Project Proposal: A FPGA accelerator for YOLO CNN based on weight quantization and data flow optimization

Botong Xiao bx2197, Haoran Jing hj2588, Terry Zhang tz2477, Yunran Zhou yz3985

Abstract—**Real-time object detection requires high throughput and power efficiency. However, many convolutional neural networks (CNNs) have frequency access to off-chip memory which causes slow processing and undesired power dissipation. In this project, we want to implement a streaming hardware accelerator with a YOLO(You-Only-Look-One) CNN. In addition, the parameters of the CNN will be quantized using binary weight and low-bit activation. Quantization would allow us to store the whole CNN in the on-chip block DRAM. Moreover, the hardware implementation of the CNN will be fully pipelined to improve hardware utilization and reusability of intermediate data. In all, the goal of this project is to eliminate off-chip memory access, improving hardware utilization to better throughput and energy efficiency.**

I. Introduction

DEEP learnging has been the most prevalent method in various task in computer vision due to the support of powerful computation devices such as GPU. Among the state-of-the-art methods, YOLO and the subsequent Sim-YOLO, YOLO-V2 demonstrate the most promising trade-off between speed and accuracy. While GPU is widely used for the training and inference of deep learning algorithms such as YOLO, recent researches have proved its inefficiency in optimization such as the the quantization of weight/activation and data access schedule. For instance, [2] has demonstrated that the training and inference of CNNs can be quantized to a very low-bit precision with insignificant loss in accuracy. This quantization enables a fast , memory efficient, and power efficient FPGA accelerator.

Based on this, [1] proposed a optimized data path to reduce the frequency of off-chip memory access. In this project, we target to replicate the design of [1]. In the following sections, the hardware and software are explained in detail. Milestones of this project is listed in the last section.

II. Specification

Fig.1 presents the overall architecture of the whole FPGA implementation. The data will be input from peripherals such as the camera through PCIE. Then the data will directly be sent to the DRAM through the YOLO DMA. The accelerator will fetch the input image data from the DRAM and perform the computation and then send the detection result back to the DRAM. Eventually, the detection result will output from the DRAM and be sent back to other peripherals such as the monitor through PCIE. The main design idea and details will be illustrated in hardware specification and software specification as follows.

A. Hardware Specification

The YOLO accelerator mainly consists of a controller, convolution layers, and buffers between each layer. Each convolution layer consists of three layers performing convolution computation, batch normalization, and max pooling, respectively.

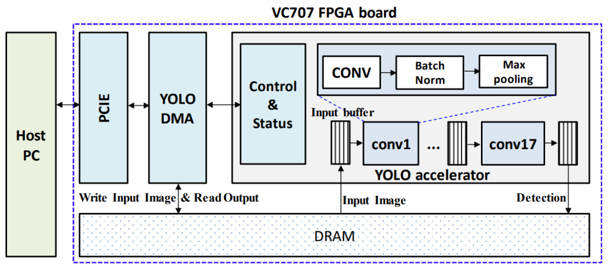


Fig. 1: The Streaming Architecture

The overall proposed structure of the YOLO accelerator is shown in Fig.2. The data input from the previous layer will first be stored in a circular buffer. The circular buffer includes four lines of SRAM. Three of them are the partial inputs from the previous layer and will be divided into sliding cubes and performed 3×3 convolution with the 3×3 kernel. Another additional line is to enable the overlapping of the computation of the current layer and the previous layer. The partial output result of convolution will be sent to the adder tree which consists of two stages of ternary adders. The partial results from adders will be stored in the line buffers, and be sent to perform batch normalization and max-pooling when the computation is completed. Eventually, the final results will be transferred to the next layer and sent back to the DRAM after output from the last convolution layer.

B. Software Specification

For the software part, our initial thought of the program will roughly mainly focus on 3 parts: I/O interface, accelerator and user interface. At the I/O interface part, the software needs to make the connection to the hardware, which means there should be an interface for the camera for object detecting, an interface for video output, making those devices communicate and transfer data. The second part is the accelerator, containing

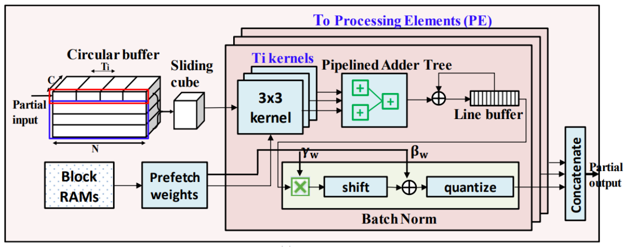


Fig. 2: FPGA implementation of convolutional layers

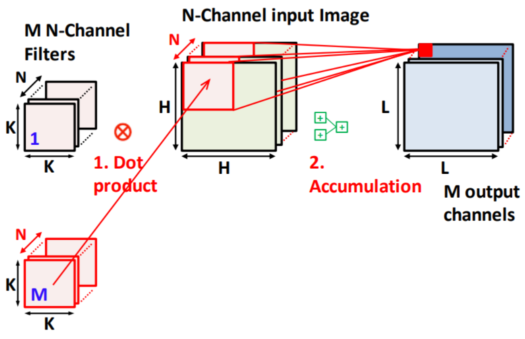


Fig. 3: Operation of convolutional layers

the convolution layer and buffer which has mentioned above. The work for software is to create those buffers from RAM and also for the algorithm of the accelerator algorithm. The basic principle of the CNN network is shown below at left:

For the CNN network, weight must be used for learning, which could cause a lot of computational resources. Our plan is to calculate the weight off-board, then add the weight to the FPGA. Due to limited RAM resources, those algorithms need optimization in order to make room for buffer. Our plan is to use the method from [1] to reduce the weight re-use, the abstract method from [1] is shown above at right. After optimization, we should be able to put the buffer and algorithm onto the FPGA RAM. And the last part is the user interface. Because it is a project for object detection, it should have a direct image or video output to show the result of the object detection on a screen. Users can choose to activate and deactivate the function, where the object detected should be highlighted and labeled.

