w^h_{kf}$. The second upward pass is computed using Eq. (1) with activities (y', h') replacing (y, h). The three passes are shown in Figure 2.

To compute the changes $\Delta_p w^h_{kf}$ in w^h_{kf} (and the changes in w^x and w^y) resulting from presentation p, we use the activation preceding rules to compute, for given vectors x_p and y_p : $h_{pk} \to y'_{pj} \to h'_{pk}$. Then, the learning rules for the connection weights w^h, w^x, and w^y, and the bias weights w^{hb} and w^{yb}, are:

$$\begin{aligned}
\Delta_p w^h_{kf} &= \epsilon\,[h_{pk} \times XFAC_{pf} \times YFAC_{pf} - h'_{pk} \times XFAC_{pf} \times Y'FAC_{pf}]\,,\\
\Delta_p w^x_{if} &= \epsilon\,[x_{pi} \times YFAC_{pf} \times HFAC_{pf} - x_{pi} \times Y'FAC_{pf} \times H'FAC_{pf}]\,,\\
\Delta_p w^y_{jf} &= \epsilon\,[y_{pj} \times XFAC_{pf} \times HFAC_{pf} - y'_{pj} \times XFAC_{pf} \times H'FAC_{pf}]\,,\\
\Delta_p w^{hb}_k &= \epsilon\,(h_{pk} - h'_{pk}),\\
\Delta_p w^{yb}_k &= \epsilon\,(y_{pj} - y'_{pj}),
\end{aligned} \tag{2}$$

where ϵ is a learning rate (chosen so that the weight changes are very small per presentation),[2] and $Y'FAC$ and $H'FAC$ are defined as are $YFAC$ and $HFAC$, but with y' and h' replacing y and h.

To reduce computing time, it is customary to average the RHS of the above rules over a subset (a "mini-batch", hereinafter referred to simply as a "batch") of presentations, and to update the weights only at the end of each batch. The overall structure of the training algorithm is then the following.

```
for epoch = 1:numEpochs
    for batch = 1:numBatches
        Activation: Given each input vector pair (x_p, y_p) in the batch,
                    compute h_p, y'_p, and h'_p.
        Learning: Compute Δw^h, Δw^x, Δw^y, Δw^yb,
                    and Δw^hb as the averages over p of Eqns. 2.
    end for-batch

  Optionally calculate diagnostics (e.g., reconstruction error
      Σ_{p,j,batches} (y_pj,batch − y'_pj,batch)^2).
end for-epoch
```

The pseudocode here (and the corresponding FPGA design) implements the above algorithm. In addition, our design (a) accelerates the processing of an entire batch by using a pipeline in which stalls have been minimized; (b) rearranges the computation of certain matrix multiplications to save memory and avoid substantial pipeline stall; (c) saves memory by reducing the number of intermediate results that must be reused; and (d) balances the cost of devoting separate hardware to each of the three operations in Figure 2. Phases 1–6 implement the activation step (step #3), using two phases each to compute h_p, y'_p, and h'_p. Phases 7–12 implement the learning (weight update) step #4.

We will use N_c to denote the number of presentations (cases) in each batch. We define N to be the maximum over the number of nodes in each of the four sets of nodes in a processing stage: x, y, h, and the factors f. It will be seen that, for each case, N time steps are required to process each of phases 1–6 in the pipeline, and N additional time steps are required to process all of phases 7–10 simultaneously. Phases 11 and 12 are executed just once per batch (with the last case) since it must await computation of intermediate results that require all cases in the batch to have been processed through phase 6.[3] Even though a phase may complete one case in fewer than N time steps (because the corresponding layer may contain fewer than N nodes), the pipeline structure requires waiting for each phase to be completed before its results are advanced to the next phase. This will be discussed further in conjunction with Figure 7.

During fRBM training (and ANN training in general), if the number of training presentations is not too large, the entire set is typically (but not necessarily) presented to the network during each epoch.

[2]Additional "weight decay" and/or "momentum" terms are often added to this and other ANN learning rules to improve convergence. These terms are omitted here for simplicity, but adding them to a hardware design is straightforward.

[3]The implementation used here for phase 11 (similarly for phase 12) requires N_c time steps. We assume that $2N_c < N$; this is typically the case, and can be assured since one is free to choose N_c for performance and convenience, whereas N is set by the ANN architecture.

Detailed pseudocode for the activation and learning steps is shown here for a single batch. The asterisk (*) in this pseudocode means a scalar multiplication which is represented as ($\times$) in other sections. The notation "SUM_i $\{\cdots\}$" means: compute $\{\cdots\}$ for each i in parallel, then sum over i using an adder tree. The weights WX, WY, WH, WYB, and WHB are initialized only at the start of the ANN simulation, and are updated during phases 8–12.

```
for every case p in batch:
 #each case takes N time steps for each phase and uses N processors

   # PHASE 1
   for every factor f (serially):
      XFAC[f] = SUM_i {x[p][i]*WX[i][f]}
      YFAC[f] = SUM_j {y[p][j]*WY[j][f]}
      XYFAC[f] = XFAC[f]*YFAC[f]
      for every h-node k (in parallel):
         if f=1
            tem[k] = XYFAC[1]*WH[k][1]
         else
            tem[k] += XYFAC[f]*WH[k][f]
         end if
      end for-k
   end for-f

   # PHASE 2
   for every h-node k:
      hprob[k] = logistic( tem[k] + WHB[k] )
      generate rand_value[k]
      if hprob[k] > rand_value[k]
         h[p][k] = 1
      else
         h[p][k] = 0
      end if
   end for-k

   # PHASE 3
   for every factor f (serially):
      HFAC[f] = SUM_k {h[p][k]*WH[k][f]}
      XHFAC[f] = XFAC[f]*HFAC[f]
      YHFAC[f] = YFAC[f]*HFAC[f]
      # Then:  same as for-k loop of phase 1, but replace (XYFAC, WH, k) by (XHFAC, WY, j)
               resp.
   end for-f

   # PHASE 4
   # Same as phase 2, but replace (k, WHB, hprob, h) by (j, WYB, y'prob, y') resp.

   # PHASE 5
   for every factor f (serially):
      Y'FAC[f] = SUM_j {y'[p][j]*WY[j][f]}
      XY'FAC[f] = XFAC[f]*Y'FAC[f]
      # Then: same as for-k loop of phase 1, but replace XYFAC by XY'FAC
   end for-f

   # PHASE 6
   # Same as phase 2, but replace (hprob, h) by (h'prob, h') resp.

   # PHASES 7-10 (all done simultaneously for the same case p)
   for every factor f (serially):

      # PHASE 7
      H'FAC[f] = SUM_k {h'[p][k]*WH[k][f]}
      XH'FAC[f] = XFAC[f]*H'FAC[f]
      Y'H'FAC[f] = Y'FAC[f]*H'FAC[f]
```

```
      # PHASE 8
      if p=1
         (Delta_WY)[j][f] = y[1][j]*XHFAC[f]-y'[1][j]*XH'FAC[f] #all j in parallel
      if p>1 and p<Nc
         (Delta_WY)[j][f] += y[p][j]*XHFAC[f]-y'[p][j]*XH'FAC[f] #all j in parallel
      if p=Nc
         WY[j][f] += (eps/Nc)*( (Delta_WY)[j][f]+y[Nc][j]*XHFAC[f]-y'[Nc][j]*XH'FAC[f]) #all
                     j in parallel
      end if

      # PHASE 9
      # Same as phase 8, but replace (Delta_WY, j, y, XHFAC, y', XH'FAC, WY) by
      # (Delta_WX, i, x, YHFAC, x, Y'H'FAC, WX) resp.

      #PHASE 10
      # Same as phase 8, but replace (Delta_WY, j, y, XHFAC, y', XH'FAC, WY) by
      # (Delta_WH, k, h, XYFAC, h1, XY'FAC, WH) resp.
   end for-f
end for-p

# Phases 11 and 12 are executed ONLY at the completion of each batch of Nc cases
# PHASE 11
for every y-node j (in parallel):
 #takes Nc time steps & N processors
   Delta_WYB[j] = y[1][j]-y'[1][j]
   for pp=2 through pp=Nc-1
      Delta_WYB[j] += y[pp][j]-y'[pp][j]
   end for-pp
   WYB[j] += (eps/Nc)*( Delta_WYB[j]+y[Nc][j]-y'[Nc][j] )
end for-j

# PHASE 12
# Same as phase 11 but replace (Delta_WYB, j, y, y', WYB) by (Delta_WHB, k, h, h',
  WHB) resp.

# END
```

