Comparison of Deep Neural Network implementation in FPGA and ASIC

Md Mehedi Hassan Galib
Computer Engineering Department
University of Maryland, Baltimore, County
Maryland-21250, USA
mgalib1@umbc.edu

***Abstract*—In this work, we compare the hardware inference accelerator of compute-intensive DNN model for FPGA and ASIC implementation platform. FPGAs are an attractive choice for DNNs since they offer a programmable substrate for acceleration, however, obtaining both performance and energy efficiency with FPGAs is a laborious task even for expert hardware designers. On the other hand, ASIC implementation addresses this problem of FPGA, however, suffers in time to market constraints since DNN is ever-evolving. Hence, we adopt a six layered DNN architecture with two different activation function to point out the trade-off between FPGA and ASIC implementation to detect handwritten digit using mnist dataset. Our result shows that ASIC inference engine achieved up to 5.79% improvement in performance and reduced 78.4% power penalty compared to FPGA implementation.**

***Index Terms*—Deep neural network, activation function, FPGA implementation, ASIC implementation, area, power, performance.**

I. Introduction

In recent years, deep neural networks (DNNs) are experiencing tremendous growth for enhancing system intelligence in a wide range of applications such as vision, robotics, video analytics, speech recognition, natural language processing, targeted advertising, and web search [1]–[8] through demanding computational expensive, power hungry and longer execution time resources [9], [10]. These resource hungry algorithms are typically run on central processing unit (CPU)s or graphics processing unit (GPU)s, which are not specifically tuned to run machine learning algorithms. Hence, researchers develop several hardware accelerator to accelerate these resource hungry algorithms. For instance, AlphaGo Zero [11] is a computer program implemented on the tensor processing unit (TPU) [12] designed by Google. A TPU is a domain-specific custom application specific integrated circuit (ASIC) designed to implement machine learning algorithms. The TPU is 15-30 times faster and 30-80 times more power efficient than modern GPUs and CPUs.

Although ASICs provide significant improvement in power, performance and, efficiency for DNNs accelerator implementation [13], [14], they fail to cope with the ever-evolving DNN models. Furthermore, ASICs and customized cores come at the price of high non-recurring engineering costs over long design periods. On the other hand, field programmable gate arrays (FPGAs) offer similar advantages to ASICs such as Google's TPU by providing the ability to configure hardware specific to machine learning algorithms, including parallel execution. Moreover, FPGA implementation offers the advantage over ASIC designs of increased flexibility and decreased design to implementation time. However, FPGAs still require extensive hardware design expertise and long design cycles. In fact, several research works [15], [16] have made extensive efforts to provide FPGA accelerators for specific DNN models, or parts of DNN computation, targeted for a particular FPGA platform. Using FPGAs as an acceleration platform for DNNs is challenging as they offer a limited preset on-chip memory and often possess limited off-chip bandwidth, both of which are critical for high performance. This restriction is particularly limiting for FPGAs since ASIC designs can circumvent this issue by optimally allocating die area to on-chip memory for a single or set of target DNNs. However, FPGAs represent an intermediate point between the efficiency of ASICs and the programmability of general purpose processors, they are an attractive alternative for accelerating DNNs.

Hence, in this work, to compare performance, area, and power between FPGA and ASIC based accelerator, we choose a DNN architecture to detect handwritten digit through training from the modified national institute of standards and technology (MNIST) database of handwritten digits. Handwritten digit recognition is a classic problem in the field of image recognition and an excellent way to evaluate the performance of algorithms on classification problems [17], [18]. In particular, the key contributions in this work are:

- To build FPGA and ASIC inference engine based on the weights trained by the software implemented of DNN.
- To compare the performance and power between FPGA and ASIC inference engine for different types of DNN activation function.
- To compare the area, power, and performance between two technology nodes of ASIC implementation.
- ASIC inference engine achieved up to 5.79% improvement in performance and reduced 78.4% power penalty compared to FPGA implementation.

In Section II, the background of proposed work is provided.

Special thanks to Dr. Riadul Islam, course instructor of CMPE 641, Department of Computer Engineering, University of Maryland, Baltimore, County, MD-21250, USA.

In Section III, we discuss about our methodology for implementing FPGA and ASIC accelerator. Section IV describes the results obtained in this work, while Section V, concludes this work.