III. Milestones

TABLE I: Milestones

Fully understand the CNN algorithm	Mar. 4
Design the RTL implementation for the accelerator	Apr. 22
Verify the accelerator RTL implementation functionality	Apr. 29
Develop the software to connect the accelerator with peripherals	May. 6
Perform verification for the whole design	May. 13

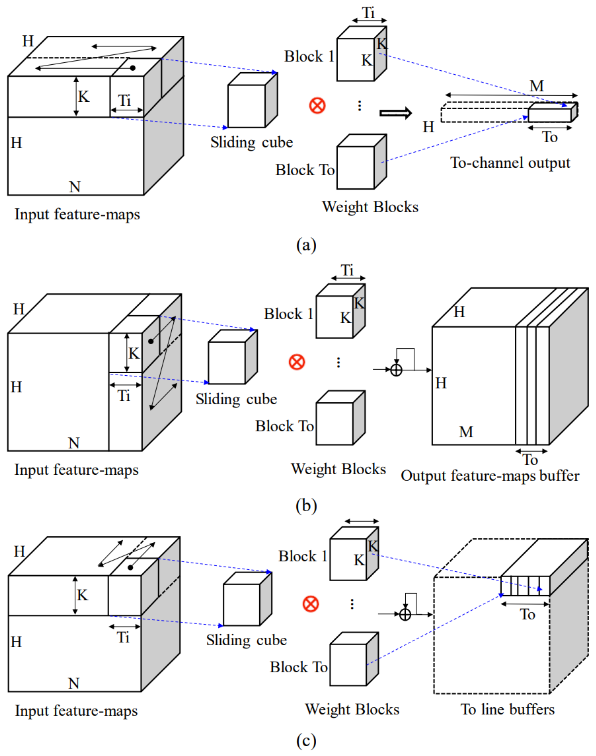


Fig. 4: Different streaming schedule (a) No weight reuse (b) Full weight reuse (c) Line based weight reuse

References

[1] D. T. Nguyen, T. N. Nguyen, H. Kim and H. Lee, "A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 8, pp. 1861-1873, Aug. 2019, doi: 10.1109/TVLSI.2019.2905242.

[2] D. T. Nguyen, H. Kim, H. -J. Lee and I. -J. Chang, "An Approximate Memory Architecture for a Reduction of Refresh Power Consumption in Deep Learning Applications," 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 2018, pp. 1-5, doi: 10.1109/ISCAS.2018.8351021.

The Design Document

Botong Xiao bx2197, Haoran Jing hj2588, Terry Zhang tz2477, Yunran Zhou yz3985

I. Introduction

An FPGA accelerator for YOLO CNN based on weight quantization and data flow optimization In the computer vision area, object detection is a challenging task. This project is aiming to design an accelerator for YOLO(You-Only-Look-One) CNN on the FPGA board. The YOLO is a single neural network predicting the object bounding boxes which perform the best trade-offs between accuracy and latency. The whole design idea will be illustrated in hardware and software parts separately.
For the software part, it is mainly responsible for interacting with the environment. To be more specific, it will first receive the image data from the camera through the USB port. Then the software part will manage the input data in a specific way and stream them into the accelerator through the device driver. After the computation by the accelerator, the results will be retrieved and sent back to the software part and performed post-processing. Eventually, the detection results will be shown on the screen through a VGA port.
For the hardware part, it consists of 17 convolution layers and 4 max-pooling layers. Each convolution layer includes adder trees to perform the convolution computation and an accumulator to perform the batch normalization. The parameters for batch normalization and convolutional kernels can be computed ahead and preloaded into the DRAM. The weights we use in this network are in binary which takes only 1 bit and most of the output and intermediate results are quantized to 6bits, therefore the memory resource on FPGA is able to fit all the parameters preloaded. As a result, all the computation will be performed on-chip, and there is no necessary to access data off-chip. The memory resource budget will be illustrated more specifically in section 4.

II. Block Diagram

The approximate block diagram of our design is shown in Fig.1. To make a project for object detection, we decided to use a USB camera for image sensing. The video data coming from the camera will be cut into frames by the driver, then sent to the CNN driver through the Avalon bus. Then software streaming logic should present as YOLO Accelerator in the FPGA hardware, which can process the incoming data in multiple layers to achieve YOLO-CNN implementation. The output data will be sent to the VGA port, connecting a VGA monitor to indicate the final object detection result.

The main feature of the hardware part is the YOLO CNN accelerator. The YOLO accelerator contains an input buffer, a controller, DRAM, and multiple convolution layers. There are 21 layers, which contain 4 max-pooling layers and 17 convolution layers in our design. In each convolution layer, there are 3 sub-layers performing convolution computation, batch normalization, and max pooling. For the previous convolution layer, its input will be sent to a circular buffer for storage. These buffers include 4 lines of SRAM, where 3 of them are partial inputs from the previous layer, performing 3*3 convolution with the 3*3 kernel. The remaining line is for overlapping computation. The partial output will be sent to an adder, where the results are then stored in the line buffer for batch normalization and max-pooling computation. The final results are transferred to the next layer until the overall layer computation finishes. In the batch normalization part, the activations also need to be shifted and quantized to get the partial output.

The main job of the software side is to provide the correct data flow for the whole system: from camera input to CNN accelerator then to screen output. The first part is the drivers mentioned in the block diagram. The YOLO CNN driver will read and write the actual data stored in the buffer and make convolution computations. A camera driver should have the function of recognizing the USB protocol for data input. At last, when the CNN network has finished its processing, the output data flow was received and shown properly by the VGA screen driver, whose main function is to convert the output data into the VGA signal that the monitor could handle. As a result, we can obtain the real-time image of the camera with CNN processed object-detection boundaries.

The optimized streaming protocol is shown in Fig.2. We first use preload weight data and batch normalization parameters into on-chip memory. Then, we will be using a USB video camera to capture the frame data. The frame data of the camera will be communicated to the FPGA board using the UVC protocol.

After the frame data is successfully transmitted. A CNN device driver will stream the data into the CNN accelerator block by block. To increase scalability and maximize intermediate data reuse, our CNN will compute the data in a special streaming order. The software will stream the frame data in the following way to ensure correct functionality.

In the input feature map, The number of channels is N. In our case, the frame input from the camera has three channels: RGB. The sliding cube indicates the data being sent to the Yolo CNN accelerator. As we can see from the graph, data is passed in along the width of the whole frame data. When the first horizontal layer that has height k is passed, we move on to pass the next horizontal layer. After the CNN accelerator

