

Hardware for Machine Learning Systems

Pooyan Jamshidi
UofSC

The slides are mainly based on a NeurIPS'15 tutorial by William Dally

Typical Deep Learning System Stack

High level Packages

Programming API

Gradient Calculation (Differentiation API)

Computational Graph Optimization and Execution

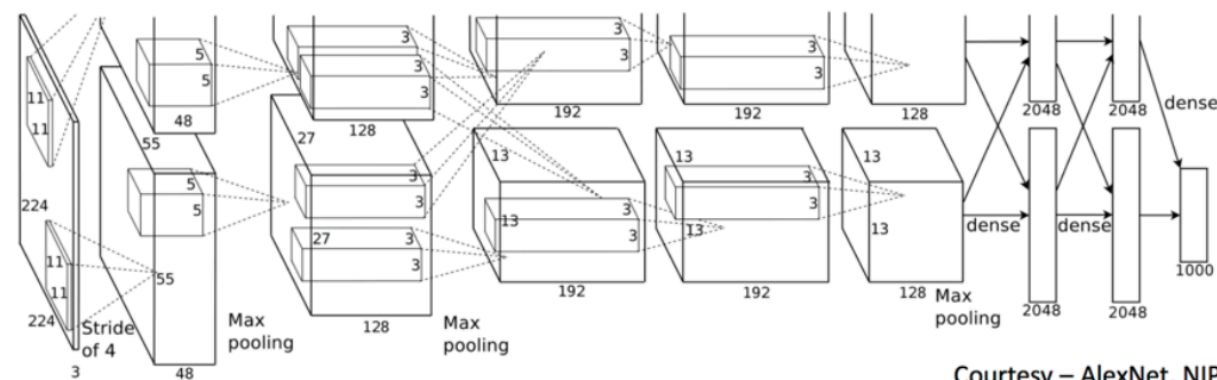
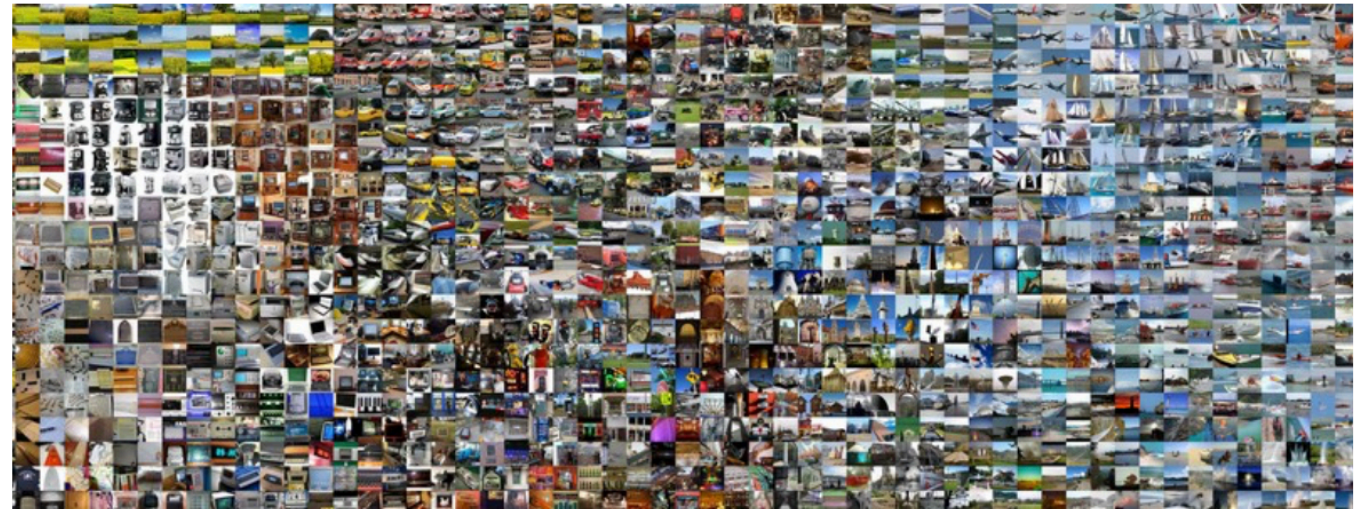
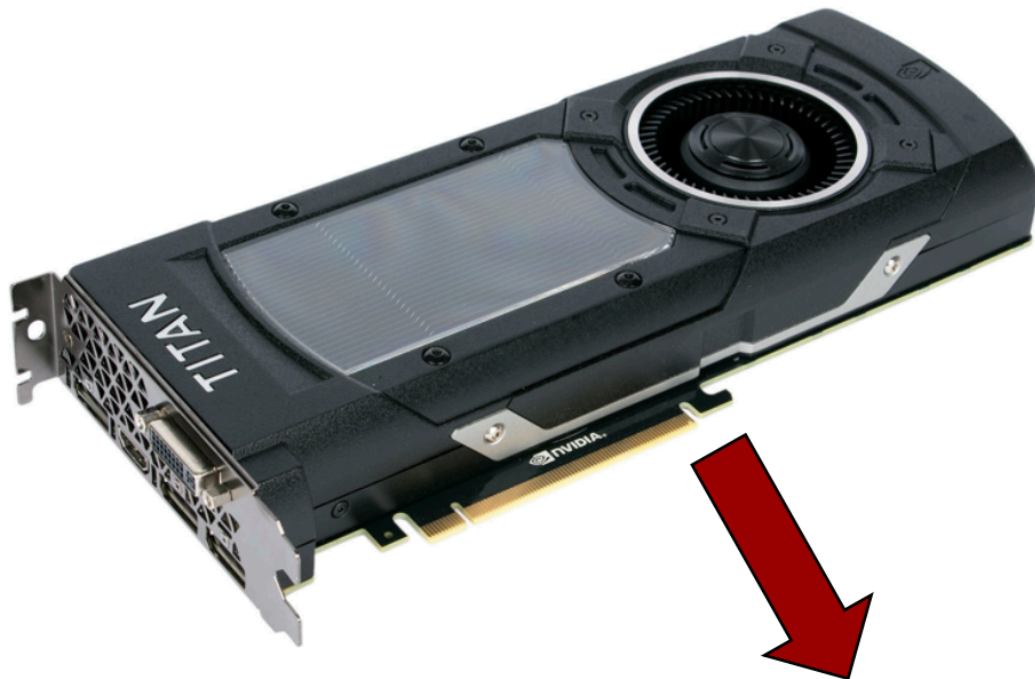
Runtime Parallel Scheduling

GPU Kernels, Optimizing Device Code

Accelerators and Hardwares

Architecture

Hardware and Data enable DNNs



Courtesy – AlexNet, NIPS 2012

The need for Speed

- Larger data sets and models lead to better accuracy but also increase computation time. Therefore progress in deep neural networks is limited by how fast the networks can be computed.
- Likewise the application of convnets to low latency inference problems, such as pedestrian detection in self driving car video imagery, is limited by how fast a small set of images, possibly a single image, can be classified.

More data → Bigger Models → More Need for Compute
But Moore's law is no longer providing more compute...

Problem

Acceleration

- Run a network faster (Performance, inf/s)

- Run a network more efficiently

- Energy (inf/J)
- Cost (inf/s\$)

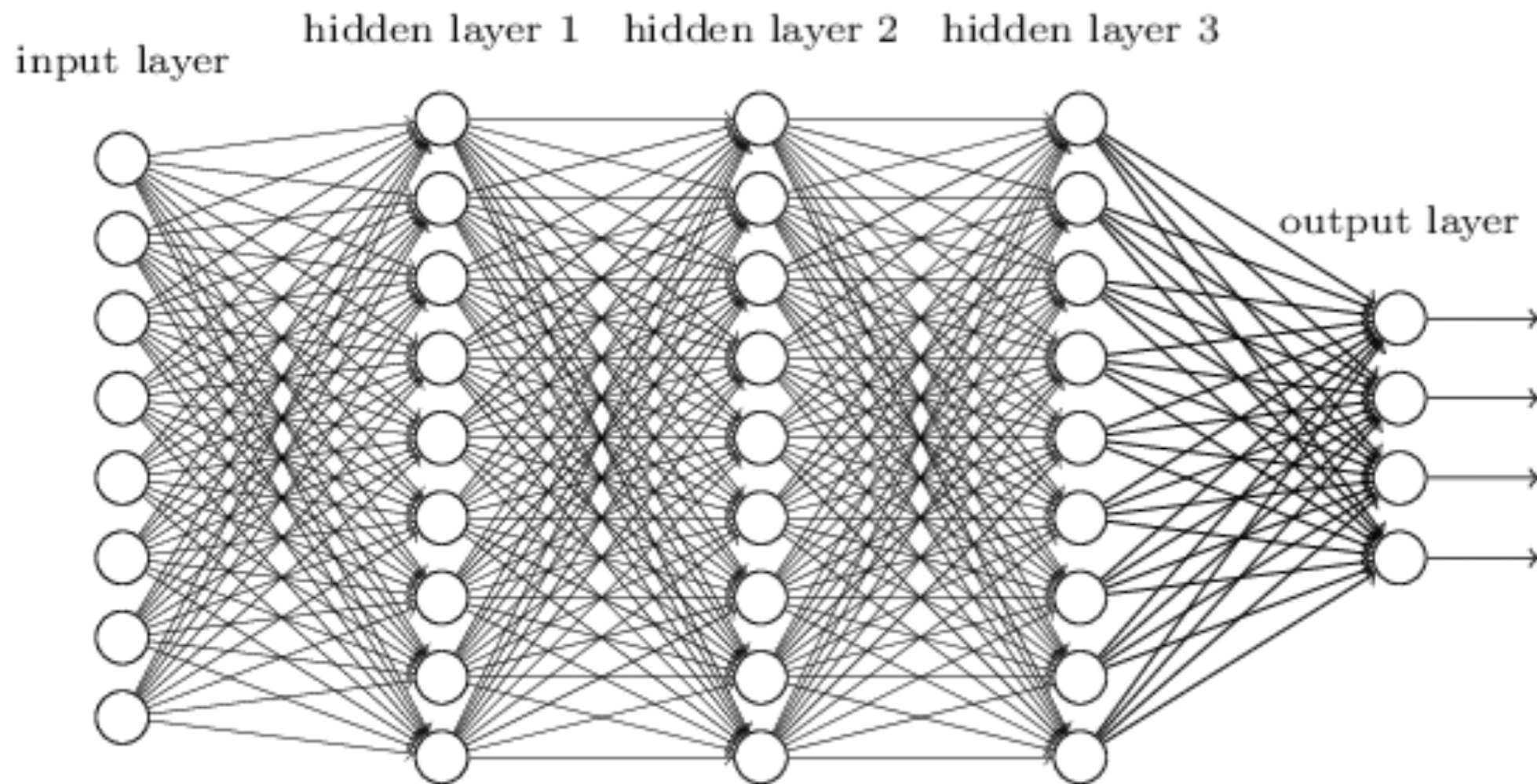
- Inference

- Just running the network forward

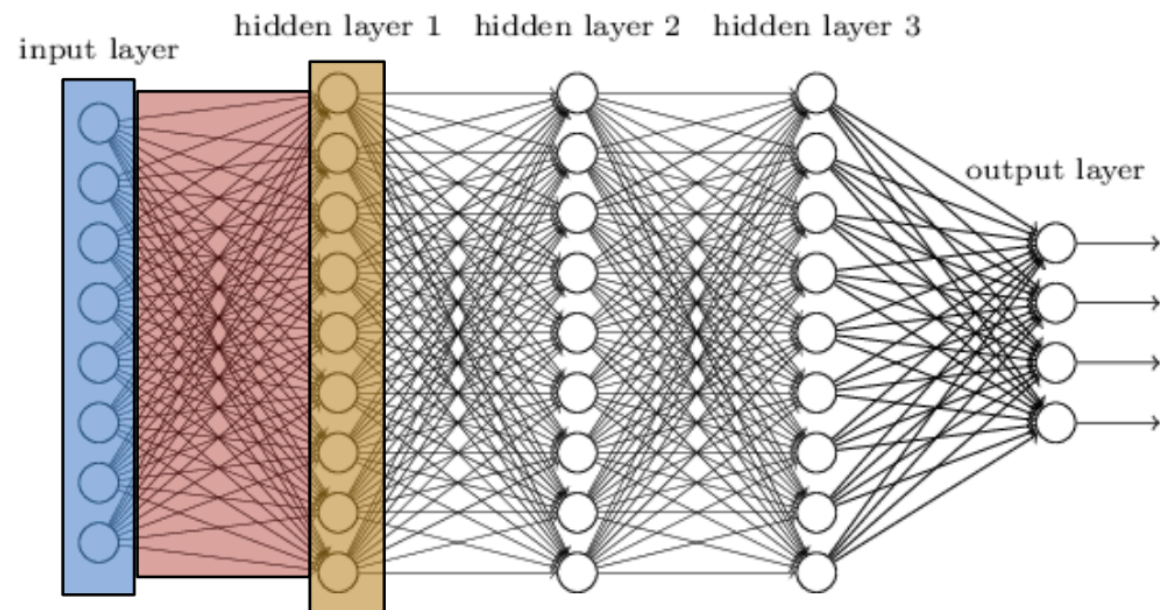
- Training

- Running the network forward
- Back-propagation of gradient
- Update of parameters

What Network? DNNs, CNNs, and RNNs

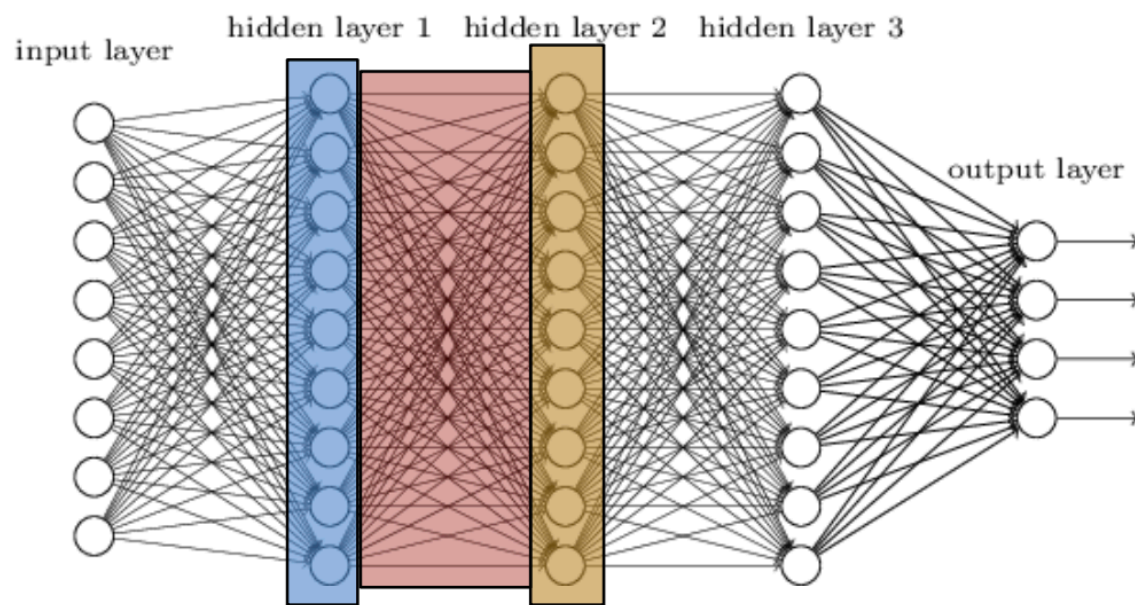


DNN, key operation is dense $M \times V$



$$\begin{array}{c} \text{Output activations} \\ b_i \end{array} = \begin{array}{c} W_{ij} \\ \text{weight matrix} \end{array} \times \begin{array}{c} \text{Input activations} \\ a_j \end{array}$$

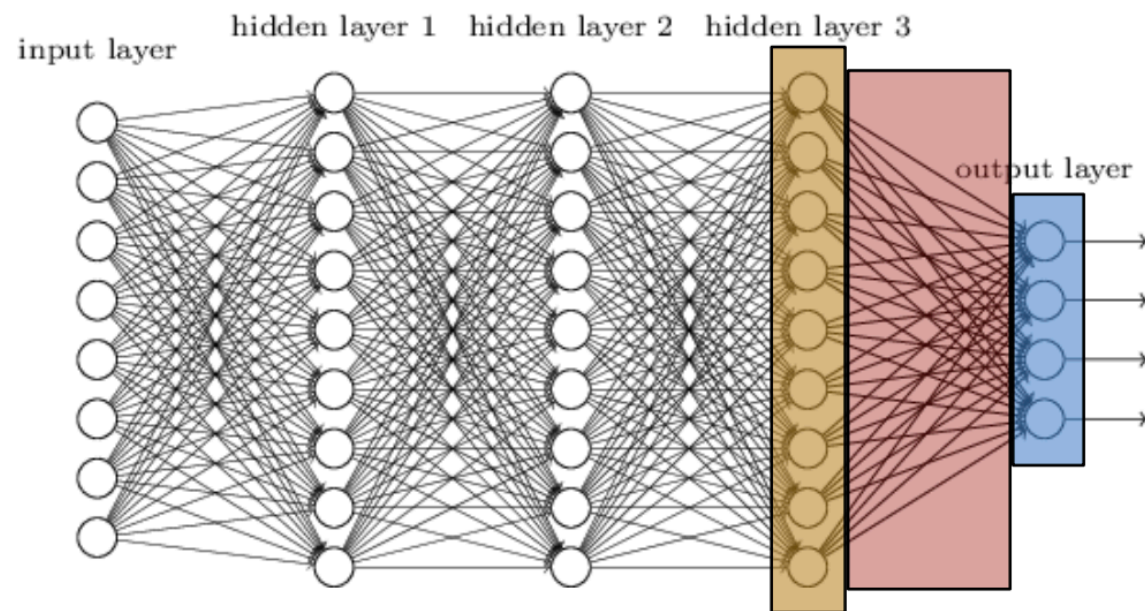
DNN, key operation is dense $M \times V$



Repeat for each layer

$$\begin{array}{c} \text{Output activations} \\ \mathbf{b}_i \end{array} = \begin{array}{c} \mathbf{W}_{ij} \\ \text{weight matrix} \end{array} \times \begin{array}{c} \text{Input activations} \\ \mathbf{a}_j \end{array}$$

