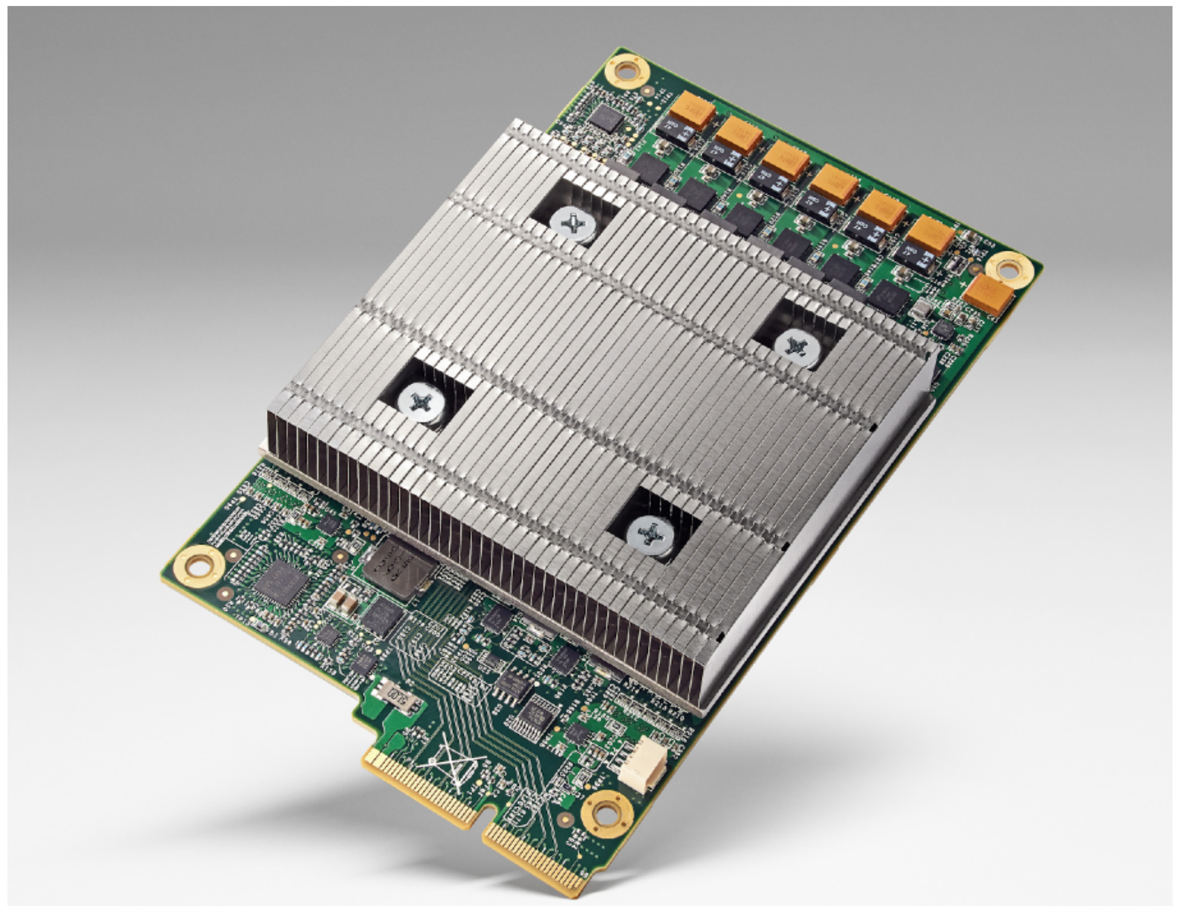
Lecture 12:

Hardware Acceleration of DNNs

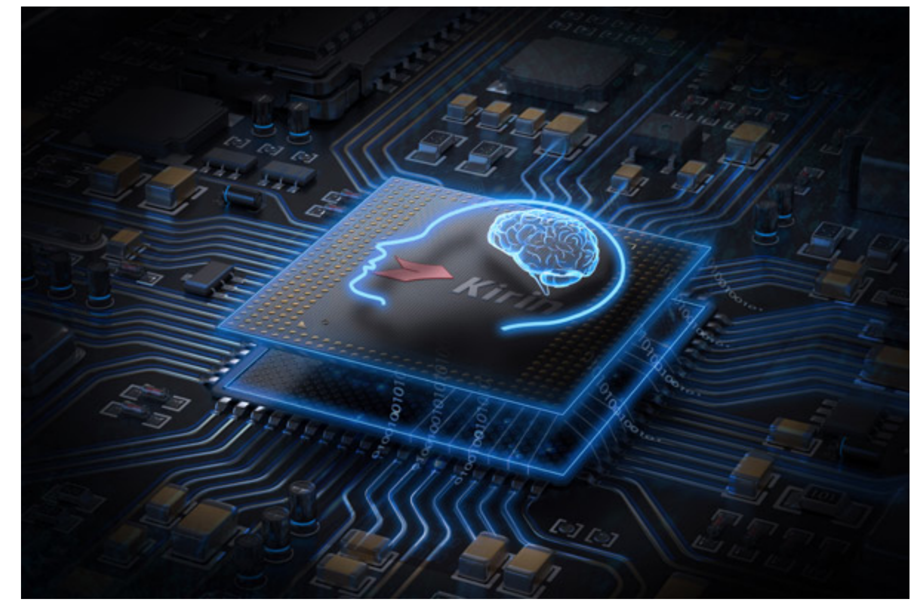
Visual Computing Systems

Stanford CS348V, Winter 2018

Hardware acceleration for DNNs

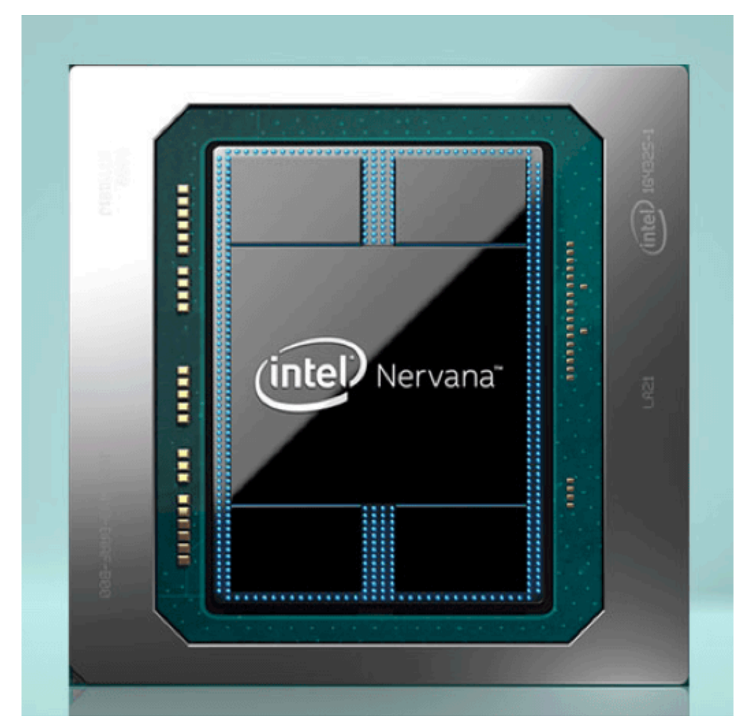


Google TPU:

