

# Hardware Architectures for Deep Neural Networks

ISCA Tutorial

June 24, 2017

Website: <http://eyeriss.mit.edu/tutorial.html>



Massachusetts  
Institute of  
Technology



nVIDIA®

# Speakers and Contributors

---



**Joel Emer**

*Senior Distinguished  
Research Scientist*

**NVIDIA**

*Professor*

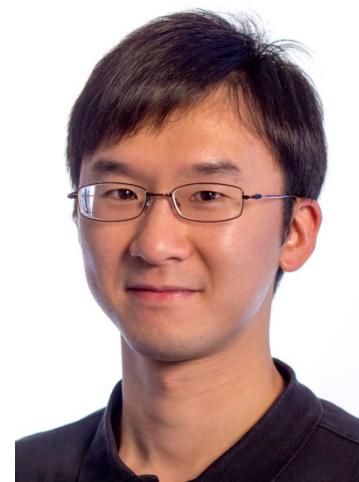
**MIT**



**Vivienne Sze**

*Professor*

**MIT**



**Yu-Hsin Chen**

*PhD Candidate*

**MIT**



**Tien-Ju Yang**

*PhD Candidate*

**MIT**

# Outline

---

- **Overview of Deep Neural Networks**
- **DNN Development Resources**
- **Survey of DNN Hardware**
- **DNN Accelerators**
- **DNN Model and Hardware Co-Design**

# Participant Takeaways

---

- **Understand the key design considerations for DNNs**
- **Be able to evaluate different implementations of DNN with benchmarks and comparison metrics**
- **Understand the tradeoffs between various architectures and platforms**
- **Assess the utility of various optimization approaches**
- **Understand recent implementation trends and opportunities**

# Resources

---

- **Eyeriss Project:** <http://eyeriss.mit.edu>
  - Tutorial Slides
  - Benchmarking
  - Energy modeling
  - Mailing List for updates
    - <http://mailman.mit.edu/mailman/listinfo/eems-news>
  - **Paper based on today's tutorial:**
    - V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, “*Efficient Processing of Deep Neural Networks: A Tutorial and Survey*”, arXiv, 2017



Follow @eems\_mit

# **Background of Deep Neural Networks**

# Artificial Intelligence

---

## Artificial Intelligence

“The science and engineering of creating intelligent machines”

- John McCarthy, 1956

# AI and Machine Learning

---

**Artificial Intelligence**

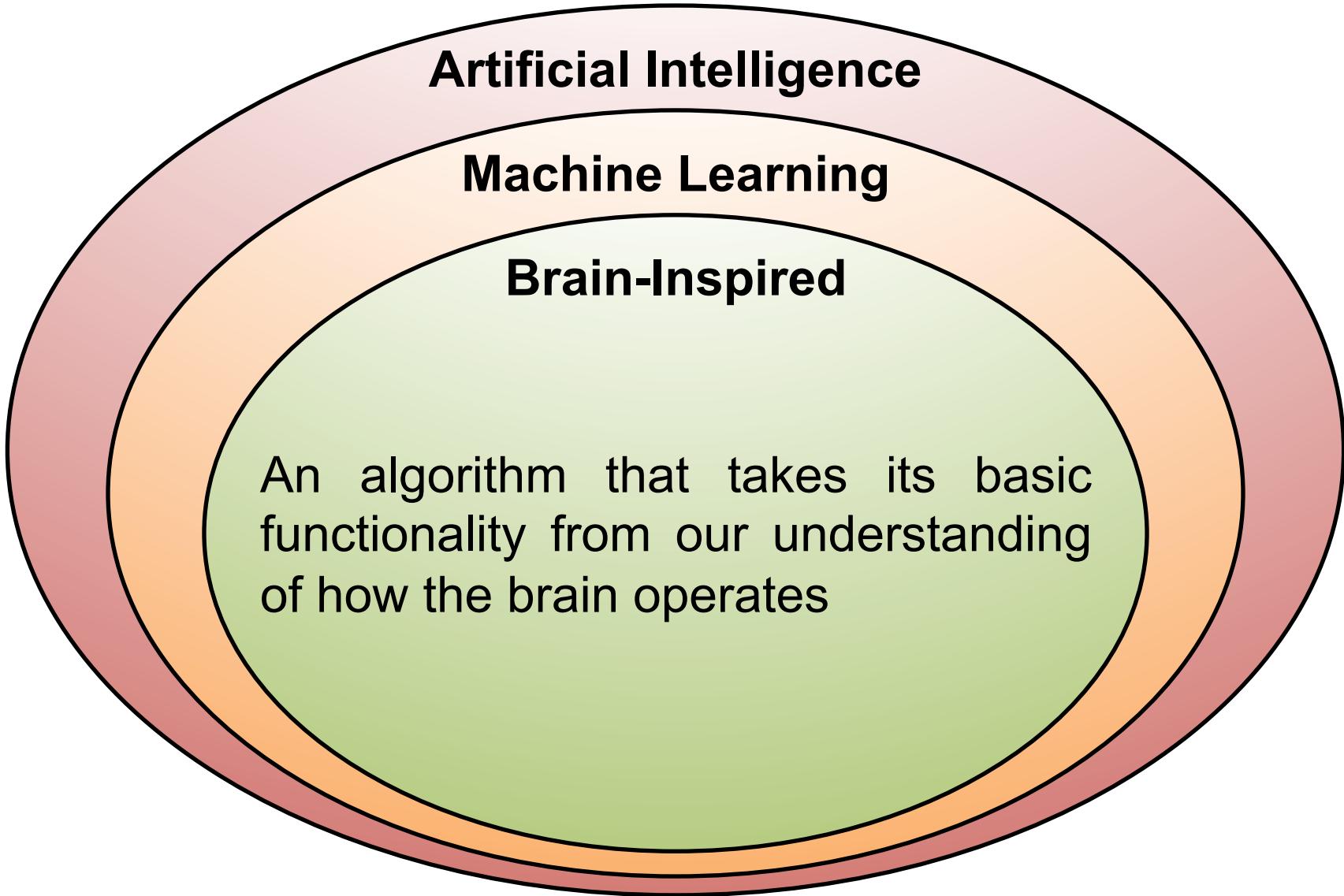
**Machine Learning**

“Field of study that gives computers the ability to learn without being explicitly programmed”

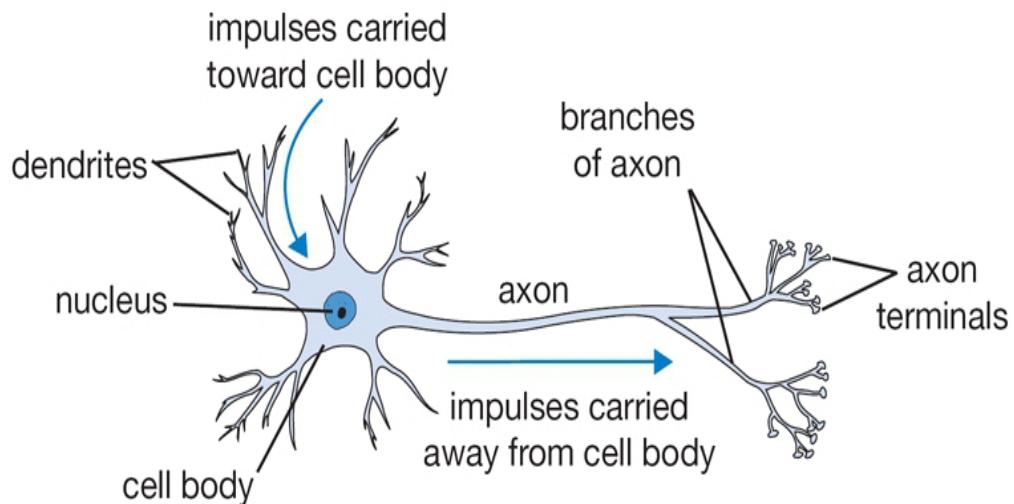
– Arthur Samuel, 1959

# Brain-Inspired Machine Learning

---

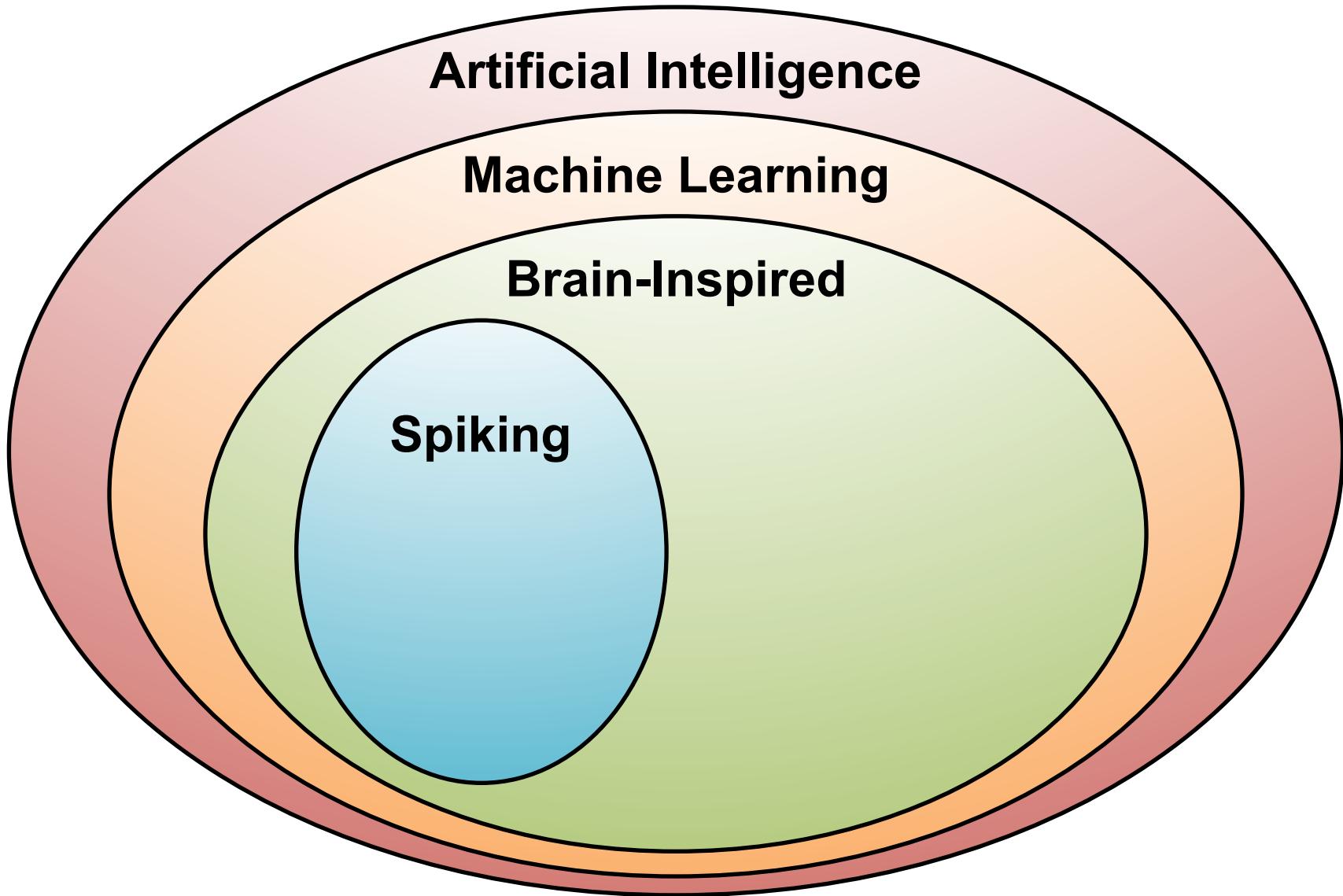


# How Does the Brain Work?



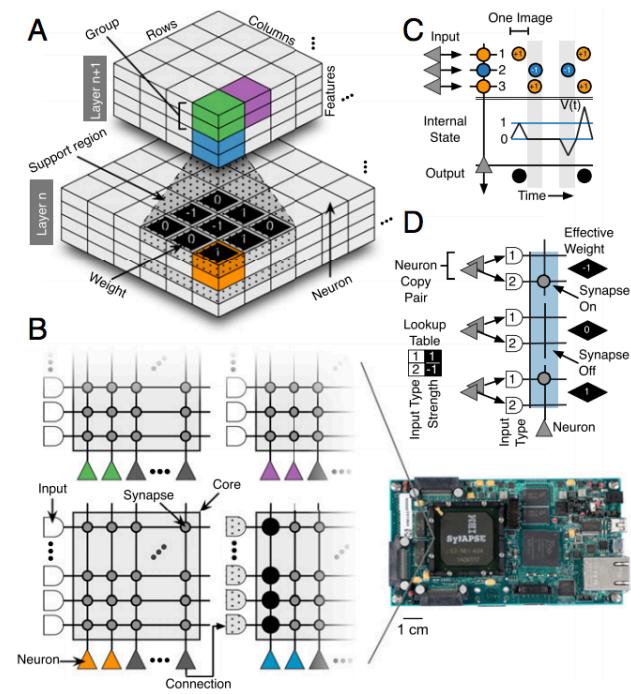
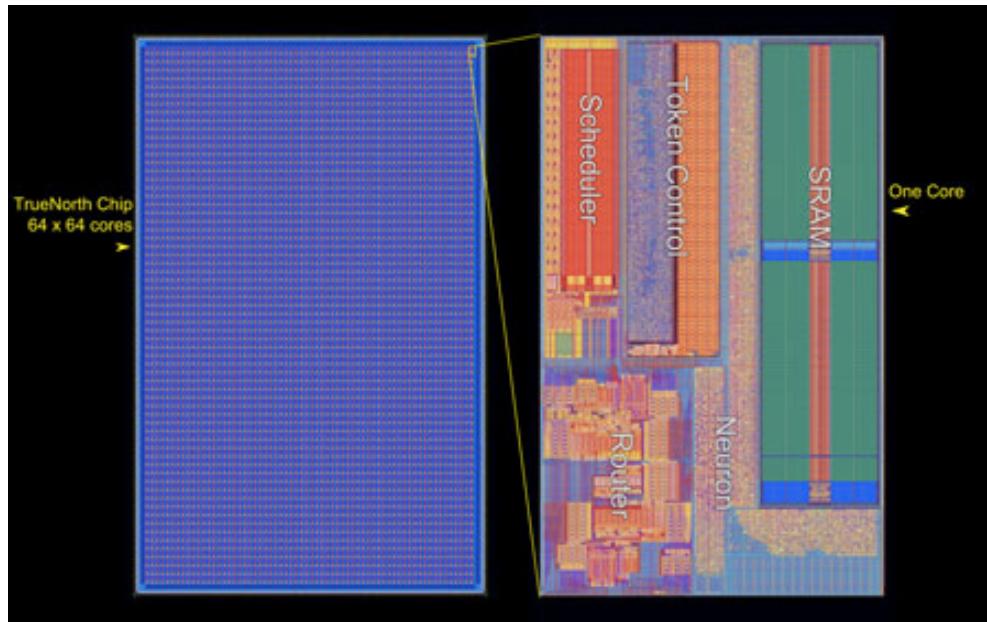
- The basic computational unit of the brain is a **neuron**  
→ 86B neurons in the brain
- Neurons are connected with nearly  $10^{14} – 10^{15}$  **synapses**
- Neurons receive input signal from **dendrites** and produce output signal along **axon**, which interact with the dendrites of other neurons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

# Spiking-based Machine Learning



# Spiking Architecture

- Brain-inspired
- Integrate and fire
- Example: IBM TrueNorth

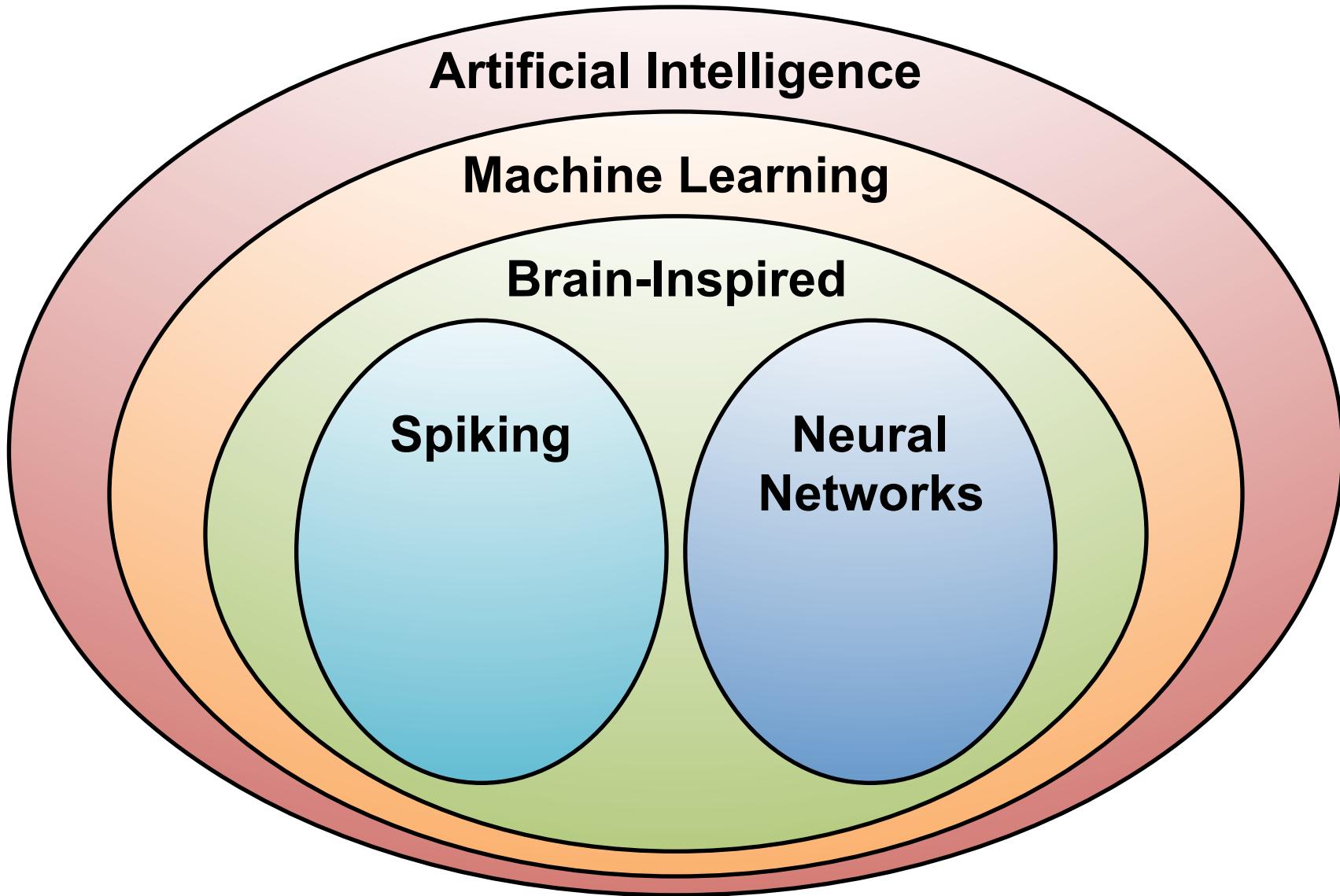


[Merolla et al., Science 2014; Esser et al., PNAS 2016]

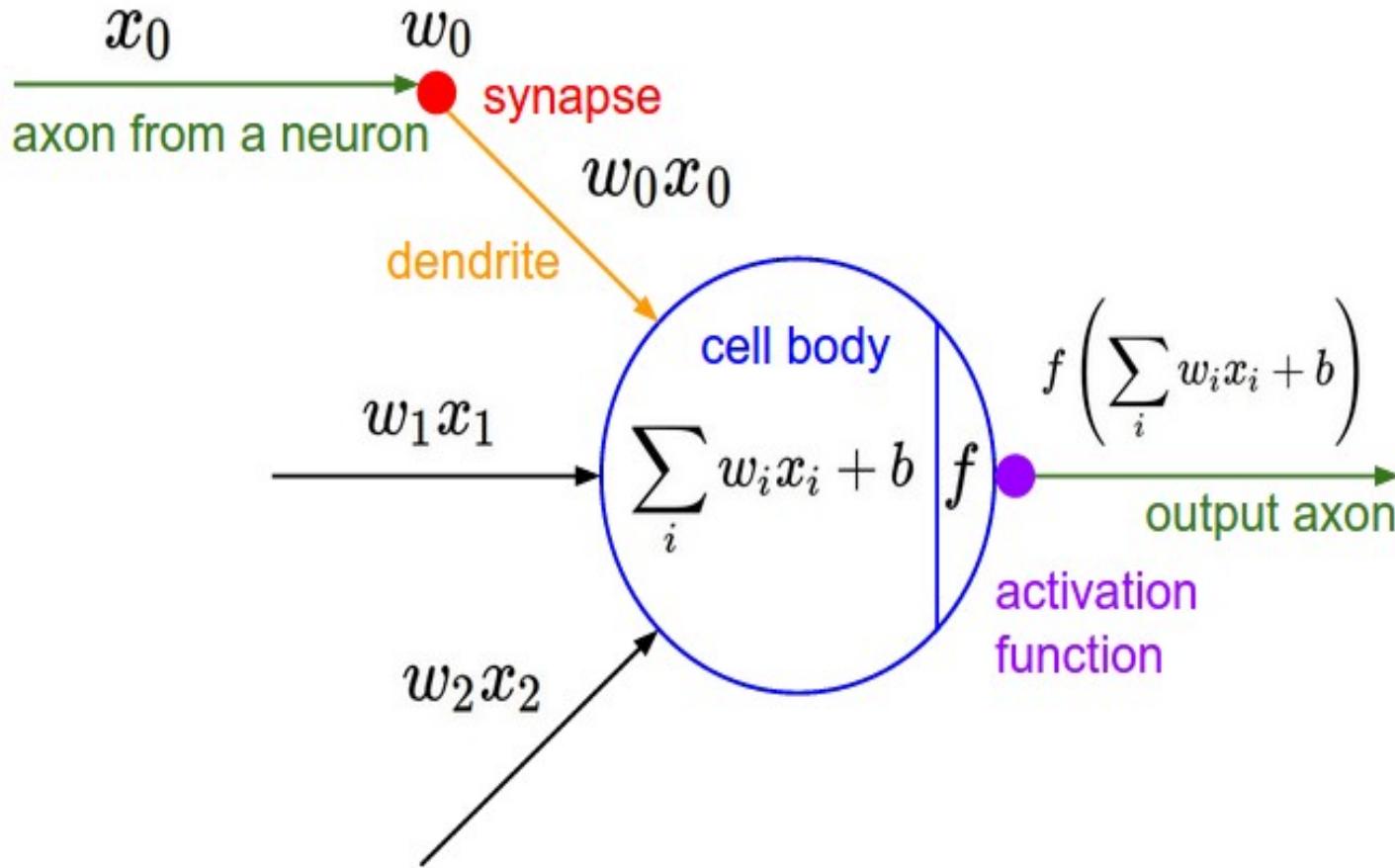
<http://www.research.ibm.com/articles/brain-chip.shtml>

# Machine Learning with Neural Networks

---

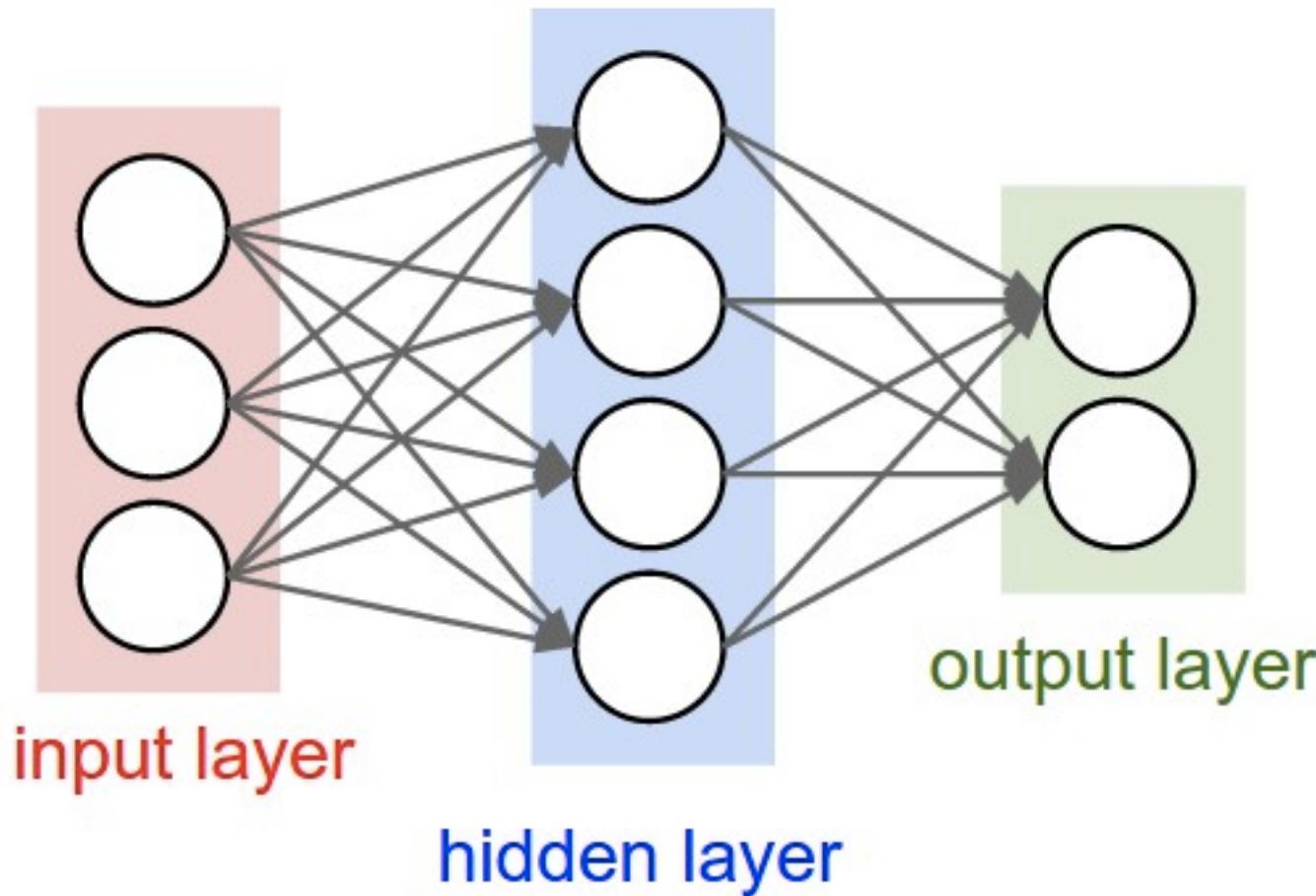


# Neural Networks: Weighted Sum



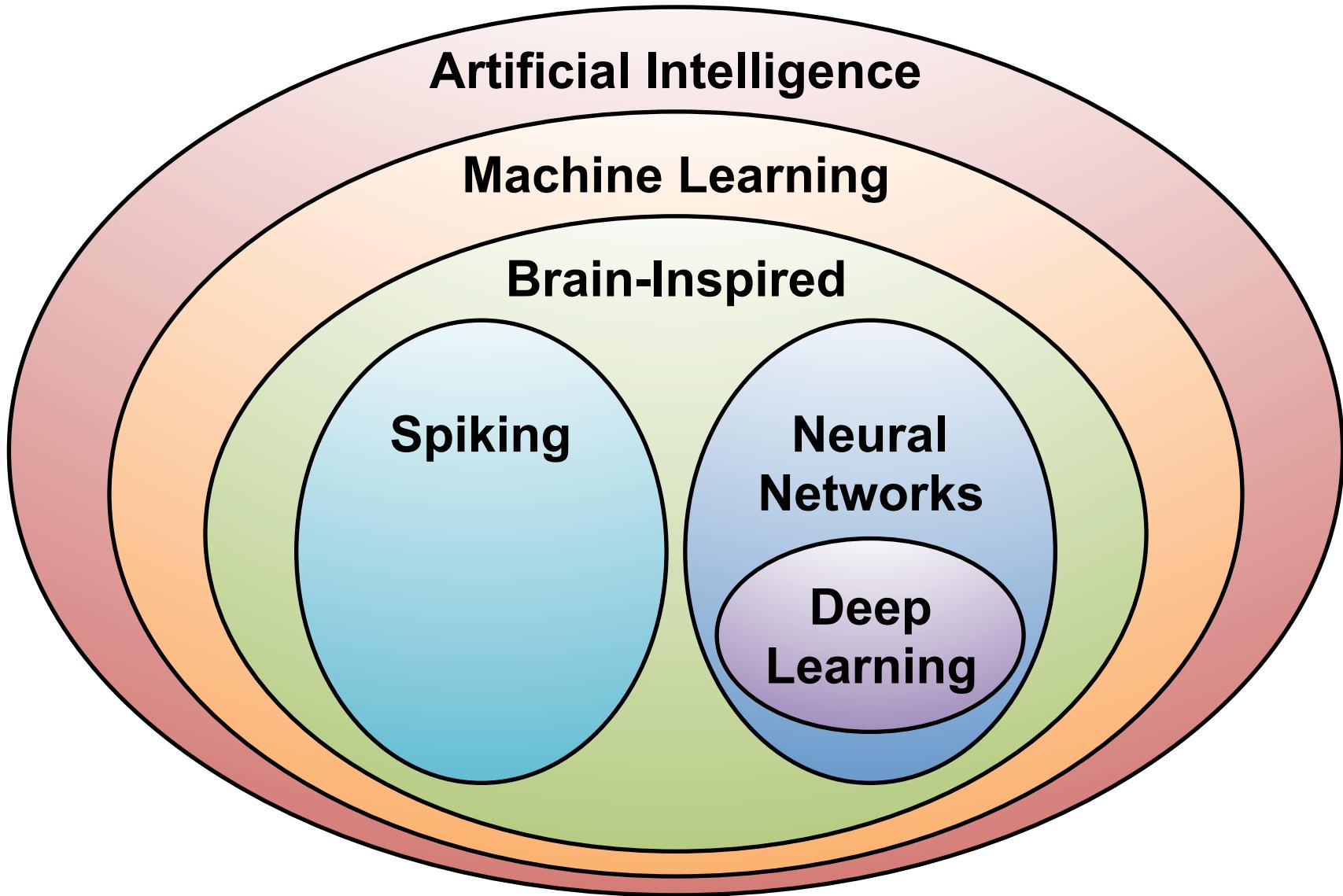
# Many Weighted Sums

---

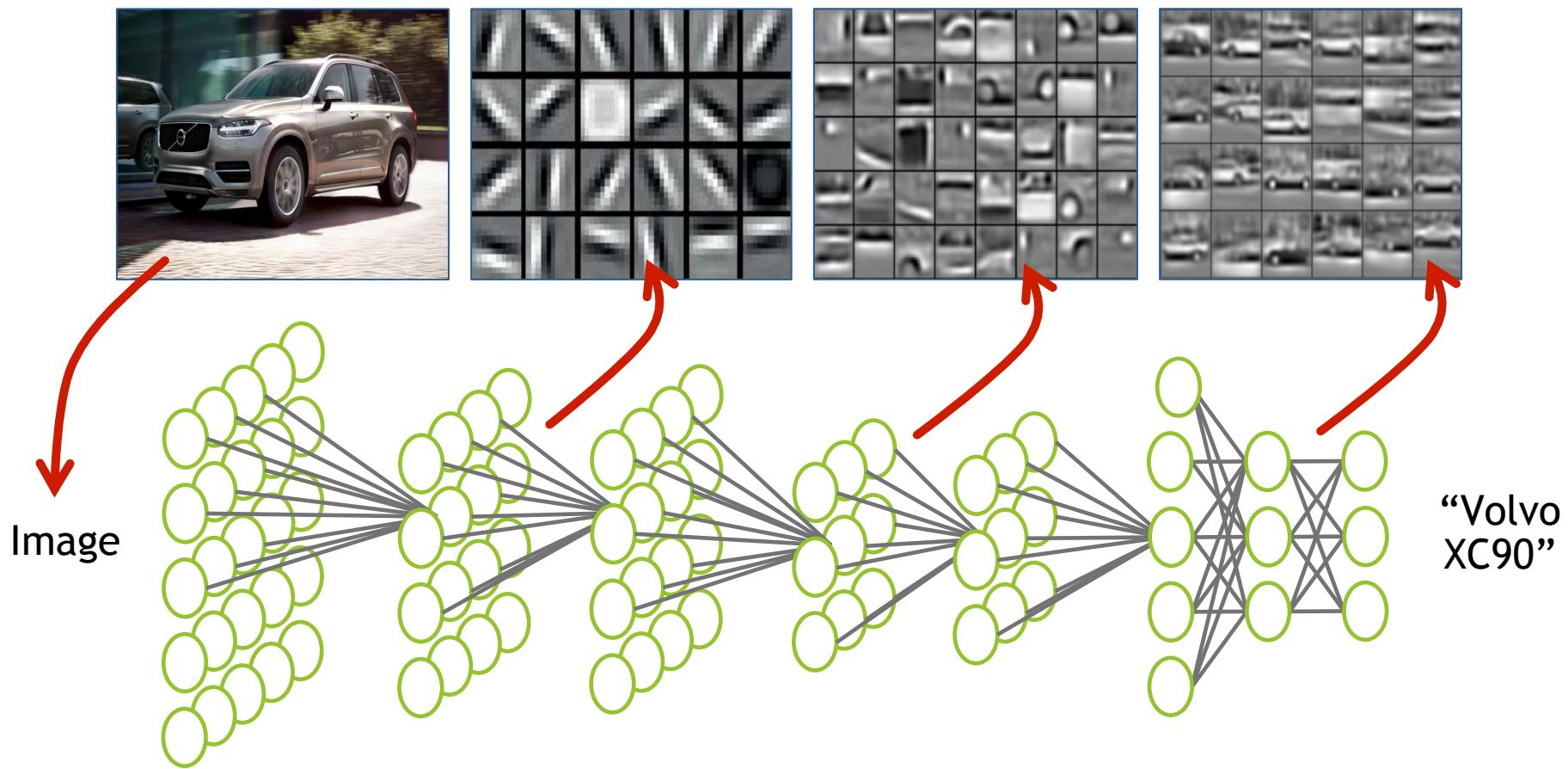


# Deep Learning

---



# What is Deep Learning?



# Why is Deep Learning Hot Now?

Big Data Availability

GPU Acceleration

New ML Techniques

facebook

350M images uploaded per day

Walmart

2.5 Petabytes of customer data hourly

YouTube

300 hours of video uploaded every minute



# ImageNet Challenge

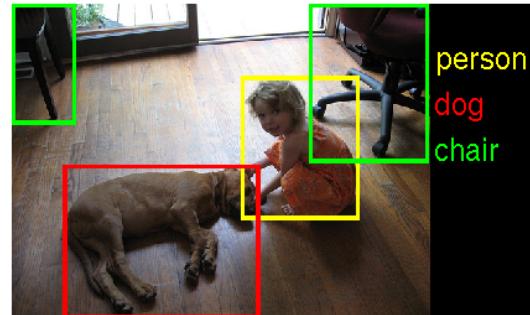


## Image Classification Task:

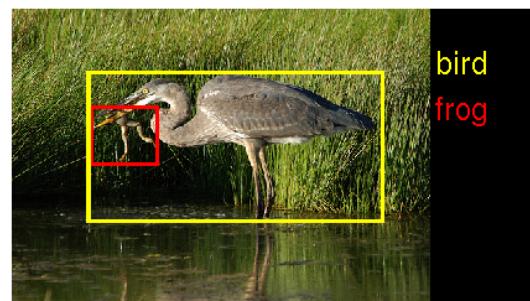
1.2M training images • 1000 object categories

## Object Detection Task:

456k training images • 200 object categories



person  
dog  
chair

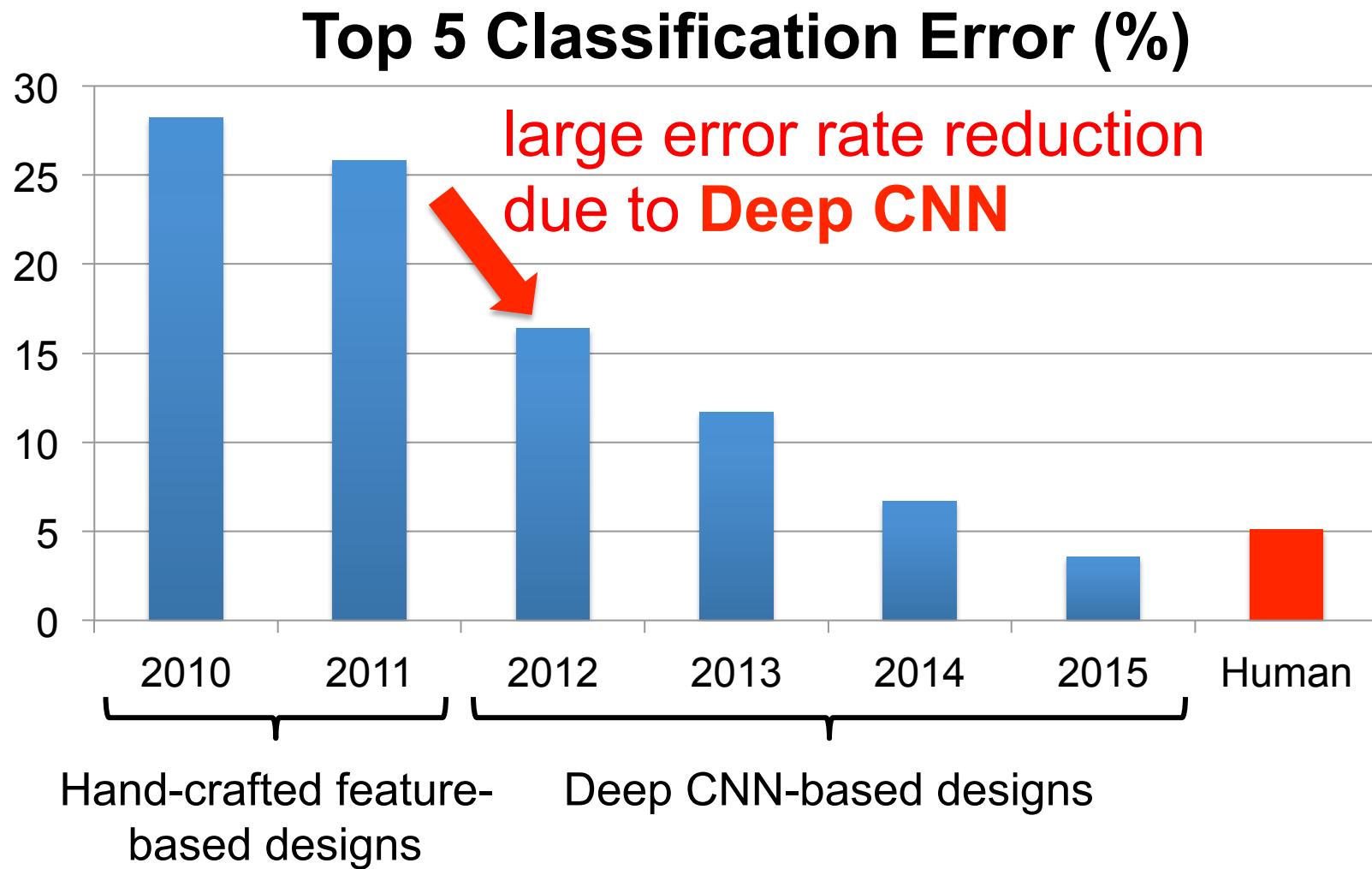


bird  
frog

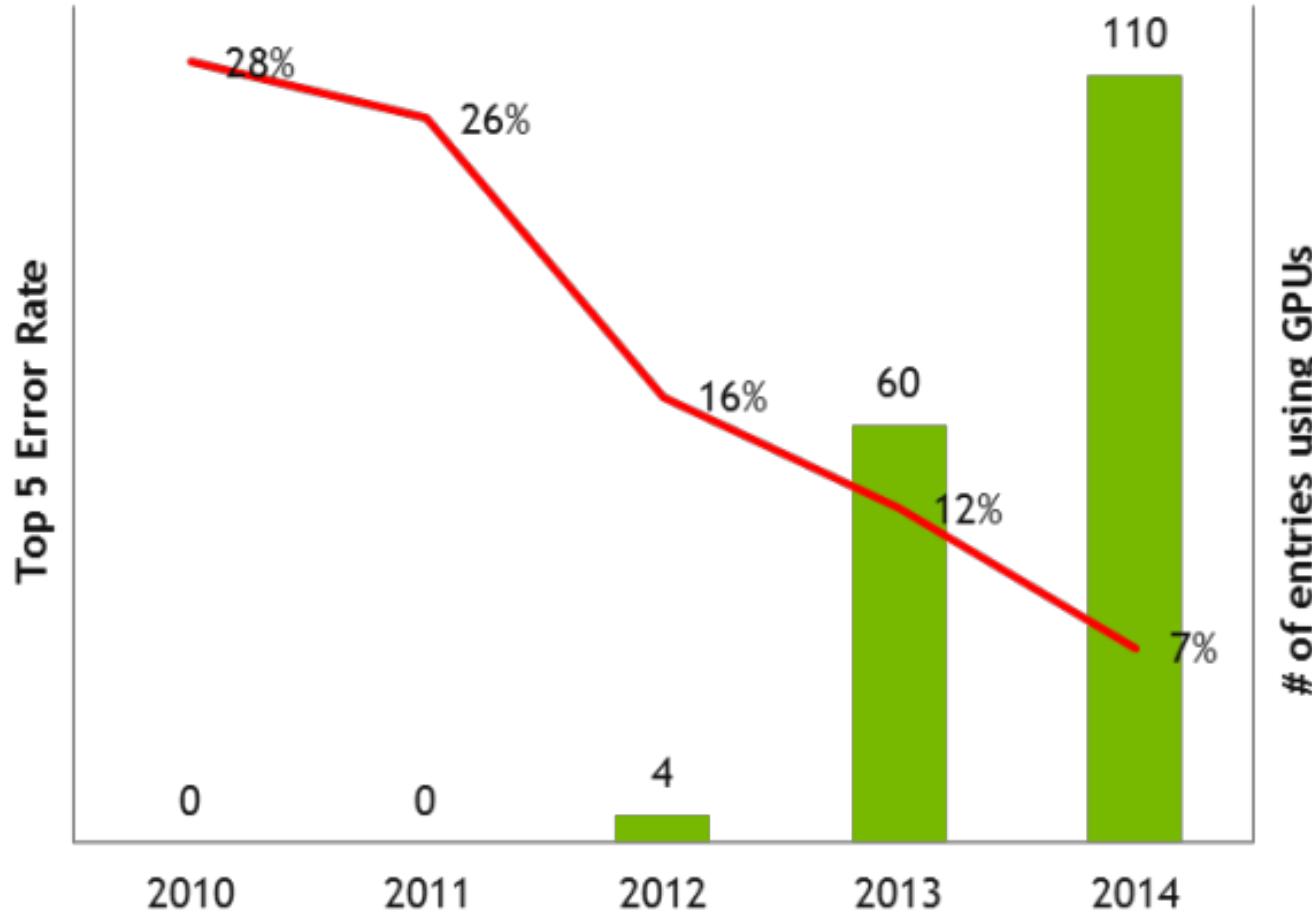


person  
hammer  
flower pot  
power drill

# ImageNet: Image Classification Task



# GPU Usage for ImageNet Challenge



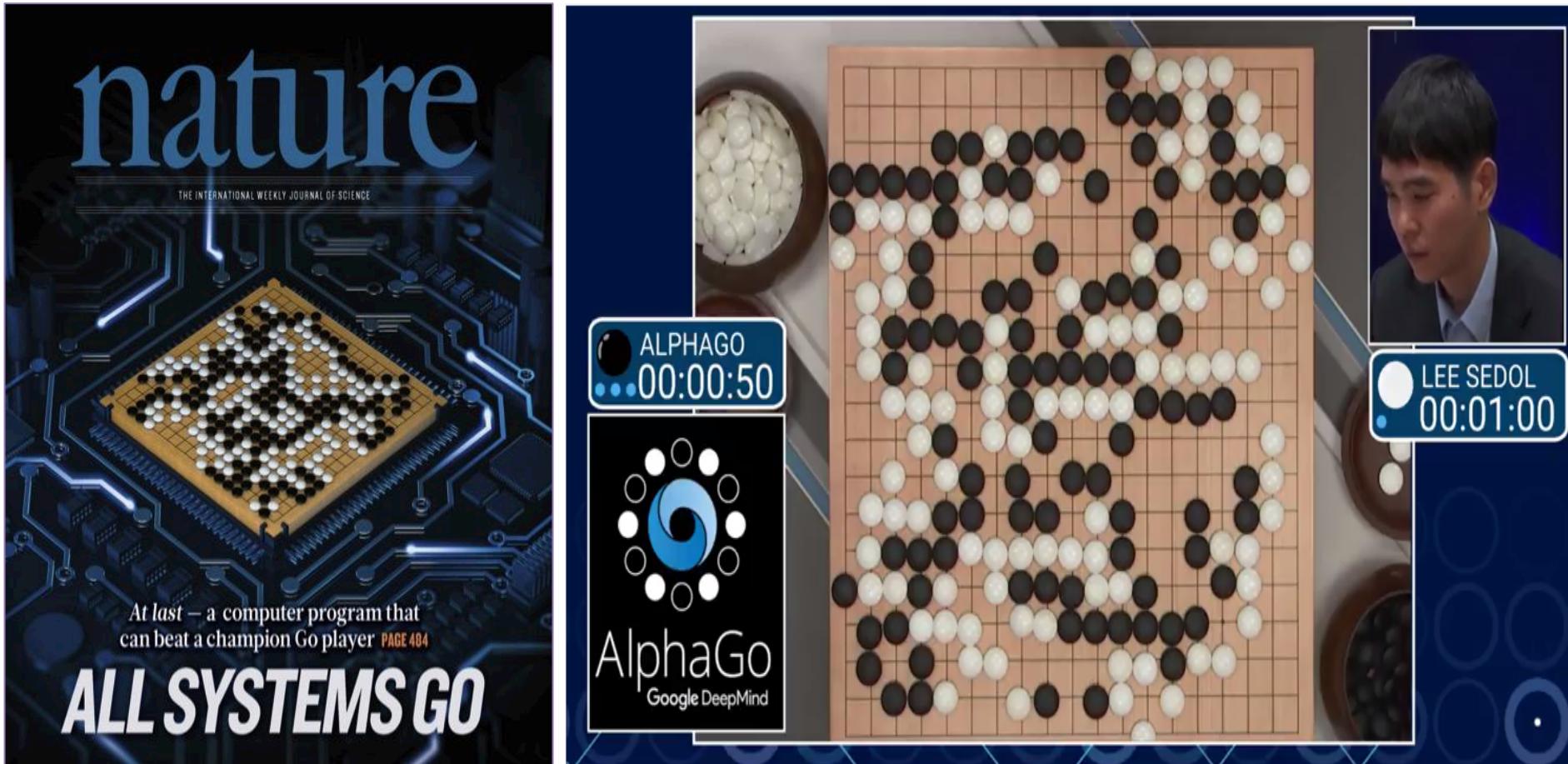
# Established Applications

---

- **Image**
  - Classification: image to object class
  - Recognition: same as classification (except for faces)
  - Detection: assigning bounding boxes to objects
  - Segmentation: assigning object class to every pixel
- **Speech & Language**
  - Speech Recognition: audio to text
  - Translation
  - Natural Language Processing: text to meaning
  - Audio Generation: text to audio
- **Games**

# Deep Learning on Games

## Google DeepMind AlphaGo

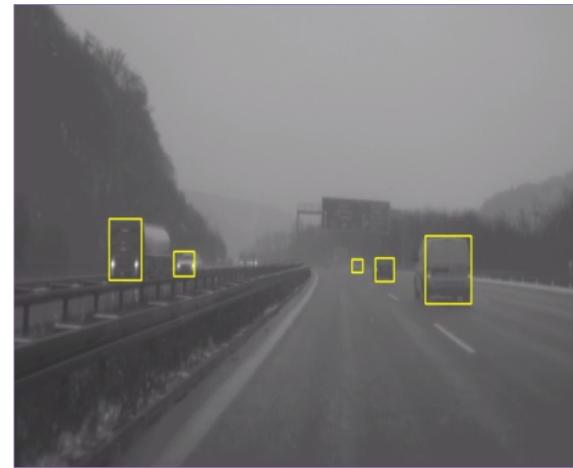
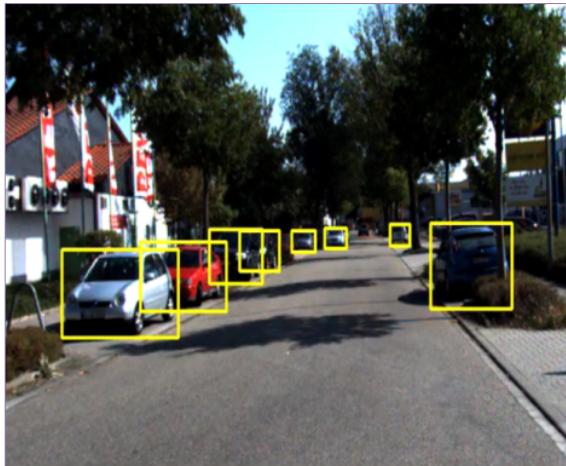


# Emerging Applications

---

- **Medical** (Cancer Detection, Pre-Natal)
- **Finance** (Trading, Energy Forecasting, Risk)
- **Infrastructure** (Structure Safety and Traffic)
- Weather Forecasting and Event Detection

# Deep Learning for Self-driving Cars



# Opportunities

---

From EE Times – September 27, 2016

**”Today the job of training machine learning models is limited by compute, if we had faster processors we’d run bigger models...in practice we train on a reasonable subset of data that can finish in a matter of months. We could use improvements of several orders of magnitude – 100x or greater.”**

– Greg Diamos, Senior Researcher, SVAIL, Baidu

# **Overview of Deep Neural Networks**

# DNN Timeline

---

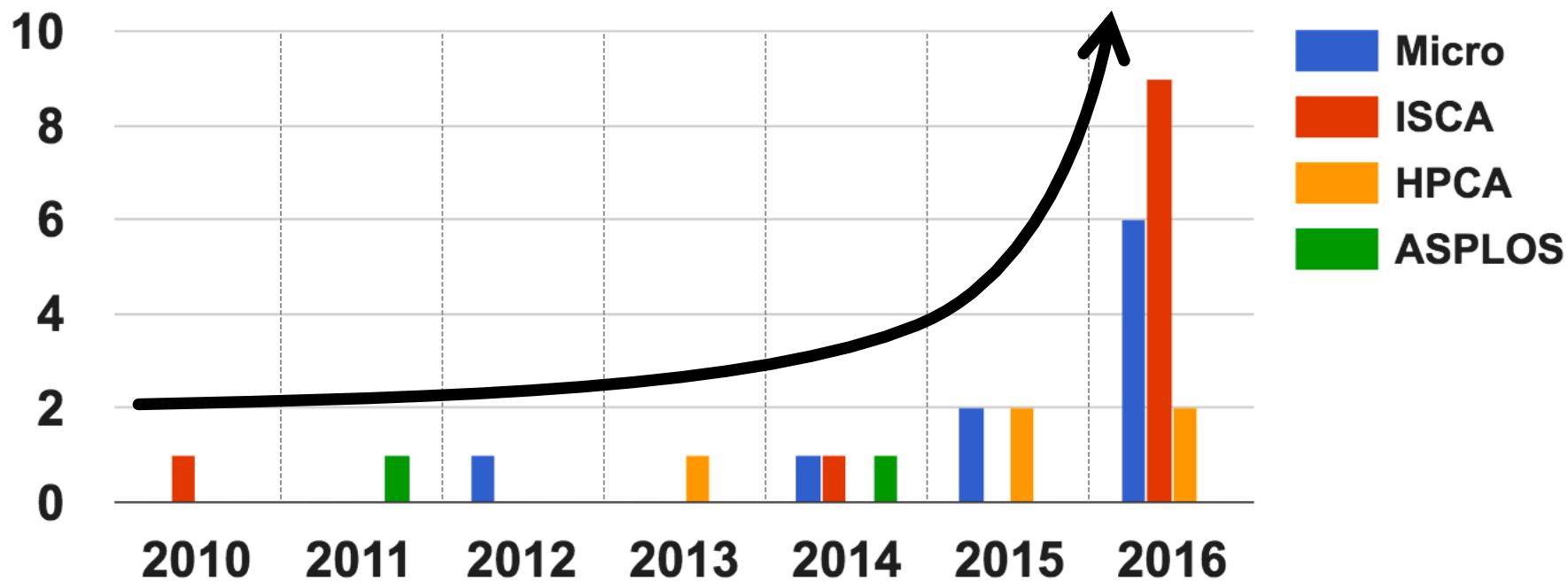
- **1940s: Neural networks were proposed**
- **1960s: Deep neural networks were proposed**
- **1989: Neural network for recognizing digits (LeNet)**
- **1990s: Hardware for shallow neural nets**
  - Example: Intel ETANN (1992)
- **2011: Breakthrough DNN-based speech recognition**
  - Microsoft real-time speech translation
- **2012: DNNs for vision supplanting traditional ML**
  - AlexNet for image classification
- **2014+: Rise of DNN accelerator research**
  - Examples: Neuflow, DianNao, etc.

# Publications at Architecture Conferences

---

- **MICRO, ISCA, HPCA, ASPLOS**

**# of Publications over the Years**



# So Many Neural Networks!

A mostly complete chart of

## Neural Networks

©2016 Fjodor van Veen - asimovinstitute.org

○ Backfed Input Cell

○ Input Cell

△ Noisy Input Cell

○ Hidden Cell

○ Probabilistic Hidden Cell

△ Spiking Hidden Cell

○ Output Cell

○ Match Input Output Cell

○ Recurrent Cell

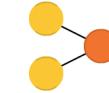
○ Memory Cell

△ Different Memory Cell

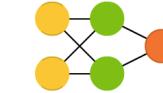
● Kernel

○ Convolution or Pool

Perceptron (P)



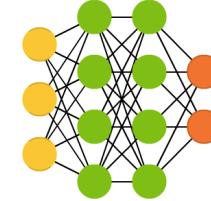
Feed Forward (FF)



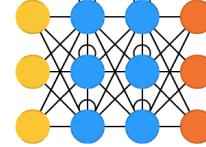
Radial Basis Network (RBF)



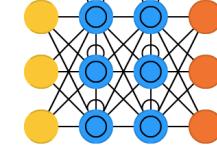
Deep Feed Forward (DFF)



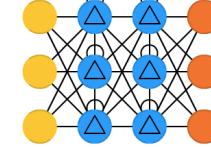
Recurrent Neural Network (RNN)



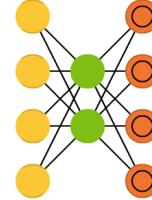
Long / Short Term Memory (LSTM)



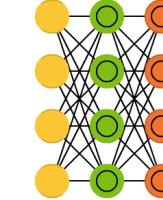
Gated Recurrent Unit (GRU)



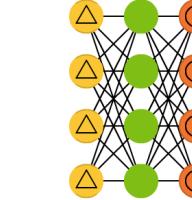
Auto Encoder (AE)



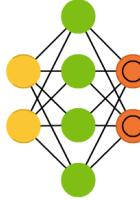
Variational AE (VAE)



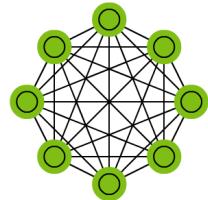
Denoising AE (DAE)



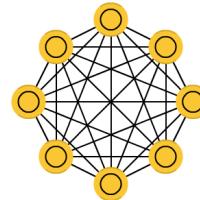
Sparse AE (SAE)



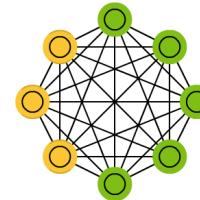
Markov Chain (MC)



Hopfield Network (HN)



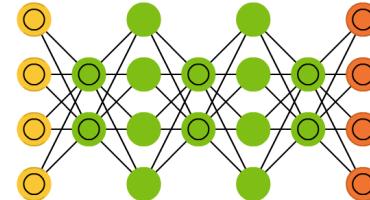
Boltzmann Machine (BM)



Restricted BM (RBM)

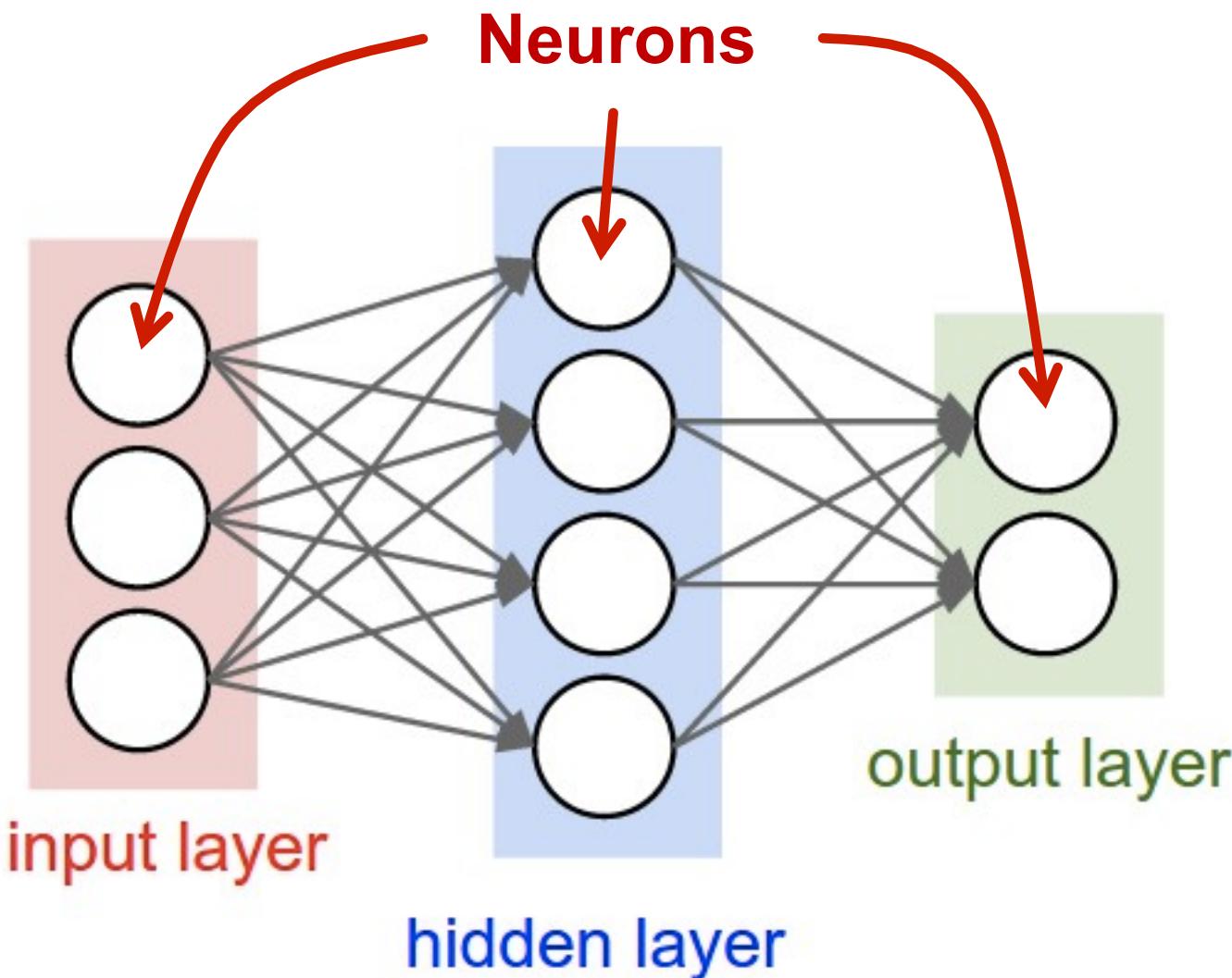


Deep Belief Network (DBN)



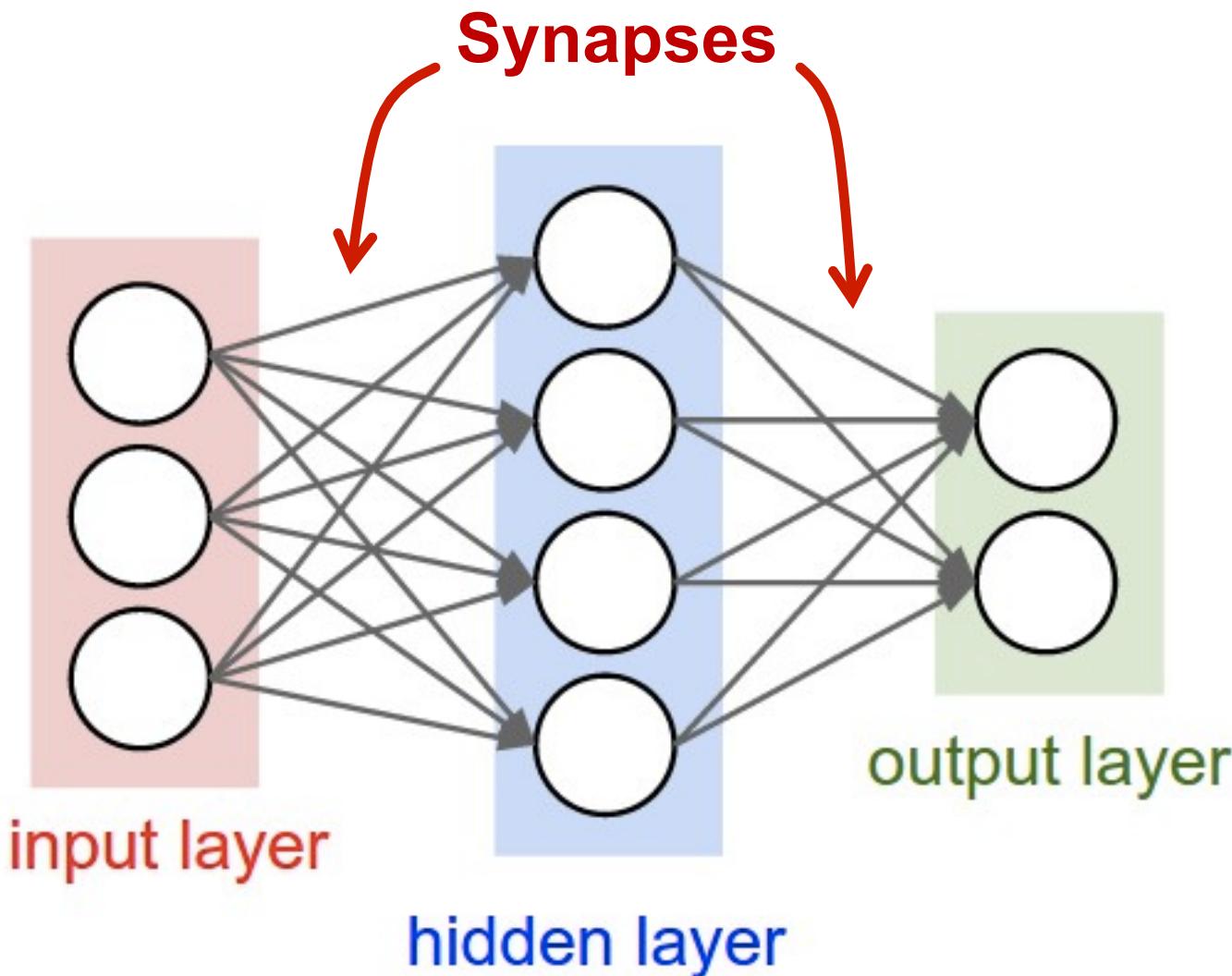
# DNN Terminology 101

---



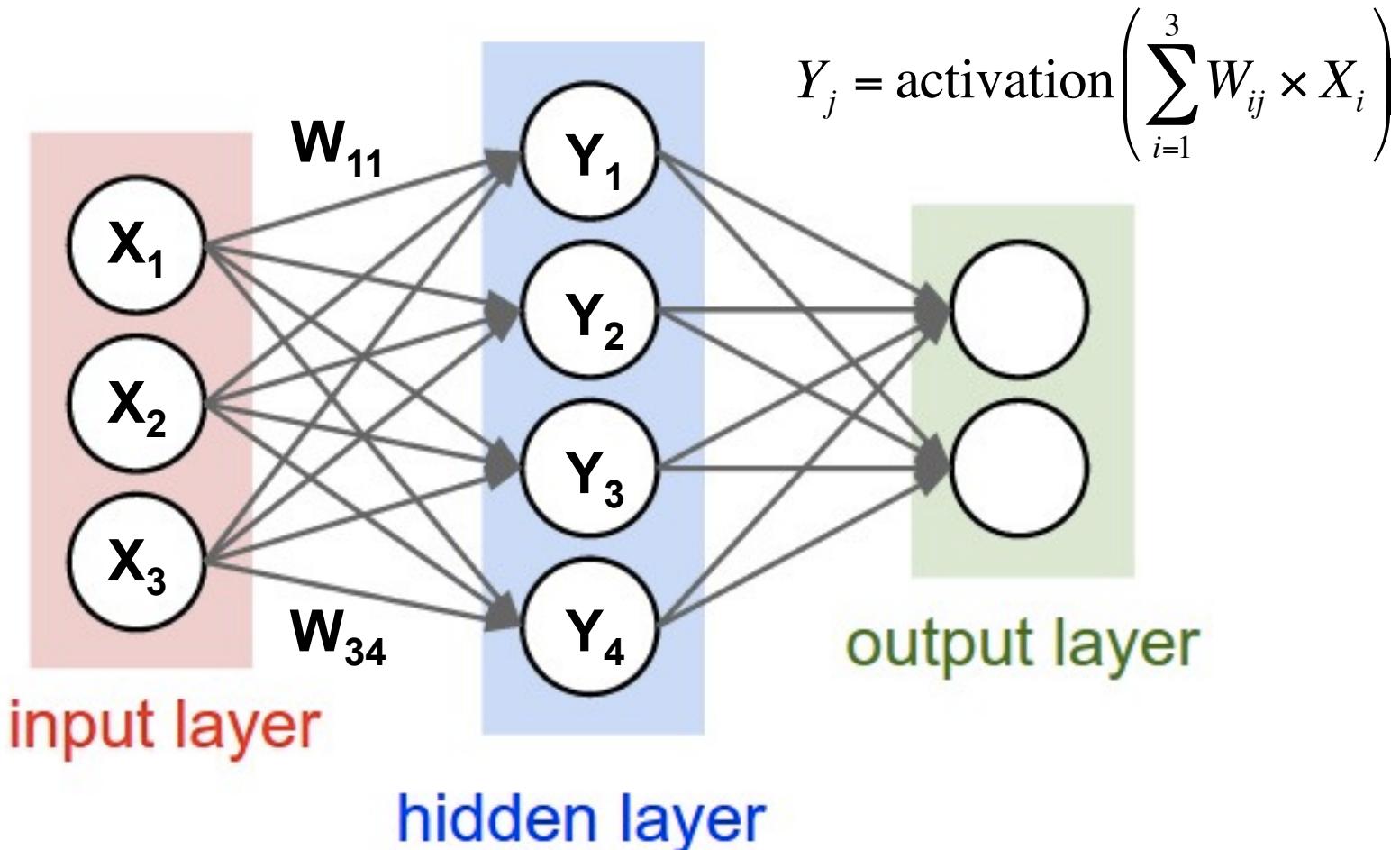
# DNN Terminology 101

---



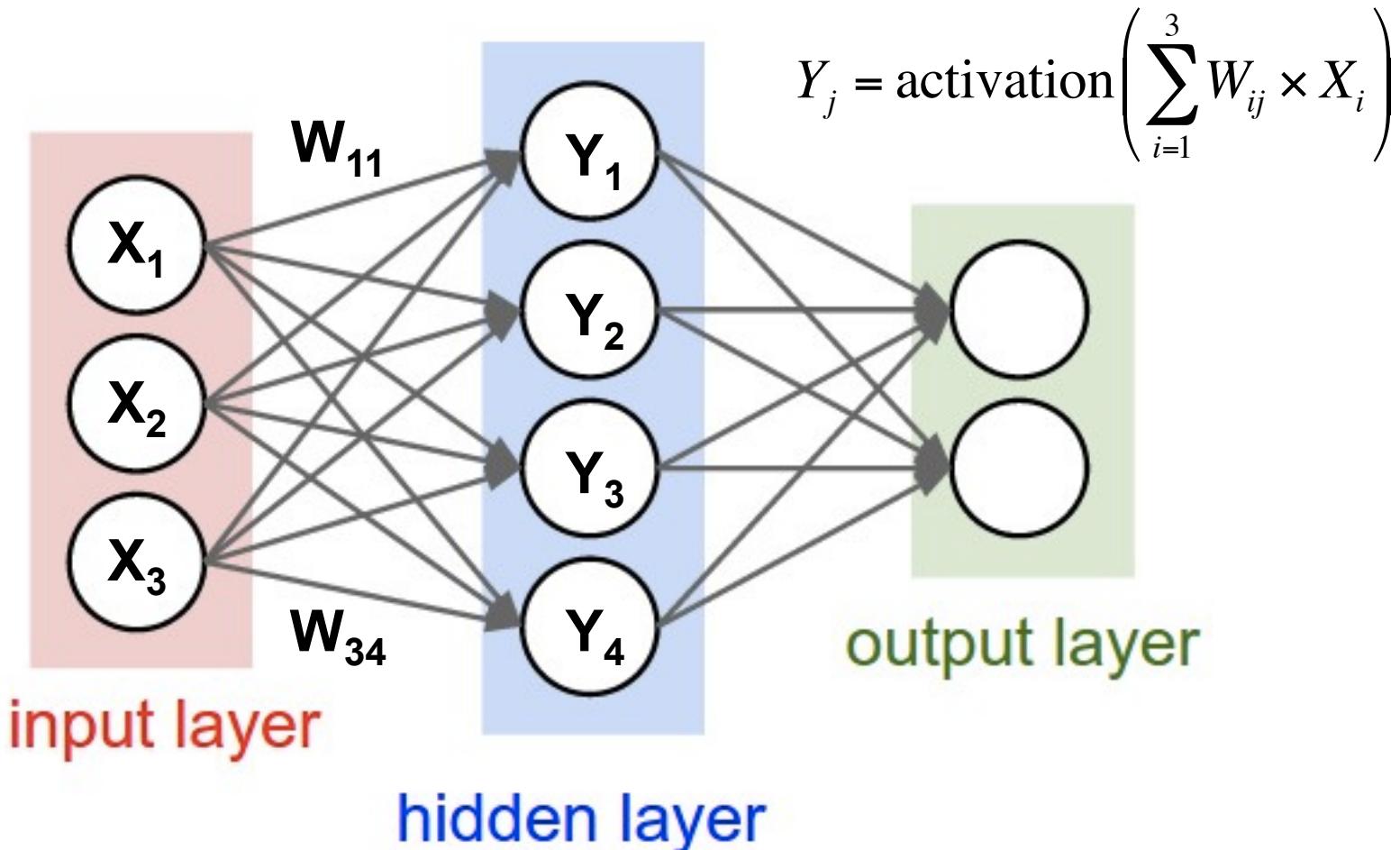
# DNN Terminology 101

Each **synapse** has a **weight** for neuron **activation**

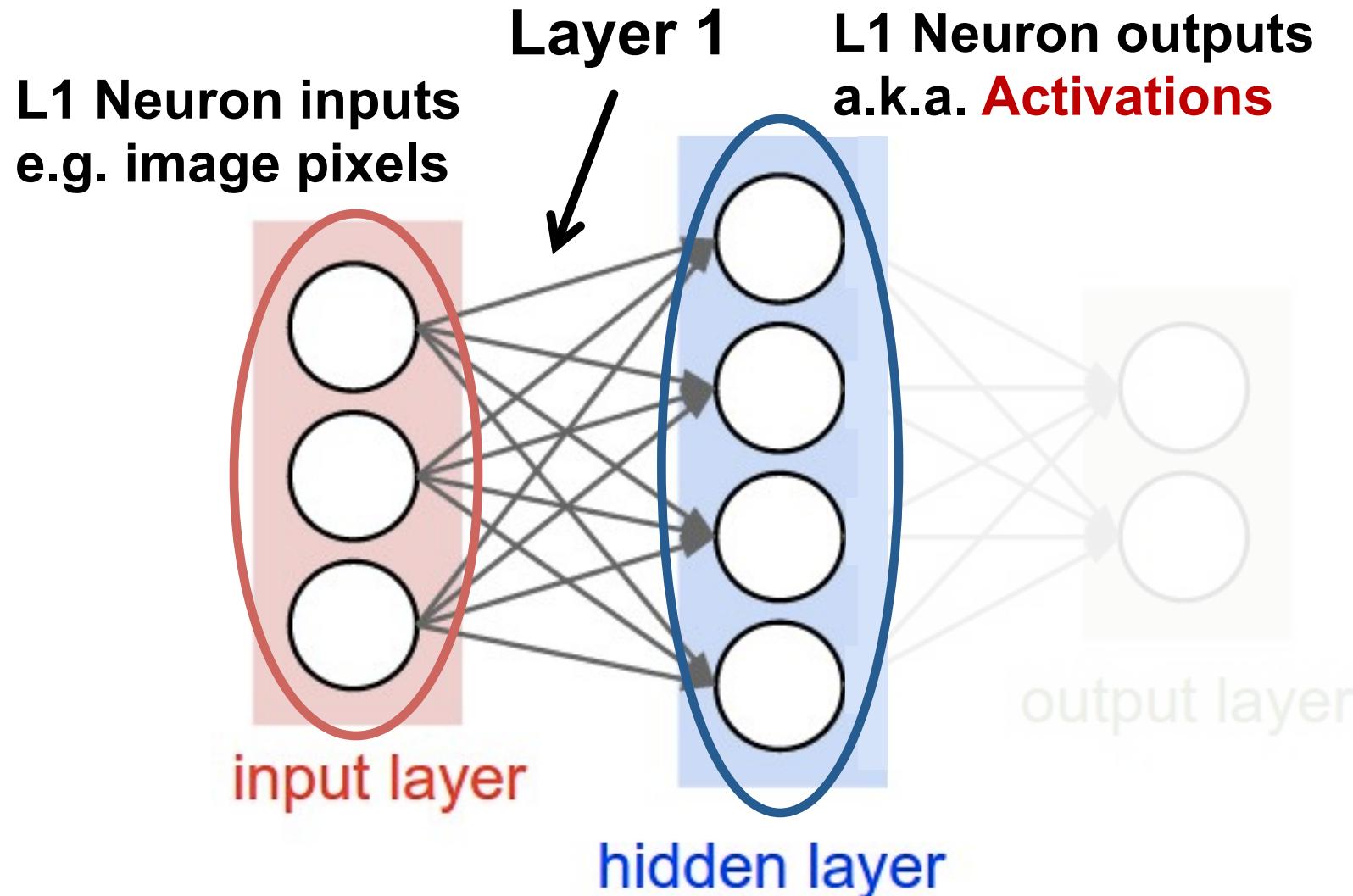


# DNN Terminology 101

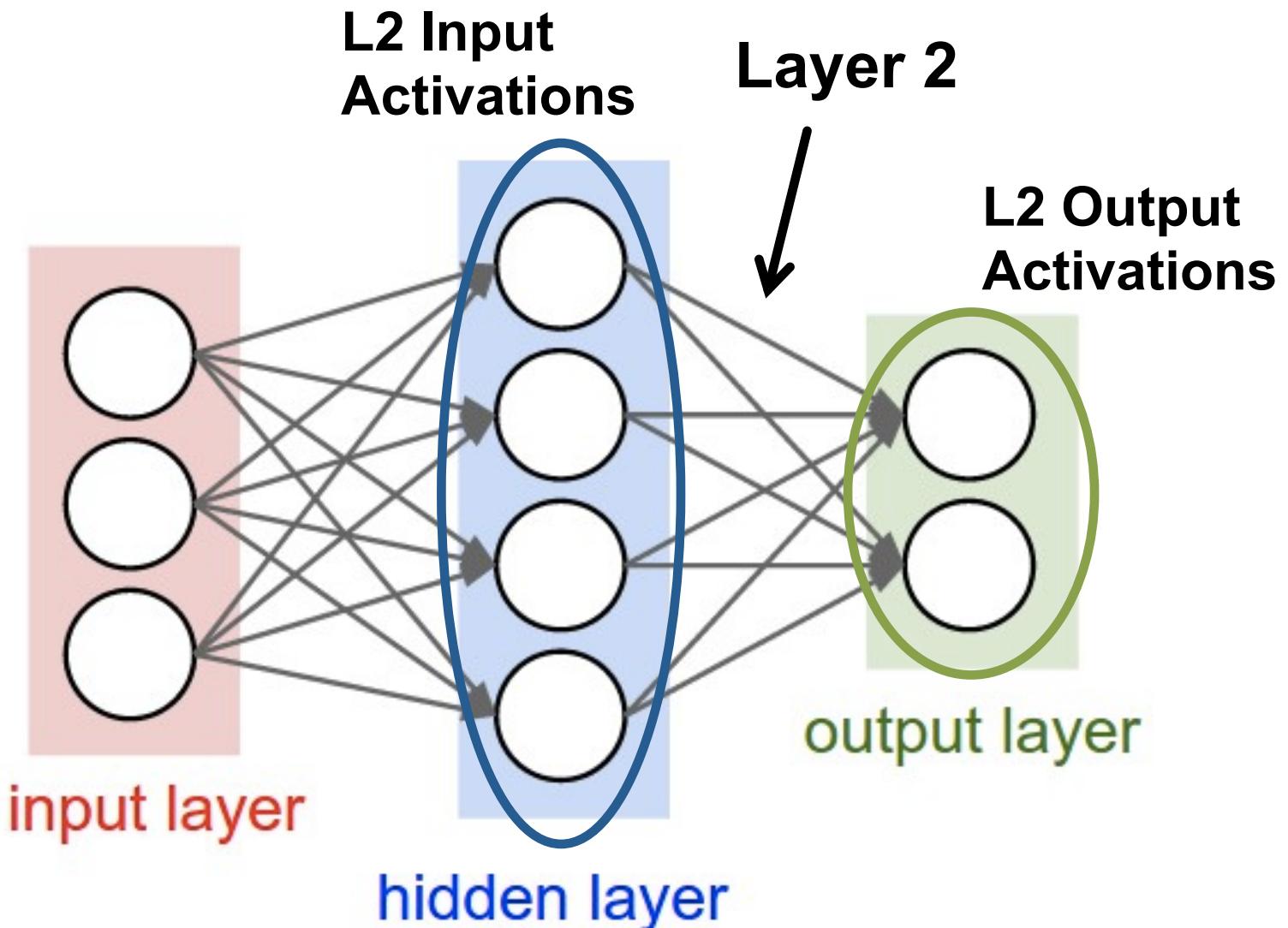
**Weight Sharing:** multiple synapses use the **same weight value**



