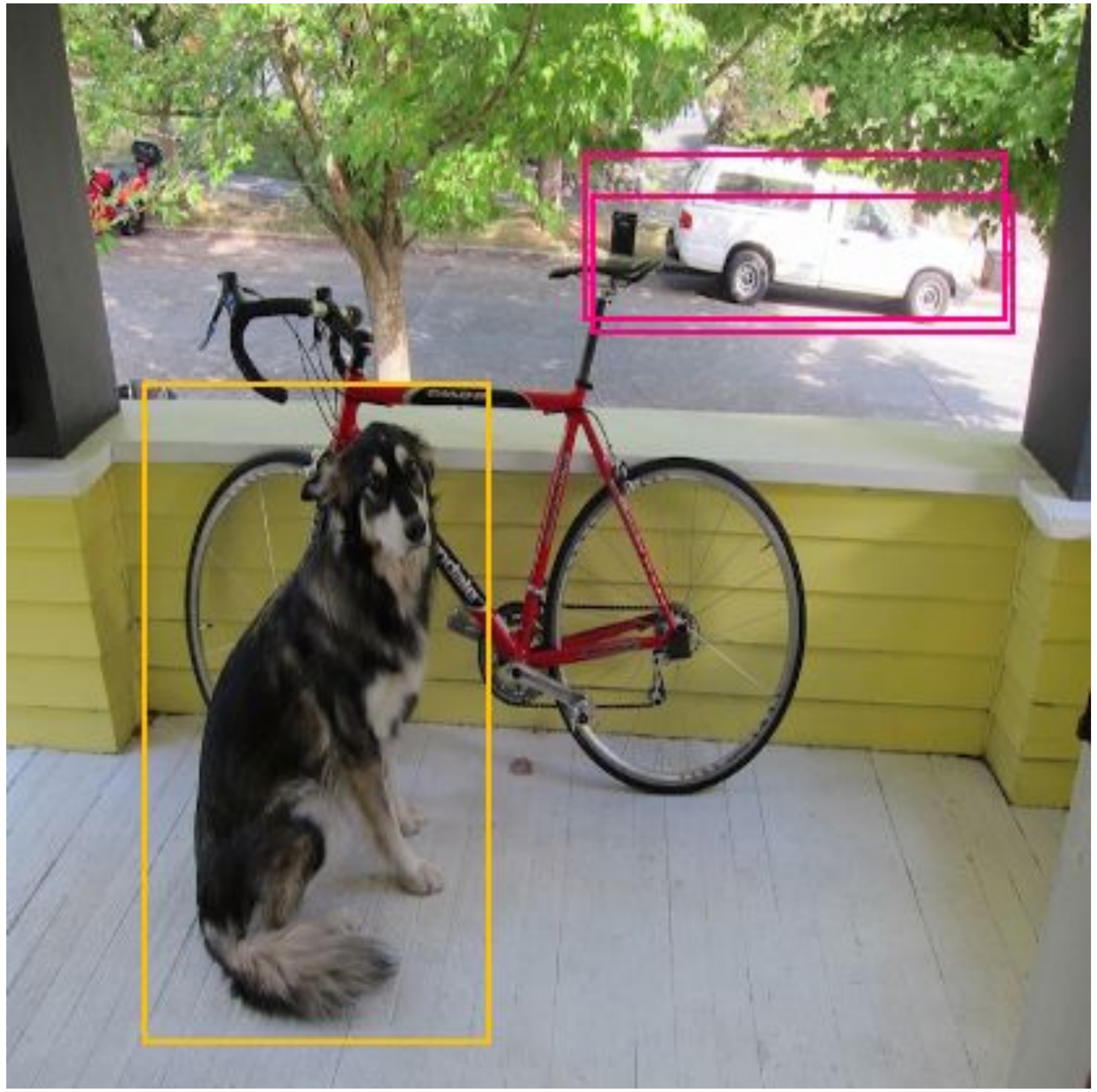
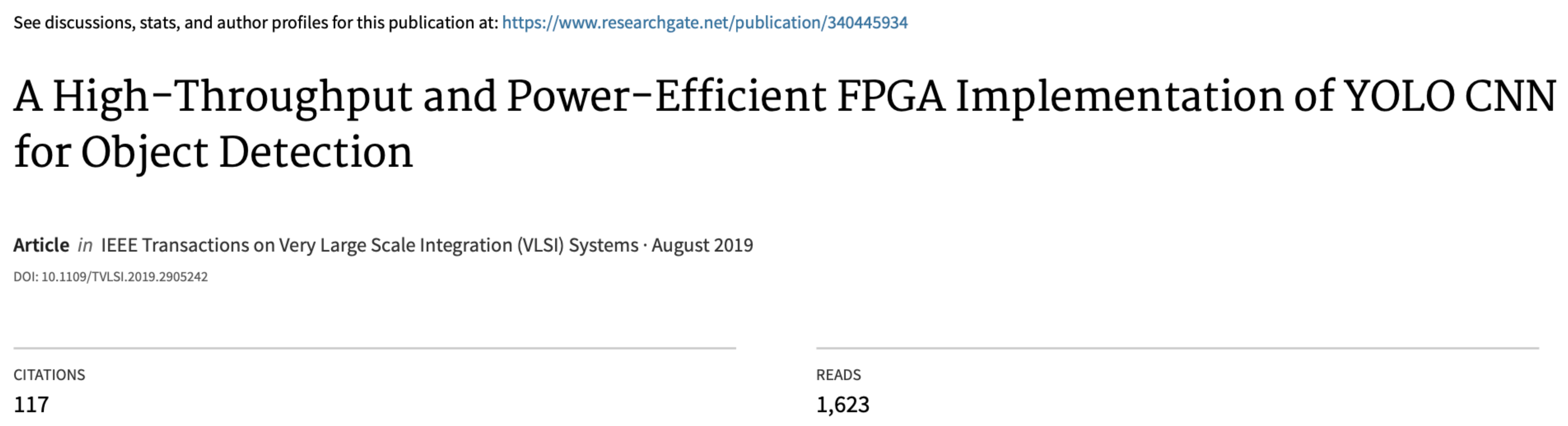
FPGA Accelerator for Yolo-CNN



