Neural Network Implementation on FPGA

Hongxin Kong
Department of Electrical and Computer Engineering
Texas A&M University
College Station, USA
konghongxin911@tamu.edu

Lang Feng
Department of Electrical and Computer Engineering
Texas A&M University
College Station, USA
flwave@tamu.edu

***Abstract*—Nowadays, various of machine learning approaches emerged due to the sufficient hardware and software performance. Neural network is the most prevalent topic in machine learning. Benefitted from modern computer architectures, its powerful structure equipped it high capability in the field of regression and classification. However, due to the low difficulty of implementation, software-based neural networks are still used more frequent than hardware-based. In this paper, we try to propose a hardware-based neural network architecture for training and inference. The proposed architecture is highly configurable and implemented on field programmable gate array (FPGA).**

***Index Terms*—neural network, hardware, FPGA**

I. Introduction

Recent years, with high performance hardware and software emerging, machine learning approaches become more practical to implement. A typical area of machine learning, neural network, becomes a popular approach for doing various of machine learning tasks. With the demand of high performance and low overhead neural networks, sometimes software-based neural networks cannot satisfy the requirements, thus, many researches focus on implementing neural networks by hardware [1].

Neural networks are networks with neurons and connections. One each of the connection, there is weight. The input data will propagate from the input to all the neurons with the weights and activation functions. By doing this, a nonlinear mapping from the input data to the output data is realized, thus, it can be used as classification or regression. There are mainly two parts in neural networks. One is reference, which means forward propagation of the input. When a neural network is well built, users can do reference to get their results. Another is training. During training, the weights of the neural networks will converge to the values that make the neural networks perform the similar pattern as the given learning reference outputs. This is also called learning. Benefit from this property, when users give the neural networks enough inputs and corresponding correct outputs, neural networks can automatically learn the pattern by updating weights using backpropagation.

Field-programmable gate array (FPGA) is a prevalent platform for hardware implementation. Its agility property gives ditital hardware designers chances to emulate their designs before tapeout. Moreover, due to the convenience for updating, some designs even use FPGA as a target platform for implementations. In our work, we will use FPGA as a platform for verify our ideas. Theoretically, our design can also be manufactured as application-specific integrated circuit (ASIC).

Neural networks on hardware have been explored long time ago, but it is still not widly used. Neural networks which are based on FPGA are also proposed by many researchers [2]. Some works like [3, 4] proposed FPGA-based neural network implementations and have advantages over normal software-based implementations. However, there are still many spaces for improving current FPGA-based neural network architectures. We plan to improve the neural network architecture based on the work [3].

In the conclusion, our work has the following contributions:

- Proposed a real system that can fully finish the training tasks of the fully connected neural networks.
- A detailed implementation of hardware neural networks with configurable layers and neurons.

In the following sections, we first discuss some previous works in Section II, then introduce the fully connected neural networks used in our work in Section III. The system platform is proposed in Section IV, and the FPGA-based neural network architecture is proposed in Section V. We will then discuss the detailed implementation details in Section VI. Finally, Section VII and Section VIII show the experiment results and conclusions.

II. Previous works

Neural networks were proposed long time ago [5, 6, 7], but they are not widely used because of the limited computing resources. Neural networks need great amount of memory and high performance parallel computing resources which are not usually valid in the last century.

In this century, with the increased computer performance and GPU computing, neural networks almost master the area of artificial intelligence and machine learning. More and more neural networks structures are proposed, like convolutional neural networks [8] and generative adversarial networks [9]. It is shown that general central processing unit is no longer suitable for neural network operations, but highly paralleled computing, like GPU, can benefit more to neural networks.

Furthermore, researchers have explored to use specific hardware for neural networks, like tensor processing unit [10, 11] and spike neural networks [12]. The hardware based neural networks show great performance comparing to regular software based neural networks.

Field programmable gate array (FPGA) is a modern platform for hardware design. Its high configurability enable engineers or researchers quickly implement their hardware designs. Works related with FPGA-based neural networks have been proposed [3, 4]. However, due to the FPGA resource limitation, previous works usually implement a fixed trained neural networks with only inference function. This will degrade the value of FPGA and make the design impractical. Therefore, in our work, unlike [3], we implement a configurable fully connected neural network that can be applied with configurable hyperparameters. Meanwhile, our neural network can also be trained in FPGA.

