DNN Model and Hardware Co-Design

ISCA Tutorial (2017)

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

Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang

Approaches

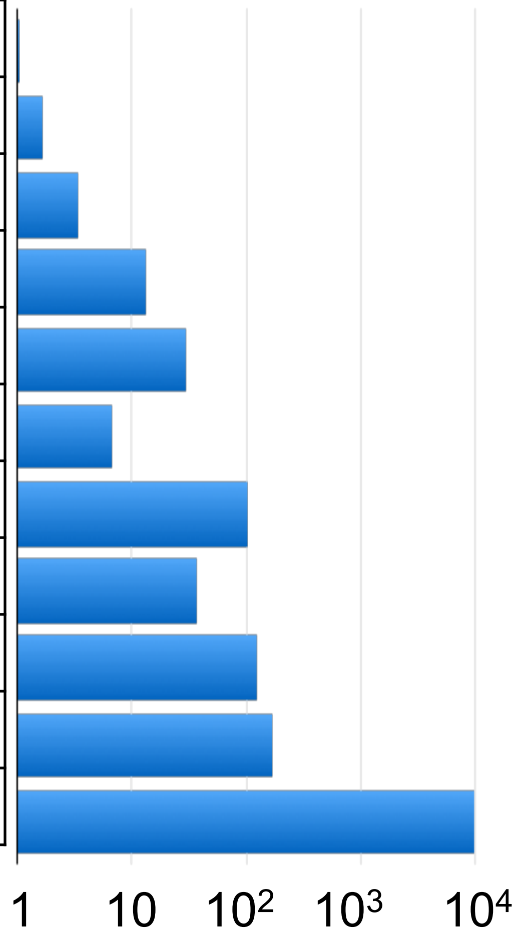
- Reduce size of operands for storage/compute
  - Floating point → Fixed point
  - Bit-width reduction
  - Non-linear quantization

- Reduce number of operations for storage/compute
  - Exploit Activation Statistics (Compression)
  - Network Pruning
  - Compact Network Architectures

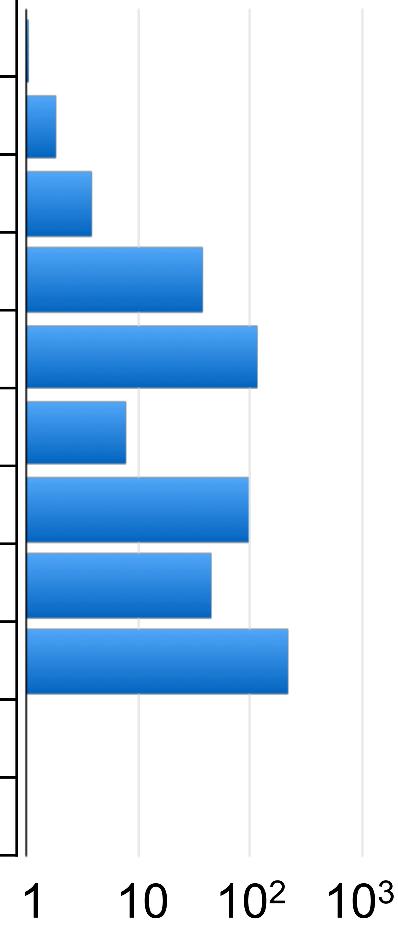
Cost of Operations

Operation:	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32b FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Relative Energy Cost

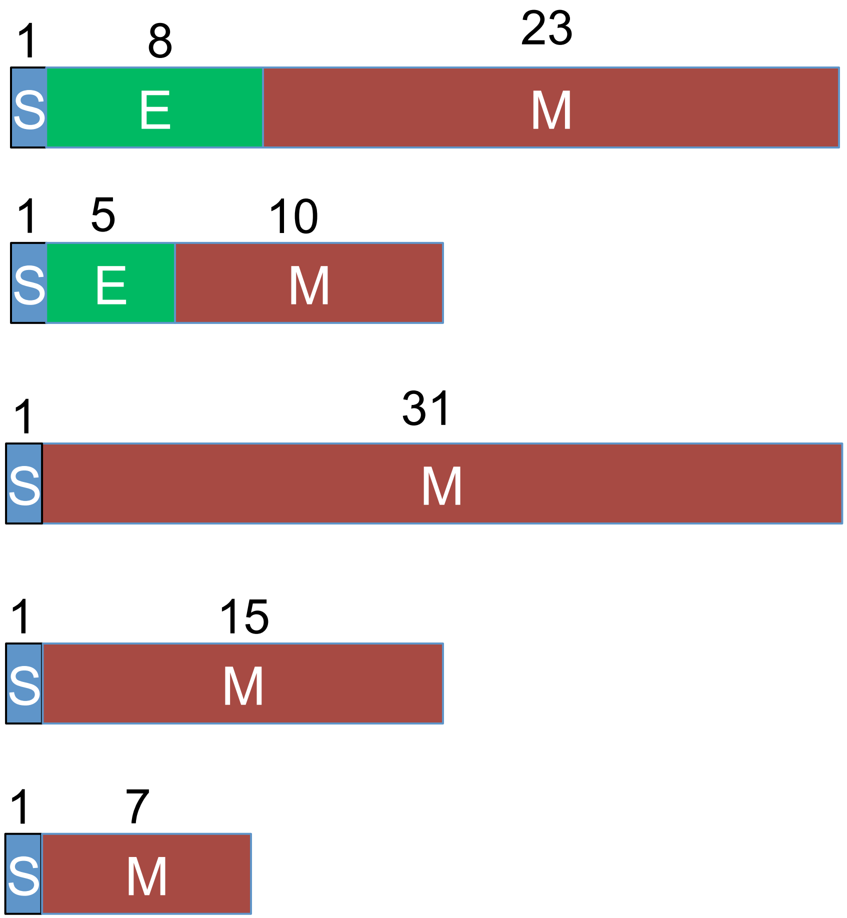


Relative Area Cost



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

Number Representation



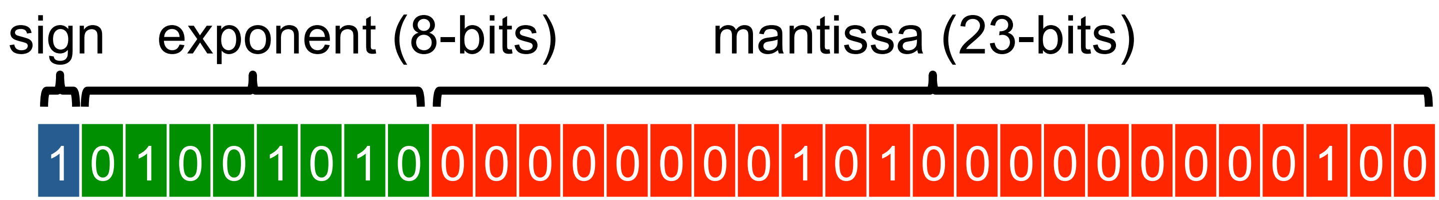
	Range	Accuracy
FP32	$10^{-38} – 10^{38}$	.000006%
FP16	6×10^{-5} - 6×10^{4}	.05%
Int32	$0 – 2 \times 10^{9}$	½
Int16	$0 – 6 \times 10^{4}$	½
Int8	0 – 127	½

Image Source: B. Dally

Floating Point → Fixed Point

Floating Point

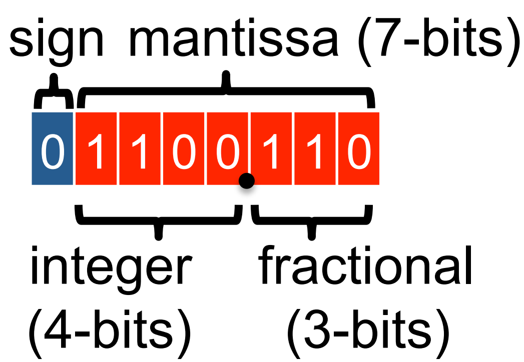
32-bit float



-1.42122425 x 10^{-13} ***s*** = 1 ***e*** = 70 ***m*** = 20482

Fixed Point

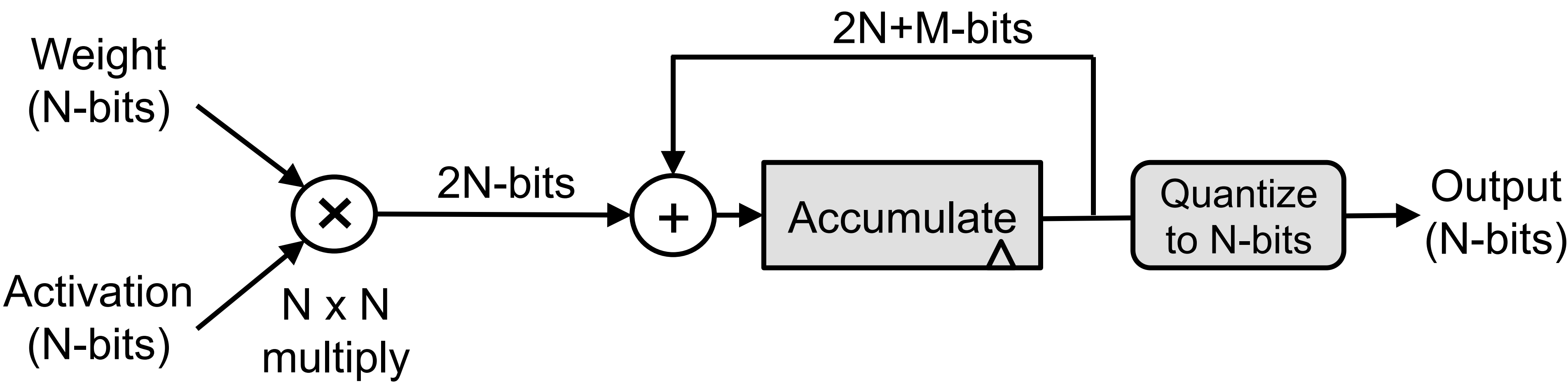
8-bit fixed



12.75 ***s*** = 0 ***m***=*102*

N-bit Precision

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



Dynamic Fixed Point

Floating Point

	sign	exponent (8-bits)	mantissa (23-bits)
32-bit float	1	0 1 0 0 1 0 1 0	0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0
-1.42122425 x 10^{-13}	***s*** = 1	***e*** = 70	***m*** = 20482

Fixed Point

8-bit dynamic fixed

sign: 0 | mantissa (7-bits): 1 1 0 0 . 1 1 0

integer ([7-***f***]-bits): 1 1 0 0 | fractional (***f***-bits): 1 1 0

12.75 ***s*** = 0 ***m***=*102* ***f*** = 3

8-bit dynamic fixed

sign: 0 . | mantissa (7-bits): 1 1 0 0 1 1 0

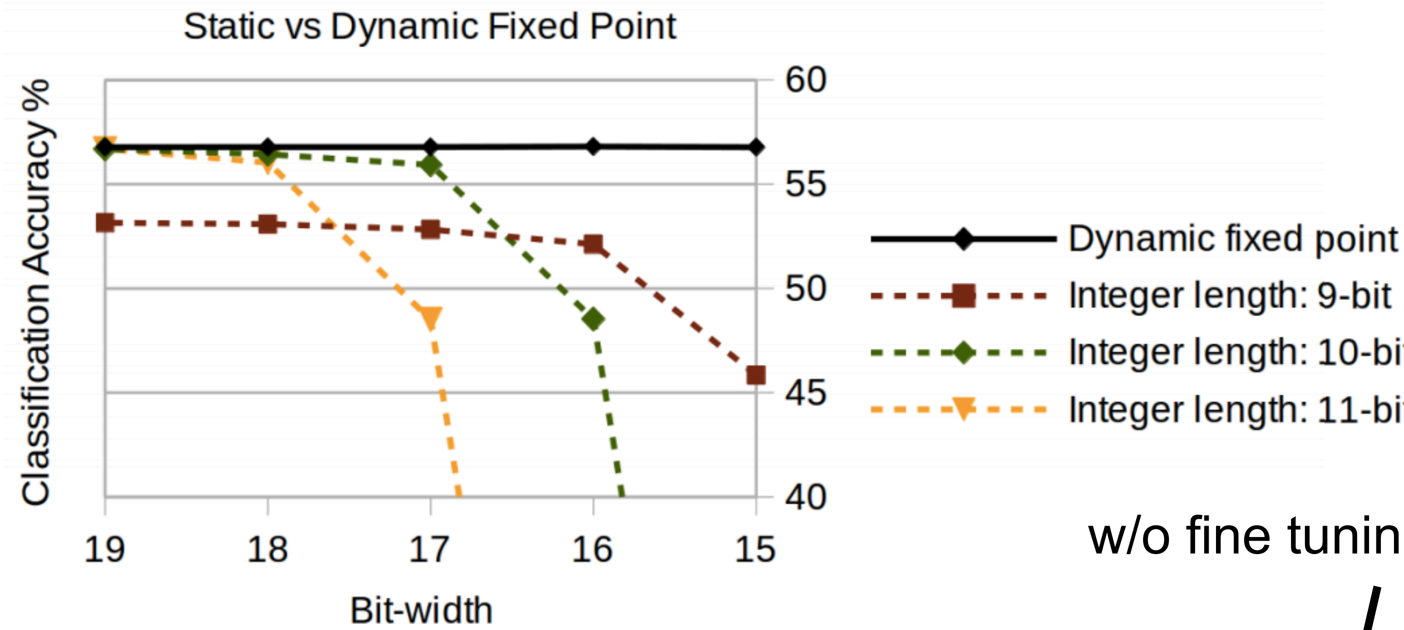
fractional (***f***-bits)

0.19921875 ***s*** = 0 ***m***=*102* ***f*** = 9

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

Impact on Accuracy

Top-1 accuracy on of CaffeNet on ImageNet



w/o fine tuning

	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%)

[Gysel et al., Ristretto, ICLR 2016]

Avoiding Dynamic Fixed Point

Batch normalization 'centers' dynamic range

AlexNet
(Layer 6)

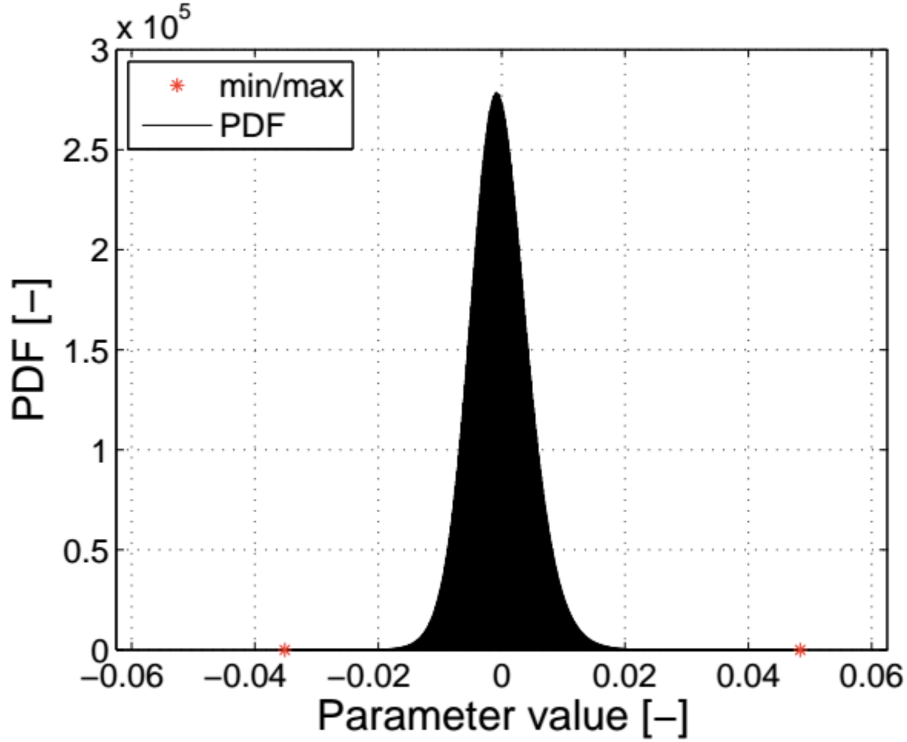


Image Source: Moons et al, WACV 2016

'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)

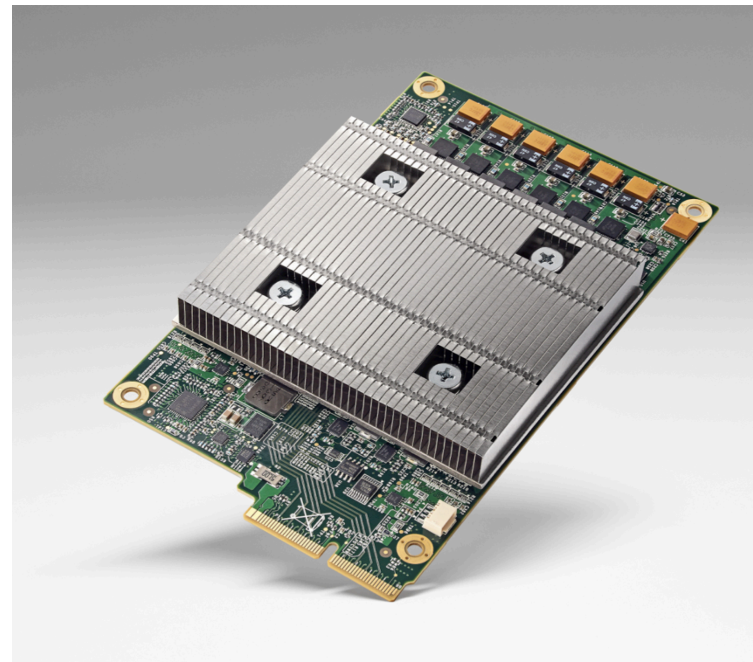
Google's Tensor Processing Unit (TPU)

" 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**."