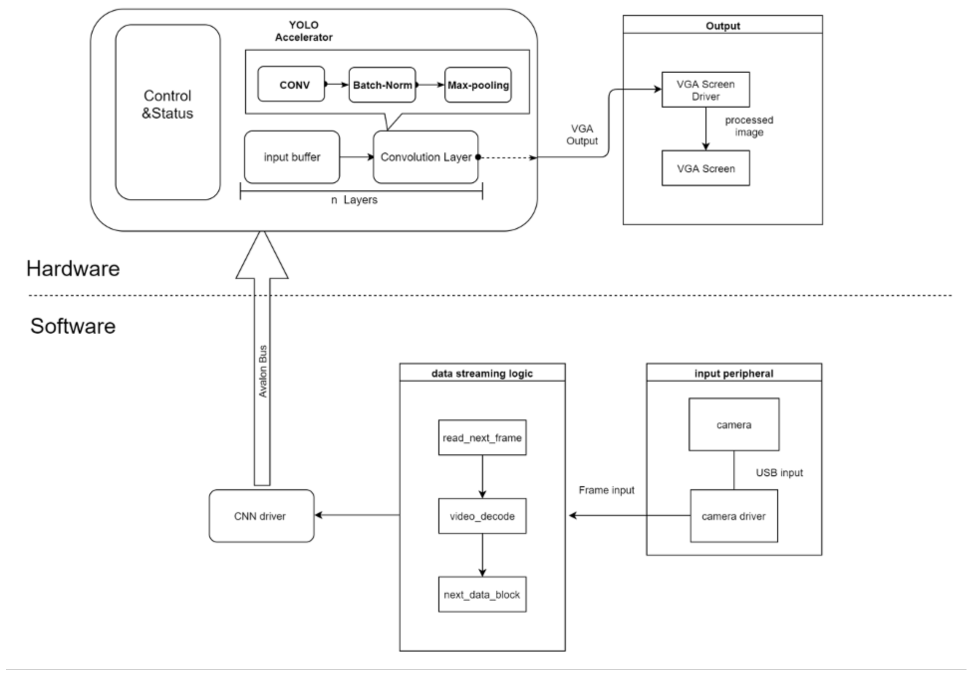


Fig. 1: Block diagram

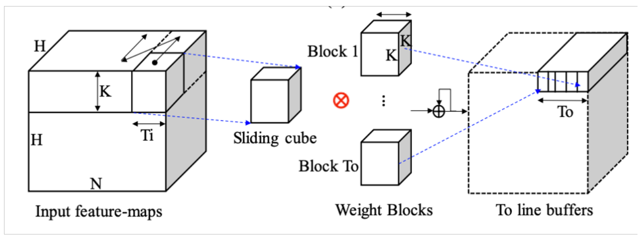


Fig. 2: Data flow

finishes inferencing, the output will be communicated directly to the monitor screen using VGA protocol. Through raster scanning, the monitor will show the input frame with the detected object enclosed in a square.

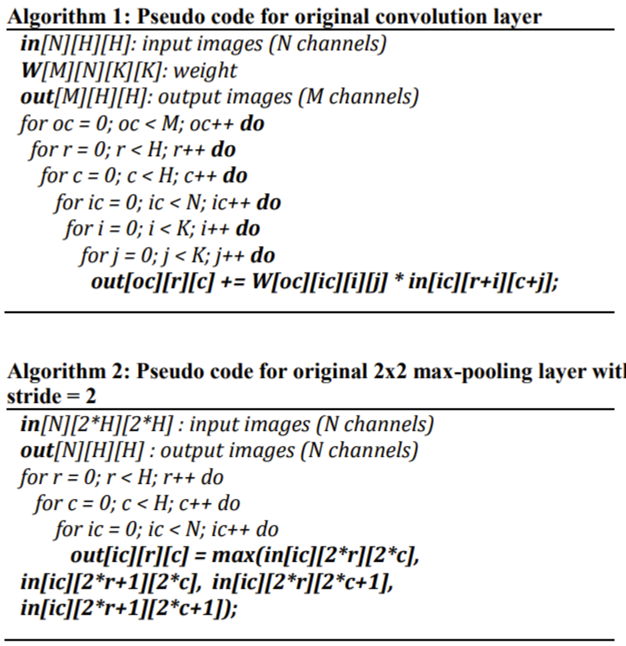
III. Algorithms

The main algorithm of the accelerator is to implement convolution computation and max-pooling. The pseudo-code for the convolution layer and max-pooing layer is shown in Fig.3.

In each convolution layer, the convolution computation is always followed by batch normalization. The original batch normalization is shown below.

$$y = \frac{\gamma^{(i)}(act - \mu^{(i)})}{\sqrt{[\sigma^{(i)}]^2 + \epsilon}} + \beta^{(i)} \tag{1}$$

where y and act are the outputs of batch-normalization and convolutional computation, respectively. $\mu(i)$, $[\sigma(i)]^2$ are channel-wise mean and variance of activations, respectively.



Algorithm 1: Pseudo code for original convolution layer

```
in[N][H][H]: input images (N channels)
W[M][N][K][K]: weight
out[M][H][H]: output images (M channels)
for oc = 0; oc < M; oc++ do
  for r = 0; r < H; r++ do
    for c = 0; c < H; c++ do
      for ic = 0; ic < N; ic++ do
        for i = 0; i < K; i++ do
          for j = 0; j < K; j++ do
            out[oc][r][c] += W[oc][ic][i][j] * in[ic][r+i][c+j];
```

Algorithm 2: Pseudo code for original 2x2 max-pooling layer with stride = 2

```
in[N][2*H][2*H] : input images (N channels)
out[N][H][H] : output images (N channels)
for r = 0; r < H; r++ do
  for c = 0; c < H; c++ do
    for ic = 0; ic < N; ic++ do
      out[ic][r][c] = max(in[ic][2*r][2*c],
in[ic][2*r+1][2*c], in[ic][2*r][2*c+1],
in[ic][2*r+1][2*c+1]);
```

Fig. 3: Algorithm

$\gamma(i)$ and $\beta(i)$ are the channel-wise scale and bias, respectively.

However, the original batch normalization is not easy for the hardware to implement. Therefore, the batch normalization has been optimized as below:

$$y = x_{W(i)} \times \gamma_w^{(i)} + \beta_w^{(i)} \tag{2}$$

where $\gamma w(i)$ and $\beta w(i)$ are the new scale and bias factors that can be computed beforehand by the software part and preloaded into the DRAM of FPGA:

$$\gamma_w^{(i)} = \frac{\mu_W^{(i)} \times \gamma^{(i)}}{\sqrt{[\sigma^{(i)}]^2 + \epsilon}} \tag{3}$$

$$\beta_w^{(i)} = -\frac{\mu_W^{(i)} \times \mu^{(i)}}{\sqrt{[\sigma^{(i)}]^2 + \epsilon}} + \beta^{(i)} \tag{4}$$

With the optimized batch normalization and preloaded scale and bias factors, the accelerator only requires one multiplication and one addition to perform the batch normalization.

For the software part, in addition to the scale and bias factors computation mentioned before, it will also implement the algorithm written in C language to build a golden block for the whole design. The actual hardware implementation may perform some quantization for the activation to save the hardware resource consumption and eliminate off-chip access, which may lead to some accuracy loss. But the software still could use the simple and original version of algorithm to build a golden block. By analyzing the results of the golden block and accelerator, we can identify whether the accuracy of accelerator output is acceptable and therefore verify the correctness of the accelerator.

IV. Resource Budget

The FPGA resource used in this accelerator is reported in [1]. The network structure, weight size, LUT and FF resource budget are Shown below.