III. Fully Connected Neural Network

In this section, we briefly introduce the reference and training of the fully connected neural network. A fully connected neural network is composed with neurons of many layers. Assume the input layer is layer 0, then the output of each neurons in layer i (i>0) will connect to the inputs of all the neurons in layer i+1.

For reference, for neuron i, it will first multiply each input with the corresponding weight, add all the results, and input the added result into an activation function. In this paper, we use sigmoid function as our activation function. The overall equation for reference is:

$$y = \phi(\Sigma_j(x_i w_{ij})) \tag{1}$$

Where j is the index of the neuron in the previous layer, and w_{ij} is the weight from neuron j to neuron i. ϕ is the sigmoid function: $\phi(v) = \frac{1}{1+exp(-v)}$.

For training, it will be more complex. The training of fully connected neural network follows the backpropogation of the gradient decent. for each neuron j, the weight change will always follow:

$$\Delta w_{ji} = \eta \delta_j x_i \tag{2}$$

Where j is the neuron index of the next layer, η is the gradient decent ratio, and δ_j is:

$$\delta_j = y_j(1 - y_j) \times error \tag{3}$$

The $\phi\prime$ is the difference of the active function, and the error is:

$$error = \Sigma_k(\delta_k w_k j) \tag{4}$$

Where k is the index of neuron j's next layer's neuron. If neuron j is at the output layer, the error is:

$$error = -(d_j - y_j) \tag{5}$$

Where d_j is the correct output of neuron j.

By using the equations above, we can update the weight and realize the training.

IV. System platform

Our proposed neural network architecture is implemented in an SoC platform shown in Figure 1. Due to the limited memory size of FPGA, additional processor with Linux system is used for transferring data. To train a neural network, the system will follow these steps:

- Linux system allocates a part of memory in the kernel.
- Processor loads the training set and testing set from files to allocated memory.
- FPGA reads the training set's data by the physical address through AXI bus.
- FPGA trains the neural network.
- FPGA reads the testing set's data, does inference, and returns the result to the memory.

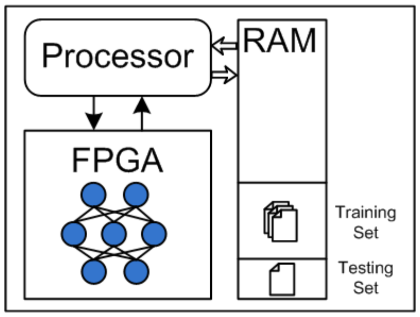


Fig. 1. System platform.

V. FPGA-based Neural Network Architecture

Given a neural network structure, theoretically it can always be implemented by hardware logical circuits. However, each FPGA only has limited hardware resources. Some previous works, like [3], only implement a specific neural network structure for one kind of use. For our work, we propose a neural network structure with configurable hyperparameters that can be trained given the dataset.

For each hardware-based neuron, it should have two parts: reference and training. Therefore, we design the neuron as shown in Figure 2.

In Figure 2, circles with $\times$ means multiply, circles with $+$ means add, x is the input of the neuron, w is the weight, η is the gradent decent ratio, Δw is the difference of the weight, and δ is the δ in Equation 3.

Besides the architecture for the neuron, we also need an global architecture for network. We illustrate it in Figure 3.

The rectangles with "Nx" in Figure 3 represent the neurons illustrated in Figure 2. Moreover, except controller and RAM, other rectangles represent the buffer for specific data. The RAM is the memory of the processor, which can be accessed by AXI bus from FPGA side.

The number of the neuron instances, m, can be configured by user before synthesis, according to the hardware resources of the FPGA. However, the number of neuron instances will not affect the training and reference of the target neural

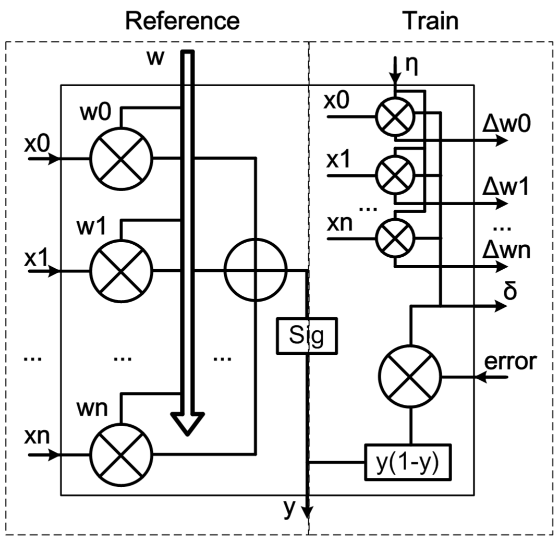


Fig. 2. The architecture of one neuron in the FPGA.