II. Back ground

In our work, we considered a fully connected DNN architecture [3], where every neuron in a particular layer is connected with every other neuron in the previous layer. In our architecture as illustrated in Fig. 1, we consider a six layered DNN architecture, where we have one input layer, one output layer (softmax for software implementation and hardmax for hardware implementation) and, four hidden layers and every neuron in these hidden layers as well as the output layer is connected with every neuron in the previous layer. This is also a feed forward neural network that means the information always goes only in one direction from input layer towards the output layer, which is unlike for recurrent neural networks where we have feedback patterns.

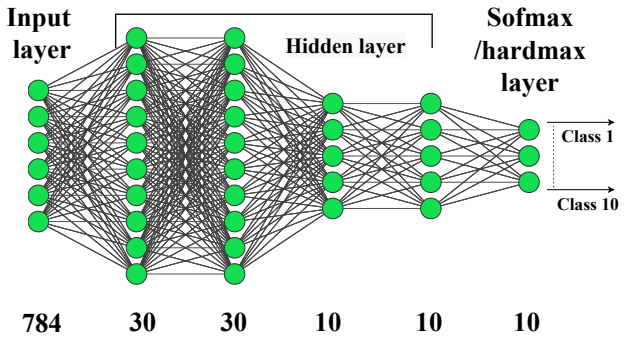

Fig. 1: Illustration of our DNN architecture considered for handwritten digit recognition of mnist dataset having 28x28 input image and 10 different output classes.

In the training phase of DNN, we have to train each neuron of the network with different weights as well as the biases in order to transfer the information efficiently. Then, we employ our pre-trained network for hardware implementation in order to avoid hardware expensive algorithms like back propagation algorithms which have differential terms. Moreover, although we can implement these expensive algorithms using lot of hardware resources, in most cases, we have to do training only once and these resources waste after training. Again, training phase is not a time critical thing, since our motivation for using hardware for DNN is for acceleration to perform inference faster, which means improvement in real time performance in detection or classification. Hence, in my network, I trained my DNN in computer with all sophistication and find the weight and bias values and directly used them in the hardware design [19].

Another important aspect of hardware implementation is to represent numbers in hardware for neural networks. The input to the DNN (pixel values of the grey scale of handwritten digit) are unsigned, however, the trained weights and biases can be both positive or negative having fractional part. For representing fractional part in hardware, we have two different choices such as *IEEE floating point representation* [20] or *fixed point representation.* Moreover, *IEEE floating point representation* has two more choices such as 32-bit representation termed as single precision implementation or 64-bit implementation called double precision implementation. The advantage of using IEEE Format is to represent very large numbers using floating-point 32-bit or 64-bit by adopting resource-hungry implementation and very difficult manipulation of floating-point numbers. Hence, in this work, we adopt fixed point representation for representing all the numbers in the hardware, since the input to the neural network is always normalized either between 0 to 1 if we considered always positive or between - 1 to +1 if we considered negative [9].

III. Methodology

As we are focusing hardware implementation of neural networks, the methodology of implementing a fully connected neural network is totally different from the software implementation. In fully connected DNN architecture, every neuron in a particular layer is connected with every other neuron in the previous layer.

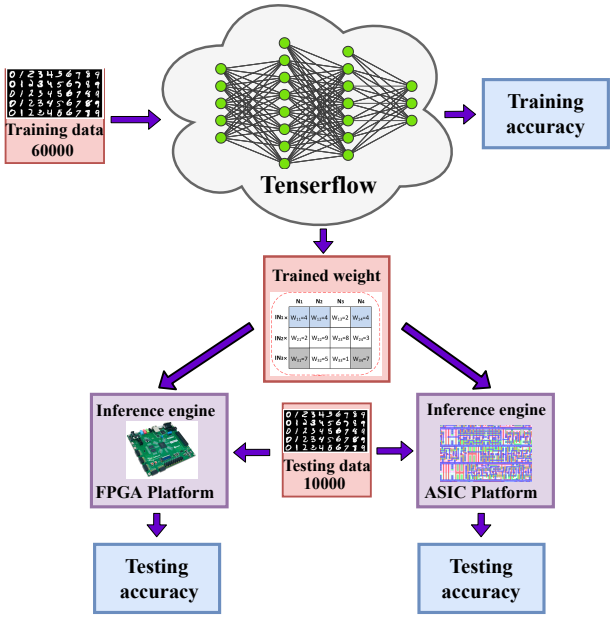

Fig. 2: Illustration of DNN hardware implementation methodology, where training of network is done using Tensorflow framework and then import the trained weight in FPGA and ASIC platform for inference to accelerate real time classification.

