

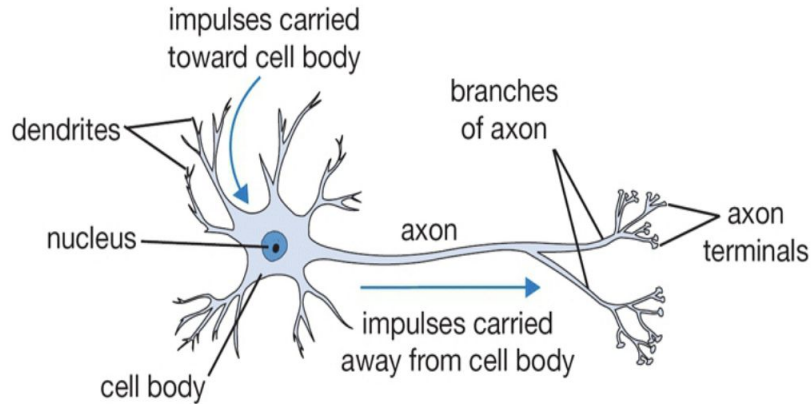
# DNN Accelerators and HLS

*Qijing (Jenny) Huang*

# Outline

1. Deep Neural Network (DNN)
2. Design Methodology
3. Accelerator Architecture
4. High-level Synthesis (HLS)

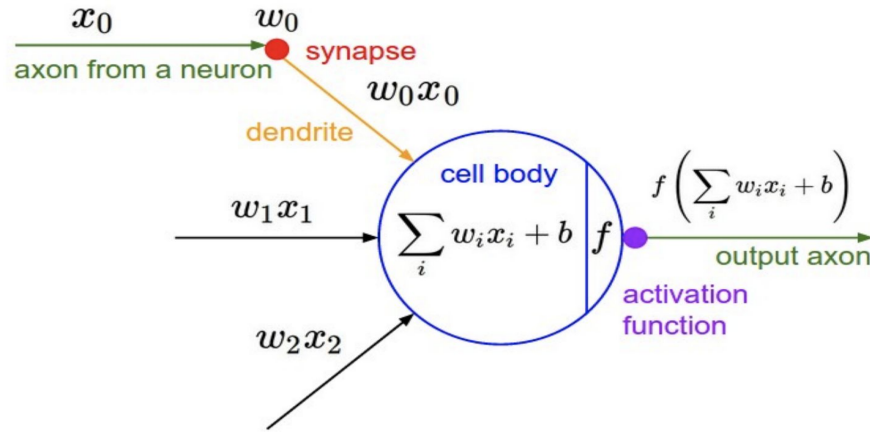
# Learning from the Brain



## Integrate and Fire

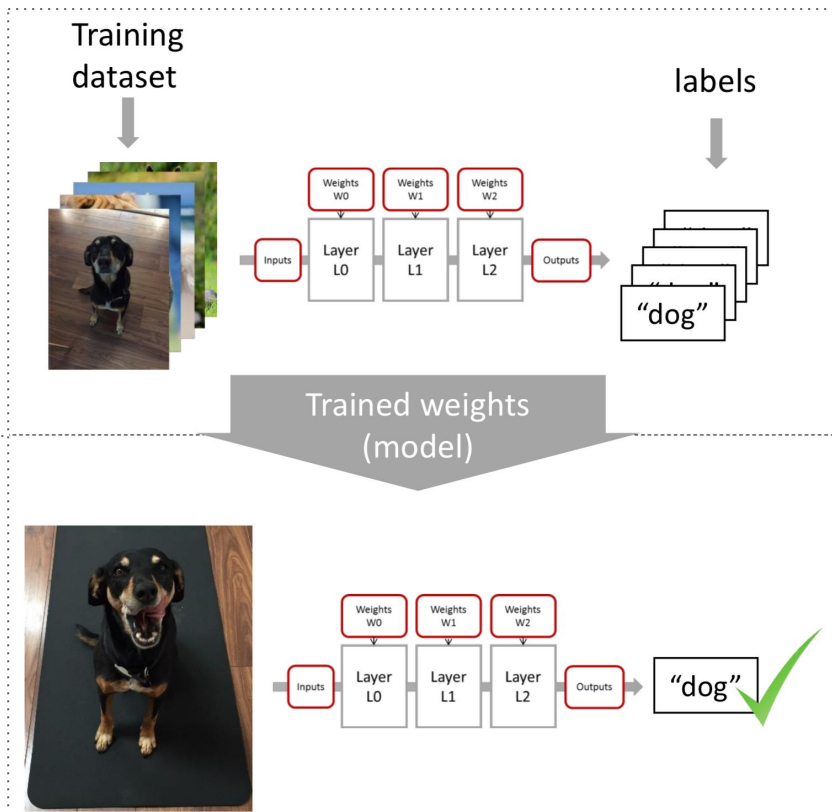
- The basic computational unit of the brain is a **neuron**
  - 86B neurons in the brain
- Neurons are connected with nearly  $10^{14} - 10^{15}$  **synapses**
- Neurons receive input signal from **dendrites** and produce output signal along **axon**, which interact with the dendrites of other neurons via **synaptic weights**
- **Synaptic weights** – learnable & control influence strength

# Neural Networks



- NNs are usually feed forward computational graphs constructed from one or more layers
- The “Neuron” computes:
  - **Integrate** - typically linear transform (dot-product of receptive field)
  - **Fire** - followed by a non-linear “activation” function

# Training vs. Inference



## Training

Process for a machine to learn by optimizing models (weights) from labeled data.

Typically computed in the cloud

## Inference

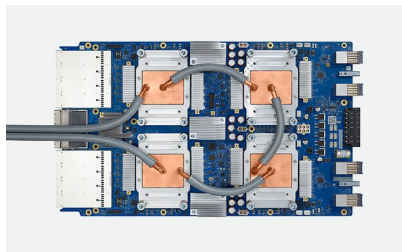
Using trained models to predict or estimate outcomes from new inputs.

Deployment at the edge

# Many AI Chips ~ 85 AI chip companies worldwide

## In the Cloud (Training + Inference)

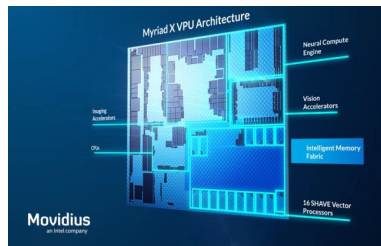
- 10s TFLOPs
- 10s MB on-chip memory
- 8 - 32 bit precision
- 700 MHz - 1 GHz
- 10-100s Watts



Cloud TPU v3 (45 TFLOP/s)

## At the Edge (Inference)

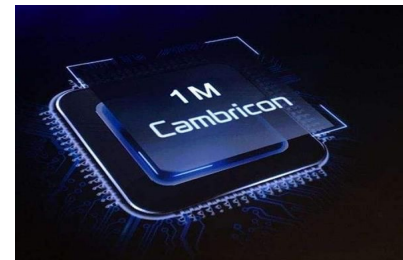
- 100s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 50 MHz - 400 MHz
- 1-10s Watts



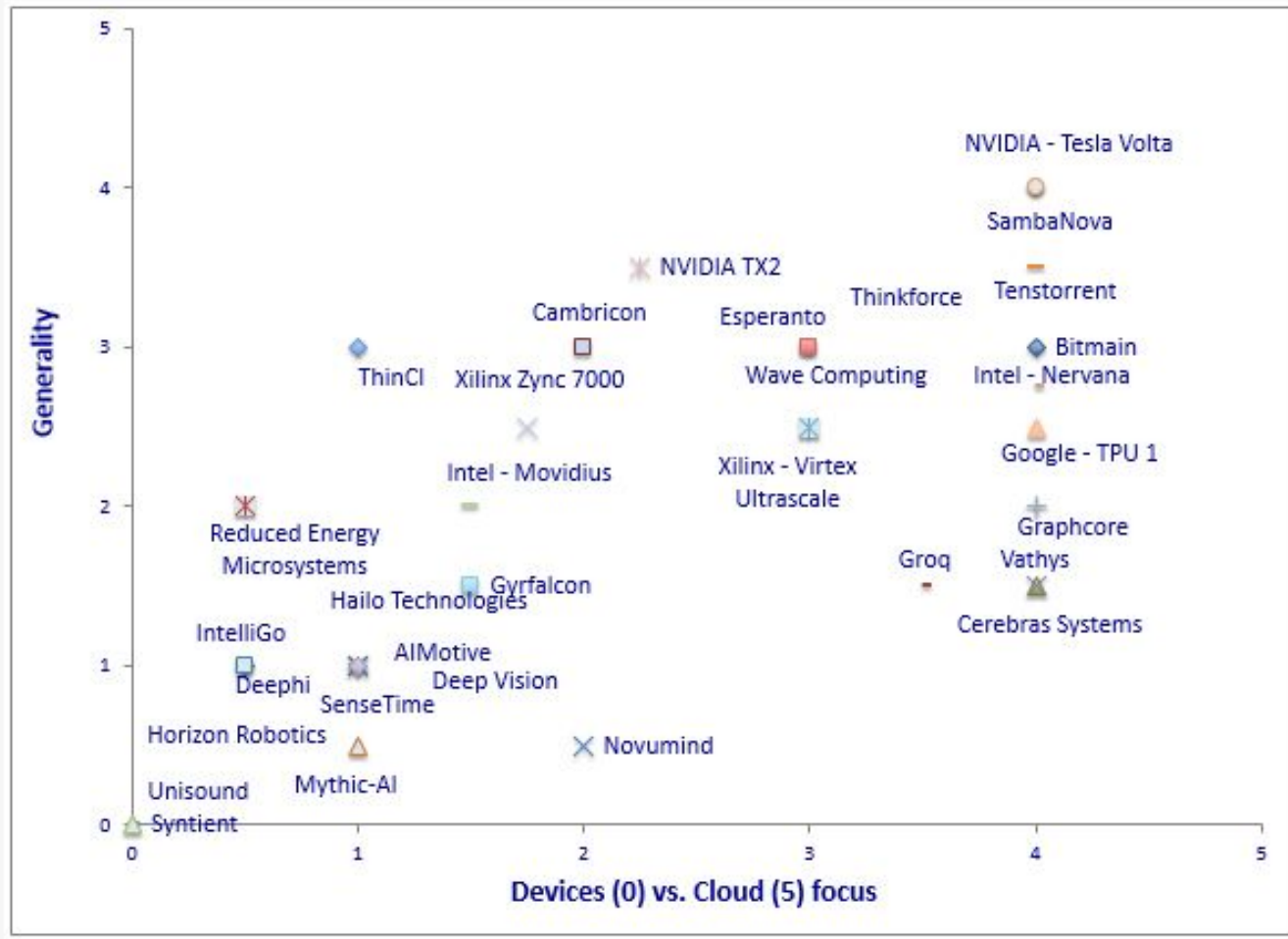
Intel Movidius (4 TFLOP/s)

## In the Edge SoC/SiP (Inference)

- 10s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 600 MHz - 1 GHz
- 10-100s mWatts



Cambricon-1M IP



\* Image from [https://www.electronicproducts.com/Digital\\_ICs/Designer\\_s\\_Guide\\_Selecting\\_AI\\_chips\\_for\\_embedded\\_designs.aspx](https://www.electronicproducts.com/Digital_ICs/Designer_s_Guide_Selecting_AI_chips_for_embedded_designs.aspx)

# Computer Vision Applications



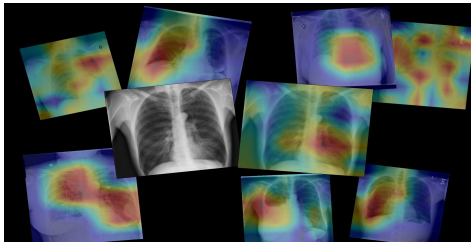
Autonomous Vehicles



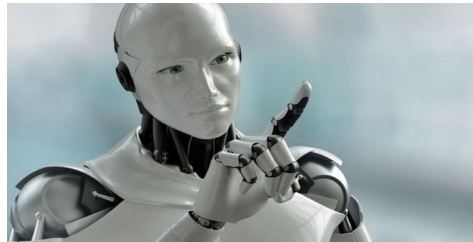
Security Camera



Drones



Medical Imaging



Robots



Mobile Applications



# Computer Vision Tasks

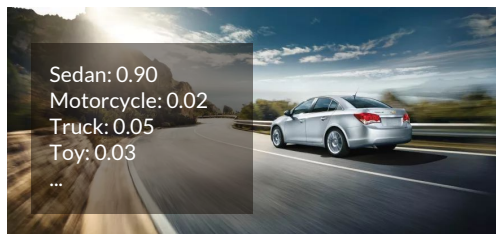
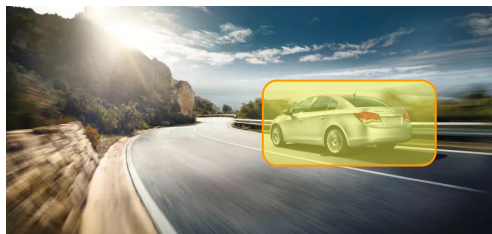


