



Designing AI Processors

Chia-Chi Tsai (蔡家齊)
cctsai@gs.ncku.edu.tw

AI System Lab
Department of Electrical Engineering
National Cheng Kung University

Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
- Roofline Model
- Energy

Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
- Roofline Model
- Energy

How Can We Compare Designs?



- Key metrics
 - Accuracy
 - Throughput
 - Latency
 - Energy consumption
 - Power consumption
 - Hardware cost
- Importance of each of these metrics
- Factors that affect each metric

Accuracy

- Indicate the quality of the result for a given task
 - The units used to measure accuracy depend on the task
 - Image classification
 - Top-1, Top-5 accuracy
 - Image detection
 - Mean Average Precision (mAP)
- Factors that affect accuracy
 - Difficulty of the task and dataset
 - Object detection or semantic segmentation is more difficult than classification
 - Evaluating hardware using well-studied, widely used DNN models, tasks, and datasets

Accuracy

- Motivated by the impact of the SPEC benchmarks for general purpose computing
- MLPerf
 - Serve as a common set of well-studied DNN models to evaluate the performance
 - Put together a broad suite of DNN models
 - Various types of DNNs
 - Various tasks
 - Image classification, object identification, translation, speech-to-text, recommendation, sentiment analysis, and reinforcement learning
 - Enable fair comparison of various software frameworks, hardware accelerators

Throughput and Latency

- Throughput
 - Amount of data that can be processed that can be completed in a given time period
 - Reported as $\frac{\text{Number of operations}}{\text{seconds}}$
 - Applications that action needs to be taken based on the analysis requires high throughput
 - E.g., Real-time video processing, security, terrorist prevention medical diagnosis, drug discovery
- Latency
 - $\text{time}_{\text{result is generated}} - \text{time}_{\text{input data arrives to a system}}$
 - Real-time interactive applications require low latency
 - E.g., AR, Autonomous navigation, robotics

Throughput vs. Latency

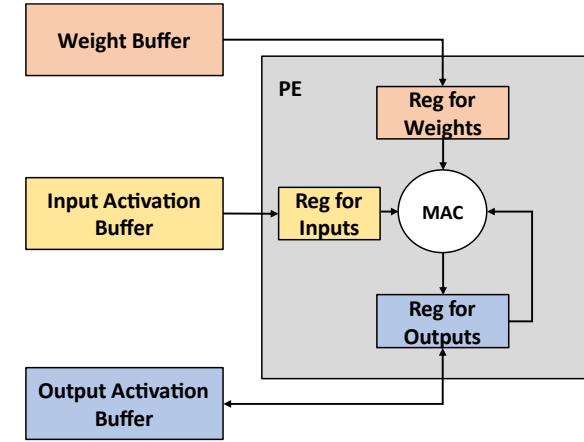
- Pitfall
 - Throughput and latency can be directly derivable from one another
- Batching input data
 - Increase throughput but increase latency in real-time video processing
 - E.g., at 30 frames per second and a batch of 100 frames, some frames will experience at least 3.3 second delay
- Real-time navigation require low latency but can bear low throughput

Factors that Affect Throughput

$$\frac{\text{Inferences}}{\text{second}} = \frac{\text{operations}}{\text{seconds}} \times \frac{1}{\frac{\text{operations}}{\text{inference}}}$$

Dictated by DNN hardware and DNN models

Dictated by the DNN model



Consider a system comprised of multiple processing elements (PEs)

PE - a simple or primitive core that performs a single MAC operation

$$\frac{\text{operations}}{\text{seconds}} = \left(\frac{1}{\frac{\text{cycles}}{\text{operation}}} \times \frac{\text{cycle}}{\text{second}} \right) \times \text{number of PEs} \times \text{utilization of PEs}$$

Peak throughput for a single PE

Amount of parallelism

degradation due to the inability of the architecture to effectively utilize the PEs

Increase Throughput

- Increase $\frac{\text{cycle}}{\text{second}}$ → higher clock frequency → reducing critical path
 - Affect by design of the MAC
- Increase *number of PEs* → MAC perform in parallel
 - If Area cost of the system is fixed
 - Increasing the area density of the PE
 - Trading off on-chip storage area for more PEs
 - Affect the utilization of the PEs,
 - Increasing the density of PEs
 - Reducing the logic associated with delivering operands to a MAC
 - Multiple MACs shared Control logic
 - E.g., SIMD and SIMT with CPUs and GPUs

Increase Throughput

$$\text{utilization of PEs} = \frac{\text{number of active PEs}}{\text{number of PEs}} \times \text{utilization of active PEs}$$

Ability to distribute the workload to PEs How efficiently those active PEs are processing the workload.

- Number of active PEs - number of PEs that receive work
 - Distribute the workload to as many PEs as possible
- Ability to distribute the workload
 - Determined by the flexibility of the architecture
 - Network-on-Chip(NoC) matters !

Increase Throughput

With the constraint of NoC

- Number of active PEs
 - Determined by the specific allocation of work to PEs by the mapping process
 - Involves the placement and scheduling in space and time of every MAC operation onto the PEs
 - Mapper – a compiler for the DNN hardware
- Utilization of the active Pes
 - Timely delivery of work to the Pes
 - Affected by
 - Bandwidth and latency of the memory hierarchy and network
 - Imbalance of work allocated across PEs
 - Native sparsity often result in lower utilization

Bandwidth Requirements

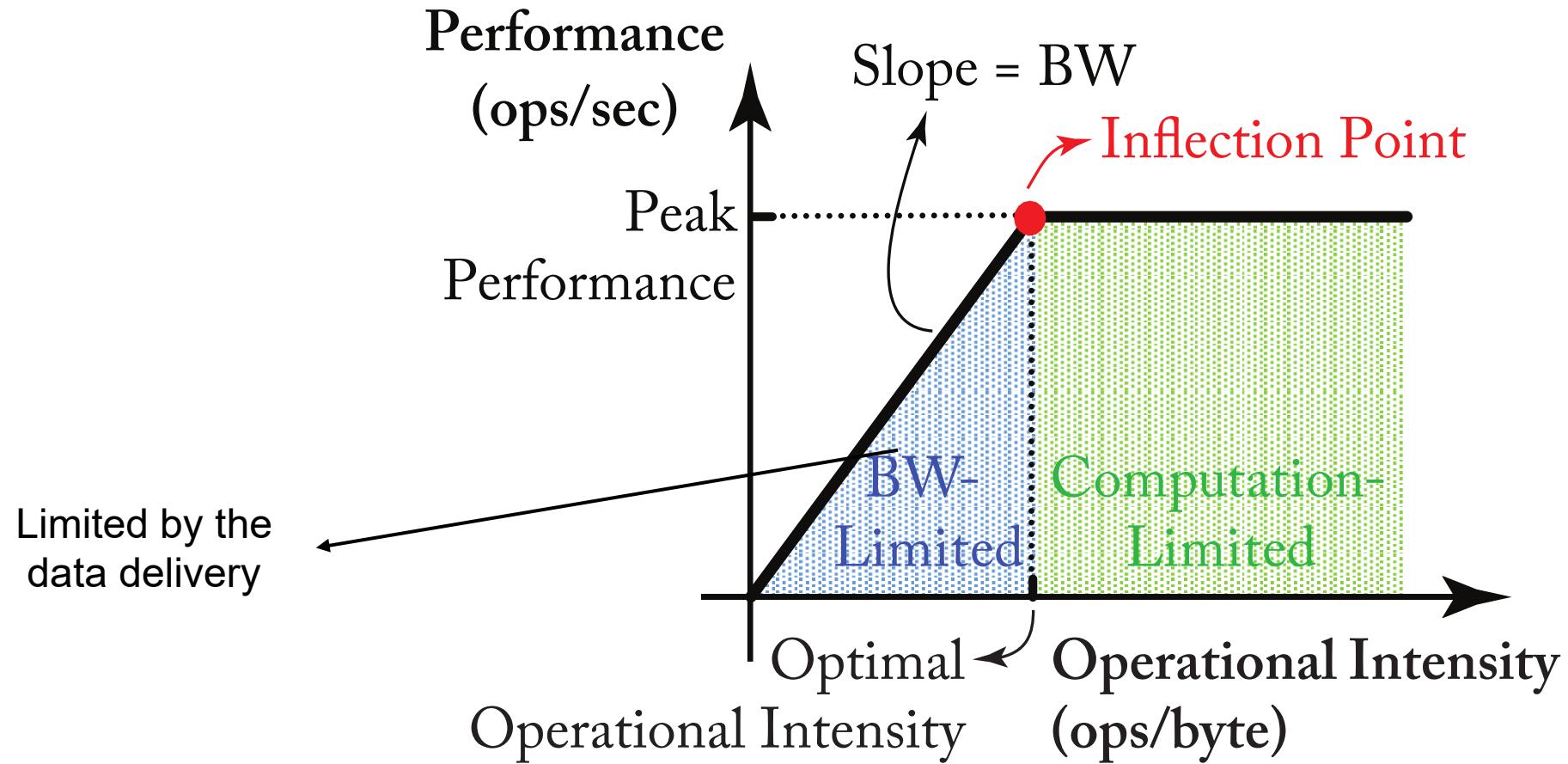
- Affected by the amount of data reuse available in DNN models
- Amount of data reuse can be exploited by the **memory hierarchy** and **dataflow**
- Dataflow determines the order of operations and where data is stored and reused
- E.g., Larger batch size → higher amount of data reuse

Interplay Between

- Reduce the likelihood that a PE needs to wait for data is to store some data locally near or within the PE
 - Increasing the chip area of on-chip storage
 - Reduce the number of PEs

How to balance areas of PEs and on-chip storage?

Roofline Model



Design goal: operate as close as possible to the peak *operations per second* for the operation intensity of a given workload

How DNN Models Affects

- $\frac{\text{operations}}{\text{inference}}$ depends on the DNN model only
- $\frac{\text{operations}}{\text{seconds}}$ depends on both DNN model and hardware
 - Efficient DNN models
 - Reduce the number of MAC operations
 - reduce number of *operations per inference*
 - Wide range of layer shapes
 - Poor utilization of PEs
 - Reduce the overall *operations per second*
 - Number of ineffectual operations is a function of both the DNN model and the input data
 - E.g., MAC accumulate anything multiplied by zero → sparsity

Effectual and Ineffectual Operations

$$\frac{\text{operations}}{\text{cycle}} = \frac{EO + UIO}{\text{cycle}} \times \frac{EO}{EO + UIO} \times \frac{1}{\frac{EO}{\text{operations}}}$$

Constant for a given hardware accelerator design Ability of the hardware to exploit ineffectual operations Related to amount of sparsity and depends on the DNN model

EO: effectual operations
UIO: unexploited ineffectual operations

- Amount of sparsity increase $\rightarrow \frac{\text{operations}}{\text{cycle}}$ increase
 - Requires additional hardware \rightarrow increase critical path, reduce area density of PE
 \rightarrow reduce $\frac{\text{operations}}{\text{second}}$
 - Trade off

Reducing Precision

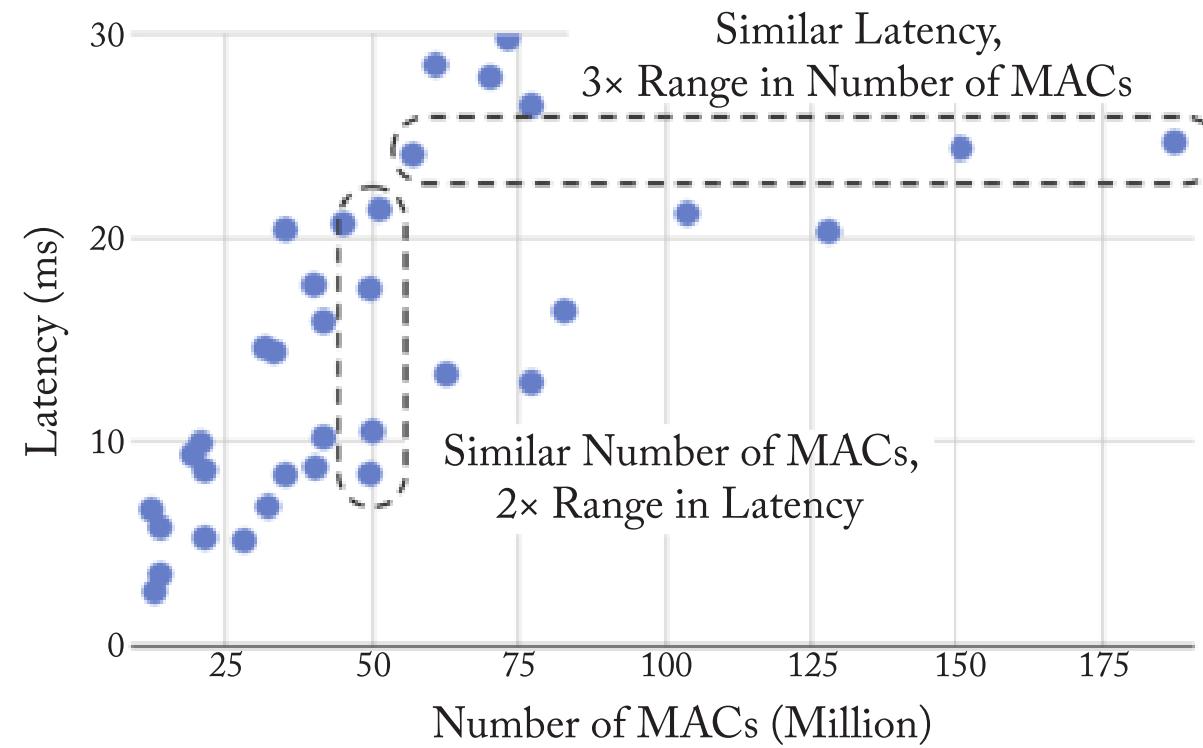
- Increase the number of $\frac{\text{operations}}{\text{second}}$
- Lower precision(bits)
 - lower bandwidth requirement
 - increase utilization of Pes
- Multiple levels of precision requires addition hardware
 - increase critical path, reduce area density of PE
 - reduce $\frac{\text{operations}}{\text{second}}$

Factors that Affect Inferences per Second

Factor	Hardware	DNN Models	Input Data
Operations per inference		V	
Operations per cycle	V		
Cycles per second	V		
Number of PEs	V		
Number of active PEs	V	V	
Utilization of active PEs	V	V	
Effectual operations out of (total) operations		V	V
Effectual operations plus unexploited ineffectual operations per cycle	V		

MAC Alone as an Performance Estimation

- Measured on Pixel phone



MAC operations is not a good predictor of latency...

Design efficient DNN models with hardware in the loop !

Energy Efficiency and Power Consumption

- Energy Efficiency
 - Given unit of energy, How much data that can be processed
 - Edge processing require high energy efficiency
 - $\frac{\text{number of operations}}{\text{joule}}$
- Power consumption
 - Amount of energy consumed per unit time
 - Increase power consumption → increase heat dissipation
 - Thermal Design Power (TDP) limited max power consumption
 - $\frac{\text{watts/joules}}{\text{seconds}}$

Relation between Throughput

$$\frac{\text{inferences}}{\text{second}} \leq \max\left(\frac{\text{joules}}{\text{second}}\right) \times \frac{\text{inferences}}{\text{joules}}$$

- increasing the $\frac{\text{inferences}}{\text{joules}}$
 - increase $\frac{\text{inferences}}{\text{second}}$
 - increase throughput

Factors that Affect Energy Efficiency

$$\frac{\text{inferences}}{\text{joules}} = \frac{\text{operations}}{\text{joules}} \times \frac{1}{\frac{\text{operations}}{\text{inference}}}$$

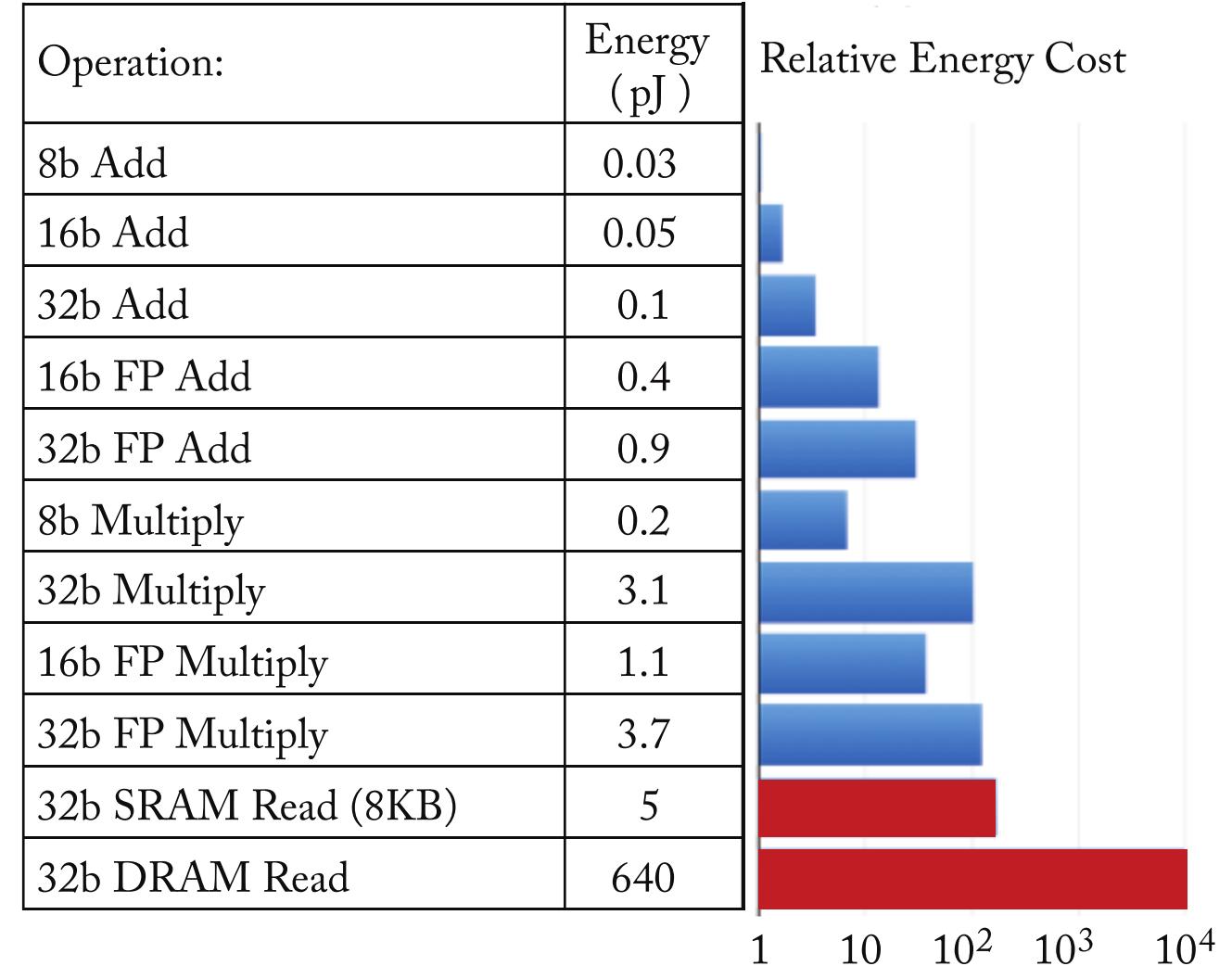
Dictated by both the hardware and DNN model Dictated by DNN model

- $\text{Energy}_{total} = \text{Energy}_{data} + \text{Energy}_{MAC}$

- $\frac{\text{joules}}{\text{operation}} = \alpha \times C \times V_{DD}^2$
 - α : switching activity
 - C: switching capacitance
 - V_{DD} : supply Voltage

Energy Consumptions

- 45nm process
- Energy consumption is dominated by the data movement
 - Capacitance of data movement tends to be much higher than the capacitance for arithmetic operations(PEs)
- Larger memories and longer interconnects
 - Consume more energy
- **Optimizing data movement is the key !**



Operations per Joules

$$\frac{\text{operations}}{\text{joules}} = \frac{EO + UIO}{\text{joules}} \times \frac{EO}{EO + UIO} \times \frac{1}{\frac{EO}{\text{operations}}}$$

Constant for a given hardware accelerator design

Ability of the hardware to exploit ineffectual operations

Related to amount of sparsity and depends on the DNN model

- A function of the ability of the hardware to exploit sparsity to avoid performing ineffectual MAC operations

Hardware Cost

- Process technology
- Amount of on-chip storage and compute → Area
 - PEs in DNN accelerator
 - Cores in CPUs and GPUs
 - Digital Signal Processing(DSP) Engines in FPGAs
- Off-chip bandwidth
 - Complexity of the packaging
 - PCB design
 - Off-chip DRAM, NVLink, etc.

Flexibility

- Range of DNN models and tasks that can be supported on the DNN processor
- Ability of the software environment to maximally exploit the capabilities of the hardware
- Two aspect of support for different DNN models
 - Functionally support
 - Maintain efficiency
- Different network architectures
- Different levels of precision
- Different degrees of sparsity
- Different types of DNN layers and computation beyond MAC operations

Scalability

- How well a design can be scaled up to achieve higher throughput and energy efficiency
 - Size of the Chip
 - Multiple chips
 - PE, on-chip storage
- Usually non-linear when scaling up
 - Reduced utilization of PEs
 - Increase cost of data movement

Metrics for DNN Hardware



- Accuracy
 - Quality of result for a given task
- Throughput
 - Analytics on high volume data
 - Real-time performance (e.g., video at 30 fps)
- Latency
 - For interactive applications (e.g., autonomous navigation)
- Energy and Power
 - Edge and embedded devices have limited battery capacity
 - Data centers have stringent power ceilings due to cooling costs
- Hardware Cost
 - Money

Specifications to Evaluate Metrics

- Accuracy
 - Difficulty of dataset and/or task should be considered
- Throughput
 - Number of cores (include utilization along with peak performance)
 - Runtime for running specific DNN models
- Latency
 - Include batch size used in evaluation
- Energy and Power
 - Power consumption for running specific DNN models
 - Include external memory access
- Hardware Cost
 - On-chip storage, number of cores, chip area + process technology

Example: Metrics of Eyeriss Chip

ASIC Specs	Input
Process Technology	65nm LP TSMC (1.0V)
Total Core Area (mm ²)	12.25
Total On-Chip Memory (kB)	192
Number of Multipliers	168
Clock Frequency (MHz)	200
Core area (mm ²) /multiplier	0.073
On-Chip memory (kB) / multiplier	1.14
Measured or Simulated	Measured

Metric	Units	Input
Name of CNN Model	Text	AlexNet
Top-5 error classification on ImageNet	#	19.8
Supported Layers		All CONV
Bits per weight	#	16
Bits per input activation	#	16
Batch Size	#	4
Runtime	ms	115.3
Power	mW	278
Off-chip Access per Image Inference	MBytes	3.85
Number of Images Tested	#	100

Comprehensive Coverage

- **All metrics** should be reported for fair evaluation of design tradeoffs
- Examples of what can happen if certain metric is omitted:
 - **Without the accuracy given for a specific dataset and task**, one could run a simple DNN and claim low power, high throughput, and low cost – however, the processor might not be usable for a meaningful task
 - **Without reporting the off-chip bandwidth**, one could build a processor with only multipliers and claim low cost, high throughput, high accuracy, and low chip power – however, when evaluating system power, the off-chip memory access would be substantial
- Are results measured or simulated? On what test data?

Evaluation Process

- The evaluation process for whether a DNN system is a viable solution for a given application might go as follows:
 1. Accuracy determines if it can perform the given task
 2. Latency and throughput determine if it can run fast enough and in real-time
 3. Energy and power consumption will primarily dictate the form factor of the device where the processing can operate
 4. Cost, which is primarily dictated by the chip area, determines how much one would pay for this solution

Outline

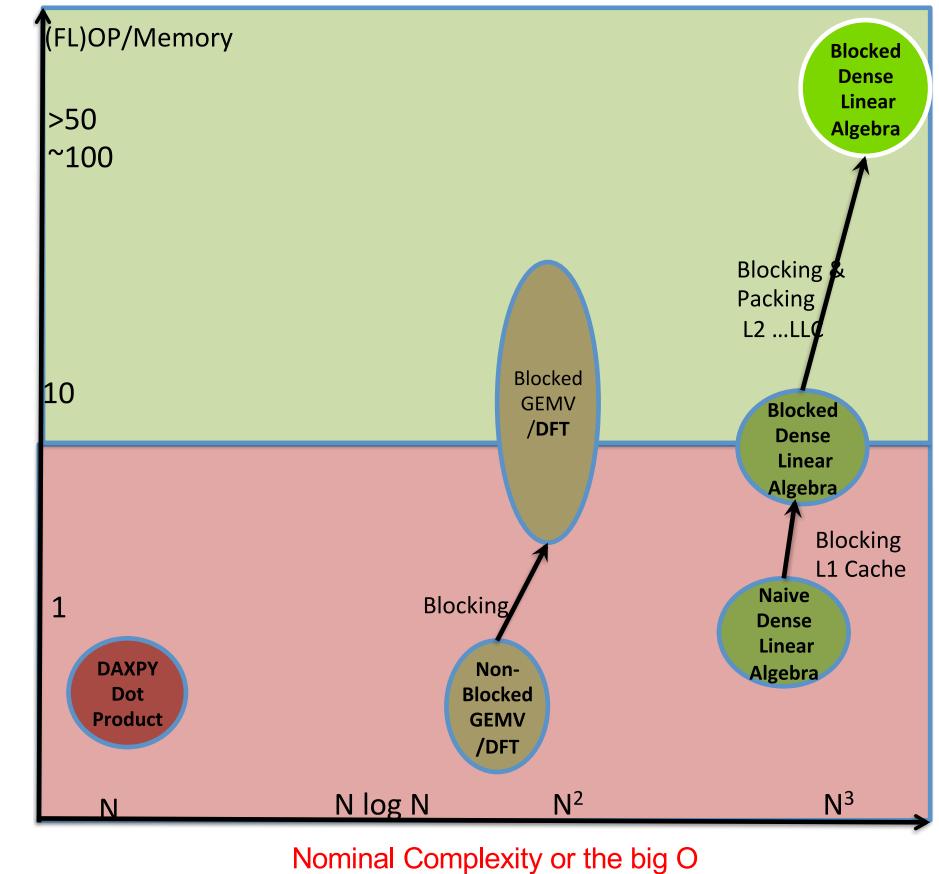
- Benchmarking Metrics
- GEMM Accelerator Design
 - GEMM
 - Systolic and Other Variation
 - Hardware for GEMM
- DNN Accelerator Design
- Roofline Model
- Energy

Basic Linear Algebra Subprograms (BLAS)

- The building blocks of most of scientific computing
- Application codes either call these routines directly, indirectly through other libraries that themselves call the BLAS, or via environments like Matlab
- Three Levels:
 - Level 1: vector-vector operations
 - Level 2: matrix-vector operations
 - Level 3: matrix-matrix operations
- Implementing level-3 BLAS in terms of GEneral Matrix-matrix Multiplication (GEMM) kernels
 - **Only a small set of kernels needed to be highly optimized**

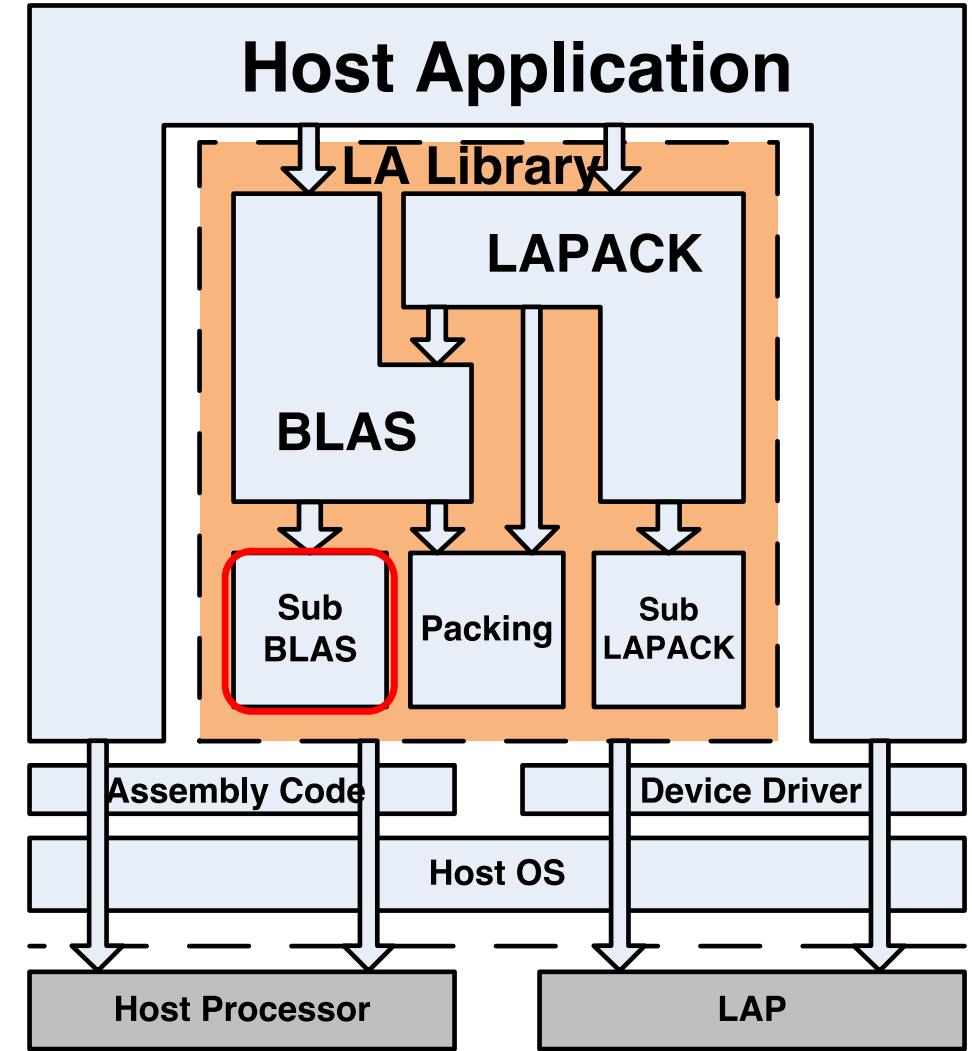
Basic Linear Algebra Subprograms (BLAS)

BLAS Level	Example	# of mem refs	# of flops	Flop /Memory
1	Axpy Dot product	$3n$	$2n$	$2/3$
2	GEMV Matrix-vector mult.	n^2	$2n^2$	2
3	GEMM Matrix-matrix mult.	$4n^2$	$2n^3$	$n/2$



Why GEMM?

- Linear Algebra Package (LAPACK) level
 - Matrix factorizations
- Basic Linear Algebra Subroutines (BLAS)
 - Matrix-matrix and matrix-vector operations
- Inner kernels
 - Hand-optimized
- Accelerators
 - Optimize GEMM
- **GEMM is crucial for many critical applications**
 - Especial for DNNs



Native GEMM Implementation



```
1 C = C + A*B  
2 for i = 1~n  
3   for j = 1~n  
4     for k = 1~n  
5       C(i,j) = C(i,j) + A(i,k) + B(k,j)
```

- Flops = $2 \times n^3 \rightarrow O(n^3)$
- Operate on $3 \times n^2$ words of memory
- Questions
 - What is the performance of this code?
 - Where is the cause of problem with this code?
 - How can we improve it ?
 - What is an easy way to increase the performance with minimum code change?

Recursive GEMM



- Divide & conquer
- How does it work with non-squared matrices?
- Why does it improve performance?

$$\begin{array}{c} \text{A} \\ \begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \end{array} \times \begin{array}{c} \text{B} \\ \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} \end{array} = \begin{array}{c} \text{C} \\ \begin{array}{|c|c|} \hline A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \\ \hline \end{array} \end{array}$$

$$\begin{bmatrix} A_1 \\ A_2 \end{bmatrix} B = \begin{bmatrix} A_1 B \\ A_2 B \end{bmatrix}$$

Case 1

$$[A_1 \quad A_2] \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} = [A_1 B_1 + A_2 B_2]$$

Case 2

$$A [B_1 \quad B_2] = [AB_1 \quad AB_2]$$

Case 3

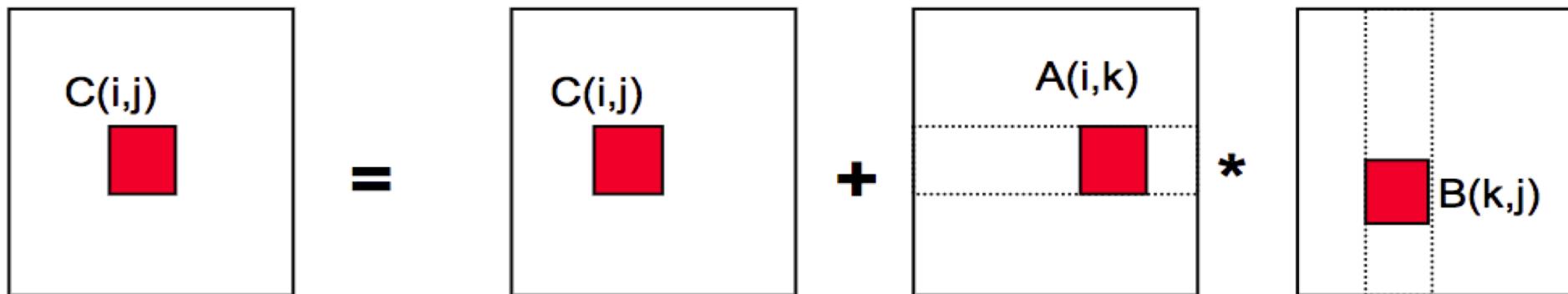
Blocked GEMM



Consider A, B, C to be $n \times n$ matrix viewed as $N \times N$ matrices of $b \times b$ subblocks,

where $b = \frac{n}{N}$ is called the **block size**

```
1 for i = 1 to N
2   for j = 1 to N
3     {read block C(i,j) into fast memory}
4     for k = 1 to N
5       {read block A(i,k) into fast memory}
6       {read block B(k,j) into fast memory}
7       C(i,j) = C(i,j) + A(i,k) * B(k,j) //do matrix multiply on block
8       {write block C(i,j) back to slow memory}
```



GEMM Kernel

- Rank-1 update

- Update matrix by adding outer product of two vectors to it

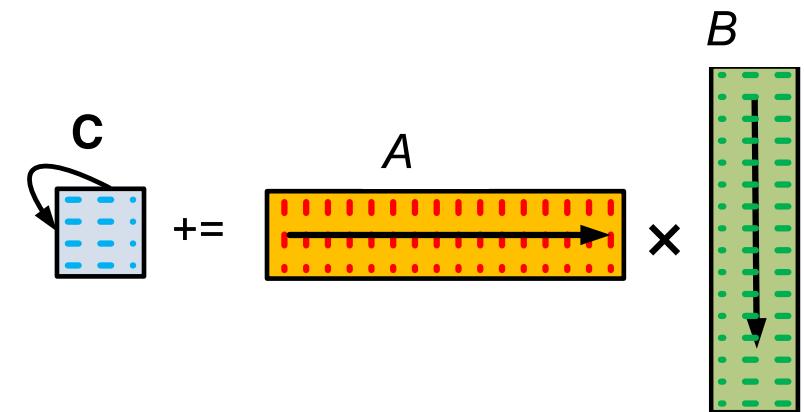
$$\begin{bmatrix} \gamma_{0,0} & \cdots & \gamma_{0,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} & \cdots & \gamma_{3,3} \end{bmatrix} += \begin{bmatrix} \alpha_{0,0} \\ \vdots \\ \alpha_{3,0} \end{bmatrix} [\beta_{0,0} \quad \dots \quad \beta_{0,3}]$$

- Matrix multiplication as a series of rank-1 updates

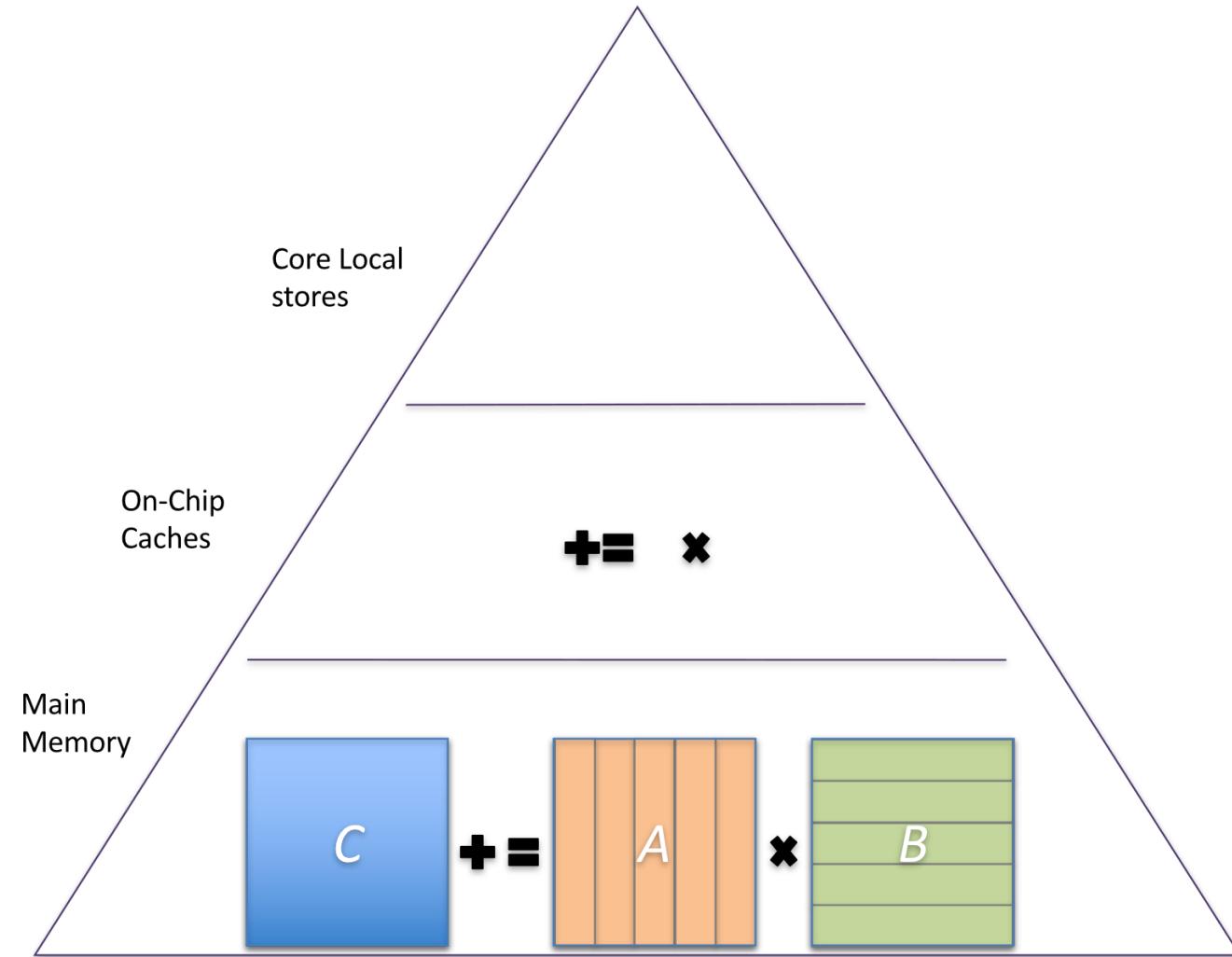
- Let C, A, and B be 4×4 , $4 \times k_c$, and $k_c \times 4$ matrices

- $C += AB$ can be computed as:
for $i = 0 \sim k_c - 1$

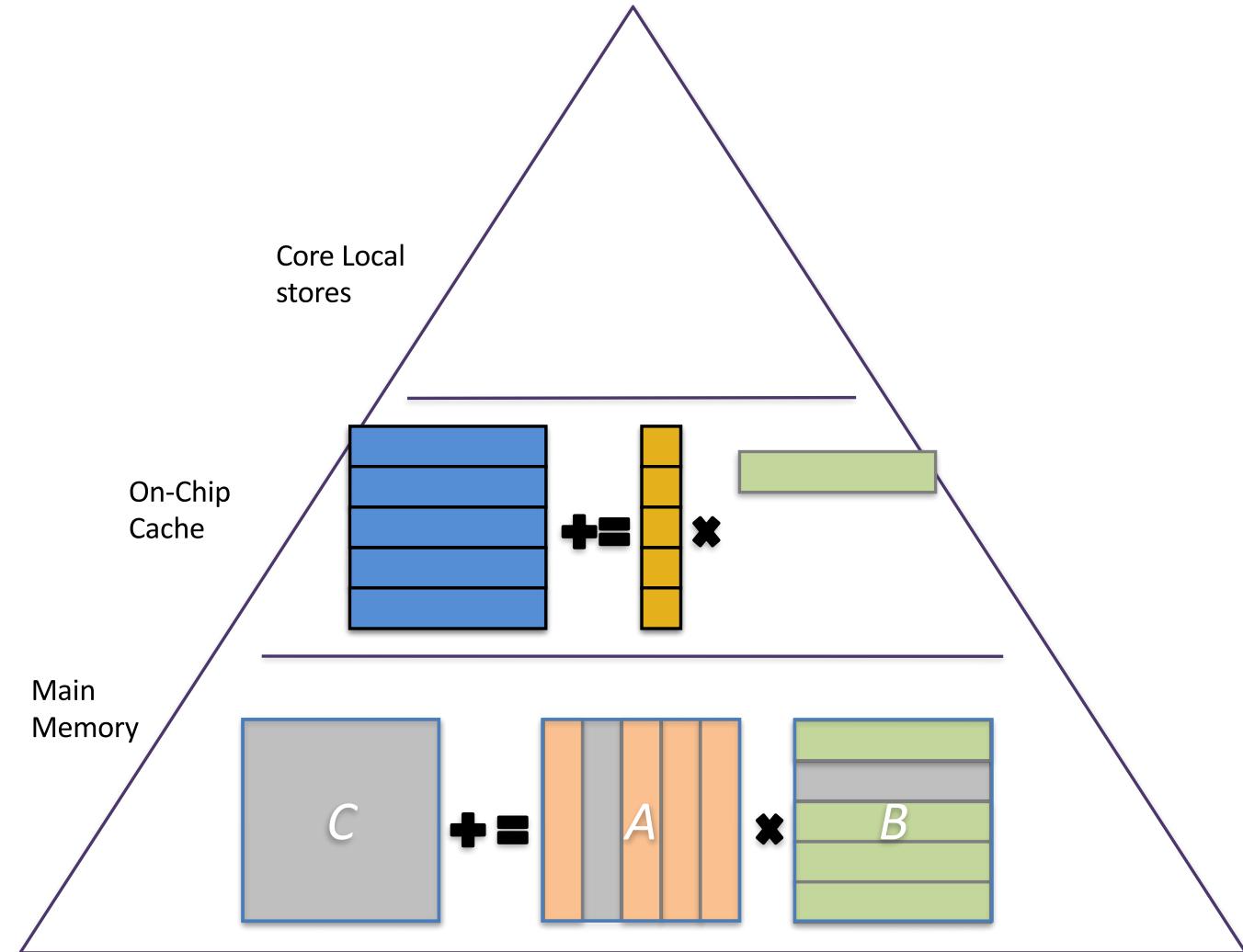
$$\begin{bmatrix} \gamma_{0,0} & \cdots & \gamma_{0,3} \\ \vdots & \ddots & \vdots \\ \gamma_{3,0} & \cdots & \gamma_{3,3} \end{bmatrix} += \begin{bmatrix} \alpha_{0,0} \\ \vdots \\ \alpha_{3,0} \end{bmatrix} [\beta_{0,0} \quad \dots \quad \beta_{0,3}]$$



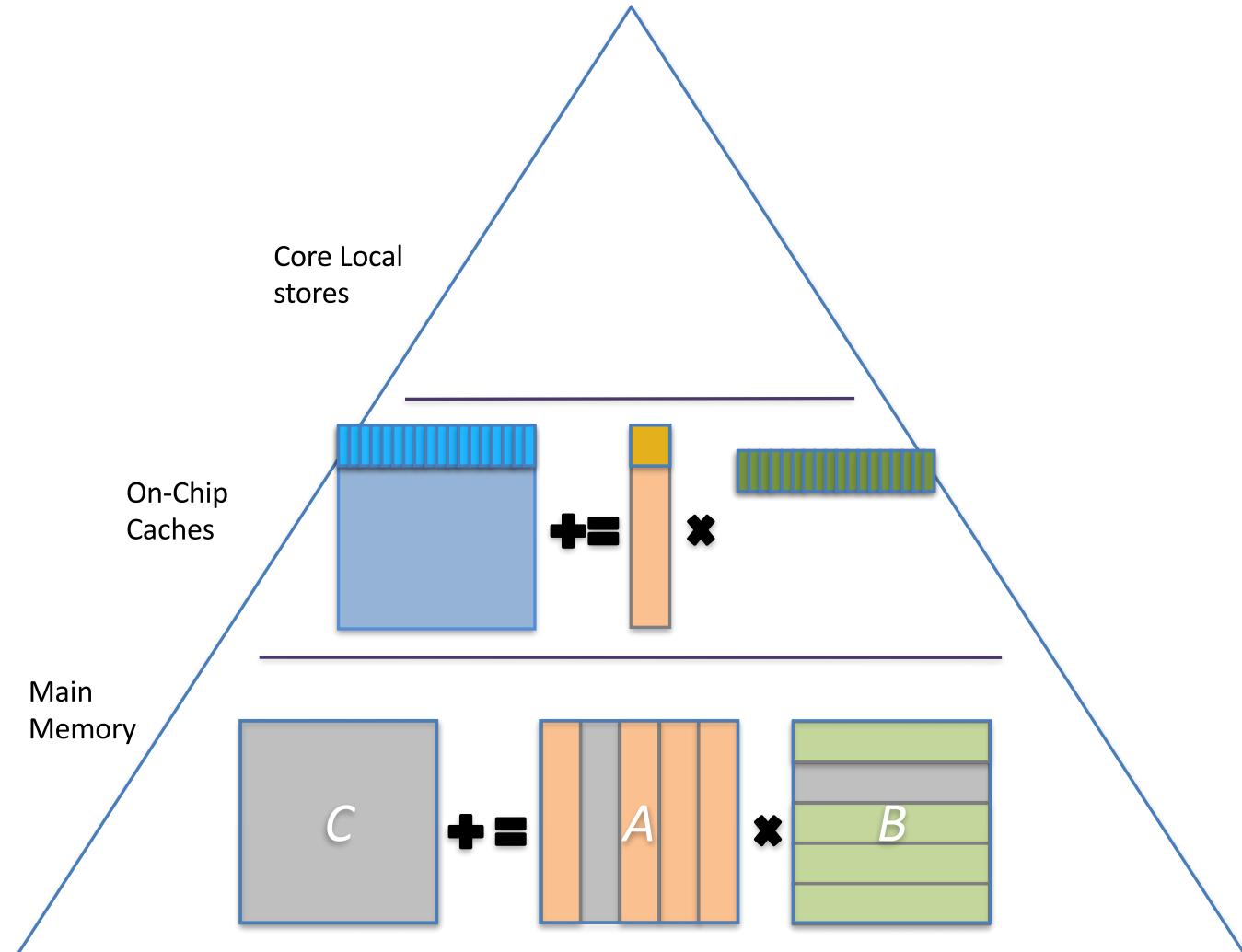
Memory Hierarchy



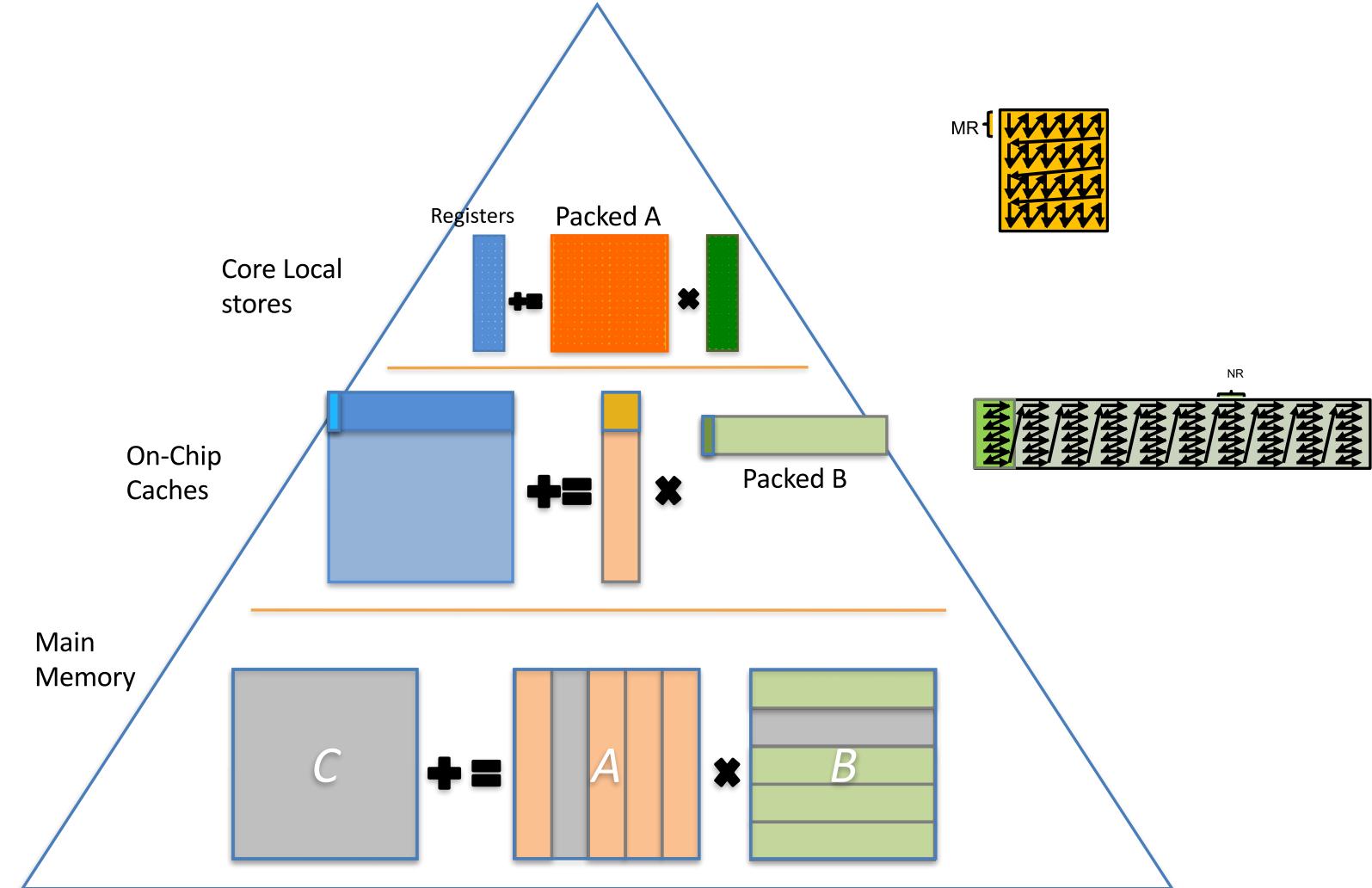
Memory Hierarchy



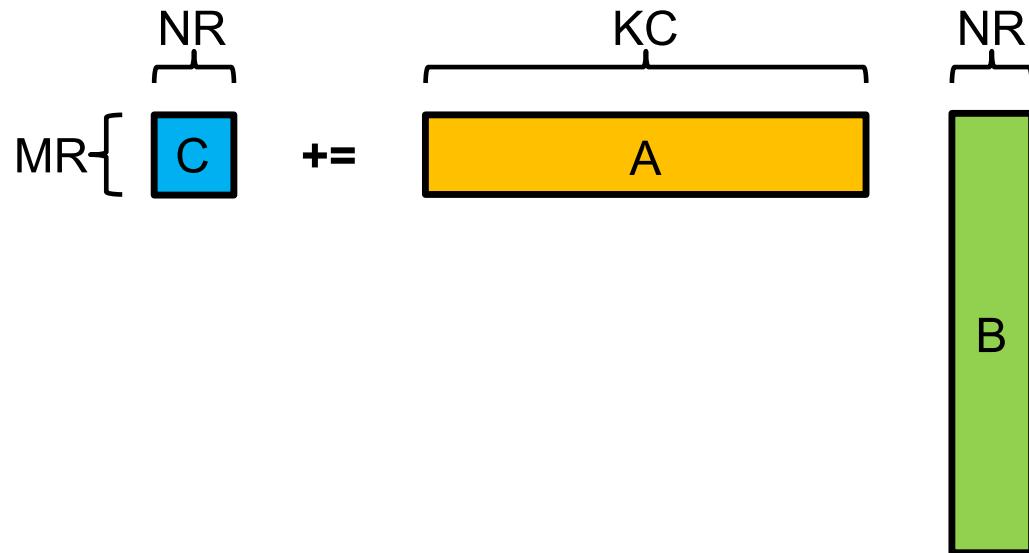
Memory Hierarchy



Memory Hierarchy

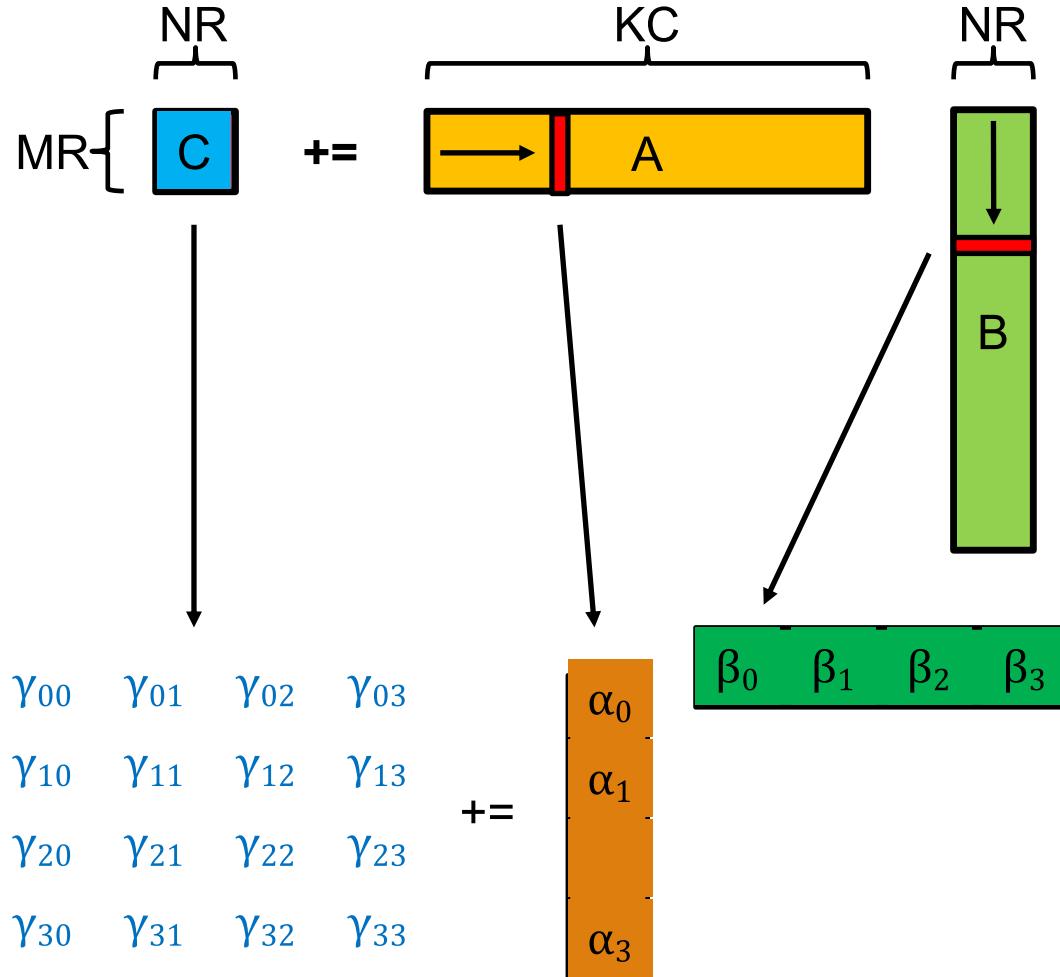


The GEMM Micro-kernel



```
1 for (0 to NC : NR)
2   for (0 to MC : MR)
3     for (0 to KC : 1)
4       {block-dot product}
```

The GEMM Micro-kernel



```

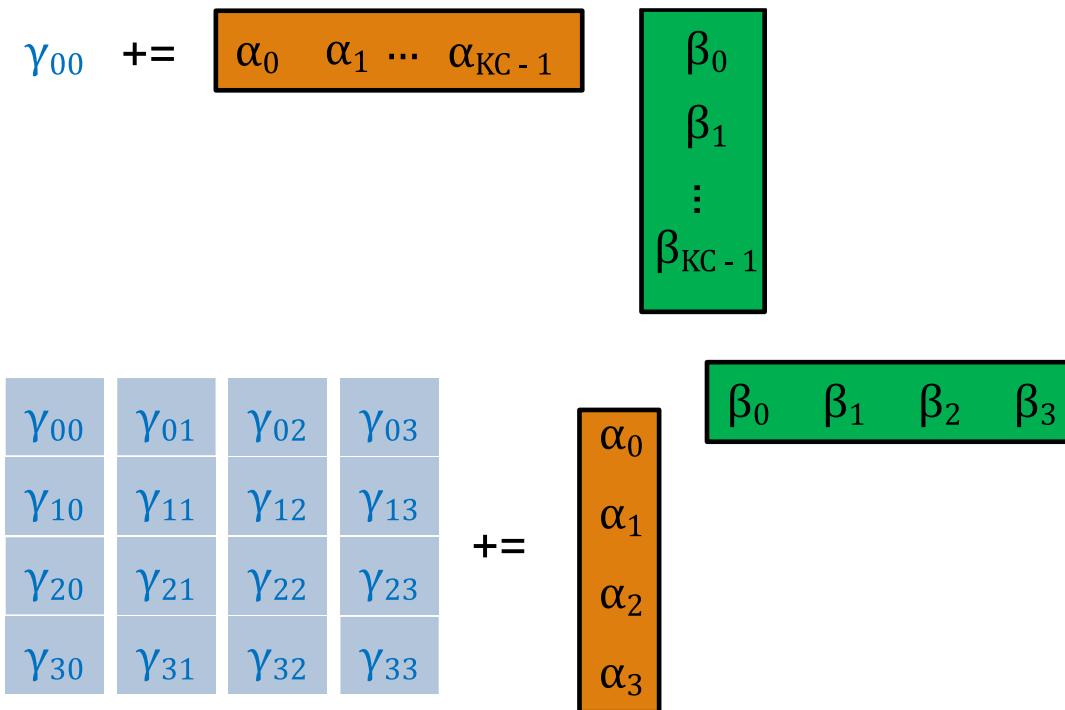
1 for (0 to NC : NR)
2   for (0 to MC : MR)
3     for (0 to KC : 1)
4       {block-dot product}
  
```

Typical micro-kernel loop iteration (“block-dot product”)

- Load column of packed A
- Load row of packed B
- Compute outer product
- Update C (kept in registers)

The GEMM Micro-kernel

- Why $M_R \times N_R$ block-dot product?
 - Why not a simple scalar dot product?
 - Ratio of floating-point operations to memory operations

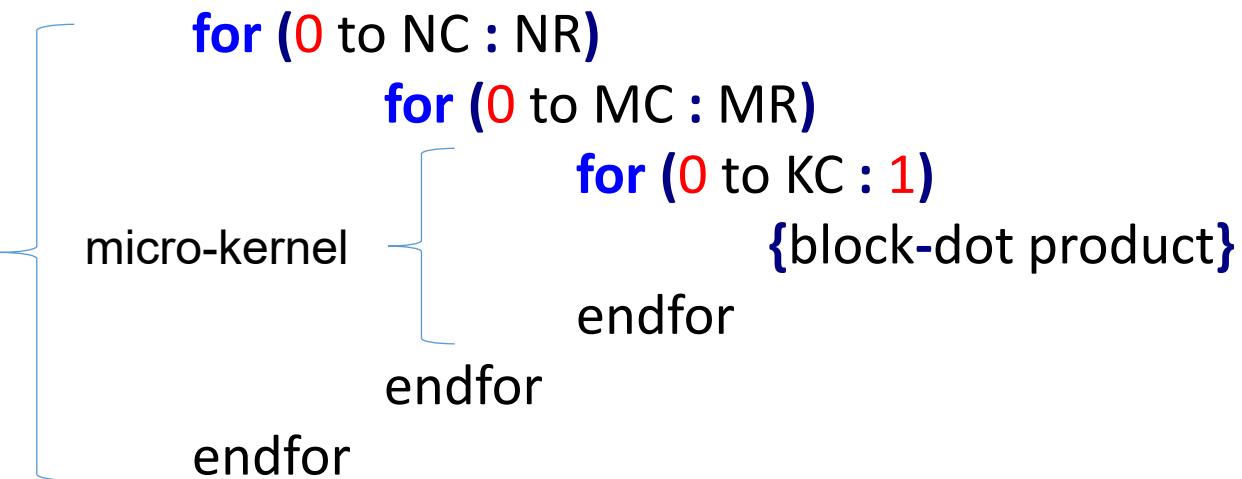


$$\frac{2K_c}{2K_c + 2} \approx 1$$

$$\frac{2M_R N_R K_c}{(M_R + N_R)K_c + 2M_r N_r} \approx \frac{2M_R N_R}{(M_R + N_R)}$$

The GEMM Micro-kernel

- BLIS exposes outer two loops of the inner kernel
 - Key observation: Virtually all of the differences between level-3 inner kernels reside in the outer two loops (e.g. loop bounds)
 - All inner kernels (which we call “macro-kernels”) are provided as part of the BLIS framework (C99)
 - Inner-most exposed loop simply calls micro-kernel



```
for (0 to NC : NR)
    for (0 to MC : MR)
        for (0 to KC : 1)
            {block-dot product}
        endfor
    endfor
endfor
```

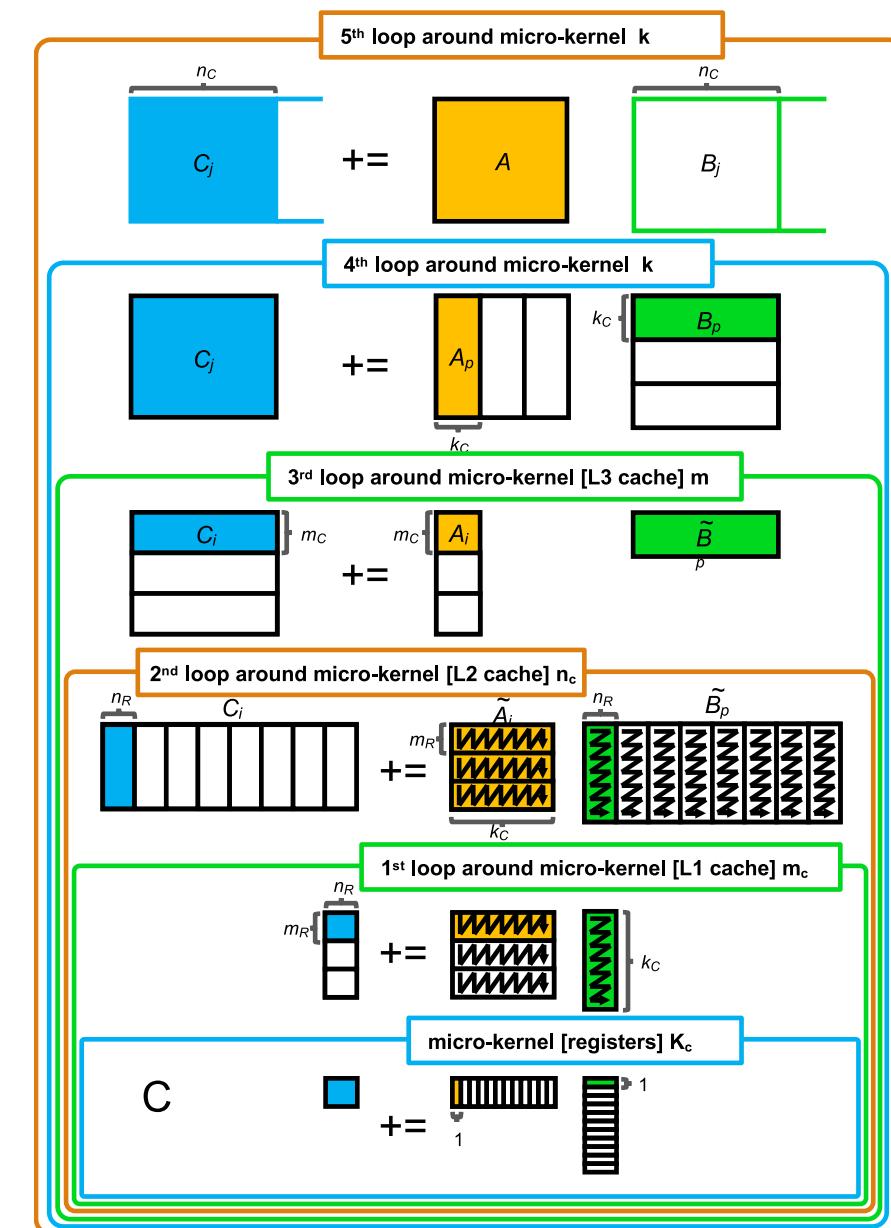
Recap GEMM Loops



- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c \ n_c \ m \ k \ n$
- $m_0 \ n_0 \ k_0 \ m_1 \ n_1 \ m_2 \ k_1 \ n_2$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
    for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
         $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
        for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
             $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
        for  $j_r = 0, \dots, n_c - 1$  in steps of  $n_r$ 
            for  $i_r = 0, \dots, m_c - 1$  in steps of  $m_r$ 
                for  $p_r = 0, \dots, k_c - 1$  in steps of 1
                     $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
                     $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
                     $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 
    
```



Recap GEMM Loops

- Rules for each new character

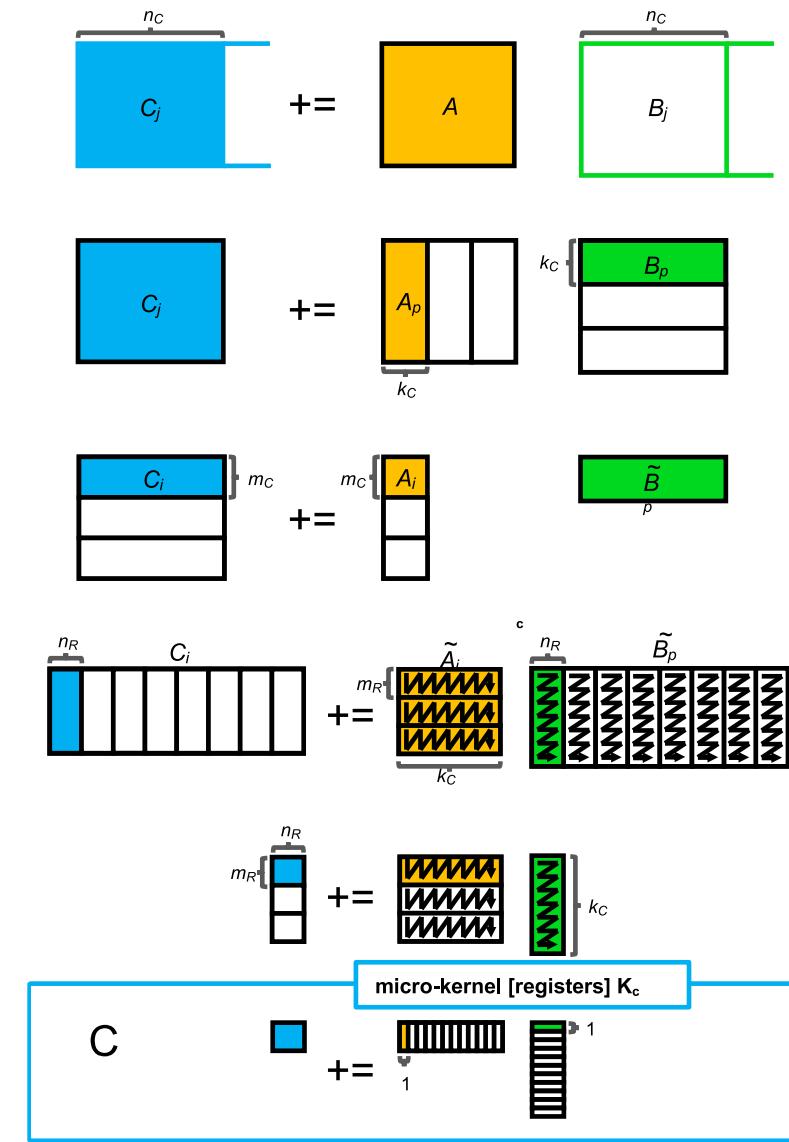
- Buffers
- Re-fetch rate

- $m_r \ n_r \ k_c$
- $m_0 \ n_0 \ k_0$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
  for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
     $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
    for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
       $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
    _____
    for  $j_r = 0, \dots, n_c - 1$  in steps of  $n_r$ 
      for  $i_r = 0, \dots, m_c - 1$  in steps of  $m_r$ 
        for  $p_r = 0, \dots, k_c - 1$  in steps of 1
           $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
           $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
           $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 