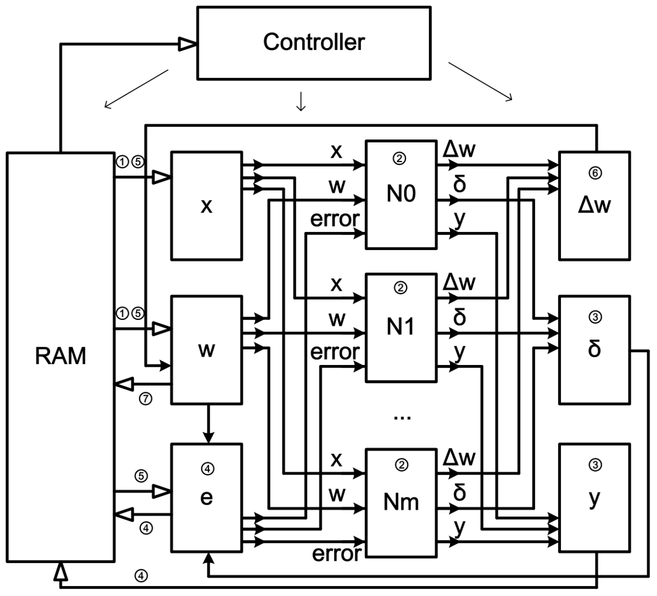


Fig. 3. The global architecture of the FPGA design.

network structure required by the user. We will get into detailed steps of the architecture.

The controller is in charge of all the processes in the FPGA. When the training data is ready, the controller will fetch the information from RAM. The fetched information contains the target neural network structure (number of neurons, layers, etc). According to the target neural network, the data will be read from the RAM and trained.

When we do the reference for one layer, FPGA will perform as the following steps:

- 1. FPGA will read the input data and weight from RAM into the buffer.
- 2. FPGA will choose at most m neurons which haven't done reference in this layer, and input the data and weights to each neuron instance that stands for a neuron in the layer. Each neuron instance will then output the results.
- 3. The results will be stored in the buffers.
- 4. The output of each neuron, y, will be used to calculate the error of the neuron along with weights. The error and output will be stored in the RAM.
- Do the above steps until all the neurons in this layer are referenced. Then move to next layer.

If we need to train the neural network, we need first do the reference. After that, the errors are stored in the RAM. FPGA will do the following for at most m neurons in one layer:

- 5. FPGA will load the recorded data, weights and errors to the buffers.
- 6. FPGA will calculate the differences of weights.
- 7. The results will be used to update the weights, and the weights will be stored back to RAM.
- Do the above steps until all the neurons in this layer are trained. Then move to previous layer.

Note that although we use limited number of neuron instances (in the case above, we use m instances), the instances can change their roles to many neurons in the user defined neural networks. This means, the proposed architecture can realize the training of fully connected neural networks with any kinds of hyperparameters.

VI. Implementation

We use Intel DE1-SoC board for implementation. In this board, it contains an ARM processor and an FPGA.

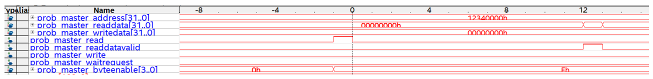
A. Data Transferring

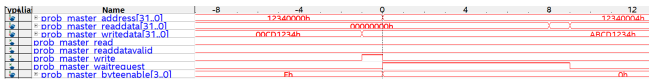
FPGA has limited memory, thus, we use the memory (DRAM) of the ARM processor, which is 1GB. In the Linux system running on the ARM processor, we allocate 512MB memory for data transferring between DRAM and FPGA.

To realize this, we have to use kernel memory but not user memory, since we cannot get the physical address of the user memory directly. The first step to do this is to write a kernel module. We need the source code of the Linux system for DE1-SoC, which can be found in [13]. In the kernel module, we need to use "kmalloc" to allocate 512MB memory and display the physical address of it, which can be realized in kernel module. The start address of our case is 0x0e800000.

