FPGA-based Convolutional Neural Network Accelerator

Ke Xu

Xingyu Hou

Manqi Yang

Wenqi Jiang

COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

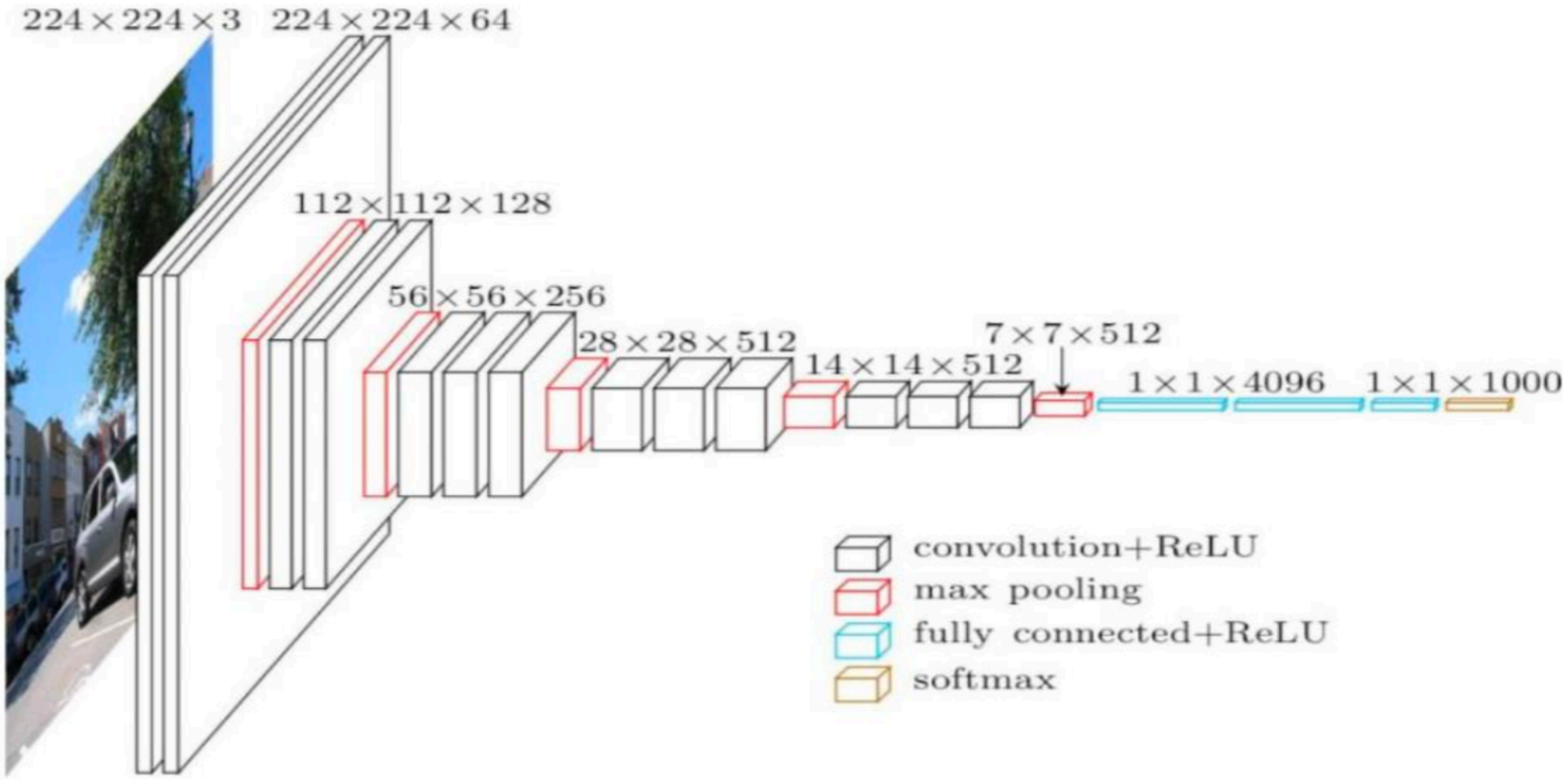
Outline

- Background
- Software Implementation
  - Python / C implementation of VGG-16
  - Profiling and acceleration strategy
  - Dynamic fixed point conversion / operation
- Hardware Implementation
  - SDRAM and DMA
  - Dataflow design
  - PE implementation
- Conclusion

