High Performance Training of Deep Neural Networks Using Pipelined Hardware Acceleration and Distributed Memory

Raghav Mehta[1], Yuyang Huang[2], Mingxi Cheng[3], Shrey Bagga[4], Nishant Mathur[4],
Ji Li[4], Jeffrey Draper[4], and Shahin Nazarian[4]

[1]Mentor, A Siemens Business, Wilsonville, OR, USA
[2]Nvidia, Shanghai, China
[3]Duke University, Durham, NC, USA
[4]University of Southern California, Los Angeles, CA, USA

***Abstract*—Recently, Deep Neural Networks (DNNs) have made unprecedented progress in various tasks. However, there is a timely need to accelerate the training process in DNNs specifically for real-time applications that demand high performance, energy efficiency and compactness. Numerous algorithms have been proposed to improve the accuracy, however the network training process is computationally slow. In this paper, we present a scalable pipelined hardware architecture with distributed memories for a digital neuron to implement deep neural networks. We also explore various functions and algorithms as well as different memory topologies, to optimize the performance of our training architecture. The power, area, and delay of our proposed model are evaluated with respect to software implementation. Experimental results on the MNIST dataset demonstrate that compared with the software training, our proposed hardware-based approach for training process achieves 33X runtime reduction, 5X power reduction, and nearly 168X energy reduction.**

***Keywords*—Deep learning, neural network, hardware design.**

I. Introduction

Deep Neural Networks (DNN) have recently advanced to encompass various areas of problem solving requiring analysis of large sets of unclassified data [1], [2]. The current trends in DNN implementation warrants use of a large number of hidden layers to achieve high accuracy levels that are required in many applications like image recognition (ILSVRC) [3], object detection (COCO), audio detection [4] and natural language processing [5]. For example, ResNet can have anywhere between 34 to 152 hidden layers depending on application [6]. Higher number of hidden layers allows the DNN to extract large number of features from a complex training dataset resulting in high accuracy predictions and categorization.

The hidden layers in a DNN comprise of nodes for which activation values are computed based on partial product of previous node values, node interconnection edges and activation function. Nonlinear activation functions, such as rectified linear unit (ReLU), logistic or sigmoid and hyperbolic tangent, are commonly used in a DNN [7], [8]. Weights of the interconnecting edges are assigned during the training phase, which is then validated and tested. Thus, training a DNN requires forward propagation of activation values based on previous node values to arrive at output for each input vector. The error - difference between expected output and generated output for each vector - is then back propagated to adjust edge weights of individual hidden layers [9]. Calculation of activation value for each node is a dense matrix multiplication with a series of multiply-accumulate operations. A large number of such nodes amongst an ever growing number of hidden layers makes the task of training a neural network highly computationally intensive and time consuming [10]. The situation worsens for DNN applications requiring frequent retraining due to changing dataset or efficiency requirements. This has pushed the capabilities of conventional hardware like CPUs to their limit leading to unrealistic amounts of training time, necessitating the use of specialized hardware. GPUs, are thus, increasingly being used for such applications, as they are able to effectively exploit very high degree of parallelism offered by DNNs. Moreover, GPUs offer high performance through a large number of floating point units along with high bandwidth on-chip and off-chip memories designed specifically for DNNs. However, these advancements have led to high implementation and power consumption costs as well.

FPGAs can offer low cost and power implementation capabilities. Their reprogrammable nature also makes them suitable for DNN applications requiring frequent retraining. However, they are yet to catch up with GPUs in performance terms due to which their applications for commercial DNN training and implementation has been quite limited. Off late, FPGAs have tried to bridge this gap through rapid advancements in technology, by offering a very high number of floating point units with high-speed-high-bandwidth memories [11]. However, the re-programmable nature of FPGAs warrants the need for additional circuitry inherently limiting their capabilities. Thus, use of FPGAs for training of DNNs is ideal for applications requiring small number of hidden layers and nodes.

