LLVGG16: Low Latency VGG16 Accelerator for Cloud Integration

Büşranur Yılmaz
Kirman Serdar
LLVGG16
GitHub Link: https://github.com/DHLSan/LLVGG16

Abstract

Artificial Neural Networks (ANN) contain computationally intensive complex operations and they are based on a collection of connected units or nodes called artificial neurons. They are used in many diverse areas such as detecting faults, enhancing medical applications, improving financial issues, and providing intelligent information for our daily-life. Neural networks consist of many layers to be computed and they require huge amounts of computations over real/integer/bit values on the specialized computing platform. High-speed and low-latency performance are important constraints in many different applications. VGG16[1] is based on convolutional neural networks (CNN) computations and it applies the technique of deepening the network to increase the performance of the CNN. VGG16 consists of 16 layers and 138 million parameters so its train process is highly slow. It heavily depends on multiplication and addition operation over the input and weight data. This introduces huge computational complexity. There are coarse-grained and fine-grained parallelism over the operations of CNN layers and can be described in hardware to outperform the CPU or GPU implementations. In this study, our main objective is to design a low-latency VGG16 hardware accelerator for a cloud integration. We present high-level hardware-based code-optimization methods for each layer in VGG16 to be accelerated in an FPGA-based cloud. High-level optimizations based on HLS (high-level synthesis) were proposed for the acceleration on VGG16 computation. We used Vivado HLS tool to describe our hardware accelerators for VGG16 computation through a high level language C implementation.

1. Introduction

Today, CNN is used in many tasks such as object detection, classification and semantic segmentation. The basic process of CNNs is to obtain an output by moving a specified filter step by step over the image. CNN consists of many layers. These layers are the convolution layer where the convolution process is performed, the pooling layer where the height and width information is reduced by keeping the number of channels constant in the image to reduce the computation complexity, and the fully connected layer used to connect the neurons between the layers. [2] There are many different network models such as LeNet-5[3], AlexNet[4], VGG etc. These were tested on different datasets and have different accuracy and speed values. VGG16 is preferred as a neural network model in our project as seen in Figure 1. VGG16 consists of 16 layers. It has about 138 million parameters. One of the differences from other models is that it uses 3x3 and 2x2 convolution layers. Each convolution layer aims to create a feature map that is smaller and deeper than the input image. In addition, VGG16 won the ILSVRC (ImageNet)[5] competition in 2014. While CNNs provide high accuracy, they have high computational complexity. There is also parallelism on CNN layers' operations and can be defined in hardware to perform better than CPU or GPU implementations, and in this way, the latency caused by computational complexity can be reduced. Main goal in project is to design an FPGA-based accelerator to calculate the processing speed of each image as millisecond frames per second, without observing a decrease in the top1 accuracy value of the VGG16's ImageNet dataset.

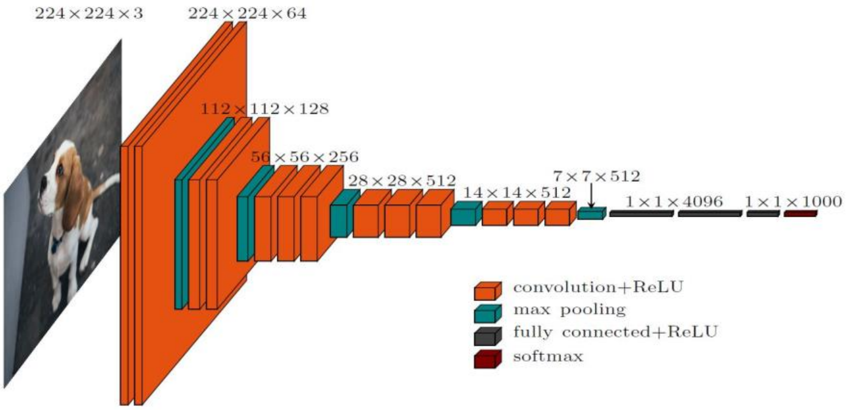


Figure 1. VGG16 Architecture

1.1. Related Works

Vinayak Gokhale et al.[6], designed an efficient hardware accelerator called Snowflake for a convolutional neural network. Their main goal is to design a scalable, low power consumption accelerator, efficient. They have implemented their system on the Xilinx Zynq XC7Z045 APSoC device. There are several important parameters mentioned throughout the paper. These parameters are trace, vector and MAC. Trace is defined as the contiguous memory region that must be accessed as part of the computation required to produce a single output pixel. Multiple traces are required to process and accumulate an output pixel. The size of the traces contingents on the number of input channels in that layer and the size of the kernel. Dividing the compute into traces allows to reduce latency associated with branches, load, stores, and other non-computational instructions. As a result, Snowflake arranges calculation into traces. In this study, 16 bit fixed point is used. Each vector, called small data blocks, contains sixteen words. MAC is the step that calculates the product of two numbers and adds that product to an adder. The sixteen words of a vector are divide between a group of multiply-accumulate units(vMAC). Multiple MAC units working together to produce output is called cooperative mode. The benefit of the cooperative mode is that it reduces the SRAM on chip necessary each MAC. Each MAC requires a feature-map and kernel to handle one layer. Maps are shared on MACs. The architecture of the system consists of control core, compute core consisting of compute units, data distribution network and memory interface. While Snowflake was able to achieve a result of 98 frames per second on AlexNet, they were able to achieve a result of 34 frames per second on GoogleNet[7]. It achieved a 91% computational efficiency, and designed a more efficient system compared to other models.

Gan Feng et al.[8], designed an energy efficient FPGA based accelerator for convolutional neural networks. They worked with LeNet-5 architecture on the Zynq 7000 platform. The main purpose of this study is to design an energy efficient and FPGA based high speed CNN accelerator for LeNet-5. Realizing this accelerator by using floating point and fixed point approximation, and reducing the hardware resources used by the activation function with some approximations, which is extensively used by the convolution and pooling layers, are among its other goals. They also aim to achieve parallelism between convolution layers and achieve a reduction in the number of cycles. They obtained 3 different approaches and made a comparison between them. These approaches are the floating-point approximation with optimized or not, and the fixed-point approximation. These have been tested on the Zynq 7000 platform. It was observed that the LeNet-5 accelerator, which was designed with the fixed-point approximation, achieved an average result of 0.151ms on 1000 images in order to estimate the MNIST dataset with an error rate of 0.99%, and gave a faster result compared to other systems.

