차세대 반도체 디지털 혁신공유대학 사업단

POLARIS

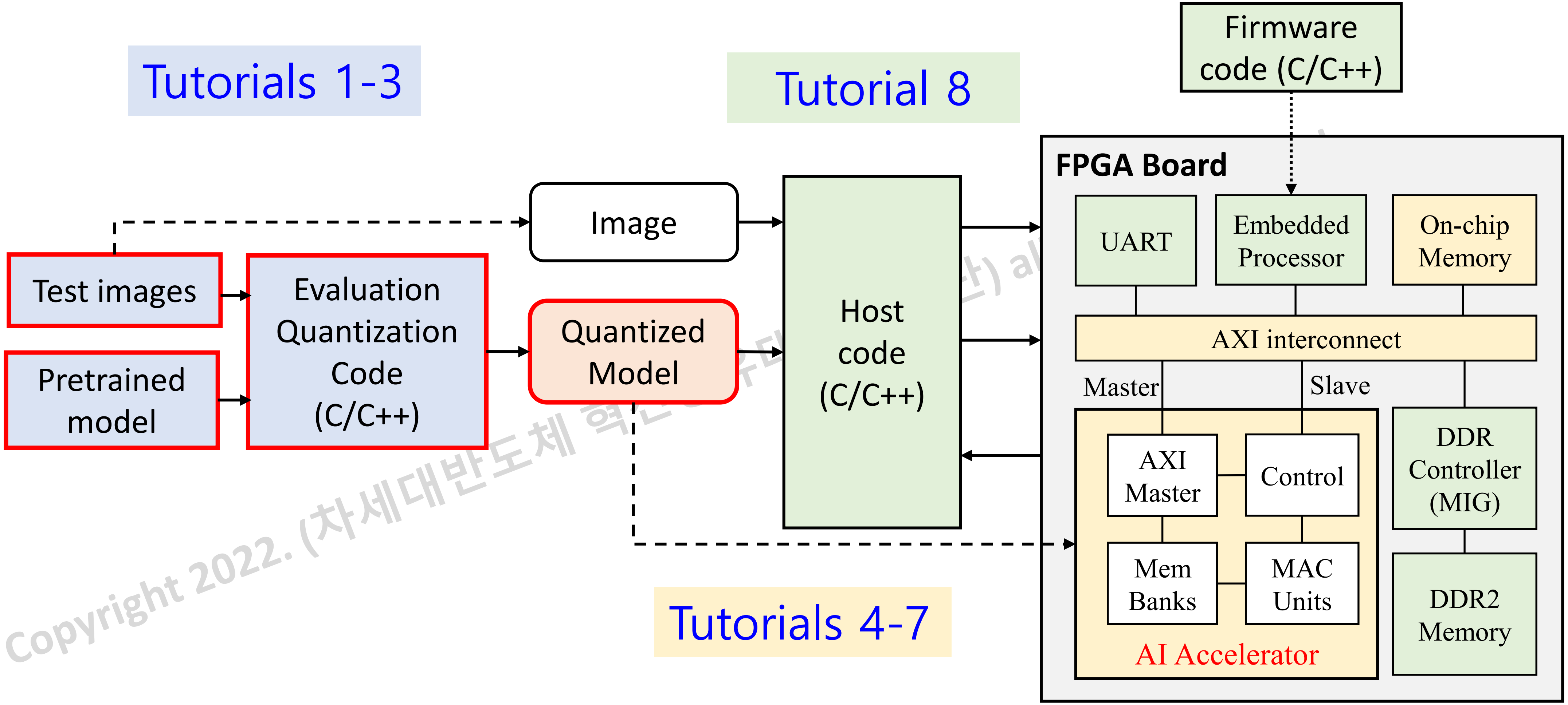
Deep Learning Hardware 설계 경진대회
Model quantization, data preparation

2022.02.14 (Mon)

Road map

- Review
- Accelerator
- Quantization
- Reference S/W

Top structure and tutorials



Test image

- Each test image has **three color channels**: red, green and blue.
  - Image size: height=1080, width=1920, channel=3
    $\Rightarrow$ 6,220,800 (=1080×1920×3) bytes
  - Stored in a compressed format (jpeg or jpg) (402KB)

205	204	204	204	204	204	202	202
208	207	207	207	207	206	205	204
209	208	208	207	207	207	205	204
206	206	206	205	205	204	202	201

236	235	234	234	232	232	231	231
239	238	237	237	235	234	234	233
241	240	238	237	236	236	234	233
238	238	236	235	234	233	231	230

238	237	234	234	233	233	229	229
241	240	237	237	236	235	232	231
240	239	238	237	234	234	232	231
237	237	236	235	232	231	229	226

YOLO input

- How to make an input for a YOLO network?
  - An input image is rescaled in a square RGB image, i.e., width=height=320
  - Maintain the aspect ratios of objects after rescaling
    - Put a rescaled image at the center of a 320x320 image


```
[net]
# Testing
batch=1
subdivisions=1
# Training
#batch=64
#subdivisions=2
width=320
height=320
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 50200
policy=steps
steps=40000,45000
scales=.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
```

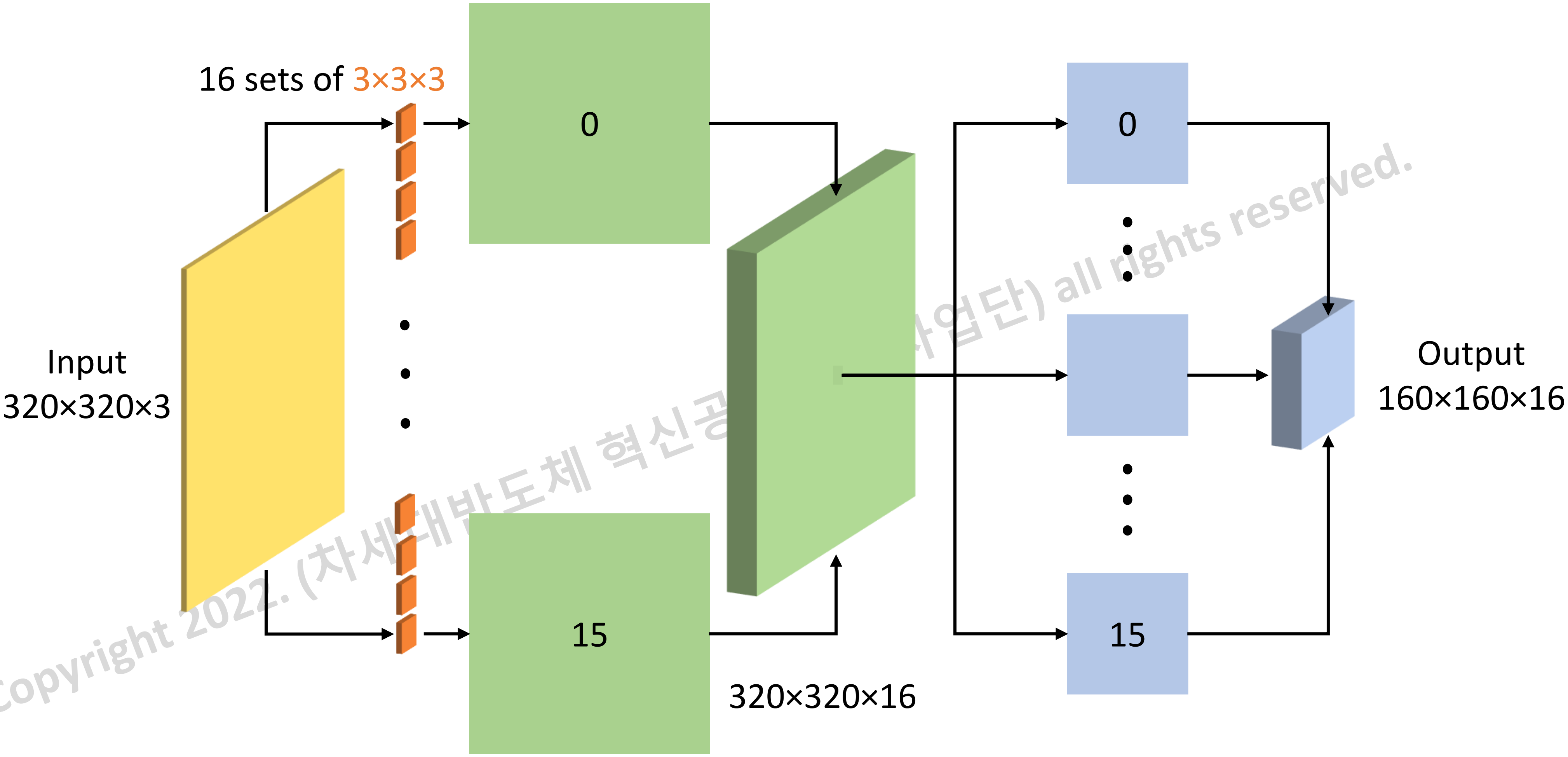
Tiny-YOLO-v3 (AIX)

- A pretrained model is defined by two files
  - yolov3-tiny-aix2022.cfg:
    - Network's configuration
  - yolov3-tiny-aix2022.weights (3354 KB)
    - 32-bit floating point parameters