ASICs on the other hand have the capability to offer much higher performance as they typically comprise of 20-25X more

*Contact person for this paper is Dr. Shahin Nazarian (shahin.nazarian@usc.edu) logic cells as compared to a similar FPGA and by extension a GPU [11], [12]. ASICs are also highly power and area efficient due to their application specific nature [13–15]. However, their use has been limited due to high initial costs of development and their relative inflexibility to be reconfigured [16], [17].

We therefore, propose an ASIC implementation of a hardware accelerator to expedite the training process of complex neural nets. This can allow widespread adoption of cognitive systems and deep learning on smaller platforms. ASICs can also be configured to have a customized distributed memory to allow high-speed-high-bandwidth memory access eliminating the threat of memory bottleneck.

Without any loss of generality, we present the ASIC designs of basic operations for the backpropagation training algorithm, which is the most popular training algorithm proposed by Rumelhart for DNN [9]. Then the entire accelerator is implemented with the gradient descent for training a general multilayer feed forward neural network. A customized distributed memory architecture is also proposed for this accelerator to assist in fast training of neuron by providing a high-speed-high-bandwidth memory structure. We have confirmed the high efficiency of our accelerator architecture for training purposes, by evaluating its hardware costs as well in addition to its performance on a small customized network as well as a classic medium-sized DNN. We also analyze the memory bandwidth requirements of such a hardware and analyze the performance of our proposed memory architecture in meeting those requirements.

The contribution of this work is twofold. First, we design three essential blocks in HDL for basic operations in DNN training with high accuracy, including forward propagation, backpropagation, and gradient descent. These blocks enable fast on-the-fly incremental learning of each neuron. Second, the proposed hardware accelerator for fast training is evaluated on a customized network for distinguishing a particular color in an RGB color configuration, and also on a DNN for the MNIST dataset for fast analysis [18]. The proposed design significantly speeds up the training process with reduced power and energy compared with software training on a general processor. Memory bandwidth limitations have been overcome through a layered memory architecture that employs distributed memory architecture for training DNN. Additionally, the proposed memory organization is both fast and scalable.

Experimental results on the MNIST dataset demonstrate that our proposed hardware-based approach achieves 33X runtime reduction, 5X power reduction, and 168X energy reduction, compared with software training.

II. Related Works

Recently, DNNs have pushed Artificial Intelligence (AI) limits in a wide range of tasks [19]. Today's DNNs are almost exclusively trained on many very fast and power-hungry Graphic Processing Units (GPUs) [20], but the software training is not efficient for a large sized network because of the exponential increase in the training time.

With the powerful parallelism capability of DNNs, the ASIC approach offers a viable solution to this computationally intensive process. Hence, the recent focus has been the development of hardware-based DNNs to build efficient adaptive systems. There are lots of ongoing research on hardware friendly classifiers and optimization of machine learning algorithms. Some of the most noticeable recent works are DaDianNao [16], Eyeriss [21], and EIE [22]. The DianNao accelerator is used for the fast and low-energy execution of the inference of large CNNs and DNNs in a small form factor [16]. Eyeriss provides several key improvements for inference in convolutional neural networks [21]. The EIE, an energy-efficient engine is optimized to operate on compressed deep neural networks and accelerated the resulting sparse matrix-vector multiplication with weight sharing. EIE is shown to significantly reduce the energy compared to GPUs [22]. Authors in [23] attempt to implement an independent component analysis algorithm that tracks changes in inputs for creation, training and deployment of machine learning models on hardware (FPGA) which offers considerably higher throughput.

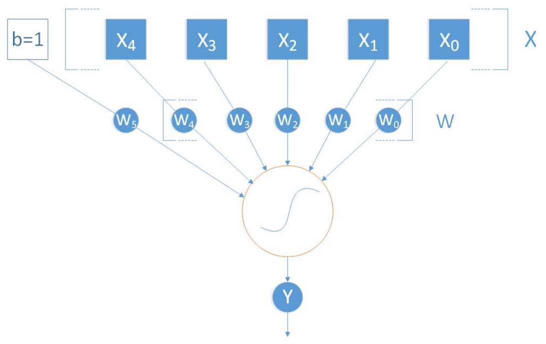

Fig. 1. A single layer neural network with five inputs, one bias unit (b = 1) and a single output.

