FPGA-based Convolutional Neural Network Accelerator

CSEE4840-Spring2019-Report

Ke Xu, Xingyu Hou, Manqi Yang, Wenqi Jiang

Columbia University

1 Introduction

VGG-16 is a popular convolutional neural network structure. In this project, we purpose to implement an FPGA-based accelerator for VGG-16. On the software side, we first implement the network in python and C. After analyzing the data property of input images, we decide to apply dynamic fixed-point strategy and build a set of tools for fixed-point conversions and operations. On the hardware side, the HPS communicate with FPGA through AXI and AXI Light-weight Bridge to load the data into SDRAM and give the address message and enable signal. In order to reuse the data as many times as possible, we design a dataflow of Processing Elements (PE) which contains 64 MAC units.

2 Background

2.1 Convolutional Neural Network (CNN)

Convolutional Neural Networks (CNN) is a type of neural networks and it is widely used in Computer Vision tasks such as Image Classification, Object Detection, and Semantic Segmentation.

Different from the Multi-layer-perceptron (MLP) model, in which each input is multiplied by its own weights, the weights in CNN can be shared. In CNN, the same filter (also known as 'weights' or 'kernel') moves over the input feature maps, and at each position, an output value is calculated. This means the same weights are used by the entire input, which saves memory.

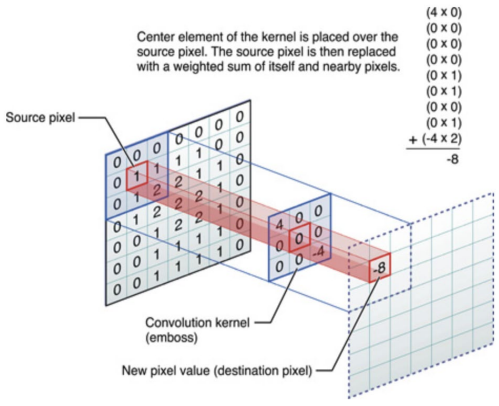

Fig 2.1 Convolution visualization

2.2 VGG16 Overview

VGG16 is a CNN model that can achieve 92.7% top-5 test accuracy. The overall structure of VGG16 is shown in Fig 2.2.

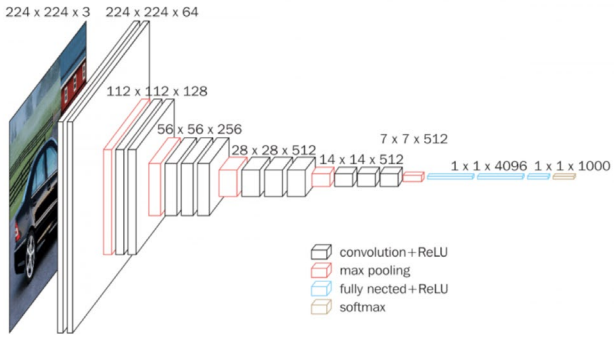

Fig 2.2 VGG16 structure overview

As shown in the figure, there are 4 types of layers: convolutional layers, max pooling layers, fully-connected layers, and softmax function. The input of the neural network is an image with a size of 224 x 224 x 3. The filters are 3×3 matrices and the stride of which is fixed to 1. The padding size is always 1, while max-pooling is performed over a 2×2 pixel window, with a stride of 2.

We also have 3 fully-connected (FC) layers. The first two has 4096 channels, while the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). Finally, there is a softmax layer, which transforms the output into a probability distribution over the 1000 classes.

2.3 Winograd Algorithm

We also considered optimizing the convolution from the mathematics perspective and one of these algorithms is Winograd. Winograd is used to reduce the multiplication times of convolutions. This algorithm is widely used for GPU accelerations for convolutional neural networks.

