International Journal of Computer Information Systems and Industrial Management Applications.
ISSN 2150-7988 Volume 11 (2019) pp. 046-054
© MIR Labs, www.mirlabs.net/ijcisim/index.html

Received: 16 Dec, 2018; Accepted: 16 March, 2019; Publish: 17 April, 2019

Hardware Accelerators for Neural Processing

Shilpa Mayannavar[1] and Uday Wali[2]

[1] C-Quad Research, KLE Dr M S Sheshgiri College of Engineering & Technology,
Belagavi-590008, Karnataka, India
mayannavar.shilpa@gmail.com

[2] Department of EEE, KLE Dr M S Sheshgiri College of Engineering & Technology,
Belagavi-590008, Karnataka, India
udaywali@gmail.com

***Abstract*: There has been a great change in the computing environment after the introduction of deep learning systems in every day applications. The requirements of these systems are so vastly different from the conventional systems that a complete revision of the processor design strategies is necessary. Processors capable of streamed SIMD, MIMD, Matrix and systolic arrays do offer some solutions. As many new neural structures will be introduced over next years, new processor architectures need to evolve. In spite of the variability of Artificial Neural Network (ANN) structures, some feature will be common among them. We have tried to implement the hardware components required for most of the ANNs. This paper highlights some of the key issues related to hardware implementation of neural networks and suggests some possible solutions. However, the arena remains very open for innovation. Low precision arithmetic and approximation techniques suitable for acceleration of computational load of neural networks have been implemented and their results have been presented. We also show that for a given ratio of area occupied by serial multiplier to that of a parallel multiplier, a threshold exists beyond which the serial multipliers have a distinct performance advantage over parallel multipliers. Need for multi-operand operations and methods to implement them have been discussed.**

***Keywords*:** Artificial Intelligence, Artificial Neural Networks, Deep Learning, Hardware Accelerators, Low Precision Arithmetic, Neural Network Processor.

I. Introduction

Development of commercial applications using Deep Learning (DL) has set a pace of technological advancements that is unprecedented in history of computing. Artificial Neural Networks (ANNs) used in DL, generally called Deep Neural Networks (DNNs) process data in massively parallel slow low precision streams in contrast to single (or few) processor, linear memory, fast computation of high precision arithmetic used by conventional computers. When ANNs are implemented on conventional processors, data transfer over limited bus bandwidth becomes a bottleneck. Data flow structures with large number of compute cores would be more suitable for ANN implementations. However, data sharing among thousands of compute cores and their synchronization poses challenges in computer architecture. The non-linear activation functions that limit the computational output to a small numeric range, precision and accuracy also need attention. Therefore, DNN implementations present a completely different computational environment which needs to be addressed by new hardware architectures to achieve better performance.

Neural computing demands availability of special purpose processors with thousands of compute cores. To some extent, this need has been met by use of Graphics Processing Units (GPUs) [1], [2], [3], [4]. They have been used in variety of applications including image processing, robotics, image interpretation, natural language processing and in many other intelligent use cases. Operations like scaling, rotation and transformations frequently used in GPUs are also useful in neural computations. Hardware capable of vector, matrix and tensor representations of data can exploit massive parallelism of these architectures and provide a method to speed up the computations. However, several differences do exist: e.g., non-linear activation functions like sigmoid, hyperbolic tangent (tanh), reLu and softmax are regularly required in neural computations. Interconnections in neural network can be more varied than in case of GPU operations. Clearly, there are several features in neural computations that have no equivalent in graphics processing. Many research organizations including ARM [5], Imagination Technologies, Google [6], Intel [7], Apple, Cadence Tensilica, Cambricon [8], IBM [9] et al. are developing hardware to implement neural architectures. These processors were designed to address requirements of specific neural networks e.g., Google TPU is best suited for TensorFlow implementations, TrueNorth from IBM is suited for Multi Layer Perceptron implementations, Cambricon AI chip used in Huawei's Kirin-970 mobile processors are suitable for vector and matrix operations, etc.

While special purpose chips are being developed for DL applications, traditional processors are implementing extended instructions to support such applications. Notably, Intel's Advanced Vector Instructions (AVX), broadcast instructions, Fused Multiply Accumulate (FMA) Packed Single Precision and streaming SIMD instructions have reduced the performance gaps between CPU and GPUs. These instructions reduce the need to transfer data between memory and registers during repeated computation of

multiply-add cycles used in many DL architectures. Further developments in DL architectures will require redesign of processors and accelerators There are new types of DL structures available viz., Auto Resonance Network (ARN) [10], Generative Adversarial Networks (GAN) [11], Explainable Neural Networks [12], [13], that have been useful in various applications. As expected, it is necessary to identify and redesign hardware units that can accelerate the processing while using such networks.

We discuss some of the issues relevant to design of hardware accelerators and processors for neural computation. Section II gives a brief overview of the computational needs of contemporary neural networks. Some of the issues discussed include implementation of a new low precision number format, approximation of activation functions using piece-wise-linear (PWL) and second order interpolation (SOI), effect of precision on accuracy of result, multiplier, multi-operand addition, etc are presented in section III. Conclusion and references for further research are given at the end.