However, none of the aforementioned works have focused on accelerating the training process of DNN using the ASIC approach.

III. Proposed Acceleration for Fast Training in Neural Networks

A. Background of Neural Network Training

We adopt a typical artificial neural network which consists of interconnected computing units which display features of biological network to accomplish a pattern recognition task [24]. Each unit has N inputs, weights for each input and computes a weighted sum to produce an output. These output values and external inputs determine the activation and output of the next cell in the network. Figure 1 shows a single layer network with five inputs and one bias which results in one output.

Forward Propagation for Activation: We consider a neural network with L Layers with x_0 as the input vector, x_L and h denotes the actual and predicted vector, respectively [25]. We use sigmoid activation function, which takes the weighted inputs and restricts them in continuous range of values between 0 and 1. From 0 representing no firing to 1 being fully saturated firing [26].

Backpropagation Training: The backpropagation algorithm makes use of the concept of gradient descent to calculate minimum of error function in weights and optimize the weight

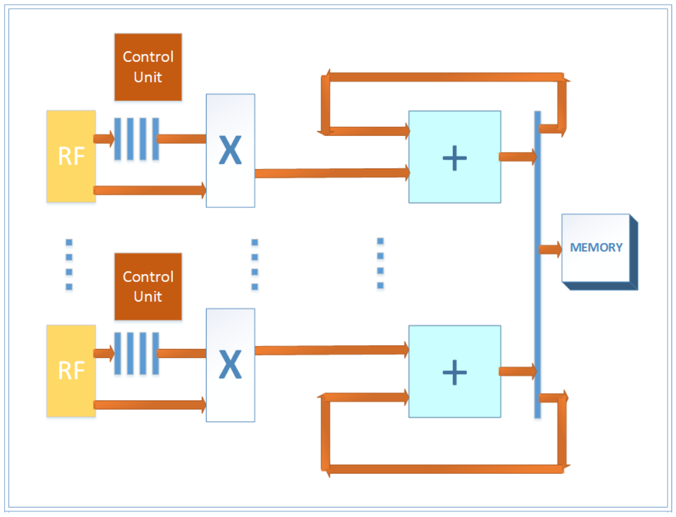

Fig. 2. ALU replicated for a neural network layer

of the network iteratively [27]. The largest loss function weight contribution is then determined and gradient descent is used to change the direction towards minimum loss path. Finally, the activation functions are updated as part of forward trace again and the process is continued for each batch to achieve the actual output with minimum error.

B. Proposed Neuron Accelerator Design

We first implement a fully digital HDL hardware of a scalable neural network accelerator and explore complication that the hardware implementation acknowledges compared to software in the training process and time. In the Section IV.A we implement, optimize and examine the productivity of a single neuron. In Section IV.B we explain our proposed memory architecture customized to offer maximum memory bandwidth to forward and backward units during training phase. In Section IV.C we explain how to connect neurons to a DNN using an example DNN that distinguishes a color component in an RGB pixel.

To analyze the efficacy of our approach, a simple neural network was chosen that contains an input layer, two hidden layers and an output layer. The input and hidden layers consists of three neurons each and the output layer consists of a single neuron.

The digital neuron design is composed of three blocks, the first being forward propagation, followed by backward propagation and finally gradient descent. Each of these blocks are considered as separate modules in HDL and integrated to perform functionality for a neuron.

The floating point arithmetic unit has been pipelined to ensure fixed latency outputs. We have used 32-bit IEEE 754 floating point standard in forward propagation and backpropagation algorithms.

Figure 2 shows architecture design of the ALU considered for neurons in one layer.

