Fast Learning on Slow Hardware

Tor Aamodt (UBC)

Fastpath Workshop, MICRO 2022

Oct 2, 2022

Acknowledgements

Negar Goli

Aamir Raihan

Jonathan Lew

Dave Evans

Also: Wenyi Gong, Yunpeng Liu

NSERC CRSNG

NSERC COHESA

Neural Networks are Expensive to Train

Classification

Detection

Game AI

Self Driving

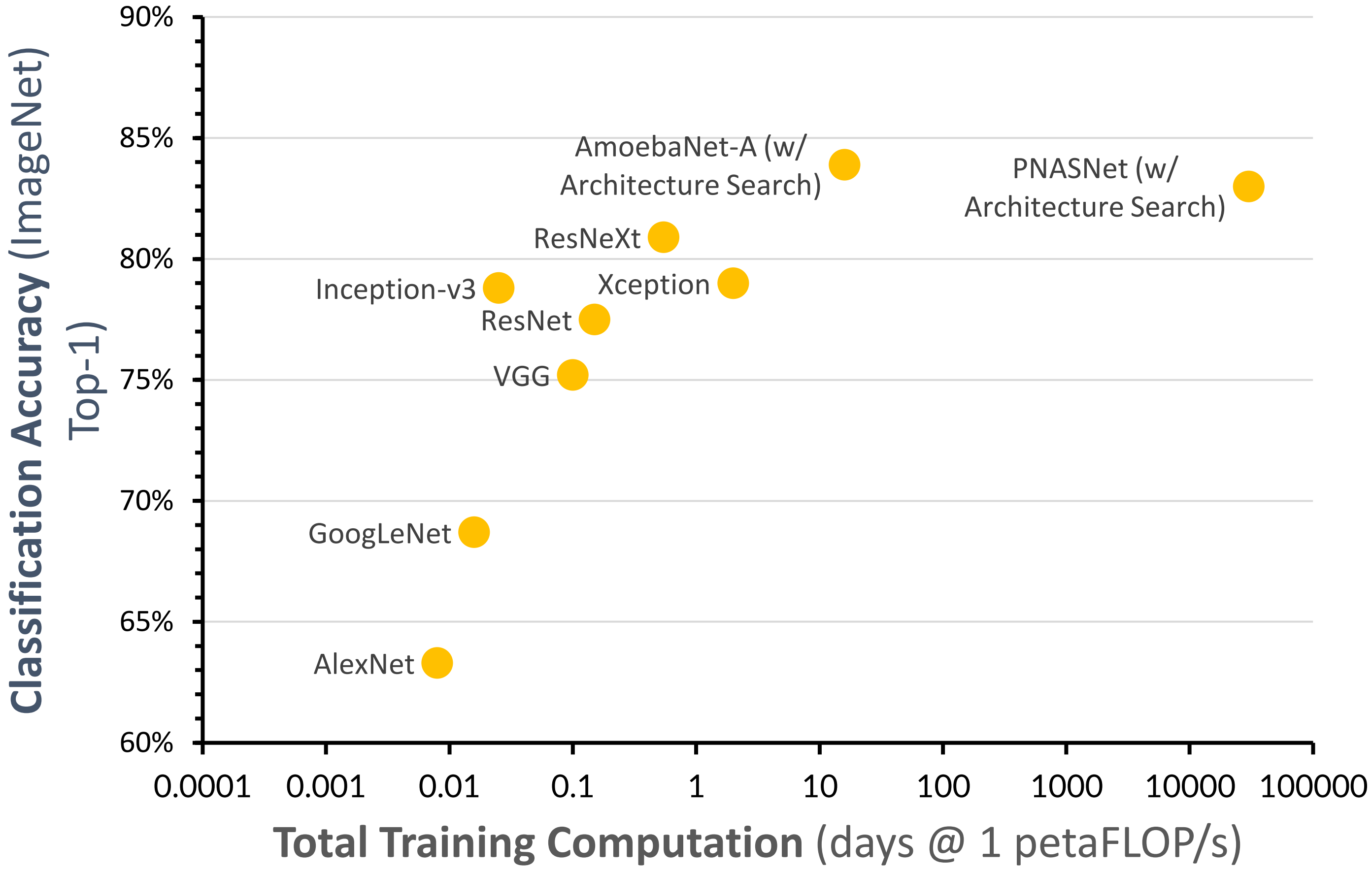
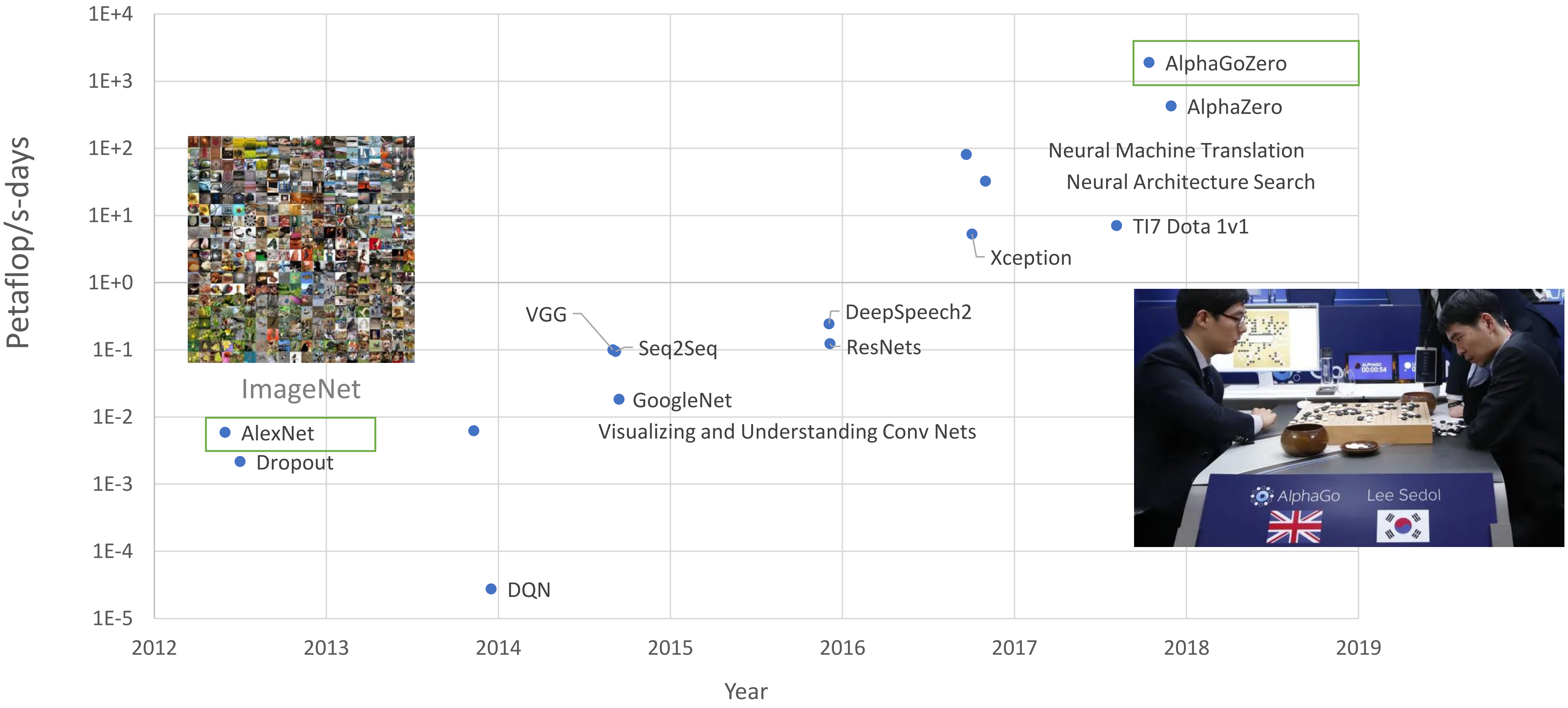


Image sources: https://tinyurl.com/y9bzxzsk https://tinyurl.com/msuc7mav

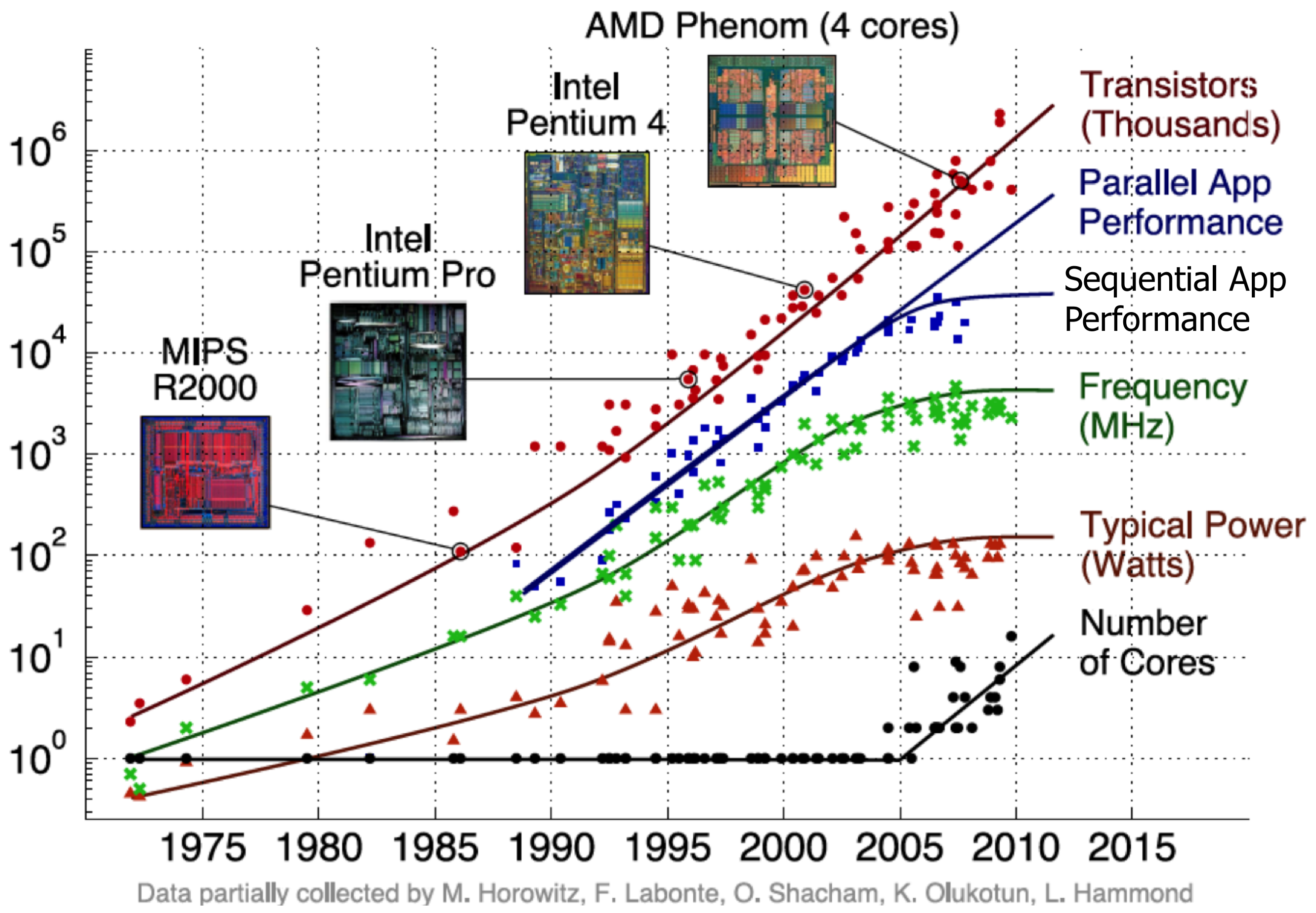
Figure reproduced from openai.com/blog/ai-and-compute/

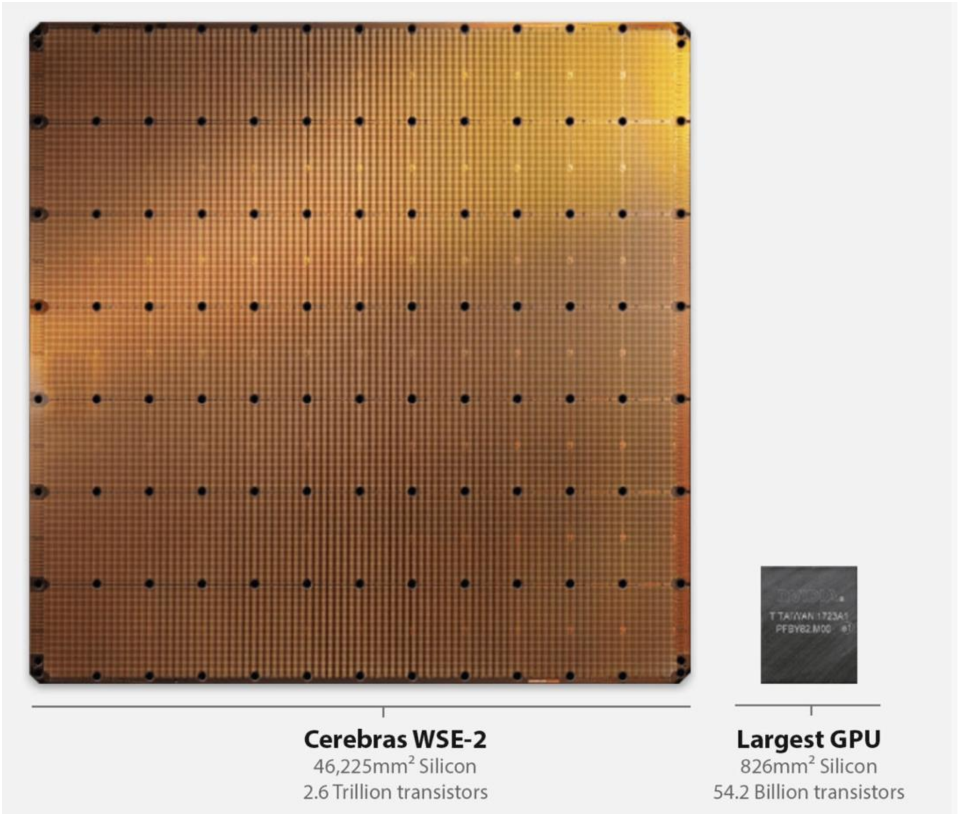
300,000× Increase in Training Compute



Graph reproduced from openai.com/blog/ai-and-compute/

Transistor Scaling will Reach Limits (eventually)



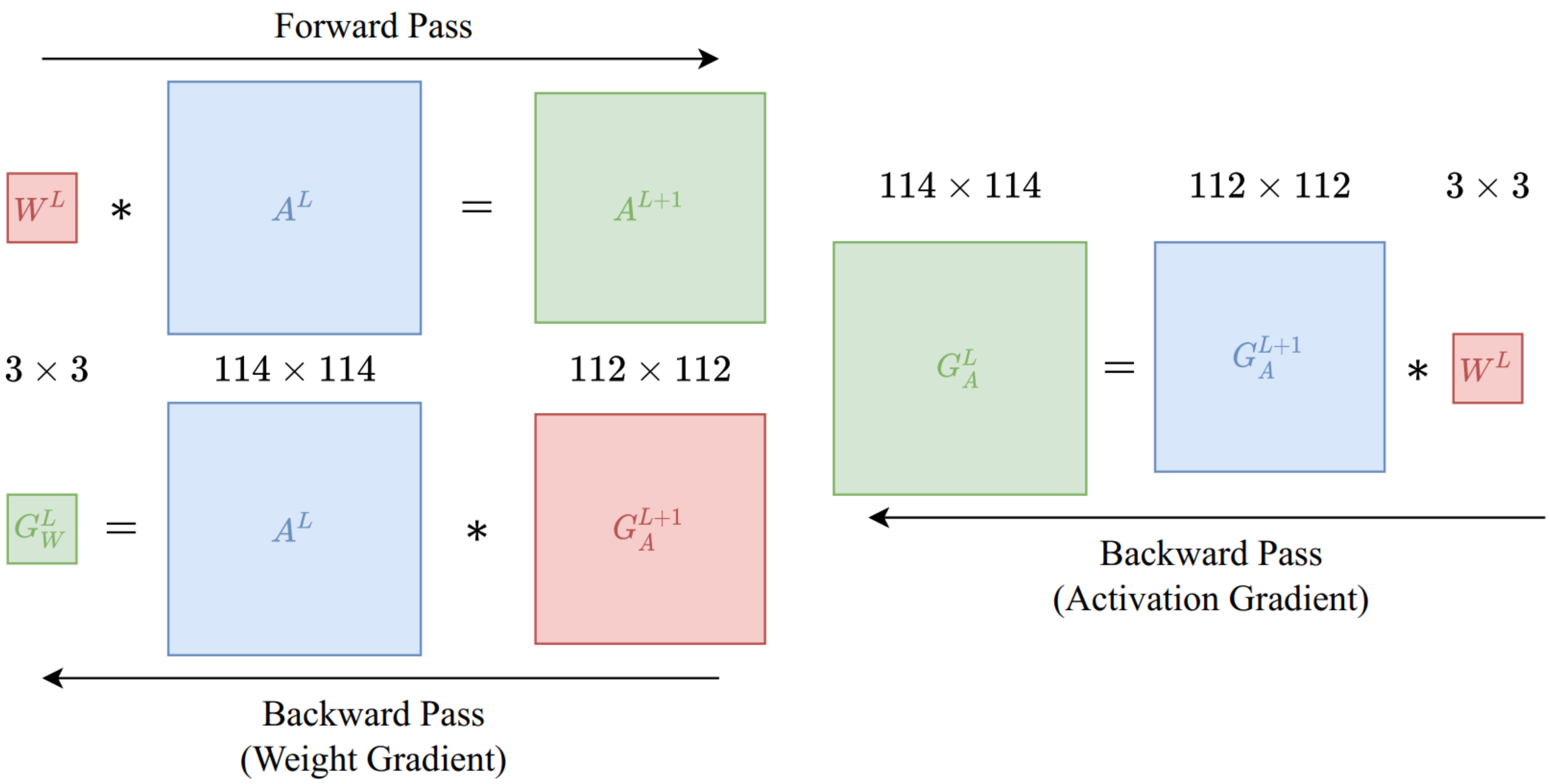


[image: cerebras.ai]

How to improve training speed?