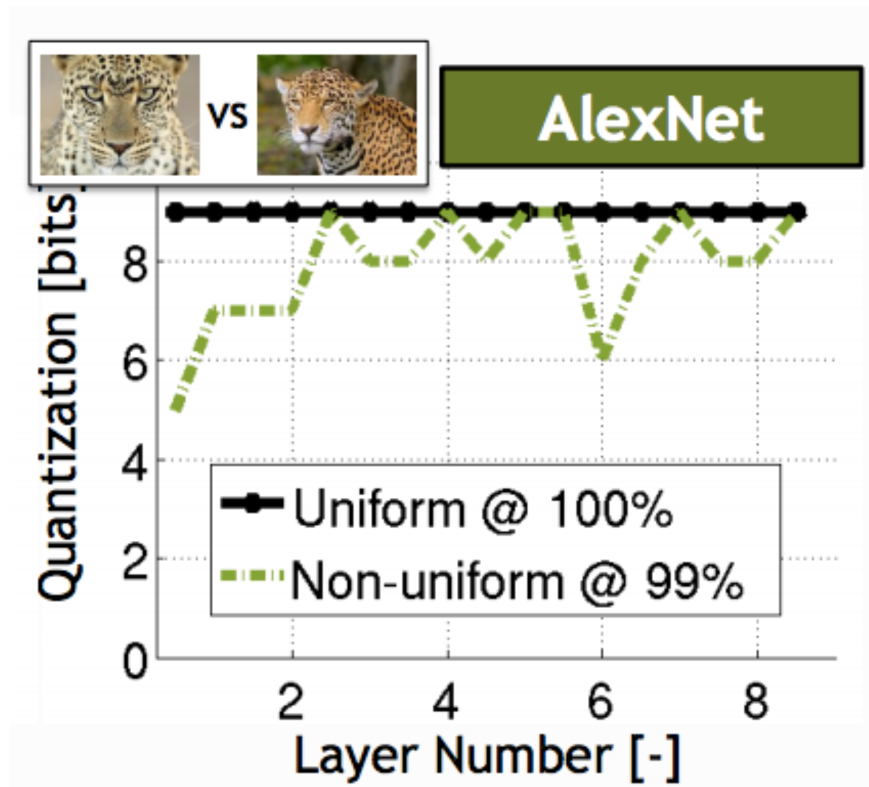
- Next Platform (May 19, 2016)



[Jouppi et al., ISCA 2017]

Precision Varies from Layer to Layer

Tolerance	Bits per layer (I+F)
AlexNet (F=0)	
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



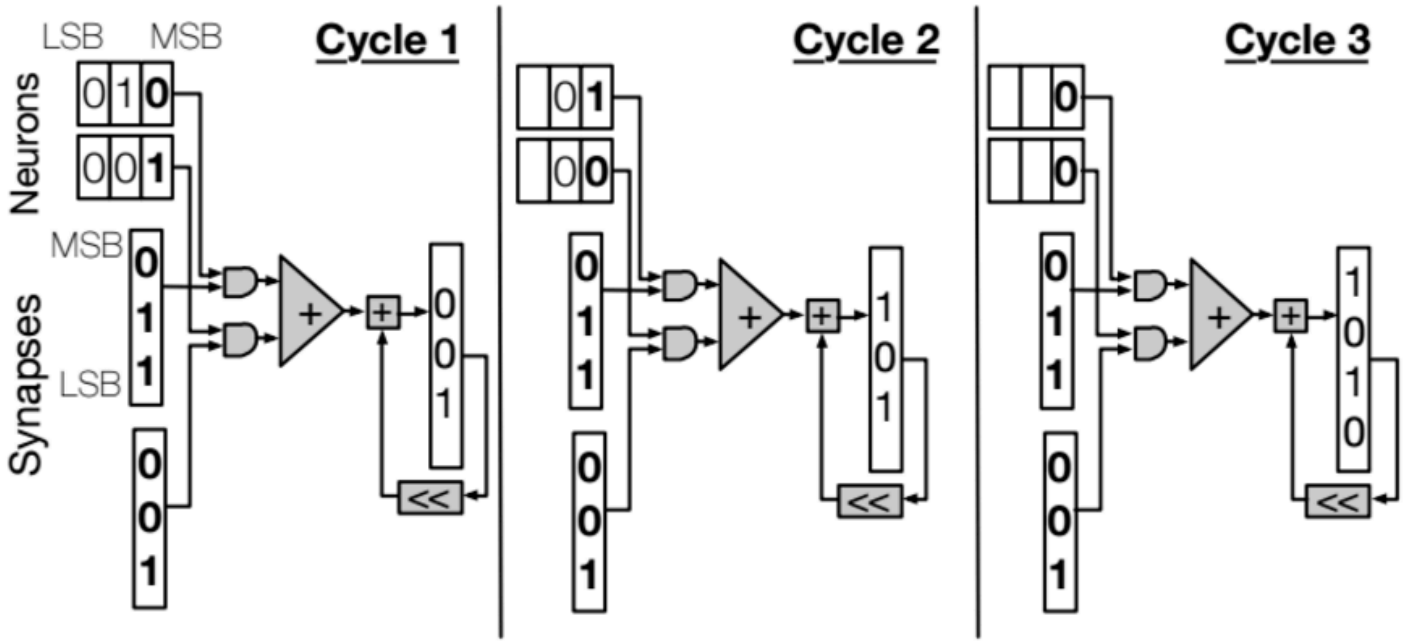
[Judd et al., ArXiv 2016]

[Moons et al., WACV 2016]

Bitwidth Scaling (Speed)

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

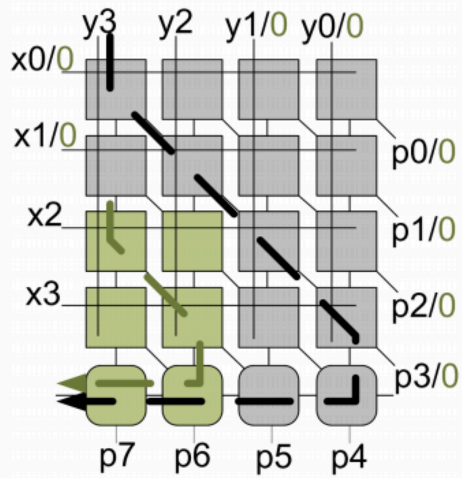
$$\sum_{i=0}^{N_i-1} s_i \times n_i = \sum_{i=0}^{N_i-1} s_i \times \sum_{b=0}^{P-1} n_i^b \times 2^b = \sum_{b=0}^{P-1} 2^b \times \sum_{i=0}^{N_i-1} n_i^b \times S_i$$

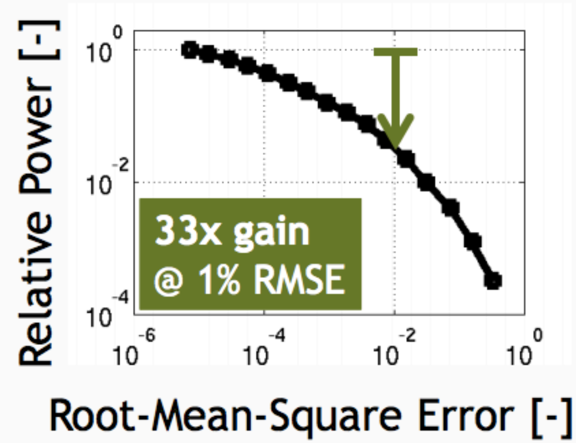


[Judd et al., Stripes, CAL 2016]

Bitwidth Scaling (Power)

Reduce Bit-width → Shorter Critical Path → Reduce Voltage





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

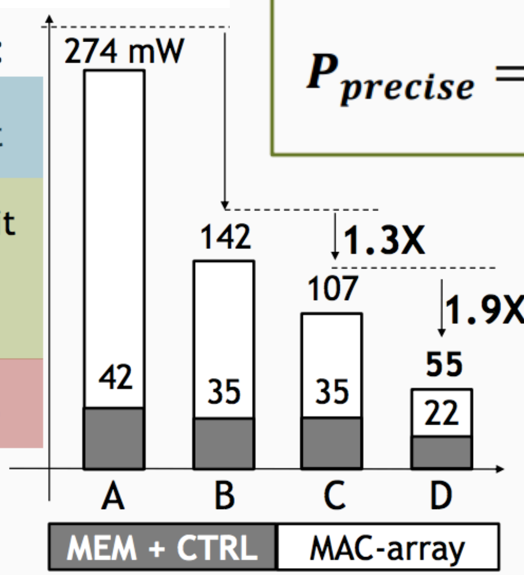
AlexNet Layer 2 example:

A. 2D-baseline @ 16 bit

B. Precision-Scaling @ 7-7 bit

C. Voltage-Scaling @ 0.9 V

D. Sparse operation guarding



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

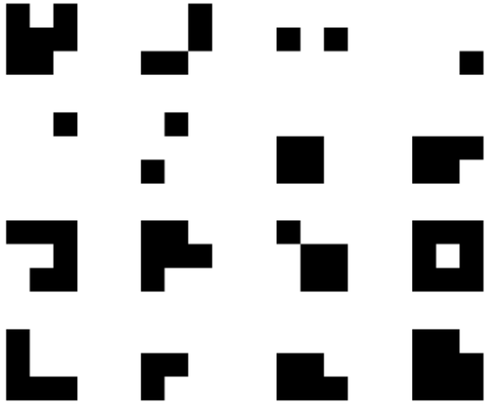
[Moons et al., VLSI 2016]

Binary Nets

- **Binary Connect (BC)**
  - Weights {-1,1}, Activations 32-bit float
  - MAC → addition/subtraction
  - Accuracy loss: **19%** on AlexNet

    [Courbariaux, NIPS 2015]

Binary Filters



- **Binarized Neural Networks (BNN)**
  - Weights {-1,1}, Activations {-1,1}
  - MAC → XNOR
  - Accuracy loss: **29.8%** on AlexNet

    [Courbariaux, arXiv 2016]

Scale the Weights and Activations

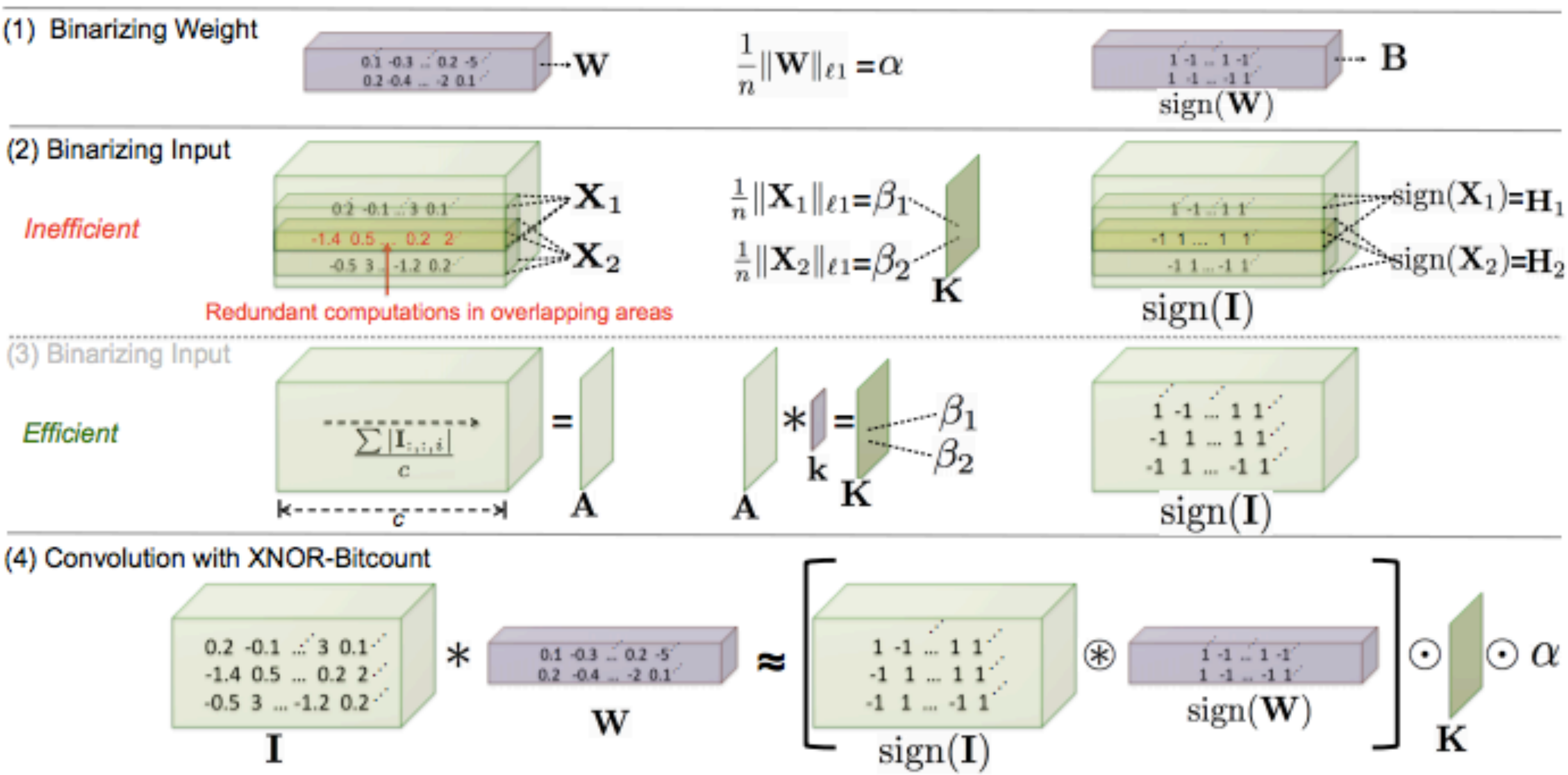
- **Binary Weight Nets (BWN)**
  - Weights {-α, α} → except first and last layers are 32-bit float
  - Activations: 32-bit float
  - α determined by the l_1-norm of all weights in a layer
  - Accuracy loss: **0.8%** on AlexNet
- **XNOR-Net**
  - Weights {-α, α}
  - Activations $\{-\beta_i, \beta_i\}$ → except first and last layers are 32-bit float
  - β_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 (α, β_i) can change per layer or position in filter

[Rastegari et al., BWN & XNOR-Net, ECCV 2016]

XNOR-Net



[Rastegari et al., BWN & XNOR-Net, ECCV 2016]

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

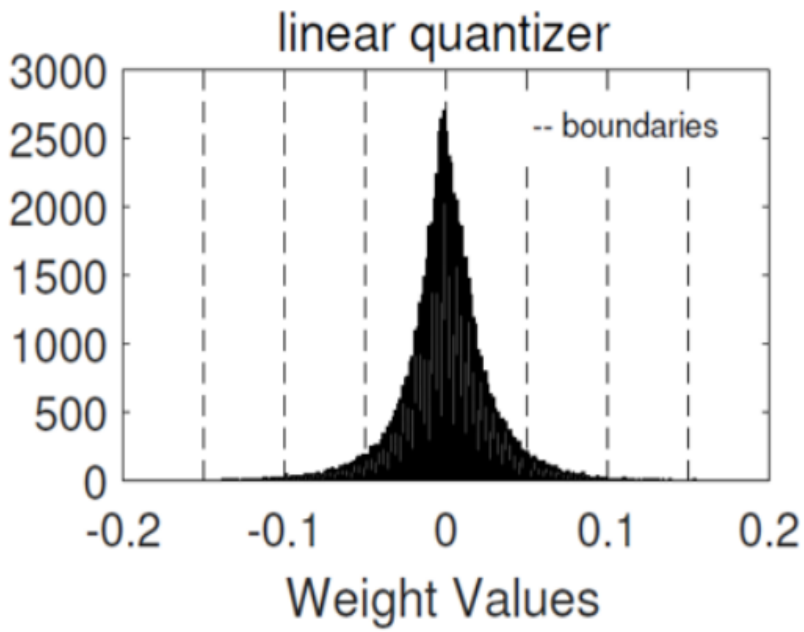
- **Precision** refers to the **number of levels**
  - Number of bits = $\log_2$ (number of levels)

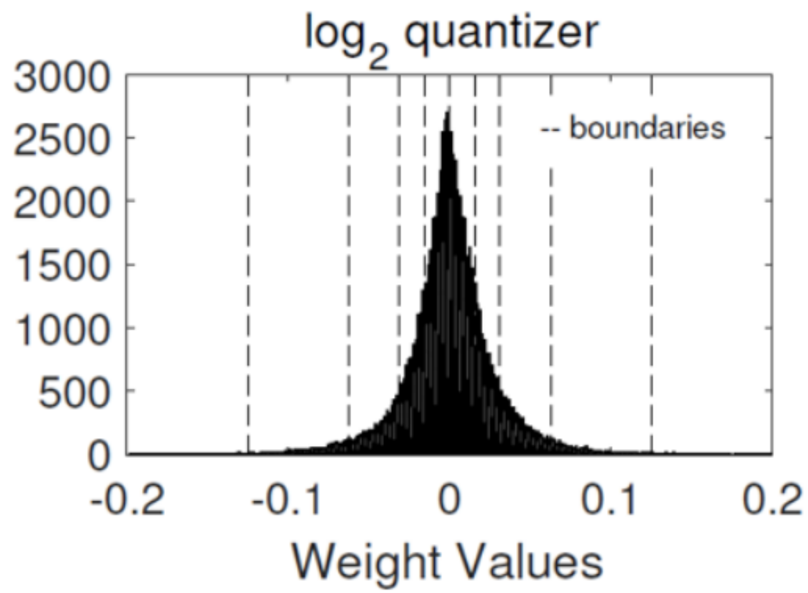
- **Quantization:** mapping data to a smaller set of **levels**
  - Linear, e.g., fixed-point
  - Non-linear
    - Computed
    - Table lookup

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

Computed Non-linear Quantization

Log Domain Quantization





Product = X * W

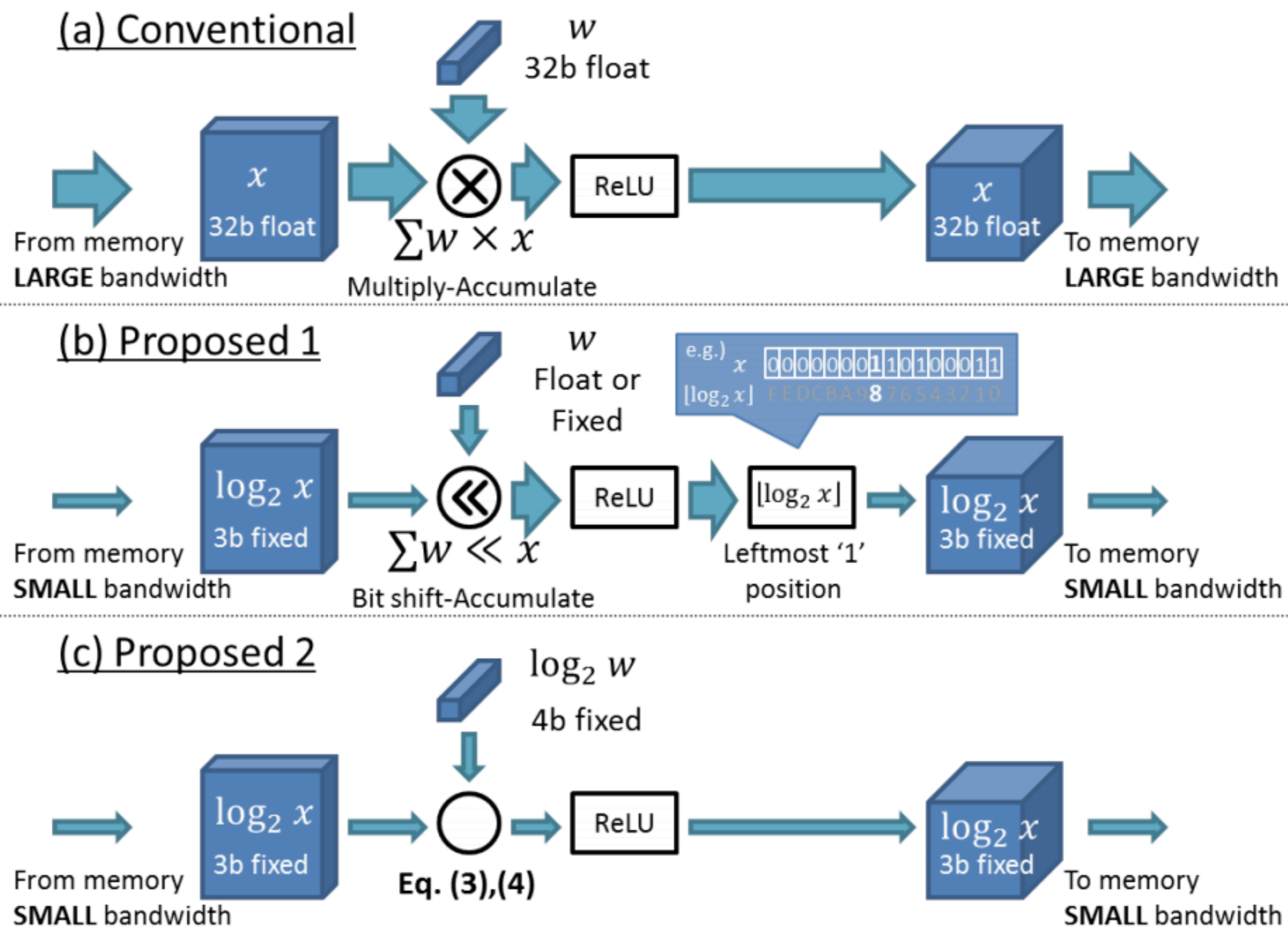
Product = X << W

[Lee et al., LogNet, ICASSP 2017]

