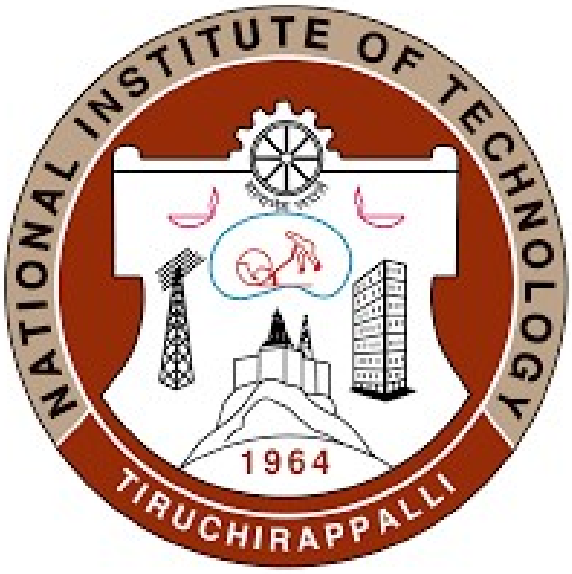
INTERNSHIP REPORT

NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI



Name : BILAKANTI RAMYA SRI
Roll No. : 108116016
Branch : ECE

Project on

HIGHLY PARALLEL RESTRICTED BOLTZMANN MACHINE(RBM)

Place of intern

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY, BANGALORE

Under the guidance of **Dr. Nanditha Rao**

Duration: 15th May – 15th July 2019

Contents:

INTRODUCTION ABOUT THE INSTITUTE:

The International Institute of Information Technology Bangalore, a Deemed University, popularly known as IIIT-B, was established in 1999 with a vision to contribute to the IT world by focusing on education and research, entrepreneurship and innovation. The Institute is a registered not-for-profit society funded jointly by the Government of Karnataka and the IT industry.

Since its inception, IIIT-B, with its unique model of education, research, and industry interaction, has grown in stature to become an institution of considerable repute in academic as well as corporate circles. The Institute works in partnership with the corporate sector, while retaining the freedom of an academic institution. It is inspired by other renowned institutions, and also strives to emulate an academic culture that is on par with the best international institutions.

Mission:

To build on the track record set by India in general and Bangalore in particular, to enable India to play a key role in the global IT scenario through a world class institute with a focus on education and research, entrepreneurship and innovation.

Vision:

To contribute to the IT world by focusing on education and research, entrepreneurship and innovation

ABSTRACT

GROWING attention has been paid to deep learning (DL) owing to its successful performance in practically useful applications, e.g., classification and prediction, exceeding conventional machine learning algorithms. Nevertheless, significant computational time and large power dissipation are demanded until the structure is completely optimized with good enough parameters to obtain intelligent operations. The learning algorithms used for modeling executions are slow for large-scale applications; therefore, hardware-efficient calculation methods adjustable for DL are needed. Most learning time is spent iteratively updating connection weights of the network. The aim of unsupervised DL is to find some structure in the data and cluster individual data into groups; their representative model is the restricted Boltzmann machine (RBM). RBMs can be used in a hybrid deep network as deep belief networks

Restricted Boltzmann Machines (RBMs) are an effective model for machine learning; however, they require a significant amount of processing time. In this study, we propose a highly parallel, highly flexible architecture that combines small and completely parallel RBMs. This proposal addresses problems associated with calculation speed and exponential increases in circuit scale. We show that this architecture can optionally respond to the trade-offs between these two problems. Furthermore, our FPGA implementation performs at a 134 times processing speed up factor with respect to a conventional CPU. In this work we have implemented the highly parallel RBM architecture which gives comparatively high speedup

1. Modeling Soft Errors at the Device and Logic Levels for Combinational Circuits

The following content is taken directly from the paper. Soft errors are transient errors caused mainly due to high-energy particle strikes from cosmic radiation. Such radiation directly or indirectly induces localized ionization capable of upsetting internal data states. The particle strikes can directly occur on state elements such as memories, flip-flops, and latches and change their state. Additionally, state elements can latch incorrect values propagated from strikes that occur in combinational elements. Alpha particles, high-energy cosmic ray particles, and neutron-induced Boron-10 (^{10}B) fission are the most significant sources of soft errors. Of these, in the absence of ^{10}B due to the elimination of Boro phospho silicate Glass (BPSG), soft errors caused by high-energy cosmic-ray-induced neutrons are the most dominant . Cosmic rays of galactic origins interact with the Earth's atmosphere to produce a cascade of energetic particles. These particles include neutrons, protons, electrons, muons, pions, and gamma rays . These energetic particles interact with silicon either directly or indirectly, producing electron-hole pairs. Under the right conditions, these electron-hole pairs get collected and create a pulse, which, if significantly large, can cause an error. This form of an upset is also called a Single Event Upset (SEU). As it is possible to recover from these errors, these are also called soft errors. Soft-Error Analysis Toolset (SEAT) is a hierarchical toolset that models soft errors across the device, logic, and architectural levels. SEAT-Device Analysis (SEAT-DA), models soft error by using nuclear and device physics tools, with an aim of creating a transient current waveform library that captures different process and operating conditions that impact the Soft-Error Rate (SER). With reducing pipeline depth and downscaling of nodal capacitance and supply voltages, radiation induced soft errors in the combinational logic is gaining increasing attention and is expected to become as important as directly induced errors on state elements. However, the simulation results in indicate that at the 65-nm node for commercial CPUs, the combinational logic does not seem to be a large contributor to the overall product/chip SER. However, the combination of logical, electrical, and time window masking effects in data paths make it difficult to accurately predict the SER in combinational circuits. In this paper, we propose a new approach to estimate the SER of logic circuits, attempting to capture electrical, logic, and latch window masking concurrently. Our approach is applied to designs that use cell libraries characterized for soft-error analysis and utilizes analytical equations to model the propagation of a current pulse to the input of a state element. A tool

known as SEAT Logic Analyzer (SEAT-LA) has been developed using the above methodology. The input to SEAT-LA is a gate level netlist of a combinational circuit synthesized using cells pre-characterized for soft-error analysis. The Burst Generation Rate (BGR) proposes that if the energy collected in the sensitive volume, as well as the charge collected in the sensitive volume, exceeds a critical charge, an error is said to have occurred. The energy of the ions is calculated by using nuclear codes. NUSPAs (Monte-Carlo-based nuclear codes) are used to model the reaction products from the neutron-Si (n-Si) interactions, and FEM-based methods are used to model the device behavior. This radiation-induced current pulse is abstracted as follows

$$I(t) \propto \frac{Q}{t} * \sqrt{\frac{t}{T}} * \exp\left(\frac{-t}{T}\right)$$

where Q refers to the amount of charge collected due to the particle strike. The parameter T is the time constant for the charge collection process and is a property of the CMOS process used for the device. In the case of neutron-induced soft errors, calculating Q and T is not trivial, as it is dependent on the ions generated by the n-Si interaction, the device structure, and other physical properties. In our methodology, we use Monte Carlo N Particle (MCNP) codes for modeling n-Si interaction, TRIM for estimating the charge deposition, and a commercial 3D device simulator (Synopsys Davinci) for capturing the device effects on charge collection. In addition, our methodology is technology independent. A probabilistic approach is adopted, in which the electrical and timing window (tw) masking are determined based on the capacitance evaluation and pull up/pull down current drive at each node. In [28], electrical masking is categorized into two effects: increase in rise, and fall time and delay degradation. It models these two separately and then combines them to model electrical masking. In contrast, our approach characterizes each cell in the library. A mathematical model based on setup and hold time was used for tw estimation, whereas the electrical masking effect was determined using noise rejection curves on various gates. In our soft-error analysis approach, we propose a new method to account for logical, electrical, and tw masking together rather than as separate entities. Our approach is based on modeling the pulse propagation in terms of the delay of each gate and characterizing the tw for the flip-flops. The inaccuracies in modeling the masking effects of soft errors in combinational logic arise due to the absence of accurate models for the glitch propagation used in these algorithms.

However, the magnitude of the output glitch alone is not sufficient, as the information on glitch width is also required for effective modeling. Hence, in this work, a mathematical expression for the pulse width is used based on approximating the output voltage to a trapezoidal or a triangular pulse. The pulse

should occur within the setup and hold time of the latching element for a soft error. Pulses that occur outside this window, as in two of the cases illustrated, do not result in soft error.

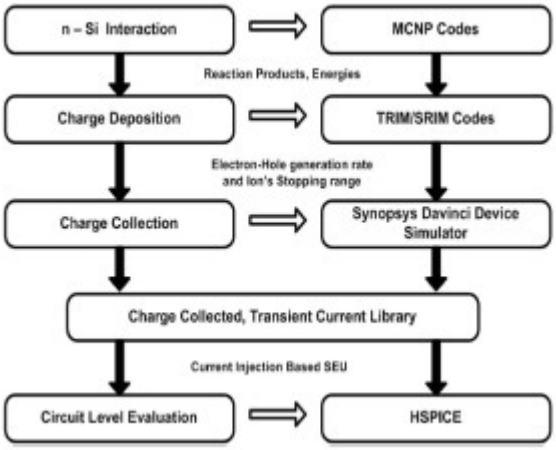


Fig: SEAT-DA tool flow

It models both charge deposition and charge collection. MCNP is used for studying neutron, photon, electron, or coupled neutron/photon/ electron transport. n-Si reactions can be classified into two main groups: elastic and inelastic. MCNP can model both elastic and inelastic scattering. Elastic scattering, due to the low mass of the neutrons, does not produce significant ionizations. In contrast, inelastic reactions occur when the neutron enter the nucleus and the unstable nucleus disintegrates to smaller particles. TRIM is used to calculate the stopping power of ions. In terms of susceptibility, a transient pulse caused by inelastic scattering is of higher magnitude than elastic scattering errors. However, it should be noted that these are fewer in number in comparison to elastic scattering. A circuit designed for these conditions will be immune to errors due to elastic scattering. Due to Alpha-particle source-drain penetration (ALPEN), if a particle strike passes through both the source and the drain at near-grazing incidence, a significant but short-lived source-drain conduction current that mimics the on state of the transistor is generated. The electron-hole pairs are introduced in the simulation as a charge column. Neutrons are highly penetrative and have a very low cross section. Of the total 5 million neutrons simulated, we saw only 128 collisions. However, these 128 collisions can result in different reactions and, hence, different reaction products. We find that at a low energy, one or two reactions dominate, whereas at a higher energy, more reactions start occurring. The typical current pulse height is about 1.8 mA for the dominant ions, and the charge collection time is about 0.8 ns. The magnitude of the current pulse is dependent on the energy and the size of the ion. This indicates the reduction in contributions of drift processes like ALPEN and funneling. Diffusion-dominated current pulses have a long decay time,

resulting in a long tail and very low peak value, drift processes dominate the charge collection. We find that NMOS can collect more charge and hence is more susceptible to soft errors. Although the SEU generates a transient current pulse, the operation of the circuit is actually affected by the resulting voltage glitch caused by the collected current. Even though the current transients settle in about 1.4 ns, the voltage transients last for longer time. As we increase the nodal capacitance, the magnitude of the voltage transient reduces. In fact, for an increase in the capacitance by five times, the magnitude of the pulse reduces by half. For about 500 times of increase in capacitance, the voltage transient is almost negligible.