Background

- Convolutional Neural Networks (CNN)
  - Computer Vision
    - Image Classification, Object Detection, Semantic Segmentation
  - Mainly composed of convolutions and matrix multiplications
  - Both these computations are highly parallelizable
- Dedicated Hardware
  - CPU: latency oriented; not good at massively parallel computations
  - FPGA: by using many Processing Elements (PE), FPGA can compute many output elements in parallel

VGG16



VGG16 Architecture

Software Simulation

- For reference, download a Keras-based VGG16 implementation and weights of the model
- Reproduce the VGG16 model using Python, including convolution layers, fully-connected layers, pooling layers and activation functions
- Compare the result with the Keras model to verify the correctness
- Port the python implementation to C for later use

Software Simulation

Part of our python and C implementations

```python
def conv_forward(x, w, b, pad, stride):
    """..."""
    batch, height, width, channels = x.shape
    num_of_filters, filter_height, filter_width, channels_f = w.shape
    assert channels == channels_f

    new_height = int(np.floor((height - filter_height + 2 * pad) / stride) + 1)
    new_width = int(np.floor((width - filter_width + 2 * pad) / stride) + 1)
    A = np.zeros((batch, new_height, new_width, num_of_filters))
    x_pad = np.zeros((batch, height + 2*pad, width+2*pad, channels))

    for bt in range(batch):
        for i in range(height):
            for j in range(width):
                for cn in range(channels):
                    x_pad[bt,i+pad,j+pad,cn] = x[bt,i,j,cn]

    for bt in range(batch):
        for ft in range(num_of_filters):
            for i in range(new_height):
                for j in range(new_width):
                    A[bt,i,j,ft] = b[ft] + np.sum(w[ft,:,:,:] *
                                   x_pad[bt, i*stride: i*stride + filter_height,
                                   j * stride: j*stride + filter_width,:])

    return A
```

```c
// output_feature_map[batch_index][output_feature_map_index]
int current_output_feature_map_index = batch_index * fc_output_size +
    output_feature_map_index;

// initialize to 0
output_feature_map[current_output_feature_map_index] = 0;

for (int input_feature_map_index = 0;
        input_feature_map_index < fc_input_size;
        input_feature_map_index++) {

    // output_feature_map[batch_index][output_feature_map_index] +=
    //      input_feature_map[batch_index][input_feature_map_index] *
    //      kernel[output_feature_map_index][input_feature_map_index]

    // input_feature_map[batch_index][input_feature_map_index]
    int current_input_feature_map_index = batch_index * fc_input_size +
        input_feature_map_index;

    // kernel[output_feature_map_index][input_feature_map_index]
    int current_kernel_index = input_feature_map_index * fc_output_size +
        output_feature_map_index;

    // do multiplication, add to previous value
    output_feature_map[current_output_feature_map_index] +=
        input_feature_map[current_input_feature_map_index] *
        kernel[current_kernel_index];
}
// add bias: bias[current_output_feature_map_index]
output_feature_map[current_output_feature_map_index] +=
    bias[output_feature_map_index];
```

Algorithm Optimization - Winograd

- Winograd
  - Memory consuming (need extra space to store intermediate results)
  - Reduce 1/3 multiplications when using our dataflow pattern, while consumes about 2x of memory usage