Log Domain Computation

Only activation in log domain

Both weights and activations in log domain

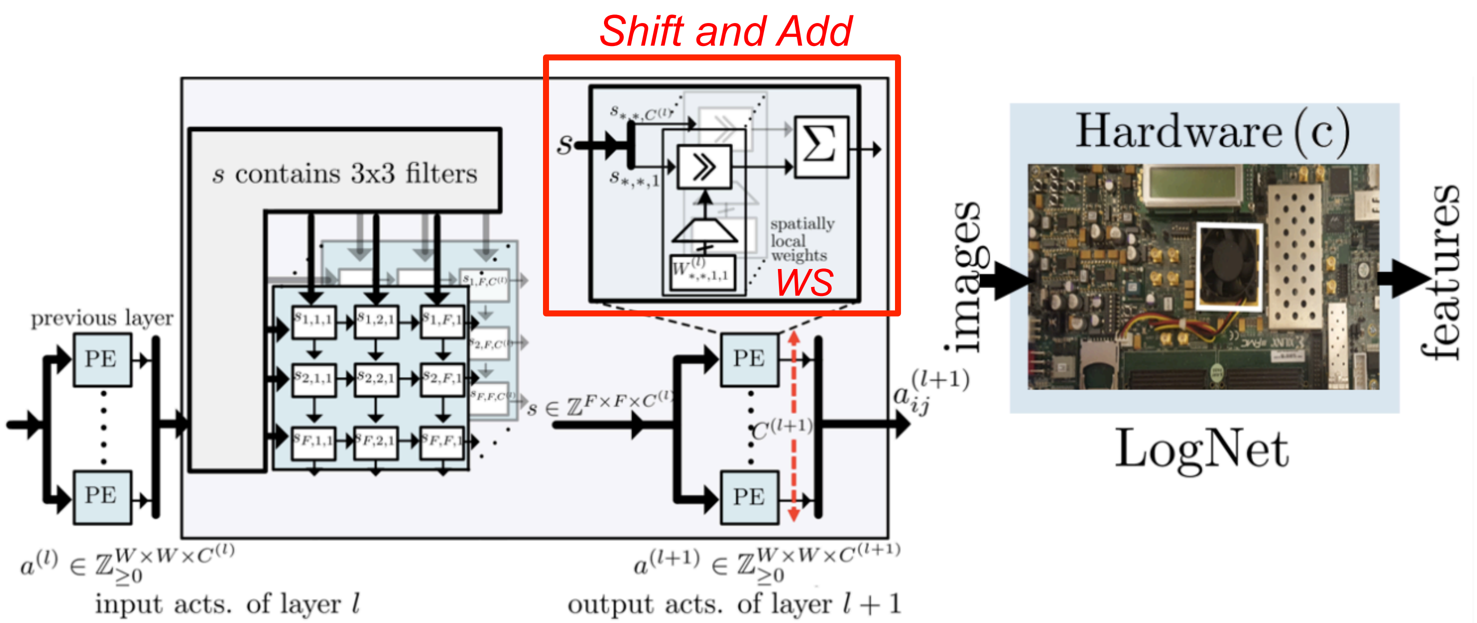


max, bitshifts, adds/subs

[Miyashita et al., arXiv 2016]

Log Domain Quantization

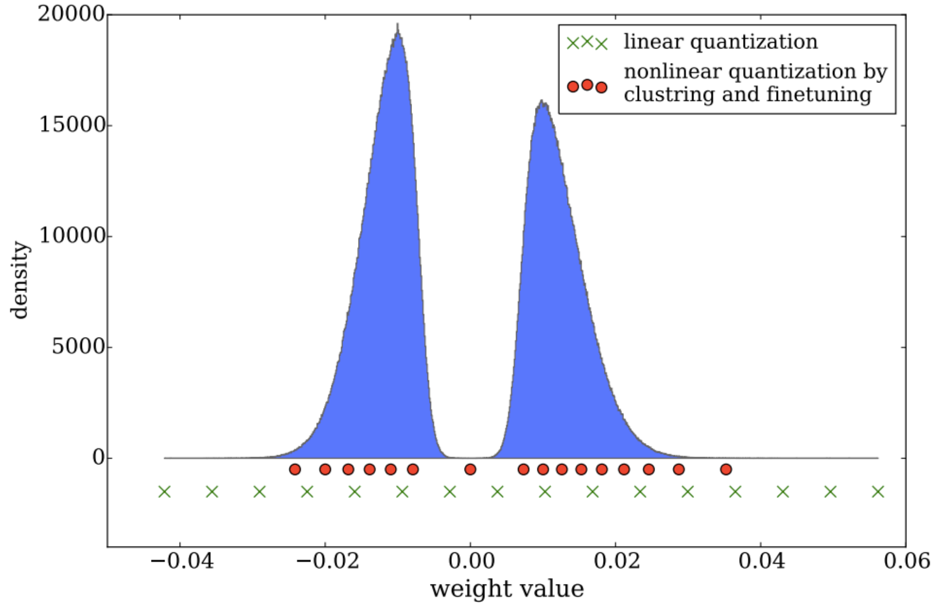
- **Weights: 5-bits for CONV, 4-bit for FC; Activations: 4-bits**
- Accuracy loss: 3.2% on AlexNet



[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

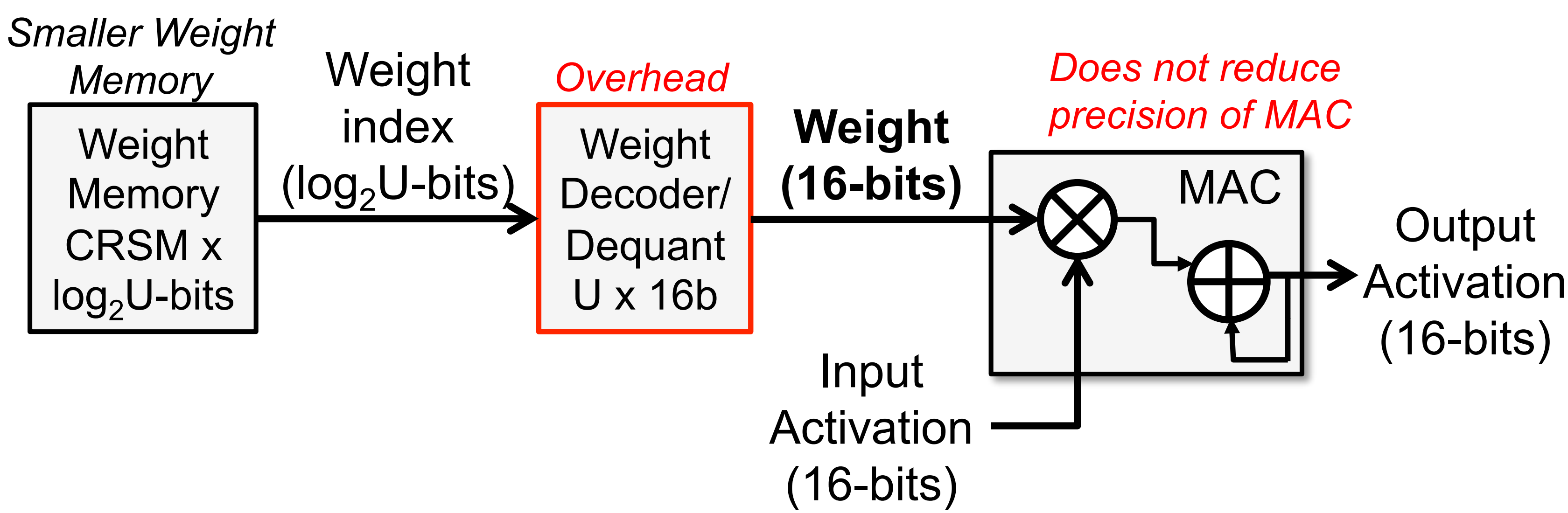
[Han et al., ICLR 2016]

- **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

[Han et al., Deep Compression, ICLR 2016]

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

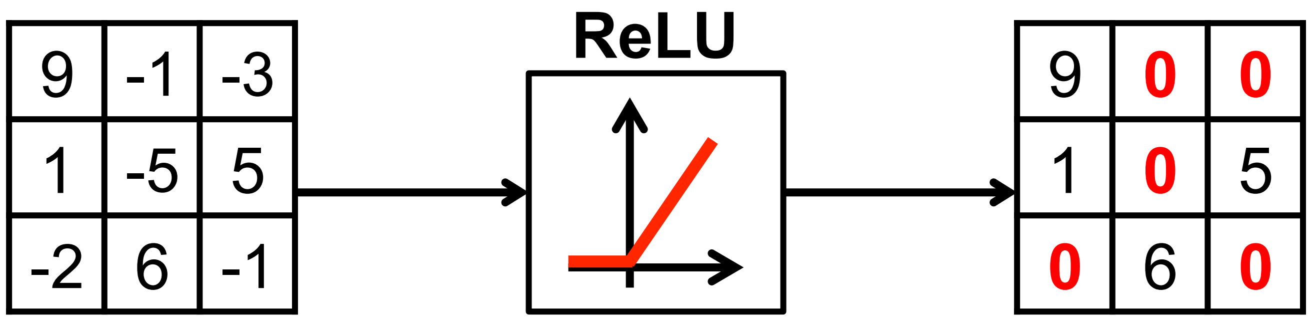
Full list @ [Sze et al., arXiv, 2017]

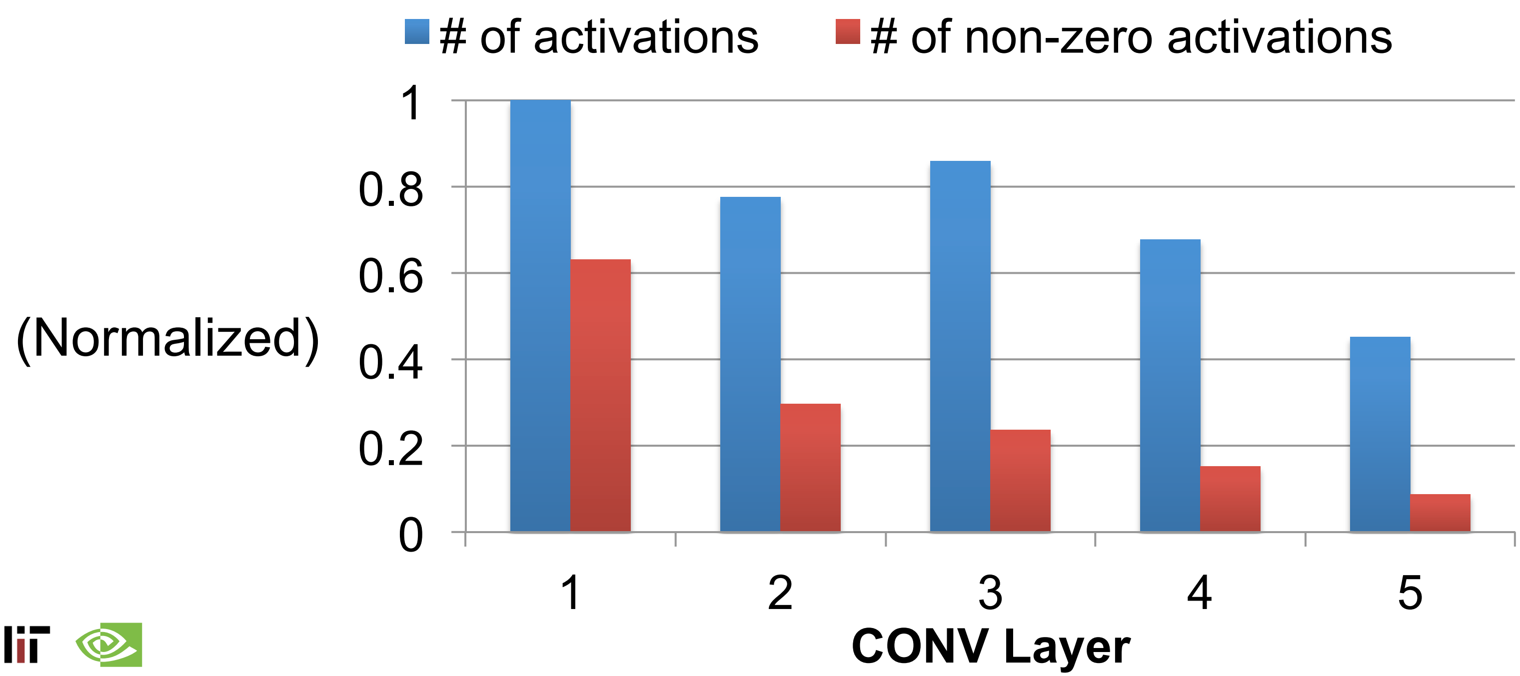
Reduce Number of Ops and Weights

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

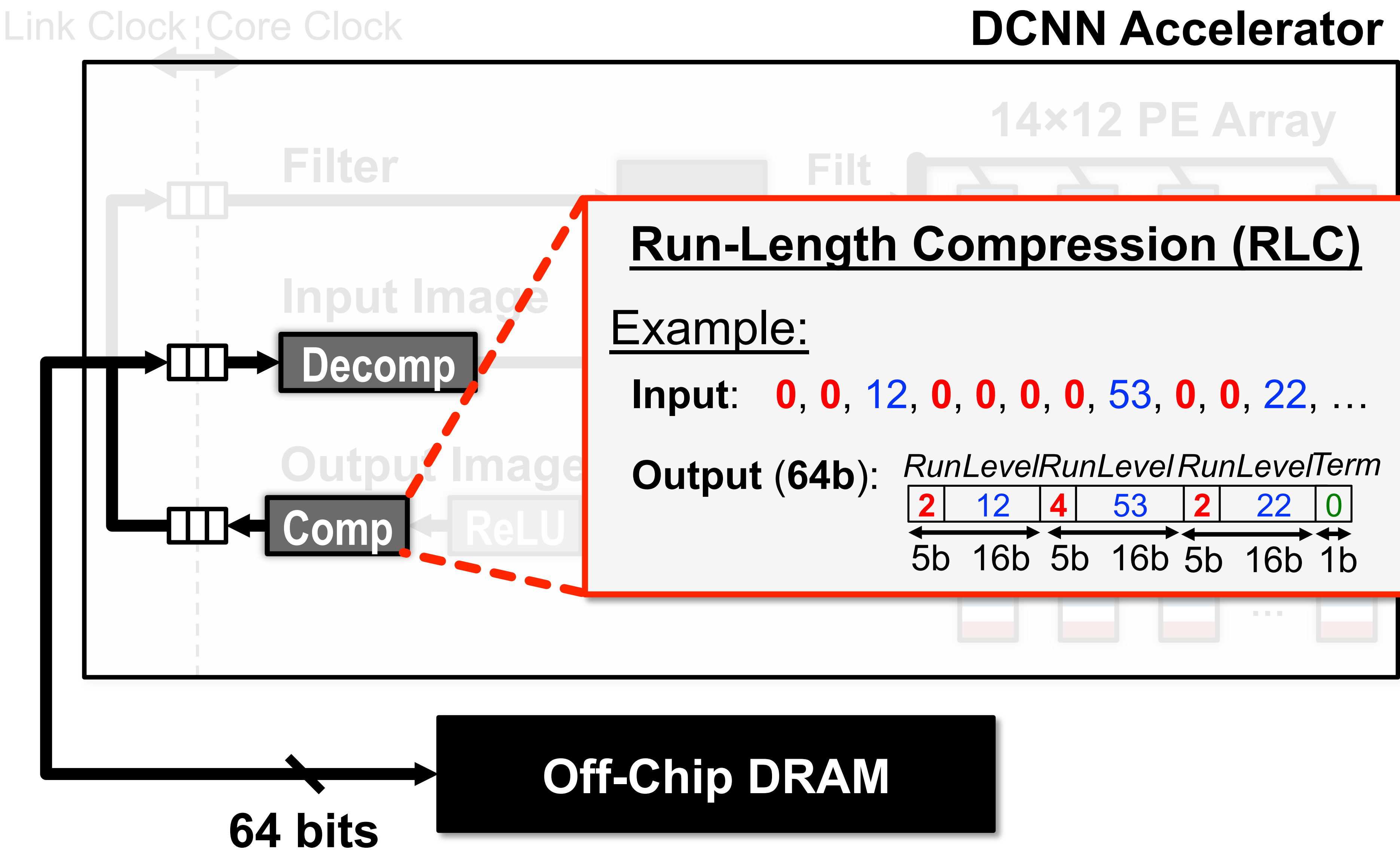
Sparsity in Fmaps

Many **zeros** in **output fmaps** after **ReLU**



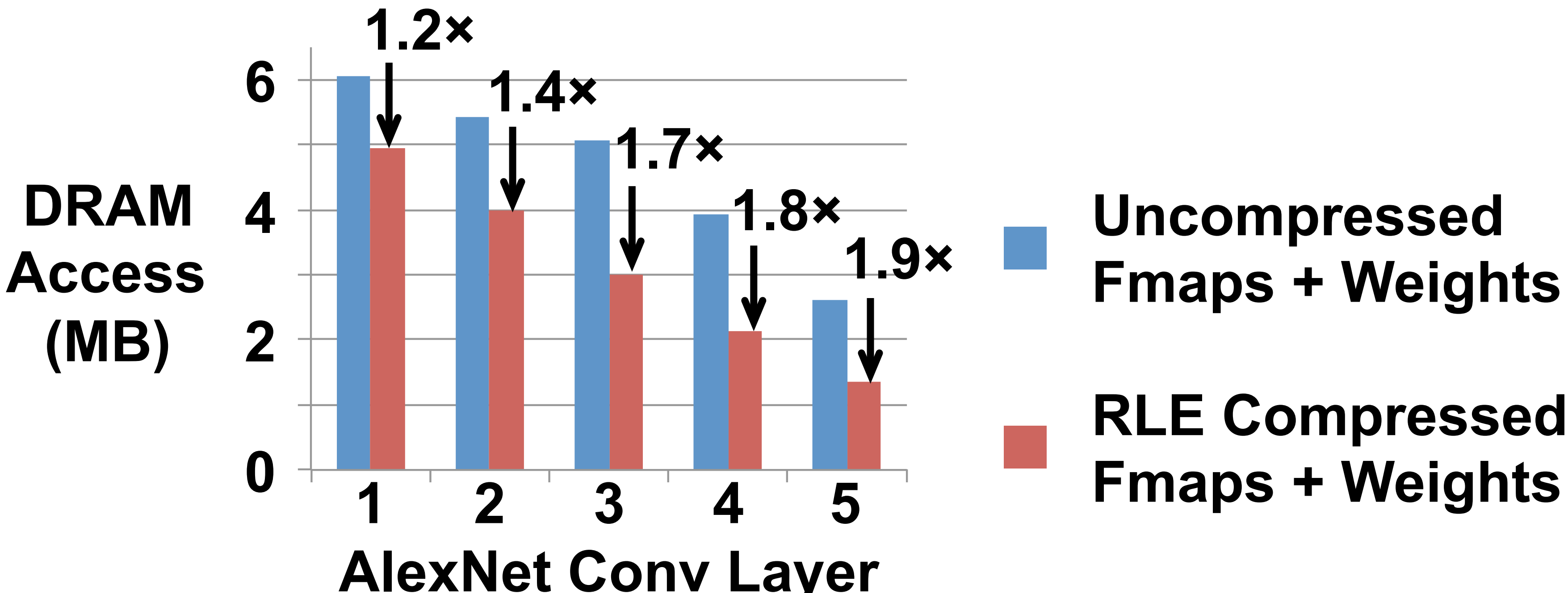


I/O Compression in Eyeriss



[Chen et al., ISSCC 2016]

