Lecture Series on
Hardware for Deep Learning

Part 3: Computing Convolutions

Dr. Adam Teman

EnICS Labs, Bar-Ilan University

1 April 2020

EnICS
Emerging Nanoscaled
Integrated Circuits and Systems Labs

Tradition of Excellence
Bar-Ilan University
אוניברסיטת בר-אילן

Outline

Mapping Matrix Multiplication
Computational Transforms
Accelerator Architectures
Dataflow Taxonomy
Mapping Matrix Multiplication
Computational Transforms
Accelerator Architectures
Dataflow Taxonomy
EnICS
Emerging Nanoscaled Integrated Circuits and Systems Labs
Bar-Ilan University
אוניברסיטת בר-אילן

Mapping Matrix Multiplication

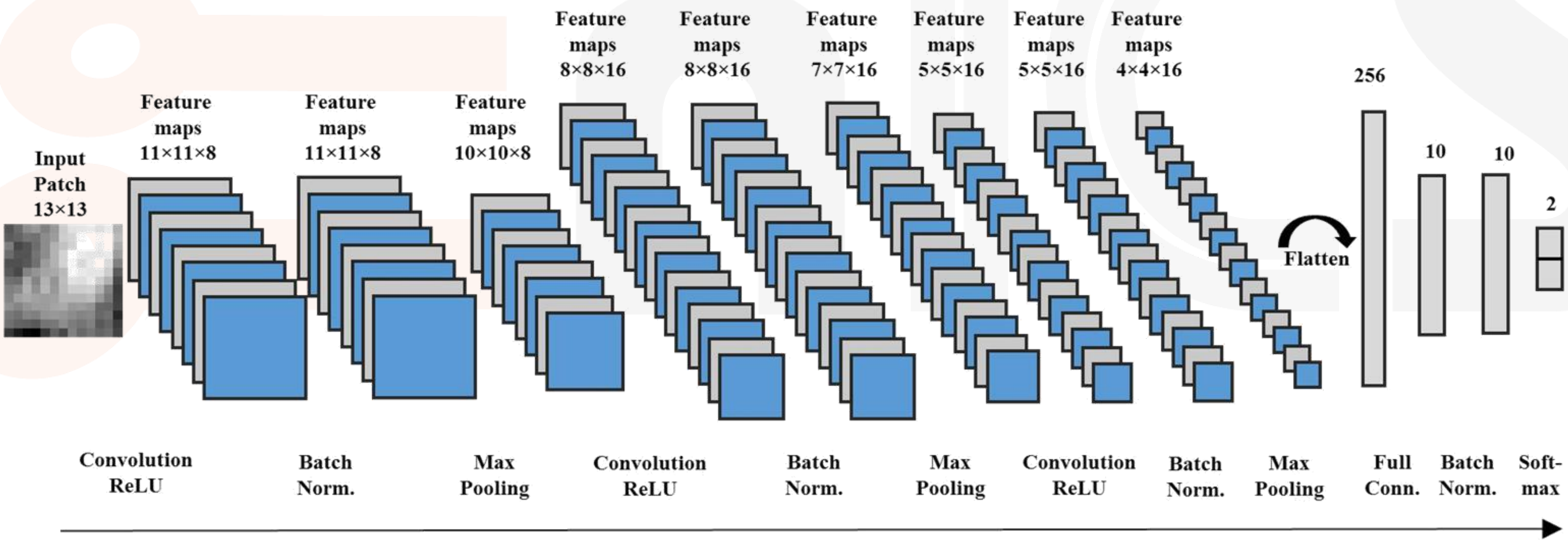
Deep Convolutional Neural Networks

- Convolutions account for **more than 90% of overall computation**, dominating **runtime** and **energy consumption**

Output fmaps (O)
Biases (B)
Input fmaps (I)
Filter weights (W)

$$O[n][m][x][y] = \text{Activation}\left(B[m] + \sum_{i=0}^{R-1}\sum_{j=0}^{S-1}\sum_{k=0}^{C-1} I[n][k][Ux+i][Uy+j] \times W[m][k][i][j]\right),$$

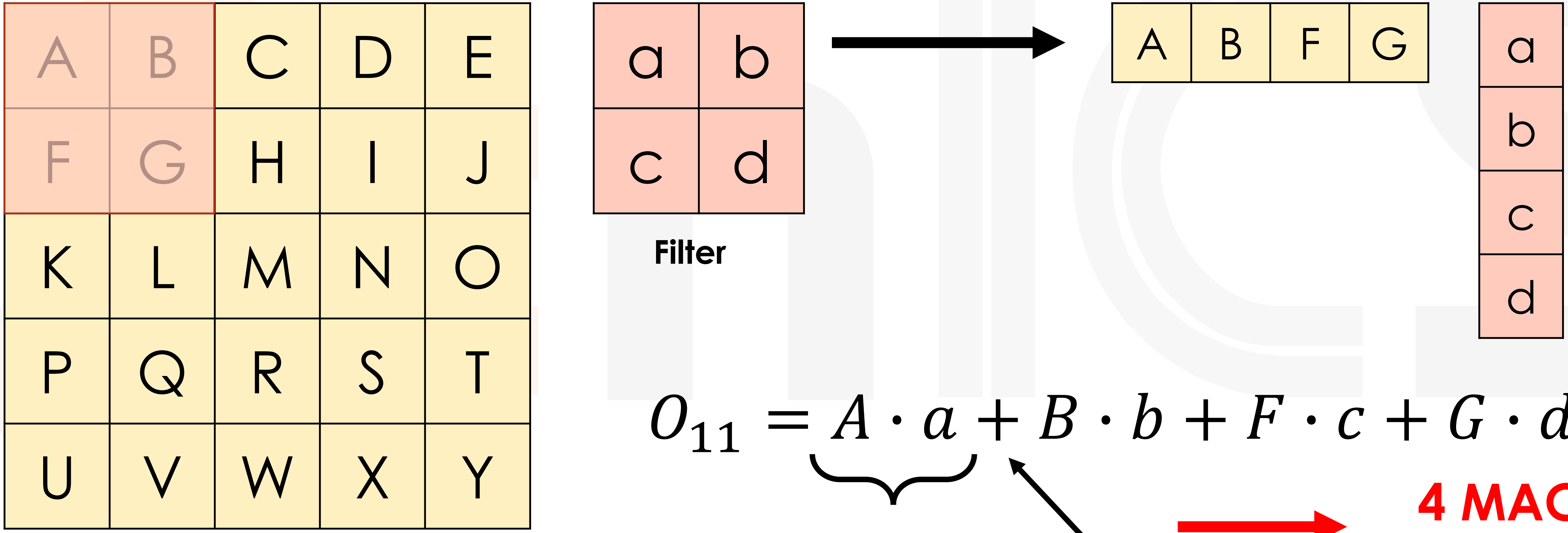
Source: Sze, MIT



Source: Hailong Li, et al. 2019

Convolutions are Dot-Products

- **Flatten window of interest into 1-D dot product**



Input Feature Map

Filter

$$O_{11} = \underbrace{A \cdot a}_{\text{Multiply}} + B \cdot b + F \cdot c + G \cdot d$$

Accumulate

4 MAC Operations

Toeplitz Matrix

- Convert the input matrix to a Toeplitz Matrix to enable a matrix multiplication for the entire convolutional layer

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} * \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 11 & 12 \\ 21 & 22 \end{bmatrix}$$

Input fmap

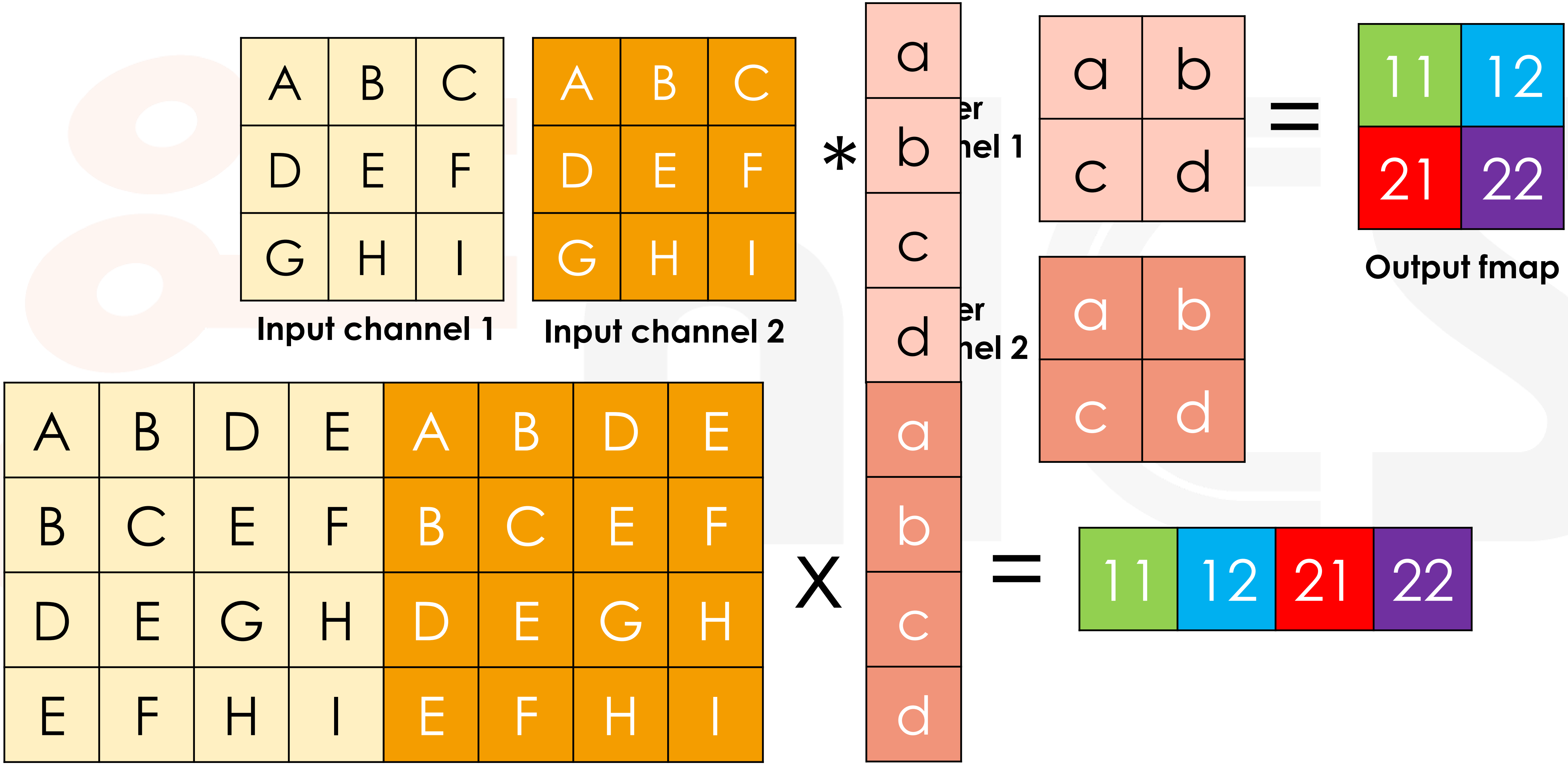
Filter

Output fmap

4x4 ifmap
(instead of 3x3)

$$\begin{bmatrix} A & B & D & E \\ B & C & E & F \\ D & E & G & H \\ E & F & H & I \end{bmatrix} \times \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 11 & 12 & 21 & 22 \end{bmatrix}$$

Toeplitz Matrix – Multiple Input Channels



Toeplitz Matrix – Multiple Input and Output Channels

A	B	D	E	A	B	D	E
B	C	E	F	B	C	E	F
D	E	G	H	D	E	G	H
E	F	H	I	E	F	H	I

Input channel 1 Input channel 2

X

Filter 1	Filter 2
a	a
b	b
c	c
d	d
a	a
b	b
c	c
d	d

=

Output channel 1

11	12	21	22
11	12	21	22

Output channel 2

Note that several software packages can map these conversions, e.g. OpenBLAS, Intel MKL for CPU and cuBLAS, cuDNN for GPU

Computational Transforms

Emerging Nanoscaled
Integrated Circuits and Systems Labs
Bar-Ilan University
אוניברסיטת בר-אילן

Computational Transforms

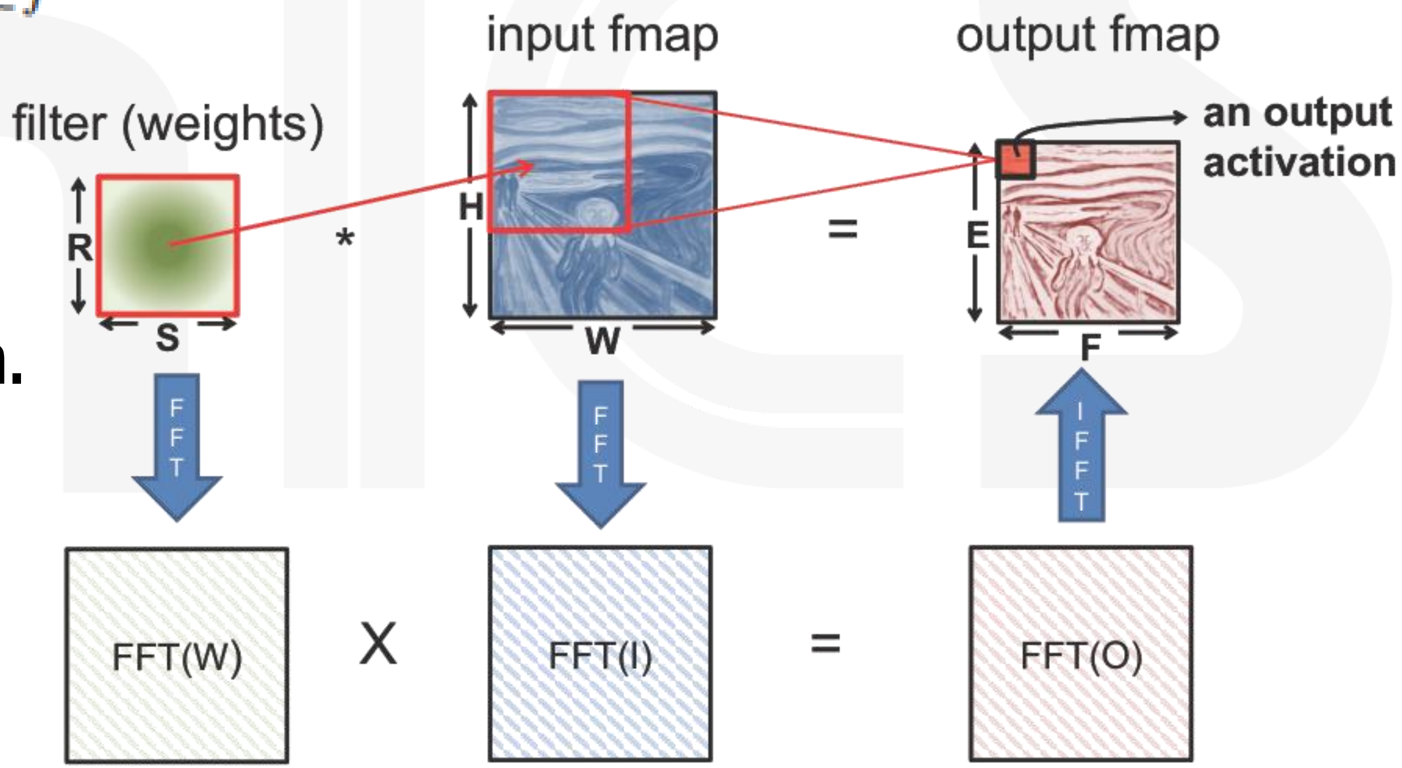
- **Multiplications are an expensive operation**
  - Much more expensive than addition
- **Several computational transforms are commonly applied to reduce the number of multiplications:**
  - Fast Fourier Transform
  - Winograd's Algorithm
  - Strasen's Algorithm

Fast Fourier Transform (FFT)

- Convolution can be computed using FFT [Mathieu et al., 2014]:

$$y_{(s,j)} = \sum_{i \in f} x_{(s,i)} \star w_{(j,i)} = \sum_{i \in f} \mathcal{F}^{-1}\left(\mathcal{F}(x_{(s,i)}) \circ \mathcal{F}(w_{(j,i)})^*\right)$$

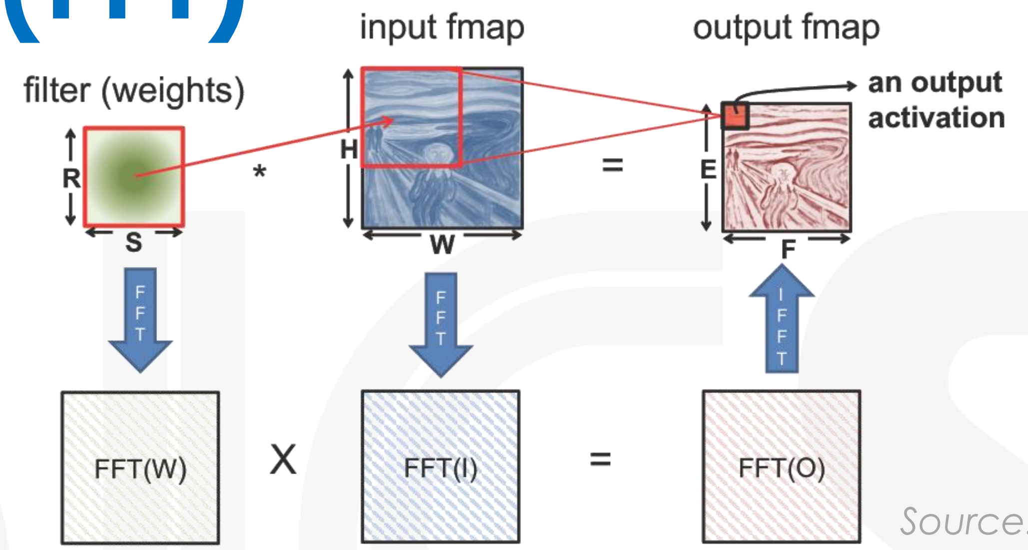
- Convert filter and input to frequency domain to make convolution a simple multiply then convert back to space domain.