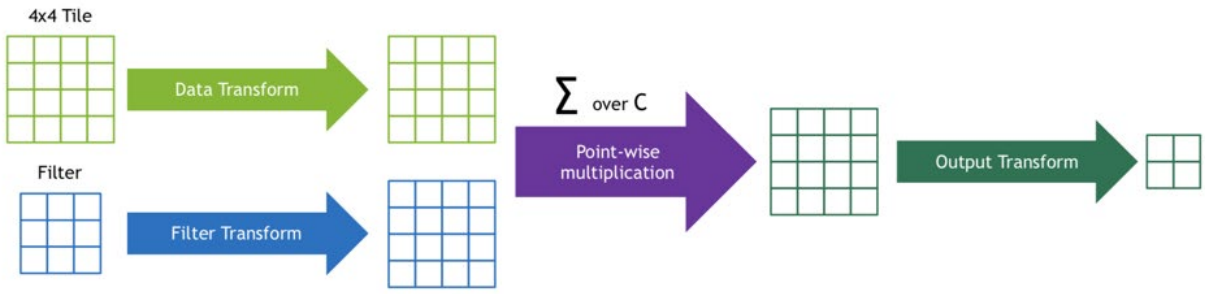

Fig 2.3 Winograd algorithm

The process of Winograd is shown in Fig 2.3. First, the input feature map and the convolution filter will be transformed into matrices of the same dimension. Then, the element-wise multiplication is used for these two matrices. Finally, another transformation will be implemented to output the result. As shown in Fig 2.3, the direct convolution will consume 36 multiplications, while Winograd only consumes 16, which is a 2.25x speedup if we only consider multiplication time.

However, Winograd is a memory consuming method. As shown in the figure, it will generate some intermediate results, which need extra space to store. In our project, due to the design of Processing Elements (PE) and dataflow, the convolutions are divided into one-dimensional array multiplications. In this case, by using Winograd, only ⅓ multiplications are reduced, while we need to spend 2 times of memory to store intermediate results. Therefore, we decide not to use Winograd in our project.

2.4 Computational Complexity and Memory Consumption

In this section, we will analyze the computational complexity and memory consumption of convolutional layers and fully-connected layers. By measuring these factors, we draw some conclusions which decides our acceleration strategy layer.

2.4.1 Computational Complexity

For computational complexity, we mainly consider the times of multiplications, which is the most time-consuming operations in our project. In convolutional layers, there are 6 factors in total: 4 for the convolution filter itself and 2 for the input feature map size. The complexity of a fully-connected layer, on the other hand, only related to the 2 dimensions of the weight matrix.

	Convolutional Layer	Fully-connected Layer
Computational Complexity	conv_height * conv_width * conv_channel * conv_number * input_width * input_height	fc_height * fc_width
Memory Consumption	conv_height * conv_width * conv_channel * conv_number	fc_height * fc_width

Table 2.1 Analysis of convolutional layers and fully-connected layers

	Convolution Layers	Fully-connected Layers	Ratio (conv / fc)
Weights number	14,710,464	123,633,664	0.12x
Multiplications number	16,271,474,688	123,633,664	131.61x

Table 2.1 Comparison between convolutional layers and fully-connected layers

2.4.2 Memory Consumption

For memory consumption, since we use fixed-point 16 to store all the data, we only need to consider the parameter numbers. In this case, the convolution layers only have 4 factors, which

are height, width, channel number, and filter number of the convolution filter. The memory consumption of fully-connected layer is same as the previous section, only related to the 2 dimensions of the weight matrix.

