# ECE 598NSG/498NSU
# Deep Learning in Hardware
# Fall 2020

## DNN Accelerators: Systolic Arrays, TPU and Eyeriss Case Studies

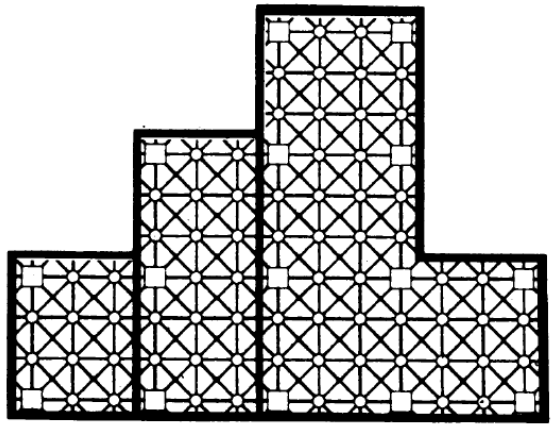
Naresh Shanbhag
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

http://shanbhag.ece.uiuc.edu

# Outline

- Systolic Arrays
- TPU Case Study
- Eyeriss Case study

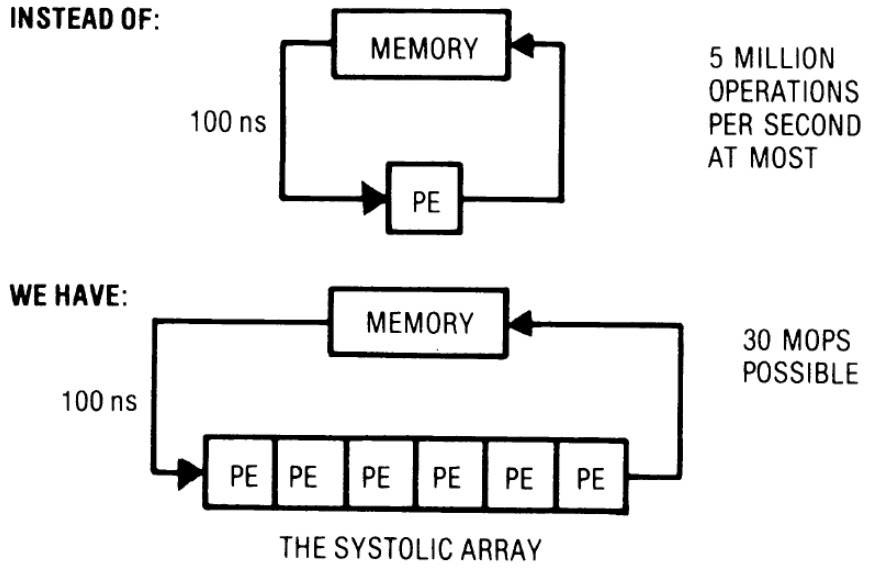
# Systolic Architectures



# *Why Systolic Architectures?*

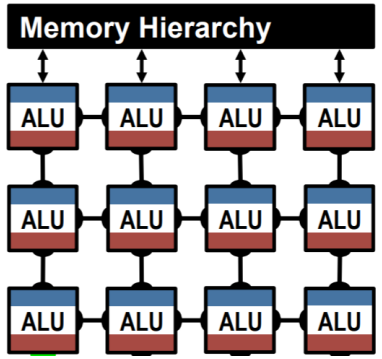
**H. T. Kung**
**Carnegie-Mellon University**

[Kung, IEEE Comp'1982]




- Main idea: have multiple PEs communicate locally to reduce memory bandwidth requirements
- The idea of systolic architectures has been around since the 80's!
- Employed in the design of today's digital DNN accelerators [Eyeriss-ISCA'16/ISSCC'16] & [TPU-ISCA'17]

**Eyeriss'16**




**TPU'17**

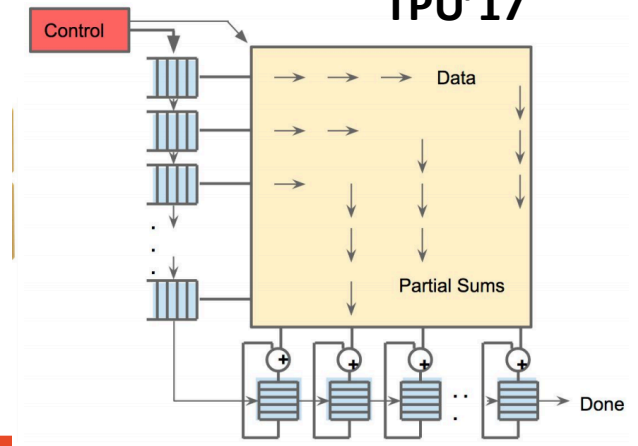
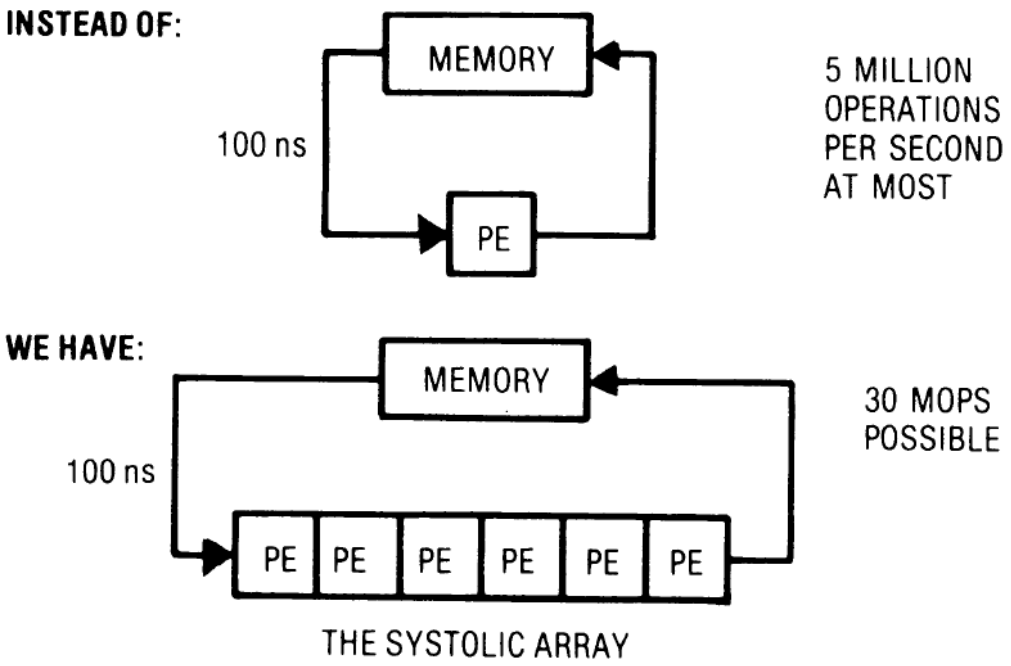


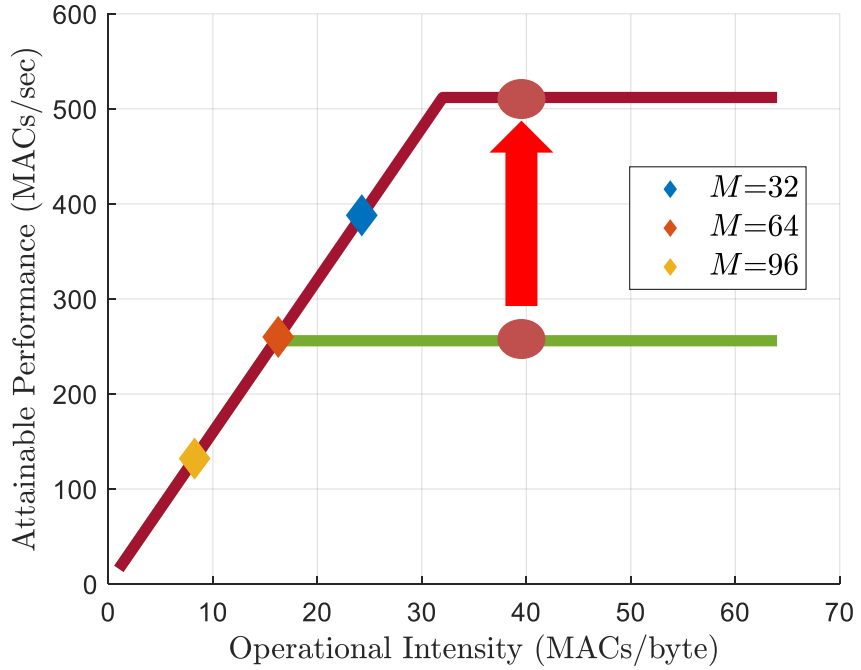

**Figure 4.** Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

# Systolic Architectures




- consists of a set of interconnected PEs, each capable of performing a simple operation (MAC for example) and small local storage
- information flows between PEs in a pipelined (synchronous) fashion

# A Roofline Perspective




- useful for compute bound problems where memory I/O is a bottleneck
- data can either be broadcasted from the main memory to a subset of PEs or streamed in from one PE to the other

# Example Workload: 1D Convolution

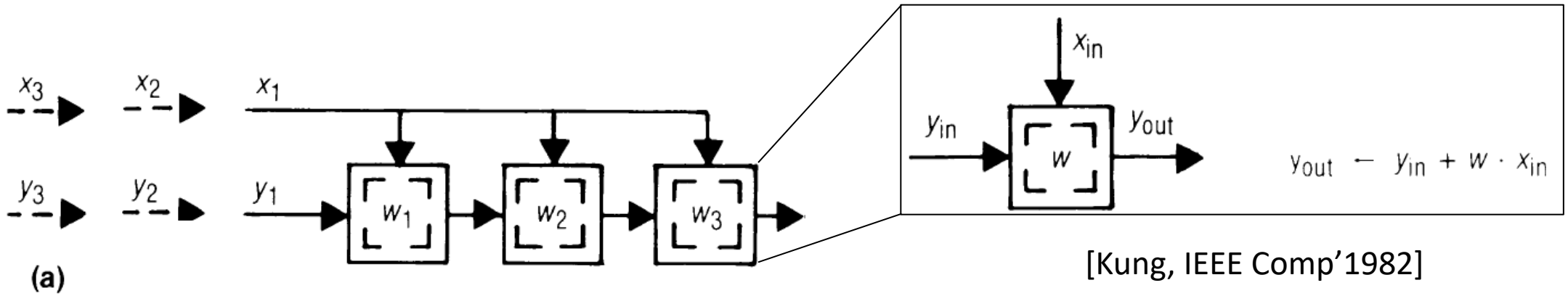
- Workload: perform a 1D "convolution" (correlator):

$$y_i = \sum_{j=1}^{M} w_j x_{i+j-1}$$

where $x \in \mathbb{R}^N, w \in \mathbb{R}^M \Rightarrow y \in \mathbb{R}^{N-M+1}$, and $M \in [N]$

assume 32b floating point operations

# Weight Stationary




[Kung, IEEE Comp'1982]

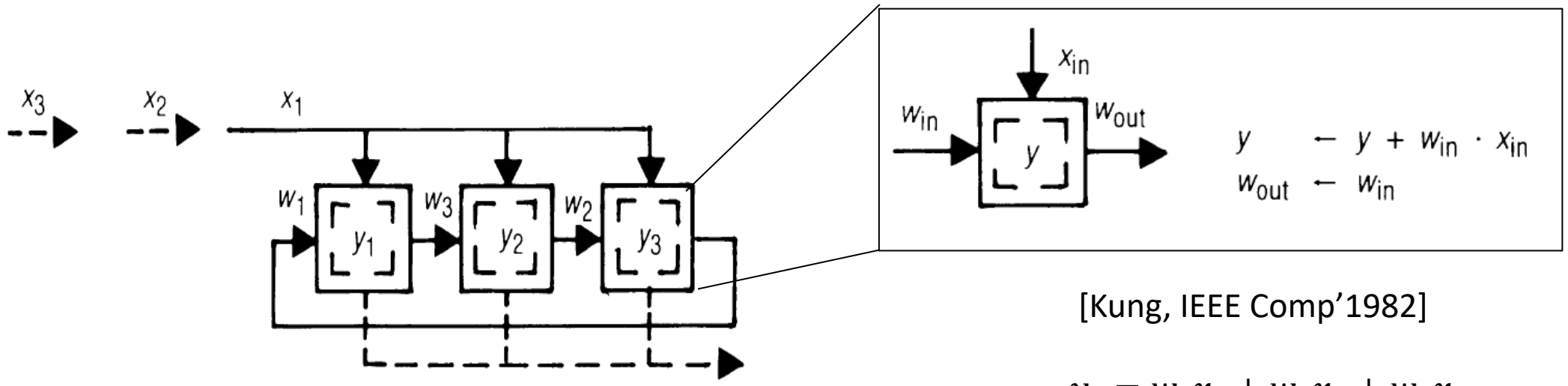
$$y_1 = w_1x_1 + w_2x_2 + w_3x_3$$

$$y_2 = w_1x_2 + w_2x_3 + w_3x_4$$

$$y_3 = w_1x_3 + w_2x_4 + w_3x_5$$

- Example with $N = 5$ & $M = 3$
- weights remain stationary in each PE
- inputs are broadcast to all PEs
- partial sums move from left to right (initially they are zero for the left-most PE)
- Right-most PE produces a full output $y_i$
- also called transpose FIR filter structure in DSP

# Output Stationary




[Kung, IEEE Comp'1982]

$$y_1 = w_1x_1 + w_2x_2 + w_3x_3$$

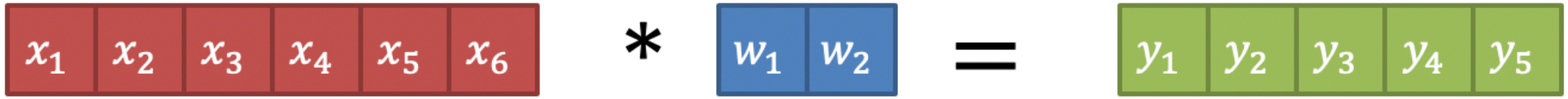
$$y_2 = w_1x_2 + w_2x_3 + w_3x_4$$

$$y_3 = w_1x_3 + w_2x_4 + w_3x_5$$

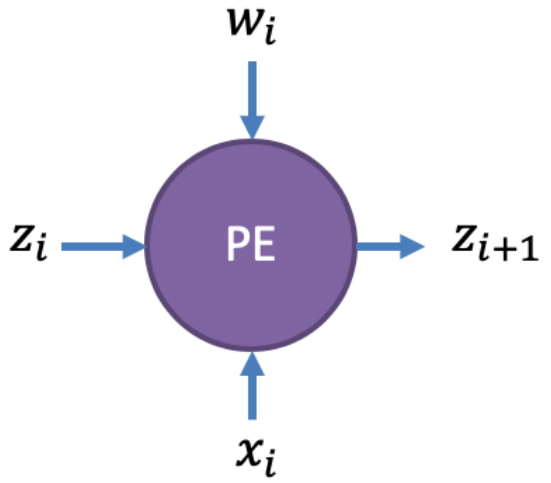
- Example with $N = 5$ & $M = 3$
- inputs are broadcast
- weights move in a loop across all PEs
- outputs are locally accumulated in each PE
- Accumulation starts when $w_1$ arrives in each PE

# Weight Stationary – animated

- Consider the 1D convolution example with $N = 6$ and $M = 2$

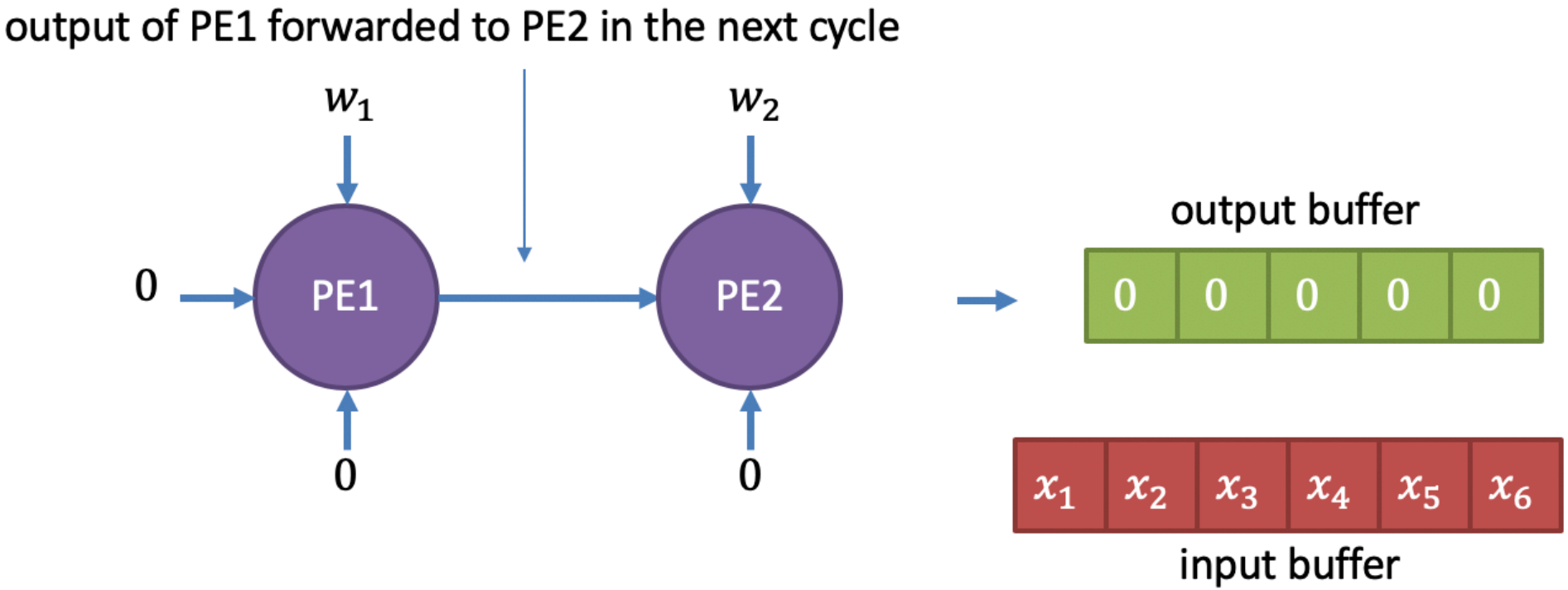



- We are given 2 PEs:
- $z_{i+1} \leftarrow z_i + w_i \times x_i$

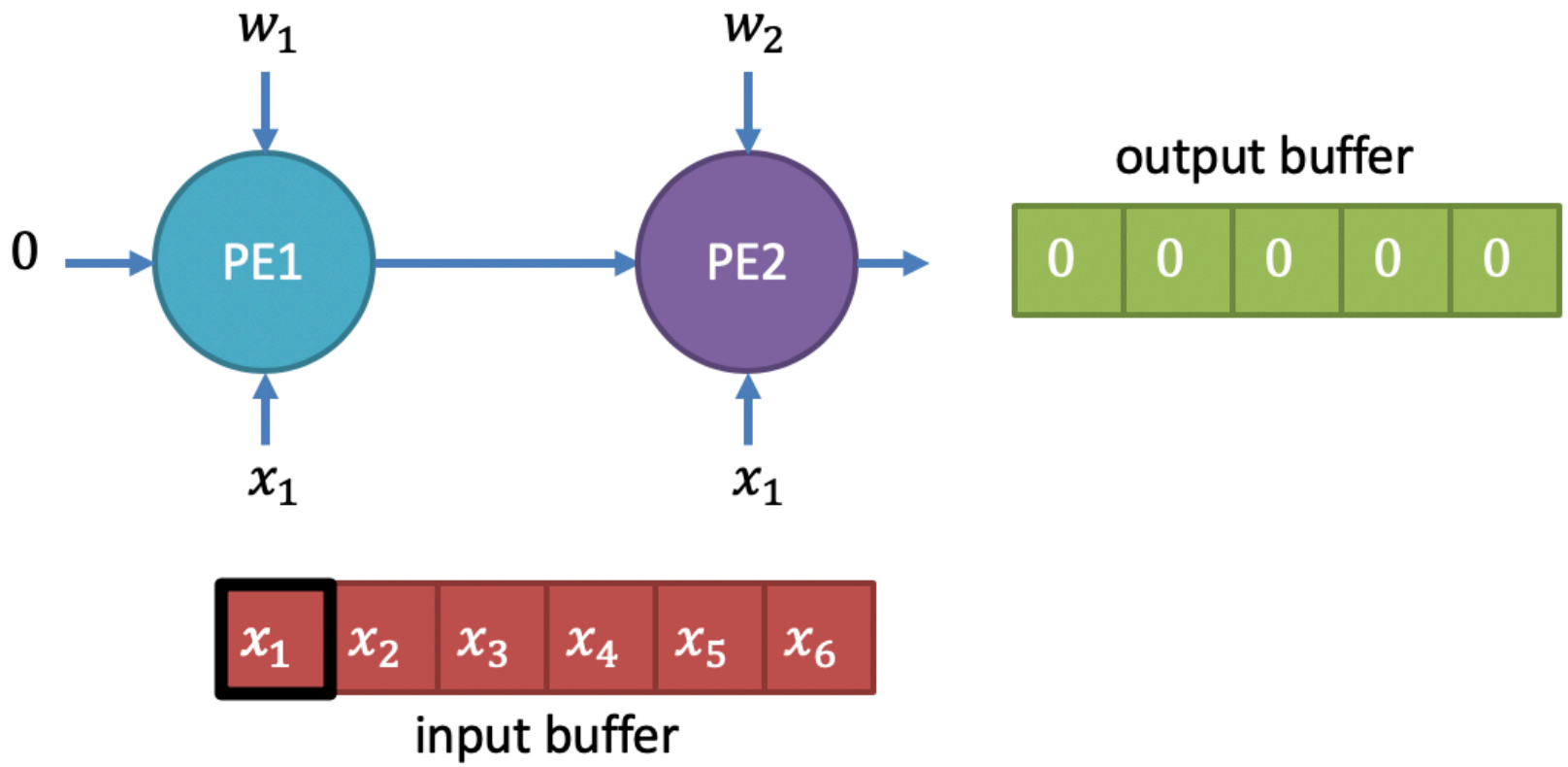


# Weight Stationary

- Store $w_1$ & $w_2$ in the scratchpads of each PE
- Inputs are being *broadcast* are stored in a buffer
- Outputs will be stored in another buffer

