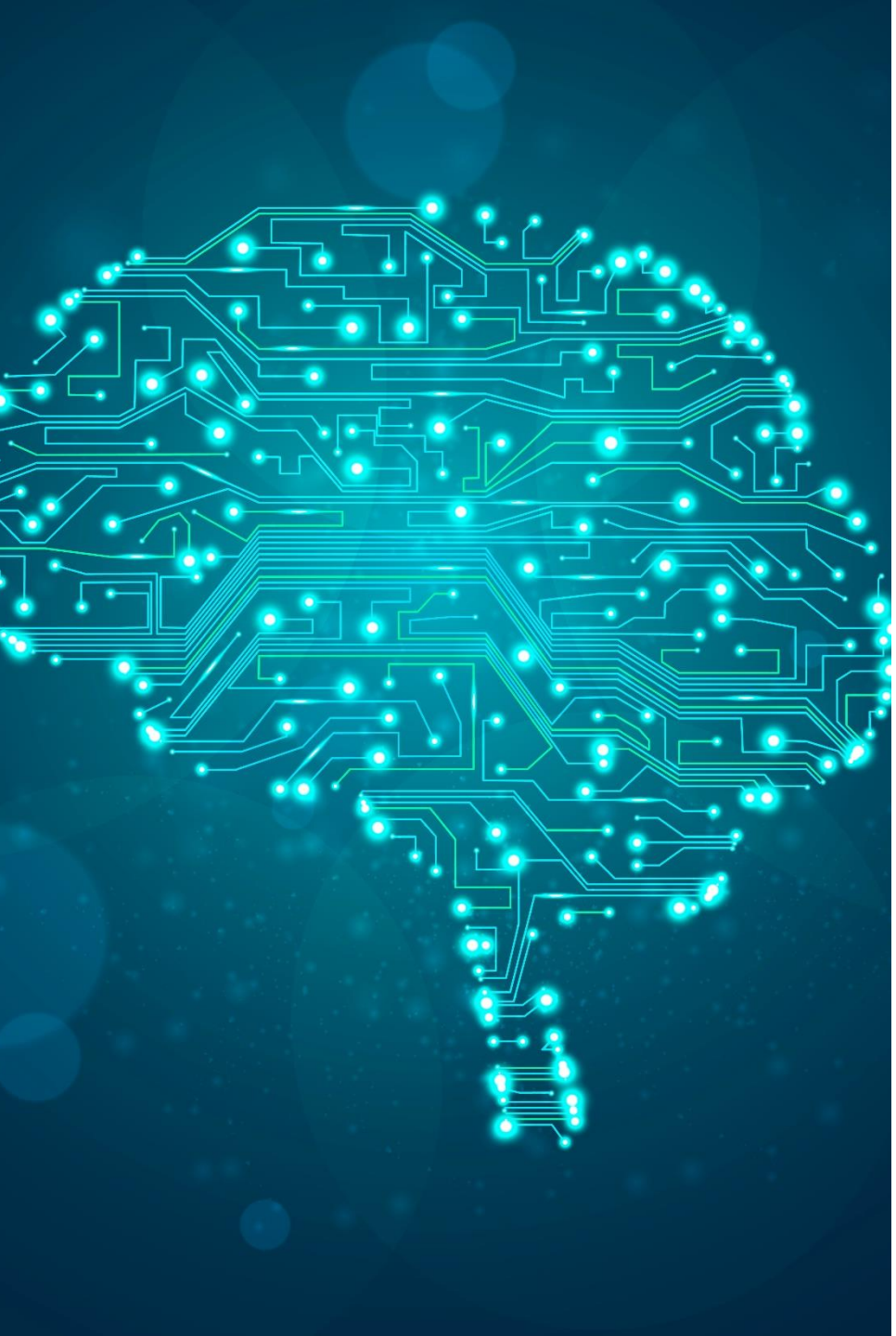


IDEC Lecture Series 2020

Hardware Acceleration for Machine Learning

Joo-Young Kim

05/06/2020

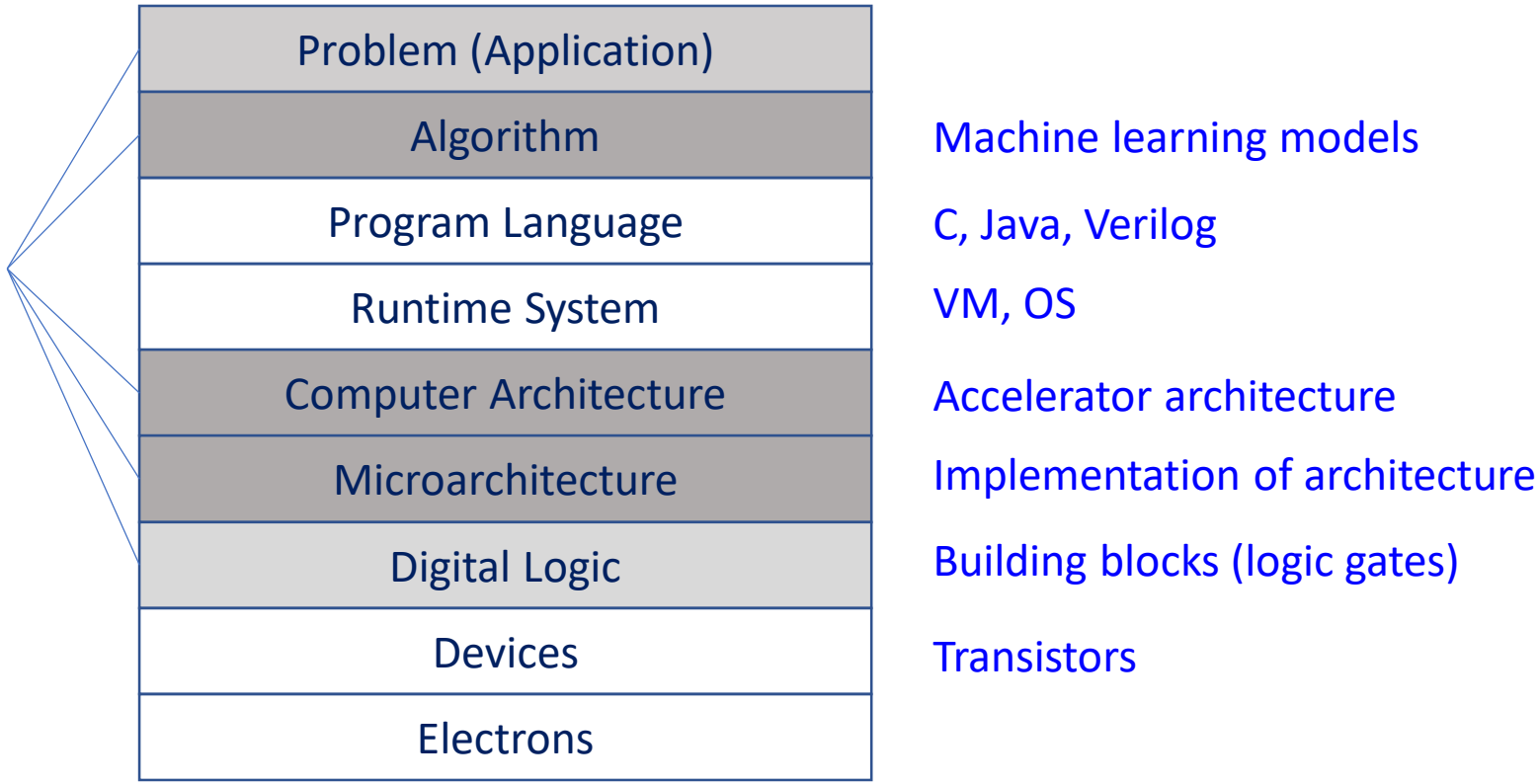
jooyoung1203@kaist.ac.kr

KAIST

Lecture Scope

- Goal
  - Understanding latest machine learning models and their computations
  - Learn how to design hardware accelerators for machine learning applications
- Prerequisites
  - Digital System Design, Introduction to Computer Architecture

Hardware Acceleration for Machine Learning



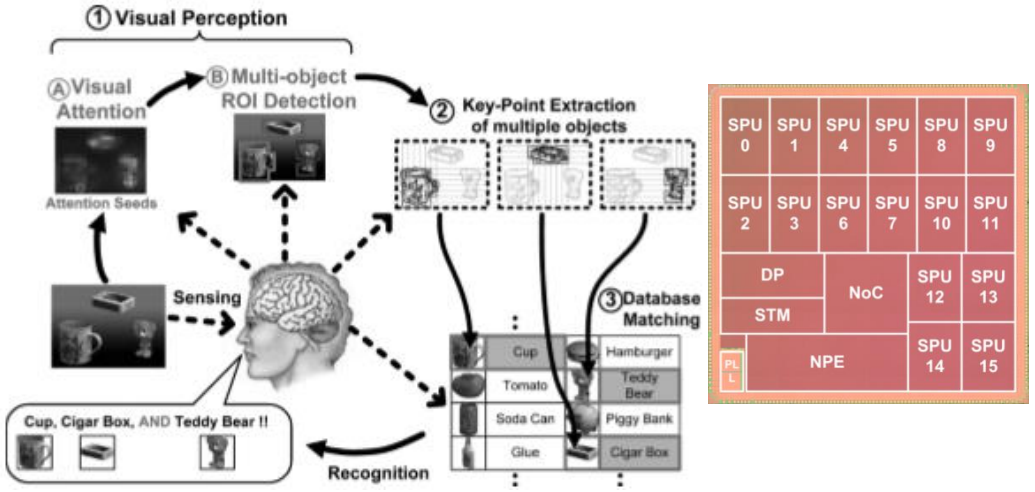
Instructor

- Prof. Joo-Young Kim
- Education: Ph. D. in EE KAIST 2010
- Work Experience

  Visiting Researcher at Microsoft Research (2010-2011)

  Researcher at Microsoft Research (2012-2017)

  Hardware engineering lead at Microsoft Azure (2018-2019)



[BONE-V Computer Vision Chip]

ROBERT MCMILLAN BUSINESS 06.16.14 06:30 AM

MICROSOFT SUPERCHARGES BING SEARCH WITH PROGRAMMABLE CHIPS

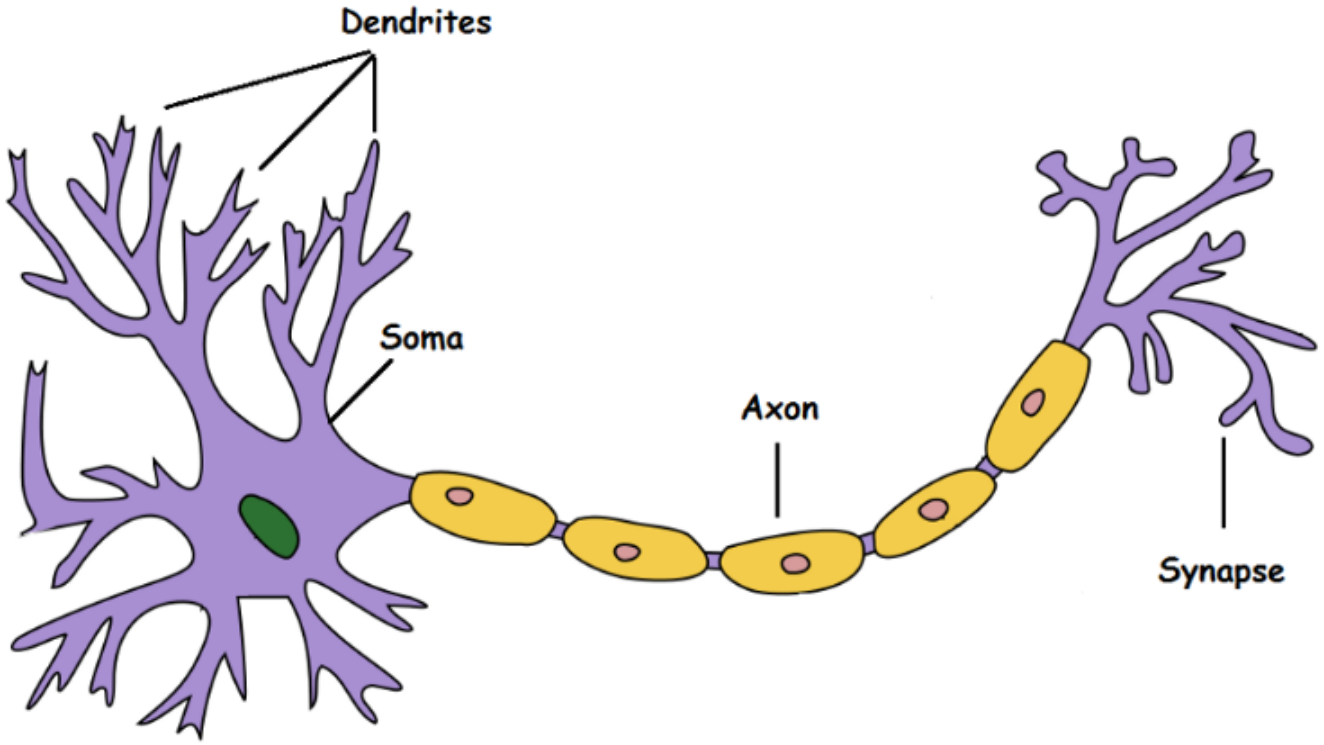
[Project Catapult]

Agenda

- Deep Neural Network Models
  - Multi-Layer Perceptron (MLP)
  - Convolutional Neural Network (CNN)
  - Recurrent Neural Network (RNN)
  - Training and backpropagation
- ML Accelerators for Mobile/Edge
  - DianNao (2014), Eyeriss (2016)
  - EIE (2016), UNPU (2019)
- ML Accelerators for Cloud Datacenters
  - TPU (2017), BrainWave (2018)
  - GPU

Biological Neuron

- Overly simplified human neuron model

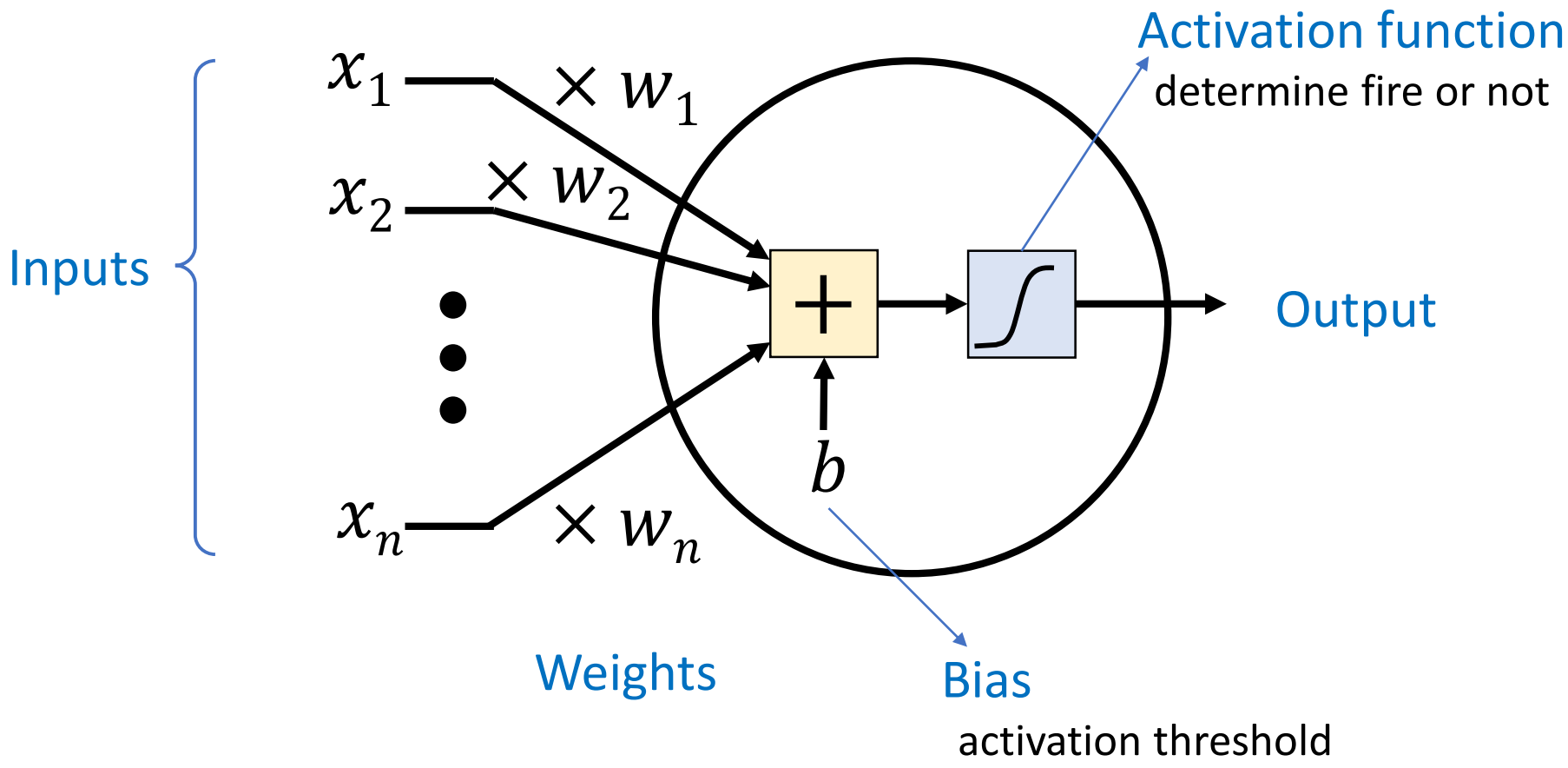


# of neurons: 100B
# of connections per neuron: 10^4
Signal sending time: 10^{-3}
Face recognition: 10^{-1}

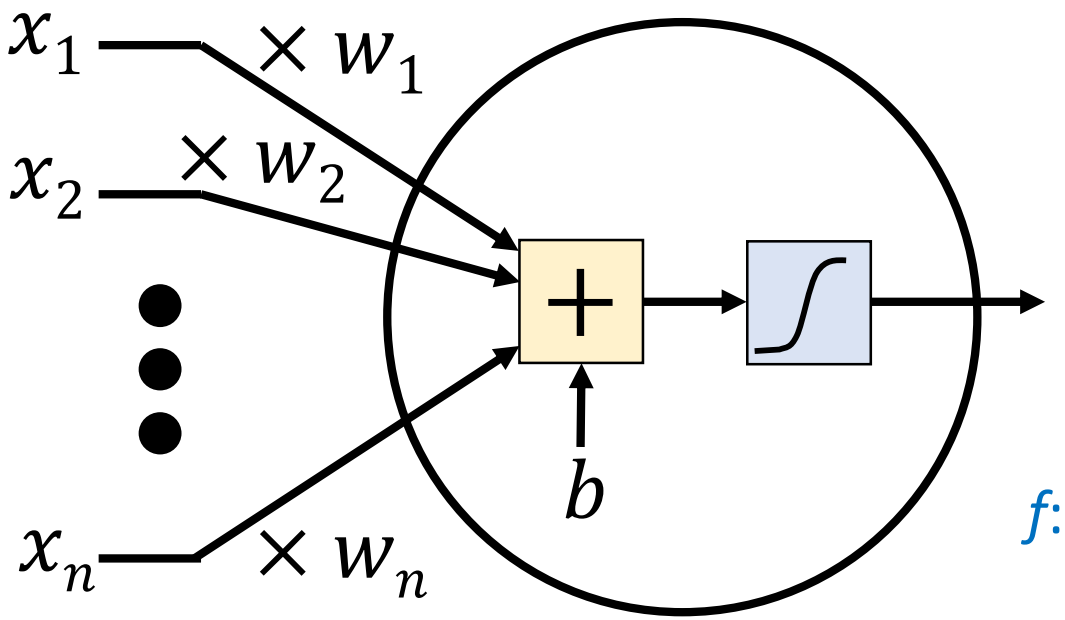
- Dendrite: receives signals from other neurons
- Soma: processes the information
- Axon/Synapse: transmits the output of this neuron to other neurons

Artificial Neuron (Perceptron, 1957)

- Frank Rosenblatt's single layer perceptron for binary classification



Equation for an Artificial Neuron

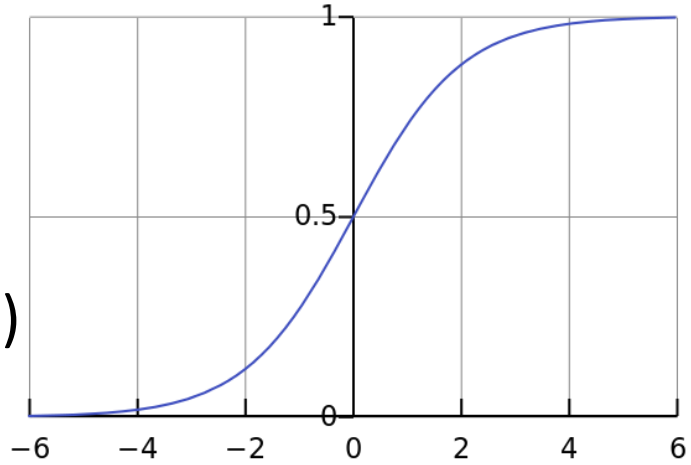


$$f(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$$

f: differentiable, non-linear function (Sigmoid, tanh, ..)

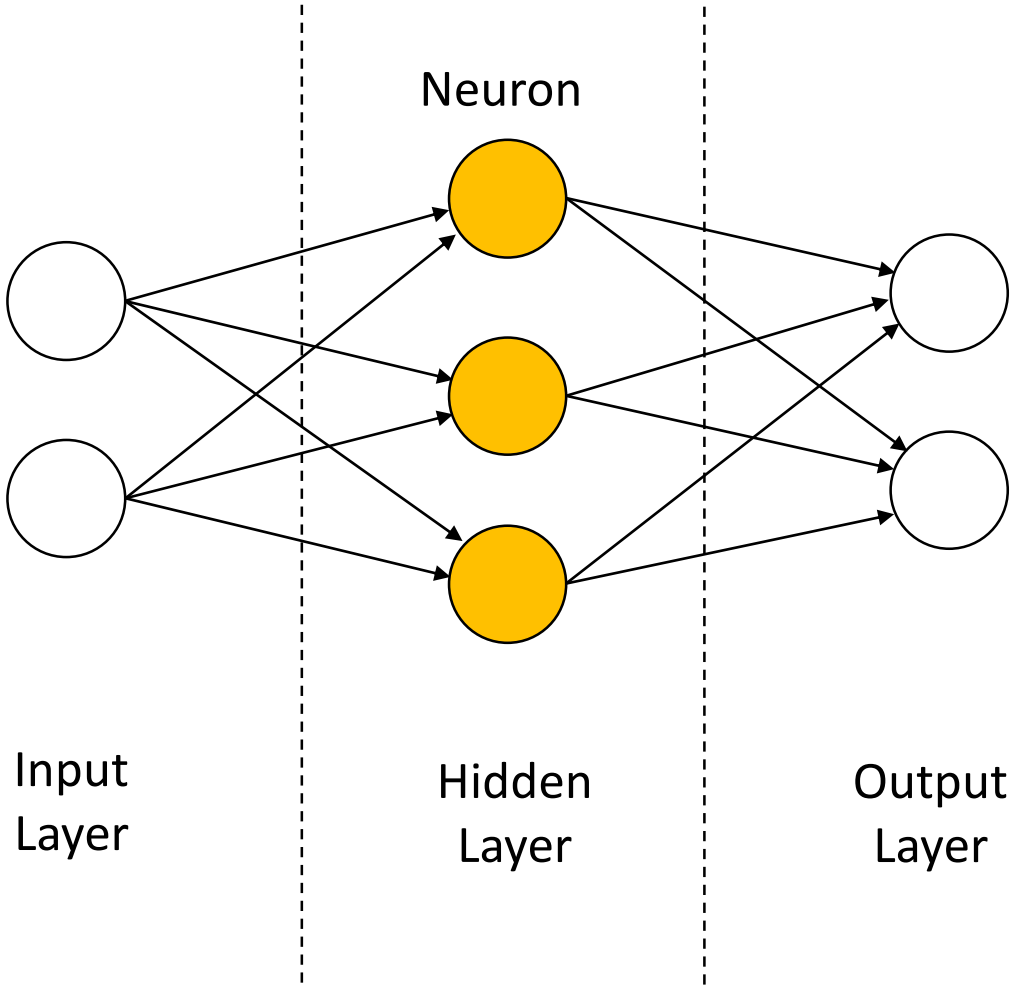
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)\cdot(1- \sigma(z))$$

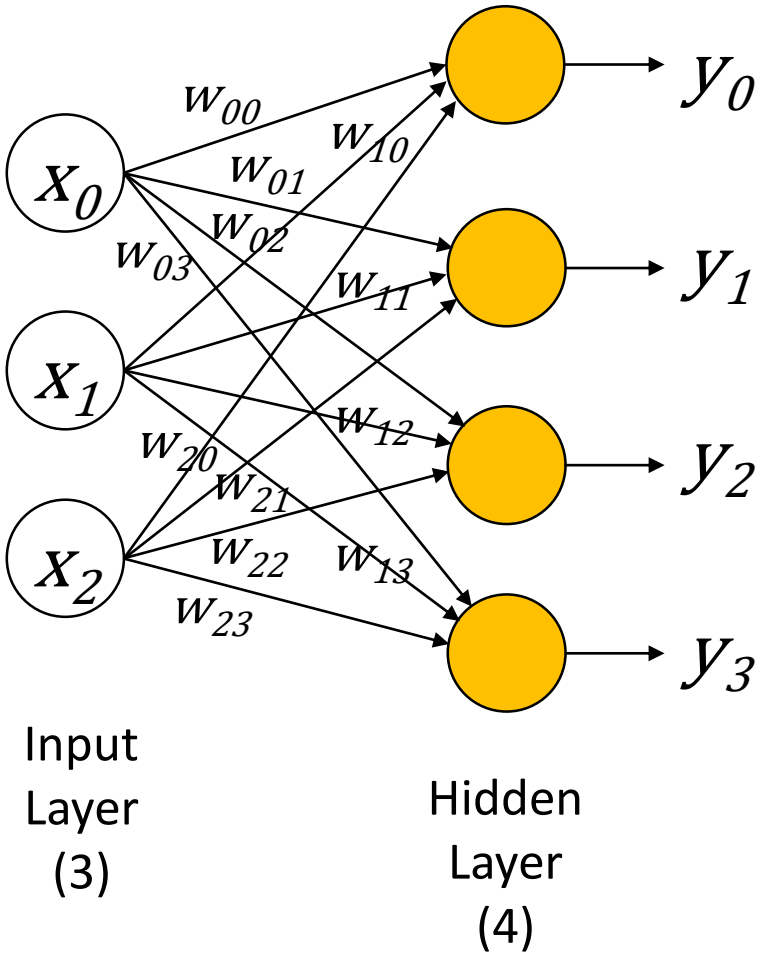


Multi-Layer Perceptron (MLP)

- Equivalent to artificial neural network before (1980s)



Equations for a Layer in MLP



$$y_0 = f(w_{00}x_0 + w_{10}x_1 + w_{20}x_2 + b_0)$$

$$y_1 = f(w_{01}x_0 + w_{11}x_1 + w_{21}x_2 + b_1)$$

$$y_2 = f(w_{02}x_0 + w_{12}x_1 + w_{22}x_2 + b_2)$$

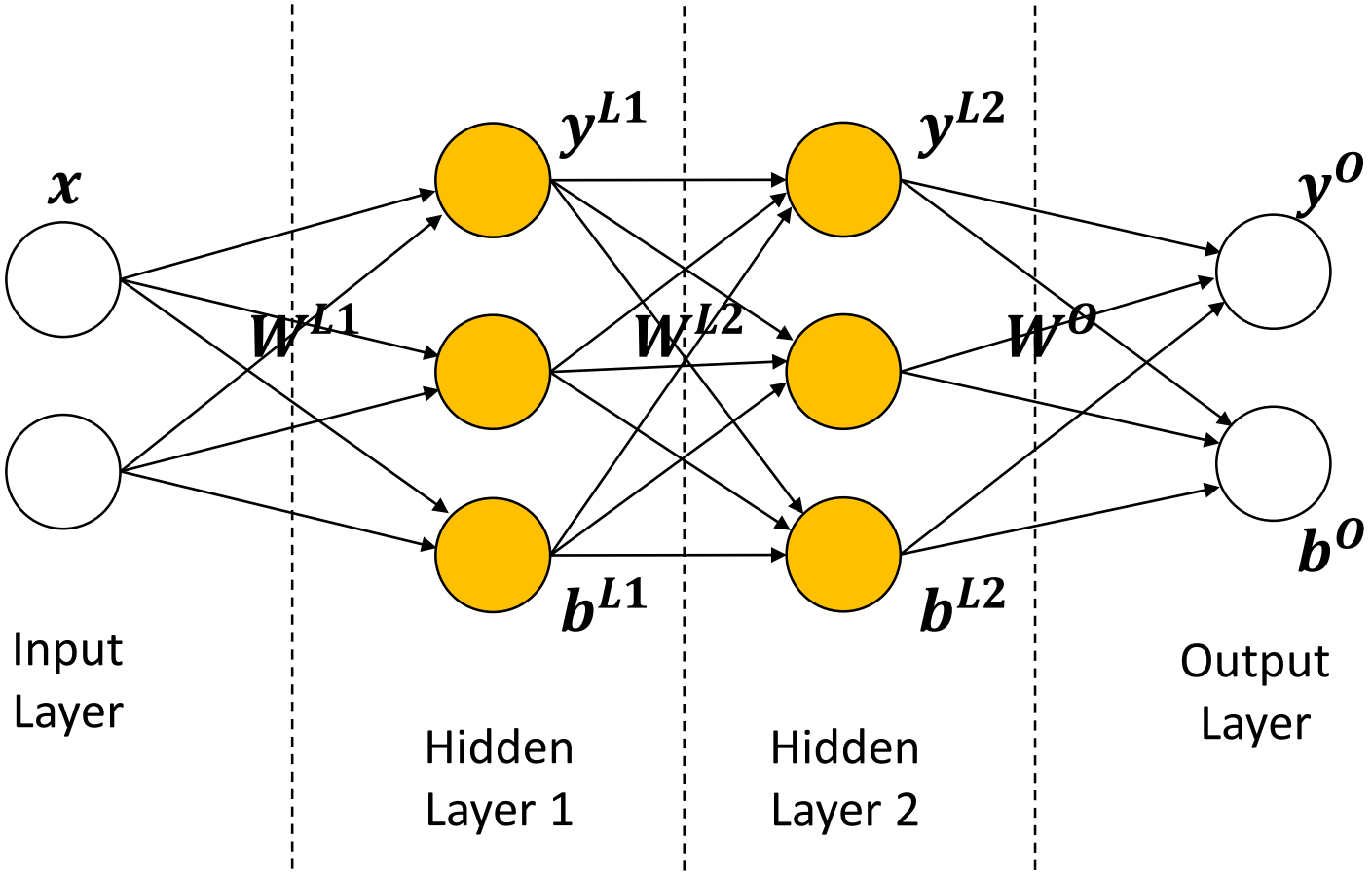
$$y_3 = f(w_{03}x_0 + w_{13}x_1 + w_{23}x_2 + b_3)$$

$$\underset{4\text{x}1}{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix}} = f\left(\underset{4\text{x}3}{\begin{bmatrix} w_{00} & w_{10} & w_{20} \\ w_{01} & w_{11} & w_{21} \\ w_{02} & w_{12} & w_{22} \\ w_{03} & w_{13} & w_{23} \end{bmatrix}} \underset{3\text{x}1}{\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}} + \underset{4\text{x}1}{\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}} \right)$$

$$\boldsymbol{y = f(Wx + b)}$$ ➔ **Matrix-Vector Multiplication!**

Equations for MLP

- Multiple hidden layers = Composition of matrix operations
- Multiple hidden layers = Deep neural network



$$y^{L1} = f^{L1}(W^{L1}x + b^{L1})$$

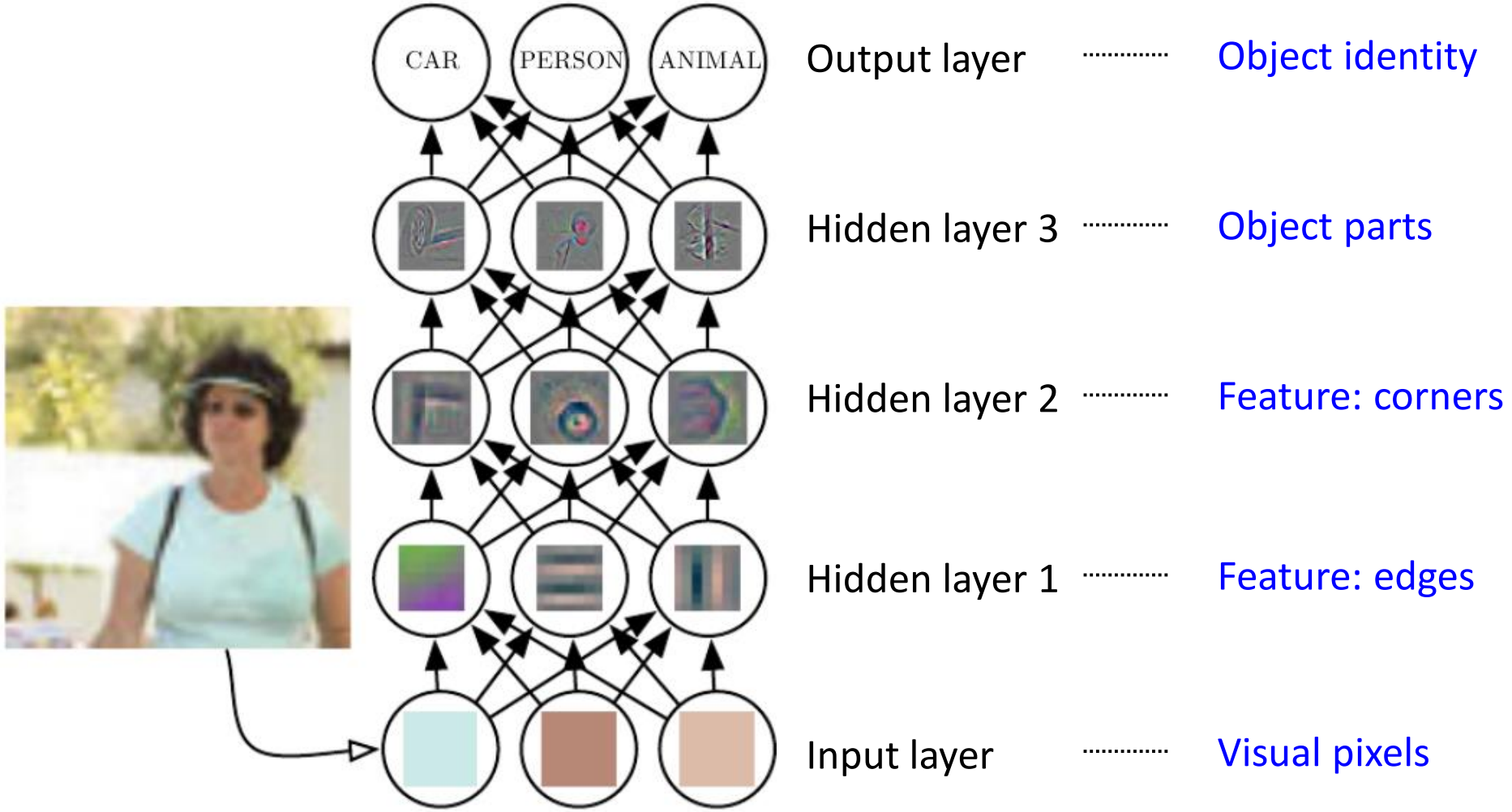
$$y^{L2} = f^{L2}(W^{L2}y^{L1} + b^{L2})$$

$$y^O = f^O(W^O y^{L2} + b^O)$$

Or

$$y^O = f^O(W^O(f^{L2}(W^{L2}(f^{L1}(W^{L1}x + b^{L1})) + b^{L2}) + b^O)$$

Deep Learning Model

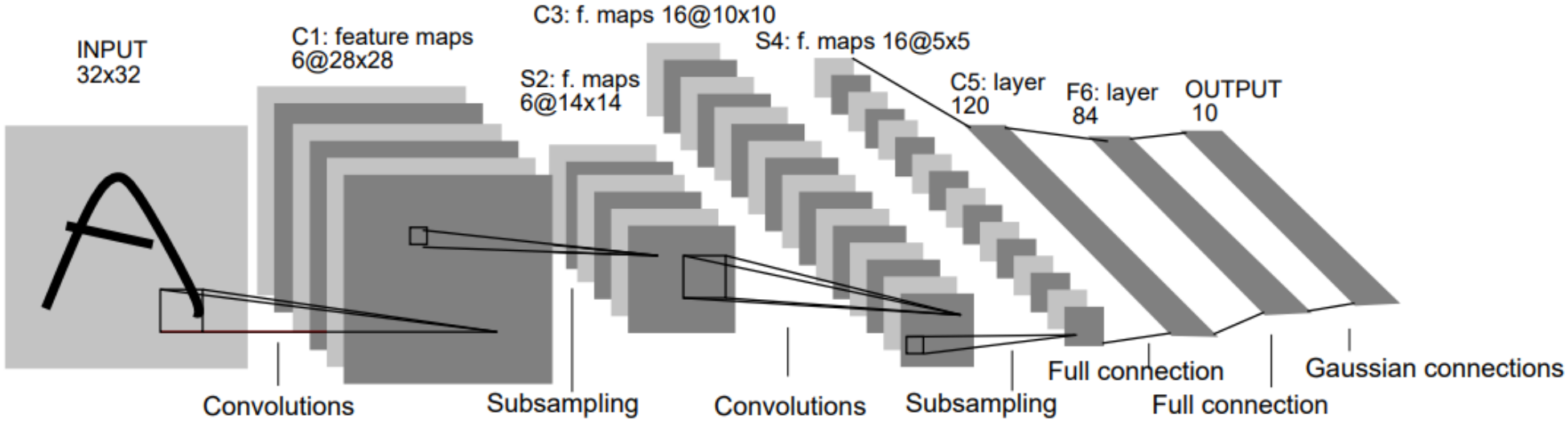


http://www.deeplearningbook.org/

Convolutional Neural Network (CNN)

- Good for image recognition and classification
- Not fully connected, less computation
  - Convolution + Pooling + Fully Connected

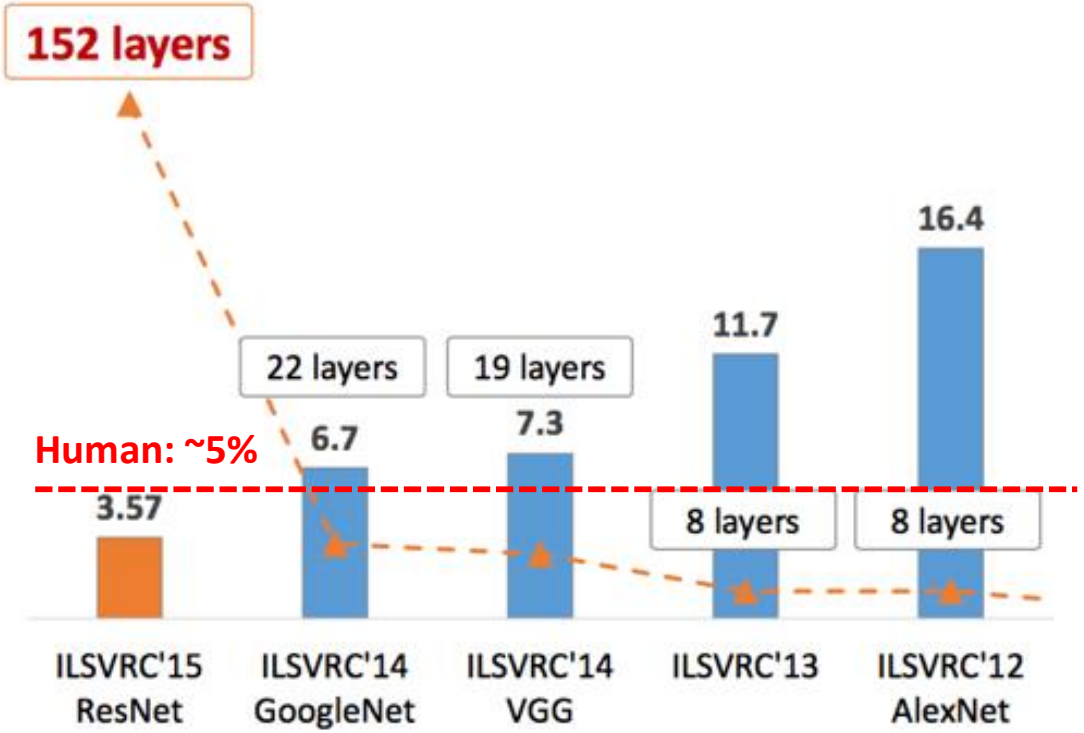
LeNet for handwriting recognition (MNIST)



Y. LeCun, "Gradient-Based Learning Applied to Document Recognition," Proceedings of IEEE 1998

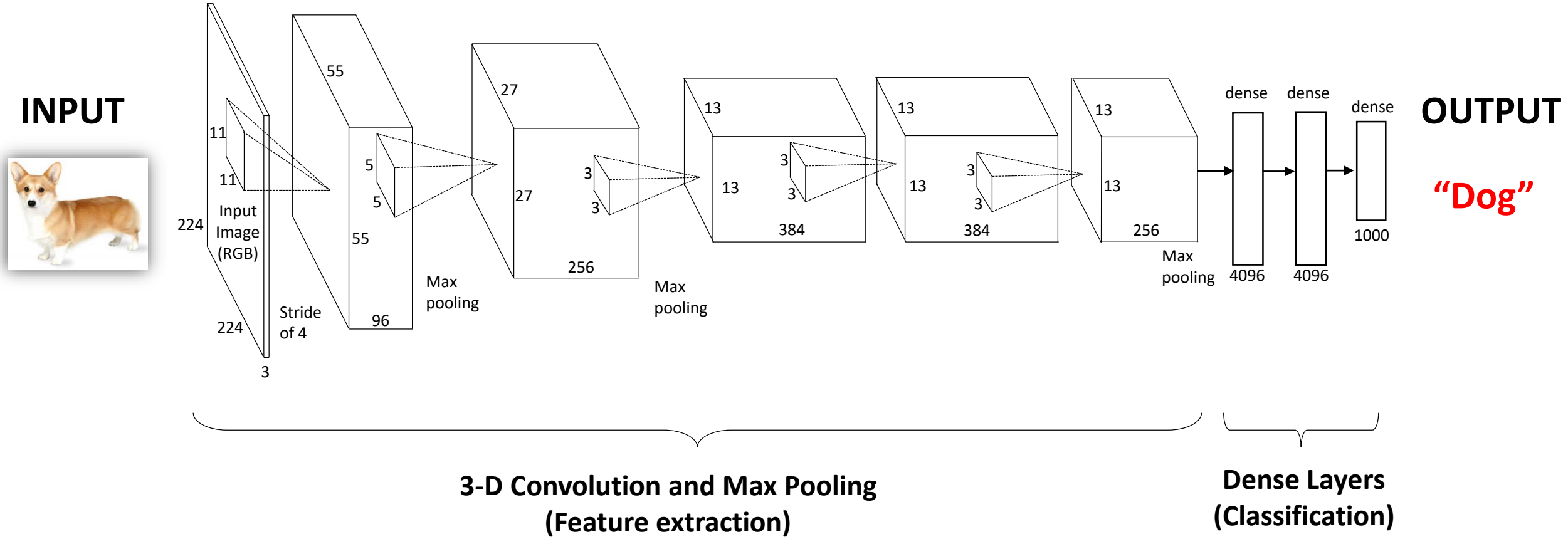
ImageNet Competition

- ImageNet Large Scale Visual Recognition Challenge (ILSVRC)
  - More than 14 million images



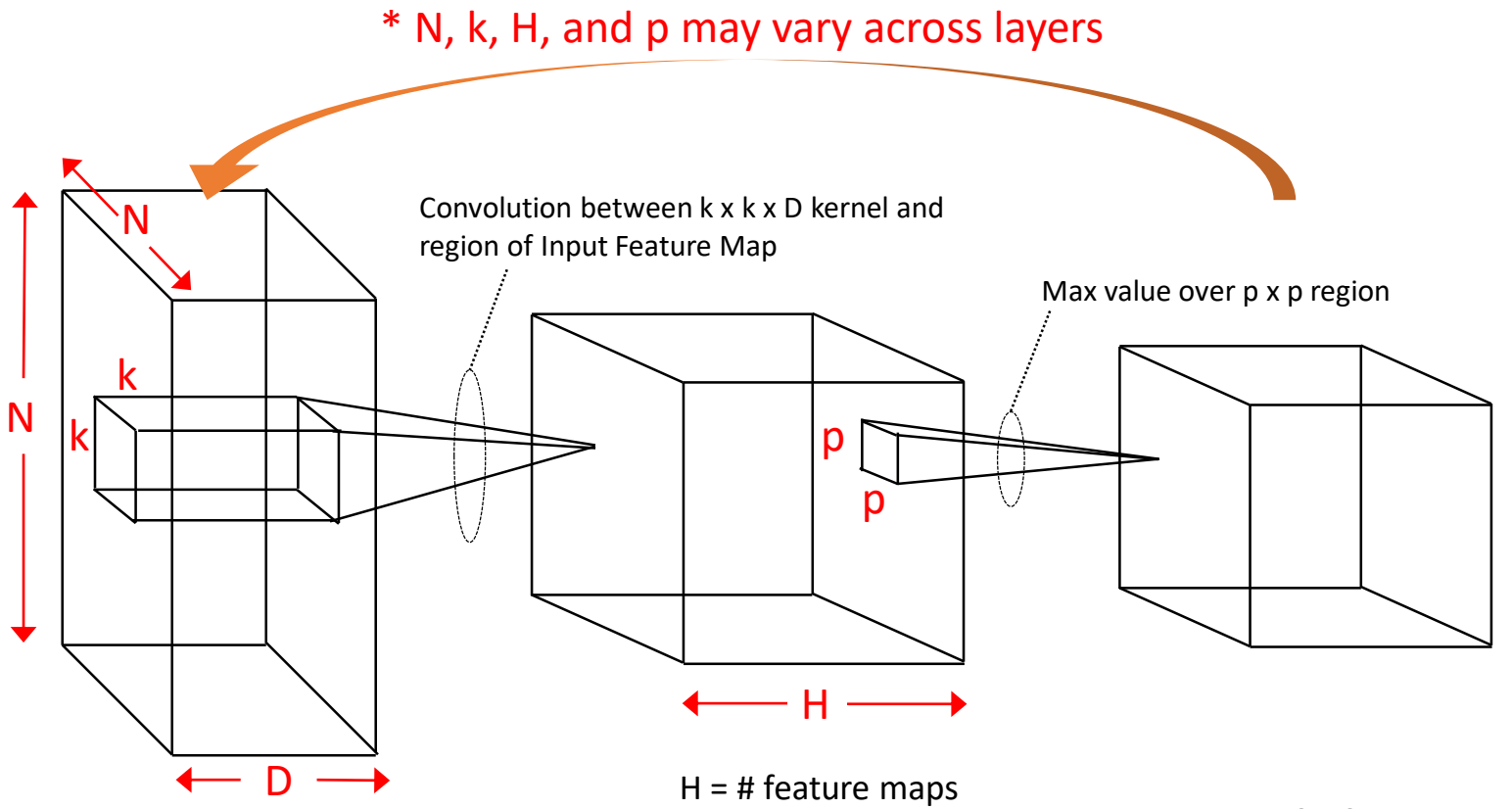
Deep Convolutional Neural Networks

- AlexNet

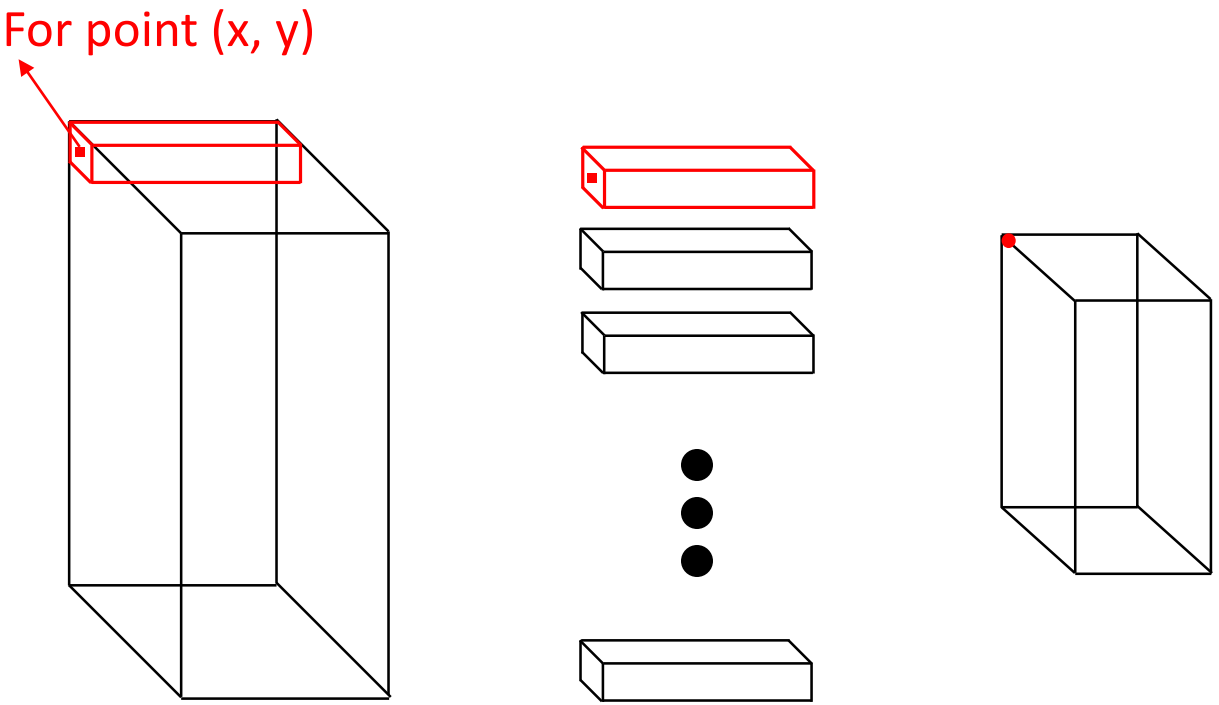


A. Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks," NIPS 2012

Convolution Layer



3-D Convolution (1/5)

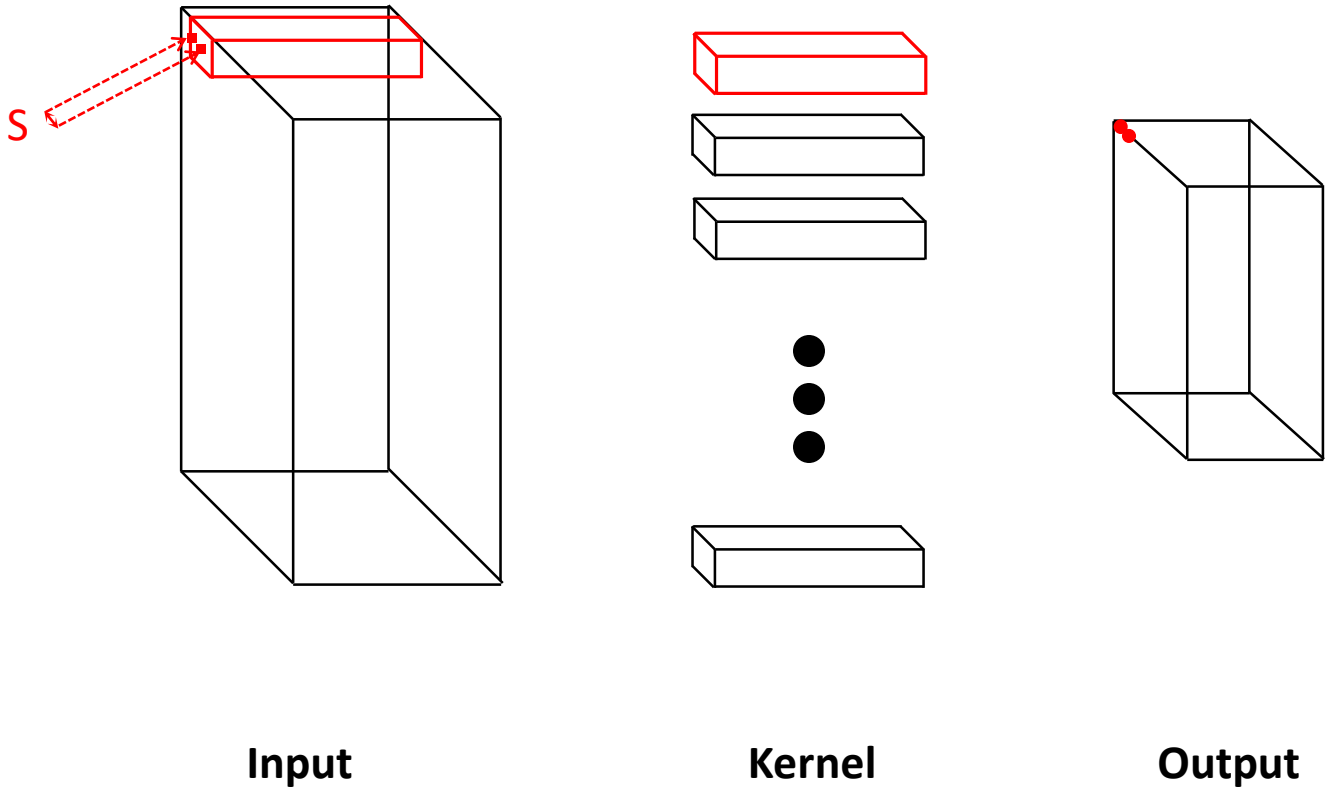


Iterate *i*, *j* and *d*

$$\sum_{d=0}^{D} \sum_{j=-k/2}^{+k/2} \sum_{i=-k/2}^{+k/2} I(x+i, y+j, d) * K(i, j, d)$$

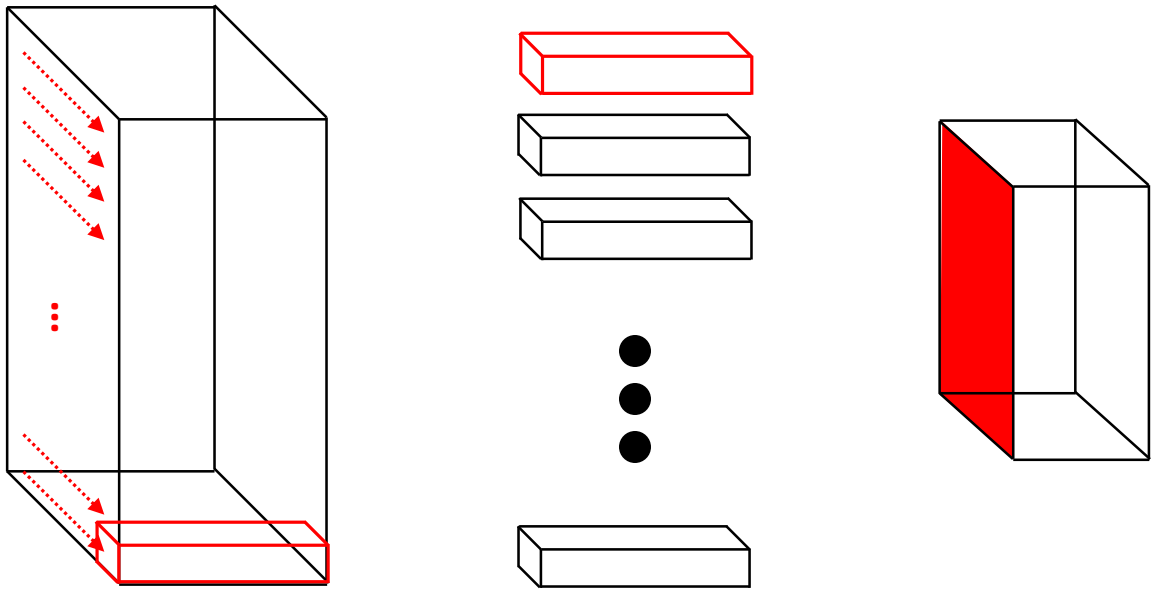
Input **Kernel** **Output**

3-D Convolution (2/5)



Sliding an input by stride S

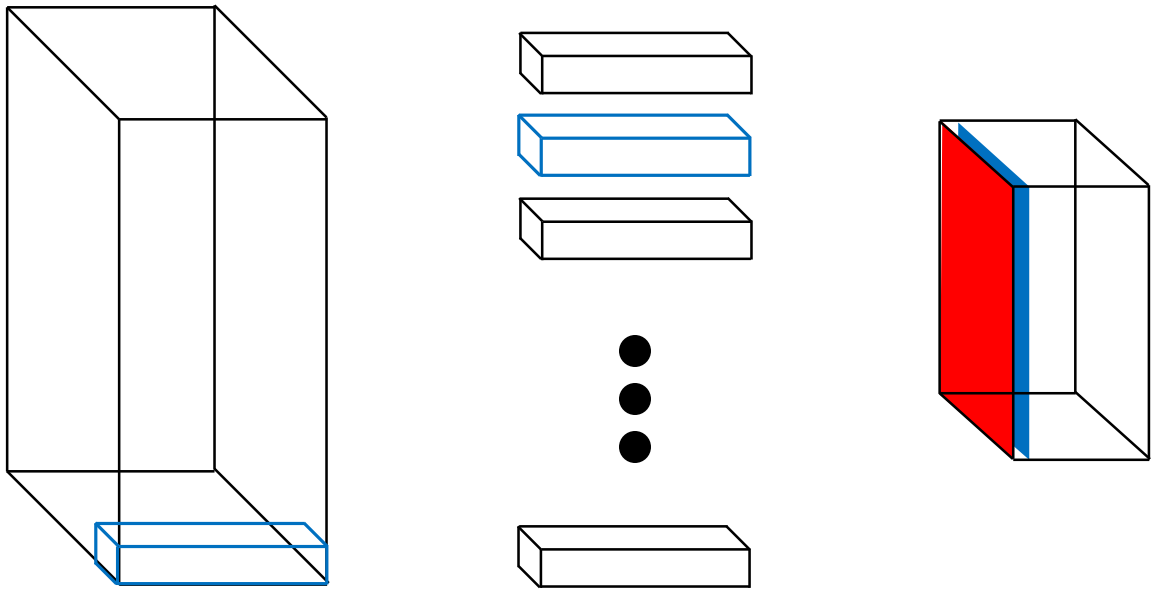
3-D Convolution (3/5)



Input **Kernel** **Output**

Keep sliding until it covers a whole input volume and produce an output feature map (2-D)

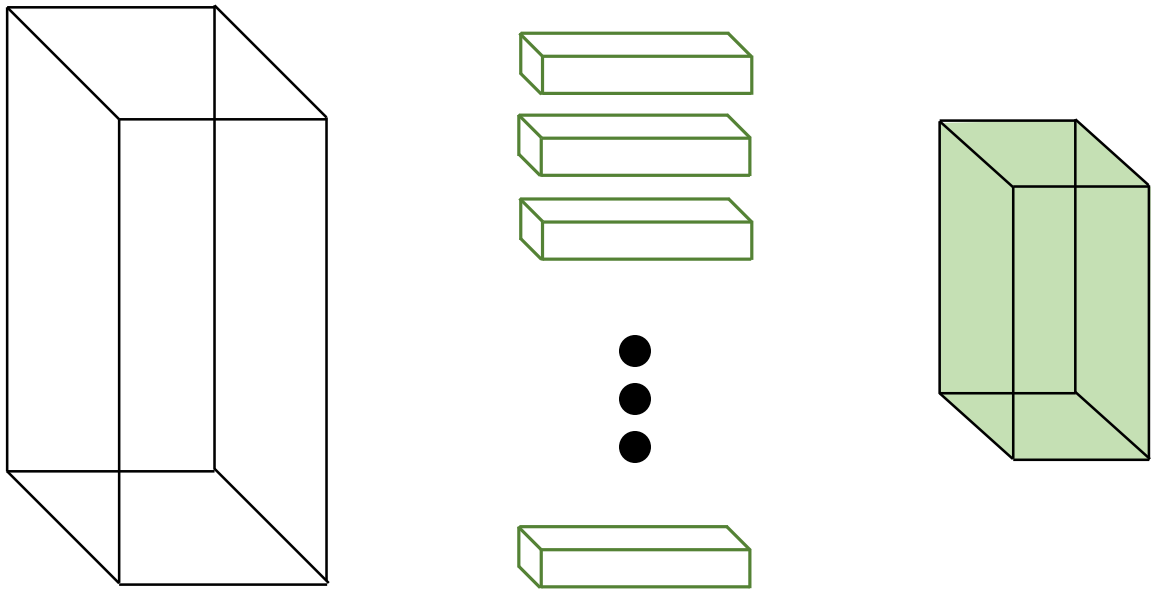
3-D Convolution (4/5)



Input **Kernel** **Output**

Repeat this for the next kernel and generate the next output feature map

3-D Convolution (5/5)



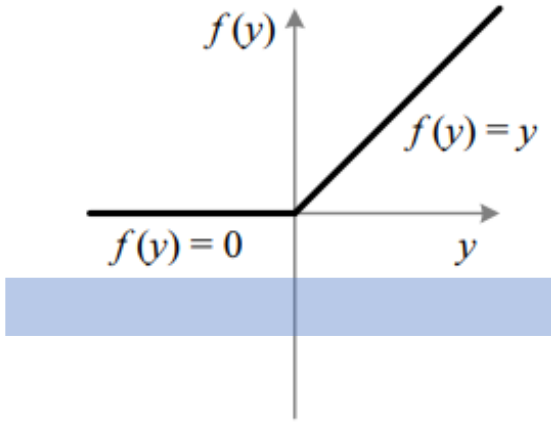
Input **Kernel** **Output**

Iterate entire kernels to
produce an output volume (3-D)

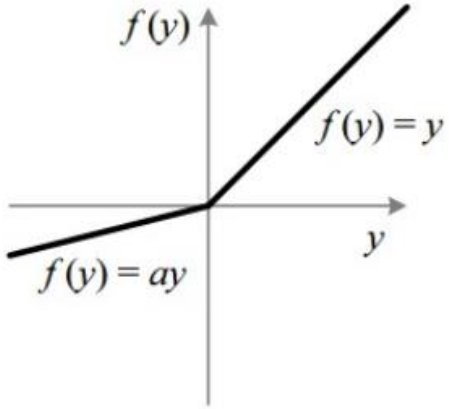
ReLU (Non-linearity)

- Rectified Linear Unit
- Non-linear activation: f(x) = max(0, x)

15	20	-15	9
19	-10	25	102
-3	115	18	11
5	78	7	-40



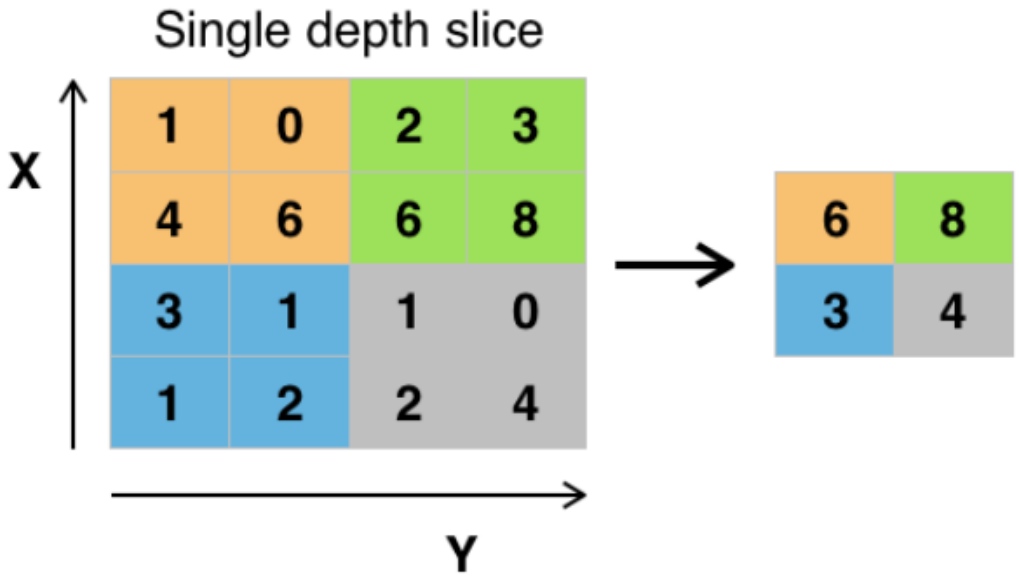
15	20	0	9
19	0	25	102
0	115	18	11
5	78	7	0



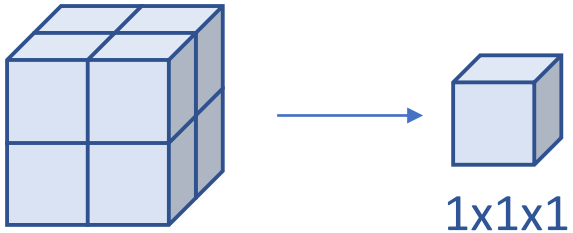
Leaky ReLU

Pooling (Subsampling)

- A form of non-linear down-sampling
- Noise reduction, computation reduction, reduce overfitting



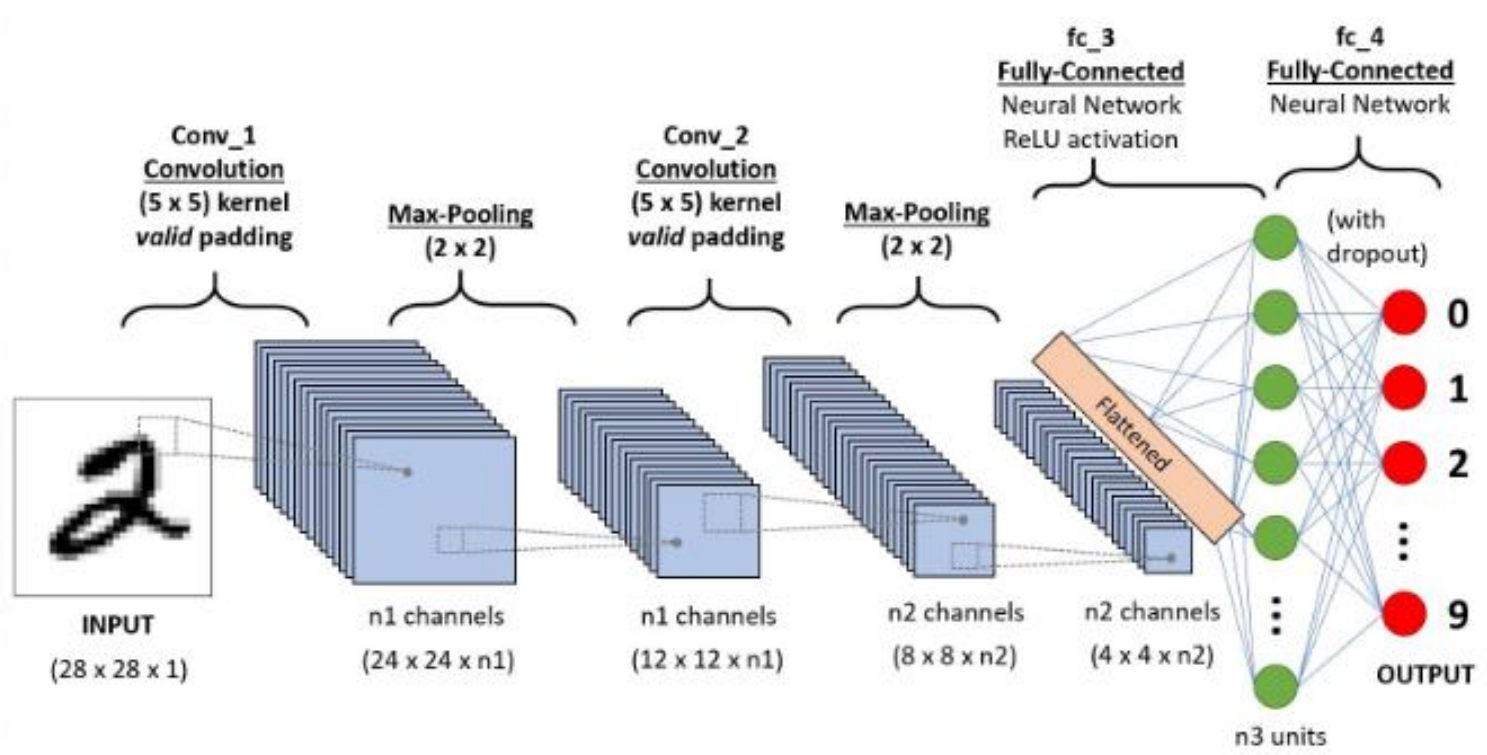
2-D Max Pooling



3-D Max Pooling

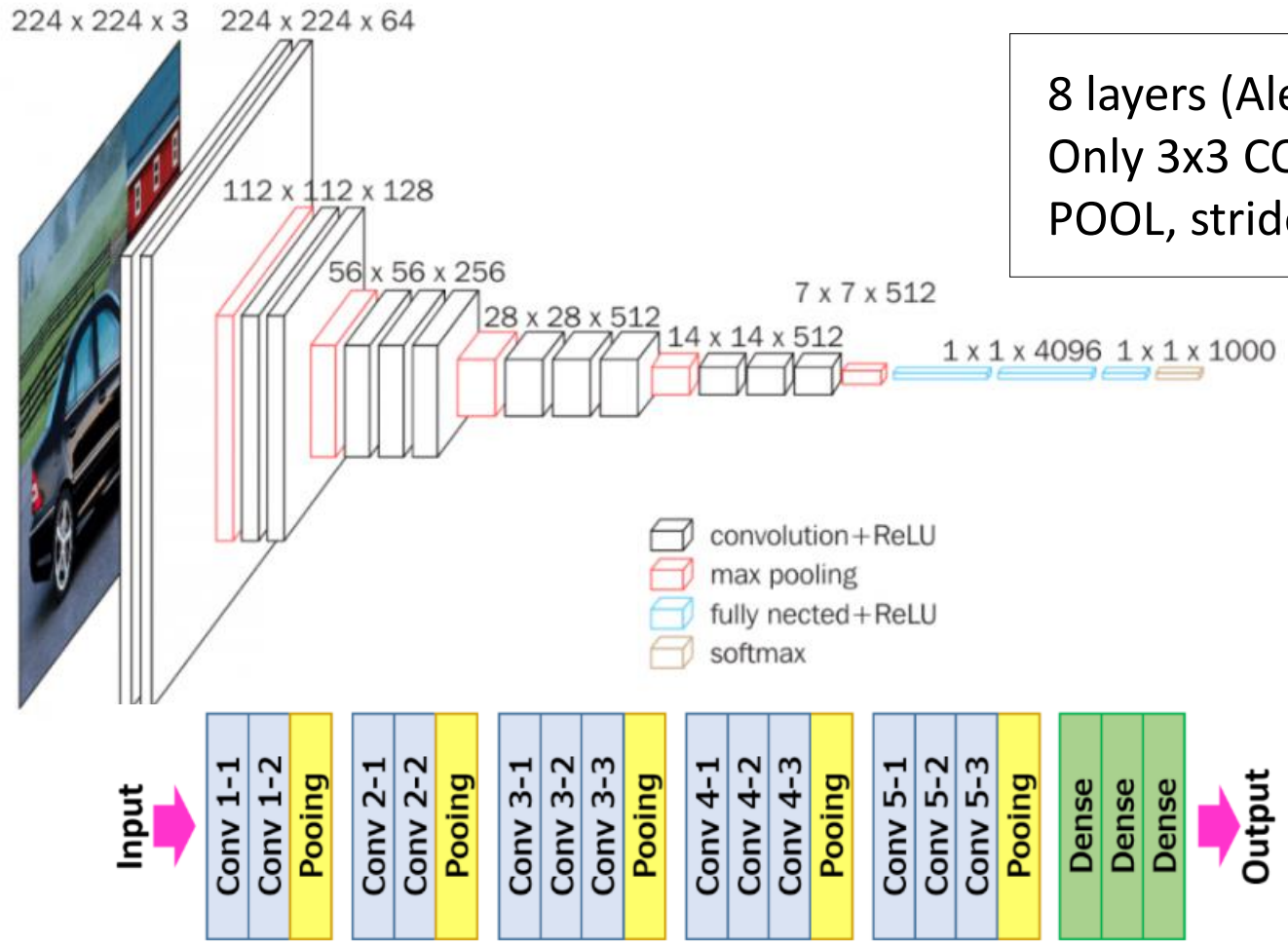
Fully Connected Layer

- Flatten output feature maps into linear neurons
- Apply MLP for classification → matrix-vector multiplication



VGGNet

- Small filters, deeper networks

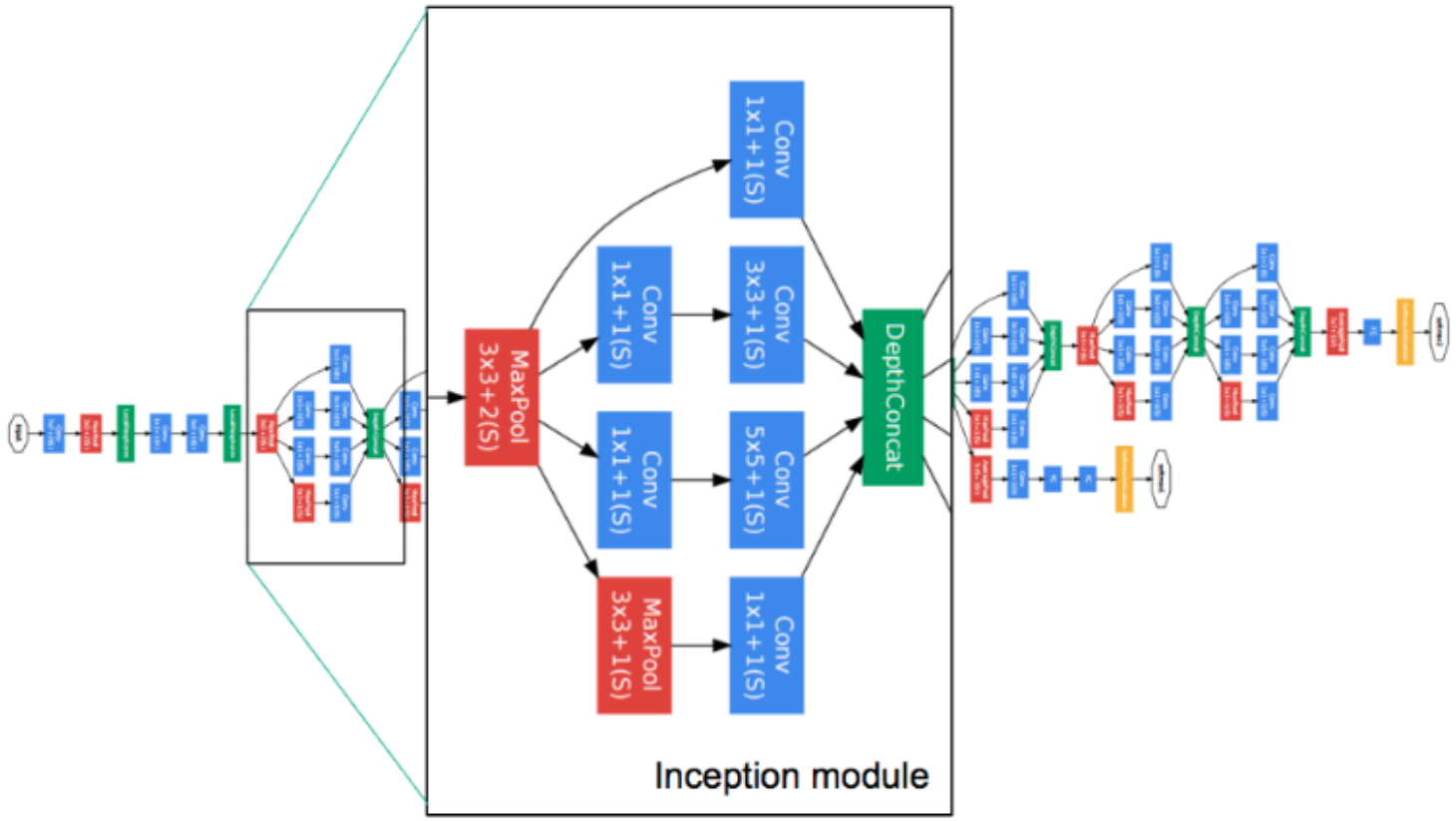


8 layers (AlexNet) -> 16-19 layers
Only 3x3 CONV, stride 1 and 2x2 MAX POOL, stride 2

K. Simonyan et al, "Very Deep Convolutional Networks for Large-Scale Image Recognition," ICLR 2015

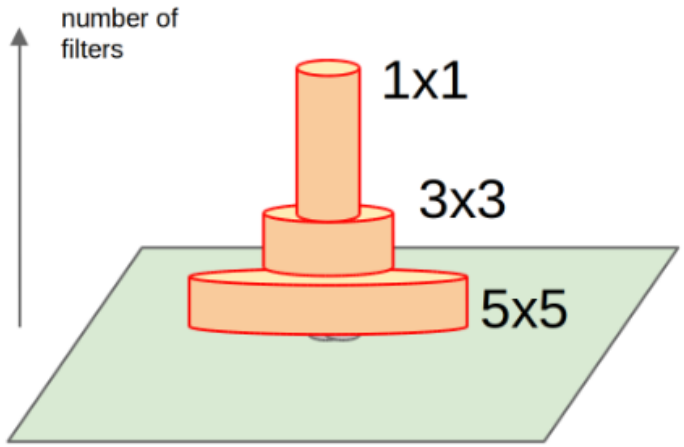
GoogleNet

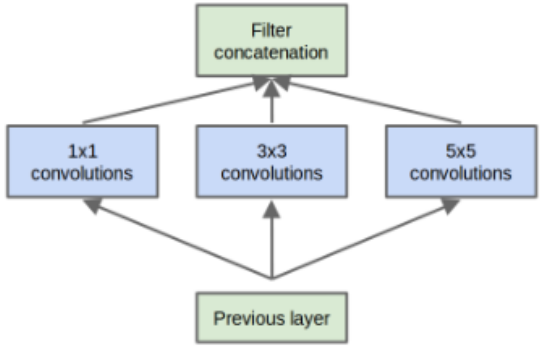
- Winner of ILSVRC 2014 (6.67% Top-5 error rate)
- 22 layer with efficient “inception” module, no FC layers
- Reduced # of parameters from 60M to 5M (12x less than AlexNet)

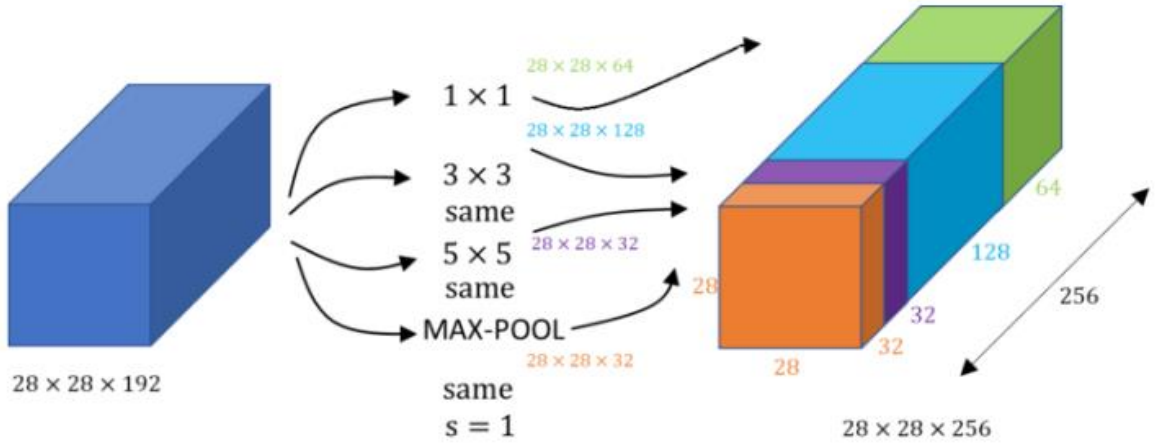


Inception Module

- Inception module handles multiple object scales with different kernel size (1x1, 3x3, and 5x5)
- All the results will be concatenated in depth direction and sent to the next layer

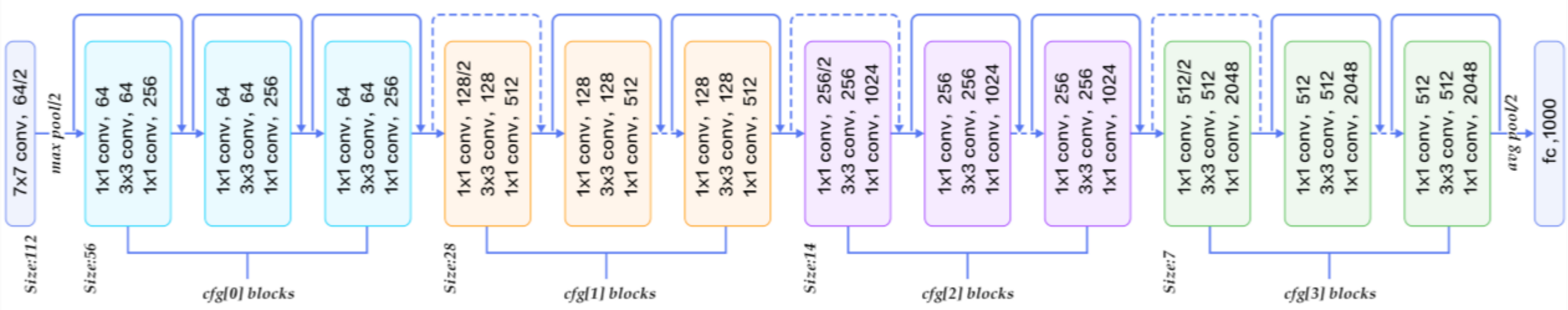






Residual Net (ResNet)

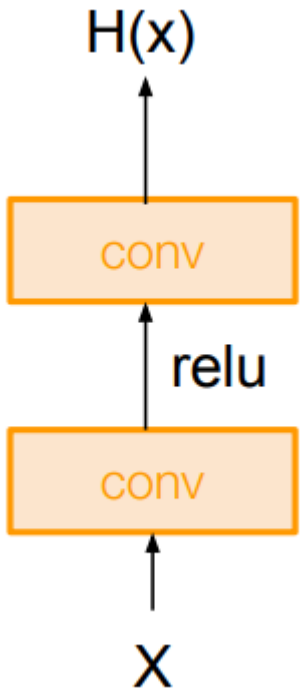
- Winner of ILSVRC 2015 (3.57% Top-5 error rate)
- 152 layers with skip connections



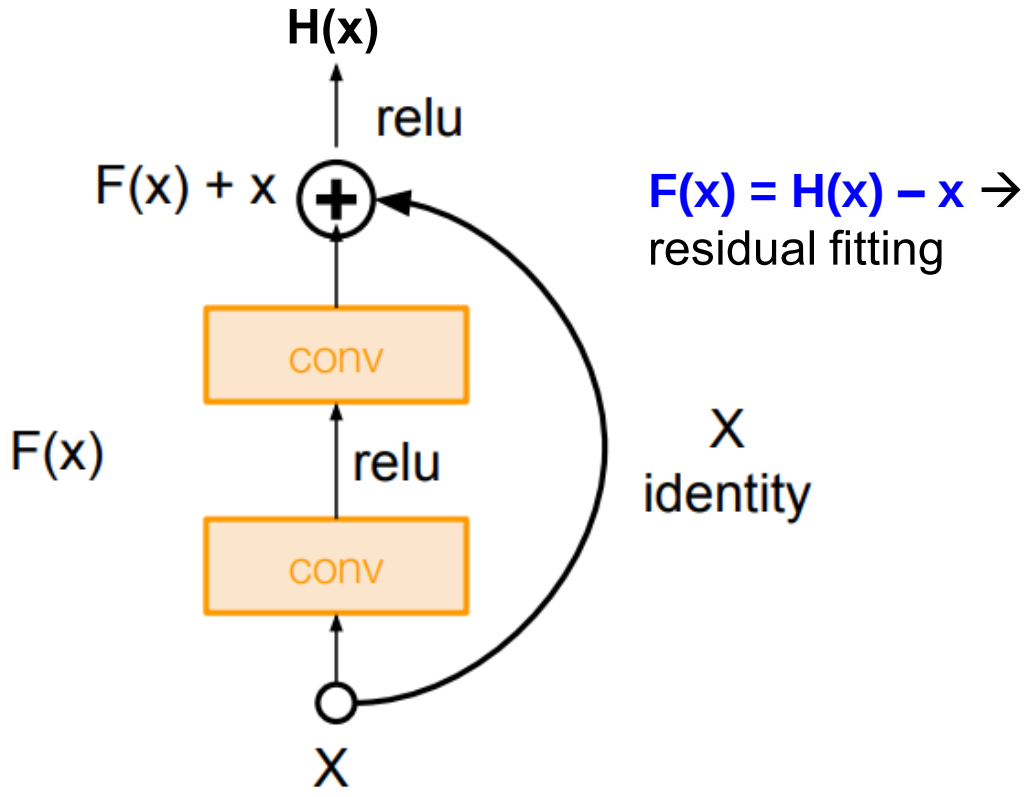
K. He et al, “Deep Residual Learning for Image Recognition,” CVPR 2016

Plain Network vs Skip Connection

- Deeper model is harder to optimize due to vanishing/exploding gradient problem
- Skip connection gives identity to mitigate vanishing gradient problem



"Plain" layers



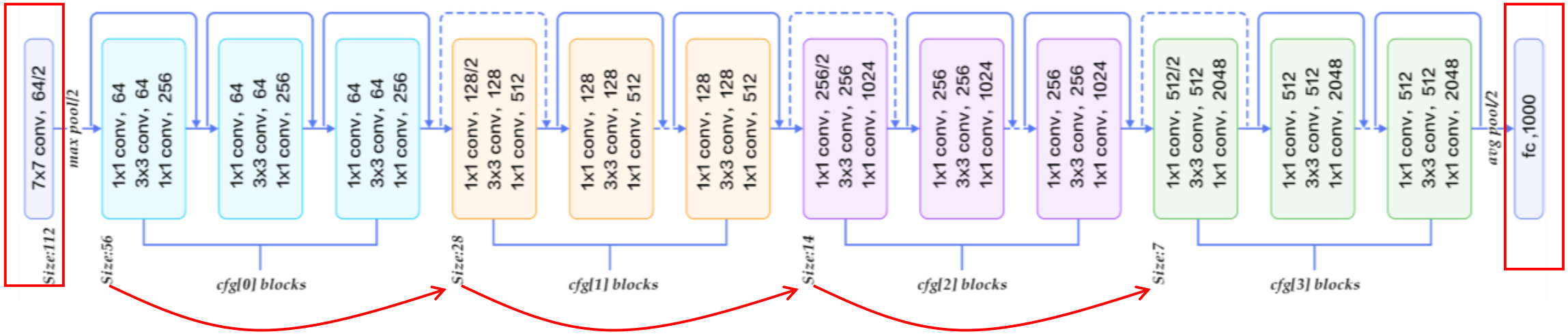
F(x) = H(x) – x →
residual fitting

Residual block

ResNet Architecture

Wide kernel only at the input

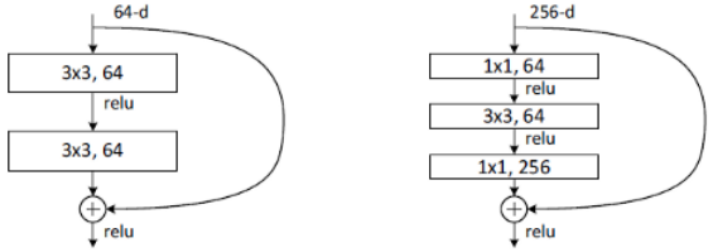
FC only at the input



Periodically down-sample with stride 2 and double # of filters

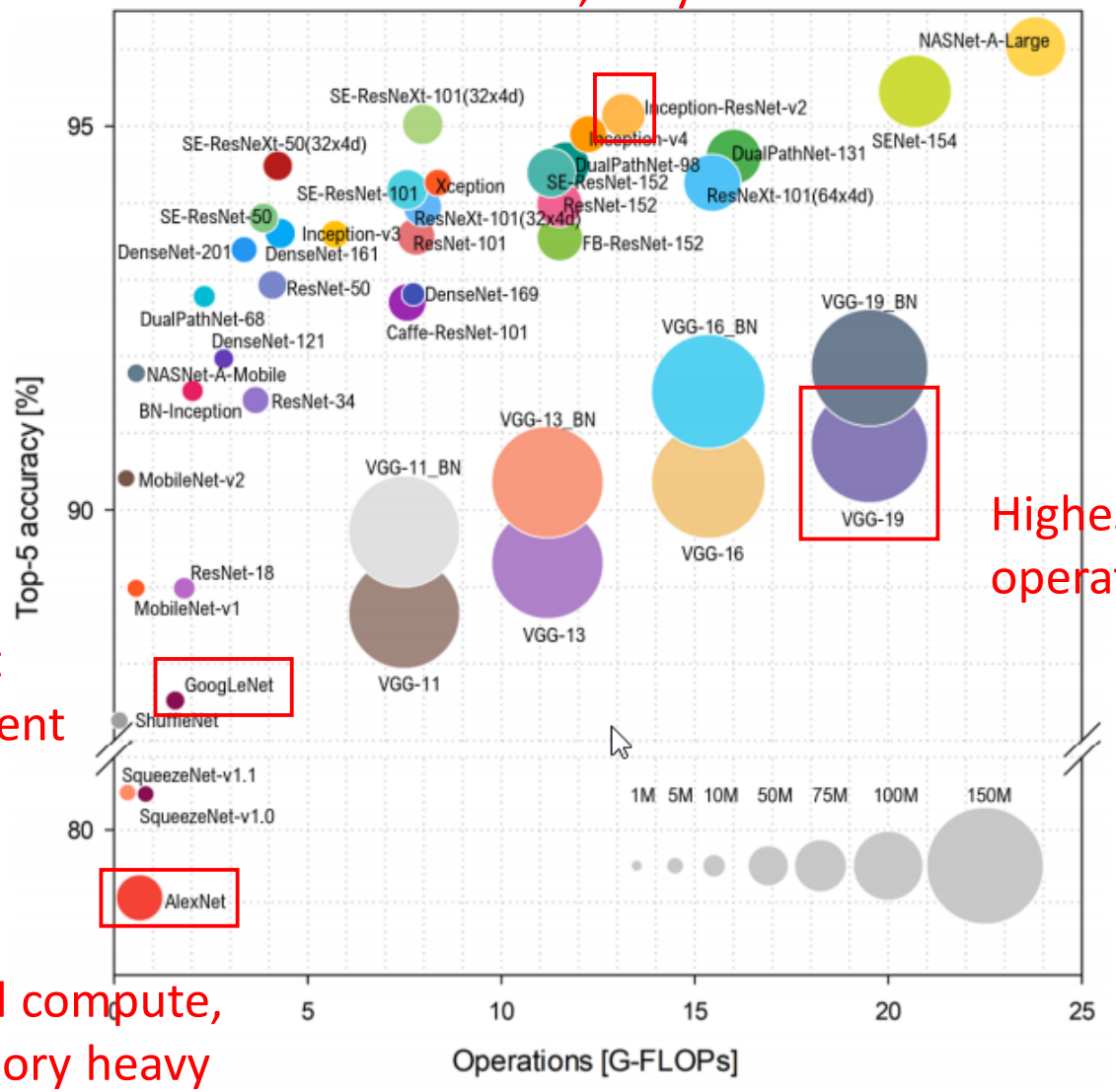
layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Bottleneck design like Inception module