```

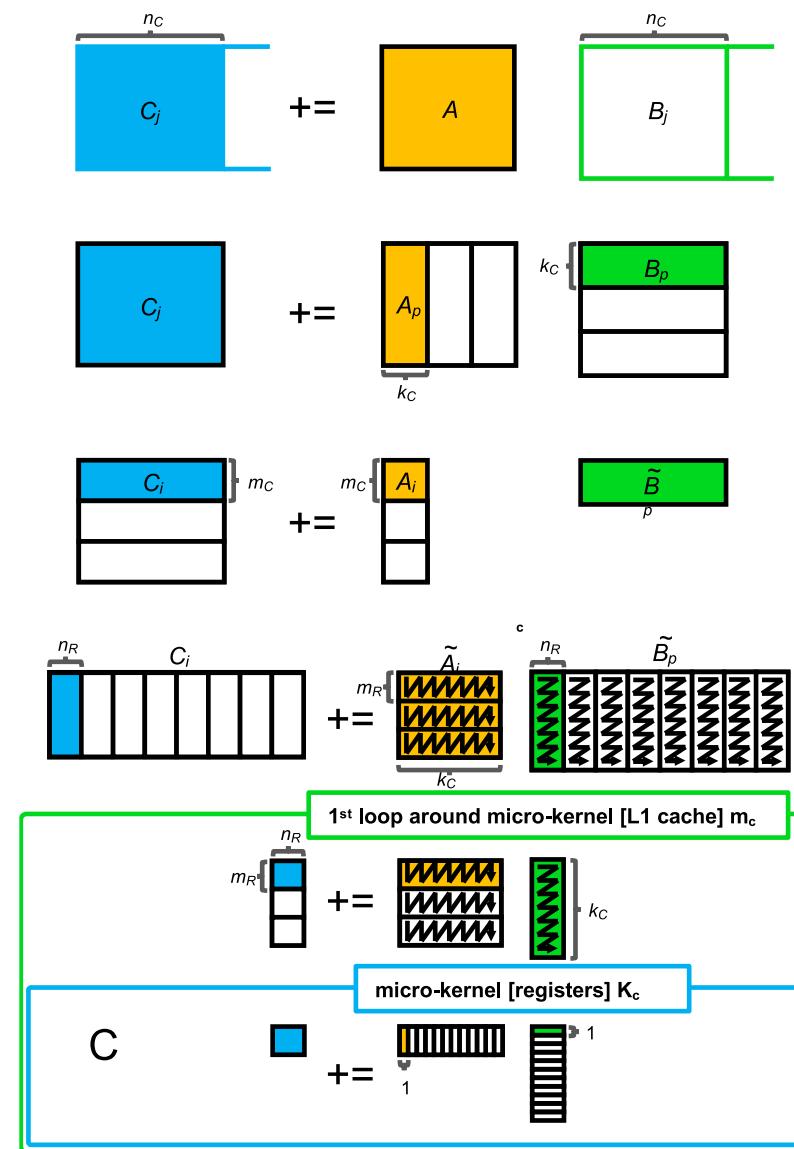


Recap GEMM Loops

- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c$
- $m_0 \ n_0 \ k_0 \ m_1$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
  for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
     $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
    for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
       $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
    _____
    for  $j_r = 0, \dots, n_c - 1$  in steps of  $n_r$ 
      for  $i_r = 0, \dots, m_c - 1$  in steps of  $m_r$ 
        for  $p_r = 0, \dots, k_c - 1$  in steps of 1
           $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
           $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
           $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 
        
```



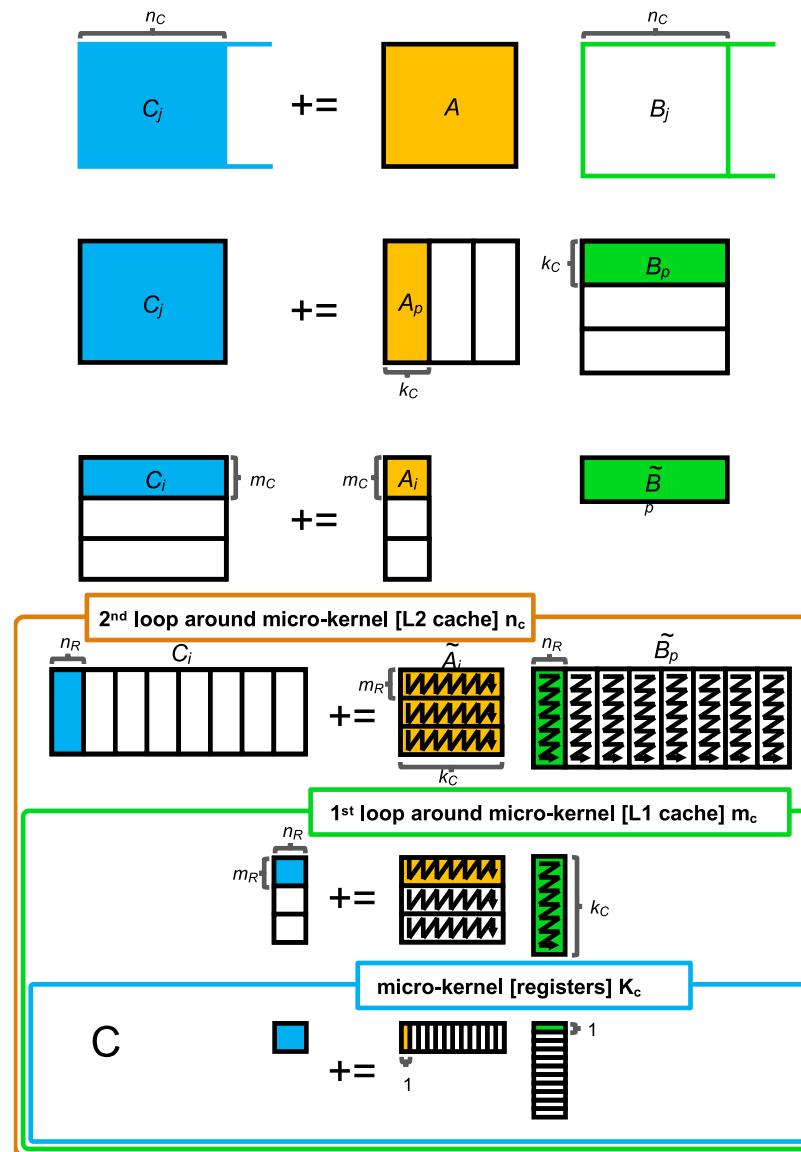
Recap GEMM Loops

- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c \ n_c$
- $m_0 \ n_0 \ k_0 \ m_1 \ n_1$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
  for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
     $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
    for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
       $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
    for  $j_r = 0, \dots, n_r - 1$  in steps of  $n_r$ 
      for  $i_r = 0, \dots, m_c - 1$  in steps of  $m_r$ 
        for  $p_r = 0, \dots, k_c - 1$  in steps of 1
           $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
             $= A_c(i_r : i_r + m_r - 1, p_r)$ 
             $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 

```



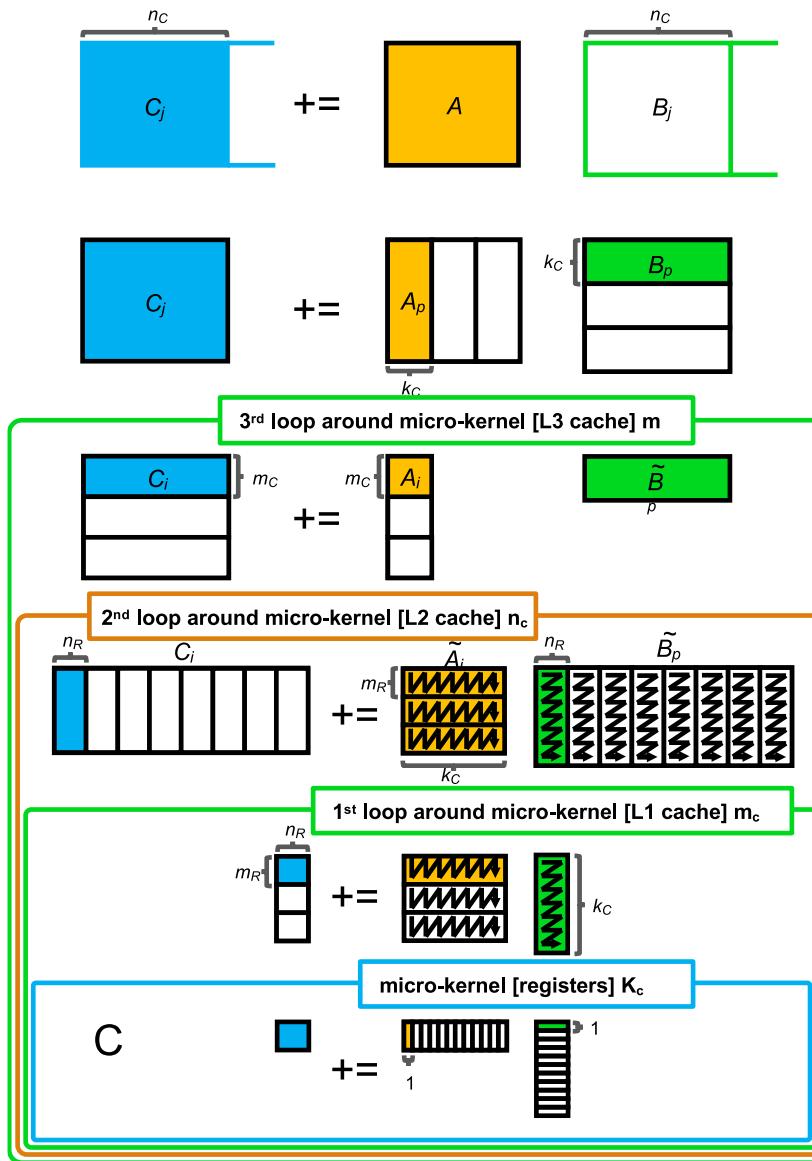
GEMM Loops

- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c \ n_c \ m$
- $m_0 \ n_0 \ k_0 \ m_1 \ n_1 \ m_2$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
    for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
         $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
        for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
             $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
        for  $j_r = 0, \dots, n_c - 1$  in steps of  $n_r$ 
            for  $i_r = 0, \dots, m_c - 1$  in steps of  $m_r$ 
                for  $p_r = 0, \dots, k_c - 1$  in steps of 1
                     $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
                     $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
                     $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 

```



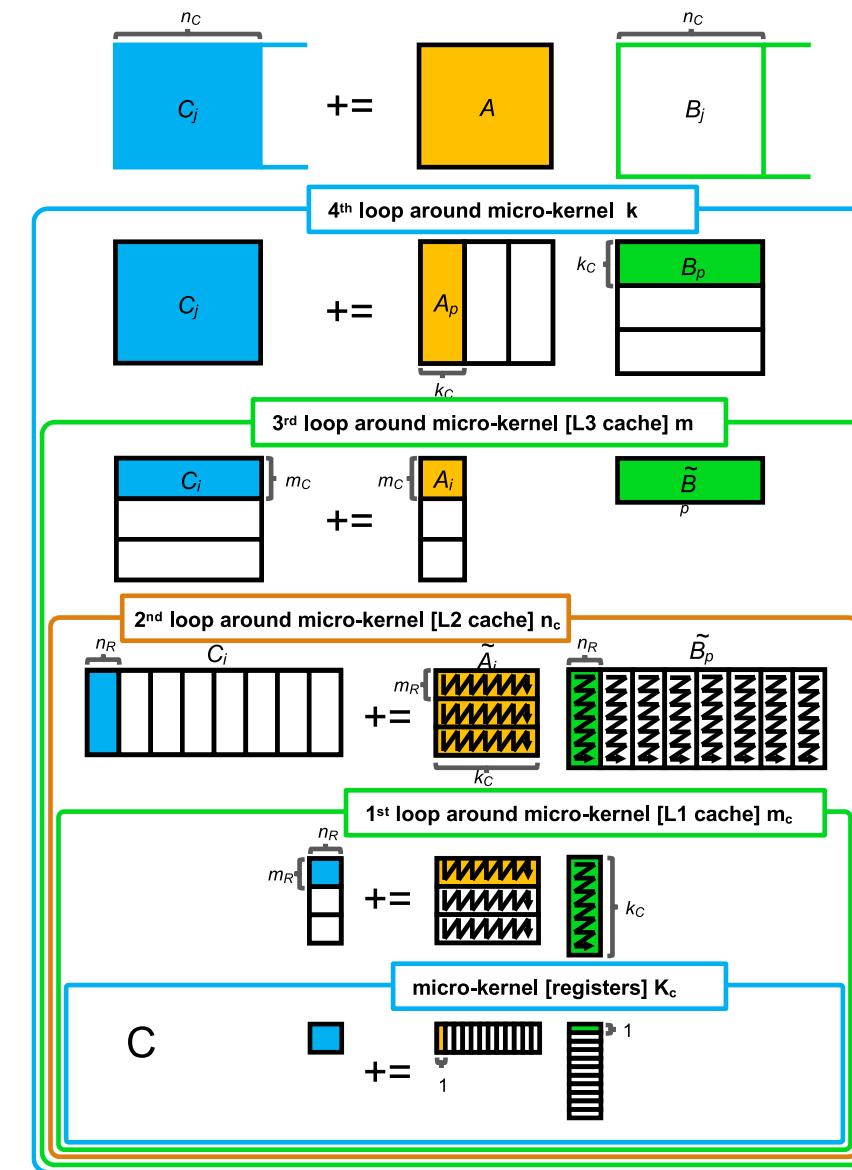
Recap GEMM Loops

- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c \ n_c \ m \ k$
- $m_0 \ n_0 \ k_0 \ m_1 \ n_1 \ m_2 \ k_1$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
    for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
         $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
        for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
             $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
    for  $j_r = 0, \dots, n_r - 1$  in steps of  $n_r$ 
        for  $i_r = 0, \dots, m_r - 1$  in steps of  $m_r$ 
            for  $p_r = 0, \dots, k_c - 1$  in steps of 1
                 $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
                 $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
                 $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 

```



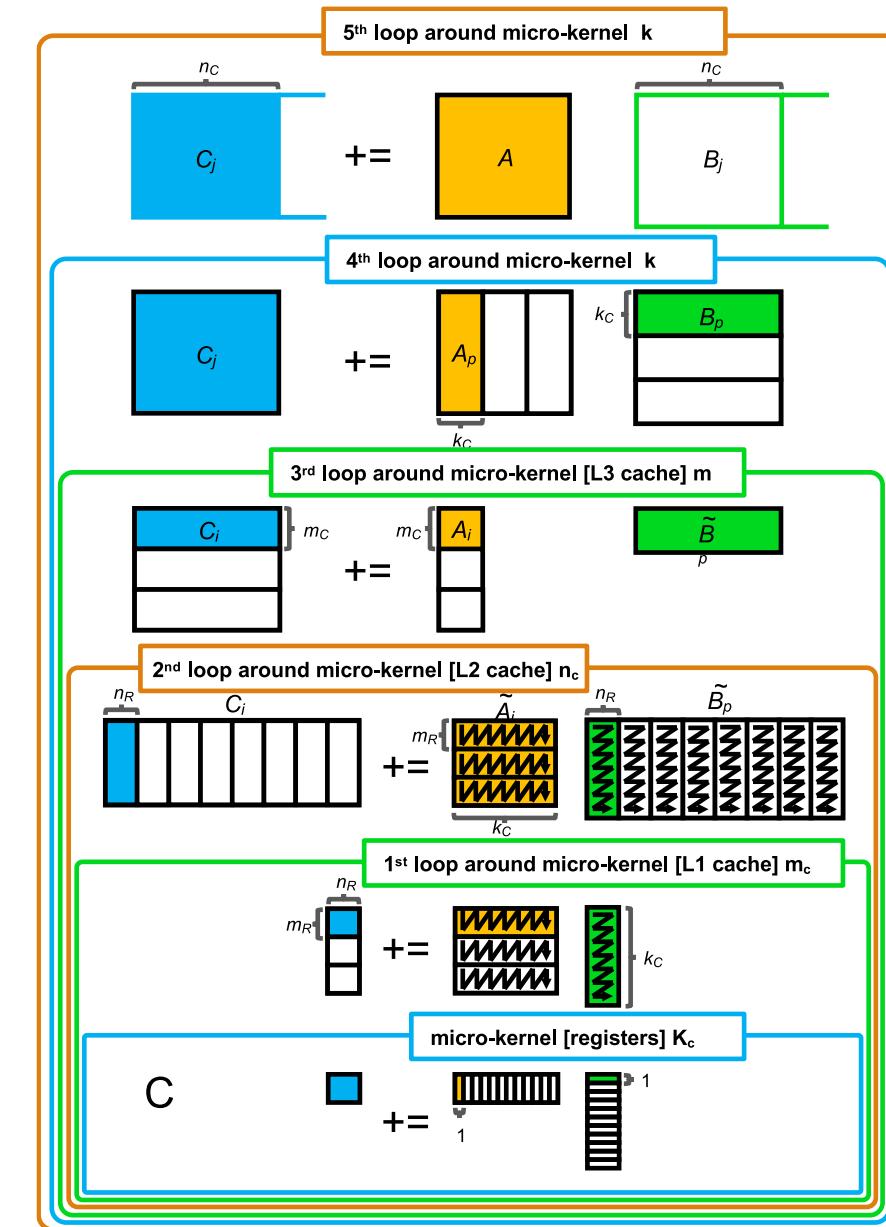
Recap GEMM Loops

- Rules for each new character
 - Buffers
 - Re-fetch rate
- $m_r \ n_r \ k_c \ m_c \ n_c \ m \ k \ n$
- $m_0 \ n_0 \ k_0 \ m_1 \ n_1 \ m_2 \ k_1 \ n_2$

```

for  $j_c = 0, \dots, n - 1$  in steps of  $n_c$ 
    for  $p_c = 0, \dots, k - 1$  in steps of  $k_c$ 
         $B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$ 
        for  $i_c = 0, \dots, m - 1$  in steps of  $m_c$ 
             $A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$ 
    for  $j_r = 0, \dots, n_r - 1$  in steps of  $n_r$ 
        for  $i_r = 0, \dots, m_r - 1$  in steps of  $m_r$ 
            for  $p_r = 0, \dots, k_c - 1$  in steps of 1
                 $C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$ 
                 $+ = A_c(i_r : i_r + m_r - 1, p_r)$ 
                 $\cdot B_c(p_r, j_r : j_r + n_r - 1)$ 

```



Design Hardware for GEMM



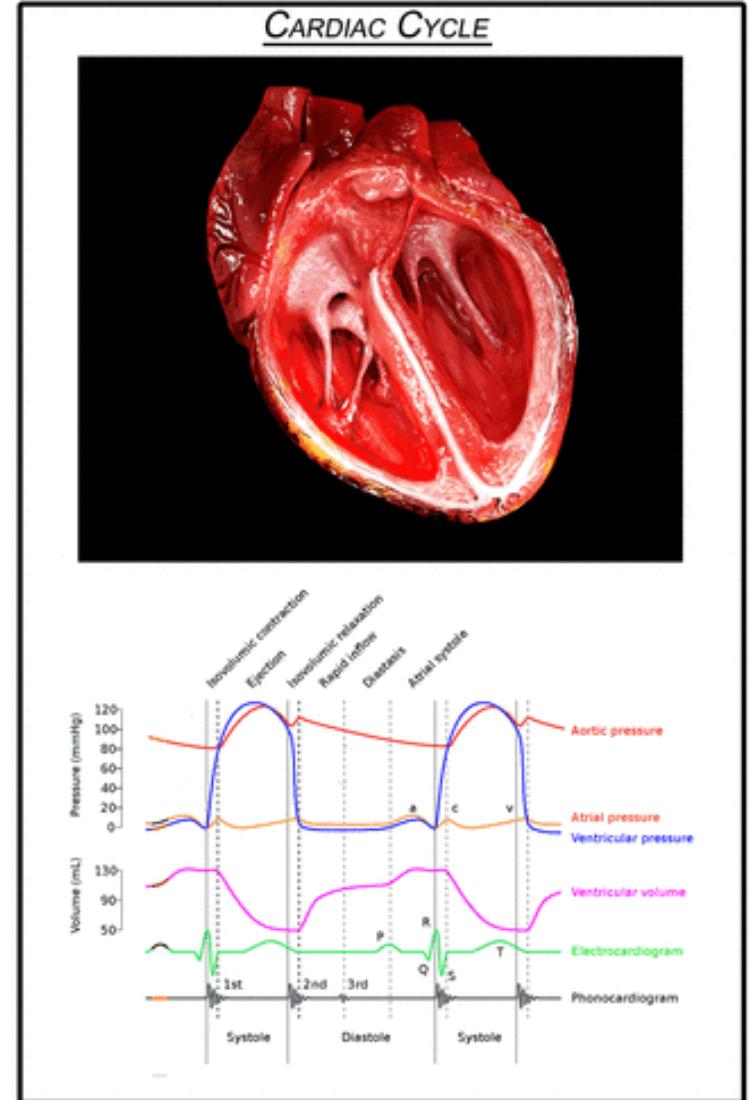
- Systolic Arrays
 - Systolic Arrays
 - GEMM Systolic solution
- Other
 - Other parallel GEMM algorithm

Outline

- Benchmarking Metrics
- GEMM Accelerator Design
 - GEMM
 - Systolic and Other Variation
 - Hardware for GEMM
- DNN Accelerator Design
- Roofline Model
- Energy

Systolic Arrays

Systolic arrays are **hardware structures built for fast and efficient operation of regular algorithms that perform the same task with different data at different time instants**. Systolic arrays replace a pipeline structure with an array of processing elements that can be programmed to perform a common operation.





Definitions of Systolic Array

“Imagine n simple processors arranged in a row or an array and connected in such a manner that each processor may exchange information with **only its neighbors** to the right and left. The processors at either end of the row are used for input and output. Such a machine constitutes the simplest example of a systolic array.”



Definitions of Systolic Array

“Systolic Arrays are **regular arrays** of simple finite state machines, where each finite state machine in the array is identical...A systolic algorithm relies on data from different directions arriving at cells in the array at regular intervals and being combined.”



Definitions of Systolic Array

“By **pipelining**, processing may proceed concurrently with input and output, and consequently overall execution time is minimized. Pipelining plus multiprocessing at each stage of a pipeline should lead to the best-possible performance.”

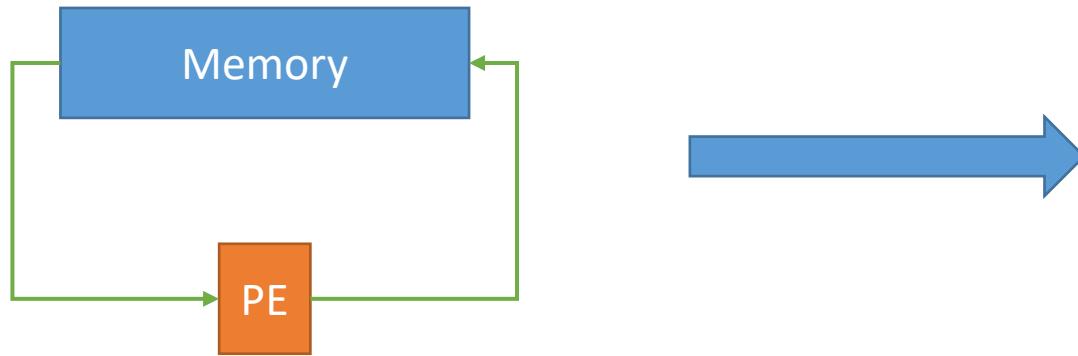
Systolic Arrays

- This is a form of pipelining, sometimes in more than one dimension
- The term **systolic** was first used in this context by H.T. Kung, then at CMU; it refers to the “pumping” action of a heart
- Machines have been constructed based on this principle, notable the iWARP, fabricated by Intel

Systolic Arrays

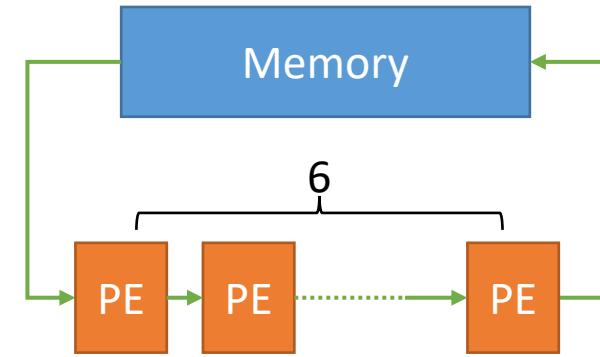
- Balancing computation with I/O

Instead of:



5 million operations
per second at most

Systolic Array



30 million operations
per second possible

Systolic Architectures

- Replace single processor with an **array of regular processing elements**
- Orchestrate data flow for **high throughput** with **less memory access**
- Different from linear pipelining
 - Nonlinear array structure
 - multidirection data flow
 - each PE may have (small) local instruction and data memory
- Different from SIMD: each PE may do something different
- Initial motivation: VLSI Application-Specific Integrated Circuits (ASICs)
- Represent algorithms directly by chips connected in regular pattern



Systolic Matrix Multiplication

- Processors are arranged in a 2-D grid
- Each processor accumulates one element of the product
- The elements of the matrices to be multiplied are **pumped through** the array

Data Preparation for Systolic

- We need to modify the input data, like so:

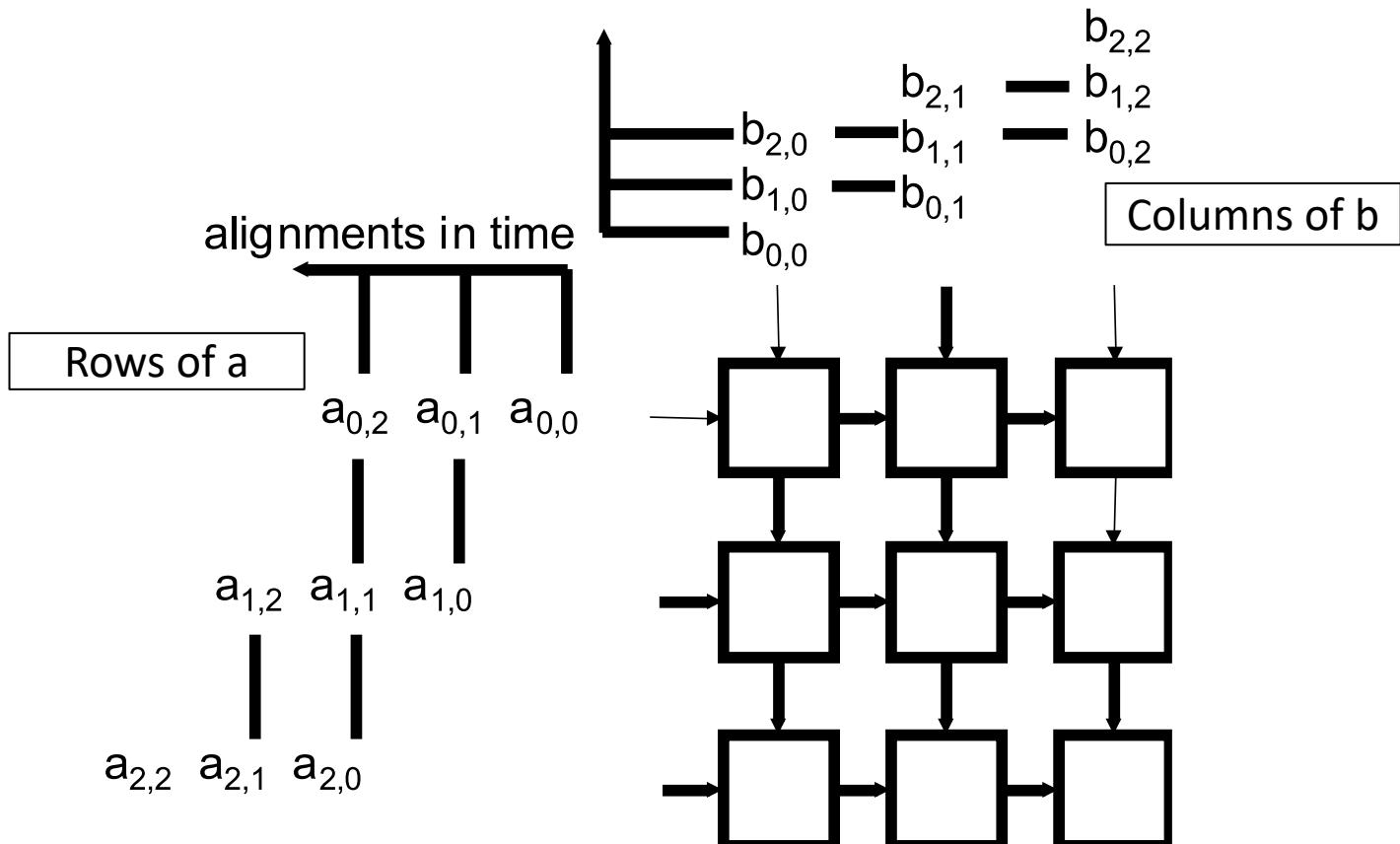
Flip columns 1 & 3 →
$$\begin{matrix} a_{13} & a_{12} & a_{11} \\ a_{23} & a_{22} & a_{21} \\ a_{33} & a_{32} & a_{31} \end{matrix}$$

Flip rows 1 & 3 →
$$\begin{matrix} b_{31} & b_{32} & b_{33} \\ b_{21} & b_{22} & b_{23} \\ b_{11} & b_{12} & b_{13} \end{matrix}$$

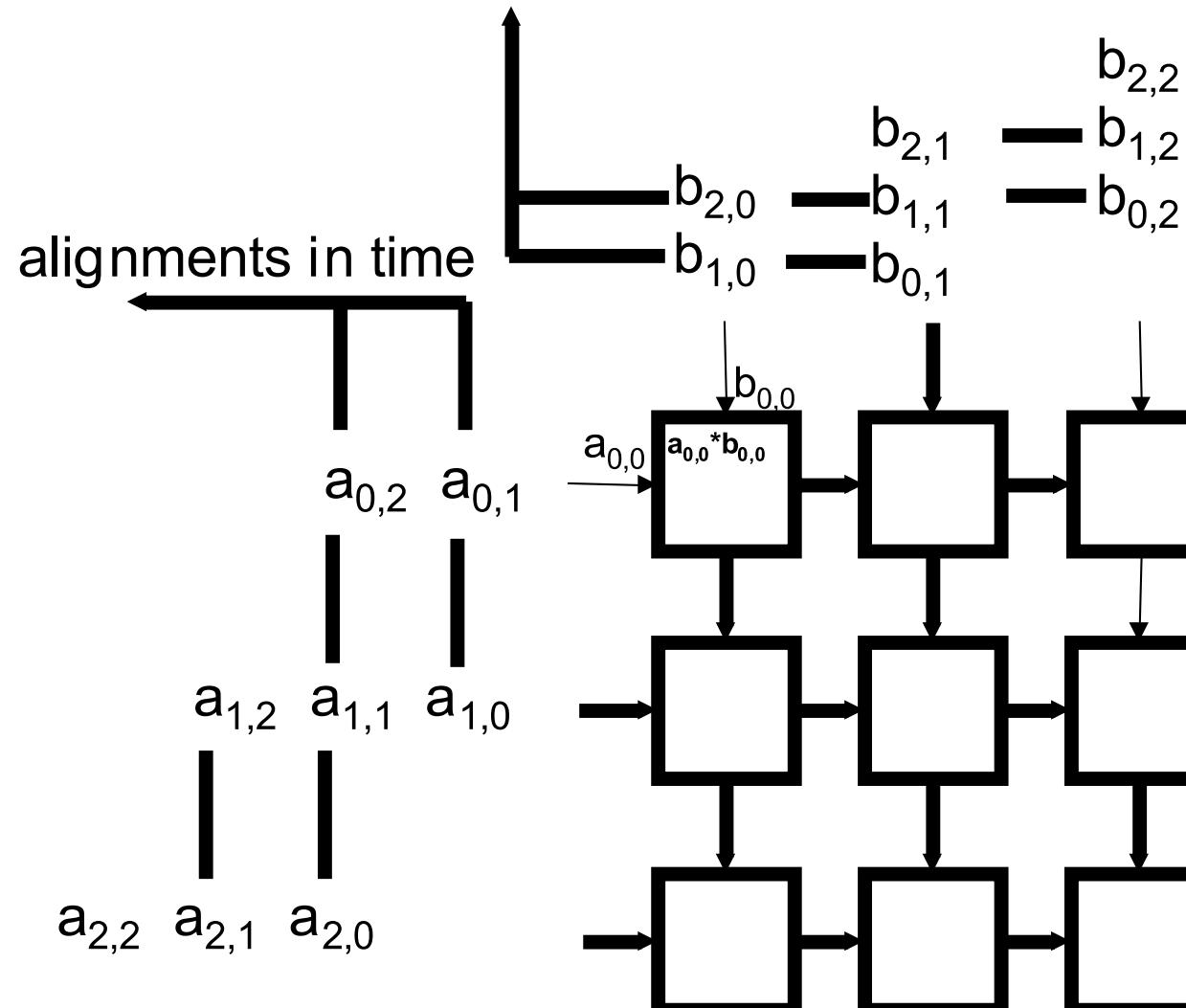
and finally stagger the data sets for input.

Systolic Matrix Multiplication

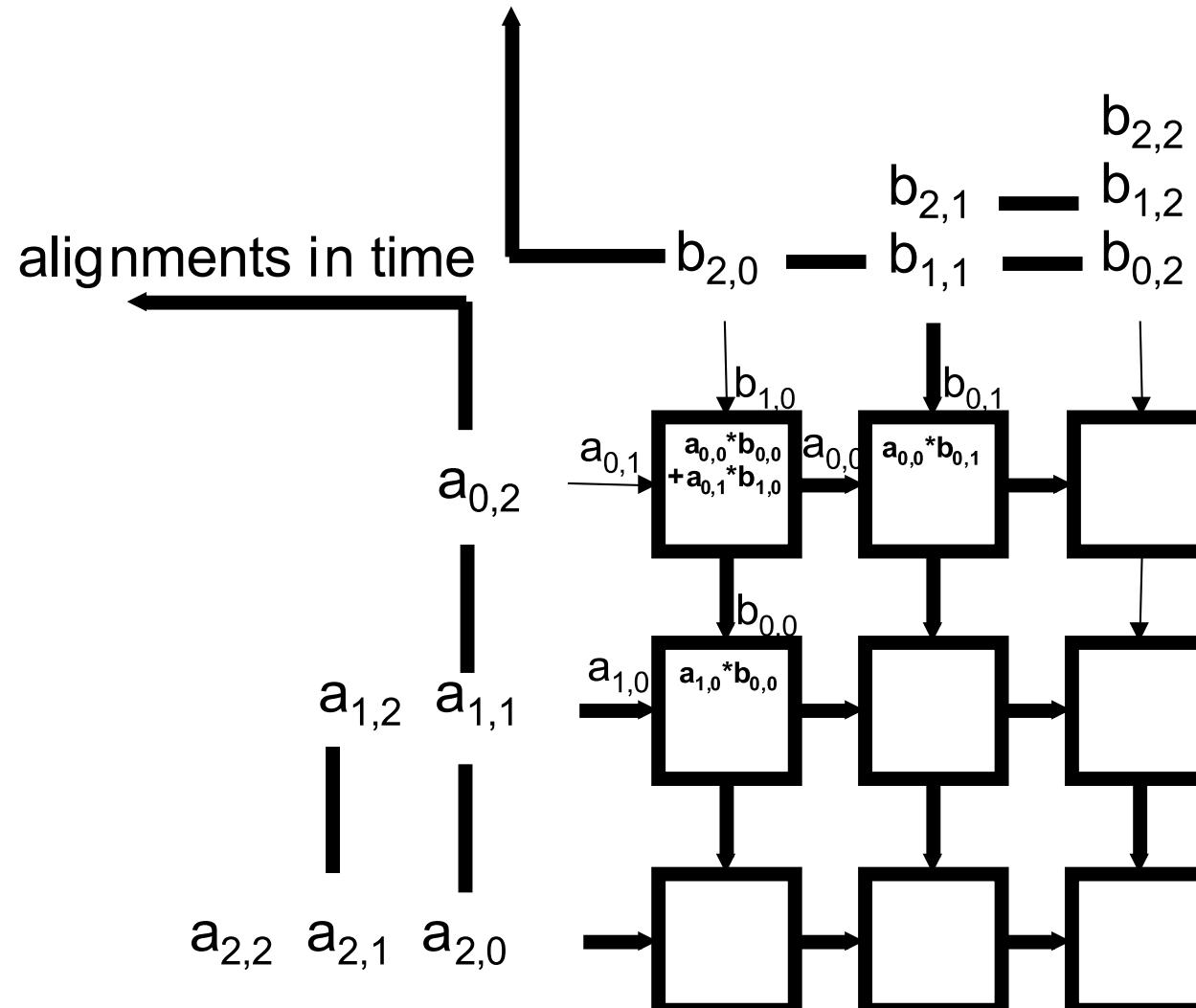
- 3x3 matrices multiplications
- Processors arranged in a 2-D grid
- Each processor accumulates one element of the product



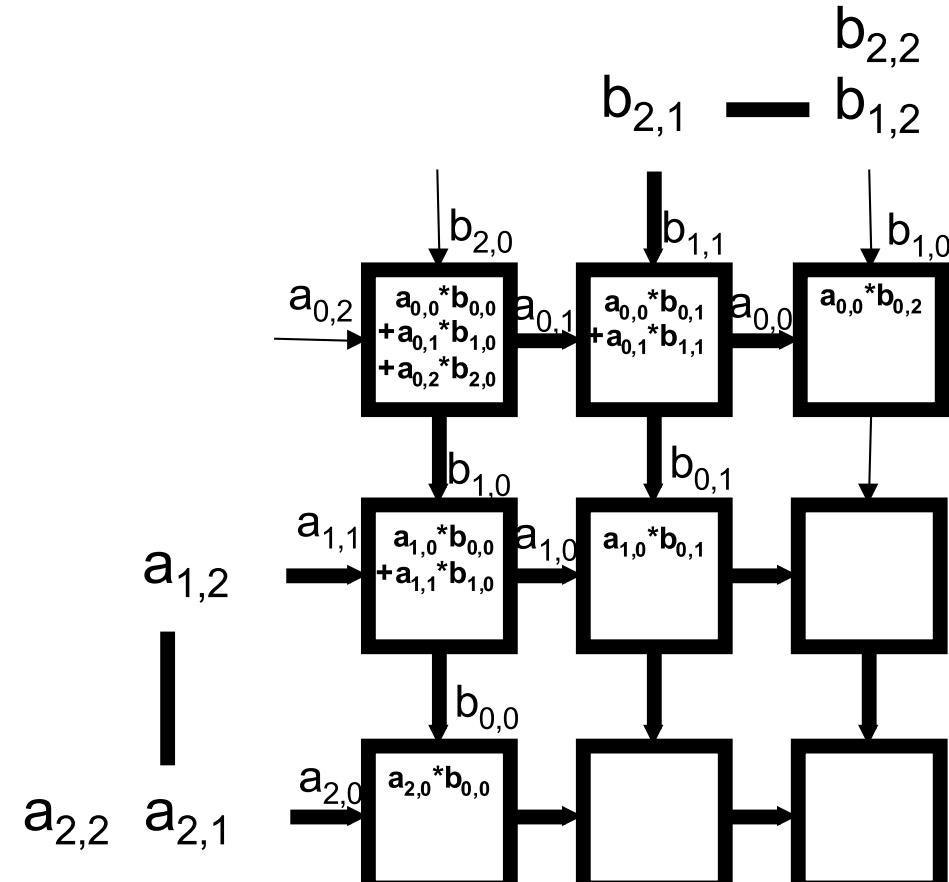
Systolic Matrix Multiplication



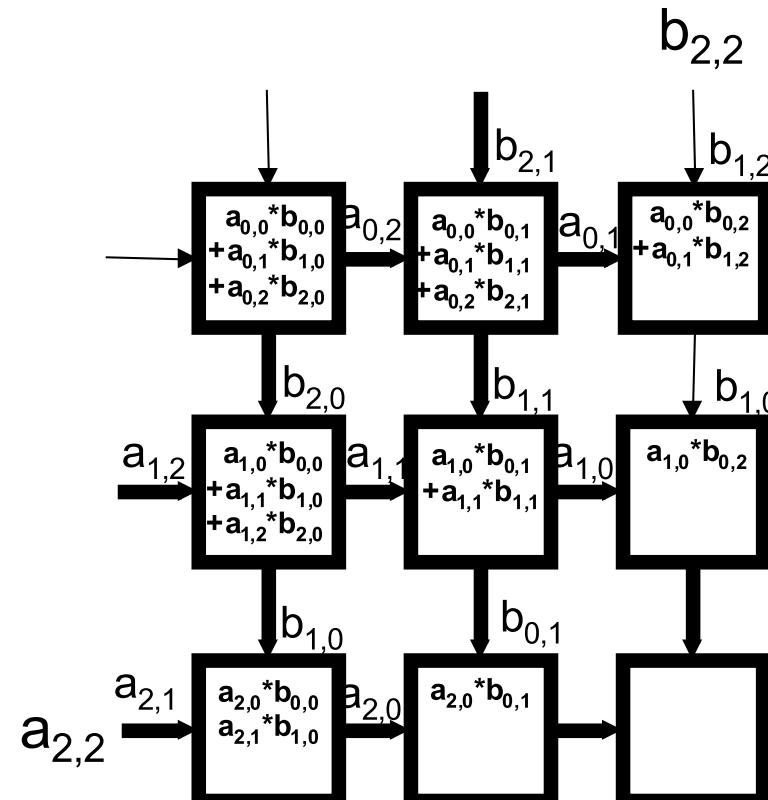
Systolic Matrix Multiplication



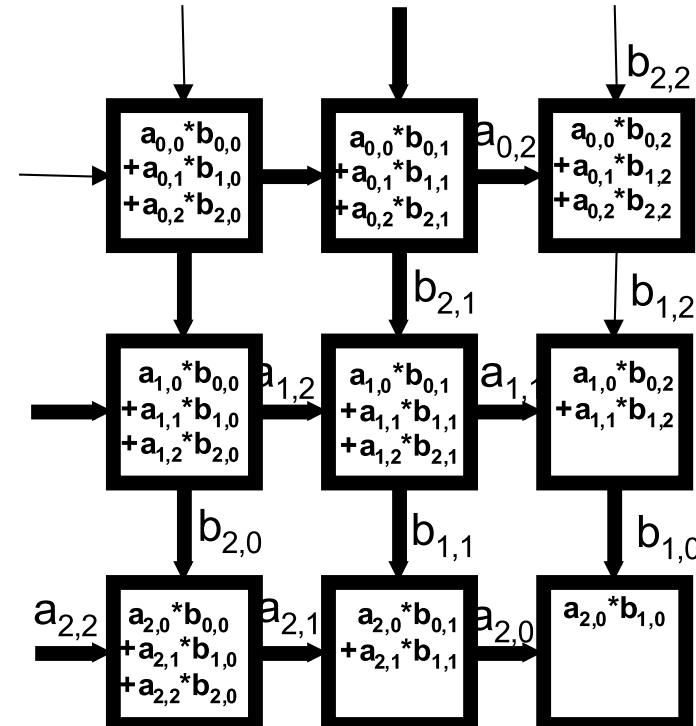
Systolic Matrix Multiplication



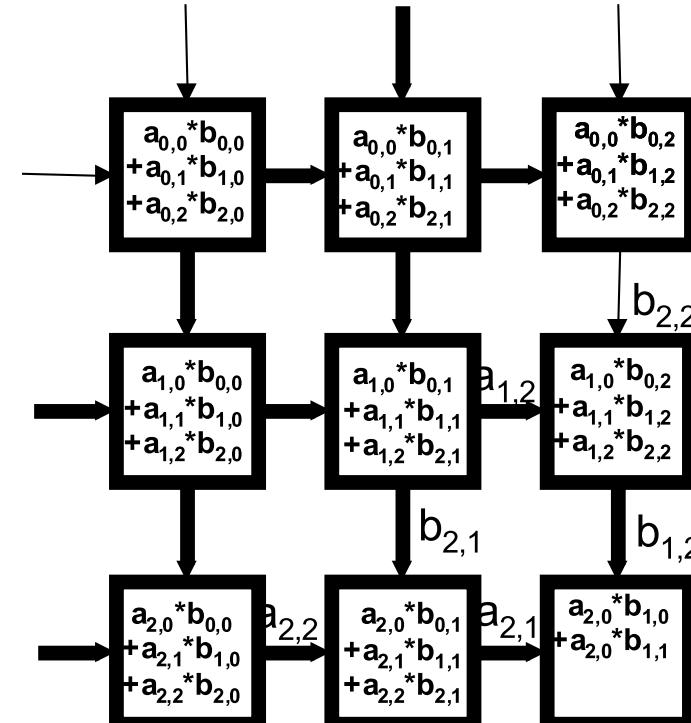
Systolic Matrix Multiplication



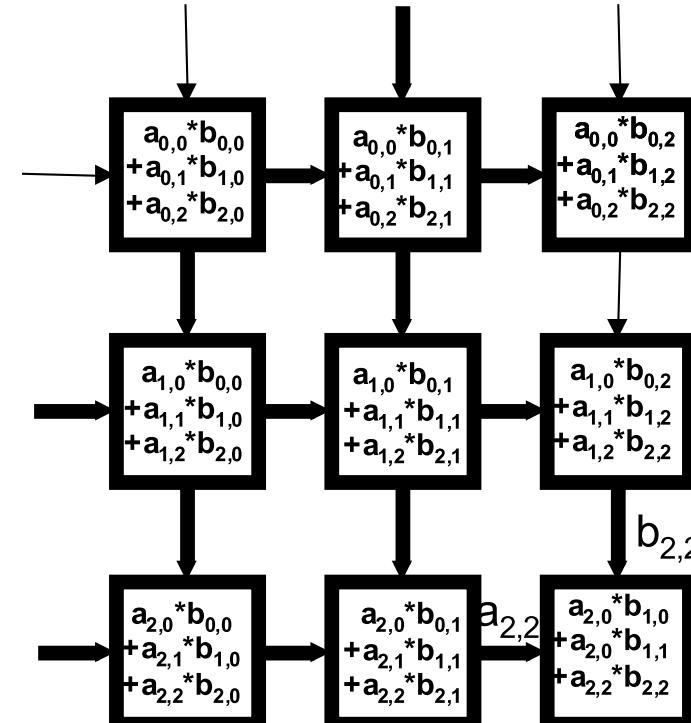
Systolic Matrix Multiplication



Systolic Matrix Multiplication



Systolic Matrix Multiplication



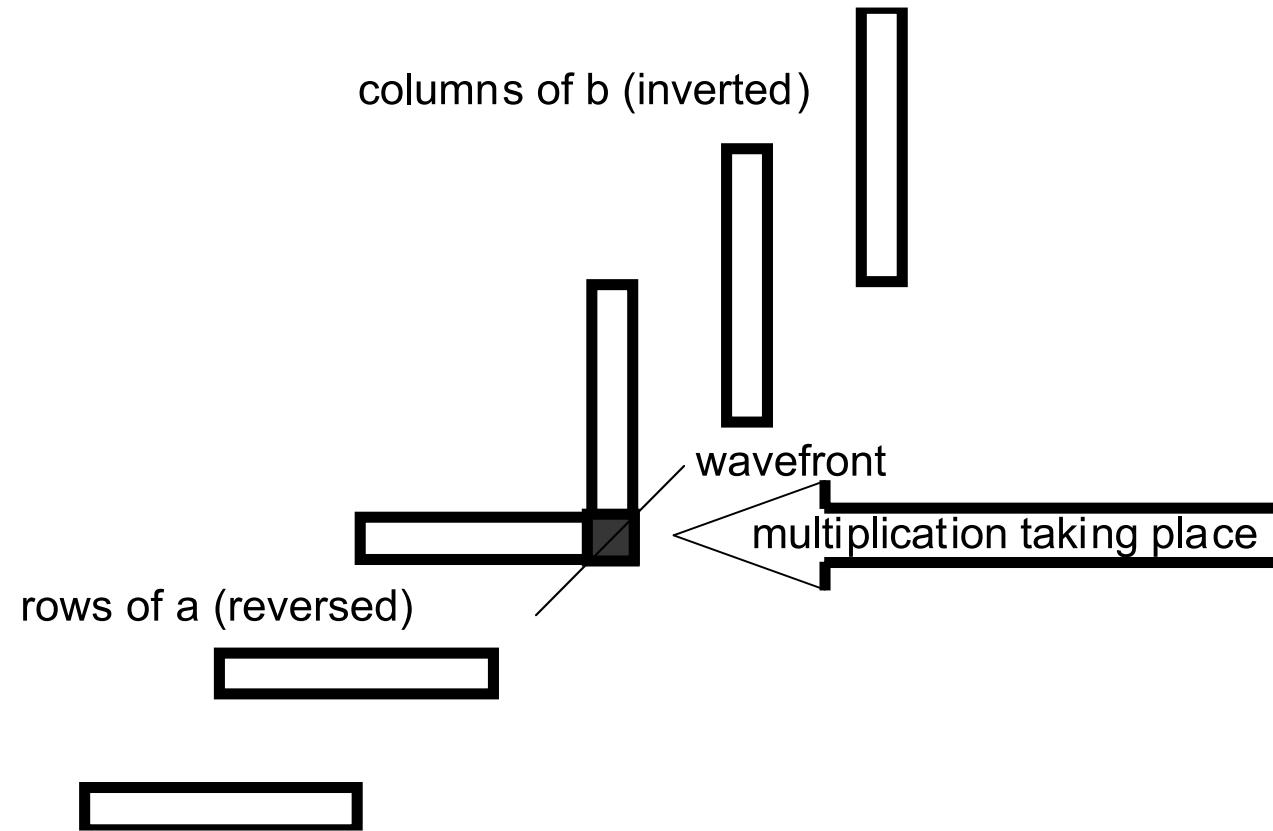


Strips View of Systolic Array

- Let's take another view of systolic multiplication
 - Consider the rows and columns of the matrices to be multiplied as **strips** that are **slide past each other**.
 - The strips are staggered so that the correct elements are multiplied at each time step.

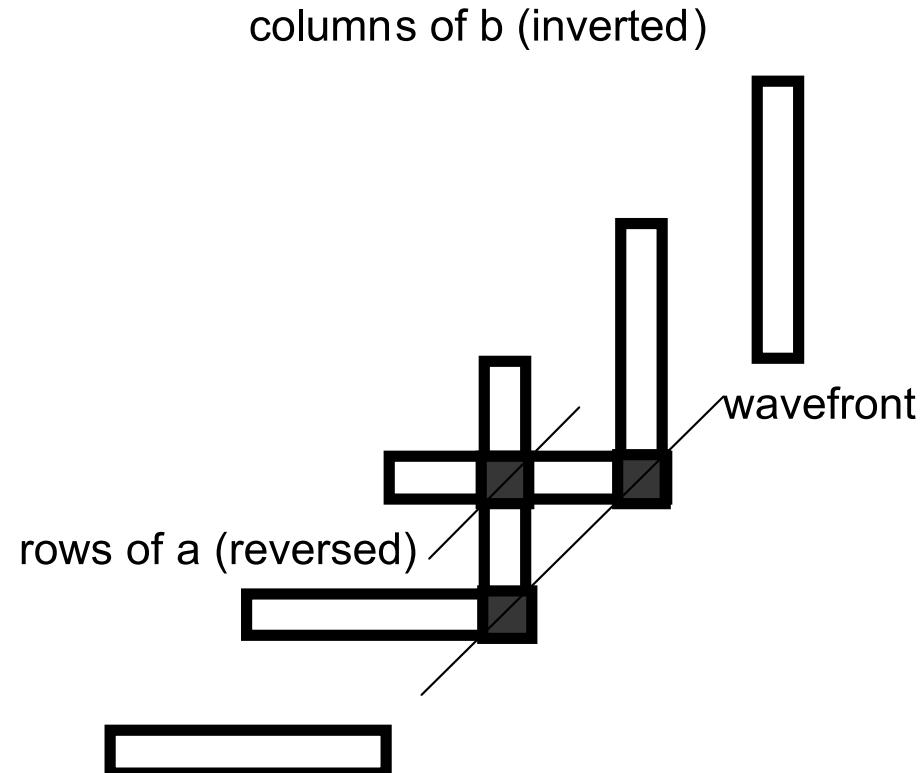
Strips View of Systolic Array

- First step



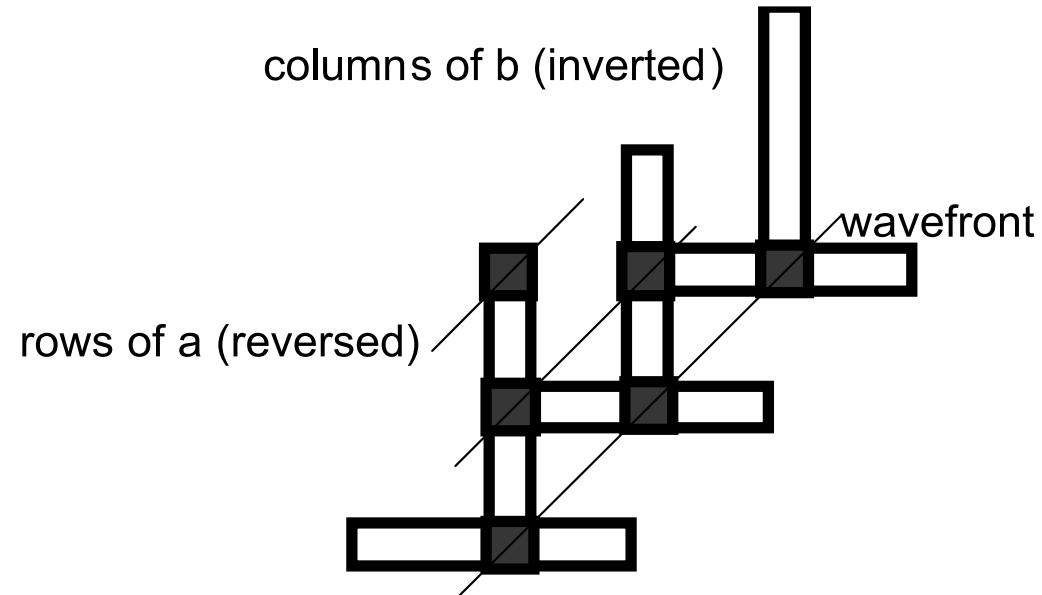
Strips View of Systolic Array

- Second step



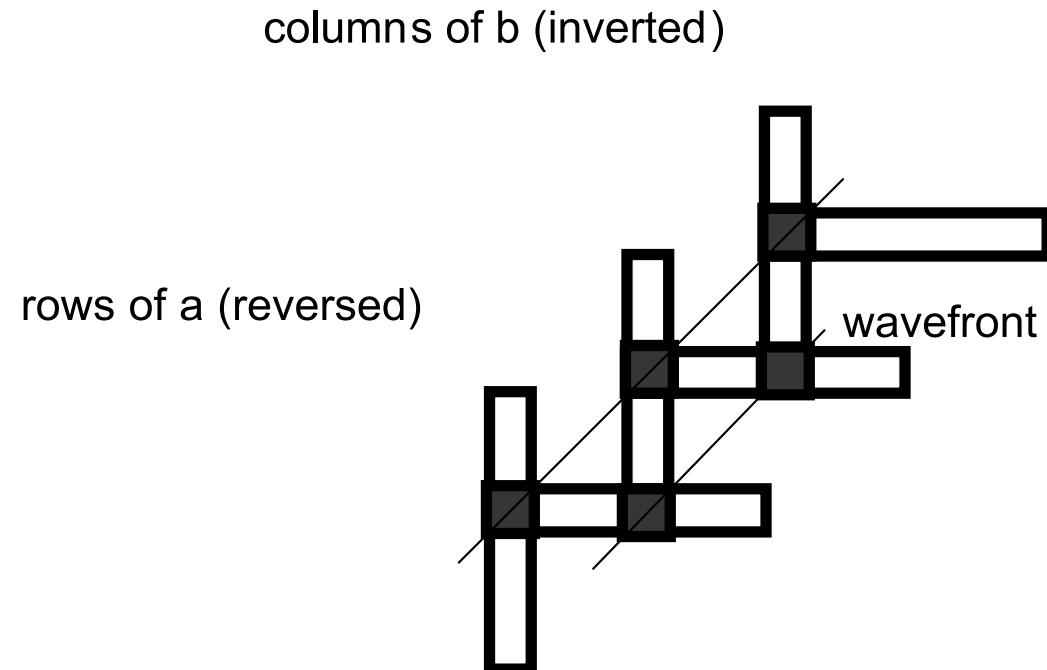
Strips View of Systolic Array

- Third step



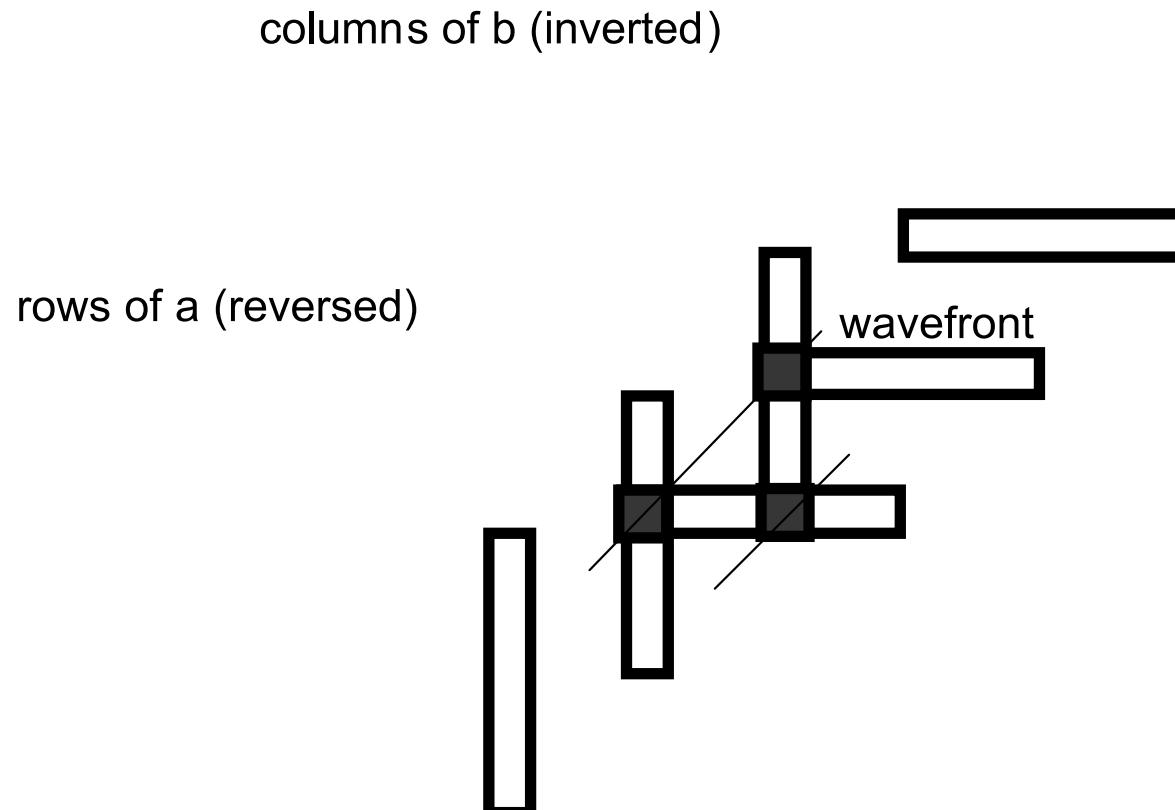
Strips View of Systolic Array

- Fourth step



Strips View of Systolic Array

- Fifth step

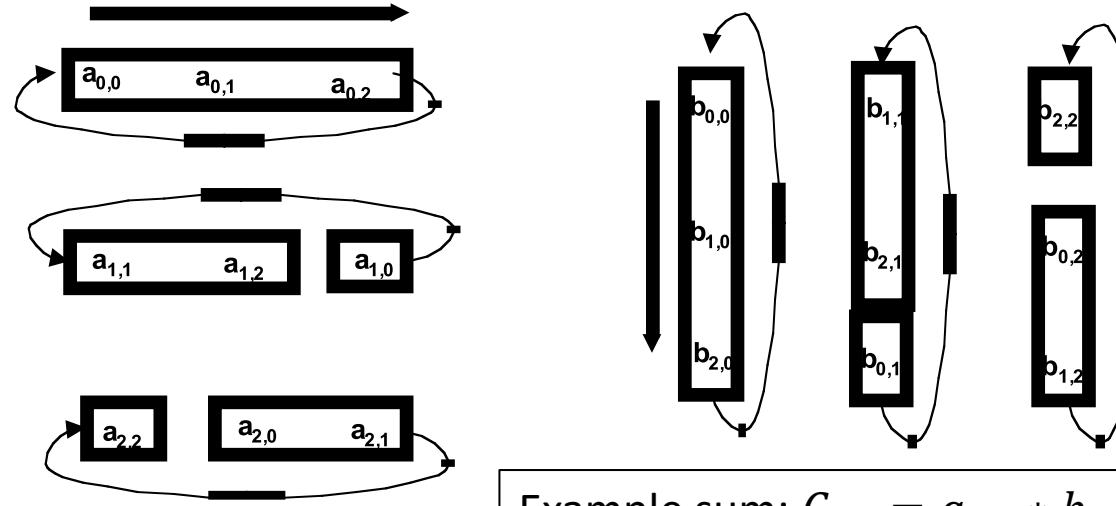


Cannon's Method

- Rather than have some processors idle
 - wrap the array rows and columns so that **every processor is doing something on each step.**
- In other words, rather than feeding in the elements, they are **rotated around**
- Starting in an initially staggered position as in the systolic model.
- We also change the order of products slightly, to make it correspond to more natural storage by rows and columns
- Not including the loading which can be done in parallel for a time of 1, this technique is effectively N.

Cannon Variation

Note that the **a diagonal** is in the **left column** and the **b diagonal** is in the **top row**.



