# Designing DNN Accelerators

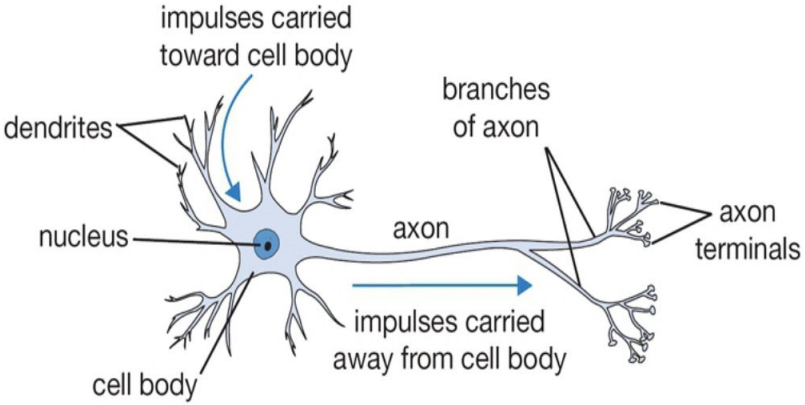
Qijing Jenny Huang

# Outline

1. Deep Neural Network (DNN) Basics
2. DNN Accelerators
3. High-level Synthesis (HLS)

# DNN Basics

# Learning from the Brain

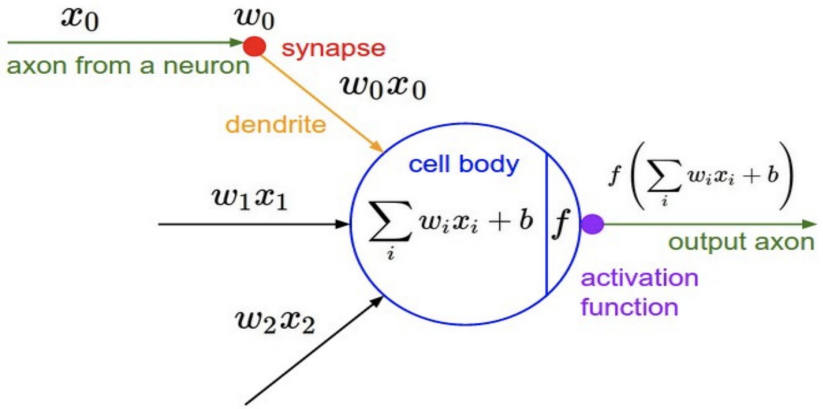



Integrate and Fire

- 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 the  influence strength

* Slide from http://cs231n.github.io/

# Neural Networks

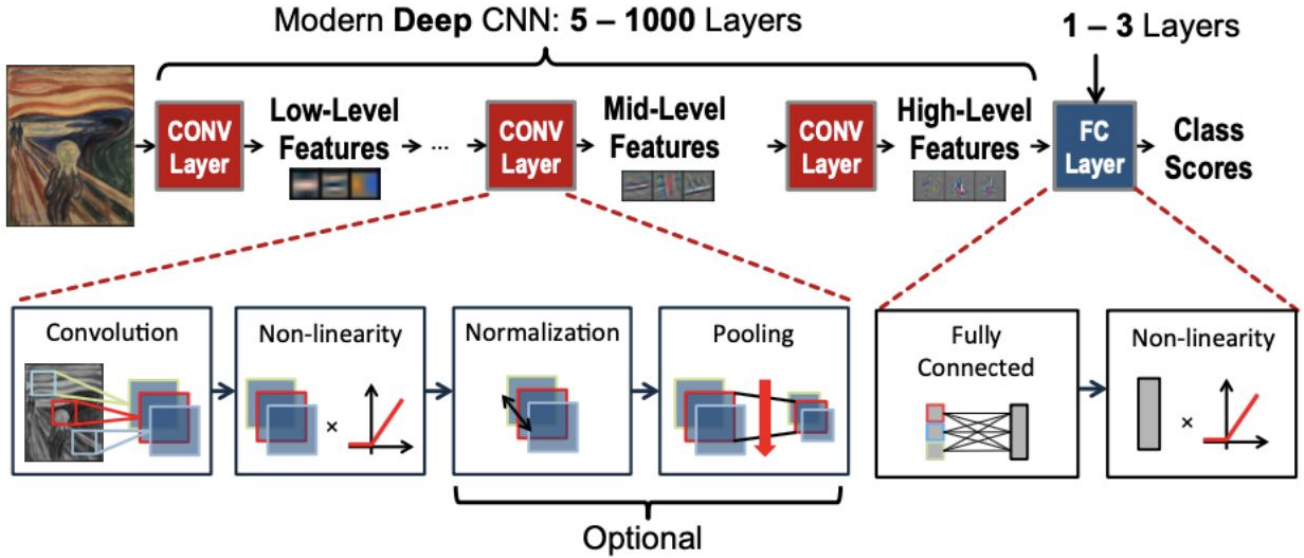



- NNs are usually feed forward computational graphs constructed from many computational “Neurons”
- The “Neurons”:
  - **Integrate** - typically linear transform (dot-product of receptive field)
  - **Fire** - followed by a non-linear “activation” function

* Slide from http://cs231n.github.io/

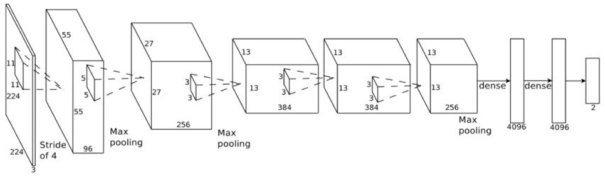
# Deep Neural Networks (DNN)

- An Neural Network with multiple layers between the inputs and outputs

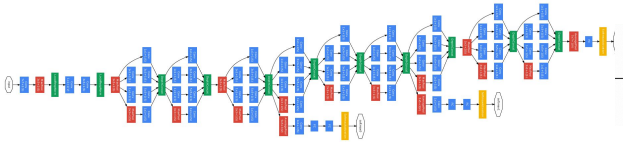



* Image from Eyeriss Tutorial: http://eyeriss.mit.edu/tutorial.html

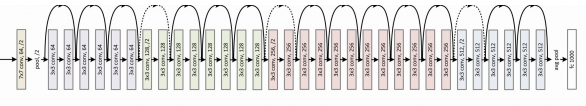
# DNN Examples



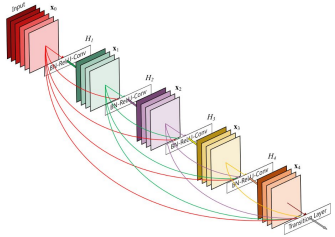

AlexNet 2012 (8 layers)



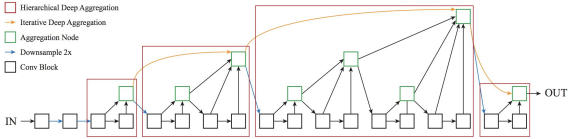
GoogLeNet 2014 (22 layers)



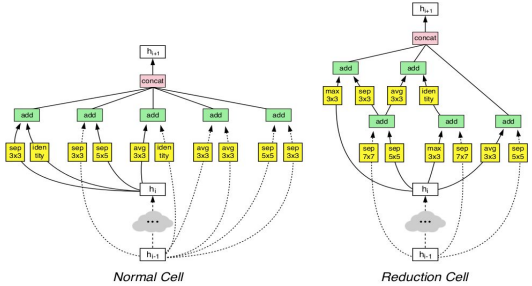
ResNet 2015 (152 layers)



DenseNet 2016 (dense connections)

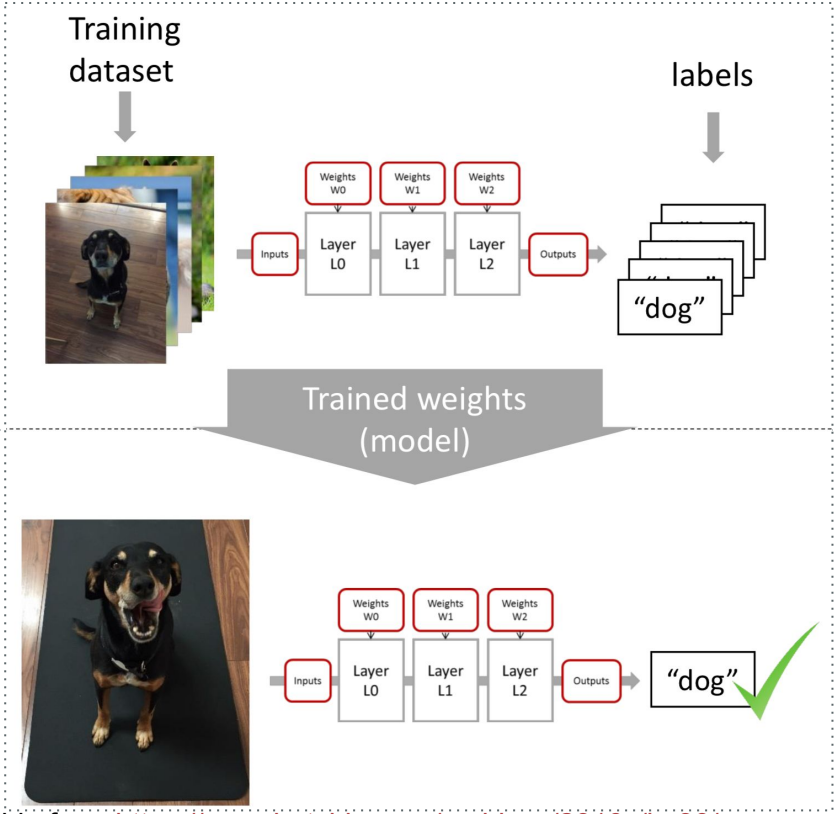



DLA 2017 (deep aggregation)




NasNet 2017 (NAS design)

# Training vs. Inference




## Training (supervised)

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

## Inference

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

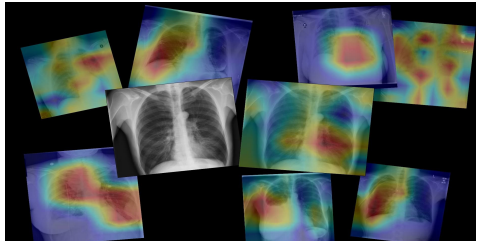
* Slide from https://www.hotchips.org/archives/2010s/hc30/