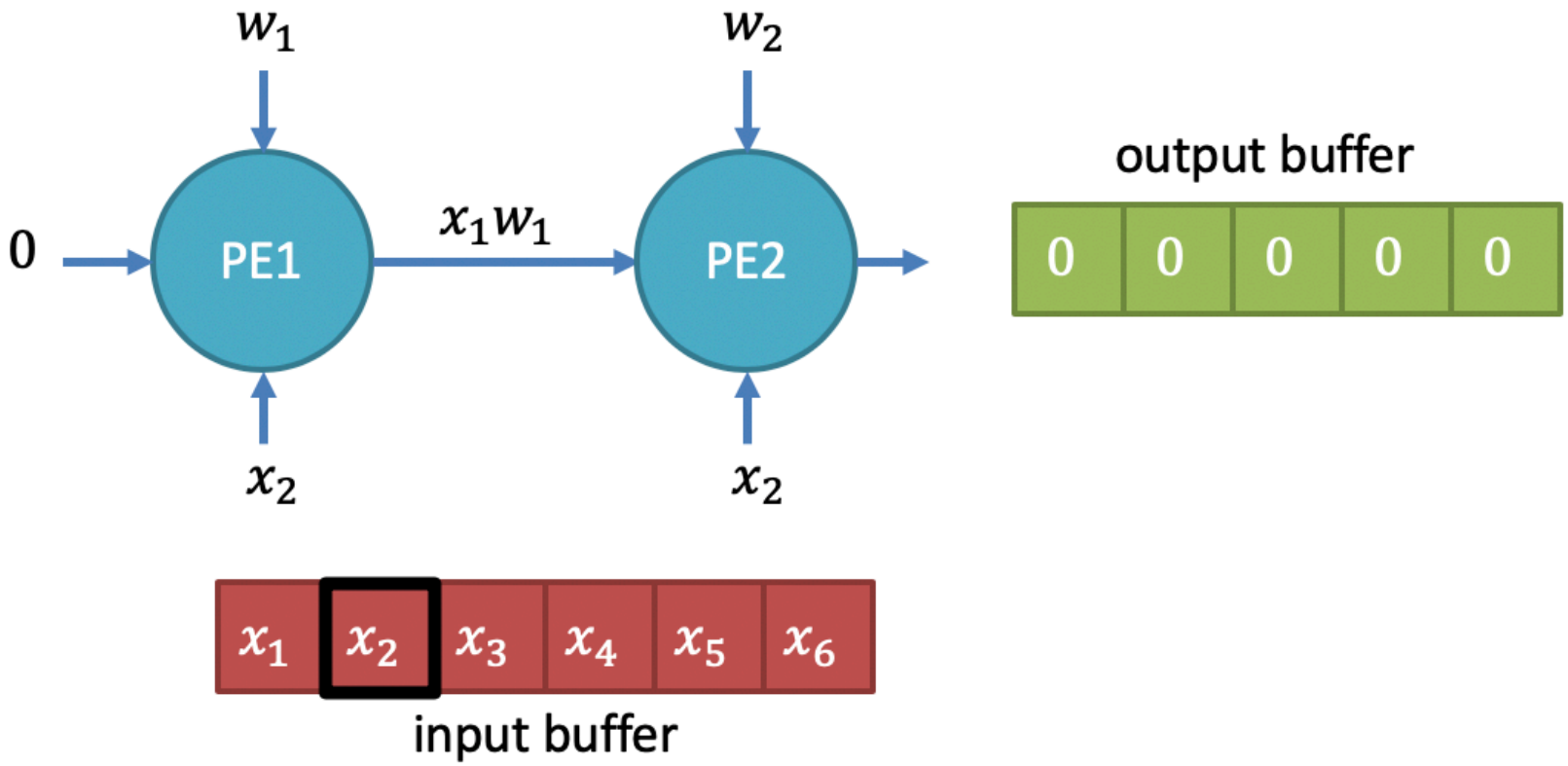


# Weight Stationary – cycle 1



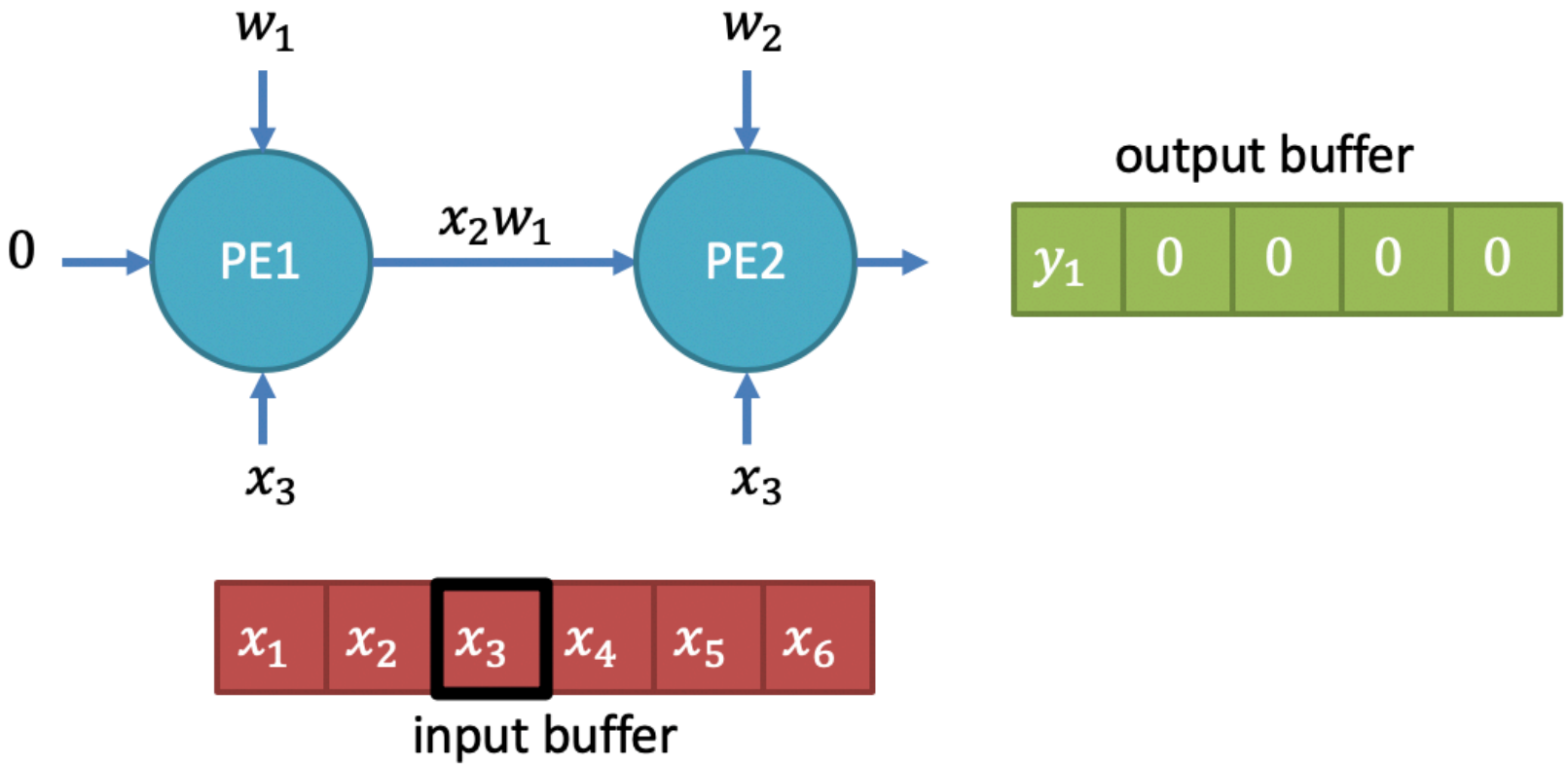

- $x_1$ is broadcasted to the two PEs
- PE1 performs $z = x_1 \times w_1 + 0$
- PE2 remains idle

# Weight Stationary – cycle 2



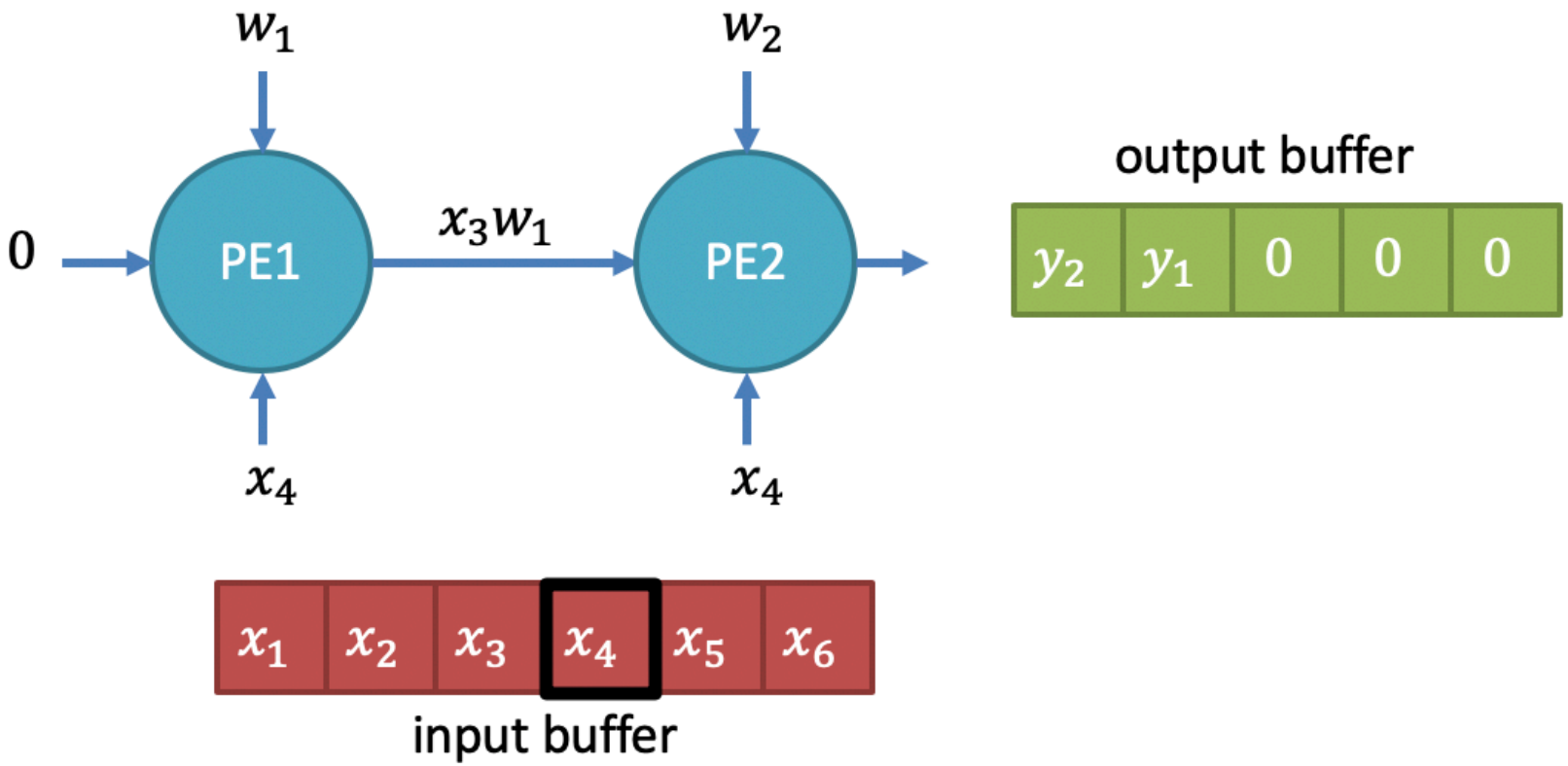

- $x_2$ is broadcasted to the two PEs
- PE1 performs $z = x_2 \times w_1 + 0$
- PE2 performs $z = x_2 \times w_2 + x_1 \times w_1$

# Weight Stationary – cycle 3



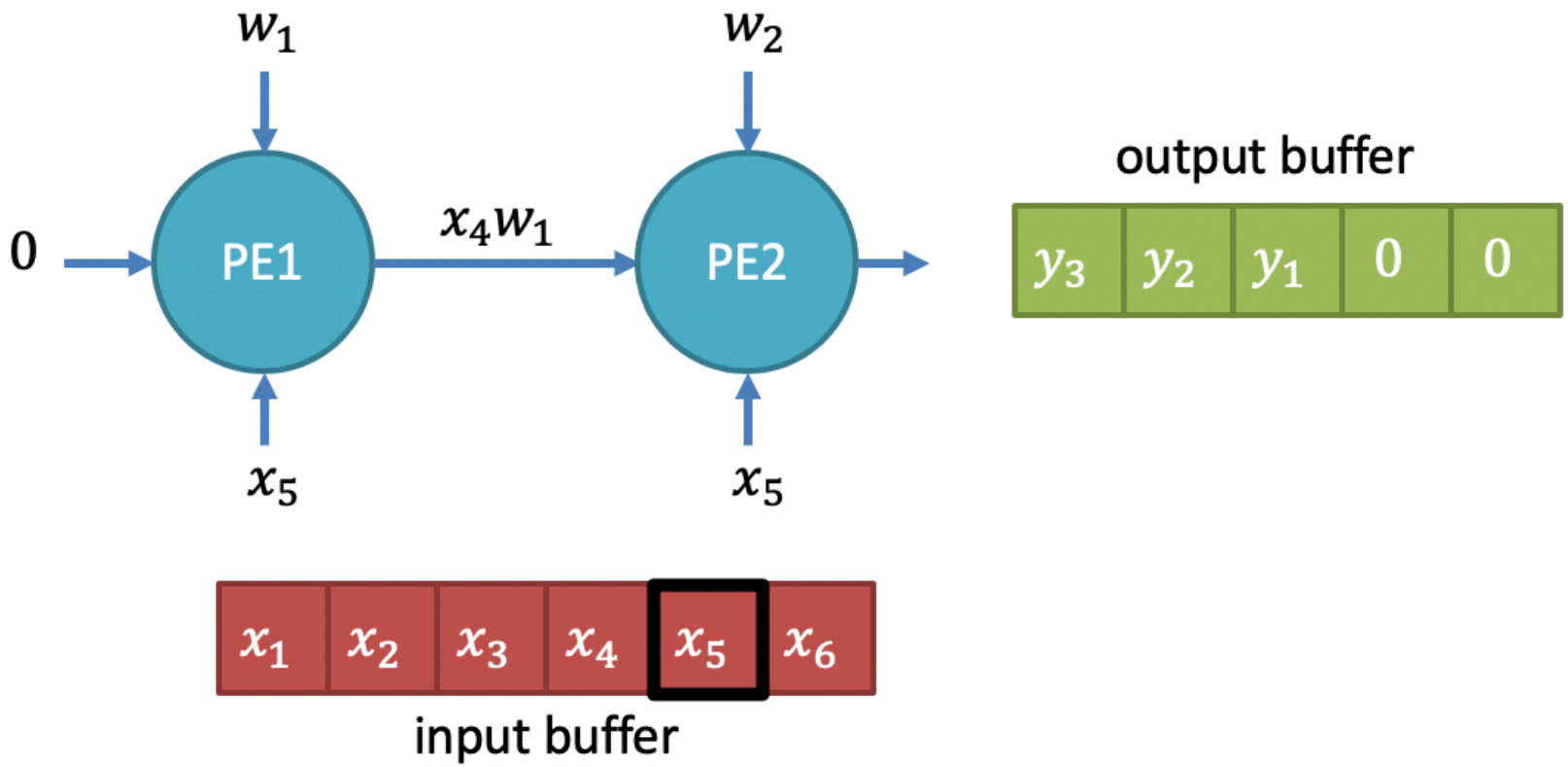

- $x_3$ is broadcasted to the two PEs
- PE1 performs $z = x_3 \times w_1 + 0$
- PE2 performs $z = x_3 \times w_2 + x_2 \times w_1$

# Weight Stationary – cycle 4



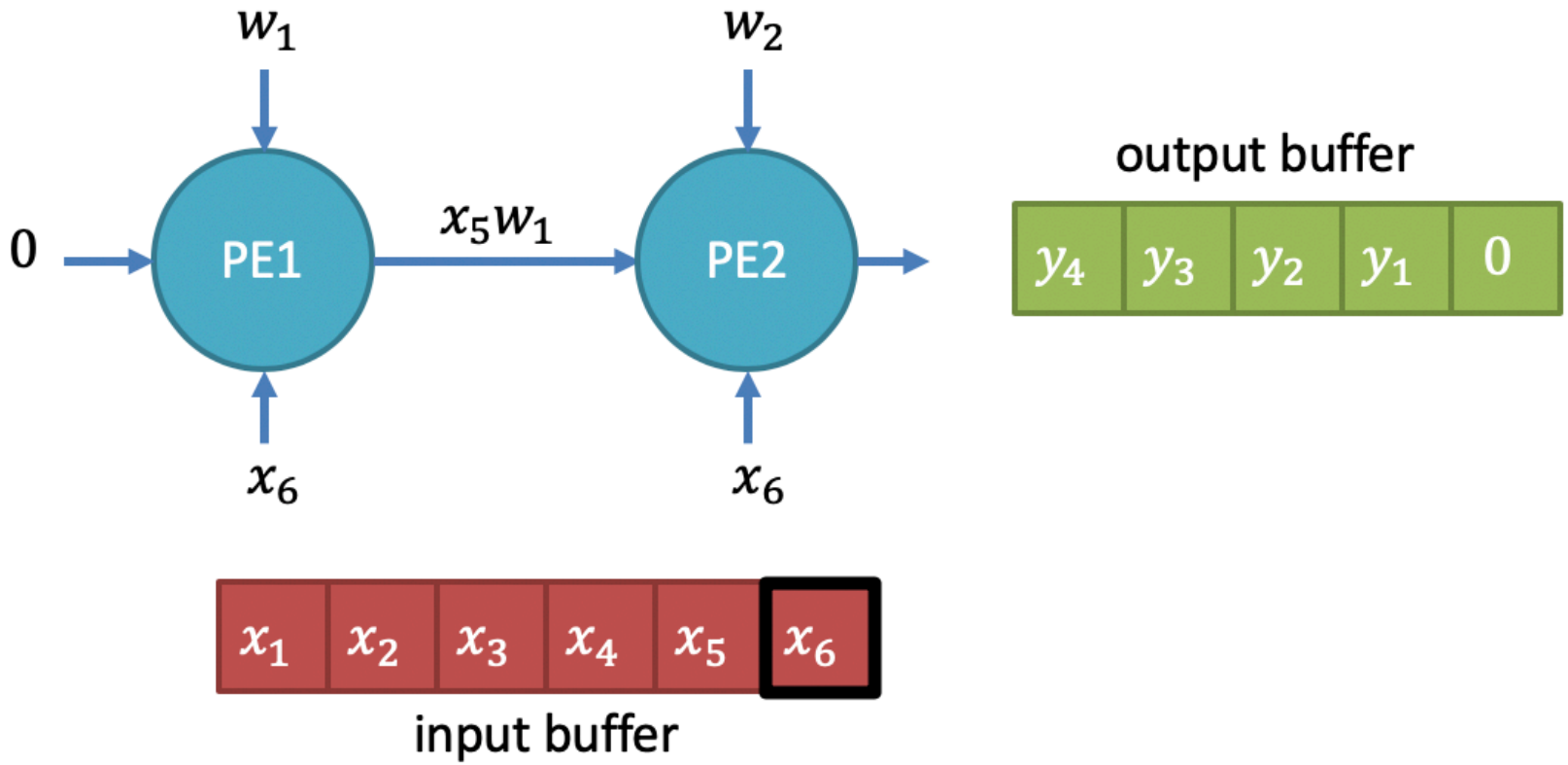

- $x_4$ is broadcasted to the two PEs
- PE1 performs $z = x_4 \times w_1 + 0$
- PE2 performs $z = x_4 \times w_2 + x_3 \times w_1$

# Weight Stationary – cycle 5



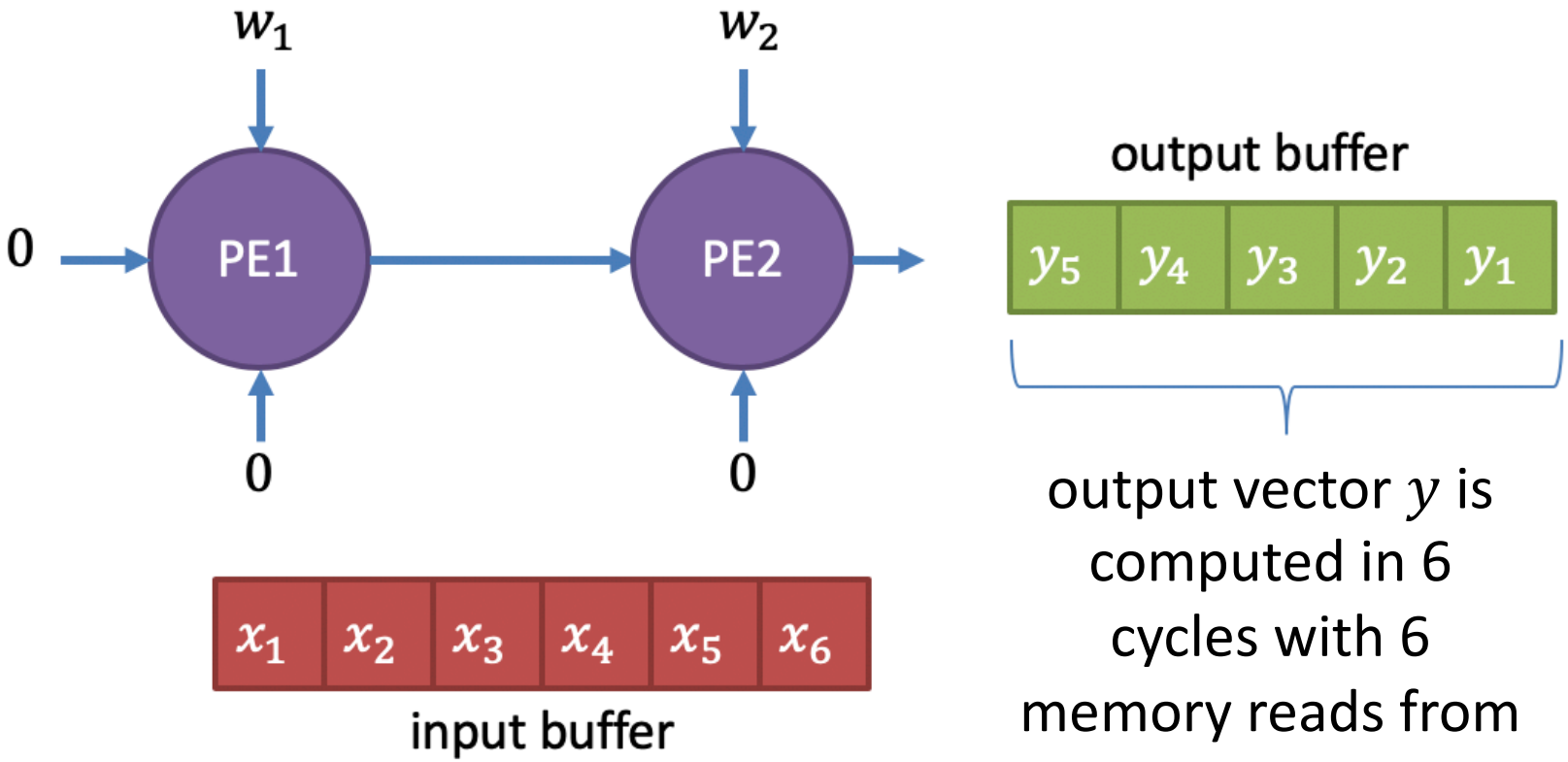

- $x_5$ is broadcasted to the two PEs
- PE1 performs $z = x_5 \times w_1 + 0$
- PE2 performs $z = x_5 \times w_2 + x_4 \times w_1$

# Weight Stationary – cycle 5




- $x_6$ is broadcasted to the two PEs
- PE1 performs $z = x_6 \times w_1 + 0$
- PE2 performs $z = x_6 \times w_2 + x_5 \times w_1$