Botong Xiao, Terry Tingrui Zhang, Haoran Jing

Background



A High-Throughput and Power-Efficient FPGA Implementation of YOLO CNN for Object Detection

Article *in* IEEE Transactions on Very Large Scale Integration (VLSI) Systems · August 2019

DOI: 10.1109/TVLSI.2019.2905242

CITATIONS	READS
117	1,623

Software

CNN architecture

Quantization parameter tuning

Network Structure

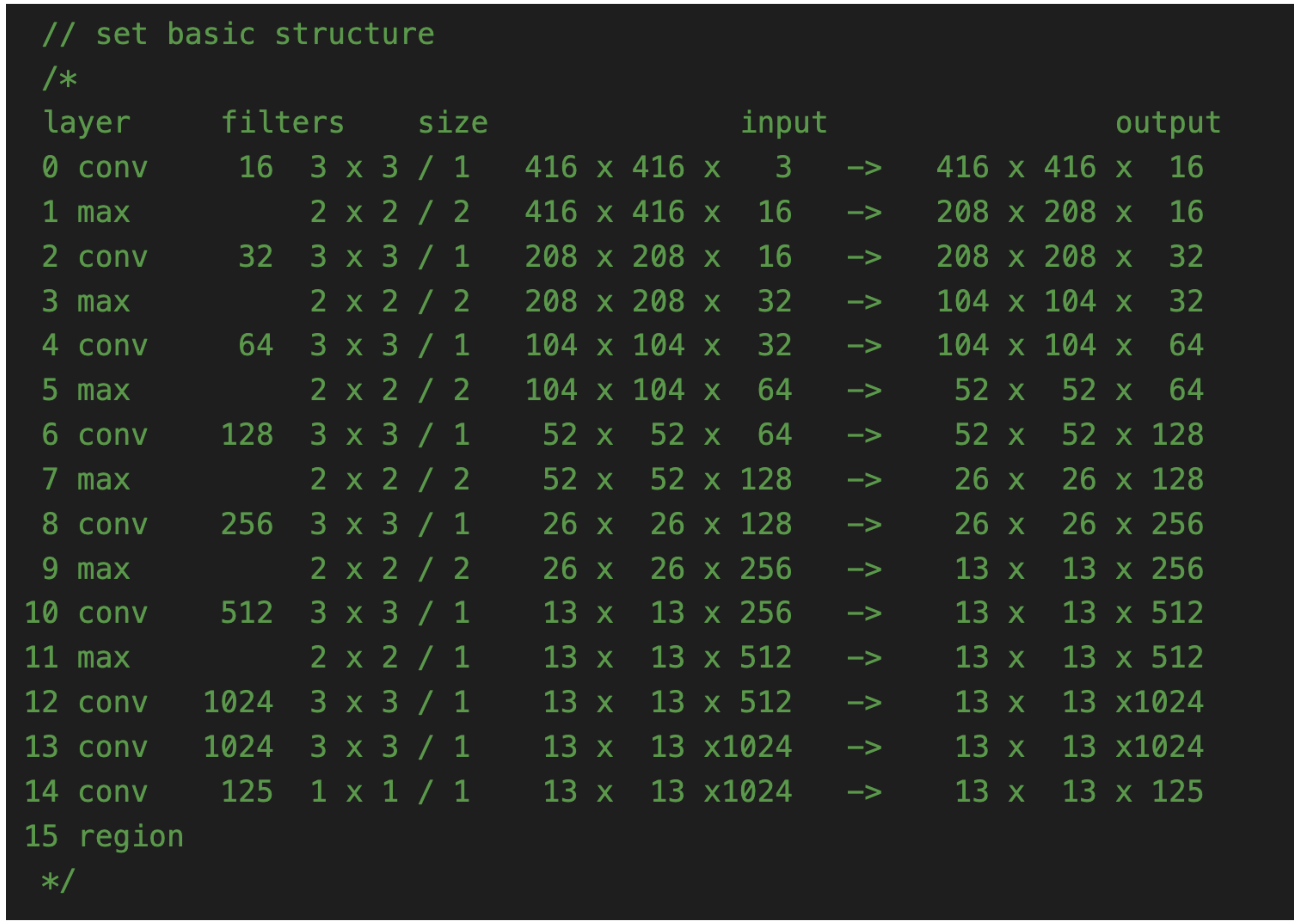
```
struct network{
    // input dimension parameter
    int width;
    int height;
    int channels;
    // pointer to each layer
    layer *layers;
    // pointer to calculation result
    float *output;
    float *input;
    int8_t *int8_output;
    int8_t *int8_input;
}
```

Fp32 weight: https://pjreddie.com/darknet/yolov2/

```
struct layer{
    // dimension parameter
    LAYER_TYPE type;
    int size_in;
    int size_out;
    int num_ouput_channel;
    int num input channel;
    int kernel_size;
     ......

    // parameter to draw output boxes
    int classes;
     ......

    // quantization parameter
    float amax_w;
     ......

    // pointer to data and calculation output
    // data are quantized during loading
    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);
};
```

Inference

```
for (int i = 0; i < net->num_of_layer; ++i)
{
        cur_layer = net->layers[l];
        // loop each layer
        cur_layer.int8_forward(cur_layer, *net);
        net->int8_input = cur_layer.int8_output;
        net->input = cur_layer.output;
}
```



```
 // set basic 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
 */
```