- Exploit data parallelism
  - Each node has complete model, handles subset of all training data, accumulate gradients across nodes (size of model limited by node memory capacity)
- Model parallelism
  - Split model across compute nodes (e.g., AlexNet, Tesla Dojo)
- Reduce computations
  - Reduce number of iterations (e.g., batch normalization, ADAM, etc…)
  - Reduce computation per iteration (e.g., stochastic depth, sparsity)
- ML hardware accelerators
  - Inference, training or both
  - Datacenter: GPU, Google TPU, Cerebras WSE, Graphcore IPU, Huawei DaVinci
  - Edge: Apple Neural Engine, Samsung NPU, Tesla FSD SoC, …

Example: ResNet Convolution



Convolutional Neural Network Training

Activation Gradient (G_A^{L+1})

2	-3
0	0

*

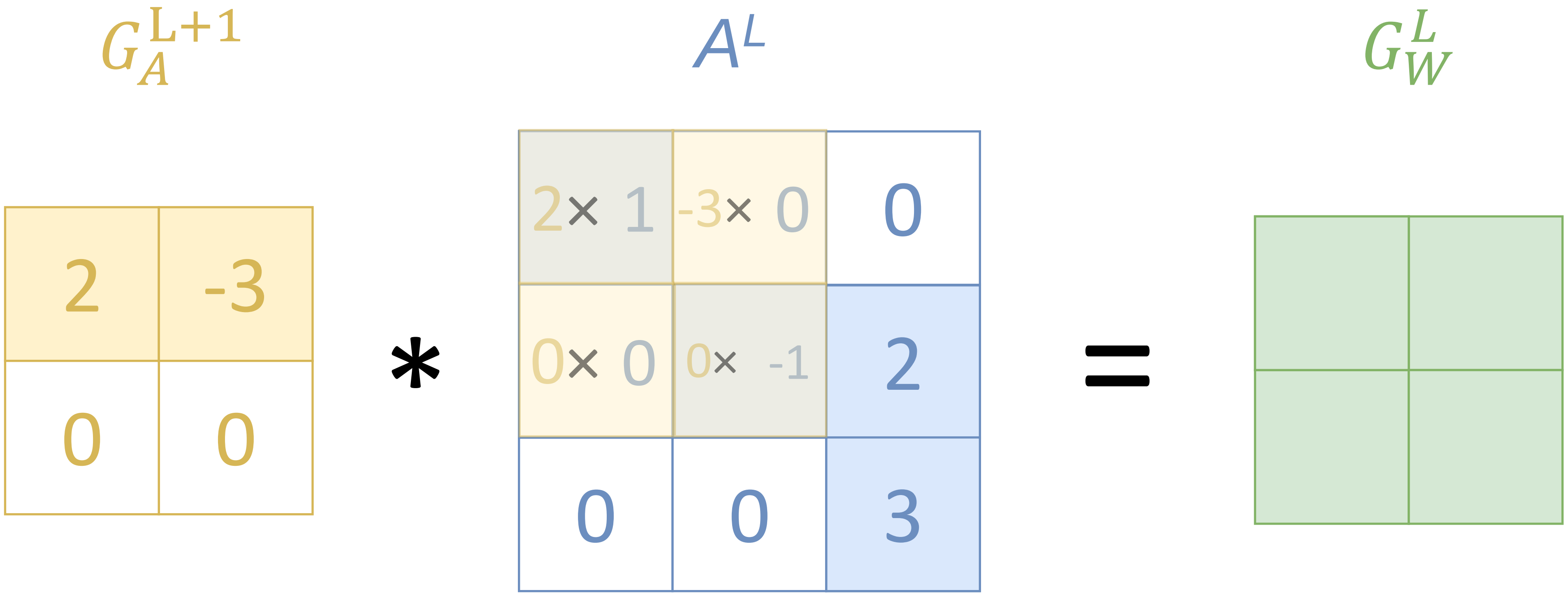
Activation (A^L)

1	0	0
0	-1	2
0	0	3

=

Weight Gradient (G_W^L)

Convolutional Neural Network Training



Convolutional Neural Network Training

$$G_A^{L+1} \qquad A^L \qquad G_W^L$$

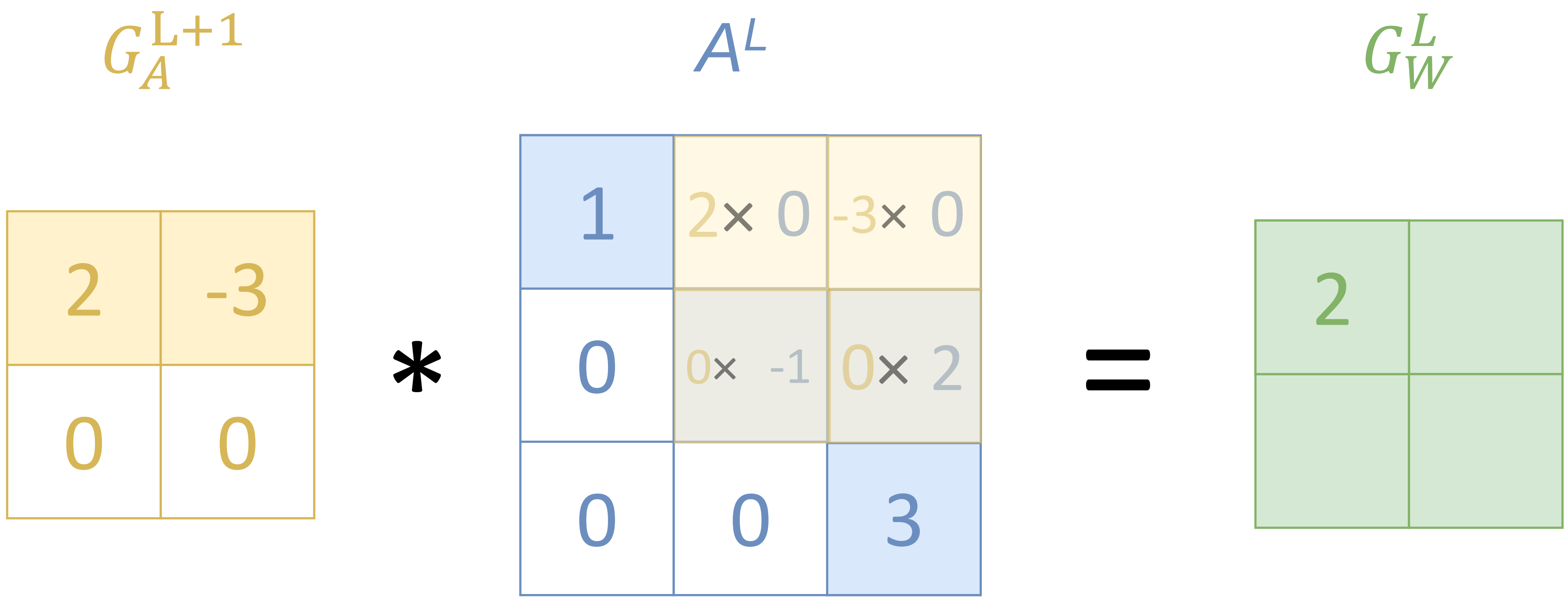
$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & \\ & \end{bmatrix}$$

Convolutional Neural Network Training

$$G_A^{\mathrm{L+1}} * A^L = G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & \\ & \end{bmatrix}$$

Convolutional Neural Network Training



Convolutional Neural Network Training

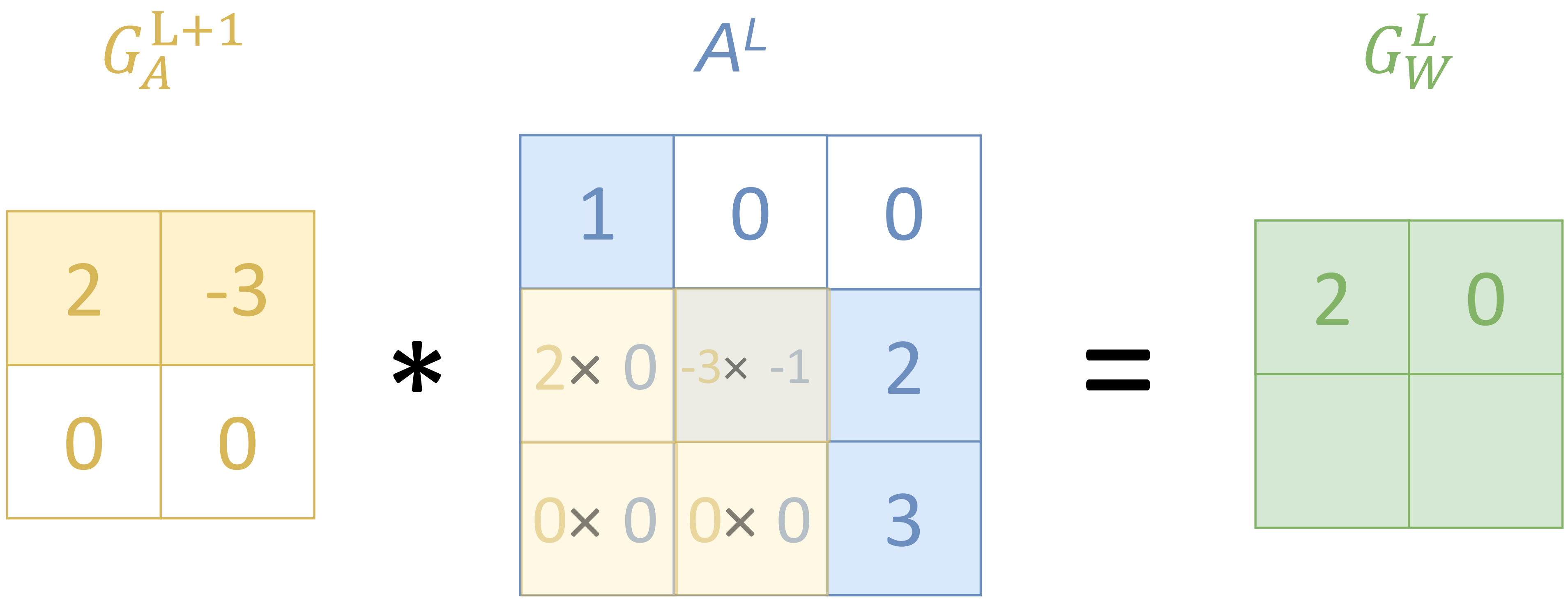
$$G_A^{L+1} \quad * \quad A^L \quad = \quad G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ & \end{bmatrix}$$

Convolutional Neural Network Training

$$\underset{G_A^{L+1}}{\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix}} * \underset{A^L}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix}} = \underset{G_W^L}{\begin{bmatrix} 2 & 0 \\ \; & \; \end{bmatrix}}$$

Convolutional Neural Network Training



Convolutional Neural Network Training

$$\underset{G_A^{L+1}}{\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix}} * \underset{A^L}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix}} = \underset{G_W^L}{\begin{bmatrix} 2 & 0 \\ 3 & \end{bmatrix}}$$

Convolutional Neural Network Training

$$G_A^{L+1} * A^L = G_W^L$$

G_A^{L+1}

2	-3
0	0

*

A^L

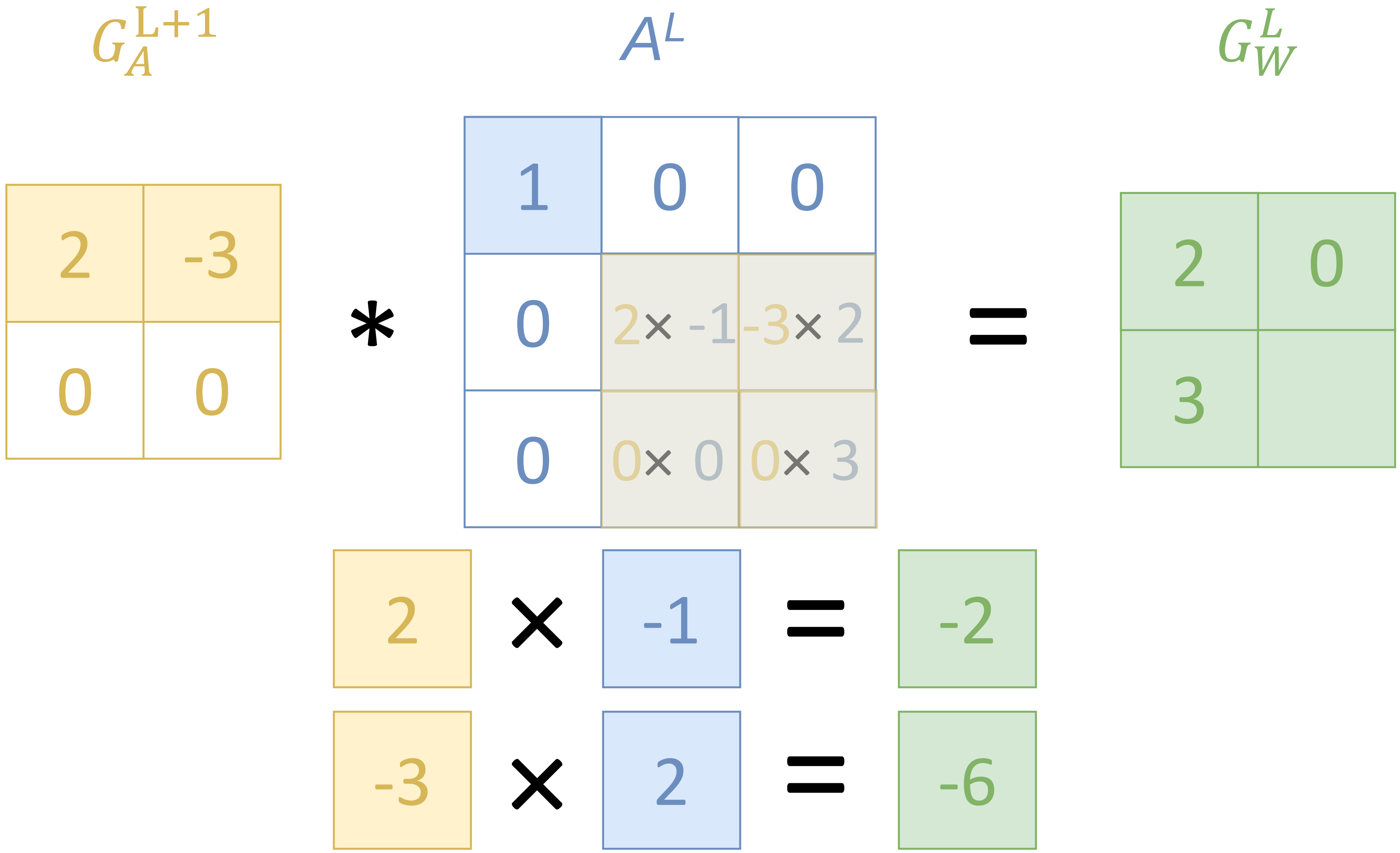
1	0	0
0	-1	2
0	0	3

=

G_W^L

2	0
3	

Convolutional Neural Network Training



Convolutional Neural Network Training

$$\underset{G_A^{\mathrm{L+1}}}{\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix}} * \underset{A^L}{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix}} = \underset{G_W^L}{\begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}}$$

