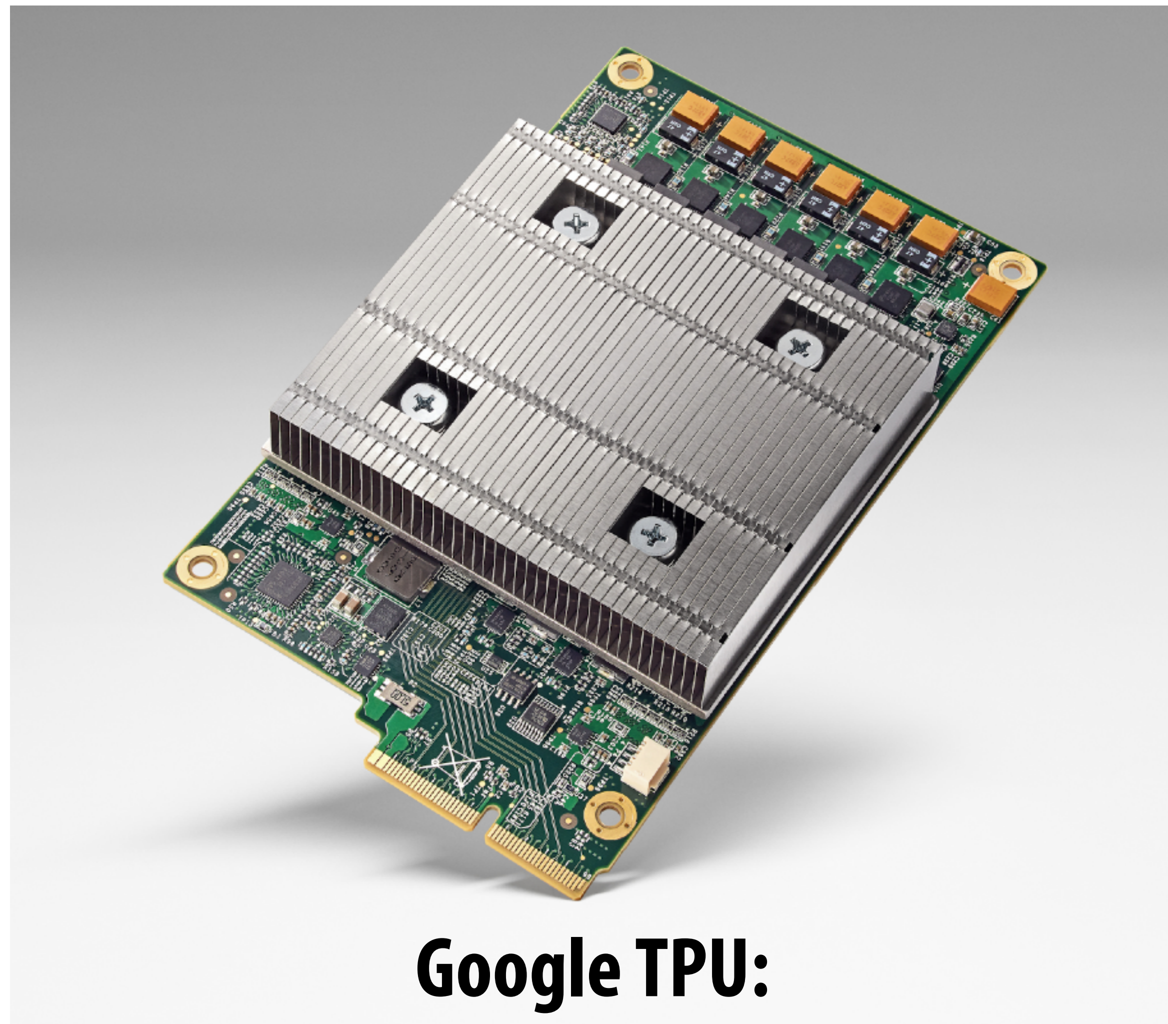
Lecture 10:

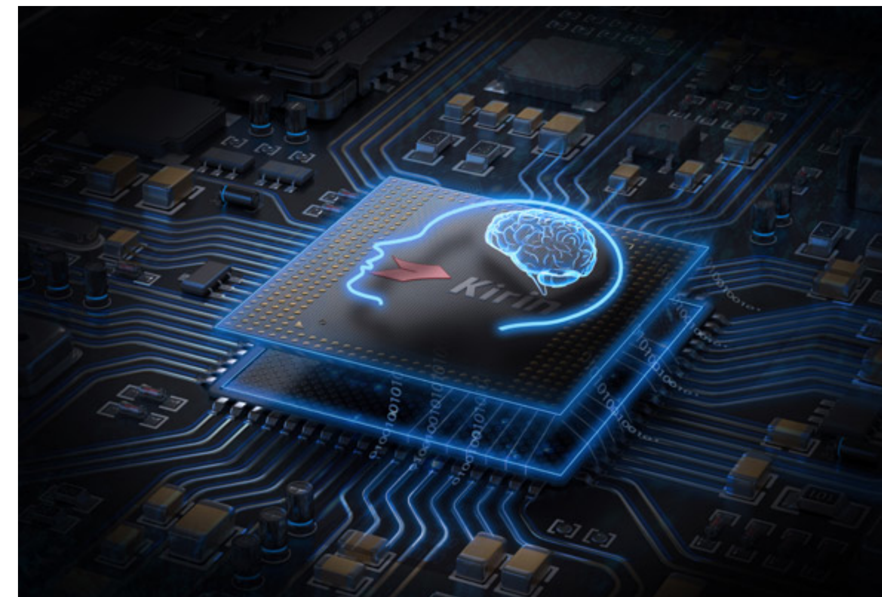
Hardware Acceleration of DNNs

Visual Computing Systems
Stanford CS348K, Fall 2018

Hardware acceleration for DNNs

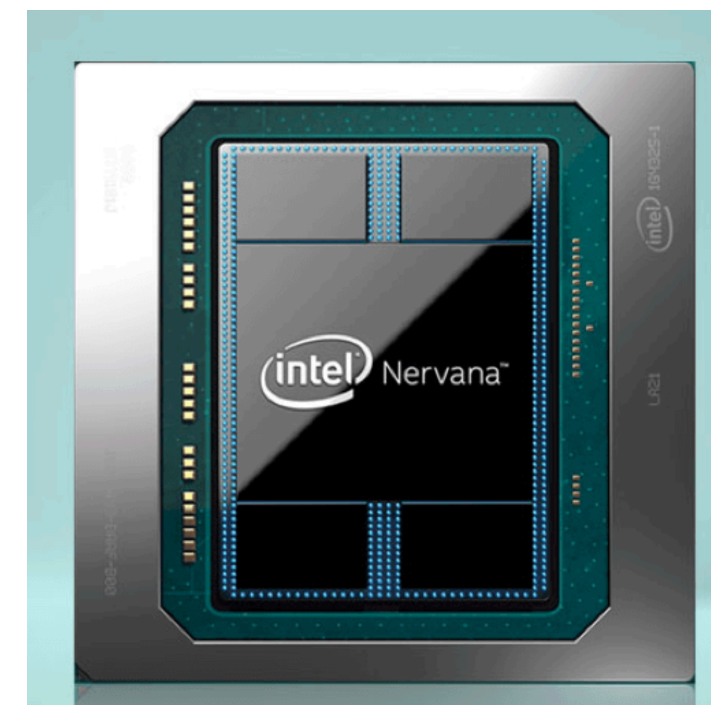


Google TPU:

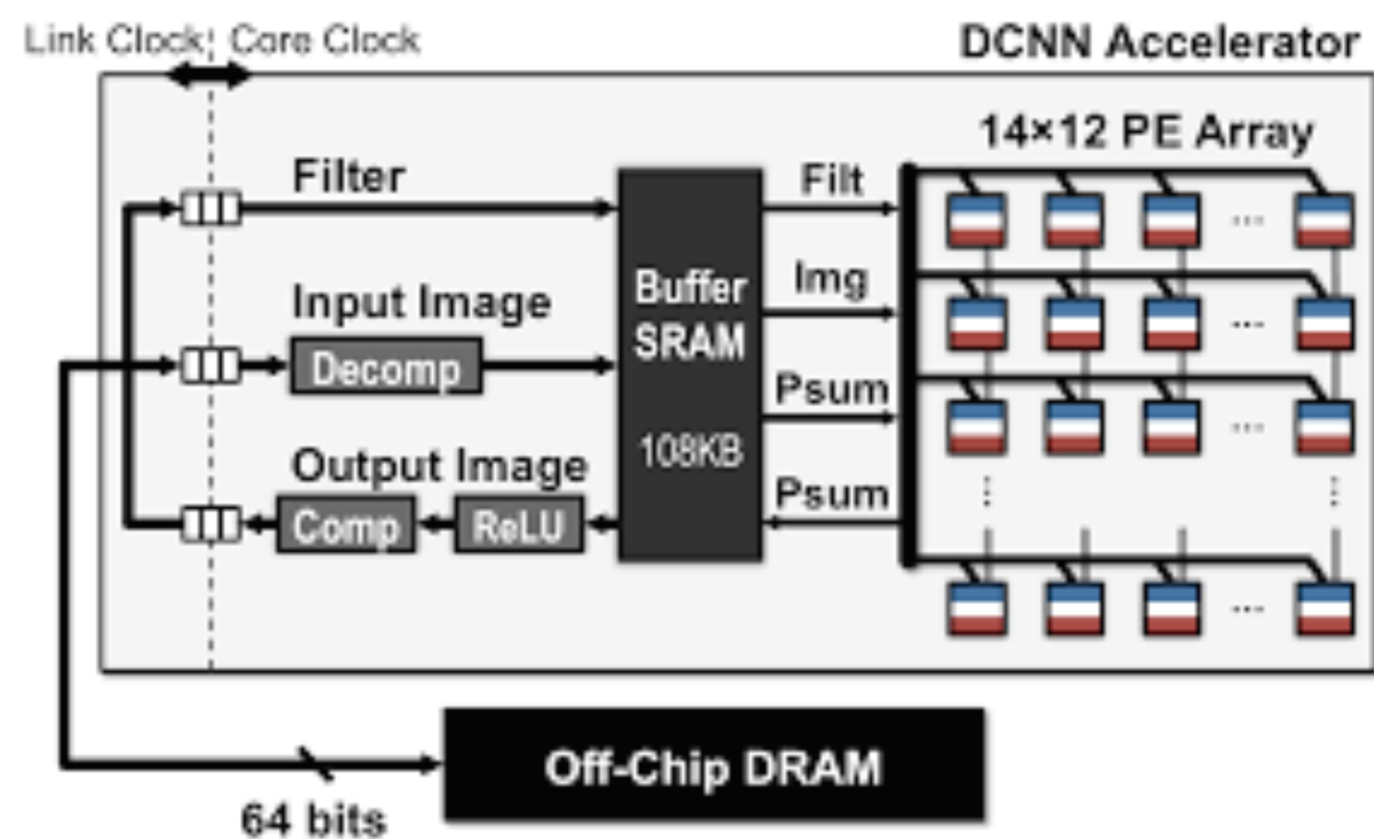


Huawei Kirin NPU

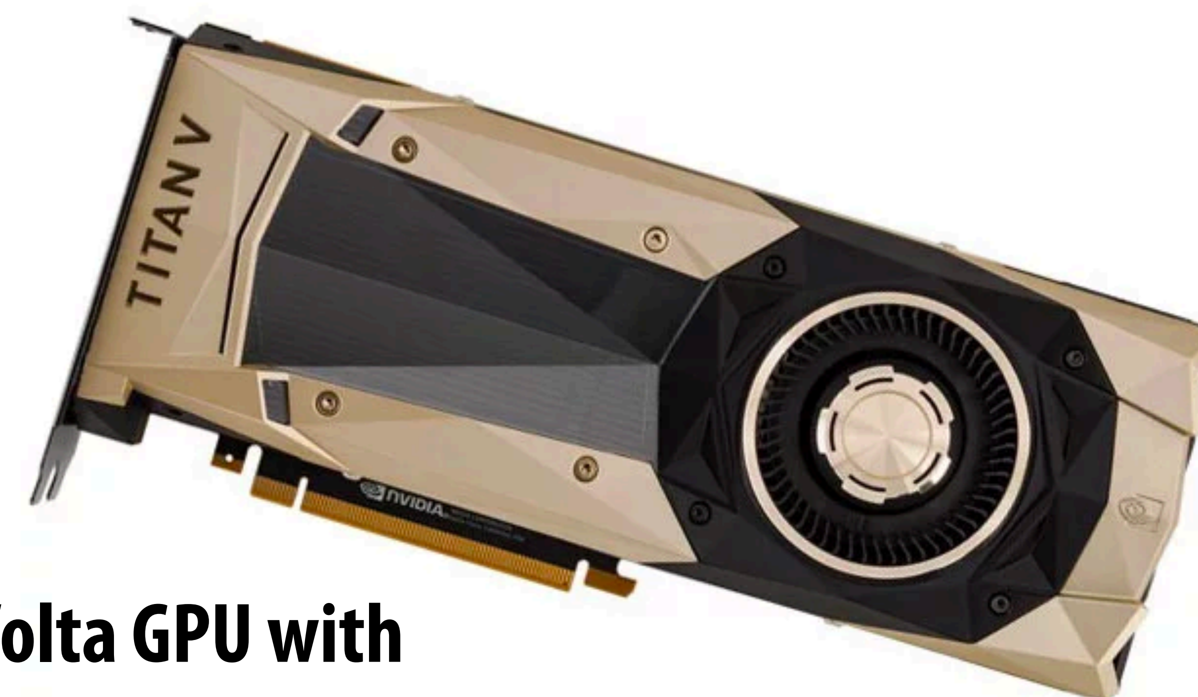
Apple Neural Engine



Intel Lake Crest
Deep Learning Accelerator



MIT Eyeriss



Volta GPU with
Tensor Cores

And many more…

IC Giants	Intel, Qualcomm, Nvidia, Samsung, AMD, Apple, Xilinx, IBM, STMicroelectronics, NXP, MediaTek, HiSilicon	12
Cloud/HPC	Google, Amazon_AWS, Microsoft, Aliyun, Tencent Cloud, Baidu, Baidu Cloud, HUAWEI Cloud, Fujitsu	9
IP Vendors	ARM, Synopsys, Imagination, CEVA, Cadence, VeriSilicon	6
Startups in China	Cambricon, Horizon Robotics, DeePhi, Bitmain, Chipintelli, Thinkforce	6
Startups Worldwide	Cerebras, Wave Computing, Graphcore, PEZY, KnuEdge, Tenstorrent, ThinCI, Koniku, Adapteva, Knowm, Mythic, Kalray, BrainChip, AImotive, DeepScale, Leepmind, Krtkl, NovuMind, REM, TERADEEP, DEEP VISION, Groq, KAIST DNPU, Kneron, Vathys, Esperanto Technologies	26