After training is complete, the network may be run in execution-only mode, during which output activities h are computed as in phases 1 and 2, but no weight updates are performed, and y' and h' are not computed.

3. SCALABLE HARDWARE ARCHITECTURE OF FACTORED RESTRICTED BOLTZMANN MACHINE ARTIFICIAL NEURAL NETWORK

3.1. The Proposed Pipelined and Parallel Architecture

Matrix multiplication frequently used in the fRBM and other ANN algorithms is one of the highest hardware cost operators used in the algorithms. Accordingly efficient implementation of the operators is very important for reasonable resource utilization and performance enhancement. As shown in the pseudocode, the connection weights WX, WY, and WH, the batch of X and Y input cases, and intermediate results H, Y', H', etc. are each represented by a two-dimensional matrix in the simulation code. To increase throughput at reasonable hardware cost, the proposed hardware architecture uses a coarse-grained pipeline architecture and employs parallel units such as multiple adders, multipliers, and accumulators.

For the coarse grain pipeline architecture, we have divided the weight update algorithm into 12 processing phases which are represented in the pseudocode and Figure 3. Phases 1 and 2 ("Generate") compute the hidden layer node values H from the input values X and Y; phases 3 and 4 ("Reconstruct") compute Y' values (which tend to

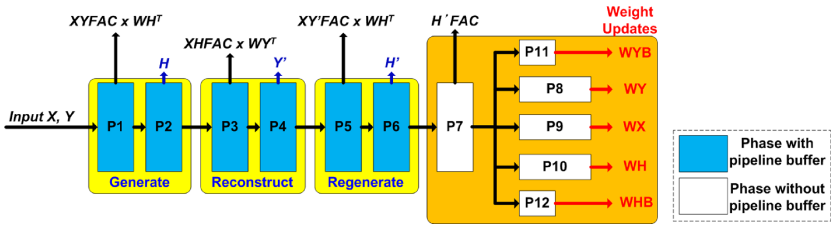

Fig. 3. Twelve phases of the proposed fRBM ANN hardware architecture for one weight update operation.

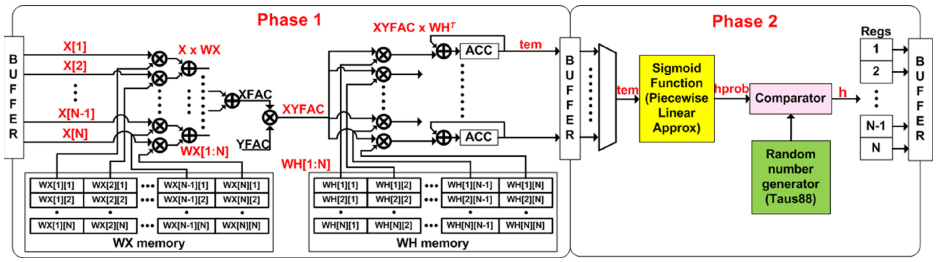

Fig. 4. A hardware block diagram of the phase 1 and 2 for the Generate step.

approximate Y as training proceeds); and phases 5 and 6 ("Regenerate") use Y' as the "input" for computing H'. Of these phases, the even-numbered ones implement the sigmoid function and the stochastic generation of binary node values.

Phase 7 computes terms needed for the weight updates. Phases 8, 9, and 10 accumulate (over the cases in a batch) the changes to be made to the weight values WX, WY, and WH, and phases 11 and 12 accumulate changes to the the bias weights WHB and WYB.

Phases 1–3 and 8–10 include the matrix multiplication operators, which consist of a vector multiplier and a tree of adders or a vector multiplier and accumulators. Each of phases 1–6 has a pipeline stage buffer for a coarse grain pipeline architecture to hold its stage results, since each of phases 2–7 has a data dependency on a previous stage. On the other hand, phases 8–12 can operate simultaneously, since they have no data dependency on one another.

After all phases have operated on all the cases within a batch, the accumulated weight changes are applied to the weights WX, WY, WH, WYB, and WHB, and these updated weights are used for the next batch of inputs.

3.2. Hardware Design of the Three Step Operations to Generate H, Y' and H'

The FPGA implementation of the fRBM algorithm follows closely the pseudocode outlined in Section 2.2. The data flow is dominated by three basic computations: matrix multiplication, the sigmoid function, and random number generation. Furthermore, the patterns of computation for each of the Generate, Reconstruct and Regenerate steps are similar. Figure 4 shows a detailed hardware block diagram for the Generate step, corresponding to phases 1 and 2 in the pseudocode. The result of this step is to update the values of the hidden nodes for a given presentation of the X and Y input sets.

Phases 1–10 are performed for each case p in turn.

As shown in Figure 4 and the pseudocode, the first phase comprises the following steps: Serially for each f, the partial products $x[p][i] \times WX[i][f]$ are computed in parallel for all i using a vector multiplier, then summed over i by an adder tree

to produce $XFAC[p][f]$. To produce this matrix multiplication of input x and weight WX, parallel multipliers and a tree of adders are utilized. Note that these multipliers reduce to simple masks (AND gates) when using binary inputs x. A similar structure, not shown in Figure 4, is used to compute $YFAC[p][f]$. The scalar multiplication $XYFAC[p][f] = XFAC[p][f] \times YFAC[p][f]$ is then performed. Next, the product term $XYFAC[p][f] \times WH[k][f]$ is computed for all k in parallel. An accumulator associated with each k sums these terms (over f) as the loop over f proceeds, to yield $tem[k] = \sum_f XYFAC[p][f] \times WH[k][f]$. This corresponds to a matrix multiplication of $XYFAC$ by the *transpose* of the WH matrix. Note that in order to multiply by a weight matrix (WX in phase 1, WH in phase 3, etc.) we use parallel multipliers and an adder tree; but in order to multiply by the *transpose* of a weight matrix (WH in phase 1, WY in phase 3, etc.) we accumulate the product terms one at a time for N_h (number of hidden layer nodes) clock cycles. After the N_h clock cycles, each accumulator outputs an element value of the *tem* matrix.