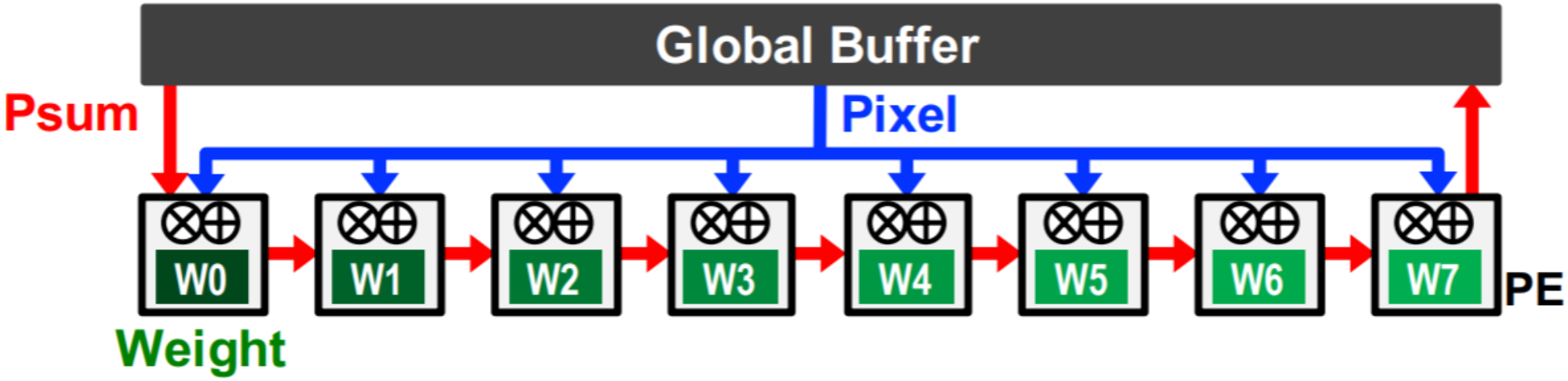
# Weight Stationary – cycle 6




output vector $y$ is computed in 6 cycles with 6 memory reads from the input buffer

- $y_5$ is collected from PE2
- PE1 remains idle
- PE2 remains idle

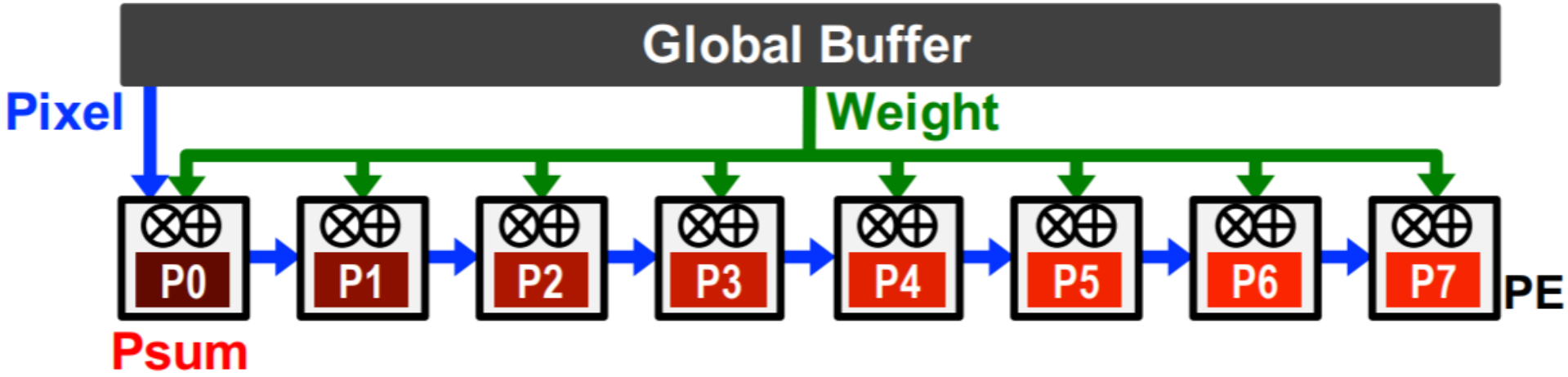
# Weight Stationary (WS)




[Eyeriss-ISCA'17]

- minimize weight read energy consumption by maximizing convolutional and weight reuse
- no need to write-back weights during inference

# Output Stationary (OS)



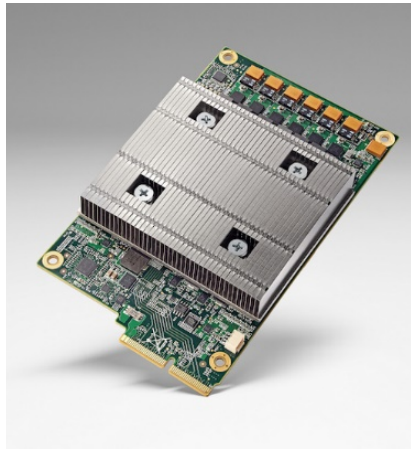

[Eyeriss-ISCA'17]

- minimize partial sum read/write energy consumption by maximizing local accumulation of partial sums
- final sum (activations) may need to be written back

# Tensor Processing Unit (TPU)

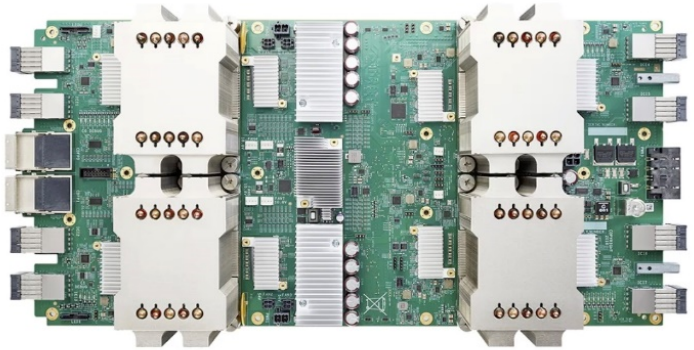
# Google's Cloud TPU Family

## TPU V1



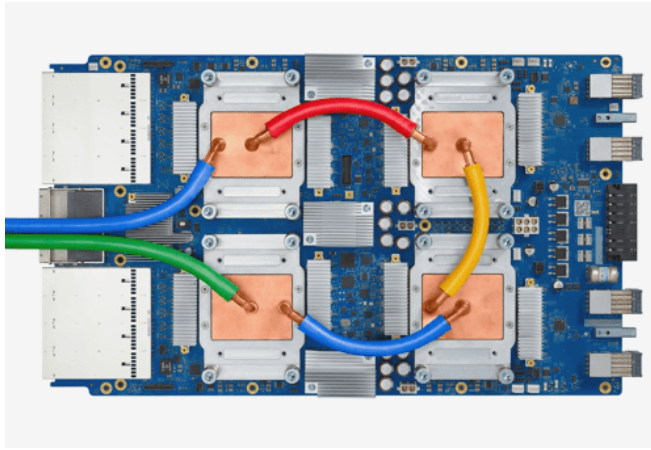
inference only
8b fixed-point

## TPU V2



inference & training
fixed-point & floating-point

## TPU V3



inference & training
fixed-point & floating-point

- TPUs are application-specific integrated circuits (ASICs) developed by Google for accelerating machine learning workloads in the cloud
- How does it work?

# TPU V1

## In-Datacenter Performance Analysis of a Tensor Processing Unit™

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon

*Google, Inc., Mountain View, CA USA*

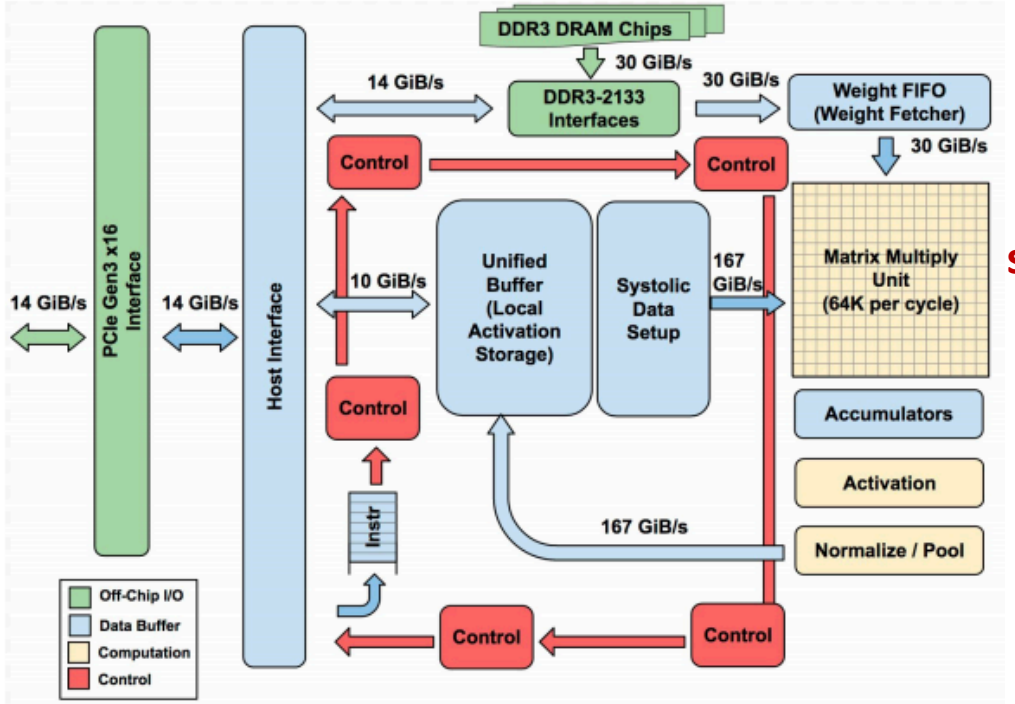
Email: {jouppi, cliffy, nishantpatil, davidpatterson}@google.com

*To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26, 2017.*

- First TPU was published in ISCA'17!
- Designed to accelerate NN workloads (**MLPs**, **CNNs**, and **LSTMs**) in Google's datacenters
- Performs **inference** only, using 8b **fixed-point** operations
- Uses a matrix multiply **systolic architecture**

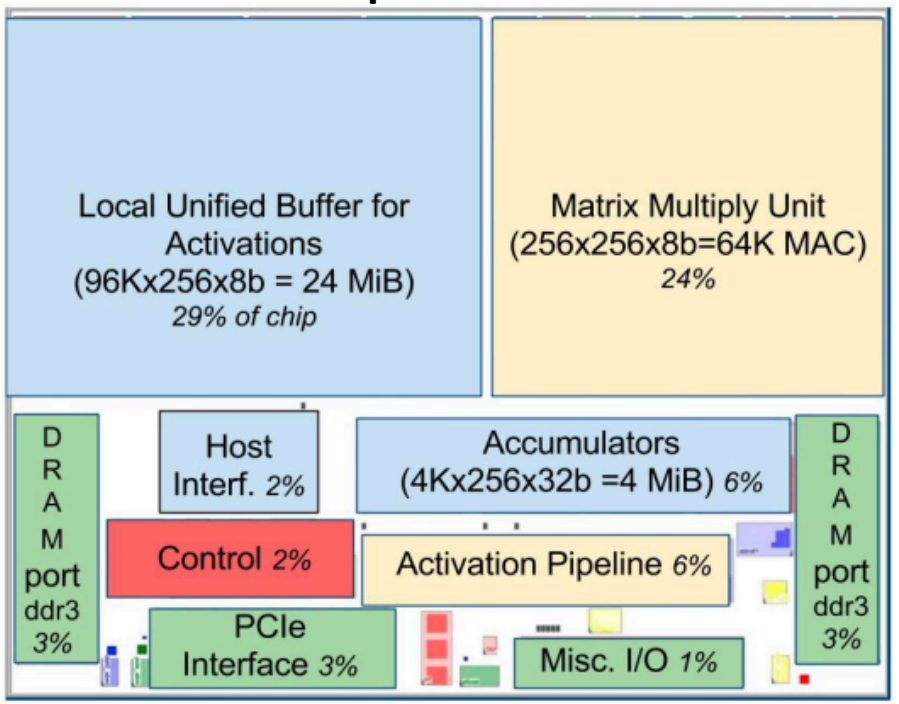
# TPU Architecture

### Block Diagram




systolic array

### Chip Floor Plan

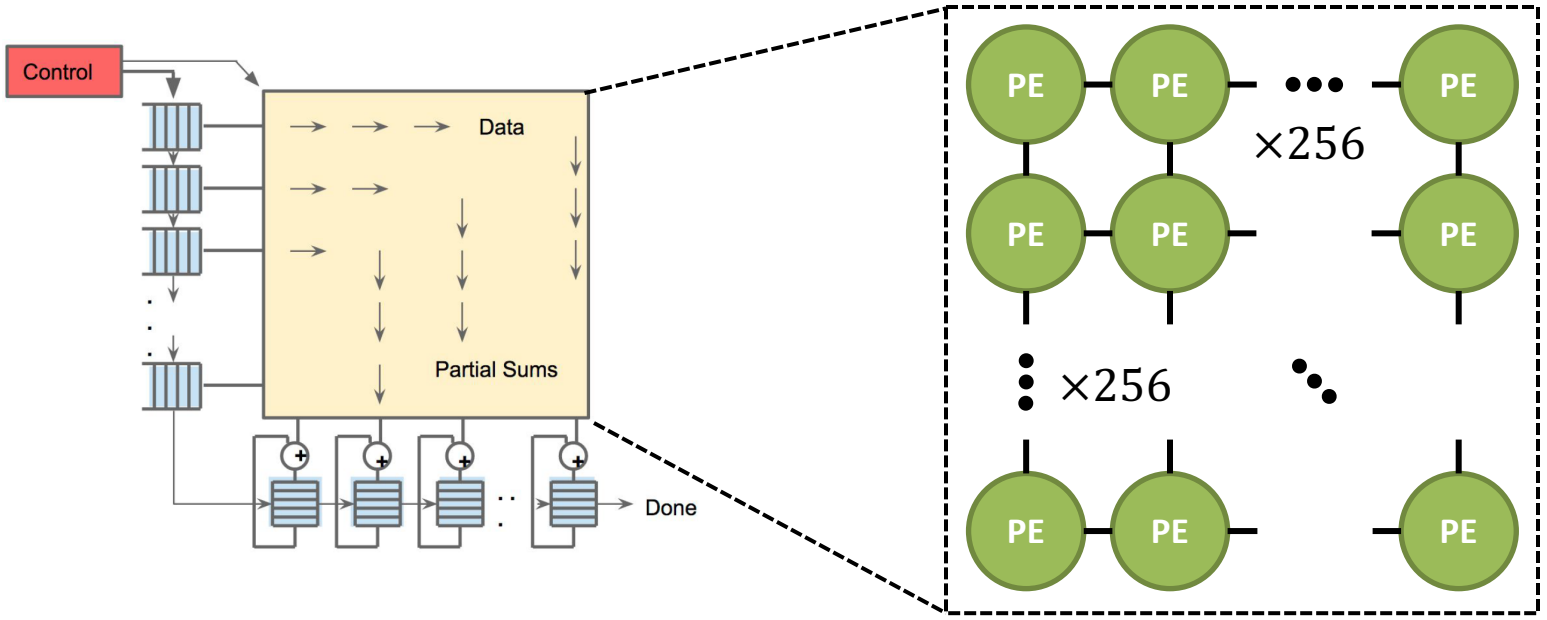



- used as a co-processor on PCIe I/O bus much like a GPU
- uses a CISC-like instruction set sent via host server
- dedicated weight FIFO for fast weight loading (30 GiB/s)
- Unified buffer stores local input/output activations

- LUB and MMU → ~2/3 of die area
- 24MiB chosen to pitch-match to MMU
- control is 2% of die area

[TPU-ISCA'17]

# Matrix Multiply Unit (MMU)




- systolic architecture: 256×256 PE array, each PE can perform 8b MAC
- inputs are streamed in from left to right;
- partial sums are accumulated downwards and are collected in 32-bit wide accumulators below the matrix unit (4MiB = 4096 x 256 x 32 bits)
- weights are pre-fetched and remain local to the PE => TPU is **weight stationary**
- produces one 256-element partial sum per clock cycle.

# MMU Systolic Architecture - Explained

- Matrix multiplication needs to occur between a batched input matrix $X$ $(B\times256)$ and a weight matrix $W$ $(256\times256)$ to produce a batched output matrix $Y$ $(B\times256)$ :

$$Y = X\times W$$

- Operation takes $B$ pipelined cycles to finish
- Weights are first fetched from DRAM (8GB) via a weight FIFO and loaded into the PE array
- Inputs (outputs) are loaded (stored) from (to) the 24MB unified buffer

# Example of MMU Computation

- Consider the simplified example for 3×3 weight matrix:

$$\begin{bmatrix} y_{x0} & y_{x1} & y_{x2} \\ y_{u0} & y_{u1} & y_{u2} \\ y_{z0} & y_{z1} & y_{z2} \\ y_{v0} & y_{v1} & y_{v2} \\ y_{p0} & y_{p1} & y_{p2} \\ y_{q0} & y_{q1} & y_{q2} \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & x_2 \\ u_0 & u_1 & u_2 \\ z_0 & z_1 & z_2 \\ v_0 & v_1 & v_2 \\ p_0 & p_1 & p_2 \\ q_0 & q_1 & q_2 \\ \vdots & \vdots & \vdots \end{bmatrix} \times \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \\ w_{20} & w_{21} & w_{22} \end{bmatrix}$$

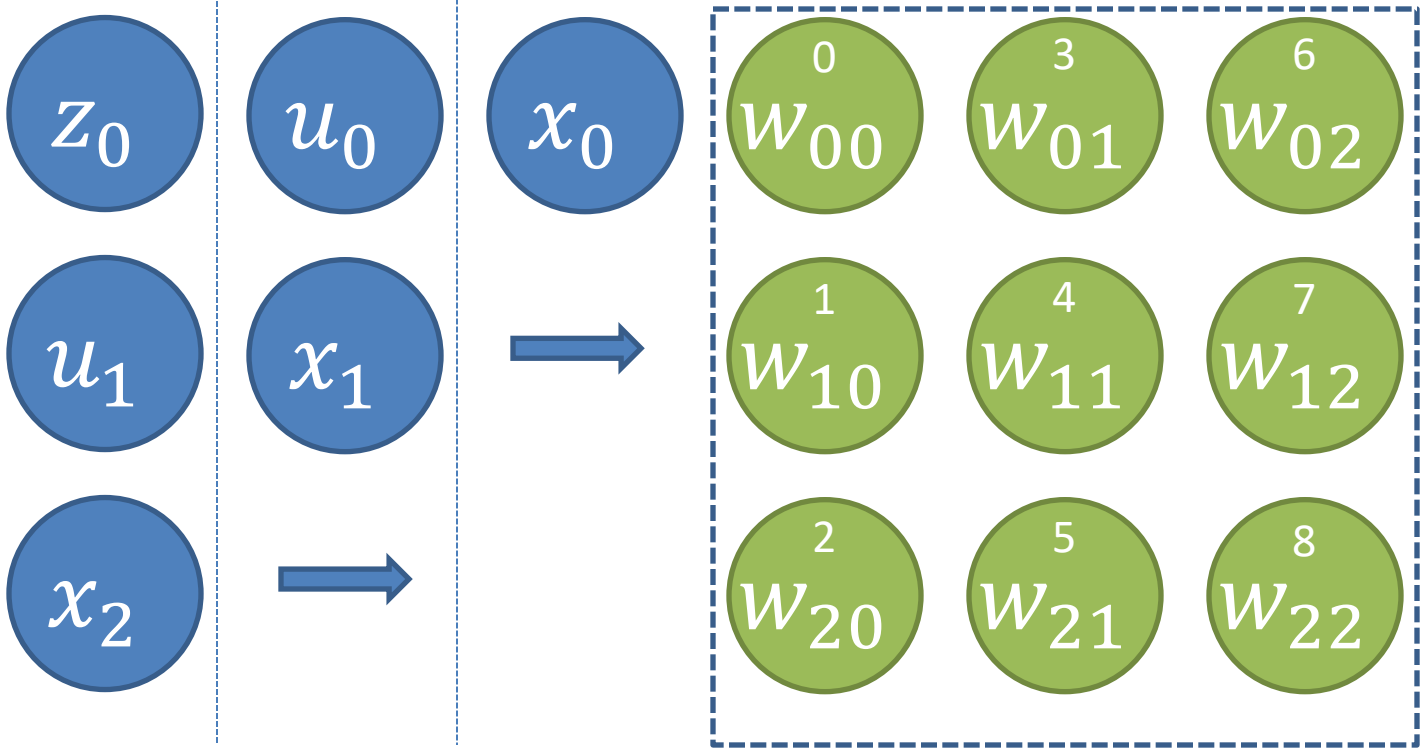
$$Y \qquad\qquad X \qquad\qquad W$$

- How would the MMU process this?