3 Software Implementation

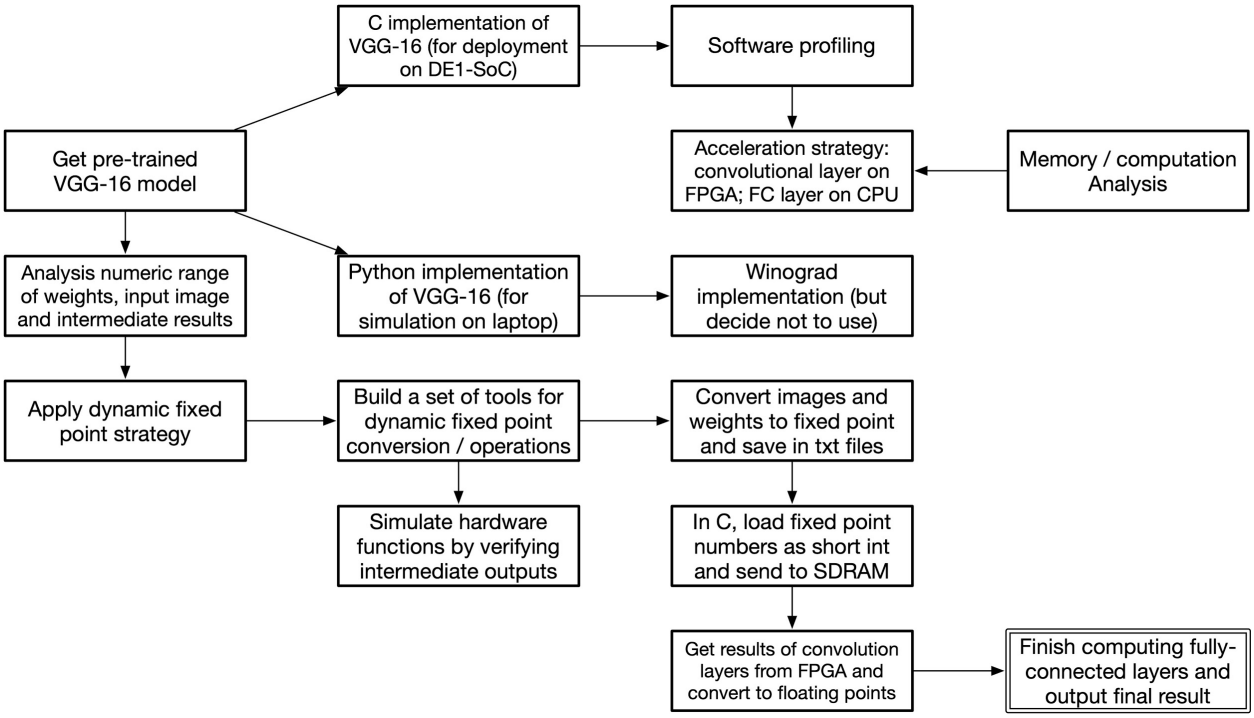

Fig 3.1 Software workflow

On the software side, we first get a pre-trained VGG16 model. Then, we implement the model by ourselves in both python and C. Then, we do software profiling to decide which part should be accelerated by FPGA. After analyzing the numeric ranges of the input images, filter weights, and intermediate outputs, we decide to use dynamic fixed-point strategy. Thus, we build a set of tools for both fixed-point conversion and operations. The fixed-point operations are for hardware function simulations. After converting input images and weights to fixed-point number, we save them into text files to be loaded in our C program. These fixed-point numbers will be feed into SDRAM. Finally, we get the results of convolutional layers from FPGA and finish the fully-connected layers in C.

3.1 Pretrained Model

We download a Keras-based VGG16 implementation and the pre-trained weights of the model. Keras is a high-level deep learning API runs on the top of TensorFlow. It can generate intermediate and final result, which will be helpful for our verification. We will feed the input into this Keras model generated with these weights, then print out the intermediate and final result for later verification.

3.2 Python / C implementation

Then we reproduce the VGG16 model using Python, including convolution layers, fully-connected layers, pooling layers and activation functions (ReLU and softmax). As said before, Keras is used for verification, we compare the result with the Keras model to verify the correctness.

Then we port the python implementation to C code so that we can run it on DE1-SoC and do data transfer with FPGA.

3.3 Software Profiling & Acceleration Strategy

After implementing VGG-16 in C, we profile the program to see the time consumption distribution over all the functions. Here, we mainly compare convolution layers and fully-connected layers, that is because max pooling and activation function are negligible in profiling, and loading data from hard drive is necessary and cannot be accelerated by FPGA.

We do the profiling using Intel i5-8259U. As shown in Table 3.1, convolution layers dominate the time consumption in VGG-16 network, while fully-connected layers only consume less than 5% of time during the whole computation process.

	Convolution Layers	Fully-connected Layers
Time Consumed / sec	92.02	4.15
Time Percentage / %	95.67	4.32

Table 3.1 Software Profiling

If we want to accelerate both convolution layers and fully-connected layers on the FPGA, some on-board resources such as DSPs should be assigned to fully-connected layers. Considering the large amount of data we are going to transfer, i.e. more than 200 MBytes which is beyond the capacity of SDRAM, we are not able to pipeline the computation process. In this case, the convolution IPs would be idle when fully-connected layers are being computed and vice versa. Also, transporting the large amount of weights of fully-connected layers to SDRAM consumes too much time, i.e. more than 30 seconds. Because these reasons, it would not be a good choice to accelerate both convolutional layers and fully-connected layers on the FPGA. Thus, the strategy we apply for our FPGA accelerator is that we only compute convolutional layers on the FPGA while using CPU to finish the fully-connected layers.