# DNN Applications

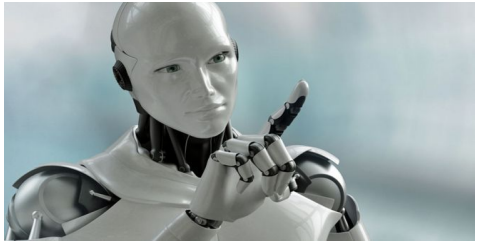
Autonomous Vehicles

Security Camera

Drones



Medical Imaging



Robots

Mobile Applications

# Computer Vision (CV) Tasks

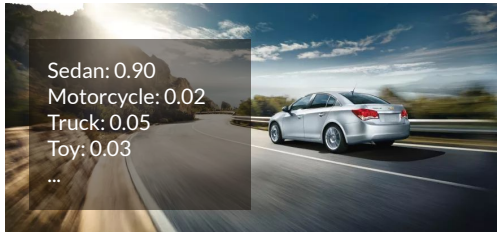



Image Classification

Object Detection

Semantic Segmentation

Super Resolution


Activity Recognition

# Nature Language Processing (NLP) Tasks

Text Classification

Information Extraction

Conversational Agent

Information Retrieval

Question Answering Systems

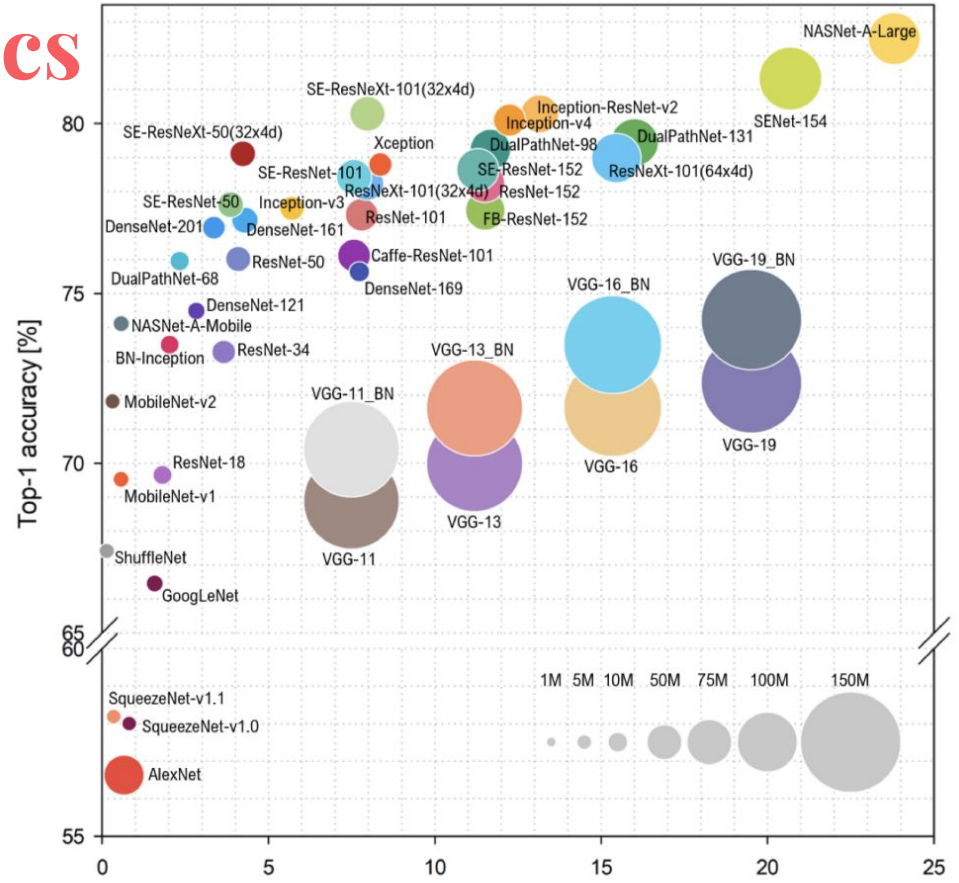
* Image from “Practical Natural Language Processing”: https://github.com/practical-nlp/practical-nlp

# Many Other Tasks

- Recommendation Systems (DLRM)
- Machine Translation (Transformer and  GNMT)
- Deep Reinforce Learning (AlphaGo)

# DNN Evaluation Metrics

1. Accuracy
2. Computation Complexity
3. Model Size




* Image from “MLPerf Inference Benchmark”: https://arxiv.org/abs/1911.02549

# DNN Accelerators

# Many AI Chips

> 112 AI chip companies worldwide (https://github.com/basicmi/AI-Chip)

## In the Cloud (Training + Inference)

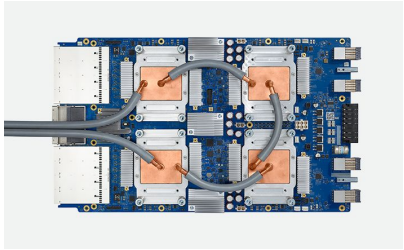
- 10s TFLOPs
- 10s MB on-chip memory
- 8 - 32 bit precision
- 700 MHz - 1 GHz
- 10-100s Watts

## At the Edge (Inference)

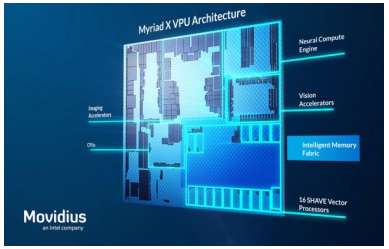
- 100s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 50 MHz - 400 MHz
- 1-10s Watts

## In the Edge SoC/SiP (Inference)

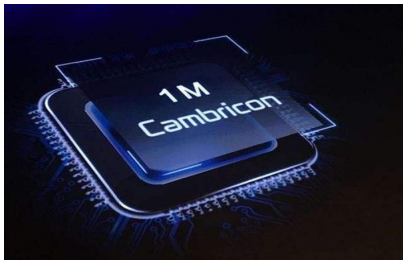
- 10s-1000s GFLOPs
- 100s KB on-chip memory
- 1 - 16 bit precision
- 600 MHz - 1 GHz
- 10-100s mWatts



Cloud TPU v3 (45 TFLOP/s)

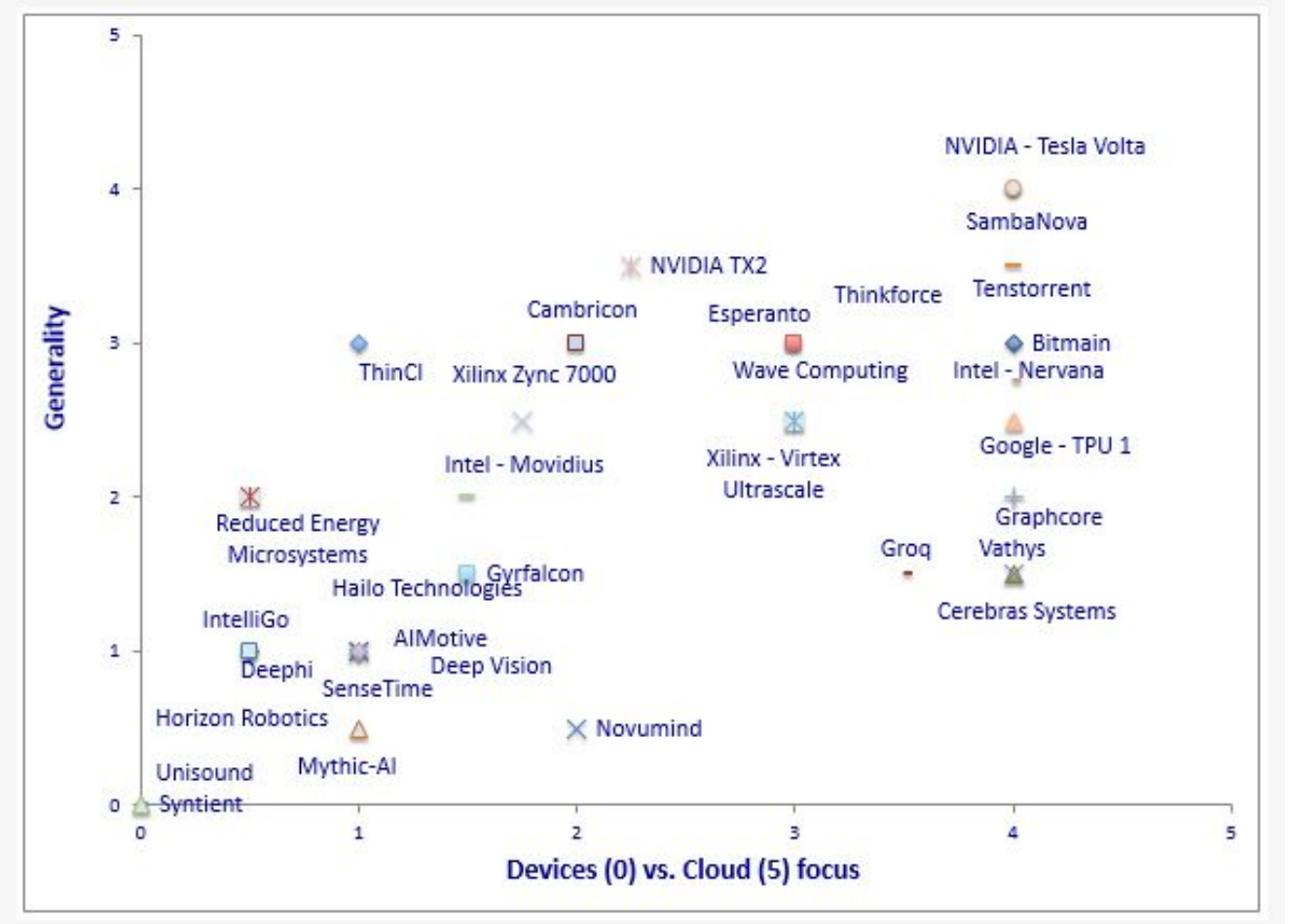



Intel Movidius (4 TFLOP/s)




Cambricon-1M IP

* Data adapted from Prof. Kurt Keutzer's talk at DAC 2018




* Image from https://www.electronicproducts.com/Digital_ICs/Designer_s_Guide_Selecting_AI_chips_for_embedded_designs.aspx

# Accelerator Evaluation Metrics

1. Throughput
   - Frames per second
2. Latency
   - Time to finish one frame
3. Power
4. Energy
5. Hardware Cost
   - Resource Utilization

Benchmarks:


https://mlperf.org/

# Example Hardware Comparison

| | # | Metric | Google TPU v3 | Nvidia V100 | Nvidia A100 | Cerebras WSE | GraphCore IPU1 | GraphCore IPU2 |
|---|---|---|---|---|---|---|---|---|
| Raw Metrics | 1 | Technology node | >12nm (16 nm est.) | TSMC 12 nm | TSMC 7 nm | TSMC 16 nm | TSMC 16 nm | TSMC 7 nm |
| | 2 | Die Area (mm2) | <648 (600 est.) | 815 | 826 | 46225 | 900 (est.) | 823 |
| | 3 | Transistor Count (B) | 11 (est.) | 21 | 54.2 | 1200 | 23.6 | 59.4 |
| | 4 | Architecture | Systolic Array | SIMD | SIMD | SIMD | SIMD | SIMD |
| | 5 | Theoretical TFLOPS (16-bit mixed precision) | 123 | 125 | 312 | 2500 | 125 | 250 |
| | 6 | Freq (GHZ) | 0.92 | 1.5 | 1.4 | Unknown | 1.6 | Unknown |
| | 7 | DRAM Capacity (GB) | 32 | 32 | 80 | N/A | N/A | 112 |
| | 8 | DRAM BW (GB/sec) | 900 | 900 | 2039 | N/A | N/A | 64 (est.) |
| | 9 | Total SRAM Capacity | 32MB | 36 MB (RF+L1+L2) | 87 MB (RF+L1+L2) | 18 GB | 300 MB | 900 MB |
| | 10 | SRAM BW (TB/sec) | Unknown | 224 @RF + 14 @L1 + 3 @L2 | 608 @RF+ 19 @L1 + 7 @L2 | 9000 | 45 | 47.5 |
| | 11 | Max TDP (Watts) | 450 | 450 | 400 | 20K | 150 | 150 (est.) |
| Efficiency Metrics | 12 | GEMM Achievable TFLOPS | 98% (120 TFLOPS) | 88% (110 TFLOPS) | 93% (290 TFLOPS) | Unknown | 47% (58 TFLOPS) | 61% (154 TFLOPS) |
| | 13 | Energy Efficiency (Achievable GEMM TFLOPS/Max Watts) | 0.26 | 0.24 | 0.72 | Unknown | 0.39 | 1.0 |
| | 14 | Theoretical Energy Efficiency (Theoretical TFLOPS/Max Watts) | 0.27 | 0.27 | 0.78 | 0.125 | 0.83 | 1.6 |
| | 15 | Memory Capacity (GB) | 16 | 32 | 80 | 18 | 0.3 | 112 |
| | 16 | Memory Efficiency (FLOP/DRAMByte) | 133 | 122 | 158 | N/A | N/A | Unknown |
| | 17 | Memory Efficiency (FLOP/SRAMByte) | Unknown | 32 | 35 | Unknown | 1.28 | 3.2 |
| | 18 | Area Efficiency (Achievable TFLOPS/mm2) | 0.2 | 0.13 | 0.35 | Unknown | 0.06 | 0.17 |
| | 19 | Area Efficiency (Achievable TFLOPS/BTran) | 11 | 5.2 | 5.3 | Unknown | 2.5 | 2.6 |