CNN Model Benchmark

Efficient, very accurate



Most efficient

Small compute, memory heavy

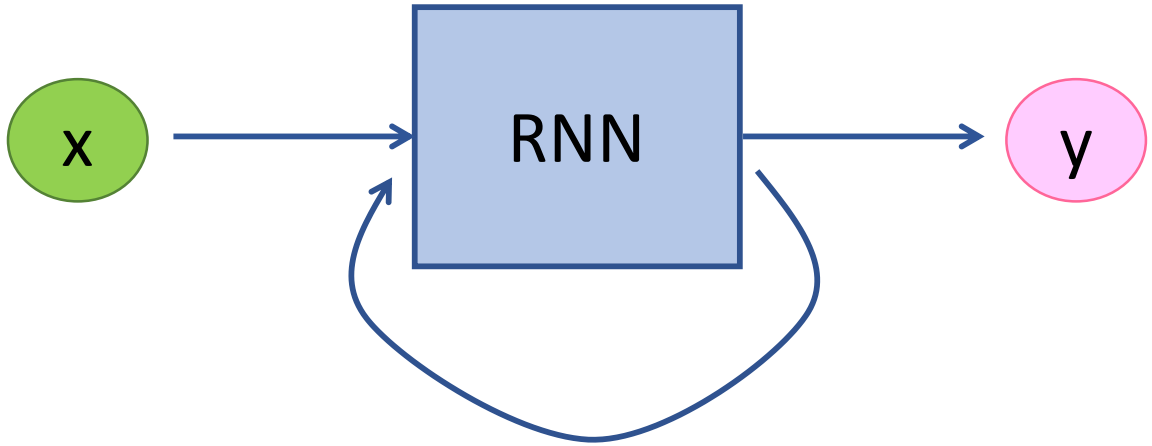
Highest memory, most operations

- Computation: ~25 Giga Floating Point Operation Per Second
- Model size: ~150M parameters
- Top-5 Accuracy: > 95%

S. Bianco et al, "Benchmark Analysis of Representative Deep Neural Network Architectures," IEEE Access

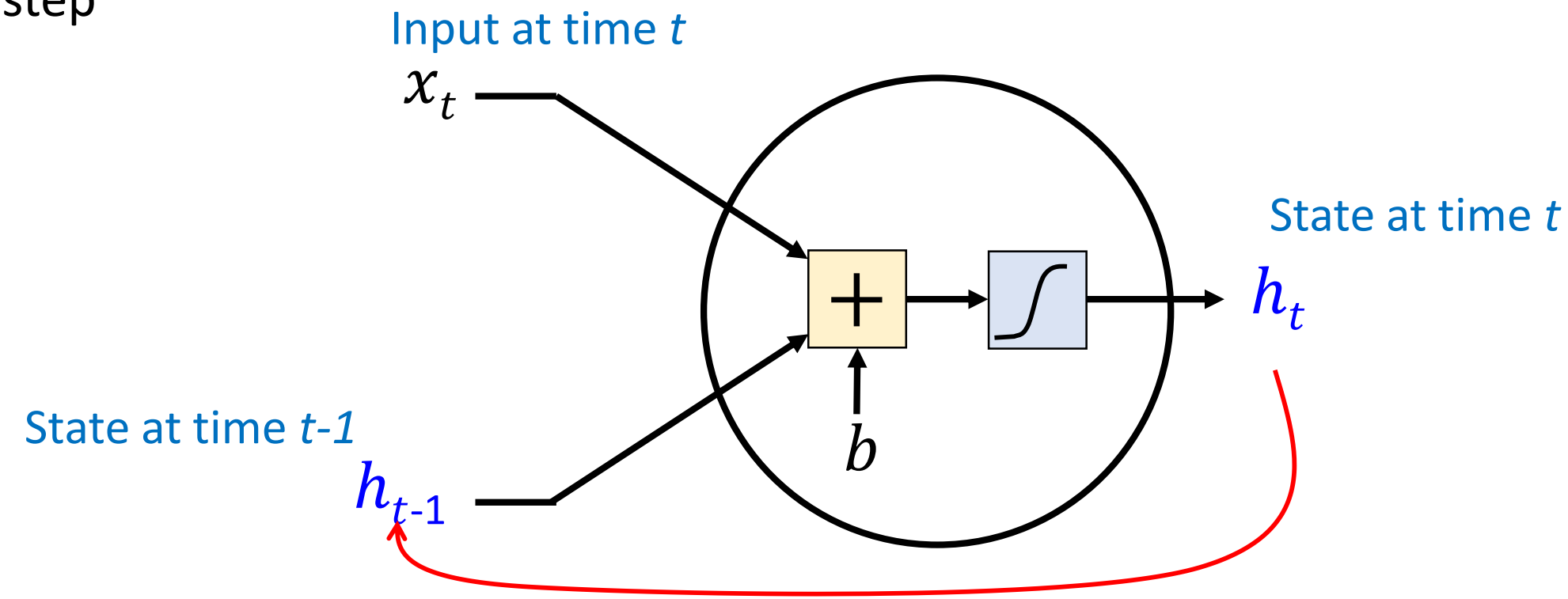
Recurrent Neural Network (RNN)

- A class of neural networks where connections between nodes form a directed graph along a temporal sequence
- Designed to recognize data's sequential characteristics and predict the next
- Good for speech recognition and natural language processing



Recurrent Neural Network

- Process a sequence of vectors x by applying a recurrence formula at every time step



$$h_t = f(W_h h_{t-1} + W_x x_t)$$

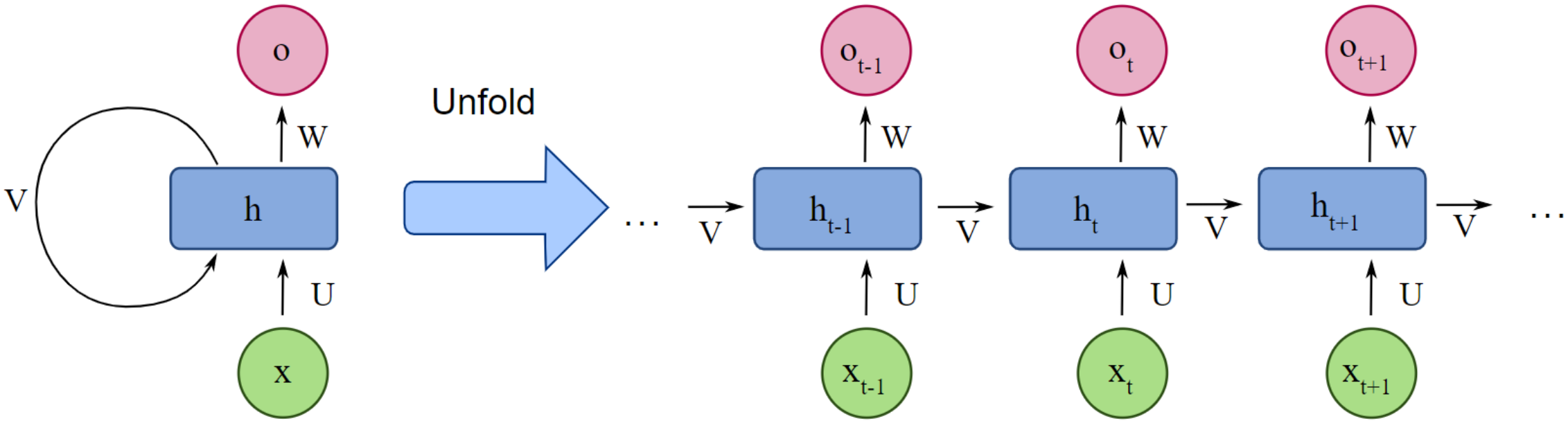
New state

old state

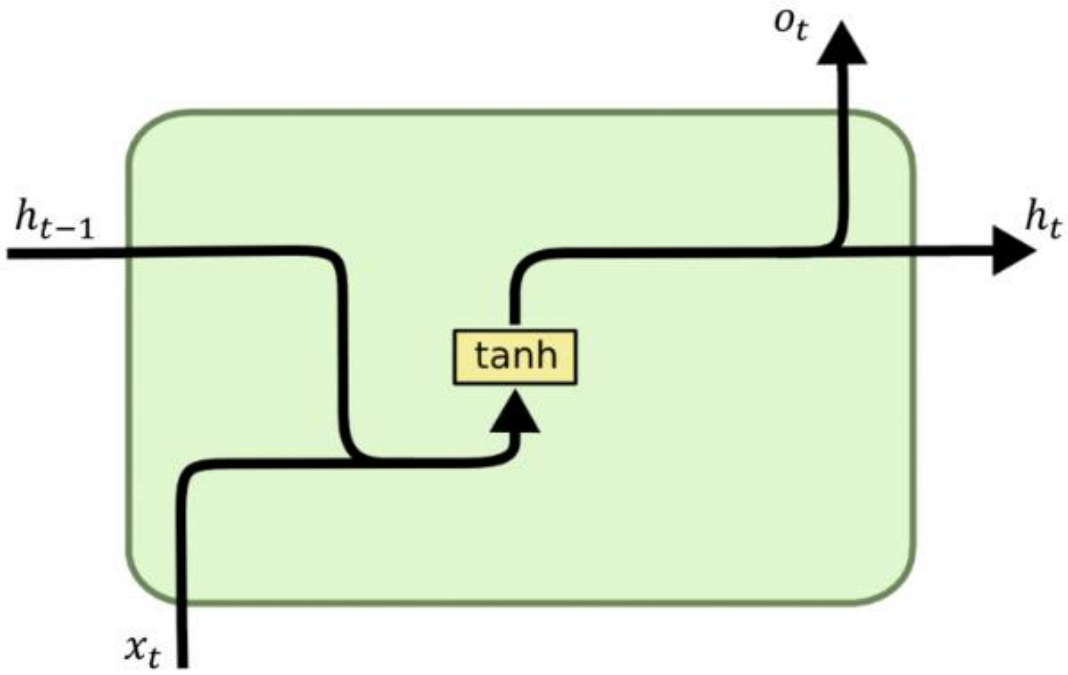
Input vector at some time stamp

RNN Computation

- RNN requires Matrix-vector multiplications like MLP, but with additional weight matrix due to feedback loop and multiple time steps



RNN Computation



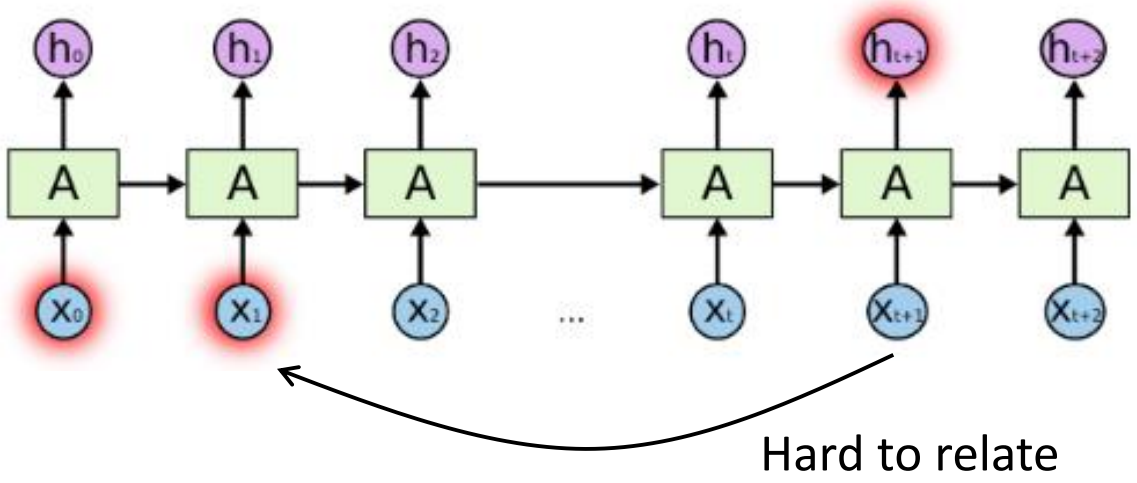
Usually tanh is used for activation

$$h_t = \sigma_h(U_h x_t + V_h h_{t-1} + b_h)$$

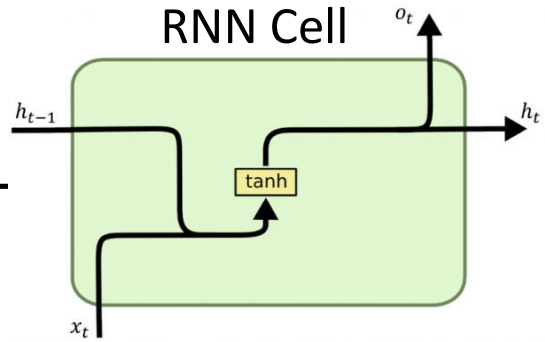
$$o_t = \sigma_o(W_h h_t + b_o)$$

RNN Problem

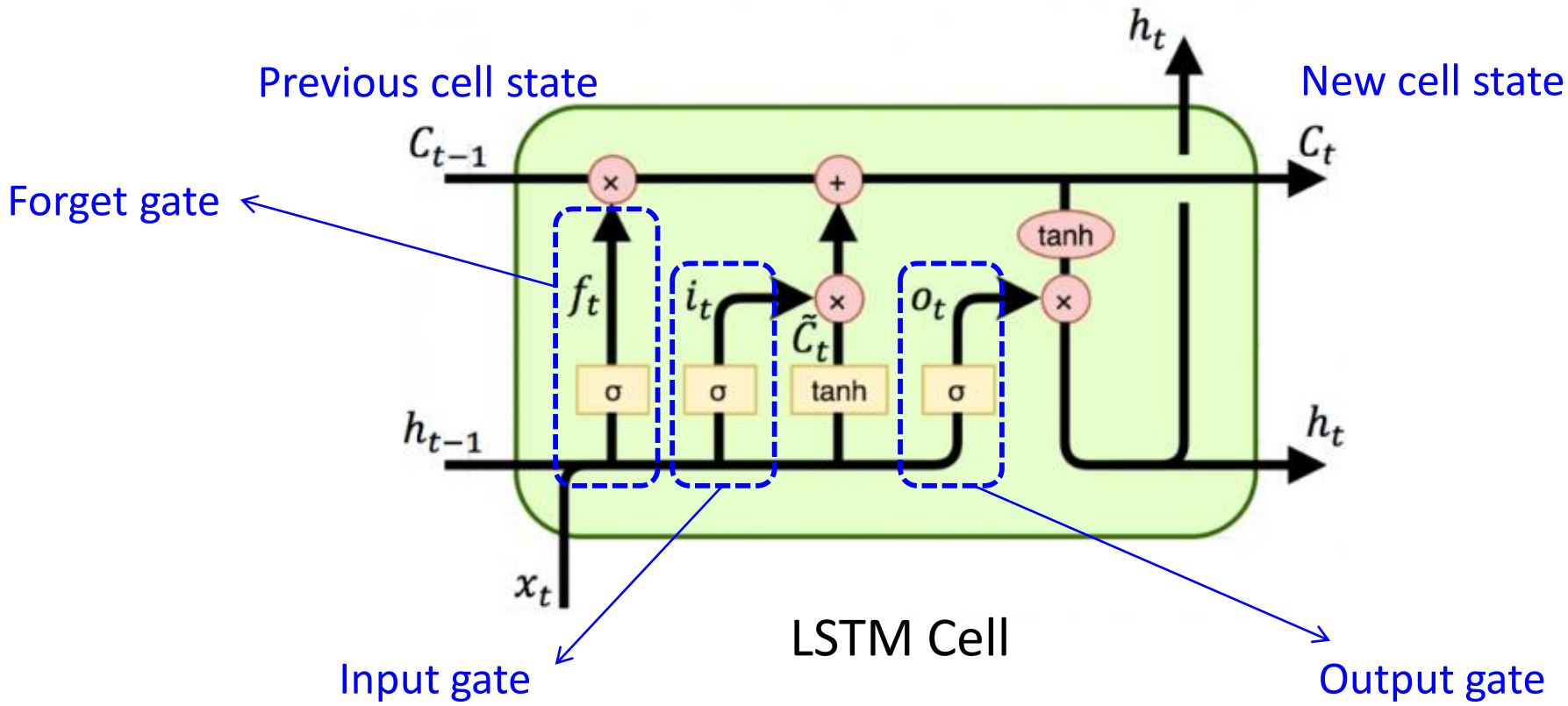
- We expect a temporal prediction from RNN, but it has a problem in long-term dependency due to vanishing gradient problem
- “I grew up in Korea. I can speak fluent <?>”



Long Short Term Memory (LSTM)

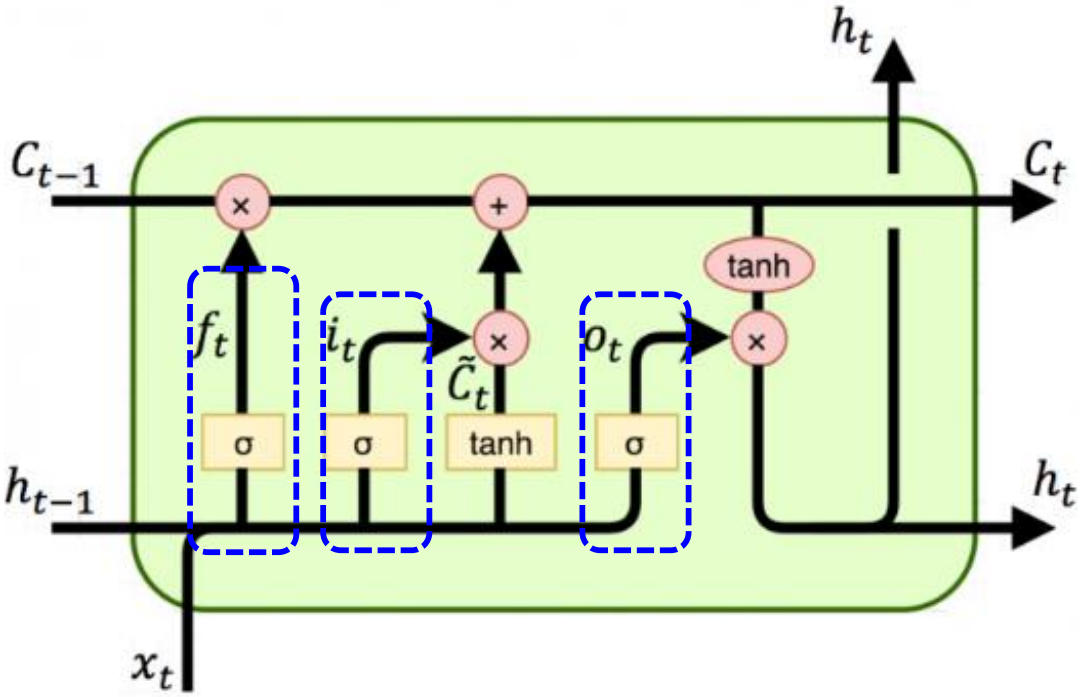


- A new type of RNN to solve vanishing gradient problem
- Multiple switch gates with bypass units → remember for longer time stamps



S. Hochreiter and J. Schmidhuber, "Long Short Term Memory", Neural Computation 1997

LSTM Computation



Forget, input gate

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$

Cell state

$$\tilde{c}_t = tanh(U_c x_t + W_c h_{t-1} + b_c)$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{c}_t$$

Output

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$

$$h_t = o_t * \tanh(C_t) = y_t$$

Training

- Supervised learning: given training data consisting of pairs of inputs and outputs, find all the network weights and biases which correctly match them
- Practically, define the cost function and update weights and biases to minimize it

Final output of neural network

Training sample output

$$\underset{weights,\,biases}{argmin}\left(C = \frac{1}{2}\left\|\boldsymbol{y^o} - \boldsymbol{y_t}\right\|^2\right)$$

Backpropagation

- Initialize network weights/biases
- For each training sample:
  1. **Forward propagation**: an input vector goes through neural network
  2. Compute error signal at output
  3. **Backpropagate** error signals through network
  4. **Update weights/biases** in order to decrease the difference between the predicted output and ground truth (=training set)
- Repeat until network is trained

Gradient Descent

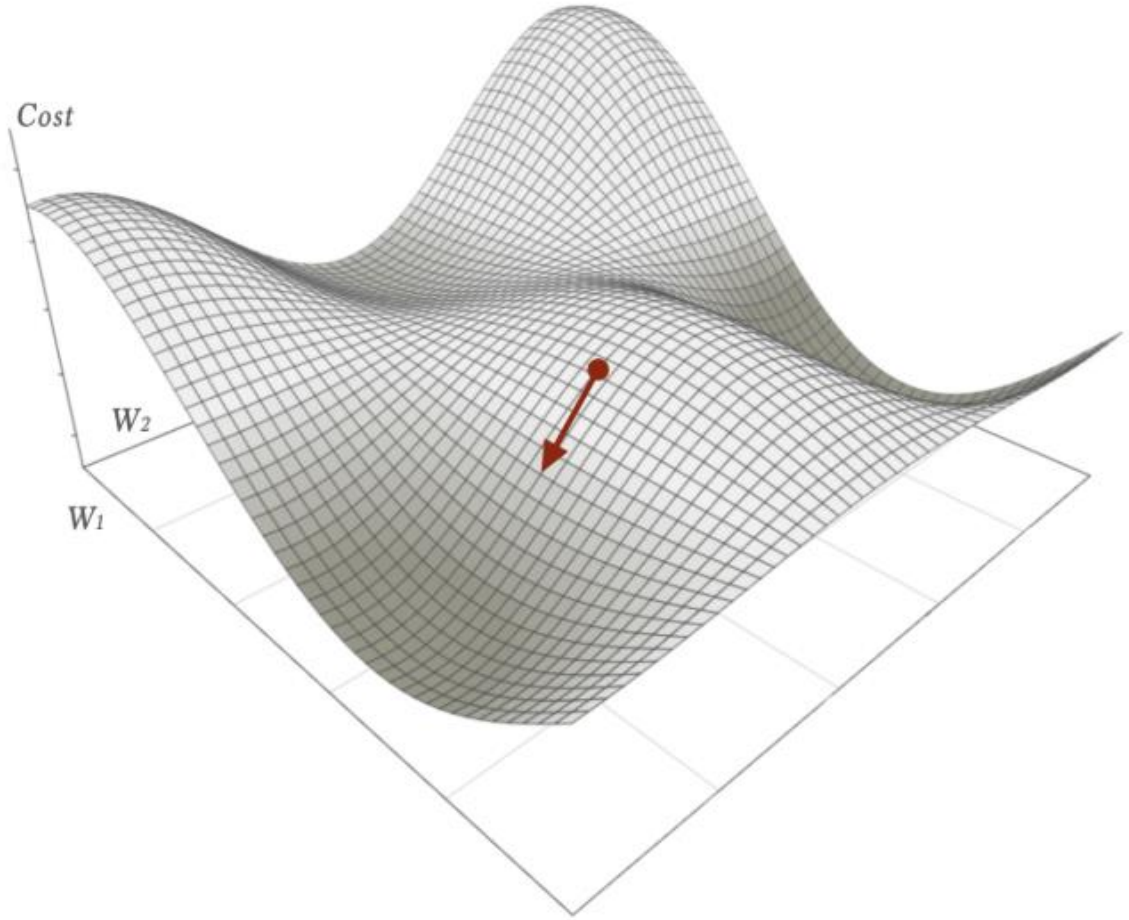
- An optimization method to minimize a cost function by iteratively moving in the direction of **steepest descent** as defined by the negative of the **gradient.**

$$W^+ = \mathrm{W} - \eta \nabla C$$

Learning rate

Gradient operator

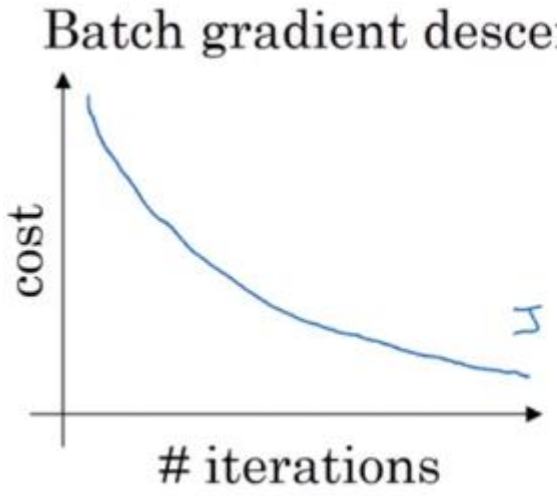
$$\begin{bmatrix} w_1^+ \\ w_2^+ \\ \vdots \\ w_n^+ \end{bmatrix} = \begin{bmatrix} \mathrm{w}_1 \\ \mathrm{w}_2 \\ \vdots \\ \mathrm{w}_n \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial \mathrm{C}}{\partial \mathrm{w}_1} \\ \frac{\partial \mathrm{C}}{\partial \mathrm{w}_2} \\ \vdots \\ \frac{\partial \mathrm{C}}{\partial \mathrm{w}_n} \end{bmatrix}$$

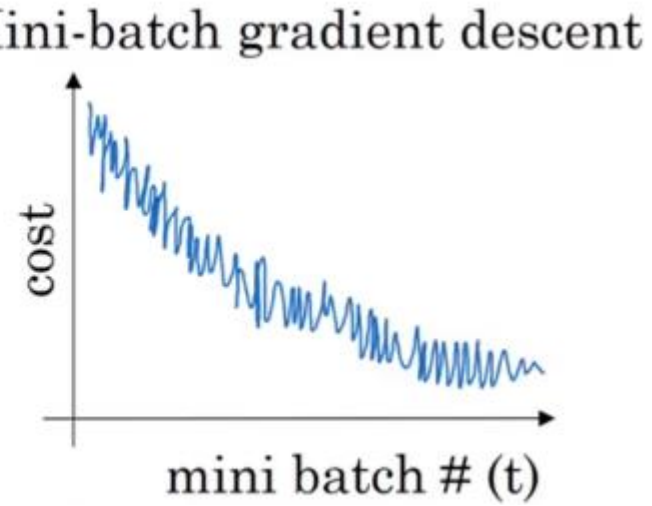


Computing partial derivative of cost function with respect to each w is key!

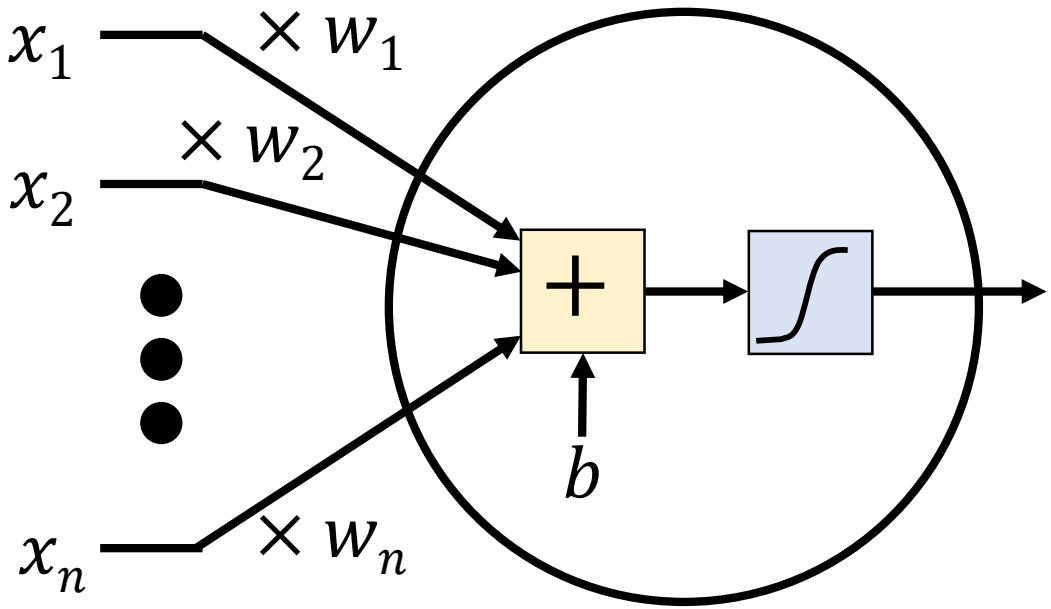
Stochastic Gradient Descent (SGD)

- Batch gradient descent finds the steepest descent for all given data set every step → not feasible for large training data sets in deep learning
- Stochastic process with mini-batching
  - Splits training set into lots of smaller batches and apply gradient descent on each batch one after the other
  - Mini-batch size is important: slow if it is too large, noisy if it is too small





Single Neuron



$$f(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$$

How to compute $\dfrac{\partial C}{\partial w_i}$?

Single Neuron

$$y = f(\underbrace{w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b}) $$

Z: output right before activation

$$\frac{\partial C}{\partial w_i} = \frac{\partial(0.5 * (y - y_t)^2)}{\partial w_i} = (y - y_t) \cdot \frac{\partial(y - y_t)}{\partial w_i}$$

$$= (y - y_t) \cdot \frac{\partial y}{\partial w_i}$$

Yt: training sample, irrelevant to Wi

$$= (y - y_t) \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w_i}$$

Chain rule

$$= (y - y_t) \cdot f'(z) \cdot x_i$$

Single Neuron

Gradient = error * derivative of activation * input

$$\frac{\partial C}{\partial w_i} = (y - y_t) \cdot f'(z) \cdot x_i = \delta \cdot x_i$$

Weight update from gradient descent is $W^+ = \mathrm{W} - \eta \nabla C$

$$w_i^+ = w_i - \eta \cdot \delta \cdot x_i$$

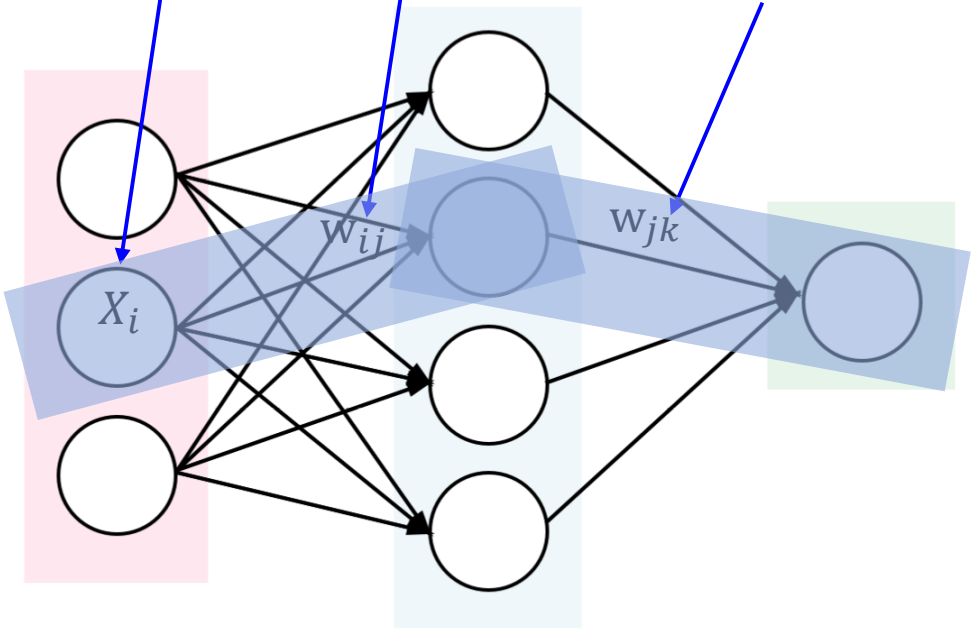
Multi-Layer Neurons

- Looking at only a single path

Xi: input neuron i

Wij: weight from input i to hidden layer neuron j

Wjk: weight from hidden layer neuron j to output k



$$\boldsymbol{y = f(W_{jk} \cdot f\left(W_{ij}X_i + b_j\right) + b_k)}$$

f: Non-linear activation function

Multi-Layer Neurons

Hj: output of hidden layer

$$y = f(W_{jk} \cdot f(W_{ij}X_i + b_j) + b_k)$$

Zk: output right before activation

$$\frac{\partial C}{\partial w_{jk}} = \frac{\partial(0.5 * (y - y_t)^2)}{\partial w_{jk}} = (y - y_t) \cdot \frac{\partial(y - y_t)}{\partial w_{jk}}$$

$$= (y - y_t) \cdot \frac{\partial y}{\partial w_{jk}}$$

Yt: training sample, irrelevant to Wjk

$$= (y - y_t) \cdot \frac{\partial y}{\partial z_k} \cdot \frac{\partial z_k}{\partial w_{jk}}$$

Chain rule

$$= (y - y_t) \cdot f'(z_k) \cdot h_j$$

Multi-Layer Neurons

Gradient = error * derivative of activation * output of previous layer

$$\frac{\partial C}{\partial w_{jk}} = (y - y_t) \cdot f'(z_k) \cdot h_j = \delta \cdot h_j$$

From output to input layer, apply **chain rule** longer

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial (0.5 * (y - y_t)^2)}{\partial w_{ij}} = (y - y_t) \cdot \frac{\partial (y - y_t)}{\partial w_{ij}}$$

$$= (y - y_t) \cdot \frac{\partial y}{\partial w_{ij}}$$

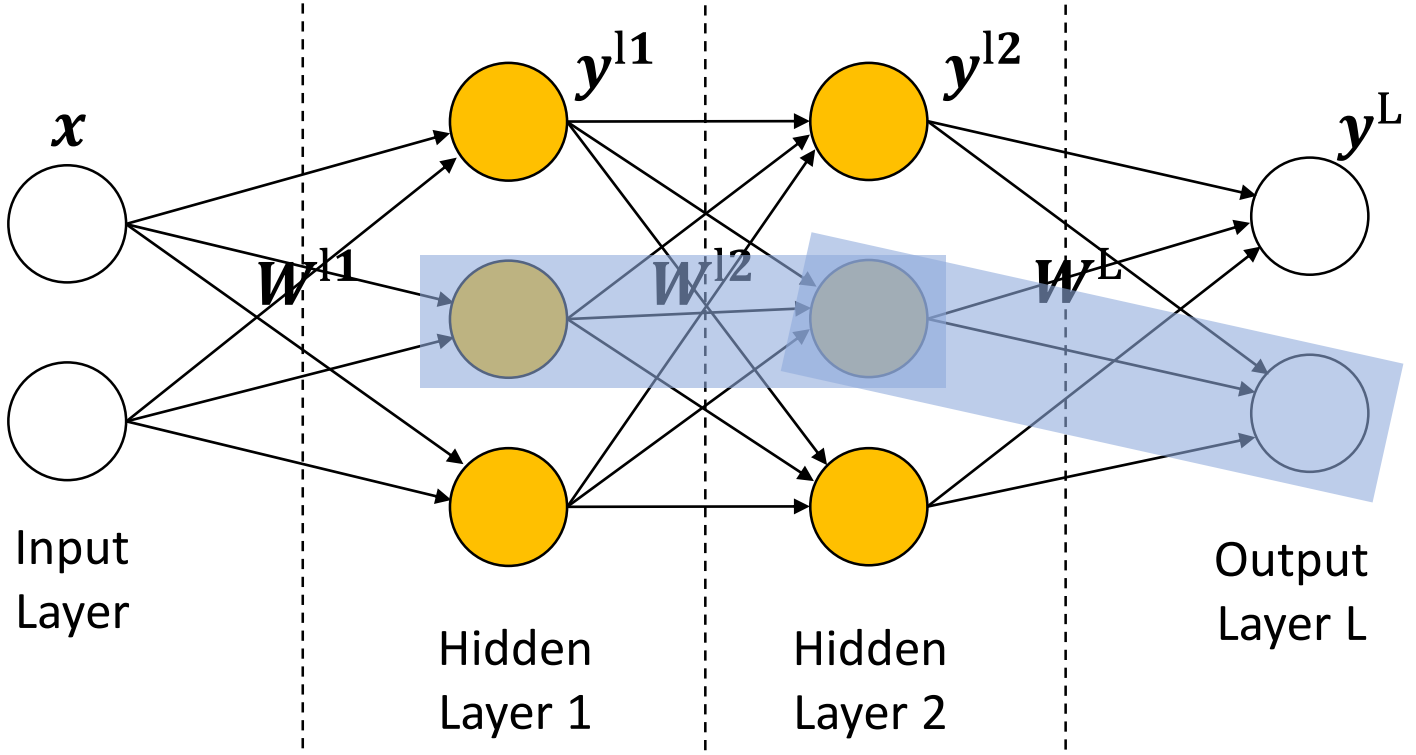
$$= (y - y_t) \cdot \frac{\partial y}{\partial z_k} \cdot \frac{\partial z_k}{\partial h_j} \cdot \frac{\partial h_j}{\partial k_j} \cdot \frac{\partial k_j}{\partial w_{ij}}$$

kj: output right before activation of hj

Chain rule

$$= (y - y_t) \cdot f'(z_k) \cdot w_{jk} \cdot f'(k_j) \cdot x_i$$

Multi-Layer Neurons



	Propagation error	Weight update
For output layer L:	$\boldsymbol{\delta^L = (y - y_t) \cdot f^{L\prime}(z^L)}$	$\boldsymbol{w^{L+} = w^L - \eta \cdot \delta^L \cdot y^{L-1}}$
For layer l < L:	$\boldsymbol{\delta^l = \delta^{l+1} \cdot w^{l+1} \cdot f^{l\prime}(z^l)}$	$\boldsymbol{w^{l+} = w^l - \eta \cdot \delta^l \cdot y^{l-1}}$

General Case

- We've only considered a single path -> need to consider all paths
- Equation for each path will be same except indexes
  - Propagation error will be a vector
  - Weight update will be a matrix

- Propagation error

	Single path	Generalization
Output layer L	$\boldsymbol{\delta^L = (y - y_t) \cdot f^{L\prime}(z^L)}$	$\boldsymbol{\delta^L = \nabla C \odot f^{L\prime}(z^L)}$
Layer l < L	$\boldsymbol{\delta^l = \delta^{l+1} \cdot w^{l+1} \cdot f^{l\prime}(z^l)}$	$\boldsymbol{\delta^l = ((w^{l+1})^{\mathrm{T}} \cdot \delta^{l+1}) \odot f^{l\prime}(z^l)}$

- Weight update

$$\boldsymbol{w^{l+} = w^l - \eta \cdot \delta^l \cdot y^{l-1}}$$

ML Accelerators for Mobile/Edge - I

- DianNao (ASPLOS 2014)
  - T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning,” ASPLOS 2014
- Eyeriss (ISCA 2016, JSSC 2017)
  - Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks,” ISCA 2016
  - Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, “"Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," JSSC 2017

Motivation

- Previous accelerators only focused on computational part
- CNNs and DNNs are characterized by their large size
- **DianNao** – accelerator for large-scale CNNs and DNNs with state-of-the-art machine learning algorithms
  - High throughput: 452GOP/s
  - Small footprint: 3.02mm^2, 485mW
  - 117x faster, 21x more energy efficient than a 2GHz x86 core with 128-bit SIMD extension

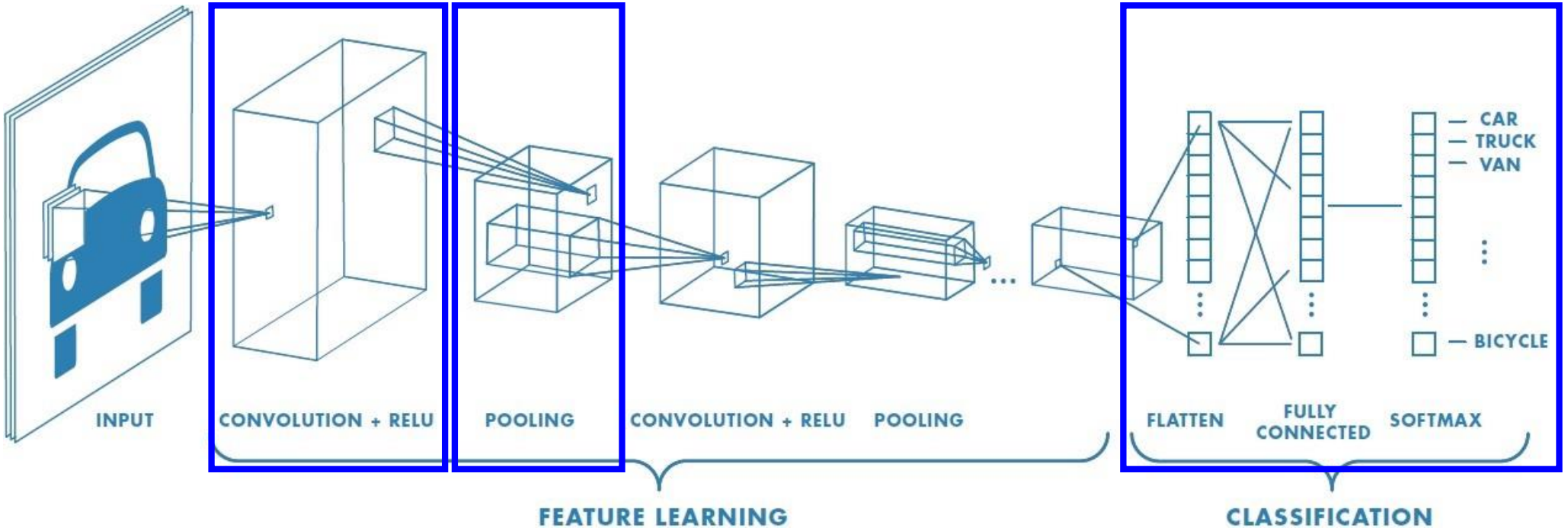
Goal: High Throughput & Small-Footprint

- Improving **memory transfer** behavior
  - Should be first-order concern at Amdahl's law
  - Minimizing memory transfer & perform efficiently

- Inference (feed-forward) vs Training (backward)
  - Focused on feed forward due to technical and market consideration
  - Targeted much larger market of end-users, who needs fast inference with offline training
  - Next version DaDianNao targeted both Inference and Training

Y. Chen, et al. "DaDianNao: A Machine-Learning Supercomputer," Micro 2014

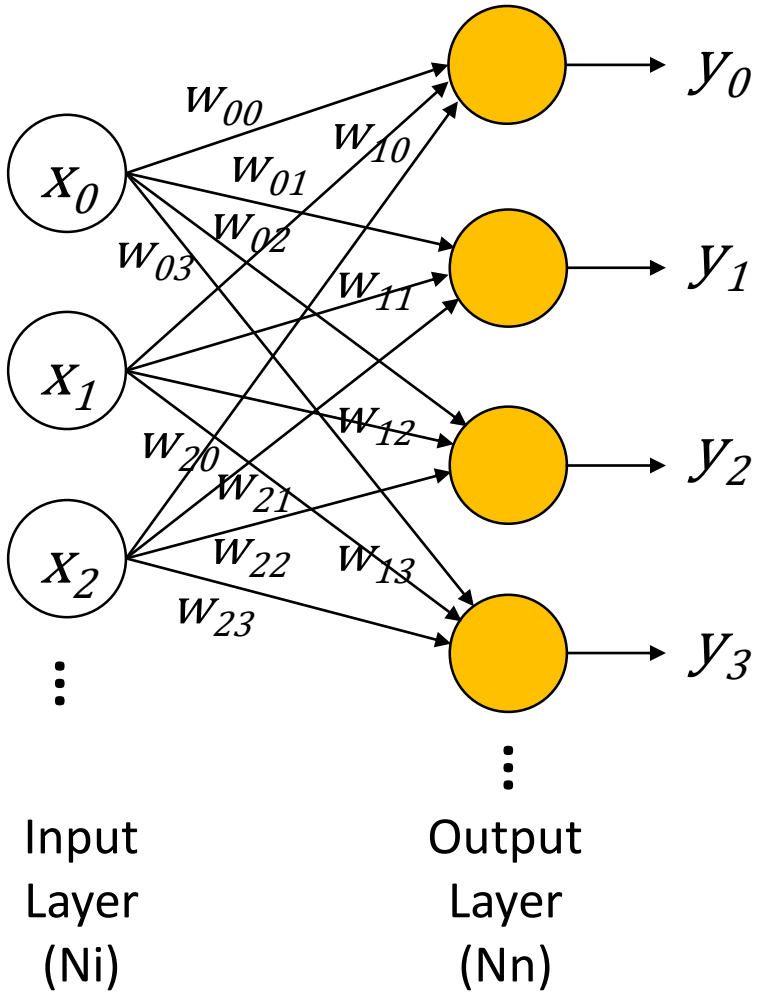
Target Layers



- DianNao supports 3 types of neural network layers
  - Convolution + Activation layer
  - Pooling layer
  - Classifier layer

Classifier Layer

- For Ni input neurons and Nn output neurons..

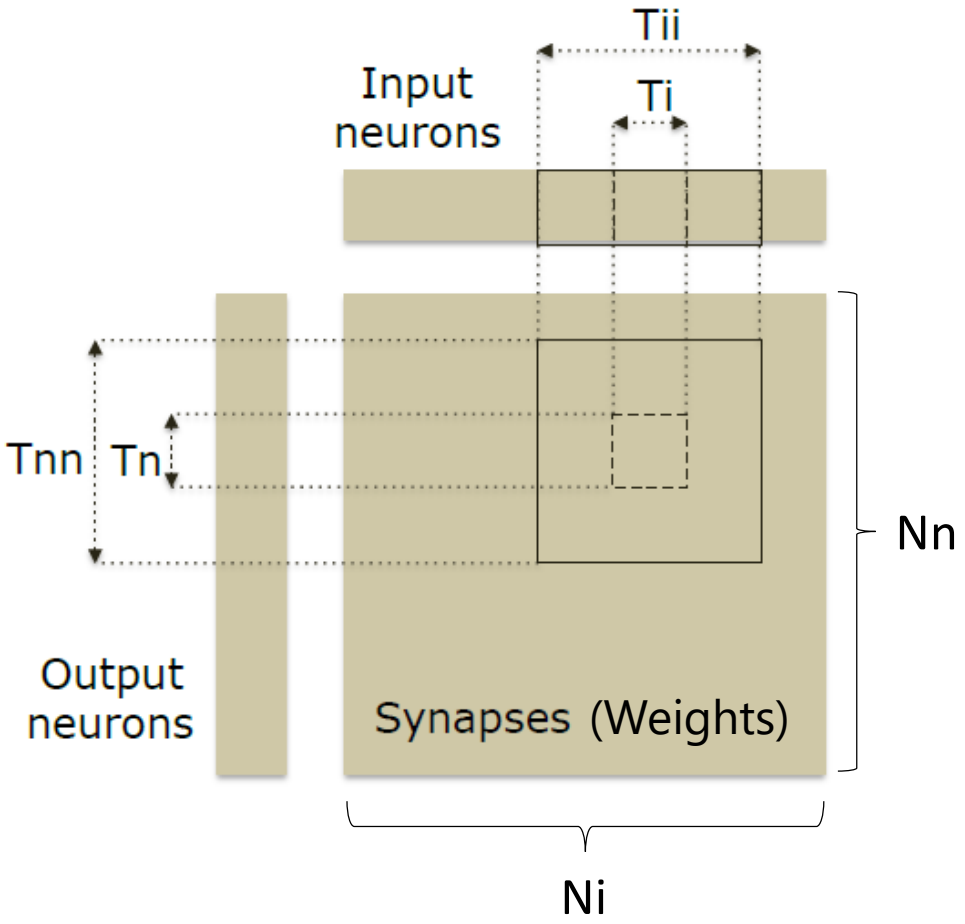


$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

x = Ni x 1 (vector)
y, *b* = Nn x 1 (vector)
W = Nn x Ni (matrix)

Classifier Layer

- Tiling Ni x Nn to Tii x Tnn to reduce working memory size
- Computational unit size is Ti x Tn



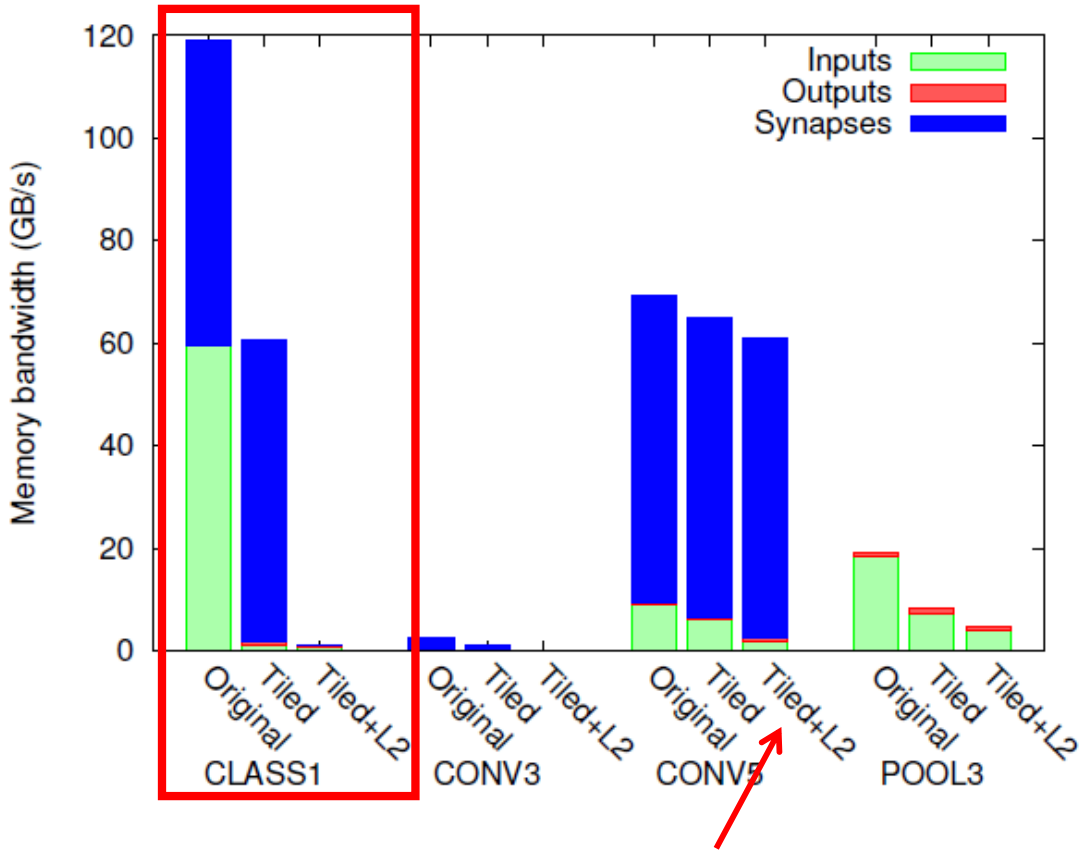

```
for (int nnn = 0; nnn ¡ Nn; nnn += Tnn) { // tiling for output neurons;
  for (int iii = 0; iii ¡ Ni; iii += Tii) { // tiling for input neurons;
    for (int nn = nnn; nn ¡ nnn + Tnn; nn += Tn) {
      for (int n = nn; n ¡ nn + Tn; n++)
        sum[n] = 0;
      for (int ii = iii; ii ¡ iii + Tii; ii += Ti)
        // — Original code —
        for (int n = nn; n < nn + Tn; n++)
          for (int i = ii; i < ii + Ti; i++)
            sum[n] += synapse[n][i] * neuron[i];
      for (int n = nn; n < nn + Tn; n++)
        neuron[n] = sigmoid(sum[n]);
} } }
```

MV mul for [TnxTi] x [Tix1]

Input reuse

Classifier Layer

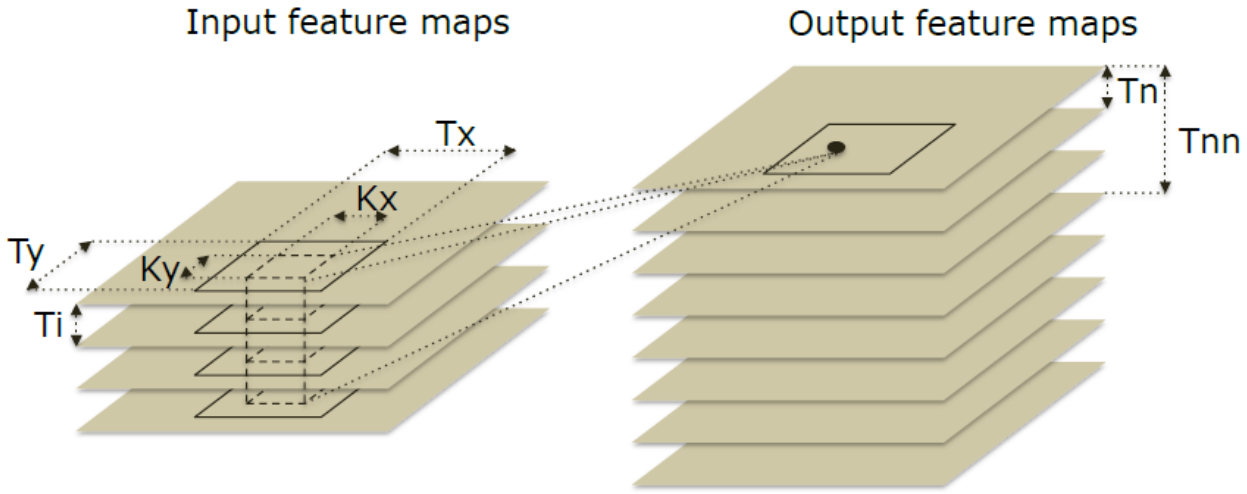
- Input/Output neurons
  - Can fit L1 thanks to tiling
- Synapses (filter weights)
  - No reuse within layer
  - Reused across network invocations (for each new input data)
  - Large L2 cache can store all network Synapses



~100M synapses cannot fit L2

Convolutional Layer

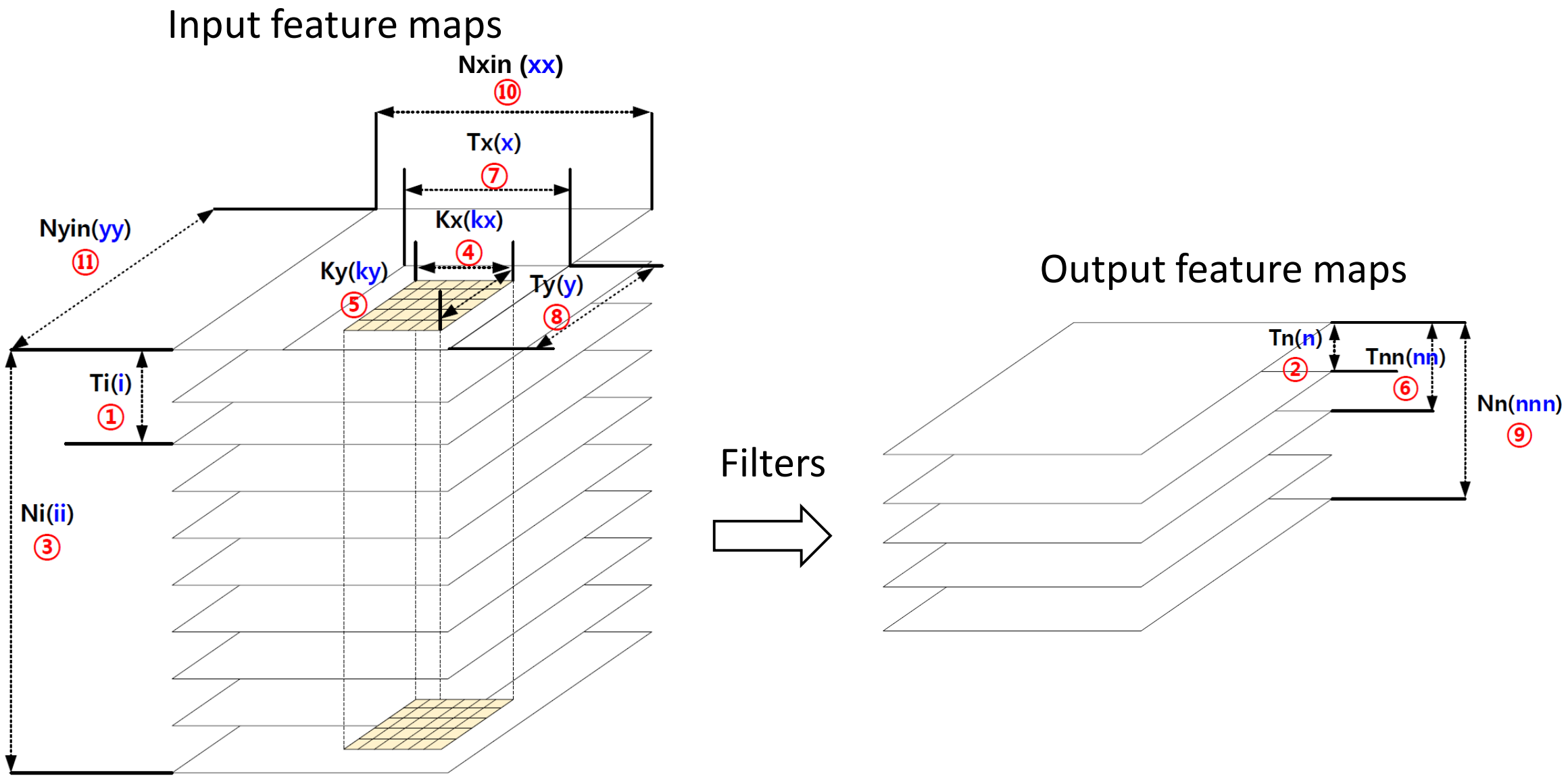
Tile size: Tx * Ty * Ni




```
for (int yy = 0; yy ¡ Nyin; yy += Ty)  {
  for (int xx = 0; xx ¡ Nxin; xx += Tx)  {
    for (int nnn = 0; nnn ¡ Nn; nnn += Tnn)  {
      // — Original code — (excluding nn, ii loops)
      int yout = 0;
      for (int y = yy; y < yy + Ty; y += sy) {  // tiling for y;
        int xout = 0;
        for (int x = xx; x < xx + Tx; x += sx) {  // tiling for x;
          for (int nn = nnn; nn < nnn + Tnn; nn += Tn) {
            for (int n = nn; n < nn + Tn; n++)
              sum[n] = 0;
            // sliding window;
            for (int ky = 0; ky < Ky; ky++)
              for (int kx = 0; kx < Kx; kx++)
                for (int ii = 0; ii < Ni; ii += Ti)
                  for (int n = nn; n < nn + Tn; n++)
                    for (int i = ii; i < ii + Ti; i++)
                      // version with shared kernels
                      sum[n] += synapse[ky][kx][n][i]
                                * neuron[ky + y][kx + x][i];
                      // version with private kernels
                      sum[n] += synapse[yout][xout][ky][kx][n][i]}
                                * neuron[ky + y][kx + x][i];
            for (int n = nn; n < nn + Tn; n++)
              neuron[yout][xout][n] = non_linear_transform(sum[n]);
          } xout++; } yout++;
} } } }
```

Sliding Kx * Ky * Ti

Convolutional Layer



Convolutional Layer

- Reuse opportunities for input and output neurons
  - Sliding window used to scan input layer

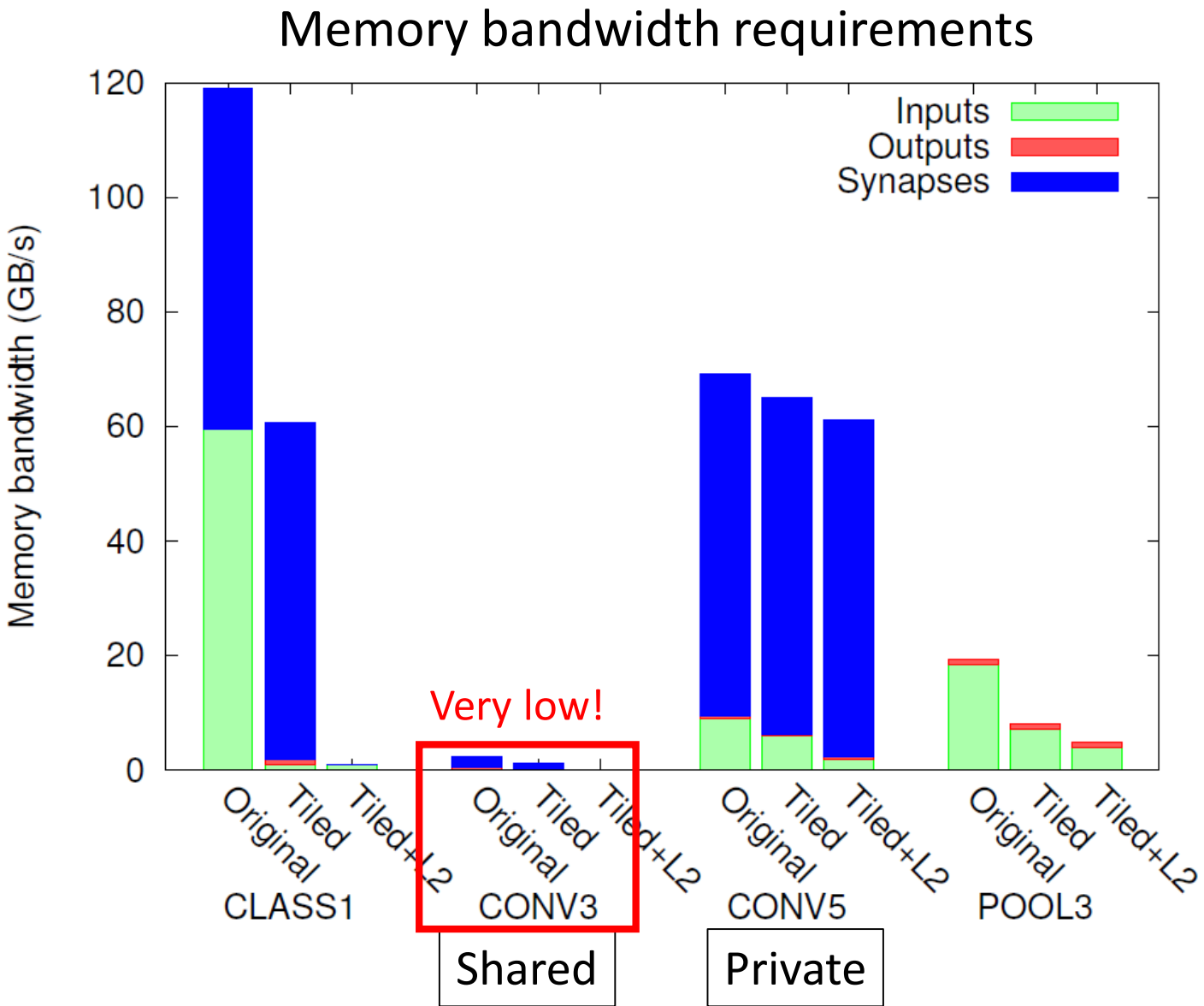
    $\frac{K_x \times K_y}{s_x \times s_y}$ reuses at most (s_x, s_y are stride size)
  - Reuse input neurons across output feature maps

    N_n(=# of output feature maps) reuse

    Tiling is not needed as one kernel $K_x \times K_y \times N_i$ fits in L1 cache (N_i: ~ a few hundreds)

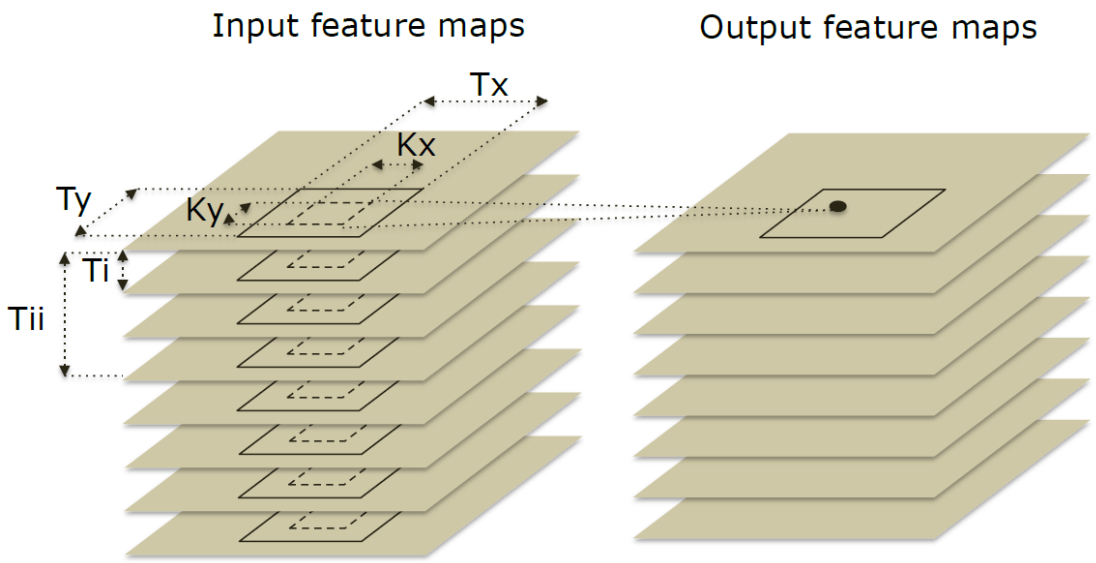
    If not, apply tiling again
- Synapses
  - Shared kernels are reused across all output feature map locations → total bandwidth requirement is very low
  - If the total shared kernels capacity $K_x \times K_y \times N_i \times N_o$ exceeds L1 cache (not likely), we can tile output feature maps by Tnn → $K_x \times K_y \times N_i \times T_{nn}$

Convolutional Layer



Pooling Layer

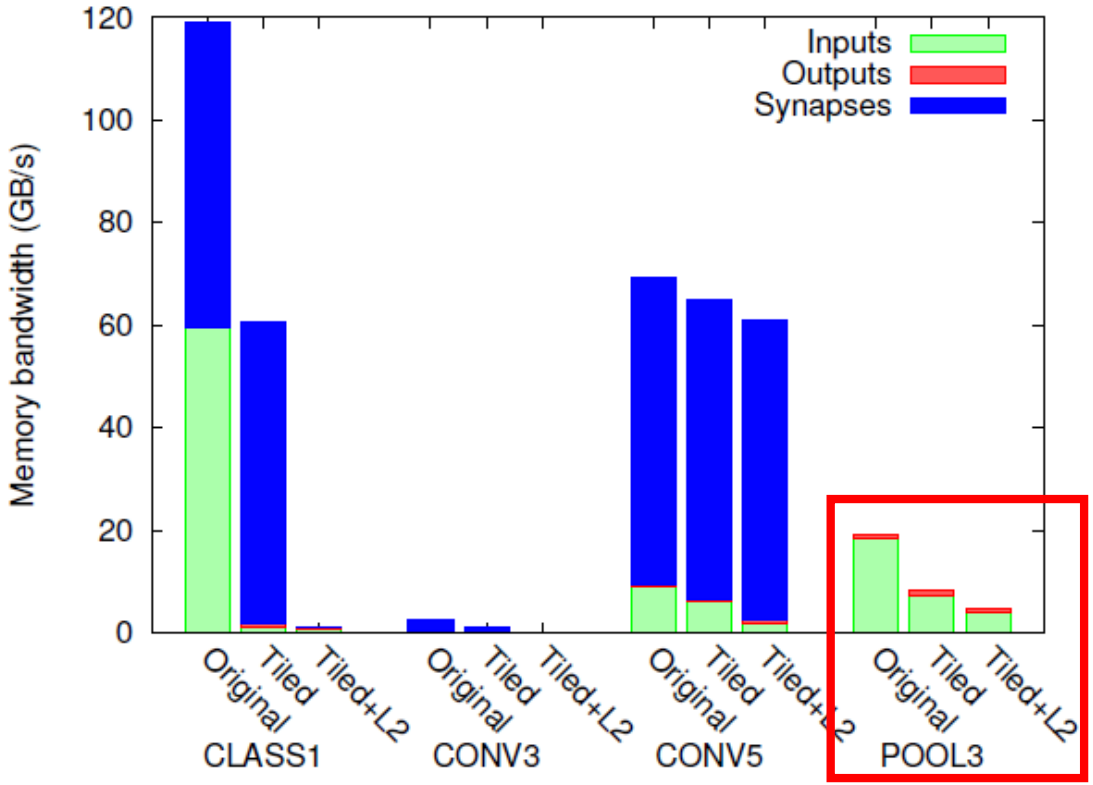
Tile size: Tx * Ty * Ni




```
for (int yy = 0; yy ¡ Nyin; yy += Ty)  {
  for (int xx = 0; xx ¡ Nxin; xx += Tx)  {
    for (int iii = 0; iii ¡ Ni; iii += Tii)
      // — Original code — (excluding ii loop)
      int yout = 0;
      for (int y = yy; y < yy + Ty; y += sy) {
        int xout = 0;
        for (int x = xx; x < xx + Tx; x += sx) {
          for (int ii = iii; ii < iii + Tii; ii += Ti)
            for (int i = ii; i < ii + Ti; i++)
              value[i] = 0;
            for (int ky = 0; ky < Ky; ky++)
              for (int kx = 0; kx < Kx; kx++)
                for (int i = ii; i < ii + Ti; i++)
                  // version with average pooling;
                  value[i] += neuron[ky + y][kx + x][i];
                  // version with max pooling;
                  value[i] = max(value[i], neuron[ky + y][kx + x][i]);
        } } } }
        // for average pooling;
        neuron[xout][yout][i] = value[i] / (Kx * Ky);
        xout++; } yout++;
} } }
```

Pooling Layer

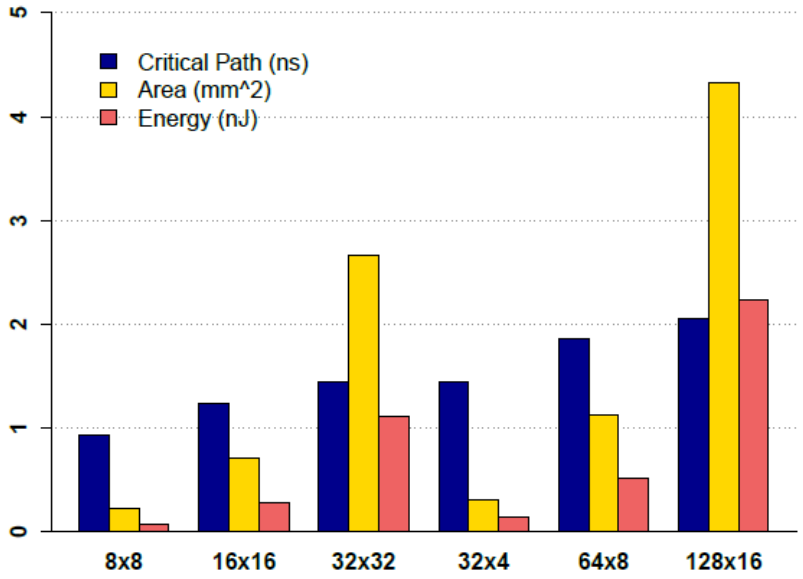
- Characteristics
  - No kernel weights
  - Number of input and output feature maps are the same
- An output feature map element is determined only by $K_x \times K_y$ input feature map
- Reuse only comes from the sliding window



Tiling effect is not so large

Scaling Up Model Size

- Naive hardware implementation
  - Neurons: logic circuit
  - Synapses: Latches or RAM
- Area, energy, delay grow quadratically with the number of neurons
  - Time-sharing of physical neurons + use of on-chip RAM to store synapses and intermediate neurons values
  - However, large models cannot fit in on-chip → interplay between computation and memory hierarchy becomes the key

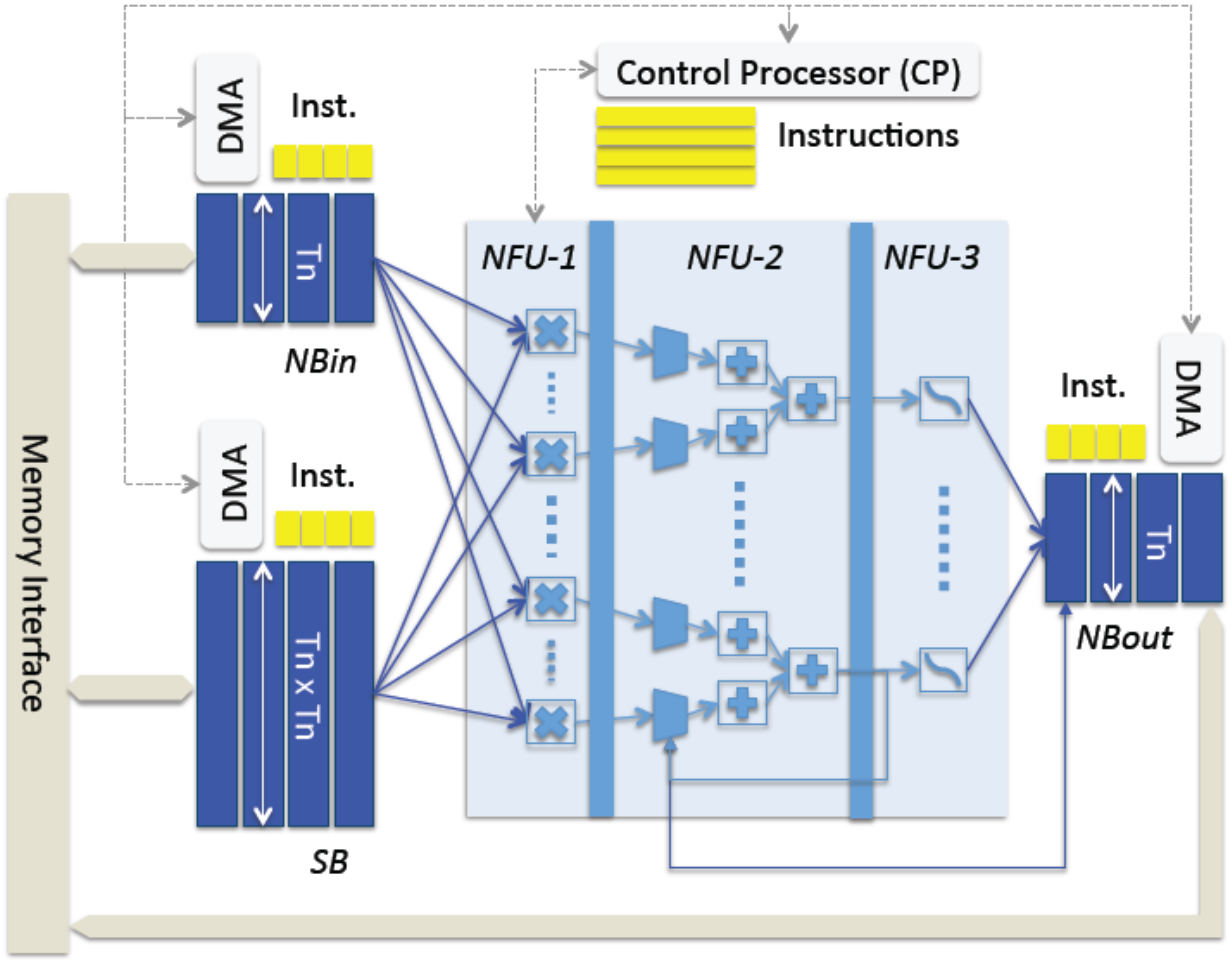


Accelerator for Large Neural Networks

- Main Components
  - Neural Functional Unit (NFU)
  - Input buffer for input neurons (NBin)
  - Output buffer for output neurons (NBout)
  - Synapse buffer (SB)
  - Control processor (CP)

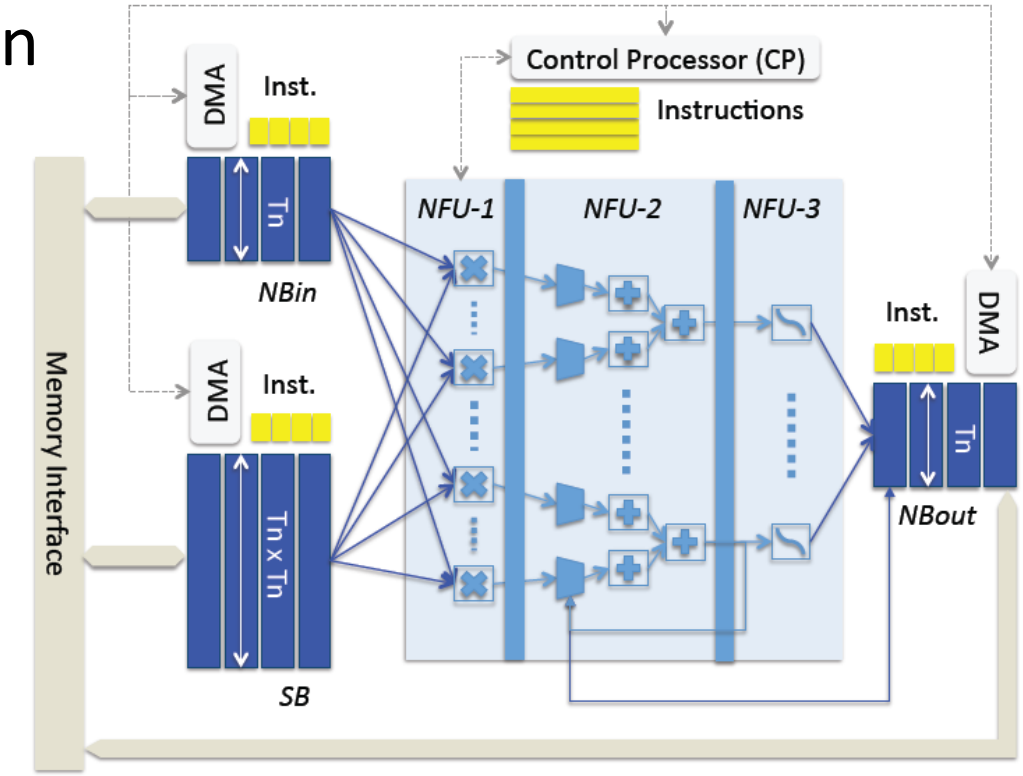
Neural Functional Unit

- Design principle: decomposition of a layer into computational blocks of Ti x Tn
  - T_i inputs neurons
  - T_n outputs neurons
  - i & n loops for both classifier and convolutional layer
  - i loops for pooling layers



Neural Functional Unit

- 3 stage implementation



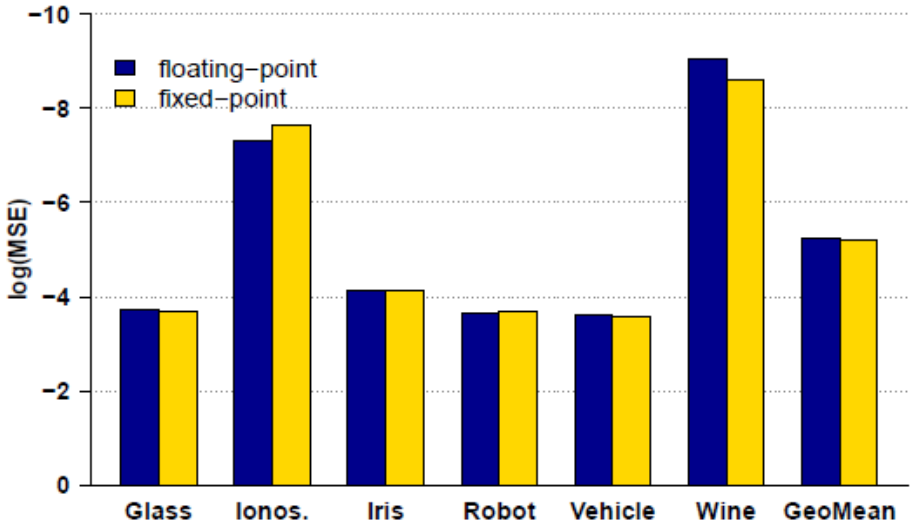
	NFU-1 (Multiplication)	NFU-2 (Addition/Max)	NFU-3 (Activation)
Classifier	O	O	O (Sigmoid)
Convolution	O	O	O (ReLu)
Pooling	X	O	X

Neural Functional Unit

- 16-bit fixed-point arithmetic operators
  - Smaller area, lower power consumption

Type	Area (μm^2)	Power (μW)
16-bit truncated fixed-point multiplier	1309.32	576.90
32-bit floating-point multiplier	7997.76	4229.60

- 32-bit floating-point is not necessary for inference, no impact on accuracy



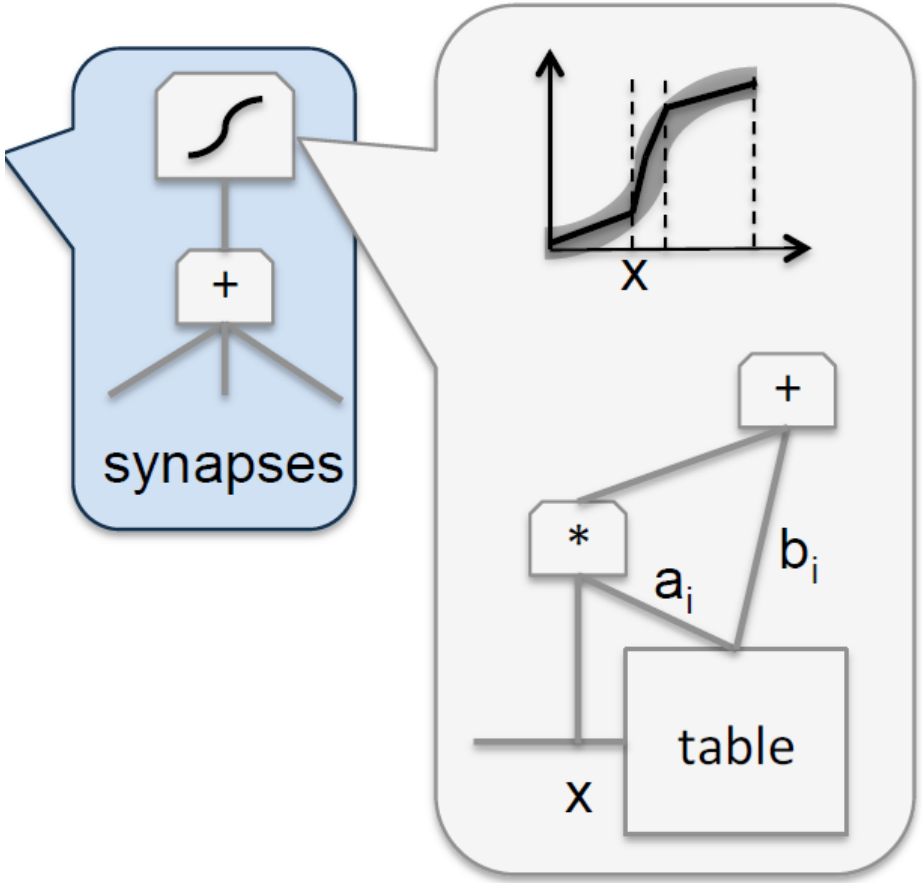
UCI data set

Type	Error Rate
32-bit floating-point	0.0311
16-bit fixed-point	0.0337

MNIST

Neural Functional Unit

- Activation function



- Piecewise linear interpolation
- Negligible loss of accuracy

  $f(x) = a_i \times x + b_i, x \in [x_i, x_{i+1}]$
- Coefficient a_i, b_i are stored in a small RAM
- Can have other functions (sigmoid, tanh,..) by changing coefficients

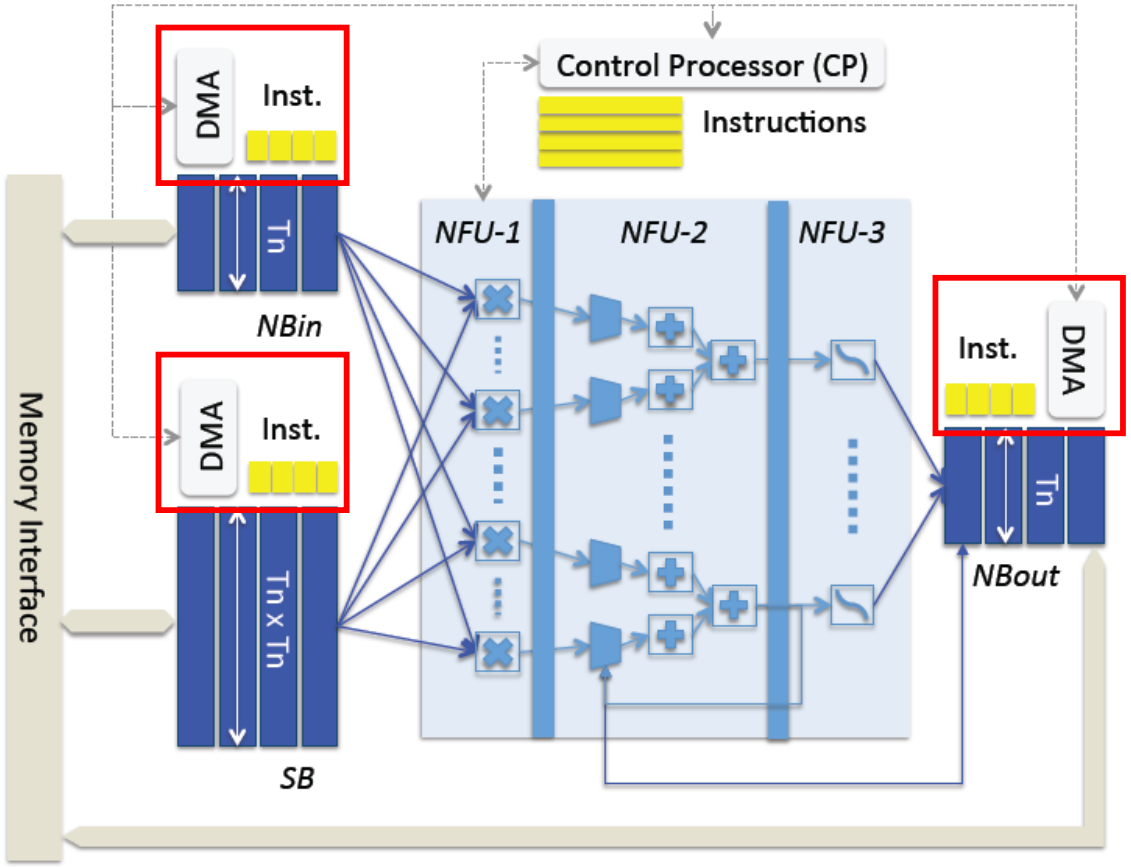
Storage Buffer Structure

- NBin, NBout, SB, NFU Registers
  - Cache is excellent for general-purpose computer
  - Not optimal for accelerator due to cache access overhead and cache conflicts
- Use Scratchpad (local SRAM) instead
  - Efficient storage
  - Easy exploitation of locality with focusing on a few algorithms
- Split buffers
  - Tailor each buffer's data width to appropriate one
  - Avoid conflicts based on known locality behaviors

	Width
NBin	$T_i (or T_n) \times 2\ byte$
SB	$T_i (or T_n) \times T_n \times 2\ byte$
NBout	$T_n \times 2\ byte$

Exploiting Locality of Inputs and Synapses

- DMA for each buffer
  - Two load DMAs, one store DMA
  - DMA instructions from control processor decouples memory transfer and NFU computations
  - Preload next data if possible to mitigate long access time



Exploiting Locality of Inputs and Synapses

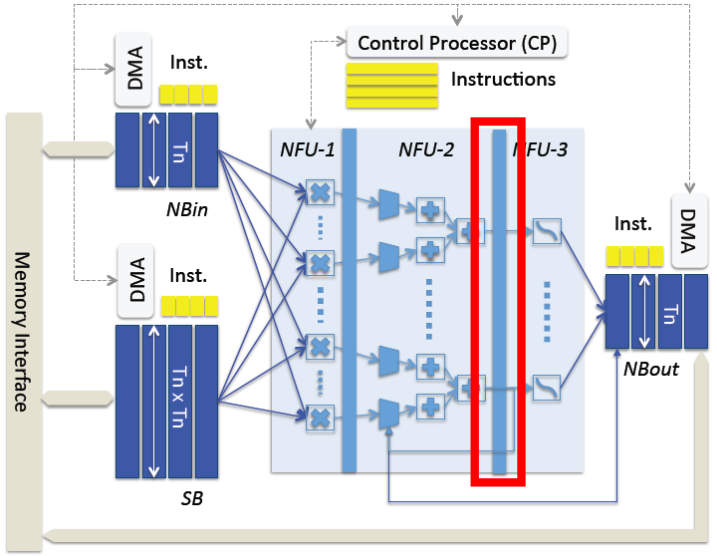
- Perform local transpose in NBin for pooling layer
  - Convolution layer iterates depth-wise first
  - In pooling, k_x, k_y should be inner most index
  - Transpose them by loading along i and store in k_x, k_y

Load order	NBin order
ky=0, kx=0, i=0	ky=0, kx=0, i=0
ky=0, kx=0, i=1	ky=1, kx=0, i=0
ky=0, kx=0, i=2	ky=0, kx=0, i=1
ky=0, kx=0, i=3	ky=1, kx=0, i=1
ky=1, kx=0, i=0	ky=0, kx=0, i=2
ky=1, kx=0, i=1	ky=1, kx=0, i=2
ky=1, kx=0, i=2	ky=0, kx=0, i=3
...	...