2. Method

2.1. Software

In the software part, we implemented VGG16 with pre-trained weights in Python and we took some part of the [9] work for C language implementation. Then, we created an experimental setup to calculate accuracy drop. After preparing the experimental setup, we have determined some changes to the software to increase efficiency and prediction speed. The block diagram of the software part shown in Figure 2.

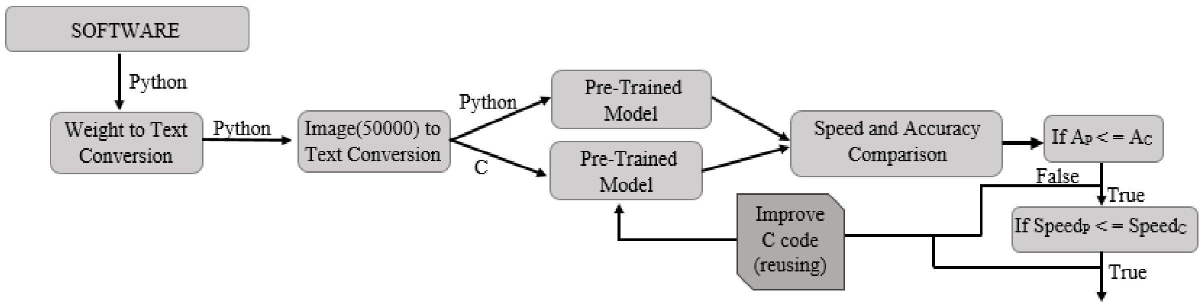


Figure 2. Design Workflow of Software Part

2.1.1. Pre-trained Model and Python / C implementation

We use a VGG16 model with C code implementation and pre-trained model weights that are trained with Keras-based VGG16 model in our project. Keras is a deep learning framework runs on the top of the Tensorflow which is developed by Google. The pre-trained VGG16 model can predict class with a bounding box for every single 1000 classes. In our work Keras based VGG16 model will be helpful for verification of accuracy between C implemented VGG16 model. In this work we use a reproduced C implementation of VGG16. All convolution layers, fully connected layers, pooling layers and activations functions implemented in C language. After implementation of model, ImageNet ILSVRC2012 dataset used for validation of VGG16 was reproduced for C language format. Our general working principle is based on C-based VGG16 prediction and verification with Keras based VGG16 model.

2.2. Hardware

In the hardware part of the project, it is aimed to implement the pre-trained VGG16 code created in the C programming language on FPGA using hardware resources. In this direction, arrangements were made on the code in order to use the pre-trained VGG16 code on the Vivado HLS platform. Afterwards, it was observed that it was not possible to accelerate the entire model in one piece in hardware, and the model was divided into parts and prepared to use certain parts in hardware. In Figure 3, the block diagram of the hardware part of the project is shown.

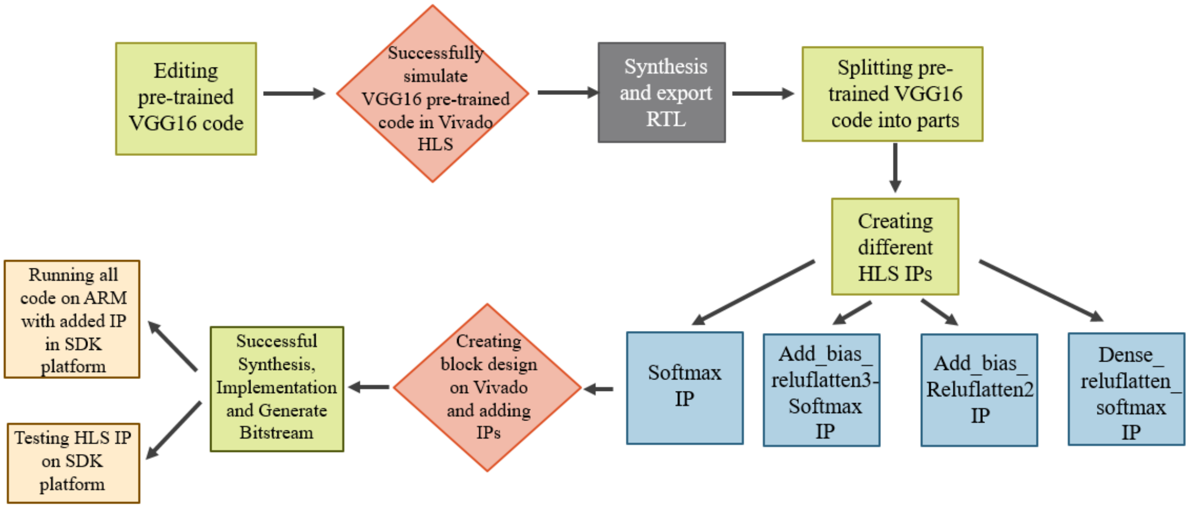

Editing pre-trained VGG16 code
Successfully simulate VGG16 pre-trained code in Vivado HLS
Synthesis and export RTL
Splitting pre-trained VGG16 code into parts
Creating different HLS IPs
Softmax IP
Add_bias_ reluflatten3-Softmax IP
Add_bias_ Reluflatten2 IP
Dense_ reluflatten_ softmax IP
Creating block design on Vivado and adding IPs
Successful Synthesis, Implementation and Generate Bitstream
Running all code on ARM with added IP in SDK platform
Testing HLS IP on SDK platform