II. Computations in Contemporary Deep Learning Networks

Several Deep Neural Network structures like Recurrent Neural Networks (RNNs), Auto-encoders, Restricted Boltzmann Machines, Radial Basis Functions, word2vec, etc have contributed to the modern rush in DL but two structures that stand tall are Convolutional Neural Networks (CNNs) [14],[15] and Long Short-Term Memory (LSTM) [16]. Several implementations of CNNs are presently available. They all have multiple Convolution layers, pooling layers, activation layers like Softmax and ReLU, Fully connected layers and decision layers at the output. Combining them in a particular order to address data specific requirements gives them varying recognition and classification capabilities. LSTM networks are derived from RNN but have additional types of nodes, e.g. the central carousal, forget gates (remember gates), etc. LSTM networks also use conventional activation functions like hyperbolic tan (tanh) and sigmoid ($1/(1+e^{-x})$). CNNs have been useful in image classification while LSTM networks are useful in time series classification tasks. There are several other types of networks like Generative Adversarial Networks (GAN) that are capable of not just understanding but synthesis of creative products like digital paintings and music, that are very much like the ones produced by practicing artists [11].

Several DL development platforms are available for development and deployment. Few popular ones are Eclipse DL4J Java libraries, Theano [17] from Univ. of Montreal (2007), Caffe [18] from Univ. of California at Berkeley (Dec 2013), TensorFlow [19] from Google (Nov 2015), PyTorch by Facebook (Oct 2016), etc. Most of these platforms implement low level code in C or C++ and provide Python API for easy implementation. Most of them can run on CPU or use GPU for faster operations. Nvidia CUDA GPUs have contributed to the developments in many of these libraries. Recently pre-trained networks have been made available as web services. In fact, some of the progress in AI has been due to availability of these web services. AWS from Amazon, Microsoft Azure and Google Cloud Platform, Google Collaboration are significant. Most of the DNNs require millions of data records, images, audio patches, etc to be processed to train the networks. This presents a critical bottleneck for development. Therefore, availability of special purpose hardware is necessary for further development in this field. Some of the key issues to be considered while designing accelerators for neural computing are discussed here below.

A. *Instruction set*

Instruction set of conventional processors implement procedure flow, while neural networks are best implemented as data-flow machines. Current GPU implementations use Multiple Instruction Multiple Data (MIMD) or streamed SIMD (Single Instruction Multiple Data) where a continuous stream of data is presented to the data-flow machine with implicit or programmed computational chain. Most of the operations in neural networks can be performed in parallel. Therefore, scalar, vector and tensor operations must be included as the part of the instructions. Introduction of newer instructions like Intel AVX hint at the changes necessary to move forward. DL and AI workloads require computational chains to be implemented without intervening memory calls to be computationally efficient. Otherwise, the memory–register data transfer overhead will dominate the computational time. Streamed-data instructions persist over long data sets, performing the same operation over the entire data set. This reduces the time required to perform fetch-decode steps to be performed at every unit of data. Similarly, systolic arrays and data flow architectures can be used to perform Fused Multiply Accumulate instructions.

B. *Training and Runtime Environments*

In a conventional processor, an algorithm works the same way during development time (debug) and run time environments. However, the scenario is very different when handling DL systems: They require computationally intensive training, which has no equivalence in conventional algorithms. Training process actually builds the 'program.' Training DL systems is a slow process as millions of weights have to be computed and adjusted to reduce the overall error in classification or recognition. The run time environment of DL systems is largely feed forward and hence fast.

Building processors with additional hardware to facilitate training will make the chip area-inefficient and hence expensive. Tuning the hardware to runtime will deteriorate already slow training time. Implementing a trained network has different set of constraints from those present during training. We need to explore if there are any DL architectures where the training and run-time are not very different. Otherwise, the task would be to make the processor efficient during training as well as during runtime.

C. *On chip CPU count*

Conventional processors have a single CPU per chip. More modern processor chips may have Quad-cores or Octa-cores or may be Hexa-cores. These architectures are excellent for running multi-threaded applications. But, the number of parallel computations possible in DNNs can run into millions. Some of the contemporary special purpose processors and GPUs have thousands of 'compute cores.' For example, NVidia Tesla V100 GPU [20] contains 84 streaming

multiprocessors, each with 64 32 bit floating point units, 64 integer units, 8 tensor units, etc bringing the throughput to 125 TFLOPS (Tera Floating Point Operations per Second). This is only a beginning of the ramp expected ahead. With such a large number of compute cores on chip, many design challenges can be anticipated.

D. Core Instructions

We have come a long way in device density from the days when RISC/CISC classification was relevant to the area of chip. MIMD and streaming SIMD designs rule the contemporary GPU designs. As the processors design has to specifically address the computational load of DL implementations, attention has to be given to reusable partitions. Most of the DNNs contain vector/tensor operations that can be implemented as trees of compute cores. Networks like CNN use specific operations like convolution and pooling, both of which can be implemented as a combined sum of product operation. Similarly weighted sum of Multi-Layer Perceptron (MLP) can also be implemented using multiply and accumulate units.

On the other hand implementing non-linear activation functions can be a challenge. Using Taylor series expansion for such functions is computationally wasteful. We may observe that neural computations can tolerate small errors in computation and hence use of low precision arithmetic is often sufficient. Therefore, study of approximation methods for implementation of activation function may be conducted. Section III B presents implementation of piece-wise linear and second order approximations for implementing activation functions.

Many neural networks require weights to be updated during training. Weights are calculated as matrix or tensor equations, which are also implemented using multiply and accumulate units. These calculations compute new values of weights that need to be transferred to corresponding units before starting the next round of computations. This bulk transfer of data between computational units and memory needs to be addressed at the instruction level as it is a basic operation.

Several other candidates for hardware implementation of instructions may exist, which need to be explored with reference to the type of network being implemented.