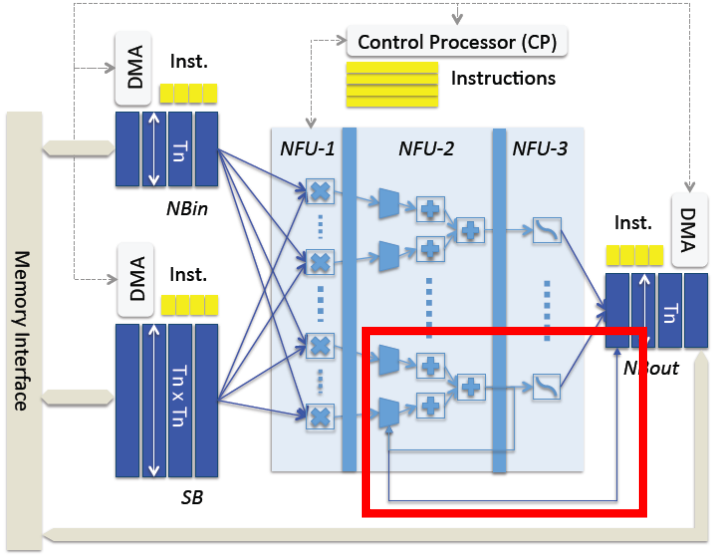
Local transpose (Ky = 2;Kx = 1;Ni = 4).

Exploiting Locality of Outputs

- Dedicated registers
  - Stores partial sums in the pipeline registers
  - Remove unnecessary data transfer between NFU and buffers



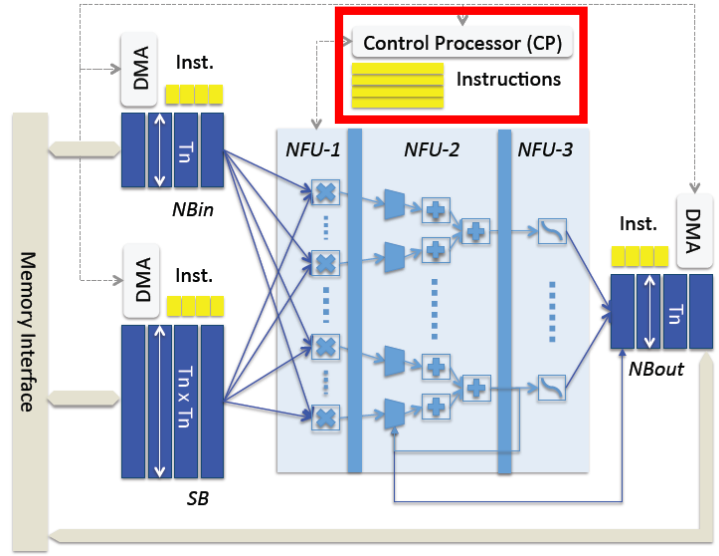
- Temporal use of NBout as circular buffer
  - Where to send T_n partial sums when the input neurons in NBin are used for a new set of T_n output neurons?
  - Instead sending them back to memory, leverage NBout



Control Processor

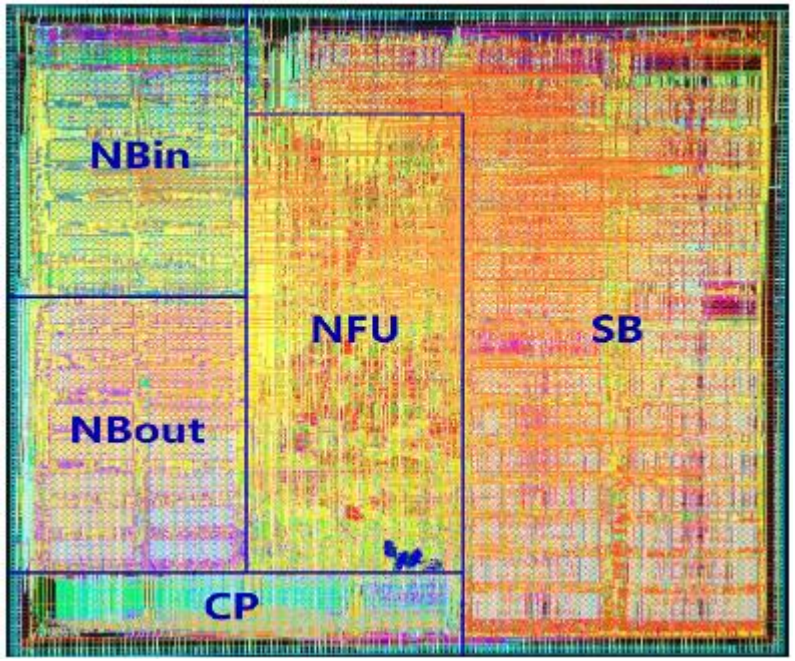
- Control instructions for NFU/SB/NBin/NBout for more flexibility
  - Drives the executions of 3 DMAs
  - Generate control signals for NFU pipelines
  - Not like a traditional processor that has full programmability

CP	SB				NBin							NBout				NFU						
END	READ OP	REUSE	ADDRESS	SIZE	READ OP	REUSE	STRIDE	STRIDE BEGIN	STRIDE END	ADDRESS	SIZE	READ OP	WRITE OP	ADDRESS	SIZE	NFU-1 OP	NFU-2 OP	NFU-2 IN	NFU-2 OUT	NFU-3 OP	OUTPUT BEGIN	OUTPUT END



Implementation Result

- 65nm process, only layout



Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,023,077		485		1.02
Combinational	608,842	(20.14%)	89	(18.41%)	
Memory	1,158,000	(38.31%)	177	(36.59%)	
Registers	375,882	(12.43%)	86	(17.84%)	
Clock network	68,721	(2.27%)	132	(27.16%)	
Filler cell	811,632	(26.85%)			
SB	1,153,814	(38.17%)	105	(22.65%)	
NBin	427,992	(14.16%)	91	(19.76%)	
NBout	433,906	(14.35%)	92	(19.97%)	
NFU	846,563	(28.00%)	132	(27.22%)	
CP	141,809	(5.69%)	31	(6.39%)	
AXIMUX	9,767	(0.32%)	8	(2.65%)	
Other	9,226	(0.31%)	26	(5.36%)	

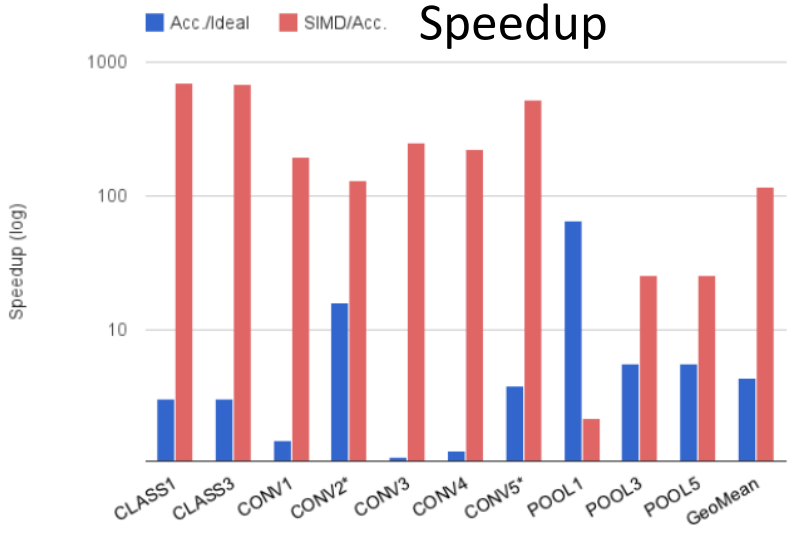
Benchmark Layers

Layer	N_x	N_y	K_x	K_y	N_i	N_o	Description
CONV1	500	375	9	9	32	48	Street scene parsing
POOL1	492	367	2	2	12	-	(CNN) [13], (e.g.,
CLASS1	-	-	-	-	960	20	identifying "building", "vehicle", etc)
CONV2*	200	200	18	18	8	8	Detection of faces in YouTube videos (DNN) [26], largest NN to date (Google)
CONV3	32	32	4	4	108	200	Traffic sign
POOL3	32	32	4	4	100	-	identification for car
CLASS3	-	-	-	-	200	100	navigation (CNN) [36]
CONV4	32	32	7	7	16	512	Google Street View house numbers (CNN) [35]
CONV5*	256	256	11	11	256	384	Multi-Object
POOL5	256	256	2	2	256	-	recognition in natural images (DNN) [16], winner 2012 ImageNet competition

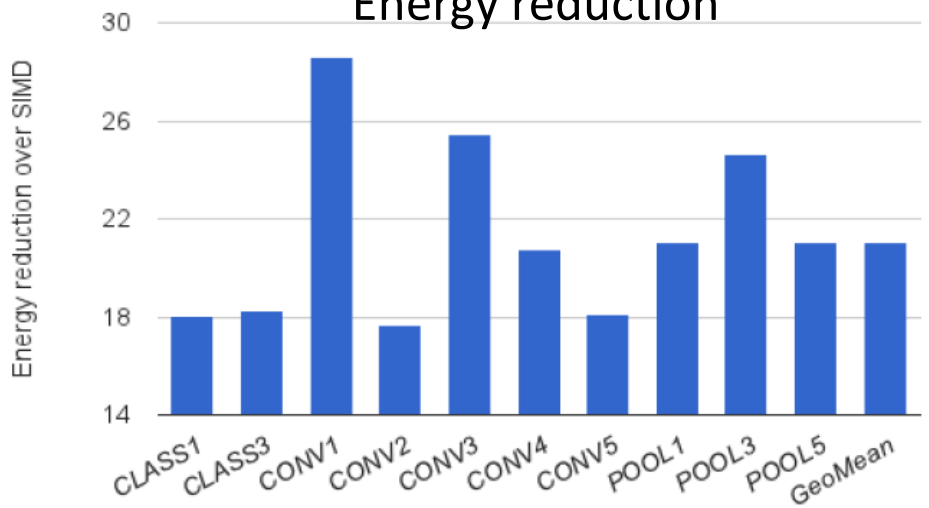
Experimental Results

- Compare against a x86 core (128-bit SIMD with SSE/SSE2 at 2 GHz)

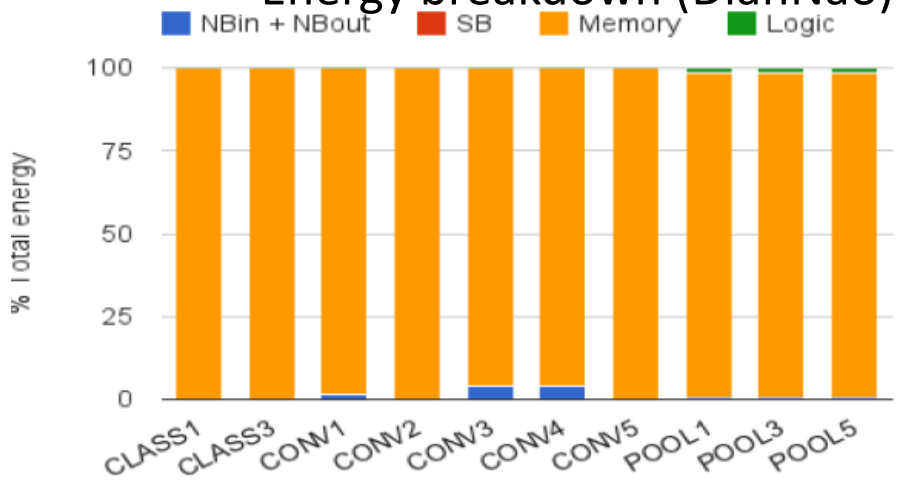
Speedup



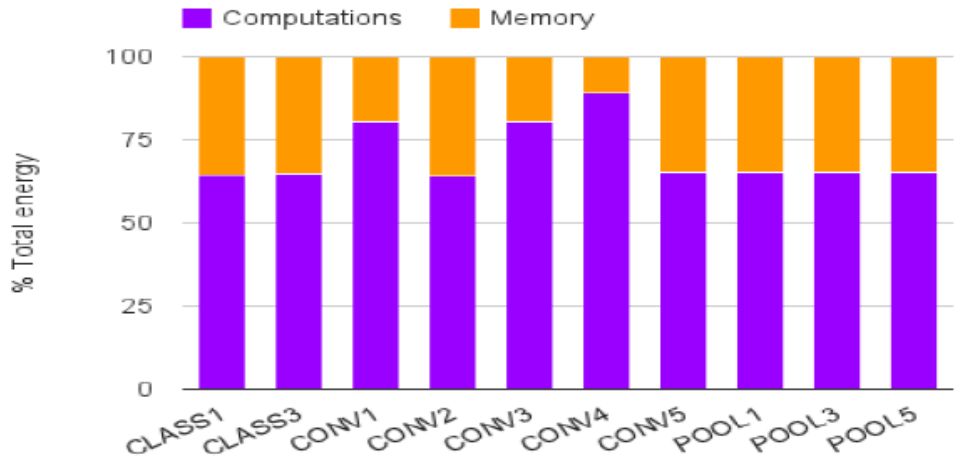
Energy reduction



Energy breakdown (DianNao)

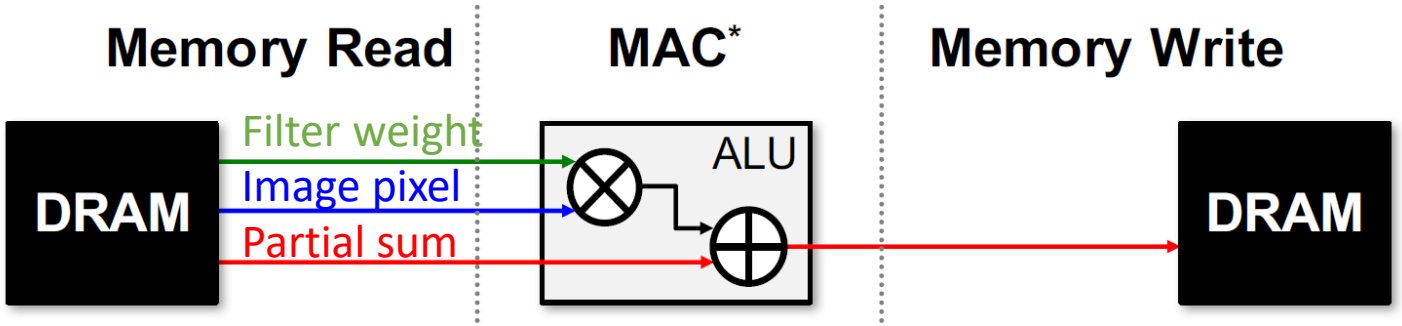


Energy breakdown (SIMD)



Eyeriss

- Accelerator for deep convolutional neural networks (CNN)
  - Large amount of data
  - Significant data movement
  - More details at http://eyeriss.mit.edu



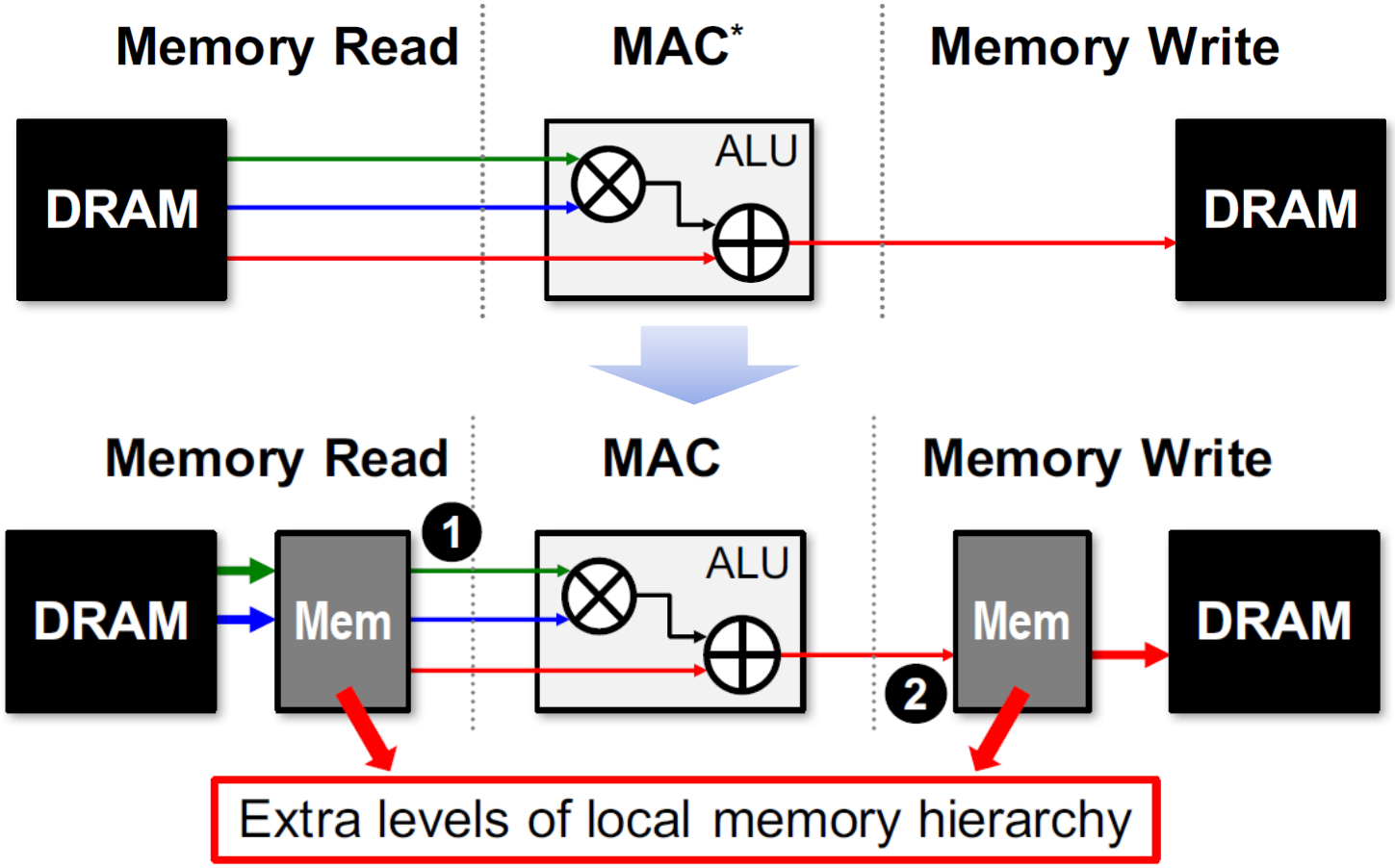
MAC: Multiply-and-Accumulate

Example: AlexNet requires **724M** MACs and **2896M** DRAM accesses (if we assume all memory R/W are DRAM)

Goal: Minimize external (DRAM) data movement!

Memory Hierarchy

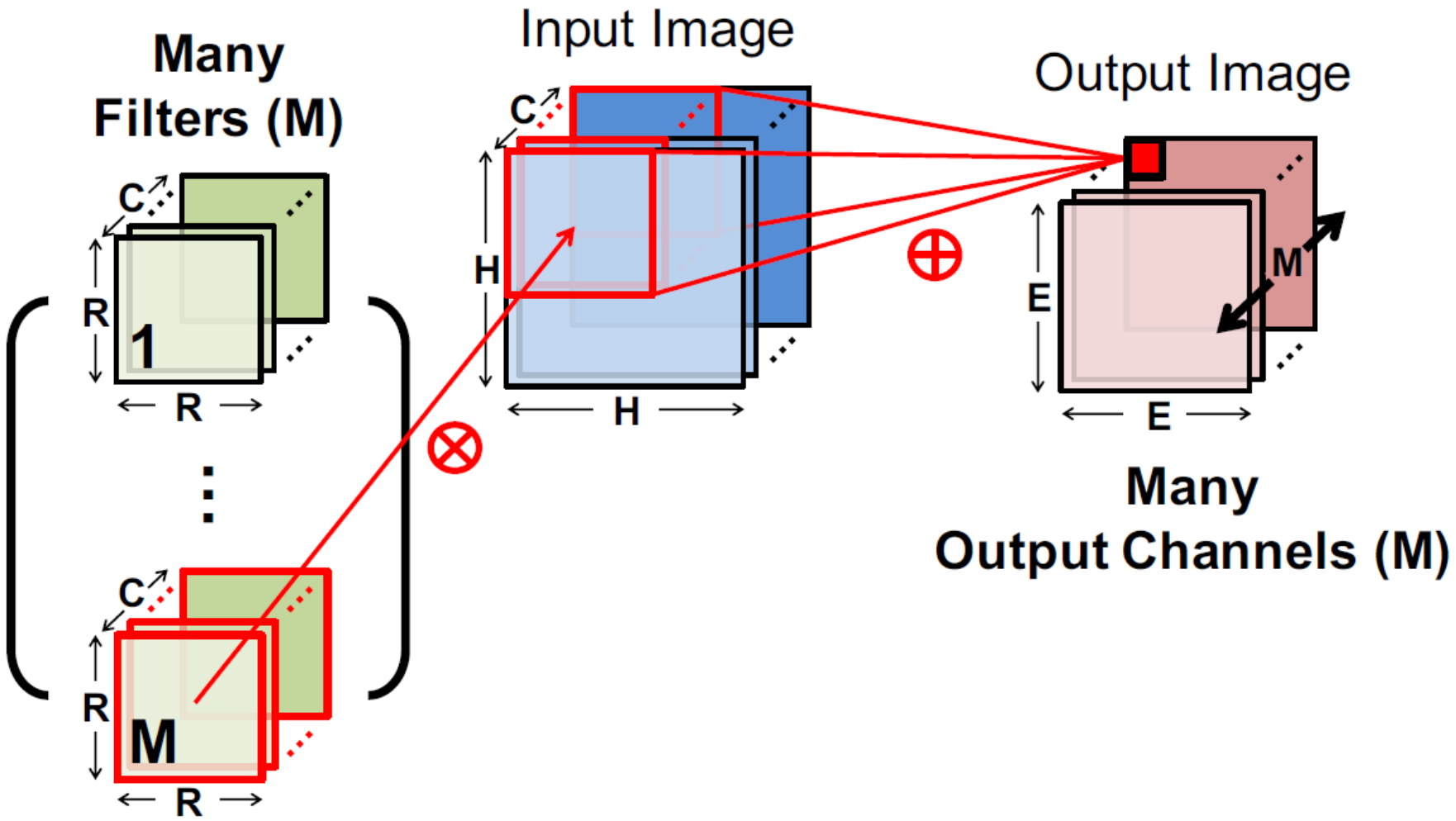
- Additional memories between DRAM and computation unit



Opportunities: 1. Data reuse 2. Local accumulation

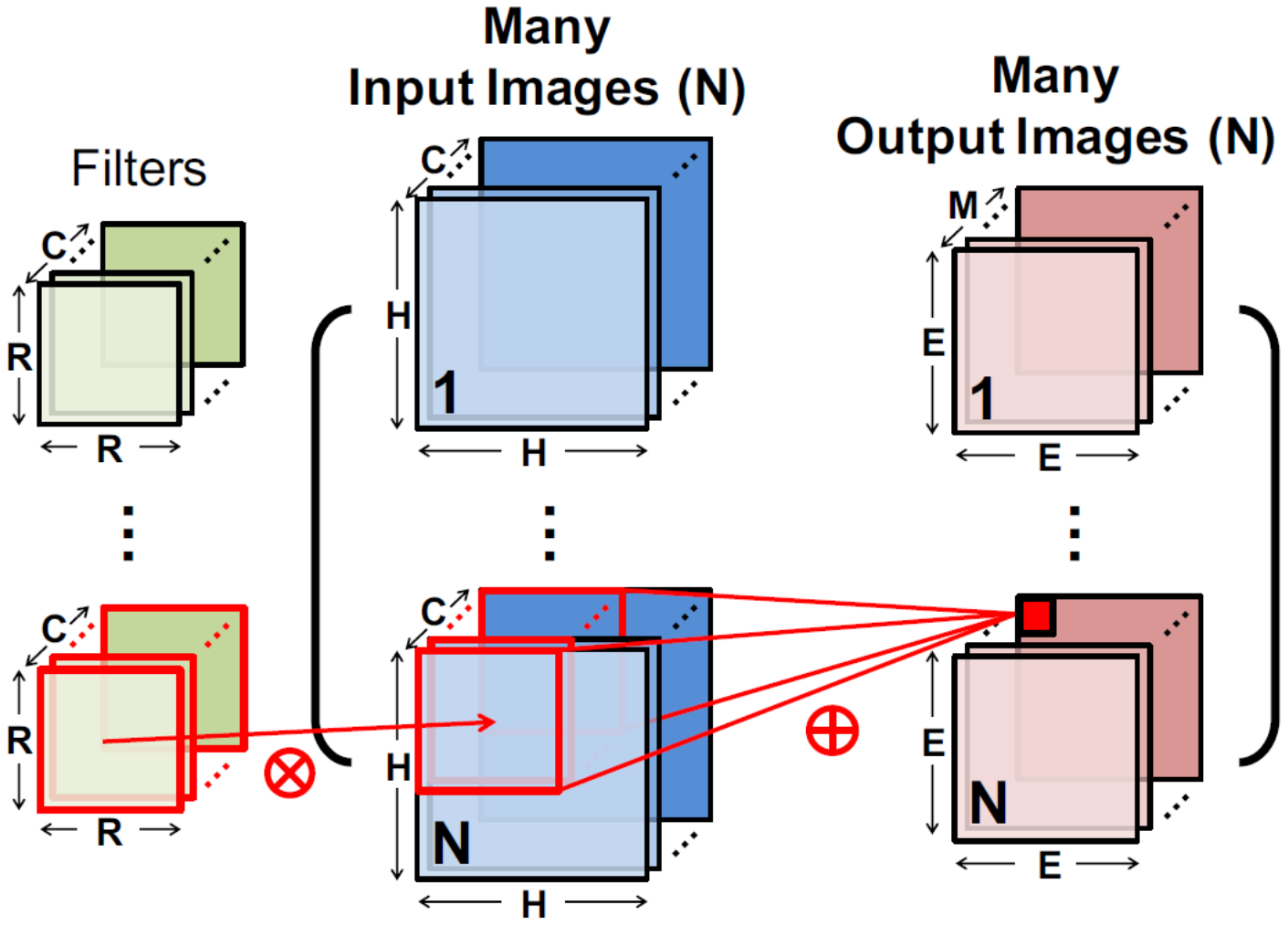
3-D Convolution Review

- 3-D input volume x Multiple 3-D kernel filters = 3-D output volume



3-D Convolution Review

- General case: many filters, many inputs, many outputs



Three Types of Data Reuse

- Data reuse: on-chip data reuse without external memory accesses

Convolutional Reuse

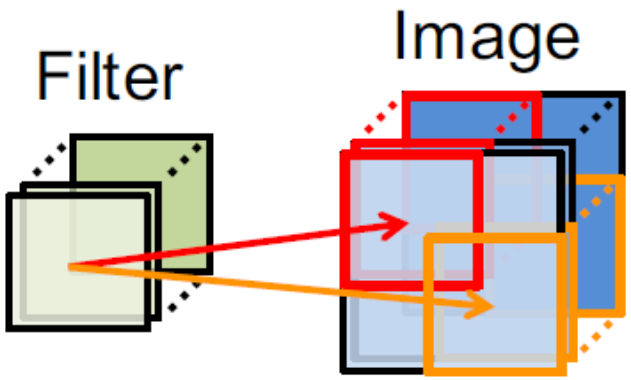
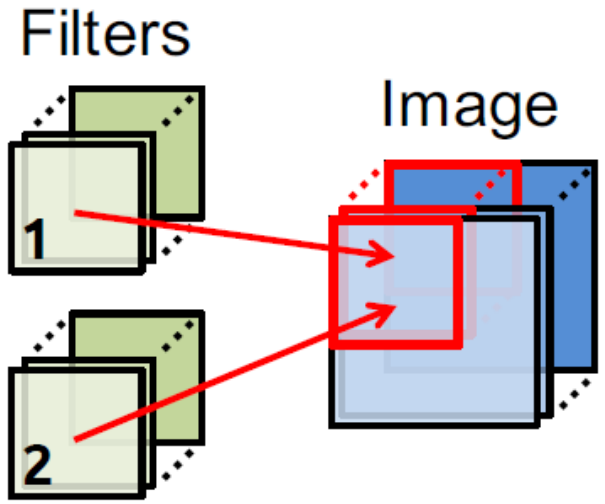
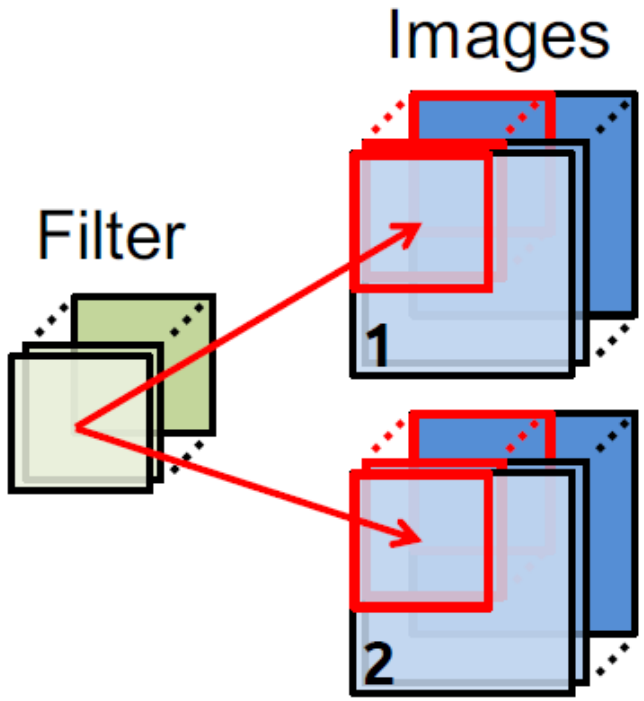


Image Reuse

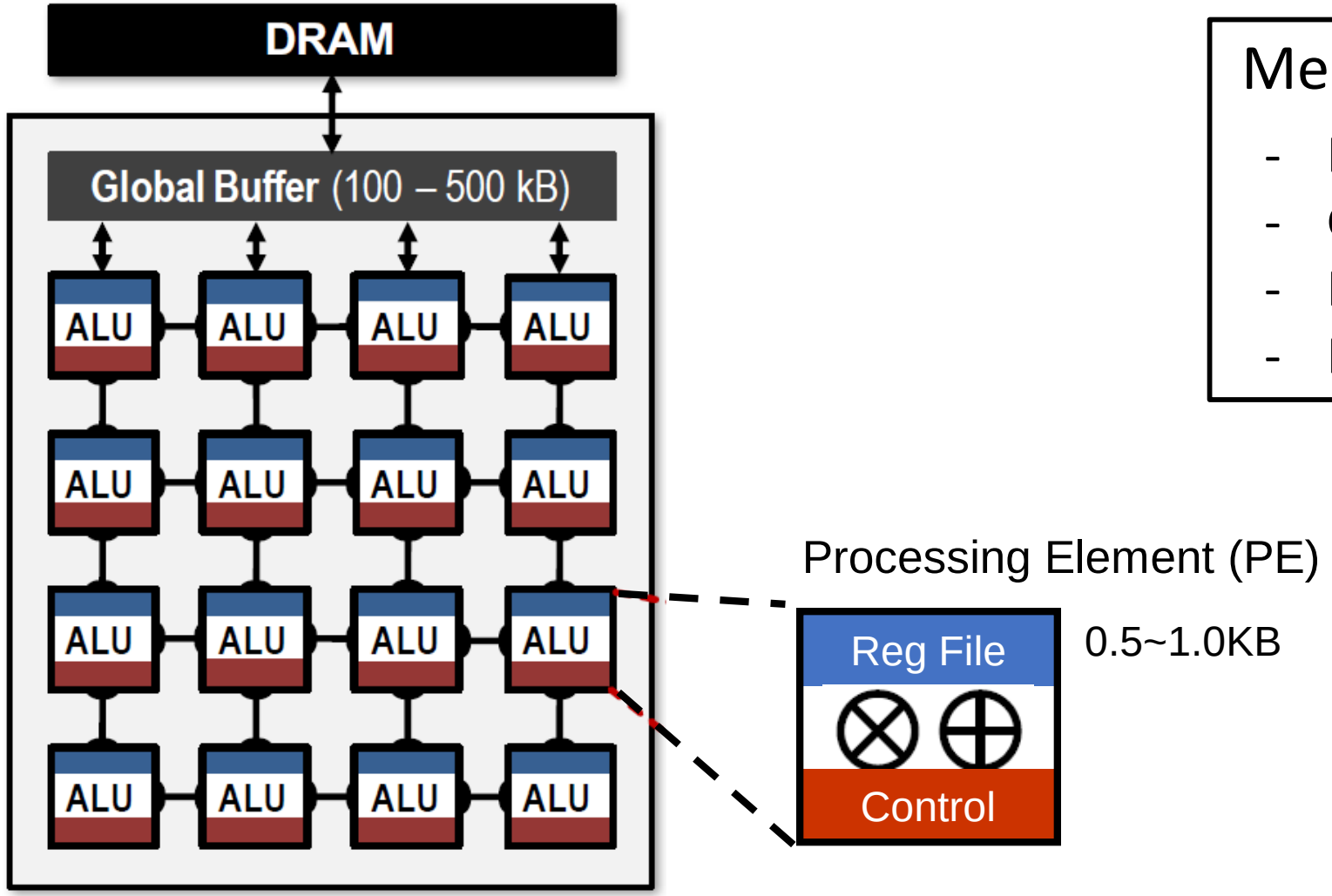


Filter Reuse



Spatial Architecture for CNN

- 2-D array processor

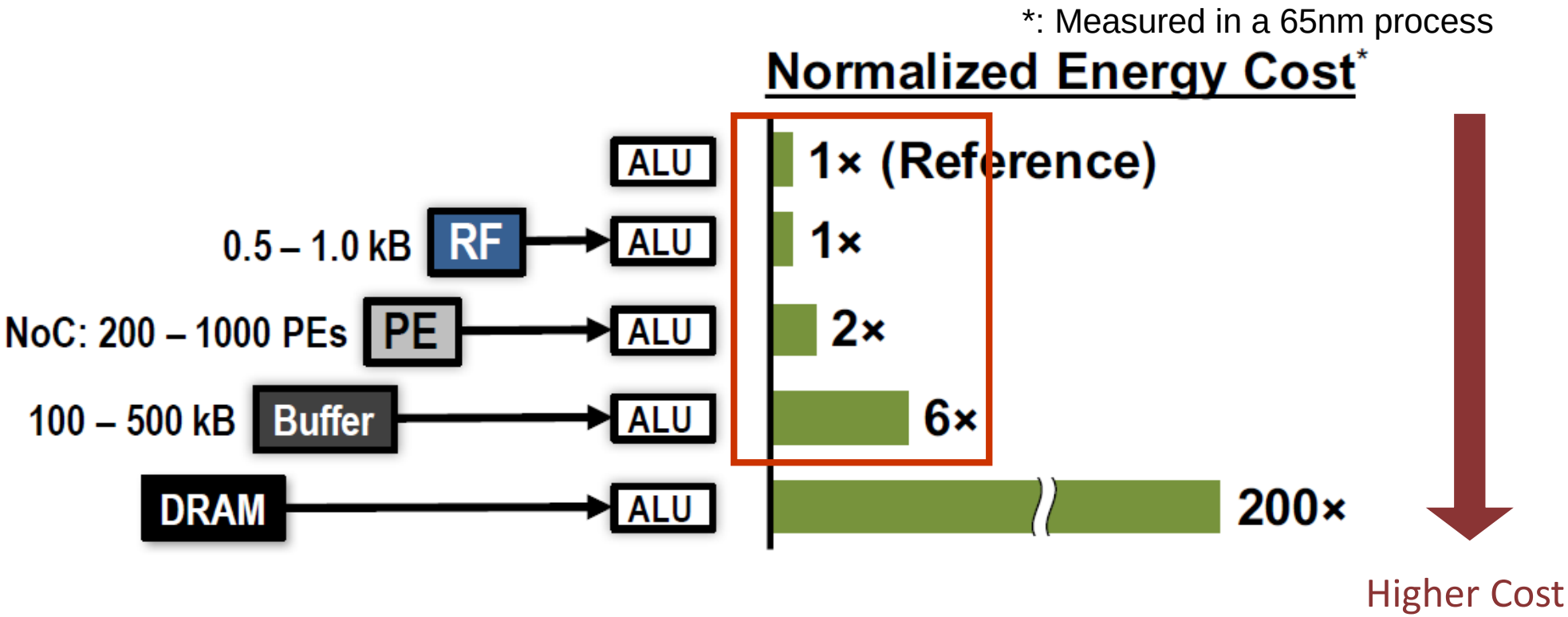


Memory hierarchy

- External DRAM
- Global buffer
- Direct inter-PE network
- PE local register file

Data Access Cost

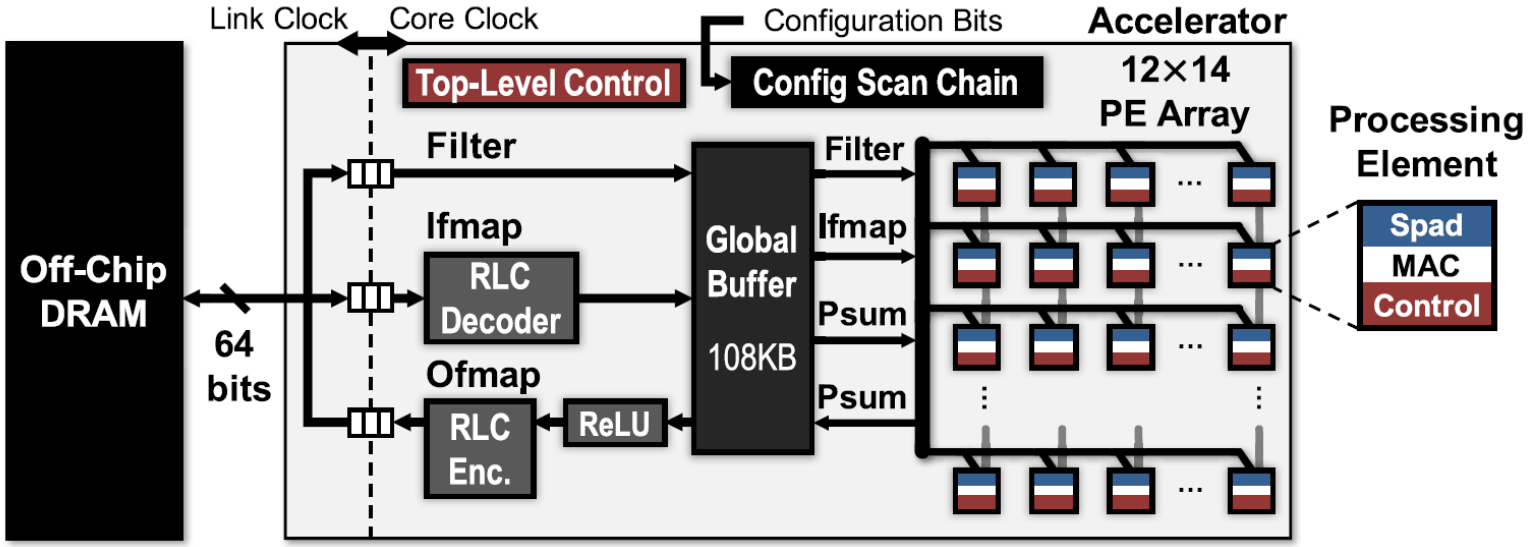
- Energy cost increases exponentially as data travels off-chip memory



Avoid DRAM access & use local data access up to global buffer

How to Leverage Low-Cost Local Data Access

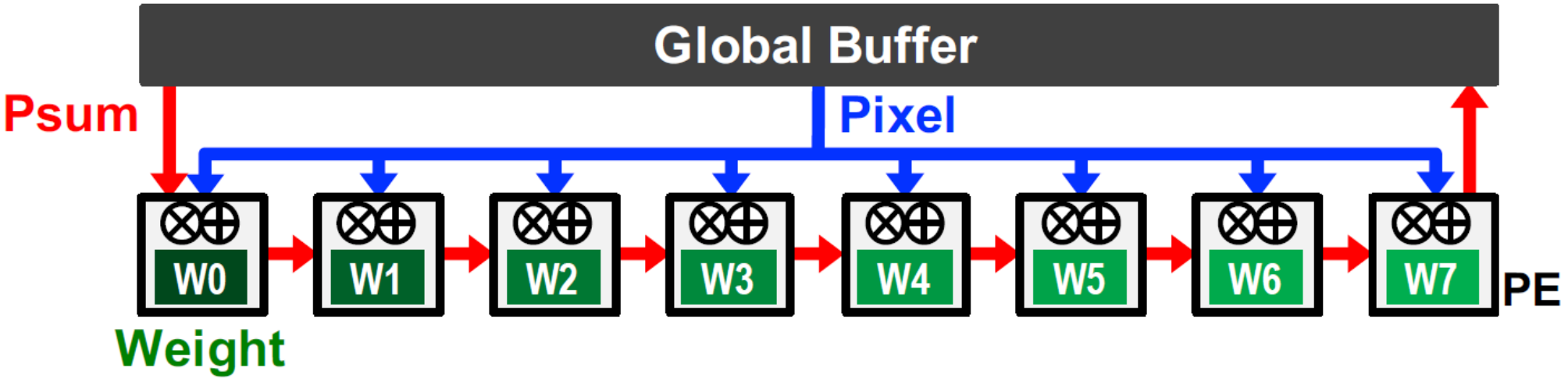
- Data reuse
  - Convolutional reuse / Image reuse / Filter reuse
- Local accumulation
  - Partial sum is accumulated in the scratch pads of PEs and written to Global Buffer
  - No access to DRAM is required until the output feature map is fully calculated



How to map data and process them (dataflow) are important!

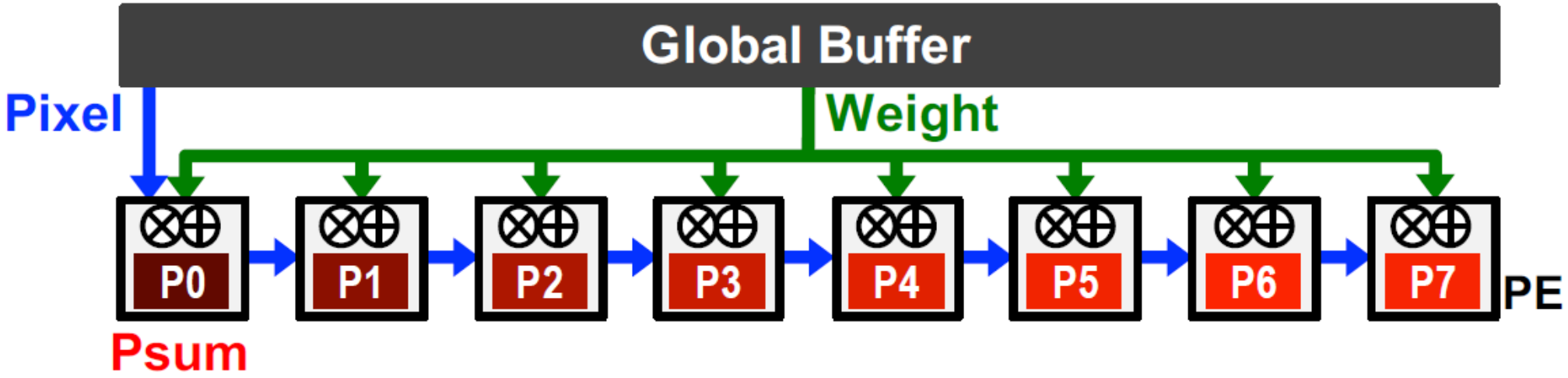
Weight Stationary (WS) Dataflow

- Weight data are pinned in PE local memories while input data and psum are moving from Global Buffer
  - Maximize weight data reuse
  - Minimize weight read energy consumption



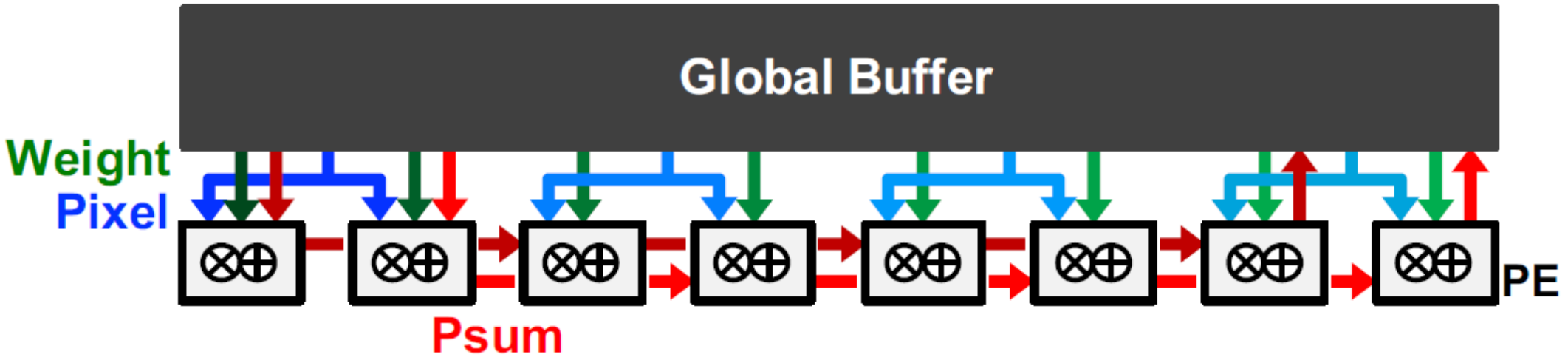
Output Stationary (OS) Dataflow

- Maximize partial sum accumulation
  - Reduce the number of partial sum fetch/store as much as possible
  - Minimize partial sum read/write energy consumption



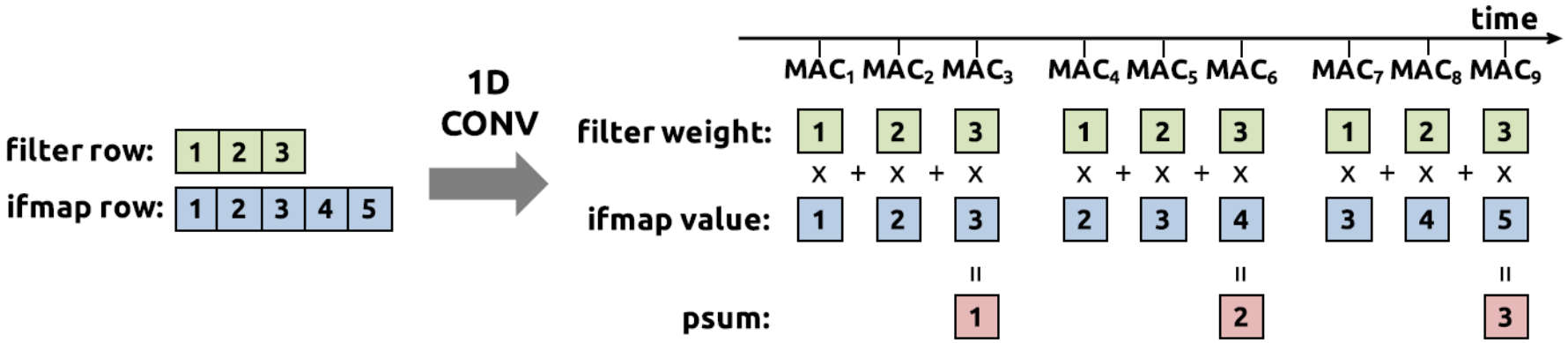
No Local Reuse (NLR)

- Use a large global buffer as shared storage
  - Generic memory model
  - Reduce DRAM access energy consumption



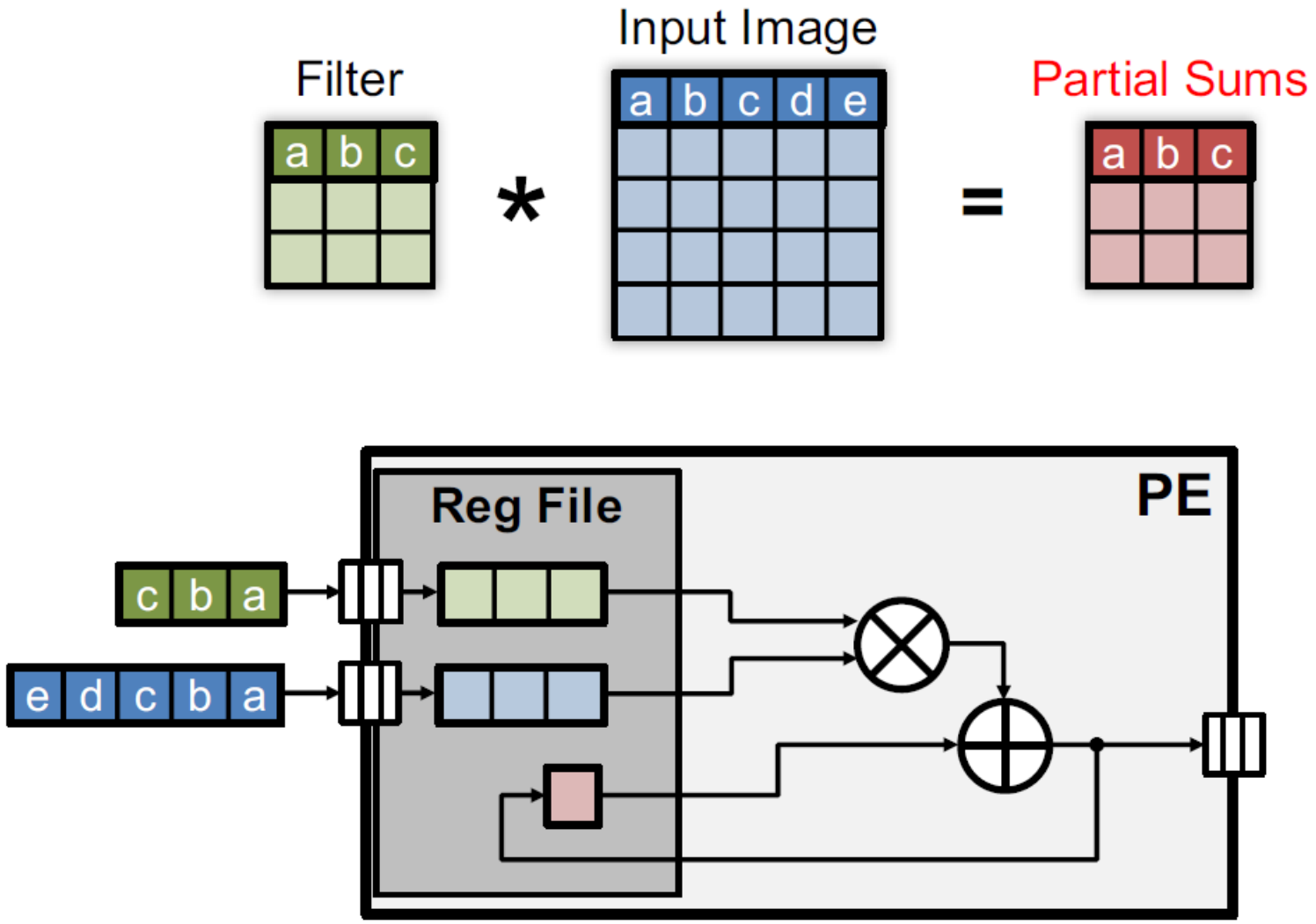
Row Stationary (RS)

- Maximize row convolutional reuse in register files
  - Keep a filter row and image sliding window in register files
- Maximize row psum accumulation in register files

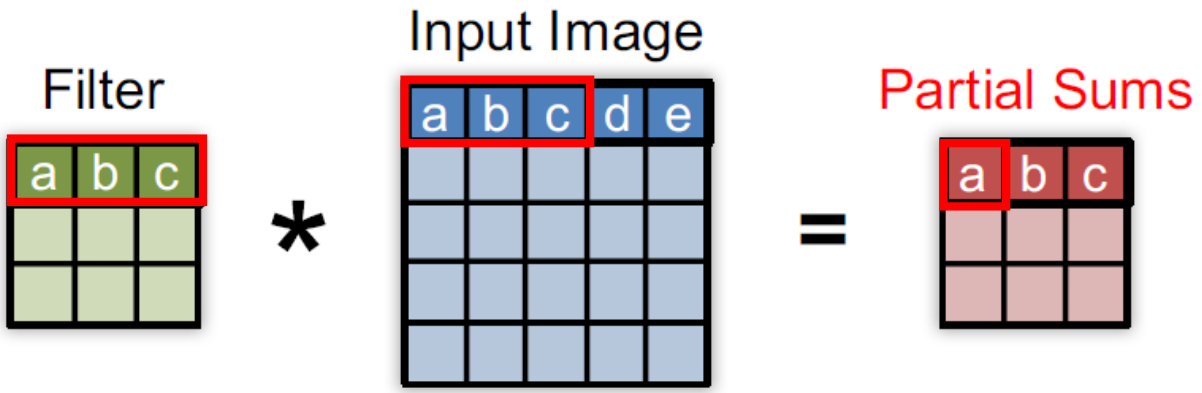


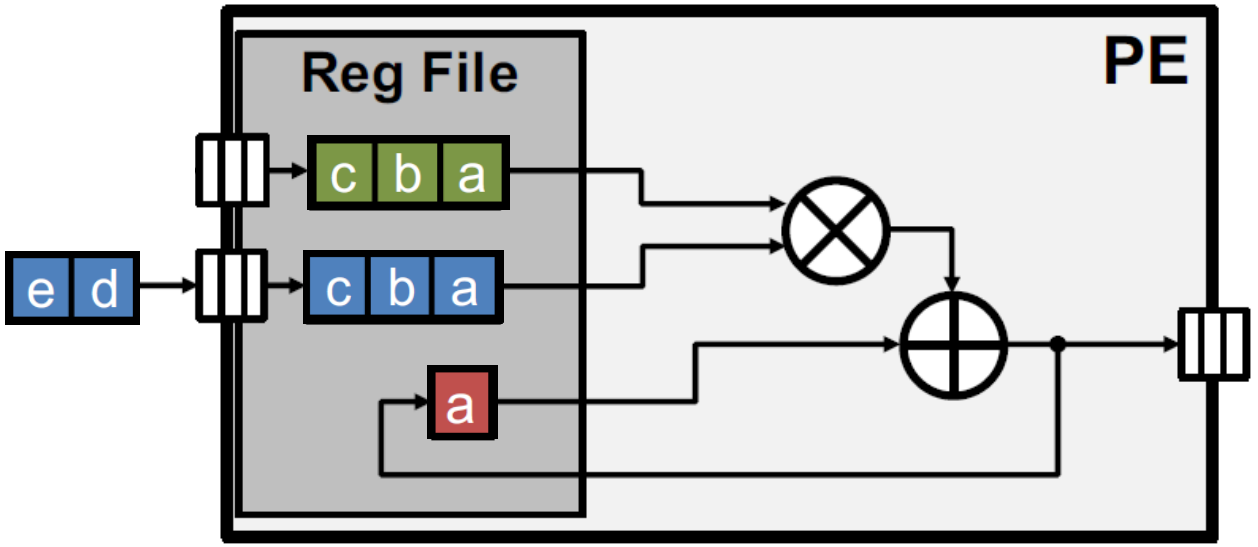
1-D convolution primitive in a PE

1-D Row Convolution in PE

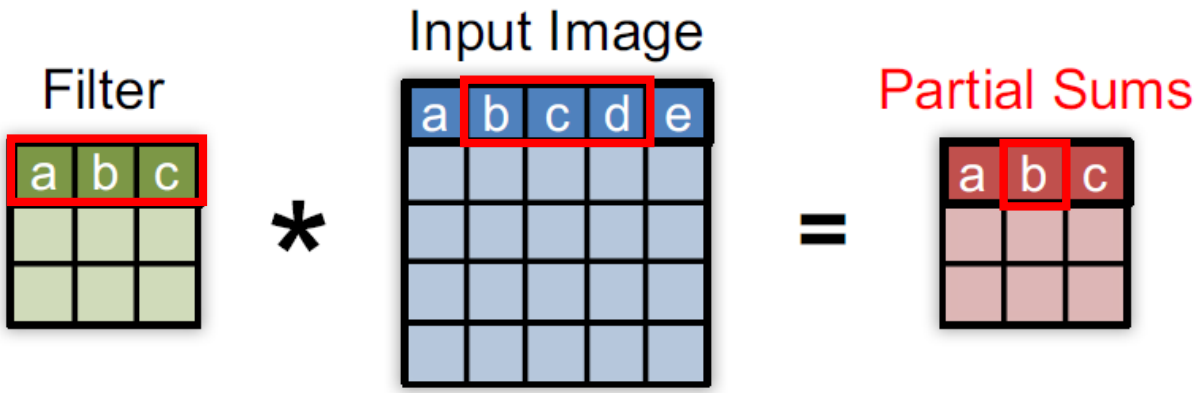


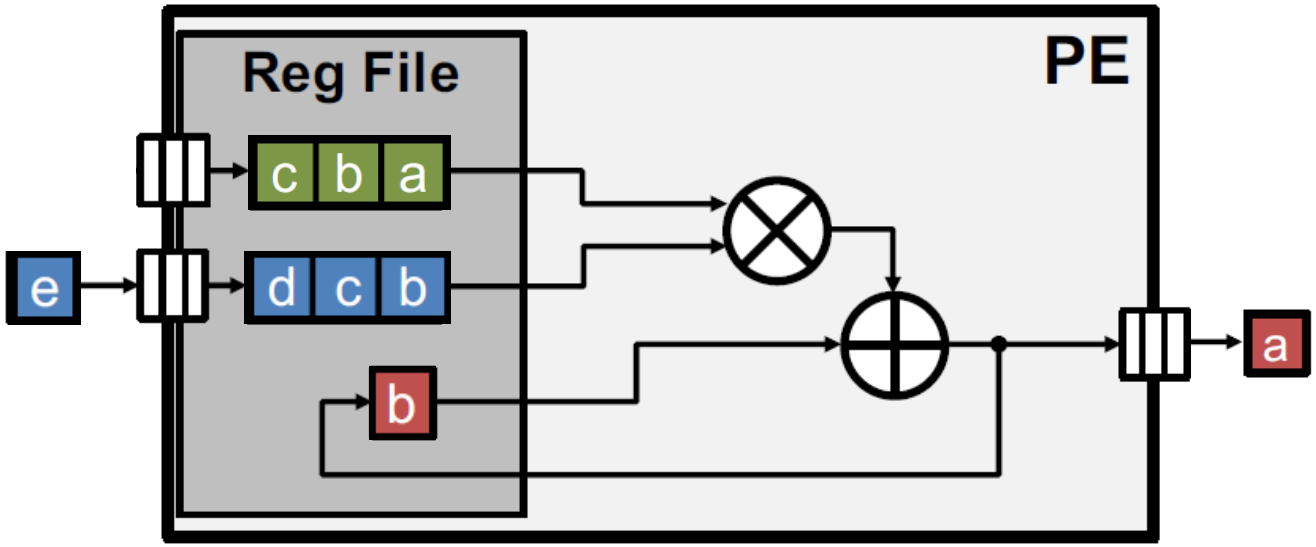
1-D Row Convolution in PE



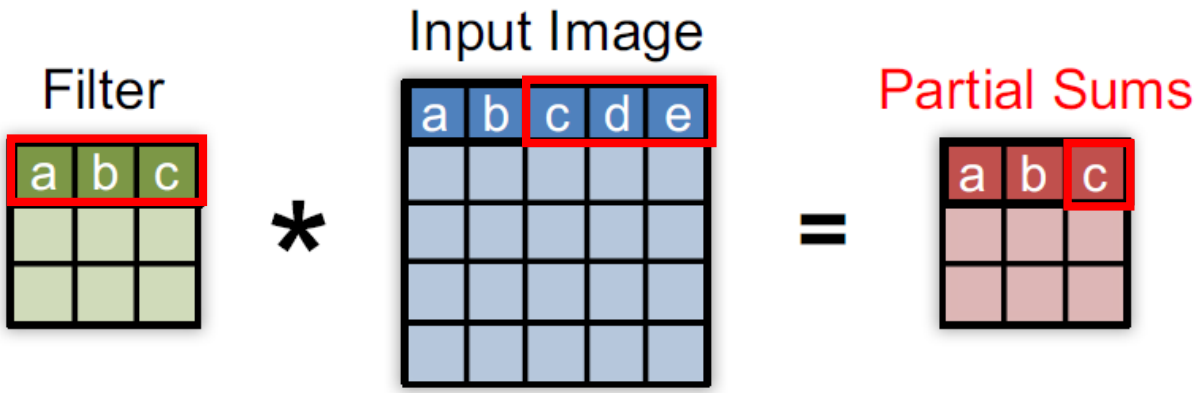


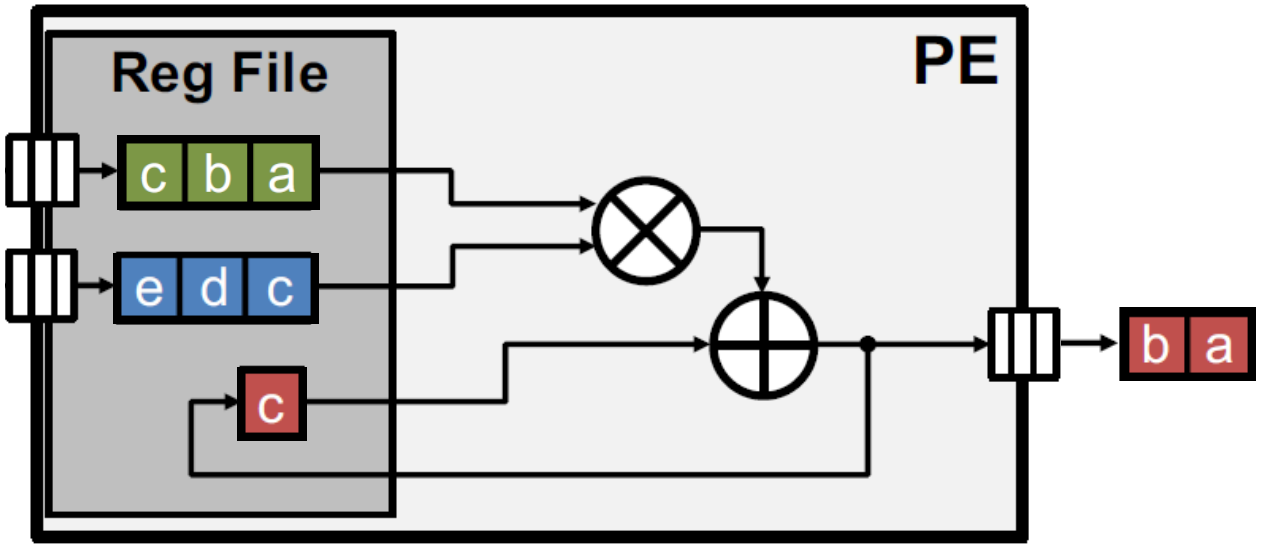
1-D Row Convolution in PE





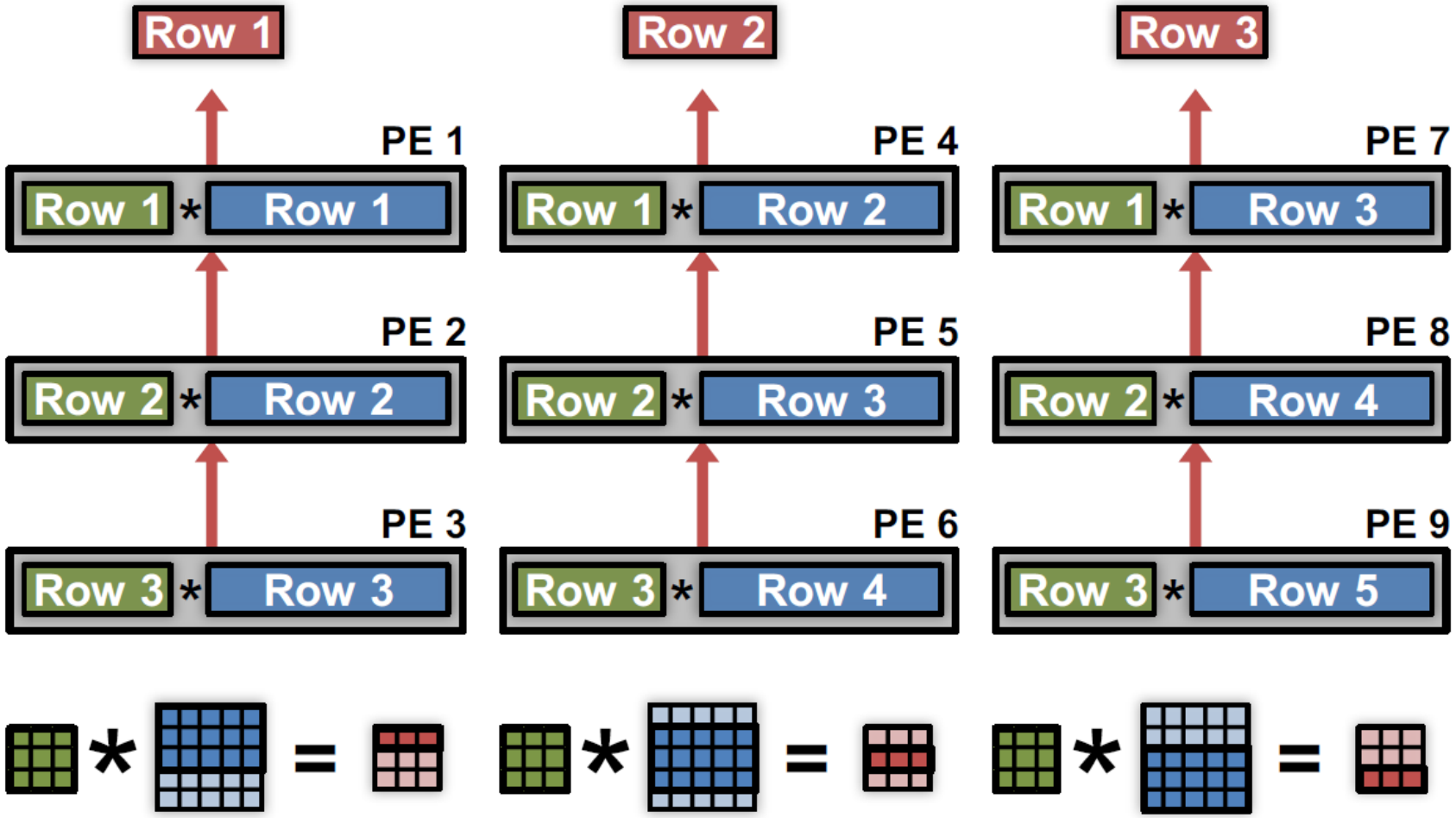
1-D Row Convolution in PE





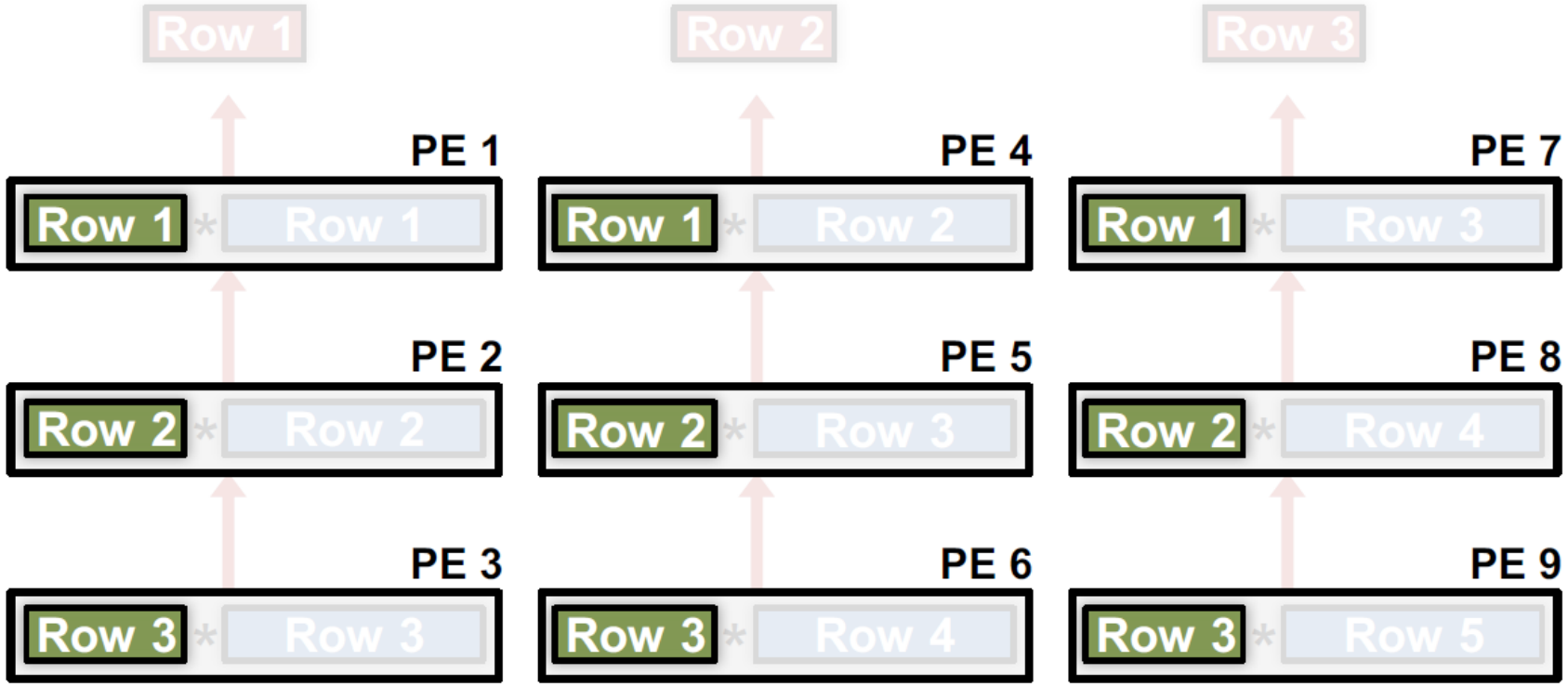
2-D Convolution in PE Array

- Each PE computes on a filter row and an input row



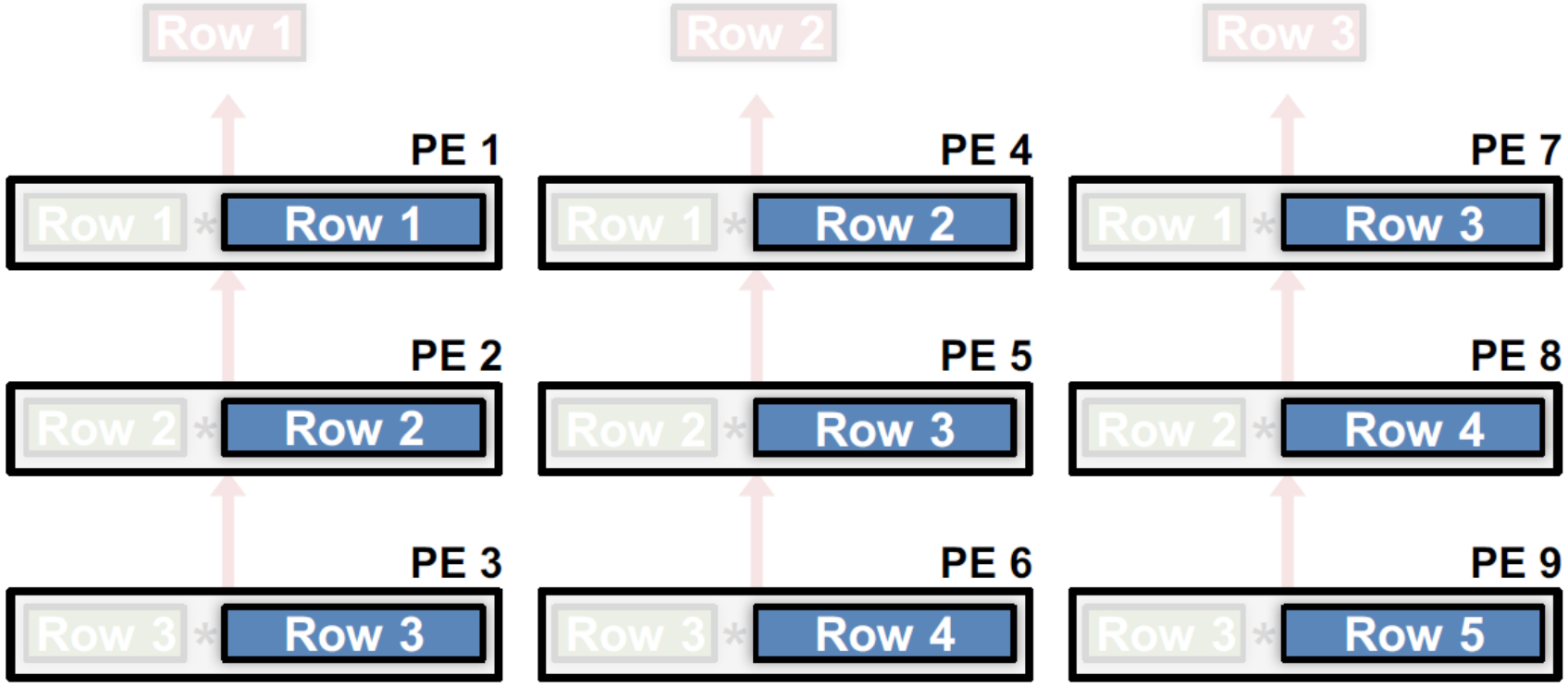
Convolutional Reuse

- Filter rows are reused across PEs horizontally



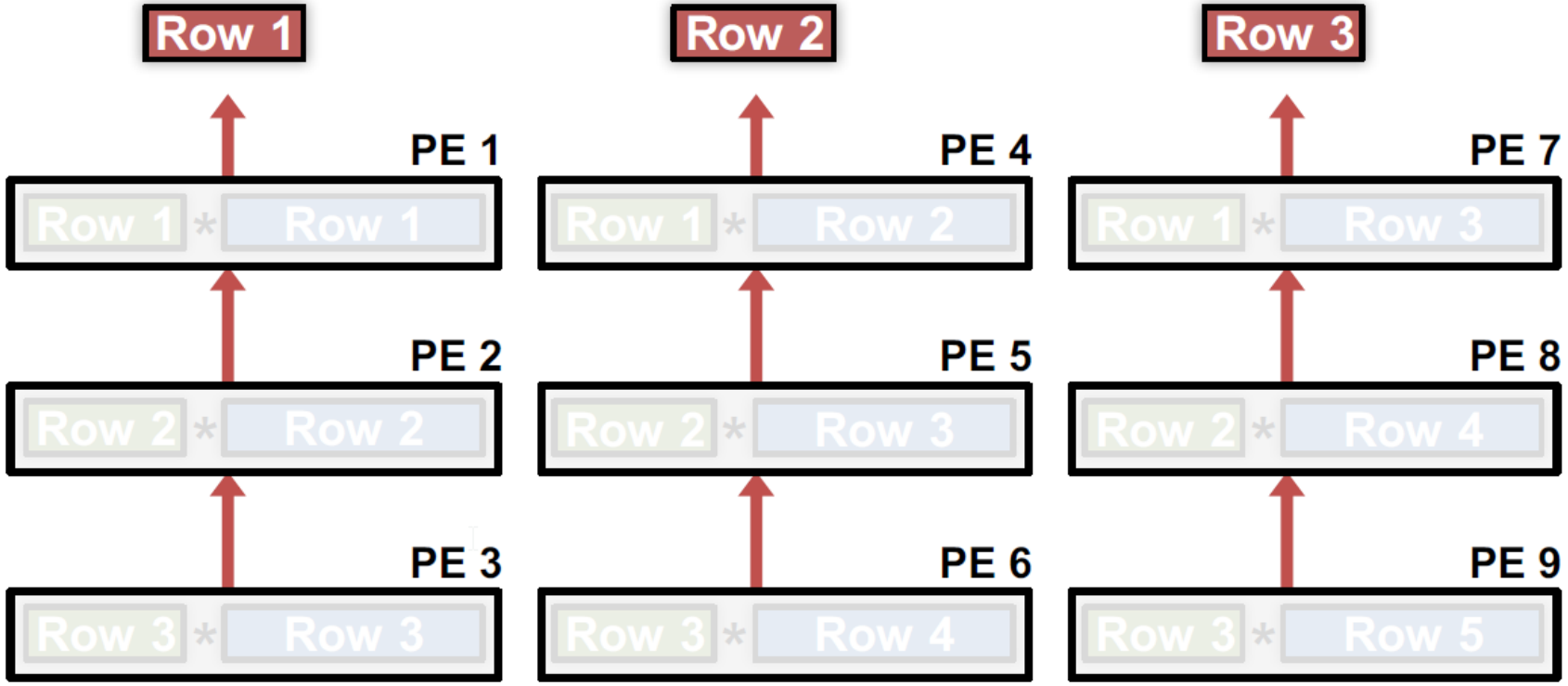
Convolutional Reuse

- Image rows are reused across PEs diagonally

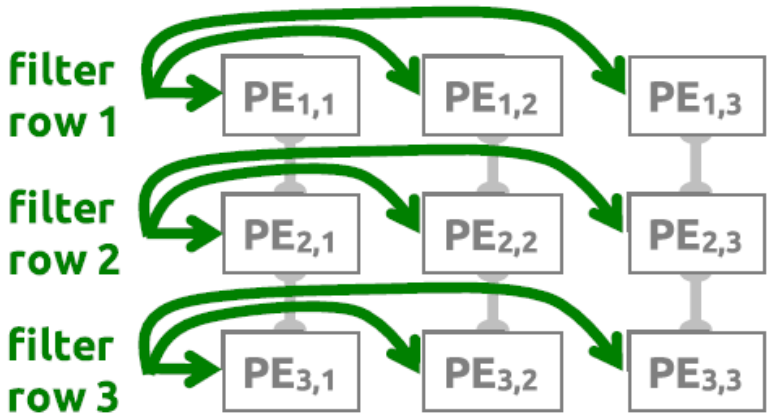


2-D Accumulation

- Partial sums accumulate across PEs vertically



Row Stationary Summary



Weight rows are shared **horizontally**

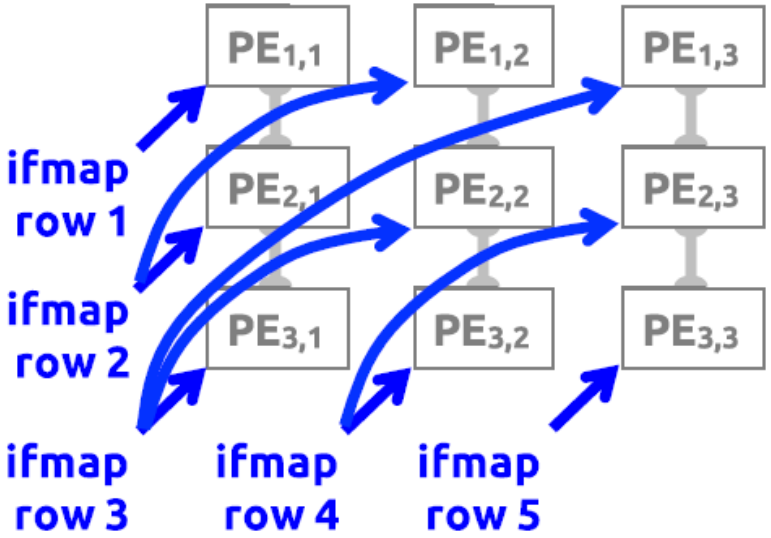
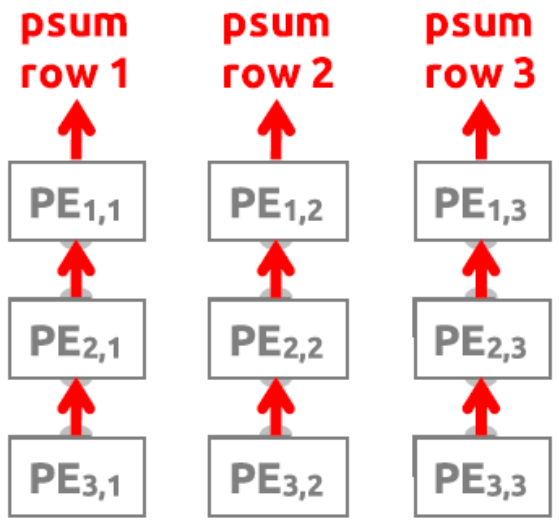
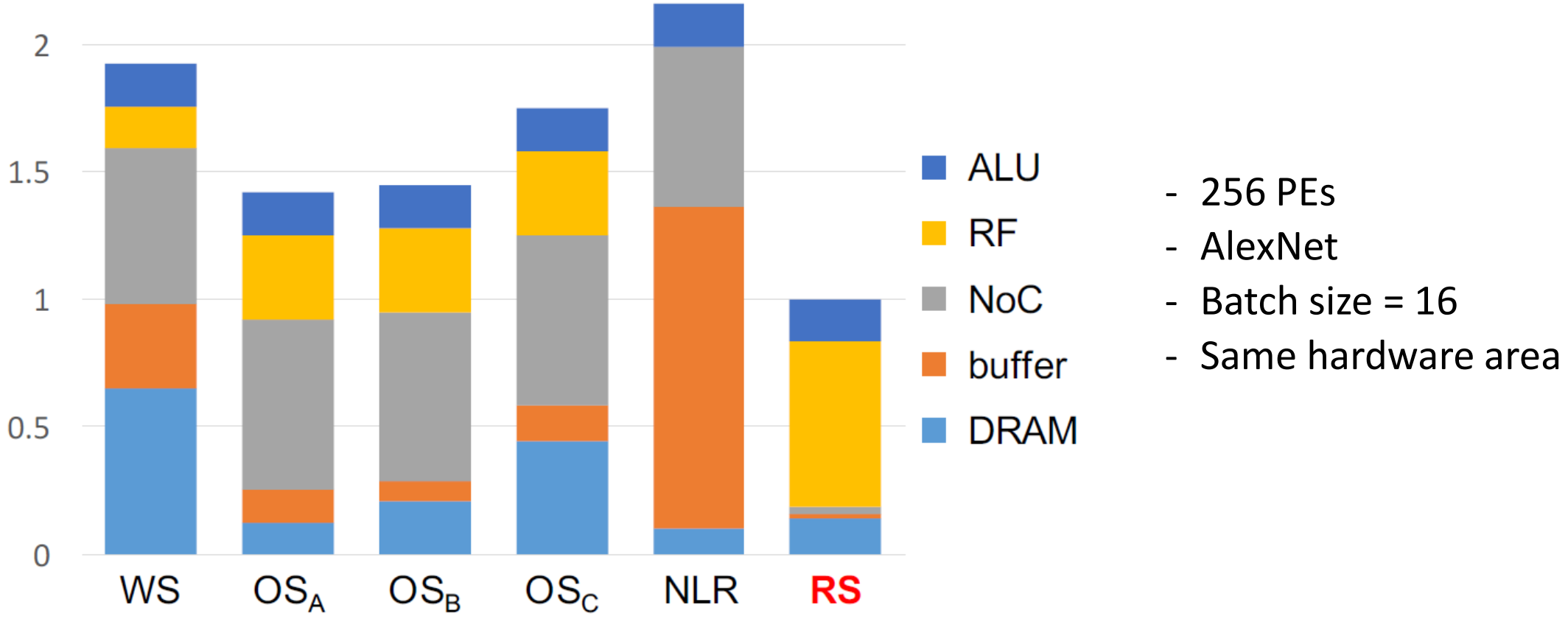


Image rows are shared **diagonally**



Partial sums are accumulated **vertically**

Simulation Results

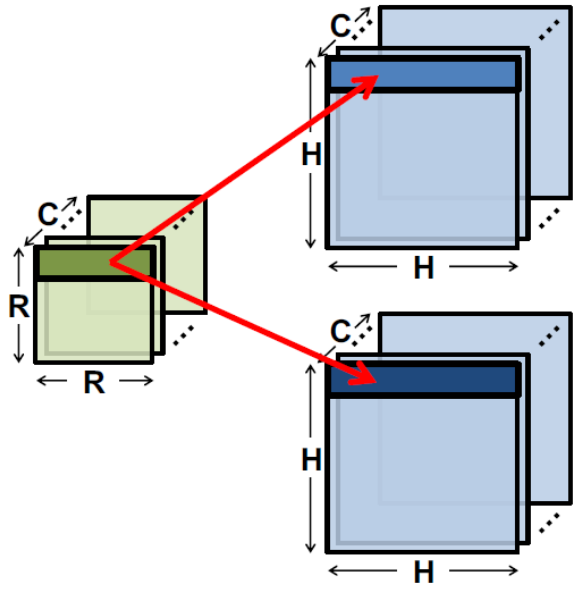


- 256 PEs
- AlexNet
- Batch size = 16
- Same hardware area

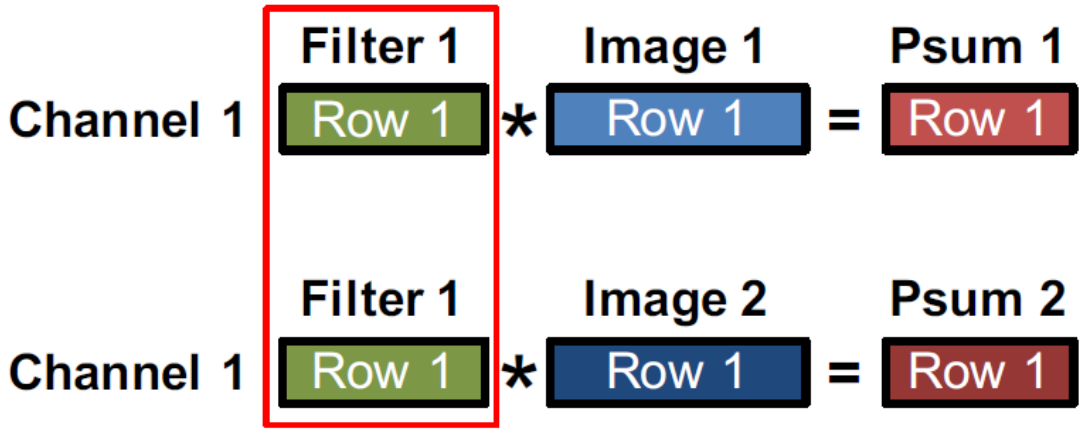
RS is 1.4x–2.5x more energy efficient than other dataflows