Layer	Type	Size / Stride	Filter number	Input Size	Output Size	Out bit width	PF*** (Ti, To)
0	C*	3×3 / 1	32	416×416×3	416×416×32	6	(3, 32)
1	M**	2×2 / 2		416×416×32	208×208×32	6	(8, 8)
2	C*	3×3 / 1	64	208×208×32	208×208×64	6	(8, 8)
3	M**	2×2 / 2		208×208×64	104×104×64	6	N/A
4	C*	3×3 / 1	128	104×104×64	104×104×128	6	(8, 8)
5	C*	1×1 / 1	64	104×104×128	104×104×64	6	(8, 8)
6	C*	3×3 / 1	128	104×104×64	104×104×128	6	(8, 8)
7	M**	2×2 / 2		104×104×128	52×52×128	6	N/A
8	C*	3×3 / 1	256	52×52×128	52×52×256	6	(8, 8)
9	C*	1×1 / 1	128	52×52×256	52×52×128	6	(8, 8)
10	C*	3×3 / 1	256	52×52×128	52×52×256	6	(8, 16)
11	M**	2×2 / 2		52×52×256	26×26×256	6	N/A
12	C*	3×3 / 1	512	26×26×256	26×26×512	6	(16, 8)
13	C*	1×1 / 1	256	26×26×512	26×26×256	6	(8, 16)
14	C*	3×3 / 1	512	26×26×256	26×26×512	6	(16, 8)
15	C*	1×1 / 1	256	26×26×512	26×26×256	6	(8, 16)
16	C*	3×3 / 1	512	26×26×256	26×26×512	4	(16, 8)
17	M**	2×2 / 2		26×26×512	13×13×512	4	N/A
18	C*	3×3 / 1	1024	13×13×512	13×13×1024	6	(8, 16)
19	C*	1×1 / 1	512	13×13×1024	13×13×512	4	(16, 8)
20	C*	3×3 / 1	1024	13×13×512	13×13×1024	6	(8, 16)
21	C*	1×1 / 1	125	13×13×1024	13×13×125	16	(16, 5)

Note: C*=Convolution, M**=Maxpool, PF***= Parallelism Factors

Fig. 4: The network structure and the weight quantization

[1] implemented the accelerator with a different FPGA devices but we anticipate a similar amount of resource will be used in this project.

Networks	Quantization	Accuracy (%)	Weight size (MB)	Complexity (GOP)
(1) YOLO - v2	Full precision	75.88	258	34.9
	1-b W, 32-b A	71.56	8.1	17.45
	1-b W, 6-b A	71.11	8.1	17.45
(2) Sim-YOLO-v2	Full precision	72.0	79.74	18.95
	1-b W, 32-b A	66.99	2.54	9.48
	1-b W, 6-b A	65.76	2.54	9.48
(3) Sim-YOLO-v2 FPGA	Full precision	66.79	58.28	17.18
	1-b W, 32-b A	64.95	1.88	8.59
	1-b W, 6-b A	65.07	1.88	8.59
	1-b W, 4-to-6-b A	64.16	1.88	8.59

Fig. 5: weight size w/ and w/o quantization

Features	**Performance w/o batch**	**Performance w/ batch**
Device	Virtex-7 VC707 FPGA	
Operating frequency	200 MHz	
Block RAMs (18 Kb)	1144 (55.5%)	
DSPs	272 (9.7%)	
LUTs – FFs	155.2K (51.1%) – 115K (18.9%)	
mAP	64.16%	
DRAM bandwidth	47.2 MB/s	84.96 MB/s
Frame rate (416 × 416)	60.72 fps	109.3 fps
Throughput	1043 GOPS	1877 GOPS
Power	11.11 W	18.29 W

Fig. 6: Implementation results from [1]

V. Hardware-software interface

Since the only interaction between our CNN accelerator and the software side is one-way data passing, the Hardware-software interface is straightforward, the data and control signals are passed using an Avalon bus. Since we will be implementing real-time object detection, we will use h2f_axi_master to transport data ensuring data throughput. Each pixel has 8*3 bits (8 bits for each of the 3 channels). Each data transmission will contain 256 bits, or 32 pixels to fully utilize the data width of axi_master.

References

[1] D. T. Nguyen, T. N. Nguyen, H. Kim and H. Lee, ”A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection,” in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 8, pp. 1861-1873, Aug. 2019, doi: 10.1109/TVLSI.2019.2905242.

[2] D. T. Nguyen, H. Kim, H. -J. Lee and I. -J. Chang, ”An Approximate Memory Architecture for a Reduction of Refresh Power Consumption in Deep Learning Applications,” 2018 IEEE International Symposium on Circuits and Systems (ISCAS), 2018, pp. 1-5, doi: 10.1109/ISCAS.2018.8351021.

A FPGA accelerator for YOLO CNN based on weight quantization and data flow optimization

Botong Xiao, Terry Tingrui Zhang, Haoran Jing

1 Introduction

DEEP learnging has been the most prevalent method in various task in computer vision due to the support of powerful computation devices such as GPU. Among the stateof-the-art methods, YOLO and the subsequent Sim-YOLO, YOLO-V2 demonstrate the most promising trade-off between speed and accuracy. While GPU is widely used for the training and inference of deep learning algorithms such as YOLO, recent researches have proved its inefficiency in optimization such as the the quantization of weight/activation and data access schedule. For instance, [2] has demonstrated that the training and inference of CNNs can be quantized to a very low-bit precision with insignificant loss in accuracy. This quantization enables a fast , memory efficient, and power efficient FPGA accelerator.

2 Software

The software implementation of the project mainly contains a simulation of the hardware part in the software domain. In this section, we will first introduce the convolution neural network architecture structure. Then, we will dive deep into the convolution layer and the max-pooling layer. Moreover, we will illustrate the quantization and de-quantization method that we used in the software simulation to better mimic the hardware behavior.

2.1 CNN architecture

The table below shows the whole network structure.

```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024
   13 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024
   14 conv    125  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 125
   15 region
```

The following software realizes the structure above.

```
struct network{
    // input dimension parameter
    int width;
    int height;
    int channels;
    // pointer to each layer
    layer *layers;
    // pointer to calculation result
    float *output;
    float *input;
    int8_t *int8_output;
    int8_t *int8_input;
}

struct layer{
    // dimension parameter
    LAYER_TYPE type;
    int size_in;
    int size_out;
    ....
    // parameter to draw output boxes
    int classes;
    ....
    // quantization parameter
    float amax_w;
    ....
    // pointer to data and calculation output
    int8_t *int8_weights;
    int8_t *int8_biases;
    int8_t *int8_scales;
    int8_t *int8_rolling_mean;
    int8_t *int8_rolling_variance;
    int8_t *int8_output;
    ....
   // forward function pointer
   void (*int8_forward)   (struct layer, struct network);
};
```

The network struct contains pointers to its layers and each layer struct contains the type, size information, and the pointers to layer parameters that are fixed during inference. Weights, bias, scale, mean, and variance are all of type int8_t. These parameters are quantized during loading and the floating-point counterparts can be found on "https://pjreddie.com/darknet/yolov2/". The quantization method will be introduced in a later section. As we can see there are 16 layers in the network. The 15th region layer is used to draw the detected object boundary. Thus, we will not spend time discussing the region layer and focus on the convolution layer and max-pooling layer.