3.4 Fixed Point Computation

FPGAs are good at fixed point computation, so we want to take advantage of that. However, one of the challenges when deploying fixed point computation in our project is that the weights, input image and intermediate results have very different ranges. As shown in Fig 3.2, we can see the ranges and average values of the intermediate results. In layer 7, for example, the maximum absolute value of the results is more than 20,000, while in layer 16, 22.73 is the maximum. That is challenging because it would be hard to decide where to put the decimal point when doing fixed point computation: suppose we use 16-bit fixed point number and use one of the bits as sign, if we use all of the rest 15 bits to express integer part, then we can

express a number that is larger than 20,000; however, we will lose too much precision in those layers which have small ranges.

```
Intermediate outputs of each layer:
Layer 1:        max:    955.85339       mean:   26.134333
Layer 2:        max:    3689.4717       mean:   150.90759
Layer 3:        max:    8024.4507       mean:   207.26363
Layer 4:        max:    11275.772       mean:   229.63486
Layer 5:        max:    15204.325       mean:   284.25253
Layer 6:        max:    15991.411       mean:   329.49692
Layer 7:        max:    20796.828       mean:   199.77206
Layer 8:        max:    12113.225       mean:   174.05199
Layer 9:        max:    6186.3657       mean:   97.339676
Layer 10:       max:    2941.2952       mean:   21.802027
Layer 11:       max:    1955.9144       mean:   22.766939
Layer 12:       max:    953.95642       mean:   9.6327209
Layer 13:       max:    552.76392       mean:   1.3137941
Layer 14:       max:    41.791012       mean:   1.7962291
Layer 15:       max:    15.469688       mean:   0.58563447
Layer 16:       max:    22.734434       mean:   1.0392156
```

Fig 3.2 Ranges of intermediate results

We solve this problem by using the strategy of dynamic fixed point: in each layer we assign different digits to integer and decimal parts. The digits assignment is shown in Fig 3.3. During the fixed-point multiplication on the FPGA, we input 2 16-bit number and the result is a 32-bit number. Then, according to the digit assignments, we which 16 bits of the 32-bit results should we keep.

```
Digits allocation to integer part and decimal part:
(Use fixed point 16, with 1 digit as sign)
Input:          Max:    255.0           Integer digits: 8       Decimal digits: 7
Layer 1:        Max:    955.85339       Integer digits: 10      Decimal digits: 5
Layer 2:        Max:    3689.4717       Integer digits: 12      Decimal digits: 3
Layer 3:        Max:    8024.4507       Integer digits: 13      Decimal digits: 2
Layer 4:        Max:    11275.772       Integer digits: 14      Decimal digits: 1
Layer 5:        Max:    15204.325       Integer digits: 14      Decimal digits: 1
Layer 6:        Max:    15991.411       Integer digits: 14      Decimal digits: 1
Layer 7:        Max:    20796.828       Integer digits: 15      Decimal digits: 0
Layer 8:        Max:    12113.225       Integer digits: 14      Decimal digits: 1
Layer 9:        Max:    6186.3657       Integer digits: 13      Decimal digits: 2
Layer 10:       Max:    2941.2952       Integer digits: 12      Decimal digits: 3
Layer 11:       Max:    1955.9144       Integer digits: 11      Decimal digits: 4
Layer 12:       Max:    953.95642       Integer digits: 10      Decimal digits: 5
Layer 13:       Max:    552.76392       Integer digits: 10      Decimal digits: 5
Layer 14:       Max:    41.791012       Integer digits: 6       Decimal digits: 9
Layer 15:       Max:    15.469688       Integer digits: 4       Decimal digits: 11
Layer 16:       Max:    22.734434       Integer digits: 5       Decimal digits: 10
```

Fig 3.3 Dynamic fixed-point strategy

In this project, we need a set of tools for fixed point conversions and operations. The conversion functions used to convert model weights and input images to fixed point and feed to the FPGA. The fixed-point operation functions are used to simulate what happens in hardware functions and they are useful for hardware debugging.

We build this set of fixed-point tools in python. For conversions, they support both singular and a tensor of bidirectional fixed point to floating point conversion. For operations, they support singular or a tensor of fixed-point addition, multiplication, inverse, shift and ReLU. We also build other functions such as detecting how many digits of integer part should we allocate in a fixed-point number.

4 Hardware Design

4.1 Hardware Structure

Hardware design mainly contains three parts: PE calculation, convolution state machine and DMA controller. The structure plot shows as follow. We also need two different memory parts: SDRAM for data temporary storage and on-chip memory for calculation demand.