Image credit: Shan Tang [https://medium.com/@shan.tang.g/a-list-of-chip-ip-for-deep-learning-48d05f1759ae]

Modern NVIDIA GPU
(Volta)

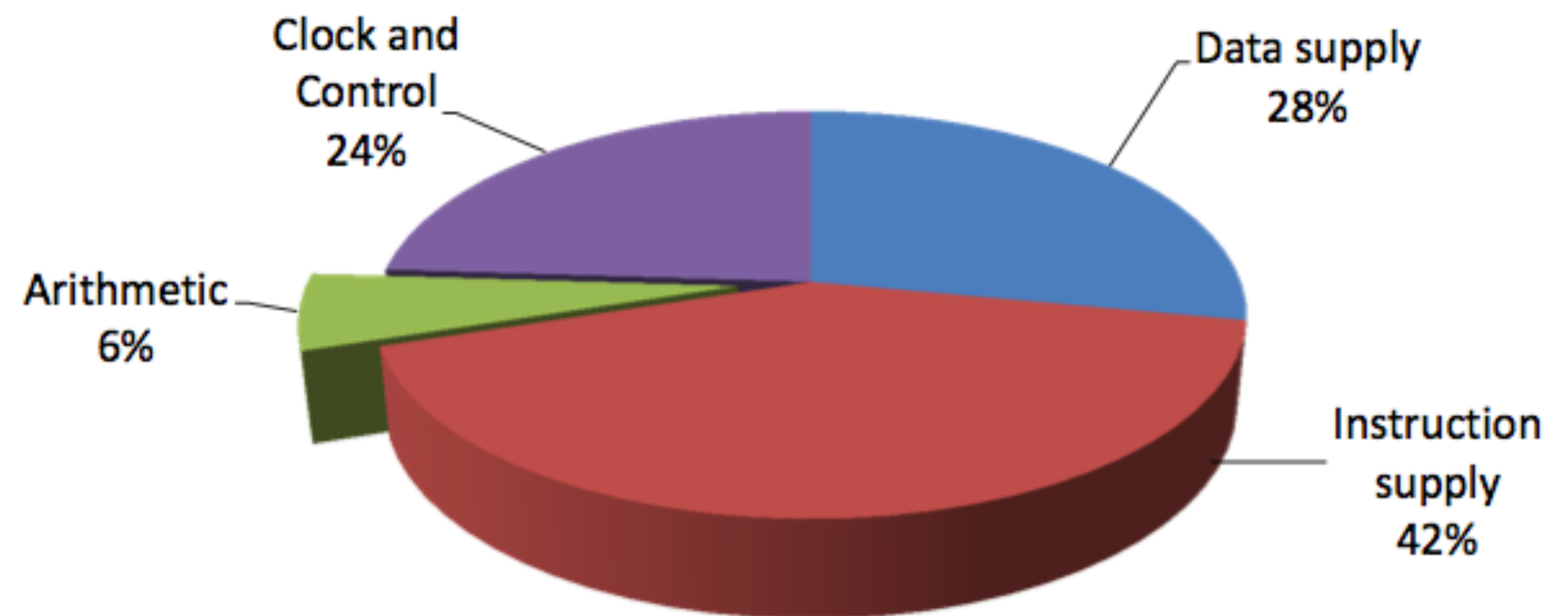
Recall: properties of GPUs

- "Compute rich": packed densely with processing elements
  - Good for compute-bound applications

- Good, because dense-matrix multiplication and DNN convolutional layers (when implemented properly) are compute bound

- But recall cost of instruction stream processing and control in a programmable processor:

Note: these figures are estimates for a CPU:



Efficient Embedded Computing [Dally et al. 08]
[Figure credit Eric Chung]

One solution: more complex instructions

- Fused multiply add (ax + b)
- 4-component dot product x = A dot B
- 4x4 matrix multiply
  - AB + C for 4x4 matrices A, B, C

- Key principle: amortize cost of instruction stream processing across many operations of a single complex instruction

Volta GPU

Single instruction to perform 2x4x4x4 + 4x4 ops

Each SM core has:

64 fp32 ALUs (mul-add)

32 fp64 ALUs

8 "tensor cores"

Execute 4x4 matrix mul-add instr

A x B + C for 4x4 matrices A,B,C

A, B stored as fp16, accumulation with fp32 C

There are 80 SM cores in the GV100 GPU:

5,120 fp32 mul-add ALUs

640 tensor cores

6 MB of L2 cache

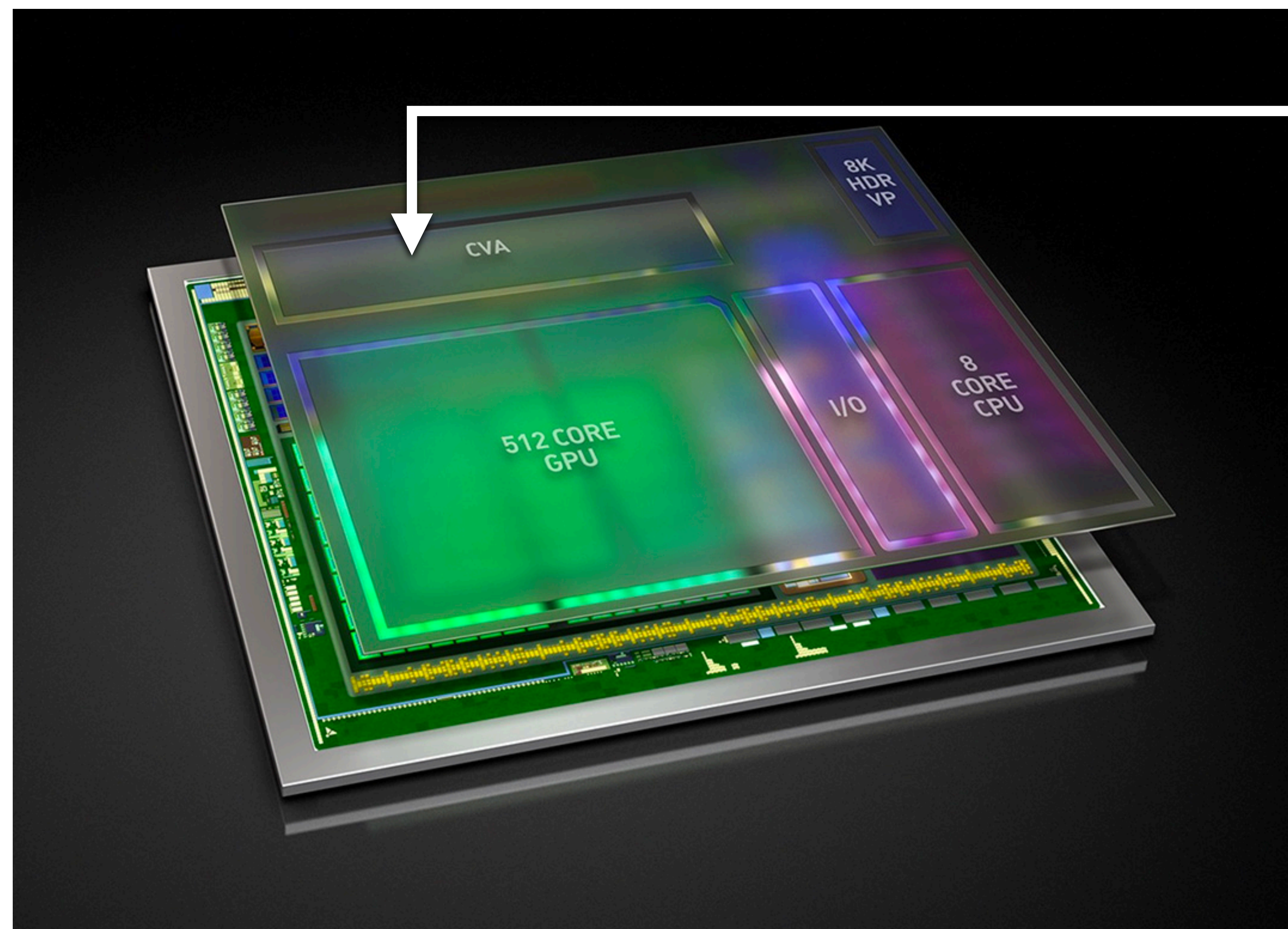
1.5 GHz max clock

= 15.7 TFLOPs fp32

= 125 TFLOPs (fp16/32 mixed) in tensor cores

Efficiency estimates *

- Estimated overhead of programmability (instruction stream, control, etc.)
  - Half-precision FMA (fused multiply-add) 2000%
  - Half-precision DP4 (vec4 dot product) 500%
  - Half-precision MMA (matrix-matrix multiply + accumulate) 27%



NVIDIA Xavier (SoC for automotive domain)

Features a Computer Vision Accelerator (CVA), a custom module for deep learning acceleration (large matrix multiply unit)

But only 2x more efficient than Volta MMA instruction despite being highly specialized component. (includes optimization of gating multipliers if either operand is zero)

* Estimates by Bill Dally using academic numbers, SysML talk, Feb 2018

Google TPU
(version 1)

Google's TPU

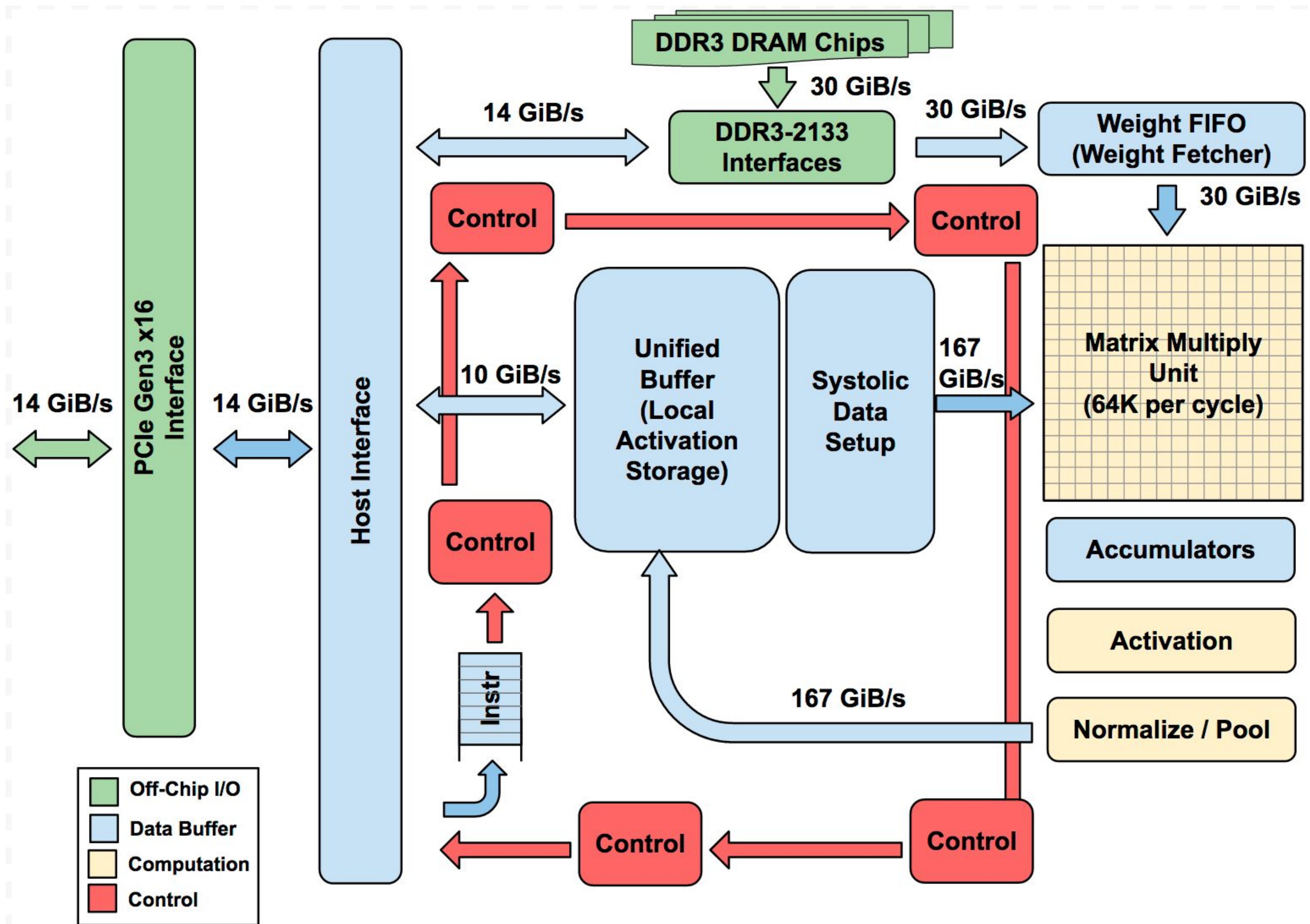
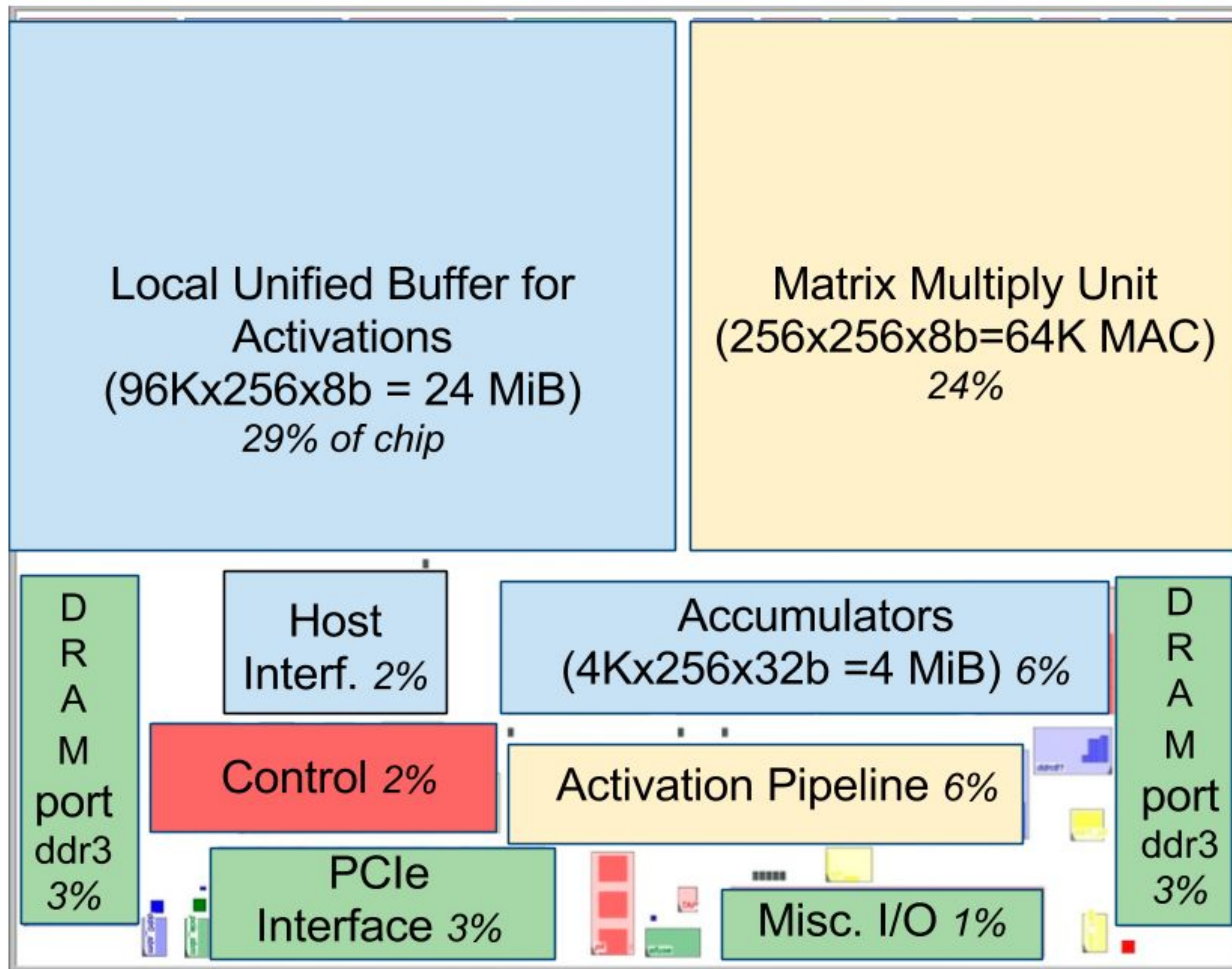
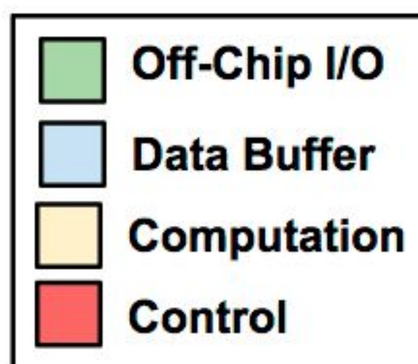