2.2 Convolution layer

There are several operation inside the int8_forward function of the convolution layer. First convolution is computed on weights that is the int8_weights pointer and the corresponding input. Then, we apply batch normalization to the convolution output using mean and variance in the int8_rolling_mean and int8_rolling_variance pointer. Batch normalization is used to set the mean and variance of each kernel output to 0 and 1. In addition, we apply scaling, bias and activation to the layer output. At the end of the forward function, de-quantization is apply to the int8_t output result. The code is in the "Code" section of the report.

2.3 Max-pooling layer

The forward function of the max-pooling layer is relatively simple, and is used to reduce the size of the output. In our CNN, max-pool of size 2×2 and a stride of 2 is used to reduce the dimension of the output by half. The code is in the "Code" section of the report.

2.4 Quantization and De-quantization

Quantization and de-quantization are the most challenging part of software implementation and requires fine-tunning of the quantization parameter. The following process shows how to quantize one 32-bit floating point number to an 8-bit integer. Let w be a floating point number and let a_w be an positive float number that $w \in [-a_w , a_w]$, then the quantized int8 value w_q can be written as

$$w_q = \text{round}(\frac{w}{a_w} * 128)$$

De-quantization is not as straightforward as quantization, since de-quantization are applied at the end of the computation. Let w be a weight in floating point and let a_w be an positive float number that $w \in [-a_w , a_w]$. Let x be the input data in floating point and let a_x be an positive float number that $x \in [-a_x , a_x]$. Let m be the mean in floating point number and let a_m be an positive float number that $m \in [-a_m , a_m]$. Let v be the variance in floating point and let a_v be an positive float number that $v \in [-a_v , a_v]$. Then the quantized convolution computation with batch normalization can be written as

$$\frac{w_q \times x_q - m_q}{v_q} = \frac{wx}{v} \times \frac{128\, a_v}{a_w a_x} - \frac{m}{v} \times \frac{a_v}{a_m}$$

If

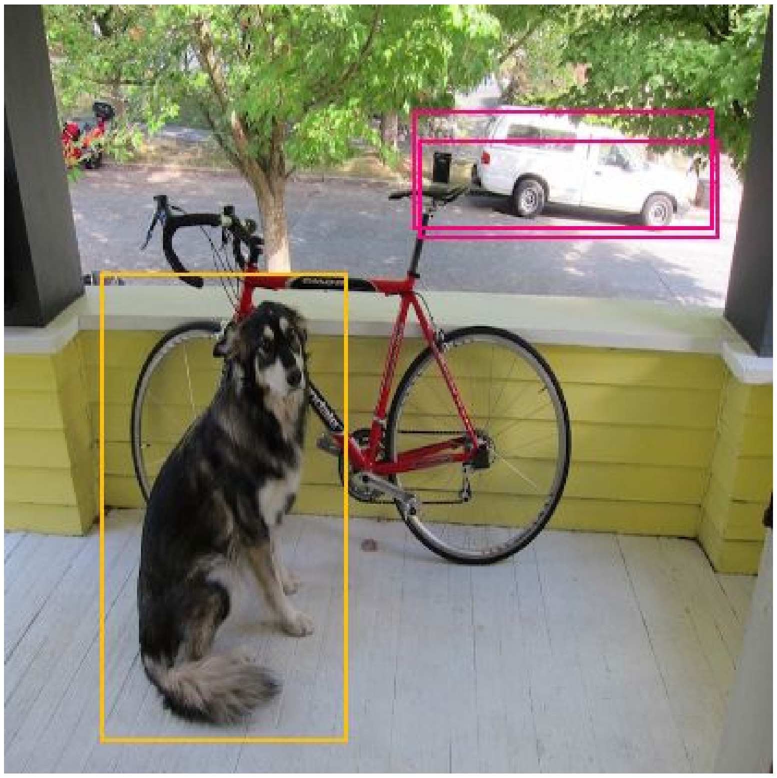
$$\frac{128\, a_v}{a_w a_x} = \frac{a_v}{a_m}$$

Then the de-quantized value can be simply written as

$$\frac{wx - m}{v} = \frac{w_q \times x_q - m_q}{v_q} \times \frac{a_m}{a_v}$$

However, the choosing the quantization parameter a_w, a_x, a_m, a_v can be quite intricate. As the value of these parameter increases, there precision of the quantized CNN can suffer, since the 8-bit integer needs to represent a larger range of floating point number.

2.5 Result



3 Hardware

The hardware implementation of the YOLO CNN accelerator mainly consists of three components: Circular Buffer, Convolution layer and Max-pooling layer. The implementation details will be illustrated below.

3.1 Circular Buffer

The function of the circular buffer is to partially store the input images and the weight parameter needed in the convolution layer. The way of streaming input images into the circular buffer is shown in the figure. The first row of TI layers, which is 4 in this example, will be stored in the first row of the circular buffer, shown as red blocks in the figure. Then it will move to the next TI layers of input images until all the first row of the input channels of input images are stored. Then, it will move to the second row and repeat the operation. After storing three rows of circular buffer, the host can initiate the output of data and launch the convolution layer because it is sufficient for the convolution layer to perform the computation. The circular buffer will output 9 input image data as well as 9 weight parameters in parallel, and feed them into the convolution layer. As data streams out of the circular buffer, the address pointer will automatically go to the next row. At the same time, the circular buffer can still be stored with new data concurrently, which increases the efficiency. This is also the reason why it is called a "circular" buffer, because it can keep storing data and streaming out data simultaneously. Actually, its working principle is very similar to an asynchronous FIFO. However, one thing worth mentioning is that the data structure inside the circular buffer is a little bit different. One buffer for the same address is designed to fit a TI*6 bit data. The data structure arranged in this way is very beneficial for sending out the data to the convolution layer, although it may be a little bit difficult for storing input images into the circular buffer. The Avalon Bus only can support up to 32 bit bandwidth, and the most common value for the TI is 8 and 16, which means that it cannot be passed through the Avalon Bus in the same cycle. Therefore, the buffer has to be stored multiple times to fit one buffer storage for the same address.