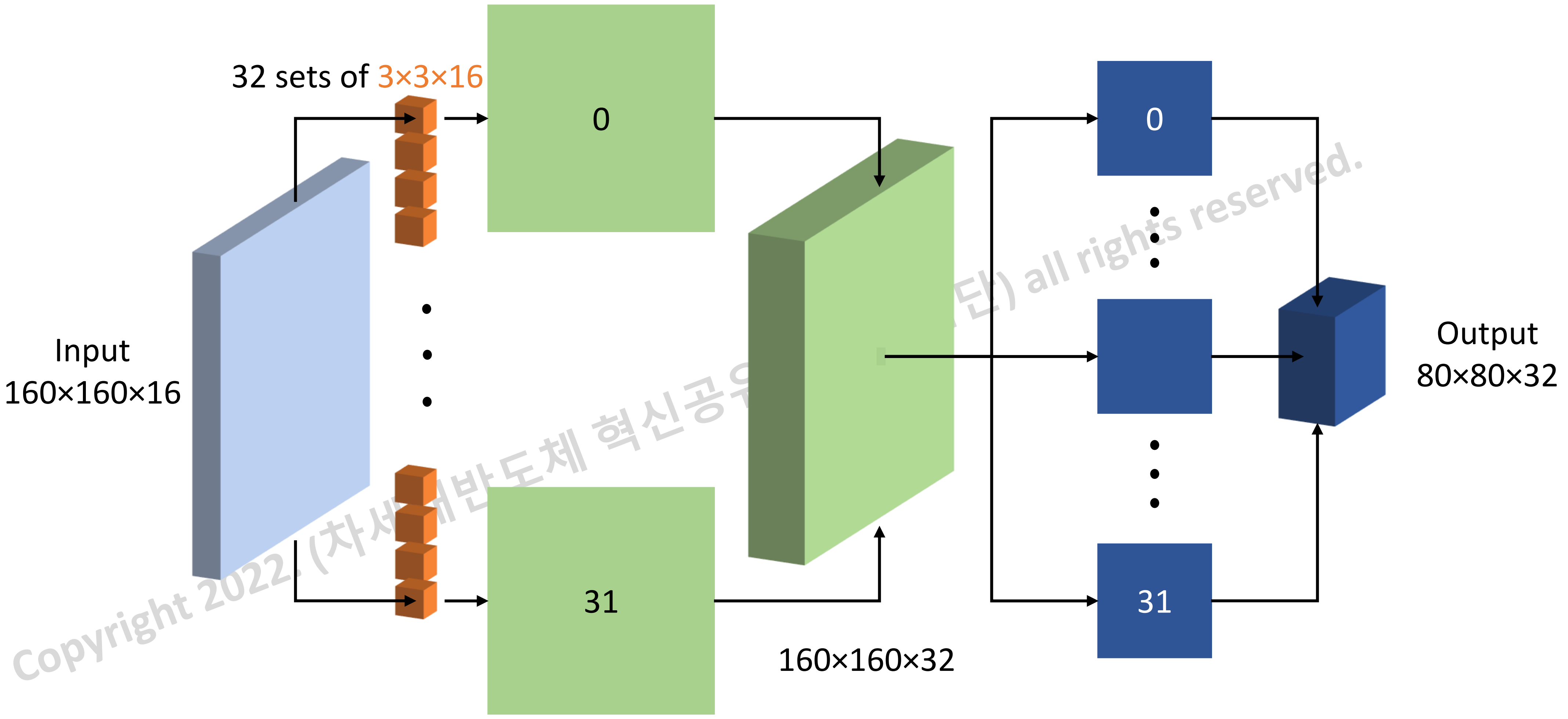
- Tiny-YOLOv3 inference
  - 22 layers
    - 11 convolutional layers
      - 3x3: eight layers, 1x1: three layers
    - 6 max pooling layer (one has stride = 1)
    - Route, upsample and yolo layers
  - Inputs: image (320x320x3) and filters
  - Outputs
    - Layer 13: 10x10x195, Layer 20: 20x20x195

layer	type	filter	input	output
0	conv	3x3x3x16	320x320x3	320x320x16
1	max		320x320x16	160x160x16
2	conv	3x3x16x32	160x160x16	160x160x32
3	max		160x160x32	80x80x32
4	conv	3x3x32x64	80x80x32	80x80x64
5	max		80x80x64	40x40x64
6	conv	3x3x64x128	40x40x64	40x40x128
7	max		40x40x128	20x20x128
8	conv	3x3x128x128	20x20x128	20x20x128
9	max		20x20x128	10x10x128
10	conv	3x3x128x128	10x10x128	10x10x128
11	max		10x10x128	10x10x128
12	conv	3x3x128x128	10x10x128	10x10x128
13	conv	1x1x128x195	10x10x128	10x10x195
14	yolo			
15	route	12		10x10x128
16	conv	1x1x128x128	10x10x128	10x10x128
17	upsample		10x10x128	20x20x128
18	route	17,8		20x20x128
19	conv	3x3x128x128	20x20x256	20x20x128
20	conv	1x1x128x195	20x20x128	20x20x195
21	yolo			

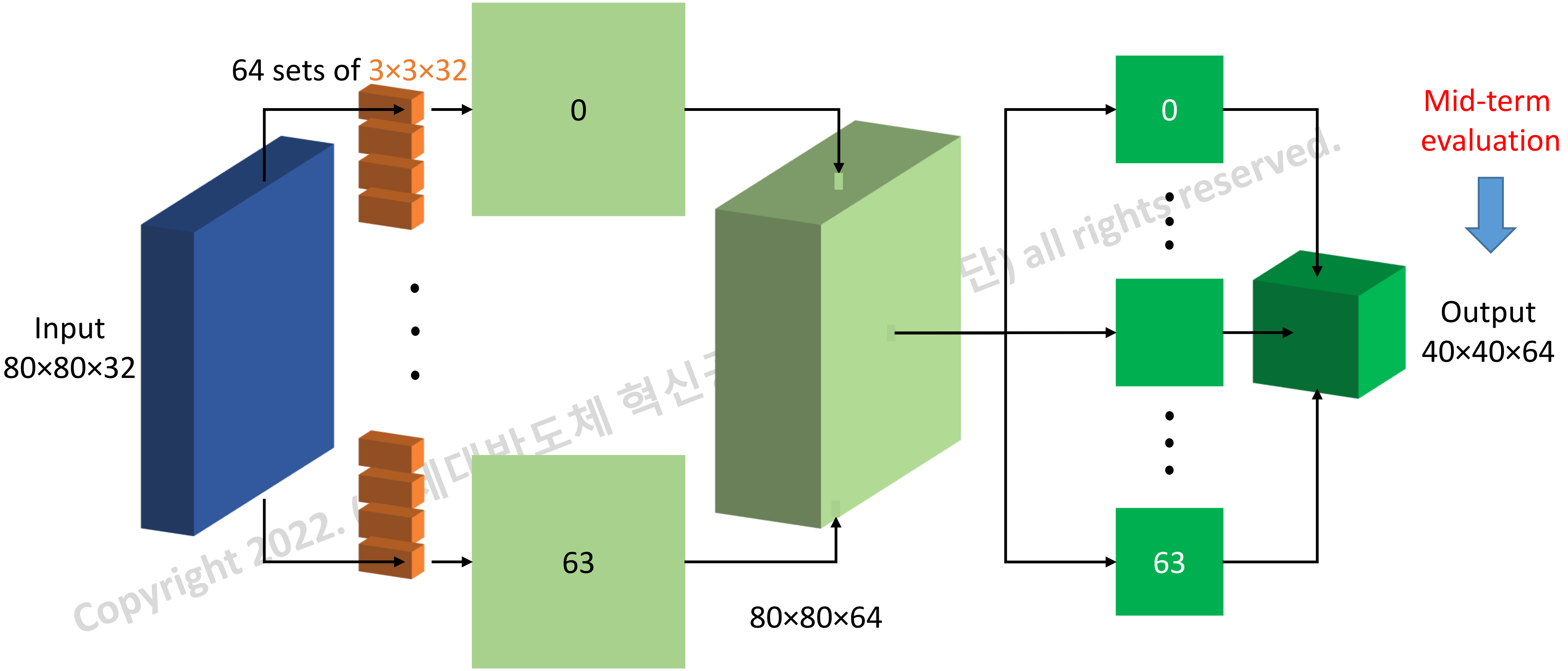
Convolution + max-pooling: Layers 0, 1



Convolution + max-pooling: Layers 2, 3



Convolution + max-pooling: Layers 4, 5

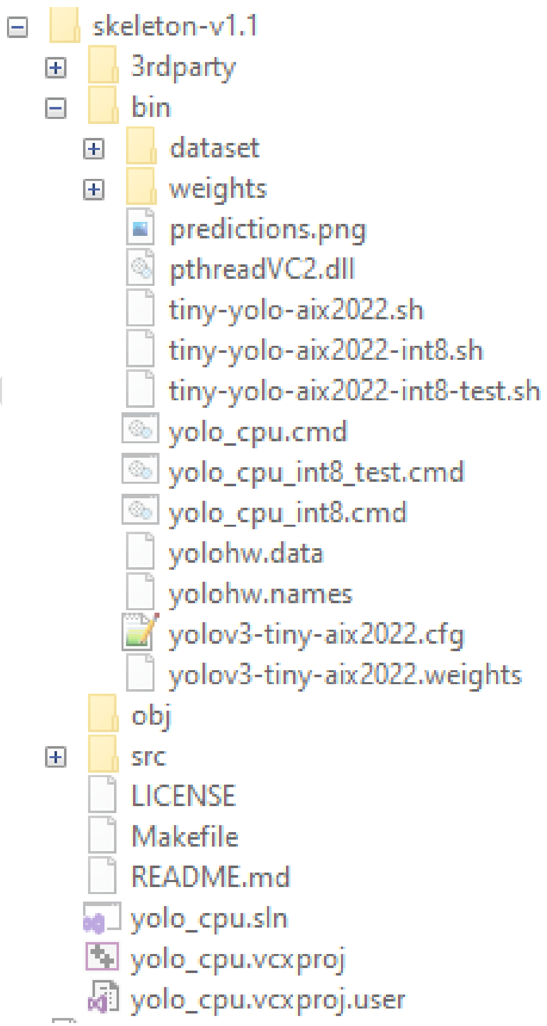


Basic operations

- Convolution
  - 3D convolution
- Batch normalization
  - Batch norm folding
- Activation
  - Leaky rectified Unit
- Max pooling

skeleton v1.1

- We release skeleton v1.1 which supports Windows users.
  - No change
    - src/
    - bin/dataset
    - the model (yolov3-tiny-aix2022.cfg, and yolov3-tiny-aix2022.weights).
- Changes
  - Add 3rdparty/
  - Add Visual studio (VS) project files:
    - yolo_cpu.sln, yolo_cpu.vcxproj, and yolo_cpu.vcxproj.user
  - Add example scripts
    - bin/yolo_cpu.cmd
    - bin/yolo_cpu_int8.cmd
    - bin/yolo_cpu_int7_test.cmd