DNN, key operation is dense $M \times V$



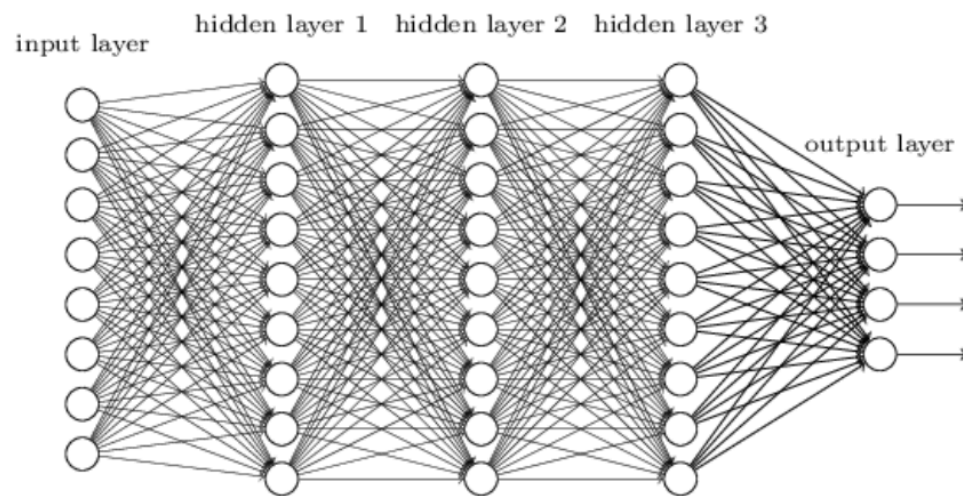
Backpropagation just does
this backward

A diagram illustrating the backpropagation operation as a matrix multiplication. It shows a vertical yellow rectangle labeled b_i (Input gradient) on the left, followed by an equals sign, a large red square labeled W_{ij} (weight matrix) in the center, followed by a multiplication sign, and a vertical blue rectangle labeled a_j (Output gradient) on the right. Below each rectangle is its corresponding label: "Input gradient" for b_i , "weight matrix" for W_{ij} , and "Output gradient" for a_j .

$$b_i = W_{ij} \times a_j$$

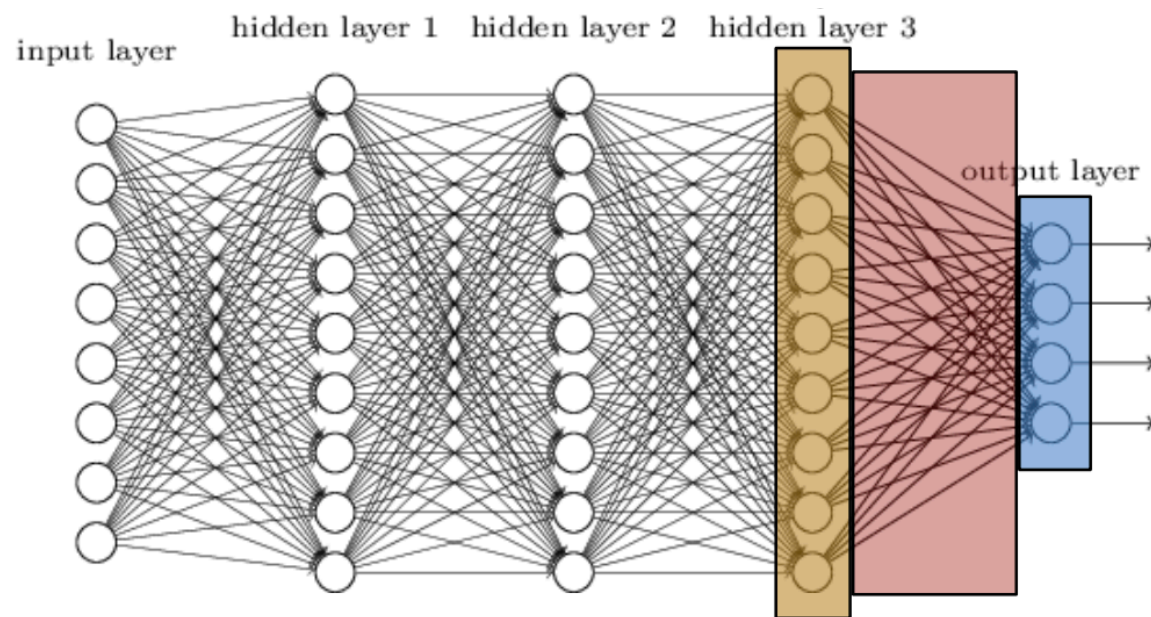
Input gradient weight matrix Output gradient

Training, and Latency Insensitive Networks can be Batched – operation is $M \times M$ – gives re-use of weights



$$\begin{array}{c} \text{Output activations} \\ \mathbf{b_{ik}} \end{array} = \begin{array}{c} \mathbf{W_{ij}} \\ \text{weight matrix} \end{array} \times \begin{array}{c} \mathbf{a_{jk}} \\ \text{Input activations} \end{array}$$

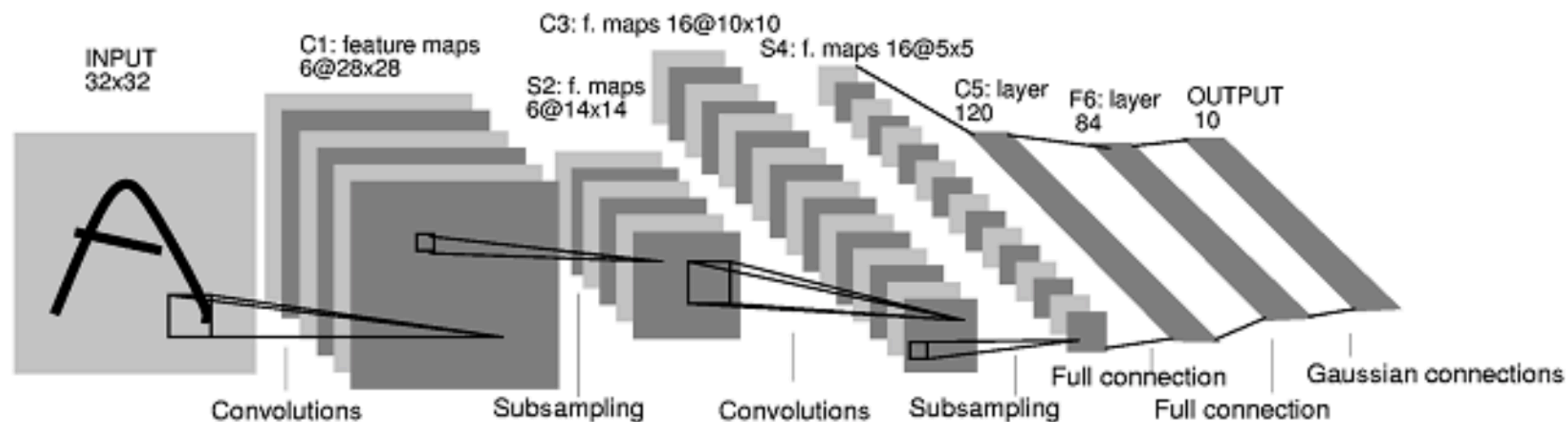
For real time you can't batch And there is
sparsity in both weights and activations
key operations is $\text{spM} \times \text{spV}$



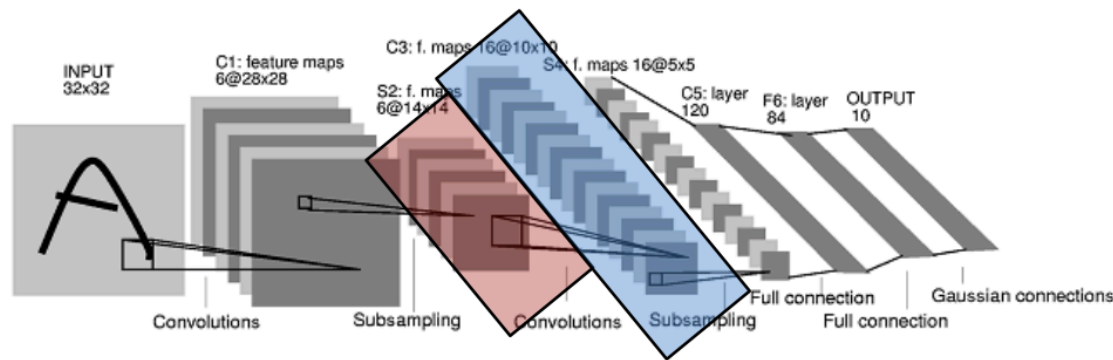
Backpropagation just does
this backward

$$\begin{array}{c} \text{Input gradient} \\ b_i \end{array} = \begin{array}{c} \text{weight matrix} \\ W_{ij} \end{array} \times \begin{array}{c} \text{Output gradient} \\ a_j \end{array}$$

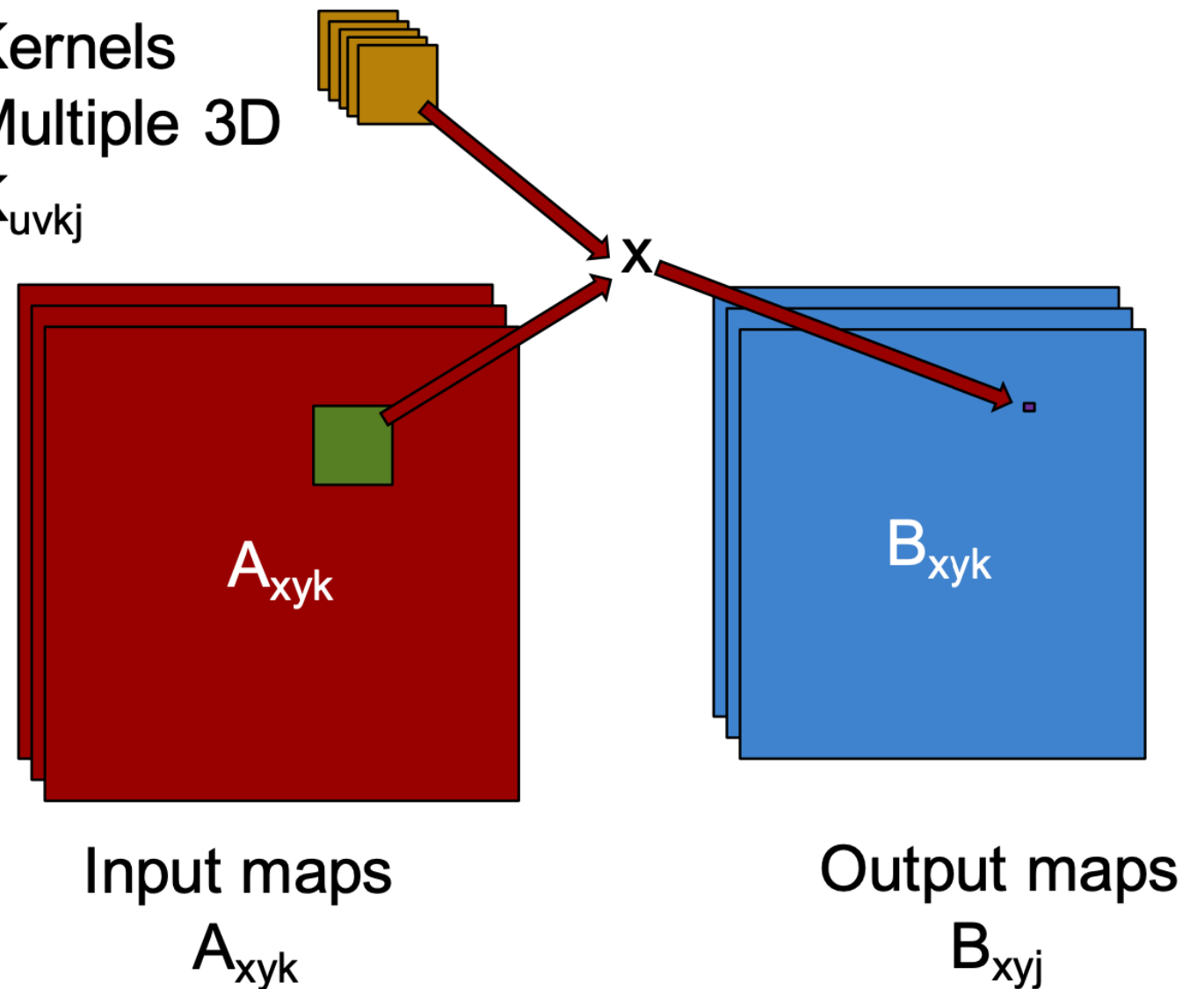
CNNs – For Image Inputs, Convolutional stages act as trained feature detectors



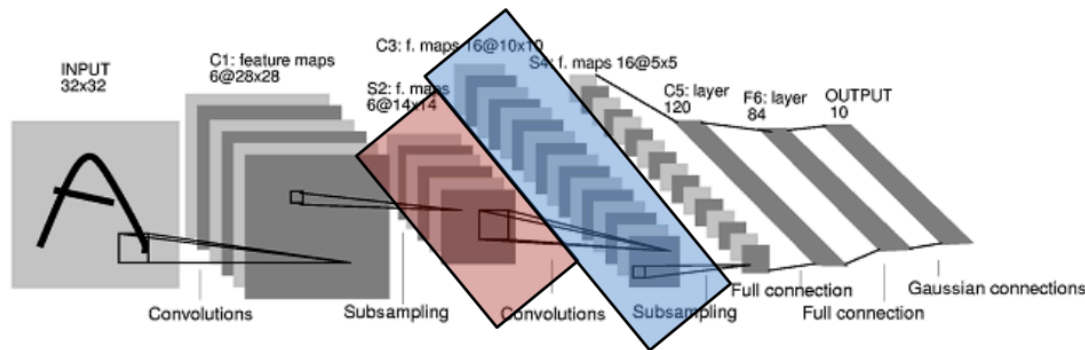
CNNs require Convolution in addition to $M \times V$