Filter Reuse

- Multiple images



Same filter for different images



Share the same filter row &
Concatenate rows from different images

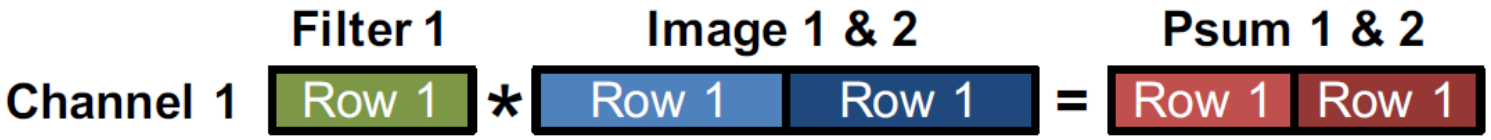
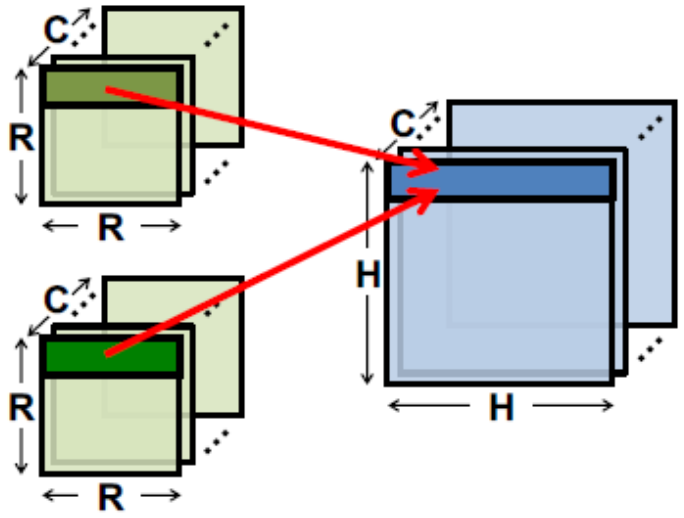
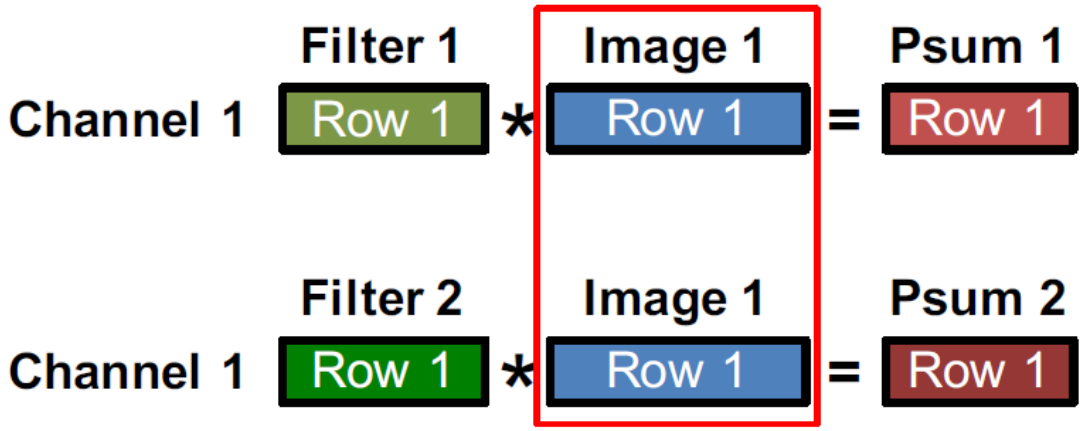


Image Reuse

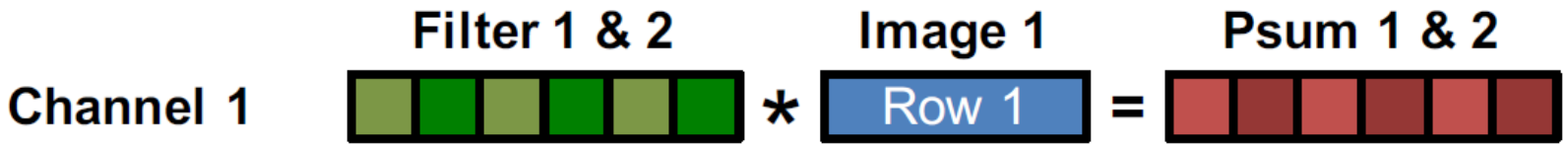
- Multiple filters



Same image for different filters

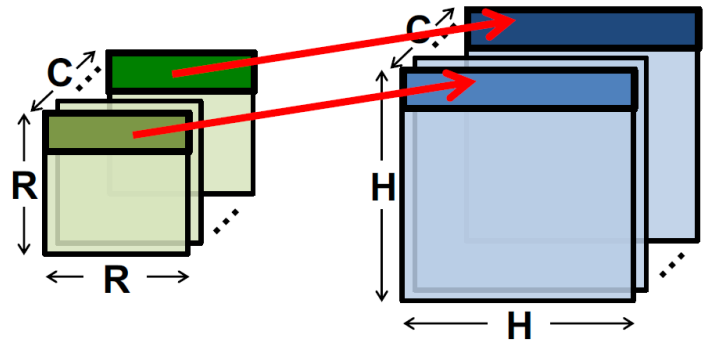


Share the same image row &
Interleave filter rows

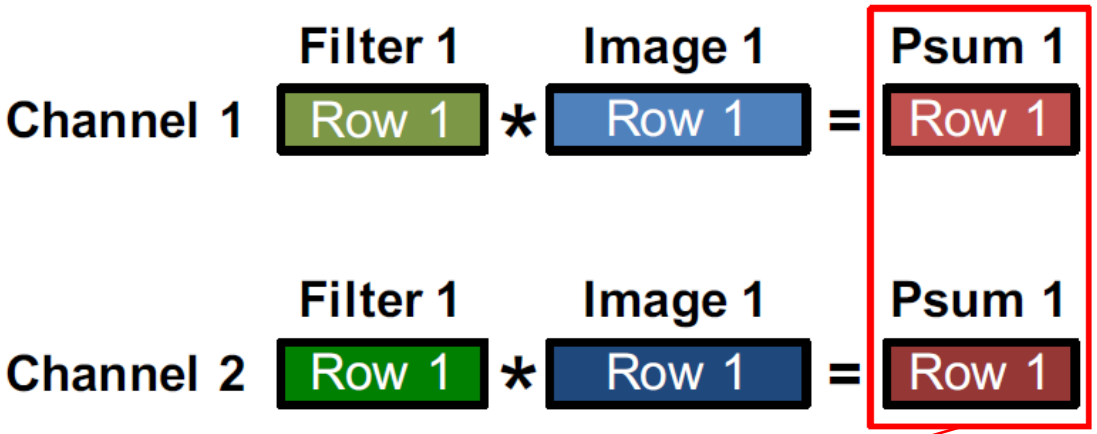


Channel Accumulation

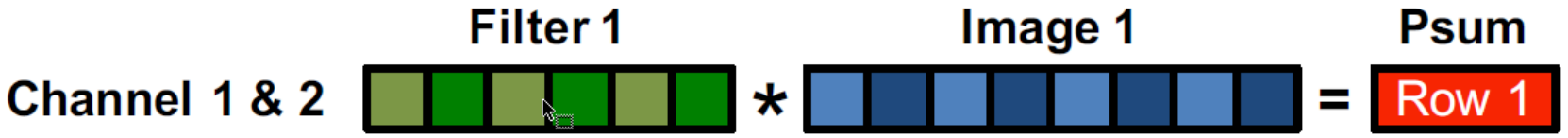
- Multiple channels



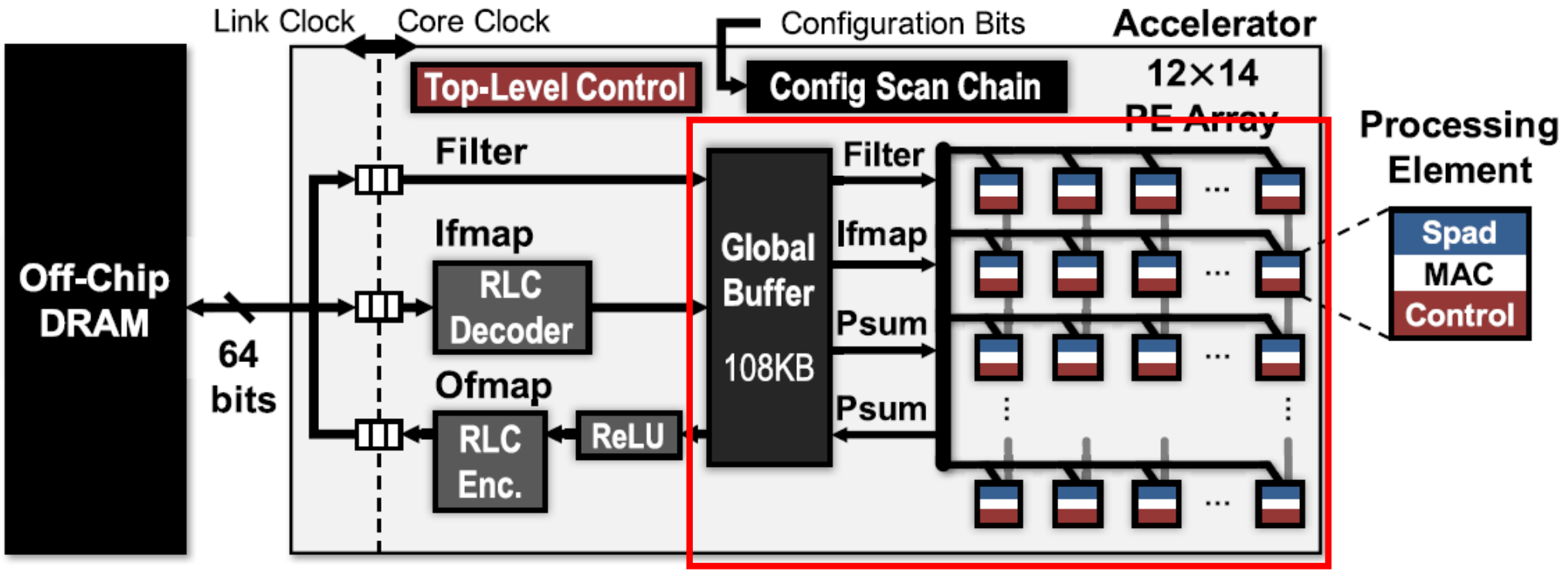
Need to accumulate Psums



Interleave channels



Hardware Architecture

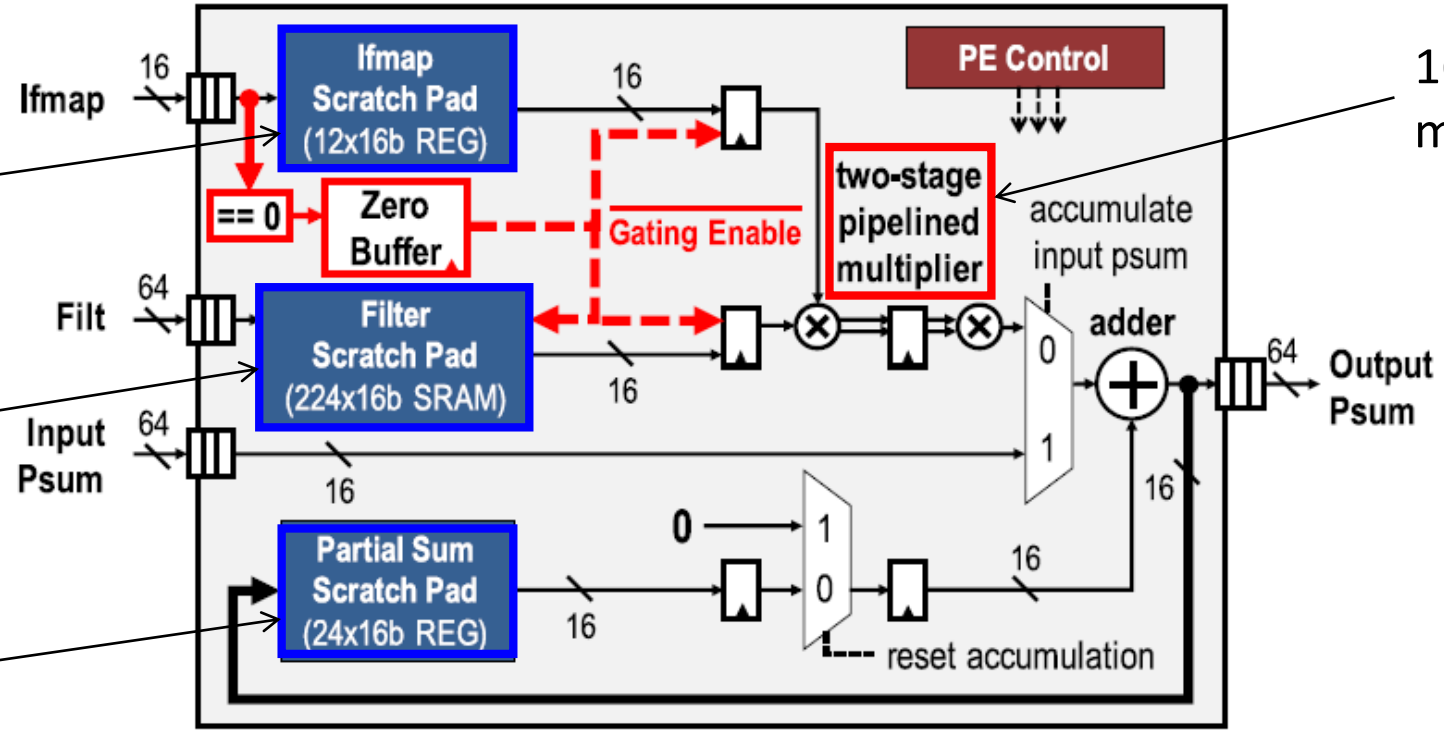


108KB
Global Buffer

12 x 14 PE Array

PE Architecture

- Have minimum set of scratchpads to run p (# of filters) and q (# of channels) primitives at the same time



Register file: store only a row

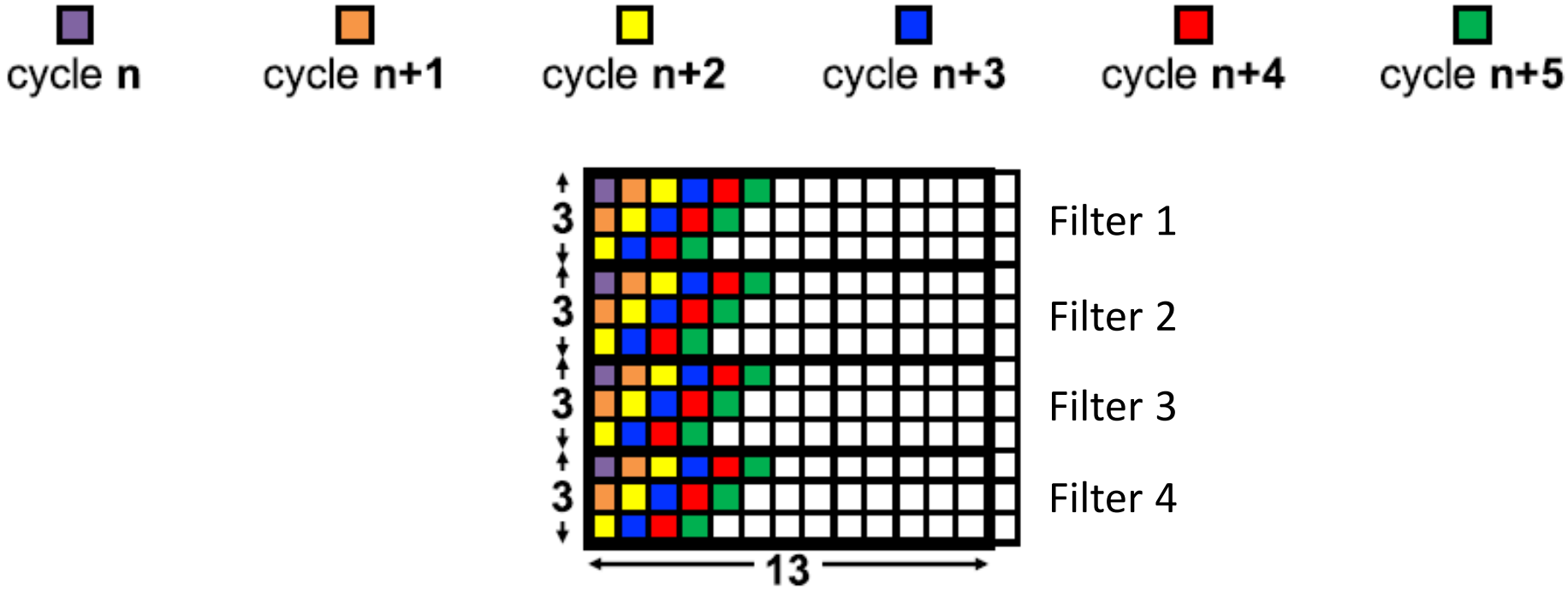
SRAM: large to have (p x q) filters

Register file: store only a row or a couple

16-bit two stage multiplier

Data Mapping to Spatial Array

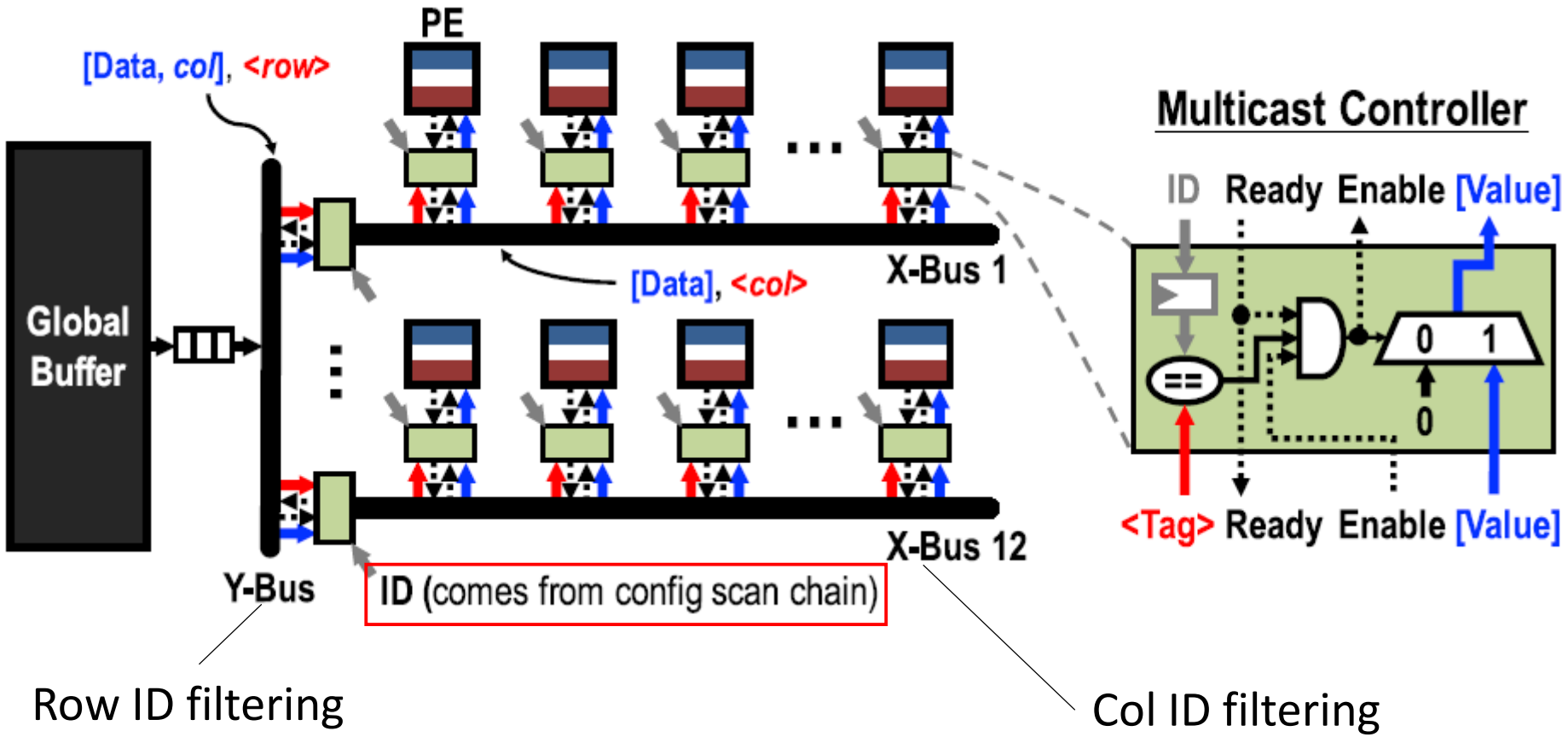
- If you want to map a CONV3 layer of AlexNet..
  - Image data needs to go diagonally
  - Run 13 pixels in a row and 4 filters at the same time based on 12 x 14 PE array



PEs with same color receives same image data

Global Input Network

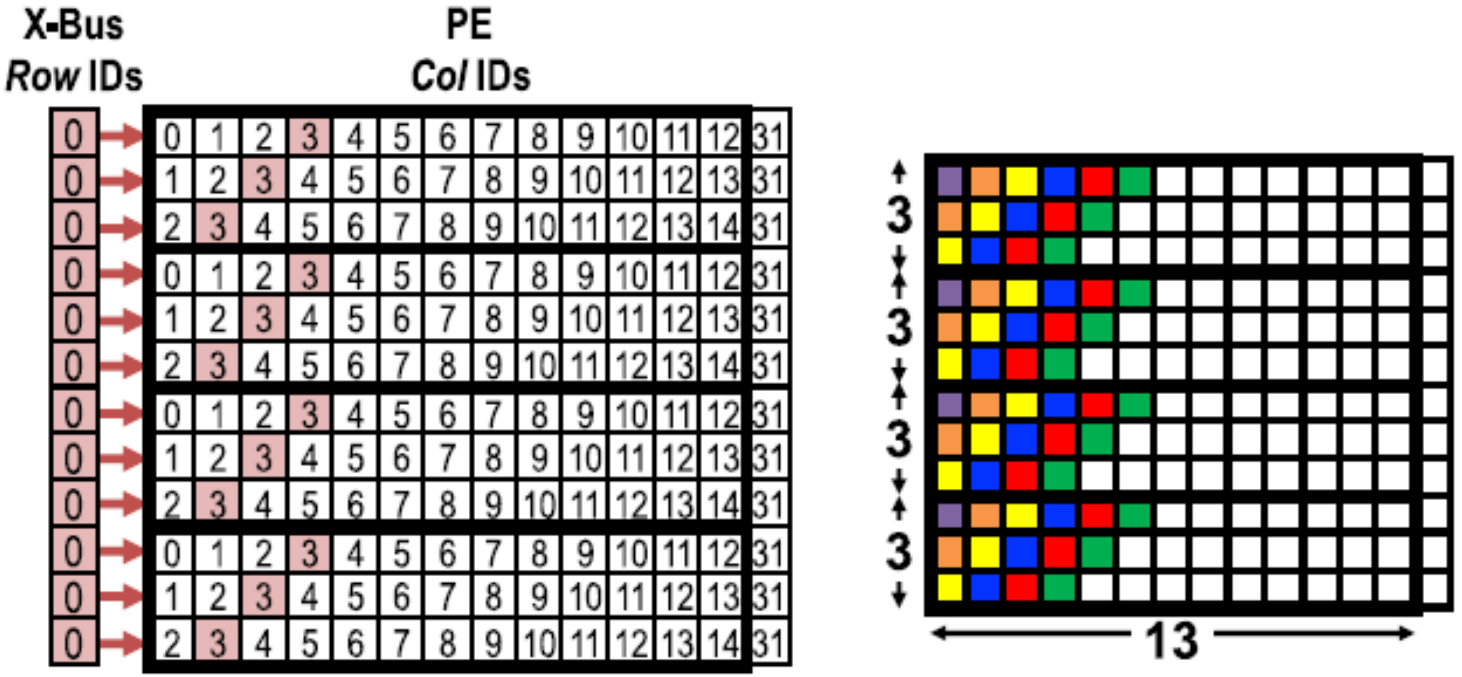
- Each data from global buffer is tagged with a (Row ID, Col ID) to send data only to desired PEs



ID Configuration

- Top controller configs the Row ID and Col ID through config scan chain
- Send (Row ID, Col ID) tag together with image data

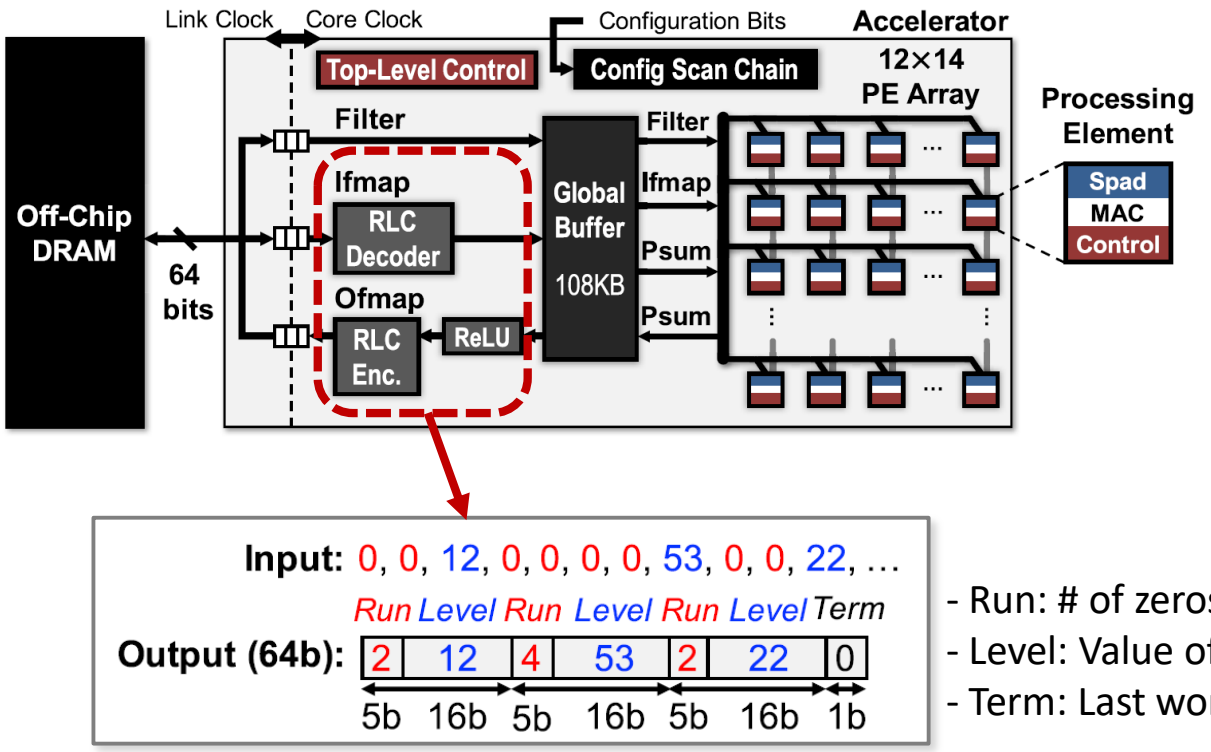
Send image data ■ with tag (0, 3)

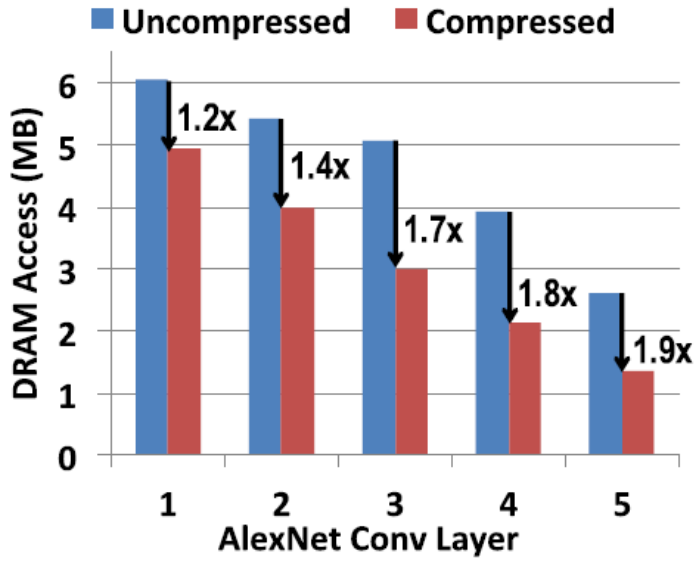


Various convolution kernels (3x3, 5x5, ..) can be mapped with updating configurations

Run-Length Compression

- Compression reduces DRAM accesses that are the most energy consuming data movement
- Consecutive zeros are represented using only 5 bits number (Run) while the original data length is 16 bits

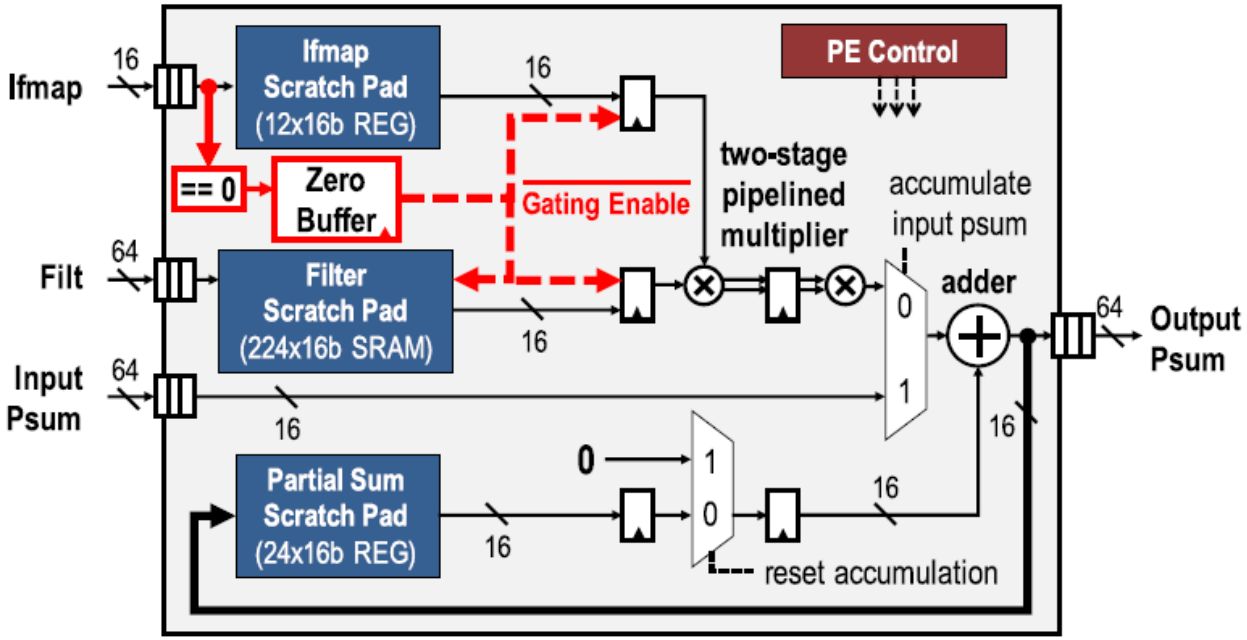




- Run: # of zeros
- Level: Value of non-zero data
- Term: Last word Indicator

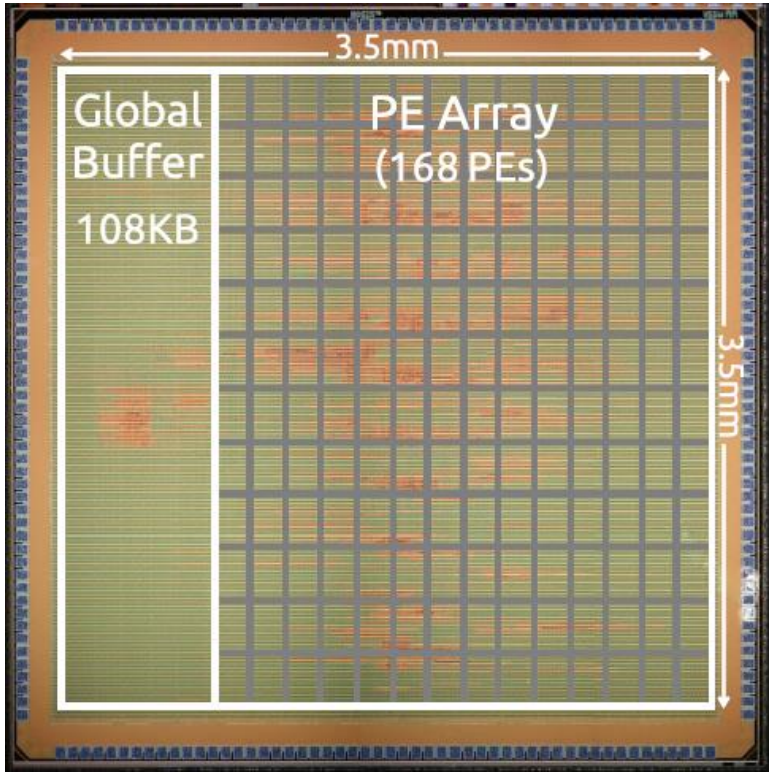
Zero Skipping

- After activation such as ReLU, feature maps tend to have more zeros (=sparser)



- If the input feature map data is zero, zero buffer records its location
- When the zero data is read, registers for MAC will be gated
- Help achieve low-power consumption

Implementation Result



Technology	TSMC 65nm LP 1P9M
Chip Size	4.0 mm × 4.0 mm
Core Area	3.5 mm × 3.5 mm
Gate Count (logic only)	1176k (2-input NAND)
On-Chip SRAM	181.5K bytes
Number of PEs	168
Global Buffer	108.0K bytes (SRAM)
Scratch Pads (per PE)	filter weights: 448 bytes (SRAM) feature maps: 24 bytes (Registers) partial sums: 48 bytes (Registers)
Supply Voltage	core: 0.82–1.17 V I/O: 1.8 V
Clock Rate	core: 100–250 MHz link: up to 90 MHz
Peak Throughput	16.8–42.0 GMACS
Arithmetic Precision	16-bit fixed-point
Natively Supported CNN Shapes	filter height (R): 1–12 filter width (S): 1–32 num. of filters (M): 1–1024 num. of channels (C): 1–1024 vertical stride: 1, 2, 4 horizontal stride: 1–12

Discussion

- Programmability of DianNao and Eyeriss
  - How general are these accelerators? Can you map other algorithms to the accelerators?
  - Do you have any ideas to make the accelerators more programmer friendly?

- Scalability of DianNao and Eyeriss
  - If we scale Eyeriss's PE array to 32 x 32, will it perform better? If yes, how much?

- Limitations of DianNao and Eyeriss
  - What are the limitations and weaknesses of the accelerators?
  - If you were the designer, how can you make them better and why?

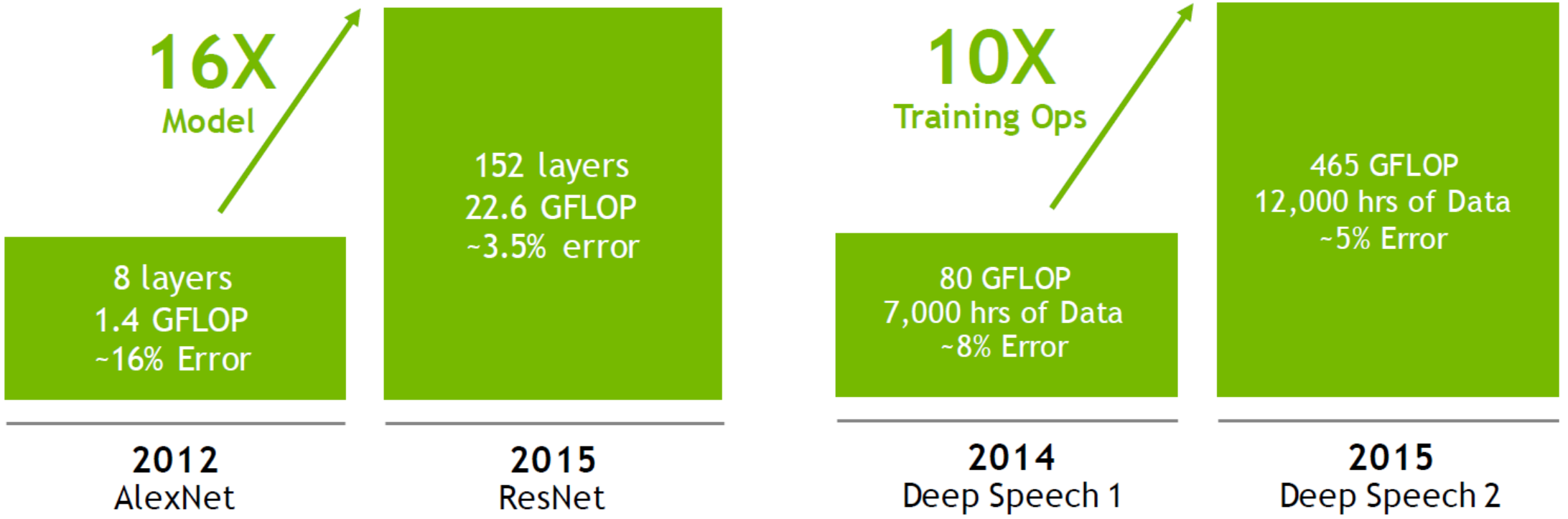
- DianNao vs Eyeriss
  - If you were a user/customer/company owner, which processor are you going to choose and why?

ML Accelerators for Mobile/Edge - II

- EIE (ISCA 2016)
  - S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," ISCA 2016
  - S. Han, J. Pool, J. Tran, W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," NIPS 2015
  - S. Han, H. Mao, W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," ICLR 2016
  - Image Credit: S. Han, "Efficient Methods and Hardware for Deep Learning," Stanford CS231n Lecture Slides
- UNPU (JSSC 2019)
  - J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, H.-J. Yoo, "UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision," JSSC 2019

Motivation: Model Size

- Models are getting larger and larger!



Motivation: Speed and Cost

- Training time with fb.resnet.torch using 4 M40 GPUs (2017)

	Error rate	Training time
ResNet 18	10.76%	2.5 days
ResNet 50	7.02%	5 days
ResNet 101	6.21%	1 week
ResNet 152	6.16%	1.5 weeks

- AlphaGo has 1920 CPUs and 280 GPUs, $3000 electric bill per game (2016)

Latest Numbers

- DAWNBench (https://dawn.cs.stanford.edu/benchmark/)

Image Classification on ImageNet

Training Time

All Submissions

Objective: Time taken to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.

Rank	Time to 93% Accuracy	Model	Hardware	Framework
1 Mar 2020	0:02:38	ResNet50-v1.5 *Apsara AI Acceleration(AIACC) team in Alibaba Cloud* source	16 ecs.gn6e-c12g1.24xlarge (AlibabaCloud)	AIACC-Training 1.3 + Tensorflow 2.1
2 May 2019	0:02:43	ResNet-50 *ModelArts Service of Huawei Cloud* source	16 nodes with InfiniBand (8*V100 with NVLink for each node)	Moxing v1.13.0 + TensorFlow v1.13.1
3 Dec 2018	0:09:22	ResNet-50 *ModelArts Service of Huawei Cloud* source	16 * 8 * Tesla-V100(ModelArts Service)	Huawei Optimized MXNet
4 Sep 2018	0:18:06	ResNet-50 *fast.ai/DIUx (Yaroslav Bulatov, Andrew Shaw, Jeremy Howard)* source	16 p3.16xlarge (AWS)	PyTorch 0.4.1
5 Sep 2018	0:18:53	Resnet 50 *Andrew Shaw, Yaroslav Bulatov, Jeremy Howard* source	64 * V100 (8 machines - AWS p3.16xlarge)	ncluster / Pytorch 0.5.0a0+0e8088d

Training Cost

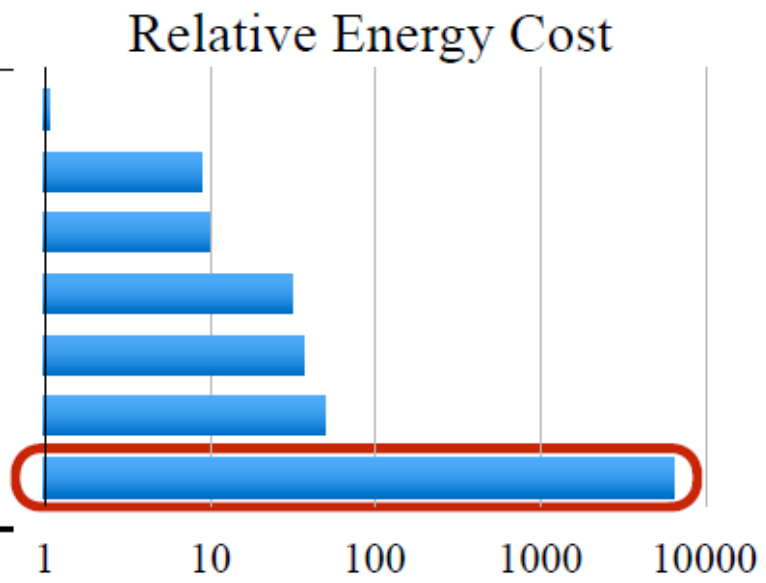
All Submissions

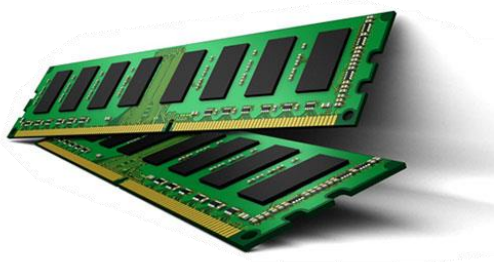
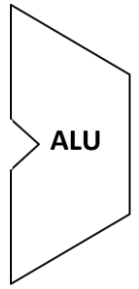
Objective: Total cost of public cloud instances to train an image classification model to a top-5 validation accuracy of 93% or greater on ImageNet.

Rank	Cost (USD)	Model	Hardware	Framework
1 Mar 2020	$7.43	ResNet50-v1.5 *Apsara AI Acceleration(AIACC) team in Alibaba Cloud* source	1 ecs.gn6e-c12g1.24xlarge (AlibabaCloud)	AIACC-Training 1.3 + Tensorflow 2.1
2 Sep 2018	$12.60	ResNet50 *Google Cloud TPU* source	GCP n1-standard-2, Cloud TPU	TensorFlow v1.11.0
3 Mar 2020	$14.42	ResNet50-v1.5 *Apsara AI Acceleration(AIACC) team in Alibaba Cloud* source	16 ecs.gn6e-c12g1.24xlarge (AlibabaCloud)	AIACC-Training 1.3 + Tensorflow 2.1
4 Aug 2019	$19.00	Resnet 50 *Chuan Li* source	Lambda GPU Cloud - 4x GTX 1080 Ti	ncluster / Pytorch 1.0.0
5 Apr 2019	$20.89	ResNet50 *Setu Chokshi (MS AI MVP | PropertyGuru)* source	Azure ND40s_v2	PyTorch 1.0

Where is the Energy Consumed?

Operation	Energy [pJ]
32 bit int ADD	0.1
32 bit float ADD	0.9
32 bit Register File	1
32 bit int MULT	3.1
32 bit float MULT	3.7
32 bit SRAM Cache	5
32 bit DRAM Memory	**640**

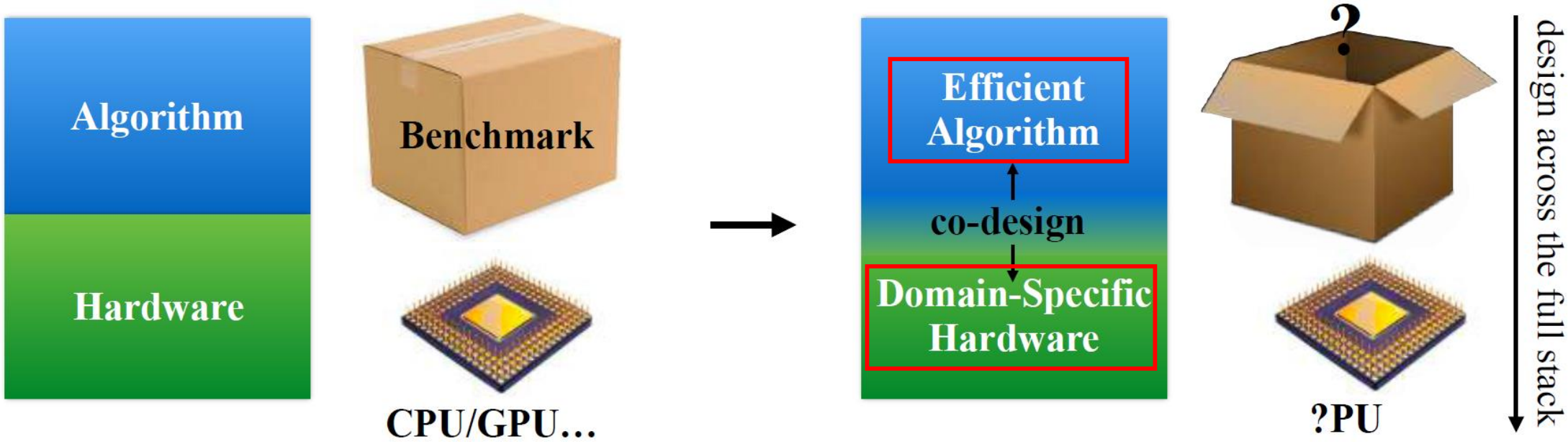


1 x  ≈ 1000 x 

Large Model = More Energy → Need to Compress Model!

SW-HW Co-Design

- Improving energy efficiency
  - Separated computing stack for old benchmarks
  - Break the boundary between algorithm and hardware



Compressed DNN Models with Specialized Hardware

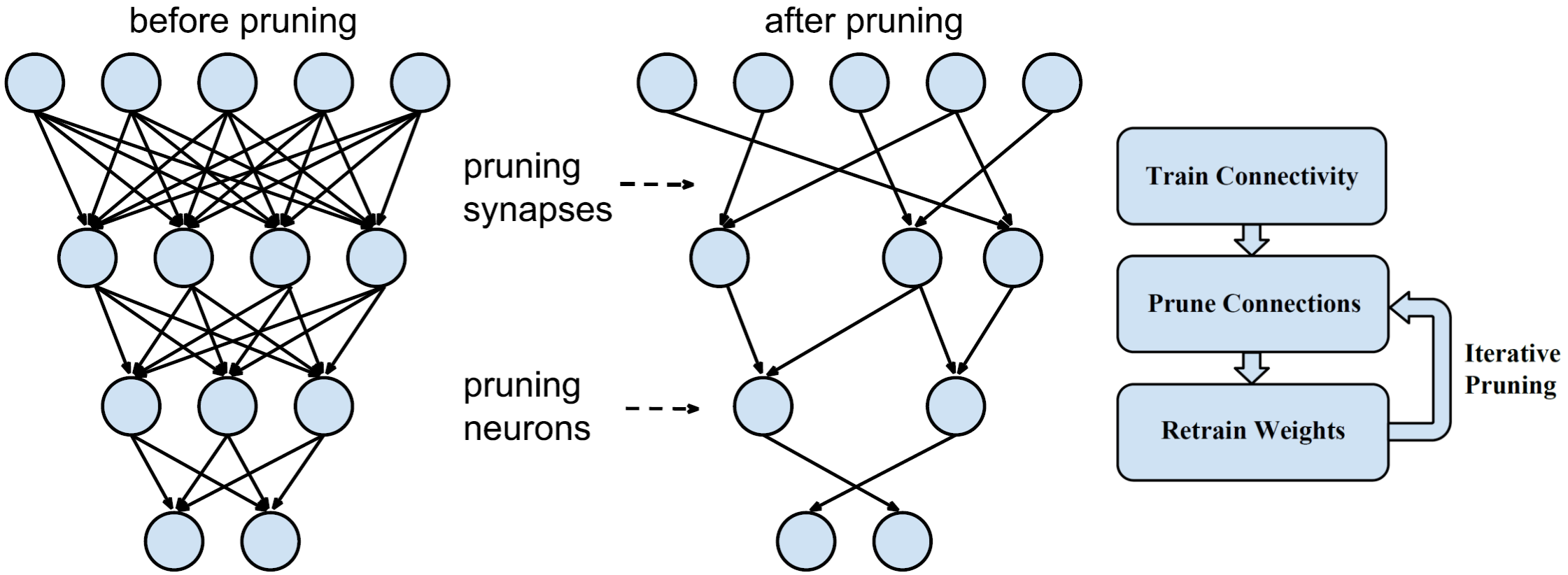
Deep Compression

- A method to compress DNN models without loss of accuracy through a combination of pruning and weight sharing

Network	Top-1 Error	Top-5 Error	Model Size	Compress Rate
LeNet-300-100	1.64%	-	1070 KB	
LeNet-300-100 Compressed	1.58%	-	**27 KB**	**40×**
LeNet-5	0.80%	-	1720 KB	
LeNet-5 Compressed	0.74%	-	**44 KB**	**39×**
AlexNet	42.78%	19.73%	240 MB	
AlexNet Compressed	42.78%	19.70%	**6.9 MB**	**35×**
VGG-16	31.50%	11.32%	552 MB	
VGG-16 Compressed	31.17%	10.91%	**11.3 MB**	**49×**
Inception-V3	22.55%	6.44%	91 MB	
Inception-V3 Compressed	22.34%	6.33%	**4.2 MB**	**22×**
ResNet-50	23.85%	7.13%	97 MB	
ResNet-50 Compressed	23.85%	6.96%	**5.8 MB**	**17×**

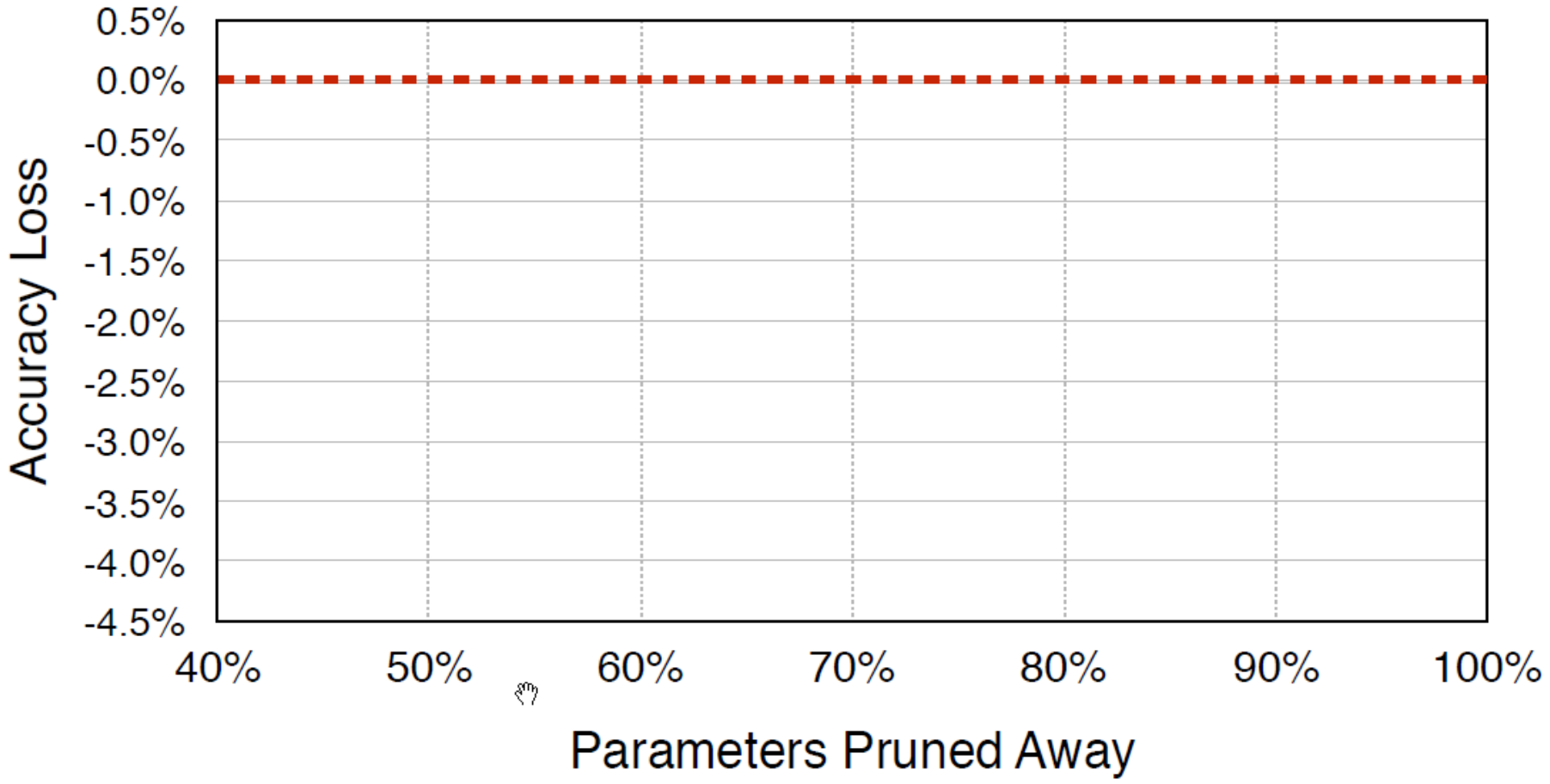
Pruning

- Process of masking out unnecessary synapses and neurons in DNN



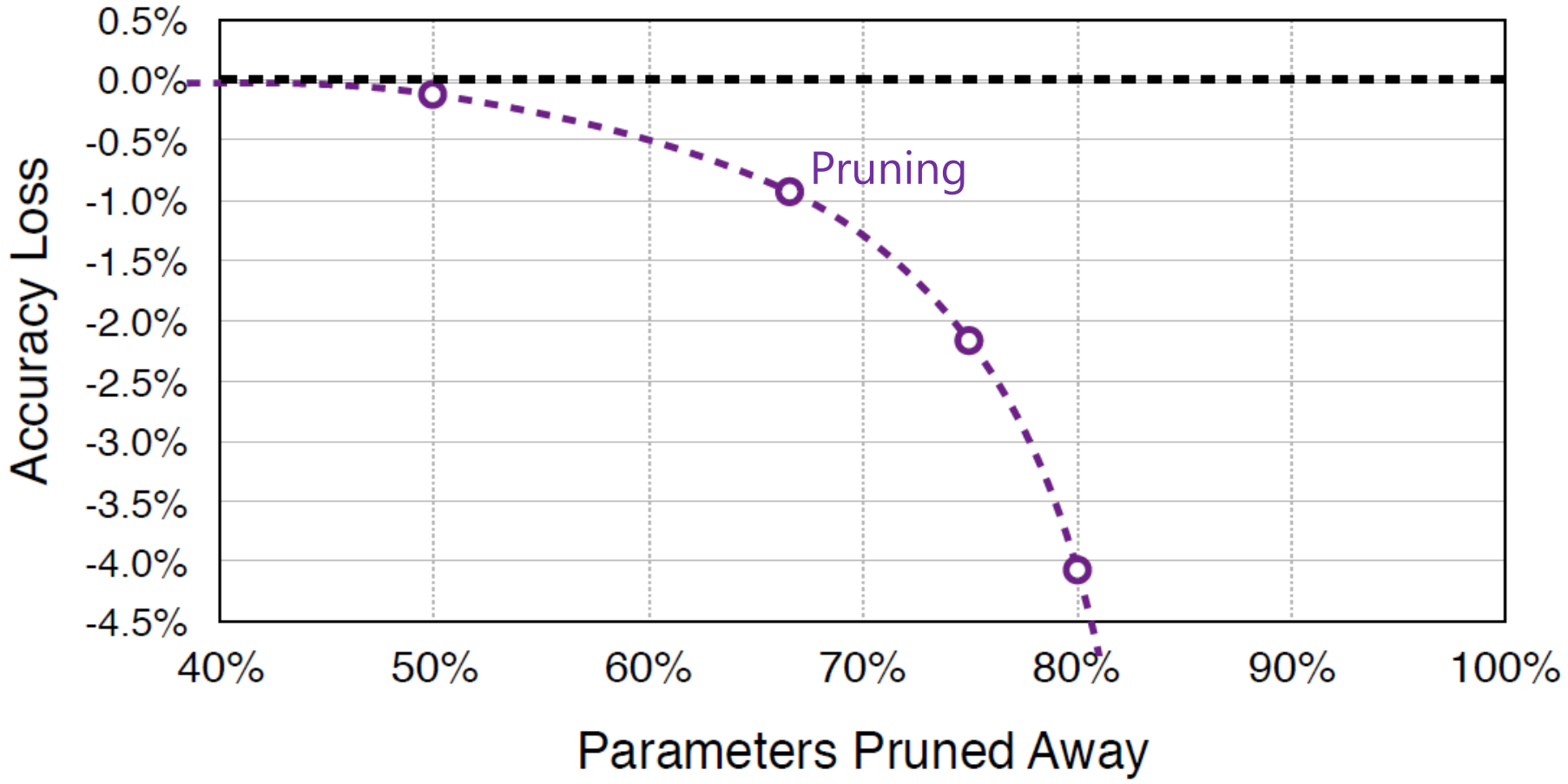
Pruning Neural Networks

- Train connectivity
  - Determine the importance of weight based on its absolute value



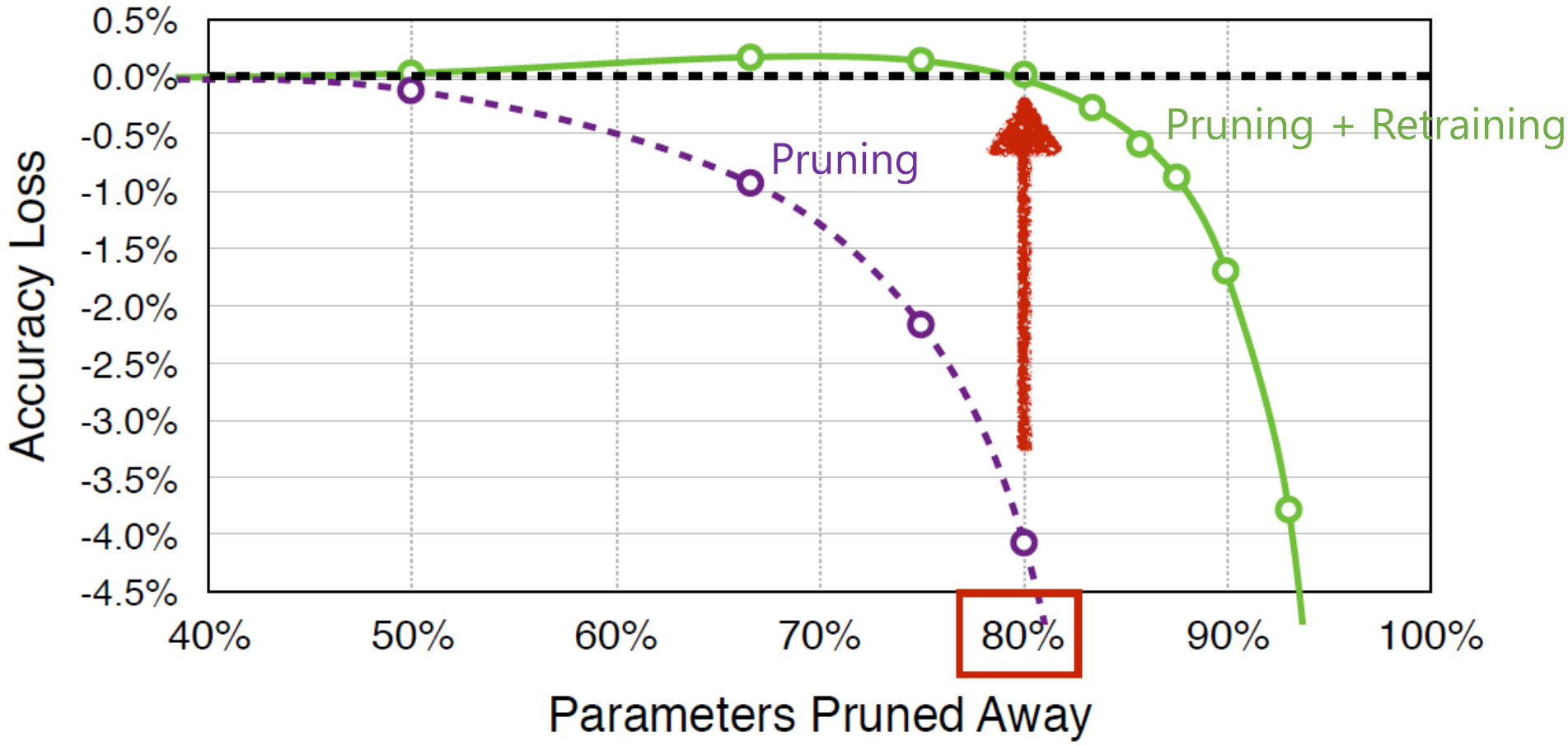
Pruning Neural Networks

- Prune connections
  - Prune connections that has weights below a threshold (hyper-parameter)
  - Threshold determines compression ratio and prediction accuracy



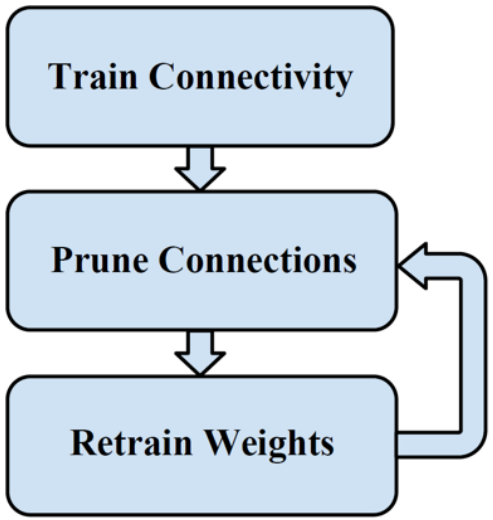
Pruning Neural Networks

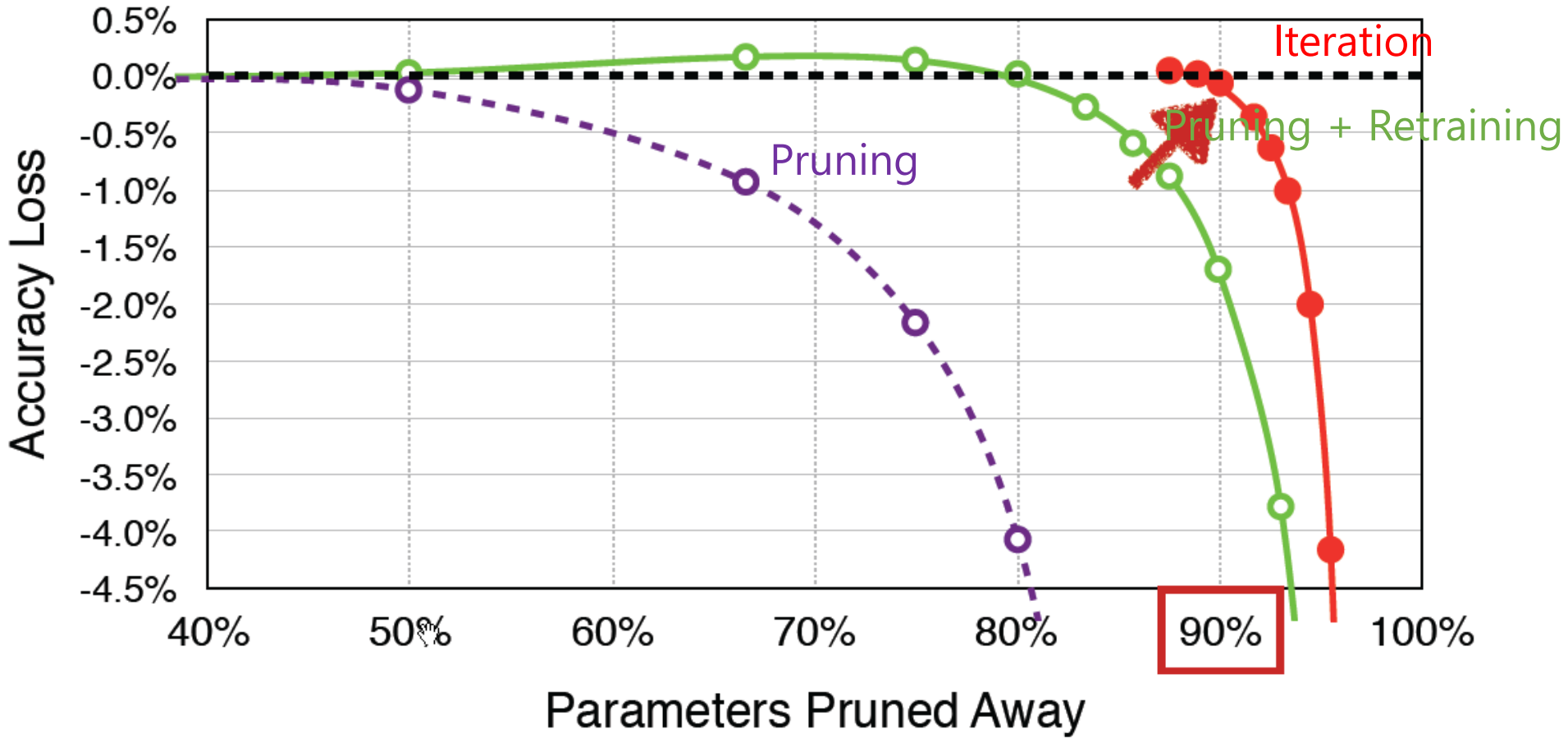
- Retrain weights
  - Train the weights for the pruned connections → without retraining, the accuracy will be significantly degraded



Pruning Neural Networks

- Iteratively retrain to optimize accuracy

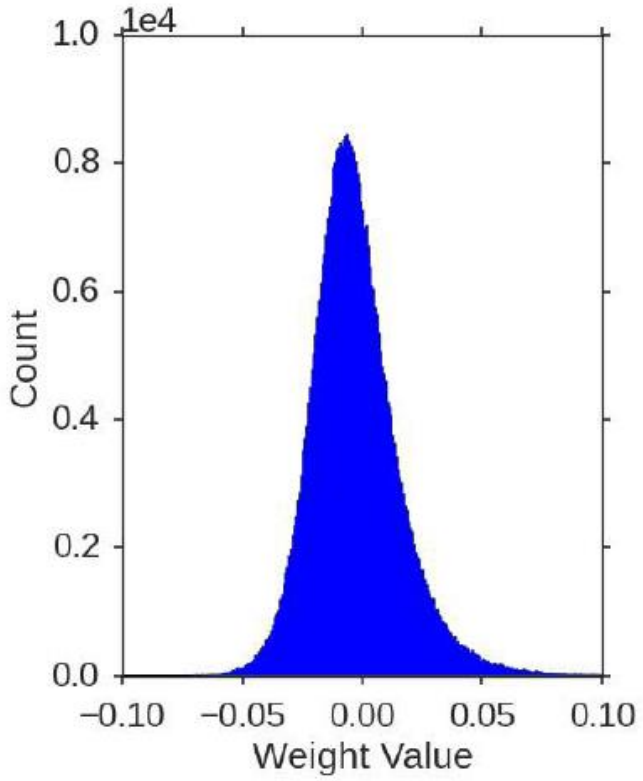




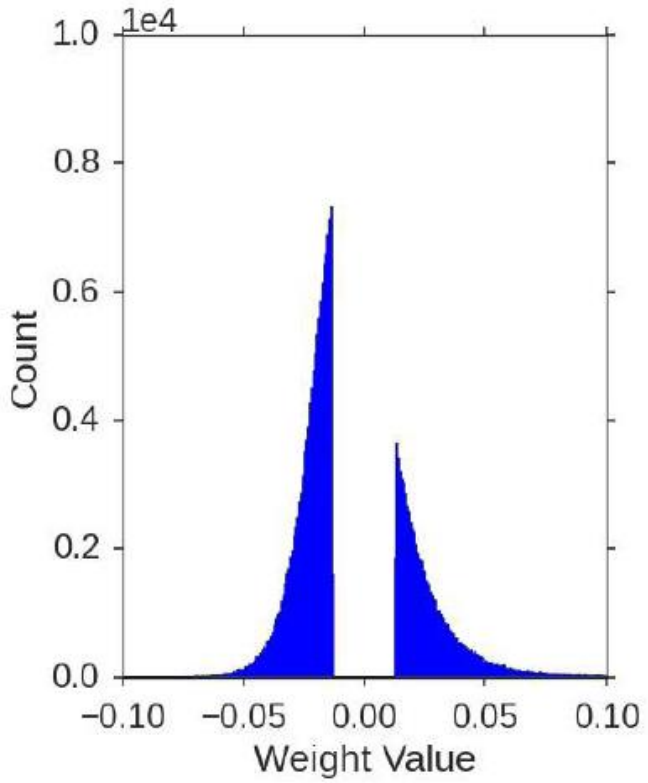
Model connections: 60M → 6M

Weight Distribution

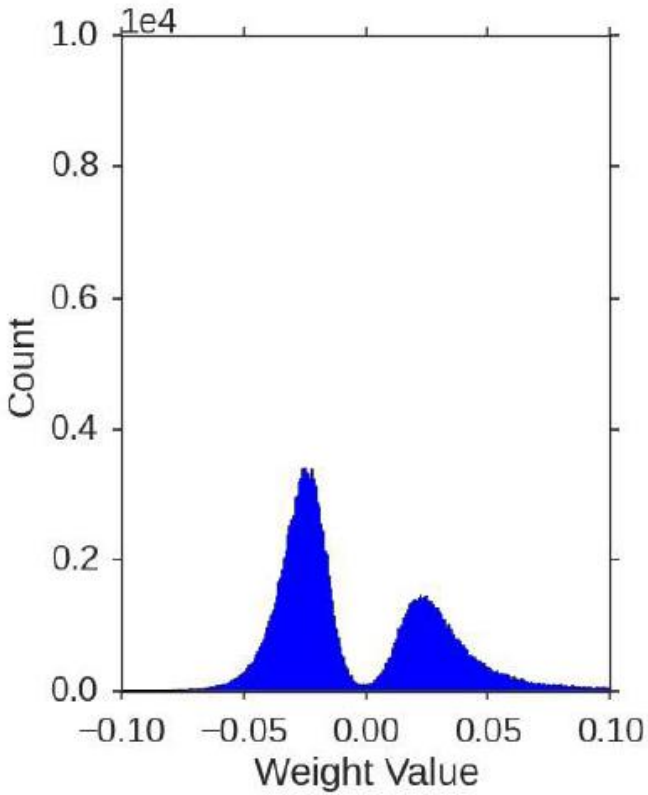
Original



After Pruning



After Retraining

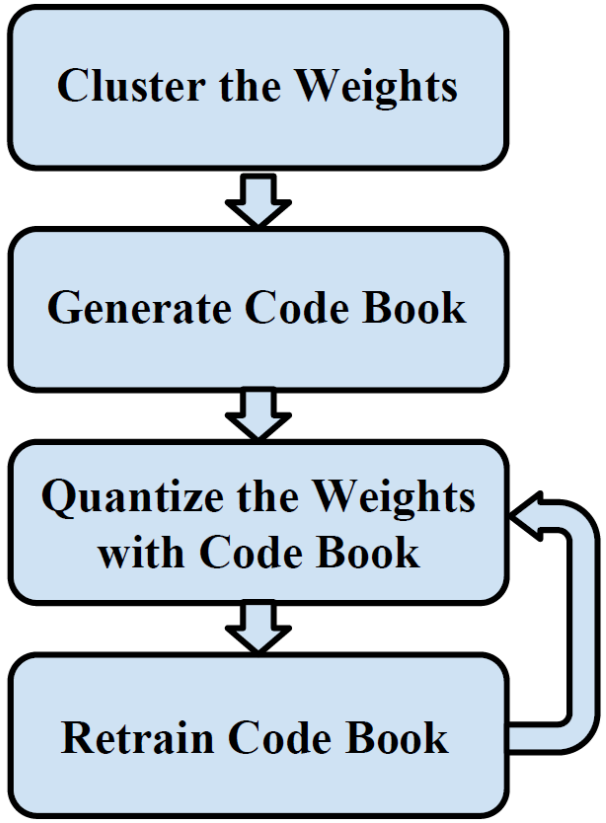


Quantization and Weight Sharing

- Reducing the number of bits required to represent each weight

2.09, 2.12, 1.92, 1.87, …

2.0

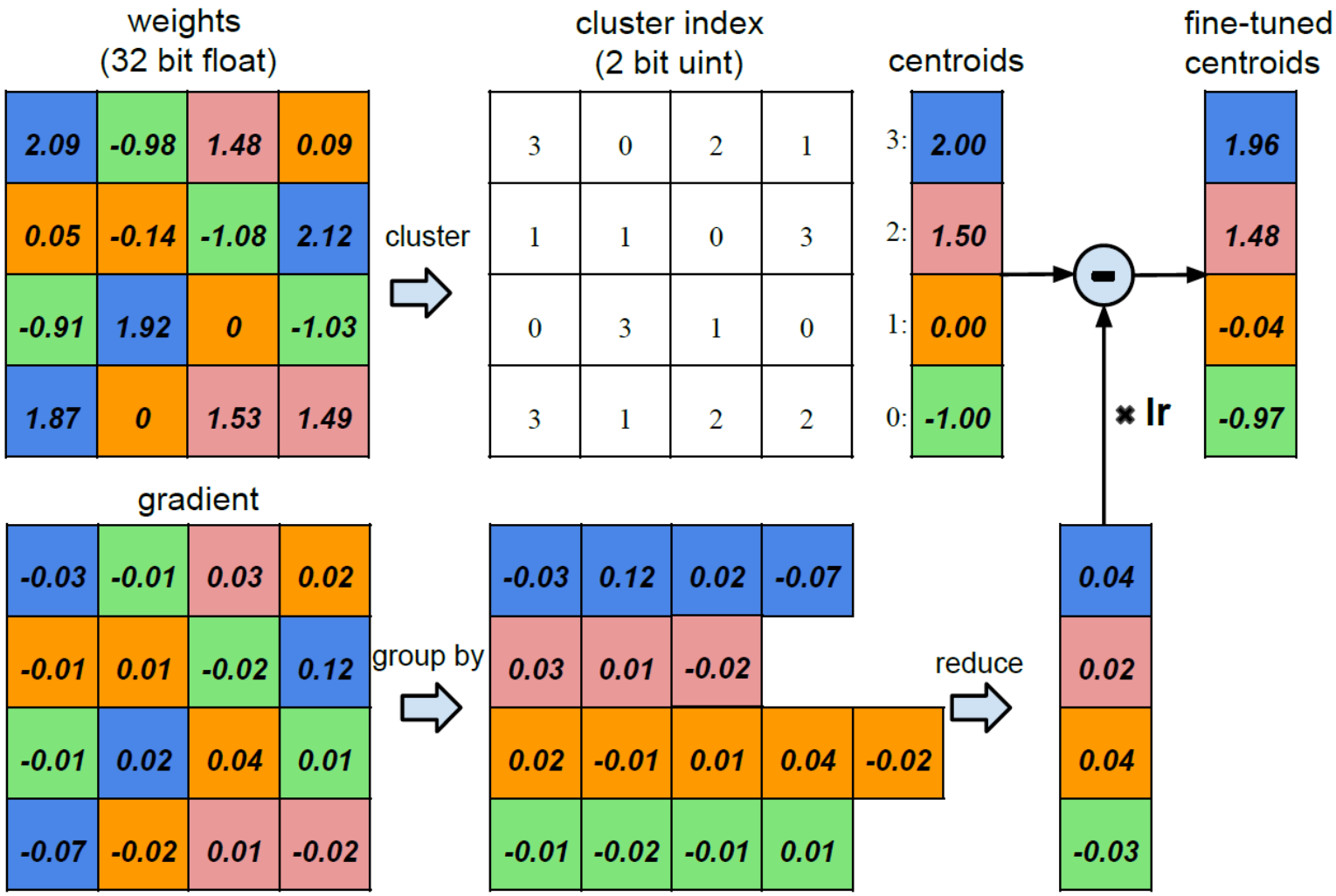


32 bit

4 bit

8x less memory!

Quantization and Weight Sharing



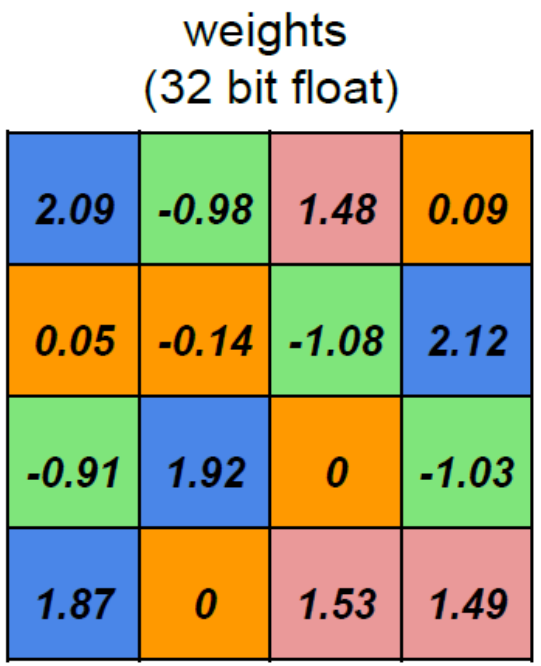
Clustering Weights

- Similar weights are grouped by *K-means clustering

weights
(32 bit float)

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

cluster

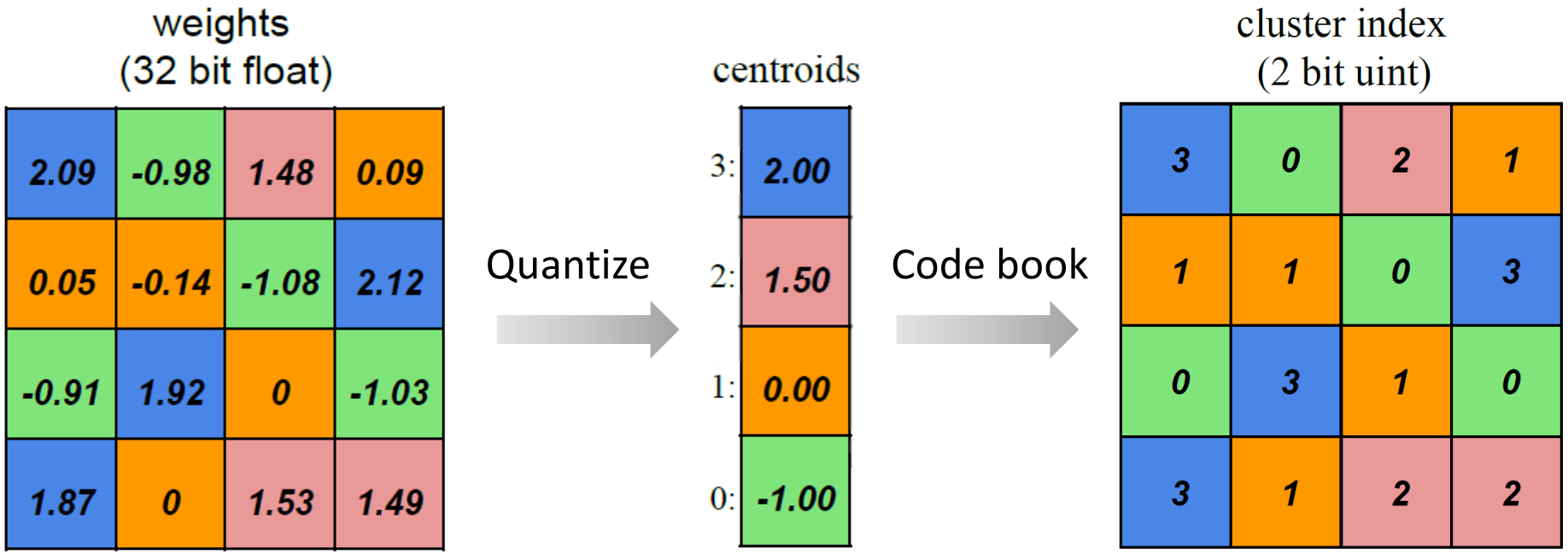


*K-means clustering: partitioning N original weights $W = \{w_1, w_2, \ldots, w_N\}$ into K clusters $C = \{c_1, c_2, \ldots, c_K\}$, N>K as to minimize the within-cluster sum of squares

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2$$

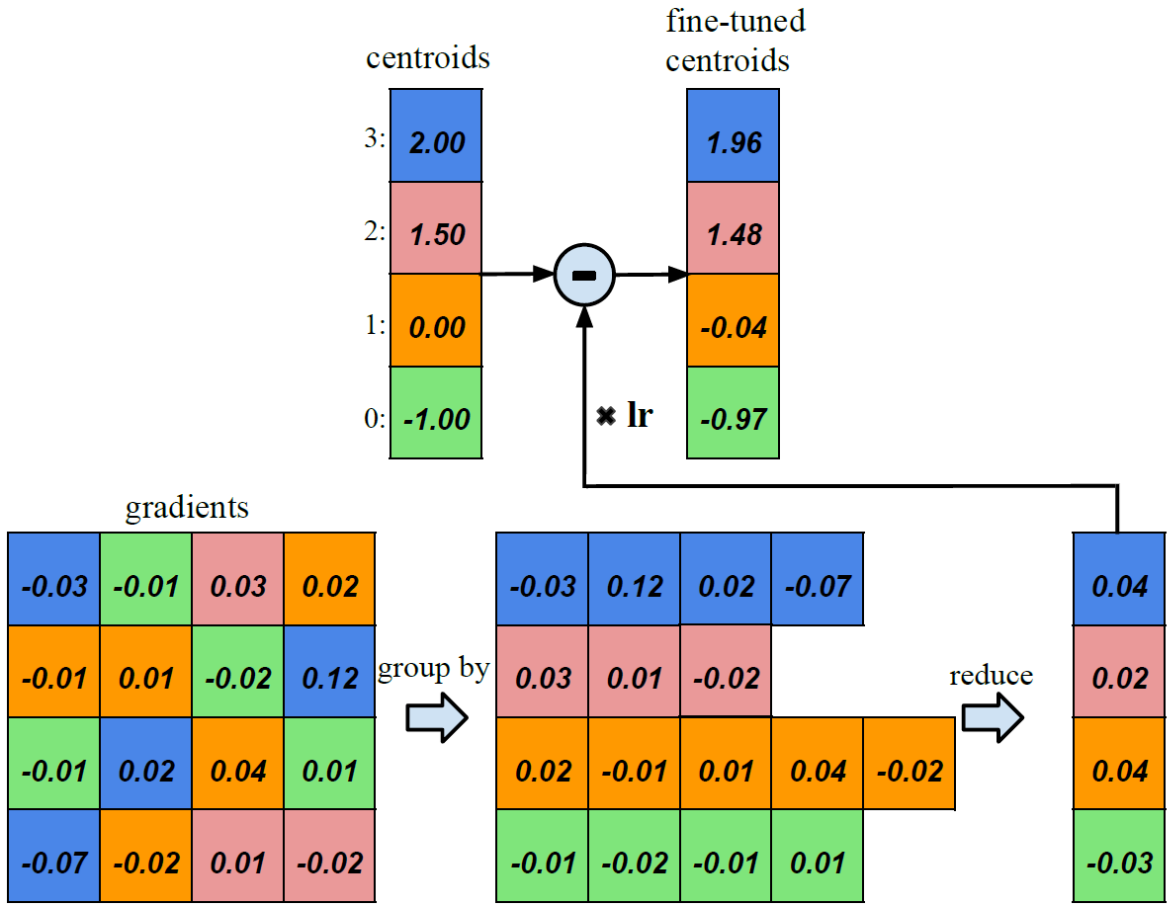
Quantization & Codebook Generation

- Quantize the weights and make a codebook



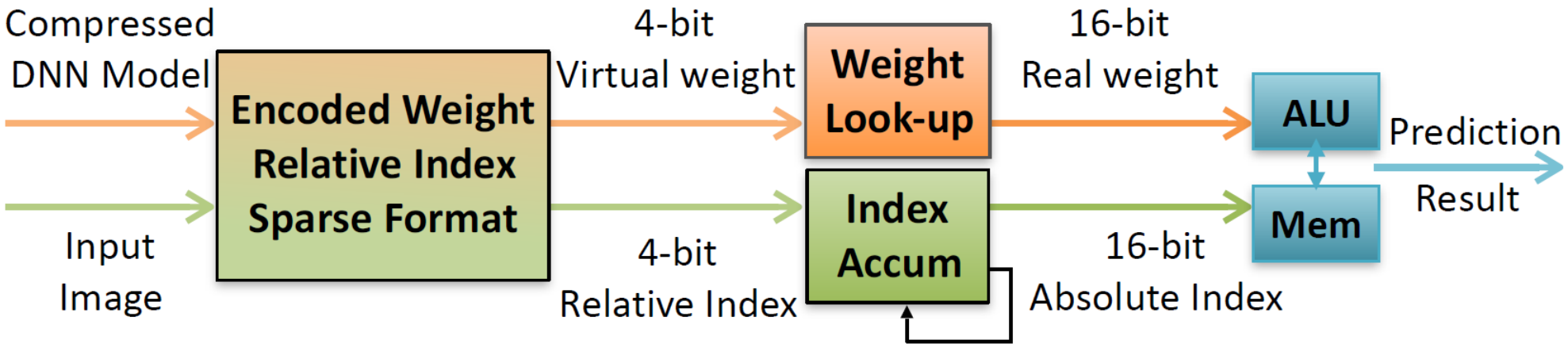
Retrain Code Book

- Train the code book using gradients calculated in back-propagation



Efficient Inference Engine (EIE)

- Hardware accelerator for deep compression



DNN Compression Computation

- Pruning makes matrix W sparse (density ranges from 4% to 25%)
- Weight sharing replaces each weight W_{ij} with a 4-bit index I_{ij} into a shared table S

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij} a_j\right) \longrightarrow b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}] a_j\right)$$

- X_i is the set of columns j for which $W_{ij} \neq 0$
- Y is the set of indices j for which $a_j \neq 0$
- S is the shared table of 16 possible weight values
- I_{ij} is the four-bit index to the shared weight that replaces W_{ij}

→ Perform MAC only for non-zero inputs and weights with indexing

Compressed Sparse Column (CSC) Format

- Efficient format for sparse matrix representation
- Each column of the matrix is represented by v, z and p
  - v: non-zero weights (4-bit index)
  - z: the number of zeros before the corresponding entry in v (4-bit)
  - p: vector pointer of each column (# of non-zero values in a column j = $p_{j+1} – p_j$)

CSC Example

0	0	0	2
4	0	0	7
0	0	1	0
0	0	3	9

Non-zero weight value

v	4	1	3	2	7	9

Relative row index

z	1	2	0	0	0	1

Column vector pointer

p	0	1	1	3	6

# of non-zero in column = $p_{j+1} - p_j$

CSC Example

0	0	0	2
4	0	0	7
0	0	1	0
0	0	3	9

Non-zero weight value

v

4	1	3	2	7	9

Relative row index

z

1	2	0	0	0	1

Column vector pointer

p

0	1	1	3	6

# of non-zero in column = $p_{j+1} - p_j$

CSC Example

0	0	0	2
4	0	0	7
0	0	1	0
0	0	3	9

Non-zero weight value

v

4	1	3	2	7	9

Relative row index

z

1	2	0	0	0	1

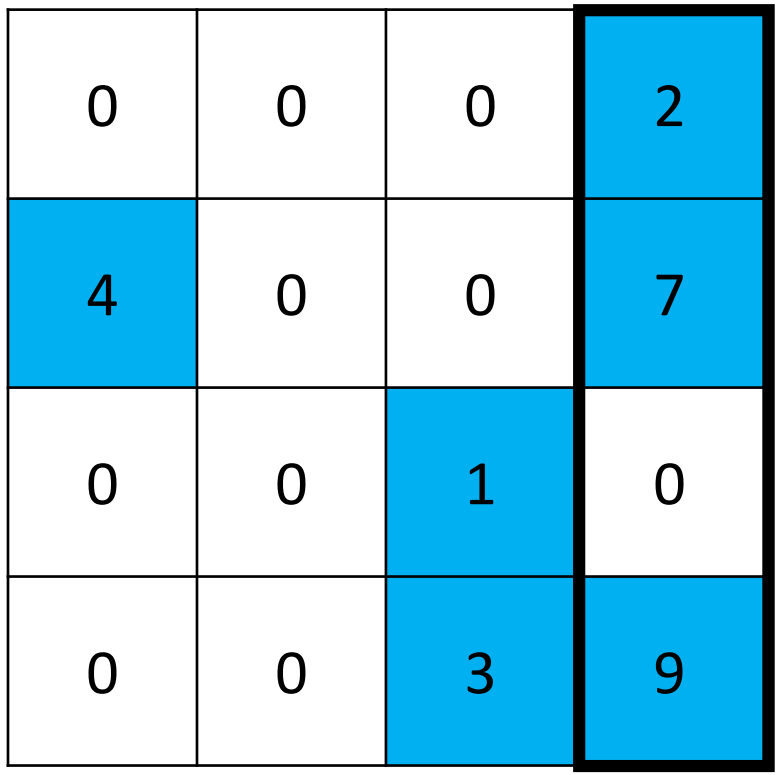
Column vector pointer

p

0	1	1	3	6

# of non-zero in column = $p_{j+1} - p_j$

CSC Example



Non-zero weight value

v

4	1	3	2	7	9

Relative row index

z

1	2	0	0	0	1

Column vector pointer

p

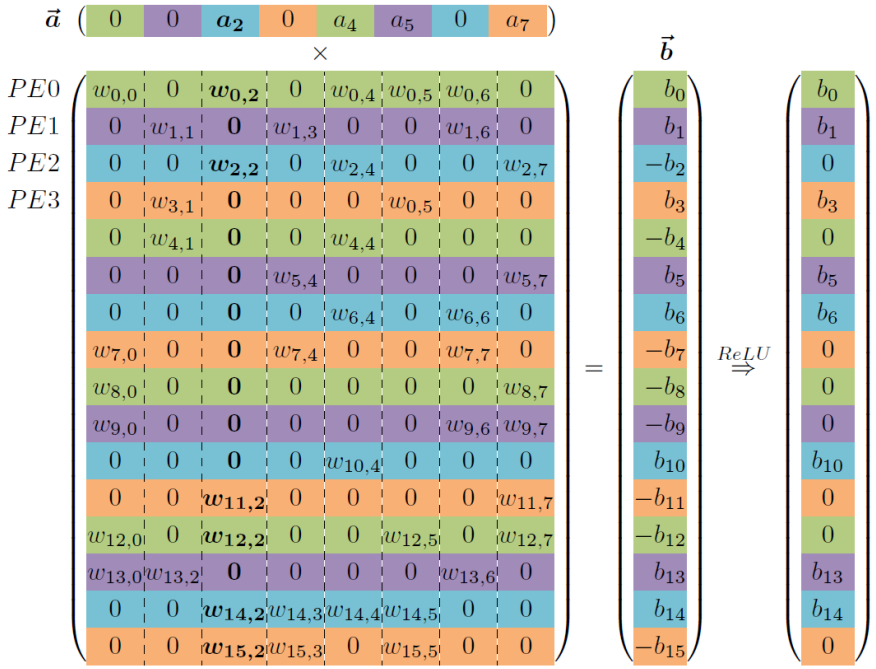
0	1	1	3	6

# of non-zero in column = $p_{j+1} - p_j$

Parallelizing Compressed DNN

- Distribute matrix rows over multiple processing elements (PEs)
  - Among *N* PEs, PE_k holds all rows W_i, output activations b_i, and input activations a_i for which i (mod *N*) = k
  - The columns of W_i are CSC format about each subset of the columns