Compression Reduces DRAM BW

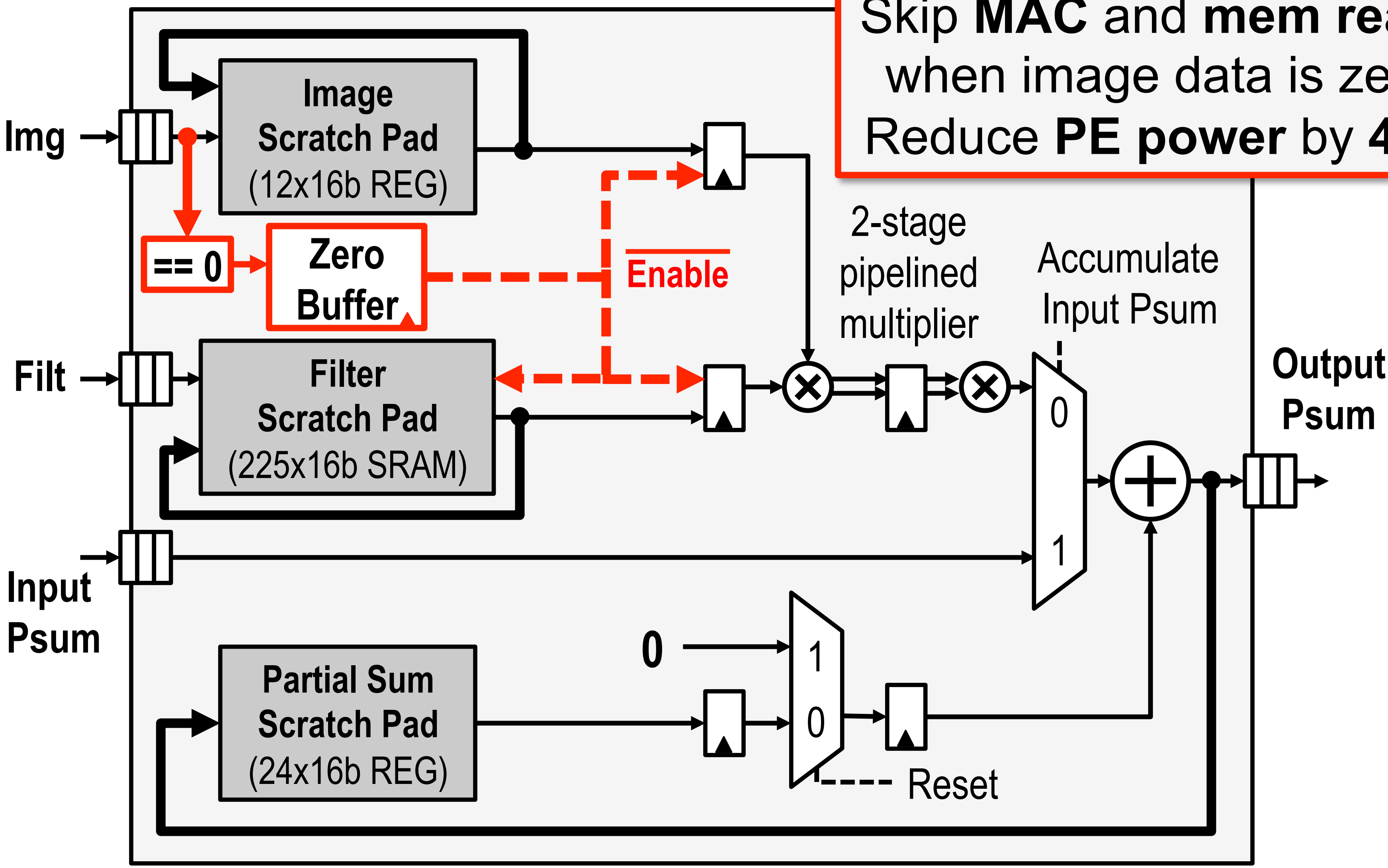


Simple RLC within 5% - 10% of theoretical entropy limit

[Chen et al., ISSCC 2016]

Data Gating / Zero Skipping in Eyeriss

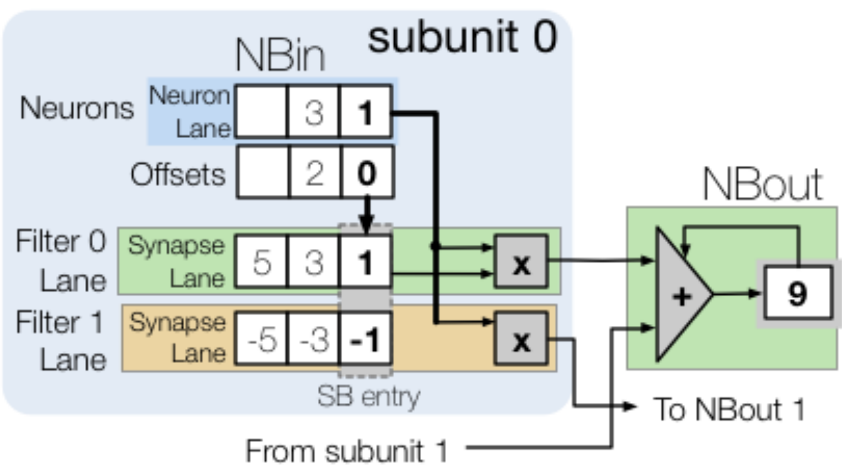
Skip **MAC** and **mem reads** when image data is zero.
Reduce **PE power** by **45%**

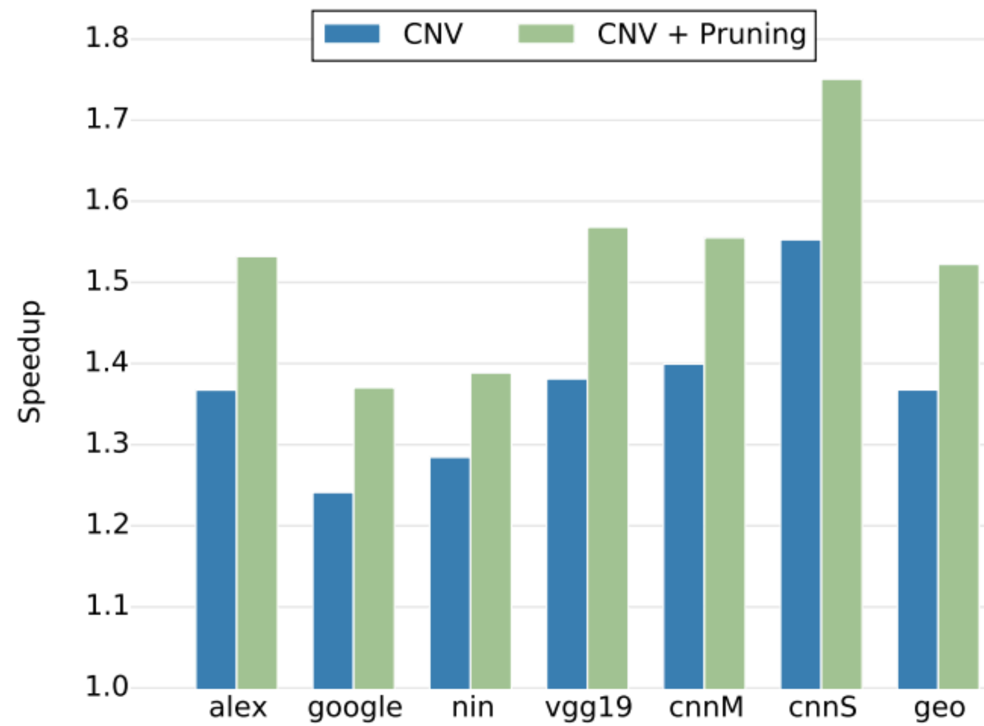


[Chen et al., ISSCC 2016]

Cnvlutin

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



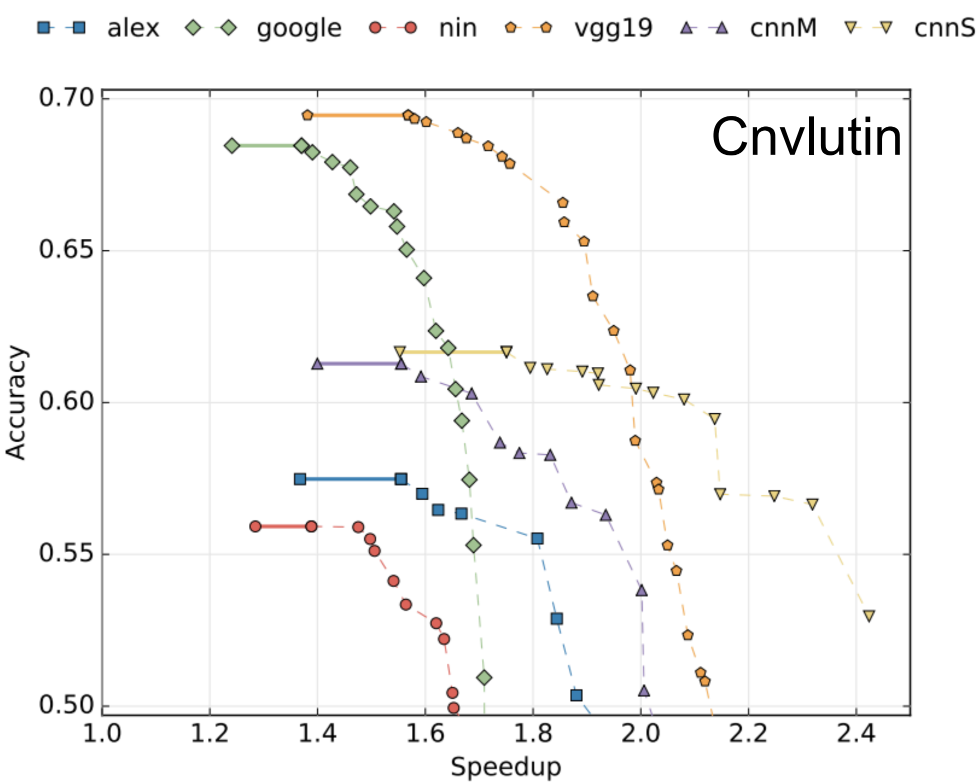


[Albericio et al., ISCA 2016]

Pruning Activations

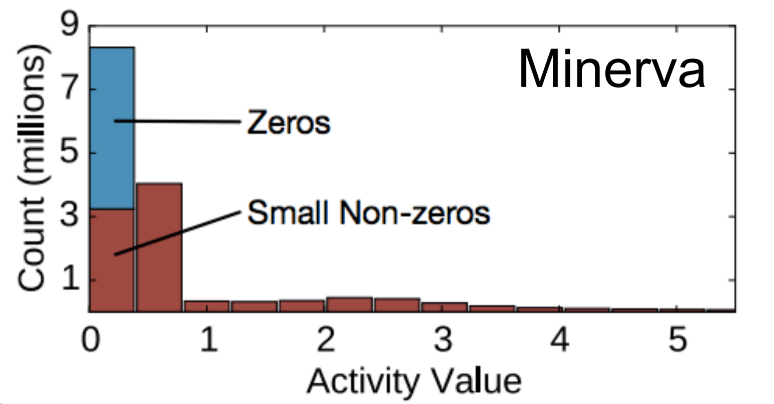
Remove small activation values

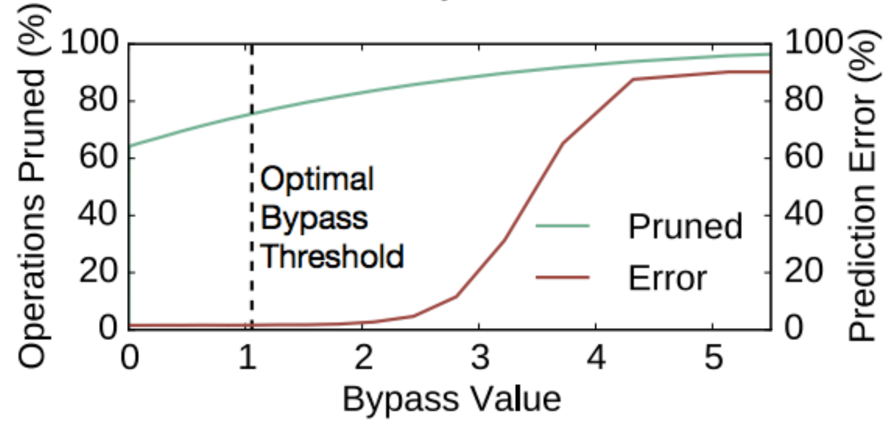
Speed up 11% (ImageNet)



[Albericio et al., ISCA 2016]

Reduce power 2x (MNIST)



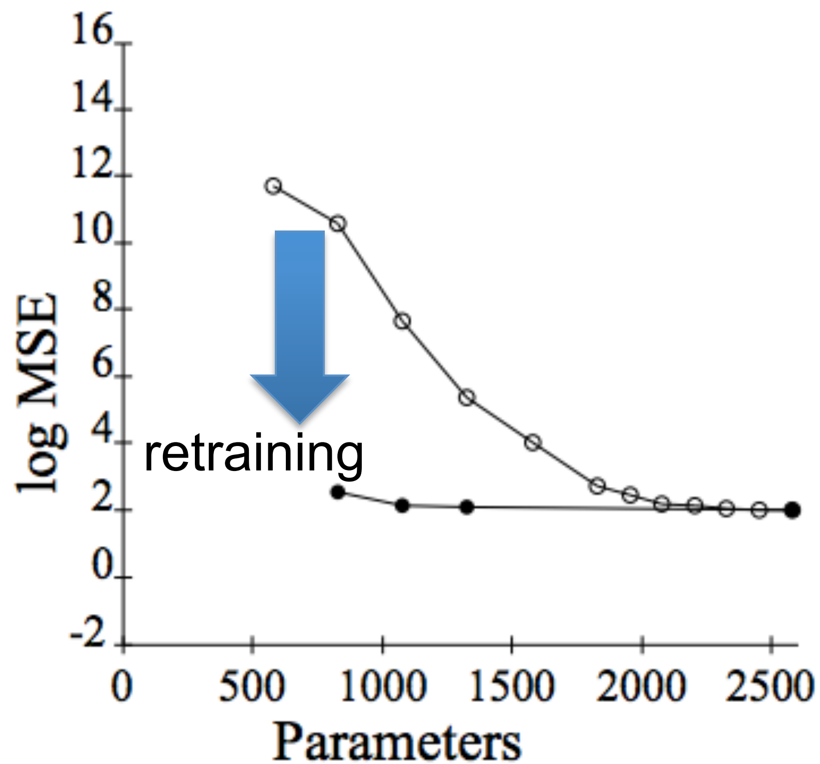


[Reagen et al., ISCA 2016]

Pruning – Make Weights Sparse

- **Optimal Brain Damage**

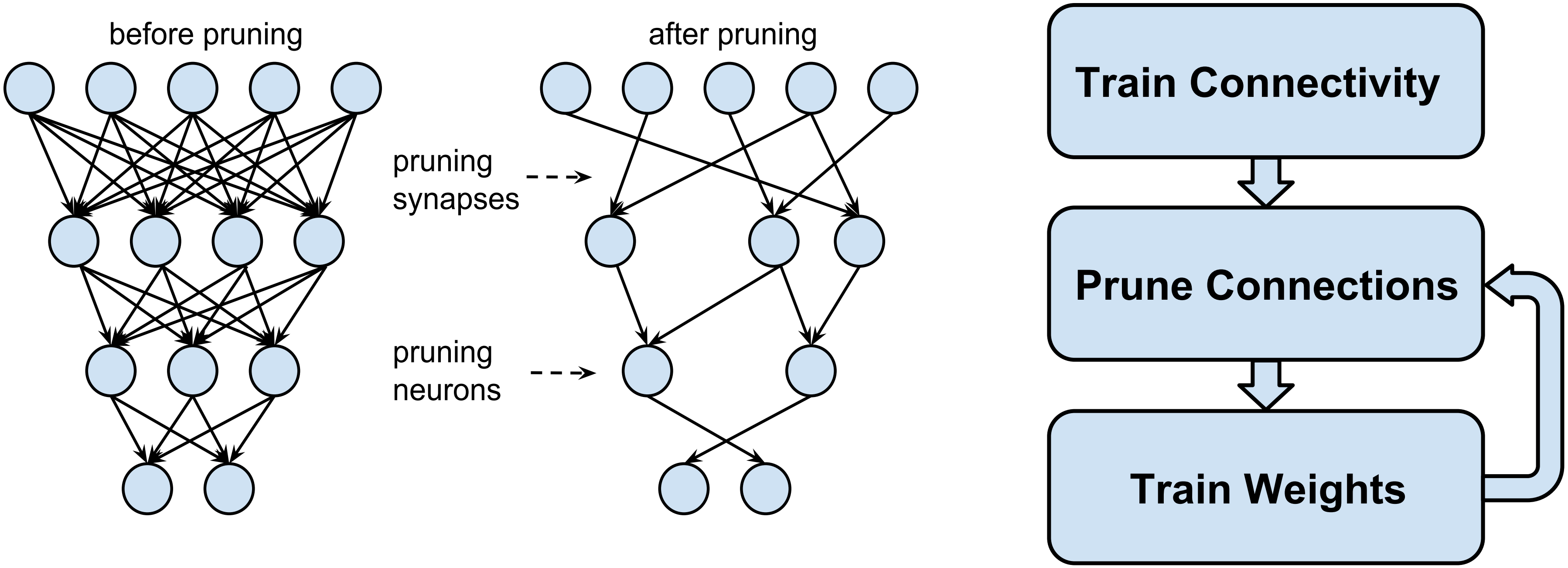
1. **Choose a reasonable network architecture**
2. **Train network until reasonable solution obtained**
3. **Compute the second derivative for each weight**
4. **Compute saliencies (i.e. impact on training error) for each weight**
5. **Sort weights by saliency and delete low-saliency weights**
6. **Iterate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