1) Forward Unit: We first implemented a fully digital neuron with ability to compute activation. For this activation function, we perform a sigmoid operation for linear regression. For hardware implementation a near approximation of the

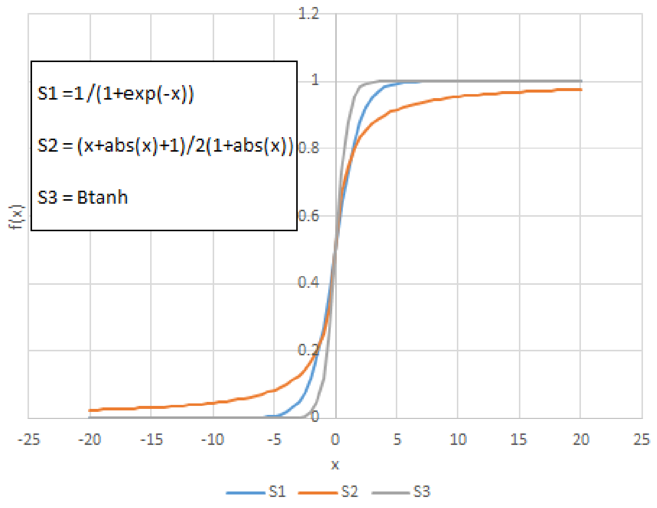

Fig. 3. Sigmoid functions and alternative activation functions

sigmoid function is considered and scaled accordingly. Note that the design can be applied to other activation functions as well.

From the several options for sigmoid we chose the above since it was implementable and works well when synthesized. Figure 3 shows a different sigmoid that we considered for our design.

We build a tailored design and instruction set to operate in pipeline mode with dedicated 32 bit IEEE 754 single-precision floating-point adders and multipliers for forward, backpropagation and gradient calculations. Each node components include a control unit, interfaces and ALU units for calculation of activation. The ALU is pipelined to produce 1 result per clock.

2) Backpropagation Unit: This unit is the most complex module of the hardware structure that implements the backpropagation algorithm for training process and produces results for the next module. It provides the ability to back propagate and calculate δ error on scalable inputs and compute derivative of cost function. In order to save area, the forward unit makes use of the same ALU along with some additional control signals. This module contains separate sub-modules that compute different functions in pipelined stages. Using actual output, the first submodule computes the δ for the last layer such that δ is the difference between activation of its own compute in forward unit and the actual output. This process is repeated for each neuron in the network. The next submodule calculates Δ^l i.e is the sum of product of activation and δ.

$$\Delta^l_{ij} = \Delta^l_{ij} + a^l_j \cdot \delta^{(l+1)}_i \tag{1}$$

where i denotes iteration for node j in layer l. The last submodule computes the partial derivative of the cost function which is very much dependent on the sum of product Δ^l and the number of training iterations m for the training process.

3) Gradient Descent Unit: This module also uses the ALU architecture and updates the weights of the network using cost function and learning rate α. As expected, with large α equal to 0.1, the training process is faster but the swing between 0 and 1 is very small. While with smaller value of α equal to 0.01 the descent is slower but the accuracy and swing are higher.

C. *Distributed Memory System*

A distributed memory architecture based on frequency of data access and bandwidth requirements is proposed. A memory that utilizes a layered structure to provide a small and fast scratch memory with multiple access ports is envisaged.

We, therefore, propose a Synchronous Dynamic Random Access Memory (SDRAM) to store large number of training vectors and a Static Random Access Memory (SRAM) for storage of weights and intermediate results [28] [29].

The SRAM is sized according to the cumulative memory requirement based on training weights and calculation results including loss function for each layer. This provides a low latency memory access during training phase. The main SDRAM is sized as a shared memory amongst all DNN layers to store all training vectors for data processing during testing and validation stages. This memory structure increases the overall available memory bandwidth during training phase providing significant advantages as compared to a generic CPU.