* Table from
https://khairy2011.medium.com/tpu-vs-gpu-vs-cerebras-vs-graphcore-a-fair-comparison-between-ml-hardware-3f5a19d89e38

# How to design your own DNN accelerator?

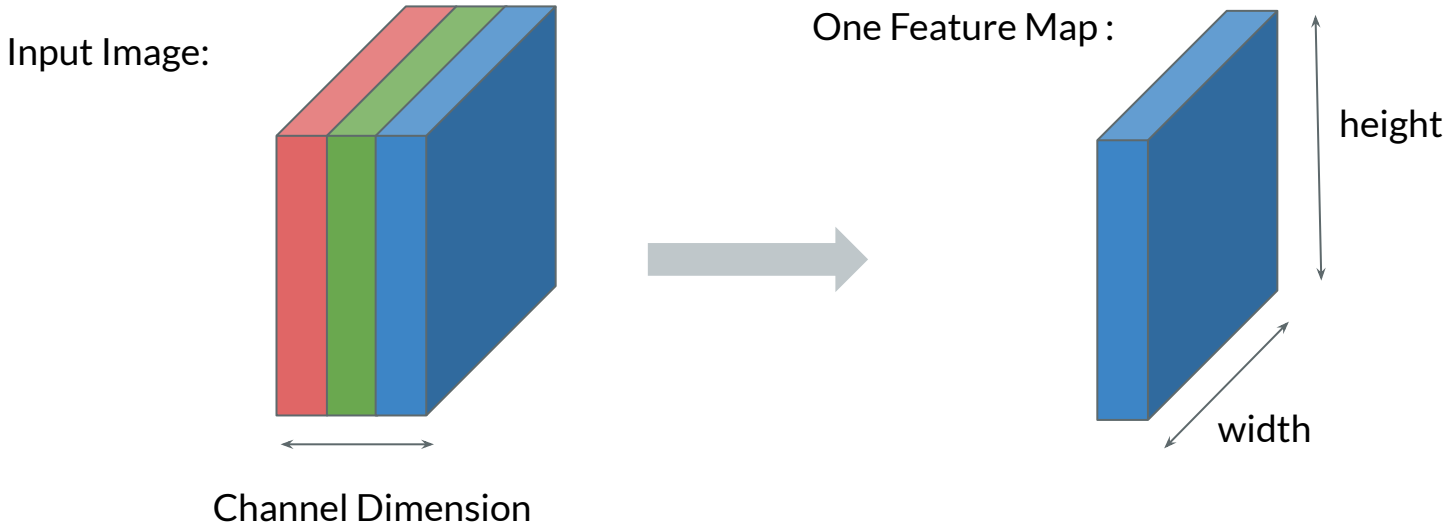
1. Understand the basic operations

# Common DNN Operations

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

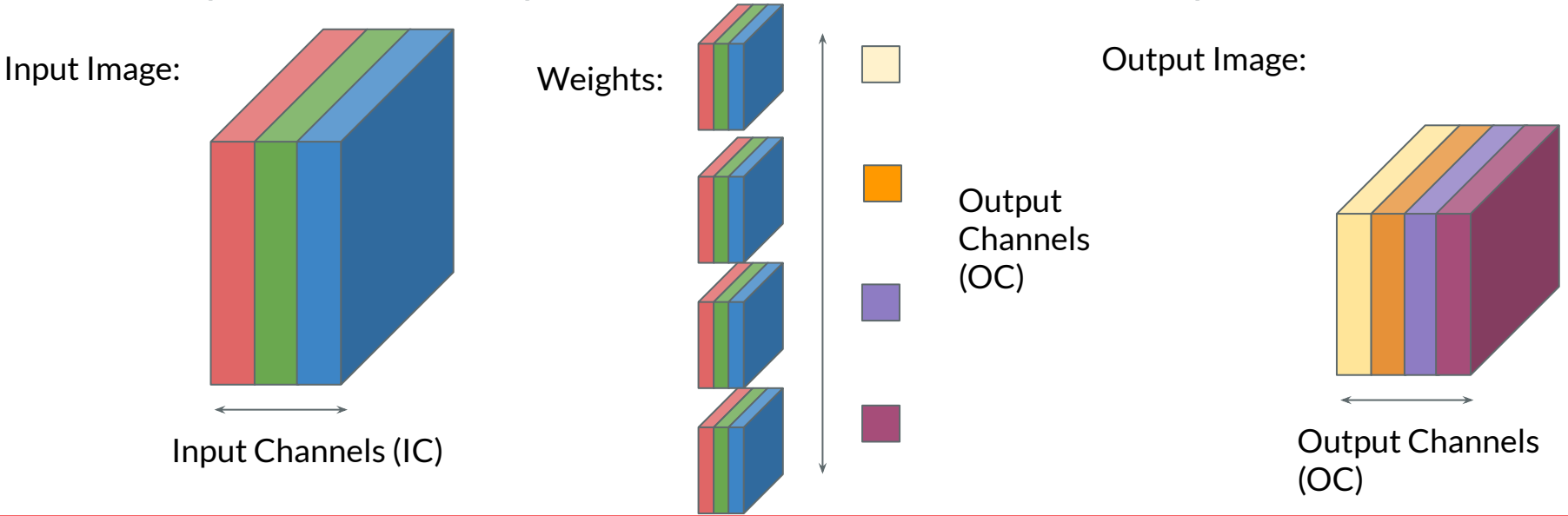
# Activation/Feature Maps

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

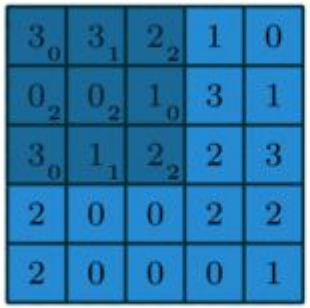


# Weights/Kernels

- 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



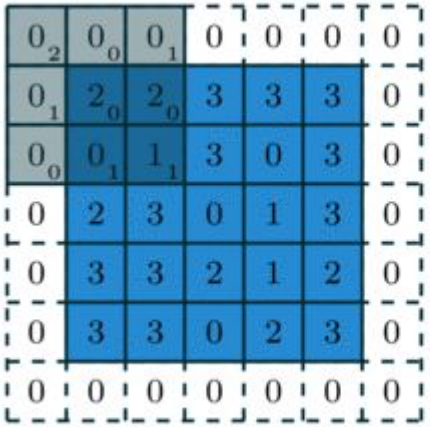
# 3x3 Convolution - Spatially




Input feature map


Output feature map

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




Input feature map

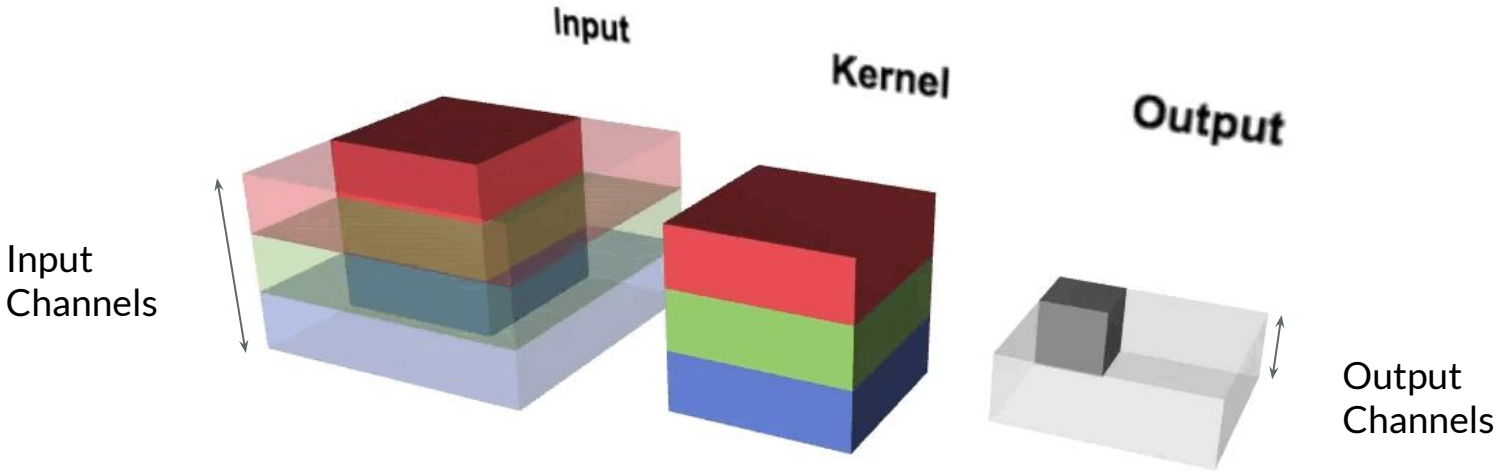

Output feature map

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

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

* gif from http://deeplearning.net/software/theano_versions/dev/_images/

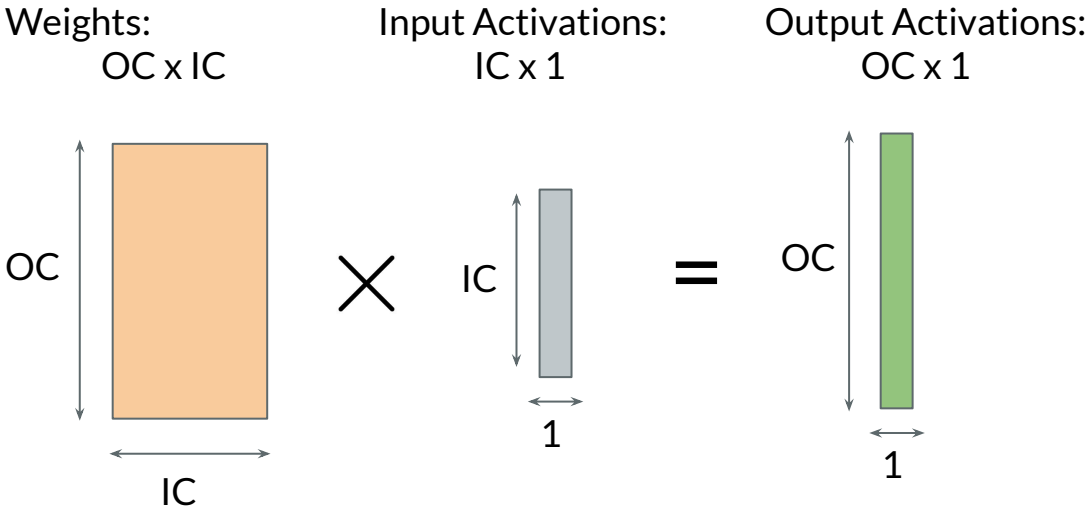
# 3x3 Convolution - 3D

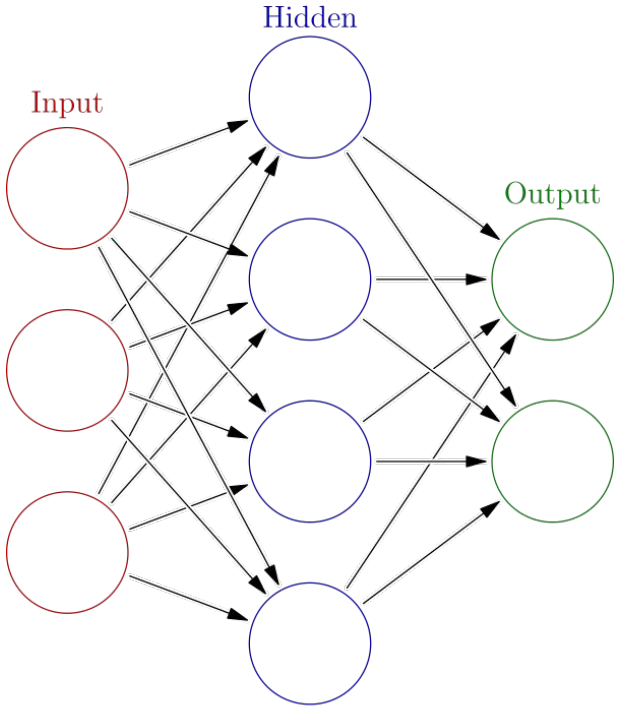



* gif from https://cdn-images-1.medium.com/max/800/1*q95f1mqXAVsj_VMHaOm6Sw.gif

# Fully-Connected Layer (FC)

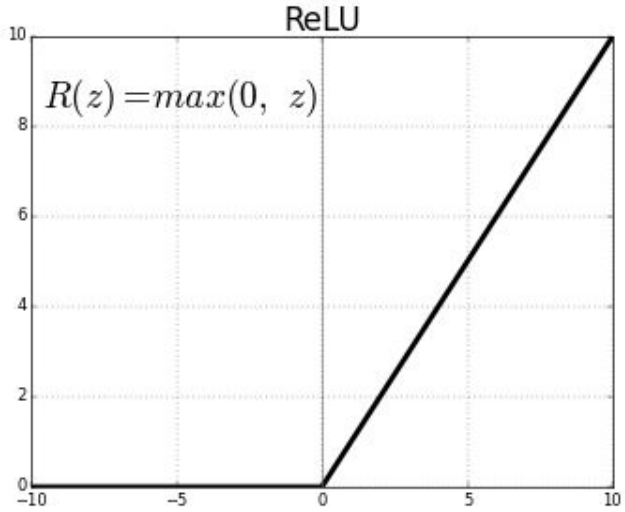
- Each input activation is connected to every output activation
- Essentially a matrix-vector multiplication