Thus the two types of matrix multiplication operations differ in their latencies for producing the first result. The matrix multiplication ($\sum_{i=1}^{N} x[p][i] \times WX[i][f]$), using the adder tree, produces its first result in one time step, whereas the multiply-accumulate matrix multiplication takes N time steps (one for each f) to produce all of the $tem[k]$ results in parallel.

The multiply-accumulate implementation enables us to avoid performing explicit transpose operations on the weight matrices. The transpose operations that we thereby avoid would have been high-cost because of the read/write operations required for memory element relocation and the additional storage required to store a transposed version of the weight matrix.

Phase 2 completes the calculation of the hidden node values[4] $h[p][k]$ by applying a sigmoid function to each $tem[k]$, followed by comparison with a generated random value, to decide whether $h[p][k]$ should be set to one or zero. The sigmoid function is implemented using a piecewise linear approximation to reduce hardware cost. The random number generator is similar to the implementation reported in Tausworthe [1965]. To reduce the hardware footprint, one $h[k]$ is calculated at a time. Alternatively, one could have implemented k sigmoid functions and random number generators in parallel to reduce the latency of this phase.

Phases 3,4 (Reconstruct) and 5,6 (Regenerate) have very similar hardware structures to Phases 1,2 (Generate) described here, and hence will not be described further.

3.3. Hardware Design of Weight Update Operation

Phases 7 through 12 are used for updating weight values WX, WY, WH, WYB, and WHB. These phases are divided into three categories in terms of design architecture. First, phase 7 is used to compute the matrix $H'FAC$ needed for updating WX and WY. Second, phases 8, 9 and 10 are used to calculate updated values for WX, WY, and WH. The mathematical equations for these in the algorithm are similar, hence each of these phases is implemented using same hardware structure. Finally, phases 11 and 12 are used to calculate updated values for WYB and WHB over all inputs in a given batch.

Phase 7 is easily implemented with a vector multiplier and a tree of adders to produce an element result of the $H'FAC$ matrix per timestep.

The matrix multiplications of phases 8–10 are handled in a special way, in order to make efficient pipelining possible. Consider, for example, phase 8 (see pseudocode). The overall purpose of this phase is to compute $\Delta(WY)_{jf} = \sum_p Y_{pj}(XHFAC)_{pf} -$

[4]The addition of the bias weight w_k^{hb} to $tem[k]$, before performing the sigmoid function, is omitted from this figure.

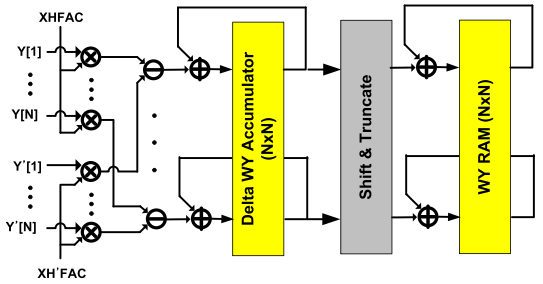

Fig. 5. Hardware diagram of phase 8 for updating weights WY.

$\sum_p (Y')_{pj}(XH'FAC)_{pf}$ for all (j,f). If we were to compute each sum on the right-hand side (RHS) using matrix multiplication in the ordinary way, each element (j,f) of the first RHS sum would depend on all of column j of the matrix Y and all of column f of the matrix $XHFAC$; that is, on the matrix elements of Y and $XHFAC$ for all cases p in the batch. Phase 8 would then need to wait for $N(N_c - 1) + 1$ time steps before it could start to compute the RHS. This would require a long pipeline stall in the architecture following phase 7, and storing all of the $XHFAC$ matrix in memory until all cases p have been processed.

Instead, as shown in the pseudocode and in Figure 7, we pass results for each case p from each phase to the next as soon as these results are available. Thus, as soon as phase 6 has completed processing case p, phases 7 and 8 can start to process this case, using only the elements (p,f) of $XHFAC$ and (p,j) of Y for that p to compute that case's contribution to the update term $\Delta(WY)$. Intermediate computations for each case p are not saved after they have been used by the downstream phases. This avoids the need to store the entire $XHFAC$ matrix (and others), although a pipeline stall of $6N$ time steps (following phase 10) is unavoidable.

Figure 5 shows the hardware implementation of phase 8. Two sets of vector multipliers on the left compute the terms $Y \times XHFAC$ and $Y' \times XHFAC'$, respectively. The difference is accumulated over the number of cases in the batch to form $\Delta(WY)$, which is then shifted and truncated before combining with the current WY values to produce the new WY results. The shift and truncation implements the pseudocode's multiplication by (ϵ/N_c), where this factor is preferably chosen to equal 2^{-s} for integer $s > 0$. The order of processing (compute difference for each case, then accumulate differences, then shift and add to the previous weights) is important for preserving numerical precision, since in general the difference for each case is much smaller than either term for that case, and the total difference ΔWYB is much smaller than WYB.

Phase 11 is implemented very similarly to phase 8, except that there are only N bias weights $WYB[j]$ rather than N^2 connection weights $WY[j]\,[f]$ to be updated. The difference vector $Y - Y'$ is computed and accumulated over all cases in the batch to form ΔWYB, which is shifted and truncated, then combined with the current WYB to produce the new WYB vector. The phase requires only N, rather than N^2, timesteps, and is executed only as the last case of a batch is being processed by phases 7–10 (see Figure 7). Phase 12 updates WHB in a fashion similar to phase 11, and therefore will not be described further.

3.4. Internal Memory Architecture

The current implementation is dominated by combinational logic and RAM blocks, due to the parallel realization of multiply-accumulate structures described earlier, as well as the need for storing and forwarding intermediate data during processing in

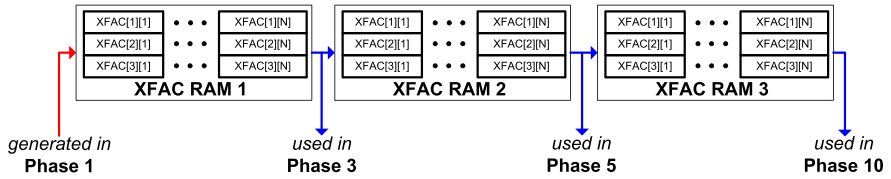

Fig. 6. A block diagram of memory use of the X factor.

the pipeline. Efficient data handling is therefore critical to address the RAM block usage. The remainder of this section describes our memory architecture with the goal of minimizing FPGA block RAM resource utilization.

The RAM blocks are divided into two categories, one to store weight values and the other for temporary data, which is used only during the current weight update operation. The RAM blocks for storing weight values are in turn divided into two categories, for connection weight values such as WX, WY, and WH and for bias weight values such as WYB and WHB. Each of the two categories has different storage requirements.

The required memory size for holding weight values WX, WY, and WH is on the order of $N^2 \times BW_{weights}$ bits where $BW_{weights}$ is the bit width for each weight value, and we assume for ease of description that the numbers of x, y, h, and f nodes are all of order N. Each of these memory blocks is arranged as N (actually N_f) rows each having $N \times BW_{weights}$ bits. These relatively wide input and output data ports facilitate the processing of a full row of matrix computations in parallel every time step.