Figure credit: Jouppi et al. 2017

TPU area proportionality





Compute ~ 30% of chip

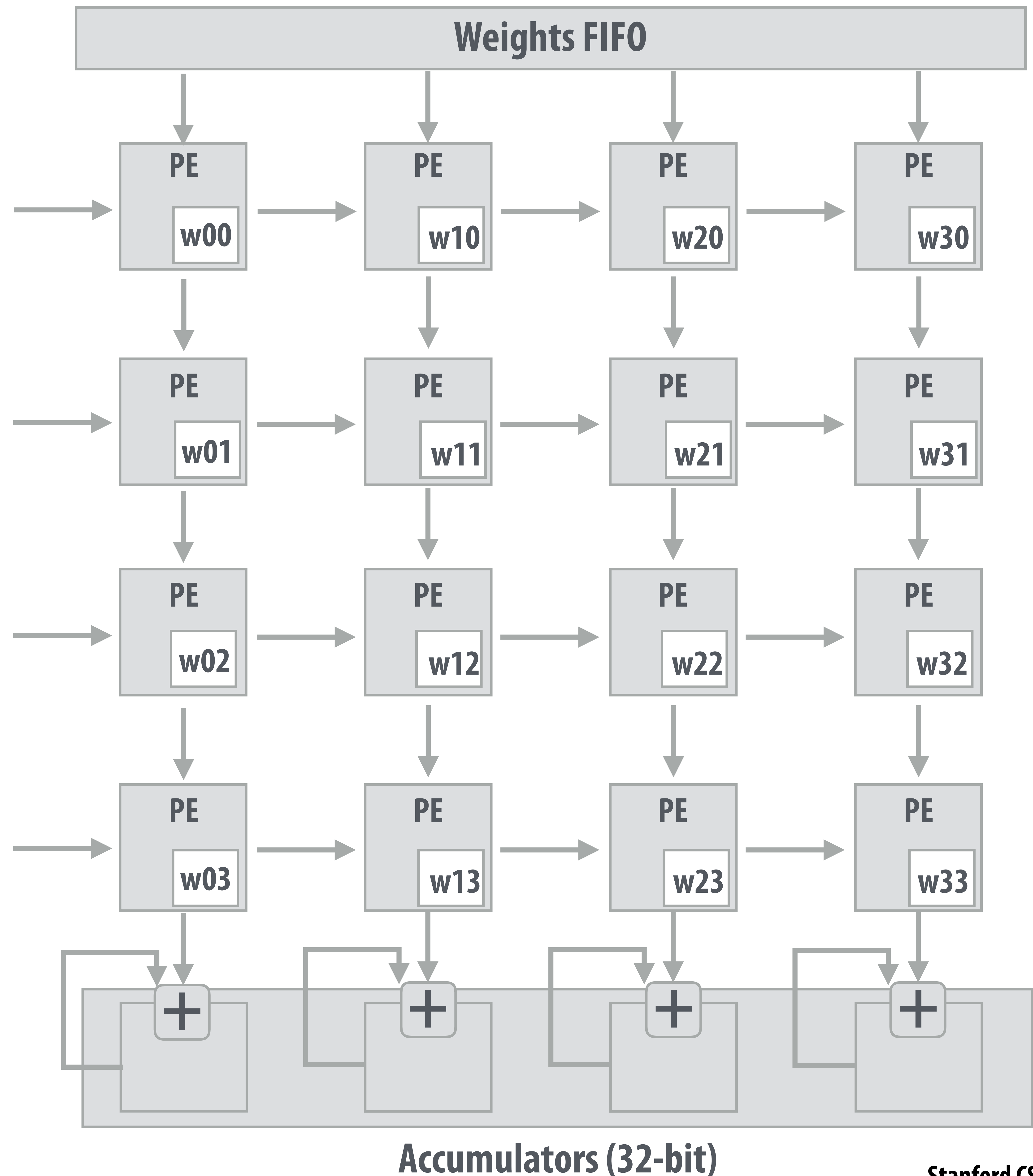
Note low area footprint of control

Key instructions:

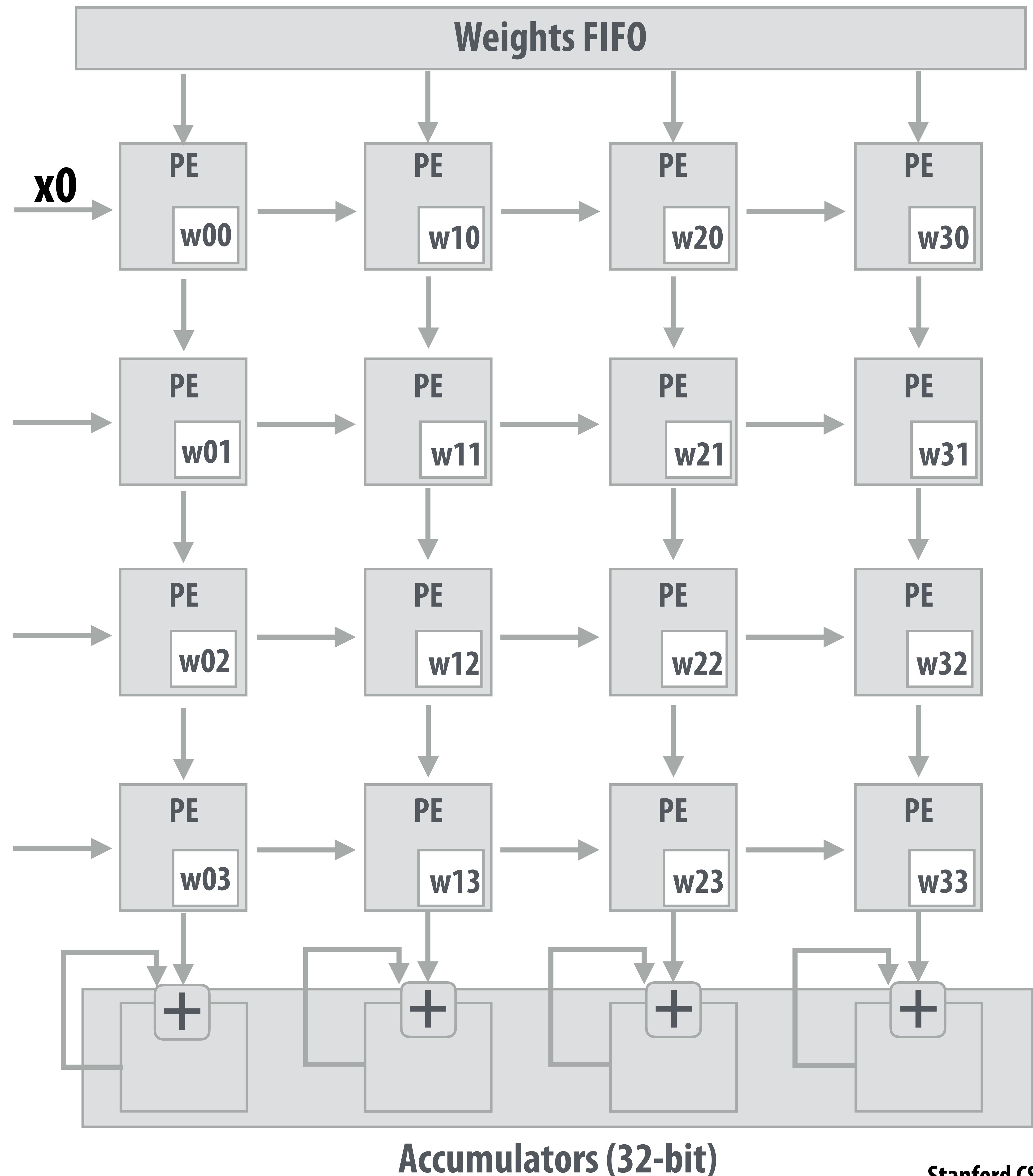
- **read host memory**
- **write host memory**
- **read weights**
- **matrix_multiply / convolve**
- **activate**

Figure credit: Jouppi et al. 2017

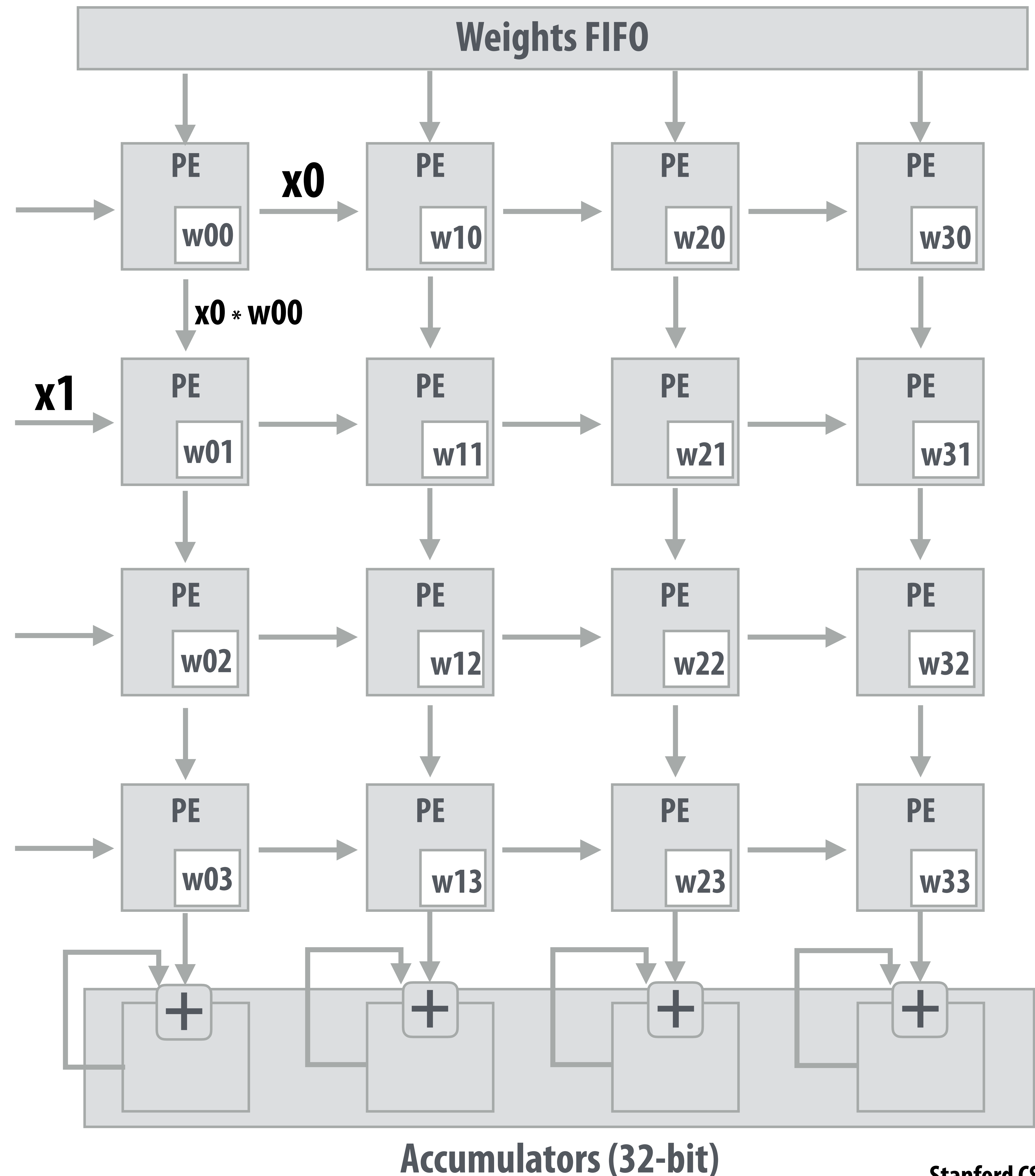
Systolic array (matrix vector multiplication example: $y=Wx$)