How to compile and run: UNIX

- Report the accuracy of all 60 items
- The 32-bit model achieves the mean average precision (mAP) is 83.20%.

```
class_id = 30, name = coffee_mate_french_vanilla,        ap = 97.68 %
class_id = 31, name = pepperidge_farm_milk_chocolate_macadamia_cookies,          ap = 71.63
class_id = 32, name = kitkat_king_size,          ap = 85.32 %
class_id = 33, name = snickers,          ap = 36.60 %
class_id = 34, name = toblerone_milk_chocolate,          ap = 97.04 %
class_id = 35, name = clif_z_bar_chocolate_chip,         ap = 98.70 %
class_id = 36, name = nature_valley_crunchy_oats_n_honey,        ap = 72.28 %
class_id = 37, name = ritz_crackers,     ap = 97.73 %
class_id = 38, name = palmolive_orange,          ap = 55.47 %
class_id = 39, name = crystal_hot_sauce,         ap = 100.00 %
class_id = 40, name = tapatio_hot_sauce,         ap = 0.00 %
class_id = 41, name = nabisco_nilla_wafers,      ap = 0.00 %
class_id = 42, name = pepperidge_farm_milano_cookies_double_chocolate,   ap = 0.00 %
class_id = 43, name = campbells_chicken_noodle_soup,     ap = 0.00 %
class_id = 44, name = frappuccino_coffee,        ap = 0.00 %
class_id = 45, name = chewy_dips_chocolate_chip,         ap = 34.73 %
class_id = 46, name = chewy_dips_peanut_butter,          ap = 0.00 %
class_id = 47, name = nature_vally_fruit_and_nut,        ap = 0.00 %
class_id = 48, name = cheerios,          ap = 0.00 %
class_id = 49, name = lindt_excellence_cocoa_dark_chocolate,     ap = 0.00 %
class_id = 50, name = hersheys_symphony,         ap = 0.00 %
class_id = 51, name = campbells_chunky_classic_chicken_noodle,   ap = 0.00 %
class_id = 52, name = martinellis_apple_juice,   ap = 0.00 %
class_id = 53, name = dove_pink,         ap = 0.00 %
class_id = 54, name = dove_white,        ap = 0.00 %
class_id = 55, name = david_sunflower_seeds,     ap = 0.00 %
class_id = 56, name = monster_energy,    ap = 0.00 %
class_id = 57, name = act_ii_butter_lovers_popcorn,      ap = 0.00 %
class_id = 58, name = coca_cola_glass_bottle,    ap = 0.00 %
class_id = 59, name = twix,      ap = 0.00 %
 for thresh = 0.24, precision = 0.83, recall = 0.71, F1-score = 0.76
 for thresh = 0.24, TP = 1362, FP = 282, FN = 562, average IoU = 62.79 %

 mean average precision (mAP) = 0.831957, or 83.20 %
Total Detection Time: 32.000000 Seconds
(base) truongnx@marlin:~/aix2022/skeleton-v1.1/bin$
```

Objectives

- Accelerator
  - Motivation
  - FPGA board and DSP

- Quantization
  - Background
  - Quantization
    - Post-training quantization
    - Training-aware quantization
- Code
  - Flow
  - Evaluation (mAP)

Road map

- Review
- Accelerator
- Quantization
- Reference S/W

Motivating example: Layer 0

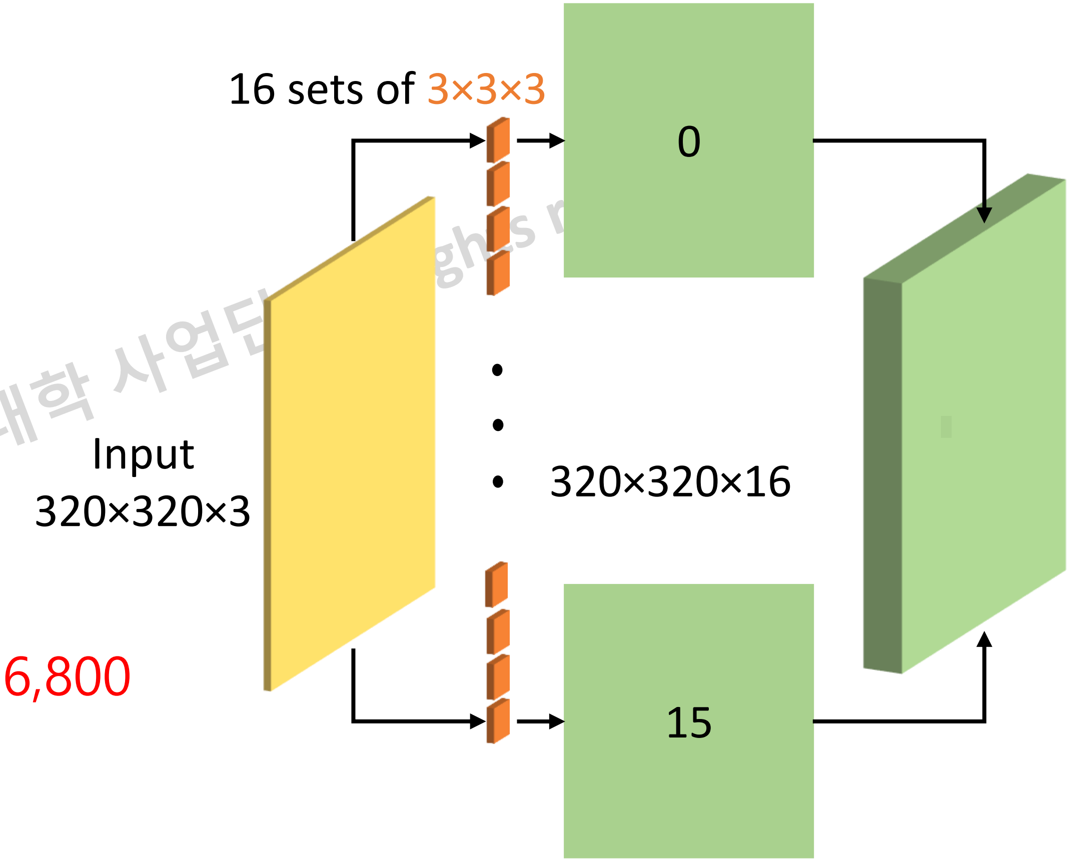
- The number of output pixels is 1,638,400 (=320×320×16)
  - Each output is calculated by

$$y = \sum_{i=0}^{3\times3\times3-1} W_i * x_i$$

Where W and x are weights and inputs

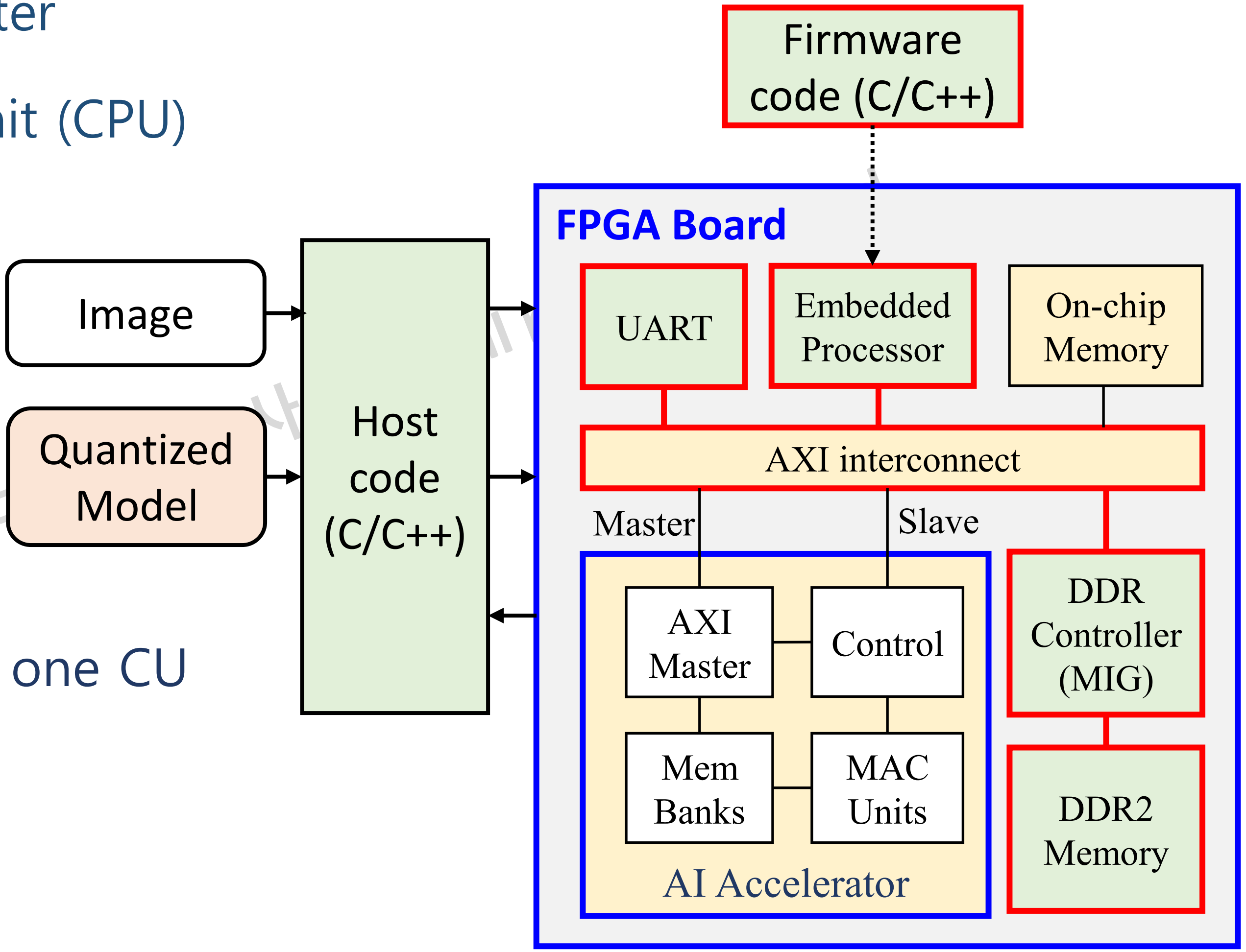
$\Rightarrow$ 27 multiplication operations

- The no. of multiplication operations is 44,236,800 (=320×320×16×27)



Accelerator

- Conventional general purpose computer
  - Processor or central processing unit (CPU)
    - One computing unit (CU)
  - Memory
  - Input/Output (IO)
- Performance issue:
  - Example: **44,236,800** times to use one CU
- How to boost the performance?