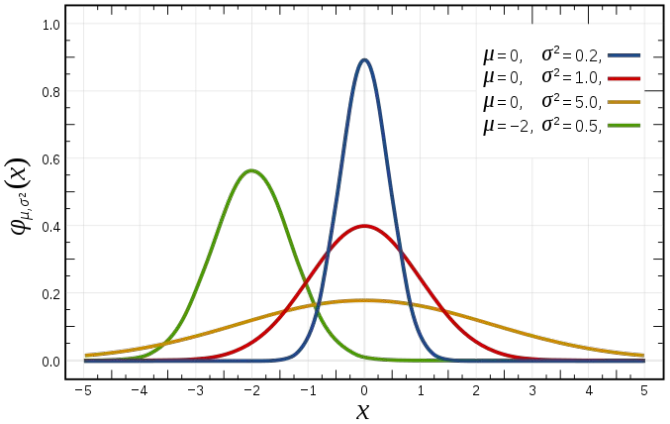
# ReLU Activation Function

- 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: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: $\gamma$, $\beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

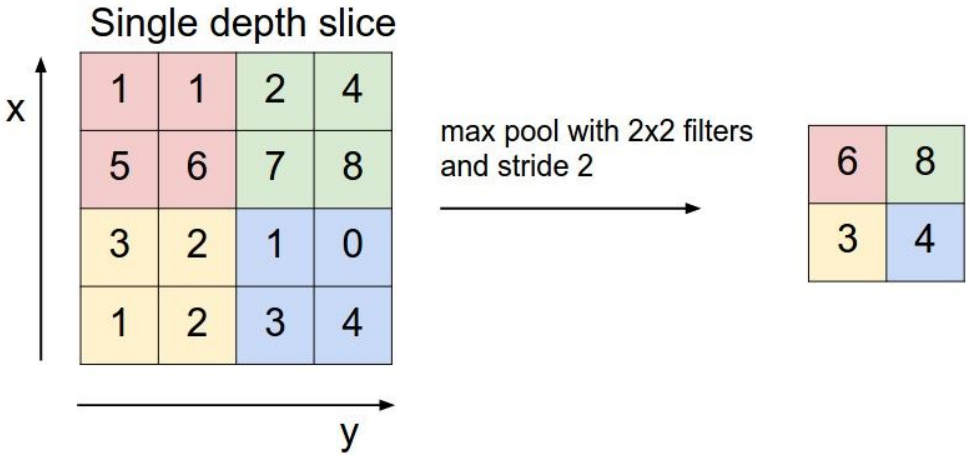
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

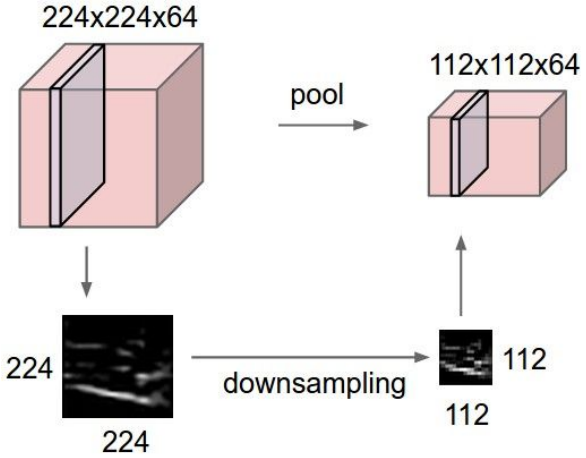
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

* images from https://en.wikipedia.org/wiki/Normal_distribution

# Pooling

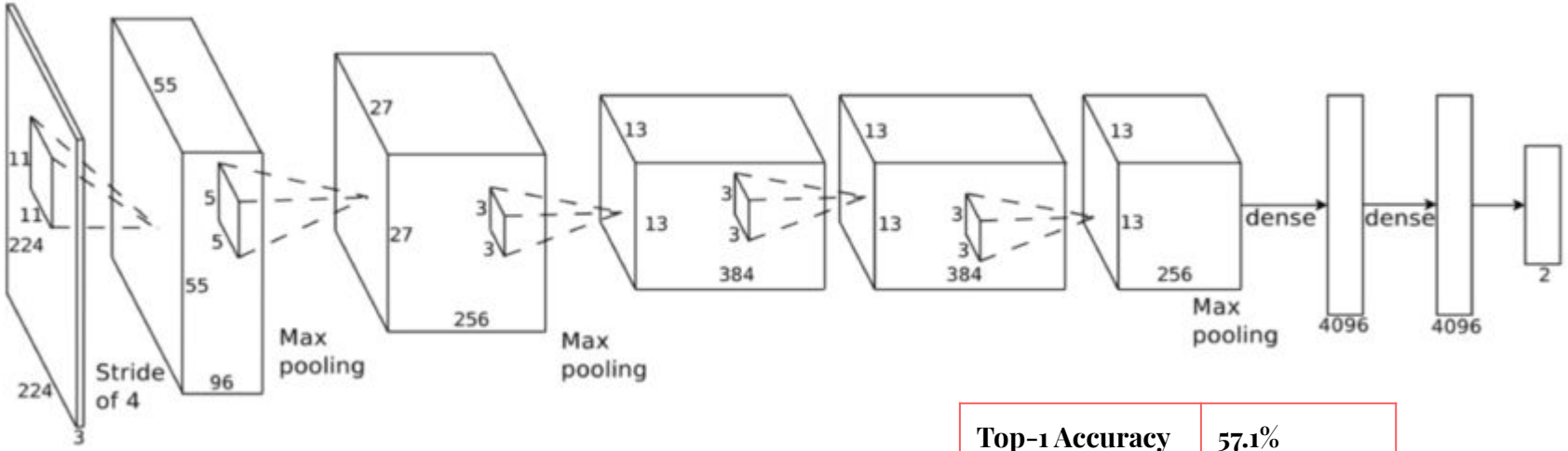
- Downsamples
  - Takes the maximum
  - Takes the average
- Operates at each feature map independently



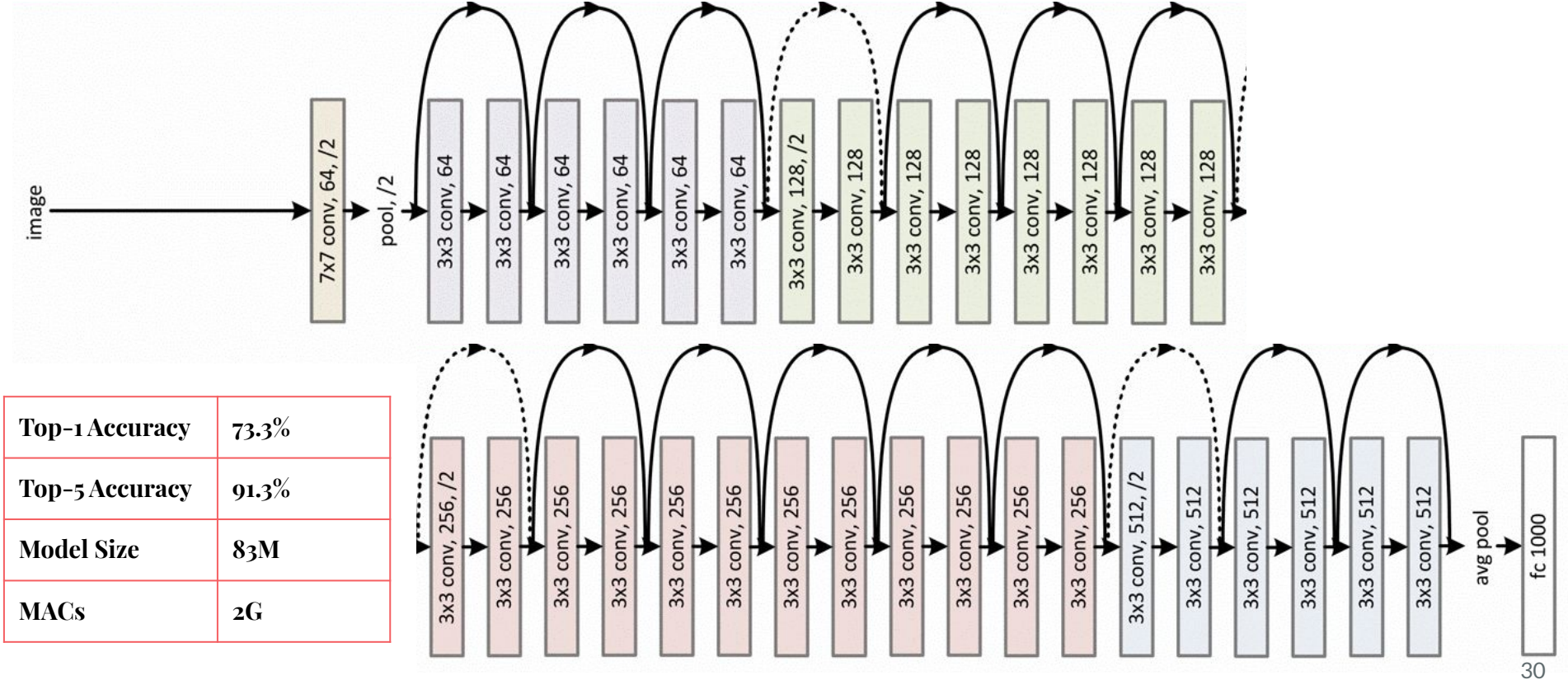

* images from http://cs231n.github.io/convolutional-networks/

# Full DNN Example: AlexNet




| | |
|---|---|
| **Top-1 Accuracy** | **57.1%** |
| **Top-5 Accuracy** | **80.2%** |
| **Model Size** | **61M** |
| **MACs** | **725M** |