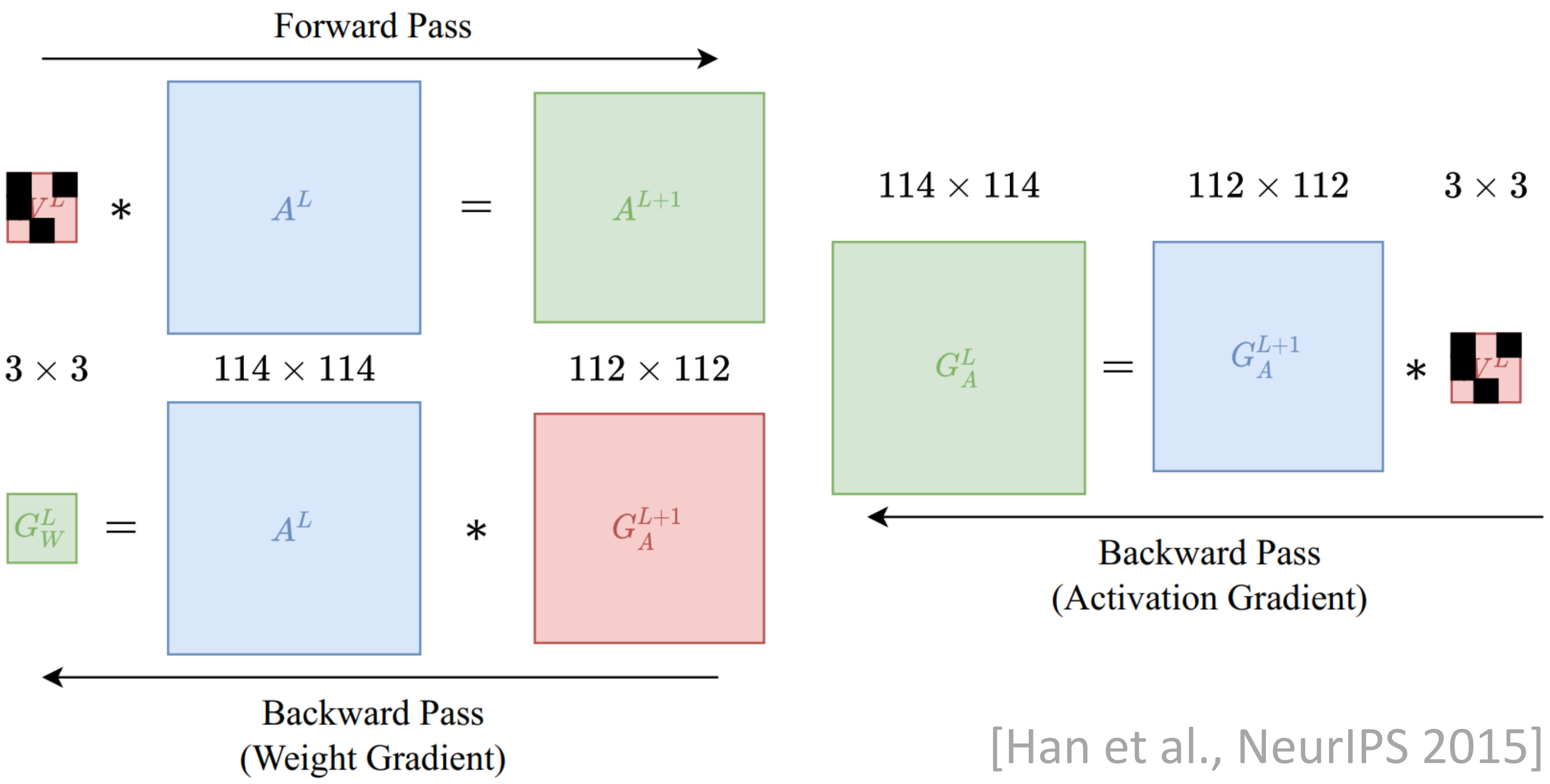
Encouraging sparsity

- Convolutions essentially perform many dot-products:

$$y = [10\ 0\ 3\ 2]\ [1\ 5\ 0\ 0]^T = 10*1 + 0*5 + 3*0 + 2*0 = 10*1 = 10$$

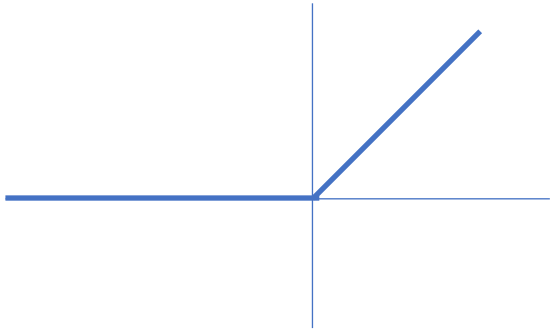
- Multiplications by zero can be skipped.
- Exploiting this can reduce computations and/or model size.
- Much work on this in past ~5 years. Extensions to fully connected layers, RNNs, transformers, etc…

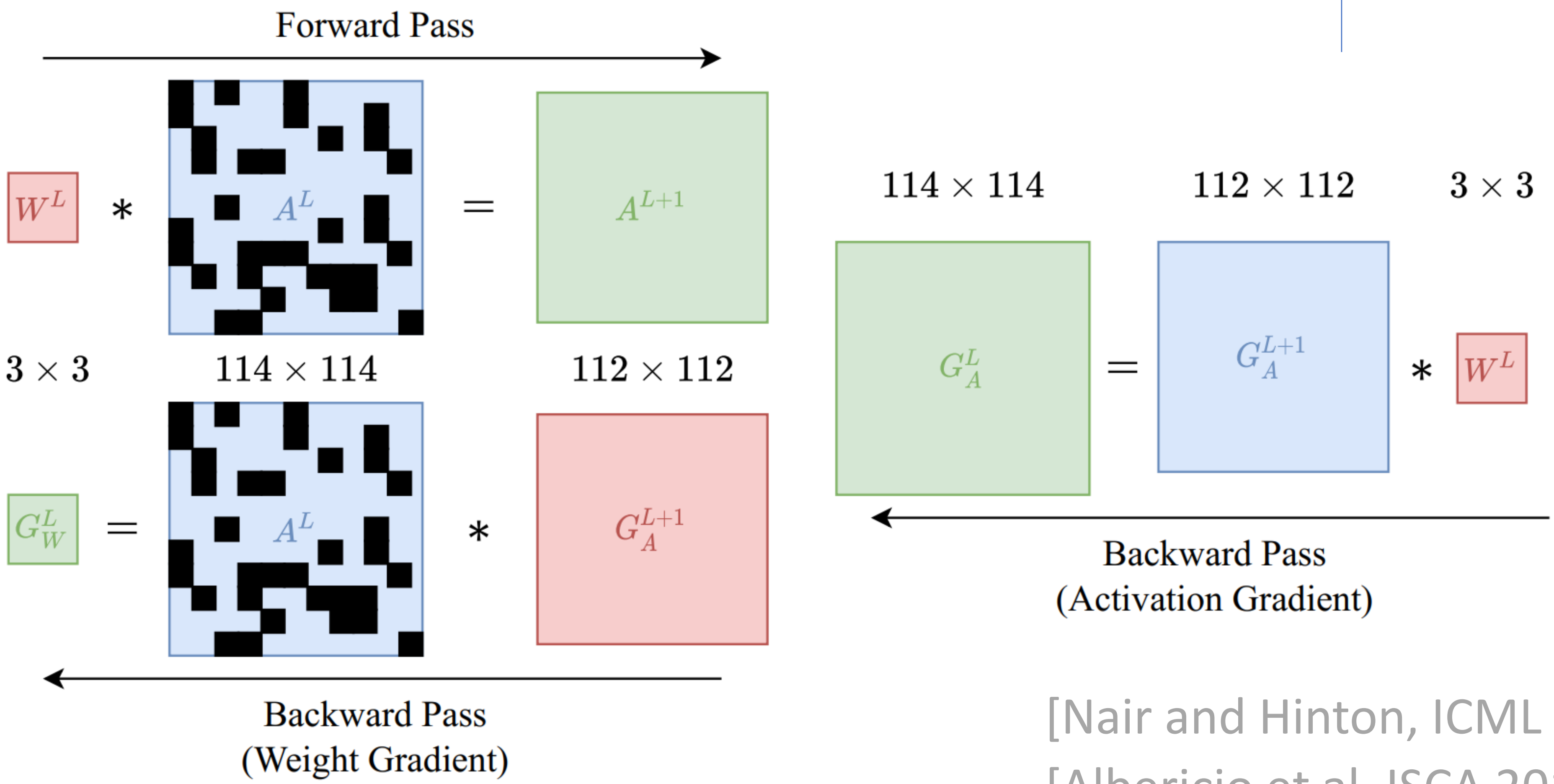
Weight Pruning



[Han et al., NeurIPS 2015]

ReLU

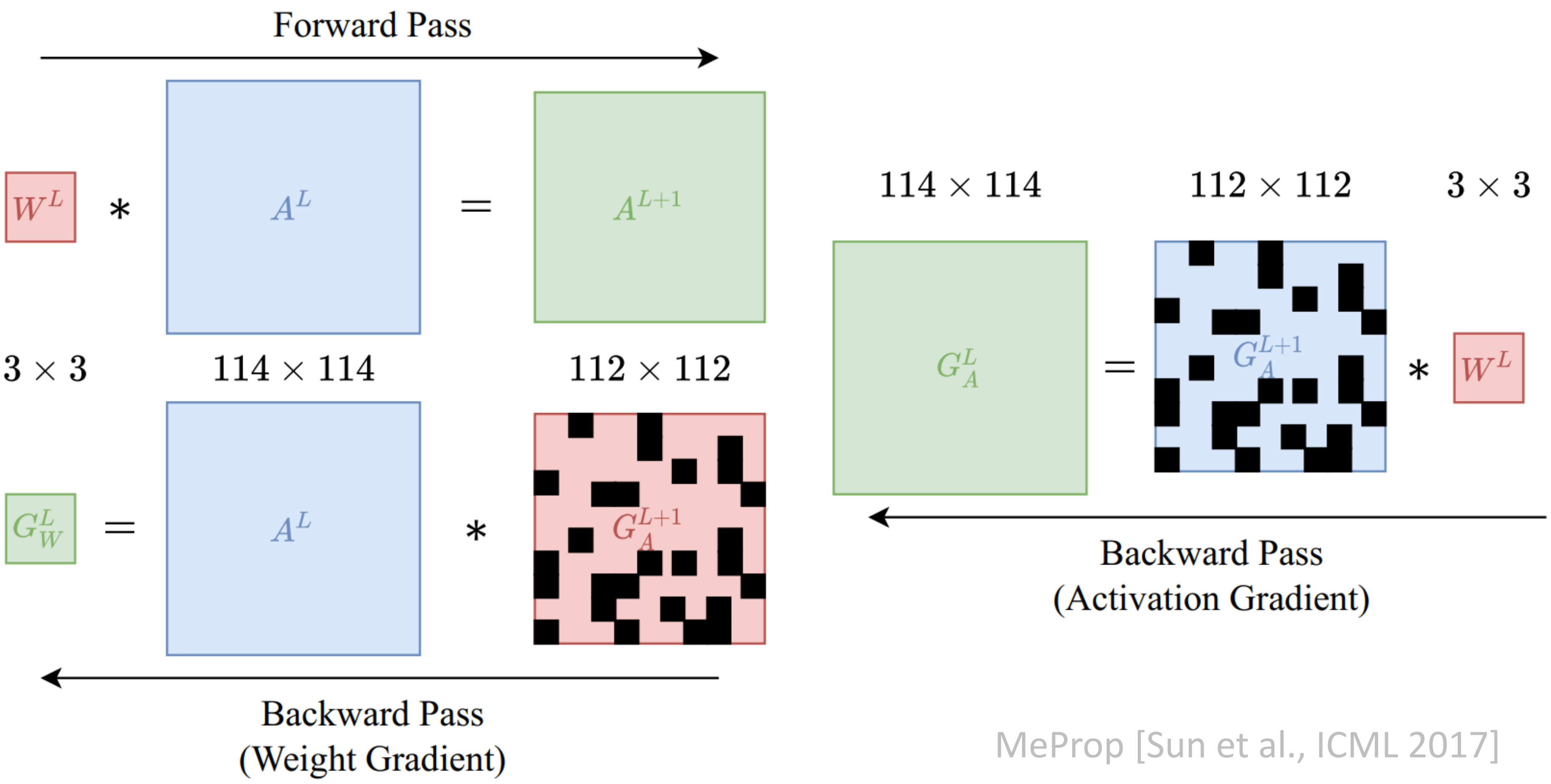




[Nair and Hinton, ICML 2010]

[Albericio et al. ISCA 2016]

Sparse Gradients

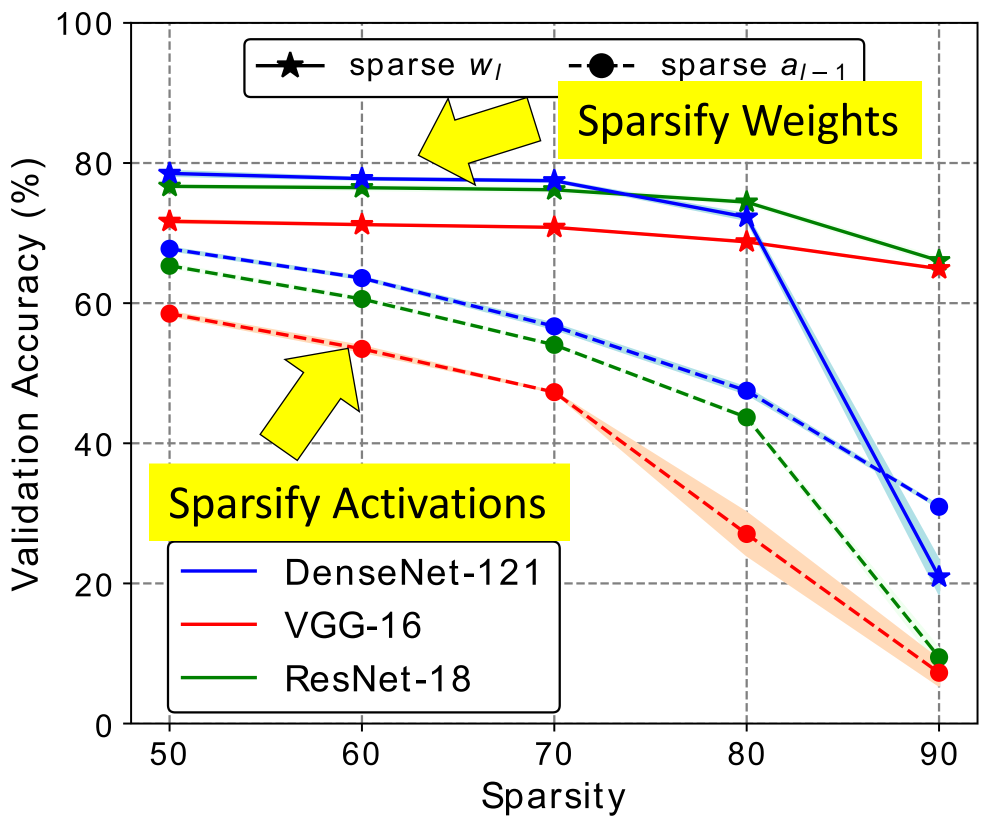


MeProp [Sun et al., ICML 2017]

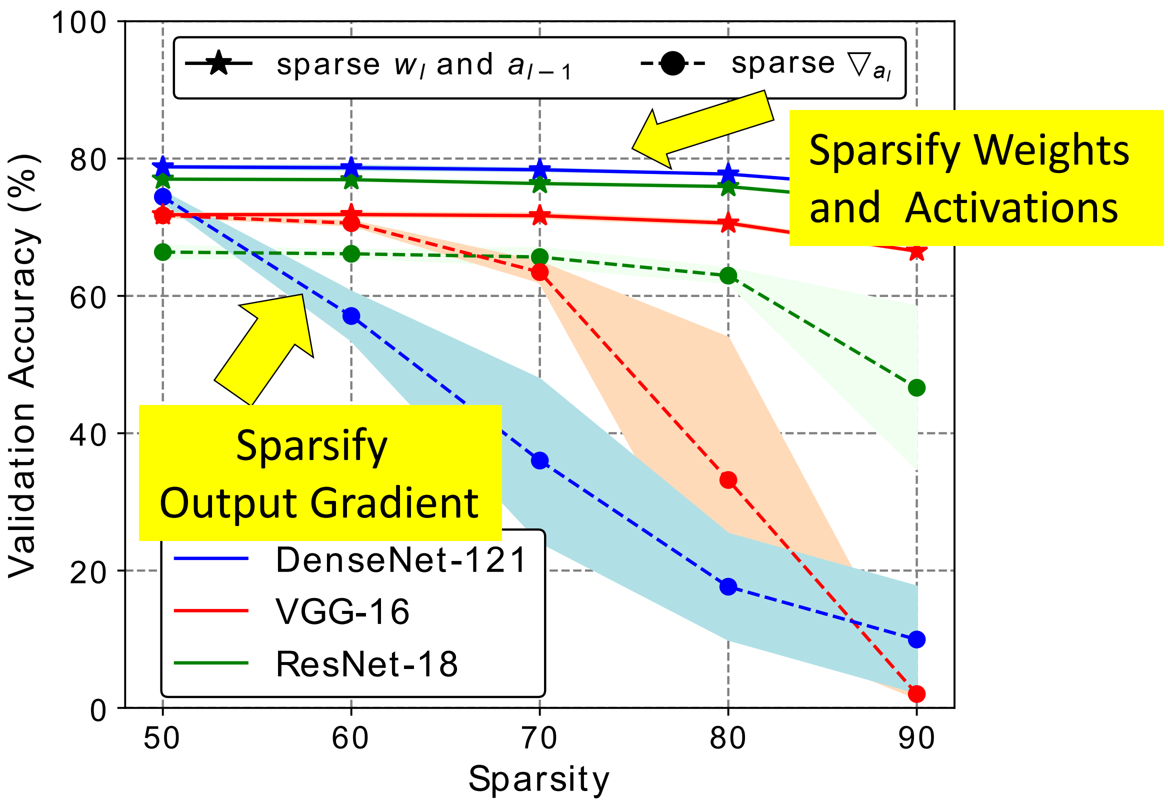
ReSprop [Goli and Aamodt, CVPR 2020]

Sparse Weight Activation Training (SWAT)

FORWARD PASS



BACKWARD PASS



Implication: In forward pass sparsify weights (but not activations).