# DNN Terminology 101

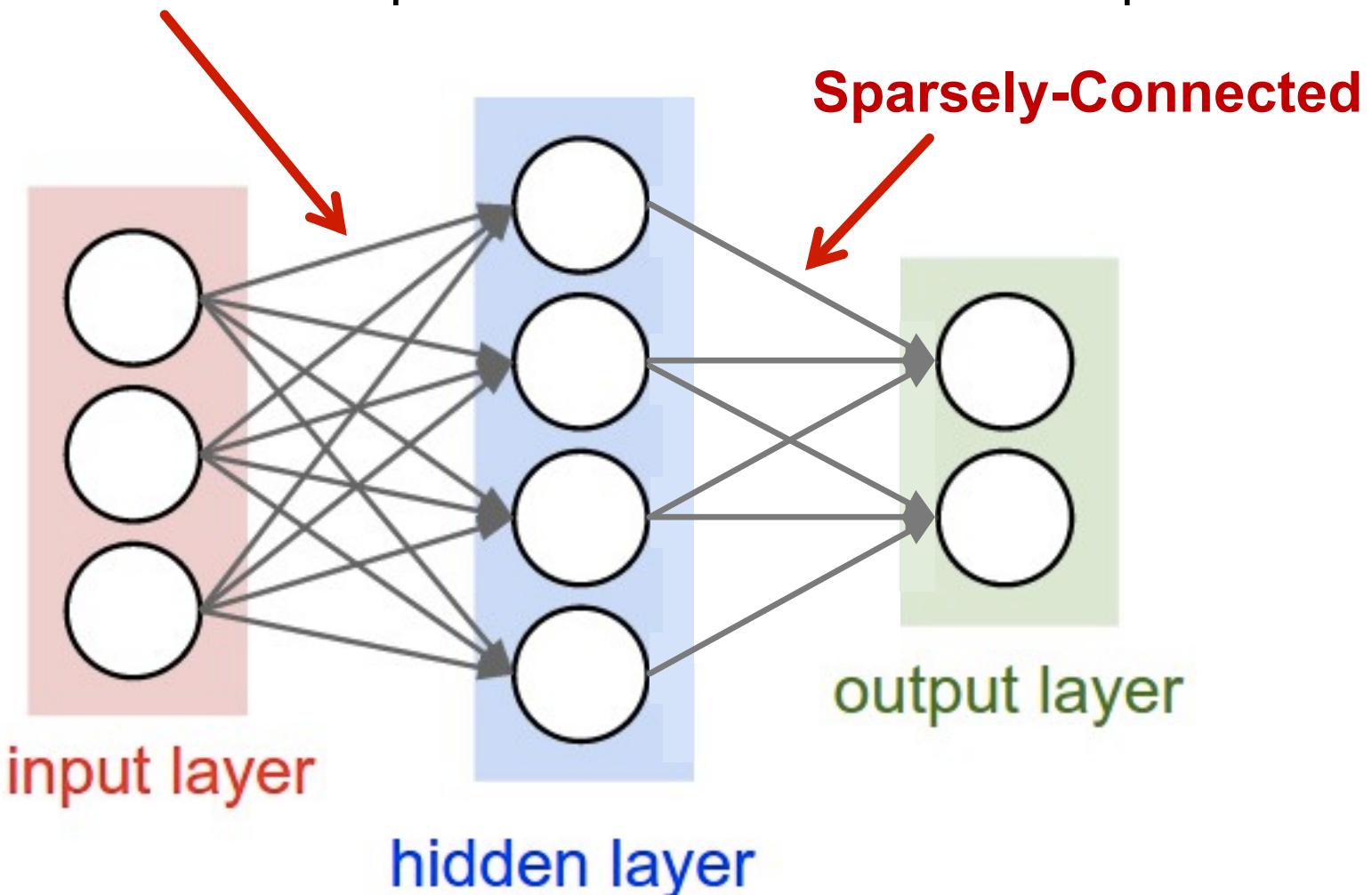


# DNN Terminology 101

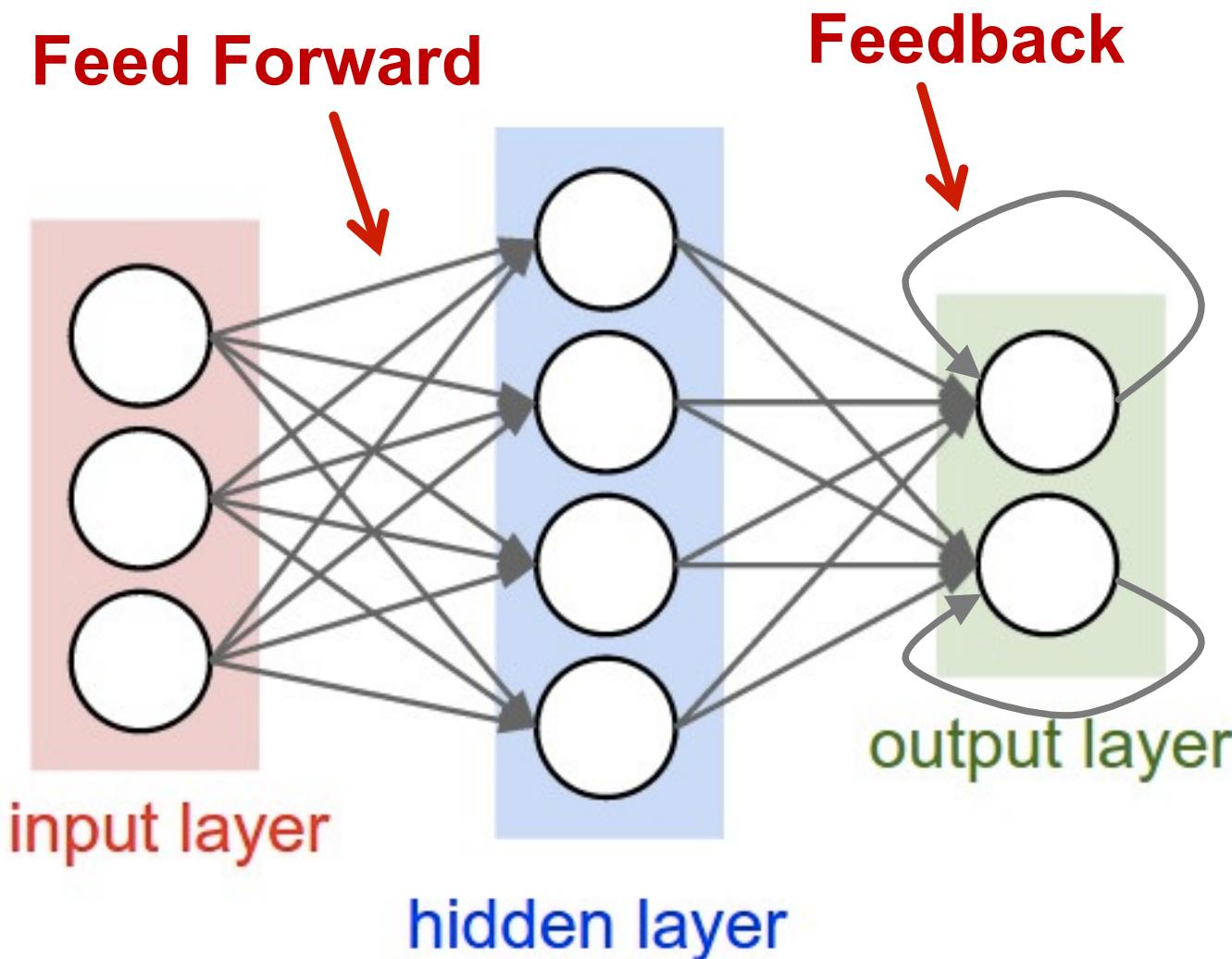


# DNN Terminology 101

**Fully-Connected**: all i/p neurons connected to all o/p neurons



# DNN Terminology 101



# Popular Types of DNNs

---

- **Fully-Connected NN**
  - feed forward, a.k.a. multilayer perceptron (MLP)
- **Convolutional NN (CNN)**
  - feed forward, sparsely-connected w/ weight sharing
- **Recurrent NN (RNN)**
  - feedback
- **Long Short-Term Memory (LSTM)**
  - feedback + storage

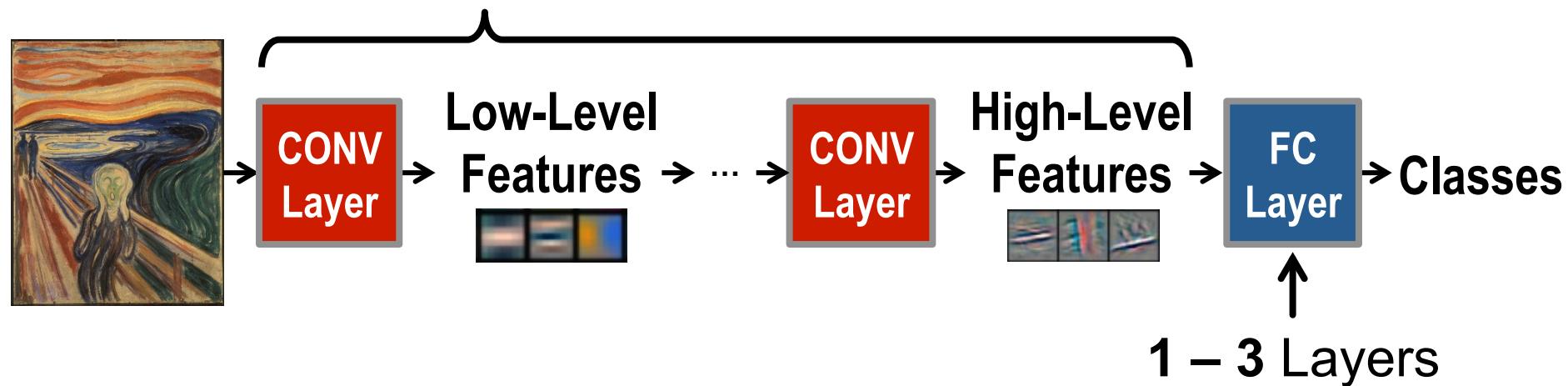
# Inference vs. Training

---

- **Training:** Determine weights
  - **Supervised:**
    - Training set has inputs and outputs, i.e., labeled
  - **Unsupervised:**
    - Training set is unlabeled
  - **Semi-supervised:**
    - Training set is partially labeled
  - **Reinforcement:**
    - Output assessed via rewards and punishments
- **Inference:** Apply weights to determine output

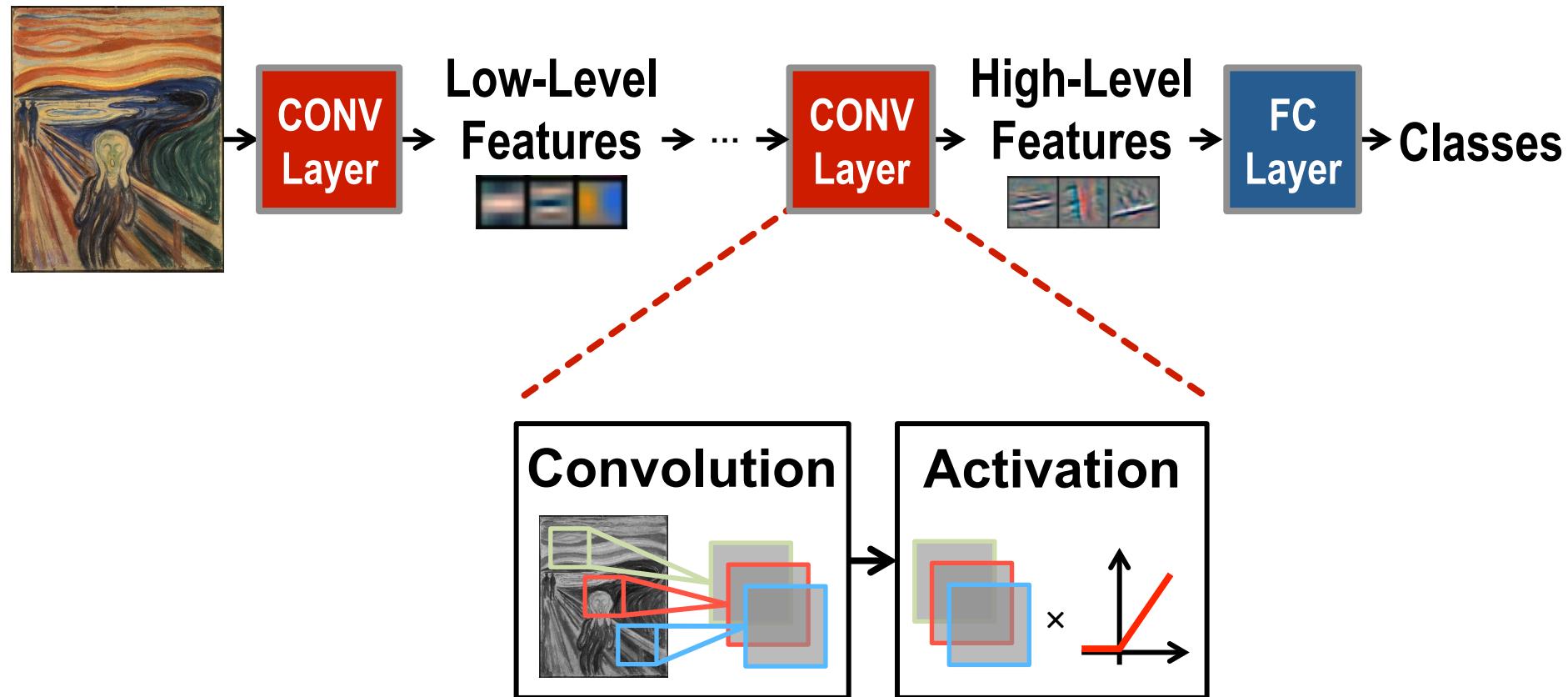
# Deep Convolutional Neural Networks

Modern Deep CNN: 5 – 1000 Layers



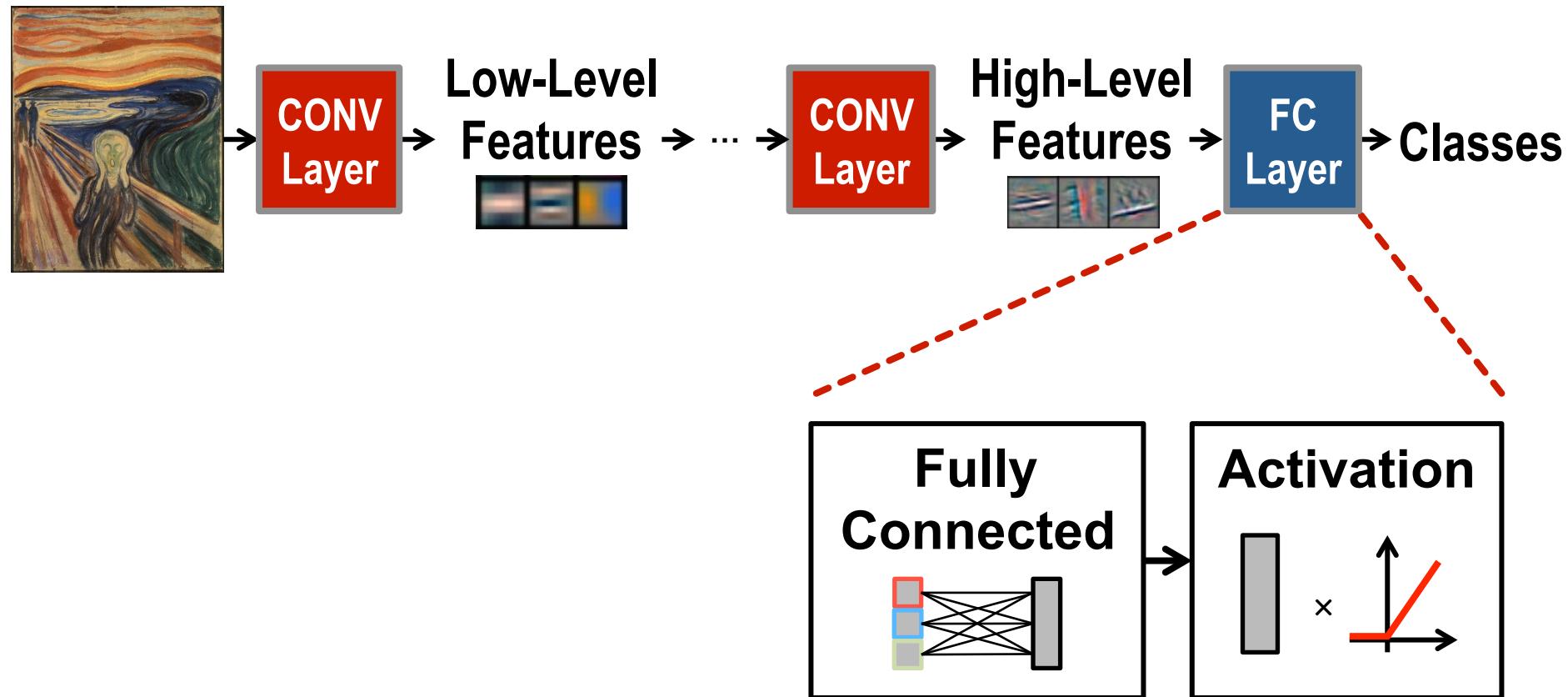
# Deep Convolutional Neural Networks

---



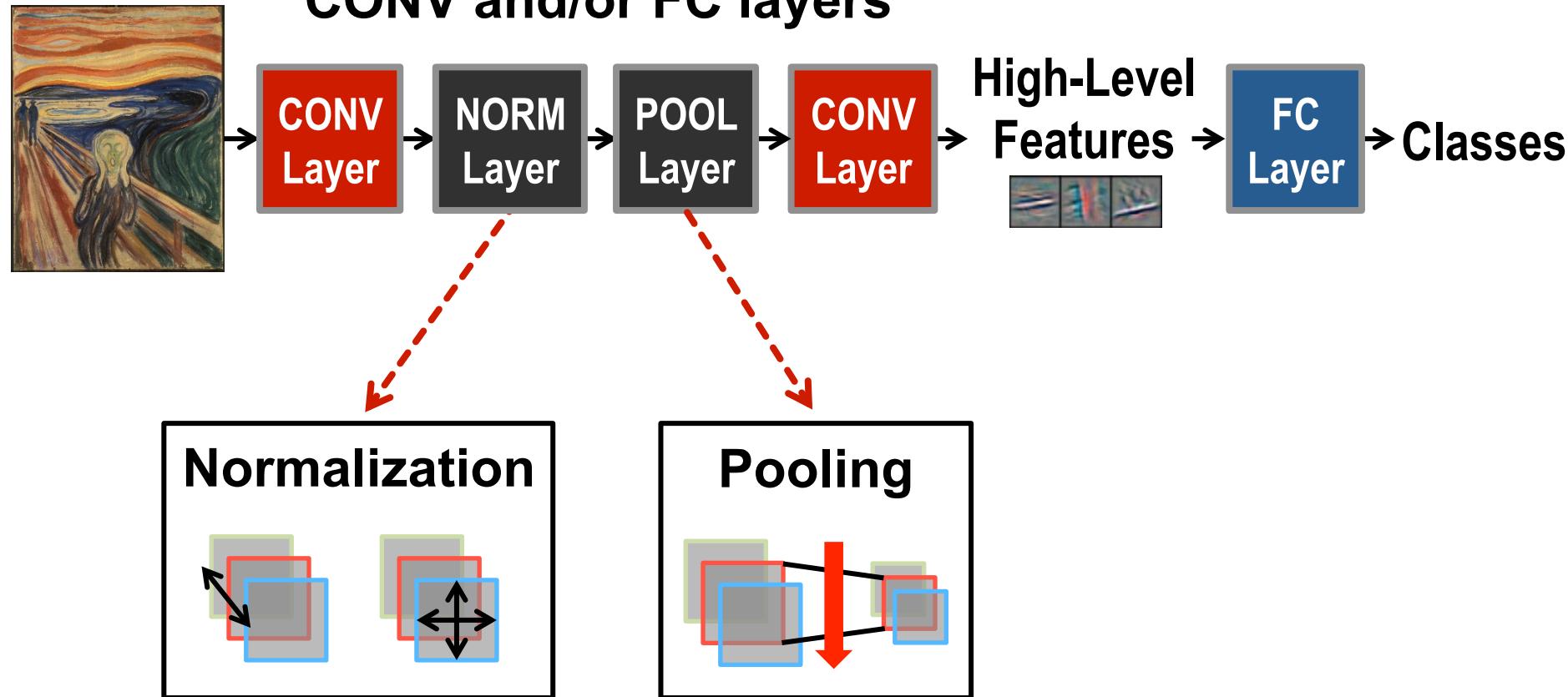
# Deep Convolutional Neural Networks

---

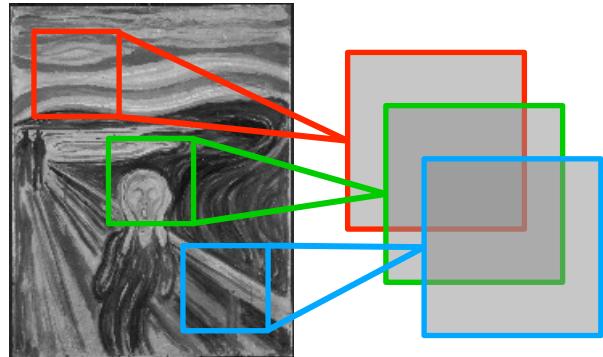
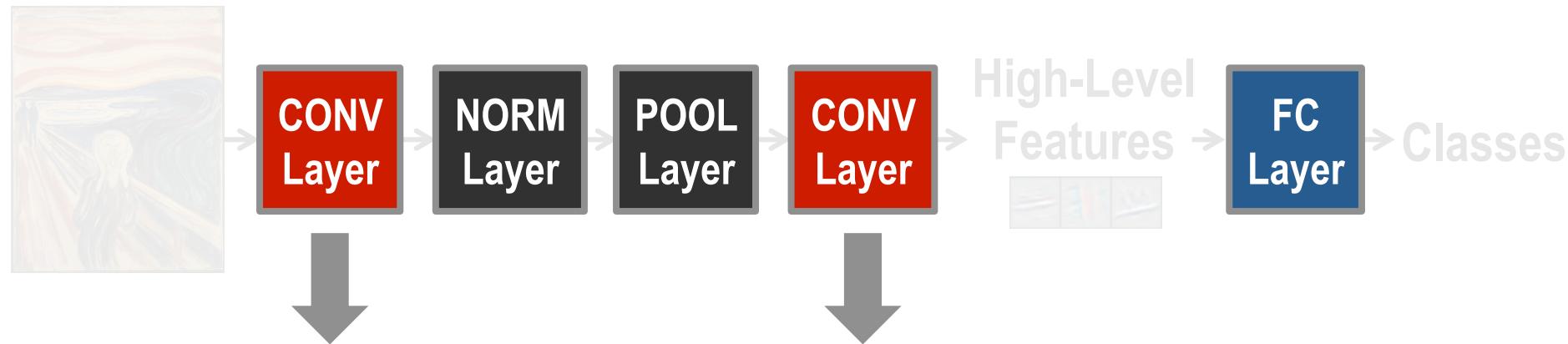


# Deep Convolutional Neural Networks

Optional layers in between CONV and/or FC layers



# Deep Convolutional Neural Networks



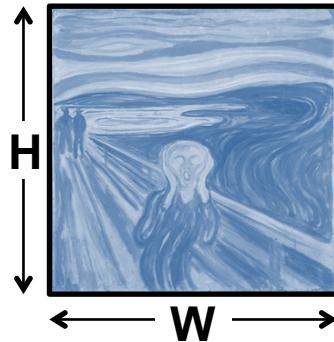
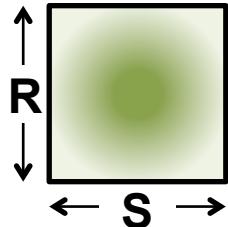
**Convolutions** account for more than 90% of overall computation, dominating **runtime** and **energy consumption**

# Convolution (CONV) Layer

---

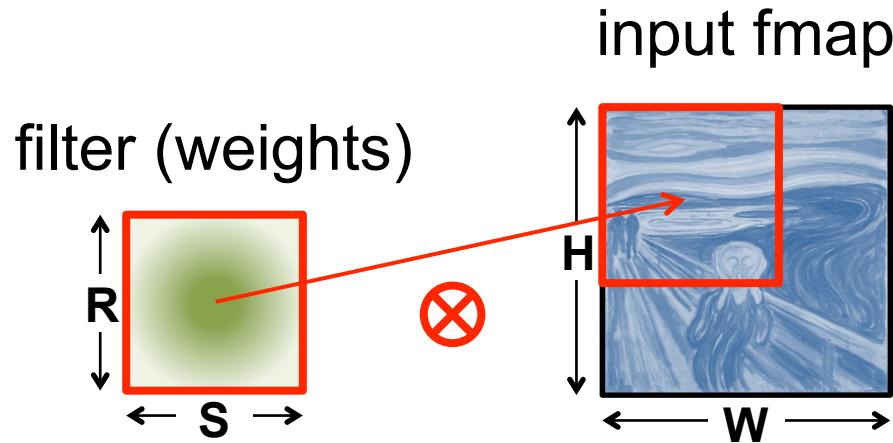
a plane of input activations  
a.k.a. **input feature map (fmap)**

filter (weights)



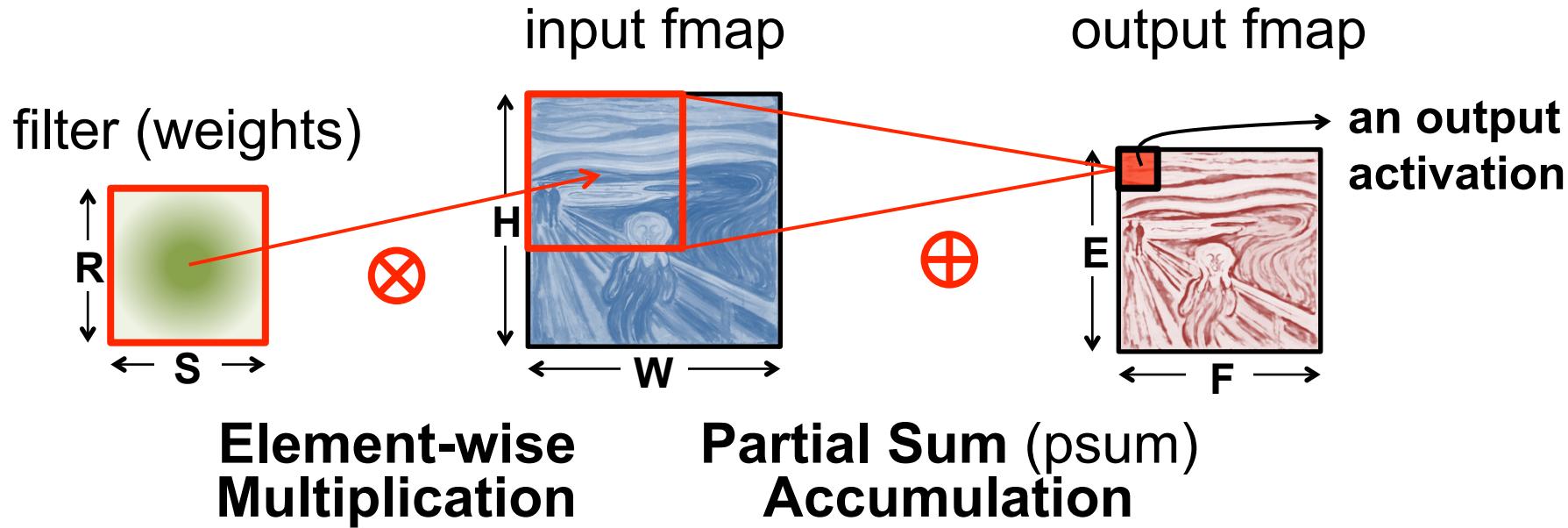
# Convolution (CONV) Layer

---

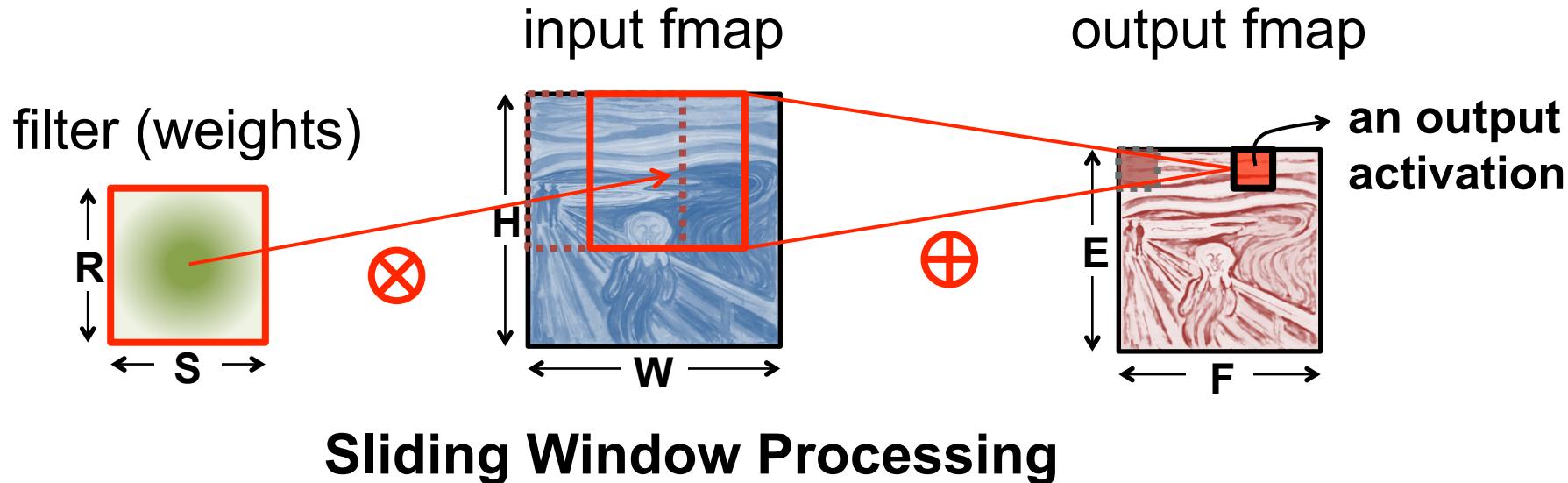


**Element-wise  
Multiplication**

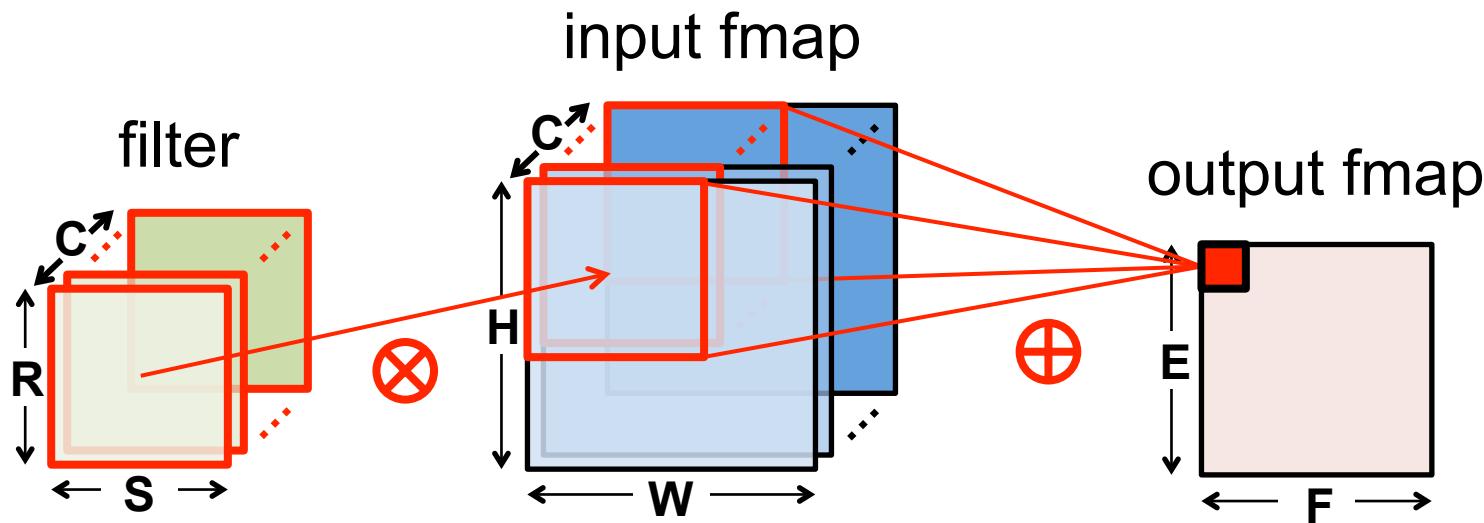
# Convolution (CONV) Layer



# Convolution (CONV) Layer

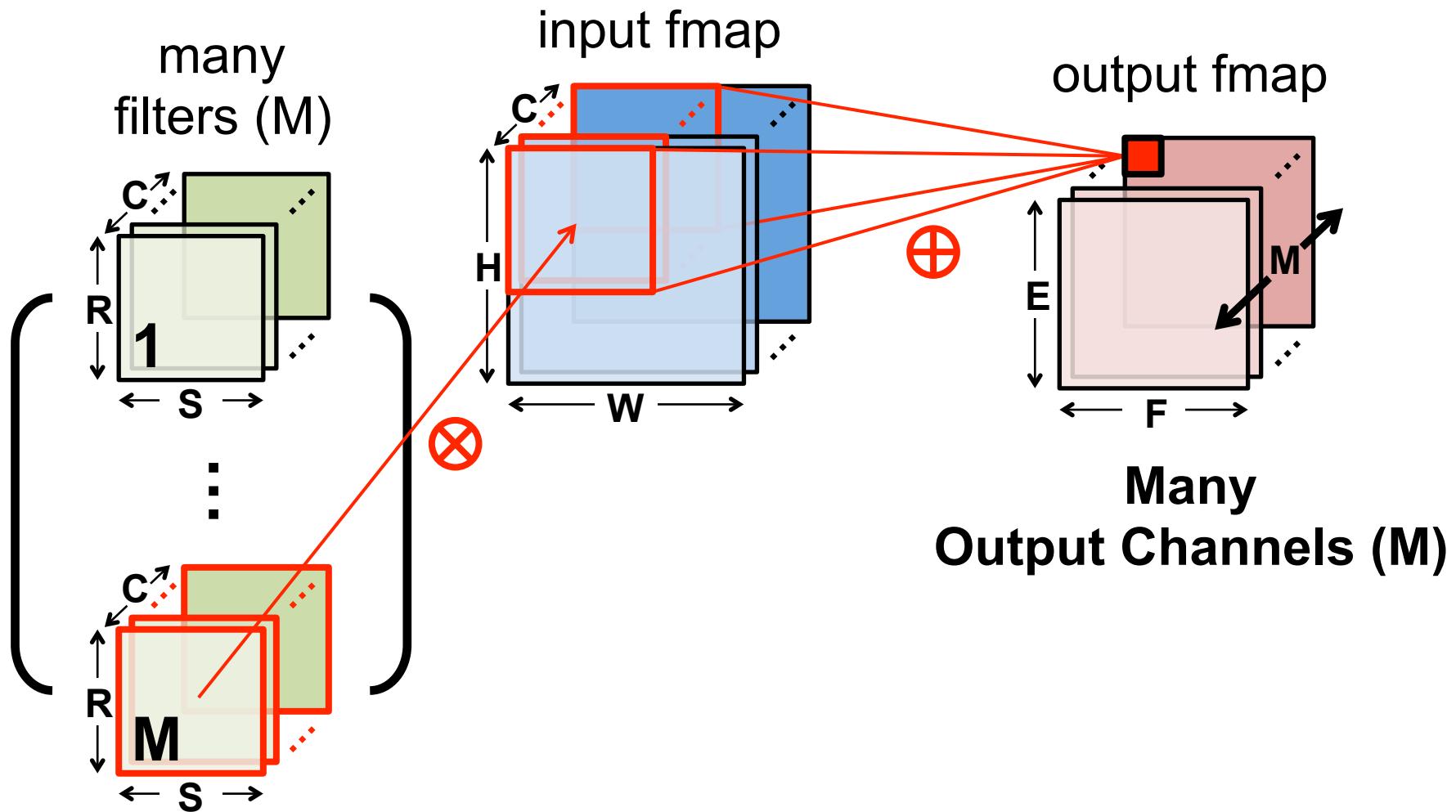


# Convolution (CONV) Layer

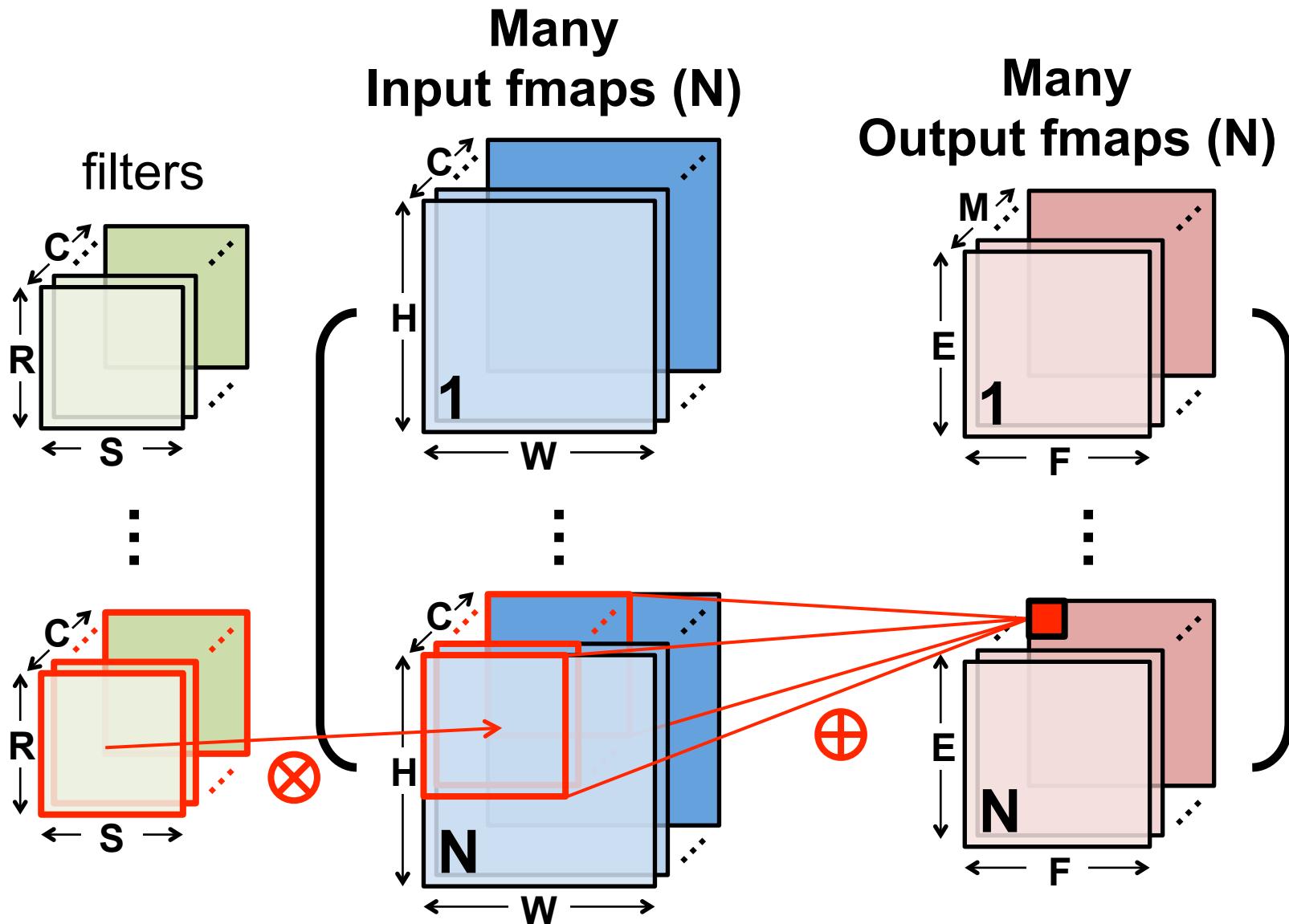


**Many Input Channels (C)**

# Convolution (CONV) Layer



# Convolution (CONV) Layer



# CNN Decoder Ring

---

- **N** – Number of **input fmaps/output fmaps** (batch size)
- **C** – Number of 2-D **input fmaps /filters** (channels)
- **H** – Height of **input fmap** (activations)
- **W** – Width of **input fmap** (activations)
- **R** – Height of 2-D **filter (weights)**
- **S** – Width of 2-D **filter (weights)**
- **M** – Number of 2-D **output fmaps** (channels)
- **E** – Height of **output fmap** (activations)
- **F** – Width of **output fmap** (activations)

# CONV Layer Tensor Computation

Output fmmaps (O)

Input fmmaps (I)

Biases (B)

Filter weights (W)

$$\mathbf{O}[n][m][x][y] = \text{Activation}(\mathbf{B}[m] + \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} \sum_{k=0}^{C-1} \mathbf{I}[n][k][Ux+i][Uy+j] \times \mathbf{W}[m][k][i][j]),$$

$$0 \leq n < N, 0 \leq m < M, 0 \leq y < E, 0 \leq x < F,$$

$$E = (H - R + U)/U, F = (W - S + U)/U.$$

Shape Parameter	Description
$N$	fmap batch size
$M$	# of filters / # of output fmap channels
$C$	# of input fmap/filter channels
$H/W$	input fmap height/width
$R/S$	filter height/width
$E/F$	output fmap height/width
$U$	convolution stride

# CONV Layer Implementation

## Naïve 7-layer for-loop implementation:

```

for (n=0; n<N; n++) {
  for (m=0; m<M; m++) {
    for (x=0; x<F; x++) {
      for (y=0; y<E; y++) {
        o[n][m][x][y] = B[m];
        for (i=0; i<R; i++) {
          for (j=0; j<S; j++) {
            for (k=0; k<C; k++) {
              o[n][m][x][y] += I[n][k][Ux+i][Uy+j] * W[m][k][i][j];
            }
          }
        }
        o[n][m][x][y] = Activation(o[n][m][x][y]);
      }
    }
  }
}

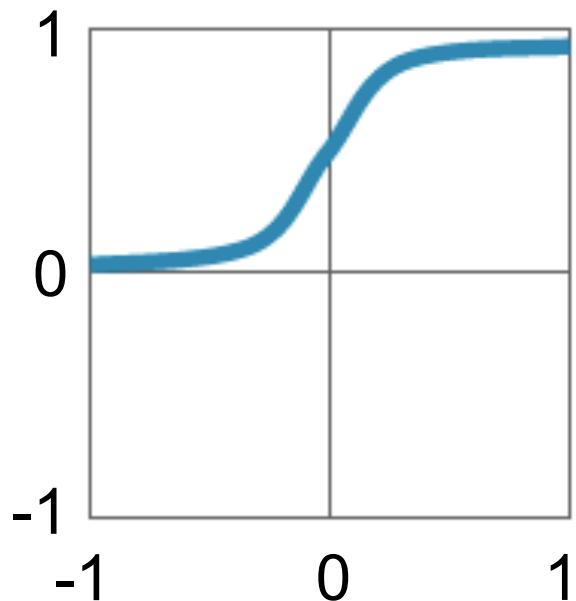
```

} for each output fmap value

# Traditional Activation Functions

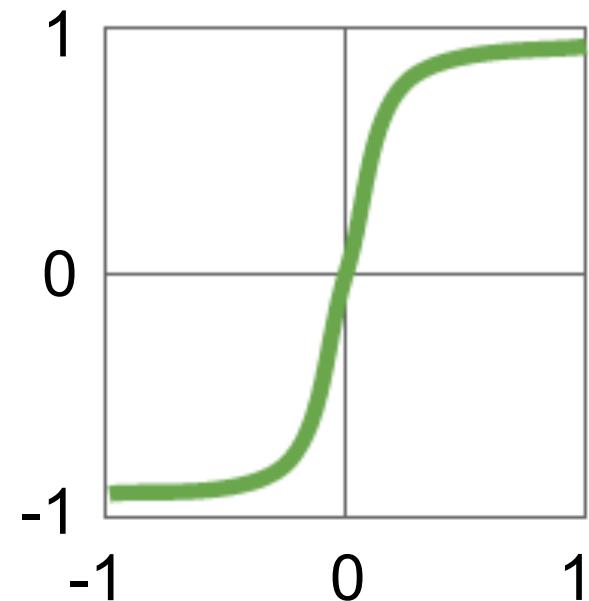
---

**Sigmoid**



$$y = 1 / (1 + e^{-x})$$

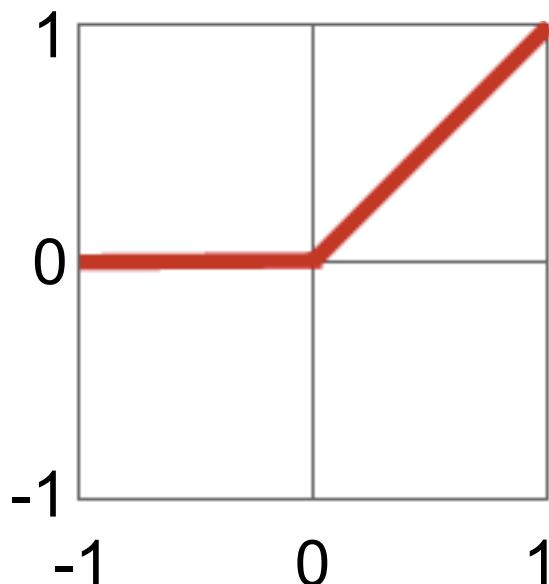
**Hyperbolic Tangent**



$$y = (e^x - e^{-x}) / (e^x + e^{-x})$$

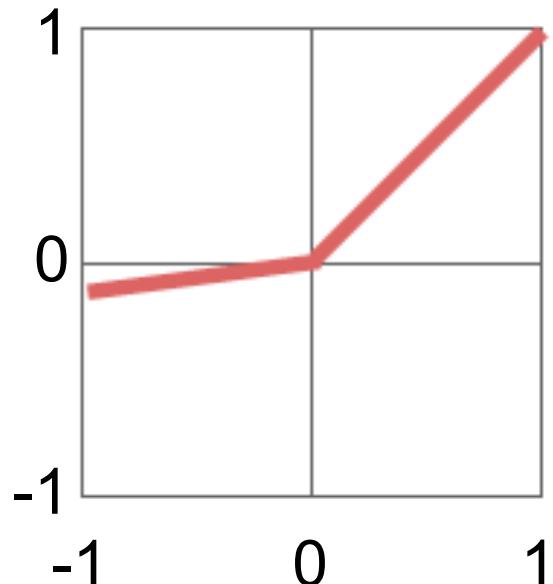
# Modern Activation Functions

Rectified Linear Unit  
(ReLU)



$$y = \max(0, x)$$

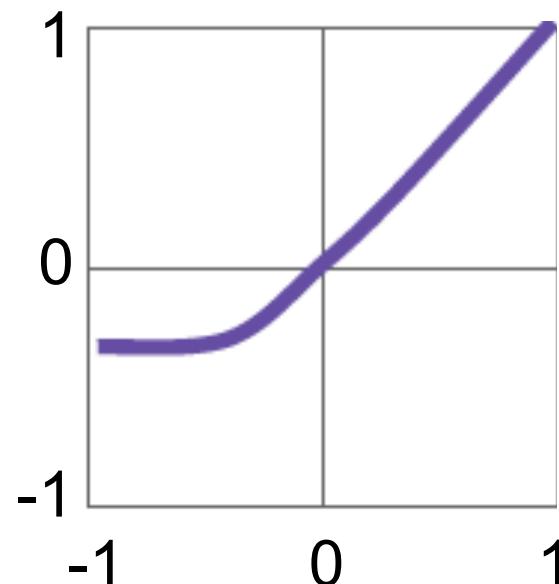
Leaky ReLU



$$y = \max(\alpha x, x)$$

$\alpha = \text{small const. (e.g. 0.1)}$

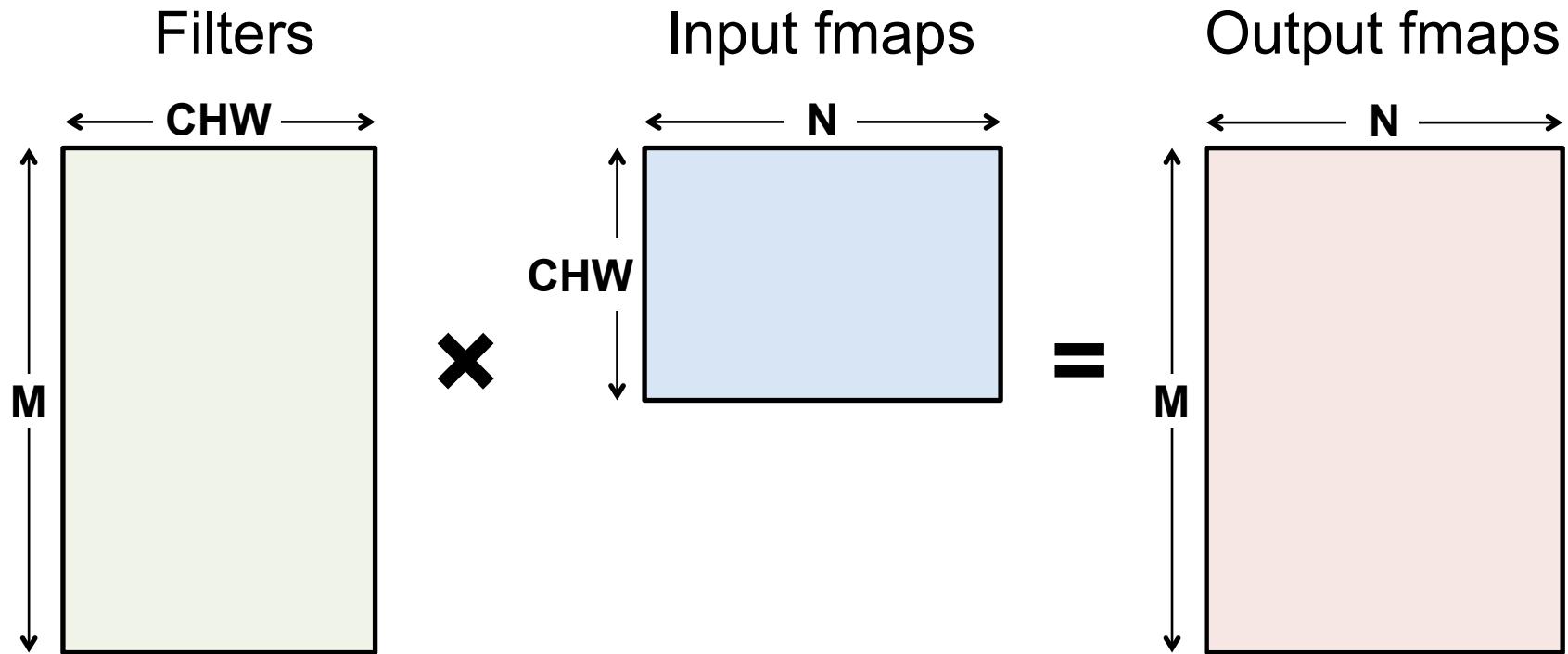
Exponential LU



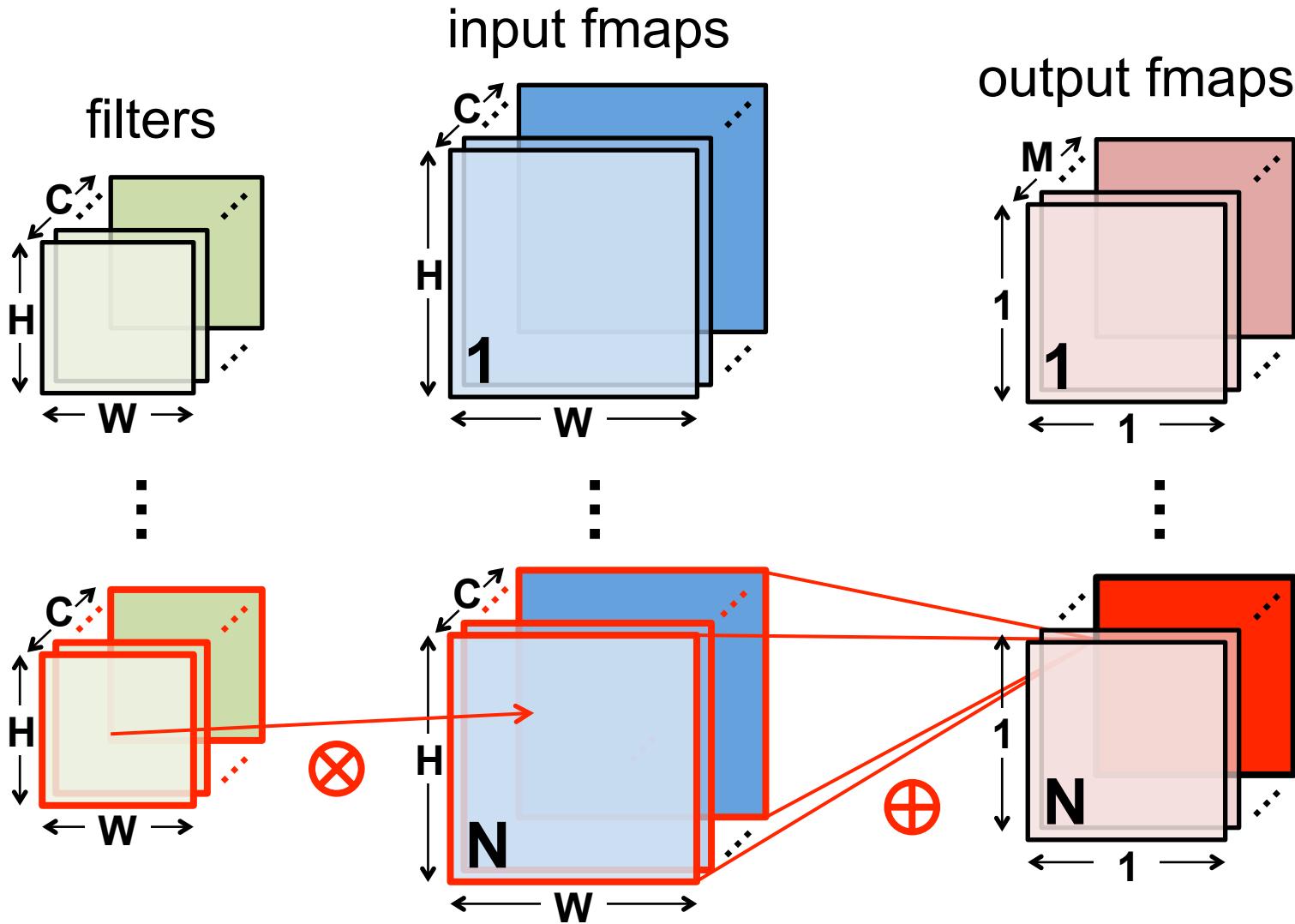
$$y = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$$

# Fully-Connected (FC) Layer

- Height and width of output fmmaps are 1 ( $E = F = 1$ )
- Filters as large as input fmmaps ( $R = H, S = W$ )
- Implementation: **Matrix Multiplication**

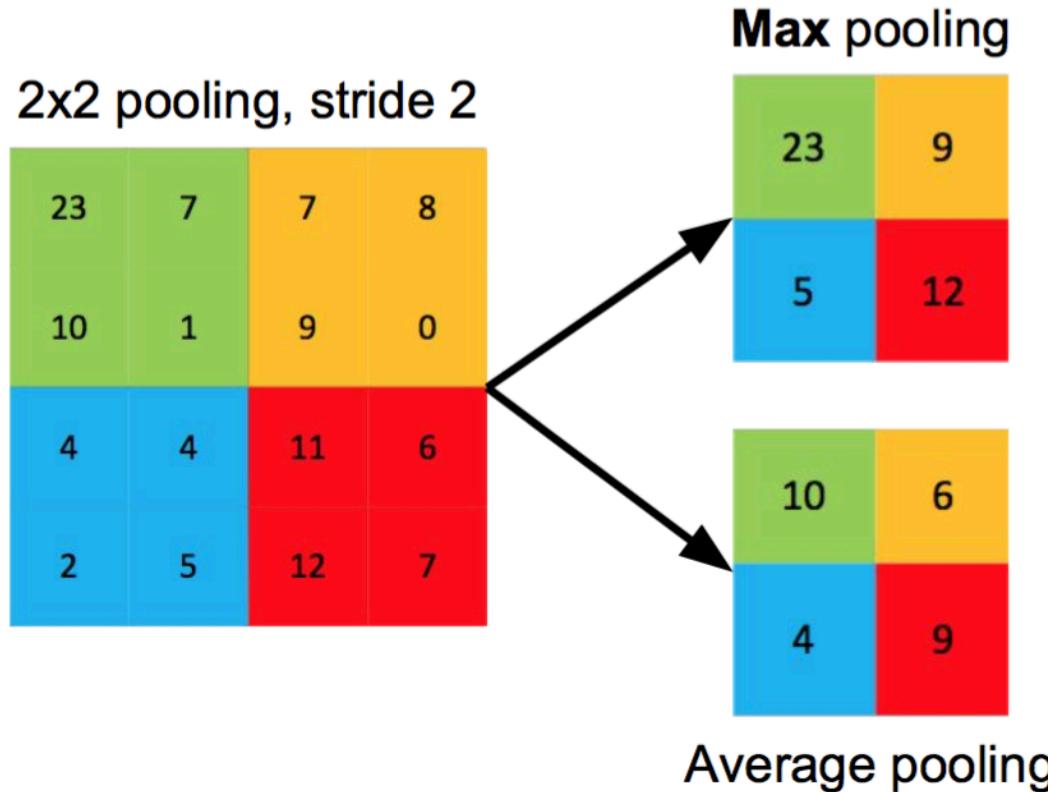


# FC Layer – from CONV Layer POV



# Pooling (POOL) Layer

- Reduce resolution of each channel independently
- Overlapping or non-overlapping → depending on stride



Increases translation-invariance and noise-resilience

# POOL Layer Implementation

## Naïve 6-layer for-loop max-pooling implementation:

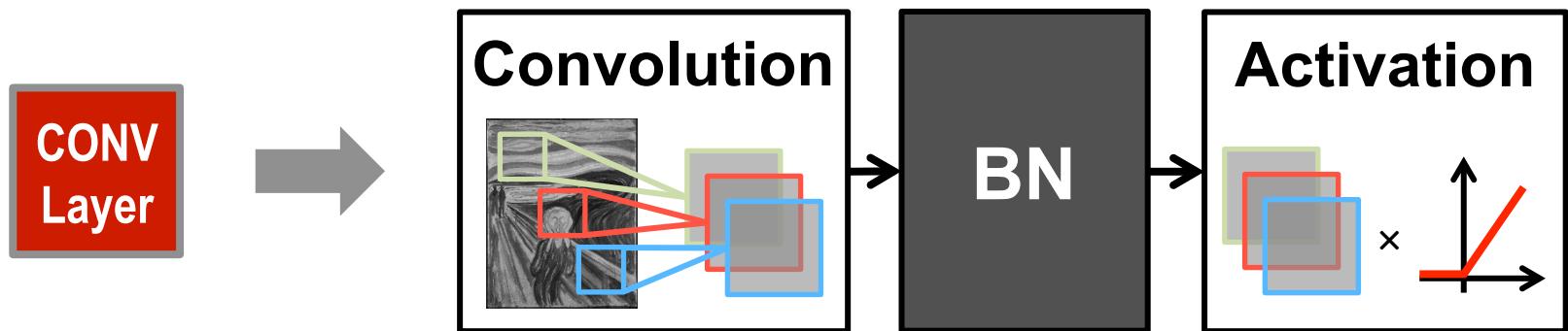
```
for (n=0; n<N; n++) {  
    for (m=0; m<M; m++) {  
        for (x=0; x<F; x++) {  
            for (y=0; y<E; y++) {  
                max = -Inf;  
                for (i=0; i<R; i++) {  
                    for (j=0; j<S; j++) {  
                        if (I[n][m][Ux+i][Uy+j] > max) {  
                            max = I[n][m][Ux+i][Uy+j];  
                        }  
                    }  
                }  
                o[n][m][x][y] = max;  
            }  
        }  
    }  
}
```

} for each pooled value

} find the max with in a window

# Normalization (NORM) Layer

- **Batch Normalization (BN)**
  - Normalize activations towards  $\text{mean}=0$  and  $\text{std. dev.}=1$  based on the statistics of the training dataset
  - put **in between** CONV/FC and **Activation function**



Believed to be key to getting high accuracy and faster training on very deep neural networks.

# BN Layer Implementation

---

- The normalized value is further scaled and shifted, the parameters of which are learned from training

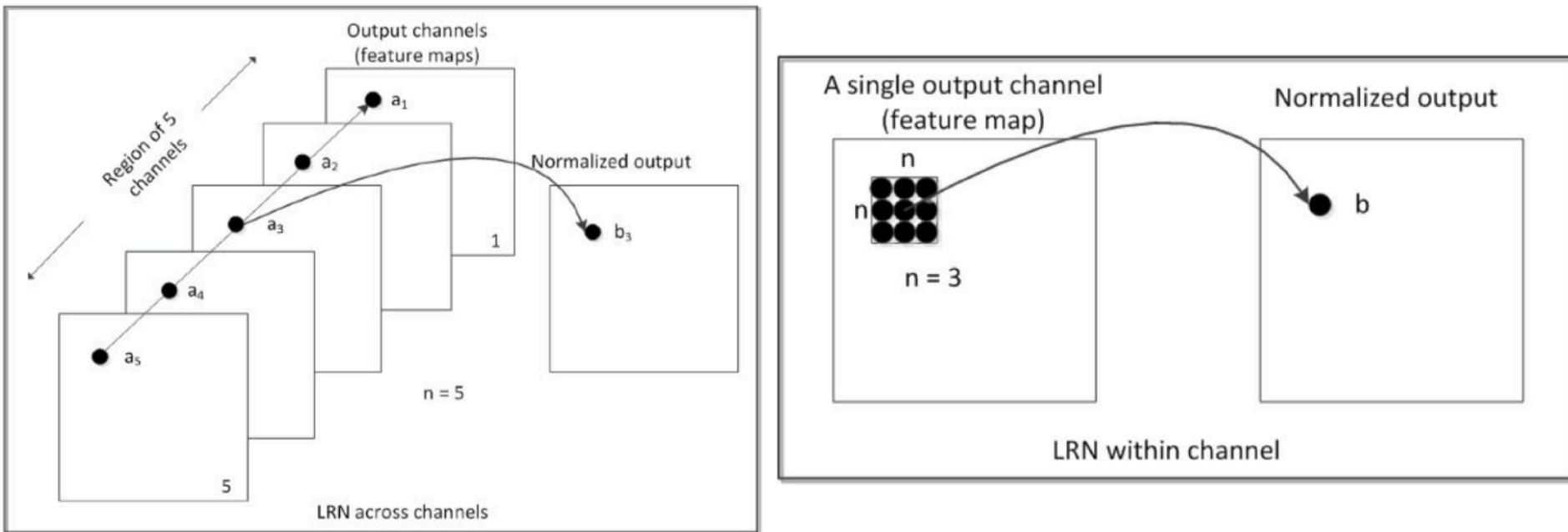
$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \gamma + \beta$$

Annotations for the BN formula:

- data mean** (red) points to  $\mu$
- data std. dev.** (red) points to  $\sigma^2$
- learned scale factor** (blue) points to  $\gamma$
- learned shift factor** (blue) points to  $\beta$
- small const. to avoid numerical problems** (gray) points to  $\epsilon$

# Normalization (NORM) Layer

- **Local Response Normalization (LRN)**
  - Tries to mimic the inhibition scheme in the brain



Now deprecated!

# Relevant Components for Tutorial

---

- **Typical operations that we will discuss:**
  - Convolution (CONV)
  - Fully-Connected (FC)
  - Max Pooling
  - ReLU

# **Survey of DNN Development Resources**

## **ISCA Tutorial (2017)**

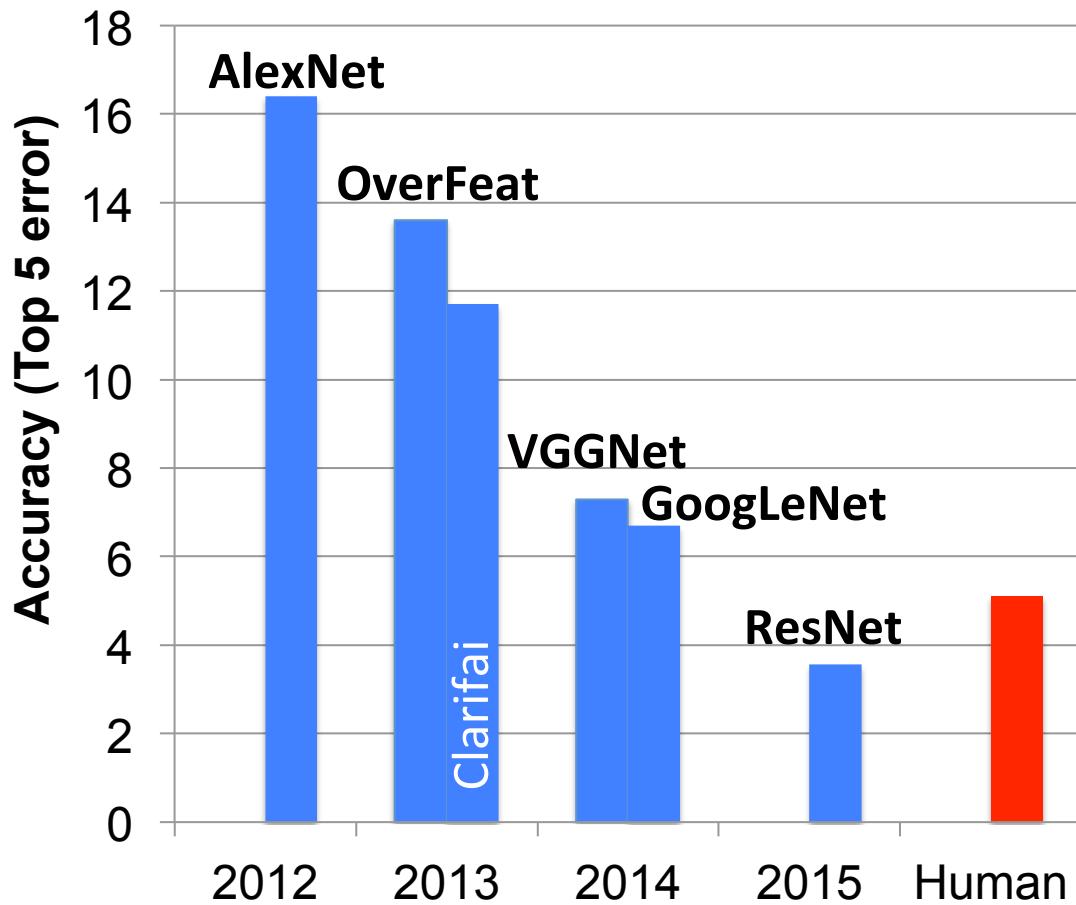
Website: <http://eyeriss.mit.edu/tutorial.html>

Joel Emer, Vivienne Sze, Yu-Hsin Chen

# Popular DNNs

- **LeNet (1998)**
- **AlexNet (2012)**
- **OverFeat (2013)**
- **VGGNet (2014)**
- **GoogleNet (2014)**
- **ResNet (2015)**

**ImageNet: Large Scale Visual Recognition Challenge (ILSVRC)**



# LeNet-5

CONV Layers: 2

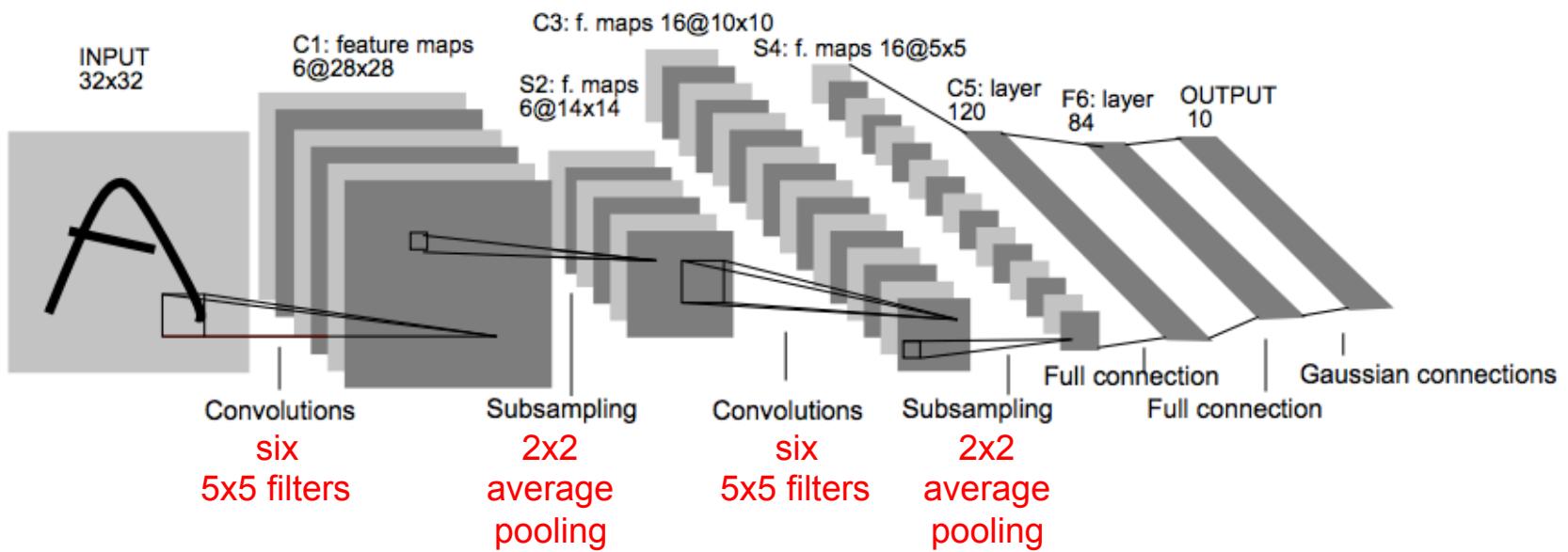
Fully Connected Layers: 2

Weights: 60k

MACs: 341k

Sigmoid used for non-linearity

**Digit Classification!**



# AlexNet

CONV Layers: 5

ILSCVR12 Winner

Fully Connected Layers: 3

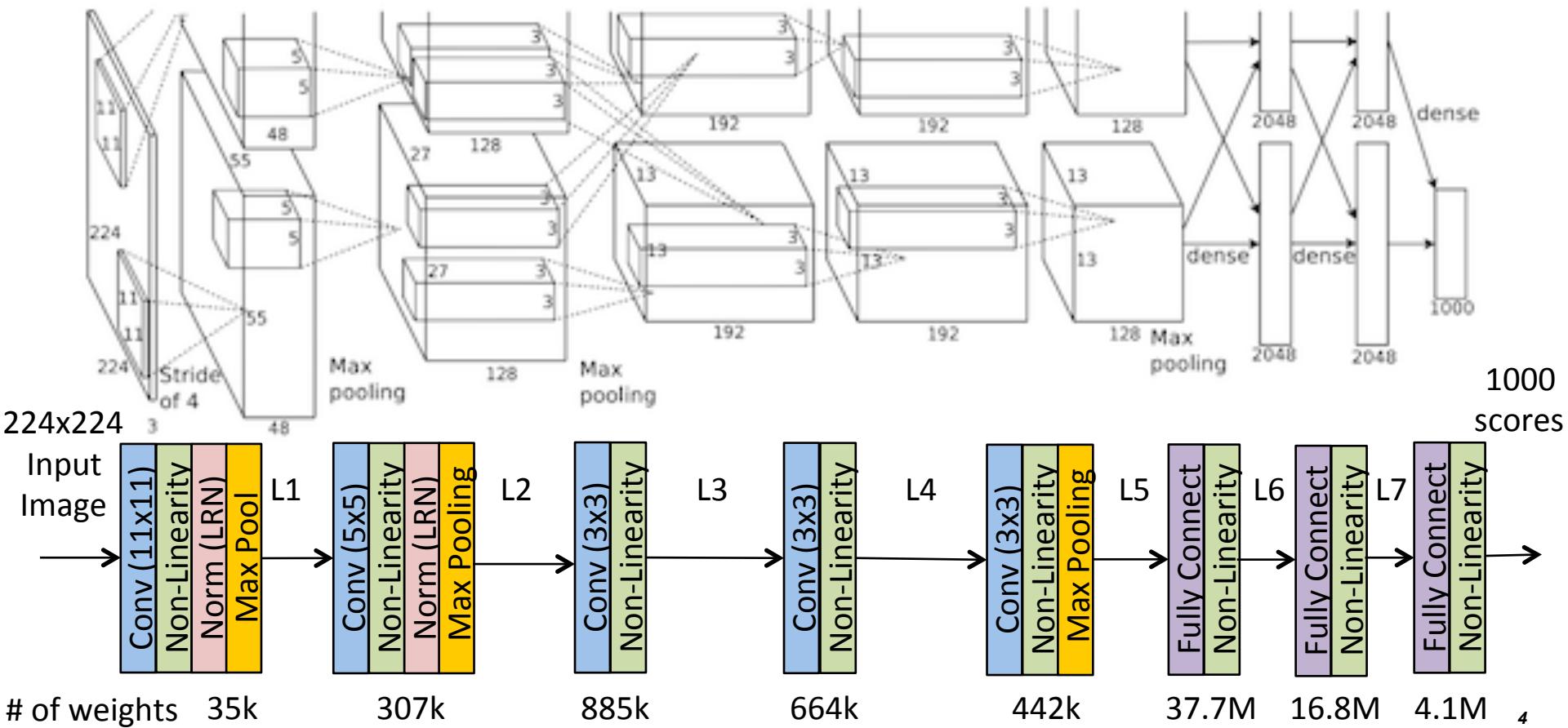
Uses Local Response Normalization (LRN)

Weights: 61M

MACs: 724M

ReLU used for non-linearity

[Krizhevsky et al., NIPS, 2012]

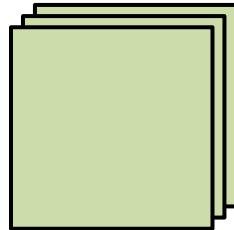


# Large Sizes with Varying Shapes

## AlexNet Convolutional Layer Configurations

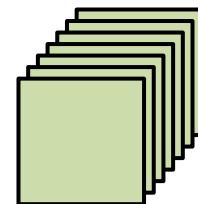
Layer	Filter Size (RxS)	# Filters (M)	# Channels (C)	Stride
1	11x11	96	3	4
2	5x5	256	48	1
3	3x3	384	256	1
4	3x3	384	192	1
5	3x3	256	192	1