# Full DNN Example: ResNet-34

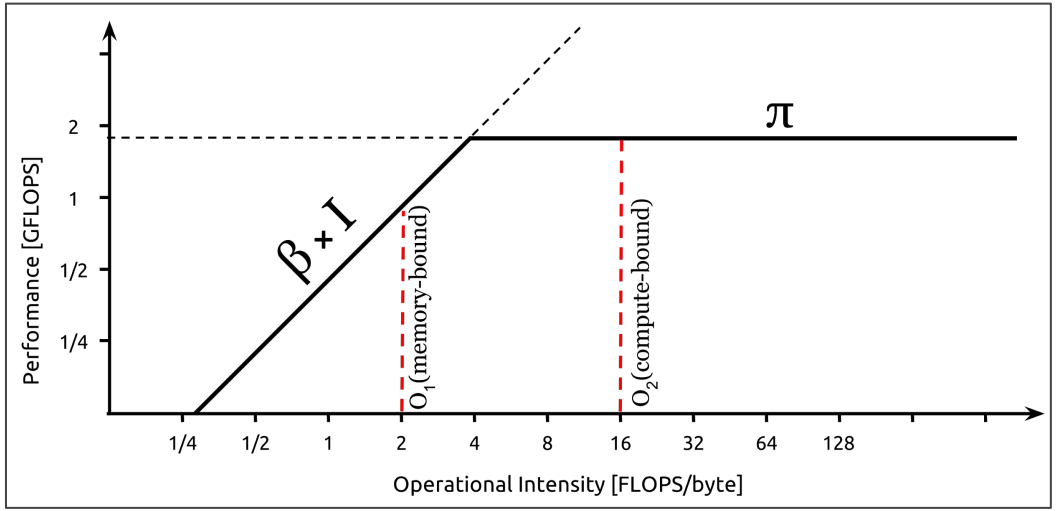



| | |
|---|---|
| **Top-1 Accuracy** | **73.3%** |
| **Top-5 Accuracy** | **91.3%** |
| **Model Size** | **83M** |
| **MACs** | **2G** |

# How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload

# The Roofline Model




- π - the peak compute performance
- β - the peak bandwidth
- *I* - the arithmetic intensity

- The attainable throughput P:

$$P = \min \begin{cases} \pi \\ \beta \times I \end{cases}$$

- **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

Image from https://en.wikipedia.org/wiki/Roofline_model

# The Roofline Model

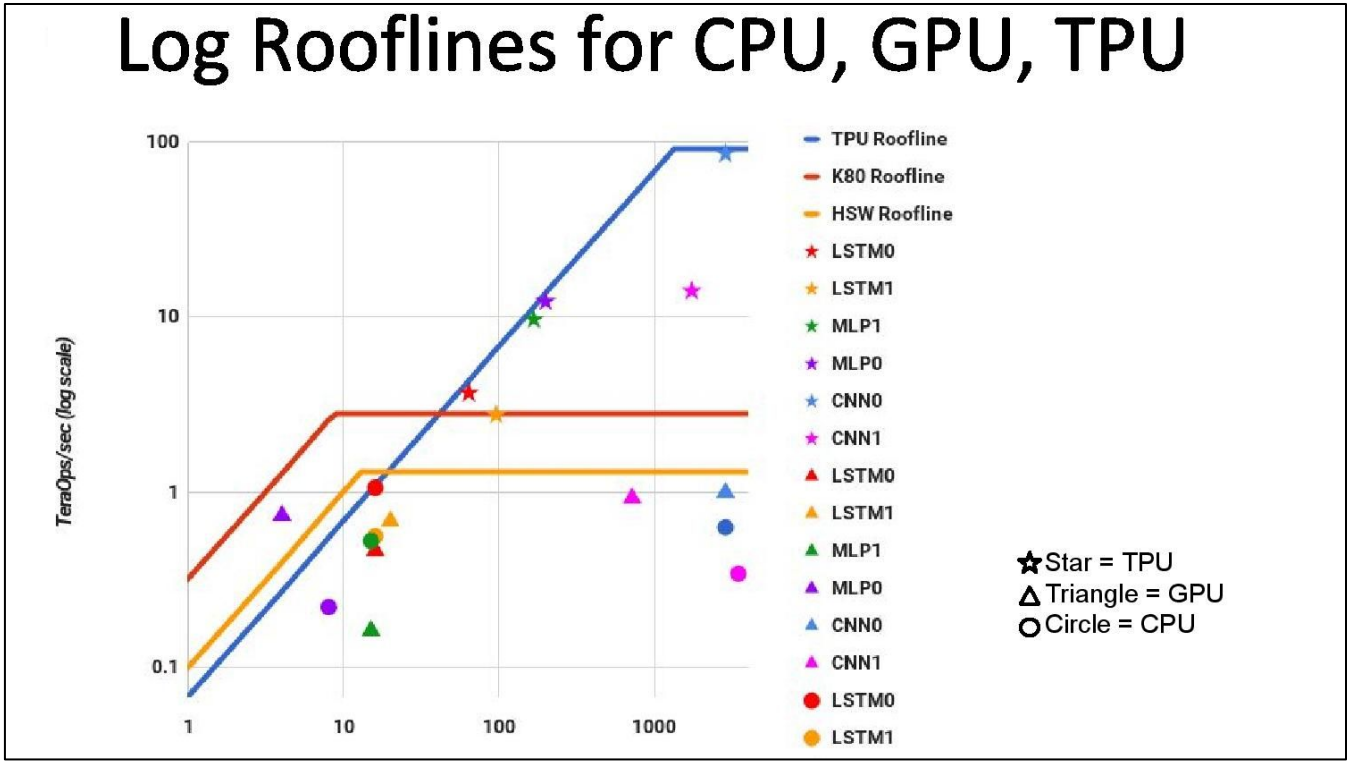


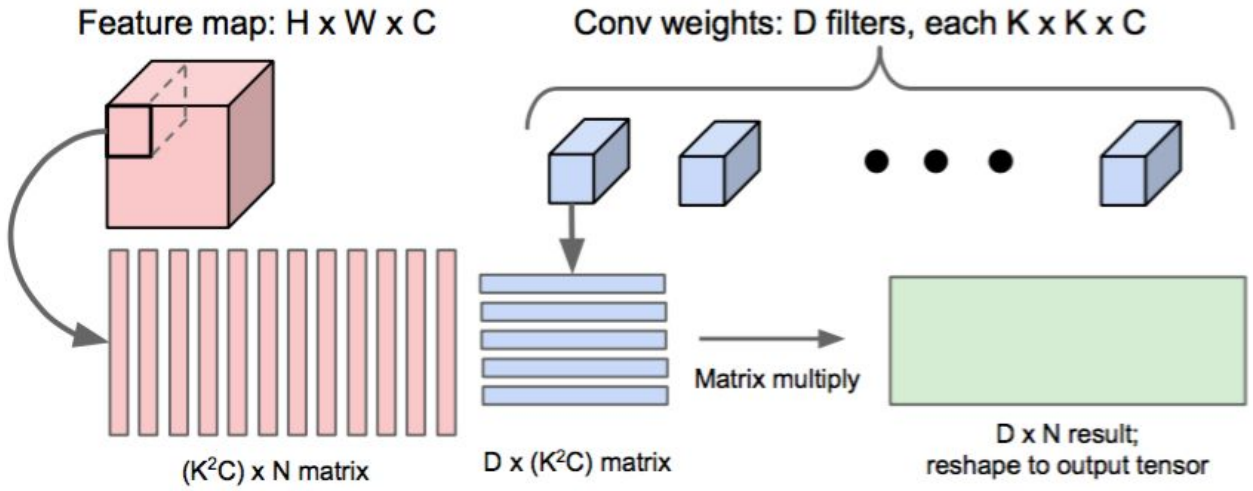

Figure from https://arxiv.org/ftp/arxiv/papers/1704/1704.04760.pdf

# How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload
3. Compare different design options

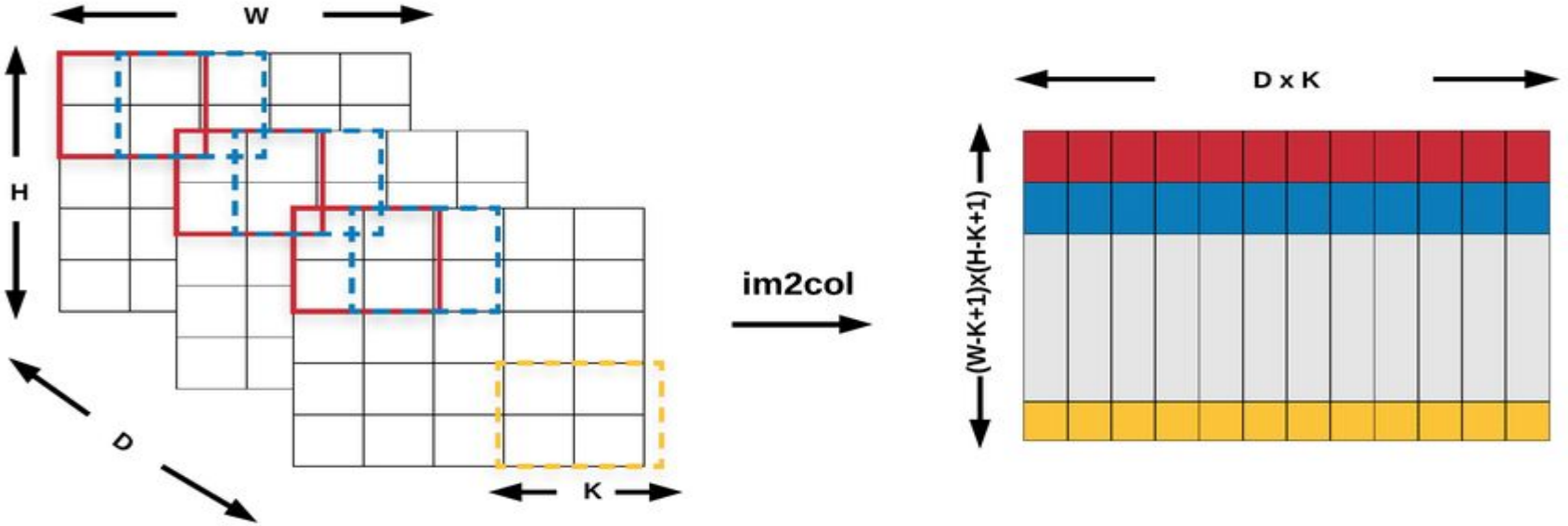
# Conv Mapping 1: Matrix-Matrix Multiplication

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



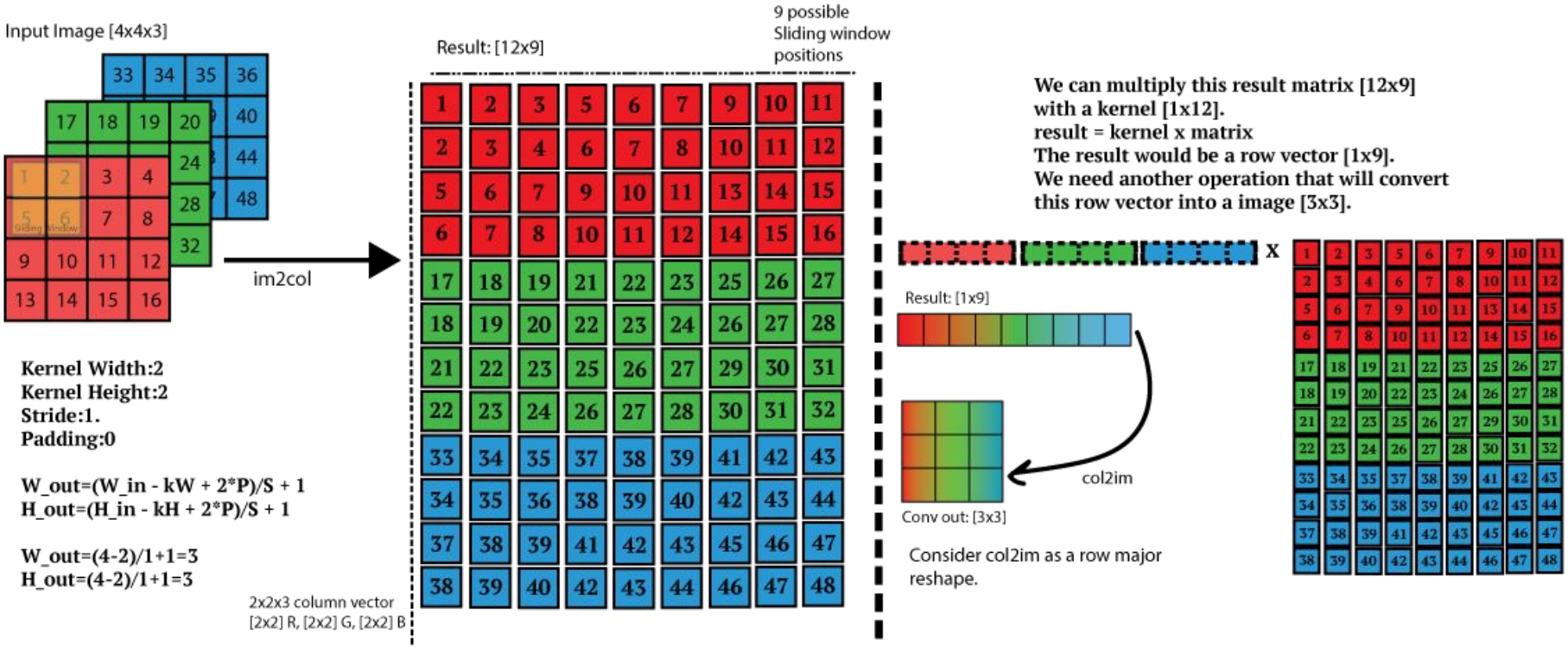

* Image from http://nmhkahn.github.io/CNN-Practice

# Im2col Transform




* from https://www.researchgate.net/publication/327070011_Accelerating_Deep_Neural_Networks_on_Low_Power_Heterogeneous_Architectures

