Verilog Implementation of a Fully Connected Neural Network Layer

Mohammad Nadeem

***Abstract*—In recent years, there has been an emergence in the use of machine learning and artificial intelligence algorithms. The range of applications for these algorithms is vast and ranges from computer vision onboard drones to big data analytics. AI algorithms are computationally intensive and often are run in the cloud or on a GPU on a local computer. They also have a large energy consumption. As the algorithms become more and more computationally intensive, there arises a need to implement custom hardware to speed up the tasks and consume less energy.**

One class of AI algorithms are neural networks. Neural networks, on a basic level, consist of interconnected layers of neurons. The trend in recent years has been to increase the number of layers in a neural network. The trend will continue and the number of layers will continue to increase necessitating custom hardware for acceleration. This report largely deals with creating a layer of a neural network based in hardware.

I. Introduction

Machine learning applications have become commonplace in the recent years. With large amounts of data obtained by companies like Google, Facebook and Microsoft, the gathered data can be used to efficiently train machine learning models to make predictions and provide a more customized user experience and more customized ad targeting. Primarily, algorithms for these purposes are run in the cloud which as of now, for the most part, offer enough computational power to sufficiently run these algorithms. However, in the future this might not be the case. Google has already started targeting this issue by deploying Tensor Processing Units (TPU), a custom accelerator for machine learning, into their cloud infrastructure. Their servers were not going to keep up to demands of machine learning applications and they needed to scale their infrastructure or design their own ASIC necessitating the TPU. Amazon, on the other hand, has deployed programmable FPGAs for customer use and has many GPUs available for acceleration.

Artificial intelligence has proliferated into other spaces as well. Convolutional neural networks have become a staple of sophisticated computer vision techniques, involving hundreds of layers of neurons. Machine learning is also being applied to smart cities, smart grids and autonomous driving. In the field of autonomous driving, algorithmic predictions are time critical. In other applications of artificial intelligence, time may not a critical factor. But, in the case of autonomous cars, and other embedded applications, time becomes a critical factor. Especially, in the case of autonomous driving, a small time difference can determine life or death. Thus, hardware acceleration becomes a necessity.

Machine learning has also found a home in robotics. Reinforcement learning algorithms, especially, are very useful in robotics for motor (movement) control among other applications. Deep learning is also very prominent. In the case of robotics, energy is a large issue. Thus, we can not necessarily run intensive AI algorithms on these systems due to energy constraints. An energy efficient solution for neural network computation would enable more advanced robotics research. A simple solution to enable better algorithmic performance and reduce energy consumption in robotics applications would be to communicate with the cloud and run processes there. Another solution involves creating custom hardware modeled in Verilog for synthesis and then either fabricate the chip or deploy the design on an FPGA.

The solution presented in this report will be a layer of a neural network modeled in hardware. This solution should allow faster computation for neural network based algorithms. The layer can either be deployed onto an FPGA for use in robotics or be fabricated into a chip as well.

A. Neural Networks

Neural networks are methods of doing machine learning. They are inspired by neurons within the human brain and the connections between them. In human bodies, our nervous system is composed of neurons connected together. Neurons communicate with each other through electric signals. At rest, the electric potential of the neuron is called resting potential. Then, the potential increases and the neuron fires. Similarly, a neuron in a neural network is an artificial neuron. An artificial neuron is a mathematical function, known as an activation function. The activation function has an input of vector x. There might also be a bias which will be added to the output. An activation function could be linear or non-linear (sigmoid, ReLU, etc.).

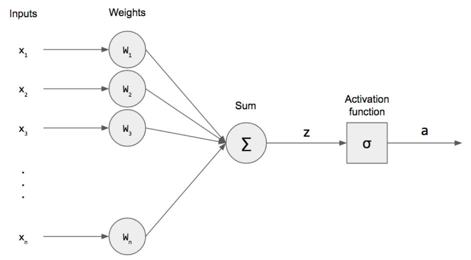

Fig. 1: A diagram of how a perceptron works.

There are many different kinds of neural networks. One type of neural network is known as a multi-layer perceptron. A multi-layer perceptron is composed of perceptrons. Perceptrons are a rather simplistic neuron. Perceptrons sums the inputs and applies the activation function to this sum and feeds it to the output. Fig. 1 shows the workings of the perceptron. Our hardware accelerator will target a perceptron. A fully connected neural network is one in which all of the outputs of the previous layer are fed into all of the neurons of the next as seen in Fig. 2. Thus, the input to a neuron can be written as an input vector x. The mathematical equation for a neuron in a fully connected layer can be written as: $y = \sigma(xW)$ where σ is the activation function and xW is the product of the input vector and weight matrix.