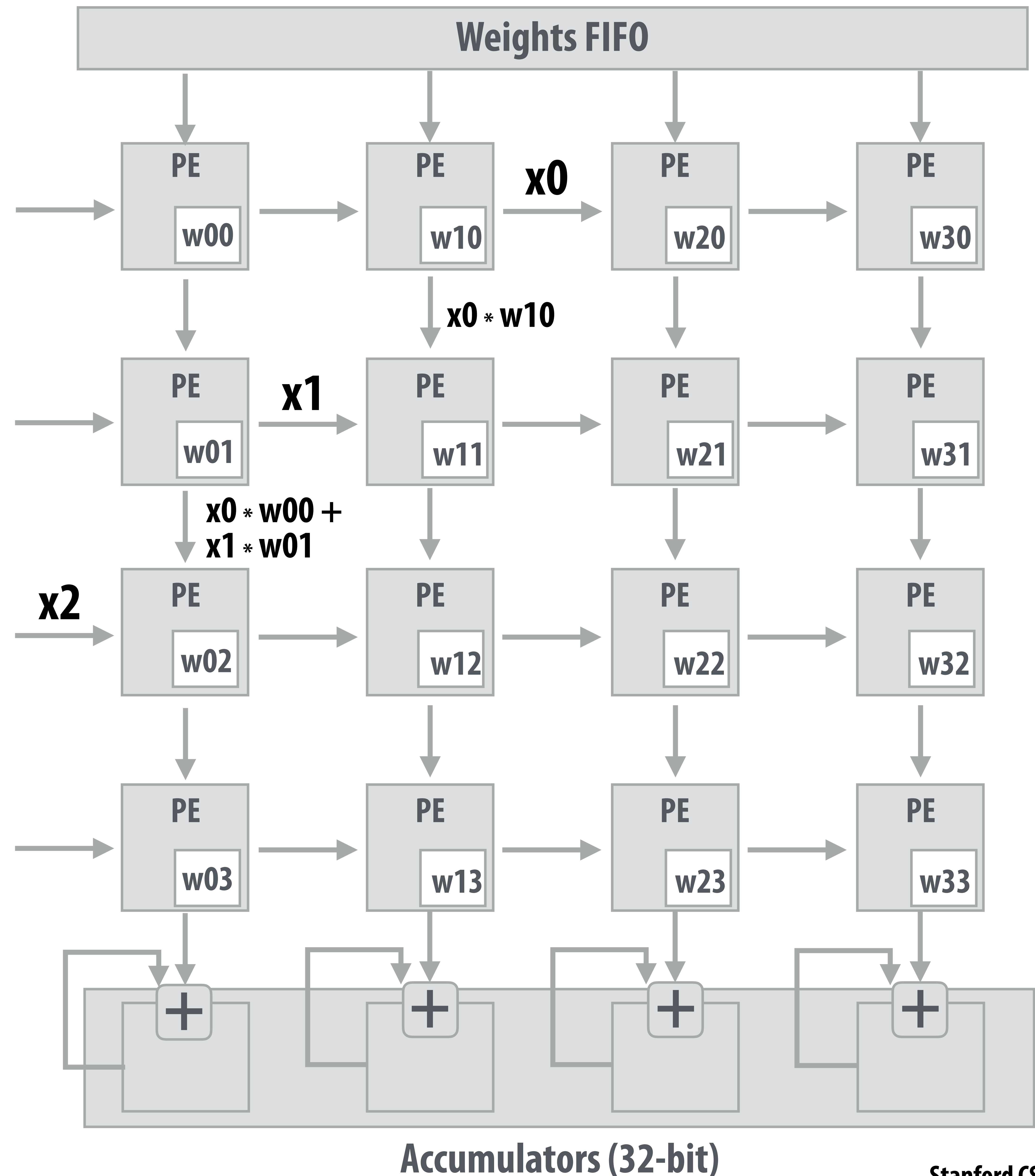
Systolic array (matrix vector multiplication example: $y=Wx$)



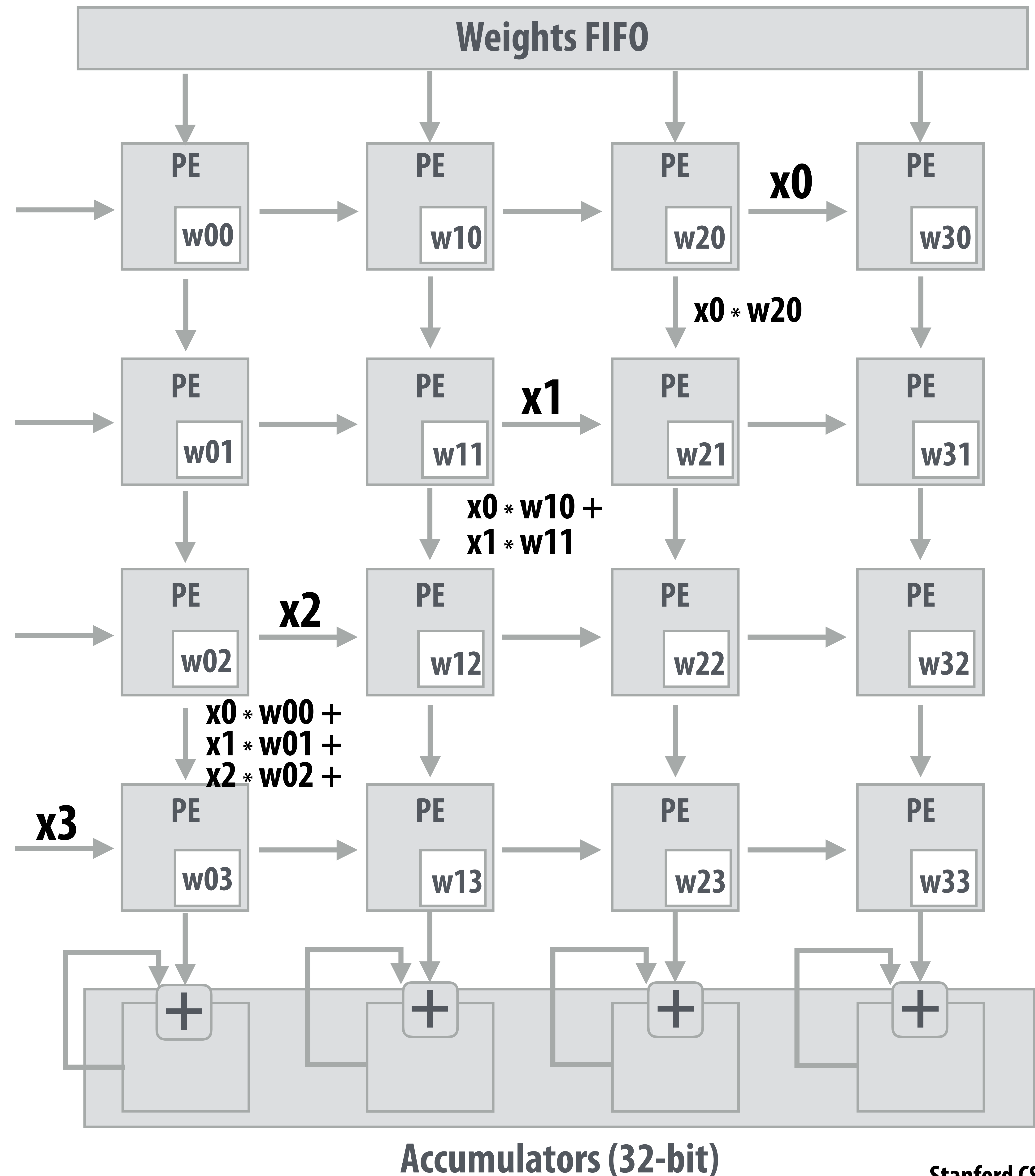
Systolic array (matrix vector multiplication example: $y=Wx$)



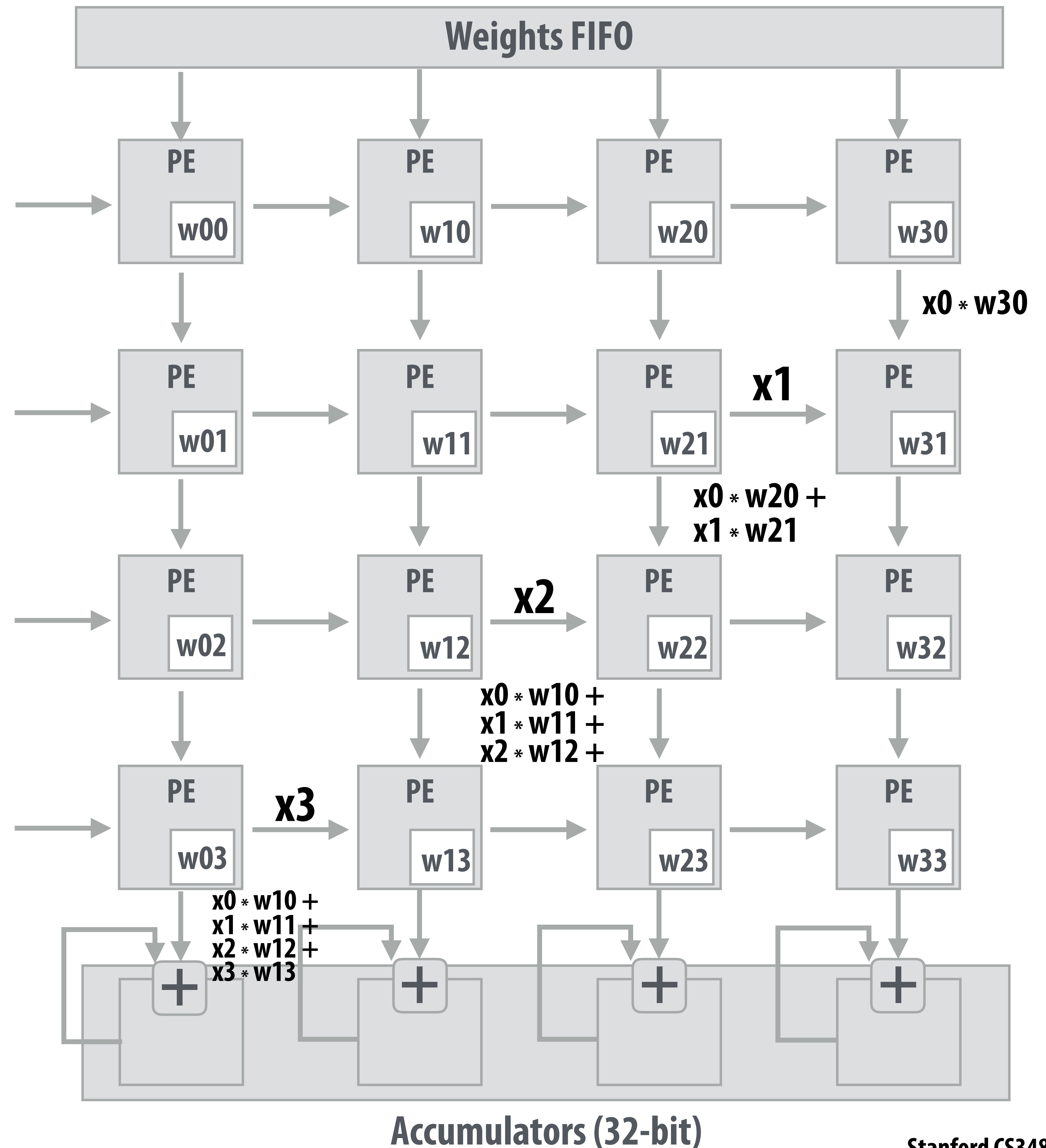
Systolic array (matrix vector multiplication example: $y=Wx$)



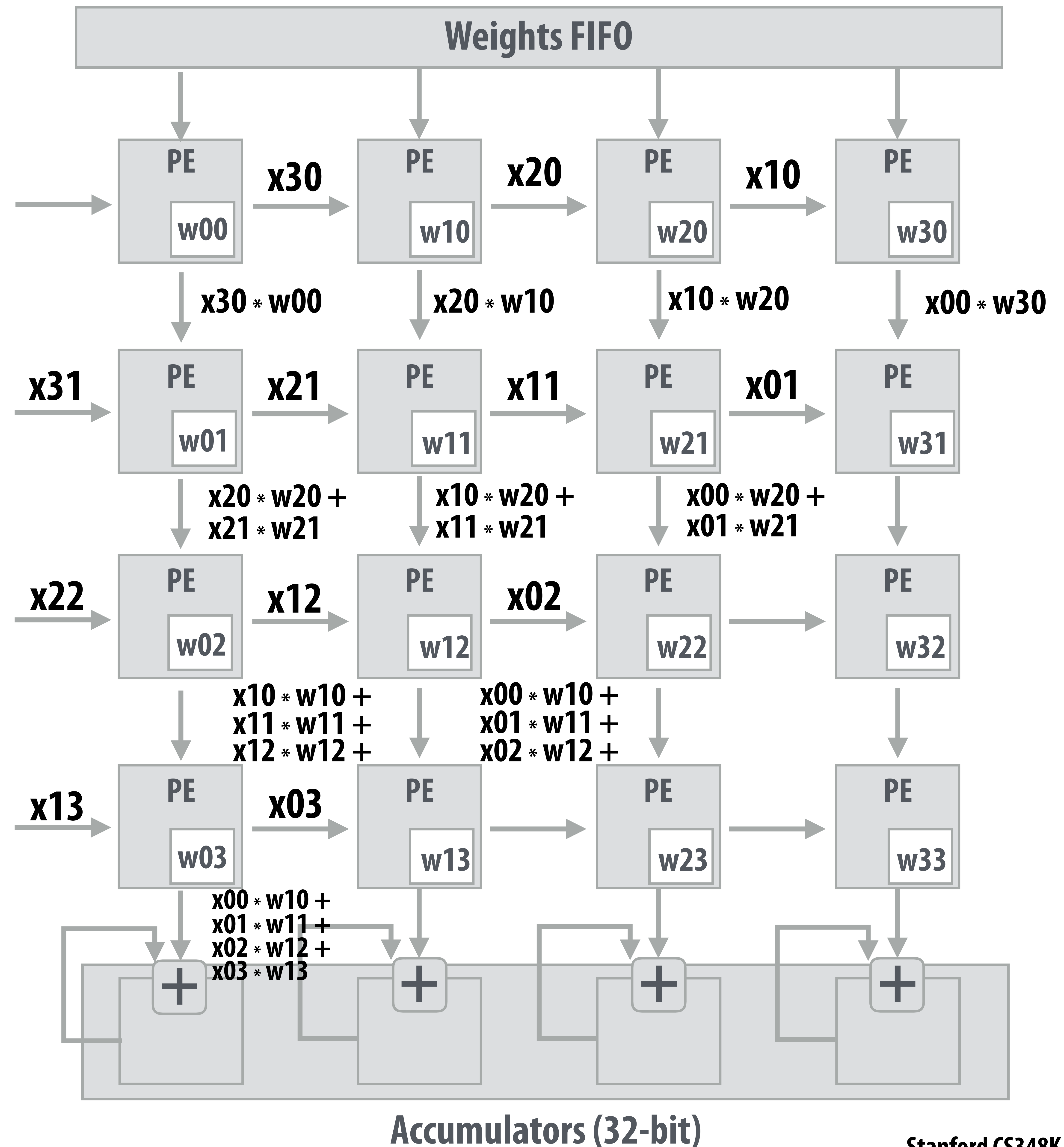
Systolic array (matrix vector multiplication example: $y = Wx$)