**Image to column operation (im2col)**
**Slide the input image like a convolution but each patch become a column vector.**




* Image from https://github.com/numforge/laser/wiki/Convolution-optimisation-resources

# Optimization: Winograd Algorithm

**Winograd** performs convolution in a transformed domain to reduces the total number of multiplications.

GEMM Example:

Inputs: $f = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix}, g = \begin{bmatrix} -1 \\ -2 \\ -3 \end{bmatrix}$

Transformed Inputs:

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Result:

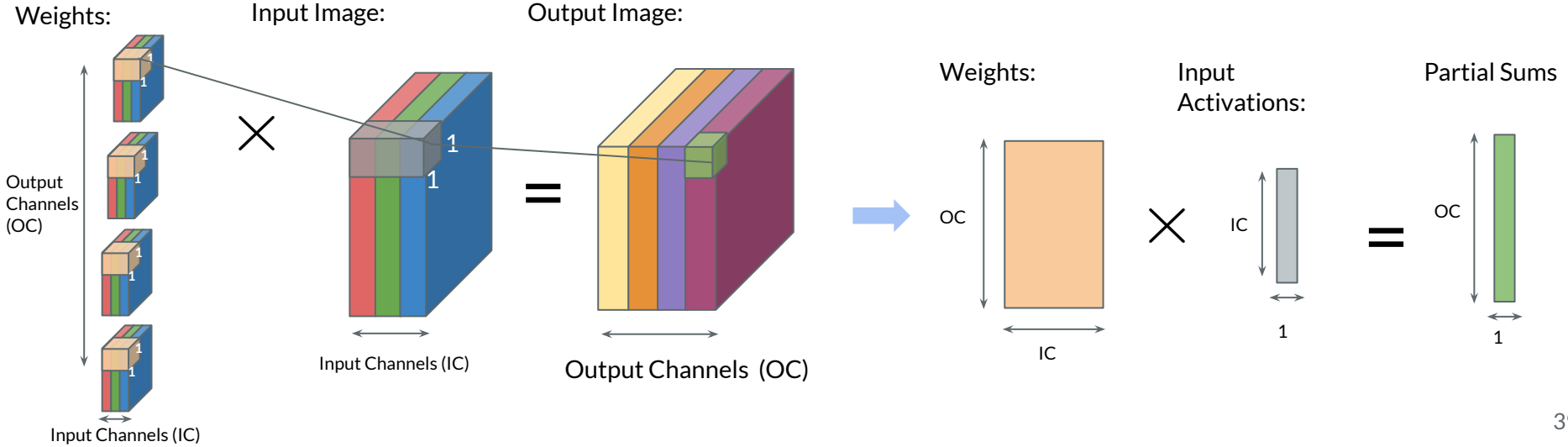
$$\begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix}$$

6 MUL

4 MUL

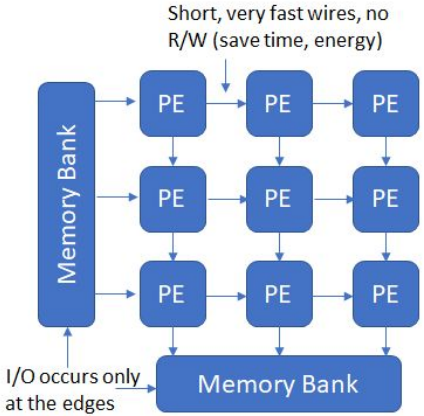
# Conv Mapping 2: Matrix-Vector Multiplication

- 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 x K pixels (K is the kernel size)