[Lecun et al., NIPS 1989]

Pruning – Make Weights Sparse

Prune based on ***magnitude*** of weights



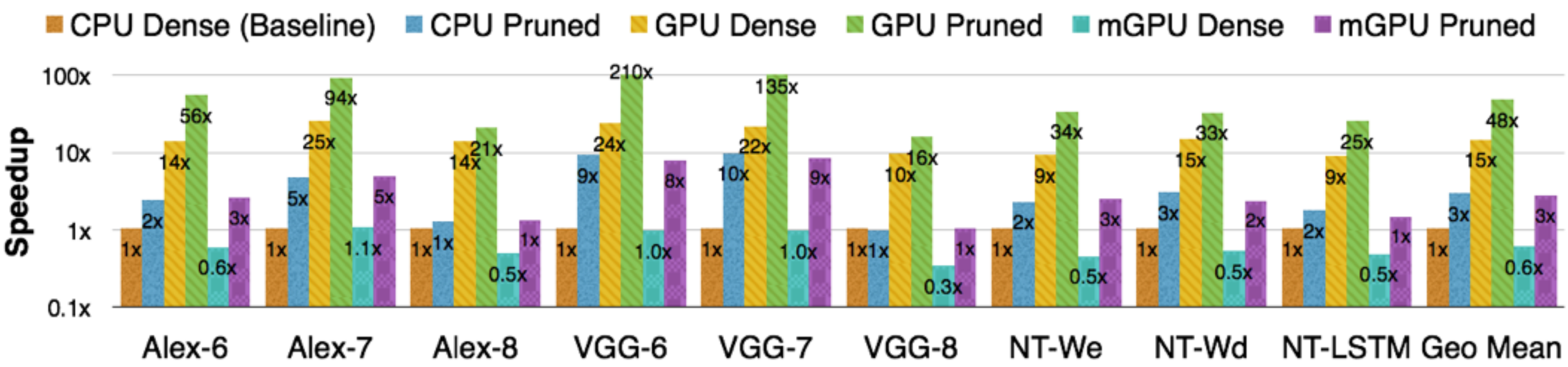
Example: *AlexNet*
Weight Reduction: *CONV layers 2.7x, FC layers 9.9x*
(Most reduction on fully connected layers)
Overall: *9x weight reduction, 3x MAC reduction*

[Han et al., NIPS 2015]

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

[Han et al., NIPS 2015]

Key Metrics for Embedded DNN

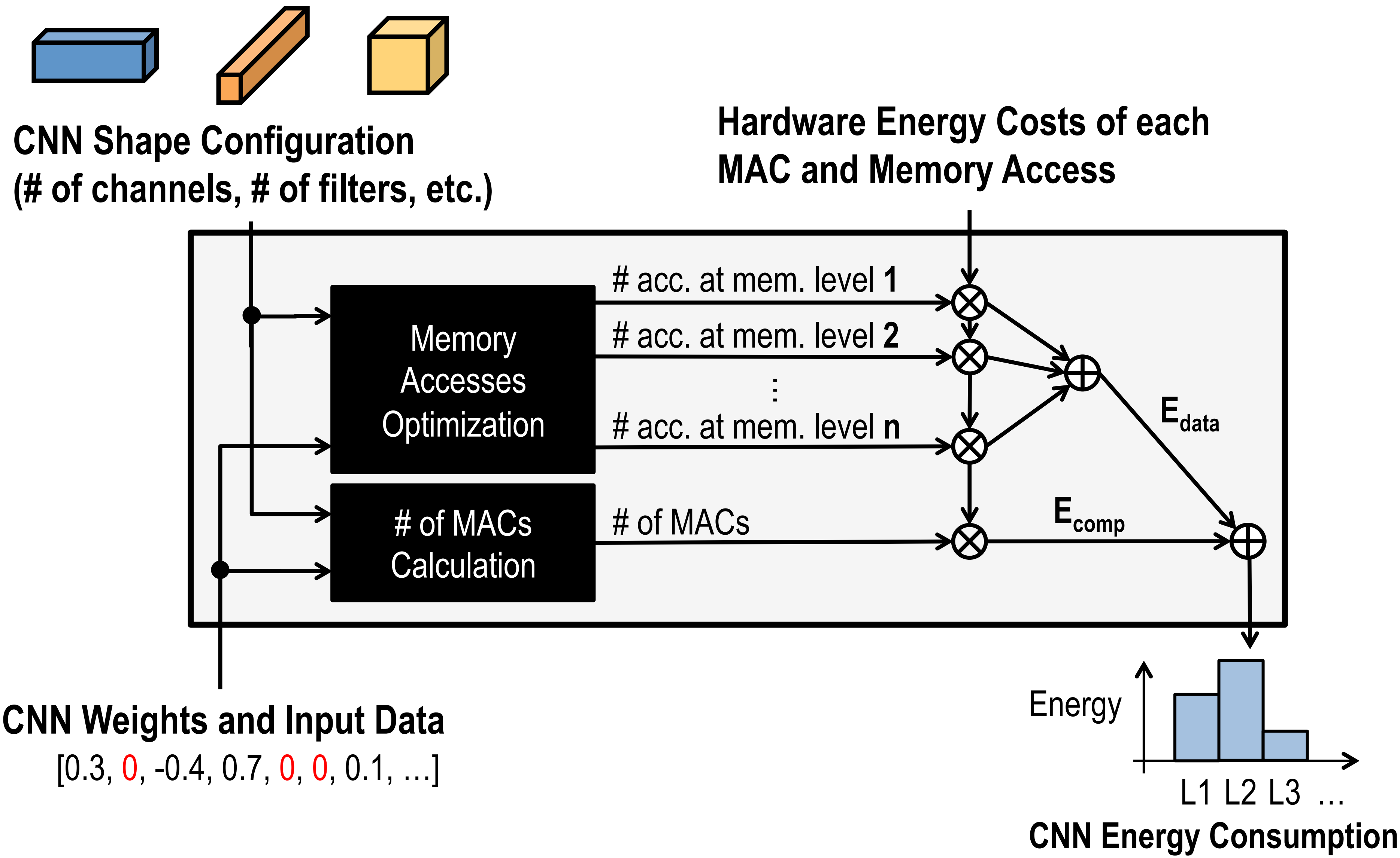
- **Accuracy → Measured on Dataset**
- **Speed → Number of MACs**
- **Storage Footprint → Number of Weights**
- **Energy → ?**

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**

[Yang et al., CVPR 2017]

Energy-Evaluation Methodology

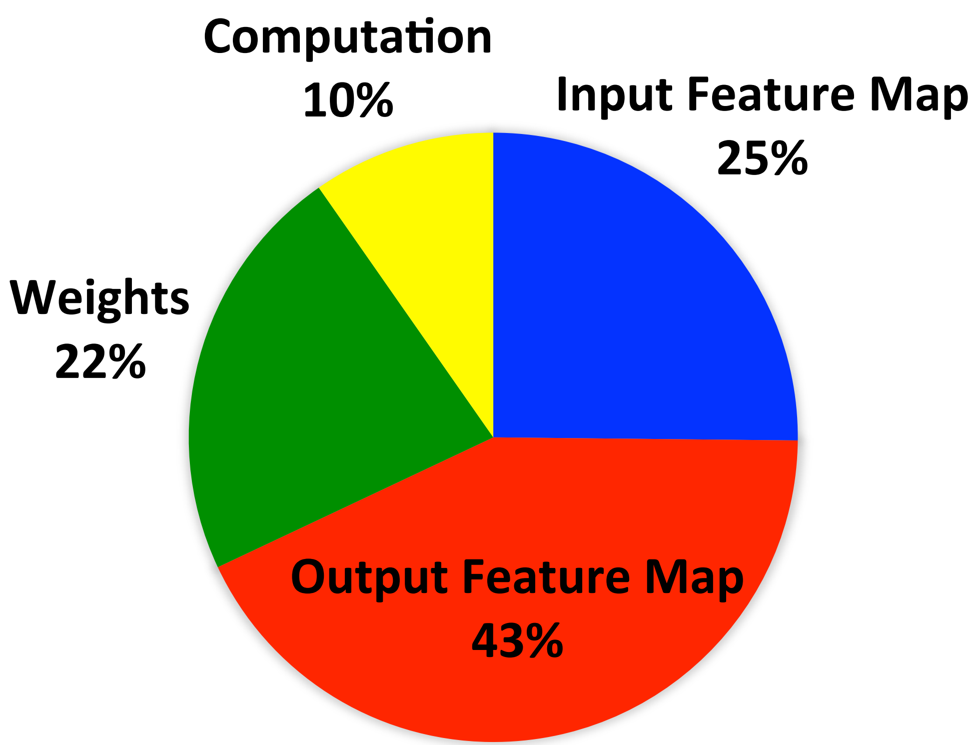


Evaluation tool available at http://eyeriss.mit.edu/energy.html

Key Observations

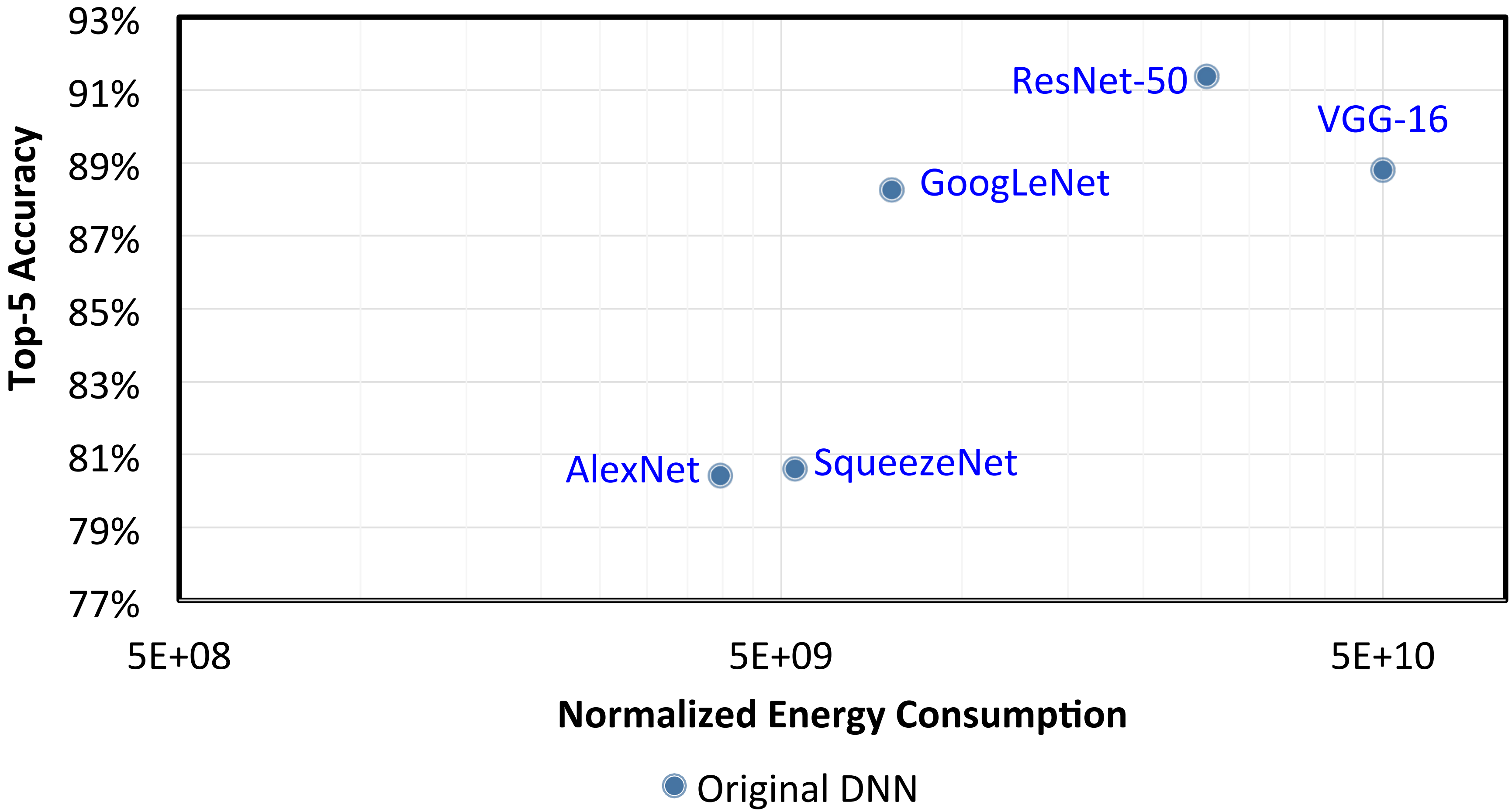
- Number of weights ***alone*** is not a good metric for energy
- **All data types** should be considered

Energy Consumption of GoogLeNet



[Yang et al., CVPR 2017]

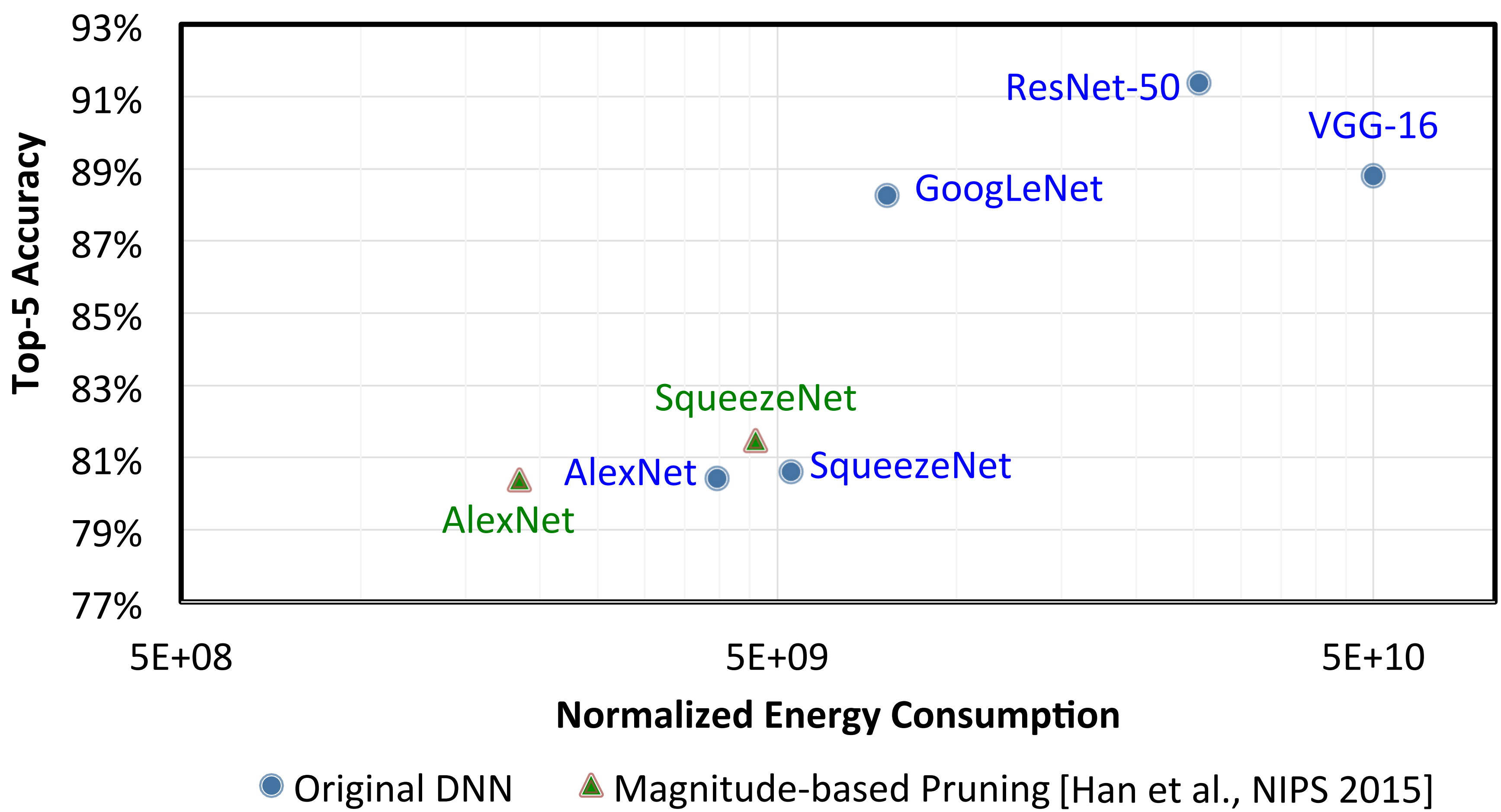
Energy Consumption of Existing DNNs



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

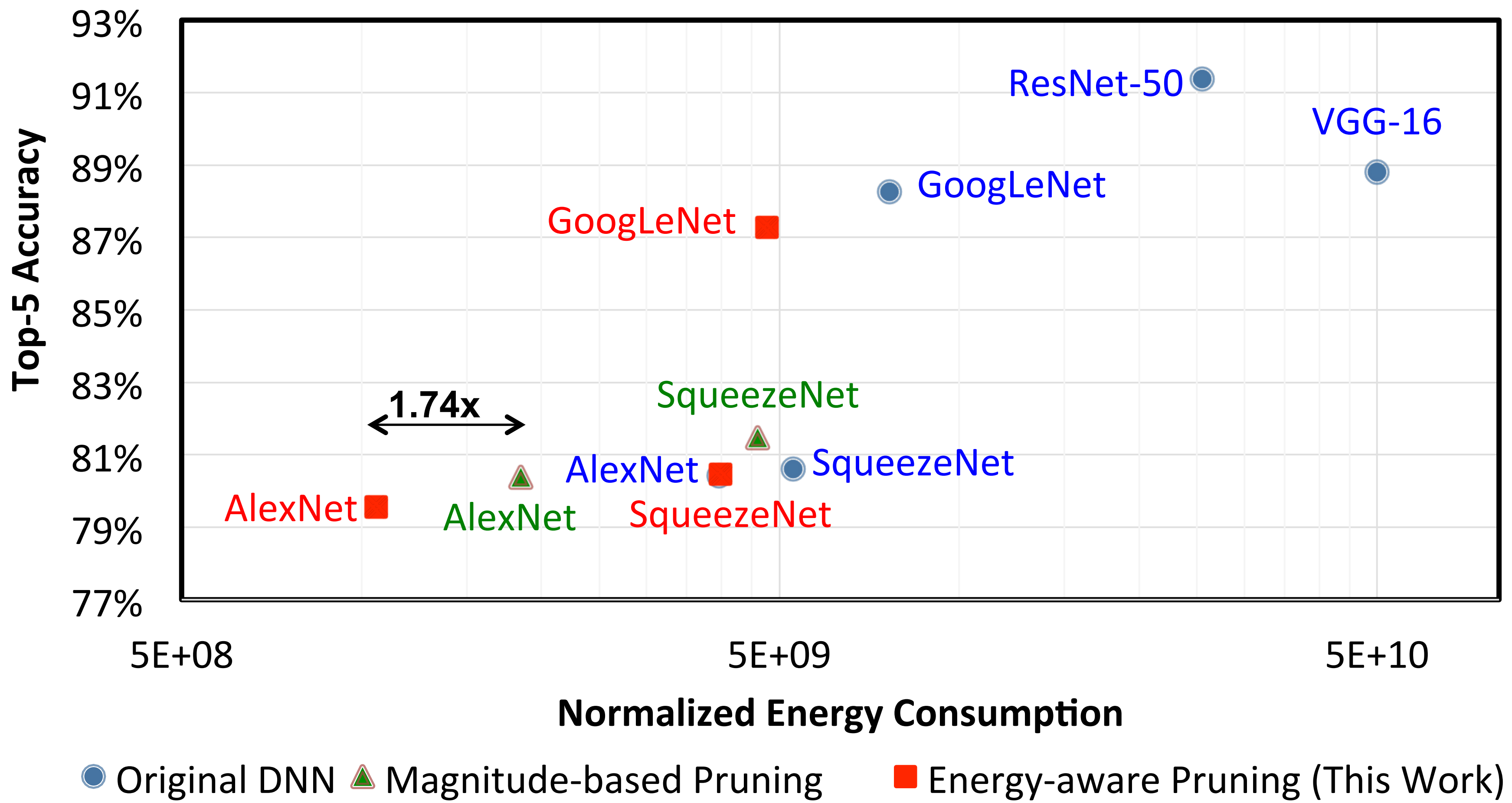
[Yang et al., CVPR 2017]