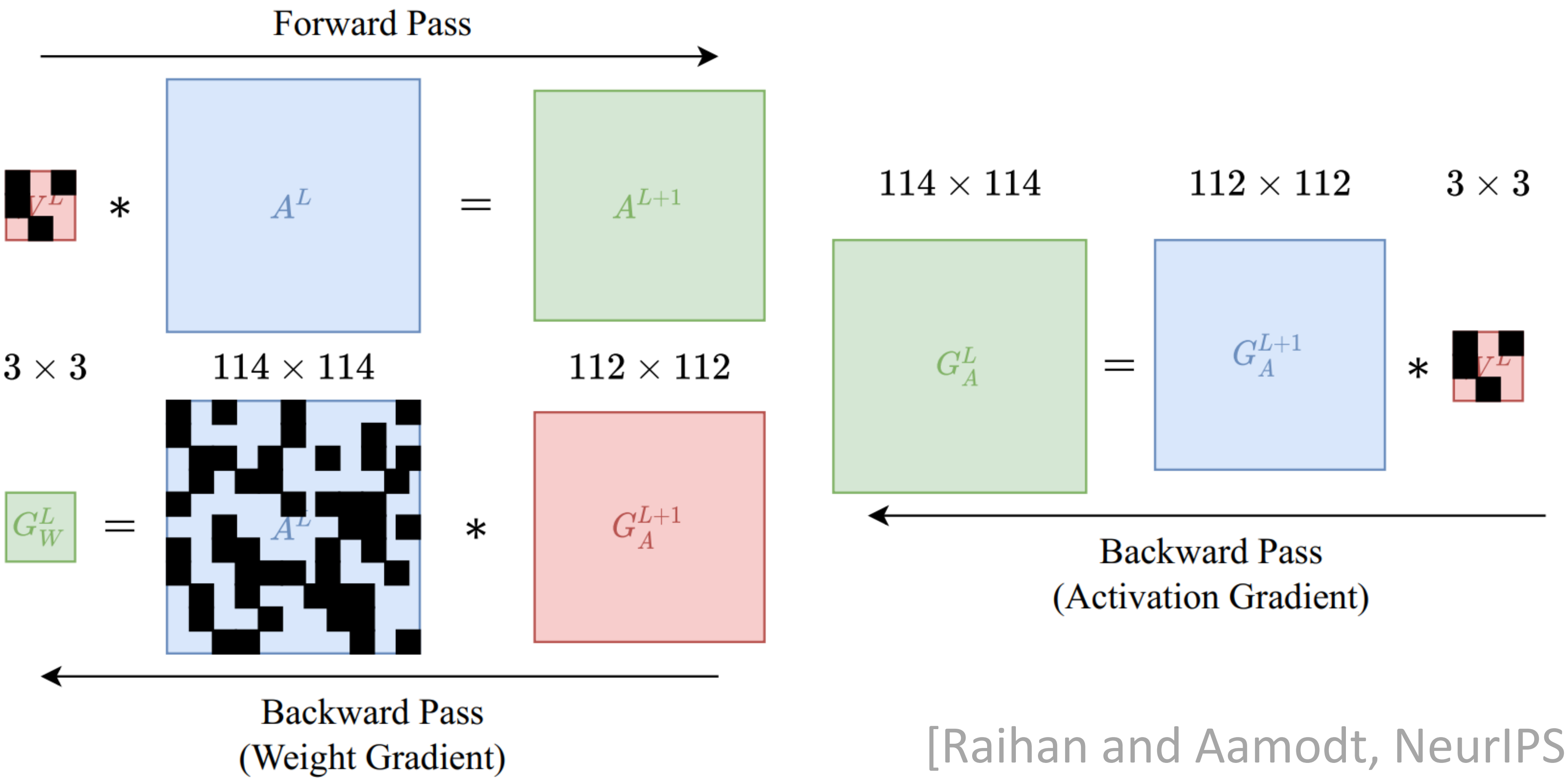
Implication: In backward pass sparsify weights and activations (but not gradients)

SWAT Algorithm (highlights)

- During each training iteration:
  - Sparse weight topology (top-k) induced, which partitions weights into **active** and **non-active** sets.
  - Forward pass:
    - use active (sparse) weights and full (dense) activations to compute layer outputs
  - Backward pass:
    - use active (sparse) weights and full (dense) gradients to compute activation gradients
    - use **sparse** activations (top-k) and dense gradients to compute **dense** weight updates

- Updating weights with dense weight gradients enables topology search (avoids “lottery ticket” problem)

Sparse Weight Activation Training (SWAT)

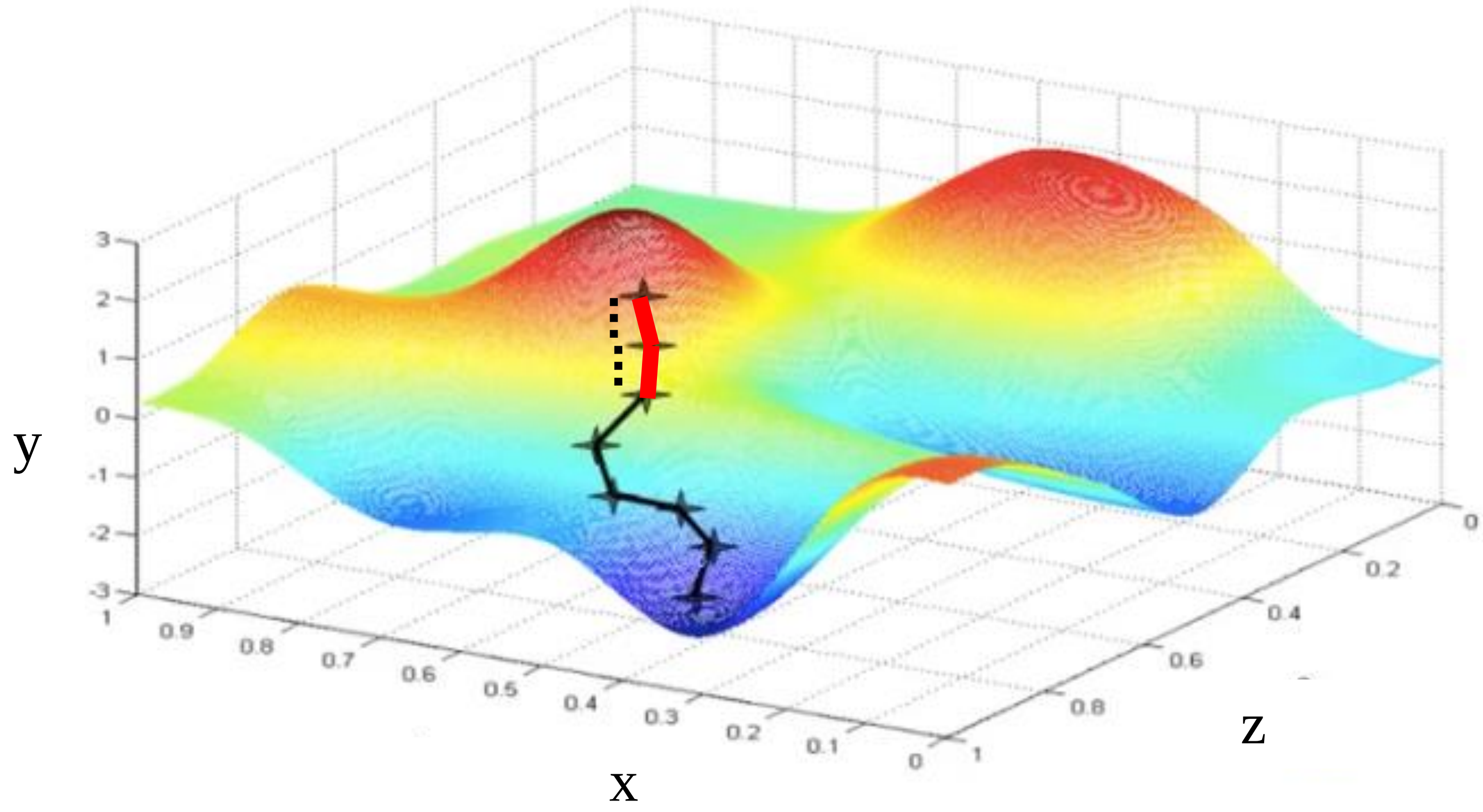


[Raihan and Aamodt, NeurIPS 2020]

Comparison of unstructured SWAT with sparse learning algorithms on the ImageNet

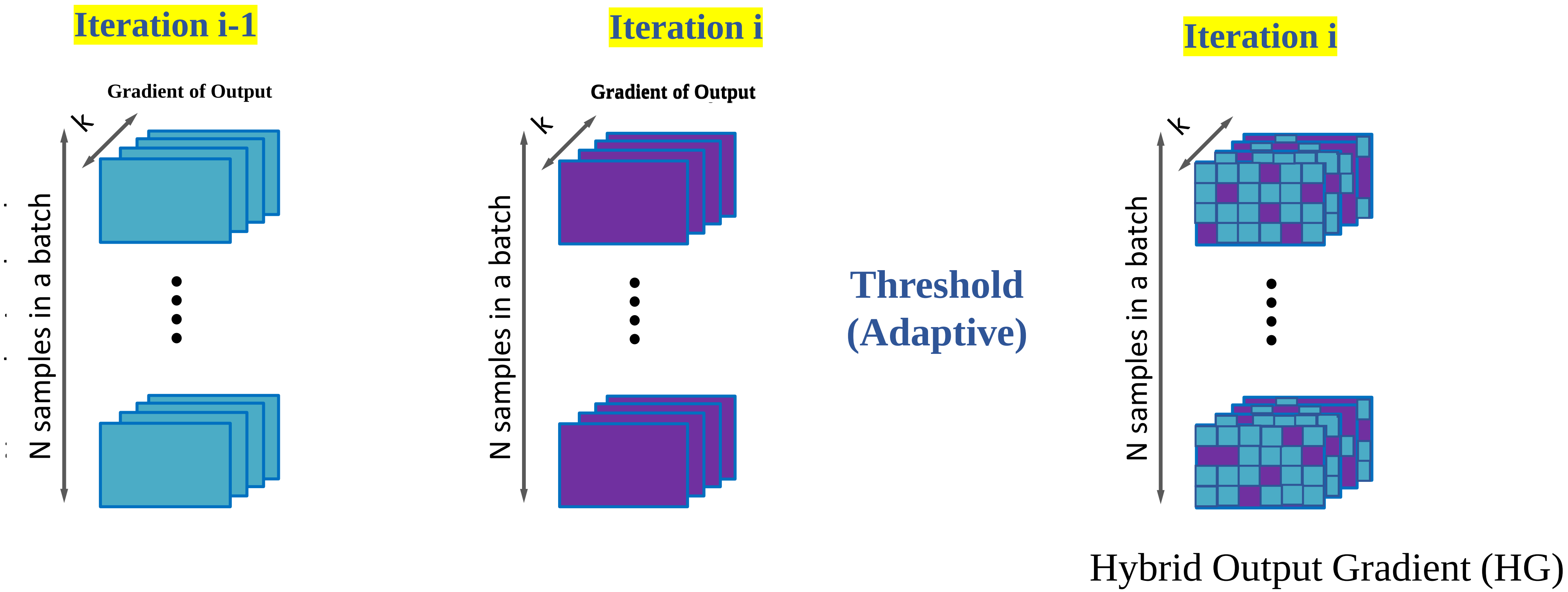
Methods	Weight Sparsity (%)	Activation Sparsity (%)	Top-1 Accuracy (%)	Accuracy Change (%)	Training FLOP ↓ (%)	Inference FLOP ↓ (%)	Model Compression(x)
SET	80	-	73.4	-3.4	58.1	73.0	3.4
	90	-	71.3	-5.5	63.8	82.1	5.0
DSR	80	-	74.1	-2.7	51.6	59.4	3.4
	90	-	71.9	-4.9	58.9	70.7	5.0
SNFS	80	-	74.9	-2.1	45.8	43.3	5.0
	90	-	72.9	-4.1	57.6	59.7	10.0
RigL	80	-	74.6	-2.2	67.2	80.0	5.0
	90	-	72.0	-4.8	74.1	90.0	10.0
DST	80.4	-	74.0	-2.8	67.1	84.9	5.0
	90.1	-	72.8	-4.0	75.8	91.3	10.0
SWAT-U	80	80	75.2	-1.6	76.1	77.7	5.0
	90	90	72.1	-4.7	85.6	87.4	10.0
SWAT-U	80	80	75.2	-1.6	76.1	77.7	5.0
	90	90	72.1	-4.7	85.6	87.4	10.0
SWAT-U	80	80	75.2	-1.6	76.1	77.7	5.0
	90	90	72.1	-4.7	85.6	87.4	10.0

ReSprop: Reuse Sparsified Backpropagation



[Goli and Aamodt, CVPR 2020]

ReSprop


Iteration i-1
Gradient of Output
k
N samples in a batch
Iteration i
Gradient of Output
k
N samples in a batch
Threshold
(Adaptive)
Iteration i
k
N samples in a batch
Hybrid Output Gradient (HG)