How to boost the performance: Compression

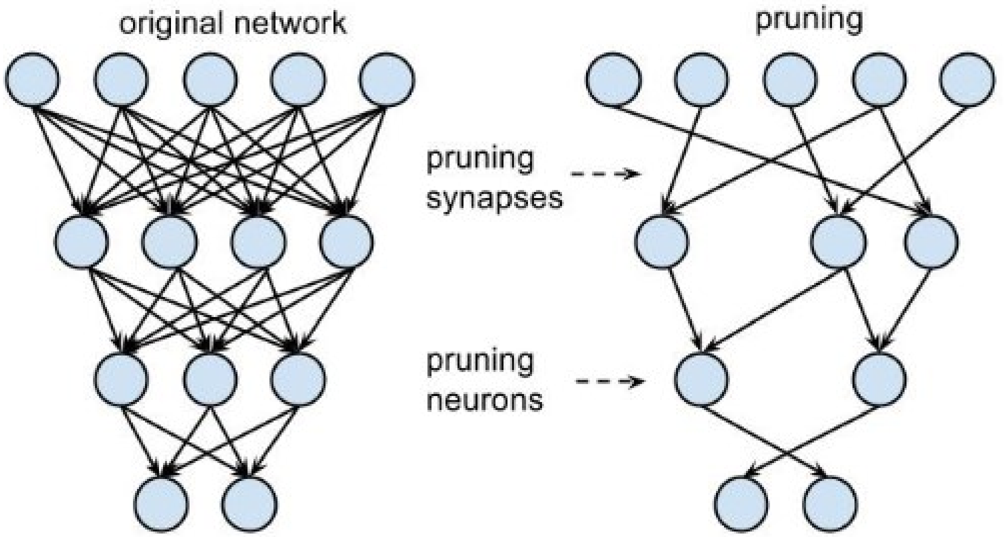
- Motivation
  - Deep Neural Networks are BIG … and getting BIGGER
  - Too big to store on-chip SRAM and DRAM accesses use a lot of energy
  - Not suitable for low-power mobile/embedded systems
- Technique to reduce size of neural networks without losing accuracy
  1) Pruning to Reduce Number of Weights
  2) Quantization to Reduce Bits per Weight
  3) Huffman Encoding

Song Han et al., "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding", ICLR 2016

Pruning

- Remove weights/synapses “close to zero”
- **Retrain** to maintain accuracy
- Repeat

➔ Require retraining (must know dataset)



$$y = \sum_{i=0}^{3\times3\times3-1} W_i * x_i$$

$i = 0$
$y = 0$
while $i < 27$
  if $w_i \neq 0$ // Skipping computation
    $m = w_i * x_i$ // Do multiplication
    $y = y + m$ // Accumulate the results
  end if
  $i = i + 1$ // Update loop index
end for

Not great on convolutional layers (our case)

Parallel Computing

- Compute a sum of *N* products

$$y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

$y_1^{(0)} = w_0 * x_0, \ldots, y_{15}^{(0)} = w_{15} * x_{15}$ // N multipliers

$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \ldots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$ // N/2 adders

$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \ldots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$ // N/4 adders

$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \ldots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$ // N/4 adders

$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$ // N/4 adders

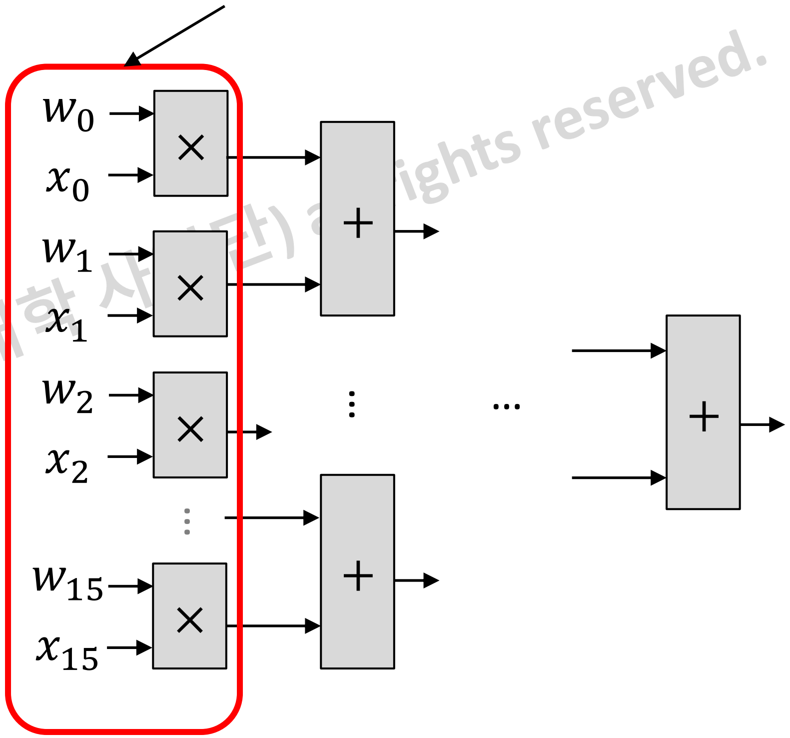
$y = y_1^{(4)}$ // Output

Parallel Computing: Multipliers

- Compute a sum of *N* products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

N block/module of multipliers



- Pseudo code

$y_1^{(0)} = w_0 * x_0, \ldots, y_{15}^{(0)} = w_{15} * x_{15}$

$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \ldots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$

$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \ldots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$

$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \ldots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$

$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$

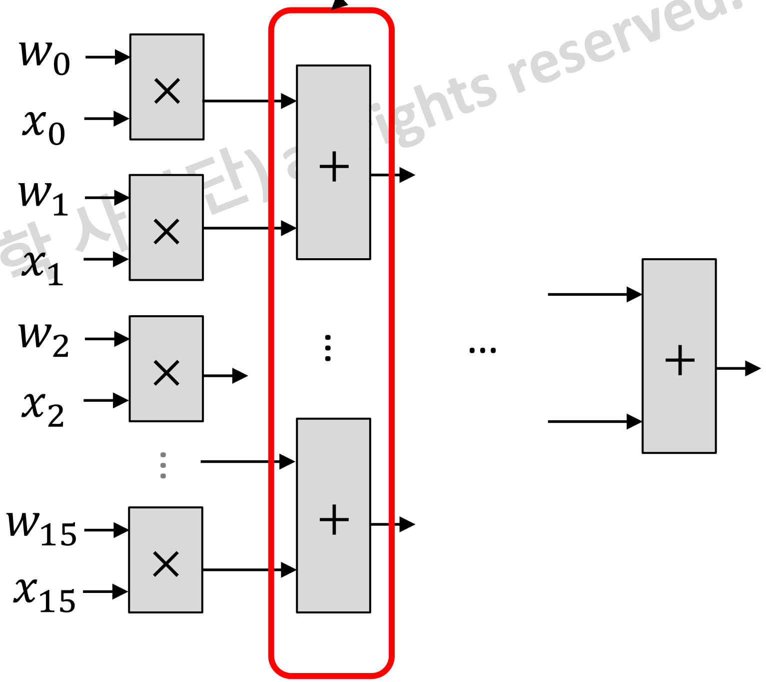
$Y = y_1^{(4)}$

Parallel Computing: Adder tree (level=1)

- Compute a sum of *N* products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

N/2 block/module of adders



- Pseudo code

$y_1^{(0)} = w_0 * x_0, \ldots, y_{15}^{(0)} = w_{15} * x_{15}$

$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \ldots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$

$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \ldots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$

$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \ldots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$

$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$

$Y = y_1^{(4)}$

Parallel Computing: Adder tree (level=2)

- Compute a sum of *N* products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

$y_1^{(0)} = w_0 * x_0, \dots, y_{15}^{(0)} = w_{15} * x_{15}$

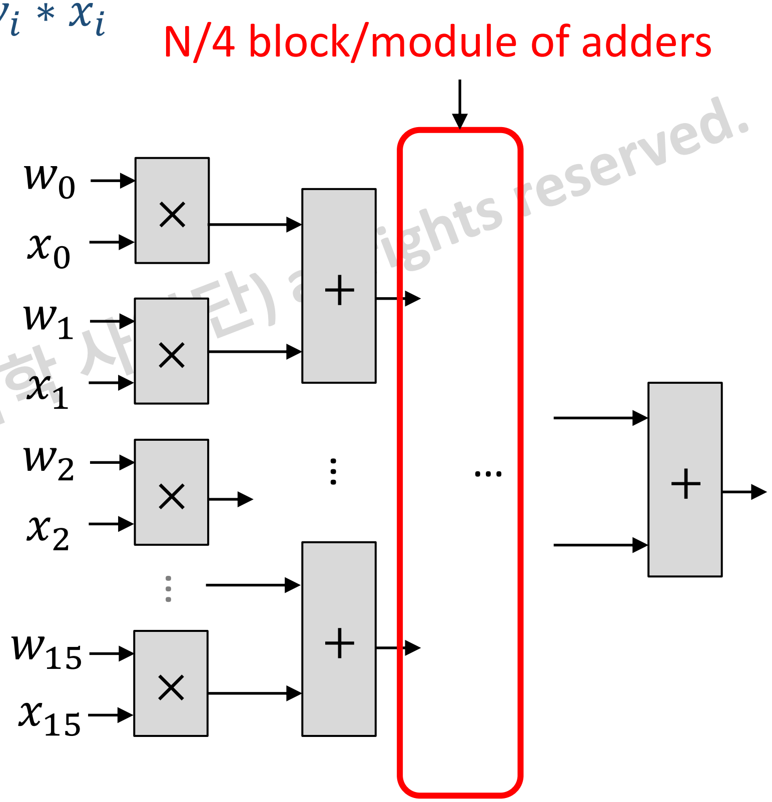
$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \dots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$