Image Classification



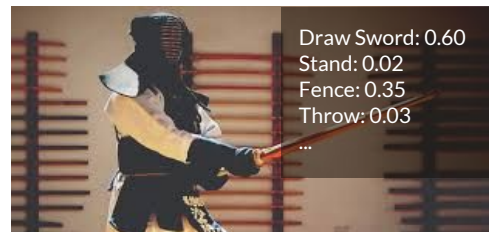
Object Detection



Semantic Segmentation



Super Resolution



Activity Recognition

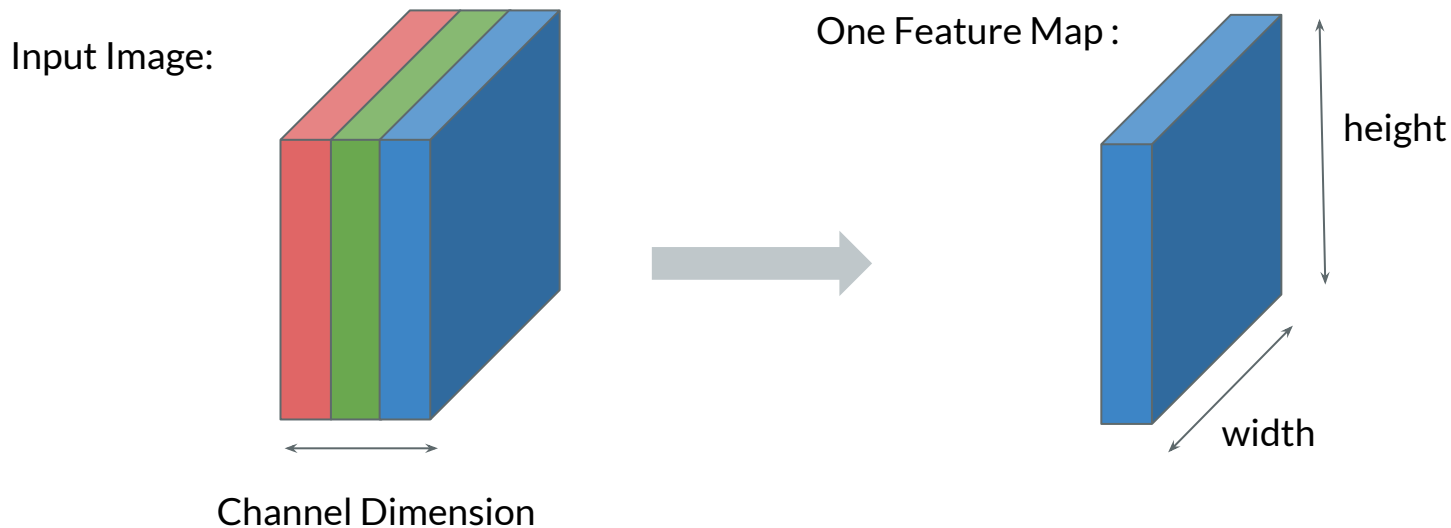
# Deep Neural Network

# Common Operations

- Convolution (Dilated, Transposed, 3D and etc.)
- ReLU
- Pooling (Average, Max)
- Fully-Connected
- Batch Normalization

# Activation/Feature Maps

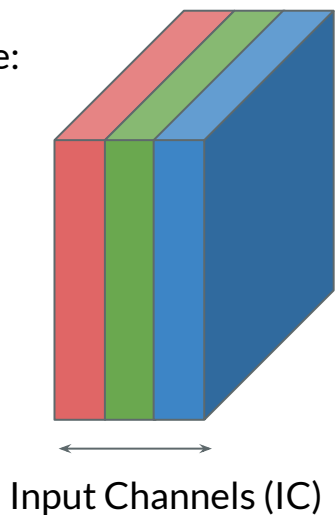
- Input images have three dimensions with RGB channels
- Intermediate data have more channels after performing convolution
- We refer to them as feature maps



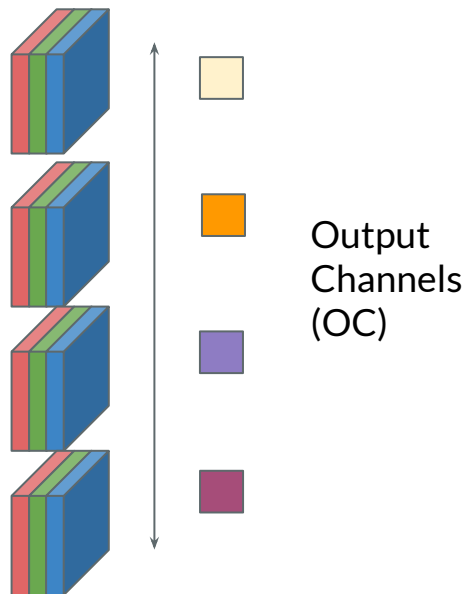
# Weights/Kernels

- weights for full convolution typically have four dimensions:
  - input channels, width, height, output channels
- input channel size matches the channel dimension of input features
- output channel size specifies the channel dimension of output features

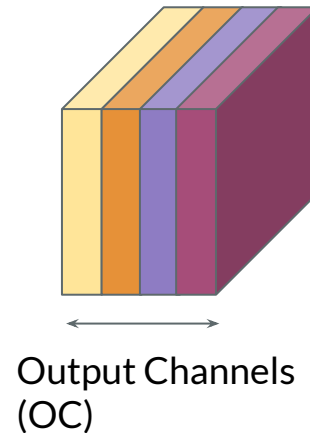
Input Image:



Weights:



Output Image:



# 3x3 Convolution - Spatially

3 <sub>0</sub>	3 <sub>1</sub>	2 <sub>2</sub>	1	0
0 <sub>2</sub>	0 <sub>2</sub>	1 <sub>0</sub>	3	1
3 <sub>0</sub>	1 <sub>1</sub>	2 <sub>2</sub>	2	3
2	0	0	2	2
2	0	0	0	1

Input feature map

12	12	17
10	17	19
9	6	14

Output feature map

0 <sub>2</sub>	0 <sub>0</sub>	0 <sub>1</sub>	0	0	0	0
0 <sub>1</sub>	2 <sub>0</sub>	2 <sub>0</sub>	3	3	3	0
0 <sub>0</sub>	0 <sub>1</sub>	1 <sub>1</sub>	3	0	3	0
0	2	3	0	1	3	0
0	3	3	2	1	2	0
0	3	3	0	2	3	0
0	0	0	0	0	0	0

Input feature map

1	6	5
7	10	9
7	10	8

Output feature map

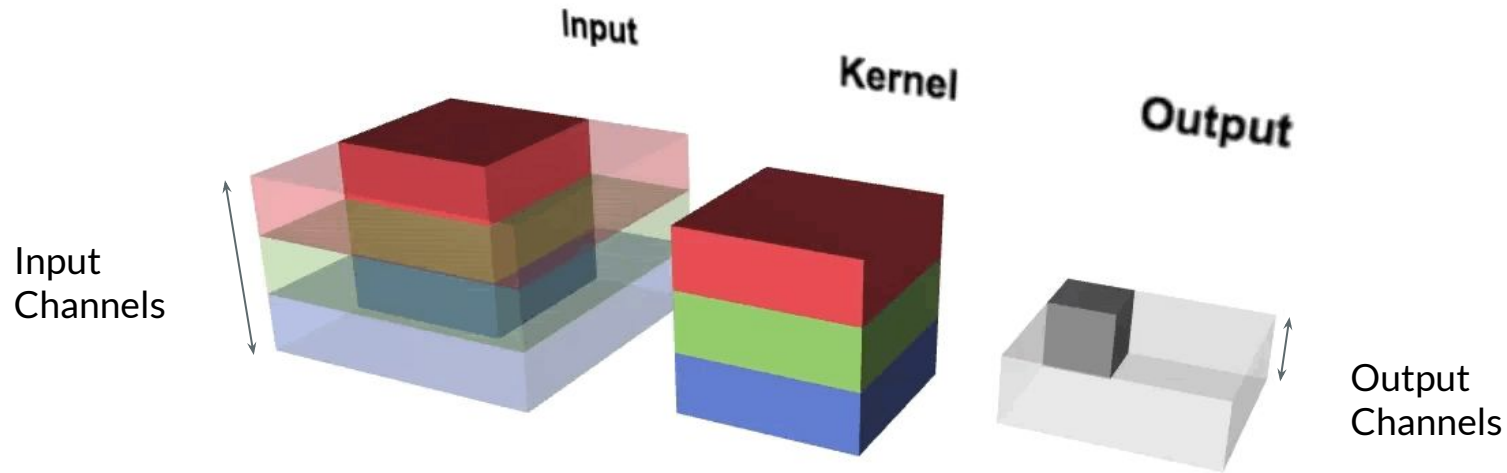
- 3x3 Conv with No Stride, No Padding
- Weights =  $[[0, 1, 2], [2, 2, 0], [0, 1, 2]]$

- 3x3 Conv with Stride 2, Padding 1
- Weights =  $[[2, 0, 1], [1, 0, 0], [0, 1, 1]]$

$$O_{00} = I_{00} \times W_{00} + I_{01} \times W_{01} + I_{02} \times W_{02} + I_{10} \times W_{10} + I_{11} \times W_{11} + I_{12} \times W_{12} + I_{20} \times W_{20} + I_{21} \times W_{21} + I_{22}$$

\* gif from [http://deeplearning.net/software/theano\\_versions/dev/\\_images/](http://deeplearning.net/software/theano_versions/dev/_images/)

# 3x3 Convolution - 3D



# 3x3 Convolution - 3D

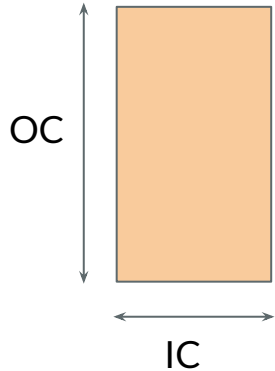
<http://cs231n.github.io/assets/conv-demo/index.html>



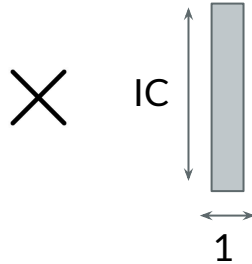
# Fully-Connected Layer (FC)

- Each input activation is connected to every output activation
- Essentially a matrix-vector multiplication

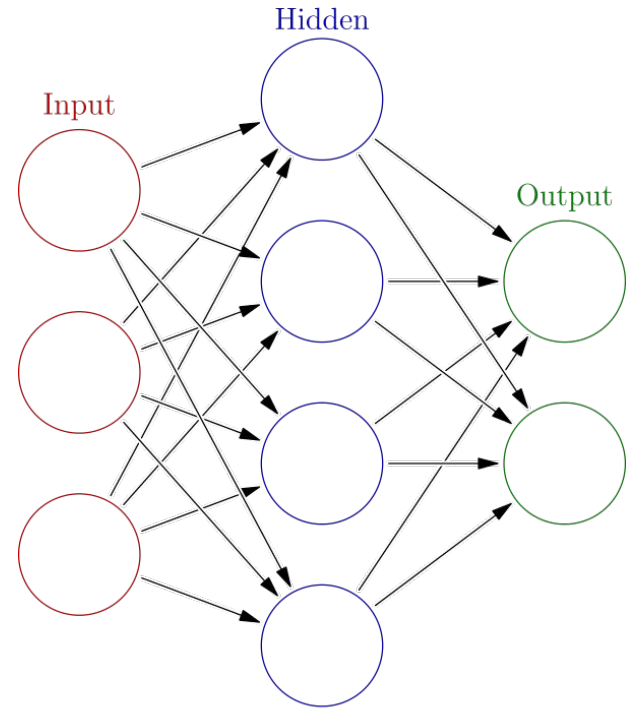
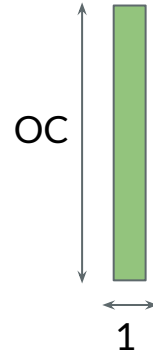
Weights:  
 $OC \times IC$