E. Efficiency in Massively Parallel Operations

Operations like multiplication can be performed serially or in parallel. Parallel implementations will run several times faster than serial implementations, but at the cost of silicon area. Given the area of a parallel unit, it is possible to implement several serial units in the same area. As the number of parallel units increases, the number of serial units increases faster. Therefore, a threshold/cross-over of area efficiency vs. speed of operation will exist. When the number of parallel units is small, as in case of conventional processors, the parallel units have a performance advantage. However, .as the number of parallel units exceeds a threshold, throughput of combined serial units will exceed that of parallel units. Such situations will be present in implementation of processors for DNNs. A sample study is presented in section III C.

F. Re-configurability

Section II B, presented a case where the computational demands during training and runtime are different. Does re-configurability of hardware holds the key to training – runtime dilemma? One of the strong applications of reconfigurable hardware is Software Defined Radio (SDR), where modulation type and parameters can be changed when requirements change. For example, same SDR board may be used to transmit signals with different types of modulation schemes like AM, FM, FSK, etc depending on the end user requirements. Similar situation exists in case of DL also. The type of network to be used depends very strongly on the type of input, e.g., CNN for image classification and LSTM for audio or time series classification. If the processor is designed with only one type of DNN in mind, it may become very inefficient to implement other type of DNN. So there is a need to estimate common needs of DNNs and define a maximal but 'functionally complete' implementation that allows easy re-configurability. It will be interesting to study how much of the DL hardware can be reconfigured with respect to functionality and reconfiguration overhead.

G. Memory Bottleneck and Computation in Memory

As the number of compute cores in DL hardware tends to be large, the need to transfer bulk data between heap memory, local memory and registers get complex. The number of busses that can be realized on a chip will be a limiting factor. Small number of busses means larger memory latency. Large number of busses will not only occupy space on chip but partition the chip area into isolated sections. We have to make two observations here: (a) Linear Memory organization is inefficient for DNN implementations and (b) use of traditional bus oriented architecture becomes a bottleneck. Currently, transfer rates on NVidia Tesla V100 with a matrix type of organization achieves a transfer rate of 900GB/s to 82 streaming multiprocessor units using 4096bit HBM2 memory interface. There is a need to rethink of memory design as well as transfer mode. A good hierarchical memory organization may use high speed serial transfer between heap and core but use multiple buses within complex compute cores. Actual organization depends on the processor design, intended application, etc but much like the hierarchical memory (flash, heap, L1 cache, L2 cache, …), the data transfer also needs to be hierarchical. Again, high speed serial transfers may be more feasible than parallel bus transfers.

H. Numerical Precision and Accuracy of Computation

Most of the ANNs will use limiting activation functions. That means the output of a neuron is always bound to an asymptotic limit. Successive transfer of data between nodes will encounter these limiting activation functions multiple times and the output will be limited to a small value. Multiplication or addition has very little effect on the output of a node once it reaches the asymptotic limit. Note that the derivate of the activation functions approaches zero when the output is approaching asymptotic limits. The derivative is significant only in a limited range of the number scale. Therefore, the range of input values where the output will significantly change is limited to a small range of values. This effect is generally called as the problem of vanishing or exploding gradient.

Further, neural networks should produce useful output

even in presence of noisy environments or input. In fact noise tolerance is a hallmark of neural networks. That is why they are expected to produce correct results even when the applied input has no precedence. These conditions imply that precision of input or of computation will have a small bearing on the decision taken by a neural network. The accuracy of digital systems comes at high cost of precision, which is not necessary to achieve correctness of output using neural computations. Many of the modern DL processors use a low precision arithmetic to achieve speed. For example, Intel uses a format called flexpoint [7] for AI applications. We have been using a low precision number format for DL for some time now [22]. Section III A has some details on this format and justification for its use.

I. Multi-Operand Instructions

Most of the DNNs use sum of product computation to generate neuronal outputs. The number of inputs (synapses) to these nodes can be of the order of hundreds. For example, first convolution layer in Alexnet [23] uses nodes with 363 synapses, ConvNet uses 3072 synapses in first convolution layer as well as the pooling layer. TrueNorth analog AI chip from IBM [9] supports 256 synapses per neuron. Each such computational node has to perform as many multiplications per neuron followed by an addition. Therefore, performing two operand operations on such computation would be very inefficient in terms of throughput; most of the time would be spent moving the data between partial sum storage elements. Therefore, use of multi-operand operations would be preferable. Systolic arrays could be used to implement successive multiplication and addition. Modern instruction sets include operations like 'Fused Multiply and Accumulate Single Precision' to address this problem. Streamed SIMD architecture may also be used to compute the sum of multiple products operation. A multi-operand adder is presented in section III D.

It is interesting to note that the number of synapses of a neuron in human brain is of the order of 100,000 and that in mice is about 45,000. While the current range of processors for DNNs has an amazing complexity, we are at least three to four orders of magnitude behind the natural intelligence. This shows the complexity of processors to come.

J. Explainable Neural Networks and other Networks

One of the bottlenecks to improving the performance of deep neural networks is the ambiguity in explaining the decision the network took. Largely DNNs are seen as a black box because of the large number of inputs and synapses connecting to individual nodes in the network. Similar problems were also present in classical neural networks using gradient descent and back propagation. This problem was highlighted by [24] as the inability to assign a state or node of a network to how and why a network took a specific decision. More modern networks like CNN work on parts of the input and merge the components in successive layers of neurons. However, the association is not recorded or remembered explicitly. Therefore, it becomes difficult to express a causal relation between the input and output [25]. Newer DNNs will store such information about relations and causes as an associated part of the network. This again is similar to the semantic net of the classical AI networks. Some networks like ARN address the issue of explainable DNNs better than others. Fast implementation of tunable ARN nodes is discussed in our earlier publication [26].

III. Hardware Implementations

This section contains discussion on some of the experiments with implementation of some of the hardware modules required for various Deep Neural Network Architectures.