Inside the Forward Function

Convolution Layer

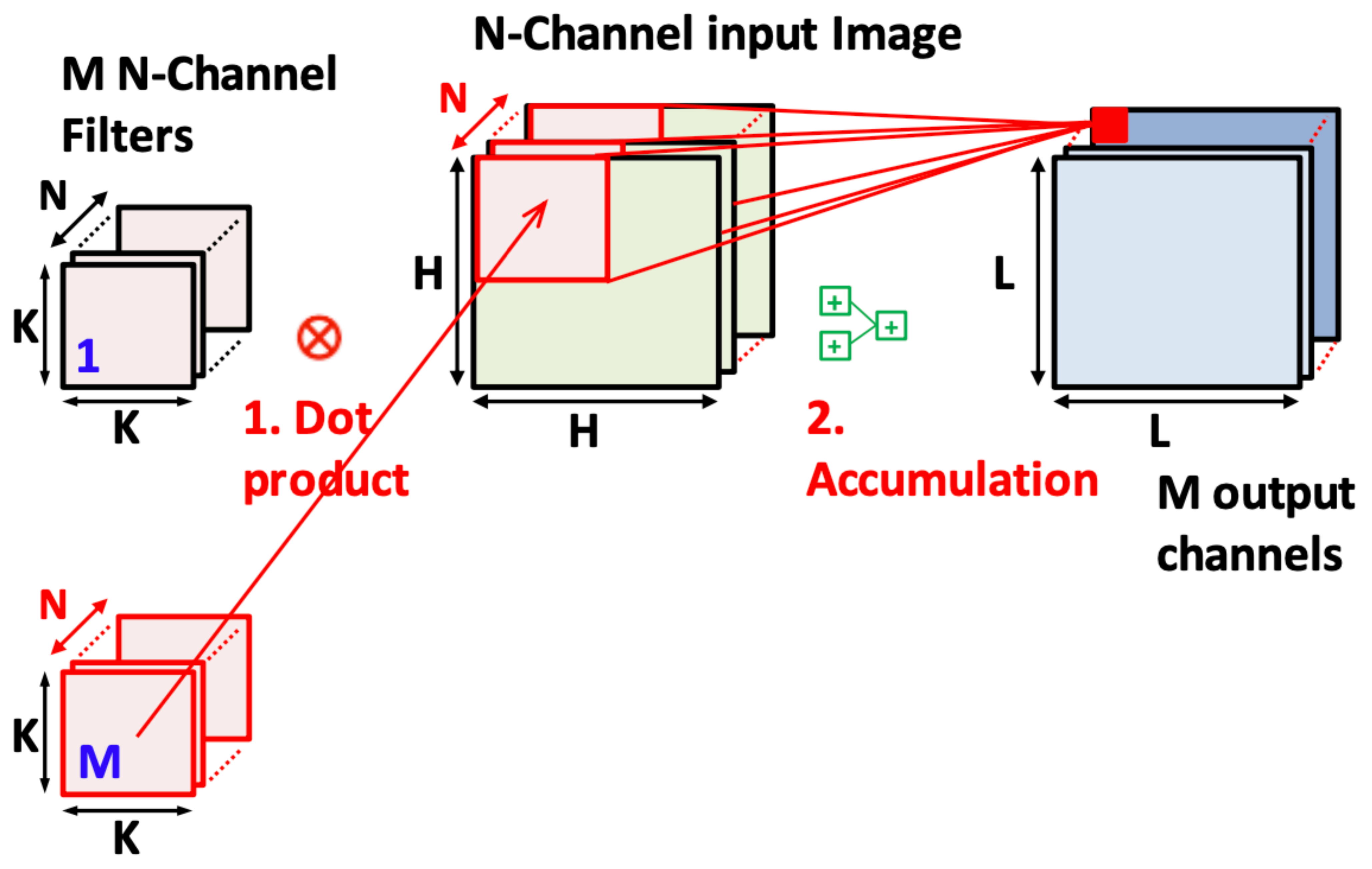
- Convolution (int8)
- Batch normalization (int8)

  //Set the mean and variance of each kernel output to 0 and 1
- Apply Scaling (int8)

  // make each kernel output have an equal weight
- Apply Bias (int8)
- Activation (int8)
- Apply Dequantization

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;
                  // get one kernel weight 8bit
                  int8_t int8_kernel_value = int8_weights[col_offset_kernel +
                                                         row_offset_kernel +
                                                         input_channel_offset_kernel +
                                                         output_channel_offset_kernel];
                  // get one input data 8bit
                  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; }
```

Apply Dequantization

- Int8 quantization of one float number

- Dequantization of the convolution result

Int8 quantization

Let w be a float weight . Let x be a float input . Let m be a float mean . Let v be a float variance .

Find a_w that $w \in [-a_w , a_w]$. Find a_x that $x \in [-a_x , a_x]$. Find a_m that $m \in [-a_m , a_m]$. Find a_v that $v \in [-a_v , a_v]$.

Then the quantized value can be found as below

$$w_q = \text{round}(w/a_w \times 128)$$

$$x_q = \text{round}(x/a_x \times 128)$$

$$m_q = \text{round}(m/a_m \times 128)$$

$$v_q = \text{round}(v/a_v \times 128)$$

Int8 quantization of one number

