2014 International Joint Conference on Neural Networks (IJCNN)
July 6-11, 2014, Beijing, China

Computation of Deep Belief Networks Using Special-Purpose Hardware Architecture

Byungik Ahn

***Abstract*—The computation of deep belief networks (DBNs) requires a large number of arithmetic operations, which can be handled only by arithmetic operators. However, the operators are utilized only a small fraction of the time (1 – 5%) when they are computed by general-purpose computers. In this paper, a special-purpose hardware architecture that computes DBNs using a large number of arithmetic operators with a utilization rate greater than 60% is proposed. On the basis of neuron machine architecture, the computation units in the system are controlled according to stage operation table, which specify the sequence of the computation stages; thus, the complicated procedure of the DBN can be carried out in hardware. Moreover, the usage of the memory space is considerably improved by using offset addressing. The proposed schemes are implemented on a hardware simulator coded in MATLAB and on an FPGA chip. The full source code of the hardware simulator is available at a website. The readers can execute the code on the fly and reproduce the proposed schemes. The FPGA implementation achieves a lower computational time by a factor greater than 100 compared to a PC.**

I. Introduction

RECENTLY, deep belief network (DBN) models have captured significant interest from the machine-learning community and brought the resurgence of neural networks. The superior ability of DBNs in learning features that capture higher-order correlations in unsorted data has allowed DBNs to be applied successfully in many application domains such as classification, data mining, recognition, and robotics [1].

Training a DBN involves training several restricted Boltzmann machines (RBMs) and requires a considerable amount of time even for modern CPUs. General-purpose graphics processing units (GPUs) are gaining popularity because they provide a lowering of the computational time by a factor of 5 – 50 compared to x86 CPUs. However, most general-purpose computers, including systems with the CPUs and GPUs, are highly inefficient in terms of utilization of arithmetic operators.

For example, the experiment in [2] shows that it took 40 min to train 60000 samples with a 784×800 RBM on an Intel dual-core i5-2410M 2.3-GHz CPU. According to the RBM algorithms, the computation requires approximately 4×10^{11} arithmetic operations, as will be discussed later in this paper. In this CPU, computations are handled by one Advanced Vector Extensions (AVX) block contained in each core (two cores in the CPU), and each AVX can sustain four double-precision (DP) FP operations per clock cycle. Therefore, the full throughput of the CPU that can compute in 40 min is 4.4×10^{13} arithmetic operations. This estimation shows that the arithmetic operators effectively work for only approximately 1% of the full CPU time. Moreover, the proportion of arithmetic operators in the CPU hardware resources is extremely low. The number of transistors used for the CPU is 6.24×10^8 [3], whereas a DP FP multiplier can be implemented using only 2×10^5 transistors [4].

In the case of GPU, a 720-MHz NVIDIA GeForce 460, for example, has 336 cores each computing one DP FP fused multiply-add operation per clock cycle. A state-of-art RBM implementation on this GPU computed the same RBM network 46 times faster than the aforementioned CPU [2], which resulted in an operator utilization rate of 3.2%. Similar results on other CPU and GPU implementations have been reported in [5]. This implies that conventional computers use only a small fraction of the time and hardware resources for the computation.

In fact, this inefficiency originates partly from the functionality required to maintain the systems to be general-purpose: all-purpose stored-program architecture, centralized memory, compiled code from high-level programming languages, and so forth. However these features may not be necessary when a computation device is dedicated for computing neural networks or DBNs. If we increase the operator utilization rate, the chip size and power consumption can be considerably reduced.

In this paper, a special-purpose hardware architecture that computes a DBN with a high arithmetic-operator utilization rate is described. The average utilization rate of hundreds of floating-point operators is maintained at a value greater than 60% throughout the computation, and the proportion of hardware resources that are used for the operators is greater than 80%.

The proposed architecture is based on neuron machine (NM) hardware architecture. Regarding the NM, we proposed in [7] a preliminary hardware structure for neural networks in which communication between neurons is performed only using a set of memory combinations and a number of synaptic computations can be carried out in parallel. However, this hardware structure can only compute simple perceptron models. In [8], the architecture was modified so that neural models with complex synaptic and neuronal functions can be accommodated, and a spiking neural network with complex functions was implemented using the extended architecture, even though it supports only feed-forward networks and lacks schemes to run back-propagation models. A memory called reverse-mapping memory (R memory) was proposed in [9]

Jerry Byungik Ahn was with KT (Korea Telecom), Seoul, Korea. He is now with Neurocomputings.Com, Seoul, Korea (phone: +82-10-3010-1540; e-mail: jerry.ahn@neurocomputings.com).

978-1-4799-1484-5/14/$31.00 ©2014 IEEE

and can be used for sharing synaptic weights between feed-forward and backward networks without moving or copying weight data.

However, previous NM systems have two major problems. First, the memory space required to store the neuron outputs can be excessive when the synaptic parallelism is tuned to be large. Second, the parameters used to operate the system are scattered in the control circuits and even fixed in the hardware. This may be acceptable for computing simple networks, but it makes the system intractable as the neural network grows large and complex, as in DBN.