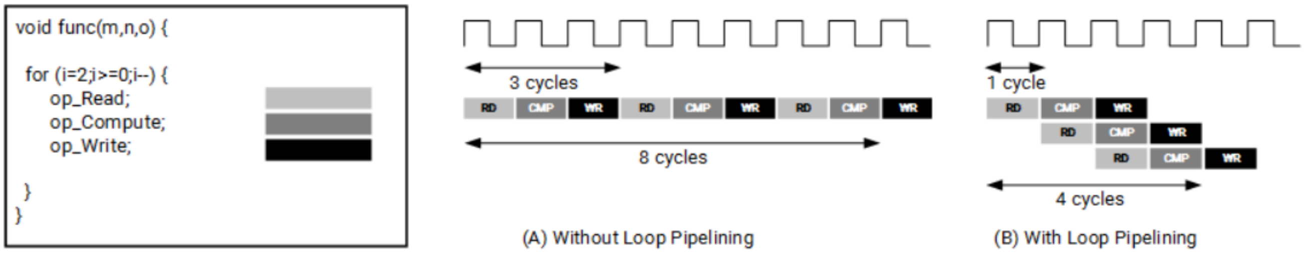
Figure 3. Design Workflow of Hardware Part

2.2.1. Arrangements in The VGG16 Pre-Trained

During this phase, it was necessary to make arrangements in the VGG16 pre-trained code used in this project. The VGG16 pre-trained code we used included dynamic memory allocation functions that Vivado HLS does not support such as malloc(), calloc() and free(). We changed the malloc() and calloc() structures to implement our project on Vivado HLS. With the malloc() and calloc() functions, we replaced the pointers created by allocating a certain place in memory with array structures of certain size. Since we define these structures without using malloc() function, the use of free() function is not necessary. In the next step, the structures defined as double pointers are converted to array structure because Vivado HLS does not support the double pointer structure. In the next stage, the weight and bias information of the convolution and dense layers of the trained model used in the predict stage were separated for each layer. Depending on this process, convolution, max pooling layers are also arranged.

2.2.2. Designing Hardware Accelerator with Vivado HLS

One of the most important stages of the project is to design a good hardware in terms of performance, speed and energy. RTL languages are very costly and it also takes a long time to design hardware with RTL languages. Using HLS, it is possible to create hardware designs both quickly and at low cost. HLS can be defined as the design process that converts programs written in programming languages such as C, C++ or System C to RTL level. The VGG16 model we prefer in our project is a large architecture with 16 layers. Since the Zedboard development kit used within the scope of the project has limited resources, it is not possible to keep all the weight files of the model in the hardware. Likewise, it was not possible to define large Array structures on the HLS platform for this reason. Within the scope of the project, several different IPs were created on the Vivado HLS platform to be accelerated in hardware. These IPs are Softmax, ReLU Flatten and Softmax, Add Bias ReLU Flatten respectively. While creating IPs, pragmas offered by HLS were used. The first of these is the pragma HLS pipeline. With this pragma, a function or loop is pipelined. It is used to start a function or loop once per clock cycle. The figure shown below shows how a function with and without a pipeline makes a difference in terms of clock cycle.

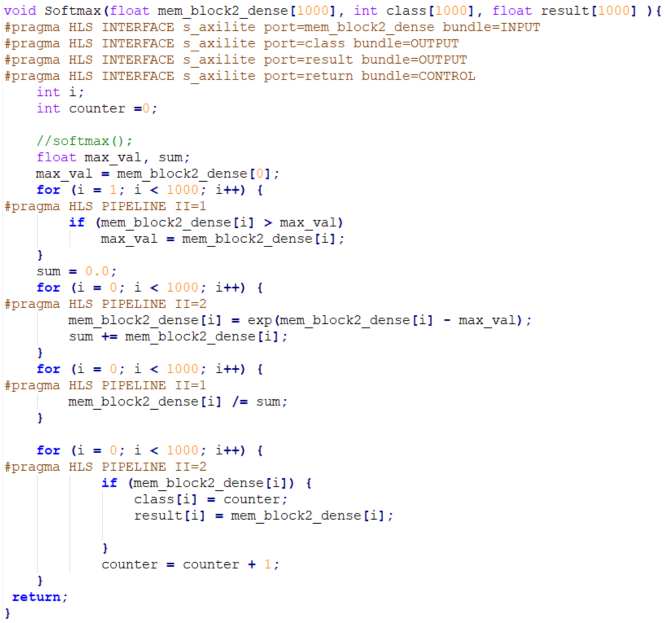


```
void func(m,n,o) {

 for (i=2;i>=0;i--) {
     op_Read;
     op_Compute;
     op_Write;

 }
}
```

Figure 4. Loop Pipeline [11]

- **Softmax IP**

In our project, there are functions belonging to the operations performed in deep learning models such as the convolution function where the convolution operation is performed, and the max-pooling function where the max-pooling operation is performed. In this section, we decided to accelerate the Softmax function, which is part of the project and the final stage of our architecture, in hardware. We added our function on Vivado HLS. We put our remaining project code on the test bench and simulated it. After an error-free simulation, we performed the synthesis process and converted it into RTL code and created the Figure 2 below shows our Softmax IP and HLS code. The created IP has one input, two outputs and a control input to use IP. Since the model is trained with the ImageNet dataset, there are 1000 classes to detect in our functions. With the module called OUTPUT, we learn which class the prediction belongs to and how accurately it predicts it.



```
void Softmax(float mem_block2_dense[1000], int class[1000], float result[1000] ){
#pragma HLS INTERFACE s_axilite port=mem_block2_dense bundle=INPUT
#pragma HLS INTERFACE s_axilite port=class bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=result bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL
    int i;
    int counter =0;

    //softmax();
    float max_val, sum;
    max_val = mem_block2_dense[0];
    for (i = 1; i < 1000; i++) {
#pragma HLS PIPELINE II=1
        if (mem_block2_dense[i] > max_val)
            max_val = mem_block2_dense[i];
    }
    sum = 0.0;
    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=2
        mem_block2_dense[i] = exp(mem_block2_dense[i] - max_val);
        sum += mem_block2_dense[i];
    }
    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=1
        mem_block2_dense[i] /= sum;
    }

    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=2
            if (mem_block2_dense[i]) {
                class[i] = counter;
                result[i] = mem_block2_dense[i];

            }
            counter = counter + 1;
    }
 return;
}
```