Kernels
Multiple 3D
 K_{uvkj}



CNNs require Convolution in addition to $M \times V$



Kernels
Multiple 3D
 K_{uvkj}



$\times$

6D Loop

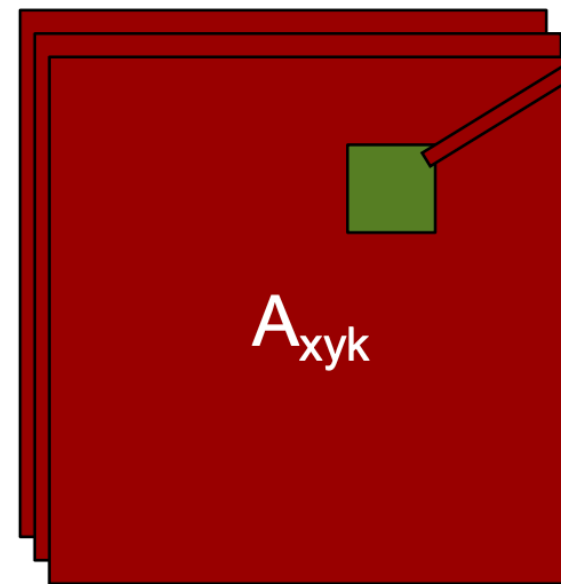
For each output map j

For each input map k

For each pixel x, y

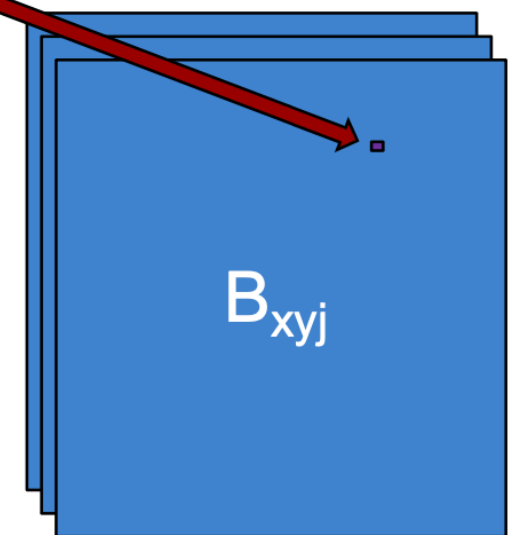
For each kernel element u, v

$$B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}$$



Input maps

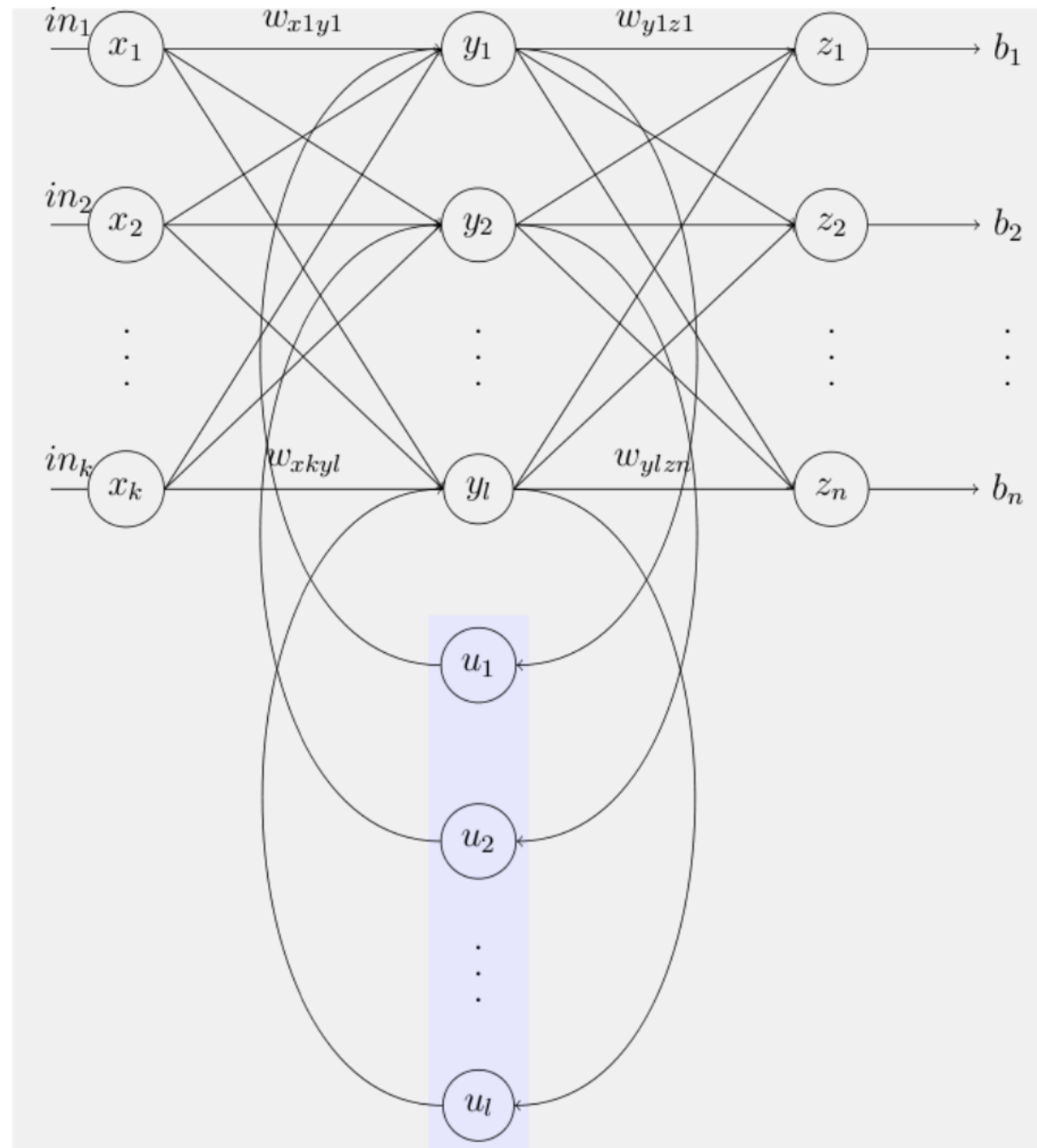
A_{xyk}



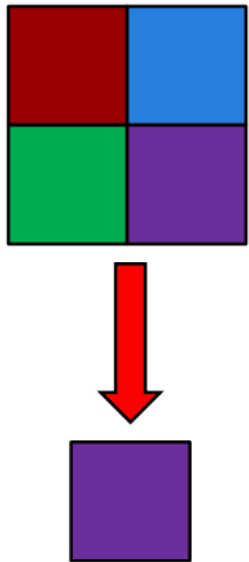
Output maps

B_{xyj}

RNNs



Some Other Operations



Pooling

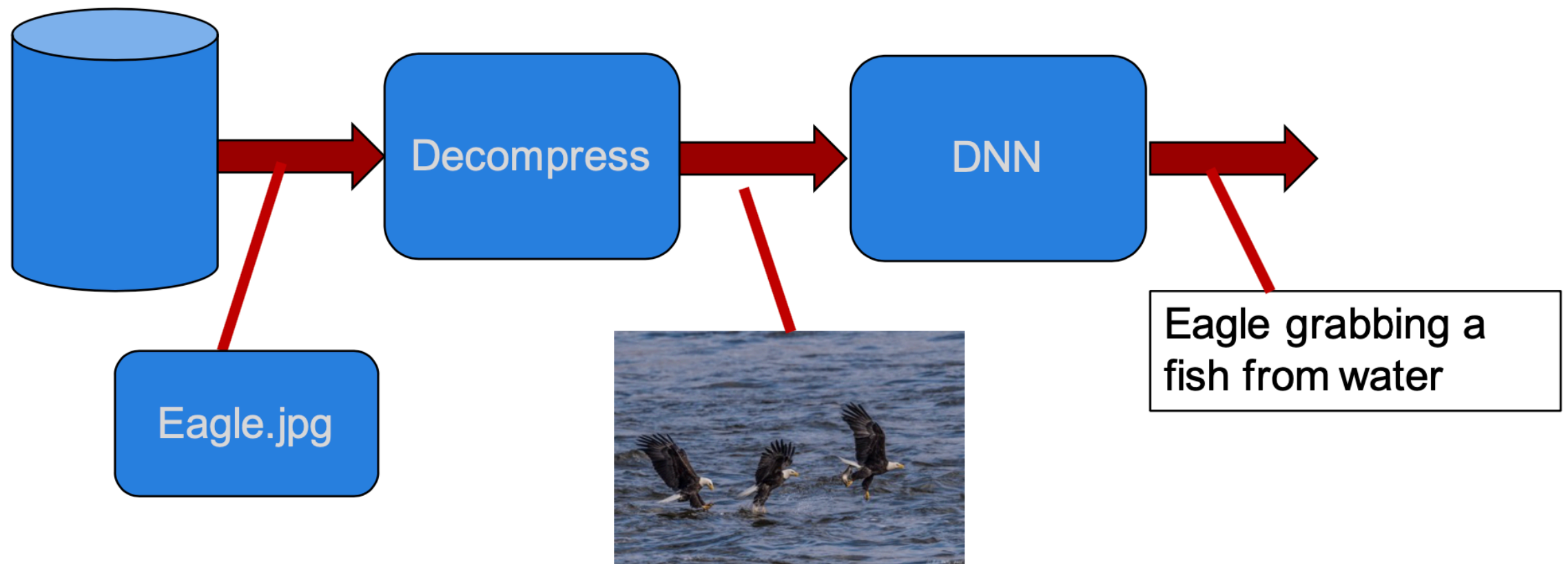


ReLU
(or other non-linear function)

$$w_{ij} += \alpha a_j g_i$$

Weight Update

Infrastructure



Summary of the Problem

- Run DNNs, CNNs, and RNNs
 - For training and inference
 - Can batch if not latency sensitive
- Optimize
 - Speed inf/s
 - Efficiency inf/J, inf/s\$
- Key operations are
 - MxV
 - MxM if batched
 - May be sparse (spM x spV)
 - Convolution
- Also
 - Pooling, non-linear operator (ReLU), weight update

Baseline

Baseline Performance Xeon E5-2698 – Single Core



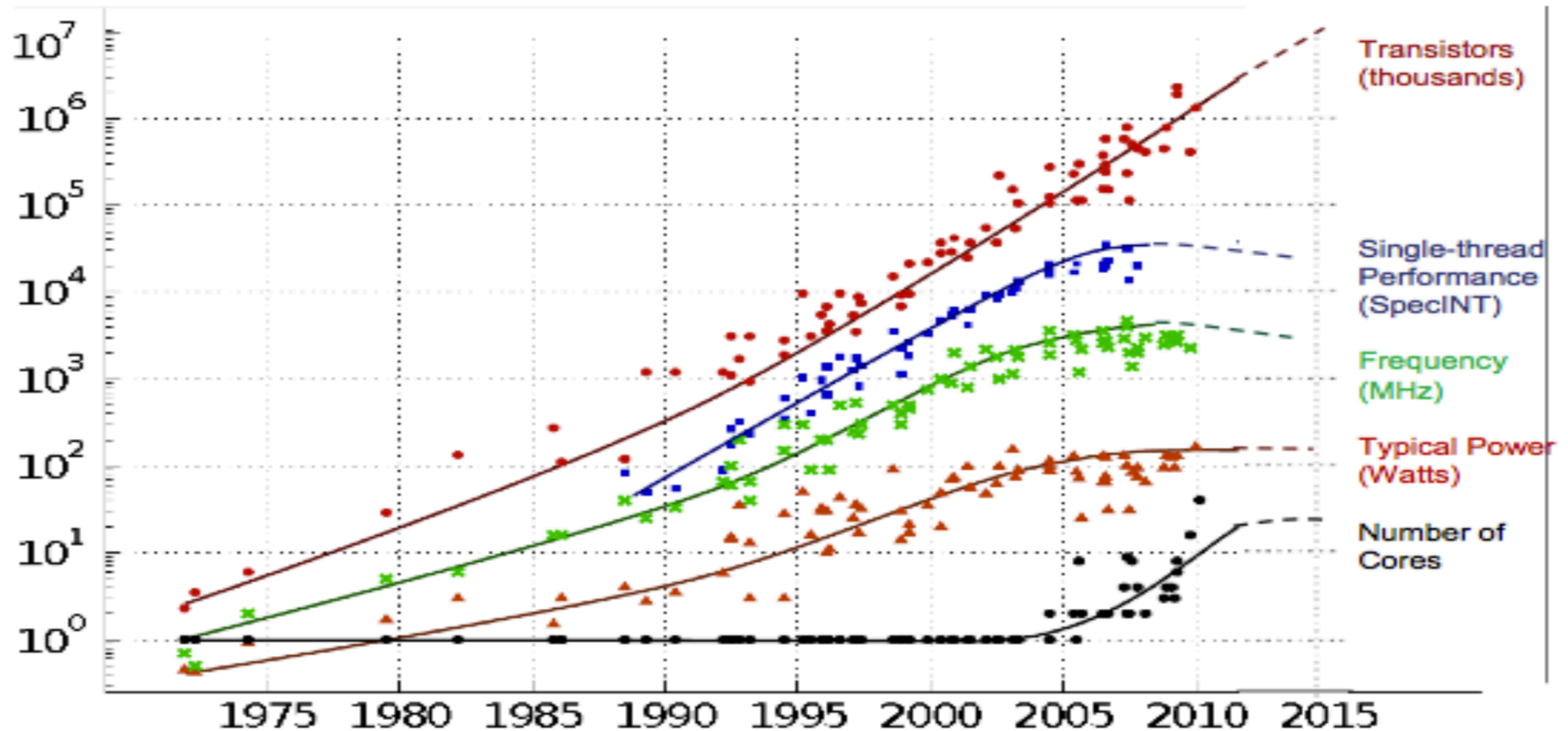
AlexNet – inference, batched

30 f/s

3.2 f/J

Most ops on AVX (SIMD) units

Moore's law made CPUs 300x faster than in 1990 But its over...