Another type of neural network is the convolutional neural network (CNN). A CNN is similar to a multi-layer perceptron except that some layers are convolutional layers. A convolutional layer is a layer which convolutes the input with a convolutional matrix. CNNs are extremely useful in computer vision. The final type of neural network that we will discuss is the recurrent neural network (RNN). In RNNs, the output of some layer is fed into the input of the layer before it.

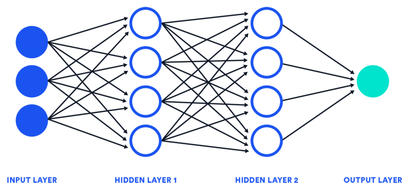

Fig. 2: An example of a fully connected neural network.

II. Related Work

Much work has been done in the previous years in terms of hardware acceleration for machine learning. Google has designed a tensor processing unit (TPU) to speed machine learning calculations for deployment in servers [1]. The main component of the TPU is a systolic matrix multiply unit (MMU). The MMU contains 256x256 multiplier-accumulators for 8 bit numbers. The products are 16 bits and are collected in the accumulators. Weights are stored in off chip 8 GiB DRAM and are staged into a FIFO buffer. The TPU uses a CISC architecture and a 4 stage pipeline.

Song Han, et. al, of Stanford University, have worked on an accelerator named EIE (Efficient Inference Engine) which has support for matrix sparsity [2]. First, the deep neural network (DNN) is compressed via deep compression. The sparse weight matrix is then represented by two vectors. One vector stores the weights of the matrix while the other vector stores the amount of zeroes occurring before a matrix element. This is known as compressed sparse column (CSC) format. Architecture wise, the EIE is composed of an array of processing elements (PE). Each PE has an activation queue for load balancing. The PE uses two pointers to read the sparse matrix inputs. The arithmetic unit of the EIE does a multiply-accumulate operation. It is found to be much faster than a CPU and GPU. The EIE uses 16 bit fixed point numbers to help save energy. The accuracy loss was not significant; there was only a 0.5% loss in accuracy.

Chen, et. al., a team composed of researchers from MIT and NVIDIA Research, have designed a spatial architecture known as Eyeriss [3]. They introduce a novel dataflow known as row stationary (RS). The RS dataflow basically simplifies convolution operations into 1D convolutions. These 1D convolutions can then be run in parallel on each processing element. The 1D convolution is first logically mapped to a logical PE. The convolutions are then mapped to physical PEs.

Deng, et. al, have proposed PermDNN which is an efficient compressed DNN architecture [4]. PermDNN uses permuted diagonal matrices. The non-zero entries are located in the diagonals or permuted diagonals. Forward propagation was used for inference while back propagation was used for training. Architecture wise, PermDNN is an array of PEs. The array of PEs enables parallel computation. Each PE has it's own SRAM for use. Each PE performs column wise processing of the matrix. The architecture supports ReLU and hypertangent activation functions.

Chen, et. al, have worked on a machine learning supercomputer known as DaDianNao which is composed of distributed mesh of nodes with each node having a lot of storage and having neural computational units called Neural Functional Units (NFUs) [5]. The amount of storage is sufficient to store the entire neural network. The synapses will be stored close to neurons to speed the process and use less energy. They adopted a tile based design such with an NFU in each tile. The NFU itself is composed of multiple blocks like an adder block and multiplier block. Similar work has been done by the same group on the DianNao [6]. The DianNao is an accelerator for machine learning. It also incorporates an NFU. The NFU is composed of a staggered pipeline of arithmetic operations (add, multiply, etc.). They use 16-bit fixed point arithmetic operators and point out that even 8 bits or less do not affect the accuracy of neural networks all too much. They split the storage into three parts: the input buffer which they call NBin, the output buffer which they call NBout and the synapse buffer which they abbreviate as SB. The NBin buffer is implemented as a circular buffer. Similar work has also been done on the ShiDianNao, an accelerator for computer vision applications which incorporates an NFU as well [7]. The NFU for the ShiDianNao is a 2D mesh of processing elements (PEs). Each PE is either mapped to an output neuron or it is shared between the input neurons. The PE is capable of addition, multiplication, etc. The storage is similar to the DianNao.

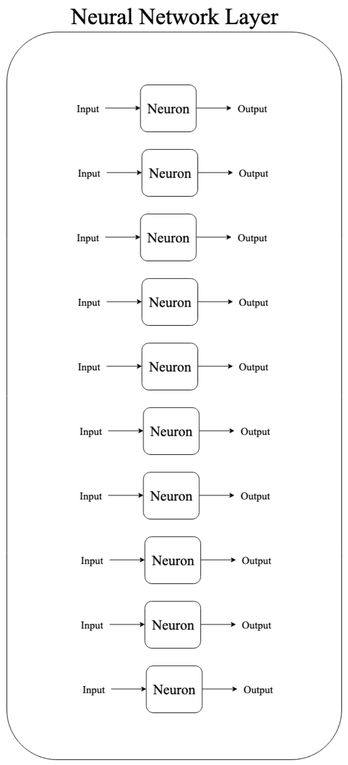

Fig. 3: High level model of the hardware implementation of a neural network layer.