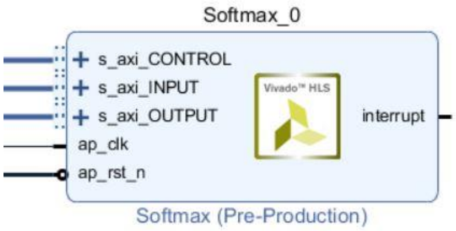
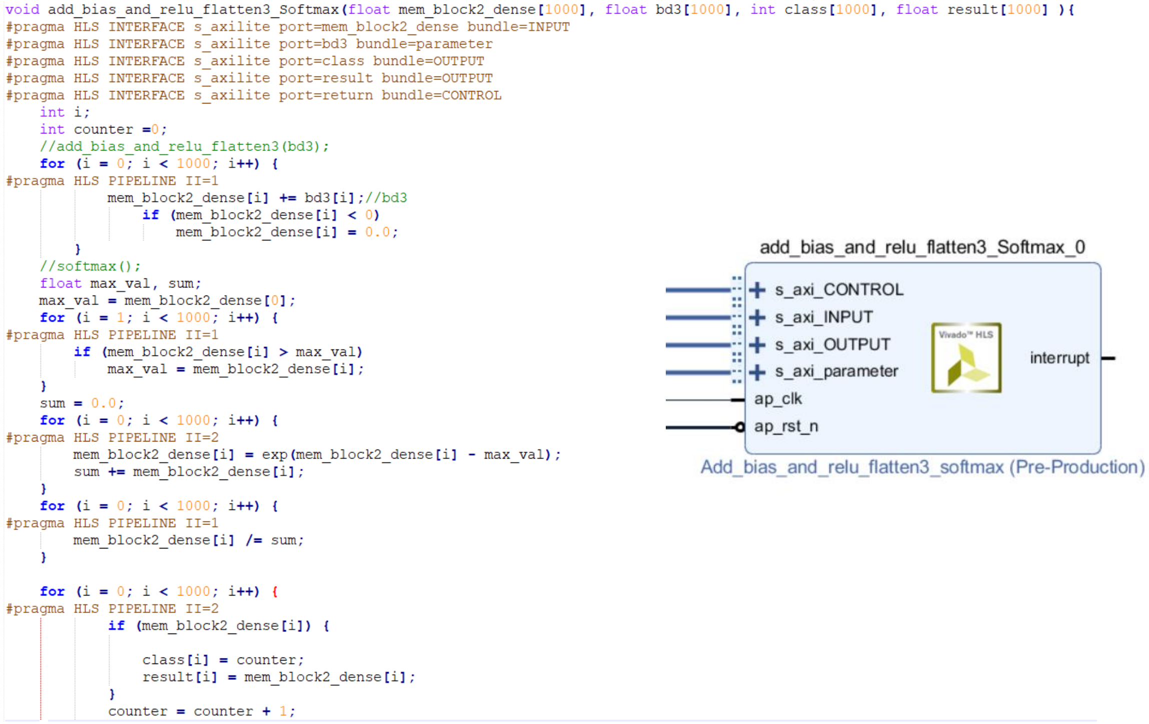


Figure 5. Softmax IP and HLS code

- **Add Bias ReLU Flatten and Softmax**

One step before the Softmax function, there was a function in which the dense layer biases were added and the Rectified Linear Unit (ReLU) operation was performed as the activation function. We created a new IP by combining this function with the Softmax function, which resulted in the last prediction map. The generated HLS IP and source code are shown in the figure below. In the created IP, there are the INPUT where the input will be given, the parameter where the bias information will be sent, CONTROL where the IP control will be made, and OUTPUT where the output will be taken.



```
void add_bias_and_relu_flatten3_Softmax(float mem_block2_dense[1000], float bd3[1000], int class[1000], float result[1000] ){
#pragma HLS INTERFACE s_axilite port=mem_block2_dense bundle=INPUT
#pragma HLS INTERFACE s_axilite port=bd3 bundle=parameter
#pragma HLS INTERFACE s_axilite port=class bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=result bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL
    int i;
    int counter =0;
    //add_bias_and_relu_flatten3(bd3);
    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=1
            mem_block2_dense[i] += bd3[i];//bd3
                if (mem_block2_dense[i] < 0)
                    mem_block2_dense[i] = 0.0;
        }
    //softmax();
    float max_val, sum;
    max_val = mem_block2_dense[0];
    for (i = 1; i < 1000; i++) {
#pragma HLS PIPELINE II=1
        if (mem_block2_dense[i] > max_val)
            max_val = mem_block2_dense[i];
    }
    sum = 0.0;
    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=2
        mem_block2_dense[i] = exp(mem_block2_dense[i] - max_val);
        sum += mem_block2_dense[i];
    }
    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=1
        mem_block2_dense[i] /= sum;
    }

    for (i = 0; i < 1000; i++) {
#pragma HLS PIPELINE II=2
            if (mem_block2_dense[i]) {

                class[i] = counter;
                result[i] = mem_block2_dense[i];
            }
            counter = counter + 1;
```

add_bias_and_relu_flatten3_Softmax_0
s_axi_CONTROL
s_axi_INPUT
s_axi_OUTPUT
s_axi_parameter
ap_clk
ap_rst_n
Vivado™ HLS
interrupt
Add_bias_and_relu_flatten3_softmax (Pre-Production)

Figure 6. Add Bias ReLU Flatten and Softmax IP and HLS code

- **Add Bias ReLU Flatten2**

The dense3 function comes just before the Add Bias and ReLU Flatten3 and Softmax function. The dense layer is a deeply connected neural network layer; This means that every neuron in the dense layer receives input from all the neurons of its previous layer. Values used in dense layers are actually parameters that can be trained and updated with the help of back propagation. Dense layers add an interesting non-linear feature so they can model any mathematical function. The Dense3 function requires a 2-dimensional weight dense input at this stage, and when the input size is reduced to one dimension, an array of size 4096000 is formed. After creating the system in Vivado and adding the IP packager, an error was received during the synthesis phase due to insufficient computer memory, and it could not continue. Therefore, it has been decided to accelerate the Add Bias and ReLU flatten 2 function, which comes before the dense3 function, in hardware. IP was created in the Vivado HLS. The figure below shows the system in which Add Bias and ReLU flatten and Add Bias and ReLU flatten2-Softmax IPs are combined.