To maximize concurrency of operations, two memory blocks are used for each set of weight values. One block acts as a scratch-pad accumulator for the ΔW weight increments, while the other stores the current weight values used for computations. After each batch, the cumulative delta weights are right-shifted and added to the current weight values to produce the updated weights for the next iteration. This architecture guarantees no contention for memory accesses to any of the weight values. Furthermore, each memory block is dual-ported, enabling a new value to be written at the same time that the next value to be updated is being read. Note that the proposed architecture doesn't require additional memory to store the transposed matrix of these weight values, as explained in Section 3.2.

For each of the bias weight vectors WYB and WHB, the required storage array is of size $N \times BW_{\text{weights}}$ bits. Each array has N entries and BW_{weights} bits per entry. These arrays are also dual-ported.

Our architecture takes advantage of the streaming nature of processing to substantially reduce the amount of memory needed for temporary data (e.g., intermediate results). Consider, for example, the $XFAC$ matrix which is the product of the input vector X and the weight matrix WX. In general, a memory area of size $N^2 \times BW_{\text{data-path}}$ would be needed to store this matrix. However, because the processing pipeline is purely feed-forward with no stalls, we only buffer the minimum number of matrix rows needed to hold the data from generation to consumption. The number of rows is a function of the pipeline structure. For $XFAC$, the matrix is computed in phase 1 and used in phases 3, 5, and 7 (see pseudocode). A buffer capable of storing 3 rows of partial results of the $XFAC$ matrix is enough to hold the data between phases 1 and 3. Similarly, three more rows are needed to reach phase 5, and another three rows of buffering are required to hold the data for processing at phase 7, bringing the total amount of buffer storage required to nine rows of data. Hence, the total amount of storage needed to handle the $XFAC$ matrix throughout the pipeline is only $9N \times BW_{\text{data-path}}$ bits. Note that this scales linearly with the number of nodes N since the pipeline depth is constant, whereas storing the full $XFAC$ matrix would scale quadratically. Figure 6

shows the three-stage buffering of *XFAC* matrix rows. Similar savings are realized for other intermediate data, including $YFAC, XYFAC, H, XHFAC, YHFAC, Y', H'$, and $Y'FAC$.

3.5. The Processing Components

A number of processing primitives are used to implement the fRBM ANN algorithm, including adders and multipliers for matrix multiplication, sigmoid function, random number generators, and shifters. This section describes some of the implementation choices for these processing elements and the tradeoffs involved therein.

Since the computations are dominated by matrix multiplications, we chose to implement a set of parallel multipliers followed by an adder tree for reduction to handle a full row of processing in parallel. For problem sizes where N is up to a few hundred nodes, this seems to be a reasonable tradeoff between performance and implementation complexity. For larger problem spaces, one may have to limit the parallel computations to a fraction of a row to manage the circuit size.

The sigmoid function is implemented using a piecewise linear approximation of the non-linear function (PLAN), similar to that described in Amin et al. [1997] and Larkin et al. [2006]. This technique is very popular in hardware implementations of ANN algorithms to eliminate the need for exponential and division operators required for an accurate implementation of the sigmoid function. These operators are typically complex and would require significant hardware area.

For random number generation, the "Tausworthe 88" random number generator was adopted because it provides a very high degree of randomness yet can be efficiently implemented in hardware [Tausworthe 1965]. In addition, for a given initial seed, it provides a repeatable sequence of output values. This is highly desirable since it enables cycle-by-cycle comparison of the hardware implementation against a software reference code, thereby aiding the logic verification process.

3.6. Operation Timing of the Fine-Grain Parallel Processing and the Coarse-Grain Pipeline Architecture

As discussed in previous sections, the proposed architecture is comprised of a coarse-grain pipeline consisting of 12 stages. Since matrix multiplication is the dominant operation in the fRBM ANN algorithm, we implement parallel multiplier structures capable of operating on all elements in a matrix row in parallel, thus generating one element of matrix result every time step.[5] Thus, one iteration to process a whole batch of inputs requires NN_c time steps for each of phases 1 to 10, as shown in Figure 7, where N_c is the number of cases in the batch and N is the number of inputs per case. Without pipelining, these ten phases of computation would require $10NN_c$ time steps to complete. Moreover, it would also require significantly larger memory structures to store intermediate results produced from each phase. The coarse-grain pipeline architecture maximizes performance and minimizes storage requirements by overlapping these phases of computation to the maximum extent possible. Since the computations are row-oriented, for each of phases 2 to 7, the earliest point in time a given phase can start is as soon as the first row of inputs is available. This is N time steps later than the start of the previous phase. Phases 7 to 10 can all start in parallel, since there is no data dependencies among their computations. Phases 11 and 12 each require only N_c time steps to complete and are pipelined to start as soon as enough data is available from the preceding stages, as illustrated in Figure 7.

[5]We use "time step" to denote the unit of time in the coarse-grain pipeline and "clock" to denote the unit of time in the fine-grain pipeline. One time step can consume one or more clocks.

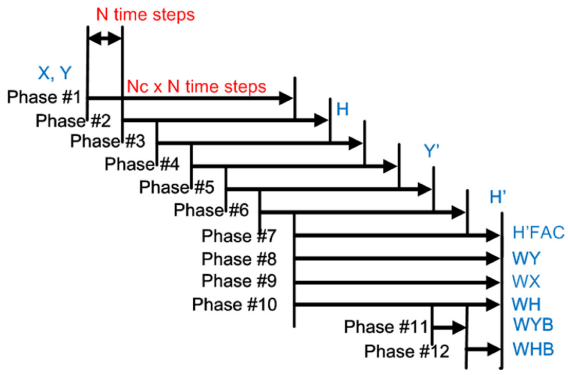

Fig. 7. Timing of the coarse-grain pipeline architecture. Each of phases 1–10 takes NN_c timesteps to execute; phases 11 and 12 take N_c timesteps each. Note that phases 11 and 12 are executed only at the end of each batch, and not for each case within the batch.

From this timing diagram, the number of time steps per batch is[6]

$$N_{\text{time-steps}} = (N_c + 6)N. \tag{3}$$

In addition to the coarse-grain pipeline, fine-grain pipelining is introduced to achieve higher clock rate implementations. Fine-grain pipelining in this context refers to executing each of the time steps in multiple clock cycles. For example, the hardware multiplier can be pipelined to use two clock cycles instead of one. Since the computations are mostly feed-forward and have no feedback paths, this well-known hardware technique can be readily applied throughout this design. Fine-grain pipelining increases the number of clock cycles required to process each batch (and adds a small delay due to additional register stages), but allows one to increase the clock frequency, enabling an overall decrease in processing time per batch.[7]

3.7. Performance Metric

A convenient performance metric of ANN systems is the number of connection update terms computed per second ('CUPS'). We define a "connection update term" for an RBM as the contribution of a single training case (input presentation) to the change in a single weight. Then, for a software implementation of an fRBM, CUPS = {[number of input presentations per batch]×[number of weights being updated]}/T_{SW}(sec), or

$$\text{CUPS}_{\text{SW}} = \{N_c \times N_f(N_x + N_y + N_h)\}/T_{\text{SW}}, \tag{4}$$

where T_{SW} is the training time per batch. (This ignores the small contribution from the update of the bias weights.)