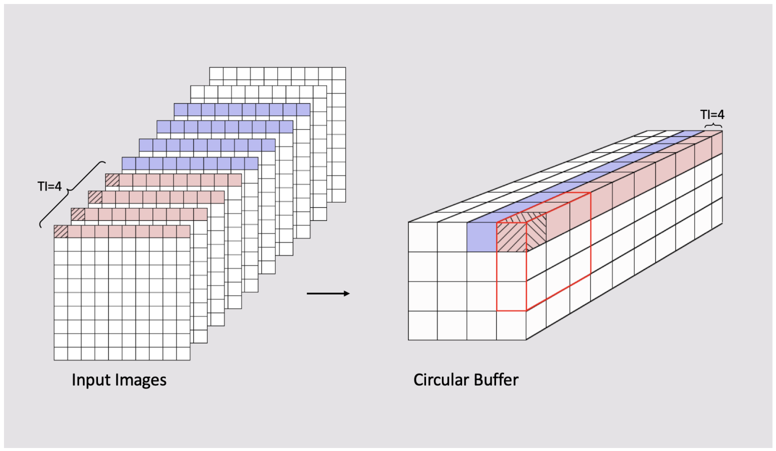


Figure 1: Circular buffer

3.2 Convolution layer

The architecture of the convolution layer is shown in the figure. When the 9 input image data as well as 9 weight parameters, which is shown as the 3*3 kernel in the figure, are input to the convolution layer, they will first go through a two stage pipelined ternary adder tree. Then the intermediate computation results will be stored in the line buffer. After computing for TI times, the accumulation for one data in the output layer is temporarily done, and the data will be stored in the next address in the line buffer. After computing for one row, it will go back to the first data of this row with next TI layers, and then it will accumulate again with the intermediate data stored in the line buffer. Therefore, the depth of the line buffer should be the same as the input image size. After completing all the computations for one row, the data will be simultaneously streamed into the batch normalization part inside the convolution layer. The batch normalization is quite easy to implement inside the hardware because it only requires the multiplication and accumulation operation. And then it will go to the RELU part, which performs the function that remains the same value for the positive data and divided by 8 for the negative data. And dividing by 8 can be easily implemented as shifting three bits in the hardware, which is quite easy to achieve. Eventually, it will also perform the quantization before eventually sending out the data. The quantization function in the hardware design is to remain the value if the data value stays within the specific range. And if it is out of range, it will just remain the maximum or the minimum value of the range instead of the original value. The output data eventually will either be sent back to the host or next to the max-pooling layer.

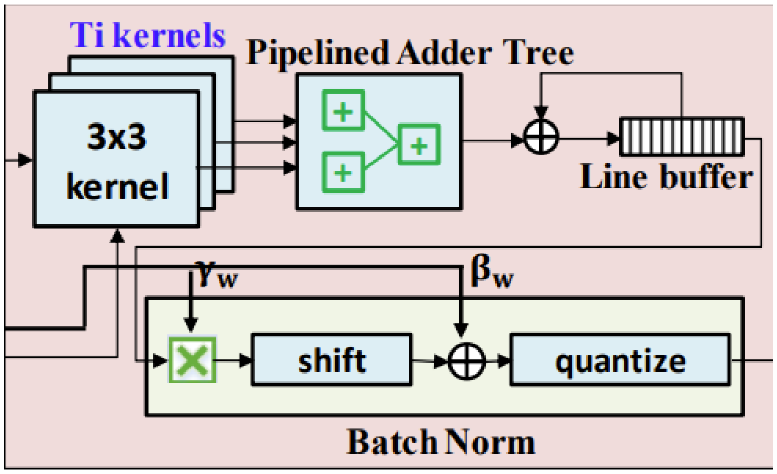


Figure 2: Convolution layer

3.3 Max-pooling layer

The architecture of the max-pooling layer is shown in the figure. All the max-pooling in the YOLO CNN network is all 2*2 max-pooling. When the data streams in the max-pooling layer, it will be controlled by a ping pong control signal. When the first data comes in, the ping pong signal is low and the data will be first stored in a register for one cycle. And in the next cycle, the ping pong control signal will be set to 1, and then, instead of being stored in the register, it will compare with the data stored in the register. The larger one of the comparisons will be stored in the line buffer. The comparison will keep being performed until it compares all the data in one row. Because the data are compared in pairs and only the larger one will be stored in the line buffer, the depth of the line buffer only requires half of the input image size. After feeding one row, the row signal will be set to 1, and the data in row 1 will also repeat the step mentioned before. But differently, after comparison, the larger data will not be stored in the line buffer. Instead, they will compare with the corresponding data stored in the line buffer, and eventually output the larger one, completing the max-pooling function.

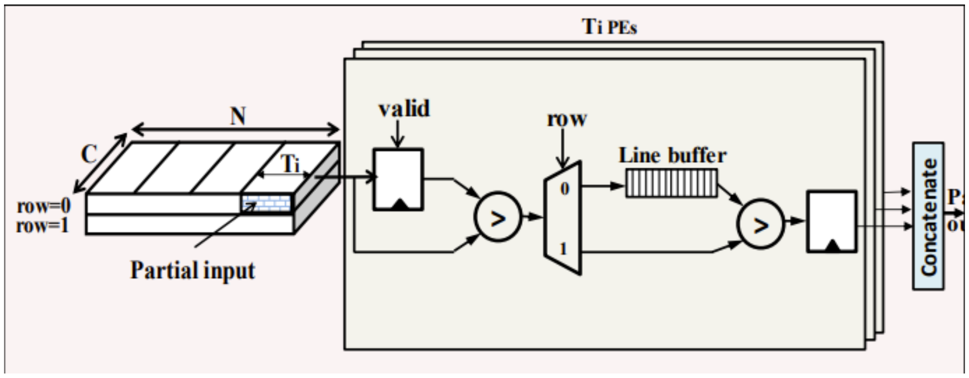


Figure 3: Max-pooling layer

3.4 Simulation results

To accommodate the different configuration of different layers, such as the input size of the image or the TI parameter, all the components have been designed with configurable parameters, which ensures that it can be easily modified and meet the requirements of different layers. In addition, all the components designed above have been verified through the simulation. The simulation results of circular buffer, convolution layer and max-pooling layer are shown below. For being easily implemented, the circular buffer design has been connected to the convolution layer, which can directly perform the convolution computation without additional control.