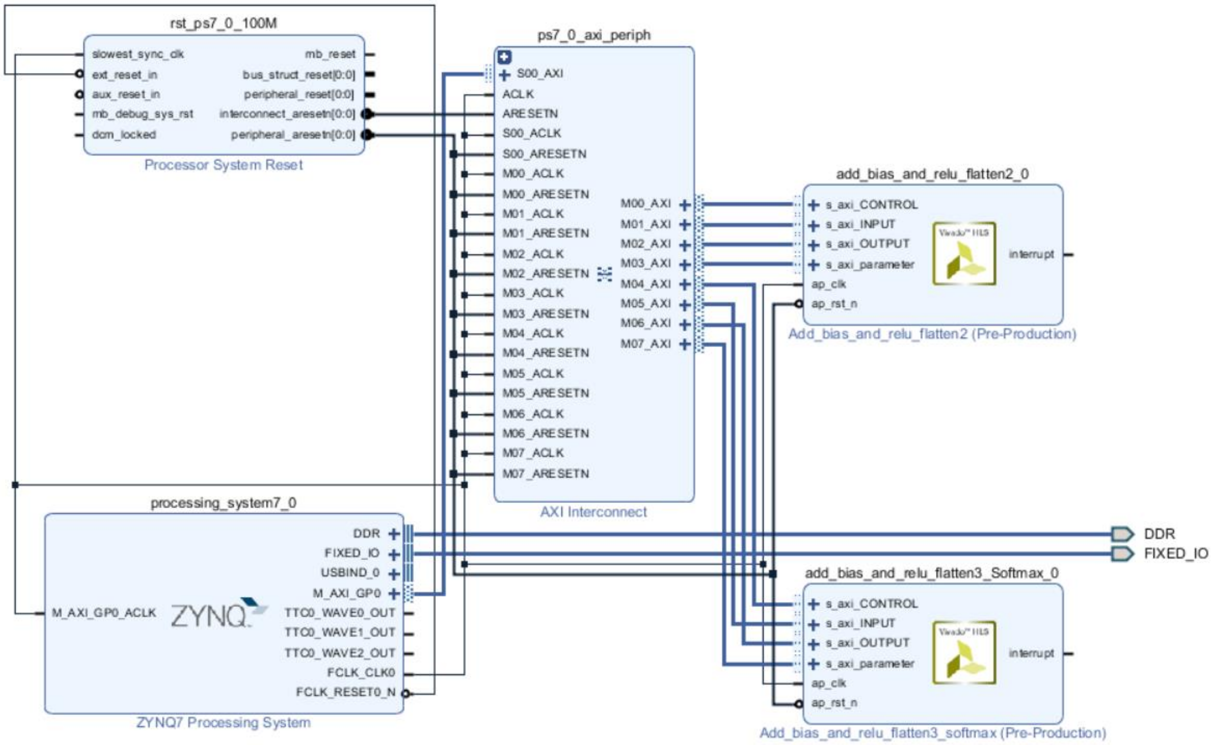


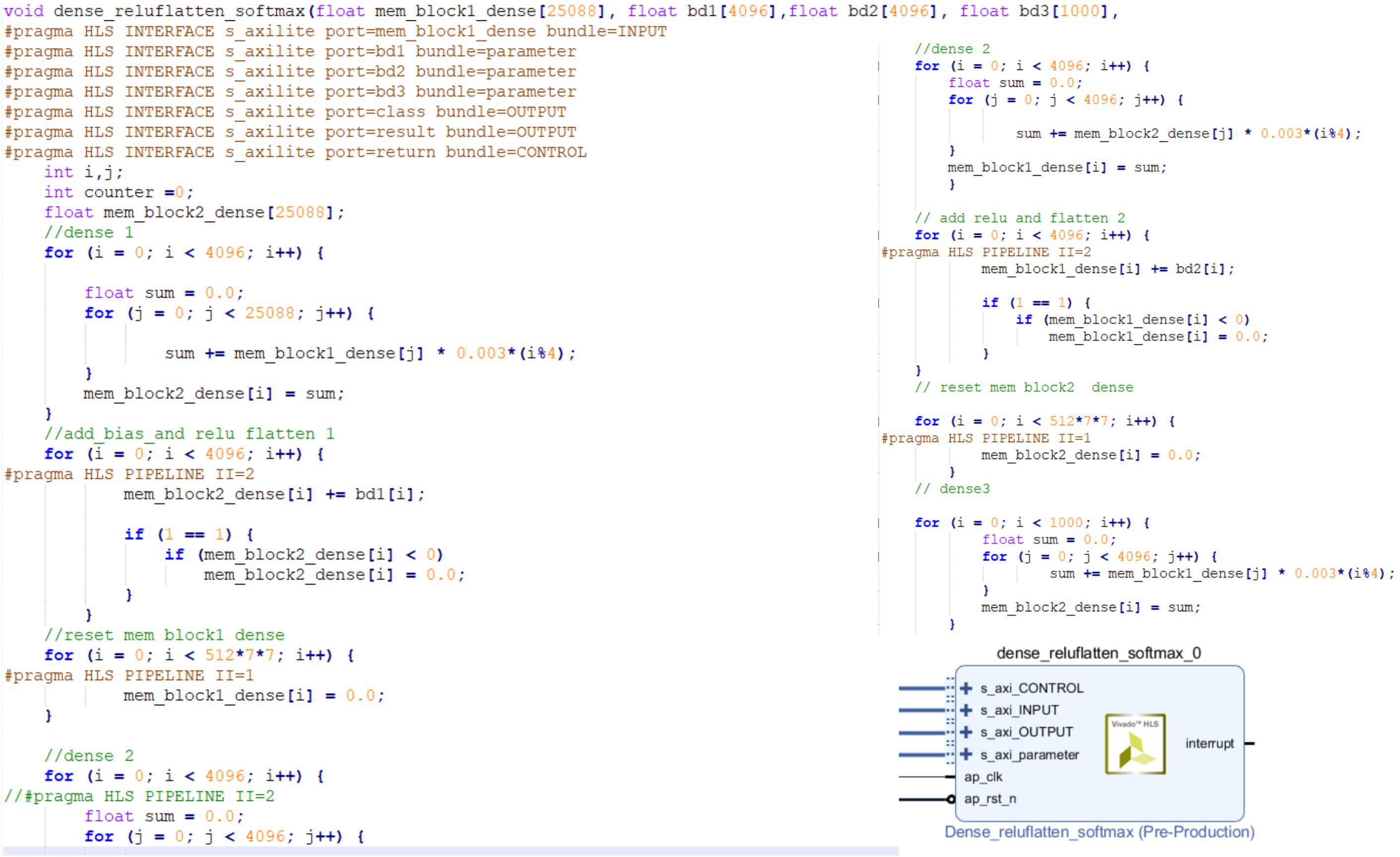
Figure 7. Add Bias ReLU Flatten3 - Softmax IP and Add Bias ReLU Flatten2 System Integration