Layer 1



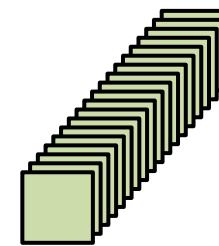
**34k Params**  
**105M MACs**

Layer 2



**307k Params**  
**224M MACs**

Layer 3



**885k Params**  
**150M MACs**

# VGG-16

CONV Layers: 13

Fully Connected Layers: 3

Weights: 138M

MACs: 15.5G

Also, 19 layer version

Reduce # of weights

stack 2

3x3 conv

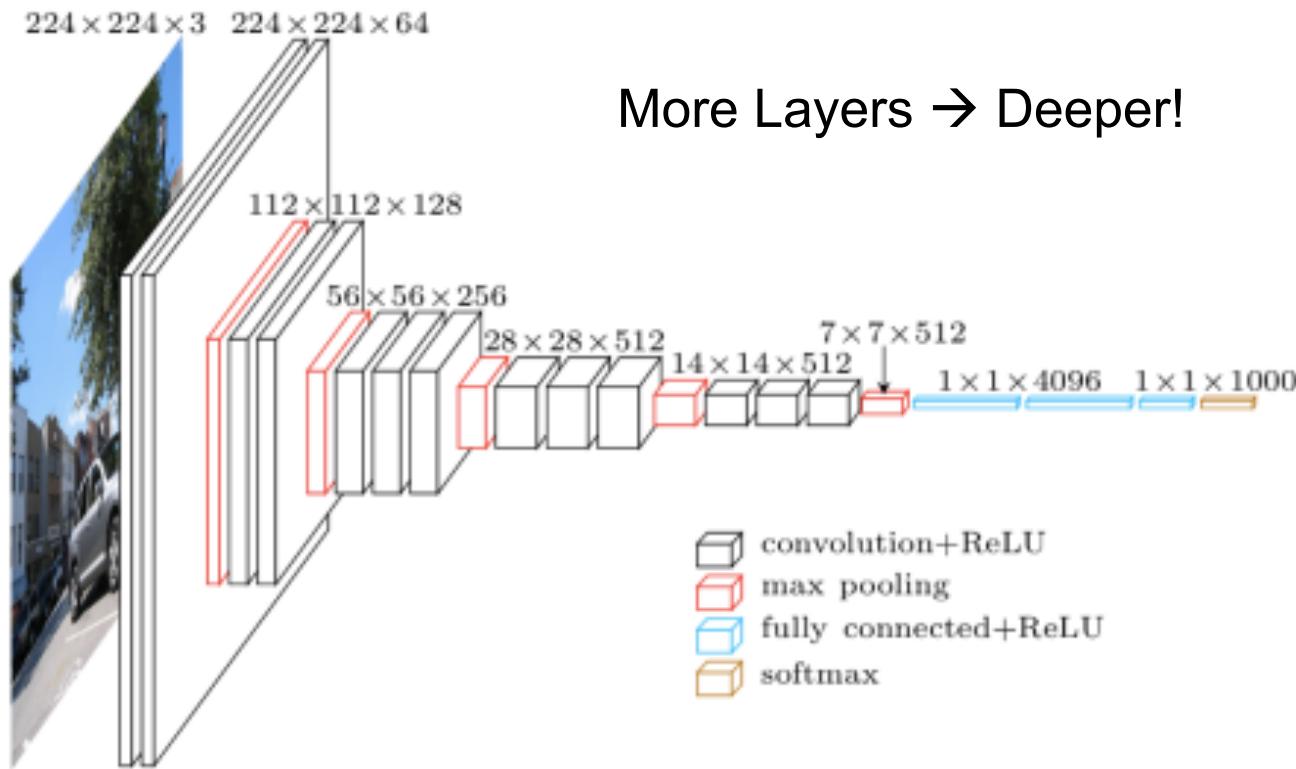
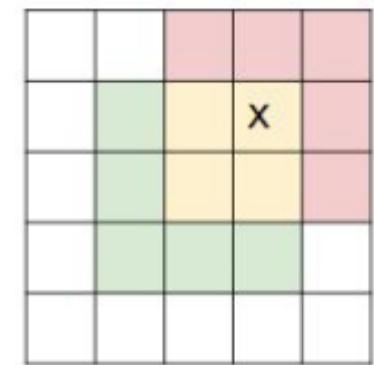


Image Source: <http://www.cs.toronto.edu/~frossard/post/vgg16/>



for a 5x5 receptive field

[figure credit  
A. Karpathy]

# GoogLeNet (v1)

CONV Layers: 21 (depth), 57 (total)

Fully Connected Layers: 1

Weights: 7.0M

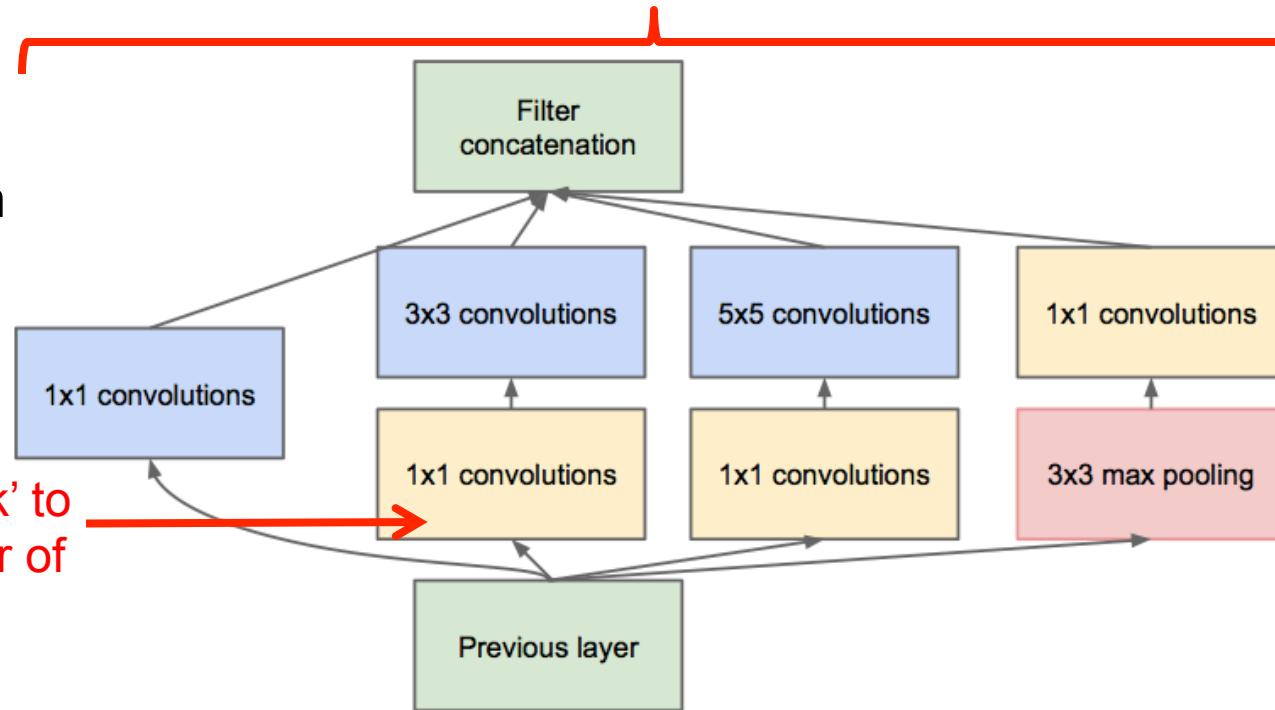
MACs: 1.43G

Also, v2, v3 and v4  
ILSVRC14 Winner

parallel filters of different size has the effect of  
processing image at different scales

## Inception Module

1x1 'bottleneck' to  
reduce number of  
weights



# GoogLeNet (v1)

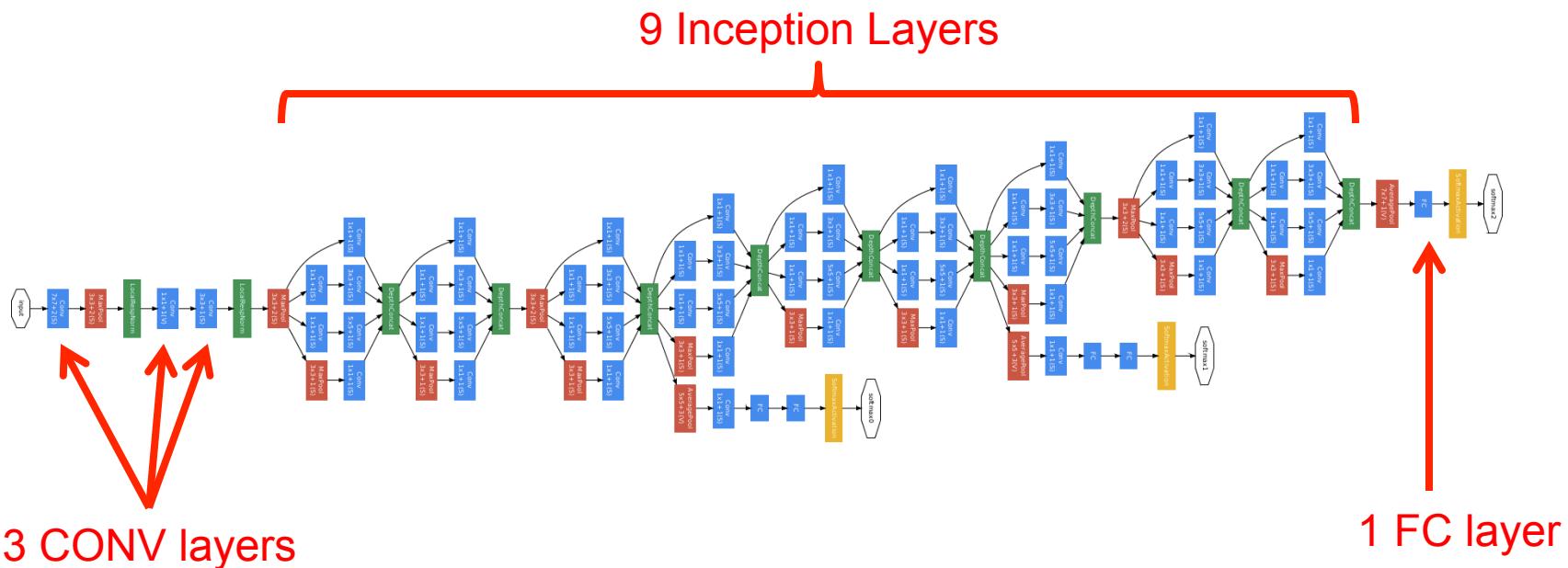
CONV Layers: 21 (depth), 57 (total)

Fully Connected Layers: 1

Weights: 7.0M

MACs: 1.43G

Also, v2, v3 and v4  
ILSVRC14 Winner



# ResNet-50

CONV Layers: 49

Also, 34, 152 and 1202 layer versions

Fully Connected Layers: 1

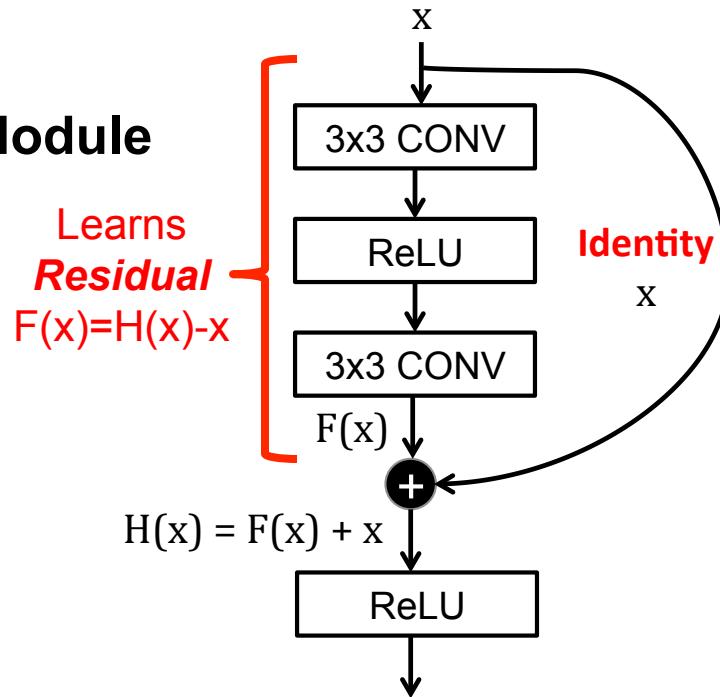
ILSVRC15 Winner

Weights: 25.5M

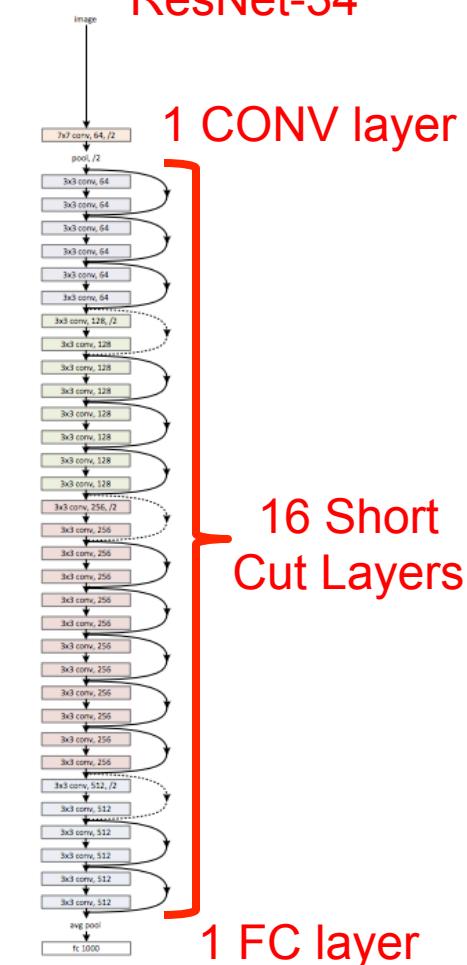
ResNet-34

MACs: 3.9G

## Short Cut Module



Helps address the vanishing gradient challenge for training very deep networks



# Revolution of Depth

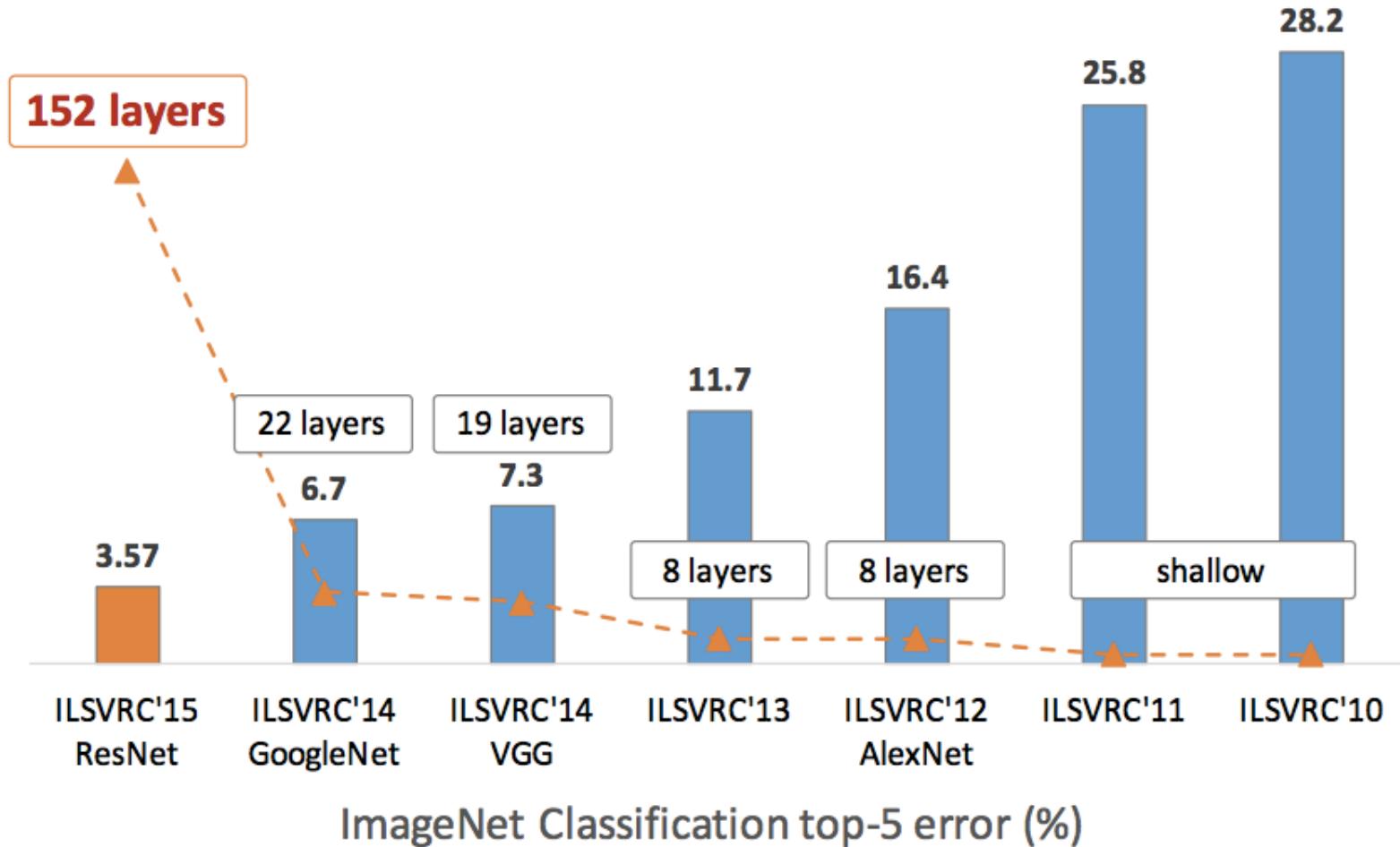


Image Source: [http://icml.cc/2016/tutorials/icml2016\\_tutorial\\_deep\\_residual\\_networks\\_kaiminghe.pdf](http://icml.cc/2016/tutorials/icml2016_tutorial_deep_residual_networks_kaiminghe.pdf)

# Summary of Popular DNNs

Metrics	LeNet-5	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Top-5 error	n/a	16.4	7.4	6.7	5.3
Input Size	28x28	227x227	224x224	224x224	224x224
<b># of CONV Layers</b>	<b>2</b>	<b>5</b>	<b>16</b>	<b>21 (depth)</b>	<b>49</b>
Filter Sizes	5	3, 5, 11	3	1, 3, 5, 7	1, 3, 7
# of Channels	1, 6	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	6, 16	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1	1, 4	1	1, 2	1, 2
# of Weights	2.6k	2.3M	14.7M	6.0M	23.5M
# of MACs	283k	666M	15.3G	1.43G	3.86G
<b># of FC layers</b>	<b>2</b>	<b>3</b>	<b>3</b>	<b>1</b>	<b>1</b>
# of Weights	58k	58.6M	124M	1M	2M
# of MACs	58k	58.6M	124M	1M	2M
<b>Total Weights</b>	<b>60k</b>	<b>61M</b>	<b>138M</b>	<b>7M</b>	<b>25.5M</b>
<b>Total MACs</b>	<b>341k</b>	<b>724M</b>	<b>15.5G</b>	<b>1.43G</b>	<b>3.9G</b>

CONV Layers increasingly important!

# Summary of Popular DNNs

---

- **AlexNet**
  - First CNN Winner of ILSVRC
  - Uses LRN (deprecated after this)
- **VGG-16**
  - Goes Deeper (16+ layers)
  - Uses only 3x3 filters (stack for larger filters)
- **GoogLeNet (v1)**
  - Reduces weights with Inception and only one FC layer
  - Inception: 1x1 and DAG (parallel connections)
  - Batch Normalization
- **ResNet**
  - Goes Deeper (24+ layers)
  - Shortcut connections

# Frameworks

---

## Caffe\*

Berkeley / BVLC  
(C, C++, Python, MATLAB)

## theano

U. Montreal  
(Python)

Also, CNTK, MXNet, etc.

More at: <https://developer.nvidia.com/deep-learning-frameworks>



## TensorFlow

Google  
(C++, Python)



Facebook / NYU  
(C, C++, Lua)

# Example: Layers in Caffe

## Convolution Layer

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  ...
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
  ...
}
```

## Non-Linearity

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

## Pooling Layer

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 2
    stride: 2 ...
}
```

# Benefits of Frameworks

---

- **Rapid development**
- **Sharing models**
- **Workload profiling**
- **Network hardware co-design**

# Image Classification Datasets

- **Image Classification/Recognition**
  - Given an entire image → Select 1 of N classes
  - No localization (detection)

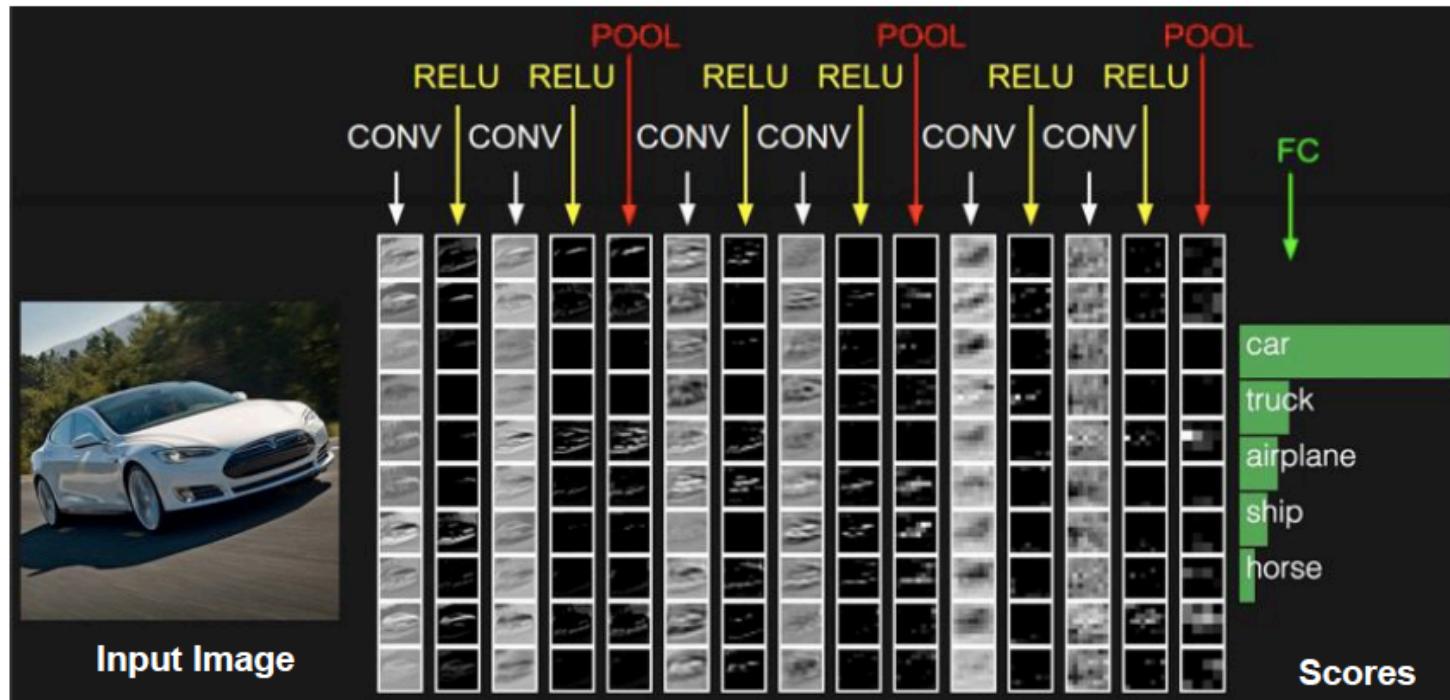


Image Source: Stanford cs231n

Datasets affect difficulty of task

# MNIST

---

## Digit Classification

28x28 pixels (B&W)

10 Classes

60,000 Training

10,000 Testing

LeNet in 1998

(0.95% error)



ICML 2013

(0.21% error)

3	6	8	1	7	9	6	6	9	1
6	7	5	7	8	6	3	4	8	5
2	1	7	9	7	1	2	8	4	6
4	8	1	9	0	1	8	8	9	4
7	6	1	8	6	4	1	5	6	0
7	5	9	2	6	5	8	1	9	7
1	2	2	2	2	3	4	4	8	0
0	2	3	8	0	7	3	8	5	7
0	1	4	6	4	6	0	2	4	3
7	1	2	8	7	6	9	8	6	1

## Object Classification

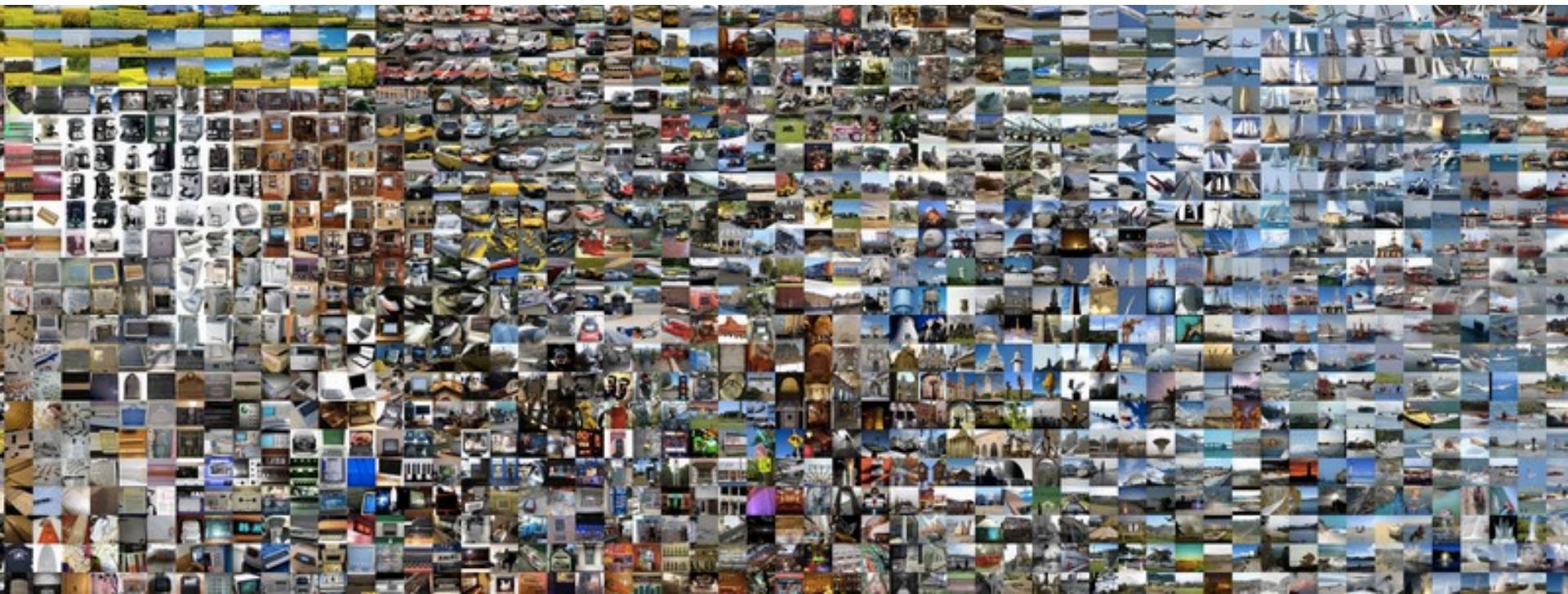
~256x256 pixels (color)

1000 Classes

1.3M Training

100,000 Testing (50,000 Validation)

Image Source: <http://karpathy.github.io/>





**Fine grained  
Classes  
(120 breeds)**

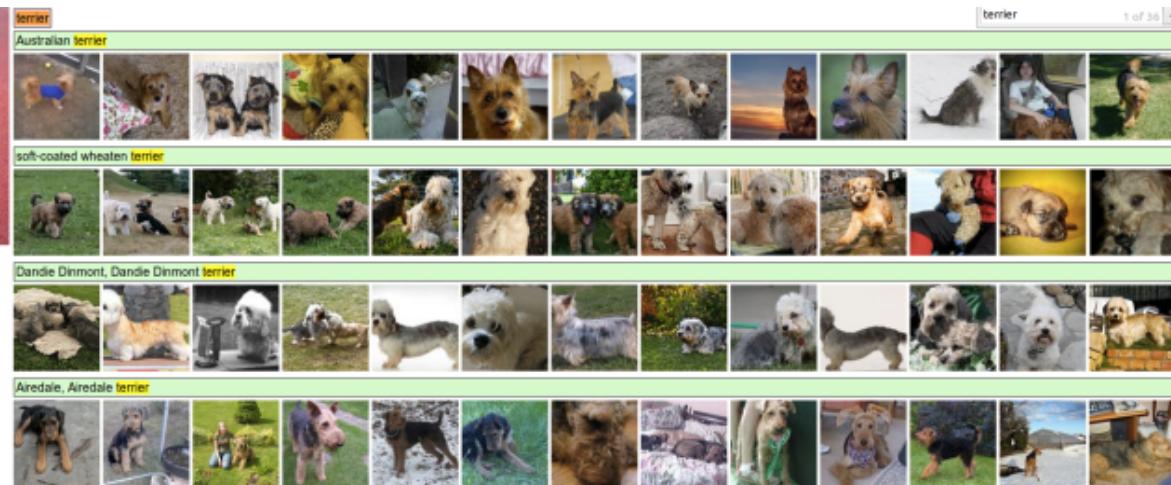


Image Source: <http://karpathy.github.io/>

**Top-5 Error**  
Winner 2012  
(16.42% error)



Winner 2016  
(2.99% error)

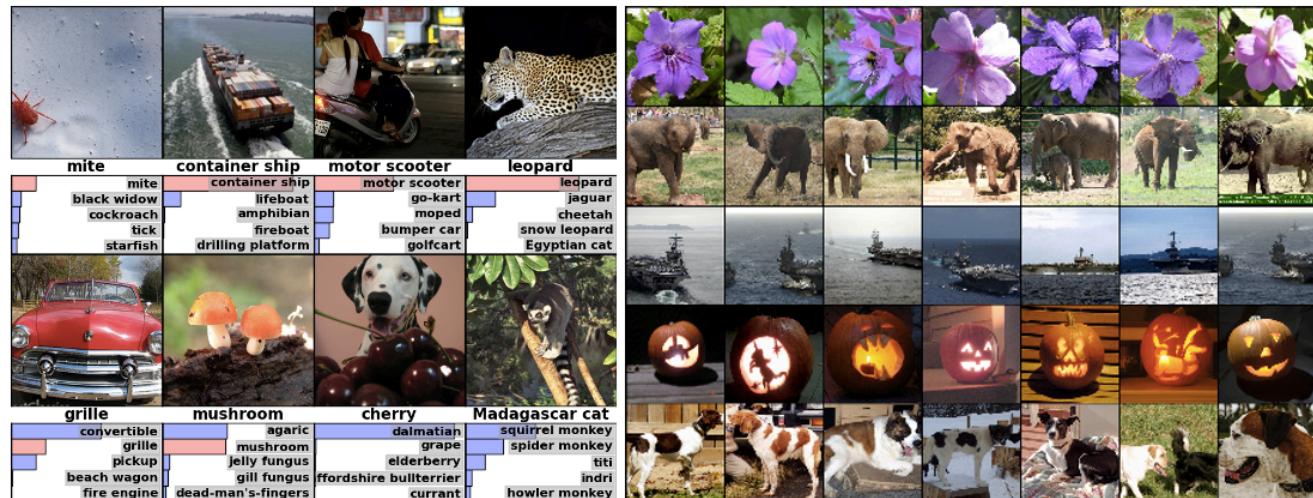


Image Source: Krizhevsky et al., NIPS 2012

# Image Classification Summary

---

	MNIST	IMAGENET
Year	1998	2012
Resolution	28x28	256x256
Classes	10	1000
Training	60k	1.3M
Testing	10k	100k
Accuracy	0.21% error (ICML 2013)	2.99% top-5 error (2016 winner)

[http://rodrigob.github.io/are\\_we\\_there\\_yet/build/  
classification\\_datasets\\_results.html](http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html)

# Next Tasks: Localization and Detection

## Image classification



Ground truth

Steel drum  
Folding chair  
Loudspeaker

Accuracy: 1

Scale  
T-shirt  
Steel drum  
Drumstick  
Mud turtle

Accuracy: 1

Scale  
T-shirt  
Giant panda  
Drumstick  
Mud turtle

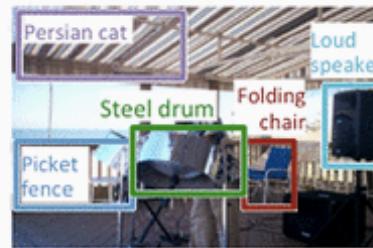
Accuracy: 0

## Steel drum

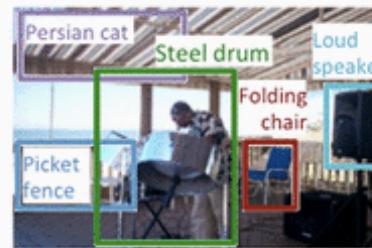
## Single-object localization



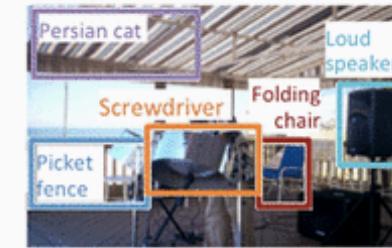
Ground truth



Accuracy: 1

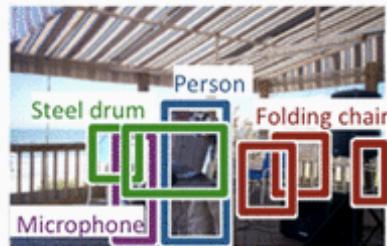


Accuracy: 0



Accuracy: 0

## Object detection



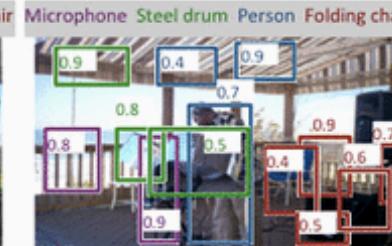
Ground truth



AP: 1.0 1.0 1.0 1.0



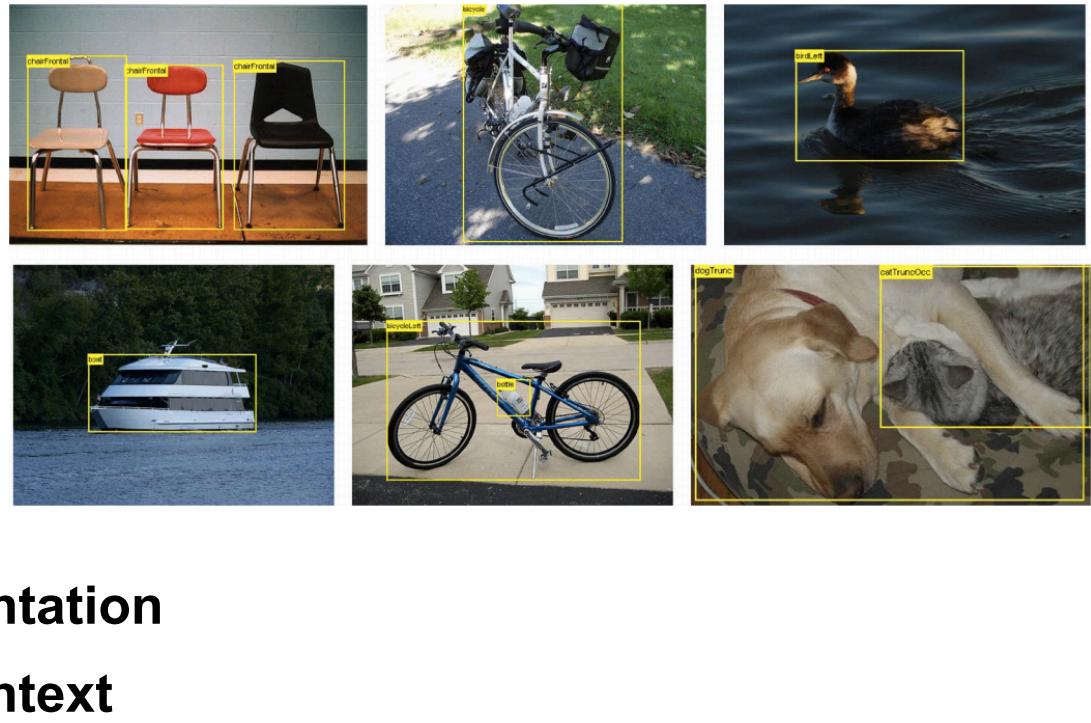
AP: 0.0 0.5 1.0 0.3



AP: 1.0 0.7 0.5 0.9

# Others Popular Datasets

- **Pascal VOC**
  - 11k images
  - Object Detection
  - 20 classes
- **MS COCO**
  - 300k images
  - Detection, Segmentation
  - Recognition in context



# Recently Introduced Datasets

---

- **Google Open Images (~9M images)**
  - <https://github.com/openimages/dataset>
- **Youtube-8M (8M videos)**
  - <https://research.google.com/youtube8m/>
- **AudioSet (2M sound clips)**
  - <https://research.google.com/audioset/index.html>

# Summary

---

- **Development resources presented in this section enable us to evaluate hardware using the appropriate DNN model and dataset**
  - Difficult tasks typically require larger models
  - Different datasets for different tasks
  - Number of datasets growing at a rapid pace

# **Survey of DNN Hardware**

**ISCA Tutorial (2017)**

Website: <http://eyeriss.mit.edu/tutorial.html>

# CPUs Are Targeting Deep Learning

---

## Intel Knights Landing (2016)



- 7 TFLOPS FP32
- 16GB MCDRAM – 400 GB/s
- 245W TDP
- 29 GFLOPS/W (FP32)
- 14nm process

## Knights Mill: next gen Xeon Phi “optimized for deep learning”

Intel announced the addition of new vector instructions for deep learning (AVX512-4VNNIW and AVX512-4FMAPS), October 2016

# GPUs Are Targeting Deep Learning

---

## Nvidia PASCAL GP100 (2016)



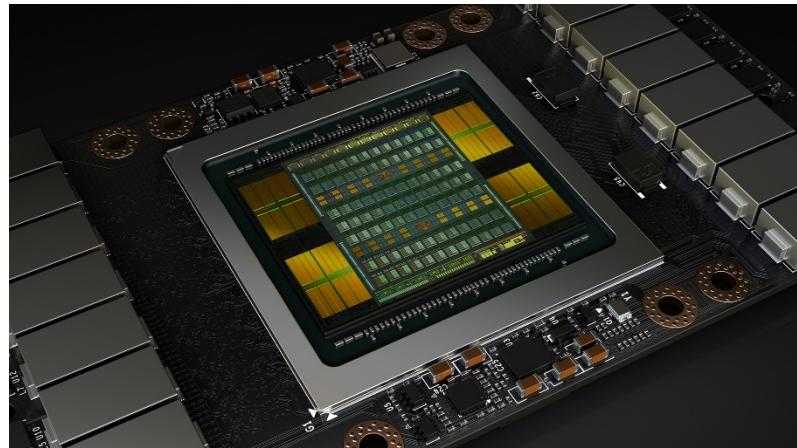
- 10/20 TFLOPS FP32/FP16
- 16GB HBM – 750 GB/s
- 300W TDP
- 33/67 GFLOPS/W (FP32/FP16)
- 16nm process
- 160GB/s NV Link

Source: Nvidia

# GPUs Are Targeting Deep Learning

---

## Nvidia VOLTA GV100 (2017)



- 15 TFLOPS FP32
- 16GB HBM2 – 900 GB/s
- 300W TDP
- 50 GFLOPS/W (FP32)
- 12nm process
- 300GB/s NV Link2
- Tensor Core....

Source: Nvidia

# GV100 – “Tensor Core”

---

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

Tensor Core....

- 120 TFLOPS (FP16)
- 400 GFLOPS/W (FP16)

# Systems for Deep Learning

---

## Nvidia DGX-1 (2016)



- 170 TFLOPS
- 8× Tesla P100, Dual Xeon
- NVLink Hybrid Cube Mesh
- Optimized DL Software
- 7 TB SSD Cache
- Dual 10GbE, Quad IB 100Gb
- 3RU – 3200W

Source: Nvidia

# Cloud Systems for Deep Learning

---

## Facebook's Deep Learning Machine

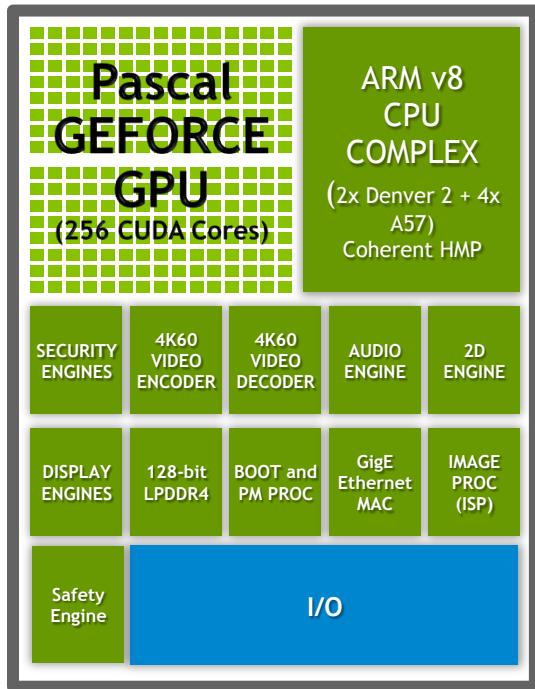


- Open Rack Compliant
- Powered by 8 Tesla M40 GPUs
- 2x Faster Training for Faster Deployment
- 2x Larger Networks for Higher Accuracy

Source: Facebook

# SOCs for Deep Learning Inference

## Nvidia Tegra - Parker



- GPU: 1.5 TeraFLOPS FP16
- 4GB LPDDR4 @ 25.6 GB/s
- 15 W TDP  
(1W idle, <10W typical)
- 100 GFLOPS/W (FP16)
- 16nm process

**Xavier:** next gen Tegra to be an “AI supercomputer”

Source: Nvidia

# Mobile SOCs for Deep Learning

---

## Samsung Exynos (ARM Mali)

### Exynos 8 Octa 8890



- GPU: 0.26 TFLOPS
- LPDDR4 @ 28.7 GB/s
- 14nm process

Source: Wikipedia

# FPGAs for Deep Learning

---



## Intel/Altera Stratix 10

- 10 TFLOPS FP32
- HBM2 integrated
- Up to 1 GHz
- 14nm process
- 80 GFLOPS/W



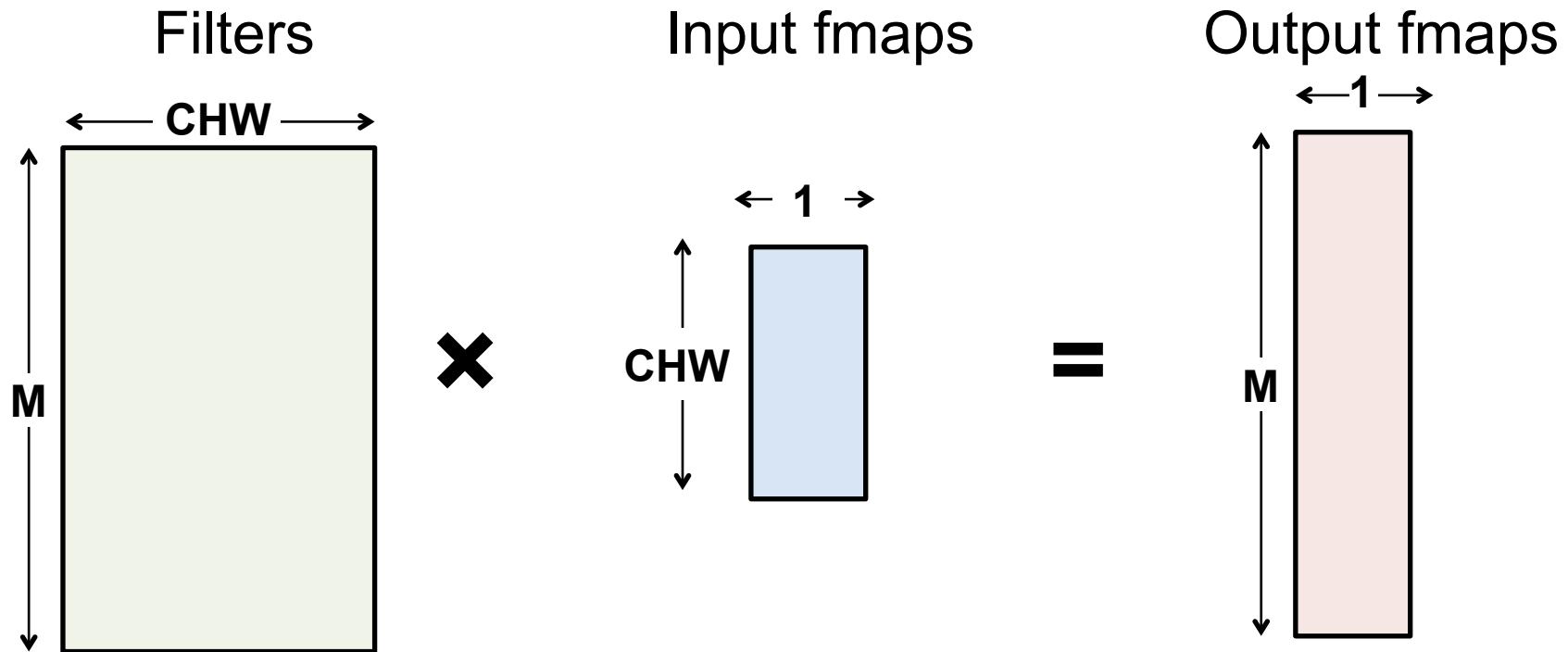
## Xilinx Virtex UltraSCALE+

- DSP: up to 21.2 TMACS
- DSP: up to 890 MHz
- Up to 500Mb On-Chip Memory
- 16nm process

# **Kernel Computation**

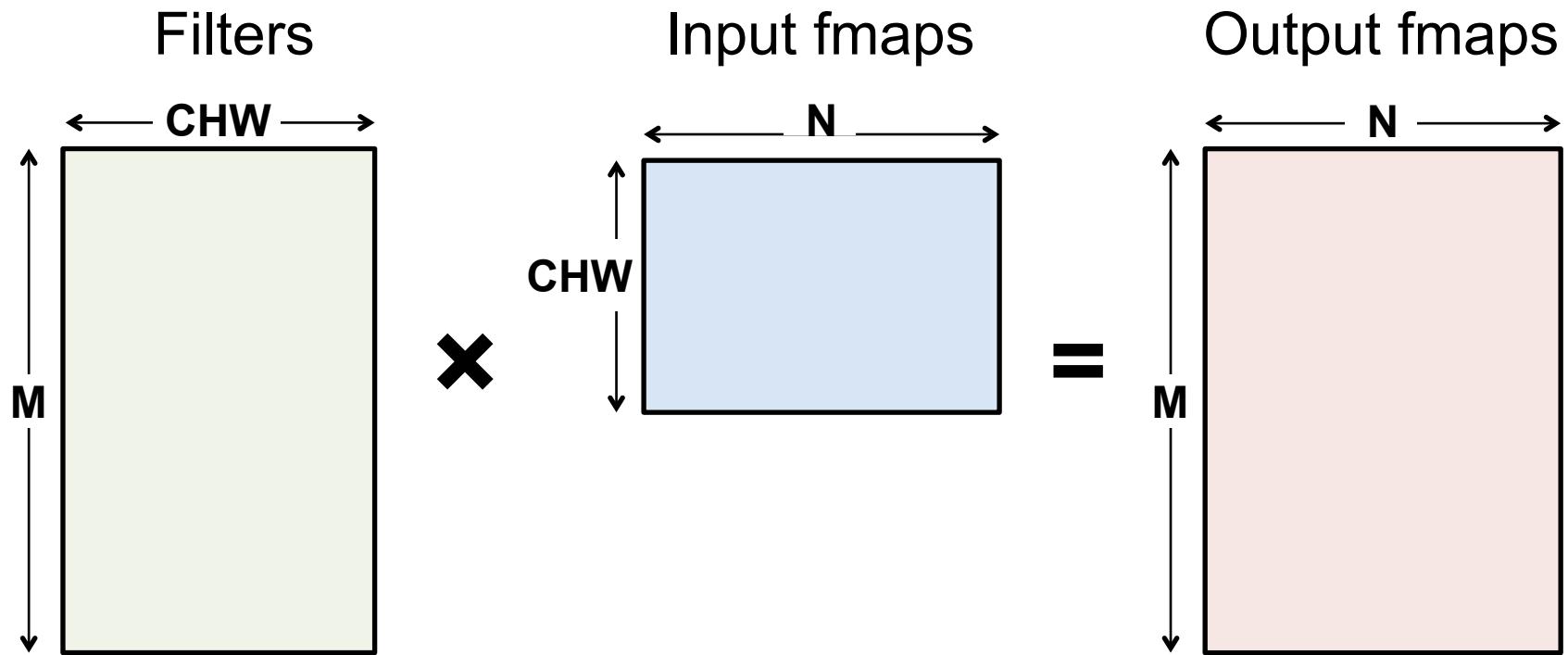
# Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



# Fully-Connected (FC) Layer

- Batching (N) turns operation into a Matrix-Matrix multiply



# Fully-Connected (FC) Layer

---

- Implementation: **Matrix Multiplication (GEMM)**
  - **CPU:** OpenBLAS, Intel MKL, etc
  - **GPU:** cuBLAS, cuDNN, etc
- Optimized by tiling to storage hierarchy

# Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**

Convolution:

<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4	$*$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	$=$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4
1	2																				
3	4																				
1	2	3																			
4	5	6																			
7	8	9																			
1	2																				
3	4																				



**Toeplitz Matrix  
(w/ redundant data)**

Matrix Mult:

<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4	$\times$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>4</td><td>5</td></tr><tr><td>2</td><td>3</td><td>5</td><td>6</td></tr><tr><td>4</td><td>5</td><td>7</td><td>8</td></tr><tr><td>5</td><td>6</td><td>8</td><td>9</td></tr></table>	1	2	4	5	2	3	5	6	4	5	7	8	5	6	8	9	$=$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4
1	2	3	4																									
1	2	4	5																									
2	3	5	6																									
4	5	7	8																									
5	6	8	9																									
1	2	3	4																									

# Convolution (CONV) Layer

- Convert to matrix mult. using the **Toeplitz Matrix**

Convolution:

<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4	$*$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td>9</td></tr></table>	1	2	3	4	5	6	7	8	9	$=$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td></tr><tr><td>3</td><td>4</td></tr></table>	1	2	3	4
1	2																				
3	4																				
1	2	3																			
4	5	6																			
7	8	9																			
1	2																				
3	4																				



**Toeplitz Matrix  
(w/ redundant data)**

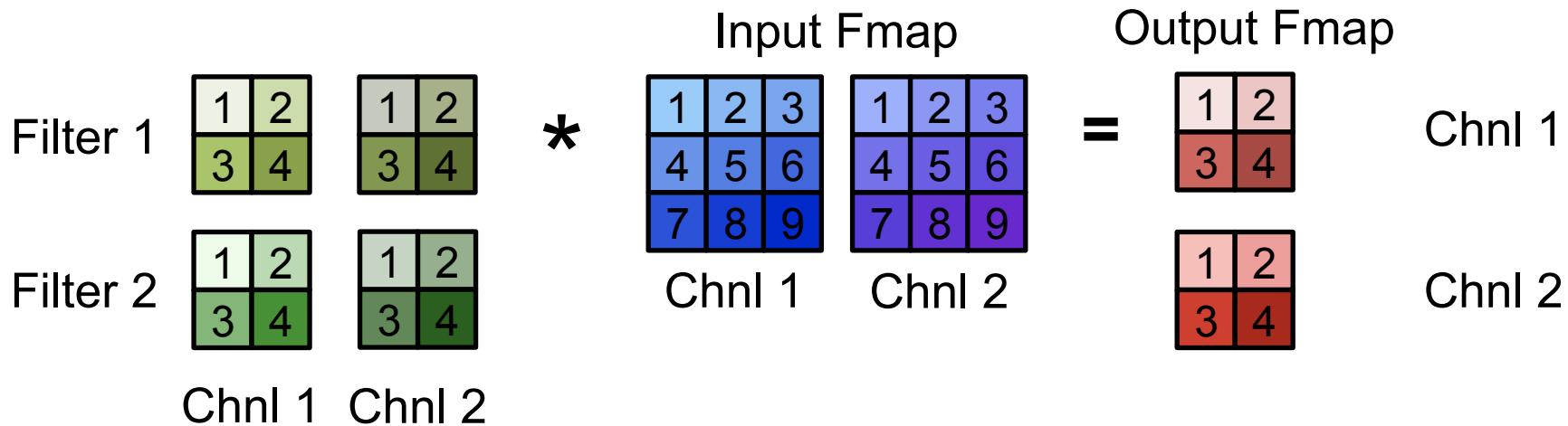
<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4	$\times$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>4</td><td>5</td></tr><tr><td>2</td><td>3</td><td>5</td><td>6</td></tr><tr><td>4</td><td>5</td><td>7</td><td>8</td></tr><tr><td>5</td><td>6</td><td>8</td><td>9</td></tr></table>	1	2	4	5	2	3	5	6	4	5	7	8	5	6	8	9	$=$	<table border="1" style="border-collapse: collapse; text-align: center;"><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr></table>	1	2	3	4
1	2	3	4																									
1	2	4	5																									
2	3	5	6																									
4	5	7	8																									
5	6	8	9																									
1	2	3	4																									

Matrix Mult:

**Data is repeated**

# Convolution (CONV) Layer

- Multiple Channels and Filters



# Convolution (CONV) Layer

- Multiple Channels and Filters

**Toeplitz Matrix**  
(w/ redundant data)

	Chnl 1				Chnl 2			
Filter 1	1	2	3	4	1	2	3	4
Filter 2	1	2	3	4	1	2	3	4

$$\times \begin{array}{|c|c|c|c|} \hline 1 & 2 & 4 & 5 \\ \hline 2 & 3 & 5 & 6 \\ \hline 4 & 5 & 7 & 8 \\ \hline 5 & 6 & 8 & 9 \\ \hline 1 & 2 & 4 & 5 \\ \hline 2 & 3 & 5 & 6 \\ \hline 4 & 5 & 7 & 8 \\ \hline 5 & 6 & 8 & 9 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 1 & 2 & 3 & 4 \\ \hline 1 & 2 & 3 & 4 \\ \hline \end{array}$$

Chnl 1

Chnl 2

Chnl 1

Chnl 2

# Computational Transforms

# Computation Transformations

---

- **Goal: Bitwise same result, but reduce number of operations**
- **Focuses mostly on compute**

# Gauss's Multiplication Algorithm

---

$$(a + bi)(c + di) = (ac - bd) + (bc + ad)i.$$

4 multiplications + 3 additions

$$k_1 = c \cdot (a + b)$$

$$k_2 = a \cdot (d - c)$$

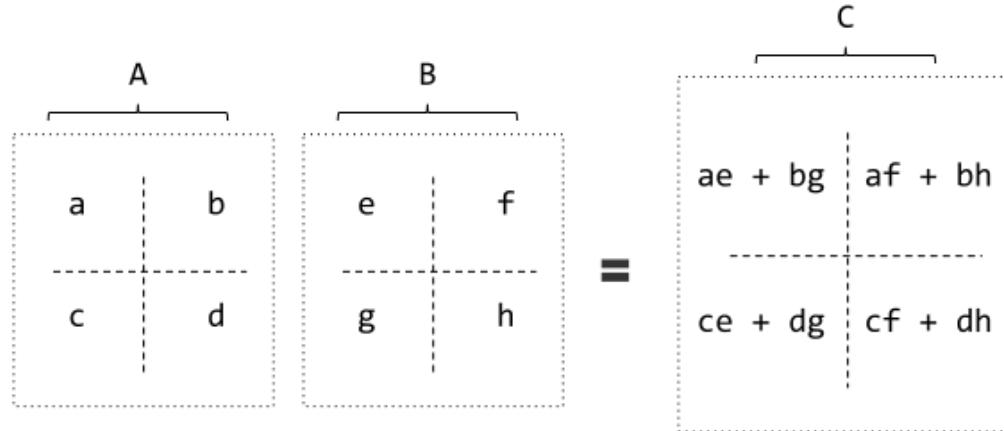
$$k_3 = b \cdot (c + d)$$

$$\text{Real part} = k_1 - k_3$$

$$\text{Imaginary part} = k_1 + k_2.$$

3 multiplications + 5 additions

# Strassen



8 multiplications + 4 additions

$$\begin{aligned}P1 &= a(f - h) \\P2 &= (a + b)h \\P3 &= (c + d)e \\P4 &= d(g - e)\end{aligned}$$

$$\begin{aligned}P5 &= (a + d)(e + h) \\P6 &= (b - d)(g + h) \\P7 &= (a - c)(e + f)\end{aligned}$$

$$AB = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

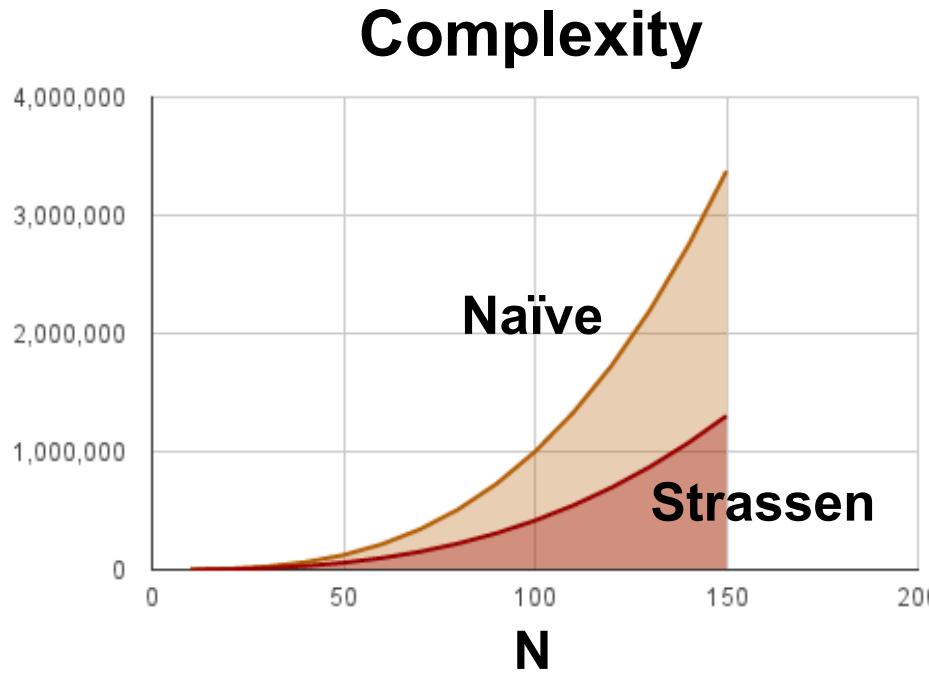
7 multiplications + 18 additions

7 multiplications + 13 additions (for constant B matrix – weights)

# Strassen

---

- Reduce the complexity of matrix multiplication from  $\Theta(N^3)$  to  $\Theta(N^{2.807})$  by reducing multiplication



Comes at the price of reduced numerical stability and requires significantly more memory

# Winograd 1D – F(2,3)

---

- Targeting convolutions instead of matrix multiply
- Notation:  $F(\text{size of output}, \text{filter size})$

$$F(2, 3) = \begin{matrix} \text{input} & & \text{filter} \\ \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} & \times & \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} \end{matrix}$$

6 multiplications + 4 additions

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

4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

# Winograd 2D - F(2x2, 3x3)

- 1D Winograd is nested to make 2D Winograd

Filter			Input Fmap				Output Fmap	
$g_{00}$	$g_{01}$	$g_{02}$	$d_{00}$	$d_{01}$	$d_{02}$	$d_{03}$	$y_{00}$	$y_{01}$
$g_{10}$	$g_{11}$	$g_{12}$	$d_{10}$	$d_{11}$	$d_{12}$	$d_{13}$	$y_{10}$	$y_{11}$
$g_{20}$	$g_{21}$	$g_{22}$	$d_{20}$	$d_{21}$	$d_{22}$	$d_{23}$		
			$d_{30}$	$d_{31}$	$d_{32}$	$d_{33}$		

\*

=

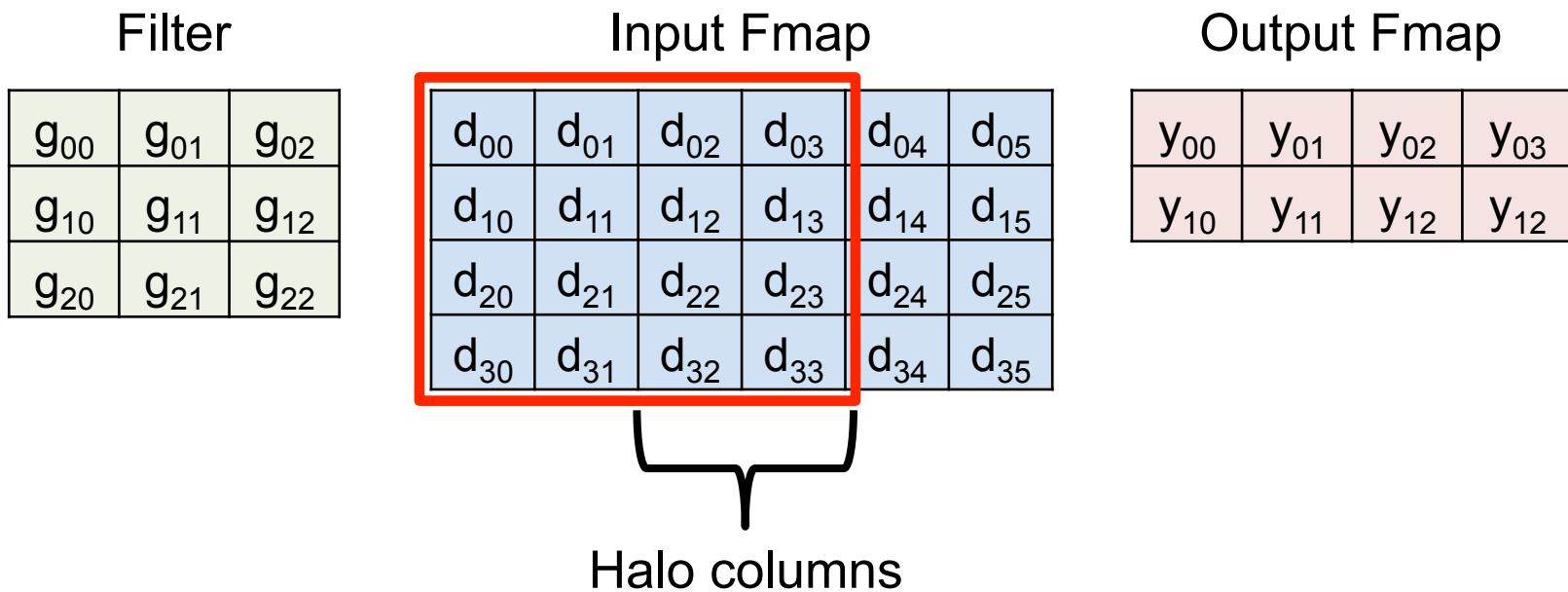
**Original:** 36 multiplications

**Winograd:** 16 multiplications  $\rightarrow$  2.25 times reduction

# Winograd Halos

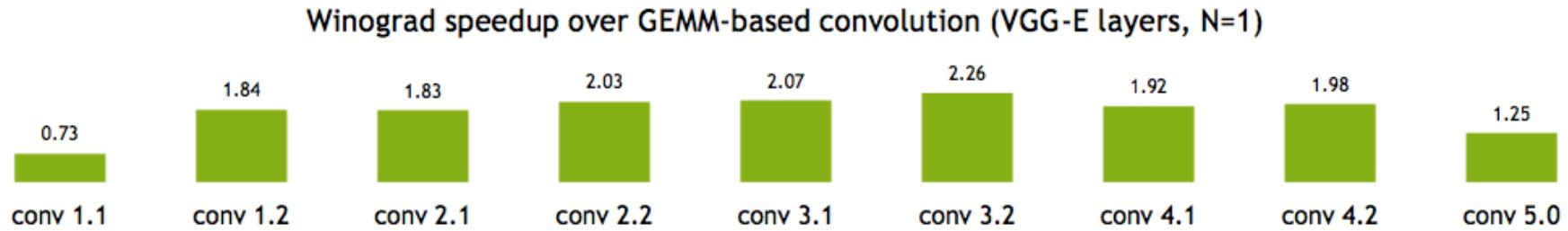
---

- Winograd works on a small region of output at a time, and therefore uses inputs repeatedly



# Winograd Performance Varies

Optimal convolution algorithm depends on convolution layer dimensions



Meta-parameters (data layouts, texture memory) afford higher performance

Using texture memory for convolutions: **13% inference speedup**

(GoogLeNet, batch size 1)

Source: Nvidia

# Winograd Summary

---

- **Winograd is an optimized computation for convolutions**
- **It can significantly reduce multiplies**
  - For example, for 3x3 filter by 2.25X
- **But, each filter size is a different computation.**

# Winograd as a Transform

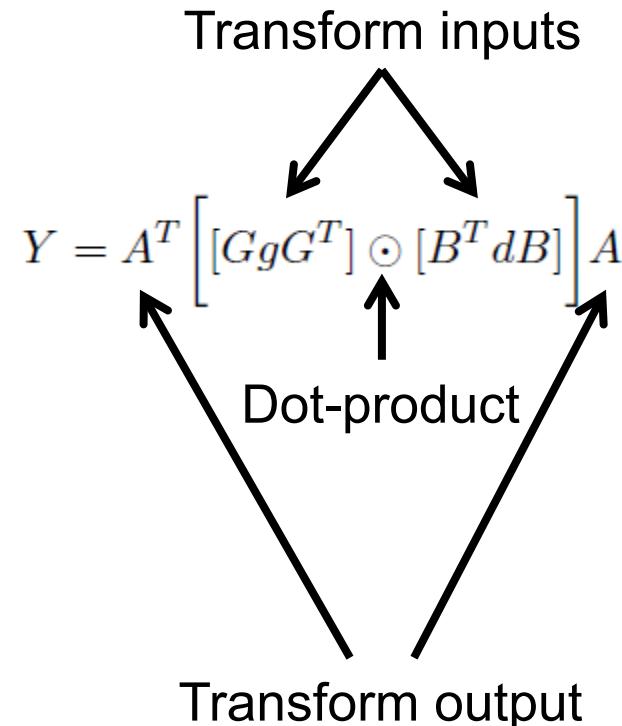
$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

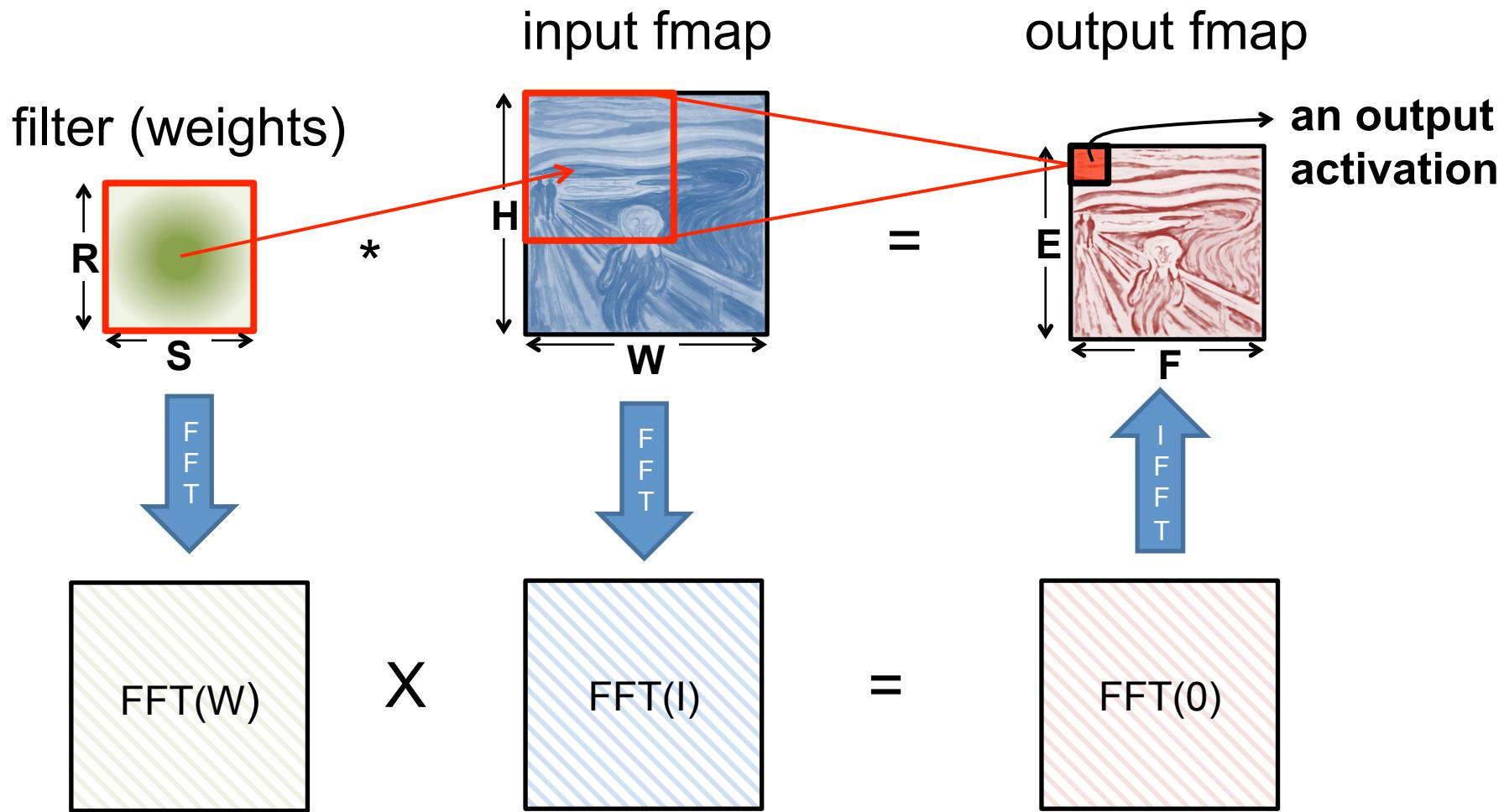
filter  $g = [g_0 \ g_1 \ g_2]^T$

input  $d = [d_0 \ d_1 \ d_2 \ d_3]^T$



$GgG^T$  can be precomputed

# FFT Flow



# FFT Overview

---

- Convert filter and input to frequency domain to make convolution a simple multiply then convert back to time domain.
- Convert direct convolution  $O(N_o^2 N_f^2)$  computation to  $O(N_o^2 \log_2 N_o)$
- So note that computational benefit of FFT decreases with decreasing size of filter

# FFT Costs

---

- **Input and Filter matrices are ‘0-completed’,**
  - i.e., expanded to size  $E+R-1 \times F+S-1$
- **Frequency domain matrices are same dimensions as input, but complex.**
- **FFT often reduces computation, but requires much more memory space and bandwidth**

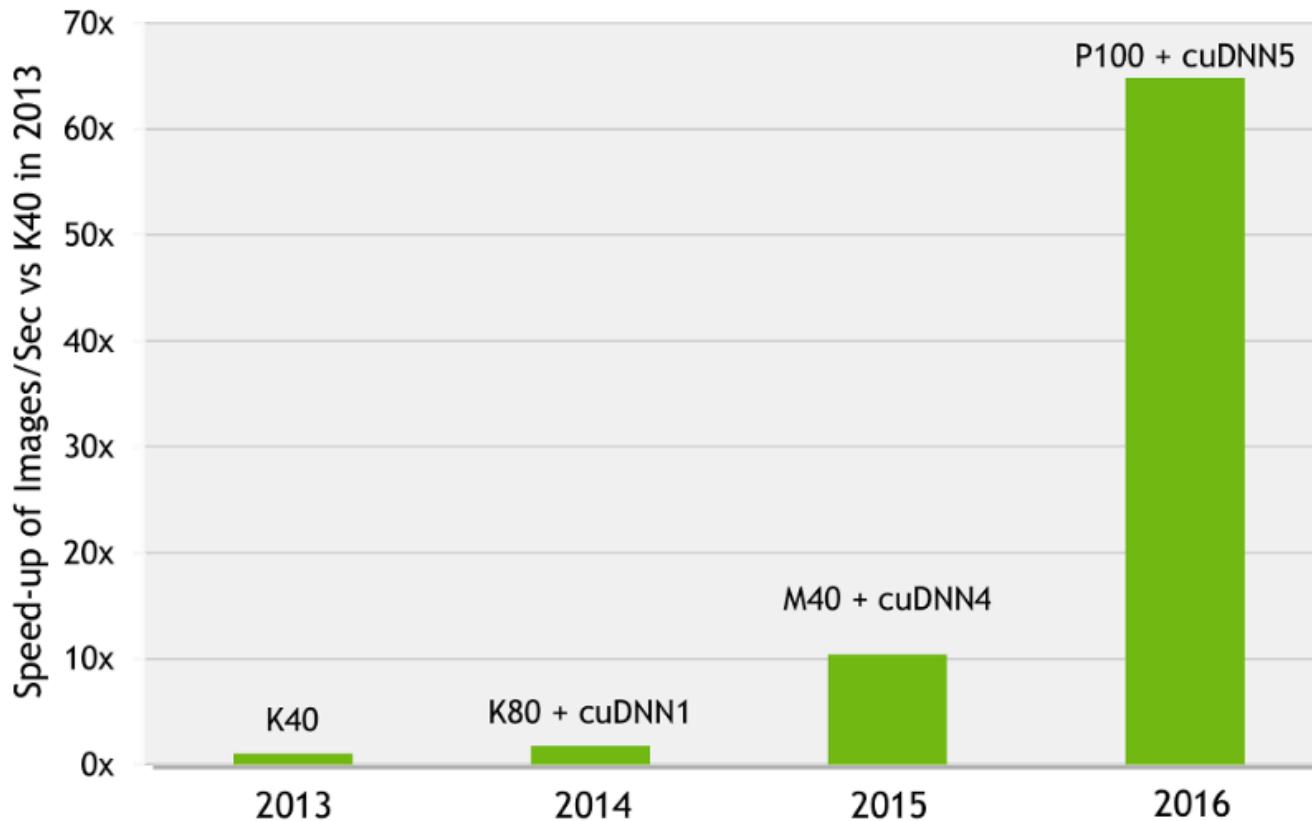
# Optimization opportunities

---

- **FFT of real matrix is symmetric allowing one to save  $\frac{1}{2}$  the computes**
- **Filters can be pre-computed and stored, but convolutional filter in frequency domain is much larger than in time domain**
- **Can reuse frequency domain version of input for creating different output channels to avoid FFT re-computations**

# cuDNN: Speed up with Transformations

60x Faster Training in 3 Years



AlexNet training throughput on:

CPU: 1x E5-2680v3 12 Core 2.5GHz. 128GB System Memory, Ubuntu 14.04

M40 bar: 8x M40 GPUs in a node, P100: 8x P100 NVLink-enabled

Source: Nvidia

# **DNN Accelerator Architectures**

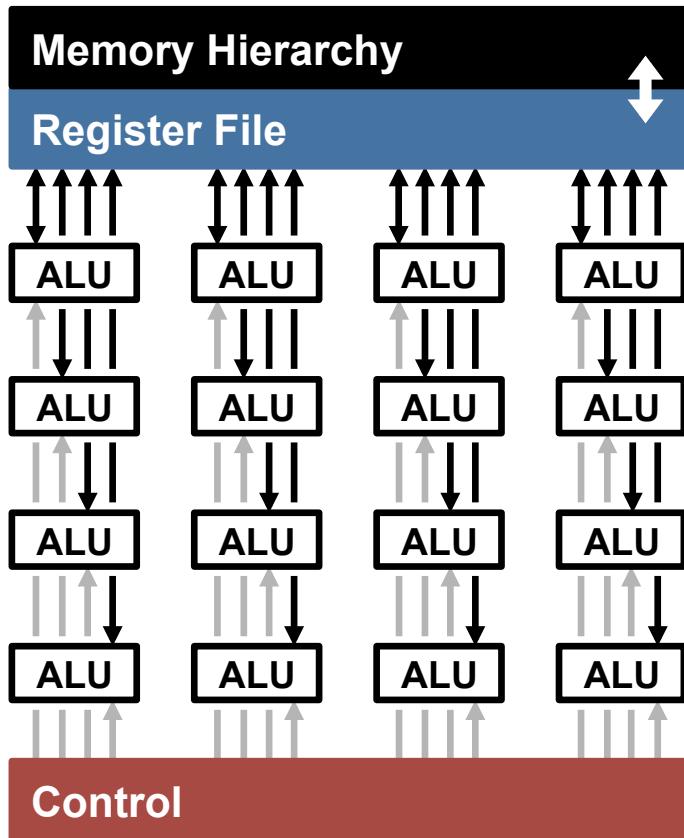
## **ISCA Tutorial (2017)**

Website: <http://eyeriss.mit.edu/tutorial.html>

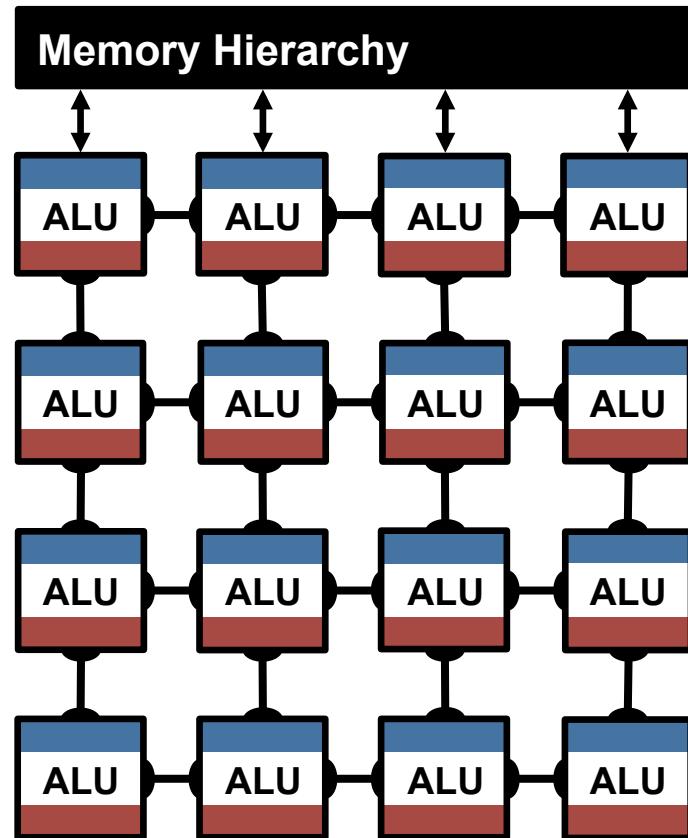
Joel Emer, Vivienne Sze, Yu-Hsin Chen

# Highly-Parallel Compute Paradigms

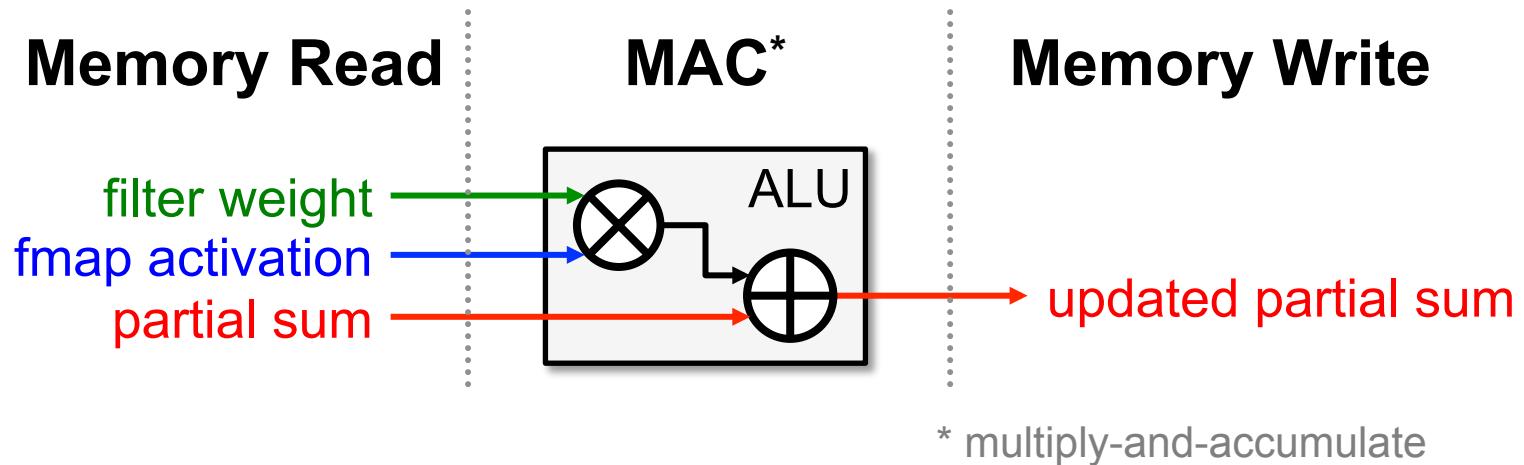
Temporal Architecture  
(SIMD/SIMT)