After the memory is allocated, we use the first 128MB for training and testing data, the next 256MB for weights, and the last 128MB for other information. Before we start training, the training and testing data will be loaded beforehand, by a user level program. This user level program will map the memory starting from 0x0e800000 to a pointer, then read the dataset and write the data into the first 128MB.

On FPGA side, FPGA will access the memory through the physical address. It will need AXI bus to access. Since we use Intel FPGA, we can create a system platform by using Platform Designer in Quartus 17.1. Benefit from the build-in modules, we can first build the complete ARM to FPGA basic system and then manually split the bus between JTAG and AXI in the source code, and thus, we can use the same way that JTAG uses to control the AXI bus interface. The waveform of read from and write to AXI bus is shown in Figure 4 and Figure 5. As we can see, it needs about 10 FPGA clock cycles to access the memory.


Name
-8
-4
0
4
8
12
prob_master_address[31..0]
prob_master_readdata[31..0]
prob_master_writedata[31..0]
prob_master_read
prob_master_readdatavalid
prob_master_write
prob_master_waitrequest
prob_master_byteenable[3..0]
12340000h
00000000h
00000000h
0h
Fh
Fig. 4. The waveform of reading 4 bytes through AXI bus.


Name
-8
-4
0
4
8
12
prob_master_address[31..0]
prob_master_readdata[31..0]
prob_master_writedata[31..0]
prob_master_read
prob_master_readdatavalid
prob_master_write
prob_master_waitrequest
prob_master_byteenable[3..0]
12340000h
12340004h
00000000h
00CD1234h
ABCD1234h
Fh
0h
Fig. 5. The waveform of writing 4 bytes through AXI bus.

B. FPGA-based Neural Network Implementation

1) Neuron: Since hardware circuits has no flexibility, which means their structure cannot be changed while using them, so we can only design fixed number of neurons and use them for the whole neural networks. In our design, for each neuron, we allocate 32 inputs with each of them 32-bit. This is enough for most cases. For the activation function, we use ReLu, which is very easy for hardware circuit because there is not multiplication or division. The netlist of each neuron is shown in Figure 6.

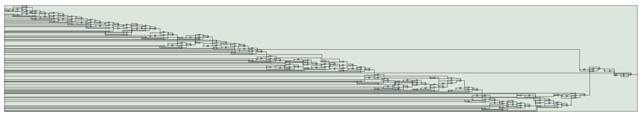


Fig. 6. The netlist of one neuron.

As we can see, the synthesized netlist has long critical path. The reason is the result of the neuron is calculated by the sum of each weighted input, where the sum is a serial process. Therefore, we cannot allocate numerous input ports due to the reason that this will incur a longer critical path and thus, incur timing error.

2) Data Format: In our implementation, each input of a neuron is 32-bit. We use 1's complement number, which means, the most significant bit indicates if the number is positive (0) or negative (1). The reminding bits are for the value of the number. The output of the neuron is:

$$y = \phi(\Sigma_j(x_i w_{ij})) = \phi(x_0 w_{00} + x_1 w_{01} + ...) \tag{6}$$

To do the application, the multiplier of our design will get 2 1's complement inputs, and output 2's complement number to do addition. After the addition, we use ReLu as the activation function, so it will transform all the positive 2's complement to 1's complement as the next layer's input, and transform all the negative number to 0.

3) Controller: Note that due to some technical reasons, we haven't implemented the design of the training part. We only implemented the inference part.

The controller is implemented by a finite state machine (FSM). As shown in Figure 7, The FSM has 6 state, which are IDLE, LOADIN, LOADW, LOADB, TRANS, FINISH. We will introduce each state in detail.

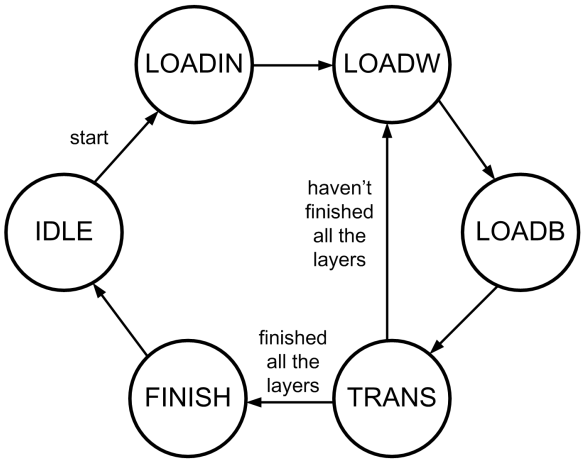


Fig. 7. The finite state machine in the controller.