Input Activations:  
 $IC \times 1$

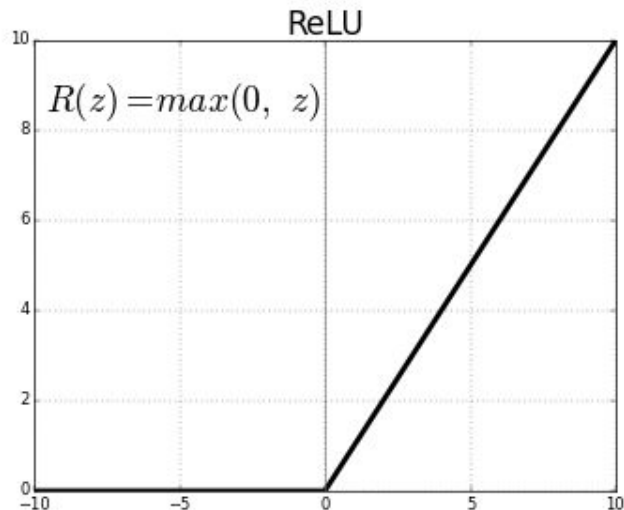


Output Activations:  
 $OC \times 1$



# ReLU Activation Function

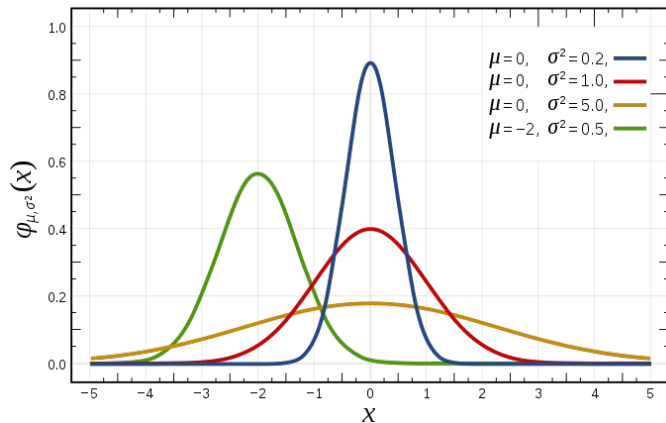
- Implements the concept of “Firing”
- Introduces non-linearity
- Rectified Linear Unit
  - $R(z) = \max(0, z)$
- Not differentiable at 0



# Batch Normalization (BN)

- Shifts and scales activations to achieve zero-centered distribution with unit variance

- Subtracts mean
- Divides by standard deviation



**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;  
Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

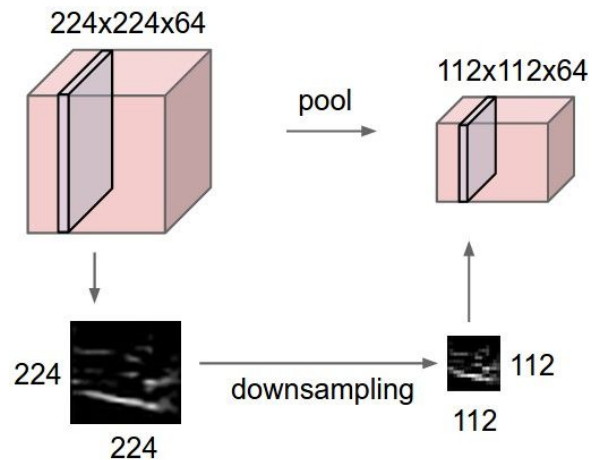
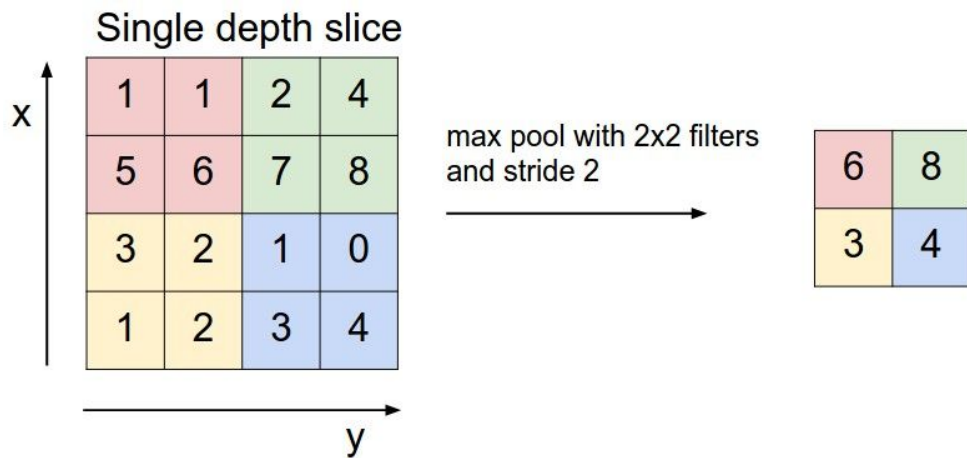
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

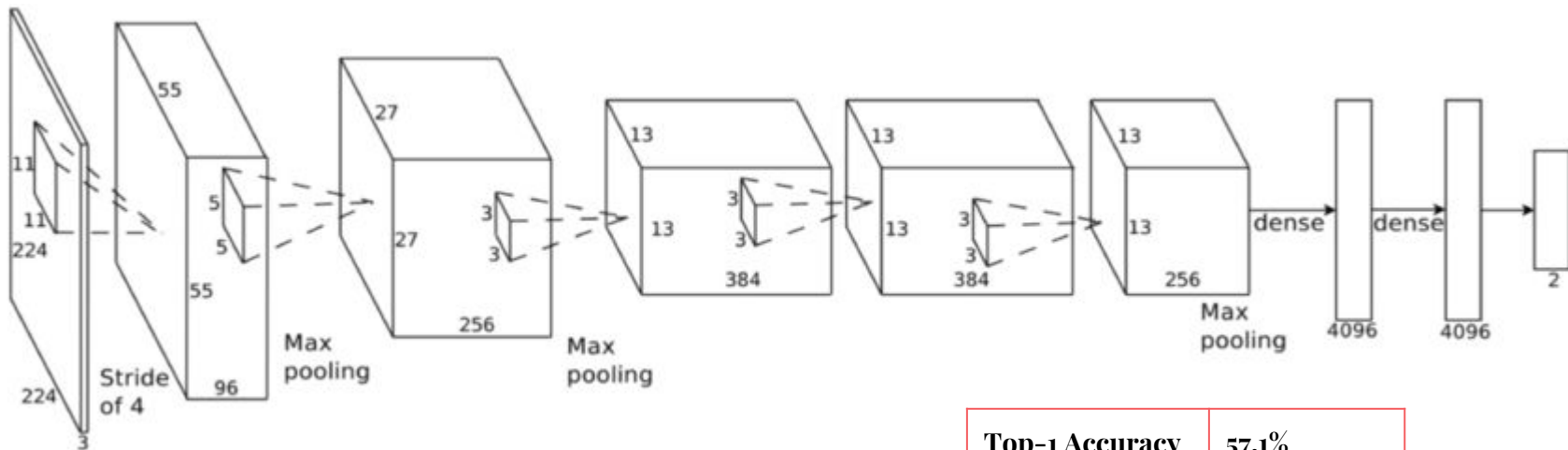
$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

# Pooling

- Downsamples
  - Takes the maximum
  - Takes the average
- Operates at each feature map independently

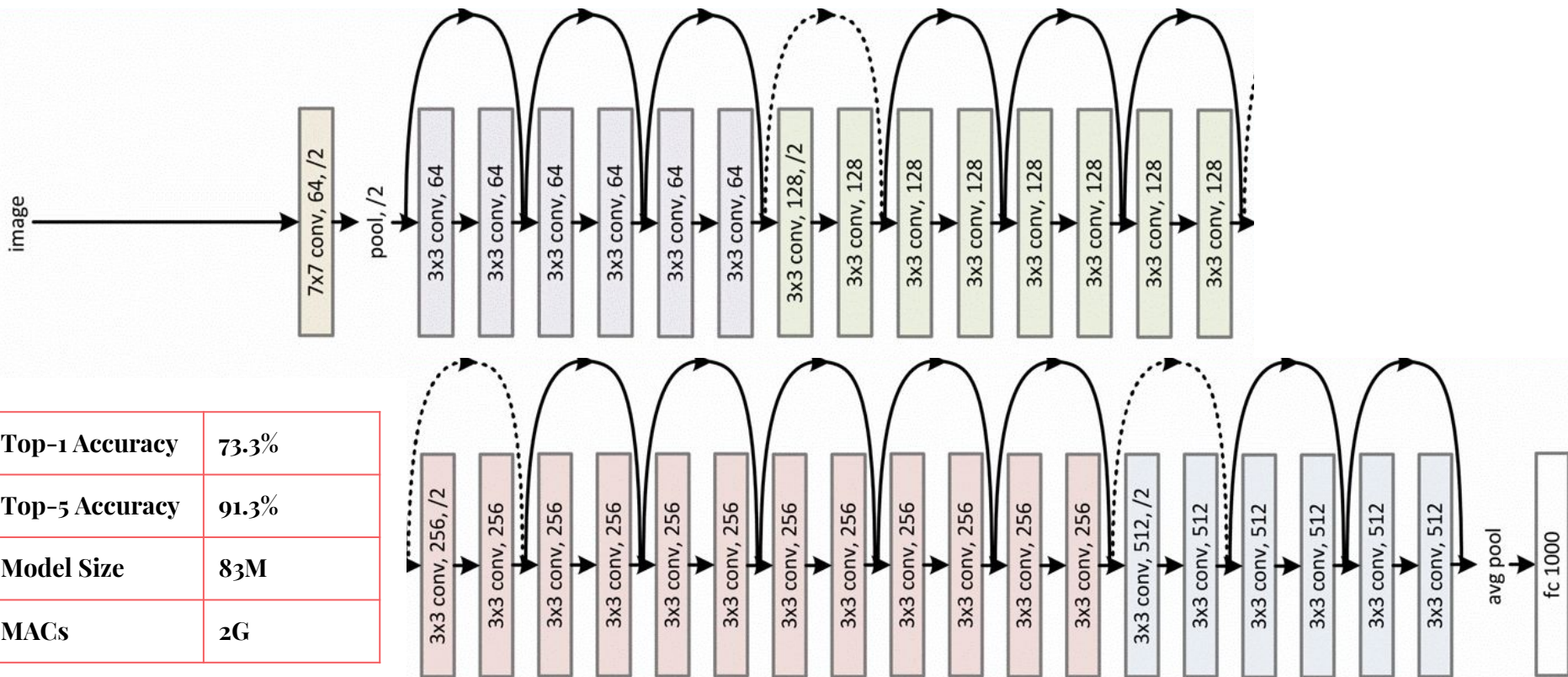


# Full DNN Example: AlexNet



<b>Top-1 Accuracy</b>	<b>57.1%</b>
<b>Top-5 Accuracy</b>	<b>80.2%</b>
<b>Model Size</b>	<b>61M</b>
<b>MACs</b>	<b>725M</b>