We have two kinds of memory accesses: (i) DNN to SRAM (ii) SRAM to SDRAM

Access time is optimized in the proposed memory architecture by parallelizing the two types of memory accesses. For instance, lets consider a base case where 10,000 (say) training vectors are in SDRAM and 256 training vectors stored in our SRAM. At the end of completion of 64 vectors, as we begin to process the 65th vector, our multi ported SRAM would swap out the previously used 64 training vectors with new vectors from SDRAM cache. With a conservative estimate that SRAM is twice as fast as SDRAM, by the time 192 vectors would have been processed, 64 new vectors would have been swapped in. This overlapping of memory accesses can lead to significant reductions in memory bandwidth requirements.

Moreover, with a memory requirement of less than 1kB per neuron, the proposed memory model is highly scalable. This is especially suited for DNN applications where a couple of hundred to a few thousand neurons are anticipated for any real world application.

D. *Training the Proposed Accelerator*

Here, we explain how to connect each neurons to a DNN using a small network example. We consider a network shown in Figure 4 having two hidden layers with each neurons implemented as logic circuits. With each neuron having the ability to forward propagate and back propagate we connect the dots through the weights. The complete system is coded in hardware descriptive language and evaluated based on ASIC synthesis. Based on the operation defined by the control unit all the nodes in each layer process their functions in parallel. All the modules are connected using the interfaces and communicate based on the acknowledgments. This approach can be used in embedded applications to provide both low energy and faster training.

During the forward trace, the activation a is calculated for all the nodes in first layer and an acknowledgment is sent to the next layer to start computation. The acknowledgment is passed until first hypothesis is computed. Following the forward trace, δ error is calculated for each neuron in all hidden layers in parallel while back propagating from the last layer towards the first. Each hardware neuron performs multiplication of inputs and synapses, addition of all the products and saves the δ error value in the memory. This is done for m iterations corresponding to m training sets in a batch of size b. It is observed that pipelined architecture improves the performance by 60% as compared to non-pipelined.

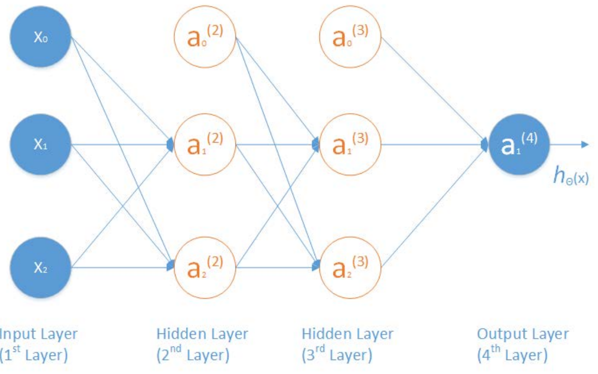

Fig. 4. Network architecture of a multi-layer neural network (Two hidden layers with three neurons, one output layer with one neuron and three input units)

$$\delta^{(l)} = (\theta^{(l)})^T.\delta^{(l+1)}.*a^{(l)}.*(1-a^{(l)}) \qquad (2)$$

where θ represents the feature or the weight, a is the activation function, and l represents computation for a particular layer.

Considering S_l units in layer l, the inputs θ^l is sized (S_l+1) by S_{l+1}, δ^{l+1} is sized S_{l+1} by 1, a^l is sized (S_l+1) by 1 and the output error for layer l i.e. δ^l is sized (S_l+1) by 1. After each iteration of nearly 250 clocks the final accumulated output stored in the memory can be used as a part of computing the gradient of the error function.

IV. Experimental Results

To compare the efficiency of our hardware accelerator for training, we have used the 45nm NanGate Open Cell Library [30] and Synopsys Design Compiler to synthesize our architecture. We have also implemented the corresponding neural networks for each experiment, also in software utilizing TensorFlow [31], and run the Python implementation on a 12-core 2.66GHz Intel i7 CPU with 8GB memory.