Products computed at each step:

Step 1

$a_{0,0} * b_{0,0}$	$a_{0,1} * b_{1,1}$	$a_{0,2} * b_{2,2}$
$a_{1,1} * b_{1,0}$	$a_{1,2} * b_{2,1}$	$a_{1,0} * b_{0,2}$
$a_{2,2} * b_{2,0}$	$a_{2,0} * b_{0,1}$	$a_{2,1} * b_{1,2}$

Example sum: $C_{0,2} = a_{0,2} * b_{2,2} + a_{0,1} * b_{1,2} + a_{0,0} * b_{0,2}$

Step 2

$a_{0,2} * b_{2,0}$	$a_{0,0} * b_{0,1}$	$a_{0,1} * b_{1,2}$
$a_{1,0} * b_{0,0}$	$a_{1,1} * b_{1,1}$	$a_{1,2} * b_{2,2}$
$a_{2,1} * b_{1,0}$	$a_{2,2} * b_{2,1}$	$a_{2,0} * b_{0,2}$

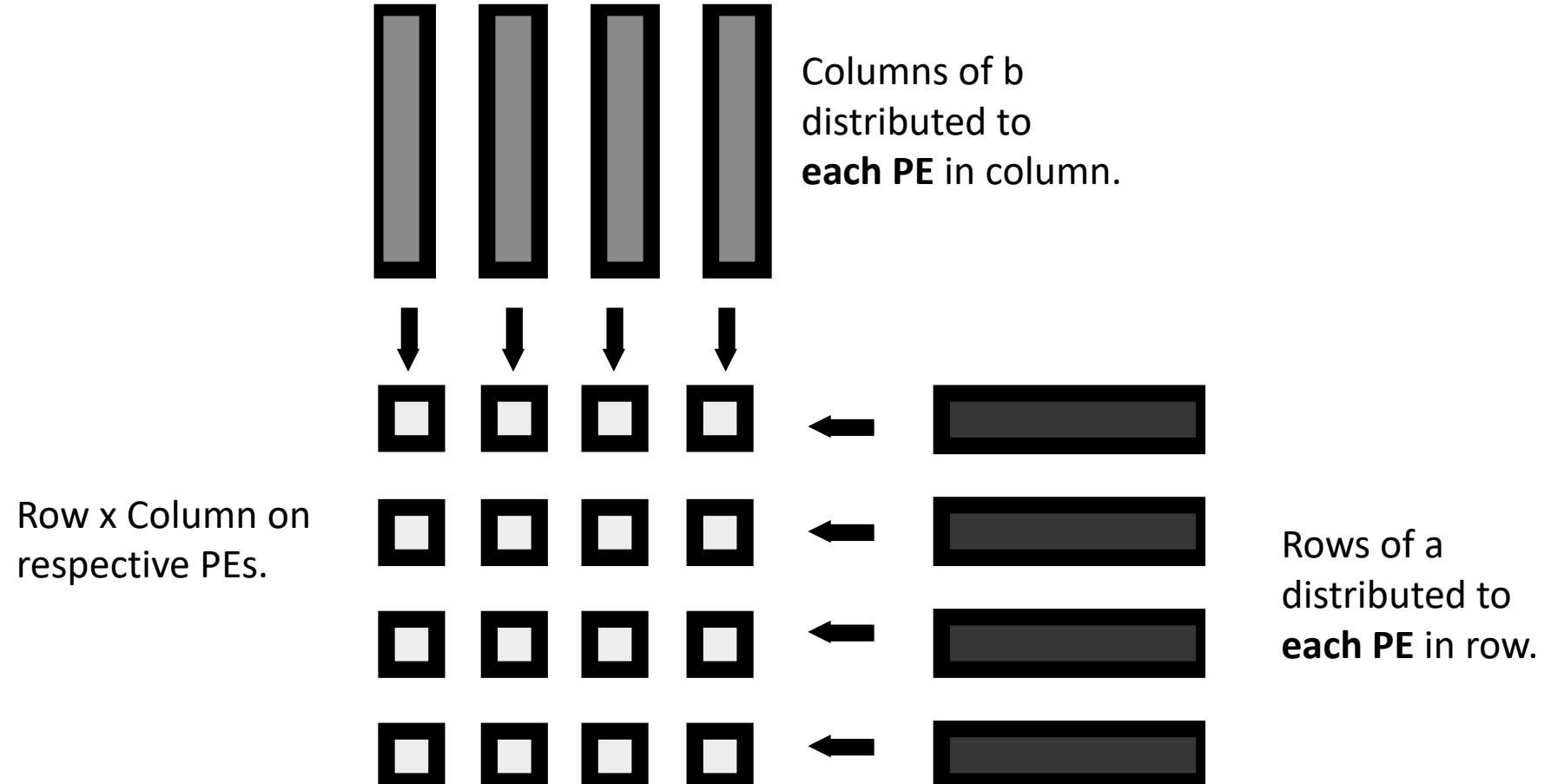
Step 3

$a_{0,1} * b_{1,0}$	$a_{0,2} * b_{2,1}$	$a_{0,0} * b_{0,2}$
$a_{1,2} * b_{2,0}$	$a_{1,0} * b_{0,1}$	$a_{1,1} * b_{1,2}$
$a_{2,0} * b_{0,0}$	$a_{2,1} * b_{1,1}$	$a_{2,2} * b_{2,2}$

Application of Cannon's Technique

- Consider matrix multiplication of 2 $n \times n$ matrices on a distributed memory machine, on say, n^2 processing elements.
- An obvious way to compute is to **think of the PE's as a matrix**, with each computing one element of the product.
- We would send each row of the matrix to n processors and each column to n processors.
- **In effect, in the obvious way, each matrix is stored a total of n times.**

Obvious Matrix Multiply



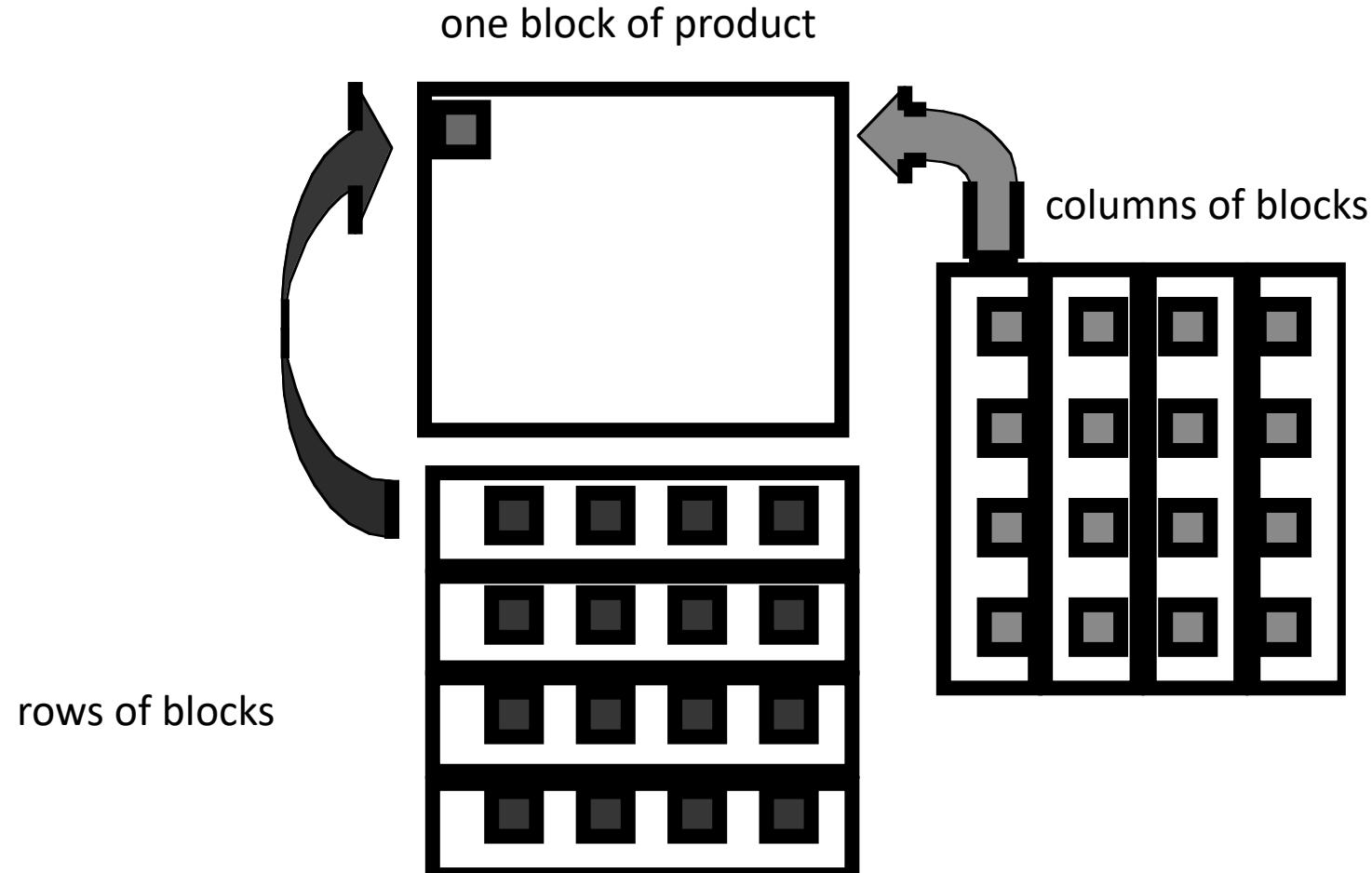
Cannon's Method

- Cannon's method **avoids storing each matrix n times**, instead cycling (“piping”) the elements through the PE array.
 - It is sometimes called the “pipe-roll” method.
- The problem is that this cycling is typically **too fine-grain to be useful** for element-by-element multiply.

Partitioned Multiplication

- Partitioned multiplication **divides the matrices into blocks**
- It can be shown that **multiplying the individual blocks** as if elements of matrices themselves gives the matrix product.

Block Multiplication





Cannon's Method is Fine for Block Multiplication

- The blocks are aligned initially as the elements were in our description
- At each step, entire blocks are transmitted down and to the left of neighboring PE's
- Memory space is conserved

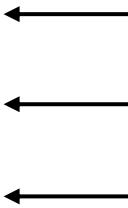
Fox's Algorithm

- Also for block matrix multiplication, it has a resemblance to Cannon's algorithm.
- The difference is that on each cycle:
 - A row block is **broadcast** to every other processor in the row.
 - The column blocks are rolled cyclically

Fox's Algorithm

Step 1

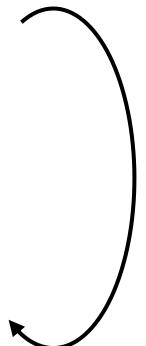
$a_{0,0} * b_{0,0}$	$a_{0,0} * b_{0,1}$	$a_{0,0} * b_{0,2}$
$a_{1,1} * b_{1,0}$	$a_{1,1} * b_{1,1}$	$a_{1,1} * b_{1,2}$
$a_{2,2} * b_{2,0}$	$a_{2,2} * b_{2,1}$	$a_{2,2} * b_{2,2}$



A different row block of a is broadcast in each step.

Step 2

$a_{0,1} * b_{1,0}$	$a_{0,1} * b_{1,1}$	$a_{0,1} * b_{1,2}$
$a_{1,2} * b_{2,0}$	$a_{1,2} * b_{2,1}$	$a_{1,2} * b_{2,2}$
$a_{2,0} * b_{0,0}$	$a_{2,0} * b_{0,1}$	$a_{2,0} * b_{0,2}$

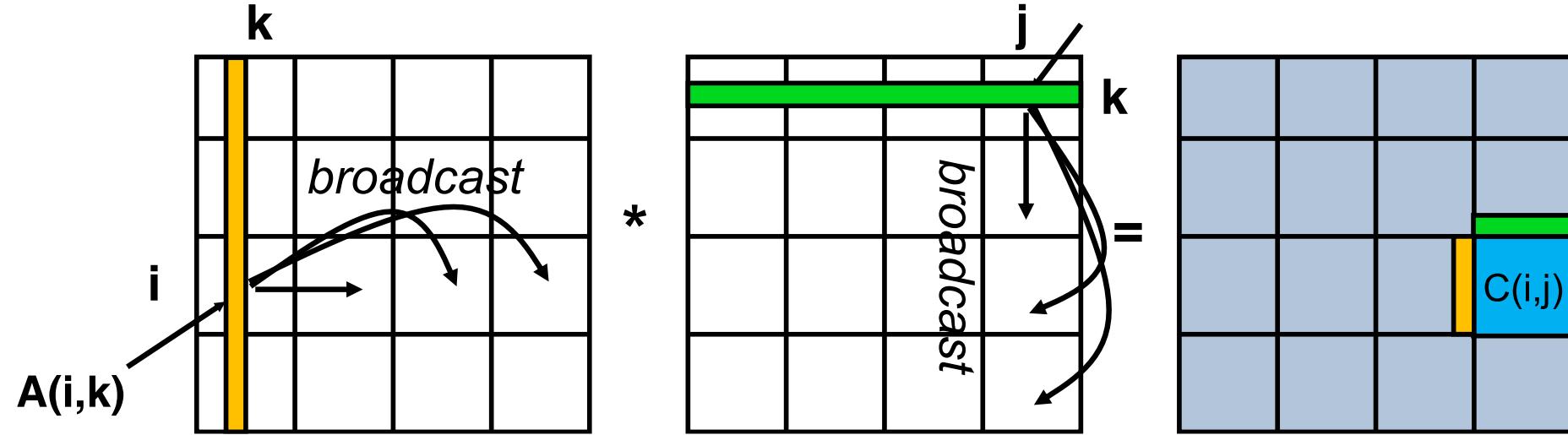


b columns are rolled

Step 3

$a_{0,2} * b_{2,0}$	$a_{0,2} * b_{2,1}$	$a_{0,2} * b_{2,2}$
$a_{1,0} * b_{0,0}$	$a_{1,0} * b_{0,1}$	$a_{1,0} * b_{0,2}$
$a_{2,1} * b_{1,0}$	$a_{2,1} * b_{1,1}$	$a_{2,1} * b_{1,2}$

SUMMA Algorithm



```

1 for (k = 0 ~ n/b-1)
2
3   spatial_for(i = 1 ~ pr)
4     owner of A(i,k) broadcasts it to whole processor row
5   spatial_for(ij= 1 ~ pc)
6     owner of B(k,j) broadcasts it to whole processor column
7   Receive A(i,k) into Acol
8   Receive B(k,j) into Brow
9   C_myproc = C_myproc + Acol * Brow

```

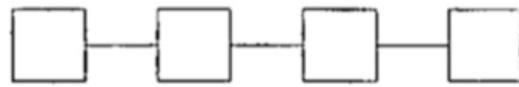
where b is the block size
b= # cols in $A(i,k)$ and # rows in $B(k,j)$
in parallel
in parallel

Remarks

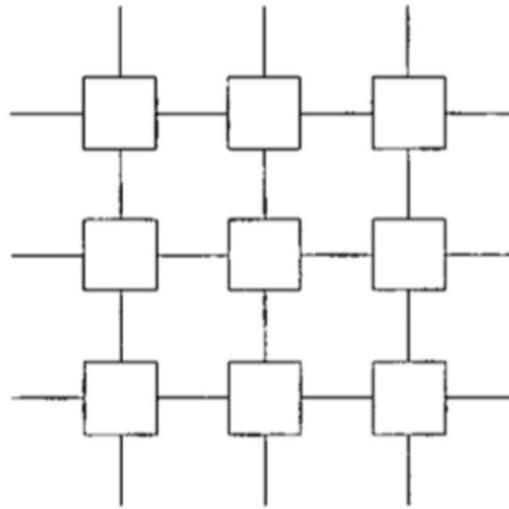


- Cannon's algorithms
 - Roll-roll-multiply
- Fox's algorithm
 - Broadcast-roll-multiply
- SUMMA
 - Broadcast-broadcast-multiply

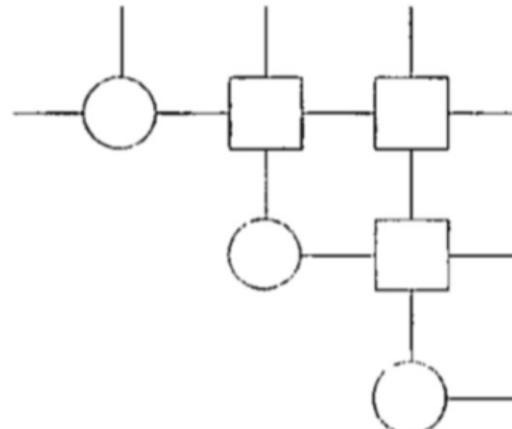
Systolic Array Architectures



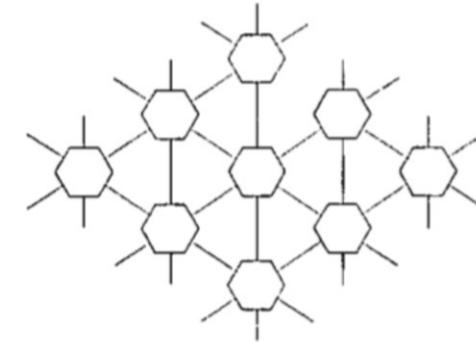
Linear Array



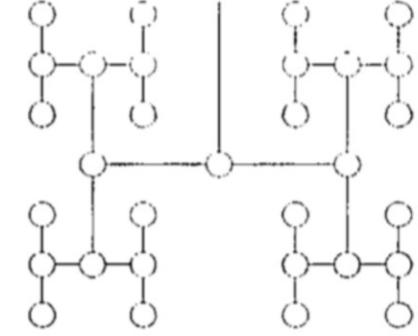
Orthogonal Array



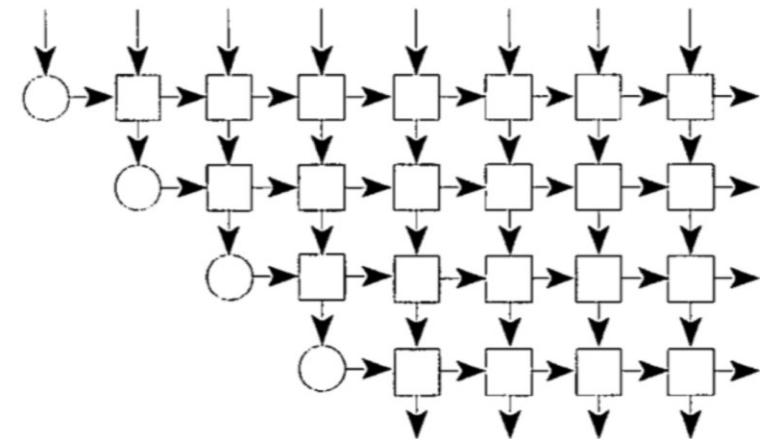
Triangular Array



Hexagonal Array

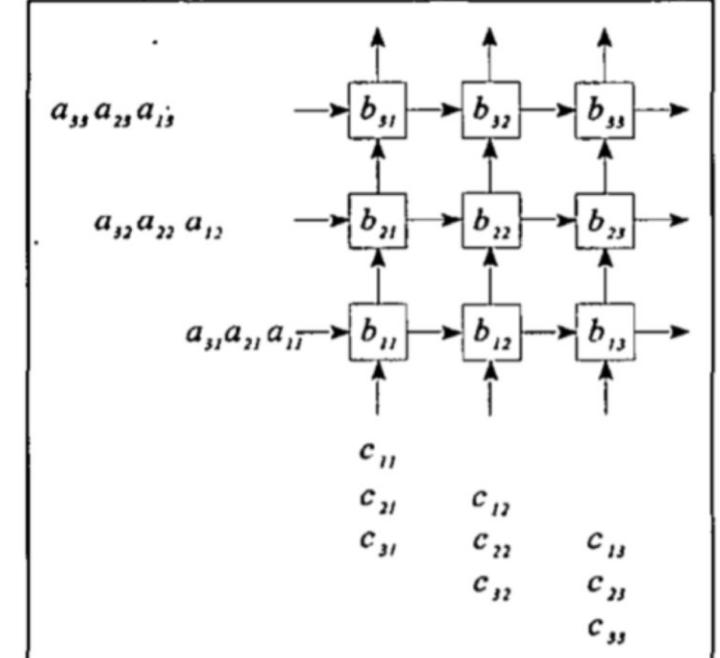
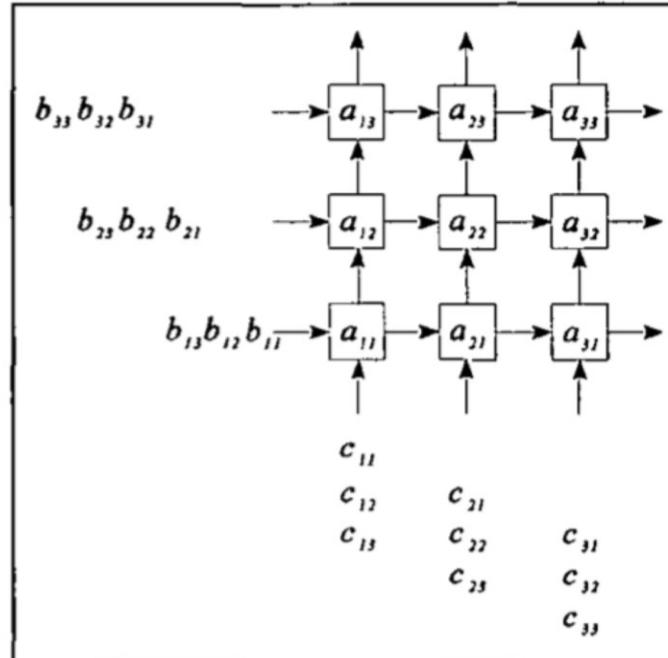
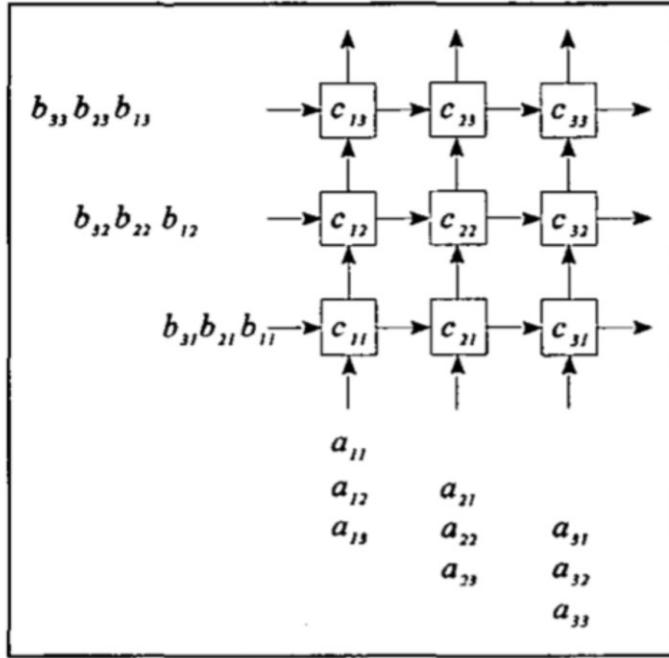


Binary Tree



Trapezoid

$[C] = [A] \times [B]$ Variation



- Keep C resident
- Stream A,B

- Keep A resident
- Stream B,C

- Keep B resident
- Stream A,C

Outline

- Benchmarking Metrics
- GEMM Accelerator Design
 - GEMM
 - Systolic and Other Variation
 - Hardware for GEMM
- DNN Accelerator Design
- Roofline Model
- Energy

Systolic GEMM Google TPU v1

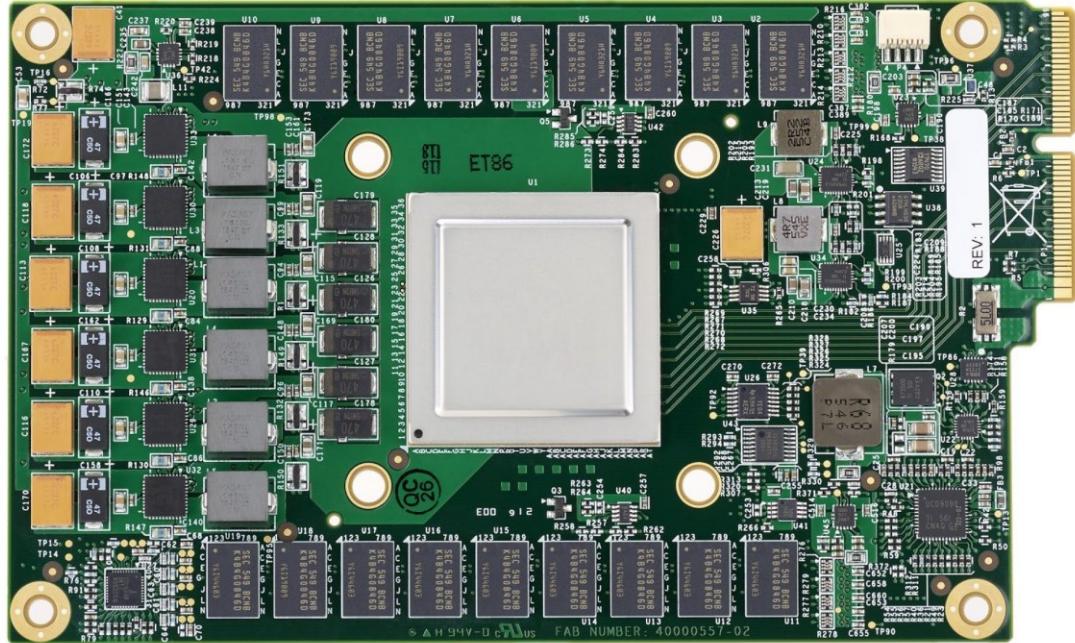


Figure 3. TPU Printed Circuit Board. It can be inserted in the slot for an SATA disk in a server, but the card uses PCIe Gen3 x16.

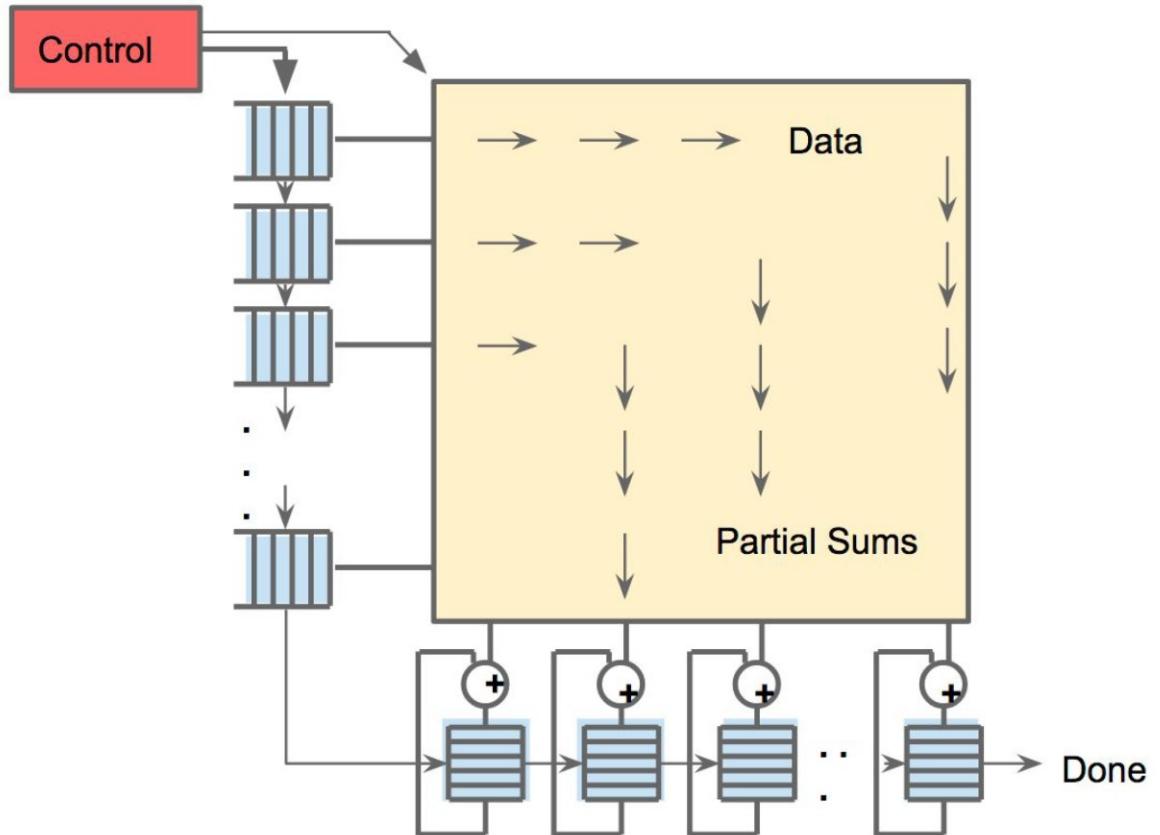


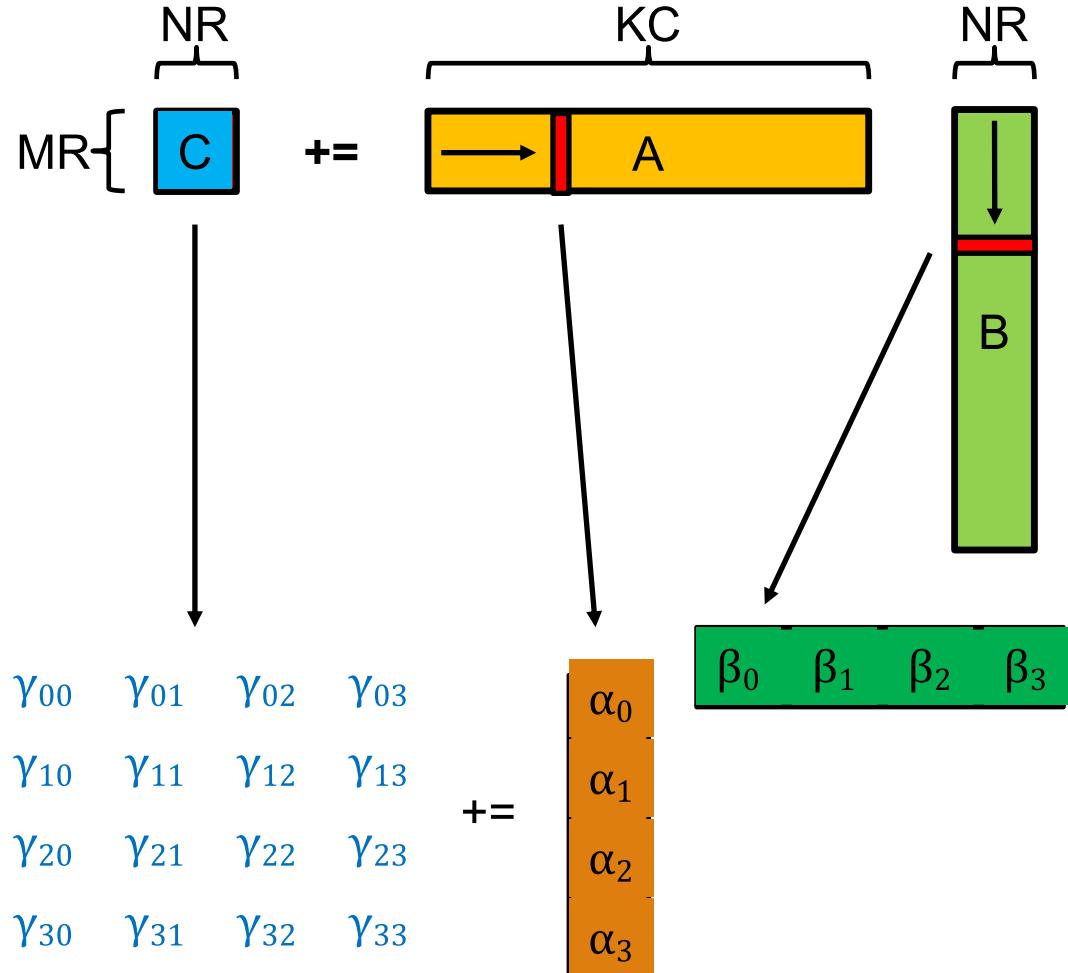
Figure 4. Systolic data flow of the Matrix Multiply Unit. Software has the illusion that each 256B input is read at once, and they instantly update one location of each of 256 accumulator RAMs.

Designing GEMM Accelerator



- What If we wanted to design a GEMM accelerator?
- What Algorithm to pick?
 - Systolic
 - Fox
 - Canon
 - SUMMA
- What does the architecture look like?
- What was the GEMM micro kernel?
- What is the rank-1 update?

Recap GEMM micro-kernel



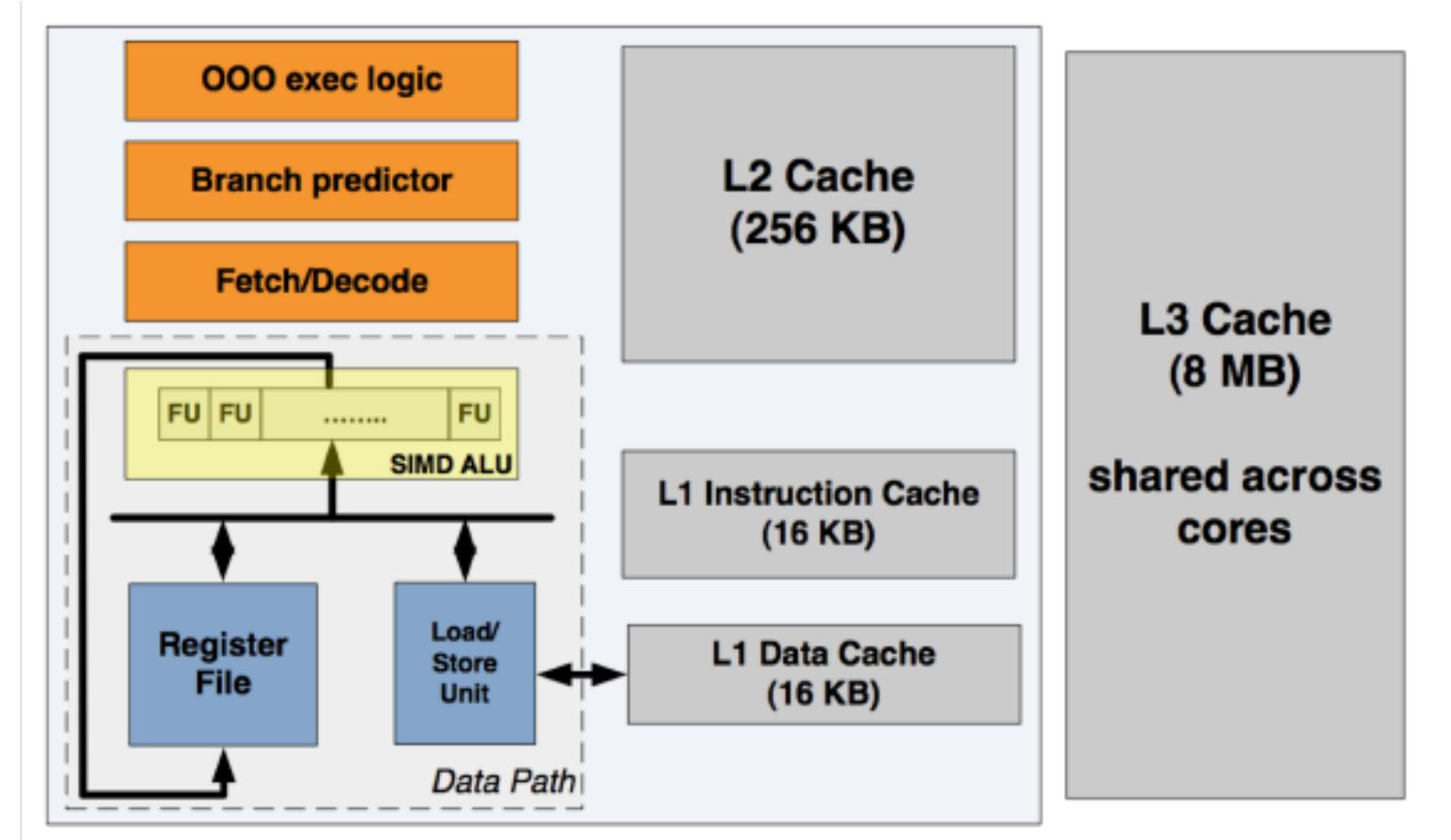
```

for ( 0 to NC: NR )
  for ( 0 to MC: MR )
    for ( 0 to KC: 1 )
      // block-dot product
    endfor
  endfor
endfor
  
```

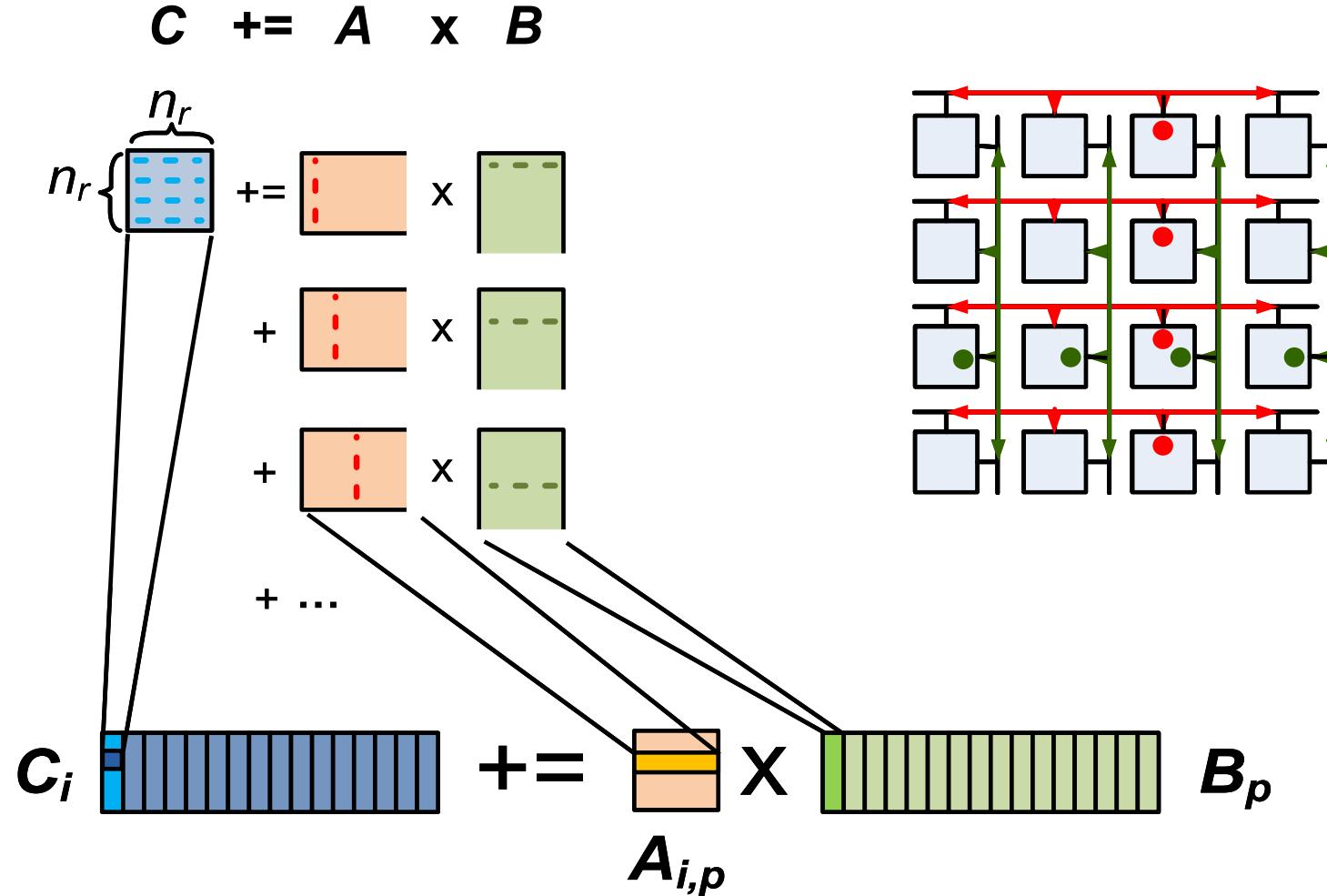
- Typical micro-kernel loop iteration (“block-dot product”)
 - Load column of packed A
 - Load row of packed B
 - Compute outer product
 - Update C (kept in registers)

Custom vs. General Purpose Design

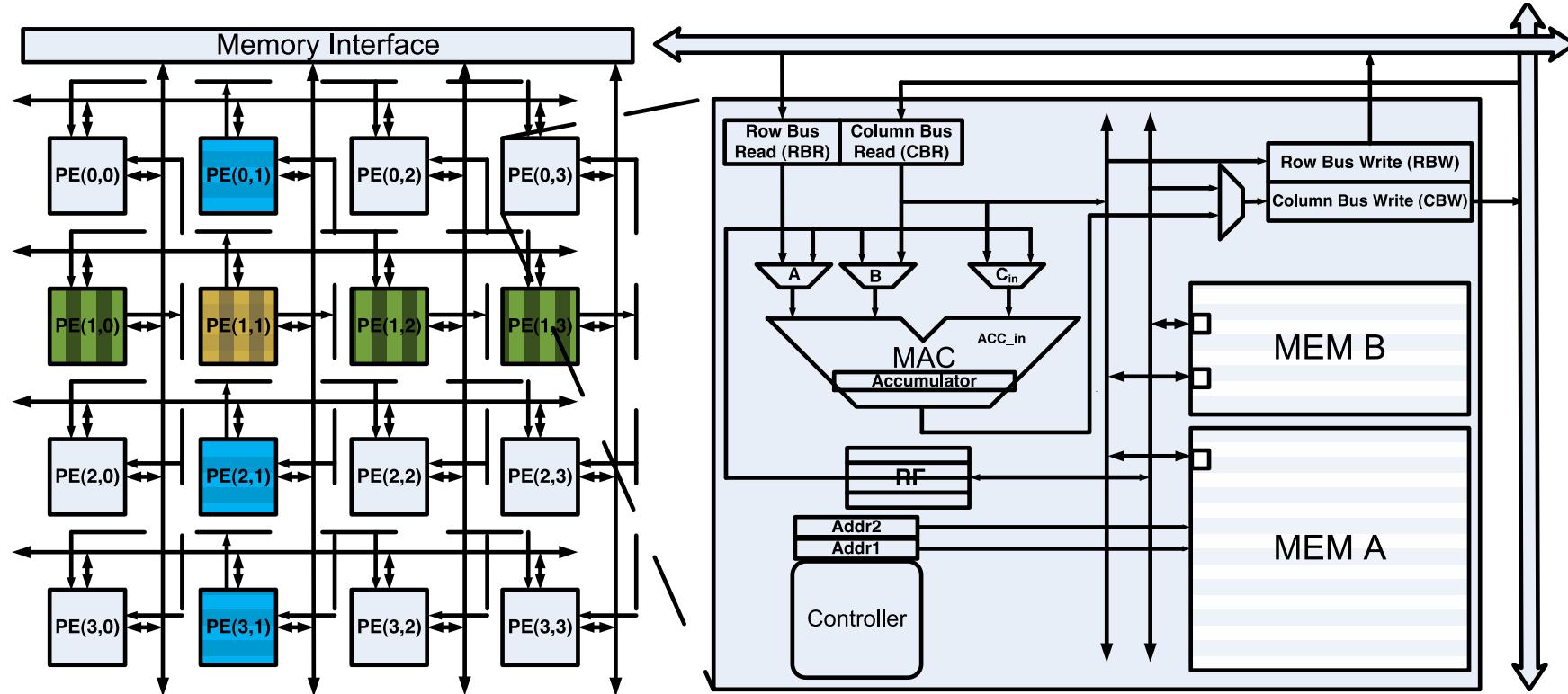
- What do we need ?
- What don't we need?
- Caches?
- Register file?
- ALUs?
- OoO execution?
- Branch predictor?
- Fetch/Decode Unit



The “Rank-1” Machine: Linear Algebra Core

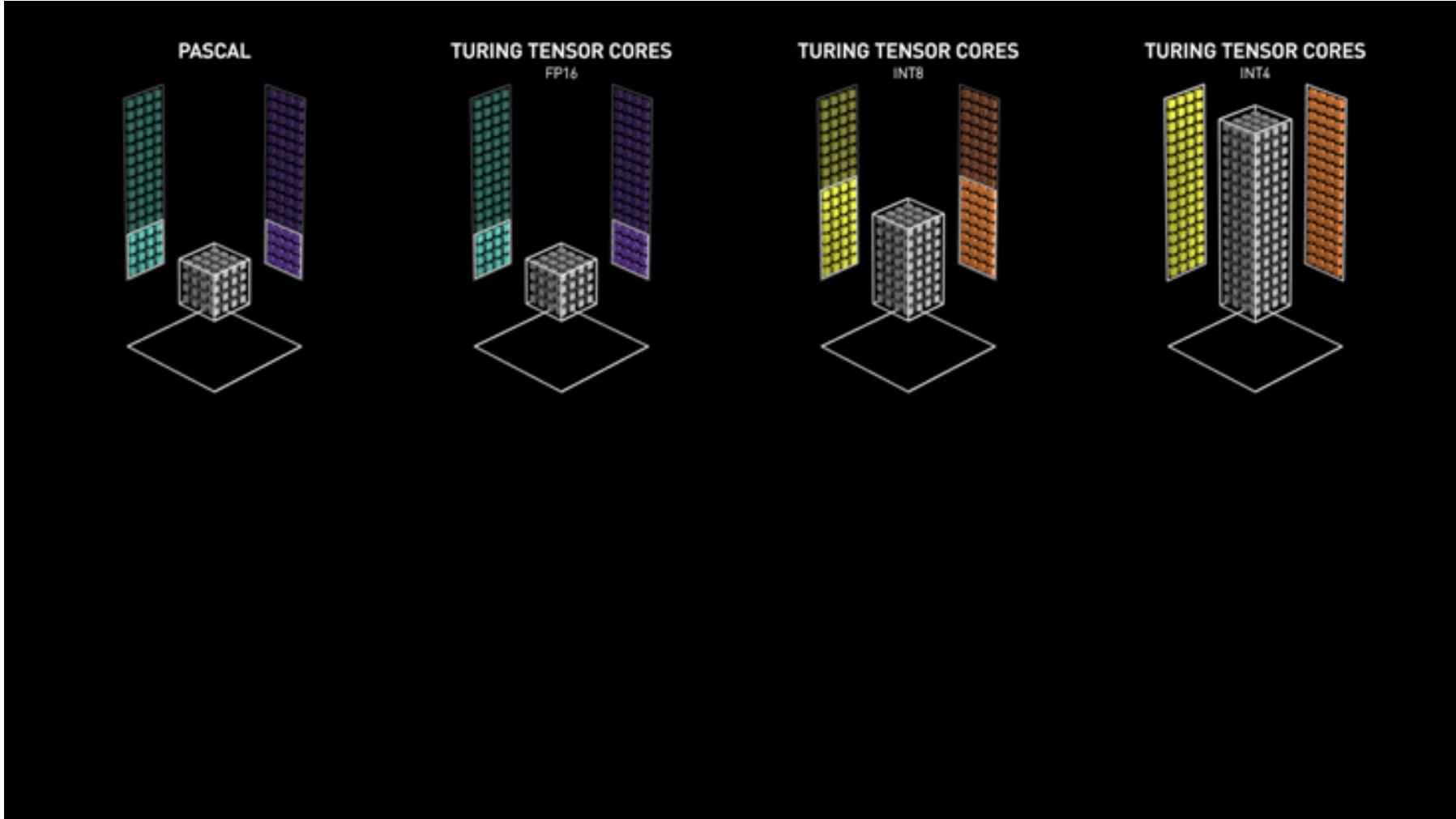


RANK-1 Hardware

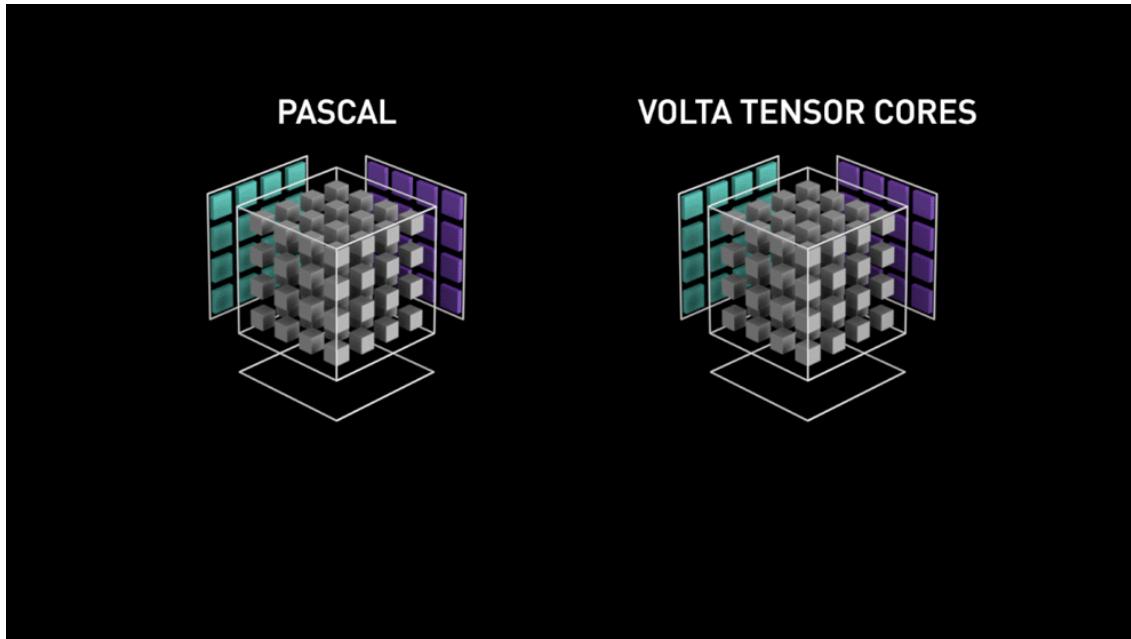


- Scalable 2-D array of $n_r \times n_r$ processing elements (PEs)
 - Specialized floating-point units with 1 MAC/cycle throughput
 - Broadcast busses (no need to pipeline up to $n_r=16$)
 - Distributed memory architecture
 - Distributed, PE-local control

NVIDIA Tensor Cores



NVIDIA Tensor Cores



$$D = \left(\begin{array}{cccc} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{array} \right) \left(\begin{array}{cccc} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{array} \right) + \left(\begin{array}{cccc} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{array} \right)$$

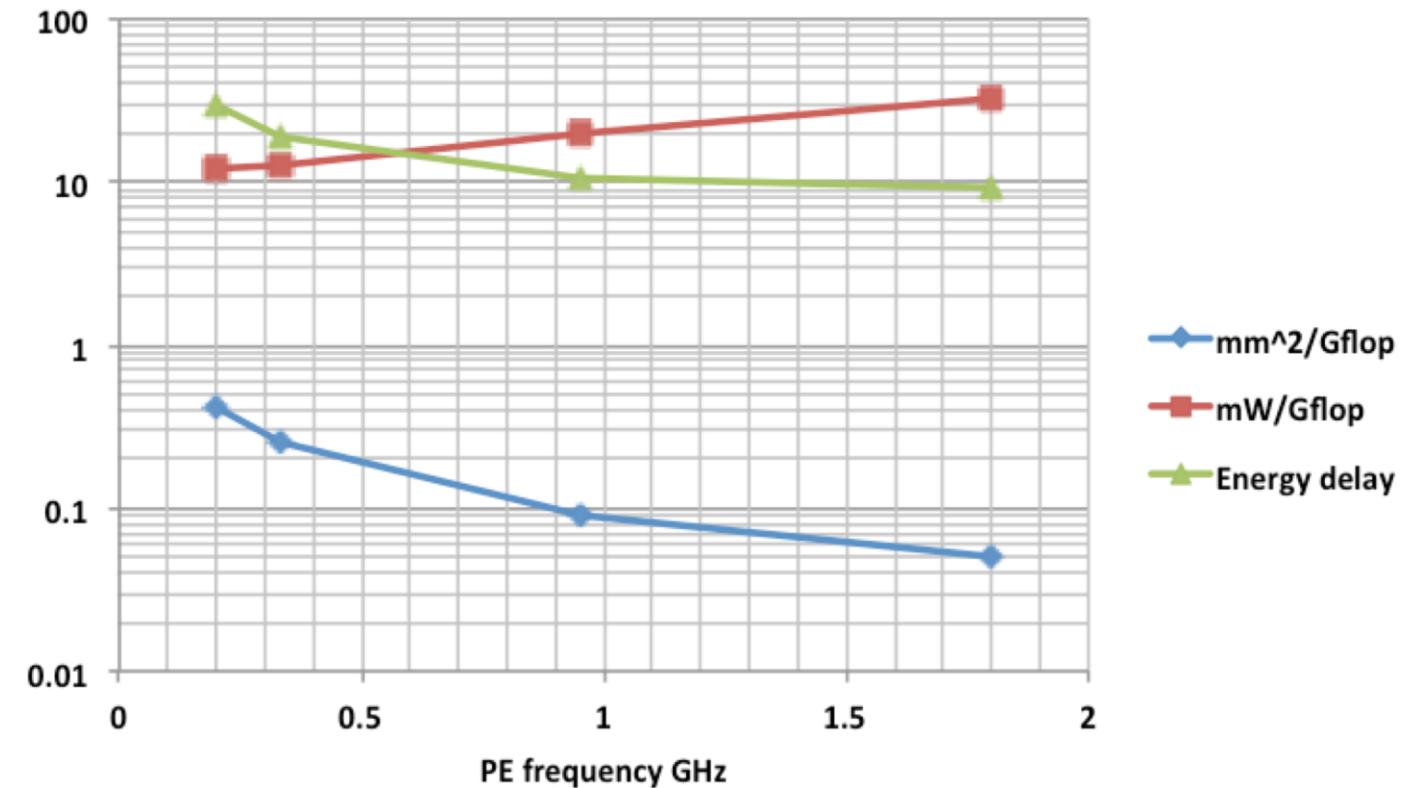
FP16 or FP32 FP16 FP16 FP16 or FP32

Math vs. Memory

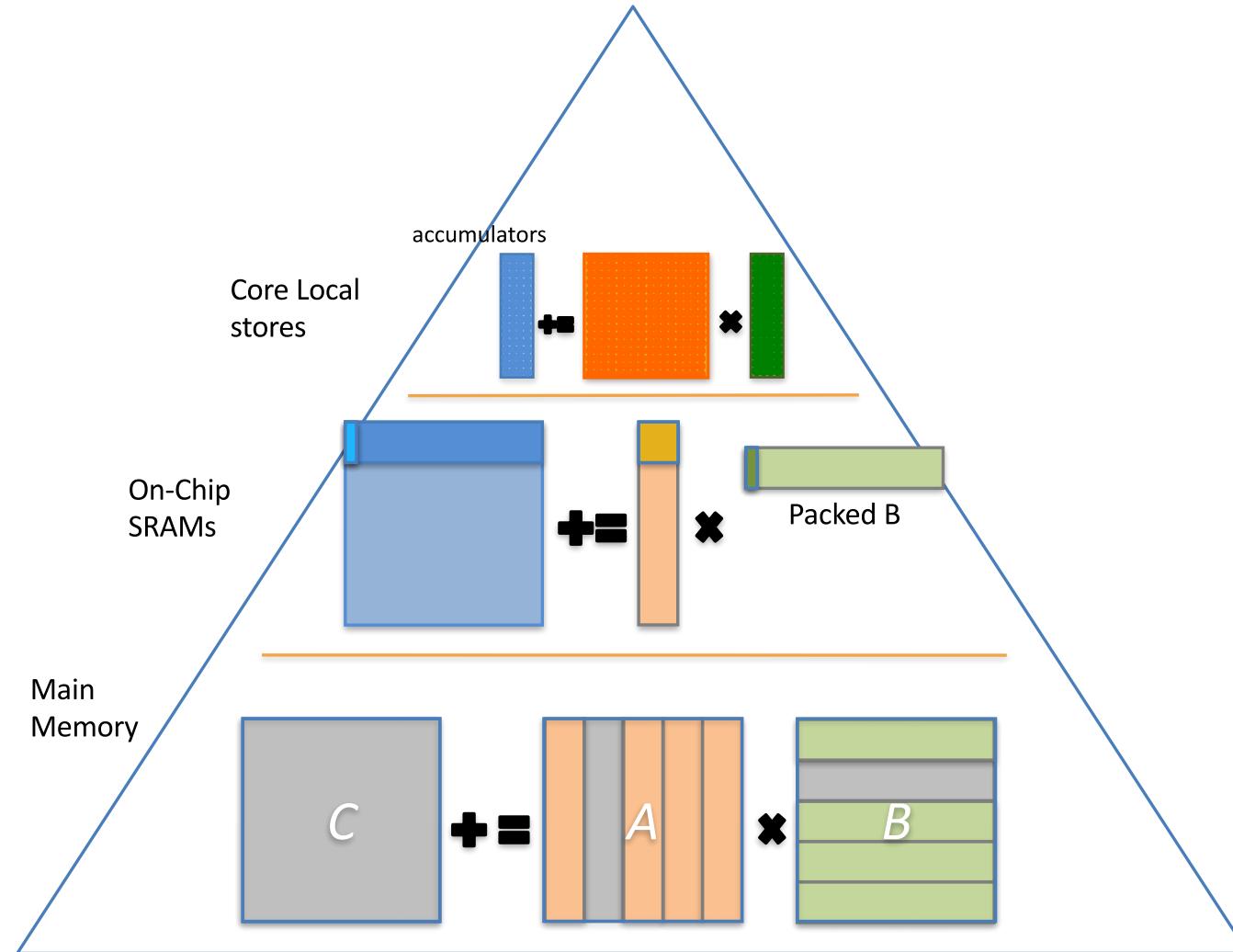
Operation	16 bit (integer)		64 bit (DP-FP)	
	E/op PJ	vs. add	E/op PJ	vs. Add
Add	0.18	1.0x	5	1.0x
Multiply	0.62	3.4x	20	4.0x
16-word register file	0.12	0.7x	0.34	0.07x
64-word register file	0.23	1.3x	0.42	0.08x
4k-word SRAM	8	44x	26	5.2x
32k-word SRAM	11	61x	47	9.4x
DRAM	640	3556x	2560	512x

PE-Level Implementation @45nm

- Area
 - SRAM Dominates
- Power
 - MAC Dominates
- Double precision
 - 60GFlops/W upper limit
- 1 GHz sweet spot of performance vs. efficiency



Build Memory Hierarchy Around It



GEMM Formal Derivation

- Loop representation string
- Rules for each new character
 - Buffers
 - Re-fetch rate

• $m_r n_r k_c m_c n_c m k n$

• $m_0 n_0 k_0 m_1 n_1 m_2 k_1 n_2$

for $j_c = 0, \dots, n-1$ **in** steps of n_c

for $p_c = 0, \dots, k-1$ **in** steps of k_c

$B(p_c : p_c + k_c - 1, j_c : j_c + n_c - 1) \rightarrow B_c$

// Pack into B_c

for $i_c = 0, \dots, m-1$ **in** steps of m_c

$A(i_c : i_c + m_c - 1, p_c : p_c + k_c - 1) \rightarrow A_c$

// Pack into A_c

for $j_r = 0, \dots, n_c - 1$ **in** steps of n_r

// macro-kernel

for $i_r = 0, \dots, m_c - 1$ **in** steps of n_r

for $p_r = 0, \dots, k_c - 1$ **in** steps of 1 // Micro-kernel

$C_c(i_r : i_r + m_r - 1, j_r : j_r + n_r - 1)$

$+ = A_c(i_r : i_r + m_r - 1, p_r)$

$\cdot B_c(p_r, j_r : j_r + n_r - 1)$

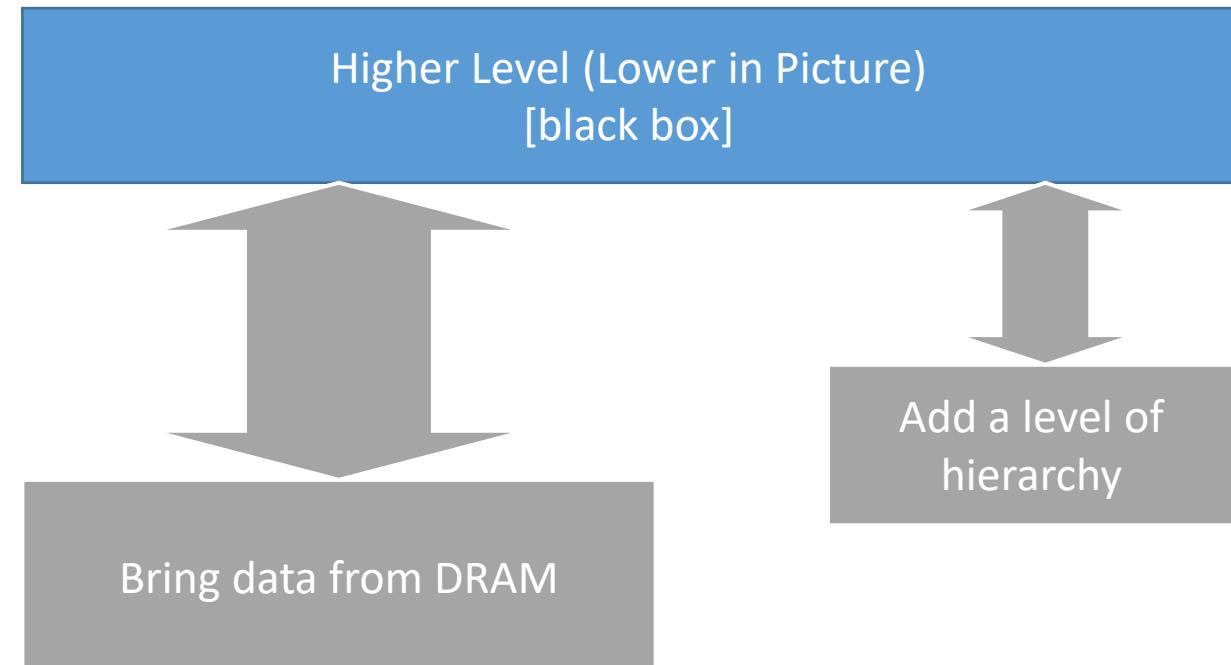
Recursive Formulation

- Fixed order as follows

N_{i-1} M_{i-1} K_{i-1} N_i

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Type	Buffer Size	Buffer Refetch Rate



Recursive Formulation

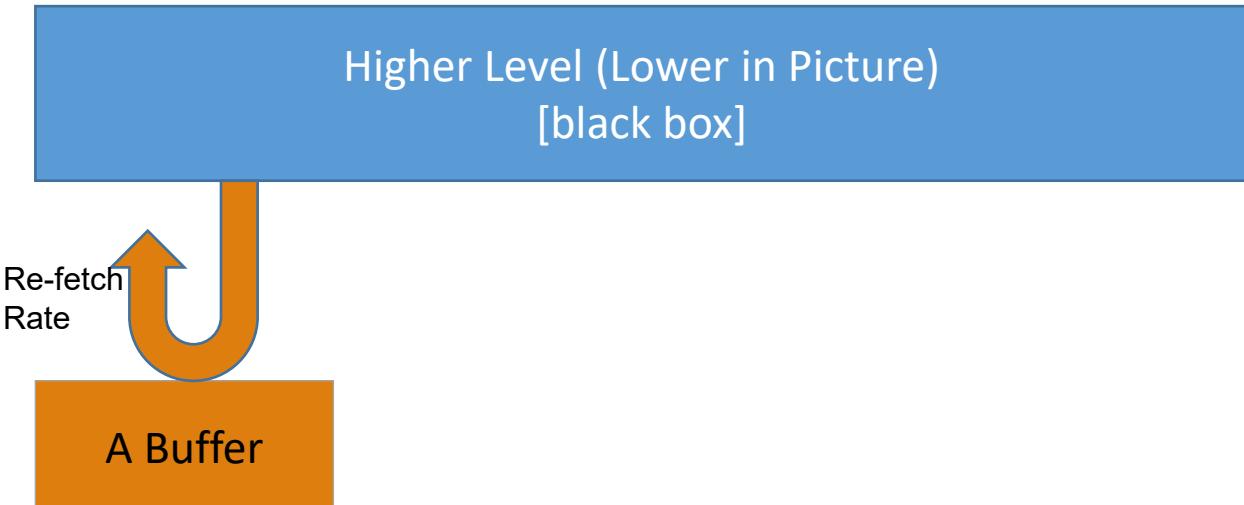
- Fixed order as follows

N_{i-1}M_{i-1}K_{i-1} **N_i**

- From left to right observing
 - N: New A

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Type	Buffer Size	Buffer Refetch Rate
A _i	M _{i-1} K _{i-1}	N _i /N _{i-1}



Recursive Formulation

- Fixed order as follows

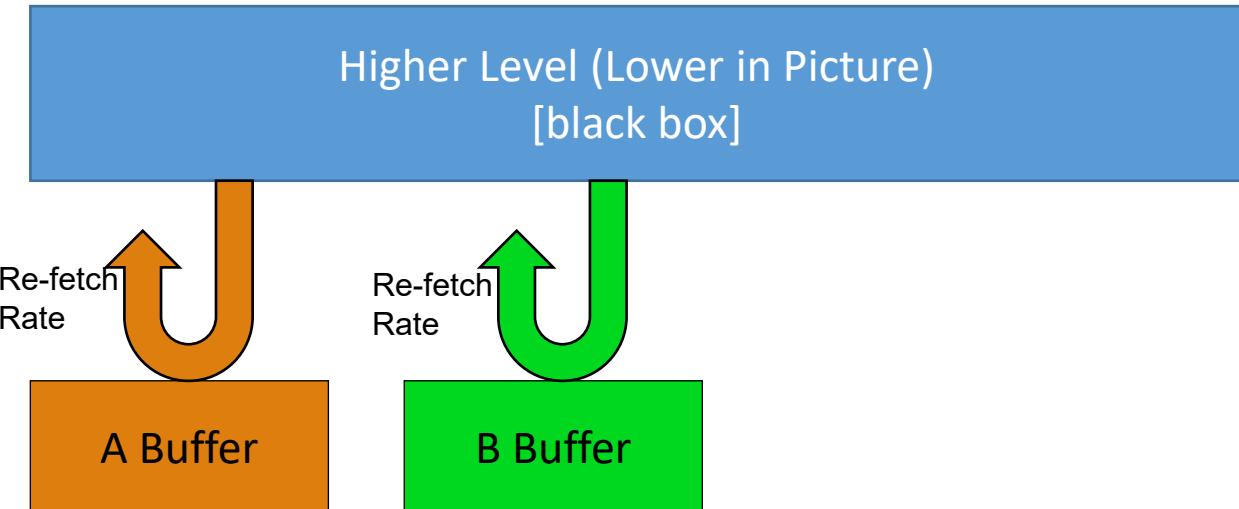
N_{i-1}M_{i-1}K_{i-1} **N_iM_i**

- From left to right observing

- N: New A
- M: New B

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Type	Buffer Size	Buffer Refetch Rate
A _i	M _{i-1} K _{i-1}	N _i /N _{i-1}
B _i	N _{i-1} K _{i-1}	M _i /M _{i-1}



Recursive Formulation

- Fixed order as follows

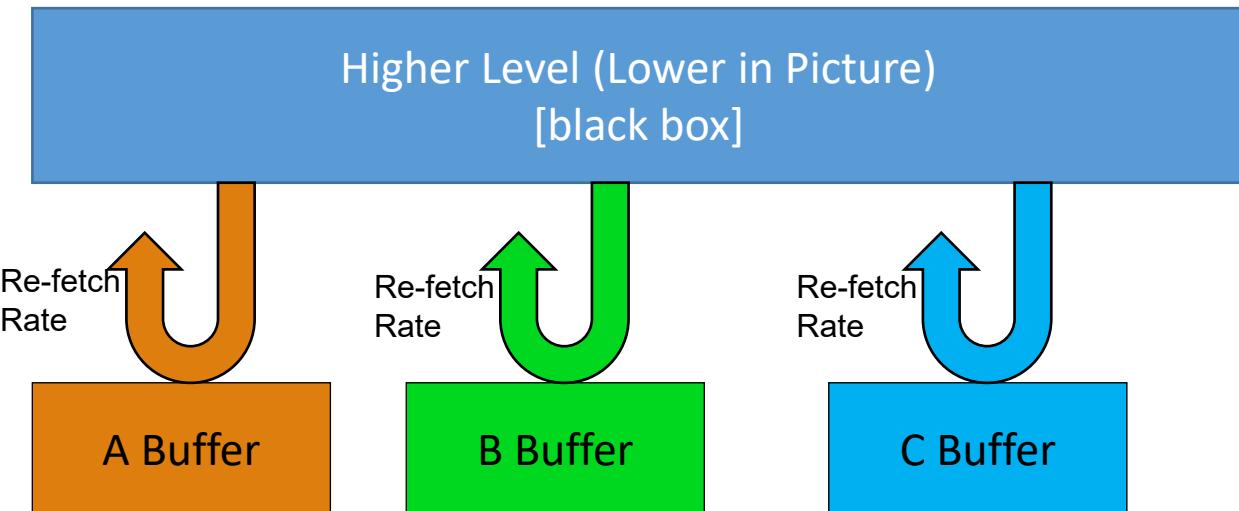
N_{i-1}M_{i-1}K_{i-1} **N_iM_iK_i**...