```
int8_t int8_quantize(float x, float amax, int bitnum)
{    //     x: value to be quantized;     x in [-amax , amax];       bitnum: target bit number
    int8_t xq, out_max, out_min;  float x_dq;
    out_max = pow(2, (bitnum - 1))-1; // for int8 out_max = 127
    out_min = -pow(2, (bitnum - 1)); // for int8 out_min = -128
    xq = round(x / amax * out_max);
    // clipping to prevent overflow
    xq = (xq > out_max) ? out_max : xq;
    xq = (xq < out_min) ? out_min : xq;
    return xq;
}
```

Dequantization

Result of Convolusion: $\dfrac{w_q \times x_q - m_q}{v_q} = \dfrac{wx}{v} \times \dfrac{128\, a_v}{a_w a_x} - \dfrac{m}{v} \times \dfrac{a_v}{a_m}$

If $\dfrac{128\, a_v}{a_w a_x} = \dfrac{a_v}{a_m}$

then the dequantized value equals

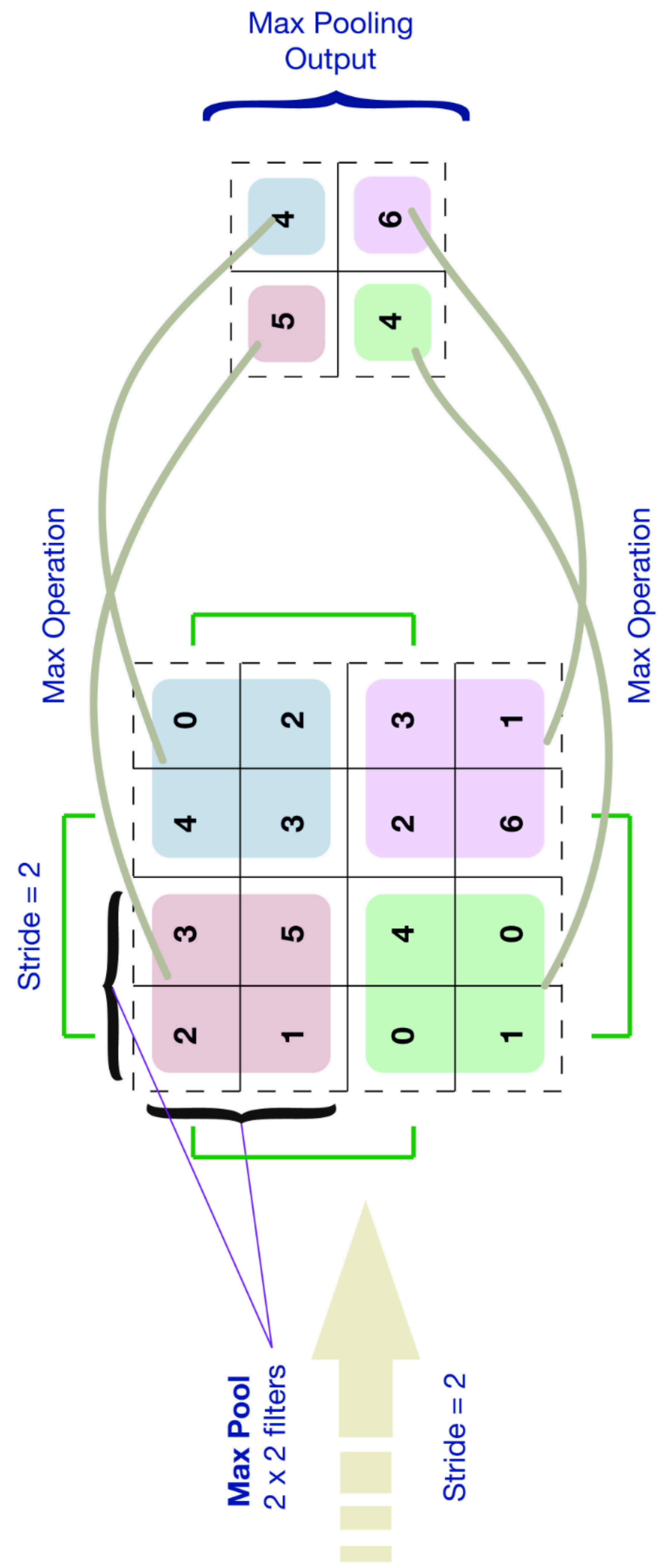
$$\frac{wx - m}{v} = \frac{w_q \times x_q - m_q}{v_q} \times \frac{a_m}{a_v}$$

Int8 dequantization of Convolution

```
int8_t int8_dequantize(int8_t x, float a_m, float a_v)

{

    Return (int)x*a_m / a_v;