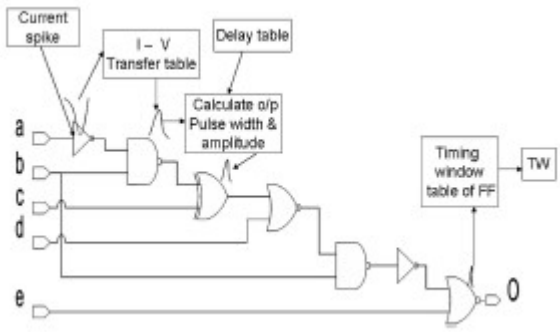


Fig: SEAT-LA: estimation methodology.
The SER can be calculated as

$$SER = (\sum_{p} P\mathrm{p} \sum_{\mathrm{n}} \mathrm{PN.\,Pnode})$$

Where p is the number of pulses, Pp is the probability of a certain current pulse being generated due to a particle hit at the node N, and n is the total number of nodes in the circuit. SEAT-LA also has a maximum speed up of 27,000 times over the HSPICE simulation, whereas the average speedup is 15,000. Based on n-Si interaction, we found that the charge collected in the silicon depends on the reaction products and their energy. The charge collection is through diffusion process. We also found that the charge collection is weakly dependent on voltage, substrate bias, and angle.

2. Error Tolerance Analysis of Deep Learning Hardware Using a Restricted Boltzmann Machine Toward Low-Power Memory Implementation

The following content is taken directly from the paper. Growing attention has been paid to deep learning (DL) owing to its successful performance in practically useful applications, e.g., classification and prediction, exceeding conventional learning algorithms. Nevertheless, significant computational time and large power dissipation are demanded until the structure is completely optimized with good enough parameters to obtain intelligent operations. Most learning time is spent iteratively updating connection weights of the network. The aim of unsupervised DL is to find some structure in the data and cluster individual data into groups; their representative model is the restricted Boltzmann machine (RBM). RBMs can be used in a hybrid deep network as deep belief networks. The structural property of RBMs is associated with expected robustness against memory errors due to its possibly redundant network topologies; however, the detailed fault tolerance of DBN assuming highly scalable hardware has never been studied. A memory block that stores structural data in order to reduce the power consumption, particularly for cache memory.

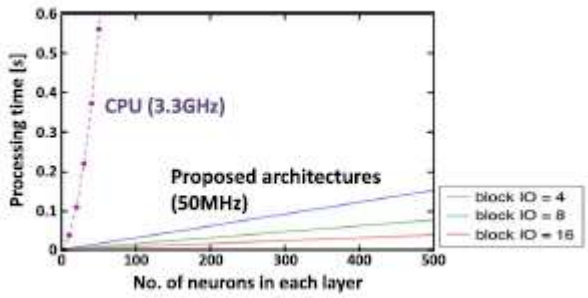


Fig: Architectural simulation of processing speed.

RBMs are a stochastic neural network model consisting of two layers (a visible layer and a hidden layer). The network is an undirected graphical model in which the neurons in one layer are fully connected to the neurons in a second layer. RBMs have three parameters: connection weights, visible biases, and hidden biases. RBMs learn in an unsupervised manner as they are updated. At this point, all visible layer and hidden layer combinations are calculated. Second, the visible layer is calculated using sampling results from the hidden layer as input. The size ratio of the visible and hidden layers of the RBMs is equal to 1:1. For each developed model, time is linear with respect to the data size, whereas the

conventional CPU operation at 3.3 GHz has an exponential characteristic due to the quadratic relationship between hidden and visible nodes that also reflects into the simulated software. Our hardware shows fast speed with respect to connection updates per seconds (CUPS) even using a slower FPGA clock rate than other developments. The Mixed National Institute of Standards and Technology (MNIST) handwritten data set is used to analyze classification accuracy [1]. As the parameters learned by the DBN are in the form of a matrix for each RBM, the fault-injection function replaces a number of elements with a substitute value in memories. The DBN maintains a good accuracy of 97% after 15 epochs even considering initial faults altering RBMs in the memory and the fact that the accuracy depends on the initial errors before the fine-tuning. Results show that tolerance is higher in static faults than in dynamic faults. When the faults are injected during the sequences, different tendencies can be observed, particularly for higher error rates, resulting in sum-of-error values much higher than their initial values. From these analyses, we found memory fault tolerance caused by the following: 1) sticking involved nodes to logic 0; 2) lowered absolute value; and 3) a plus sign replacing a minus sign. Bit vanishing severely deletes all the bits of the value, which means the error type of "0." Bit flipping at the integer or precision bits makes random values, e.g., +0.5/−0.5. It is suggested that cache memory with errors lower than approximately 3% can be used without any error correction. SRAM delay increases at a higher rate than CMOS logic delay, as the supply voltage is decreased. By decreasing the supply voltage for low-power integrated circuits, operating conditions of SRAM should be retained.

This brief has quantified the robustness of a DBN consisting of our custom RBM hardware. The classification accuracy shows remarkable robustness under various error rates in memory for possible learning cases. The results confirm robustness against memory errors, which is promising for DL hardware. Due to this promising feature, power can be expected to be reduced by 1/3 in the 130-nm node SRAM for the DL hardware in conventional CMOS and the originally presented SONOS-based technology.

3. Large-scale Deep Unsupervised Learning using Graphics Processors

The following content is taken directly from the paper. The promise of unsupervised learning methods lies in their potential to use vast amounts of unlabeled data to learn complex, highly nonlinear models with millions of free parameters. Unfortunately, with current algorithms, parameter learning can take weeks using a conventional implementation on a single CPU. If the goal is to deploy better machine learning applications, the difficulty of learning large models is a severe limitation. Analogously, in our view, scaling up existing DBN and sparse coding models to use more parameters, or more training data, might produce very significant performance benefits. Sparse coding exhibits a qualitatively different and highly selective behavior called "end-stopping" when the model is large, but not otherwise. Popular learning algorithms such as logistic regression, linear SVMs and others can be easily implemented in parallel on multi core architectures, by having each core perform the required computations for a subset of input examples, and then combining the results centrally (Dean & Ghemawat, 2004; Chu et al., 2006). However, standard algorithms for DBNs and sparse coding are difficult to parallelize with such "data-parallel" schemes, because they involve iterative, stochastic parameter updates, where any update depends on the previous updates. There is of course a tradeoff this parallelism is obtained by devoting many more transistors to data processing, rather than to caching and control flow, as in a regular CPU core. This puts constraints on the types of instructions and memory accesses that can be efficiently implemented. Thus, the main challenge in successfully applying GPUs to a machine learning task is to redesign the learning algorithms to meet these constraints as far as possible. The GPU hardware provides two levels of parallelism: there are several multiprocessors (MPs), and each multiprocessor contains several stream processors (SPs) that run the actual computation. The computation is organized into groups of threads, called "blocks", such that each block is scheduled to run on a multiprocessor, and within a multiprocessor, each thread is scheduled to run on a stream processor. All threads also have access to a much larger GPU-wide "global memory" (currently up to 4 GB) which is slower than the shared memory, When memory accesses are coalesced, the hardware can perform them in parallel for all stream processors, and the effective access speed (between the stream processors and the global memory) is several times faster than the access speed between a CPU and RAM. A partial solution is to perform memory transfers only in large batches, grouped over several computations. Thus, efficient use of the GPU's parallelism requires careful consideration of the data flow in the application. DBNs are multilayer neural

network models that learn hierarchical representations for their input data. DBN is greedily built up layer-by-layer, starting from the input data. Each layer is learnt using a probabilistic model called a restricted Boltzmann machine (RBM). Since each update requires a Gibbs sampling operation, and the updates have to be applied over many unlabeled examples to reach convergence, unsupervised learning of the parameters can take several days to complete on a modern CPU. Sparse coding is an algorithm for constructing succinct representations of input data. Sparse coding attempts to learn that each handwritten character is composed of only a few building blocks, such that each input x can be represented as a linear combination of a few basis vectors:. Such a higher-level representation can then be applied to classification tasks, where it leads to good results even with limited labeled data. For problems with high dimensional inputs and large numbers of basis vectors, the first step is particularly time consuming as it involves a non-differentiable objective function, and the overall learning algorithm can take several days. First, memory transfers between RAM and the GPU's global memory need to be minimized, or grouped into large chunks. For machine learning applications, we can achieve this by storing all parameters permanently in GPU global memory during learning. A second requirement is that the learning updates should be implemented to fit the two level hierarchy of blocks and threads, in such a way that shared memory can be used where possible, and global memory accesses can be coalesced. Often, blocks can exploit data parallelism, while threads can exploit more fine-grained parallelism.

$$P(h|x) = \text{vectorSigmoid}(b + wT\ x)$$
$$P(x|h) = \text{vectorSigmoid}(c + wh)$$

We extend our method to learning deep belief networks with "overlapping patches". These overlapping patch RBMs can be stacked on top of each other, such that the second-layer RBM contains hidden units connected locally to first-layer hidden units, and so on. The GPU method is between 12 to 72 times faster. The speedup obtained is highest for large RBMs, where the computations involve large matrices and can be more efficiently parallelized by using a large number of concurrent blocks. Contrastive divergence learning in this model can be implemented by using convolutions to perform the Gibbs sampling operation h|x. For small to medium filter (patch) sizes, spatial convolution can be implemented very efficiently using GPUs, by having each block read a filter into shared memory, then reading the input image column by-column into shared memory, and finally aggregating the output elements affected by that filter and that input image column. This algorithm uses fine-grained parallelism by having each thread compute just one coordinate of the solution. Such highly multithreaded execution is especially well-suited for graphics processors, as the hardware is able to hide memory latency by scheduling other threads that are not blocked on memory

accesses, and leads to high utilization of the available cores. By effectively parallelizing this step, our GPU method is up to 15 times faster than a dual-core implementation. Graphics processors are able to exploit finer-grained parallelism than current multi core architectures or distributed clusters. They are designed to maintain thousands of active threads at any time, and to schedule the threads on hundreds of cores with very low scheduling overhead. In contrast, the two-level parallelism offered by GPUs is much more powerful: the top-level GPU blocks can already exploit data parallelism, and GPU threads can further subdivide the work in each block, often working with just a single element of an input example.

4. A HIGHLY SCALABLE RESTRICTED BOLTZMANN MACHINE FPGA IMPLEMENTATION

The following content is taken directly from the paper. Restricted Boltzmann Machines (RBMs) - the building block for newly popular Deep Belief Networks (DBNs) are a promising new tool for machine learning practitioners. Our design uses a highly efficient, fully-pipelined architecture based on 16-bit arithmetic for performing RBM training on an FPGA. A Deep Belief Network is a multilayer generative model where each layer encodes statistical dependencies between the units in the layer below it; it is trained to maximize the likelihood of its training data. Modem FPGAs contain a large amount of configurable logic elements, which allow custom designs for complicated algorithms to be built. First, our approach generalizes the data representation of the network from binary numbers to fixed-point numbers. Modem FPGAs contain abundant multiplier resources, whether they are used or not - we exploit these resources so that a wider range of experiments can be conducted on the RBM accelerator.

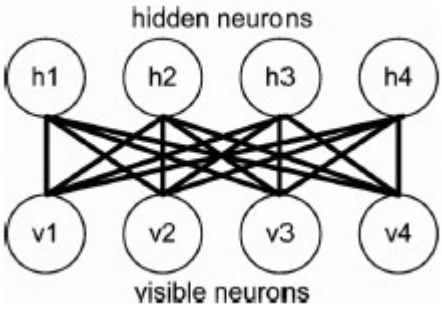


Fig: Illustration of RBM Network.