The FSM will start as IDLE. In this state, the controller will wait for the start signal. Once it receives, FSM will go to LOADIN state.

In LOADIN state, the controller will load the input data into the buffer, and that is the input of the neuron instances. After this, controller will get into LOADW state and then LOADB state, which are to load the first layer's weights and biases. Note that we designed the interface between the controller and the AXI to load the data.

After all the data is loaded, controller will get into TRANS state. This state is to let data transfer inside all the neurons, get the output, and transfer the output to the input buffer. Note that the input buffer originally contains the input data, but after there is output of the first layer, the data in the input buffer is replaced by the output, which represent the input of the second layer. After this, state will change to LOADW and LOADB again, to read the second layer's weights and biases.

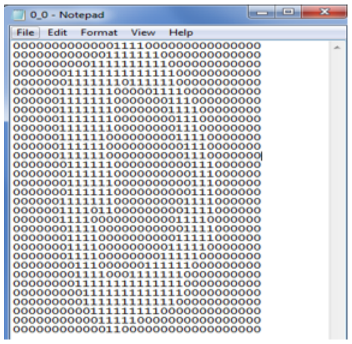
After all the layers are calculated, controller will get into FINISH state and output the final data. Since we transfer the output of the neuron to the input side, and load weights and biases layer by layer, we only need one layer of the neurons, and this can make our design flexible and user can train as many layers as they configured, only if the number of neurons in each layer is in the range of the neuron instances in FPGA.

VII. Experiments and Results

A. Experiment Setup

We use Intel DE1-SoC board to evaluate our design. The board contains an ARM Cortex-A9 processor, 1GB DRAM, and a Cyclone V FPGA. The software for implementing the hardware is Quartus, and the software for simulation is

Modelsim. The dataset for training and inference is MNIST dataset. One example of the data in the dataset is shown in Figure 8. This is the example of number "0", where white space is encoded in 0, and line is encoded in 1. The size of the figure is 32*32. When implementation, we decrease the resolution of each row by 2, which means, it changes to 32*16. This is better for hardware since the hardware has limit bandwidth. Also, this won't decrease the accuracy much. The neural network we used has 3 layers, where the number of the neurons are 32, 30 and 10.



```
00000000000000111100000000000000
00000000000001111111000000000000
00000000000111111111100000000000
00000001111111111111110000000000
00000001111111011111110000000000
00000011111110000011110000000000
00000011111110000000111000000000
00000011111110000000111100000000
00000011111110000000011100000000
00000011111110000000011100000000
00000011111100000000001111000000
00000011111100000000000111000000
00000011111100000000000111000000
00000001111110000000000011100000
00000001111110000000000011100000
00000001111110000000000011100000
00000001111110000000000011100000
00000011111110000000000111100000
00000011110110000000000111100000
00000011110000000000000111100000
00000001111000000000000111100000
00000001111000000000001111100000
00000001111000000000011111000000
00000001111000000000111110000000
00000000111000000011111100000000
00000000111100011111111000000000
00000000111111111111111000000000
00000000011111111111111000000000
00000000011111111111110000000000
00000000001111111111000000000000
00000000000001111100000000000000
00000000000000110000000000000000
```

Fig. 8. One example of the data in the dataset.

B. Netlist

In this part, we will propose the result of the netlist synthesized by Quartus. In Figure 9, it shows the netlist of the top level entity used for simulation. The database is the simulated interface that can output the data and weights. The controller is the controller we discussed in the former section. The database can be replaced by the real interface which communicated with AXI, which shown in Figure 11, which is a part of the netlist in Figure 10.

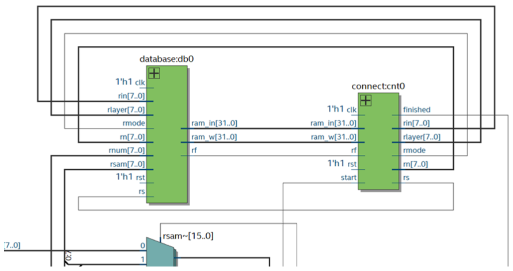


Fig. 9. The top level netlist for simulation.