$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \dots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$

$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \dots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$

$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$

$Y = y_1^{(4)}$



Parallel Computing: Adder tree (level=4)

- Compute a sum of *N* products

$$Y = \sum_{i=0}^{15} w_i * x_i$$

- Pseudo code

$y_1^{(0)} = w_0 * x_0, \ldots, y_{15}^{(0)} = w_{15} * x_{15}$

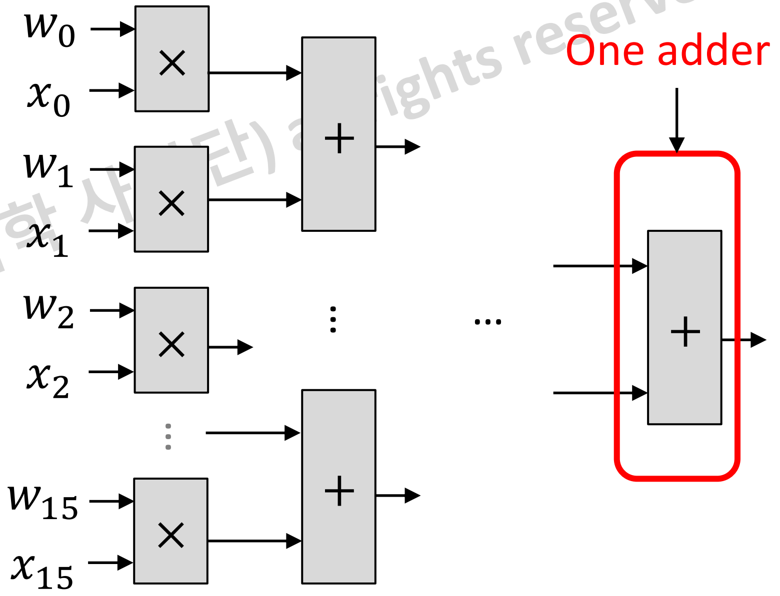
$y_1^{(1)} = y_0^{(0)} + y_1^{(0)}, \ldots, y_7^{(1)} = y_{14}^{(0)} + y_{15}^{(0)}$

$y_1^{(2)} = y_0^{(1)} + y_1^{(1)}, \ldots, y_3^{(2)} = y_6^{(1)} + y_7^{(1)}$

$y_1^{(3)} = y_0^{(2)} + y_1^{(2)}, \ldots, y_1^{(3)} = y_2^{(2)} + y_3^{(2)}$

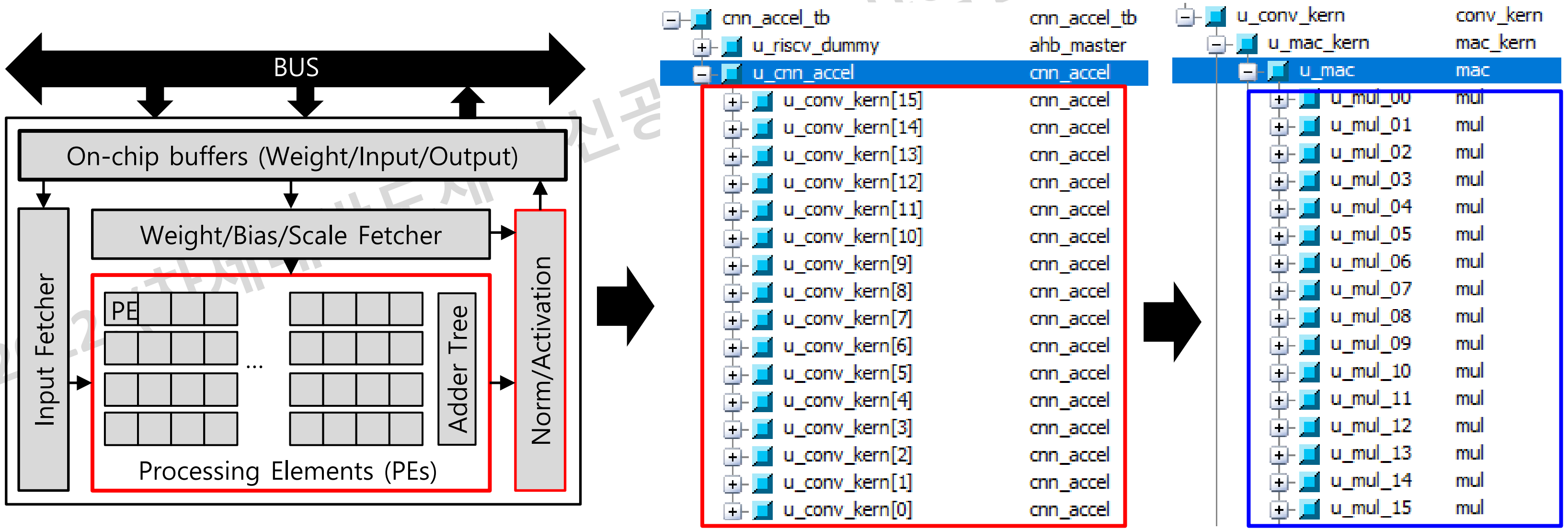
$y_1^{(4)} = y_0^{(3)} + y_1^{(3)}$

$Y = y_1^{(4)}$



Accelerator

- Processing Element (PE) Array
  - Perform convolution/activation/quantization operations.
  - To: The number of convolutional kernels (output feature maps)
  - Ti: The number of multipliers in a kernel
- The number of multipliers is : To× Ti
  - **256 (=16×16) multipliers run in parallel**

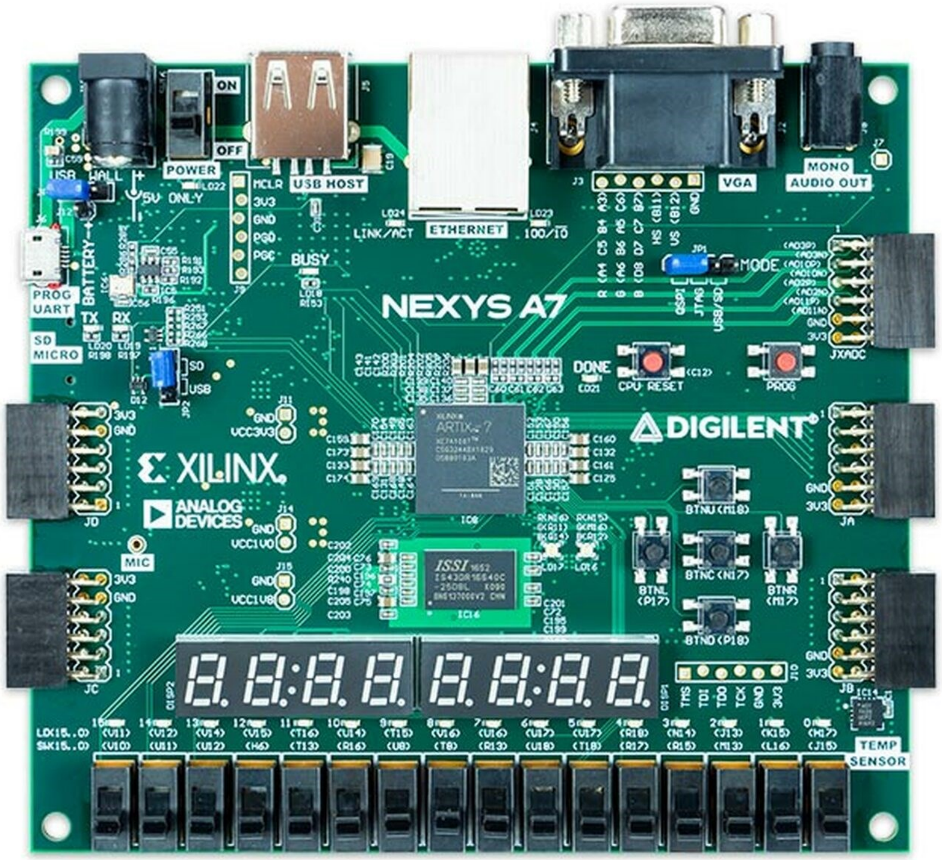


To=16

Ti=16

Nexys A7 FPGA board

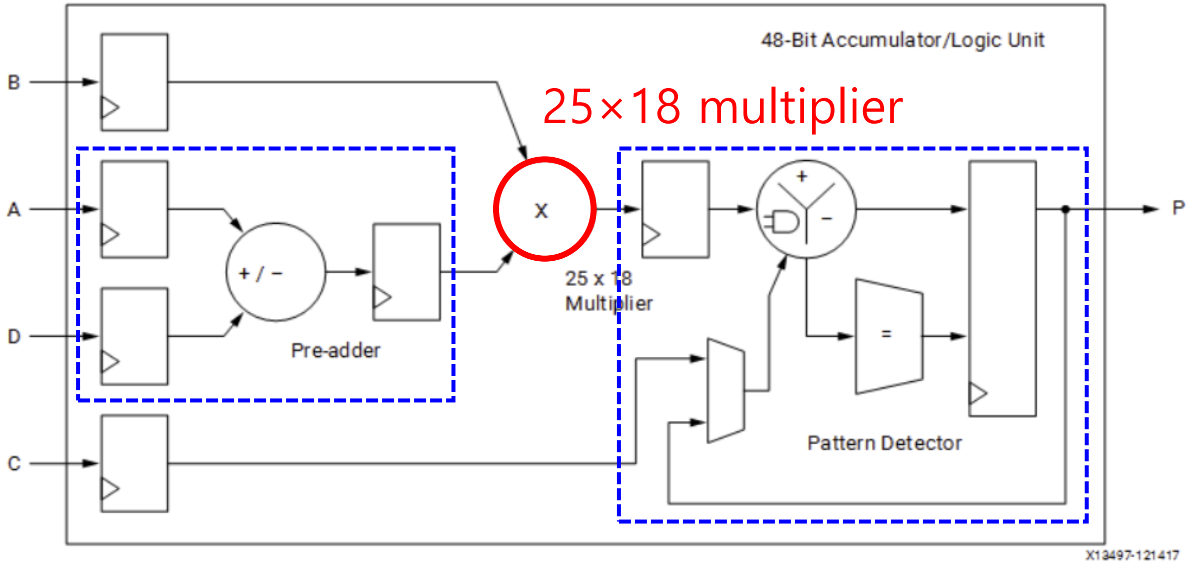
- Xilinx Artix-7 FPGA XC7A100T-1CSG324C
- 15,850 logic slices
  - Each with four 6-input LUTs and 8 FFs
- 4,860 Kbits of fast block RAM
- 240 DSP slices
  - Dedicated to multiplication and accumulation (MAC)
- Internal clock speeds exceeding 450 MHz
- 128 MB DDR2 Memory
- USB-JTAG port for FPGA programming and communication