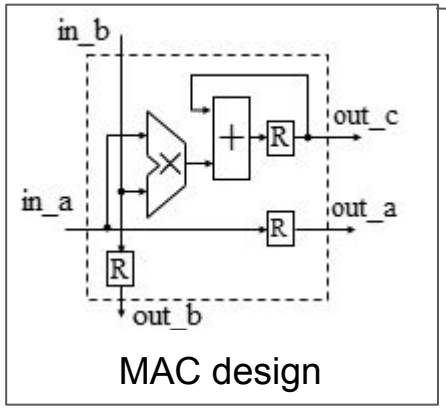


# Implementation: Systolic Array

- **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

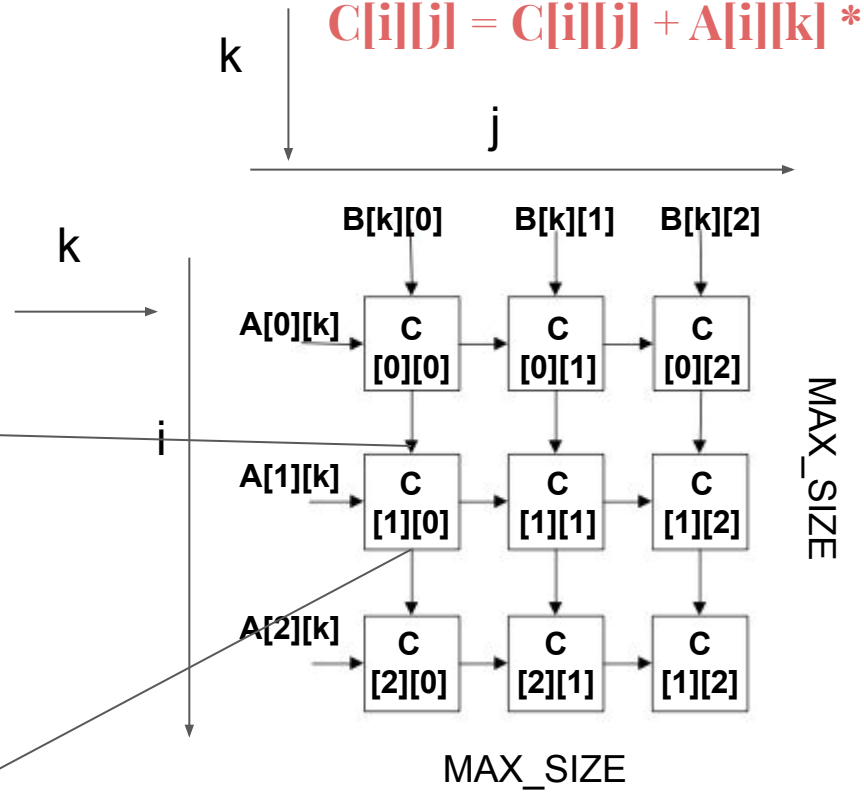



System Architecture



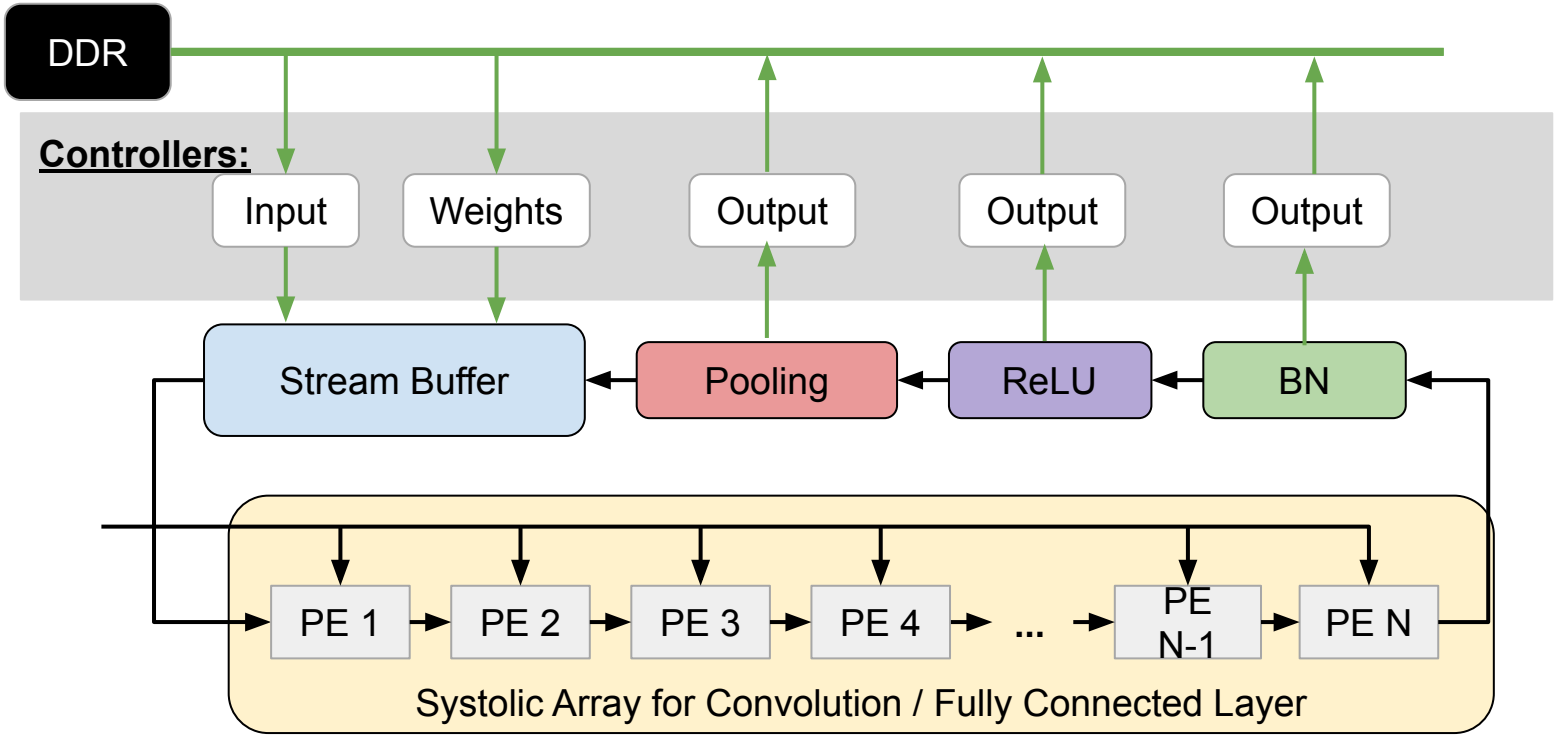

MAC design

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

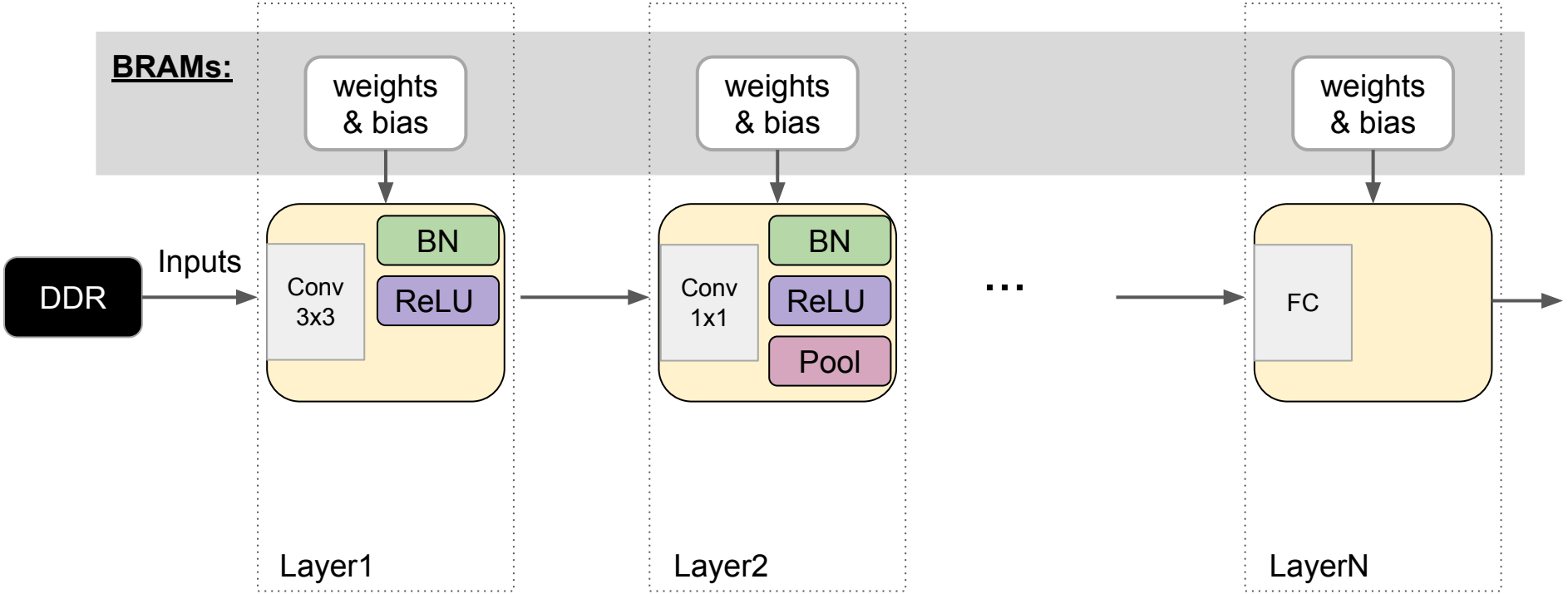



* Images from http://www.telesens.co/2018/07/30/systolic-architectures/

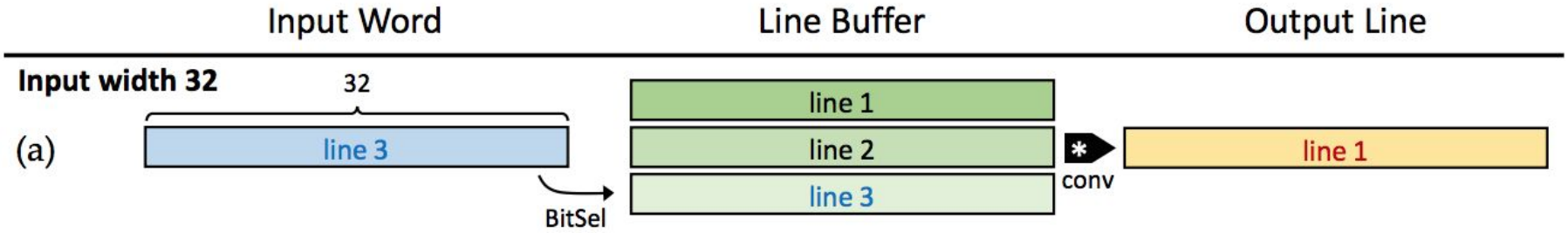
# DNN Accelerator Design 1: Layer-based



# DNN Accelerator Design 2: Spatially-mapped



# Line-Buffer Design

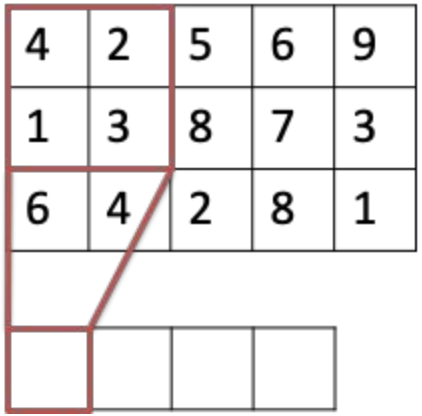



- Buffers inputs to perform spatial operations
- Buffers inputs for reuse to improve the arithmetic intensity

* Ritchie Zhao, et al. Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)

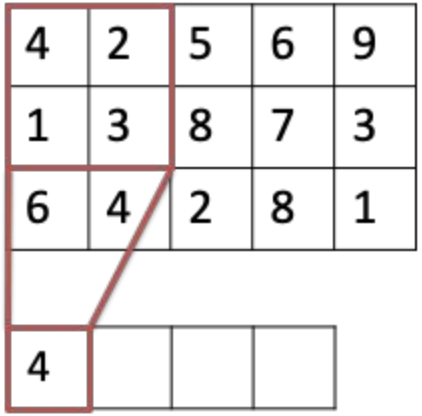
# Line-Buffer Execution Model

- 2x2 Max Pooling



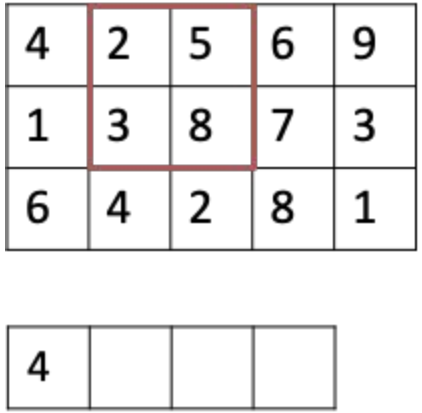
# Line-Buffer Execution Model

- 2x2 Max Pooling



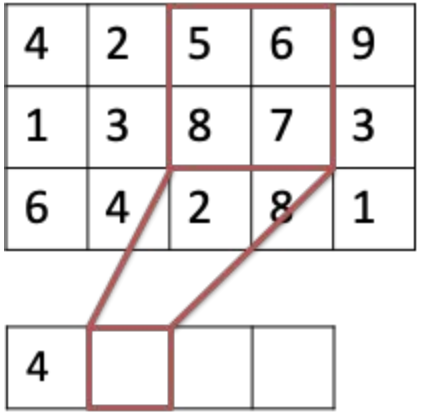
# Line-Buffer Execution Model

- 2x2 Max Pooling



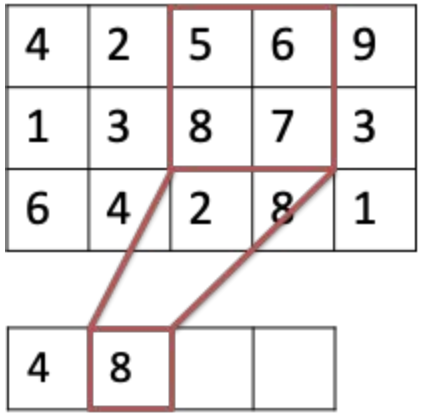
# Line-Buffer Execution Model

- 2x2 Max Pooling



# Line-Buffer Execution Model

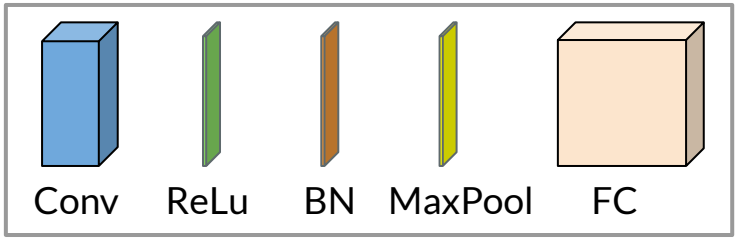
- 2x2 Max Pooling

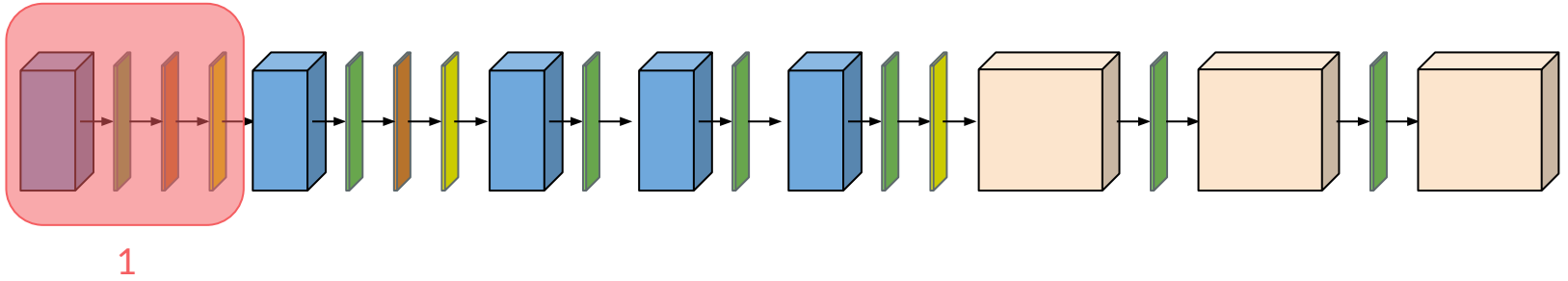


# How to design your own DNN accelerator?

1. Understand the basic operations
2. Analyze the workload
3. Compare different design options
4. Develop software runtime