A. Number Representation

The accuracy and speed of computation in neural networks depend on the numerical representation. The performance of a neural network is largely dependent on how the inputs, weights and output of a neuron are represented. The output of ARN nodes is limited to a small range. The input that produces output is also limited to a small range of numbers around resonance. Therefore, it is possible to translate the input range to a smaller range. The spread of input around the resonance point is limited to plus or minus 1. Therefore, it was sufficient to use a 16 bit fixed point number with 12 bits to represent the fractional part. Figure 2 shows that using 12 bits is enough to limit the accuracy of number to 3 fractional digits after calculations.

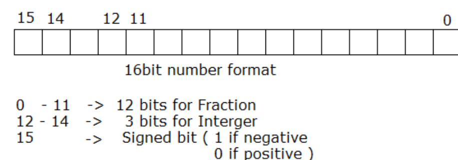


Figure 1. Number format

bit position	decimal fraction
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.0009765625
11	0.00048828125
12	0.000244140625

Figure 2. Selection of 12 fractional bits to achieve 3-fractional digit accuracy

It is worth noting that the Silicon area required for implementing a design with 16-bit representation is considerably less as compared to 32-bit design. Due to the noise tolerant nature of neural networks, some loss in precision is acceptable. Therefore, use of 16-bit was preferred to any other higher bit width. The proposed 16-bit number format is shown in Figure 1. The reason behind choosing 12 bits for fraction is to get the accuracy up to 3 fractional digits is illustrated in Figure 2.

Recently, Intel has announced its numerical format called flexpoint, designed for deep learning systems [7], [21]. It is reported that, performance of a network with 16-bit flexpoint closely matches with that of 32-bit floating point. Most of the current research work on accelerators is focusing on the numerical format [27], [28] because the performance depends

on the speed of computation rather than precision and accuracy of calculation. As the number of modules to be realized on the accelerators is large, saving in silicon area due to reduced precision gains importance.

B. Activation Functions

In Artificial Neural Networks (ANNs), an activation function is defined as the action potential required for firing of a neuron. The response of a neuron is non-linear function of the applied stimulus. The output of a neuron in ANN is calculated as a weighted sum of inputs followed by the activation function. Several types of activation functions are used in ANNs, viz., sigmoid, tanh, reLu and softmax, each with different capabilities. The non-linear curves for these activation functions are shown in Figure 3.

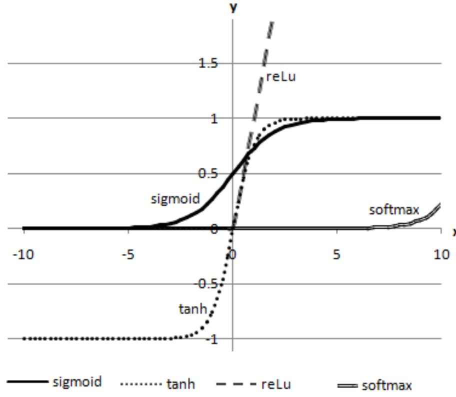


Figure 3. Activation functions in ANN

Sigmoid is an activation function having an 'S' shaped curve as shown in Figure 3. It is defined as

$$y = \frac{1}{1+e^{-x}} \tag{1}$$

Equation (1) involves an exponent, which may be expanded using Taylor series as

$$y = \frac{1}{1+1-x+\frac{x^2}{2!}-\frac{x^3}{3!}+\ \dots} \tag{2}$$

It can be seen from Equation (2) that, it is computationally expensive and hence direct implementation of this is inefficient. To address this challenge we have used two approximation methods namely Piece Wise Linear (PWL) and Second Order Interpolation (SOI). Implementing the sigmoid function using direct method and using approximation methods is discussed and compared in our earlier publication [20].

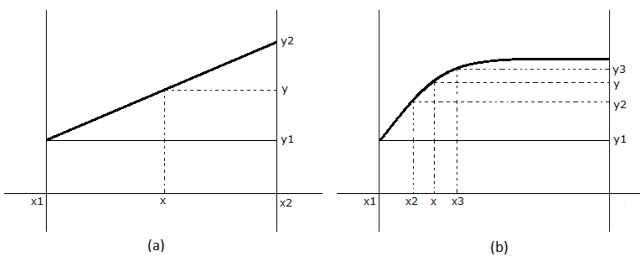


Figure 4. Approximation methods (a) PWL (b) SOI

Consider the approximation curve shown in Figure 4(a), two points on the curve (x_1, y_1) and (x_2, y_2) are assumed to be known and they are stored in the look-up-table (LUT). Intermediate values are approximated by a straight line. Therefore, we can use a simple linear equation to calculate the sigmoid of any given value x as

$$y = m\,(x - x_1) + y_1 \tag{3}$$

Where, m is the slope of the curve defined by;

$$m = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) \tag{4}$$

The accuracy of this method varies depending on the distance between the known points. We have experimented this with uniform and non-uniform distances. The accuracy increases by decreasing the distance between two known points but it will also increase the size of LUT and hence more Silicon area is required. To balance this trade-off, non-uniform distance is introduced between the points where the error is maximum.