To increase the accuracy of comparisons, we have used three methods of power/energy measurement for the software, namely, (i) Intel Power Gadget [32] calculation, (ii) system profile measurement, and (iii) Thermal Design Power (TDP) [33] estimation. More precisely, the first method, based on Intel Power Gadget [32], monitors the real-time power consumption of the computer in the idle mode and also while training using TensorFlow and reports the difference, which would be the power consumption of the DNN training. The second method retrieves the system power profile for the idle mode and also the average power during training and reports the difference. The TDP method also reports the difference of power for the idle mode and the training duration.

In order to compare the performance of hardware and software-based approach, we conduct three experiments: (i) single neuron evaluation, (ii) small network evaluation, and (iii) practical large-scale network evaluation based on MNIST, in Section IV-A, Section IV-B, and Section IV-C, respectively.

Lastly, our proposed distributed memory system and unified memory system for a small neural network was estimated using cacti tool.

A. Single Neuron Evaluation

This experiment evaluates the architecture of a single neuron for area, time and power consumption. There are two ways to implement the sigmoid function: (i) using a separate ALU unit that shows accuracy up to 7-th decimal place, and (ii) using lookup table (LUT) that nearly approximates the output. As expected the first implementation consumes much more time, with nearly 3.5x of the clock cycles compared to the look up table implementation. Therefore, we adopt the LUT approach to implement sigmoid function.

The relation between the scaled inputs and the number of clocks required for a single neuron that performs forward trace, backpropagation and gradient calculation is shown in Table I. The complete area is reported to be 51171.5 nm^2. Total dynamic power of this neuron is 4.47 mW, whereas cell leakage power is 269.36 μW.

TABLE I
CLOCK ANALYSIS FOR A SINGLE HARDWARE NEURON

No. of Inputs	2	4	8	16	32	64	128
No. of Clock Cycles	152	225	368	661	1,192	2,325	4,647

The synthesized neuron requires a $4ns$ clock and so a two-input cell takes $0.61ms$ for one iteration while software implementation for a neuron takes $15.5ms$. Figure 5 illustrates the significant duration differences between the software and our proposed hardware implementations.

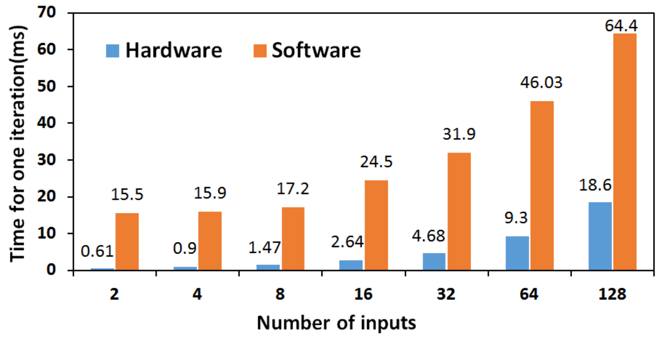

Fig. 5. Time duration of one iteration for one neuron, as a function of number of inputs

B. Small Network Evaluation

The network shown in Figure 4 is also synthesized and reports an area of $248.44\mu m^2$. Total dynamic power of this network is $23.88mW$ and cell leakage power is $1.31mW$. The two input network is evaluated using simple data set to examine the output to be red or not red, considering the average of blue and green as the other input. We evaluated two synthesized versions with $4ns$ clock, one with lookup table sigmoid and other using 32-bit IEEE floating-point ALU. The network was trained with a set of 6000 examples and tested on 1000 examples. It took 10.6 ms to train and 1.7 ms to test the LUT based network. For the ALU based sigmoid network, it took 35.4 ms to train and 5.9 ms to test.

The same network is modeled and implemented in software and evaluated for the same data set to examine red and not red. It takes 22s to train and 2.4s to test the considered data set, which means our ASIC implementation would speed up the process of training by a factor of 1000.