To avoid confusion, we emphasize that we count each case's contribution to a weight update as one "connection update term" when computing CUPS, even though the weight is actually changed only once per batch. This is appropriate because (a) the time required to increment each weight w by the cumulated Δw at the end of a batch in a software implementation is negligible compared with the time required to compute

[6]This equals $N + 6N/N_c$ time steps per case, so the processing time per case (or per epoch) decreases as N_c is increased, by up to sevenfold when compared with the case $N_c = 1$. However, when N_c is made too large, the weights are updated too rarely, so that the number of epochs required for convergence is increased. The optimal N_c value is task-dependent.

[7]For brevity, the discussions of timing and performance in this article do not take fine-grain pipelining into account (i.e., they assume that one time step equals one clock cycle), with the exception of Section 5.2 regarding the choice of clock frequency.

the Δw terms for each case, and (b) the total training time is insensitive to N_c over a reasonable range of N_c values.[8] Care should be taken when comparing CUPS values in the literature, regarding the exact definition used and its dependence on parameters such as N_c.

For the hardware implementation based on the proposed pipeline architecture, T_{HW} (the training time per batch of the proposed hardware architecture) = [the number of clock cycles required to process one batch] / [the machine clock frequency F in cycles/sec], or $T_{\mathrm{HW}} = N_{\max}(N_c + 6)/F$ where $N_{\max} \equiv \max(N_x, N_y, N_h, N_f)$. Thus, by replacing T_{SW} in the Eq. (4) with T_{HW} term

$$\mathrm{CUPS}_{\mathrm{HW}} = \frac{N_f(N_x + N_y + N_h)F}{N_{\max}} \frac{N_c}{N_c + 6} \approx 3NF\,, \tag{5}$$

the approximation holding in the case that $N_{x,y,h,f}$ are approximately equal (denoted by N) and $N_c \gg 6$.

The HW-to-SW performance ratio thus scales as N, because the HW is sized to perform N multiplications in parallel, and N is specified during HW design compilation, up to the limits of HW capacity. Furthermore, the HW updates the three weight matrices WX, WY, and WH in parallel, in contrast to a serial SW computation.

3.8. A Virtualized fRBM ANN Architecture

The proposed hardware architecture is highly scalable due to its fine-grain parallelism and coarse-grain pipeline architecture. Throughout the processing pipeline, operations are performed on a whole row and/or column (size N) of matrix elements in parallel, thereby achieving accelerated throughput and reduced latency compared to a serial implementation. In case of a single FPGA implementation, our experimental result in Section 5 shows the maximum number of neuron nodes to be 256, which is small.

To address these physical limitations, we propose to extend the current architecture so that it can model a potentially large virtual fRBM ANN using a smaller number of physical nodes. To achieve this, we exploit a time-division multiplexing technique, whereby the physical nodes are re-used to perform different portions of the fRBM computations at different times. A key challenge in pursuing this approach is to minimize the stalling of the pipeline due to switching the computational elements among the different portions of the virtual network, thereby maximizing utilization of the most expensive computational resources, namely the multiply-accumulate engines.

For definiteness, we will consider the case where the physical number of nodes implemented on the platform is N and the virtual number of nodes[9] is twice that, or $2N$.

A naive implementation of phase 1 – that is, without the modifications to the original architecture that we will introduce – would require $8N$ timesteps to finish phase 1 for a single case. Therefore, only one case could enter the pipeline every $8N$ [rather than $(2N)^2/N = 4N$] timesteps, thereby approximately doubling the processing time per batch compared with the method we will now describe.

Figure 8 shows a block diagram of our modified architecture for the "Generate" step to support the time-division multiplexed operation, virtually increasing neurons in a platform with limited hardware resources. To increase concurrent operation and re-

[8]Therefore, if we were to define CUPS as counting just one update per connection per batch instead, the CUPS value would be approximately inversely proportional to N_c, the number of cases per batch (since execution time per batch goes roughly as N_c in our implementation). But N_c can be chosen arbitrarily over a wide range, rendering that alternative definition useless for comparisons.

[9]The virtual ANN network is assumed to have $2N$ input X-nodes, $2N$ input Y-nodes, $2N$ hidden H-nodes, and $2N$ factor nodes.

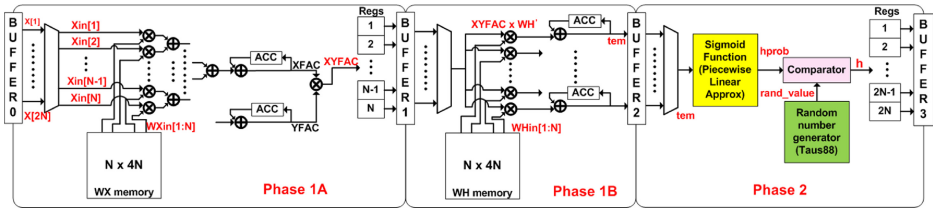

Fig. 8. A hardware block diagram for the Generate step of the virtualized architecture.

duce pipeline stalls, we split Phase 1 into two phases, 1A and 1B, by inserting a buffer stage. During phase 1A, *XFAC*, *YFAC*, and *XYFAC* are computed for each factor f (see pseudocode). To compute *XFAC* for a single factor f, 2 timesteps are needed. In the first timestep $x \times WX$ is accumulated over half of the input nodes $x[1 \ldots N]$ and the partial result is saved in a temporary register. In the second timestep, the other half of the input nodes, $x[N+1 \cdots 2N]$, are computed and added to the first half. Similar concurrent operations are done for *YFAC*. In this way the elements $XYFAC[f]$ are computed serially, one element every two timesteps. Therefore a full row (including $2N$ elements) of *XYFAC* is ready in $4N$ timesteps.

Phase 1B performs the remaining computation of Phase 1; namely, it incrementally computes $tem[k] \; += \; XYFAC[f] \times WH[k]\,[f]$, serially for all f, and in parallel for N k-values at a time. Since k and f each goes from 1 to $2N$, and we only have N multiply-accumulate engines, this computation also takes a total of $4N$ timesteps to complete for each presentation in an input batch.

During the first $2N$ timesteps of phase 1B we compute $tem[1 \cdots N]$ using N parallel engines, serially adding the contributions from factors $f = 1 \cdots 2N$ one term at a time; during the last $2N$ timesteps we compute $tem[N+1 \cdots 2N]$. The first half of phase 1B cannot finish until phase 1A is completed; and 1B "consumes" one f per timestep whereas 1A produces one f every two timesteps; so 1B can start $2N$ timesteps (and finish $6N$ timesteps) after 1A starts.