- Reduces multiplication complexity
  - $O(N_o^2 N_f^2)$ to $O(N_o^2 \log_2 N_f)$

Source: Sze, MIT

Fast Fourier Transform (FFT)



Source: Sze, MIT

- **Benefits**
  - Reduced complexity.
  - The larger the convolution kernel, the better the performance
- **However**
  - Filter sizes are often small…
  - Coefficients in the frequency domain are complex
  - Difficult to exploit sparsity
- **FFT often reduces computation, but requires much more memory space and bandwidth.**

<u>FFT Optimization opportunities</u>

- FFT of real matrix is symmetric
  - save ½ the computes
- Filters can be pre-computed and stored
  - But filter in frequency domain is large
- Reuse FFT of input fmaps for creating different output channels
- Accumulate across channels before IFFT

Algorithmic Optimizations

- **Winograd's** and **Strassen's** **Algorithms are algorithmic optimizations.**
- **As a simple example, take Gauss's Multiplication Algorithm**

$$(a+bi)\,(c+di) = \underbrace{(ac+bd)+(bc+ad)i}_{\substack{\text{4 Multiplications} \\ \text{3 additions}}}$$

$$\underbrace{\begin{aligned} k_1 &= c(a+b) \\ k_2 &= a(d-c) \\ k_3 &= b(c+d) \end{aligned}}_{\substack{\text{3 Multiplications} \\ \text{3 additions}}}
\qquad
\underbrace{\begin{aligned} &\text{Real Part} = k_1 - k_3 \\ &\text{Imaginary Part} = k_1 + k_2 \end{aligned}}_{\text{2 additions}}$$

Strassen's Algorithm

- **Reduces the complexity of matrix multiplication from $O(N^3)$ to $O(N^{2.807})$ by reducing multiplications**

$$\overbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}^{A}\overbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}^{B}=\underbrace{\overbrace{\begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}}^{C}}_{\text{8 Multiplications+4 additions}}$$

$$\begin{aligned} P1 &= a(f-h) \\ P2 &= (a+b)h \\ P3 &= (c+d)e \\ P4 &= d(g-e) \end{aligned} \qquad \begin{aligned} P5 &= (a+d)(e+h) \\ P6 &= (b-d)(g+h) \\ P7 &= (a-c)(e+f) \end{aligned}$$

$$C=\underbrace{\begin{bmatrix} P5+P4-P2+P6 & P1+P2 \\ P3+P4 & P1+P5-P3-P7 \end{bmatrix}}_{\substack{\text{7 Multiplications + 18 additions} \\ \text{or 7 Multiplications + 13 additions} \\ \text{(for constant B matrix)}}}$$

- **Comes at the price of reduced numerical stability and requires significantly more memory**

Winograd 1D – F(2,3)

- **Targeting convolutions instead of matrix multiply**
- **Notation:** $F(\text{size of output, filter size})$

$$\begin{array}{cccc} & \text{input} & \text{filter} & \\ F(2,3) = & \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} & \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} & = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \end{array}$$

6 multiplications + 4 additions

- **Applied to blocks of the input. Efficiency depends on filter and block size.**

$$\begin{array}{cccc} & \text{input} & \text{filter} & \\ F(2,3) = & \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} & \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} & = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix} \end{array}$$

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

4 multiplications + 12 additions + 2 shifts

4 multiplications + 8 additions (for constant weights)

[Lavin et al., CVPR 2016]

Winograd 2D - F(2x2, 3x3)

- **1D Winograd is nested to make 2D Winograd**

Filter * Input Fmap = Output Fmap

$$\begin{bmatrix} g_{00} & g_{01} & g_{02} \\ g_{10} & g_{11} & g_{12} \\ g_{20} & g_{21} & g_{22} \end{bmatrix} * \begin{bmatrix} d_{00} & d_{01} & d_{02} & d_{03} \\ d_{10} & d_{11} & d_{12} & d_{13} \\ d_{20} & d_{21} & d_{22} & d_{23} \\ d_{30} & d_{31} & d_{32} & d_{33} \end{bmatrix} = \begin{bmatrix} y_{00} & y_{01} \\ y_{10} & y_{11} \end{bmatrix}$$

Source: Sze, MIT

- **Winograd is an optimized computation for convolutions**

Original: 36 multiplications

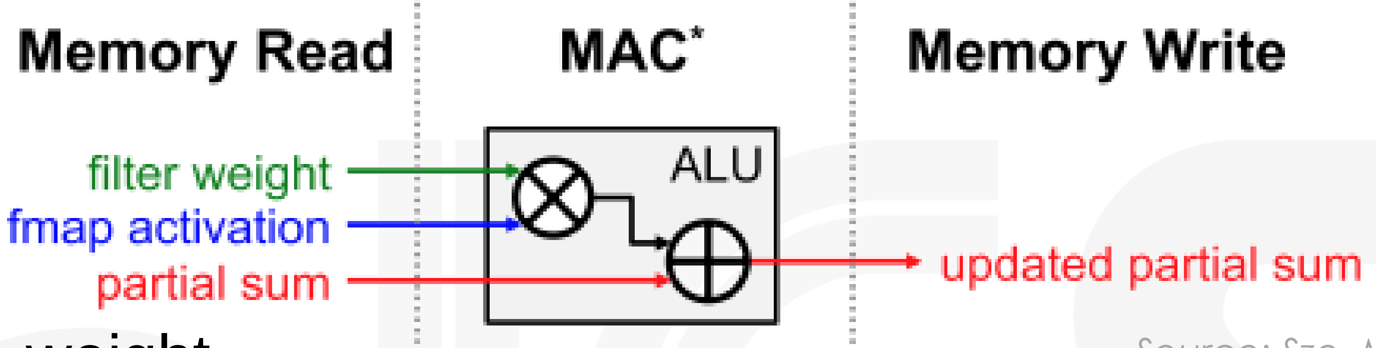
Winograd: 16 multiplications → 2.25 times reduction

- **It can significantly reduce multiplies**
  - For example, for 3x3 filter by 2.25X
- **But, each filter size (and output size) is a different computation.**

Accelerator Architectures

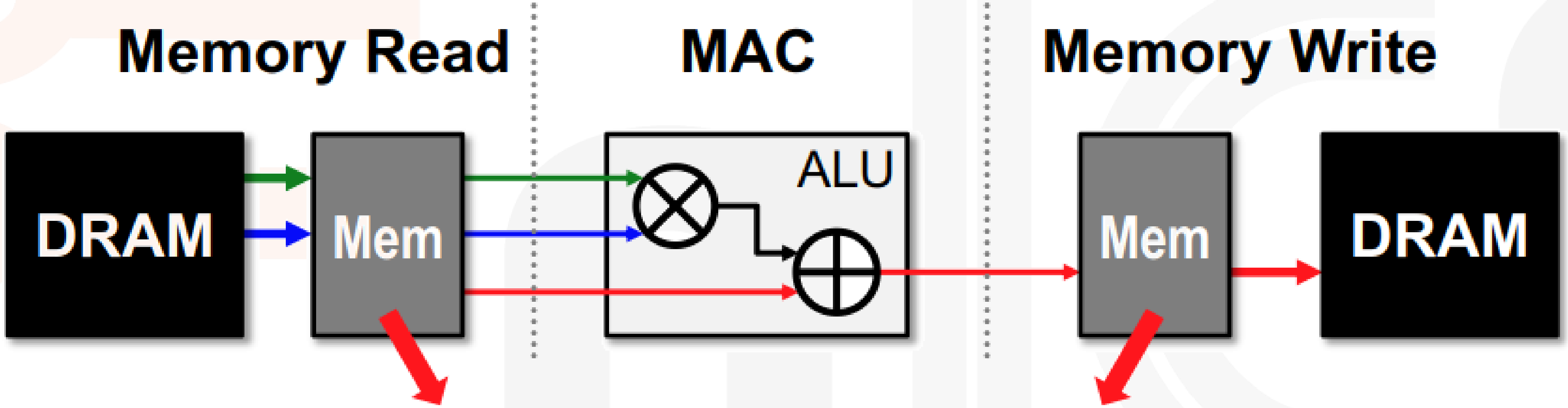
Memory Access Bottleneck

- **How is a MAC calculated?**
  - Read the fmap activation
  - Read the filter weight
  - Multiply the activation and the weight
  - Read the partial sum
  - Add the multiplication product to the sum
  - Store the updated partial sum
- **Therefore, each MAC requires four memory accesses!**
  - For example, AlexNet requires nearly 3 Billion memory accesses per inference
  - In the worst case, each memory access is to DRAM
  - DRAM accesses are much slower and much higher energy consumers than computation.



Source: Sze, MIT

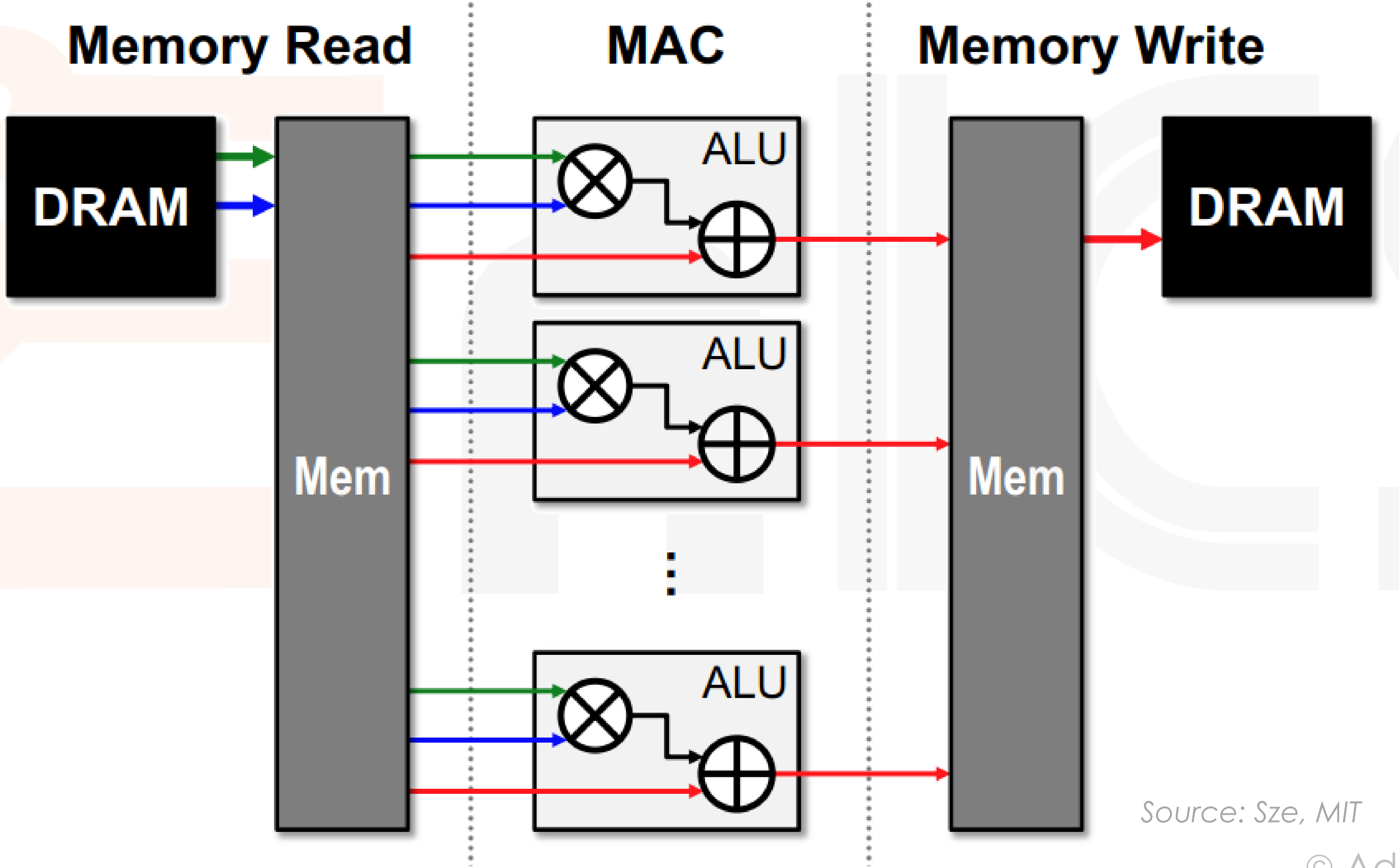
Leverage Local Memory for Data Reuse



Extra levels of local memory hierarchy
Smaller, but **Faster** and **more Energy-Efficient**

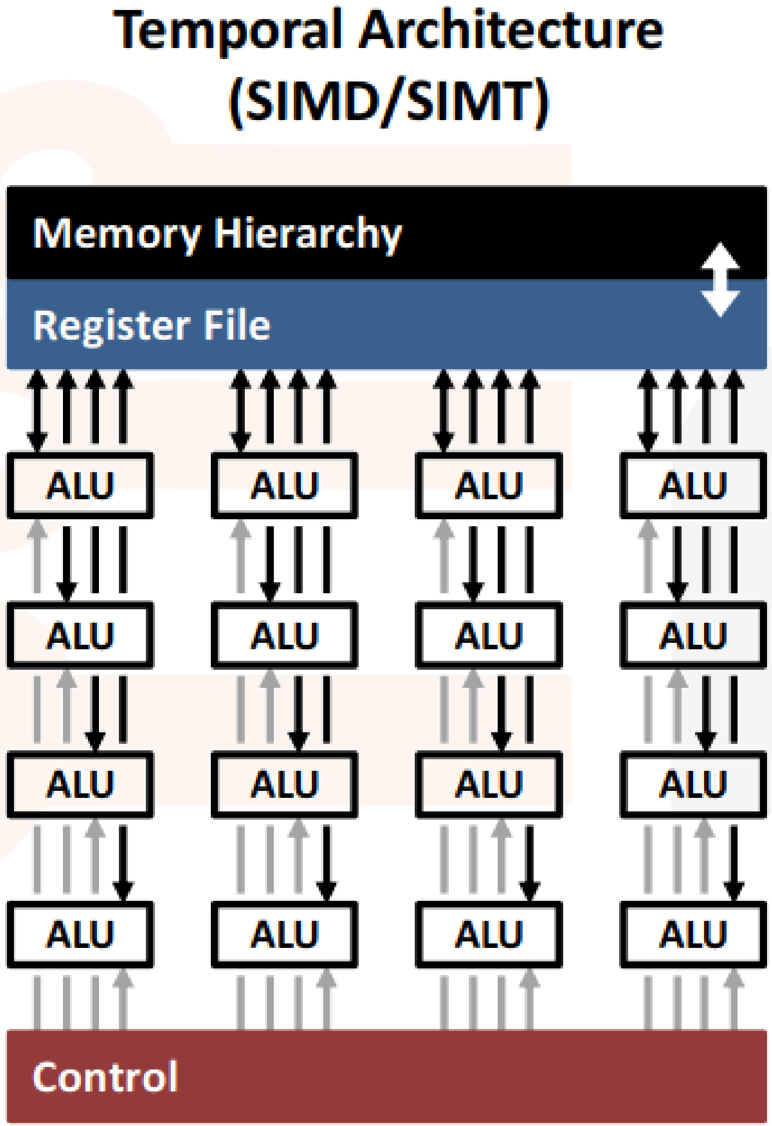
Source: Sze, MIT

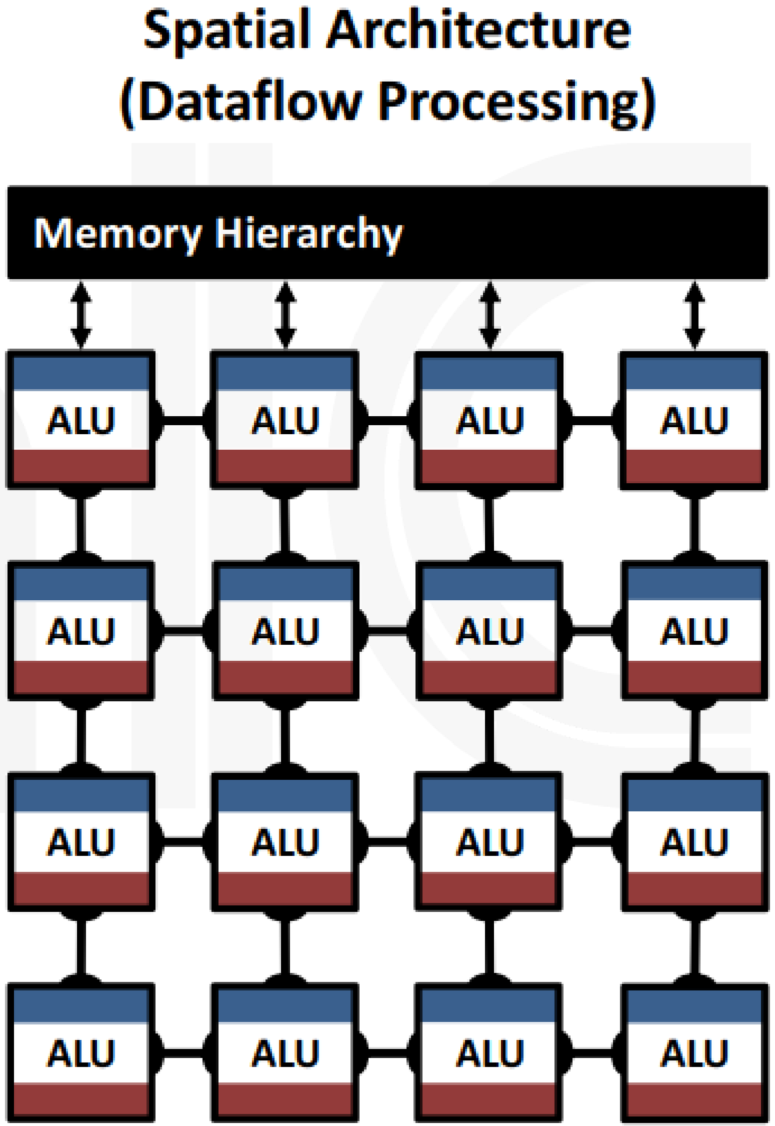
Leverage Parallelism for Higher Performance



Source: Sze, MIT

Highly-Parallel Compute Paradigms



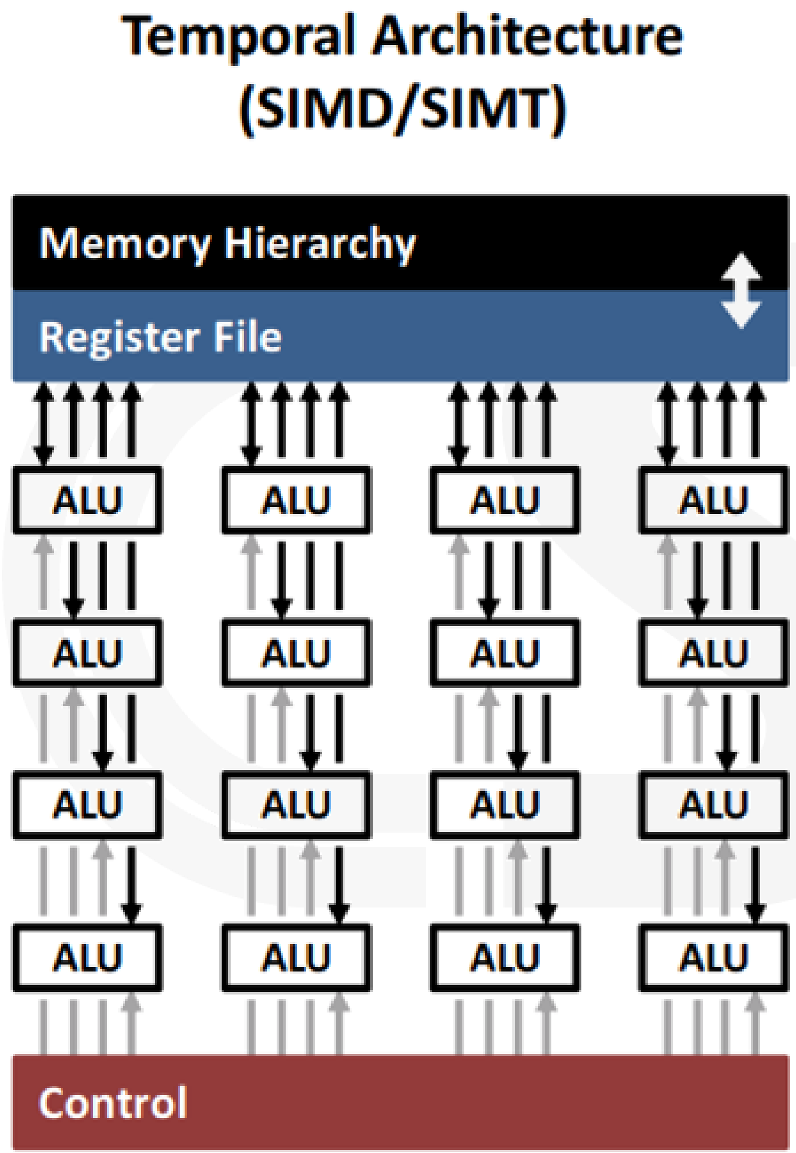


Source: Sze, MIT

Temporal Architectures

- **Mostly in CPUs and GPUs**
- **Improve parallelism through vectors (SIMD) or parallel threads (SIMT)**
- **Centralized control for a large number of ALUs**
  - Each ALU can fetch from the memory hierarchy
  - ALUs cannot communicate directly

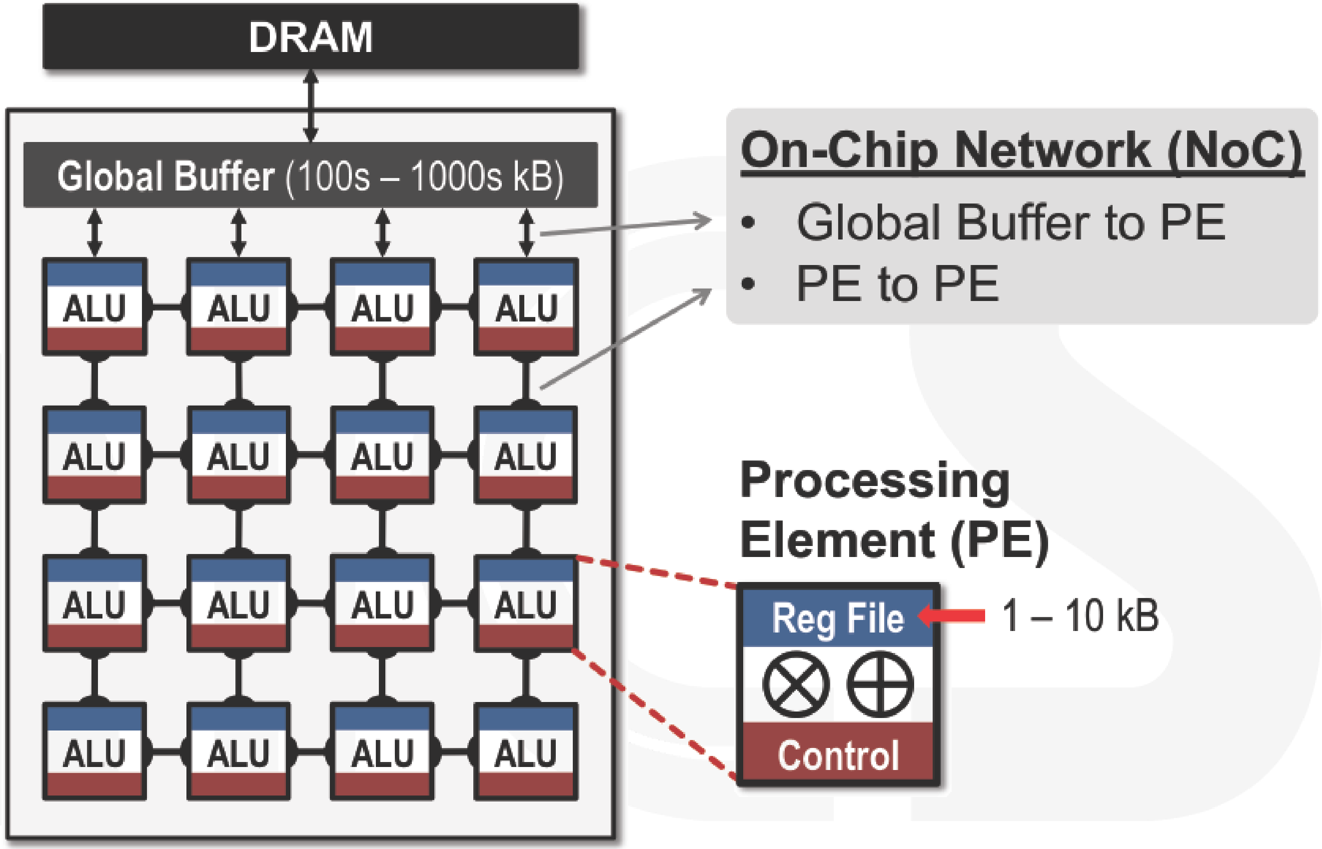
- ***Computational transformations on the kernel can reduce the number of multiplications to increase throughput***