III. Solution

A. RTL Behavior and Modeling

We will model the behavior of the neural network layer in Verilog. We will then run a self-checking testbench to verify the functionality of the hardware. We are not concerned with place and route or synthesis for this report.

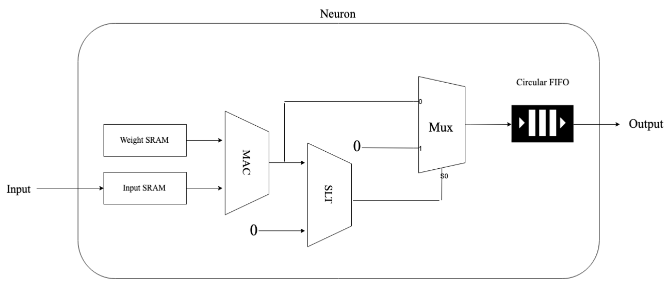

Fig. 4: Schematic of hardware implementation of neuron.

B. *Method and Model Description*

The top level design of the neural network layer involves neurons in parallel with each other as seen in Fig. 3. The spatial parallelism afforded by this design allows us to: a) scale our hardware easily and b) compute the output of a neural network quickly. Each neuron has an input and an output which is discussed below. The number representation chosen was a 16-bit fixed point number. This representation allows faster computations as opposed to floating point numbers. The accuracy loss of using 16-bit fixed point numbers is not significant as mentioned previously. Another benefit of using 16-bit fixed point representation is that all arithmetic operations for integers work the same on fixed point representations. Floating point numbers would not afford us this advantage.

The basic equation of a fully connected neural network is $y = \sigma(xW)$ where σ is the activation function and xW is the weight-input product. In our hardware implementation, each neuron has an input and an output. The output is a 32-bit fixed point number which is the output of the activation function. The input is 8 16-bit fixed point numbers. We store the inputs into SRAM which is unique to each neuron and read from this SRAM. The SRAM can be updated as the computation is occurring saving cycles to write to the SRAM. The weights are also stored in SRAM. Each neuron has it's own SRAM to store weights located near it. This will allow faster times because the SRAM is part of the neuron. The weight input product can be easily calculated by a multiplier accumulator. We will not need to worry about the bias since we can think of the bias as an input with a weight of 1 [2]. The output of this multiplier accumulator will have to go through an activation function. The most popular is ReLU. We will use ReLU as the activation function. To implement ReLU in hardware, we check if the output of the multiplier accumulator is less than 0. The output of this strictly less than operation will either be 0 or 1. We can pipe this output to a multiplexer. If the output is 0, we will choose the output of the multiplier accumulator. Otherwise, we will choose 0 to be the output of the neuron. The outputs of the neuron are written to a circular FIFO buffer which can be read from. The design of the neuron is shown in Fig. 4.

C. *Data Description*

The data we will be using to test the neural network layer is a random collection of inputs to make sure we obtain the desired output. We will make sure that the multiplier accumulator operation is working correctly and that the ReLU implementation is working as expected as well.

IV. Experimental Procedure and Results

A. *Experimental Setup*

In order to test the functionality of the hardware, we will use a test-bench to verify that the outputs are as expected as per the input. We will measure the number of clock cycles taken for each operation to give a measure of performance of the accelerator.

B. *Results*

For each neuron, the multiplier accumulator takes 9 clock cycles for completing the operation. The ReLU output takes another two clock cycles and then writing to the buffer and outputting the value takes another two clock cycles resulting in a total of 13 clock cycles for an output of a neuron to be generated. For a 1 MHz clock, we can generate approximately, 76,000 outputs per second. If we increase the size of the inputs from 8 16-bit inputs to about 50 16-bit inputs and 50 16-bit weights, we will need 55 clock cycles for an output to be generated. For a 1 MHz clock, we can still generate about 18,000 outputs per second. For a neural network with 100,000 neurons, we can generate a result in about 5-6 seconds which is rather quick.

V. Conclusion

The implementation of the neural network in Verilog works as expected and should speed up computations for neural networks. If we fallback to using shared SRAM to store weights and inputs, we can implement the design on an FPGA. Alternatively, we can custom print the board for use as well.

For the future, we can actually synthesize the hardware and place and route to see how much space the hardware takes up and how much power it will use. In the future, we can scale the hardware to more than one layer and enable our hardware to support convolution as well. Convolution is used a lot in robotics for computer vision applications and acceleration of the computation would be very beneficial. Also, we can enable the hardware to support other activation functions as well like hypertangent or sigmoid among others.

References

[1] 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 *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[2] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[3] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[4] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, "Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 189–202.

[5] 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.

[6] 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," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[7] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3. ACM, 2015, pp. 92–104.