}
```

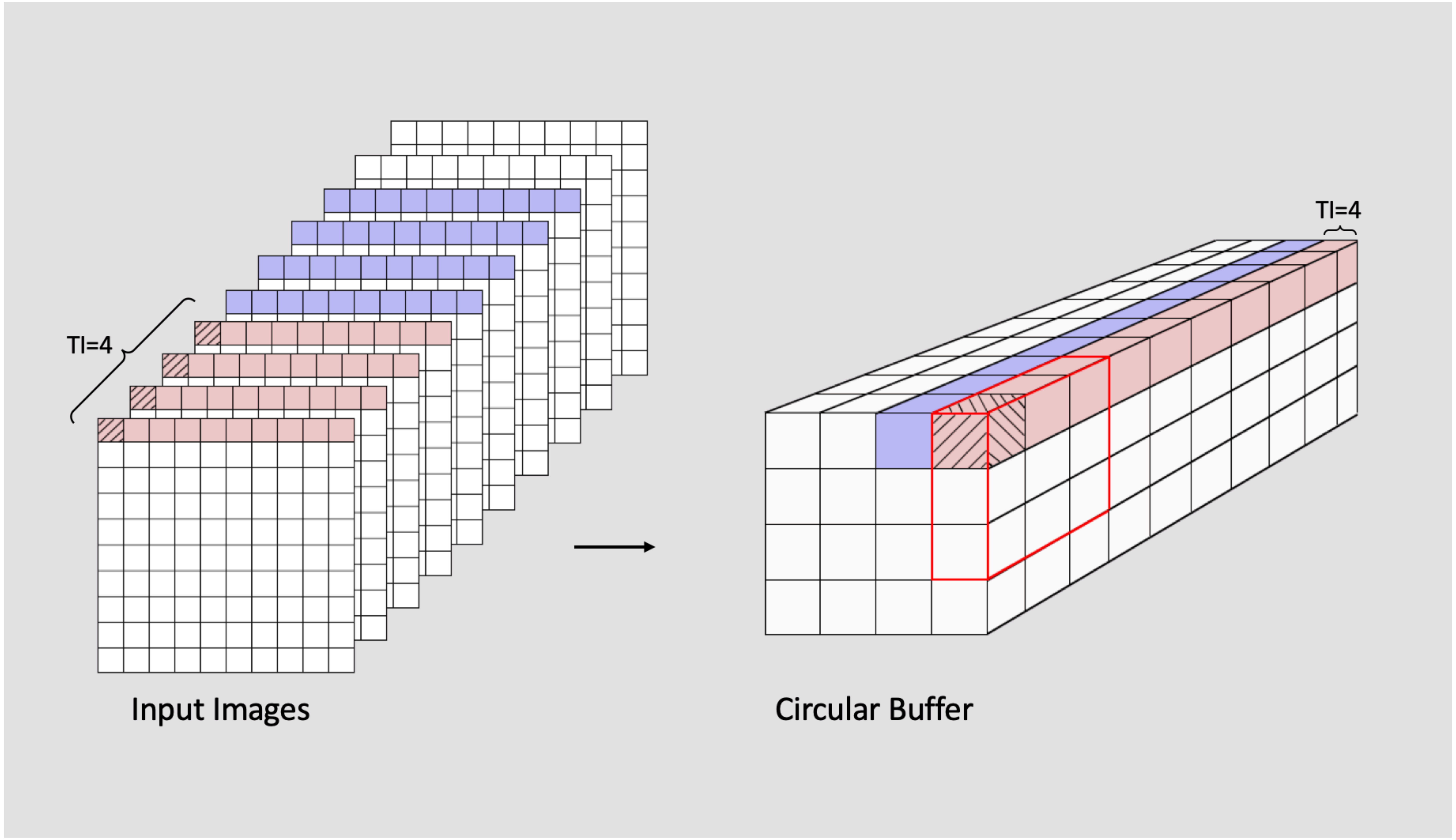
Maxpooling Layer

```
for (int ic = 0; ic < input_channel; ++ic) {
    int channel_offset_out = ic * size_out * size_out;
    int channel_offset_in = ic * size_in * size_in;
    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;
            int out_index = col_offset + row_offset + channel_offset_out;
            float max = -FLT_MAX;
            for (int h = 0; h < filter_size; ++h) { // for each row in filter
                int h_input = r * stride + h;
                for (int w = 0; w < filter_size; ++w) { // for each col in filter
                    int w_input = c * stride + w;