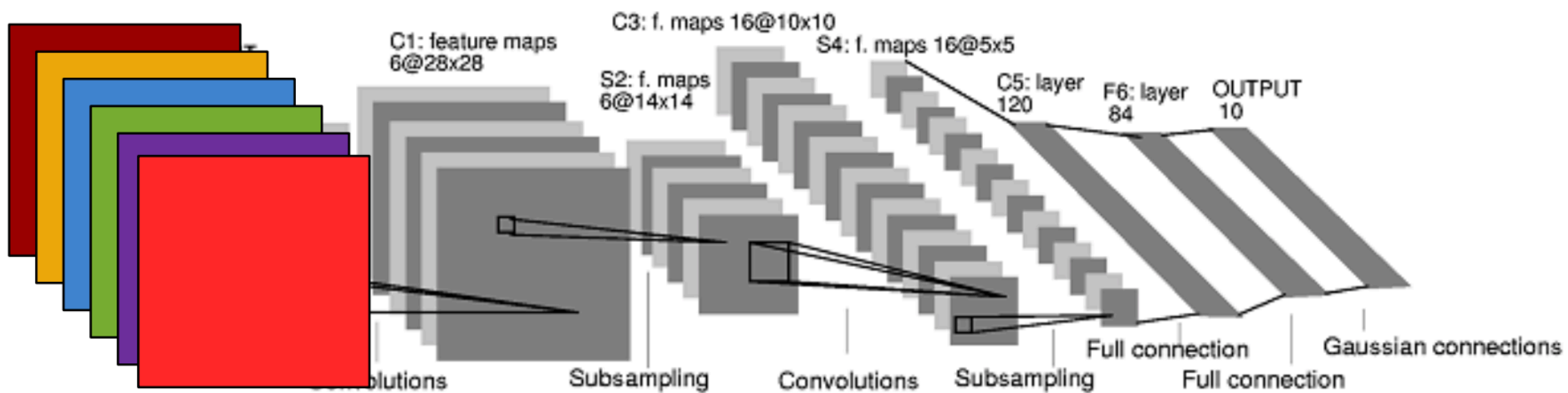


Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

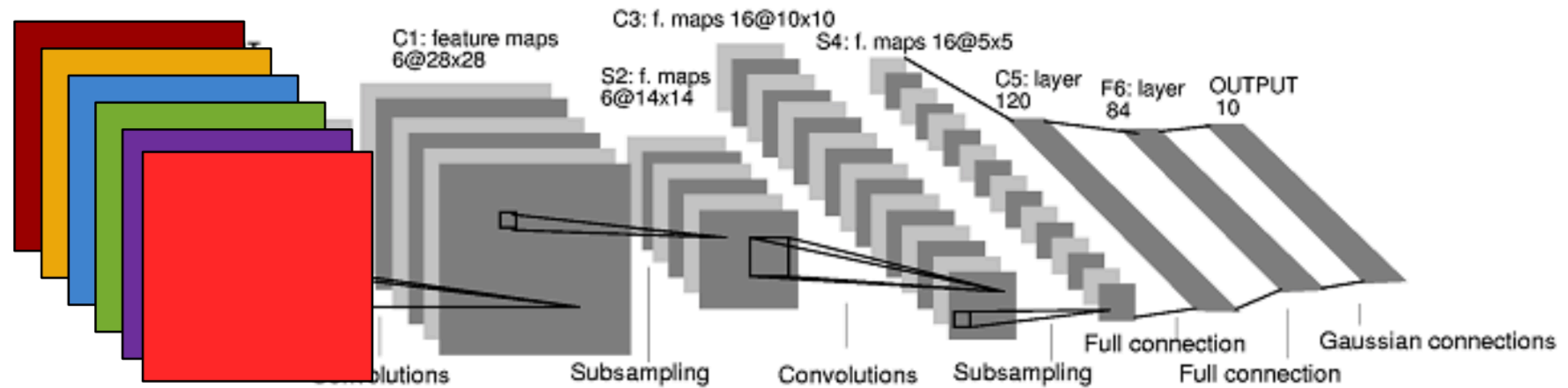
Parallelization

**To go faster, use more
processors**

Lots of parallelism in a DNN

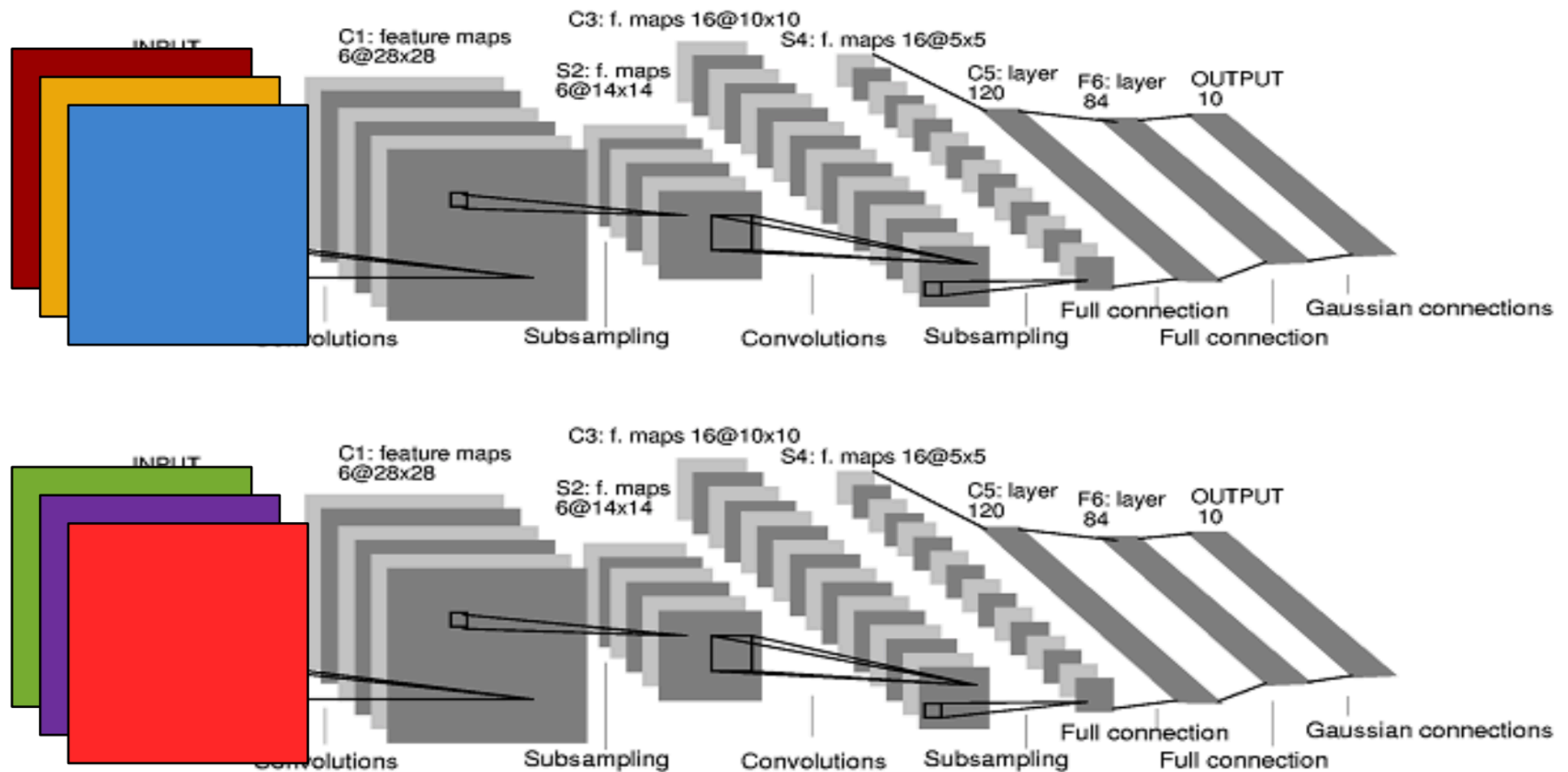


Lots of parallelism in a DNN

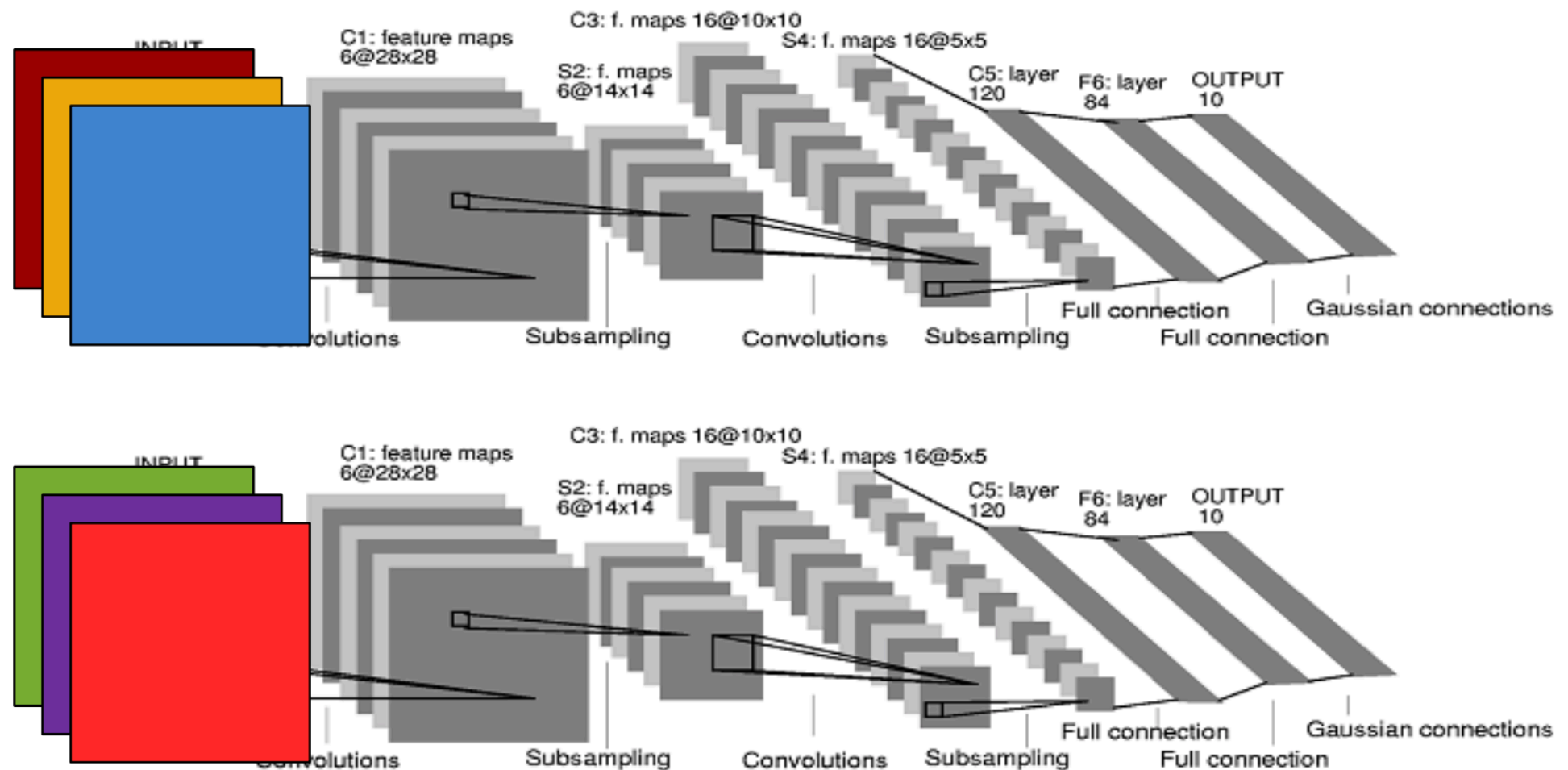


- Inputs
- Points of a feature map
- Filters
- Elements within a filter
- Multiplies within layer are independent
- Sums are reductions
- Only layers are dependent
- No data dependent operations
=> can be statically scheduled

Data Parallel – Run multiple inputs in parallel

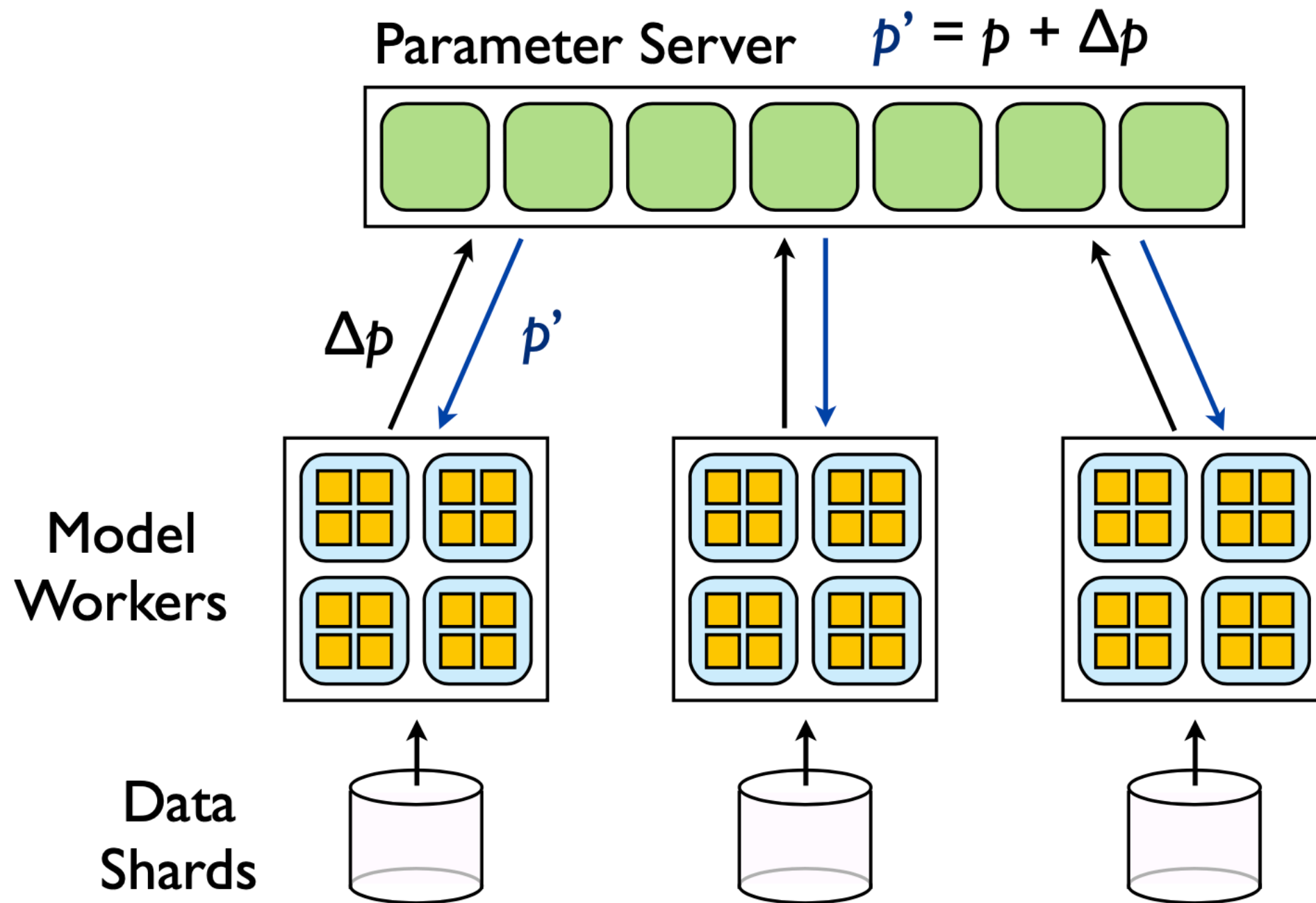


Data Parallel – Run multiple inputs in parallel



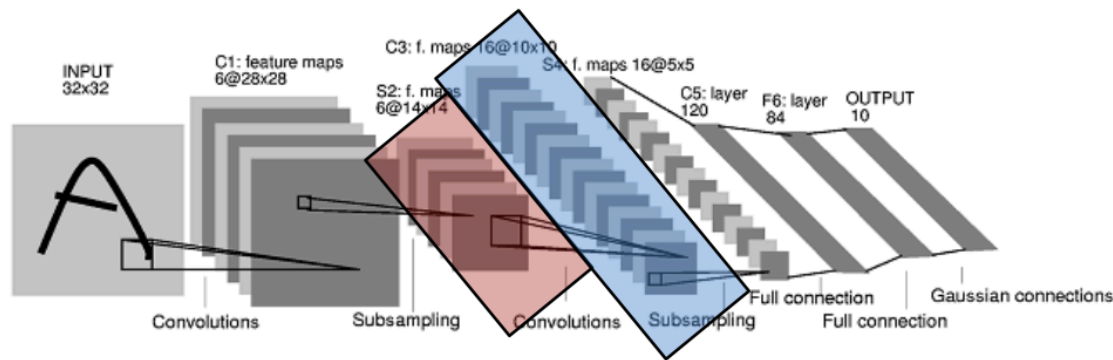
- Doesn't affect latency for one input
- Requires P-fold larger batch size
- For training requires coordinated weight update

Parameter Update



Model Parallel
Split up the Model – i.e.
the network

Model-Parallel Convolution



6D Loop

For each output map j

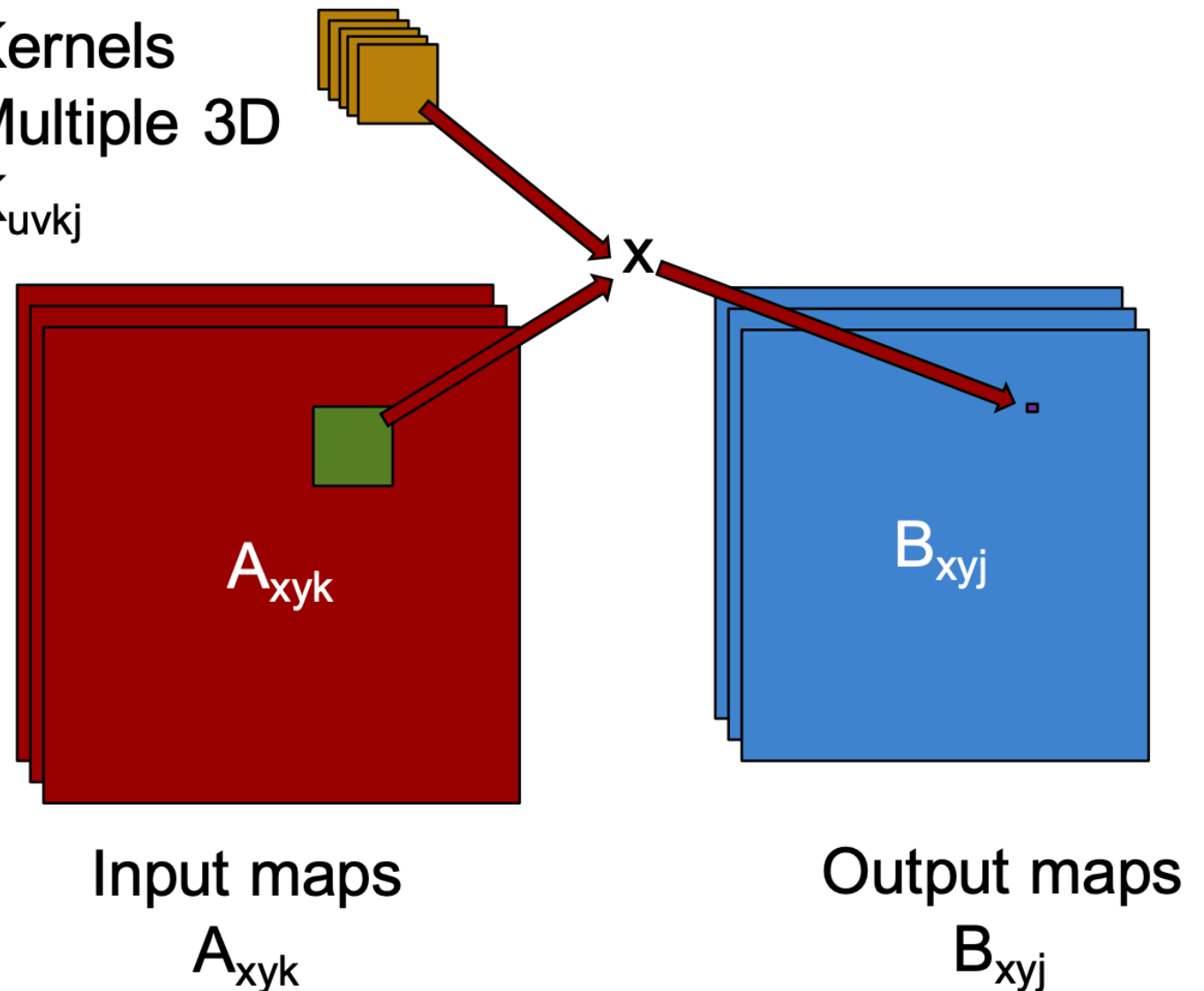
For each input map k

For each pixel x,y

For each kernel element u,v

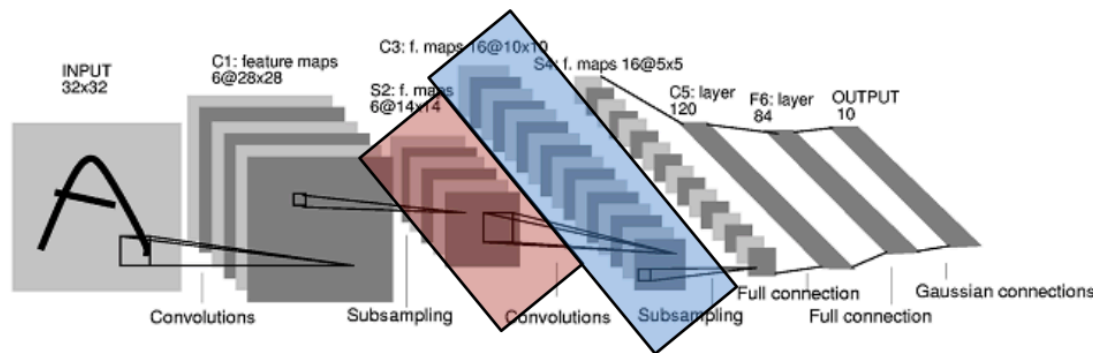
$$B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}$$

Kernels
Multiple 3D
 K_{uvkj}



Model-Parallel Convolution

– by output region (x,y)