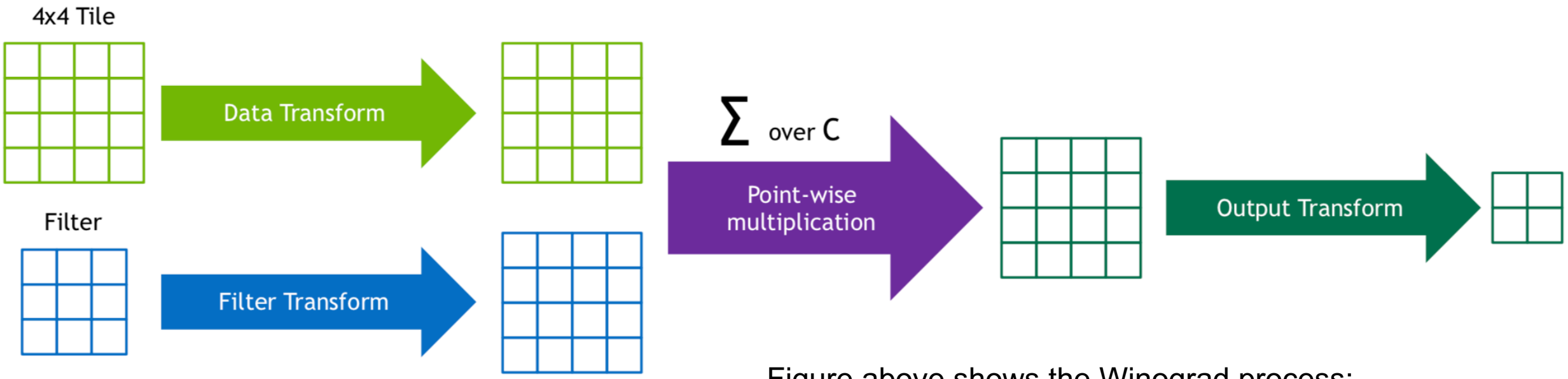


Figure above shows the Winograd process:

- Directly convolution: 4 * 3 * 3 = 36 multiplications
- Winograd convolution: 4 * 4 = 16 multiplications
- 2.25x speed up

Software Profiling

- Implement the software profiling in C to see which parts should we accelerate on FPGA
- Neglect max pooling, ReLU and softmax function, since the time they consume is negligible
- Time consumed comparison between convolution layer and fully-connected layer using i5-8259U (without loading weights and data)

	Convolution Layers	Fully-connected Layers
Time Consumed / sec	92.02	4.15
Time Percentage / %	95.67	4.32

Software Profiling

- Time complexity analysis
  - Convolutional layer:

    O(conv_height * conv_width * conv_channel * conv_number * input_width * input_height)

    e.g. 3 x 3 x 256 x 512 x 28 x 28 = 924,844,032
  - Fully-connected layer:

    O(fc_height * fc_width)

    e.g. 4,096 x 4,096 = 16,777,216
  - Convolutional layer > fully-connected layer

Software Profiling

- Memory consuming analysis
  - Convolutional layer:

    O(conv_height * conv_width * conv_channel * conv_number)

    e.g. 3 x 3 x 256 x 512 = 1,179,648
  - Fully-connected layer:

    O(fc_height * fc_width)

    e.g. 4,096 x 4,096 = 16,777,216
  - Convolutional layer < fully-connected layer

Software Profiling

- Computation intensive VS memory access intensive

	Convolution Layers	Fully-connected Layers	Ratio (conv / fc)
Weights number	14,710,464	123,633,664	0.12x
Multiplications number	16,271,474,688	123,633,664	131.61x

- Accelerator strategy
  - Compute convolutional layers on FPGA
  - Compute fully-connected layers using CPU
  - If we compute both these layers on FPGA
    - allocate some FPGA resources, e.g. DSPs, to fully-connected layers, which will slow down convolutions
    - copy weights (>200M bytes) from DRAM to SDRAM, which is time-consuming (>30s)

Fixed Point Computation

- FPGA is good at fixed point operations, so we use fixed point instead of floating point to do convolutions
- Challenge:
  - Weights, input image and intermediate results have different ranges
  - Can not use a unified decimal point place, e.g. in the middle of a fixed point:1100.0011

```
Intermediate outputs of each layer:
Layer 1:        max:    955.85339       mean:   26.134333
Layer 2:        max:    3689.4717       mean:   150.90759
Layer 3:        max:    8024.4507       mean:   207.26363
Layer 4:        max:    11275.772       mean:   229.63486
Layer 5:        max:    15204.325       mean:   284.25253
Layer 6:        max:    15991.411       mean:   329.49692
Layer 7:        max:    20796.828       mean:   199.77206
Layer 8:        max:    12113.225       mean:   174.05199
Layer 9:        max:    6186.3657       mean:   97.339676
Layer 10:       max:    2941.2952       mean:   21.802027
Layer 11:       max:    1955.9144       mean:   22.766939
Layer 12:       max:    953.95642       mean:   9.6327209
Layer 13:       max:    552.76392       mean:   1.3137941
Layer 14:       max:    41.791012       mean:   1.7962291
Layer 15:       max:    15.469688       mean:   0.58563447
Layer 16:       max:    22.734434       mean:   1.0392156
```