The accuracy can further be improved by Second Order Interpolation using three point approximation. The second order curve as shown in Figure 4(b) is taken for this approximation. In this method, three points (x_1, y_1), (x_2, y_2) and (x_3,y_3) are assumed to be known. The second order coefficients a, b, c are calculated using the formula given in equations (5) to (7).

$$a = \frac{((x_2-x_1)(y_3-y_1))-((x_3-x_1)(y_2-y_1))}{(x_2-x_1)(x_3-x_1)(x_3-x_2)} \tag{5}$$

$$b = \frac{(y_2-y_1)-a(x_2^2-x_1^2)}{(x_2-x_1)} \tag{6}$$

$$c = y_1 - ax_1^2 - bx_1 \tag{7}$$

When the new input is entered, it will be compared with the stored values from LUT to fetch the values of coefficients a, b and c. The output is calculated using the equation (8).

$$y = (ax + b)x + c \tag{8}$$

The structure of LUT for both the methods is shown in Figure 5. With non-uniform distance, the size of a look up table for PWL is 60 bytes and for SOI is 80 bytes. Table 1 summarizes the computation complexity involved in the implementation of a sigmoid function using direct and the proposed approximation methods. It can be noticed that, the number of operations (addition, multiplication and fetch) required for sigmoid function using approximation methods is very less compared to Taylor series expansion given in equation (2).

x	y	m
0	0.5	0.2499
0.25	0.5621	0.2411
⋮	⋮	⋮
0.75	0.6791	0.2075

(a)

x	a	b	c
0	-0.0077	0.2506	0.5
0.25	-0.2231	0.2578	0.4991
⋮	⋮	⋮	⋮
0.75	-0.0426	0.2821	0.4915

(b)

Figure 5. Structure of a look-up-table for (a) PWL, (b) SOI approximation methods

Method	Operations					No. of clock cycles
	Look up	Add	Sub	Mul	Div	
Eq.(2)	0	6	6	65	10	N/A
PWL	3	1	1	1	0	16
SOI	3	2	0	2	0	28

Table 1. Computational complexity of sigmoid function.

It is also important to note here that, the gradient of sigmoid curve is significant only for few values of x. It can be easily seen from the sigmoid curve that, for any values of x<-5, the output is 0 and for any values of x >5, the output is 1. As the input values can be between -5 to 5 and the output value can be between 0 and 1, our number format shown in Figure 1 supports this requirement and therefore, we have used this number format to represent input and output in all our implementations.

The PWL method involves few numbers of operations as compared to SOI, therefore, it will require less clock cycles. However, with SOI it is possible to obtain greater accuracy than with PWL method. There is a trade-off between speed and the area. The simulation result of sigmoid implementation using PWL and SOI method is shown in Figure 6 and Figure 7 respectively.

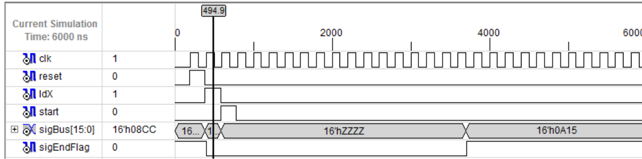


Figure 6. Simulation result for sigmoid using PWL approximation

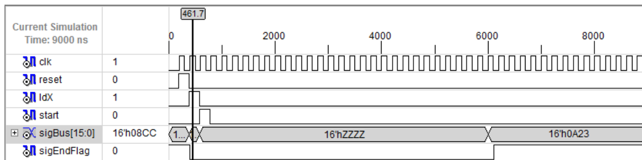
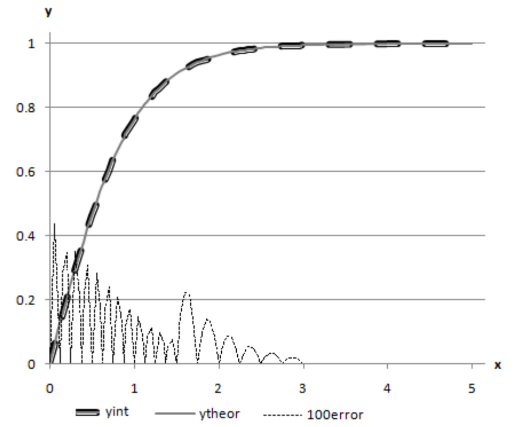


Figure 7. Simulation result for sigmoid using SOI approximation

A hyperbolic tangent (*tanh*) is another activation function used in neural networks. It also has an S-shaped curve like a sigmoid function, but with the R between (-1, 1). The equation for a *tanh* function is given in equation (9).

$$y = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{9}$$

The concept of approximating a curve can be applied to *tanh* function also. The error deviation of 0.43% with PWL and 0.018% with SOI at non-uniform distance is noted. The results of approximation are shown in Figure 8. Other activation functions like reLu and softmax are quite straight forward and they can be directly implemented without much of a problem.



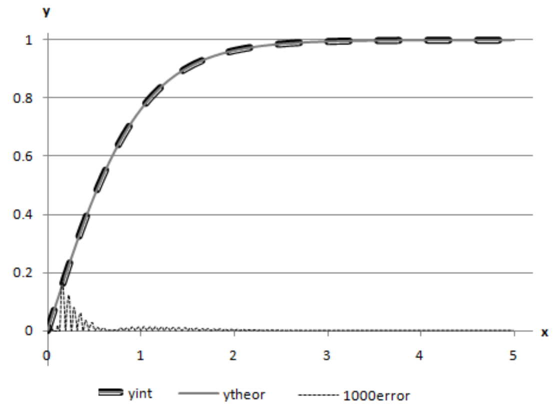


Figure 8. Result of approximating *tanh* curve at non-uniform distance using (a) PWL and (b) SOI

C. *Multiplier*

The workload handled by neural networks is very different than that of conventional processors. As the neural networks deal with massively parallel operations, we need thousands of processor cores to perform the same task at a time. To appreciate the design parameters for neural network processors, we need to understand various types of hierarchically organized, data driven deep learning neural network architectures [29], [30], [31].