A new control scheme is proposed in this paper, in which the procedure of the computation stages is described in a table and a limited form of the stage-level programming can be carried out. This scheme simplifies the system control and enables optimization of the system. Further, an efficient sequence of computational stages for the DBN is also proposed, by which the memory space required can be considerably reduced.

The proposed architecture is implemented on a hardware simulator and on an FPGA. One of difficulties in presenting new ideas for a hardware architecture is that it is very difficult for the readers to reproduce the proposed schemes. In this paper, a self-contained hardware simulator simulating the proposed system is developed in MATLAB, and the full source code is available at a website. Even though the source code is coded in a sequential language, it is cycle-accurate and simulates various hardware components such as shift registers and pipelined FP operators. The reader can execute the simulator on the fly and verify the operation of the proposed system. The simulator code can also be translated into hardware description languages such as VHDL without difficulty and implemented in hardware.

This paper does not state improvements in the DBN algorithm or better arithmetic operations. The proposed schemes focus on how to maximize the utilization of existing arithmetic operators to compute DBN algorithms.

Section II of this paper reviews the DBN and the NM hardware architecture, and Section III describes the proposed schemes. In Section IV, the implementation details and hardware simulator code are described. The experiment results and conclusions are presented in Sections V and VI, respectively.

II. Background

A. DBN

An RBM is composed of a visible layer and hidden layers with connections between the layers but not between units within the same layer. DBNs are a composition of RBM networks where the hidden layer of each RBM serves as the visible layer for the next RBM. This leads to a fast layer-by-layer unsupervised training procedure, where contrastive divergence is applied to each sub-network in turn, starting from the lowest pair of layers.

For a given training sample $\mathbf{v}$, the connection weights of an RBM network are updated as

$$\Delta w_{ij} = \varepsilon\left(vpos_i \cdot hpos_j - vneg_i \cdot hneg_j\right) \quad (1)$$

$$p\left(hpos_j = 1 \mid \mathbf{v}\right) = ppos_j = \sigma\left(b_j + \sum_{i=1}^{I} vpos_i \cdot w_{ij}\right) \quad (2)$$

$$p\left(vneg_i = 1 \mid \mathbf{h}\right) = \sigma\left(a_i + \sum_{j=1}^{J} hpos_j \cdot w_{ij}\right) \quad (3)$$

$$p\left(hneg_j = 1 \mid \mathbf{v}\right) = pneg_j = \sigma\left(b_j + \sum_{i=1}^{I} vneg_i \cdot w_{ij}\right) \quad (4)$$

$$\begin{aligned} \Delta a_i &= \varepsilon\left(vpos_i - vneg_i\right) \\ \Delta b_j &= \varepsilon\left(hpos_j - hneg_j\right), \end{aligned}$$

where w_{ij} is the weight of the bidirectional connection between the visible neuron v_i and the hidden neuron h_j; ε is a learning rate; $vpos_i$ is the training sample for v_j; $hpos_j$ is the state of h_j generated form the training sample; $vneg_i$ and $hneg_j$ are the reconstructed states of v_i and h_j, respectively; a_i and b_j are biases for v_i and h_j, respectively, and $\sigma(x)$ is the sigmoid function $1 / (1 + e^{-x})$ [10]. Algorithm 1 shows a typical computation of DBN with multiple hidden layers.

Algorithm 1: DBN procedure

```
1: T ← training sample
2: for each RBM_l from bottom to top
3:   for each sample s in T
4:     train RBM_l with s
5:     add h∋hpos_j in T_new
6:   end for
7:   T ← T_new; empty T_new
8: end for
```

As the computation is dominated by connection-specific arithmetic, training one connection requires approximately 3 multiplications and three additions for (2) – (4), and two multiplications and two additions (including subtraction) for (1). Therefore training a 784×800 network with 60000 samples, for example, requires 1.9×10^{11} multiplications and 1.9×10^{11} additions. Although there are some variations among DBN algorithms, the differences are not significant with regard to computational complexity [1].

The computation of the DBN presents a few problems when it is computed with special-purpose hardware: (1) the computation process is too complicated to implement in hardware, requiring several stages of different computations and (2) if all training samples are trained before training the next RBM, as is the case in most software implementations, a large memory space is required for $hpos_j$ in order to use $vpos_i$ in the next RBM.

B. NM Architecture

In the NM architecture, the computational model of the neural network is:

$$y_j = f_N\left(\sum_{i=0}^{p_j - 1} f_S\left(y_{m_{ij}}, w_{ij}, \ldots\right), \ldots\right) \quad (5)$$

where y_j is the state of the jth neuron, f_N is a neuronal function, p_j is the number of synapses (connections) on the jth neuron, f_S

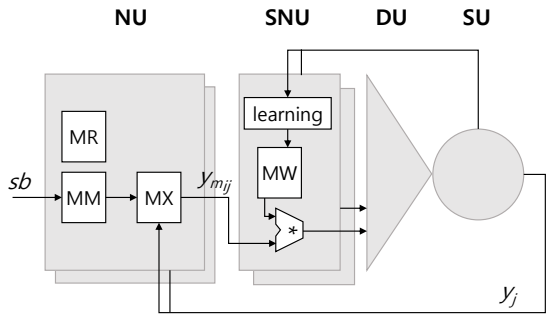

Fig. 1. Configuration of NM architecture. NM is characterized by a memory structure in NU that maps the outputs of the neurons to the inputs of other neurons, which enables a large number of connections can be computed simultaneously.

is a synaptic function, m_{ij} is the index number of the neuron connected on the ith synapse of the jth neuron, and w_{ij} is the weight of the ith synapse connected on the jth neuron.