Systolic array (matrix vector multiplication example: $y=Wx$)



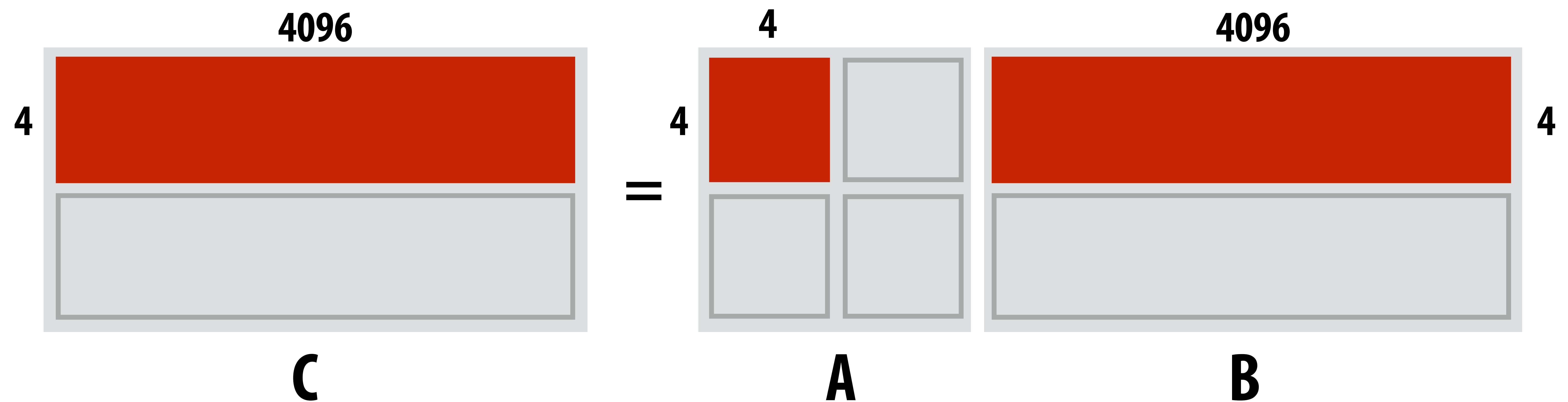
Systolic array (matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

Building larger matrix-matrix multiplies

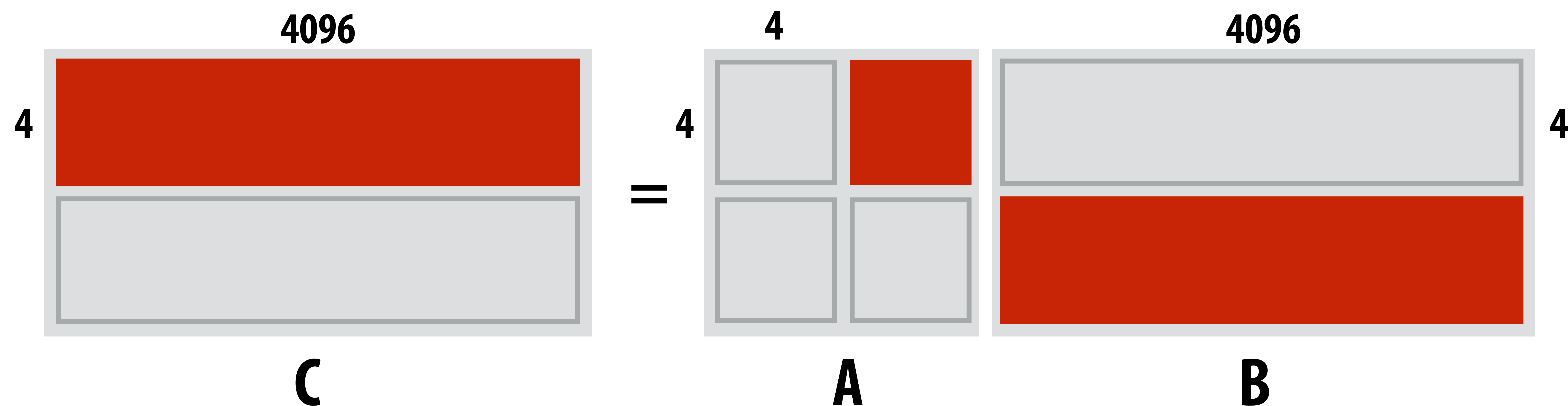
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

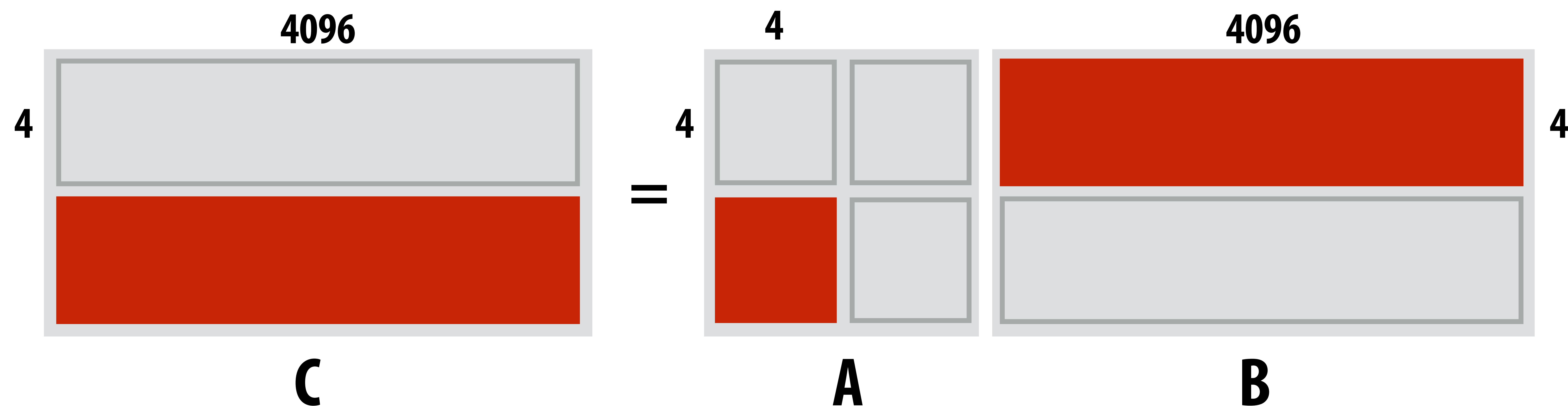
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

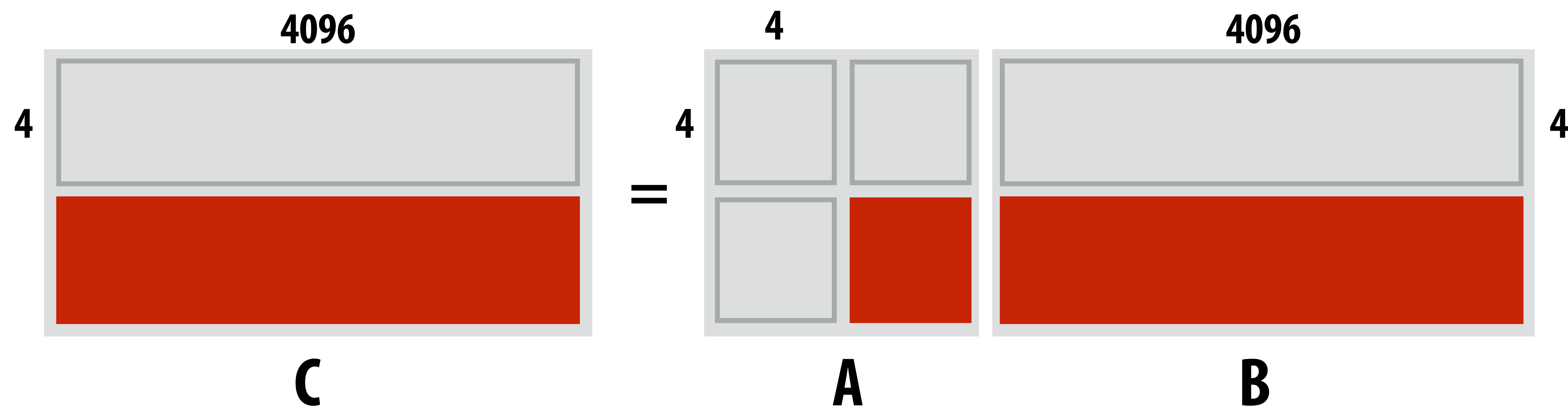
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

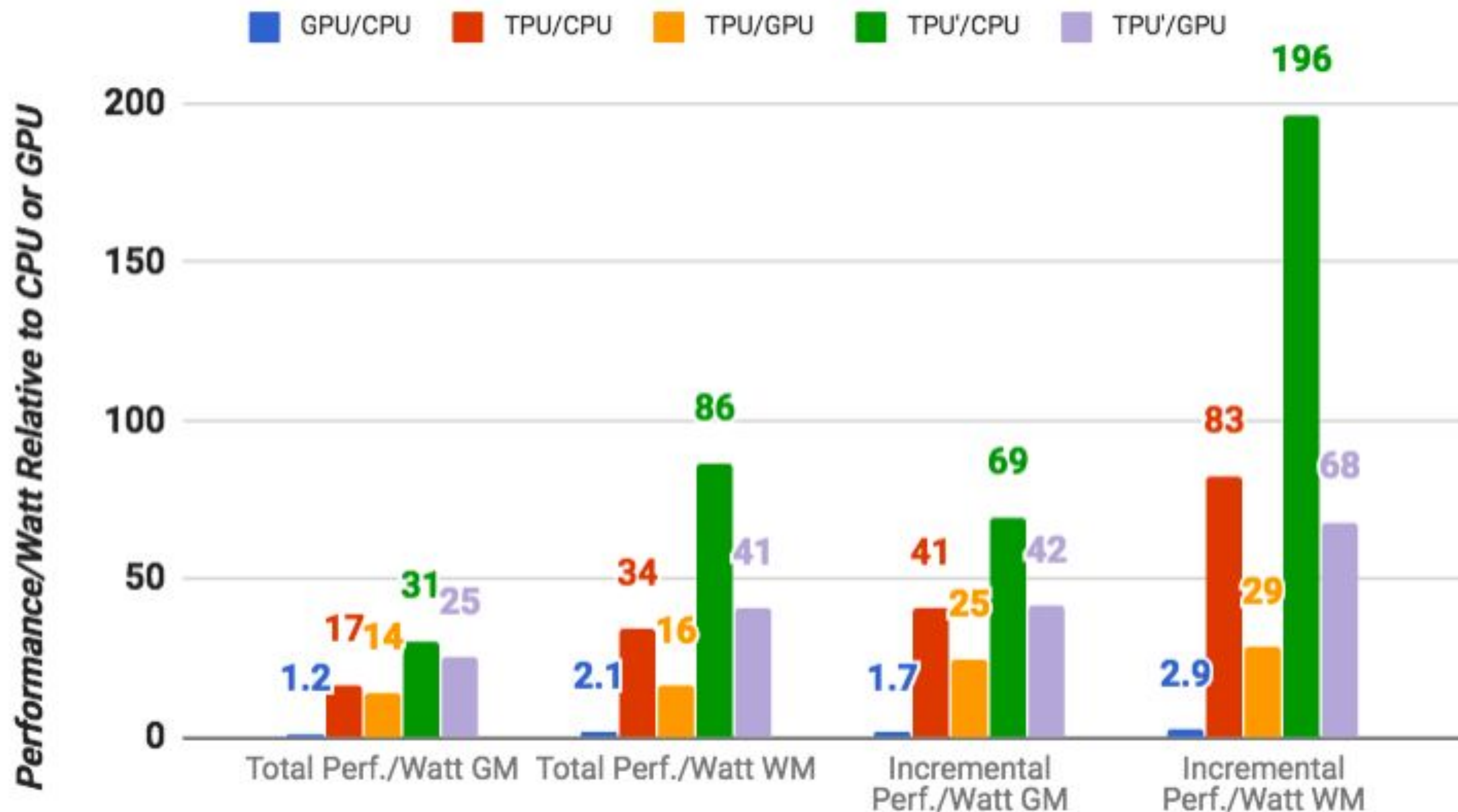
Building larger matrix-matrix multiplies

Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

TPU Performance/Watt



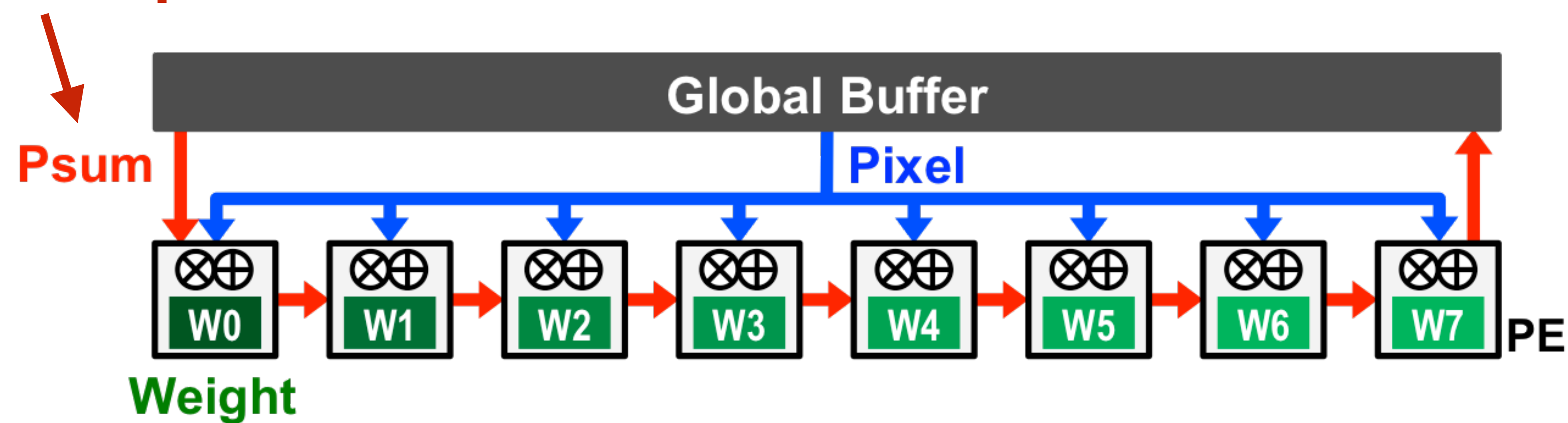
GM = geometric mean over all apps
WM = weighted mean over all apps

total = cost of host machine + CPU
incremental = only cost of TPU

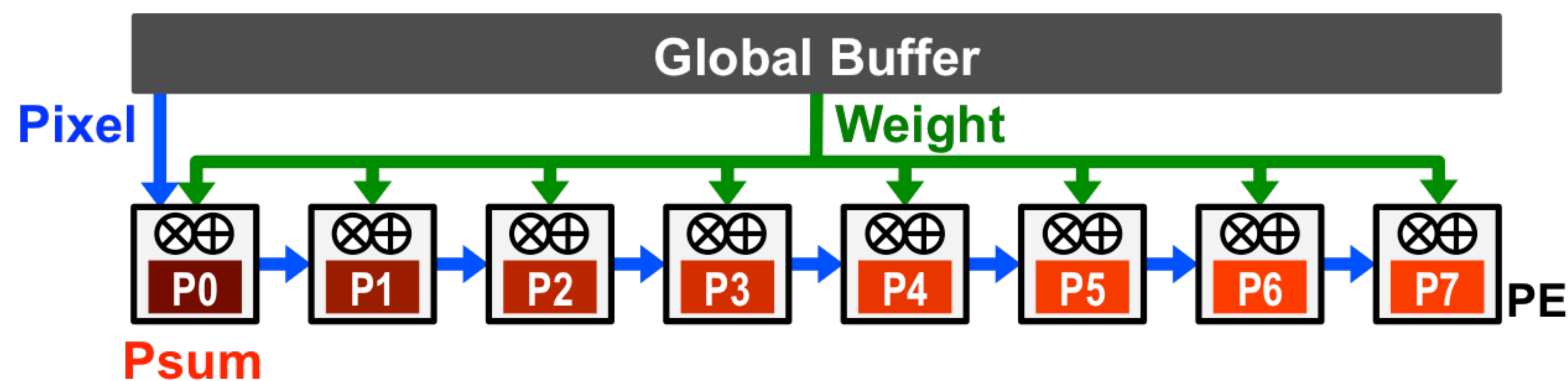
Figure credit: Jouppi et al. 2017

Alternative scheduling strategies

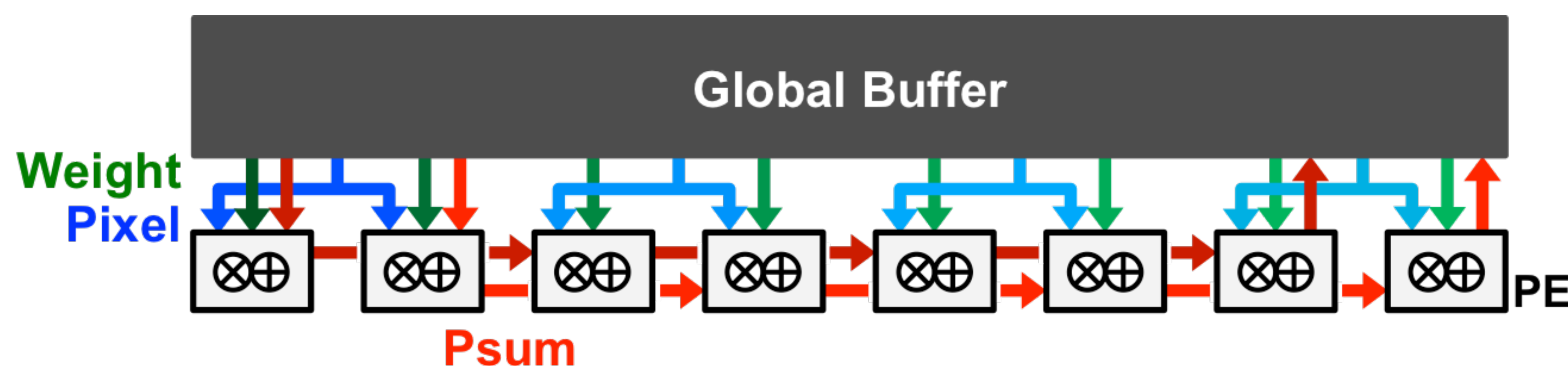
Psum = partial sum



(a) Weight Stationary



(b) Output Stationary



(c) No Local Reuse

TPU was weight stationary:
weights kept in register at PE
each PE gets different pixel
partial sum pushed through array (array has one output)

Output stationary:
each PE computes one output
push input pixel through array
each PE gets different weight
each PE accumulates locally into output

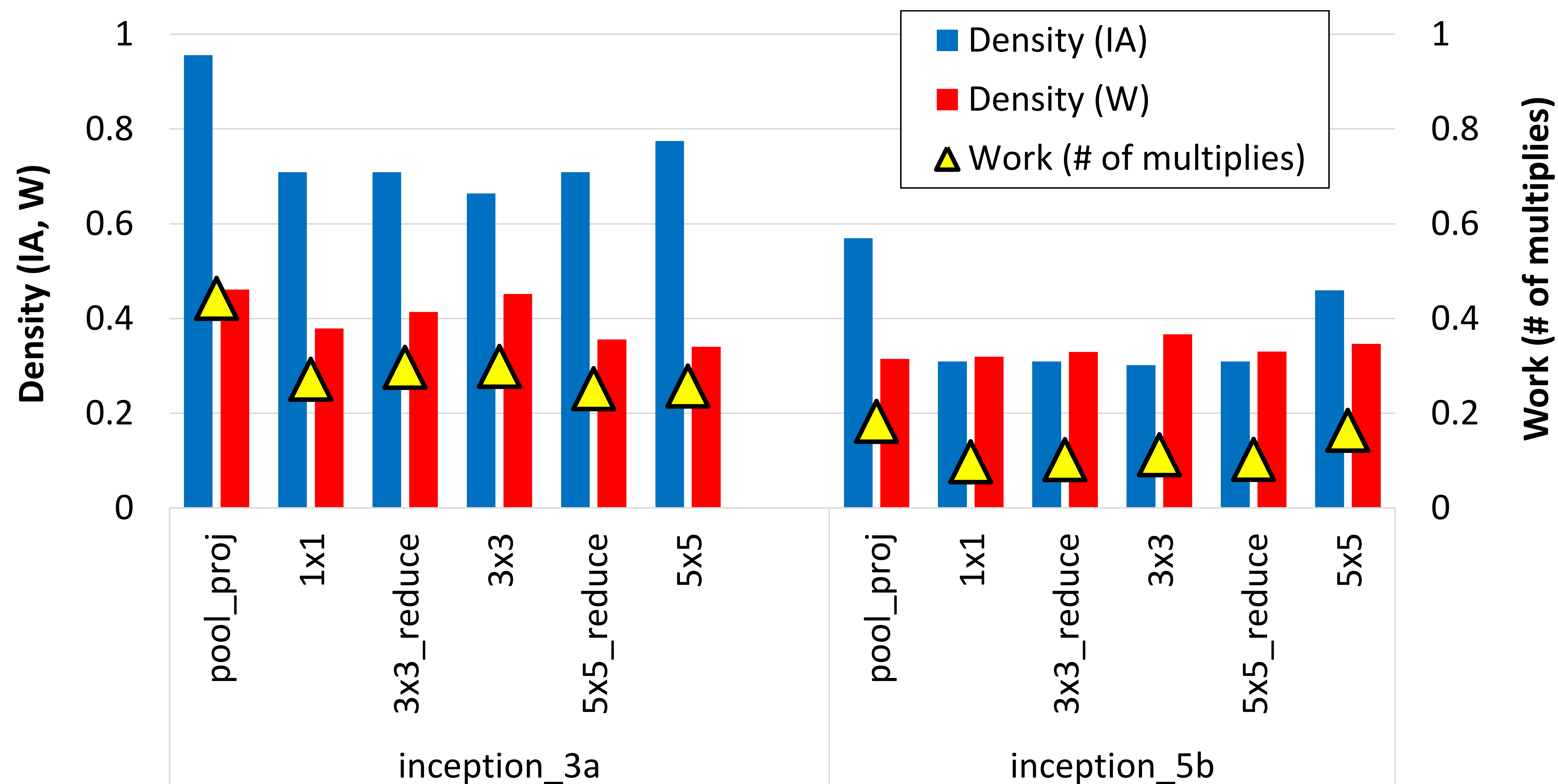
Takeaway: many DNN accelerators can be characterized by the data flow of input activations, weights, and outputs through the machine. (Just different "schedules"!)

Figure credit: Sze et al. 2017

Exploiting sparsity

Arch... sity

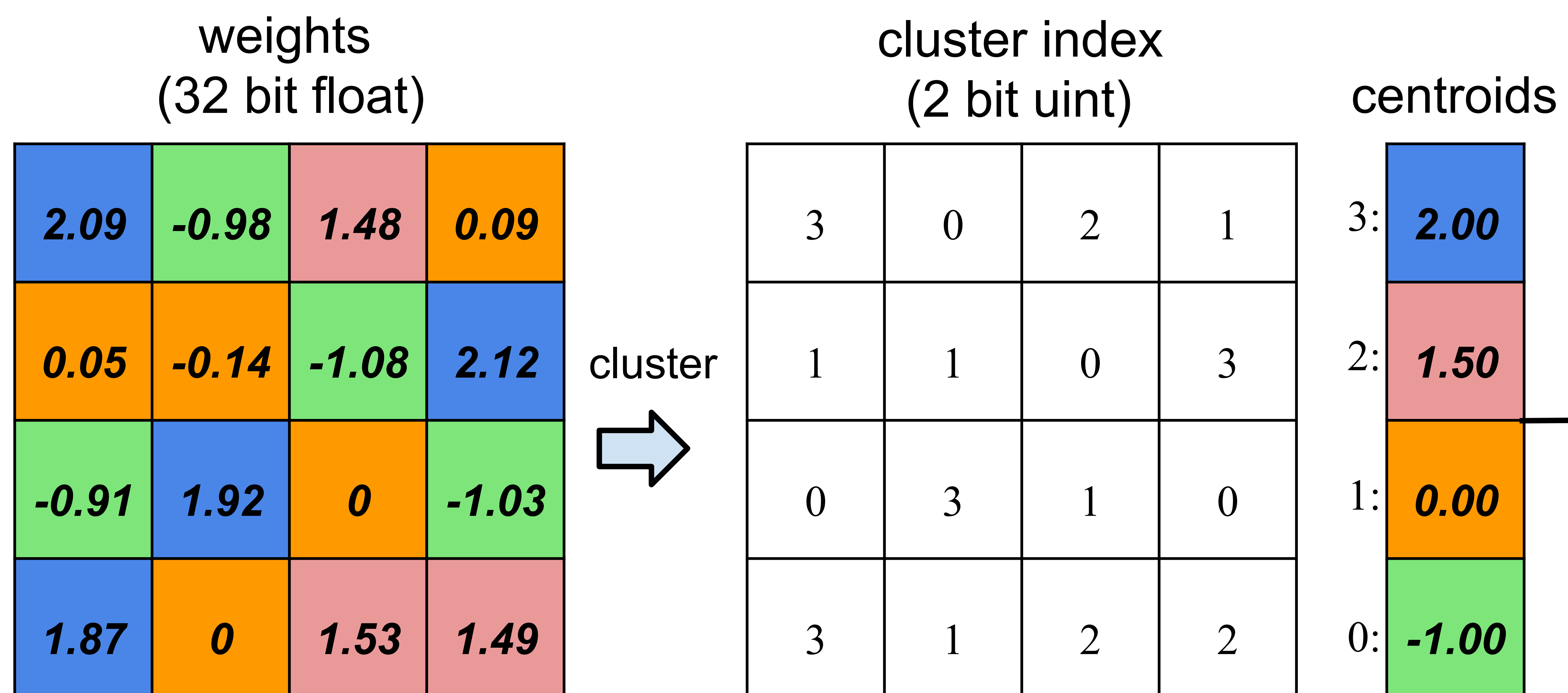
- Cor...
- If h...
  -
  - Don't move data from register file to ALU (save energy)
  - But ALU is idle (so computation doesn't run faster, just saves energy)



(b) GoogLeNet

Recall: model compression

- Step 1: sparsify weights by truncating weights with small values to zero
- Step 2: compress surviving non-zeros
  - Cluster weights via k-means clustering
  - Compress weights by only storing index of assigned cluster (lg(k) bits)



[Han et al.]