0: idle
1: idle
2: idle
3: idle
4: idle
5: idle
6: idle
7: idle
8: idle

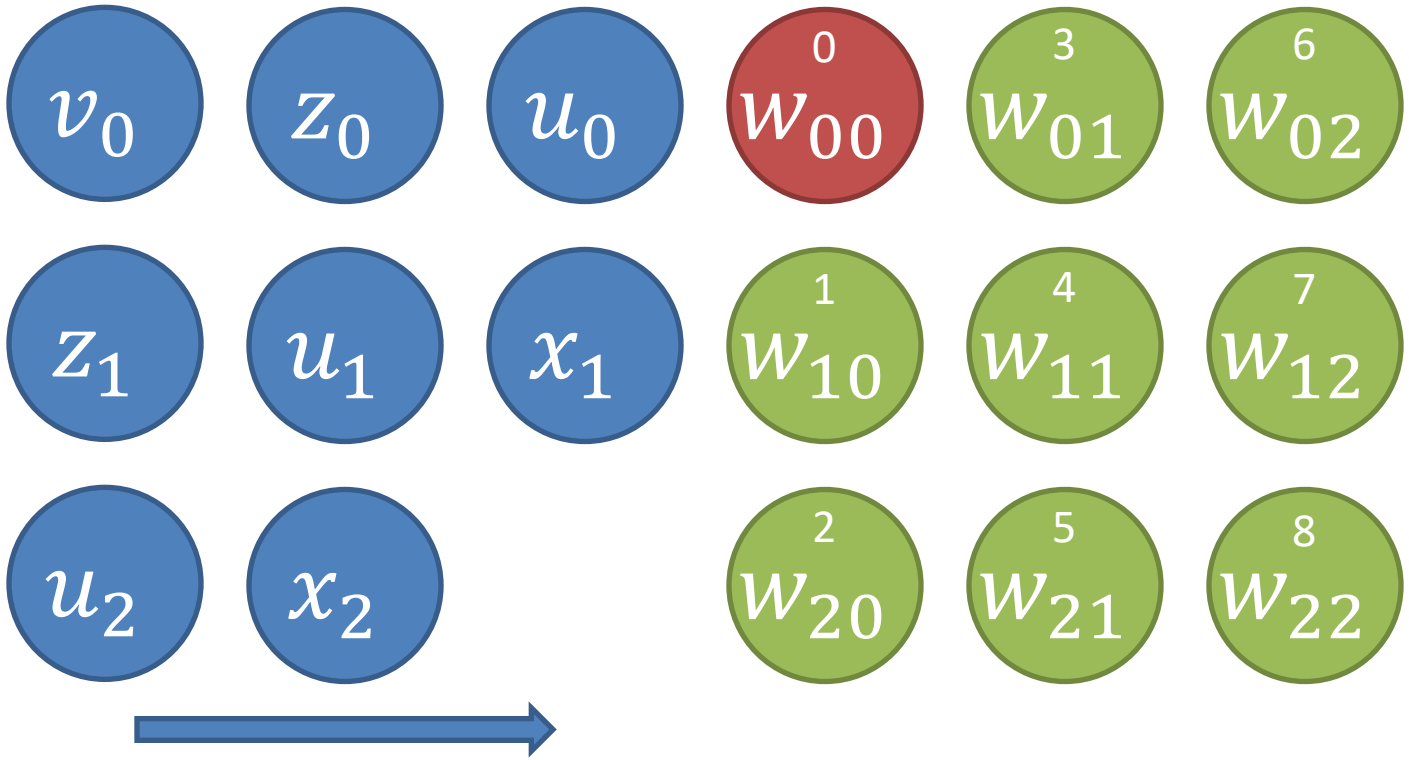
Status of each PE at the current cycle

Inputs are loaded and streamed cycle by cycle from the unified buffer

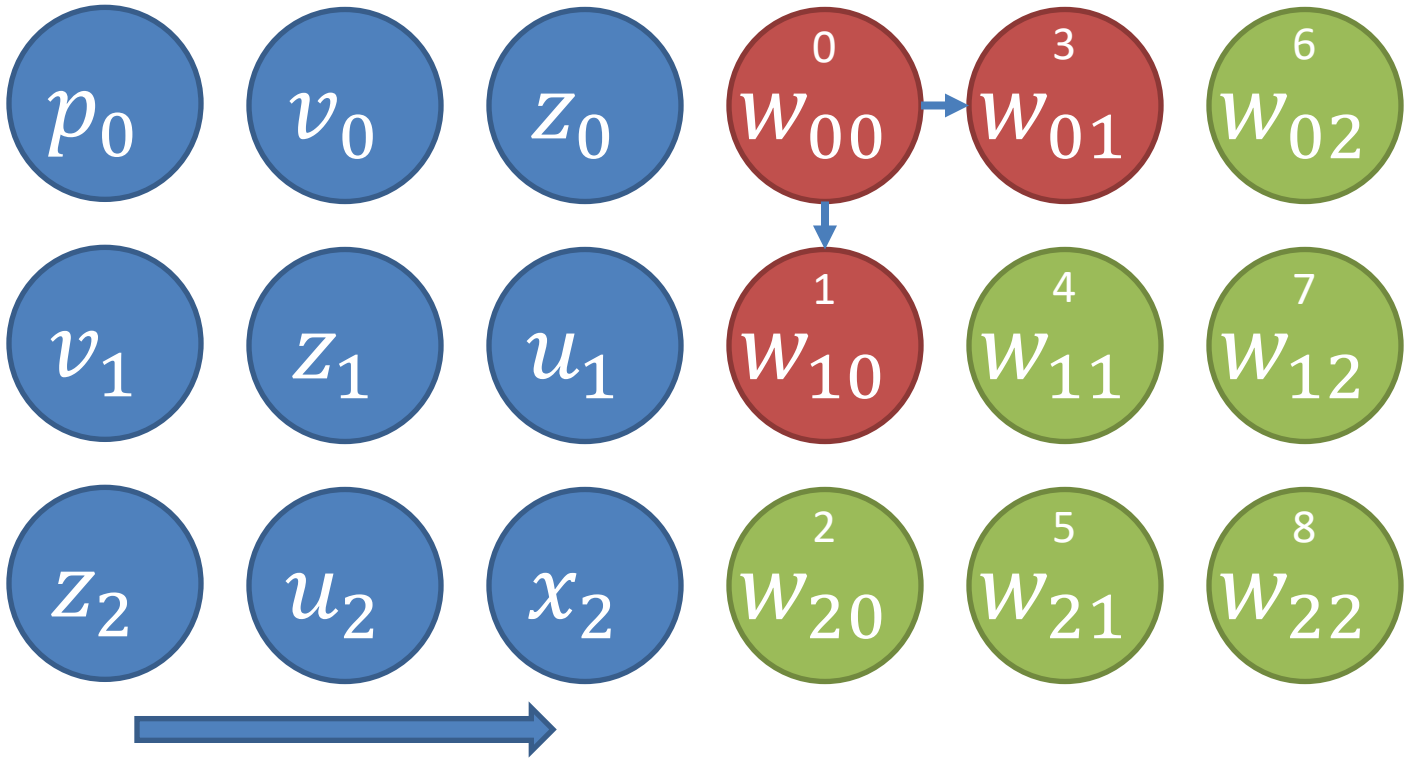



3×3 PE array with weights pre fetched via dedicated weights FIFO

0: computing $w_{00}x_0$
1: idle
2: idle
3: idle
4: idle
5: idle
6: idle
7: idle
8: idle

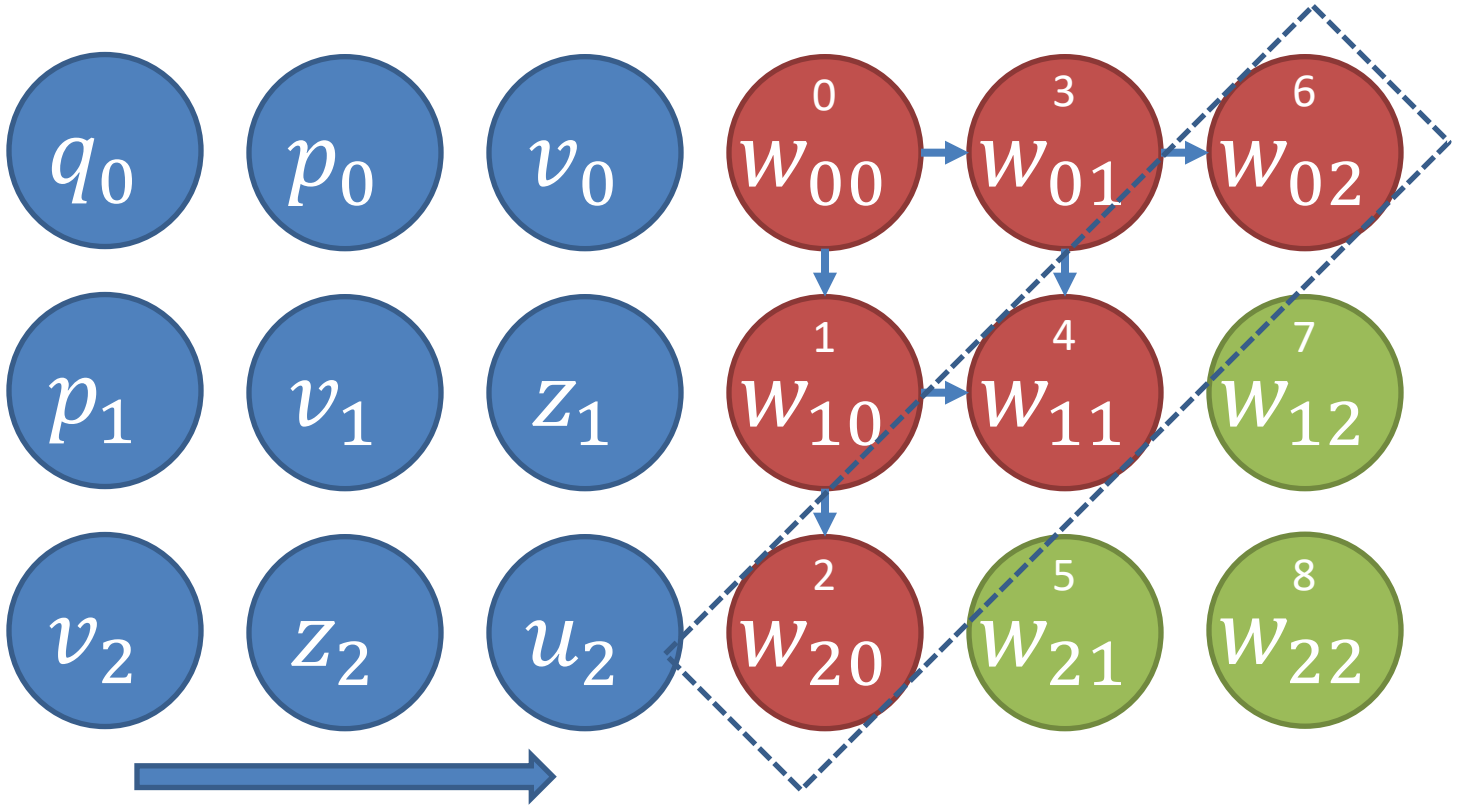


0: computing $w_{00}u_0$
1: computing $w_{00}x_0 + w_{10}x_1$
2: idle
3: computing $w_{01}x_0$
4: idle
5: idle
6: idle
7: idle
8: idle



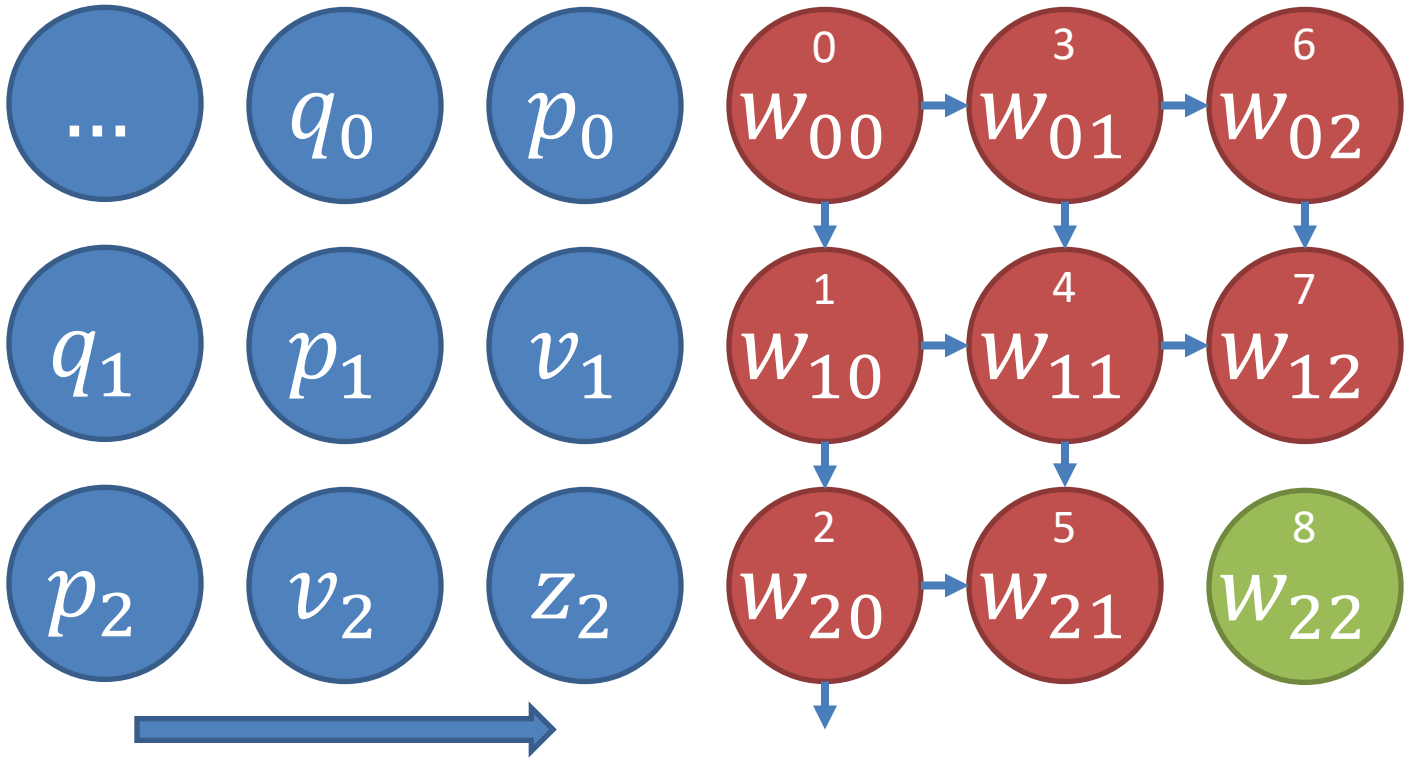
0: computing $w_{00}z_0$
1: computing $w_{00}u_0 + w_{10}u_1$
2: computing $w_{00}x_0 + w_{10}x_1 + w_{20}x_2$
3: computing $w_{01}u_0$
4: computing $w_{01}x_0 + w_{11}x_1$
5: idle
6: computing $w_{02}x_0$
7: idle
8: idle

First DP is done, will be flushed out next cycle



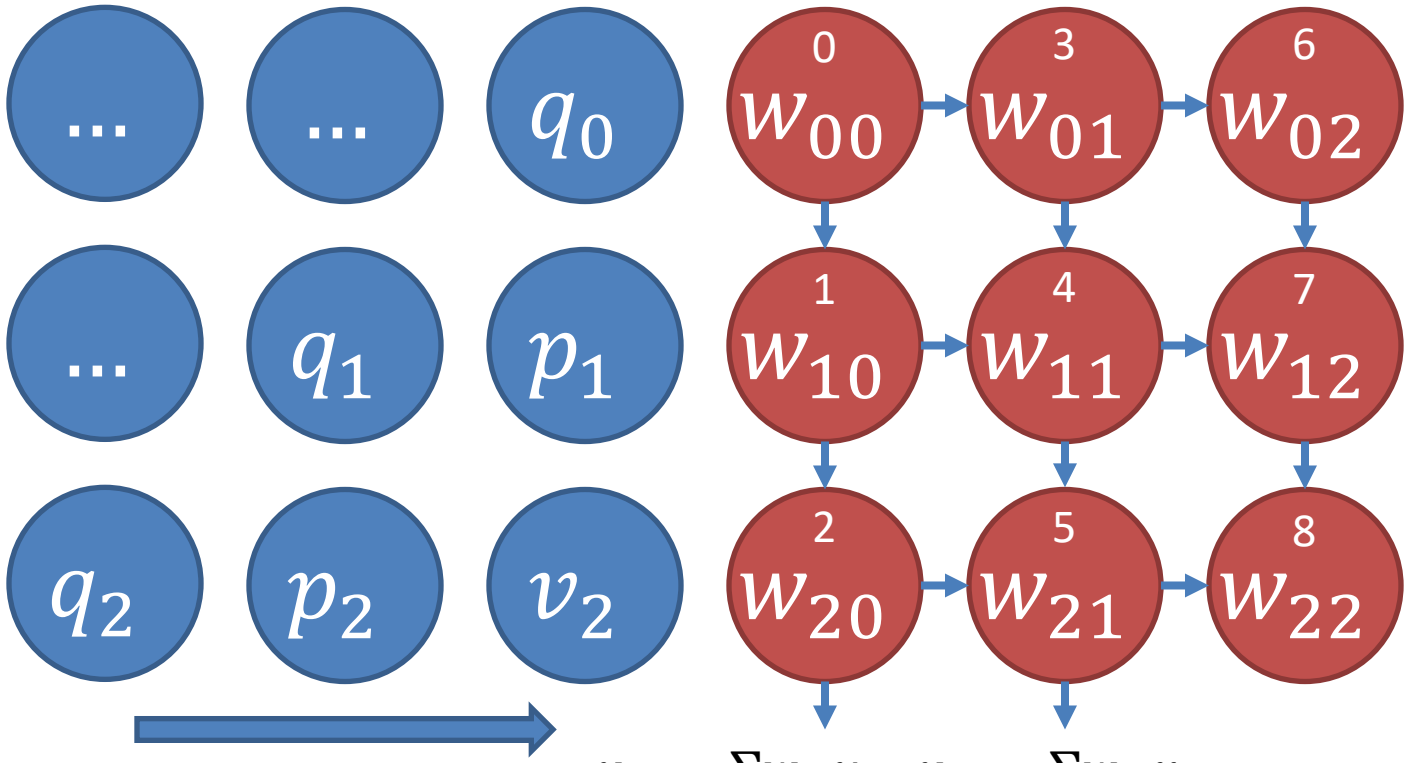

Each input vector is diagonally propagating

0: computing $w_{00}v_0$
1: computing $w_{00}z_0 + w_{10}z_1$
2: computing $w_{00}u_0 + w_{10}u_1 + w_{20}u_2$
3: computing $w_{01}z_0$
4: computing $w_{01}u_0 + w_{11}u_1$
5: computing $w_{01}x_0 + w_{11}x_1 + w_{21}x_2$
6: computing $w_{02}u_0$
7: computing $w_{02}x_0 + w_{12}x_1$
8: idle

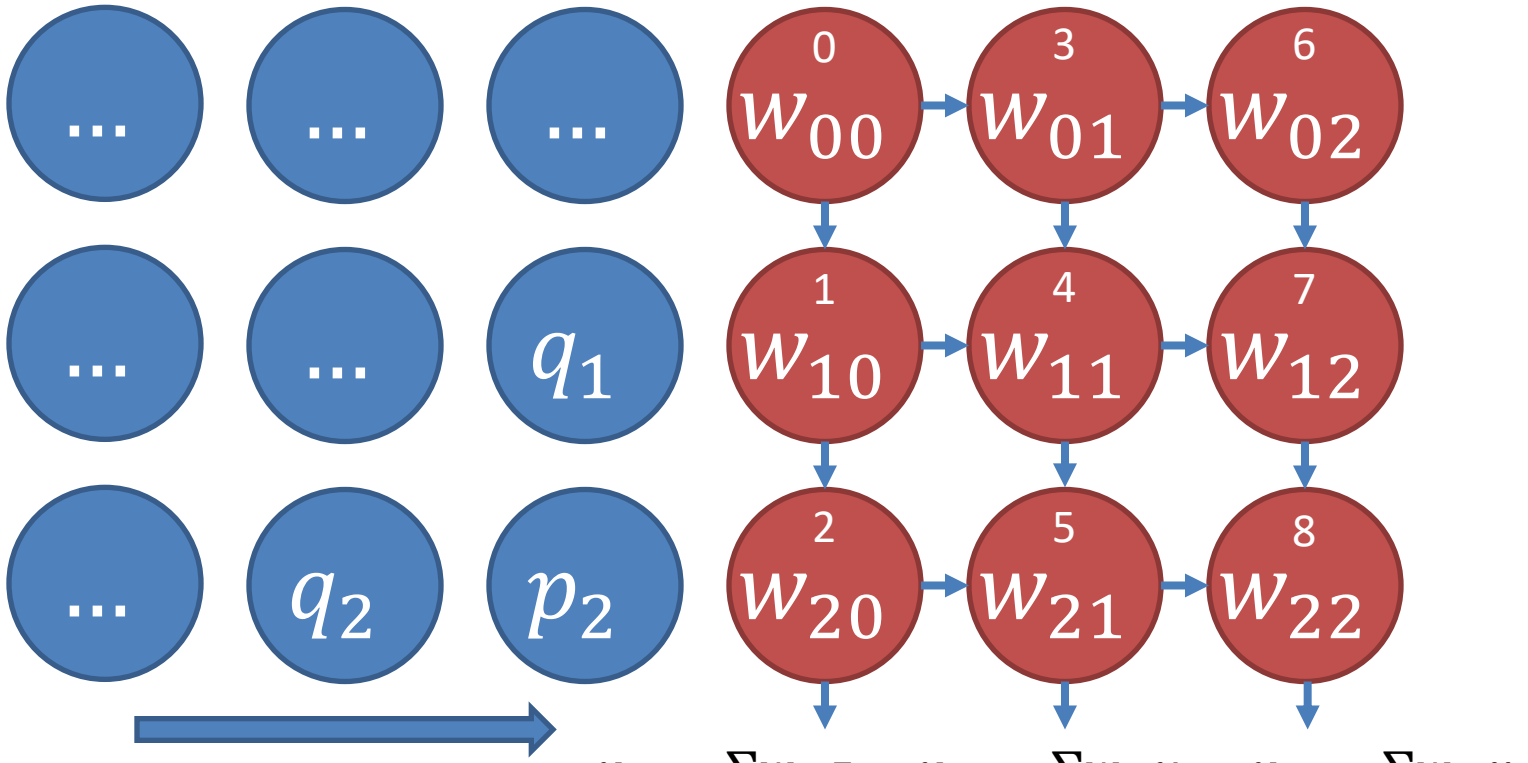



$y_{x0} = \sum w_{0i}x_i$ stored in accumulator

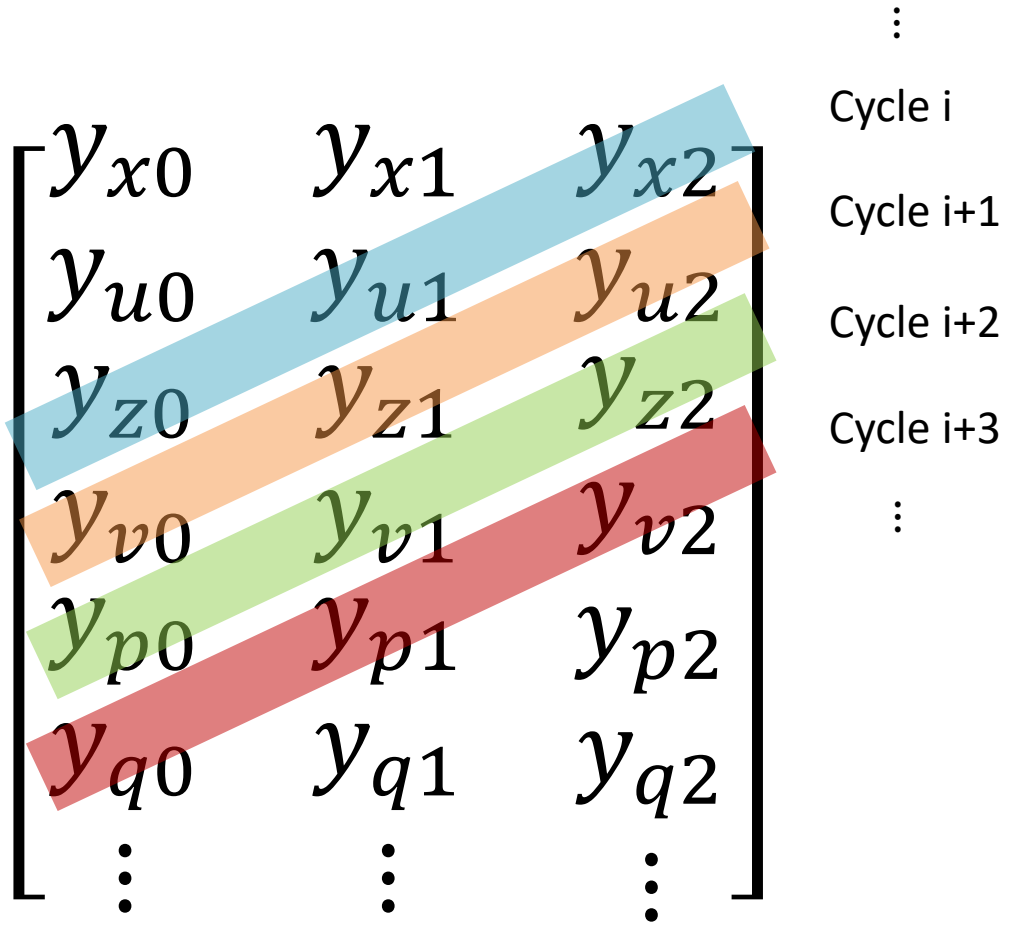
0: computing $w_{00}p_0$
1: computing $w_{00}v_0 + w_{10}v_1$
2: computing $w_{00}z_0 + w_{10}z_1 + w_{20}z_2$
3: computing $w_{01}v_0$
4: computing $w_{01}z_0 + w_{11}z_1$
5: computing $w_{01}u_0 + w_{11}u_1 + w_{21}u_2$
6: computing $w_{02}z_0$
7: computing $w_{02}u_0 + w_{12}u_1$
8: computing $w_{02}x_0 + w_{12}x_1 + w_{22}x_2$



0: computing $w_{00}q_0$
1: computing $w_{00}p_0 + w_{10}p_1$
2: computing $w_{00}v_0 + w_{10}v_1 + w_{20}v_2$
3: computing $w_{01}p_0$
4: computing $w_{01}v_0 + w_{11}v_1$
5: computing $w_{01}z_0 + w_{11}z_1 + w_{21}z_2$
6: computing $w_{02}v_0$
7: computing $w_{02}z_0 + w_{12}z_1$
8: computing $w_{02}u_0 + w_{12}u_1 + w_{22}u_2$