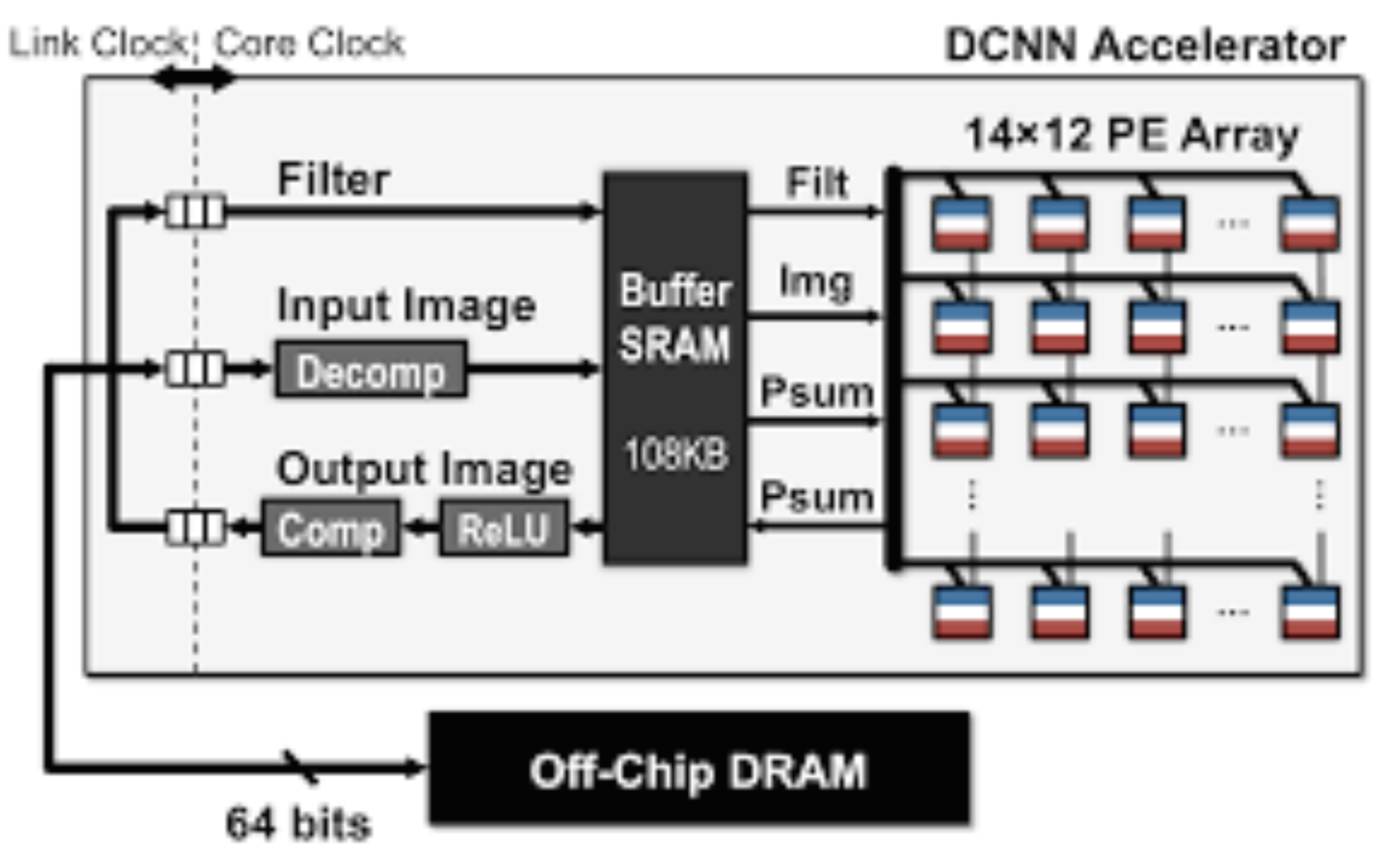


Huawei Kirin NPU

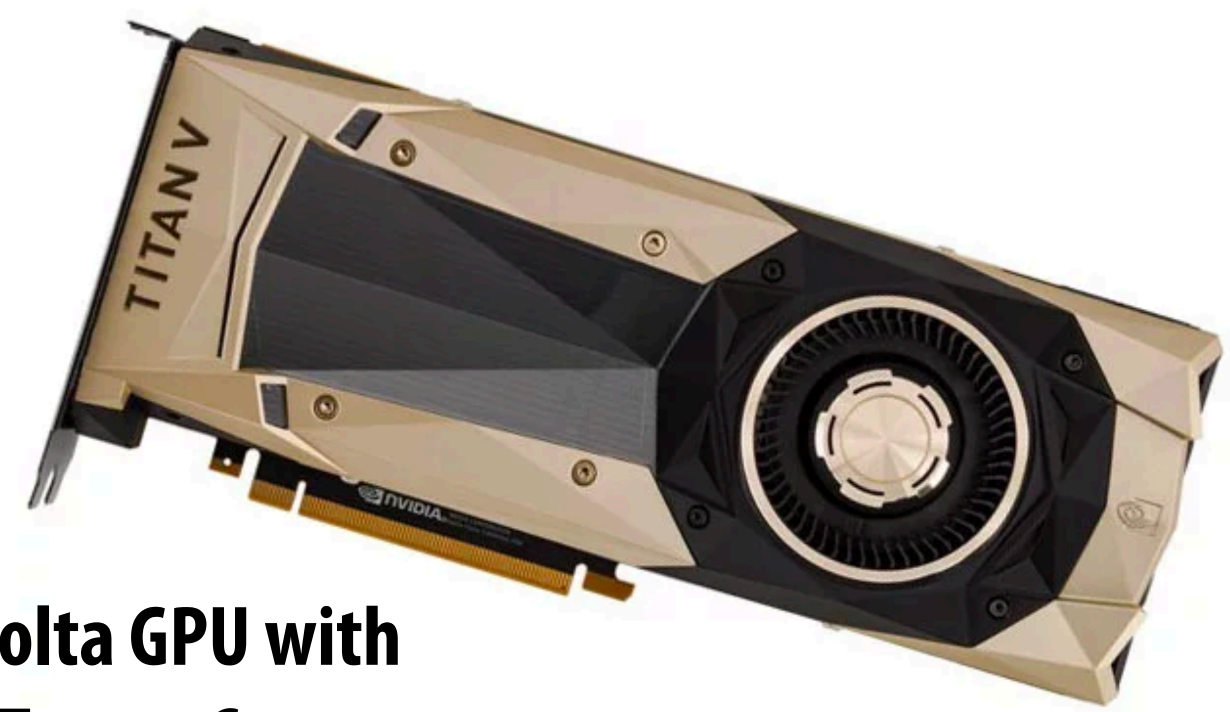
Apple Neural Engine



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)

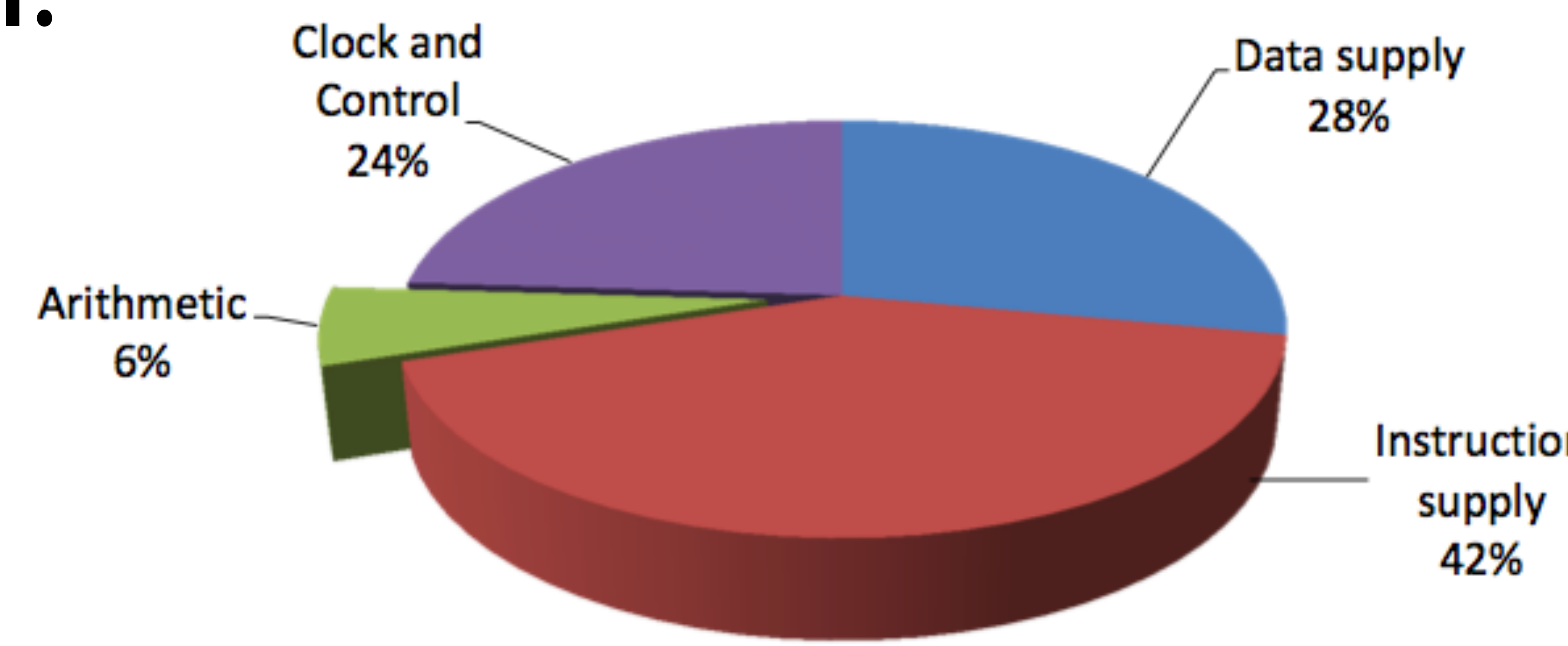
Recall properties of GPUs

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

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

- But also remember 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]

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

Google TPU
(version 1)

Discussion: workloads

- **What did TPU paper state about characteristics of modern DNN workloads at Google?**