In Figure 11, "read_wbin" is the real interface interact with AXI bus and read the inputs, weights and biases. When we want to implement our design in this SoC system, we can replace the database in Figure 9 with the part in Figure 11.

Another netlist results are the netlist of the controller. We first implemented our neural network structure as it is. This means, we instantiate the same number of the neurons as the hidden layer and output layer, and connect them in the fixed way. The netlist is shown in Figure 13.

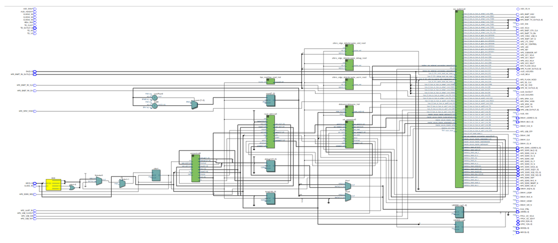


Fig. 10. The top level netlist when using real SoC system.

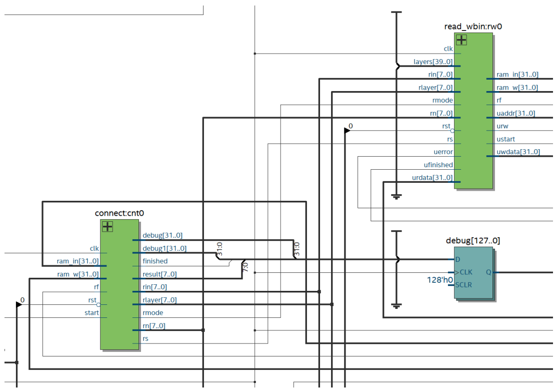


Fig. 11. A part of the top level netlist when using real SoC system.

Then the proposed design is implemented (inference part). It is shown in Figure 12. Note that the this figure is rotated left by 90 degree, so the critical path is still not long. As we can see, because we need register to store the output of each layer, the structure is more complicated than the directly mapped structure.

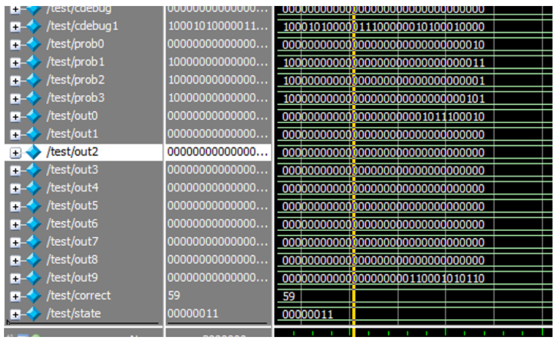
Fig. 13. The netlist of the directly mapped controller.

C. Accuracy

The most important challenge for hardware base neural network is the precision. Using hardware circuit cannot use as much bits as the user want. So usually we will lose precision. Fortunately, neural network is a kind of approximate computing, thus, sometimes losing precision is acceptable. However, we should also test how much accuracy it will lose. After trained on PC, our neural network model can achieve

Fig. 12. The netlist of the proposed controller.

64% accuracy for the 100 testing data. To compare with this, we test the same 100 data on the FPGA circuit. The result in Figure 14 shows that 59 of them are correct, which means the accuracy is 59%. The decreasing of the accuracy is due to the precision, but it is totally acceptable.



```
/test/cdebug     00000000000000...  00000000000000000000000000000000
/test/cdebug1    1000101000011...   1000101000001110000001010010000
/test/prob0      00000000000000...  00000000000000000000000000000010
/test/prob1      10000000000000...  10000000000000000000000000000011
/test/prob2      10000000000000...  10000000000000000000000000000001
/test/prob3      10000000000000...  10000000000000000000000000000101
/test/out0       00000000000000...  00000000000000000000001011100010
/test/out1       00000000000000...  00000000000000000000000000000000
/test/out2       00000000000000...  00000000000000000000000000000000
/test/out3       00000000000000...  00000000000000000000000000000000
/test/out4       00000000000000...  00000000000000000000000000000000
/test/out5       00000000000000...  00000000000000000000000000000000
/test/out6       00000000000000...  00000000000000000000000000000000
/test/out7       00000000000000...  00000000000000000000000000000000
/test/out8       00000000000000...  00000000000000000000000000000000
/test/out9       00000000000000...  00000000000000000001100010101110
/test/correct    59                 59
/test/state      00000011           00000011
```

Fig. 14. The accuracy of our design.