Multiplication and addition are the most frequently used operations in neural computations. Among these, multiplication is considered as the most complicated operation as it takes several clock cycles to complete. To speed-up the computation time required for multiplication, parallel multipliers were introduced with an additional overhead of complexity. Considering Multi Layer Perceptron, there are about 10-20M multiplications required. Thousands of multipliers would be required to implement such large number of operations. As the number of parallel multipliers increases, it would be inefficient to use parallel multipliers in massively parallel environment. Multiplication is area, power and time consuming operation, therefore special care must be taken for the design of multiplier. We have studied the performance of both serial and parallel multipliers in such massively parallel environment, result of which is shown in Figure 9. As the number of operations cross 30, serial multiplier will perform better in terms of speed and Silicon area.

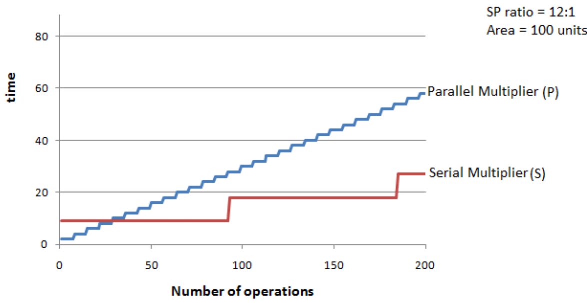


Figure 9. Performance comparison of Serial and Parallel multiplier in massively parallel environment.

As the performance of serial multiplier is better than the parallel multiplier, we have considered serial multiplier for all our further implementations. The implementation details of serial parallel are available in our earlier publication [31]. It is noticed that, for N-bit multiplication serial multiplier will require N+1 clock cycles and the accuracy of up-to 3 fractional digits is obtained with the 16-bit number format. Many researchers are working on the design of multipliers for neural network hardware [33], [34]. Lotric et al. have designed approximate multiplier for use in feed forward networks. The exact multipliers require large resources and consume more power and time. It is reported that, the exact multipliers can be replaced by approximate multipliers to increase the energy efficiency.

D. Multi-Operand Adder

Hardware implementation of neural network operations is inefficient on conventional processors. When massively parallel operations of DNNs are implemented using conventional processors, they require large number of memory transfers that tend to clog the buses and consume a lot of power and computation time. Most of the modern neural network implementations use Graphic Processor Units (GPUs) to speed-up the computation time. A typical neuron in an ANN, will have N-inputs, which are multiplied by the synapse weight, followed by an adder and an activation function. So to generalize, for N-input neuron, N-1 number of additions would be required. As it is mentioned in the earlier sections neural networks are data-intensive and involve massively parallel operations on a huge amount of data. Therefore, increase in the number of inputs will in-turn increase the number of operations.

The traditional processor architectures, involve two-operand instructions. In order to speed-up the computation, it would be appropriate to increase the number of operands for each instruction. The number of operands required for an operation will depend on the type of the neural network and the end application for which it is being used. It is necessary to identify such operations and make a provision in the instruction for flexible number of operands. For example, MNIST image recognition using ARN [35] has 49 inputs in the first layer and 16 inputs in the second layer. Therefore, a 16 and 49-operand adder would be required for that implementation. Alexnet [23] has 363 inputs in the first layer, and therefore 363-operand adder will be required for application built using Alexnet. Therefore, the number of operands in a network will vary depending on the requirement of the application.

There are some important observations need to be noted in the design of multi-operand adder; (a) Number of bits required to represent sum and carry, (b) computation time and area optimization, (c) number of operands, (d) complexity of the implementation etc. A 2-operand, N-bit adder will produce 1-bit carry and N-bit sum. However, this will not hold good for multi-operand addition. We need to look at, how the multi-operand addition will affect the sum and carry bits as compared to the 2-operand addition. The following theorem will give the details on this.

Theorem: *An upper bound on value of the carry is numerically equal to the number of operands minus one, irrespective of the number of digits or the number system used; i.e., if there are N operands, the upper bound on the value of carry is N-1.*

For example, the upper bound on carry for 4-operand addition is 3, which can be represented using 2-bits in binary (11_2). For 7-operand addition, carry is 6 (110_2) and so on. This upper bound holds for all number systems and number of operands. Some examples of N-operand addition are shown in Figure 10. Formal proof of the theorem will be published elsewhere.

Bin	Dec	Hex		
111	999	FFF	N=3	C=2
111	999	FFF		
+ 111	+999	+FFF		
10 101	2 997	2 FFD		
100 011	4 995	4 FFB	N=5	C=4
111 000	7 992	7 FF8	N=8	C=7
111 111	8 991	8 FF7	N=9	C=8
1110 000	15 984	F FF0	N=16	C=15

Figure 10. An illustrative examples of multi-operand addition.

The upper bound on the carry is given in Figure 10, N represents the number of operand and C represents the carry. Based on this theorem, a basic module of 4x4 adder designed. Further, this module can be used to design for any operand addition without much difficulty. In our earlier publication we have presented the design of 16x16 adder using this module [36].

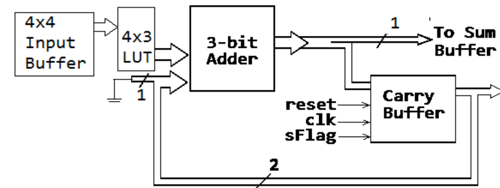


Figure 11. 4-Operand, 4-bit adder block.

Figure 11 shows basic working of a 4bit 4 operand adder which constitutes the basic building block for implementation of adders with more number of operands. As illustrated in Figure 10, a look-up table for 1 bit 4 operand adder is built and stored in 4x3 LUT. LSB of the LUT represents the column sum and other two bits represent carry to higher columns. These units can be arranged in a daisy chain to add multiple operands. One such implementation for 16 bit 16 operand adder is shown in Figure 12. Both of these adder blocks have been implemented in Xilinx verilog.