An NM is a register-transfer level (RTL) special-purpose hardware architecture for simulating neural networks. The NM is characterized by a memory structure that maps the outputs of the neurons to the inputs of other neurons. A memory unit called network unit (NU) takes a newly computed neuron output y_j and produces a multiple $y_{m_{ij}}$ for a specific neuron simultaneously at every clock cycle.

The NU is composed of P modules, each with two individual memories: MM and MX. Figure 1 shows a typical configuration of an NM system including the internal structure of the NU. In each memory module, the output of MM is connected to the address input of MX, and the output of MX becomes one of P $y_{m_{ij}}$ outputs of the NU. MX is a dual-port memory in which read and write operations can be carried out simultaneously. The write ports of all MX memories in the NU are connected together which become the y_j input of the NU. The content of the kth MM and MX are:

$$MM_k(b) = m_{ij}$$
$$MX_k(j) = y_j,$$

where

$$i = \mathrm{mod}(b \times P + k,\ bpn \times P)$$
$$j = \lfloor b / bpn \rfloor$$
$$bpn = \lceil \max p_j / P \rceil.$$

If $i \geq p_j$, a special value is stored in MM, indicating that it is a null connection [7].

With these memories, addressing the sb input sequentially from 0 to $(N \times bpn) - 1$ will make the NU produce P $y_{m_{ij}}$ outputs at a time starting from the first P connections of first neuron to the last P connections of the last neuron, where N is the total number of neurons. The set of P connections that are produced at the same clock cycle is called a synaptic bunch (SB).

The NU enables synapse units (SNUs) to compute the synaptic functions of P connections simultaneously, a dendrite unit (DU) to sum the results of the SNUs, and a soma unit (SU) to compute the neuronal functions, one after another. The new neuron output, y_j, computed at the SU is then stored in the MX memories in the NU via the y_j input. There are MW memories at the SNU for w_{ij}; their contents are stored similarly to the MM case. The connection from the SU to the SNU in Figure 1 is for Hebbian learning [7]. For back-propagation models, a memory called MR is included in each memory module in the NU to store the reverse mapping information of the network topology [9]. In case of a multi-layered network, each computation stage is computed in sequence, as if they are separate networks, and neuron outputs computed in the previous stages are shared by means of MX memories.

By designing the computational units as fully pipelined circuits, the system can compute P connections per clock cycle and one neuron output per bpn clock cycles.

The NM architecture has a number of advantages: communication between the neurons is accomplished just by accessing memories requiring no communication overheads; network topology information is stored only in MM (forward network) and MR (backward) memories with no restriction on the network topology; weight memories are distributed and embedded in the computational circuits and the data paths of the memories are short; a large number of connections can be computed simultaneously by multiple SNUs; large-scale pipelining parallelism can be obtained from the computational units and the pipeline delay results in little impact on the overall performance [7]; and simple and uniform structure without necessarily requiring a main computer.

However, there are a few problems with this architecture; it requires a large memory space for MX to maintain the duplicated values of y_j, and controlling the system becomes difficult as the number of hidden layers increases.

III. Proposed Schemes for DBN

In previous works, a control unit (CU) was composed of combinational and sequential logics without structures. In our design, the CU is more organized to deal with the complex computation scenarios of the DBN.

A. Stage Operation Table

In our scheme, offset logics are placed in front of the address inputs of all memories and the memory offset is controlled by the CU. For instance, if the offset for the MM memories is set to 6000, and the sb input is sequentially scanned in the range from 0 to 6499, a range of 6000 – 12499 is addressed from all the MM memories. This scheme makes memory space re-locatable and reusable, and the number of bits needed to represent the memory address to be reduced.

A table called a stage operation table (SOT) is used to control the computation sequence. It consists of a series of records each containing the information required to control the respective stage. At the beginning of the stage the CU reads the record and sets the control registers. During the computation, circuits in the system reference the register to select multiplexers or to set the counters, for example. Figure 2 shows an SOT as an example. The first record denotes an input-only stage (type 1), in which a training sample with a size of 784 is loaded into the memories. The second record indicates that the stage is a positive hidden layer (type 2) with 500 neurons, with each neuron having 13 SBs and a total of

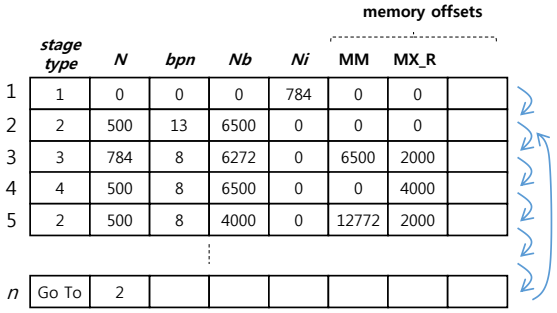

Fig. 2. Example of a stage operation table.