DSP48

- The DSP48 block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA
- The computational chain in the DSP48 contains an add/subtract unit connected to a multiplier connected to a final add/subtract/accumulate engine.
  - Implement complicated functions of the form, e.g. P=Bx(A+D)+C

⇒ The parallel factor (e.g. To×Ti) is constrained by the number of DSPs



https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/uwa1504034294196.html

Road map

Review

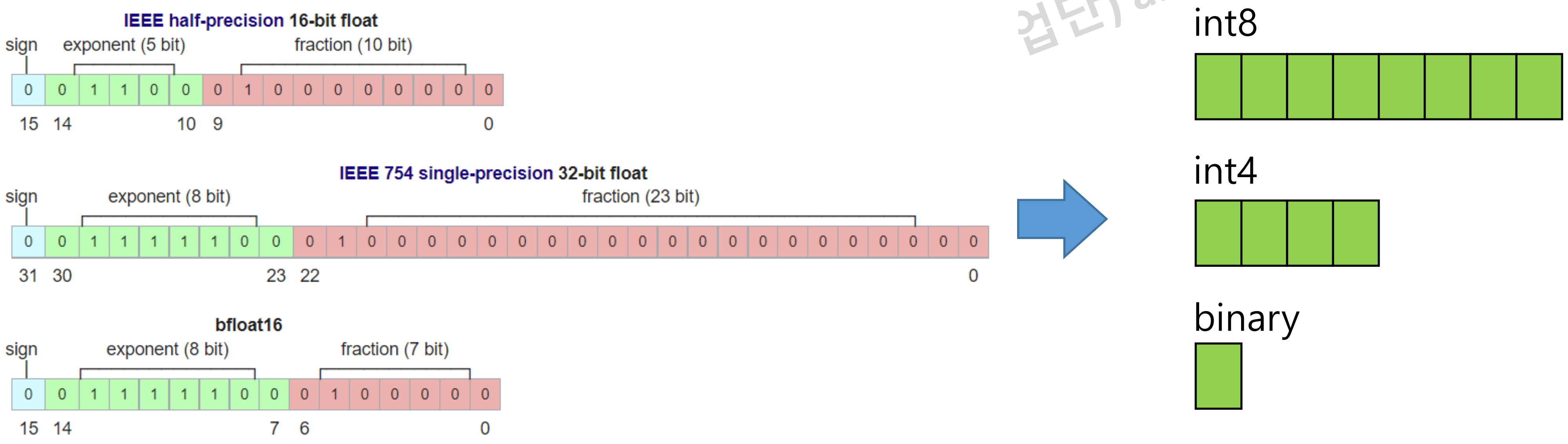
Accelerator

Quantization

Reference S/W

Quantization

- Quantization refers to mapping values from fp32 a lower precision format
  - Specified by
    - Format
    - Mapping type
    - Granularity



Quantization

- A quantization scheme be an affine mapping of integers q to real numbers r, i.e. of the form $r = S(q - Z)$, For some constants S and Z
- For 8-bit quantization, q is quantized as an 8-bit integer
  - For B-bit quantization, q is quantized as an B-bit integer.
  - Some arrays, typically bias vectors, are quantized as 16/32-bit integers
- The constant S (for "scale") is an arbitrary positive real number.
- The constant Z (for "zero-point") is of the same type as quantized values q, and is in fact the quantized value q corresponding to the real value 0.


```
template<typename QType>  // e.g. QType=uint8
struct QuantizedBuffer {
  vector<QType> q;        // the quantized values
  float S;                // the scale
  QType Z;                // the zero-point
};
```

B. Jacob, et. al, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", CVPR 2018.

Example

- Mapping values from fp32 to a 8-bit integer format
  - All quantized values are in {-128, -127, …, 127}

fp32

-0.3302	-0.3016	-0.1170
0.3708	-0.2676	-0.1463
-0.0760	-0.0007	0.3297

int8 and Quantization error

$S = 1/128$

-43	-39	-15
47	-35	-19
-10	-1	42

0.0057	0.0031	0.0002
0.0036	0.0058	0.0021
0.0021	0.0071	0.0016

$S = 1/256$

-85	-78	-30
94	-69	-38
-20	-1	84

0.0018	0.0031	0.0002
0.0036	0.0019	0.0021
0.0021	0.0032	0.0016

$S = 1/342$

-113	-104	-41
126	-92	-51
-26	-1	112

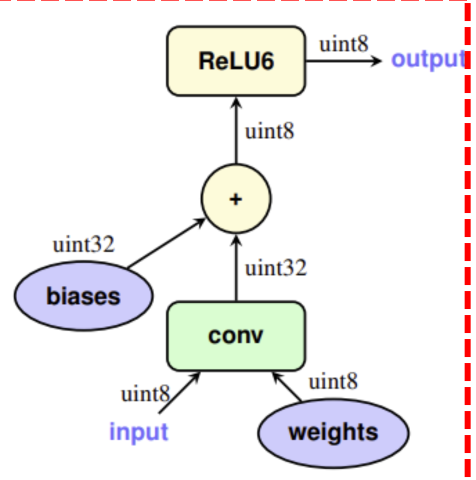
0.0002	0.0025	0.0029
0.0024	0.0014	0.0028
0.0000	0.0022	0.0022

int8

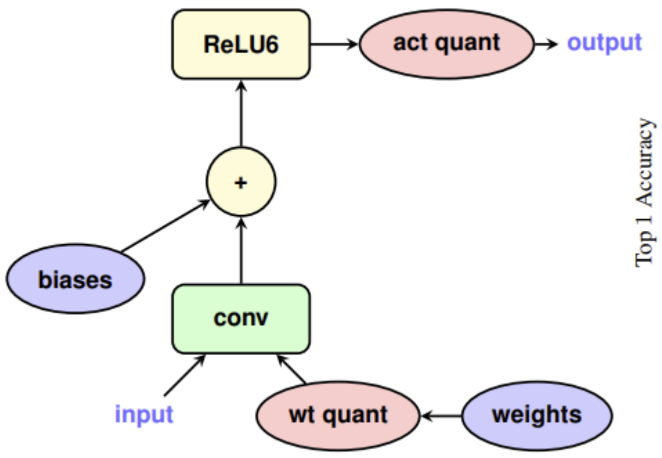
Quantization error

Quantization

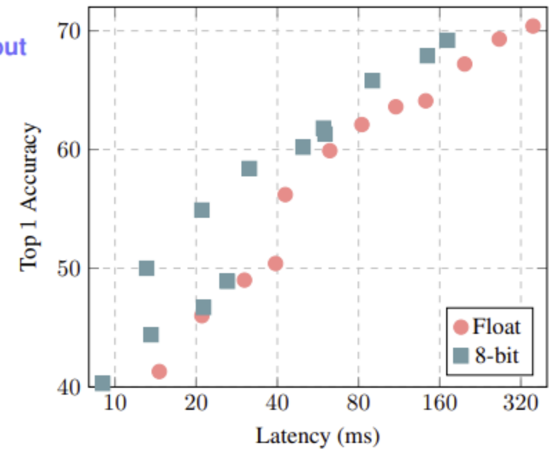
- Integer-arithmetic-only inference of a convolution layer
  - The input and output are represented as 8-bit integers
  - The convolution involves 8-bit integer operands and a 32-bit integer accumulator.
  - The bias addition involves only 32-bit integers
  - The ReLU6 nonlinearity only involves 8-bit integer arithmetic



(a) Integer-arithmetic-only inference



(b) Training with simulated quantization



(c) ImageNet latency-vs-accuracy tradeoff

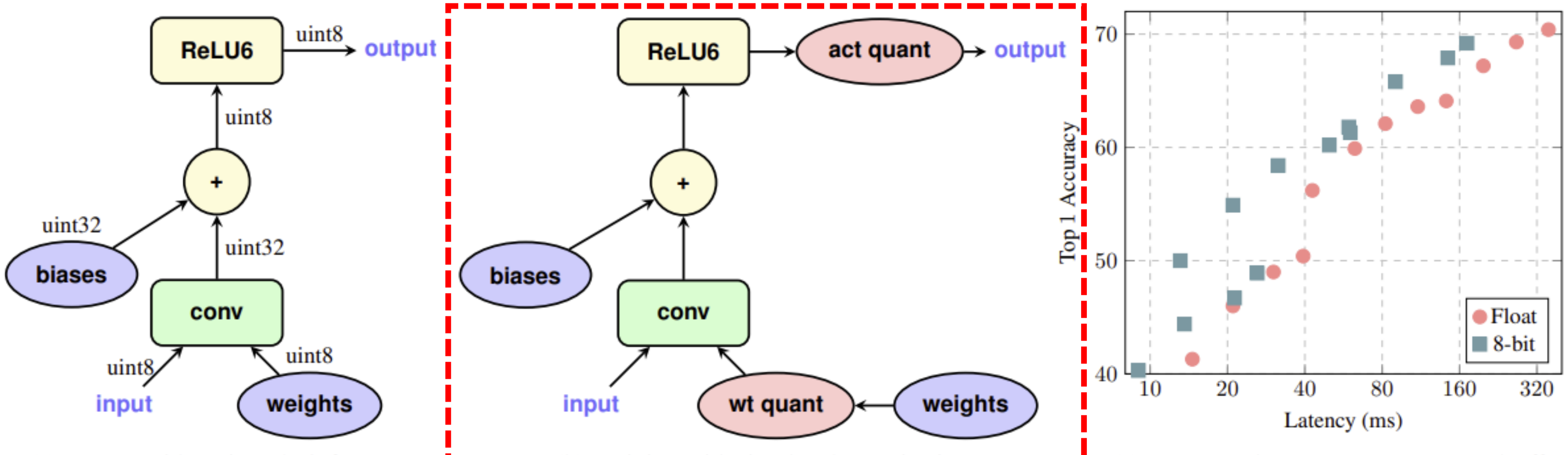
B. Jacob, et. al, "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", CVPR 2018.