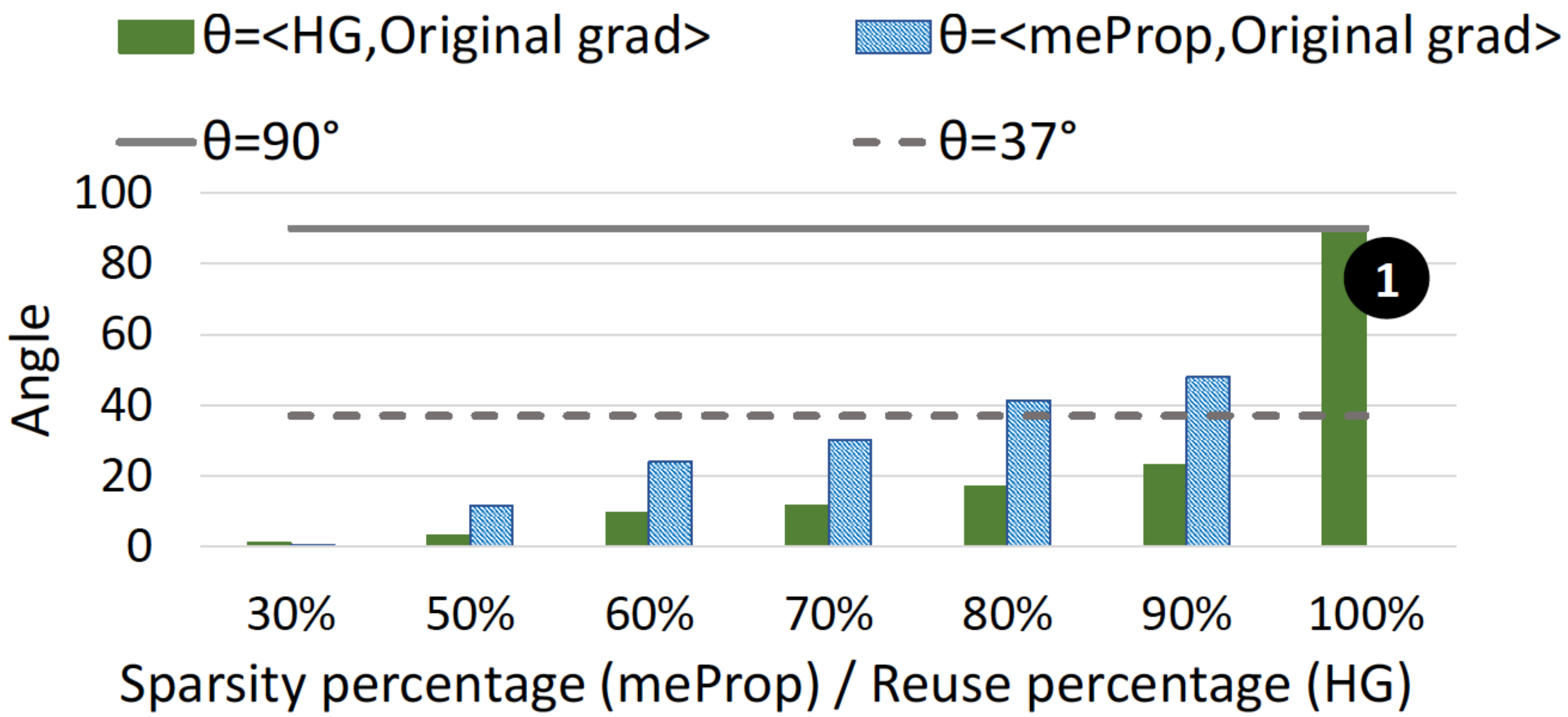
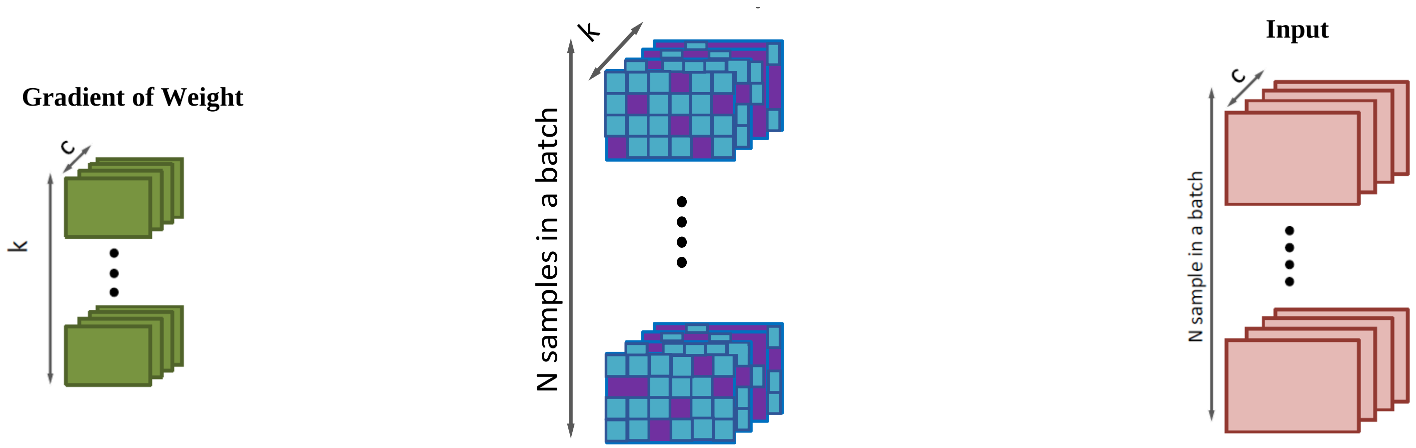
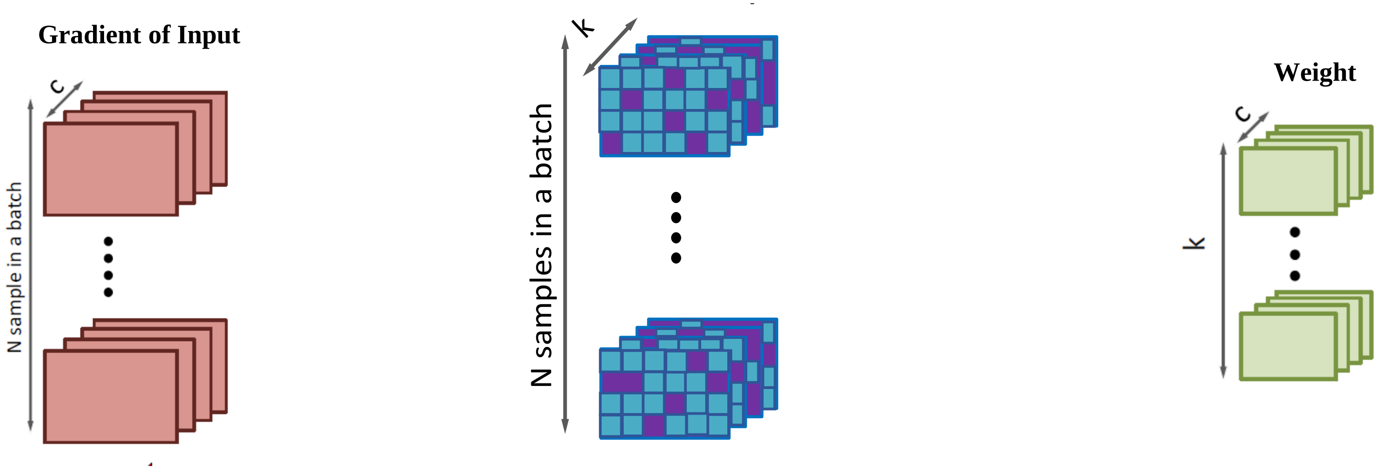


Figure 2. HG and meProp angles for different reuse percentages and sparsities, respectively. The angle is calculated by finding the average angle of all layers while training ResNet-18 on CIFAR-10 for 100 iterations (batch size=128).

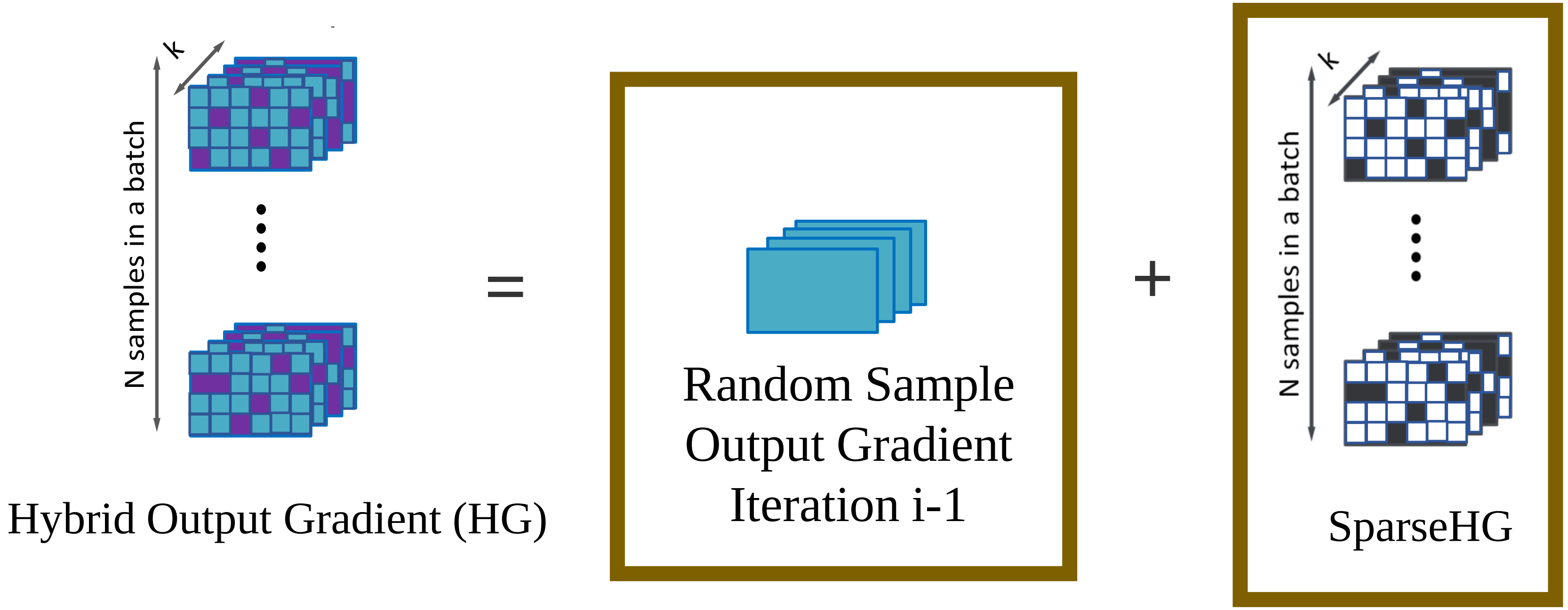


Gradient of Weight = Hybrid Output Gradient ⊗ Input



Gradient of Input = Hybrid Output Gradient ⊗ Weight

Iteration i:



Adaptive Threshold

1) Reuse

2) Use sparse gradients for expensive convolutions in backward pass

Fast

RS	Algorithm	ResNet34	WRN-50-2	VGG16
50%	ReSprop	73.08	78.69	70.09
	W-ReSprop	73.21	78.81	70.41
70%	ReSprop	67.12	73.34	68.73
	W-ReSprop	72.73	78.25	70.01
90%	ReSprop	63.78	67.72	60.76
	W-ReSprop	**72.44**	**77.93**	**69.46**
Baseline		73.34	78.88	70.50

Table 4. Top 1 validation accuracy of ReSprop and W-ReSprop algorithms at different reuse-sparsity constraints on the ImageNet dataset.

Hardware Support for Sparsity

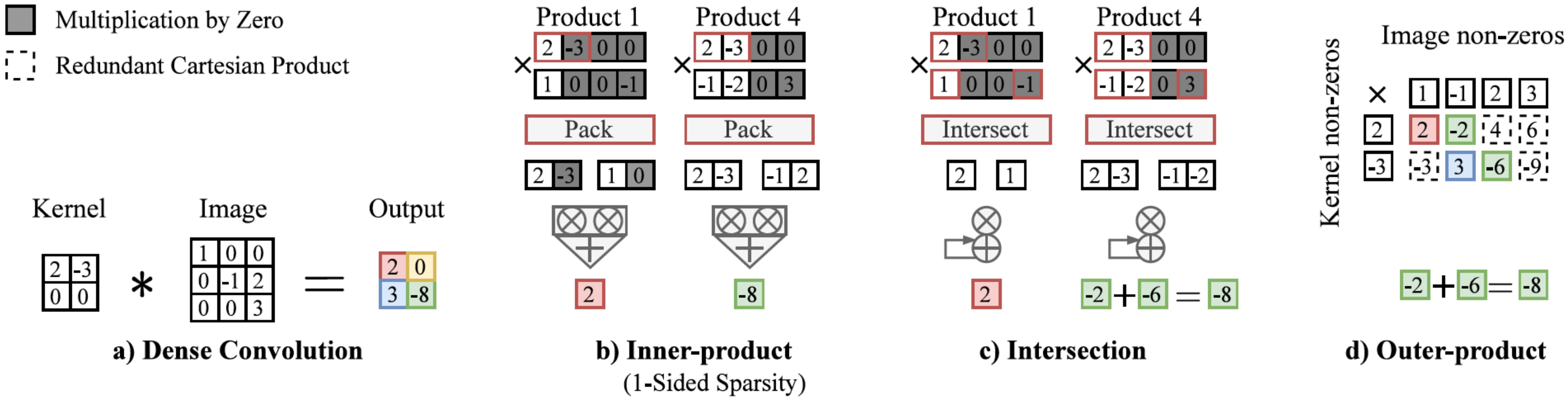
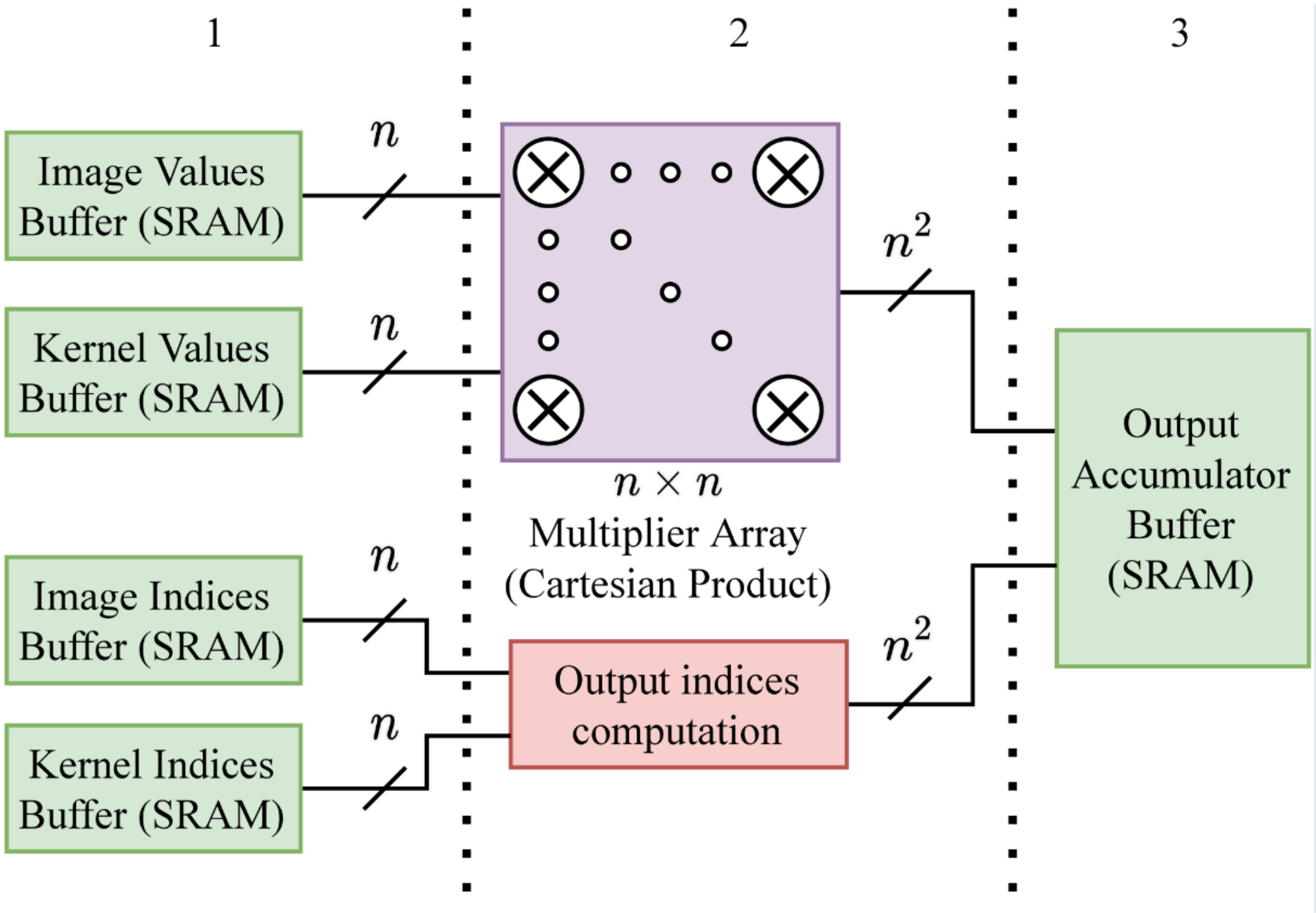


Figure 2: Convolution accelerator classes, showing zero products and Redundant Cartesian Products (RCPs). a) An example convolution of a 2×2 kernel and 3×3 image, b) Inner product/dot product, c) Intersection/streaming and, d) Outer-product.