6500 SBs with zero offset for MM, and so forth. At the end of the table, a special record called a "go to" statement is specified, and the CU moves the current record pointer, as indicated by the second field, i.e., record 2.

B. Operation Sequence

In the proposed system, the memory space of each MX is divided into three blocks, denoted by MX{1}, MX{2}, and MX{3}. Each block is used to store the neuron values computed from one layer. Note that all MX memories have the same contents as mentioned earlier. The operation sequence for a DBN can be described by Algorithm 2, where *opr*(MX{*i2*}←MX{*i1*}) indicates that the neuron values in the MX{*i1*} block in this stage are read for computing the operation *opr*, and the results are written to the MX{*i2*} block.

Algorithm 2: NM DBN procedure

```
 1: MX{1} ← training sample 1;
 2: for each sample s in T
 3:   source = 1; target = 2;
 4:   for each RBMl from bottom to top
 5:     positive_hidden(MX{target} ← MX{source});
 6:     negative_visible(MX{3} ← MX{target});
 7:     {
 8:       negative_hidden(no_save ← MX{3});
 9:       update_weights;
10:       if RBM1 is top-most layer
11:         MX{1} ← next sample;
12:       end if;
13:     }
14:     switch source and target;
15:   end for
16: end for
```

There are four stage types in this procedure. The type 1 stage is carried out only at the beginning of the procedure, in which the first training sample is loaded in MX{1} without computation (line 1). Then, three types of stages are processed to compute each layer. In the type 2 (line 5) and type 3 (line 6) stages, the $hpos_j$ and the $vneg_i$ are computed, respectively. In the type 4 stage (lines 8 – 12), the $hneg_j$ is computed. In this stage, updating the connection weights and reading next training data (in the case of last layer) are also carried out simultaneously.

The use of the MX blocks is depicted in Figure 3, when there are three RBM stages. Nine stages are computed (1) – (9) for one training sample, and then the stages for the next sample follow.

Although a total of $L \times 3$ stages are computed for each

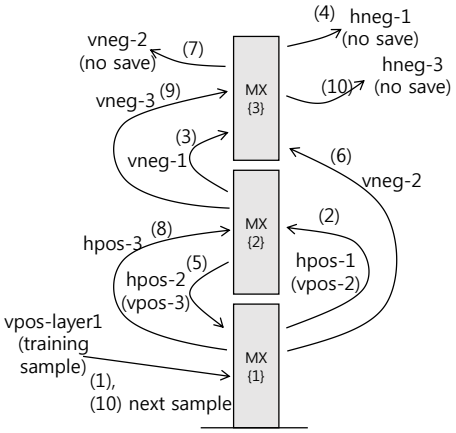

Fig. 3. Use of MX memory space for each training sample.

training sample, the MX space for only three stages are required by reusing spaces, where L is the number of layers in the DBN. In addition, the MM data used in type 2 stage can be reused in the type 3 stage by using the memory offset scheme and the SOT. In addition, the MW space for the type 2 stage is not stored; therefore approximately one-third of the MM and MW space can also be saved. Furthermore extra memory space for the values of $hpos_j$ is not required because all layers are computed consecutively before the next training sample is processed.

C. Use of MR memory

The use of MR memory requires a connection-placement algorithm. Although a method was briefly addressed in [9], its detail was omitted. The algorithm and the contents of MR memories can be described by Algorithm 3.

Algorithm 3: Connection placement

```
1: for j = 0 to J - 1
2:   for i = 0 to I - 1
3:     Store m_ij, w_ij at the mod(i + j, bpn_f * P)th
       place of neuron j,in MM and MW memories,
       respectively.
4:     Store the reference to w_ij in MW that saved
       in the previous step (line 3), at the
       mod(i + j, bpn_b * P)th place of neuron i in
       MR memory.
5:   end for
6: end for
```

Here bpn_f and bpn_b denote the values of *bpn* in the forward and backward networks, respectively. This algorithm ensures that connections in both forward and reverse networks are located at the same SNU, and MR reference the weight in the forward network. The use of MR enables the same weight address to be accessed in both networks.

IV. Implementations

A. System Design

Figure 4 shows the schematics of a design of the proposed system. In our simulator implementation, $P = 64$. Therefore there are 64 memory modules and SNUs although only one of