A. Mnist Dataset

In this paper, we adopt nnist dataset of handwritten digits as illustrated in Fig. 3, which contains a training set of 60,000 images with labels, and a testing set of 10,000 images with labels. Each input image 28 × 28 pixels (784 total pixels) having a value between 0 to 255, hence, we chose input layer

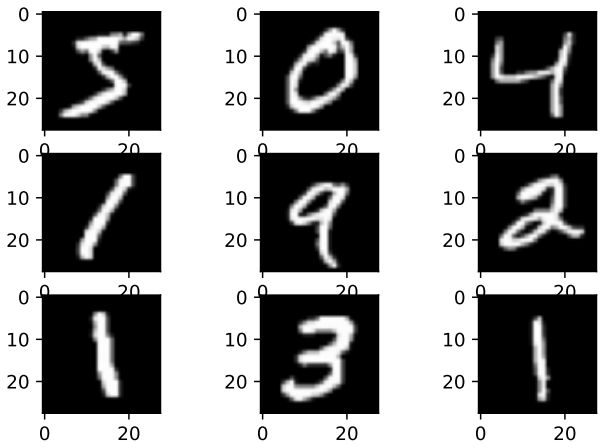

Fig. 3: Input images of handwritten digits having 28x28 pixel in mnist training dataset

as 784. For software training phase, we employ unchanged training images set, however, since perform testing in the hardware, we convert the pixel value of test images from 0 to 255 to 0 to 1 through normalization and then perform binary transformation using fixed point number representation. Since, the possible integer value is 1 for normalize image pixel, we choose one bit of fixed point representation as integer part and rest of the bits as fractional part. Moreover, since these input pixels are unsigned number, we perform unsigned binary transformation.

B. Software training

At first, we perform training for our DNN architecture using Tensorflow framework of python. For training, we choose two different types of activation function such as non-linear sigmoid activation function and linear activation function.

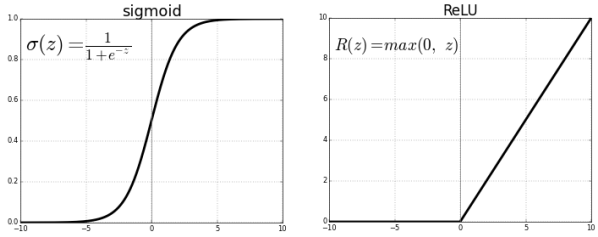

Fig. 4: For comparison between hardware realization of non-linear and linear activation function, we choose non-linear sigmoid and linear relu activation function.

C. Activation function

The type of activation function of a neuron defines the output of that particular neuron for a given input or set of inputs. Hence, many neural networks use non-linear functions such as sigmoid $\frac{1}{1+e^{-x}}$ or hyperbolic tangent $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x)}}$ as activation functions as depicted in Fig. 4. However, implementing a non-linear activation function in hardware is very difficult and resource intensive, hence, we pre-calculate these activation function and store in a read only memory (ROM)s. These ROMs are also known as Look Up Tables (LUTs). These LUT ROMs are built either using block random access memory (RAM)s or distributed RAMs (FFs and LUTs) in FPGA or using static random access memory (SRAM) in ASIC implementation.

Unlike non-linear activation function, linear activation function like ReLU activation $max(0, x)$ requires less complex computations as shown in Fig. 4. We realize this activation function as a comparator to check the maximum value between 0 and the input provided. However, as we use truncated the input based on the data width, the relu activation function accuracy falls because of the approximation.

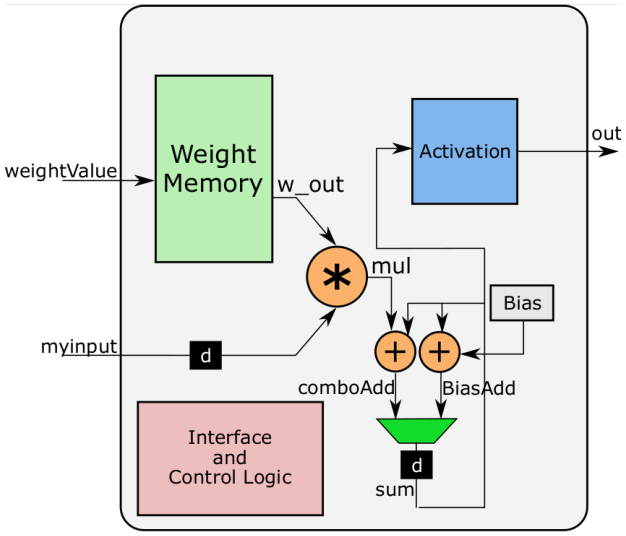

Fig. 5: FPGA realization of a neuron using weight memory, multiplier, adder, and activation function.