[Lew et al., ISCA 2022]

Exploiting Two-Sided Dynamic Sparsity: SCNN



Parashar et al. (2017)

Convolutional Neural Network Training

$G_A^{\mathrm{L+1}}$

2	-3
0	0

$*$

A^L

1	0	0
0	-1	2
0	0	3

$=$

G_W^L

2	0
3	-8

Cartesian (Outer) Product

$$\underbrace{\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix}}_{G_A^{L+1}} * \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix}}_{A^L} = \underbrace{\begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}}_{G_W^L}$$

1	-1	2	3

2
-3

Cartesian Product

$$G_A^{L+1} * A^L = G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

G_A^{L+1} \ A^L	1	-1	2	3
2	2	-2	4	6
-3	-3	3	-6	-9

Cartesian Product

$$G_A^{L+1} \ast A^L = G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} \ast \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

G_A^{L+1} \ A^L	1	-1	2	3
2	2	-2	4	6
-3	-3	3	-6	-9

Cartesian Product

$$G_A^{L+1} * A^L = G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

G_A^{L+1} \ A^L	1	-1	2	3
2	2	-2	4	6
-3	-3	3	-6	-9

$$2 \times -1 = -2$$

$$-3 \times 2 = -6$$

Cartesian Product

$$G_A^{L+1} * A^L = G_W^L$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

G_A^{L+1} \ A^L	1	-1	2	3
2	2	-2	4	6
-3	-3	3	-6	-9

The Problem: Redundant Cartesian Products

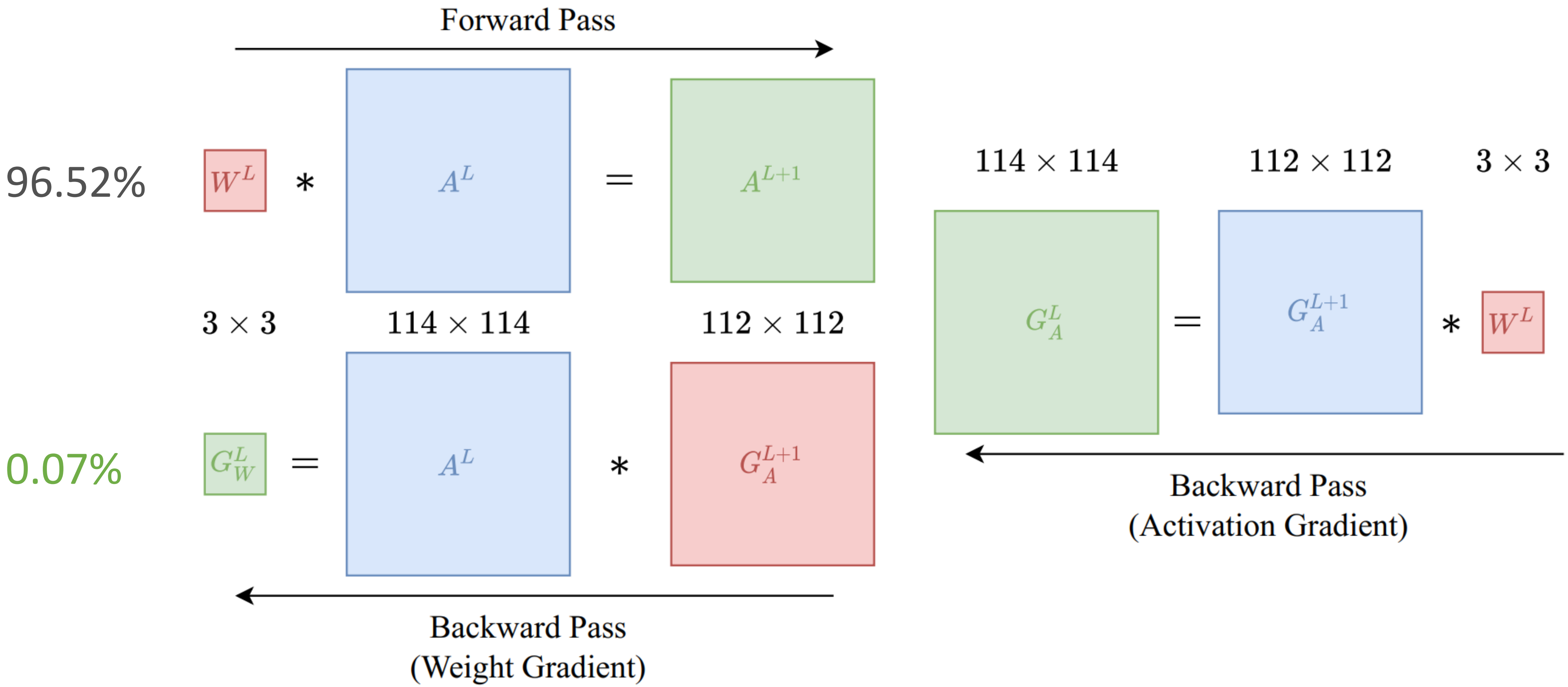
$$G_A^{L+1} = \begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} \quad A^L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} \quad G_W^L = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

$$\begin{bmatrix} 2 & -3 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 2 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & -8 \end{bmatrix}$$

G_A^{L+1} \ A^L	1	-1	2	3
2	2	-2	4	6
-3	-3	3	-6	-9

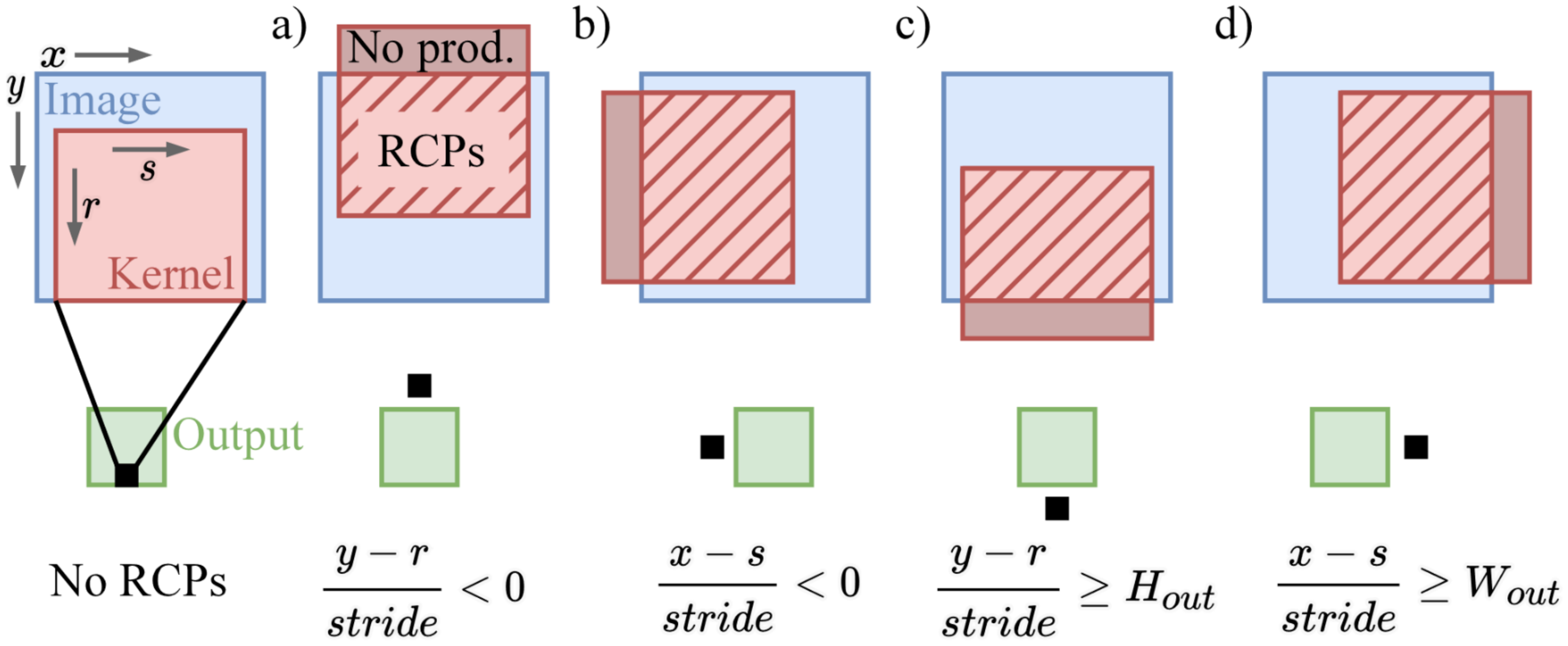
Redundant Cartesian Products (RCPs) dominate during **training**

$$\text{Outer-product Efficiency} = \frac{H_{out} \times W_{out}}{H \times W}$$



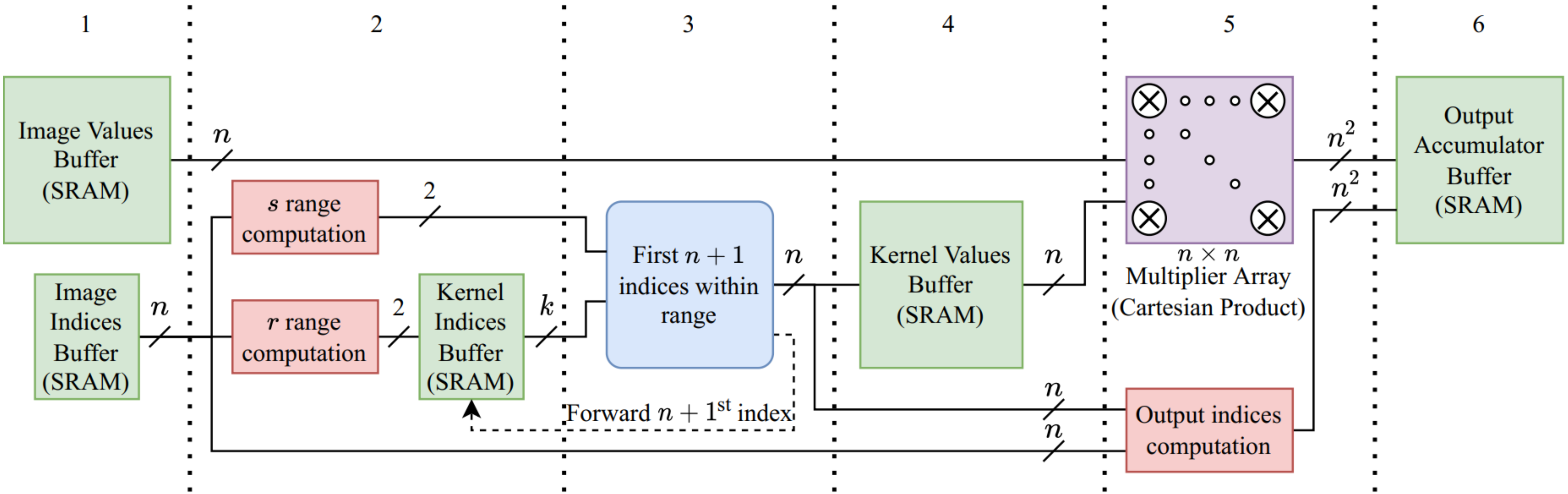
[Lew et al., ISCA 2022]

Redundant Cartesian Products



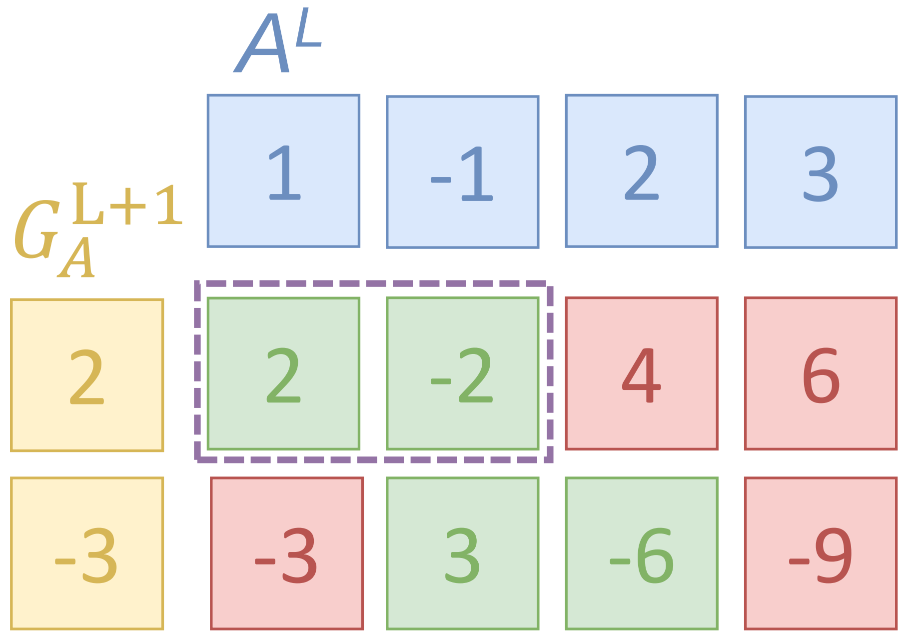
[Lew et al., ISCA 2022]

Anticipator Accelerator (ANT)

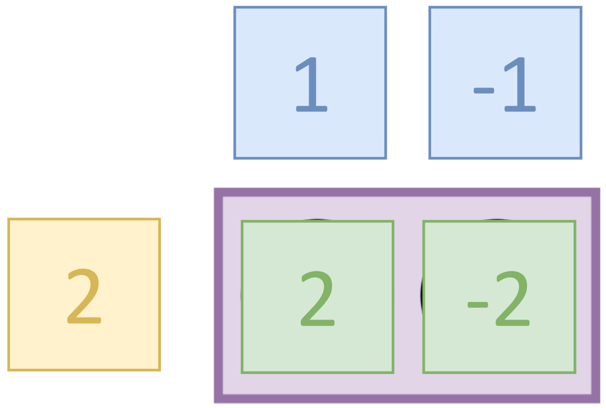


[Lew et al., ISCA 2022]