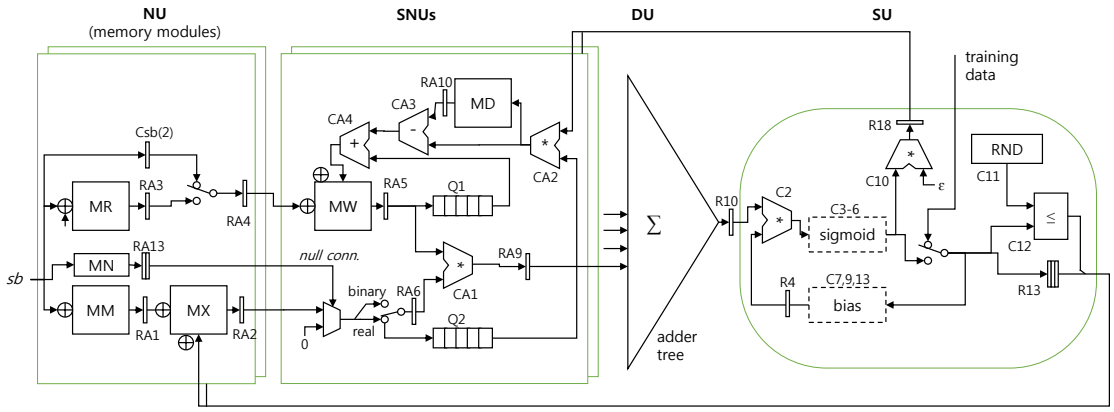

Fig. 4. Full schematics of implemented system.

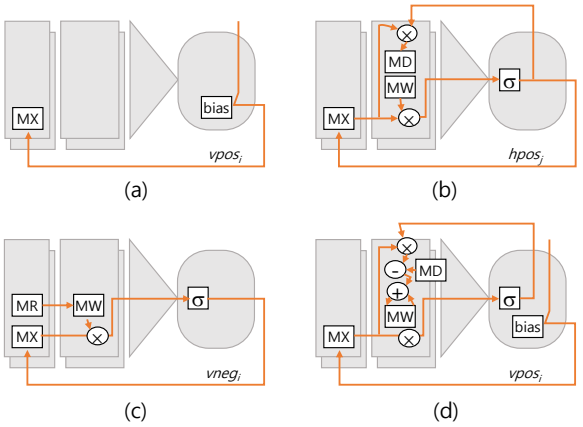

Fig. 5. Dataflow in each type of stages: (a) type 1, (b) type 2, (c) type 3, and (d) type 4 stages.

them is shown in the figure. The symbol names preceded by *CA* and *C* are for arithmetic operators, *RA* and *R* are for shift registers, and *Q* is for first-in first-out (FIFO) queues. The detailed circuits for the DU, and the sigmoid and bias functions in the SU are not shown for brevity. The number of arithmetic operators used for the SNU, DU, and SU are 256, 64, and 11, respectively. All arithmetic operators are assumed to have pipeline latencies of six clocks. A 1-bit-wide memory called MN is additionally included in each memory module in the NU in order to store the indication of null connections. When this bit is set, the input of the connection is set to zero and the value of the weight remains at zero in the SNU and therefore does not affect the computation result.

The flow of data in each type of stages is shown in Figure 5. In the type 1 stage, the values of a training sample are sequentially read in the SU and stored in MX{1} ($vpos_i$) as shown in Figure 5(a). In the type 2 stage, the values of $vpos_i$ are read from the NU, and both the $ppos_j$ and $hpos_j$ are computed by the SU and tied together and stored in the MX memories. At the same time, the values of $vpos_i$ from the NU are queued in Q2 until $ppos_j$ is computed in the SU, and $\varepsilon \times vpos_i \times ppos_j$ (positive product) is computed by C10 and CA2 and then stored in a memory called MD (Figure 5(b)). In the type 3 stage, the binary states of hidden neurons, the values of $hpos_j$, are read from the NU, and the values of $vneg_i$ are computed by the SU and stored in the MX memories, as shown in Figure 5(c). In this reverse-network stage, the contents of MR memories are used to address the connection weights stored for the forward network. In type 4 stage, the values of $vneg_i$ are read from the NU and the values of $hneg_j$ are computed by the SU. Meanwhile, the values of $vneg_i$ and w_{ij} in the SNU are queued in Q2 and Q1, respectively. When $pneg_j$ is computed at the SU, the $vneg_i$ is read from the Q2 and $\varepsilon \times vneg_i \times pneg_j$ (negative product) is computed by C10 and CA2. In the same clock cycle, the value of the positive product previously stored in MD is read, and the difference of the positive and negative products is computed by CA3. The value of w_{ij} previously queued in Q1 is read and added with the output of CA3 by CA4. The result of CA4 is then saved in MW, as shown in Figure 5(d).

The data rate at the output of the NU and SNUs are 64 connections per clock cycle, and the number of SBs (groups of 64 connections) processed in each stage are 6500, 6272, and 6500, when the size of RBM is 784×500. A timing gap between consecutive stages is required to flush out the pipelines. Even when the pipeline delays become large (greater than 100 clock cycles), their effect on the overall performance is small.

B. *Hardware Simulator in MATLAB*

The hardware simulator code provided at the website in [11] simulates the circuit in Figure 4 that computes a 784×500×500×2000 DBN that is capable of training 28×28 pixel grayscale images. This code can run on MATLAB without additional files.