16 x 8 matrix to 4 PEs

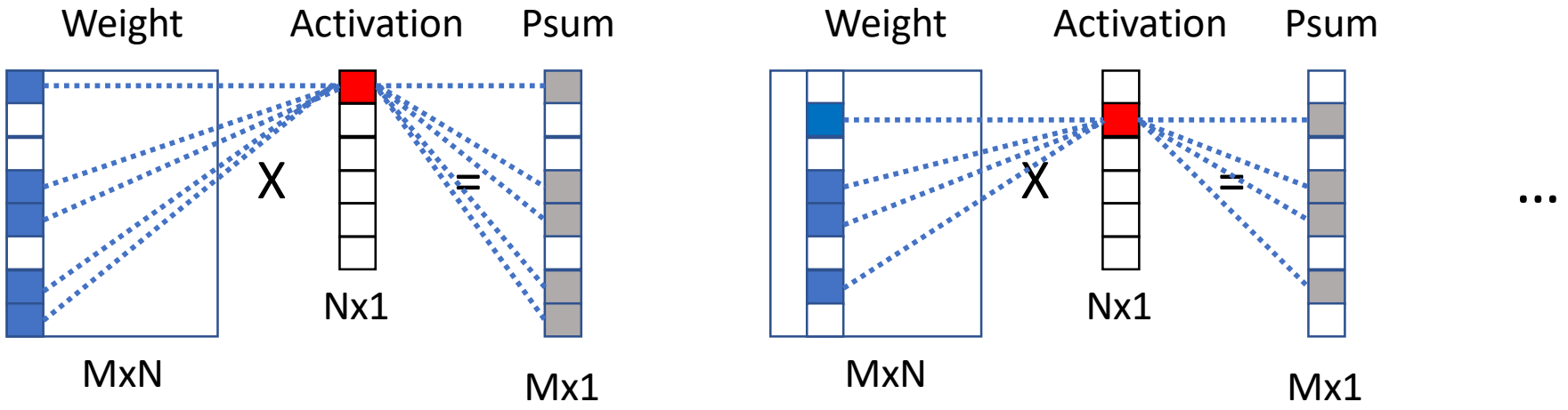


PE_0 in CSC format

Virtual Weight	$W_{0,0}$	$W_{8,0}$	$W_{12,0}$	$W_{4,1}$	$W_{0,2}$	$W_{12,2}$	$W_{0,4}$	$W_{4,4}$	$W_{0,5}$	$W_{12,5}$	$W_{0,6}$	$W_{8,7}$	$W_{12,7}$
Relative Row Index	0	1	0	1	0	2	0	0	0	2	0	2	0
Column Pointer	0	3	4	6	6	8	10	11	13				

Parallelizing Compressed DNN

- Parallelize matrix-vector multiplication by interleaving the rows of the matrix W over PEs
  1. Scan the input vector a to find its non-zero value a_j and broadcast a_j along with its index j to all PEs
  2. Each PE multiplies a_j by non-zero elements in its portion of column W_j
  3. Accumulate the partial sums from all PEs

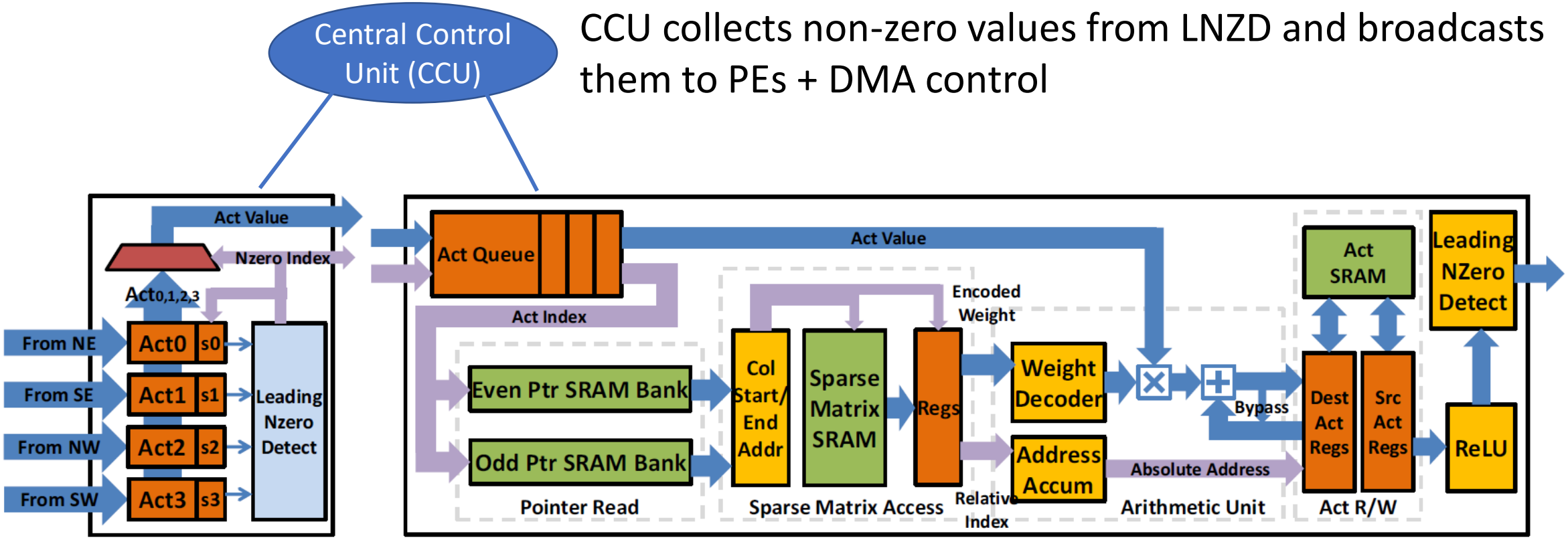


1st input multiplies with 1st column

2nd input multiplies with 2nd column

Load imbalance problem due to different number of non-zeros of each PE

Overall Architecture

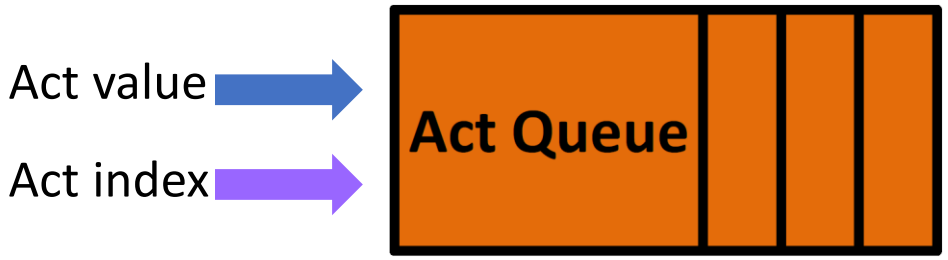


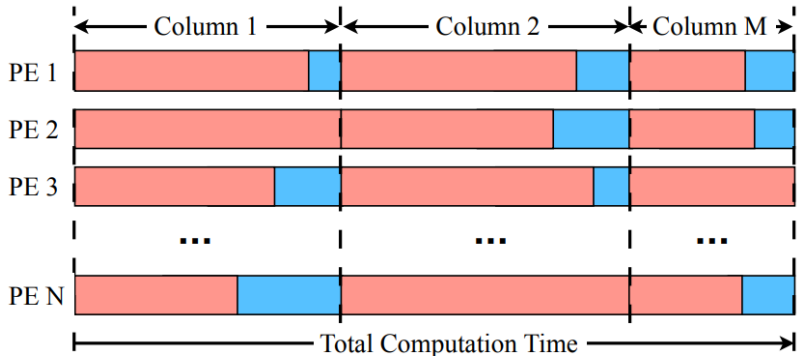
Leading Non-zero Detection Node (LNDZ)

Processing Element (PE)

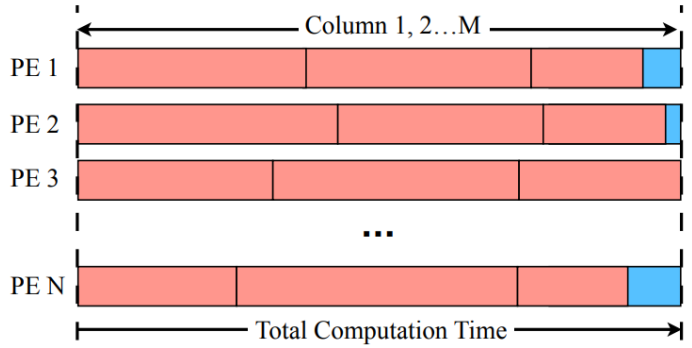
Activation Queue and Load Balancing

- CCU broadcasts non-zero elements of input activation vector a_j and their index j to activation queue
- Activation queue allows each PE to build up a backlog of work to spread out load imbalance





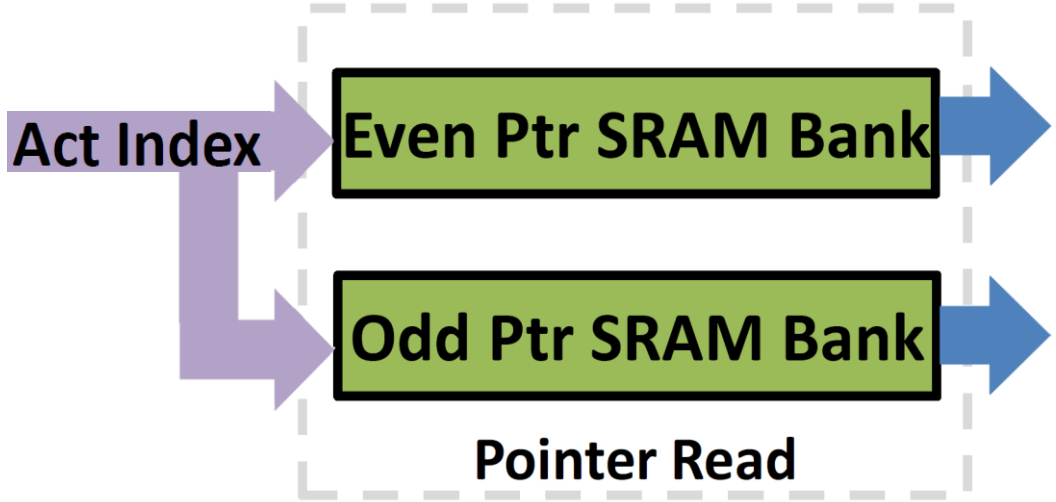
Without Activation Queue



With Activation Queue

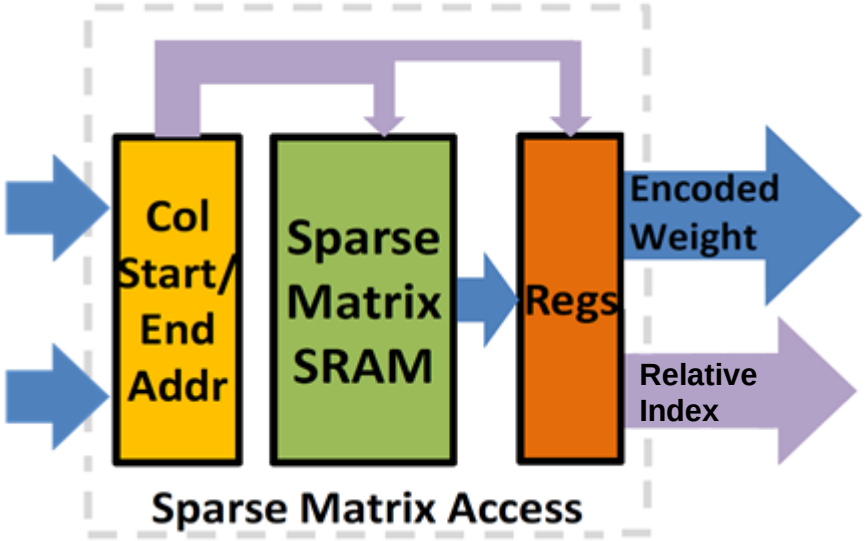
Pointer Read Unit

- Look up the start and end pointer p_j and p_{j+1} for further search of v and z
- Dual-bank design to read both pointers in one cycle (p_j and p_{j+1} will always be in different banks)



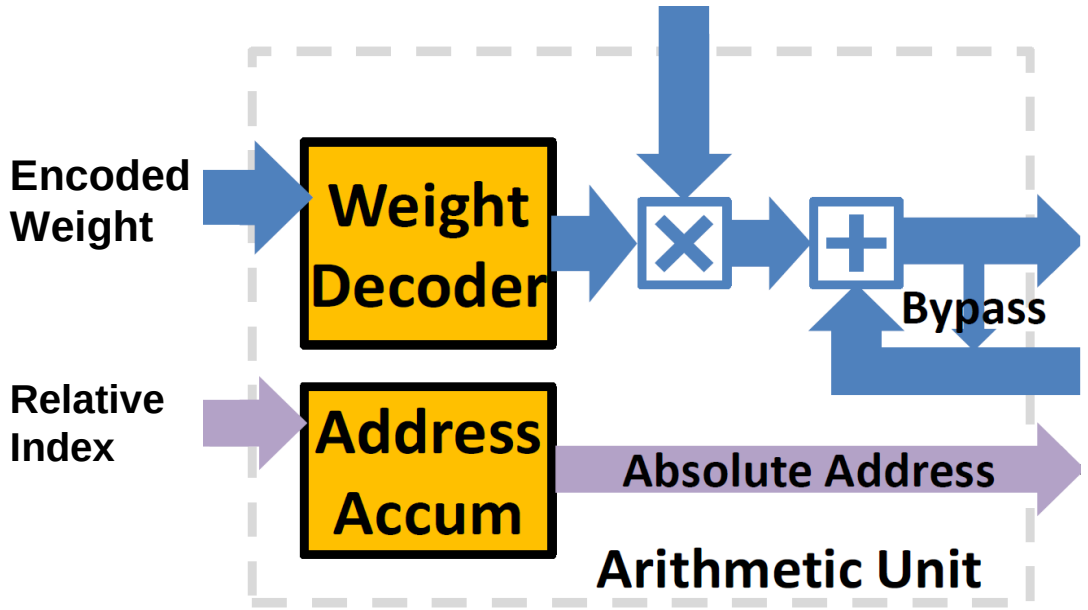
Sparse Matrix Read Unit

- Using the start and end pointer (p_j and p_{j+1}), read non-zero elements from the 64-bit wide sparse-matrix SRAM
- Each (v, z) entry contains one 4-bit element v and one 4-bit element z
  → One SRAM read contains 8 (v, z) entries for efficiency



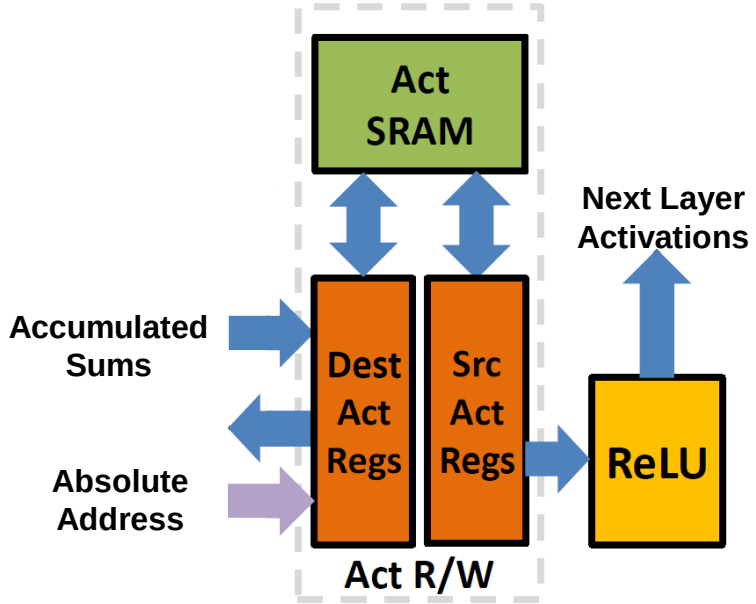
Arithmetic Unit

- Arithmetic unit receives a (v, z) entry from the sparse matrix read unit and performs multiply-accumulate operation
- 4-bit encoded index v is decoded to 16-bit fixed-point number via a codebook lookup
- A bypass path is provided to route the output of the adder to its input if the same accumulator is selected on two adjacent cycles

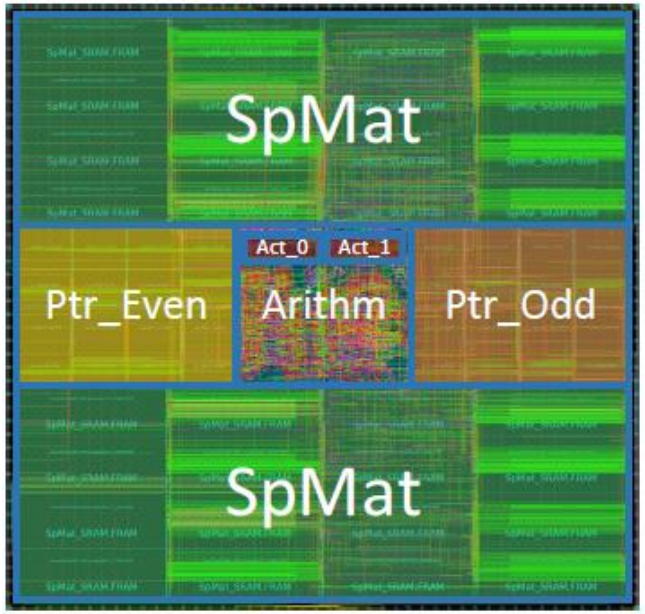


Activation Read/Write

- Contains two activation register files that accommodate the source and destination activation values during a single FC layer computation
- The source and destination register files switches their role for next layer → no additional memory movement needed
- 64 16-bit activation per register file → 4K activations across 64 PEs



Implementation Results

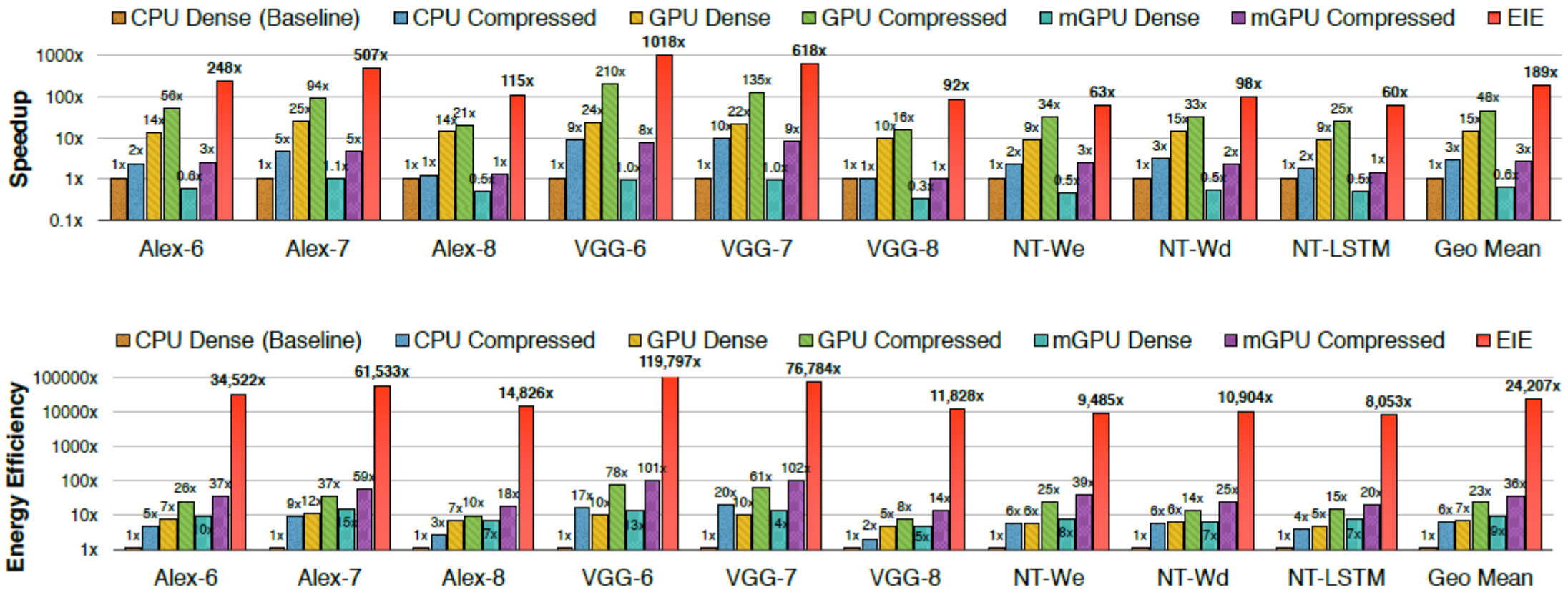


Layout of one PE in TSMC 45nm

	Power (mW)	(%)	Area (μm^2)	(%)
Total	9.157		638,024	
memory	5.416	(59.15%)	594,786	(93.22%)
clock network	1.874	(20.46%)	866	(0.14%)
register	1.026	(11.20%)	9,465	(1.48%)
combinational	0.841	(9.18%)	8,946	(1.40%)
filler cell			23,961	(3.76%)
Act_queue	0.112	(1.23%)	758	(0.12%)
PtrRead	1.807	(19.73%)	121,849	(19.10%)
SpmatRead	4.955	(54.11%)	469,412	(73.57%)
ArithmUnit	1.162	(12.68%)	3,110	(0.49%)
ActRW	1.122	(12.25%)	18,934	(2.97%)
filler cell			23,961	(3.76%)

Experimental Results

- CPU vs GPU vs Mobile GPU vs EIE
- Uncompressed vs Compressed



Comparison

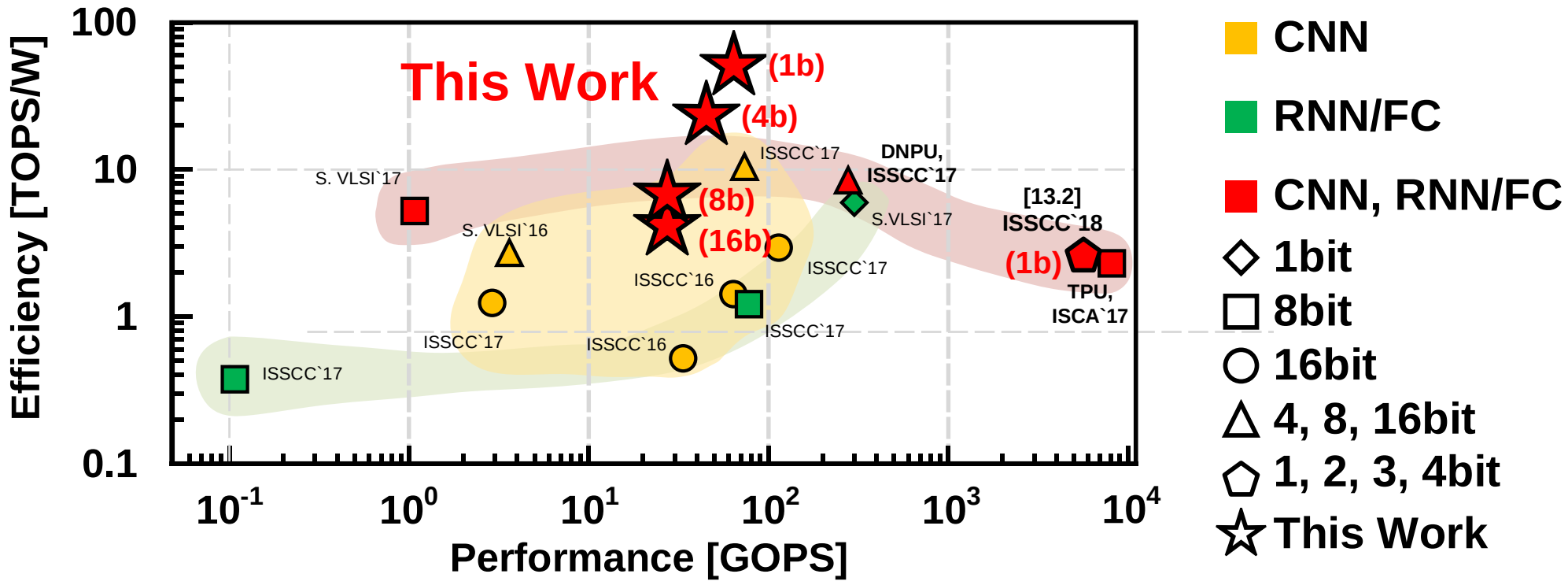
- MxV performance is evaluated on FC7 layer of AlexNet
- EIE architecture is scalable from one PE to over 256 PEs

COMPARISON WITH EXISTING HARDWARE PLATFORMS FOR DNNS.

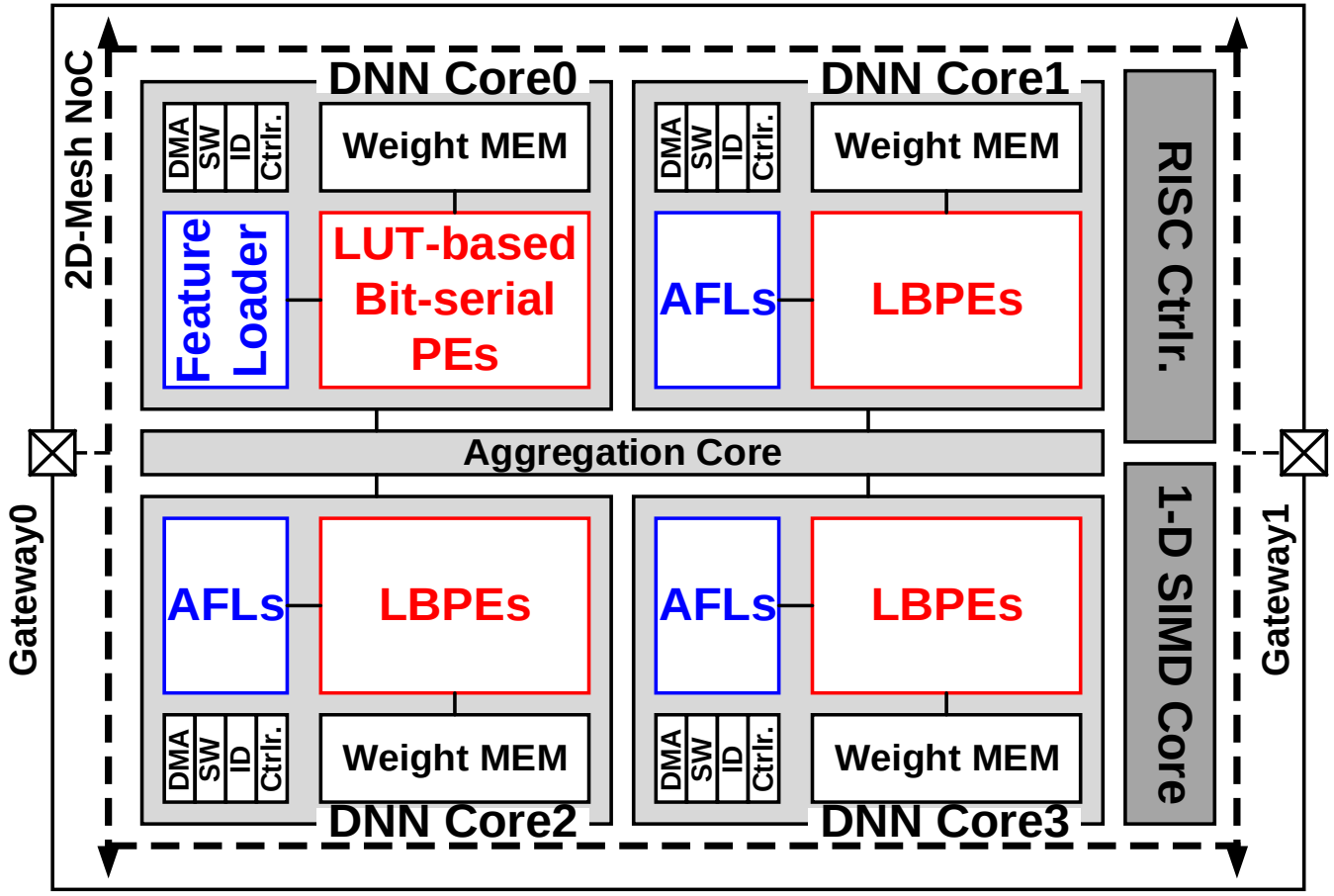
Platform	Core-i7 5930K	GeForce Titan X	Tegra K1	A-Eye [14]	Da-DianNao [11]	True-North [40]	**EIE (ours, 64PE)**	**EIE (28nm, 256PE[3])**
Year	2014	2015	2014	2015	2014	2014	2016	2016
Platform Type	CPU	GPU	mGPU	FPGA	ASIC	ASIC	ASIC	ASIC
Technology	22nm	28nm	28nm	28nm	28nm	28nm	45nm	28nm
Clock (*MHz*)	3500	1075	852	150	606	Async	800	1200
Memory type	DRAM	DRAM	DRAM	DRAM	eDRAM	SRAM	SRAM	SRAM
Max DNN model size (*#Params*)	<16G	<3G	<500M	<500M	18M	256M	84M	336M [4]
Quantization Stategy	32-bit float	32-bit float	32-bit float	16-bit fixed	16-bit fixed	1-bit fixed	4-bit fixed	4-bit fixed
Area (mm^2)	356	601	-	-	67.7	430	40.8	63.8
Power (*W*)	73	159	5.1	9.63	15.97	0.18	0.59	2.36
M×V Throughput (*Frames/s*)	162	4,115	173	33	147,938	1,989	81,967	426,230
Area Efficiency (*Frames/s/mm²*)	0.46	6.85	-	-	2,185	4.63	2,009	6,681
Energy Efficiency (*Frames/J*)	2.22	25.9	33.9	3.43	9,263	10,839	138,927	180,606

UNPU Motivation

- DNN ASIC trend (2018)
  - Support CNN, RNN, and FC DNN
  - Support narrow & multi-bit precision



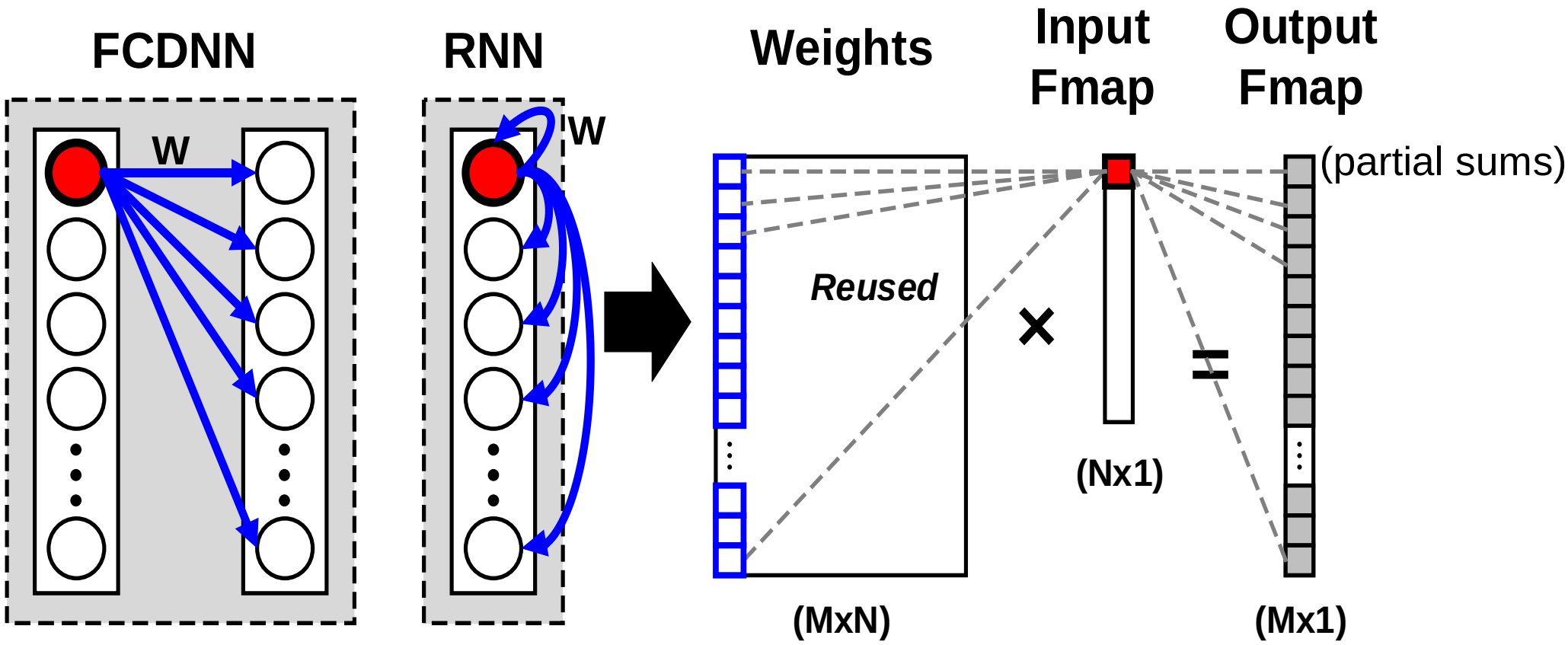
UNPU Architecture



- Aligned Feature Map Loader (AFL) supports both DNN/RNN and CNN workloads runtime
- LUT-based Bit-serial PE (LBPE) enables fully-variable bit precision for weight data
- Weight memory
  - 48KB SRAM

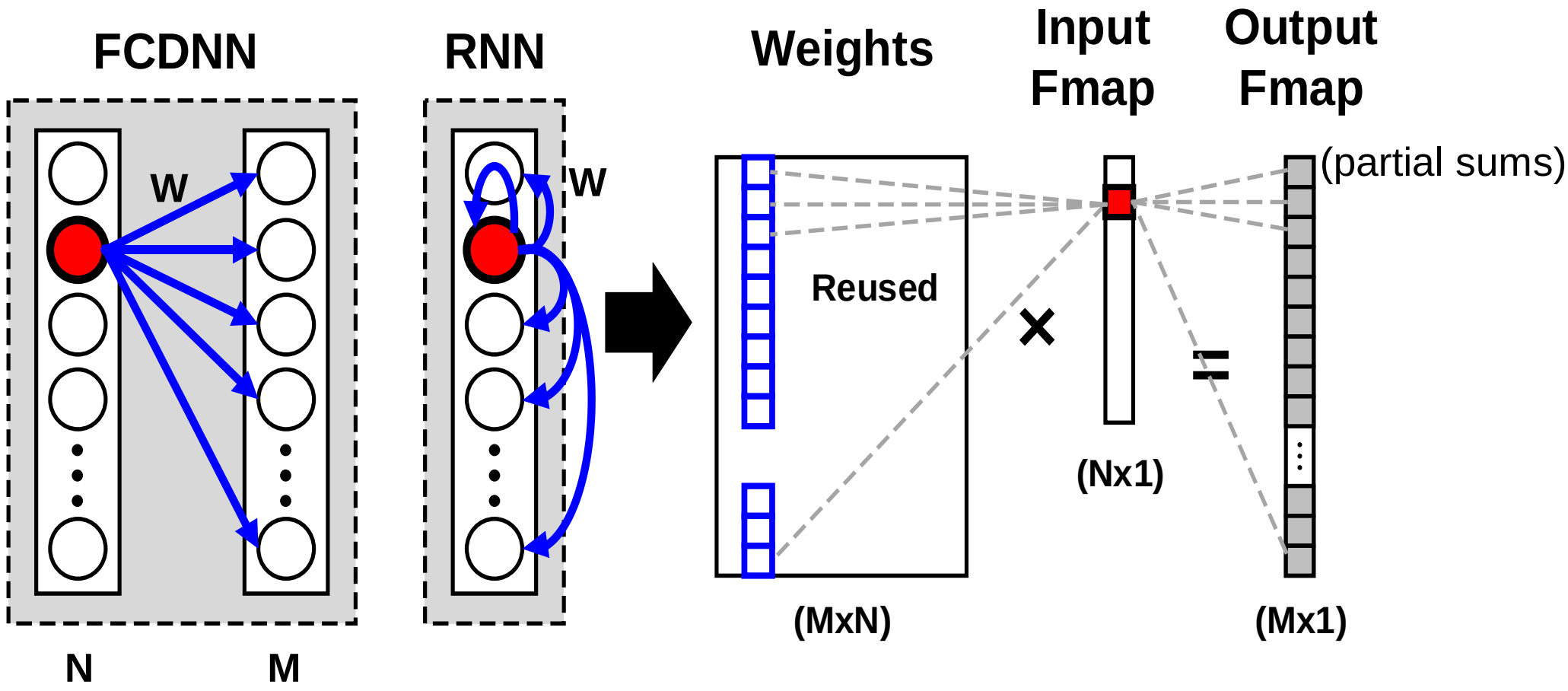
FC/RNN Operation

- Input feature map reuse



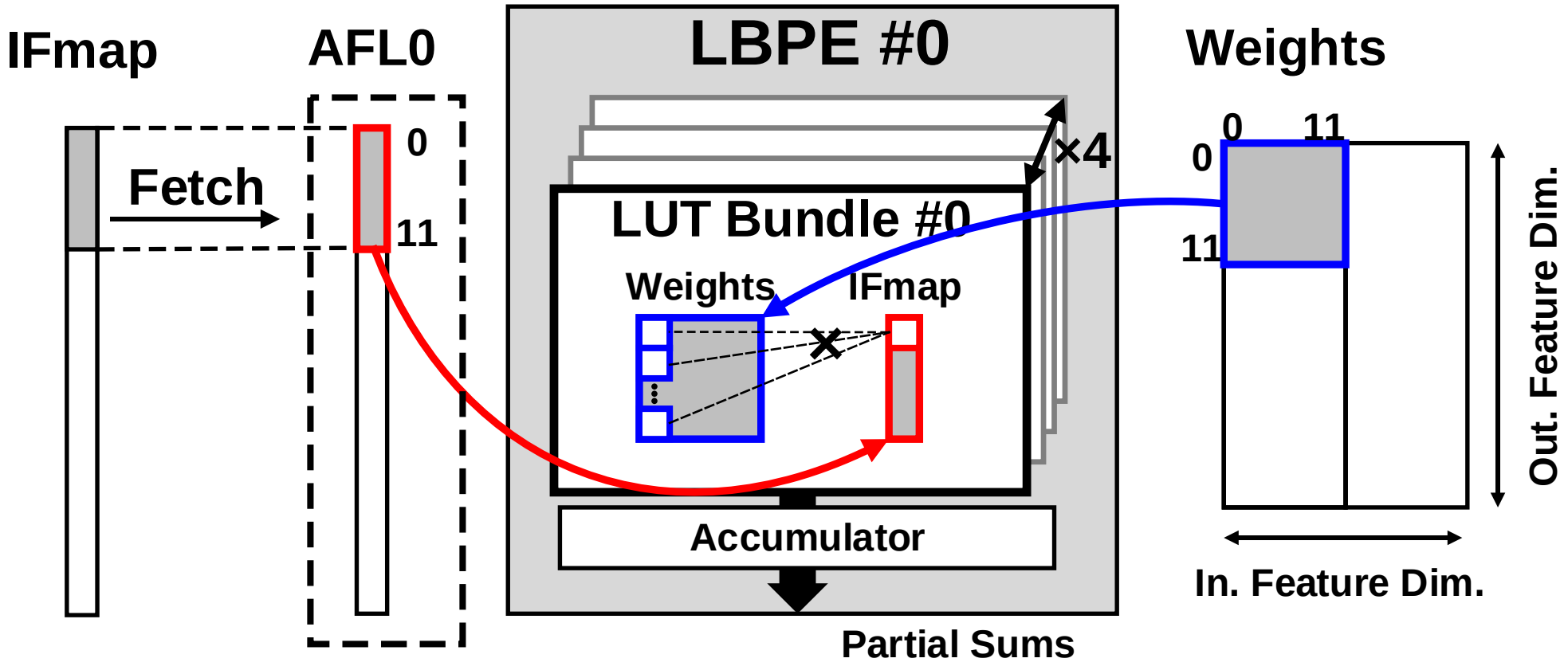
FC/RNN Operation

- Input feature map reuse



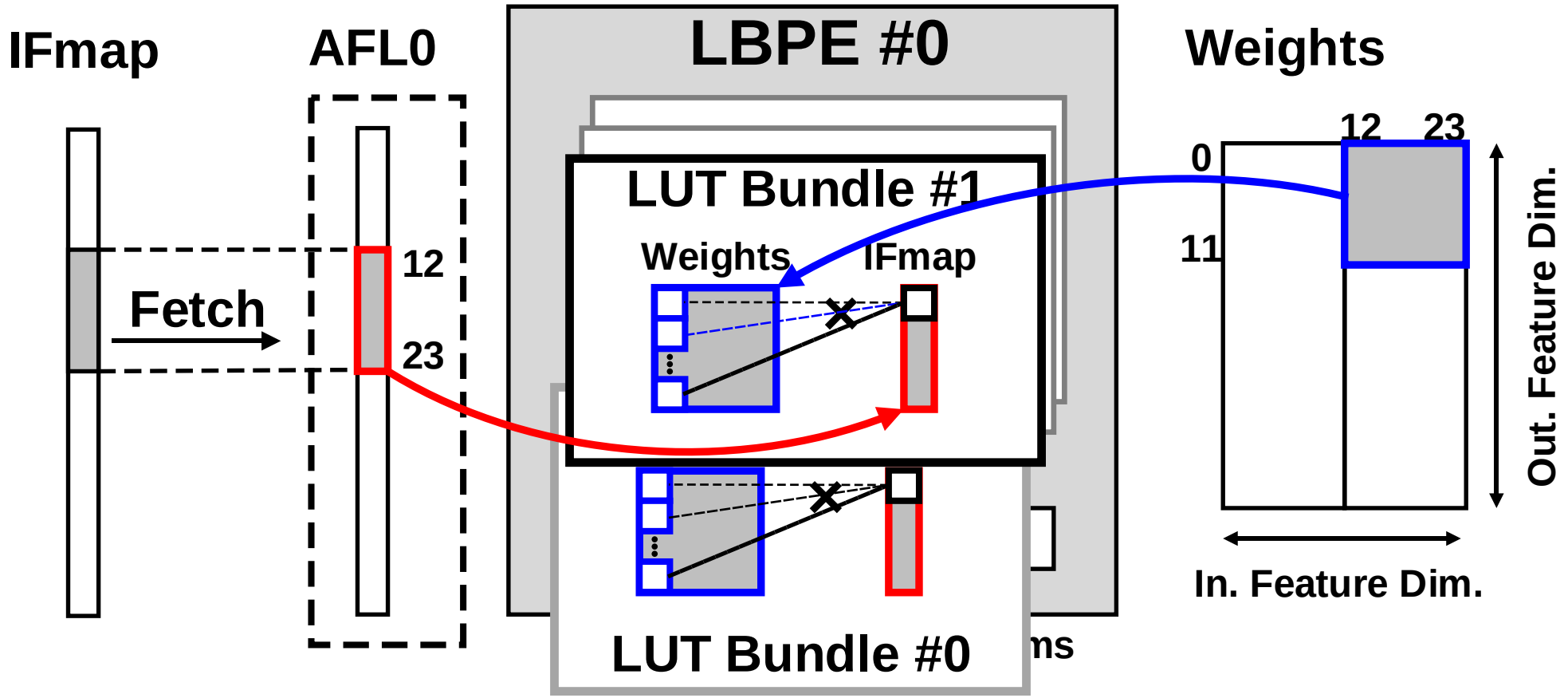
FC/RNN Mapping to Unified DNN Core

- Input feature map in LUT bundle is reused



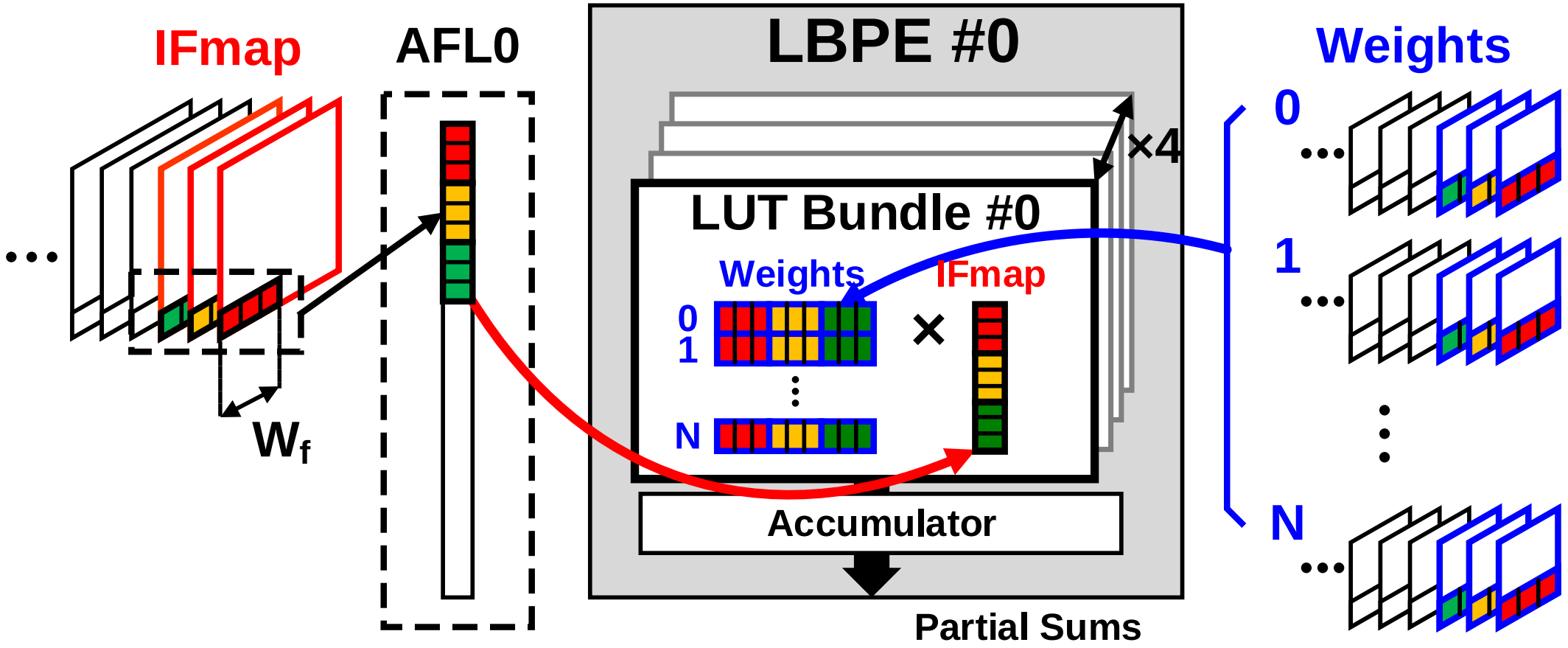
FC/RNN Mapping to Unified DNN Core

- Different input feature map location maps to other LUT bundles



CNN Mapping to Unified DNN Core

- Convert 2-D input feature map into a vector like im2col



CNN Mapping to Unified DNN Core

- Different input feature map location maps to other LUT bundles

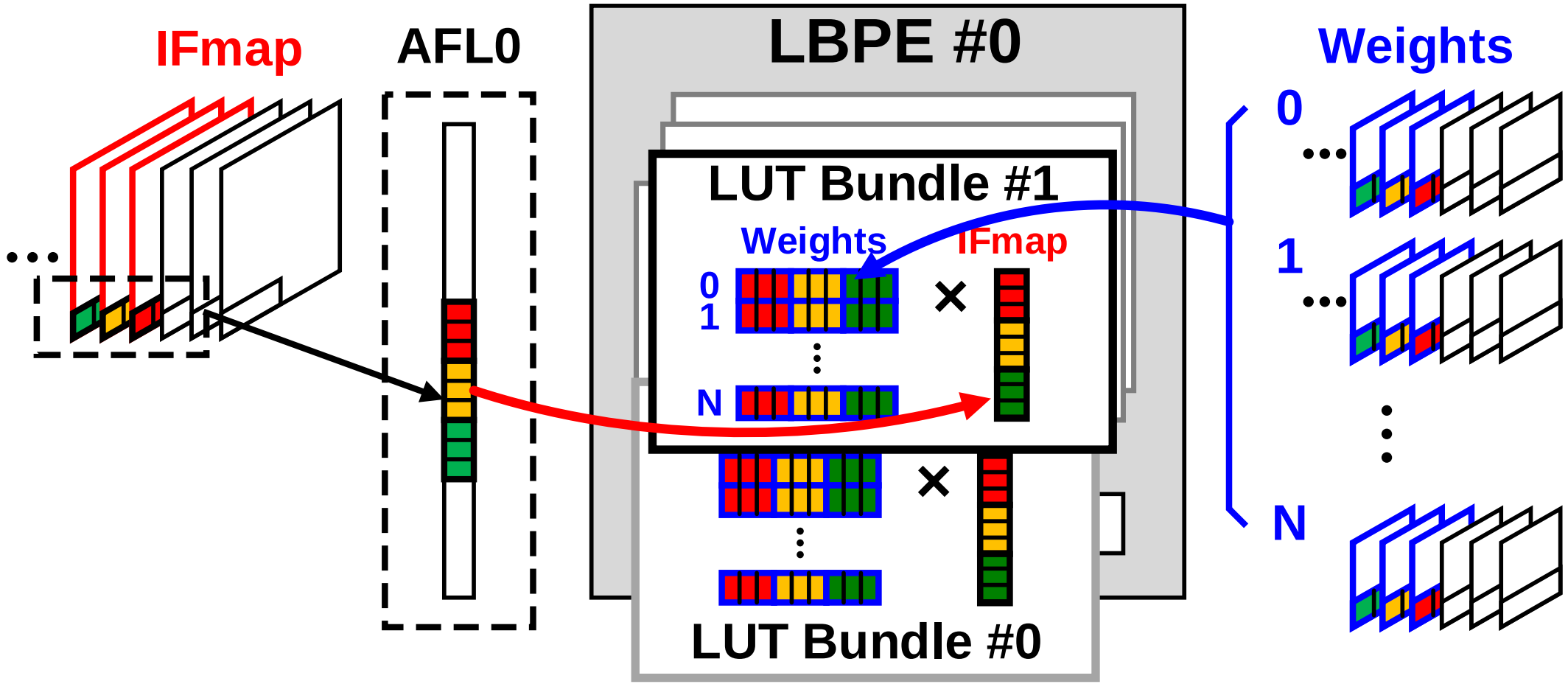


Table-based Bit-Serial Processing

- Accumulate partial products from LSB to MSB bit-by-bit each cycle

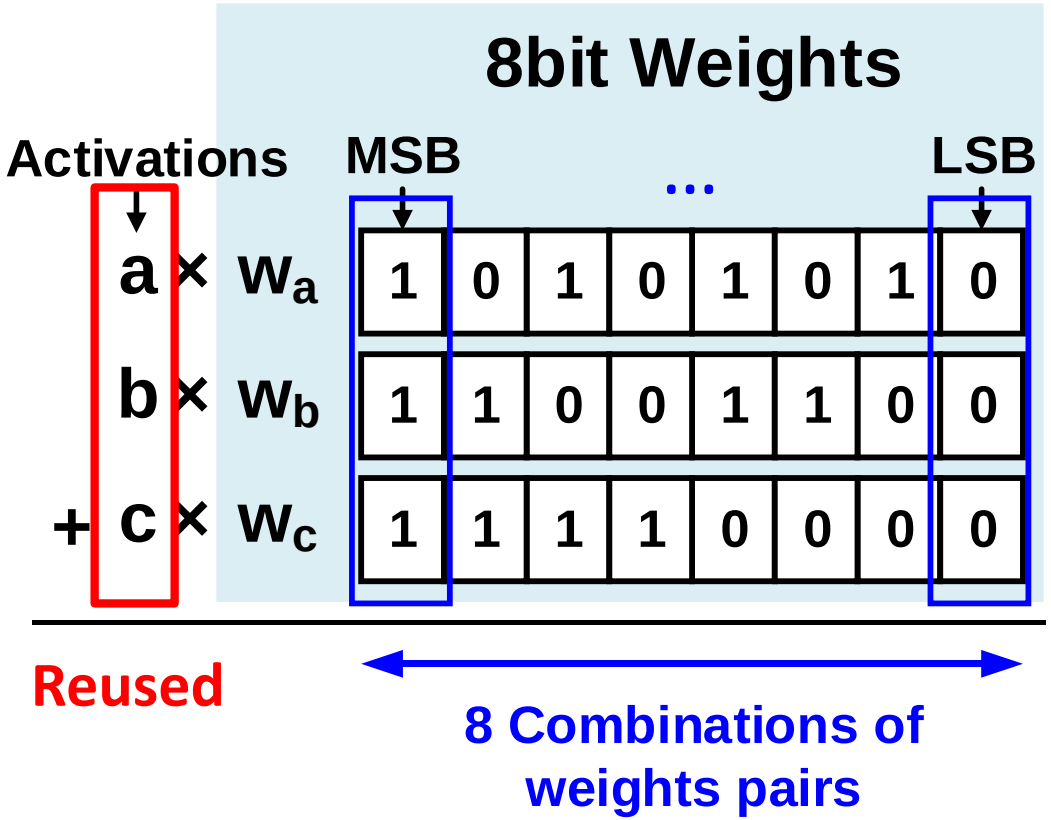
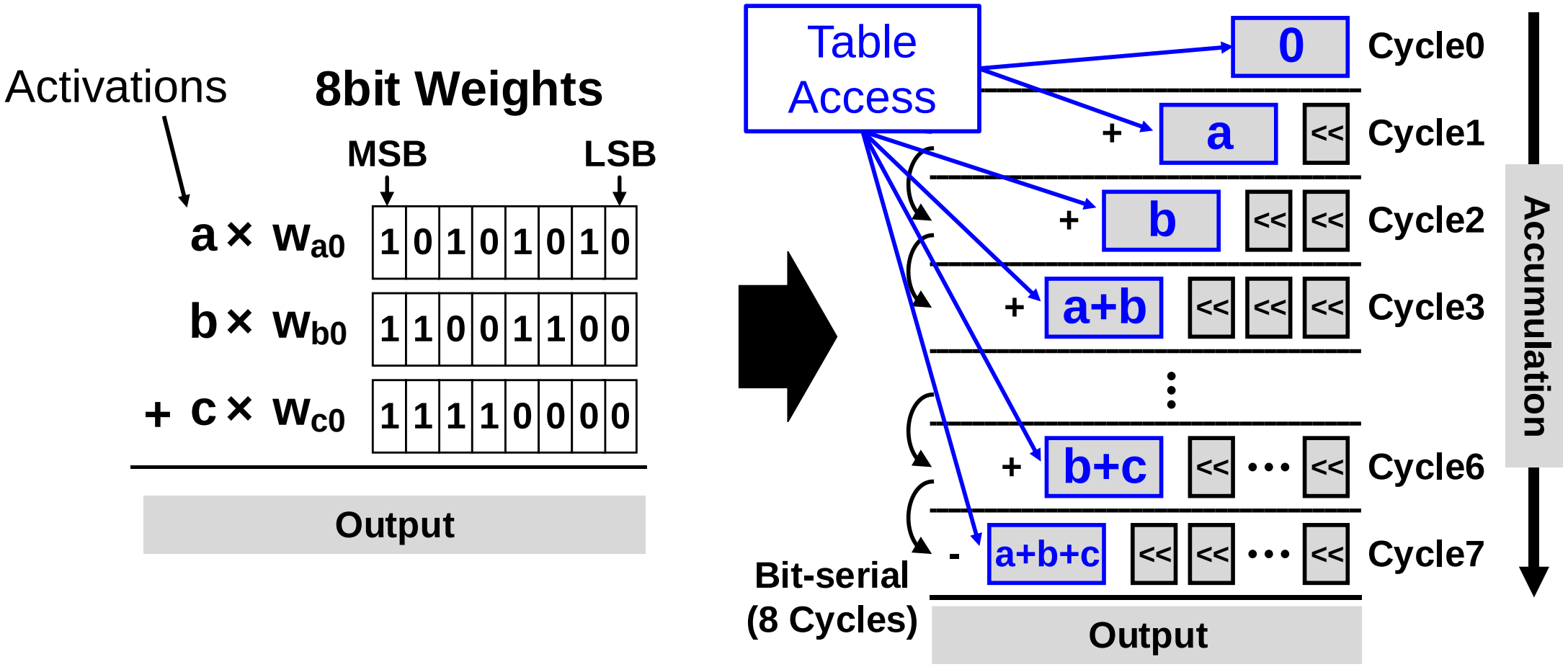


Table for Partial Product

Index $[W_aW_bW_c]$	Value
$000_{(2)}$	0
$001_{(2)}$	c
$010_{(2)}$	b
$011_{(2)}$	b + c
$100_{(2)}$	a
$101_{(2)}$	a + c
$110_{(2)}$	a + b
$111_{(2)}$	a + b + c

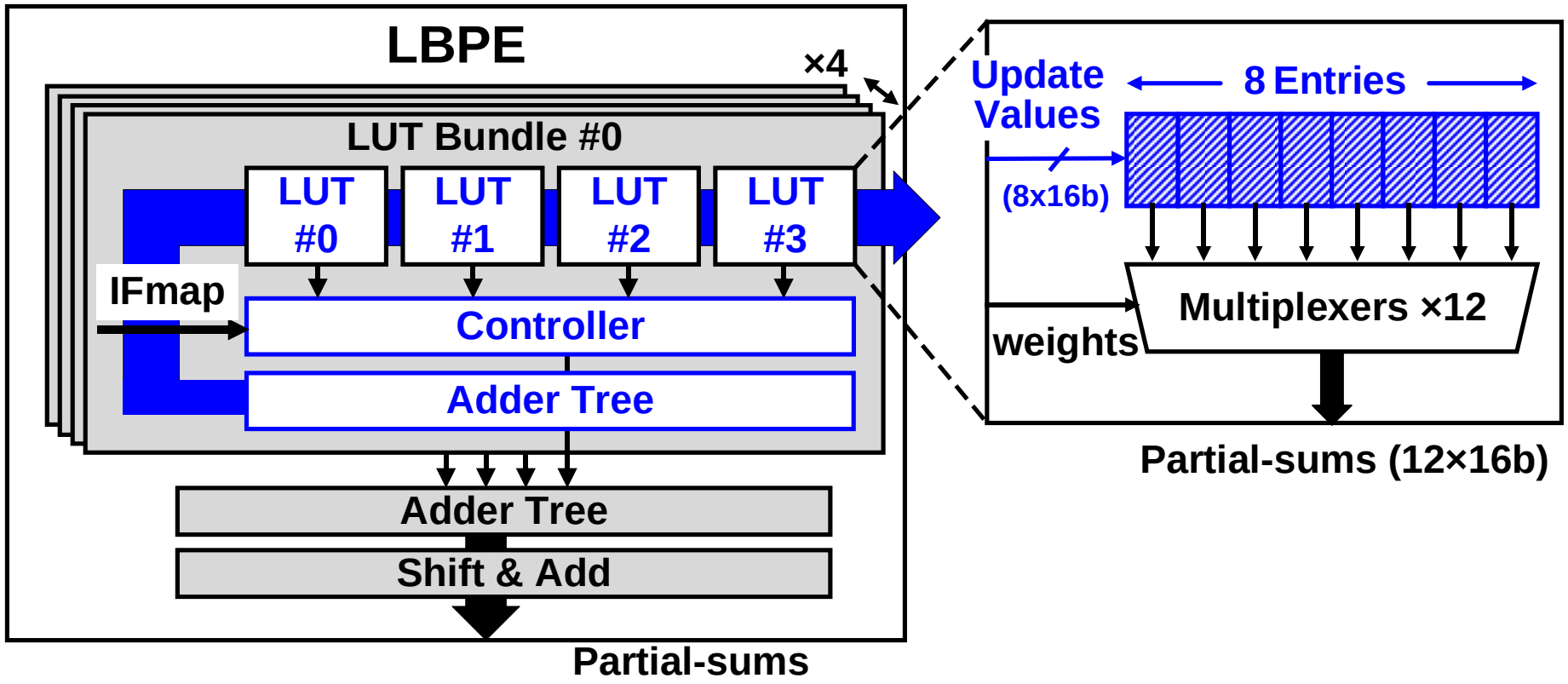
LBPE Operation Example

- Bit-serial MAC by table look-up and shift-accumulate



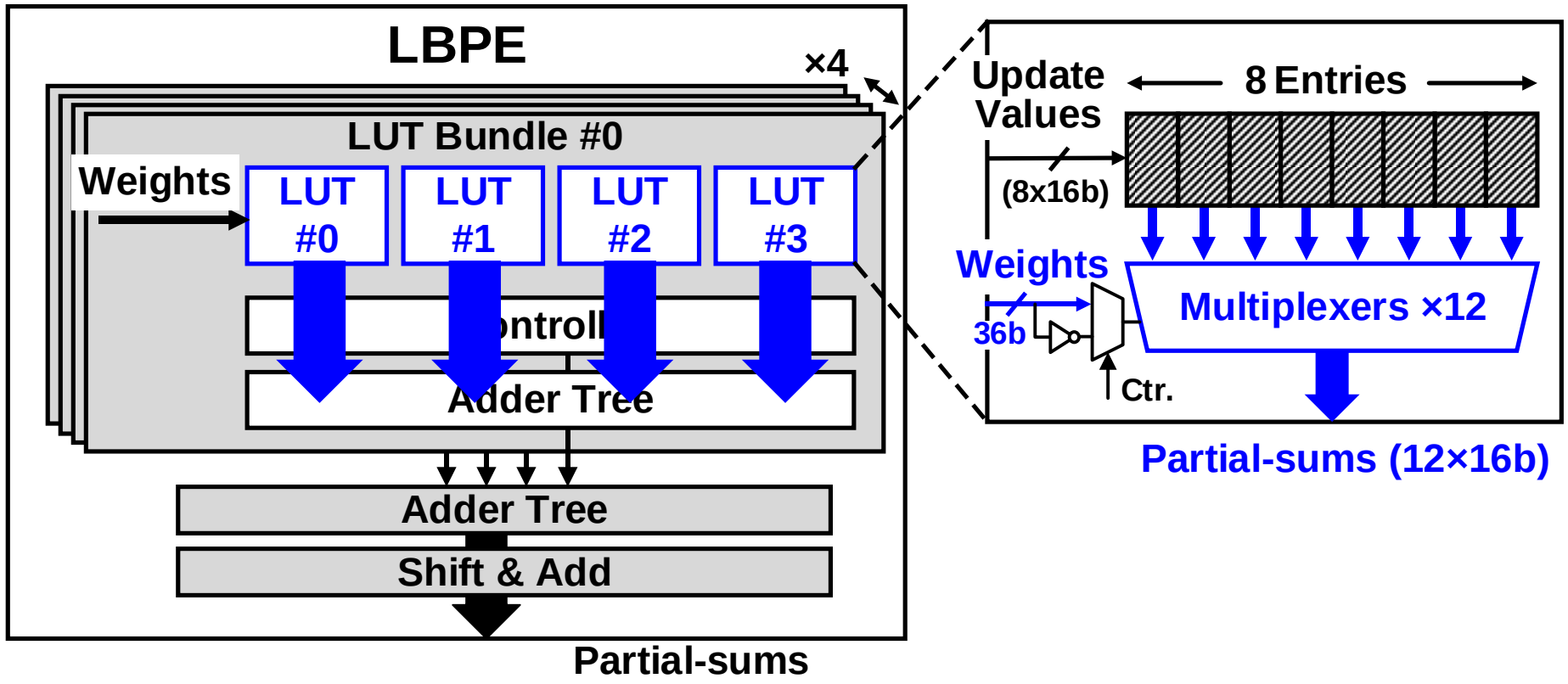
LBPE Architecture

- Prep phase: update LUTs with pre-calculated 8 psums for 3-input MAC
  - 8 cases: 0, a, b, c, a+b, a+c, b+c, a+b+c



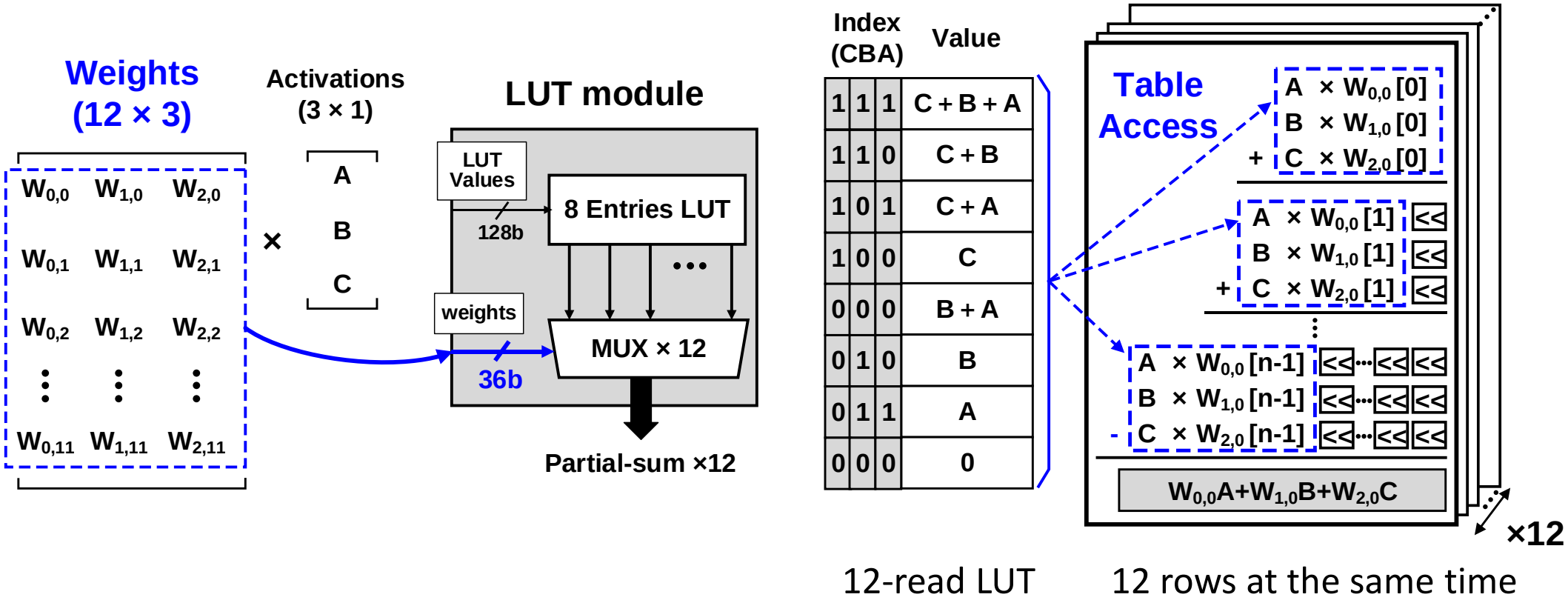
LBPE Architecture

- Compute phase: table look-up and accumulate partial sums
- Can handle up to 12 weight data (12 x {1b, 1b, 1b})



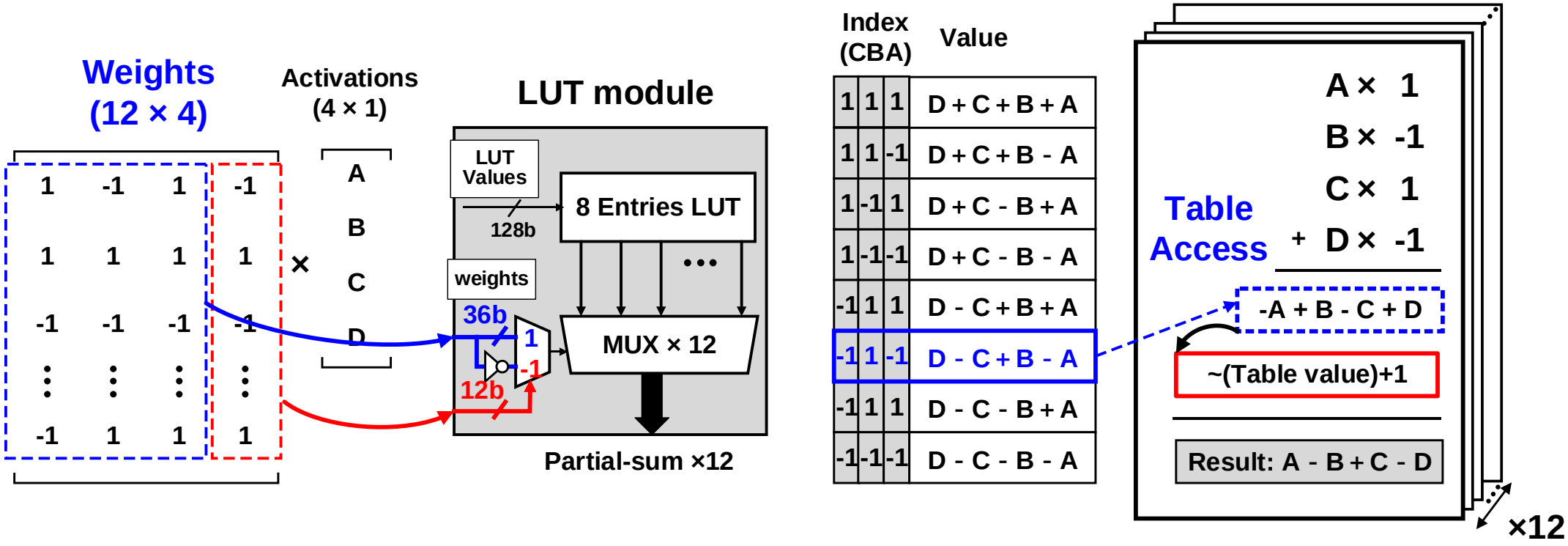
LBPE Mode

- Multi-bit weights (2, 3, …, 16 bit)
  - One LUT takes N cycles for 12 N-bit 3-input MACs
  - 36 MACs / N per cycle for N-bit weight



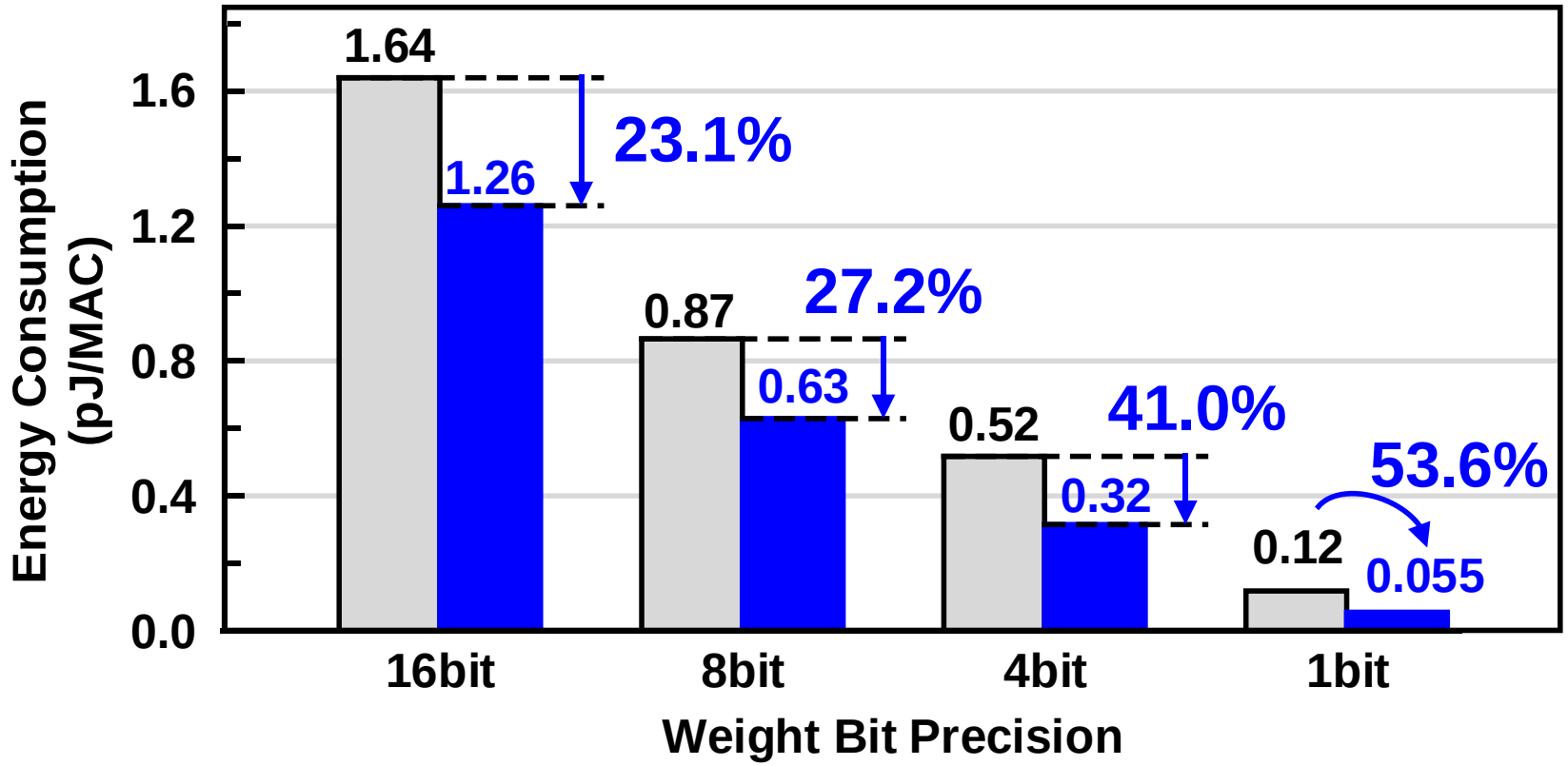
LBPE Mode

- Binary weights (1-bit weight)
  - Do additional operation (2's complement + 1) based on D value
  - One LUT can perform 12 4-input MACs / cycle



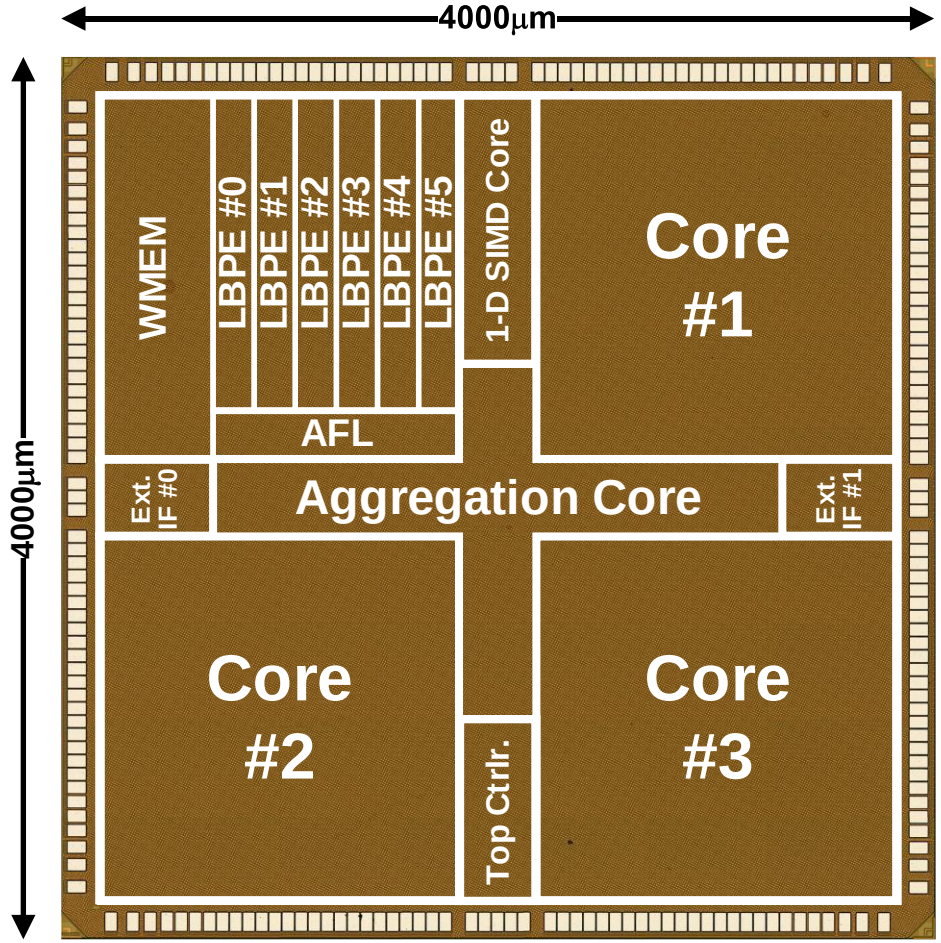
Experimental Results

- Fixed-point MAC vs LBPE MAC (1 LBPE = 16 LUTs)



Samsung 65nm Logic Process, Synopsys PrimeTime, 200MHz, 1.2V,
Multi-bit Mode: 576MACs/(N Cycles) (N=4,8,16), Binary Mode: 768MACs/cycle (N=1)

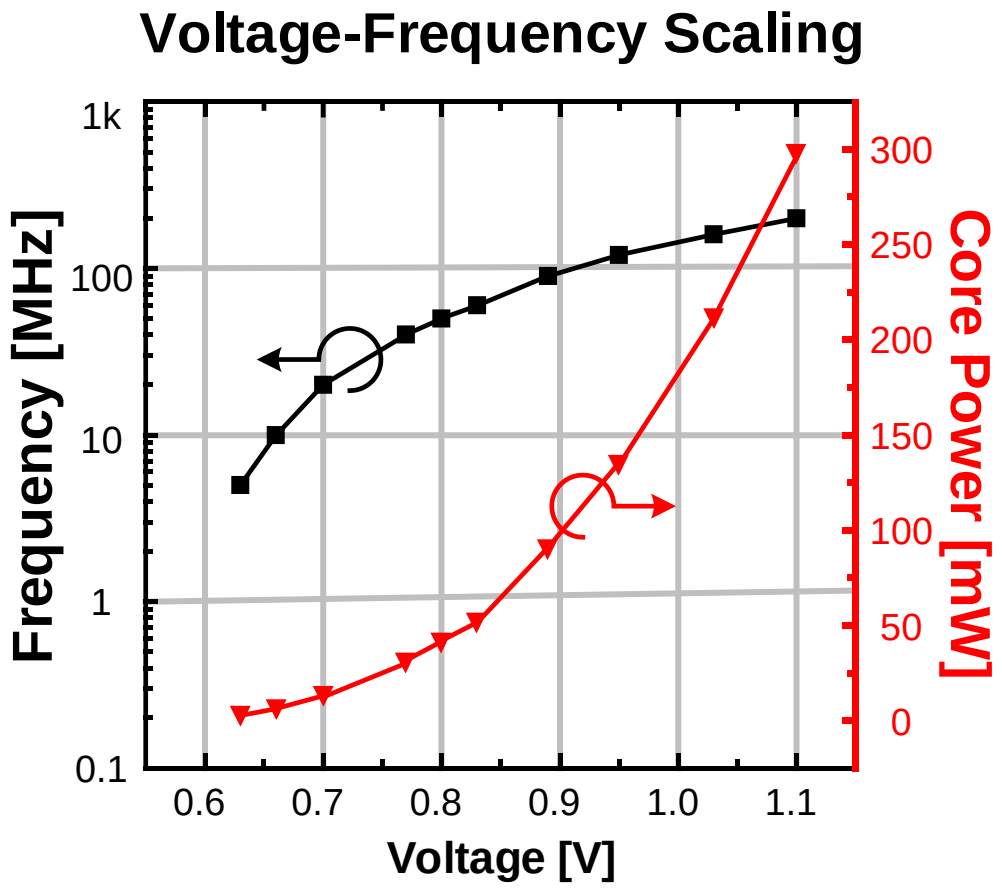
Chip Photo

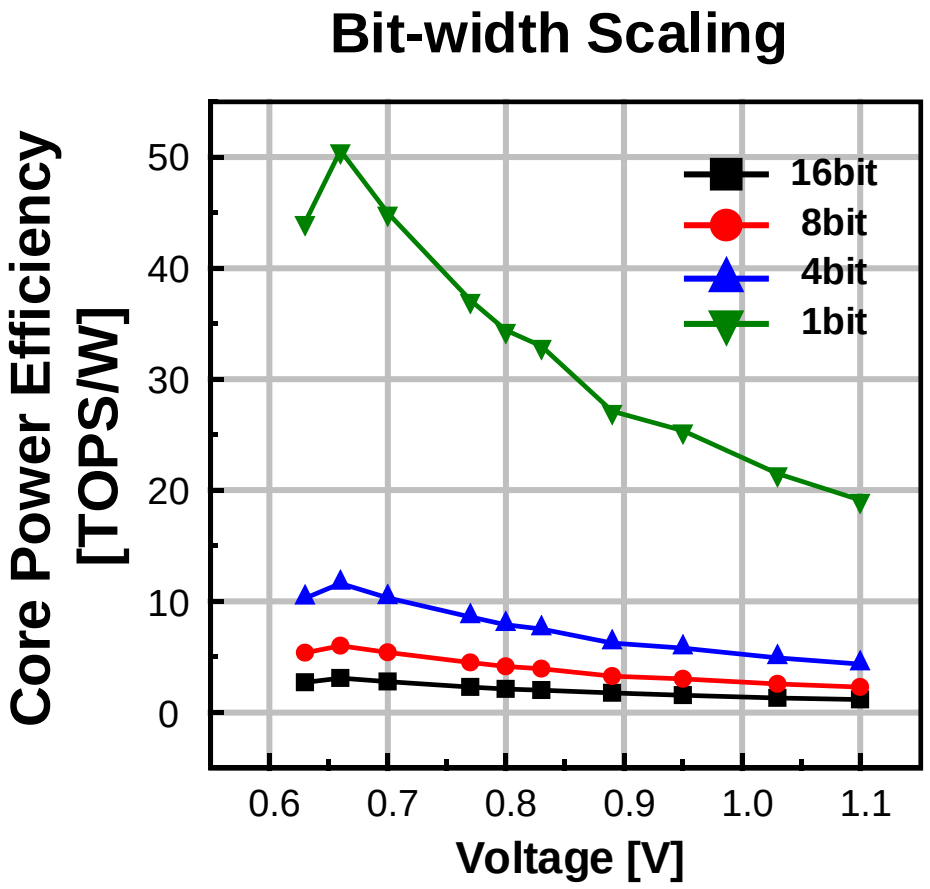


	Specification
Technology	65nm Logic CMOS
Area [mm^2]	16
SRAM [KB]	256
Supply voltage [V]	1.1
Frequency [MHz]	200
Weight Precision [Bit]	1 – 16
Power [mW]	297 @ 200MHz, 1.1V
	3.2 @ 5MHz, 0.63V
Power Efficiency [TOPS/W]	50.6 (1b Weight)
	3.08 (16b Weight)

Voltage-Frequency & Bit-Width Scaling

- Measured on 5x5 convolution operation





Comparison

	DNPU ISSCC`17	S. Yin S.VLSI`17	QUEST ISSCC`18	This Work
Purpose	CNN, RNN	CNN, RNN	CNN, RNN	CNN, RNN
Technology	65nm LP	65nm	40nm	65nm LP
Area [mm^2]	16	19.4	121.6	16
PE Bit-precision [bit]	**4, 8, 16**	4, 8, 16,	1 – 4	**1 – 16**
Performance [GOPS]	1,200 (4b)	410 (4b)	7,490 (1b) 1,960 (4b)	**7,372 (1b)** **1,382 (4b)**
Power-Efficiency [TOPS/W]	**8.1 (4b)**	5.1 (4b)	2.27 (1b) 0.59 (4b)	**50.6 (1b)** **11.6 (4b)**
Area Eff. [GOPS/mm^2]	**75 (4b)**	21.1 (4b)	**61.6(1b)** 16.2 (4b)	**461 (1b)** **86.4 (4b)**
Power [mW]	**35 – 279**	4 – 447	3300	**3.2 - 297**

Discussion

- How much do you think you can reduce the model size?
  - Trade-off b/w application accuracy and model size, computation efficiency
  - Do you need an automation process from original model to compressed model?
  - Do you have any other ideas to reduce the model size further except pruning, quantization, narrow-precision?
- Low-power hardware implementation
  - Like UNPU used pre-calculated LUT based MAC operation, do you have any ideas for low-power MAC implementation?
- Software-Hardware Co-Design
  - It is booming! Tolerating a bit of accuracy (or even without losing any), you can get lots of computational benefits
  - Discuss SW/HW co-design ideas for energy-efficient ML accelerators

ML Accelerators for Cloud Datacenters

- TPU (ISCA 2017)
  - Norman P. Jouppi, et al. “In-Datacenter Performance Analysis of a Tensor Processing Unit,” ISCA 2017
  - C. Chao and B. Saeta, "Cloud TPU: Codesigning Architecture and Infrastructure," HotChips 2019 Tutorial
  - David Patterson, "Evaluation of the Tensor Processing Unit: A Deep Neural Network Accelerator for the Datacenter", NAE Regional Meeting, April 2017
- BrainWave (ISCA 2018)
  - J. Fowers, et al. “A Configurable Cloud-Scale DNN Processor for Real-Time AI,” ISCA 2018
  - E. Chung, et al. “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave,” IEEE Micro 2018
- GPU architecture
  - K. Fatahalian, “Graphics and Imaging Architectures,” CMU class fall 2011
  - NVIDIA whitepaper, “NVIDIA’s Next Generation CUDA Compute Architecture: Fermi,” 2009
  - J. Choquette, “Volta: Programmability and Performance,” Hot Chips 2017

Motivation

- Three kinds of neural networks are popular (2016-2017)
  - Multi-Layer Perceptron (MLP)
  - Convolutional Neural Networks (CNN)
  - Recurrent Neural Networks (RNN)

- Six neural networks applications that represent 95% of inference workload in Google's datacenters (July 2016)

Name	*LOC*	*Layers*					*Nonlinear function*	*Weights*	*TPU Ops / Weight Byte*	*TPU Batch Size*	*% of Deployed TPUs in July 2016*
		FC	*Conv*	*Vector*	*Pool*	*Total*					
MLP0	100	5				5	ReLU	20M	200	200	61%
MLP1	1000	4				4	ReLU	5M	168	168	
LSTM0	1000	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1500	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1000		16			16	ReLU	8M	2888	8	5%
CNN1	1000	4	72		13	89	ReLU	100M	1750	32	