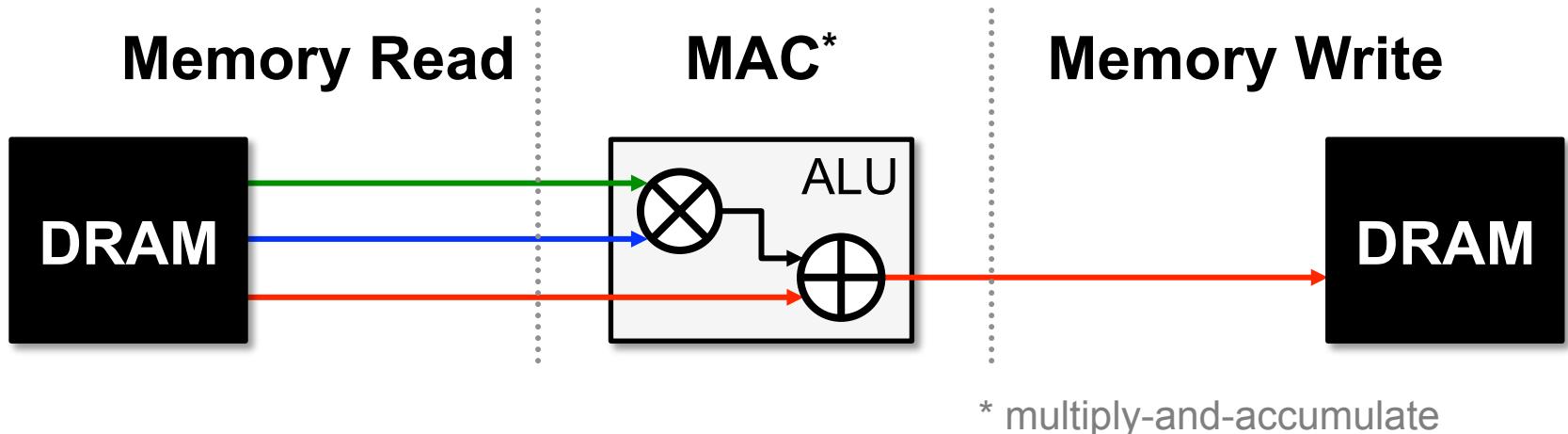
Spatial Architecture  
(Dataflow Processing)



# Memory Access is the Bottleneck



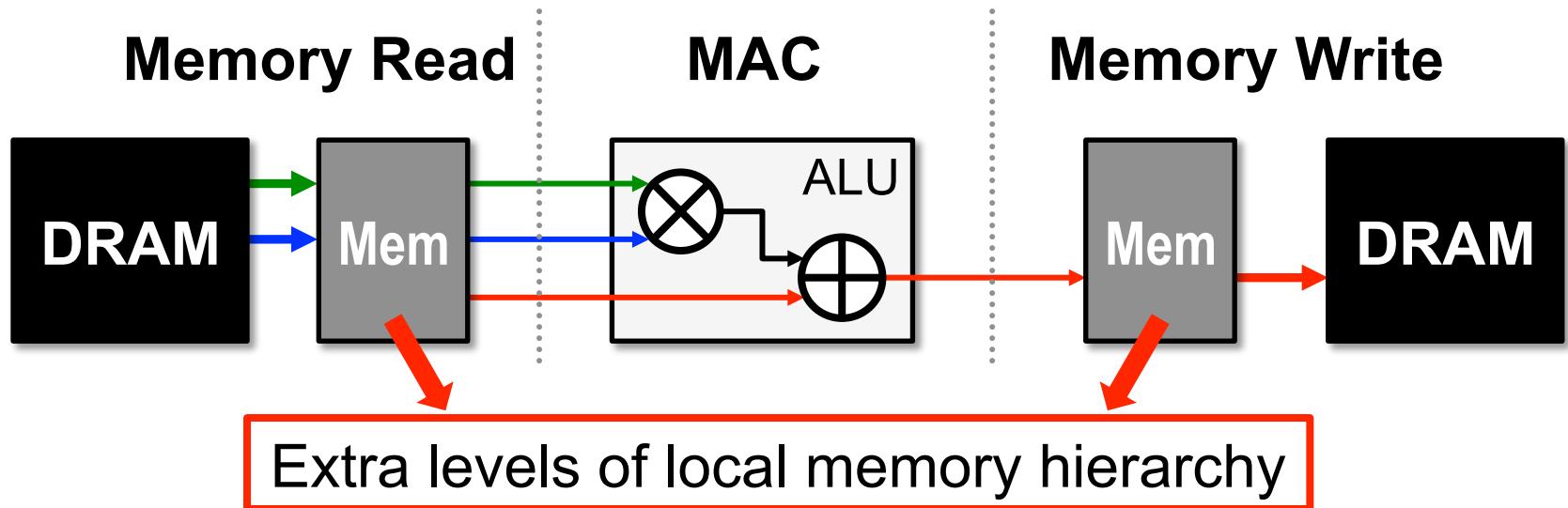
# Memory Access is the Bottleneck



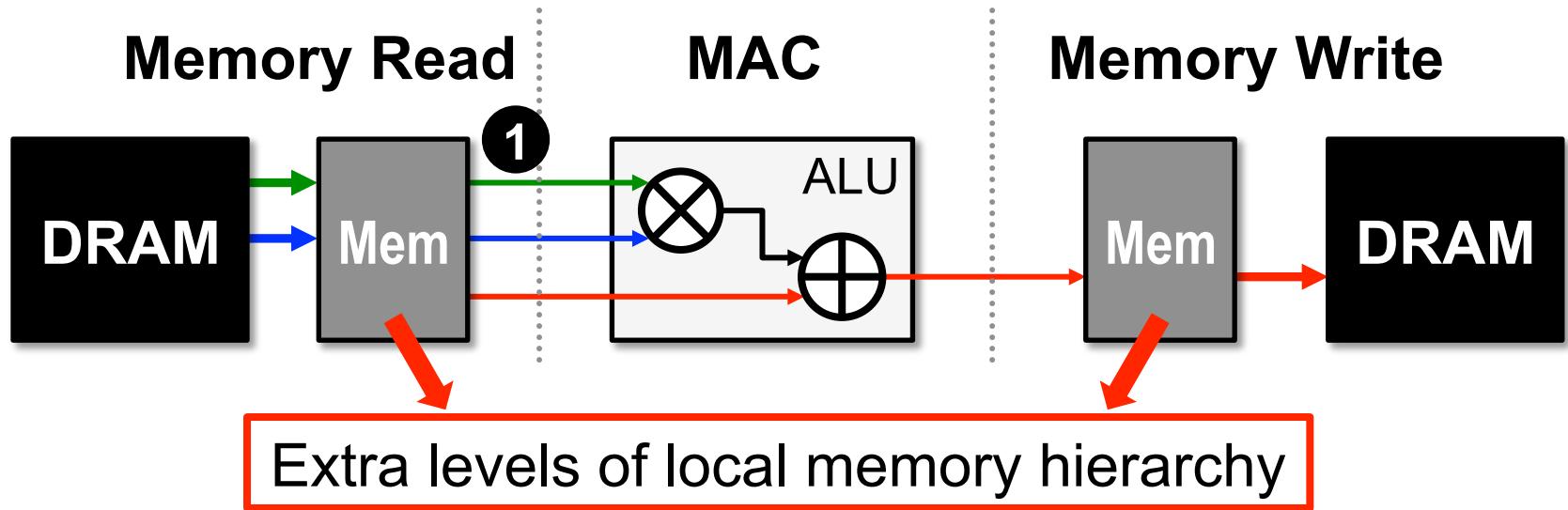
Worst Case: all memory R/W are **DRAM** accesses

- Example: AlexNet [NIPS 2012] has **724M** MACs  
→ **2896M** DRAM accesses required

# Memory Access is the Bottleneck



# Memory Access is the Bottleneck

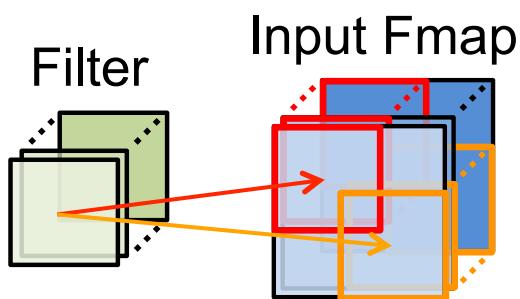


Opportunities: ① **data reuse**

# Types of Data Reuse in DNN

## Convolutional Reuse

CONV layers only  
(sliding window)

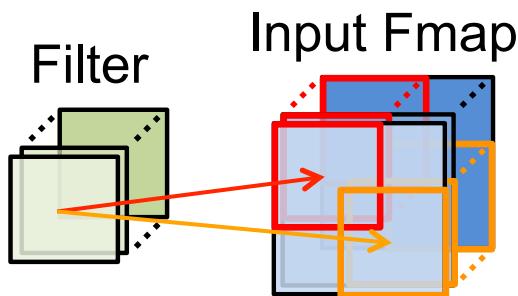


Reuse: **Activations**  
**Filter weights**

# Types of Data Reuse in DNN

## Convolutional Reuse

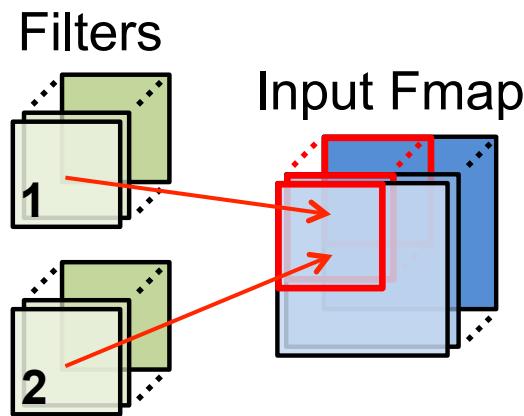
CONV layers only  
(sliding window)



Reuse: **Activations**  
**Filter weights**

## Fmap Reuse

CONV and FC layers

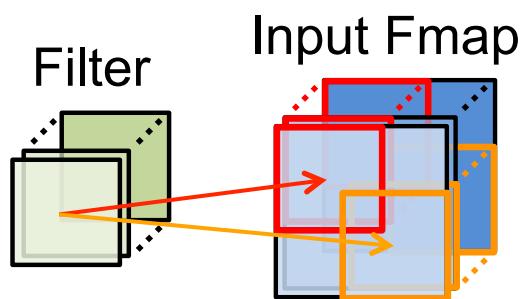


Reuse: **Activations**

# Types of Data Reuse in DNN

## Convolutional Reuse

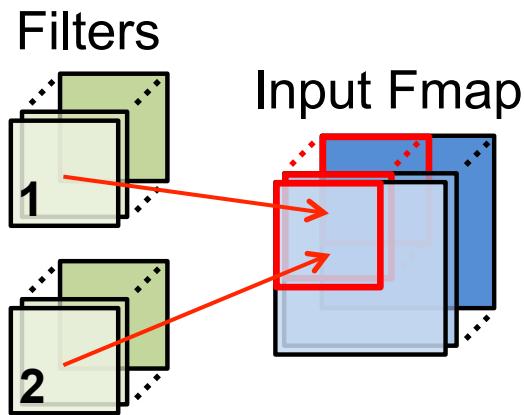
CONV layers only  
(sliding window)



Reuse: **Activations**  
**Filter weights**

## Fmap Reuse

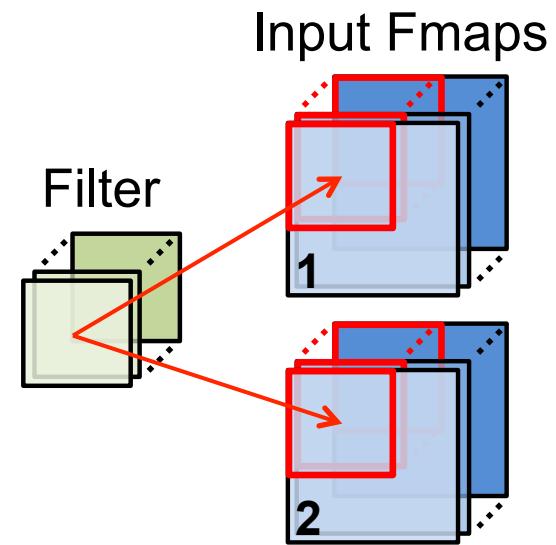
CONV and FC layers



Reuse: **Activations**

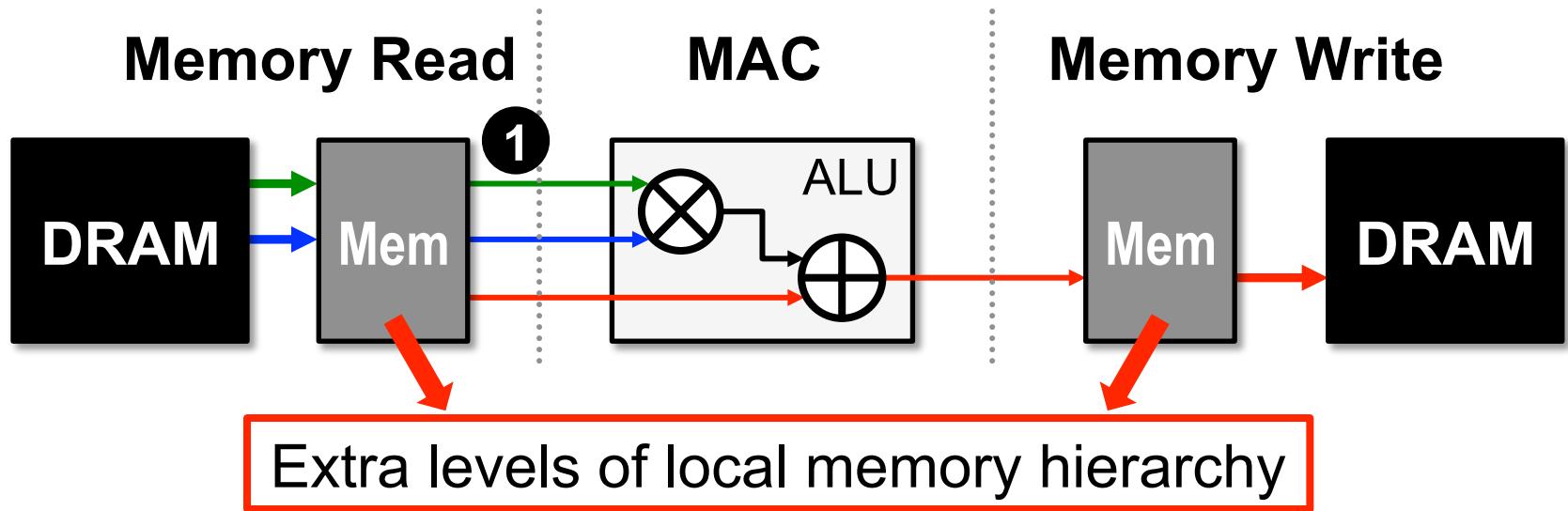
## Filter Reuse

CONV and FC layers  
(batch size > 1)



Reuse: **Filter weights**

# Memory Access is the Bottleneck

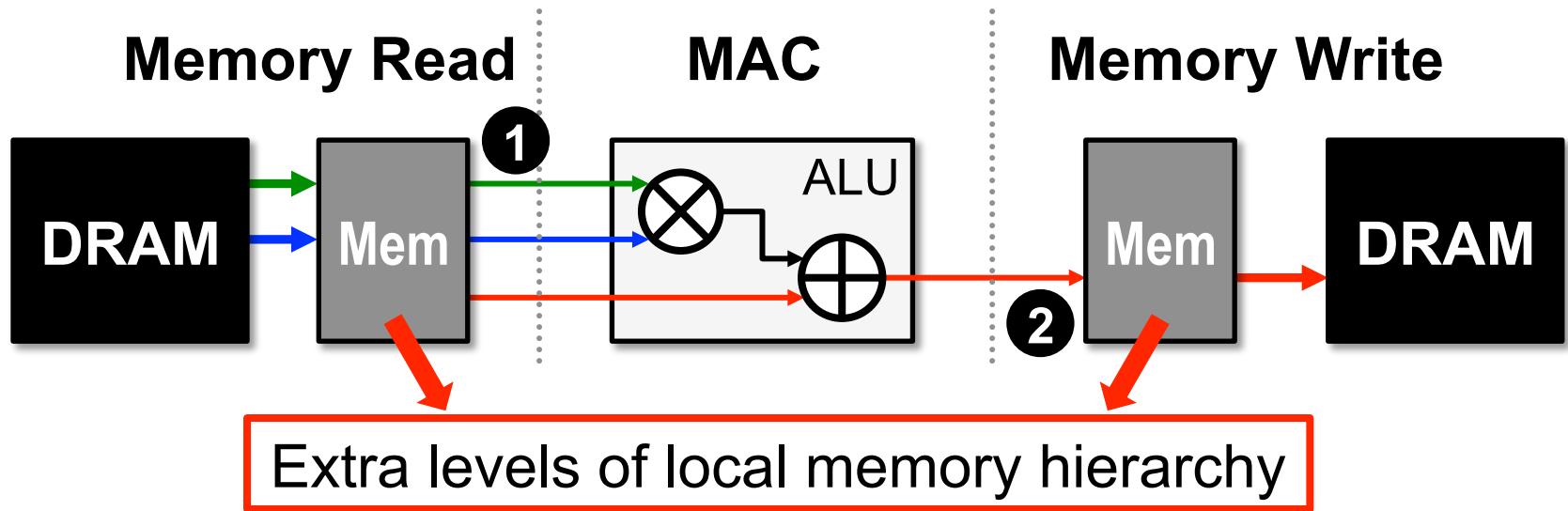


Opportunities: **1** data reuse

- 1** Can reduce DRAM reads of **filter/fmap** by up to **500x\*\***

\*\* AlexNet CONV layers

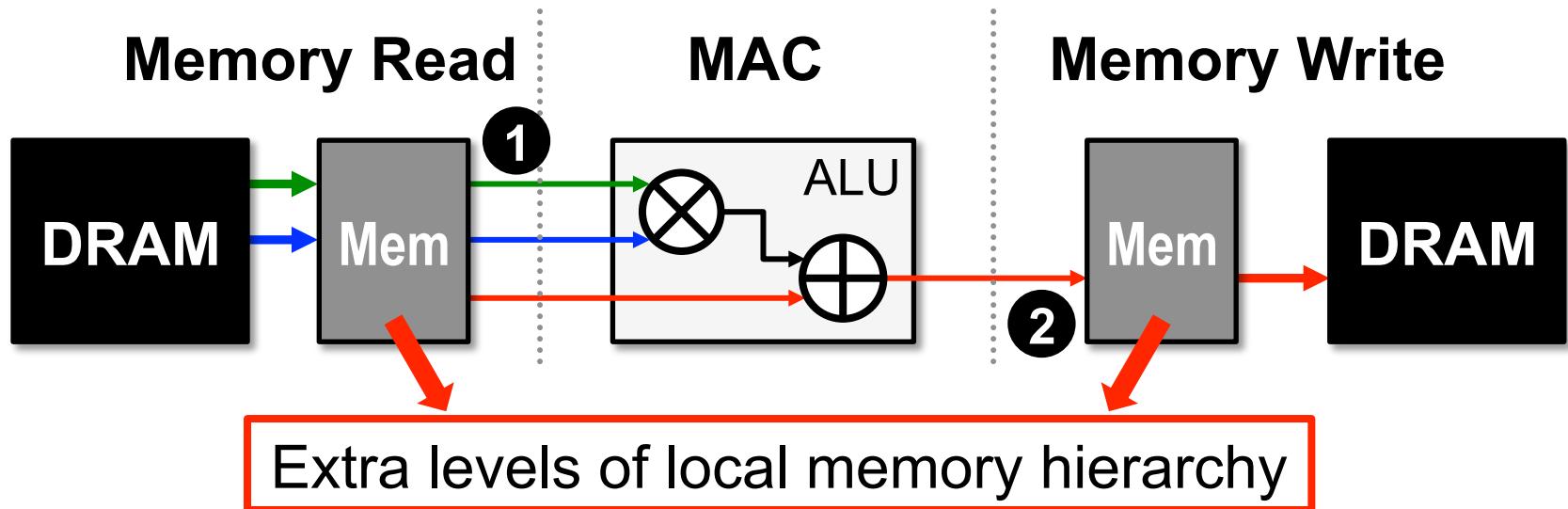
# Memory Access is the Bottleneck



Opportunities: **1** data reuse **2** local accumulation

- 1** Can reduce DRAM reads of **filter/fmap** by up to **500x**
- 2** **Partial sum** accumulation does **NOT** have to access DRAM

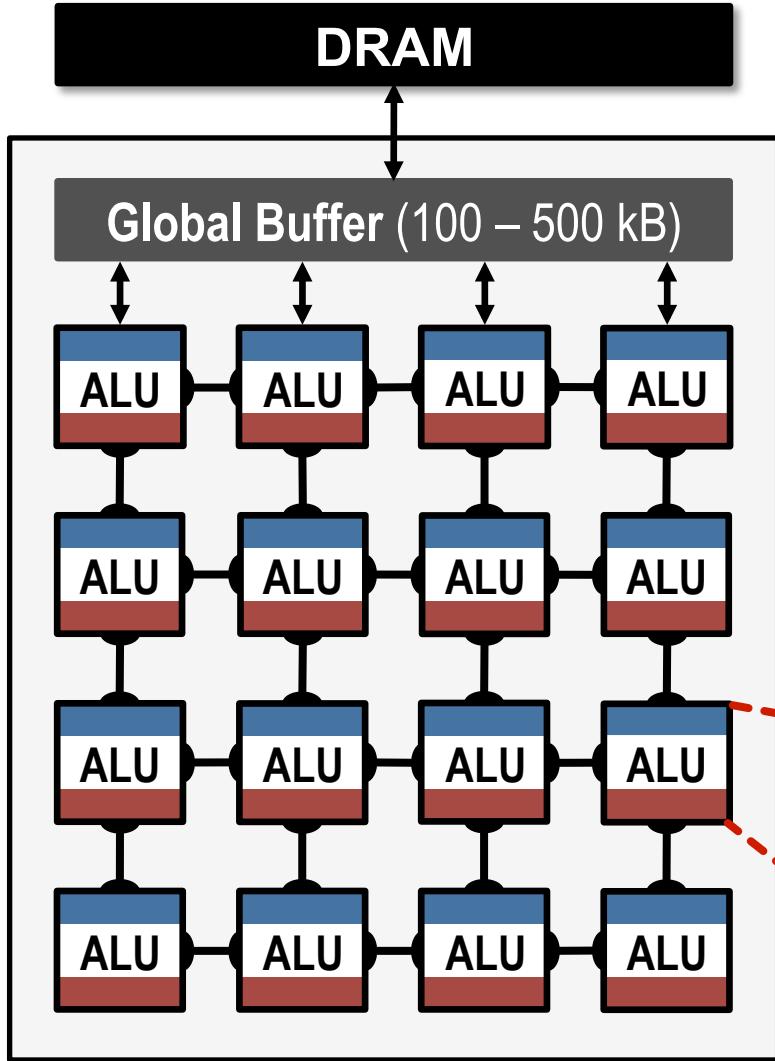
# Memory Access is the Bottleneck



Opportunities: **1** data reuse **2** local accumulation

- 1** Can reduce DRAM reads of **filter/fmap** by up to **500x**
  - 2** **Partial sum** accumulation does **NOT** have to access DRAM
- Example: DRAM access in AlexNet can be reduced from **2896M** to **61M** (best case)

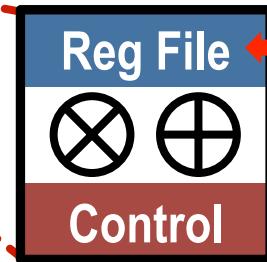
# Spatial Architecture for DNN



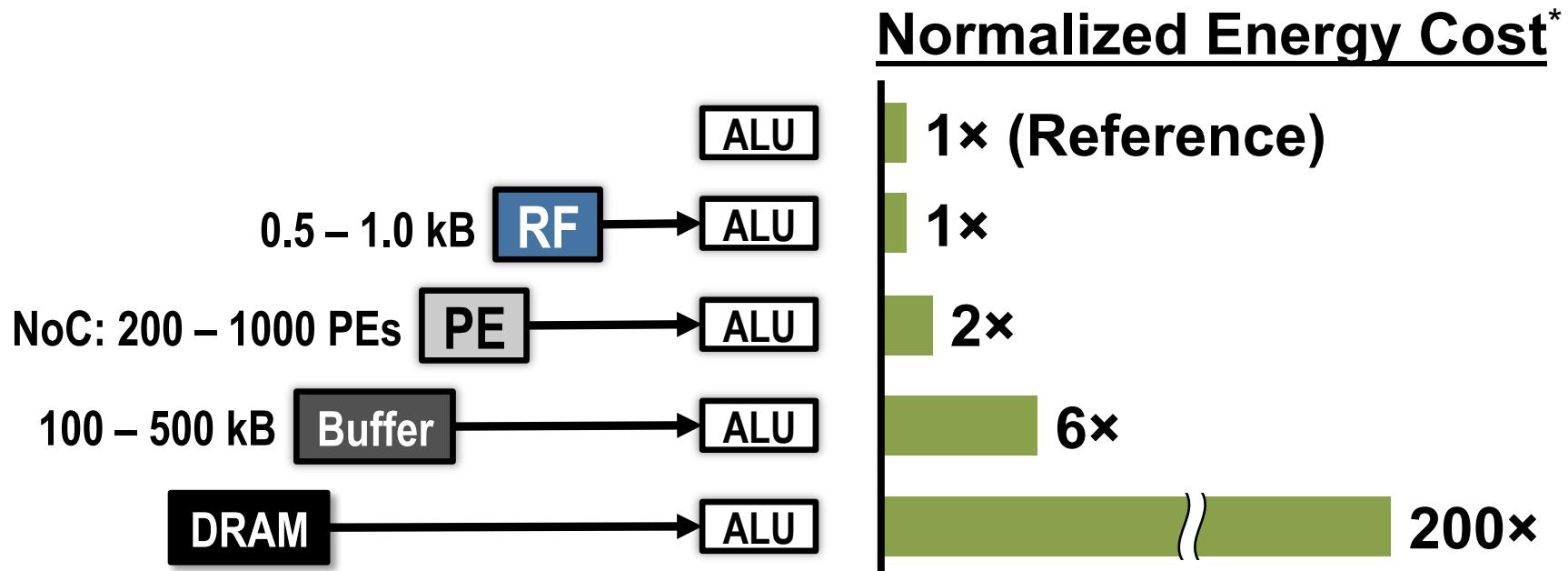
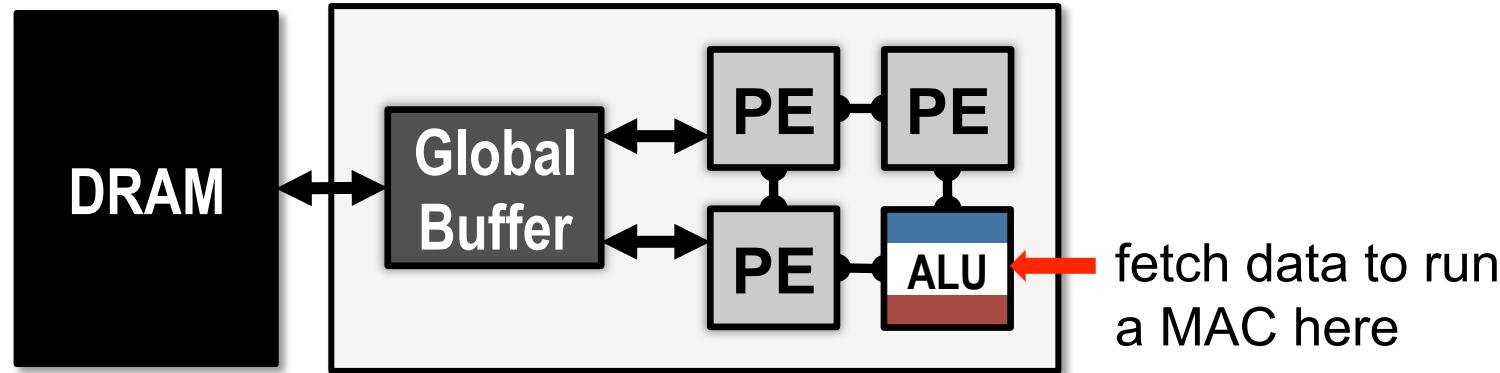
## Local Memory Hierarchy

- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

## Processing Element (PE)

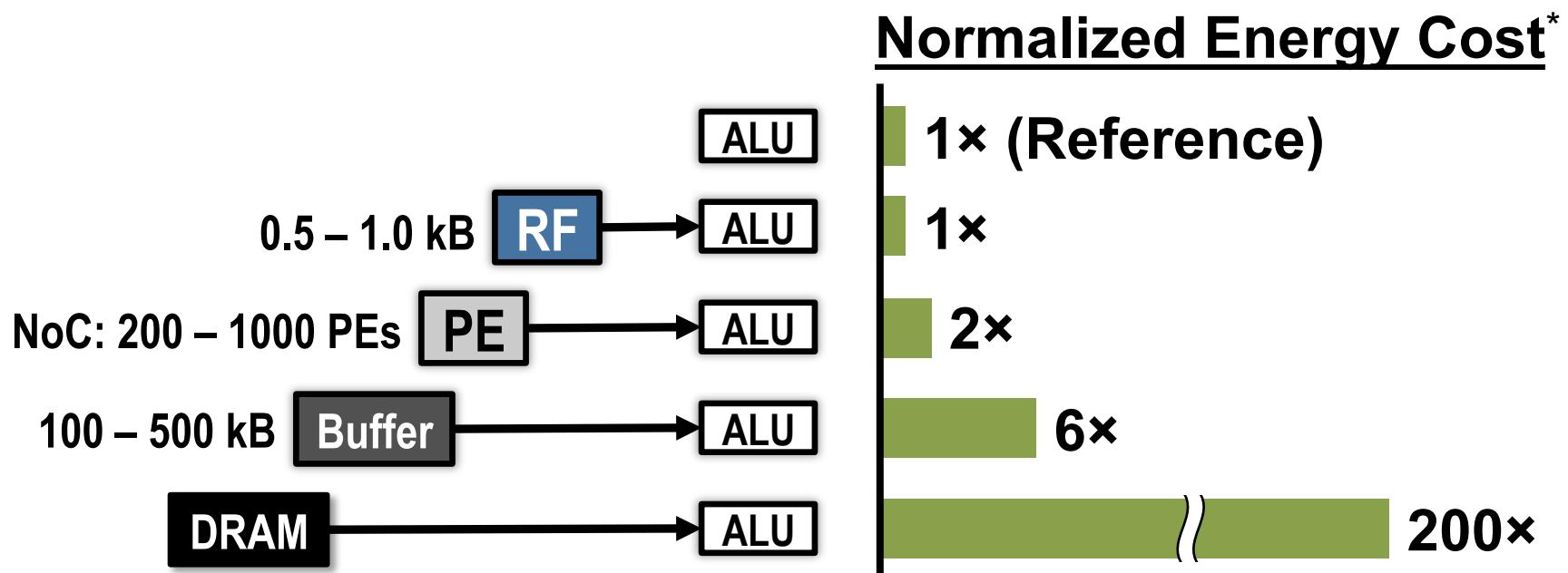


# Low-Cost Local Data Access



# Low-Cost Local Data Access

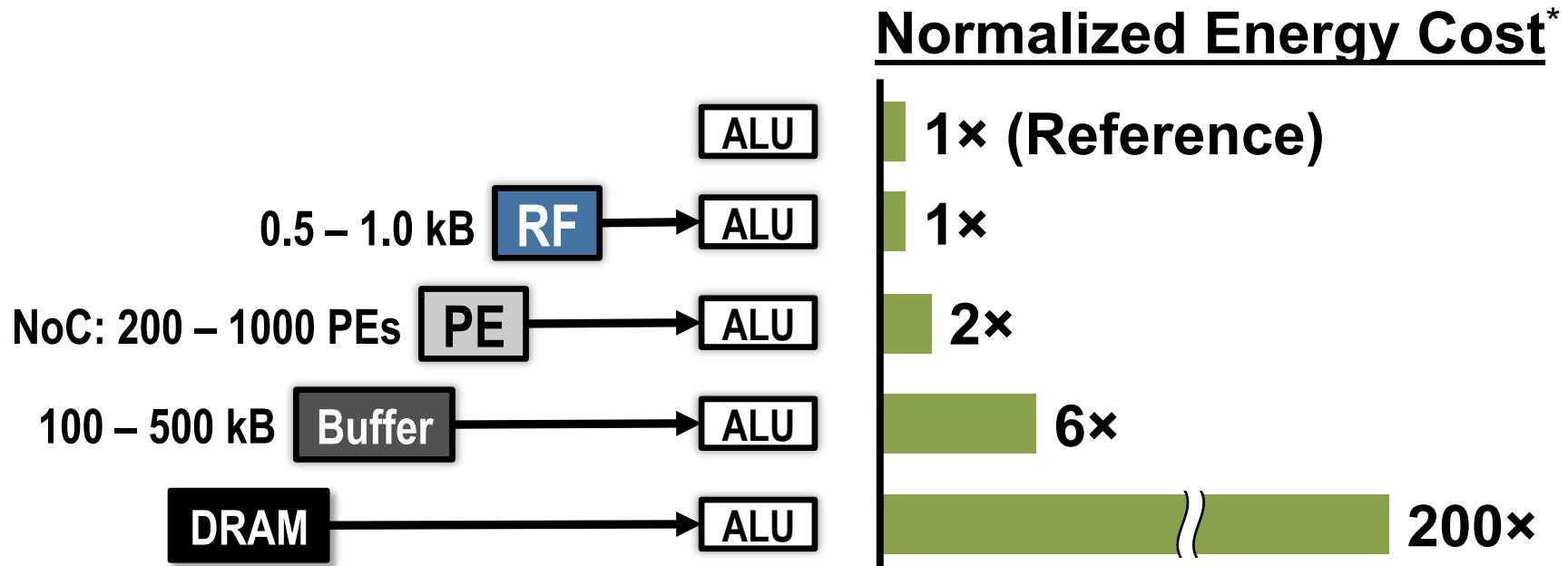
How to exploit ① data reuse and ② local accumulation with *limited* low-cost local storage?



# Low-Cost Local Data Access

How to exploit ① data reuse and ② local accumulation with *limited* low-cost local storage?

specialized processing dataflow required!

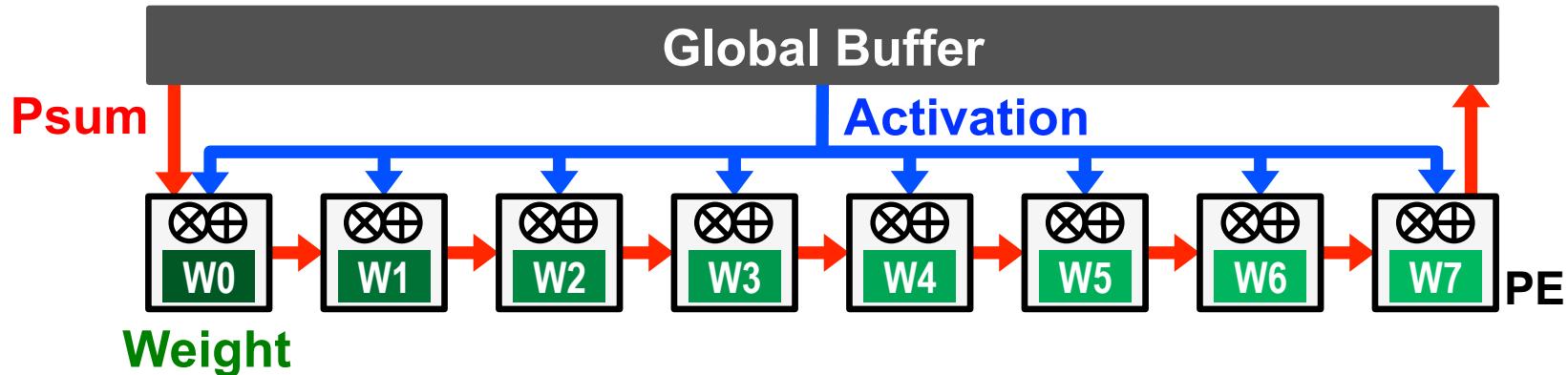


\* measured from a commercial 65nm process

# Dataflow Taxonomy

- Weight Stationary (WS)
- Output Stationary (OS)
- No Local Reuse (NLR)

# Weight Stationary (WS)

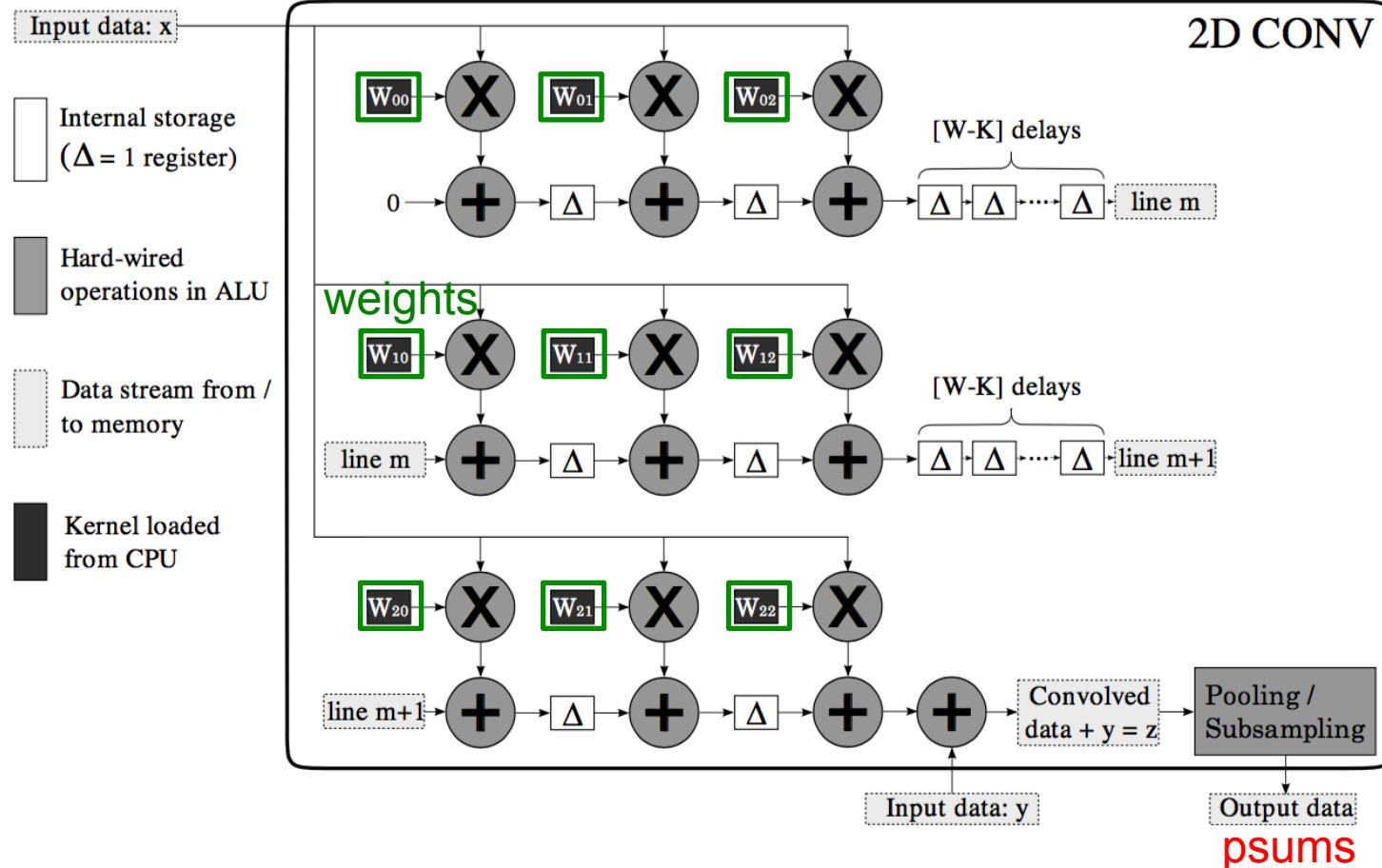


- **Minimize weight** read energy consumption
  - maximize convolutional and filter reuse of weights
- **Broadcast activations** and **accumulate psums** spatially across the PE array.

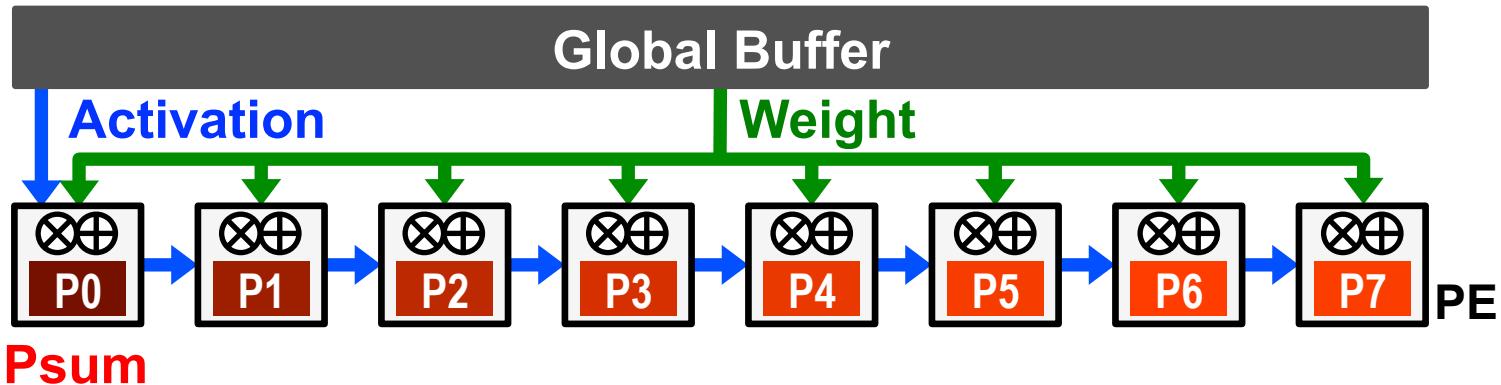
# WS Example: nn-X (NeuFlow)

## A $3 \times 3$ 2D Convolution Engine

activations



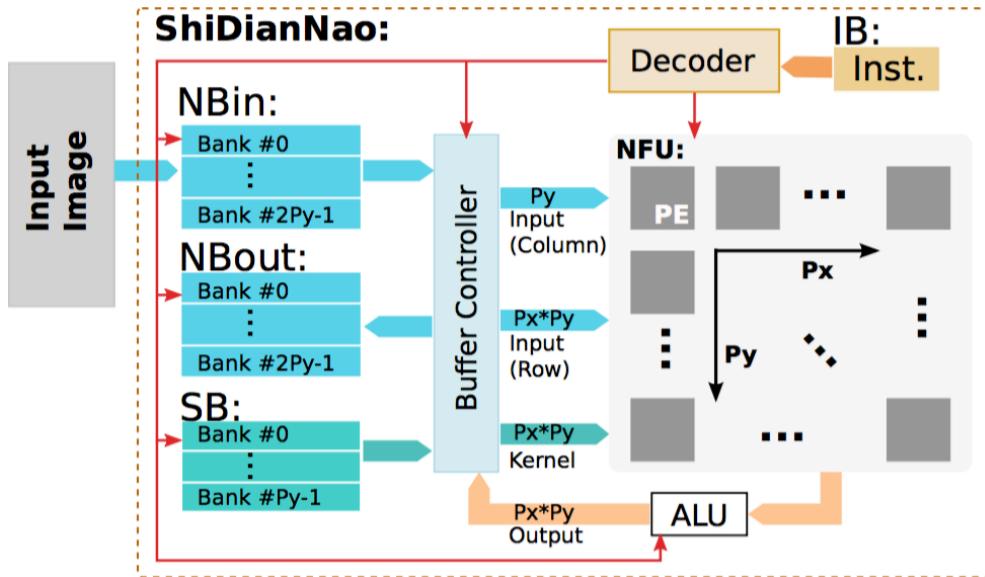
# Output Stationary (OS)



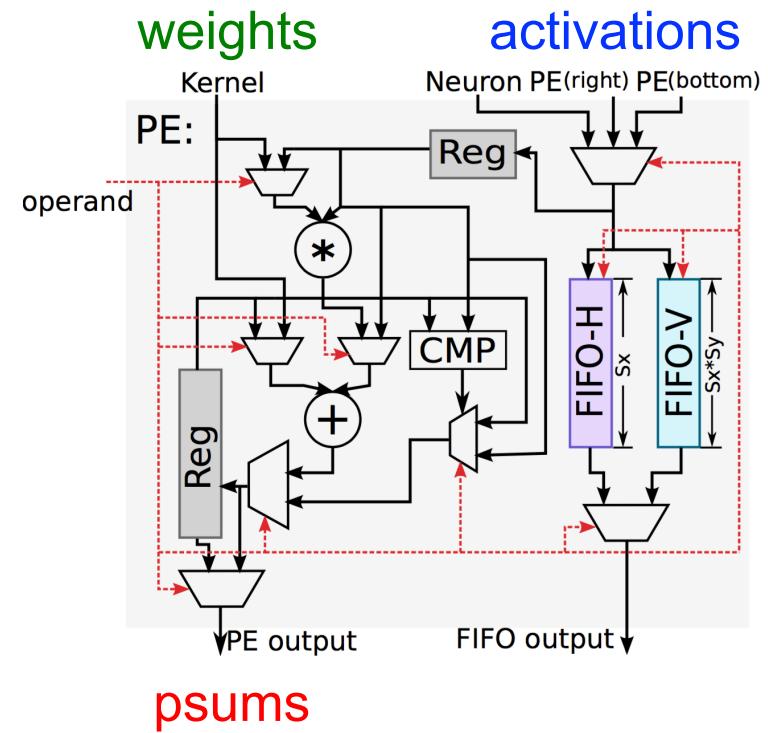
- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation
- **Broadcast/Multicast filter weights** and **reuse activations** spatially across the PE array

# OS Example: ShiDianNao

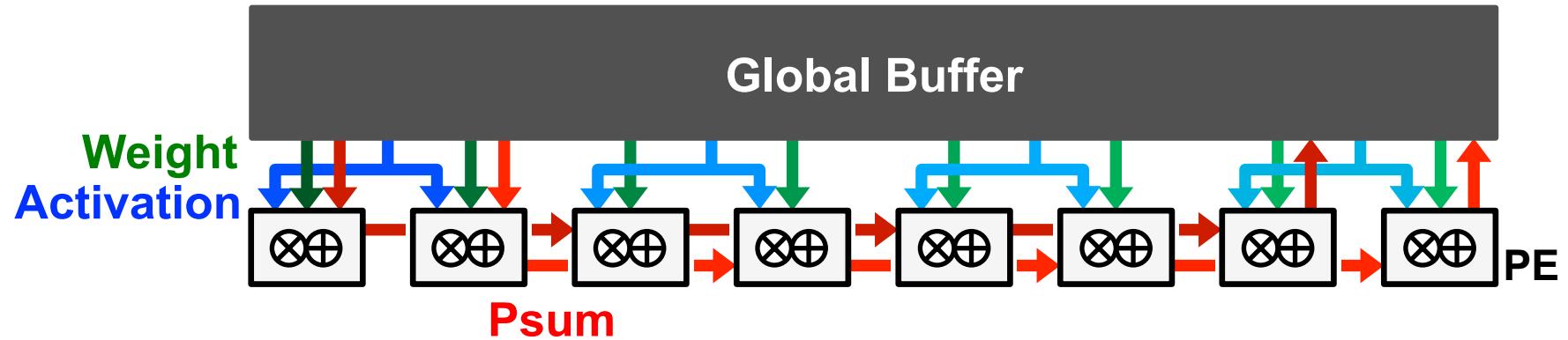
## Top-Level Architecture



## PE Architecture

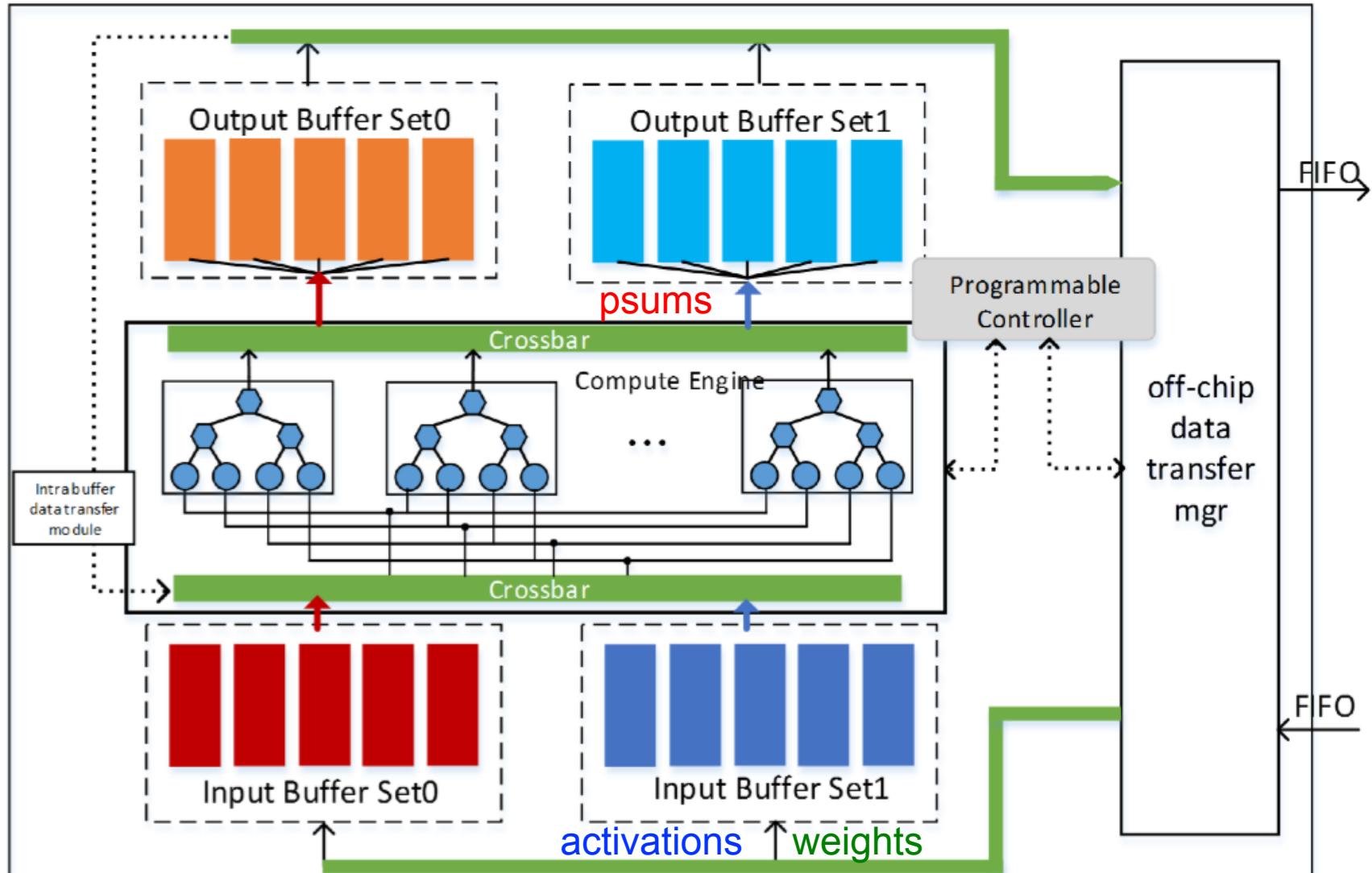


# No Local Reuse (NLR)



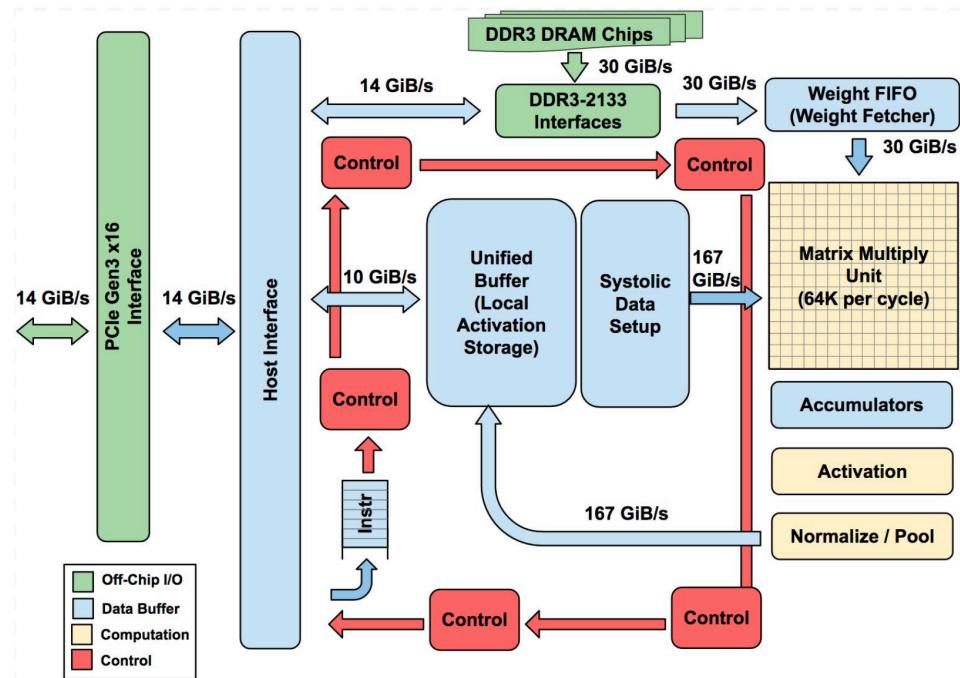
- Use a **large global buffer** as shared storage
  - Reduce **DRAM** access energy consumption
- Multicast **activations**, single-cast **weights**, and accumulate **psums** spatially across the PE array

# NLR Example: UCLA

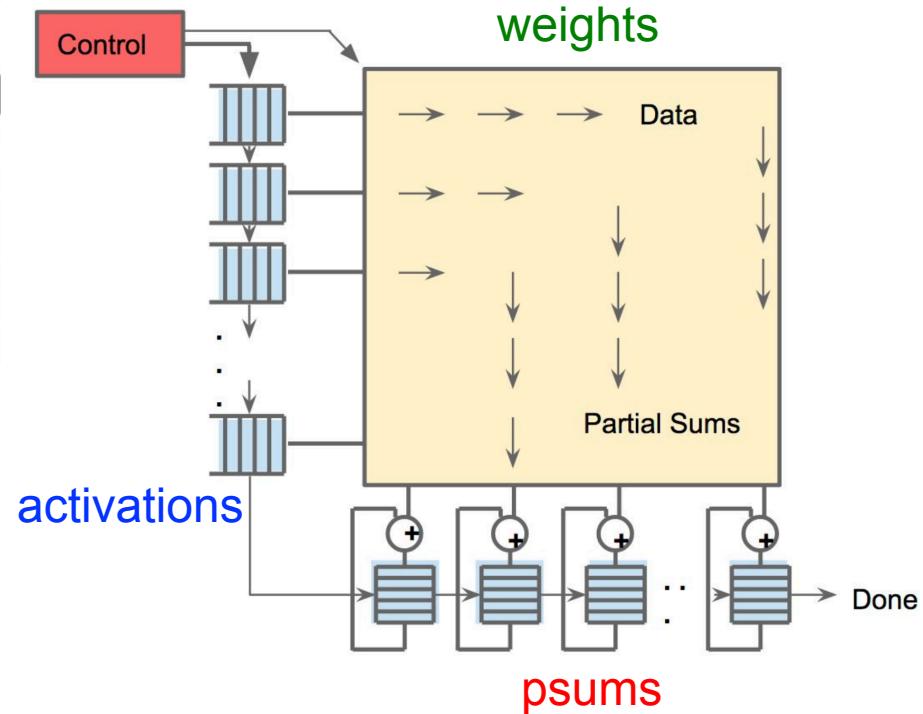


# NLR Example: TPU

## Top-Level Architecture



## Matrix Multiply Unit



# Taxonomy: More Examples

---

- **Weight Stationary (WS)**

[Chakradhar, *ISCA* 2010] [nn-X (NeuFlow), *CVPRW* 2014]

[Park, *ISSCC* 2015] [ISAAC, *ISCA* 2016] [PRIME, *ISCA* 2016]

- **Output Stationary (OS)**

[Peemen, *ICCD* 2013] [ShiDianNao, *ISCA* 2015]

[Gupta, *ICML* 2015] [Moons, *VLSI* 2016]

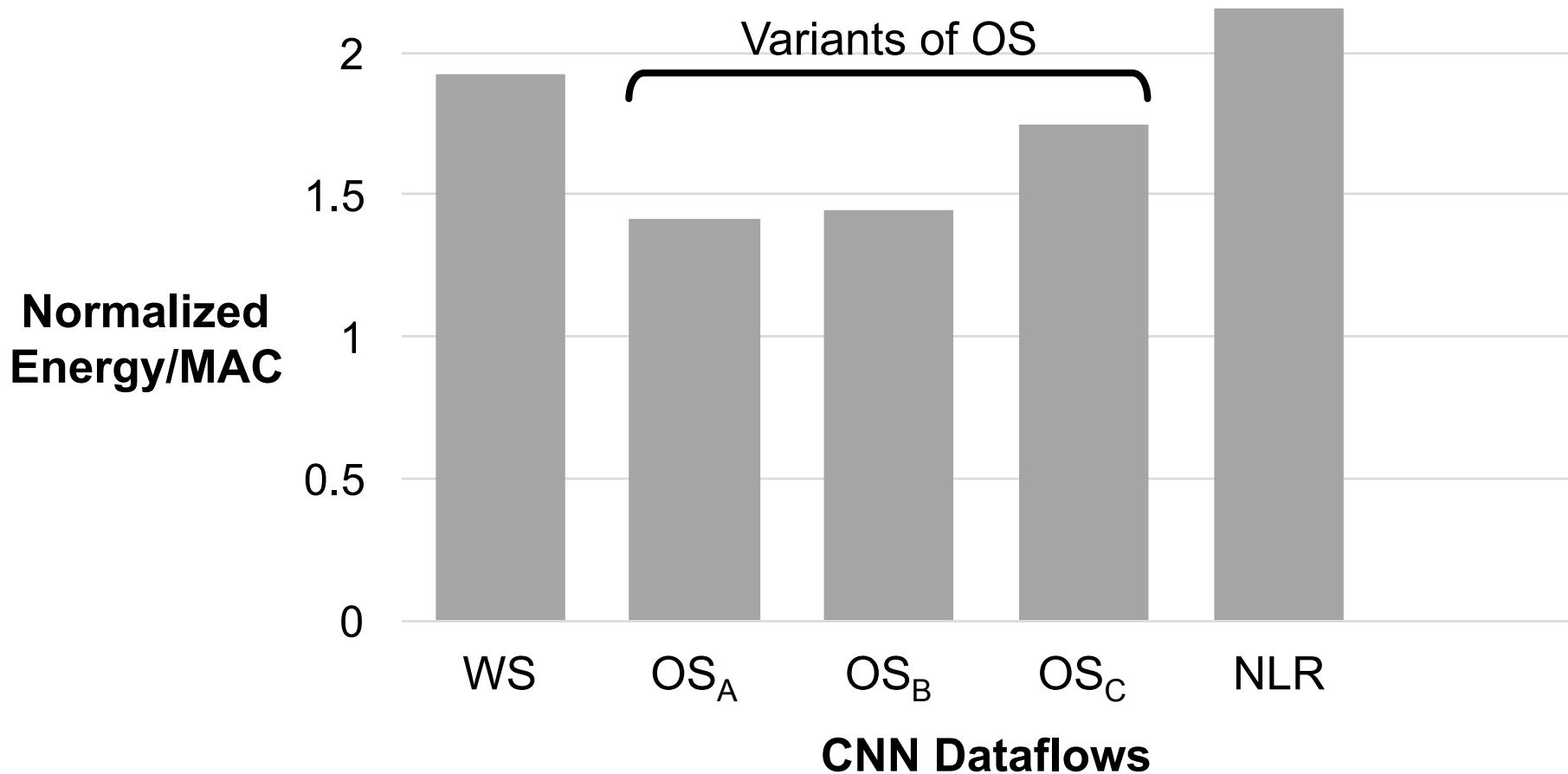
- **No Local Reuse (NLR)**

[DianNao, *ASPLOS* 2014] [DaDianNao, *MICRO* 2014]

[Zhang, *FPGA* 2015] [TPU, *ISCA* 2017]

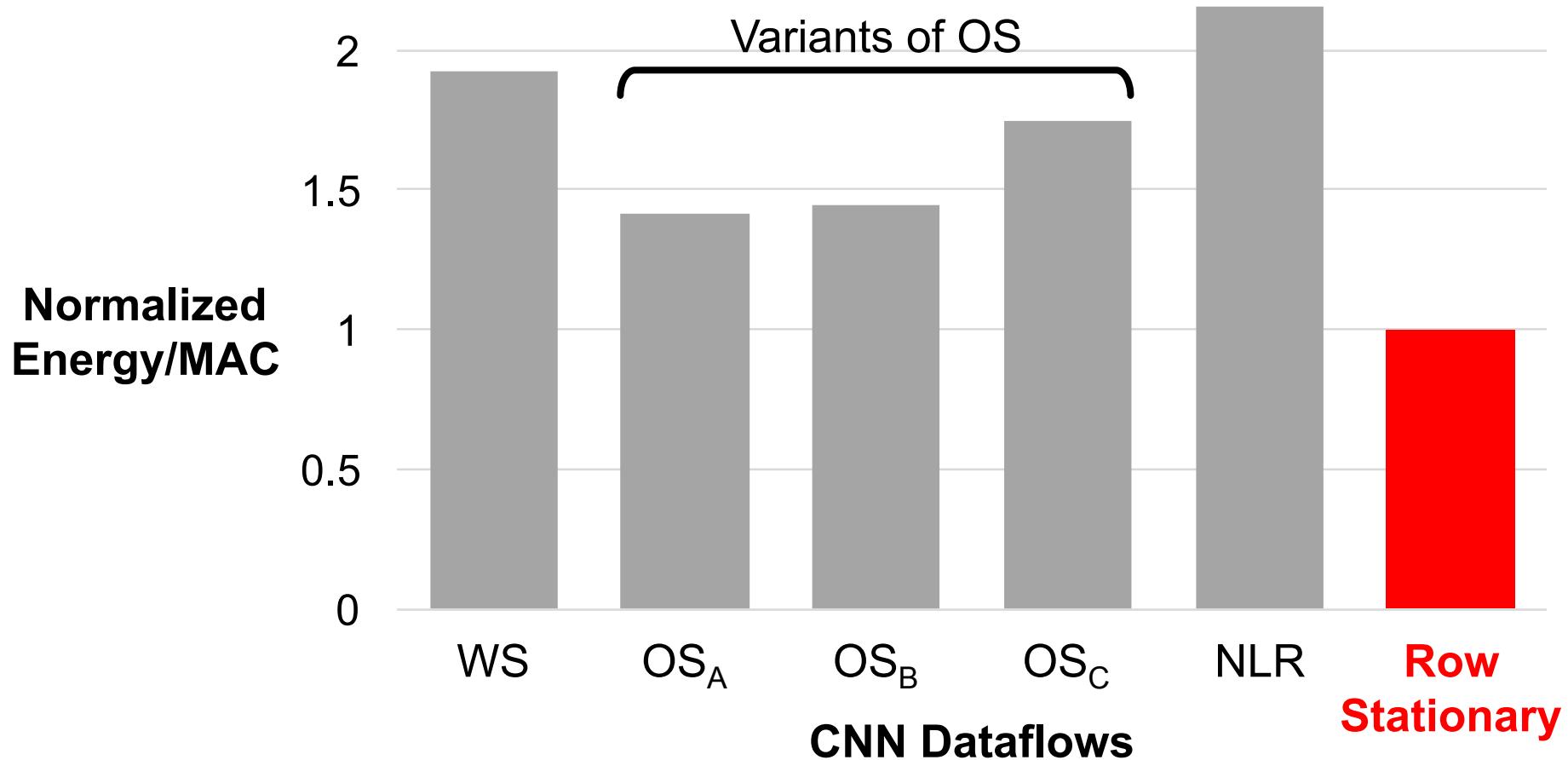
# Energy Efficiency Comparison

- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16



# Energy Efficiency Comparison

- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16

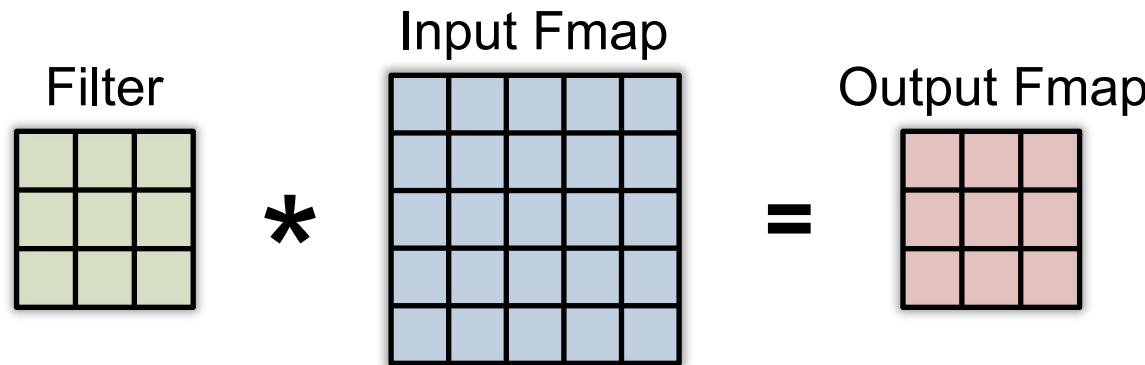


# Energy-Efficient Dataflow: Row Stationary (RS)

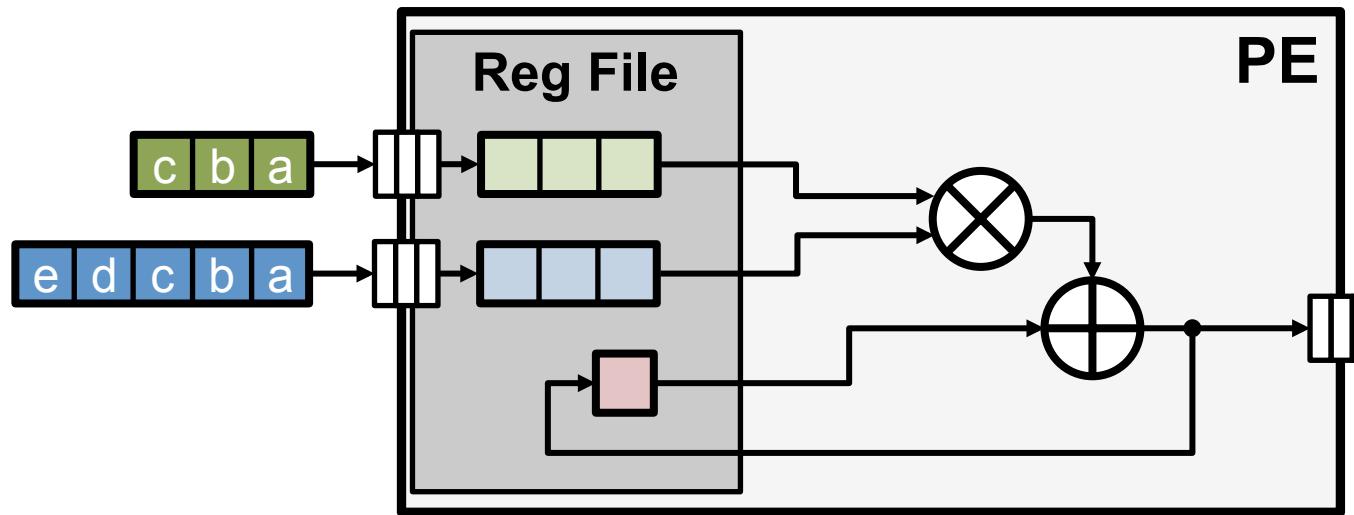
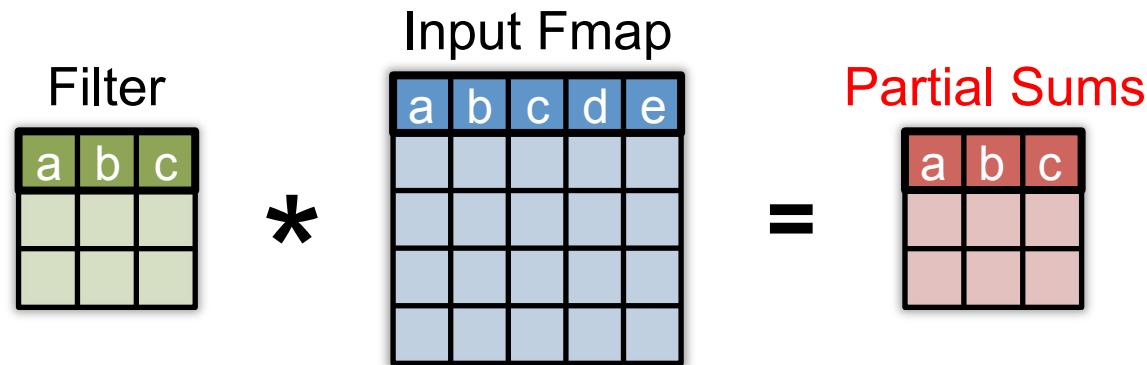
- **Maximize** reuse and accumulation at **RF**
- Optimize for **overall** energy efficiency instead for *only* a certain data type

# Row Stationary: Energy-efficient Dataflow

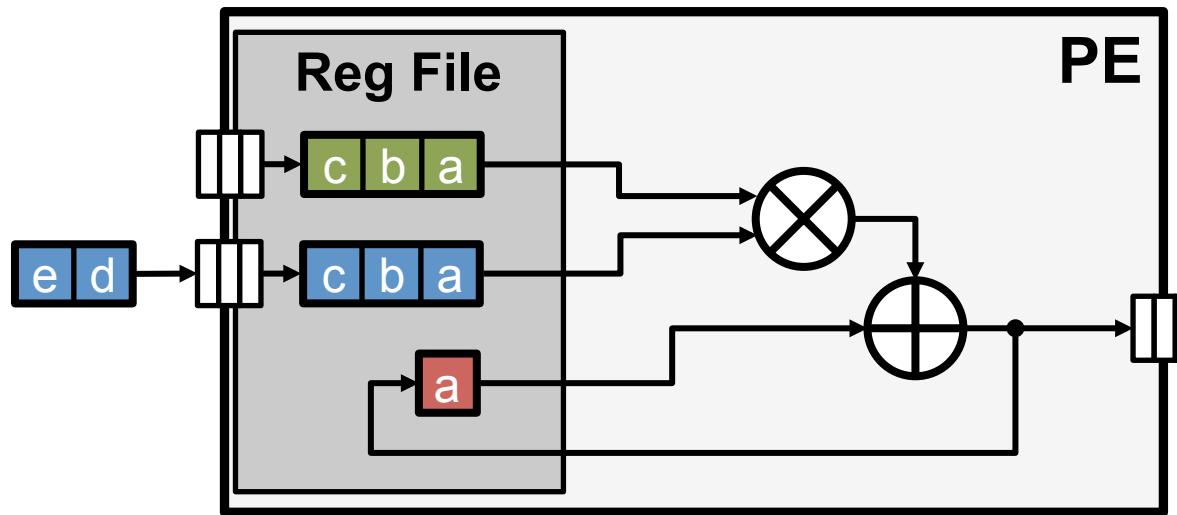
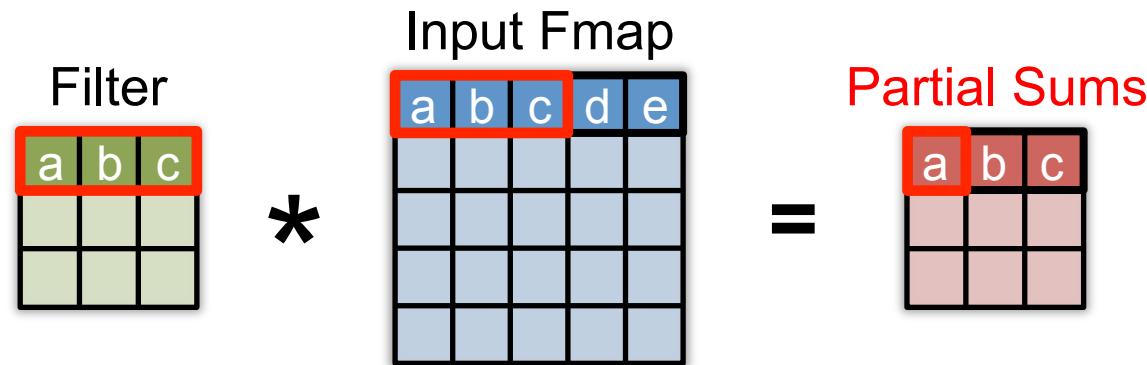
---