Source: Sze, MIT

Spatial Architectures

- **Used in ASIC and FPGA designs**
- **Use dataflow processing**
  - The ALUs form a processing chain that passes data between them
  - Each ALU can have its own control logic and regfile/ scratchpad memory
- **Each ALU+local memory**
  = Processing Engine (PE)

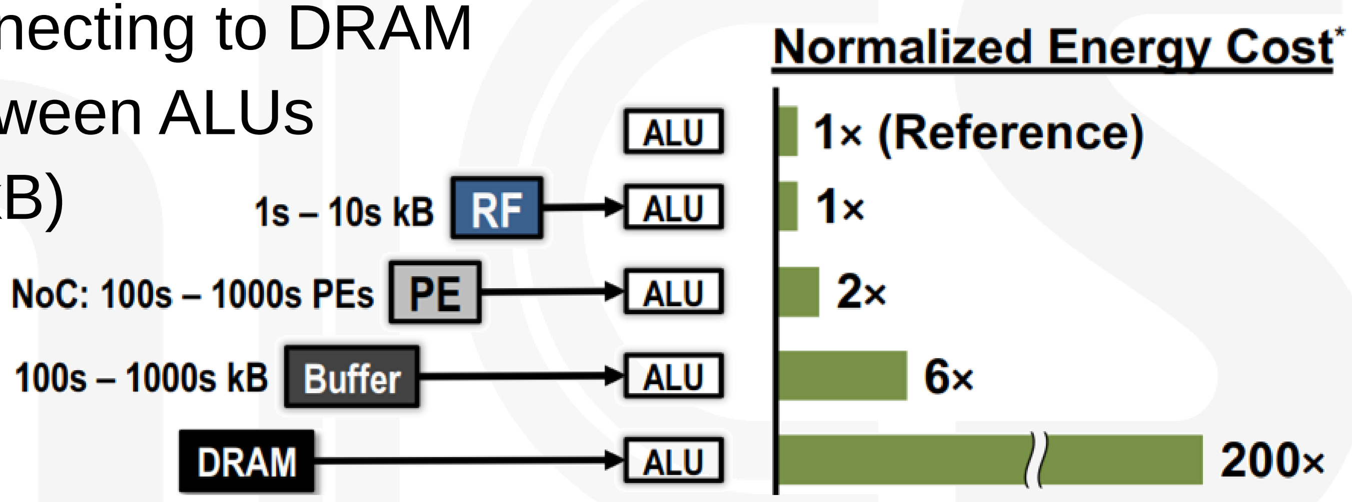


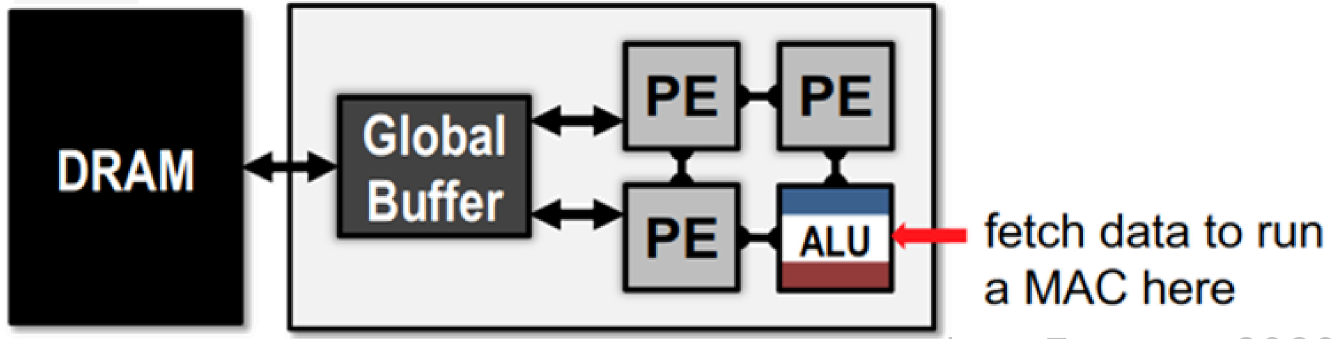
Source: Sze, MIT

- ***Can increase data reuse from low cost memories to reduce energy consumption***

Multi-Level Low-Cost Data Access

- **To reduce the energy cost of data movement, accelerators with spatial architectures can be used:**
  - Large global buffer (100s kB) connecting to DRAM
  - Inter PE network to pass data between ALUs
  - Local register file within PE (few kB)
- **Data from RF or neighbor PE lower energy than DRAM access**
  - DRAM x10^6 storage, x10^2 energy than on-chip memory
- ***Therefore, try to reuse data fetched from DRAM***



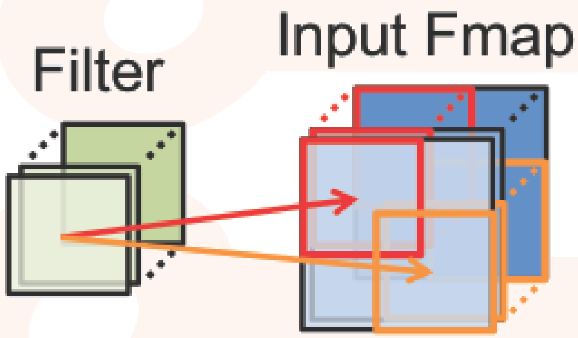


Source: Sze, MIT

Data Reuse Opportunities in DNNs

Convolutional Reuse

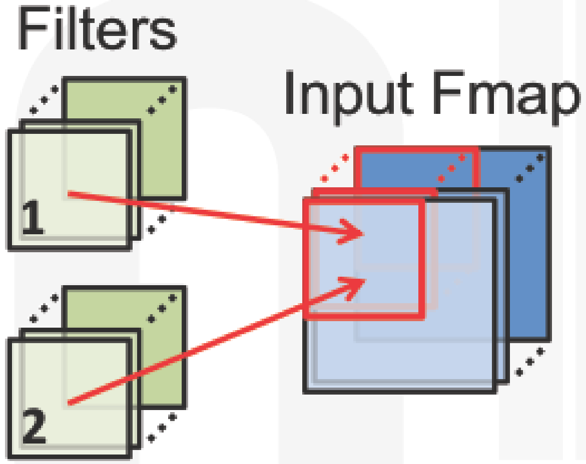
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

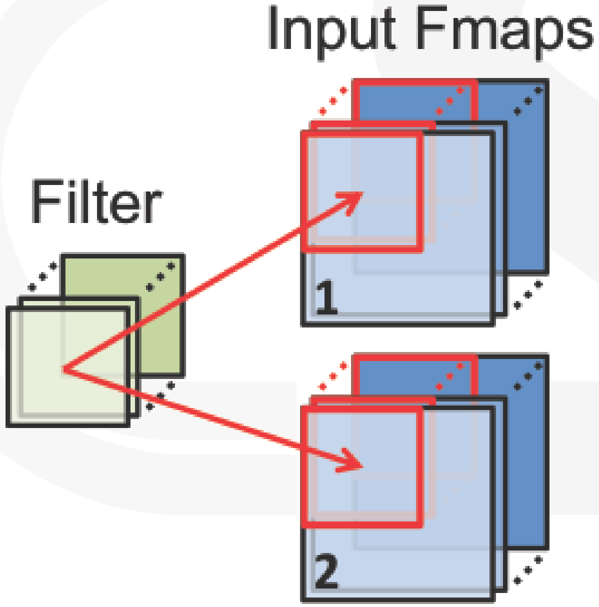
CONV and FC layers



Reuse: Activations

Filter Reuse

CONV and FC layers
(batch size > 1)



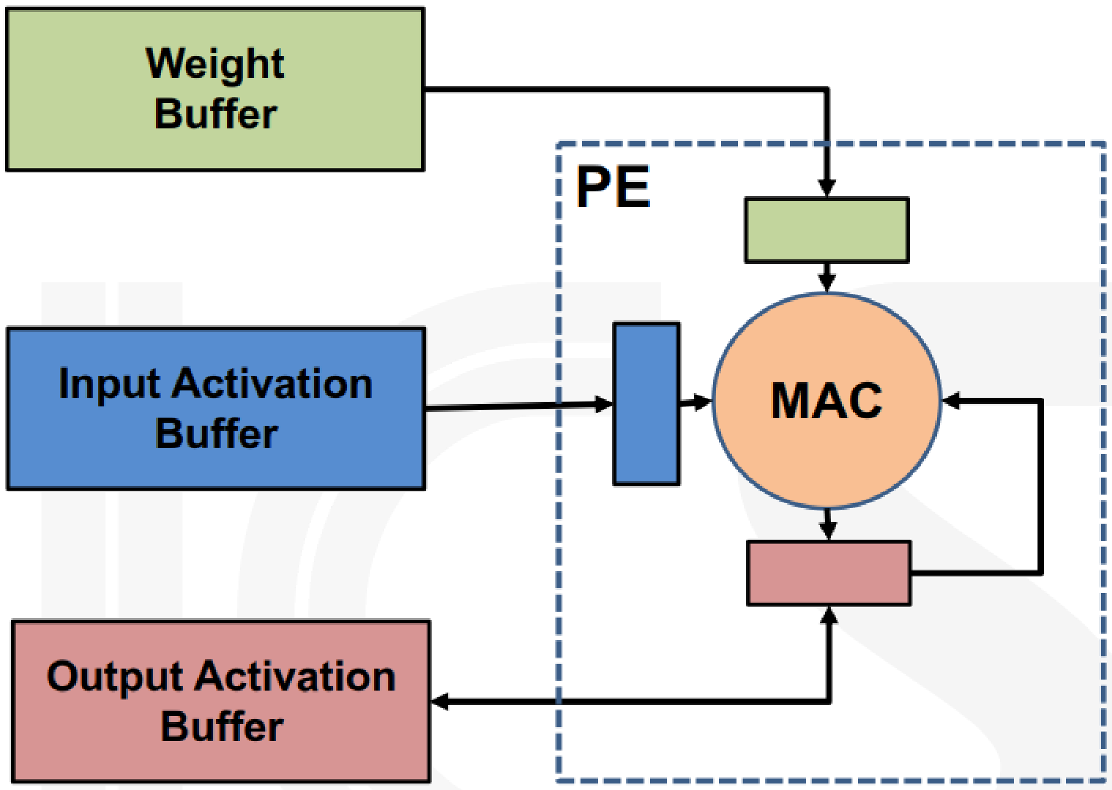
Reuse: Filter weights

Source: Sze, MIT

Dataflow Taxonomy

Dataflow Taxonomy

- **In order to manipulate data reuse in a Spatial Architecture, recently proposed dataflows can be categorized as follows:**
  - Weight Stationary (WS)
  - Output Stationary (OS)
  - Input Stationary (IS)
  - Row Stationary (RS)
  - And no-local reuse (NLR)



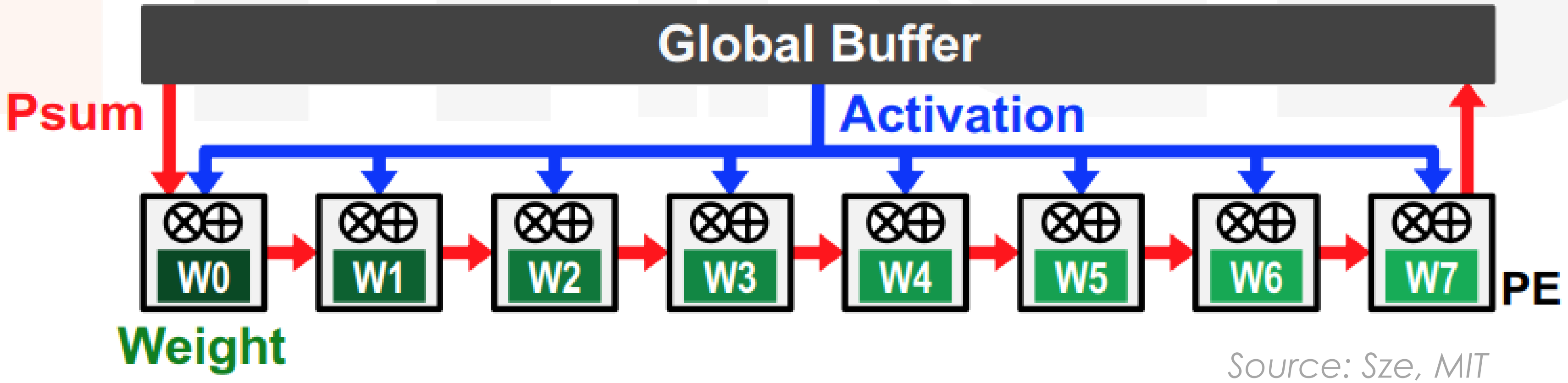
Single PE Setup

Source: Sze, MIT

- **The following section will overview these dataflow types**

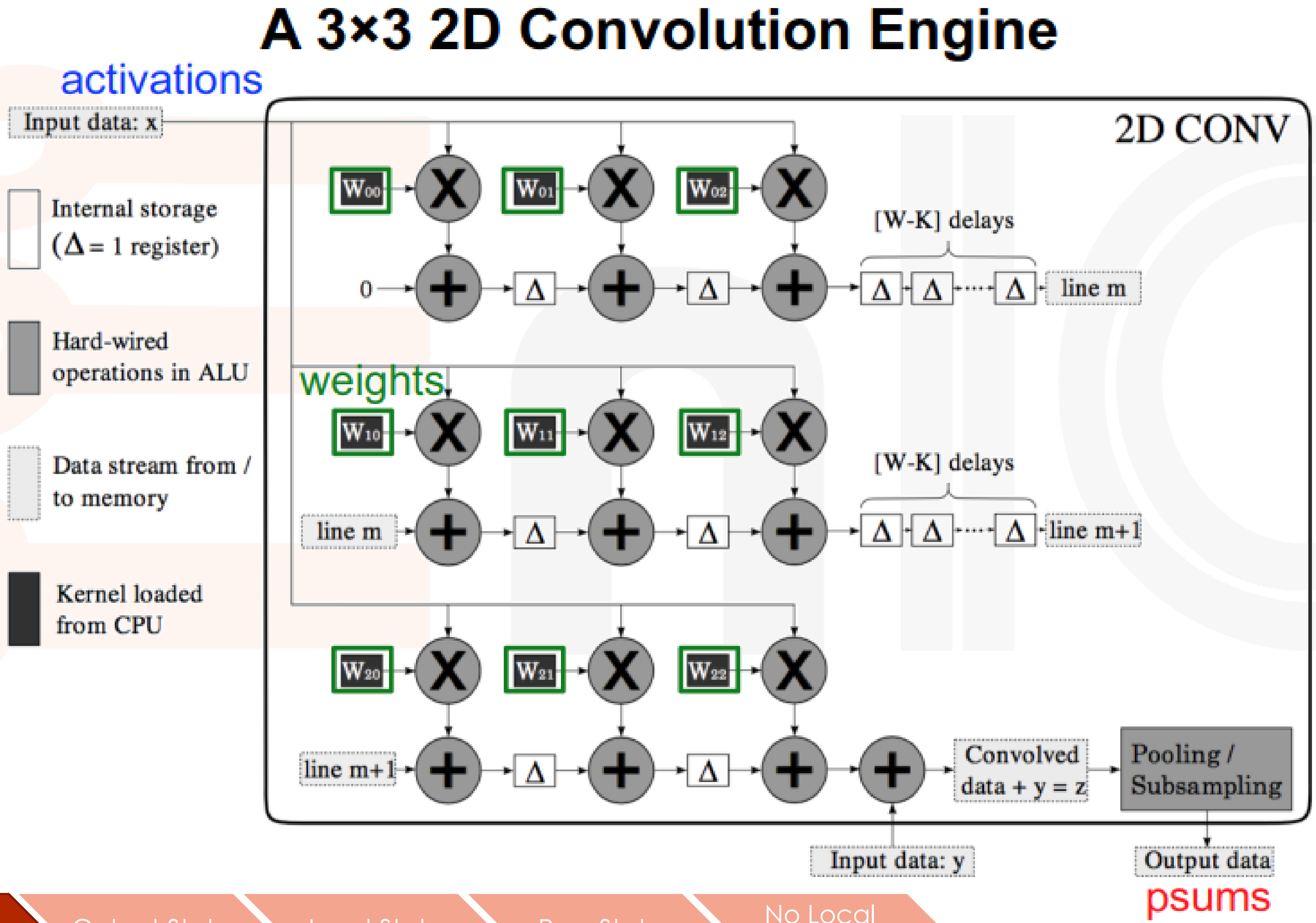
Weight Stationary (WS) Dataflow

- Minimize energy of reading weights.
  - Maximizes convolutional reuse and filter reuse of weights
- Weights are read into the RF of each PE and kept there
- Run as many MACs that use the same weight while it is in the RF
- Broadcast Input fmaps (activations) to all PEs
- Accumulate Partial sums spatially across the PE array



Source: Sze, MIT

WS Example: nn-X (NeuFlow)



[Farabet et al., ICCV 2009]

Weight Stat. | Output Stat. | Input Stat. | Row Stat. | No Local Reuse

WS Example: NVDLA (simplified)