D. FPGA realization

In FPGA implementation, we adopt a memory called a weight memory for storing the weight. The depth of the memory depends upon the number of input coming to the neuron. Hence, for a particular neuron, each location in the memory represents a weight value corresponding to that particular input pixel. For implementing this weight memory, we choose ROM architecture rather than RAM. We will store the weight values in ROM. Then, during inference phase, test input comes with a delay d. Then, each delayed input is multiplied by a weight which is resided in weight memory. This is controlled by an Interface and Control Logic block as illustrated in Fig. 5. After multiplication, the value goes to the accumulator after adding with bias. After calculating the sum, the sum goes to an activation function like sigmoid or relu to trigger the information to the next neuron.

E. ASIC implementation

After describing the FPGA implementation using hardware description language (HDL), we adopt the same HDL script

for describing the process of logical synthesis and place and route implementation. At first, logical synthesis is performed using Synopsys Design Compiler using RTL source files with the link the libraries. It is worth mentioning that, for technological comparison, we adopt both 14nm and 32 nm technology node. The logical synthesizer translates the specified RTL files to generate gate level net-list.

In placement and routing phase, we utilize the generated gate level net-list to perform following steps such as partitioning, floor-planing, placement, legalization, clock tree synthesis, power-ground synthesis, routing, detailed routing, DRC LVS, and finally GDSII.

IV. Result and Discussion

This section compares the area, power, performance, and accuracy of the software implementation of DNN with our FPGA and ASIC realization of inference engine. To perform this analysis, we adopt following experimental setup.

A. Experimental setup and data set

As mentioned in sub-section III-A, we adopt handwritten digit recognition mnist dataset (60000 training data + 10000 testing data). For software training purpose, we chose Tensorflow framework python which run in Intel® Xeon(R) Silver 4210u CPU. For FPGA implementation, we perform behavioral simulation using Xilinx® Vivado and hardware verification using zed board, a family of Zynq™-7000 development board. Furthermore, for ASIC implementation, we resort to Design Compiler (DC) provided by Synopsys for gate level net-list, and IC Compiler II from Synopsys for placment and routing phase. Additionally, for technological node comparison, we adopt Synopsys Armenia Educational Department (SAED) 14 nm and 32 nm technology nodes.

B. Training and testing accuracy

After performing training in the Tensorflow, we calculate the training accuracy of the DNN model. Then, using the testing dataset, we calculate the testing accuracy. In software testing, relu activation function has less accuracy than the sigmoid activation function because of the linear activation. However, in the hardware testing, the accuracy depends on the width of the data as well as how many bits we considered for the approximation of the activation function. It is noticed from the Table. I that, with change the data-width, the accuracy of the DNN architecture fall apart, and in case of relu activation function with 8 bits data-width, the accuracy falls from 91.96% to 2.56%.

C. FPGA resource utilization and constraints

We estimate the FPGA resource utilization with maximum clock frequency and total on chip power for all the different combinations of our DNN model in order to find the trade-off between power, performance, area, and accuracy. Our results show that, sigmoid activation function with 16 bit data-width and 32 data point for sigmoid activation function requires optimal resource utilization with higher clock speed with sacrificing a bit of on chip power. In order to calculate resource for a particular DNN model, we consider the number of LUTs, number of FF, number of block RAM and DSP block uses.

TABLE I: Relu 8 bit data-width delays worst testing performance due to large estimation in the testing data.