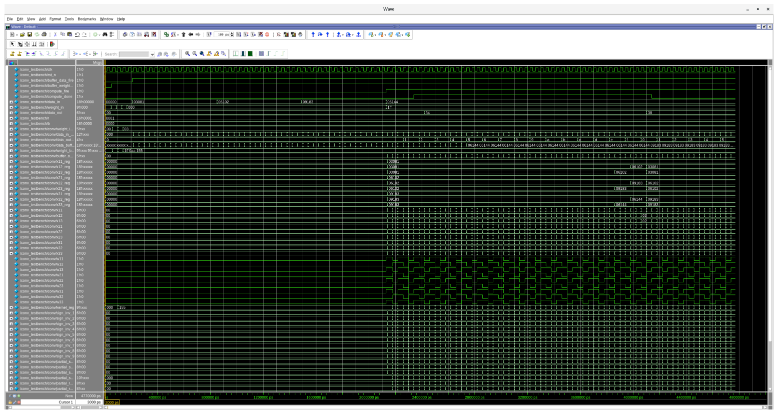


Figure 4: Simulation result for convolution layer

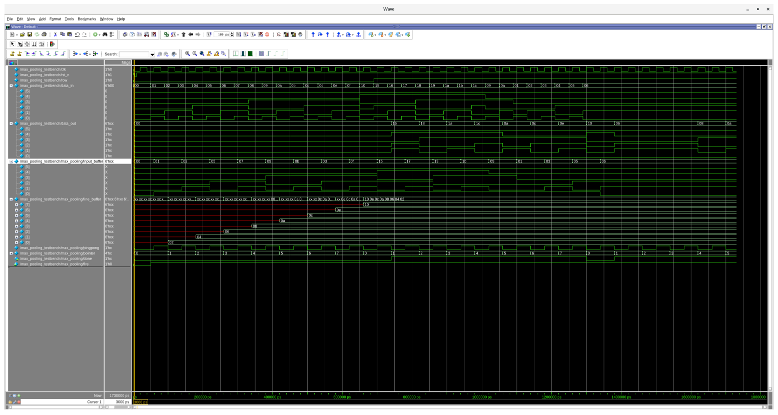


Figure 5: Simulation result for max-pooling layer

3.5 Challenge

The most important challenges are the limitation of the SRAM storage and the limited bandwidth of the Avalon Bus. The SRAM storage of the FPGA only has 0.5 MB. However, the whole weight parameters may require about 2MB On-chip SRAM, which means that it cannot fit all the parameters and the whole computation for all layers cannot be completed in the FPGA. Therefore, we have to perform the computation layer by layer, with frequently off-chip memory accessing, which no doubt largely increases the latency.

Another challenge is the limited bandwidth of Avalon Bus. Because of the limited bandwidth, we are unable to send all the data required by the convolution layer in parallel, which made me design a separate circular buffer to solve the problem. However, even with the circular buffer, it still needs multiple times to store the data into the buffer, which introduces some challenges.

4 Software Hardware Interface

Finishing the hardware and software design, it is also important to connect and transfer data between hardware and software parts, making it simple to access. One of the approaches we have designed is using DMA , a direct memory access controller between hardware on-chip memory and software SDRAM to make them communicate to each other. The pipeline block diagram is shown as figure below.

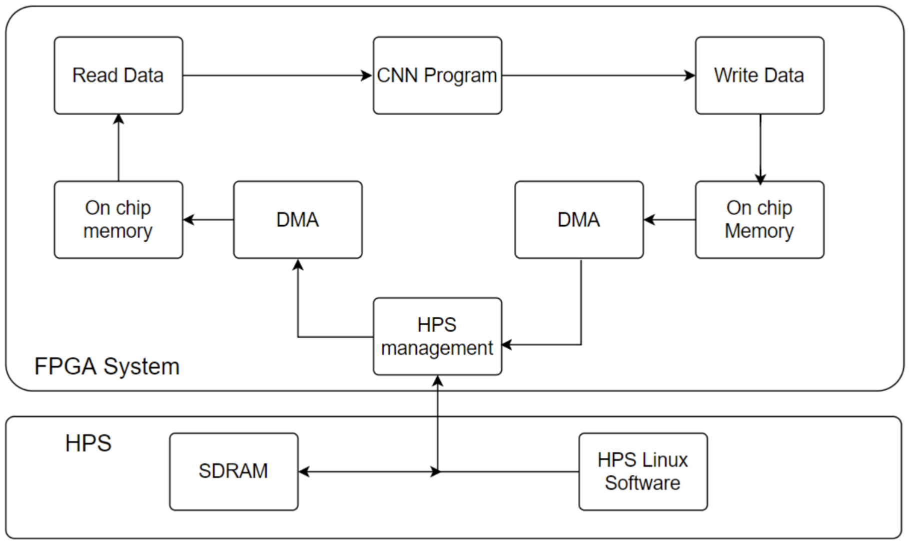


Figure 6: Pipeline interface system block diagram

The whole pipeline project is working two separate parts of the DE1-SoC board and making them communicate with each other, so it contains hardware and software parts. The software part runs on the Linux kernel in HPS through SD card, where the input data will also be stored in. The software will transfer the input data into SDRAM, then it will activate the HPS management, which is also the DMA(Direct Memory Access) controller, sending data directly to the on-chip memory by DMA. When the transmission is over, the software will send a signal to activate the hardware part, which contains read and write blocks. Read block will read data from on-chip memory to the CNN program, while the write block will write the data back to on-chip memory, which could be accessed by Linux software through DMA transmission. The logic of those two blocks is shown in Figure2.

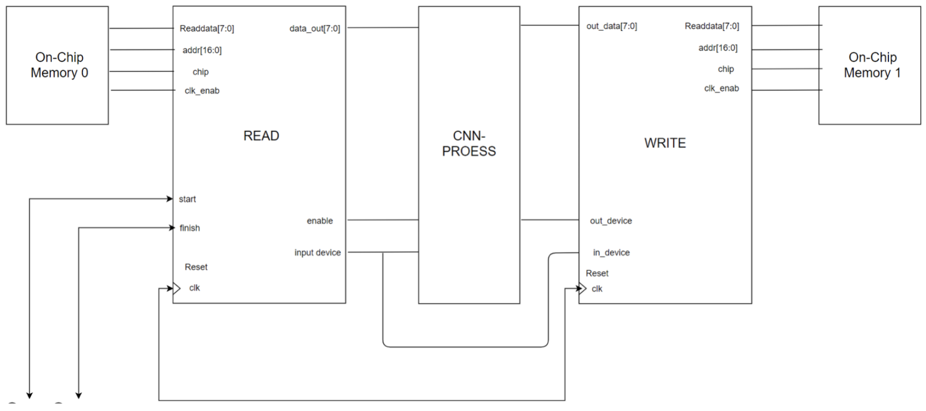


Figure 7: Read and write block logic

As we can obtain from the figure, the whole FPGA part is controlled by start and finish signals. When the start signal activates, the read block will start to read data from the on chip memory using the address and chip select. When reading is over, the input device will activate, which will inform the CNN Process, which is the algorithm block, to prepare to receive data. Then the read data will go through data_out port and enable the CNN block to process the data while the read block itself repeats the states above. After processing, the CNN block sends out the output_device signal, together with input_device signal, causing the write block to get into the next writing stage, waiting for data to come in. The incoming data from out_data will be sent to writedata port, storing the data into the on-chip memory1, which could be accessed by the DMA between SDRAM and on-chip memory, while the write block transferred to the original state. One thing that should be mentioned is that in order to make the whole block could only be controlled by start, finish and reset signal, those 2 blocks are designed to be a state machine, making it process data continuously. Each block has 5 states, with 1 idle state, 1 reset state, 1 preparation state, 1 process state and 1 stop state.