The goal of training is to learn visible-hidden connection weights and neuron activation biases such that the RBM learns to reconstruct the input data during the phase where it samples the visible neurons from the hidden neurons. Ideally, given enough layers, the user can learn very abstract features of the training set, with the intention of modeling the hierarchical learning structure of the brain. Some recent work using DBNs includes an application to classifying handwritten digits and comparison of sparse DBN output to the V2 area of the visual cortex. In a software implementation of an RBM running on a Sun Ultra Sparc T2 processor, the percentage of runtime consumed in matrix multiplication is between 90% for a small (512x512) network and 98% for a larger (2500x2500) network. Hence, the algorithm can run considerably faster by accelerating the matrix multiplication.

FPGAs are a promising technology with which to accelerate DBN training; the presence of fine-grain parallelism that can be exploited in the matrix multiplications is a promising attribute. The built-in hardware multiplication units in modem FPGAs now support very similar precision calculations. The FPGA board can also be connected to up to 19 other boards in a stack via a high speed interface – this will be used in future work to scale up the size of the DBNs that can be processed. At a high level, the RBM module has an array of weights and neurons that are fed into an array of multipliers, and then into adders to perform the matrix multiplication. After that, the RBM computes the sigmoid to obtain the probability of firing a neuron and fires the neuron using a comparator and random number generator. After the positive and negative phases, the module continues to iterate until it meets the stopping condition given by the user. Every step is pipelined, which results in a throughput of approximately 256 multiply-and-add (MADD) per cycle. The rationale for such partitioning is that wire delay increases as semiconductor technology scales, so the wire delay becomes the performance bottleneck if the placement and routing is not performed efficiently. Localization of communication is an efficient way, and possibly the only way, to fully exploit all the parallelism in modem FPGAs. Partitioning the design into multiple groups also makes the design scalable. Since most of the algorithm is performed within each group, the design can be migrated to a future device easily by instantiating more of these groups without having to worry about wire delays or routing. However, for maximum performance, the number of neurons should either be a multiple of the number of multipliers, or vastly larger than the number of multipliers.

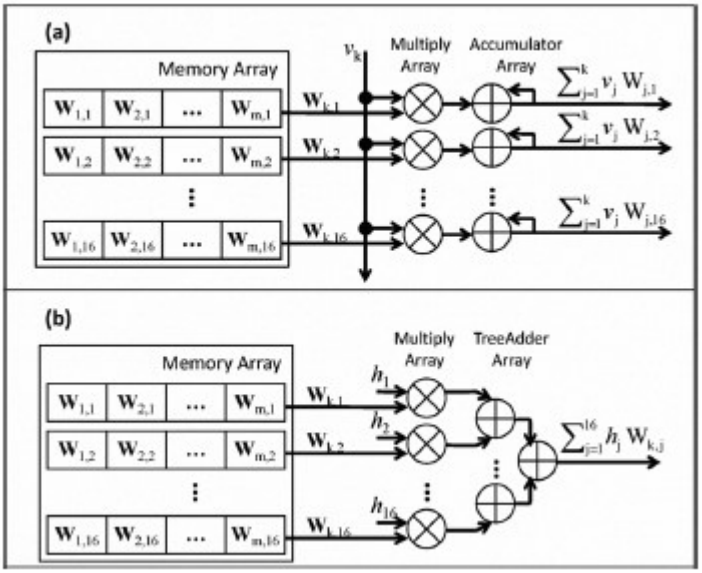


Fig. Matrix Multiplication for Computing (a) hidden neurons

(b) visible neurons

Although locating the inputs close to the multipliers is desirable, distribution of the weights is non-trivial due to a transpose operation that occurs during the visible neuron sampling phase. This transpose problem can be avoided by distributing the data such that no embedded RAM will simultaneously read out two or more elements from the same row with the same address, and no embedded RAM will contain two or more elements of the same column. Then, by using a carefully designed addressing scheme, a column or row of the matrix is read out from the memory each cycle and no communication is required for the transpose. Although this approach eliminates communication for the transpose operation, it has two major drawbacks. One is that the weight and data matrices have to be shifted before being written into the on-chip memories, requiring a sophisticated routing scheme from each RAM to the appropriate multiplier since no row or column vector of the weight matrix contains the same index number. When the embedded RAMs can no longer be equal to the number of neurons, a different weight matrix routing logic is required each time a portion of the weight matrix is streamed in, severely limiting the scalability. The update phase involves multiplying the transpose of the visible neuron matrix and the hidden neuron matrix, which is essentially the sum of outer products between the visible and hidden neurons. Since the sigmoid function in hardware is an expensive operation - requiring exponentiation and division - we instead implemented an approximate sigmoid function design called PLAN (Piecewise Linear Approximation of Nonlinear function) since it uses only a minimal number of addition and shift operations. The stochastic characteristics of an RBM are also greatly influenced by the quality of the random number generator (RNG), which is a combination of an LFSR (Linear Feedback Shift Register) and a CASR (Cellular Automata Shift Register), providing good statistical properties. It should be noted that partitioning the design may increase the total logic count. Thus, partitioning can be seen as a trade off of scalability and performance against silicon area. Our RBM system runs 25 times as fast as the single precision software implementation and 30 times as fast as the double precision implementation. Weights will need to be streamed in from external storage such as DRAM. To tackle bandwidth issues, a batch size of at least 16 will be used.

5. A Large-scale Architecture for Restricted Boltzmann Machines

The following content is taken directly from the paper. Graphics processors have significant performance benefits over CPUs in matrix operations, but do not scale to very large networks due to I/O bandwidth limitations. The trade-offs of the network configuration and hardware resources, such as memory and IO bandwidth, using which it will be possible to build a custom DBN network that is best suited for a particular application. Matrix multiplication is responsible for more than 99% of the execution time for large networks. To train an RBM, samples from a training set are used as input to the RBM through the visible neurons, and then the network alternatively samples back and forth between the visible and hidden neurons. The goal of training is to learn visible-hidden connection weights and neuron activation biases such that the RBM learns to reconstruct the input data during the phase where it samples the visible neurons from the hidden neurons. Each sampling process is essentially matrix-matrix multiply between a batch of training examples and the weight matrix, followed by a neuron activation function, which in many cases is a sigmoid function $(1/ (1 + e^{-x}))$.

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

Fig: RBM training algorithm pseudo-code.

The sampling between the hidden and visible layers is followed by a slight modification in the parameters and repeated for each data batch in the training set, and for as many epochs as is necessary to reach convergence. To fully exploit the abundant parallelism in matrix multiply operations, a hardware RBM implementation should maximize the number of multipliers that can be supported by the memory bandwidth and logic gate resources, while reserving resources for other computation such as adders or the sigmoid function. The main challenge in designing a single FPGA RBM accelerator is deciding how to perform the matrix transpose. The training algorithm requires three matrix-matrix products: VW, HW^T, and V^TH. Suppose we have M multipliers. To fully utilize the M multiplier

resources, M weights must be read from the memory blocks each cycle. Notice the memory bandwidth requirement for visible neurons is one neuron per cycle to be, while the bandwidth requirement for a hidden neuron computation is a row vector of the hidden neuron batch H. If we view $V^T H$ as the sum of the outer products of each row vector of V and H, we can broadcast the visible node each cycle, which is multiplied by a row of hidden neurons to get the outer product. We can then sum the outer products using the existing accumulator. Since the matrix multiply structure for weight updates is basically the same as the hidden neuron computation phase. A partitioning algorithm is used to distribute the work amongst multiple FPGAs while minimizing the communication. However, the inter-chip network requires communication resources that increase quadratically with the number of neurons, which makes it difficult to scale to large networks. Therefore, instead of focusing on minimizing the amount of communication, we localize the communication to allow scalability. Localization enables the user to easily migrate the same design to a future technology and take advantage of the technology by adding more modules. The few operations that do require global communication are appropriately buffered to avoid long wiring. The main issue is in how to deal with the global communication across the chips, which includes visible neuron broadcast in the hidden neuron computation phase and tree add reduction for the visible reconstruction phase.

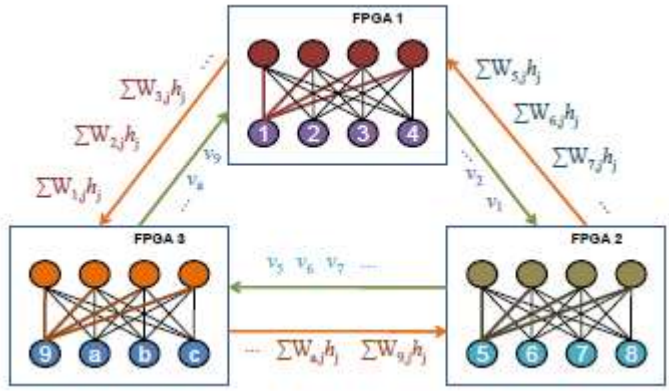


Fig: A high-level view of our architecture illustrating the inter-chip communications of a system with three FPGAs. Communications for both the visible and hidden neuron update phases are shown.

In the single FPGA implementation, visible neurons are broadcasted during the computation of the hidden neurons. However, broadcasting to multiple FPGAs would severely limit the scalability of our system. Thus, broadcasting the neurons is only done within the FPGA (with the appropriate buffering). Instead of broadcasting to all FPGAs, each FPGA passes the visible neurons it has read or received to its neighbor in one direction. Instead of performing the global add reduction all at once, we have each FPGA calculate the partial reduction for its

final destination FPGA and pass this result to the neighboring FPGA. This continues until the partial sums add up to be complete at the final destination FPGA, where the visible node is reconstructed. Since the hidden neuron computation requires one visible neuron broadcast per cycle, the I/O bandwidth requirement between the FPGAs is determined by the fact that each FPGA need only send one visible neuron to a neighbor per cycle, or less if there are more neurons than the number of multipliers per FPGA. The partial sums from visible data reconstruction also only require one partial sum per cycle. However, off-chip communication does have a certain amount of latency. To tolerate this latency, each FPGA must process its local data first while data is buffered in its input queue. Since the communication is always nearest-neighbor, the latency only has to be less than the batch size to keep the RBM pipeline full. As we increase the number of FPGAs, the number of multipliers increases proportionally, thus the number of neurons that can be computed per cycle also grows linearly. However, the embedded memory also only increases linearly, while the weight matrix increases quadratically. To overcome this issue and scale to large systems, the weight matrix must be streamed in from off-chip memory. One way to alleviate the memory bandwidth problem is to exploit the additional parallelism in the matrix multiplication by processing multiple visible training examples at once. This allows each weight to be fed into several multipliers rather than just one. However, as long as the number of neurons per FPGA does not fall below the number of multipliers, the speedup should remain approximately the same regardless of the number of neurons, since the speedup comes from the abundant multiplier resources. By exploiting the parallelism in multiple training examples, we can dramatically decreases the DRAM bandwidth requirement. However, the memory and IO bandwidth requirements both increase linearly with the clock frequency. Therefore, for a given communication capacity, it is generally more efficient to use FPGAs with more multipliers and reduce the clock frequency than attempting to gain performance by increasing the clock frequency. In conclusion, increasing the batch size helps reduce the memory bandwidth requirement. However, it is limited by the on-chip memory space, and it also increases the IO bandwidth requirement. We can exploit the sparseness of the weight matrix to increase the efficiency of computation. One simple way to make use of the sparseness is to limit In addition, for simplicity, we restrict connections of each neuron the fanout of each neuron to a constant number C to be the C nearest neighboring neurons. Then the dense matrix, if any, will occur only at the neighboring nodes. The left and right ends of the network are wrapped around such that all nodes have constant fanout. For a locally dense sparse network, the weight matrix can be considerably smaller, since the weight matrix grows as O(N) with the number of neurons due to constant fanout. This allows us to build a very large scale network, approaching the scale of

a human brain. Since graphics processor tend to perform better with larger matrices, the speedup may differ when scaled to larger networks. The energy efficient nature of FPGAs, in addition to the scalability of our design, makes our approach desirable for large-scale DBN implementations. A four-FPGA implementation has been demonstrated to show the feasibility of our approach, and showed a 46X-112X speedup with linear scalability.