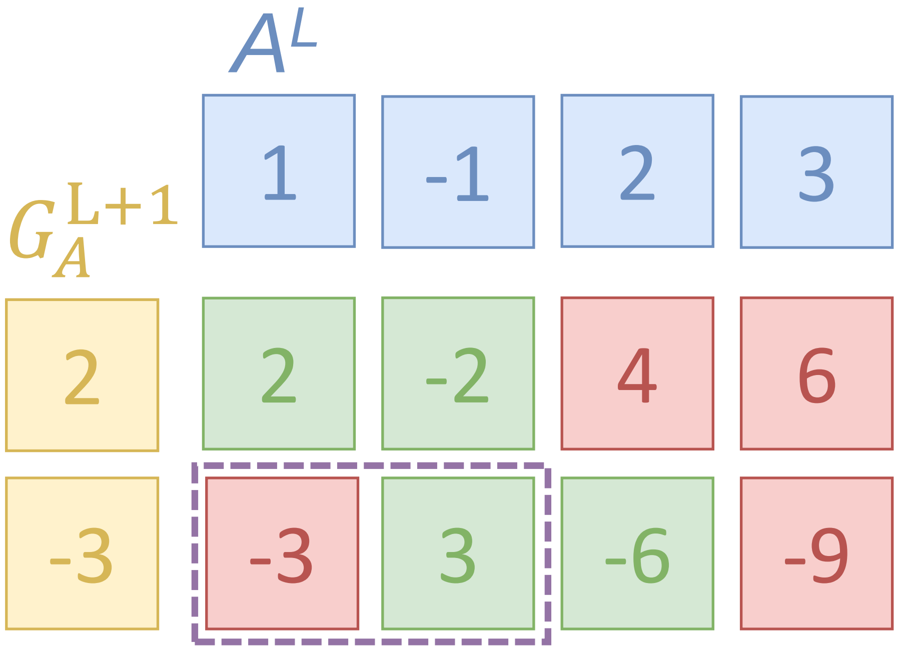
Mapping onto a Multiplier Array



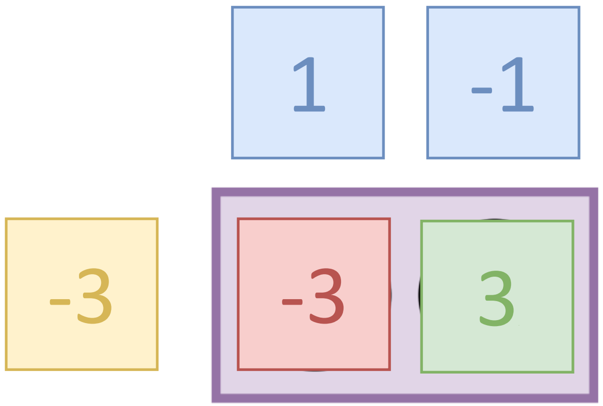
Cycle 1



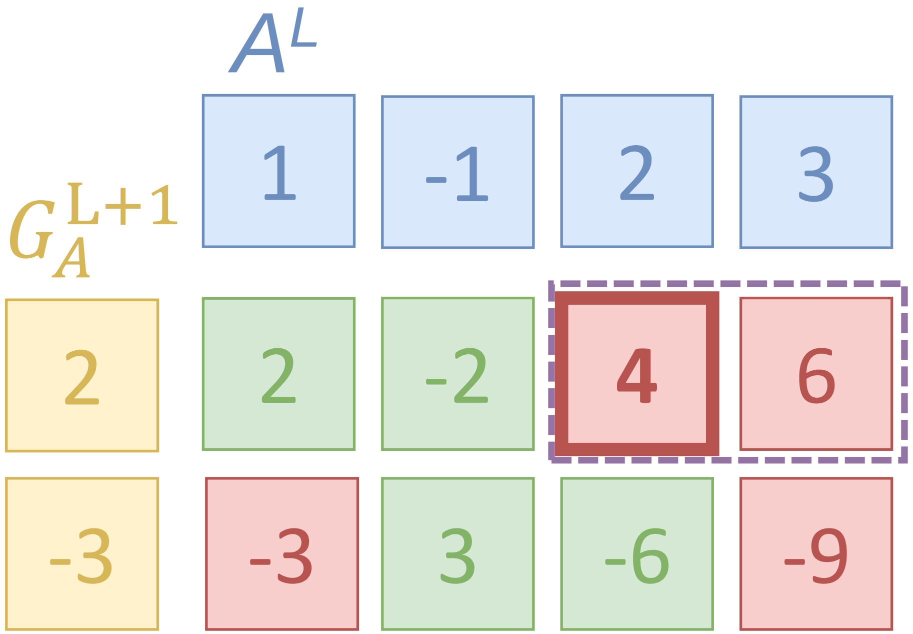
Mapping onto a Multiplier Array



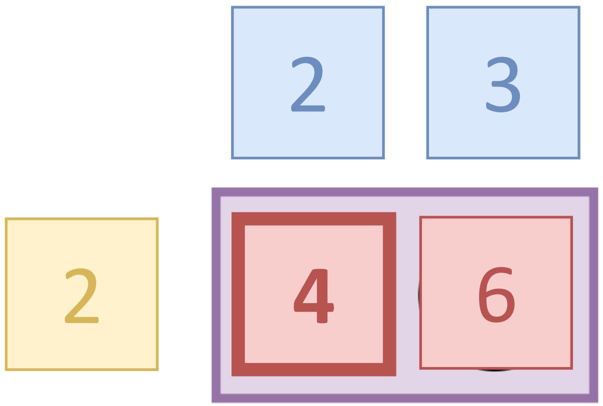
Cycle 2



Mapping onto a Multiplier Array: Skipping RCPs

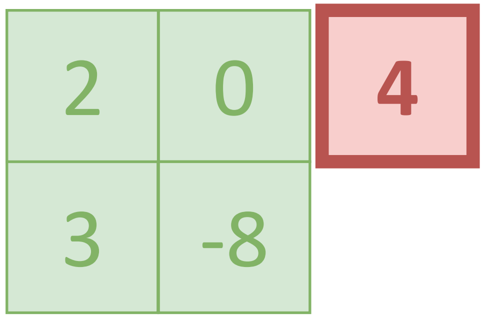


Cycle 3

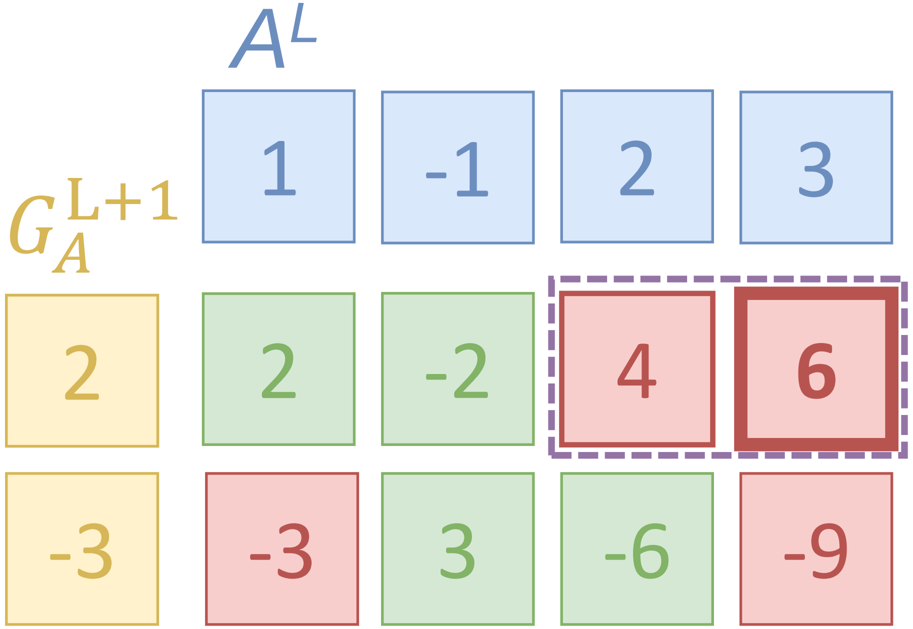


output coordinate = $\left(\frac{x-s}{stride}, \frac{y-r}{stride}\right)$

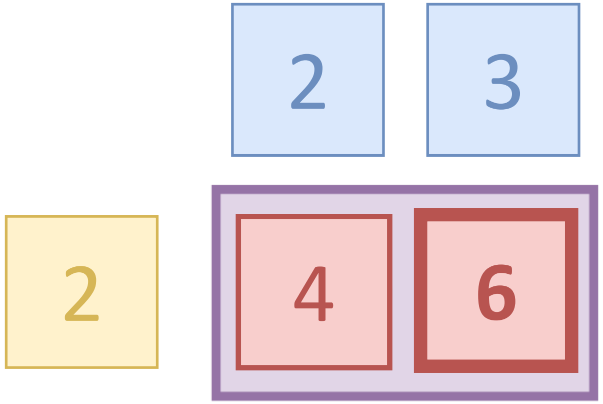
$\left(\frac{2-0}{1}, \frac{0-0}{1}\right) = (2,0)$



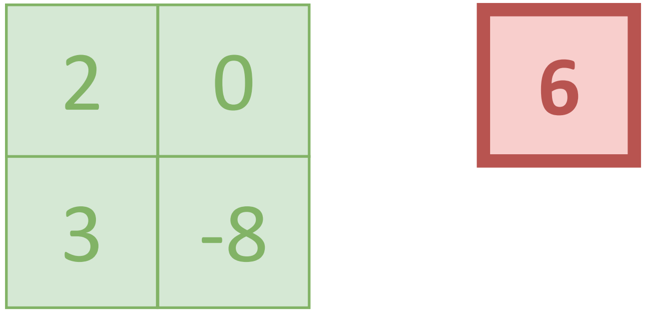
Mapping onto a Multiplier Array: Skipping RCPs



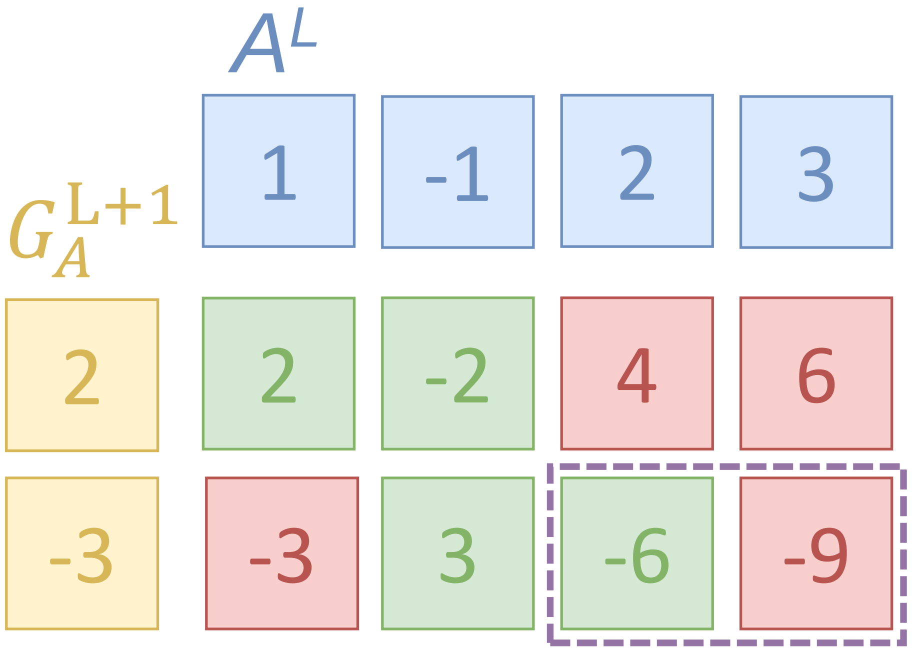
Cycle 3



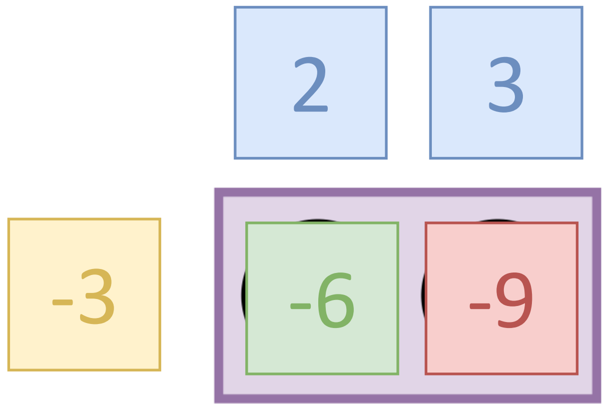
output coordinate = $\left(\frac{x - s}{stride}, \frac{y - r}{stride}\right) = \left(\frac{3-0}{1}, \frac{0-0}{1}\right) = (3,0)$

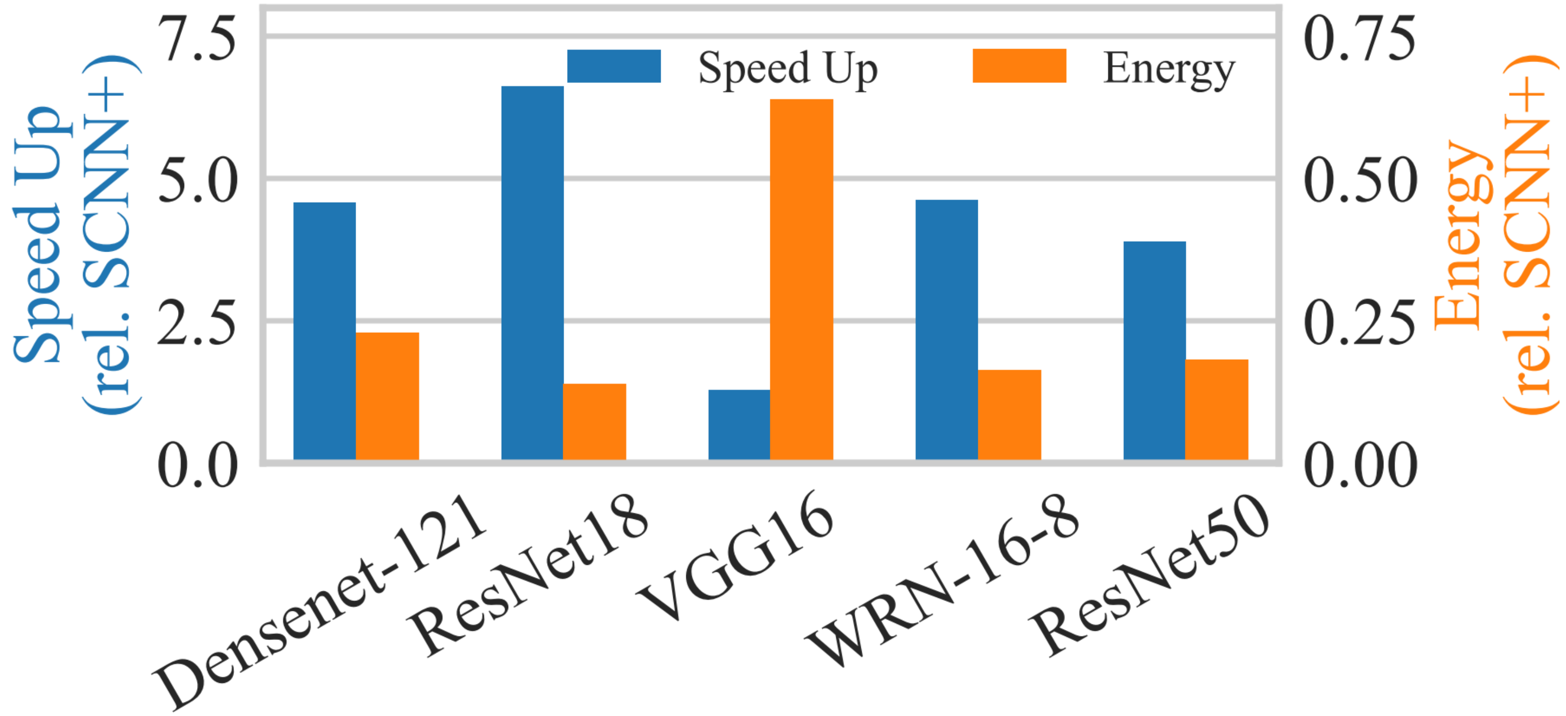


Mapping onto a Multiplier Array



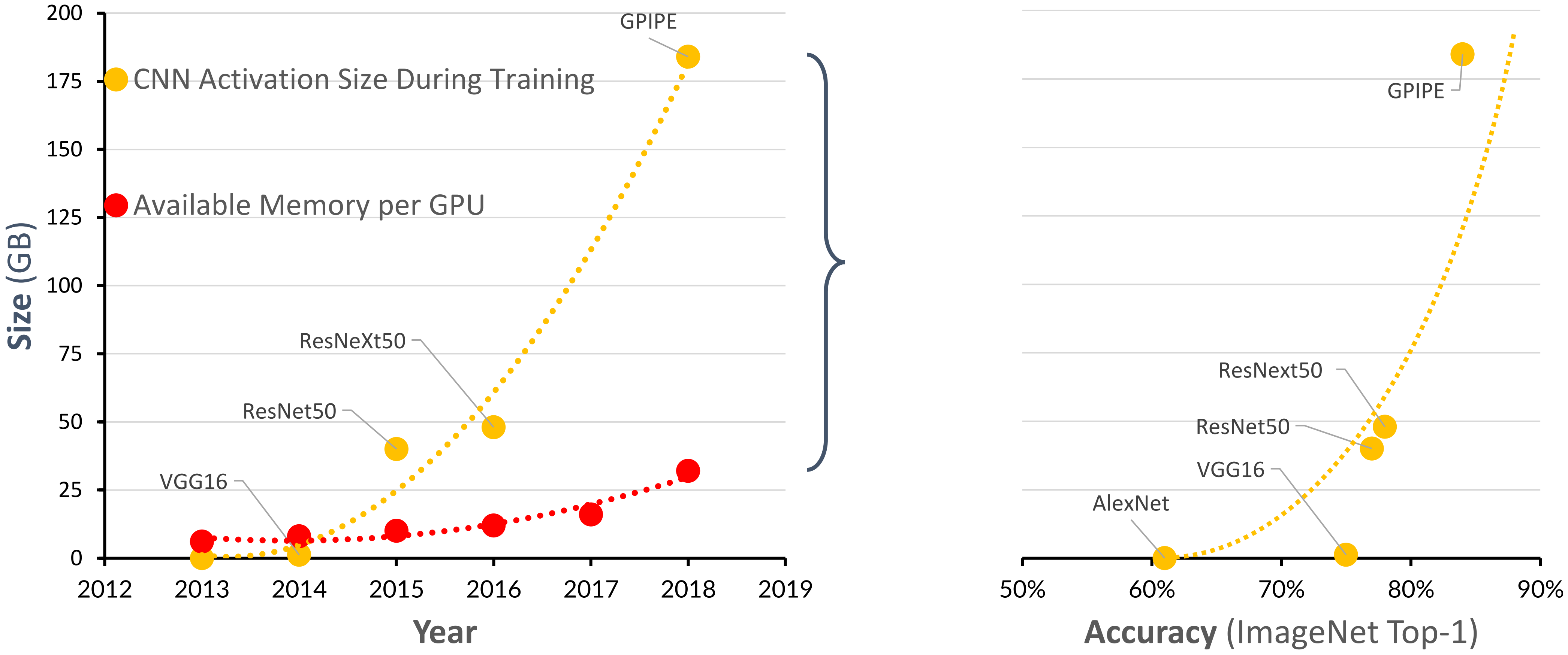
Cycle 3





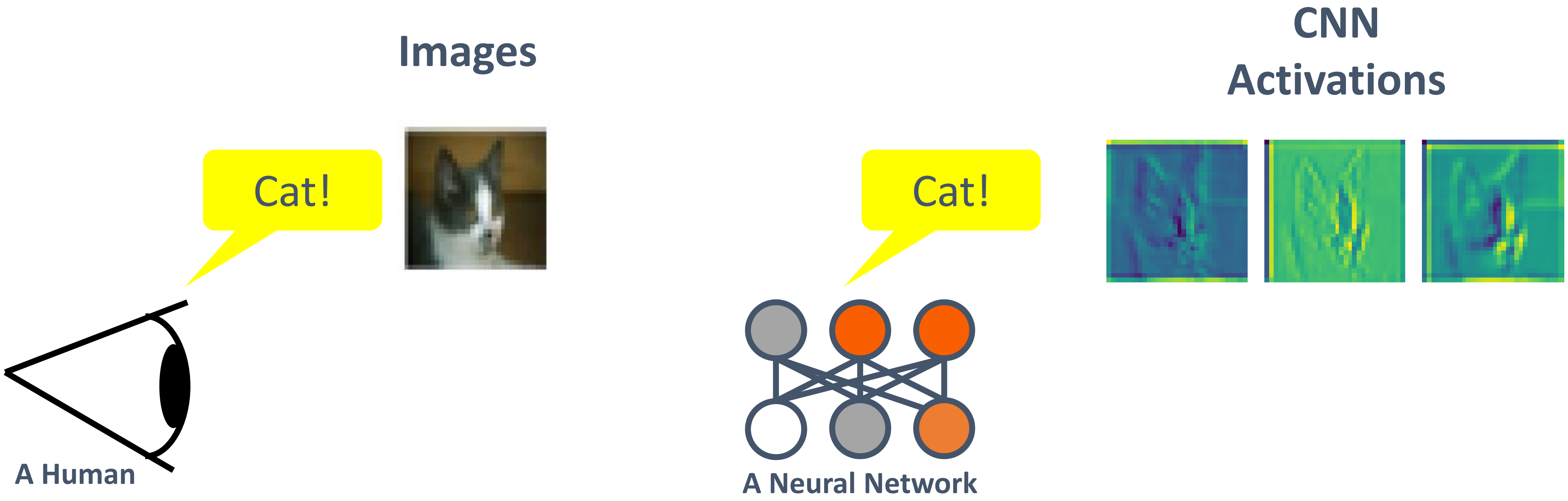
[Lew et al., ISCA 2022]