Kernels
Multiple 3D
 K_{uvkj}



$\times$

6D Loop

Forall region XY

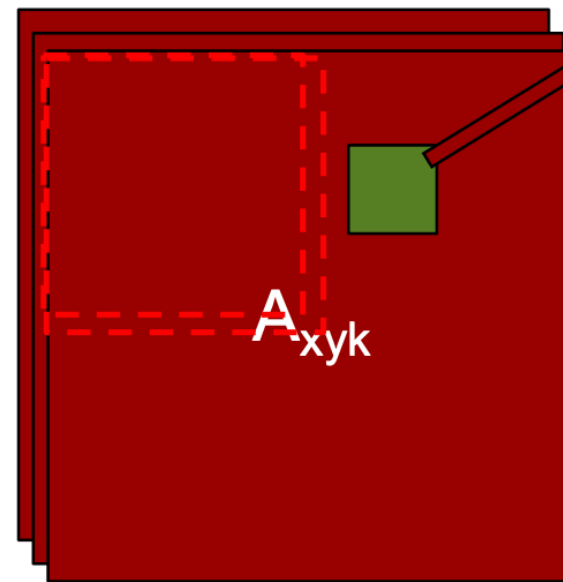
For each output map j

For each input map k

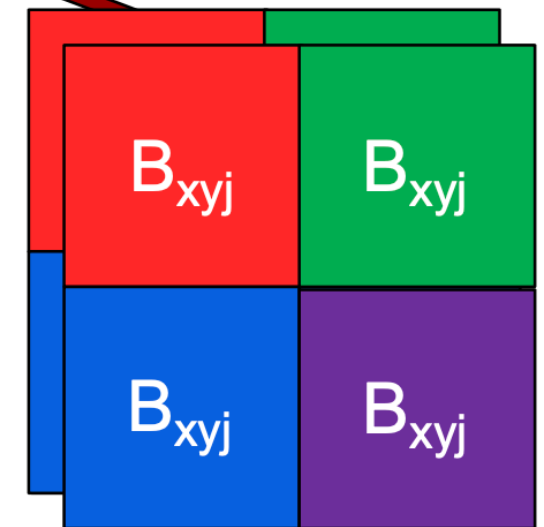
For each pixel x,y in XY

For each kernel element u,v

$$B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}$$



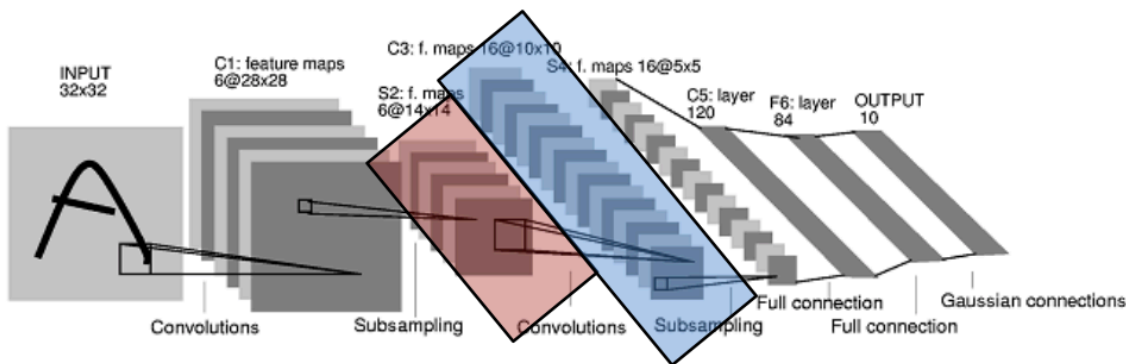
Input maps
 A_{xyk}



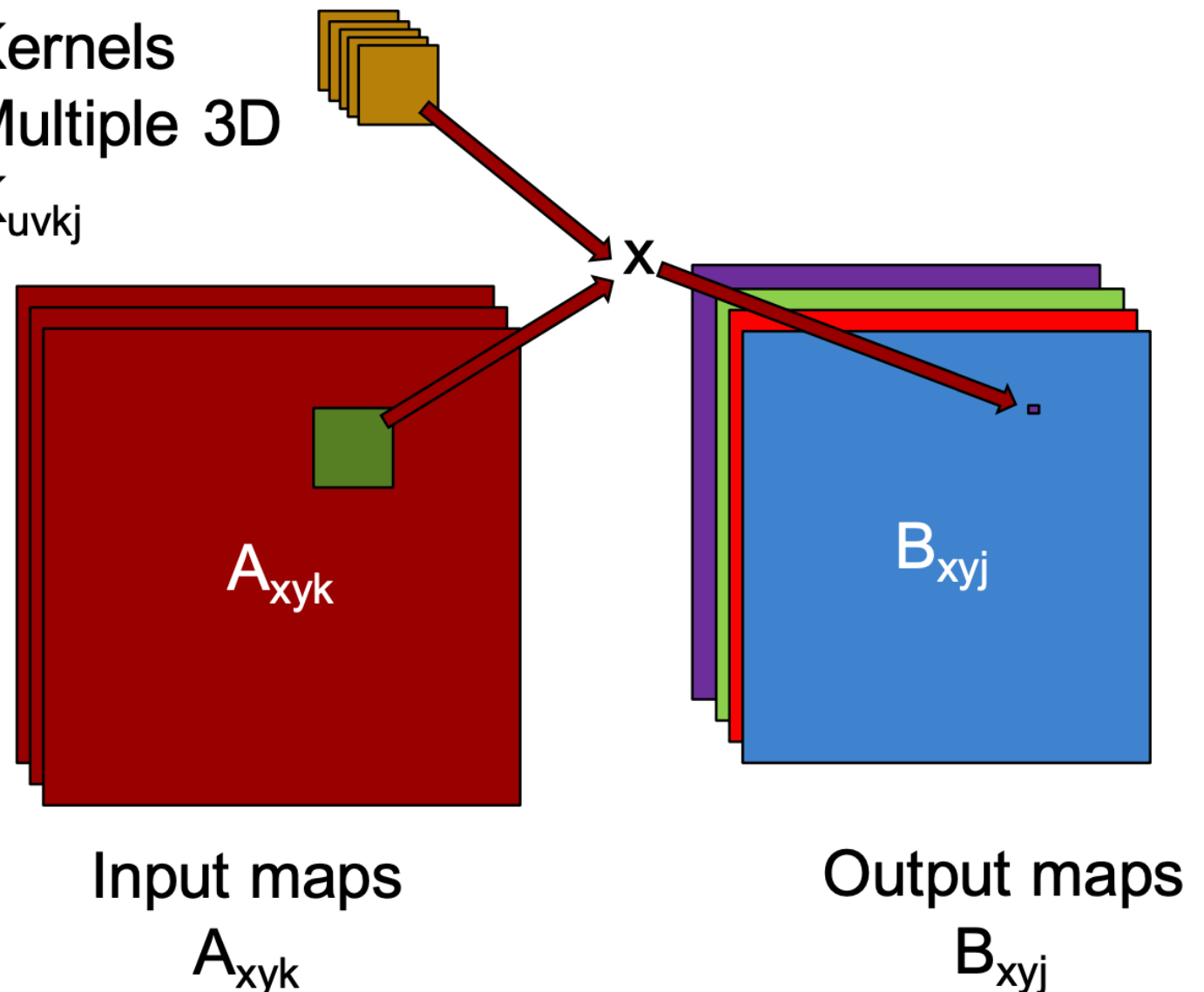
Output maps
 B_{xyj}

Model-Parallel Convolution

– By output map j (filter)



Kernels
Multiple 3D
 K_{uvkj}



6D Loop

Forall output map j

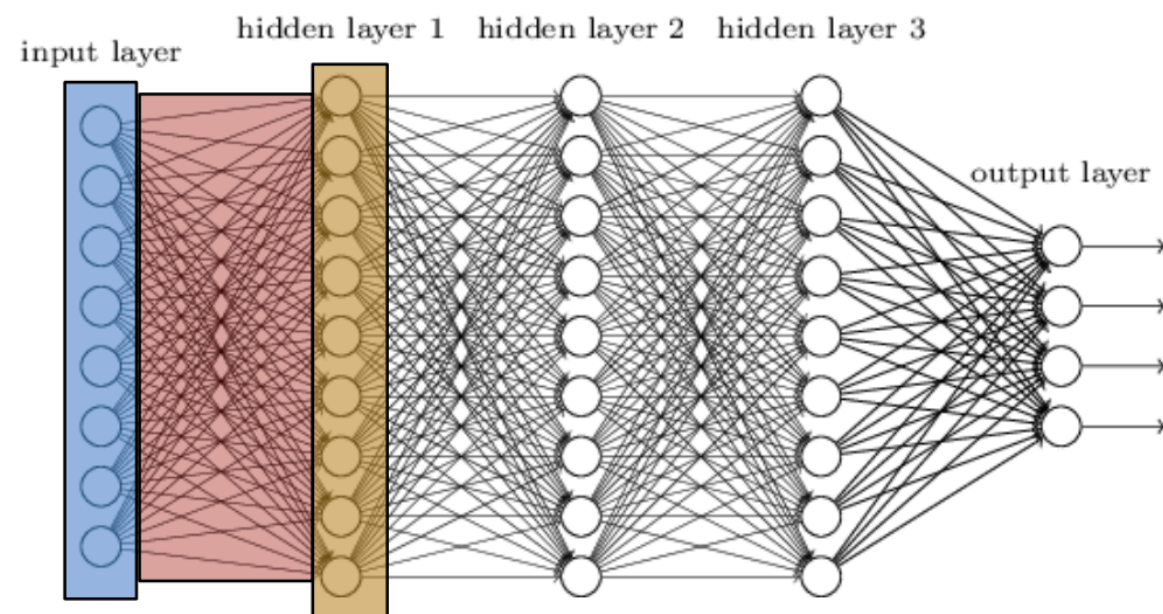
For each input map k

For each pixel x,y

For each kernel element u,v

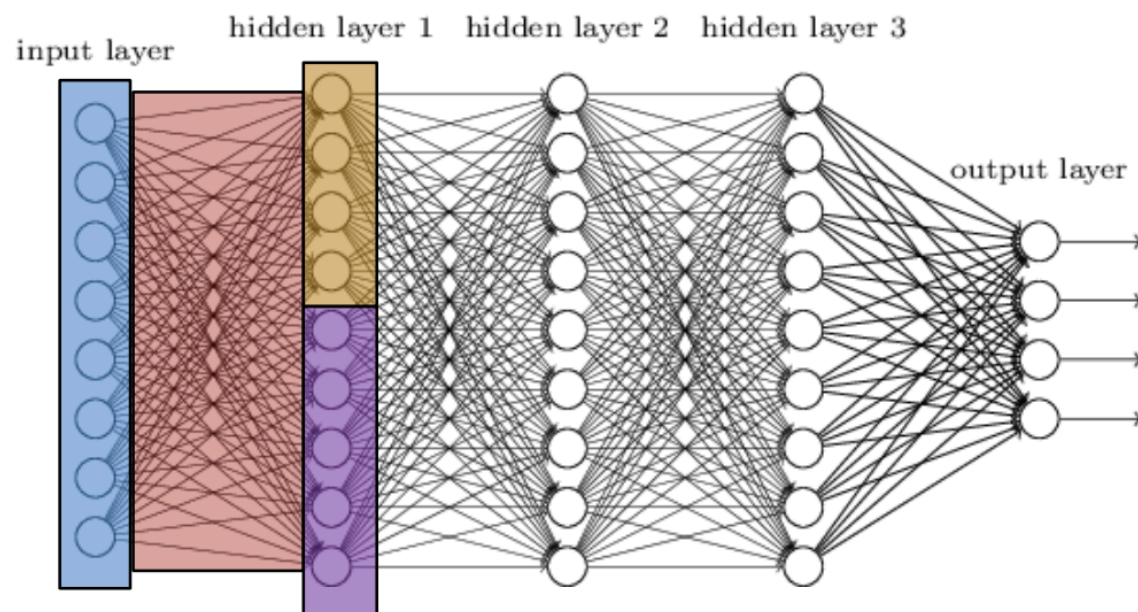
$$B_{xyj} += A_{(x-u)(y-v)k} \times K_{uvkj}$$

Model Parallel Fully-Connected Layer ($M \times V$)



$$\begin{array}{c} \text{Output activations} \\ \mathbf{b}_i \end{array} = \begin{array}{c} \mathbf{W}_{ij} \\ \text{weight matrix} \end{array} \times \begin{array}{c} \text{Input activations} \\ \mathbf{a}_j \end{array}$$

Model Parallel Fully-Connected Layer ($M \times V$)



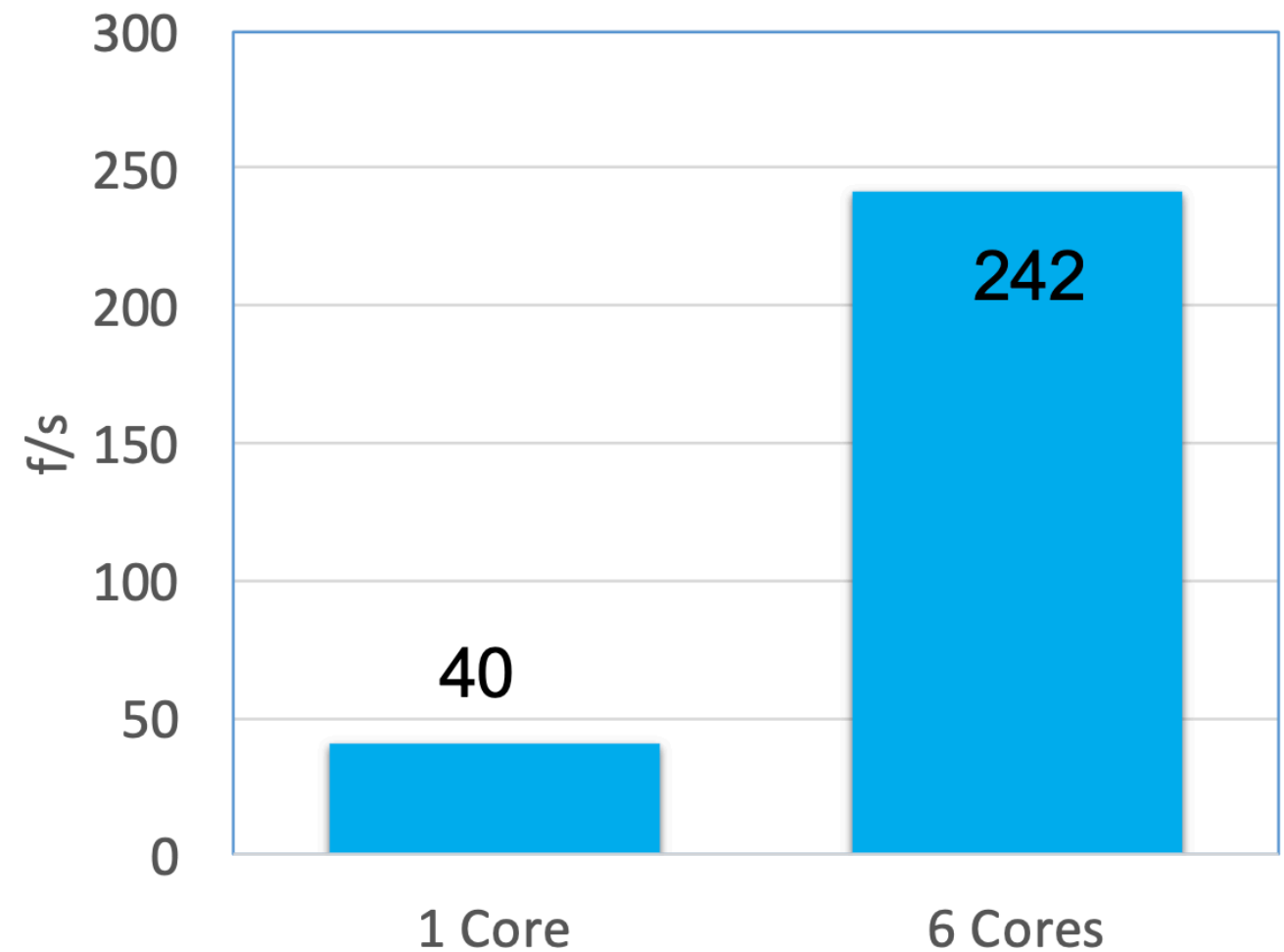
$$\begin{array}{c} \text{Output activations} \\ \begin{array}{|c|} \hline b_i \\ \hline b_i \\ \hline \end{array} \end{array} = \begin{array}{|c|} \hline W_{ij} \\ \hline W_{ij} \\ \hline \end{array} \times \begin{array}{|c|} \hline a_j \\ \hline \end{array}$$

weight matrix

Input activations

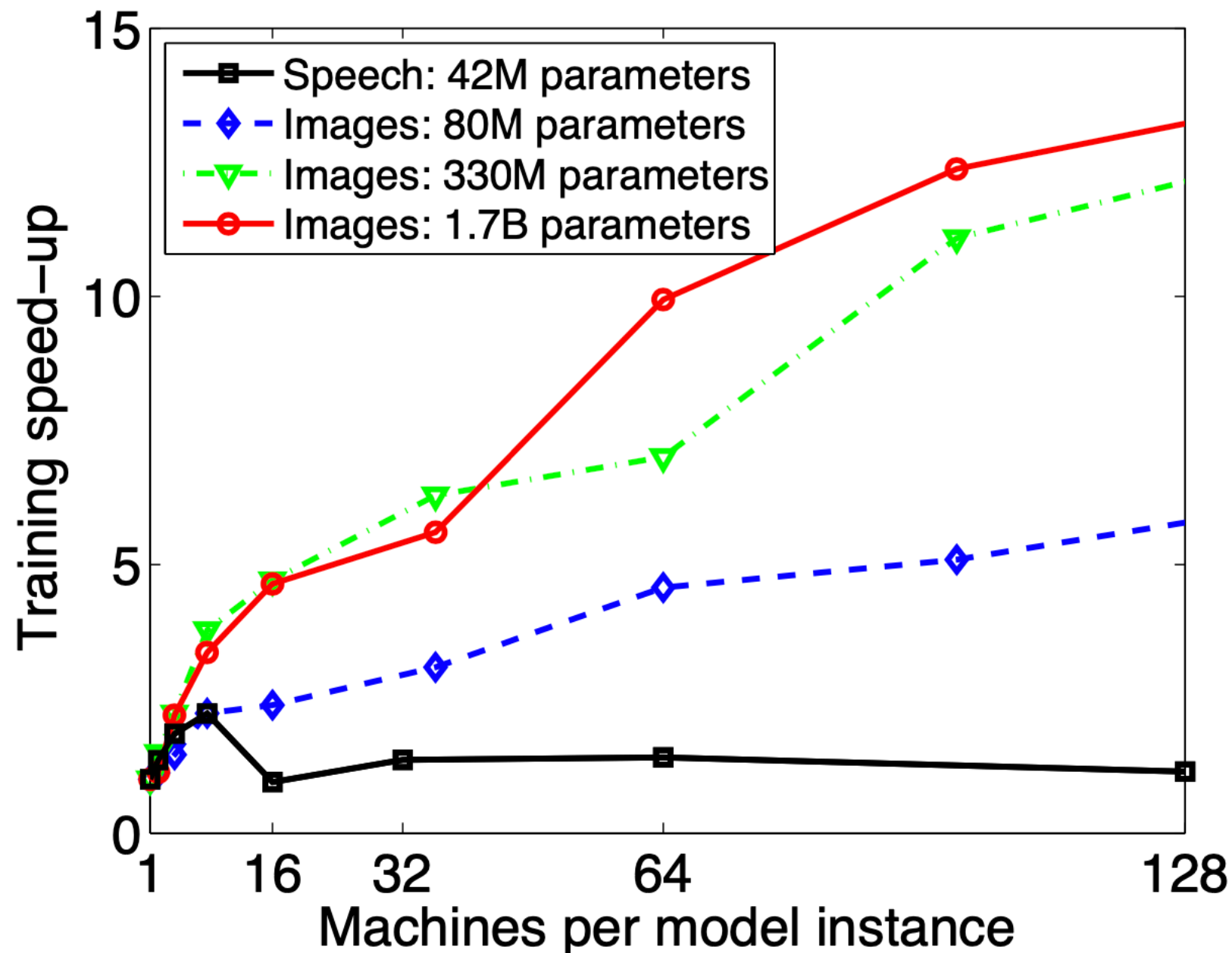
Hyper-Parameter Parallel
Try many alternative
networks in parallel

CPU Parallelism – Core i7 – 1 core vs 6 cores



NVIDIA, "Whitepaper: GPU-based deep learning inference: A performance and power analysis."