                    int index_in = w_input + h_input * size_in + channel_offset_in;
                    int valid = (h_input >= 0 && h_input < size_in &&
                                w_input >= 0 && w_input < size_in);
                    float input_val = (valid != 0) ? net.input[index_in] : -FLT_MAX;
                    max = (input_val > max) ? input_val : max;
                }
            }
            l.output[out_index] = max;
        }
    }
}
```

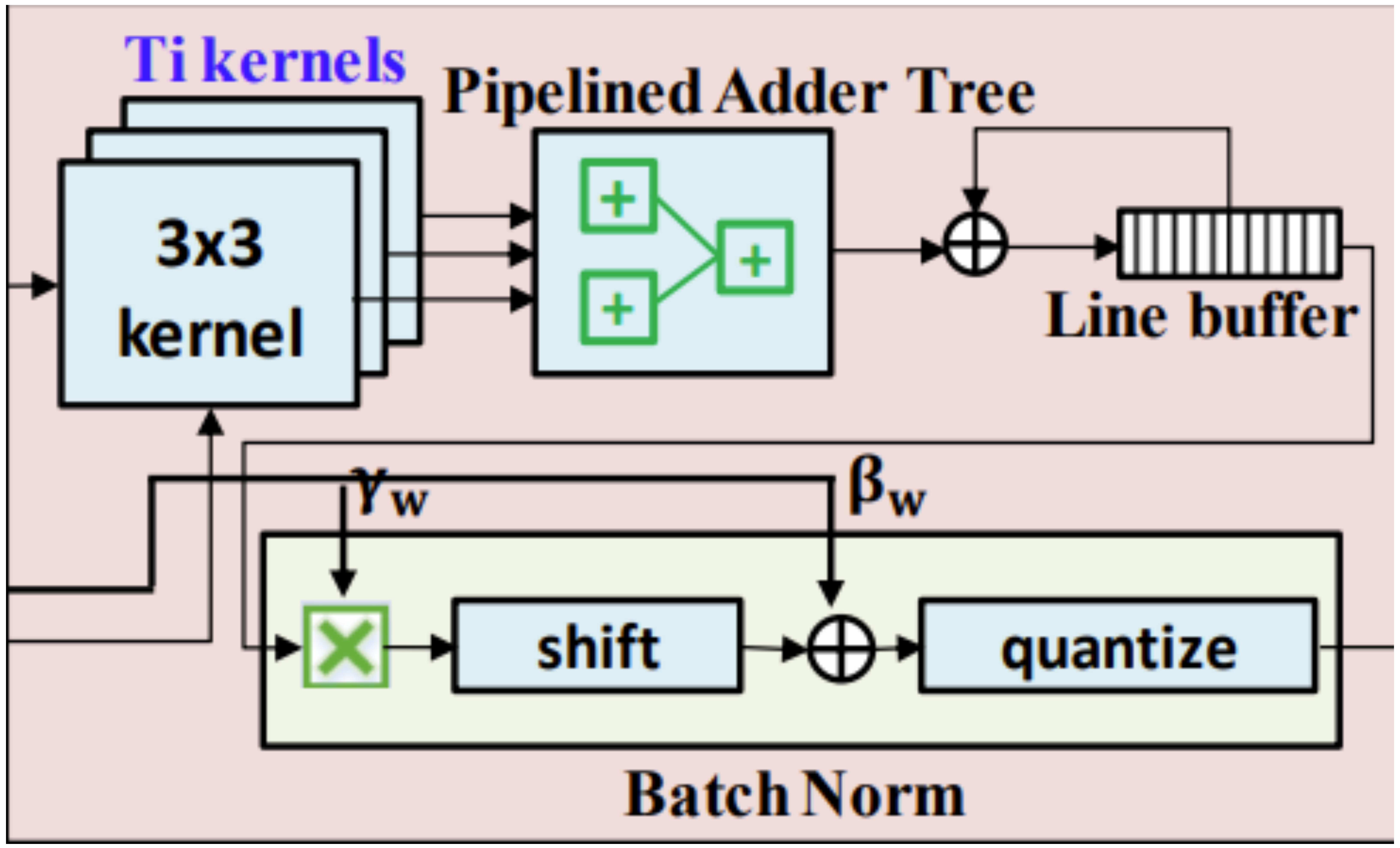


Hardware

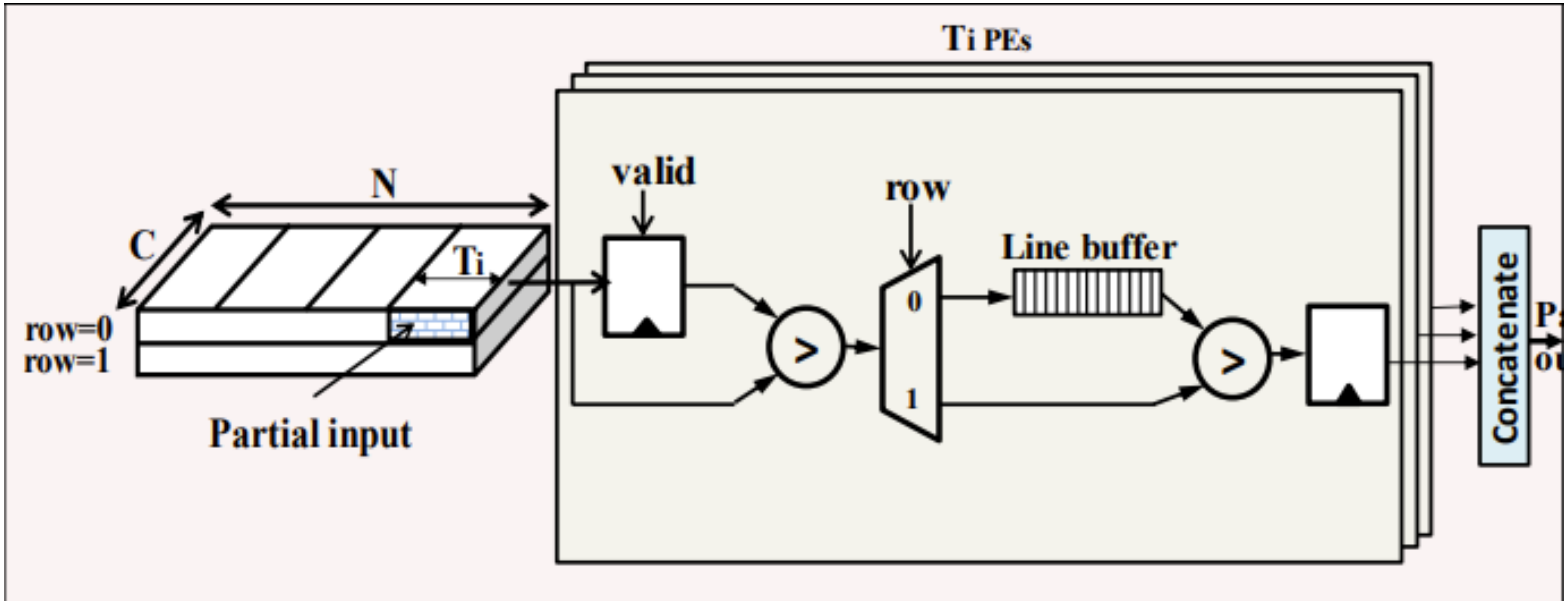
Circular Buffer



Convolution Layer

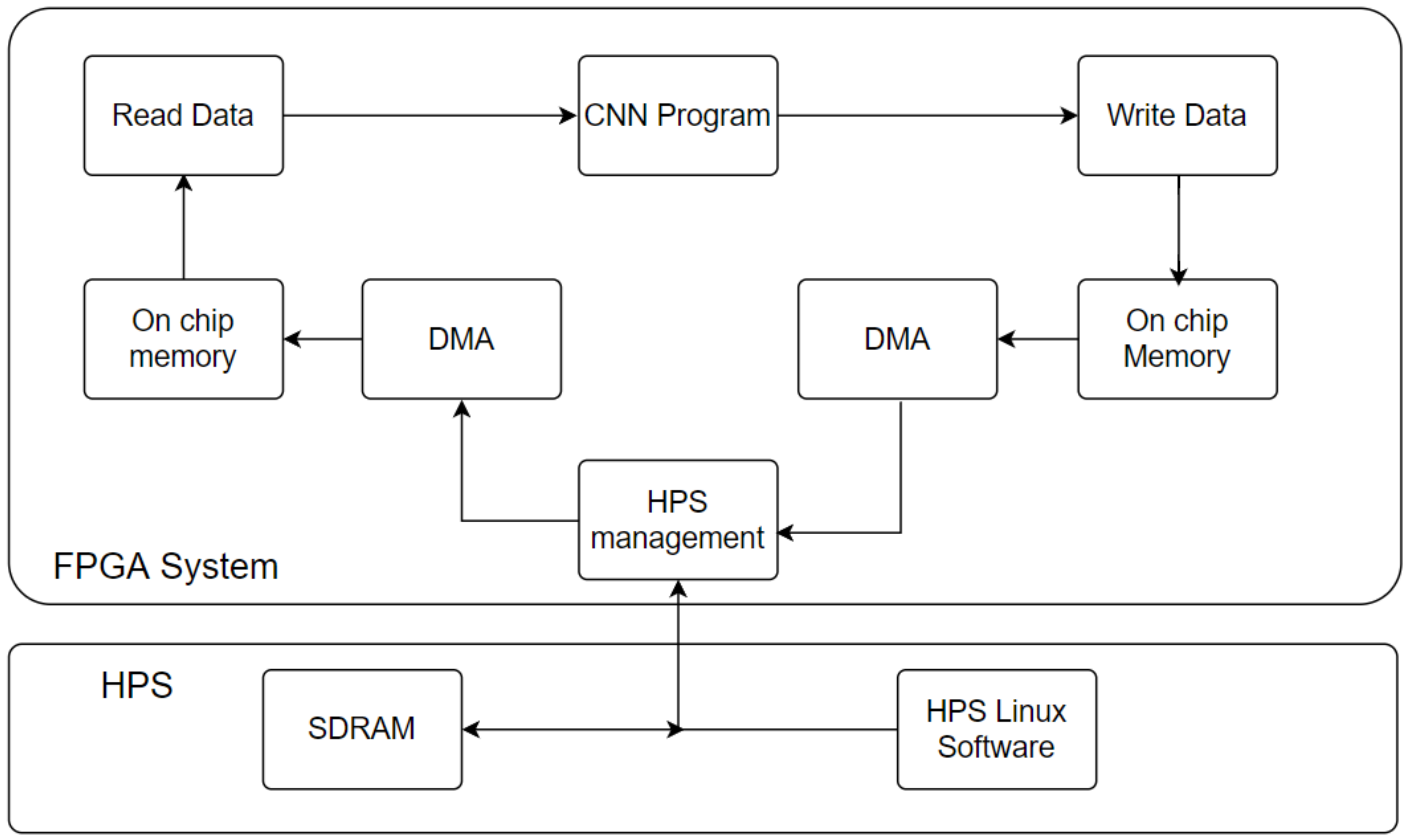