Phase 2 computes the final binary hidden node values by applying the sigmoid and probability functions to the output of phase 1B. Note that Phase 2's complexity is linear in the number of nodes, and therefore needs only $2N$ timesteps to compute its portion of a $2N$-node virtual network. (This contrasts with Phases 1A and 1B, whose complexity is quadratic in the number of nodes, hence require $4N$ timesteps each.) Note that the hardware of phase 2 will therefore be unused half the time.[10]

Since $tem[1 \cdots N]$ are all computed by timestep $4N$ (counting from the start of phase 1A), phase 2 can start then, and will complete computing $h[1 \cdots N]$ by timestep $5N$. However, phase 2 will then have to wait until phase 1B finishes computing $tem[N+1 \cdots 2N]$ at timestep $6N$, before starting to compute $h[N+1 \cdots 2N]$. Therefore, phase 2 will finish at timestep $7N$.[11]

Phases (3,4) and (5,6) are modified in the same fashion as phases (1,2). Thus phase 3A starts $7N$ timesteps after phase 1A starts, phase 5A starts $7N$ timesteps after phase 3A, and phase 7 starts $7N$ timesteps after phase 5A. Phases 7–10 thus start

[10]It is possible to share phase 2 hardware with either phase 4 or 6, eliminating some of this stall time.

[11]A technical detail: $XYFAC[2N]$ is computed during timestep $4N$ of phase 1A, and is required before phase 1B can complete computing $tem[1 \cdots N]$. This causes a lag of one timestep, which is compensated by an equal lead when phase 3A begins. Omitting derivations: Phase 1A starts at timestep 1 and ends after timestep $4N$. Phase 1B starts at $2N+2$ and ends after $6N+1$. Phase 2 runs from $4N+1$ to $5N$, and then from $6N+2$ to the end of $7N+1$. However, phase 3A can start at the beginning of $7N+1$ since its first timestep requires values $h[1 \cdots N]$ that were computed earlier. Thus, phase 3 can start $7N$ timesteps after phase 1 starts, as stated in the text.

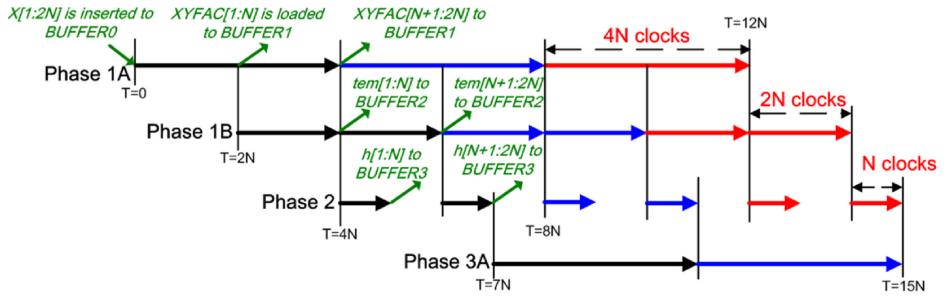

Fig. 9. A timing diagram for execution of the Generate step in the virtualized architecture.

$21N$ timesteps after phase 1A started, and require $4N$ timesteps to be executed (in parallel with one another). Therefore, a single case p flows through the pipeline in $25N$ timesteps. Each case enters the pipeline $4N$ timesteps after the preceding one, and the time it takes to flush the pipeline at the end of a batch is $21N$. Figure 9 shows a timing diagram of execution of Phase 1A, 1B, Phase 2, and Phase 3 in the virtual architecture.

Thus, the required number of clocks to process a single case using our modified architecture is

$$N_{\text{time-steps-virtual}} = 25\,N. \tag{6}$$

The performance in CUPS (connection updates per second), for the virtualized architecture having $2N$ nodes of each type, is CUPS = $N_c \times$ (# connections) $\times$ (clock frequency F) / (# timesteps for an entire batch of N_c cases), or

$$\text{CUPS} = \frac{3(2N)^2\,N_c\,F}{(4NN_c + 21N)} = \frac{3NN_cF}{N_c + 5.25} \approx 3NF \tag{7}$$

(compare Eq. (5)), the approximation holding when $N_c >> 5.25$. Thus, the performance in CUPS for $2N$ nodes of each type and N-way hardware parallelism is approximately the same as that for the nonvirtualized case with N nodes of each type and N-way hardware parallelism. Similar results apply when the virtualization is more than two-fold. This implies that the performance gain of the virtualized architecture is limited by the number of the parallel hardware operators.

Although the proposed virtualized architecture enables an increase in the maximum number of nodes that can be achieved in a physical device, the maximum number of virtual nodes is limited. As the virtual number of nodes increases, the storage requirement for the pipeline stage buffer (e.g., Buffer 3 shown in Figure 8) increases proportionally while the required number of operators is still constant. When the utilization of internal registers for a target FPGA device approaches 100 percent due to the growing storage requirement, the virtualized architecture cannot enable a further increase in the number of virtual nodes.

4. SOFTWARE MODELING AND BENCHMARK PLATFORM METHODOLOGY

The design starting point was our Matlab code implementing the software algorithm described in Section 2.2. This seed code uses double precision data type and built-in MATLAB math functions such as rand(), exp(), matrix multiplication, etc., and was re-implemented in C.

We used the following "toy"-sized problem for simulation and testing of the fRBM algorithm on the FPGA: As in Memisevic and Hinton [2010], the input vector X for each case is a random set of binary pixels laid out in 2-d. The input Y for the same

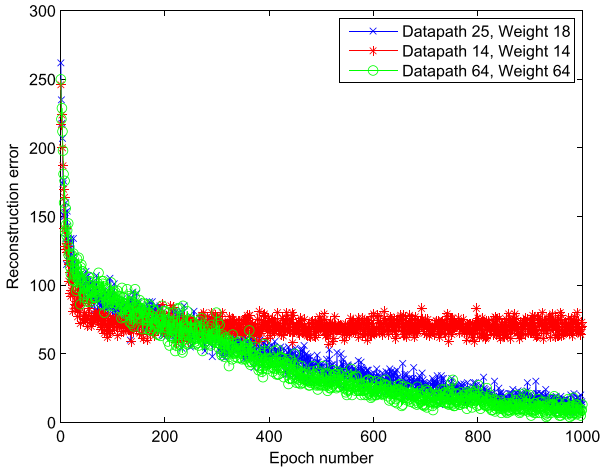

Fig. 10. Comparison of error convergence among fRBM runs (of the test problem described in Section 4) having different bit-width configurations for the data-path and weights. Reconstruction error is $\sum ||Y' - Y||^2$ after each training epoch, summed over 8 batches/epoch and 8 cases/batch.

case is the same as X, but shifted vertically and/or horizontally by a random number of pixels with respect to X, with wraparound in both dimensions. The network is to learn weights that will activate the layer-H nodes in a way that represents the horizontal and vertical shifts for each same case. (As a consequence, the trained network will also generate the Y pattern when X and H vector activations are inputted.) For the small FPGA used in our experiments, the number of nodes or factors in each layer is $N = 8$, and each image is 2×4 pixels.

Given the Matlab seed code, we constructed a bit-true and clock cycle accurate simulation model in C, in order to find optimal bit-widths for hardware designs and architectures before actual hardware implementation, and for bitwise verification of hardware computations.

In hardware implementation, reduced bit widths are commonly adopted for efficient implementation because double precision multipliers and adders are extremely costly. Bit width reduction comprises bit truncation, rounding and optionally saturation of output signals for each operator. Also, the hardware operators differ from the Matlab implementation in order to reduce hardware cost. Examples include the piecewise linear approximation of the sigmoid (logistic) function, and the truncation of every inner product during matrix multiplication.

