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

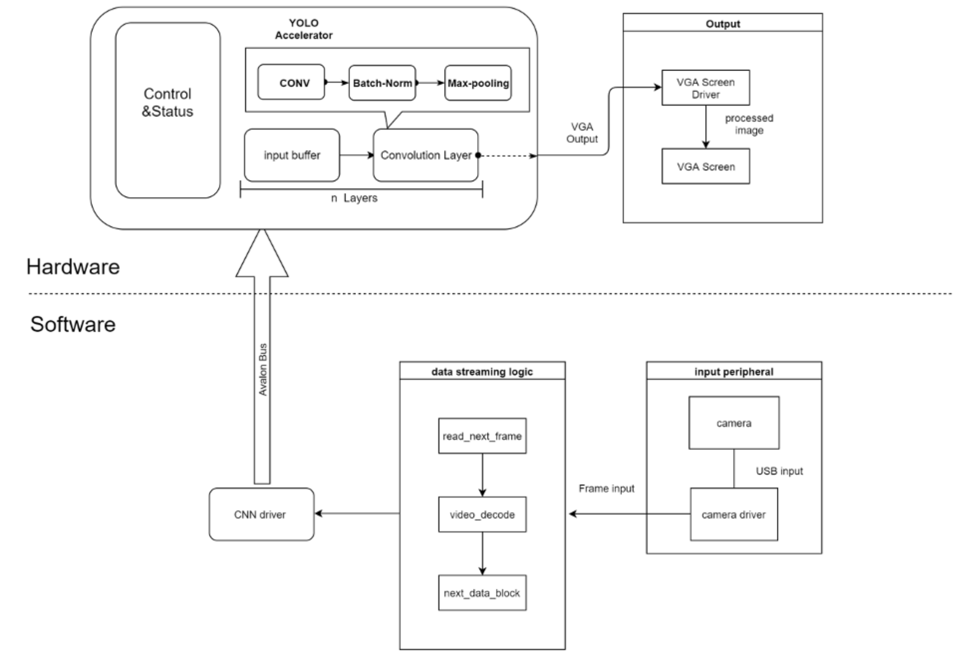


Fig. 1: Block diagram

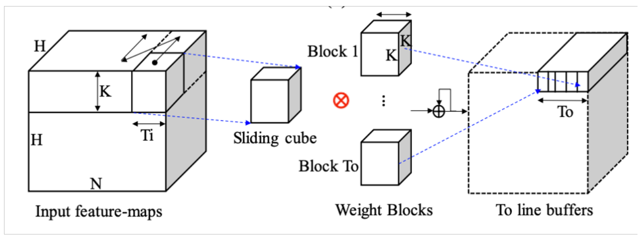


Fig. 2: Data flow

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.

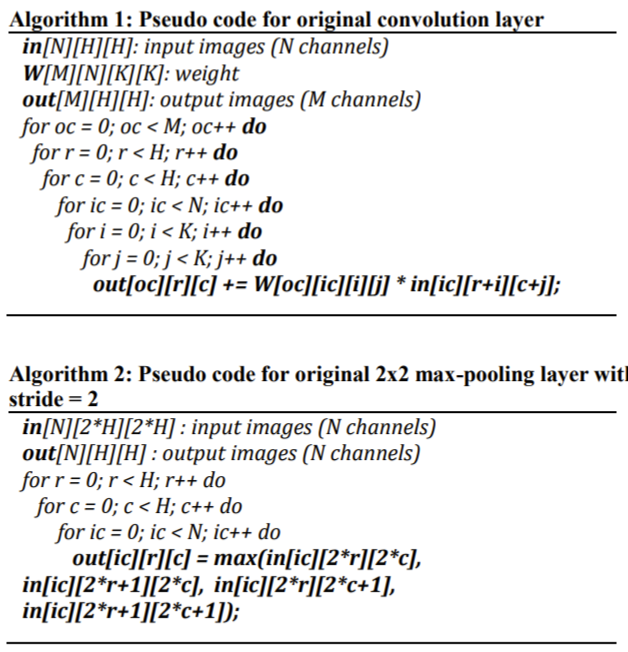
III. Algorithms

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.