Google's TPU

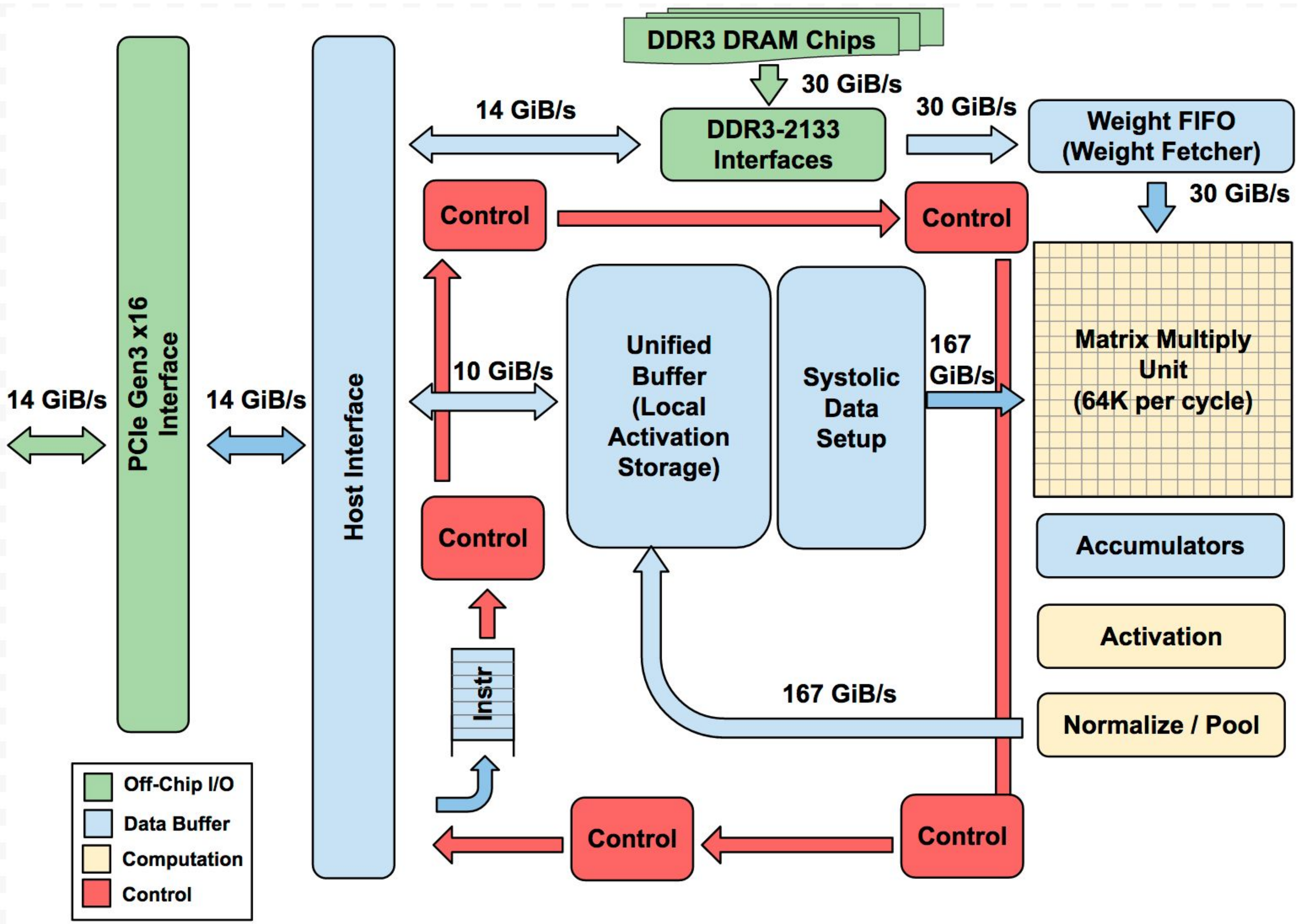
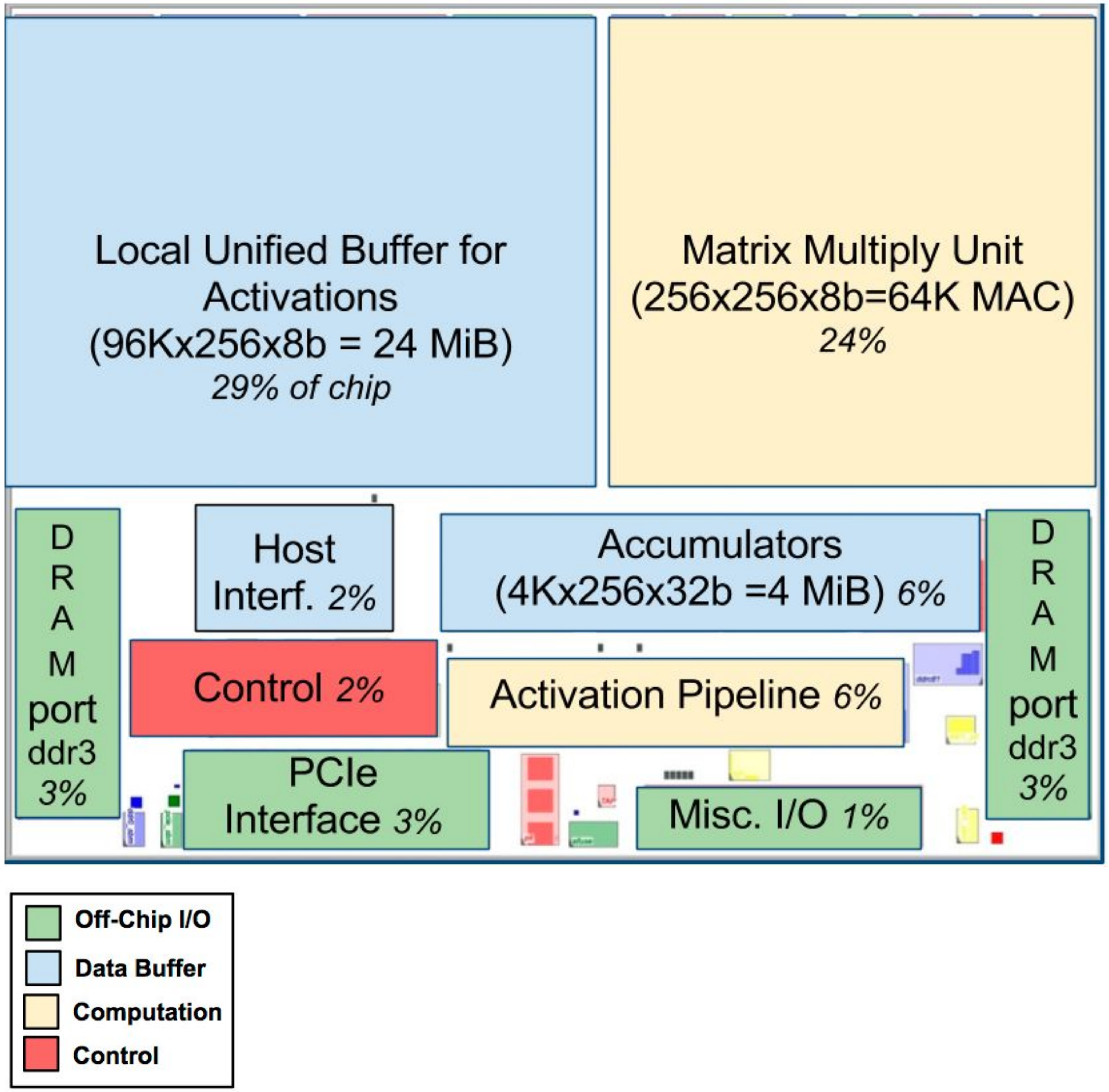


Figure credit: Jouppi et al. 2017

TPU area proportionality



Compute ~ 30% of chip

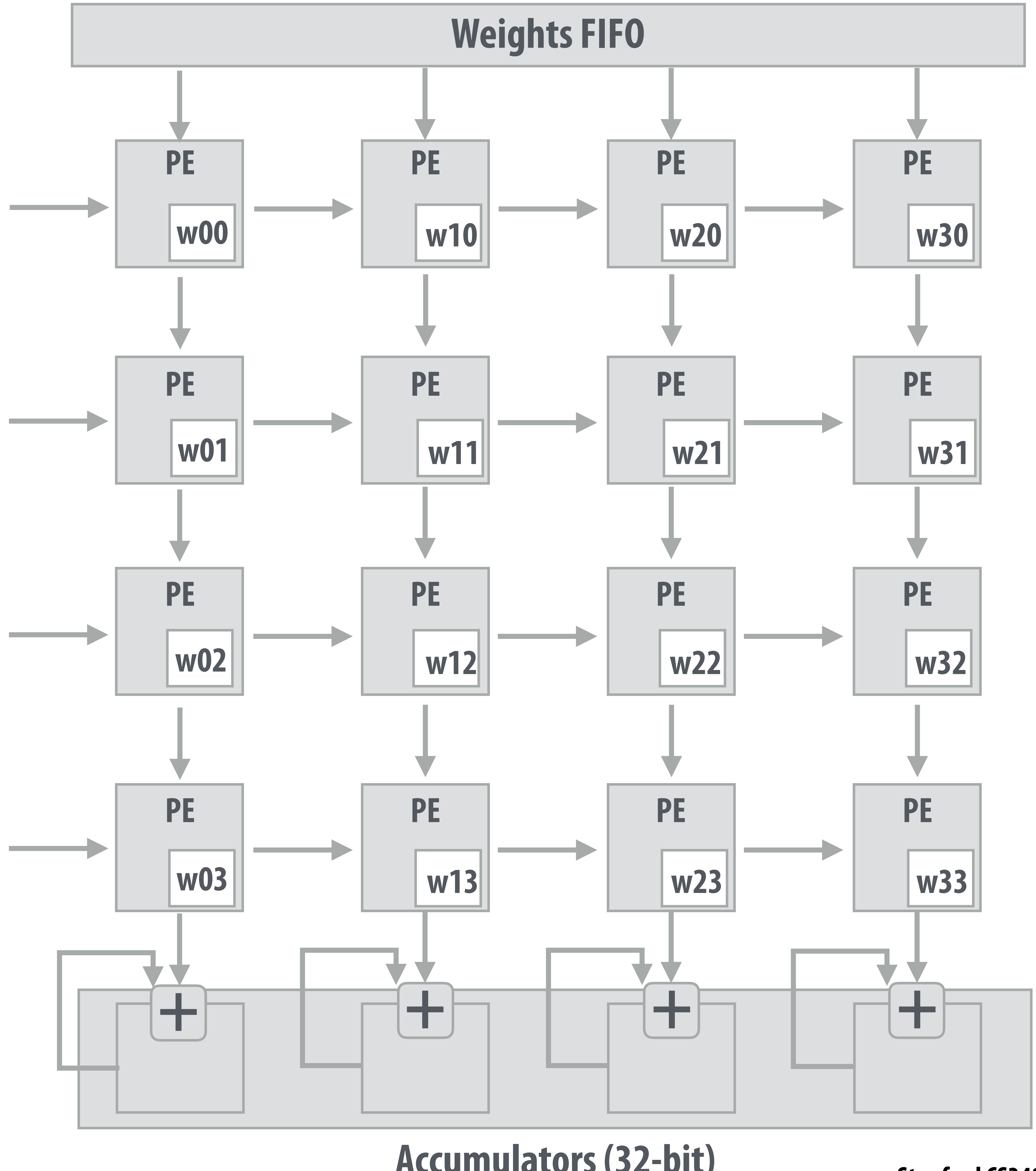
Note low area footprint of control

Key instructions:

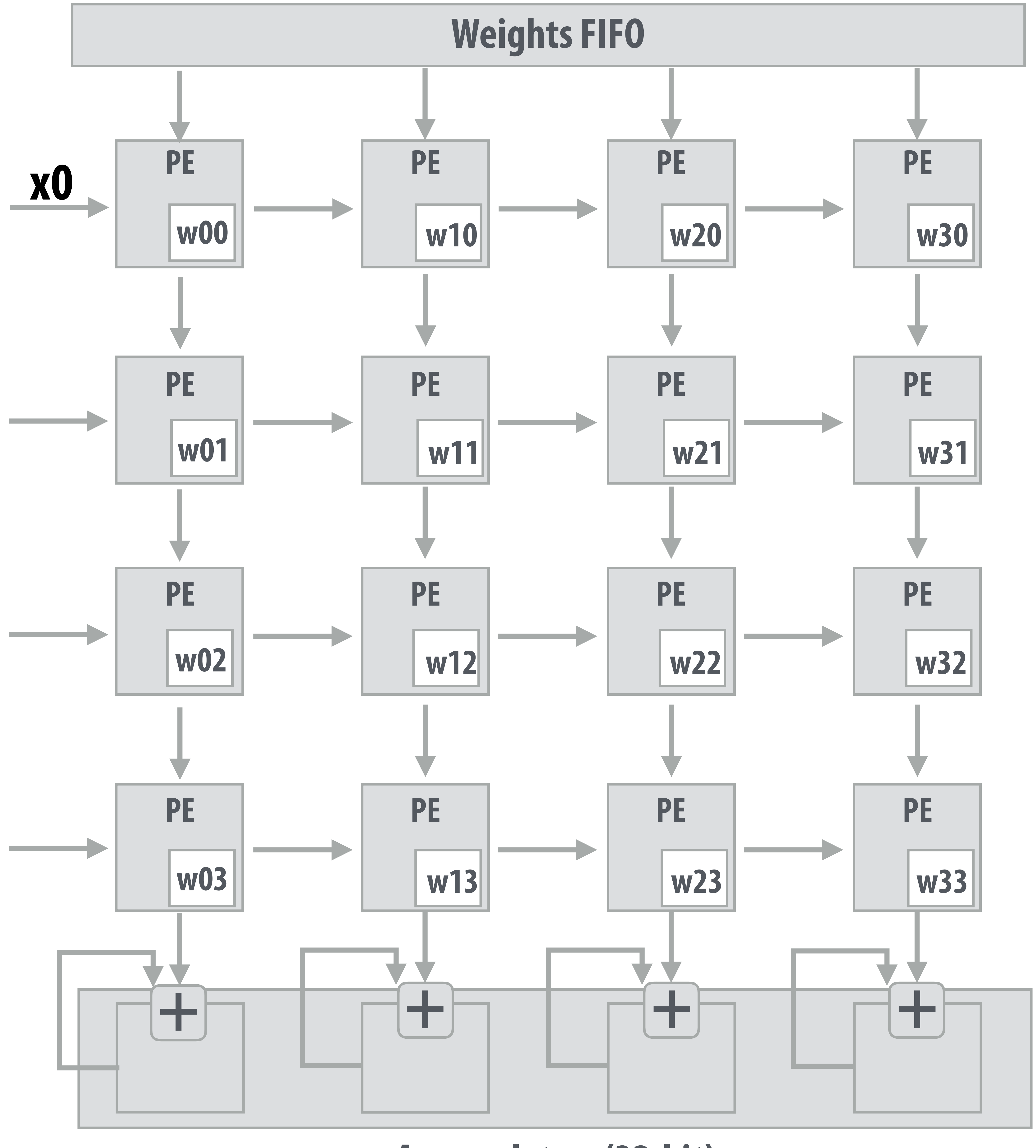
- read host memory
- write host memory
- read weights
- matrix_multiply / convolve
- activate

Figure credit: Jouppi et al. 2017

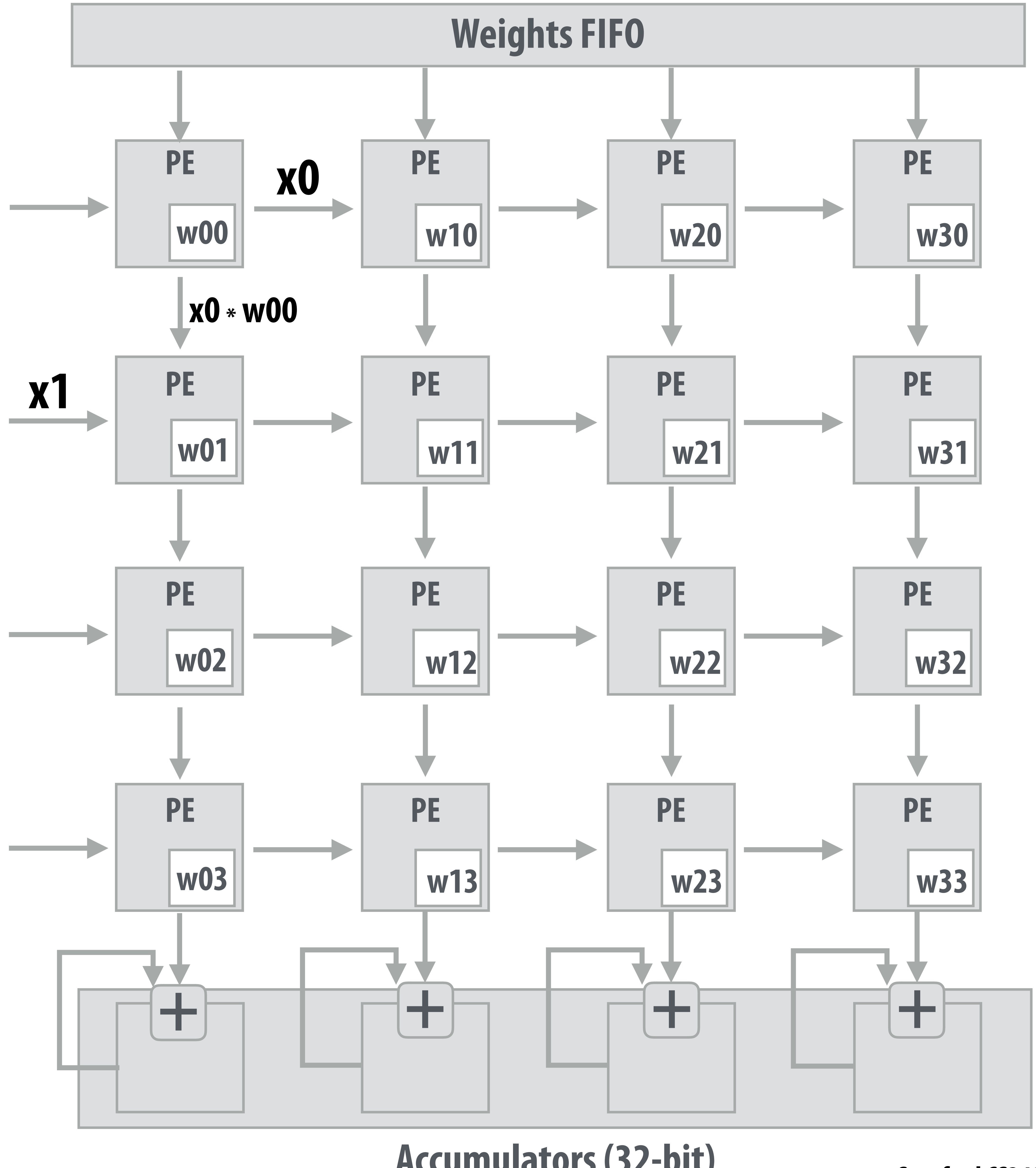
Systolic array (matrix vector multiplication example: $y=Wx$)



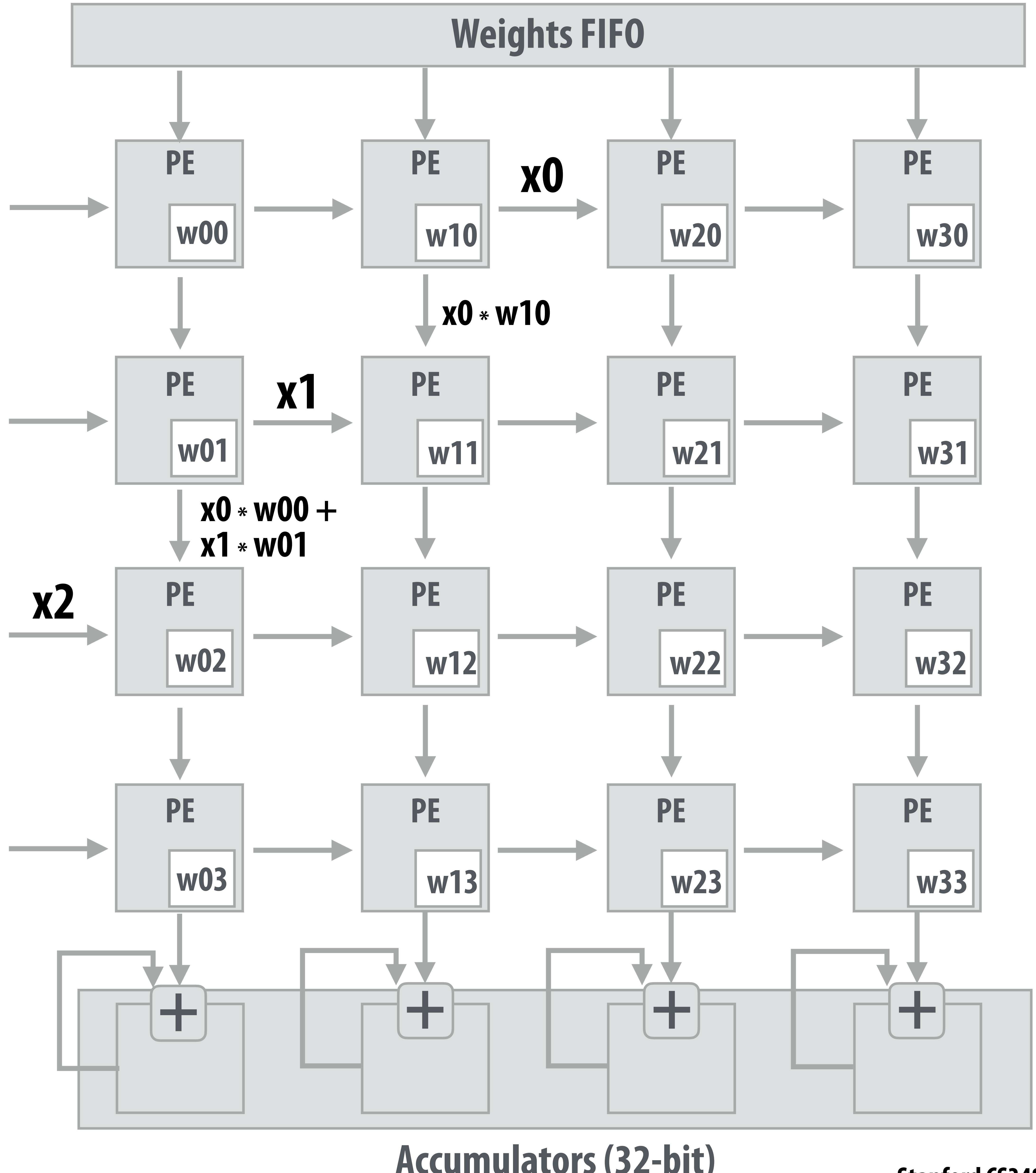
Systolic array (matrix vector multiplication example: $y = Wx$)