- **Dense – AddBias and ReluFlatten - Softmax IP**

In the next stage of the project, it was decided to accelerate the part after the convolution max-pooling layer entirely in hardware. However, the dense weights used in this part of the project are 2-dimensional and even when reduced to one dimension, an error is received from the computer memory in the Vivado synthesis stage. Therefore, in the HLS phase, the code has been edited to take different weights as seen in the example code below, not giving a specific dense weight input as an input. After this arrangement made in the code, the simulation process was performed on the HLS. It has been determined that the output is not the same as the previous outputs. After this stage, IP packager was created to be used in hardware and it was observed whether it gave the same results with the HLS simulation in the test performed on the SDK platform. The results are the same. Also created IP packager Figure 8. is also shown. It takes the output of the previous flatten function and the biases of dense1, dense2, dense3 as input. As output, class and prediction value are returned.

LLVGG16: Low Latency VGG16 Accelerator for Cloud Integration

Büşranur Yılmaz
Kirman Serdar
LLVGG16
GitHub Link: https://github.com/DHLSan/LLVGG16



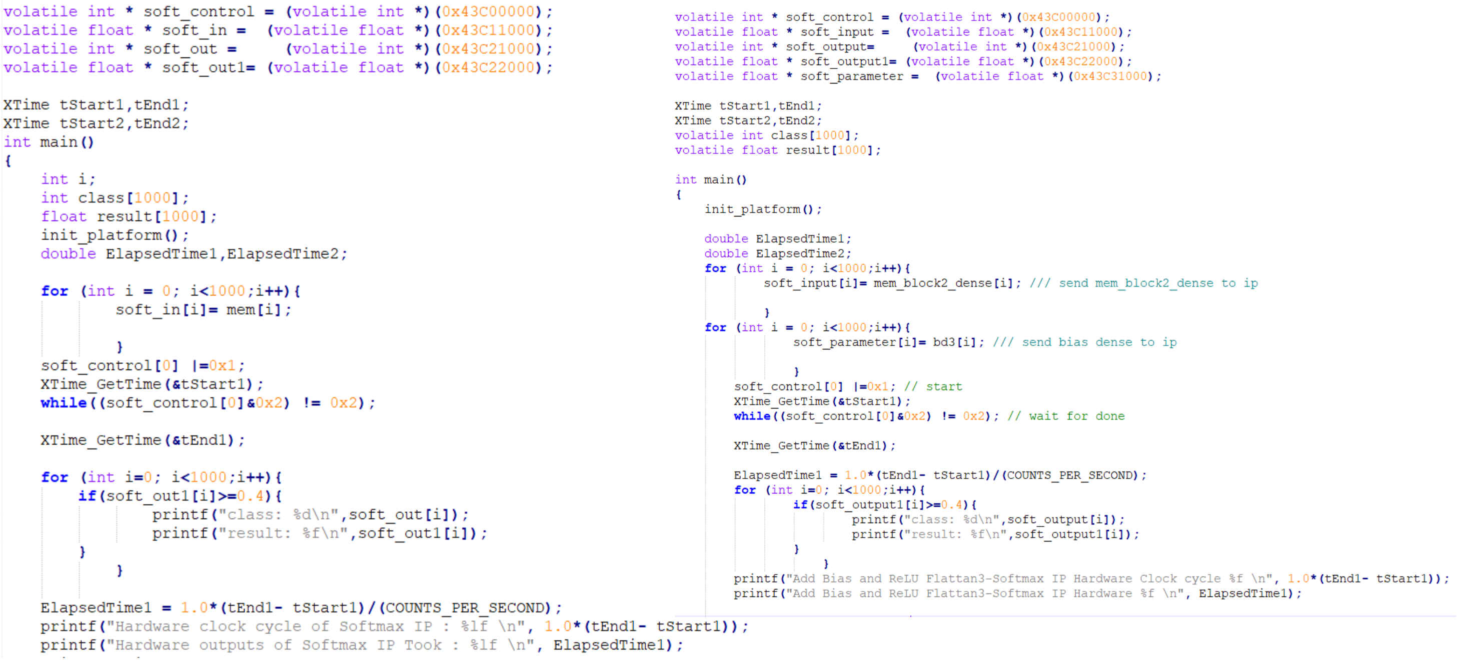
```
void dense_reluflatten_softmax(float mem_block1_dense[25088], float bd1[4096],float bd2[4096], float bd3[1000],
#pragma HLS INTERFACE s_axilite port=mem_block1_dense bundle=INPUT
#pragma HLS INTERFACE s_axilite port=bd1 bundle=parameter
#pragma HLS INTERFACE s_axilite port=bd2 bundle=parameter
#pragma HLS INTERFACE s_axilite port=bd3 bundle=parameter
#pragma HLS INTERFACE s_axilite port=class bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=result bundle=OUTPUT
#pragma HLS INTERFACE s_axilite port=return bundle=CONTROL
    int i,j;
    int counter =0;
    float mem_block2_dense[25088];
    //dense 1
    for (i = 0; i < 4096; i++) {

        float sum = 0.0;
        for (j = 0; j < 25088; j++) {

            sum += mem_block1_dense[j] * 0.003*(i%4);
        }
        mem_block2_dense[i] = sum;
    }
    //add_bias_and relu flatten 1
    for (i = 0; i < 4096; i++) {
#pragma HLS PIPELINE II=2
            mem_block2_dense[i] += bd1[i];

            if (1 == 1) {
                if (mem_block2_dense[i] < 0)
                    mem_block2_dense[i] = 0.0;
            }
        }
    //reset mem block1 dense
    for (i = 0; i < 512*7*7; i++) {
#pragma HLS PIPELINE II=1
            mem_block1_dense[i] = 0.0;
    }

    //dense 2
    for (i = 0; i < 4096; i++) {
//#pragma HLS PIPELINE II=2
        float sum = 0.0;
        for (j = 0; j < 4096; j++) {
```

```
    //dense 2
    for (i = 0; i < 4096; i++) {
        float sum = 0.0;
        for (j = 0; j < 4096; j++) {

                sum += mem_block2_dense[j] * 0.003*(i%4);
        }
        mem_block1_dense[i] = sum;
        }

    // add relu and flatten 2
    for (i = 0; i < 4096; i++) {
#pragma HLS PIPELINE II=2
            mem_block1_dense[i] += bd2[i];

            if (1 == 1) {
                if (mem_block1_dense[i] < 0)
                 mem_block1_dense[i] = 0.0;
            }
    }
    // reset mem block2  dense

    for (i = 0; i < 512*7*7; i++) {
#pragma HLS PIPELINE II=1
            mem_block2_dense[i] = 0.0;
        }
    // dense3

    for (i = 0; i < 1000; i++) {
            float sum = 0.0;
            for (j = 0; j < 4096; j++) {
                sum += mem_block1_dense[j] * 0.003*(i%4);
            }
            mem_block2_dense[i] = sum;
        }
```