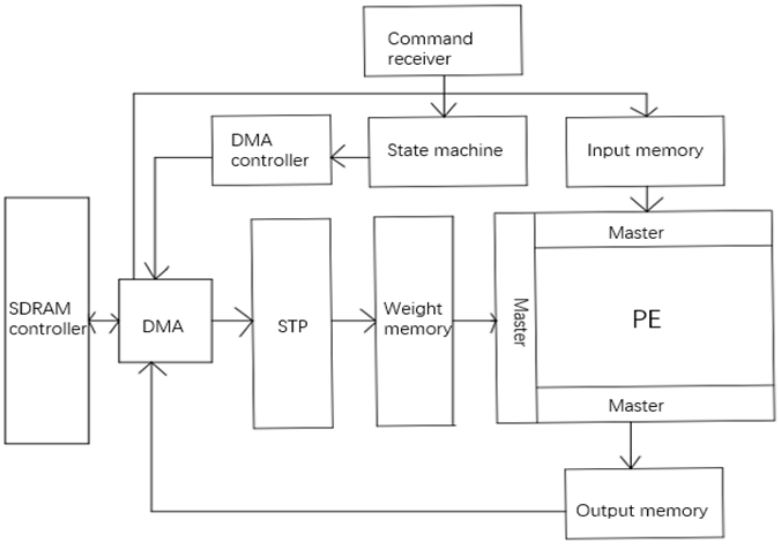

Fig 4.1 Hardware architecture

4.1.1 PE Calculation

The main advantage in FPGA is that it could work in parallel so that different jobs could be done at the same time. In order to implement the calculation for CNN with a huge amount of data, we design a special structure in PE part which contain 64 MAC units used to finish calculation with multiply and accumulation. The reason we set the PE in this way is that we want to get the result of the same point which combined with 64 layers or times of 64 layers in one time (Actually 9 times clk due to convolution core is 3x3). This means that we do not need to save the intermedia data result in On-chip memory or SDRAM, we can get the result when we put the data into PE. However, this would need the bandwidth of the On-chip memory reach 1024 bits in that each MAC unit has 16x16 bits width.

In addition, the convolution layer actually has three different kinds of data: input feature, weight for each layer and bias for each layer. This means we need three main On-chip memory regions: input feature, weight and output region. The output memory is the region we need to buffer the output after each calculation so that the final result could be transfer to SDRAM again.

4.1.2 Convolution state machine

Convolution state machine contains the upper level state flow for FPGA to calculate different layers convolution. According to command receiver information which received control signal from software part, state machine controls DMA through DMA controller to grab the data loaded in SDRAM to three different On-chip memory. The DMA is component which provided by Qsys, it works when you instruct the read address, write address and data length.

Convolution state machine also plays a role that enable the PE to calculate the data after it finishes transferring data into input and weight On-chip memory. After each time calculation, state machine also needs to write the result into output memory.

4.1.3 DMA controller

The reason we need this controller is that we want the hardware automatically steps into next stage after finishing transferring. This controller actually gives the information needed by DMA component created by Qsys. Because this DMA can only work in the situation with both read address and write address. We need to give these messages and controller it to transfer which and how many data we need in SDRAM.

We cannot only use this DMA with its original bandwidth which is 16 bits due to SDRAM. Because the PE need 1024 bits data each time, we implement a custom component in Qsys which used to transfer data from 16 bits to 1024 bits for calculation convenient. It is kinds of FIFO which could buffer the data until reach 1024 bits and then put it into On-chip memory. The same work we need to do is transferring output result from 1024 bits to 16 bits into output memory so that we build another component to finish it between PE and output On-chip memory.

4.2 Data Alignment in SDRAM

Since we need to calculate the data for a single point with 64 layers, we must arrange the data into a proper way so that we can simplify the data flow. We design two kinds of storages for weight and input feature.

As mentioned above, we have 64 MAC units in PE calculation. Thus, for each input feature point, we time it with 9 weight data due to the convolution core is 3x3. In another word, we use 64 MAC units to calculation 64 layers input feature with weight. After 9 times calculation, we can get one point of input feature output result. You can see the data arrange way in more direct way in following picture.