# At a Higher Level




After every cycle, an output diagonal will be computed

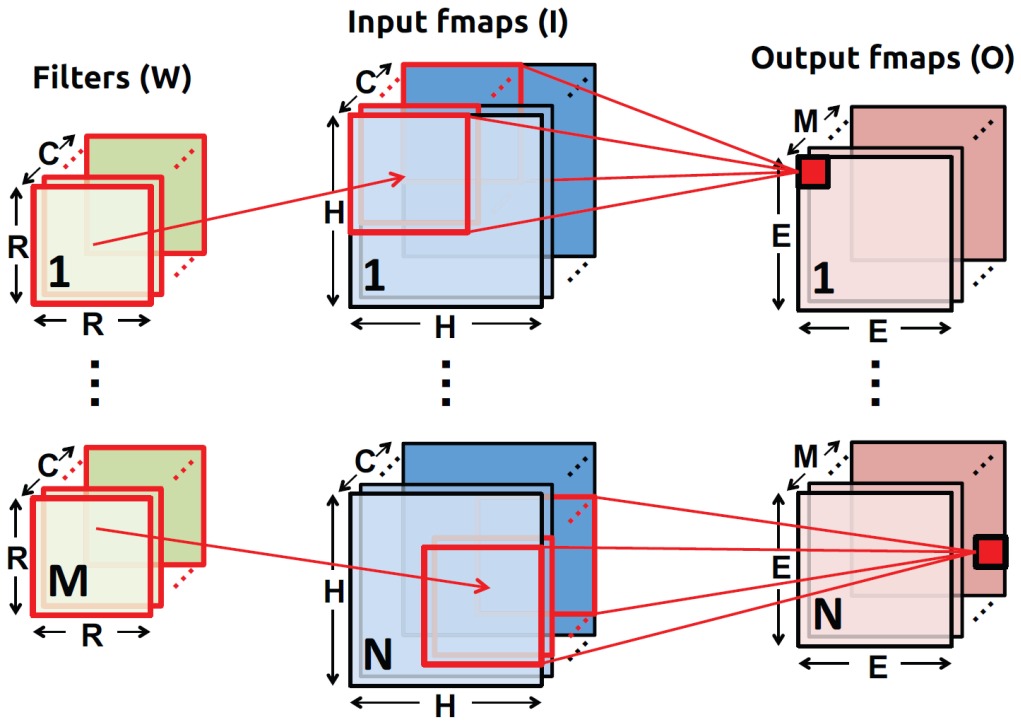
# Mapping of Neural Networks

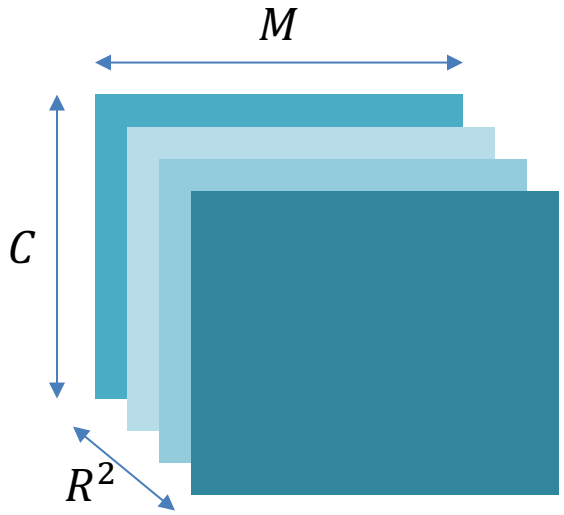
| Name | LOC | Layers | | | | | Nonlinear function | Weights | TPU Ops / Weight Byte | TPU Batch Size | % of Deployed TPUs in July 2016 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | FC | Conv | Vector | Pool | Total | | | | | |
| MLP0 | 100 | 5 | | | | 5 | ReLU | 20M | 200 | 200 | 61% |
| 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 | |

**Table 1.** Six NN applications (two per NN type) that represent 95% of the TPU's workload. The columns are the NN name; the number of lines of code; the types and number of layers in the NN (FC is fully connected, Conv is convolution, Vector is self-explanatory, Pool is pooling, which does nonlinear downsizing on the TPU; and TPU application popularity in July 2016. One DNN is RankBrain [Cla15]; one LSTM is a subset of GNM Translate [Wu16]; one CNN is Inception; and the other CNN is DeepMind AlphaGo [Sil16][Jou15].

- Mapping MLPs and LSTMs is straight forward
  - matrix multiplications are the core operations in such network architectures; Separate hardware for activation/pooling functions
- What about CNNs?
  - TPU is not optimized for CNNs (unlike Eyeriss, ShiDianNao)
  - reason is that TPU was meant for Google's datacenters, where CNN workloads don't occur as often as on edge devices (5% according to Google, at the time of publishing)

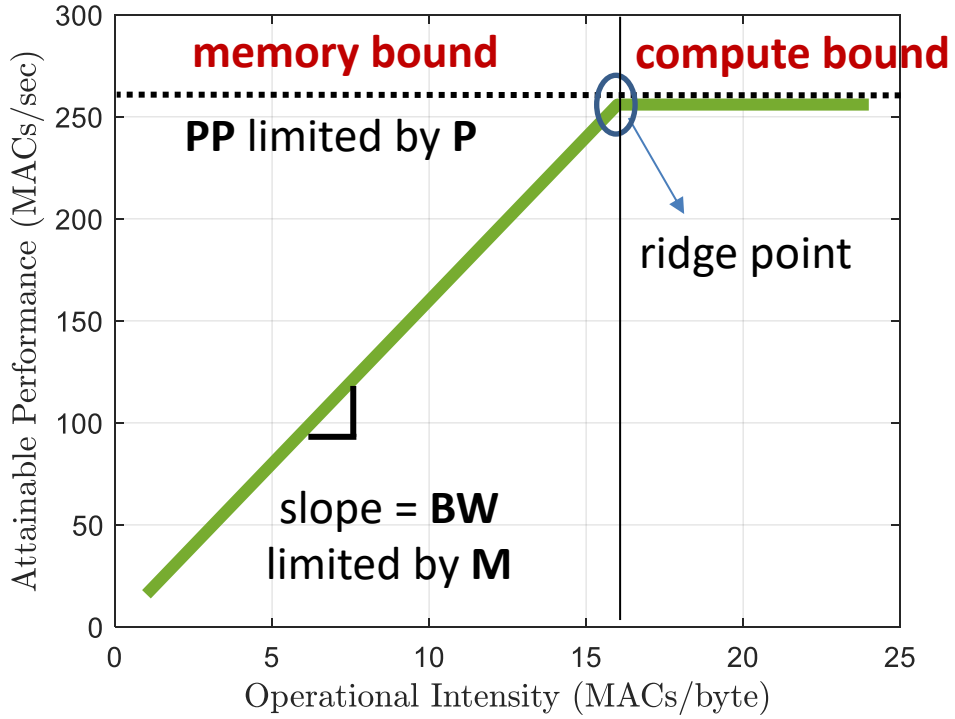
# Mapping of Convolutional Layers




- Restructure weights to have $R^2$ weight matrices, each of shape $C \times M$
- Restructure inputs into a matrix of shape $H^2N \times C$ by stacking input channels
- MM needs to be repeated $R^2$ times, result gets aggregated across $R^2$ dimension

# TPU Performance: Roofline Plots - Recap




$$\min(\text{PP}, \text{OI}\times\text{BW})$$

peak performance — operational intensity — bandwidth

# TPU Performance: Roofline Plots

- TPU runs at 700MHz clock frequency

$$\text{peak performance} = 256^2 \times 2 \times 700{,}000{,}000$$

# of MACs/cycle | 1 MAC = 2 OPs | cycles/sec

=> $\text{PP} = 92 \times 10^{12}$(OPs/sec) or 92 TeraOPs/sec

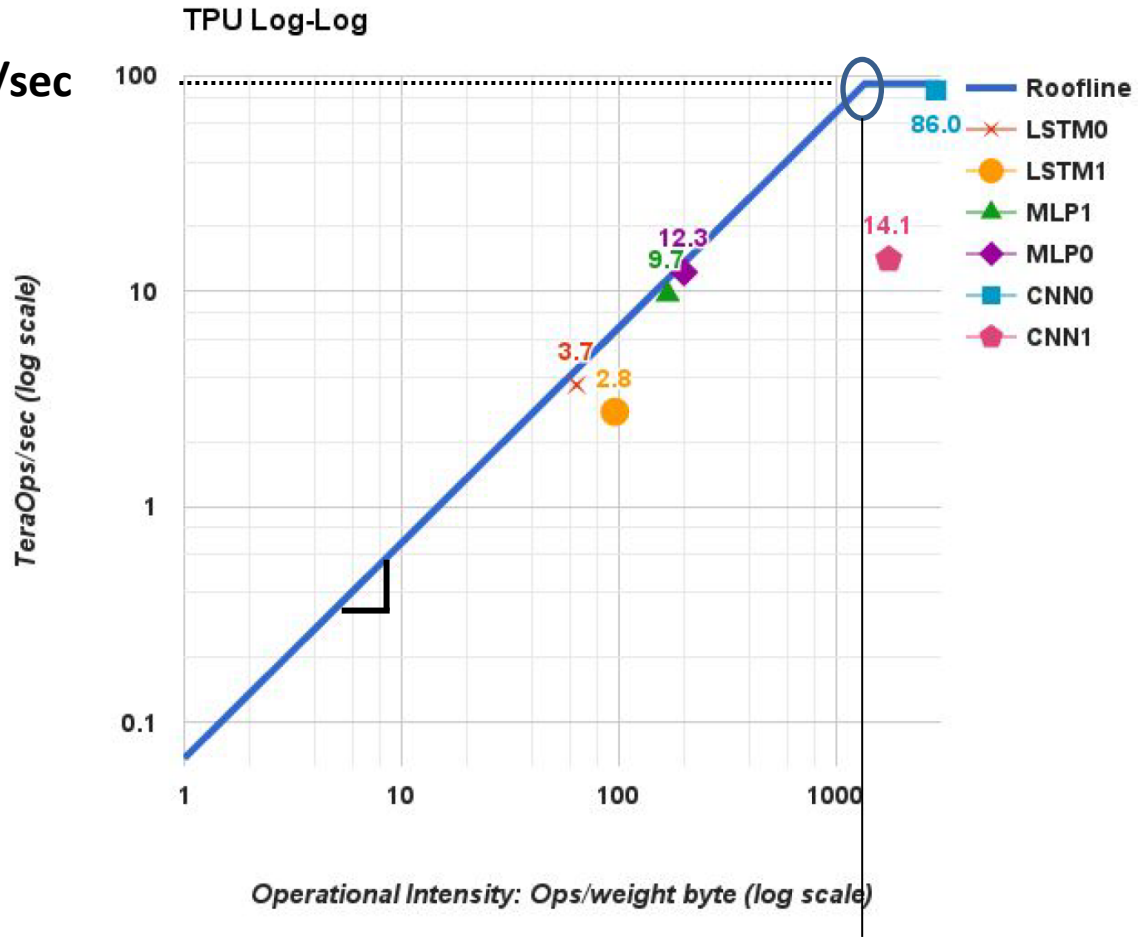
- Memory bandwidth is 34 GiB/sec (from DRAM)

# TPU Performance: Roofline Plots

- Roofline of TPU on 6 benchmark networks
  - 2 MLPs
  - 2 LSTMs
  - 2 CNNs

- Due to log-log plot, slope in memory bound regime will always **appear** as 1!
  - $\log(AP) = \log(OI) + \log(BW)$
  - How much is BW?
- Ridge point is at 1350 Ops/byte

**92 TOPs/sec**



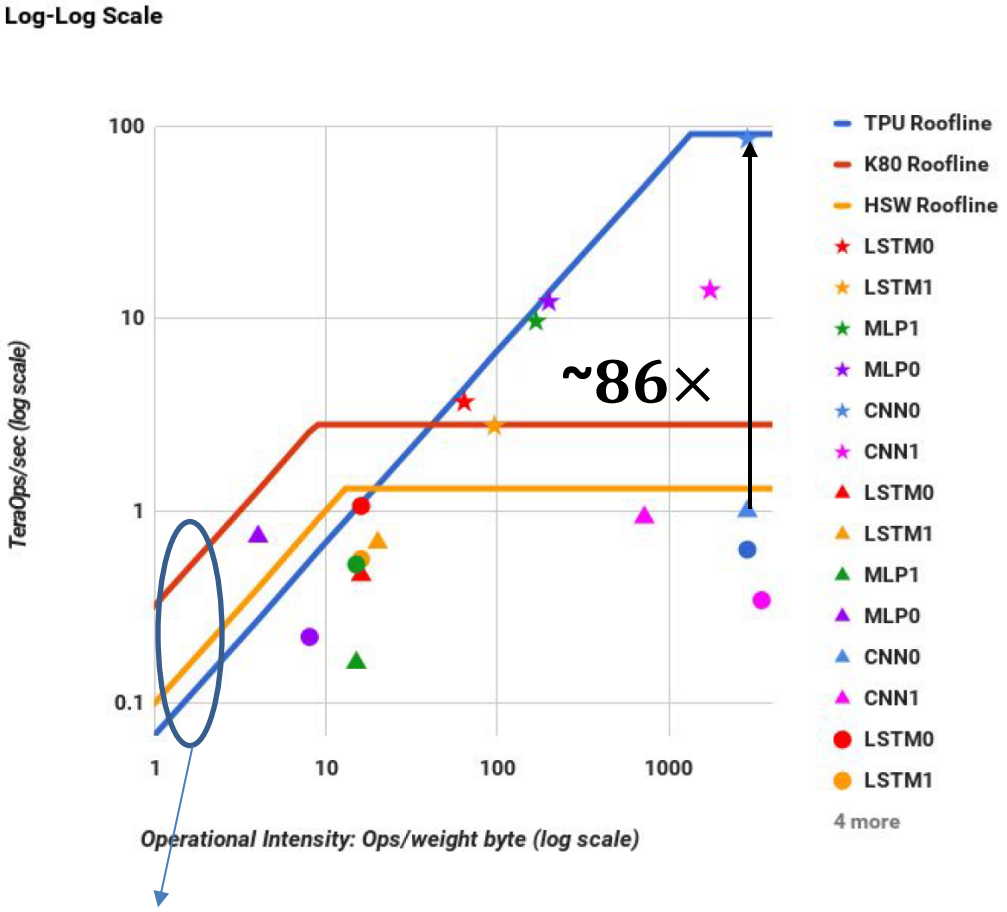

**ridge point at 1350 OPs/byte**

# TPU Performance Comparison

| Model | Die | | | | | | | | | | Benchmarked Servers | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | mm² | nm | MHz | TDP | Measured Idle | Measured Busy | TOPS/s 8b | TOPS/s FP | GB/s | On-Chip Memory | Dies | DRAM Size | TDP | Measured Idle | Measured 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 | NA* | 28 | 700 | 75W | 28W | 40W | 92 | -- | 34 | 28 MiB | 4 | 256 GiB (host) + 8 GiB x 4 | 861W | 290W | 384W |

**Table 2.** Benchmarked servers use Haswell CPUs, K80 GPUs, and TPUs. Haswell has 18 cores, and the K80 has 13 SMX processors. Figure 10 has measured power. The low-power TPU allows for better rack-level density than the high-power GPU. The 8 GiB DRAM per TPU is Weight Memory. GPU Boost mode is not used (Sec. 8). SECDEC and no Boost mode reduce K80 bandwidth from 240 to 160. No Boost mode and single die vs. dual die performance reduces K80 peak TOPS from 8.7 to 2.8. (*The TPU die is ≤ half the Haswell die size.)

- Benchmarked against:
  - Intel Haswell E5-2699 v3 (CPU)
  - NVIDIA K80 (GPU)
- All TPU stars are at or above the other 2 rooflines
- CNN0 sees almost ~86× performance improvement compared to K80 (*compute bound*)
- Why does CNN1 fall behind the PP limit?




**BW difference comes as an offset due to log-log plot**

# TPU Performance: CNN1

[TPU-ISCA'17]

| *Application* | *MLP0* | *MLP1* | *LSTM0* | *LSTM1* | *CNN0* | *CNN1* | *Mean* | *Row* |
|---|---|---|---|---|---|---|---|---|
| Array active cycles | 12.7% | 10.6% | 8.2% | 10.5% | 78.2% | 46.2% | 28% | 1 |
| Useful MACs in 64K matrix (% peak) | 12.5% | 9.4% | 8.2% | 6.3% | 78.2% | 22.5% | 23% | 2 |
| Unused MACs | 0.3% | 1.2% | 0.0% | 4.2% | 0.0% | 23.7% | 5% | 3 |
| Weight stall cycles | 53.9% | 44.2% | 58.1% | 62.1% | 0.0% | 28.1% | 43% | 4 |
| Weight shift cycles | 15.9% | 13.4% | 15.8% | 17.1% | 0.0% | 7.0% | 12% | 5 |
| Non-matrix cycles | 17.5% | 31.9% | 17.9% | 10.3% | 21.8% | 18.7% | 20% | 6 |
| RAW stalls | 3.3% | 8.4% | 14.6% | 10.6% | 3.5% | 22.8% | 11% | 7 |
| Input data stalls | 6.1% | 8.8% | 5.1% | 2.4% | 3.4% | 0.6% | 4% | 8 |
| TeraOps/sec (92 Peak) | 12.3 | 9.7 | 3.7 | 2.8 | 86.0 | 14.1 | 21.4 | 9 |

- The TPU spends less than half of its cycles performing matrix operations for CNN1
- On each of those active cycles, only about half of the $256^2$ PEs hold useful weights because some layers in CNN1 have small channel sizes
- About 35% of cycles are spent waiting for weights to load from memory into the matrix unit

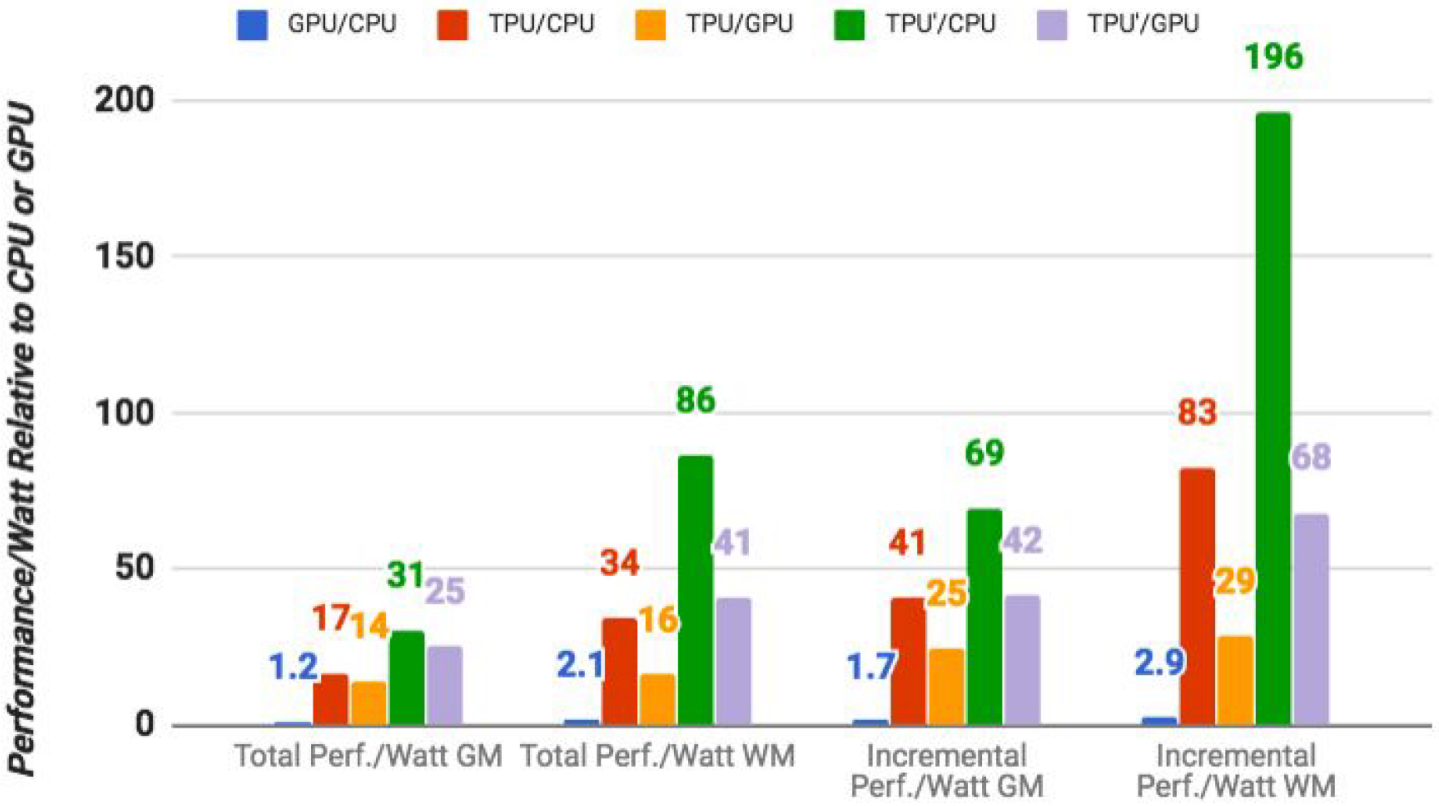
# Latency vs Throughput – MLP0

| *Type* | *Batch* | *99th% Response* | *Inf/s (IPS)* | *% Max IPS* |
|---|---|---|---|---|
| CPU | 16 | 7.2 ms | 5,482 | 42% |
| CPU | 64 | 21.3 ms | 13,194 | 100% |
| GPU | 16 | 6.7 ms | 13,461 | 37% |
| GPU | 64 | 8.3 ms | 36,465 | 100% |
| TPU | 200 | 7.0 ms | 225,000 | 80% |
| TPU | 250 | 10.0 ms | 280,000 | 100% |

- Higher batch size -> more throughput (IPS)
- But also more delayed response time
- While realizing the 7ms latency constraint*, the TPU does not give up much of its IPS (100% => 80%)

*7ms latency constraint is determined by the application developer according to Google

# Relative Performance/Watt




- TPU' is an improved TPU that uses GDDR5 memory (same as GPU)
- Total includes host server power, but incremental doesn't
- GM and WM are the geometric and weighted means

# TPU: Summary

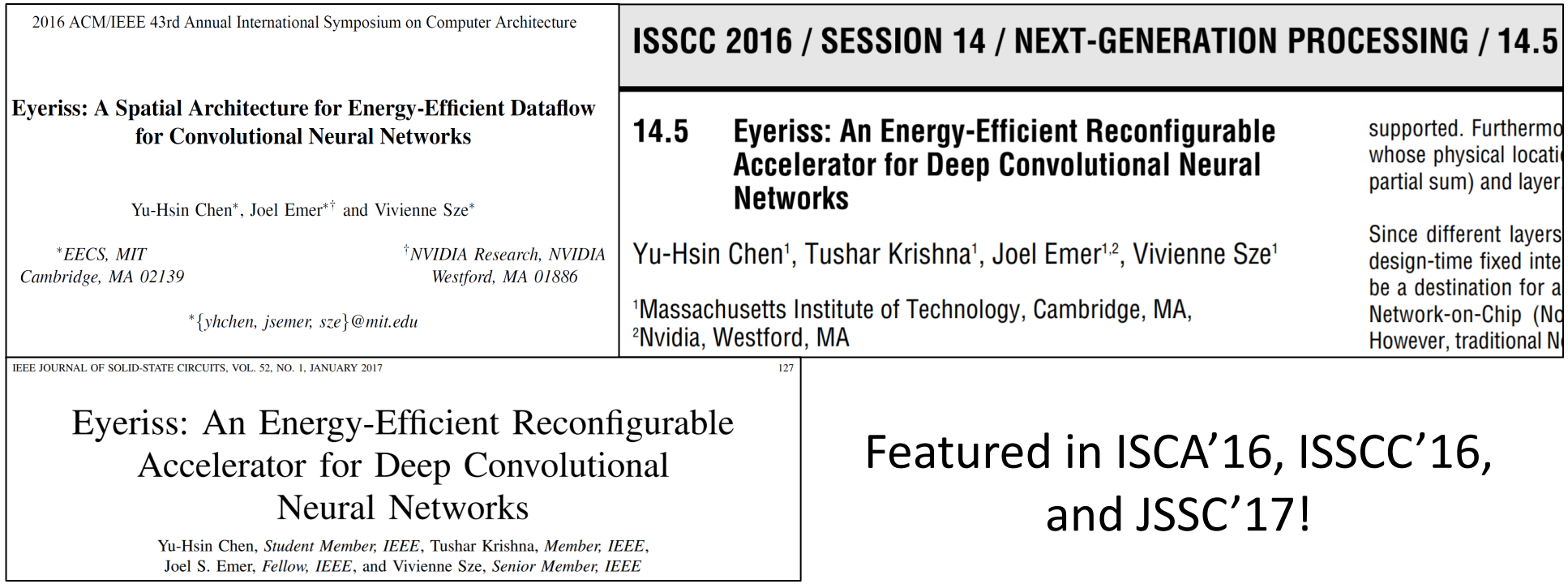
- **TPU leverages the reduction in energy and area of 8-bit integer systolic matrix multipliers over 32-bit floating-point datapaths of a K80 GPU to:**
  - pack 25 times as many MACs
  - 3.5 times the on-chip memory
  - use less than half the power of the K80 in a relatively small die.