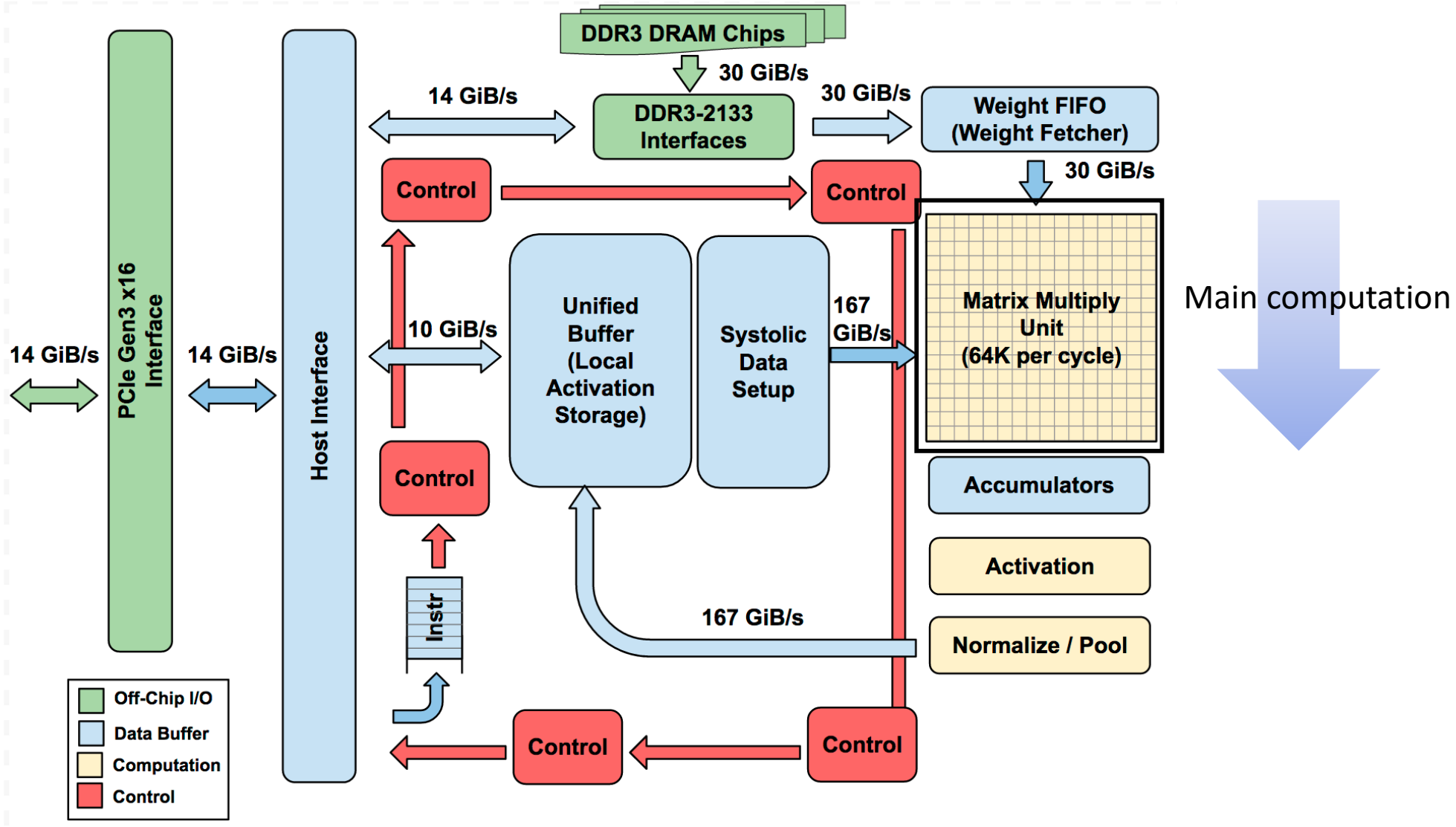
Motivation

- Views on special hardware in datacenters
  - 2006: no need to have special hardware for a few certain applications
  - 2013: people started to search by voice for 3 minutes a day using speech recognition DNNs → This will double Google datacenters' computation demands and it will be very expensive with conventional CPUs
  - Google started a high priority project to produce a custom ASIC for inference only
- Goal
  - To improve cost-performance by 10X over GPUs
  - To run whole inference models in the TPU to reduce interaction with the host CPU and to be flexible enough to match the DNN needs of 2015 and beyond
- Fast development
  - TPU was designed, built, and deployed in datacenters within 15 months

Key Design Concepts for TPU

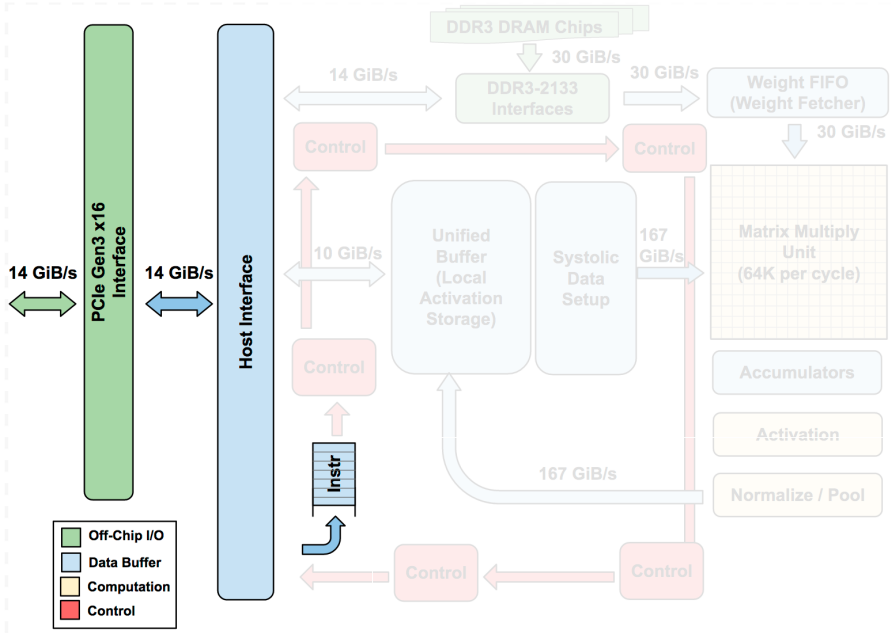
- Response time
  - Applications in Google datacenters are mostly user-facing, which leads to rigid response-time limits
- Batch size
  - To amortize access cost, same weights are reused across a batch of independent examples during inference or training
  - Large batch size improves throughput performance
- Quantization
  - 8-bit integer multiplication can be 6x less energy and 6x less area than IEEE 754 16-bit floating-point multiplication
  - 8-bit integer addition is 13x, 38x more efficient in energy and area than 16-bit

Overall Architecture



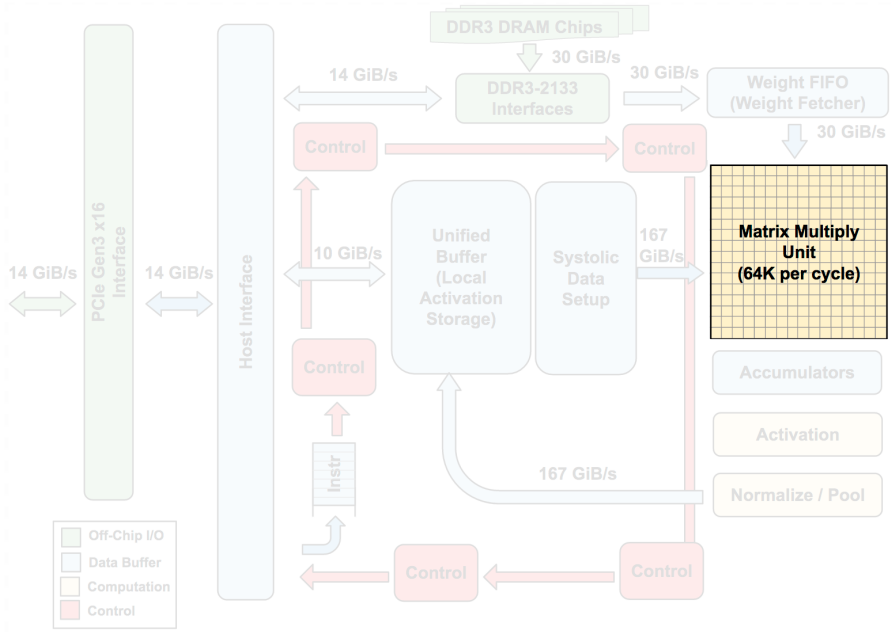
TPU Components

- Interface
  - TPU was designed to be a coprocessor on the PCIe Gen3 x16 bus like GPU, allowing it to plug into existing servers
- Instruction buffer
  - TPU instructions are sent from the host over the PCIe into an instruction buffer



TPU Components

- Matrix Multiply Unit
  - Heart of TPU: it contains 256x256 (=65,536) MACs that can perform 8-bit multiply-and-adds on signed or unsigned integers
  - It reads and writes 256 values per clock cycle and perform either a matrix multiply or a convolution
  - It holds one 64 KiB tile of weights plus one for double buffering to hide the 256 cycles it takes to shift a tile in



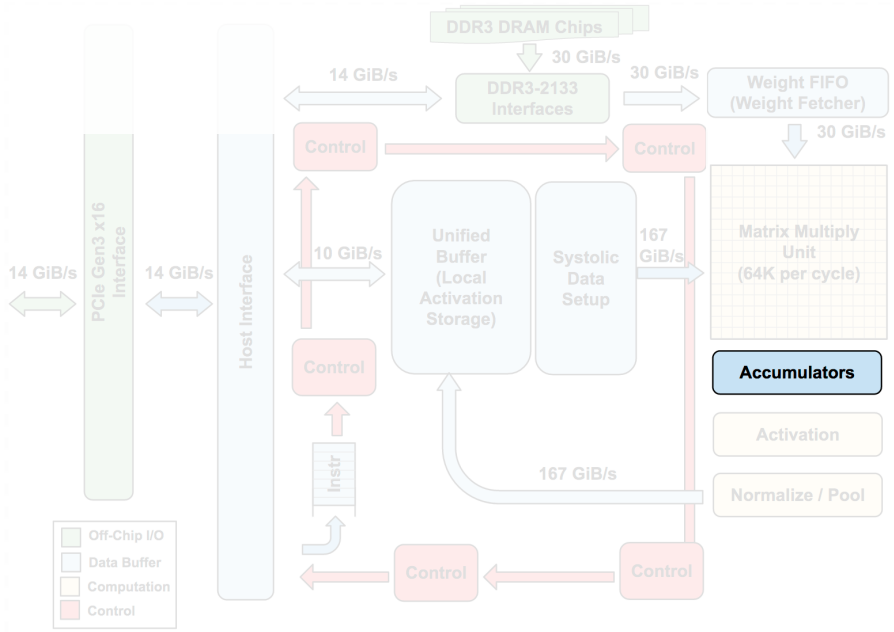
TPU Components

- Accumulators
  - 16-bit products are collected in 4 MiB of 32-bit accumulators
  - 4 MiB = 4096 x 256-element x 32-bit accumulators
  - How to pick 4096

    Peak performance based on roofline model, 1350 operations per byte

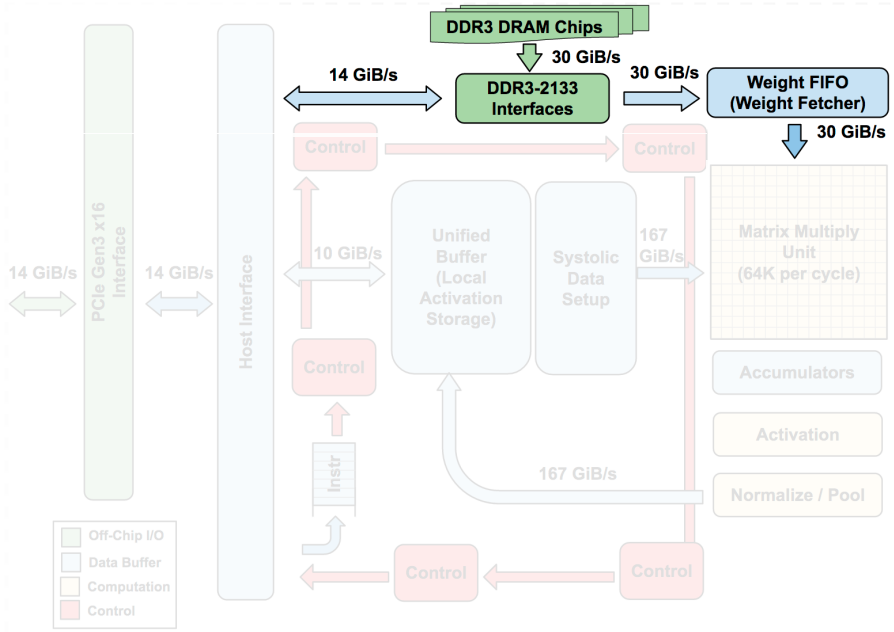
    Round up to 2048

    Double to 4096 for compiler



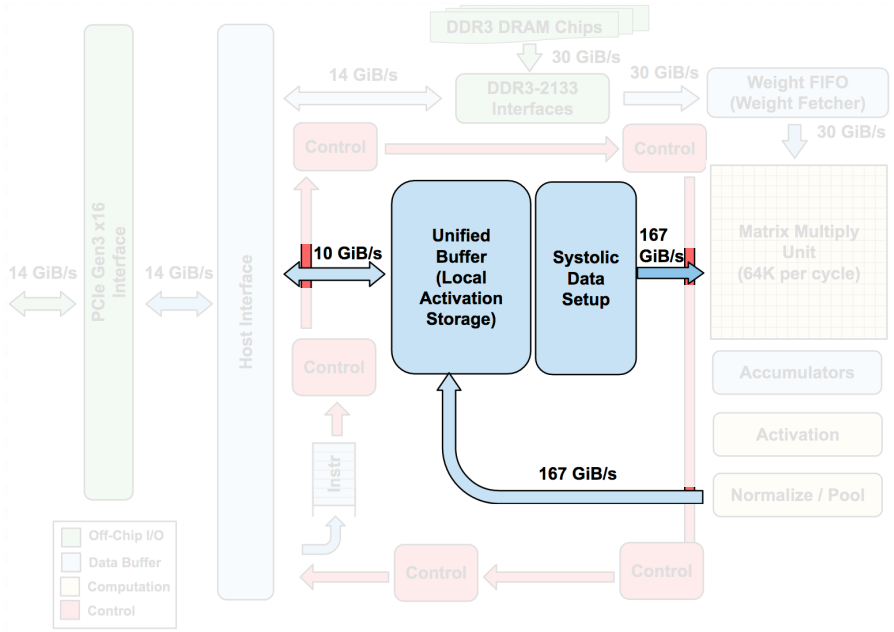
TPU Components

- Weight FIFO
  - On-chip FIFO that buffers weights for the matrix unit
  - Reads weights from an off-chip 8 GiB DRAM called Weight Memory

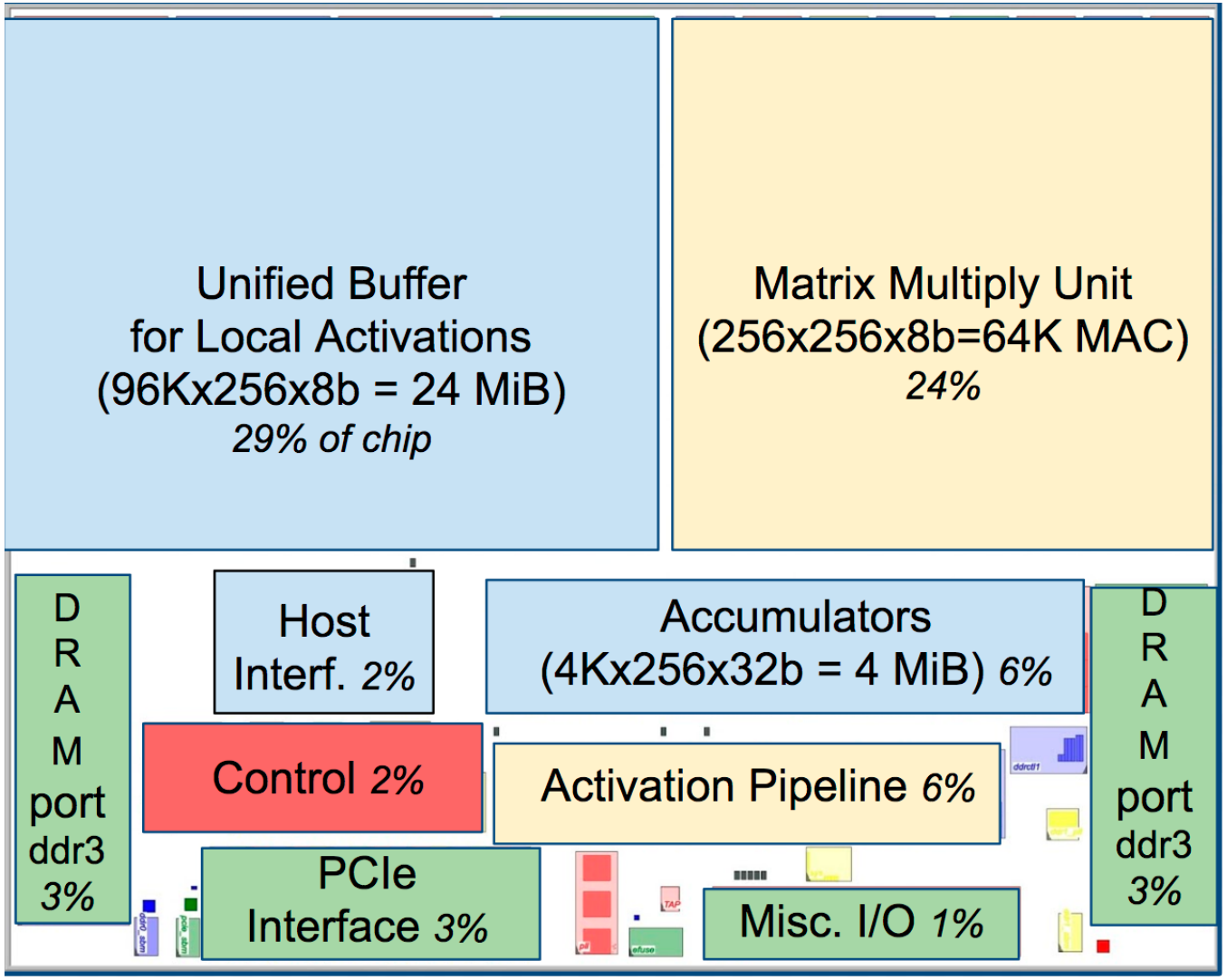


TPU Components

- Unified Buffer
  - Stores intermediate results (24 MiB)
  - Serves as inputs to the matrix unit
  - A programmable DMA controller transfers data between CPU host memory and the unified buffer



Floorplan of TPU die



- Datapath is nearly 2/3
  - Matrix Multiply Unit is 1/4
  - Unified buffer is 1/3
  - 24MiB was picked to match the pitch of the matrix unit die size
  - Control is only 2%

TPU Instructions

- TPU instructions follow CISC tradition including a repeat field
- Average clock cycles per instruction (CPI) is typically 10 to 20

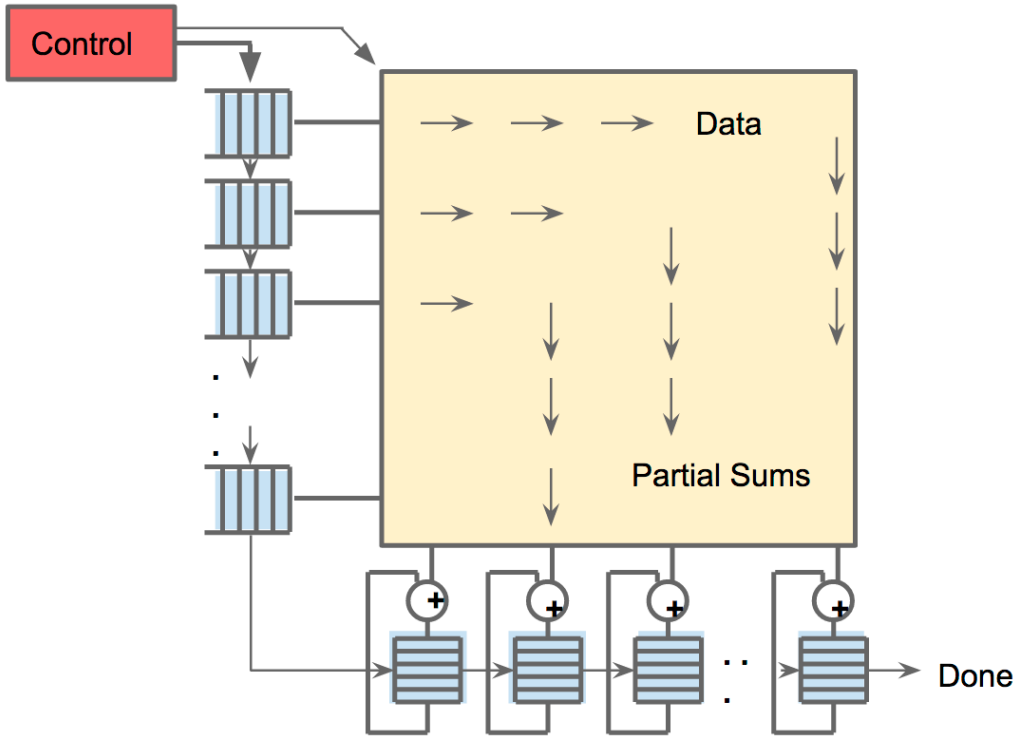
1. Read_Host_Memory reads data from the CPU host memory into the Unified Buffer
2. Read_Weights reads weights from Weight Memory (external) into the Weight FIFO (on-chip) as input to the Matrix Unit
3. MatrixMultiply/Convolve causes the Matrix Unit to perform a matrix multiply or a convolution from the Unified Buffer into the Accumulators
    - 12 byte instruction: Unified Buffer address (3 bytes), accumulator (2 bytes), length (4 bytes), opcode and flag (3 bytes)
4. Activate performs nonlinear function of artificial neurons (ReLU, Sigmoid, etc.).
    - Input is Accumulator and output is Unified Buffer
    - Performs also pooling with dedicated hardware on the die
5. Write_Host_Memory writes data from the Unified Buffer into the CPU host memory

Design Philosophy

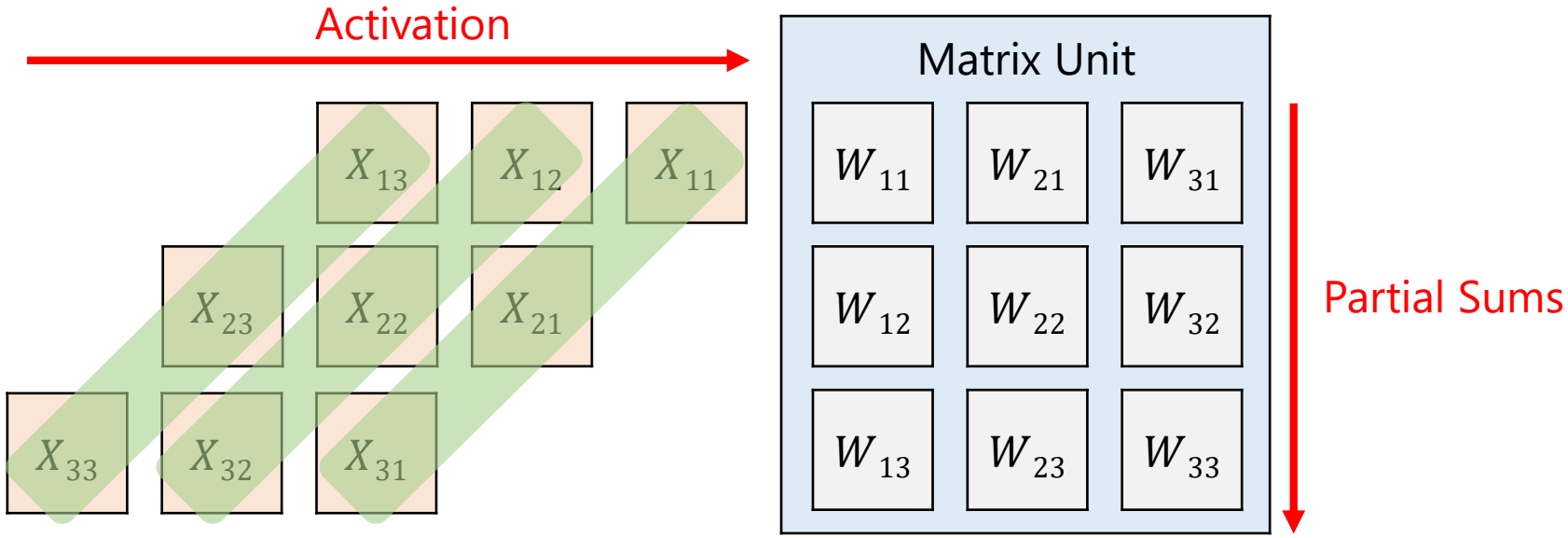
- Keep the matrix unit busy
- Overlapping a long matrix multiply instruction with others
  - Decouple access and execute
  - Pre-fetch weight data to hide its memory access latency
  - Matrix unit will stall if the input activation or weight data is not ready

Systolic Data Flow of Matrix Multiply Unit

- Systolic execution to save energy by reducing reads and writes of the Unified Buffer
- Activation data flows in from the left and weights are pre-loaded from the top
- A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront
- Control and data are pipelined
- Software is unaware of systolic nature of the matrix unit



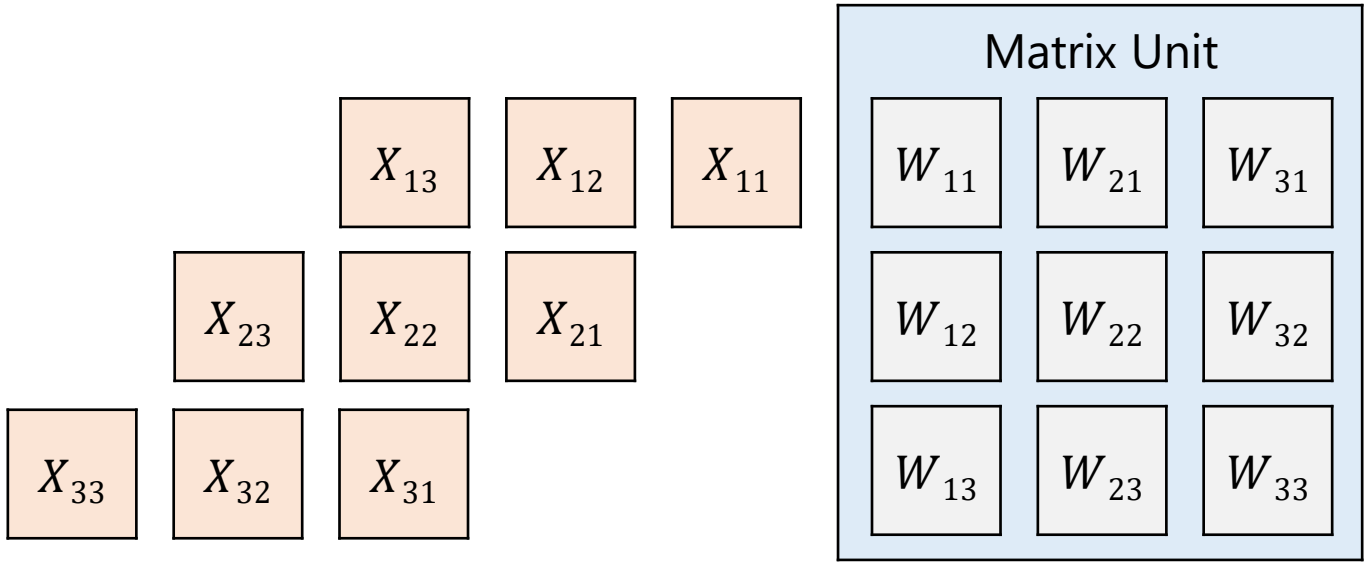
Systolic Data Flow of Matrix Multiply Unit



Computing Y = WX where W = 3x3, batch-size(X) = 3

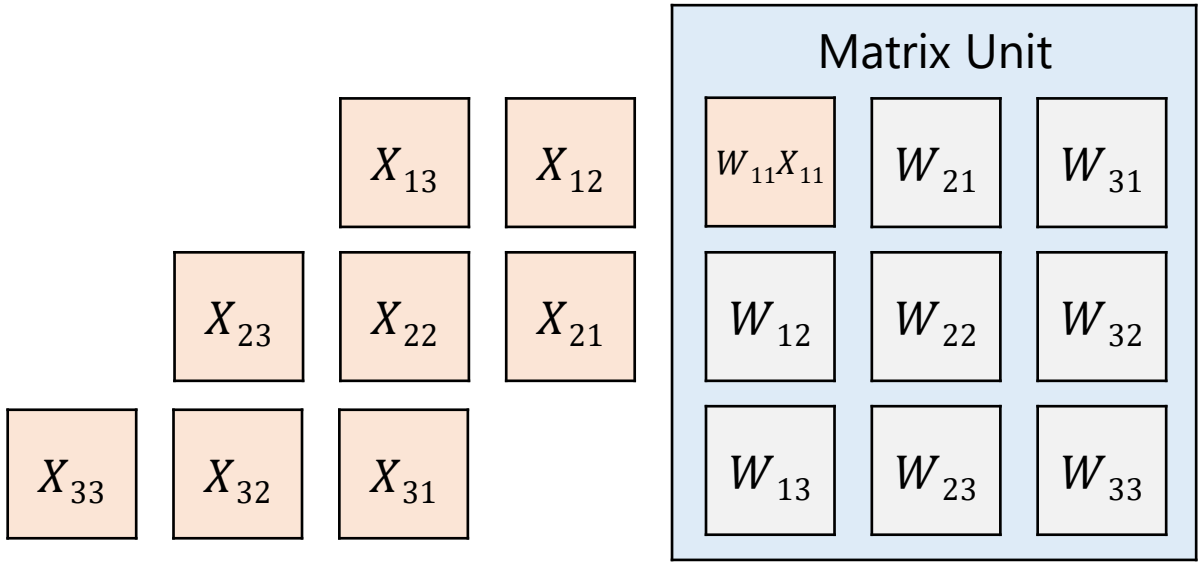
$$\begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{bmatrix} \begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} \\ Y_{21} & Y_{22} & Y_{23} \\ Y_{31} & Y_{32} & Y_{33} \end{bmatrix}$$

Systolic Data Flow of Matrix Multiply Unit



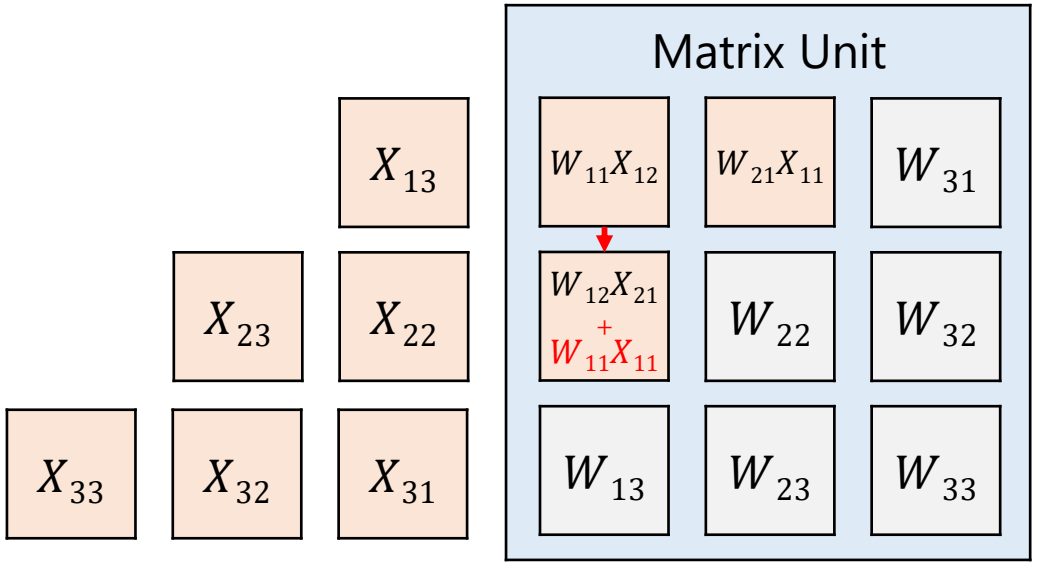
Cycle 0

Systolic Data Flow of Matrix Multiply Unit



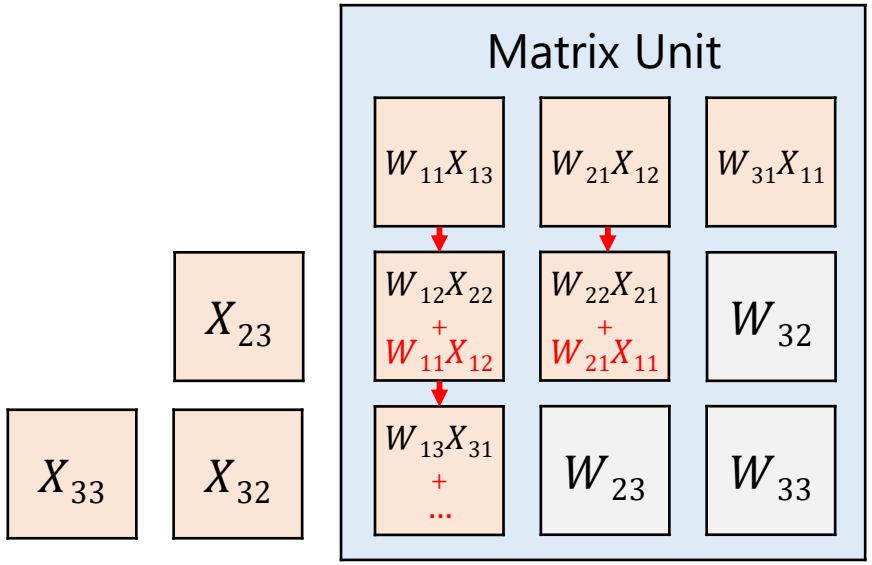
Cycle 1

Systolic Data Flow of Matrix Multiply Unit



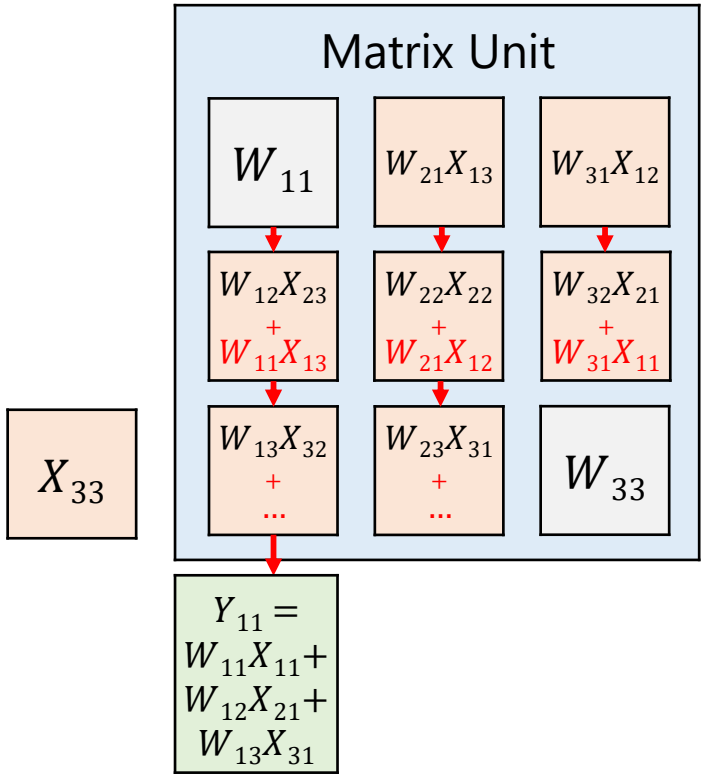
Cycle 2

Systolic Data Flow of Matrix Multiply Unit



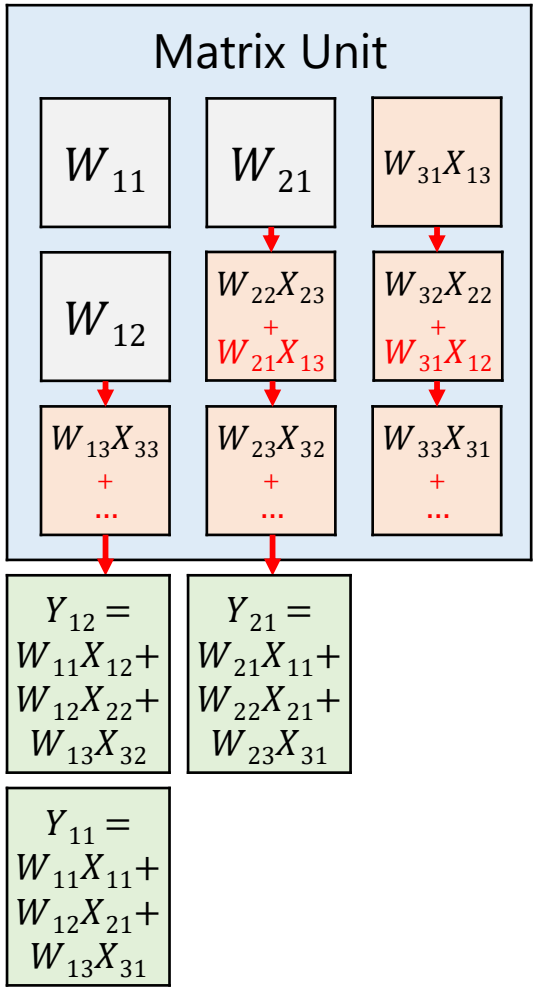
Cycle 3

Systolic Data Flow of Matrix Multiply Unit



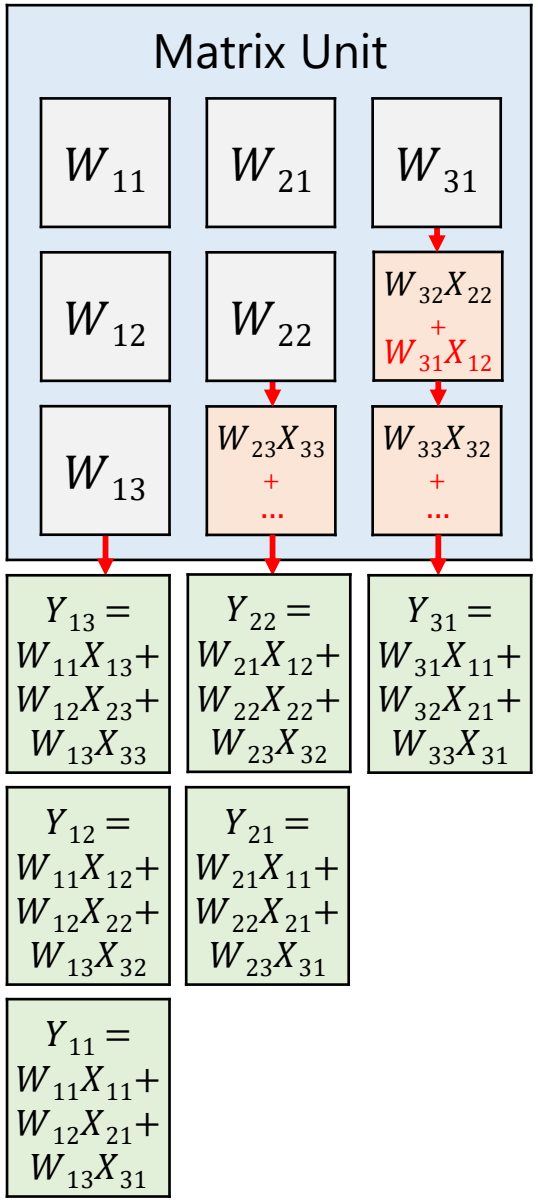
Cycle 4

Systolic Data Flow of Matrix Multiply Unit



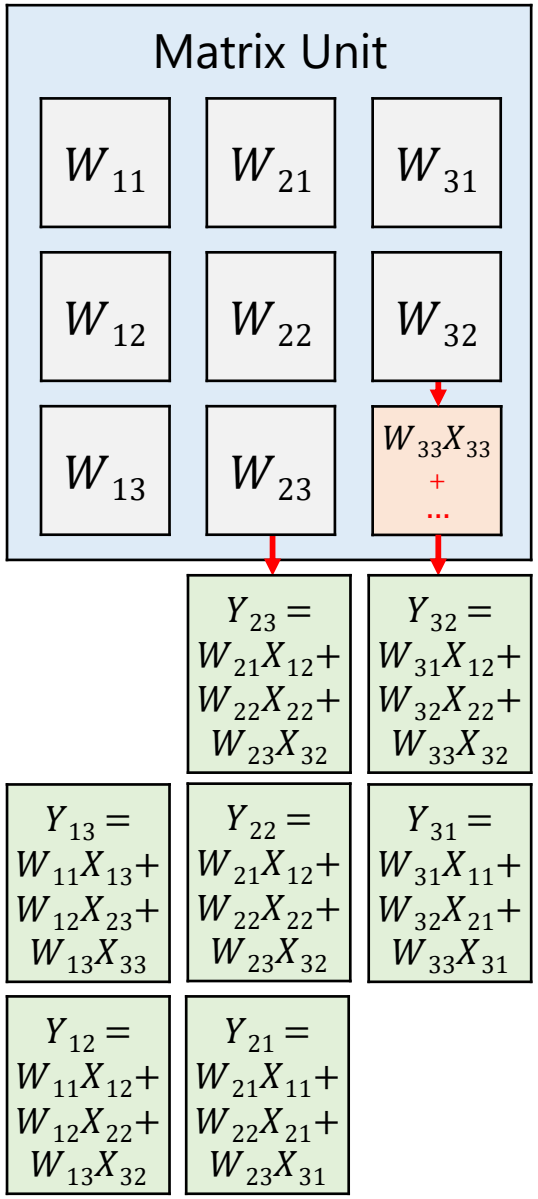
Cycle 5

Systolic Data Flow of Matrix Multiply Unit



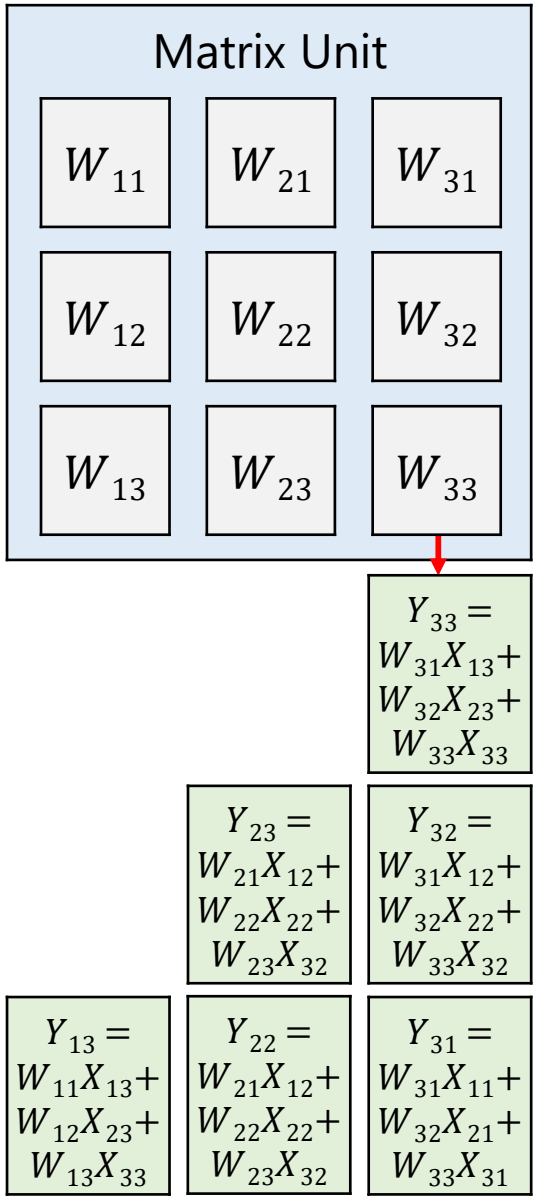
Cycle 6

Systolic Data Flow of Matrix Multiply Unit



Cycle 7

Systolic Data Flow of Matrix Multiply Unit



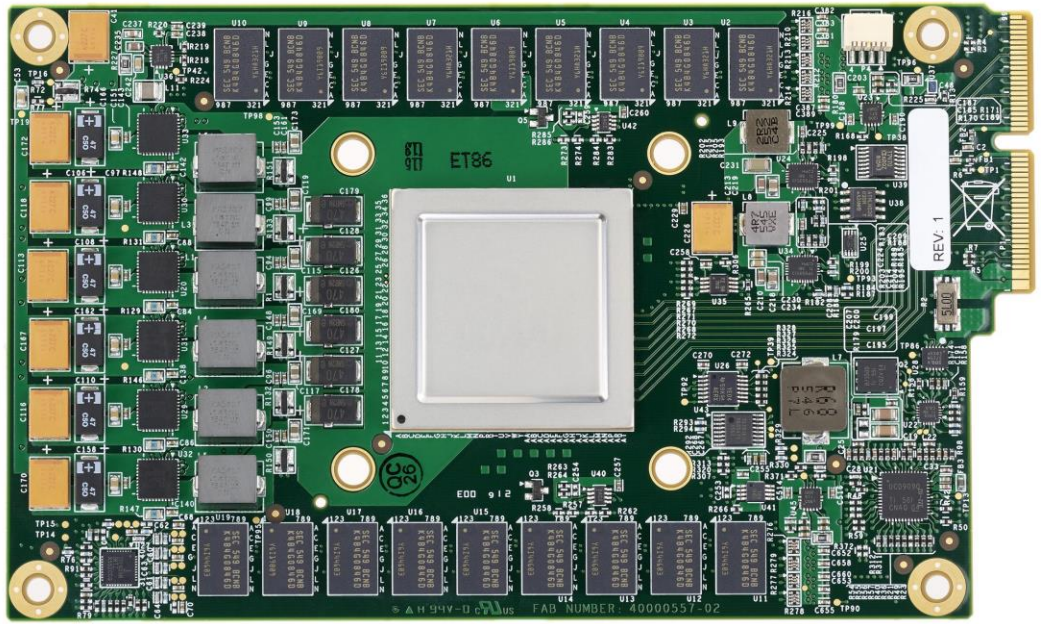
Cycle 8

Latency: 2N cycles to fill the pipelines

Throughput: N partial sums / cycle

Implement Results

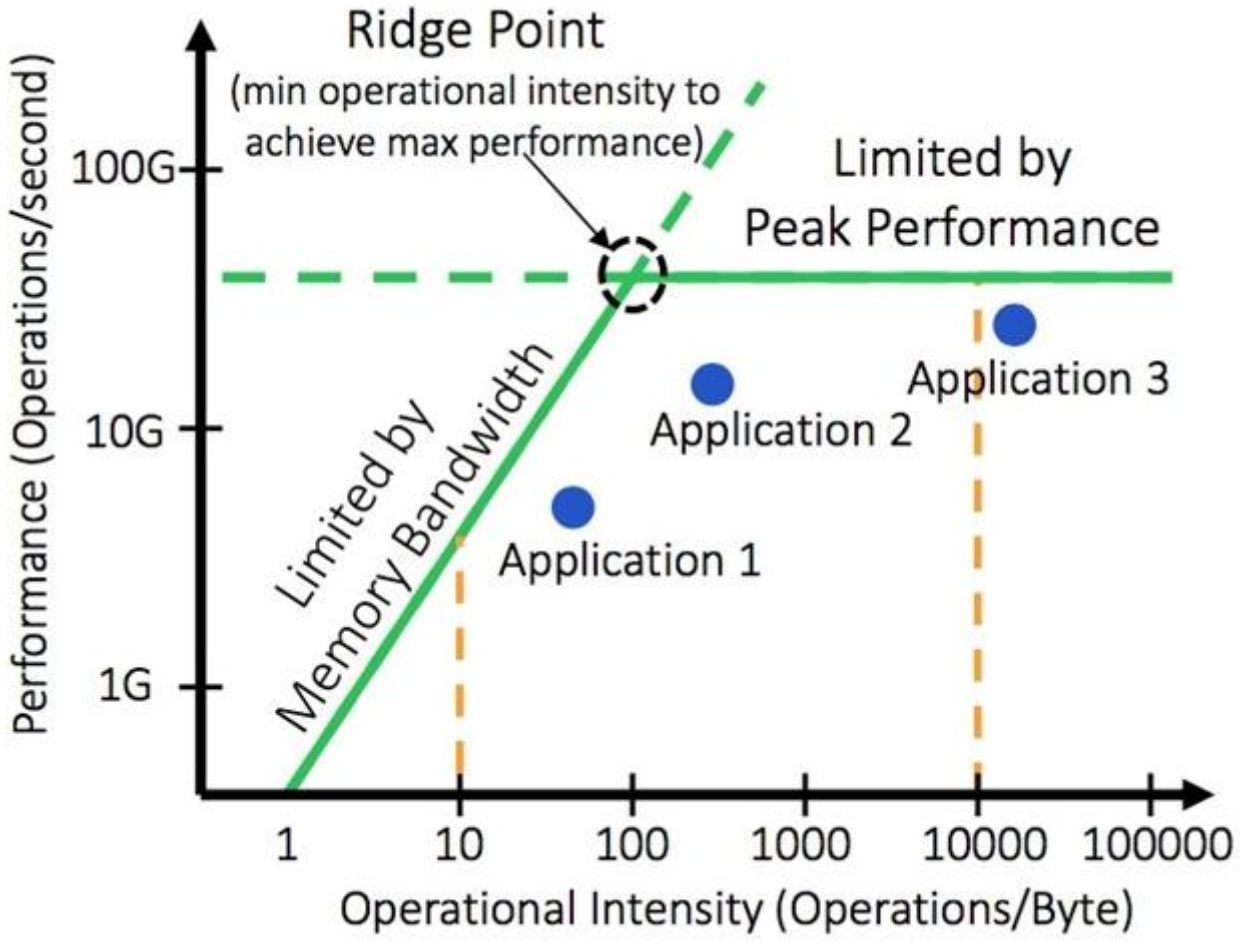
- TPU chip
  - 28nm process
  - Less than half the size of a Haswell E5 2699 v3 die
  - 700MHz operating clock
  - 92 TOPS (8b) @ 75W TDP



Model	Die										Benchmarked Servers				
	mm^2	nm	MHz	TDP	Measured		TOPS/s		GB/s	On-Chip Memory	Dies	DRAM Size	TDP	Measured	
					Idle	Busy	8b	FP						Idle	Busy
Haswell E5-2699 v3	662	22	2300	145W	41W	145W	2.6	1.3	51	51 MiB	2	256 GiB	504W	159W	455W
NVIDIA K80 (2 dies/card)	561	28	560	150W	25W	98W	--	2.8	160	8 MiB	8	256 GiB (host) + 12 GiB x 8	1838W	357W	991W
TPU	<331*	28	700	75W	28W	40W	92	--	34	28 MiB	4	256 GiB (host) + 8 GiB x 4	861W	290W	384W

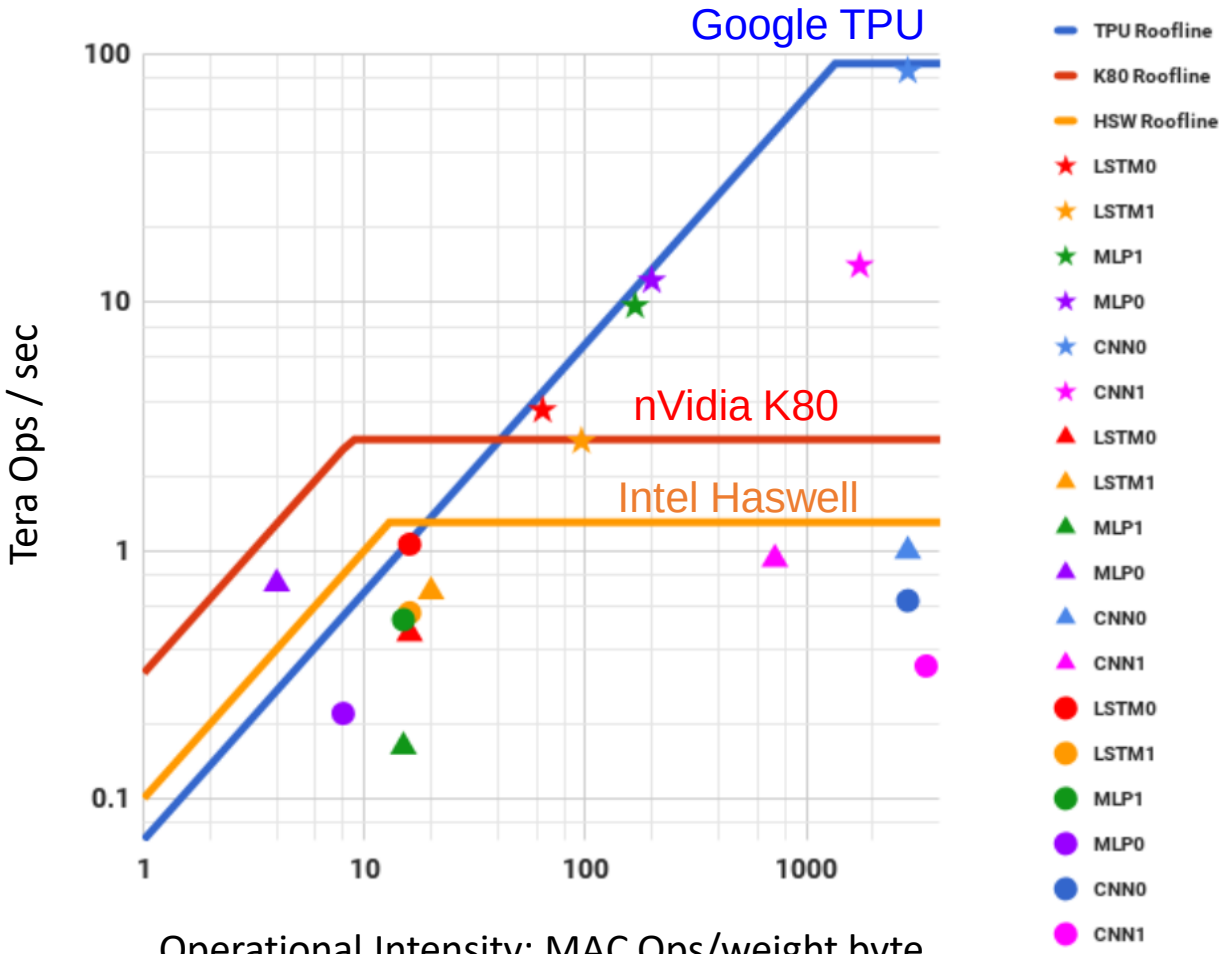
Roofline Model

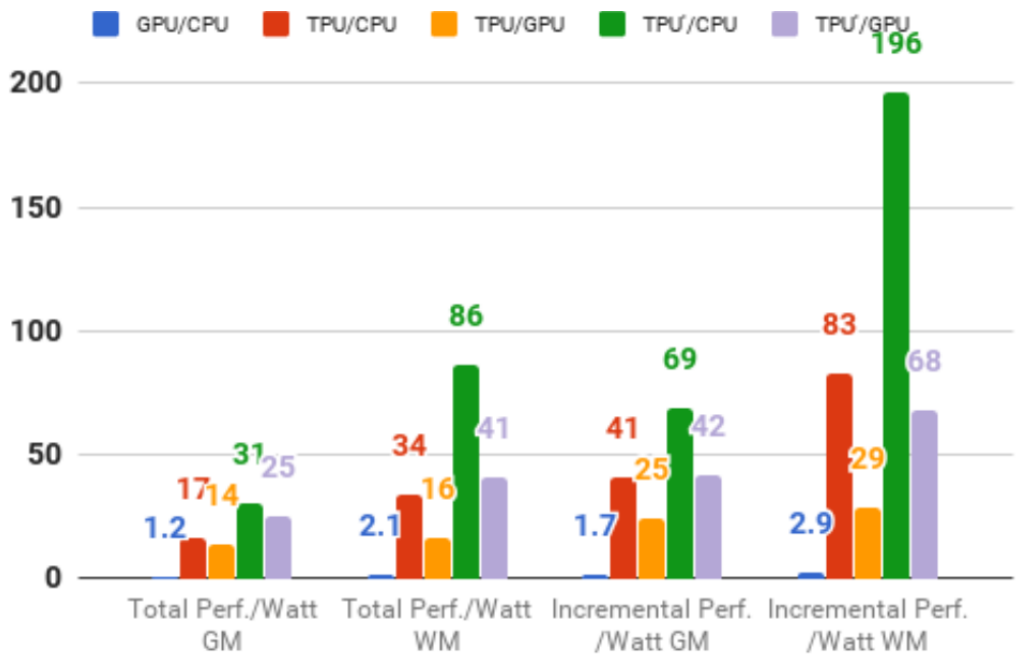
- Visually intuitive performance model
- Compute bound vs memory bound



Experimental Results

- CPU vs GPU vs TPU

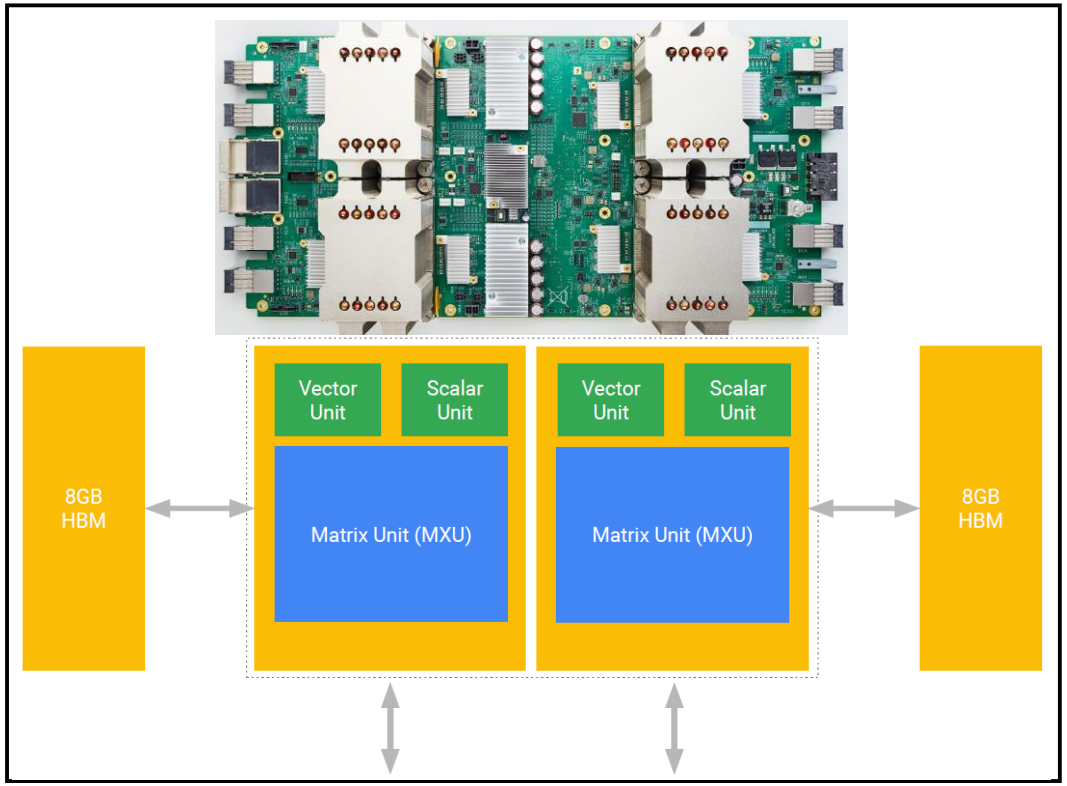




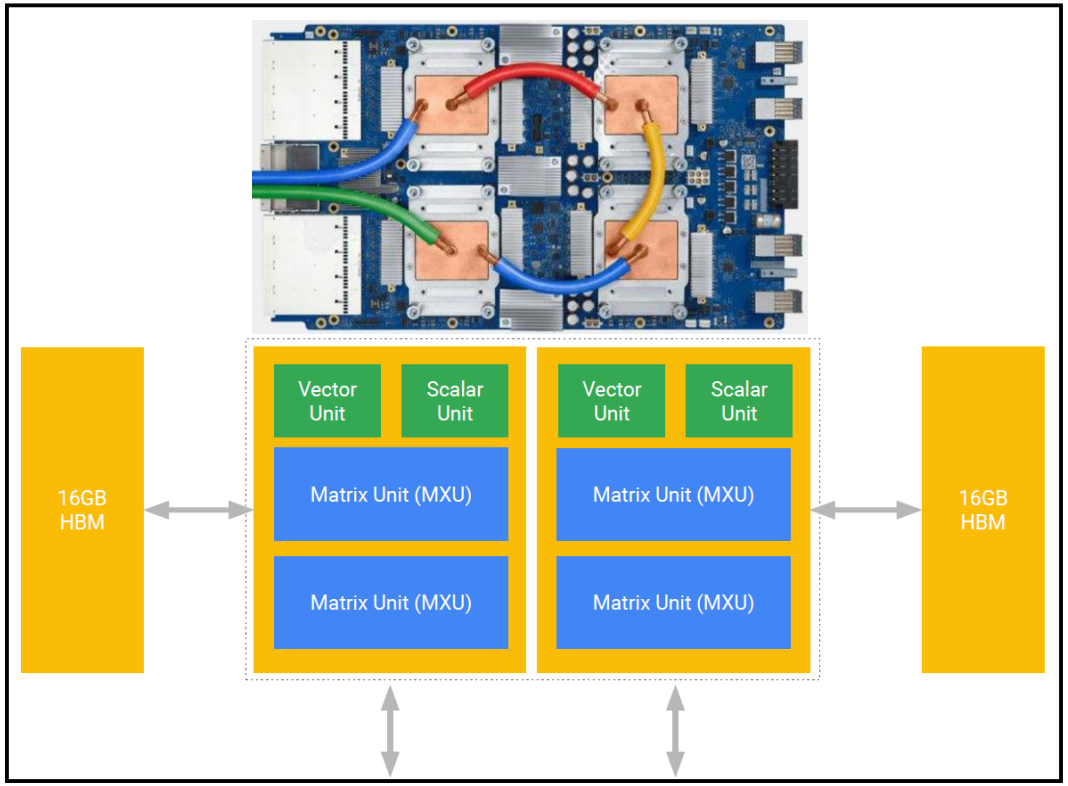
GM: geometric mean, WM: weighted mean

TPU v2 & TPU v3

- 128 x 128 systolic array (22.5 TFLOPS per core)
- float32 accumulate / bfloat16 multiplies
- 2 cores + 2 HBMs per chip / 4 chips per board



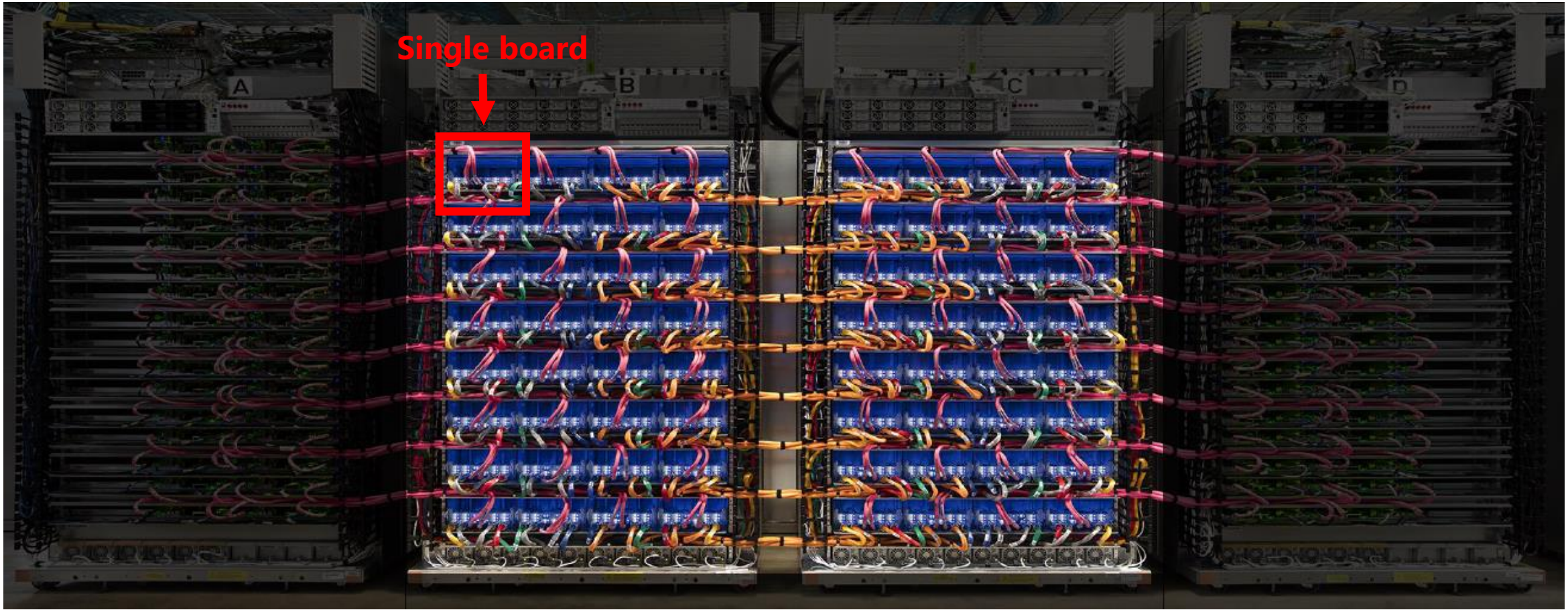
TPU v2



TPU v3

Cloud TPU v2 Pod

- Single board: 180TFLOPS + 64GB HBM
- Single pod (64 boards):11.5 PFLOPS + 4TB HBM
- 2D torus topology



Cloud TPU v3 Pod

- > 100 PFLOPS
- 32TB HBM

Project BrainWave (ISCA 2018)

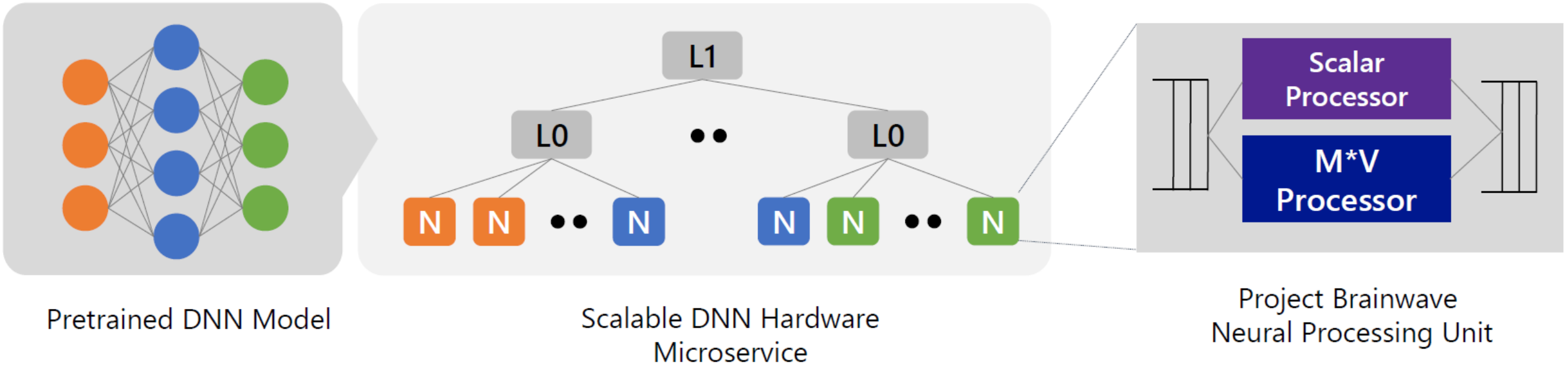
- Serving DNNs in real-time datacenter scale
- DNNs are challenging to serve in interactive services
  - Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs)
- Neural processing units (NPUs) are promising for real-time AI

- Must satisfy 3 requirements:
  - Low-latency
  - Flexible for long shelf life
  - Programmable and easy to use

BrainWave NPU

- High throughput, no batching without sacrificing latency & flexibility
- Achieves 48 TFLOPS (96,000 MACs) on Intel Stratix 10 FPGAs
- Hardware utilization at single batch up to 75%
- DeepBench RNNs < 4ms, ResNet-50 < 2ms
- Specialized and deployed on FPGAs at cloud scale

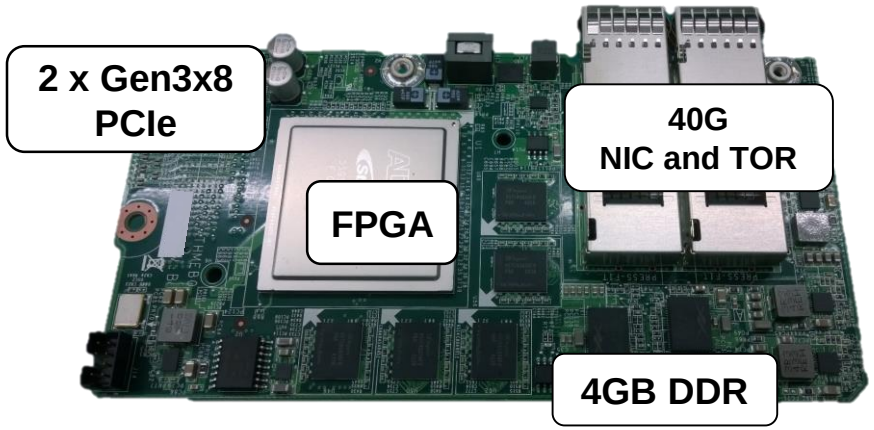
System Architecture



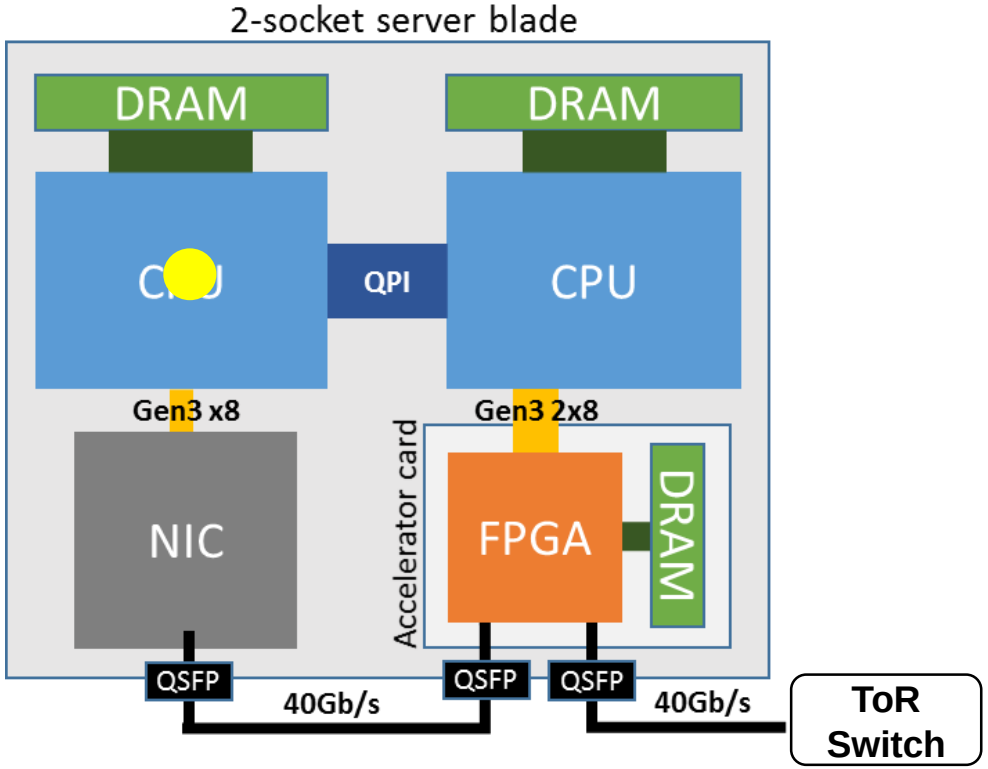
1. A software tool flow for low-friction deployment (TensorFlow -> FPGAs)
2. A distributed FPGA infrastructure for hardware microservices (Catapult)
3. A high performance soft DNN processor synthesized on FPGAs (NPU)

Project Catapult

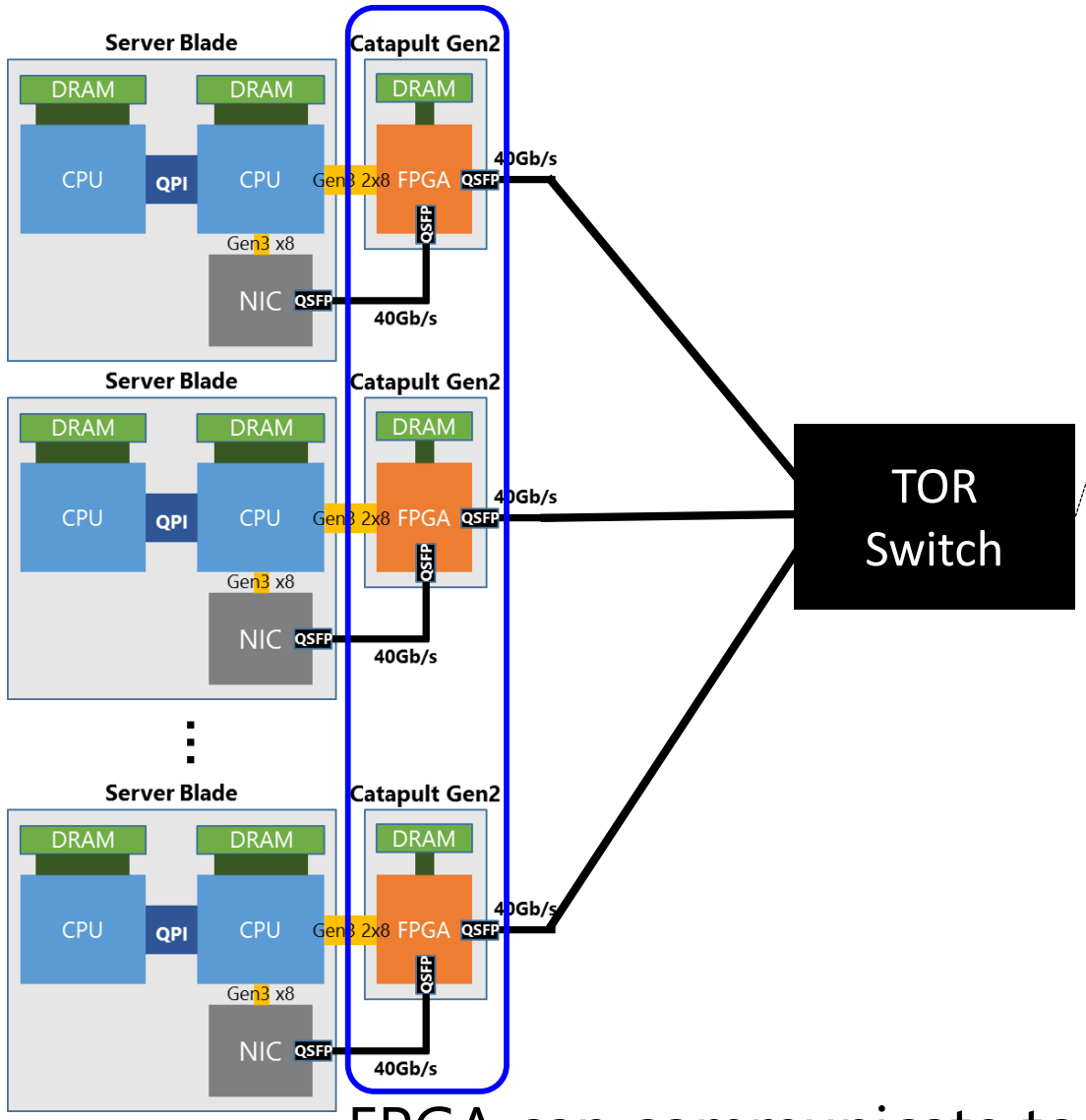
- FPGA accelerator for Microsoft datacenters
  - Dual 40Gb/s networking ports
  - Bump-in-the-Wire (NIC↔FPGA↔Switch)
  - FPGA shares datacenter network

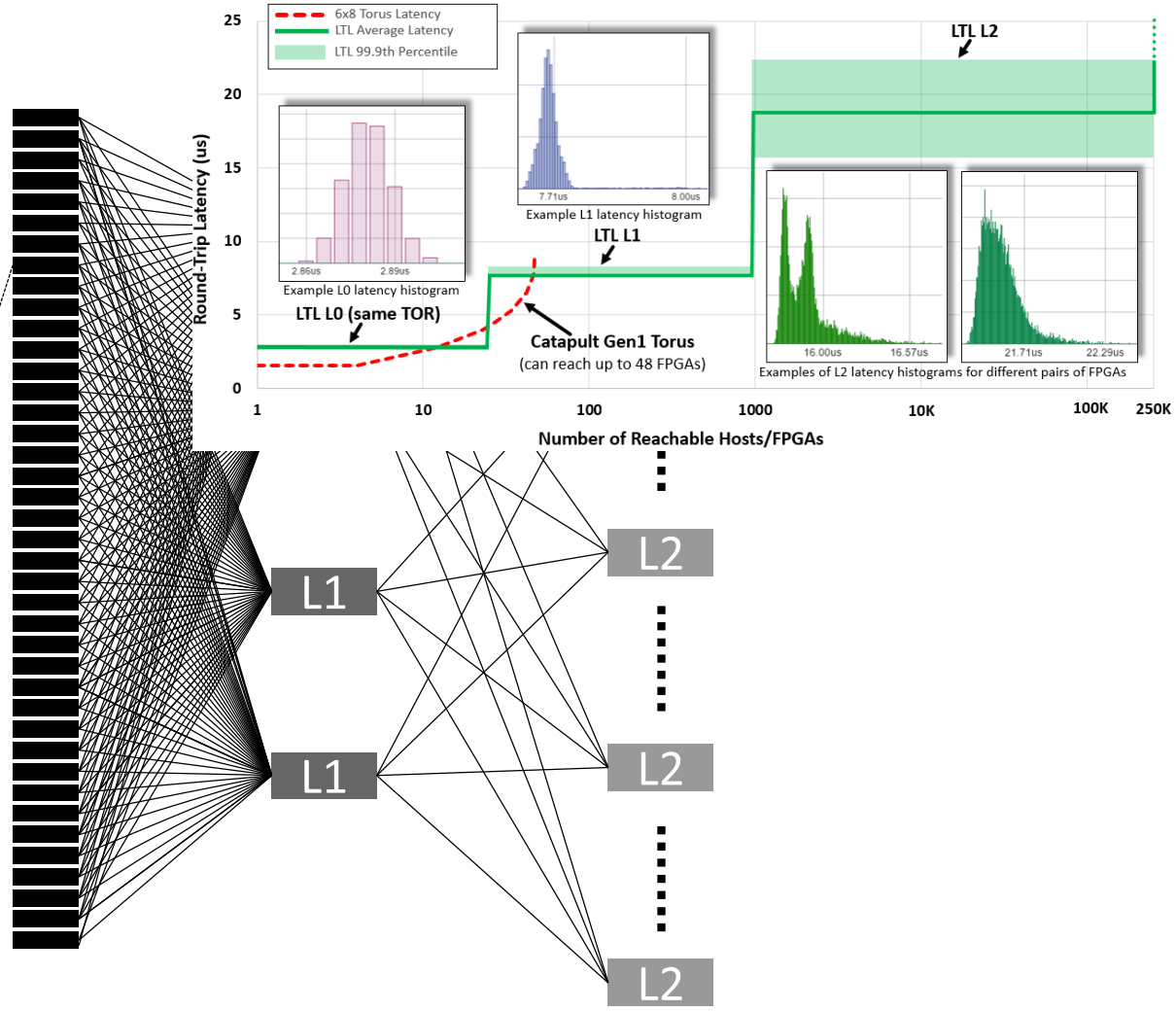


35W power budget



Accelerator Integration to Network Infrastructure

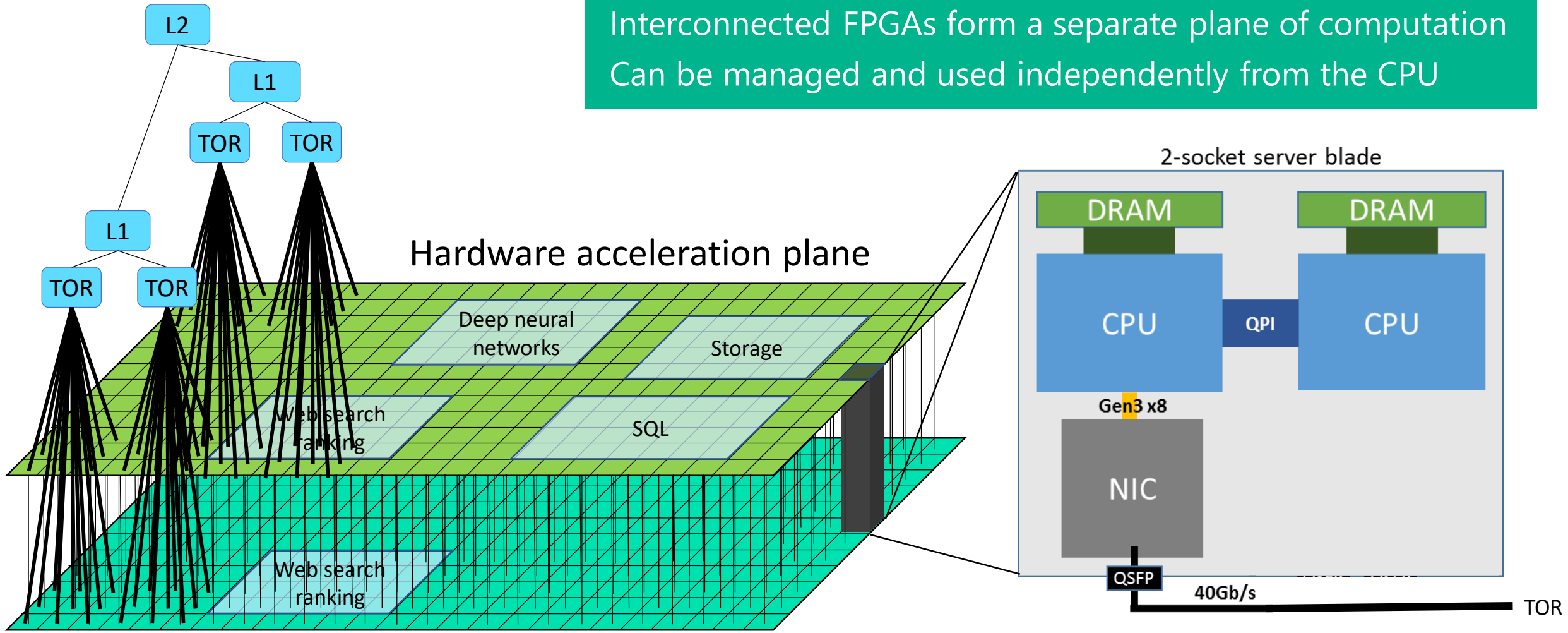




FPGA can communicate to any other FPGA in datacenter

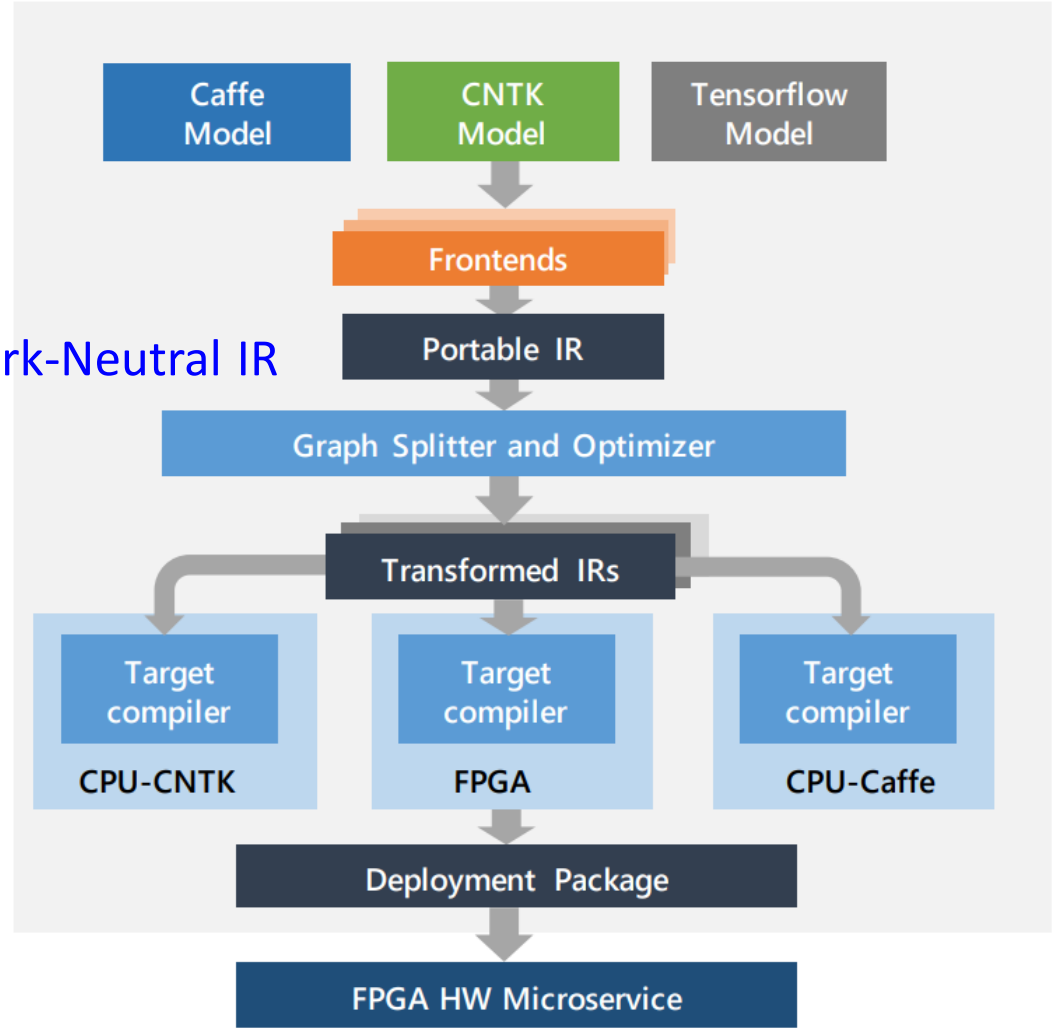
Configurable Cloud

Interconnected FPGAs form a separate plane of computation
Can be managed and used independently from the CPU

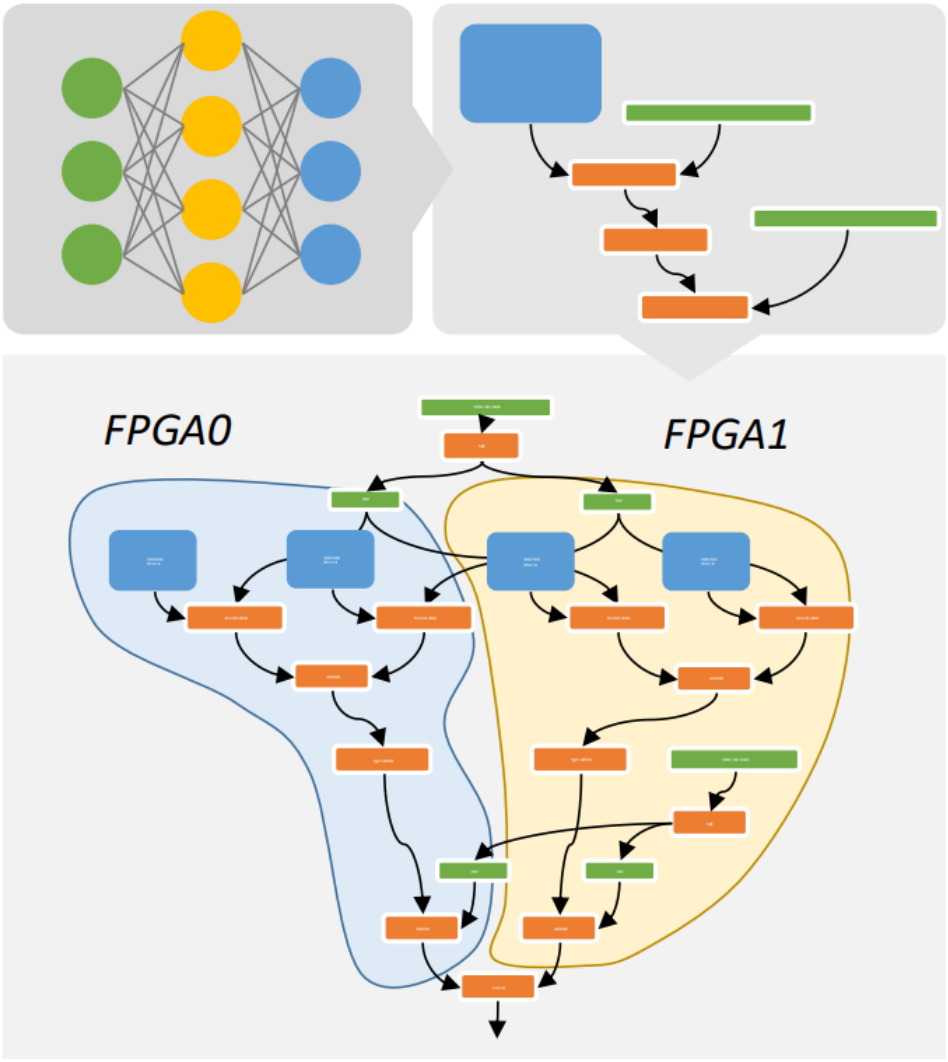


Tool Flow

Framework-Neutral IR



Leverage Catapult Infrastructure

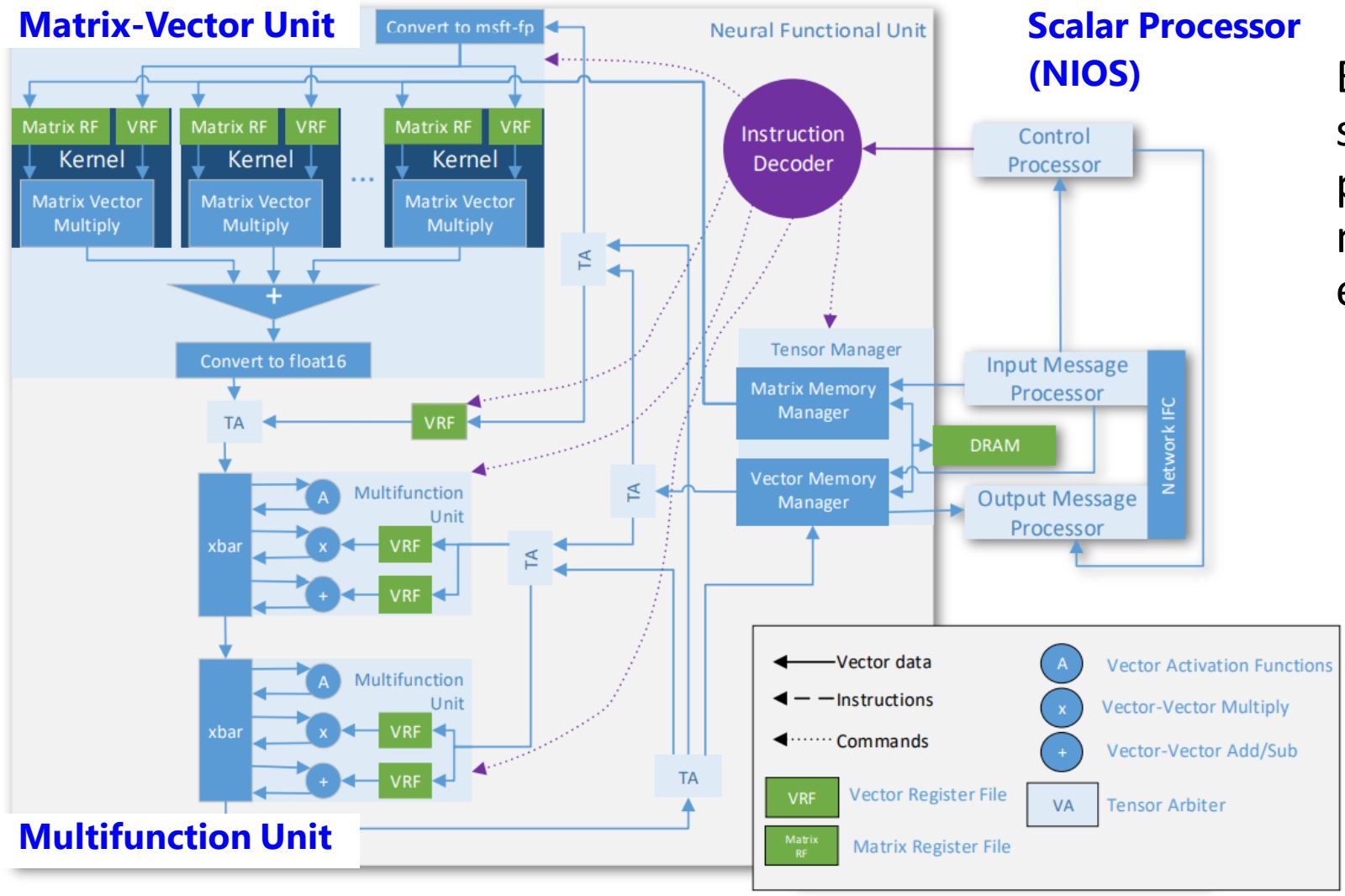


Partitions into subgraphs for target devices (CPUs, FPGAs)

BrainWave NPU

"**Mega-SIMD**" execution (> 1M ops per instruction)

Instructions operate on multiples of a native dimension N



Expose a simple, singe-threaded programming model to user w/ extensible ISA

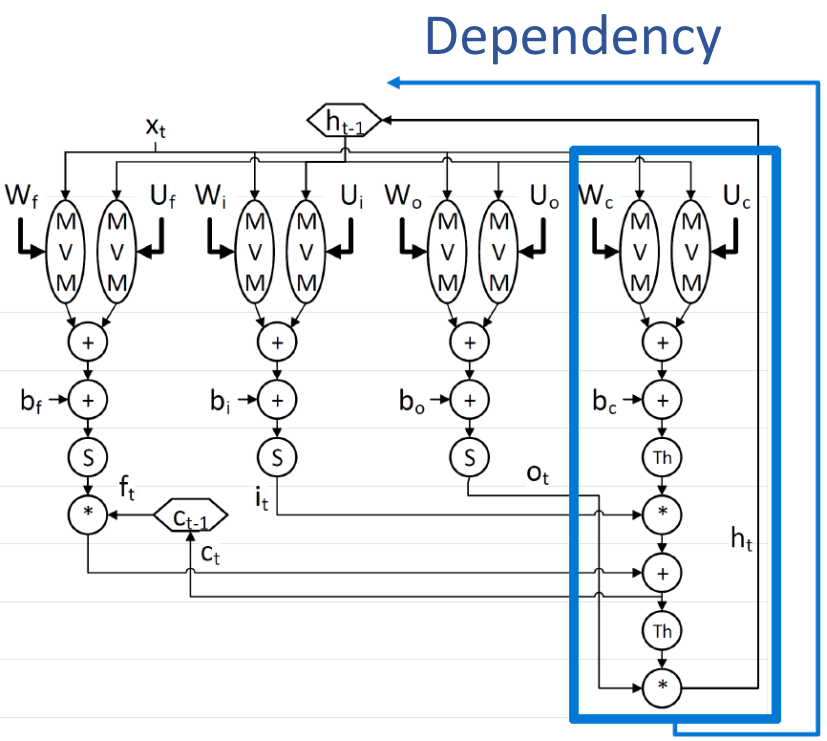
Instruction chaining for non-linear, less frequent functions

NPU Instructions

Name	Description	IN	Operand 1	Operand 2	OUT
v_rd	Vector read	-	MemID	Memory index	V
v_wr	Vector write	V	MemID	Memory index	-
m_rd	Matrix read	-	MemID (NetQ or DRAM only)	Memory index	M
m_wr	Matrix write	M	MemID (MatrixRf or DRAM only)	Memory index	-
mv_mul	Matrix-vector multiply	V	MatrixRf index	-	V
vv_add	PWV addition	V	AddSubVrf index	-	V
vv_a_sub_b	PWV subtraction, IN is minuend	V	AddSubVrf index	-	V
vv_b_sub_a	PWV subtraction, IN is subtrahend	V	AddSubVrf index	-	V
vv_max	PWV max	V	AddSubVrf index	-	V
vv_mul	Hadamard product	V	MultiplyVrf index	-	V
v_relu	PWV ReLU	V	-	-	V
v_sigm	PWV sigmoid	V	-	-	V
v_tanh	PWV hyperbolic tangent	V	-	-	V
s_wr	Write scalar control register	-	Scalar reg index	Scalar value	-
end_chain	End instruction chain	-	-	-	-

PWV = point-wise vector operation. IN = implicit input (V: vector, M: matrix, -: none). OUT = implicit output.