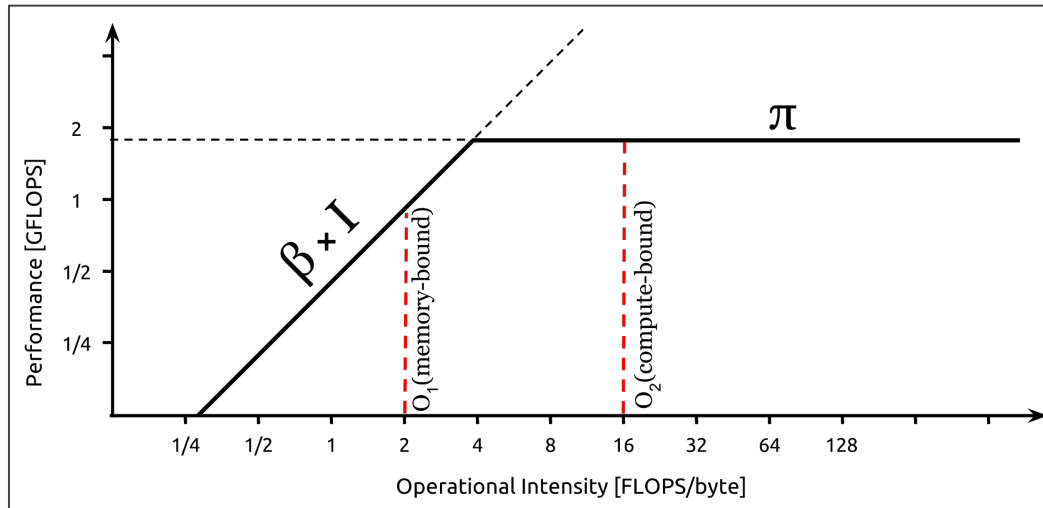
# Full DNN Example: ResNet-34



Top-1 Accuracy	73.3%
Top-5 Accuracy	91.3%
Model Size	83M
MACs	2G

# Design Methodology

# The Roofline Model



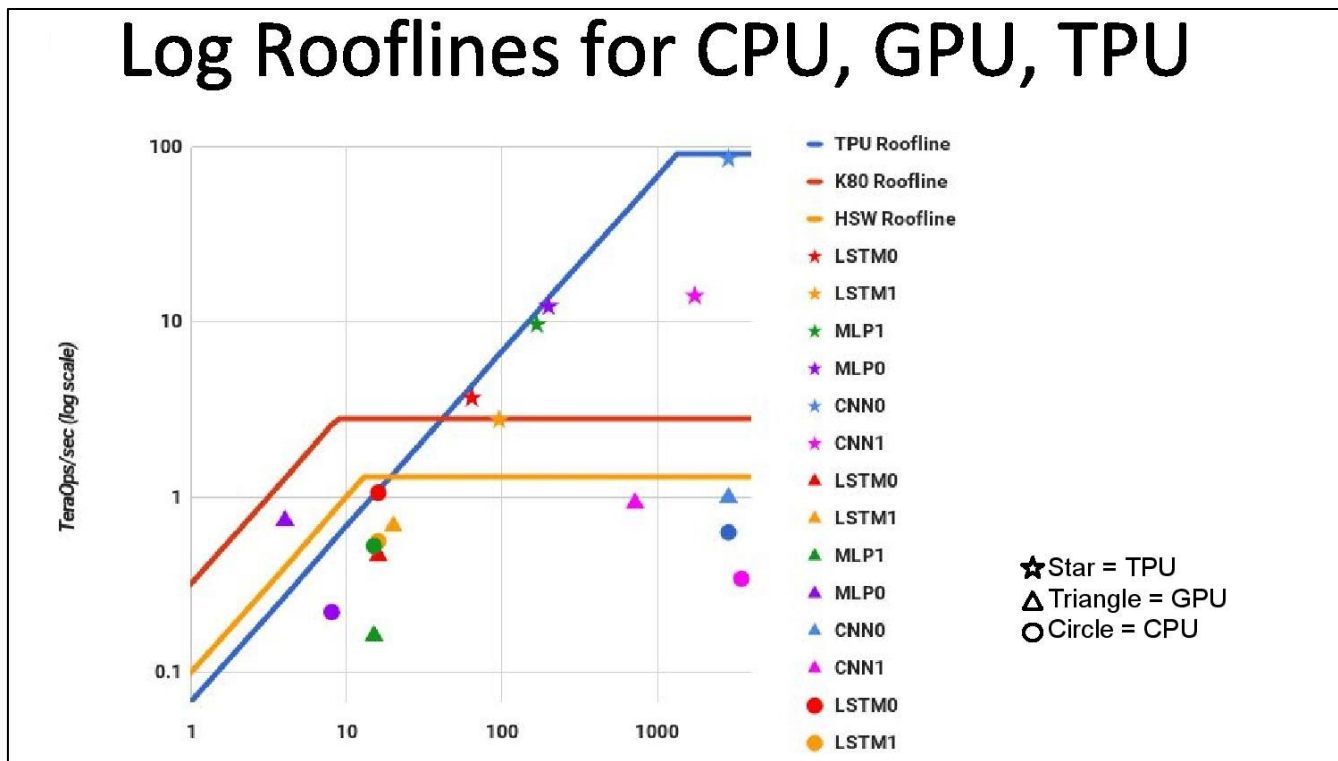
- $\pi$  - the peak compute performance
- $\beta$  - the peak bandwidth
- $I$  - the arithmetic intensity
- The attainable throughput  $P$ :

$$P = \min \left\{ \begin{array}{l} \pi \\ \beta \times I \end{array} \right.$$

- Performance is upper bounded by the peak performance, the communication bandwidth, and the operational intensity
- Arithmetic Intensity is the ratio of the compute to the memory traffic

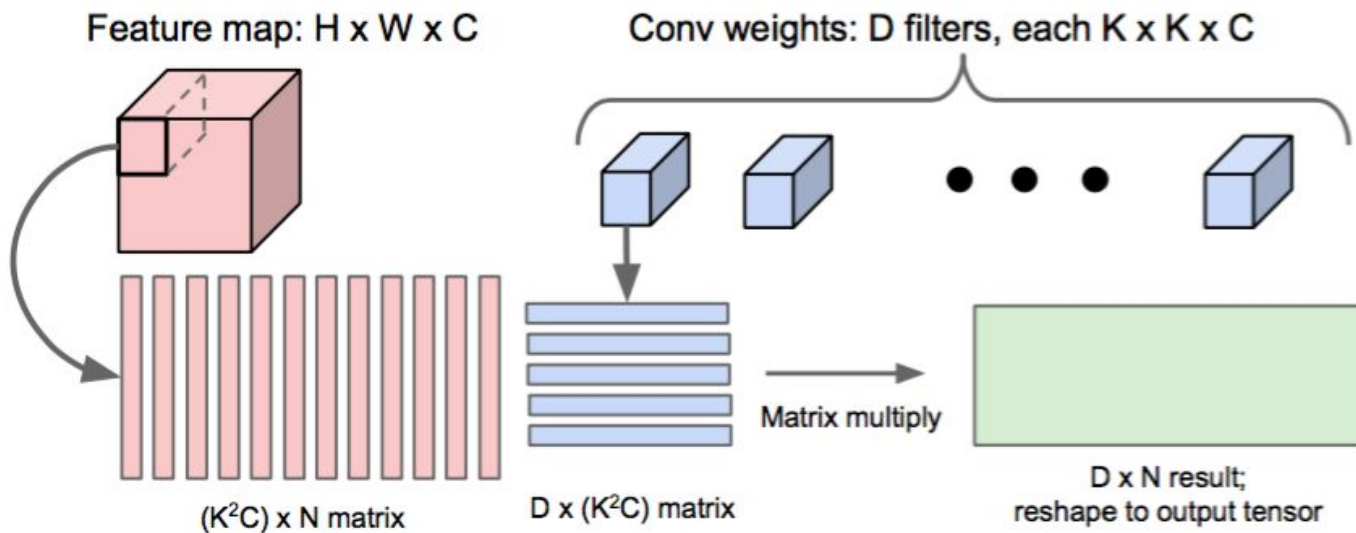


# The Roofline Model

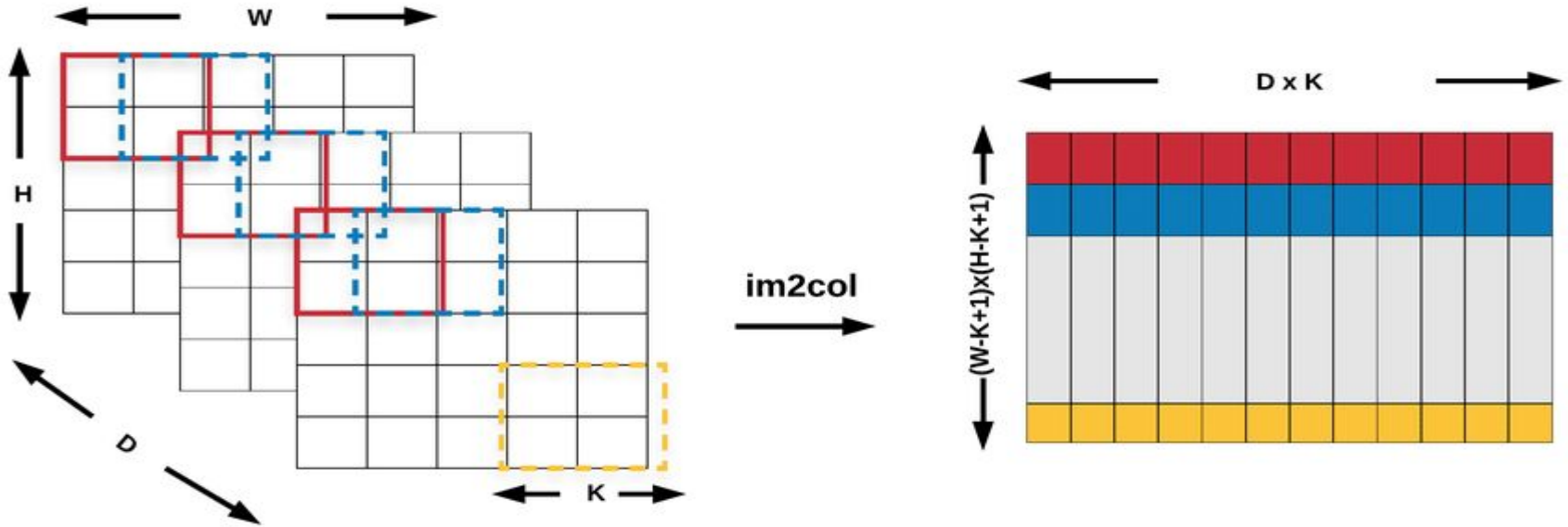


# Conv2D to Matrix-Matrix Multiplication

- **Im2Col** stores in each column the necessary pixels for each kernel map
  - Duplicates input feature maps in memory
  - Restores output feature map structure

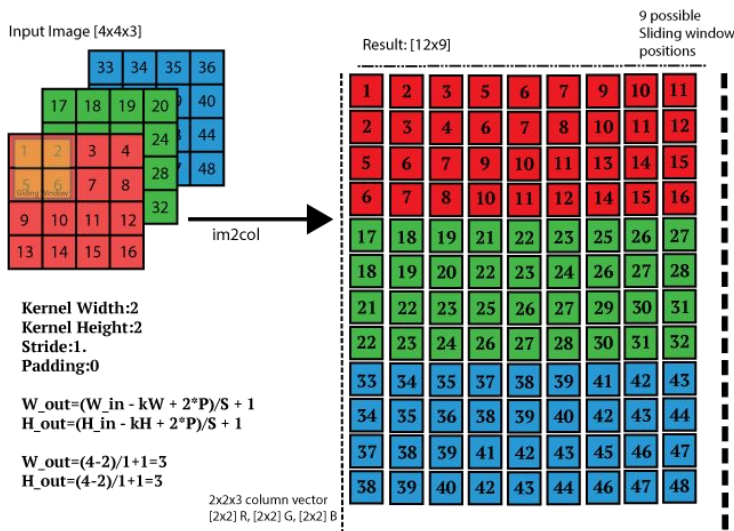


# Im2col Transform

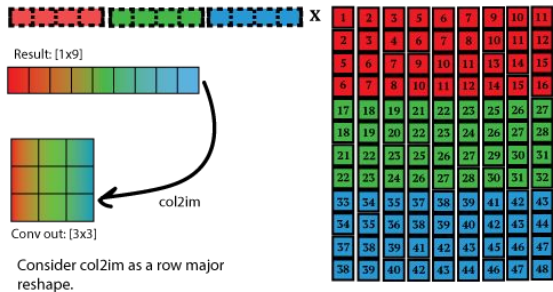


## Image to column operation (im2col)

Slide the input image like a convolution but each patch become a column vector.



We can multiply this result matrix [12x9] with a kernel [1x12].  
result = kernel x matrix  
The result would be a row vector [1x9].  
We need another operation that will convert this row vector into a image [3x3].