- **TPU maps NN layers using weight stationary mapping onto a $256 \times 256$ PE array**

- **Latency in ML applications is just as important as throughput**
  - GPUs are optimized for batched operations, more suited for training than inference

- **TPU design is optimized for MLPs and LSTMS, but not CNNs**

# Eyeriss



2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

**Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks**

Yu-Hsin Chen*, Joel Emer*† and Vivienne Sze*

*EECS, MIT
Cambridge, MA 02139

†NVIDIA Research, NVIDIA
Westford, MA 01886

*{yhchen, jsemer, sze}@mit.edu

**ISSCC 2016 / SESSION 14 / NEXT-GENERATION PROCESSING / 14.5**

**14.5 Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks**

Yu-Hsin Chen[1], Tushar Krishna[1], Joel Emer[1,2], Vivienne Sze[1]

[1]Massachusetts Institute of Technology, Cambridge, MA,
[2]Nvidia, Westford, MA

supported. Furthermo
whose physical locati
partial sum) and layer

Since different layers
design-time fixed inte
be a destination for a
Network-on-Chip (Nc
However, traditional N

IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 52, NO. 1, JANUARY 2017 127

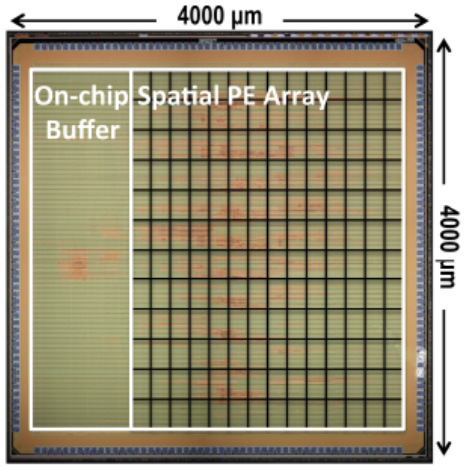
Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks

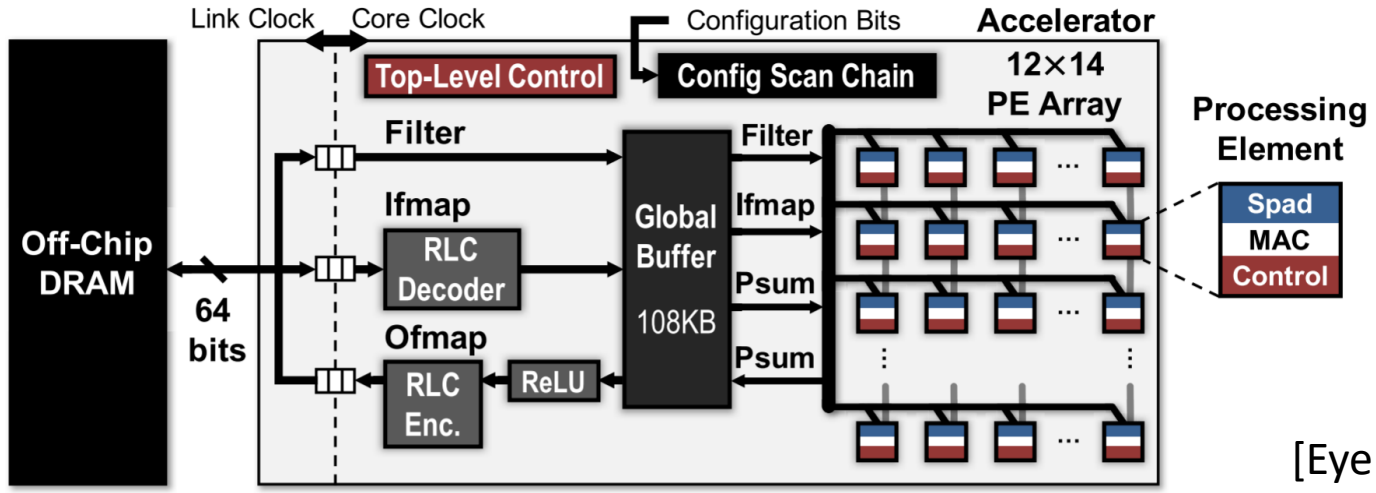
Yu-Hsin Chen, *Student Member, IEEE*, Tushar Krishna, *Member, IEEE*, Joel S. Emer, *Fellow, IEEE*, and Vivienne Sze, *Senior Member, IEEE*

Featured in ISCA'16, ISSCC'16, and JSSC'17!

- Designed to accelerate **CNN inference** using 16b **fixed-point** operations
- Two key ideas:
  - efficient dataflow
  - exploit data statistics to minimize memory fetches and computations
- Implements AlexNet in 278mW at a frame rate of 35 fps

# Eyeriss Architecture

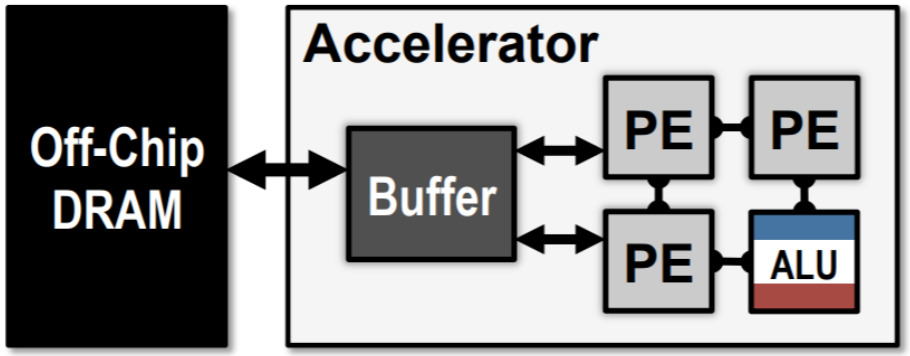


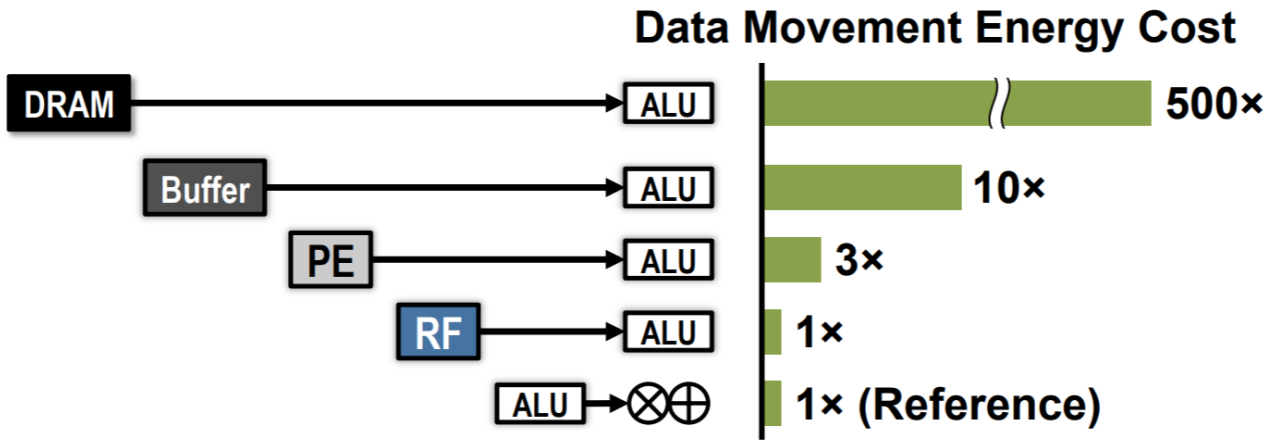




[Eyeriss-ISSCC'16]

- Systolic architecture: $12\times14$ PE array (vs $256^2$ for TPU)
- Three level memory hierarchy:
  - off-chip DRAM
  - on-chip SRAM global buffer
  - PE scratchpad
- Flexible network on chip (NoC) for data delivery from global buffer to PEs

# Importance of Data Reuse

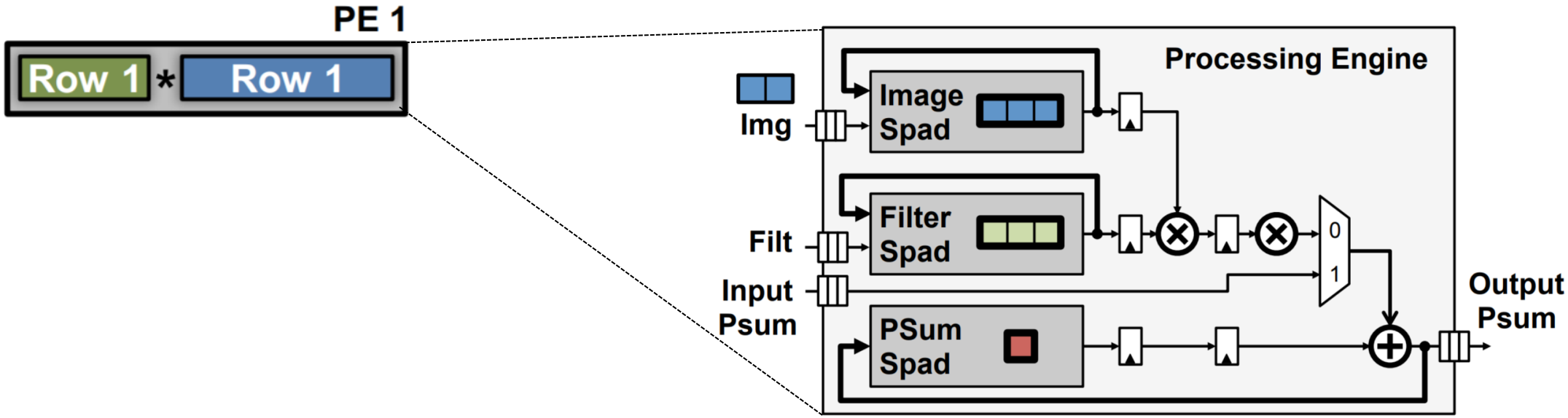


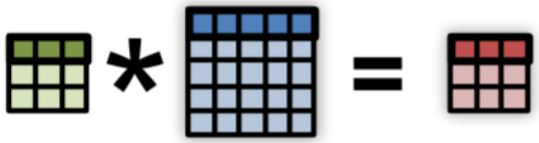




- Data movement energy **dominates**
- Goal is to minimize amount of expensive energy movements
  - Maximize local reuse of data via inter-PE communication
- How to exploit **data reuse** and **local accumulation** with limited local storage?
  - specialized processing dataflow is required!

# Row Stationary (RS) Dataflow

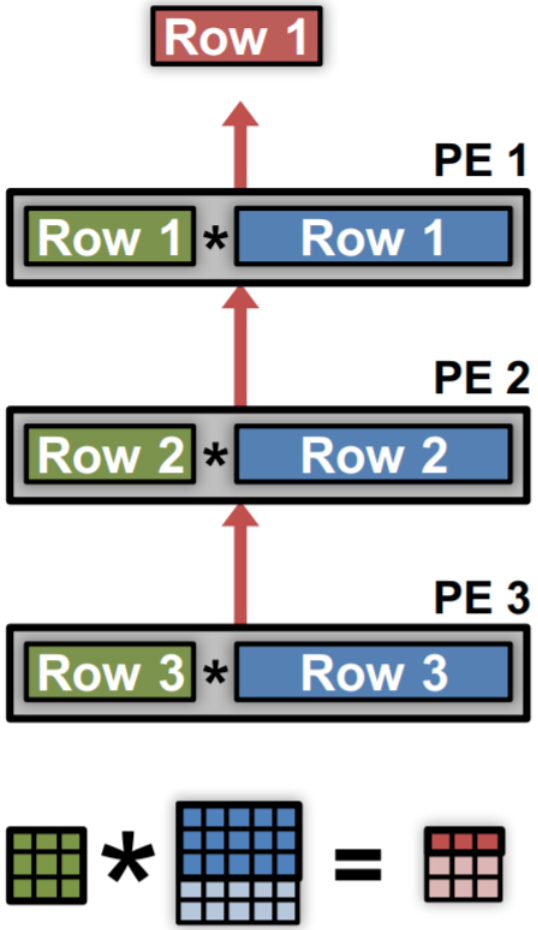
1D convolution within 1 PE



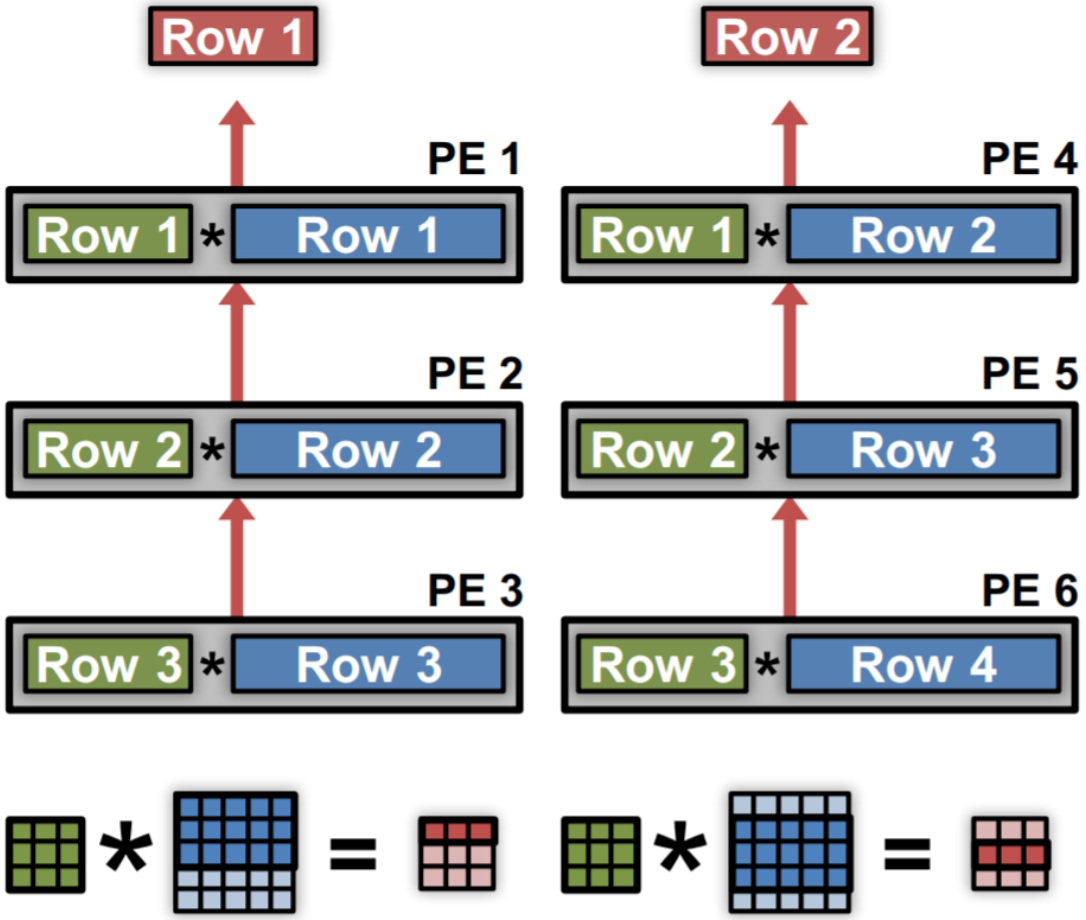
- Maximize row convolutional reuse in RF
  - Keep a **filter** row and **image** sliding window in RF
- Maximize row **psum** accumulation in RF