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

DNN Models available at http://eyeriss.mit.edu/energy.html

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:

height nChannels nZeroEntries y bottom right
width nMapsOrFilts bitwidth x top left

2 , 27 , 27 , 96 , 44 , 3.5731e+05 , 16 , 5 , 5 , 48 , 256 , 0 , 16 , 27 , 27 , 256 , 44 , 6.623e+06 , 16 , 1 , 1 , 2 , 2 , 2 , 2

Layer_Index Conf_IfMap Conf_Filt Conf_OfMap Stride Padding

- 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_or_filts 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

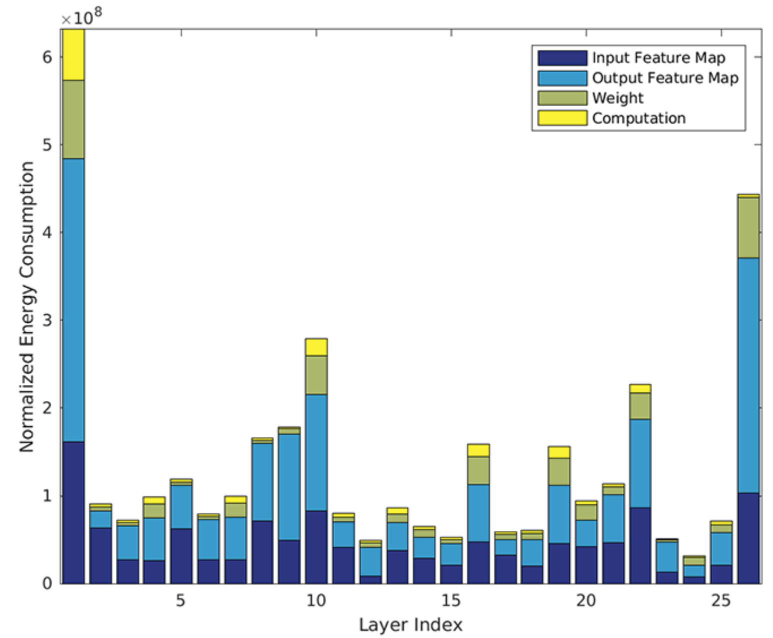
After creating your text file, follow these steps to upload your text file and run the estimation model:

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
3,26787638.0555562,39583335.5555542,3272222.77777708,2285942.77777708
4,26018817.2746958,48841502.8019458,15927826.1926396,7847418.06763958
5,62285050.8236438,49433953.294575,4188476.6472875,3227376.6472875
6,27267689.7685187,45381705.7407417,3740581.20370417,2666586.20370417
7,26787131.0480146,48586492.3413917,16216779.2956958,8136371.17069583
```

Output DNN energy breakdown across layers



[Yang et al., CVPR 2017]

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**

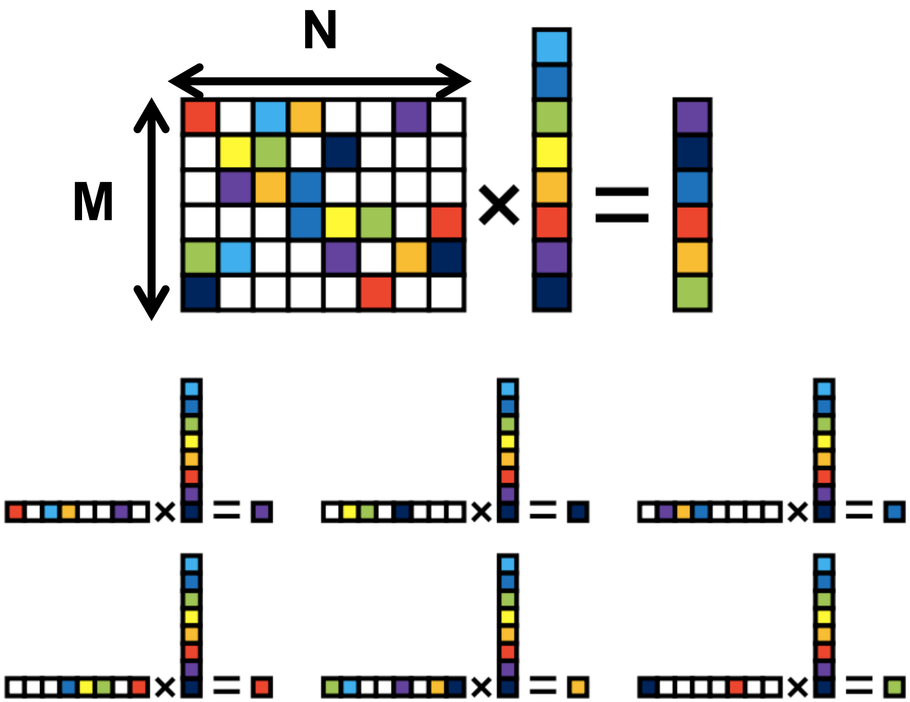
Layer	Filter / Image bits (0%)	Filter / Image BW Reduc.	IO / HuffIO (MB/frame)	Voltage (V)	MMACs/ Frame	Power (mW)	Real (TOPS/W)
General CNN	16 (0%) / 16 (0%)	1.0x		1.1	–	**288**	**0.3**
AlexNet l1	7 (21%) / 4 (29%)	1.17x / 1.3x	1 / 0.77	0.85	105	85	0.96
AlexNet l2	7 (19%) / 7 (89%)	1.15x / **5.8x**	3.2 / 1.1	0.9	224	55	1.4
AlexNet l3	8 (11%) / 9 (82%)	1.05x / 4.1x	6.5 / 2.8	0.92	150	77	0.7
AlexNet l4	9 (04%) / 8 (72%)	1.00x / 2.9x	5.4 / 3.2	0.92	112	95	0.56
AlexNet l5	9 (04%) / 8 (72%)	1.00x / 2.9x	3.7 / 2.1	0.92	75	95	0.56
Total / avg.	–	–	19.8 / **10**	–	–	**76**	**0.94**
LeNet-5 l1	3 (35%) / 1 (87%)	1.40x / 5.2x	0.003 / 0.001	0.7	0.3	25	1.07
LeNet-5 l2	4 (26%) / 6 (55%)	1.25x / 1.9x	0.050 / 0.042	0.8	1.6	35	1.75
Total / avg.	–	–	0.053 / **0.043**	–	–	**33**	**1.6**

[Moons et al., VLSI 2016; Han et al., ICLR 2016]

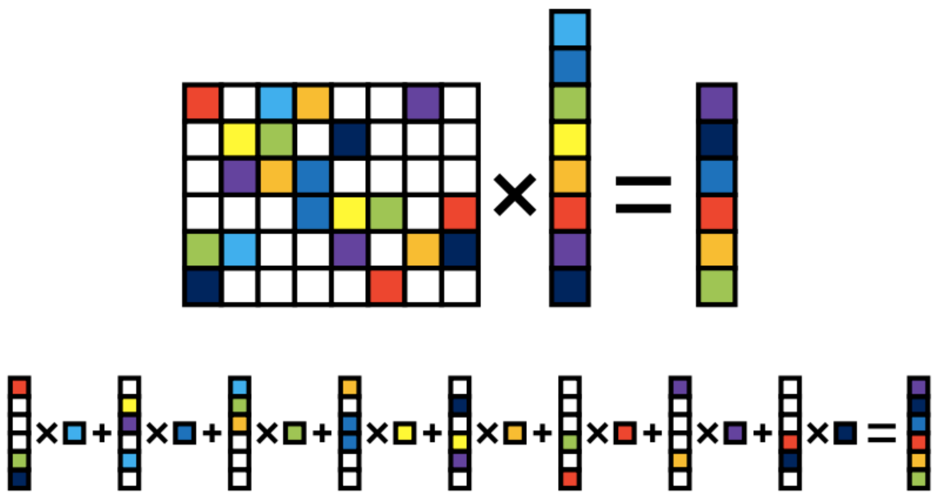
Sparse Matrix-Vector DSP

- **Use CSC rather than CSR for SpMxV**

Compressed Sparse Row (CSR)



Compressed Sparse Column (CSC)



Reduce memory bandwidth (when not **M** >> **N**)

*For DNN, **M** = # of filters, **N** = # of weights per filter*

[Dorrance et al., FPGA 2014]

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

Input

$$\begin{pmatrix} 0 & \boxed{a_1} & 0 & a_3 \end{pmatrix}$$

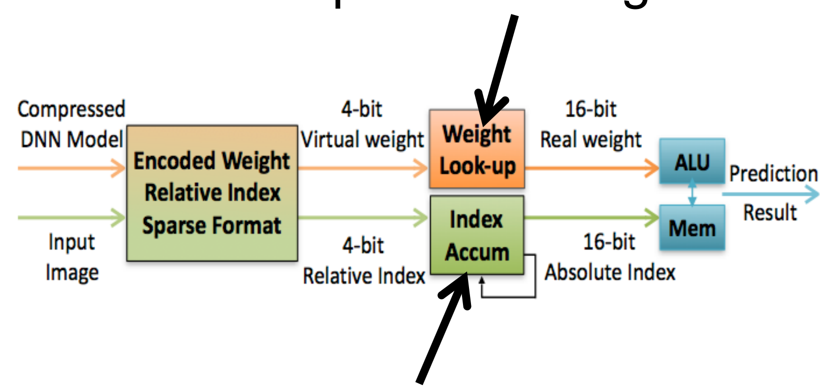
$$\times$$

Weights

Output

$$\begin{pmatrix} w_{0,0} & \boldsymbol{w_{0,1}} & 0 & w_{0,3} \\ 0 & \mathbf{0} & w_{1,2} & 0 \\ 0 & \boldsymbol{w_{2,1}} & 0 & w_{2,3} \\ 0 & \mathbf{0} & 0 & 0 \\ 0 & \mathbf{0} & w_{4,2} & w_{4,3} \\ w_{5,0} & \mathbf{0} & 0 & 0 \\ 0 & \mathbf{0} & 0 & w_{6,3} \\ 0 & \boldsymbol{w_{7,1}} & 0 & 0 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ -b_2 \\ b_3 \\ -b_4 \\ b_5 \\ b_6 \\ -b_7 \end{pmatrix} \overset{ReLU}{\Rightarrow} \begin{pmatrix} b_0 \\ b_1 \\ 0 \\ b_3 \\ 0 \\ b_5 \\ b_6 \\ 0 \end{pmatrix}$$

Dequantize Weight



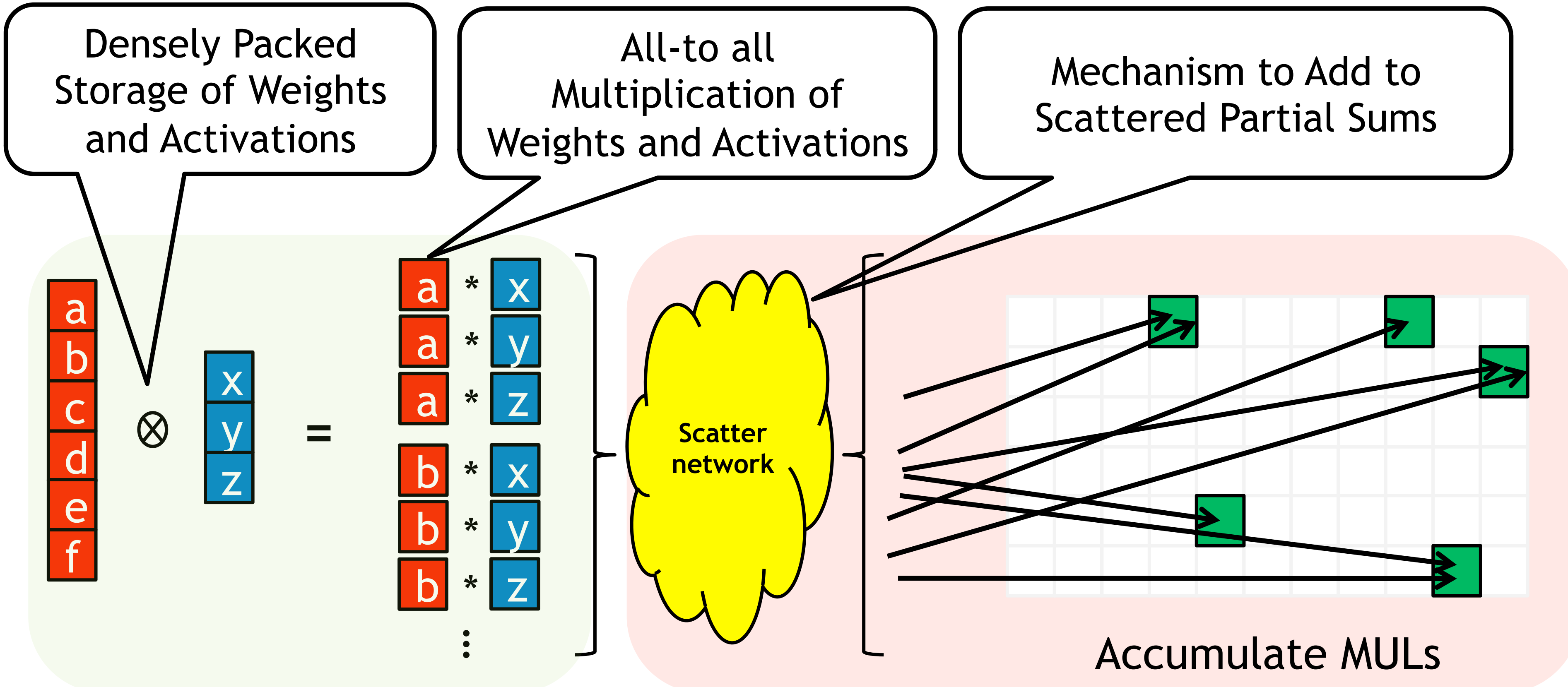
Keep track of location

Output Stationary Dataflow

[Han et al., ISCA 2016]

Sparse CNN (SCNN)

Supports Convolutional Layers



PE frontend

PE backend

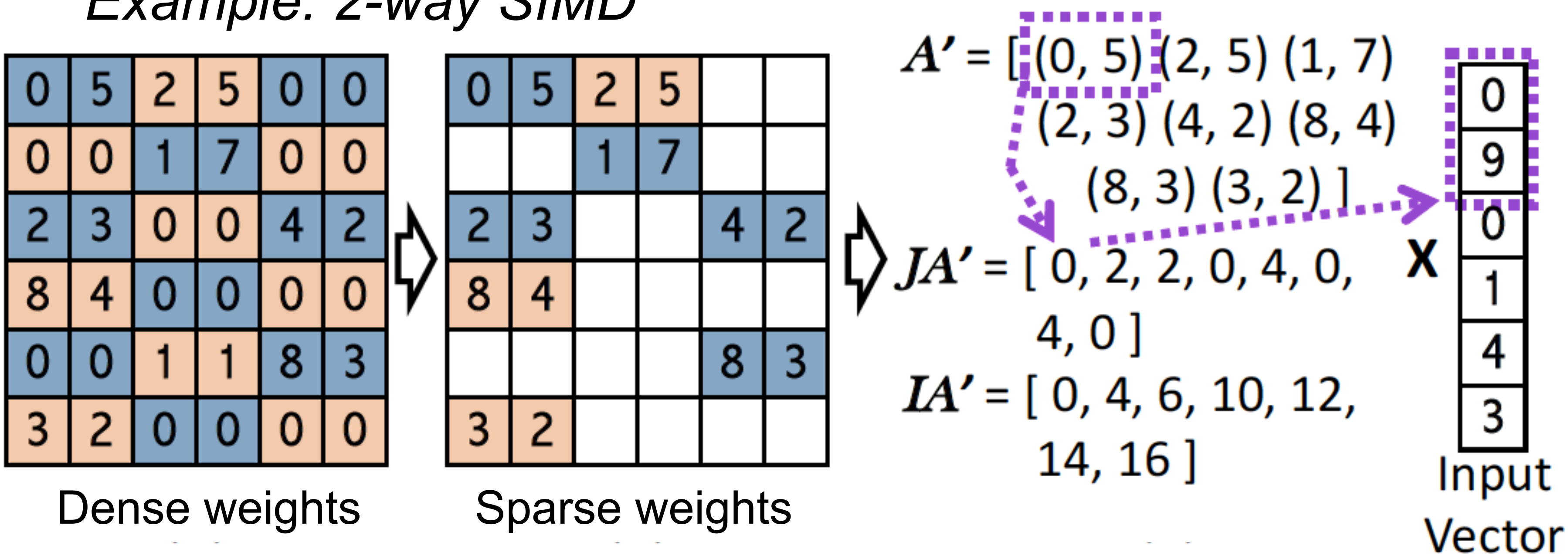
Input Stationary Dataflow

[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



[Yu et al., ISCA 2017]

Compact Network Architectures

- Break large convolutional layers into a series of smaller convolutional layers