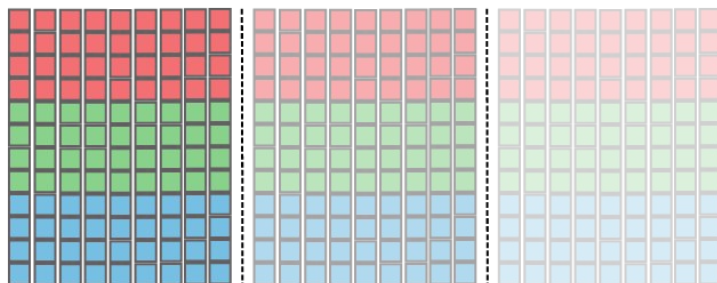
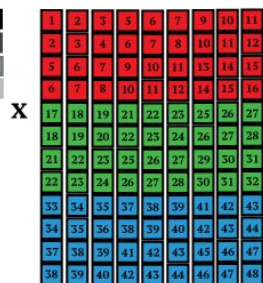


We get true performance gain when the kernel has a large number of filters, ie: F=4 and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2]. The only problem with this approach is the amount of memory

Reshaped kernel: [4x12]

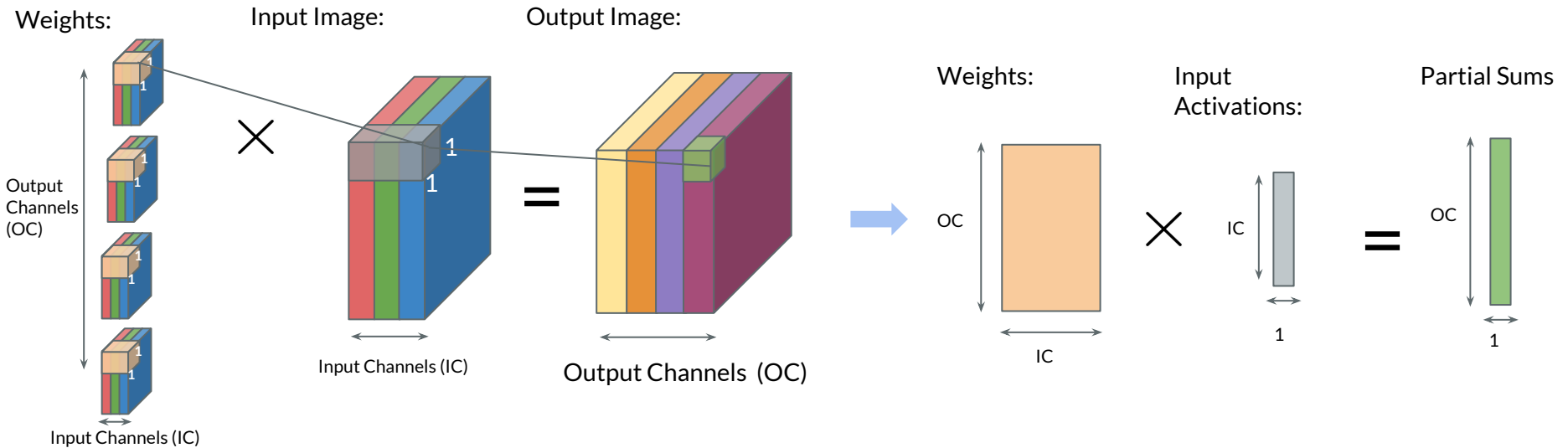


Converted input batch [12x36]



# Conv2D to Matrix-Vector Multiplication

- For each pixel, we can first perform Matrix-Vector Multiplication along the input channel dimension
- Then we can use adder-tree to aggregate the sum of  $K \times K$  pixels ( $K$  is the kernel size)



# Accelerator Architecture

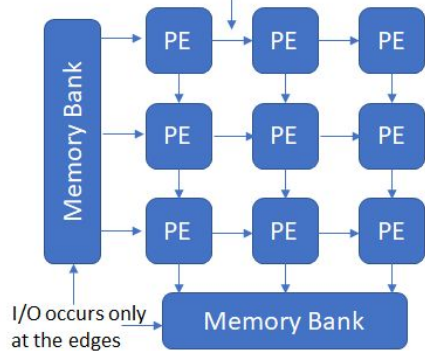
# General Architecture

# Systolic Array

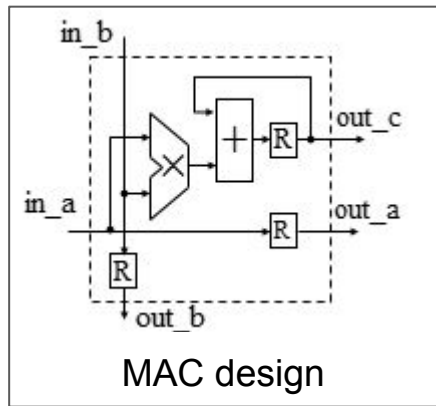
- **Systolic Array** is a homogeneous network of tightly coupled data processing units (DPUs).
- Each **DPU** independently computes a partial result as a function of the data received from its upstream neighbors, stores the result within itself and passes it downstream.
- Advantages of systolic array design:
  - Shorter wires -> lower propagation delay and lower power consumption
  - High degree of pipelining -> faster clock
  - High degree of parallelism -> high throughput
  - Simple control logic -> less design efforts



Short, very fast wires, no R/W (save time, energy)

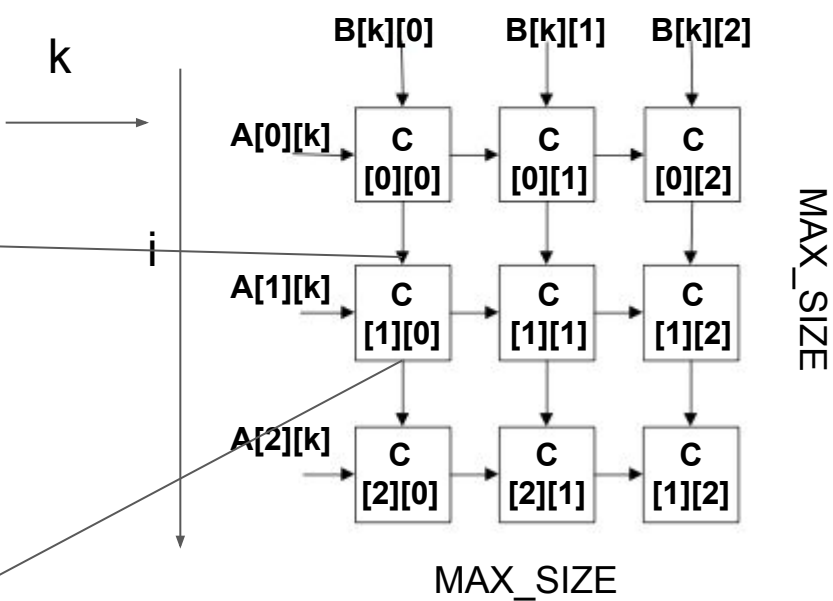


System Architecture



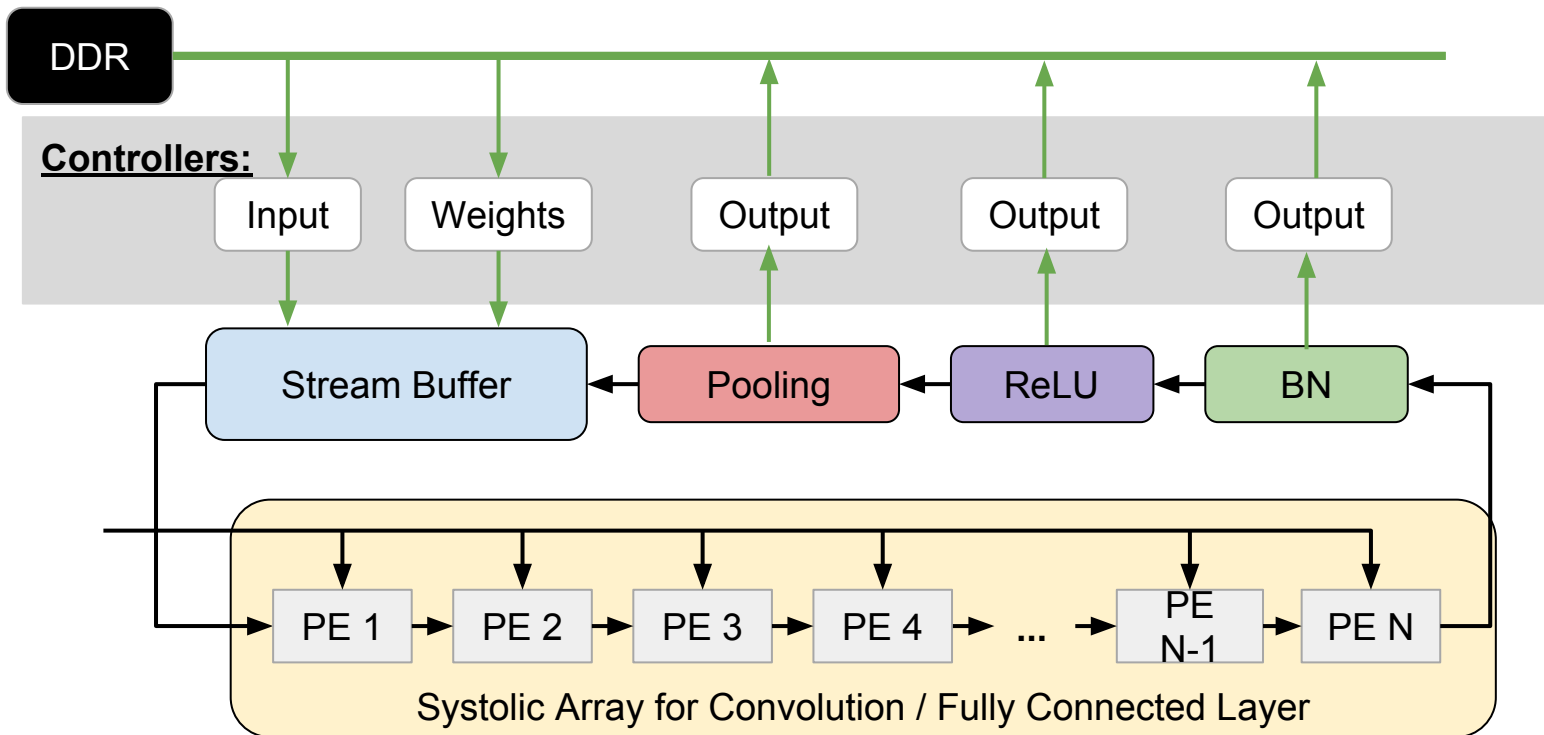
MAC design

$$C[i][j] = C[i][j] + A[i][k] * B[k][j]$$

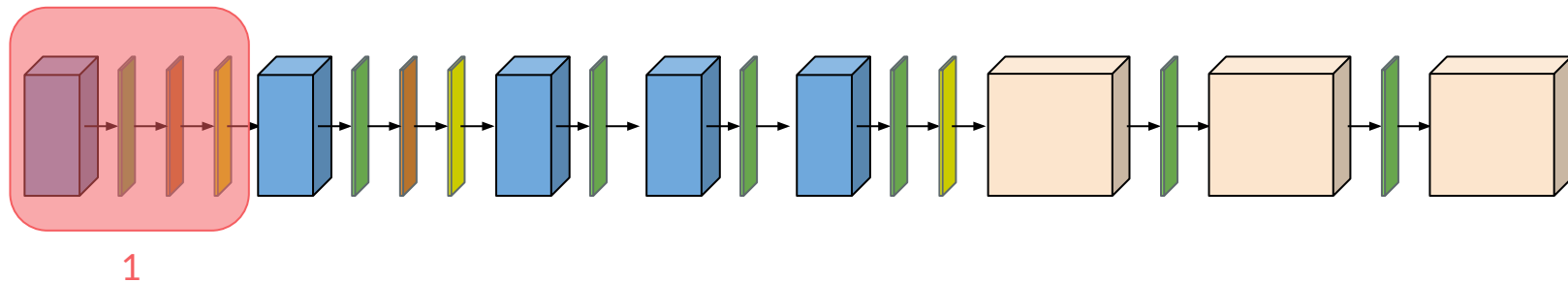
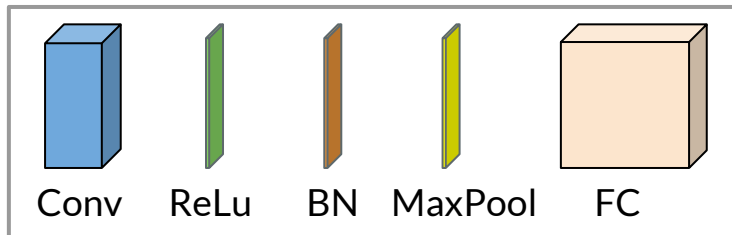


# Specialized Architecture

# Layer-based Design

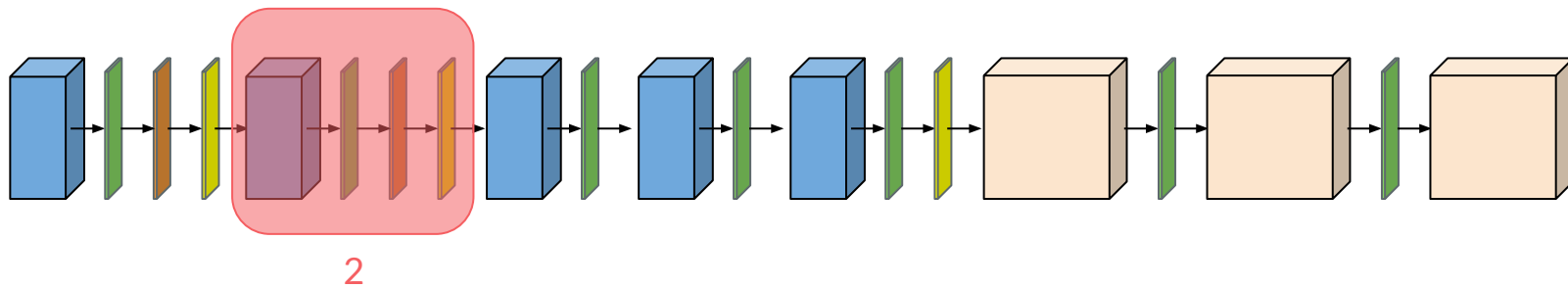
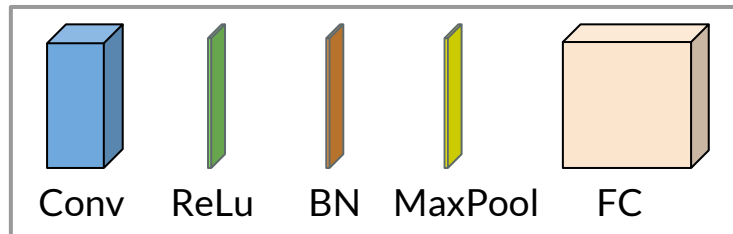