Global Buffer

Released Sept 29, 2017

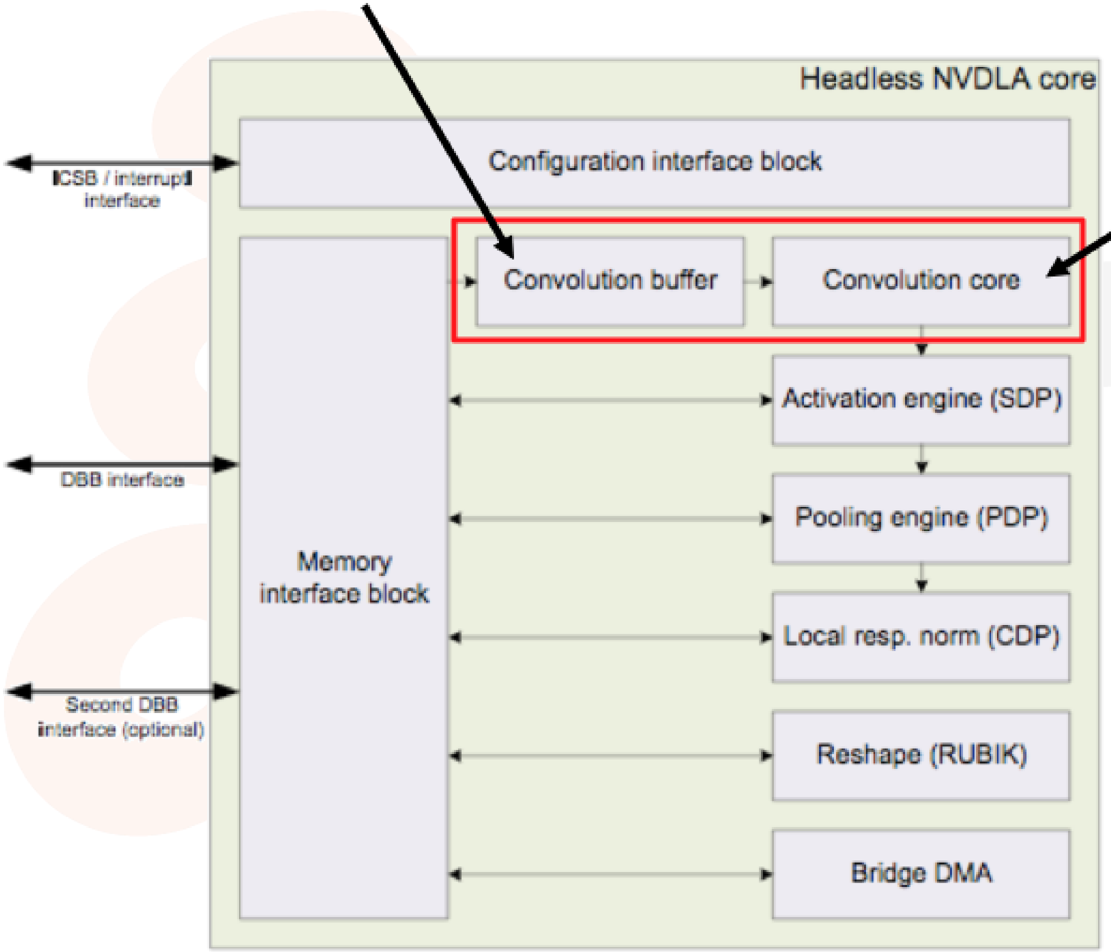
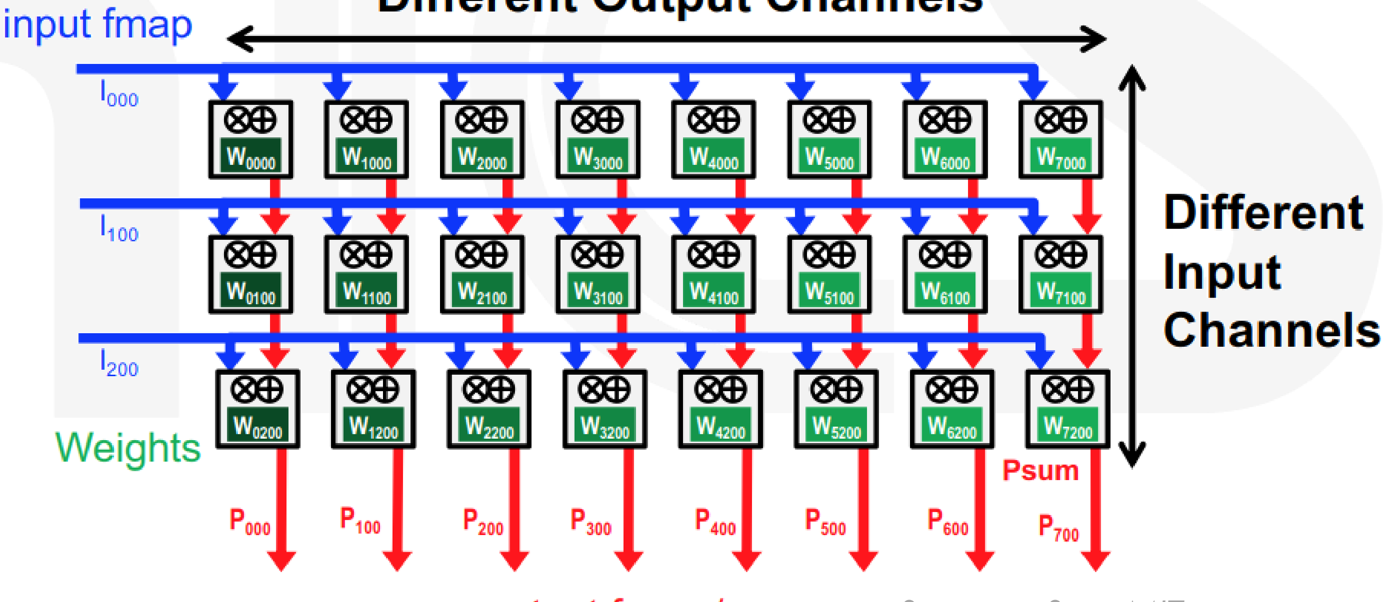


Image Source: Nvidia

PE Array: *M x C MACs*



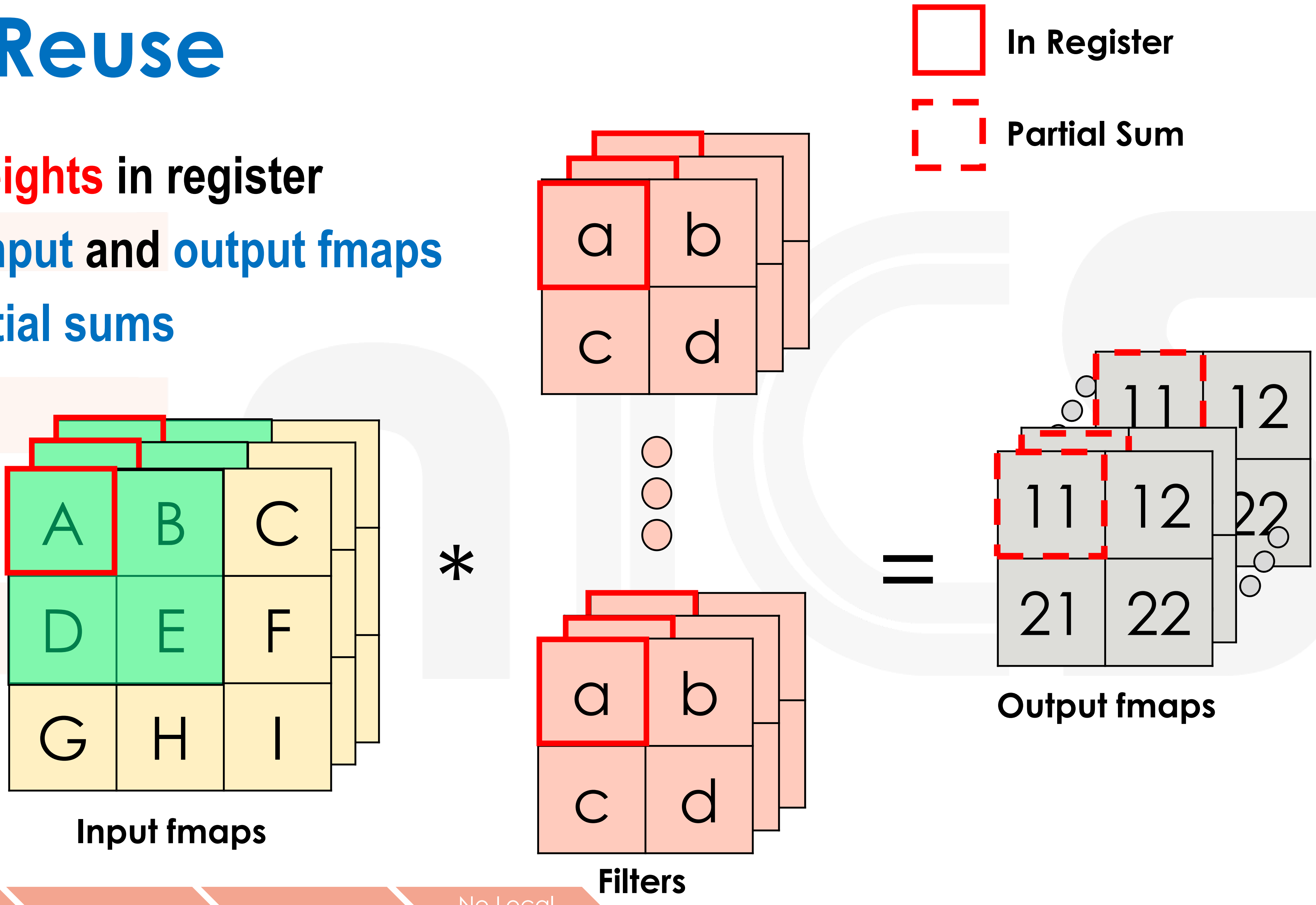
http://nvdla.org

Source: Sze, MIT

Weight Stat. | Output Stat. | Input Stat. | Row Stat. | No Local Reuse

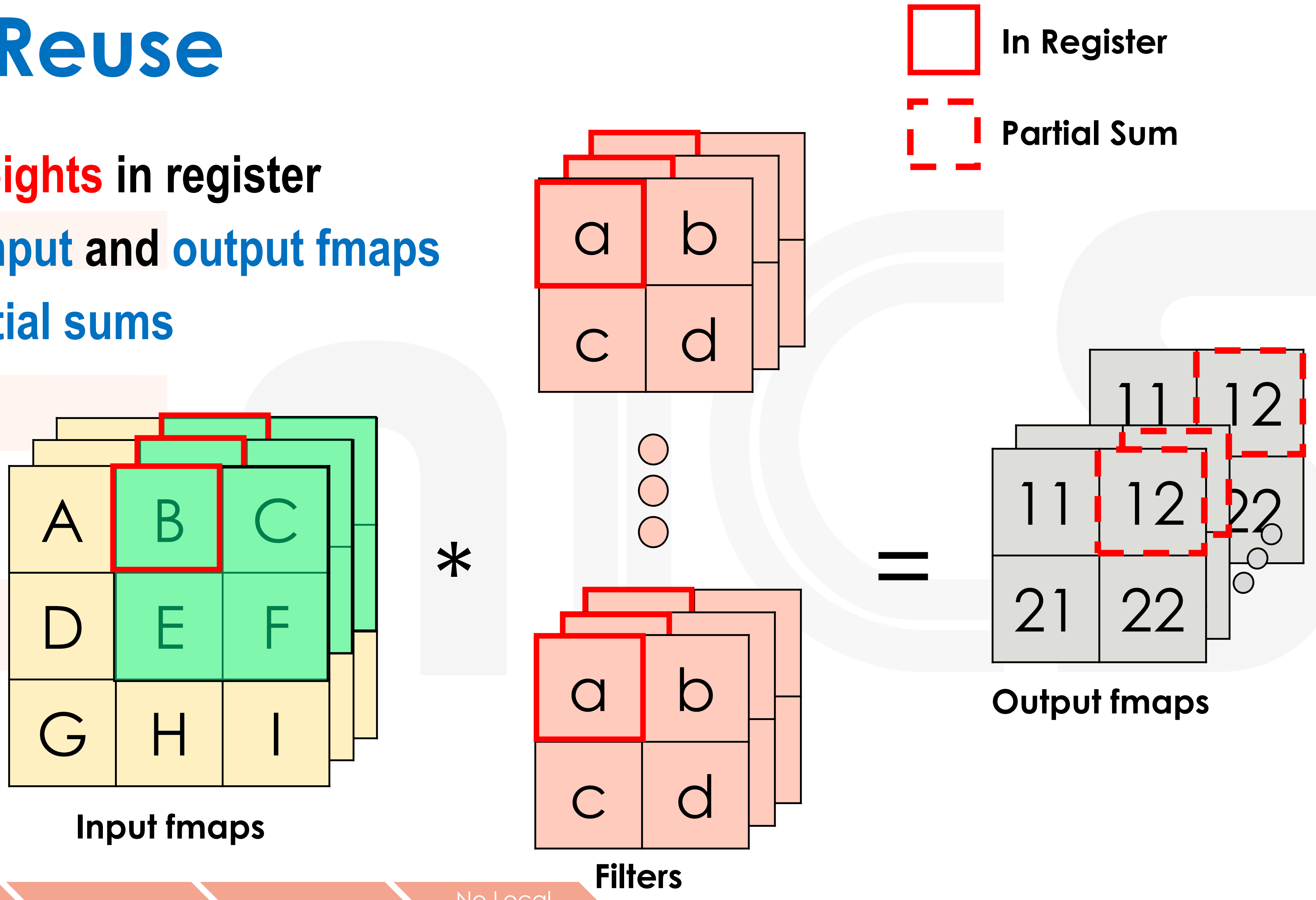
WS Data Reuse

- Keep current weights in register
- Cycle through input and output fmaps
- Accumulate partial sums



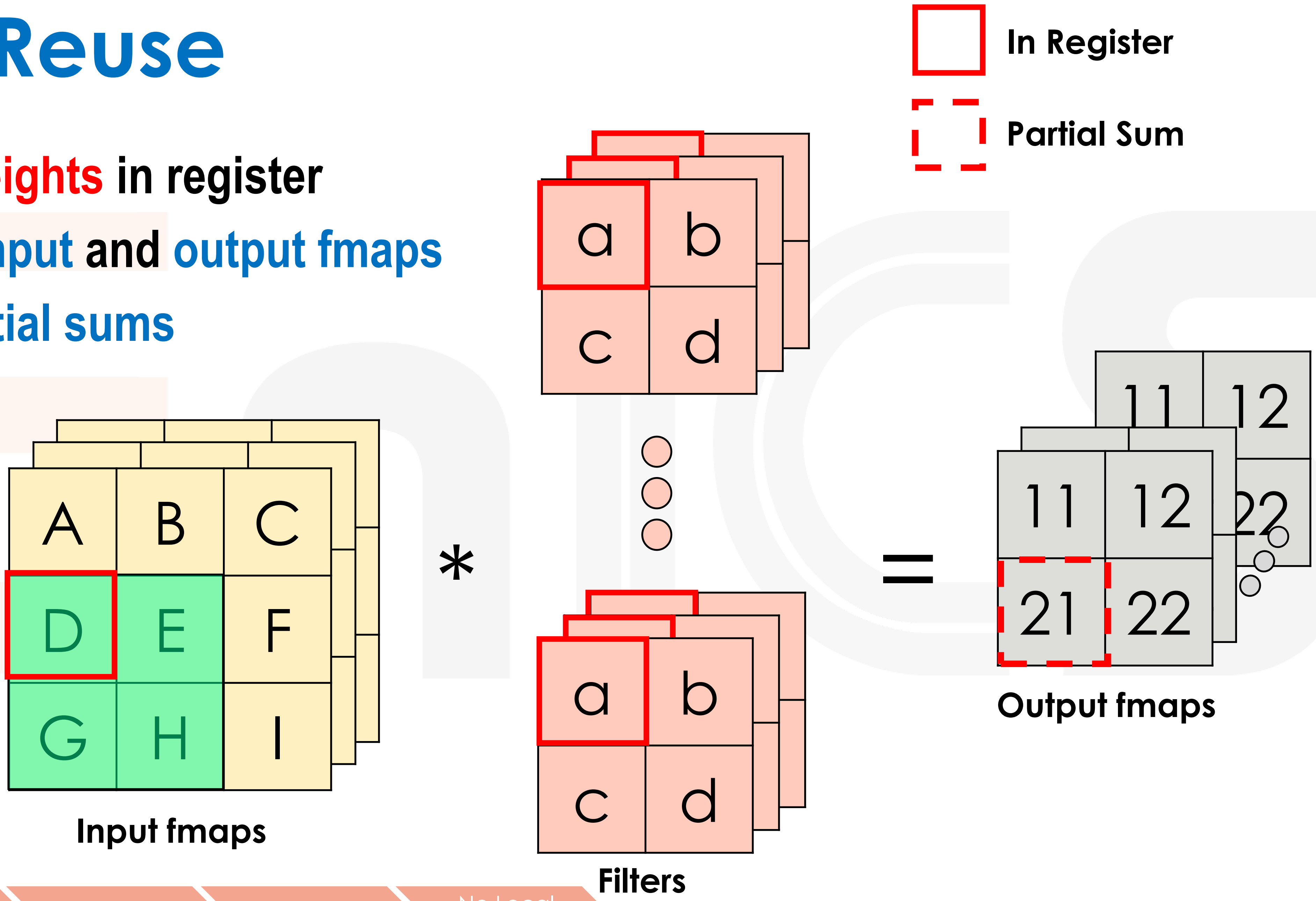
WS Data Reuse

- Keep current **weights** in register
- Cycle through **input** and **output fmaps**
- Accumulate **partial sums**



WS Data Reuse

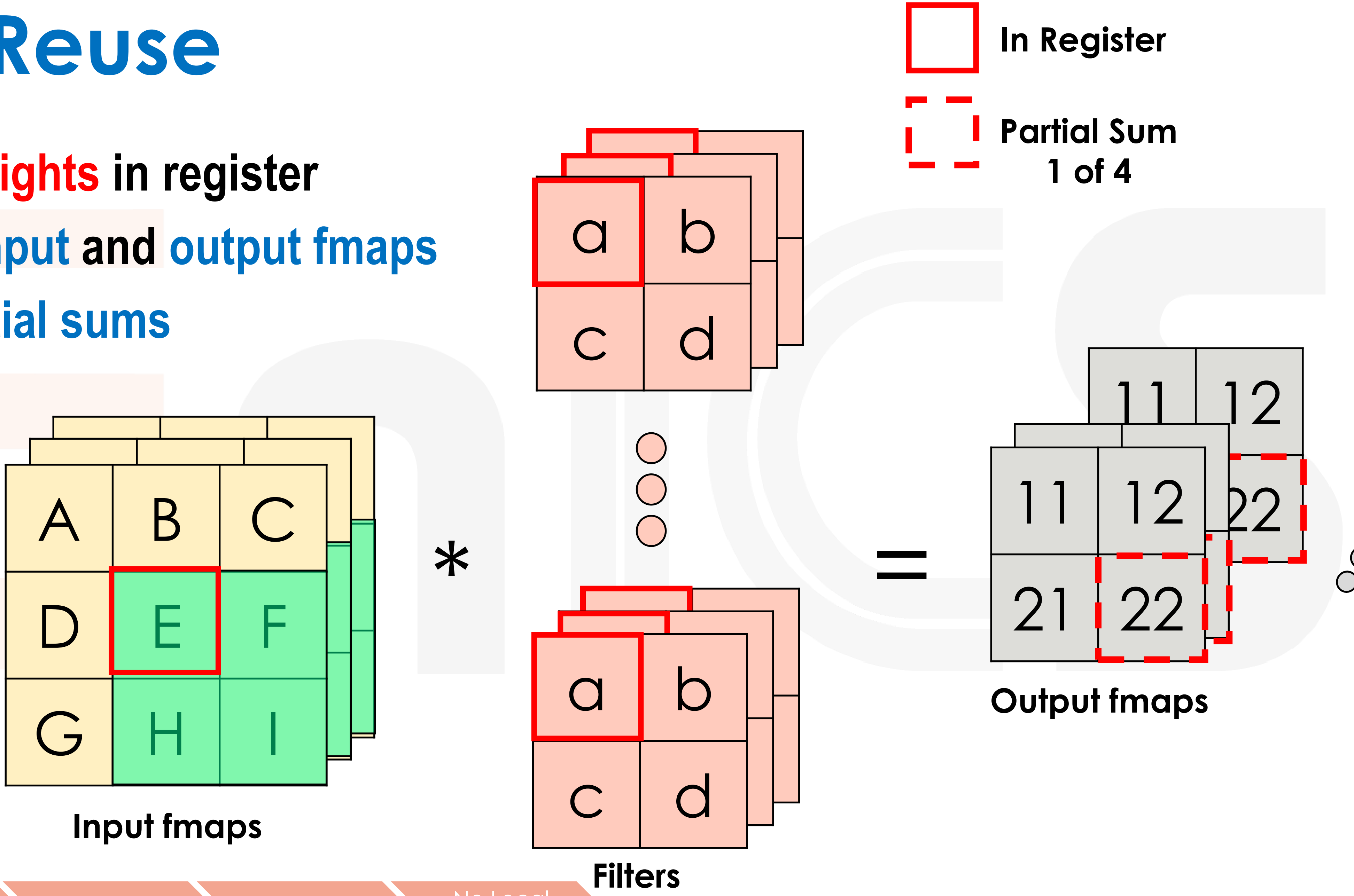
- Keep current weights in register
- Cycle through input and output fmaps
- Accumulate partial sums