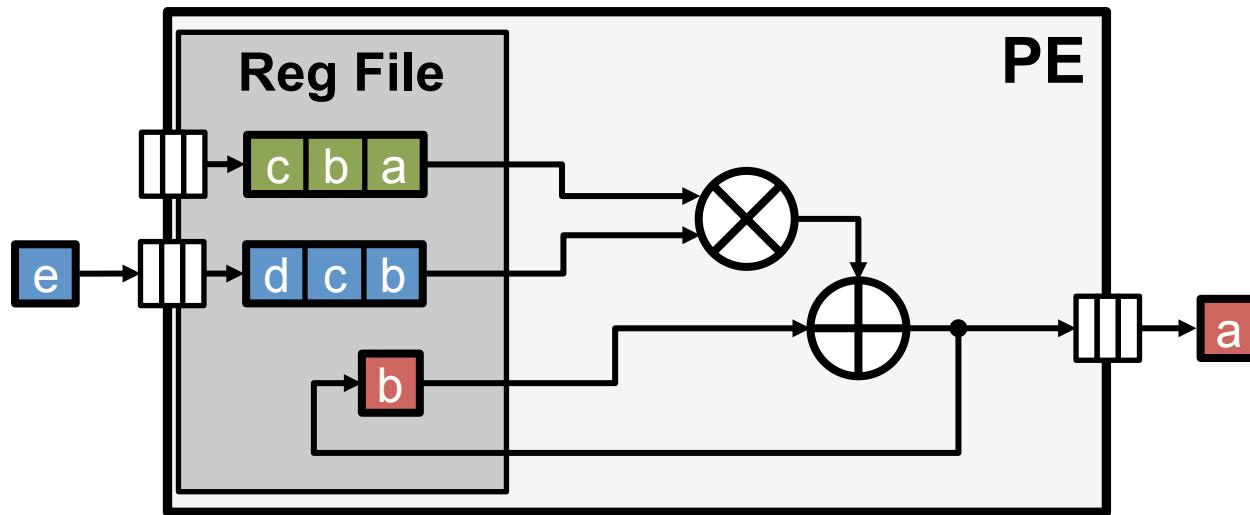
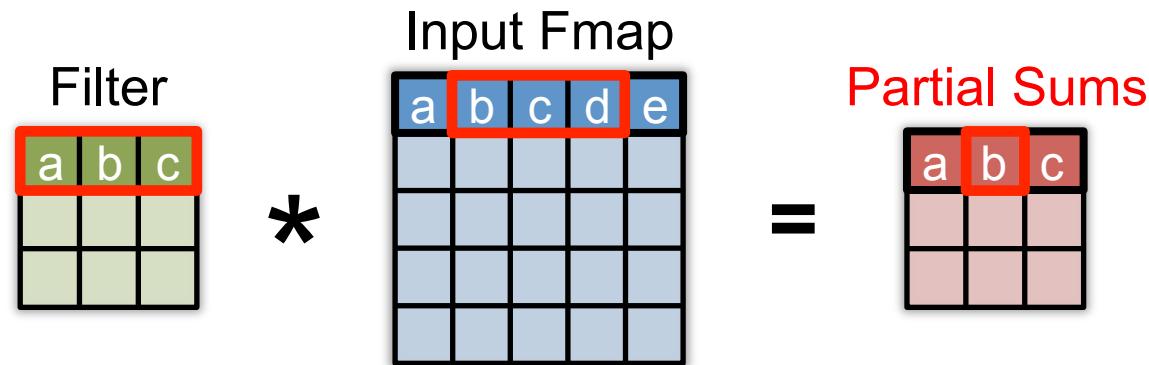
# 1D Row Convolution in PE



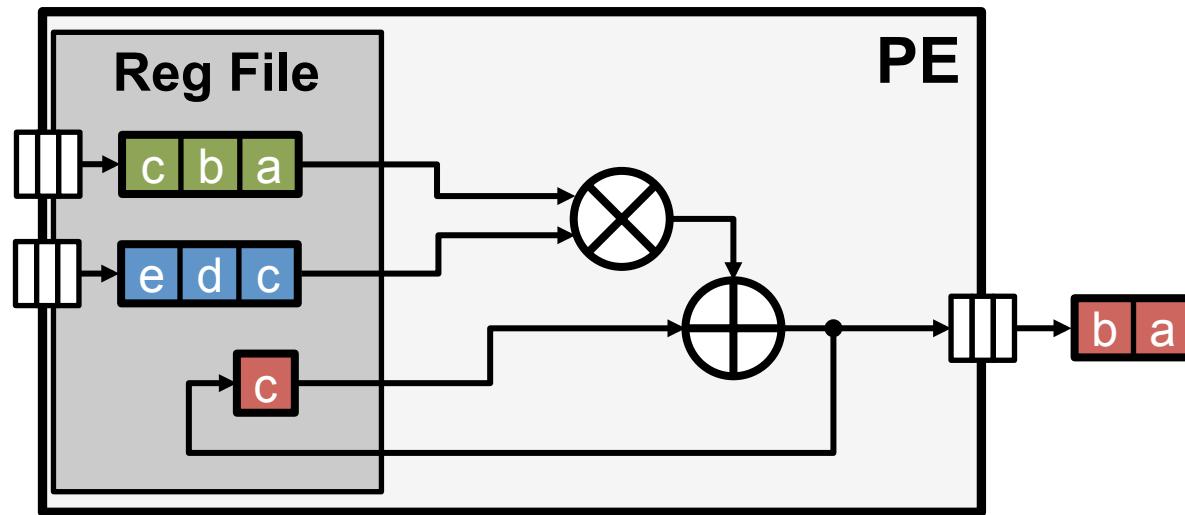
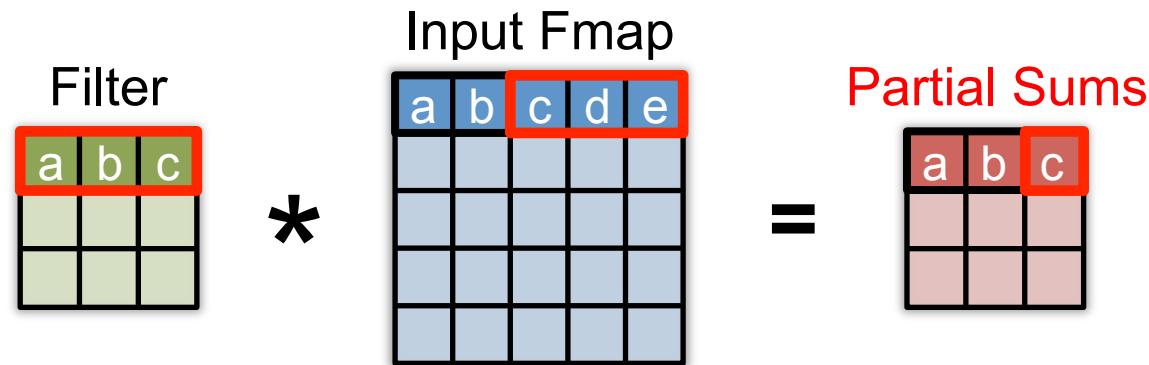
# 1D Row Convolution in PE



# 1D Row Convolution in PE

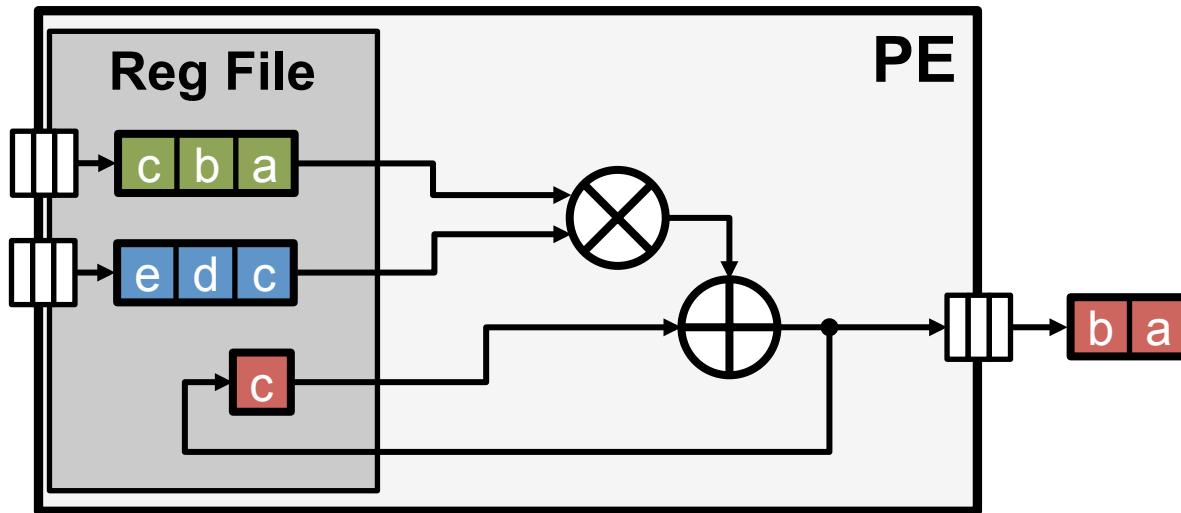


# 1D Row Convolution in PE



# 1D Row Convolution in PE

- Maximize row **convolutional reuse** in RF
  - Keep a **filter** row and **fmap** sliding window in RF
- Maximize row **psum** accumulation in RF



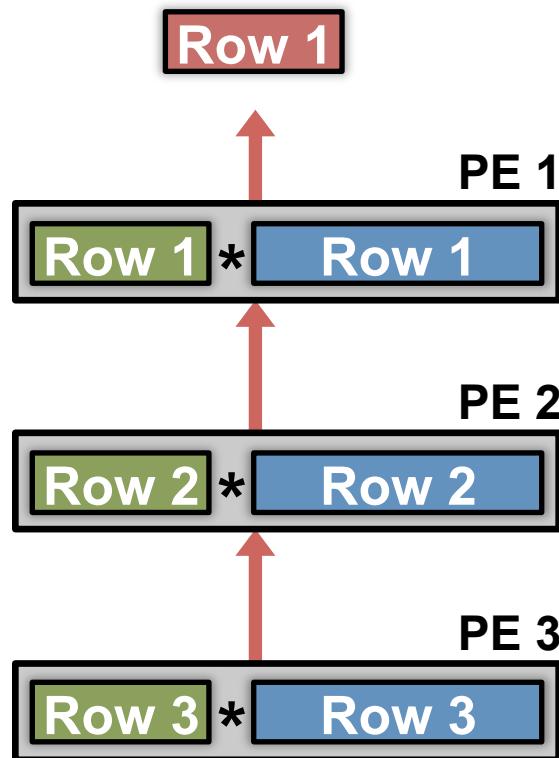
# 2D Convolution in PE Array

---



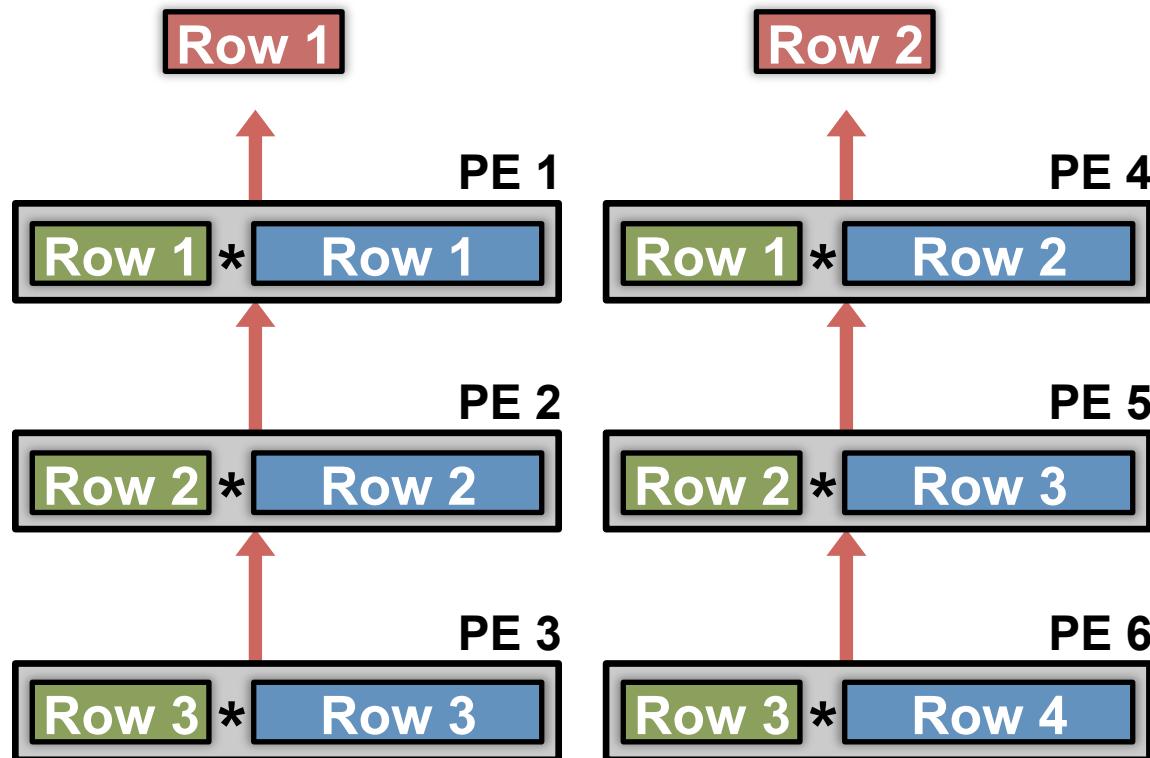
$$\begin{bmatrix} \text{green} \\ \text{green} \\ \text{green} \end{bmatrix} * \begin{bmatrix} \text{blue} & \text{blue} & \text{blue} \\ \text{blue} & \text{blue} & \text{blue} \\ \text{blue} & \text{blue} & \text{blue} \end{bmatrix} = \begin{bmatrix} \text{red} & \text{red} & \text{red} \\ \text{red} & \text{red} & \text{red} \end{bmatrix}$$

# 2D Convolution in PE Array



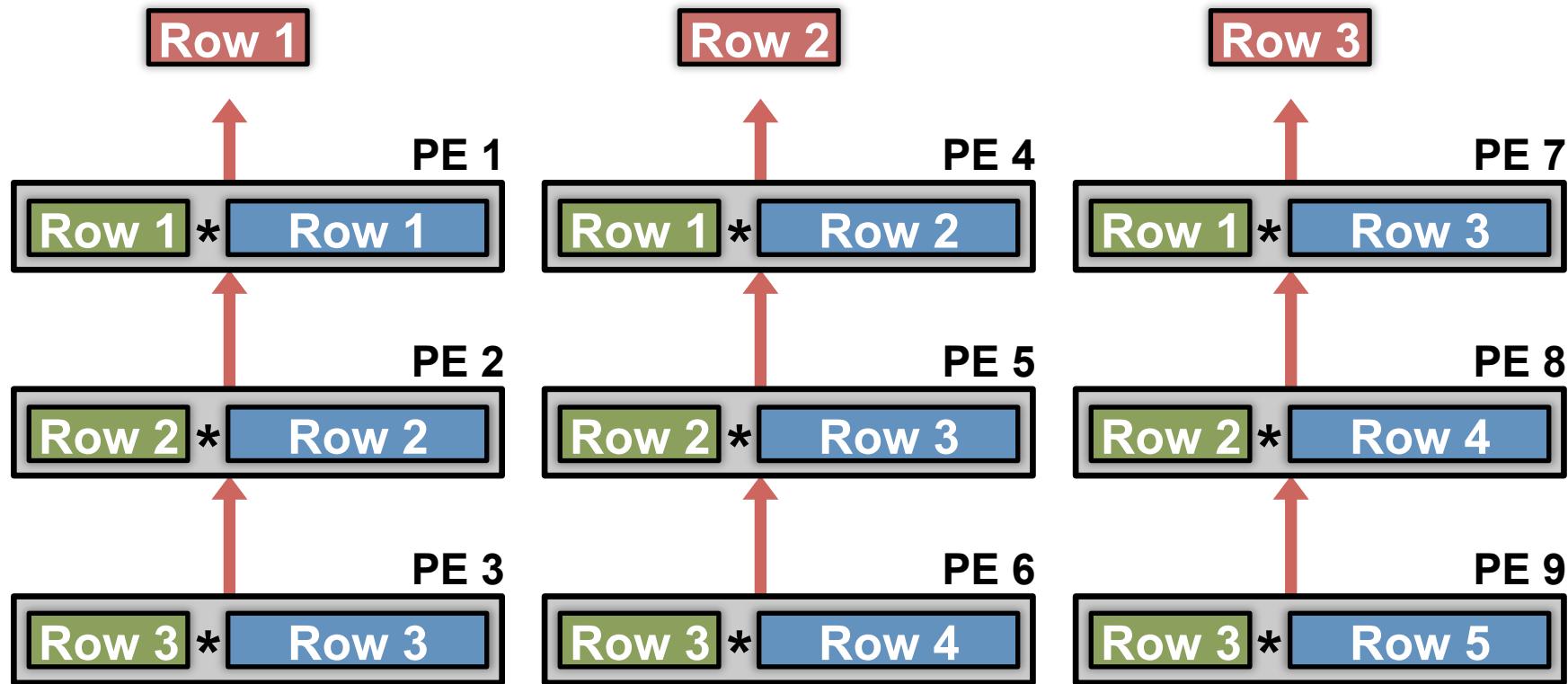
The diagram shows the result of a 2D convolution operation. On the left, a 3x3 input feature map (green squares) is multiplied by a 3x3 kernel (blue squares) using an asterisk (\*) to denote convolution. The result is an equals sign (=) followed by a 3x3 output feature map (red and pink squares), representing the computed feature map.

# 2D Convolution in PE Array



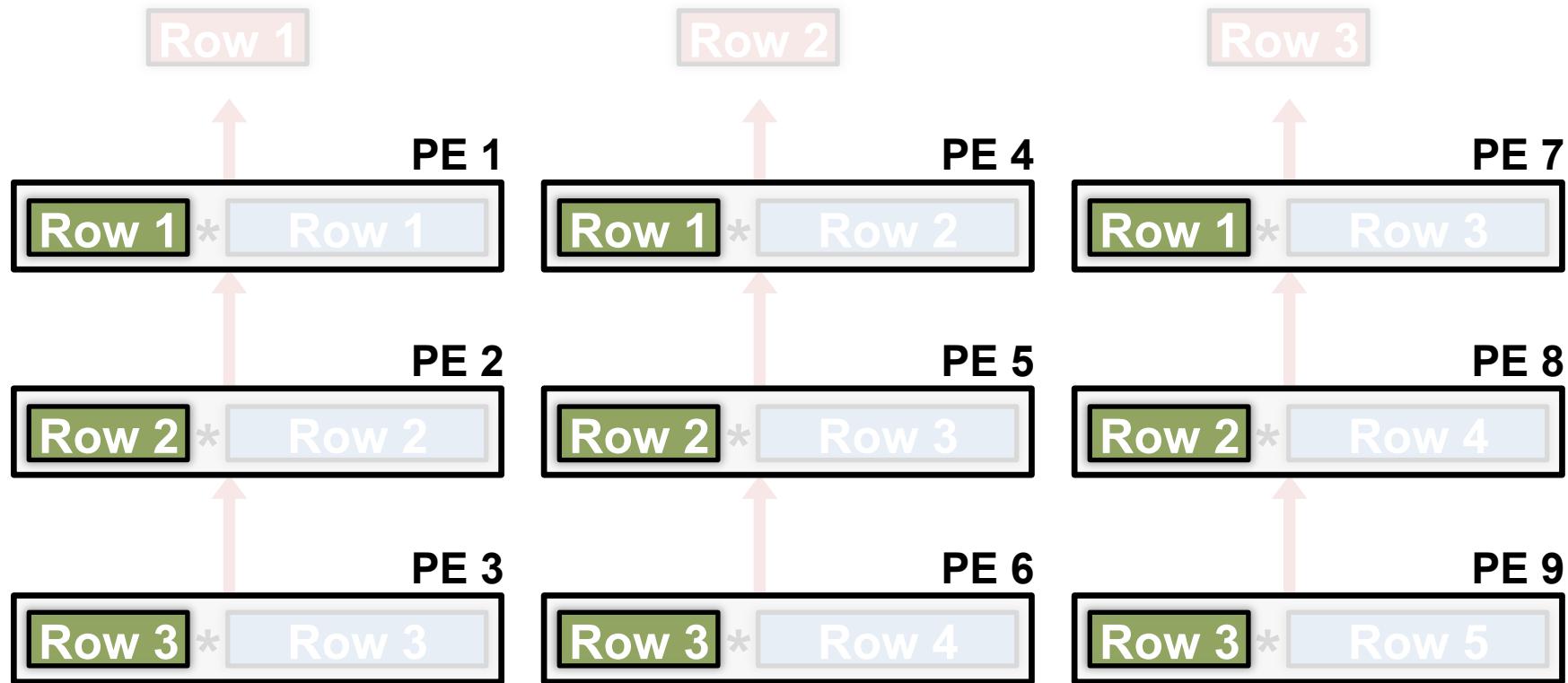
$$\begin{matrix} \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{matrix} * \begin{matrix} \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \\ \text{Row 4} \end{matrix} = \begin{matrix} \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \\ \text{Row 4} \end{matrix}$$

# 2D Convolution in PE Array



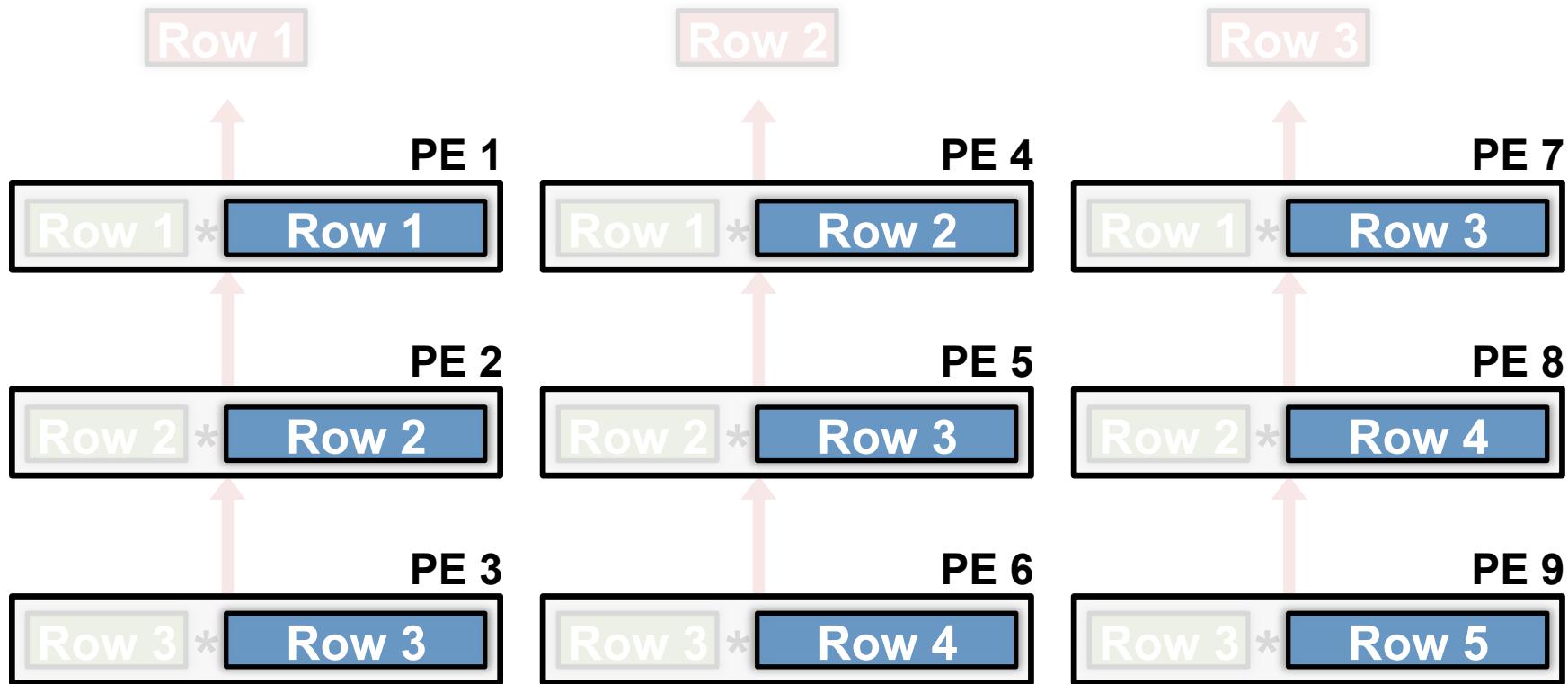
$$\begin{array}{c} \text{green} \end{array} \times \begin{array}{c} \text{blue} \end{array} = \begin{array}{c} \text{red} \end{array} \quad \begin{array}{c} \text{green} \end{array} \times \begin{array}{c} \text{blue} \end{array} = \begin{array}{c} \text{red} \end{array} \quad \begin{array}{c} \text{green} \end{array} \times \begin{array}{c} \text{blue} \end{array} = \begin{array}{c} \text{red} \end{array}$$

# Convolutional Reuse Maximized



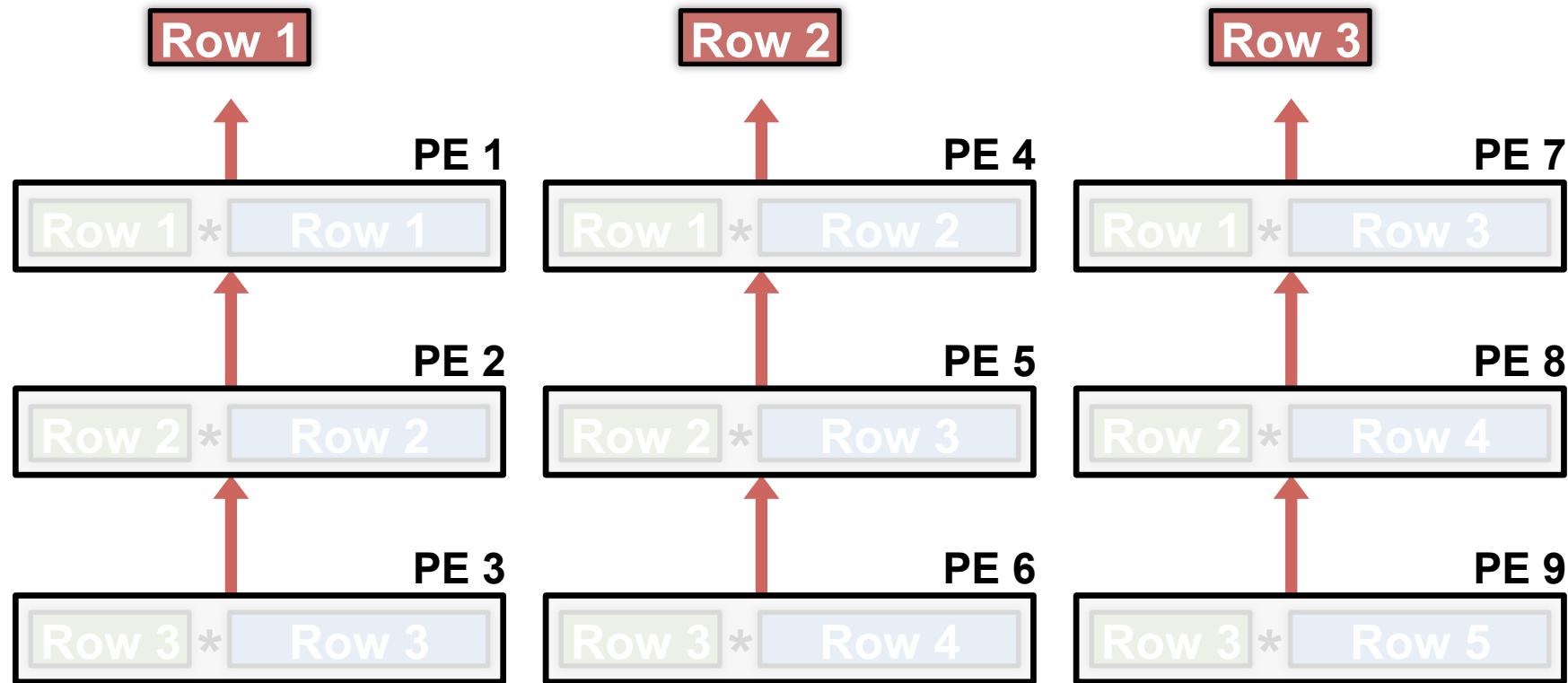
**Filter rows** are reused across PEs **horizontally**

# Convolutional Reuse Maximized



**Fmap rows** are reused across PEs **diagonally**

# Maximize 2D Accumulation in PE Array



**Partial sums** accumulate across PEs **vertically**

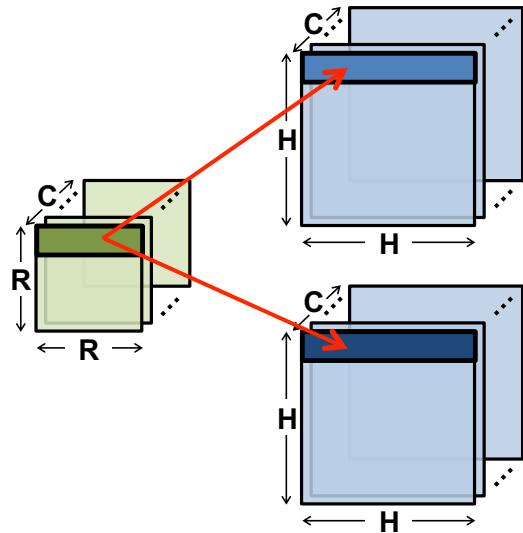
# Dimensions Beyond 2D Convolution

---

- 1 Multiple Fmaps
- 2 Multiple Filters
- 3 Multiple Channels

# Filter Reuse in PE

## 1 Multiple Fmaps



## 2 Multiple Filters

## 3 Multiple Channels

Channel 1      Filter 1      Fmap 1      Psum 1

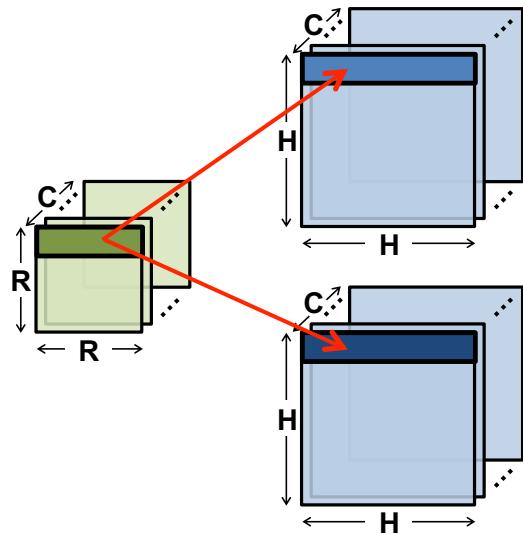
$$\text{Row 1} * \text{Row 1} = \text{Row 1}$$

Channel 1      Filter 1      Fmap 2      Psum 2

$$\text{Row 1} * \text{Row 1} = \text{Row 1}$$

# Filter Reuse in PE

## 1 Multiple Fmaps



## 2 Multiple Filters

## 3 Multiple Channels

### Channel 1

Filter 1

Row 1

Fmap 1

Row 1

Psum 1

Row 1

### Channel 1

Filter 1

Row 1

Fmap 2

Row 1

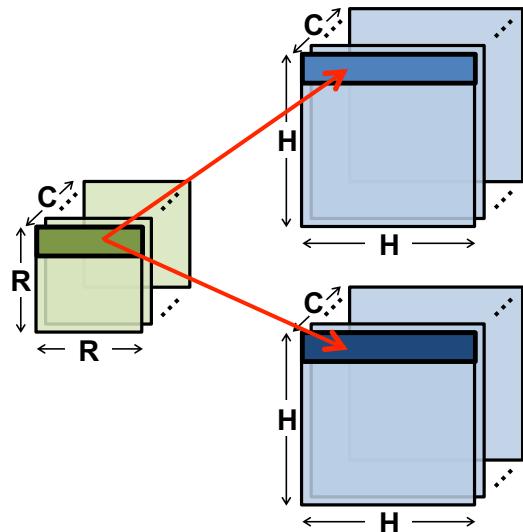
Psum 2

Row 1

share the same filter row

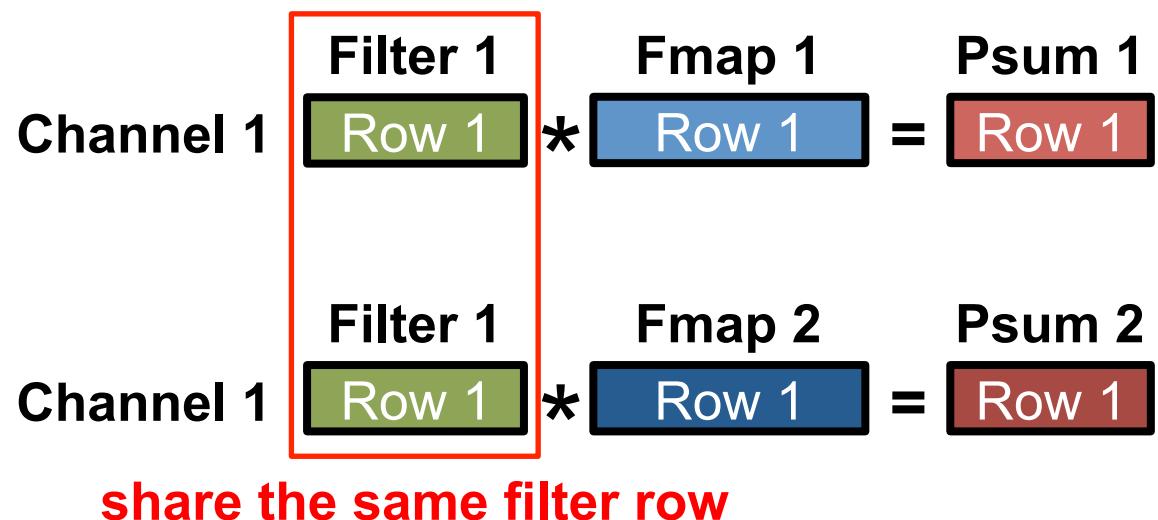
# Filter Reuse in PE

## 1 Multiple Fmaps



## 2 Multiple Filters

## 3 Multiple Channels

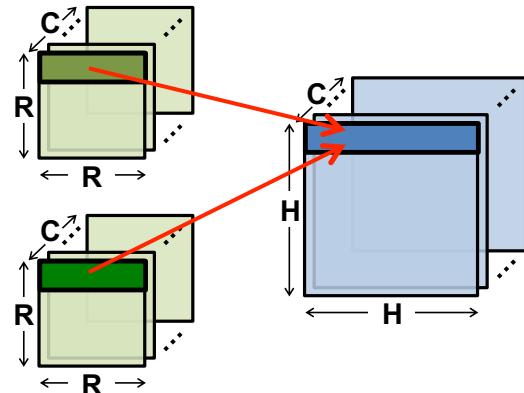


Processing in PE: concatenate fmap rows



# Fmap Reuse in PE

1 Multiple Fmaps



2 Multiple Filters

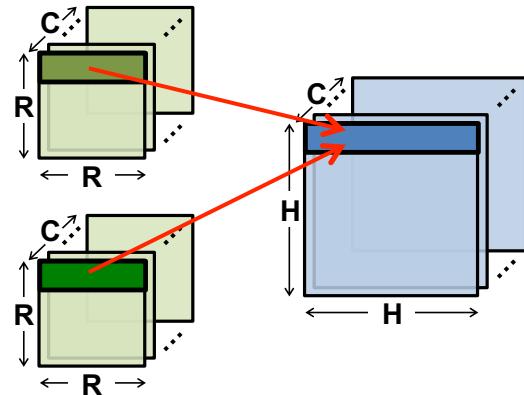
	Filter 1	Fmap 1	Psum 1
Channel 1	Row 1	* Row 1	= Row 1
Channel 1	Filter 2	Fmap 1	Psum 2

The table illustrates the computation of multiple channels using multiple filters. For each channel, the filter is applied to the feature map (Fmap) to produce a partial sum (Psum). The first row shows the computation for Channel 1 using Filter 1, resulting in Psum 1. The second row shows the computation for Channel 1 using Filter 2, resulting in Psum 2.

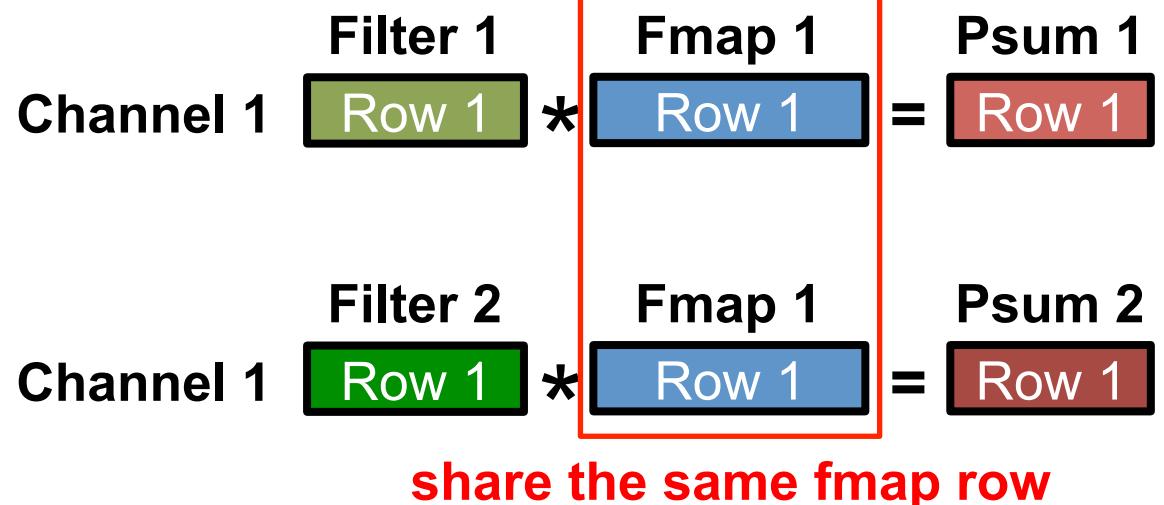
3 Multiple Channels

# Fmap Reuse in PE

1 Multiple Fmaps



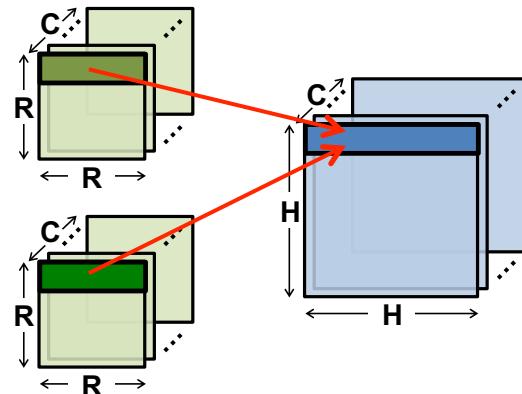
2 Multiple Filters



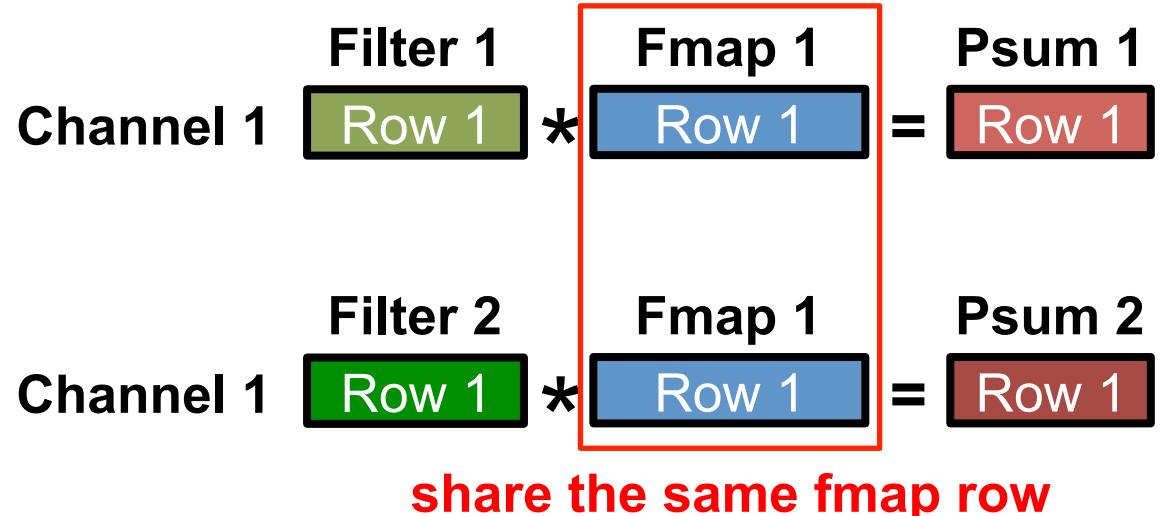
3 Multiple Channels

# Fmap Reuse in PE

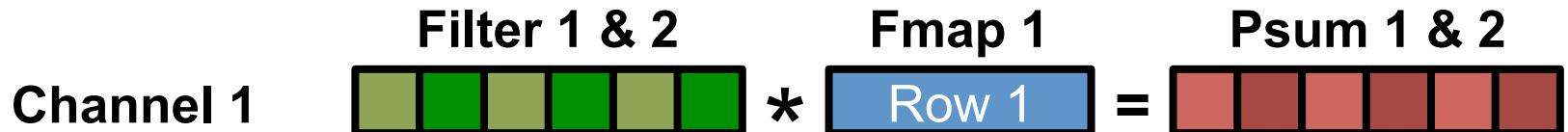
1 Multiple Fmaps



2 Multiple Filters

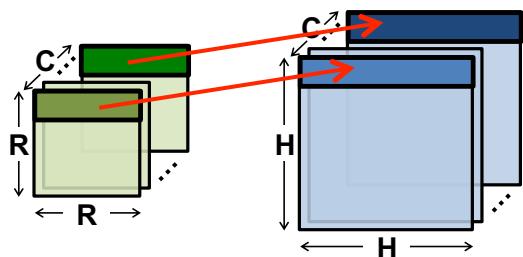


Processing in PE: interleave filter rows



# Channel Accumulation in PE

1 Multiple Fmaps



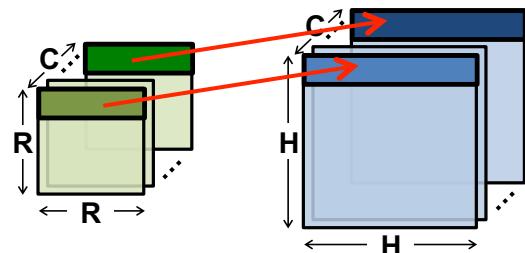
2 Multiple Filters

3 Multiple Channels

	Filter 1	Fmap 1	Psum 1
Channel 1	Row 1	* Row 1	= Row 1
Channel 2	Row 1	* Row 1	= Row 1

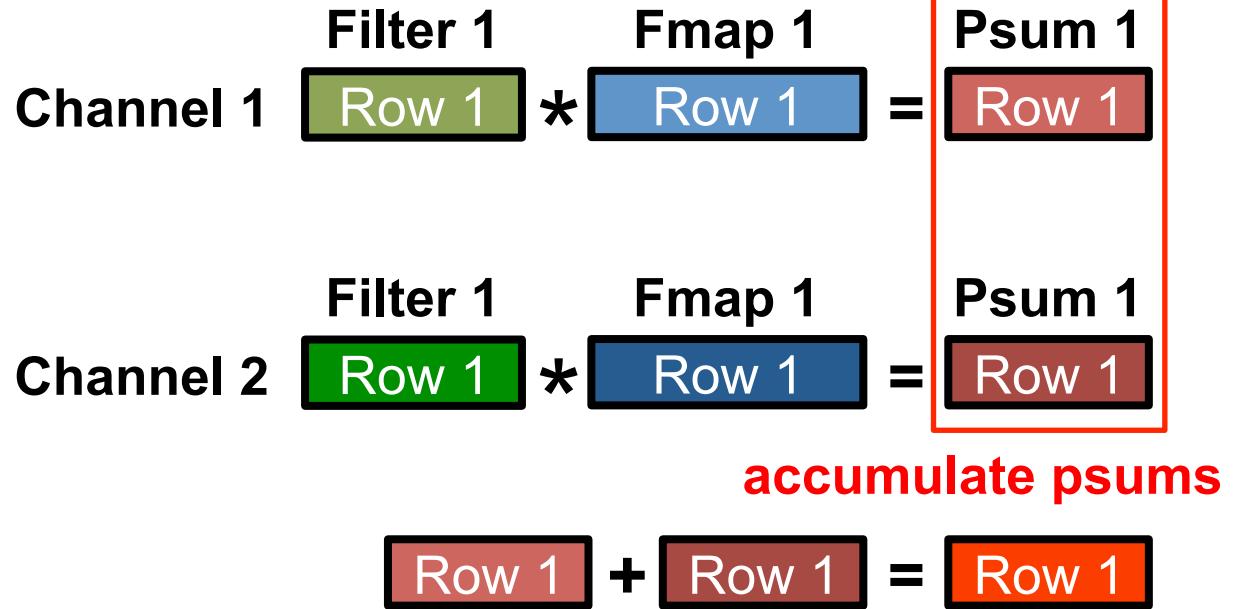
# Channel Accumulation in PE

1 Multiple Fmaps



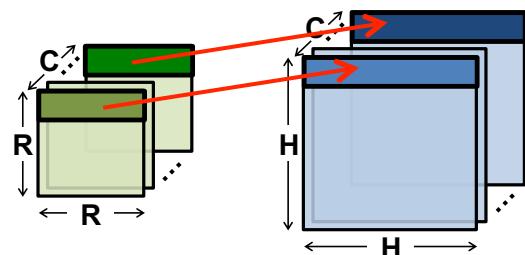
2 Multiple Filters

3 Multiple Channels



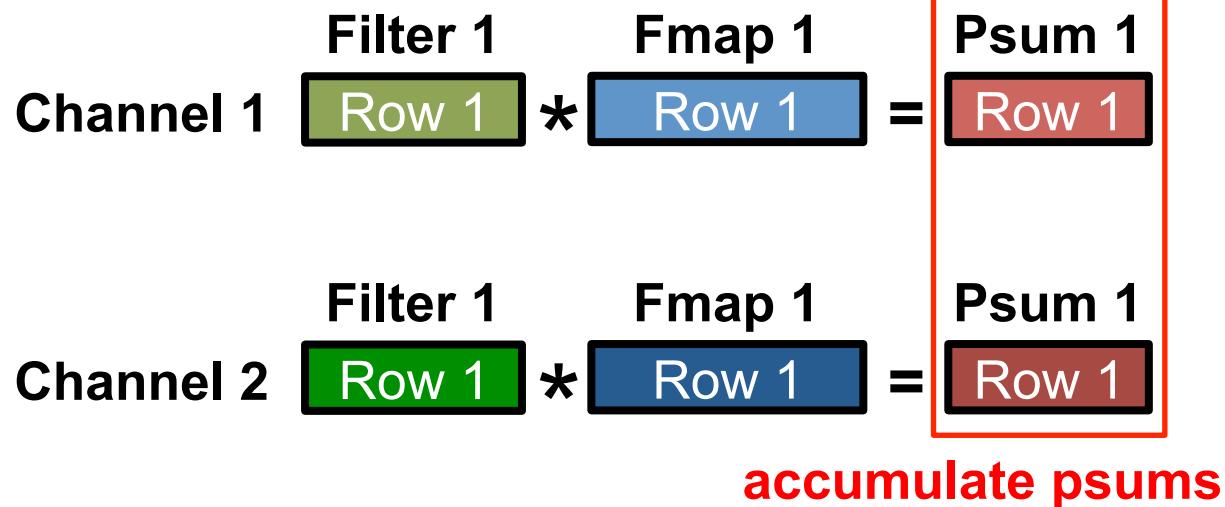
# Channel Accumulation in PE

1 Multiple Fmaps

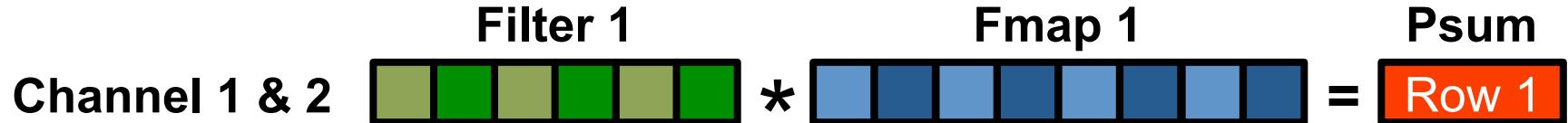


2 Multiple Filters

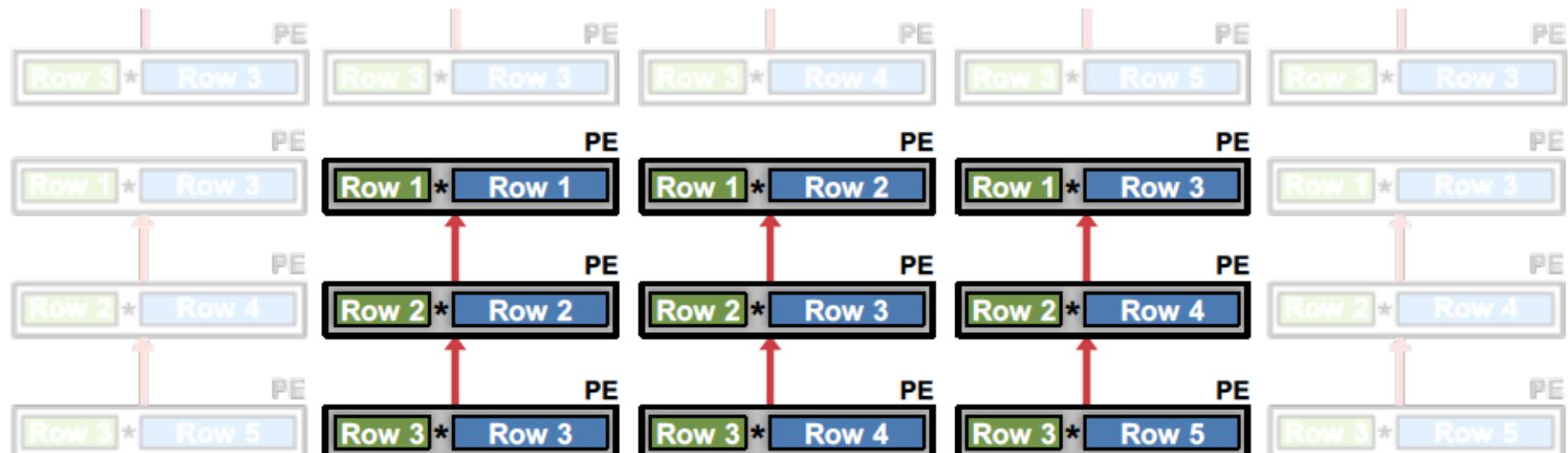
3 Multiple Channels



Processing in PE: interleave channels



# DNN Processing – The Full Picture



Multiple **fmaps**:



Multiple **filters**:



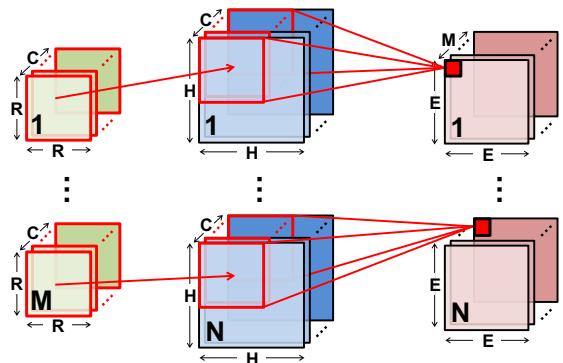
Multiple **channels**:



Map rows from **multiple fmaps**, **filters** and **channels** to same PE  
to exploit other forms of reuse and local accumulation

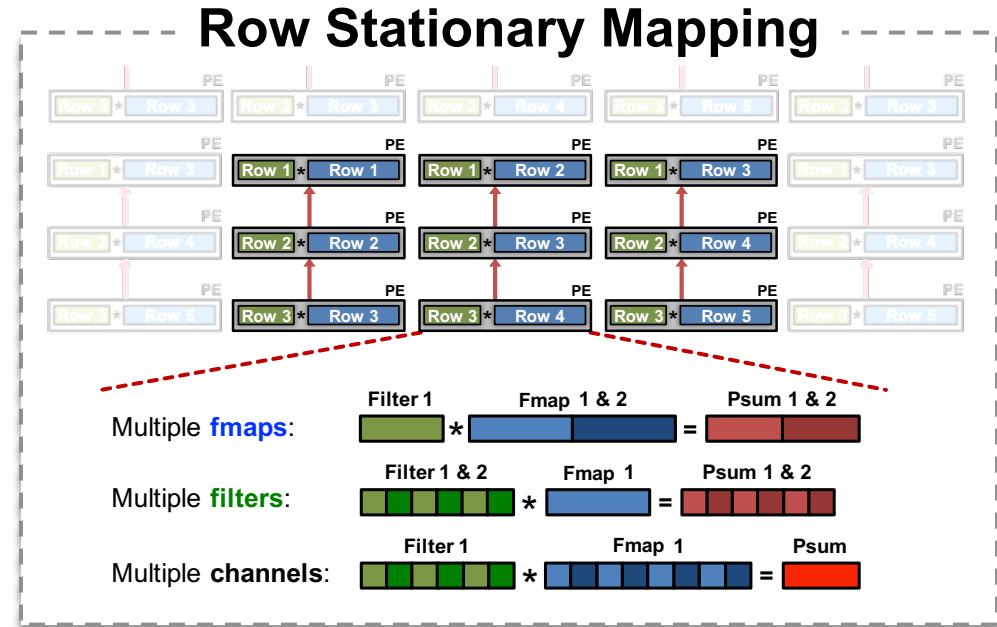
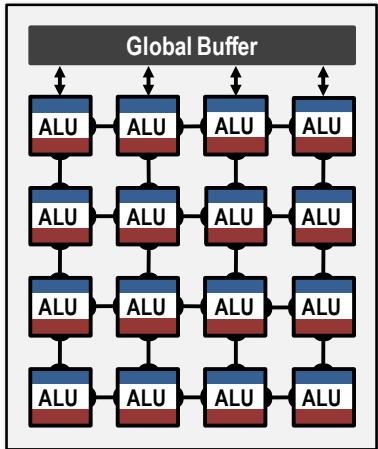
# Optimal Mapping in Row Stationary

## CNN Configurations

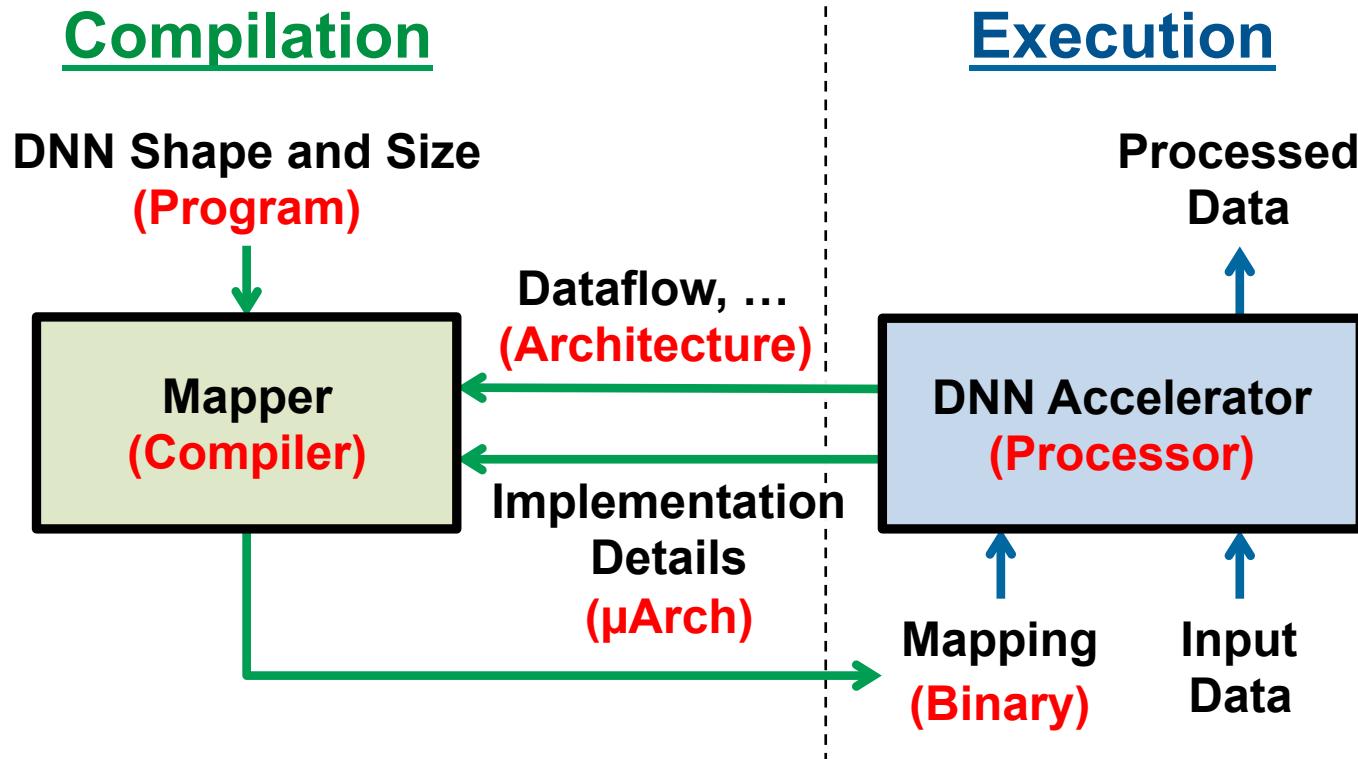


Optimization  
Compiler  
(Mapper)

## Hardware Resources



# Computer Architecture Analogy



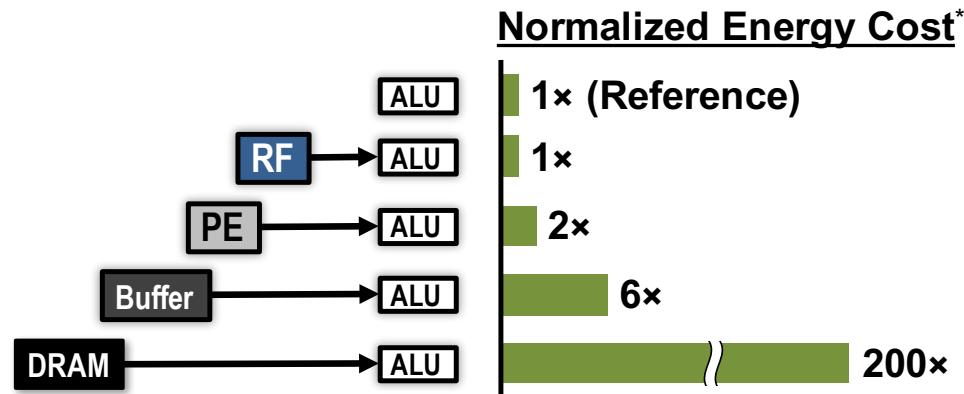
# **Dataflow Simulation Results**

# Evaluate Reuse in Different Dataflows

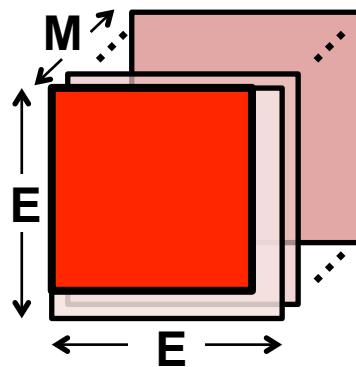
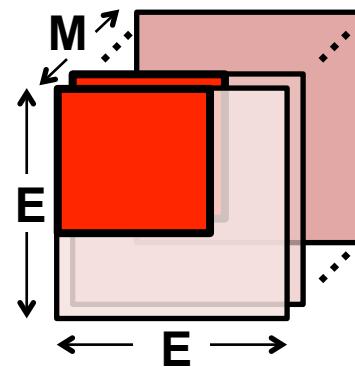
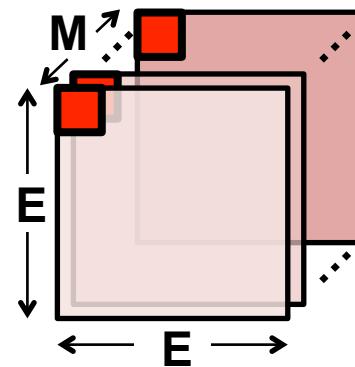
- **Weight Stationary**
  - Minimize movement of filter weights
- **Output Stationary**
  - Minimize movement of partial sums
- **No Local Reuse**
  - No PE local storage. Maximize global buffer size.
- **Row Stationary**

## Evaluation Setup

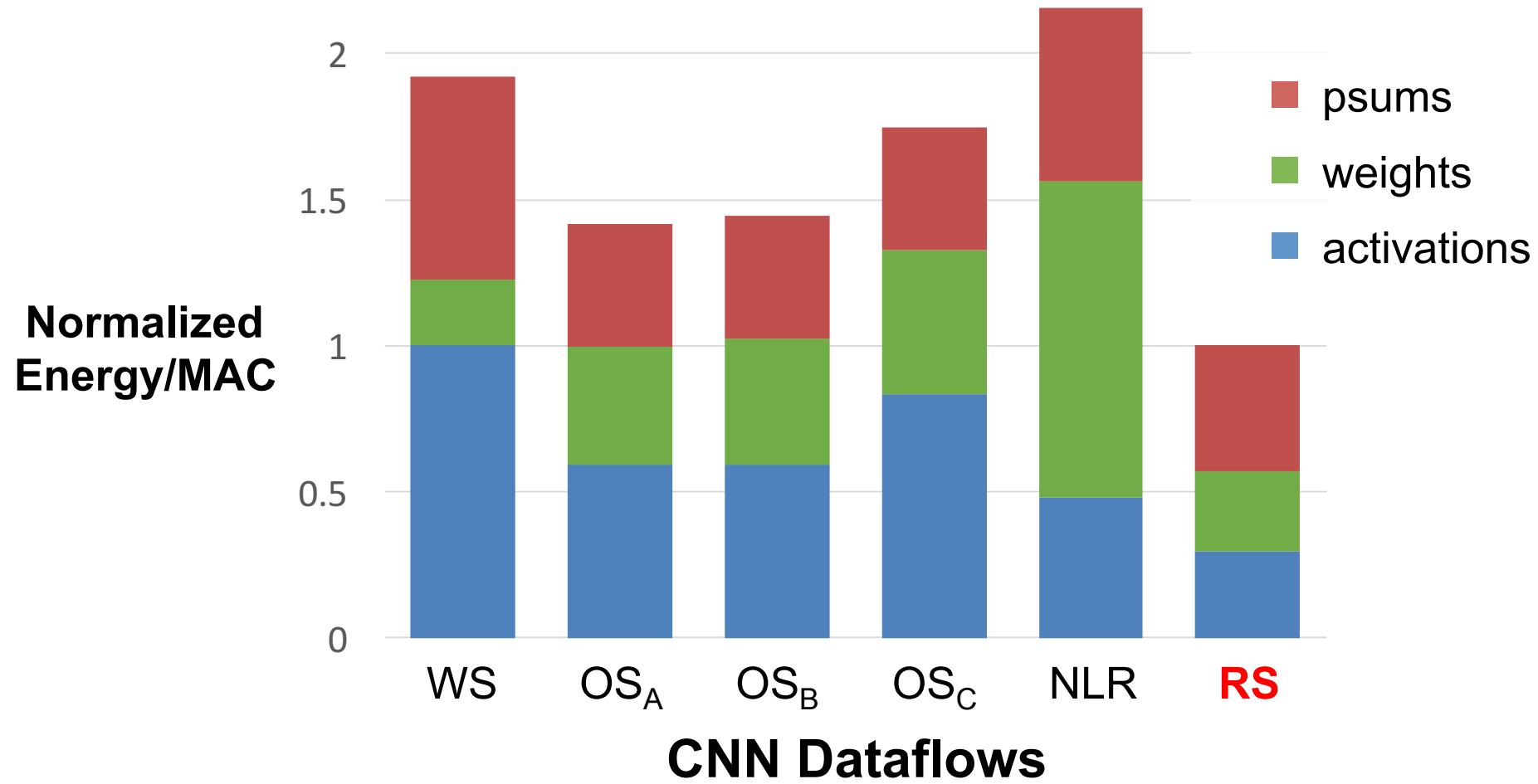
- same total area
- 256 PEs
- AlexNet
- batch size = 16



# Variants of Output Stationary

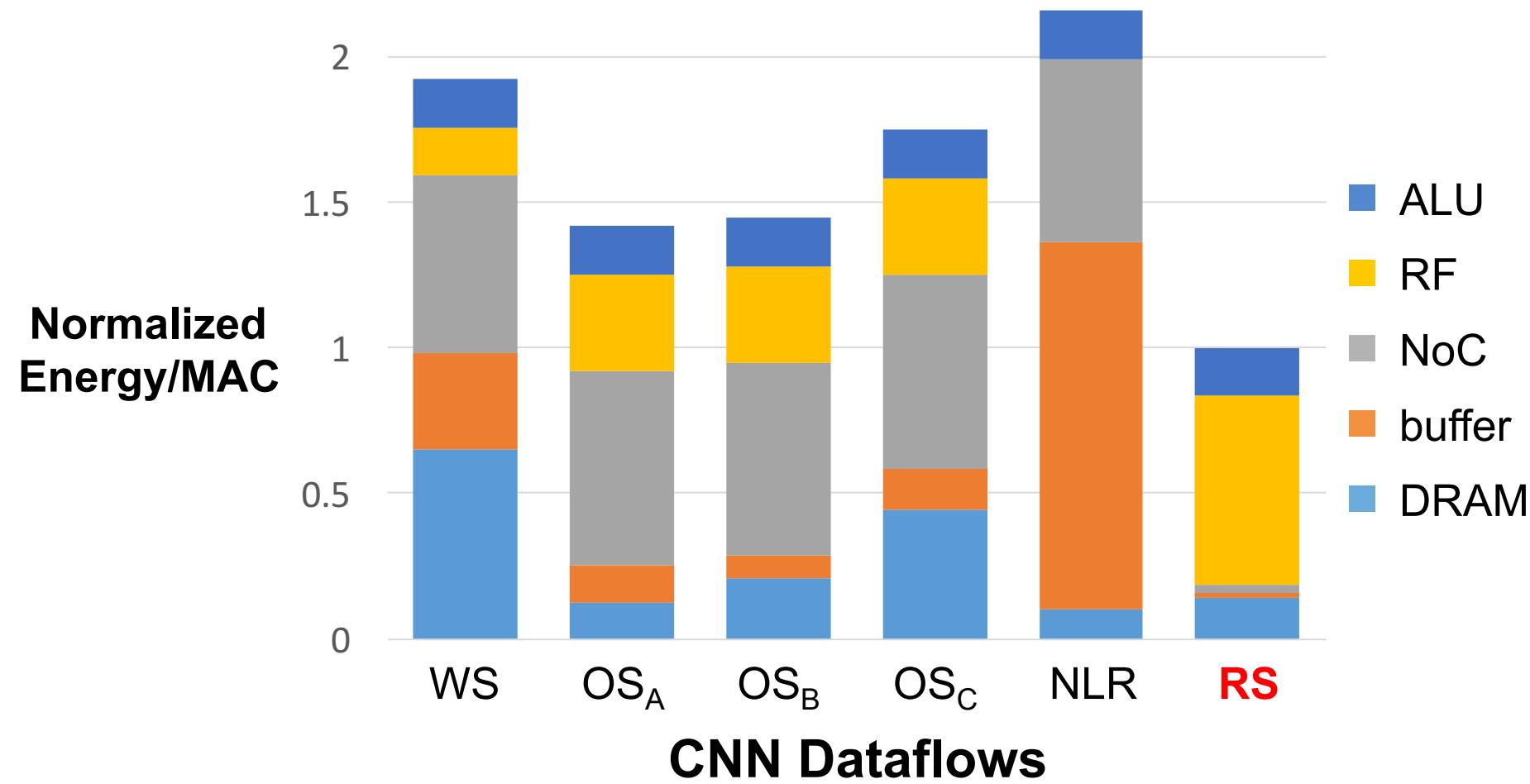
	$OS_A$	$OS_B$	$OS_C$
Parallel Output Region			
# Output Channels	Single	Multiple	Multiple
# Output Activations	Multiple	Multiple	Single
Notes	Targeting <b>CONV</b> layers		Targeting <b>FC</b> layers

# Dataflow Comparison: CONV Layers



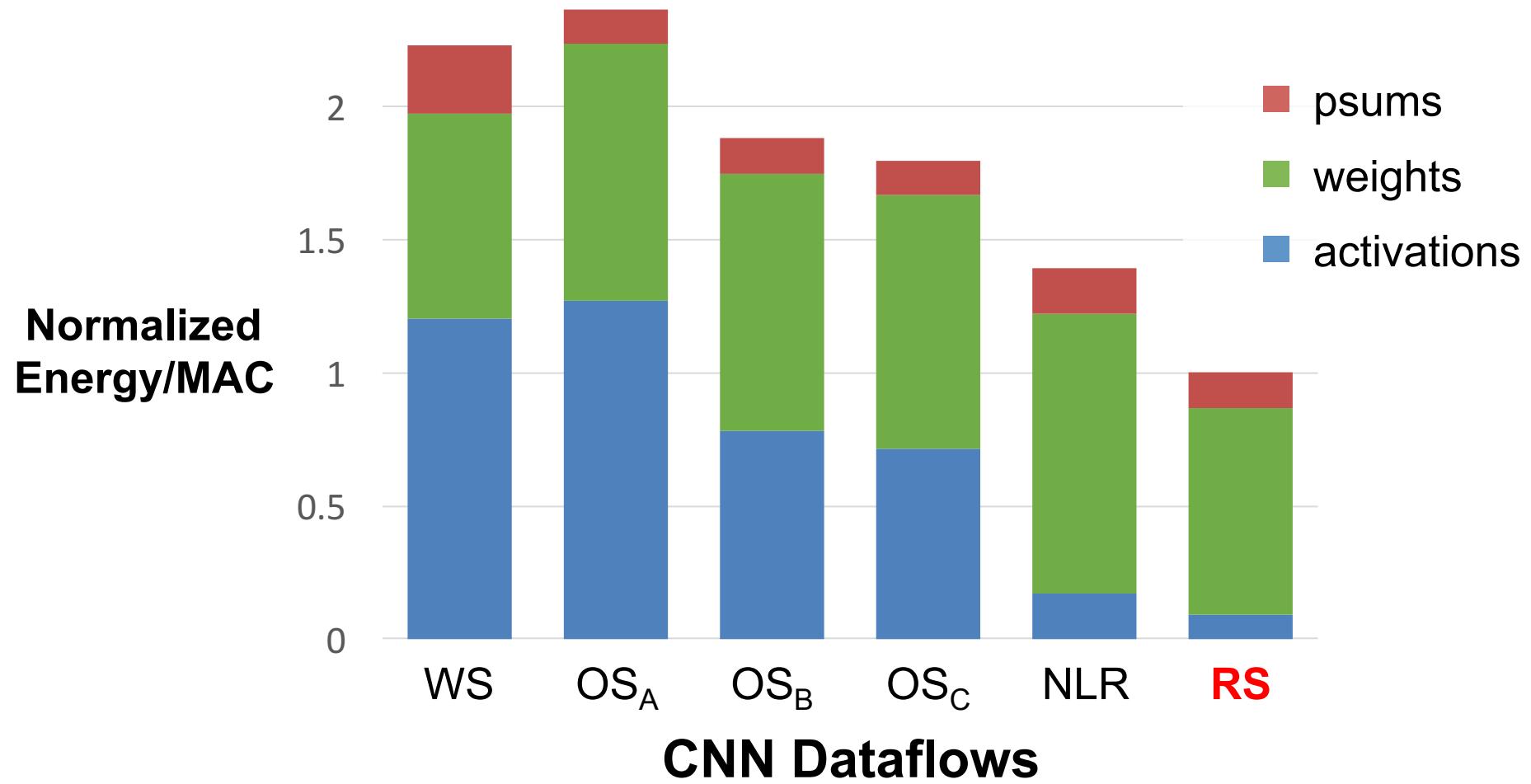
RS optimizes for the best **overall** energy efficiency

# Dataflow Comparison: CONV Layers



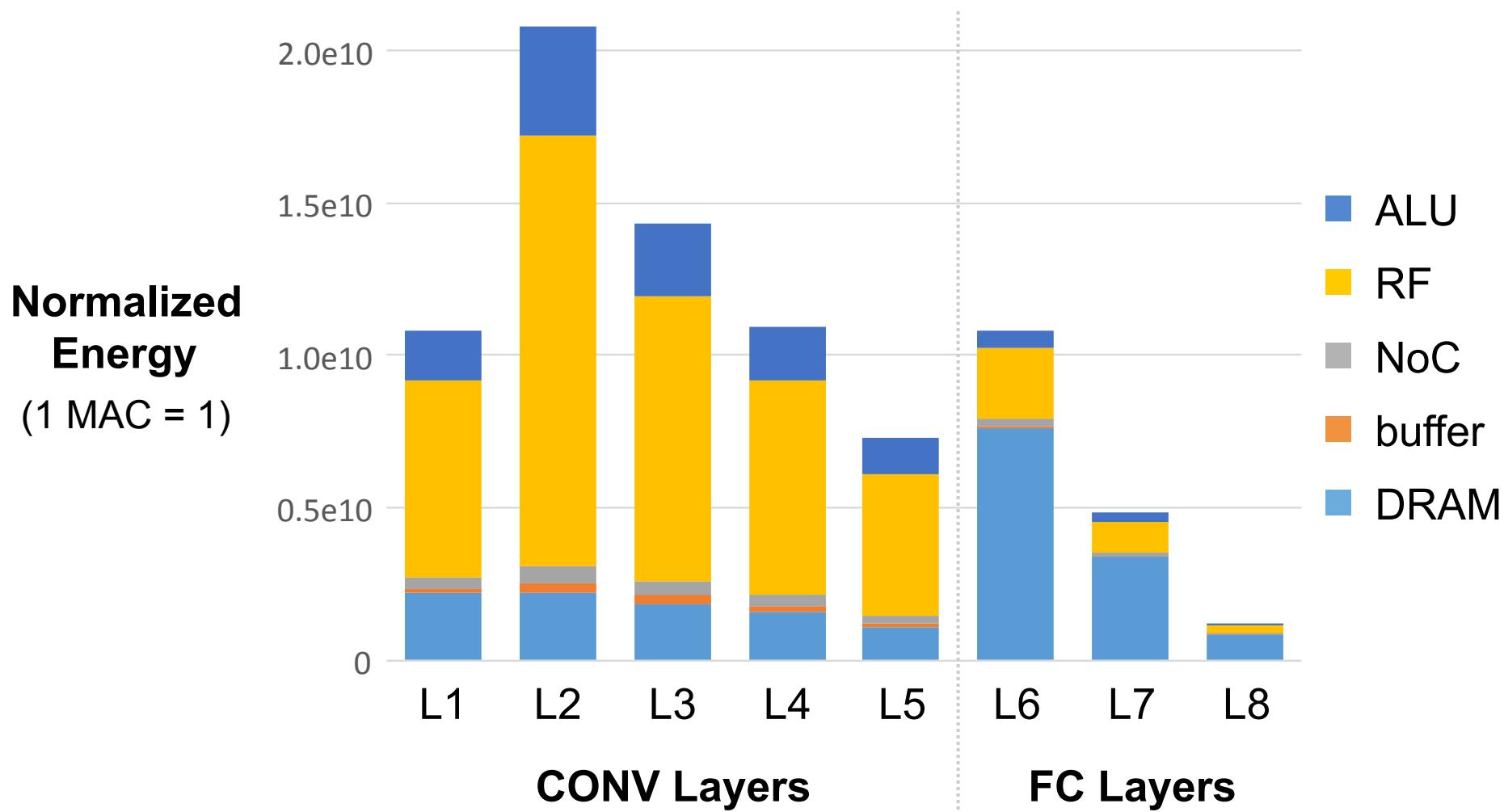
RS uses 1.4× – 2.5× lower energy than other dataflows

# Dataflow Comparison: FC Layers



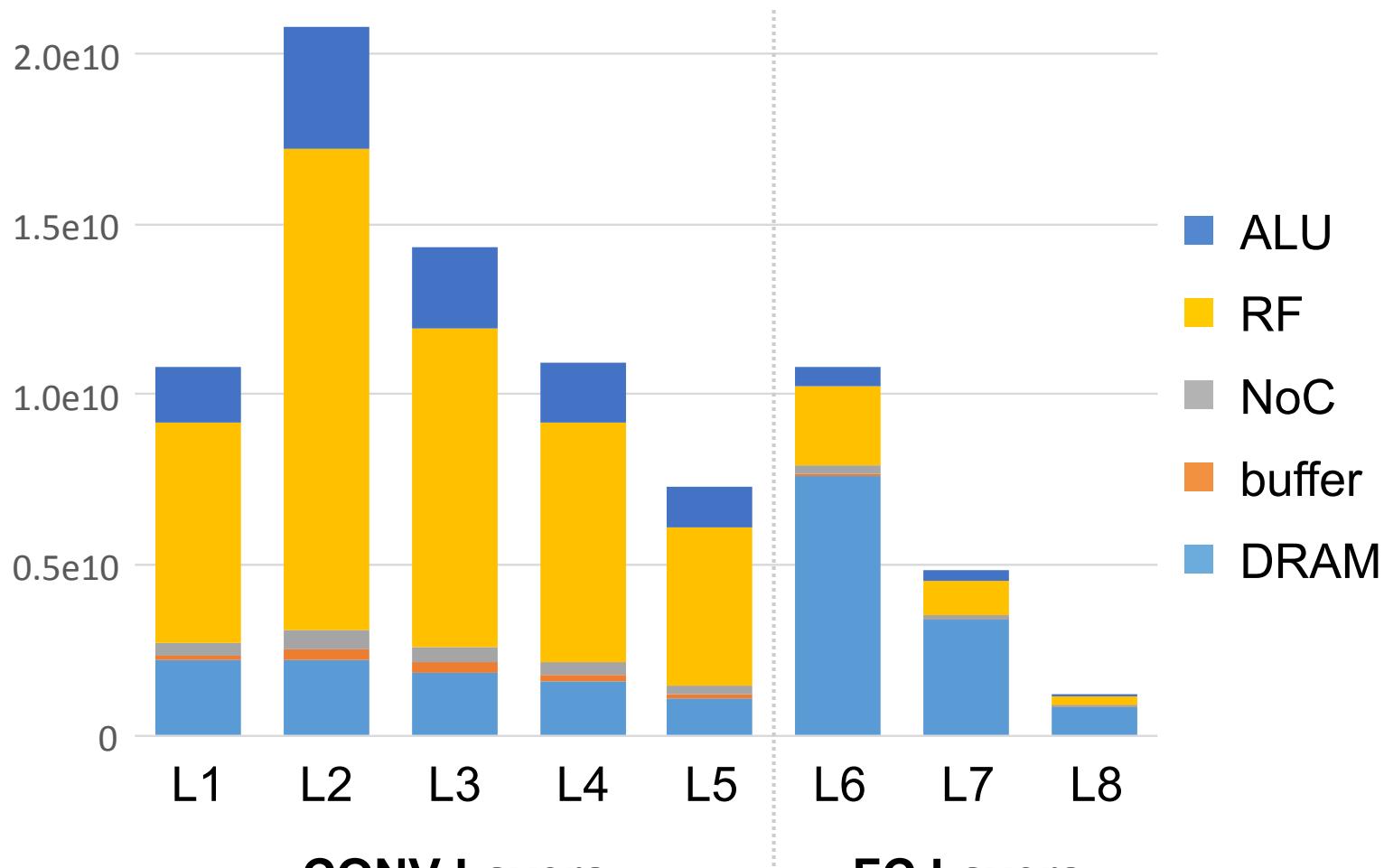
RS uses at least 1.3× lower energy than other dataflows