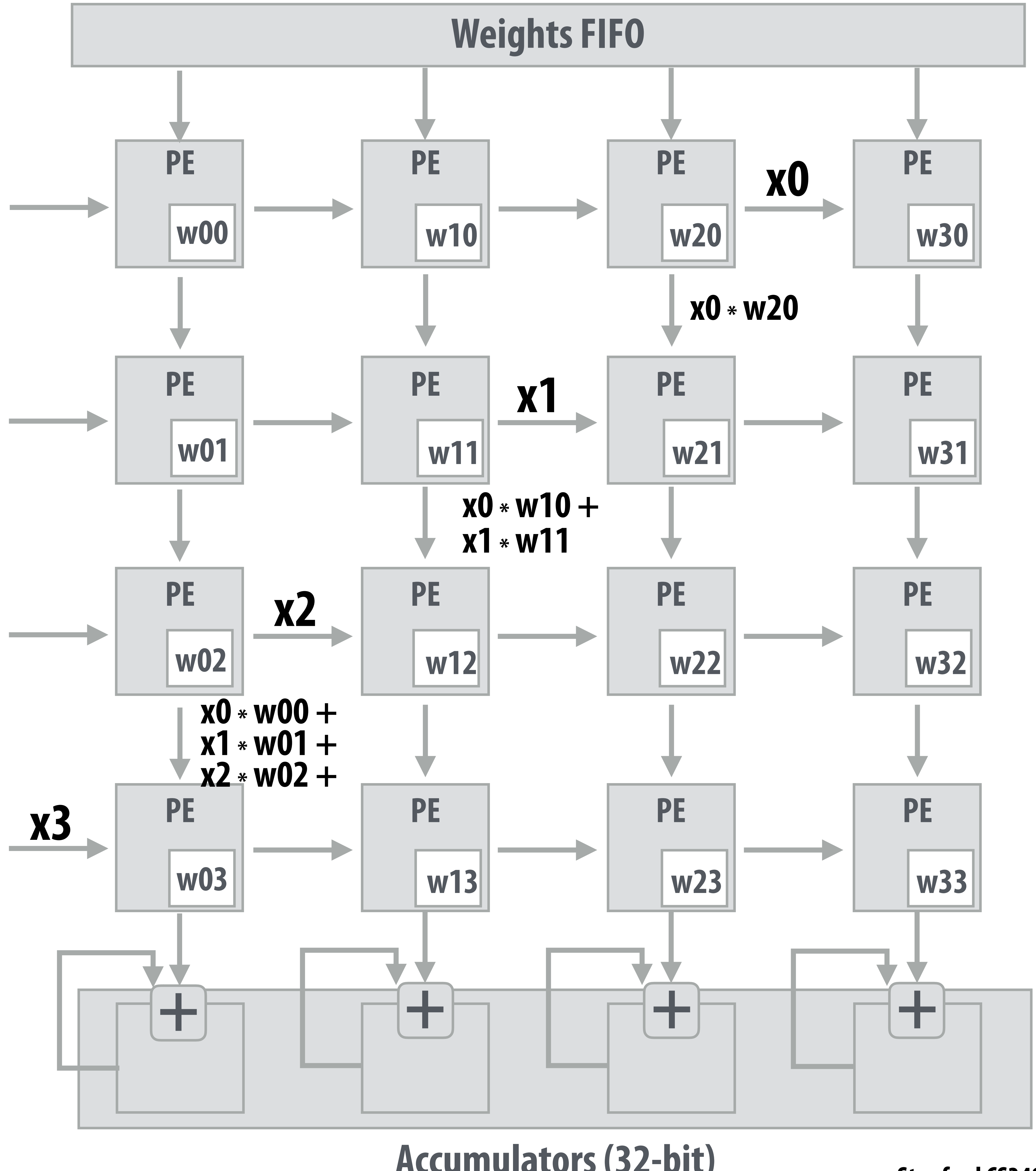
Systolic array (matrix vector multiplication example: $y = Wx$)



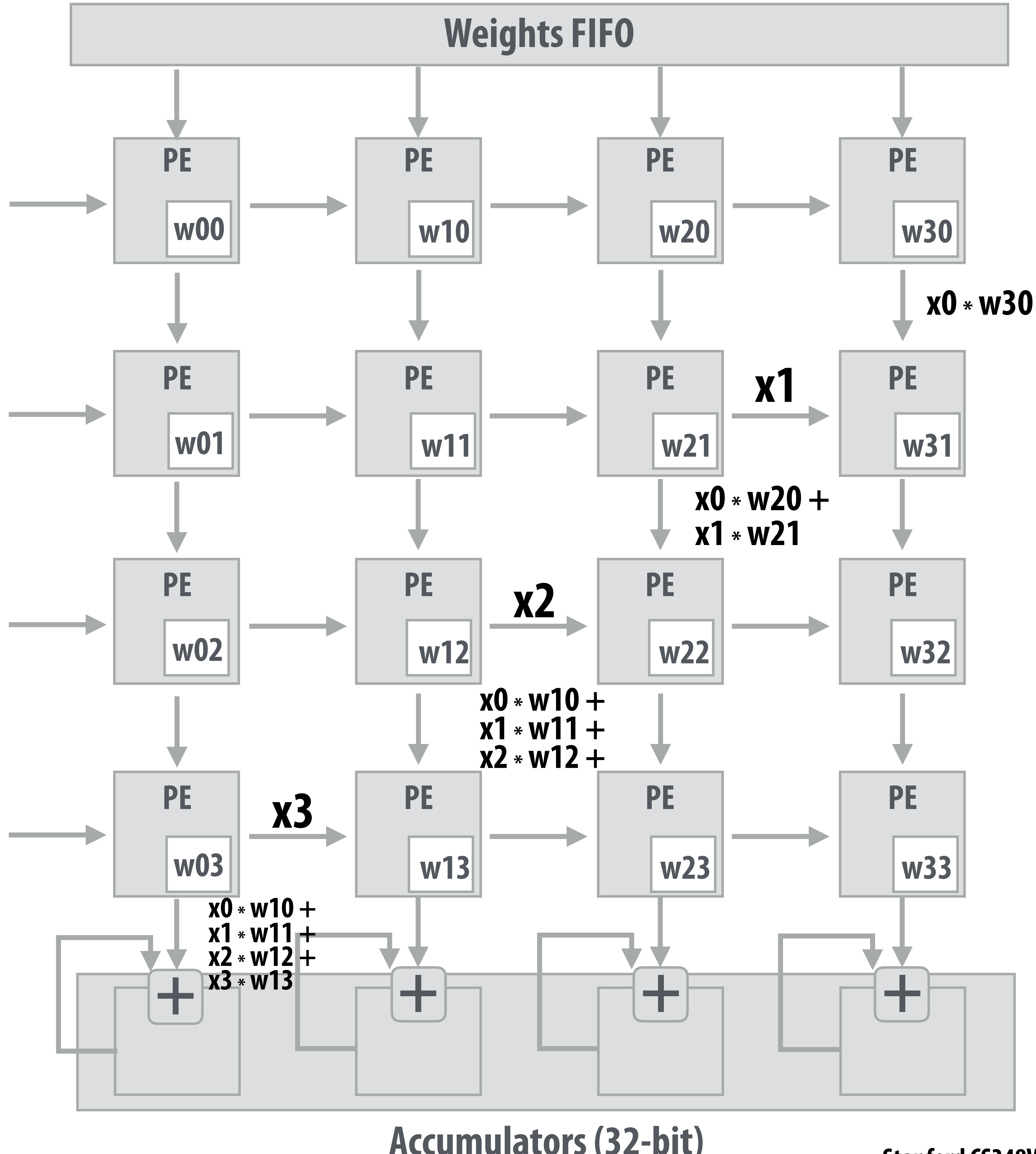
Systolic array (matrix vector multiplication example: $y = Wx$)