Weight Stat. | Output Stat. | Input Stat. | Row Stat. | No Local Reuse

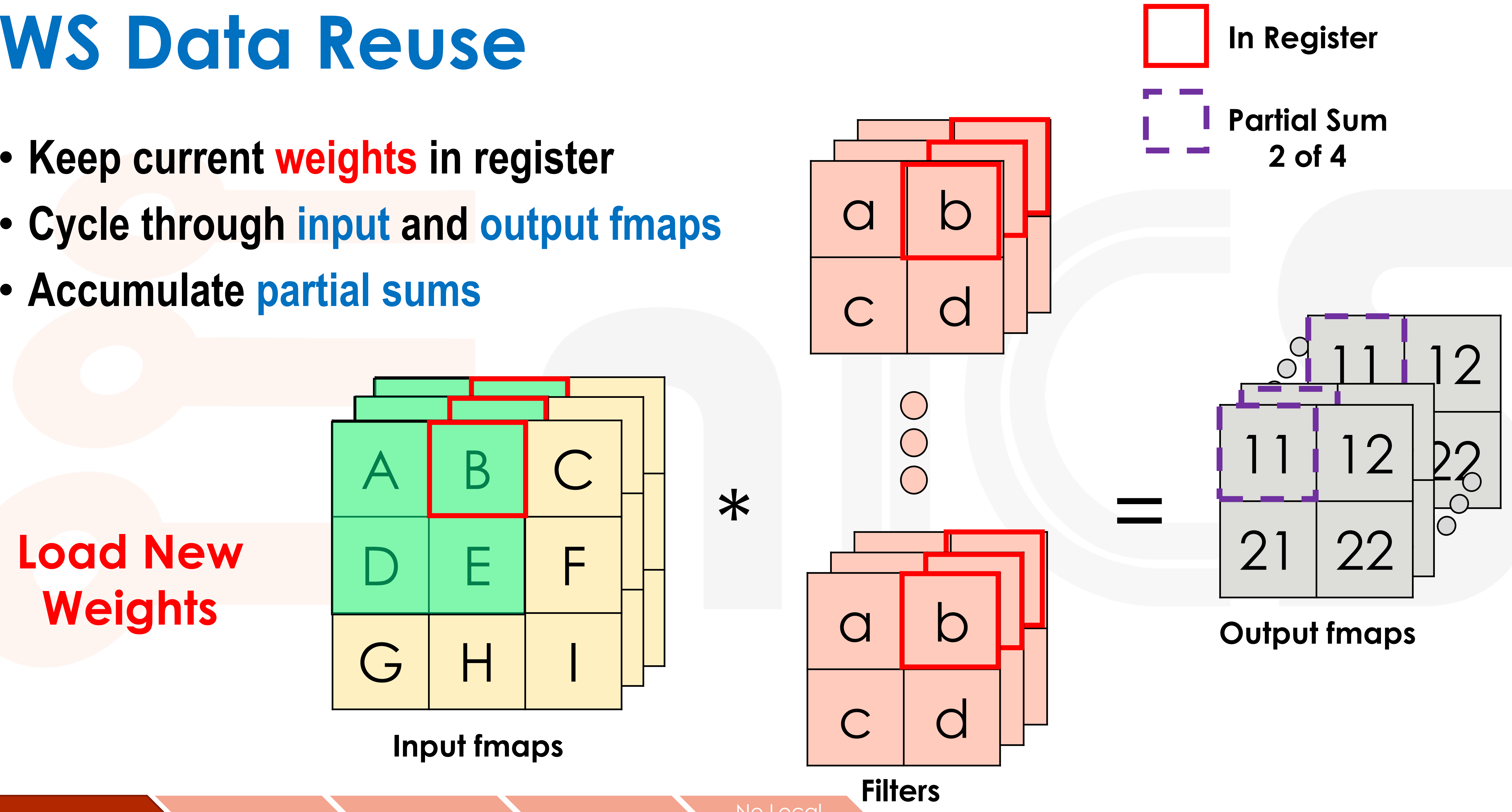
WS Data Reuse

- Keep current weights in register
- Cycle through input and output fmaps
- Accumulate partial sums



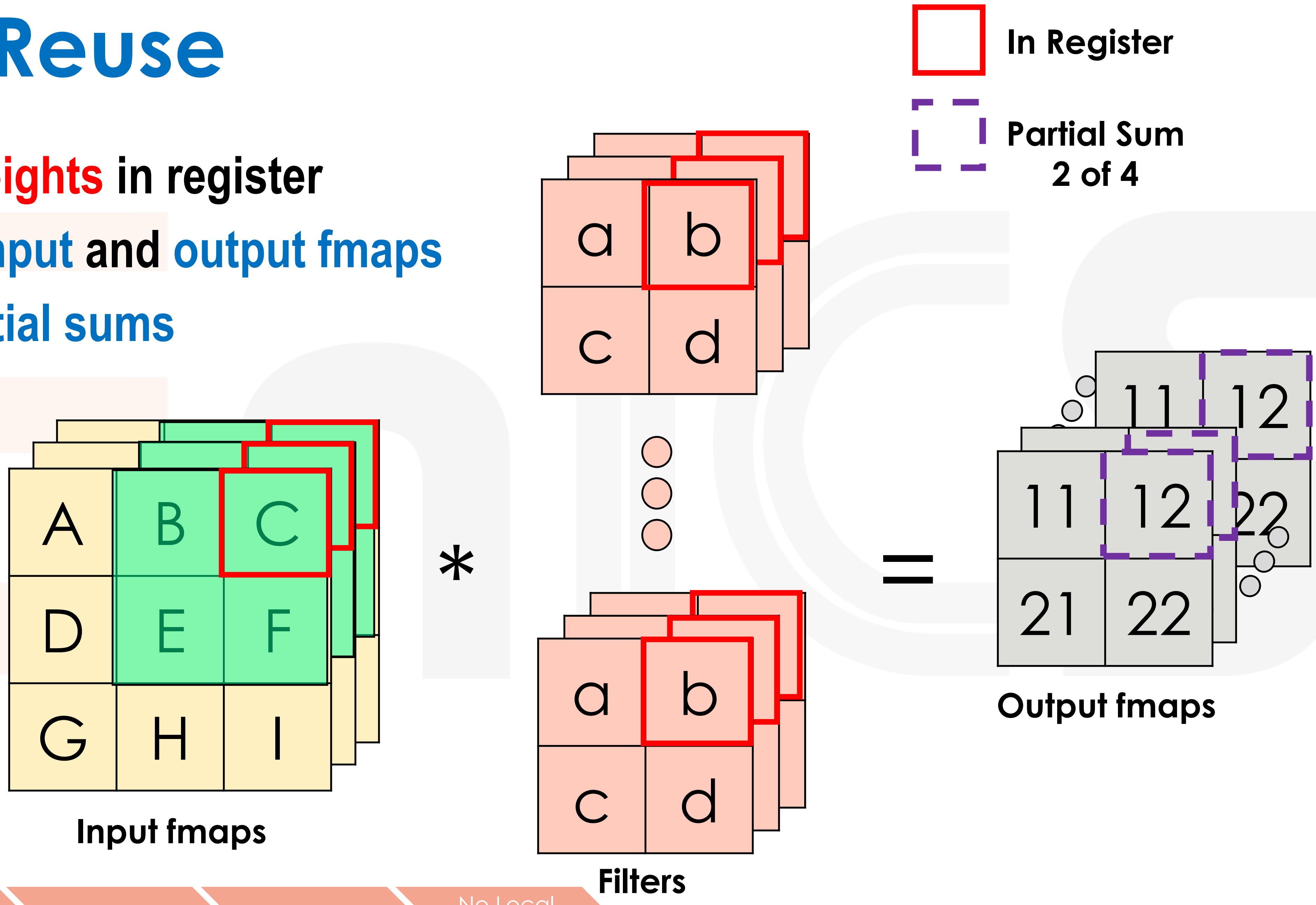
WS Data Reuse

- Keep current **weights** in register
- Cycle through **input** and **output fmaps**
- Accumulate **partial sums**



WS Data Reuse

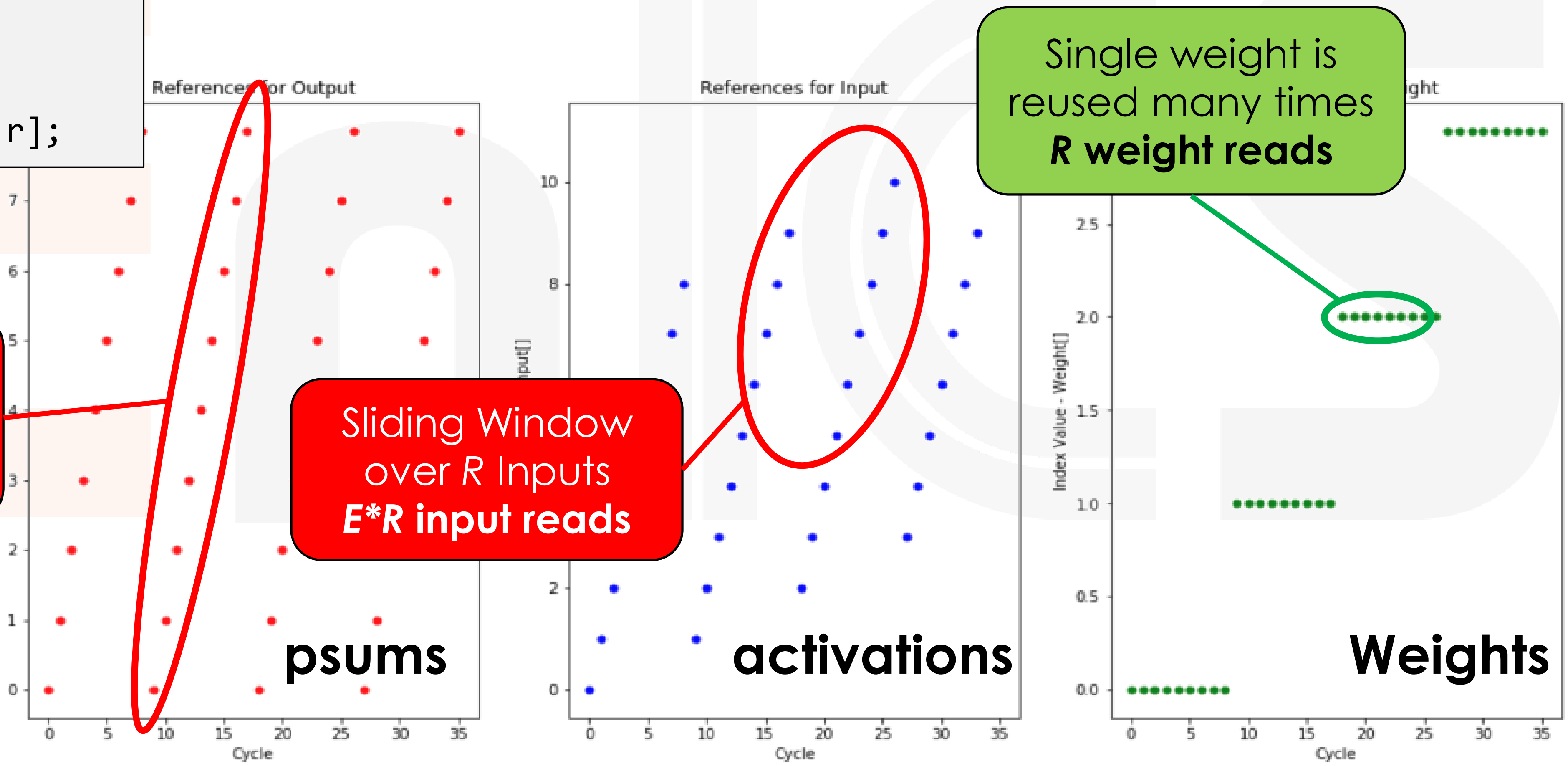
- Keep current weights in register
- Cycle through input and output fmaps
- Accumulate partial sums



Weight Stationary – Reference Pattern

In these plots:
H=12
R=4
E=9

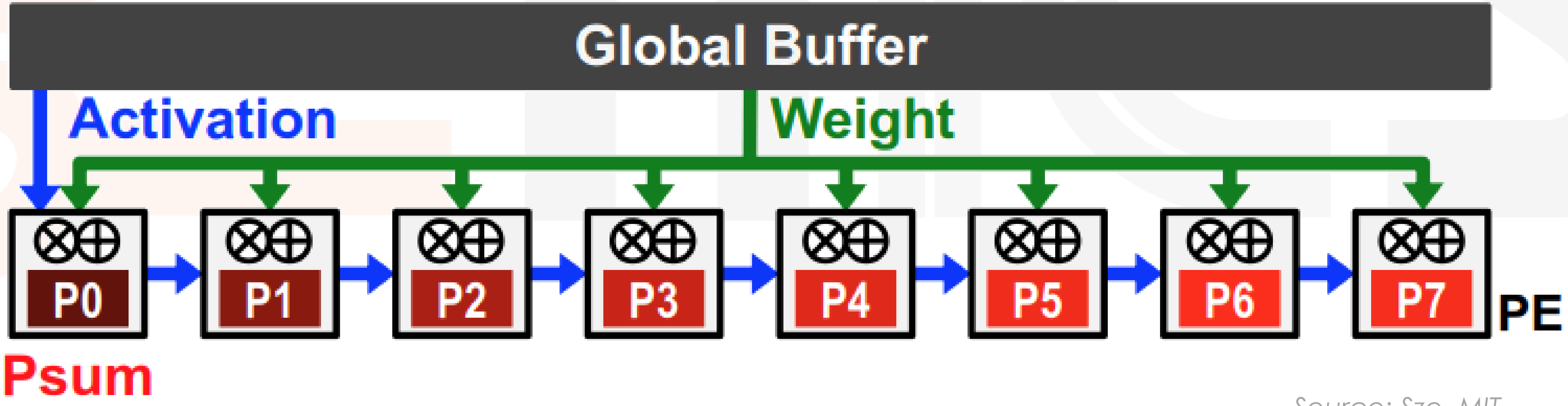
```
int I[H]; // Input activations
int W[R]; // Filter weights
int O[E]; // Output activations

for (e = 0; e < E; e++)
   for (r = 0; r < R; r++)
      O[e] += I[e+r] * W[r];
```



Source: Sze, MIT

Output Stationary (OS) Dataflow

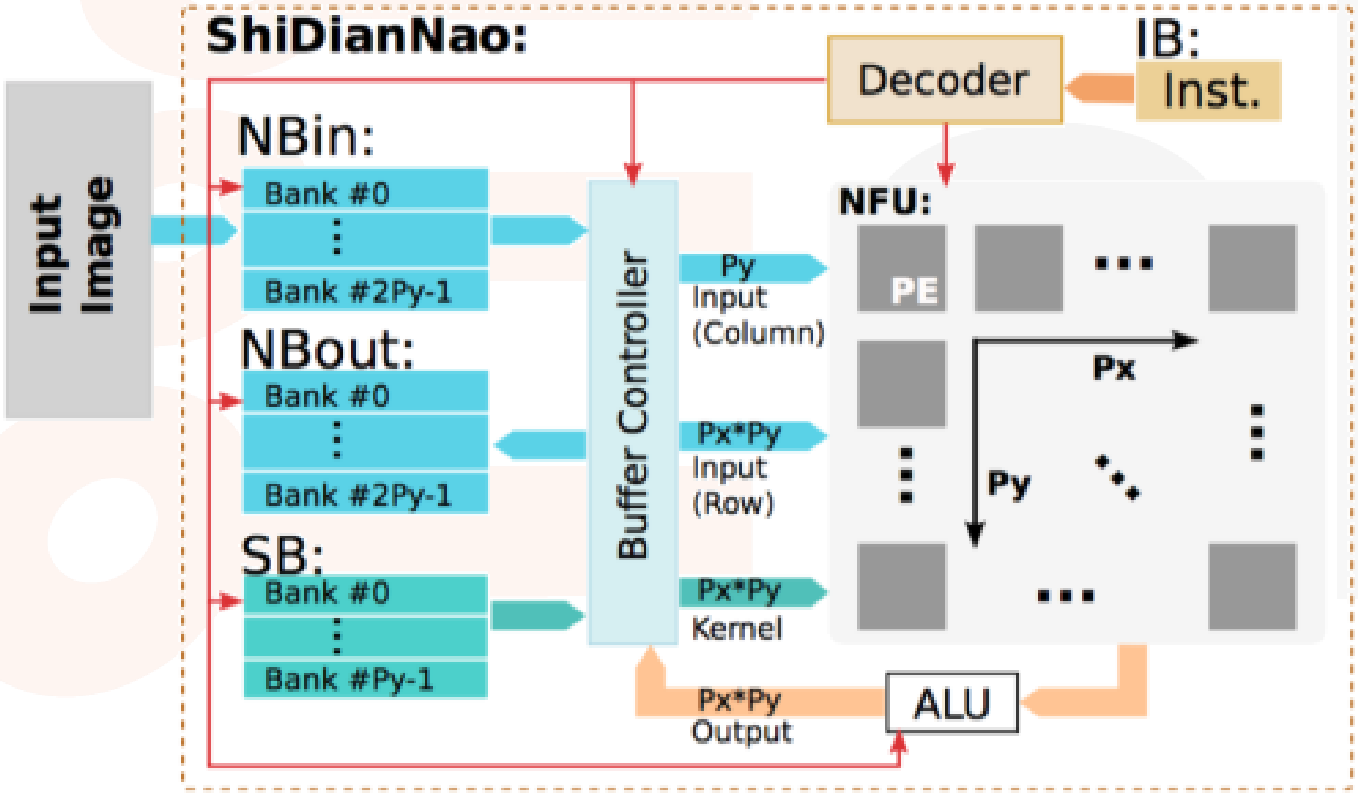
- **Minimize the energy of reading and writing partial sums.**
  - Maximize local accumulation of ofmaps Partial sums.
- **Broadcast filter weights to all PEs**
- **Stream ifmaps (activations) across the PE array for reuse**



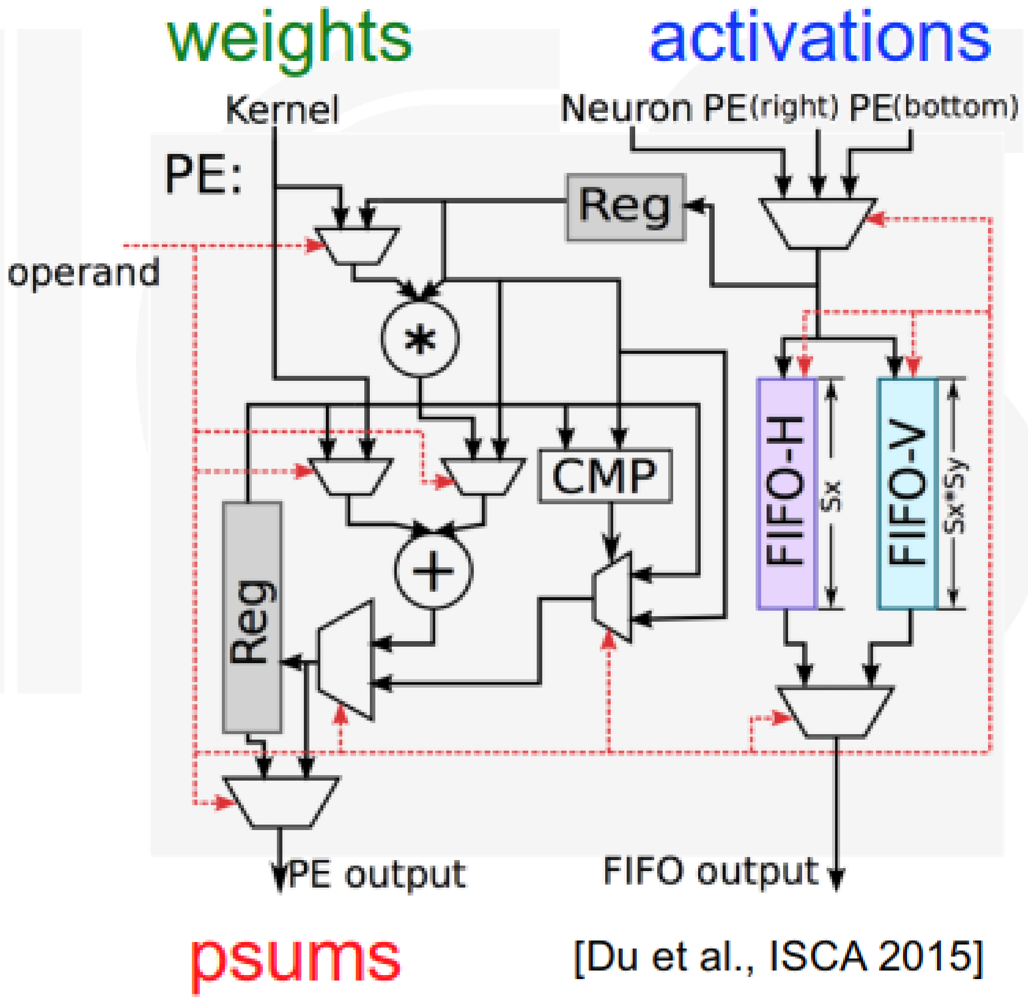
Source: Sze, MIT

OS Example: ShiDianNao

Top-Level Architecture



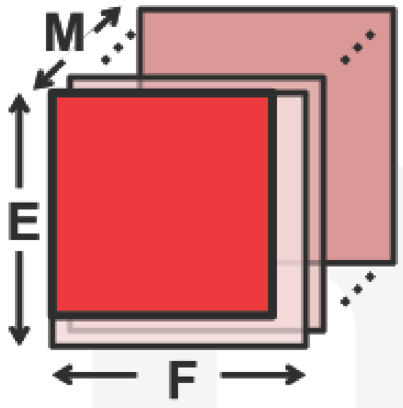
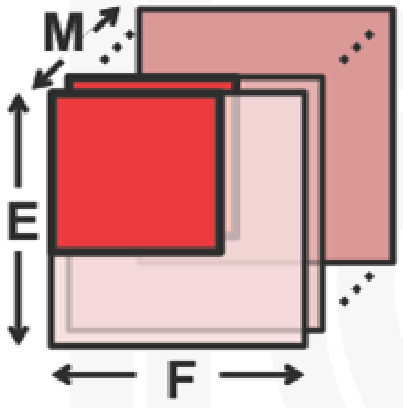
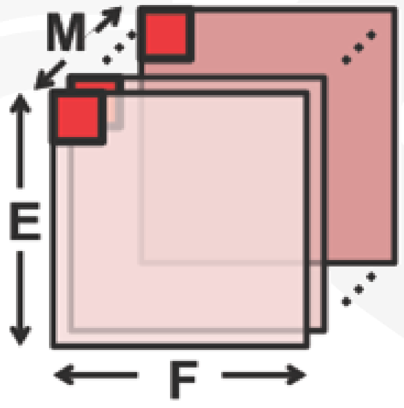
PE Architecture