6. High-Performance Reconfigurable Hardware Architecture for Restricted Boltzmann Machines

The following content is taken directly from the paper. Neural networks have captured the interest of researchers for decades because of their superior ability over traditional approaches for solving machine learning problems. However, there are significant difficulties in adapting current applications to commercial or industrial settings since software implementations on general-purpose processors lack the required performance and scalability. Sequential processors iterate through every connection in the network, which increases complexity quadratically with respect to the number of processing elements. Individual RBMs can scale up to sizes of 2000 × 500 nodes, taking weeks to train on a desktop computer. Thus, software programs of large RBMs are unable to satisfy the real-time constraints required to solve real-world problems. Furthermore, every processing element utilizes only a small fraction of the processor's resources, exacerbating the performance bottleneck and limiting its cost effectiveness. By taking advantage of the inherent parallelism in neural networks, a high-performance system capable of applications beyond research can be realized. The common solution is to achieve parallelism by creating a customized pipeline similar to the super-scalar design used by processors. Unfortunately, these systems do not provide sufficient performance and, thus, are primarily restricted to niche markets. Fixed-point arithmetic units can be used to reduce resource utilization and increase processing speed. Finally, RBMs have a high degree of data locality, which minimizes the overhead of transferring data and maximizes the computational throughput. Since the arrangement of processing elements dictates the capabilities and behavior of the network, being able to tailor the hardware to each arrangement is highly desirable. In contrast, application specific integrated circuit implementations must balance the tradeoff between performance and versatility. The primary contributions of paper are as follows:

1) a method to partition RBMs into congruent networks;
2) a collection of modular computational engines capable
of implementing a wide variety of RBM topologies;
3) a method of virtualizing the RBM architecture to implement large networks with limited hardware.

There are weighted connections between every node in opposite layers, and no connections between any nodes in the same layer of an RBM. Biases are

represented by setting the first node. The following notation system will be used: *vi* and *h j* are the binary states of the *i* th and *j*th node, where $i = \{1, \ldots, I\}$ and $j = \{1, \ldots, J\}$, in the visible and hidden layer, respectively, *wi j* is the connection weight between the *i* th and *j* th node. Alternating Gibbs sampling (AGS) and contrastive divergence (CD) learning has been found to be an effective process to determine the node states and update the weight parameters, respectively. AGS is divided into two phases: the generate and reconstruct phases. The generate and reconstruct phases equations below, respectively

$$E = -\sum_{i=1}^{I} v_i \left(\sum_{j=1}^{J} w_{i,j} h_j \right) = -\sum_{i=1}^{I} v_i E_i$$
$$= -\sum_{j=1}^{J} h_j \left(\sum_{i=1}^{I} w_{i,j} v_i \right) = -\sum_{j=1}^{J} h_j E_j.$$

Two pairs of AGS node states are used in CD learning the first pair and an arbitrary odd-numbered AGS phase. The notation CD*X* is used, where *X* is the arbitrary AGS limit. Large limits provide better approximations to gradient descent but require more processing time. In addition, the training data vectors are often grouped into batches, allowing the weights to be updated over the average of the inputted data. Large batch sizes provide smoother learning. The overall time complexity of the RBM algorithm is $O(n^2)$, which illustrates the limited scalability of implementing RBMs on sequential processors. For network sizes of 256×256, 512×512, and 256×1024, the maximum speedup achieved was 25-fold compared to single-precision MATLAB and 30-fold for double-precision MATLAB. In addition to the typical graphic processing considerations, such as coalesced memory accesses and shared memory, performance acceleration was further advanced by introducing a technique called "overlapping patches," which tile small localized RBMs. Each overlapping patch is independent, resulting in globally sparse networks with locally dense connections, greatly reducing the memory size and bandwidth requirements while providing scalability. Instead, a novel divide-and-conquer method is proposed that partitions a large RBM into an equivalent collection of smaller but congruent networks. This technique allows any implementation to create small networks that do not exhaust low-latency memory resources and are better capable of exploiting the data locality of RBMs. The partitioned AGS equations are identical to the equations of a single RBM. This method allows a large RBM to be composed of localized congruent networks for the time penalty incurred by completing the small single global computation. Only the partitioned energies and subsequent node states are transferred, both have an $O(n)$ size. The tradeoff achieved by this partitioning method is advantageous to RBM implementations since the most resource-intensive data is stored locally

which limits the transferring of data, ensuring a low communication to computation ratio. An implementation of the message passing interface (MPI) developed specifically for embedded FPGA designs, called time-multiplexed differential data-transfer message passing interface (TMD-MPI), is used to provide numerous features and benefits. The message passing paradigm is widely used in high-performance computing and TMD-MPI extends this popular protocol to hardware, the hardware RBM implementation is controlled entirely with MPI software code, using messages to abstract the hardware compute engines as computational processes, called ranks. By only transferring energies and node states, the RBMC and NSC execute the required AGS equations for a symmetric RBM network with minimal communication. The partitioning method is used to amalgamate these smaller cores to behave as a single larger RBM with coarse grain parallelism. The NSC determines each partition of node states, and is sent back to both RBMC via the EAC, ensuring consistent values. The final platform is the virtualized single FPGA platform, which illustrates how the partitioning method can be efficiently used to implement large networks with a single FPGA. The hardware cores are time multiplexed, allowing multiple RBMs to be computed with a single set of hardware. The coarse grain parallelism is maintained, as all the FPGAs can switch context independently. Further exploration into virtualized multi-FPGAs systems is left for future work. This micro programmed approach provides an efficient method for flow control and arbitration, which is nontrivial since the compute engines require shared access to the memories. Thus, by following a specific distribution of the matrix and addressing scheme, an entire row or column can be retrieved immediately with low resource utilization. The energy compute engine is responsible for calculating the energies. To complete the vector-matrix operation, it requires one of the layers and the weights. At every clock cycle, the compute engine multiplies the vector layer with one of the columns or rows in the weight matrix to generate a scalar element in the column of the energy matrix. The computation can be done with simple hardware components: AND gates, multiplexers and registered, and fixed-point adders. This binary tree of adders effectively reduces an $O(n^2)$ time complexity to $O(n)$, while only requiring $O(n)$ resources. The energy compute engine is capable of reusing the same hardware for both visible and hidden energies due to the on-demand row and column of the weight matrix. The weight update compute engine has two roles: to store the weight update term for the entire batch as well as to commit and clear the weight update terms.

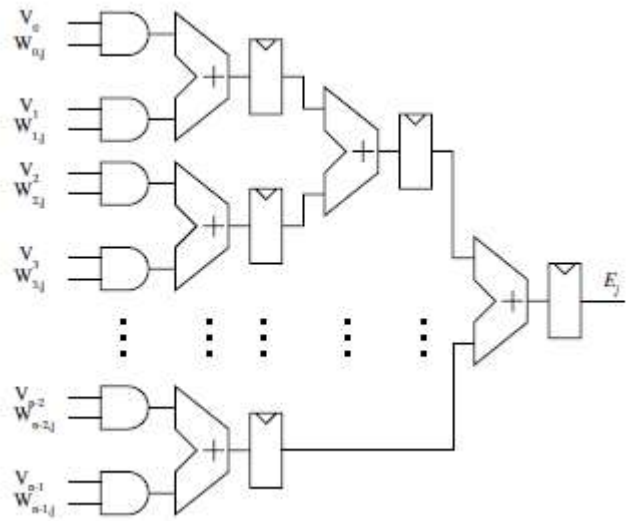


Fig: Circuit diagram of the binary adder tree.

The low-level implementation is straightforward since each element of the weight matrices is independent and, as a result, no circuit diagram is shown. Since memory update is in parallel, the time complexity is reduced from *O(n2)* to *O(n)*, while only requiring *O(n)* resources. The NSC is designed to provide the maximum throughput by converting a single energy to node state every clock cycle. A BRAM LUT implementation is an efficient method to provide a reasonable approximation for bounded transcendental functions. The results are pre-computed and stored in a BRAM, where solutions are obtained in a single read. This is effective for application-specific architectures that use a predefined set of functions. However, a BRAM LUT provides limited resolution. This hardware is called the *k*th stage piecewise linear interpolator (PLI^k), where each successive stage does one iteration of a binary search for the search point for one cycle of latency. Comparing PLI^k with LI, the error is a function of the number of stages and decreases geometrically. Thus, each PLI^k will guarantee an additional bit of precision for every stage. The average and peak error are shown in the equations below and

$$\left|v_{\mathrm{LI}} - v_{\mathrm{PLI}^k}\right|_{\mathrm{average}} = \frac{y_1 - y_0}{2^{k+2}}$$

$$\left|v_{\mathrm{LI}} - v_{\mathrm{PLI}^k}\right|_{\mathrm{peak}} = \frac{y_1 - y_0}{2^{k+1}}.$$

Using the BRAM LUT and PLI^k , a high-precision pipelined sigmoid transfer function was generated. There are two distinct implementations of the EAC as a result of the different platforms. There is a *streaming* implementation designed for multi-FPGA architectures, which takes advantage of the hardware MPI to achieve significant throughput while using limited resources. There is also a *BRAM* implementation designed for the virtualized architectures, which requires additional memory resources to store information to account for the context

switches of the RBMC. Both implementations have a similar MPI communication protocol, providing a modular and reconfigurable architecture. At a lower level, the EAC begins by initiating messages with both the RBMCs and NSCs. Once each of the compute engines is ready to transmit energies and node states, the EAC then streams data bidirectionally through its compute engine using a pipelined data path. The EAC RAM implementation is used for virtualizing the modules in the RBM architecture. The EAC RAM uses a single first-in-first-out (FIFO) data structure to store both the energies and node states. Large memories are not required and a local BRAM provides sufficient resources. This implementation allows a single hardware instantiation to be used for networks of any size. The bandwidth is maintained since the energies are transferred sequentially. After the EAC has received all the messages, it sends the summed energies to the corresponding NSC. As the node states are returned, the EAC forwards them back to the initial rank that originally had all of the energies. The BRAM utilization increases at a much faster rate than both the flipflops and LUTs, resulting in a limiting factor. Since the software implementation has $O(n^2)$ complexity while the hardware implementation is $O(n)$, the speedup is $O(n)$. The quad-FPGA platform provides coarse grain parallelism, achieving a maximum computational throughput of 3.13 GCUPS on a 256 × 256 RBM using four 128 × 128 RBM networks, resulting in a relative speedup of 145-fold over the software implementation. Although the virtualized system is considerably slower than its single-FPGA components, the overall performance is still impressive compared to the software implementation. More importantly, the relative speedup of the virtualized system increases with respect to the network size by increasing the network size, the performance of the software implementation decreases drastically, while the hardware implementation decreases marginally.The results indicated that a single FPGA obtains the best performance while the multi-FPGA platform provides additional coarse-grain parallelism. The virtualized platform lacks the performance of the previous two, but is able to scale to larger networks with fewer resources. The current implementation uses only binary-valued node states, since it resulted in simpler hardware and the majority of node states for RBMs in DBNs are binary-valued because only the bottommost visible layer can be real-valued. Extending the implementation to support binary-valued node states would result in a wider range of applications.