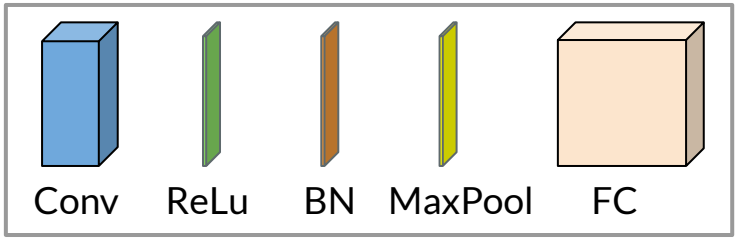
# Execution Model

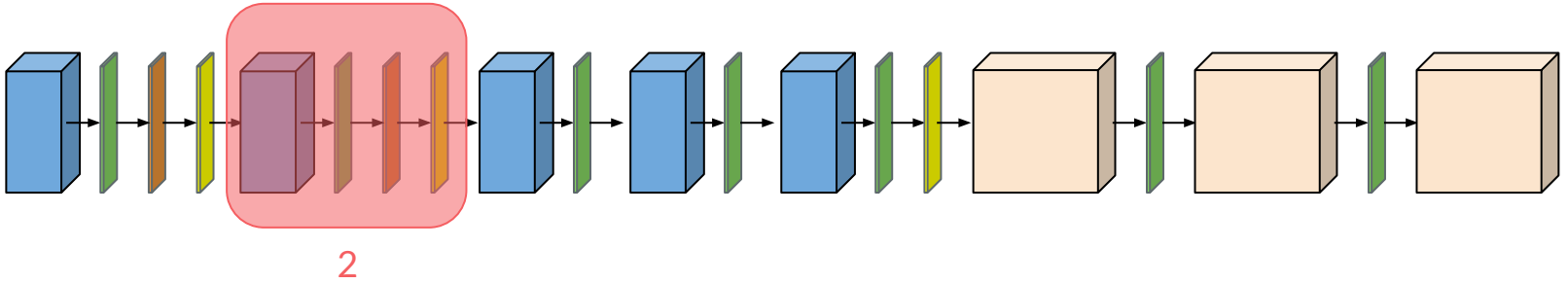



AlexNet Design

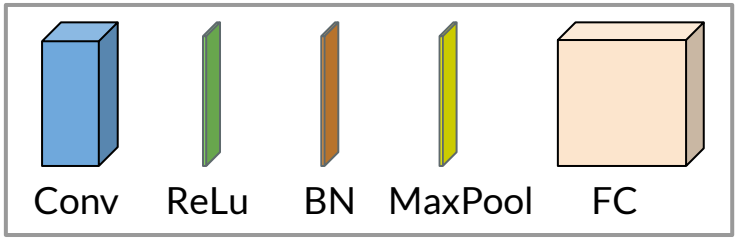
# Execution Model

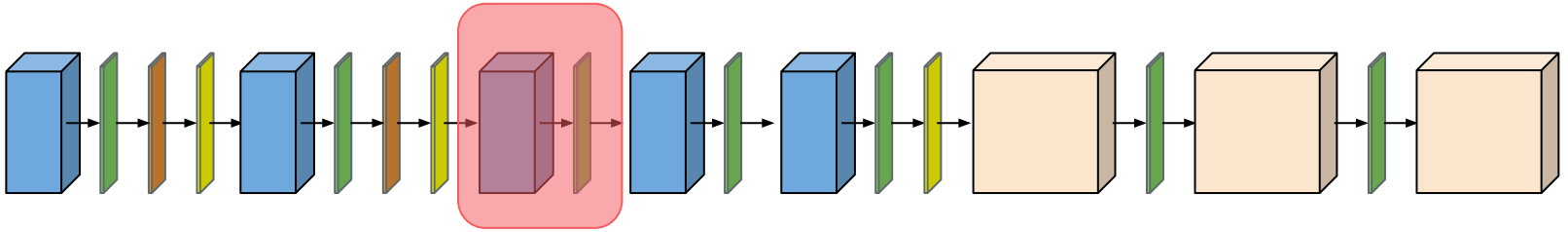



AlexNet Design

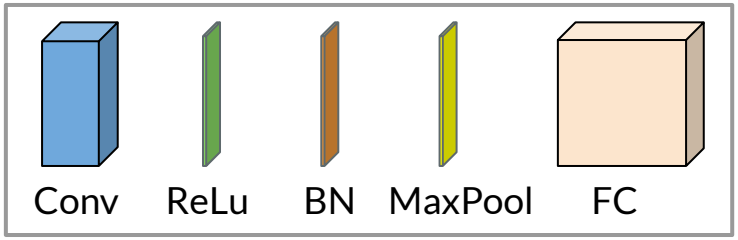
# Execution Model

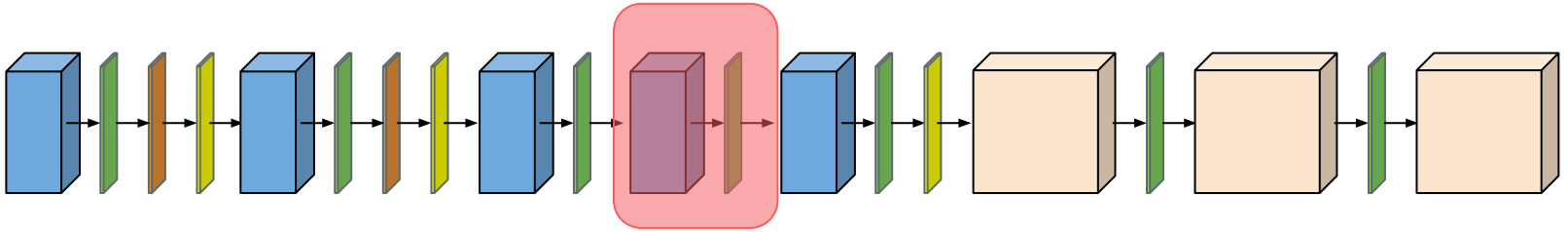



AlexNet Design

# Execution Model

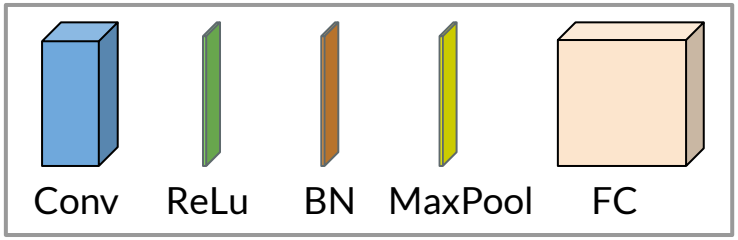


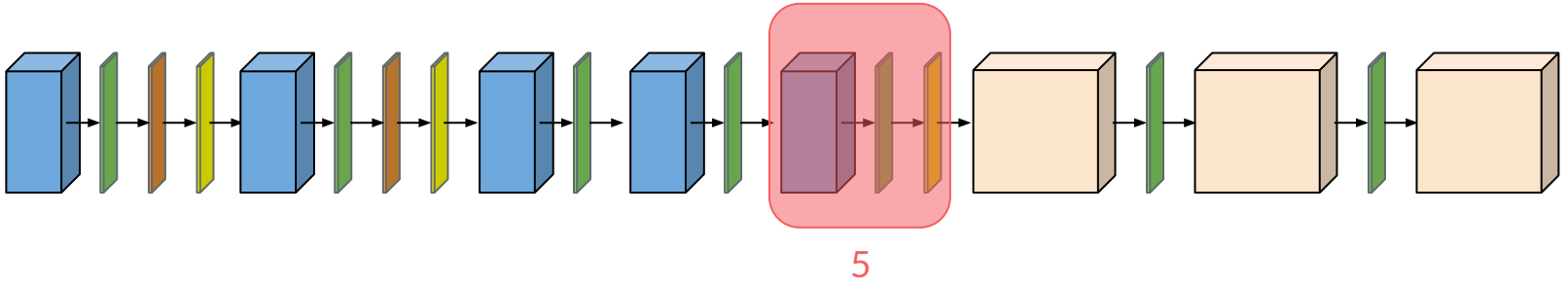



4

AlexNet Design

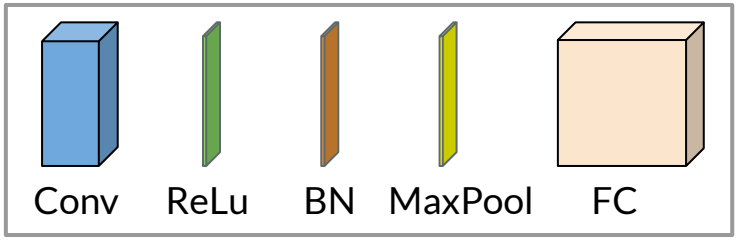
# Execution Model

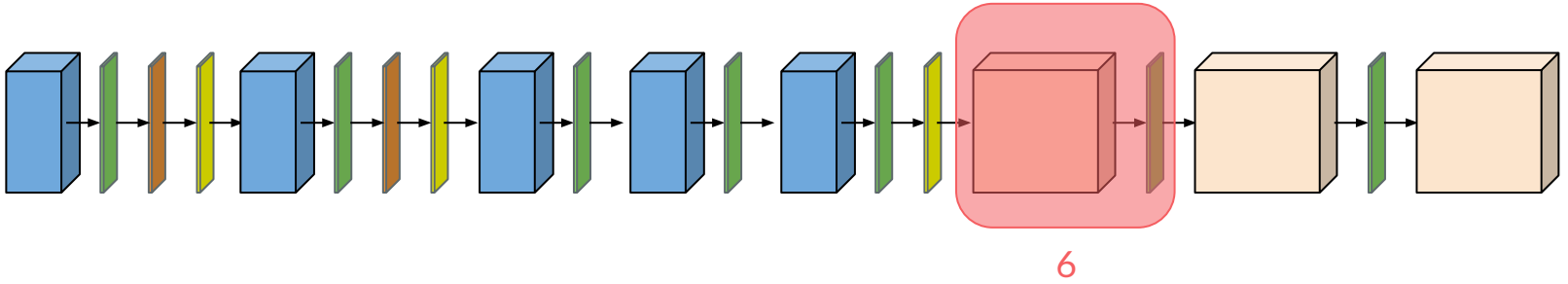



AlexNet Design

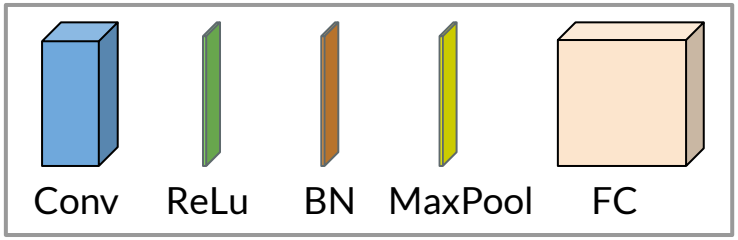
# Execution Model

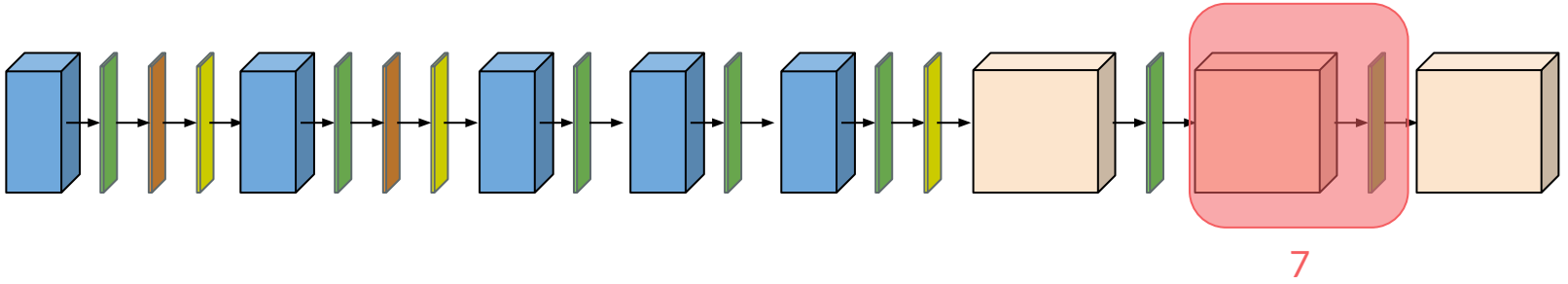



AlexNet Design

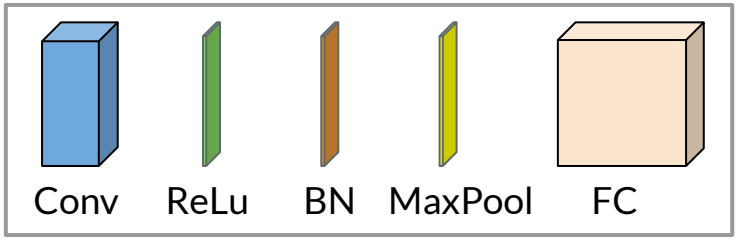
# Execution Model

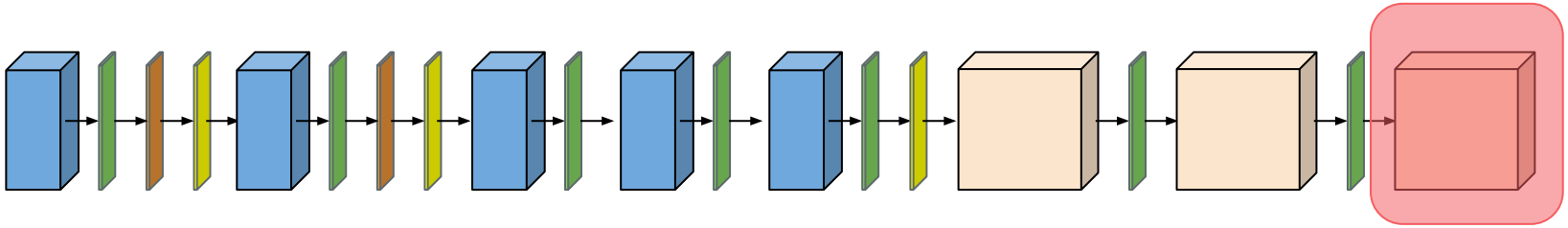



AlexNet Design

# Execution Model




AlexNet Design

# HLS

# High-Level Synthesis (HLS)

- 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

- EE290-2: Hardware for Machine Learning
- MIT Eyeriss Tutorial
- 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?