# Execution Model



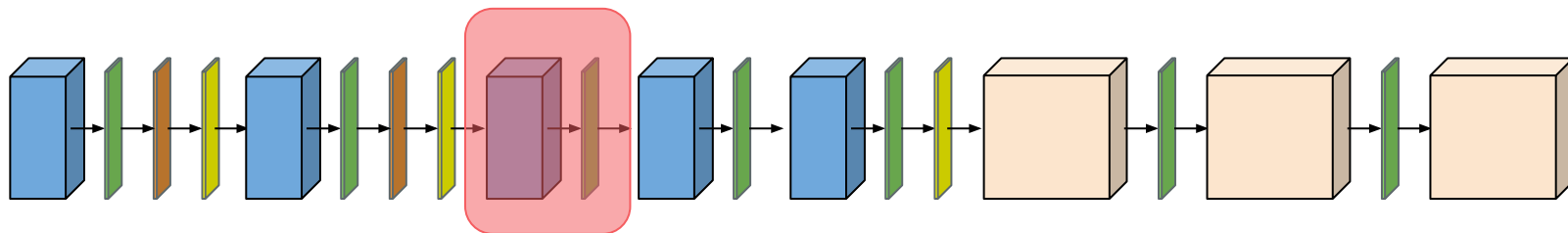
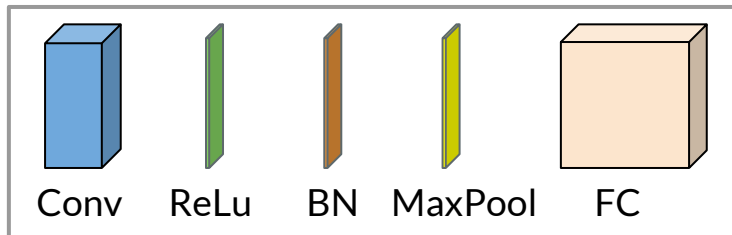
AlexNet Design

# Execution Model



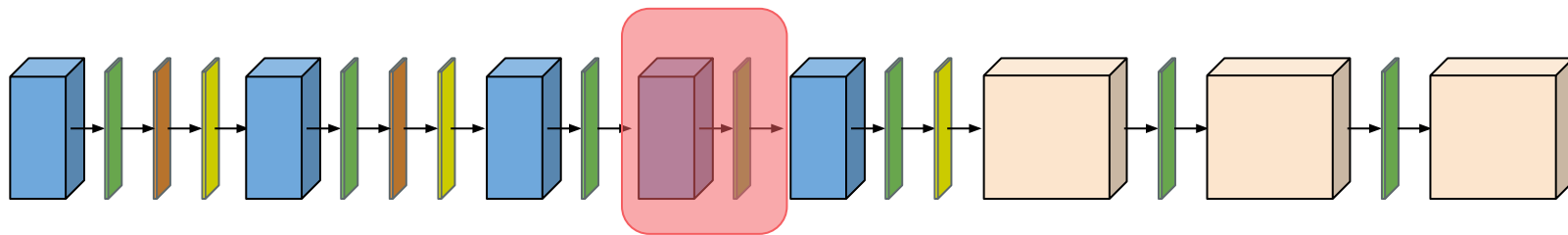
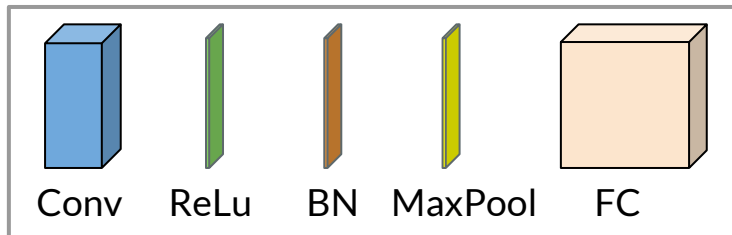
AlexNet Design

# Execution Model



3  
AlexNet Design

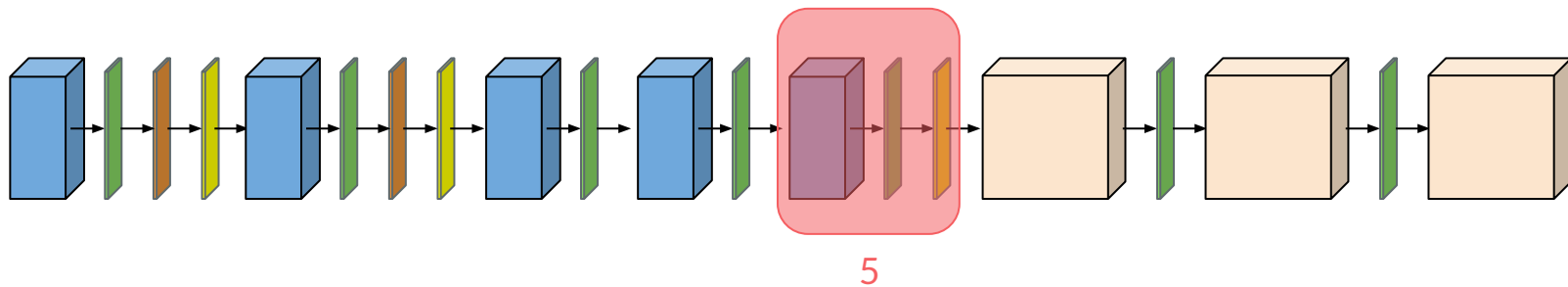
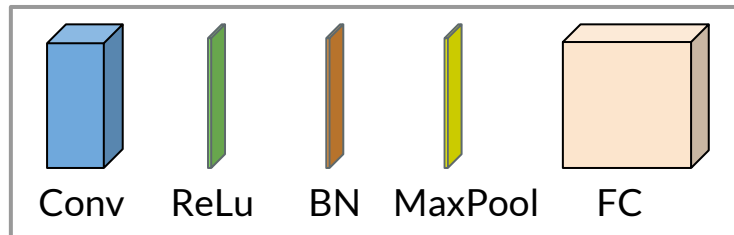
# Execution Model



4

AlexNet Design

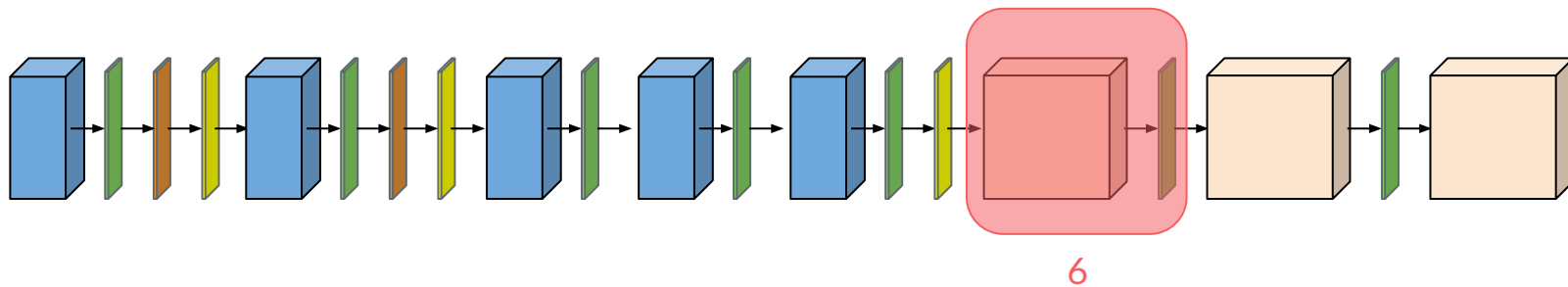
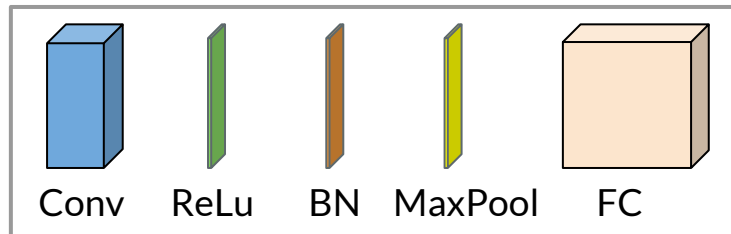
# Execution Model



AlexNet Design

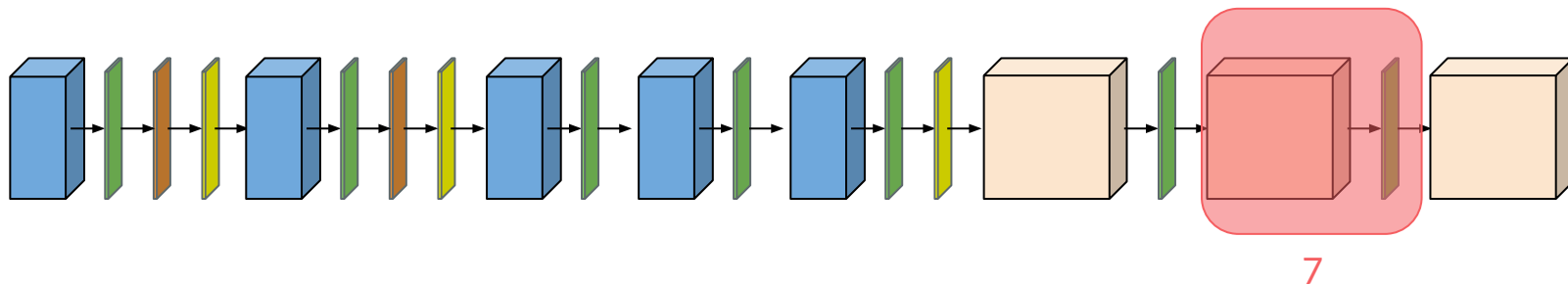
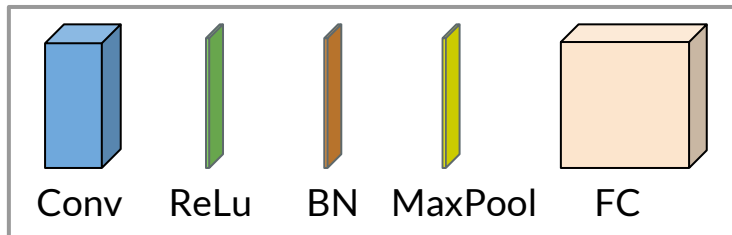


# Execution Model



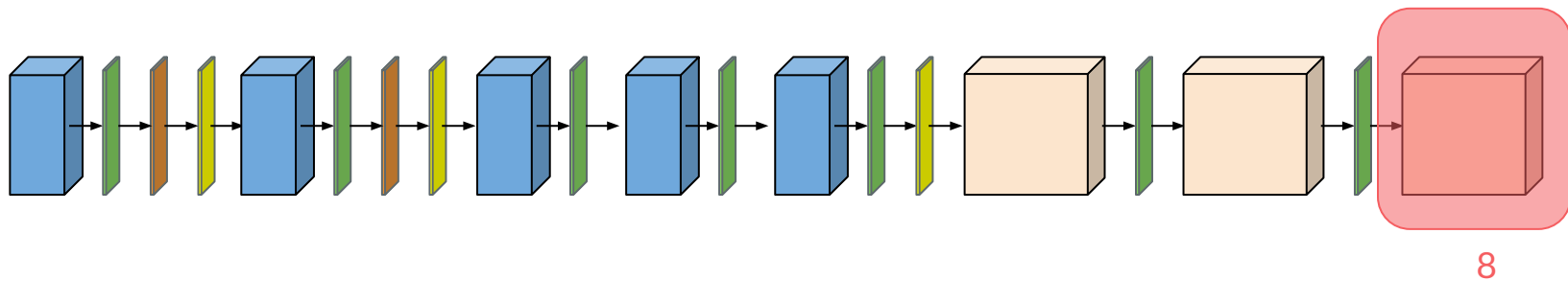
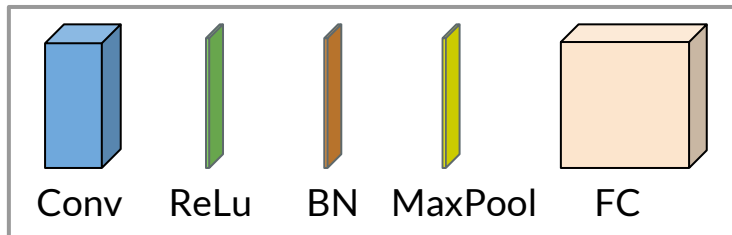
AlexNet Design