  - Fewer weights, but same effective receptive field

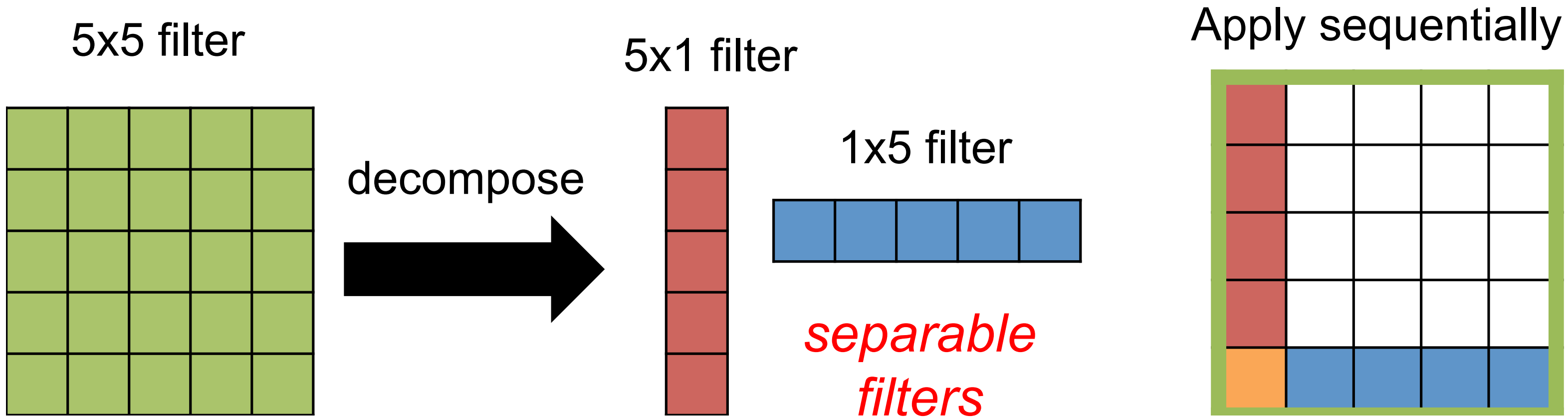
- Before Training: Network Architecture Design

- After Training: Decompose Trained Filters

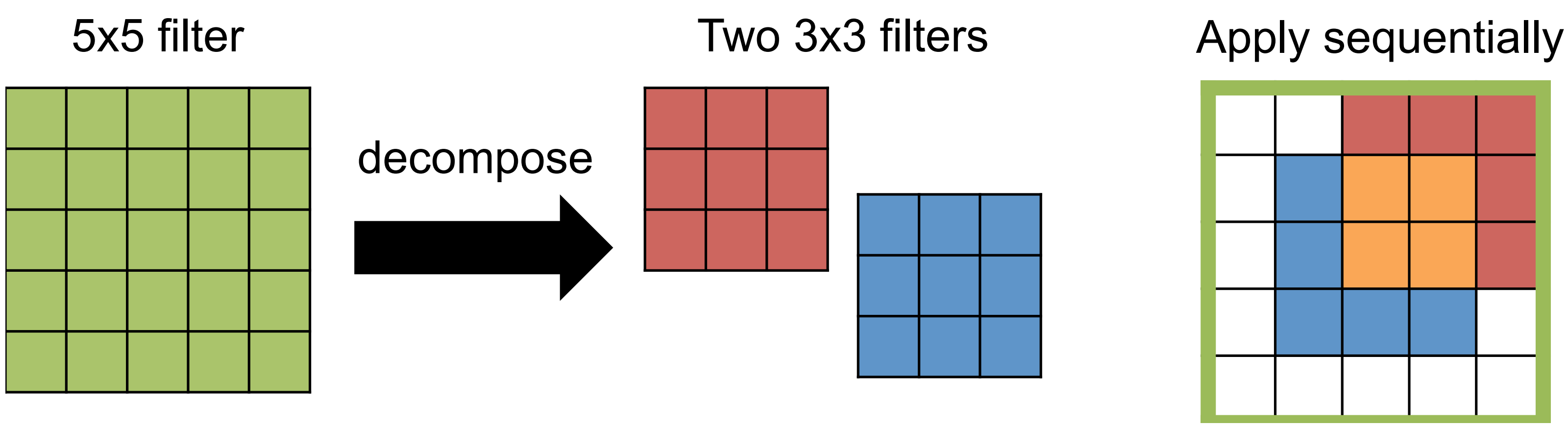
Network Architecture Design

Build Network with series of Small Filters

GoogleNet/Inception v3



VGG-16



Network Architecture Design

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

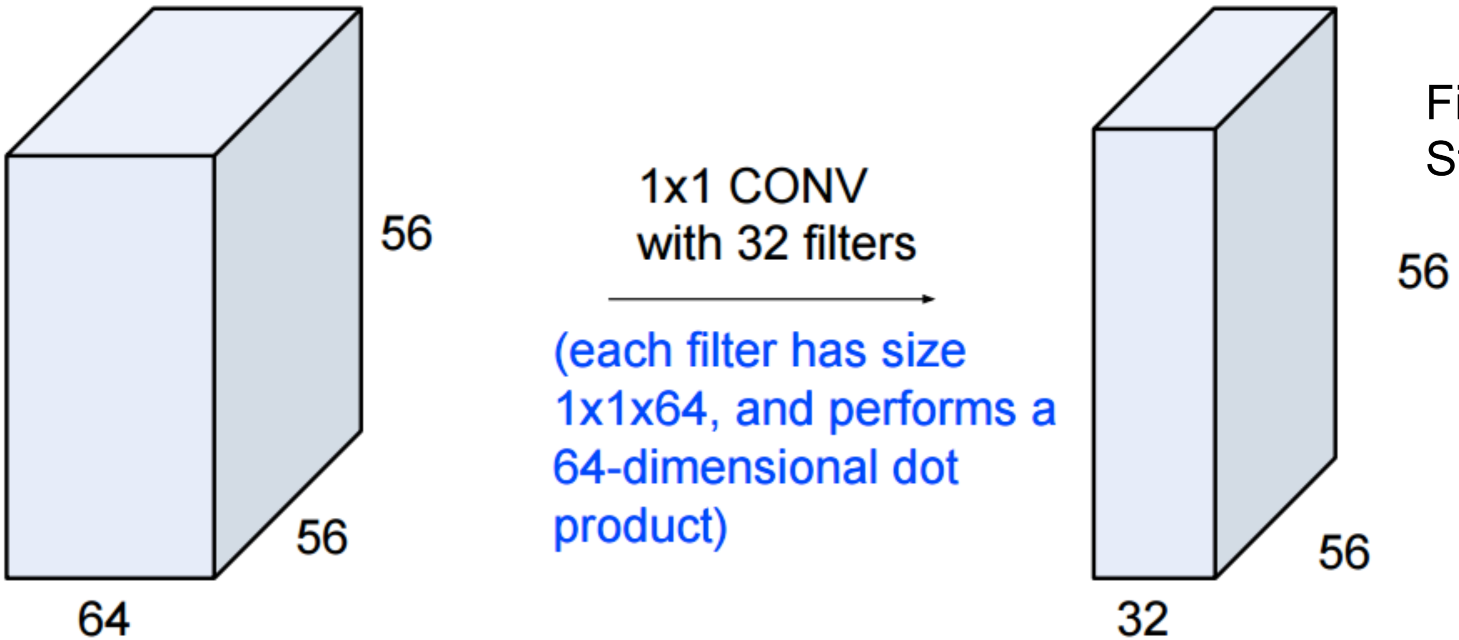


Figure Source:
Stanford cs231n

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**)

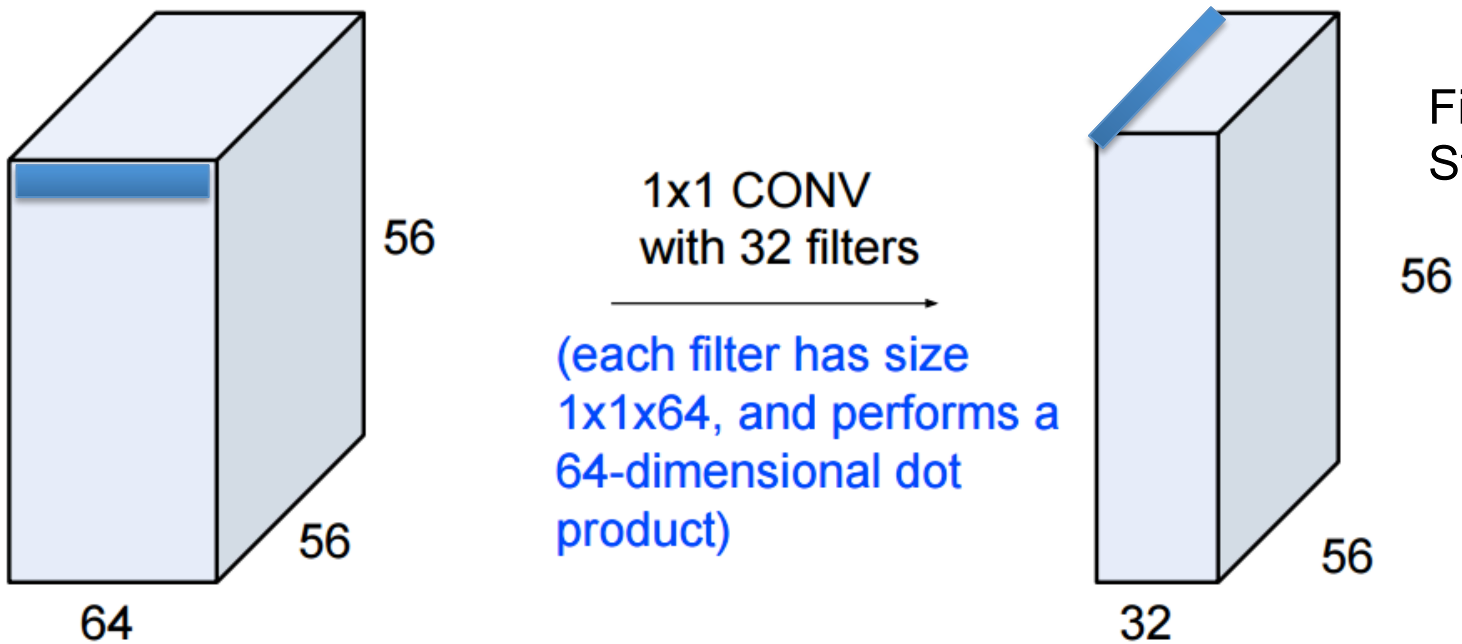


Figure Source:
Stanford cs231n

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**)

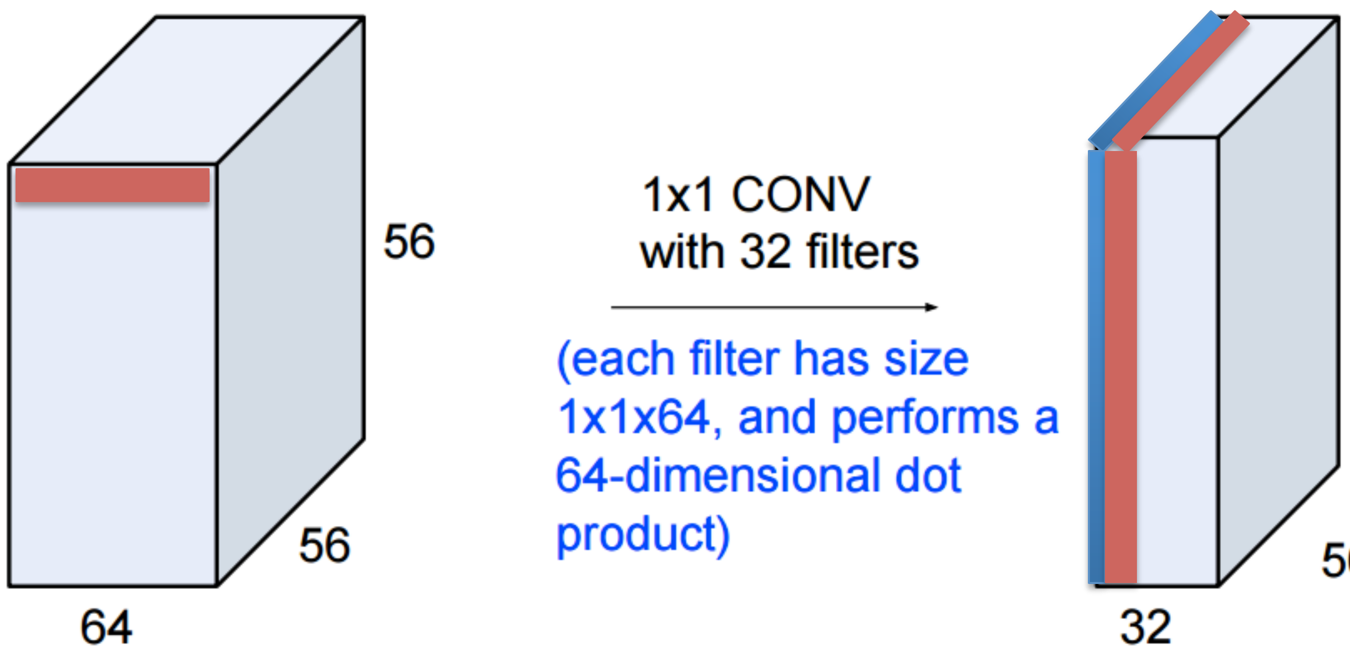


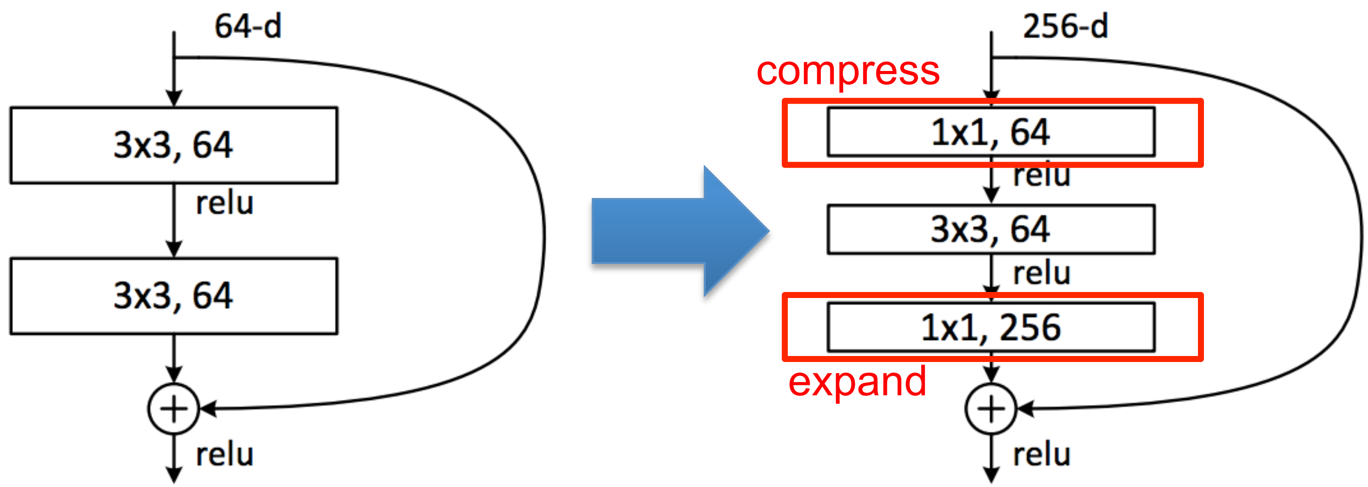
Figure Source:
Stanford cs231n

Used in Network In Network(NiN) and GoogLeNet

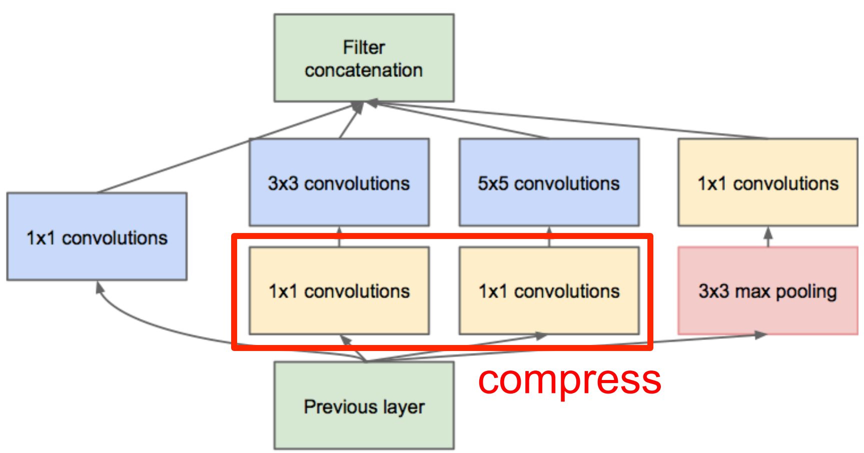
[Lin et al., ArXiV 2013 / ICLR 2014] [Szegedy et al., ArXiV 2014 / CVPR 2015]

Bottleneck in Popular DNN models

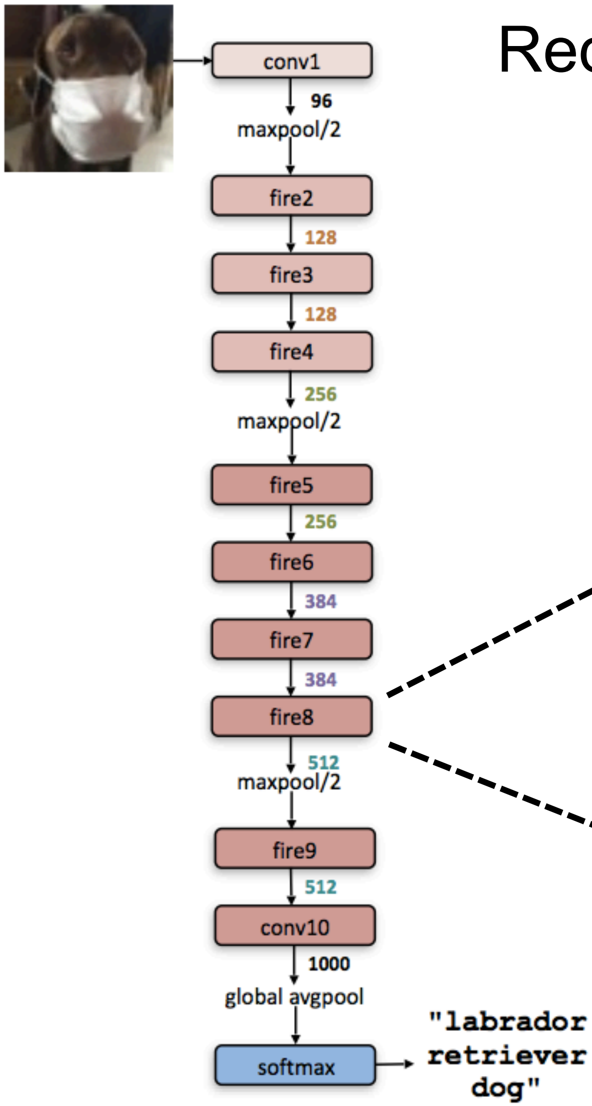
ResNet



GoogleNet



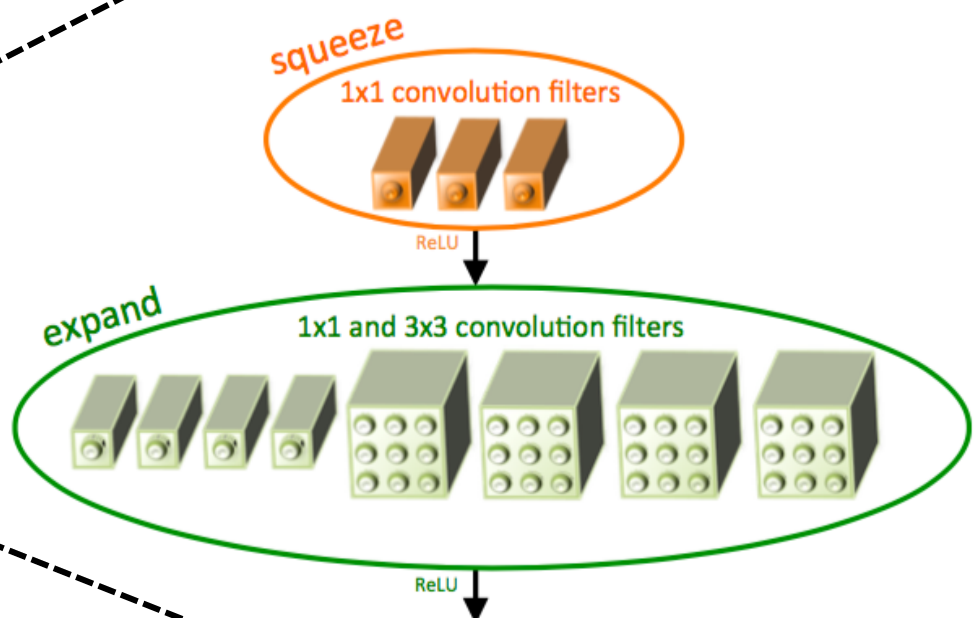
SqueezeNet



Reduce weights by reducing number of input channels by "squeezing" with 1x1

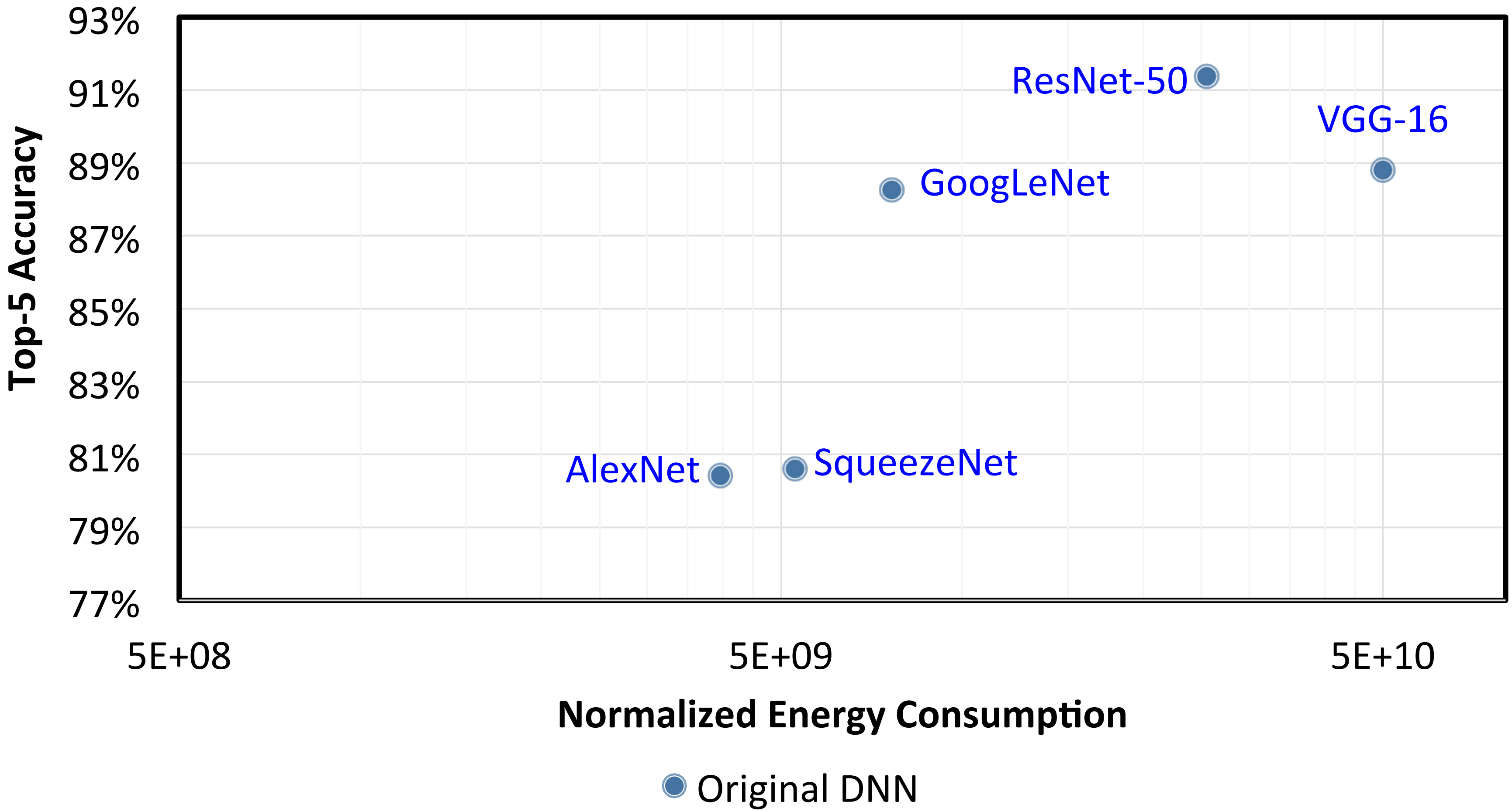
50x fewer weights than AlexNet
(no accuracy loss)

Fire Module



[F.N. Iandola et al., ArXiv, 2016]

Energy Consumption of Existing DNNs

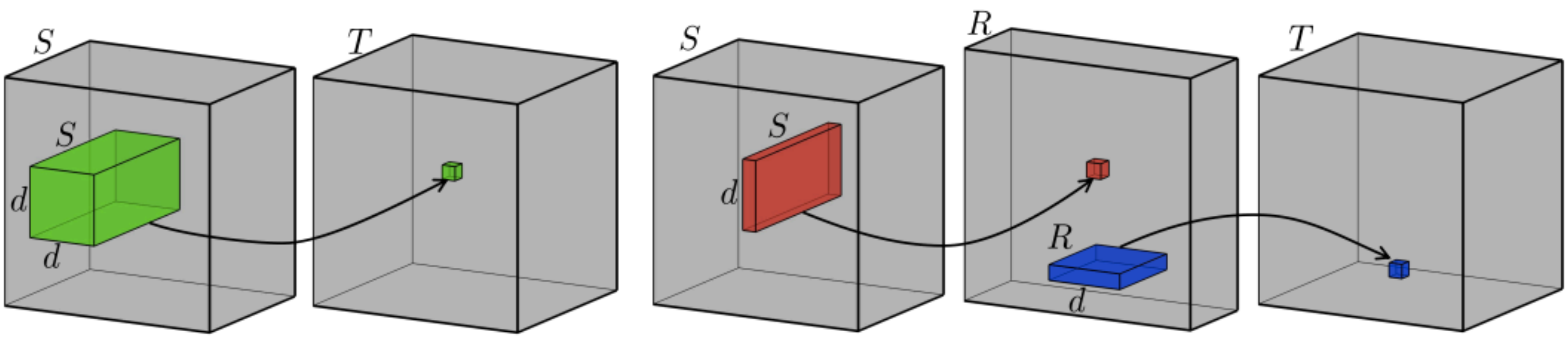


Deeper CNNs with fewer weights do not necessarily consume less energy than shallower CNNs with more weights

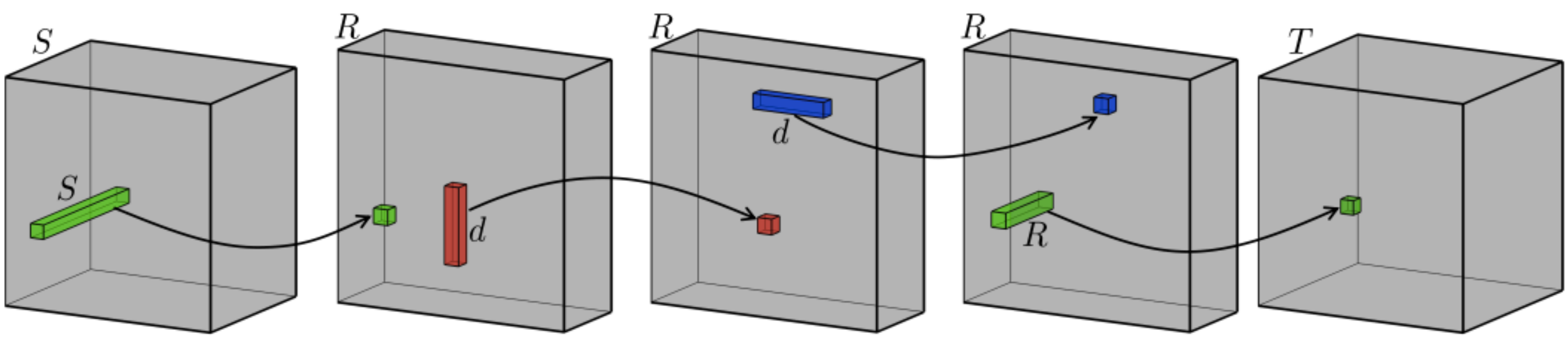
[Yang et al., CVPR 2017]

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)



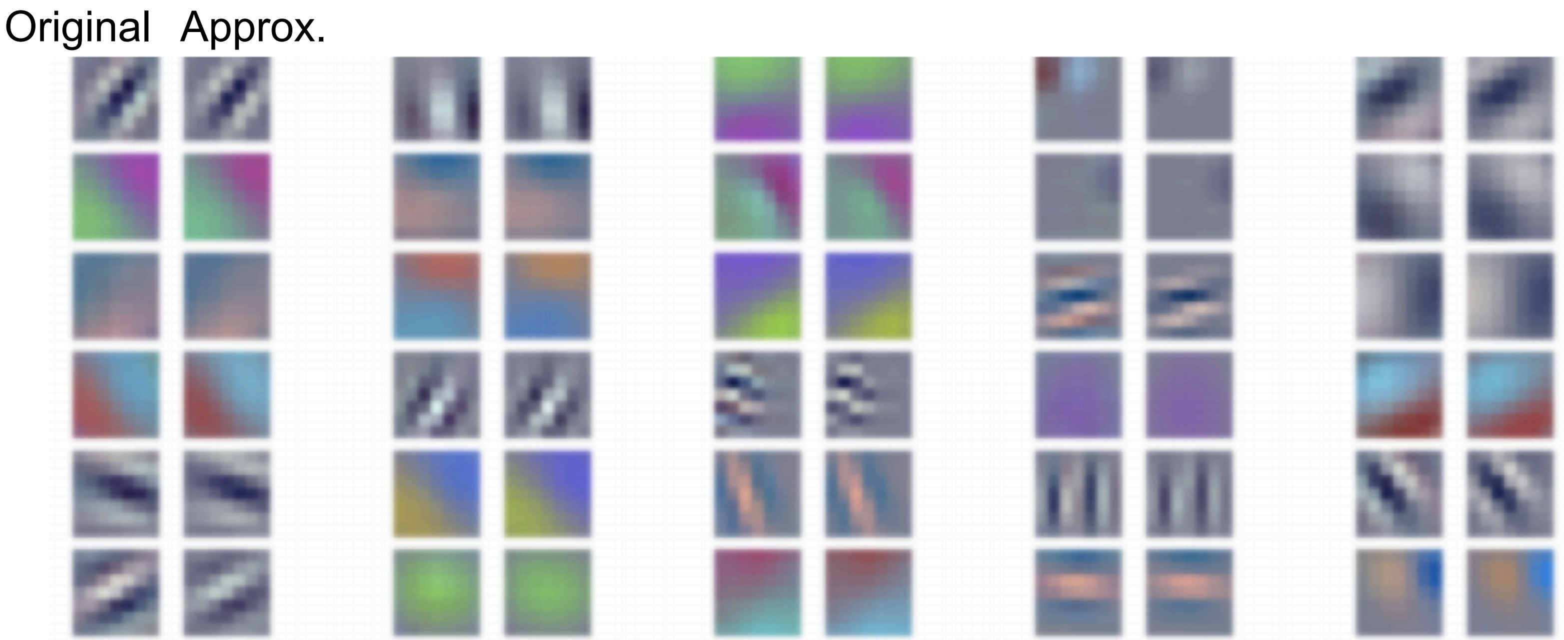
(c) CP-decomposition

R = canonical rank

[Lebedev et al., ICLR 2015]

Decompose Trained Filters

Visualization of Filters

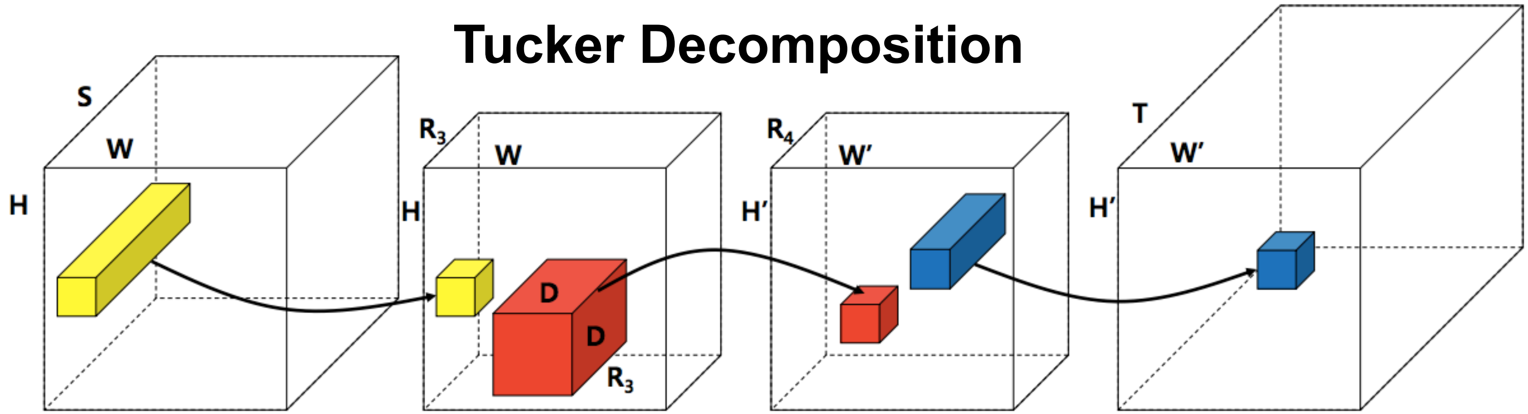


- **Speed up by 1.6 – 2.7x on CPU/GPU for CONV1, CONV2 layers**
- **Reduce size by 5 - 13x for FC layer**
- **< 1% drop in accuracy**

[Denton et al., NIPS 2014]

Decompose Trained Filters on Phone

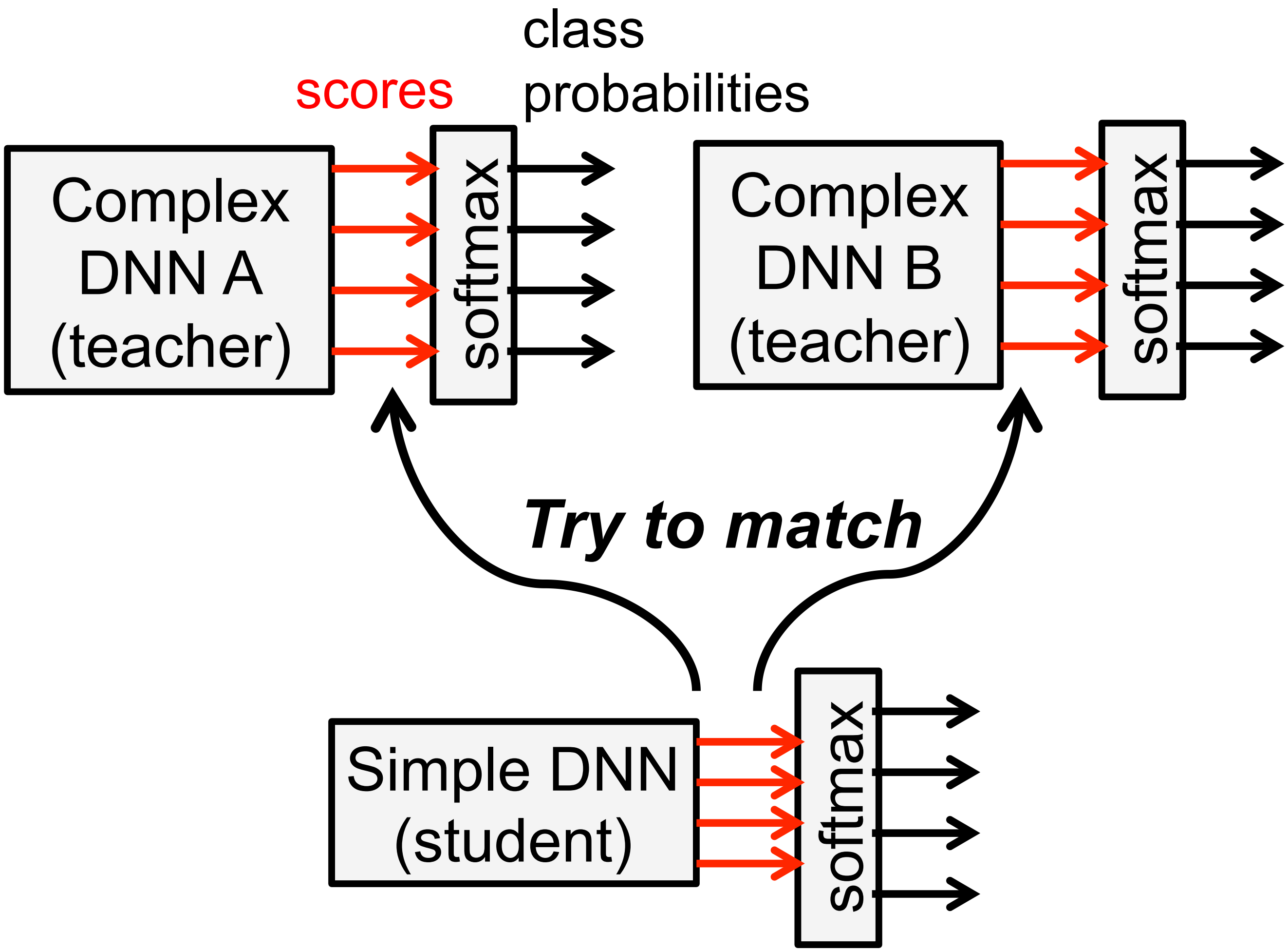
Tucker Decomposition




Model	Top-5	Weights	FLOPs	S6		Titan X
AlexNet	80.03	61M	725M	117ms	245mJ	0.54ms
*AlexNet**	78.33	11M	272M	43ms	72mJ	0.30ms
(imp.)	(-1.70)	(×5.46)	(×2.67)	(×2.72)	(×3.41)	(×1.81)
VGG-S	84.60	103M	2640M	357ms	825mJ	1.86ms
*VGG-S**	84.05	14M	549M	97ms	193mJ	0.92ms
(imp.)	(-0.55)	(×7.40)	(×4.80)	(×3.68)	(×4.26)	(×2.01)
GoogLeNet	88.90	6.9M	1566M	273ms	473mJ	1.83ms
*GoogLeNet**	88.66	4.7M	760M	192ms	296mJ	1.48ms
(imp.)	(-0.24)	(×1.28)	(×2.06)	(×1.42)	(×1.60)	(×1.23)
VGG-16	89.90	138M	15484M	1926ms	4757mJ	10.67ms
*VGG-16**	89.40	127M	3139M	576ms	1346mJ	4.58ms
(imp.)	(-0.50)	(×1.09)	(×4.93)	(×3.34)	(×3.53)	(×2.33)

[Kim et al., ICLR 2016]

Knowledge Distillation



[Bucilu et al., KDD 2006],[Hinton et al., arXiv 2015]