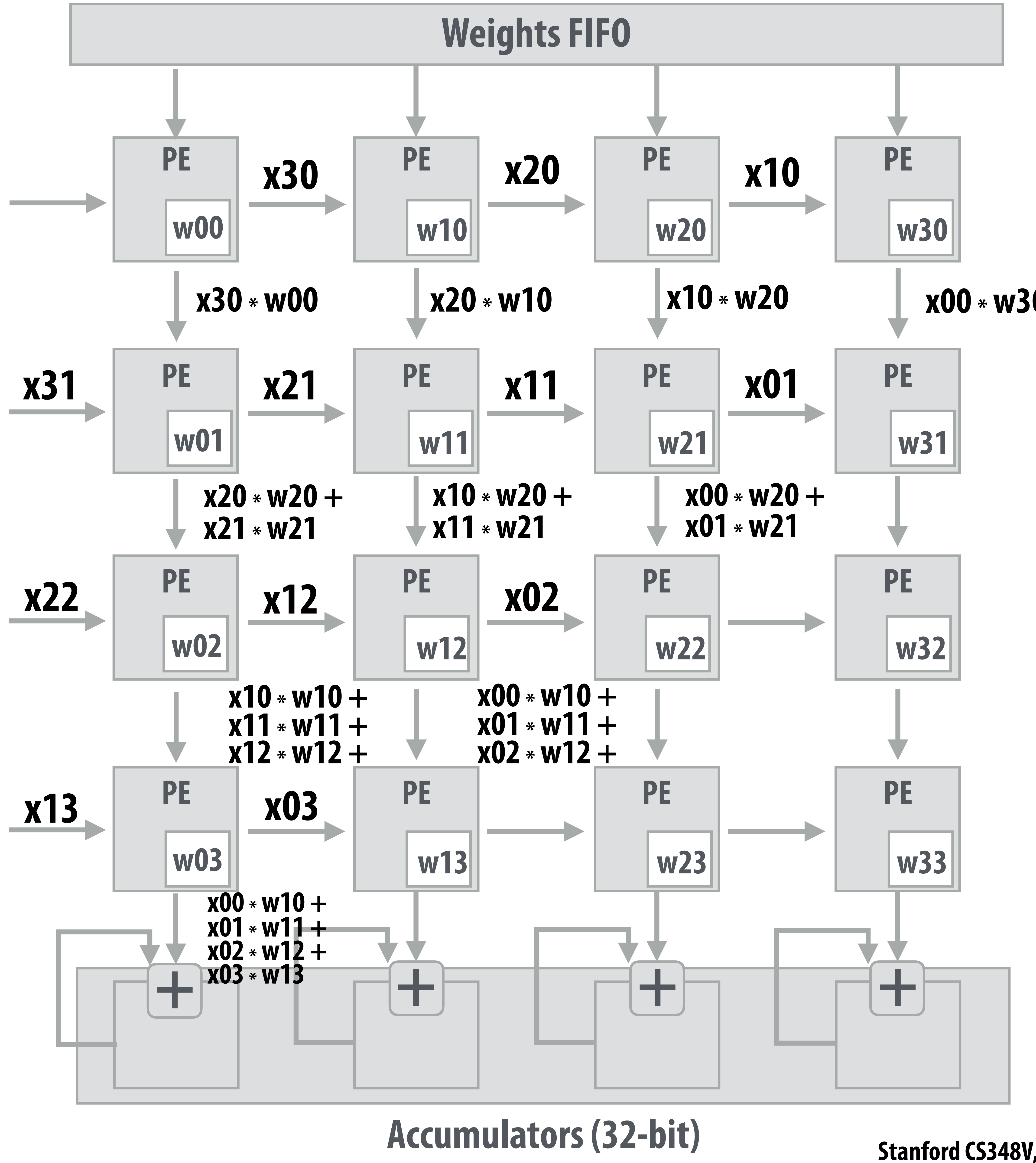
Systolic array (matrix vector multiplication example: $y=Wx$)



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



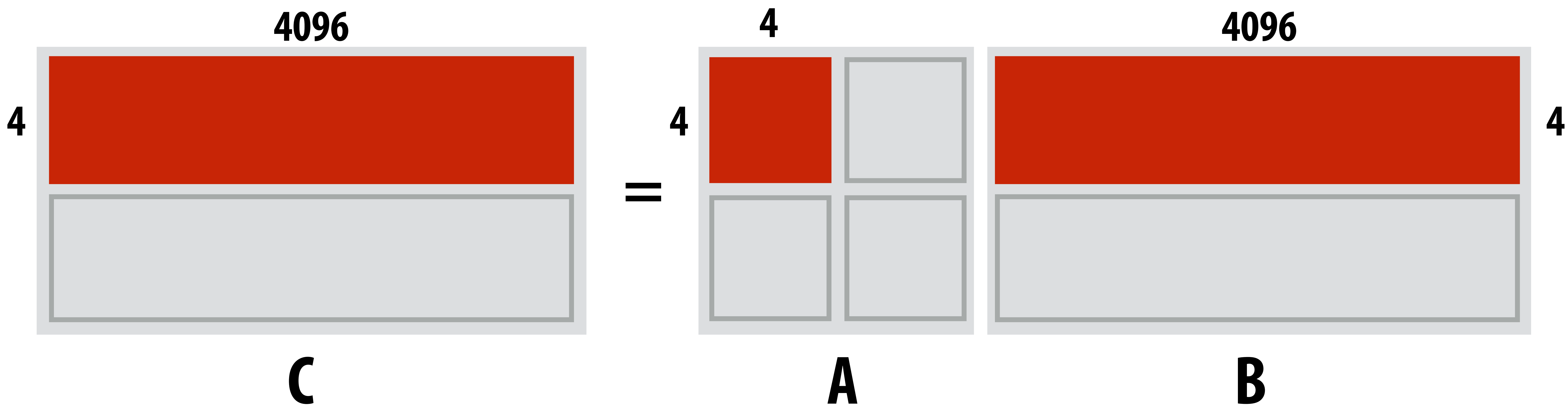
Systolic array (matrix matrix multiplication example: $Y=WX$)



Notice: need multiple 4x32bit accumulators to hold output columns

Building larger matrix-matrix multiplies

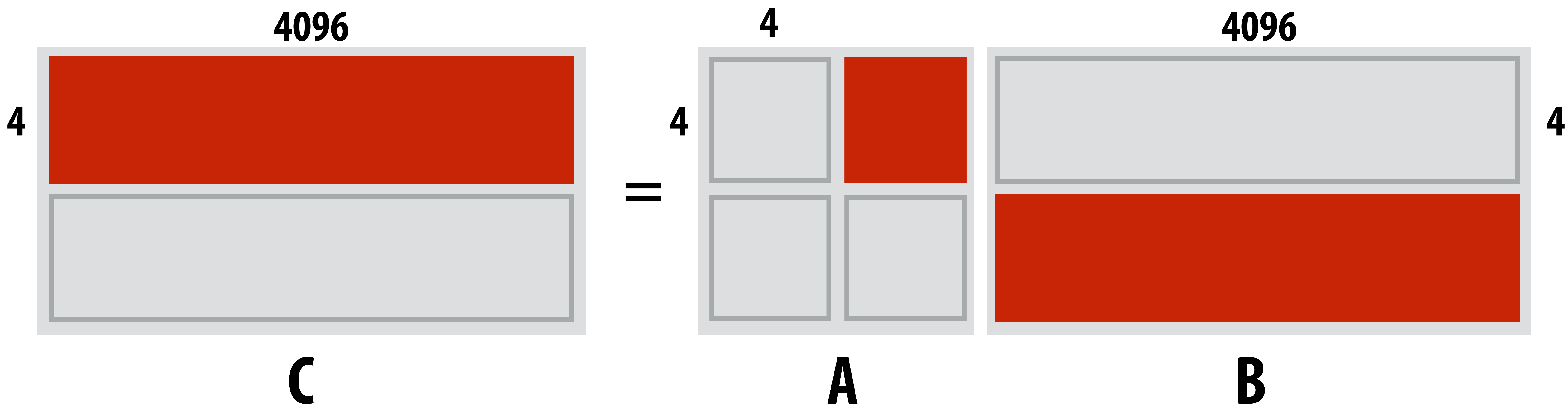
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

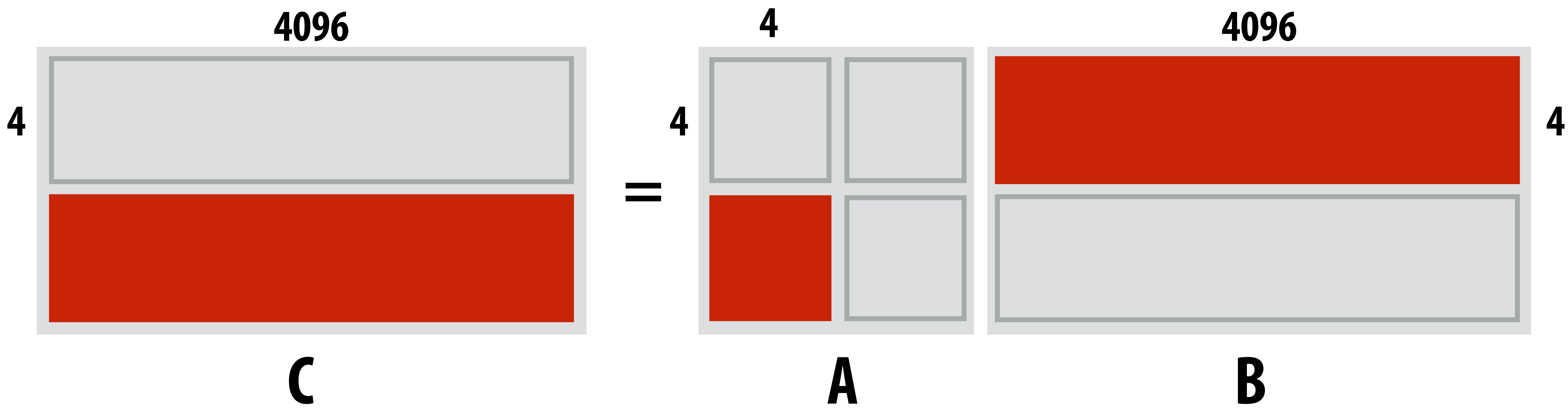
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