# Row Stationary (RS) Dataflow

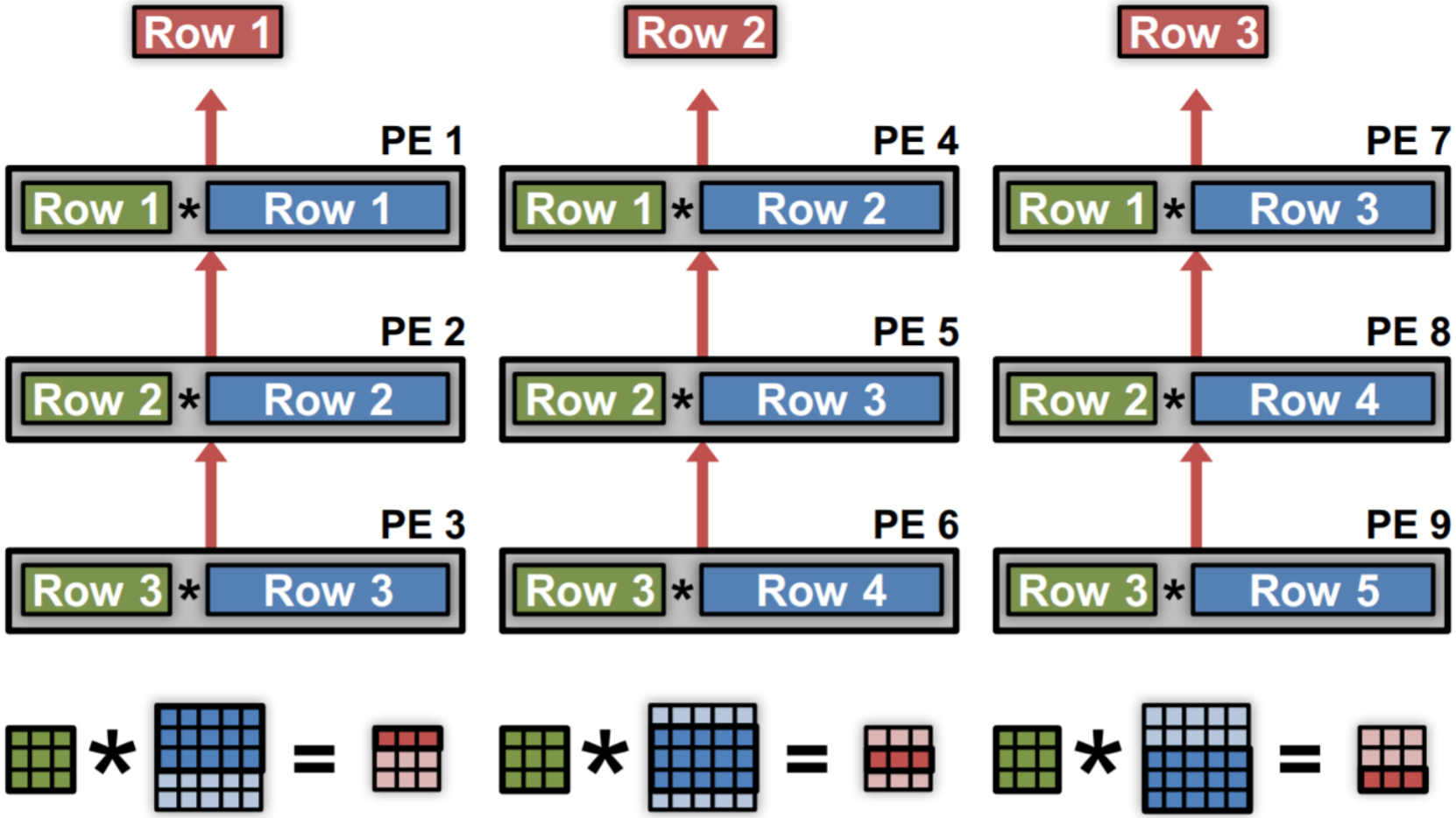



- 3 PEs can process 3 rows
- **psums** can be aggregated vertically

# Row Stationary (RS) Dataflow

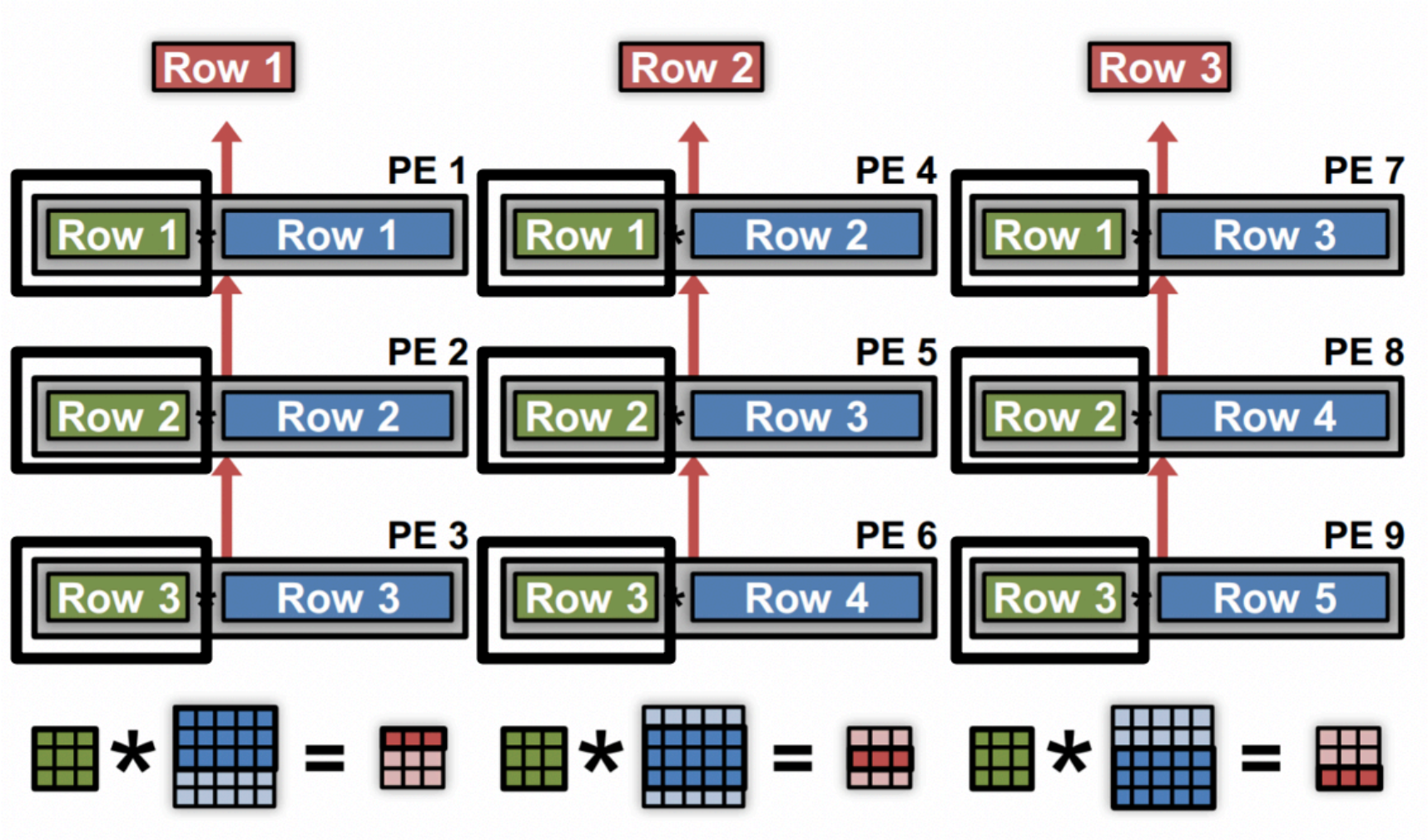


# Row Stationary (RS) Dataflow



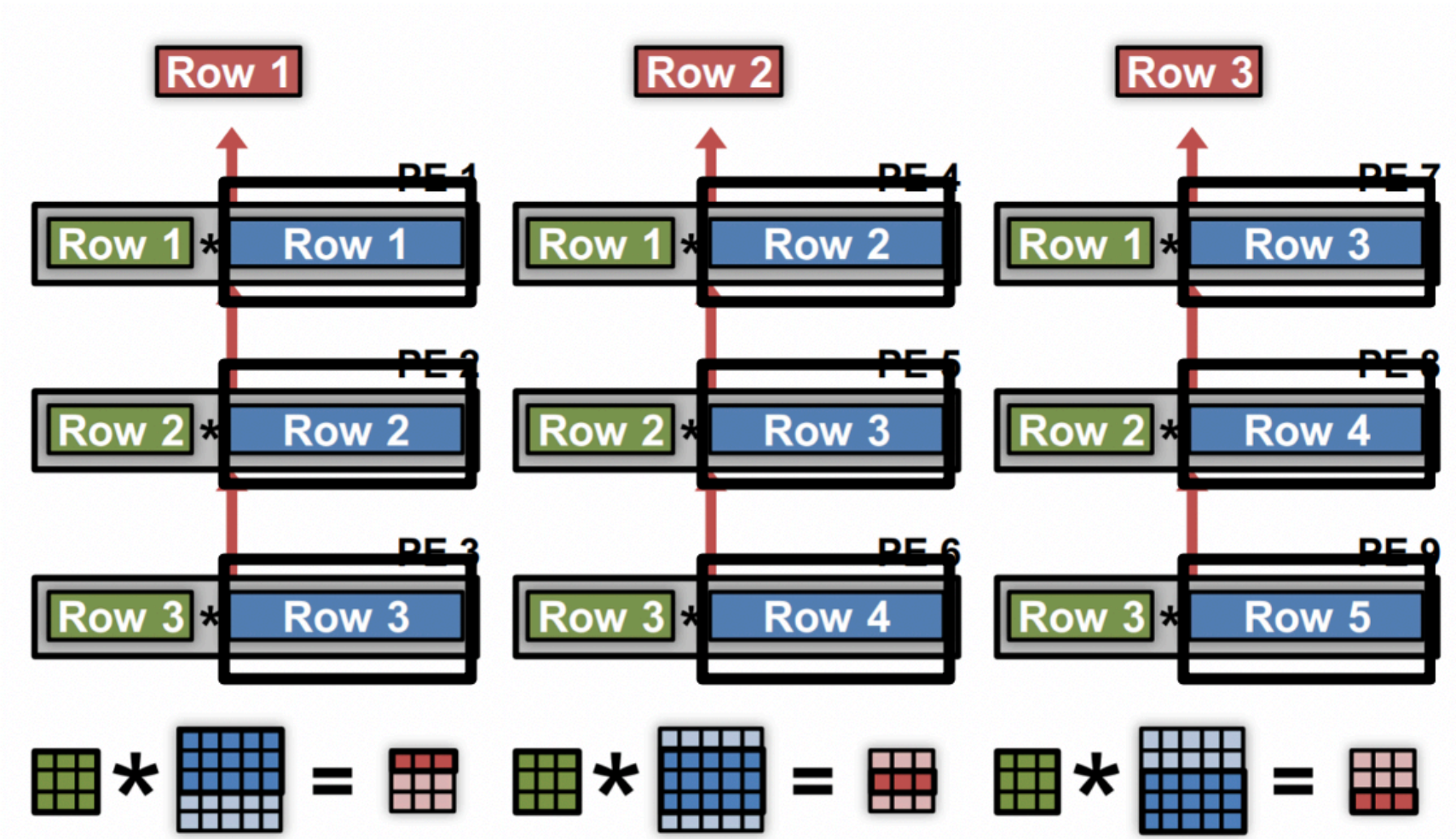

[Eyeriss-ISSCC'16]

# Row Stationary (RS) Dataflow



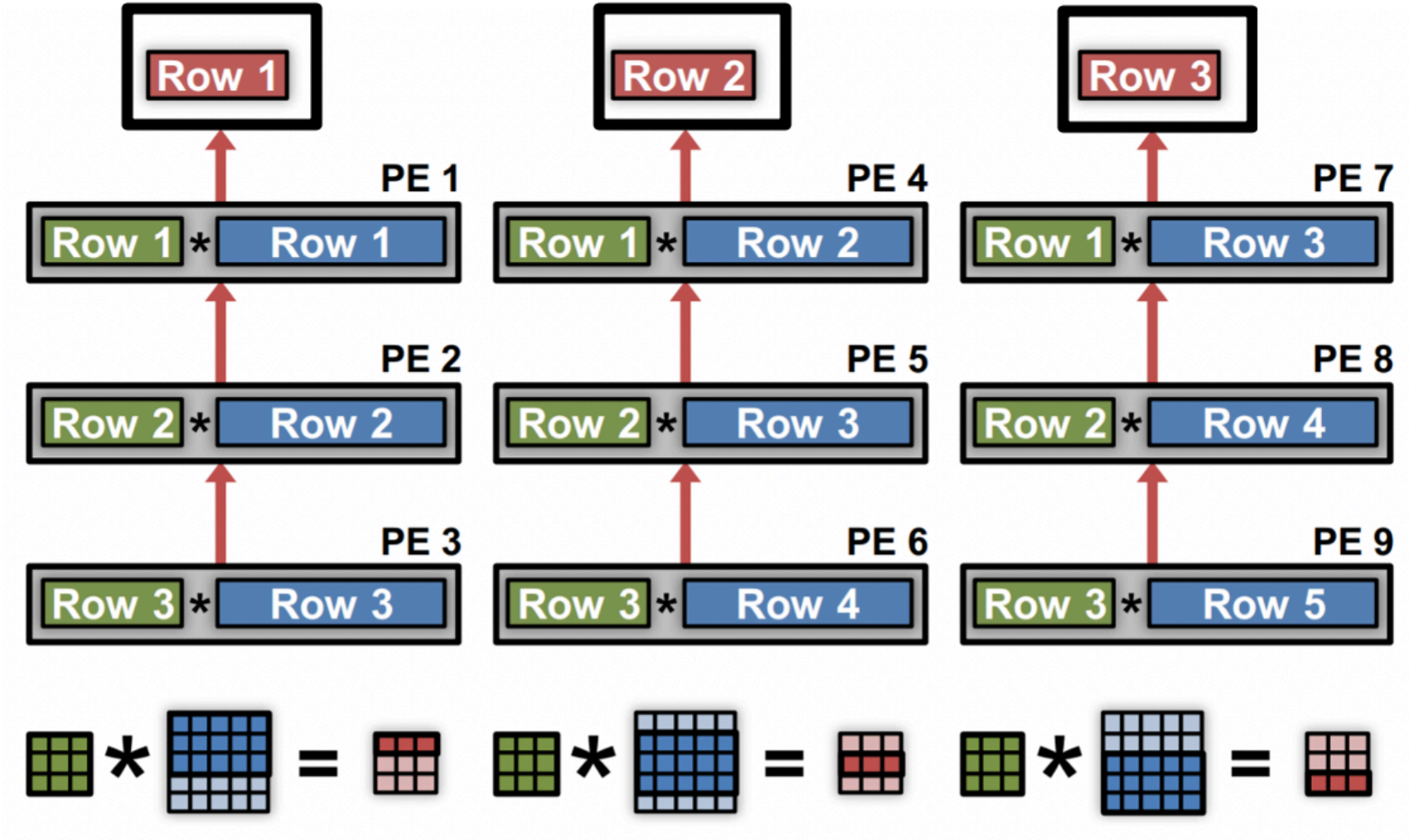

**Filter rows** are reused across PEs *horizontally*

# Row Stationary (RS) Dataflow




**Input rows** are reused across PEs *diagonally*

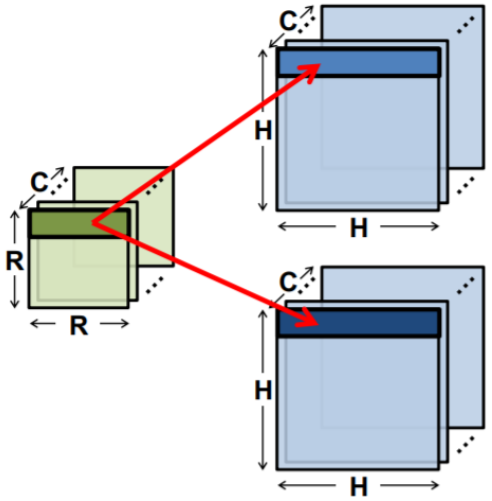
# Row Stationary (RS) Dataflow

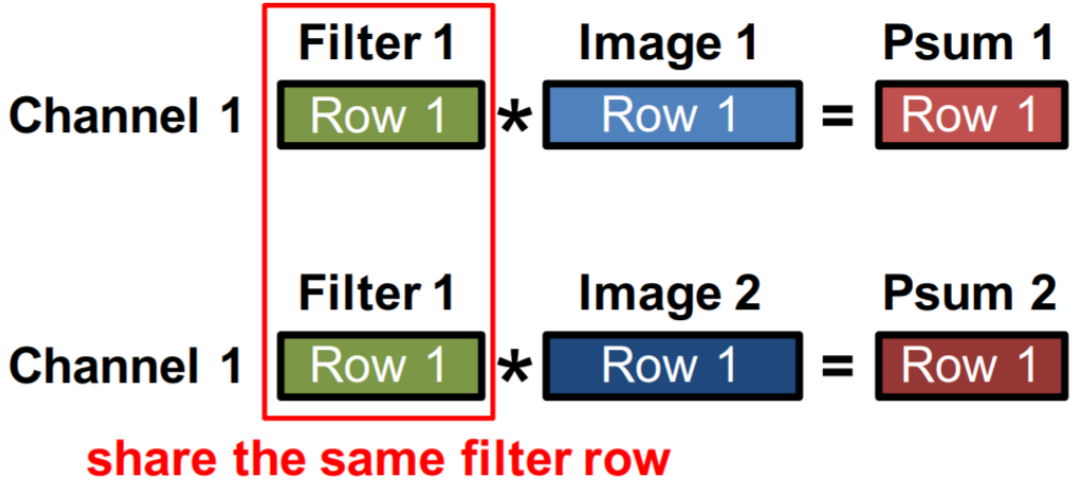



**Partial sums** accumulate across PEs *vertically*

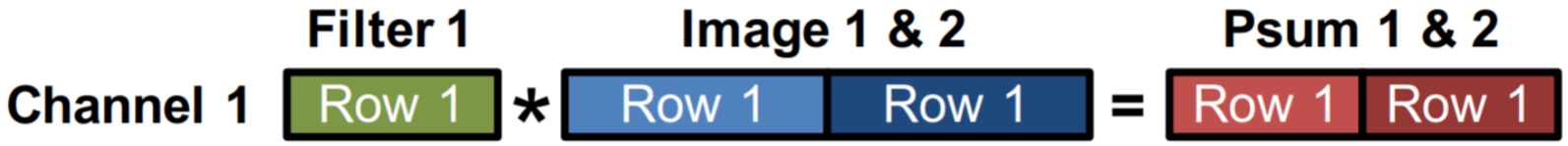
# RS Dataflow: beyond 2D Convolution

**multiple images (batch size >1): weight reuse**



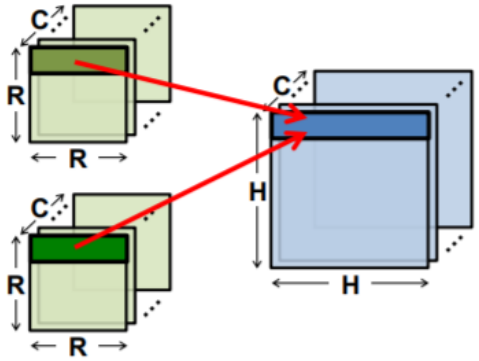

concatenate image rows in same PE:

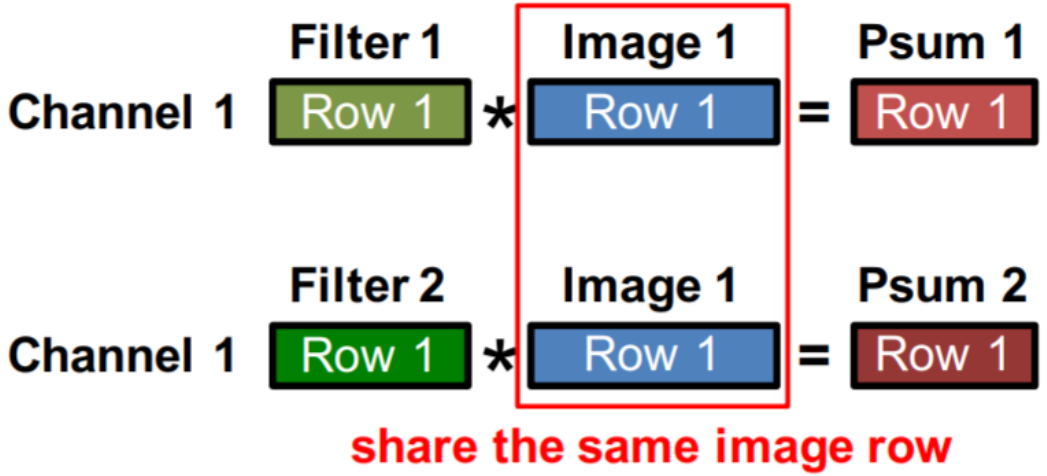



requires more local storage for **image rows** and **partial sums**

# RS Dataflow: beyond 2D Convolution

## multiple filters: input reuse



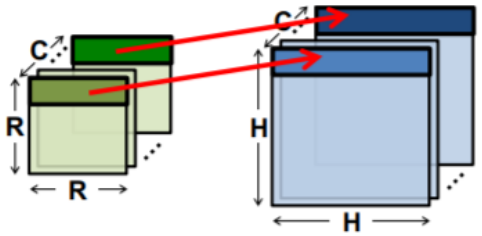



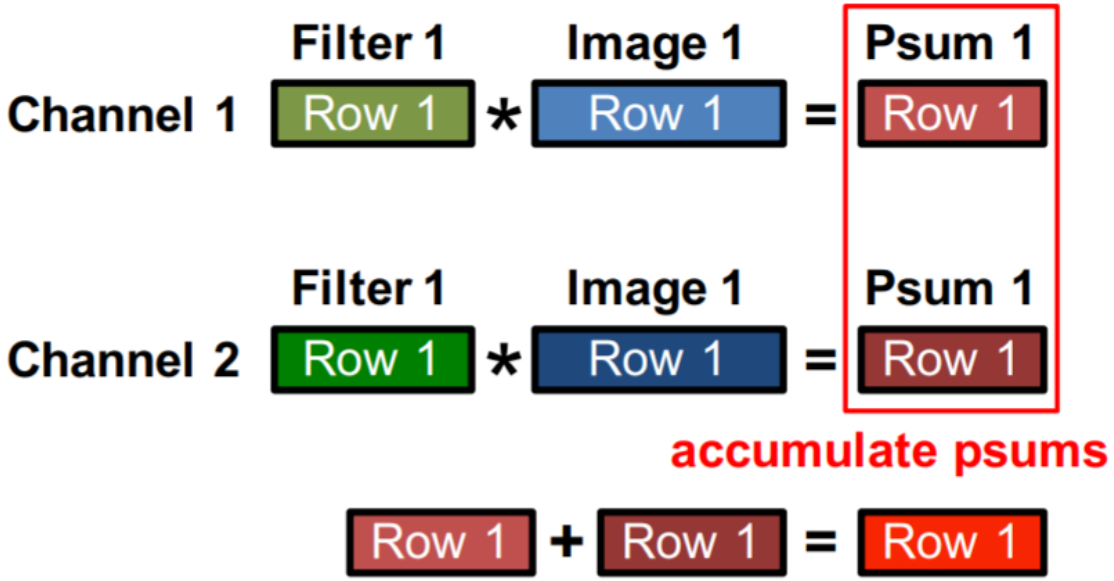

interleave filter rows in same PE:


requires more local storage for **filter rows** and **partial sums**

# RS Dataflow: beyond 2D Convolution

## multiple channels: reduce psums



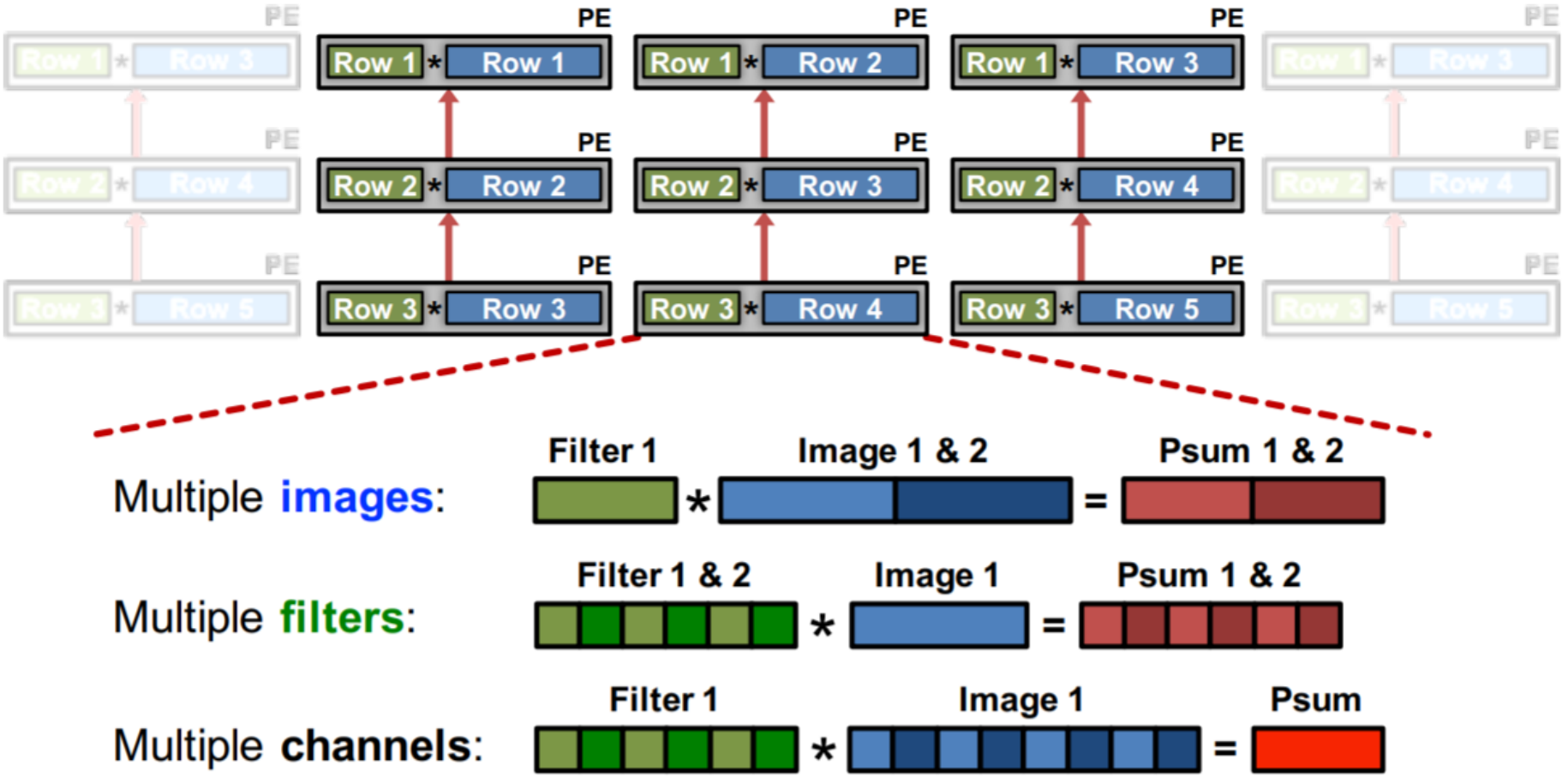




interleave channels in same PE:

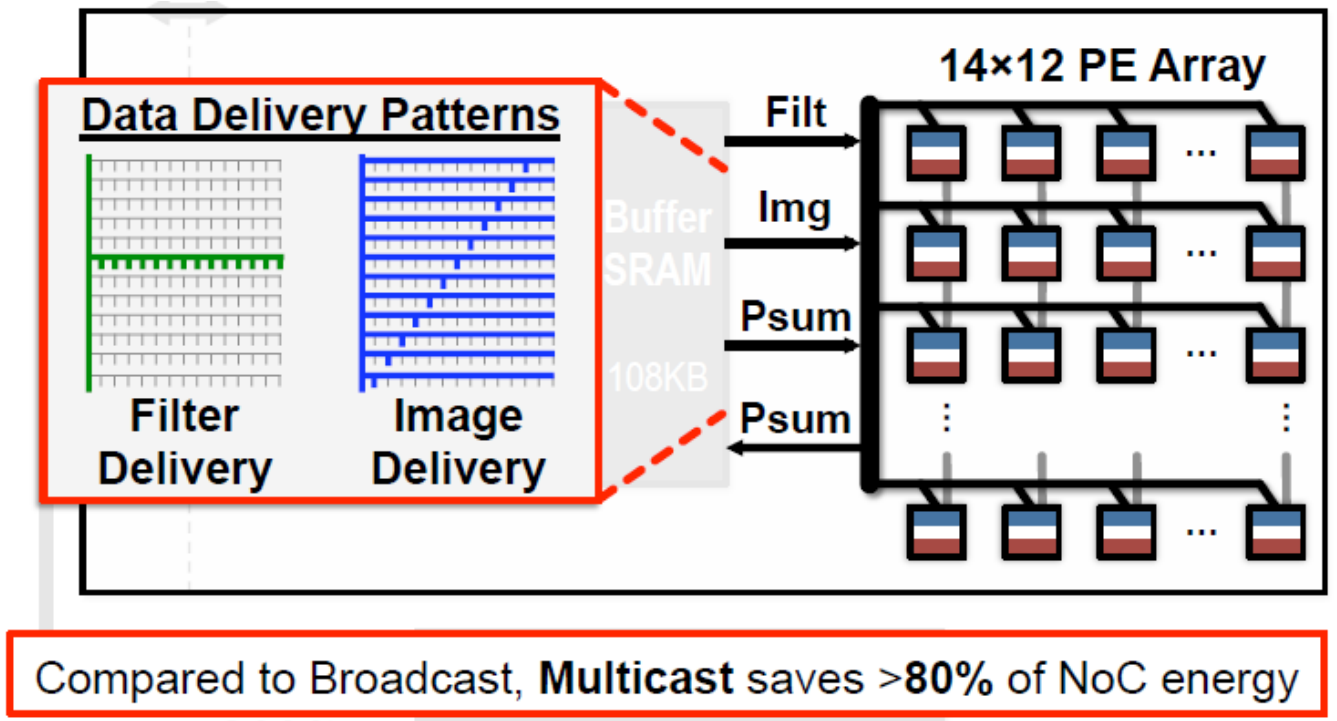

requires more local storage for **input rows** and **filter rows**

# RS Dataflow: beyond 2D Convolution

## Summary

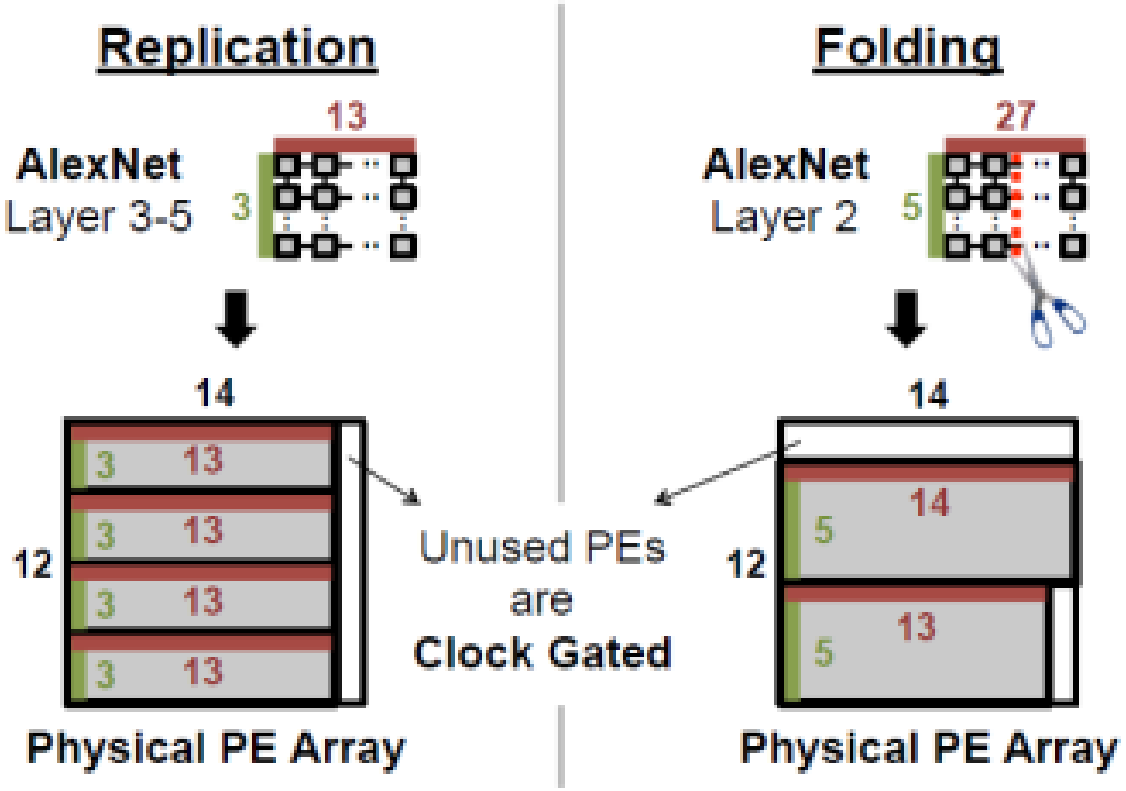


# Data Delivery with On-Chip Network



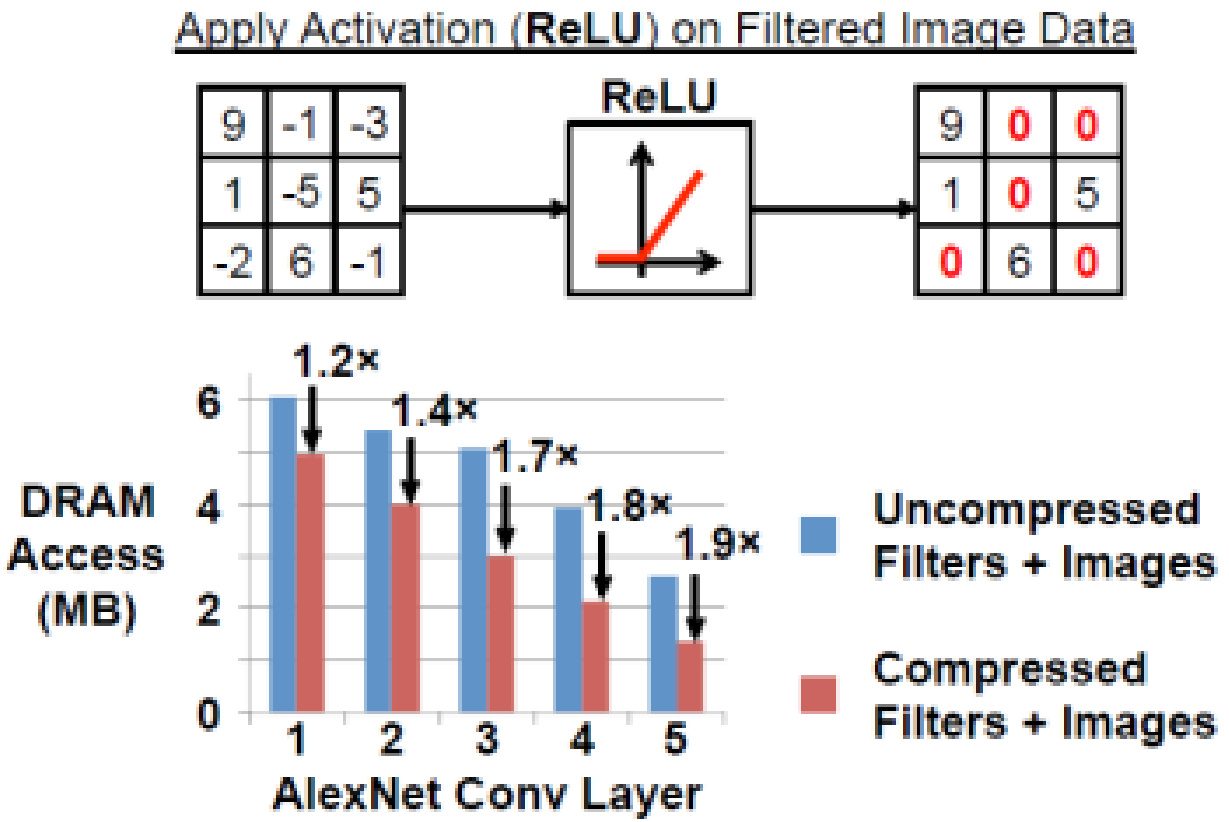

- Each PE is *configured* with a (row, col) ID at the beginning of processing
- Multicast to any subset of PEs is achieved by assigning the *same* ID to *multiple* PEs
- Separate input NoCs for filter, image, and partial sums to support high bandwidth

# Logical PE Set to Physical Set Mappings




- In order to maximize utilization of a fixed-size PE array for different shapes, use either **folding** or **replication** if the shape size is larger or smaller than the array dimension, respectively.

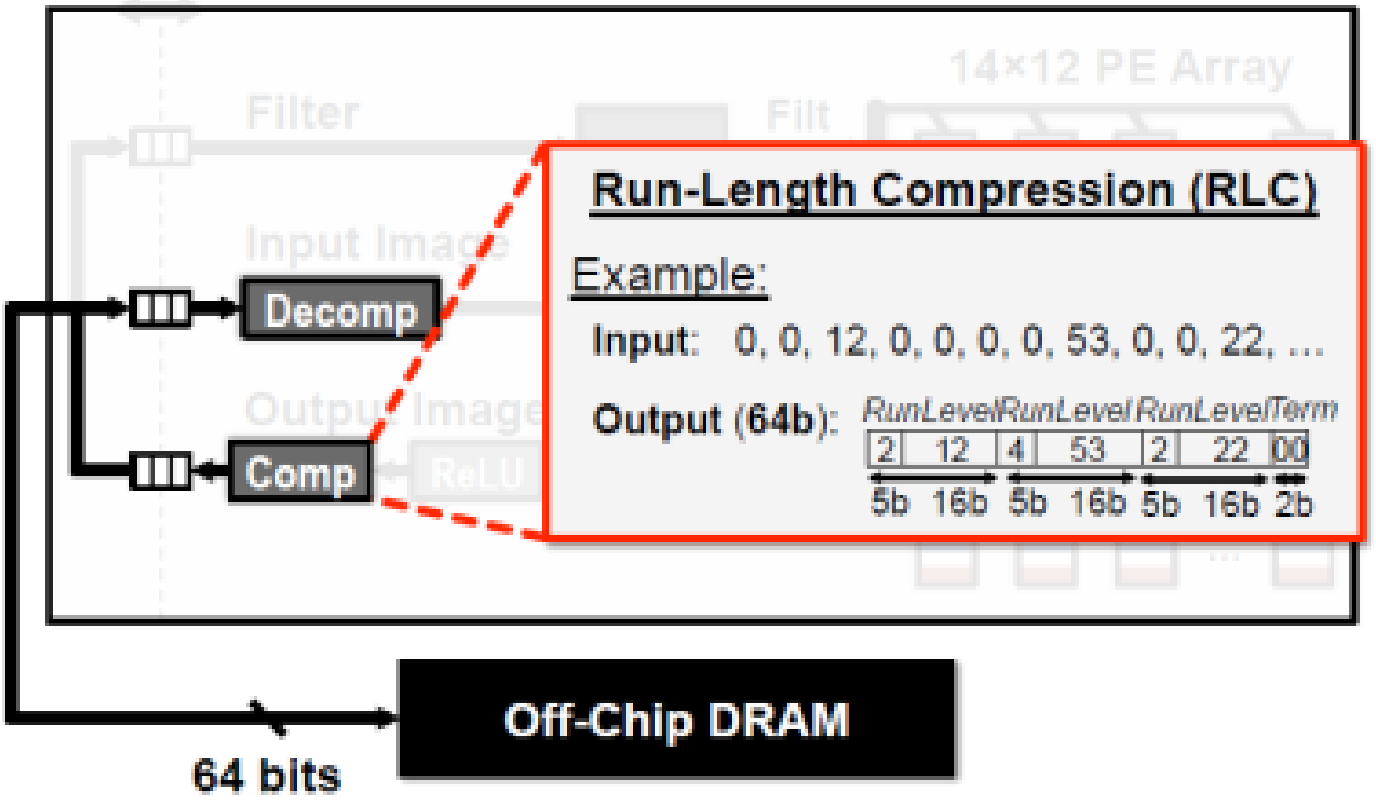
# Data Compression




- Data compression to reduce off-chip memory bandwidth, which is the most expensive data movement

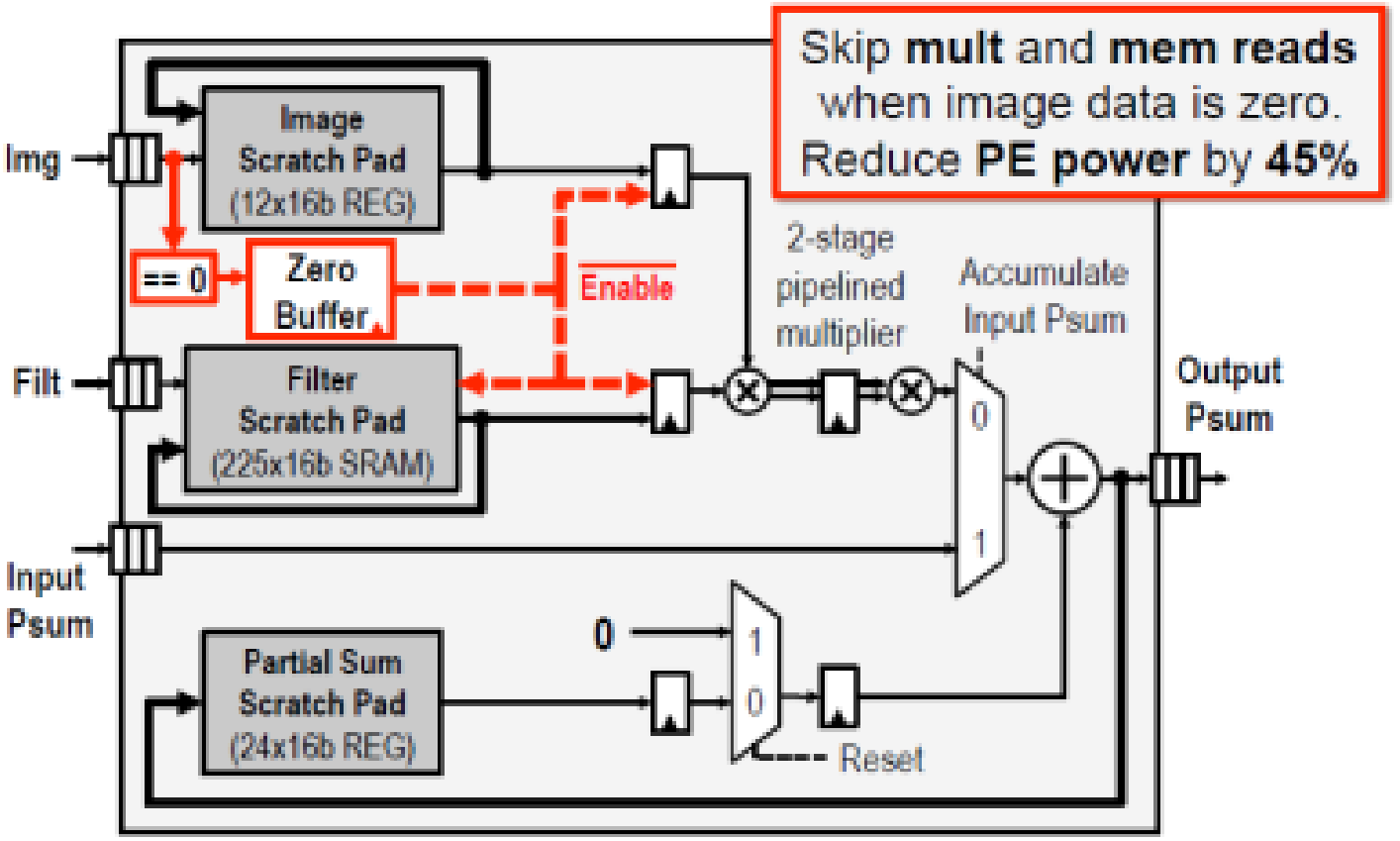
[Eyeriss-ISSCC'16]

# Data Compression




- Run-length-based compression reduces the average image bandwidth by 2×

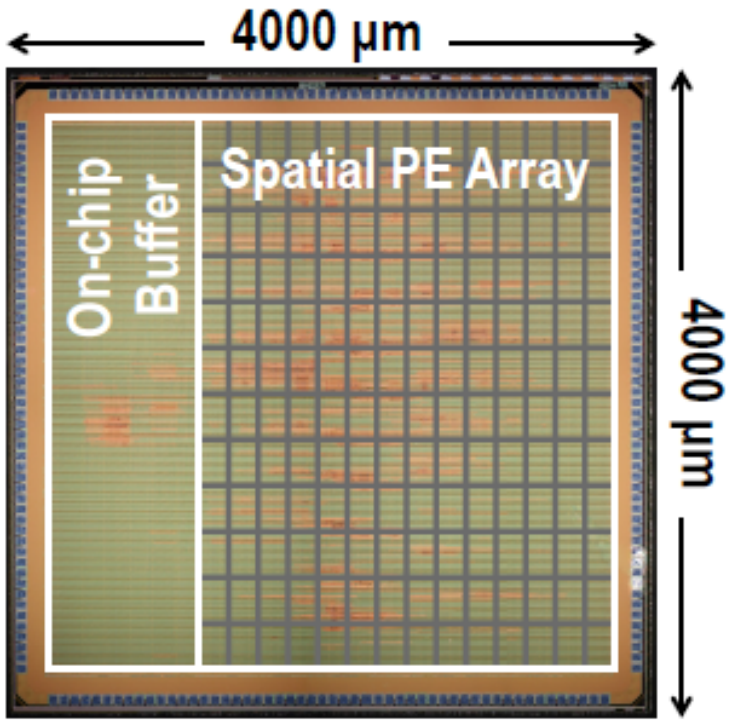
# Data Gating/Zero Skipping




- Exploit data statistics to minimize energy through zeros skipping/gating to avoid unnecessary reads and computations

# Chip Spec & Measurement Results

| | |
|---|---|
| Technology | TSMC 65nm LP 1P9M |
| Core Area | 3.5mm×3.5mm |
| Gate Count | 1852 kGates (NAND2) |
| On-Chip Buffer | 108 KB |
| # of PEs | 168 |
| Scratch Pad / PE | 0.5 KB |
| Supply Voltage | 0.82 – 1.17 V |
| Core Frequency | 100 – 250 MHz |
| Peak Performance | 33.6 – 84.0 GOPS (2 OP = 1 MAC) |
| Word Bit-width | 16-bit Fixed-Point |
| Filter Size* | 1 – 32 [width]<br>1 – 12 [height] |
| # of Filters* | 1 – 1024 |
| # of Channels* | 1 – 1024 |
| Stride Range | 1–12 [horizontal]<br>1, 2, 4 [vertical] |



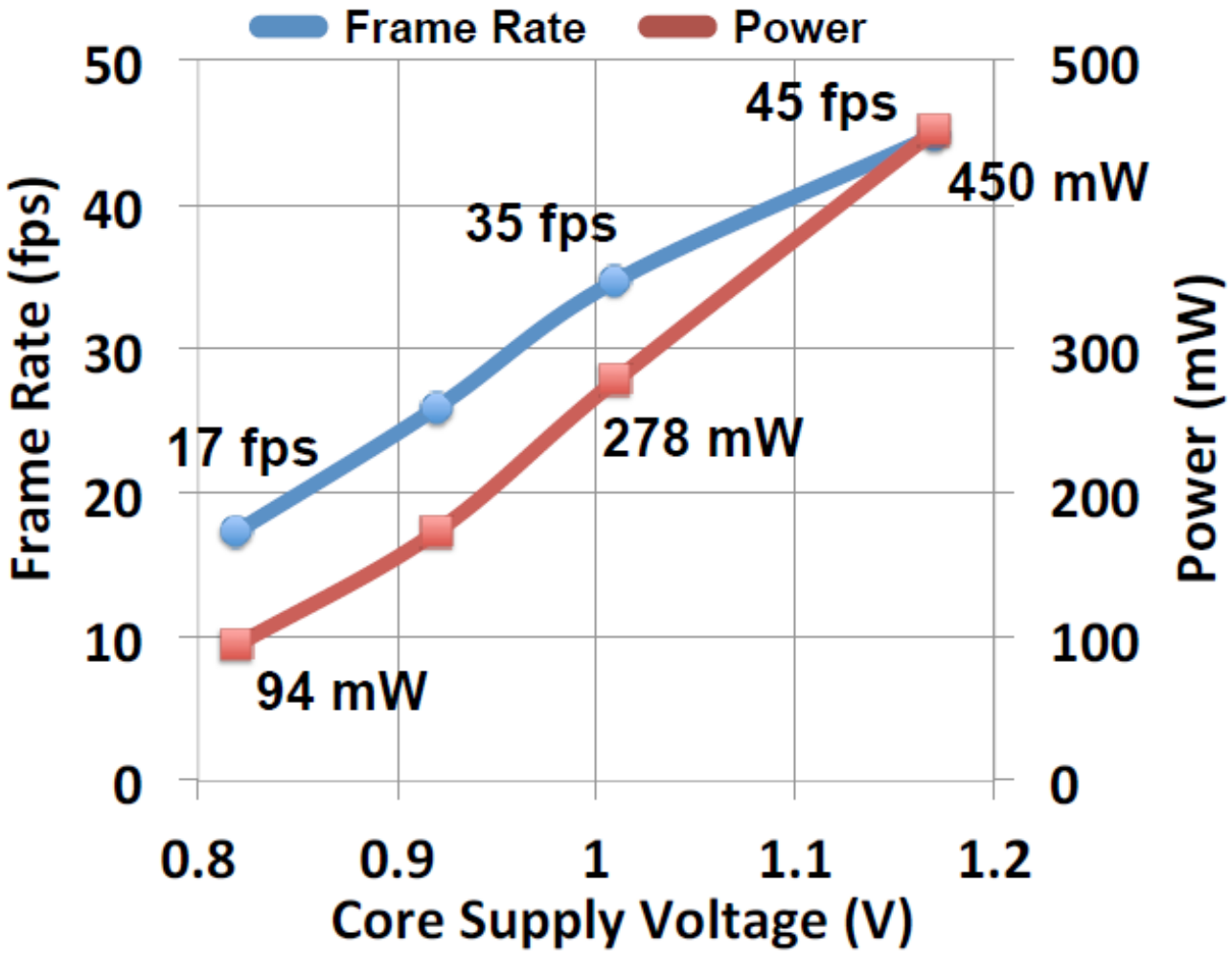

*** Natively Supported**

# Benchmark – AlexNet Performance

Image Batch Size of **4** (i.e. 4 frames of 227x227)
Core Frequency = 200MHz / Link Frequency = 60 MHz

| Layer | Power (mW) | Latency (ms) | # of MAC (MOPs) | Active # of PEs (%) | Buffer Data Access (MB) | DRAM Data Access (MB) |
|---|---|---|---|---|---|---|
| 1 | 332 | 20.9 | 422 | 154 (92%) | 18.5 | 5.0 |
| 2 | 288 | 41.9 | 896 | 135 (80%) | 77.6 | 4.0 |
| 3 | 266 | 23.6 | 598 | 156 (93%) | 50.2 | 3.0 |
| 4 | 235 | 18.4 | 449 | 156 (93%) | 37.4 | 2.1 |
| 5 | 236 | 10.5 | 299 | 156 (93%) | 24.9 | 1.3 |
| **Total** | **278** | **115.3** | **2663** | **148 (88%)** | **208.5** | **15.4** |

# AlexNet Throughput vs. Power



# Comparison with GPU

| | *This Work* | NVIDIA TK1 (Jetson Kit) |
|---|---|---|
| **Technology** | 65nm | 28nm |
| **Clock Rate** | 200MHz | 852MHz |
| **# Multipliers** | 168 | 192 |
| **On-Chip Storage** | Buffer: 108KB<br>Spad: 75.3KB | Shared Mem: 64KB<br>Reg File: 256KB |
| **Word Bit-Width** | 16b Fixed | 32b Float |
| **Throughput[1]** | 34.7 fps | 68 fps |
| **Measured Power** | 278 mW | Idle/Active[2]: 3.7W/10.2W |
| **DRAM Bandwidth** | 127 MB/s | 1120 MB/s [3] |

1. AlexNet Convolutional Layers Only
2. Board Power
3. Modeled from [Tan, SC11]

[Eyeriss-ISSCC'16]

# Eyeriss: Summary

- **A 278mW reconfigurable accelerator for state-of-the-art deep CNNs**
  - a 168-PE spatial architecture that supports an efficient dataflow to minimize data movement
  - a configurable multicast NoC that saves energy compared to a broadcast design
- **Exploits data statistics for higher efficiency**
  - compression to reduce memory bandwidth
  - zero-skipping logic to reduce PE power
- **Integrated with the Caffe DL framework and demonstrated an image classification system**

# Course Web Page

https://courses.grainger.illinois.edu/ece598nsg/fa2020/
https://courses.grainger.illinois.edu/ece498nsu/fa2020/

http://shanbhag.ece.uiuc.edu

# Updates

- include inference-level floorline and roofline plots