Fixed Point Computation

- Solution: dynamic fixed point
  - 1100.0011 VS 10.101100
  - length allocate to integer and decimal part differs from layer to layer
  - use 1000 samples to measure the intermediate output ranges of each layer
  - can be decided before runtime

```
Digits allocation to integer part and decimal part:
(Use fixed point 16, with 1 digit as sign)
Input:          Max:    255.0           Integer digits: 8       Decimal digits: 7
Layer 1:        Max:    955.85339       Integer digits: 10      Decimal digits: 5
Layer 2:        Max:    3689.4717       Integer digits: 12      Decimal digits: 3
Layer 3:        Max:    8024.4507       Integer digits: 13      Decimal digits: 2
Layer 4:        Max:    11275.772       Integer digits: 14      Decimal digits: 1
Layer 5:        Max:    15204.325       Integer digits: 14      Decimal digits: 1
Layer 6:        Max:    15991.411       Integer digits: 14      Decimal digits: 1
Layer 7:        Max:    20796.828       Integer digits: 15      Decimal digits: 0
```

Fixed Point Computation

- Conversion
  - Convert images and weights to dynamic fixed point numbers
  - Save these numbers and feed them into our C program
- Simulation
  - Dynamic fixed point operations
    - Inputs and outputs can have different decimal point place
    - e.g. 0011.1010 x 011.00000 = 01010.111 (3.625 x 3 = 10.875)
  - Simulate fixed point operations on hardware
  - Helpful when debugging hardware functions

Fixed Point Computation

- Build tools for fixed point conversions and verification
- Some of the functions we build
  - Conversion

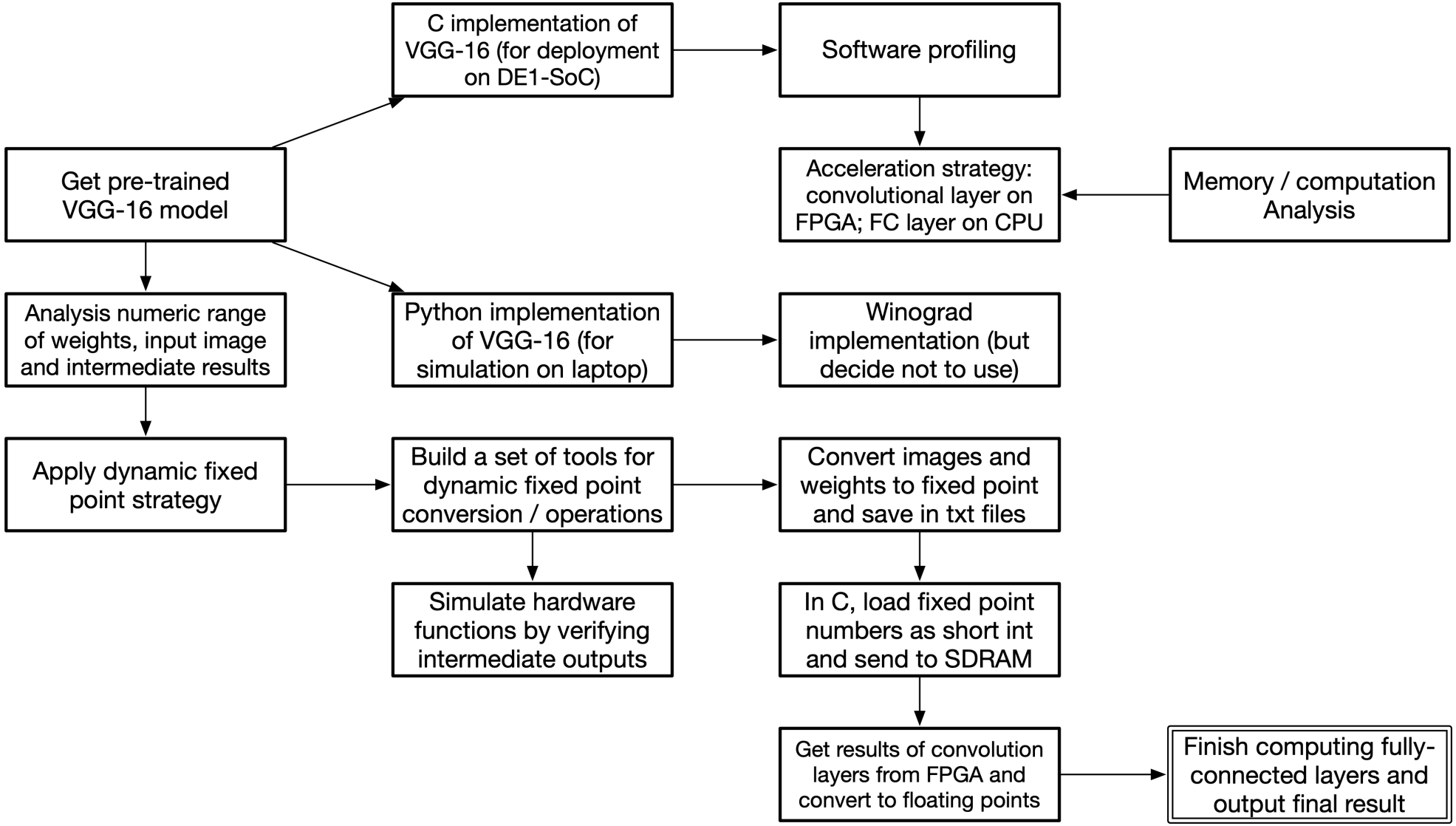
    float2fixed, fixed2float
  - Dynamic fixed point operations