This code is composed of two parts. The first part is the initialization part, which shows how the contents of the memories and SOT can be stored. The rest of the code, the clock loop, is for executing the simulation. The variable *ck* in the loop represents the system clock cycle, and each *ck* loop simulates a single clock cycle. The arrays RA and R are used for registers, and CA and C are used for arithmetic operators. Most *if* statements function as multiplexers that select from one of two values according to the condition. At the end of the clock loop, the registers and pipelined operators are shifted one step forward. The code has properties similar to hardware description languages. For example, statement "`C(2,1) =`

`R(10,2) + R(4,2);`" functions as adder C2, whose two inputs are connected from the outputs of R10 and R4. The statement "`R(8,1) = C(2,7);`" denotes the connection from the output of C2, whose pipeline delay is six, to the input of R8. The statements at the same level are exchangeable without affecting the functionality. This code can be translated into a hardware description language without difficulty. For an example, a multiplexer selecting one of two signal sources can be translated into a VHDL code as follows.

Simulator code	VHDL

`if hS==3`
`  RA(4,1,k) = RA(3,2,k);`
`else`
`  RA(4,1,k) = Csb(2);`
`end`
	`if (hS="011") then`
	`  RA4_i <= RA3_o;`
	`else`
	`  RA4_i <= Csb(2);`
	`end if;`

By default, eight digit images are used as training samples, which are embedded in the MATLAB software. This simulator can train MNIST handwritten digit database by setting a switch at the beginning of the code. During the execution of the simulator, the internal states are displayed, as shown in Figure 6.

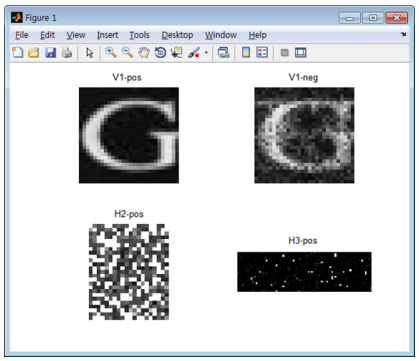

Fig. 6. Display of simulator code, showing current training sample, confabulation image, and hidden layers.

The simulator code was translated into a VHDL code and synthesized on a mid-range Xilinx Kintex 7 FPGA chip operating at a frequency of 200 MHz. The number of SNUs, P, was 32. As the capacity of the on-chip memory is limited (max. 16Mb), the FPGA implementation executed smaller DBNs (784×200 and 784×100×100×500). A total of 171 single-precision FP operators are used throughout the system. In particular, a Xilinx® exponent FP operator [12] and a 24-bit linear feedback shift register (LFSR) are used for computing the sigmoid function and random number generator in the SU, respectively. A MicroBlaze (MB) processor is synthesized for loading the MNIST training data into the main memory.

V. Results and Discussion

Figure 7 depicts the footprint of the arithmetic operators in the implemented system. Each colored dot in this figure denotes an active output of the respective arithmetic operator at a specific clock cycle. The slits in the figure originate from the null connections. The operators in the SU operate only once in a few clock cycles, as shown in the circle, while the

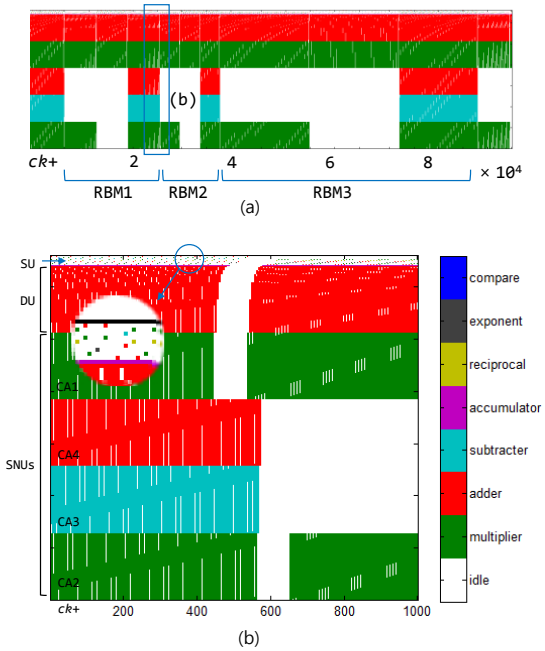

Fig. 7. Footprint of 331 arithmetic operators: (a) stages for one training sample, (b) detailed view. Each colored dot denotes an active output of the respective arithmetic operator at a specific clock cycle. The slits in the colored block originate from the null connections. Average utilization rate of the operators is 62.3%.

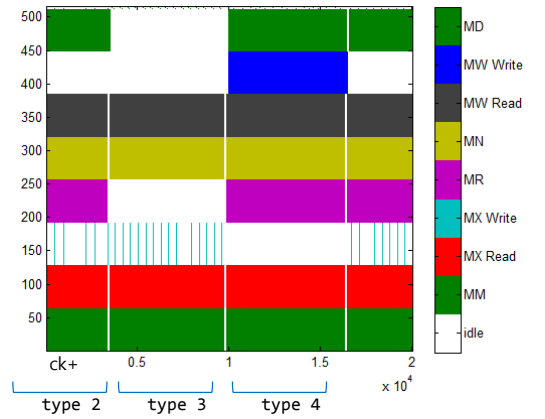

Fig. 8. Footprint of 516 memories in the system (including dual ports). Data is written on MX at intervals, whereas other memories are accessed continuously. Average memory access rate: 2.6 Tbps.

operators in the other units generate outputs continuously. All operators are fully operating in the type 4 stage, whereas only one-fourth of them are active in the type 3 stage. A total of 331 operators in the system produced their outputs for 62.3% of the number of clock cycles on average throughout the computation, as shown in Table 1.