Bigger Models, More Memory



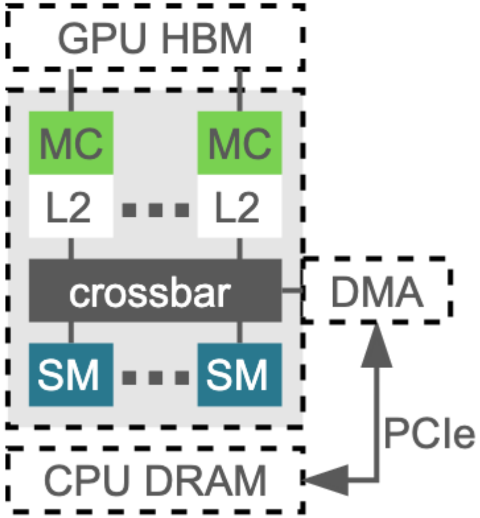
[Evans et al., ISCA 2020]

Images versus Activations

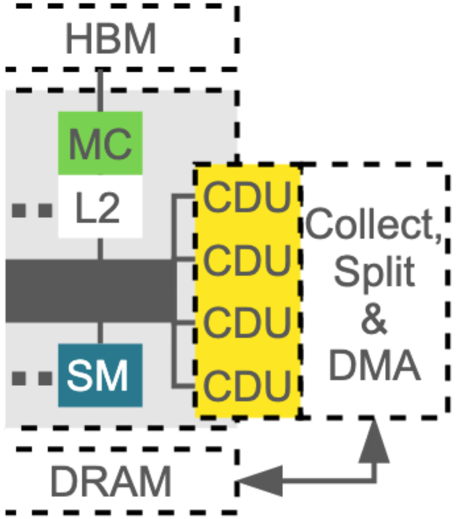


[Evans et al., ISCA 2020]

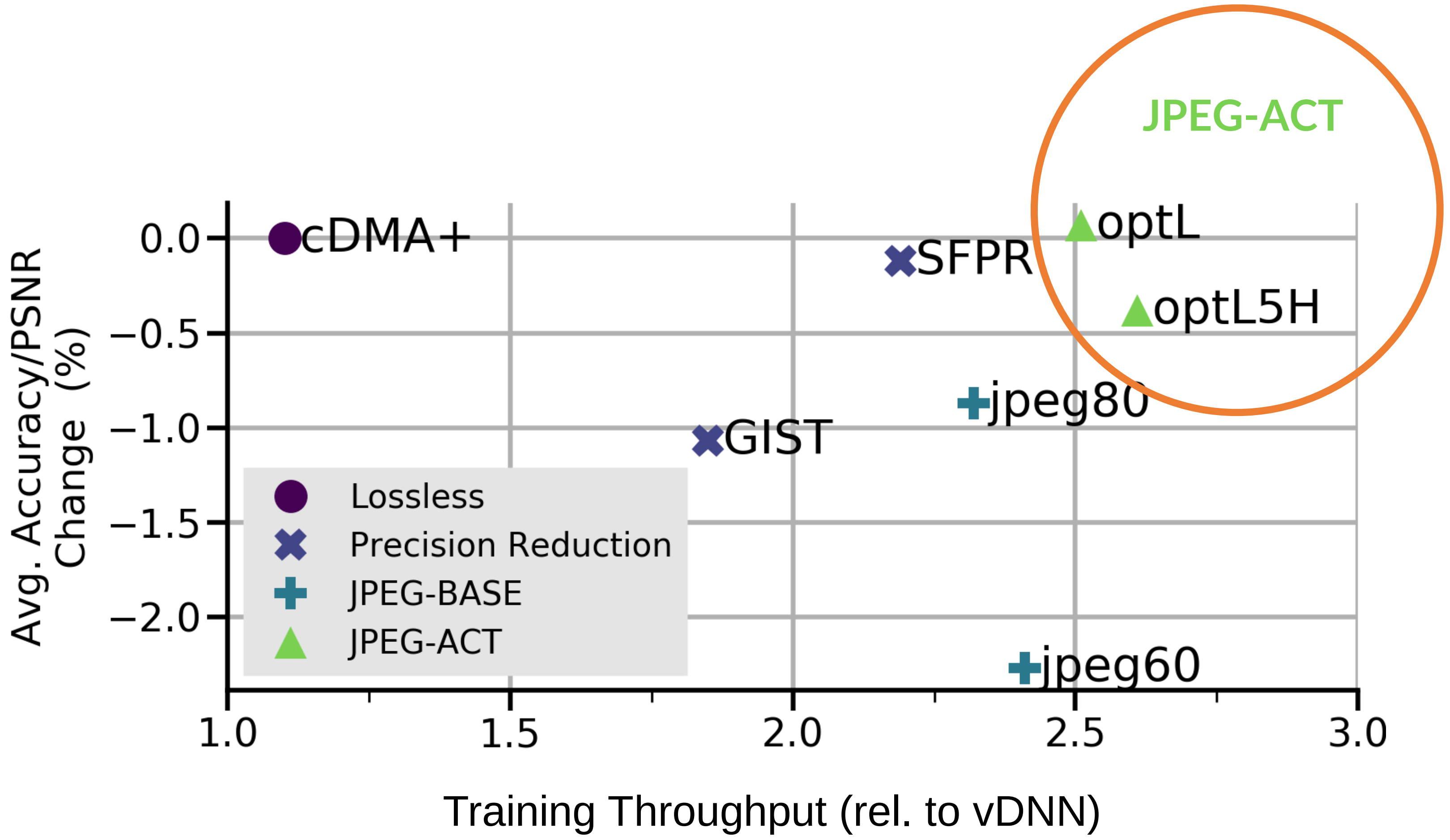
Baseline
(vDNN)



JPEG-ACT

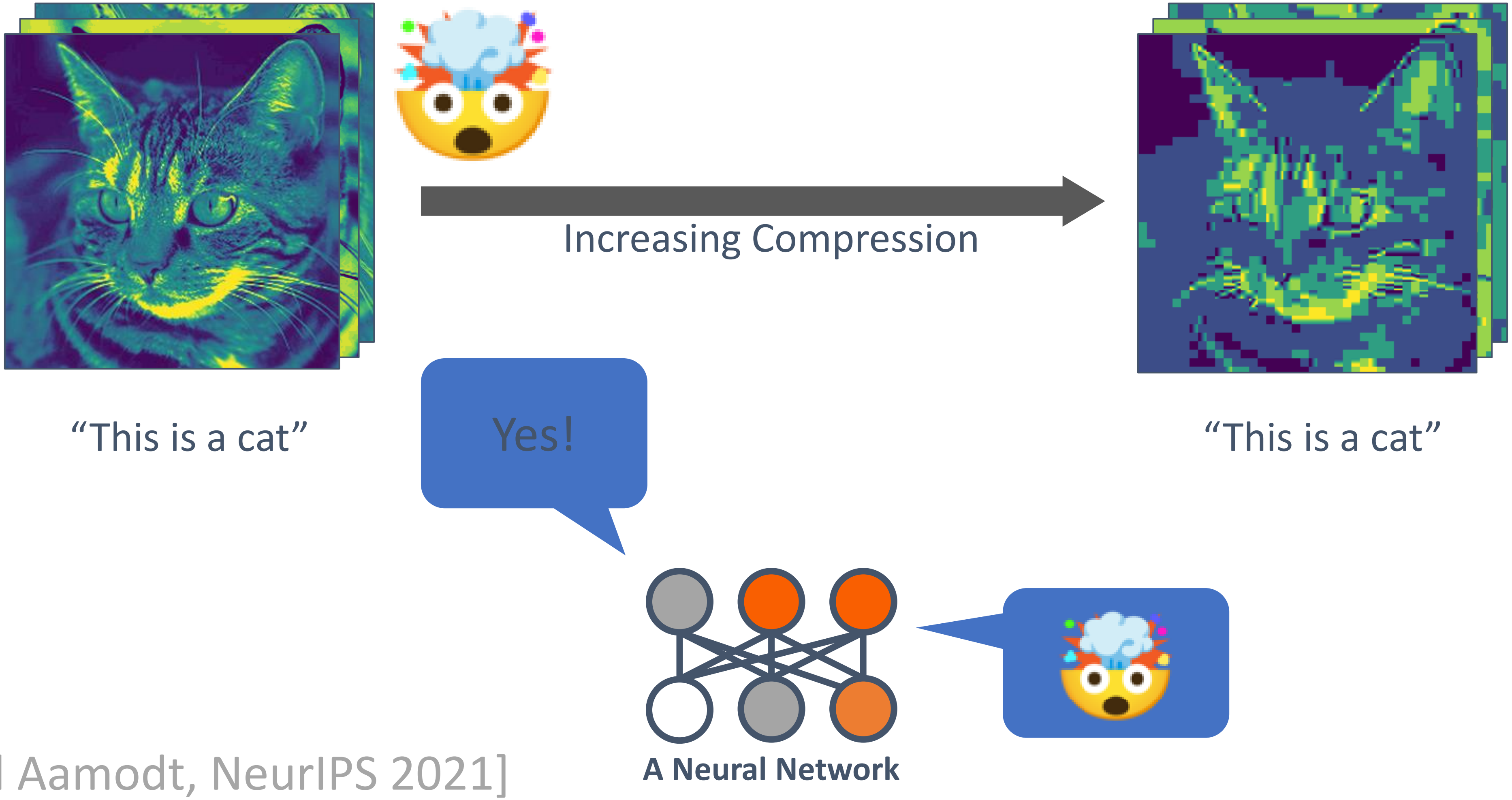


[Evans et al., ISCA 2020]

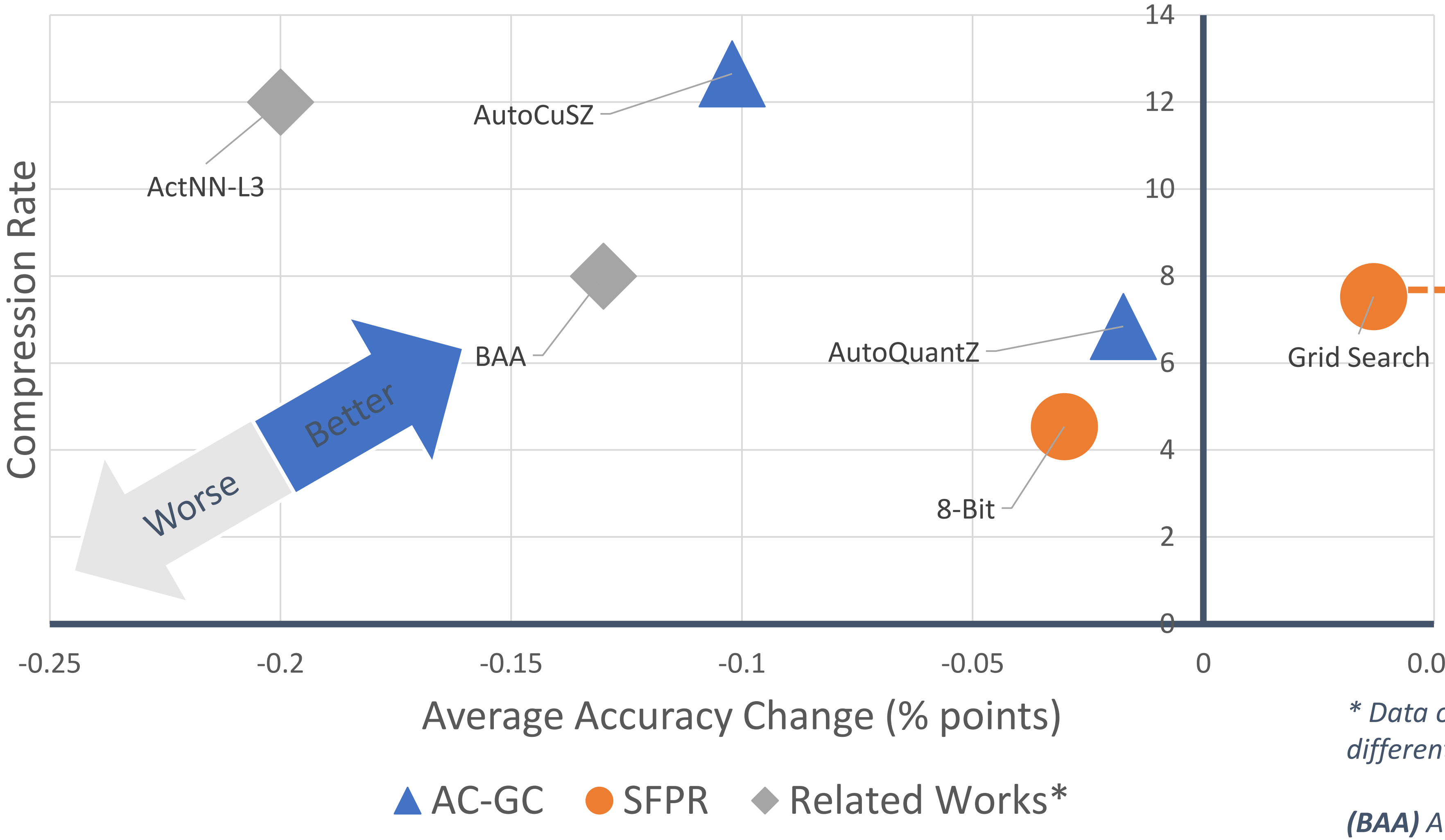


[Evans et al., ISCA 2020]

Convergence versus (Lossy) Compression



[Evans and Aamodt, NeurIPS 2021]



>16x cost
Sensitive to seeding

** Data obtained from original work; may use a different set of models from AC-GC*

***(BAA)** A. Chakrabarti, B. Moseley, in NeurIPS 2020*
***(ACTNN)** J. Chen, L. Zheng, et. al, in ICML 2021*

Summary

- Obtaining greater performance for machine learning will increasingly require shifting towards specialized hardware
- ReSprop, SWAT can reduce computation demand during training by identifying computations to elide at minimal impact on accuracy leading to sparse computations
- Efficiently supporting sparse computations in hardware is a challenge, and (even more so during training since parameters are changing)
- Lossy compression can help performance (and/or increase model size) by greatly reducing memory demands. Challenging to use during training since need to anticipate impact on validation accuracy.