Figure 8. Dense – ReluFlatten - Softmax IP and dump weight information in source code

2.2.3. Software Development Kit (SDK)

Since most functions in our project are not suitable for hardware acceleration, we decided to run functions that are not hardware accelerated on ARM processors. The created IPs were connected to the M_AXI interface of AXI interconnect and a connection was established with ARM. Software development kit (SDK) was used to test these processes. Since we created our IP's part-by-part in our project, we had to make sure that they were working correctly. We used the code snippet shown in the figure below for the created Softmax IP and Add Bias ReLU Flatten - Softmax. The most important point in this part was to make sure that the inputs sent by the SDK for the IPs on the hardware side went to the correct address.



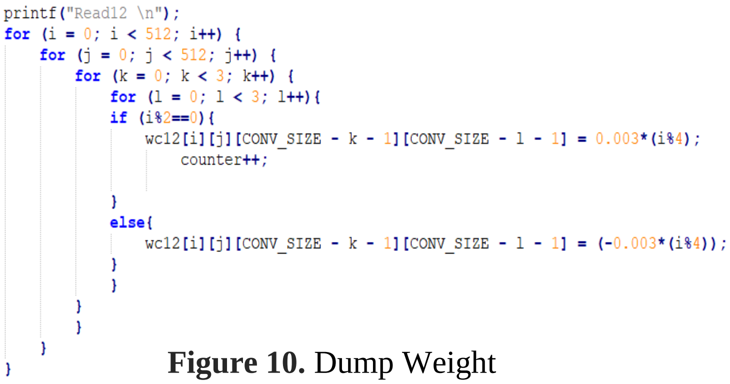
```
volatile int * soft_control = (volatile int *)(0x43C00000);
volatile float * soft_in =  (volatile float *)(0x43C11000);
volatile int * soft_out =     (volatile int *)(0x43C21000);
volatile float * soft_out1= (volatile float *)(0x43C22000);

XTime tStart1,tEnd1;
XTime tStart2,tEnd2;
int main()
{
    int i;
    int class[1000];
    float result[1000];
    init_platform();
    double ElapsedTime1,ElapsedTime2;

    for (int i = 0; i<1000;i++){
            soft_in[i]= mem[i];

            }
    soft_control[0] |=0x1;
    XTime_GetTime(&tStart1);
    while((soft_control[0]&0x2) != 0x2);

    XTime_GetTime(&tEnd1);

    for (int i=0; i<1000;i++){
        if(soft_out1[i]>=0.4){
            printf("class: %d\n",soft_out[i]);
            printf("result: %f\n",soft_out1[i]);
        }
            }

    ElapsedTime1 = 1.0*(tEnd1- tStart1)/(COUNTS_PER_SECOND);
    printf("Hardware clock cycle of Softmax IP : %lf \n", 1.0*(tEnd1- tStart1));
    printf("Hardware outputs of Softmax IP Took : %lf \n", ElapsedTime1);
```

```
volatile int * soft_control = (volatile int *)(0x43C00000);
volatile float * soft_input =  (volatile float *)(0x43C11000);
volatile int * soft_output=     (volatile int *)(0x43C21000);
volatile float * soft_output1= (volatile float *)(0x43C22000);
volatile float * soft_parameter =  (volatile float *)(0x43C31000);

XTime tStart1,tEnd1;
XTime tStart2,tEnd2;
volatile int class[1000];
volatile float result[1000];

int main()
{
    init_platform();

    double ElapsedTime1;
    double ElapsedTime2;
    for (int i = 0; i<1000;i++){
            soft_input[i]= mem_block2_dense[i]; /// send mem_block2_dense to ip

            }
    for (int i = 0; i<1000;i++){
                soft_parameter[i]= bd3[i]; /// send bias dense to ip

                }
        soft_control[0] |=0x1; // start
        XTime_GetTime(&tStart1);
        while((soft_control[0]&0x2) != 0x2); // wait for done

        XTime_GetTime(&tEnd1);

        ElapsedTime1 = 1.0*(tEnd1- tStart1)/(COUNTS_PER_SECOND);
        for (int i=0; i<1000;i++){
                if(soft_output1[i]>=0.4){
                        printf("class: %d\n",soft_output[i]);
                        printf("result: %f\n",soft_output1[i]);
                }
                    }
        printf("Add Bias and ReLU Flattan3-Softmax IP Hardware Clock cycle %f \n", 1.0*(tEnd1- tStart1));
        printf("Add Bias and ReLU Flattan3-Softmax IP Hardware %f \n", ElapsedTime1);
```

Figure 9. Simple source code of IPs

We ran all of our VGG16 pre-trained code primarily on the ARM processor. In the beginning, weight files were being read from txt file in our project. Despite its large size, we added the weights and biases of all layers to our system by making them a header file. Since the sizes were large, we could not define the 2 weight files of the convolution layers on the SDK. Instead, we adjusted the weights ourselves to get a closer value. Since the size of the weight file of the first dense layers is too large to be used in the SDK, we edited it the same way. Figure 7 shows the changes we made to the code.