Data and Model Parallel Performance



Summary of Parallelism

- Lots of parallelism in DNNs
 - 16M independent multiplies in one FC layer
 - Limited by overhead to exploit a fraction of this
- Data parallel
 - Run multiple training examples in parallel
 - Limited by batch size
- Modelparallel
 - Split model over multiple processors
 - By layer
 - Conv layers by map region
 - Fully connected layers by output activation
- Easy to get 16-64 GPUs training one model in parallel

GPUs

- To go fast, use multiple processors

- To go fast, use multiple processors
- To be efficient and fast, use GPUs

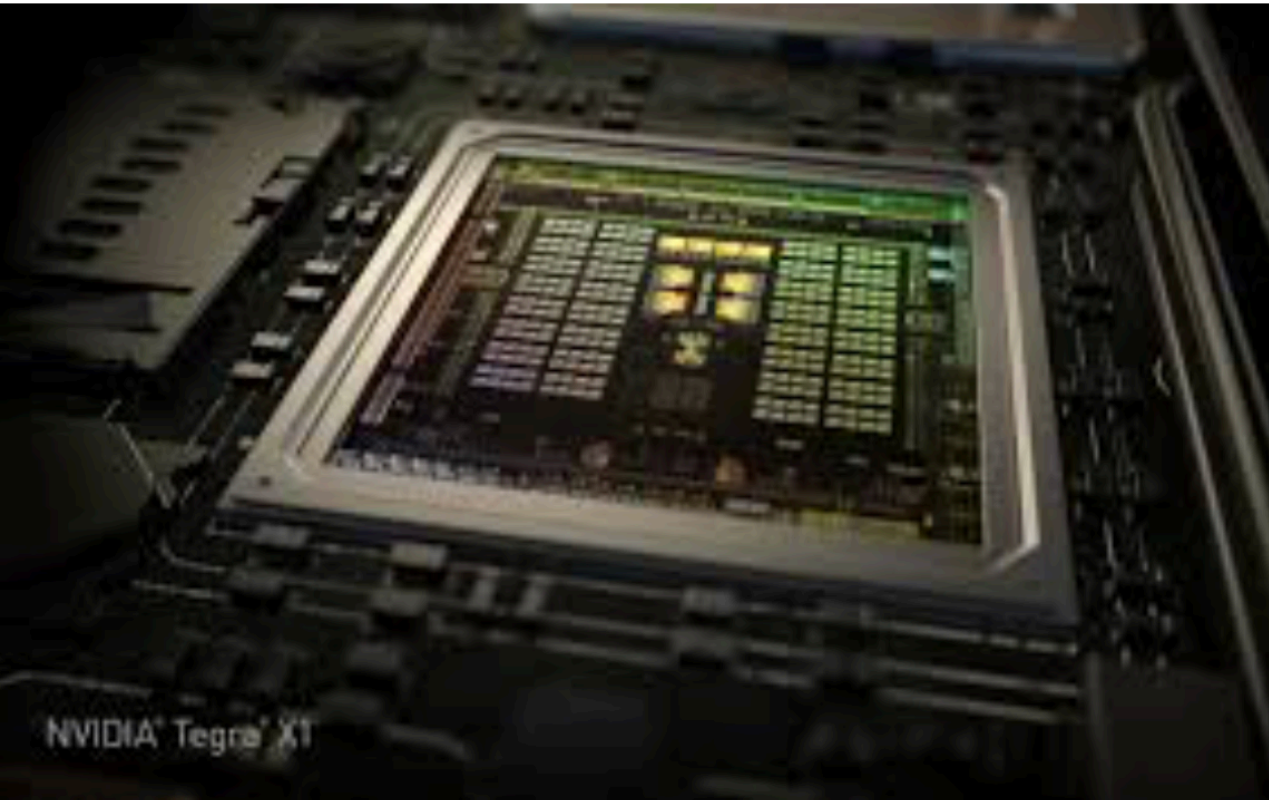
- To go fast, use multiple processors
- To be efficient and fast, use GPUs
- To be efficient and go really fast, use multiple GPUs

Titan X



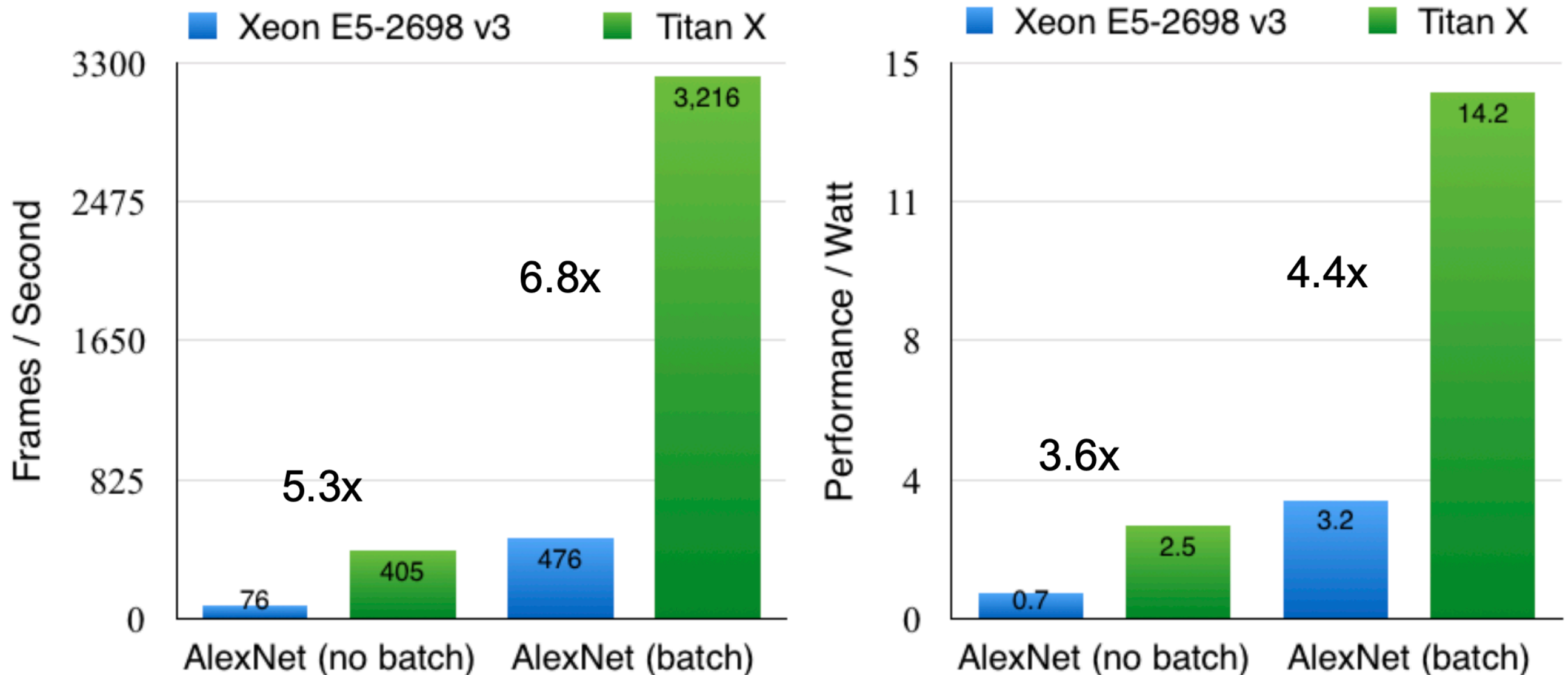
- 3072 CUDA cores @ 1 GHz
- 6 Teraflops FP32
- 12GB of GDDR5 @ 336 GB/sec
- 250W TDP
- 24GFLOPS/W
- 28nm process

Tegra X1



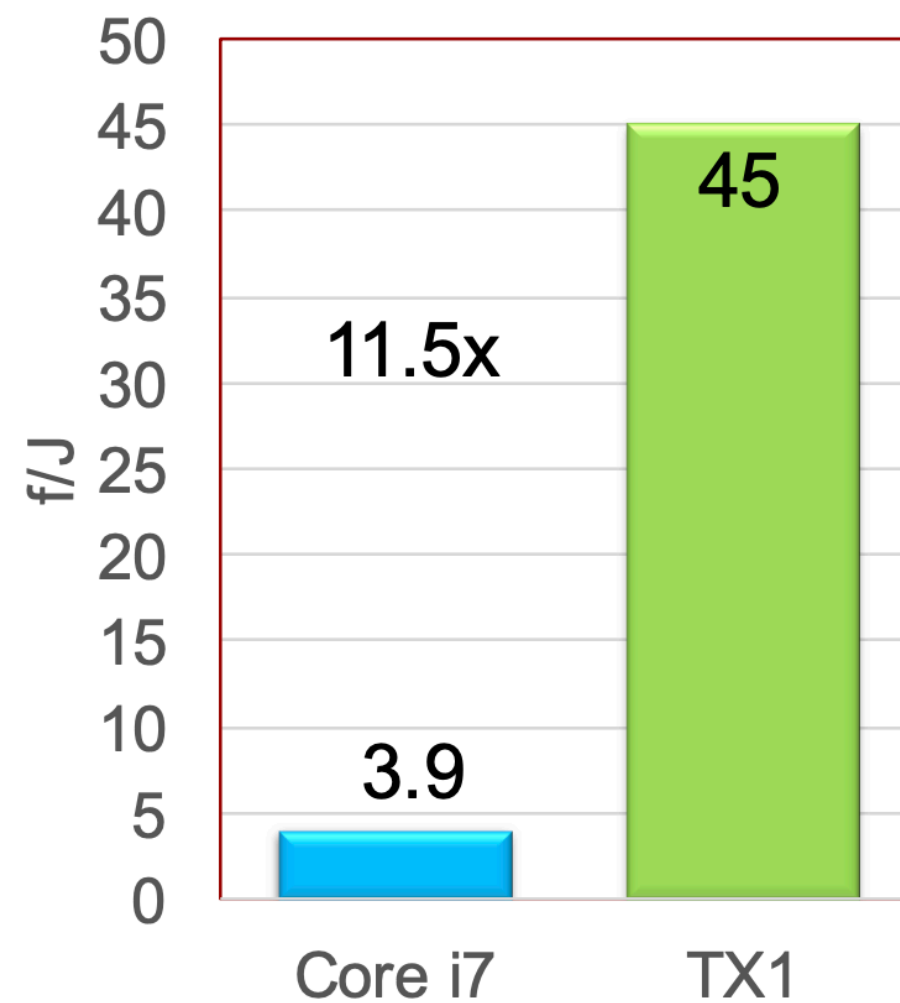
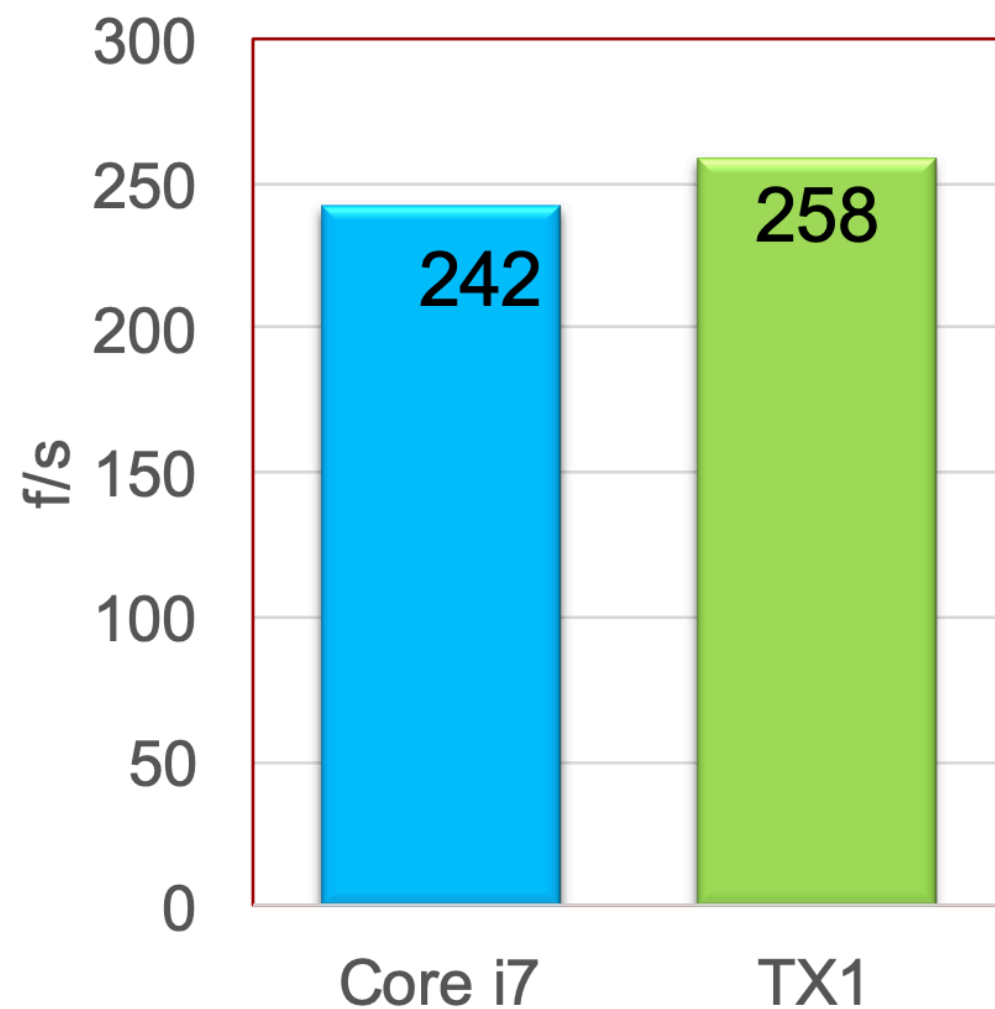
- 256 CUDA cores @ ~1 GHz
- 1 Teraflop FP16
- 4GB of LPDDR4 @ 25.6 GB/s
- 15 W TDP (1W idle, <10W typical)
- 100GFLOPS/W (FP16)
- 20nm process

Xeon E5-2698 CPU v.s. TitanX GPU



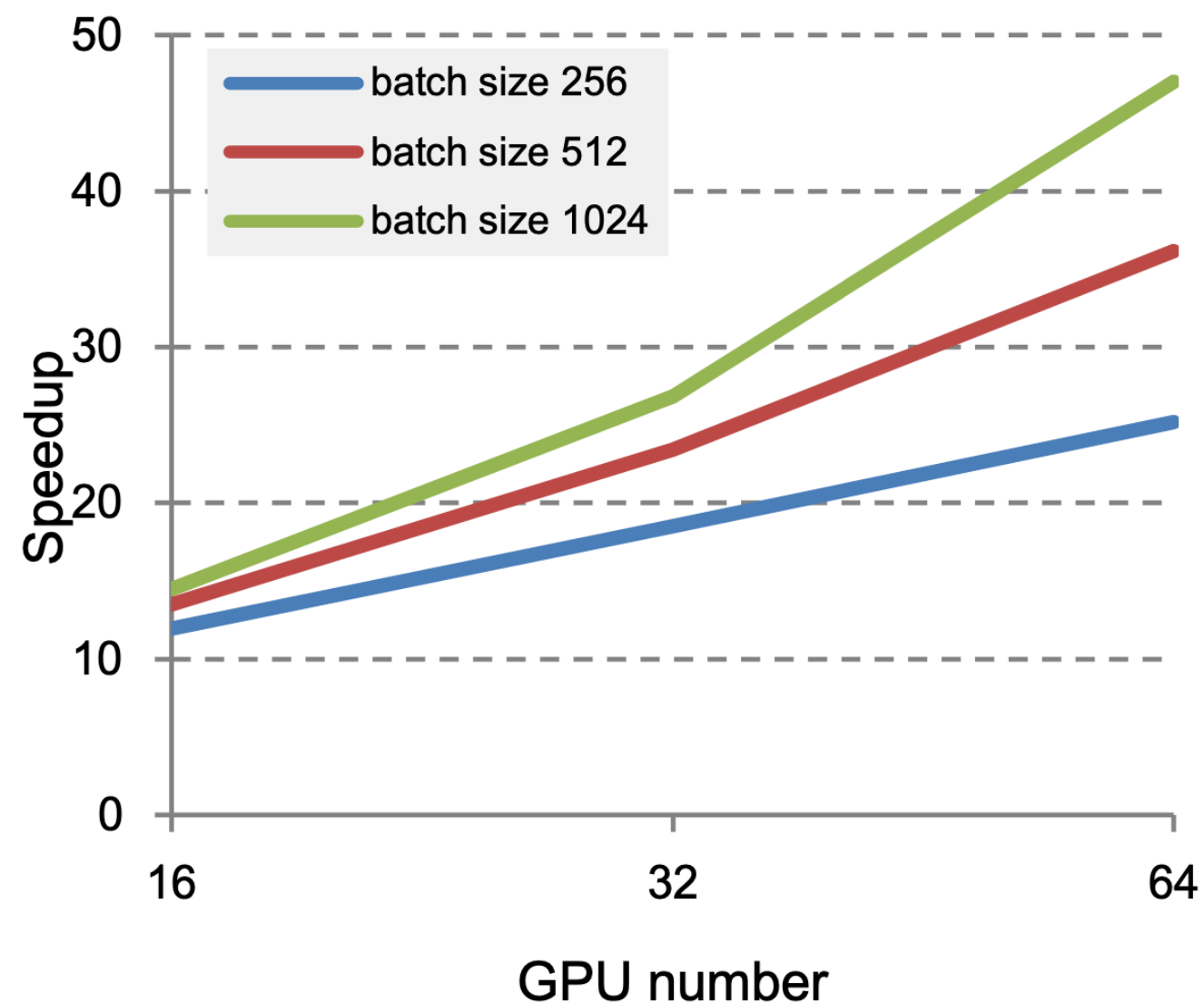
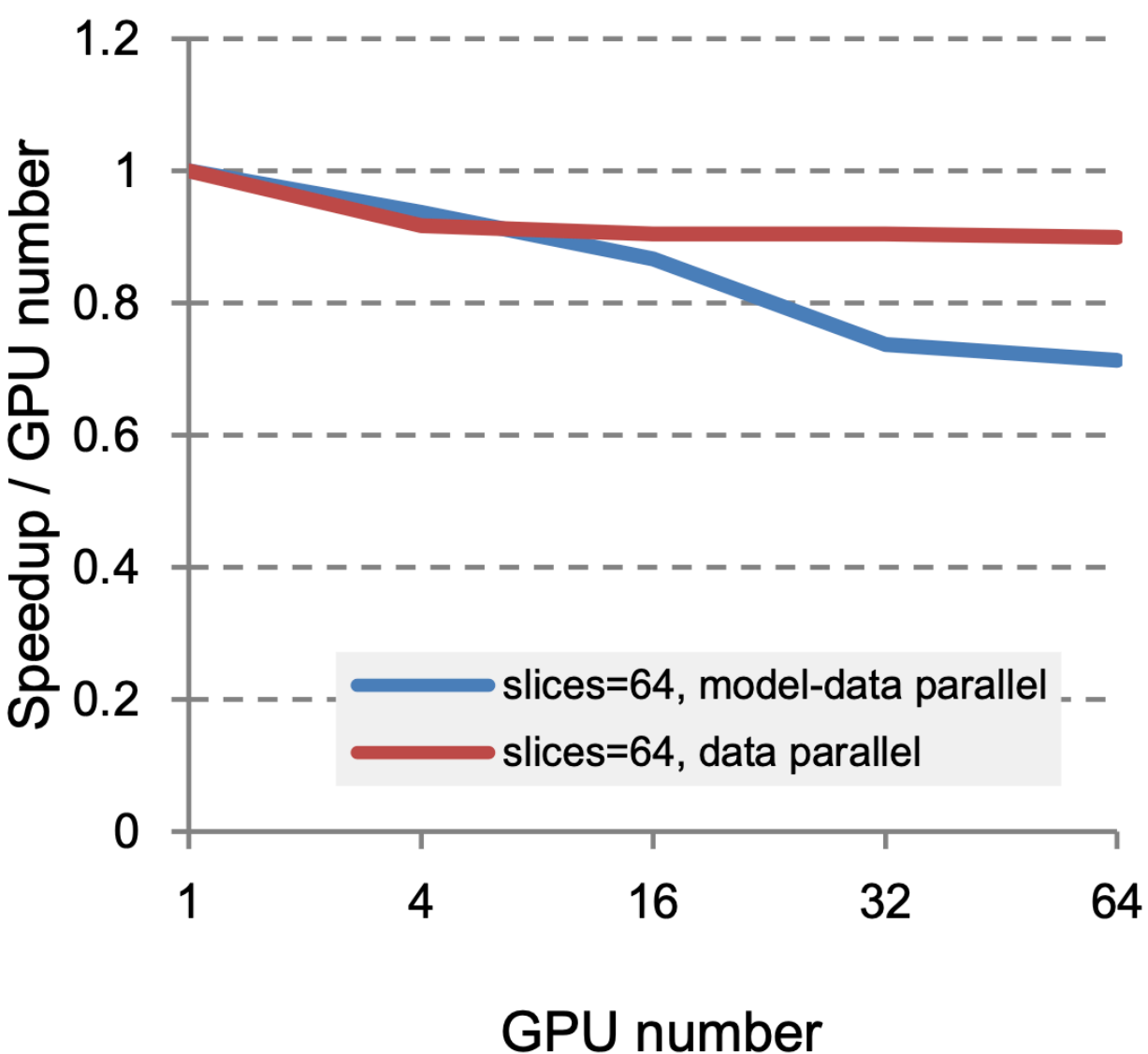
NVIDIA, "Whitepaper: GPU-based deep learning inference: A performance and power analysis."