[Figure credit: Han ICLR16]

Sparse, weight-sharing ful

$$b_i = ReLU\left(\sum_{j=0}^{n-1} W_{ij}a_j\right)$$

Fully-connected layer:
Matrix-vector multiplication of activation vector a against weight matrix W

$$b_i = ReLU\left(\sum_{j \in X_i \cap Y} S[I_{ij}]a_j\right)$$

Sparse, weight-sharing representation:
I_{ij} = index for weight W_{ij}
S[] = table of shared weight values
X_i = list of non-zero indices in row i
Y = list of non-zero indices in vector a

Sparse-matrix, vector multiplication

Represent weight matrix in compressed sparse column (CSC) format to exploit sparsity in activation vector:

```
for each nonzero a_j in a:
    for each nonzero M_ij in column M_j:
      b_i += M_ij * a_j
```

More detailed version (assumes CSC matrix):

```
int16* a_values;    // dense
PTR*   M_j_start;   // column j
int4*  M_j_values;
int4*  M_j_indices;
int16* lookup; // lookup table for
               // cluster values (from
               // deep compression paper)
```

```
for j=0 to length(a):
   if (a[j] == 0) continue;   // scan to next nonzero
   col_values = M_j_values[M_j_start[j]]; // j-th col
   col_indices = M_j_indices[M_j_start[j]]; // row idx in col
   col_nonzeros = M_j_start[j+1] - M_j_start[j];
   for i=0, i_count=0 to col_nonzeros:
      i    += col_indices[i_count];
      b[i] += lookup[col_values[i_count]] * a_values[j];
```

* Recall from deep compression paper: there is a unique lookup table for each chunk of matrix values

Parallelization of sparse-matrix-vector product

Stride rows of matrix across processing elements

Output activations strided across processing elements

$\vec{a}$ (0 0 $\mathbf{a_2}$ 0 a_4 a_5 0 a_7)

$\times$

									$\vec{b}$	ReLU
$PE0$	$w_{0,0}$	0	$\boldsymbol{w_{0,2}}$	0	$w_{0,4}$	$w_{0,5}$	$w_{0,6}$	0	b_0	b_0
$PE1$	0	$w_{1,1}$	$\mathbf{0}$	$w_{1,3}$	0	0	$w_{1,6}$	0	b_1	b_1
$PE2$	0	0	$\boldsymbol{w_{2,2}}$	0	$w_{2,4}$	0	0	$w_{2,7}$	$-b_2$	0
$PE3$	0	$w_{3,1}$	$\mathbf{0}$	0	0	$w_{0,5}$	0	0	b_3	b_3
	0	$w_{4,1}$	$\mathbf{0}$	0	$w_{4,4}$	0	0	0	$-b_4$	0
	0	0	$\mathbf{0}$	$w_{5,4}$	0	0	0	$w_{5,7}$	b_5	b_5
	0	0	$\mathbf{0}$	0	$w_{6,4}$	0	$w_{6,6}$	0	b_6	b_6
	$w_{7,0}$	0	$\mathbf{0}$	$w_{7,4}$	0	0	$w_{7,7}$	0	$-b_7$	0
	$w_{8,0}$	0	$\mathbf{0}$	0	0	0	0	$w_{8,7}$	$-b_8$	0
	$w_{9,0}$	0	$\mathbf{0}$	0	0	0	$w_{9,6}$	$w_{9,7}$	$-b_9$	0
	0	0	$\mathbf{0}$	0	$w_{10,4}$	0	0	0	b_{10}	b_{10}
	0	0	$\boldsymbol{w_{11,2}}$	0	0	0	0	$w_{11,7}$	$-b_{11}$	0
	$w_{12,0}$	0	$\boldsymbol{w_{12,2}}$	0	0	$w_{12,5}$	0	$w_{12,7}$	$-b_{12}$	0
	$w_{13,0}$	$w_{13,2}$	$\mathbf{0}$	0	0	0	$w_{13,6}$	0	b_{13}	b_{13}
	0	0	$\boldsymbol{w_{14,2}}$	$w_{14,3}$	$w_{14,4}$	$w_{14,5}$	0	0	b_{14}	b_{14}
	0	0	$\boldsymbol{w_{15,2}}$	$w_{15,3}$	0	$w_{15,5}$	0	0	$-b_{15}$	0

$= \quad \overset{ReLU}{\Rightarrow}$

Weights stored local to PEs. Must broadcast non-zero a_j's to all PEs

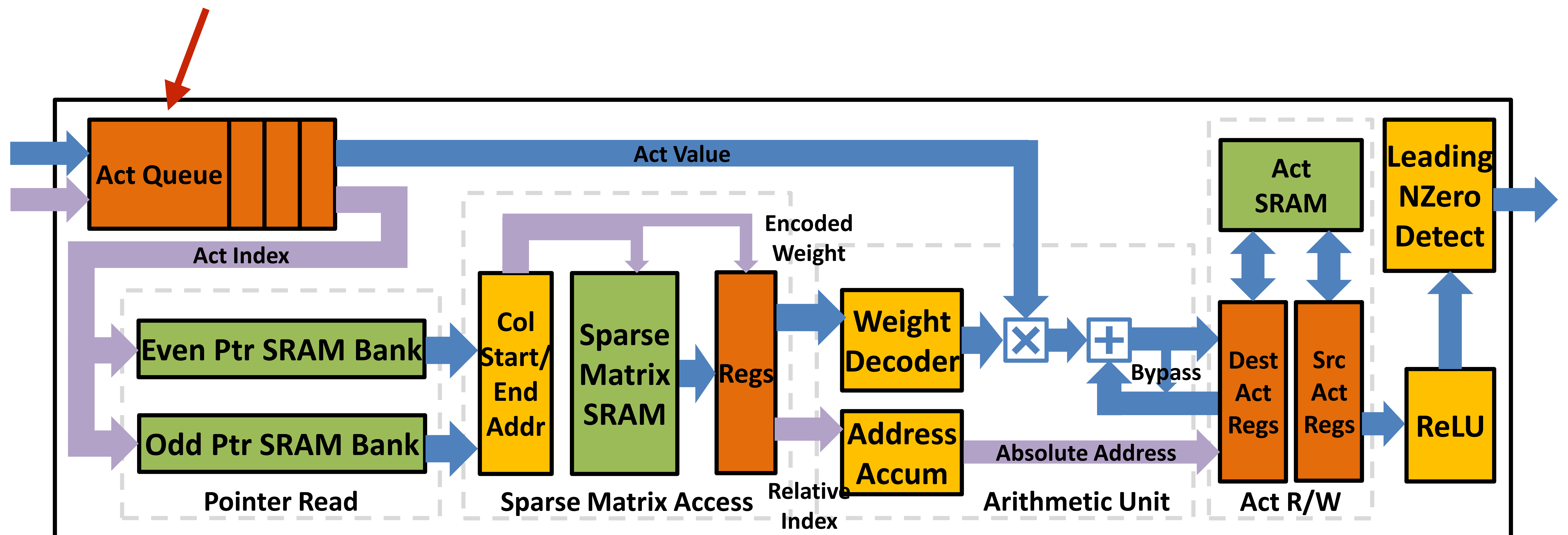
Accumulation of each output b_i is local to PE

Virtual	w	w	w	w	w	w	w	w	w	w	w	w	w

Efficient Inference Engine (EIE) for quantized sparse/matrix vector product

Custom hardware for decoding compressed-sparse representation

Tuple representing non-zero activation (a_j, j) arrives and is enqueued



EIE efficiency

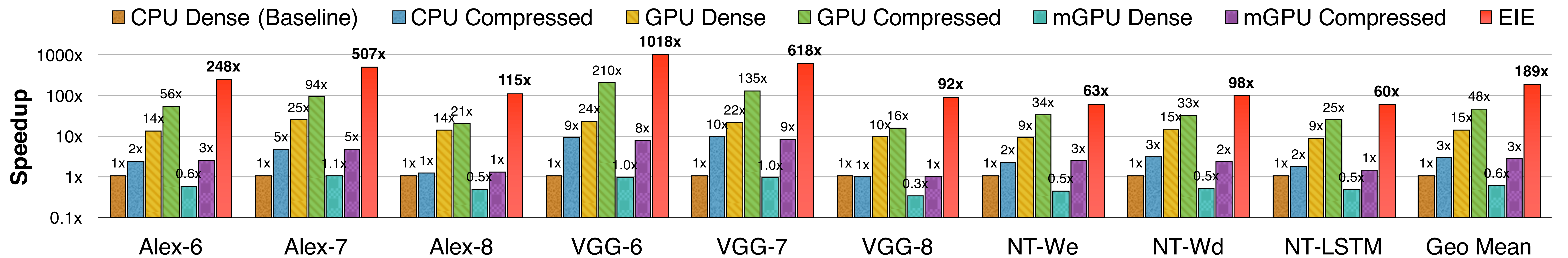
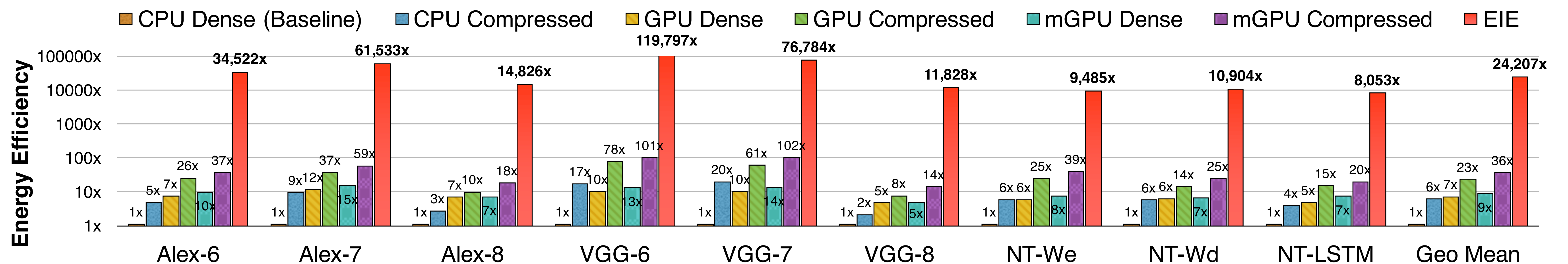


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.



CPU: Core i7 5930k (6 cores)
GPU: GTX Titan X
mGPU: Tegra K1

Warning: these are not end-to-end numbers: just fully connected layers!

Sources of energy savings:

- **Compression allows all weights to be stored in SRAM (few DRAM loads)**
- **Low-precision 16-bit fixed-point math (5x more efficient than 32-bit fixed math)**
- **Skip math on input activations that are zero (65% less math)**

A critical eye…

- EIE paper highlights performance on fully connected layers (see graph above)
  - Final layers of networks like AlexNet, VGG…
  - Common in recurrent network topologies like LSTMs

- But many state-of-the-art image processing networks have moved to fully convolutional designs (or fully connected layers are a small part of computational cost)
  - Recall Inception, MobileNet, etc..

Input stationary design (dense 1D conv example)

Assume:

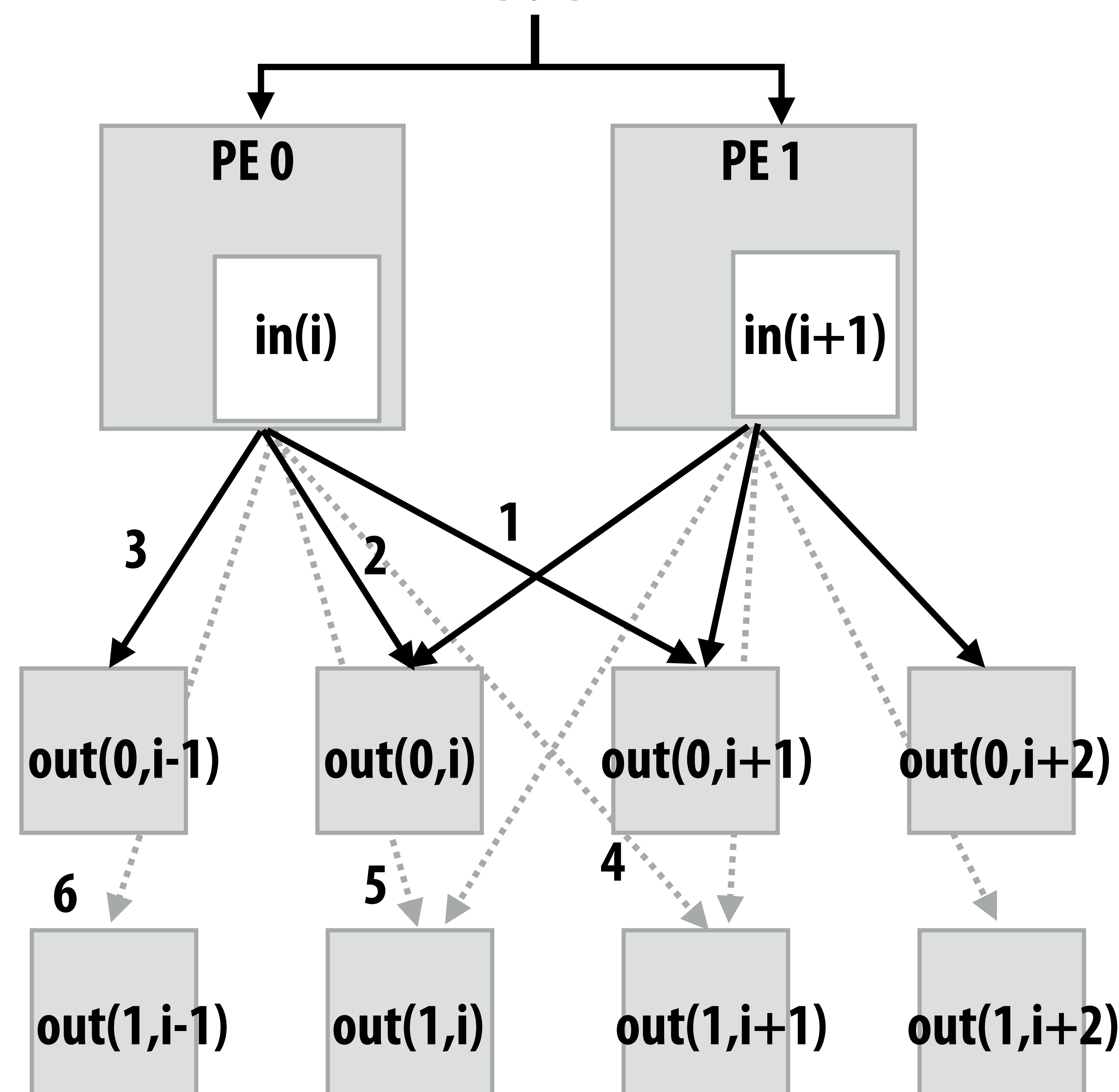
1D input/output

3-wide filters

2 output channels (K=2)

Stream Order	Weight
6	w(1,2)
5	w(1,1)
4	w(1,0)
3	w(0,2)
2	w(0,1)
1	w(0,0)

Stream of weights (2 filters of size 3)



Processing elements

Accumulators (implement +=)

Input stationary design (sparse example)

Assume:

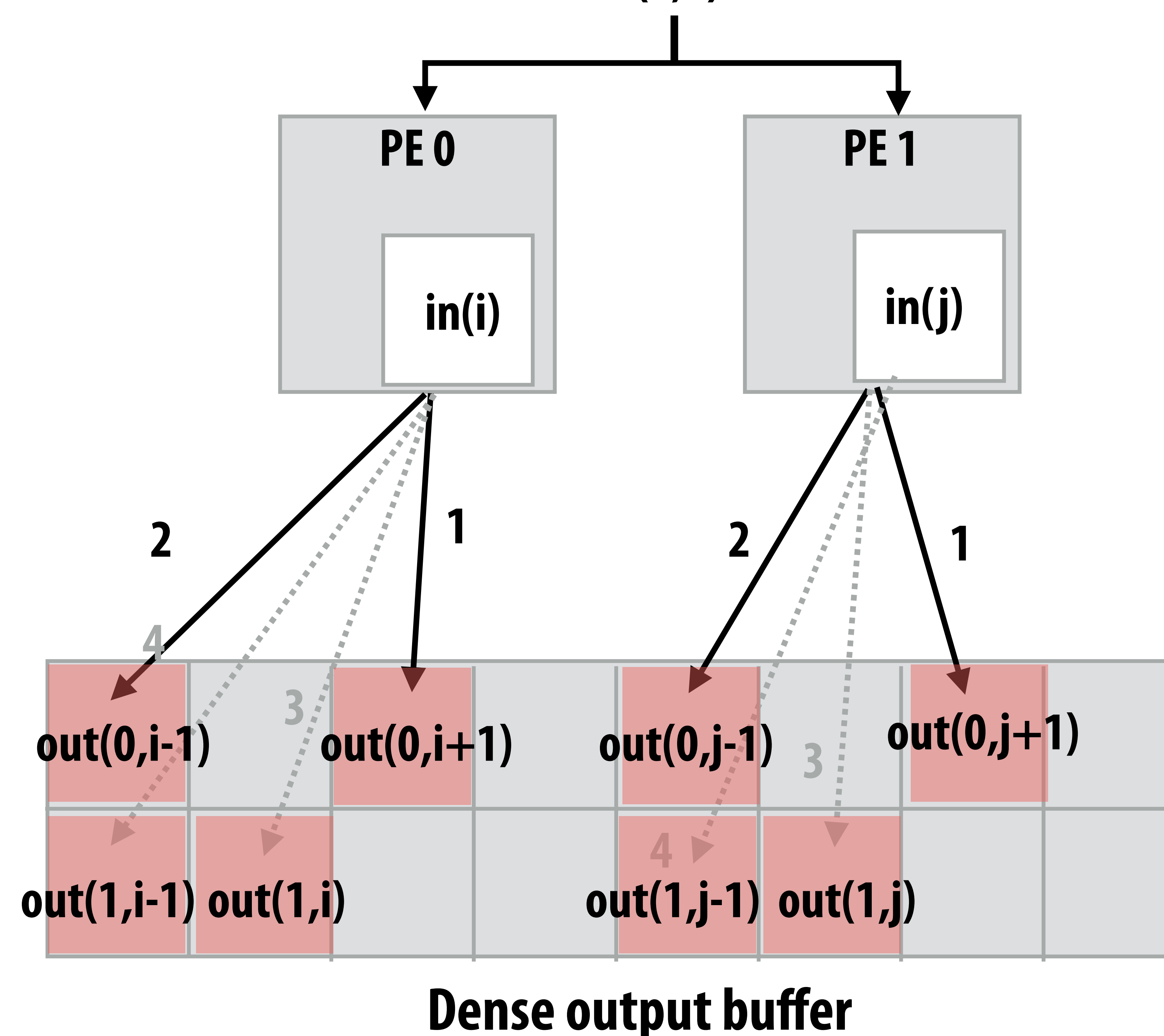
1D input/output

3-wide SPARSE filters

2 output channels (K=2)

Stream Order	Weight
4	w(1,2)
3	w(1,1)
2	w(0,2)
1	w(0,0)

Stream of sparse weights (2 filters, each with 2 non-zeros)



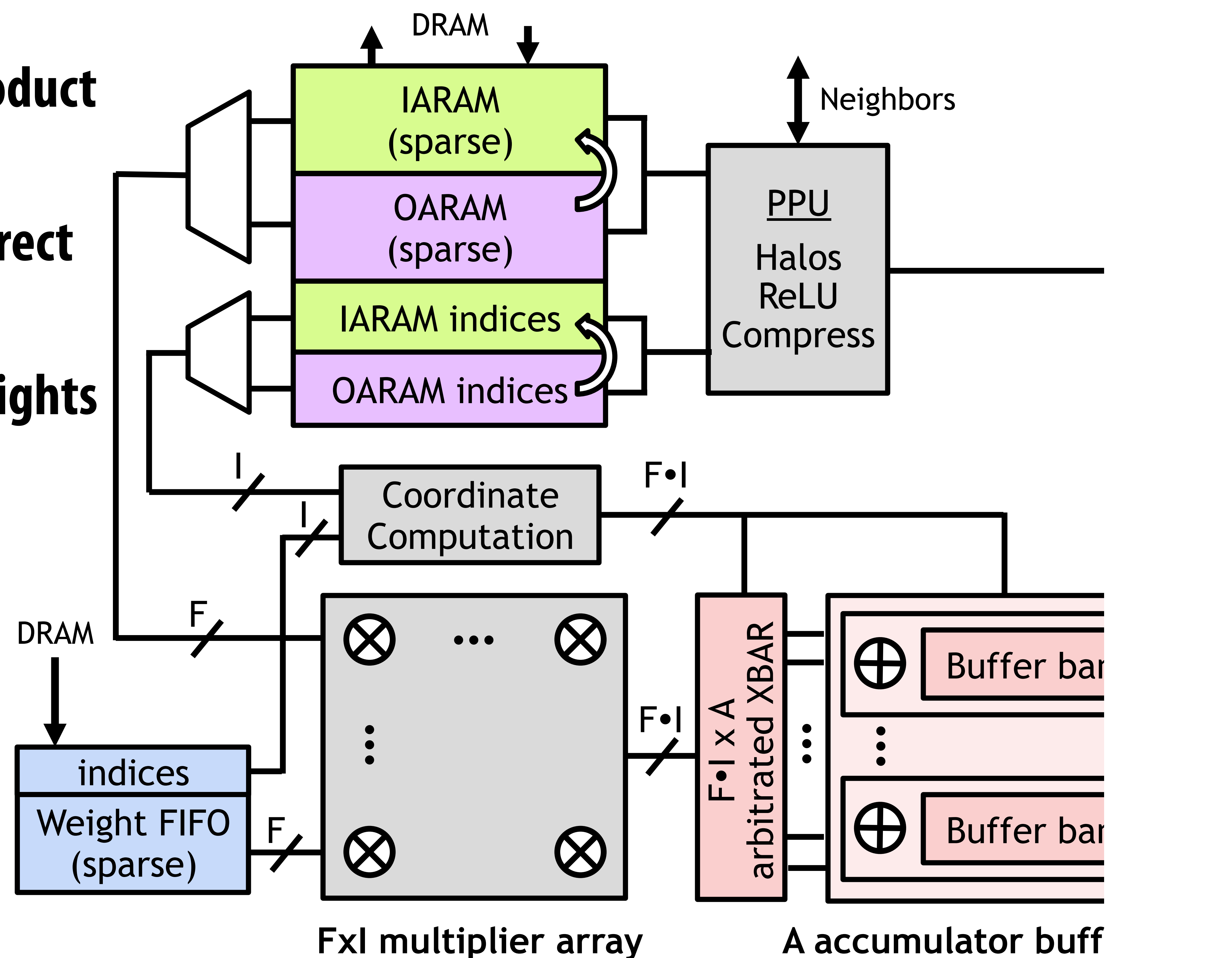
Processing elements

Accumulators (implement +=)

Note: accumulate is now a scatter

SCNN: accelerating sparse conv layers

- Like EIE: assume both activations and conv weights are sparse
- Weight stationary design:
  - Each PE receives:
    - A set of I input activations from an input channel: a list of I (value, (x,y)) pairs
    - A list of F non-zero weights
  - Each PE computes: the cross-product of these values: P x I values
  - Then scatters P x I results to correct accumulator buffer cell
  - Then repeat for new set of F weights (reuse I inputs)

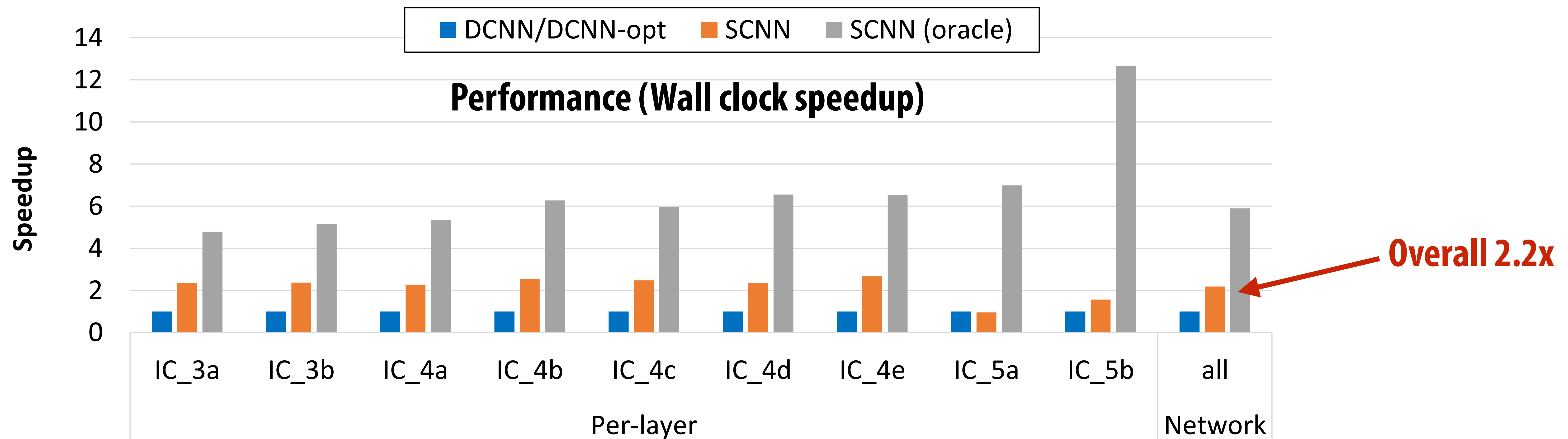


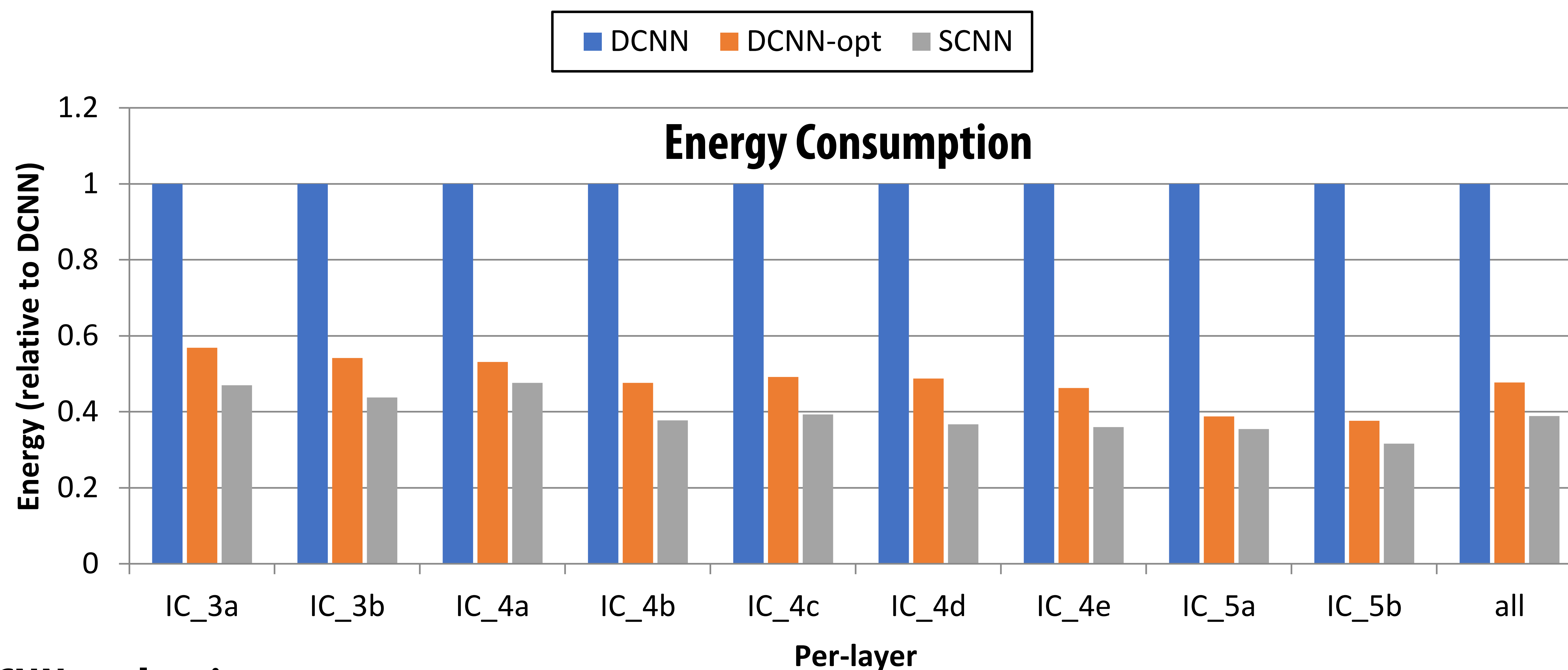
- Then, after convolution:
  - ReLU sparsifies output
  - Compress outputs into sparse representation for use as input to next layer

[Parashar et al. ISCA17]

[Parashar et al. ISCA17]

SCNN results (on GoogLeNet)





DCNN = dense CNN evaluation

DCNN-opt = includes ALU gating, and compression/decompression of activations

Summary of hardware accelerators for efficient inference

- Specialized instructions for dense linear algebra computations
  - Reduce overhead of control (compared to CPUs/GPUs)
- Reduced precision operations (cheaper computation + reduce bandwidth requirements)
- Systolic architectures for efficient communication
  - Different scheduling strategies: weight-stationary, input/output stationary, etc.
- Exploit sparsity in activations and weights
  - Skip computation involving zeros
  - Hardware to accelerates decompression of sparse representations like compressed sparse row/column