We made the same changes on Vivado HLS and tested the system in the same way. We checked to see if the results were the same. Later, we added the IP packagers we added to the places where that function was called in our code and tested our system in this way.



```
printf("Read12 \n");
for (i = 0; i < 512; i++) {
    for (j = 0; j < 512; j++) {
        for (k = 0; k < 3; k++) {
            for (l = 0; l < 3; l++){
            if (i%2==0){
                wc12[i][j][CONV_SIZE - k - 1][CONV_SIZE - l - 1] = 0.003*(i%4);
                    counter++;

            }
            else{
                wc12[i][j][CONV_SIZE - k - 1][CONV_SIZE - l - 1] = (-0.003*(i%4));
            }
            }
        }
        }
    }
}
```

Figure 10. Dump Weight

3. Empirical Setup

We measured how much the hardware elements we developed in our study would speed up our model. We have not made any changes to the board for this purpose. We used the Vivado HLS application to create our IPs. Later, we transferred our IPs, which we successfully simulated and synthesized in the Vivado HLS, to the Vivado platform. Vivado platform helps us to run our IPs with Zynq hardware. The SDK platform is involved in the part of the code required for an image prediction to run on the hardware we designed.

4. Empirical Results

We performed our first experiments on the Intel i7-7700HQ CPU. As the output of these experiments, the average processing time of an image is 3.7s. However, the reading time of the weight file is 87s. In total, we have defined 4 IPs to speed up hardware. We combined the prepared IPs with our code that will run on ARM on the SDK side. We combined the first created Softmax IP with Add Bias and ReLU flatten3 function. We calculated the contribution of these created IPs to the system and just how long it took and how long it would take on ARM if it were not created as a hardware IP. The hardware and software of the created IPs were also tested. The results of these tests are shown in Table 1. Softmax IP created with HLS works 2.4 times faster than the software function. Likewise, Add bias and Relu flatten3 - Softmax IP created with HLS works 4.5 times faster than the software function. The system, created by integrating Add bias and relu flatten2 and Add bias and Relu flatten- Softmax IPs, works 2.6 times faster than software functions. All VGG16 pre-trained code takes a total of 304.35 seconds to run on the ARM operating system.

	HARDWARE		SOFTWARE	
	CLOCK CYCLE	ELAPSED TIME	CLOCK CYCLE	ELAPSED TIME
Softmax IP	43567	0.131 ms	105564	0.317ms
Add bias and ReLU flatten3 + Softmax IP	40246	0.121 ms	181020	0.543 ms
Add bias and ReLU flatten2 IP	41023	0.123 ms	33561	0.101 ms
Add bias and ReLU flatten3 + softmax + Add bias and ReLU flatten2 Ips	81369	0.244 ms	214581	0.644 ms
Dense + Add bias and ReLU flatten + softmax	8298641635	24.895	4046817009	12.59 s

Table 1. IP Time Comparison

5. Conclusion

In this project, we tried to implement an FPGA-based Convolutional Neural Network accelerator. We implemented the VGG16 pre-trained model, prediction code and validation code in both Python and C. Then we tried to create an IP packager to edit these codes and speed them up in hardware. We could not transfer all our code to the hardware side because the architecture is large and uses too much memory. Therefore, we decided to speed up the parts of our code after the flatten function in hardware, and we created different IPs by editing the functions used in that part. As a result of all these processes, we ran some of our code on the ARM processor and some on the hardware IP. According to the speed results we obtained, we tried to accelerate the computationally intensive functions in hardware. Speed gain was obtained with Softmax, add bias and relu flatten3-softmax IPs created. In the next stages of the project, it is planned to define other functions of the VGG16 pre-trained code in the hardware. It is planned to accelerate communication to functions that involve less but more computational computing in hardware.

References

[1] K. Simonyan and A. Zisserman, ‘Very Deep Convolutional Networks for Large-Scale Image Recognition’, *ArXiv14091556 Cs*, Apr. 2015, Accessed: Apr. 25, 2021. [Online]. Available: http://arxiv.org/abs/1409.1556.

[2] S. Albawi, T. A. Mohammed, and S. Al-Zawi, ‘Understanding of a convolutional neural network’, in *2017 International Conference on Engineering and Technology (ICET)*, Antalya, Aug. 2017, pp. 1–6, doi: 10.1109/ICEngTechnol.2017.8308186.

[3] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, ‘Gradient-Based Learning Applied to Document Recognition’, p. 46, 1998.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ‘ImageNet classification with deep convolutional neural networks’, *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, ‘ImageNet: A large-scale hierarchical image database’, p. 8.

[6] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, ‘Snowflake: An efficient hardware accelerator for convolutional neural networks’, in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, Baltimore, MD, USA, May 2017, pp. 1–4, doi: 10.1109/ISCAS.2017.8050809.

[7] C. Szegedy *et al.*, ‘Going deeper with convolutions’, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, Jun. 2015, pp. 1–9, doi: 10.1109/CVPR.2015.7298594.

[8] G. Feng, ‘Energy-efficient and high-throughput FPGA-based accelerator for Convolutional Neural Networks’, p. 3.

[9] ZFTurbo, VGG16-Pretrained-C, (2017), GitHub repository, https://github.com/ZFTurbo/VGG16-Pretrained-C

[10] A. Aimar *et al.*, ‘NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps’, *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019, doi: 10.1109/TNNLS.2018.2852335.

[11]https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html