- From left to right observing

- N: New A
- M: New B
- K: New C

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Type	Buffer Size	Buffer Refetch Rate
A _i	M _{i-1} K _{i-1}	N _i /N _{i-1}
B _i	N _{i-1} K _{i-1}	M _i /M _{i-1}
C _i	M _{i-1} N _{i-1}	2K _i /K _{i-1}

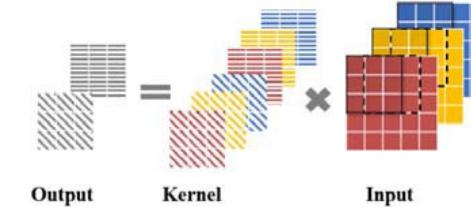
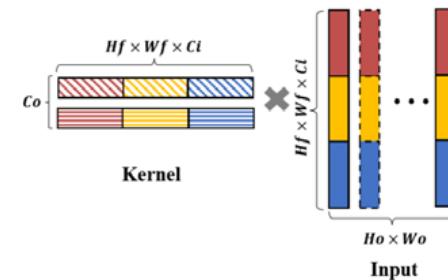


Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
 - Locality and Data Reuse
 - Dataflow Taxonomy
 - Data Orchestration
 - Network-on-Chip
 - Optimization
- Roofline Model
- Energy

Direct Convolution is Better

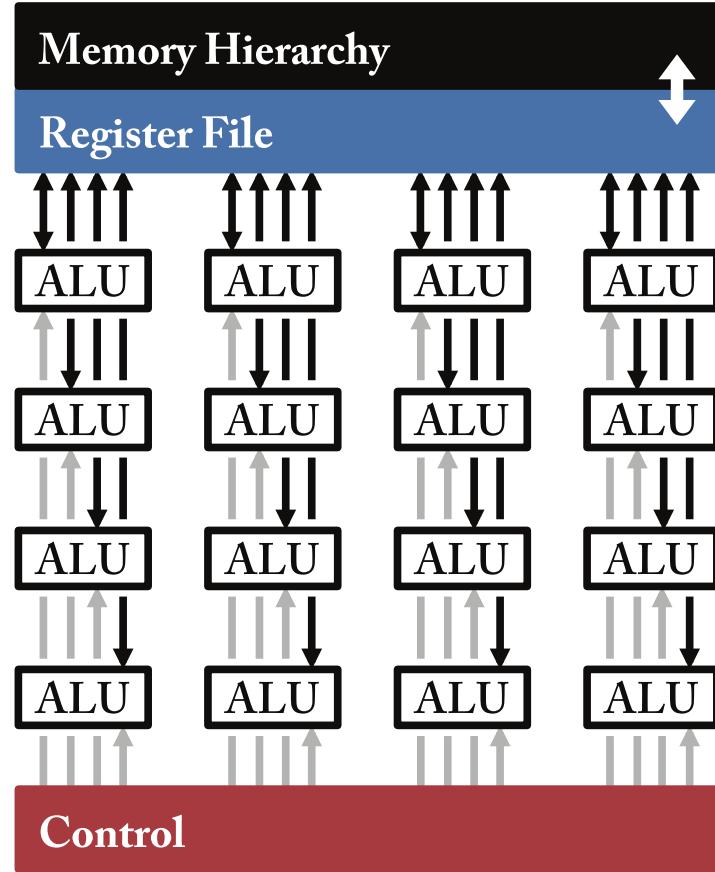
- Higher performance, zero memory overheads



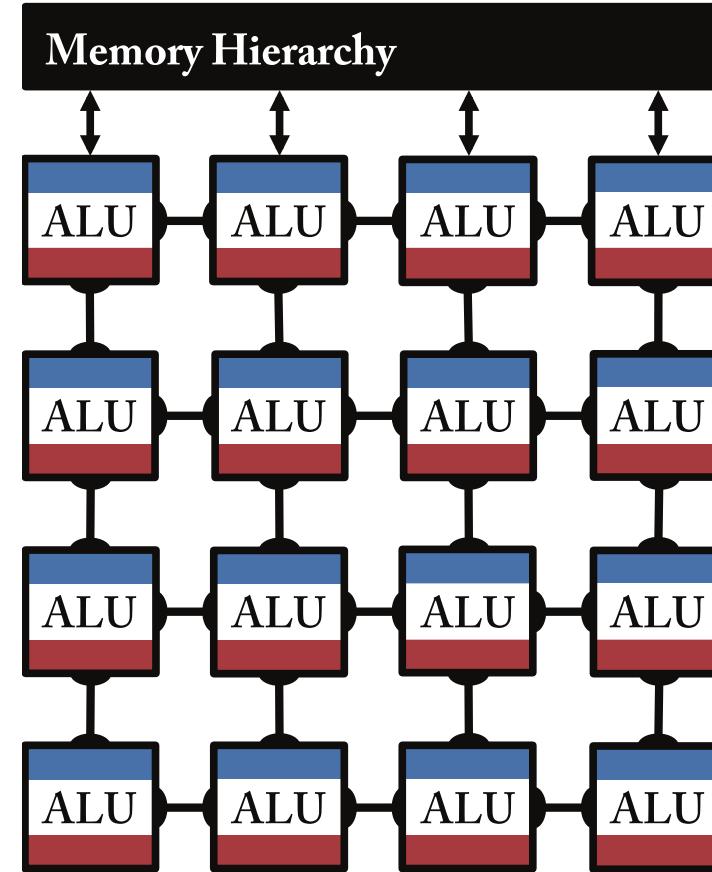
	Matrix-Matrix-Multiplication	Direct Convolution
Packing	Yes <ul style="list-style-type: none"> Over 10x additional memory for image data Performance penalty 	No <ul style="list-style-type: none"> Zero memory overheads No performance penalty
Computation Performance	Less than expected theoretic peak of GEMM	Close to system's theoretic peak

Highly-Parallel Compute Paradigms

Temporal Architecture
(SIMD/SIMT)

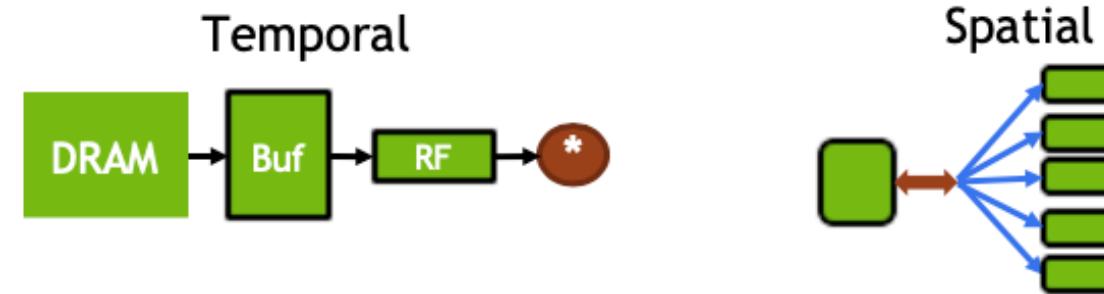


Spatial Architecture
(Dataflow Processing)



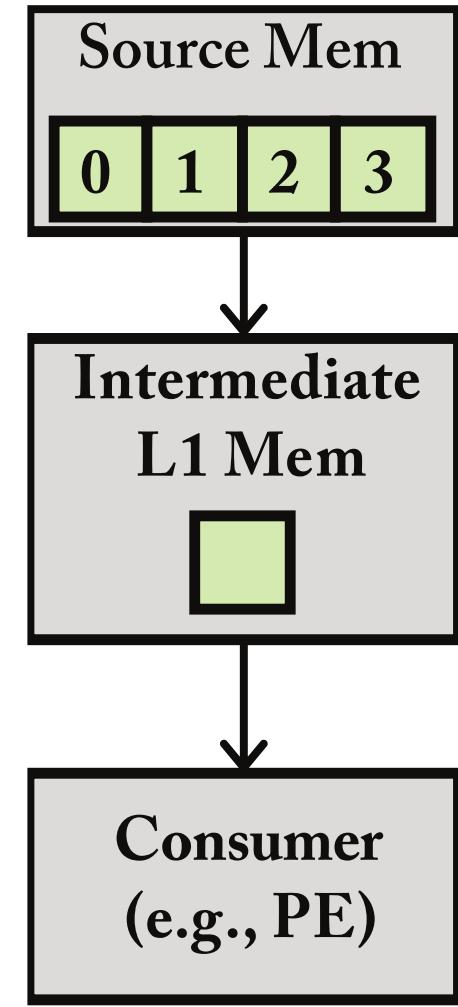
Locality: Temporal and Spatial Reuse

- **Temporal** reuse: the same data is used more than once over time by the same consumer.
- **Spatial** reuse: the same data is used by more than one consumer at different spatial locations of the hardware.



Temporal Reuse

- The same data value is used more than once by the same consumer
 - E.g. PE
- Adding an intermediate memory level to the memory hierarchy
 - Used multiple times at the intermediate level
 - Reduces the overall energy cost



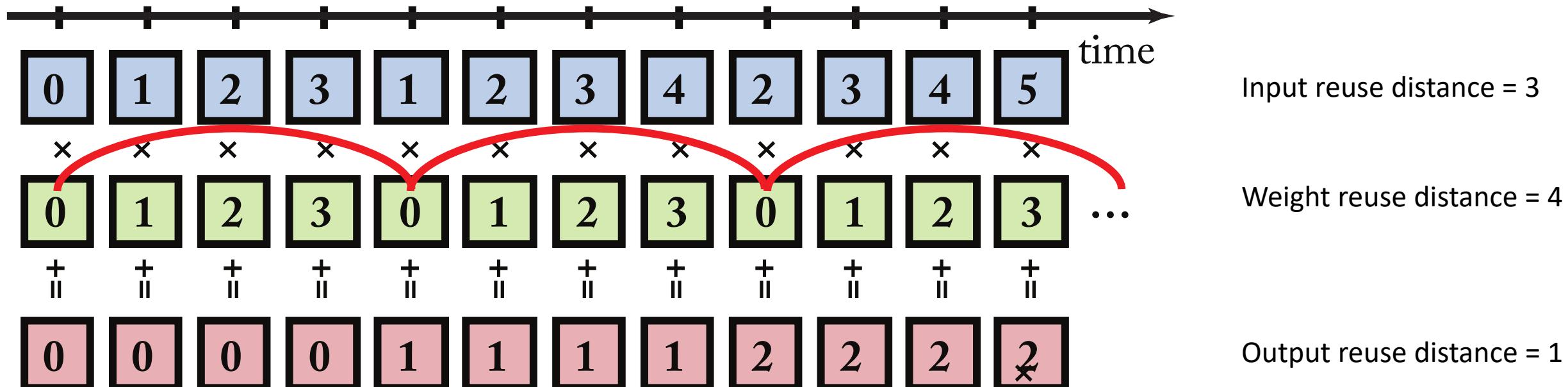
* Only storage for weights is shown

1D CNN Temporal Reuse Example

- Example 1D convolution

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

- Operation ordering #1

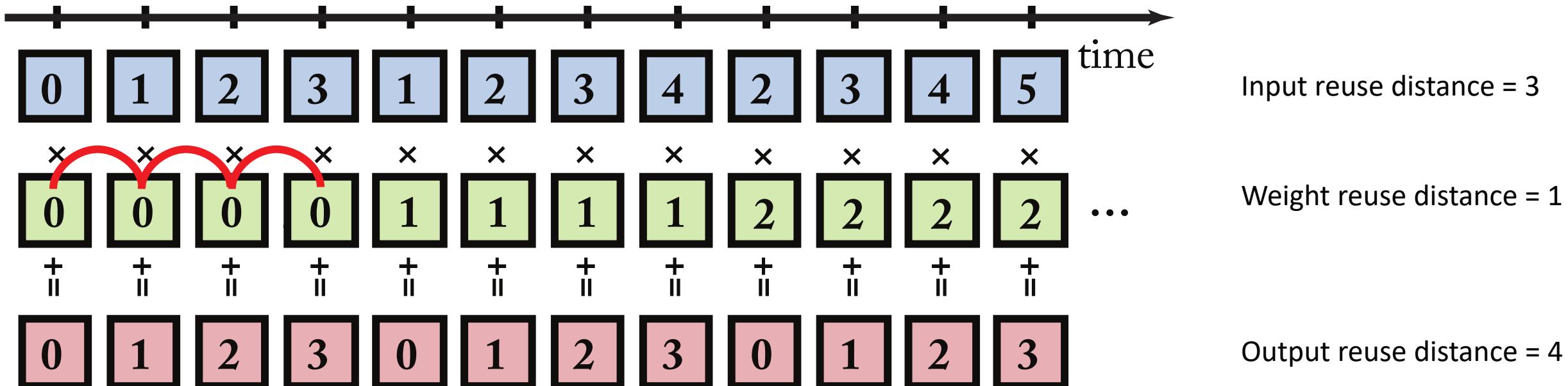


1D CNN Temporal Reuse Example

- Example 1D convolution

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}$$

- Operation ordering #2



Temporal Reuse Distance

- Temporal reuse distance
 - The **maximum number of data accesses** required by the consumer in **between the accesses to the same data value**
 - Storage capacity of the intermediate memory level limits the maximum reuse distance
 - **Storage capacity \geq maximum reuse distance**
 - Temporal reuse can be exploited
 - Not limit to weights
- Reduce the reuse distance of one data type often increase the reuse distance of other data types
- Exploited by multiple levels of the memory hierarchy

Spatial Reuse

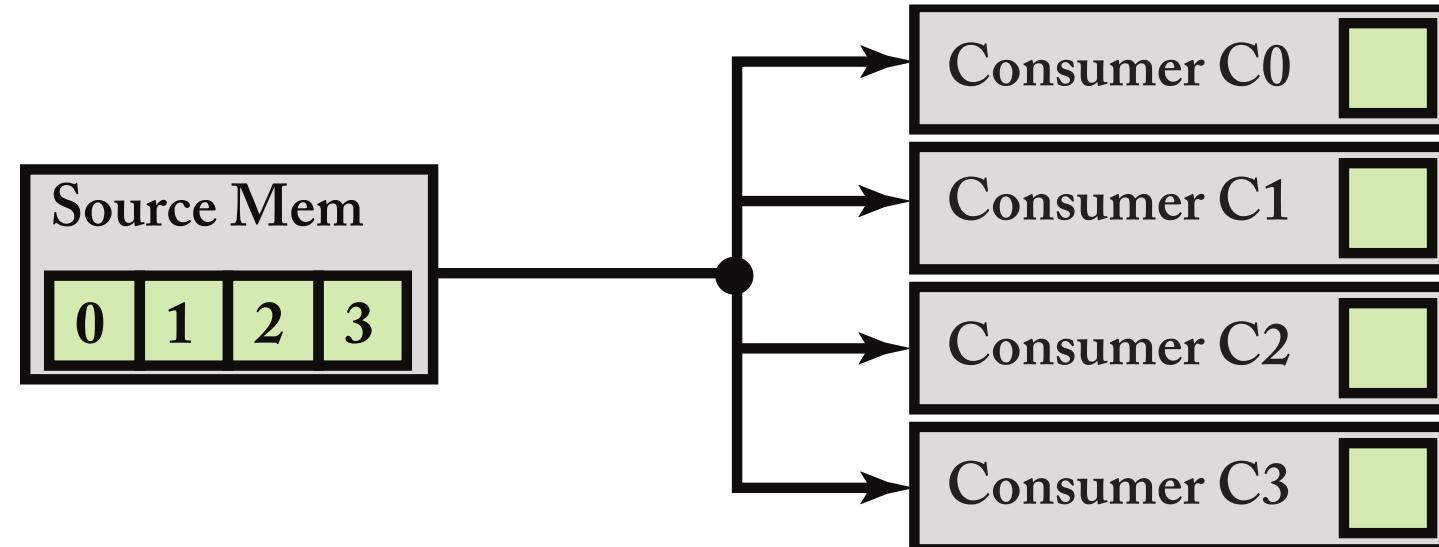
- The **same data value** is used by **more than one consumer** at **different spatial locations** of the hardware
 - E.g., a group of PEs
- Exploited by **reading the data once** from the source memory level and **multicasting it to all** of the consumers
- **Reducing the number of accesses** to the source memory level
 - Reduces the overall energy cost
- **Reducing the bandwidth required** from the source memory level
 - Helps to keep the PEs busy and therefore increases performance

Spatial Reuse Distance

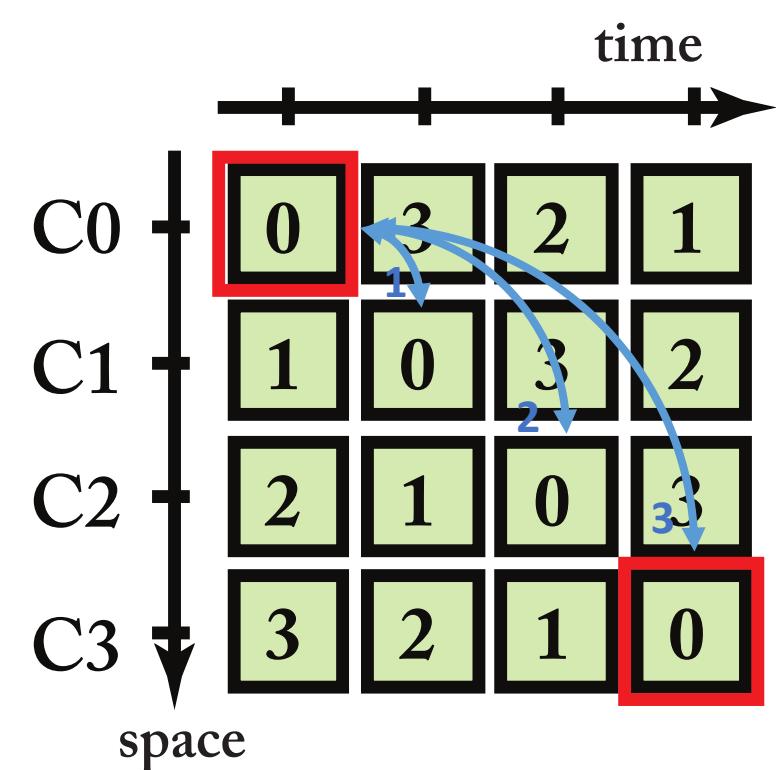
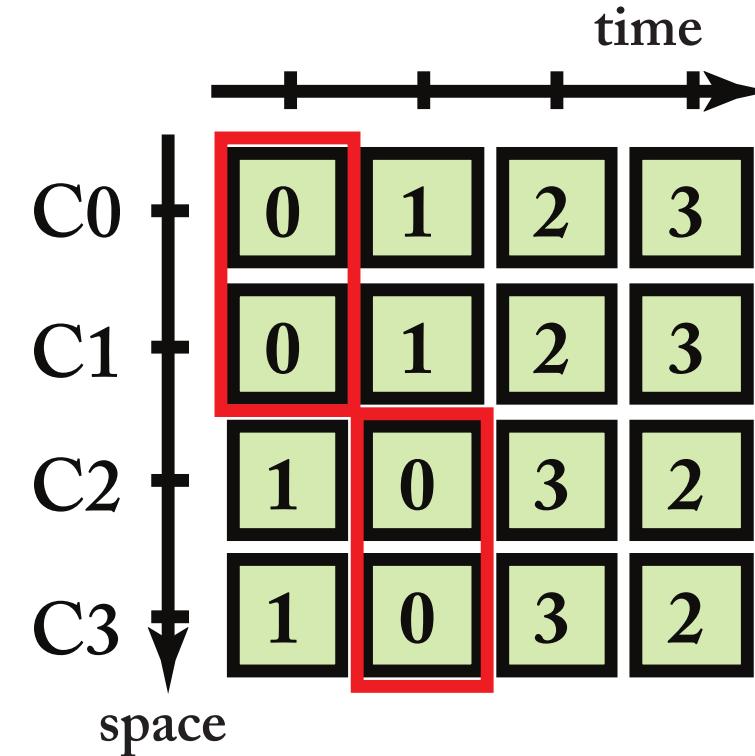
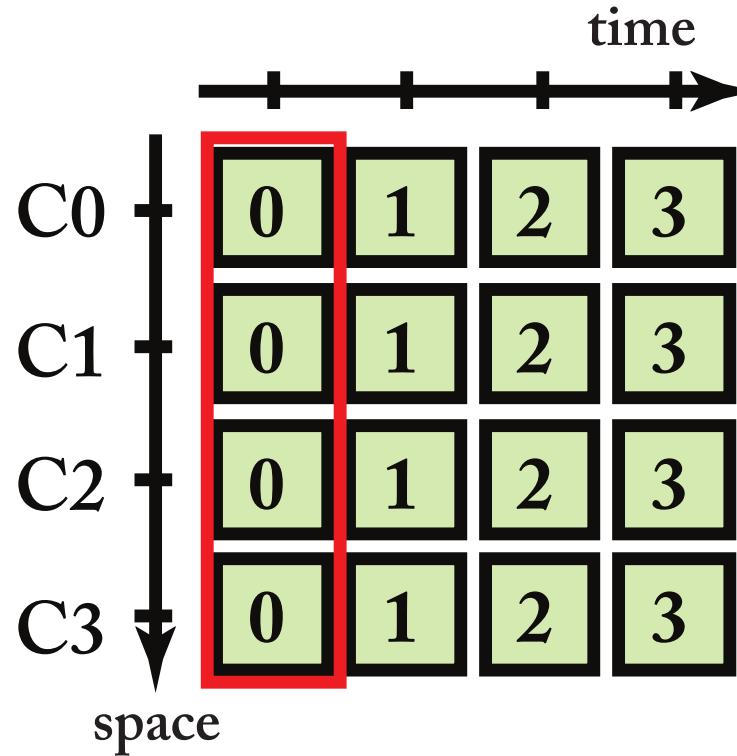
- If no storage capacity in a group of consumers
 - Spatial reuse can only be exploited by the subset of consumers that can process the data in the same cycle as the multicast of the data
- Spatial Reuse Distance
 - The **maximum number of data accesses** in between **any pair of consumers** that **access the same data value**

Memory Hierarchy for Spatial Reuse

* Only storage for weights is shown

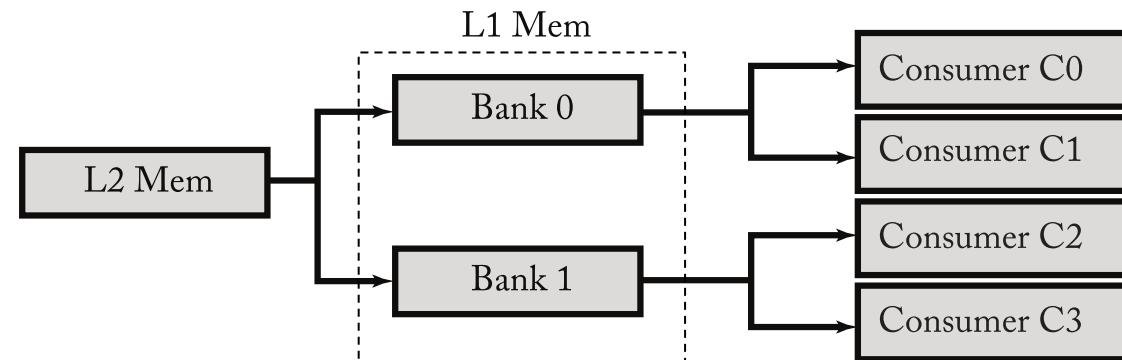


Spatial Reuse Example



Routing Data to Multiple Destination

- **Network-on-Chip(NoC)**
 - Support the data distribution patterns as derived from the ordering of operations
- Trade-off
 - Exploiting **spatial reuse at higher levels** of the memory hierarchy creates **more duplicated data** in the hardware
 - Supporting **multicast to all** consumers in a single level at a large scale can also be **expensive**



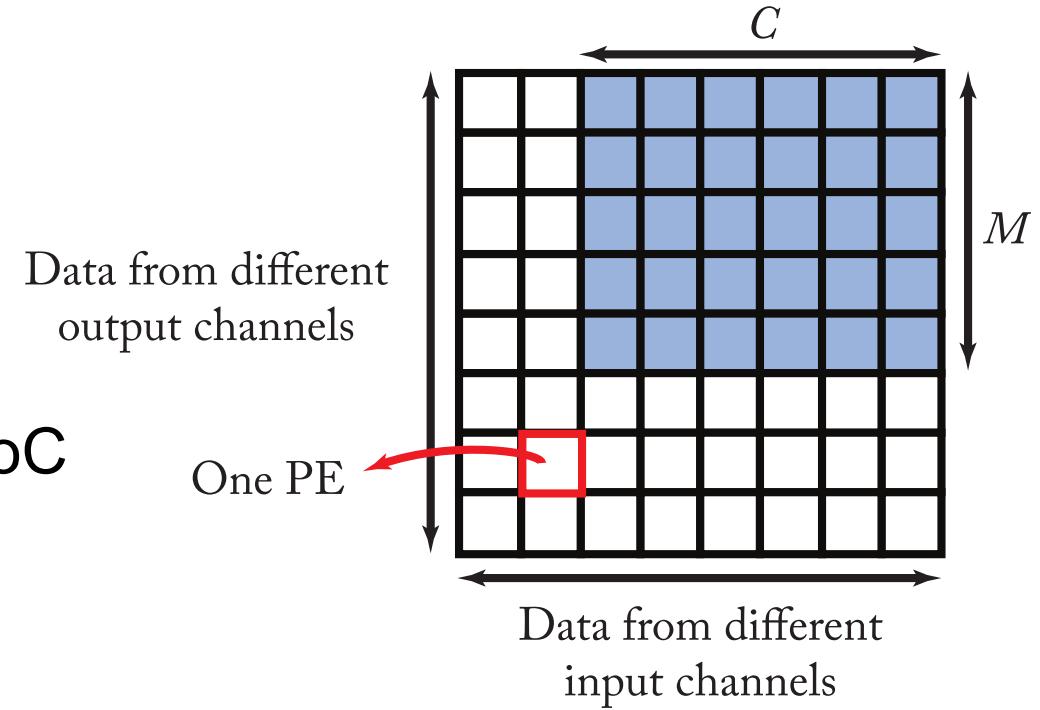
physical connectivity of the NoC between the L1 memory and the consumers limits the multicast from any banks in L1 to all consumers

Reducing Reuse Distance

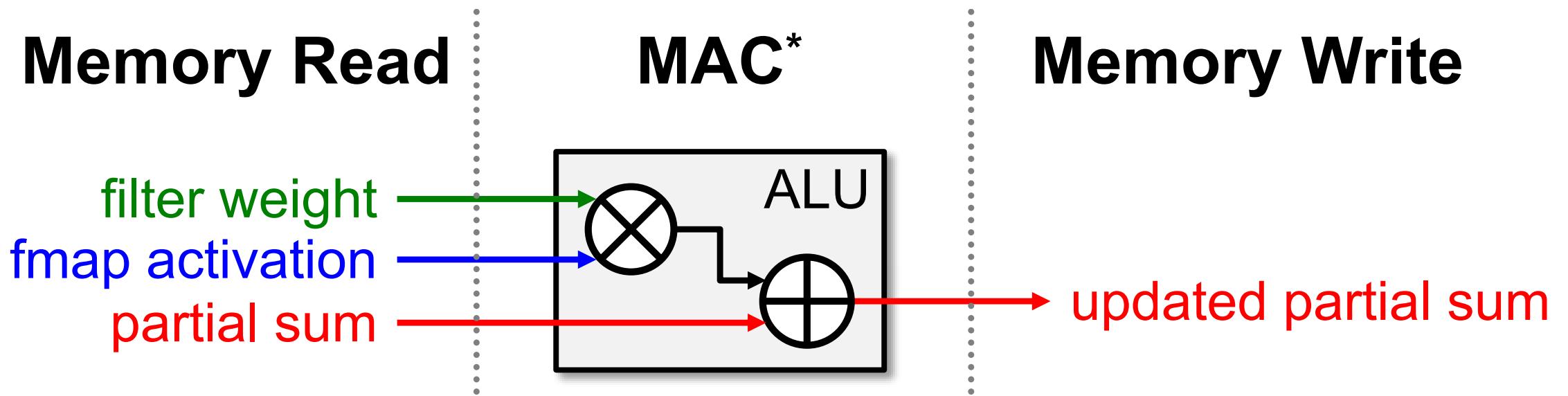
- Data stay stationary over time in the sequence of operations
 - Reduce temporal reuse distance
- Data tiling (or blocking)
 - Temporal tiling
 - Reduce temporal reuse distance
 - Reducing the reuse distance of specific data types
 - Make it smaller than the storage capacity of a certain memory level in the memory hierarchy
 - Spatial tiling
 - Reduce spatial reuse distance
 - Reusing the same data value by as many consumers as possible
 - Reducing the reuse distance so that one multicast can serve as many consumers as possible given a fixed amount of storage capacity at each consumer

Under-utilization of Spatial tiling

- Under-utilization when
 - $M <$ Number of PEs in a column or $N <$ Number of PEs in a row
 - M: number of output channels
 - N: number of input channels
- Solution
 - Multiple tiles of data run at the same time
 - Sufficient flexibility of data delivery and the NoC
 - Provide sufficient bandwidth

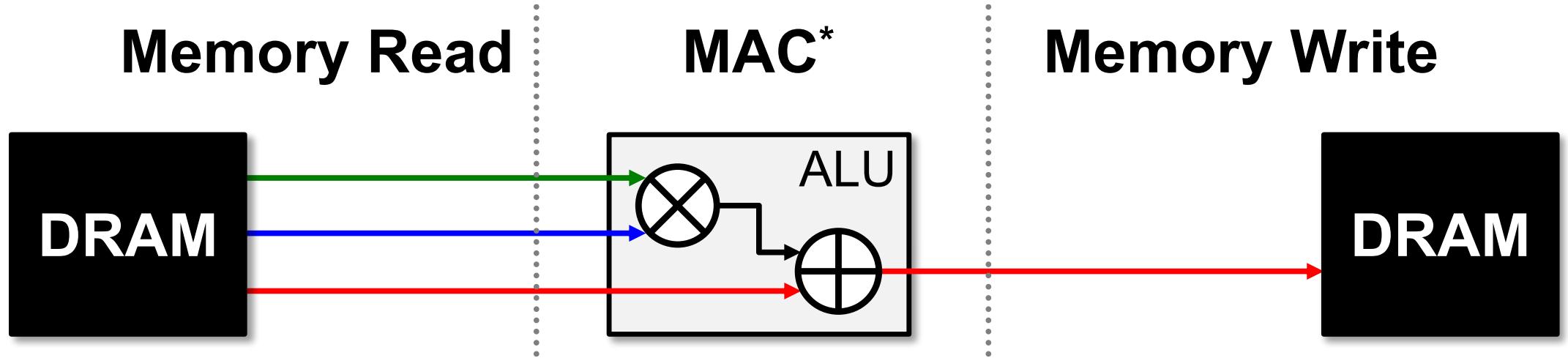


PE with Memory Access



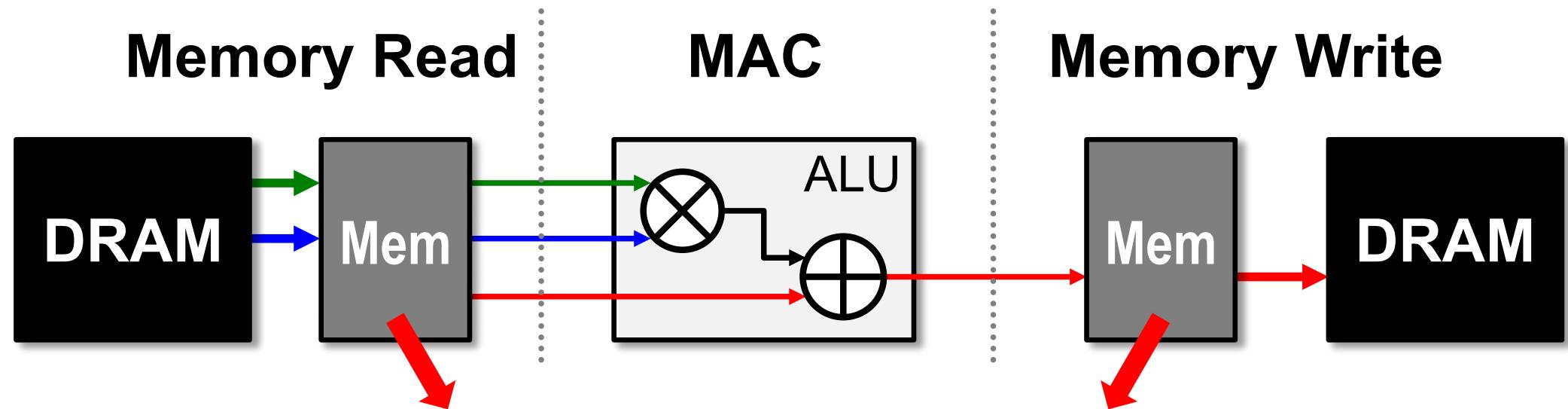
DRAM Access

- **Worse Case:** all memory R/W are **DRAM accesses**



Leverage Local Memory for Data Reuse

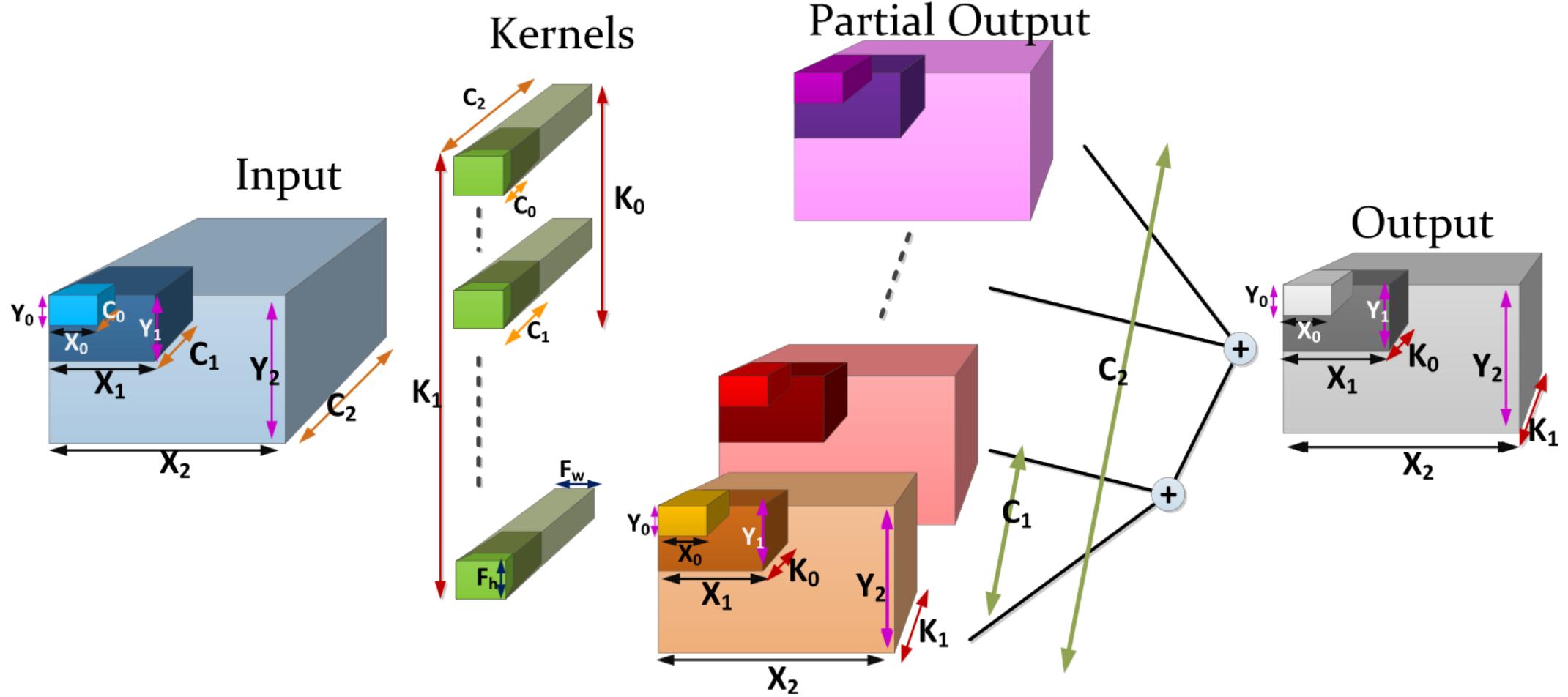
- Temporal reuse: the same data is used more than once over time by the same consumer



Extra levels of local memory hierarchy

Smaller, but Faster and more Energy-Efficient

Hierarchical Blocking of a Single Convolutional Layer



Formal Derivation of CONV

- Loop representation string
- Rules for each new character
 - Buffer
 - Re-fetch rate
- $Y_0K_0C_0X_0K_1C_1Y_1X_1$

→

```

for  $x_i = 1 : X_0 : X_1$  do
  for  $y_i = 1 : Y_0 : Y_1$  do
    for  $c_i = 1 : C_0 : C_1$  do
      for  $k_i = 1 : K_0 : K_1$  do
        for  $x_{ii} = x_i : 1 : x_i + X_0$  do
          for  $c_{ii} = c_i : 1 : c_i + C_0$  do
            for  $k_{ii} = k_i : 1 : k_i + K_0$  do
              for  $y_{ii} = y_i : 1 : y_i + Y_0$  do
                compute a convolution window;
                out[y_{ii}] [x_{ii}] [k_{ii}] += img[y_{ii} : y_{ii} + F_h] [x_{ii} : x_{ii} + F_w] [c_{ii}] × kernel[c_{ii}] [k_{ii}];
  
```

Recursive Formulation

- Fixed order as follows
 $Y_{i-1} X_{i-1} C_{i-1} K_{i-1} \dots$
→
- From left to right observing

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Name	Buffer Size	Buffer Refetch Rate

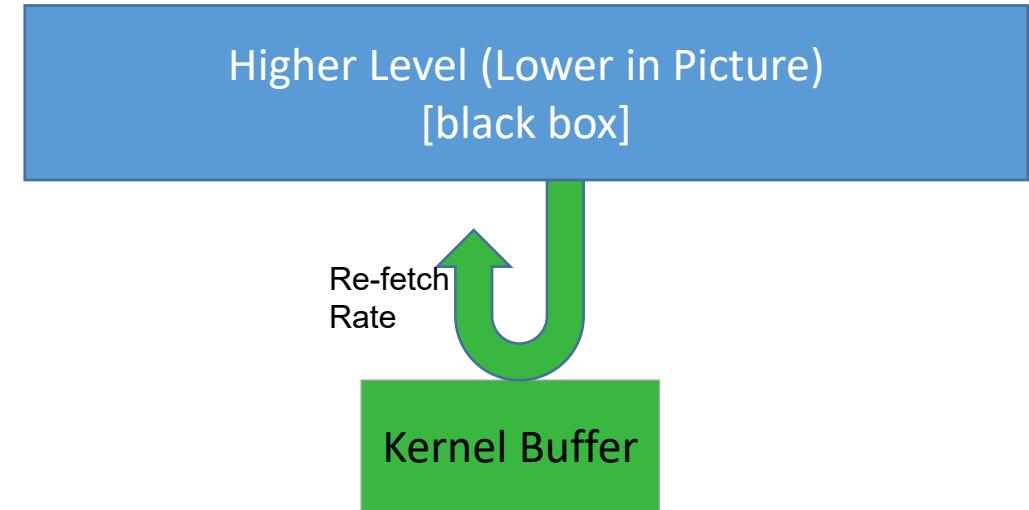
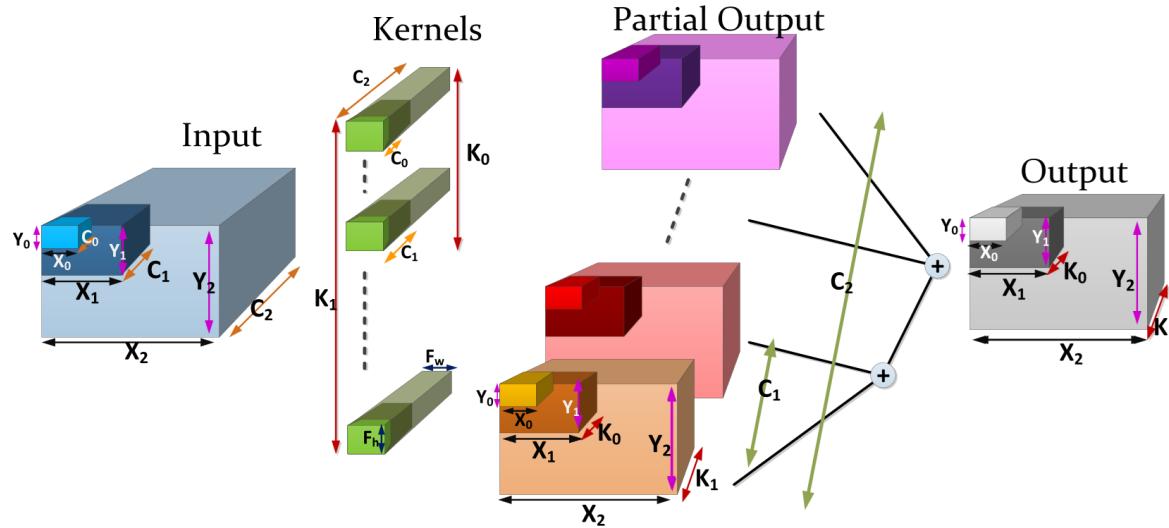
Higher Level (Lower in Picture)
[black box]

Recursive Formulation

- Fixed order as follows
 $\underline{\underline{Y_{i-1}X_{i-1}C_{i-1}K_{i-1}}} \underline{\underline{Y_iX_i}} \dots$
- From left to right observing
 - X or Y: new KB

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Name	Buffer Size	Buffer Refetch Rate
KB_i	$C_{i-1}K_{i-1}F_h F_w$	$(X_i Y_i) / (X_{i-1} Y_{i-1})$

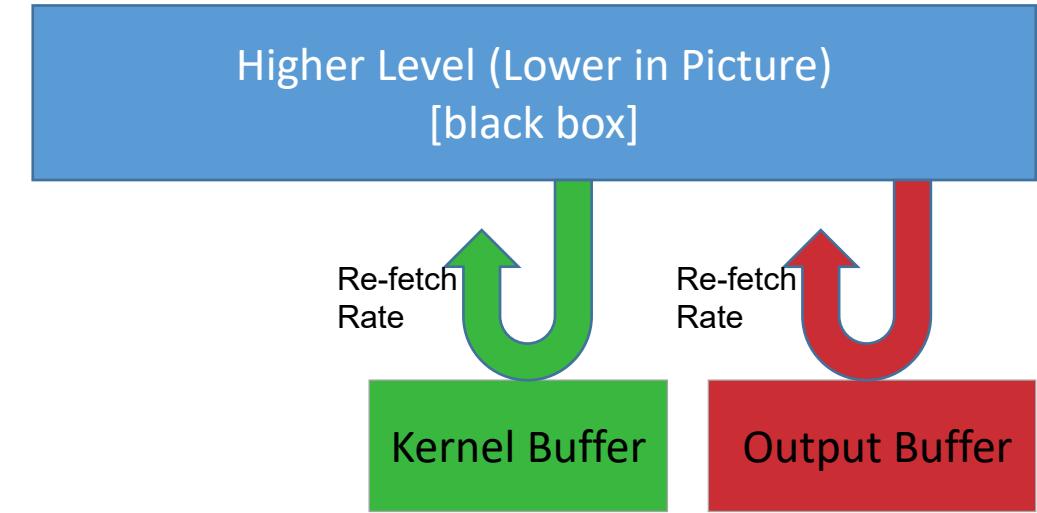
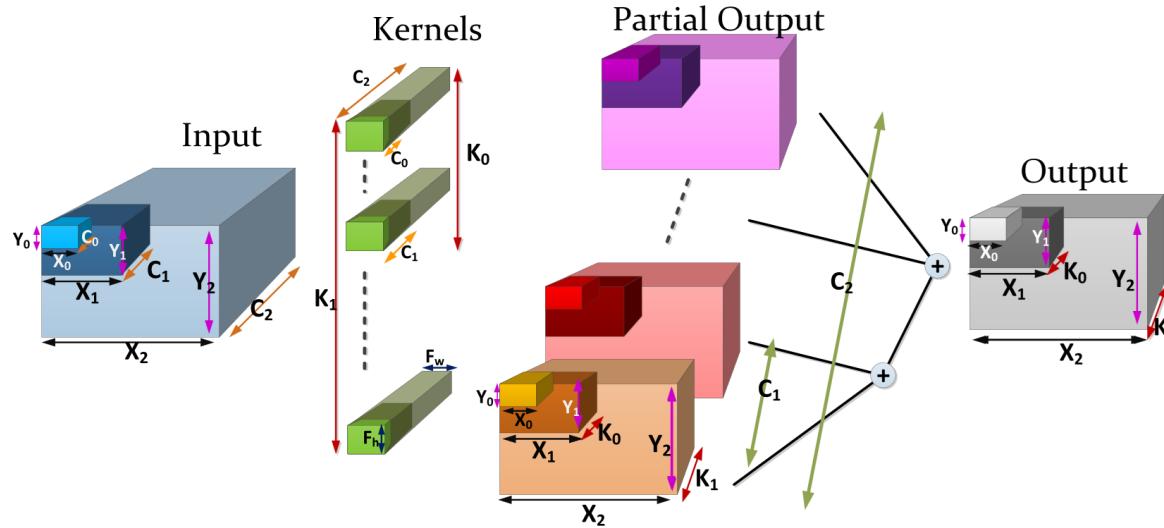


Recursive Formulation

- Fixed order as follows
 $\underline{Y_{i-1}X_{i-1}C_{i-1}K_{i-1}} \color{green}{Y_iX_i} \color{red}{C_i} \dots$
- From left to right observing
 - X or Y: new KB
 - C: new OB

Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Name	Buffer Size	Buffer Refetch Rate
KB_i	$C_{i-1}K_{i-1}F_h F_w$	$(X_i Y_i) / (X_{i-1} Y_{i-1})$
OB_i	$Y_i X_i K_{i-1}$	$2C_i / C_{i-1}$

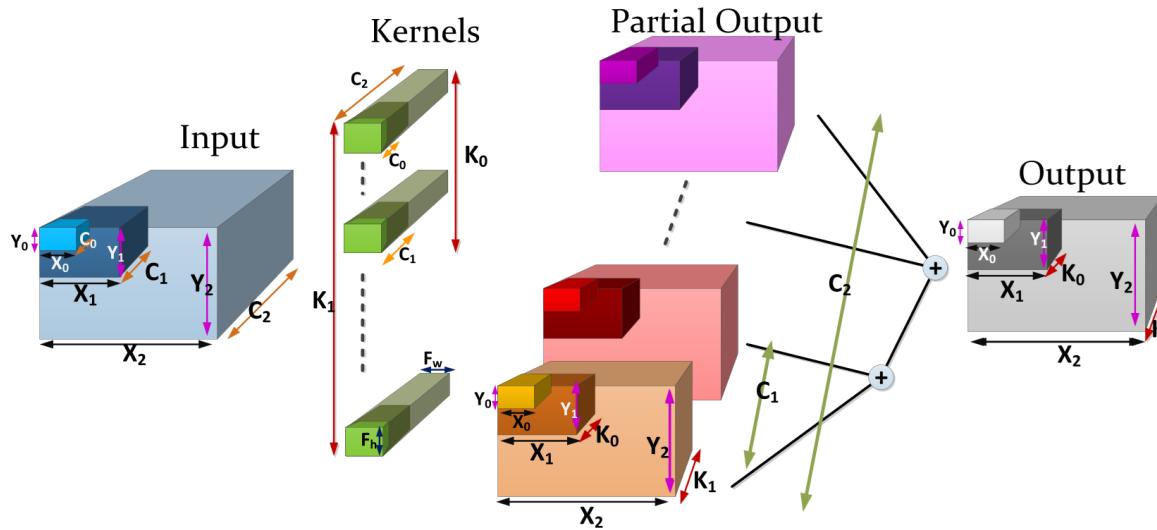


Recursive Formulation

- Fixed order as follows

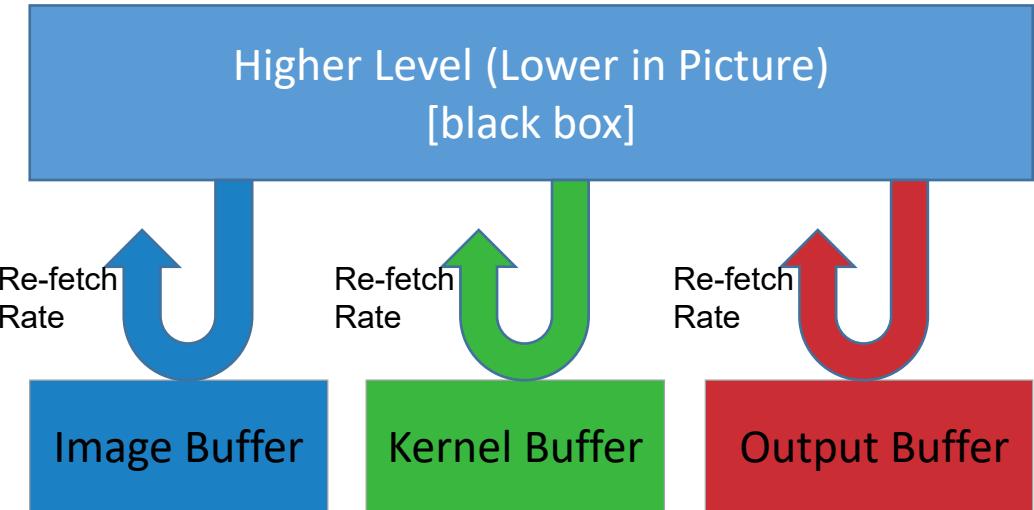
Y_{i-1}X_{i-1}C_{i-1}K_{i-1} Y_iX_iC_iK_i ...

- From left to right observing
 - X or Y: new KB
 - C: new OB
 - K: new IB



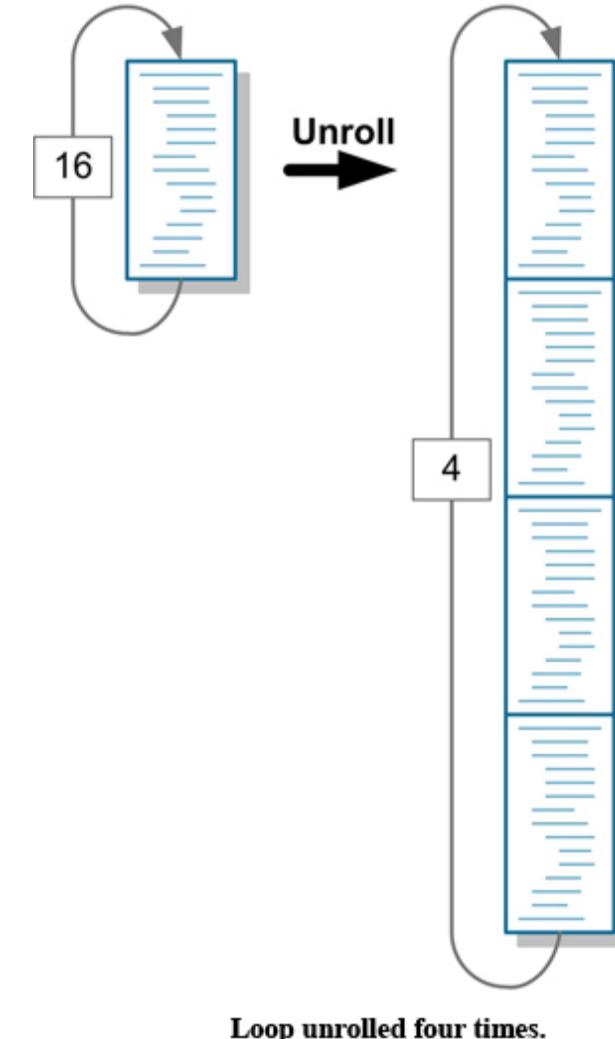
Buffer Refetch Rate: the number of times a piece of data is fetched from a certain buffer after initially being loaded into that buffer

Buffer Name	Buffer Size	Buffer Refetch Rate
KB _i	$C_{i-1}K_{i-1}F_h F_w$	$(X_i Y_i)/(X_{i-1} Y_{i-1})$
OB _i	$Y_i X_i K_{i-1}$	$2C_i/C_{i-1}$
IB _i	$(Y_i + F_h - 1)(X_i + F_w - 1)C_i$	$(K_i(Y_i + F_h - 1)(X_i + F_w - 1))/(K_{i-1}Y_i X_i)$



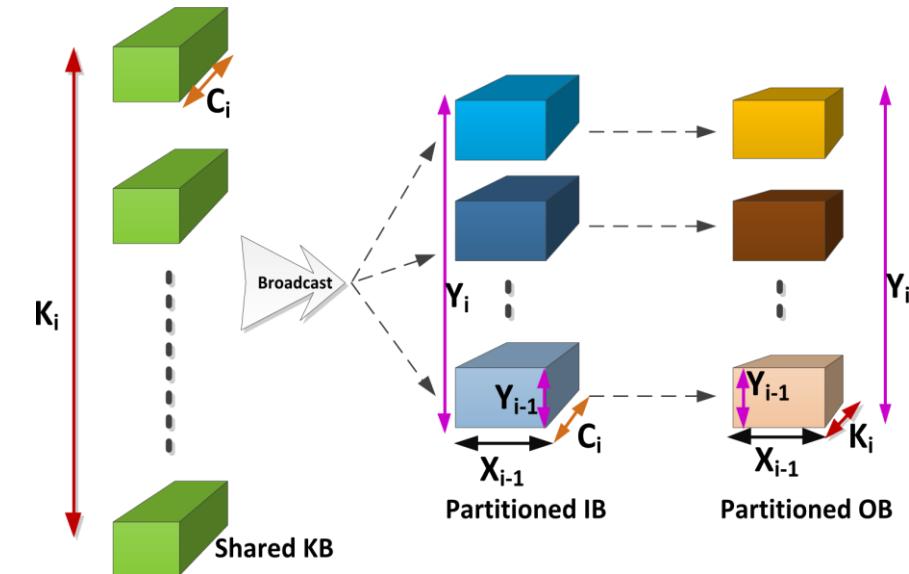
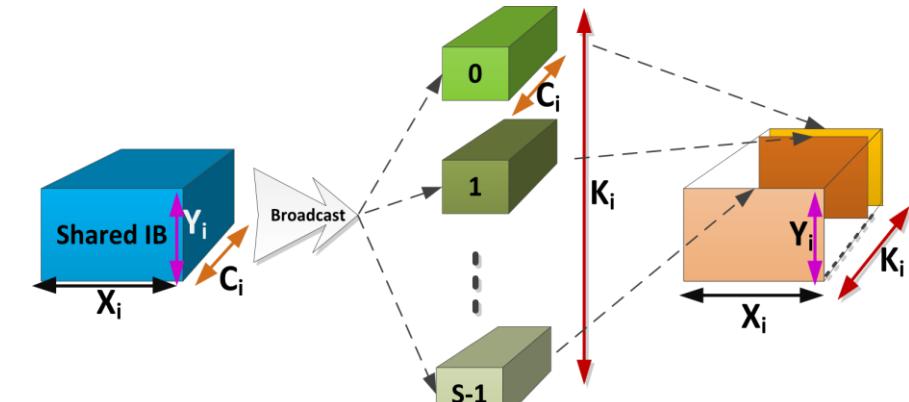
How Does Parallelism Work?

- **Unroll Loops in Space**
- Replicate Compute Units
- Perform Loops in parallel
 - SIMD
 - Multiple Cores
 - Multiple CPU/GPUs



Multi-Core Parallelism and Partitioning

- Across **channels (C)**
 - Reduction
 - Synchronization
- Across **kernels (K)**
 - Synchronization
- Across **image (X or Y)**
 - Independent
- Broadcast operation
- Partitioned memory
- Nested parallelism

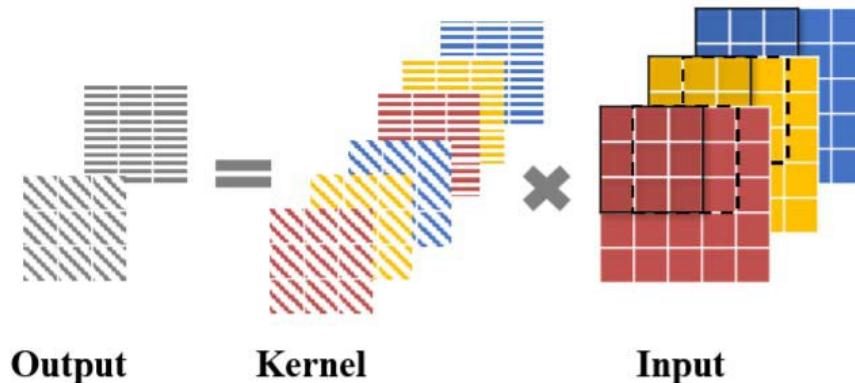


High Performance Direct Convolution



- Map application to system architecture to fully exploit system capacity

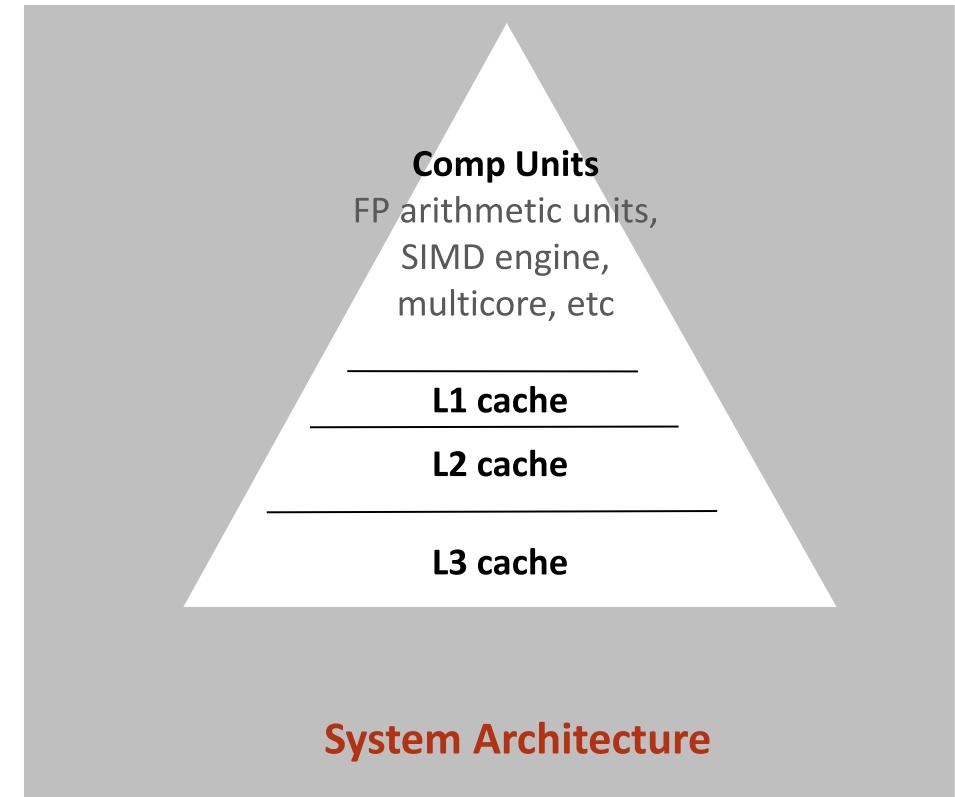
High-dimensional data tensor



Multi-level nested loops and various data access patterns

```
for i = 1 to  $C_i$  do
    for j = 1 to  $C_o$  do
        for k=1 to  $W_o$  do
            for l = 1 to  $H_o$  do
                for m = 1 to  $W_f$  do
                    for n = 1 to  $H_f$  do
                         $O_{j,k,l} += I_{i,k \times x+m, l \times s+n} \times F_{i,j,m,n}$ 
```

Direct Convolution

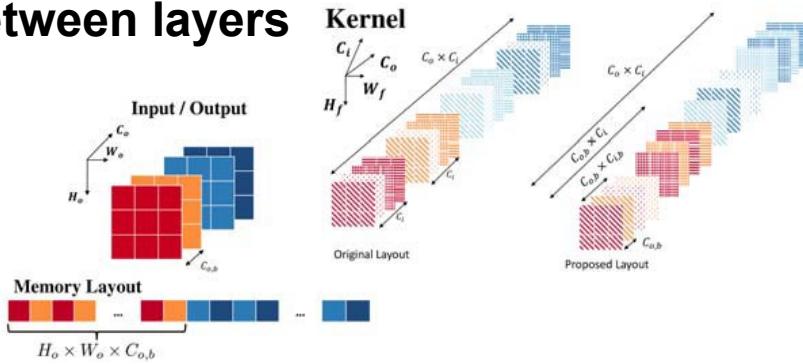


High Performance Direct Convolution



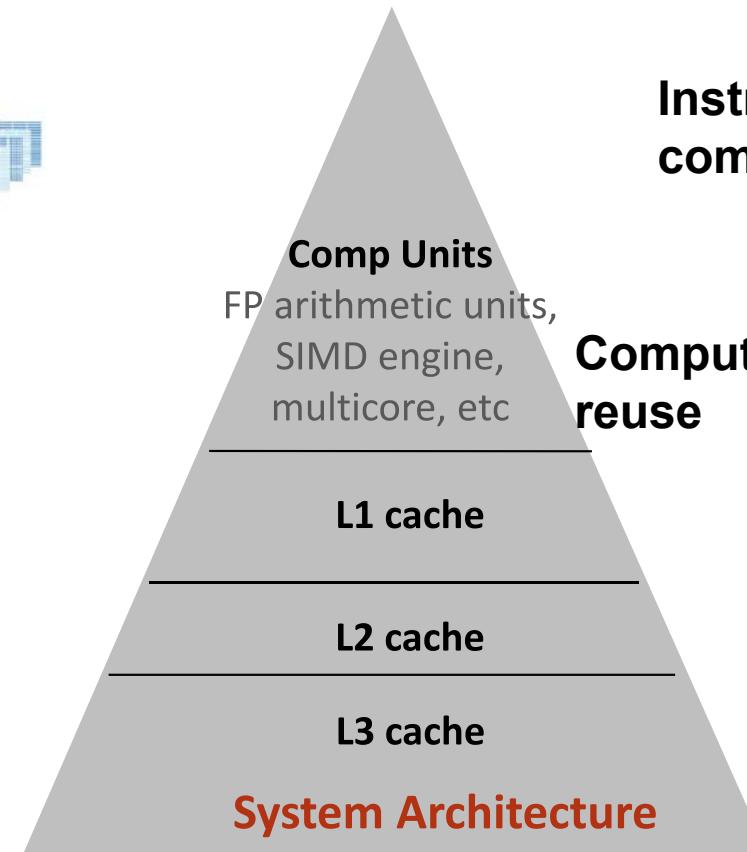
- Analytical model to achieve system theoretical peak performance

**Convolution-friendly data layout
that avoids format reshaping
between layers**



Parallelism to exploit core-level concurrency

```
for j' = 1 to Co/Co,b in parallel do
    for i' = 1 to Ci/Ci,b do
```



Instruction scheduling to saturate the computation pipeline

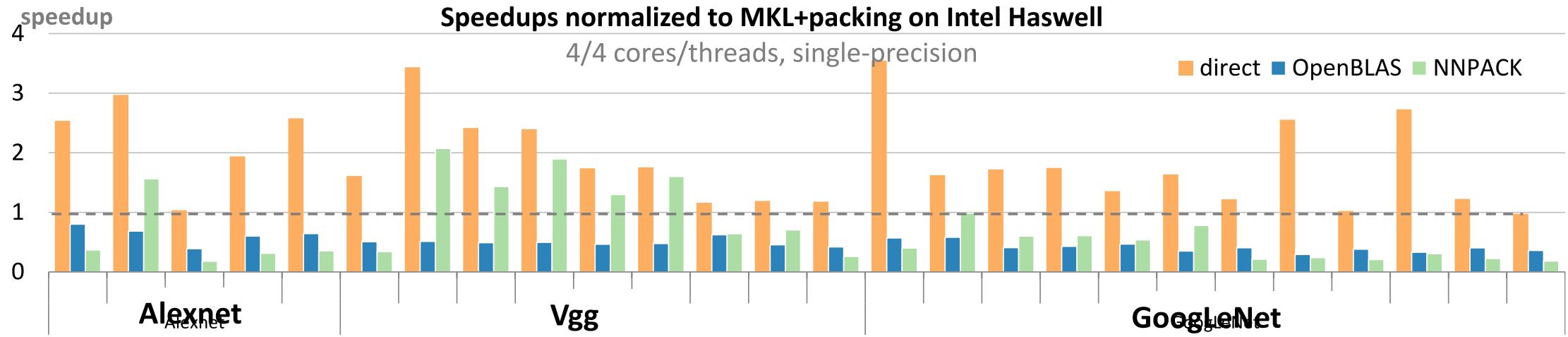
$$\varepsilon \geq N_{vec} N_{fma} L_{fma}$$

Computation reordering that maximizes data reuse

```
for i = 1 to Ci do
    for j = 1 to Co do
        for k=1 to Wo do
            for l = 1 to Ho do
                for m = 1 to Wf do
                    for n = 1 to Hf do
                        Oj,k,l += Ii,k×x+m,l×s+n × Fi,j,m,n
```

**Loop tiling and blocking
to utilize the memory hierarchies**

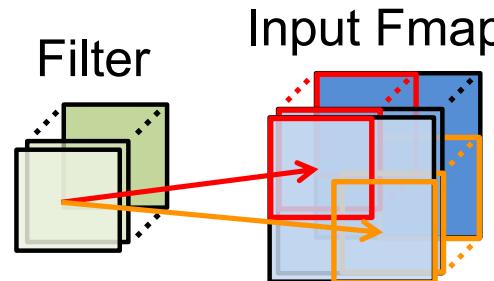
Better Performance



Types of Data Reuse in DNN

Convolutional Reuse

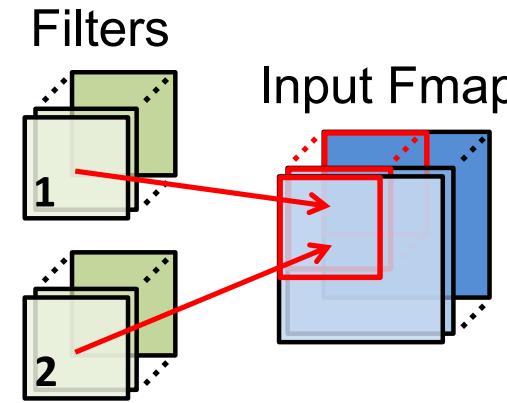
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

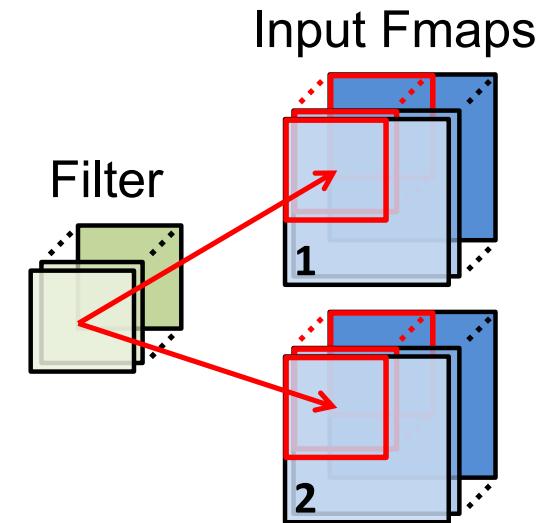
CONV and FC layers



Reuse: Activations

Filter Reuse

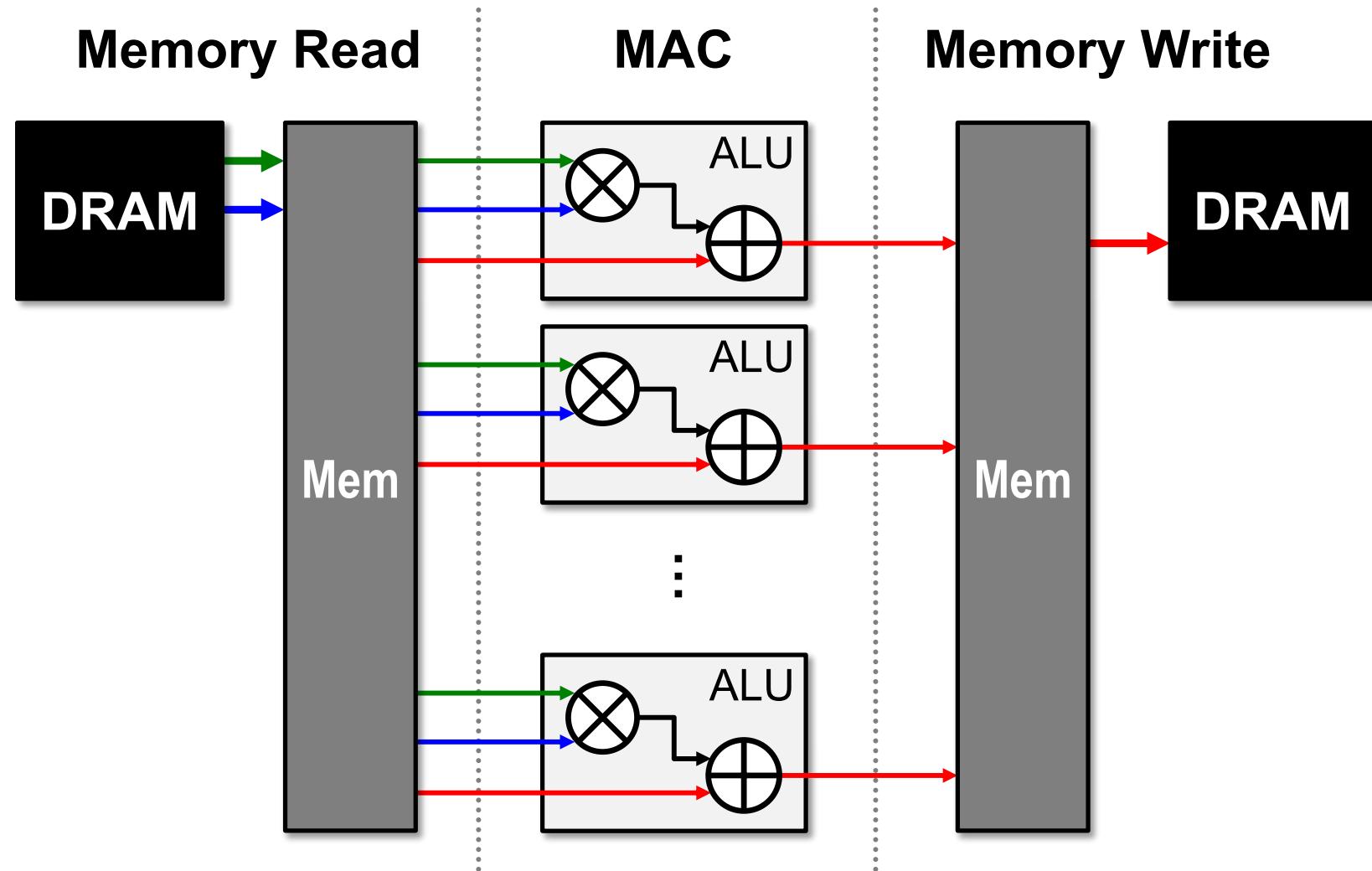
CONV and FC layers
(batch size > 1)



Reuse: Filter weights

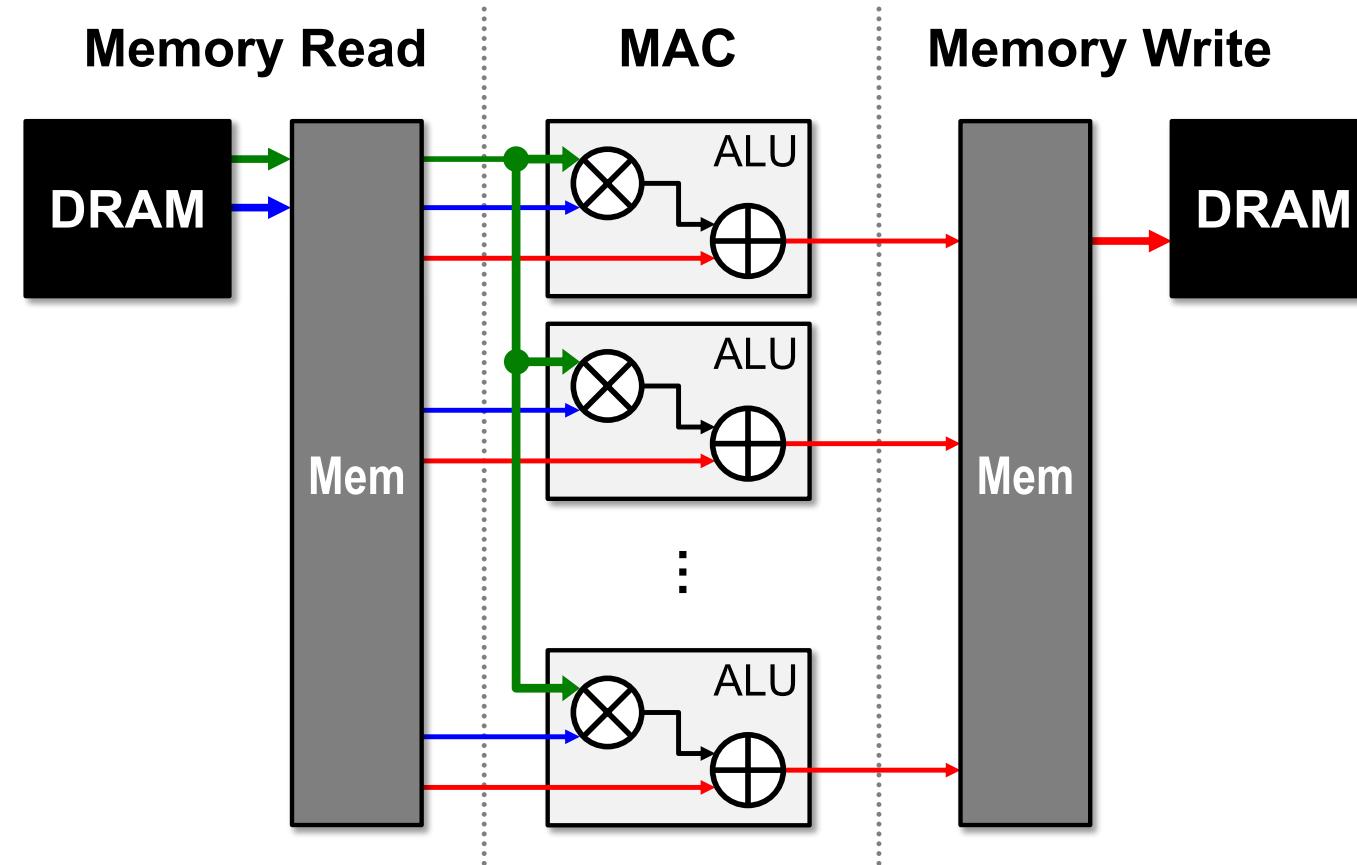
If all data reuse is exploited, DRAM accesses in AlexNet
can be reduced from **2896M** to **61M** (best case)

Parallelism: Parallel MAC Units

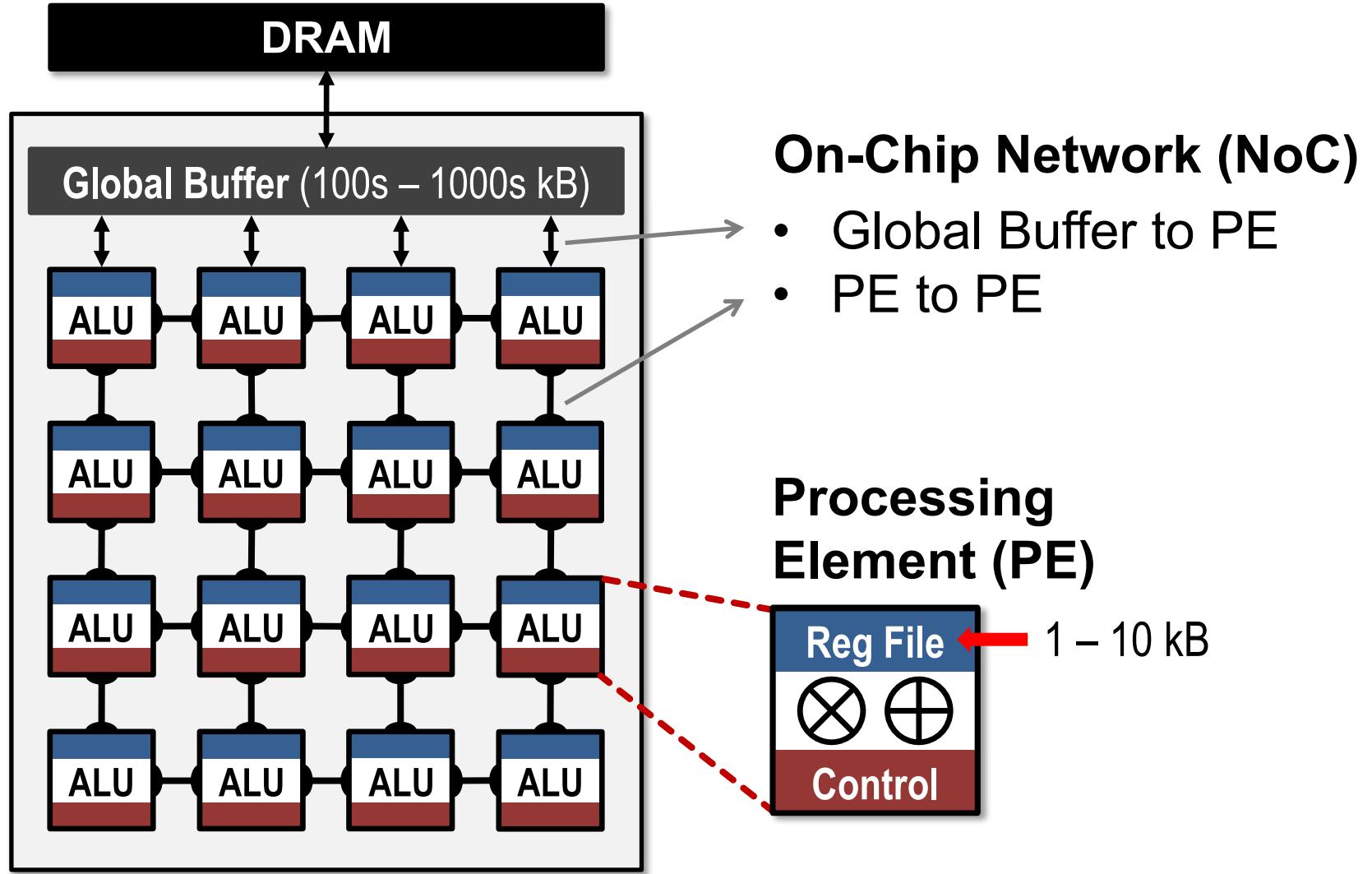


Parallelism: Spatial Data Reuse

- Spatial reuse: the same data is used by **more than one consumer** at **different spatial locations** of the hardware.