Building larger matrix-matrix multiplies

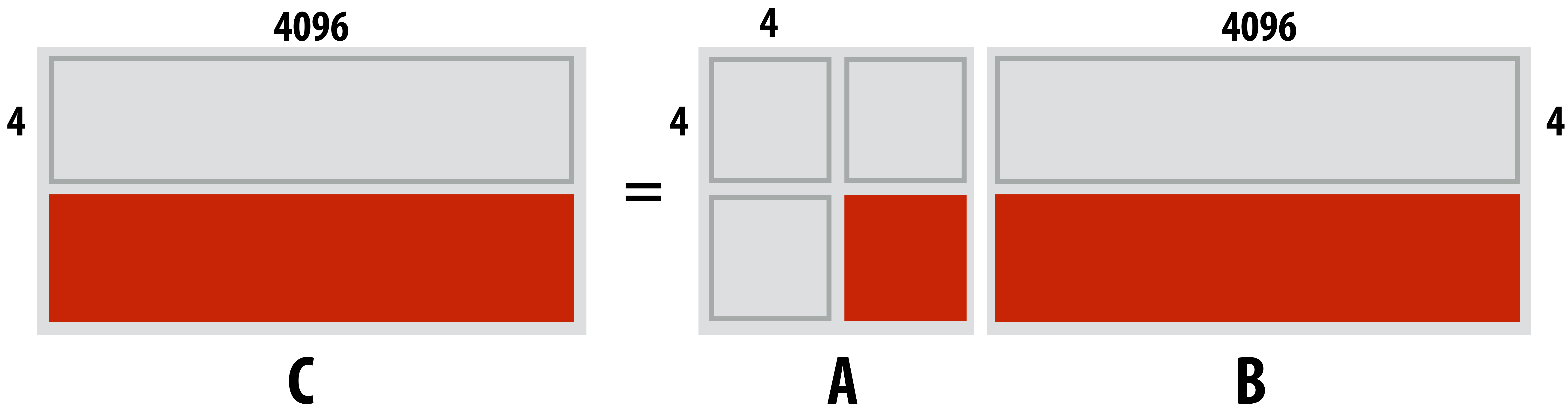
Example: A = 8x8, B= 8x4096, C=8x4096



Assume 4096 accumulators

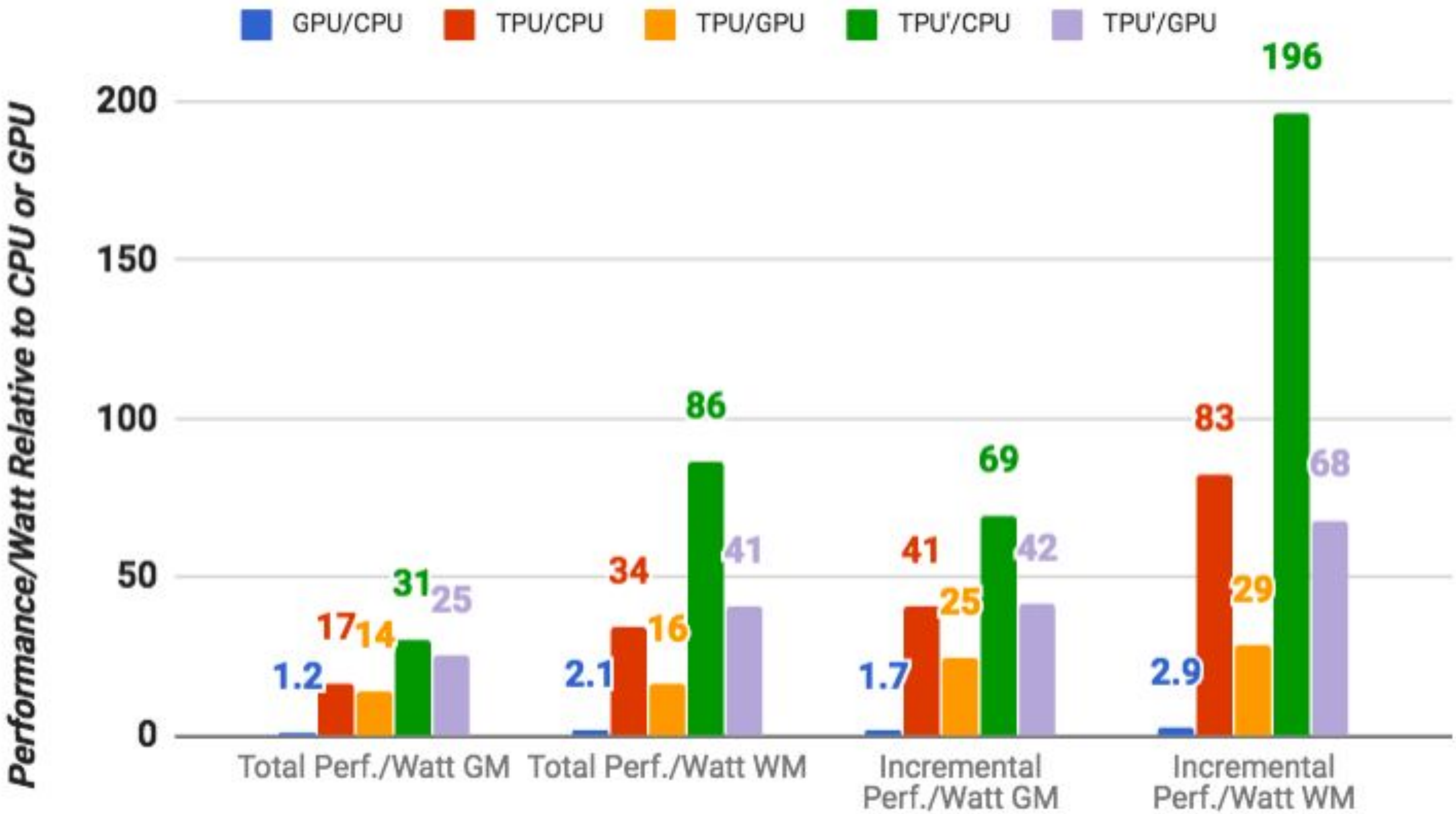
Building larger matrix-matrix multiplies

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



Assume 4096 accumulators

TPU Perf/Watt

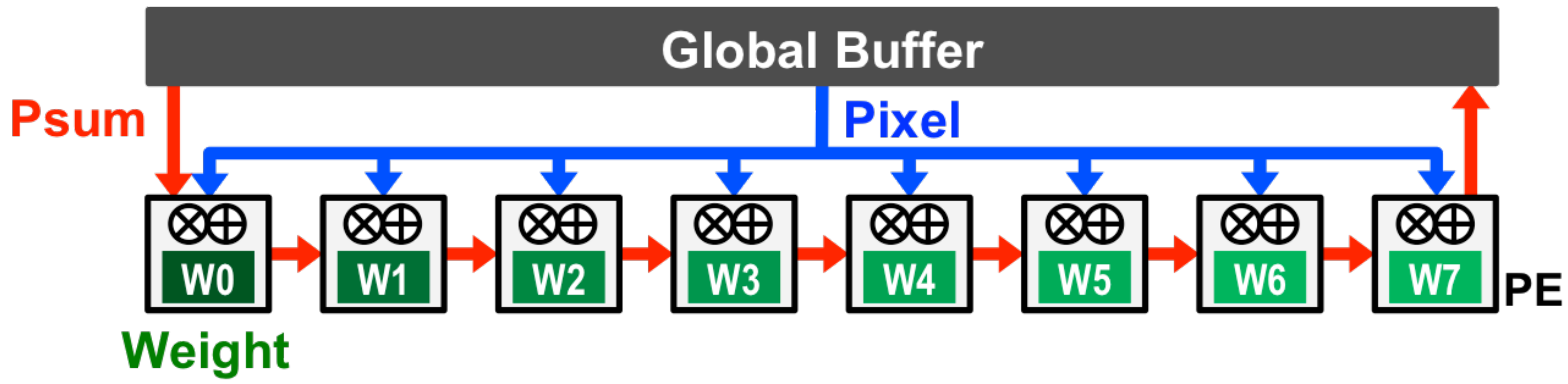


GM = geometric mean over all apps
WM = weighted mean over all apps

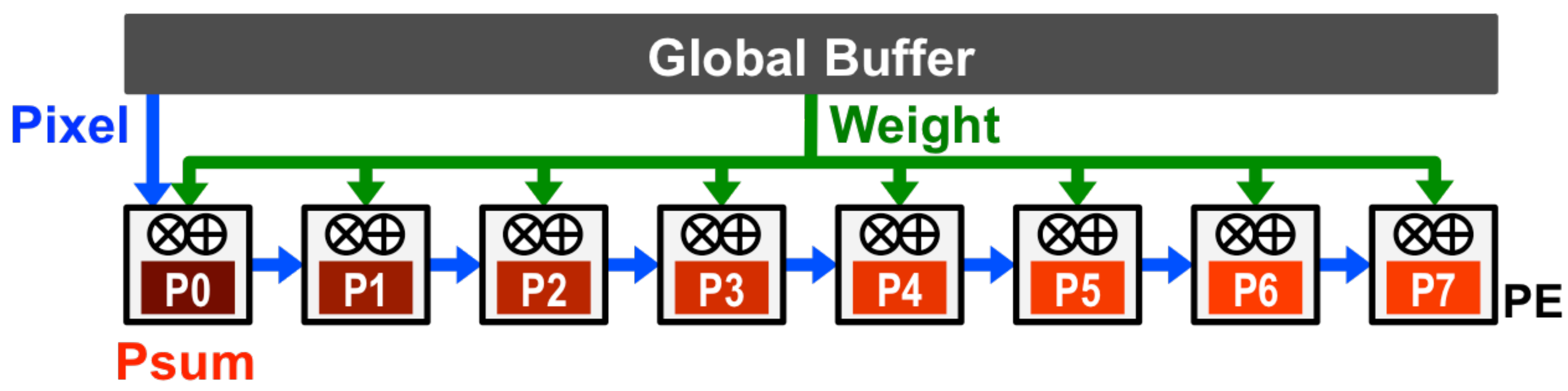
total = cost of host machine + CPU
incremental = only cost of TPU