The bit-true and clock cycle accurate simulation model generate output values slightly different from the seed model because of the approximations made. This model is used for verifying the customized hardware operators and the proposed architecture.

A principal issue when implementing a bit-true simulation model is finding optimal bit-widths for a specific target application. For ANNs, however, the required precision is algorithm- and application-dependent. Previous work suggests that 8 to 16 bits of weights and weight update precision are often satisfactory for ANN simulations [Holt and Baker 1991; Holt and Hwang 1993], and some recent FPGA implementations have used these results as a starting point.

We performed many experiments to find an optimal bit-width configuration for the fRBM ANN algorithm and our application. Figure 10 illustrates the results. The figure shows the reconstruction error $\sum_{pj} |y_j - y'_j|^2$ as training proceeds. Various configurations of bit widths for data-path and weights have been explored. (The bit width of

the data-path refers to the bit widths for each operator's output, intermediate data, sigmoid function output and random number, but not the weight values.) Three configurations of the bit-widths for data-path and weights are presented in the figure: (1) Data-path = 64 bits and weight = 64 bits; (2) Data-path = 25 bits and weight = 18 bits; and (3) Data-path = 14 bits and weight = 14 bits. Case (1) uses the seed MATLAB model and cases (2) and (3) use the bit-true simulation model. For various bit configuration sets of experiments, the simulation using 25 data-path bits and 18 weight bits closely approximate the double-precision results in convergence time and asymptotic error.

5. EXPERIMENTAL RESULTS

5.1. Experimental Setup

To verify the proposed hardware architecture, we have performed experiments on an FPGA-based board which includes a Xilinx Field-Programmable Gate Array (Virtex 5 LX330), 8 4MB SRAM memory devices, 1 GB DDR memory, and 1 GBit host Ethernet interface. The Xilinx Virtex5 LX330 device comprises 51,840 logic slices, each containing four 6-input look-up tables (LUT) and four registers, totalling over 200K LUTs and 200K registers. In addition, the device provides 192 25x18 embedded multipliers, and 10,368 Kbits of embedded Block RAMs.

The Factored RBM algorithm is first coded in MATLAB. A C model that is bit- and cycle-accurate is then coded. This model serves as a golden reference for validating the hardware implementation. The hardware description is generated automatically by a custom-developed MATLAB script. A user-specified text file includes parameter information such as the number of nodes (N_x, N_y, and N_h) and the bit widths of the real-valued operands. The script generates a synthesizable design in Verilog HDL description with the desired bit-widths of the datapaths and numbers of operators along the pipeline based on the proposed architecture. We use 25 bits of fixed precision to store the accumulated weights and 18-bit wide multipliers, which is the optimal configuration from our experiments. The generated HDL RTL is simulated using the Cadence NCSIM simulator and verified against the bit- and cycle- accurate C model for correctness. The design is then synthesized using the Synopsys Synplify Pro synthesis tool, and placed and routed using the Xilinx ISE 12.1 tool set. Finally the bit-streams are downloaded into the FPGA and the run-time results, such as converged error values, the generated Y' values which approximate the expected Y output values, and the trained weight values of the network are verified against the results of the bit- and cycle-accurate simulation model.

5.2. Hardware Cost and Performance Results

Figure 11 shows the resource utilization of the LX330 FPGA device to implement various node-size fRBM ANN systems using the proposed architecture. As the node size is increased from 8 to 16, 32, and 64, the resource utilization grows in linear proportion. The matrix multipliers that use parallel multipliers and accumulators are mapped to the embedded DSP modules of the FPGA device. The matrix multipliers that use parallel multipliers and an adder tree, the sigmoid function, the random number generator, and the adders/subtractors used in Phase 8-12 are all mapped to the LUTs. The pipeline stage buffers are mapped to Slice Registers. Storage elements for weight values and intermediate results are mapped to the Block RAM modules of the FPGA device. The bottleneck for the target Virtex5 LX330 device is the embedded DSP modules supporting up to 192 such engines. As such, the largest fRBM ANN achievable without virtualization on this device is on the order of 64 nodes, using 100% of the DSP resources, while other resources, such as LUTs, registers, and block RAM (BRAM) are

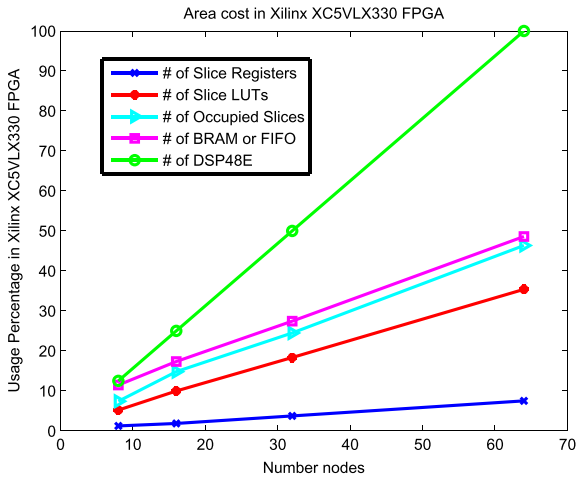

Fig. 11. Resource utilization of fRBM ANN systems using the proposed architecture on the XC5VLX330 FPGA.

Table I. Resource Utilization of other FPGA

FPGA	XC6VSX475T	XC6VLX760
node size (N)	128	256
Registers	29202 (4%)	52579 (5%)
LUTs	150839 (49%)	301272 (63%)
Occupied Slices	43944 (59%)	89792 (75%)
BRAM (32Kb, 16Kb)	261 (24%), 11 (1%)	483 (36%), 11 (1%)
DSP48E	384 (19%)	768 (88%)

at or below 50% utilization. To map larger networks on the LX330 device, we could implement additional multipliers using fabric logic to complement the DSP hardware multipliers. Alternatively, one could map larger networks by targeting other FPGA devices that are richer on DSP resources. Indeed, we experimented with two such targets, namely the Virtex6 SX475T and the Virtex-6 LX760. The Virtex-6 SX475T was chosen as a candidate target because it has the most abundant (2016) DSP resources and 2.5 times the number of logic slices found on the LX330. The Virtex6 LX760 was selected because it was the largest overall device available at the time of the experiment. Note that in these experiments we synthesized, placed and routed the designs, and created bit-streams for the three FPGAs. However, we executed physical operations only on the Virtex5 LX330 FPGA board. Thus, the performance results of Table II are based on simulations.

Table I provides hardware cost results for these two FPGAs. Using the Virtex6 SX475T FPGA as a target, the synthesis results show that N = 128 nodes, but not 256, can be implemented without virtualization. The N = 128 case utilizes 384 DSP modules (19% of the available DSP resources), 261 32Kb BRAMs and 11 16Kb BRAMs which are under 25% of the available BRAM resources, and 43944 slices or 59% of the available slices. Thus, the limiting factor to scaling to 256 nodes in this case is the number of logic slices.