7. FPGA Implementation of a Scalable and Highly Parallel Architecture for Restricted Boltzmann Machines

The following content is taken directly from the paper. Restricted Boltzmann Machines (RBMs) are an effective model for machine learning; however, they require a significant amount of processing time. In this study, we propose a highly parallel, highly flexible architecture that combines small and completely parallel RBMs. This proposal addresses problems associated with calculation speed and exponential increases in circuit scale. We show that this architecture can optionally respond to the trade-offs between these two problems. Restricted Boltzmann Machines (RBMs) are an important component of Deep Belief Networks (DBNs). as RBM scale increases, the amount of required processing also exponentially increases. The processing speed can be increased using specific hardware circuits. To achieve this goal, various RBM architectures have been proposed. However, these architectures do not fully exploit the high parallelism of the neural networks, because each layer is processed sequentially. Higher parallelism requires a substantial circuit scaling, resulting in a trade-off between calculation speed and circuit scale. In this study, we have devised an architecture that divides RBMs into blocks, to prevent exponential circuit scale increases and sequential calculations with respect to the number of blocks. RBMs are a stochastic neural network model consisting of two layers (a visible layer and a hidden layer). RBMs are configured from three parameters, *i.e.,* the connection weights, visible biases, and hidden biases. In this study, we use the binary RBMs which are the most fundamental and common type of RBMs. Hidden and visible nodes are binary. The joint distribution obeys the following model:

$$p(v,h)=\frac{1}{z}e^{-E(v,h)}$$

$$E(v,h)=-\sum_{i=1}^{n}\sum_{j=1}^{m}w_{ij}h_i v_j-\sum_{j=1}^{m}b_j v_j-\sum_{i=1}^{n}c_i h_i$$

$$Z=\sum_{v}\sum_{h}e^{-E(v,h)}$$

Because the connection of each neuron is restricted to the neurons in the complementary layer, the probability of firing of each neuron is given by

$$p(h_i=1|v)=sigmoid\left(\sum_{j=1}^{m}w_{ij}v_j+c_i\right)$$

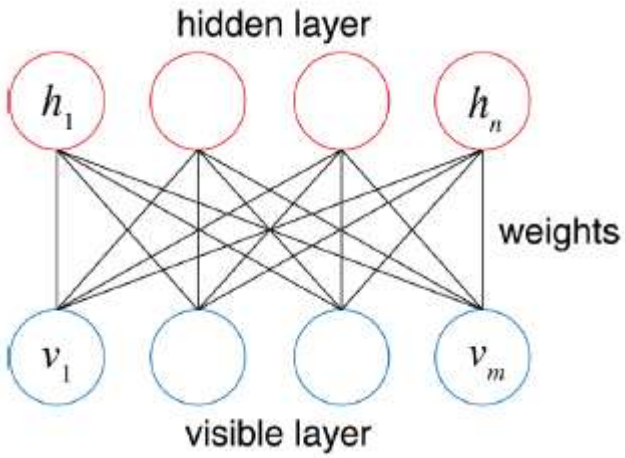


Fig: RBM model; in this case, $n = m = 4$.

$$p\left(v_j = 1 \middle| h\right) = sigmoid\left(\sum_{i=1}^{n} w_{ij} h_j + b_i\right)$$

$$\boldsymbol{sigmoid(x) = \frac{1}{1 - e^x}}$$

The RBM processing flow consists of two repeating steps. First, input data is added to the visible layer, and the hidden layer is calculated using the visible layer's values as input. At this point, all visible layer and hidden layer combinations are calculated. Second, the visible layer is calculated, using sampling results from the hidden layer as input. By repeating these steps, it is possible to approximate the update formula given by

$$\Delta w_{ij} = w_{ij} + \eta \left\{ p\left(h_i = 1 \middle| v^{(0)}\right) v_j^{(0)} - p\left(h_i = 1 \middle| v^{(k)}\right) v_j^{(k)} \right\}$$

$$\Delta b_j = b_j + \eta \left\{ v_j^{(0)} - v_j^{(k)} \right\}$$

$$\Delta c_i = c_i + \eta \left\{ p\left(h_i = 1 \middle| v^{(0)}\right) - p\left(h_i = 1 \middle| v^{(k)}\right) \right\}$$

Where η is learning rate and k is the sampling number. This update formula can be obtained by using contrastive divergence (CD) learning. To construct the DBNs, a hidden layer that has completed learning is used as a visible layer for the next RBMs. Deep networks are constructed by stacking these RBMs. After using back-propagation to perform fine-tuning, the DBNs are completed. First, input data to the RBM unit is obtained from the input buffer, and CD learning computation is repeated within this unit. When a learning process is completed, the learning data of the local memory of the update unit is added to the local memory of the RBM unit.

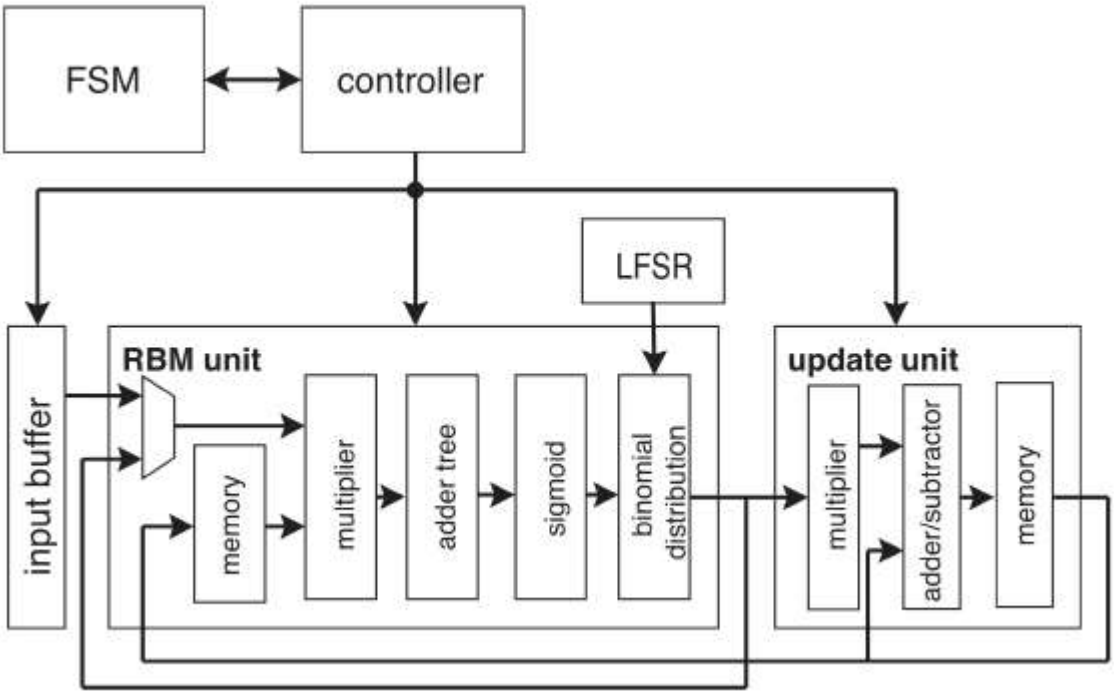


Fig: Overall data-flow.

Input data is assumed to consist of unsigned 8-bit fixed-point numbers representing continuous values from 0 through 1. The bit precision of other parameters is discussed in detail in Section 4. This proposed architecture implements the most basic binary model. Three phases occur, which are physically separated by a shift register. These registers store and propagate the value of each block, and also play the role of the pipeline registers. Phase 1 implements the sum-product operation, phase 2 the sampling operation, and phase 3 the parameter update operation. The operation of each phase is presented in the following.

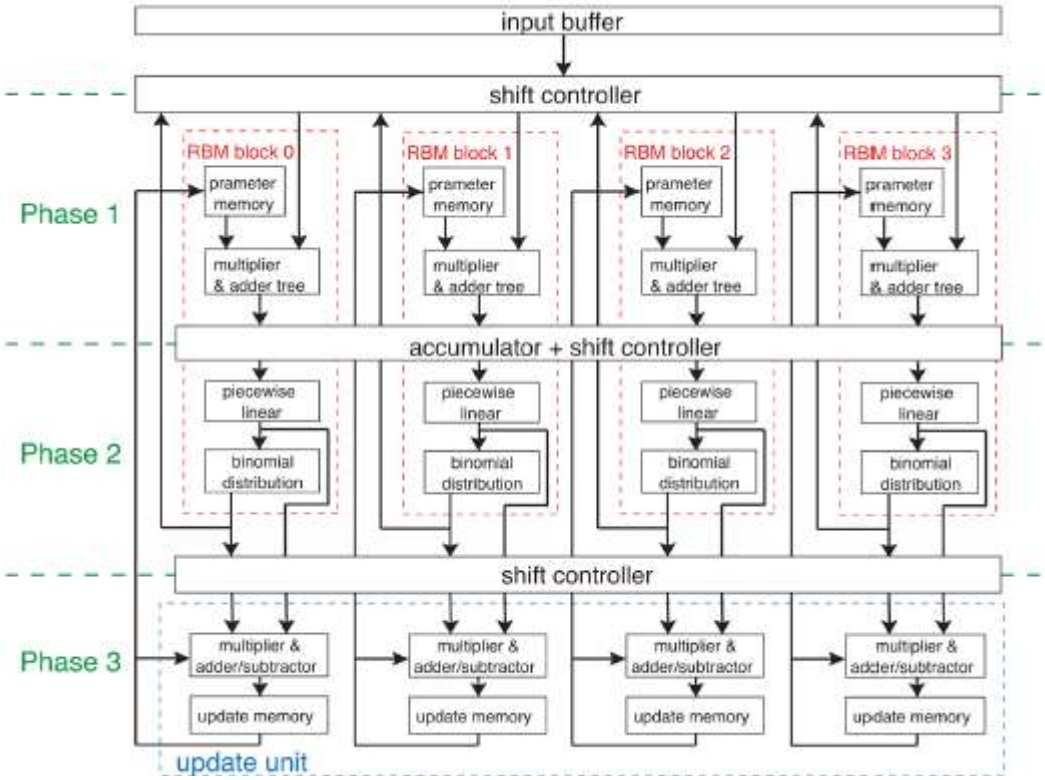


Fig: Detail block-diagram of the proposed architecture.