D. Latency

Latency is the time interval between the beginning of the inference to the end. Latency is an important metrics since it affects the performance of the neural network. Nowadays, one of the bottlenecks of neural network is the time cost. Usually, neural network requires a highly parallelized system to be deployed, which can decrease the latency. FPGA is a good platform for parallel computing. For our design, we tested the latency. Figure 15 shows the waveform of 100 data inference. It spends about 1350000ns, so it is about 6750 FPGA clock cycles for each sample, which is also an acceptable number.

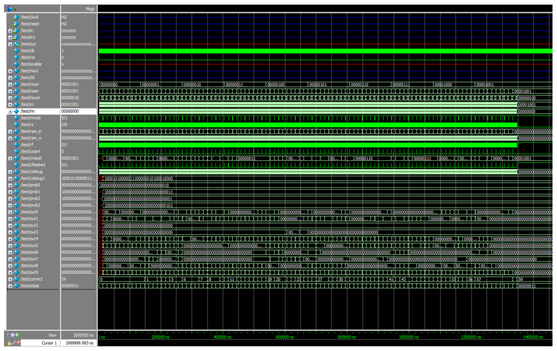


Fig. 15. The waveform of 100 data inference.

VIII. Conclusion

In our work, we deigned a configurable FPGA based neural network training architecture, and implemented the inference part on FPGA. The design can be used by a dynamically, which means the neural network structure and weights and biases do not need to be fixed. The accuracy does not decrease much and the latency is short. In the future, we plan to implement the training part of our design as well as more kinds of neural networks.

Appendices

About the division of work, Hongxin Kong trained the neural network, designed the structure of each neuron, designed the data format. Lang Feng designed the controller, designed the data transfer approach, and wrote the paper.

Attached in the top zip file, there are all source codes and samples.

There are 3 zip files.

In linux_programs.zip: database.py can generate the data for the testbench of the simulation Verilog program. NNtf.py can do training and write the weights and biases on PC. Use ”python NNtf.py s” to train and store the weights and biases. Use ”python NNtf.py l” to load the trained weights and biases and test again. trans614.c is the ARM program to transfer weights from ARM to RAM. (This program needs a file called weights) mymodule.ko is the system module to open 64MB RAM space for FPGA. writeweight.py is to write the weights and biases to the file ”weights”.

quartus_project_simulation.zip is the Verilog project of the neural network simulation. quartus_project_SoC.zip is the Verilog project of the data transferring interface with ARM.

References

[1] Y. Liao, “Neural networks in hardware: A survey.”

[2] A. R Omondi and J. C Rajapakse, *FPGA Implementations of Neural Networks*. 2006.

[3] T. V. Huynh, “Deep neural network accelerator based on fpga,” in *2017 4th NAFOSTED Conference on Information and Computer Science*, pp. 254–257, 2017.

[4] S. B. Yun, Y. J. Kim, S. S. Dong, and C. H. Lee, “Hardware implementation of neural network with expansible and reconfigurable architecture,” in *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP ’02.*, pp. 970–975 vol.2, 2002.

[5] D. Psaltis, A. Sideris, and A. A. Yamamura, “A multilayered neural network controller,” *IEEE Control Systems Magazine*, pp. 17–21, 1988.

[6] “Theory of the backpropagation neural network,” in *International 1989 Joint Conference on Neural Networks*, pp. 593–605 vol.1, 1989.

[7] D. F. Specht, “A general regression neural network,” *IEEE Transactions on Neural Networks*, pp. 568–576, 1991.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems 25*, pp. 1097–1105, 2012.

[9] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in Neural Information Processing Systems 27*, pp. 2672–2680, 2014.

[10] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[11] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, pp. 10–19, 2018.

[12] W. Maas, "Networks of spiking neurons: The third generation of neural network models," *Trans. Soc. Comput. Simul. Int.*, pp. 1659–1671, 1997.

[13] Linux source code for DE1-SoC. https://github.com/altera-opensource/linux-socfpga.