Activation	Accuracy in %			Data-width (bit)
	Training	Testing		
		Software	Hardware	
σ (10 bit)	98.3	95.23	91.96	16
σ (5 bit)	98.3	95.23	91.67	16
σ (5 bit)	98.3	95.23	7.96	8
$Relu$	95.99	91.50	71.80	16
$Relu$	95.99	91.50	2.56	8

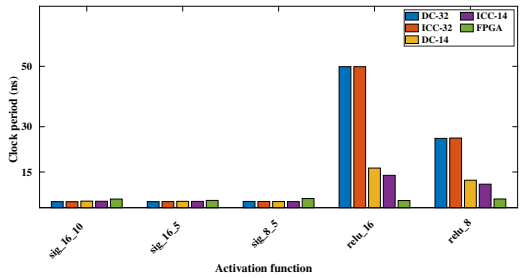

Fig. 6: Clock period comparison, where relu-16 bit data-width shows worst performance because of non-optimized design in ASIC implementation.

D. ASIC implementation

Like, FPGA resource utilization, we calculate the resource, power, and performance calculation after both logical synthesis and placement & route. It is seen from TABLE III that, like FPGA implementation, the sigmoid activation function with 16 bit data-width and 32 data point for sigmoid activation function shows optimal resource utilization with higher clock speed with sacrificing a bit of on chip power. However, the relu activation function have the worst resource and clock utilization for both of data-widths. Surprisingly, the total area of relu activation functions are very large compared to their FPGA implementation, which is because, in FPGA the distributed memory are uniformly placed throughout the FPGA fabric, however, of ASIC implementation, we have not permission to chose where we place the memory blocks.

Like the DC results, the ICCII results show almost similar area, power, and performance compared to DC as illustrated in TABLE IV. However, these ICCII results have lower in magnitude due to the optimization, which is perform in placement and routing step.

Furthermore, in case of 14nm technology node as shown in TABLE V VI, we observe the same trends in DNN model, however, the magnitude of area, power, and performance are

TABLE II: Sigmoid activation function with 16 bit data-width and 32 data point for sigmoid activation function shows optimal resource utilization with higher clock speed with sacrificing a bit of on chip power.

Activation	Resource utilization				Frequency	On chip power (w)	
	LUT	FF	DSP	BRAM	(max) MHz	Dynamic	Static
σ (10 bit)	6671	5505	160	35	165.3	0.842	0.122
σ (10 bit)	6356	5384	160	15	178.82	0.604	0.116
σ (10 bit)	9448	5223	0	15	161.00	0.348	0.11
Relu	6675	6834	160	15	180.23	0.711	0.118
Relu	9532	5095	0	0	172.44	0.208	0.107

TABLE III: Like FPGA realization, sigmoid activation function with 16 bit data-width and 32 data point for sigmoid activation function shows optimal resource utilization with higher clock speed with sacrificing a bit of on chip power (ASIC DC implementation 32 nm).

Activation	Resource utilization (μm^2)			Clock	On chip power (w)		
	Combinational	Sequential	Total area	period (ns)	Dynamic	Leakage	Total
σ	2544.49	3244.4	7103.32	5.17	0.56127	0.0015123	0.56786
σ	2848.95	3019.99	7281.69	5.18	0.7394	0.0015136	0.7537908
σ	2331.63	2961.52	6531.41	5.25	0.067394	0.0013136	0.06937908
Relu	2057280.18	2721594.63	10794115.18	49.88	0.0169165	0.001160	0.018021
Relu	1051768.84	1364241.987	4681518.49	26.13	0.00948	0.000595	0.0111

TABLE IV: ASIC ICC2 implementation of 32 nm showing almost similar magnitude like ASIC DC implementation of 32 nm, however, these values are lower in magnitude due to the optimization in place and route steps.

Activation	Resource utilization (μm^2)			Clock	On chip power (w)		
	Combinational	Sequential	Total area	period (ns)	Dynamic	Leakage	Total
σ	2544.49	3244.4	7103.32	5.17	0.56127	0.0015123	0.56786
σ	2848.95	3019.99	7297.24	5.19	0.48961	0.0014916	0.497218
σ	2361	3006.52	6563.01	5.23	0.05866	0.0012961	0.06718
Relu	2057280.18	2721594.55	6007895.7	49.87	0.00794	0.000996	0.008222
Relu	1051768.84	1364241.94	3018947.85	26.25	0.00487	0.000579	0.0051111

TABLE V: ASIC DC implementation of 14 nm showing almost similar magnitude like ASIC DC implementation of 32 nm, however, these values are lower in magnitude due to the optimization in place and route steps.