Post-training quantization

- Post-training quantization refers to quantizing both weights and activations to reduced precision, e.g., int8
- Requires estimation of statics of activations for determining quantizer parameters
- Quantizer parameters are determined by minimizing error metric:
  - KL divergence
  - Saturation error

Quantization-aware training

- Training with simulated quantization of the convolution layer.
- All variables and computations are carried out using 32-bit floating-point arithmetic.
- Weight quantization ("wt quant") and activation quantization ("act quant") nodes are injected into the computation graph to simulate the effects of quantization of the variables.
- The resultant graph approximates the integer-arithmetic-only computation graph in panel, while being trainable using conventional optimization algorithms for floating point models.



(a) Integer-arithmetic-only inference (b) Training with simulated quantization (c) ImageNet latency-vs-accuracy tradeoff

Road map

Review

Accelerator

Quantization

Reference S/W

Source files

- additionally.c *// Definitions of darknet functions used*
- additionally.h *// Declaration of darknet functions + additional functions for forward pass of yolo model*
- box.c
- box.h

  // For bounding boxes
- stb_image_write.h
- stb_image.h

  // For loading/writing images
- yolov2_forward_network.c *// Functions for forward pass of yolo network*
- **yolov2_forward_network_quantized.c** *// Functions for quantization, saving of the quantized model, and the forward pass of quantized yolo model*
- main.c *// The main functions*

You should mainly edit this file for quantization!

main.c

- void test_detector_cpu(

```
    char **names,          // List of all items (bin/yolohw.names)
    char *cfgfile,         // Configuration file (bin/yolov3-tiny-aix2022.cfg)
    char *weightfile,      // Configuration file (bin/yolov3-tiny-aix2022.weights)
    char *filename,        // Input image file
    float thresh,          // Hierarchical threshold
    int quantized,         // On/off quantization
    int save_params,       // On/off save output
    int dont_show          // Don't show
)
```

test_detector_cpu

- Parse the configuration file
  - A network architecture is stored in the variable “net”
  - Example: Layer index 0
    - Convolutional, 16 filters, filter size 3x3, padding = 1
    - Use Leaky function

```
// Detect on Image: this function uses other functions not from this file
void test_detector_cpu(char **names, char *cfgfile, char *weightfile, char *filename, float thresh, int quantized, int dont_show)
{
    //image **alphabet = load_alphabet();            // image.c
    image **alphabet = NULL;
    network net = parse_network_cfg(cfgfile, 1, quantized);    // parser.c
    if (weightfile) {
        load_weights_upto_cpu(&net, weightfile, net.n);    // parser.c
    }
    //set_batch_network(&net, 1);                    // network.c
    srand(2222222);
    yolov2_fuse_conv_batchnorm(net);
    calculate_binary_weights(net);
    if (quantized) {
        printf("\n\n Quantization! \n\n");
        quantization_and_get_multipliers(net);
    }
```

yolov3-tiny-aix2022

Layer 0

```
[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2
```

yolov3-tiny-aix2022.cfg

test_detector_cpu

- Inputs
  - Network
  - Load an input image (load_image)
    - X: image data
- Call an inference function
  - Floating point: network_predict_cpu(net, X)
  - A reference code for quantization
    - network_predict_quantized(net, X)

```
    float nms = .4;
    while (1) {
        if (filename) {
            strncpy(input, filename, 256);
        }
        else {
            printf("Enter Image Path: ");
            fflush(stdout);
            input = fgets(input, 256, stdin);
            if (!input) return;
            strtok(input, "\n");
        }
        image im = load_image(input, 0, 0, 3);              // image.c
        image sized = resize_image(im, net.w, net.h);     // image.c
        layer l = net.layers[net.n - 1];

        box *boxes = calloc(l.w*l.h*l.n, sizeof(box));
        float **probs = calloc(l.w*l.h*l.n, sizeof(float *));
        for (j = 0; j < l.w*l.h*l.n; ++j) probs[j] = calloc(l.classes, sizeof(float *));

        float *X = sized.data;
        time = clock();
        //network_predict(net, X);
#ifdef GPU
        if (quantized) {
            network_predict_gpu_cudnn_quantized(net, X);    // quantized
                                                            //nms = 0.2;
        }
        else {
            network_predict_gpu_cudnn(net, X);
        }
#else
#ifdef OPENCL
        network_predict_opencl(net, X);
#else
        if (quantized) {
            network_predict_quantized(net, X);    // quantized
            nms = 0.2;
        }
        else {
            network_predict_cpu(net, X);
        }
#endif
#endif
```

Convolutional layer in Quantized Model

Model quantization

Executed only once

- Fuse Batch Normalization into Convolutional Weights and Biases
- Quantize Weight & Bias, and Compute Input Multiplier for Each Convolutional Layer

cf.)
void yolov2_fuse_conv_batchnorm(network net)
[additionally.c]

cf.)
void do_quantization(network net)
[yolov2_forward_network_quantized.c]

Inference

- Load Quantized Weight and Bias, and Multiply the Input by the Input Multiplier
- Convolution & Add Bias
- Apply Activation Function
- Normalize the layer's output

cf.)
void forward_convolutional_layer_q(layer l, network_state state)
[yolov2_forward_network_quantized.c]

void do_quantization(network net)

```
// Input Scaling
if (counter >= net.input_calibration_size) {
    printf(" Warning: CONV%d has no corresponding input_calibration parameter - default value 16 will be
        used;\n", j);
}
l->input_quant_multiplier = (counter < net.input_calibration_size) ? net.input_calibration[counter] : 16;
// Using 16 as input_calibration as default value
// l->input_quant_multiplier = floor(l->input_quant_multiplier*pow(2,12))/pow(2,12);
++counter;

// Weight Quantization
l->weights_quant_multiplier = 32; // Arbitrarily set to 32; you should devise your own method to calculate
the weight multiplier
for (fil = 0; fil < l->n; ++fil) {
    for (i = 0; i < filter_size; ++i) {
        float w = l->weights[fil*filter_size + i] * l->weights_quant_multiplier; // Scale
        l->weights_int8[fil*filter_size + i] = max_abs(w, MAX_VAL_8); // Clip
    }
}

// Bias Quantization
float biases_multiplier = (l->weights_quant_multiplier * l->input_quant_multiplier);
for (fil = 0; fil < l->n; ++fil) {
    float b = l->biases[fil] * biases_multiplier; // Scale
    l->biases_quant[fil] = max_abs(b, MAX_VAL_16); // Clip
}
```

input_quant_multiplier = scale factor to be multiplied to the floating-point layer input before casting it into INT8

8-bit fixed-point quantization for weights

16-bit fixed-point quantization for biases

**The provided code is a naïve version of quantization. You should devise your own method to quantize the model in a reasonable fashion!*

void do_quantization(network net)

```c
// Input Scaling
if (counter >= net.input_calibration_size) {
    printf(" Warning: CONV%d has no corresponding input_calibration parameter - default value 16 will be
        used;\n", j);
}
l->input_quant_multiplier = (counter < net.input_calibration_size) ? net.input_calibration[counter] : 16;
// Using 16 as input_calibration as default value
// l->input_quant_multiplier = floor(l->input_quant_multiplier*pow(2,12))/pow(2,12);
++counter;
```

Parsed from bin/yolov3_tiny_aix2022.cfg

```
input_calibration = 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8

[net]
# Testing
#batch=1
#subdivisions=1
# Training
batch=64
subdivisions=2
width=416
height=416
channels=3
```

**If input_calibration parameters are not specified in the cfg file, default value(16) will be used.*

void do_quantization(network net)

- Two steps to quantize weights and biases
  - Multiply by a scale factor (e.g. multiplier)
  - Clipping: avoid overflow
    - Example: int8
      - If x > 127, x=127
      - If x < -128, x = -128

weights

```
// Weight Quantization
l->weights_quant_multiplier = 32; // Arbitrarily set to 32; you should devise your own method to calculate the weight multiplier
for (fil = 0; fil < l->n; ++fil) {
    for (i = 0; i < filter_size; ++i) {
        float w = l->weights[fil*filter_size + i] * l->weights_quant_multiplier; // Scale
        l->weights_int8[fil*filter_size + i] = max_abs(w, MAX_VAL_8); // Clip
    }
}
```

bias

```
// Bias Quantization
float biases_multiplier = (l->weights_quant_multiplier * l->input_quant_multiplier);
for (fil = 0; fil < l->n; ++fil) {
    float b = l->biases[fil] * biases_multiplier; // Scale
    l->biases_quant[fil] = max_abs(b, MAX_VAL_16); // Clip
}
```

forward_convolutional_layer_q

- Convolutional layer (forward_convolutional_layer_q)
  - Convert the input into int8

```
void yolov2_forward_network_q(network net, network_state state)
{
    state.workspace = net.workspace;
    int i;
    for (i = 0; i < net.n; ++i) {
        state.index = i;
        layer l = net.layers[i];

        if (l.type == CONVOLUTIONAL) {
            forward_convolutional_layer_q(l, state);
        }
        else if (l.type == MAXPOOL) {
            forward_maxpool_layer_cpu(l, state);
        }
        else if (l.type == ROUTE) {
            forward_route_layer_cpu(l, state);
        }
```

```
void forward_convolutional_layer_q(layer l, network_state state)
{

    int out_h = (l.h + 2 * l.pad - l.size) / l.stride + 1;    //
    int out_w = (l.w + 2 * l.pad - l.size) / l.stride + 1;    //
    int i, j;
    int const out_size = out_h*out_w;

    typedef int16_t conv_t;    // l.output
    conv_t *output_q = calloc(l.outputs, sizeof(conv_t));

    state.input_int8 = (int8_t *)calloc(l.inputs, sizeof(int));
    int z;
    for (z = 0; z < l.inputs; ++z) {
        int16_t src = state.input[z] * l.input_quant_multiplier;
        state.input_int8[z] = max_abs(src, MAX_VAL_8);
    }
```