# Row Stationary: Layer Breakdown



# Row Stationary: Layer Breakdown

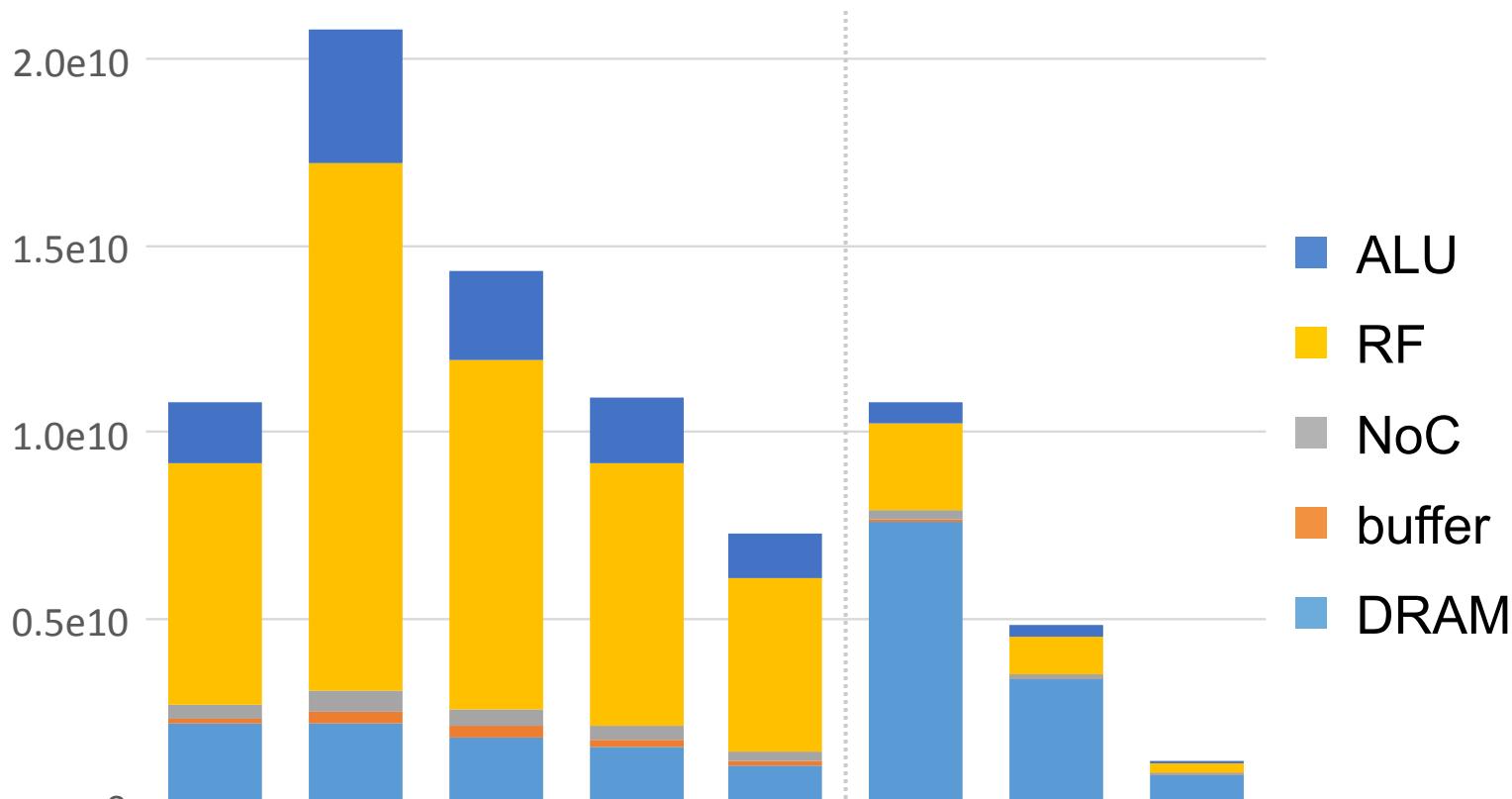
Normalized  
Energy  
(1 MAC = 1)



RF dominates

# Row Stationary: Layer Breakdown

Normalized  
Energy  
(1 MAC = 1)



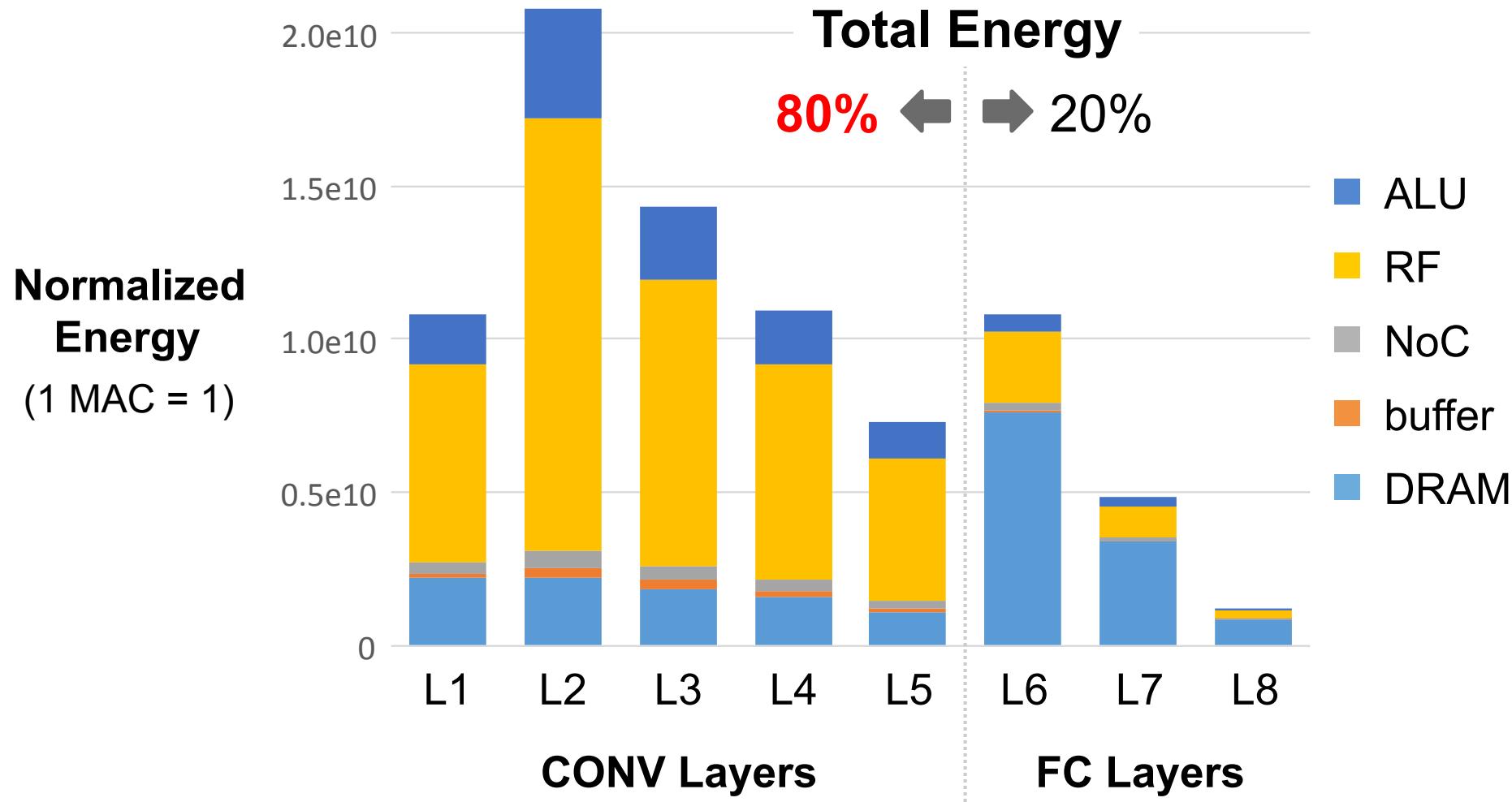
CONV Layers

RF dominates

FC Layers

DRAM dominates

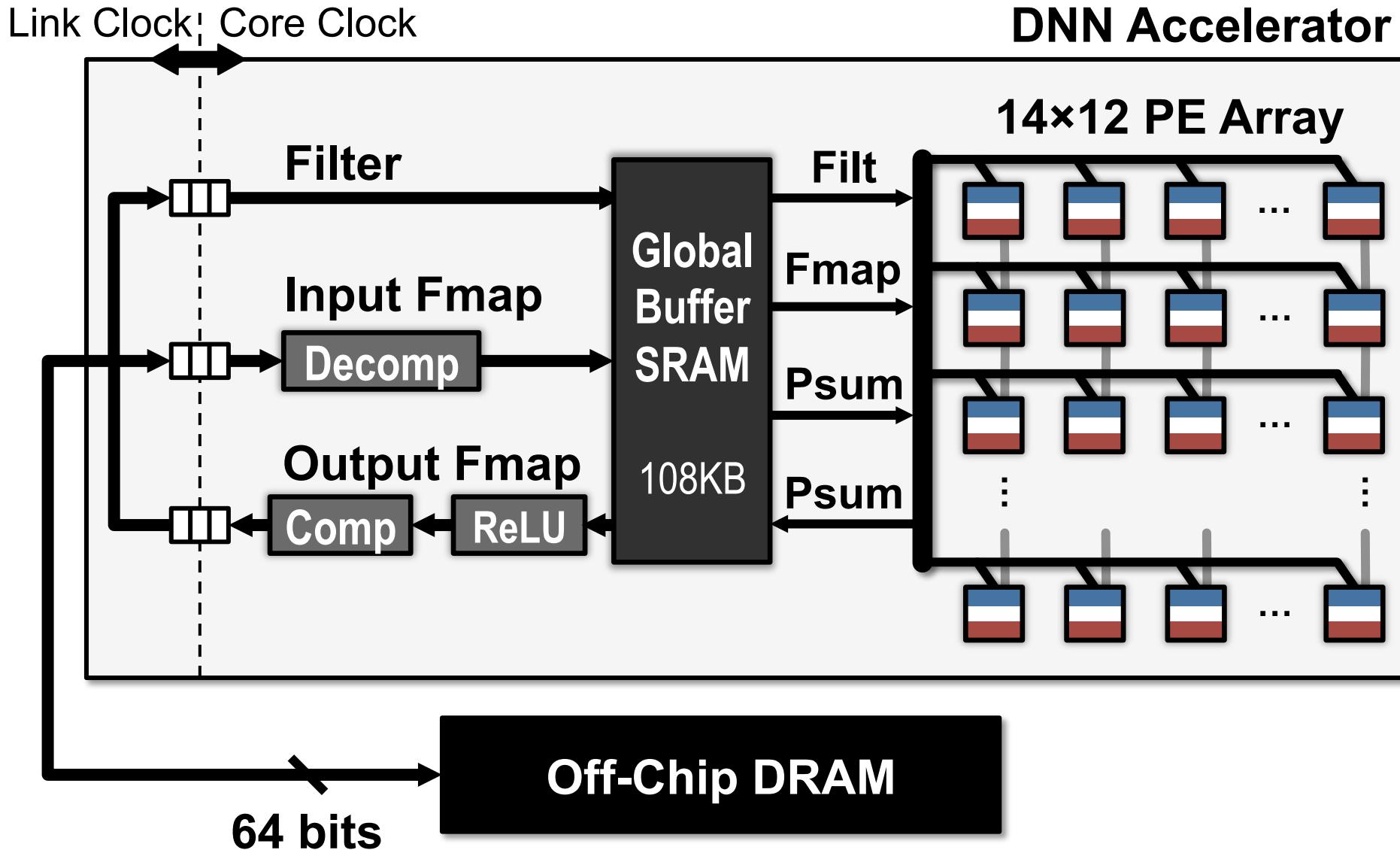
# Row Stationary: Layer Breakdown



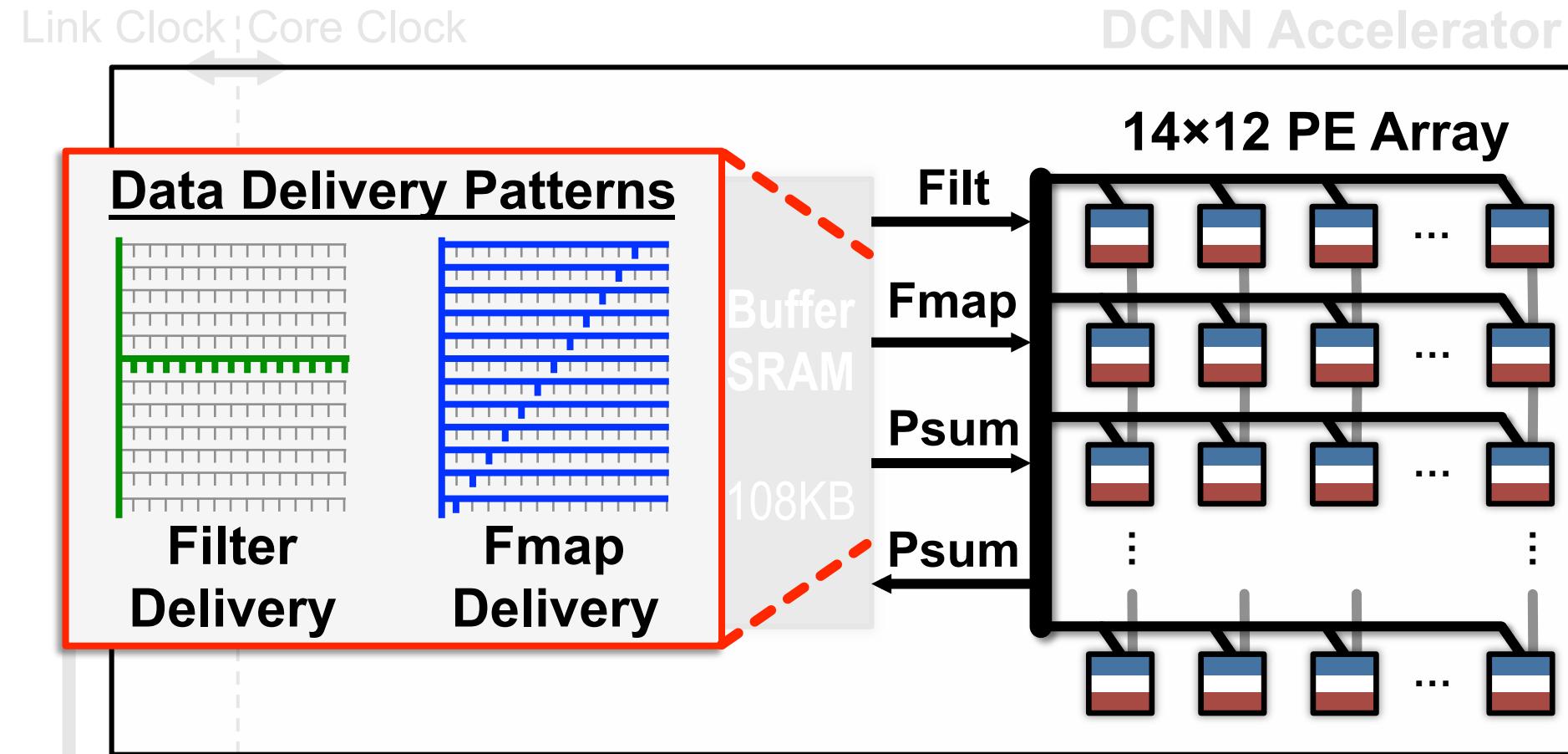
CONV layers dominate energy consumption!

# **Hardware Architecture for RS Dataflow**

# Eyeriss DNN Accelerator



# Data Delivery with On-Chip Network

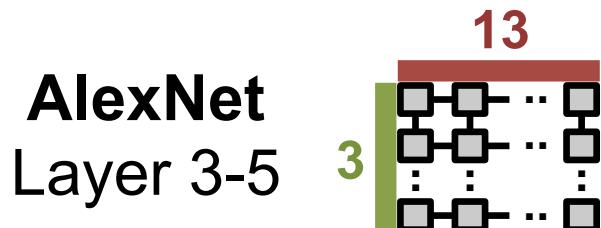


How to accommodate different shapes with fixed PE array?

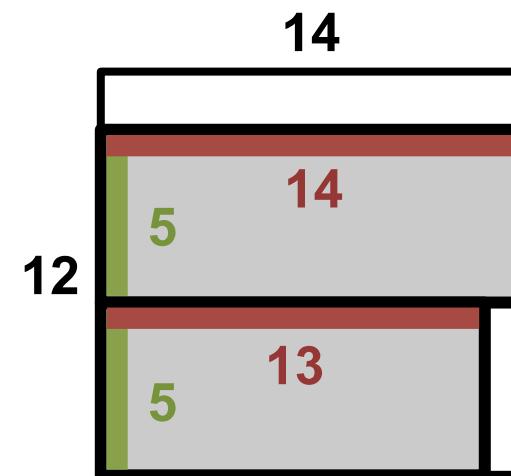
64 bits

# Logical to Physical Mappings

## Replication



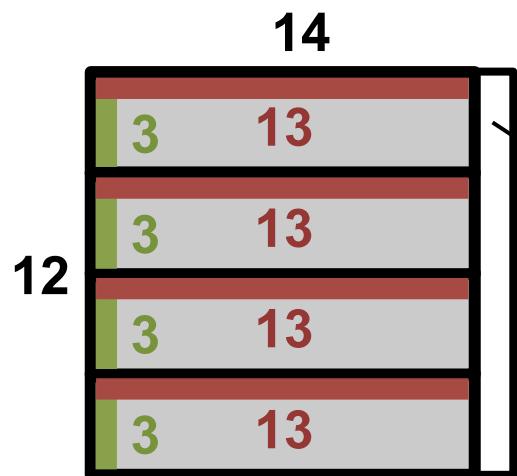
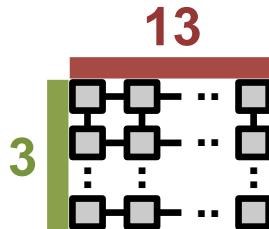
## Folding



# Logical to Physical Mappings

## Replication

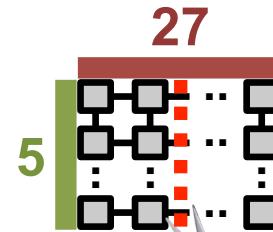
AlexNet  
Layer 3-5



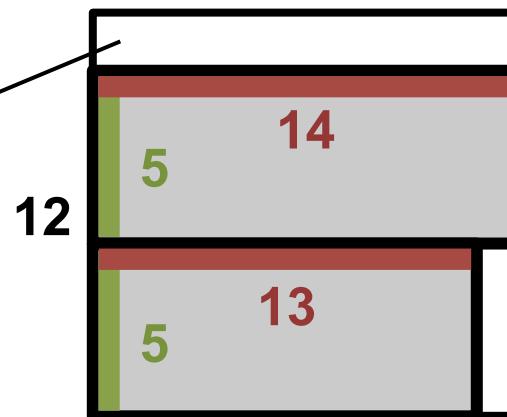
Physical PE Array

## Folding

AlexNet  
Layer 2



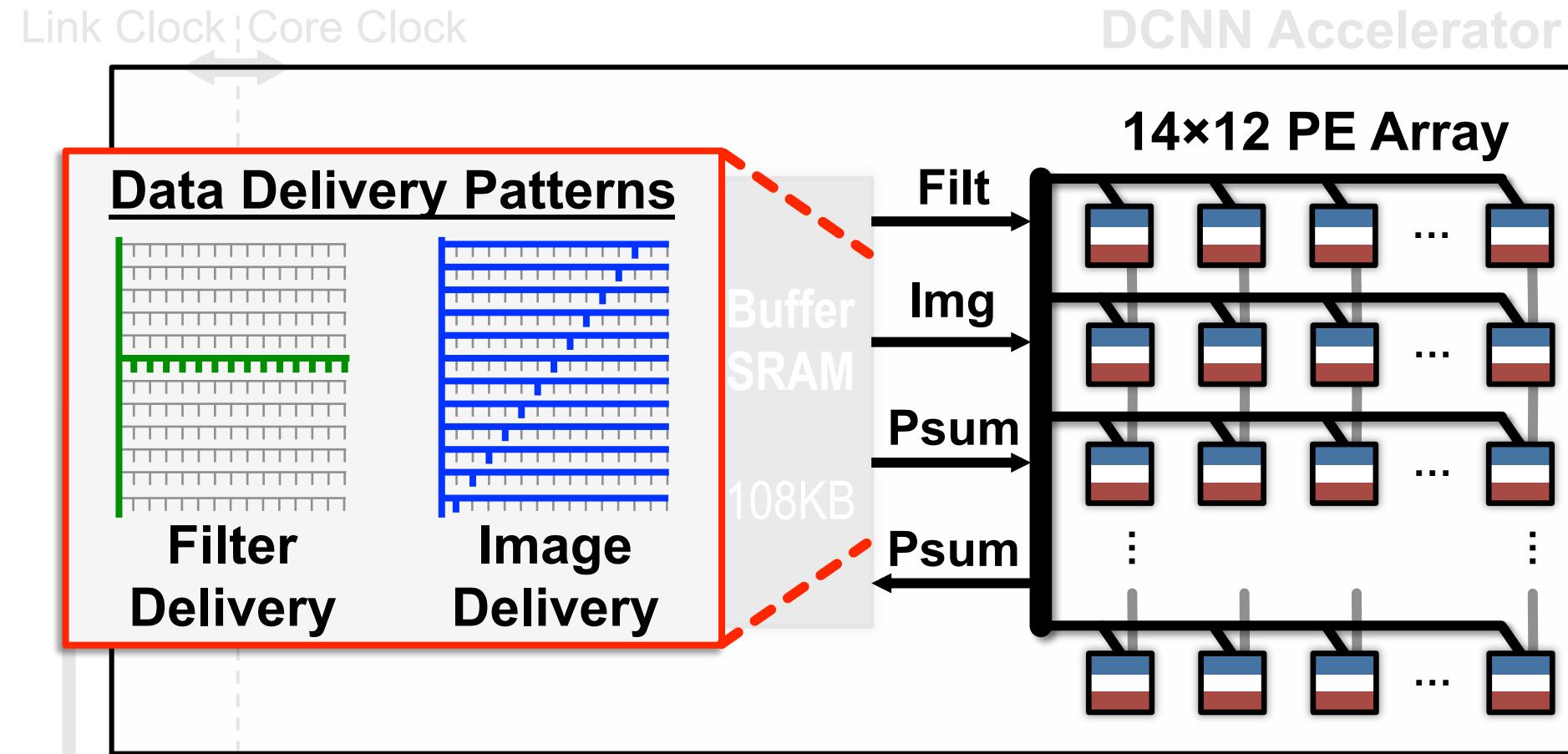
14



Physical PE Array

Unused PEs  
are  
**Clock Gated**

# Data Delivery with On-Chip Network



Compared to Broadcast, **Multicast** saves >80% of NoC energy

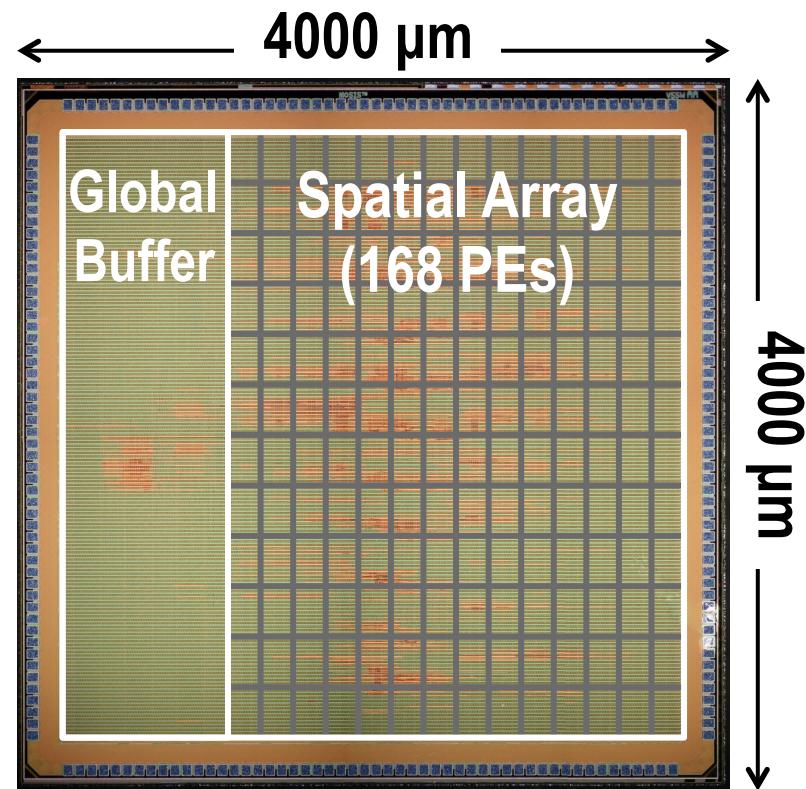
64 bits

MIT NVIDIA

70

# Chip Spec & Measurement Results

Technology	TSMC 65nm LP 1P9M
On-Chip Buffer	108 KB
# of PEs	168
Scratch Pad / PE	0.5 KB
Core Frequency	100 – 250 MHz
Peak Performance	33.6 – 84.0 GOPS
Word Bit-width	16-bit Fixed-Point
Natively Supported DNN Shapes	Filter Width: 1 – 32 Filter Height: 1 – 12 Num. Filters: 1 – 1024 Num. Channels: 1 – 1024 Horz. Stride: 1–12 Vert. Stride: 1, 2, 4



To support 2.66 GMACs [8 billion 16-bit inputs (**16GB**) and 2.7 billion outputs (**5.4GB**)], only requires **208.5MB** (buffer) and **15.4MB** (DRAM)

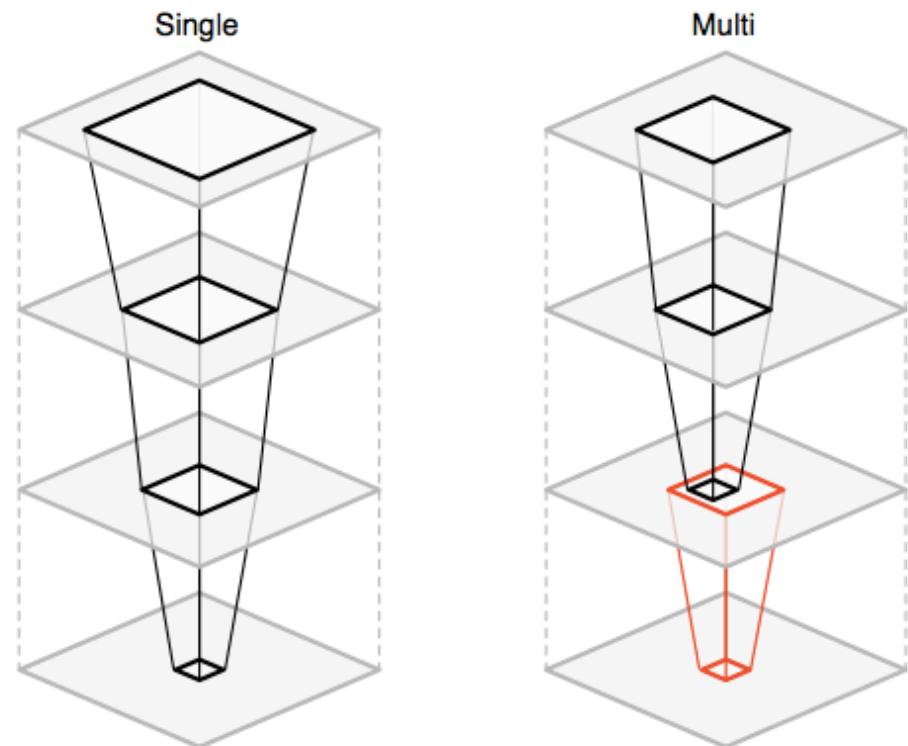
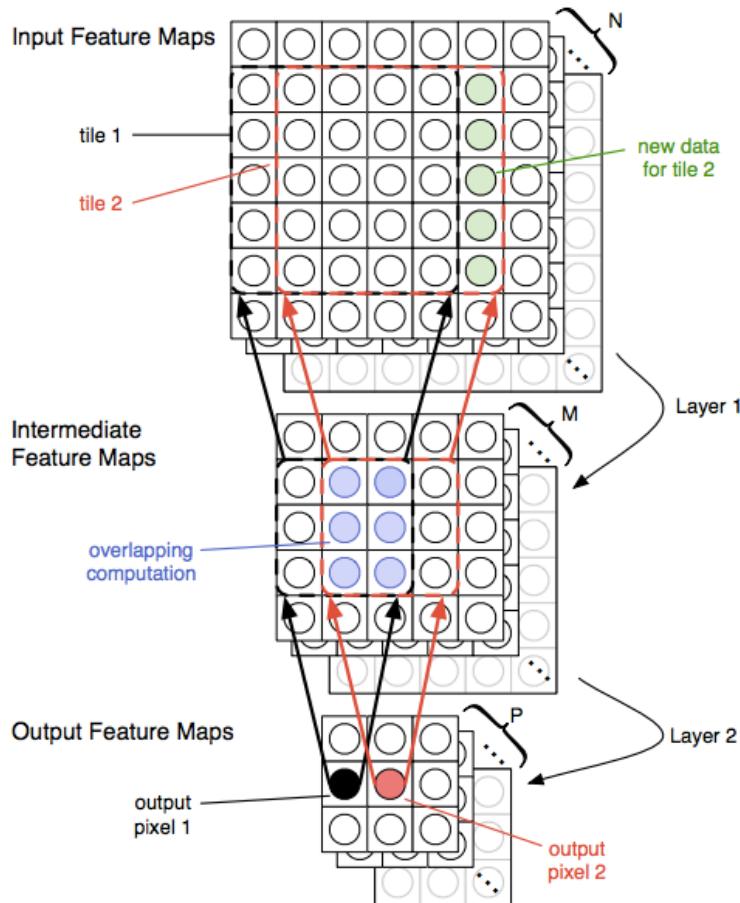
# Summary of DNN Dataflows

---

- **Weight Stationary**
  - Minimize movement of filter weights
  - Popular with processing-in-memory architectures
- **Output Stationary**
  - Minimize movement of partial sums
  - Different variants optimized for CONV or FC layers
- **No Local Reuse**
  - No PE local storage → maximize global buffer size
- **Row Stationary**
  - Adapt to the NN shape and hardware constraints
  - Optimized for overall **system energy efficiency**

# Fused Layer

- Dataflow across multiple layers



# **Advanced Technology Opportunities**

## **ISCA Tutorial (2017)**

Website: <http://eyeriss.mit.edu/tutorial.html>

Joel Emer, Vivienne Sze, Yu-Hsin Chen

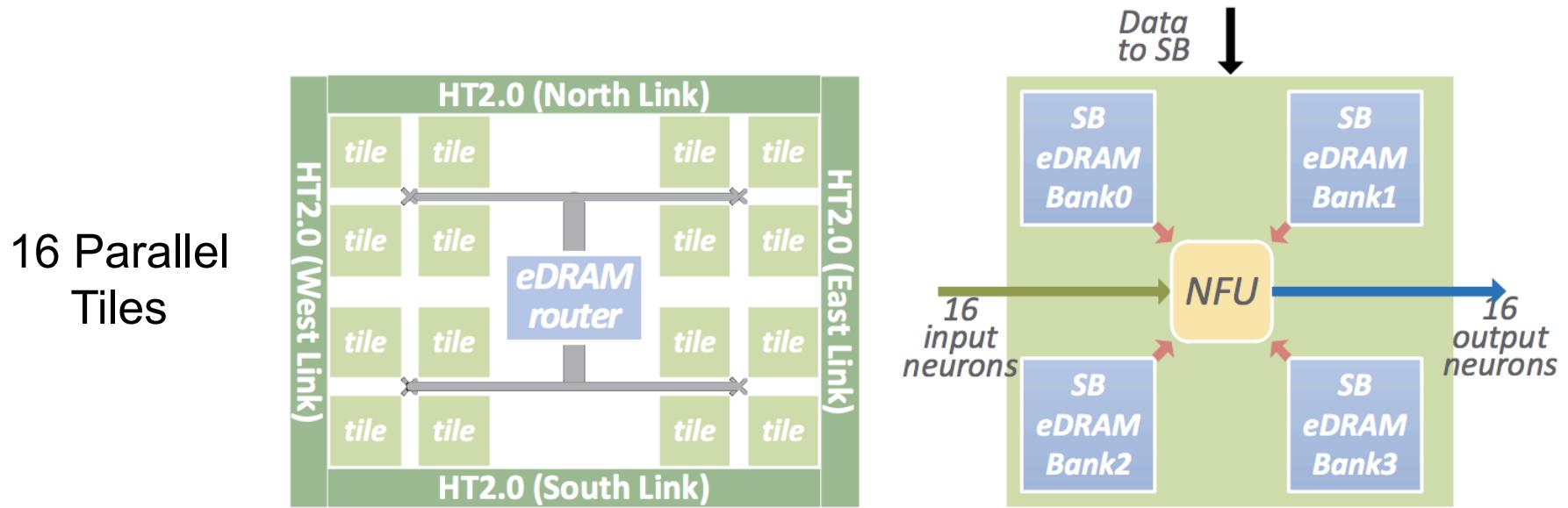
# Advanced Storage Technology

---

- **Embedded DRAM (eDRAM)**
  - Increase on-chip storage capacity
- **3D Stacked DRAM**
  - e.g. Hybrid Memory Cube Memory (HMC), High Bandwidth Memory (HBM)
  - Increase memory bandwidth

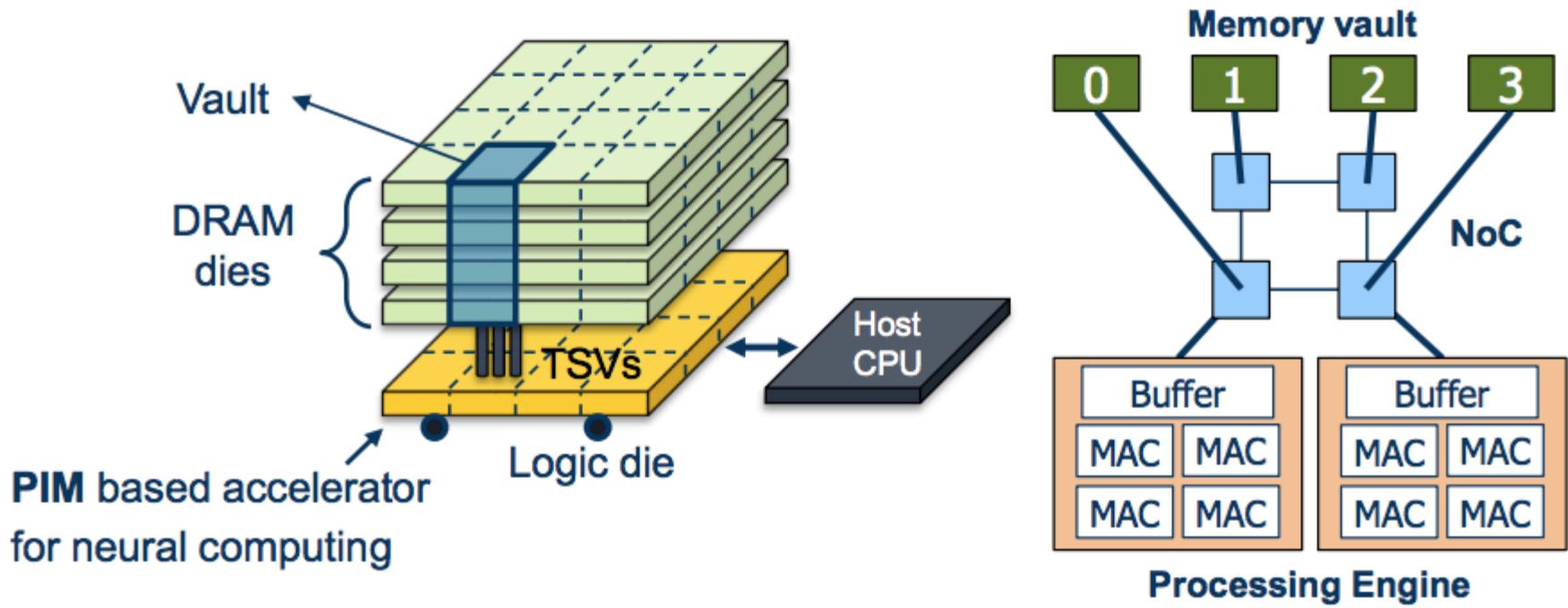
# eDRAM (DaDianNao)

- **Advantages of eDRAM**
  - 2.85x higher density than SRAM
  - 321x more energy-efficient than DRAM (DDR3)
- **Store weights in eDRAM (36MB)**
  - Target fully connected layers since dominated by weights



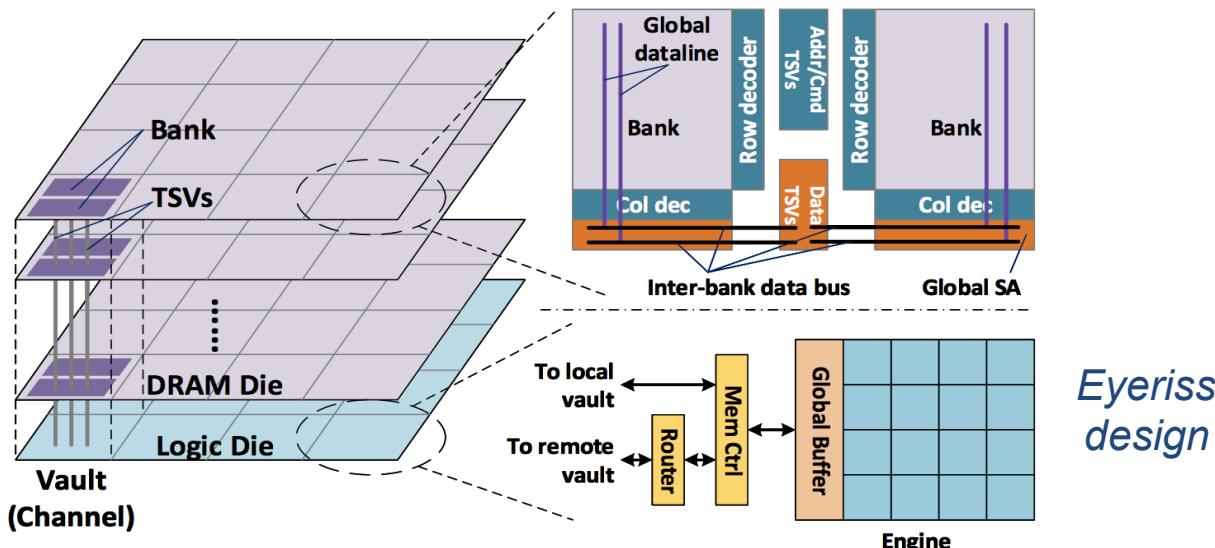
# Stacked DRAM (NeuroCube)

- NeuroCube on Hyper Memory Cube Logic Die
  - 6.25x higher BW than DDR3
    - HMC (16 ch x 10GB/s) > DDR3 BW (2 ch x 12.8GB/s)
  - Computation closer to memory (reduce energy)



# Stacked DRAM (TETRIS)

- Explores the use of HMC with the Eyeriss **spatial architecture** and **row stationary dataflow**
- Allocates **more area to the computation (PE array)** than **on-chip memory (global buffer)** to exploit the low energy and high throughput properties of the **HMC**
  - **1.5x energy reduction, 4.1x higher throughput** vs. 2-D DRAM



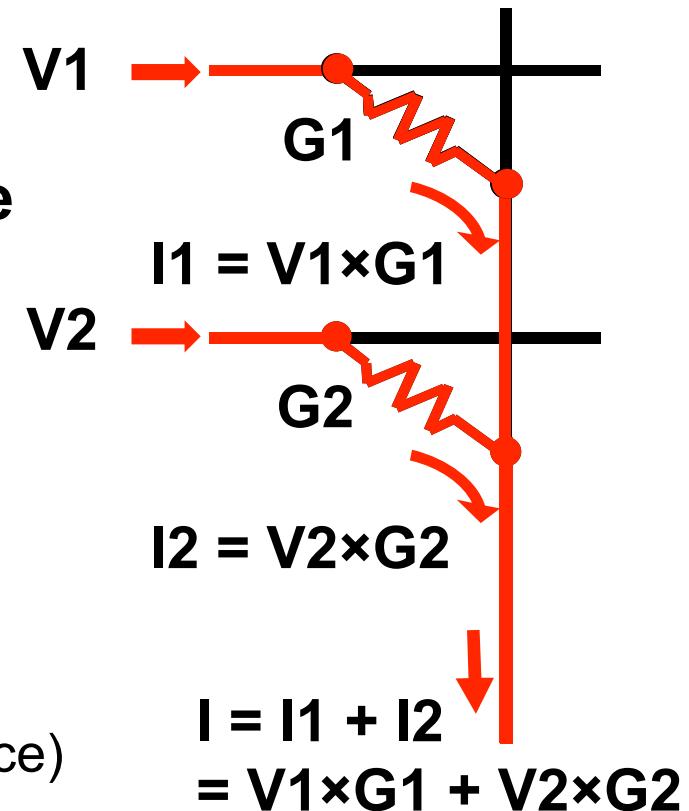
# Analog Computation

- Conductance = Weight
- Voltage = Input
- Current = Voltage  $\times$  Conductance
- Sum currents for addition

$$Output = \sum Weight \times Input$$

Input =  $V_1, V_2, \dots$

Filter Weights =  $G_1, G_2, \dots$  (conductance)



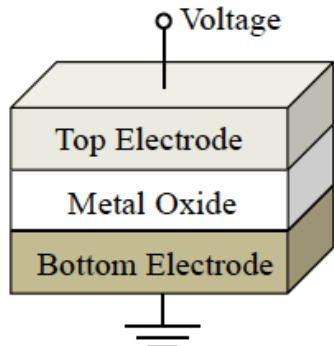
## Weight Stationary Dataflow

# Memristor Computation

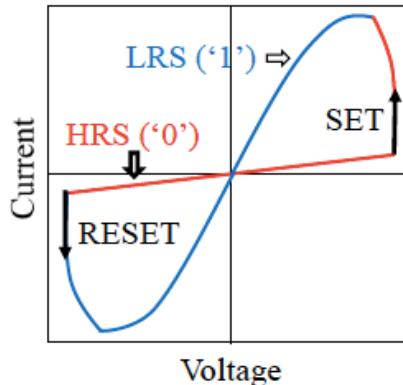
**Use memristors as programmable weights (resistance)**

- **Advantages**
  - **High Density (< 10nm x 10nm size\*)**
    - ~30x smaller than SRAM\*\*
    - 1.5x smaller than DRAM\*\*
  - **Non-Volatile**
  - **Operates at low voltage**
  - **Computation within memory (in situ)**
    - **Reduce data movement**

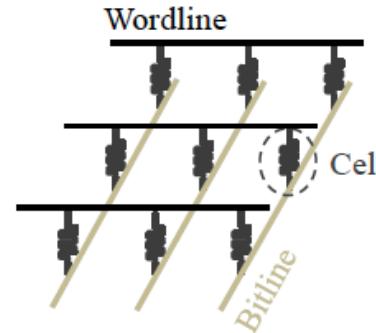
# Memristor



(a) Conceptual view  
of a ReRAM cell

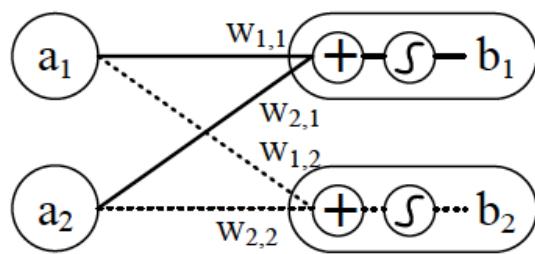


(b) I-V curve of bipolar  
switching

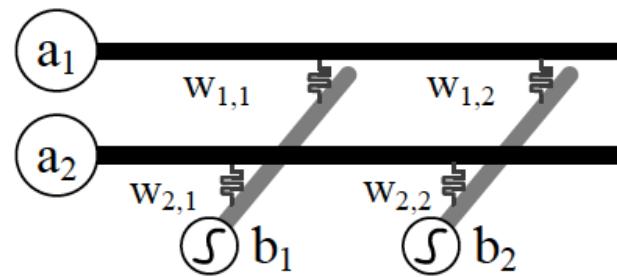


(c) schematic view of a  
crossbar architecture

$$b_j = \sigma \left( \sum_{\forall i} a_i \cdot w_{i,j} \right)$$



(a) An ANN with one input  
and one output layer



(b) using a ReRAM crossbar  
array for neural computation

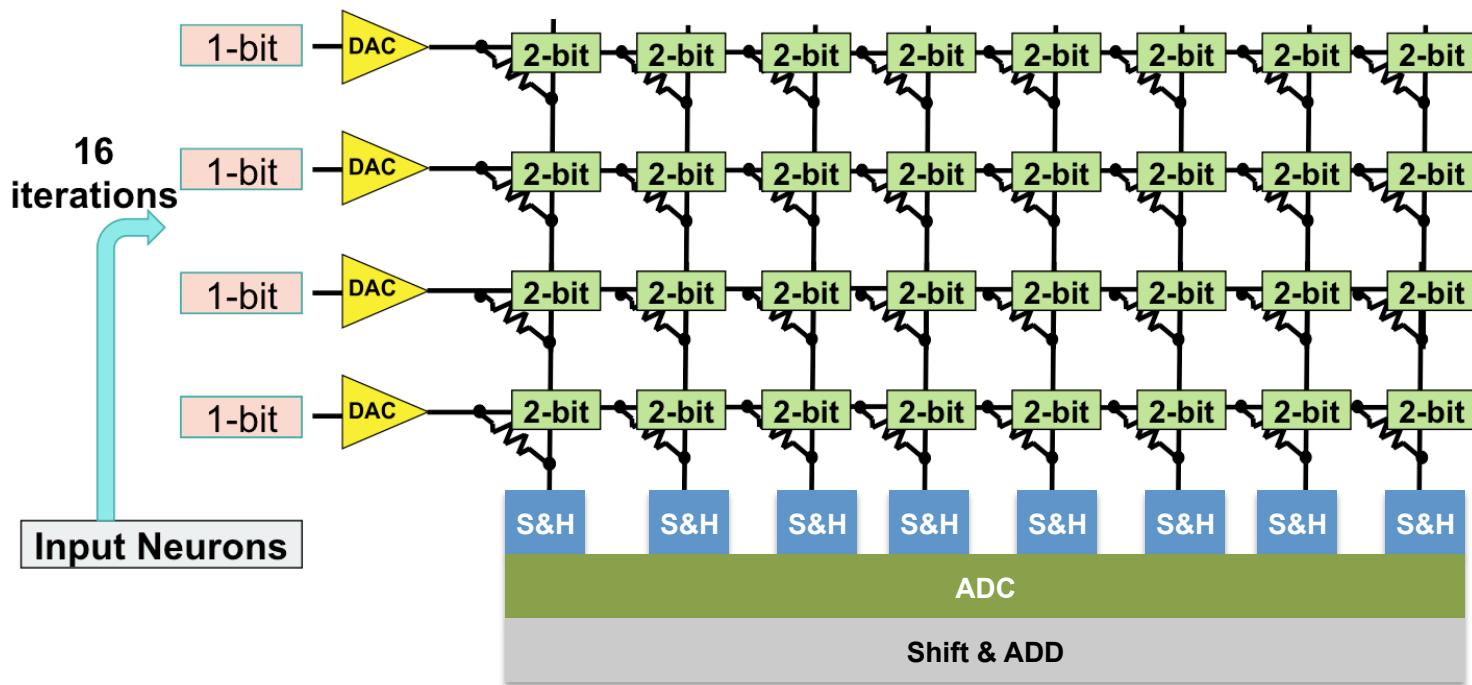
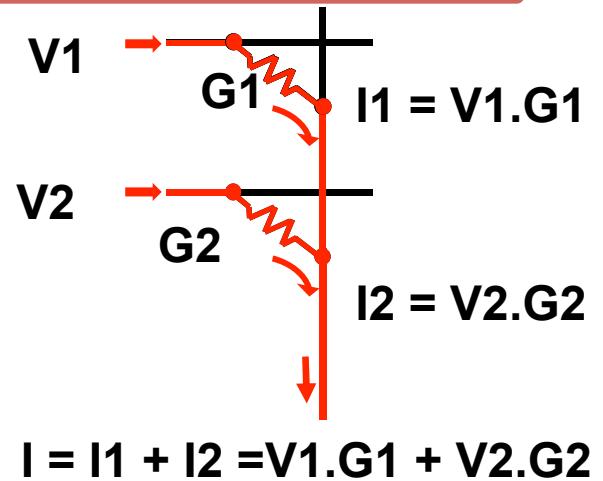
# Challenges with Memristors

---

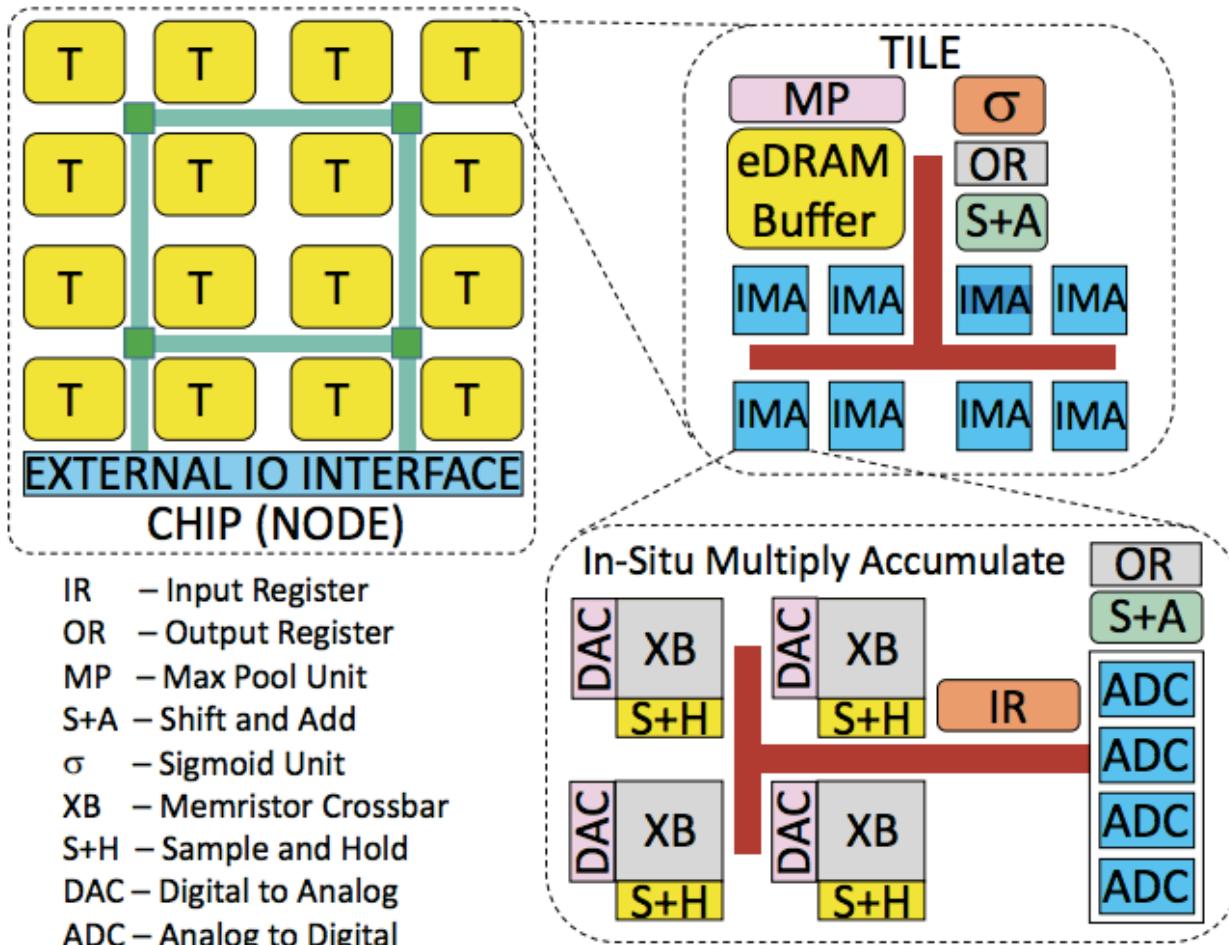
- **Limited Precision**
- **A/D and D/A Conversion**
- **Array Size and Routing**
  - Wire dominates energy for array size of  $1k \times 1k$
  - IR drop along wire can degrade read accuracy
- **Write/programming energy**
  - Multiple pulses can be costly
- **Variations & Yield**
  - **Device-to-device, cycle-to-cycle**
  - **Non-linear conductance across range**

# ISAAC

- eDRAM using memristors
- 16-bit dot-product operation
  - 8 x 2-bits per memristors
  - 1-bit per cycle computation



# ISAAC

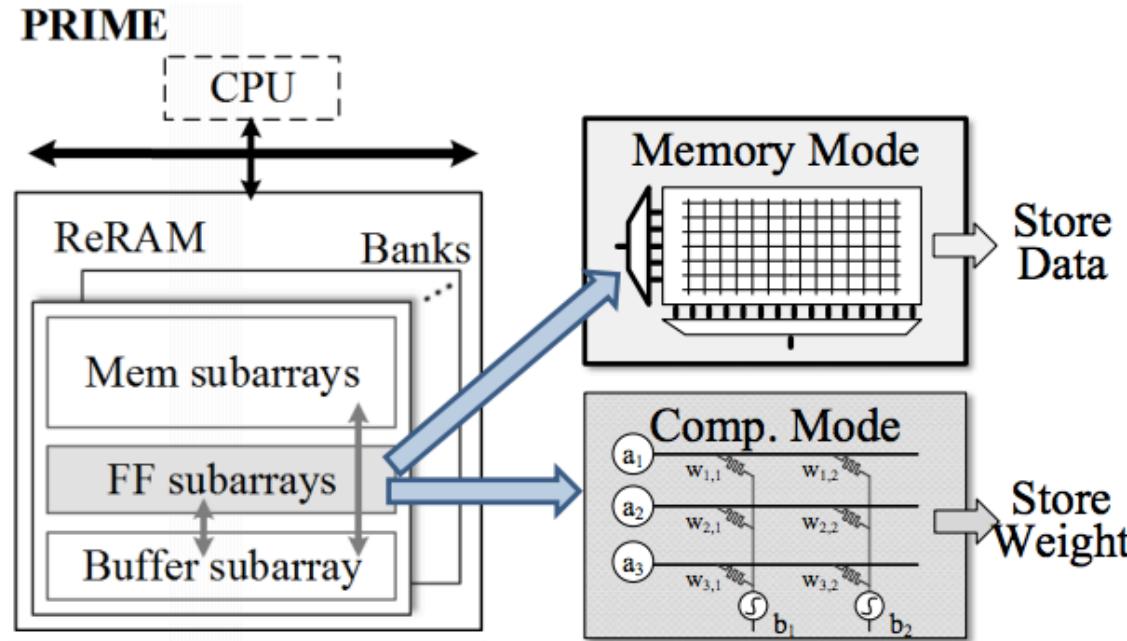


Eight 128x128 arrays per IMA

12 IMAs per Tile

# PRIME

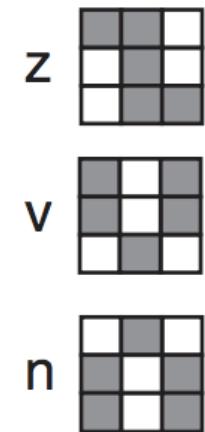
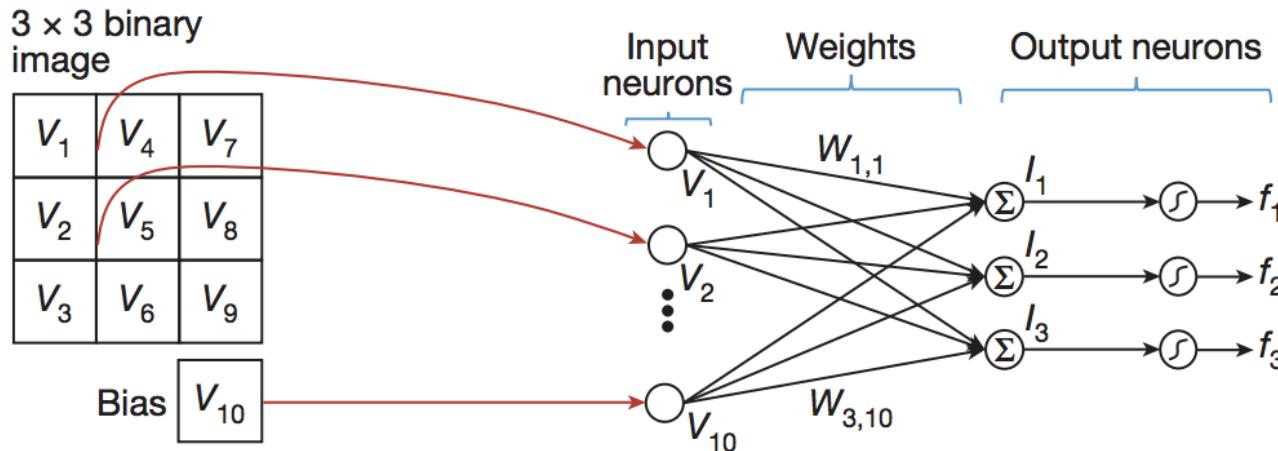
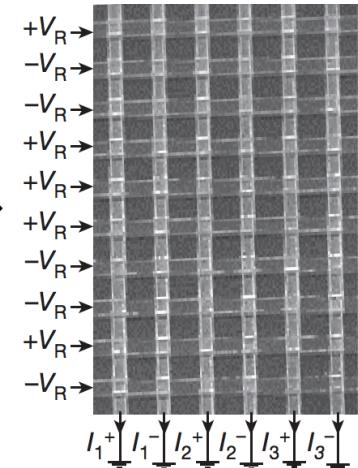
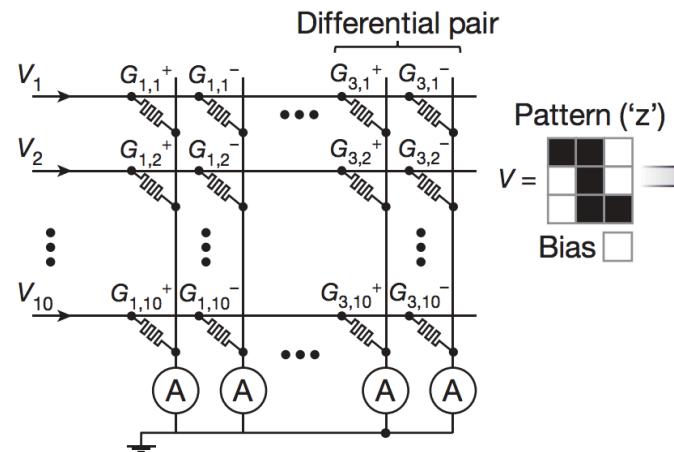
- Bit precision for each 256x256 ReRAM array
  - 3-bit input, 4-bit weight (2x for 6-bit input and 8-bit weight)
  - Dynamic fixed point (6-bit output)
- Reconfigurable to be main memory or accelerator
  - 4-bit MLC computation; 1-bit SLC for storage



# Fabricated Memristor Crossbar

- Transistor-free metal-oxide  
12x12 crossbar

- A single-layer perceptron (linear classification)
- 3x3 binary image
- 10 inputs x 3 outputs x 2 differential weights = 60 memristors

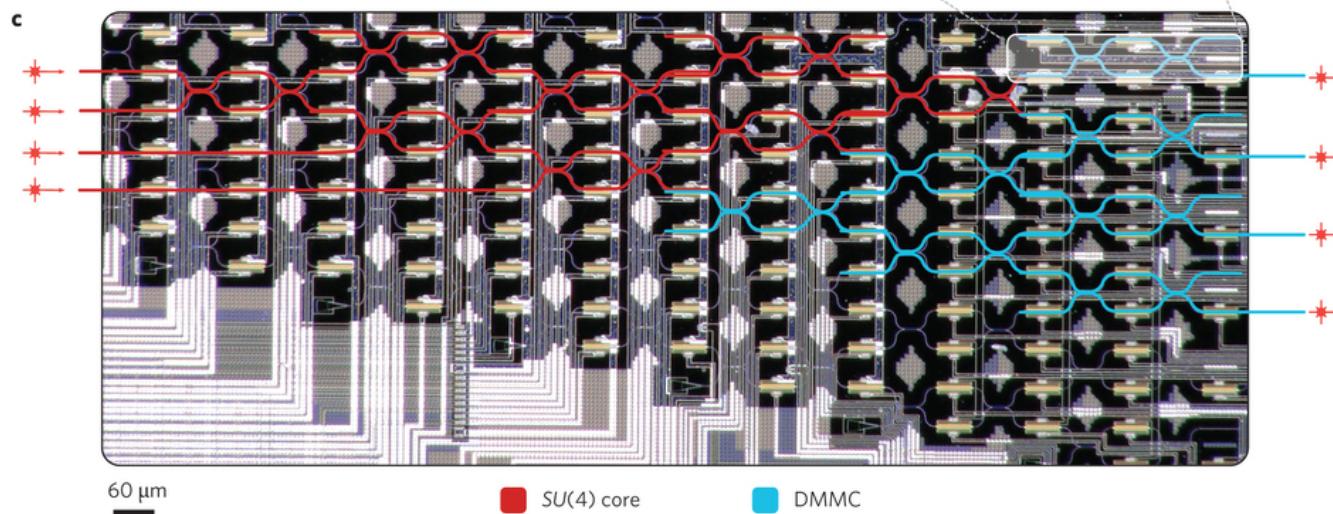
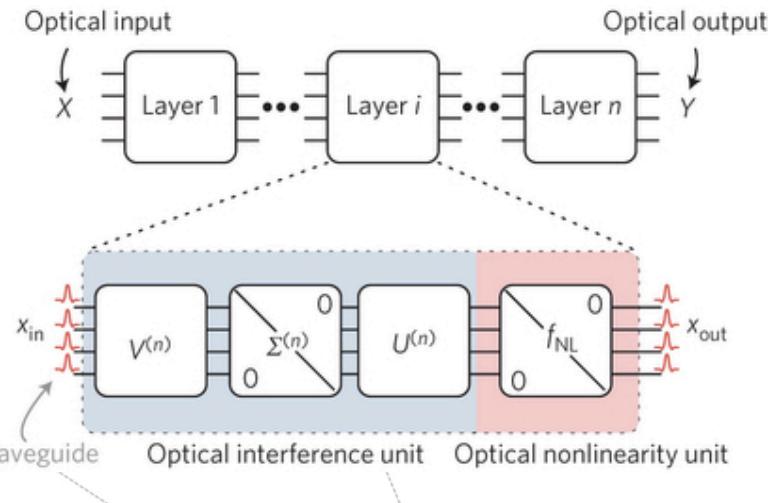


# Optical Neural Network

## Matrix Multiplication in the Optical Domain

The photodetection rate is 100 GHz

*“In principle, such a system can be at least two orders of magnitude faster than electronic neural networks (which are restricted to a GHz clock rate)”*



# **DNN Model and Hardware Co-Design**

**ISCA Tutorial (2017)**

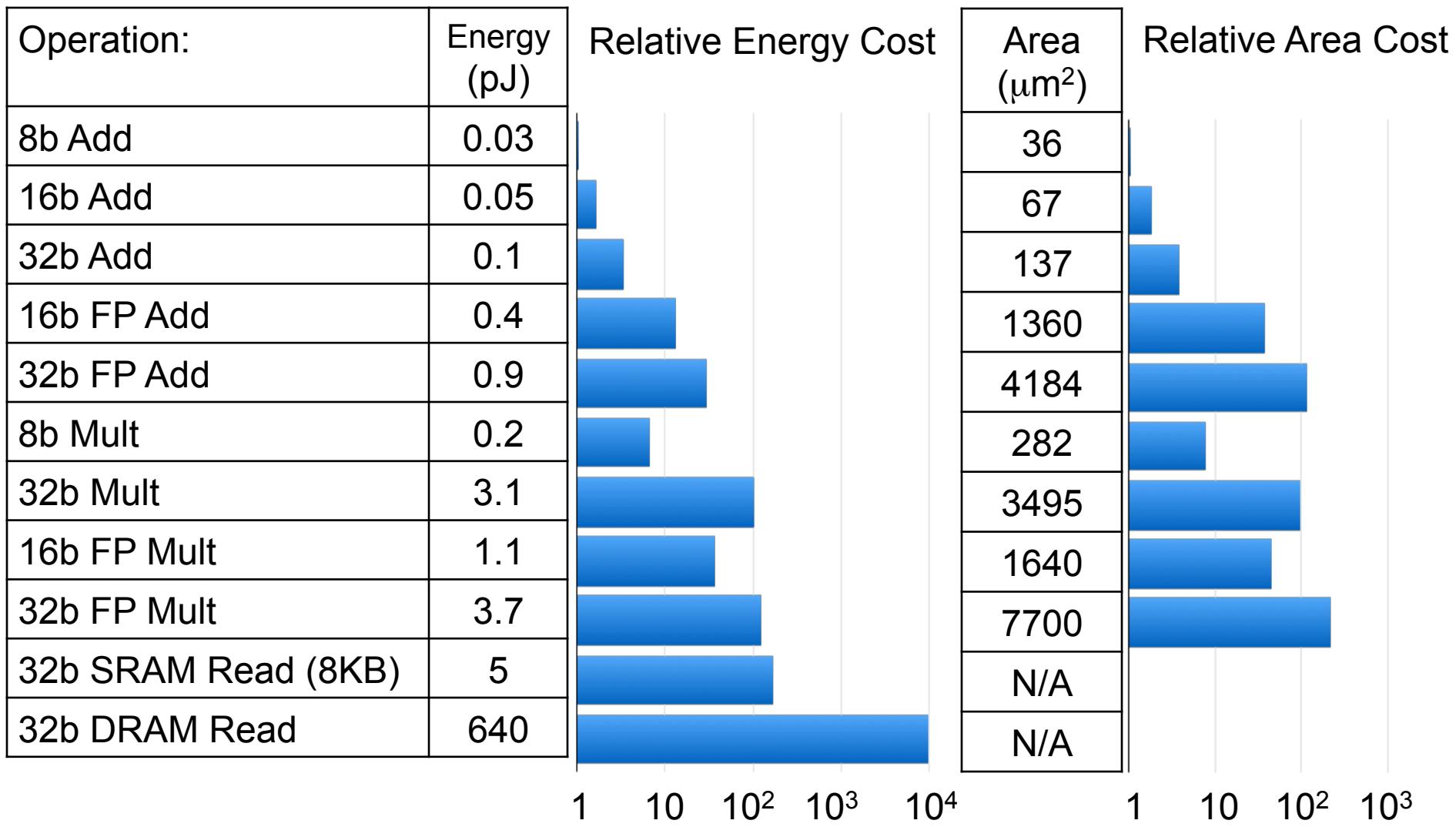
Website: <http://eyeriss.mit.edu/tutorial.html>

# Approaches

---

- Reduce size of operands for storage/compute
  - Floating point → Fixed point
  - Bit-width reduction
  - Non-linear quantization
- Reduce number of operations for storage/compute
  - Exploit Activation Statistics (Compression)
  - Network Pruning
  - Compact Network Architectures

# Cost of Operations



[Horowitz, "Computing's Energy Problem (and what we can do about it)", ISSCC 2014]

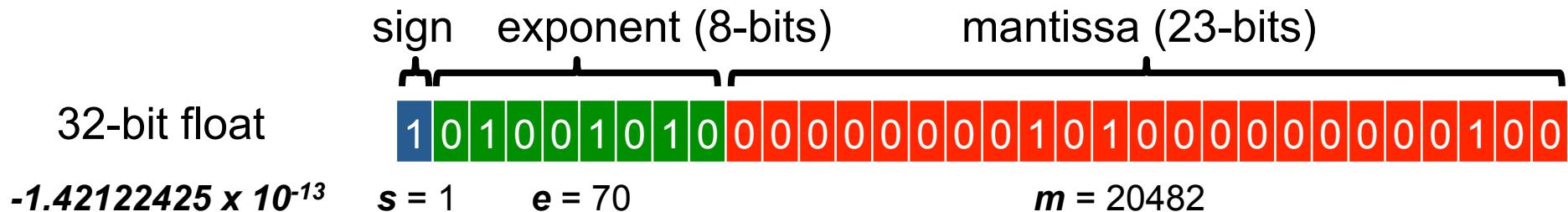
# Number Representation

---

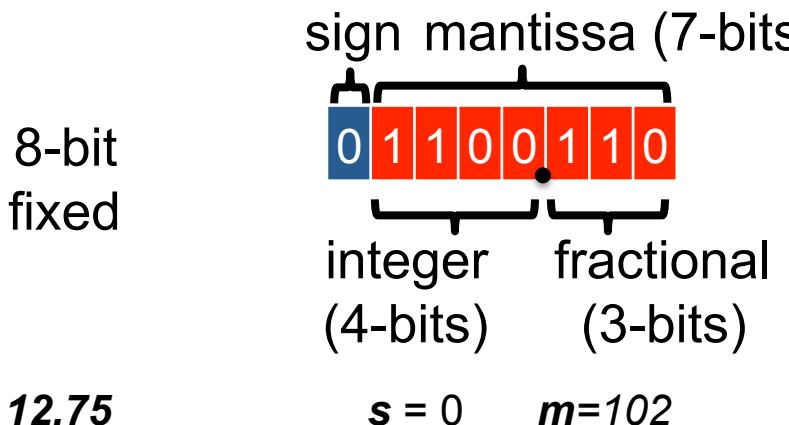
	S	E	M	Range	Accuracy
FP32	1	8	23	$10^{-38} - 10^{38}$	.000006%
FP16	1	5	10	$6 \times 10^{-5} - 6 \times 10^4$	.05%
Int32	1		31	$0 - 2 \times 10^9$	$\frac{1}{2}$
Int16	1		15	$0 - 6 \times 10^4$	$\frac{1}{2}$
Int8	1	7		$0 - 127$	$\frac{1}{2}$

# Floating Point → Fixed Point

## Floating Point

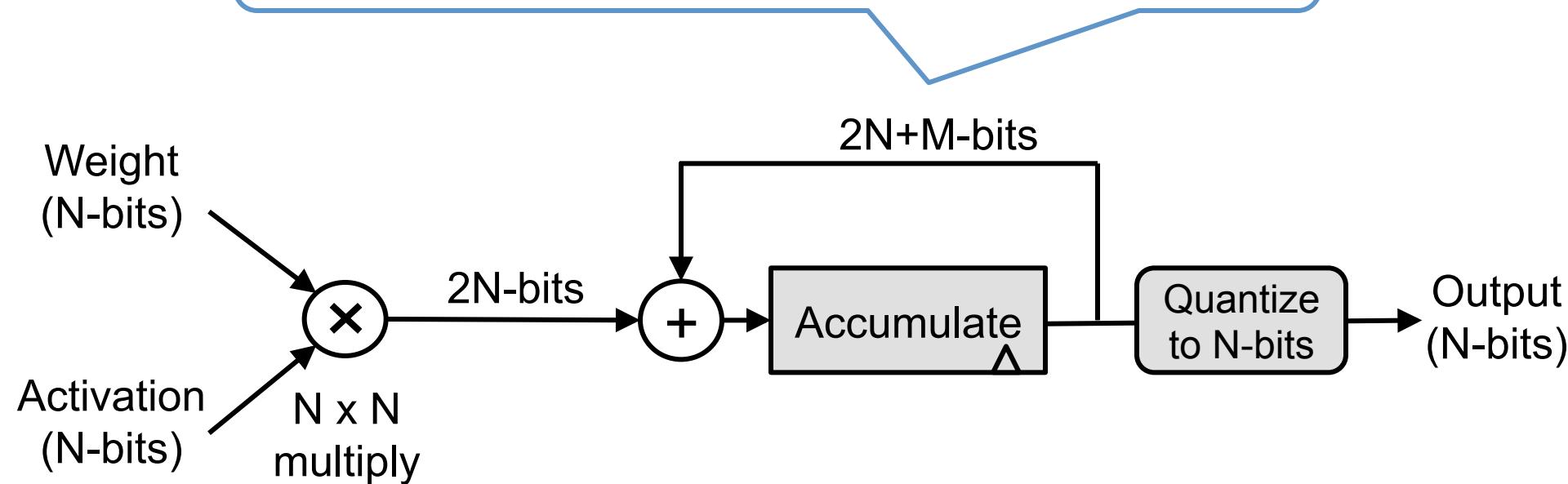


## Fixed Point



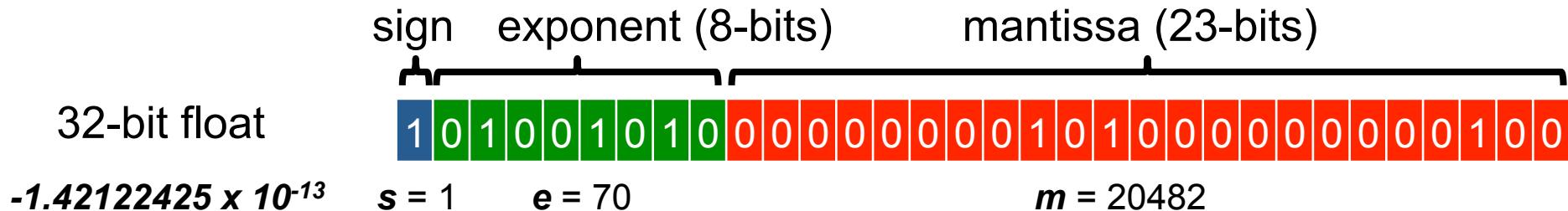
# N-bit Precision

For no loss in precision,  $\mathbf{M}$  is determined based on largest filter size (in the range of 10 to 16 bits for popular DNNs)

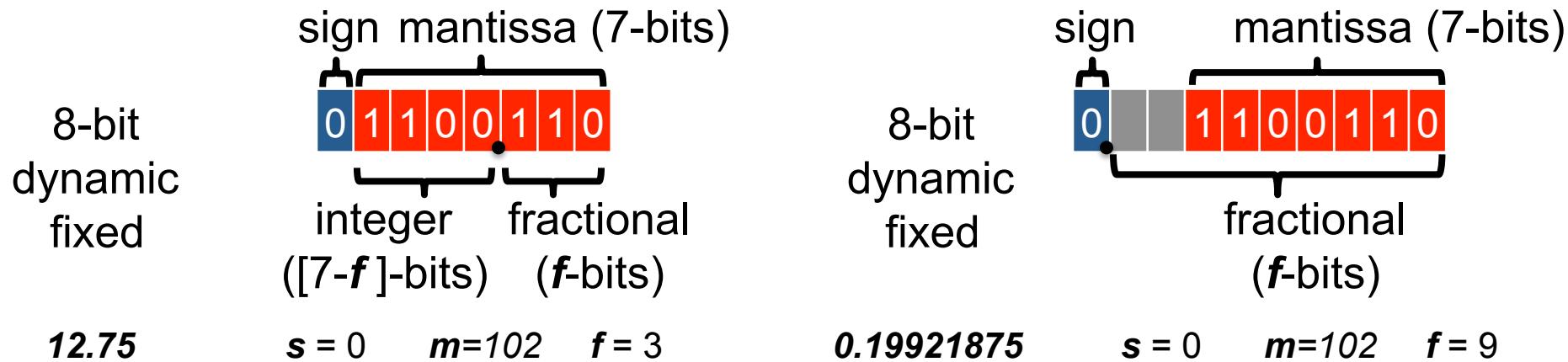


# Dynamic Fixed Point

## Floating Point



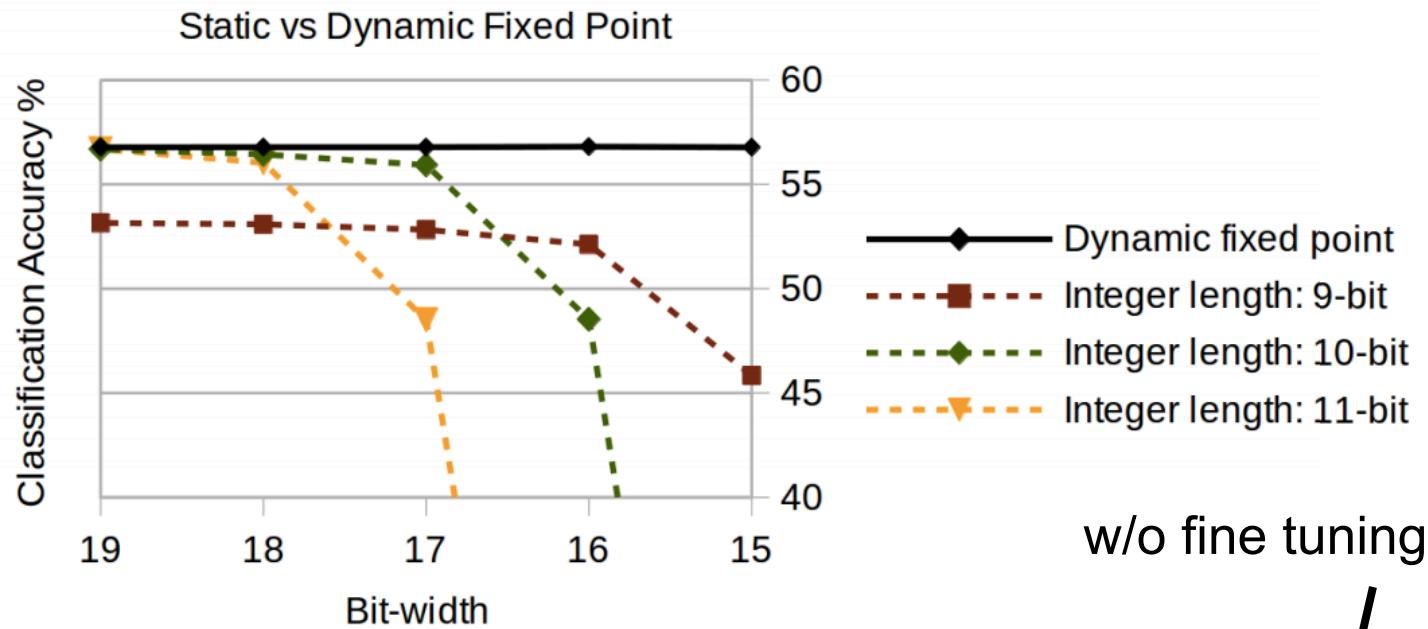
## Fixed Point



Allow  $f$  to vary based on data type and layer

# Impact on Accuracy

Top-1 accuracy  
on of CaffeNet  
on ImageNet



	Layer outputs	CONV parameters	FC parameters	32-bit floating point baseline	Fixed point accuracy
LeNet (Exp 1)	4-bit	4-bit	4-bit	99.1%	99.0% (98.7%)
LeNet (Exp 2)	4-bit	2-bit	2-bit	99.1%	98.8% (98.0%)
Full CIFAR-10	8-bit	8-bit	8-bit	81.7%	81.4% (80.6%)
SqueezeNet top-1	8-bit	8-bit	8-bit	57.7%	57.1% (55.2%)
CaffeNet top-1	8-bit	8-bit	8-bit	56.9%	56.0% (55.8%)
GoogLeNet top-1	8-bit	8-bit	8-bit	68.9%	66.6% (66.1%)

# Avoiding Dynamic Fixed Point

Batch normalization ‘centers’ dynamic range

AlexNet  
(Layer 6)

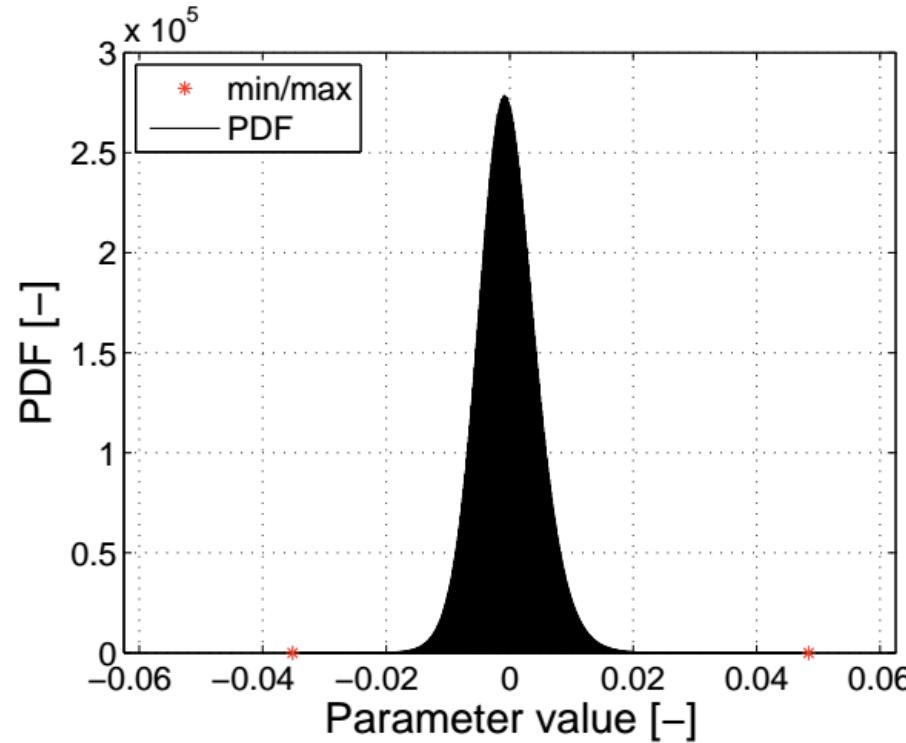


Image Source: Moons  
et al, WACV 2016

‘Centered’ dynamic ranges might reduce need for dynamic fixed point

# Nvidia PASCAL

---

**“New half-precision, 16-bit floating point instructions deliver over 21 TeraFLOPS for unprecedented training performance. With 47 TOPS (tera-operations per second) of performance, new 8-bit integer instructions in Pascal allow AI algorithms to deliver real-time responsiveness for deep learning inference.”**



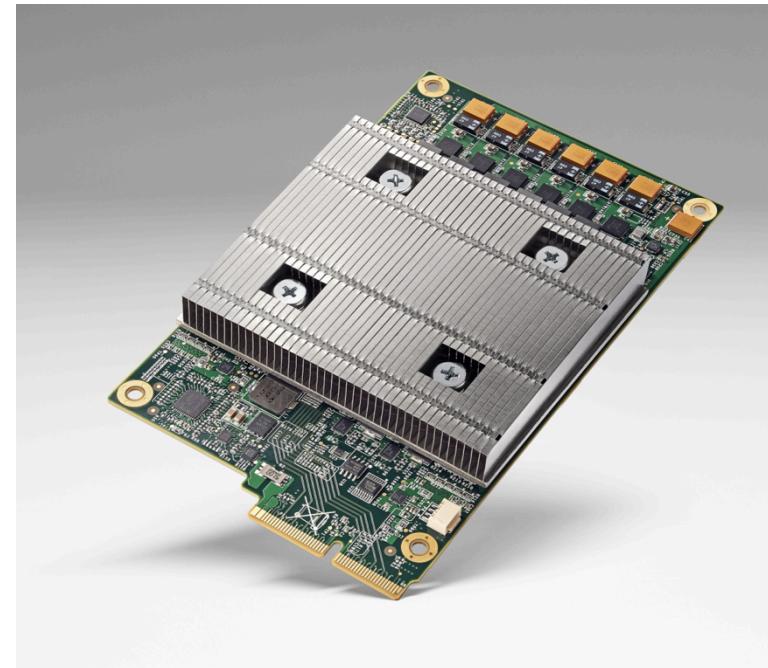
– Nvidia.com (April 2016)

# Google's Tensor Processing Unit (TPU)

“ With its TPU Google has seemingly focused on delivering the data really quickly by **cutting down on precision**. Specifically, it doesn't rely **on floating point precision like a GPU**

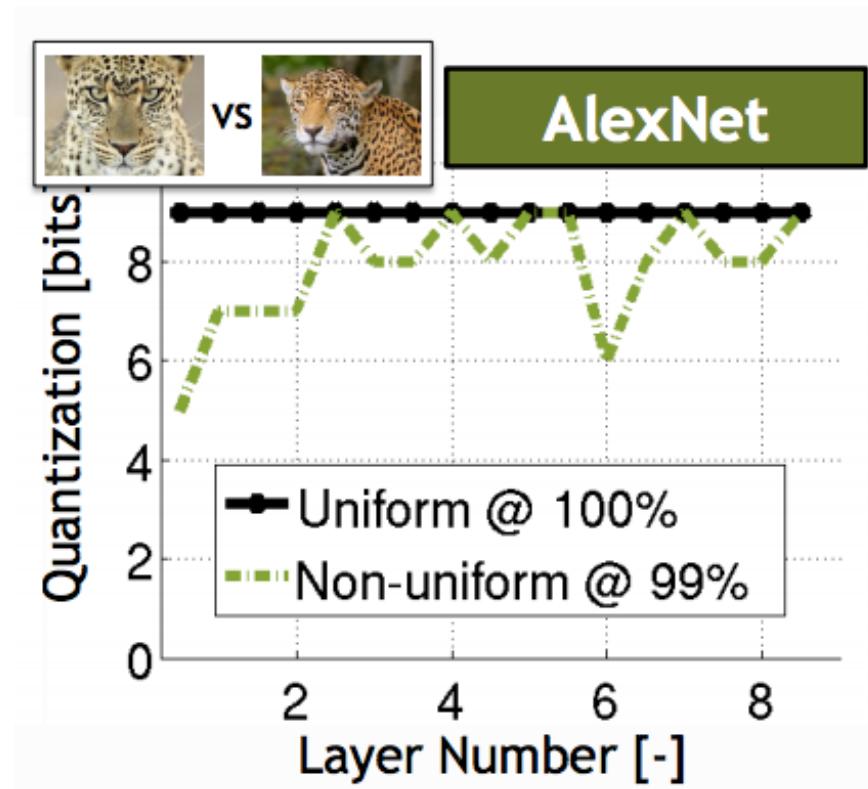
....  
Instead the chip uses integer math...TPU used **8-bit integer**.”