TABLE I
Utilization Rates of Operators and Memories

Stage type	Arithmetic operators	Memories	Memory access rate in FPGA (Gbps)
Type 1	56.0%	73.5%	832
Type 2	37.7%	49.0%	532
Type 3	93.2%	85.7%	1014
Average	62.4%	69.7%	793

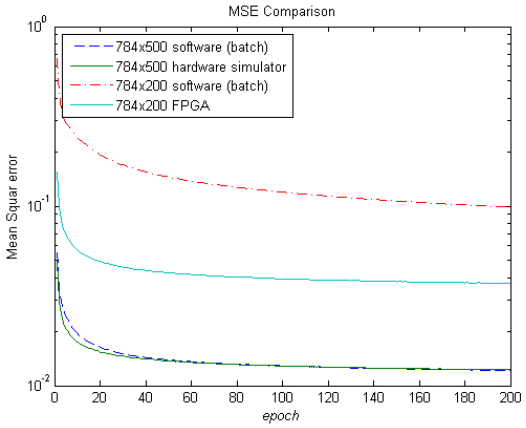

Fig. 9. Comparison of MSE with software implementations.

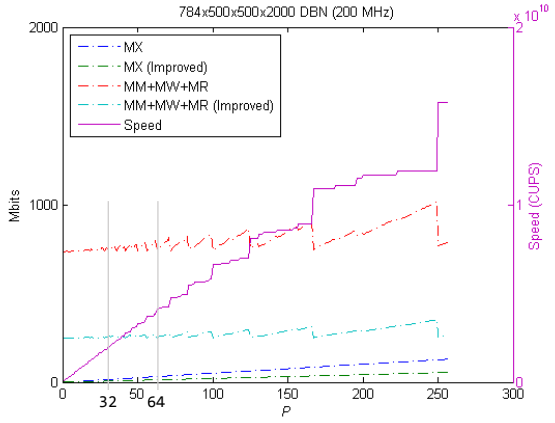

Fig. 10. Speed and memory requirements varying with P

The unevenness of computations among the stages are the major source of the idle clock cycles for the arithmetic operators. The proportions of idle cycles resulted from the null connections and pipeline delay are only 3.1% and 1.4%, respectively.

Figure 8 shows the footprint of the memories. A total of 321 memories (516 including dual ports) demonstrated a memory access rate of 2.6 Tbps throughout the computation when a 200-MHz clock is assumed. An average memory access rate of 793 Gbps is achieved for the FPGA implementation with P = 32 and single-precision data.

The results from the hardware simulator were identical to the sequential program when the initial weights and the output sequence of the random function were the same. The mean square errors (MSEs) of our implementations are compared with the results of software counterparts in Figure 9. Our systems exhibited earlier convergence than the sequential batch code that is provided in [13]. This is because batch computations require more epochs to converge [6], although they are favored for dense matrix computation in the software implementations.

TABLE II
COMPOSITION OF THE RESOURCES IN THE FPGA IMPLEMENTATION

		LUT	FF	LUT Rate
Arithmetic operators		73713	71884	86.9%
	SNU	58034	58504	68.4%
	DU	10901	9822	12.8%
	SU	4778	3559	5.6%
MB processor		10371	10239	12.2%
CU		786	2748	0.9%
Total used		84870	84871	100%
Total available on chip		203800	407600	

In Table 2, the resources used to implement the FP operators compared with the total resources synthesized for the system in the FPGA implementation. The non-operator portion includes an MB processor with a DDR memory controller, the CU, and glue logics in the system. Only a small portion of FPGA resources is used to implement the system. Note that the proportion of arithmetic operators will decrease by a factor of approximately five if fixed-point arithmetic operators are used instead of FP operators [14]. Even with the fixed-point arithmetic operators, the operating cost (non-operator portion) of our system is much less than the CPU case. Xilinx's Vivado FPGA tool estimated a power consumption of 5.1 watts for our design.

A simulated time of 9.4 s was taken for the MATLAB simulator to train a 784×800 RBM network with 60000 MNIST samples when 200-MHz system clock was assumed. The FPGA implementation trained a 784×200 RBM network with the same samples in 4.7 s, which corresponds to approximately half of the speed of the MATLAB simulator, even though the size of the network is limited by small on-chip memory space.