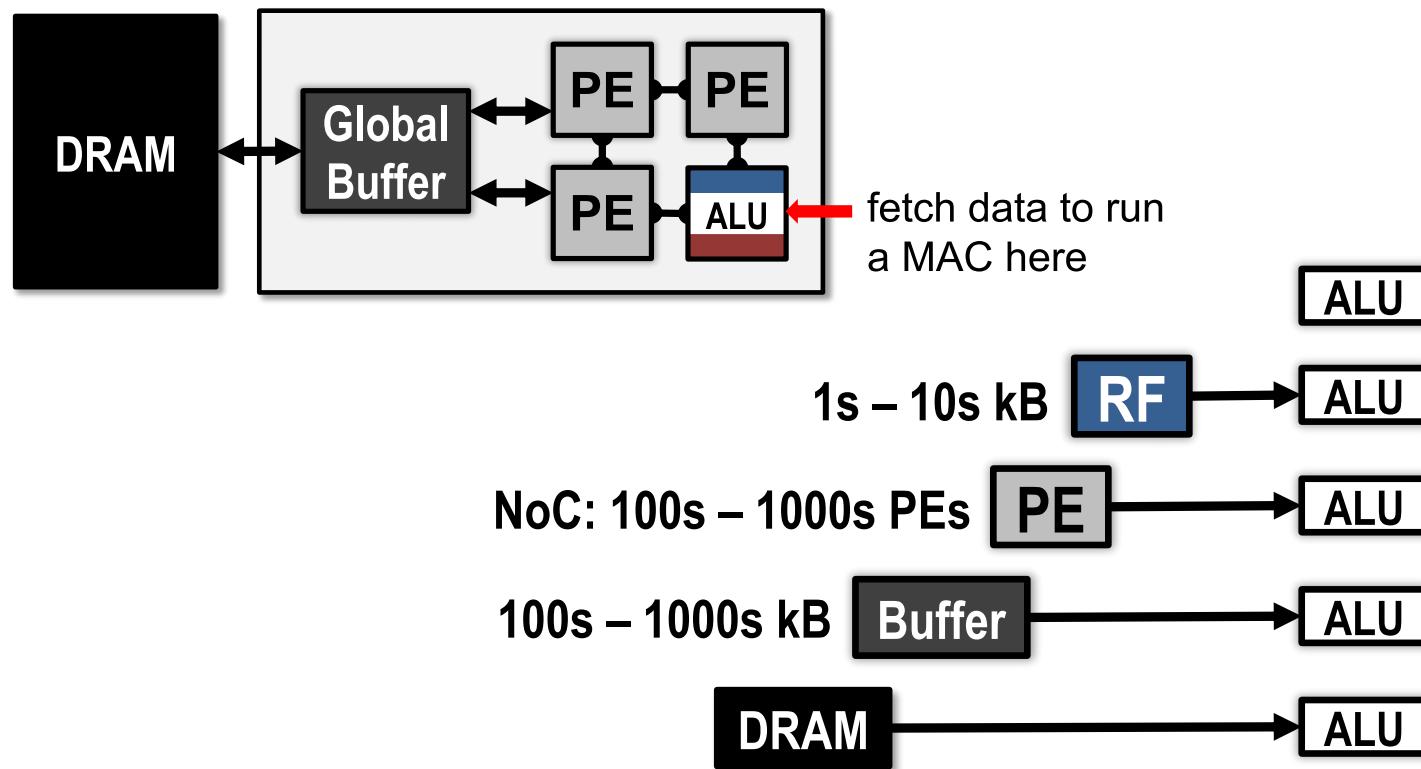


Spatial Architecture for DNN

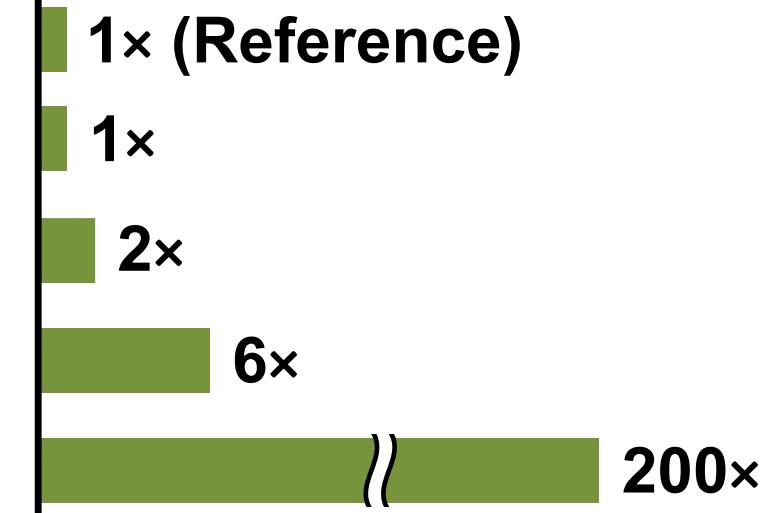


Multi-Level Low-Cost Data Access

- A Dataflow is required to maximally exploit **data reuse** with the **low-cost memory hierarchy** and **parallelism**



Normalized Energy Cost*



Recap CNN Decoder Ring

Shape Parameter	Description
N	Number of input fmmaps/output fmmaps (batch size)
C	Number of 2-D input fmmaps /filters (channels)
H	Height of input fmap (activations)
W	Width of input fmap (activations)
R	Height of 2-D filter (weights)
S	Width of 2-D filter (weights)
M	Number of 2-D output fmmaps (channels)
E	Height of output fmap (activations)
F	Width of output fmap (activations)

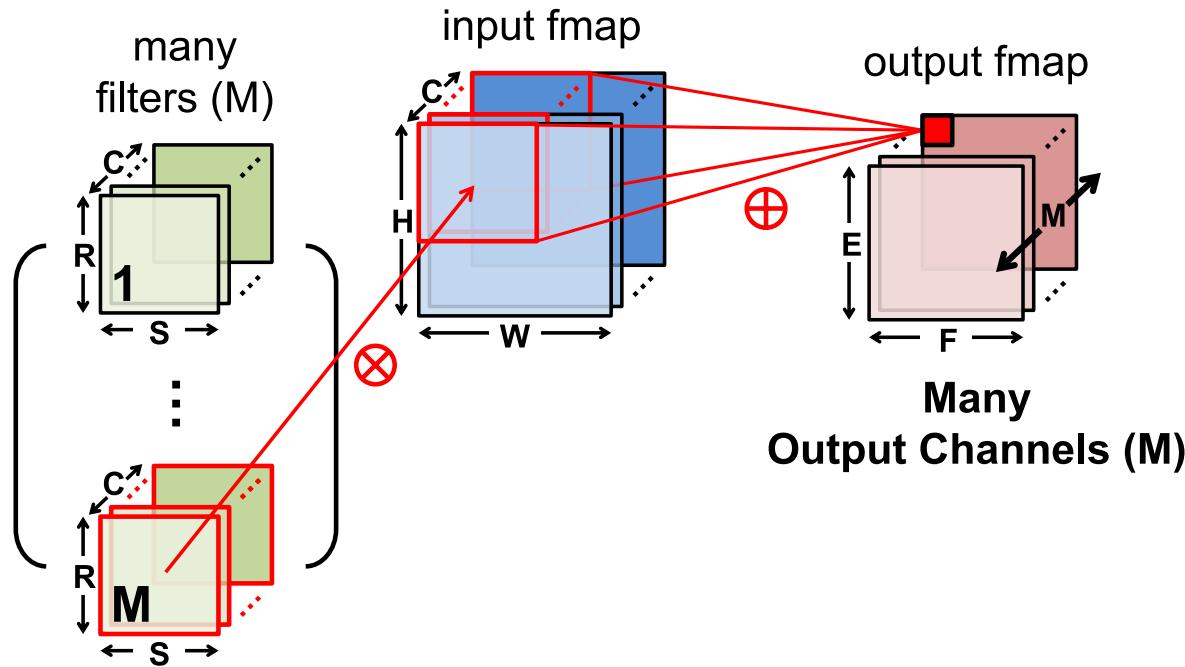
Dataflow and Loop Nest



$$O_{nmef} = \left(\sum_{crs} I_{nc(e+r)(f+s)} F_{mcrs} \right) + b_m$$

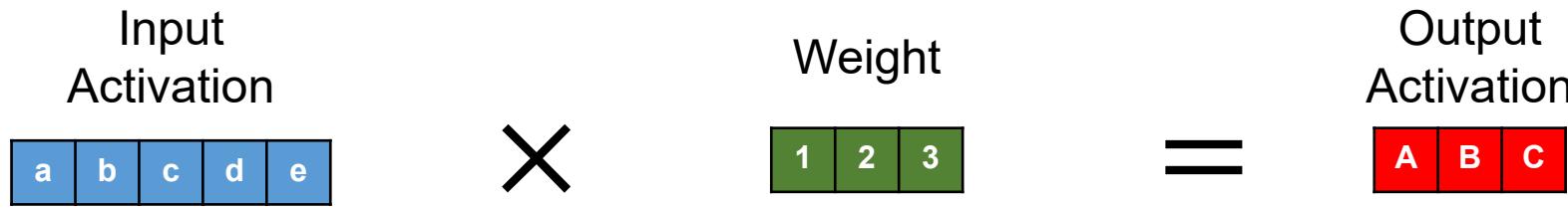
- No ordering or notion of parallelism on the individual calculations
 - But are important for Hardware for performance and efficiency computation
- Dataflow
 - Specifying an **ordering** and which calculation **run in parallel**
 - Computation order
 - Data movement order
- Loop Nest
 - Describe Various properties of a given DNN accelerator design
 - Describe Dataflow
 - **for**: temporal for, describes the temporal execution order
 - **spatial_for**: describes parallel execution

Data Reuse in DNN



- Total # of MACs:
 - RSCEFM
- Input Activation
 - Size: HWC
 - Reuse: ~RSM
- Weight
 - Size: RSCM
 - Reuse: EF
- Output Activation
 - Size: EFM
 - Reuse: RSC

1D Convolution Example



```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for r in range(R):
    for e in range(E):
        w = e + r
        o[e] += i[w] * f[r]
```

**Weight Stationary
(WS) Dataflow**

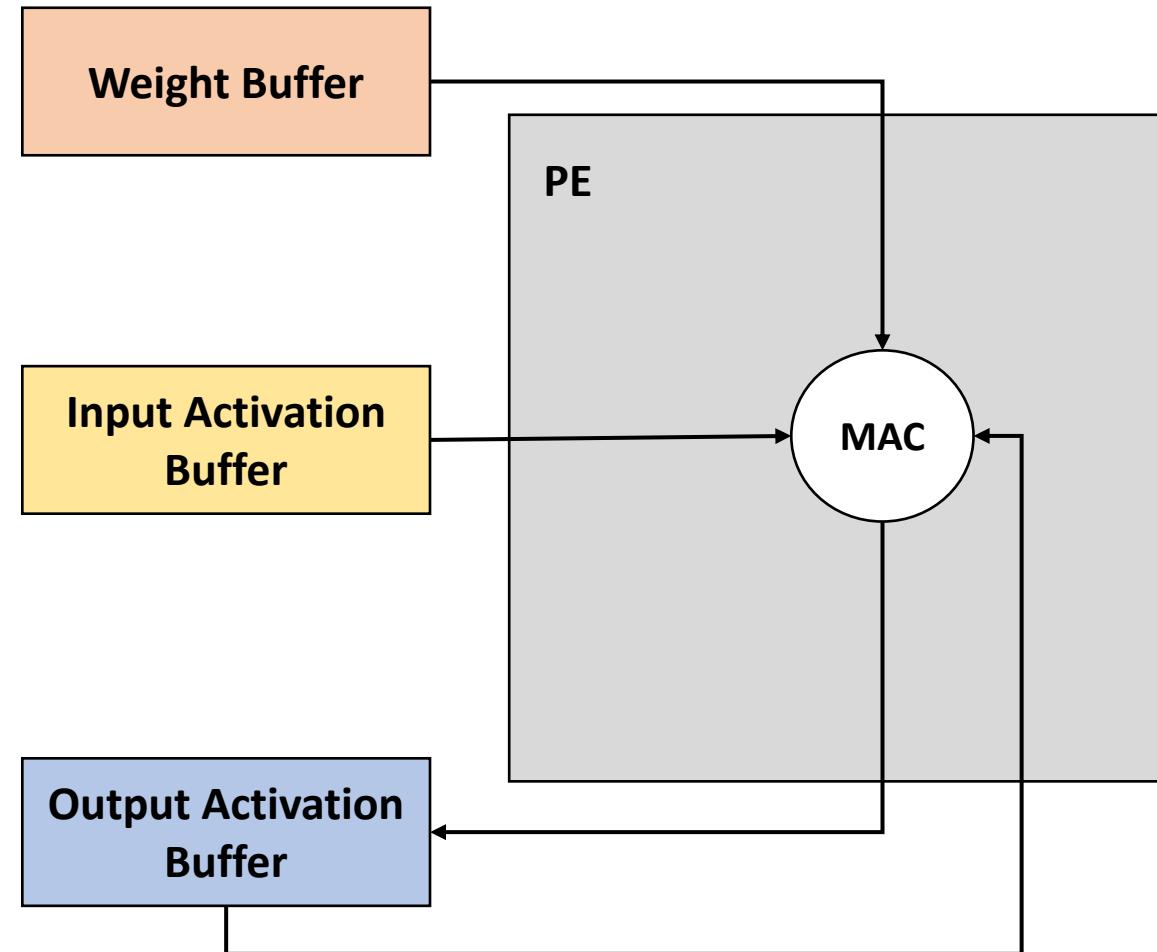
```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for e in range(E):
    for r in range(R):
        w = e + r
        o[e] += i[w] * f[r]
```

**Output Stationary
(OS) Dataflow**

```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for w in range(W):
    for r in range(R):
        e = w - r
        o[e] += i[w] * f[r]
```

**Input Stationary
(IS) Dataflow**

Single Processing Engine(PE) Setup



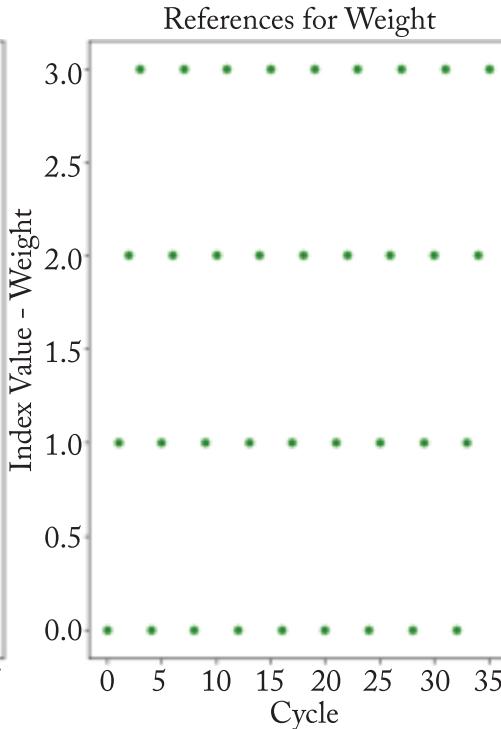
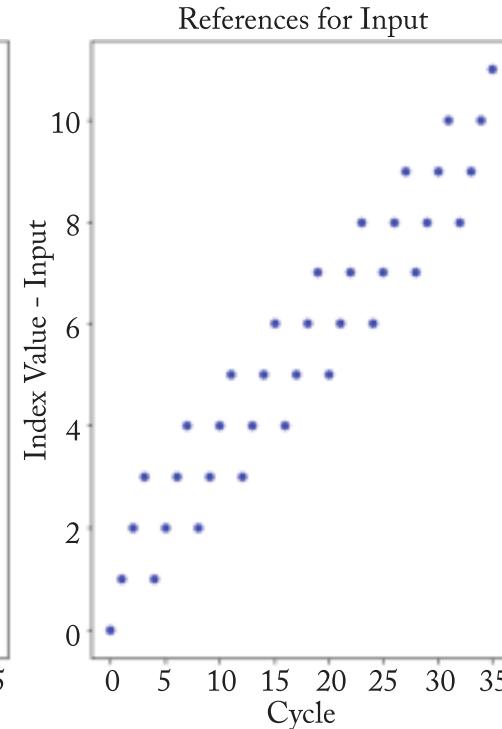
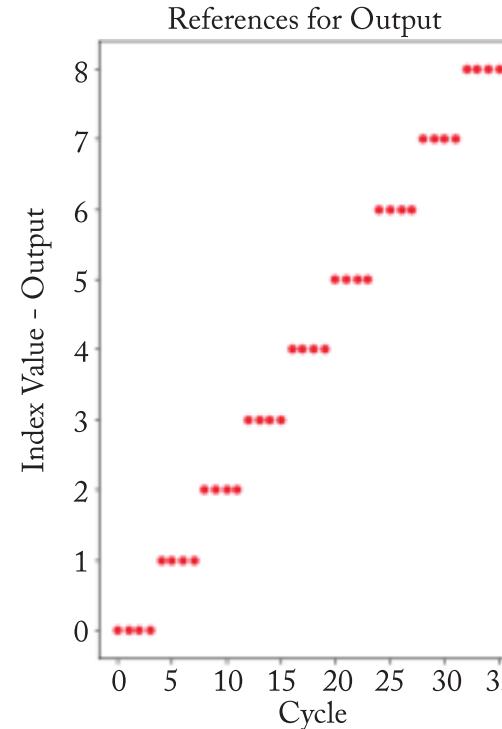
Output Stationary – Reference Pattern

```

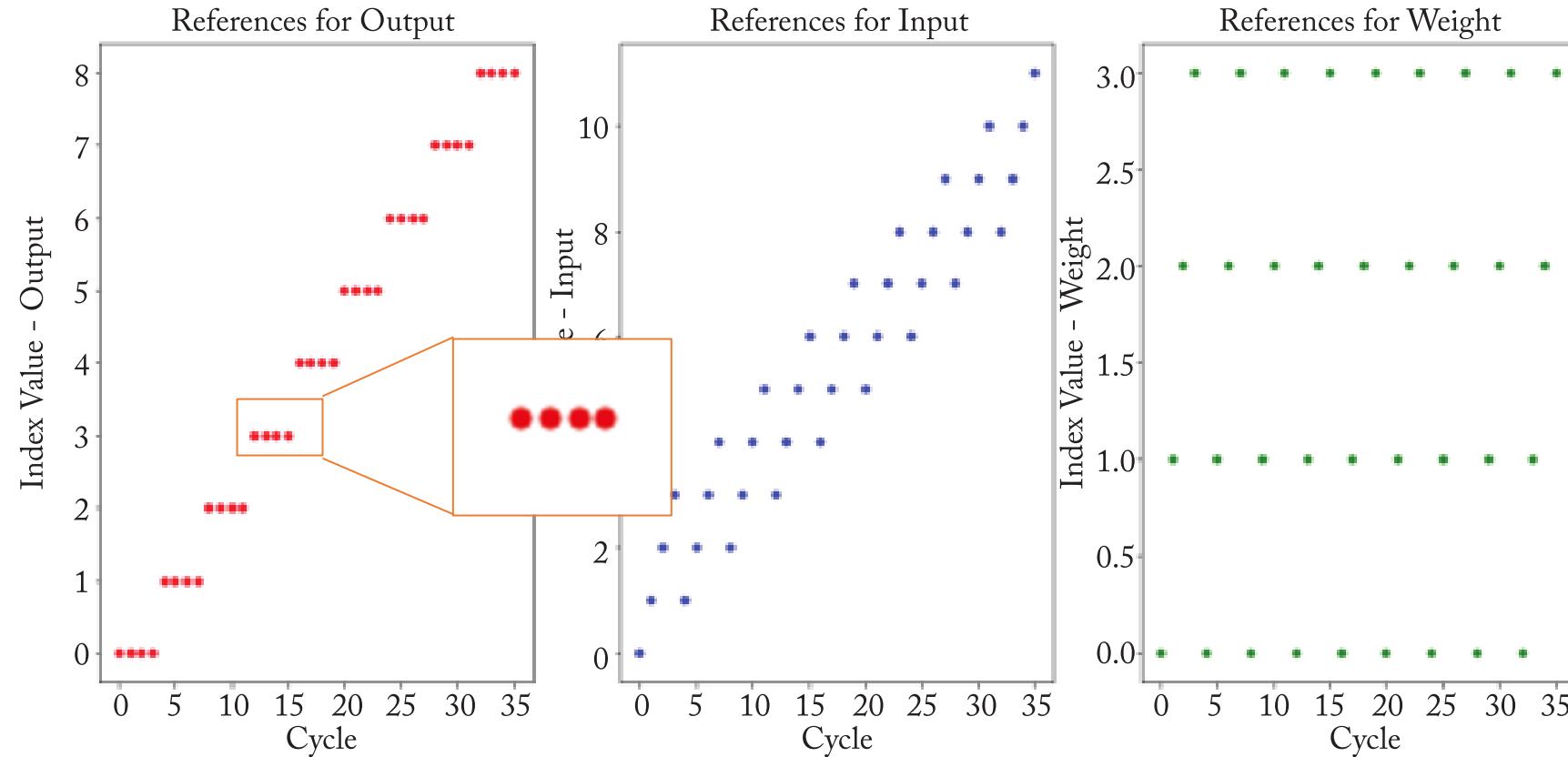
for (e = 0 ; e < E ; e++)
  for (r = 0 ; r < R ; r++)
    O[e] += I[e+r] * W[r];
  
```

Layer Shape

- $W = 12$
- $R = 4$
- $E = 9$

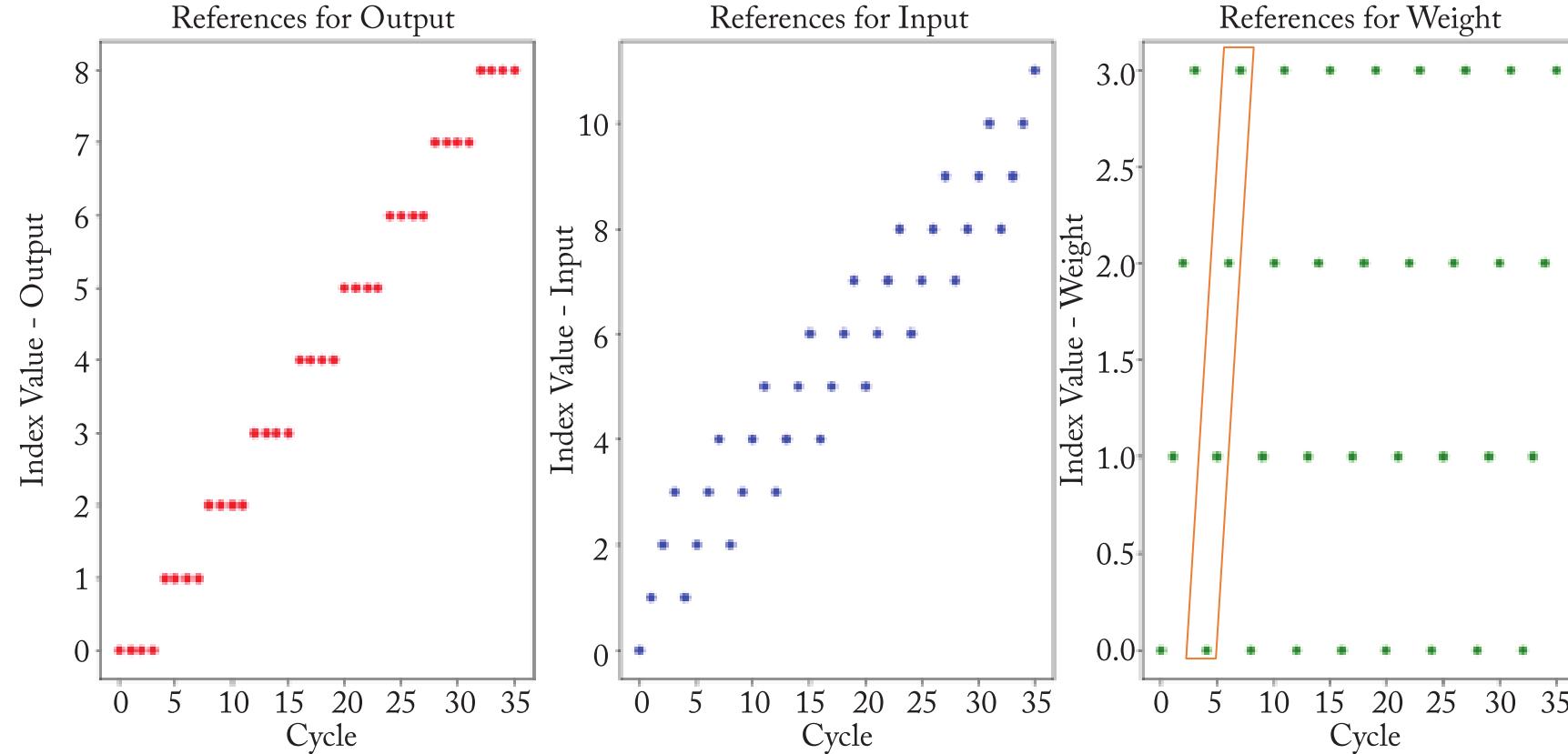


Output Stationary – Reference Pattern



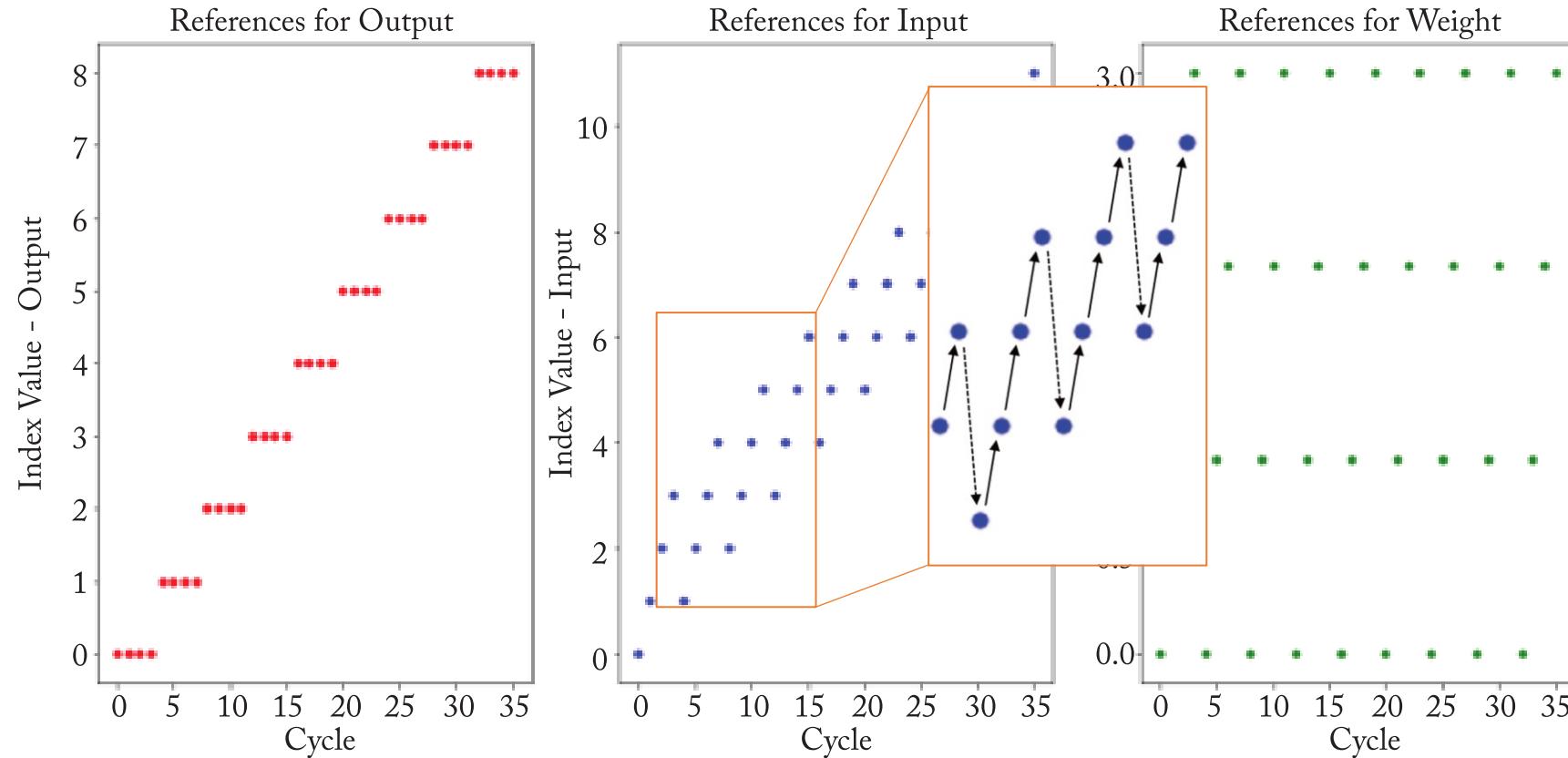
- Single output is reused many times (R)

Output Stationary – Reference Pattern



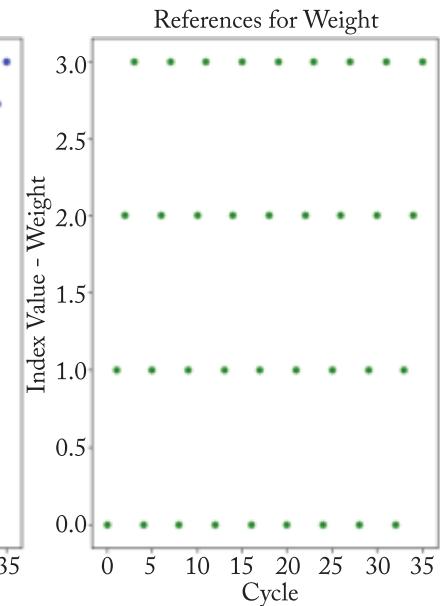
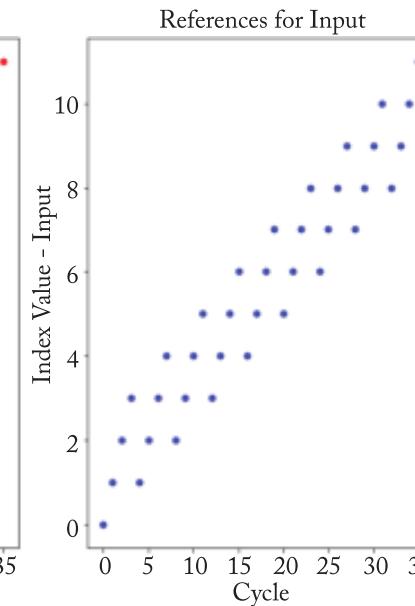
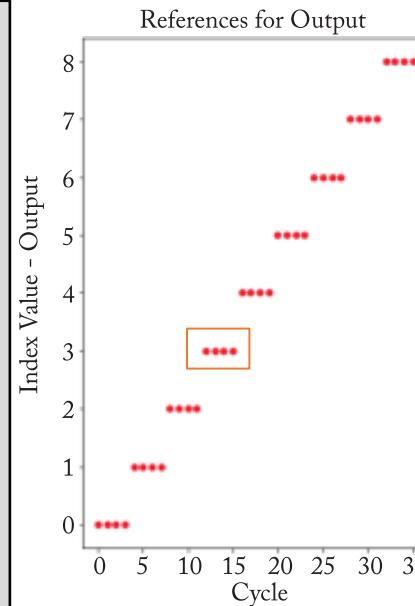
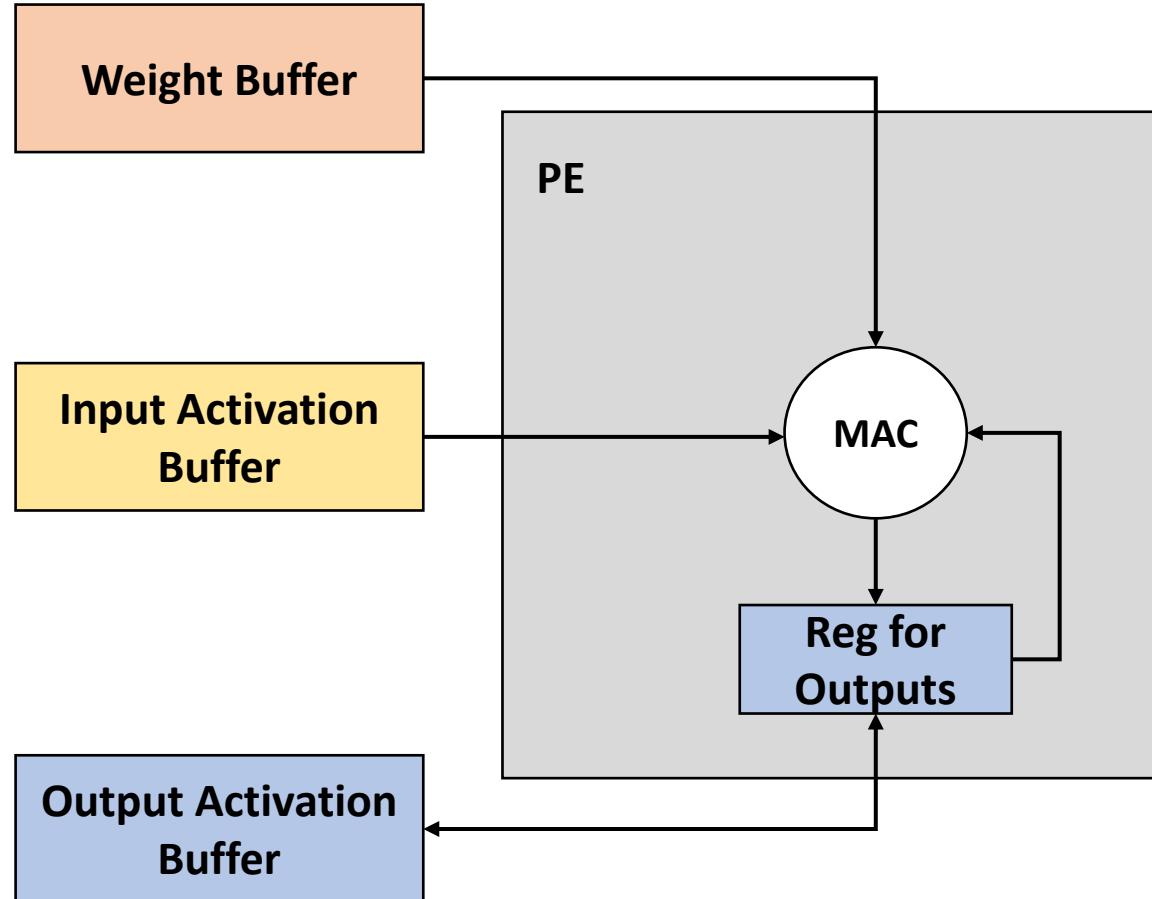
- Single output is reused many times (R)
- All weights reused repeatedly

Output Stationary – Reference Pattern



- Single output is reused many times (R)
- All weights reused repeatedly
- Sliding window of inputs (size = R)

Buffer Access Pattern (OS)

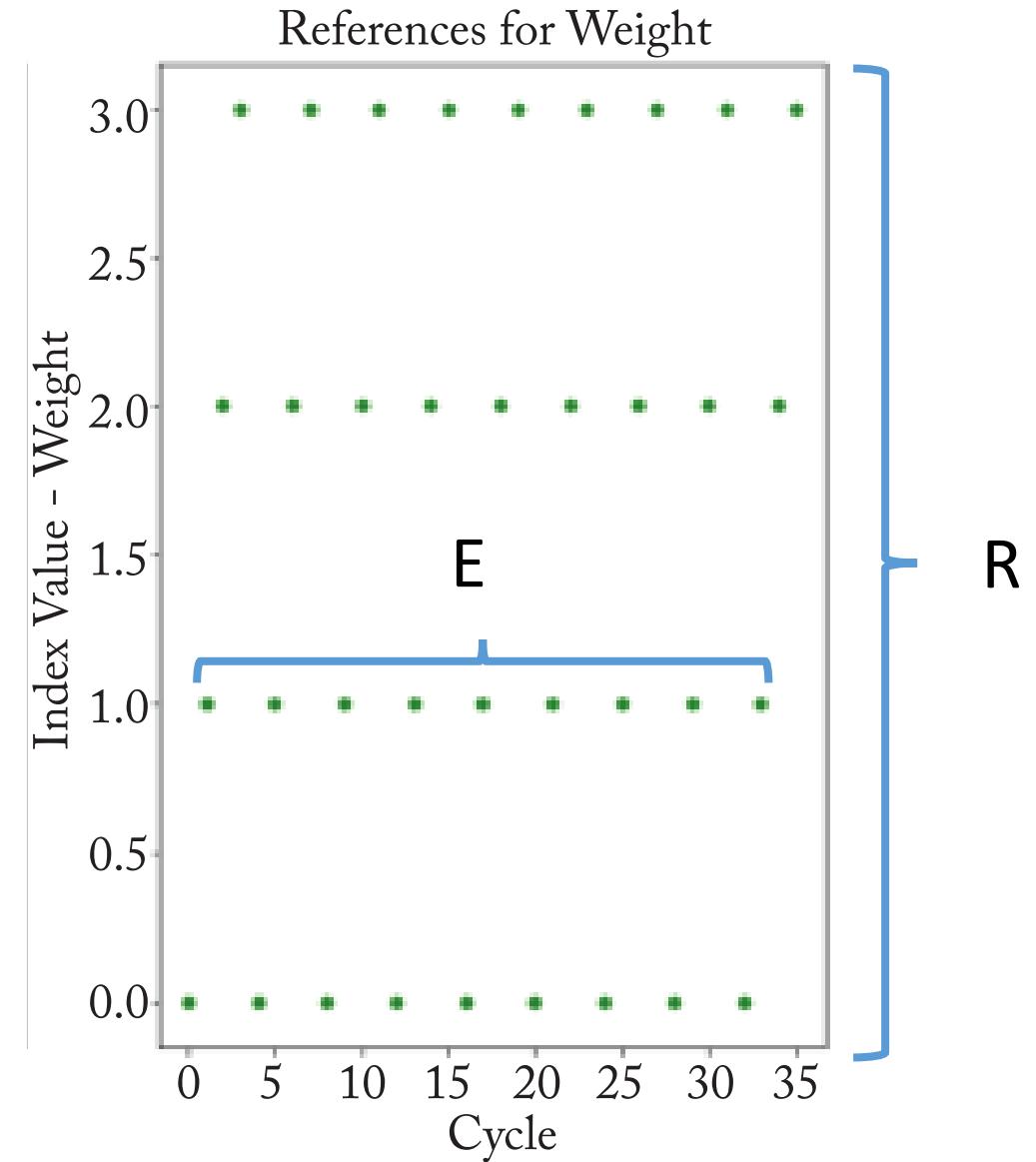


Buffer Data Accesses

```

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

	OS
MACs	$E \cdot R$
Weight Reads	$E \cdot R$
Input Reads	
Output Reads	
Output Writes	



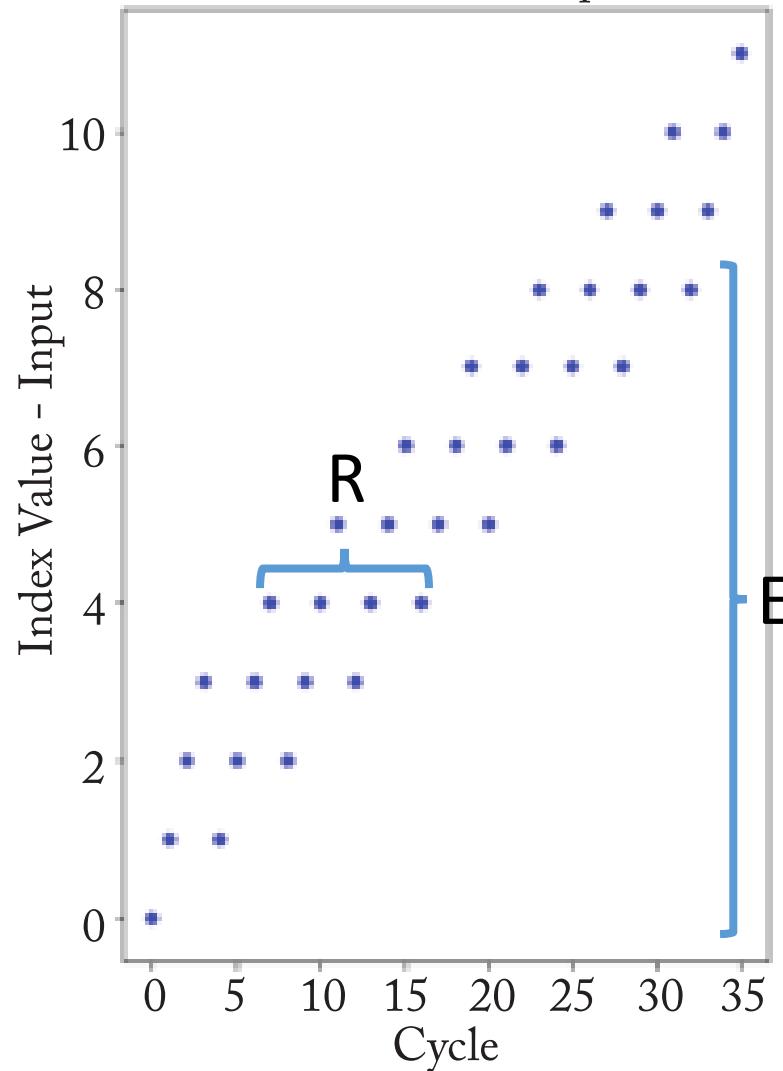
Buffer Data Accesses

```

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

	OS
MACs	$E \cdot R$
Weight Reads	$E \cdot R$
Input Reads	$E \cdot R$
Output Reads	
Output Writes	

References for Input

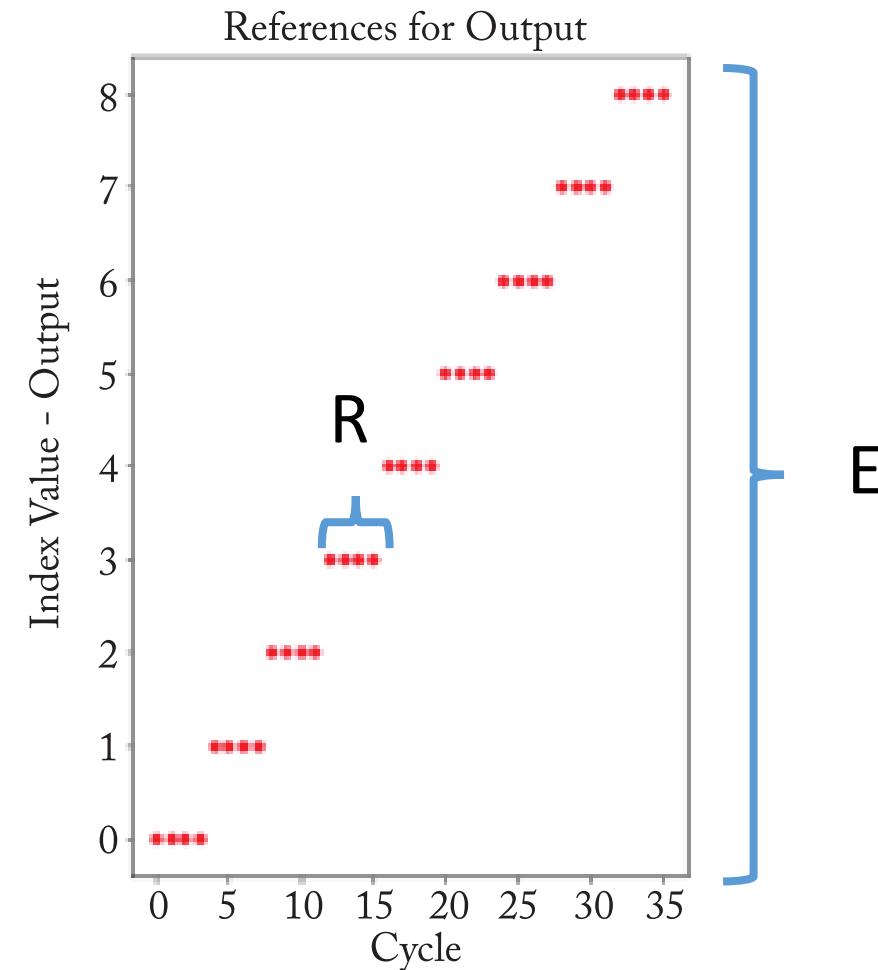


Buffer Data Accesses

```

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

	OS
MACs	$E \cdot R$
Weight Reads	$E \cdot R$
Input Reads	$E \cdot R$
Output Reads	0
Output Writes	E



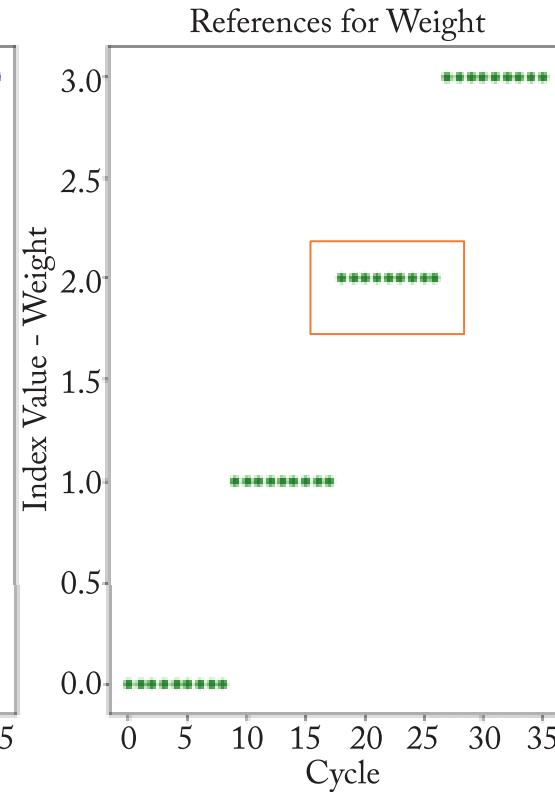
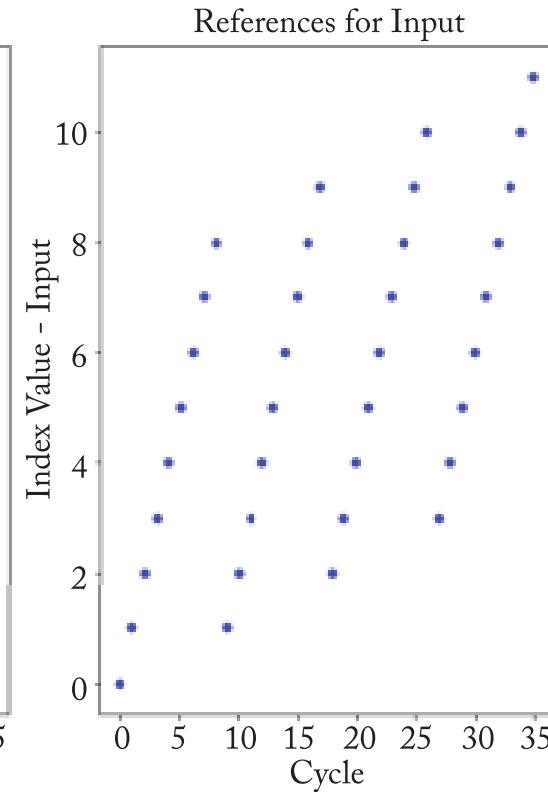
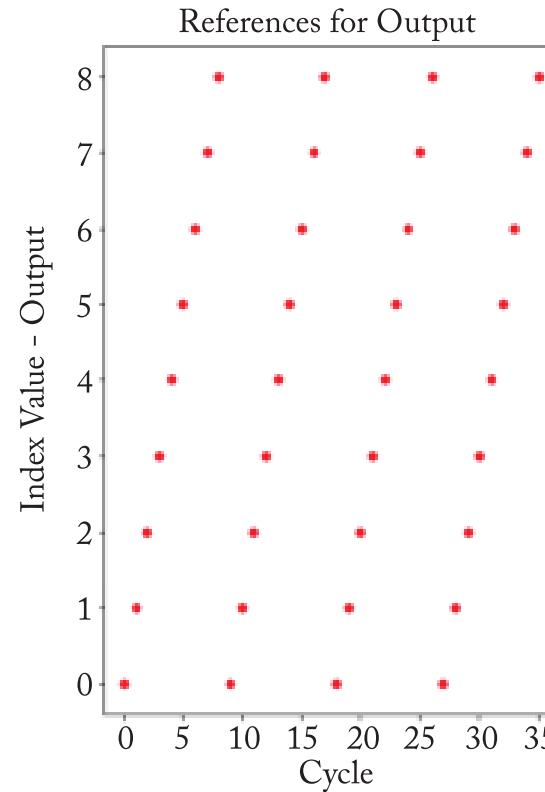
Weight Stationary – Reference Pattern



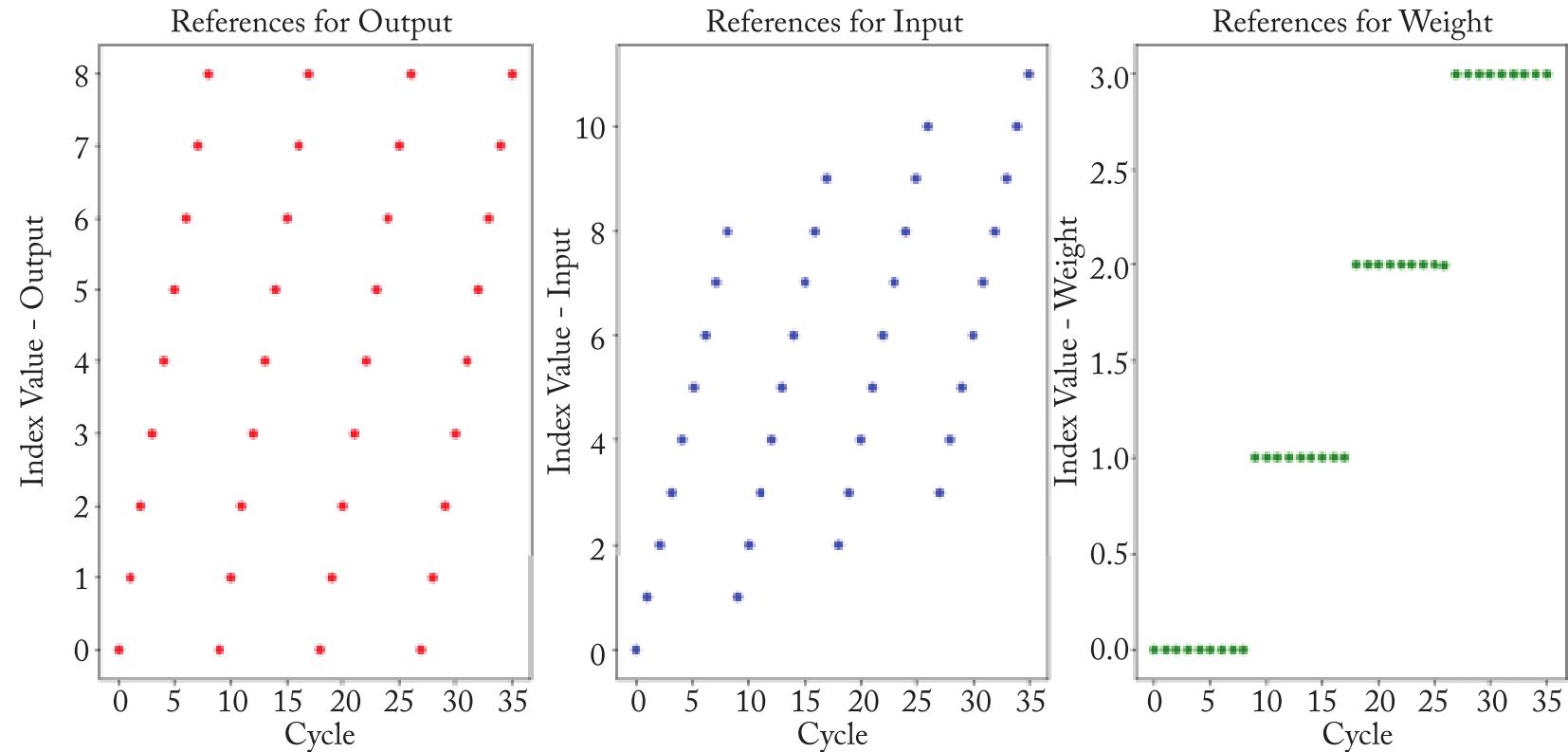
```
for (e = 0 ; e < E ; e++)  
  for (r = 0 ; r < R ; r++)  
    O[e] += I[e+r] * W[r];
```

No constraints on
loop permutations!

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

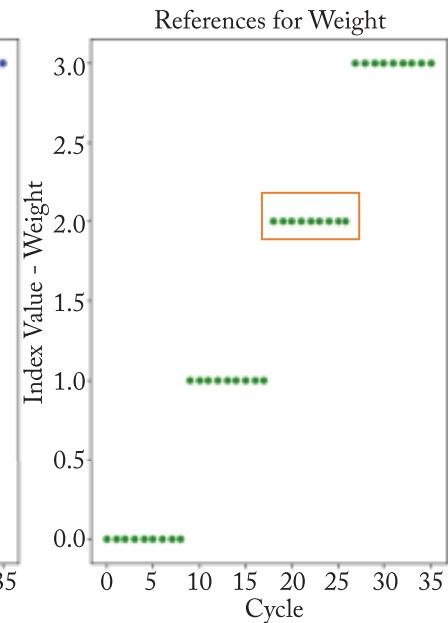
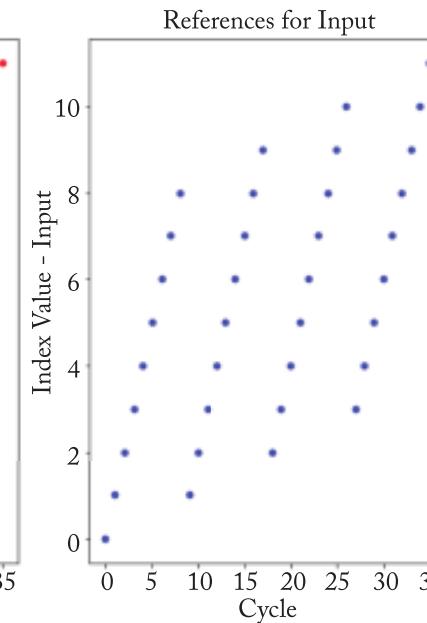
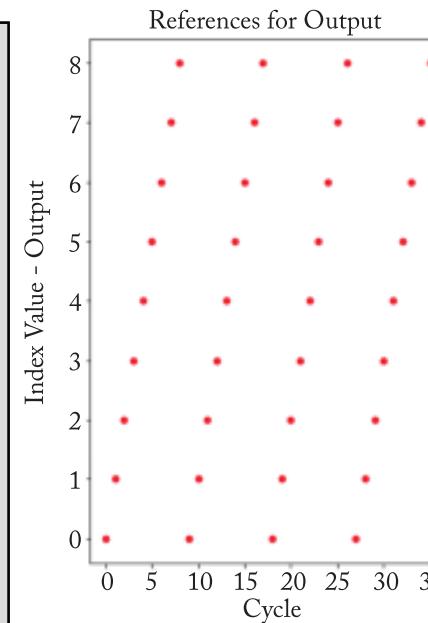
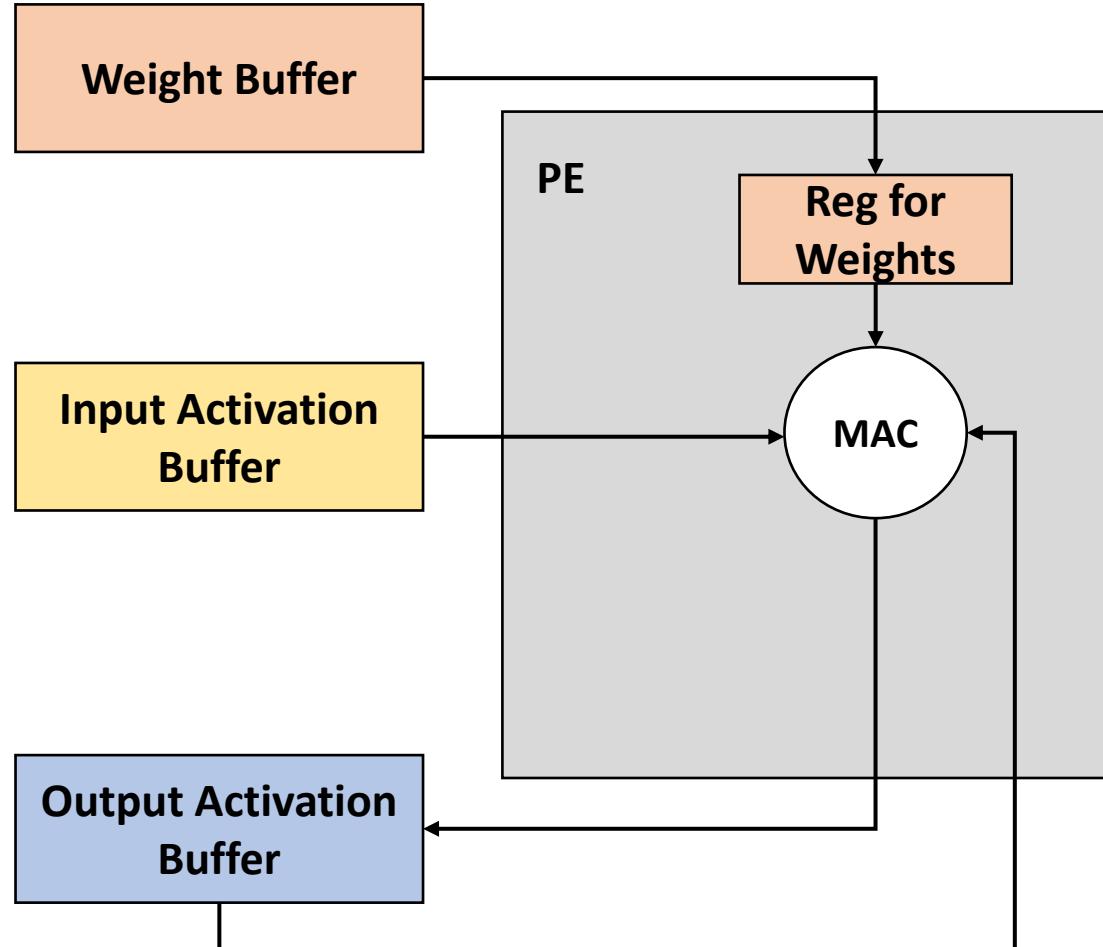


Weight Stationary– Reference Pattern



- Single weight is reused many times (E)
- Large sliding window of inputs (size = E)
- Fixed window of outputs (size = E)

Buffer Access Pattern (WS)



L1 Weight Stationary - Costs



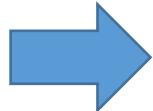
```
for (r = 0 ; r < R ; r++)
    for (e = 0 ; e < E ; e++)
        O[e] += I[e+r] * W[r];
```

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

Tiled Loop Nest For Weight-stationary