PHASE 1:

In Phase1, each RBM block multiplies all the inputs and connection weights in parallel, and calculates the respective outputs using an adder tree. During the $\boldsymbol{v} \rightarrow \boldsymbol{h}$ operation, the inputs are shifted between RBM blocks to match the local memory address. During the $\boldsymbol{h} \rightarrow \boldsymbol{v}$ operation, the accumulated outputs are shifted between RBM blocks. In this case, because the arrangement sequence of the connection weights from the local memory is not identical to the $\boldsymbol{v} \rightarrow \boldsymbol{h}$ case, the appropriate connection weights must be re-selected. This approach enables calculating data propagation in both ways using one single circuit, and is possible because each RBM block is configured on a small scale.

PHASE2:

In Phase 2, the sigmoid calculation and binomial distribution calculation are performed. After completion of the sum-product operation, a sigmoid function is applied to the sum value. This result is processed by the sampling operation which yield the output of neurons as a probability of neuronal firing as expressed in Equations. The sigmoid function is approximated as a piecewise linear function to facilitate the implementation on hardware. The piecewise linear function is given by

$$f(x)=\begin{cases} 0 & (x\le -2) \\ \frac{1}{4}x+\frac{1}{2} & (-2<x\le 2) \\ 1 & (2<x) \end{cases}$$

In order to realize this equation, an arithmetic shifter is used instead of a divider since 1/4 equals 2^{-2}. A binomial distribution calculation is also obtained from a subsequent circuit that determines the output as 0 or 1 by comparing the output of the sigmoid function to random numbers generated by an LFSR.

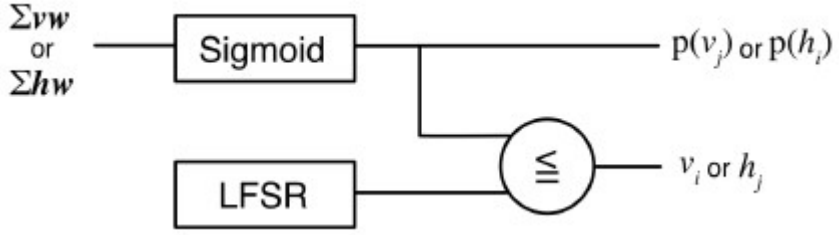


Fig: Circuit schematic of the sampling operation.

PHASE 3:

In addition, because the update unit must calculate all parameters, it resembles the structure of Phase 1. It contains a local memory to store the update data for the RBM unit; these values are constantly updated during the learning process. After all parameters are calculated, the data of the update memory is added to the data of the parameter memory. Multiplication with the learning rate is carried out prior to addition to the parameter memory. The learning rate is quantized to an integer power of 2 (like 1, 1/2, 1/4...), such that multiplication with the learning rate can be approximated to arithmetic shifts.

RESULTS:

The necessary bit precision of parameters which are weights, visible biases and hidden biases has been evaluated using a custom emulator written in a C code which can accurately simulate the proposed architecture in terms of data process and bit precision. While applying fine-tuning improves the accuracy at lower fractional bit-precision. The size ratio of the visible and hidden layers of the RBMs is equal to 1:1. For each developed model, time is linear with respect to the data size. The models partitioned into blocks exhibit relatively small linear increases, whereas the non-partitioned model exponentially increases. Moreover, because of the linear scalability, the speed can be adapted to unit RBM configurations. Twenty learning times and 100 batch size of data were simulated, showing that our implementation provides a 134× speed up factor with respect to the CPU software implementation. From these results, we conclude that the

proposed architecture could reduce circuit scale more effectively than simple parallelism. Furthermore, it could achieve both speed and circuit scaling in a linear manner with respect to the network size in terms of numbers of neurons. Consequently, designers may select the trade-off between speed and circuit scaling which is suitable to their requirements. In addition, the proposed architecture has been implemented on FPGA showing a 134× speed up factor with respect to a conventional CPU, and much faster operation compared to existing FPGA developments.

SOFTWARE USED

XILINX VIVADO

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis. Vivado represents a ground-up rewrite and re-thinking of the entire design flow (compared to ISE), and has been described by reviewers as "well conceived, tightly integrated, blazing fast, scalable, maintainable, and intuitive".

Like the later versions of ISE, Vivado includes the in-built logic simulator ISIM. Vivado also introduces high-level synthesis, with a tool chain that converts C code into programmable logic. Vivado has been described as a "state-of-the-art comprehensive EDA tool with all the latest bells and whistles in terms of data model, integration, algorithms, and performance".

FEATURES:

Vivado enables developers to synthesize their designs, perform timing analysis, examine RTL diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer. Vivado is a design environment for FPGA products from Xilinx, and is tightly-coupled to the architecture of such chips, and cannot be used with FPGA products from other vendors.

Vivado was introduced in April 2012, and is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado includes electronic system level (ESL) design tools for synthesizing and verifying C-based algorithmic IP; standards based packaging of both algorithmic and RTL IP for reuse; standards based IP stitching and systems integration of all types of system building blocks; and the verification of blocks and systems. A free version WebPACK Edition of Vivado provides designers with a limited version of the design environment.

COMPONENTS:

The **Vivado High-Level Synthesis** compiler enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL.Vivado HLS is widely reviewed to increase developer productivity, and is confirmed to support C++ classes, templates, functions and operator overloading. Vivado 2014.1 introduced support

for automatically converting OpenCL kernels to IP for Xilinx devices. OpenCL kernels are programs that execute across various CPU, GPU and FPGA platforms.

The **Vivado Simulator** is a component of the Vivado Design Suite. It is a compiled-language simulator that supports mixed-language, TCL scripts, encrypted IP and enhanced verification.

The **Vivado IP Integrator** allows engineers to quickly integrate and configure IP from the large Xilinx IP library. The Integrator is also tuned for MathWorks Simulink designs built with Xilinx's System Generator and Vivado High-Level Synthesis.

The **Vivado TCL Store** is a scripting system for developing addons to Vivado, and can be used to add to and modify Vivado's capabilities. TCL stands for Tool Command Language, and is the scripting language on which Vivado itself is based. All of Vivado's underlying functions can be invoked and controlled via TCL scripts.

DEVICE SUPPORT:

As of 2018, Xilinx recommends Vivado Design Suite for new designs with Ultrascale, Ultrascale+, Virtex-7, Kintex-7, Artix-7, and Zynq-7000.

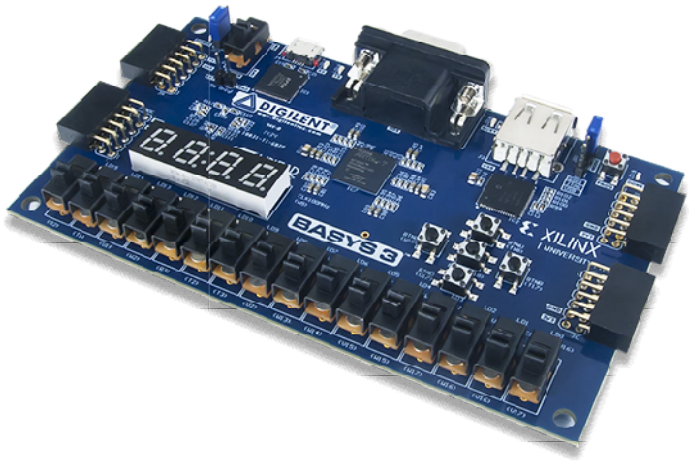
Vivado supports newer high capacity devices, and speeds the design of programmable logic and I/O. Vivado provides faster integration and implementation for programmable systems into devices with 3D stacked silicon interconnect technology, ARM processing systems, analog mixed signal (AMS), and many semiconductor intellectual property (IP) cores.

Vivado is targeted at Xilinx's larger FPGAs, and is slowly replacing Xilinx ISE as their mainline tool chain. As of 2014, Vivado covers Xilinx's mid-scale and large FPGAs, and ISE covered the mid-scale and smaller FPGAs and all CPLDs.

BOARDS USED

BASYS 3 Artix-7:

The Basys3 board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7™ Field Programmable Gate Array (FPGA) from Xilinx. With its high-capacity FPGA (Xilinx part number XC7A35T-1CPG236C), low overall cost, and collection of USB, VGA, and other ports, the Basys3 can host designs ranging from introductory combinational circuits to complex sequential circuits like embedded processors and controllers. It includes enough switches, LEDs and other I/O devices to allow a large number designs to be completed without the need for any additional hardware, and enough uncommitted FPGA I/O pins to allow designs to be expanded using Digilent Pmods or other custom boards and circuits.



- **More I/O:**

Double the user interface switches, double the number of onboard outputs, upgraded the number of external ports (moving from 6-pin single-row Pmods to 12-pin double-row Pmods) and included for the first time on a Basys class device a USB-UART bridge.

- **Modern Architecture and Modern Programming Challenges:**

Due to the migration from the Spartan-3E family to the Artix-7 class of device, the Basys 3 offers a substantial increase in hardware capabilities. With the new Artix FPGA comes 15X the logic cells (from 2,160 to 33,280) and the upgrade from multipliers to true DSP slices. It also adds over 26X the amount of RAM.

- **Industry's First SoC Strength Design Suite:**

The most significant change to the Basys 3 is the upgrade to Xilinx Vivado Design Suite (WebPACK edition available for free download from Xilinx), the most modern design tool chain used by professional engineers worldwide. Compared to ISE®, Vivado offers an improved user experience and expanded capabilities. These capabilities include block-based IP integration (which can reduce development time up to 10x) and the Vivado Logic/Serial I/O analyze. **Device/IC:** Xilinx Artix-7 FPGA (XC7A35T-1CPG236C)

Connector(s):

- USB A
- USB micro-B
- Four 12-pin Pmod ports
- VGA

Programming: Designed exclusively for the Vivado Design Suite

Features:

- Features the Xilinx Artix-7 FPGA: XC7A35T-1CPG236C
- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Five clock management tiles, each with a phase-locked loop (PLL)
- 90 DSP slices
- Internal clock speeds exceeding 450 MHz
- On-chip analog-to-digital converter (XADC)
- Digilent USB-JTAG port for FPGA programming and communication
- Designed Exclusively for Vivado Design Suite (Vivado Design Suite WebPACK edition can be downloaded for free from Xilinx). Expanded features are available through purchase of the Design Edition.
- Free WebPACK™ download for standard use.
- Micro-B USB cable not included.
- Serial Flash
- USB-UART Bridge
- 12-bit VGA output
- USB HID Host for mice, keyboards and memory sticks
- 16 user switches
- 16 user LEDs
- 5 user pushbuttons
- 4-digit 7-segment display

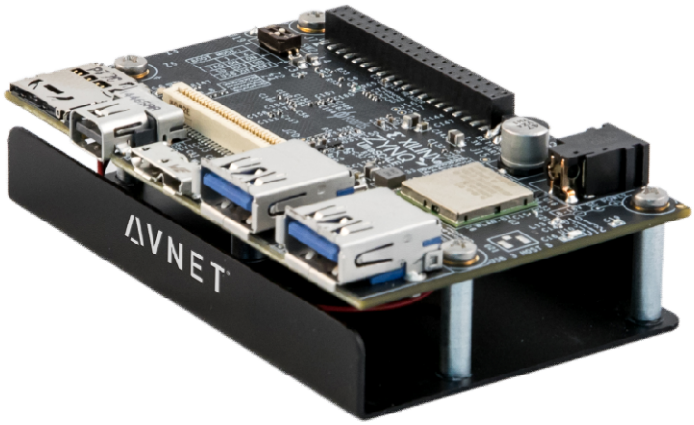
- 4 Pmod ports: 3 Standard 12-pin Pmod ports, 1 dual purpose XADC signal / standard Pmod port

ULTRA96:

Ultra96™ is an ARM-based, Xilinx Zynq UltraScale+™ MPSoC development board based on the Linaro 96Boards specification. The 96Boards' specifications are open and define a standard board layout for development platforms that can be used by software application, hardware device, kernel, and other system software developers. Ultra96 represents a unique position in the 96Boards community with a wide range of potential peripherals and acceleration engines in the programmable logic that is not available from other offerings.