Tegra X1 vs Core i7

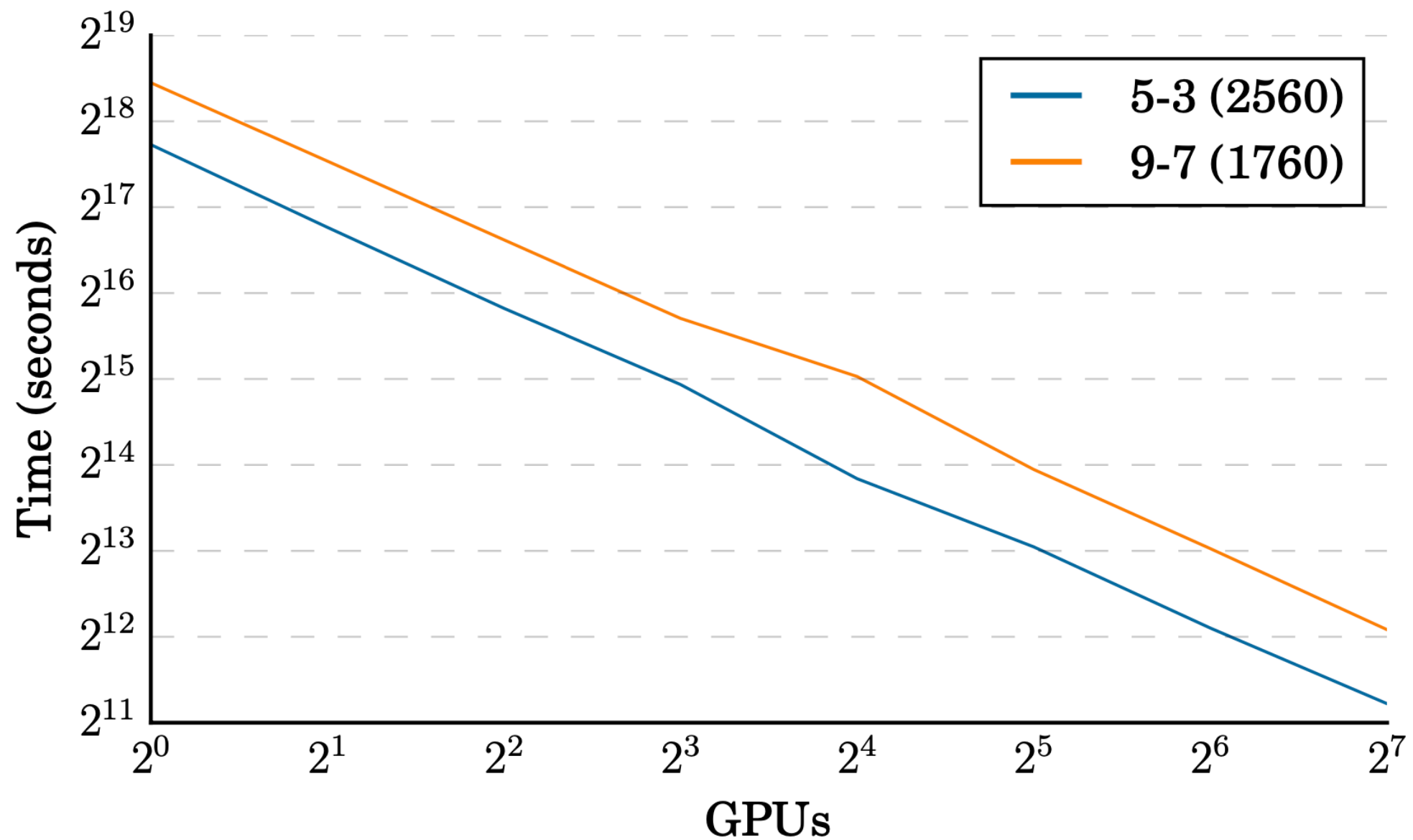


NVIDIA, "Whitepaper: GPU-based deep learning inference: A performance and power analysis."

Parallel GPUs



Parallel GPUs on Deep Speech 2



Summary of GPUs

- Titan X ~6x faster, 4x more efficient than Xeon E5
- TX1 11.5x more efficient than Core i7
- On inference
- Larger gains on training

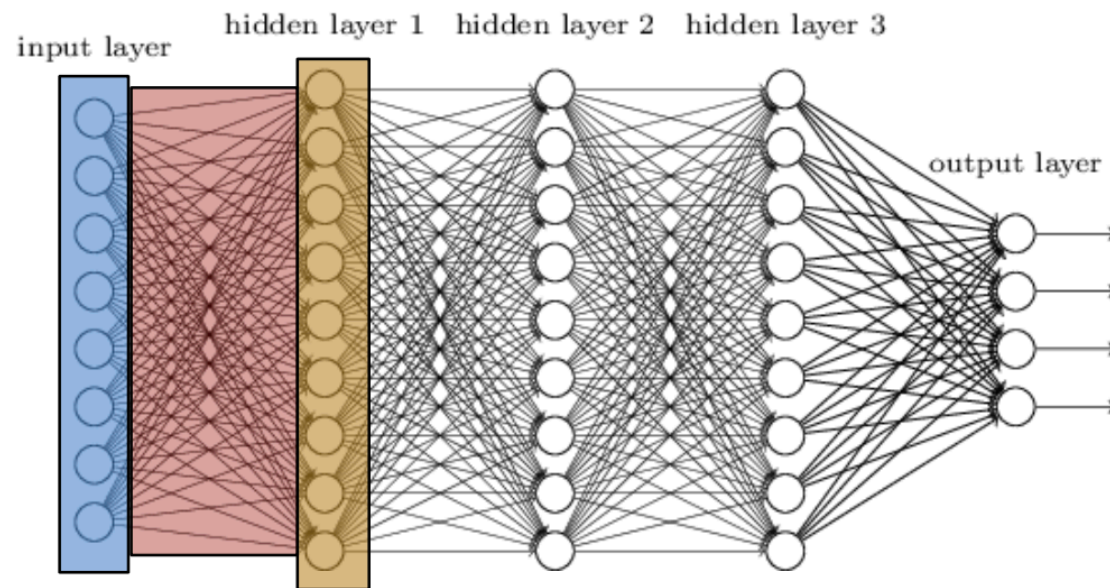
- Data parallelism scales easily to 16GPUs
- With some effort, linear speedup to 128GPUs

Reduced Precision

Benefits

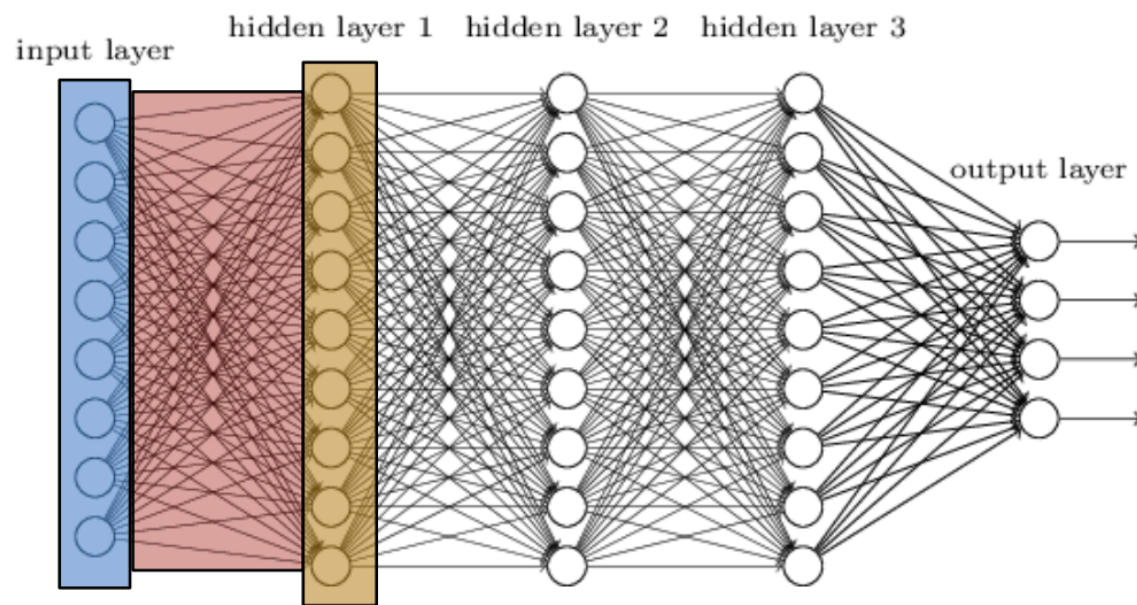
- Reducing precision
- Reduces storage
- Reduces energy
- Improves performance
- Has little effect on accuracy – to a point

DNN, key operation is dense $M \times V$

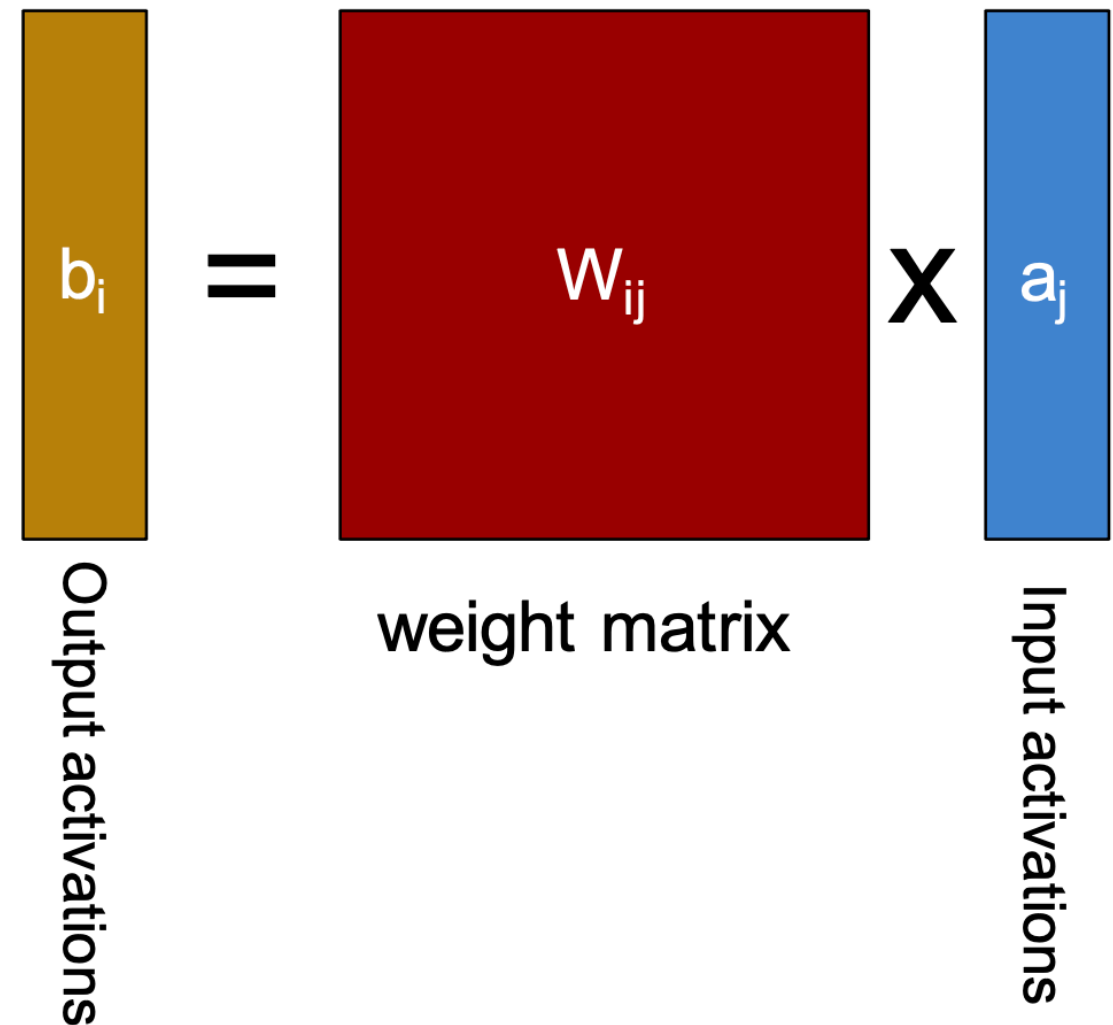


$$\begin{array}{c} \text{Output activations} \\ b_i \end{array} = \begin{array}{c} \text{weight matrix} \\ W_{ij} \end{array} \times \begin{array}{c} \text{Input activations} \\ a_j \end{array}$$

DNN, key operation is dense $M \times V$



$$b_i = f\left(\sum_j w_{ij} a_j\right)$$

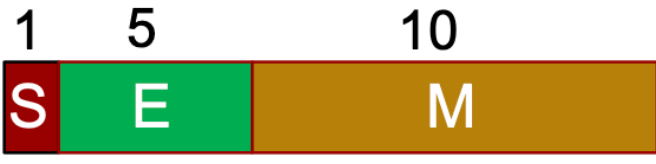
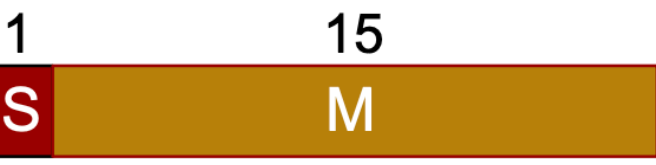


How much accuracy do we need in the computations:

$$b_i = f \left(\sum_j w_{ij} a_i \right)$$

$$w_{ij} = w_{ij} + \alpha a_i g_j$$

Number Representation

		Range	Accuracy
FP32		$10^{-38} - 10^{38}$	.000006%
FP16		$6 \times 10^{-5} - 6 \times 10^4$	.05%
Int32		$0 - 2 \times 10^9$	$\frac{1}{2}$
Int16		$0 - 6 \times 10^4$	$\frac{1}{2}$
Int8		$0 - 127$	$\frac{1}{2}$

Cost of Operations

Relative Energy Cost

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640

1 10 100 1000 10000

Relative Area Cost

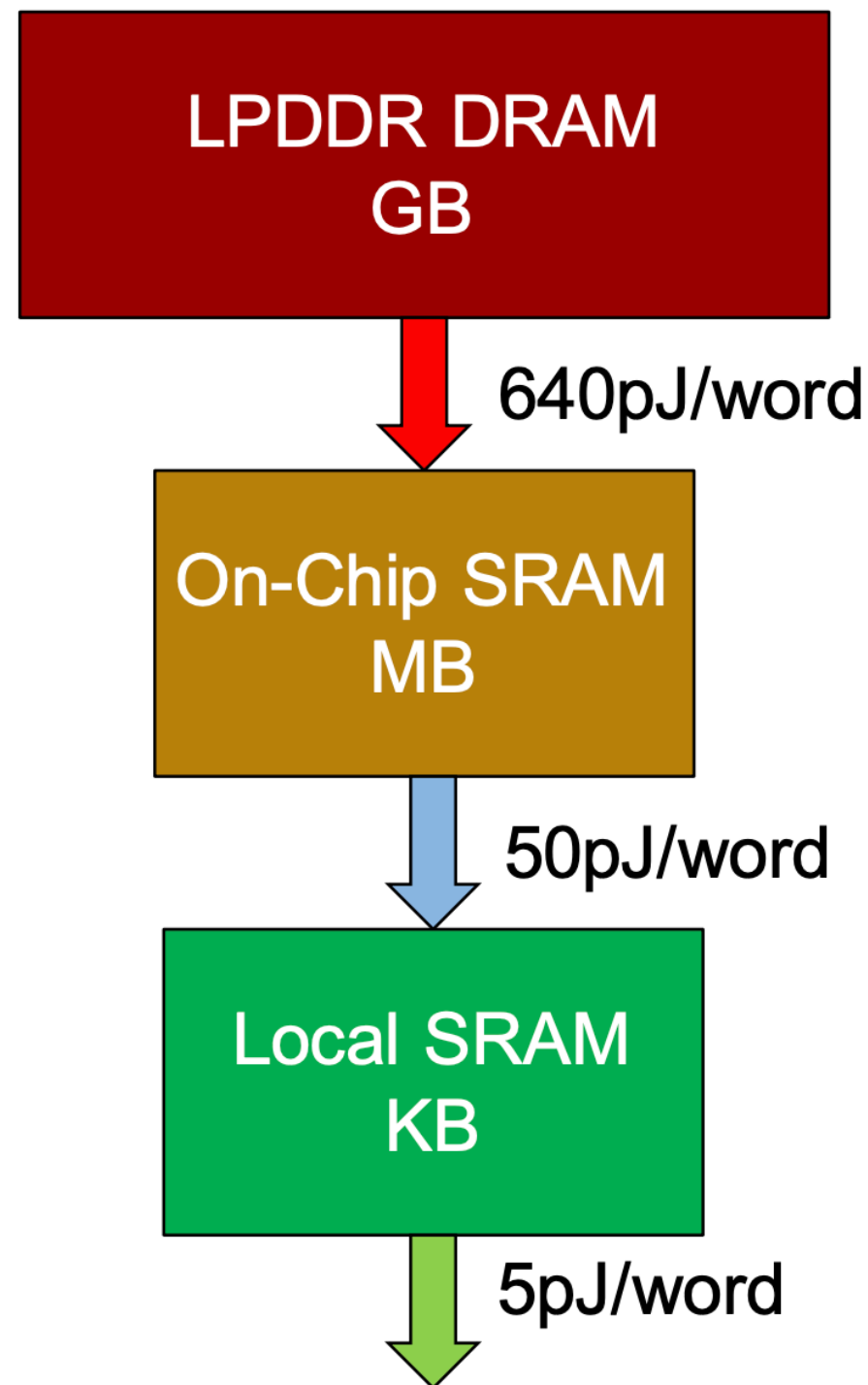
Area (μm^2)
36
67
137
1360
4184
282
3495
1640
7700
N/A
N/A