The total access time for one iteration in case of unified memory system was $52.82ns$ and for distributed memory system was $18.42ns$. This shows that our proposed memory architecture is 2.9 times faster. However, the average read power consumption was slightly increased from $14.01mW$ to $18.17mW$

TABLE II
RESULT OF NETWORK 784-100-100-10 WITH LEARNING RATE 0.005

	Accuracy			Resource Usage		
Input	Train	Validation	Test	Time/min	Power/W	Energy/Wh
100000	0.9983	0.9740	0.9720	3.60	29 ± 2	1.7400
200000	0.9988	0.9782	0.9748	7.19	29 ± 2	3.4752
300000	0.9996	0.9762	0.9755	10.76	29 ± 2	5.1572
400000	1.0000	0.9776	0.9778	14.97	29 ± 2	7.2355
500000	1.0000	0.9790	0.9788	18.18	29 ± 2	8.7870

C. Practical Large-Scale Network Evaluation using MNIST Dataset

We extrapolate the conditions and calculate the area, time and power consumption of the network having configuration 784-100-100-10 for the MNIST dataset [18]. The MNIST data set consists of 28 x 28 digit images classified into digits 0 to 9 with 60,000 examples for training and 10,000 examples on testing, nearly 10x times the data we considered for our basic model in Section IV-B. Since the inputs are 28 x 28, we consider 784 inputs cells, and 100 fully connected cells in each of the two hidden layers and finally 10 output cells to determine digit from 0 - 9. Note that this DNN architecture is also used in the experiments in [17].

For the software implementation, the DNN is trained with a different amount of training data and the learning rate is varied in the range of 0.0001 to 0.01. The corresponding test accuracy under different conditions is shown in Figure 6. Based on the simulations, the Intel Power Gadget and system profile measurement methods, listed earlier in this section, produce similar results, whereas the TDP estimation reports much higher results. In this work, we refer to the lowest power/energy reported by the three methods as the software training power/energy consumption. Detailed power usage and result of network 784-100-100-10 with learning rate 0.005 are shown in Table II.

As for the hardware implementation, our calculation shows that it takes $6.43s$ to train the MNSIT dataset on this practical hardware with 100,000 inputs, and the area and the total power are $6493.22mm^2$ and $5.37W$, respectively. Hence, compared with the software training, our hardware accelerator achieves 33X runtime reduction, 5X power reduction, and 168X energy reduction.

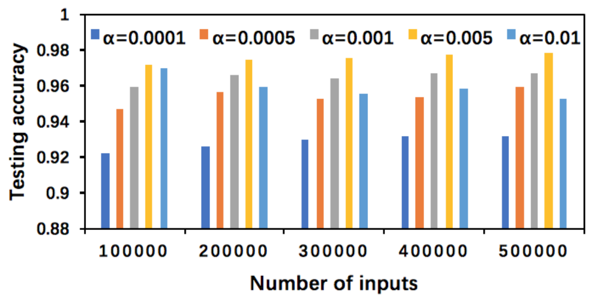

Fig. 6. Testing accuracy with different learning rates and input numbers

V. Conclusion

We presented an ASIC implementation for accelerating the training process for DNNs. A fully digital neuron was designed, which comprises a forward unit, backpropagation unit and gradient descent unit, for the forward pass, backpropagation and gradient descent calculation, respectively. Its performance was optimized by pipelining the floating point arithmetic unit and memory bandwidth issues were addressed using our proposed distributed memory architecture. Next, multiple neurons were connected to build a small DNN for a simple task, and finally the neurons were extrapolated for a standard dataset. We observed that the distributed memory architecture is 2.9 times faster but consumes slightly more power than the unified memory. We examined the trade-off between ASIC and software implementations in terms of power/energy performance and training time. Experimental results on the MNIST dataset with a 784-100-100-10 DNN showed that our proposed hardware-based approach for training process achieves 33X runtime reduction, 5X power reduction, and 168X energy reduction, as compared to the software training.

References

[1] A. Ren, Z. Li, C. Ding, Q. Qiu, Y. Wang, J. Li, X. Qian, and B. Yuan, "Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 405–418.