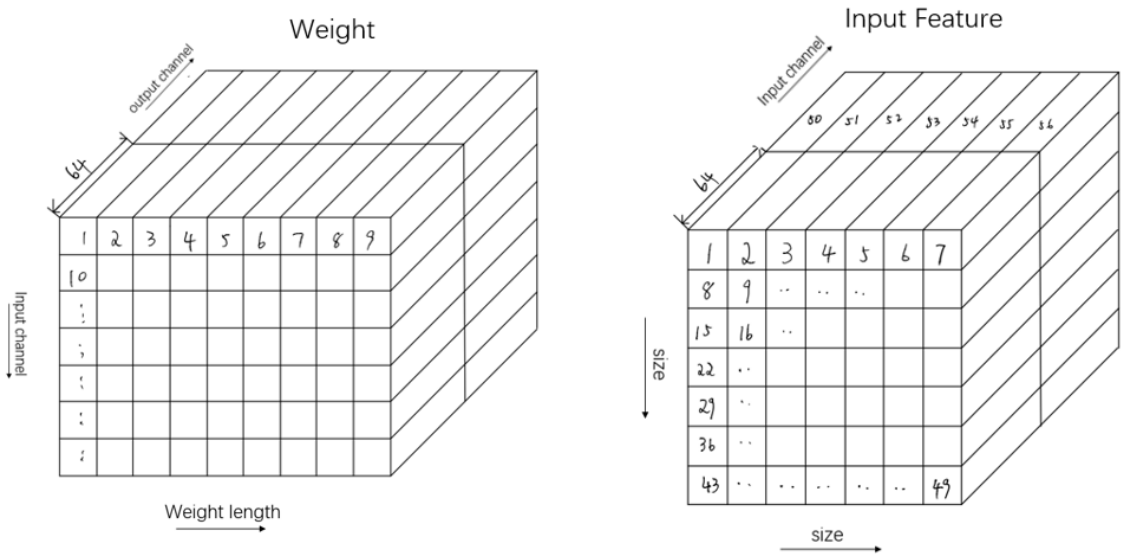

Fig 4.2 Data alignment

Each line 9 data in weight side means 3x3 core for convolution. We calculate the 64 results for each cuboid. The sum of these 9 cuboid result refer with 1 input feature and then give out 1 output feature.

We put the weight and input data in this way because DMA grab the data consecutively from SDRAM. Then, it would write these data into On-chip memory and finally to PE calculation part. This arrangement could simplify the data structure in the easiest way for later used.

4.3 Data flow

The way data used in PE is shown in Fig 4.3. The matrix contains three parameters: input channel data, output channel data (i.e. output layer data) and weight data. In fact, we load 64 layers data into PE part for one input feature point. And these amounts of data would be multiply and accumulate with 3x3 core, i.e. 9 weight data to give out a final output result which refers with one input feature. The loop would continue until the last input feature point has been calculated to output On-chip memory.

(in channel, out channel, weight)
(1,1,1)
(1,2,1)
(1,3,1)
..
(1,64,1)
(2,1,1)
(2,2,1)
..
..
(64,64,1)
(1,1,2)
(1,2,2)
..
..
(64,64,2)
..
..
(64,64,9)

Fig 4.3 Dimension specification

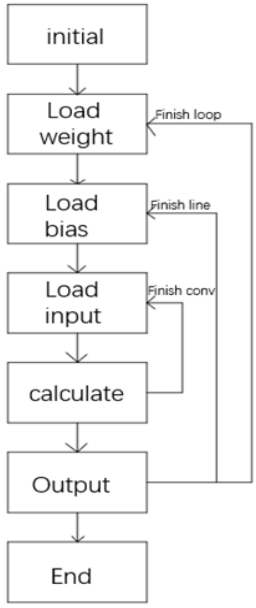

Fig 4.4 Dataflow design

As shown in Fig 4.4, after three times data transfer, the PE part can acquire all the data which need to finish calculation. The convolution state machine would start PE to work and then feedback a signal to it so that the state machine could know PE finish the job. Then, it would start loading next loop data which used for next convolution layer until the last layer finished.

5 Conclusion

In this project, we try to implement an FPGA-based accelerator for VGG16 neural network. Firstly, we implement the network structure in python and C. Then, we analyze the data property of input images and decide to apply dynamic fixed-point strategy. To implement this, we build a set of tools for fixed-point conversions and operations. On the hardware side, we use SDRAM as the bridge to transfer data between HPS and FPGA. We also design a dataflow of Processing Elements (PE) which contains 64 MAC units.