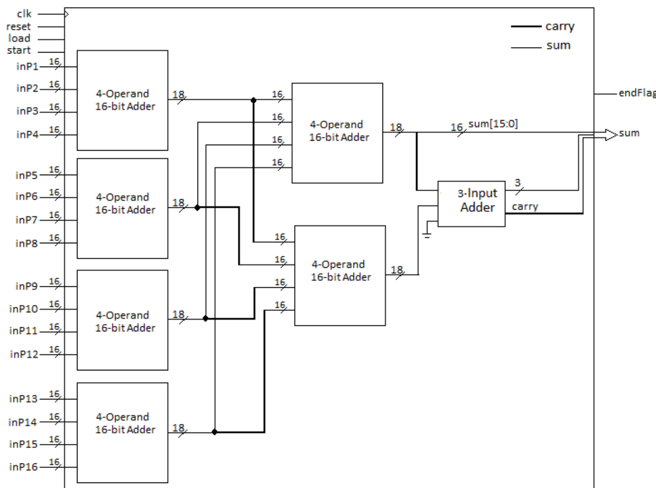


Figure 12. 16-Operand, 16-bit adder block.

IV. Conclusion

Hardware implementation of DNNs will pose multiple design challenges. They are massively parallel and their implementation is best done with data-flow architectures rather than conventional program flow. As the number of possible parallel computation is very large, serial execution would be extremely inefficient. Many modifications to instruction set are being made to bring the performance gap between conventional processors and massively parallel GPUs used to implement DL structures. The necessity to build special purpose hardware and challenges therein is discussed in section II. We have implemented basic building blocks for Deep Neural Network architectures, details of which have been presented in this paper.

Acknowledgment

The authors would like to thank C-Quad Research, Belagavi, Karnataka, India for all the support provided.

References

[1] John Nickolls, William J Dally, “The GPU computing era.”, *IEEE Micro*, 30(2), 56-69, DOI 10.1109/MM.2010.41 (March-April 2010)

[2] Shuai Che, Jiayuan Meng, Jeremy W Sheaffer, Kevin Skadron, “A performance study of general purpose applications on graphics processors.”, *Journal of Parallel and Distributed Computing*, 68(10), 1370-1380 DOI (Oct. 2008).

[3] Jose R Herrero, “Special issue: GPU computing.”, *Concurrency and Computation: Practice and Experience*, 23(7), 667-668 (2011)

[4] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, David Glasco, “GPUs and the future of parallel computing.”, *IEEE Micro*, 31(5), 7-17, (Oct 2011)

[5] Robert Elliott and Mark O’Connor, “ Optimizing Machine Learning Workloads on Power-efficient Devices.”, *White paper*, ARM, (2018).

[6] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agarwal, Raminder Bajwa et.al. “In-Datacenter performance analysis of a Tensor Processing Unit.”, *In 44th International Symposium on Computer Architecture (ISCA)*, pp. 1-12, ACM, Toronto, ON, Canada (June 2017)

[7] Urs Koster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun Bansal, William Constable, et.al, “Flexpoint: An adaptive numerical format for efficient training of Deep Neural Networks.”, *arXiv:1711.02213v2 [cs.LG], Cornell University*, (Dec. 2017).

[8] Shaoli Liu, Zidong Du, Jinhua Tao, Dong Han, Tao Luo, Yuan Xie, et.al, “Cambricon: An Instruction Set Architecture for neural networks.” *ACM/IEEE 43rd Annual International Symposium on Computer Architecture,* pp. 393-405, (2016).

[9] Filipp Akopyan, Jun Sawada, Andrew Cassidy, Rodrigo Alvarez-Icaza, John Arthur, Paul Merolla et.al., “TrueNorth: Design and tool flow of a 65mW 1 million neuron programmable neurosynaptic chip.”, *IEEE Transactions on Computer Aided Design of Intergrated Circuits and Systems*, 34(10), 1537-1557, DOI: 10.1109/ TCAD.2015.2474396, (Oct. 2015).

[10] V M Aparanji, Uday Wali and R Aparna, “A Novel Neural Network Structure for Motion Control in Joints.”, *In 1st International Conference on Electronics, Communication, Computer Technologies and Optimization Techniques*, Mysore**,** pp. 227-232, IEEE Xplore Digital Library, (2016).

[11] Ian J. Goodfellow, Jean Pouget-Abadie, Hendi Mirza, Bing Xu, Davide Warde-Farley et.al, “Generative Adversarial Nets.” *arXiv:1406.2661vl [stat.ML]* (10 June 2014).

[12] Joel Vaughan, Agus Sudjianto, Erind Brahimi, Jie Chen and Vijayan N Nair, “Explainable Neural Networks based on Additive Index Models”, *arXiv:1806.01933v1 [stat.ML] Cornell University Library*, (June 2018).

[13] Zebin Yang, Aijun Zhang and Agus Sudjianto, “Enhancing explainability of Neural Networks through architecture constraints”, *arXiv:1901.03838v1 [stat.ML] Cornell University Librar*y, (Jan 2019).

[14] Kunihiko Fukushima, “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition unaffected by Shift in Position”, *Biological Cybernetics 36*, 193-202, Springer Verlag (1980).

[15] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikane, Kevin J Lang, “Phoneme Recognition using Time-Delay Neural Networks”, *IEEE Trans. On Acoustics, Speech and Signal Processing*. 37(3) (Mar. 1989)

[16] Sepp Hochreiter, Jurgen Schmidhuber, “Long Short-Term Memory”, *Neural Computation* 9(8):1735-1780. (1997).