1 10 100 1000

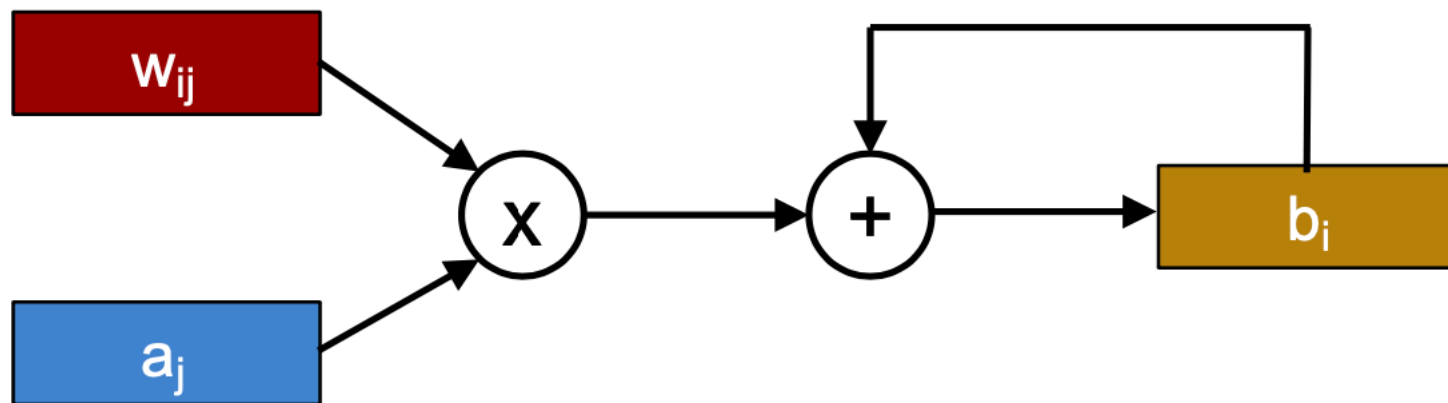
Energy numbers are from Mark Horowitz “Computing’s Energy Problem (and what we can do about it)”, ISSCC 2014

Area numbers are from synthesized result using Design Compiler under TSMC 45nm tech node. FP units used DesignWare Library.

The Importance of Staying Local



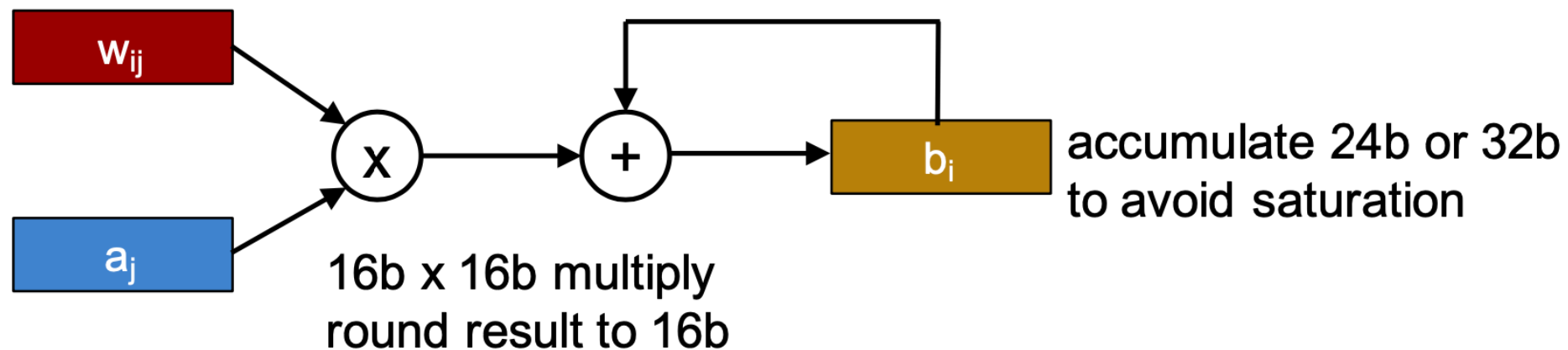
Mixed Precision



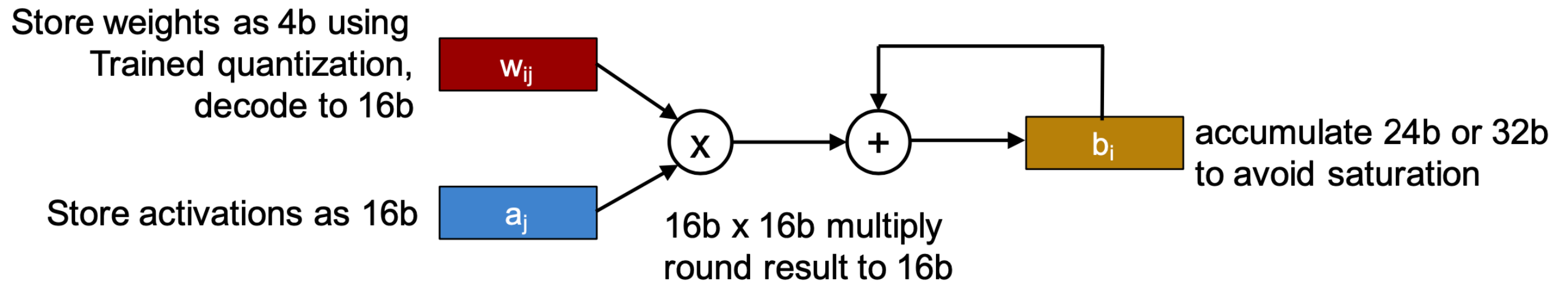
Mixed Precision

Store weights as 4b using
Trained quantization,
decode to 16b

Store activations as 16b

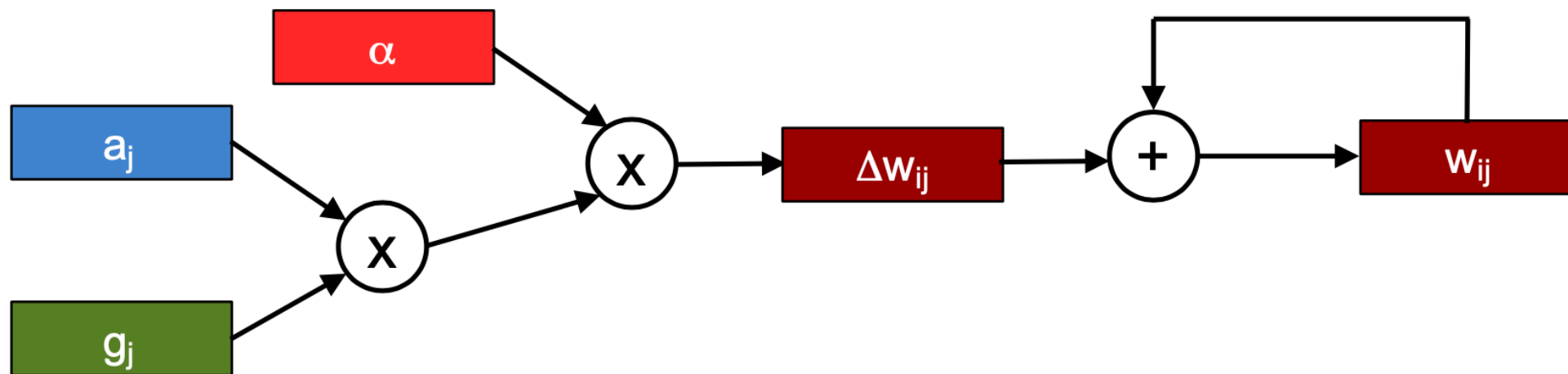


Mixed Precision



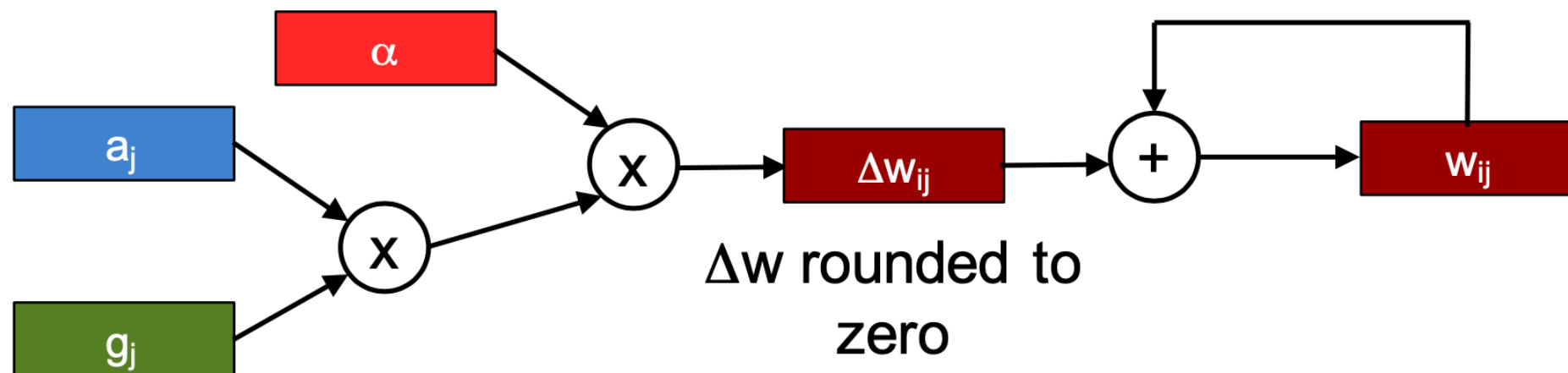
Batch normalization important to 'center' dynamic range

Weight Update



Weight Update

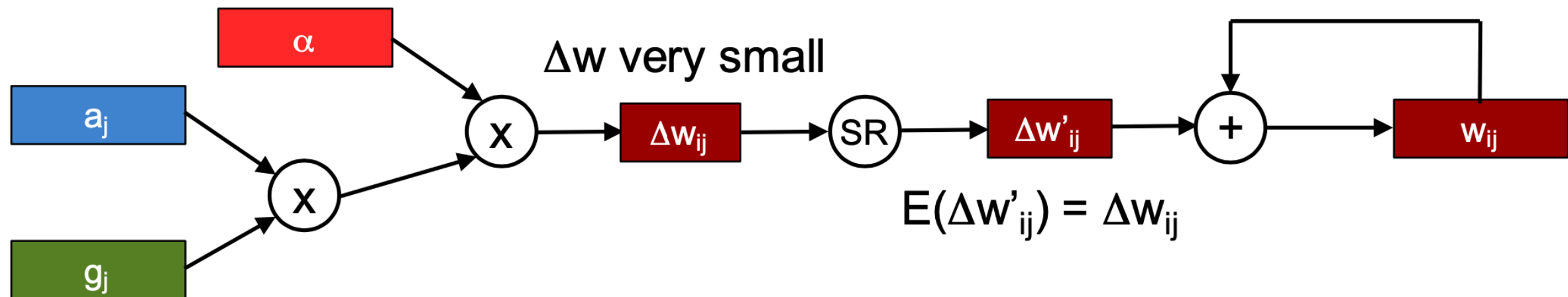
Learning rate may
be very small
(10^{-5} or less)



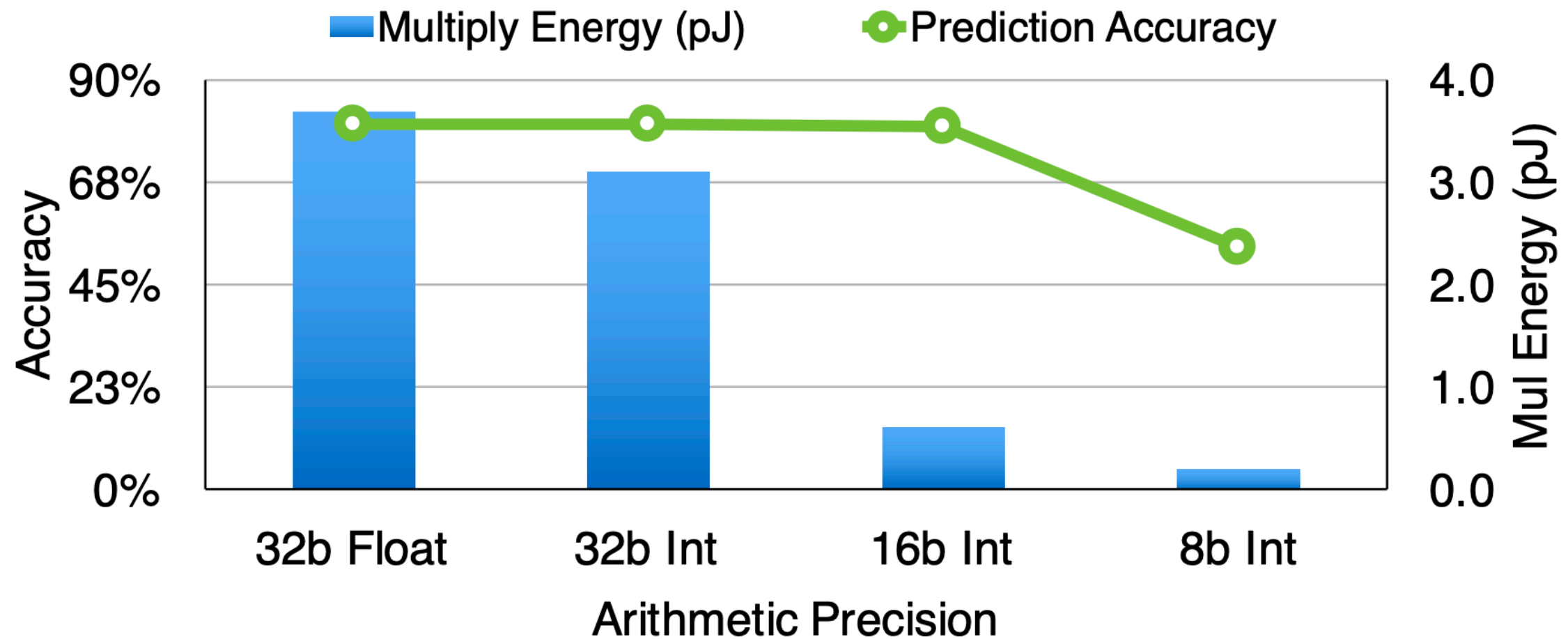
No learning!

Stochastic Rounding

Learning rate may
be very small
(10^{-5} or less)



Reduced Precision for Inference



Reduced Precision For Training

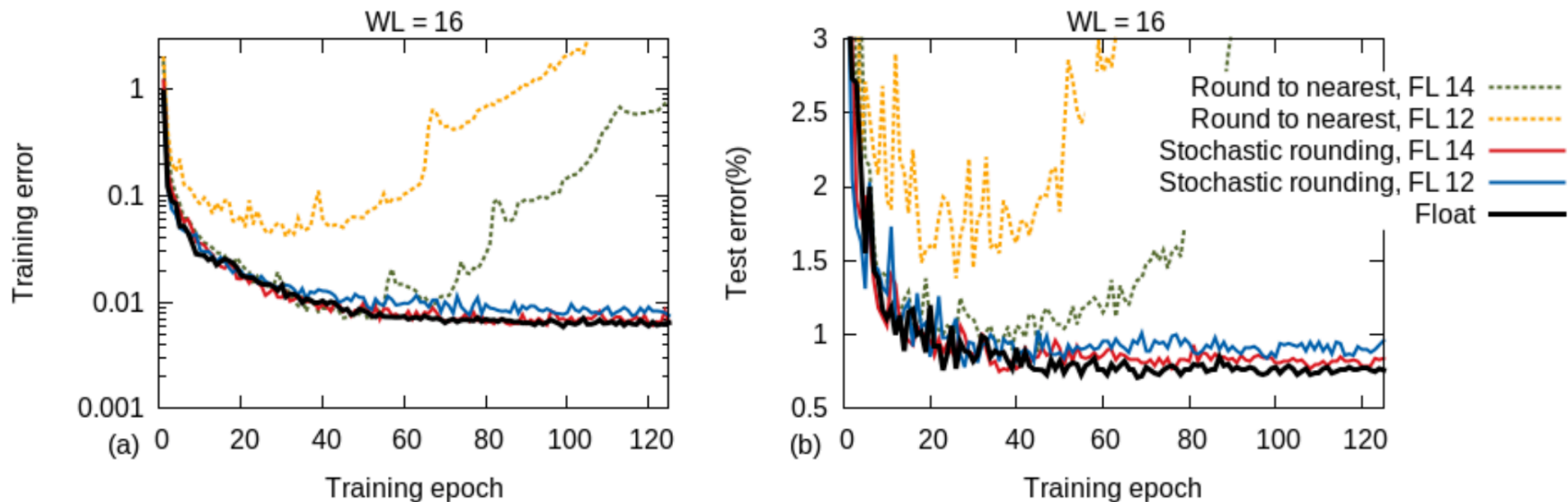


Figure 2. *MNIST dataset using CNNs*: Training error (a) and the test error (b) for training using fixed-point number representation and rounding mode set to either “Round to nearest” or “Stochastic rounding”. The word length for fixed-point numbers WL is kept fixed at 16 bits and results are shown for different fractional (integer) lengths for weights and weight updates: 12(4), and 14(2) bits. Layer outputs use $\langle 6, 10 \rangle$ format in all cases. Results using float are also shown for comparison.

Summary of Reduced Precision

- Can save memory capacity, memory bandwidth, memory power, and arithmetic power by using smaller numbers
- FP16 works with little effort
 - 2x gain in memory, 4x in multiply power
- With care, one can use
 - 8b for convolutions
 - 4b for fully-connected layers
- Batch normalization – important to ‘center’ ranges
- Stochastic rounding – important to retain small increments