[Du et al., ISCA 2015]

Weight Stat. | Output Stat. | Input Stat. | Row Stat. | No Local Reuse

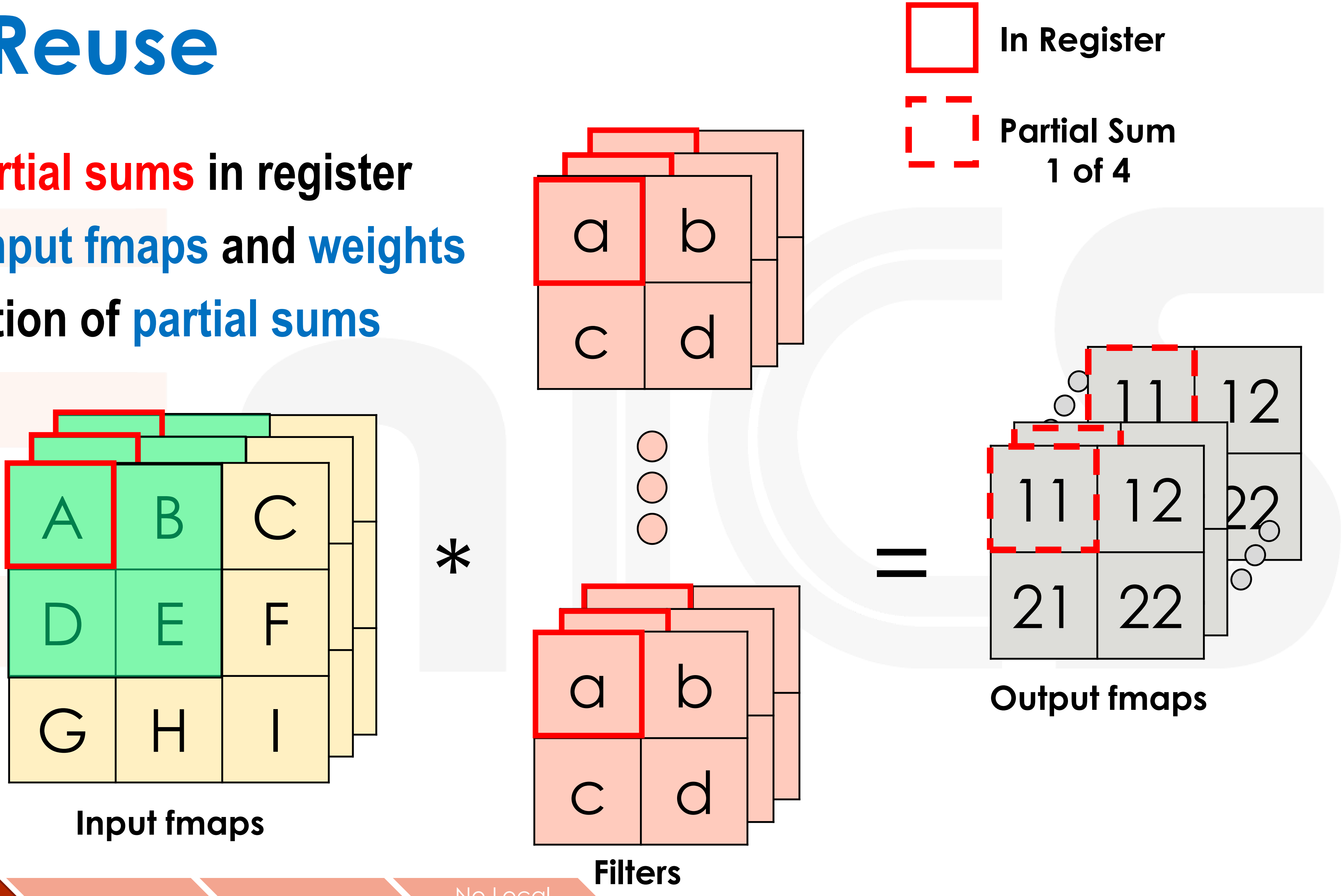
Variants of Output Stationary

	OS_A	OS_B	OS_C
Parallel Output Region			
# Output Channels	Single	Multiple	Multiple
# Output Activations	Multiple	Multiple	Single
Notes	Targeting **CONV** layers		Targeting **FC** layers

Source: Sze, MIT

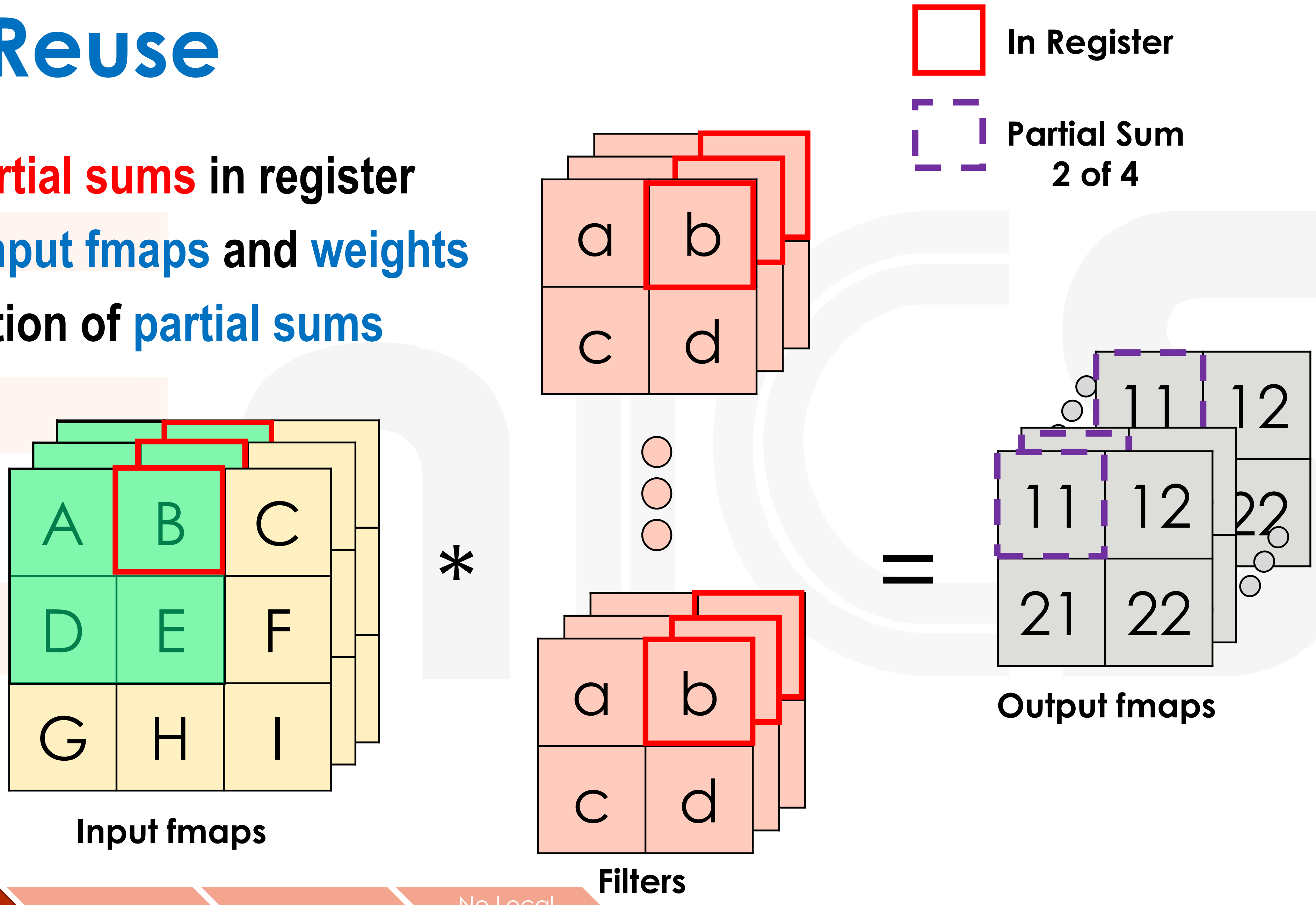
OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums



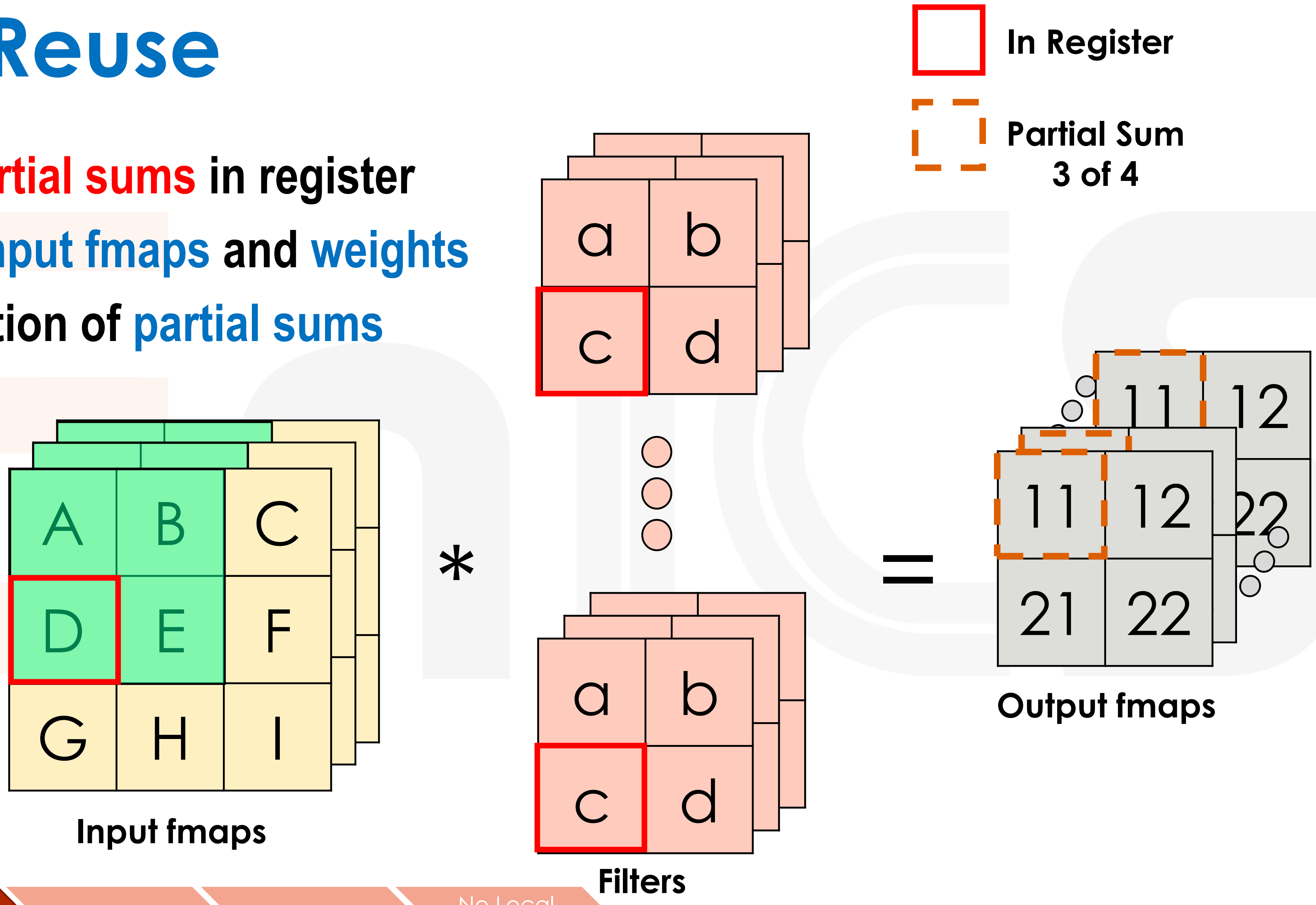
OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums



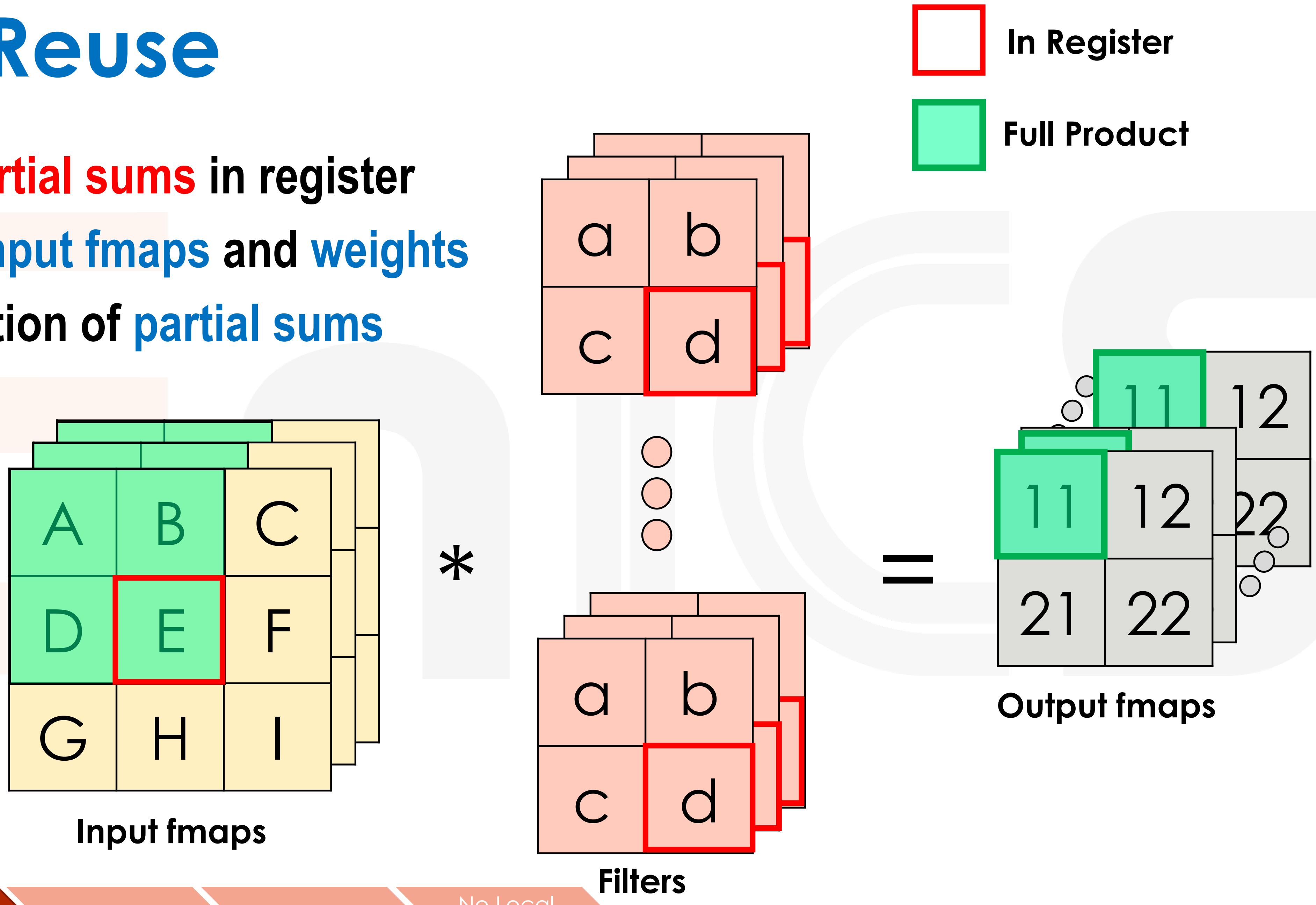
OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums



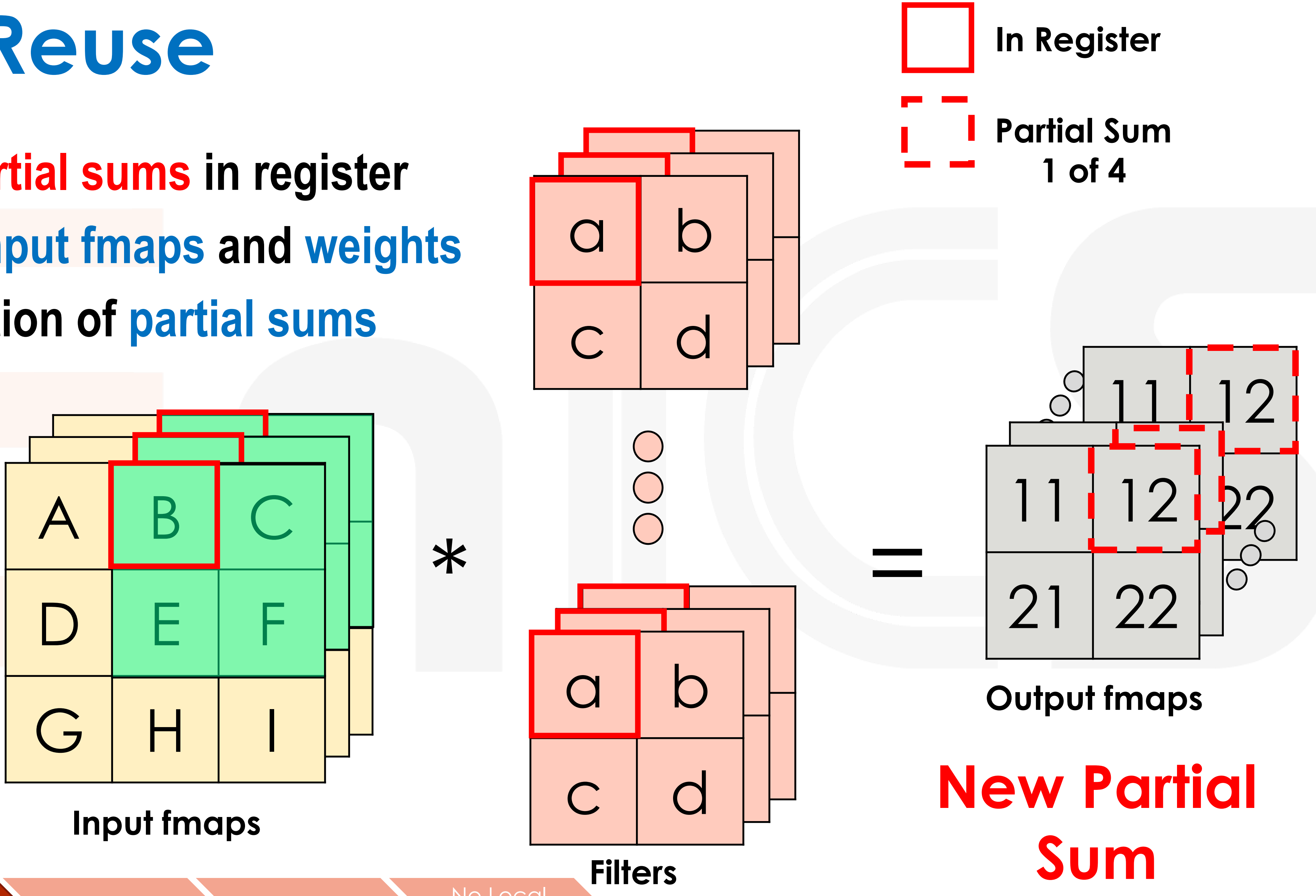
OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums



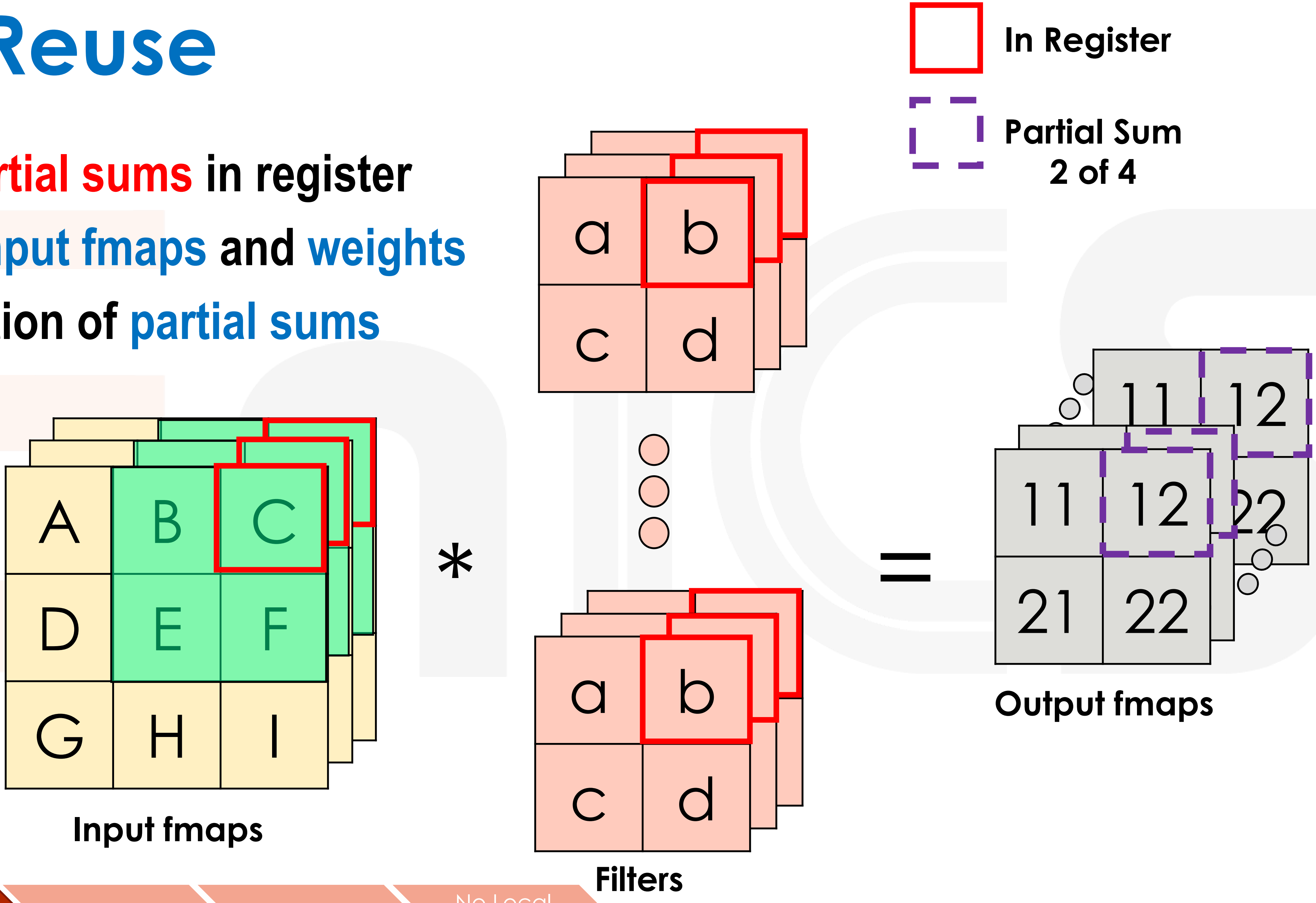
OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums



OS Data Reuse

- Keep current partial sums in register
- Cycle through input fmaps and weights
- Local accumulation of partial sums

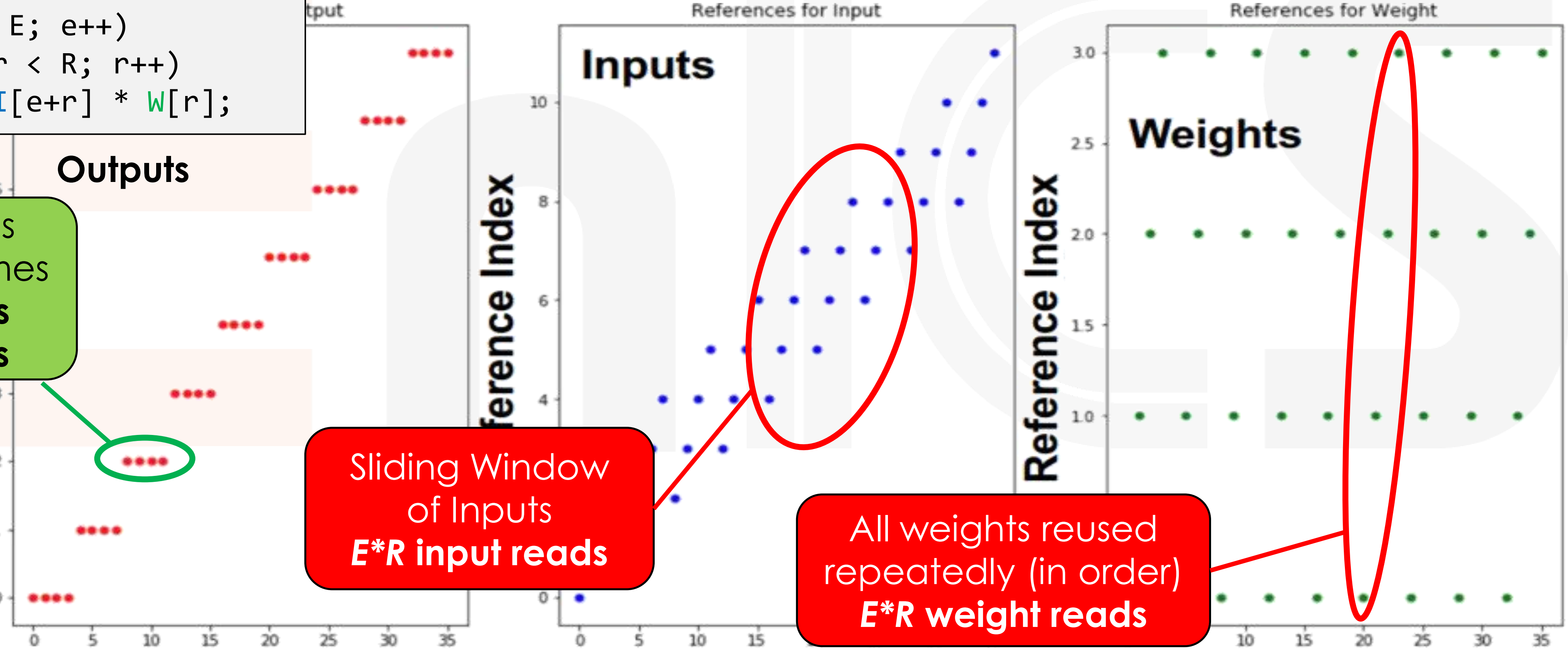


Weight Stat. | Output Stat. | Input Stat. | Row Stat. | No Local Reuse

Output Stationary – Reference Pattern

In these plots:
H=12
R=4
E=9

```
int I[H]; // Input activations
int W[R]; // Filter weights
int O[E]; // Output activations

for (e = 0; e < E; e++)
   for (r = 0; r < R; r++)
      O[e] += I[e+r] * W[r];
```



Single output is reused many times
***0* output reads**
***E* output writes**

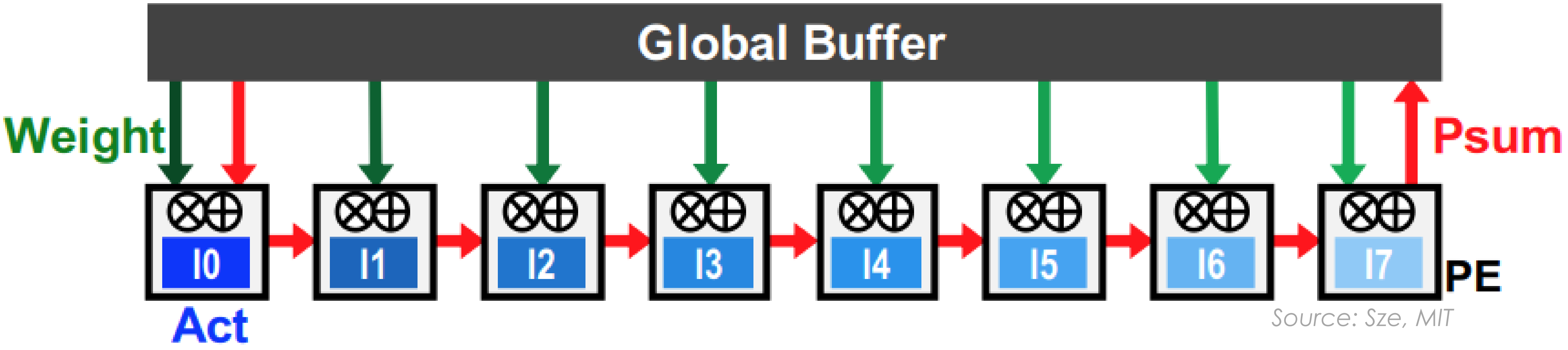
Sliding Window of Inputs
***E*R* input reads**

All weights reused repeatedly (in order)
***E*R* weight reads**

Source: Sze, MIT

Input Stationary (IS) Dataflow

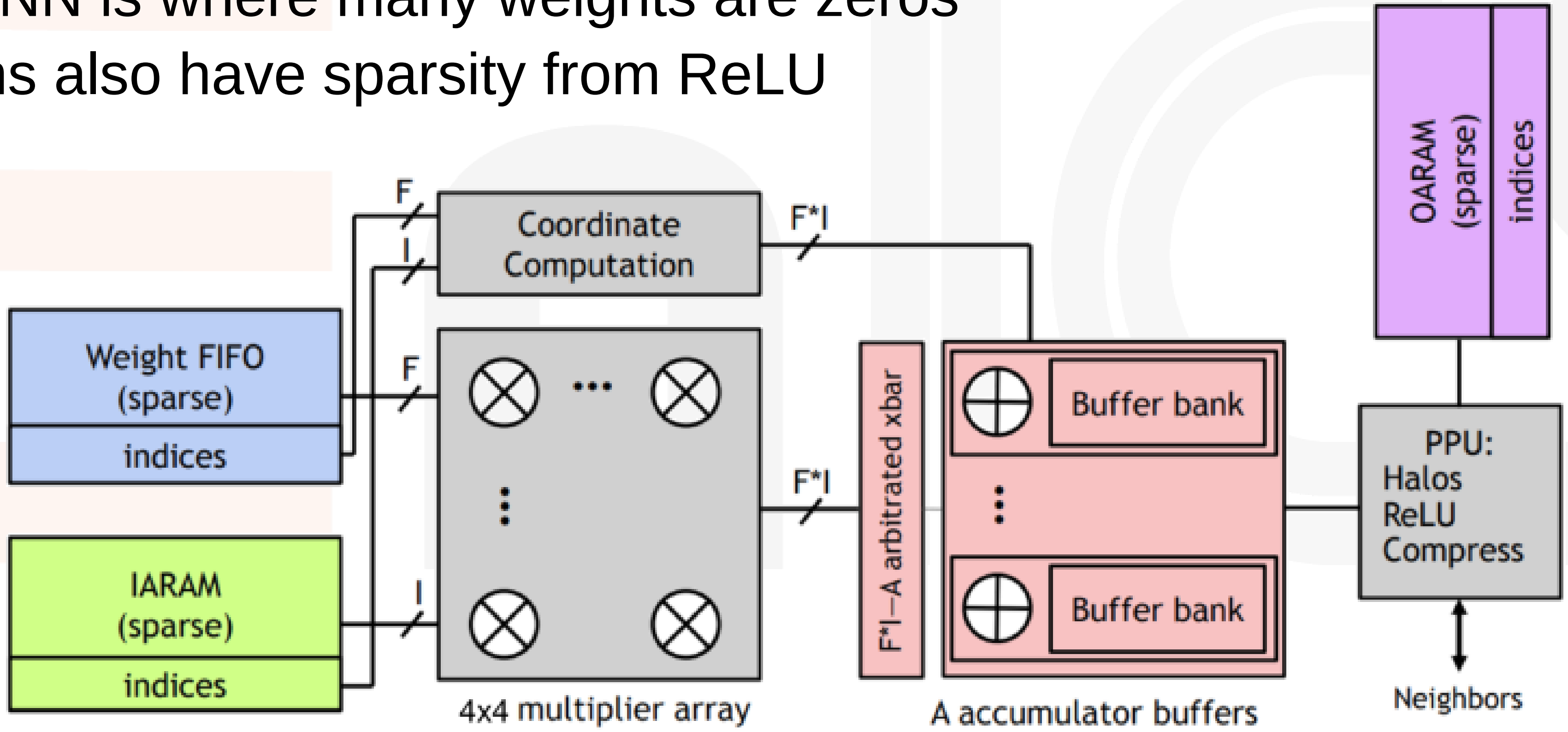
- **Minimize activation read energy consumption**
  - maximize **convolutional** and **fmap reuse** of activations
- **Unicast weights and accumulate psums spatially across the PE array.**



Source: Sze, MIT

IS Example: Sparse CNN (SCNN)

- **Used for sparse CNNs**
  - Sparse CNN is where many weights are zeros
  - Activations also have sparsity from ReLU

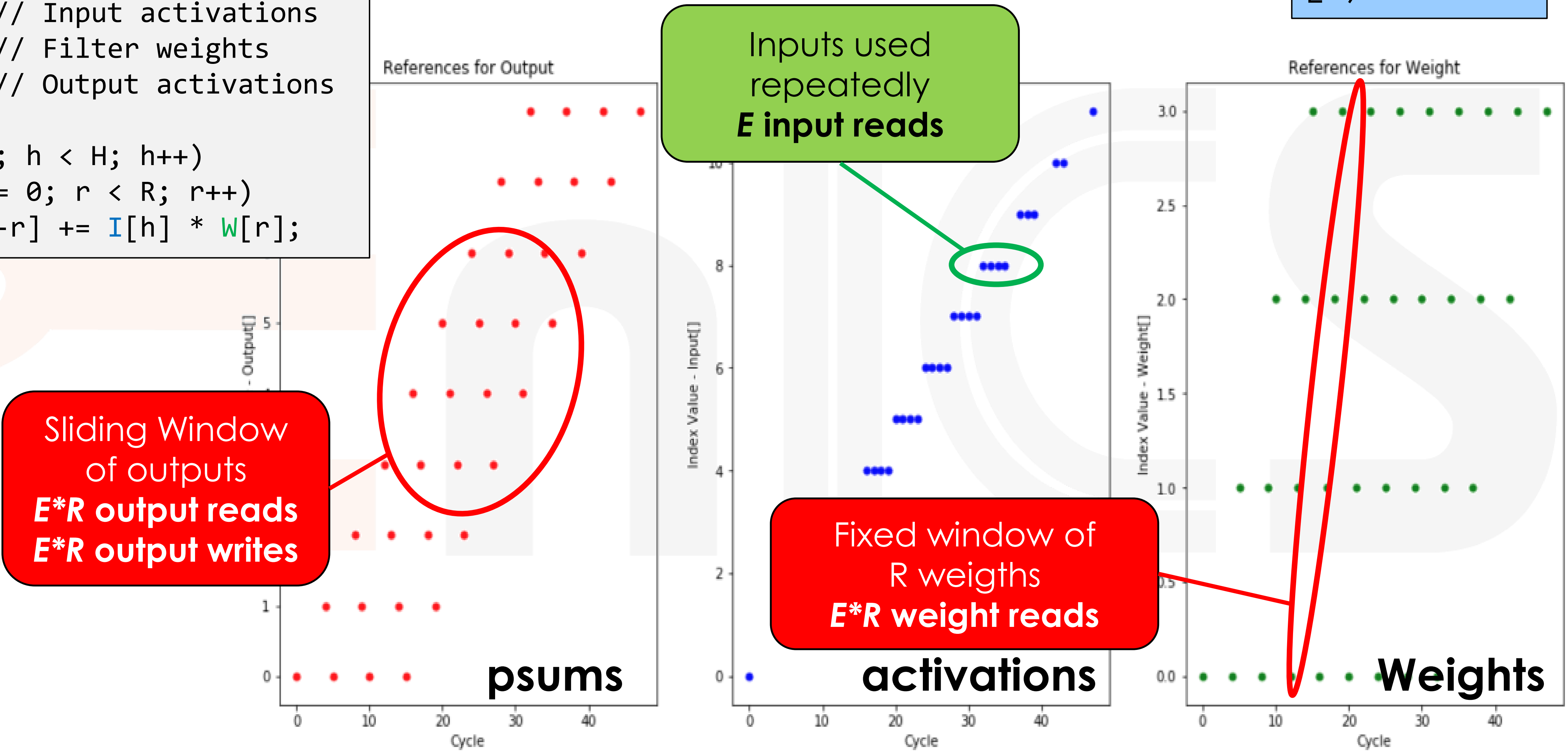


[Parashar et al., ISCA 2017]

Input Stationary – Reference Pattern

In these plots:
H=12
R=4
E=9

```
int I[H]; // Input activations
int W[R]; // Filter weights
int O[E]; // Output activations

for (h = 0; h < H; h++)
   for (r = 0; r < R; r++)
      O[h-r] += I[h] * W[r];
```



Source: Sze, MIT

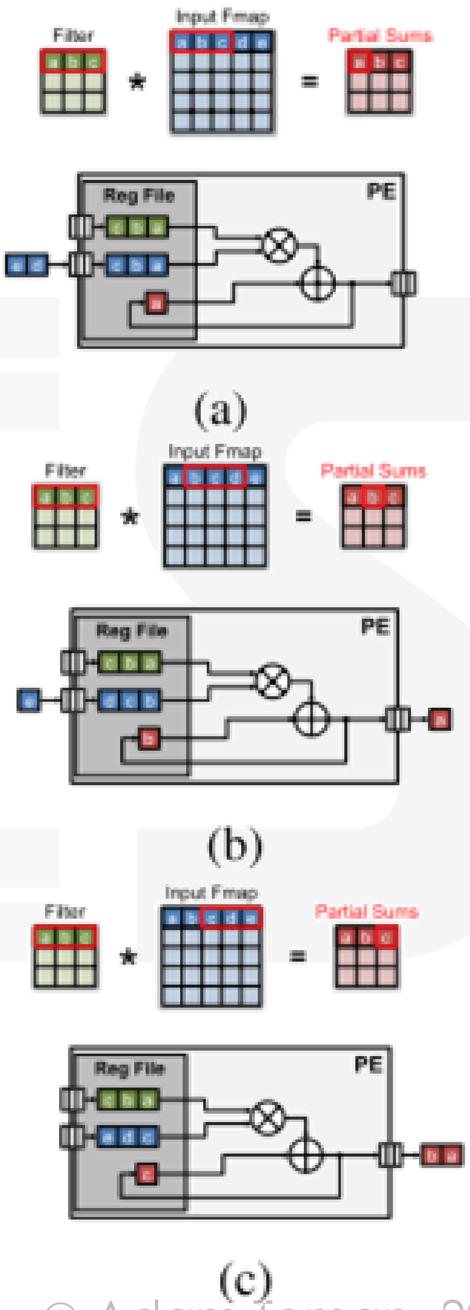
Minimum Costs

	OS	WS	IS	Min
MACs	E*R	E*R	E*R	E*R
Weight Reads	E*R	R	E*R	R
Input Reads	E*R	E*R	E	E
Output Reads	0	E*R	E*R	0
Output Writes	E	E*R	E*R	E