[17] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph P. Turian, David Warde-Farley, Yoshua Bengio, “Theano: A CPU and GPU Math Compiler in Python.”, *Proceedings of the 9th Python in Science Conf. (SCIPY)* (2010).

[18] Yangqing Jia, Evan Shelhammer, et al, “Caffe, convolutional architecture for fast feature embedding.”, *arXiv: 1408.5093v1 [cs.CV], Cornell University*, (Jun. 2014).

[19] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene brevdo, Zhifeng Chen, Craig Citro et.al, “Tensor Flow:

Large-Scale Machine Learning on Heterogeneous Distributed Systems.", *arXiv:1603.04467v2 [cs.DC], Cornell University*, (Mar. 2016)

[20] NVidia, "NVidia Tesla V100 GPU Architecture.", *white paper, NVidia* (Aug. 2017).

[21] Andres Rodriguez, Eden S, Etay Meiri, Evarist Fomenko, Young Jin K, Haihao S, Barukh Z, "Lower numerical precision Deep Learning inference and training", *White paper, Intel AI Academy* (Jan. 2018)

[22] Shilpa Mayannavar and Uday Wali, " Hardware Implementation of an Activation Function for Neural Network Processor", *In International Conference on Electrical, Electronics, Computers, Communication, Mechanical and Computing*, Vaniyambadi, Tamilnadu, IEEE Xplore Digital Library (In press), (Jan. 2018).

[23] Alex Krizhevsky, Ilya Sutskever, Geoffrey E Hinton, "ImageNet Classification with Deep Convolutional Neural networks", *In Advances in Neural Information Processing Systems* 25, pp. 1097-1105, (2012).

[24] D. Gunning, "Explainable Artificial Intelligence (XAI)" *Defense Advanced Research Projects Agency, DARPA/I20* (DARPA, 2017).

[25] Von der Malsburg, "The What and Why of Binding: The Modeler's Perspective" *Neuro*n, 24, 95–104, Copyright ©1999 by Cell Press (Sep. 1999).

[26] Shilpa Mayannavar and Uday Wali, " Fast Implementation of Tunable ARN nodes", *In 18th International Conference on Intelligent System Design and Applications* (ISDA-2018), Vellore, Tamilnadu, Springer Verlag (In press), (Dec. 2018).

[27] Parker Hill, Babak Zamirai, Shhengshuo Lu, Yu-Wei Chao, Michael Laurenzano, Meharzad Samadi Marios Papaefthymiou, Scott Mahlke, Thomas Wenisch, Jia Deng, Lingjia Tang, Jason Mars, "Rethinking Numerical Representations for Deep Neural Networks", *arXiv:1808.02513v1 [cs.LG], Cornell Digital Library*. (Aug. 2018).

[28] Marc Ortiz, Adrian Cristal, Eduard Ayguada and Marc Casa, " Low-Precision floating-point schemes for Neural Network training", *arXiv:1804.05267v1 [CS.lg]* (April 2018).

[29] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich "Going deeper with convolutions", *arXiv:1409.4842v1 [cs.CV], Cornell University* (Sep. 2014).

[30] Min Lin, Qiang Chen, Shuicheng Yan, "Network in Network", *arXiv:1312.4400v3 [cs.NE], Cornell university*, (Mar. 2014).

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition", *arXiv:1512.03385v1 [cs.CV], Cornell University*. (Dec. 2015).

[32] Shilpa Mayannavar and Uday Wali, " Performance Comparison of Serial and Parallel Multipliers in Massively Parallel Environment", *In 3rd International Conference on Electronics, Communication, Computer Technologies and Optimization Techniques*, Mysore, IEEE Xplore Digital Library (In press), (Dec. 2018).

[33] Uros Lotric, Patricio Bulic, "Applicability of approximate multipliers in hardware neural networks", *Neurocomputing Journal Elsevier*, 96(1), 57-65 (Nov. 2012).

[34] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, Kaushik Roy, "Design of Power-Efficient Approximate Multipliers for Approximate Artificial Neural Networks", *In International Conference on Computer-Aided Design*, Austin, TX, USA, IEEE/ACM DOI: http://dx.doi.org/10.1145/2966986.2967021 (November 2016).

[35] Shilpa Mayannavar and Uday Wali, "A Noise Tolerant Auto Resonance Network for Image Recognition", *In 4th International Conference on Information, Communication and Computing Technology*, Delhi, Springer Verlag, (May 2019).

[36] Shilpa Mayannavar and Uday Wali, " Design of Hardware Accelerator for Artificial Neural Networks using Multi-Operand Adder", *In 4th International Conference on Information, Communication and Computing Technology*, Delhi, Springer Verlag, (May 2019).

Author Biographies

Shilpa Mayannavar is a Research Scholar at C-Quad Research, Belagavi, Karnataka, India. She has obtained Bachelor of Engineering in Electronics and Communication Engg. (2012) and Master of Technology in VLSI Design and Embedded System (2014) from Visvesvaraya Technological University (VTU), Belagavi. Her research interests are Processor design, Artificial Intelligence and Neural Networks.

Uday Wali is a Professor in Dept. of EEE at KLE Dr M S Sheshgiri College of Engineering. & Technology, Belagavi, Karnataka India. He has obtained Bachelor of Engineering in Electrical and Electronics Engg. from Karnataka University Dharwad (1981) and Ph.D from IIT Kharagpur (1986). He is a CEO of C-Quad Computers, Desur IT Park, Belagavi. His research areas of interest are Artificial Intelligence, Neural Networks, Cognitive Radio, Processor Design and etc.