On the other hand, the Virtex-6 LX760 possesses 2.3 times the number of logic slices and 4.5 times the embedded DSP blocks compared to the original LX330 target, respectively. Targeting the LX760 device, we can achieve an fRBM ANN implementation of

Table II. Performance Results using CUPS Metric

Platform	node size (N)	f (MHz)	T (μ s)	GCUPS	Gain
Software	128	N.A.	11700	0.538	1
XC5VLX330	64	80	57.8	13.6	25.3
XC6VSX475T	128	80	217.8	28.9	53.7
XC6VLX760	256	80	845.0	59.6	110.8
Virtualized FRBM	512	80	6662.6	60.4	112.4

Table III. The Maximum Nodes without/with the Virtualized Architecture

FPGA	Basic architecture	Virtualized architecture
XC5VLX330	64	512
XC6VSX475T	128	2048
XC6VLX760	256	4096

size 256 nodes. This configuration uses 768 embedded DSP modules (89%), 483 32Kb BRAMs and 11 16Kb BRAMs (less than 37%), and 301272 slices (63%) of the target FPGA.

From these results we conclude that the optimal FPGA for the proposed architecture is currently the Virtex6 LX760 and the biggest node size on that FPGA is 256. Of course, this can be increased further by exploiting the virtualized architecture presented in Section 3.8. The network size can also be greatly increased for configurations where the number of cases per batch, N_c, is small.

In order to calculate performance metrics, we need to consider the fine grain pipelining since it directly affects the achievable implementation frequency, and also the number of cases per batch, N_c, which together with the node size N determines the number of computations required for one set of connection updates. Without any fine grain pipelining to reduce combinatorial delay paths, the implemented hardware can run at 30Mhz. All reported implementations on XC5VLX330, XC6VSX475T, and XC6VLX760 have 2 level pipelining, thus achieving a frequency of 80Mhz.

Table II shows the performance measurement results for the reference software implementation and FPGA-based hardware accelerators for various node sizes. The software implementation uses the MATLAB code and is executed on a dual-core 64-bit Intel machine running at 2.4GHz and using 4GB of main memory. The CUPS on the software implementation is approximately constant over the range of N and N_c considered in Table II. At $N = N_c = 128$ the software CUPS is 538×10^6. The T column includes the measured time for training one batch of N_c cases on each platform, and the gain column provides the performance gain compared with the ($N = N_c = 128$) software solution. For FPGA implementations without virtualization, the gain (or speedup compared to software) increases approximately linearly with the node size N, showing the scalability of the architecture. In addition, the virtualized architecture is capable of modeling a node size of 512 on a Virtex6 LX760 FPGA as shown in the last row of Table II, albeit at a CUPS rate similar to the 256-node network due to the time multiplexing of the hardware resources as described in Section 3.8.

This work also proposed a virtualized architecture that can further increase the maximum number of nodes over the basic architecture. However, the virtualized architecture is limited in its growth: storage requirements for intermediate results are a function of the number of virtual nodes. Table III compares the maximum number of nodes between the basic and the virtualized architectures for different FPGA devices.

6. CONCLUSIONS

In this work, we present what is, to the best of our knowledge, the first FPGA implementation of a Factored Restricted Boltzmann Machine (fRBM) algorithm. More generally, we show how a pipelined architecture for "mini-batch" training of an ANN can greatly accelerate learning, compared with designs that handle one (or only a few) cases at a time. We find over 100 $\times$ performance gain of the proposed architecture (compared to a software mini-batch implementation) in simulation.

In addition: (1) Our implementation accelerates the full processing pipeline, which increases performance and eliminates the need for storing intermediate results in off-chip memory. (2) Taking advantage of the reconfigurability of FPGA devices, we created an implementation generator, capable of producing different design points, rather than a single fixed implementation. (3) An efficient virtualized architecture is described that can simulate a large fRBM beyond the physical capabilities of a given FPGA device.

ACKNOWLEDGMENTS

We thank Dr. Daniel Friedman for many useful discussions.

REFERENCES

Amin, H., Curtis, K., and Hayes-Gill, B. 1997. Piecewise linear approximation applied to nonlinear function of a neural network. *IEE Proc. Circ. Dev. Syst. 144*, 6, 313–317.

Boser, B. and Sackinger, E. 2002. An analog neural network processor with programmable topology. *IEEE J. Solid-State Circ. 26*, 2017–2025.

Dias, F., Antunes, A., and Mota, A. 2004. Artificial neural networks: A review of commercial hardware. *Eng. Appl. Artif. Intell. 17*, 945–952.

Hinton, G., Osindero, S., and Teh, Y. 2006. A fast learning algorithm for deep belief nets. *Neural Computat. 18*, 1527–1554.

Holt, J. and Baker, T. 1991. Back propagation simulations using limited precision calculations. In *Proceedings of the International Joint Conference on Neural Networks*. 121–126.

Holt, J. and Hwang, J. 1993. Finite precision error analysis of neural network hardware implementations. *IEEE Trans. Comput. 42*, 3, 281–290.

Jung, S. and Kim, S. 2007. Hardware implementation of a real-time neural network controller with a DSP and an FPGA for nonlinear systems. *IEEE Trans. Indust. Elect. 54*, 265–271.

Kim, S., McMahon, P., and Olukotun, K. 2010. A large-scale architecture for restricted Boltzmann machines. In *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 201–208.

Larkin, D., Kinane, A., Muresan, V., and O'Connor, N. 2006. An efficient hardware architecture for a neural network activation function generator. In *Proceedings of the ISNN International Symposium on Neural Networks*. Vol. 144, 1319–1327.

Lindsey, C. and Lindblad, T. 1994. Review of hardware neural networks: A user's perspective. In *Proceedings of the 3rd Workshop on Neural Networks*. 26–30.

Ly, D. and Chow, P. 2010. High-performance reconfigurable hardware architecture for restricted Boltzmann machines. *IEEE Trans. Neural Netw. 21*, 1780–1792.

Maeda, Y. and Tada, T. 2003. FPGA implementation of a pulse density neural network with learning ability using simultaneous perturbation. *IEEE Trans. Neural Netw. 14*, 688–695.

Memisevic, R. and Hinton, G. 2007. Unsupervised learning of image transformations. In *Proceedings of the Symposium on Computer Vision and Pattern Recognition (CVPR'07)*.

Memisevic, R. and Hinton, G. 2010. Learning to represent spatial transformations with factored higher-order Boltzmann machines. *Neural Computat. 22*, 1473–1492.

Memisevic, R., Zach, C., Hinton, G., and Pollefeys, M. 2010. Gated softmax classification. *Neural Inf. Proc. Syst. 23*, 1603–1611.

Oh, K. and Jung, K. 2004. GPU implementation of neural networks. *Patt. Recog. 37*, 1311–1314.

Raina, R., Madhavan, A., and Ng, A. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 873–880.

Ranzato, M. and Hinton, G. E. 2010. Modeling pixel means and covariances using factorized third-order Boltzmann machines. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2551–2558.

Tausworthe, R. 1965. Random numbers generated by linear recurrence modulo two. *Math. Computat. 19*, 201–219.

Taylor, G. and Hinton, G. 2009. Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*.

Zhu, J. and Sutton, P. 2003. FPGA implementations of neural networks - A survey of a decade of progress. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications*. 1062–1066.

Received February 2013; revised July 2013; accepted September 2013