forward_convolutional_layer_q

- Processing steps
  - im2col_cpu_int8
  - Do convolution
  - Add a bias
  - Do activation
  - Normalization

```c
    // Use GEMM (as part of BLAS)
    im2col_cpu_int8(state.input_int8, l.c, l.h, l.w, l.size, l.stride, l.pad, b);
    int t;    // multi-thread gemm
    #pragma omp parallel for
    for (t = 0; t < m; ++t) {
        gemm_nn_int8_int16(1, n, k, 1, a + t*k, k, b, n, c + t*n, n);
    }
    free(state.input_int8);

    // Bias addition
    int fil;
    for (fil = 0; fil < l.n; ++fil) {
        for (j = 0; j < out_size; ++j) {
            output_q[fil*out_size + j] = output_q[fil*out_size + j] + l.biases_quant[fil];
        }
    }

    // Activation
    if (l.activation == LEAKY) {
        for (i = 0; i < l.n*out_size; ++i) {
            output_q[i] = (output_q[i] > 0) ? output_q[i] : output_q[i] / 10;
        }
    }

    // De-scaling
    float ALPHA1 = 1 / (l.input_quant_multiplier * l.weights_quant_multiplier);
    for (i = 0; i < l.outputs; ++i) {
        l.output[i] = output_q[i] * ALPHA1;
    }
```

The accelerator should perform these operations

How to Compile & Run?

e.g.) tiny-yolo-aix2022-int8.sh

```
./darknet detector map yolohw.names yolov3-tiny-aix2022.cfg yolov3-tiny-aix2022.weights -thresh 0.24 -quantized -save_params
```

Different threshold values result in different precision, recall, and F1 score.

To calculate mAP using the provided dataset, use 'map'. To visualize a detection result for an image, use 'test'.

To evaluate the quantized model, use '-quantized' option.

To save the quantized model (quantized weights, biases, and input scale factor) to bin/weights, use '-save_params' option.

mAP comparison

```
class_id = 30, name = coffee_mate_french_vanilla,        ap = 97.68 %
class_id = 31, name = pepperidge_farm_milk_chocolate_macadamia_cookies,         ap = 71.63
class_id = 32, name = kitkat_king_size,        ap = 85.32 %
class_id = 33, name = snickers,        ap = 36.60 %
class_id = 34, name = toblerone_milk_chocolate,        ap = 97.04 %
class_id = 35, name = clif_z_bar_chocolate_chip,        ap = 98.70 %
class_id = 36, name = nature_valley_crunchy_oats_n_honey,        ap = 72.28 %
class_id = 37, name = ritz_crackers,        ap = 97.73 %
class_id = 38, name = palmolive_orange,        ap = 55.47 %
class_id = 39, name = crystal_hot_sauce,        ap = 100.00 %
class_id = 40, name = tapatio_hot_sauce,        ap = 0.00 %
class_id = 41, name = nabisco_nilla_wafers,        ap = 0.00 %
class_id = 42, name = pepperidge_farm_milano_cookies_double_chocolate,    ap = 0.00 %
class_id = 43, name = campbells_chicken_noodle_soup,        ap = 0.00 %
class_id = 44, name = frappuccino_coffee,        ap = 0.00 %
class_id = 45, name = chewy_dips_chocolate_chip,        ap = 34.73 %
class_id = 46, name = chewy_dips_peanut_butter,        ap = 0.00 %
class_id = 47, name = nature_vally_fruit_and_nut,        ap = 0.00 %
class_id = 48, name = cheerios,        ap = 0.00 %
class_id = 49, name = lindt_excellence_cocoa_dark_chocolate,        ap = 0.00 %
class_id = 50, name = hersheys_symphony,        ap = 0.00 %
class_id = 51, name = campbells_chunky_classic_chicken_noodle,    ap = 0.00 %
class_id = 52, name = martinellis_apple_juice,        ap = 0.00 %
class_id = 53, name = dove_pink,        ap = 0.00 %
class_id = 54, name = dove_white,        ap = 0.00 %
class_id = 55, name = david_sunflower_seeds,        ap = 0.00 %
class_id = 56, name = monster_energy,        ap = 0.00 %
class_id = 57, name = act_ii_butter_lovers_popcorn,        ap = 0.00 %
class_id = 58, name = coca_cola_glass_bottle,        ap = 0.00 %
class_id = 59, name = twix,        ap = 0.00 %
 for thresh = 0.24, precision = 0.83, recall = 0.71, F1-score = 0.76
 for thresh = 0.24, TP = 1362, FP = 282, FN = 562, average IoU = 62.79 %

 mean average precision (mAP) = 0.831957, or 83.20 %
Total Detection Time: 32.000000 Seconds
(base) truongnx@marlin:~/aix2022/skeleton-v1.1/bin$
```

```
class_id = 30, name = coffee_mate_french_vanilla,        ap = 55.10 %
class_id = 31, name = pepperidge_farm_milk_chocolate_macadamia_cookies,         ap = 0.26 %
class_id = 32, name = kitkat_king_size,        ap = 0.23 %
class_id = 33, name = snickers,        ap = 5.05 %
class_id = 34, name = toblerone_milk_chocolate,        ap = 20.10 %
class_id = 35, name = clif_z_bar_chocolate_chip,        ap = 59.39 %
class_id = 36, name = nature_valley_crunchy_oats_n_honey,        ap = 22.12 %
class_id = 37, name = ritz_crackers,        ap = 45.45 %
class_id = 38, name = palmolive_orange,        ap = 54.55 %
class_id = 39, name = crystal_hot_sauce,        ap = 0.00 %
class_id = 40, name = tapatio_hot_sauce,        ap = 0.00 %
class_id = 41, name = nabisco_nilla_wafers,        ap = 0.00 %
class_id = 42, name = pepperidge_farm_milano_cookies_double_chocolate,    ap = 0.00 %
class_id = 43, name = campbells_chicken_noodle_soup,        ap = 0.00 %
class_id = 44, name = frappuccino_coffee,        ap = 0.00 %
class_id = 45, name = chewy_dips_chocolate_chip,        ap = 0.00 %
class_id = 46, name = chewy_dips_peanut_butter,        ap = 0.00 %
class_id = 47, name = nature_vally_fruit_and_nut,        ap = 0.00 %
class_id = 48, name = cheerios,        ap = 0.00 %
class_id = 49, name = lindt_excellence_cocoa_dark_chocolate,        ap = 0.00 %
class_id = 50, name = hersheys_symphony,        ap = 0.00 %
class_id = 51, name = campbells_chunky_classic_chicken_noodle,    ap = 0.00 %
class_id = 52, name = martinellis_apple_juice,        ap = 0.00 %
class_id = 53, name = dove_pink,        ap = 0.00 %
class_id = 54, name = dove_white,        ap = 0.00 %
class_id = 55, name = david_sunflower_seeds,        ap = 0.00 %
class_id = 56, name = monster_energy,        ap = 0.00 %
class_id = 57, name = act_ii_butter_lovers_popcorn,        ap = 0.00 %
class_id = 58, name = coca_cola_glass_bottle,        ap = 0.00 %
class_id = 59, name = twix,        ap = 0.00 %
 for thresh = 0.24, precision = 0.93, recall = 0.03, F1-score = 0.05
 for thresh = 0.24, TP = 54, FP = 4, FN = 1870, average IoU = 57.32 %

 mean average precision (mAP) = 0.336020, or 33.60 %
Total Detection Time: 27.000000 Seconds
(base) truongnx@marlin:~/aix2022/skeleton-v1.1/bin$
```

FP model: 83.20%

Naïvely quantized INT8 model: 33.60%

Try to achieve higher mAP for the quantized model by implementing better quantization!

Data preparation

- Store weights and biases in hexadecimal files (32 bits per line)
  - RTL simulation
  - Read by the Host PC to send to the FPGA board
- You can adjust this code to save activations which are used for RTL verification

```c
// Save quantized weights, bias, and scale
void save_quantized_model(network net) {
    int j;
    for (j = 0; j < net.n; ++j) {
        layer *l = &net.layers[j];
        if (l->type == CONVOLUTIONAL) {
            size_t const weights_size = l->size*l->size*l->c*l->n;
            size_t const filter_size = l->size*l->size*l->c;

            printf(" Saving quantized weights, bias, and scale for CON

            char weightfile[30];
            char biasfile[30];
            char scalefile[30];

            sprintf(weightfile, "weights/CONV%d_W.txt", j);
            sprintf(biasfile, "weights/CONV%d_B.txt", j);
            sprintf(scalefile, "weights/CONV%d_S.txt", j);

```

```c
FILE *fp_w = fopen(weightfile, "w");
for (k = 0; k < weights_size; k = k + 4) {
    uint8_t first = k < weights_size ? l->weights_int8[k] : 0;
    uint8_t second = k+1 < weights_size ? l->weights_int8[k+1] : 0;
    uint8_t third = k+2 < weights_size ? l->weights_int8[k+2] : 0;
    uint8_t fourth = k+3 < weights_size ? l->weights_int8[k+3] : 0;
    fprintf(fp_w, "%02x%02x%02x%02x\n", first, second, third, fourth);
}
fclose(fp_w);

FILE *fp_b = fopen(biasfile, "w");
for (k = 0; k < l->n; k = k + 4) {
    uint16_t first = k < l->n ? l->biases_quant[k] : 0;
    uint16_t second = k+1 < l->n ? l->biases_quant[k+1] : 0;
    fprintf(fp_b, "%04x%04x\n", first, second);
}
fclose(fp_b);
```

Incoming lecture …

- Computing units
    - DSP
    - MAC