    fixed_add, fixed_mul, fixed_shift, inverse, ReLU, etc.
  - Other functions

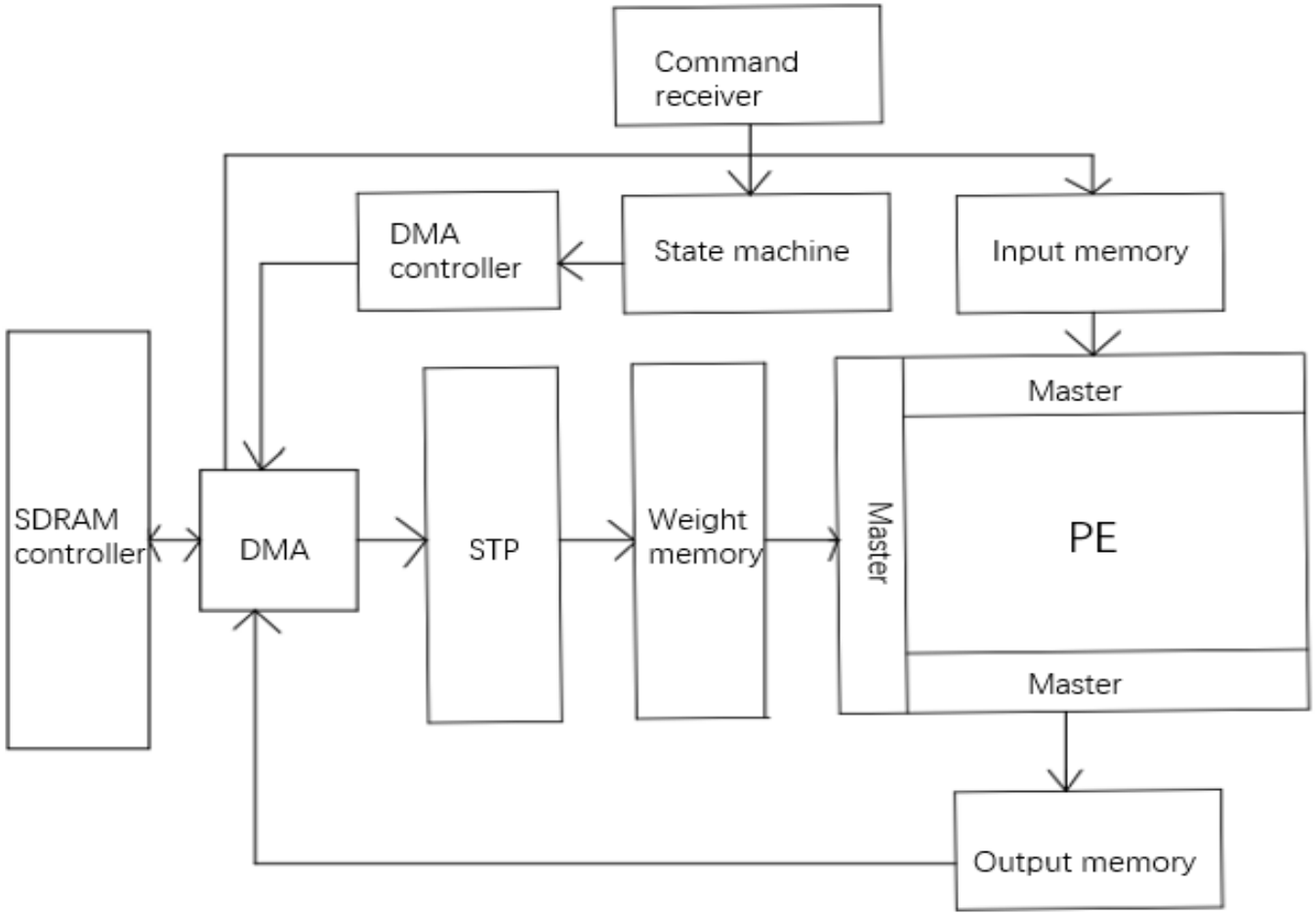
    digit_of // how many digits should we assign to integer and decimal parts

```
def fixed_mul(x1, x2, integer_x1=8, integer_x2=8, integer_result=8):
    """
    fixed point multiplication
    input: x1, x2 -> 16 bit fixed point
    integer_x1 / x2 : where is the input decimal point
    integer_result: where is the result decimal point
    return:
        result -> 16 bit fixed point
    """
    adders = []

    fill_x1 = _helper_filler1(x1)
    fill_x2 = _helper_filler1(x2)

    # save all adders
    for i in range(32):
        if fill_x1[i] == '1':
            before = i
            after = 31 - i
            filled_num = _helper_filler2(fill_x2, before, after)
            adders.append(filled_num)

    # add them up, a 30 bit number
    add_result = _helper_mul_add(adders)
    cut_result = add_result[32:]

    int_part = integer_x1 + integer_x2
    new_int_start = int_part - integer_result + 1 # sign 1 bit

    result = cut_result[new_int_start: new_int_start + 16]

    return result
```