- Next Platform (May 19, 2016)



# Precision Varies from Layer to Layer

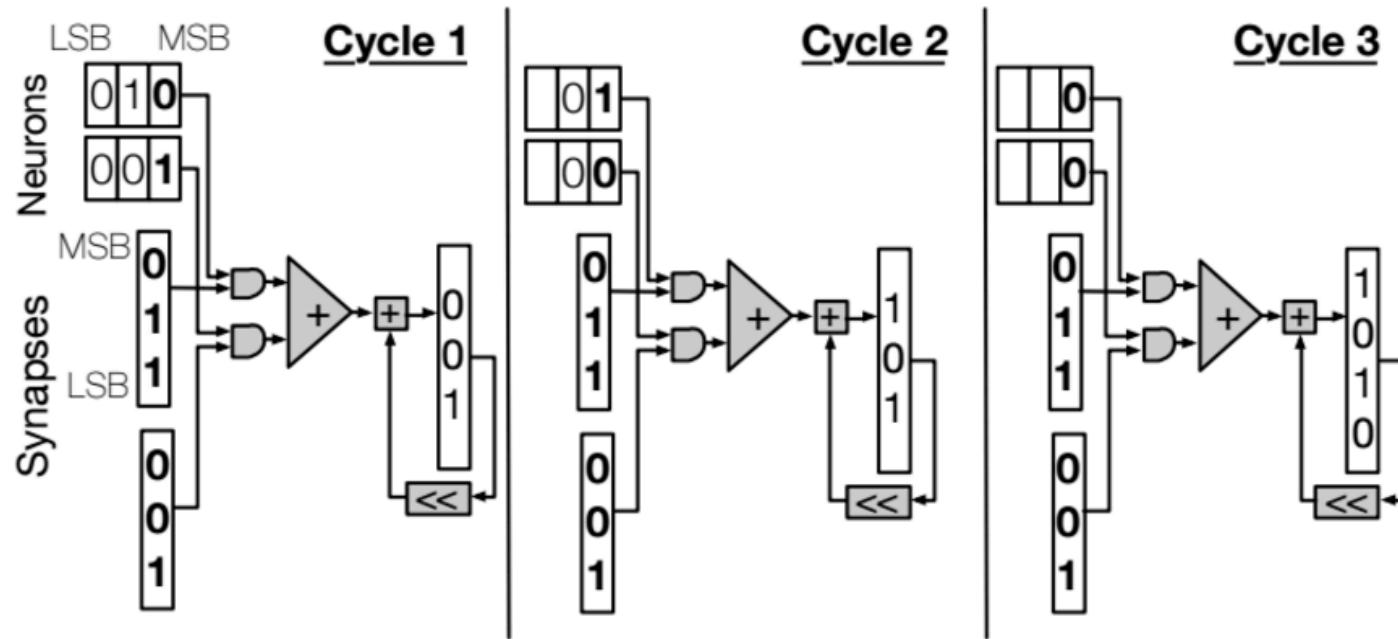
Tolerance	Bits per layer (I+F)
<b>AlexNet (F=0)</b>	
1%	10-8-8-8-8-8-6-4
2%	10-8-8-8-8-8-5-4
5%	10-8-8-8-7-7-5-3
10%	9-8-8-8-7-7-5-3



# Bitwidth Scaling (Speed)

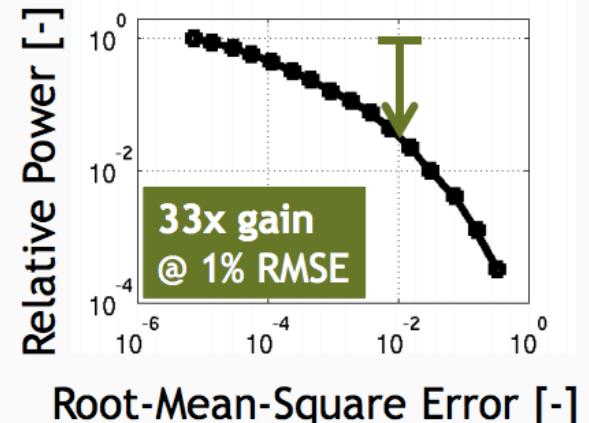
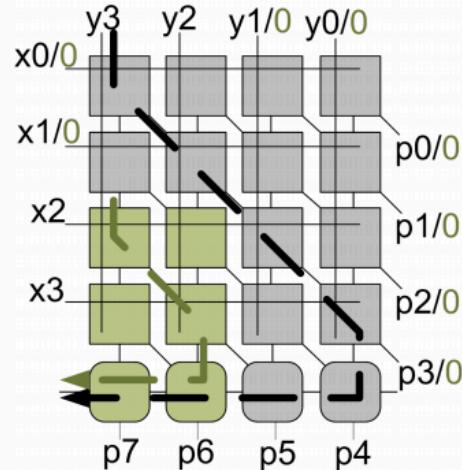
Bit-Serial Processing: Reduce Bit-width → Skip Cycles  
Speed up of 2.24x vs. 16-bit fixed

$$\sum_{i=0}^{N_i-1} s_i \times n_i = \sum_{i=0}^{N_i-1} s_i \times \sum_{b=0}^{P-1} n_i^b \times 2^b = \sum_{b=0}^{P-1} 2^b \times \sum_{i=0}^{N_i-1} n_i^b \times S_i$$



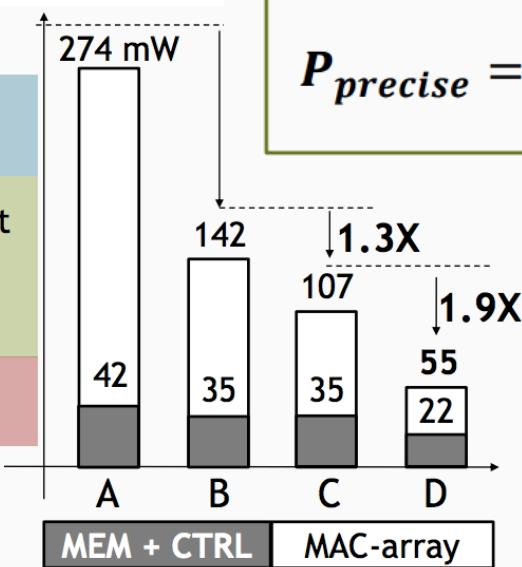
# Bitwidth Scaling (Power)

Reduce Bit-width →  
Shorter Critical Path  
→ Reduce Voltage



AlexNet Layer 2 example:

- A. 2D-baseline @ 16 bit
- B. Precision-Scaling @ 7-7 bit
- C. Voltage-Scaling @ 0.9 V
- D. Sparse operation guarding



$$P_{precise} = \alpha C f V^2 \Rightarrow P_{imprecise} = \frac{\alpha}{k_1} C f \left(\frac{V}{k_2}\right)^2$$

**Power reduction of  
2.56x vs. 16-bit fixed  
On AlexNet Layer 2**

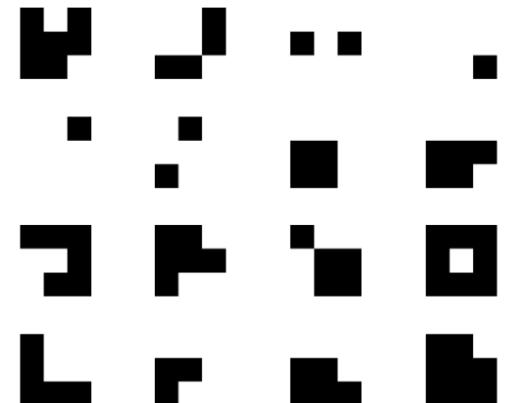
# Binary Nets

---

- **Binary Connect (BC)**

- Weights  $\{-1, 1\}$ , Activations 32-bit float
- MAC  $\rightarrow$  addition/subtraction
- Accuracy loss: **19%** on AlexNet  
[Courbariaux, NIPS 2015]

*Binary Filters*



- **Binarized Neural Networks (BNN)**

- Weights  $\{-1, 1\}$ , Activations  $\{-1, 1\}$
- MAC  $\rightarrow$  XNOR
- Accuracy loss: **29.8%** on AlexNet  
[Courbariaux, arXiv 2016]

# Scale the Weights and Activations

- **Binary Weight Nets (BWN)**

- Weights  $\{-\alpha, \alpha\}$  → except first and last layers are 32-bit float
- Activations: 32-bit float
- $\alpha$  determined by the  $l_1$ -norm of all weights in a layer
- Accuracy loss: **0.8%** on AlexNet

- **XNOR-Net**

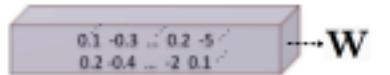
- Weights  $\{-\alpha, \alpha\}$
- Activations  $\{-\beta_i, \beta_i\}$  → except first and last layers are 32-bit float
- $\beta_i$  determined by the  $l_1$ -norm of all activations across channels  
**for given position  $i$**  of the input feature map
- Accuracy loss: **11%** on AlexNet

Hardware needs to support both activation precisions

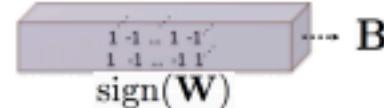
Scale factors ( $\alpha, \beta_i$ ) can change per layer or position in filter

# XNOR-Net

## (1) Binarizing Weight

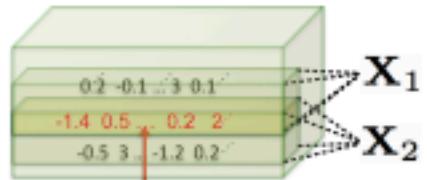


$$\frac{1}{n} \|\mathbf{W}\|_{\ell_1} = \alpha$$

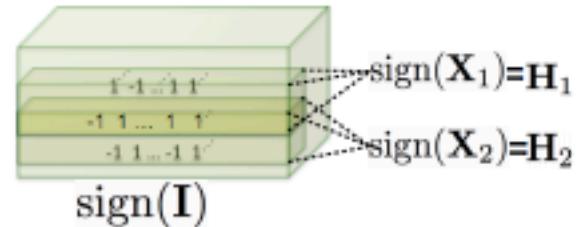


## (2) Binarizing Input

### Inefficient

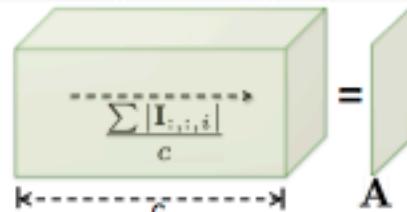


$$\frac{1}{n} \|\mathbf{X}_1\|_{\ell 1} = \beta_1$$

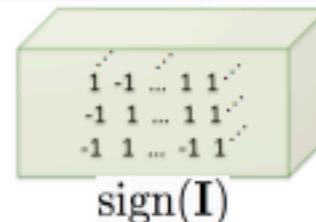


### (3) Binarizing Input

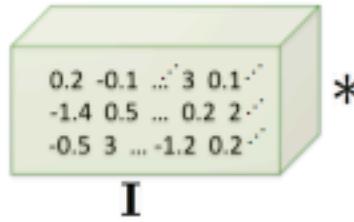
## Efficient



$$A *_{\mathbf{k}} K = \beta_1 \quad \beta_2$$



#### (4) Convolution with XNOR-Bitcount



\*  $\begin{pmatrix} 0.1 & -0.3 & 0.2 & -5 \\ 0.2 & -0.4 & -2 & 0.1 \end{pmatrix}$

# Ternary Nets

---

- **Allow for weights to be zero**
  - Increase sparsity, but also increase number of bits (2-bits)
- **Ternary Weight Nets (TWN)** [Li et al., arXiv 2016]
  - Weights  $\{-w, 0, w\}$  → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - Accuracy loss: **3.7%** on AlexNet
- **Trained Ternary Quantization (TTQ)** [Zhu et al., ICLR 2017]
  - Weights  $\{-w_1, 0, w_2\}$  → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - Accuracy loss: **0.6%** on AlexNet

# Non-Linear Quantization

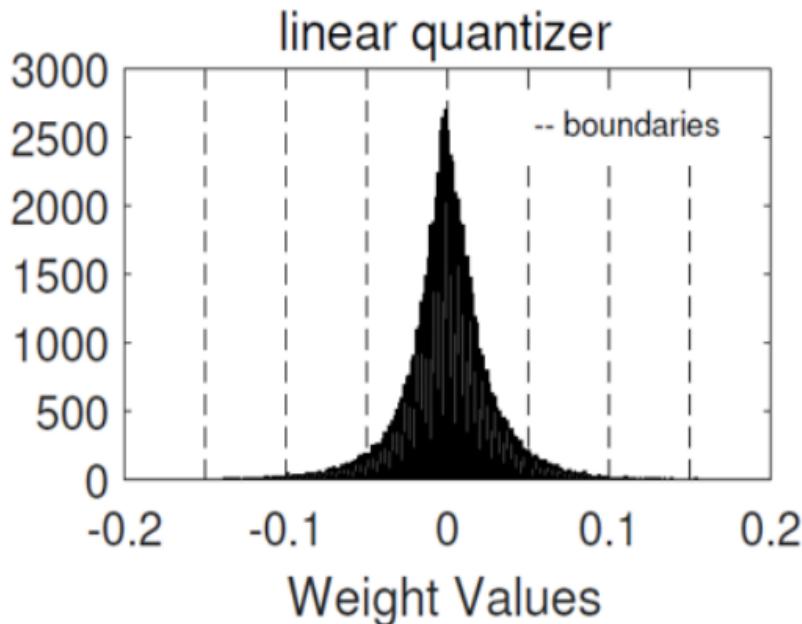
---

- **Precision** refers to the **number of levels**
  - Number of bits =  $\log_2$  (number of levels)
- **Quantization:** mapping data to a smaller set of **levels**
  - Linear, e.g., fixed-point
  - Non-linear
    - Computed
    - Table lookup

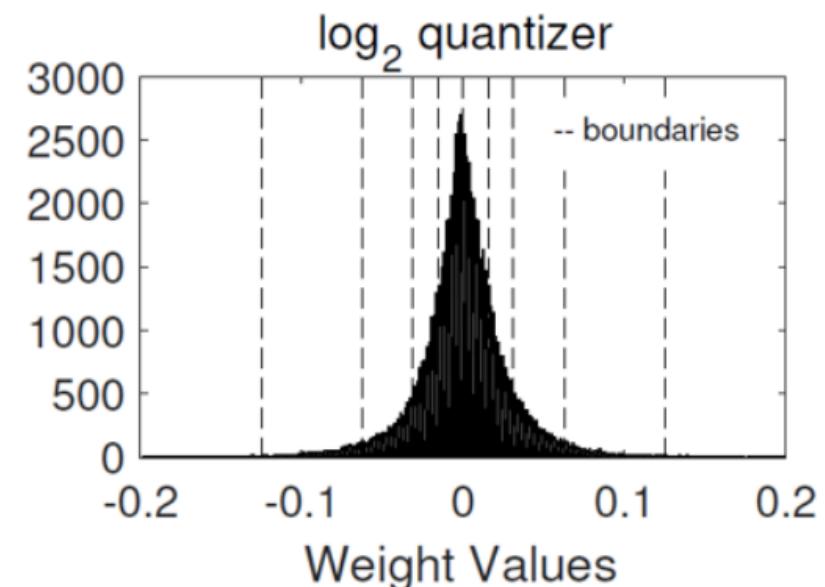
Objective: Reduce size to improve speed and/or reduce energy while preserving accuracy

# Computed Non-linear Quantization

## Log Domain Quantization



$$\text{Product} = X * W$$

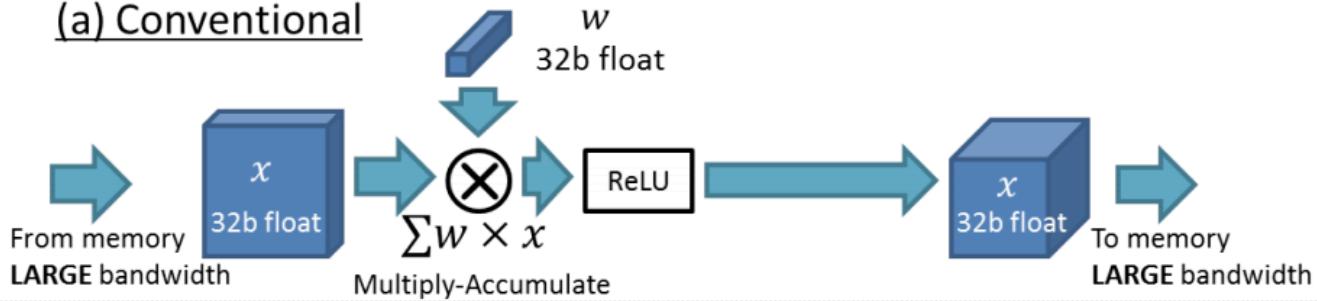


$$\text{Product} = X \ll W$$

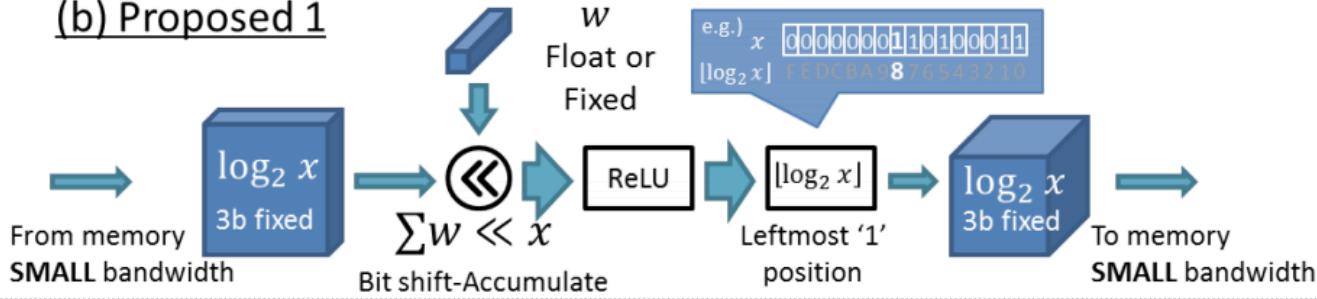
[Lee et al., LogNet, ICASSP 2017]

# Log Domain Computation

(a) Conventional

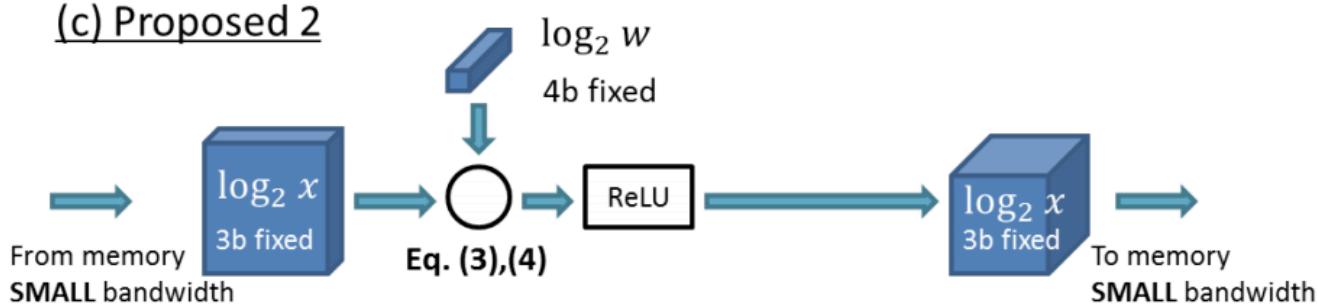


(b) Proposed 1



Only activation in log domain

(c) Proposed 2



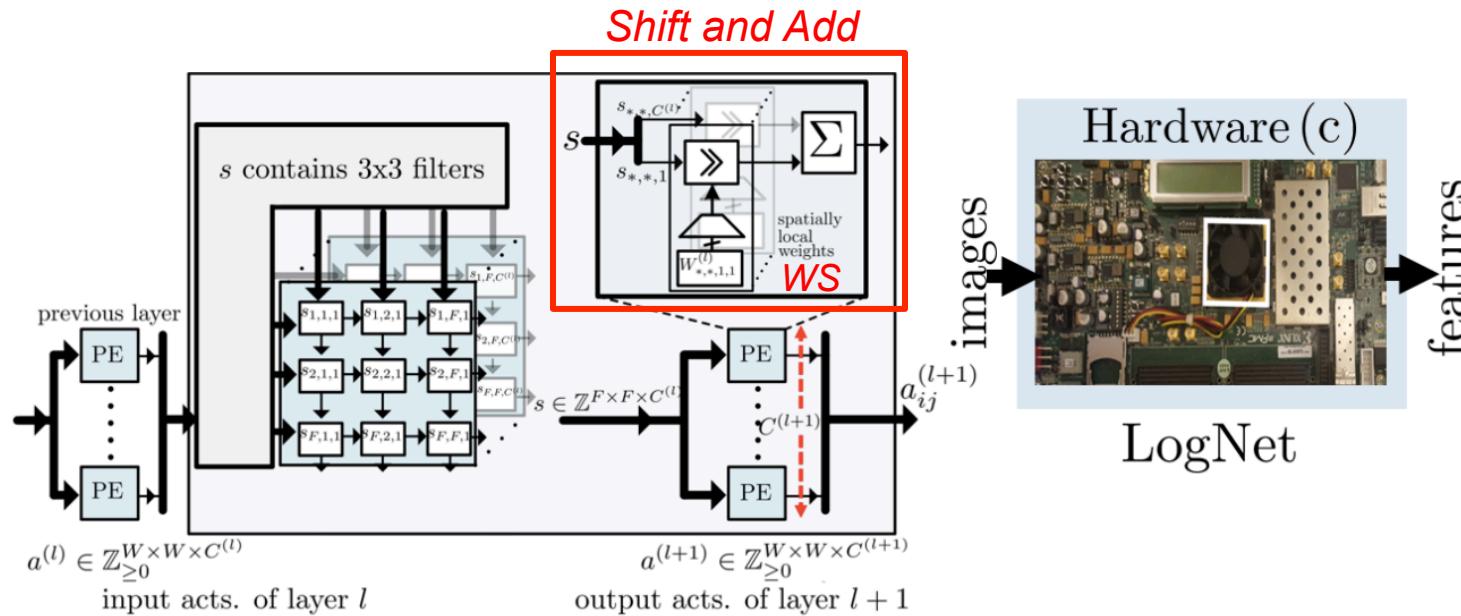
Both weights and activations in log domain

max, bitshifts, adds/subs

[Miyashita et al., arXiv 2016]

# Log Domain Quantization

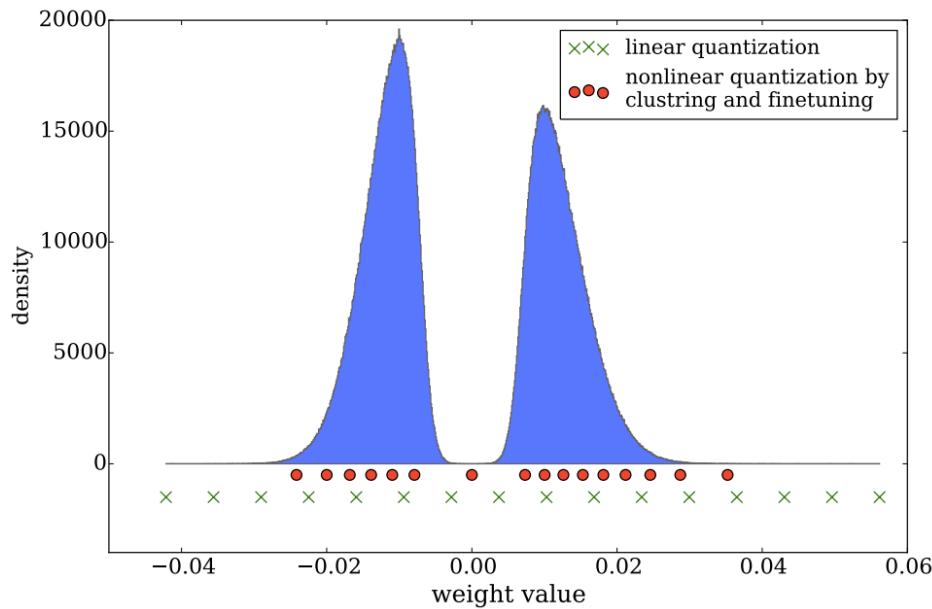
- Weights: 5-bits for CONV, 4-bit for FC; Activations: 4-bits
- Accuracy loss: **3.2%** on AlexNet



[Miyashita et al., arXiv 2016],  
[Lee et al., LogNet, ICASSP 2017]

# Reduce Precision Overview

- Learned mapping of data to quantization levels (e.g., k-means)



*Implement with  
look up table*

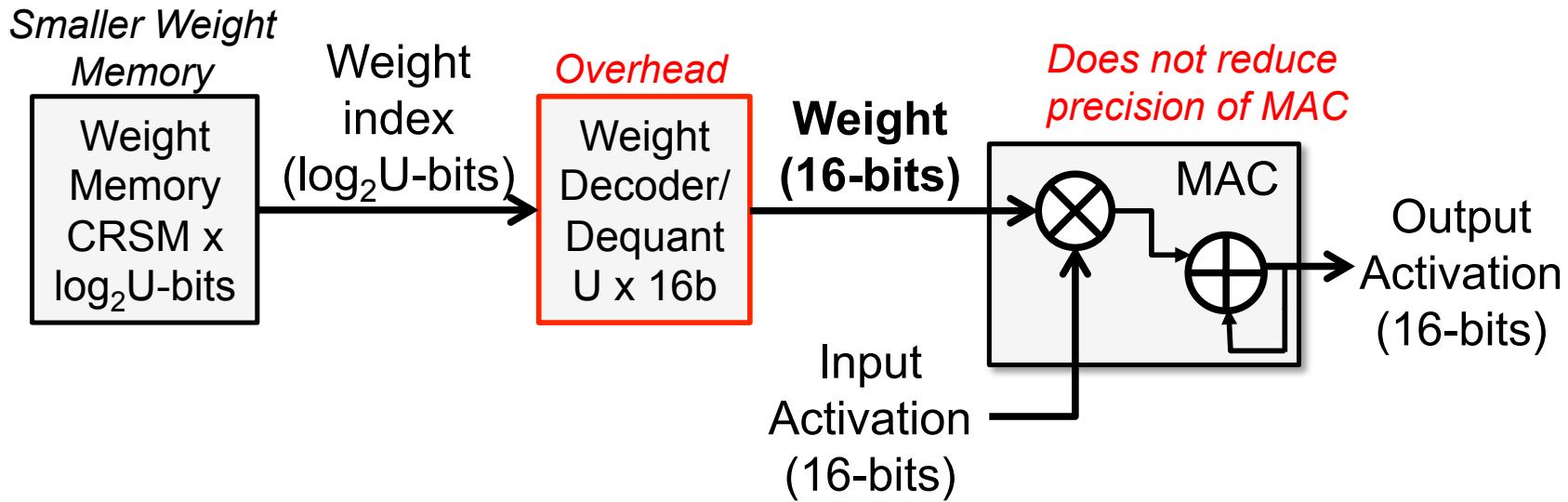
[Han et al., ICLR 2016]

- Additional Properties
  - Fixed or Variable (across data types, layers, channels, etc.)

# Non-Linear Quantization Table Lookup

**Trained Quantization:** Find K weights via K-means clustering to reduce number of unique weights *per layer* (weight sharing)

**Example:** AlexNet (no accuracy loss)  
**256 unique weights** for CONV layer  
**16 unique weights** for FC layer



Consequences: Narrow weight memory and second access from (small) table

# Summary of Reduce Precision

Category	Method	Weights (# of bits)	Activations (# of bits)	Accuracy Loss vs. 32-bit float (%)
Dynamic Fixed Point	w/o fine-tuning	8	10	0.4
	w/ fine-tuning	8	8	0.6
Reduce weight	Ternary weights Networks (TWN)	2*	32	3.7
	Trained Ternary Quantization (TTQ)	2*	32	0.6
	Binary Connect (BC)	1	32	19.2
	Binary Weight Net (BWN)	1*	32	0.8
Reduce weight and activation	Binarized Neural Net (BNN)	1	1	29.8
	XNOR-Net	1*	1	11
Non-Linear	LogNet	5(conv), 4(fc)	4	3.2
	Weight Sharing	8(conv), 4(fc)	16	0

\* first and last layers are 32-bit float

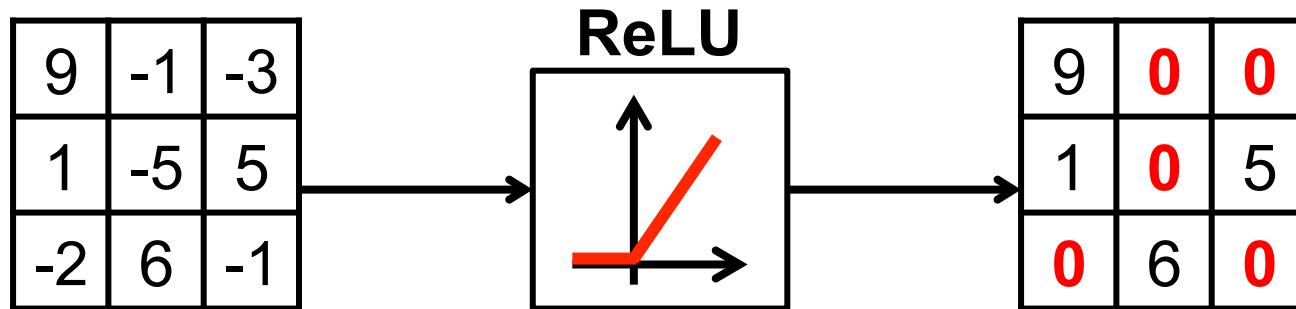
# Reduce Number of Ops and Weights

---

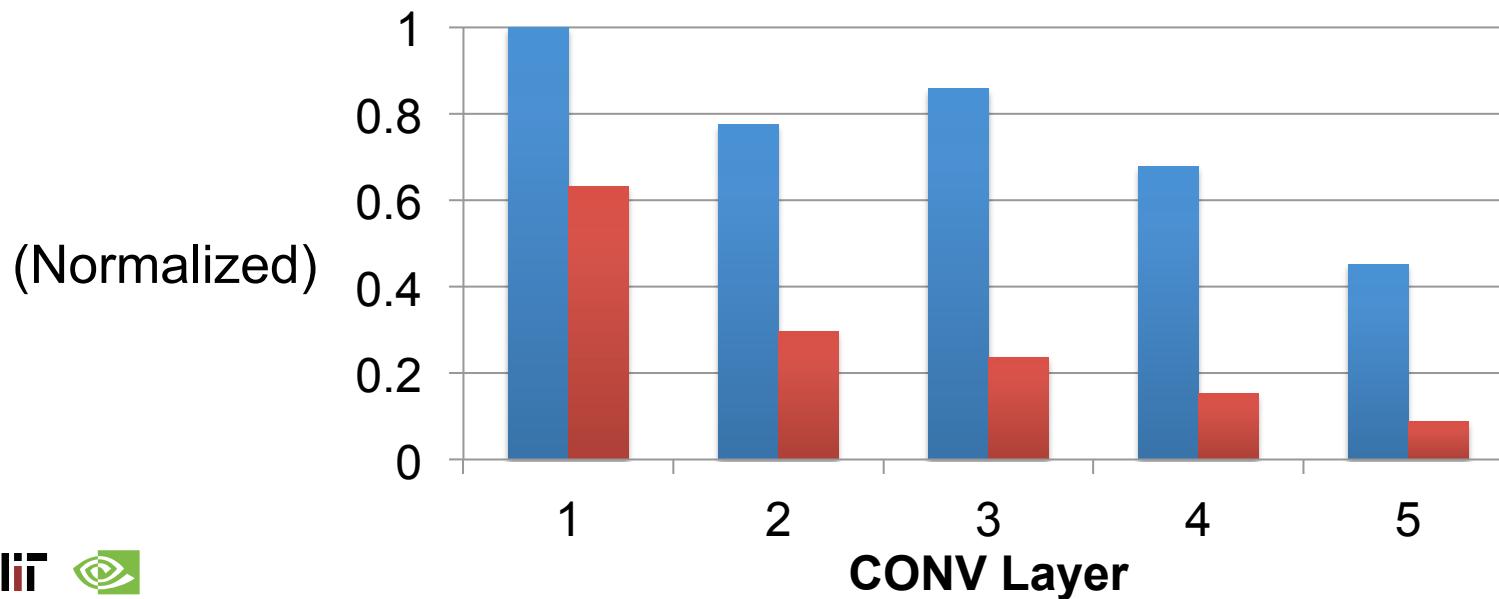
- **Exploit Activation Statistics**
- **Network Pruning**
- **Compact Network Architectures**
- **Knowledge Distillation**

# Sparsity in Fmaps

Many **zeros** in output fmaps after ReLU



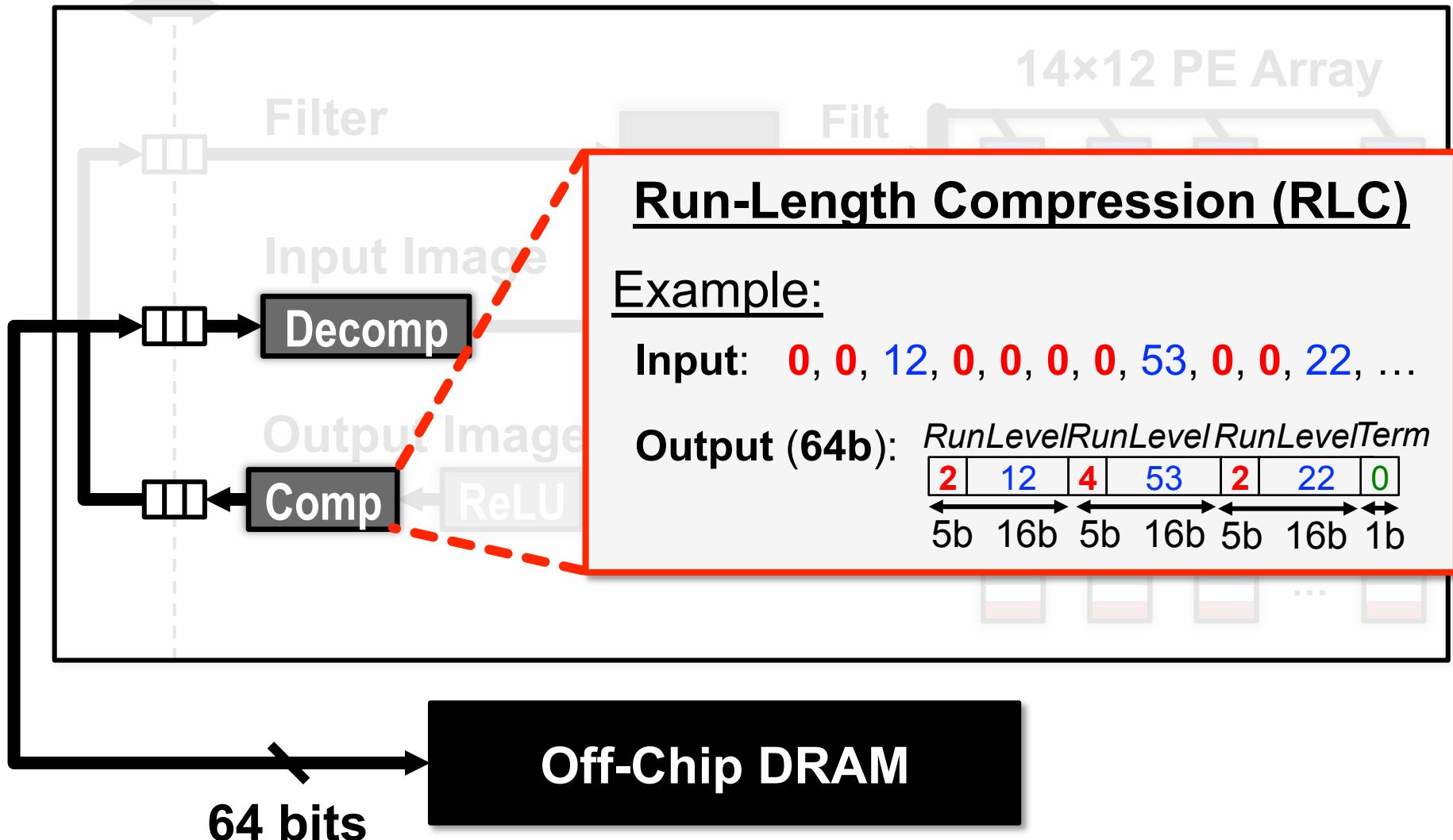
■ # of activations ■ # of non-zero activations



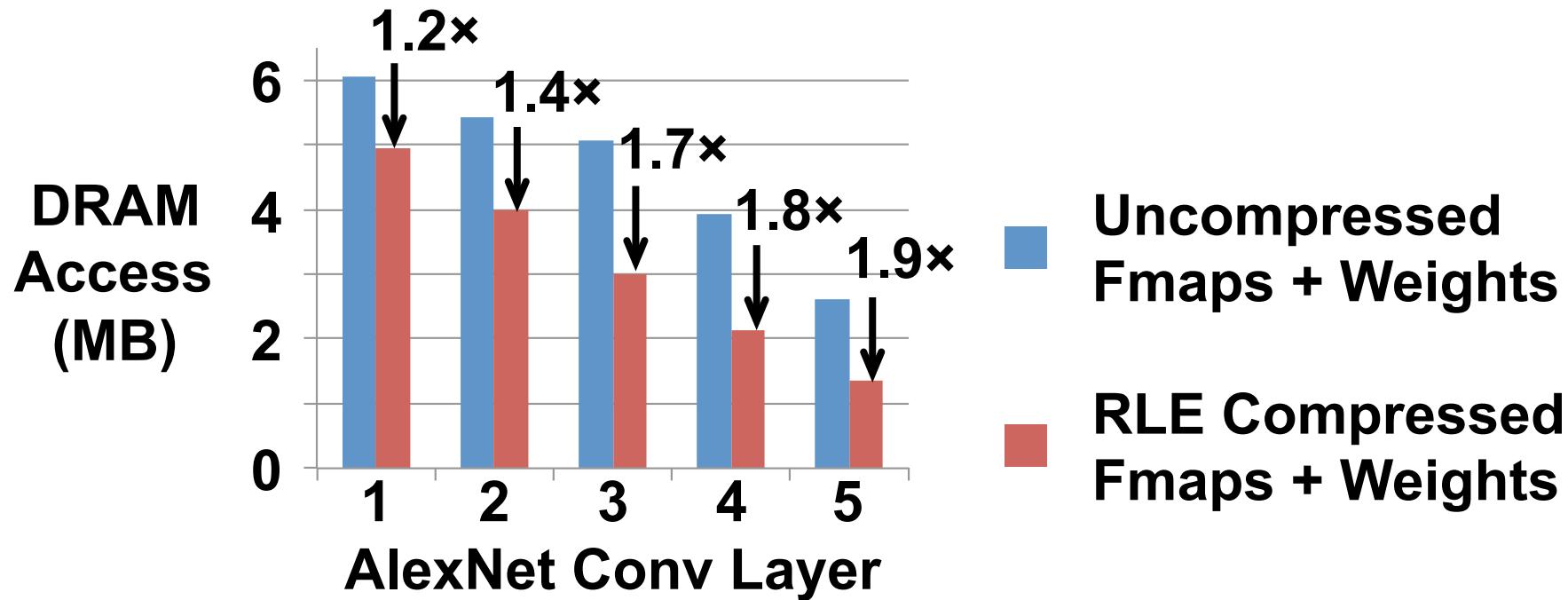
# I/O Compression in Eyeriss

Link Clock : Core Clock

DCNN Accelerator

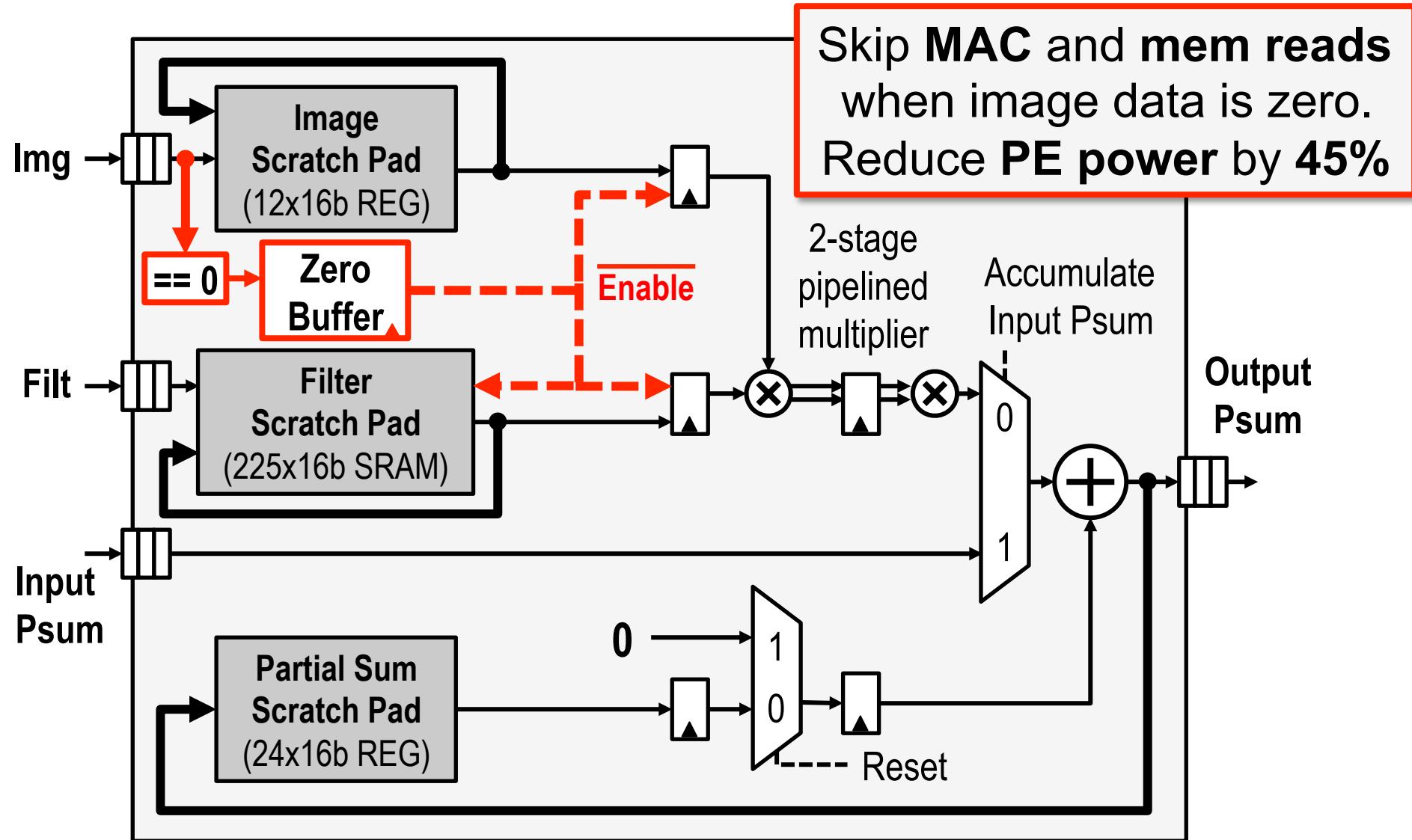


# Compression Reduces DRAM BW



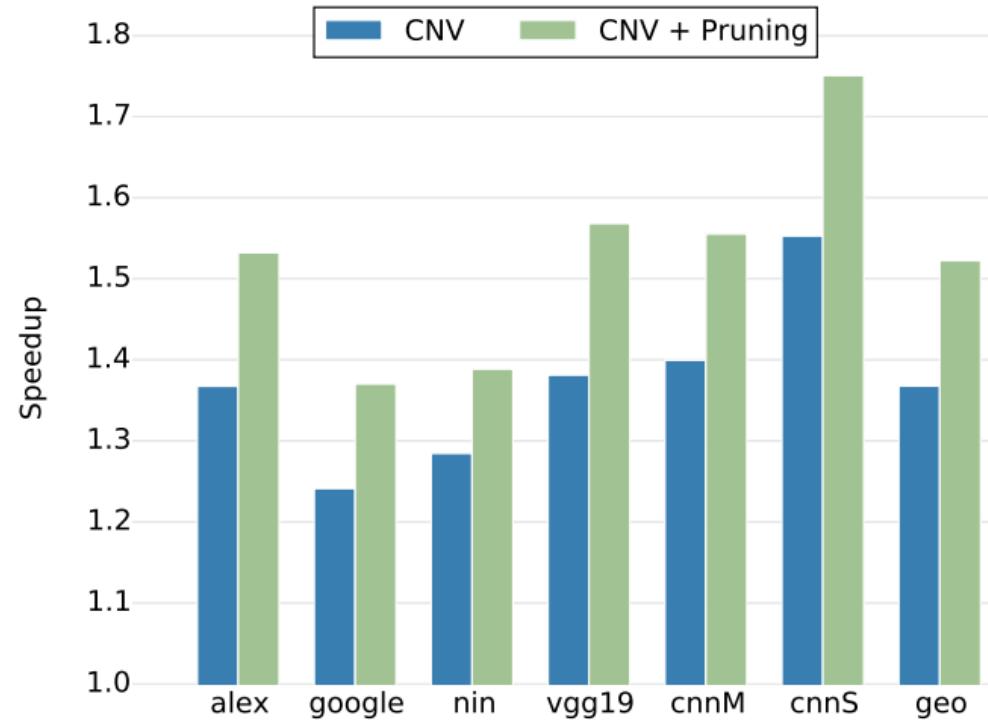
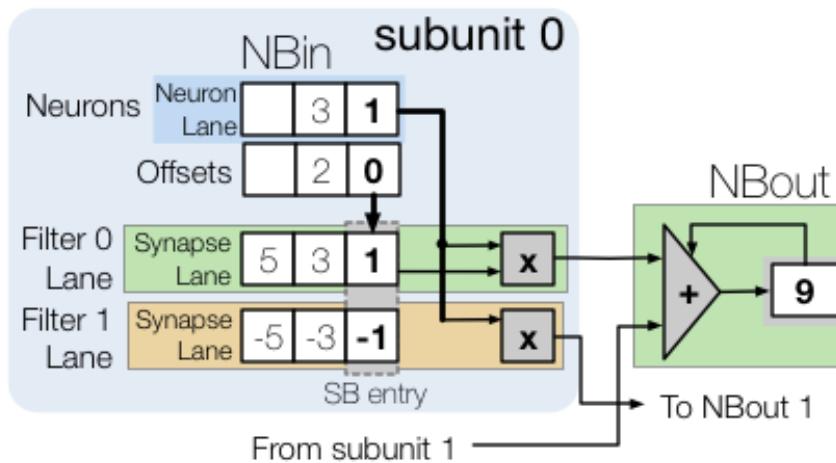
Simple RLC within 5% - 10% of theoretical entropy limit

# Data Gating / Zero Skipping in Eyeriss



# Cnvlutin

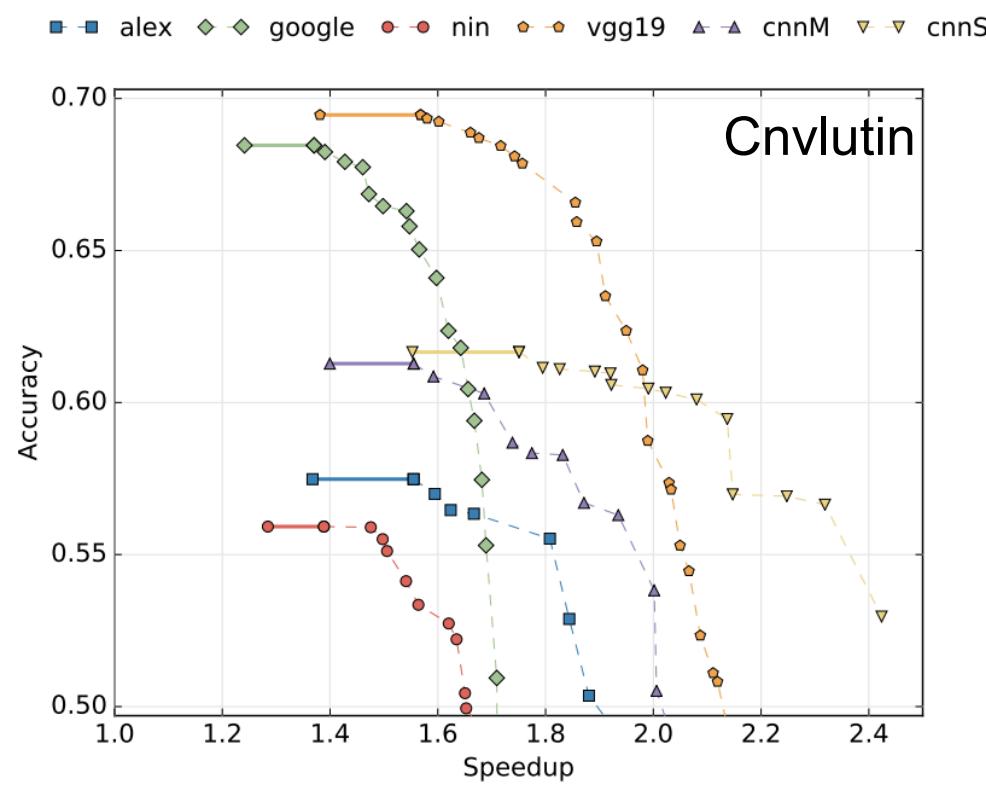
- Process Convolution Layers
- Built on top of DaDianNao (4.49% area overhead)
- Speed up of 1.37x (1.52x with activation pruning)



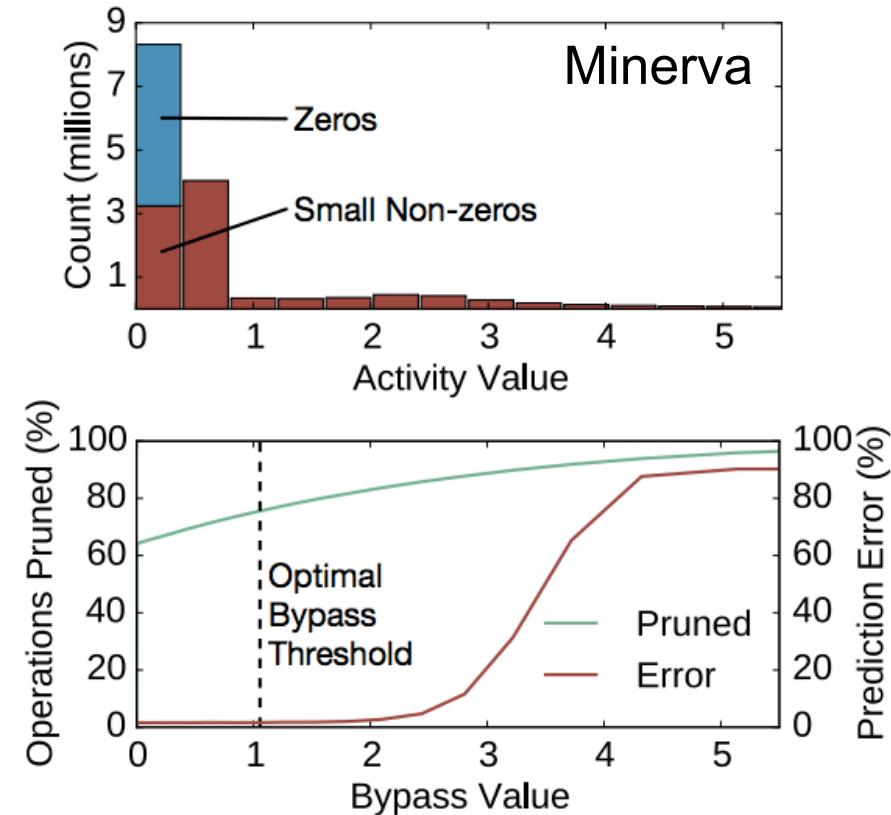
# Pruning Activations

Remove small activation values

*Speed up 11% (ImageNet)*

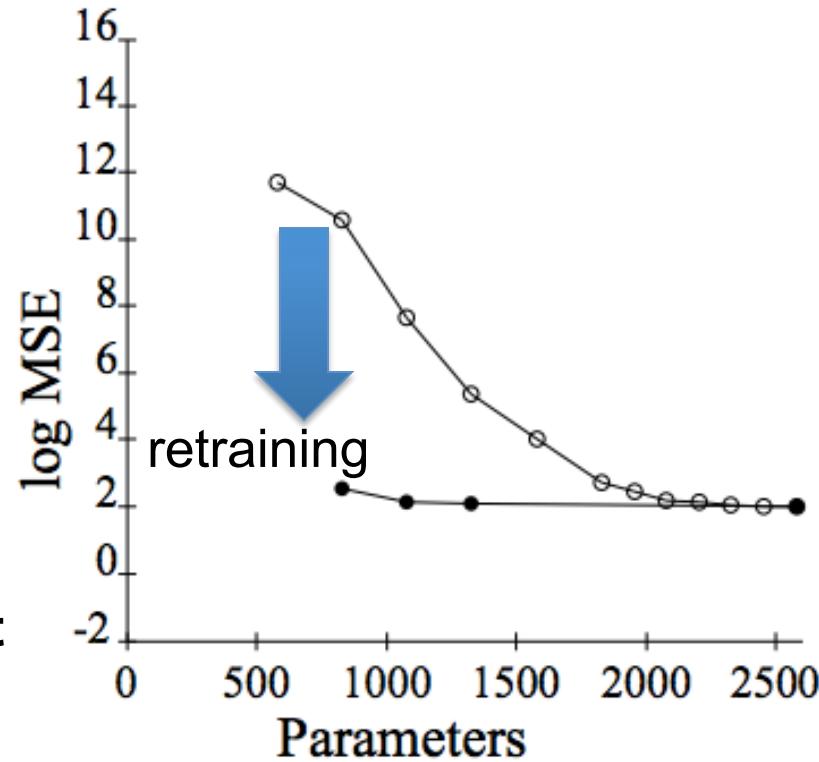


*Reduce power 2x (MNIST)*



# Pruning – Make Weights Sparse

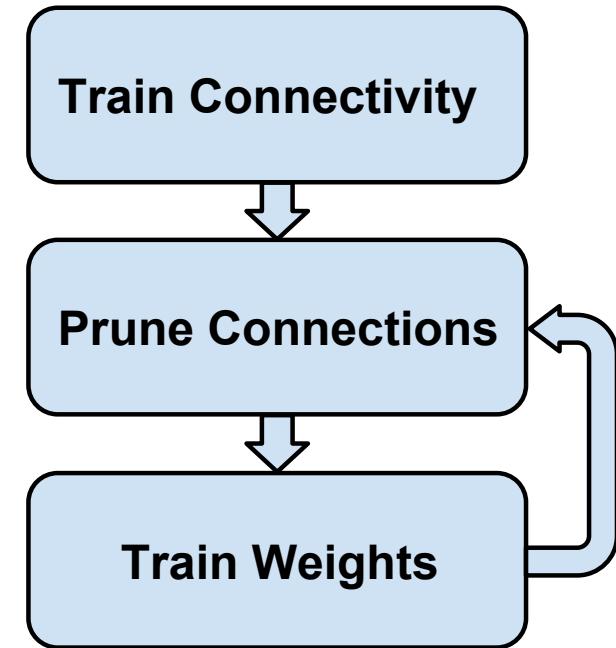
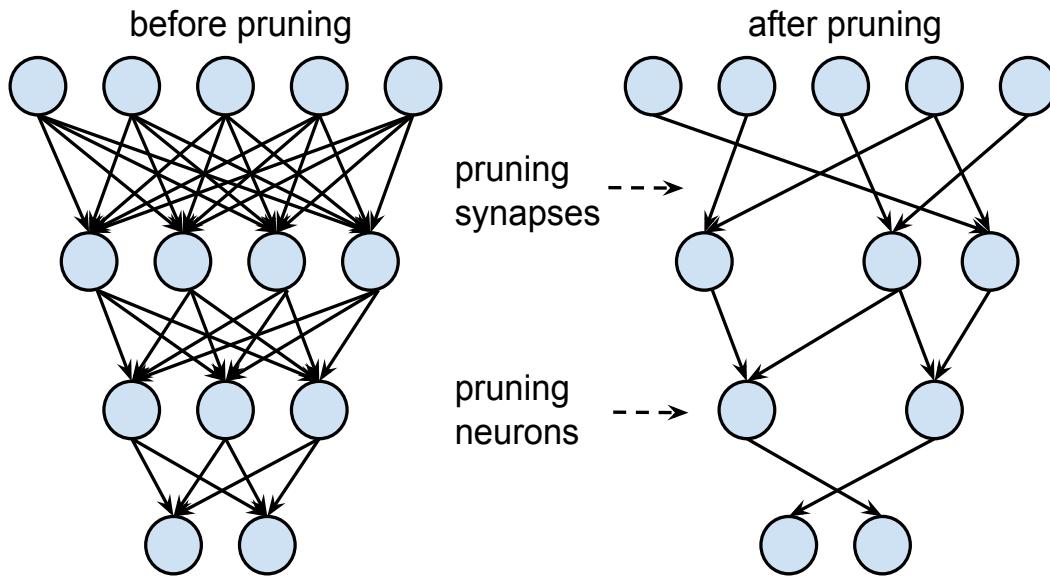
- **Optimal Brain Damage**
  1. **Choose a reasonable network architecture**
  2. **Train network until reasonable solution obtained**
  3. **Compute the second derivative for each weight**
  4. **Compute saliencies (i.e. impact on training error) for each weight**
  5. **Sort weights by saliency and delete low-saliency weights**
  6. **Iterate to step 2**



[Lecun et al., NIPS 1989]

# Pruning – Make Weights Sparse

Prune based on *magnitude* of weights



**Example:** AlexNet

**Weight Reduction:** CONV layers 2.7x, FC layers 9.9x

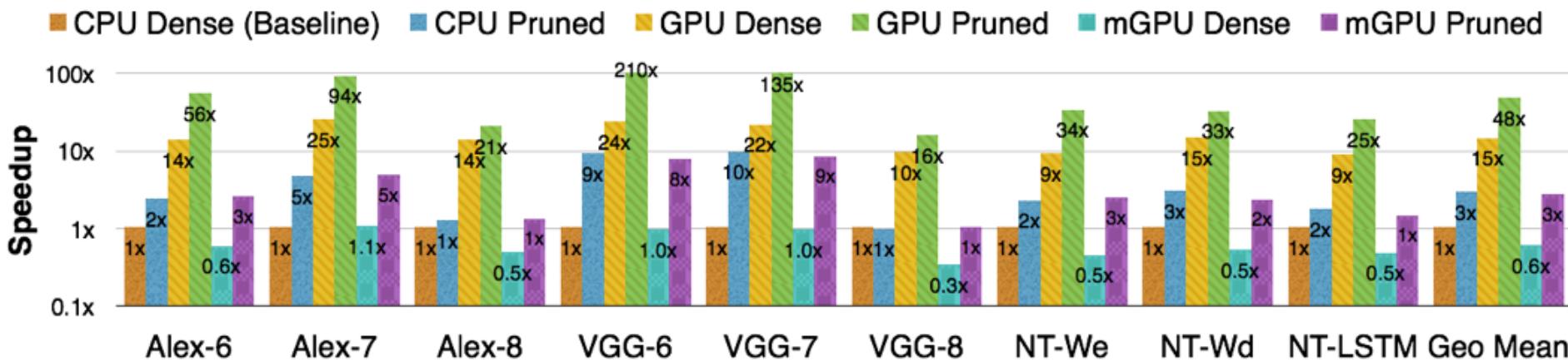
*(Most reduction on fully connected layers)*

**Overall:** 9x weight reduction, 3x MAC reduction

# Speed up of Weight Pruning on CPU/GPU

On Fully Connected Layers Only

Average Speed up of 3.2x on GPU, 3x on CPU, 5x on mGPU



Intel Core i7 5930K: MKL CBLAS GEMV, MKL SPBLAS CSRMV  
NVIDIA GeForce GTX Titan X: cuBLAS GEMV, cuSPARSE CSRMV  
NVIDIA Tegra K1: cuBLAS GEMV, cuSPARSE CSRMV

Batch size = 1

# Key Metrics for Embedded DNN

---

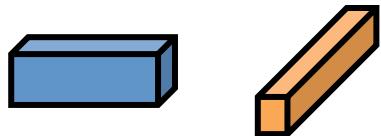
- Accuracy → Measured on Dataset
- Speed → Number of MACs
- Storage Footprint → Number of Weights
- Energy → ?