# Execution Model



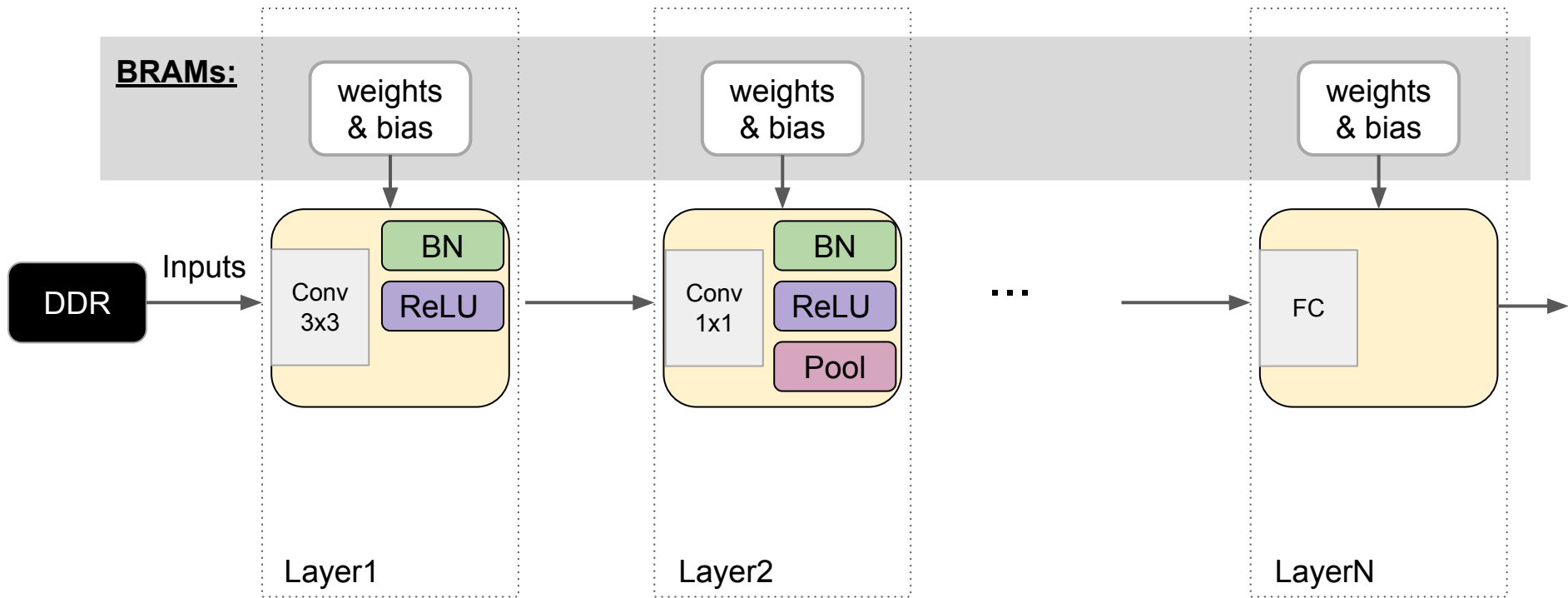
AlexNet Design

# Execution Model

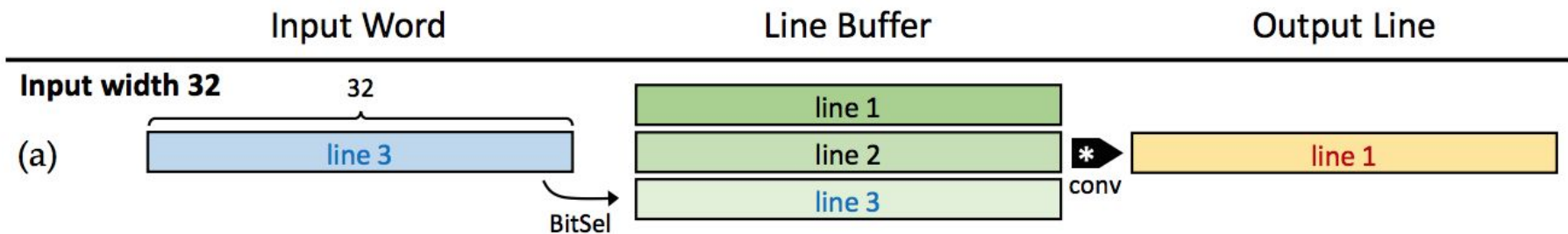


AlexNet Design

# Spatial Design



# Line-Buffer Design



- Buffers inputs to perform spatial operations
- Buffers inputs for reuse to improve the arithmetic intensity