Figure credit: Jouppi et al. 2017

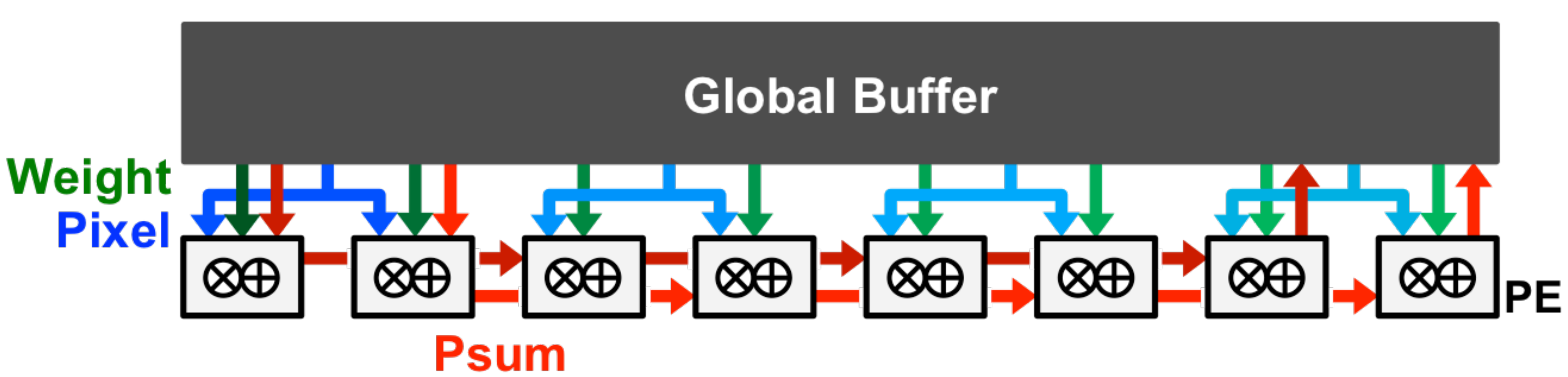
Alternative scheduling strategies



(a) Weight Stationary



(b) Output Stationary



(c) No Local Reuse

TPU was weight stationary (weights kept in register at PE)

Figure credit: Sze et al. 2017

EIE: targeting sparsified networks

Sparse, weight-sharing fully-connected layer

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

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 a

Note: activations are sparse due to ReLU

Efficient inference engine (EIE) ASIC

Custom hardware for decode and evaluate sparse, compressed DNNs

Hardware represents weight matrix in compressed sparse column (CSC) format to exploit sparsity in activations:

```
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:

```
int16* a_values;
PTR*   M_j_start;   // column j
int4*  M_j_values;
int4*  M_j_indices;
int16* lookup; // lookup table for
               // cluster values
```

```
for j=0 to length(a):
    if (a[j] == 0) continue;  // scan to nonzero
    col_values = M_j_values[M_j_start[j]];
    col_indices = M_j_indices[M_j_start[j]];
    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[M_j_values[i]] *
                a_values[j_count]
```

*** Keep in mind there's 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

$$\vec{a} \quad \begin{pmatrix} 0 & 0 & \boldsymbol{a_2} & 0 & a_4 & a_5 & 0 & a_7 \end{pmatrix}$$

$$\times$$

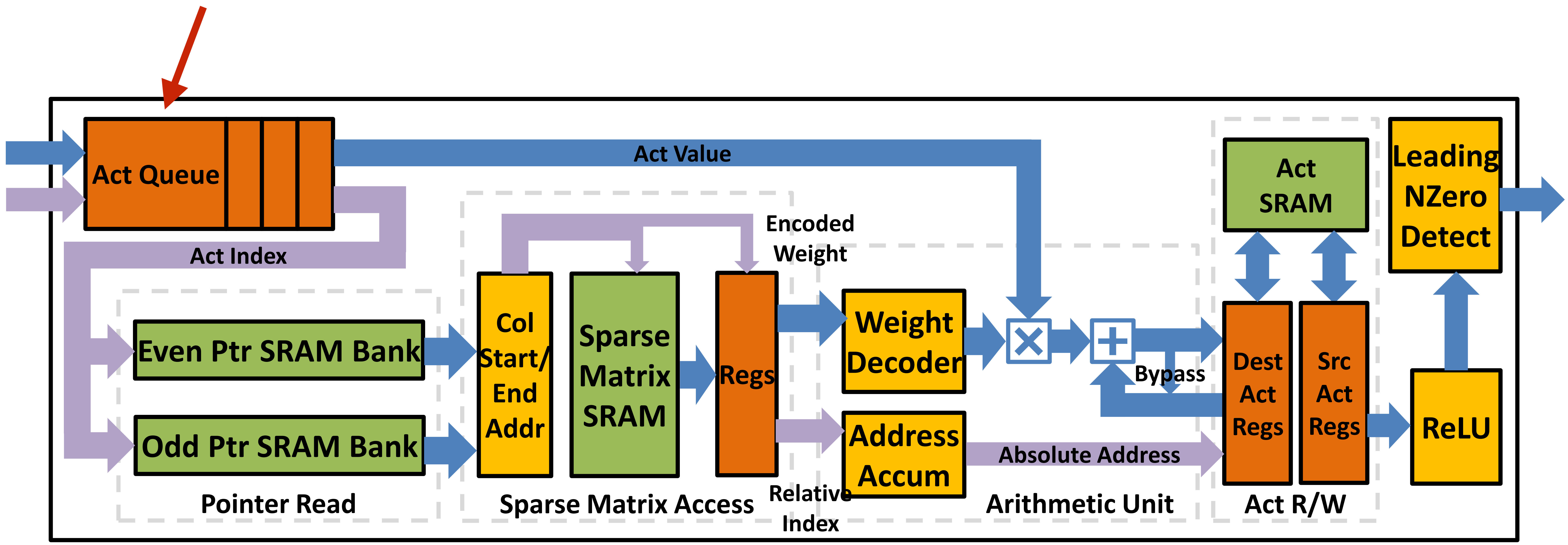
$$\begin{array}{c} PE0 \\ PE1 \\ PE2 \\ PE3 \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array}
\begin{pmatrix}
w_{0,0} & 0 & \boldsymbol{w_{0,2}} & 0 & w_{0,4} & w_{0,5} & w_{0,6} & 0 \\
0 & w_{1,1} & \mathbf{0} & w_{1,3} & 0 & 0 & w_{1,6} & 0 \\
0 & 0 & \boldsymbol{w_{2,2}} & 0 & w_{2,4} & 0 & 0 & w_{2,7}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

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

EIE unit for quantized sparse/matrix vector product

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



EIE Efficiency

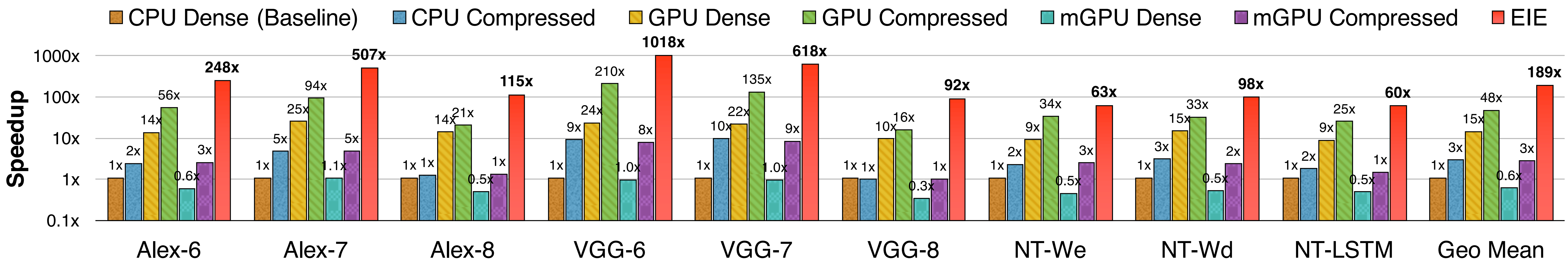
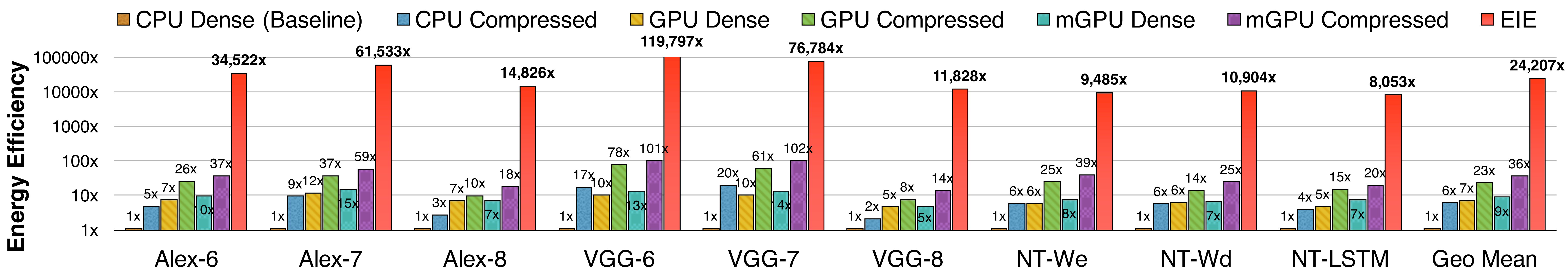


Figure 6. Speedups of GPU, mobile GPU and EIE compared with CPU running uncompressed DNN model. There is no batching in all cases.



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

Warning: these are not end-to-end: 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 inputs activations that are zero (65% less math)

Thoughts

- 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 solutions
  - Recall Inception, SqueezeNet, etc..

Summary of hardware techniques

- Specialized datapaths for dense linear algebra computations
  - Reduce overhead of control (compared to CPUs/GPUs)
- Reduced precision (computation and storage)
- Exploit sparsity
- Accelerate decompression