Example LSTM Program

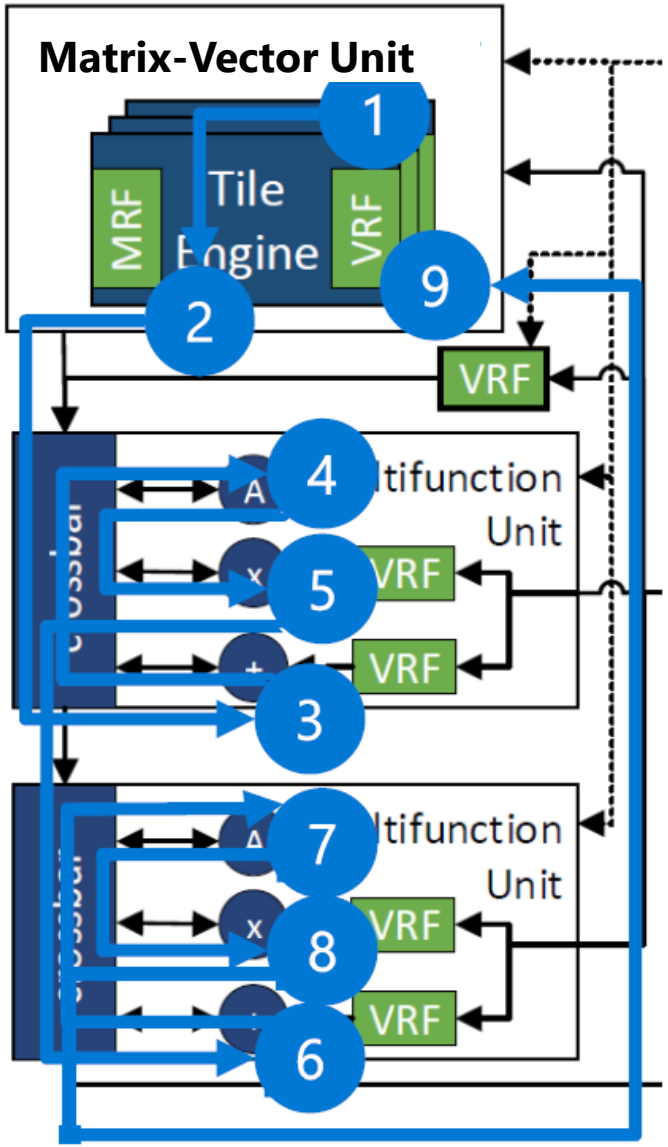



```
1.  void LSTM(int steps) {
2.    for (int t = 0; t < steps; t++) {
3.      v_rd(NetQ);
4.      v_wr(InitialVrf, xt);
5.      v_rd(InitialVrf, xt);
6.      mv_mul(Wf);
7.      vv_add(bf);
8.      v_wr(AddSubVrf, xWf);
9.      v_rd(InitialVrf, h_prev);
10.     mv_mul(Uf);
11.     vv_add(xWf);
12.     v_sigm();
13.     vv_mul(c_prev);
14.     v_wr(AddSubVrf, ft_mod);
15.     v_rd(InitialVrf, h_prev);
```

```
16.     mv_mul(Uc);
17.     vv_add(xWc);
18.     v_tanh();
19.     vv_mul(it);
20.     vv_add(ft_mod);
21.     v_wr(MultiplyVrf, c_prev);
22.     v_wr(InitialVrf, ct);
23.     v_rd(InitialVrf, ct);
24.     v_tanh();
25.     vv_mul(ot);
26.     v_wr(InitialVrf, h_prev);
27.     v_wr(NetQ);
28.   }
29. }
```

Microarchitecture

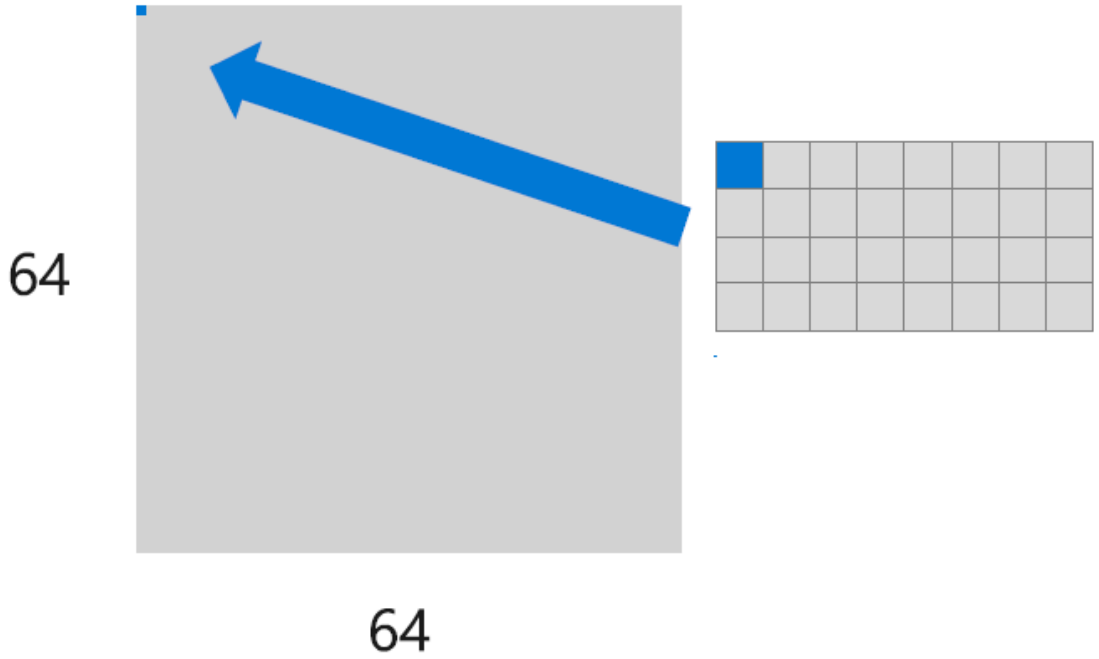
- Execute a continuous stream of instruction chains
- Instruction chaining reduces latency of critical path

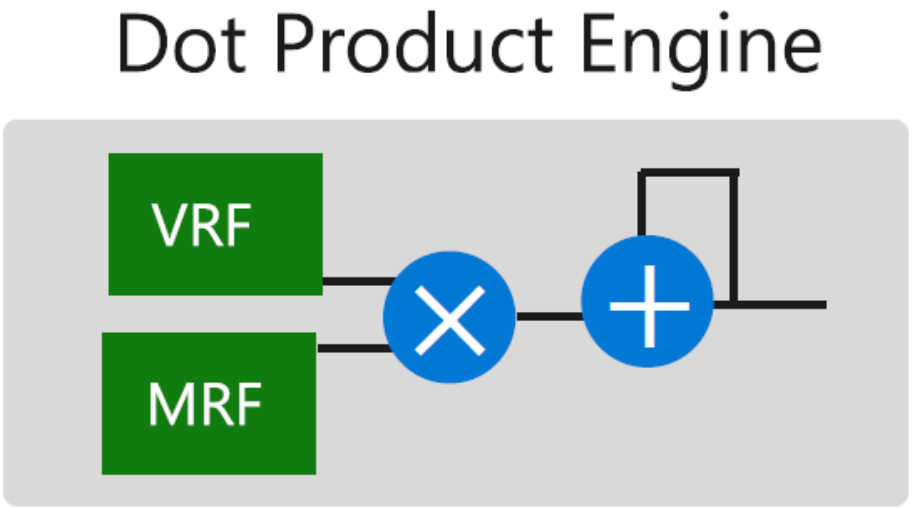


1. Read Vector x
2. M*V by W
3. Add Vector by b
4. Vector Tanh
5. Multiply Vector by i
6. Add Vector by f
7. Vector Tanh
8. Multiply by o
9. Write Vector h

Scaling M*V: Single Spatial Unit

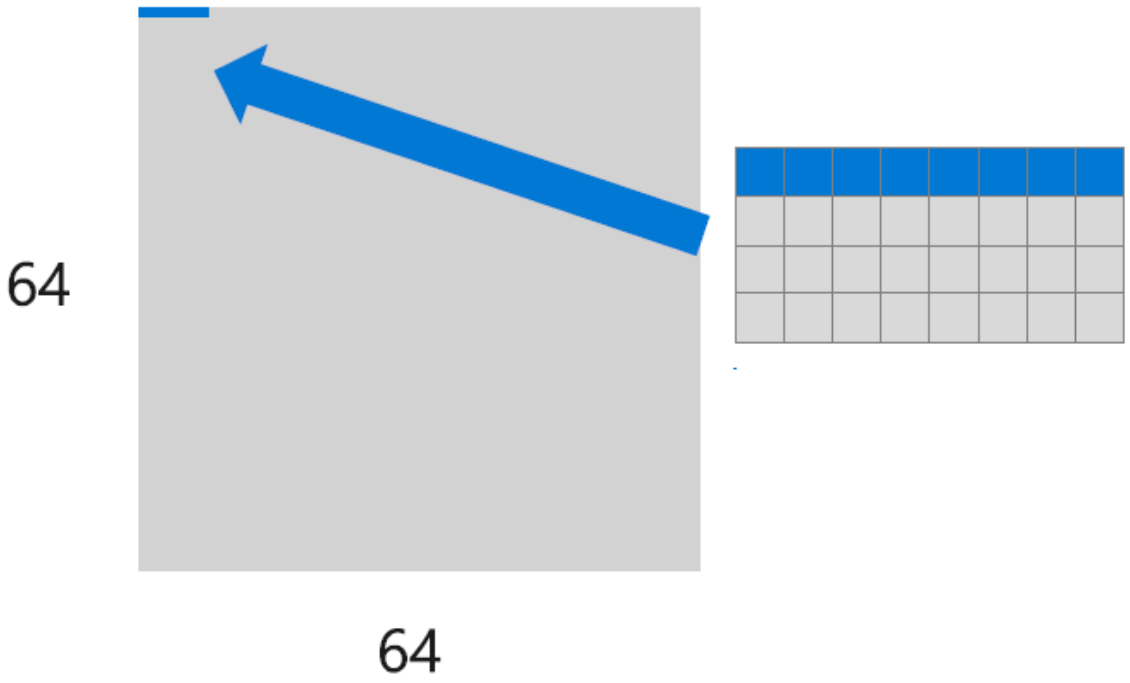
- Start with a primitive 1 MAC for M*V
- Vector and matrix register files (VRF, MRF) read 1 word/cycle

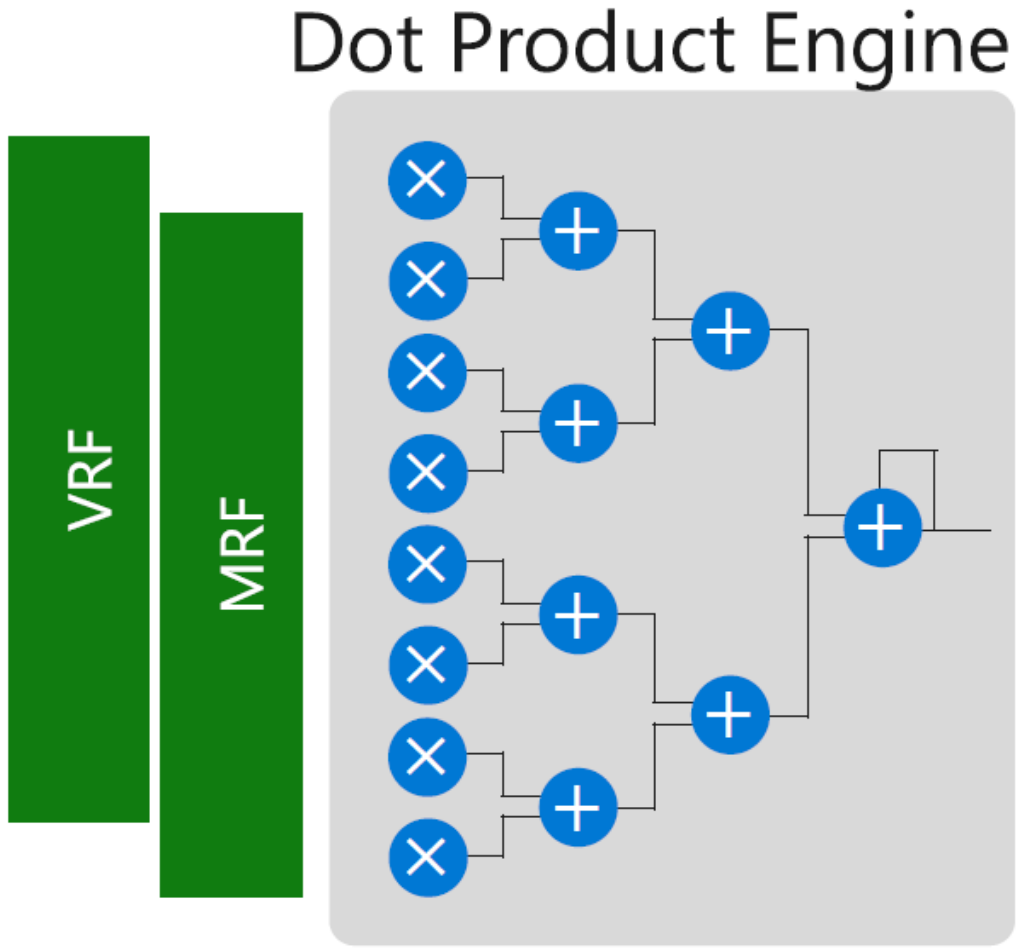




Scaling M*V: Multi-Lane Vector Spatial Unit

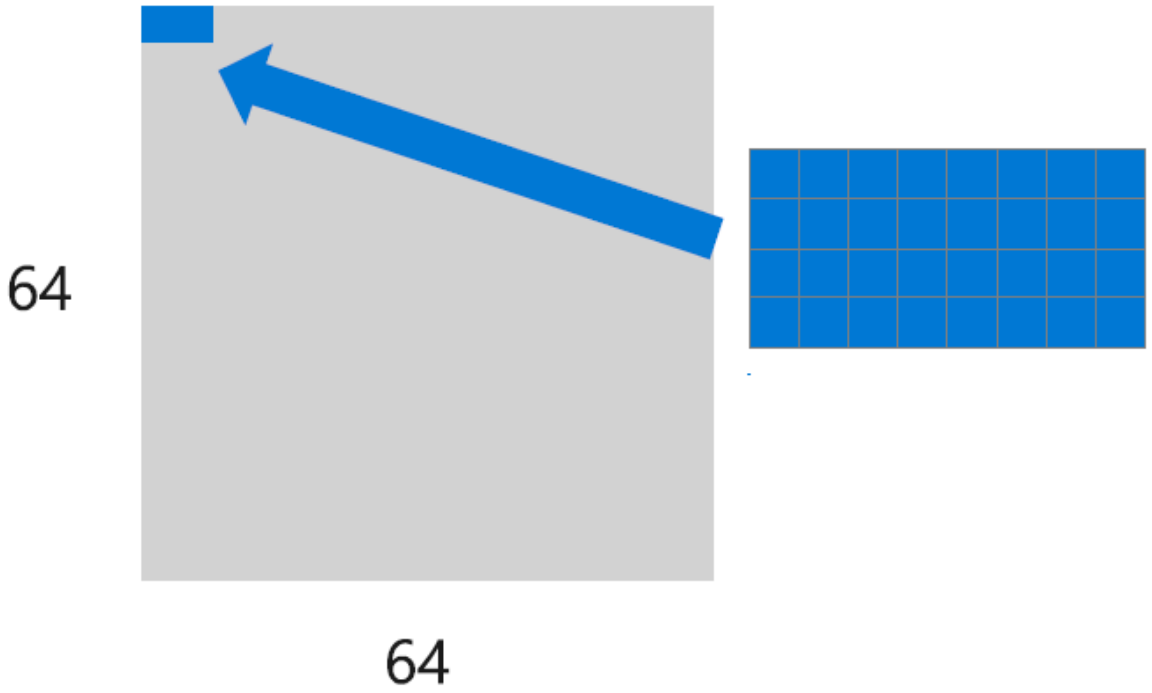
- 8 Compute lanes = 8 MACs
- Column parallelism
- More lanes = bigger RFs

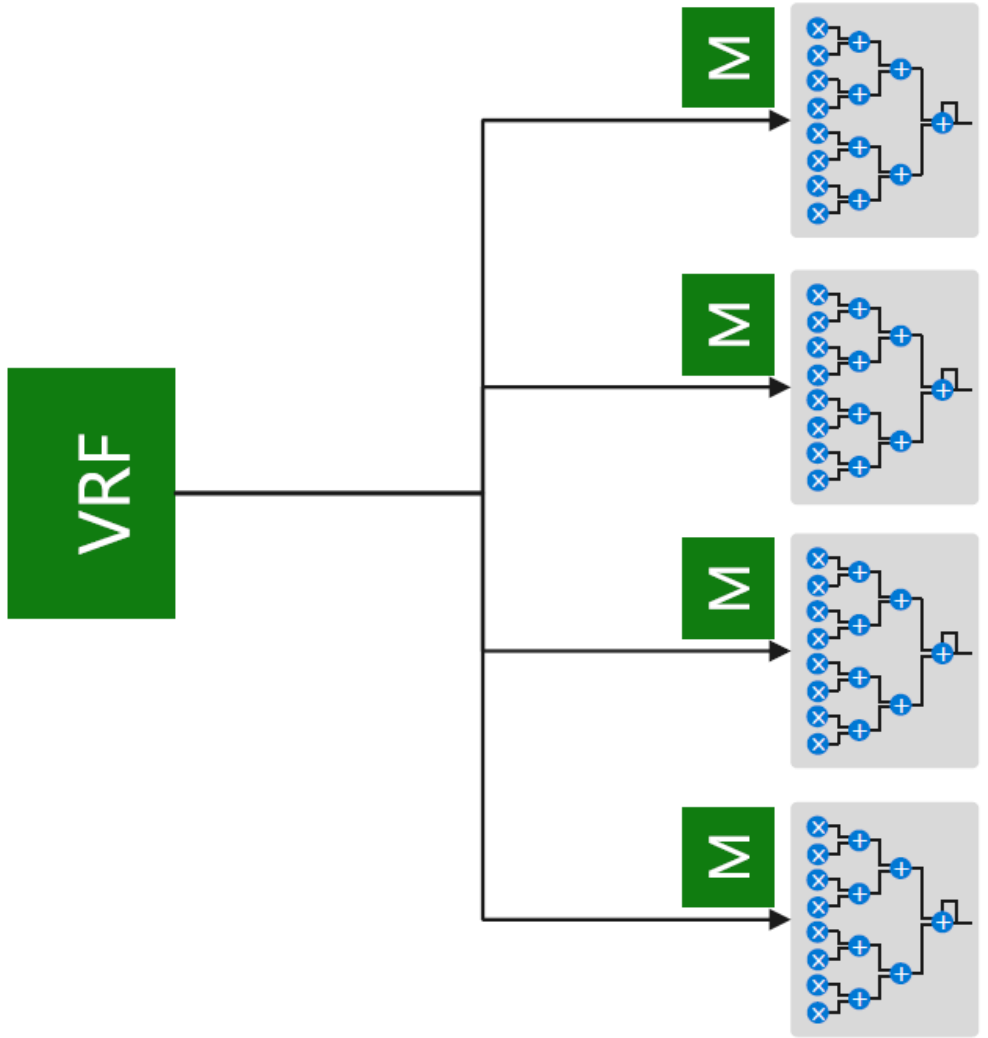




Scaling M*V: Vector Spatial Unit Replication

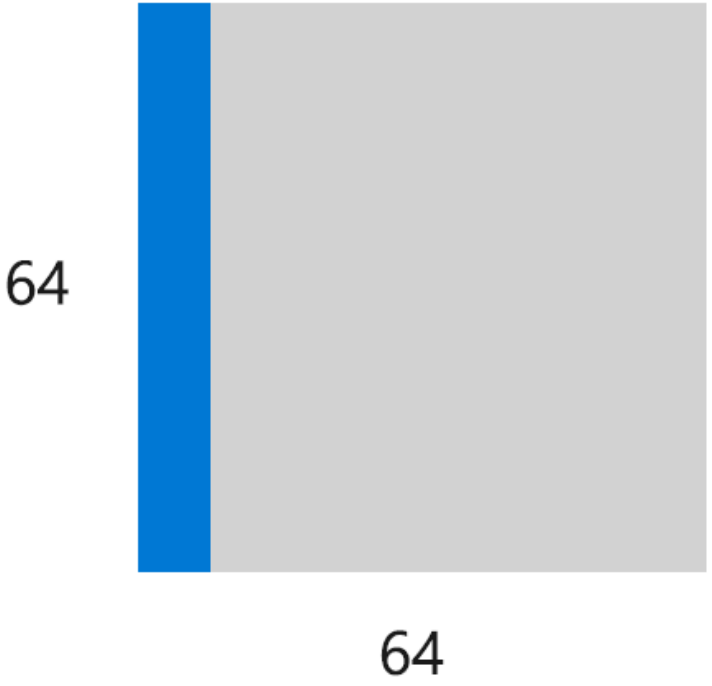
- Replicate 8-lane dot product engine (DPE) x 4 = 32 MACs
- Row parallelism, distributed MRF
- Broadcast VRF across DPEs
- # of rows = # of DPEs

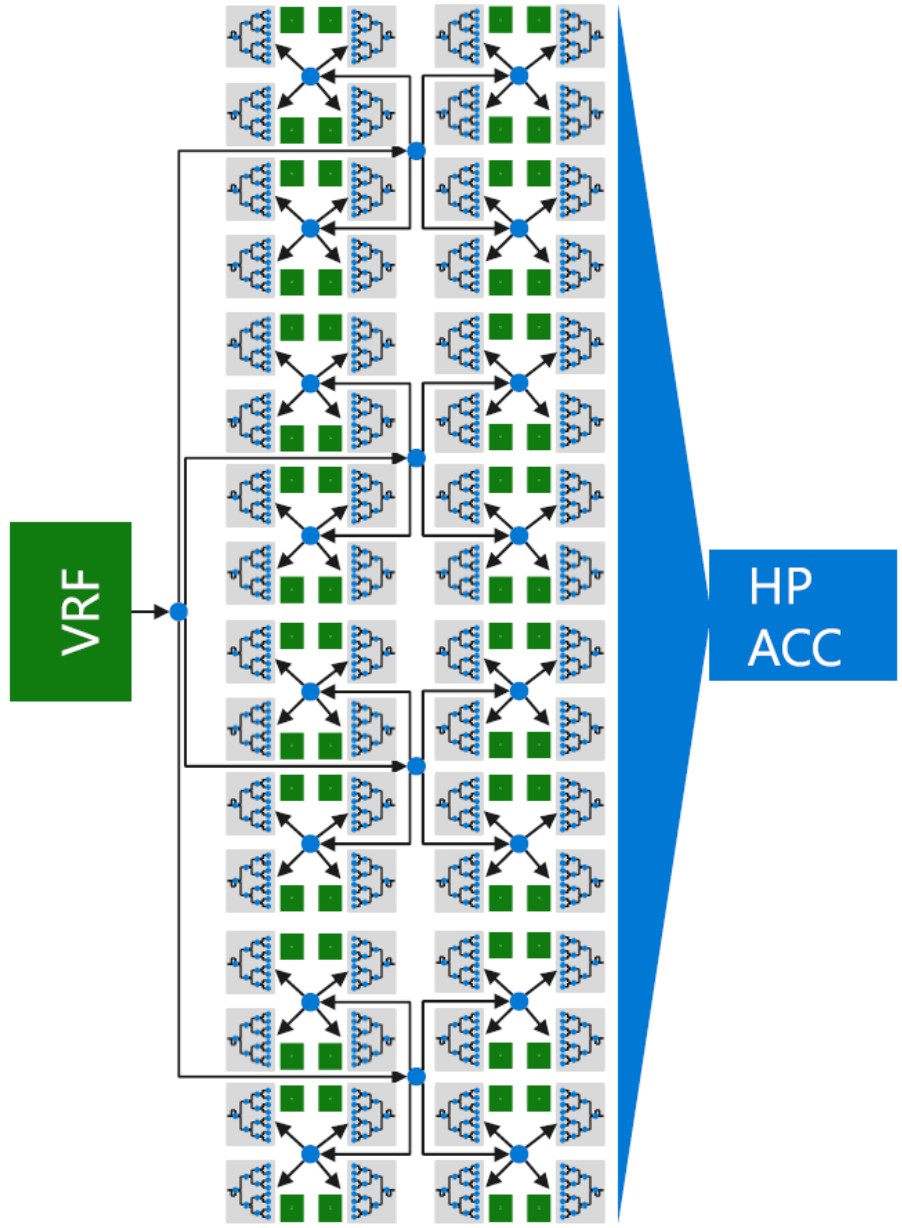




Scaling M*V: Scalable Replication

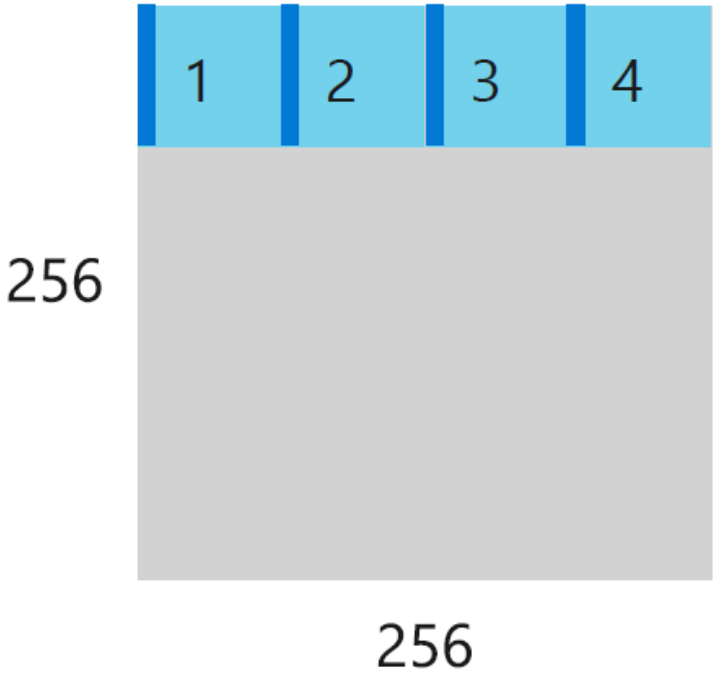
- Tile engine: native size M*V tile (16 x 4 DPEs)
  - High precision accumulator (HP ACC)
  - Registered input fan-out tree
  - Result fan-in tree (muxs)

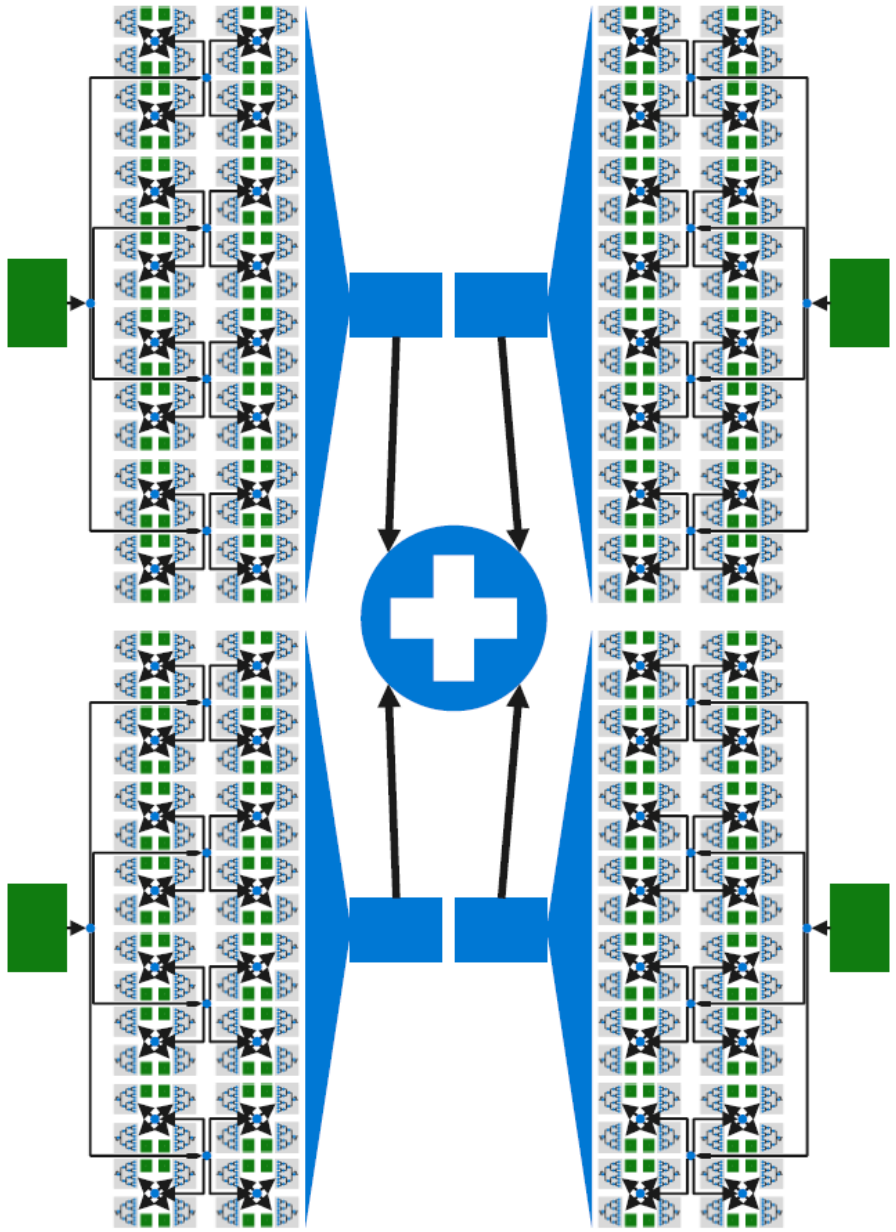




Scaling M*V: Tiling

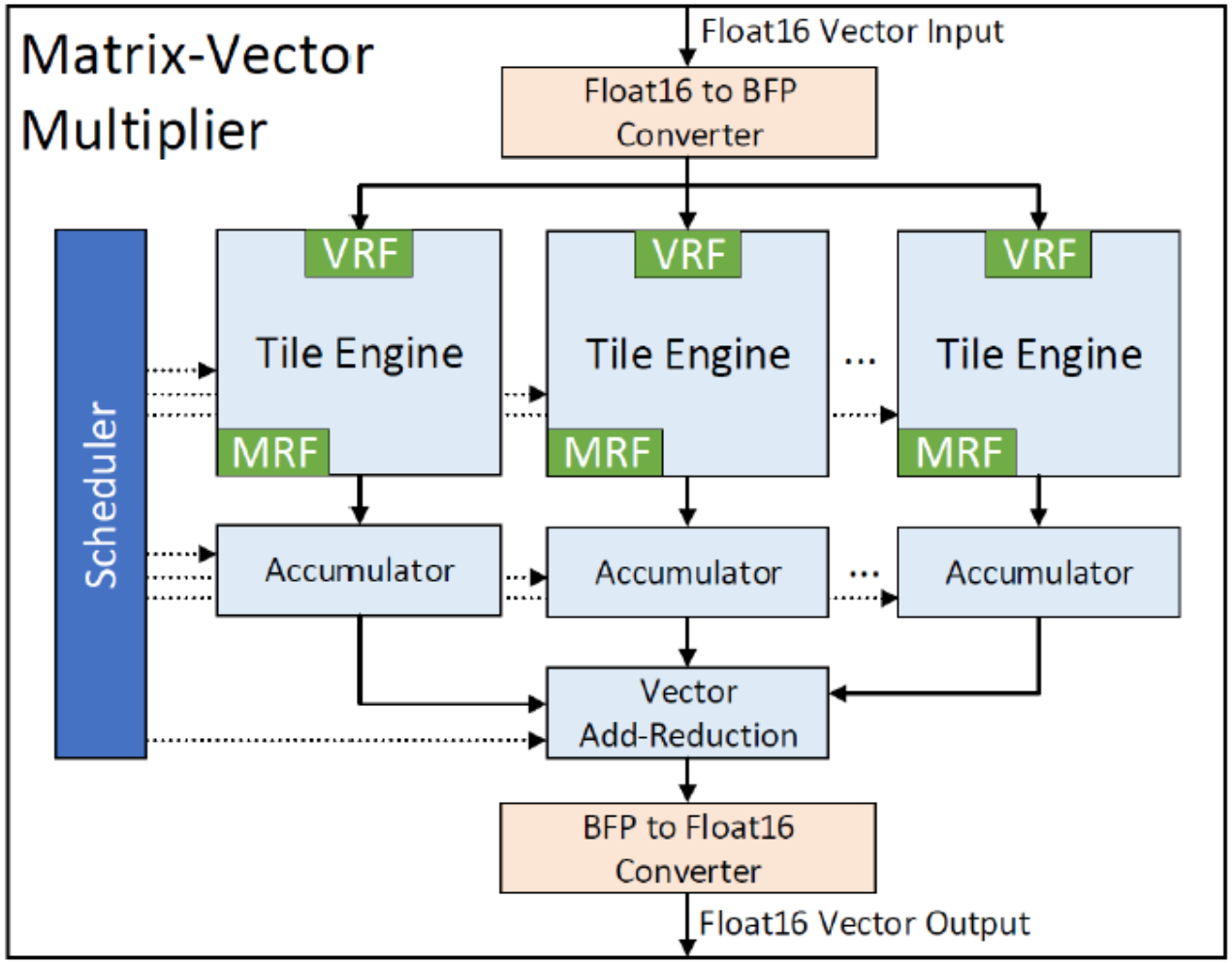
- Large matrices: tile parallelism (4 x 64 DPEs)
- Add-reduction unit
- Scaling limit: area, matrix columns





Scaling M*V: Narrow Precision Data Types

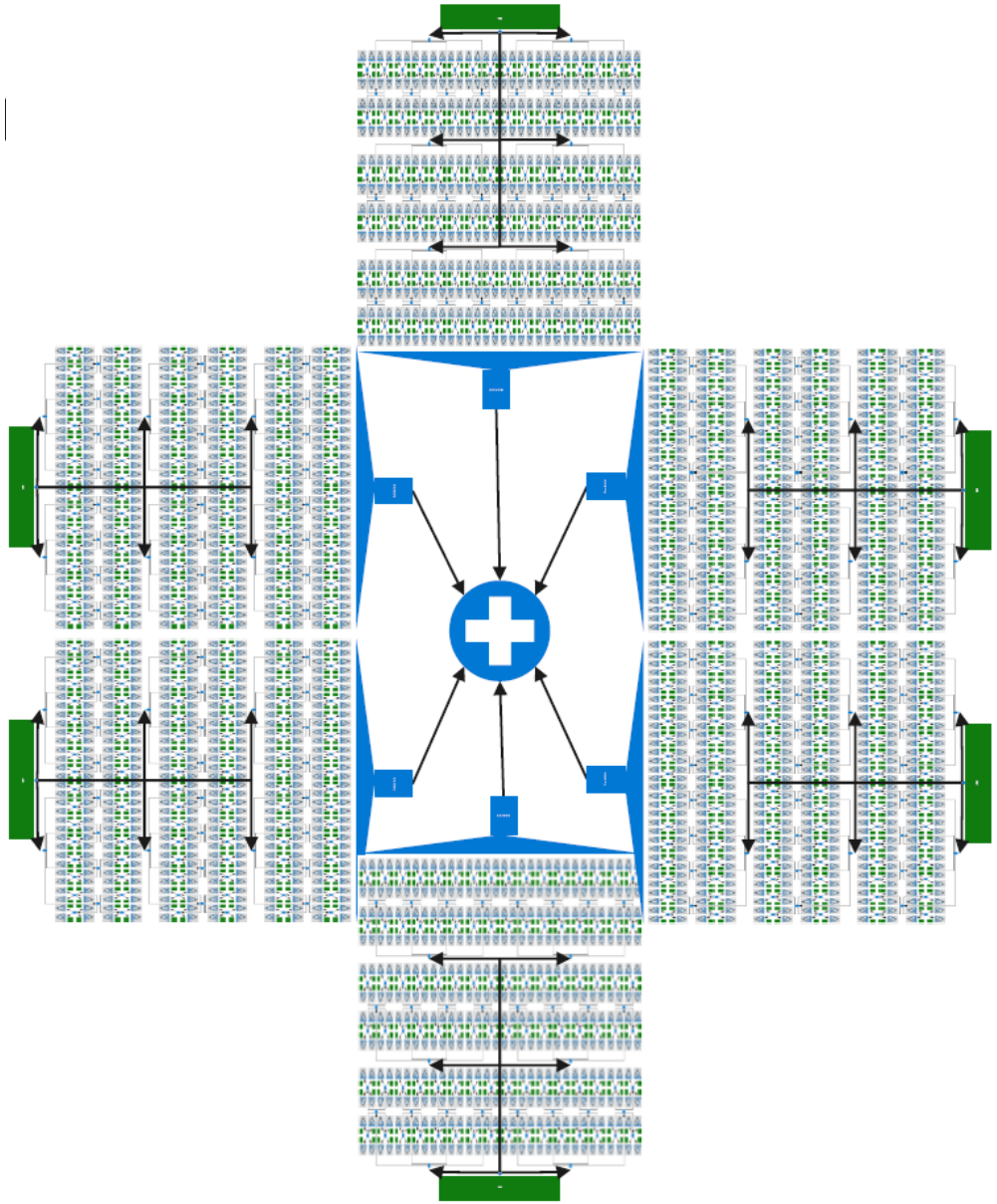
- FP8 – FP11 is sufficient
  - FP8: 1-bit sign, 5-bit exponent, 2-bit mantissa
- Block floating point (BFP)
  - Shared exponent for native vectors
  - Integer arithmetic in tile engines



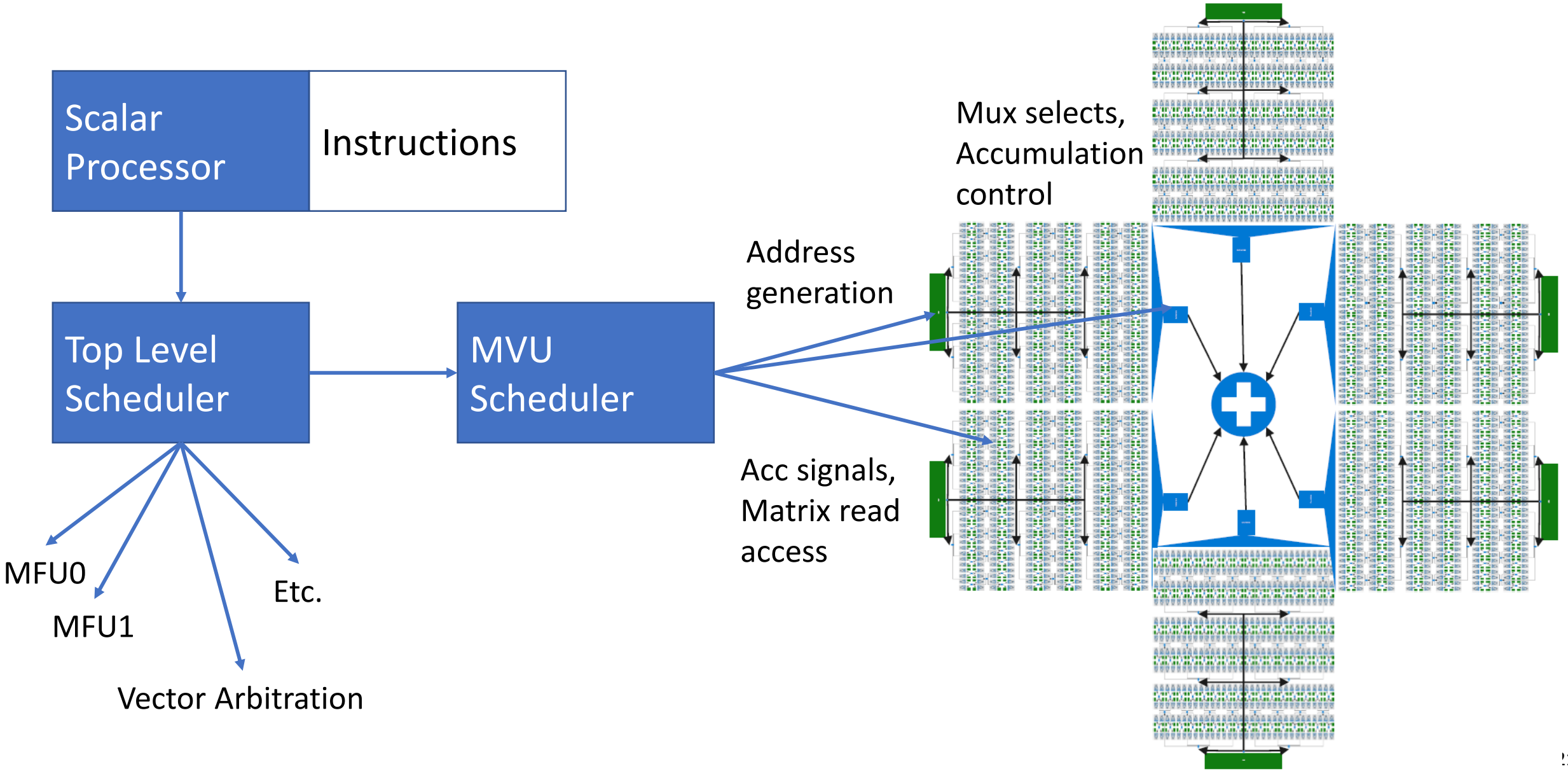
Scaling M*V: Putting All Together

- Fully scaled Block FP8 on Stratix 10
  - 6 Tile Engines
  - 400 Dot Product Engines / Tile
  - 40 lanes per DPE
  - Total: 6 x 400 x 40 =96,000 MACs

- MRF = 96,000 words / cycle

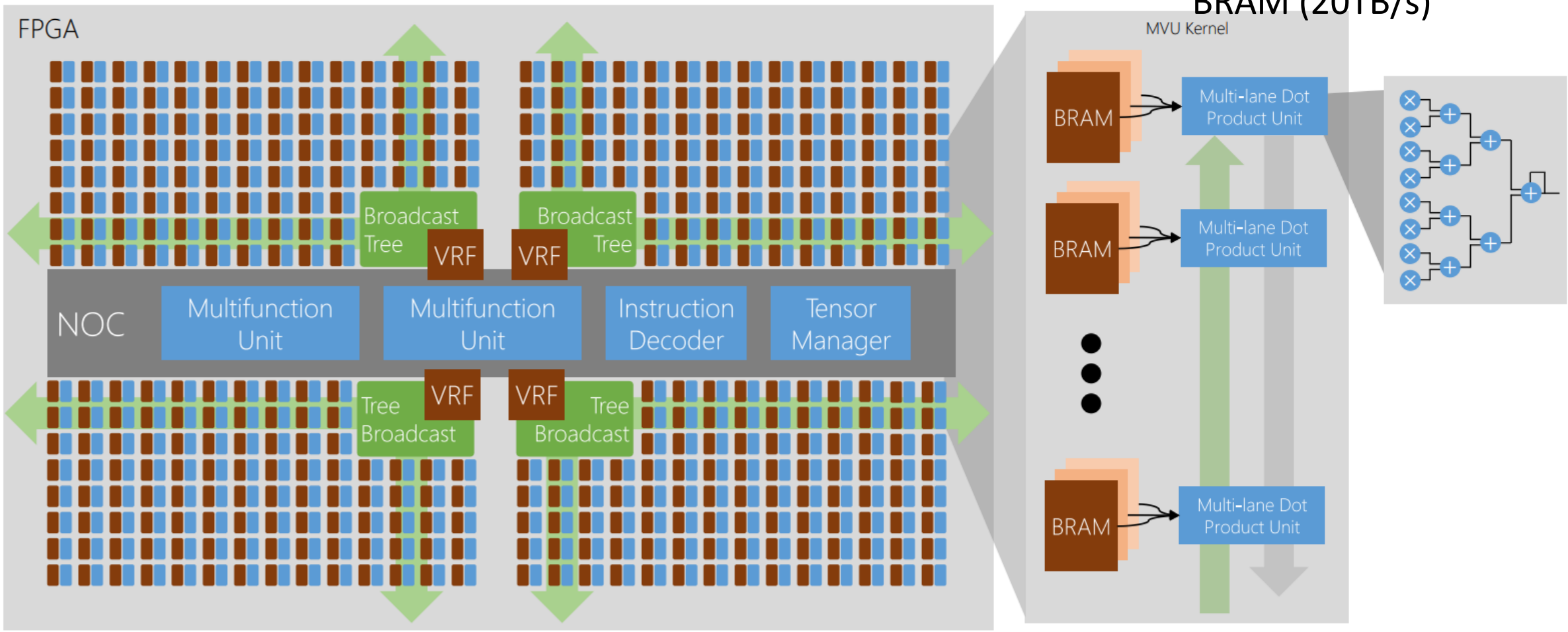


Scheduling for MVU



Spatial View

Matrices distributed row-wise across ~10K banks of BRAM (20TB/s)



Real-Time AI Evaluation

- DeepBench RNN: GRU-2816, batch=1, 71B ops/serve

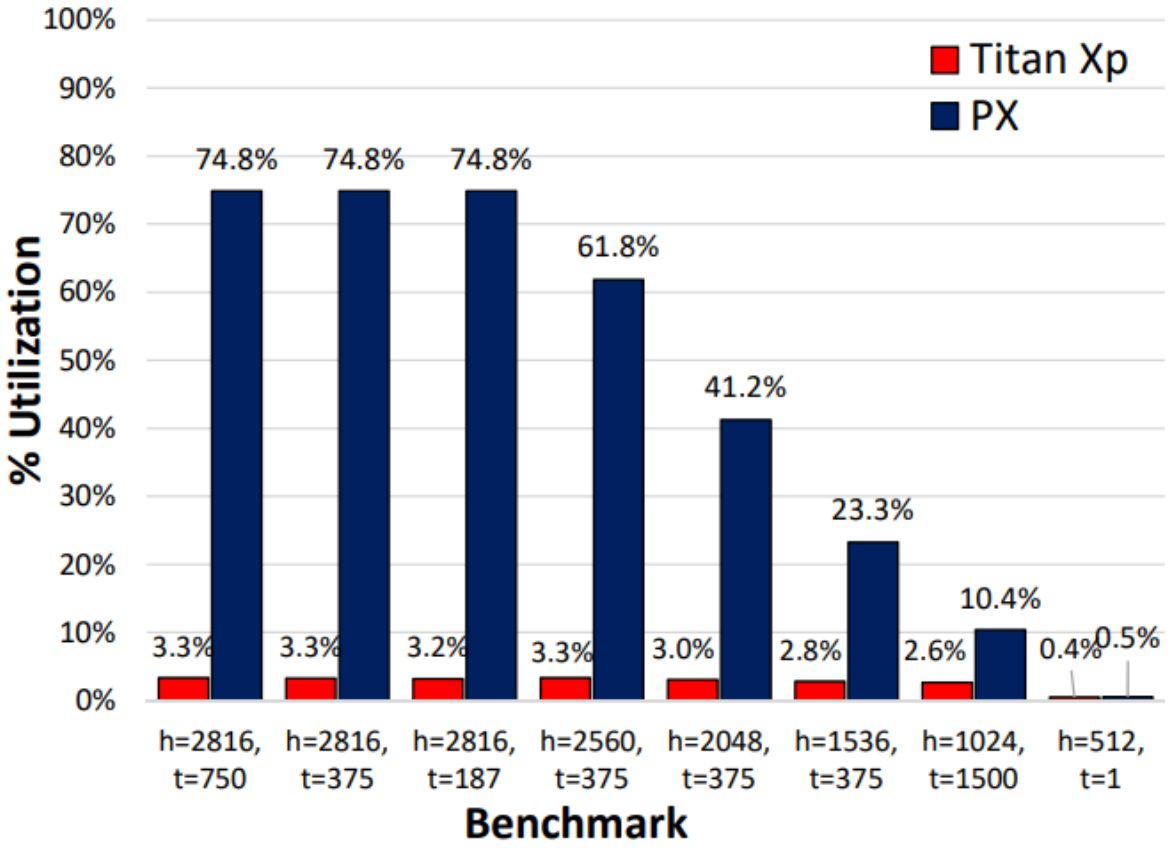
Device	Node	Latency	Effective TFLOPS	Utilization
Stratix 10 280, FP8 (250MHz)	Intel 14nm	2ms	35.9	74.8%

- CNN: ResNet-50, batch=1, 7.7 ops/serve

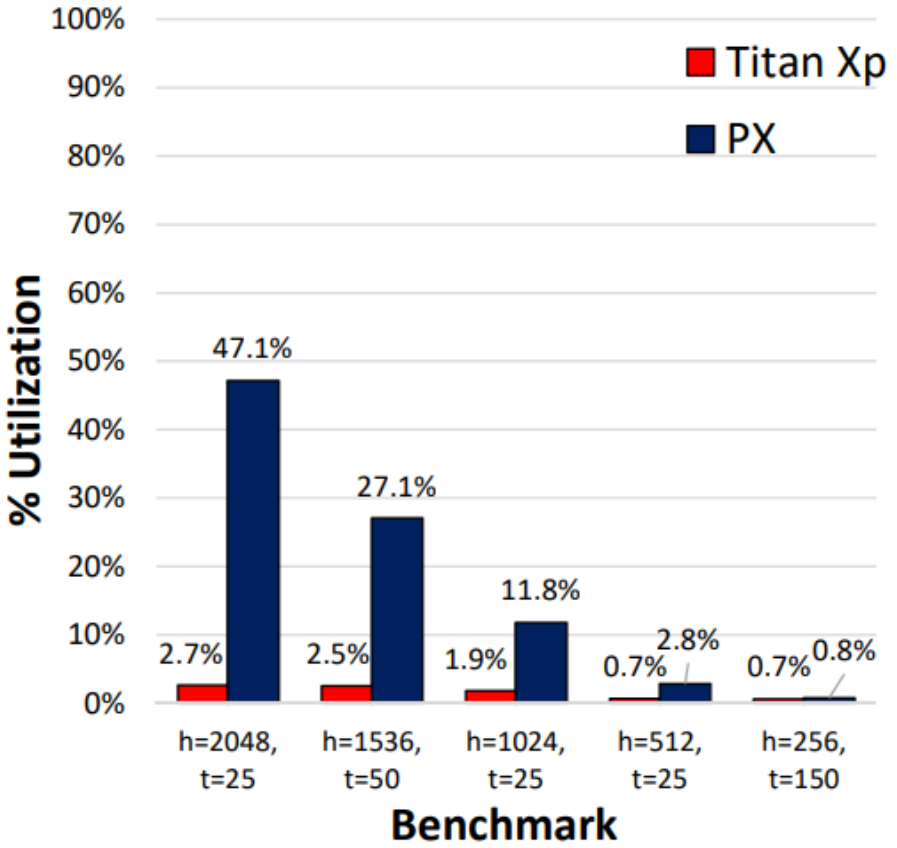
Device	Node	Latency	Effective TFLOPS	Utilization
Arria 10 1150, FP11 (300MHz)	TSMC 20nm	1.64ms	4.7	66%

Comparison to Nvidia P40

- Nvidia P40 (16nm TSMC) vs BW_A10 (20nm TSMC) on DeepBench RNN inference



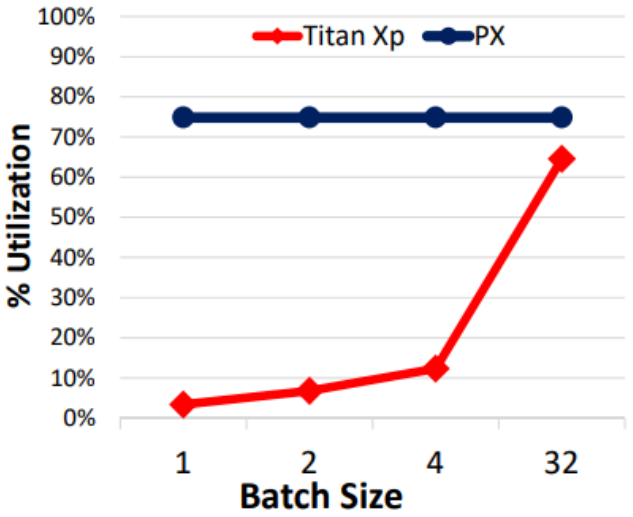
(a) GRU



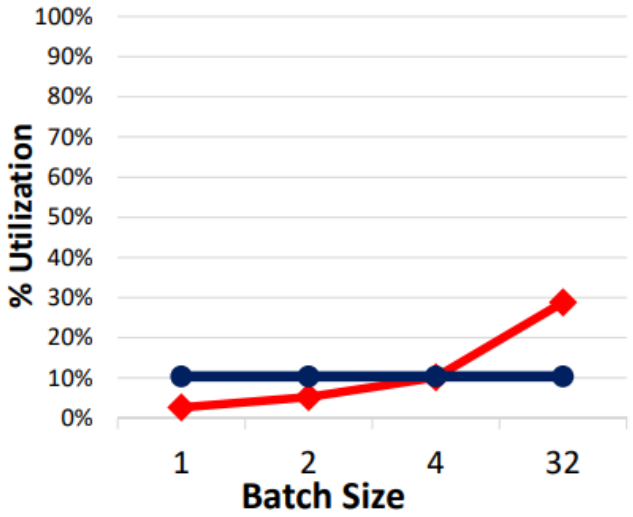
(b) LSTM

Comparison to Nvidia P40

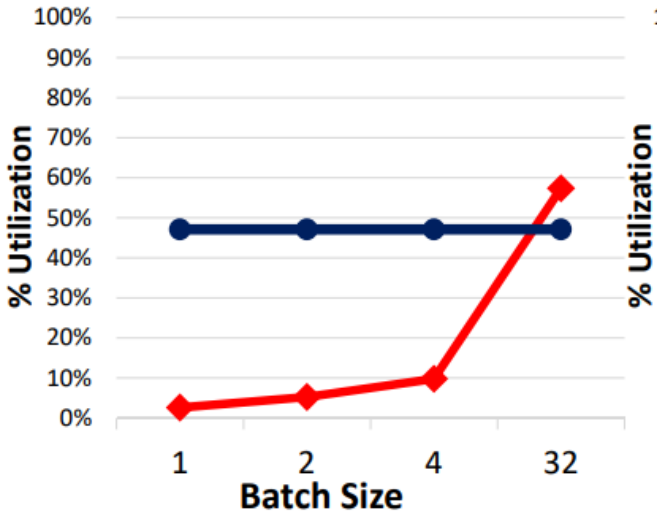
- Utilization scaling with increasing batch sizes



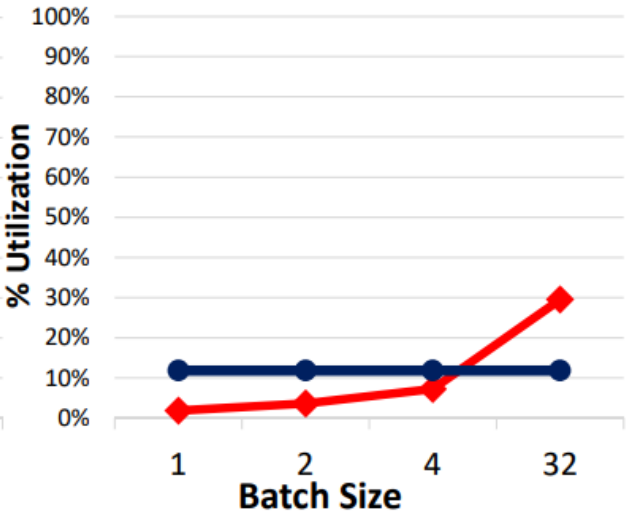
(a) GRU-2816



(b) GRU-1024

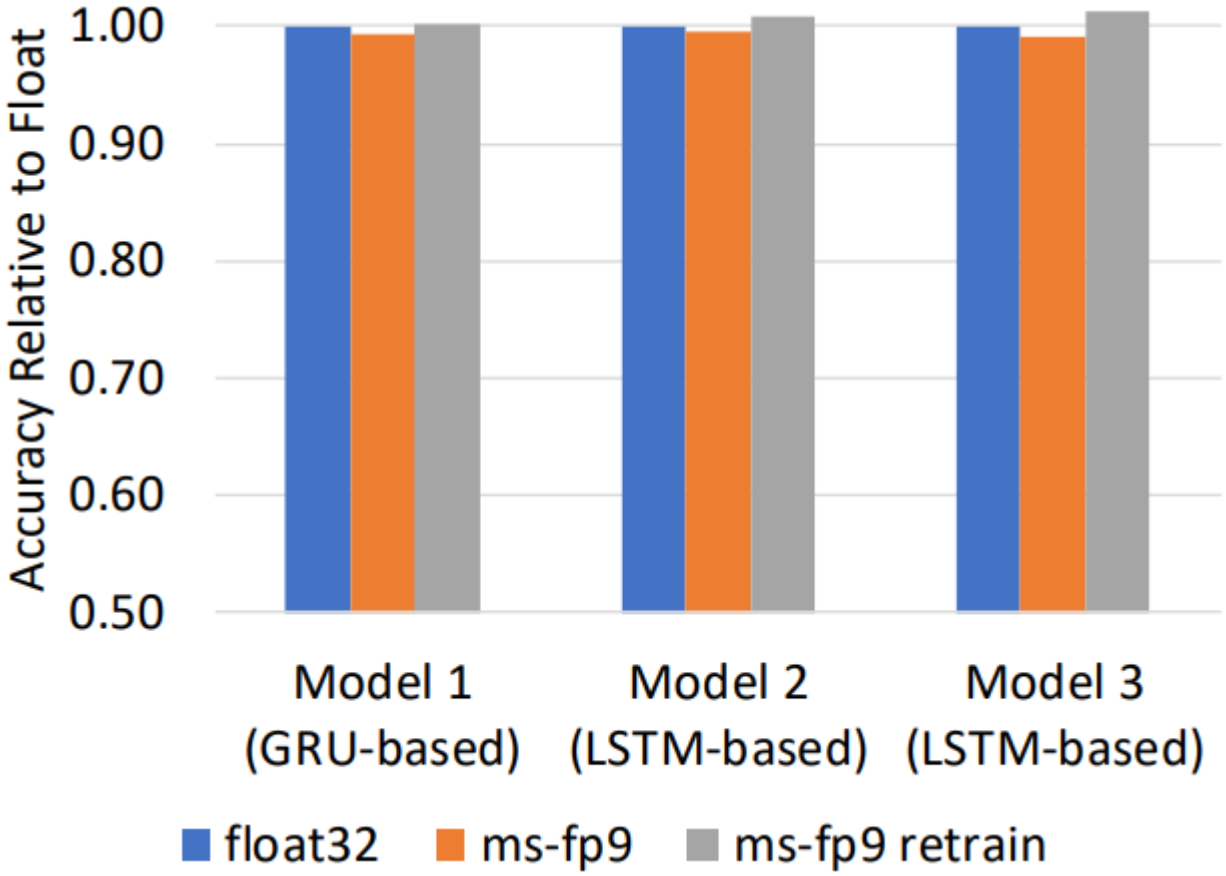


(c) LSTM-2048



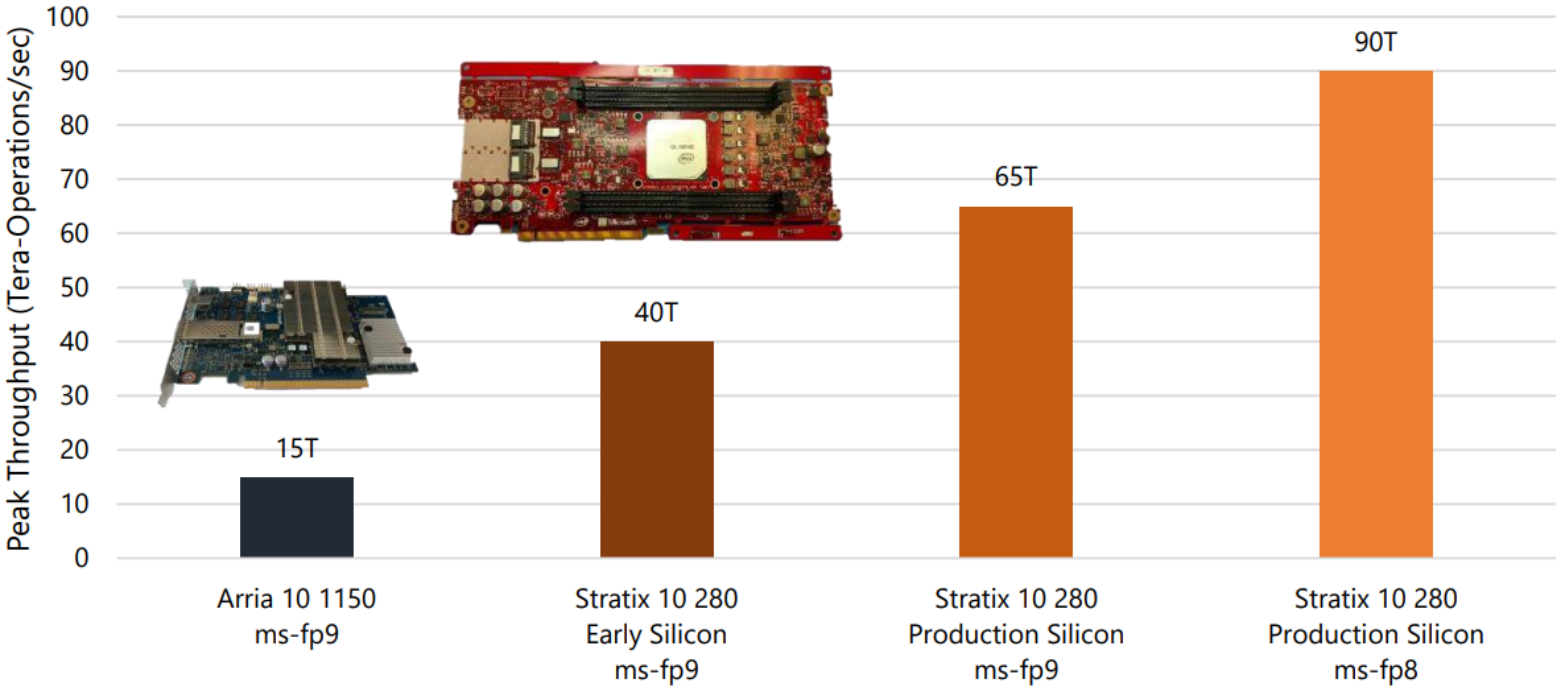
(d) LSTM-1024

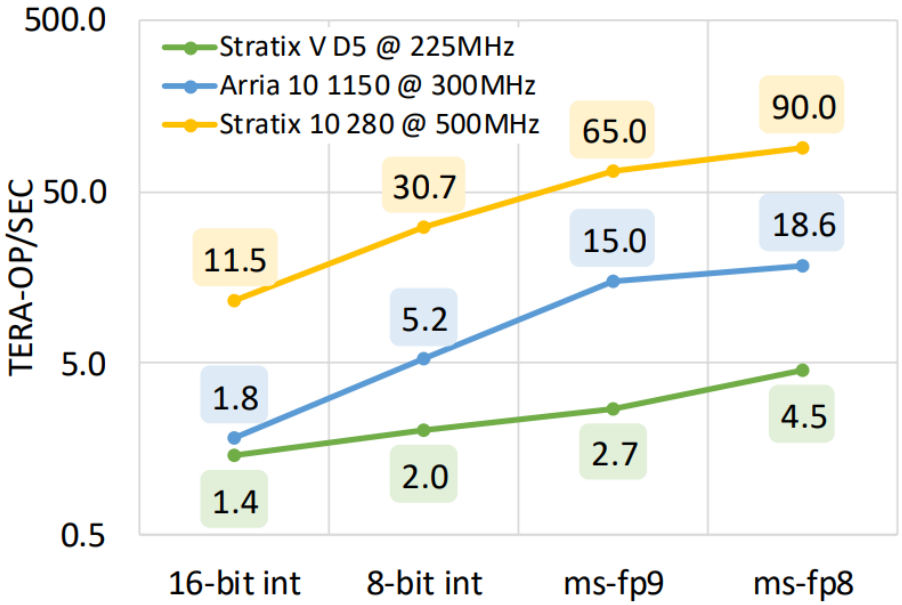
Accuracy Impact of Narrow Precision



- Measured on ResNet-50 model
- ms-fp9: proprietary 9-bit float-point data formats (mantissa trimmed to 3-bits)

Performance Scaling

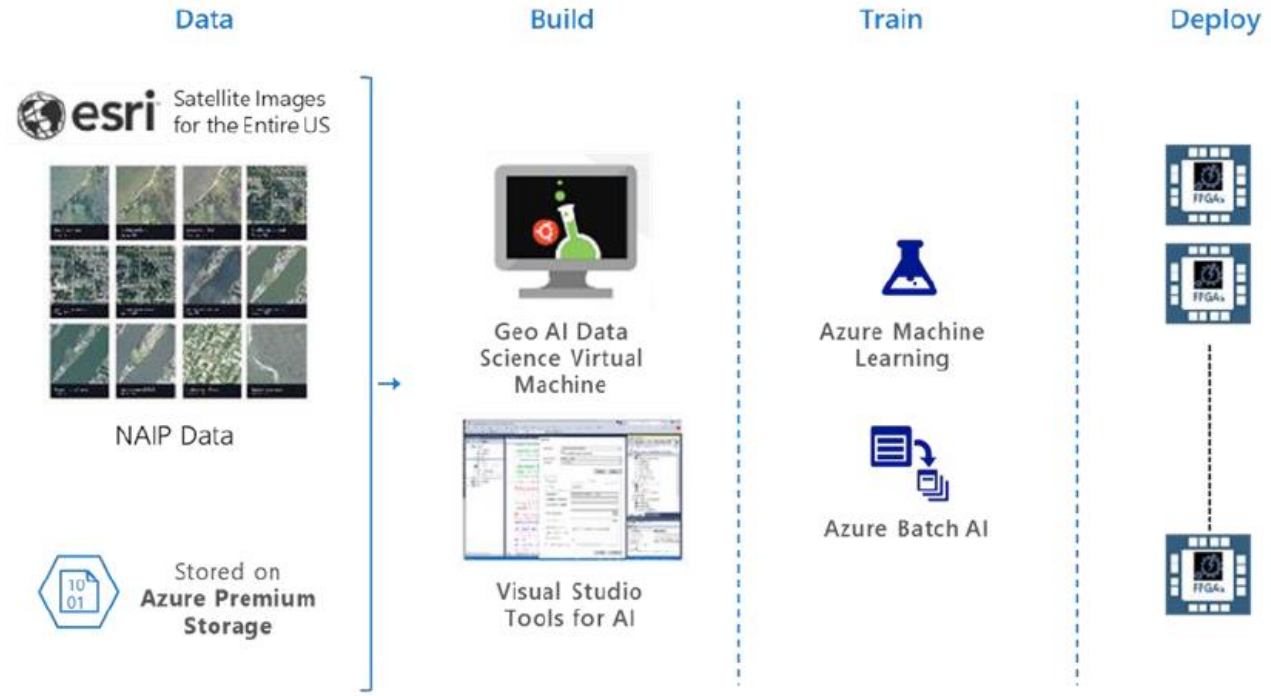




Arria 10 1150 (20nm)	Stratix 10 280 Early Silicon (14nm)
ms-fp9	ms-fp9
316K ALMs (74%)	858K ALMs (92%)
1442 DSPs (95%)	5,760 DSPs (100%)
2,564 M20Ks (95%)	8,151 M20Ks (70%)
160 GOPS/W	320 GOPS/W → 720 GOPS/W (production)

Production

- BrainWave NPU is in scale production at Microsoft
  - Powering Microsoft services such as Bing search
  - Serving real-time CNNs to Azure customers



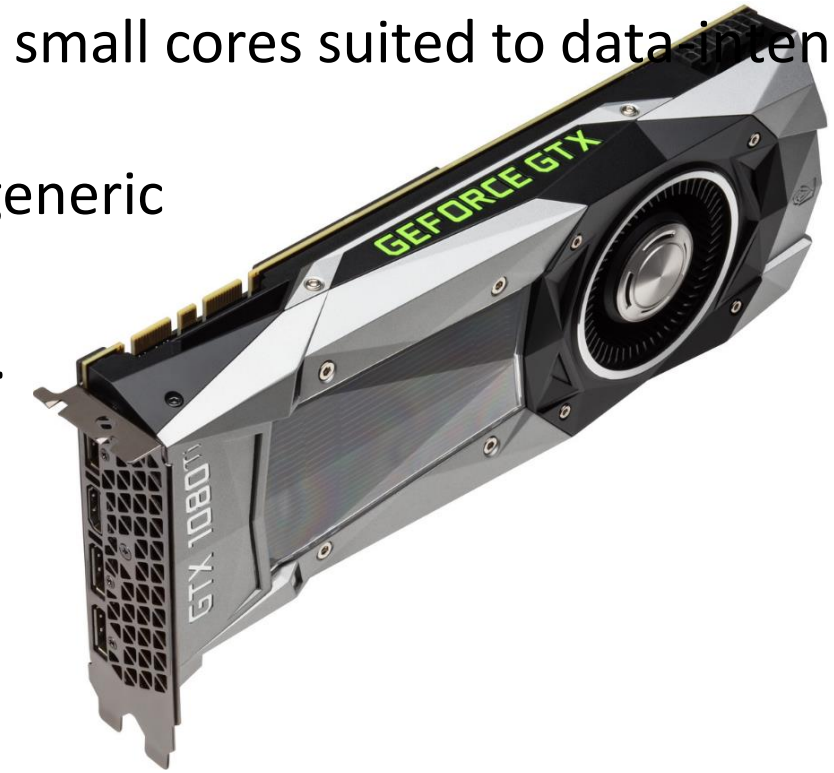
Discussion

- How are AI accelerators for cloud datacenters different from AI accelerators for mobile domain?
- Why is systolic execution energy efficient?
- Array processor vs Systolic processor vs Vector processor
- Why do big companies focus on inference first? Do you know if there are training accelerators for cloud datacenters except GPUs?

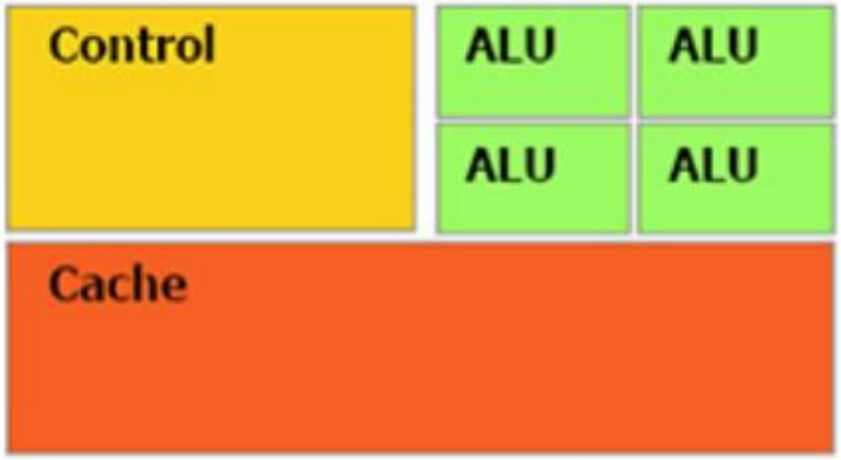
GPU Basics

- Graphics Processing Unit
  - Primarily designed for faster 3D graphics processing with fixed pipelines

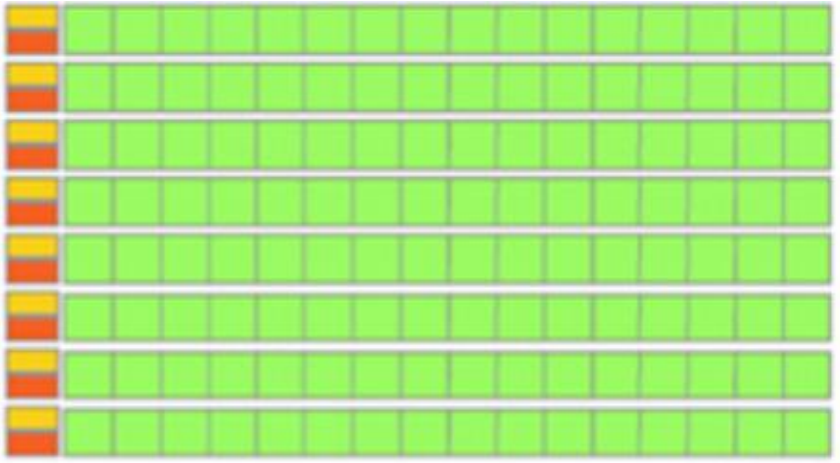
    Vertex shader, pixel shader, geometry shader, rasterizer, …
  - Rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer to display on a screen
  - Massively parallel architecture with lots of small cores suited to data-intensive applications
  - With unified shaders, GPU became more generic
  - OpenGL, CUDA programming language
  - Applications: gaming, ML, crypto mining, ..



CPU vs GPU



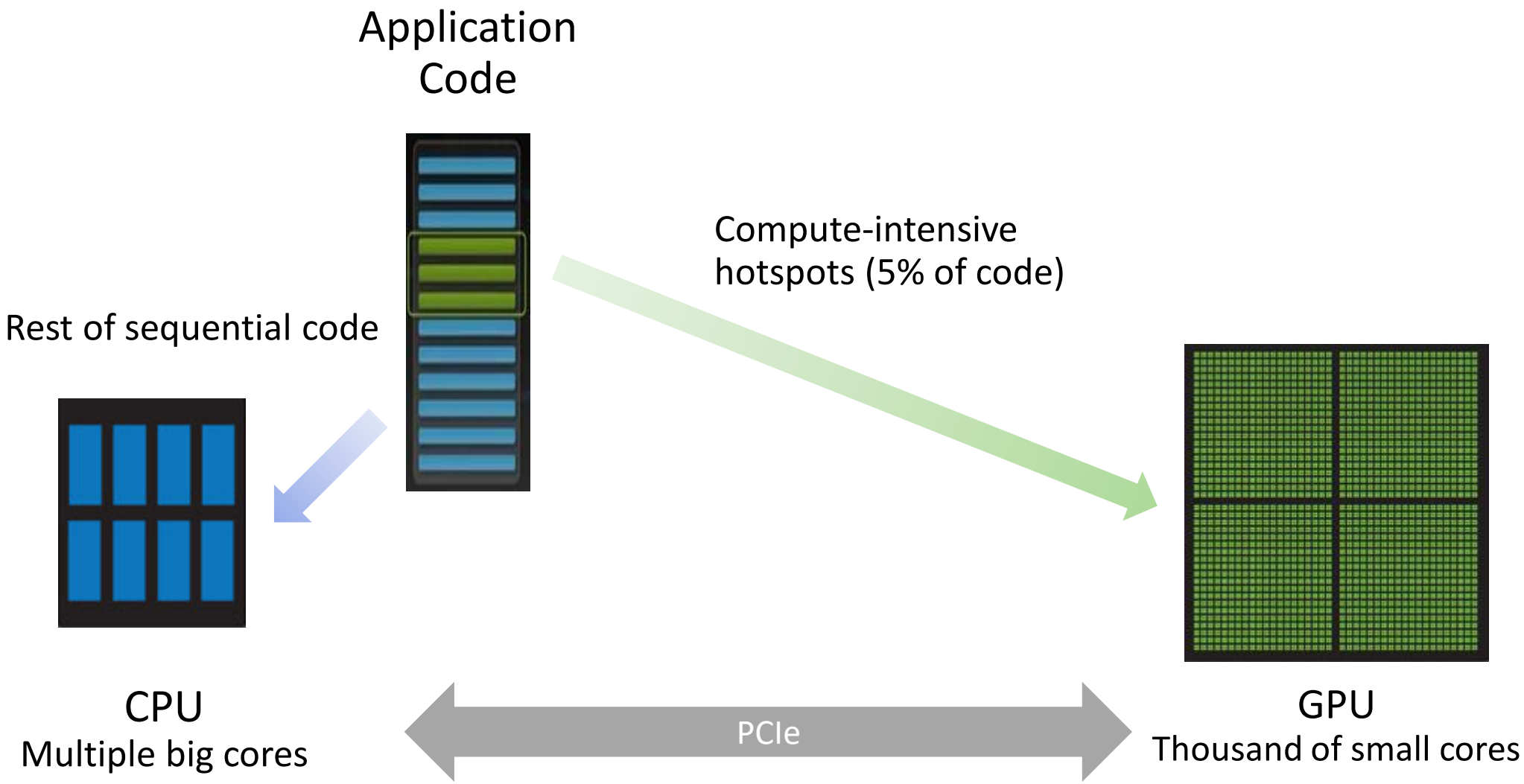
- Low compute density
- Complex control (out of order)
- Optimized for serial processing
- Few ALUs
- High clock speed
- Low latency tolerance
- Good for task parallelism



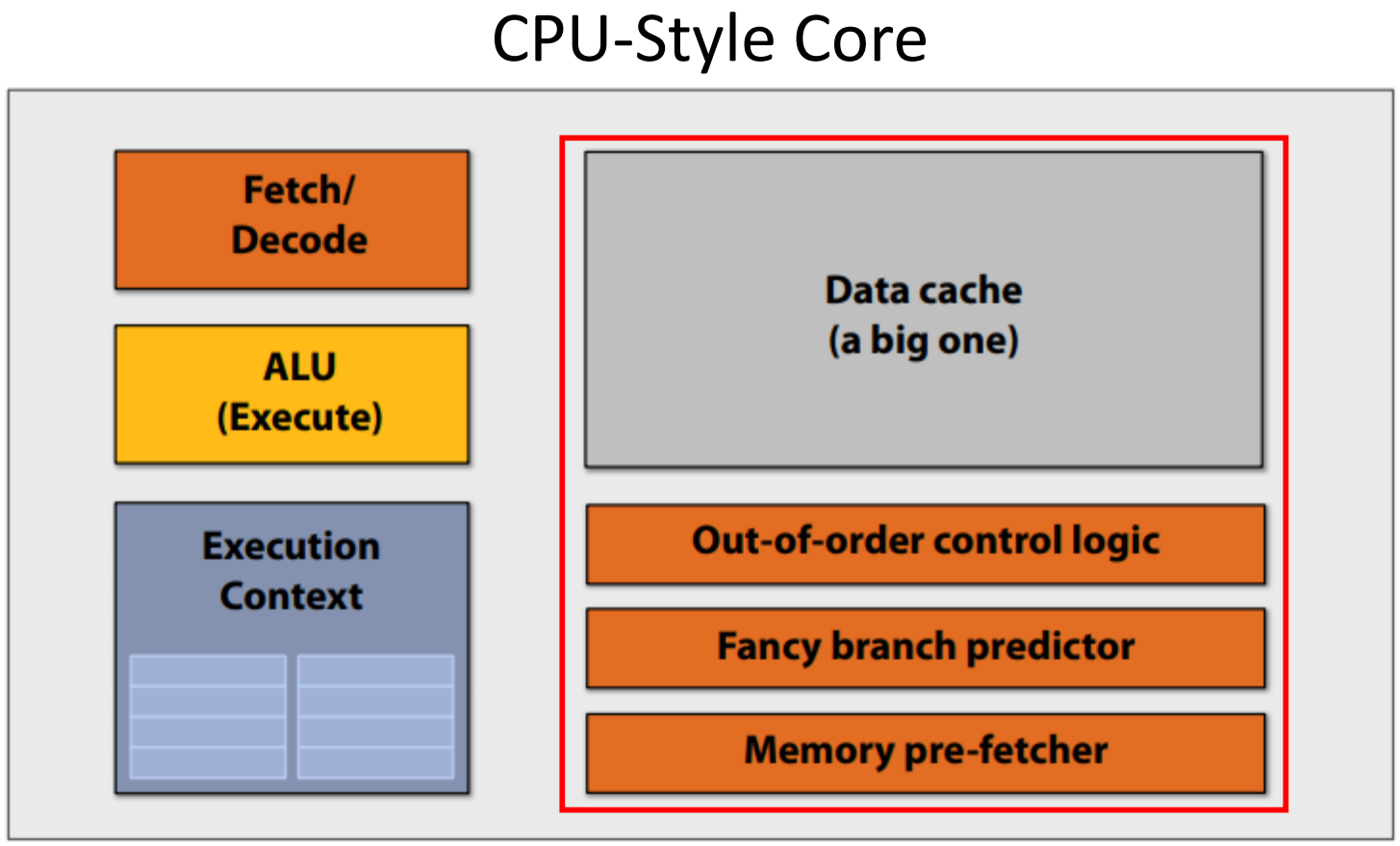
- High compute density
- High computations per memory access
- Designed for parallel processing
- Many parallel small cores
- High throughput
- High latency tolerance
- Good for data parallelism

GPU dedicates more transistors to ALU than flow control and data cache!

How GPU Acceleration Works



From CPU to GPU



Out-of-Order + Cache
= Big overhead!

Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Sliming Down

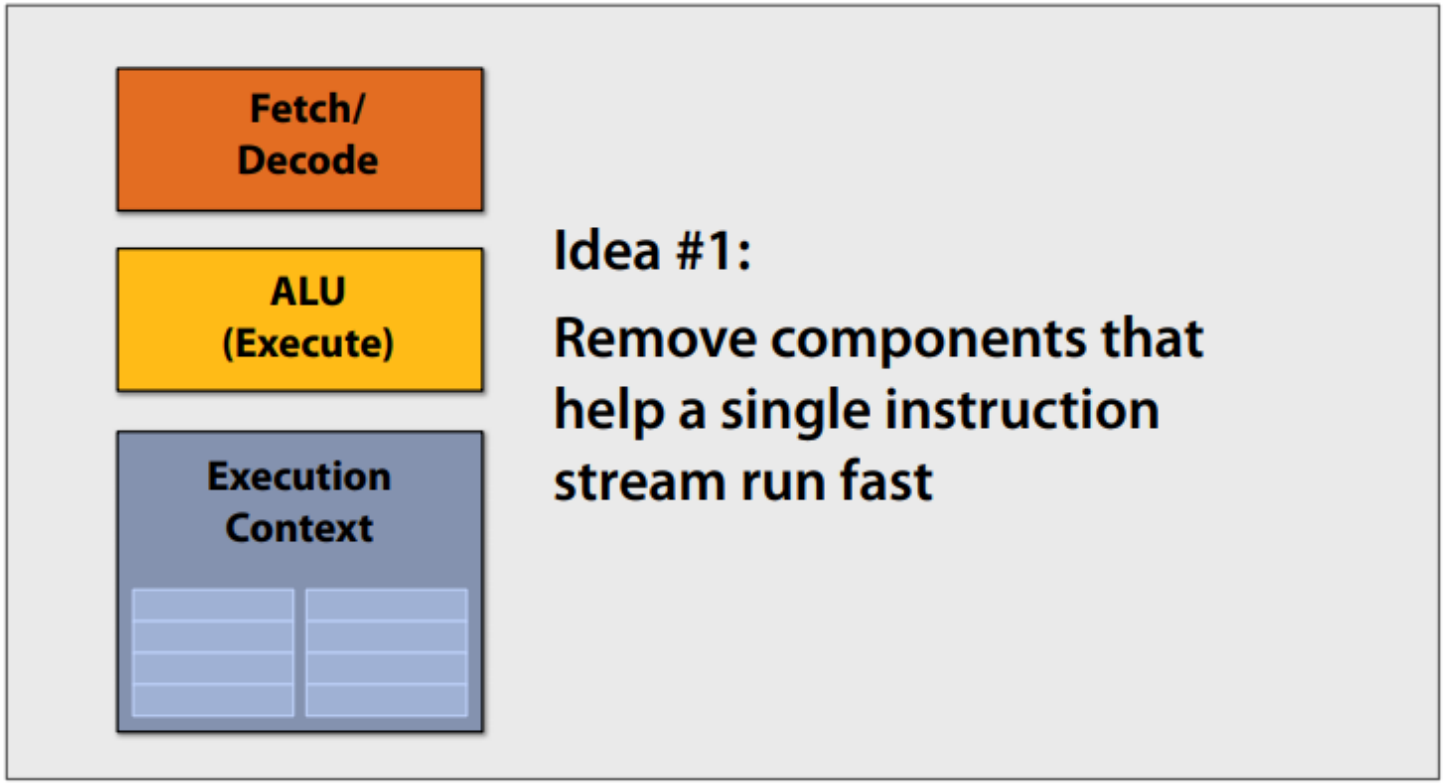


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Two Cores

- Two different code fragments are running in parallel

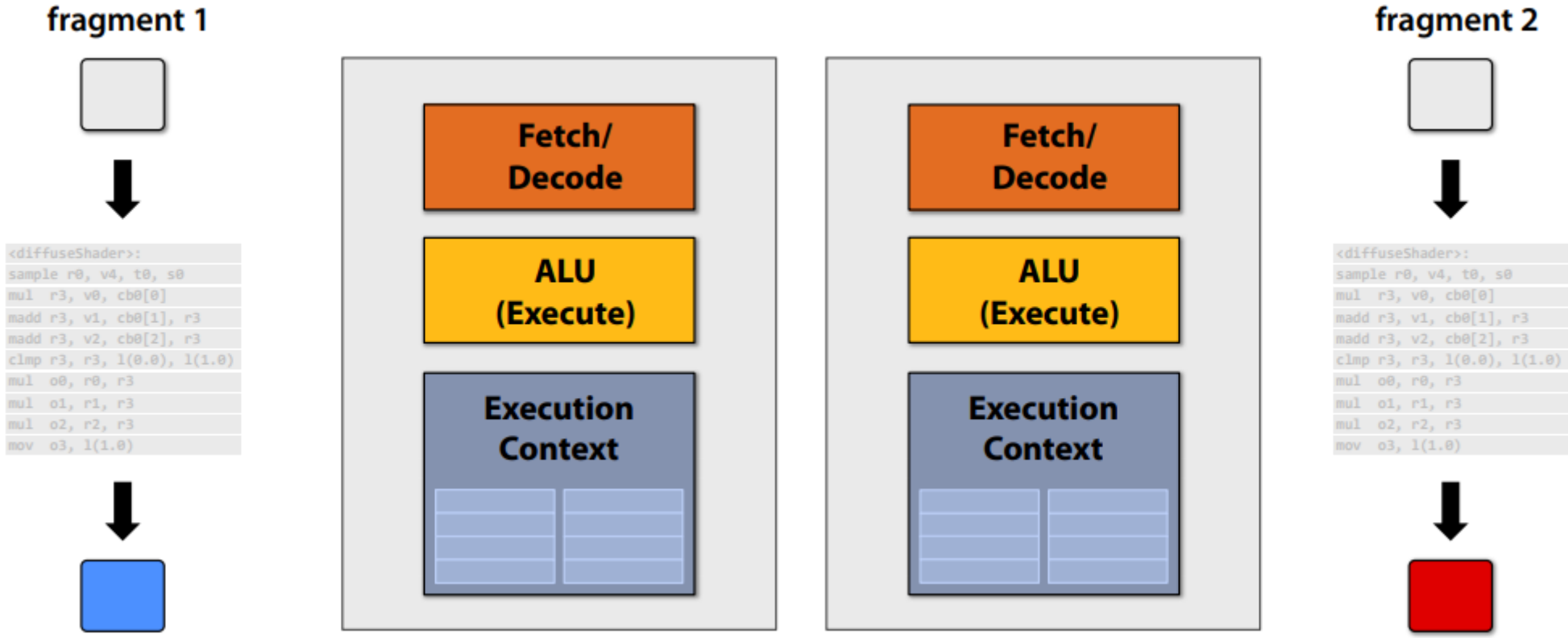
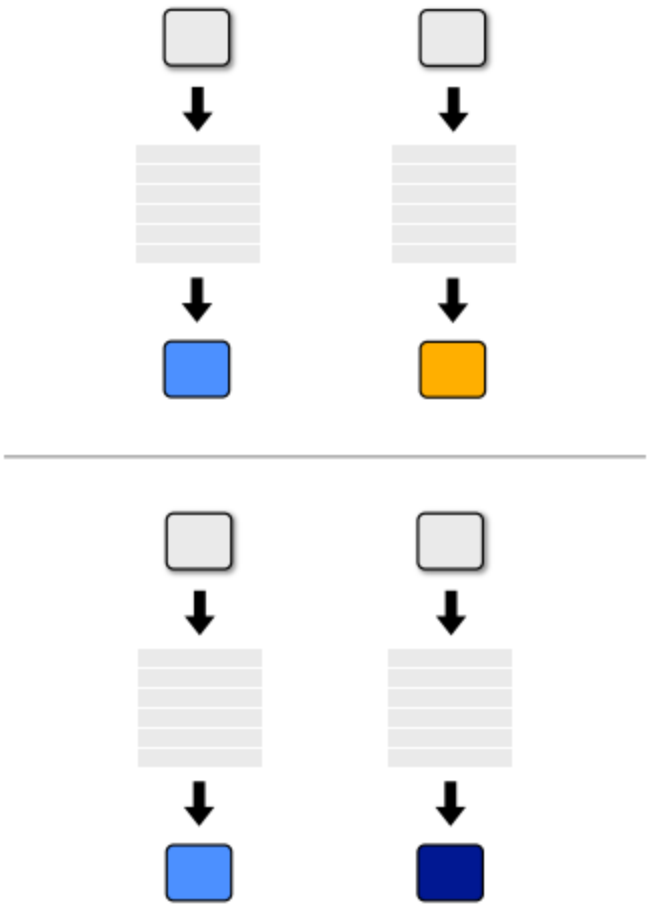


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Four Cores



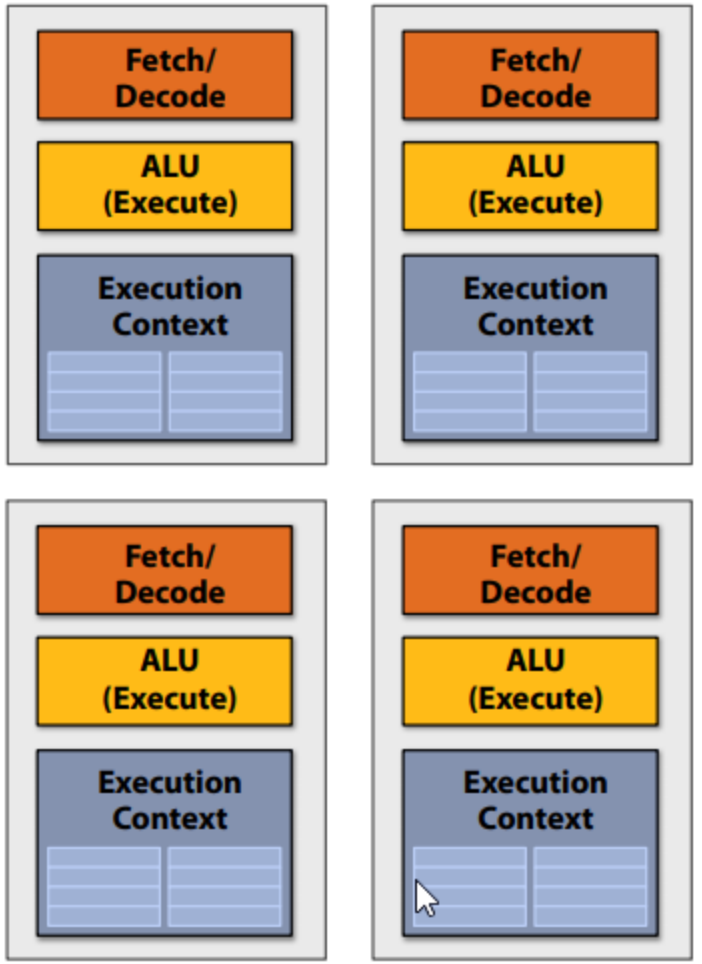
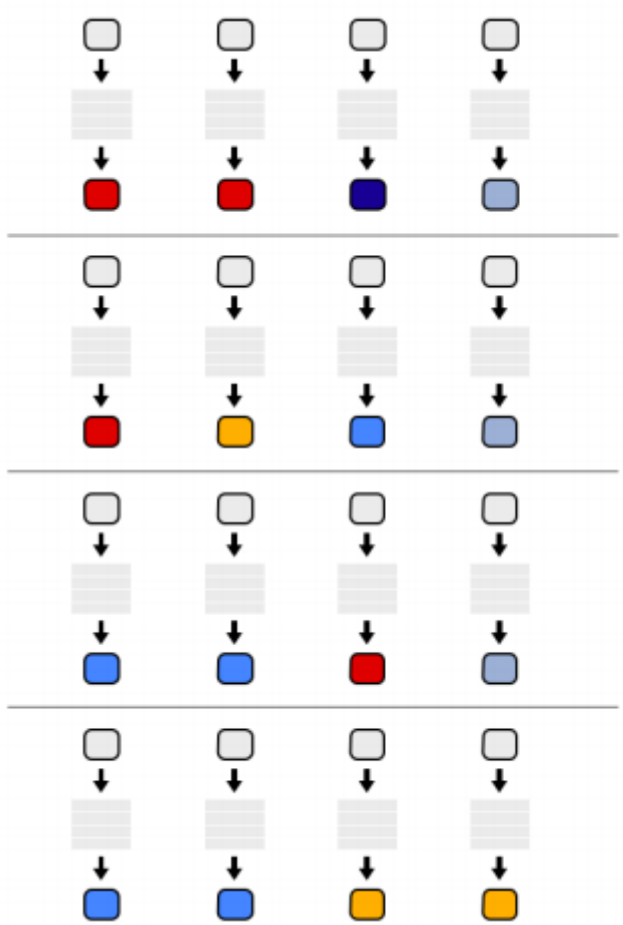
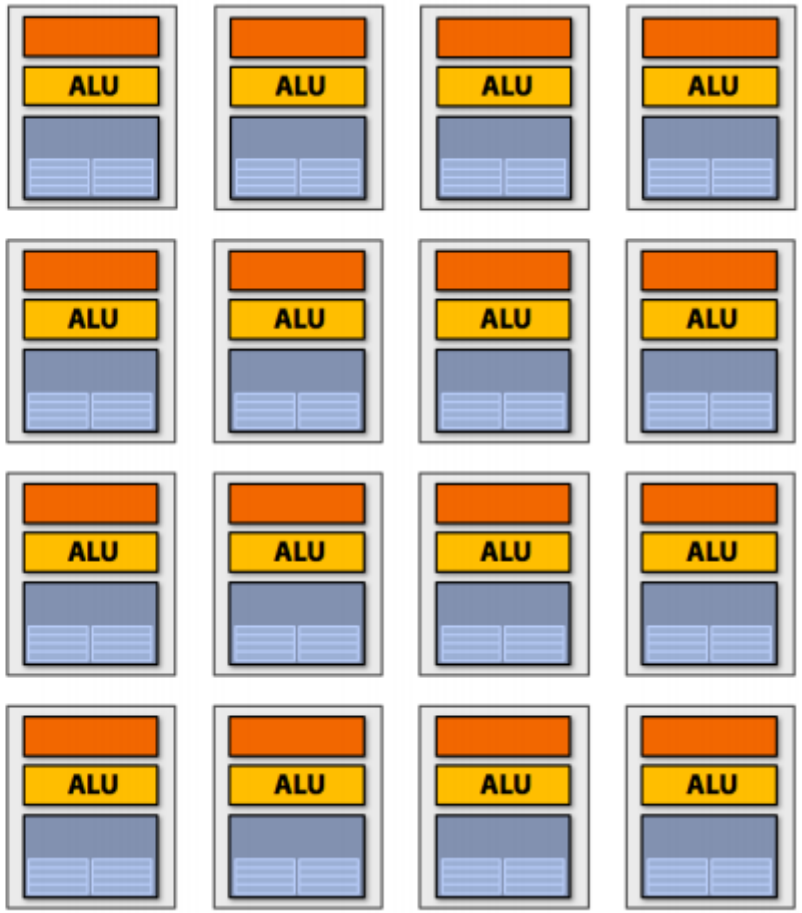


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Sixteen Cores

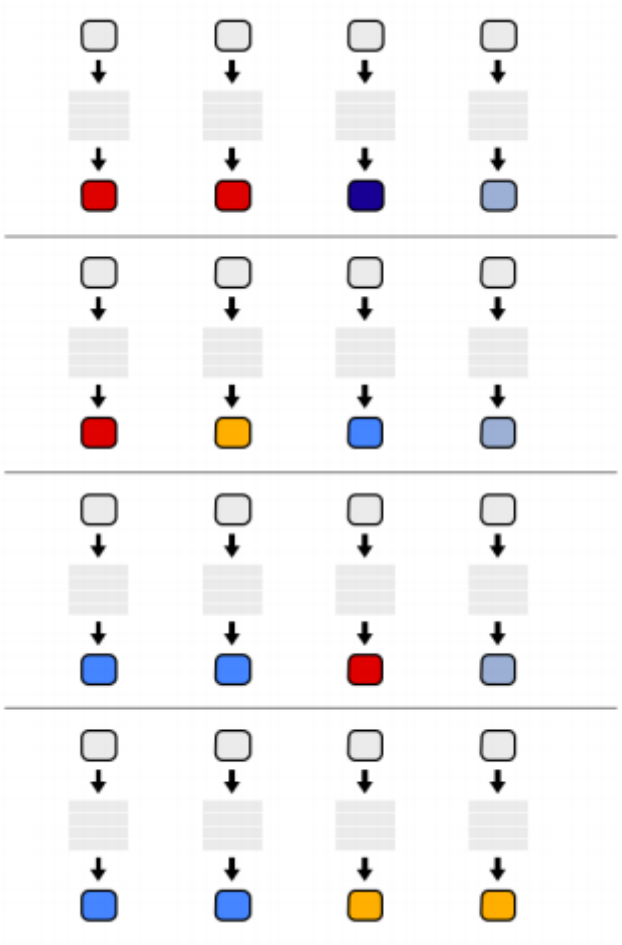




16 cores = 16 simultaneous instruction streams

Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Instruction Stream Sharing



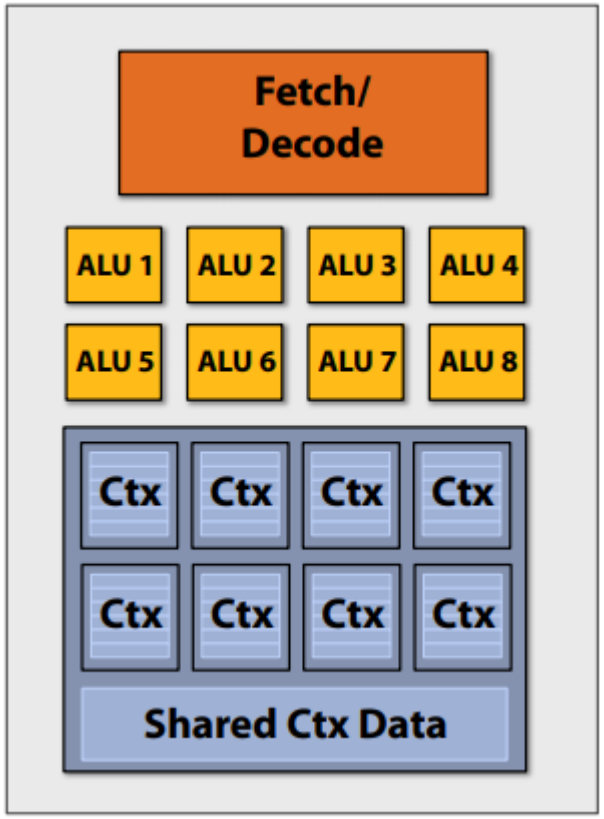
Many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Main idea of SIMT!

Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Add More ALUs



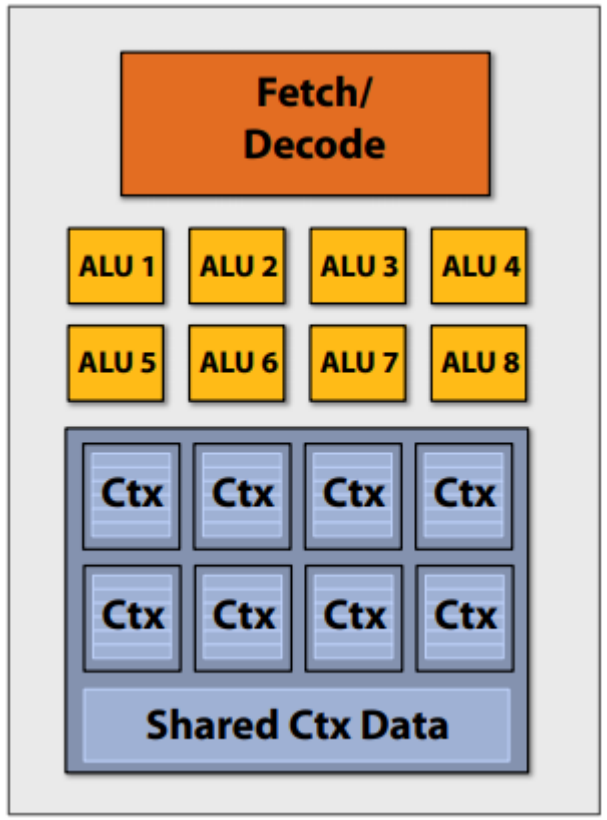
Idea #2:

Amortize cost of managing an instruction stream across many ALUs

→ SIMD processing

Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Modifying Code




```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Scalar operations
Scalar register

```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul    vec_r3, vec_v0, cb0[0]
VEC8_madd   vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd   vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp   vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul    vec_o0, vec_r0, vec_r3
VEC8_mul    vec_o1, vec_r1, vec_r3
VEC8_mul    vec_o2, vec_r2, vec_r3
VEC8_mov    o3, l(1.0)
```

Vector operations
Vector register

Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Scaling Up

128 instruction streams in parallel

16 independent groups of 8 synchronized streams

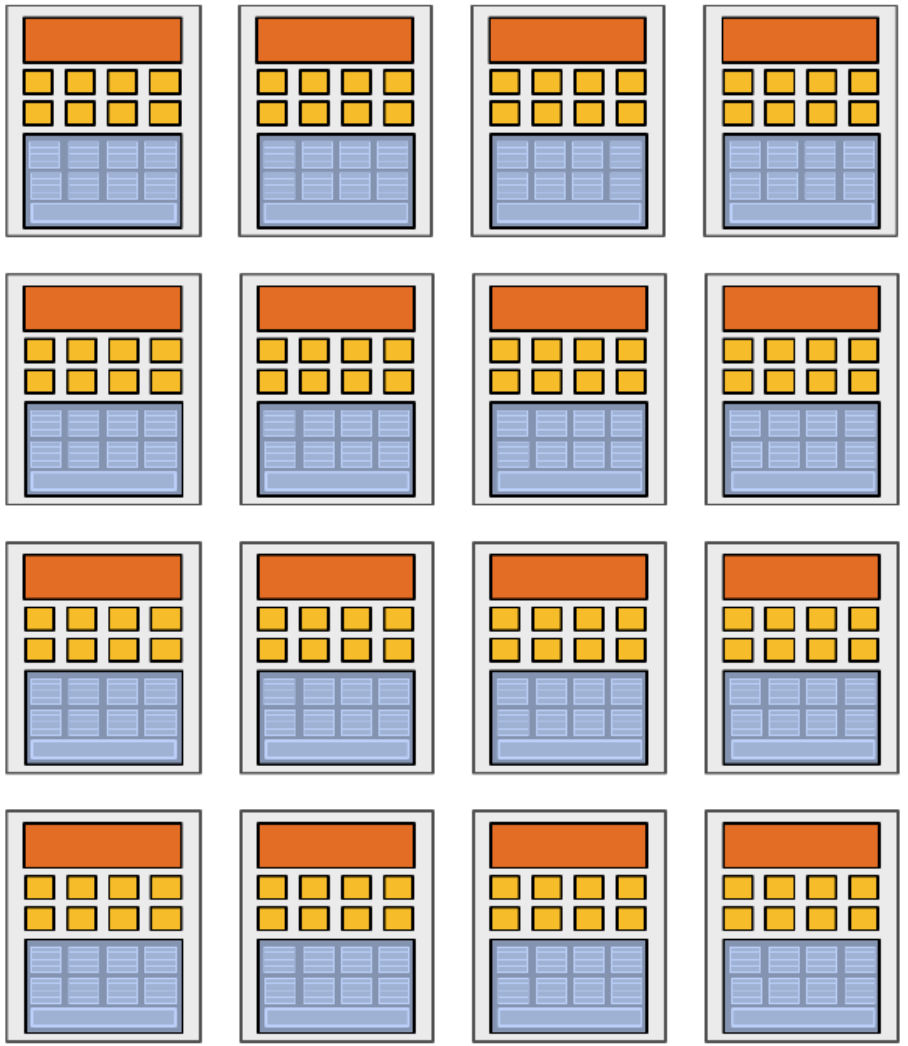


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

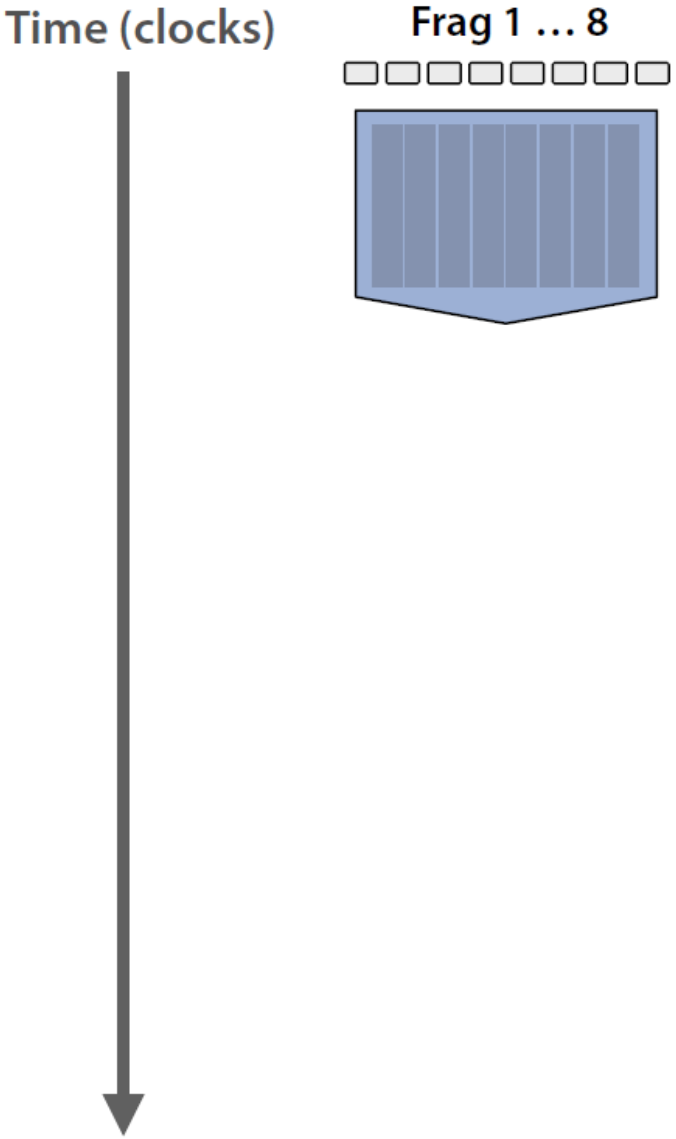
Memory Access Latency

- Stall problem
  - Stalls occur when a core cannot run the next instruction due to dependency
  - Usually memory access latency is long like 100s to 1000s of cycles
  - Fancy cache and out-of-order control logic helped avoid stalls, but removed

Idea #3:

Simultaneous Multi-Threading: Interleave processing of many code fragments through context switching

Hiding Memory Latency



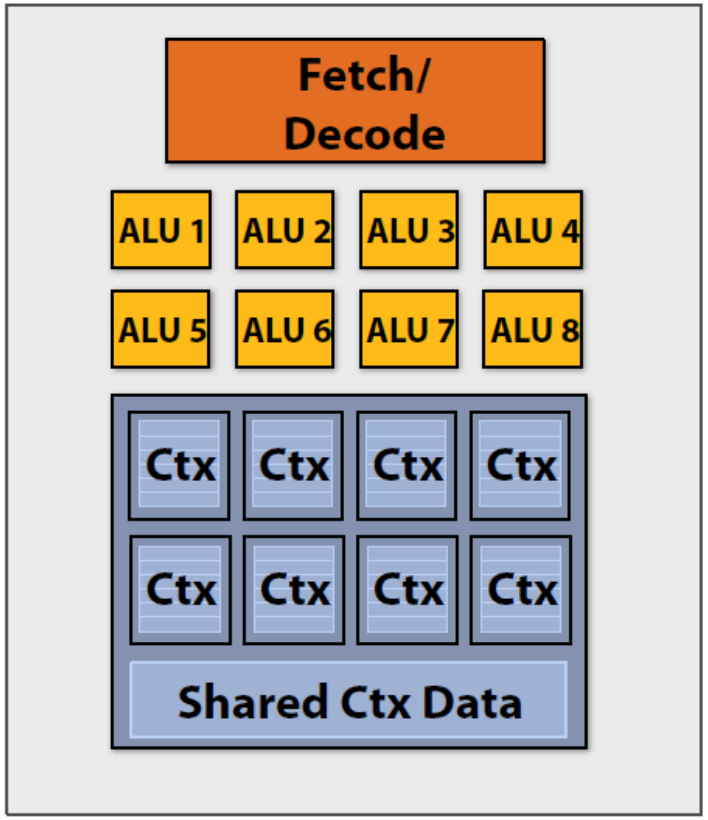


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Hiding Memory Latency

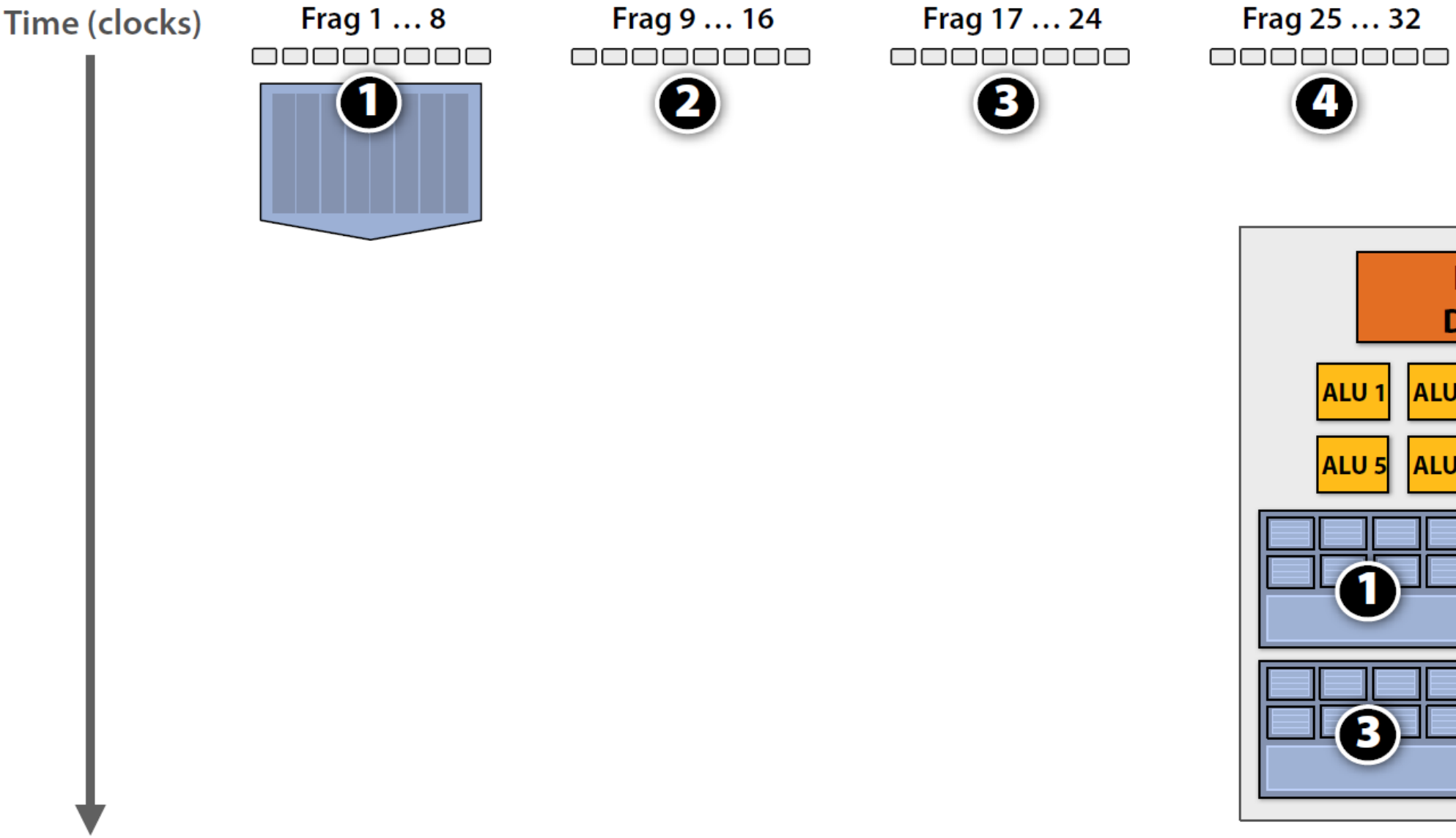


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Hiding Memory Latency

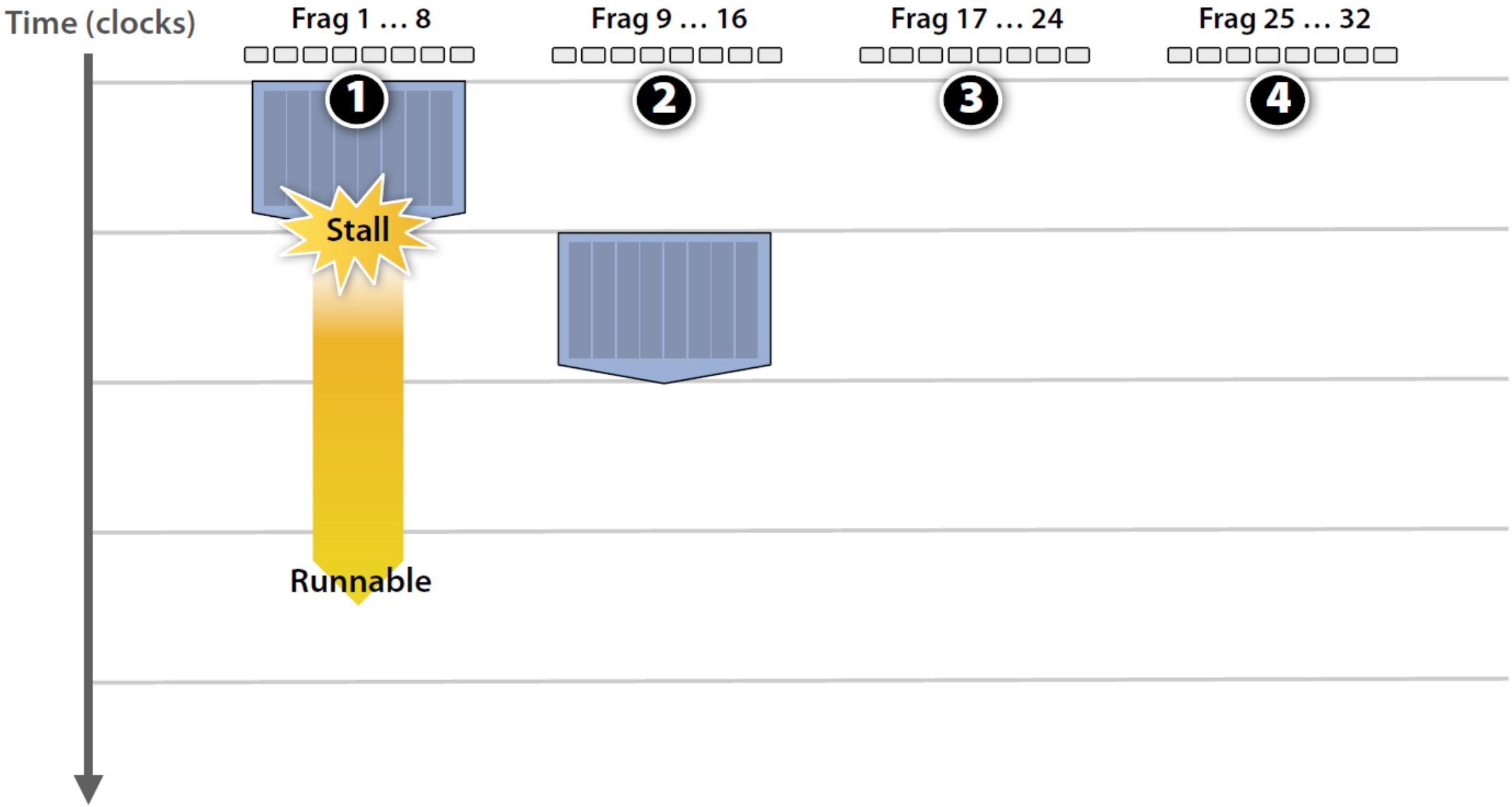
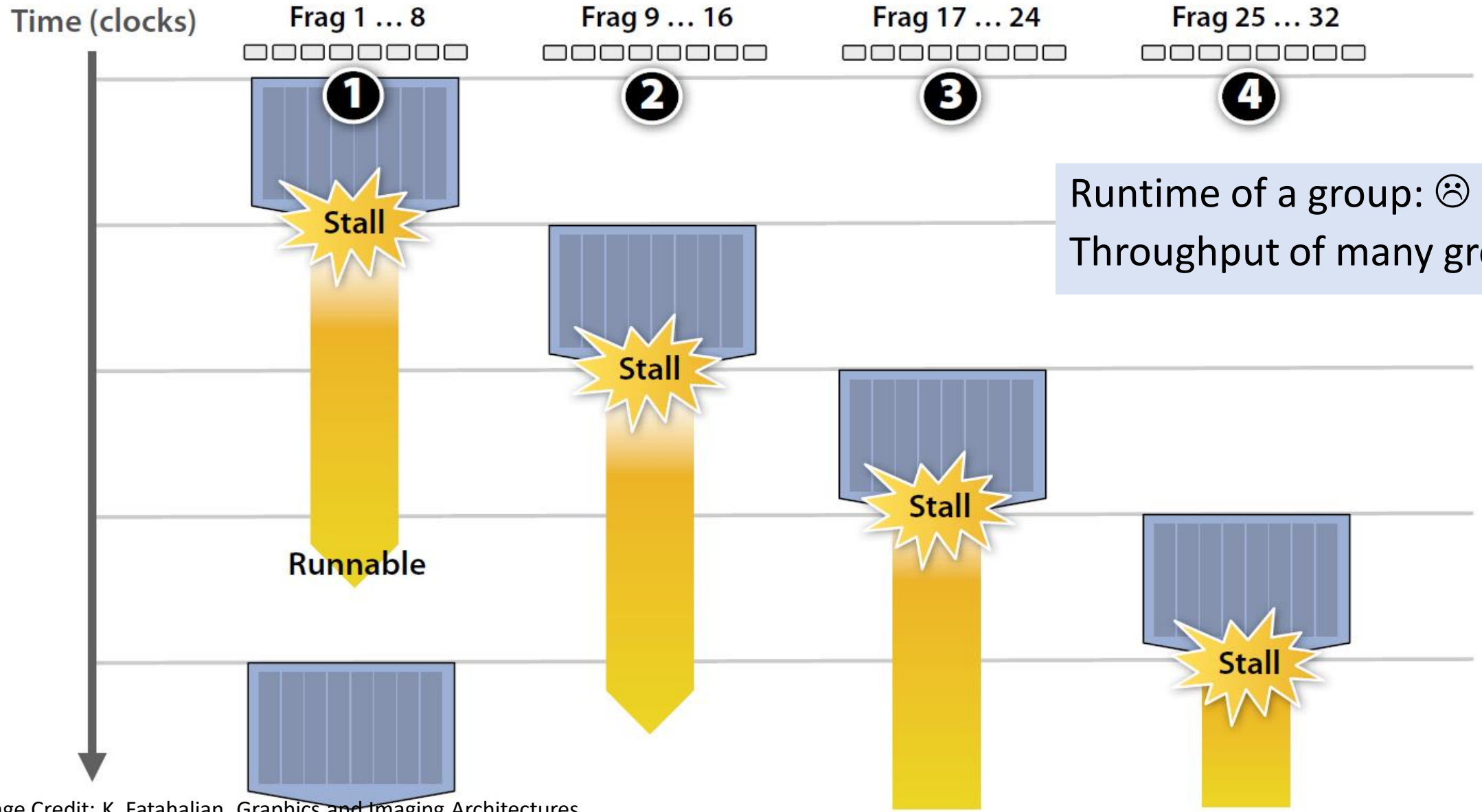


Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Throughput Computing



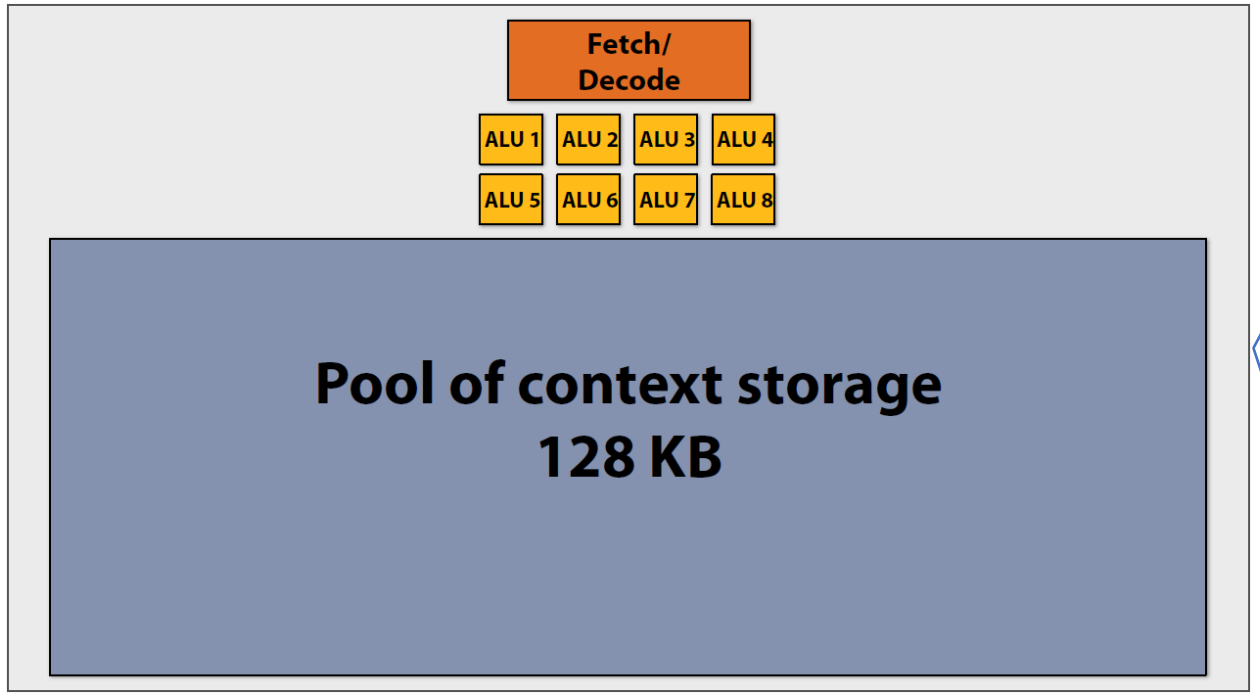
Runtime of a group: ☹

Throughput of many groups: ☺

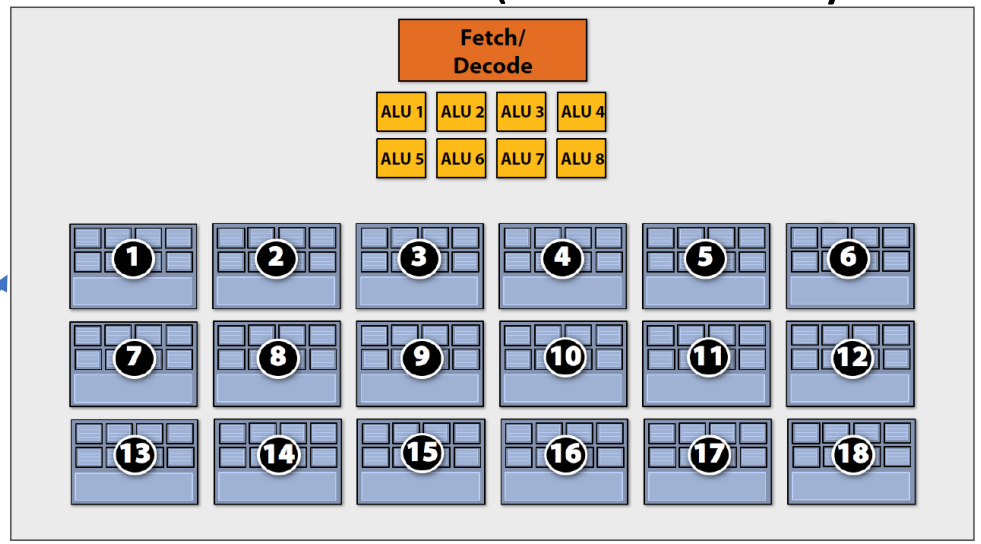
Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Storing Contexts

- Design choice



18 small contexts (more latency hiding)



12 medium contexts (bigger context size)

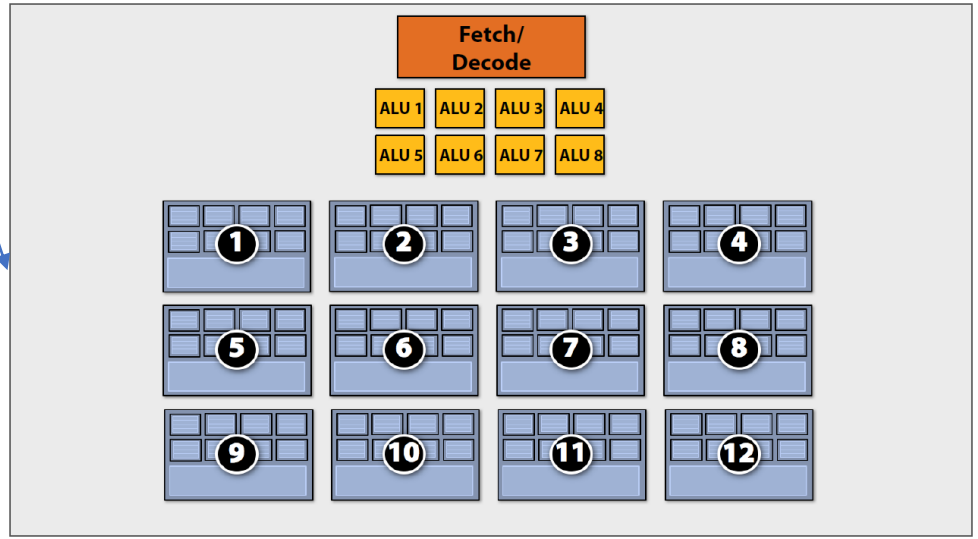
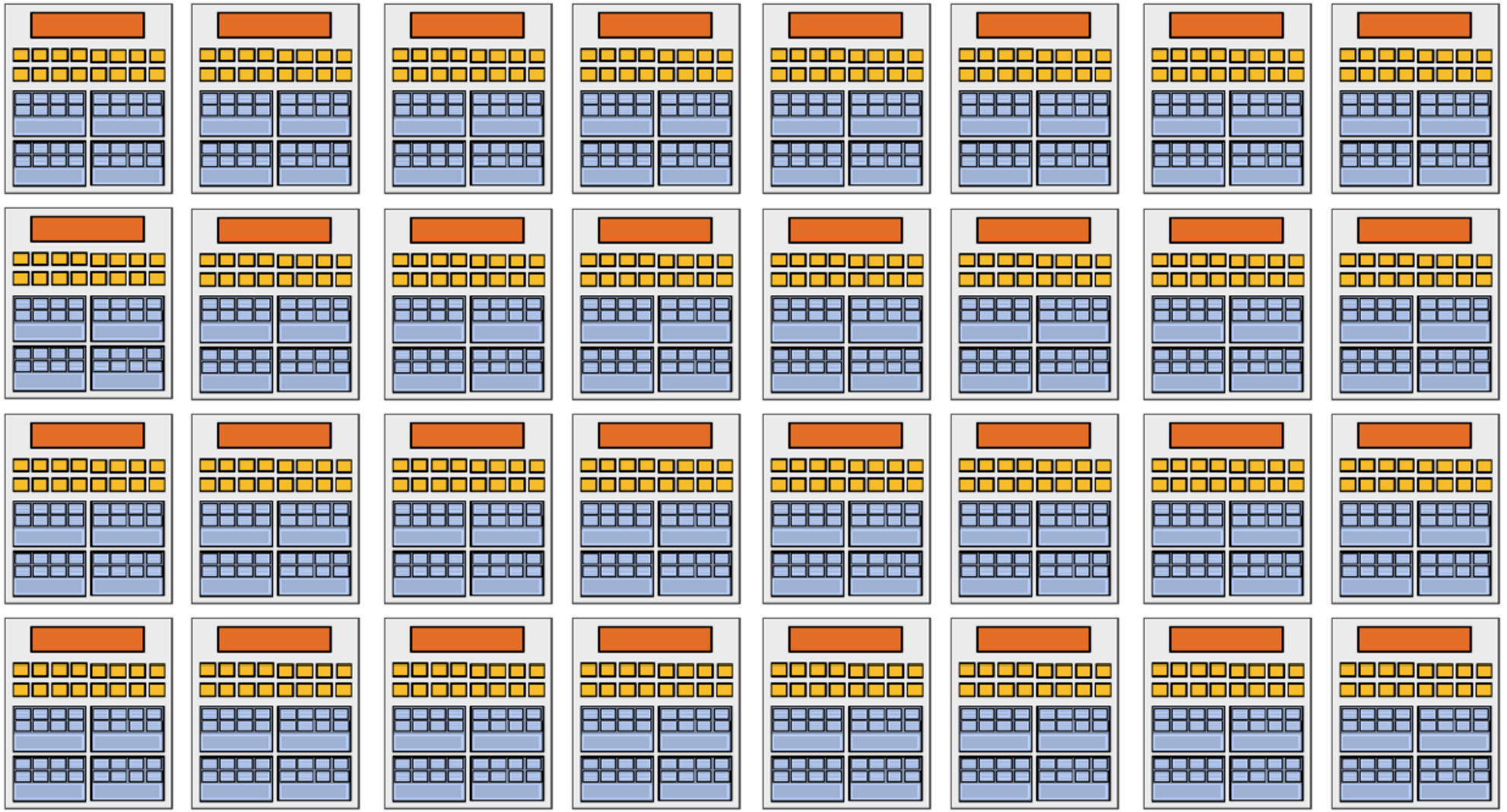


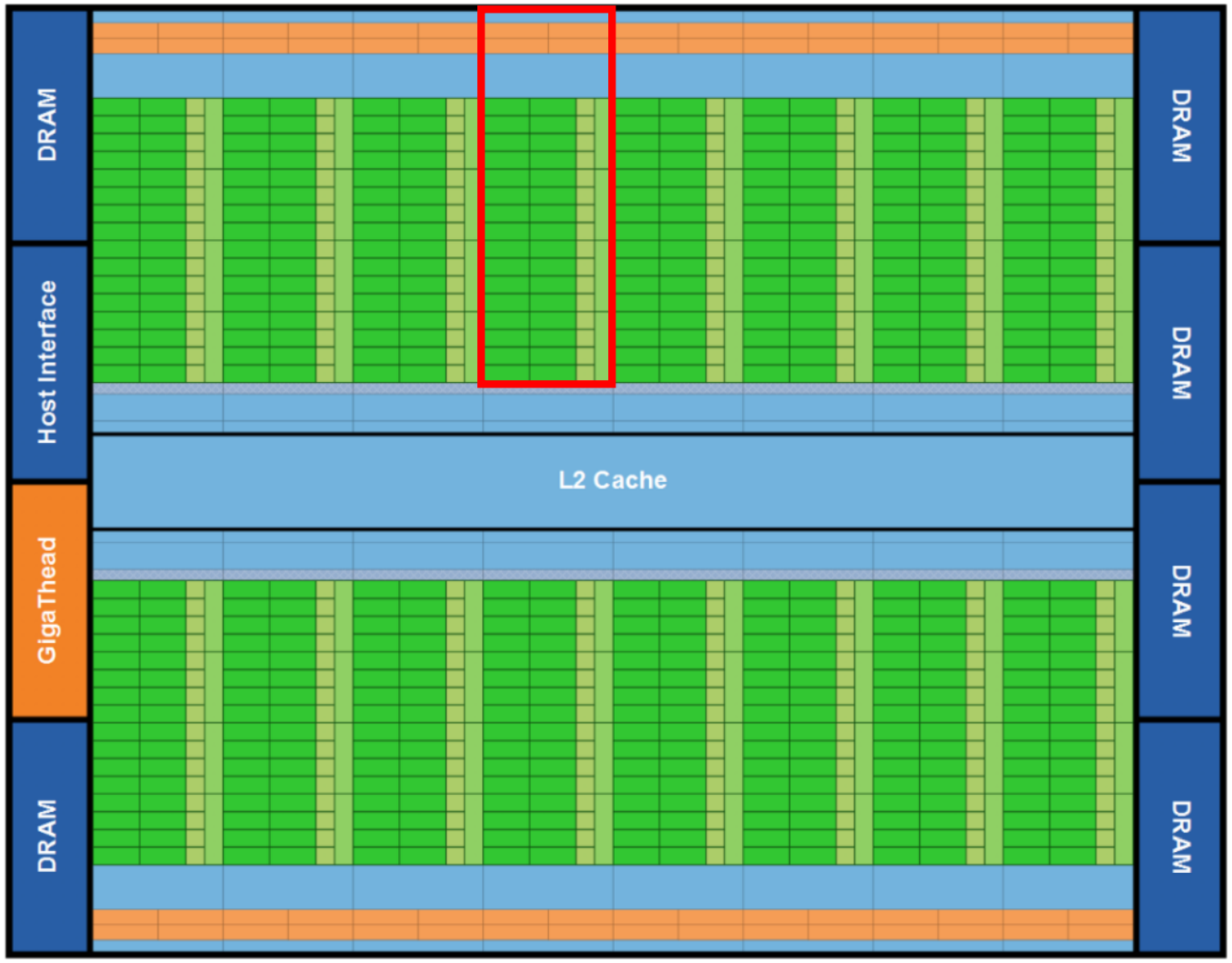
Image Credit: K. Fatahalian, Graphics and Imaging Architectures

Putting All Together



32 processor cores, 512 ALUs (16 per core) = 1 TFLOPS @ 1 GHz

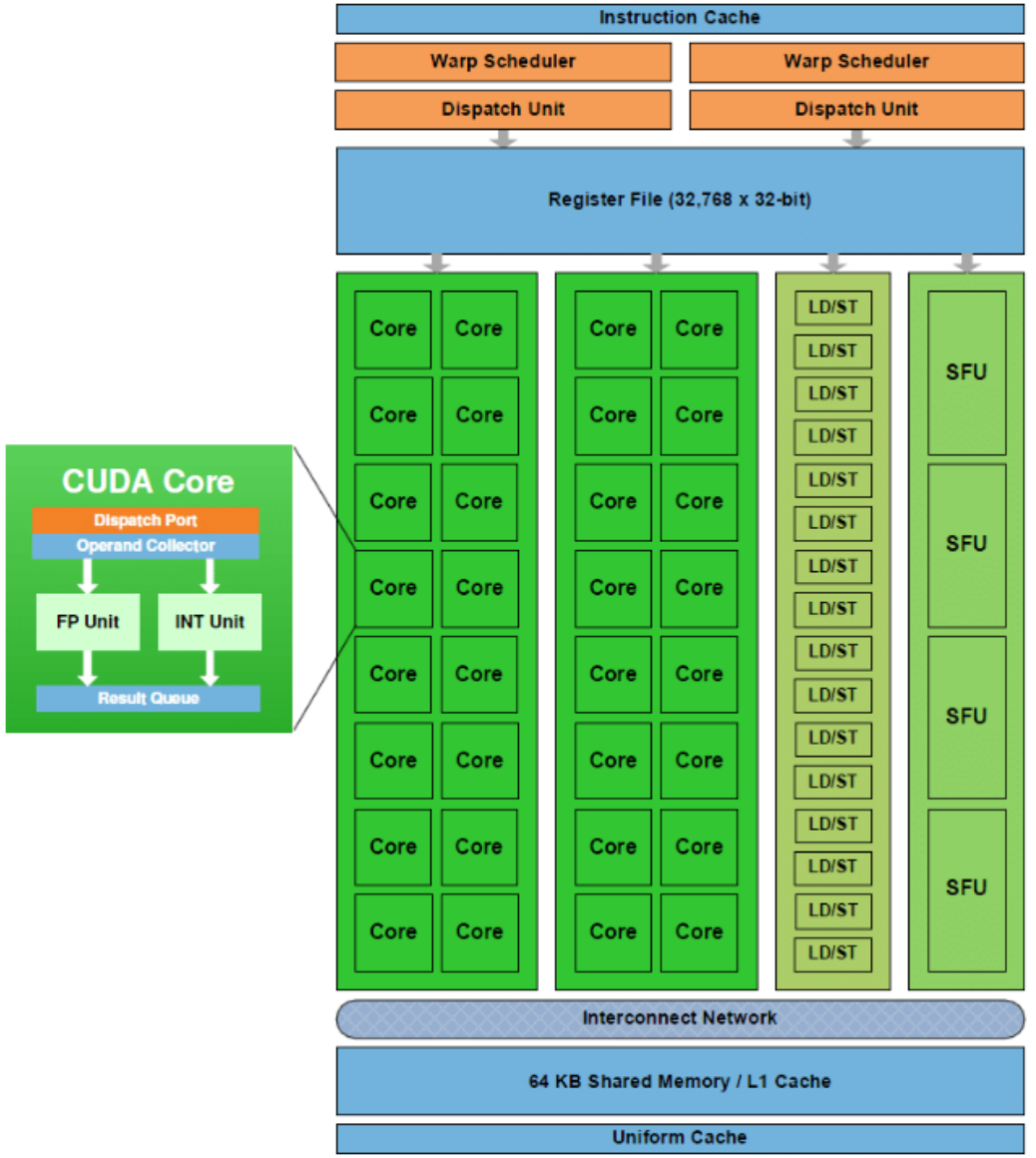
GPU Organization



- Fermi architecture (2010)
  - 16 Stream Multiprocessors
  - Common L2 cache

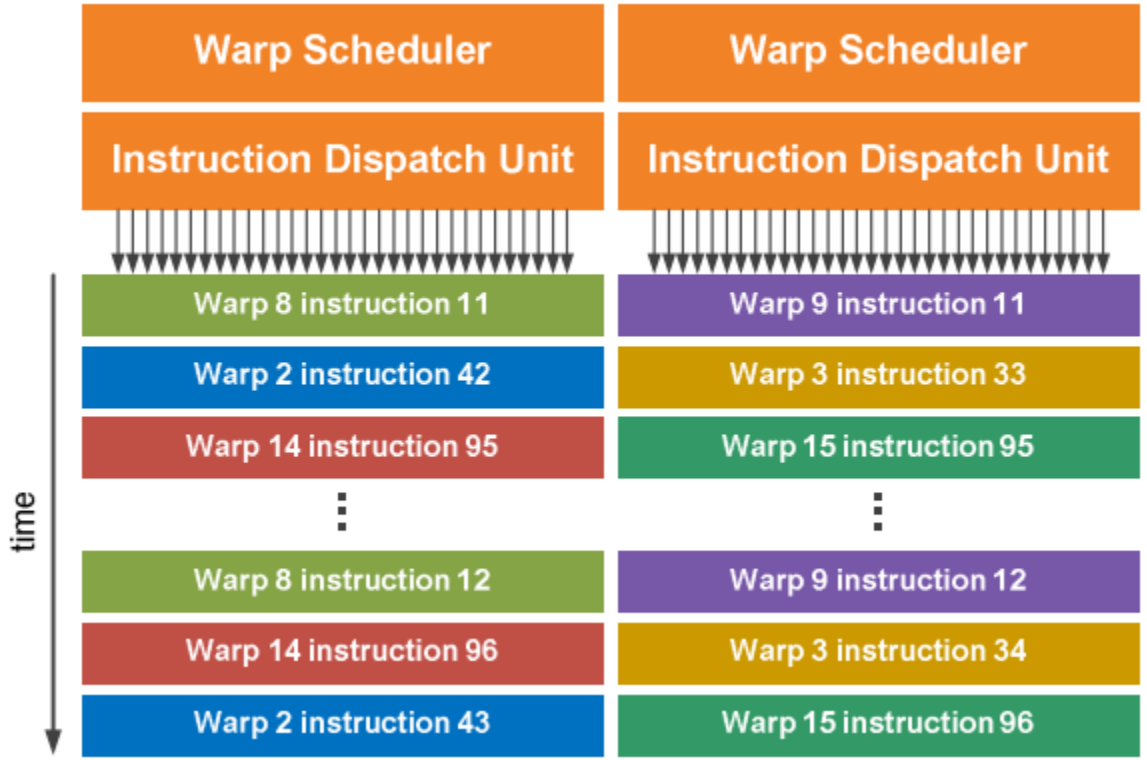
Stream Multiprocessor (SM)

- 32 CUDA cores
  - Each is an execute unit for INT and FP
- Warp scheduler & dispatch unit
  - Unified control path for CUDA cores
  - Schedules CUDA threads in a group of 32 threads called *warp*
- Large register file
- 16 load/store units
  - Support 16 threads' data transfer to cache
- 64K shared memory / L1 cache
- 4 special function unit
  - Sine, cosine, reciprocal, square root, …



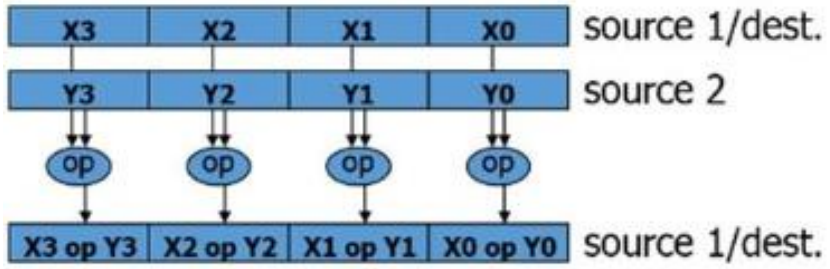
Warp Scheduler

- Warp: individual scalar instruction streams for each CUDA thread are grouped together for parallel execution on hardware
- Dual warp scheduler selects two warps and issues one instruction from each warp to a group of 16 cores, 16 LD/ST units, or 4 SFUs

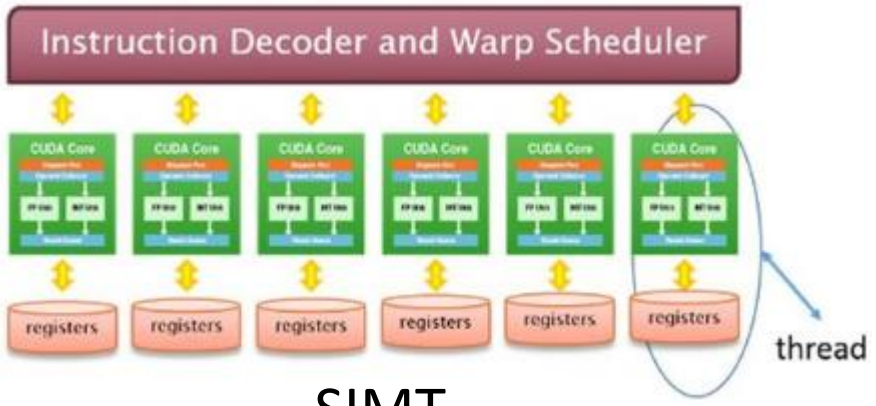


SIMT

- Single Instruction Multiple Threads
- Extension of SIMD: multiple with “threads” not “data”
- Multiple threads are processed by a single instruction in lock-step
- SIMT can do the following while SIMD can’t
  - Single instruction, multiple register sets
  - Single instruction, multiple addresses
  - Single instruction, multiple flow paths



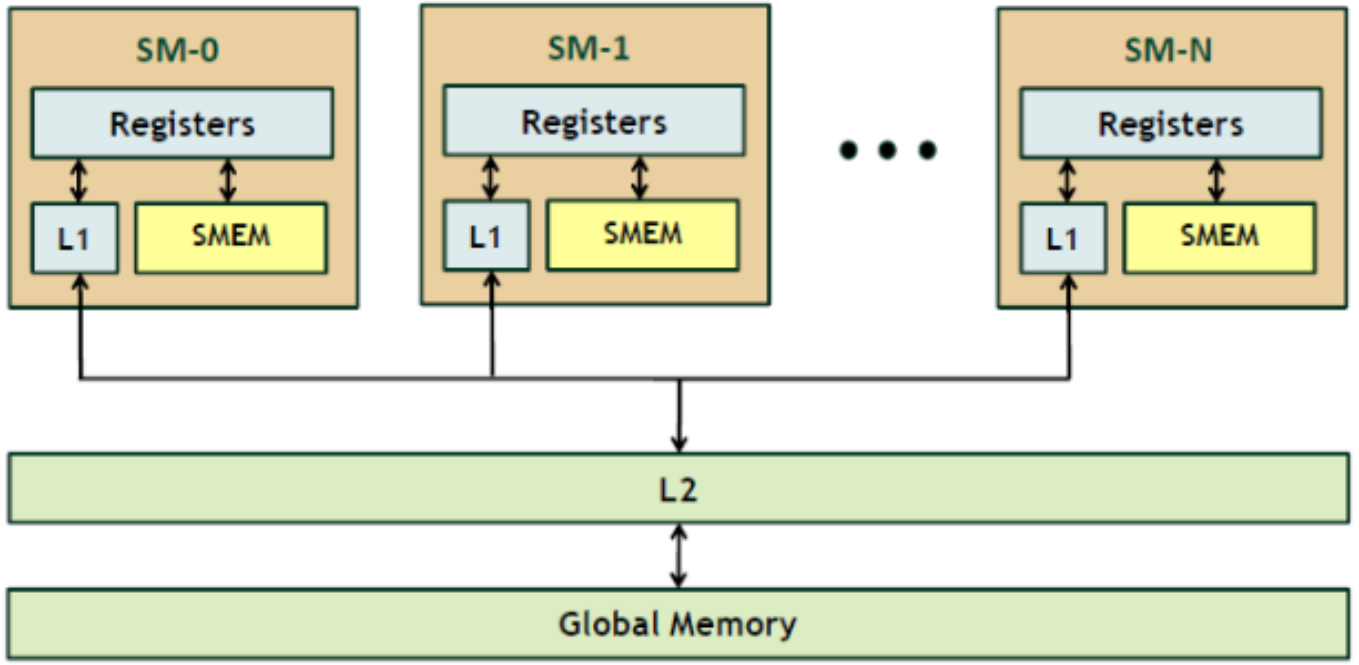
SIMD

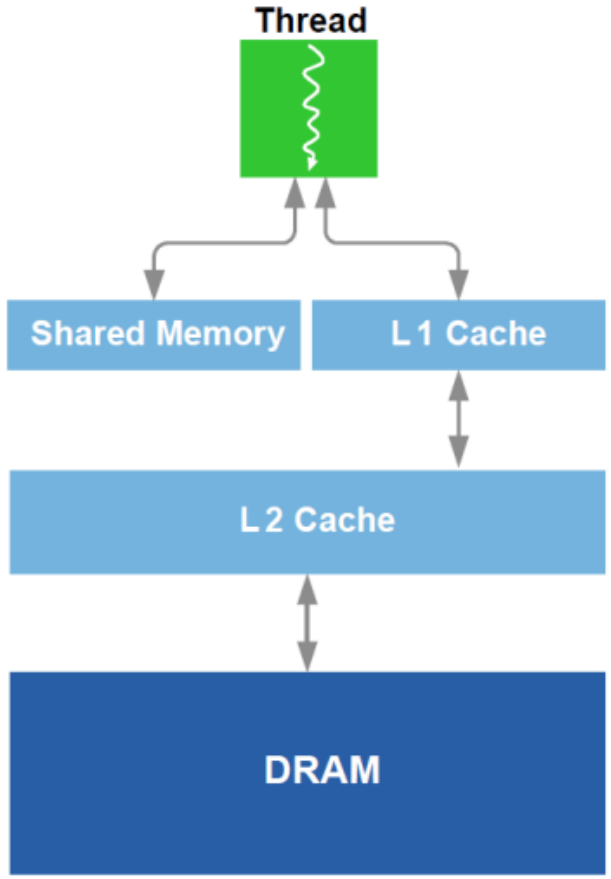


SIMT

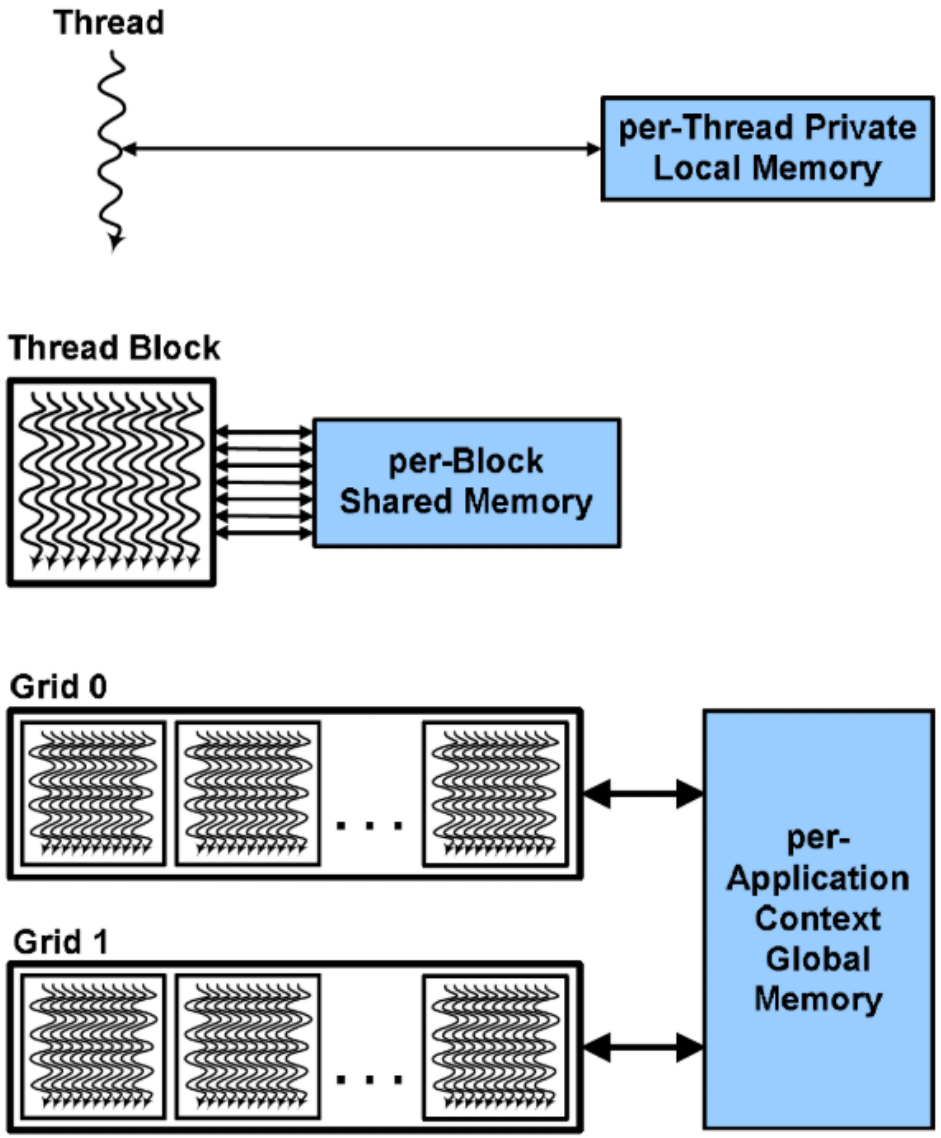
Memory Hierarchy

- Registers
- Shared memory
- L1 cache, L2 cache



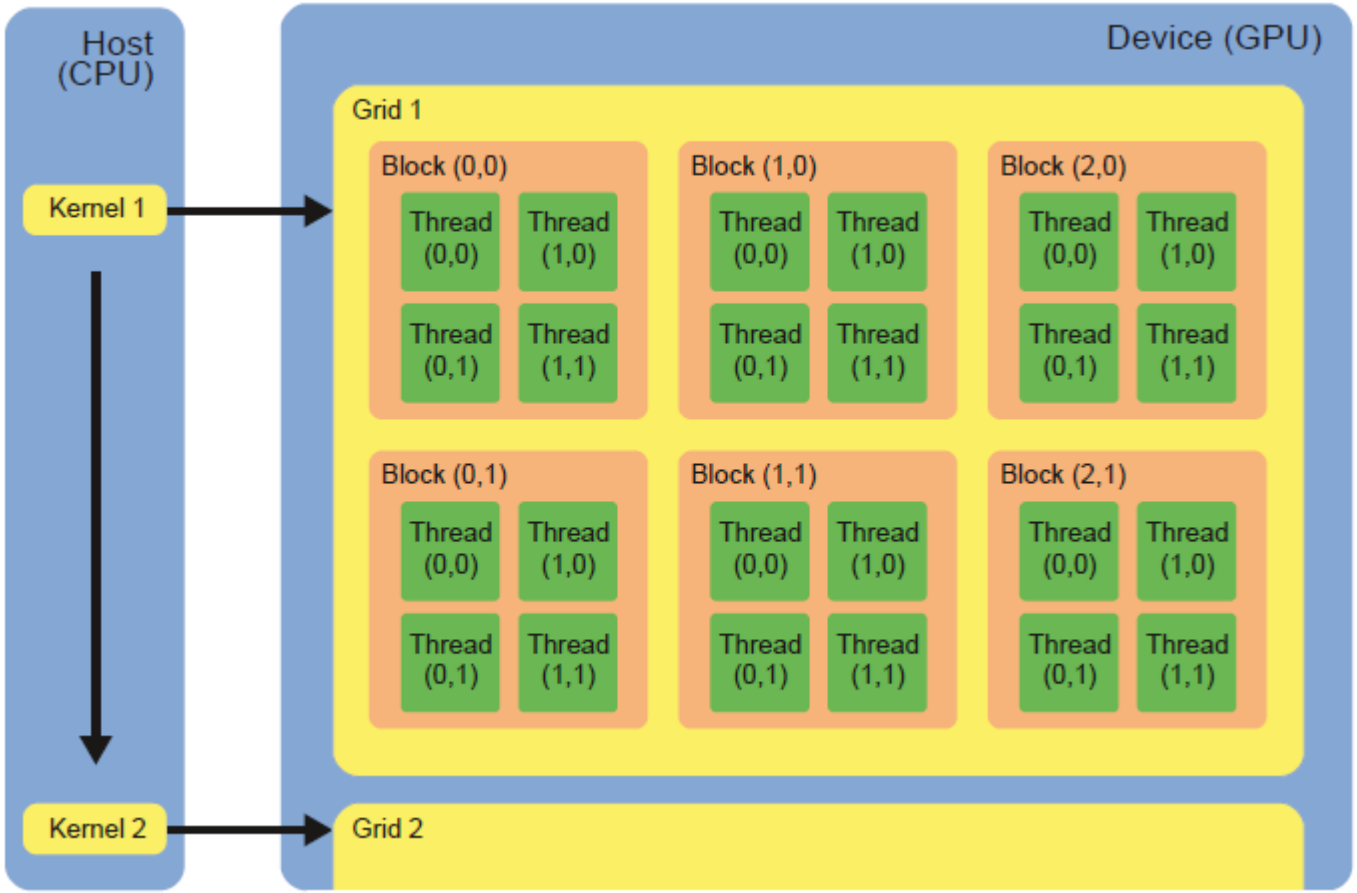


Computation Hierarchy



- Thread -> Block -> Grid
  - Up to 1024 threads forms a block
- SM executes blocks
  - Threads in a block are split into warps
- Hierarchy
  - CUDA thread uses register privately
  - Block cooperates with shared memory
  - Grid will correspond to global memory

Execution Model



Kernel: GPU program that runs on a grid of threads

Grid: a set of blocks executed on different SMs

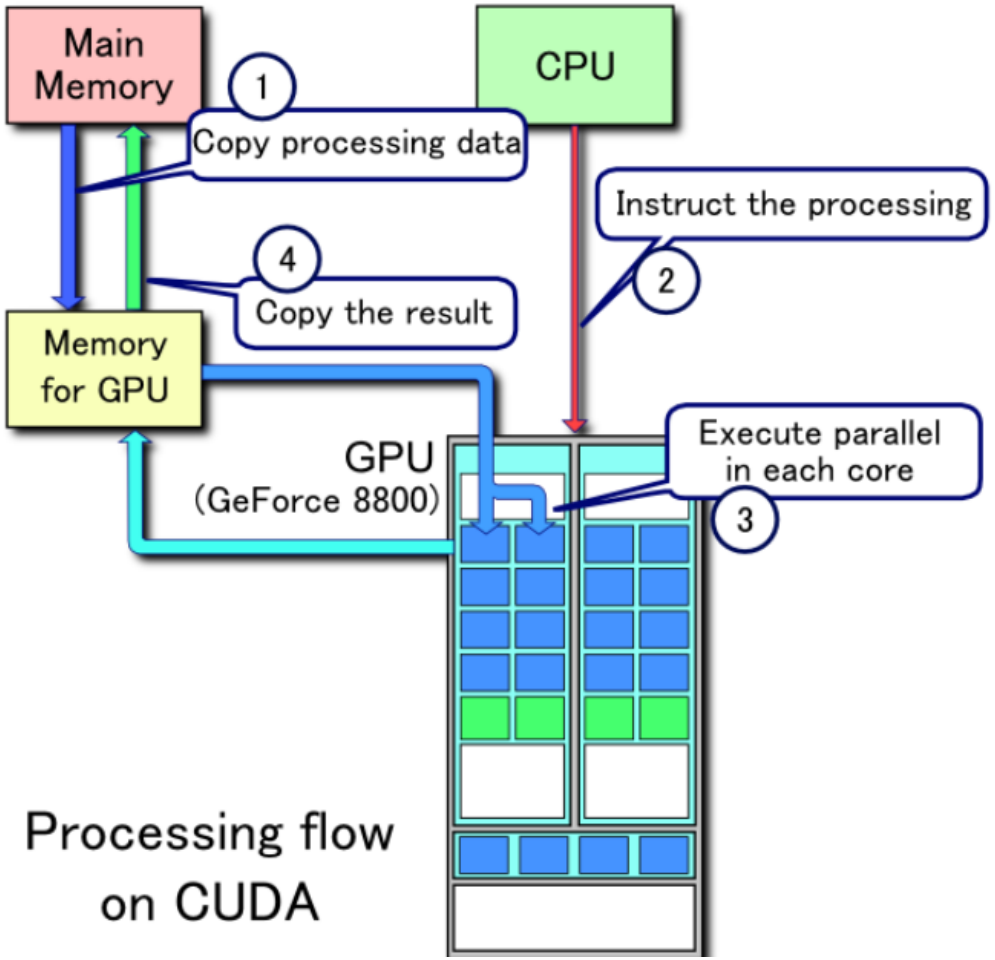
Block: a set of warps executed on the same SM

Warp: a group of 32 threads executed in lockstep

Thread: scalar execution unit

CUDA Programming Language

- Compute Unified Device Architecture
- A programming language for general-purpose GPU (GPGPU)
- An extension of C/C++
- Initially released in 2007, became de factor language for GPU programming
- Well suited for highly parallel applications
- GPU also supports other languages such as OpenGL, OpenCL



CUDA Programming Model

- 1-dimensional thread invocation

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Kernel is defined using specifier __global__

Each thread can be access through build-in variable threadIdx

Number of threads for Kernel is specified using a execution config syntax <<<…>>>

CUDA Programming Model

- 2-dimensional thread invocation

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

threadIdx is a 3 dimensional variable

1 block consists of N*N*1 threads

CUDA Programming Model

- Multi-block invocation

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread index should be calculated using block index and block dimension

N x N threads organized into multiple blocks (each block size is 16x16)

Array processing is explicitly parallelized with a few syntax extension to C++

Latest GPU for Machine Learning

21B transistors
- 815mm^2

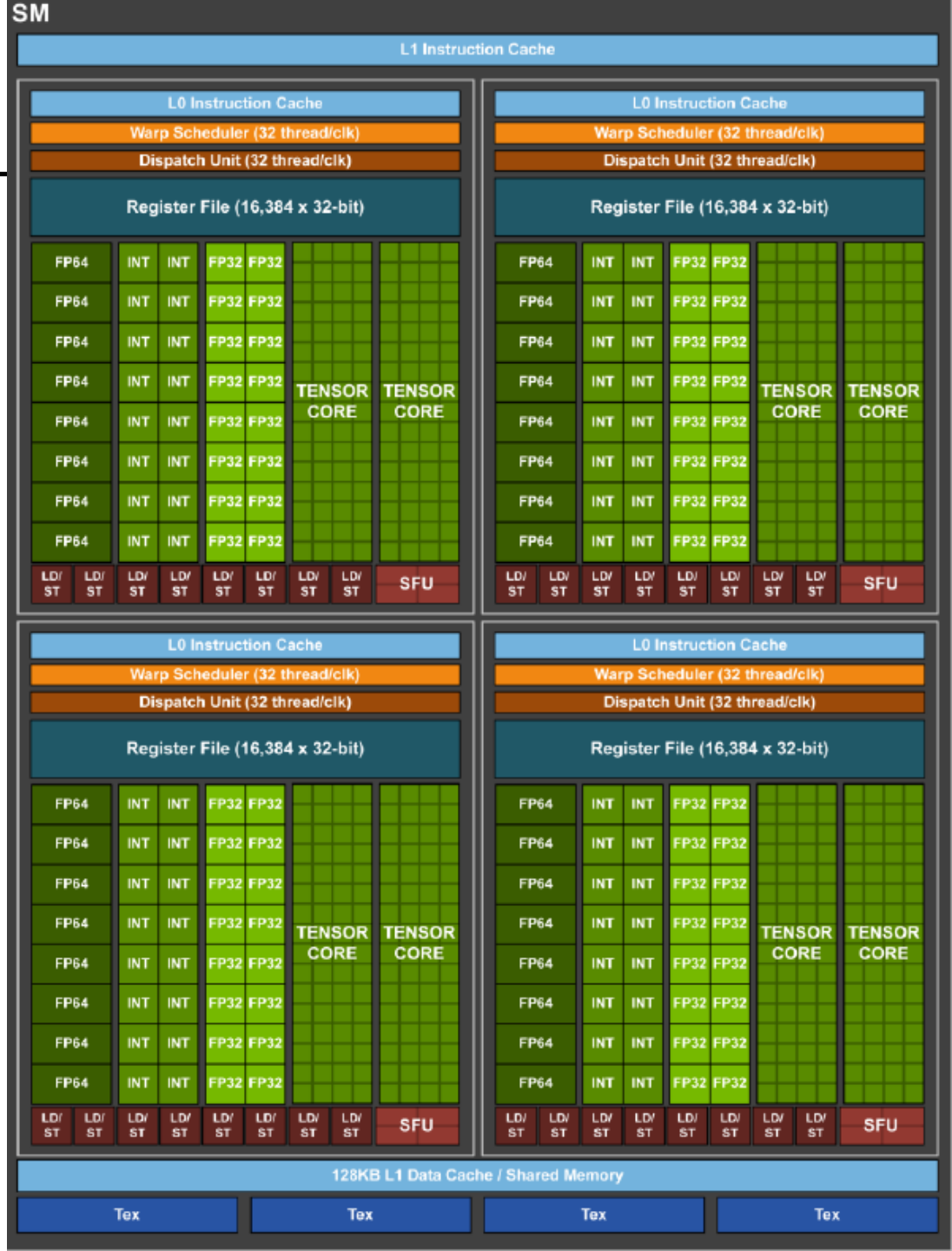
80 SMs
- 5120 CUDA cores
- 640 Tensor cores

I/O
- 16 GB HBM2
- 900 GB/s HBM2
- 300 GB/s NVLink

Volta Tesla V100

New Stream Multiprocessor

- 4 independent sub-cores
- Sub-core is similar to previous SM
  - Warp scheduler & dispatch unit
  - Register file
  - ALUs (FP64, FP32, INT)
  - Load/store
  - Tensor core
- 4 Sub-core shares L1 cache / SMEM

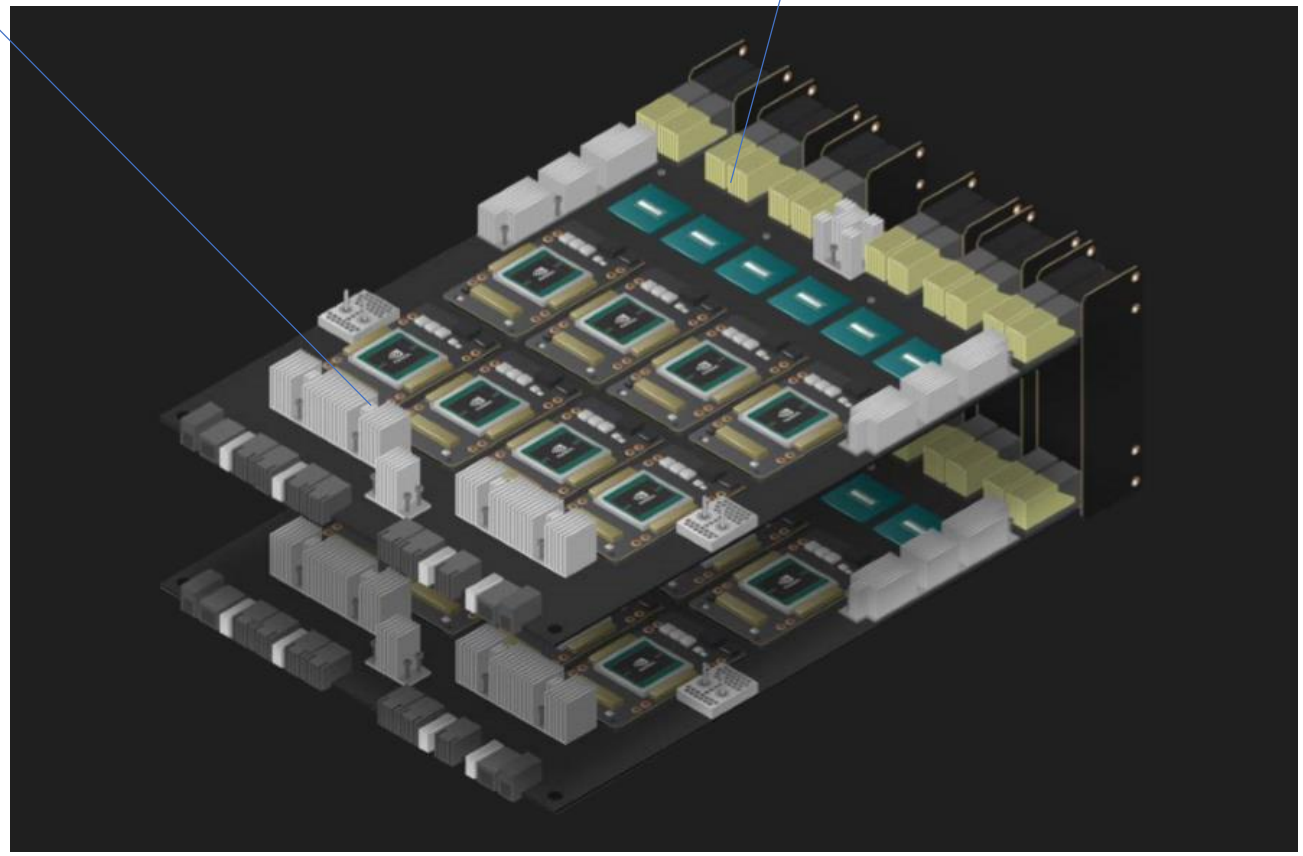


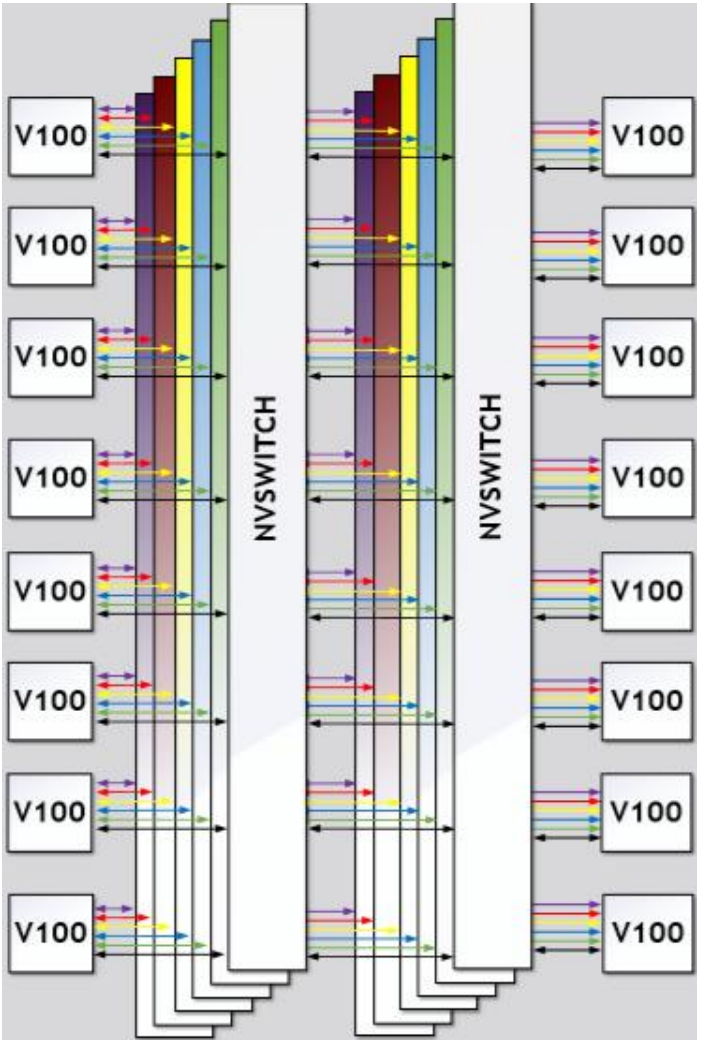
GPU Supercomputer (DGX-2)

8 x (Tesla V100 + HBM2)

6 x NVSwitch (Inter-GPU Comm.)

Two HGX-2 blades





16 V100s: 250T FLOPS, 512GB HBM, 16TB/s, 12 NVSwitches (25Gbps/ch): 2.4TB/s

Discussion

- How is training different from inference from the point of computation, data flow, and lifetime?
- Why is GPU very good for ML training?
  - Hint: input batching
- GPU dominates ML training market now. Do you think it will continue to do in the future? If not, why?
- GPU scales up computation cores, memory with HBM, and network with NVLink. What other improvements can you think from here?
- If you design your own training hardware, what would you do?