Activation	Resource utilization (μm^2)			Clock	On chip power (w)		
	Combinational	Sequential	Total area	period (ns)	Dynamic	Leakage	Total
σ	2060.48	2371.89	5149.05	5.35	0.146658	0.00024623	0.166695
σ	1742.32	2037.86	4782.94	5.28	0.10970	0.00024720	0.118605
σ	1482.61	1923.56	4108.38	5.21	0.05996	0.00012221	0.043435
Relu	299155.91	381555.3	5882155.24	16.3	0.00931981	0.00024504	0.01285
Relu	146353.76	191272.53	4910118.75	12.28	0.005627	0.00012086	0.0068062

TABLE VI: ASIC ICCII implementation of 14 nm showing almost similar magnitude like ASIC ICCII implementation of 32 nm, however, these values are lower in magnitude due to the optimization in place and route steps.

Activation	Resource utilization (μm^2)			Clock	On chip power (w)		
	Combinational	Sequential	Total area	period (ns)	Dynamic	Leakage	Total
σ	2046.82	2191.43	4907.05	5.31	0.0978	0.000247	0.0985
σ	1740.32	2157.86	4690.19	5.27	0.0721	0.000109	0.0723
σ	1425.61	1907.56	4081.38	5.19	0.0397	0.0000971	0.0398
Relu	299152.91	381551.3	5682155.24	13.91	0.00892	0.000296	0.00908
Relu	146353.76	191272.53	482114.958	10.97	0.00312	0.000162	0.00385

less than corresponding DC or ICCII implementation of 32 nm, which is obvious for the technology migration.

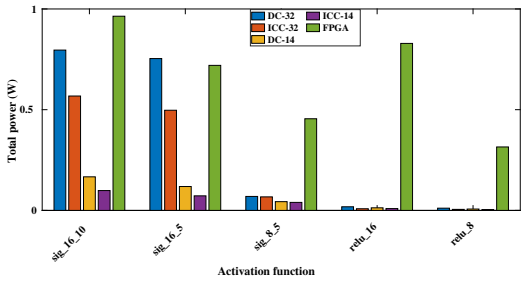

Fig. 7: Input images of handwritten digits having 28x28 pixel in mnist training dataset

E. FPGA and ASIC comparison

In case of clock period comparison as depicted in Fig. 6, FPGA clock period is larger for all the ASIC implementations for DNN model having sigmoid activation function, which is 5.79% improvement compared to the FPGA. However, ASIC implementation with relu function shows worst clock period compared to FPGA because of non-optimized design in ASIC.

In case of power calculation as illustrated in Fig. 7, all the ASIC implementations outperform the FPGA implementation for both dynamic and leakage powers. This is because, in case of FPGA, the power is calculated for overall FPGA fabric, however, in case of ASIC, the power is calculated of the implemented design. Moreover, we can implement power optimize ASIC design for all the DNN implementations and in that case the power can be reduced more for the ASIC.

V. Conclusion

In this work, we build FPGA and ASIC inference engine based on the weights trained by the software implemented of DNN. Then, we compare the performance and power between FPGA and ASIC inference engine for different types of DNN activation function and finally, we show that ASIC inference engine achieved up to 5.79% improve- ment in performance and reduced 78.4% power penalty compared to FPGA implementation.

References

[1] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 27–40.

[2] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun, "Deep image: Scaling up image recognition," *arXiv preprint arXiv:1501.02876*, vol. 7, no. 8, 2015.

[3] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 223–238.

[4] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[5] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[6] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[8] 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*. PMLR, 2013, pp. 1337–1345.

[9] J. Si and S. L. Harris, "Handwritten digit recognition system on an fpga," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2018, pp. 402–407.

[10] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[12] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[14] 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 *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 609–622.

[15] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 682–687.

[16] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. IEEE, 2010, pp. 257–260.

[17] S. Jain and R. Chauhan, "Recognition of handwritten digits using dnn, cnn, and rnn," in *International conference on advances in computing and data sciences*. Springer, 2018, pp. 239–248.

[18] K. T. Islam, G. Mujtaba, R. G. Raj, and H. F. Nweke, "Handwritten digits recognition with artificial neural network," in *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*. IEEE, 2017, pp. 1–4.

[19] K. Vipin, "Zynet: Automating deep neural network implementation on low-cost reconfigurable edge computing platforms," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 323–326.

[20] M. L. Overton, *Numerical computing with IEEE floating point arithmetic*. SIAM, 2001.