```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for r in range(R):
    for e in range(E):
        w = e + r
        o[e] += i[w] * f[r]
```

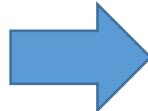


```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for e1 in range(E1):
    for r in range(R):
        for e0 in range(E0):
            e = e1 * E0 + e0
            w = e + r
            o[e] += i[w] * f[r]
```

Parallel Processing for The A Tile of Filter Weights



```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for r in range(R):
    for e in range(E):
        w = e + r
        o[e] += i[w] * f[r]
```



```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for r1 in range(R1):
    for e in range(E):
        spatial_for r0 in range(R0):
            r = r1 * R0 + r0
            w = e + r
            o[e] += i[w] * f[r]
```

Reference Pattern for Parallel Processing



```
# i[W] - input activations
# f[R] - filter weights
# o[E] - output activations
for r1 in range(R1):
    for e in range(E):
        spatial_for r0 in range(R0):
            r = r1 * R0 + r0
            w = e + r
            o[e] += i[w] * f[r]
```

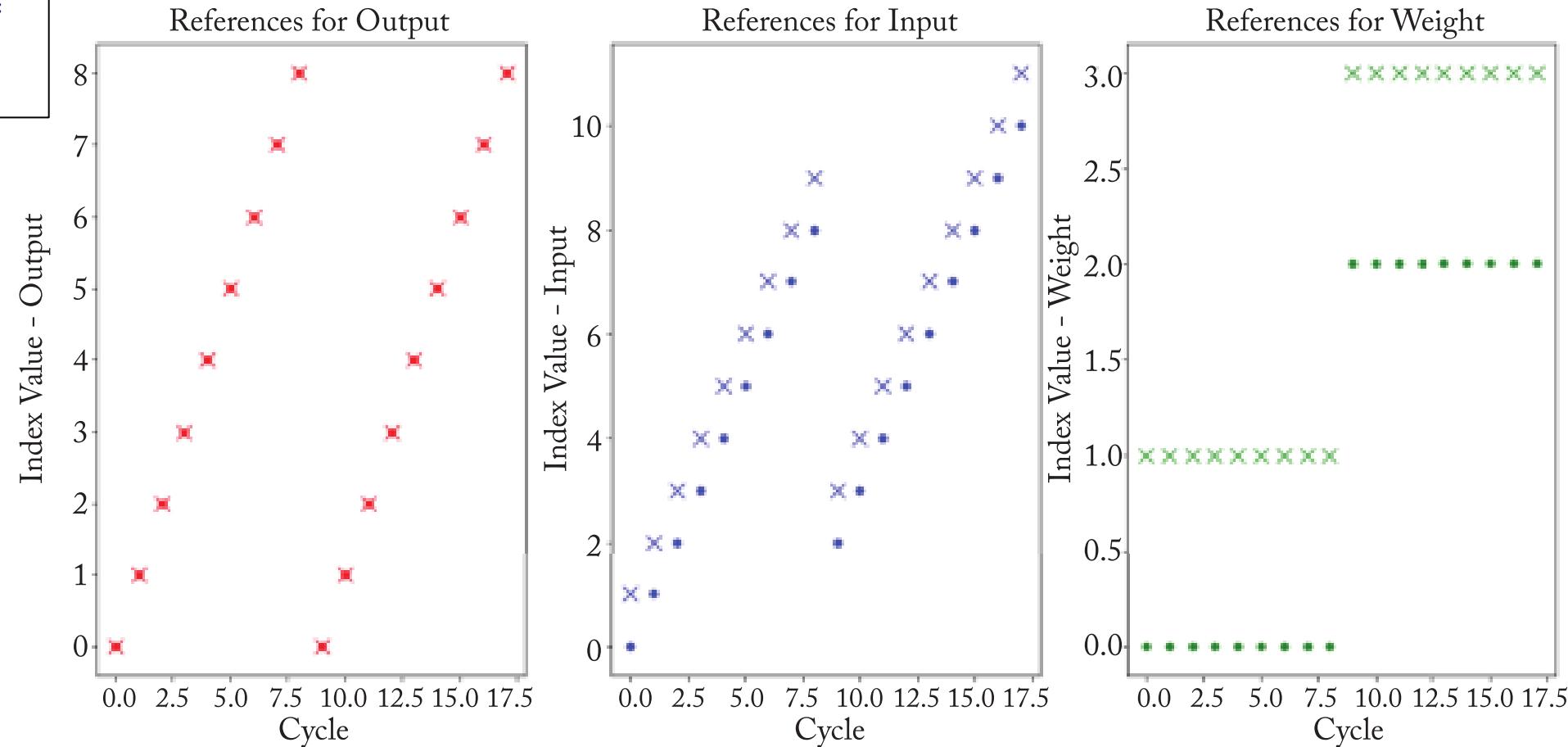
R=4

E=9

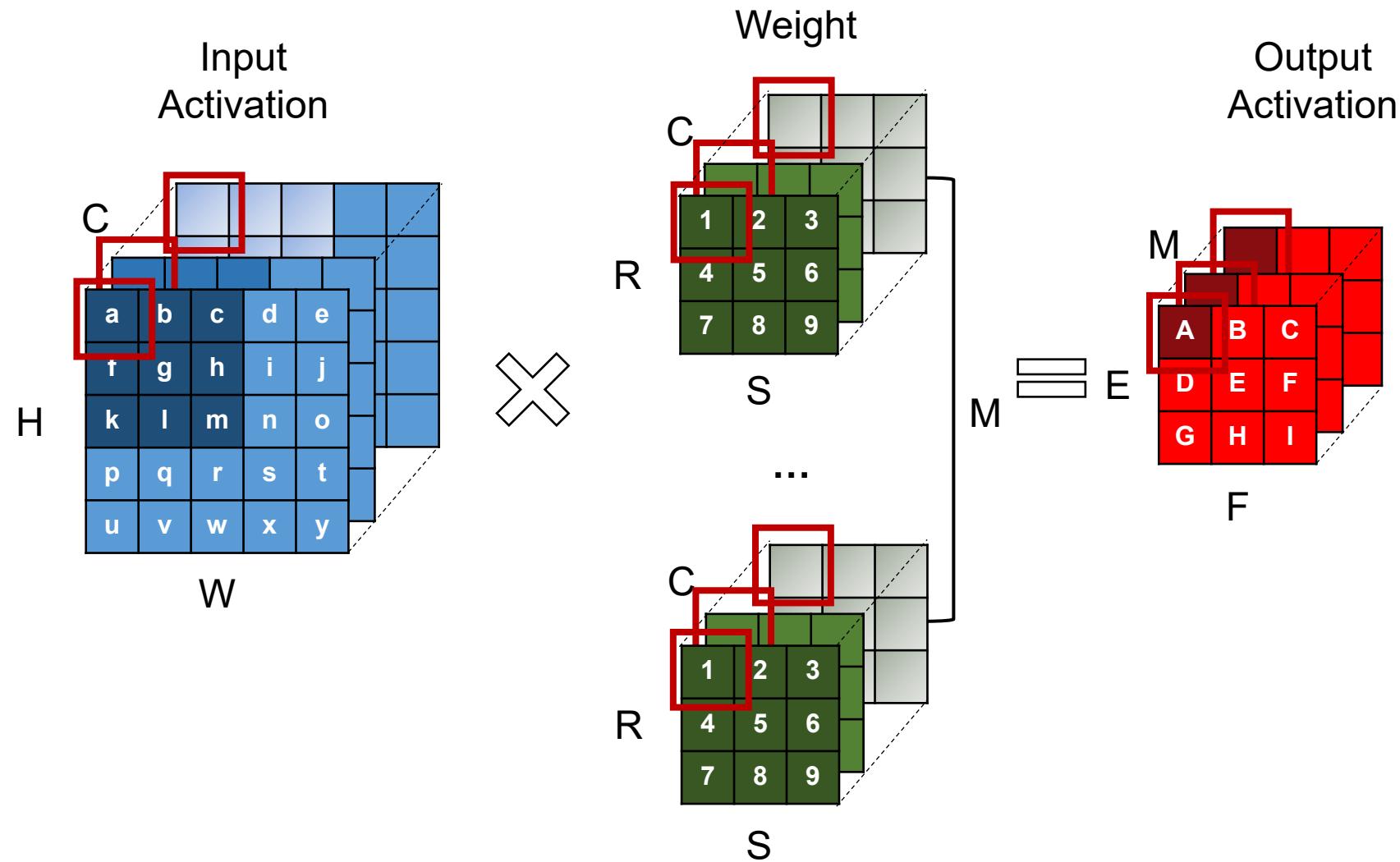
W=12

R1=2

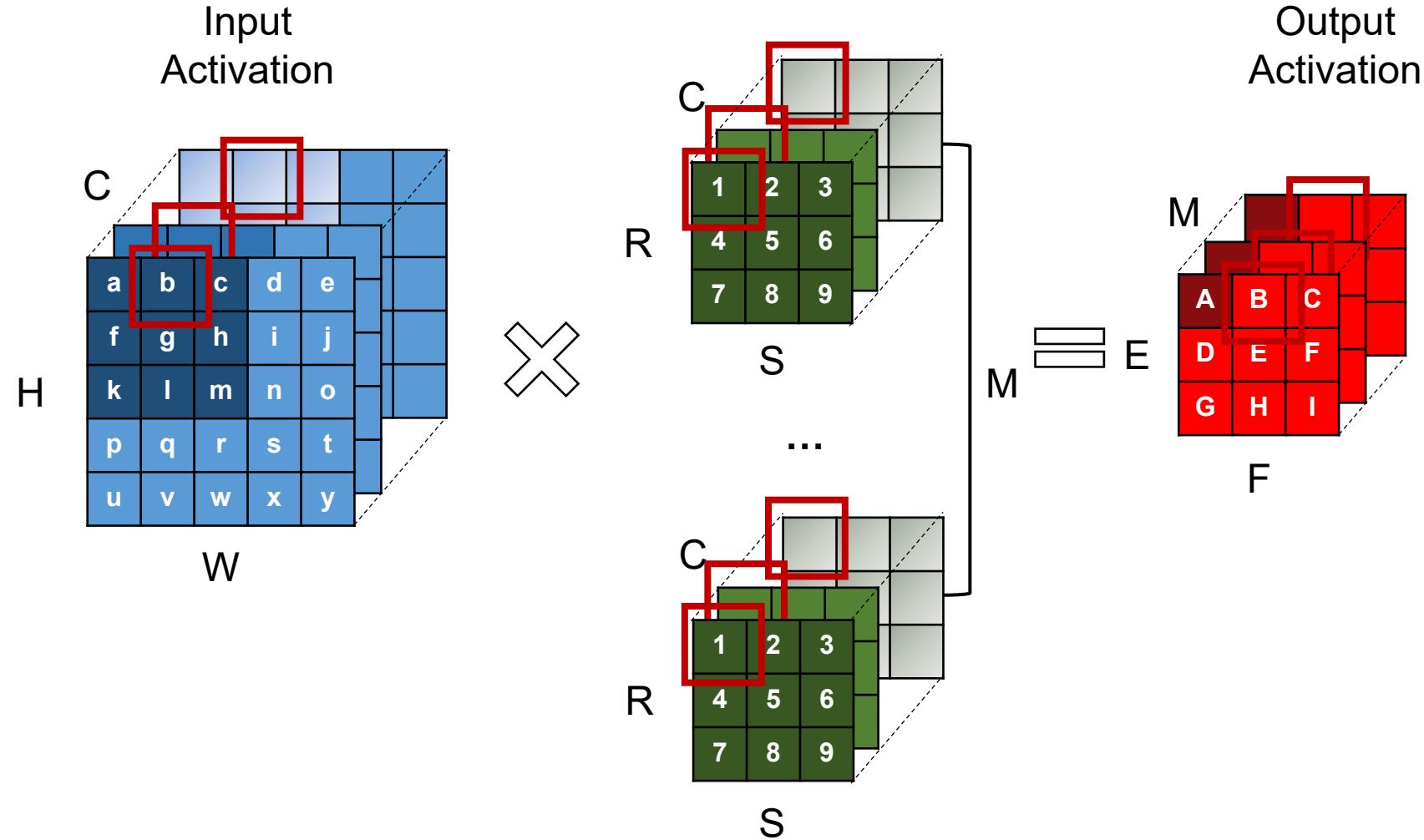
R0=2



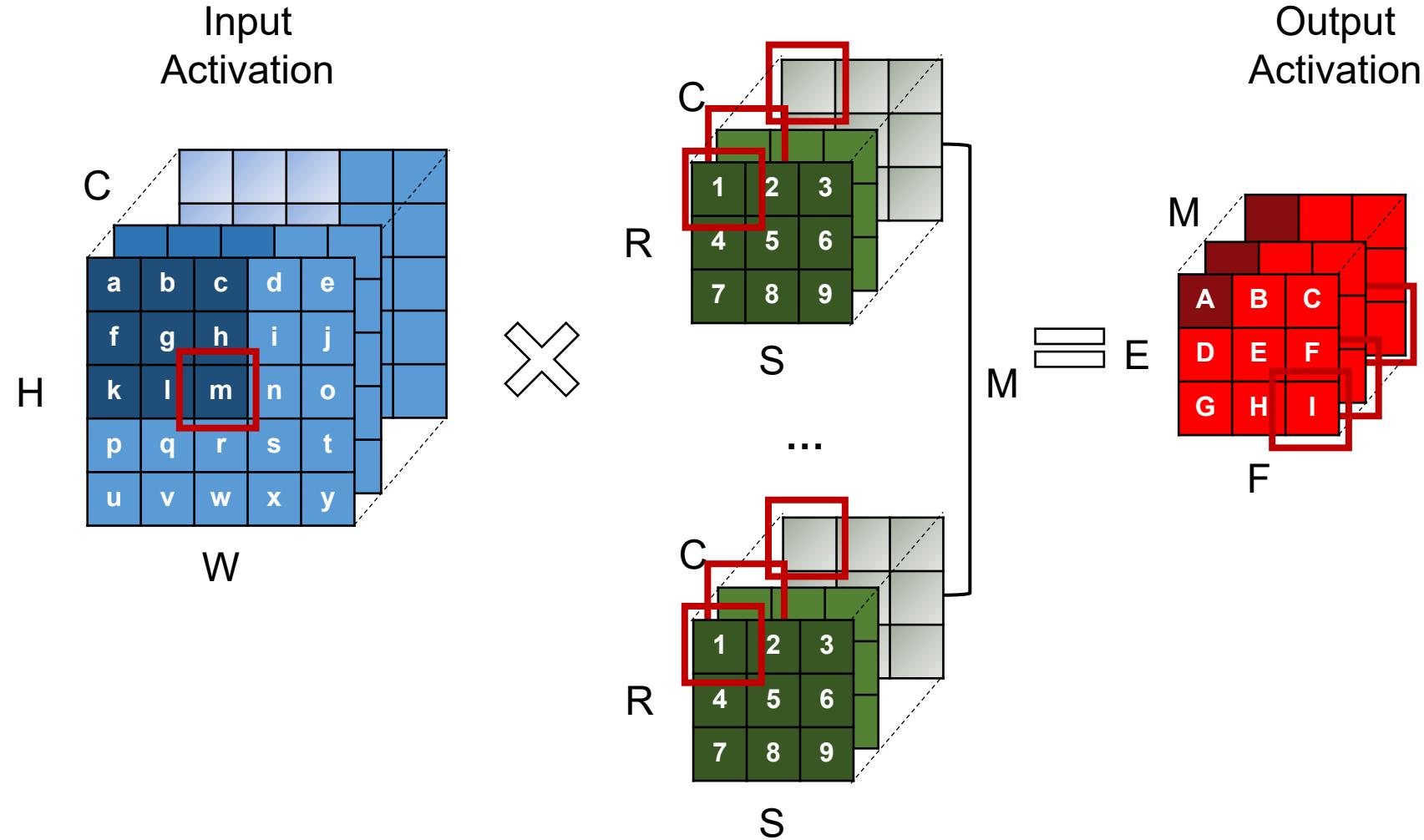
Weight Stationary Dataflow



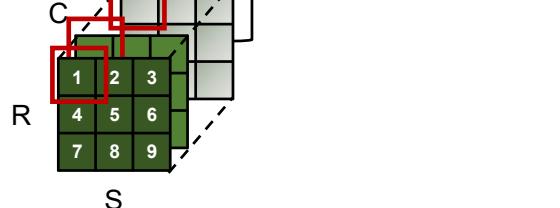
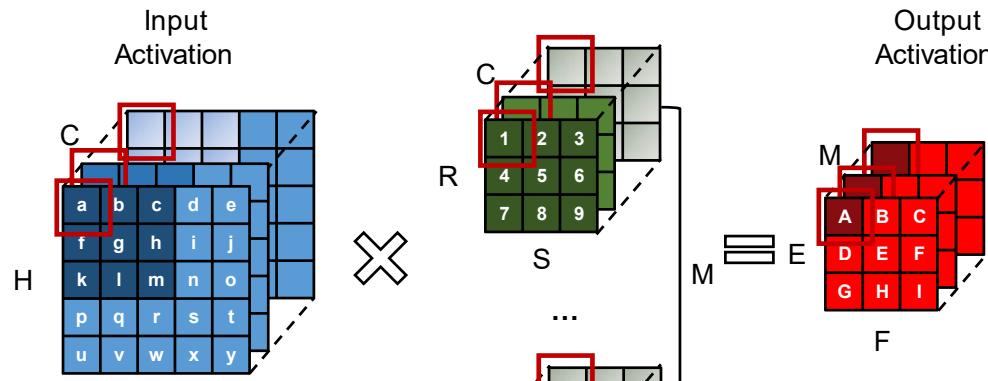
Weight Stationary Dataflow



Weight Stationary Dataflow



Weight Stationary Dataflow



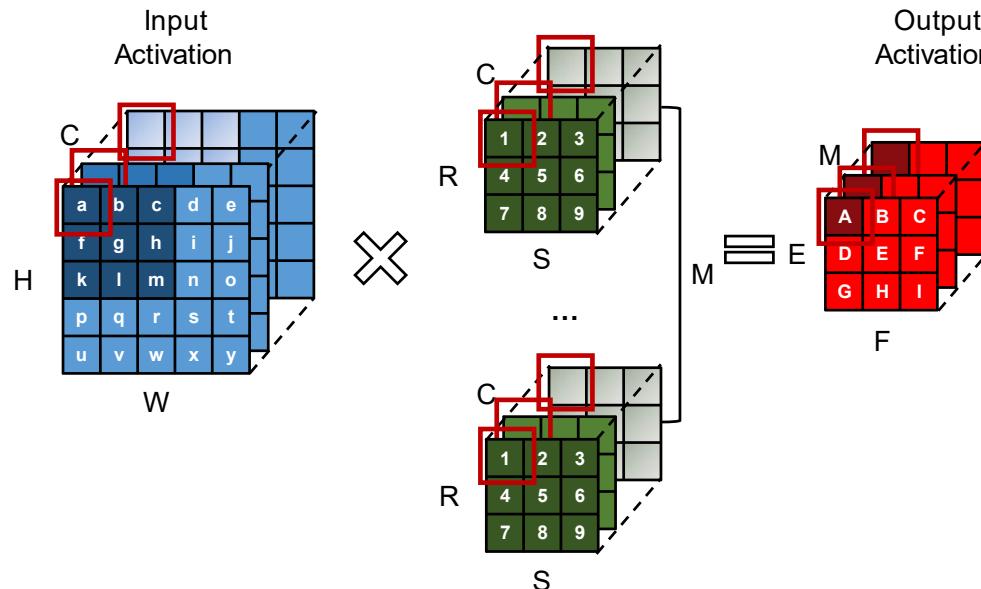
- Original nested loop:

```

for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (e=0; e<E; e++) {
            for (f=0; f<F; f++) {
                OA[n][m][e][f] = 0;
                for (r=0; r<R; r++) {
                    for (s=0; s<S; s++) {
                        for (c=0; c<C; c++) {
                            h = e * stride - pad + r;
                            w = f * stride - pad + s;
                            OA[n][m][e][f] +=
                                IA[n][c][h][w]
                                * W[m][c][r][s];
                        }
                    }
                }
            }
        }
    }
}

```

Weight Stationary Dataflow

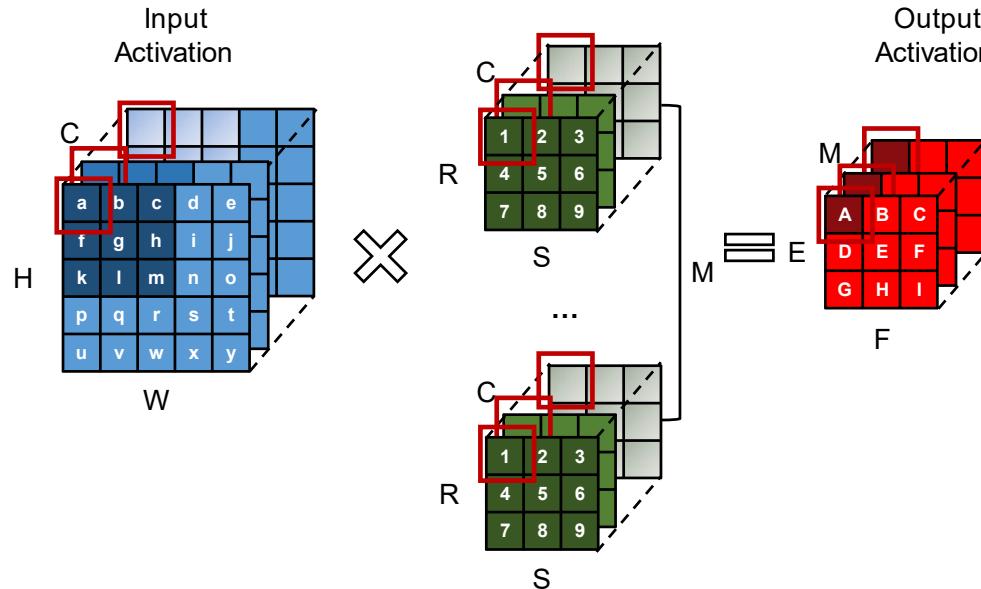


- Change temporal ordering

```

for (n=0; n<N; n++) {
    for (m=0; m<M; m++) {
        for (e=0; e<E; e++) {
            for (f=0; f<F; f++) {
                OA[n][m][e][f] = 0;
                for (r=0; r<R; r++) {
                    for (s=0; s<S; s++) {
                        for (c=0; c<C; c++) {
                            h = e * stride - pad + r;
                            w = f * stride - pad + s;
                            OA[n][m][e][f] +=
                                IA[n][c][h][w]
                                * W[m][c][r][s];
                        }
                    }
                }
            }
        }
    }
}
    
```

Weight Stationary Dataflow

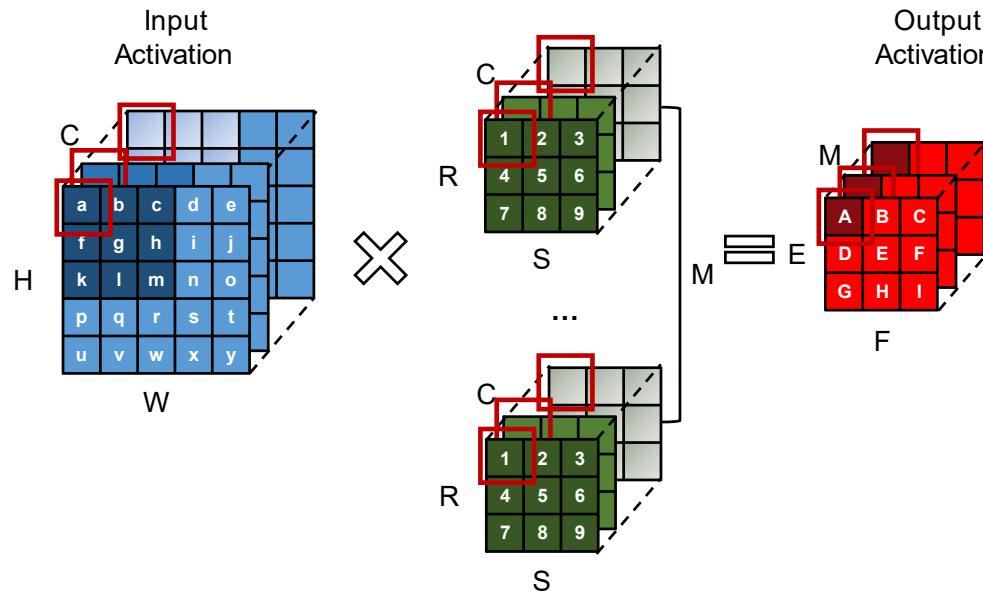


- Change temporal ordering

```

for (n=0; n<N; n++) {
    for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
                for (m=0; m<M; m++) {
                    float curr_w = W[r][s][c][m];
                    for (e=0; e<E; e++) {
                        for (f=0; f<F; f++) {
                            h = e * stride - pad + r;
                            w = f * stride - pad + s;
                            OA[n][m][e][f] +=
                                IA[n][c][h][w]
                                * curr_w;
                        }
                    }
                }
            }
        }
    }
}
    
```

Weight Stationary Dataflow



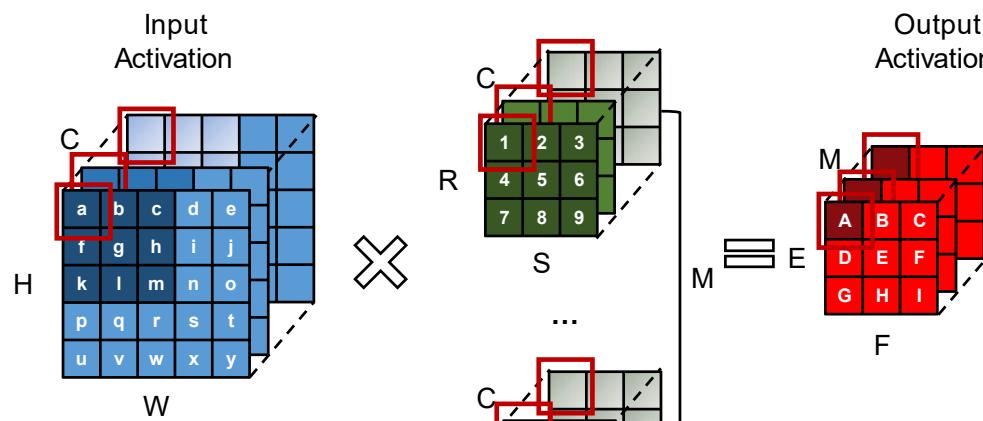
- Apply spatial parallelism

```

for (n=0; n<N; n++) {
    for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
            spatial_for (c=0; c<C; c++) {
                spatial_for (m=0; m<M; m++) {
                    float curr_w = W[r][s][c][m];
                    for (e=0; e<E; e++) {
                        for (f=0; f<F; f++) {
                            h = e * stride - pad + r;
                            w = f * stride - pad + s;
                            OA[n][m][e][f] +=
                                IA[n][c][h][w]
                                * curr_w;
                }
            }
        }
    }
}
    
```

Weight Stationary Dataflow

- Apply temporal tiling (**NVDLA Dataflow** (nvdla.org))



```

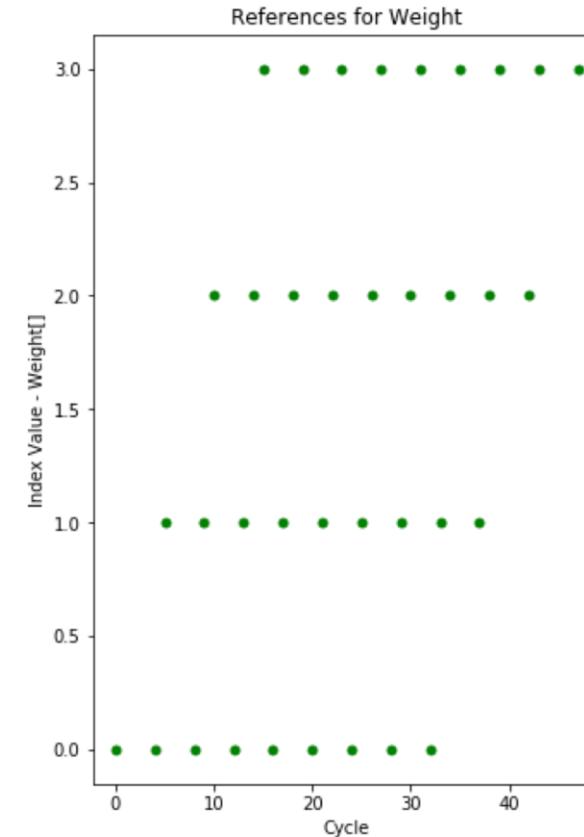
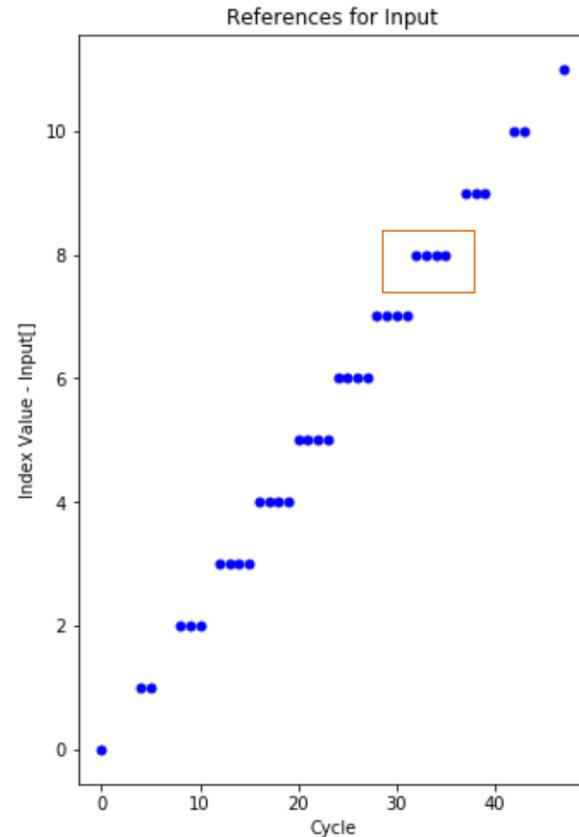
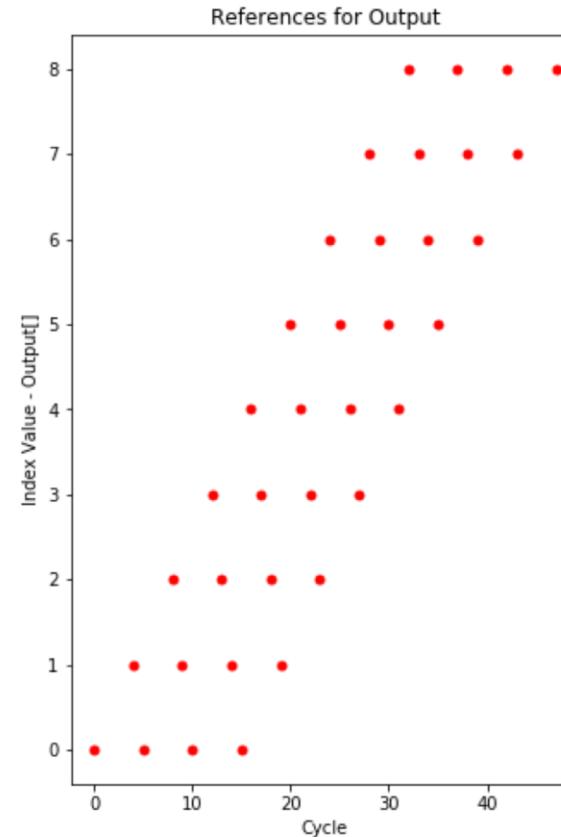
for (n=0; n<N; n++) {
    for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
            for (c_t=0; c_t<C/16; c_t++) {
                for (m_t=0; m_t<M/64; m_t++) {
                    spatial_for (c_s=0; c_s<16; c_s++) {
                        spatial_for (m_s=0; m_s<64; m_s++) {
                            int curr_c = c_t * 16 + c_s;
                            int curr_m = m_t * 64 + m_s;
                            float curr_w =
                                W[r][s][curr_c][curr_m];
                            for (e=0; e<E; e++) {
                                for (f=0; f<F; f++) {
                                    h = e * stride - pad + r;
                                    w = f * stride - pad + s;
                                    OA[n][curr_m][e][f] +=
                                        IA[n][curr_c][h][w]
                                            * curr_w;
                            }
                        }
                    }
                }
            }
        }
    }
}
    
```

Input Stationary – Reference Pattern

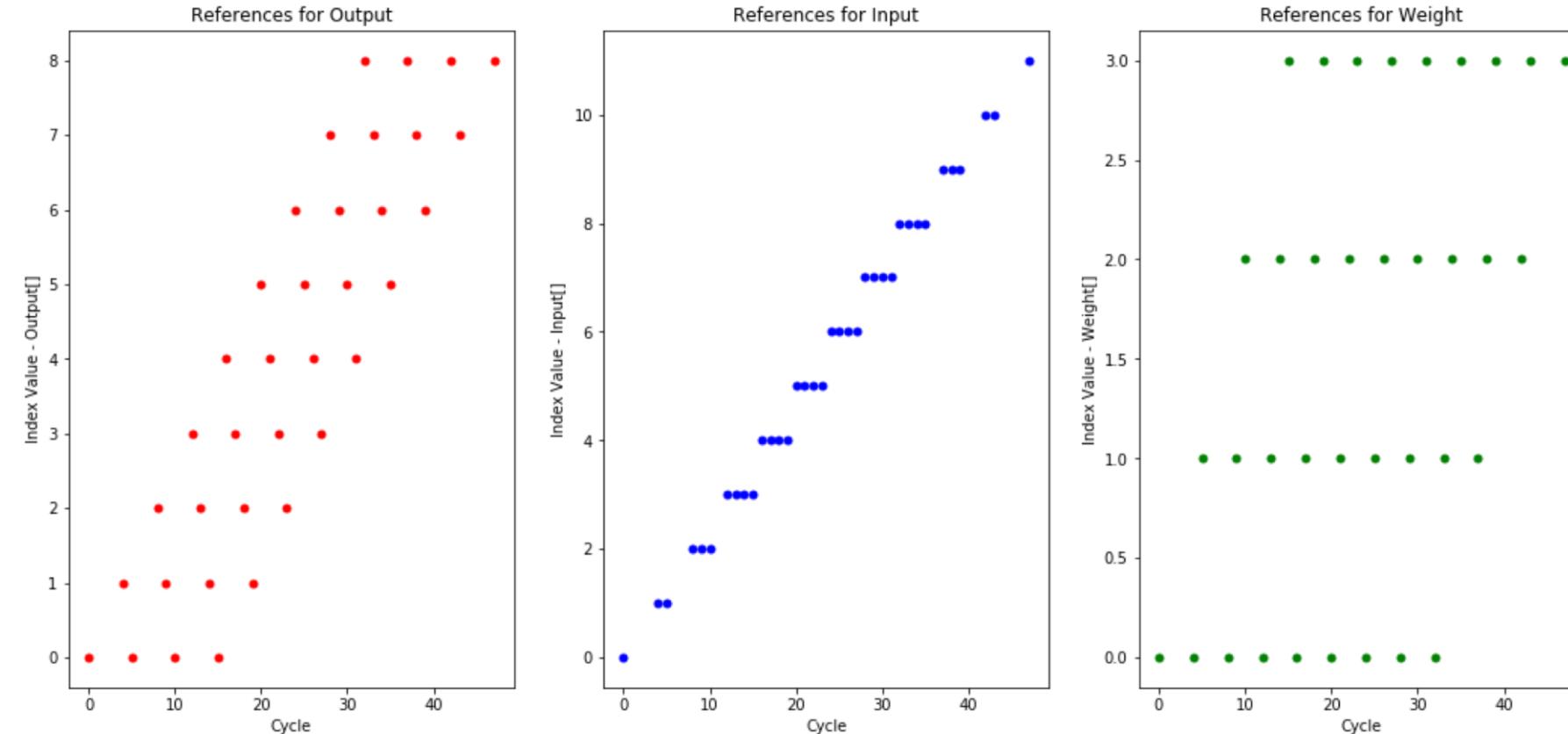
```

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

- Implement input stationary with no input index
- Beware $0 \leq w - r < E$

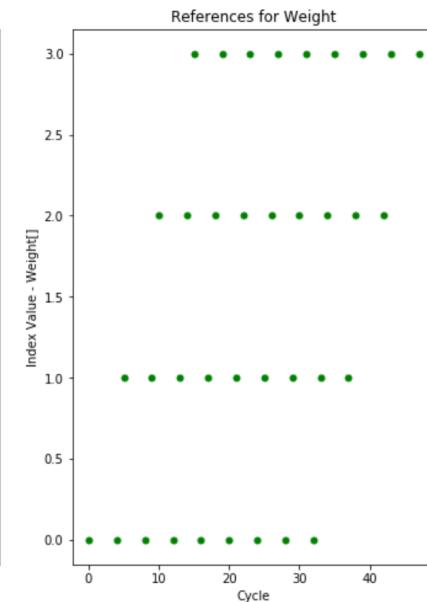
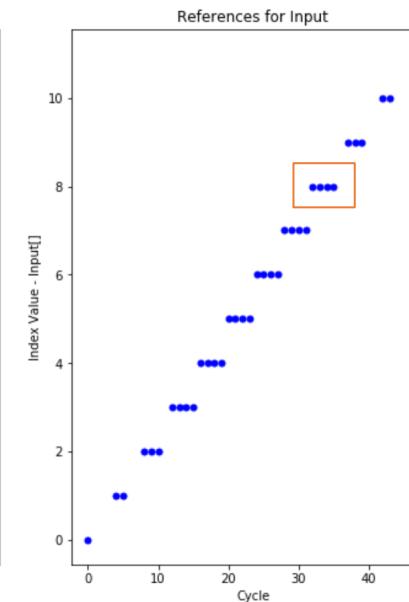
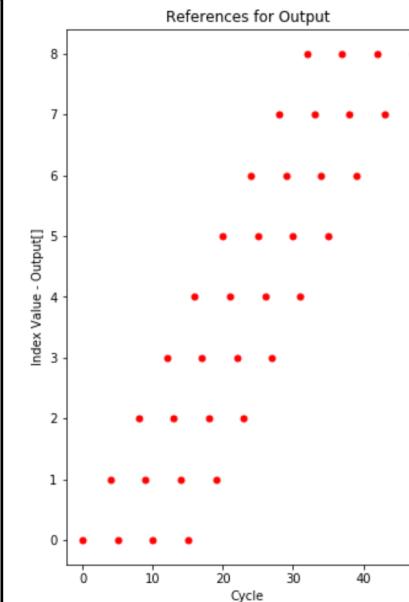
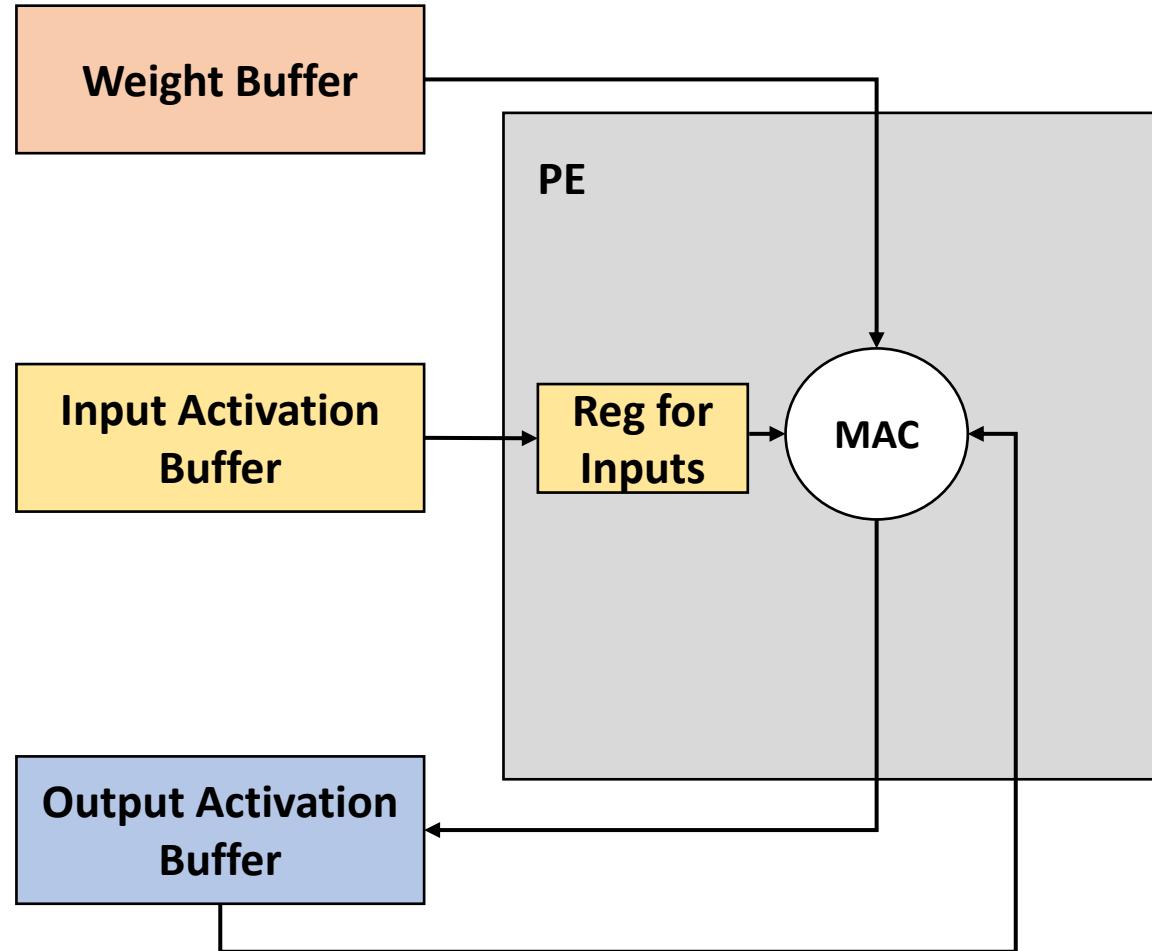


Input Stationary– Reference Pattern



- Inputs used repeatedly (R times)
- Weights reused in large window (size = R)
- Sliding window of outputs (size = R)

Buffer Access Pattern (IS)

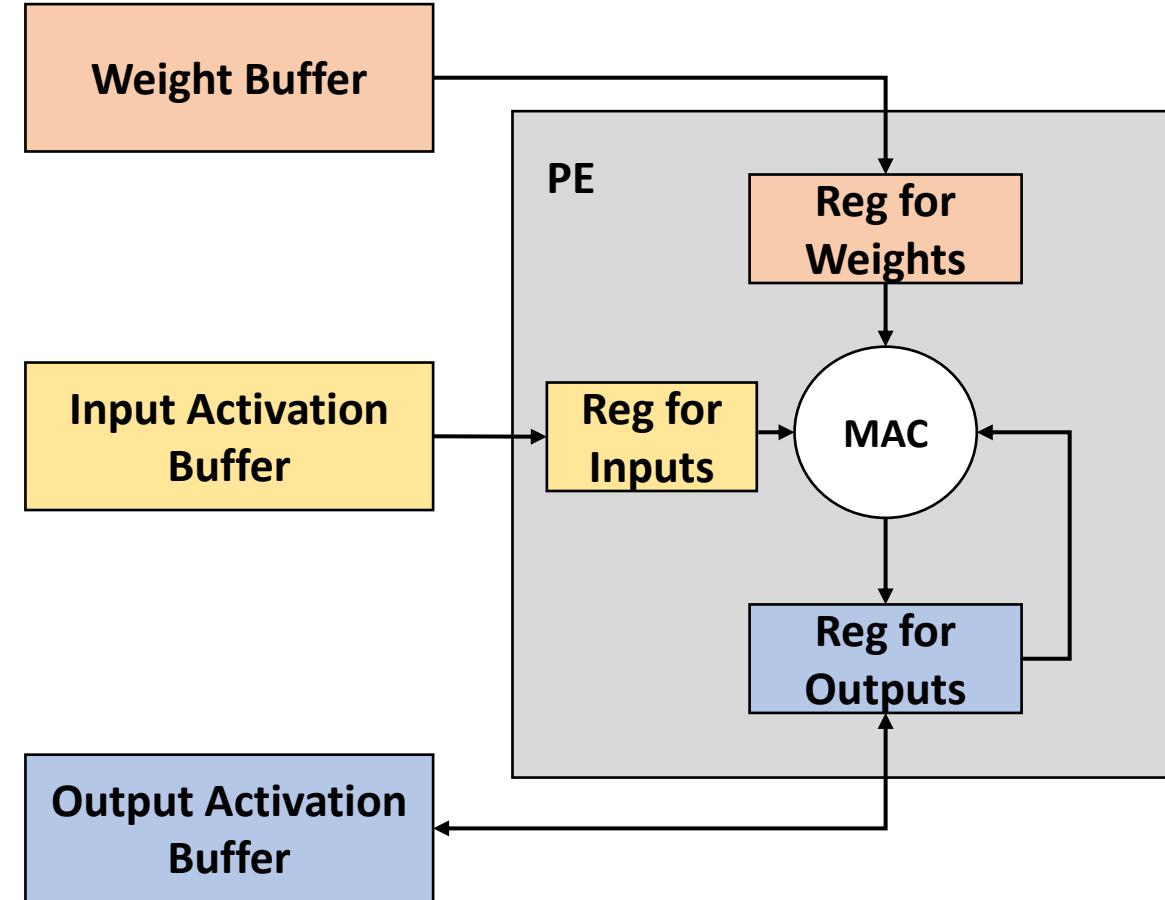


Minimum Costs

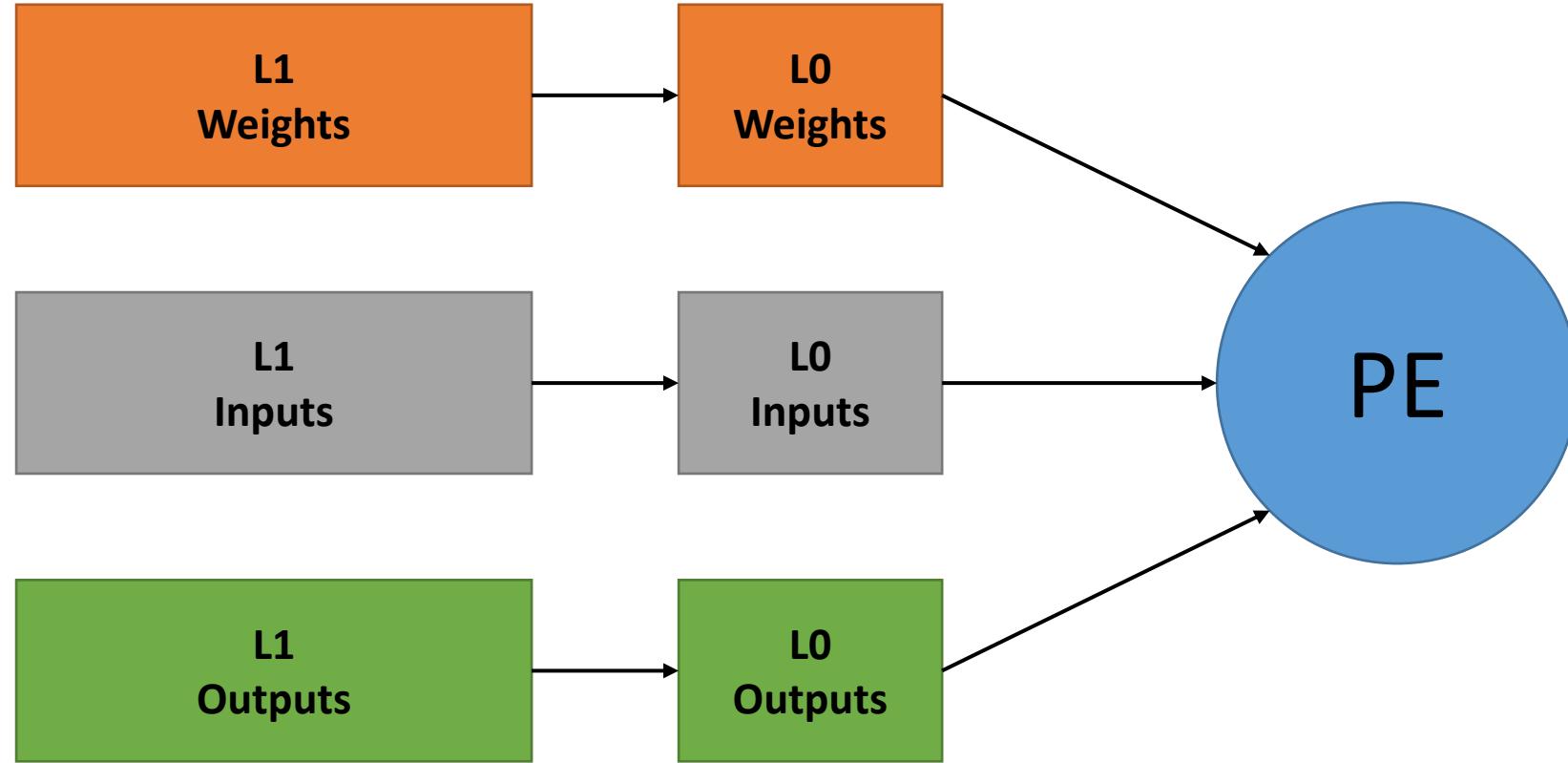
	OS	WS	IS	Min
MACs	E*R	E*R	E*R	E*R
Weight Reads	E*R	R	E*R	R
Input Reads	E*R	E*R	E	E
Output Reads	0	E*R	E*R	0
Output Writes	E	E*R	E*R	E

Assume: H ~ E

PE with Storages



Intermediate Buffering



1-D Convolution – Buffered



```
int I[H]; // Input activations
int W[R]; // Filter Weights
int O[E]; // Output activations
// Level 1
for (e1 = 0; e1 < E1; e1++)
    for (r1 = 0; r1 < R1; r1++)
        // Level 0
        for (e0 = 0; e0 < E0; e0++)
            for (r0 = 0; r0 < R0; r0++)
                O[e1*E0+e0] += I[e1*E0+e0 + r1*R0+r0] * W[r1*R0+r0];
```

Note E and R are factored, so:
 $E_0 * E_1 = E$
 $R_0 * R_1 = R$

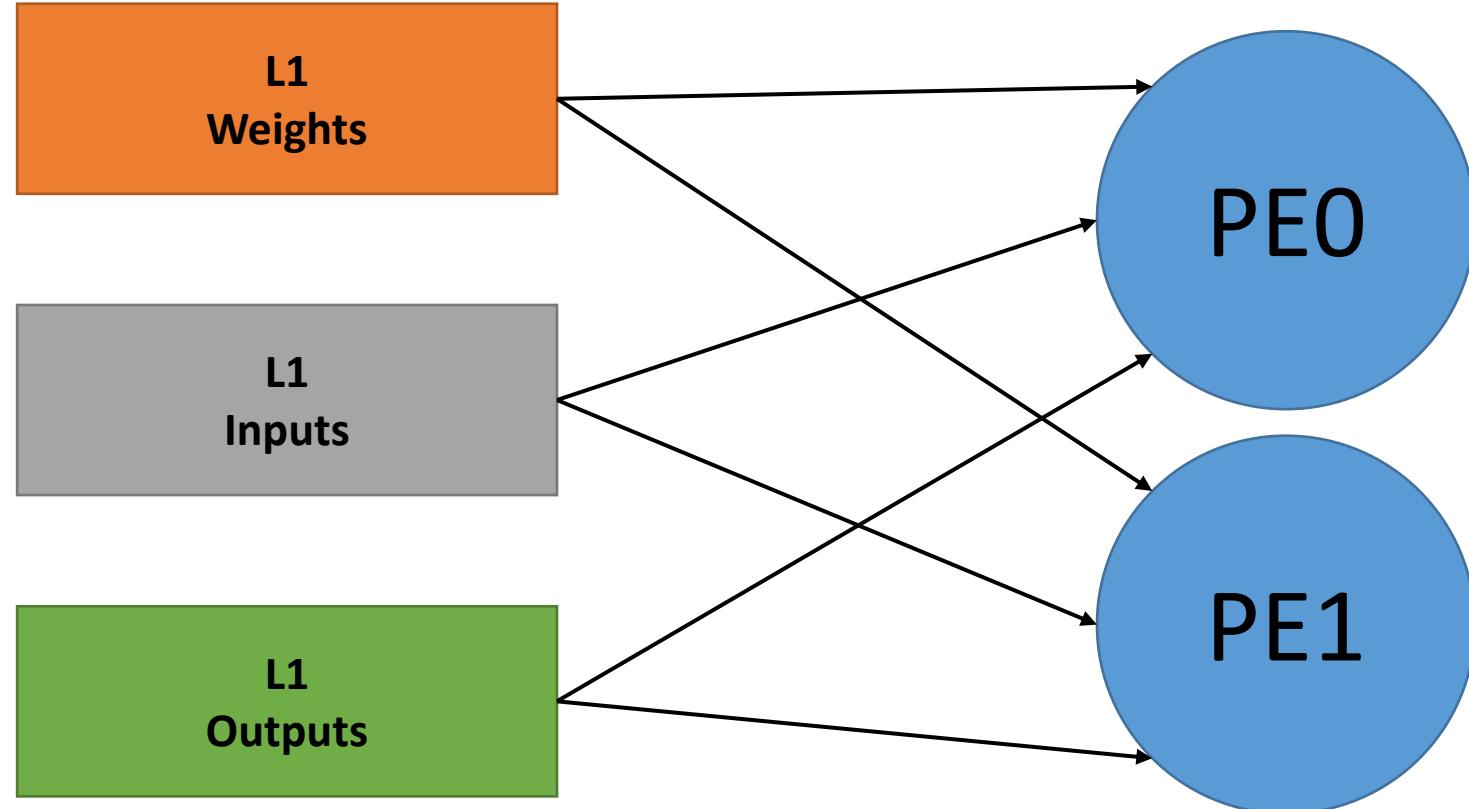
Buffer Sizes

- Level 0 buffer size is volume needed in each Level 1 iteration
- Level 1 buffer size is volume needed to be preserved and redelivered in future (usually successive) Level 1 iterations

```
int I[H]; // Input activations
int W[R]; // Filter Weights
int O[E]; // Output activations
// Level 1
for (e1 = 0; e1 < E1; e1++)
    for (r1 = 0; r1 < R1; r1++)
        // Level 0
        for (e0 = 0; e0 < E0; e0++)
            for (r0 = 0; r0 < R0; r0++)
                O[e1*E0+e0] += I[e1*E0+e0 + r1*R0+r0] * W[r1*R0+r0];
```

A legal assignment of loop limits will fit into the hardware's buffer sizes

Spatial PEs



1-D Convolution – Spatial



```
int I[W]; // Input activations
int W[R]; // Filter Weights
int O[E]; // Output activations
// Level 1
for (r1 = 0; r1 < R1; r1++)
    for (e1 = 0; e1 < E1; e1++)
        // Level 0
        spatial_for (r0 = 0; r0 < R0; r0++)
            spatial_for (e0 = 0; e0 < E0; e0++)
                O[e1*E0+e0] += I[e1*E0+e0 + r1*R0+r0] * W[r1*R0+r0];
```

Note:

- $E0 \times E1 = E$
- $R0 \times R1 = R$
- $R0 \times E0 \leq \#PEs$

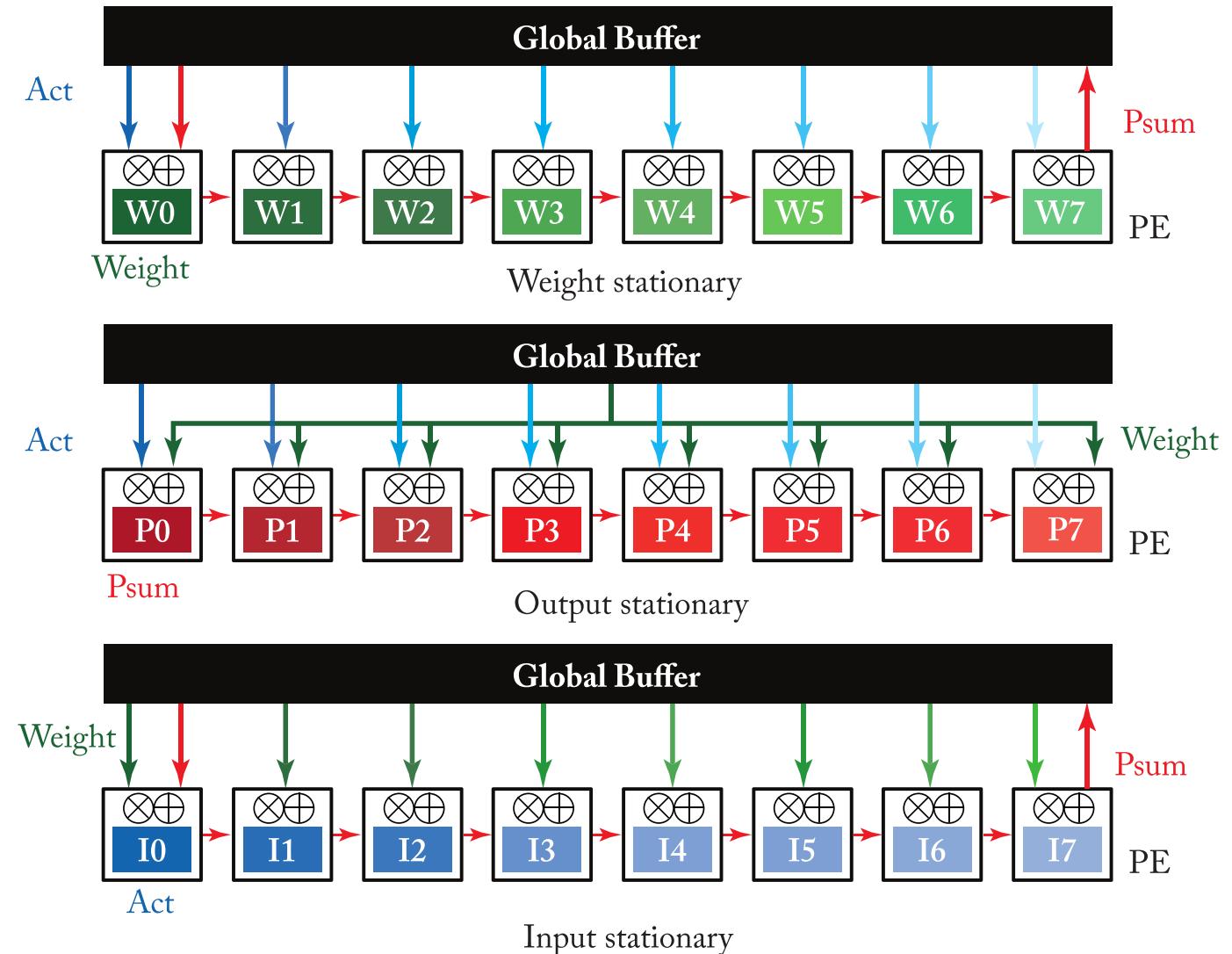
Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
 - Locality and Data Reuse
 - Dataflow Taxonomy
 - Data Orchestration
 - Network-on-Chip
 - Optimization
- Roofline Model
- Energy

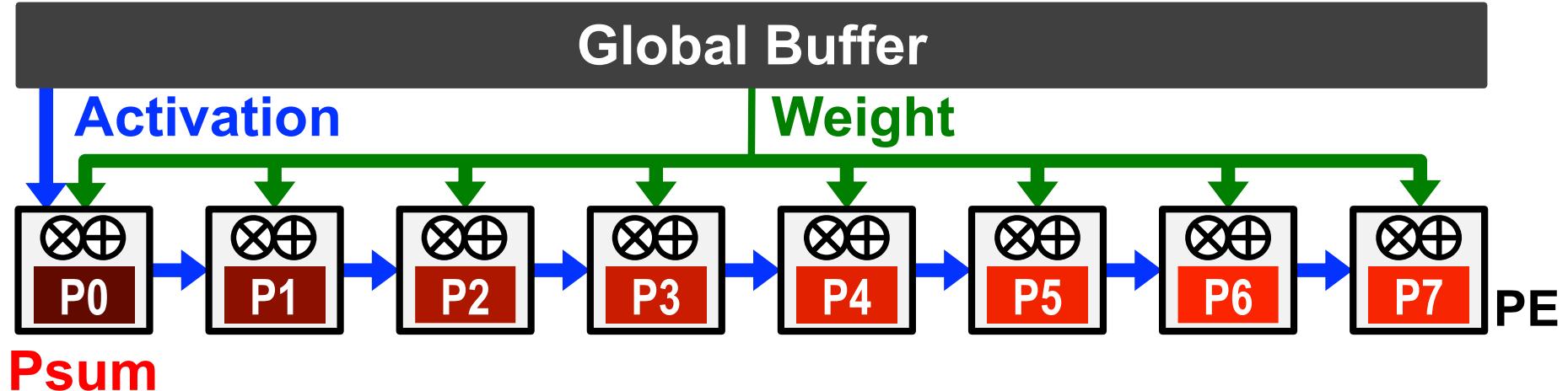
Dataflow Taxonomy

- Minimizing **data movement** is the key to achieving high **energy efficiency** for DNN accelerators
- Dataflow taxonomy:
 - **Output Stationary**: minimize movement of psums (**partial sums**)
 - **Weight Stationary**: minimize movement of weights
 - **Input Stationary**: minimize movement of inputs
- **Loop nest**
 - provides a compact way to describe various properties of a dataflow, e.g., data tiling in multi-level storage and spatial processing.