Figure 10 shows the changes in the computational speed and memory requirements of the NM systems when P is increased and the size of the DBN is 784×500×500×2000. Total memory space can be substantially reduced, and the computational speed increases linearly along P. In fact, the proposed system is a synchronous architecture and the computational speed is predictable. Using the method introduced in [5], the computational speed, in connection update per second (CUPS), can be calculated as:

$$speed = \frac{\sum_l^L |\mathbf{v}_l| \cdot |\mathbf{h}_l|}{\sum_l^L \left(\left\lceil |\mathbf{v}_l| / P \right\rceil \cdot |\mathbf{h}_l| \cdot 2 + \left\lceil |\mathbf{h}_l| / P \right\rceil \cdot |\mathbf{v}_l| + 3 \cdot d_p \right)} \times f_{ck},$$

where L is the number of RBM layers, $\mathbf{v}_l$ and $\mathbf{h}_l$ are $\mathbf{v}$ and $\mathbf{h}$ in layer l, respectively, d_p is the pipeline delay, and f_{ck} is the system clock frequency.

Table 3 compares the speed of our implementations with other known systems.

TABLE III. PERFORMANCE COMPARISON

Ref.	Type	Internal	Clock (Hz)	Speed (CPS)	Speedup over PC
This	Simulator	MATLAB RTL simulator	200 M	4.0 G	255×
This	FPGA	Xilinx Kintex 7	200 M	1.9 G	121×
[15]	GPU	1000 × GPUs (16000 cores)	N/A	38 G	2420×
[2]	GPU	NVIDIA GeForce 460 (336)	720 M	721 M	46×
[5]	GPU	NVIDIA GTX 280 (240)	1.3 G	672 M	43×
[2]	CPU	Intel i5-2410M (2)	2.3 G	15.7 M	1×
[5]	CPU	Intel Core2 Quad core	2.83 G	10.2 M	0.7×

Our systems outperform most other systems, except the one with 1000 GPUs.

High-end FPGA chips such as Xilinx Virtex 7 contain hardware resources required to build the same system as the one simulated by the MATLAB hardware simulator. However, it would not be possible to build larger systems as the total on-chip memory space in the FPGA is limited. In order to build a system for large-scale DBNs, an application-specific integrated circuit (ASIC) or a board-level design may be required.

In the systems where the computation of DBN is carried out by computing matrices, a substantial effort is required to transpose the weight matrix in order to compute $vneg_i$. In our architecture, weights in the forward and reverse networks are shared by using MR memories without moving or copying data. Further, as there is no limitation on the network topology, large sparse DBNs can be computed without decreasing the computation time, as contrasted with most matrix-based systems [6].

VI. Conclusion

An efficient special purpose hardware architecture for a DBN was proposed and implemented. By maximizing the utilization of arithmetic operators and minimizing the overheads in the usage of hardware resources, a high efficiency for the chip area and power consumption could be achieved while providing good performance. The proposed architecture can be used for high-performance real-time DBN systems.

Acknowledgment

The author gratefully acknowledges Geonho Han, Terry Ahn, and Jaehwa Lee for encouragements and valuable comments.

References

[1] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," Science, vol. 313, pp. 504-507, 2006.

[2] N. Lopes, B. Ribeiro, and J. Gonçalves, "Restricted boltzmann machines and deep belief networks on multi-core processors," in Neural Networks (IJCNN), The 2012 International Joint Conference on, 2012, pp. 1-7.

[3] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," Micro, IEEE, vol. 32, pp. 20-27, 2012.

[4] H. Makino, H. Suzuki, H. Morinaka, Y. Nakase, K. Mashiko, and T. Sumi, "A 286 MHz 64-b floating point multiplier with enhanced CG operation," Solid-State Circuits, IEEE Journal of, vol. 31, pp. 504-513, 1996.

[5] D. L. Ly, V. Paprotski, and D. Yen, "Neural networks on gpus: Restricted boltzmann machines," see http://www. eecg. toronto. edu/~ moshovos/CUDA08/doku. php, 2008.

[6] S. K. Kim, P. L. McMahon, and K. Olukotun, "A large-scale architecture for restricted boltzmann machines," in Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, 2010, pp. 201-208.

[7] J. B. Ahn, "Neuron machine: Parallel and pipelined digital neurocomputing architecture," in Computational Intelligence and Cybernetics (CyberneticsCom), 2012 IEEE International Conference on, 2012, pp. 143-147.

[8] J. B. Ahn, "Extension of neuron machine neurocomputing architecture for spiking neural networks," International Joint Conference on Neural Networks (IJCNN2013), 2013.

[9] J. B. Ahn, "Computation of Backpropagation Learning Algorithm Using Neuron Machine Architecture," in Computational Intelligence, Modelling and Simulation (CIMSim), 2013 Fifth International Conference on, 2013, pp. 23-28.

[10] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," Neural computation, vol. 18, pp. 1527-1554, 2006.

[11] Anonymous. (2014). MATLAB hardware simulator for DBN. Available: http://neurocomputings.com/jerryahn/papers/dbn

[12] Xilinx, "LogiCORE IP Floating-Point Operator v6.2," 2011.

[13] G. E. Hinton. Training a deep autoencoder or a classifier on MNIST digits. Available: http://www.cs.toronto.edu/~hinton/MatlabForSciencePaper.html

[14] Altera, "Taking advantage of advances in FPGA floating-point IP cores WF-0116.1.0," 2009.

[15] Q. V. Le, "Building high-level features using large scale unsupervised learning," in Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, 2013, pp. 8595-8598.