# Execution Model

- 2x2 Max Pooling

4	2	5	6	9
1	3	8	7	3
6	4	2	8	1

# Execution Model

- 2x2 Max Pooling

4	2	5	6	9
1	3	8	7	3
6	4	2	8	1
4				

# Execution Model

- 2x2 Max Pooling

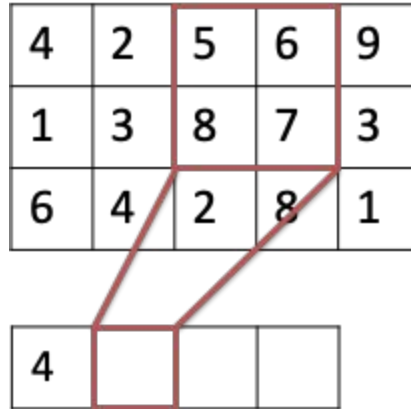
4	2	5	6	9
1	3	8	7	3
6	4	2	8	1

4			
---	--	--	--



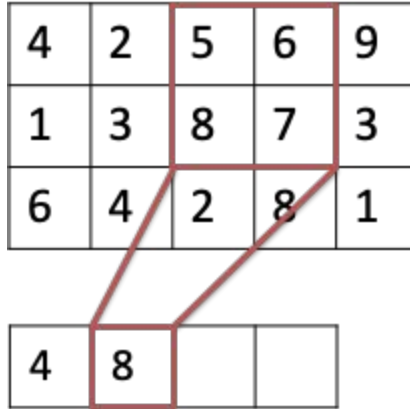
# Execution Model

- 2x2 Max Pooling



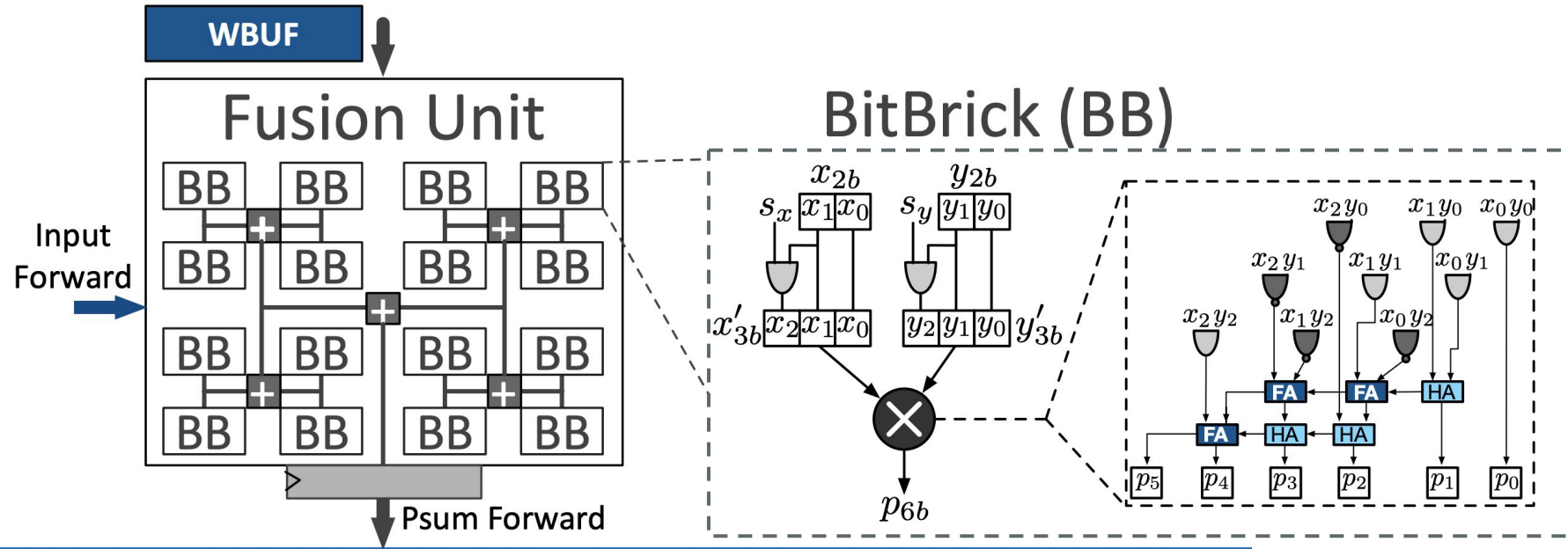
# Execution Model

- 2x2 Max Pooling

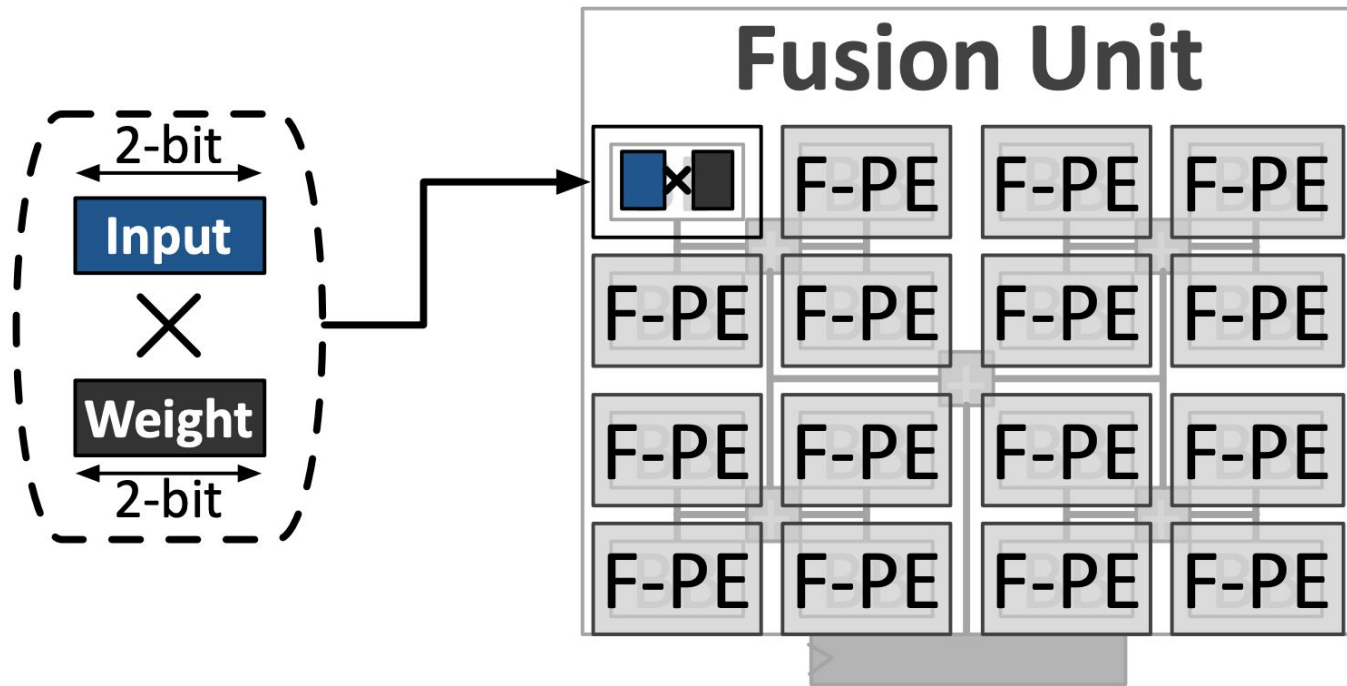


# Mixed-precision Processing Elements

# Mixed-Precision Processing Element (PE)

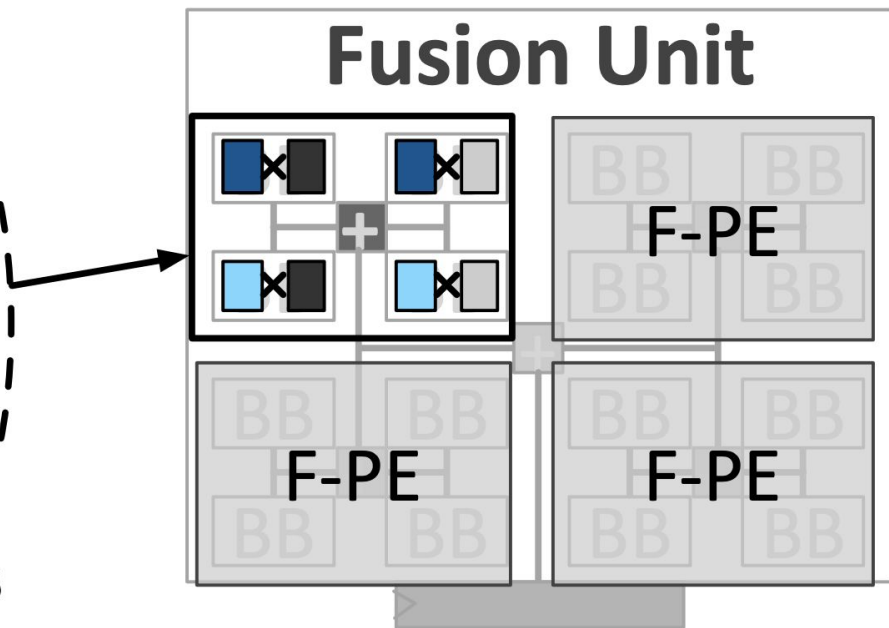
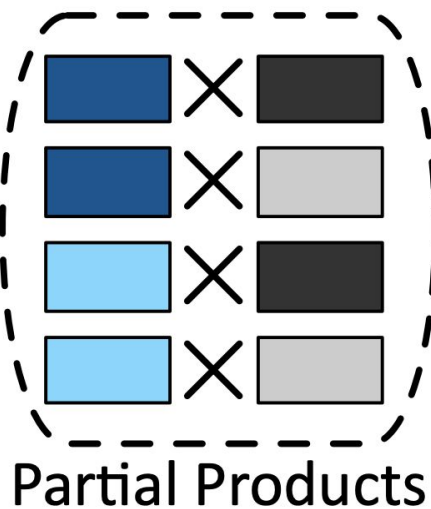
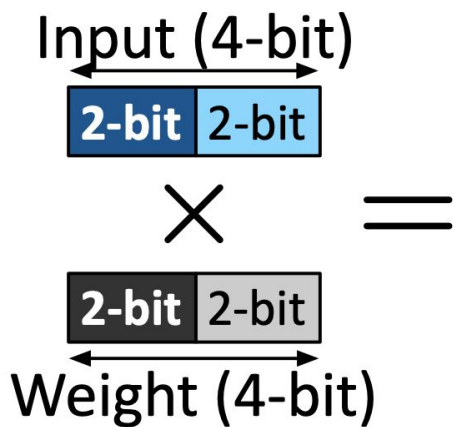


# Spatial PE: 2-bit mode



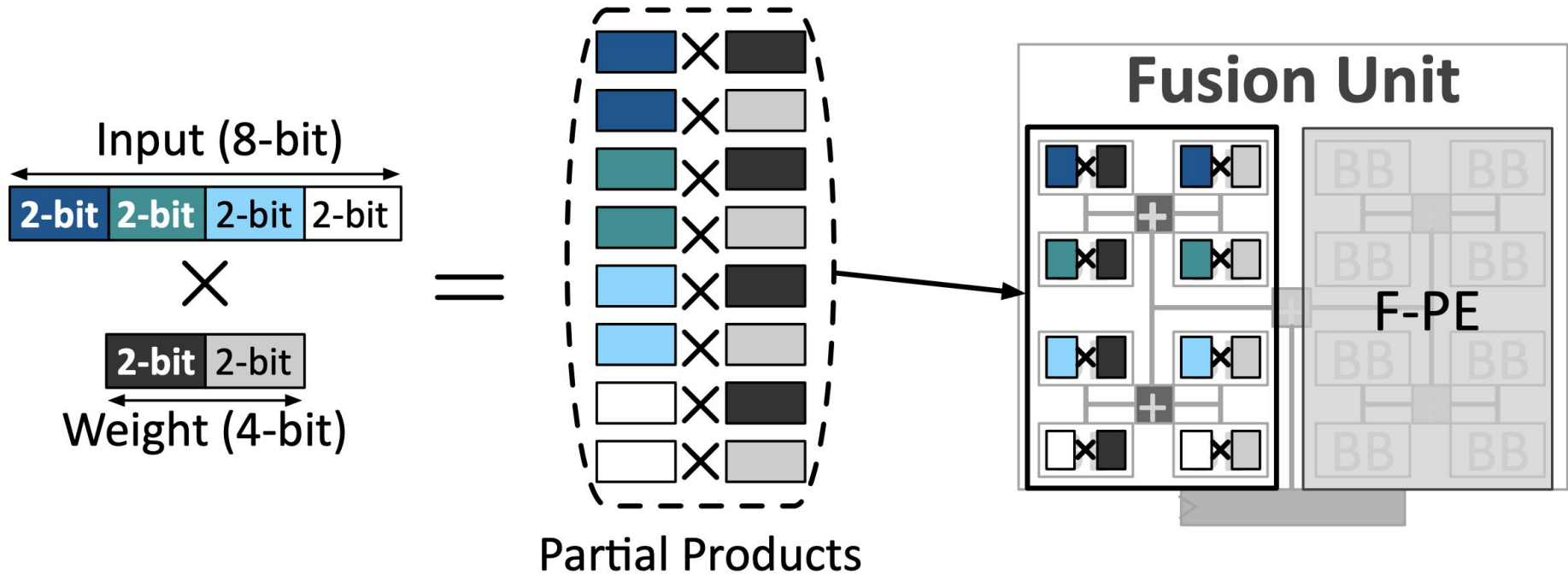
**16X Parallelism**

# Spatial PE: 4-bit mode



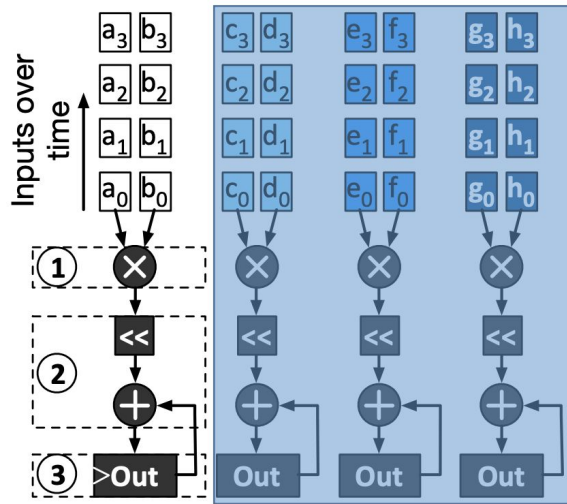
**4x Parallelism**

# Spatial Processing Element: 4-8 bit mode

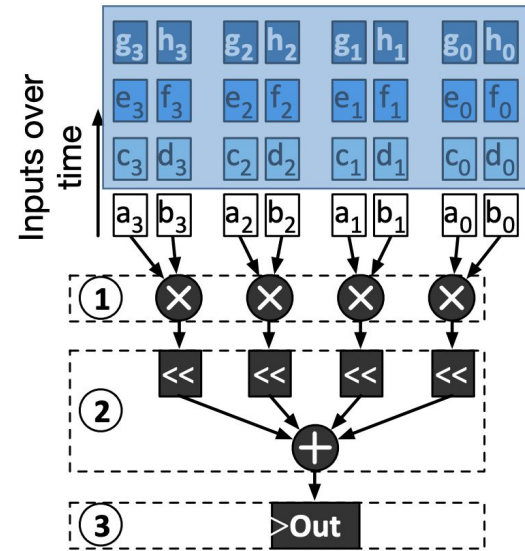


**2x Parallelism**

# Mixed-precision PE: Temporal vs. Spatial



**Bit-Serial:** Combines results over time



**Bit-Parallel:** Combines results over space

- Spatial design is normally more efficient in terms of area and power, given the same throughput



# High-level Synthesis

# High-Level Synthesis

- Allows users to specify algorithm logic in high-level languages
  - No concept of clock
  - Not specifying register-transfer level activities
- HLS compiler generates RTL based on high-level algorithmic description
  - Allocation
  - Scheduling
  - Binding
- Advantages:
  - Faster development and debugging cycles
  - More structural code
  - Focuses on larger architecture design tradeoffs

# HLS Abstraction

- High-level Languages
  - C/C++, OpenCL, GoLang
- Typical hardware mapping
  - C Function -> Verilog Module
  - Function Arguments -> Memory Ports
  - Basic Blocks (blocks without branches) -> Hardware Logic
  - Operators -> Functional Units
  - Arrays -> BRAMs
  - Control Flow Graph (CFG) -> Finite-state Machine (FSM)
- Limitations:
  - No dynamic memory allocation allowed
  - No recursion support

# Example: Matrix Multiplication

## Step 1: Partition Local Arrays

```
// Local memory to store input and output matrices
int localA[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localA dim=1 complete

int localB[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localB dim=2 complete

int localC[MAX_SIZE][MAX_SIZE];
#pragma HLS ARRAY_PARTITION variable=localC dim=0 complete
```

## Step 2: Design Systolic Array (Implicit)

```
systolic1: for(int k = 0; k < a_col; k++) {
#pragma HLS LOOP_TRIPCOUNT min=c_size max=c_size
#pragma HLS PIPELINE II=1
    systolic2: for(int i = 0; i < MAX_SIZE; i++) {
        systolic3: for(int j = 0; j < MAX_SIZE; j++) {

            // Get previous sum
            int last = (k==0) ? 0 : localC[i][j];

            // Update current sum
            // Handle boundary conditions
            int a_val = (i < a_row && k < a_col)? localA[i][k] : 0;
            int b_val = (k < b_row && j < b_col)? localB[k][j] : 0;
            int result = last + a_val*b_val;

            // Write back results
            localC[i][j] = result;

        }
    }
}
```

## Step 2: Design Systolic Array (Explicit)

```
        for (int r = 0; r < N + 2 * MAX_SIZE - 2; r++) {
#pragma HLS pipeline
        // update data (i.e., reads data from previous PE)
        for (int i = 0; i < MAX_SIZE; i++)
            for (int j = MAX_SIZE - 1; j >= 1; j--)
                localA[i][j] = localA[i][j - 1];

        for (int i = MAX_SIZE - 1; i >= 1; i--)
            for (int j = 0; j < MAX_SIZE; j++)
                localB[i][j] = localB[i - 1][j];

        // read new data from inputs
        // not ok here!
        for (int i = 0; i < MAX_SIZE; i++) {
            if (r >= i && r < i + N)
                localA[i][0] = A[i + ii * MAX_SIZE][r - i];
            else
                localA[i][0] = 0;
        }

        for (int j = 0; j < MAX_SIZE; j++) {
            if (r >= j && r < j + N)
                localB[0][j] = B[r - j][j + jj * MAX_SIZE];
            else
                localB[0][j] = 0;
        }

        // PE
        for (int i = 0; i < MAX_SIZE; i++)
            for (int j = 0; j < MAX_SIZE; j++)
                C[i + ii * MAX_SIZE][j + jj * MAX_SIZE] += localA[i][j] * localB[i][j];
    }
}
```

# Step 3: Schedule Outer Loop Control Logic and Memory Accesses

```
// Burst reads on input matrices from global memory
// Read Input A
readA: for(int loc = 0, i = 0, j = 0; loc < a_row*a_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min=c_size*c_size max=c_size*c_size
#pragma HLS PIPELINE II=1
    if(j == a_col) { i++; j = 0;}
    localA[i][j] = a[loc];
}

// Read Input B
readB: for(int loc = 0, i = 0, j = 0; loc < b_row*b_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min=c_size*c_size max=c_size*c_size
#pragma HLS PIPELINE II=1
    if(j == b_col) { i++; j = 0; }
    localB[i][j] = b[loc];
}

// Burst write from output matrices to global memory
// Burst write from matrix C
writeC: for(int loc = 0, i = 0, j = 0; loc < c_row*c_col; loc++, j++) {
#pragma HLS LOOP_TRIPCOUNT min=c_size*c_size max=c_size*c_size
#pragma HLS PIPELINE II=1
    if(j == c_col) { i++; j = 0; }
    c[loc] = localC[i][j];
}
```

\* Please see the [SDAccel page](#) for detailed source code

# Resources

- [Vivado HLS Design Hubs](#)
- [Parallel Programming for FPGAs](#)
- [Cornell ECE 5775: High-Level Digital Design Automation](#)
- [LegUp: Open-source HLS Compiler](#)
- [VTA design example](#)
- [Vivado SDAccel design examples](#)



Questions?