Dataflow Taxonomy



Output Stationary (OS)

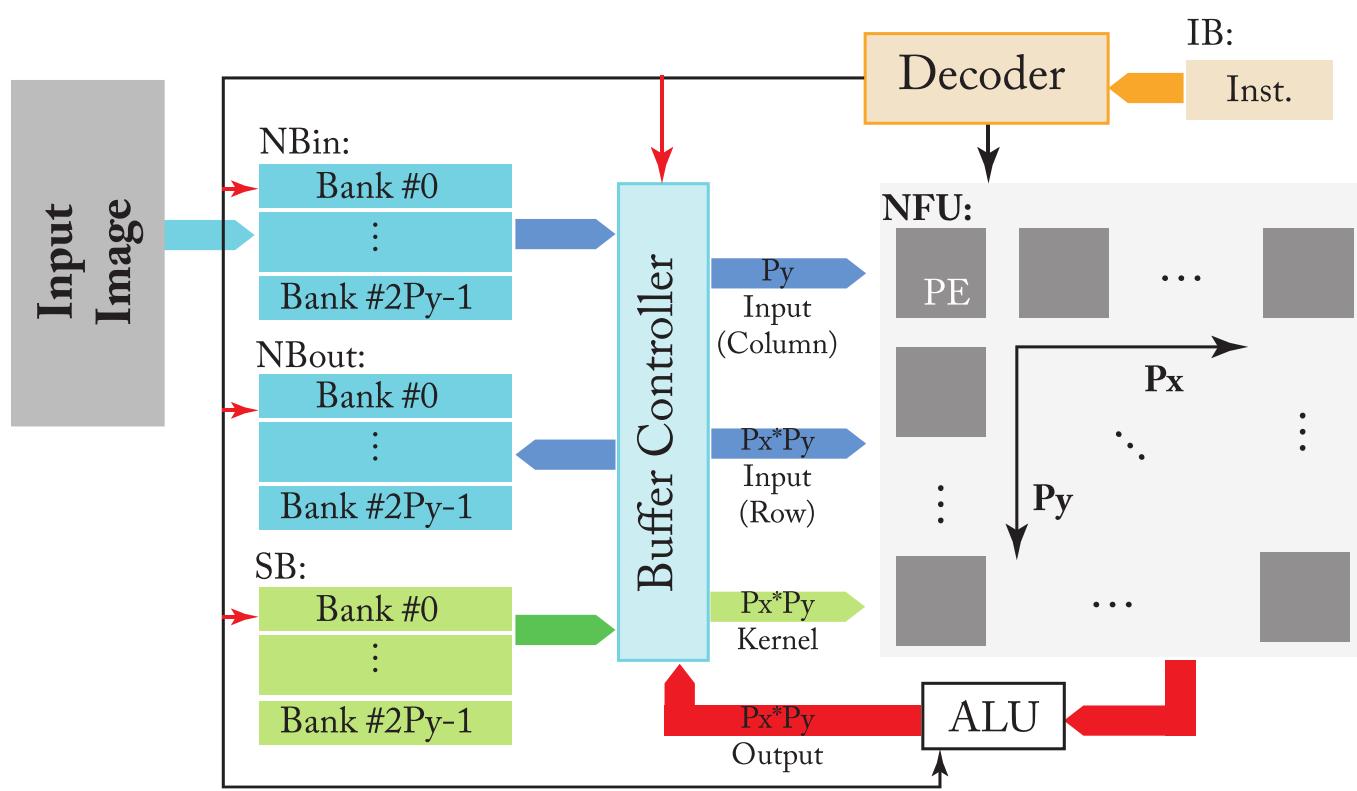


- Minimize partial sum R/W energy consumption
 - Maximize local accumulation
 - Broadcast/Multicast filter weights and reuse
 - Activations spatially across the PE array

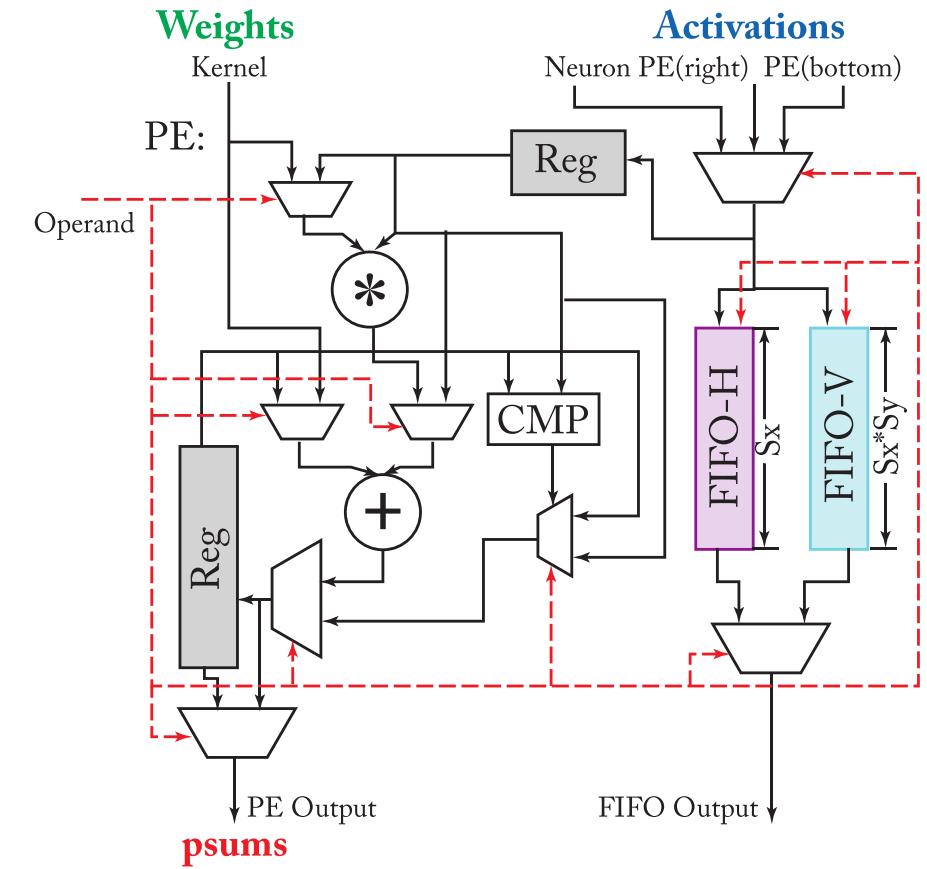
OS Example: ShiDianNao



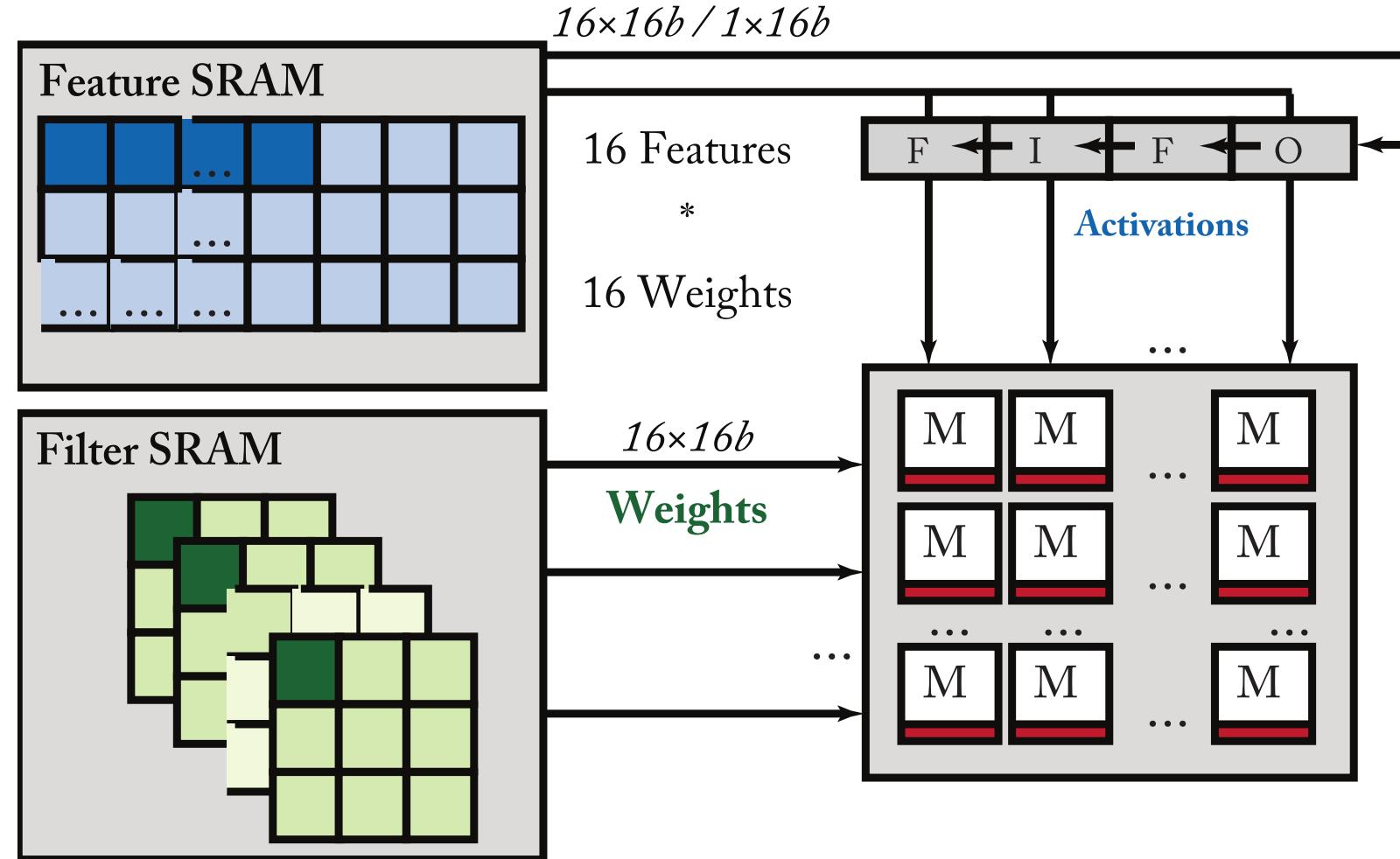
Top-Level Architecture



PE Architecture

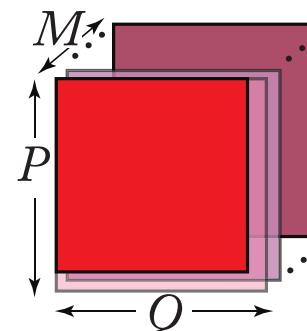
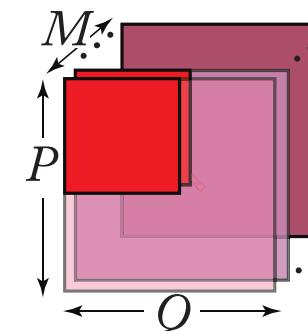
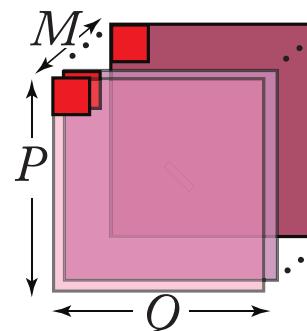


OS Example: ENVISION



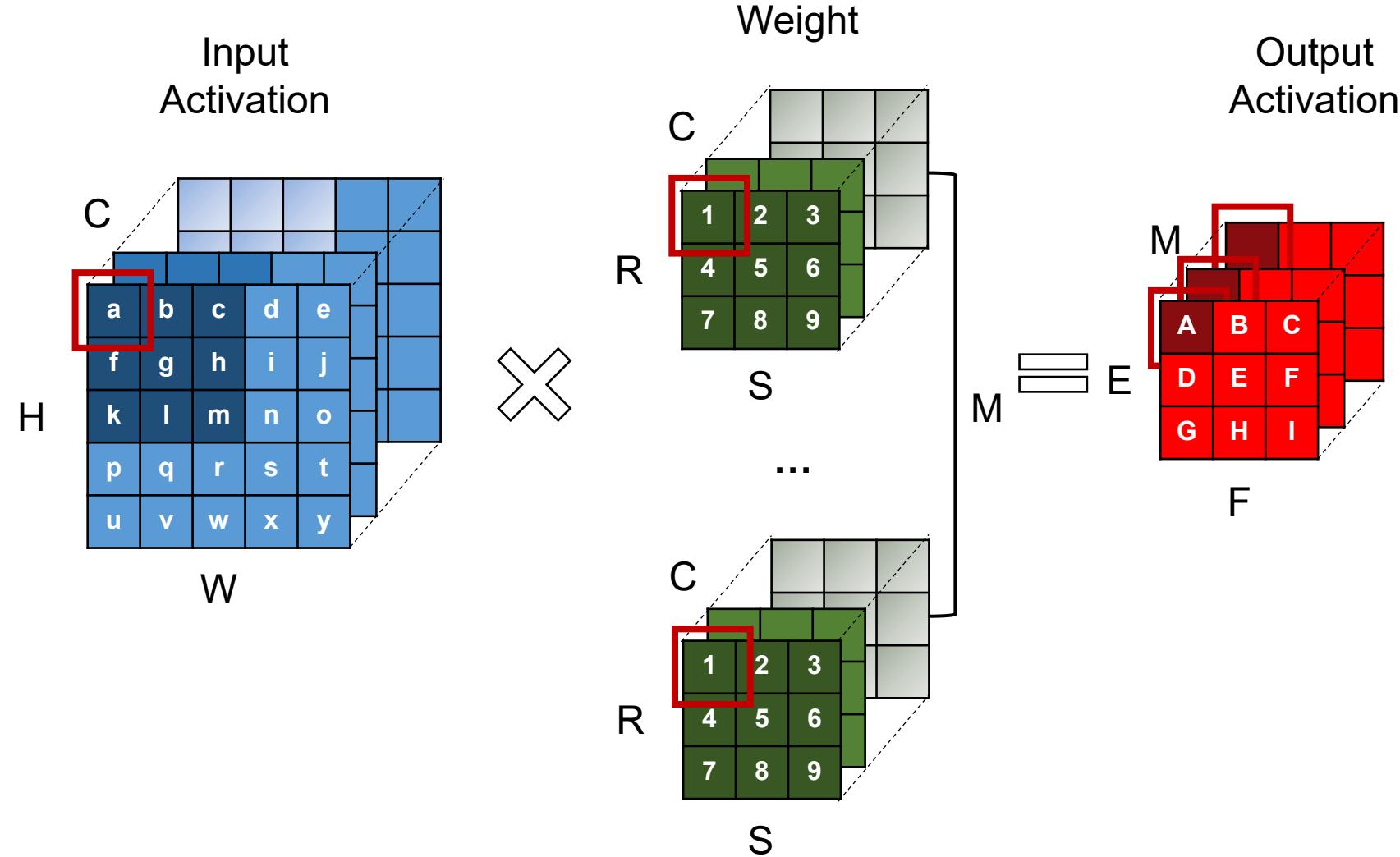
Variations of Output Stationary



	OS _A	OS _B	OS _C
Parallel Output Region	 A diagram showing a red square of size $P \times Q$ centered within a larger purple square of size $M \times N$. The red square is surrounded by a thin pink border.	 A diagram showing a red square of size $P \times Q$ centered within a larger purple square of size $M \times N$. The red square is surrounded by a thick pink border, and a small red dot is located at the bottom right corner of the red square.	 A diagram showing a red square of size $P \times Q$ centered within a larger purple square of size $M \times N$. The red square is surrounded by a thick pink border, and a diagonal line is drawn from the top-left to the bottom-right corner of the red square.
# Output Channels	Single	Multiple	Multiple
# Output Activations	Multiple	Multiple	Single
Notes	Targeting CONV Layers		Targeting FC Layers

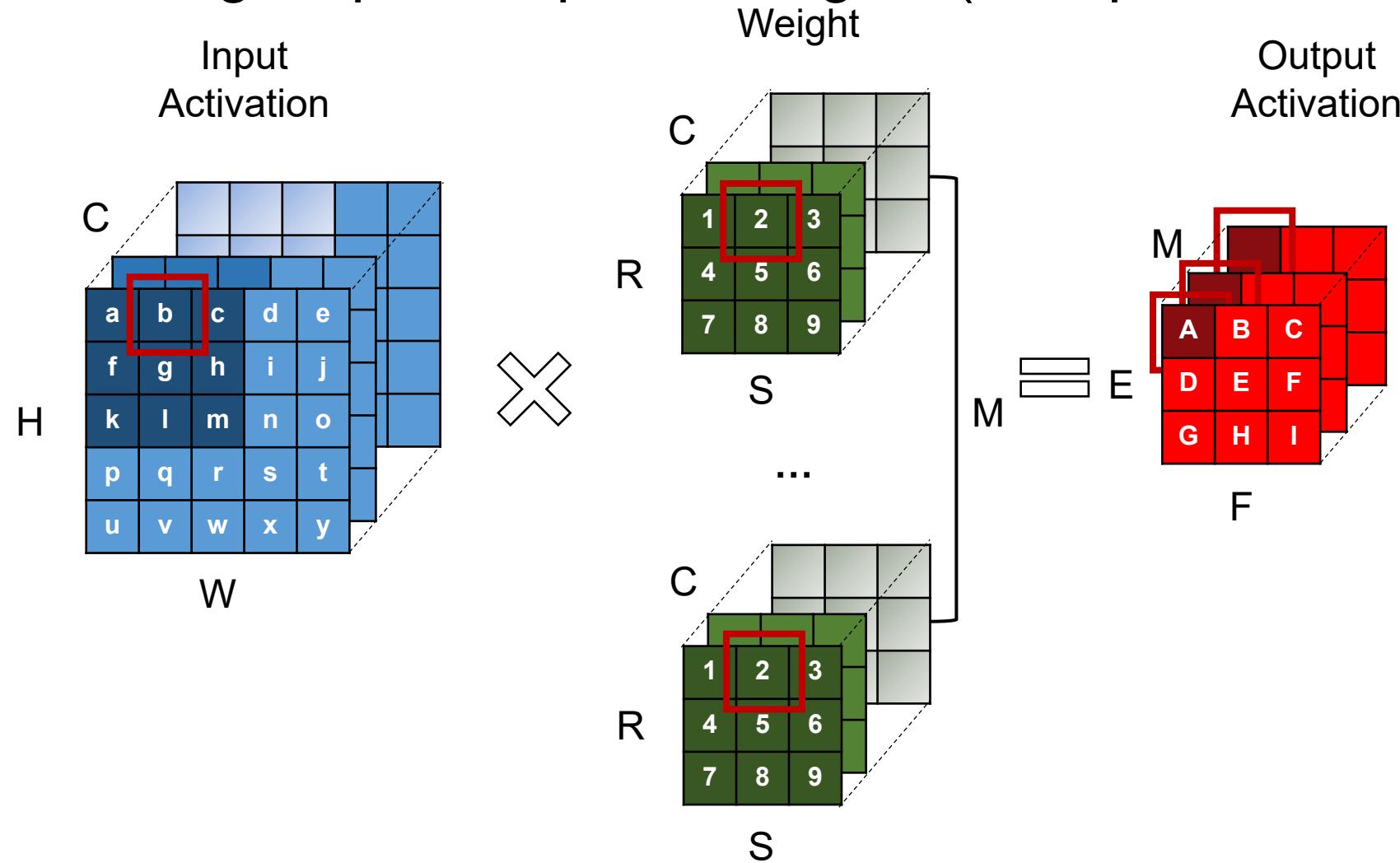
Output Stationary Dataflow

- Cycle through input fmap and weights (hold psum of output fmap)



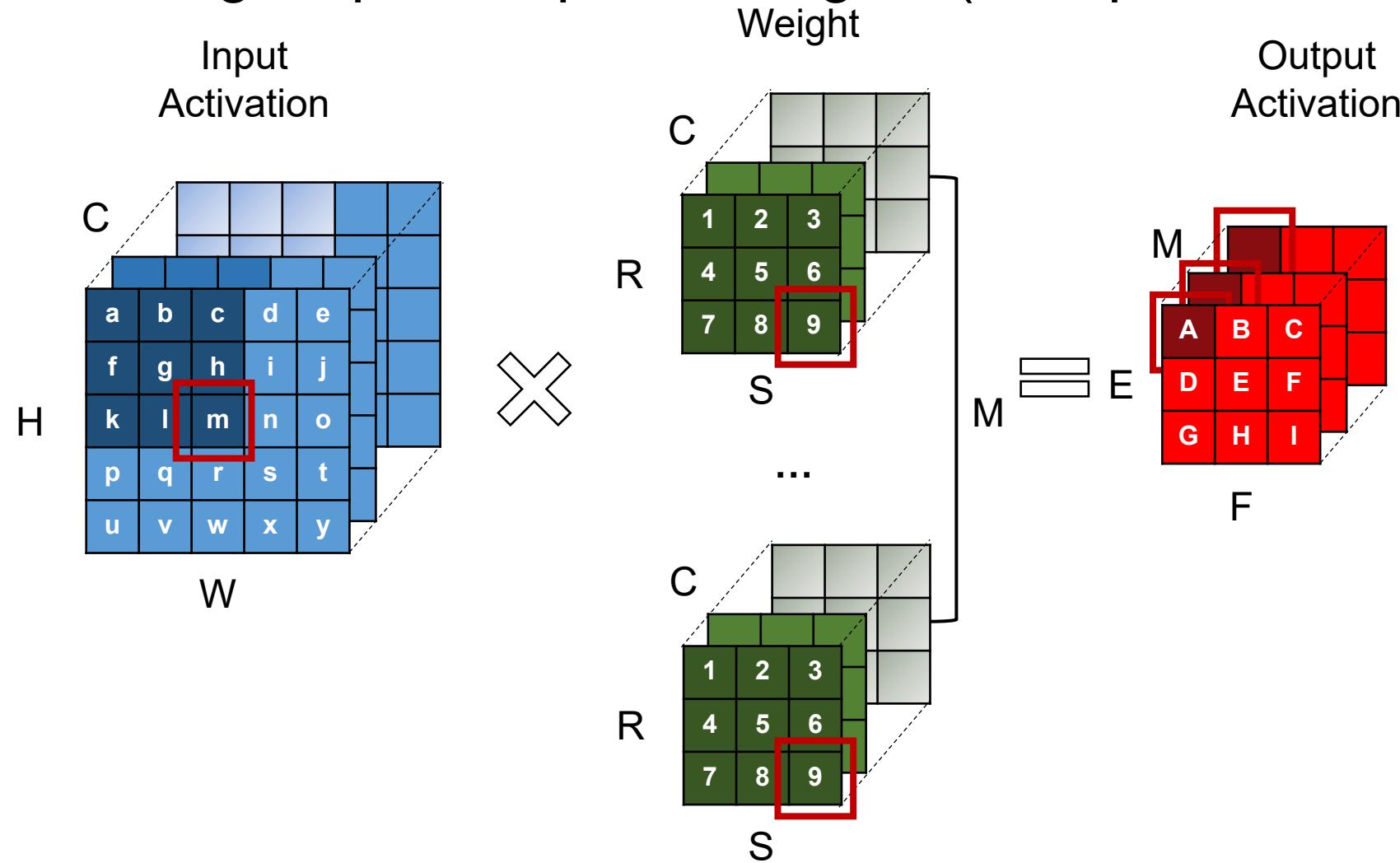
Output Stationary Dataflow

- Cycle through input fmap and weights (hold psum of output fmap)



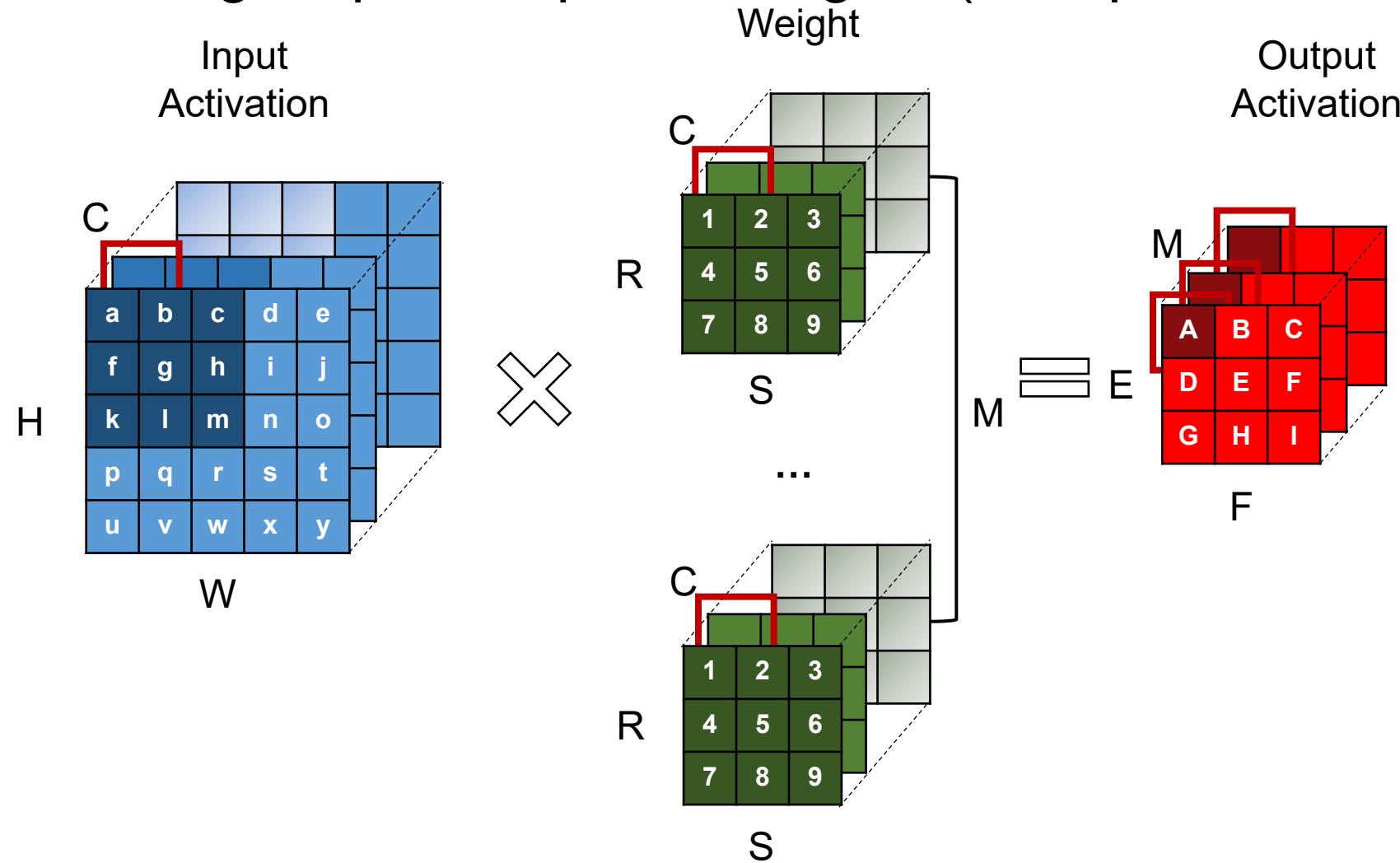
Output Stationary Dataflow

- Cycle through input fmap and weights (hold psum of output fmap)

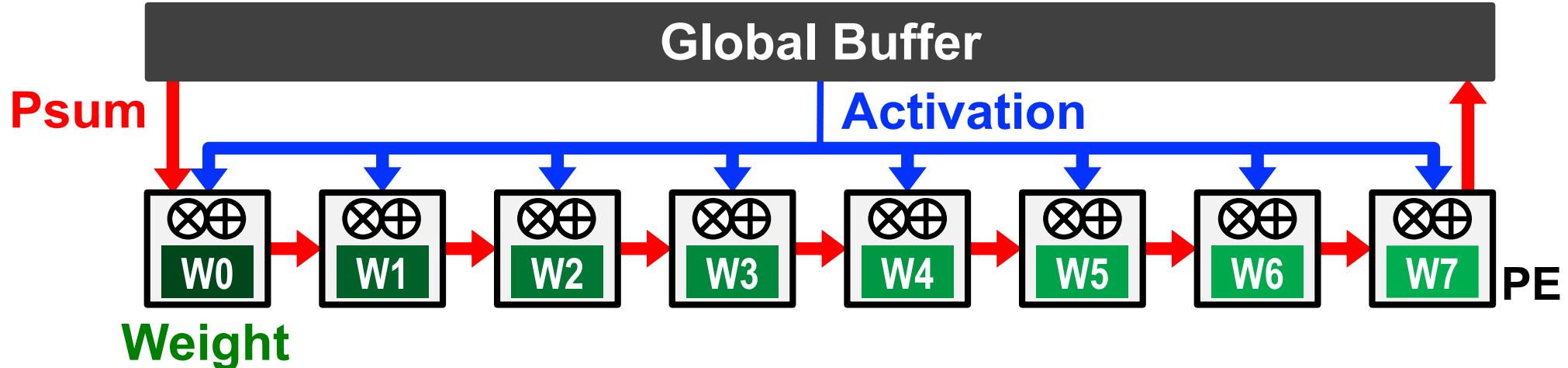


Output Stationary Dataflow

- Cycle through input fmap and weights (hold psum of output fmap)

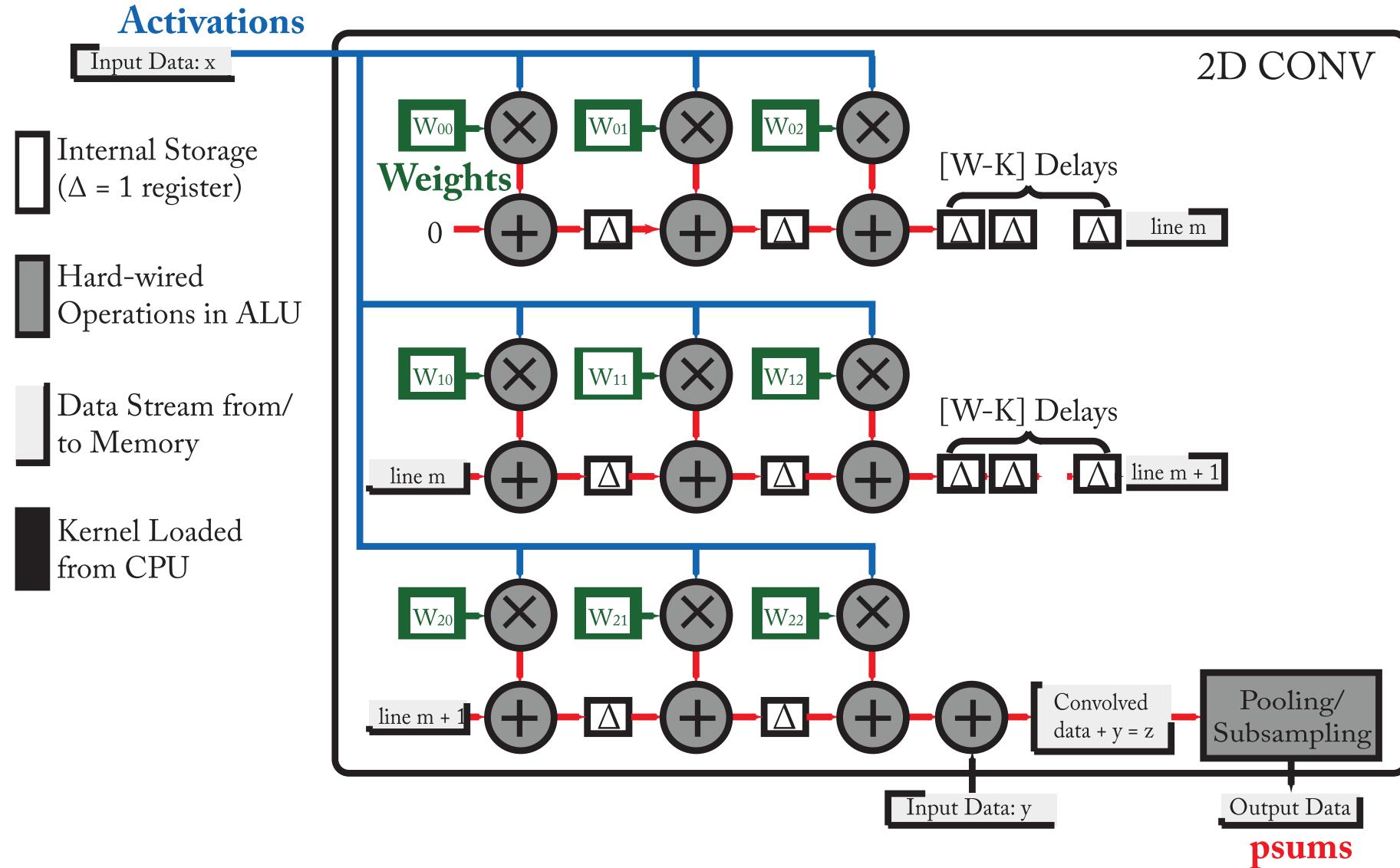


Weight Stationary (WS)

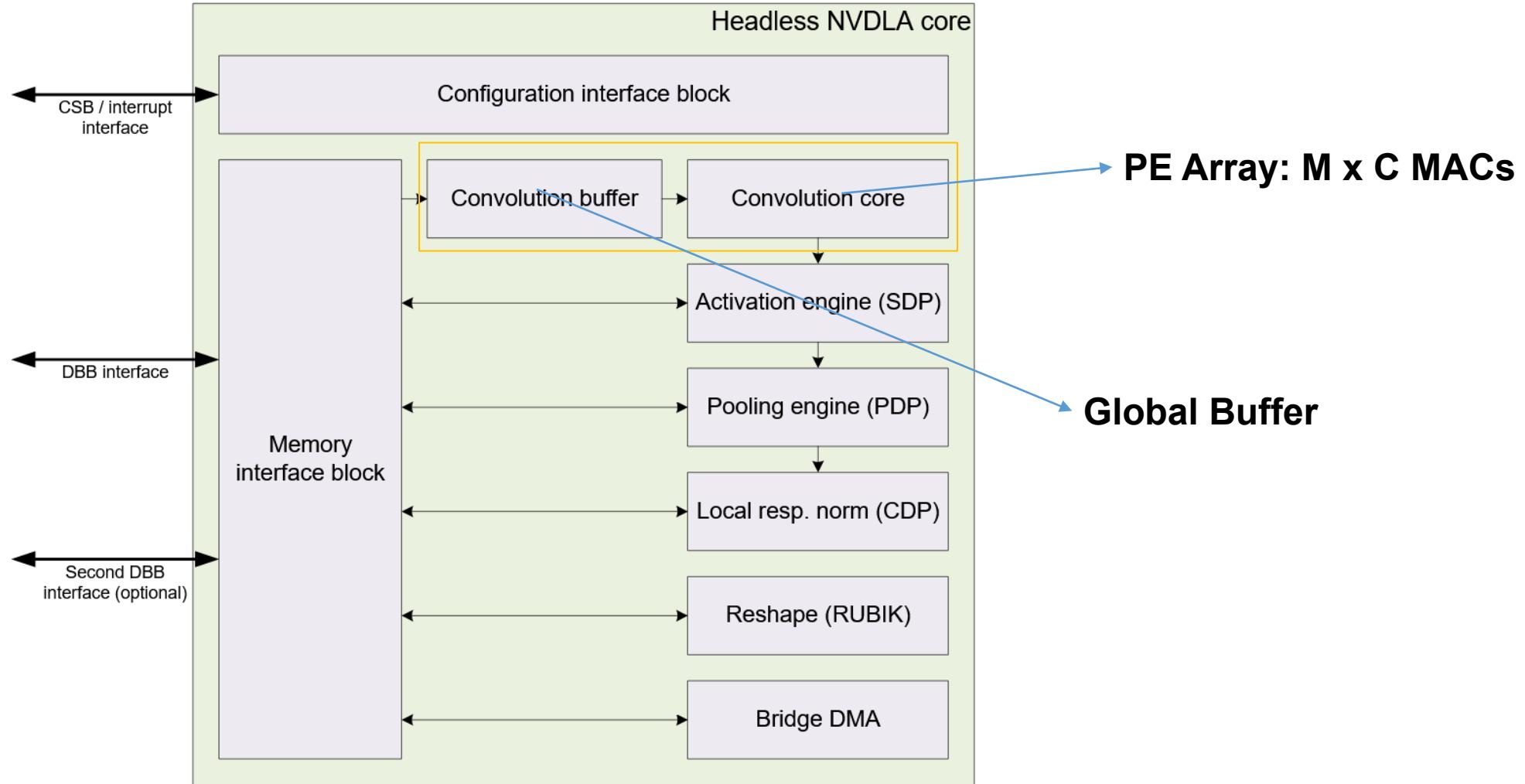


- **Minimize weight read energy consumption**
 - Maximize convolutional and filter reuse of weights
- Broadcast activations and accumulate psums
 - Spatially across the PE array.

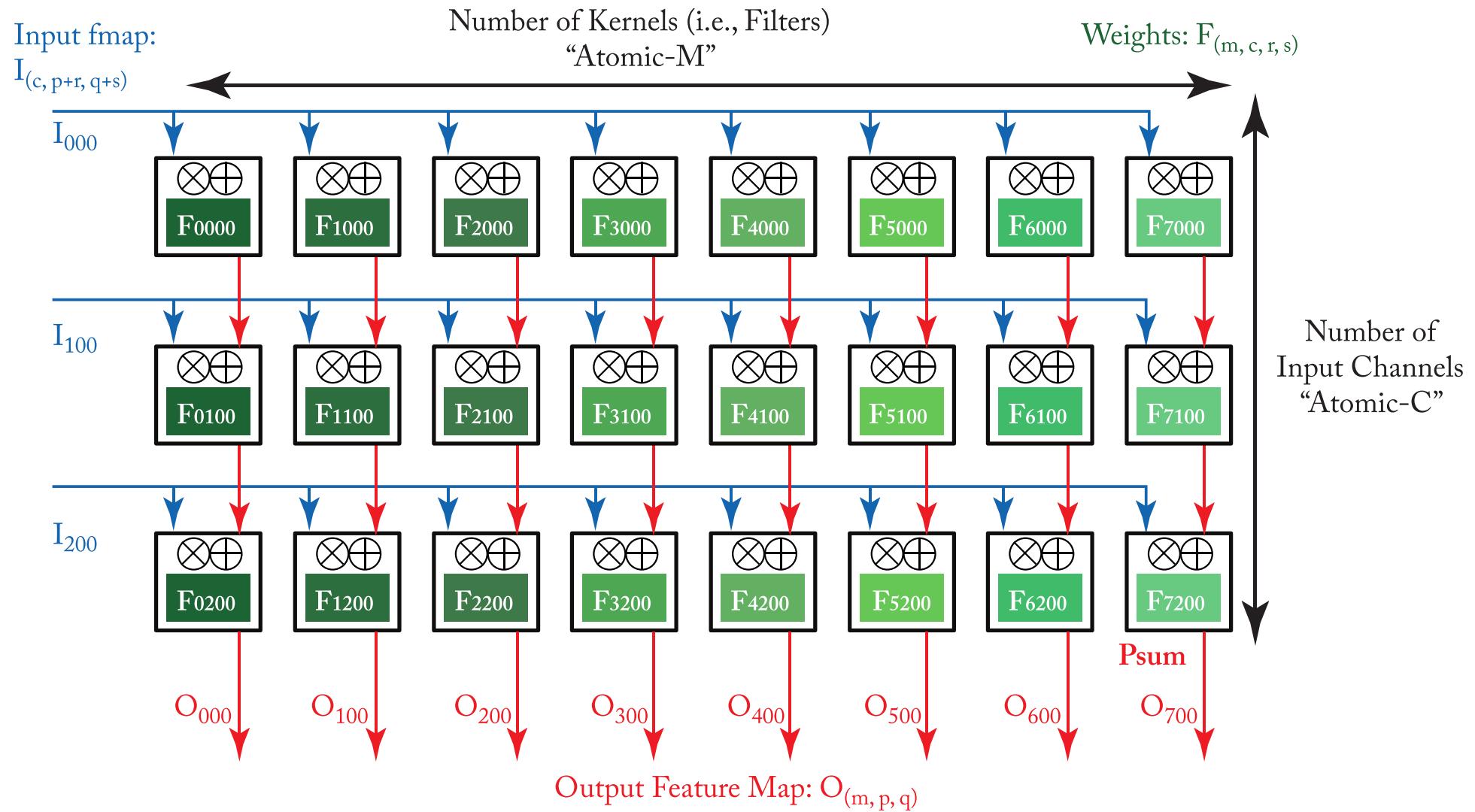
WS Example: nn-X (NeuFlow)



WS Example: NVDLA (Simplified)



WS Example: NVDLA (Simplified)



Loop Nest For Simplified NVDLA

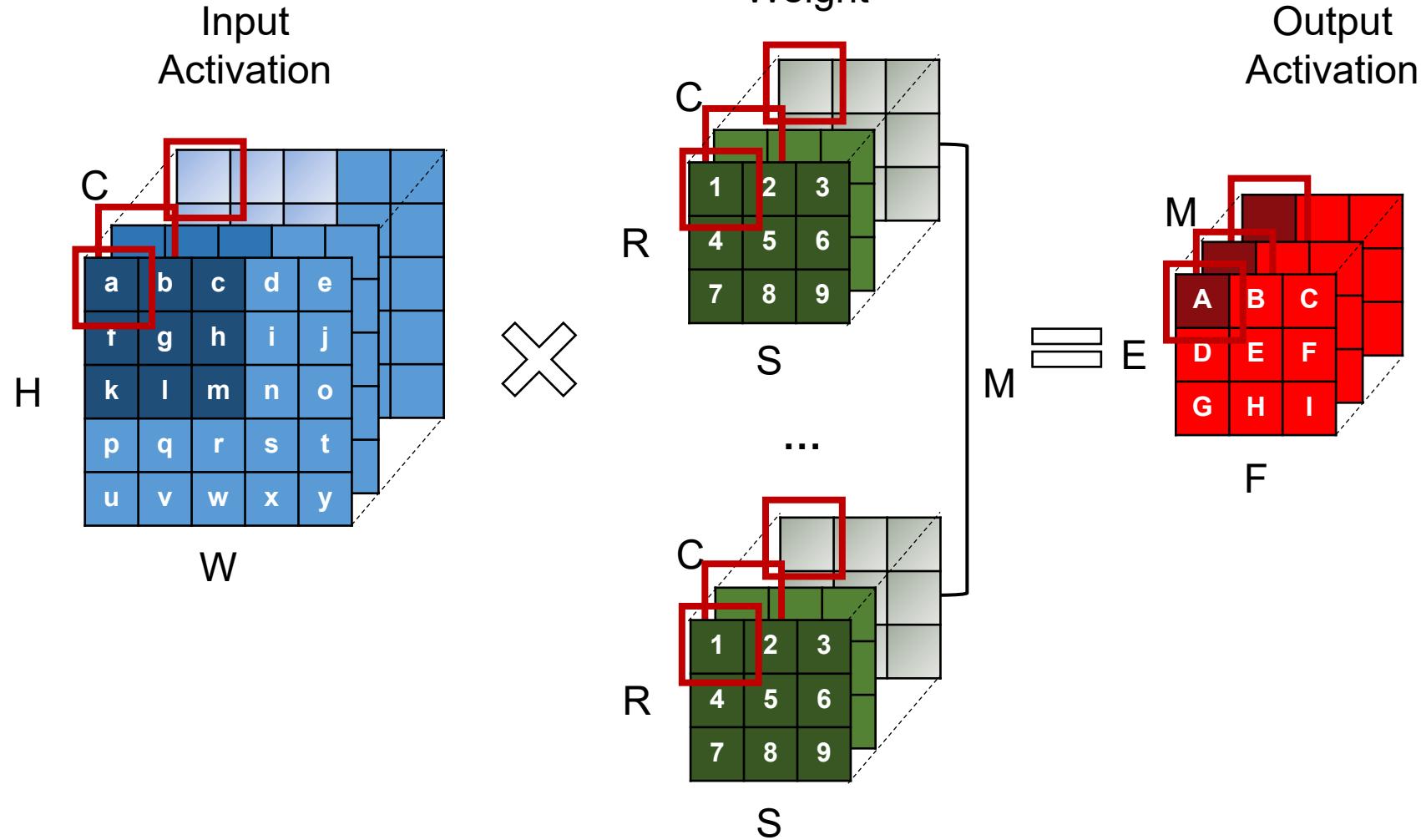


```
# i[C, H, W];           - Input activations
# f[M, C, R, S];       - Filter weights
# o[M, P, Q];          - Output activations

spatial_for m in range(M):
    spatial_for c in range(C):
        for r in range(R):
            for s in range(S):
                for p in range(P):
                    for q in range(Q):
                        o[m, p, q] += i[c, p+r, q+s] * f[m, c, r, s]
```

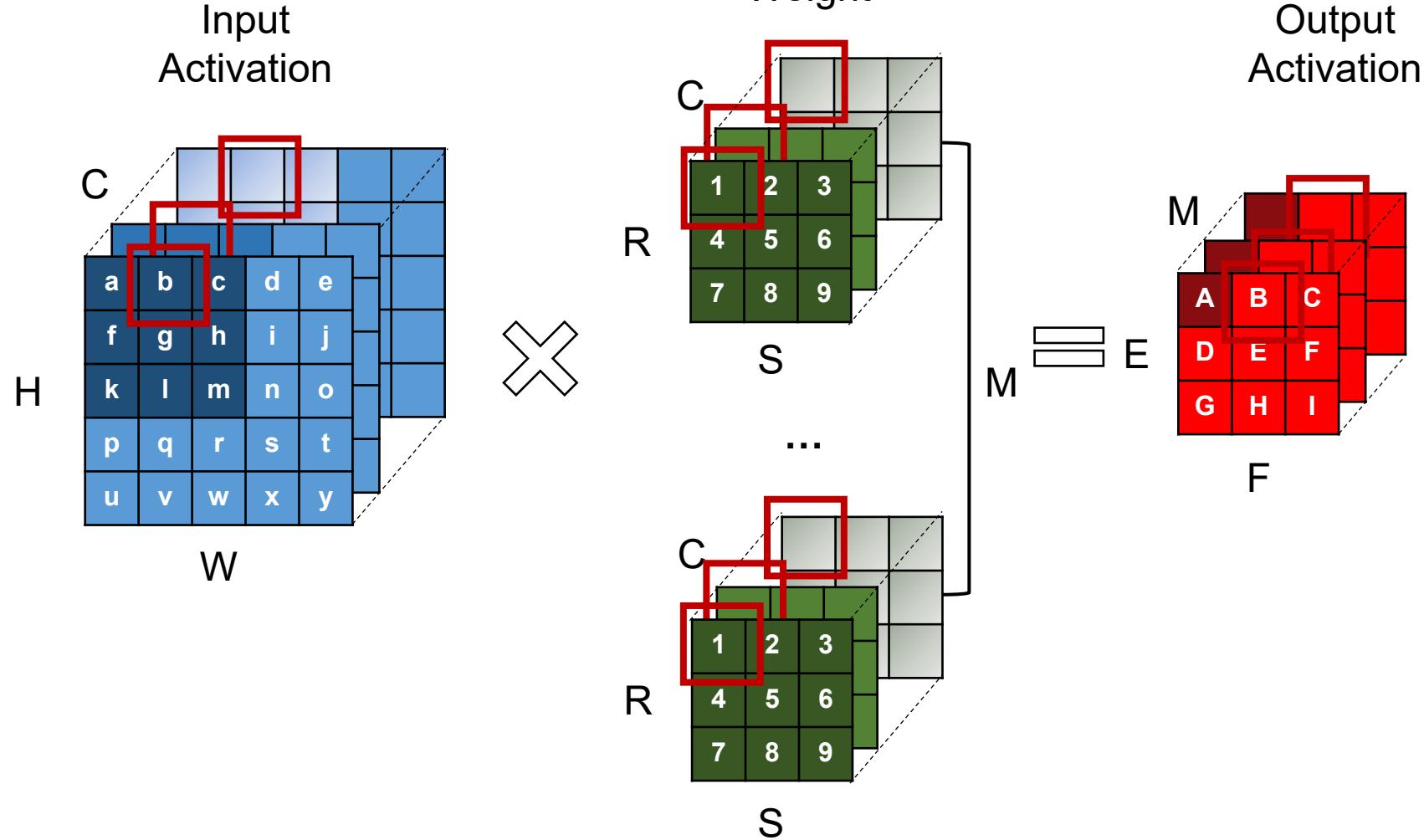
Weight Stationary Dataflow

- Cycle through input and output fmap (hold weights)



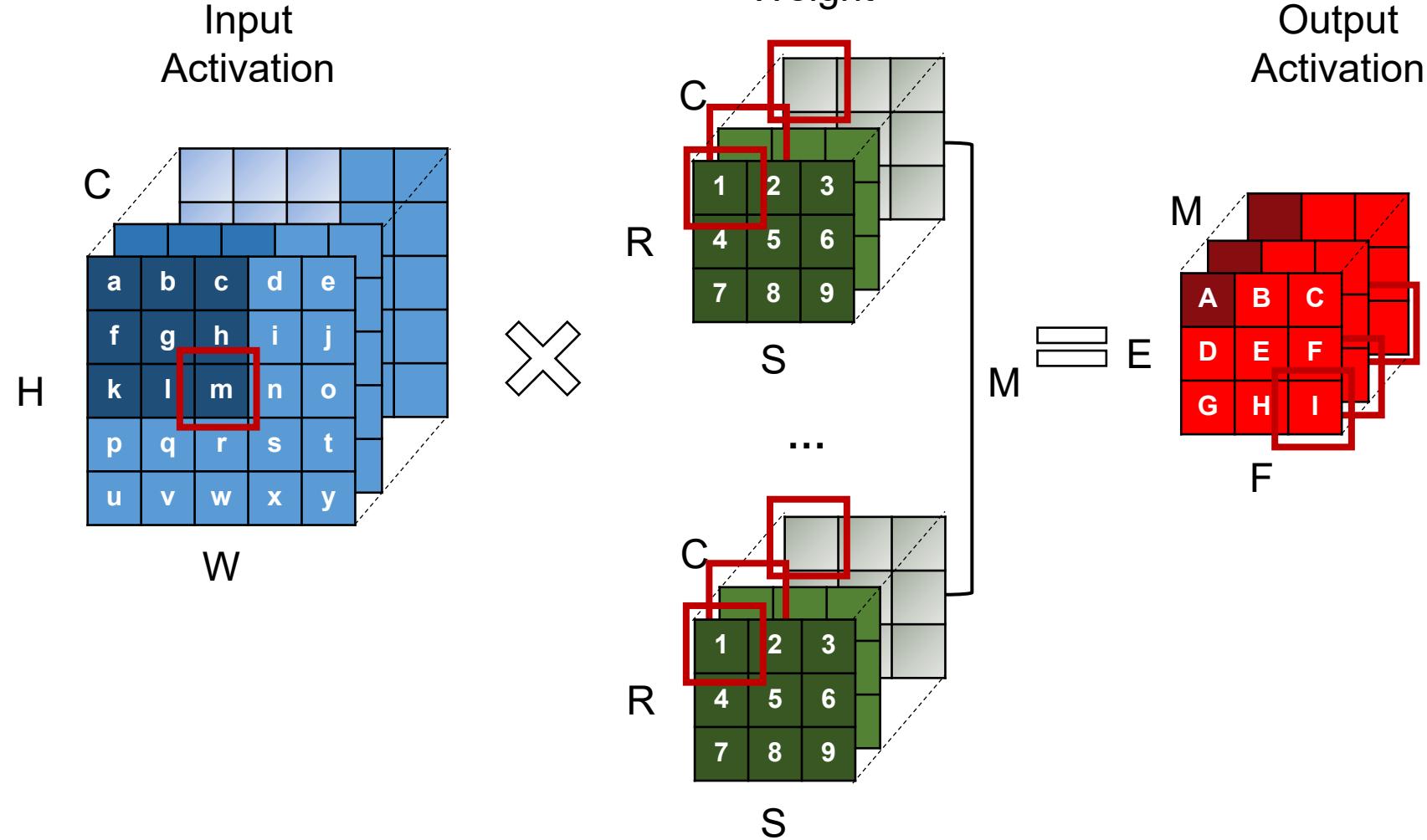
Weight Stationary Dataflow

- Cycle through input and output fmap (hold weights)



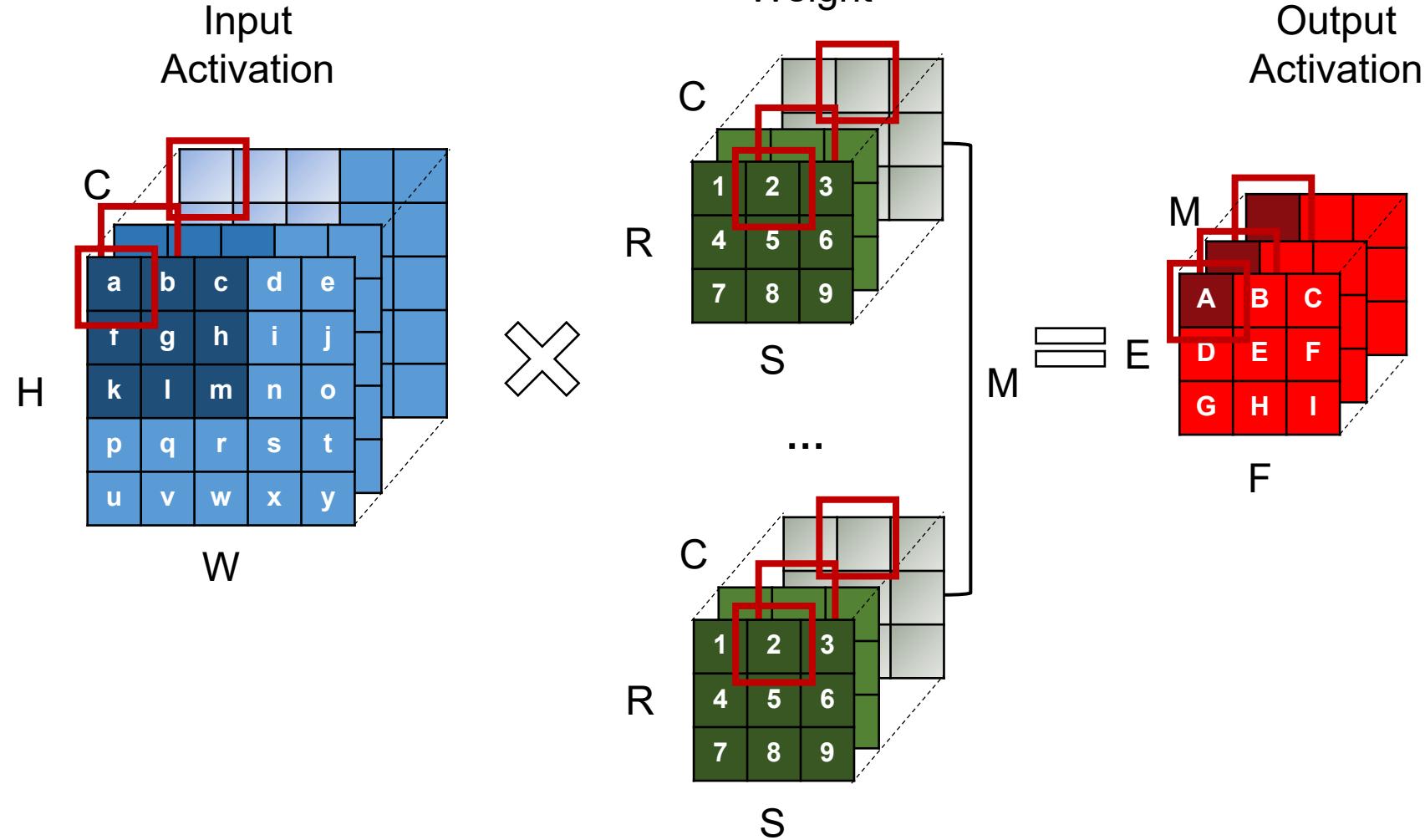
Weight Stationary Dataflow

- Cycle through input and output fmap (hold weights)



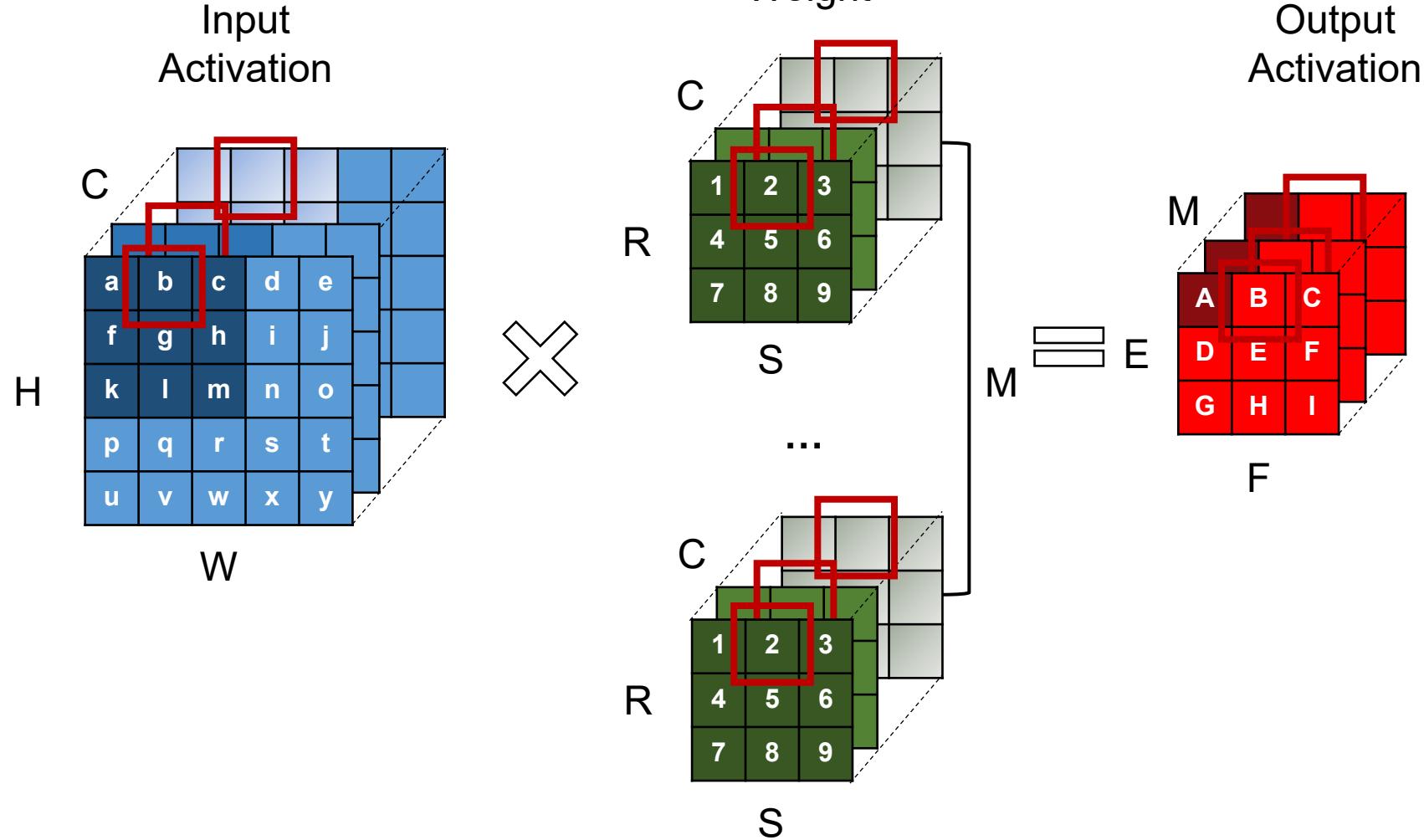
Weight Stationary Dataflow

- Load new weights

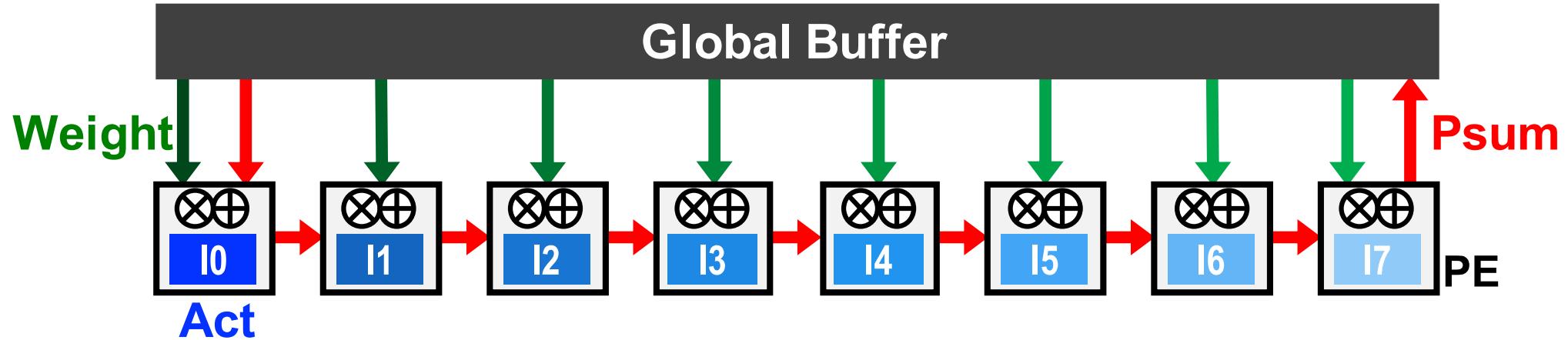


Weight Stationary Dataflow

- Cycle through input and output fmap (hold weights)

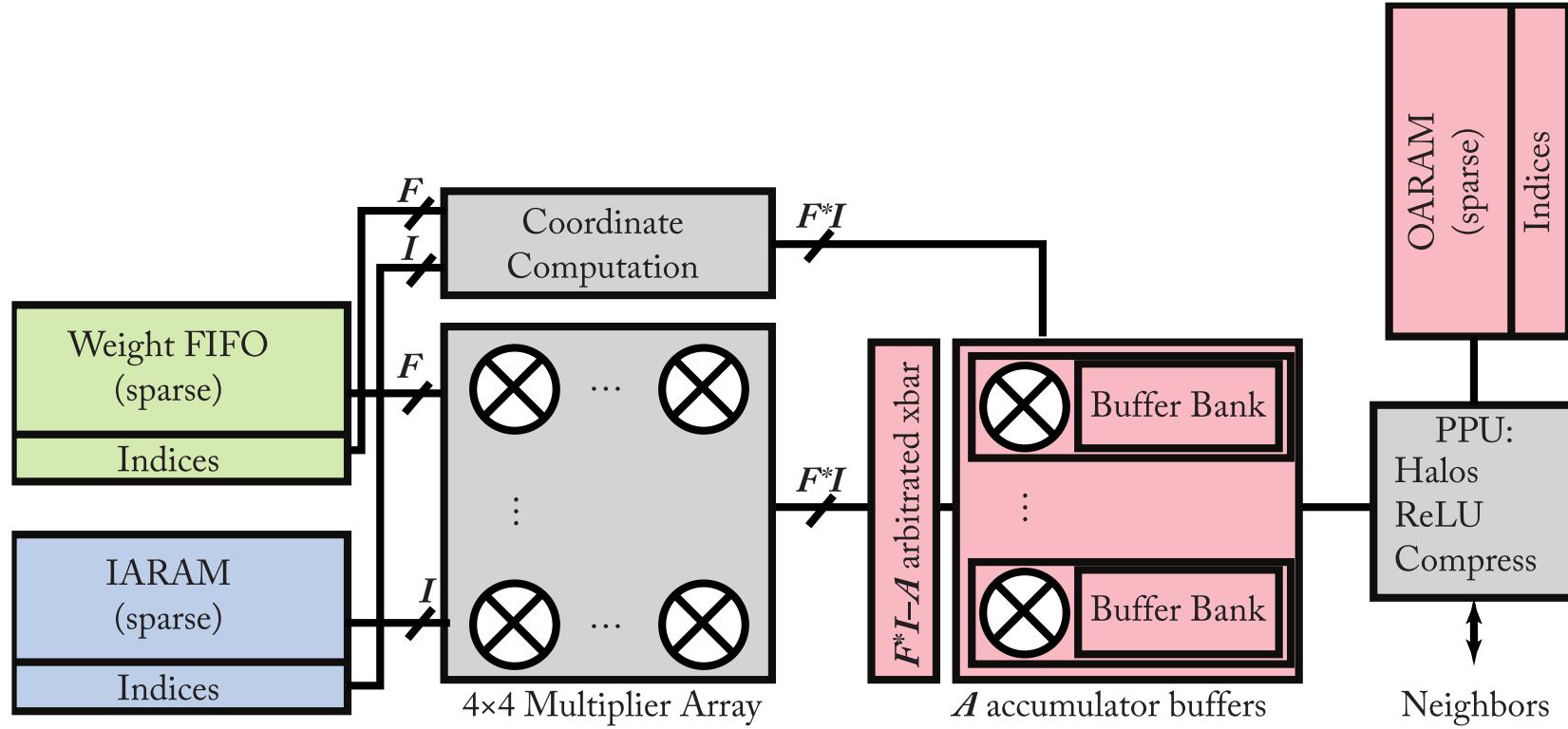


Input Stationary (IS)



- **Minimize activation read energy consumption**
 - Maximize convolutional and fmap reuse of activations
- Unicast weights and accumulate psums spatially
 - Across the PE array.
- Usually for sparse CNNs

IS Example: SCNN



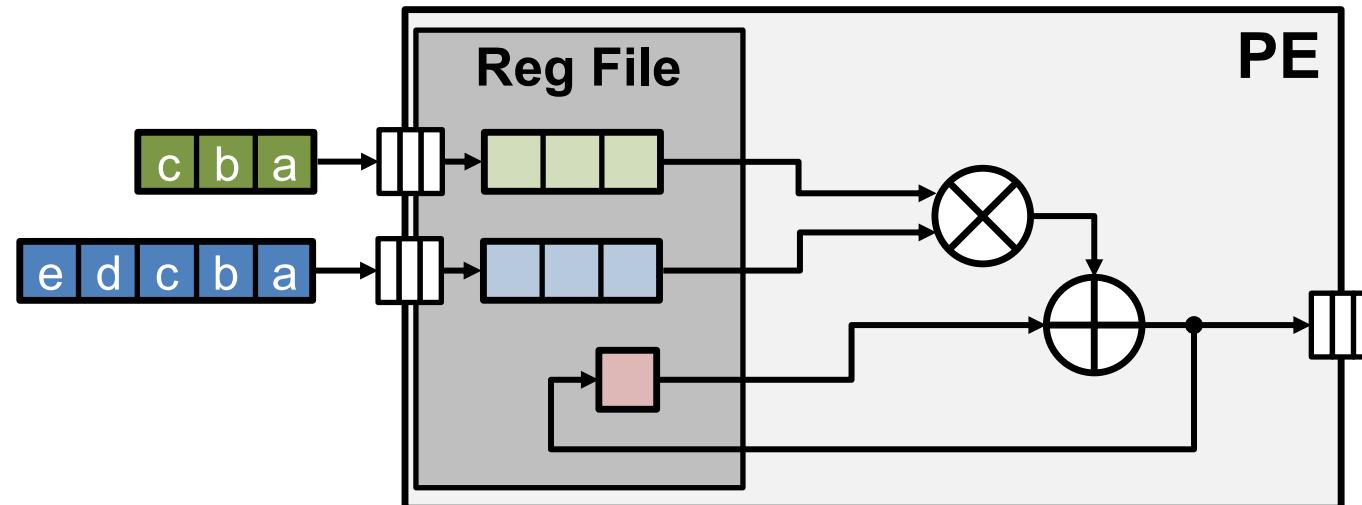
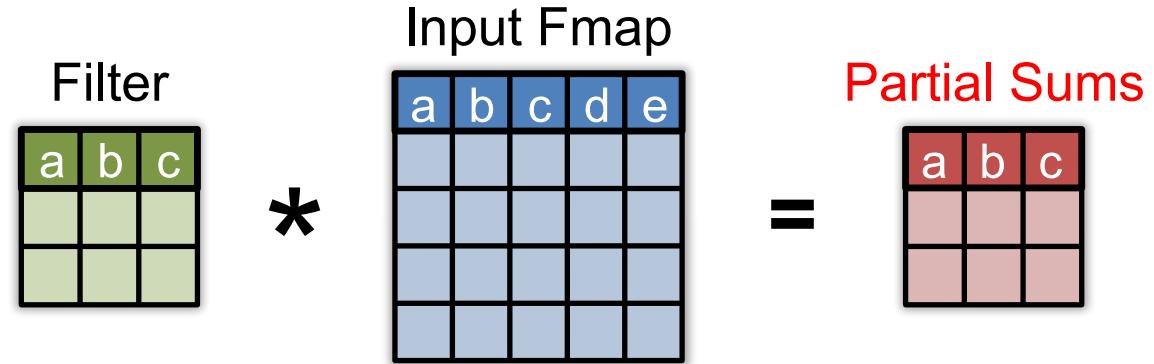
- Used for **sparse CNNs**
 - Sparse CNN is where **many weights are zeros**
 - Activations also have sparsity from ReLU

Multiple Datatype - Row Stationary (RW)

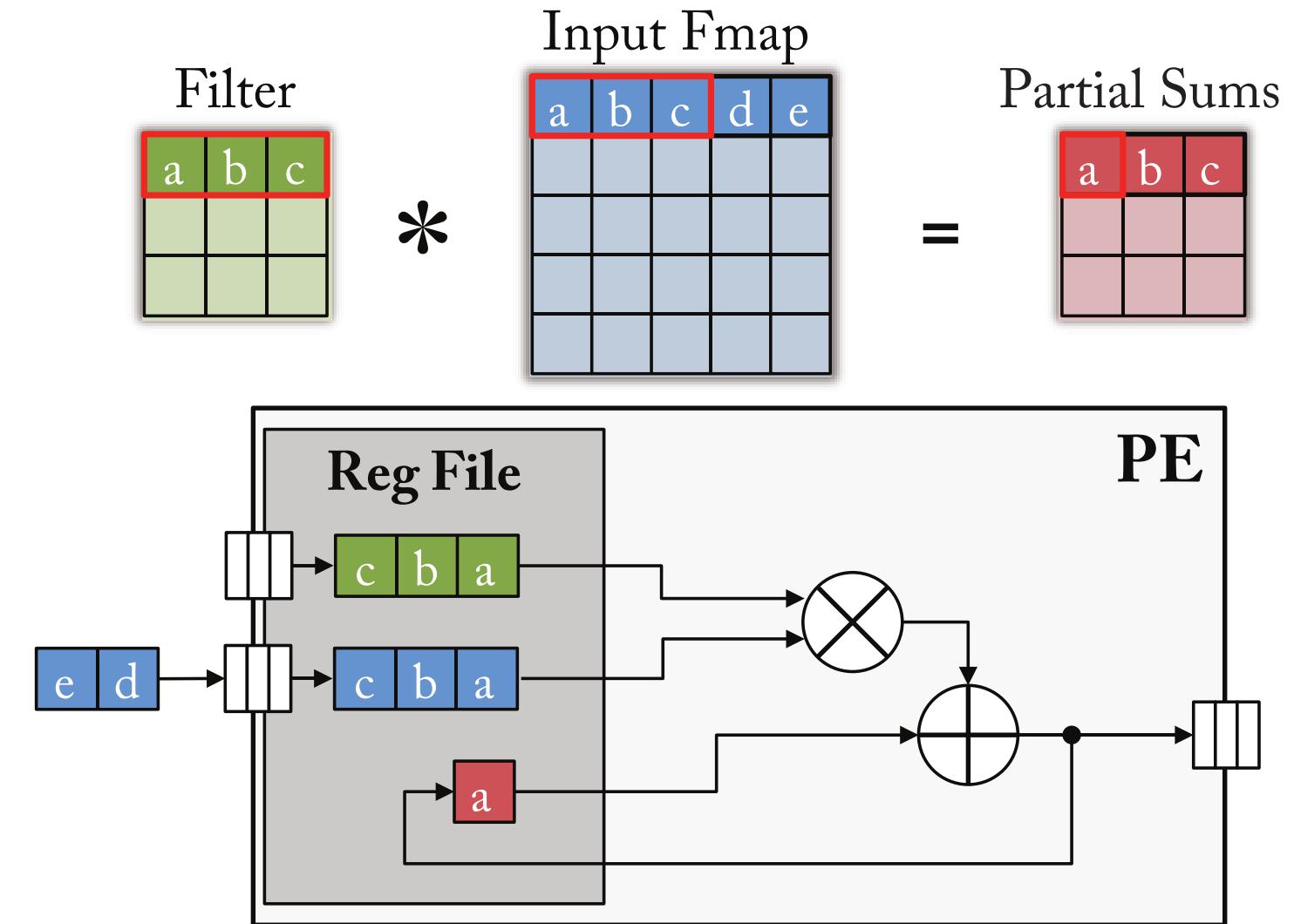


- **Eyeriss as an example**
- Maximize data reuse at Register File
 - Maximize row convolutional reuse in Register File
 - Keep a filter row and fmap sliding window in Register File
 - Maximize row psum accumulation in Register File
- **Optimize for overall energy efficiency instead for only a certain data type**