[2] Z. Li, A. Ren, J. Li, Q. Qiu, B. Yuan, J. Draper, and Y. Wang, "Structural design optimization for deep convolutional neural networks using stochastic computing," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 250–253.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[4] S. Renals and P. Swietojanski, "Neural networks for distant speech recognition," in *Hands-free Speech Communication and Microphone Arrays (HSCMA), 2014 4th Joint Workshop on*. IEEE, 2014, pp. 172–176.

[5] J.-T. Huang, J. Li, D. Yu, L. Deng, and Y. Gong, "Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 7304–7308.

[6] "types of neural nets." [Online]. Available: http://cs231n.stanford.edu/slides/2017

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[8] J. Li, Z. Yuan, Z. Li, C. Ding, A. Ren, Q. Qiu, J. Draper, and Y. Wang, "Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks," *arXiv preprint arXiv:1703.04135*, 2017.

[9] Y. Chauvin and D. E. Rumelhart, *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[10] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[11] "Will your next asic ever be an fpga?" [Online]. Available: https://www.eetimes.com/

[12] J. Li, A. Ren, Z. Li, C. Ding, B. Yuan, Q. Qiu, and Y. Wang, "Towards acceleration of deep convolutional neural networks using stochastic computing," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 115–120.

[13] Z. Yuan, J. Li, Z. Li, C. Ding, A. Ren, B. Yuan, Q. Qiu, J. Draper, and Y. Wang, "Softmax regression design for stochastic computing based deep convolutional neural networks," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*. ACM, 2017, pp. 467–470.

[14] J. Li, Z. Yuan, Z. Li, A. Ren, C. Ding, J. Draper, S. Nazarian, Q. Qiu, B. Yuan, and Y. Wang, "Normalization and dropout for stochastic computing-based deep convolutional neural networks," *Integration, the VLSI Journal*, 2017.

[15] Z. Li, A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan, "Dscnn: Hardware-oriented optimization for stochastic computing based deep convolutional neural networks," in *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE, 2016, pp. 678–681.

[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[17] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Proceedings of the 53rd Annual Design Automation Conference*. ACM, 2016, p. 124.

[18] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[21] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, 2016.

[22] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.

[23] M. Nazemi, S. Nazarian, and M. Pedram, "High-performance FPGA implementation of equivariant adaptive separation via independence algorithm for independent component analysis," *CoRR*, vol. abs/1707.01939, 2017. [Online]. Available: http://arxiv.org/abs/1707.01939

[24] T. Liu, S. Fang, Y. Zhao, P. Wang, and J. Zhang, "Implementation of training convolutional neural networks," *arXiv preprint arXiv:1506.01195*, 2015.

[25] "Backpropogation algorithm." [Online]. Available: http://ufldl.stanford.edu/wiki/index.php/Backpropagation_Algorithm

[26] S. Gomar, M. Mirhassani, and M. Ahmadi, "Precise digital implementations of hyperbolic tanh and sigmoid function," in *Signals, Systems and Computers, 2016 50th Asilomar Conference on*. IEEE, 2016, pp. 1586–1589.

[27] "A derivation of backpropogation in matrix form." [Online]. Available: https://sudeepraja.github.io/Neural

[28] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–8.

[29] Y. Wang, H. Li, and X. Li, "Re-architecting the on-chip memory sub-system of machine-learning accelerator for embedded devices," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2016, pp. 1–6.

[30] J. Knudsen, "Nangate 45nm open cell library," *CDNLive, EMEA*, 2008.

[31] J. Allaire, D. Eddelbuettel, N. Golding, and Y. Tang, *tensorflow: R Interface to TensorFlow*, 2016. [Online]. Available: https://github.com/rstudio/tensorflow

[32] S.-W. Kim, J. J.-S. Lee, V. Dugar, and J. De Vega, "Intel® power gadget," *Intel Corporation*, vol. 7, 2014.

[33] "Thermal design power." [Online]. Available: https://en.wikipedia.org/wiki/Thermal_design_power