Maxpooling


Ti PEs
N
C
Ti
row=0
row=1
Partial input
valid
row
0
1
Line buffer
>
>
Concatenate
P
o

Hardware software interface approach

HPS-FPGA Communication Pipeline



1.Receive input;

2.Put the input data in memory (SDRAM);

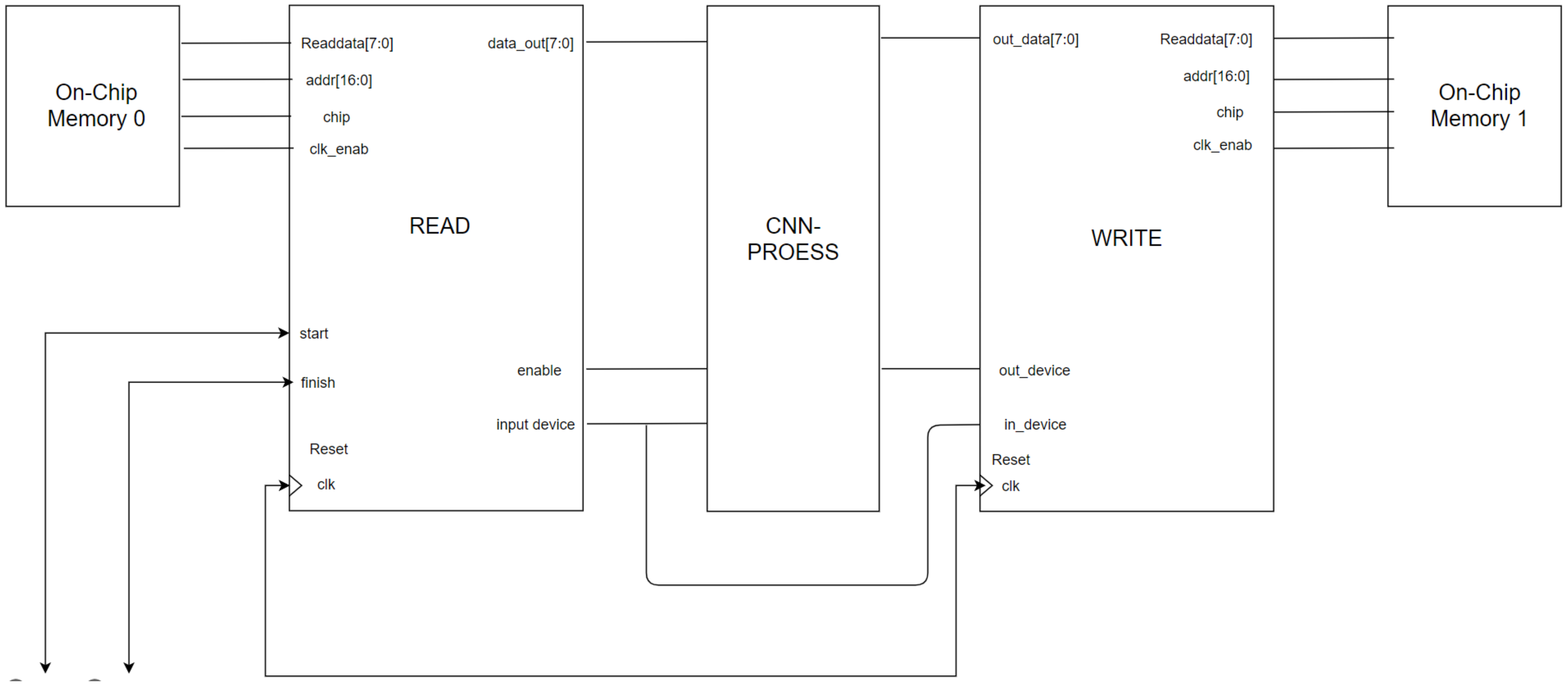
3.Data go through DMA to on-chip memory;

4.Hardware files start to read and process the whole data through YOLO-CNN network;

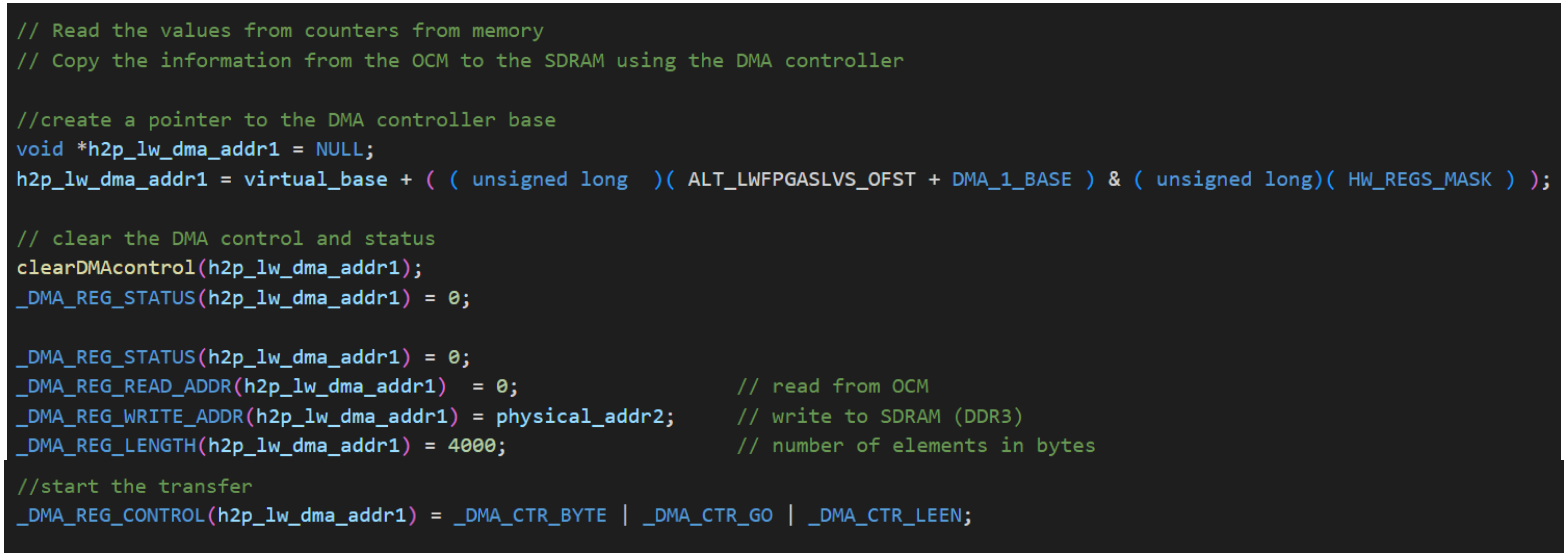
5.Output data being read and written into on-chip memory;

6. DMA transfer the data to SDRAM then read by HPS software and get the final result;

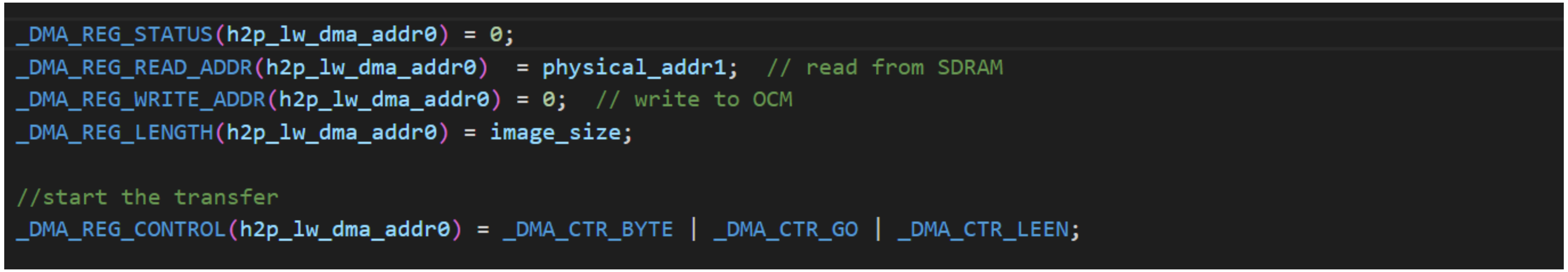
Read & Write Process Logic



DMA Application



```
// Read the values from counters from memory
// Copy the information from the OCM to the SDRAM using the DMA controller

//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_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;
```



```
_DMA_REG_STATUS(h2p_lw_dma_addr0) = 0;
_DMA_REG_READ_ADDR(h2p_lw_dma_addr0)  = physical_addr1;  // read from SDRAM
_DMA_REG_WRITE_ADDR(h2p_lw_dma_addr0) = 0;  // write to OCM
_DMA_REG_LENGTH(h2p_lw_dma_addr0) = image_size;

//start the transfer
_DMA_REG_CONTROL(h2p_lw_dma_addr0) = _DMA_CTR_BYTE | _DMA_CTR_GO | _DMA_CTR_LEEN;
```

Thank you