Assume: W ~= E

Source: Sze, MIT

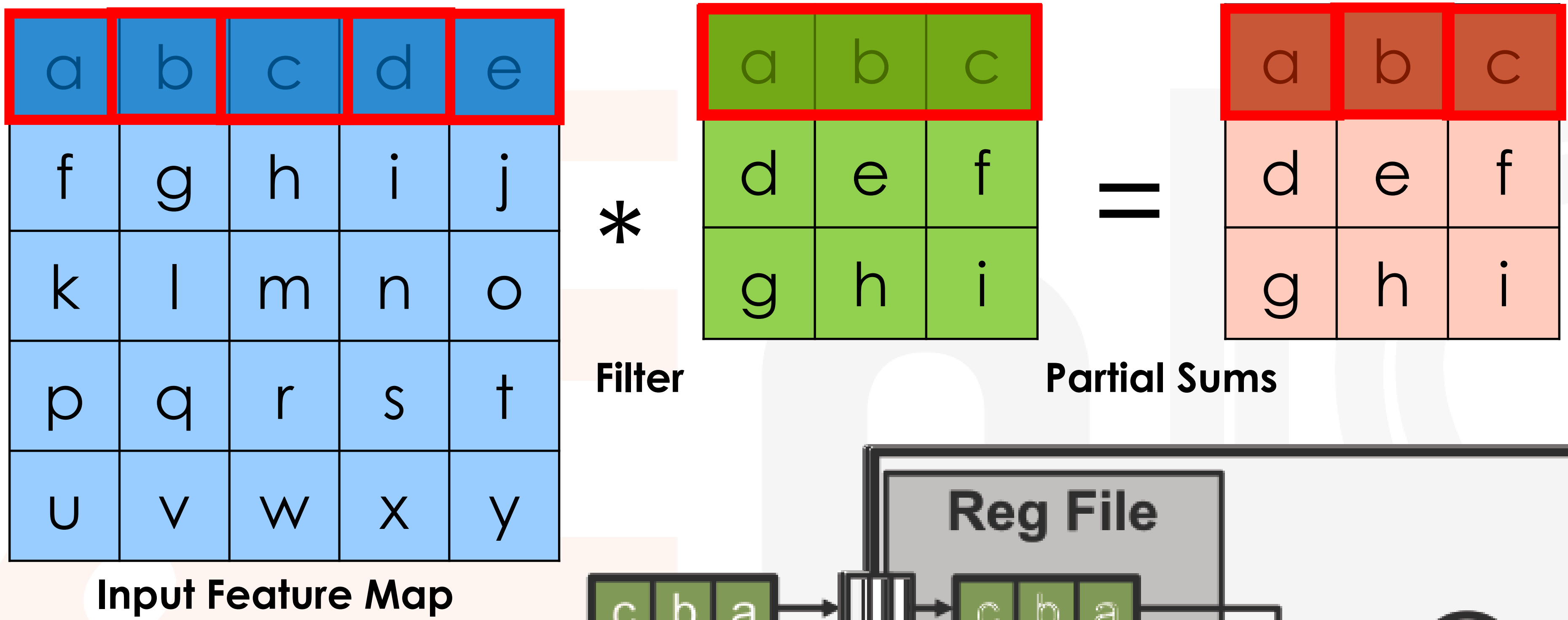
Row Stationary (RS) Dataflow



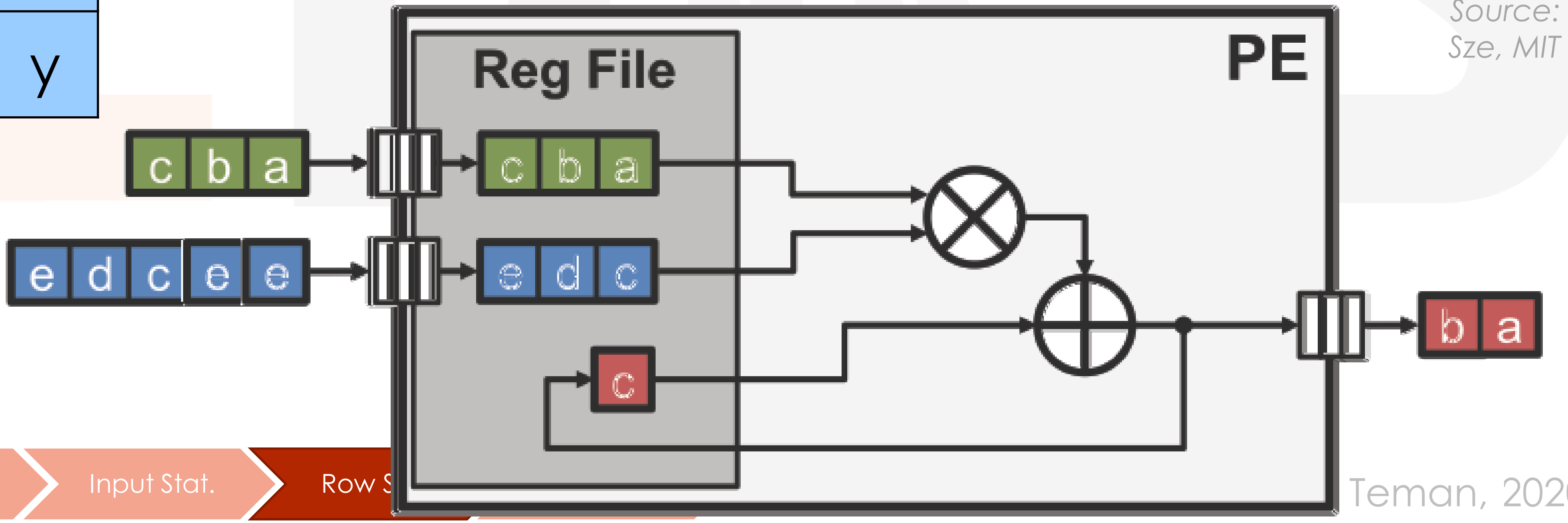
- **Maximize reuse and accumulation at the RF level for all types of data (weights, activations, partial sums)**
- **Process a 1-D row convolution in each PE**
  - Filter weights kept stationary inside the PE
  - ifmaps streamed into the PE
  - PE does MAC for each sliding window at a time →one memory space for partial sum accumulation
- **No ifmap overlap between different sliding windows**
  - Input activations kept in RF and reused
  - 1-D convolution goes through all sliding windows in the row
- **Multiple PEs complete a 2-D convolution**

Source: Sze, MIT

1D Row Convolution in PE

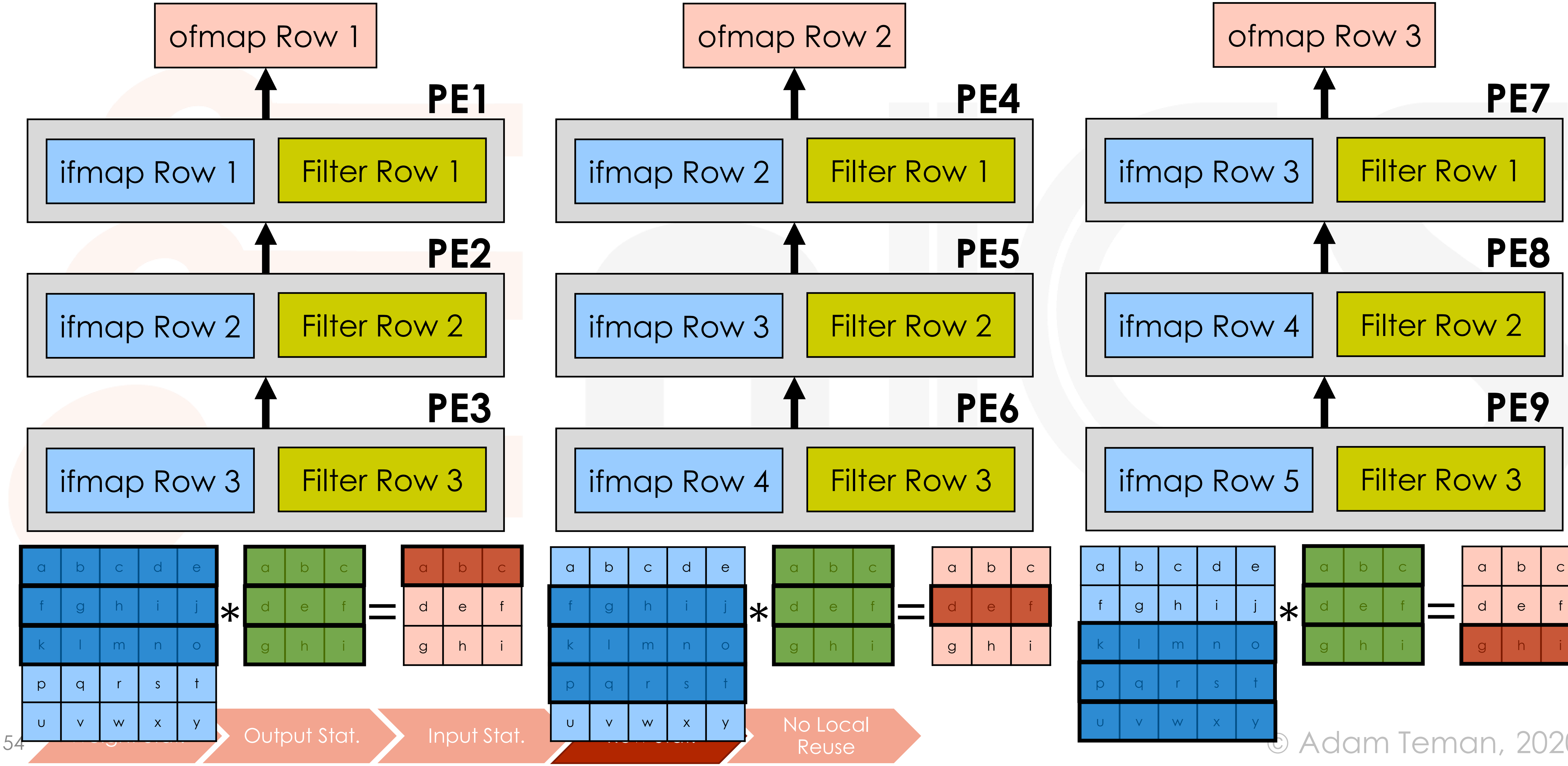


- Maximize row convolutional reuse
- Maximize row psum accumulation in RF

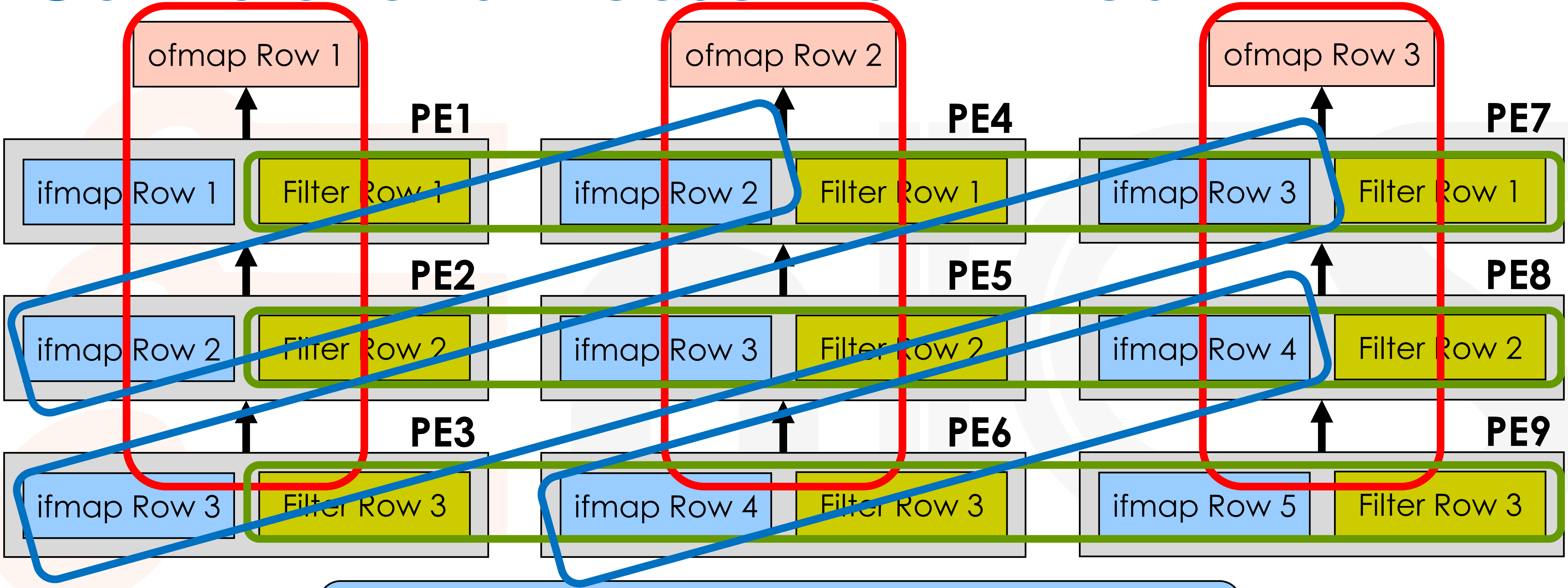


Source: Sze, MIT

2D Convolution in PE Array



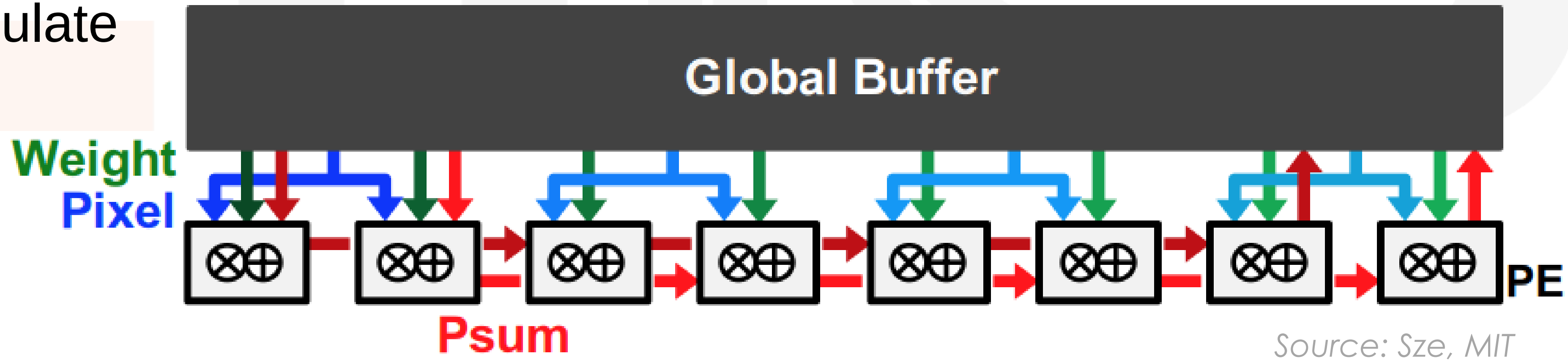
Convolutional Reuse Maximized



- Filter rows are reused across PEs horizontally
- Fmap rows are reused across PEs diagonally
- Partial sums accumulate across PEs vertically

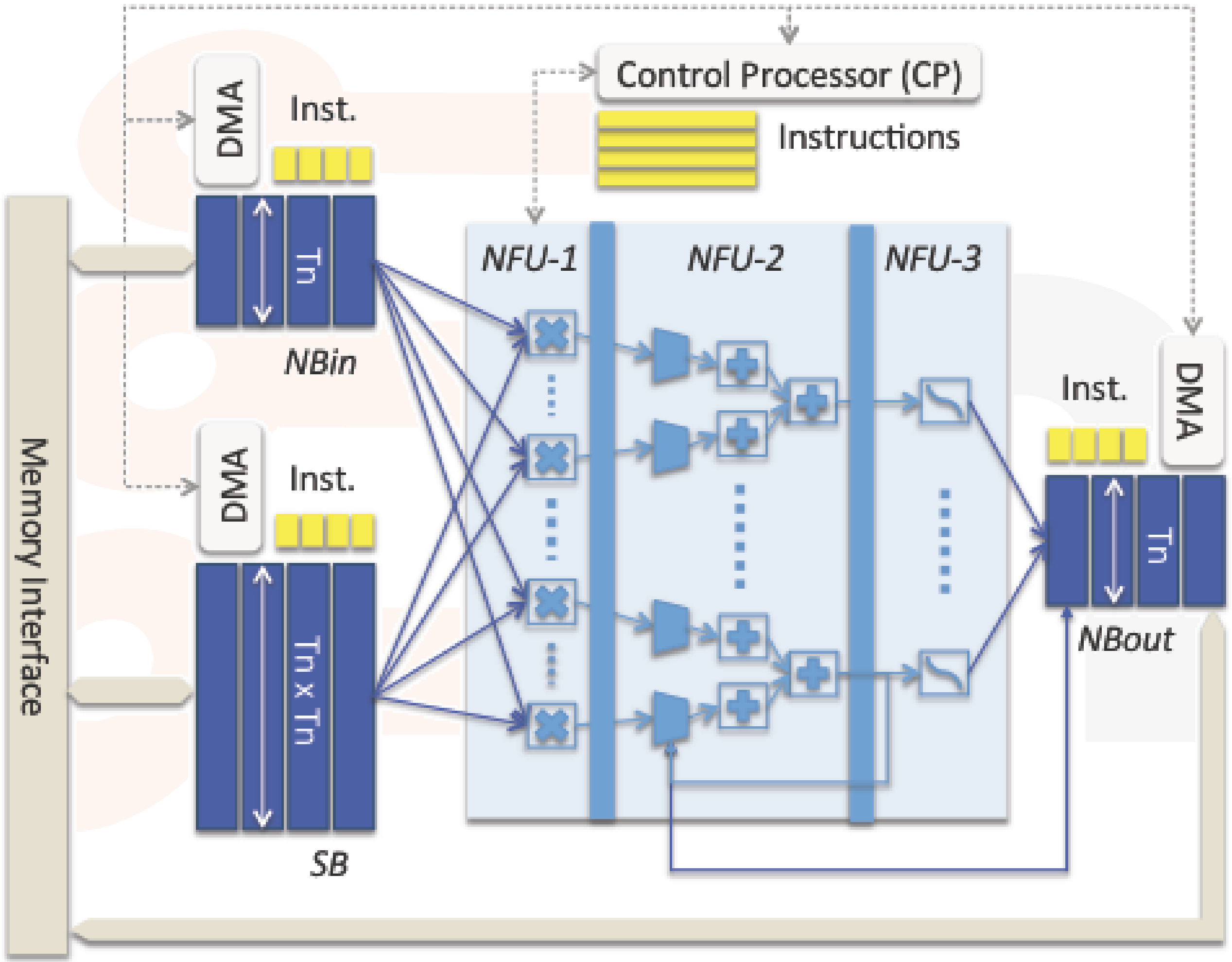
No Local Reuse (NLR) Dataflow

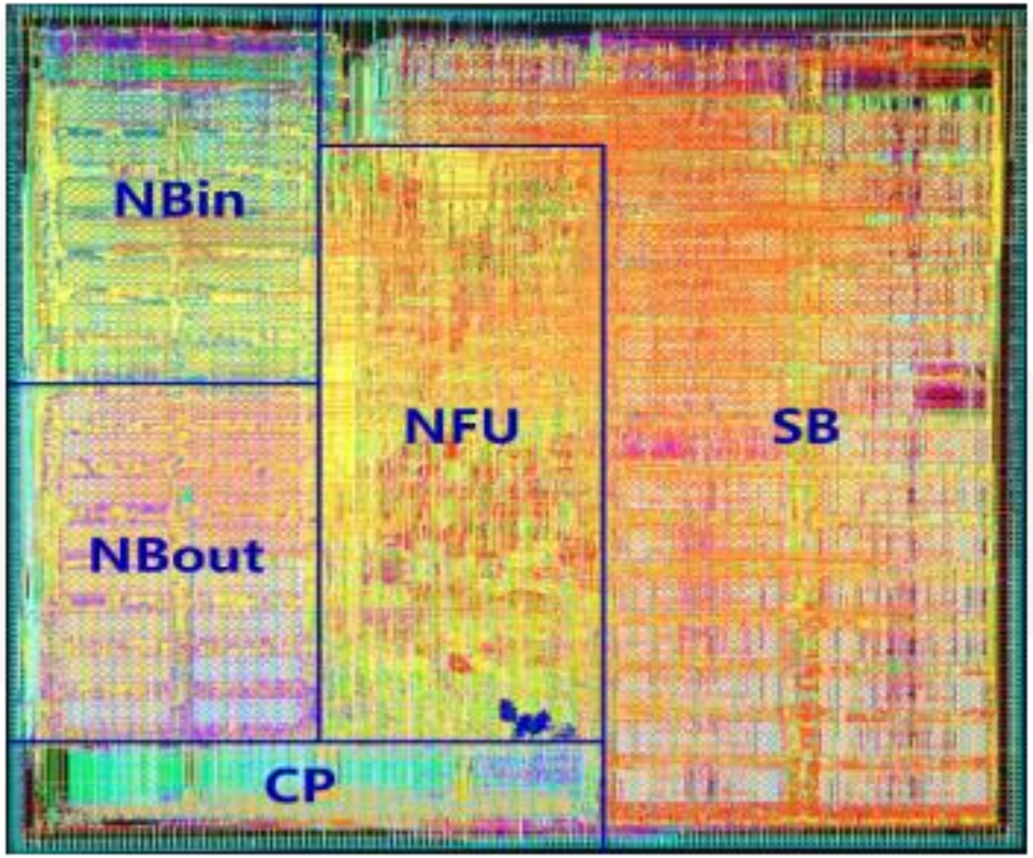
- Small RFs are energy efficient (pJ/b), but area inefficient (um^2/b)
- NLR maximizes storage capacity to minimize off-chip memory bandwidth
- All area is allocated to the global buffer.
- Nothing stationary inside PE →increased traffic on-chip.
  - Multicast activations
  - Single-cast filter weights
  - Spatially accumulate partial sums



Source: Sze, MIT

NLR Example: DianNao





Source: Chen, et al. 2014

Main References

- Sze, et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", Proceedings of the IEEE, 2017
- Sze, et al. ISCA Tutorial 2019