# Energy-Aware Pruning

---

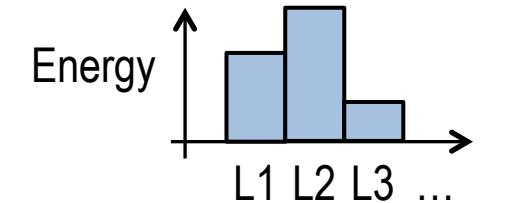
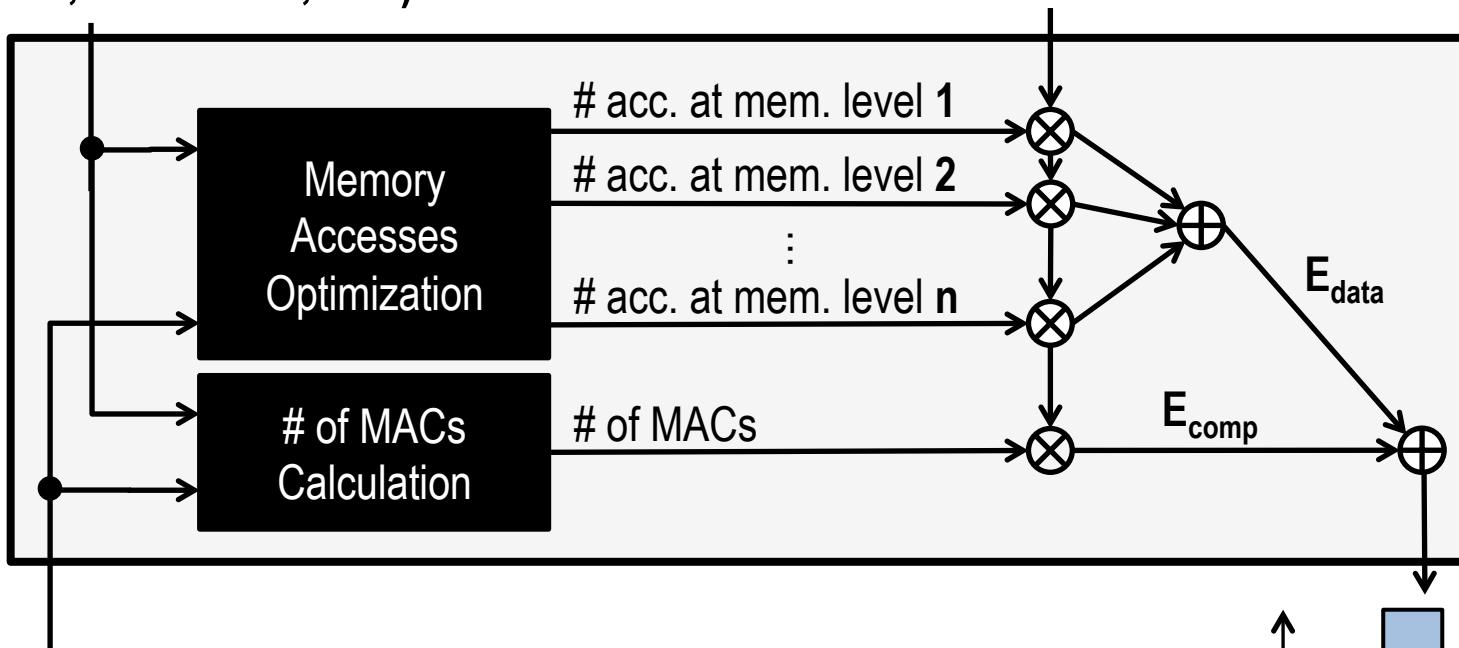
- **# of Weights alone is not a good metric for energy**
  - Example (AlexNet):
    - # of Weights (FC Layer) > # of Weights (CONV layer)
    - Energy (FC Layer) < Energy (CONV layer)
- **Use energy evaluation method to estimate DNN energy**
  - Account for data movement

# Energy-Evaluation Methodology



**CNN Shape Configuration**  
(# of channels, # of filters, etc.)

**Hardware Energy Costs of each MAC and Memory Access**



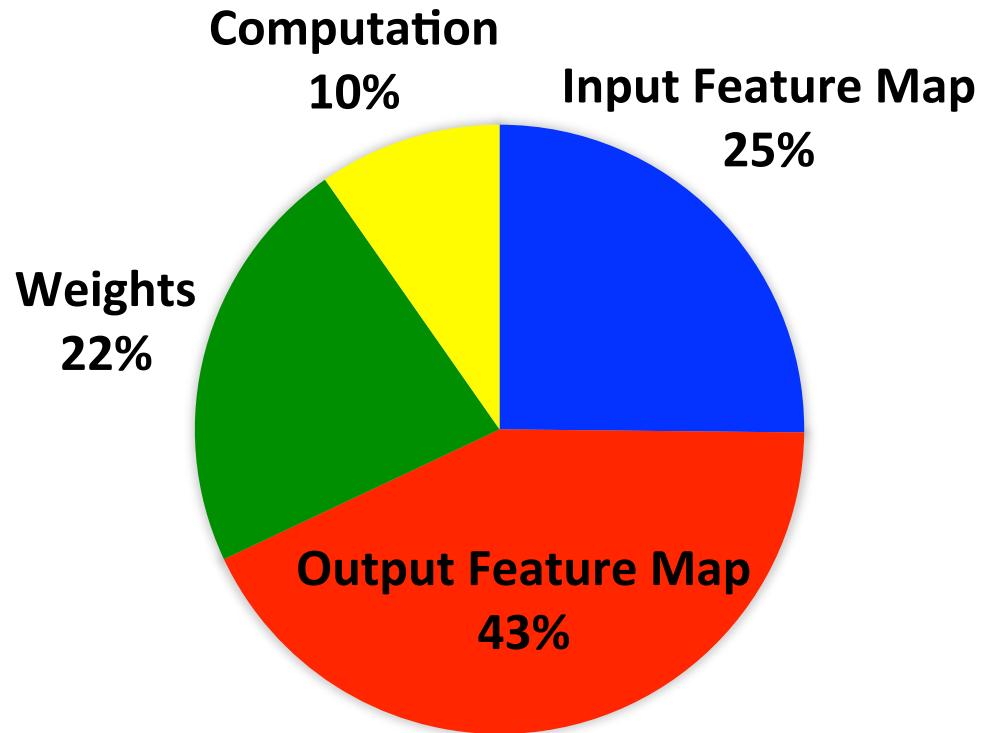
**CNN Energy Consumption**

# Key Observations

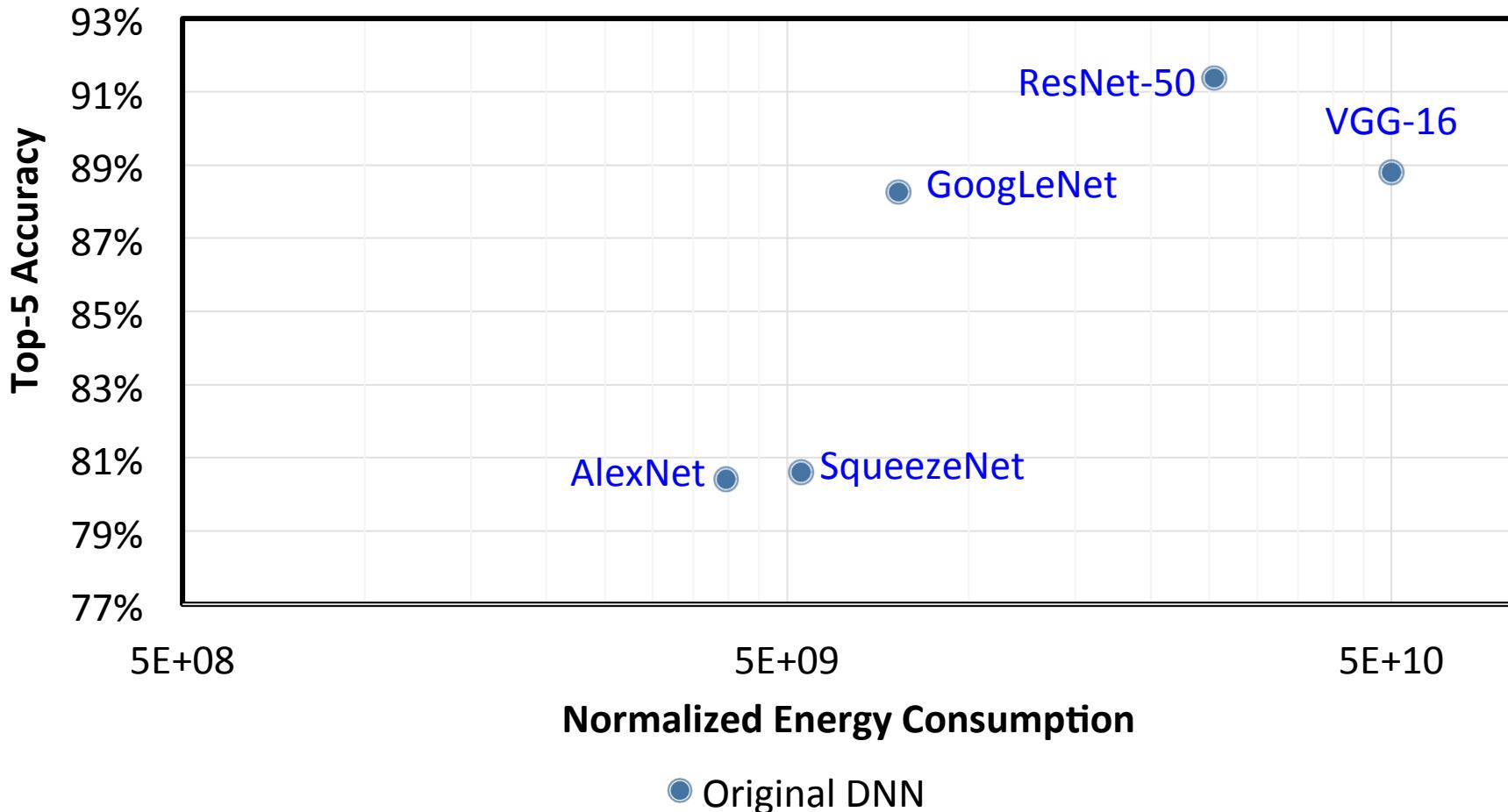
---

- Number of weights **alone** is not a good metric for energy
- **All data types** should be considered

**Energy Consumption  
of GoogLeNet**

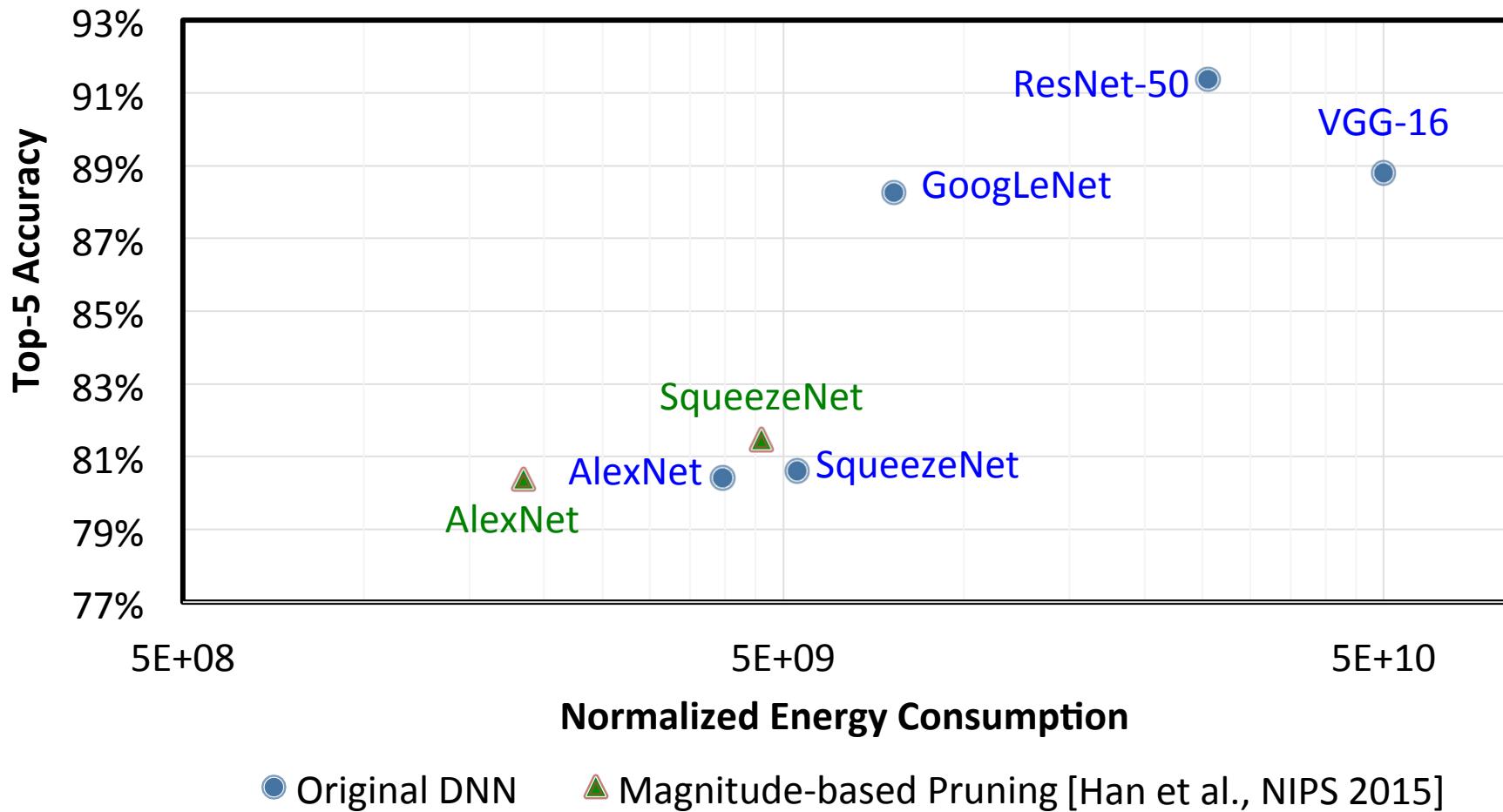


# Energy Consumption of Existing DNNs



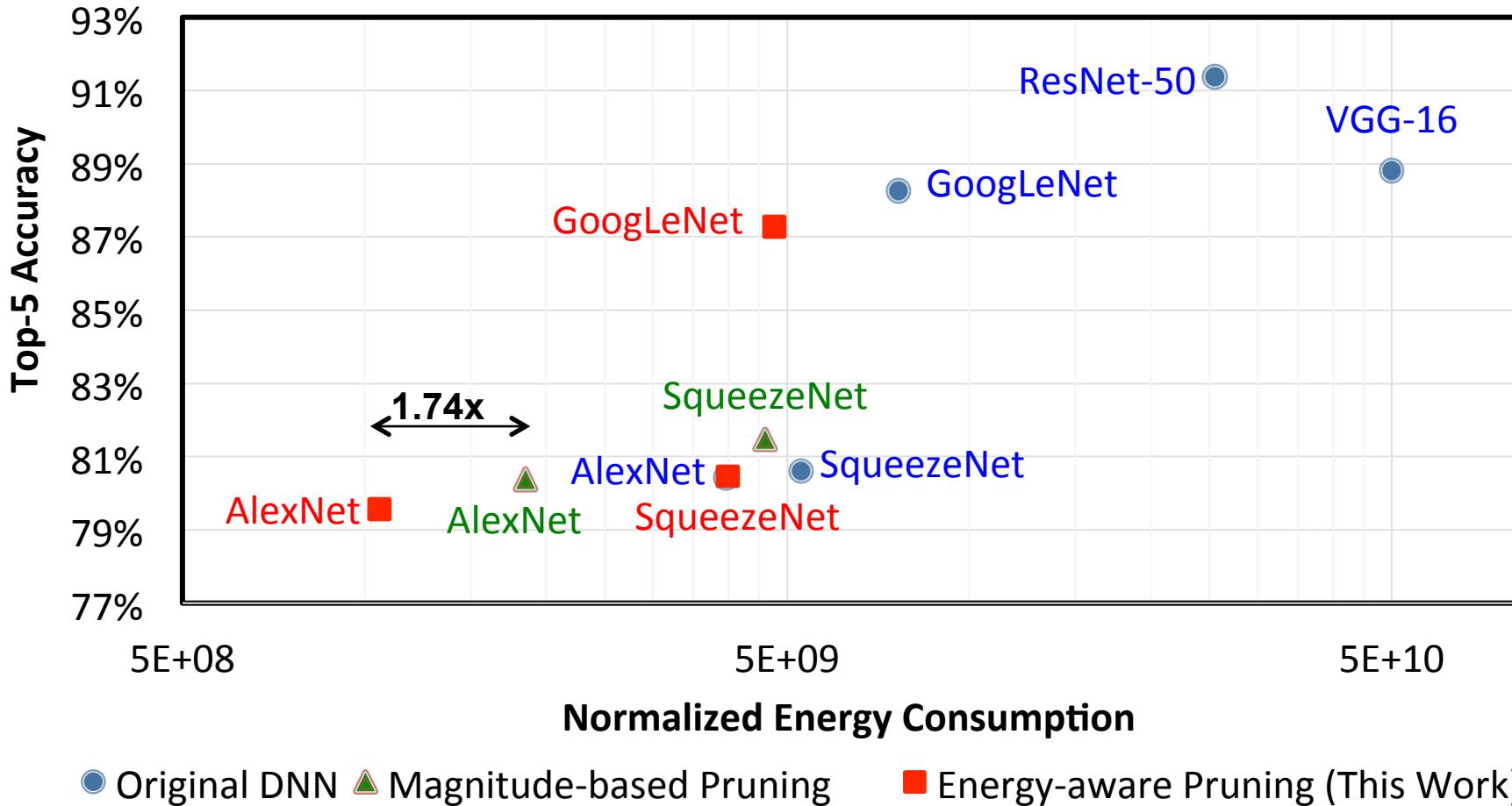
Deeper CNNs with fewer weights do not necessarily consume less energy than shallower CNNs with more weights

# Magnitude-based Weight Pruning



Reduce number of weights by **removing small magnitude weights**

# Energy-Aware Pruning



Remove weights from layers **in order of highest to lowest energy**  
**3.7x reduction in AlexNet / 1.6x reduction in GoogLeNet**

# Energy Estimation Tool

Website: <https://energyestimation.mit.edu/>

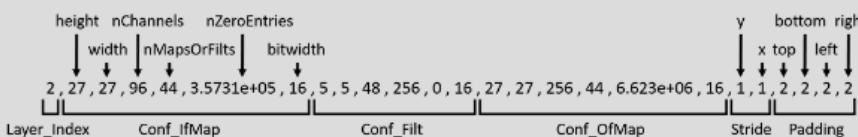
## Deep Neural Network Energy Estimation Tool

### Overview

This Deep Neural Network Energy Estimation Tool is used for evaluating and designing energy-efficient deep neural networks that are critical for embedded deep learning processing. Energy estimation was used in the development of the energy-aware pruning method (Yang et al., CVPR 2017), which reduced the energy consumption of AlexNet and GoogLeNet by 3.7x and 1.6x, respectively, with less than 1% top-5 accuracy loss. This website provides a simplified version of the energy estimation tool for shorter runtime (around 10 seconds).

### Input

To support the variety of toolboxes, this tool takes a single network configuration file. The network configuration file is a txt file, where each line denotes the configuration of a CONV/FC layer. The format of each line is:



- Layer Index: the index of the layer, from 1 to the number of layers. It should be the same as the line number.
- Conf\_IfMap, Conf\_Filt, Conf\_OfMap: the configuration of the input feature maps, the filters and the output feature maps. The configuration of each of the three data types is in the format of "height width number\_of\_channels number\_of\_maps number\_of\_zero\_entries bitwidth\_in\_bits".
- Stride: the stride of this layer. It is in the format of "stride\_y stride\_x".
- Pad: the amount of input padding. It is in the format of "pad\_top pad\_bottom pad\_left pad\_right".

Therefore, there will be 25 entries separated by commas in each line.

### Running the Estimation Model

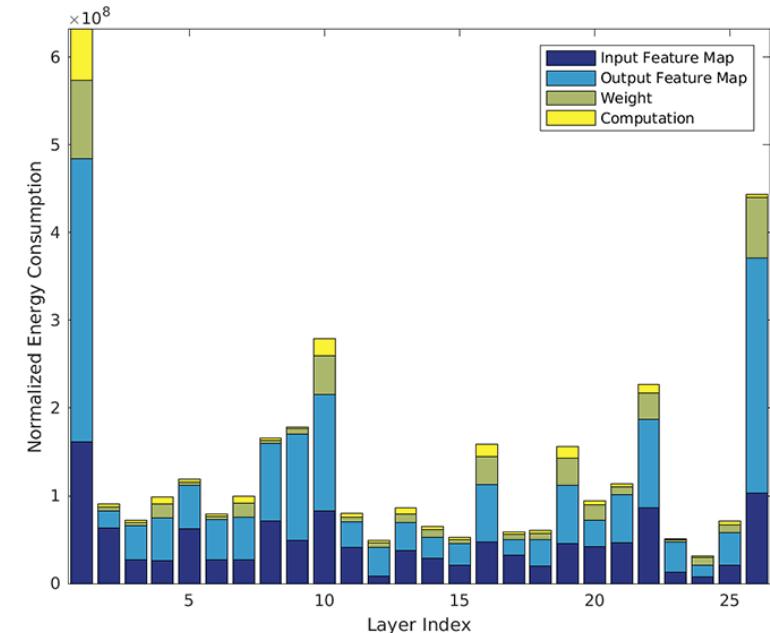
After creating your text file, follow these steps to upload your text file and run the estimation model:

1. Check the "I am not a robot" checkbox and complete the Google reCAPTCHA challenge. Help us prevent spam.
2. Click the "Choose File" button below to choose your text file from your computer.
3. Click the "Run Estimation Model" button below to upload your text file and run the estimation model.

## Input DNN Configuration File

```
Layer_Index,Input_Feature_Map,Output_Feature_Map,Weight,Computation
1,161226686.785535,323273662,88858340.625,58290651
2,63540403.7543396,19104256.6840292,4770357.50868125,3263307.50868125
3,26787638.0555562,39583335.5555542,3272222.7777708,2285942.7777708
4,26018817.2746958,48841502.8019458,15927826.1926396,7847418.06763958
5,62285050.8236438,49433953.294575,4188476.6472875,3227376.6472875
6,27267689.7685187,45381785.7407417,3740581.20378417,2666586.20370417
7,26787131.0480146,48586492.3413917,16216779.2956958,8136371.17069583
```

## Output DNN energy breakdown across layers



# Compression of Weights & Activations

- Compress weights and activations between DRAM and accelerator
- Variable Length / Huffman Coding

Example:

Value: **16'b0** → Compressed Code: **{1'b0}**

Value: **16'bx** → Compressed Code: **{1'b1, 16'bx}**

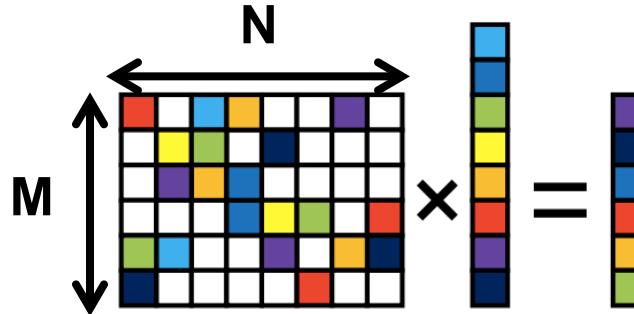
- Tested on AlexNet → 2× overall BW Reduction

Layer	Filter / Image bits (0%)	Filter / Image BW Reduc.	IO / HuffIO (MB/frame)	Voltage (V)	MMACs/Frame	Power (mW)	Real (TOPS/W)
General CNN	16 (0%) / 16 (0%)	1.0x		1.1	—	<b>288</b>	<b>0.3</b>
AlexNet I1	7 (21%) / 4 (29%)	1.17x / 1.3x	<b>1 / 0.77</b>	0.85	105	85	0.96
AlexNet I2	7 (19%) / 7 (89%)	1.15x / <b>5.8x</b>	<b>3.2 / 1.1</b>	0.9	224	55	1.4
AlexNet I3	8 (11%) / 9 (82%)	1.05x / 4.1x	<b>6.5 / 2.8</b>	0.92	150	77	0.7
AlexNet I4	9 (04%) / 8 (72%)	1.00x / 2.9x	<b>5.4 / 3.2</b>	0.92	112	95	0.56
AlexNet I5	9 (04%) / 8 (72%)	1.00x / 2.9x	<b>3.7 / 2.1</b>	0.92	75	95	0.56
Total / avg.	—	—	<b>19.8 / 10</b>	—	—	<b>76</b>	<b>0.94</b>
LeNet-5 I1	3 (35%) / 1 (87%)	1.40x / 5.2x	<b>0.003 / 0.001</b>	0.7	0.3	25	1.07
LeNet-5 I2	4 (26%) / 6 (55%)	1.25x / 1.9x	<b>0.050 / 0.042</b>	0.8	1.6	35	1.75
Total / avg.	—	—	<b>0.053 / 0.043</b>	—	—	<b>33</b>	<b>1.6</b>

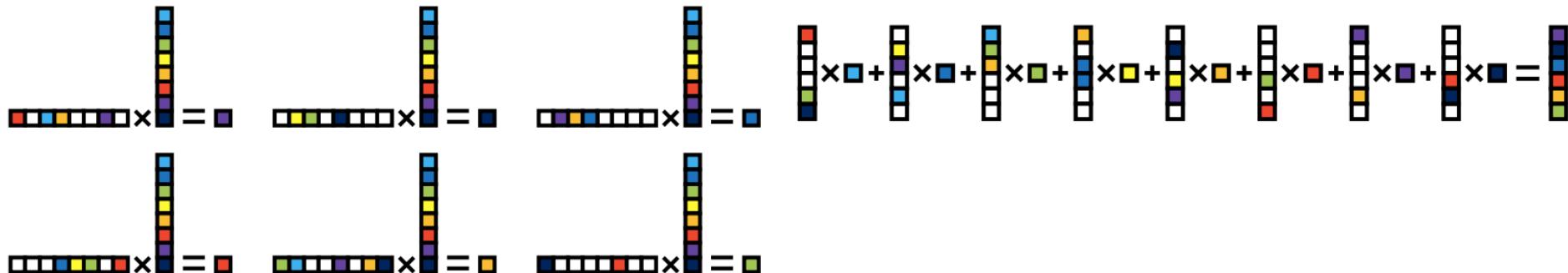
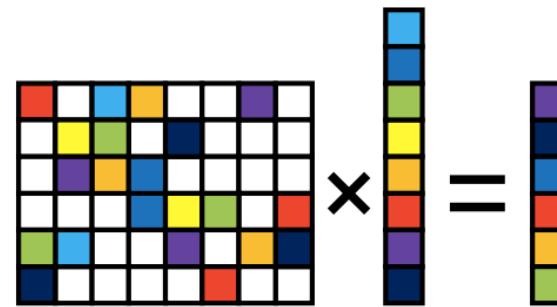
# Sparse Matrix-Vector DSP

- Use CSC rather than CSR for SpMxV

Compressed Sparse Row (CSR)



Compressed Sparse Column (CSC)

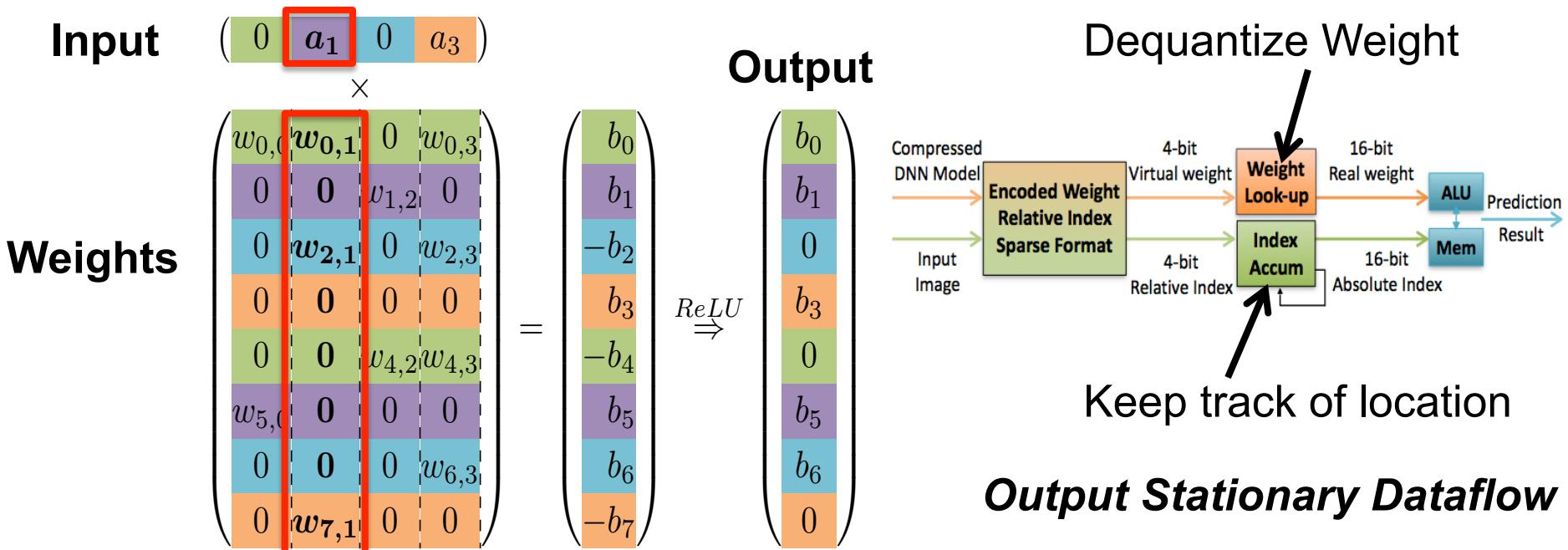


Reduce memory bandwidth (when not  $M \gg N$ )  
For DNN,  $M = \# \text{ of filters}$ ,  $N = \# \text{ of weights per filter}$

# EIE: A Sparse Linear Algebra Engine

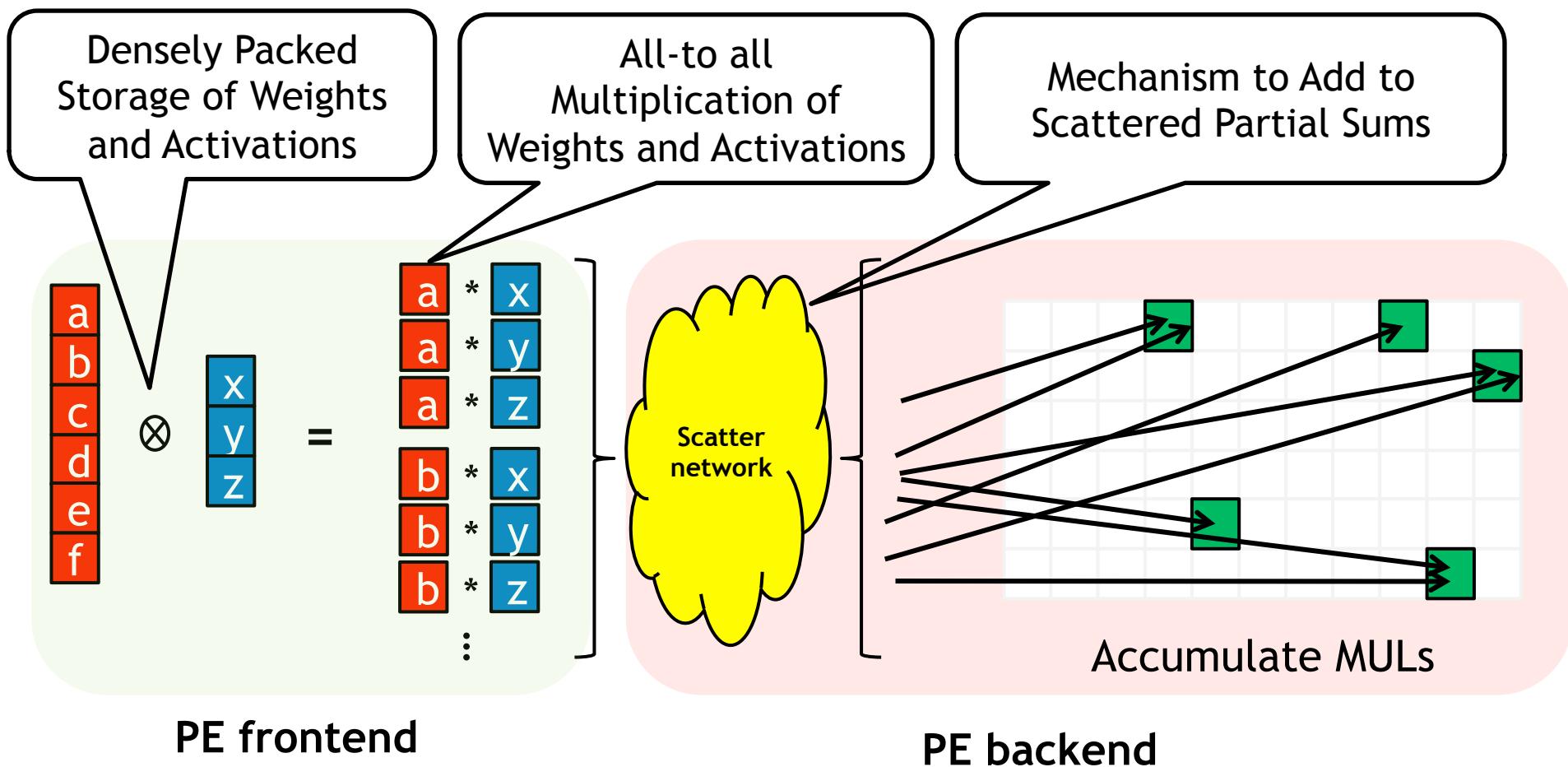
- Process Fully Connected Layers (after Deep Compression)
- Store weights column-wise in Run Length format
- Read relative column when input is non-zero

*Supports Fully Connected Layers Only*



# Sparse CNN (SCNN)

*Supports Convolutional Layers*



*Input Stationary Dataflow*

[Parashar et al., ISCA 2017]

# Structured/Coarse-Grained Pruning

- **Scalpel**

- Prune to match the underlying data-parallel hardware organization for speed up

*Example: 2-way SIMD*

0	5	2	5	0	0
0	0	1	7	0	0
2	3	0	0	4	2
8	4	0	0	0	0
0	0	1	1	8	3
3	2	0	0	0	0

Dense weights

0	5	2	5		
		1	7		
2	3	0	0	4	2
8	4	0	0	0	0
0	0	1	1	8	3
3	2	0	0	0	0

Sparse weights

$$\begin{aligned} A' &= [(0, 5) (2, 5) (1, 7) \\ &\quad (2, 3) (4, 2) (8, 4) \\ &\quad (8, 3) (3, 2)] \\ JA' &= [0, 2, 2, 0, 4, 0, \\ &\quad 4, 0] \\ IA' &= [0, 4, 6, 10, 12, \\ &\quad 14, 16] \end{aligned}$$

X

Input Vector

# Compact Network Architectures

---

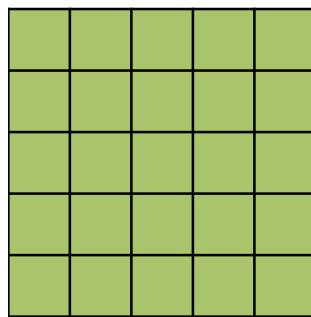
- **Break large convolutional layers into a series of smaller convolutional layers**
  - Fewer weights, but same effective receptive field
- **Before Training: Network Architecture Design**
- **After Training: Decompose Trained Filters**

# Network Architecture Design

Build Network with series of Small Filters

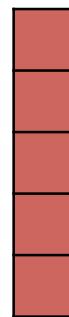
## GoogleNet/Inception v3

5x5 filter



decompose  
→

5x1 filter

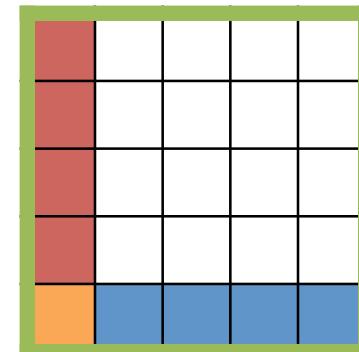


1x5 filter



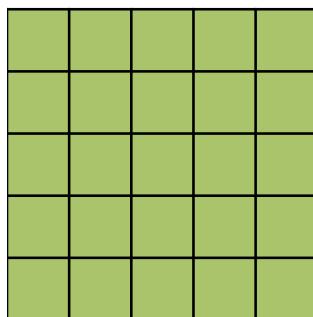
*separable  
filters*

Apply sequentially



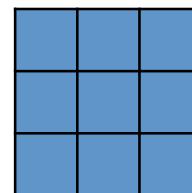
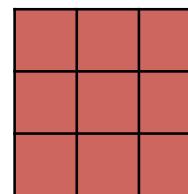
## VGG-16

5x5 filter

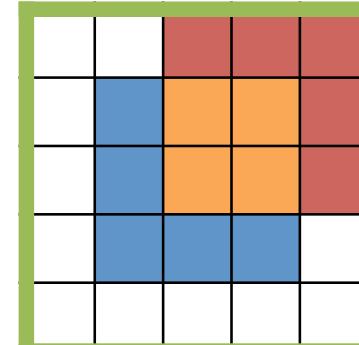


decompose  
→

Two 3x3 filters



Apply sequentially



# Network Architecture Design

Reduce size and computation with 1x1 Filter (**bottleneck**)

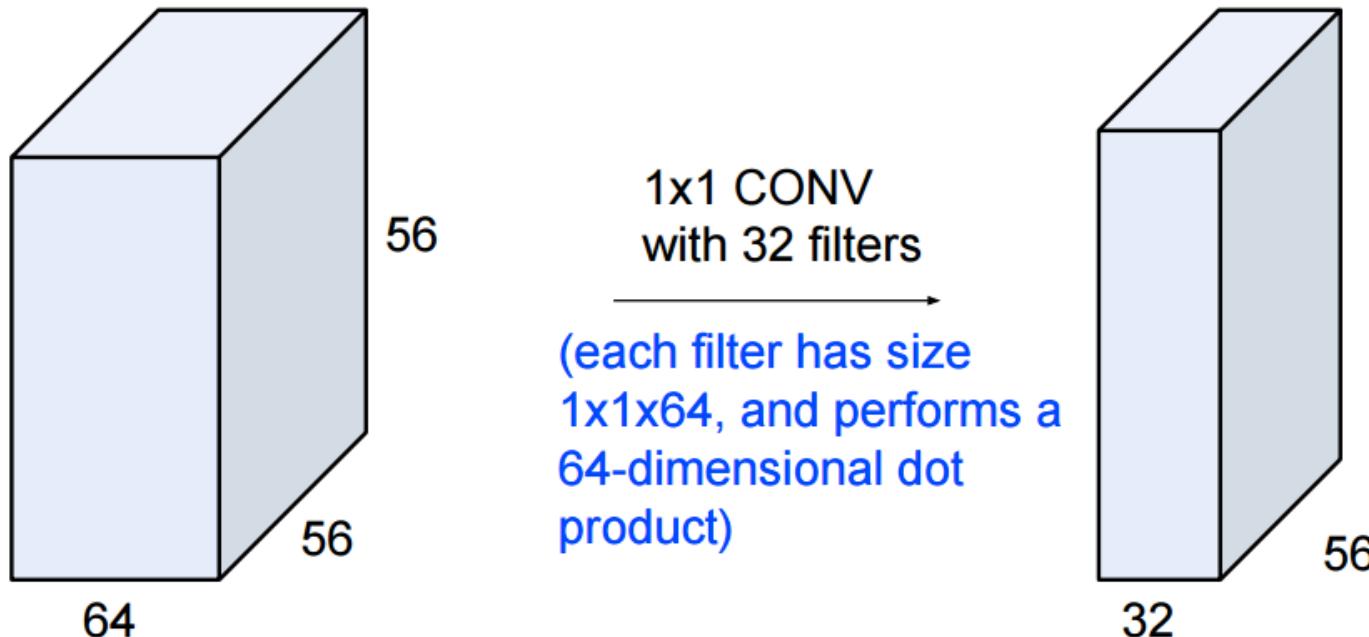


Figure Source:  
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

# Network Architecture Design

Reduce size and computation with 1x1 Filter (**bottleneck**)

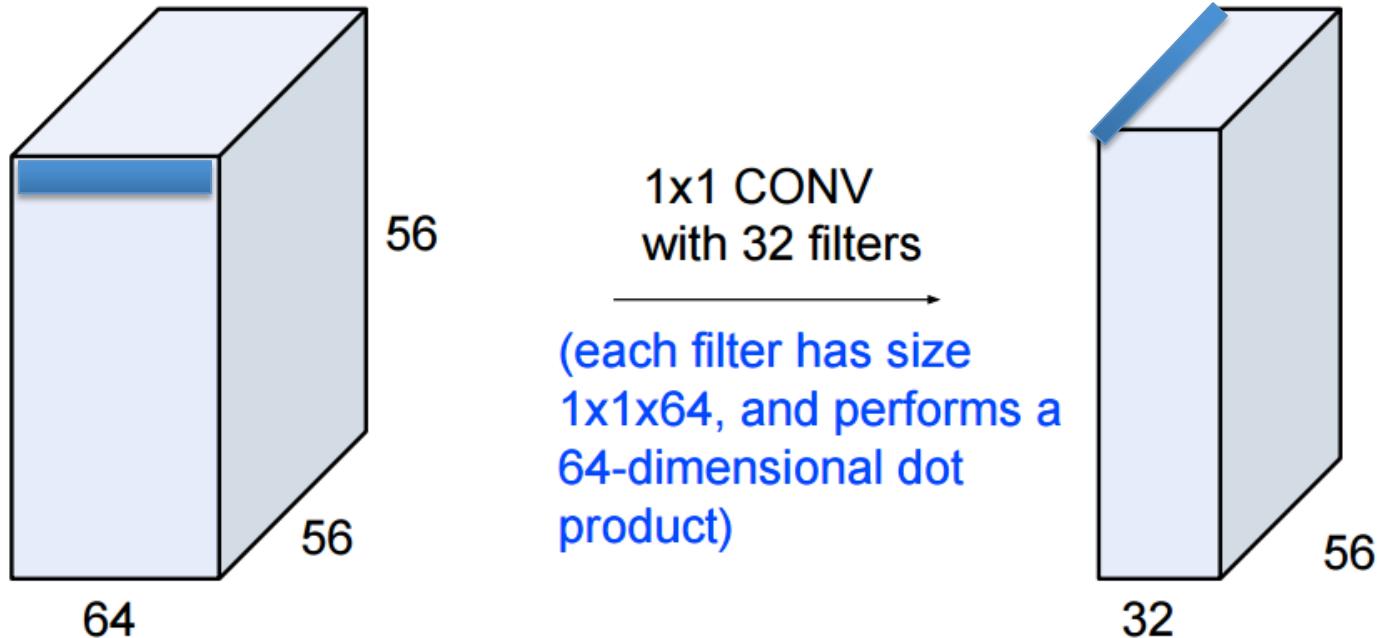


Figure Source:  
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

# Network Architecture Design

Reduce size and computation with 1x1 Filter (**bottleneck**)

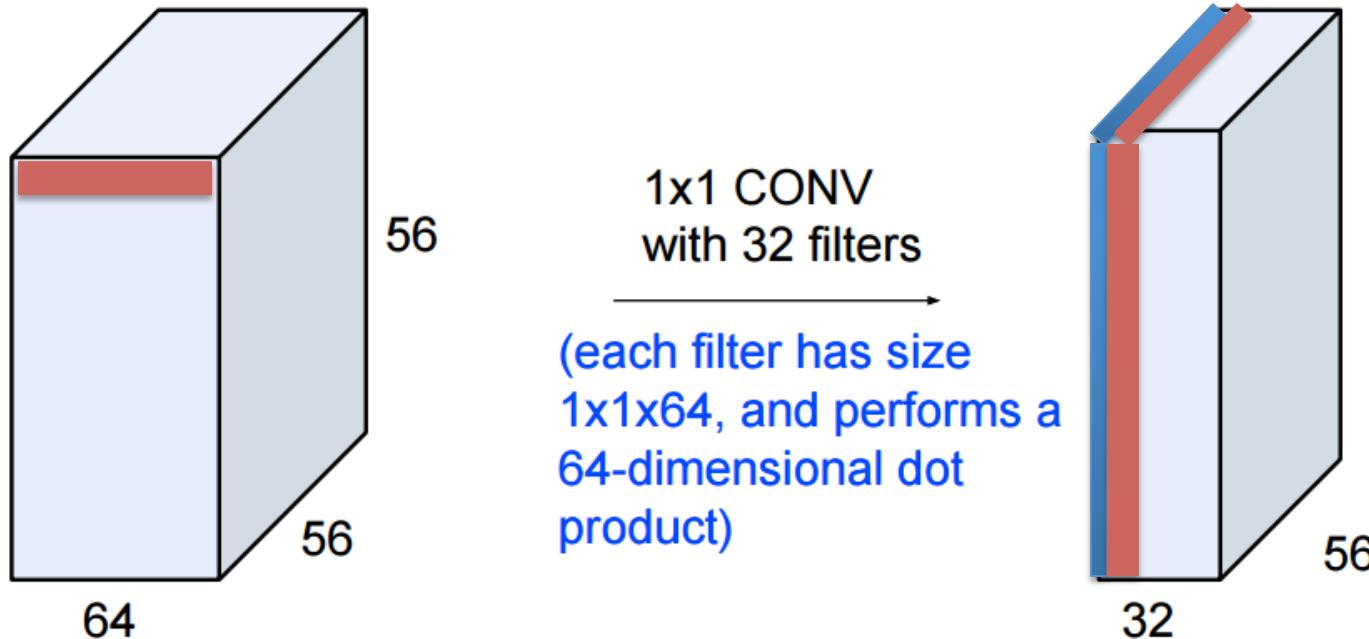


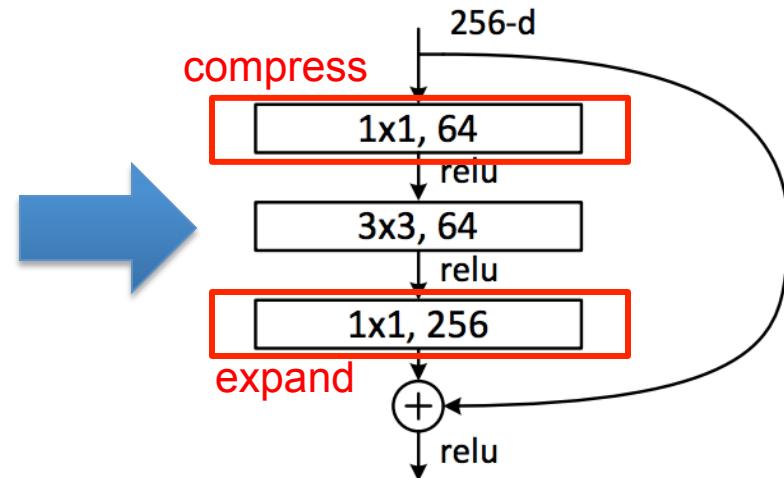
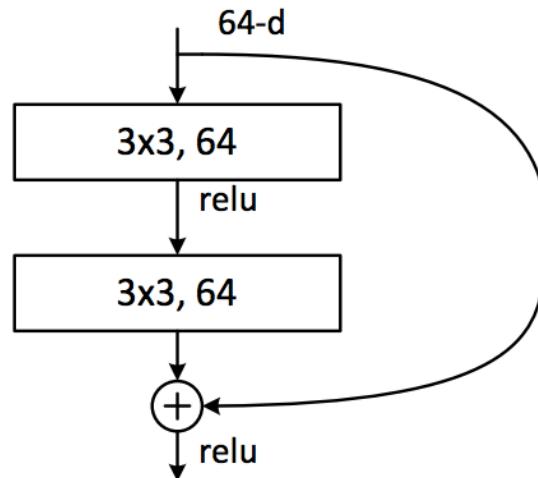
Figure Source:  
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

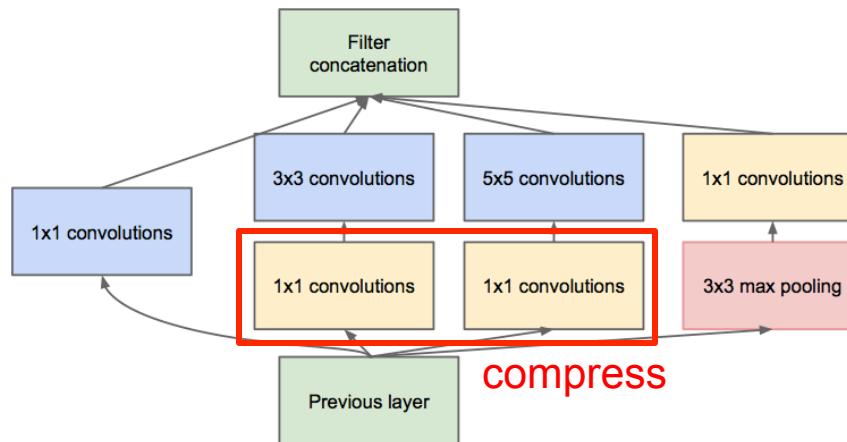
[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

# Bottleneck in Popular DNN models

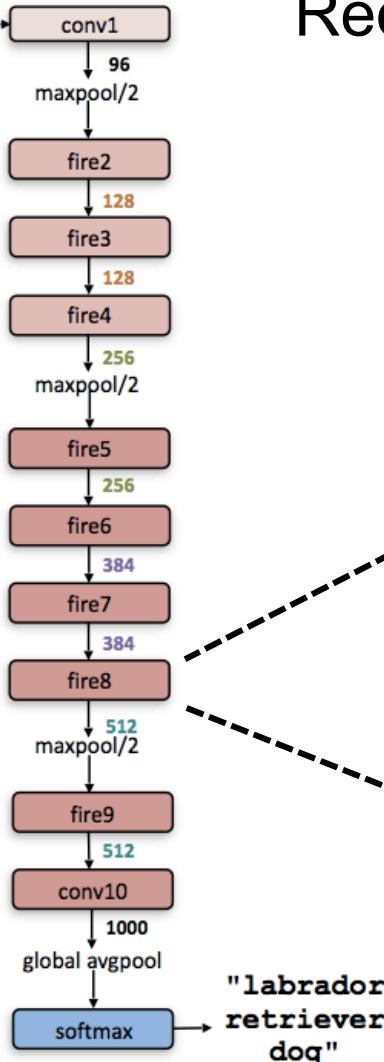
ResNet



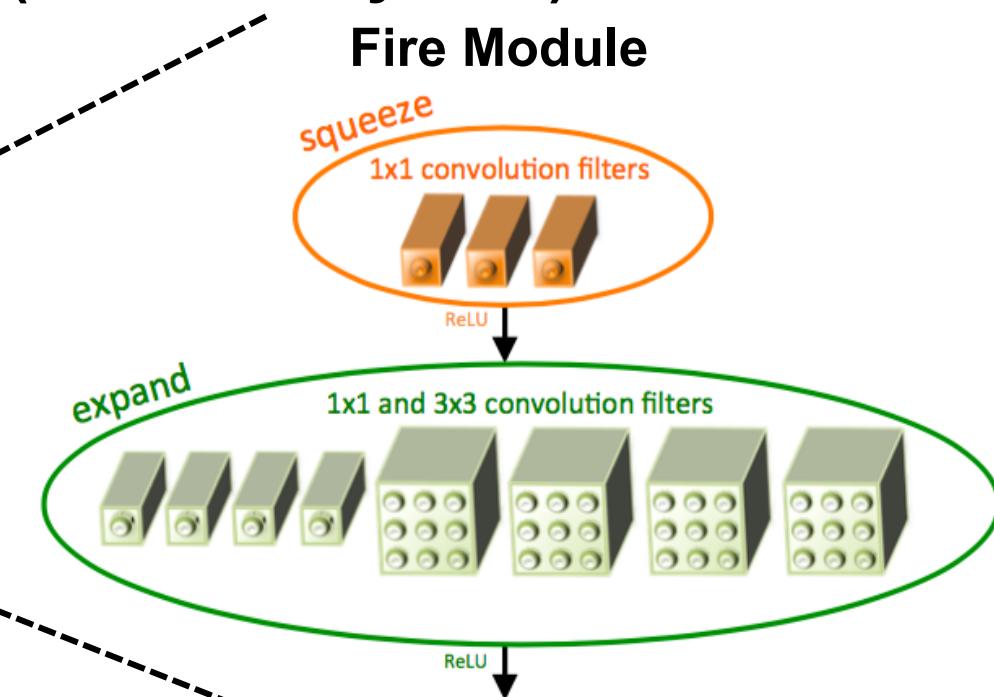
GoogleNet



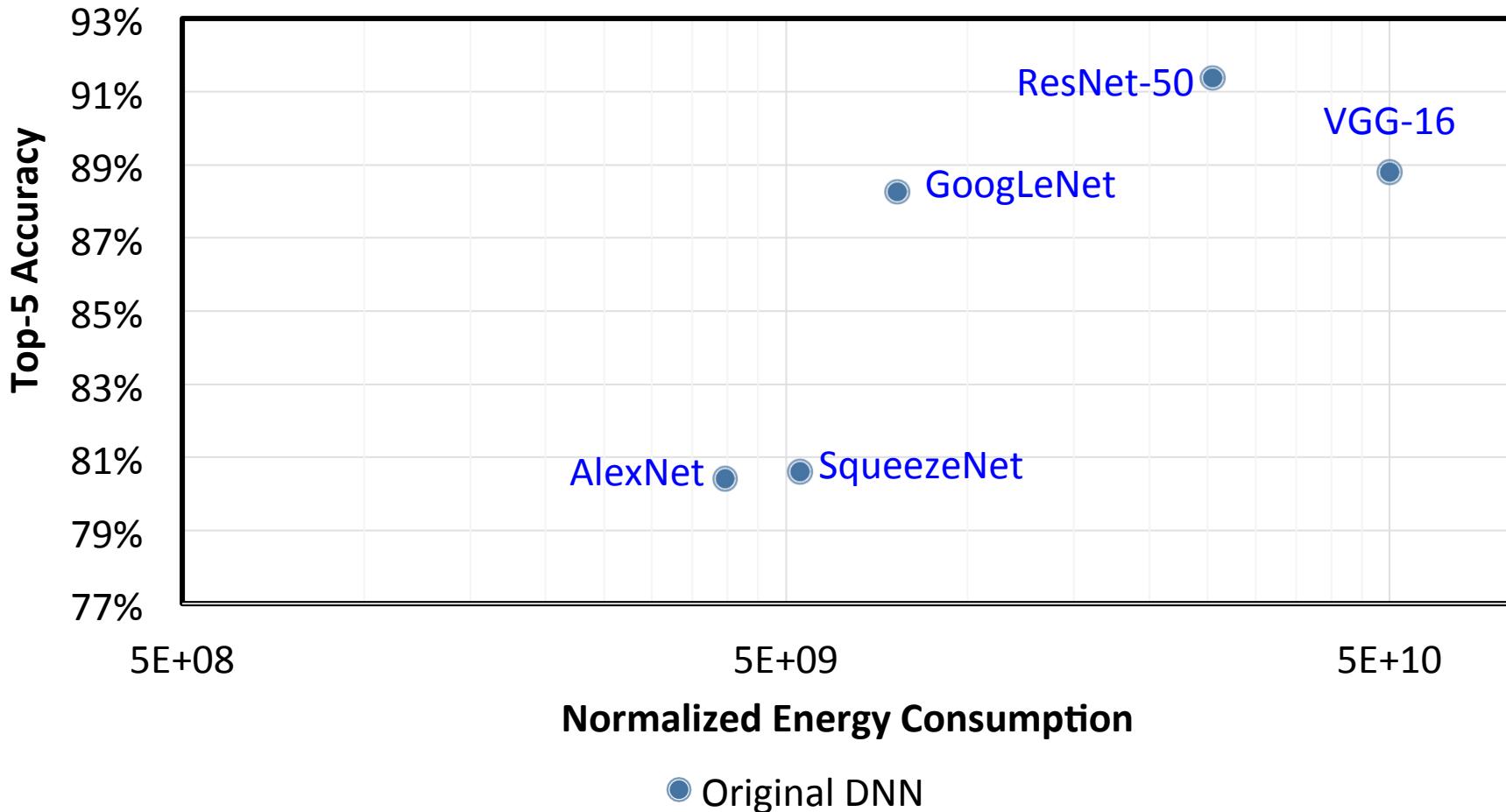
# SqueezeNet



Reduce weights by reducing number of input channels by “squeezing” with 1x1  
**50x fewer weights than AlexNet (no accuracy loss)**



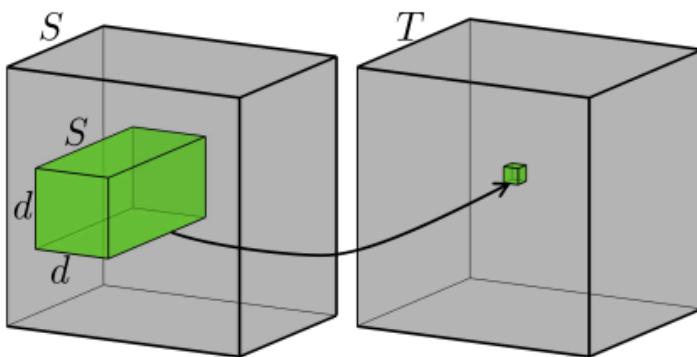
# Energy Consumption of Existing DNNs



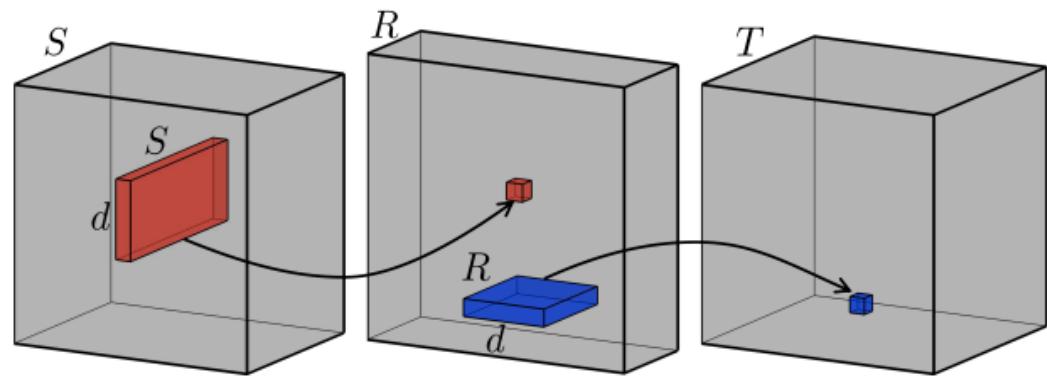
Deeper CNNs with fewer weights do not necessarily consume less energy than shallower CNNs with more weights

# Decompose Trained Filters

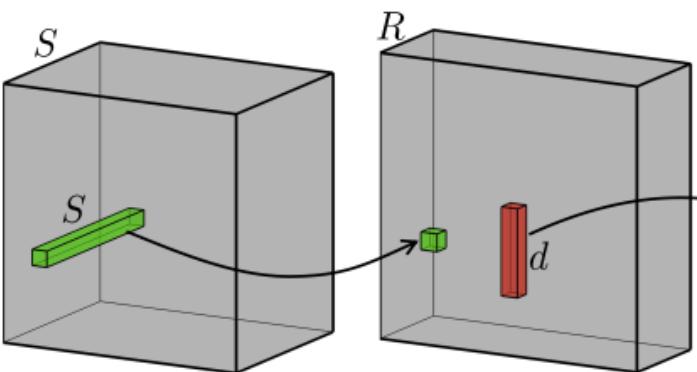
After training, perform **low-rank approximation** by applying **tensor decomposition** to weight kernel; then **fine-tune** weights for accuracy



(a) Full convolution



(b) Two-component decomposition (Jaderberg et al., 2014a)

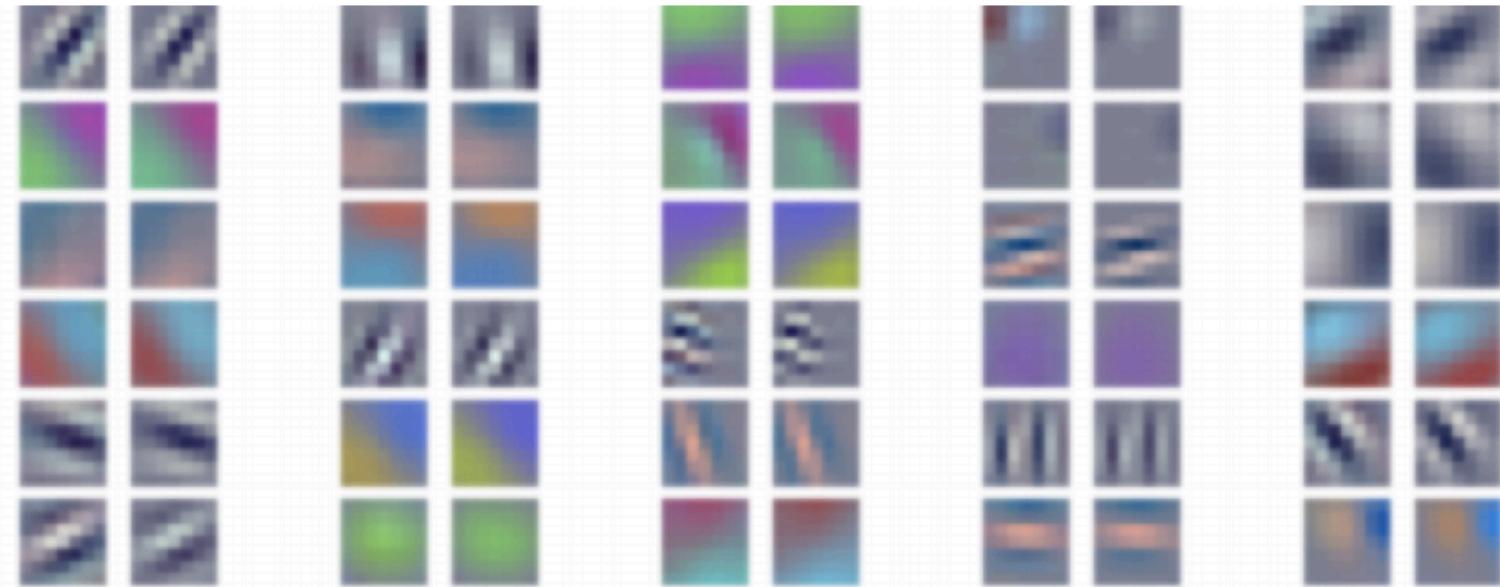


(c) CP-decomposition

# Decompose Trained Filters

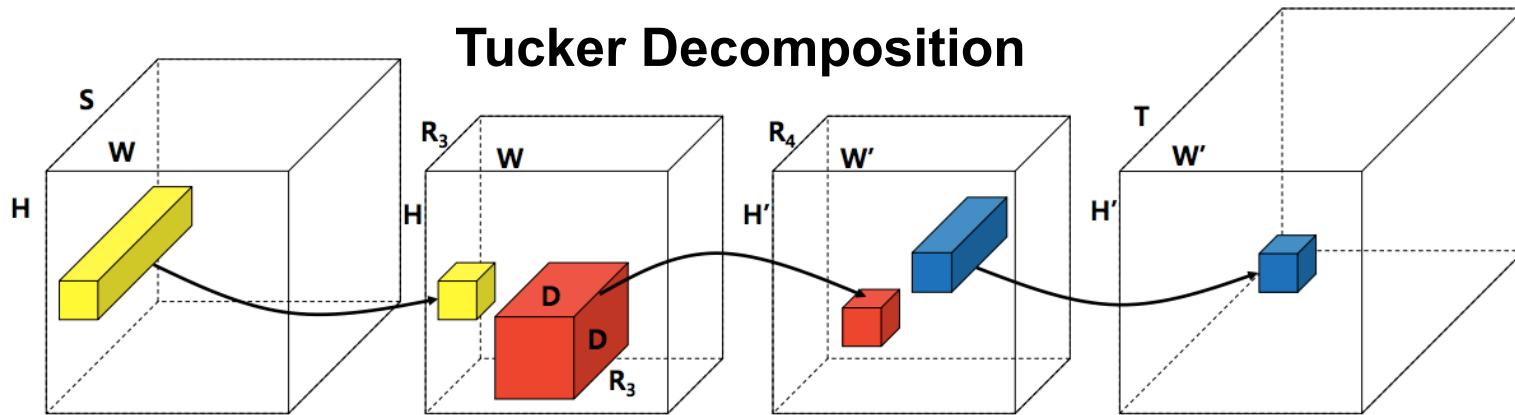
## Visualization of Filters

Original   Approx.



- **Speed up by 1.6 – 2.7x on CPU/GPU for CONV1, CONV2 layers**
- **Reduce size by 5 - 13x for FC layer**
- **< 1% drop in accuracy**

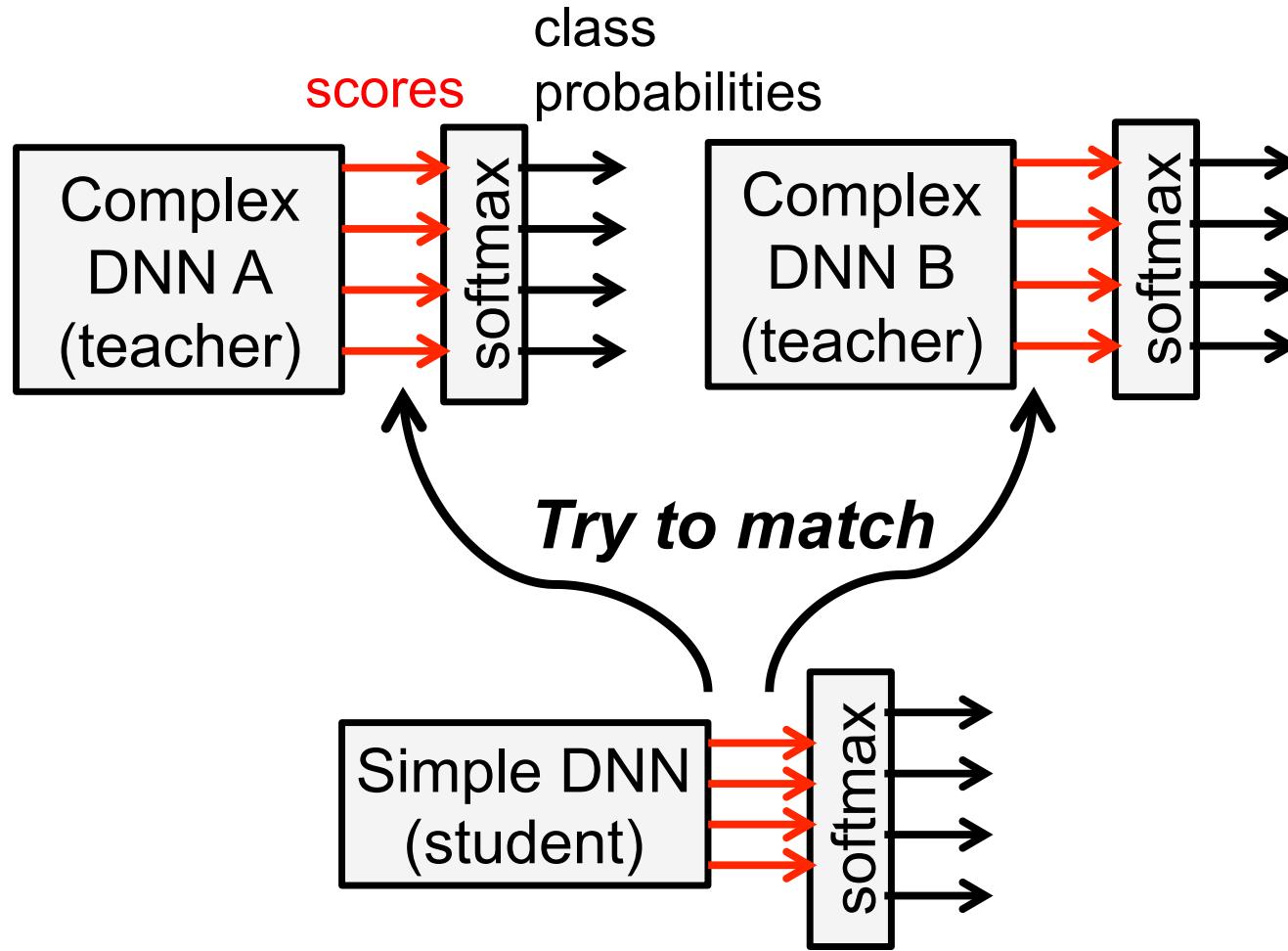
# Decompose Trained Filters on Phone



Model	Top-5	Weights	FLOPs	S6	Titan X
<i>AlexNet</i>	80.03	61M	725M	117ms	245mJ
<i>AlexNet*</i>	78.33	11M	272M	43ms	72mJ
(imp.)	(-1.70)	( $\times 5.46$ )	( $\times 2.67$ )	( $\times 2.72$ )	( $\times 3.41$ )
<i>VGG-S</i>	84.60	103M	2640M	357ms	825mJ
<i>VGG-S*</i>	84.05	14M	549M	97ms	193mJ
(imp.)	(-0.55)	( $\times 7.40$ )	( $\times 4.80$ )	( $\times 3.68$ )	( $\times 4.26$ )
<i>GoogLeNet</i>	88.90	6.9M	1566M	273ms	473mJ
<i>GoogLeNet*</i>	88.66	4.7M	760M	192ms	296mJ
(imp.)	(-0.24)	( $\times 1.28$ )	( $\times 2.06$ )	( $\times 1.42$ )	( $\times 1.60$ )
<i>VGG-16</i>	89.90	138M	15484M	1926ms	4757mJ
<i>VGG-16*</i>	89.40	127M	3139M	576ms	1346mJ
(imp.)	(-0.50)	( $\times 1.09$ )	( $\times 4.93$ )	( $\times 3.34$ )	( $\times 3.53$ )

# Knowledge Distillation

---



# **Benchmarking Metrics for DNN Hardware**

**ISCA Tutorial (2017)**

Website: <http://eyeriss.mit.edu/tutorial.html>

# Metrics Overview

---

- **How can we compare designs?**
- **Target Metrics**
  - Accuracy
  - Power
  - Throughput
  - Cost
- **Additional Factors**
  - External memory bandwidth
  - Required on-chip storage
  - Utilization of cores

# Download Benchmarking Data

---

- **Input (<http://image-net.org/>)**
  - Sample subset from ImageNet Validation Dataset
- **Widely accepted state-of-the-art DNNs  
(Model Zoo: <http://caffe.berkeleyvision.org/>)**
  - AlexNet
  - VGG-16
  - GoogleNet-v1
  - ResNet-50

# Metrics for DNN Algorithm

---

- **Accuracy**
- **Network Architecture**
  - # Layers, filter size, # of filters, # of channels
- **# of Weights (storage capacity)**
  - Number of non-zero (NZ) weights
- **# of MACs (operations)**
  - Number of non-zero (NZ) MACS

# Metrics of DNN Algorithms

Metrics	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Accuracy (top-5 error)*	19.8	8.80	10.7	7.02
Input	227x227	224x224	224x224	224x224
<b># of CONV Layers</b>	<b>5</b>	<b>16</b>	<b>21</b>	<b>49</b>
Filter Sizes	3, 5, 11	3	1, 3, 5, 7	1, 3, 7
# of Channels	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1, 4	1	1, 2	1, 2
# of Weights	2.3M	14.7M	6.0M	23.5M
# of MACs	666M	15.3G	1.43G	3.86G
<b># of FC layers</b>	<b>3</b>	<b>3</b>	<b>1</b>	<b>1</b>
# of Weights	58.6M	124M	1M	2M
# of MACs	58.6M	124M	1M	2M
<b>Total Weights</b>	<b>61M</b>	<b>138M</b>	<b>7M</b>	<b>25.5M</b>
<b>Total MACs</b>	<b>724M</b>	<b>15.5G</b>	<b>1.43G</b>	<b>3.9G</b>

\*Single crop results: <https://github.com/jcjohnson/cnn-benchmarks>

# Metrics of DNN Algorithms

Metrics	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Accuracy (top-5 error)*	19.8	8.80	10.7	7.02
<b># of CONV Layers</b>	<b>5</b>	<b>16</b>	<b>21</b>	<b>49</b>
# of Weights	2.3M	14.7M	6.0M	23.5M
# of MACs	666M	15.3G	1.43G	3.86G
<b># of NZ MACs**</b>	<b>394M</b>	<b>7.3G</b>	<b>806M</b>	<b>1.5G</b>
<b># of FC layers</b>	<b>3</b>	<b>3</b>	<b>1</b>	<b>1</b>
# of Weights	58.6M	124M	1M	2M
# of MACs	58.6M	124M	1M	2M
<b># of NZ MACs**</b>	<b>14.4M</b>	<b>17.7M</b>	<b>639k</b>	<b>1.8M</b>
<b>Total Weights</b>	<b>61M</b>	<b>138M</b>	<b>7M</b>	<b>25.5M</b>
<b>Total MACs</b>	<b>724M</b>	<b>15.5G</b>	<b>1.43G</b>	<b>3.9G</b>
<b># of NZ MACs**</b>	<b>409M</b>	<b>7.3G</b>	<b>806M</b>	<b>1.5G</b>

\*Single crop results: <https://github.com/jcjohnson/cnn-benchmarks>

# Metrics of DNN Algorithms

Metrics	AlexNet	AlexNet (sparse)
Accuracy (top-5 error)	19.8	19.8
<b># of Conv Layers</b>	<b>5</b>	<b>5</b>
# of Weights	2.3M	2.3M
# of MACs	666M	666M
# of NZ weights	<b>2.3M</b>	<b>863k</b>
# of NZ MACs	<b>394M</b>	<b>207M</b>
<b># of FC layers</b>	<b>3</b>	<b>3</b>
# of Weights	58.6M	58.6M
# of MACs	58.6M	58.6M
# of NZ weights	<b>58.6M</b>	<b>5.9M</b>
# of NZ MACs	<b>14.4M</b>	<b>2.1M</b>
<b>Total Weights</b>	<b>61M</b>	<b>61M</b>
<b>Total MACs</b>	<b>724M</b>	<b>724M</b>
<b># of NZ weights</b>	<b>61M</b>	<b>6.8M</b>
<b># of NZ MACs</b>	<b>409M</b>	<b>209M</b>

# Metrics for DNN Hardware

---

- **Measure energy and DRAM access relative to number of non-zero MACs and bit-width of MACs**
  - Account for impact of sparsity in weights and activations
  - Normalize DRAM access based on operand size
- **Energy Efficiency of Design**
  - pJ/(non-zero weight & activation)
- **External Memory Bandwidth**
  - DRAM operand access/(non-zero weight & activation)
- **Area Efficiency**
  - Total chip mm<sup>2</sup>/multi (also include process technology)
  - Accounts for on-chip memory

# ASIC Benchmark (e.g. Eyeriss)

---

ASIC Specs	
Process Technology	65nm LP TSMC (1.0V)
Clock Frequency (MHz)	200
Number of Multipliers	168
Total core area (mm <sup>2</sup> ) /total # of multiplier	0.073
Total on-Chip memory (kB) / total # of multiplier	1.14
Measured or Simulated	Measured
If Simulated, Syn or PnR? Which corner?	n/a

# ASIC Benchmark (e.g. Eyeriss)

## Layer by layer breakdown for AlexNet CONV layers

Metric	Units	L1	L2	L3	L4	L5	Overall*
Batch Size	#				4		
Bit/Operand	#				16		
Energy/ non-zero MACs (weight & act)	pJ/MAC	16.5	18.2	29.5	41.6	32.3	21.7
DRAM access/ non-zero MACs	Operands/ MAC	0.006	0.003	0.007	0.010	0.008	0.005
Runtime	ms	20.9	41.9	23.6	18.4	10.5	115.3
Power	mW	332	288	266	235	236	278

# Website to Summarize Results

---

- <http://eyeriss.mit.edu/benchmarking.html>
- Send results or feedback to: [eyeriss@mit.edu](mailto:eyeriss@mit.edu)

ASIC Specs	Input
Process Technology	65nm LP TSMC (1.0V)
Clock Frequency (MHz)	200
Number of Multipliers	168
Core area (mm <sup>2</sup> ) / multiplier	0.073
On-Chip memory (kB) / multiplier	1.14
Measured or Simulated	Measured
If Simulated, Syn or PnR? Which corner?	n/a

Metric	Units	Input
Name of CNN	Text	AlexNet
# of Images Tested	#	100
Bits per operand	#	16
Batch Size	#	4
# of Non Zero MACs	#	409M
Runtime	ms	115.3
<b>Utilization vs. Peak</b>	%	<b>41</b>
Power	mW	278
<b>Energy/non-zero MACs</b>	<b>pJ/MAC</b>	<b>21.7</b>
<b>DRAM access/non- zero MACs</b>	<b>operands /MAC</b>	<b>0.005</b>

# Implementation-Specific Metrics

---

Different devices may have implementation-specific metrics

Example: FPGAs

Metric	Units	AlexNet
Device	Text	Xilinx Virtex-7 XC7V690T
Utilization	DSP	#
	BRAM	#
	LUT	#
	FF	#
Performance Density	GOPs/slice	8.12E-04

# Tutorial Summary

---

- **DNNs are a critical component in the AI revolution**, delivering record breaking accuracy on many important AI tasks for a wide range of applications; however, it comes at the cost of **high computational complexity**
- **Efficient processing of DNNs** is an important area of research with many promising opportunities for innovation at **various levels of hardware design, including algorithm co-design**
- When considering different DNN solutions it is important to **evaluate with the appropriate workload** in term of both input and model, and recognize that they are **evolving rapidly**.
- It's important to consider a **comprehensive set of metrics** when evaluating different DNN solutions: **accuracy, speed, energy, and cost**

# Resources

---

- **Eyeriss Project:** <http://eyeriss.mit.edu>
  - Tutorial Slides
  - Benchmarking
  - Energy modeling
  - Mailing List for updates
    - <http://mailman.mit.edu/mailman/listinfo/eems-news>
  - **Paper based on today's tutorial:**
    - V. Sze, Y.-H. Chen, T.-J. Yang, J. Emer, “*Efficient Processing of Deep Neural Networks: A Tutorial and Survey*”, arXiv, 2017



Follow @eems\_mit

# References

## ISCA Tutorial (2017)

Website: <http://eyeriss.mit.edu/tutorial.html>

# References (Alphabetical by Author)

---

- Albericio, Jorge, et al. "Cnvlutin: ineffectual-neuron-free deep neural network computing," ISCA, 2016.
- Alwani, Manoj, et al., "Fused Layer CNN Accelerators," MICRO, 2016
- Chakradhar, Srimat, et al., "A dynamically configurable coprocessor for convolutional neural networks," ISCA, 2010
- Chen, Tianshi, et al., "DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning," ASPLOS, 2014
- Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," ISSCC, 2016.
- Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," ISCA, 2016.
- Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer," MICRO, 2014.
- Chi, Ping, et al. "PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory," ISCA 2016.
- Cong, Jason, and Bingjun Xiao. "Minimizing computation in convolutional neural networks." International Conference on Artificial Neural Networks. Springer International Publishing, 2014.
- Courbariaux, Matthieu, and Yoshua Bengio. "Binarynet: Training deep neural networks with weights and activations constrained to + 1 or -1." arXiv preprint arXiv:1602.02830 (2016).
- Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations," NIPS, 2015.

# References (Alphabetical by Author)

---

- Dean, Jeffrey, et al., "Large Scale Distributed Deep Networks," NIPS, 2012
- Denton, Emily L., et al. "Exploiting linear structure within convolutional networks for efficient evaluation," NIPS, 2014.
- Dorrance, Richard, Fengbo Ren, and Dejan Marković. "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs." Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays. ACM, 2014.
- Du, Zidong, et al., "ShiDianNao: shifting vision processing closer to the sensor," ISCA, 2015
- Eryilmaz, Sukru Burc, et al. "Neuromorphic architectures with electronic synapses." ISQED, 2016.
- Esser, Steven K., et al., "Convolutional networks for fast, energy-efficient neuromorphic computing," PNAS 2016
- Farabet, Clement, et al., "An FPGA-Based Stream Processor for Embedded Real-Time Vision with Convolutional Networks," ICCV 2009
- Gokhale, Vinatak, et al., "A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks," CVPR Workshop, 2014
- Govoreanu, B., et al. "10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation," IEDM, 2011.
- Gupta, Suyog, et al., "Deep Learning with Limited Numerical Precision," ICML, 2015
- Gysel, Philipp, Mohammad Motamedi, and Soheil Ghiasi. "Hardware-oriented Approximation of Convolutional Neural Networks." arXiv preprint arXiv:1604.03168 (2016).
- Han, Song, et al. "EIE: efficient inference engine on compressed deep neural network," ISCA, 2016.

# References (Alphabetical by Author)

---

- Han, Song, et al. "Learning both weights and connections for efficient neural network," NIPS, 2015.
- Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," ICLR, 2016.
- He, Kaiming, et al. "Deep residual learning for image recognition," CVPR, 2016.
- Horowitz, Mark. "Computing's energy problem (and what we can do about it)," ISSCC, 2014.
- Iandola, Forrest N., et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 1MB model size," ICLR, 2017.
- Ioffe, Sergey, and Szegedy, Christian, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," ICML, 2015
- Jermyn, Michael, et al., "Neural networks improve brain cancer detection with Raman spectroscopy in the presence of operating room light artifacts," Journal of Biomedical Optics, 2016
- Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit," ISCA, 2017.
- Judd, Patrick, et al. "Reduced-precision strategies for bounded memory in deep neural nets." arXiv preprint arXiv:1511.05236 (2015).
- Judd, Patrick, Jorge Albericio, and Andreas Moshovos. "Stripes: Bit-serial deep neural network computing." IEEE Computer Architecture Letters (2016).
- Kim, Duckhwan, et al. "Neurocube: a programmable digital neuromorphic architecture with high-density 3D memory." ISCA 2016
- Kim, Yong-Deok, et al. "Compression of deep convolutional neural networks for fast and low power mobile applications." ICLR 2016

# References (Alphabetical by Author)

---

- Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks," NIPS, 2012.
- Lavin, Andrew, and Gray, Scott, "Fast Algorithms for Convolutional Neural Networks," arXiv preprint arXiv: 1509.09308 (2015)
- LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.
- LeCun, Yann, et al. "Optimal brain damage," NIPS, 1989.
- Lin, Min, Qiang Chen, and Shuicheng Yan. "Network in network." arXiv preprint arXiv:1312.4400 (2013).
- Mathieu, Michael, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through FFTs." arXiv preprint arXiv:1312.5851 (2013).
- Merola, Paul A., et al. "Artificial brains. A million spiking-neuron integrated circuit with a scalable communication network and interface," Science, 2014
- Moons, Bert, and Marian Verhelst. "A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets." Symposium on VLSI Circuits (VLSI-Circuits), 2016.
- Moons, Bert, et al. "Energy-efficient ConvNets through approximate computing," WACV, 2016.
- Parashar, Angshuman, et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks." ISCA, 2017.
- Park, Seongwook, et al., "A 1.93TOPS/W Scalable Deep Learning/Inference Processor with Tetra-Parallel MIMD Architecture for Big-Data Applications," ISSCC, 2015
- Peemen, Maurice, et al., "Memory-centric accelerator design for convolutional neural networks," ICCD, 2013
- Prezioso, Mirko, et al. "Training and operation of an integrated neuromorphic network based on metal-oxide memristors." Nature 521.7550 (2015): 61-64.

# References (Alphabetical by Author)

---

- Rastegari, Mohammad, et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," ECCV, 2016
- Reagen, Brandon, et al. "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," ISCA, 2016.
- Rhu, Minsoo, et al., "vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design," MICRO, 2016
- Russakovsky, Olga, et al. "Imagenet large scale visual recognition challenge." *International Journal of Computer Vision* 115.3 (2015): 211-252.
- Sermanet, Pierre, et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks," CVPR, 2014.
- Shafiee, Ali, et al. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars." Proc. ISCA. 2016.
- Shen, Yichen, et al. "Deep learning with coherent nanophotonic circuits." *Nature Photonics*, 2017.
- Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).
- Szegedy, Christian, et al. "Going deeper with convolutions," CVPR, 2015.
- Vasilache, Nicolas, et al. "Fast convolutional nets with fbfft: A GPU performance evaluation." arXiv preprint arXiv:1412.7580 (2014).
- Yang, Tien-Ju, et al. "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning," CVPR, 2017
- Yu, Jiecao, et al. "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism." /SCA, 2017.
- Zhang, Chen, et al., "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," FPGA, 2015