Key Features and Benefits:

- Linaro 96Boards Consumer Edition compatible
- 85mm x 54mm form factor
- 60-pin 96Boards High speed expansion header
- 40-pin 96Boards Low-speed expansion header
- 2x USB 3.0, 1x USB 2.0 Type A downstream ports
- 1x USB 3.0 Type Micro-B upstream port
- Mini DisplayPort (MiniDP or mDP)
- Wi-Fi / Bluetooth
- Delkin 16 GB MicroSD card + adapter
- Micron 2 GB (512M x32) LPDDR4 Memory
- Xilinx Zynq UltraScale+ MPSoC ZU3EG A484



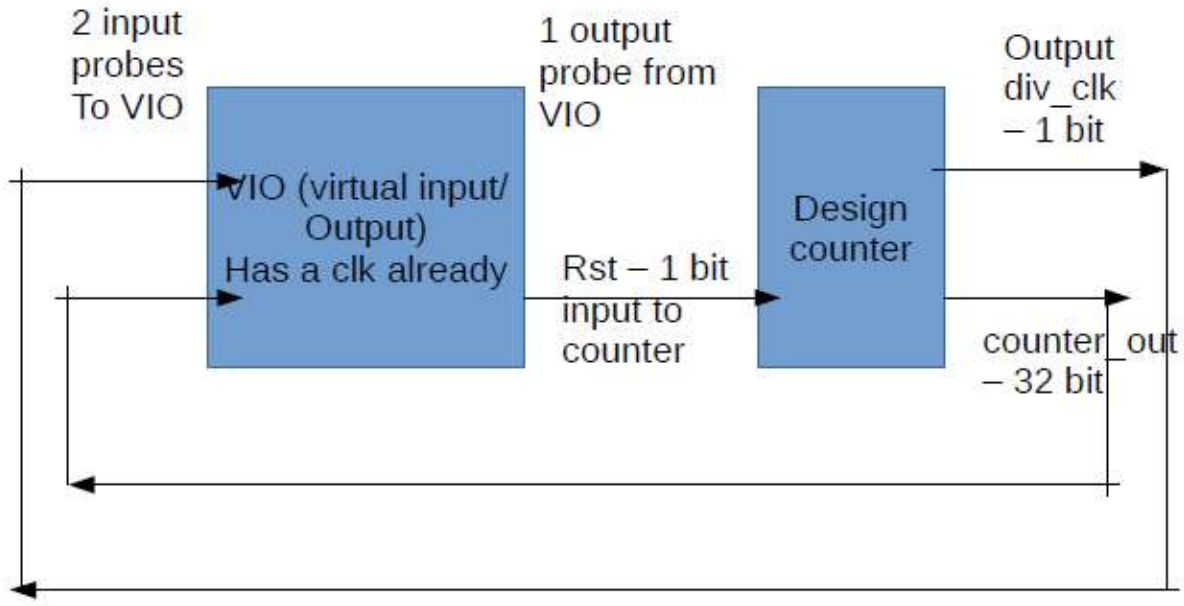
What's Included:

- 16 GB pre-loaded microSD card + adapter
- Quick-start instruction card
- Ultra96 development board
- Voucher for SDSoC license from Xilinx

IMPLEMENTATIONS

32 BIT COUNTER USING VIRTUAL INPUT OUTPUT (VIO) MODULE:

BLOCK DIAGRAM:



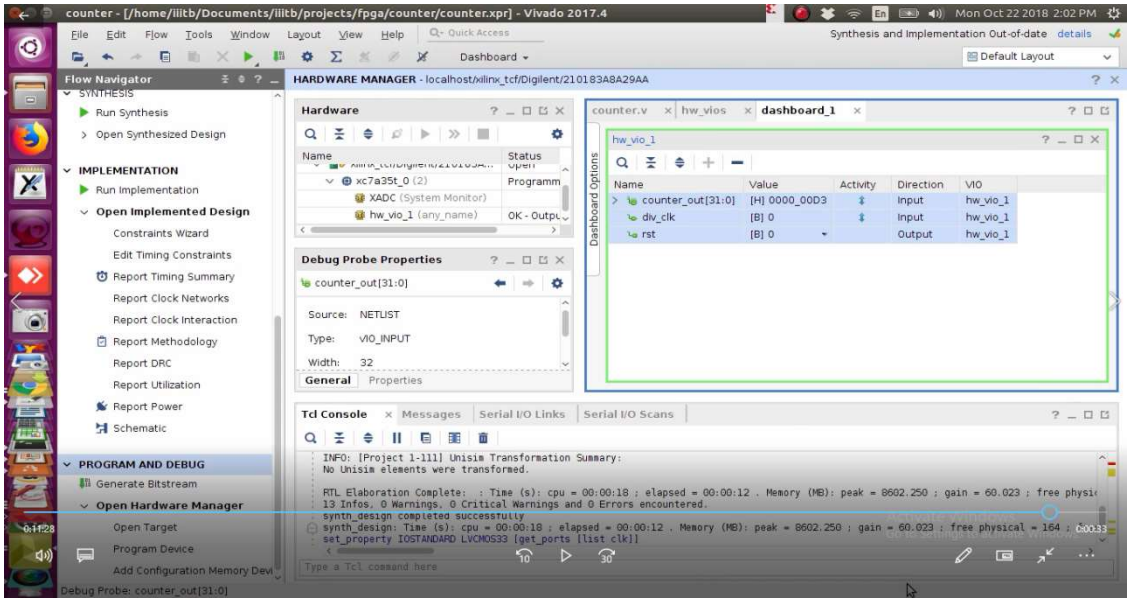
VERILOG CODE:

```
module clock_div(clk);

//We will add a VIO here. So, convert all outputs to reg and all

//inputs to wires except clk

//VIO --- Design. VIO outputs will be connected to our counter input

//Counter outputs will be VIO inputs in a feedback manner

input clk;

wire rst; //input converted to wire

reg div_clk; //this is also an output of counter

reg [25:0] delay_count;
```

```
reg [31:0] counter_out; //make this 32 bit output --> reg

//paste the VIO template here

vio_1 any_name (

  .clk(clk),                // input wire clk

  .probe_in0(div-clk),    // input wire [0 : 0] probe_in0 - 1bit probe

  .probe_in1(counter_out),    // input wire [31 : 0] probe_in1

  .probe_out0(rst)  // output wire [0 : 0] probe_out0

);

// INS

//////////clock division block//////////////////

always @(posedge clk or posedge rst)

begin

if(rst)

begin

delay_count<=26'd0;

div_clk <= 1'b0;

end

else

if(delay_count==26'd33554432)

begin

delay_count<=26'd0;

div_clk <= ~div_clk;

end

else

begin
```

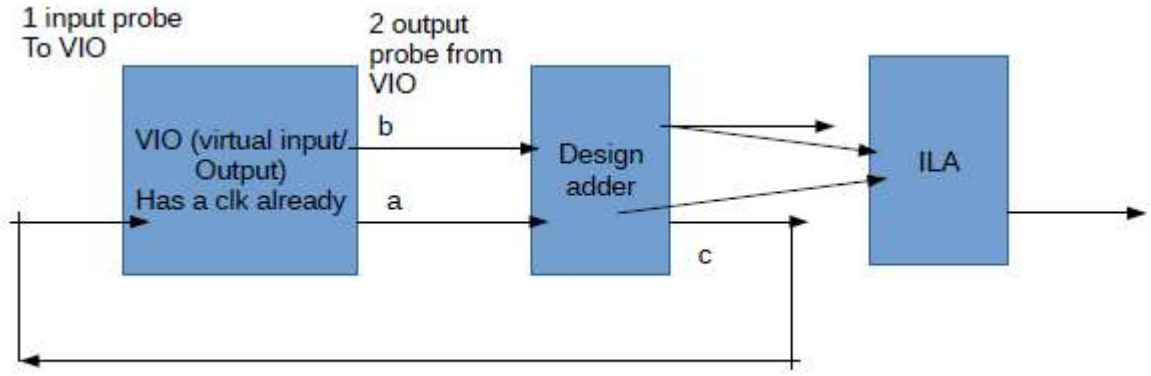
```
delay_count<=delay_count+1;

end

end

//////////////4 bit counter block//////////////////

always @(posedge div_clk or posedge rst)

begin

if(rst)

begin

counter_out<=4'd0;

//div_clk <= 1'b0;

end

else

begin

counter_out<= counter_out+1;

end

end

endmodule
```

RESULTS:



32 BIT COUNTER USING VIRTUAL INPUT OUTPUT (VIO) AND INTEGRATED LOGIC ANALYSER (ILA):

BLOCK DIAGRAM:

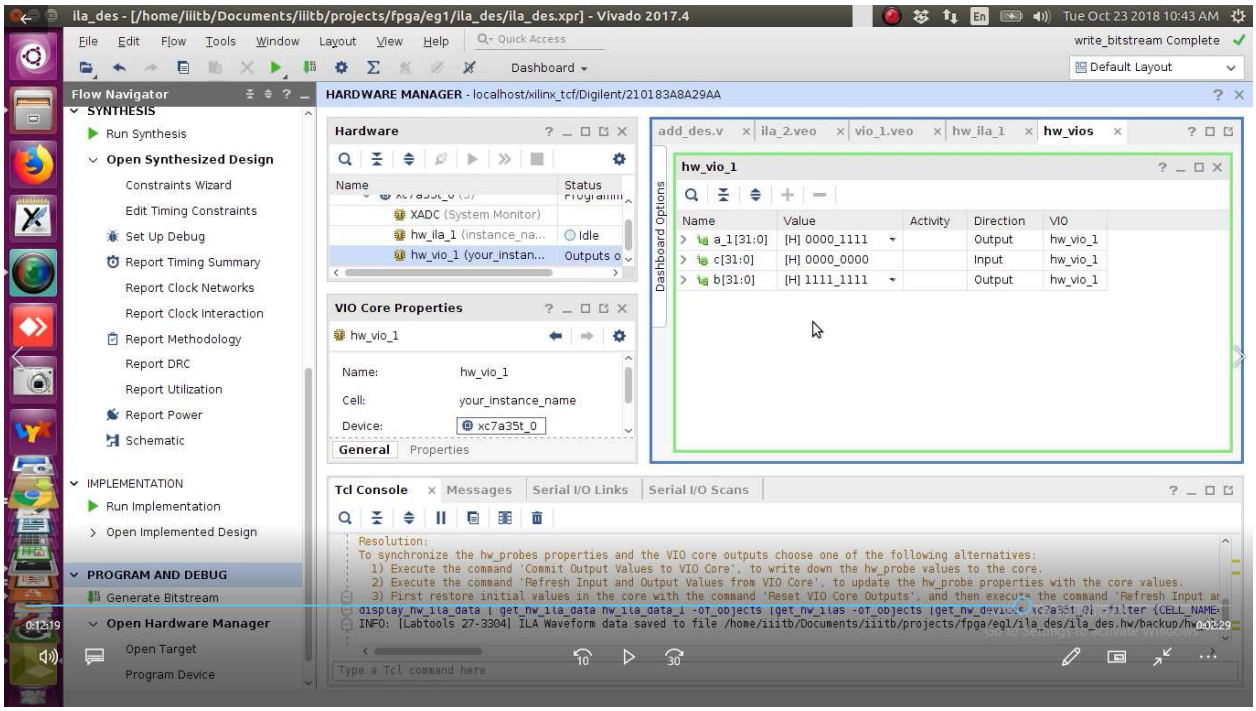


VERILOG CODE:

```
module up_counter(clk);
wire reset;
input clk;
wire [3:0] counter;
reg [3:0] counter_up;
```

```
//VIO
vio_0 your_instance_name (
  .clk(clk),                // input wire clk
  .probe_in0(counter),    // input wire [3 : 0] probe_in0
  .probe_out0(reset)  // output wire [0 : 0] probe_out0
  );
ila_0 your_instance (
            .clk(clk), // input wire clk
            .probe0(reset), // input wire [0:0]  probe0
            .probe1(counter) // input wire [3:0]  probe1
);
// up counter
always @(posedge clk or posedge reset)
begin
if(reset)
 counter_up <= 4'd0;
else
 counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

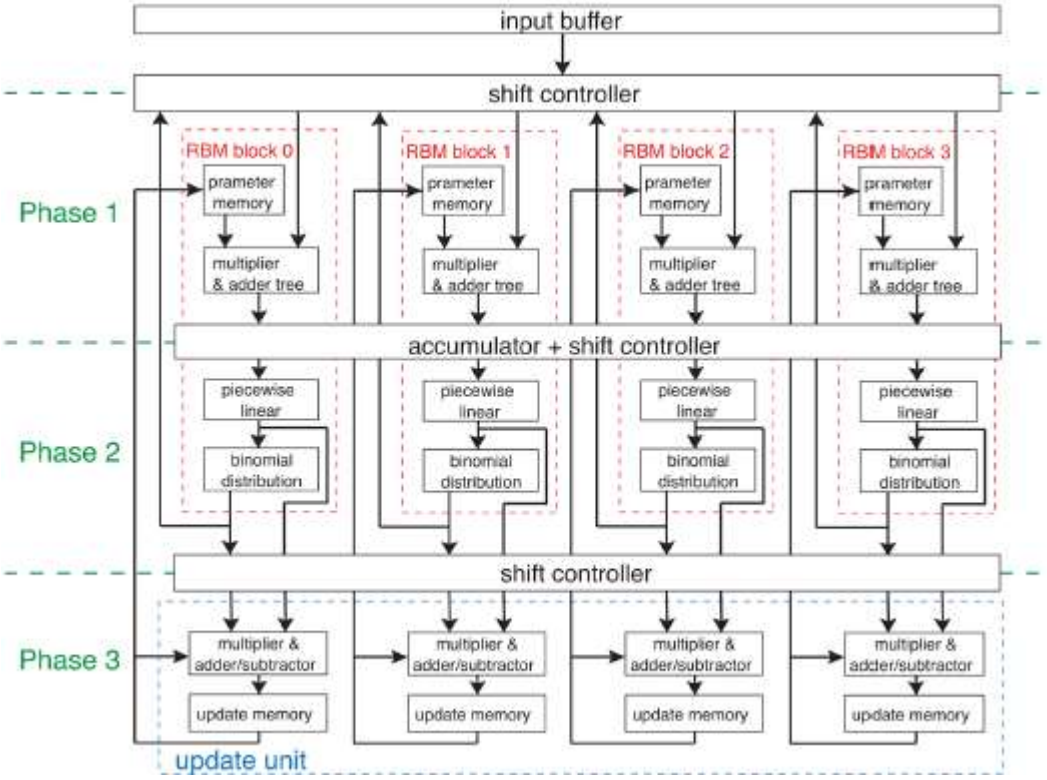
RESULTS:



RESTRICTED BOLTZMANN MACHINE WITH FOUR NODES IN A LAYER

Here we are using the architecture used in paper 7 discussed above which provides scalability and high speed up and implemented binary RBM which is divided into 4 blocks using the above architecture and each layer having 4 nodes.

BLOCK DIAGRAM:



VERILOG CODE:

```
module rbm_0(
input a,
input [3:0]wi,
input clk,
input rst,
output reg [3:0]b
);
reg [3:0]w1;
reg [3:0]w2;
reg [3:0]w3;
reg [3:0]w4;
reg[3:0]w5;
```

```
reg [3:0]w6;
reg [3:0]w7;
reg [3:0]w8;
reg [3:0]w9;
reg [3:0]p;
reg [3:0]q;
reg [3:0]r;
always@(posedge clk)
begin
if(rst)
begin
 q<=4'b0000;
 p<=4'b0000;
 w9<=wi;
 end
 else
 begin
 w9<=4'b0000;
 end
 r<=w9;
 w1<=r;
 w2<=(w1*a);
 w3=w2+q;
 q=w3;

 if(w3<=-2)
 begin
 w4<=0;
 w5<=0;
 end

 if(w3>4'b0010)
 begin
 w4<=1;
 w5<=1;
 end
 else
 begin
 w4=w3>>>2;
 if(w4>0)
 begin
 w5<=1;
 end
 else
 w5<=0;
 end
 w6=(w4*w5);
 w7=w6+p;
 p=w7;
w8=p;
b<=w8;
 end
endmodule

module arbm(
```

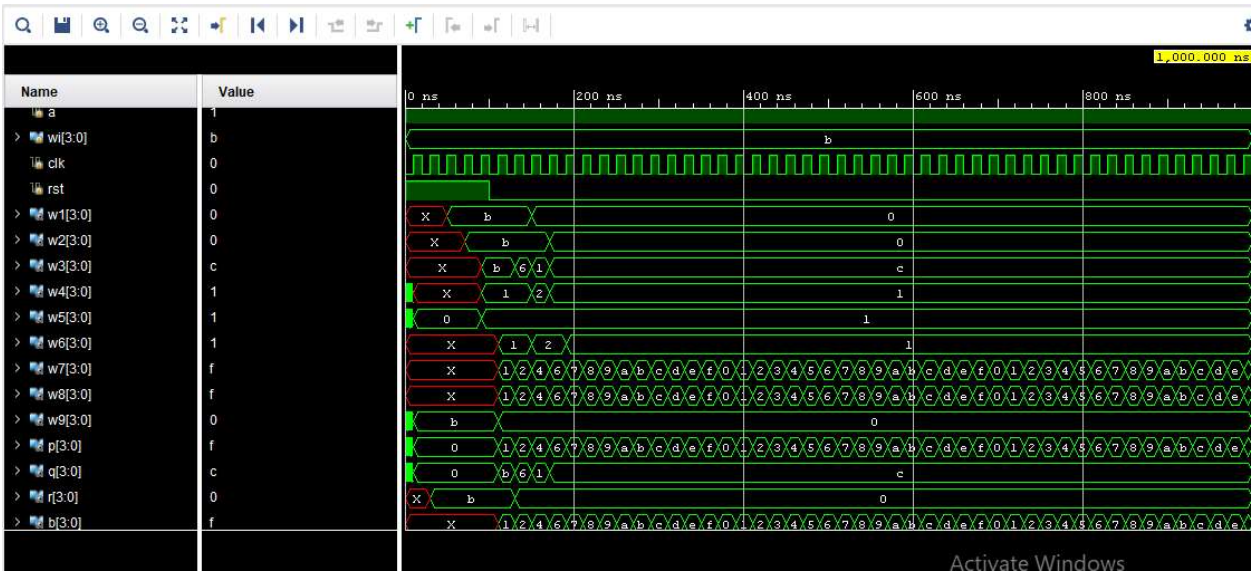
```
input a,
input [3:0]wi,
input clk,
input rst,
output reg [3:0]b
    );
    reg [3:0]w1;
    reg [3:0]w2;
    reg [3:0]w9;
    reg [3:0]r;
    reg [3:0]q;
    always@(posedge clk)
    begin
    if(rst)
    begin
     q<=4'b0000;
     w9<=wi;
     end
     else
     begin
     w9<=4'b0000;
     end
     r<=w9;
     w1<=r;
     w2<=(w1*a);
     b=w2+q;
     q=b;
     end
endmodule

module brbm(
input [3:0]a,
input clk,
output reg [3:0]w4,
output reg [3:0]w5
    );
    always@(posedge clk)
    begin
     if(a>4'b0010)
    begin
    w4<=1;
    w5<=1;
    end
    else
    begin
    w4=a>>>2;
    if(w4>0)
    begin
    w5<=1;
    end
    else
    begin
    w5<=0;
    end
    end
```

```
    end
endmodule

module crbm(
input [3:0]w4,
input [3:0]w5,
input clk,
input rst,
output reg [3:0]w8
);
reg [3:0]w6;
reg [3:0]w7;
reg [3:0]p;
  always@(posedge clk)
  begin
  if(rst)
  begin
  p<=4'b0000;
  end
  w6=(w4*w5);
  w7=w6+p;
  p=w7;
  w8=p;
   end
endmodule

module rbm(
input [3:0]a,
input clk,
input rst,
input [3:0]wi0,
input [3:0]wi1,
input [3:0]wi2,
input [3:0]wi3,
output reg [3:0]out0,
output reg [3:0]out1,
output reg [3:0]out2,
output reg [3:0]out3
);
wire [3:0]x0;
wire [3:0]x1;
wire [3:0]x2;
wire [3:0]x3;
wire [3:0]y0;
wire [3:0]y1;
wire [3:0]y2;
wire [3:0]y3;
wire [3:0]z0;
wire [3:0]z1;
wire [3:0]z2;
wire [3:0]z3;
wire [3:0]k0;
wire [3:0]k1;
wire [3:0]k2;
wire [3:0]k3;
```

```
wire [3:0]m0;
wire [3:0]m1;
wire [3:0]m2;
wire [3:0]m3;
wire [3:0]n0;
wire [3:0]n1;
wire [3:0]n2;
wire [3:0]n3;
arbm a1(a[0],wi0,clk,rst,x0);
arbm a2(a[1],wi1,clk,rst,x1);
arbm a3(a[2],wi2,clk,rst,x2);
arbm a4(a[3],wi3,clk,rst,x3);
pipo p1(x0,x1,x2,x3,clk,y0,y1,y2,y3);
brbm b1(y0,clk,z0,k0);
brbm b2(y1,clk,z1,k1);
brbm b3(y2,clk,z2,k2);
brbm b4(y3,clk,z3,k3);
pipo p2(k0,k1,k2,k3,clk,m0,m1,m2,m3);
crbm c1(z0,m0,clk,rst,n0);
crbm c2(z1,m1,clk,rst,n1);
crbm c3(z2,m2,clk,rst,n2);
crbm c4(z3,m3,clk,rst,n3);
always@(*)
begin
out0=n0;
out1=n1;
out2=n2;
out3=n3;
end
endmodule
```

RESULTS:

BASIC RBM BLOCK:



RBM divides into 4 blocks with 4 nodes in a layer, nodes and weights getting updated:

CONCLUSION

The architecture used here could achieve both speed and circuit scaling in a linear manner with respect to the network size in terms of numbers of neurons. Consequently, designers may select the trade-off between speed and circuit scaling which is suitable to their requirements. And this architecture can be modified according to requirements and applications which is been discussed in the papers above.