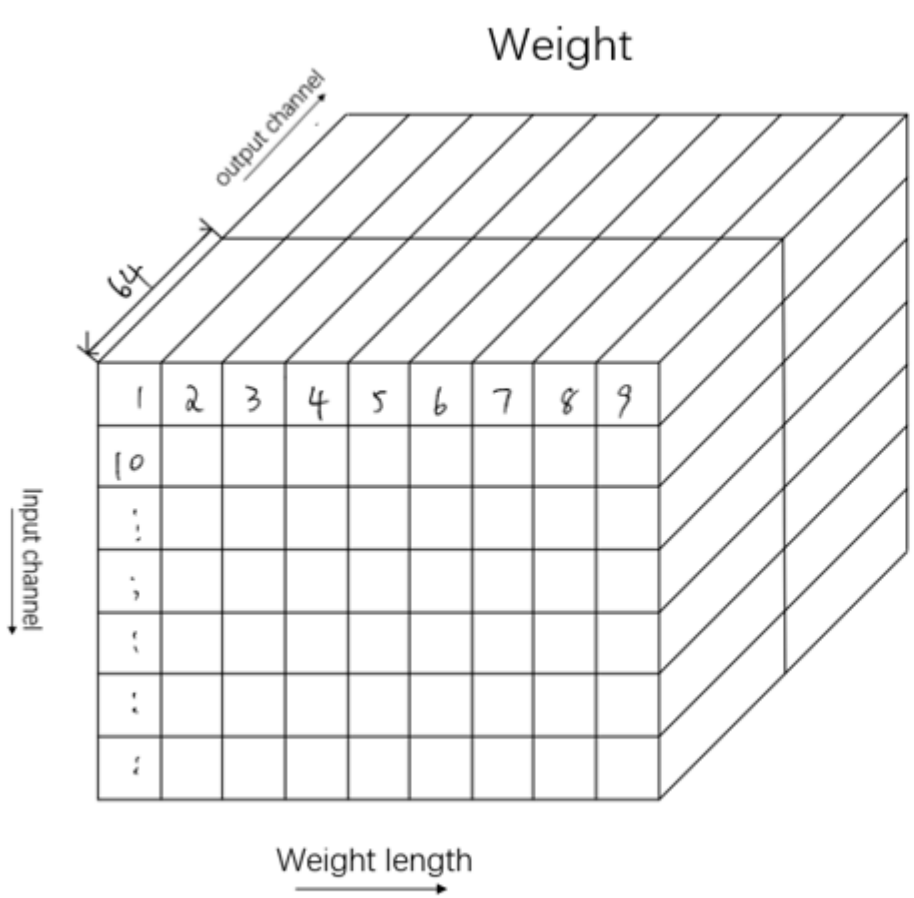
Software Summary

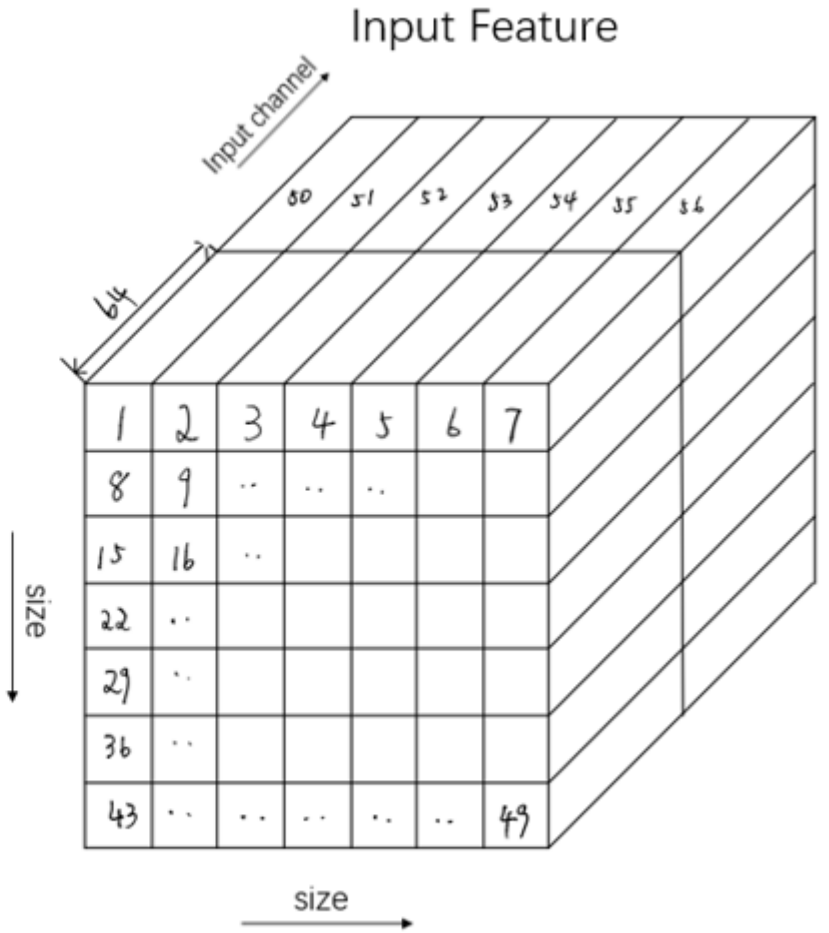


Hardware System Structure



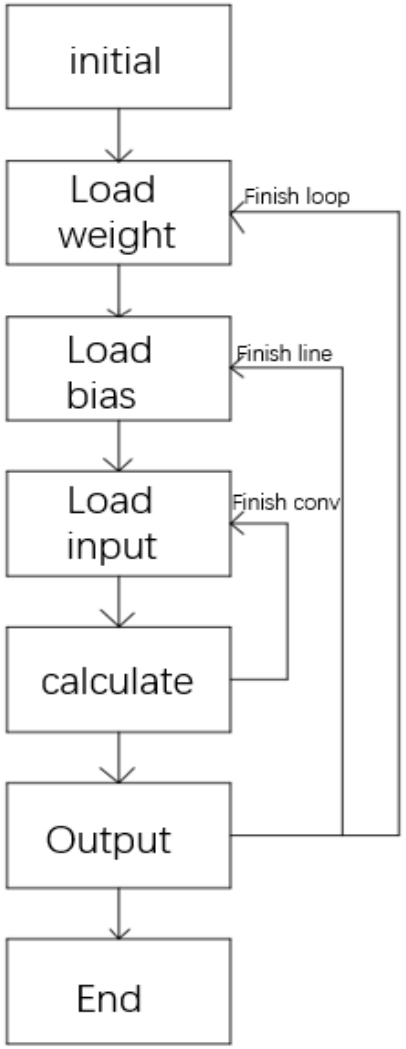
Data Alignment in SDRAM





Dataflow Design

(in channel, out channel, weight)
(1,1,1)
(1,2,1)
(1,3,1)
..
(1,64,1)
(2,1,1)
(2,2,1)
..
..
(64,64,1)
(1,1,2)
(1,2,2)
..
..
(64,64,2)
..
..
(64,64,9)



Q & A