For the software part in Linux, DMA plays an important role as an intermediate between software SDRAM and hardware on-chip memory, handling the actual data transfer between FPGA and HPS. At first, the program will read the image to SDRAM by using open_memory and mmap function. Then we need to activate the FPGA to SDRAM bridge and activate the DMA controller, then we could setup a pointer with address of SDRAM and on-chip memory, then read and write data between them using _DMA _REG_READ_ADDR function and DMA_REG_WRITE_ADDR function, which can be obtained from the code block below:

```
//create a pointer to the DMA controller base
    void *h2p_lw_dma_addr1 = NULL;
    h2p_lw_dma_addr1 = virtual_base + ( ( unsigned long  )( ALT_LWFPGASLVS_OFST + DMA_1_BASE ) & ( unsigned long)( HW_REGS_MASK ) );

    // clear the DMA control and status
    clearDMAcontrol(h2p_lw_dma_addr1);
    _DMA_REG_STATUS(h2p_lw_dma_addr1) = 0;
    _DMA_REG_READ_ADDR(h2p_lw_dma_addr1)  = 0;                  // read from OCM
    _DMA_REG_WRITE_ADDR(h2p_lw_dma_addr1) = physical_addr2;     // write to SDRAM (DDR3)
    _DMA_REG_LENGTH(h2p_lw_dma_addr1) = 4000;                   // number of elements in bytes
    //start the transfer
    _DMA_REG_CONTROL(h2p_lw_dma_addr1) = _DMA_CTR_BYTE | _DMA_CTR_GO | _DMA_CTR_LEEN;
```

To compile this into a project, we created a new quartus project, including the hardware files and the main file declaring the pins and connections to compile, while putting the software file onto the SD card off the board, but unfortunately we are unable to make it fully functional mainly because of the small on-chip memory.

5 Contribution

5.1 Terry Zhang

I wrote the software simulation for the Yolo-CNN including the CNN architecture and the parameter quantization.
The most important thing that I learned from this project is to think about the time limitation, and hardware resource limitations before jumping into something that sounds exciting. In the project, all of my teammates are in unknown territories with CNN and deep learning quantization. As a result, we spend a lot of time learning the material. In future projects, I think it will be best to start with something fundamental and add on to it, as opposed to starting with something grand and resulting in something unsatisfictory.

5.2 Botong Xiao

All the hardware part design is completed independently by Botong Xiao, including all the verilog codes, testbenches as well as all the hardware parts specifications in each report.
From this project, I realize the importance of first figuring out the limitations of resources, because resource limitations such as bandwidth and SRAM storage will not only affect the performance and the area cost of the design, it may also affect the design specification as the dataflow and computation algorithm. Therefore, it leads me to treat the resource limitations from a completely

different perspective.

5.3 Haoran Jing

Haoran Jing is in charge of the interface part between hardware and software, which contains hardware interface codes, software interface codes and quartus project creation. However, this group member failed to make the software part work.
This group member realized the importance of the interface between hardware and software, and found DMA for interconnection although it does not work properly. In future projects, this group member will be more focusing on the efficiency of the interface, which could significantly affect the data transmission, writing redundant code is hard to implement and test. Moreover, memory management is also a important part when it comes to FPGA design.

6 Code

Convolution:

```
for (int oc = 0; oc < output_channel; ++oc) {
// for each output channel
        output_channel_offset_kernel = oc * ksize * ksize *
    input_channel;
        output_channel_offset_out = oc * output_size * output_size;
        for (int ic = 0; ic < input_channel; ++ic) {
        // for each input channel
            input_channel_offset_kernel = ic * ksize * ksize;
            for (int r = 0; r < output_size; ++r) {
            // for each row
                row_offset_out = r * output_size;
                for (int c = 0; c < output_size; ++c) {
                // for each col
                    col_offset_out = c;
                    for (int i = 0; i < ksize; ++i) {
                    // for row in kernel
                        row_offset_kernel = i * ksize;
                        for (int j = 0; j < ksize; ++j) {
                        // for col in kernel
                            col_offset_kernel = j;
                            int8_t int8_kernel_value =
                            int8_weights[col_offset_kernel +
                                        row_offset_kernel +
                                        input_channel_offset_kernel +
                                        output_channel_offset_kernel];
                            int8_t int8_image_value =
                            int8_get_input_pixel(r, c, ic, i, j, pad,
                            input_size, int8_input);
                            // update output data
                            inter_out[col_offset_out +
                                row_offset_out +
                                output_channel_offset_out] +=
                                ((int8_kernel_value) *
                                (int8_image_value));
                    }
                }
            }
        }
    }
}
```

Batch Normalization:

```
 for (int oc = 0; oc < out_channel; ++oc) { // for each output channel
    int channel_offset = oc * size_out * size_out;
    for (int r = 0; r < size_out; ++r) { // for each row
        int row_offset = r * size_out;
        for (int c = 0; c < size_out; ++c) { // for each col
            int col_offset = c;
            int index = col_offset + row_offset + channel_offset;
            output[index] = (output[index] - rolling_mean[oc]) / (
    rolling_variance[oc]);
        }
    }
 }
```

Apply Scale:

```
for (int oc = 0; oc < out_channel; ++oc) {
    int channel_offset = oc * size_out * size_out;
    for (int r = 0; r < size_out; ++r) {
        int row_offset = r * size_out;
        for (int c = 0; c < size_out; ++c) {
            int col_offset = c;
            output[channel_offset + row_offset + col_offset] *= scales[
    oc];
        }
    }
}
```

Apply Bias:

```
for (int oc = 0; oc < out_channel; ++oc) {
    int channel_offset = oc * size_out * size_out;
    for (int r = 0; r < size_out; ++r) {
        int row_offset = r * size_out;
        for (int c = 0; c < size_out; ++c) {
            int col_offset = c;
            output[channel_offset + row_offset + col_offset] += biases[
    oc];
        }
    }
}
```

Apply Activation:

```
static inline float leaky_activate(float x) { return (x > 0) ? x :
    0.125 * x; }
```