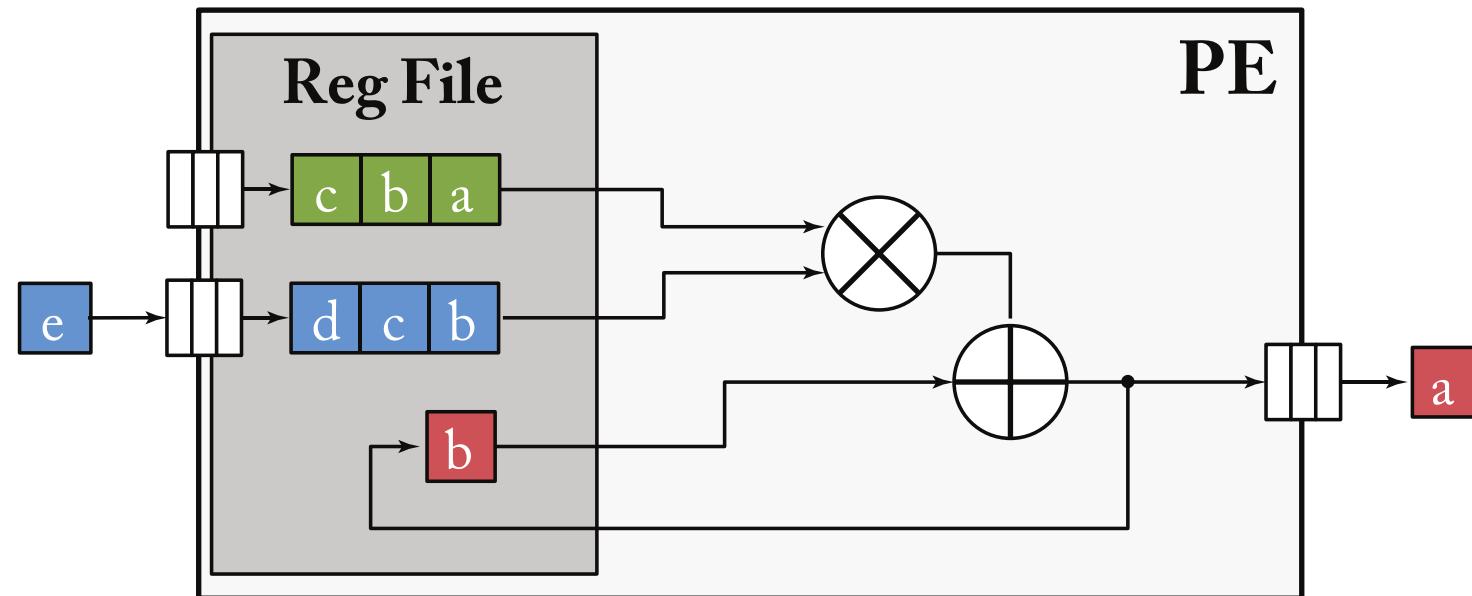
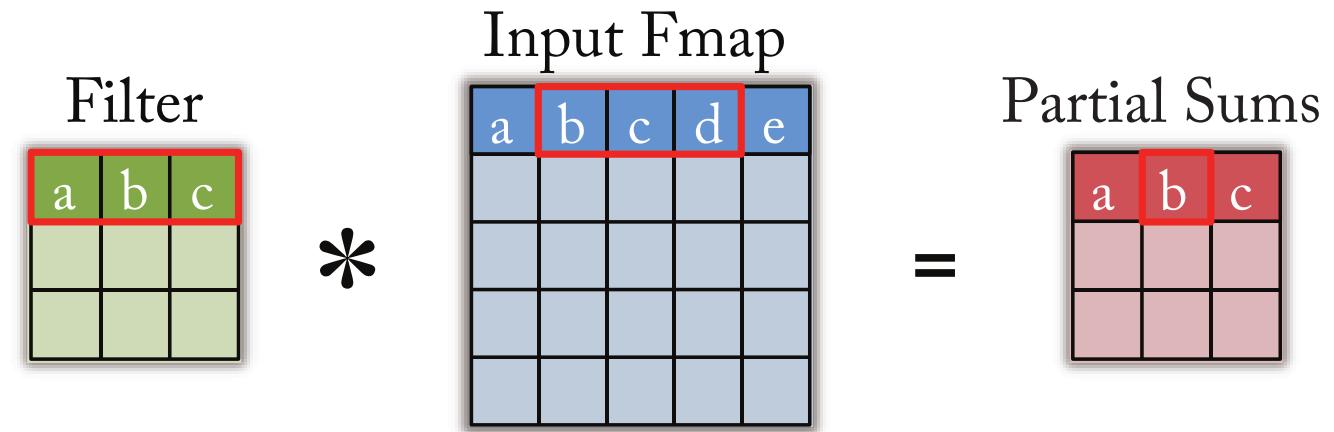
1D Row Convolution in PE



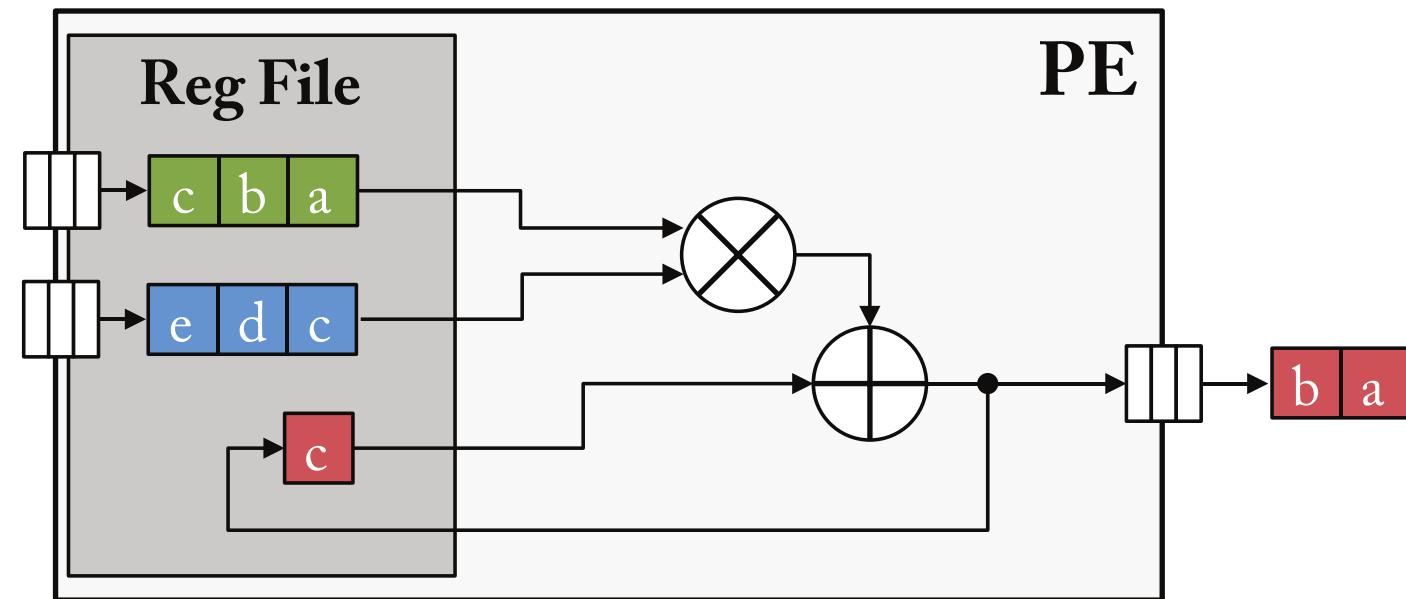
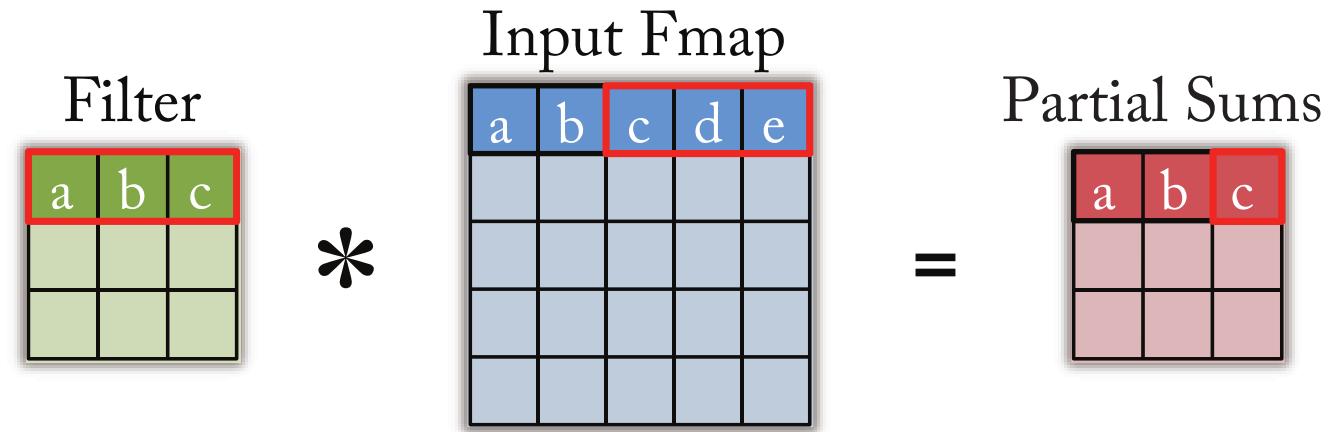
1D Row Convolution in PE



1D Row Convolution in PE



1D Row Convolution in PE

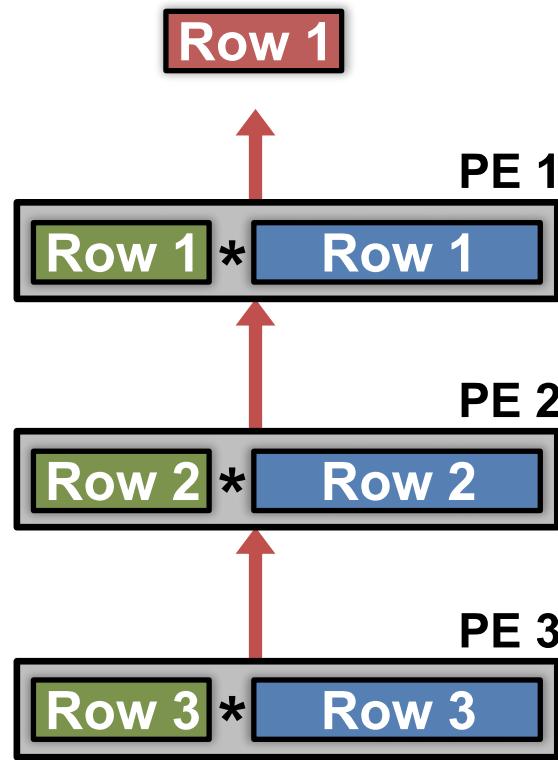


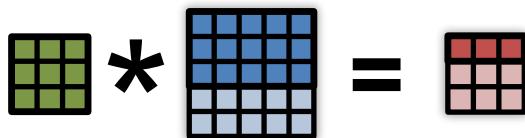
2D Convolution in PE Array



$$\begin{array}{c} \text{green} \\ \text{matrix} \end{array} * \begin{array}{c} \text{blue} \\ \text{matrix} \end{array} = \begin{array}{c} \text{red} \\ \text{matrix} \end{array}$$

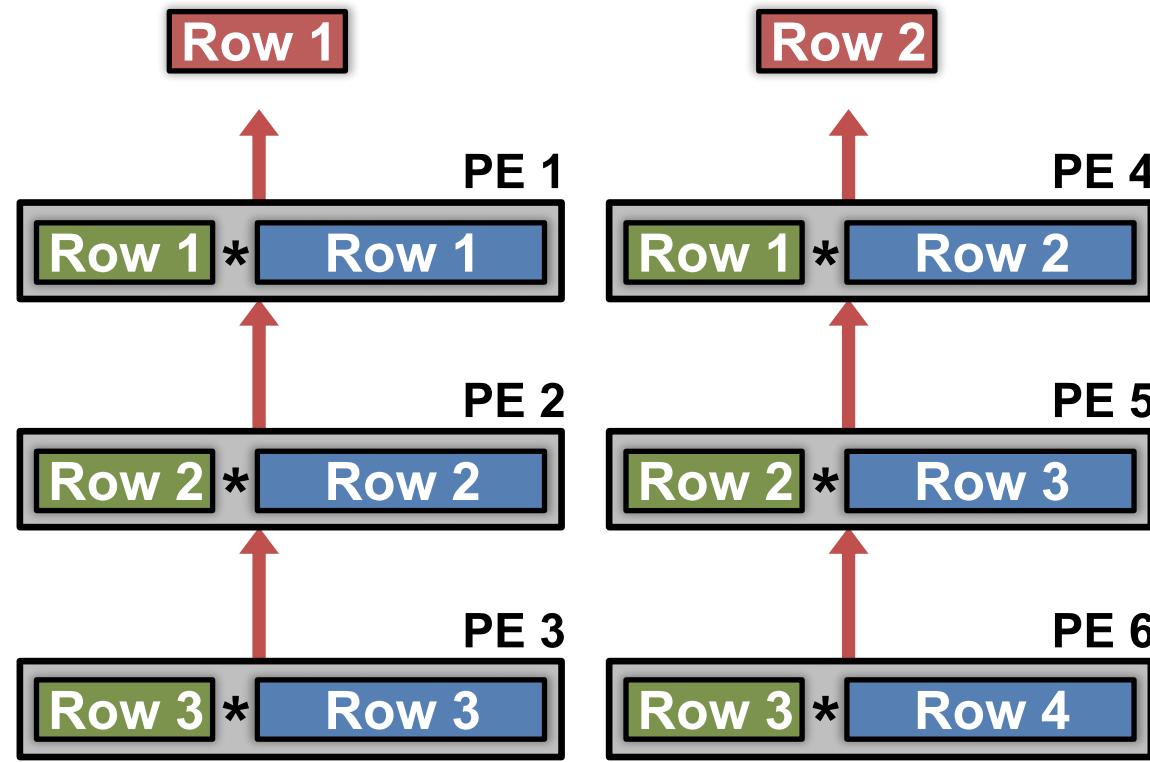
2D Convolution in PE Array





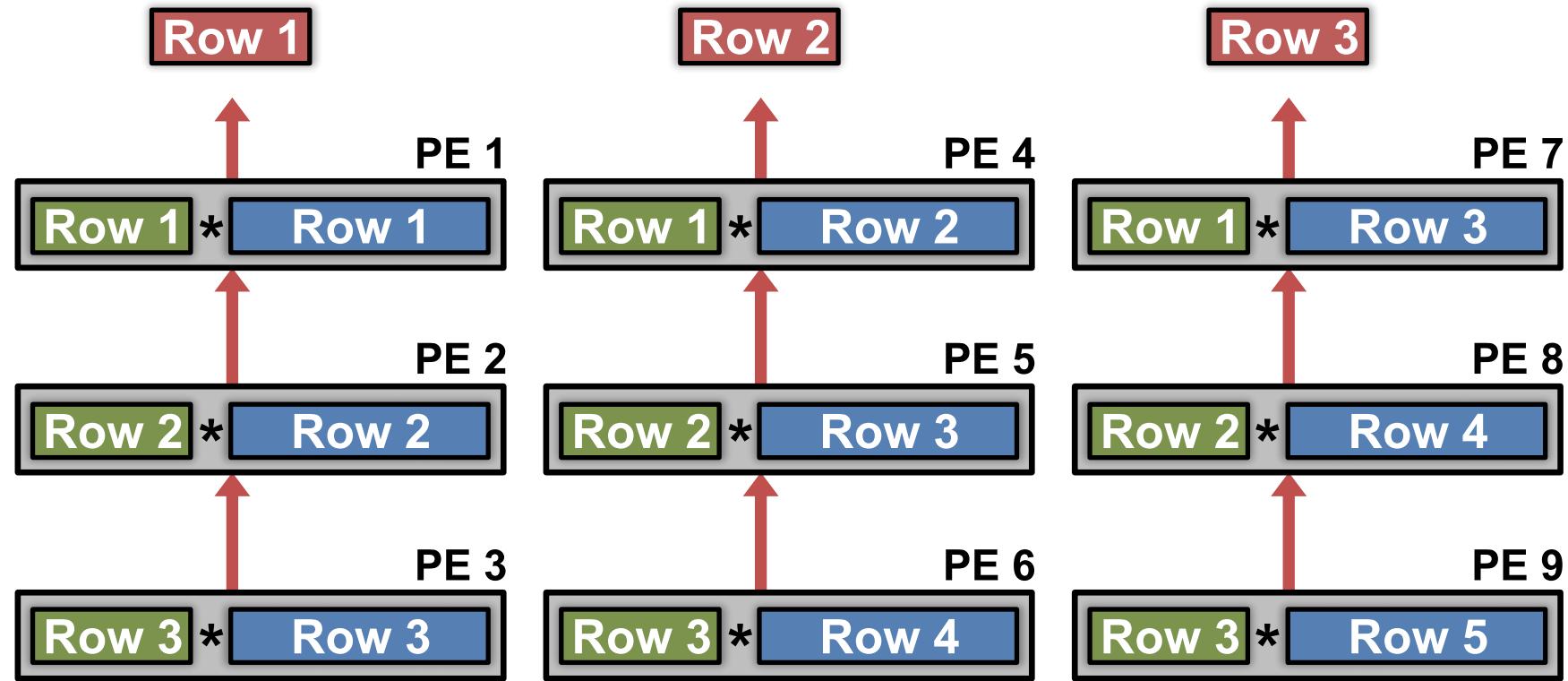
The diagram illustrates matrix multiplication. It shows three matrices: a green 2x2 matrix, a blue 2x3 matrix, and a red 2x2 matrix. An asterisk (*) symbol indicates the multiplication operation between the first two matrices, resulting in the third matrix.

2D Convolution in PE Array



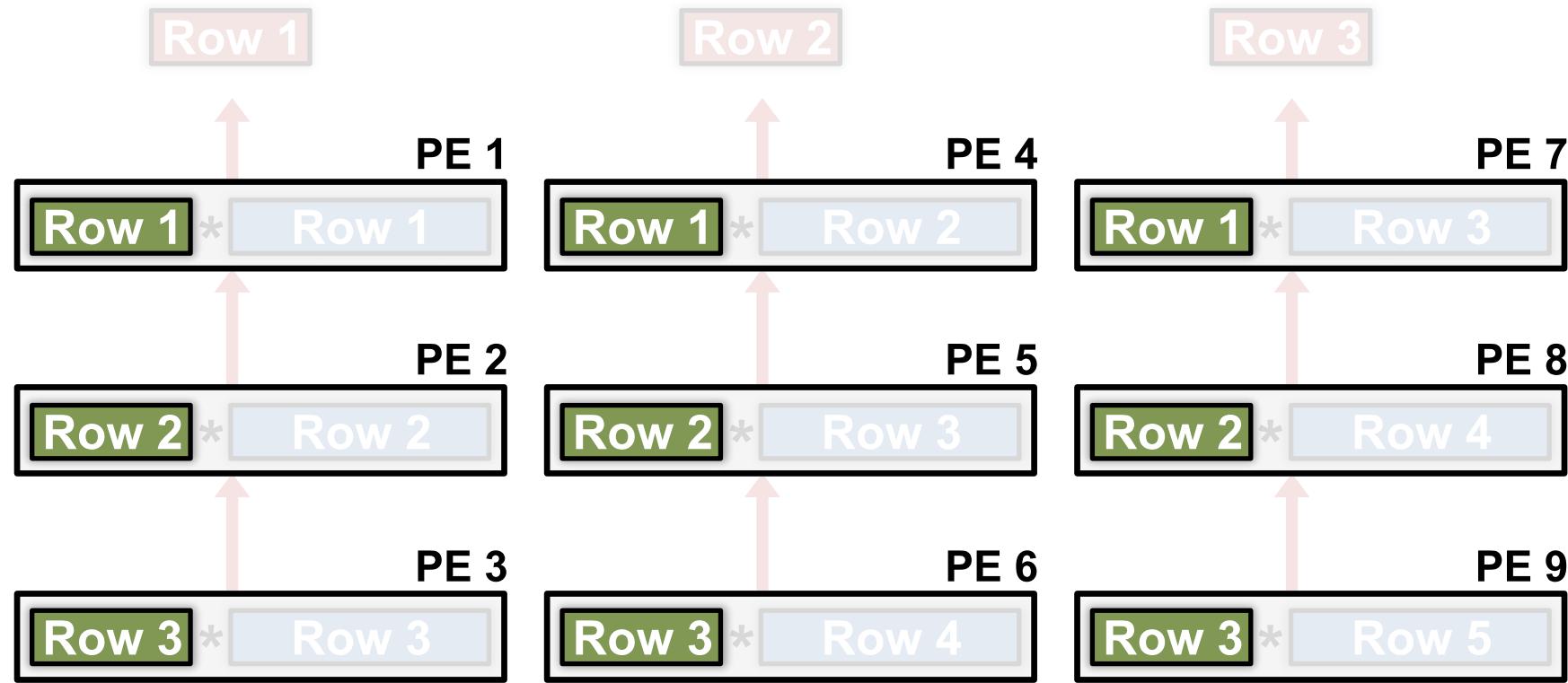
$$\begin{array}{c} \text{Green Grid} \\ \otimes \\ \text{Blue Grid} \end{array} = \text{Red Grid} \quad \begin{array}{c} \text{Green Grid} \\ \otimes \\ \text{Blue Grid} \end{array} = \text{Red Grid}$$

2D Convolution in PE Array



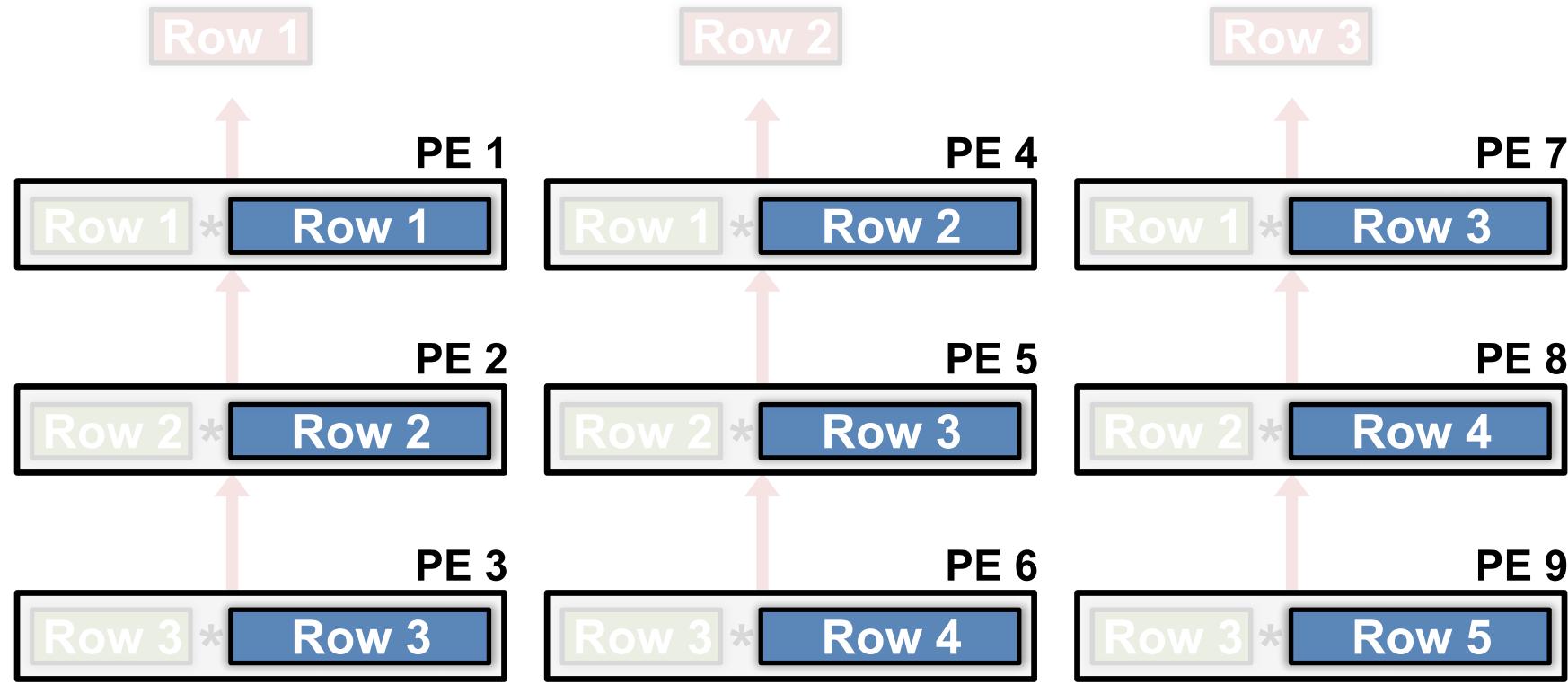
$$\begin{array}{c} \text{Green Matrix} \\ \times \\ \text{Blue Matrix} \end{array} = \text{Red Matrix}$$

Convolutional Reuse Maximized



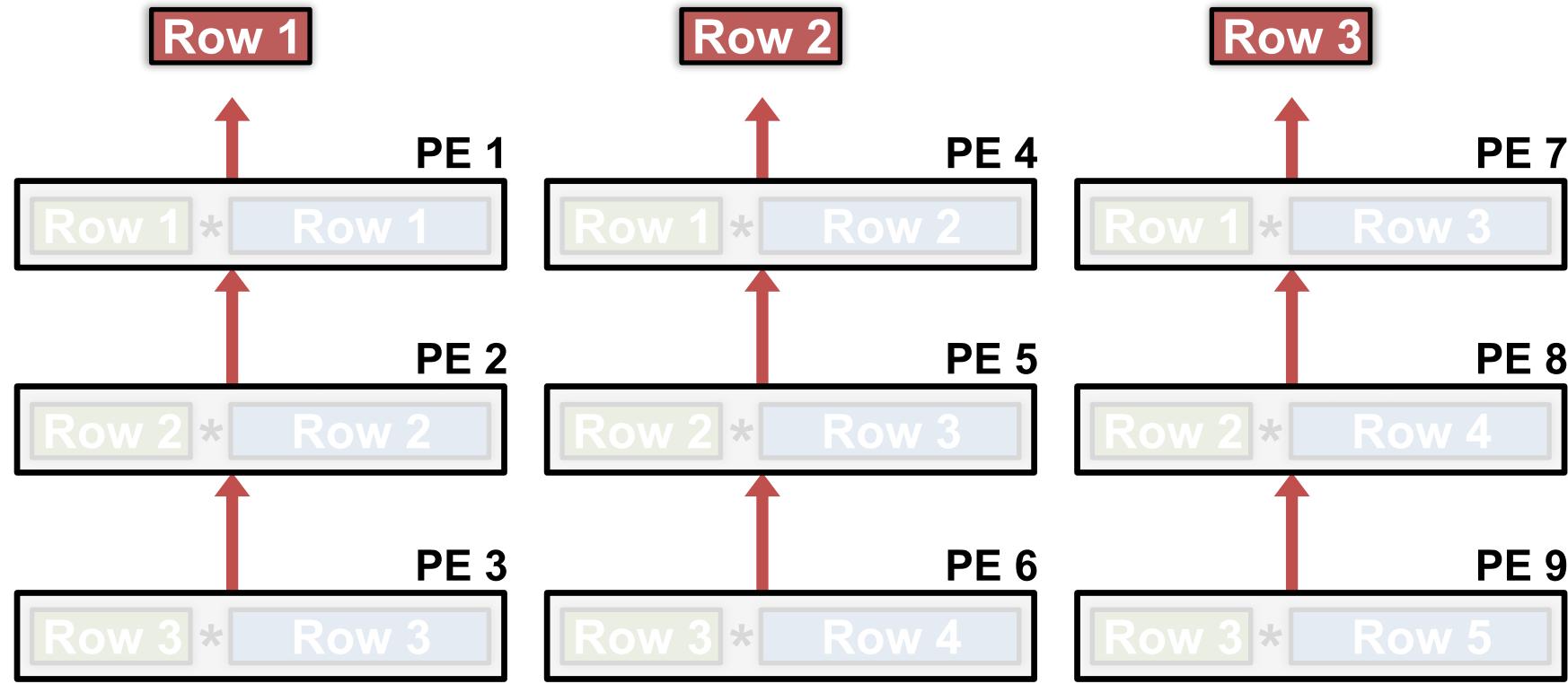
Filter rows are reused across PEs horizontally

Convolutional Reuse Maximized



Fmap rows are reused across PEs **diagonally**

Maximize 2D Accumulation in PE Array



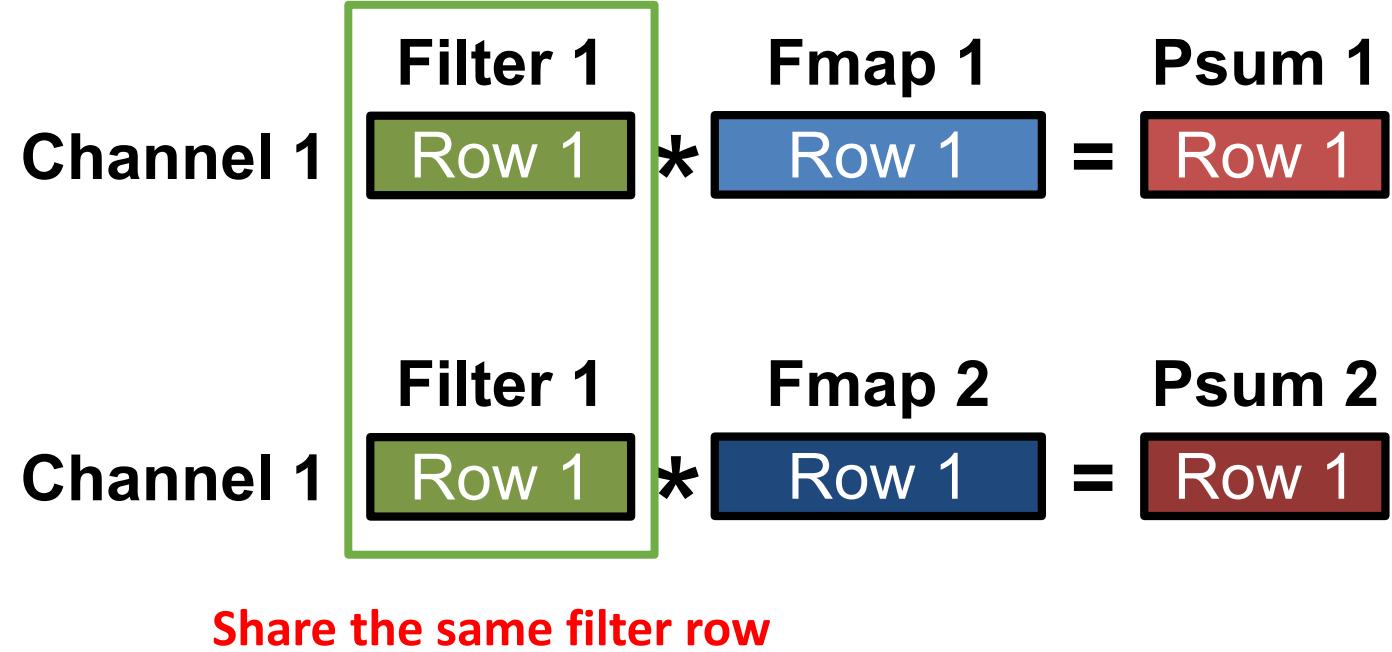
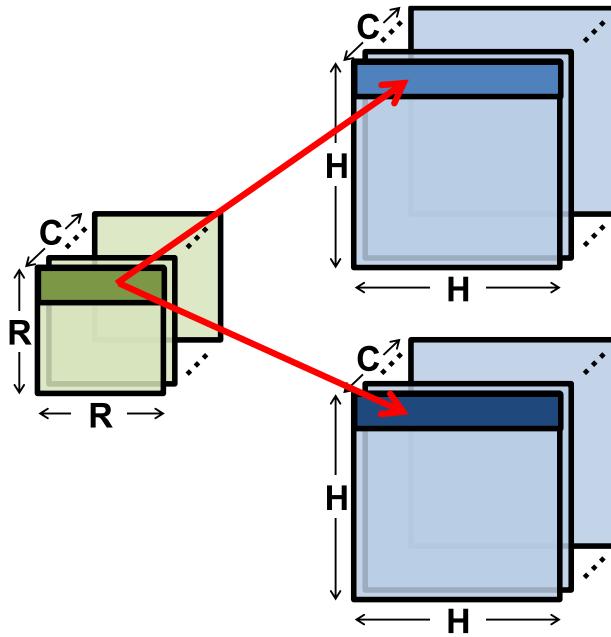
Partial sums accumulate across PEs **vertically**

Dimensions Beyond 2D Convolution



- Multiple Fmaps
- Multiple Filters
- Multiple Channels

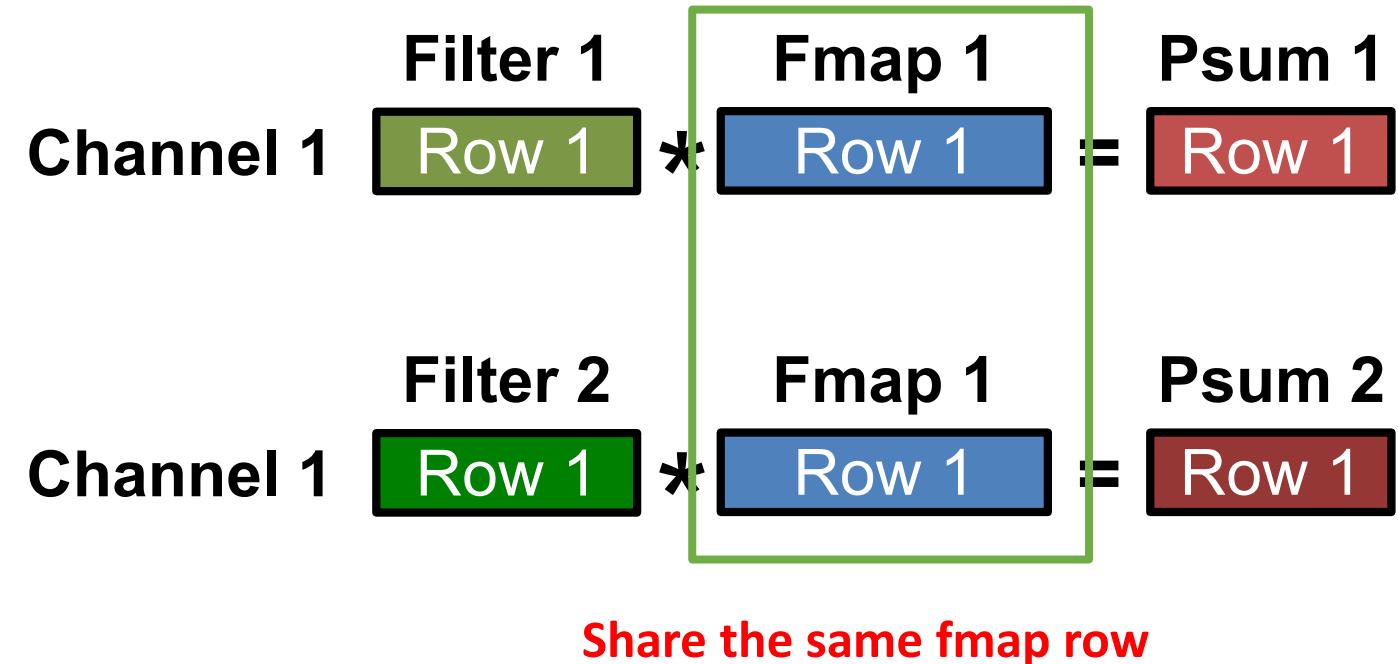
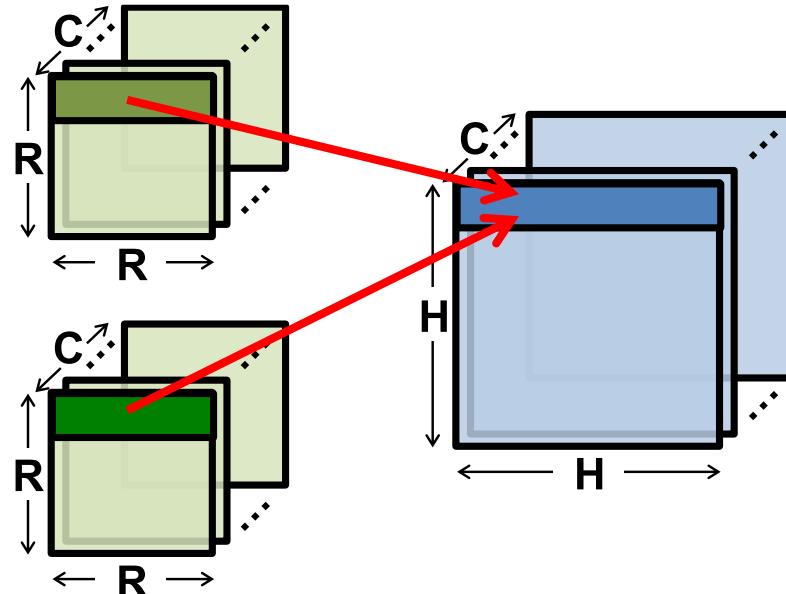
Multiple Fmaps - Filter Reuse in PE



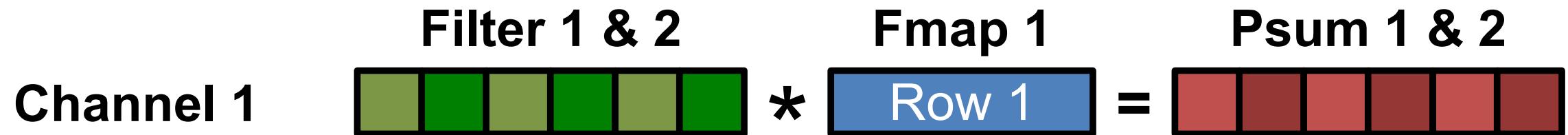
Processing in PE: concatenate fmap rows

$$\begin{array}{c}
 \text{Filter 1} \quad \text{Fmap 1 \& 2} \quad \text{Psum 1 \& 2} \\
 \text{Channel 1} \quad \text{Row 1} * \text{Row 1} \quad \text{Row 1} = \text{Row 1} \quad \text{Row 1}
 \end{array}$$

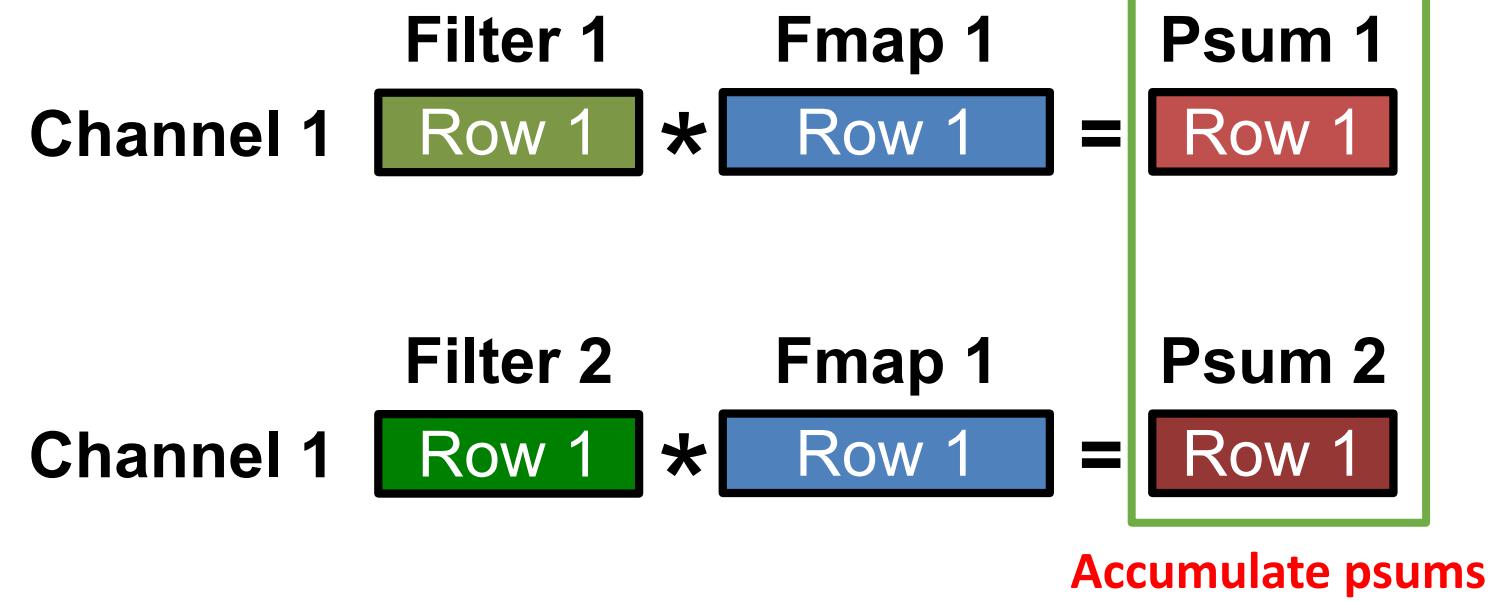
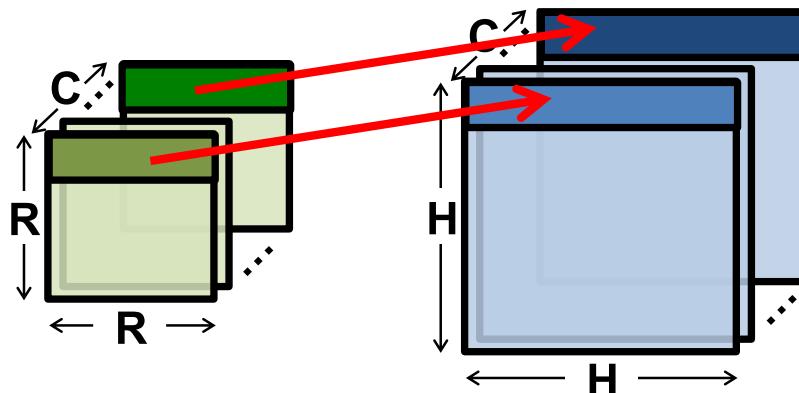
Multiple Filters - Fmap Reuse in PE



Processing in PE: interleave filter rows



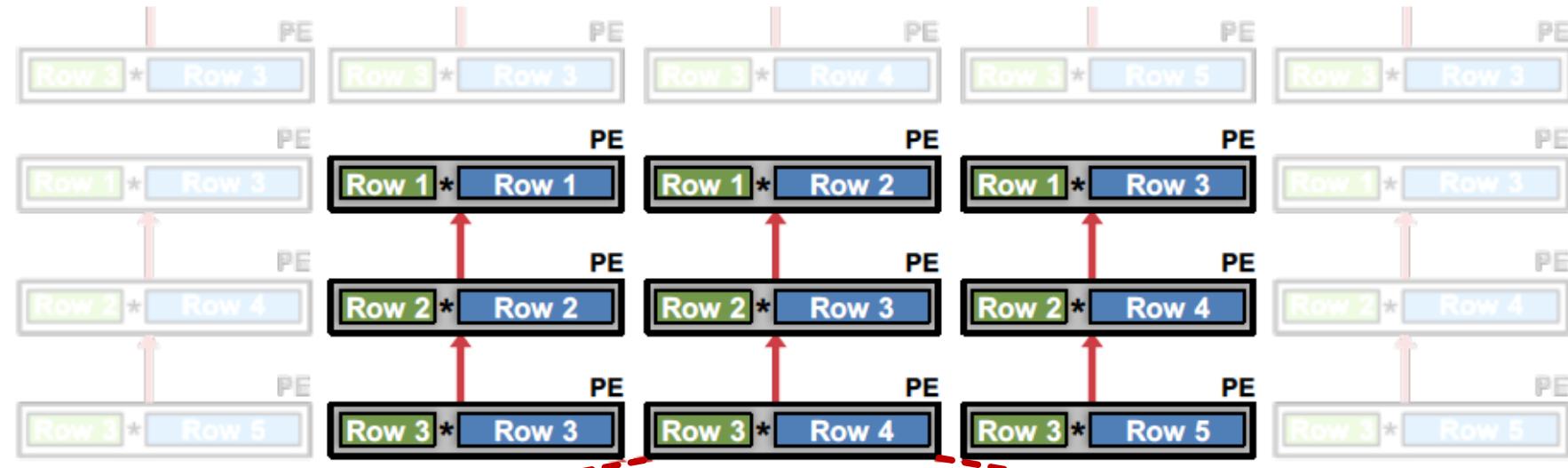
Multiple Channels - Channel Accumulation in PE



Processing in PE: interleave filter rows

	Filter 1	Fmap 1	Psum
Channel 1 & 2			
	*		Row 1

The Full Picture



Multiple **fmaps**:

$$\text{Filter 1} * \text{Fmap 1 \& 2} = \text{Psum 1 \& 2}$$

Multiple **filters**:

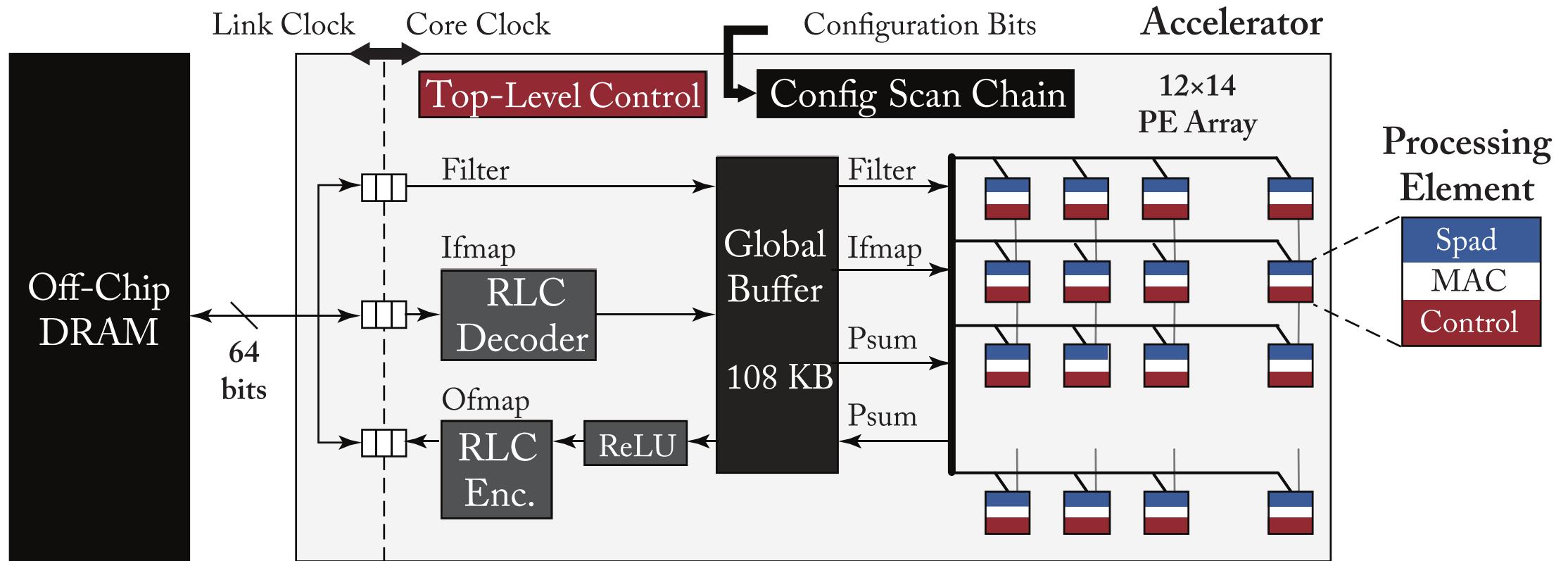
$$\text{Filter 1 \& 2} * \text{Fmap 1} = \text{Psum 1 \& 2}$$

Multiple **channels**:

$$\text{Filter 1} * \text{Fmap 1} = \text{Psum}$$

Map rows from **multiple fmaps**, **filters** and **channels** to same PE to exploit other forms of reuse and local accumulation

Eyeriss DNN accelerator



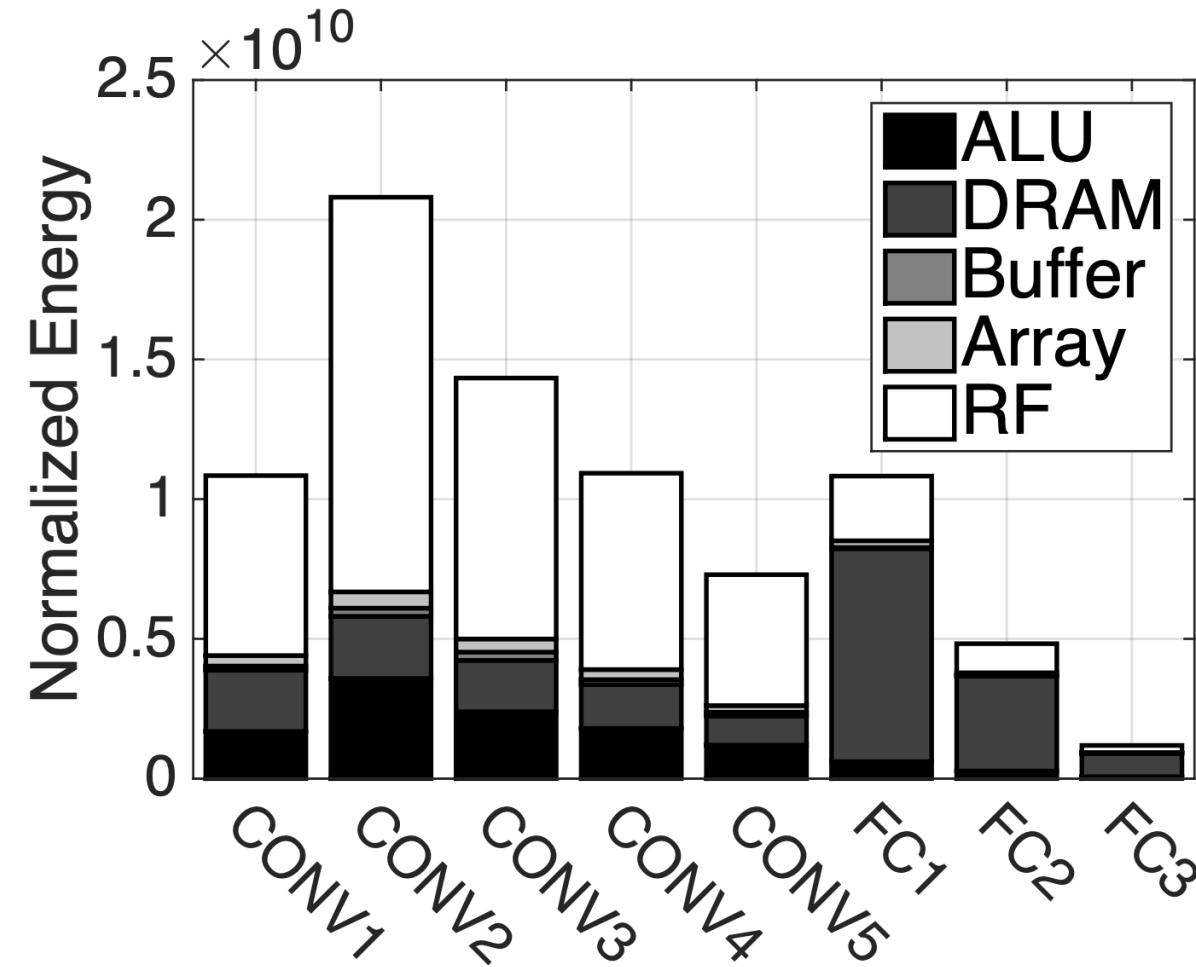
Taxonomy: More Examples

Dataflow	Recent Work
Weight Stationary	NVDLA, TPU, neuFlow, Sankaradas et al., Park et al., Chakradhar et al., Sriram et al., Origami, ISAAC, PRIME, etc.
Output Stationary	DaDianNao, DianNao, Zhang et al., Moons et al., ShiDianNao, Gupta et al., Peeman et al., Peemen, Moons, Thinker, etc.
Input Stationary	SCNN, etc.
Multiple data-type Stationary	Eyeriss v1(Row Stationary), Eyeriss v2(Row Stationary), etc.

Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
 - Locality and Data Reuse
 - Dataflow Taxonomy
 - Data Orchestration
 - Network-on-Chip
 - Optimization
- Roofline Model
- Energy

Data Movement Dominates ML HW Energy



Eyeriss, ISCA'2016

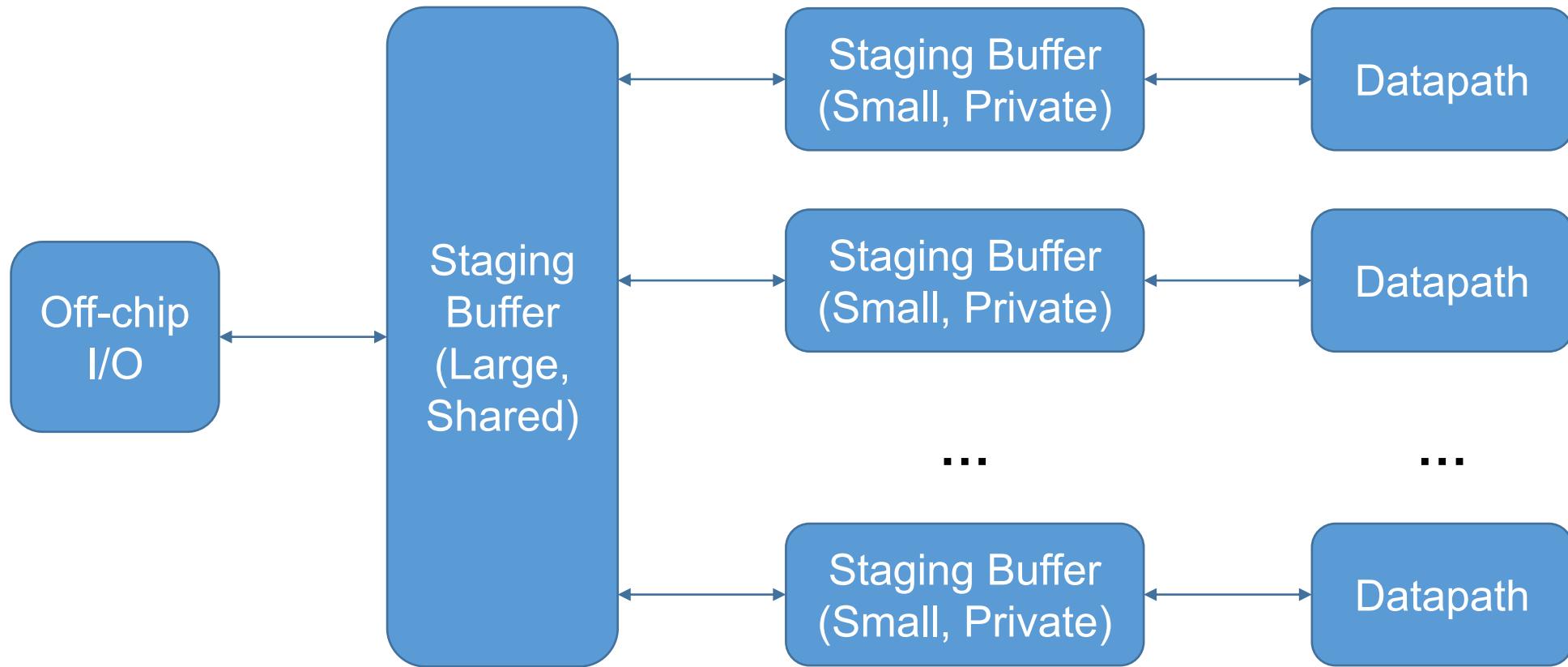
Buffer Hierarchy In ML Accelerators.

- Percentage of on-chip area that devotes to on-chip buffers

DaDianNao [5]:	48%	Eyeriss [6]:	40%-93%
EIE [18]:	93%	SCNN [35]:	57%
TPU [22]	35%	PuDianNao [27]	63%

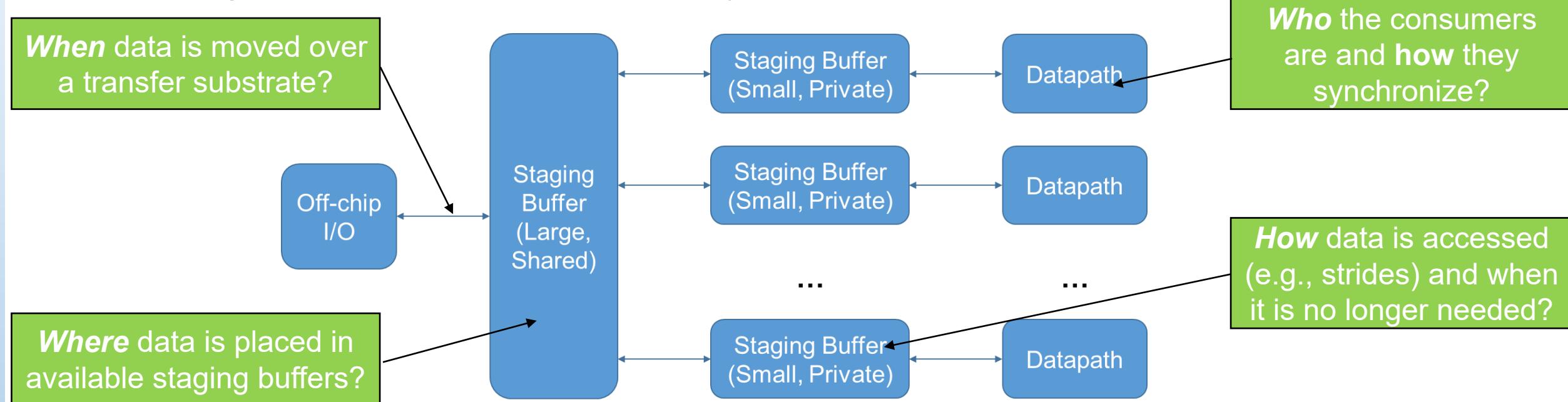
What's Data Orchestration?

- Feeding data to a functional unit exactly when it wants it.
- Removing data from a buffer exactly when it's not used.



What's data orchestration?

- Feeding data to a functional unit exactly when it wants it.
- Removing data from a buffer exactly when it's not used.



- ML accelerators use workload knowledge to optimize data orchestration at design time.

Agents in Data Orchestration



- Data Producer:
 - The agent that currently contains the requested data.
- Data Consumer
 - The agent that consumes the requested data.
- Data Requestor
 - The agent that sends out data requests.
- Data Distributor
 - The agent that distributes data requests.

Producer

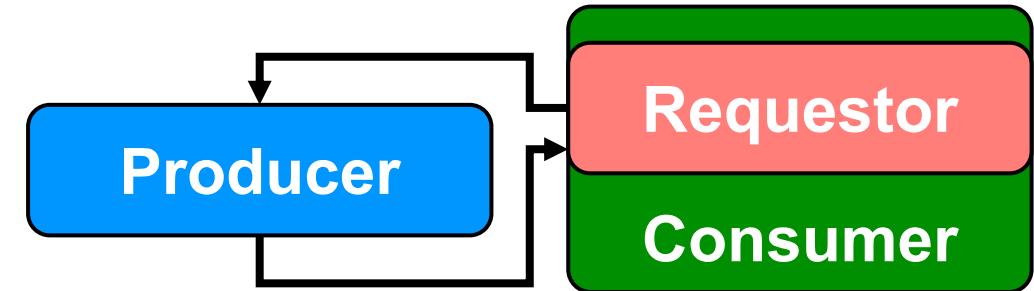
Consumer

Requestor

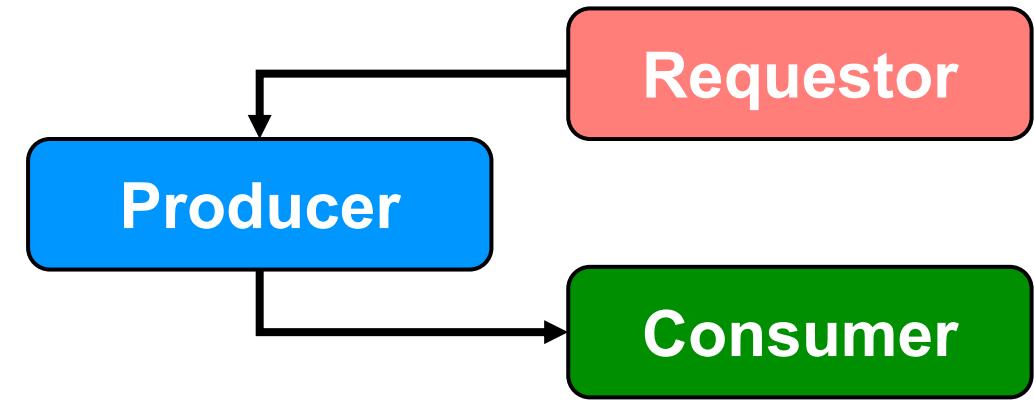
Distributor

Coupled vs. Decoupled

- Whether requestor == consumer.
- Coupled
 - Requestor is the same as Consumer.
 - Pro: Easy and intuitive synchronization.
 - Con: Reserved landing zone (e.g., registers) for incoming data.
- Decoupled
 - Requestor is different from Consumer.
 - Pro: Requestor can run ahead.
 - Con: Synchronization between requestor and consumer.



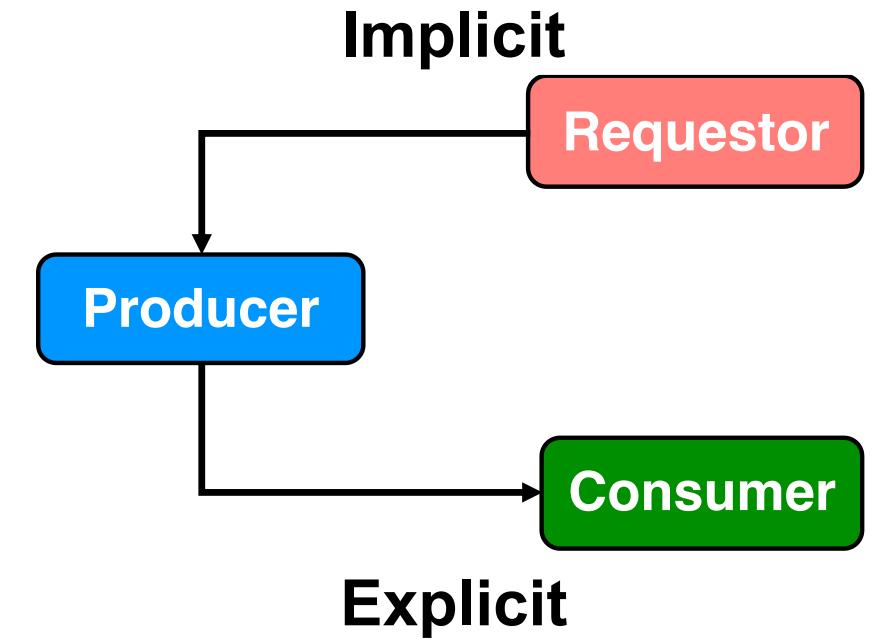
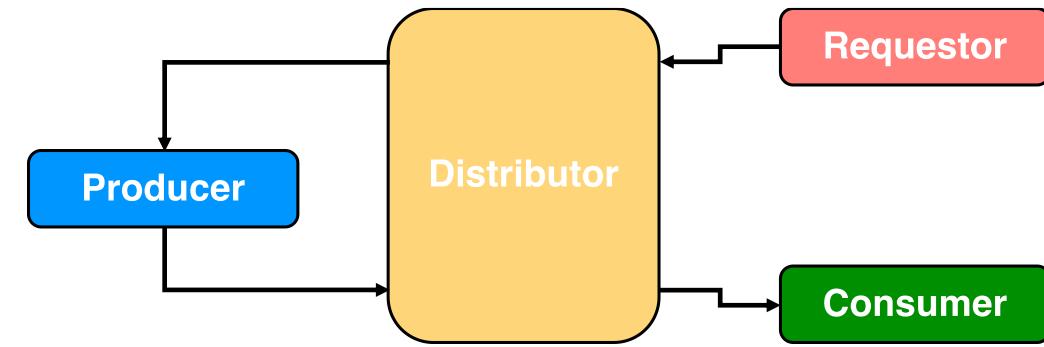
Coupled



Decoupled

Implicit vs. Explicit

- Whether requests are sent to data producer directly.
- Implicit:
 - Requestor is not aware of where the data is and when the data is evicted.
 - Pro: Easy to program; Workload agonistic.
 - Con: Area and energy overhead for tags etc.
- Explicit:
 - Requestor explicitly interacts with producer.
 - Pro: Low overhead.
 - Con: Hard to program.

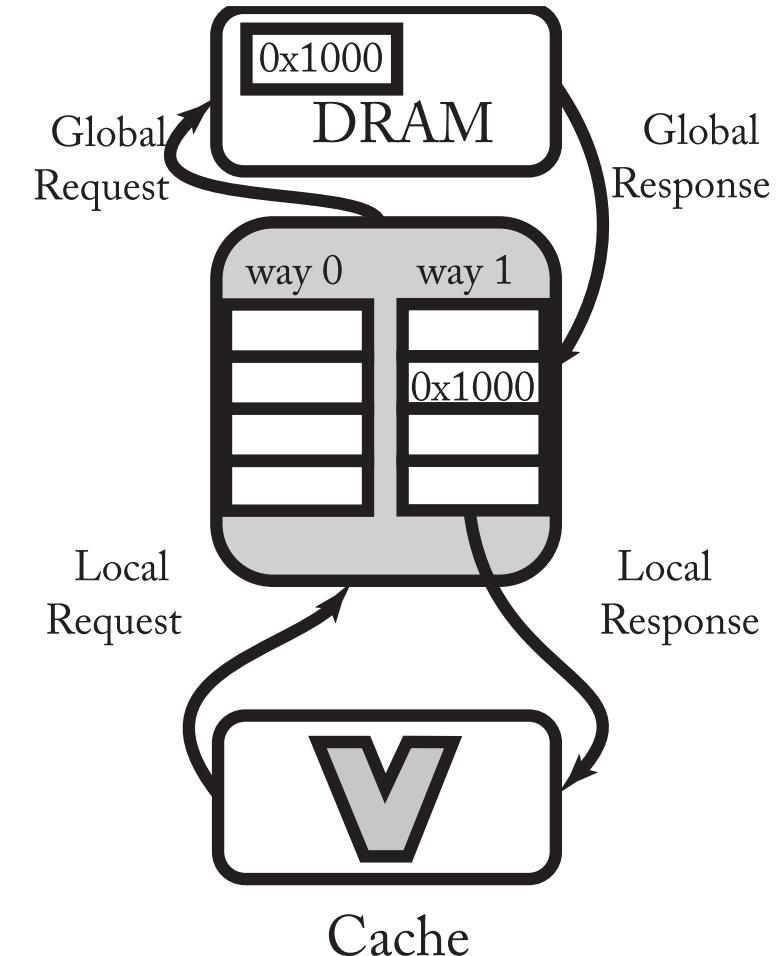


Data Orchestration Taxonomy

	Coupled	Decoupled
Implicit	Cache	Decoupled Access-Execute
Explicit	GPU shared memory	DMA

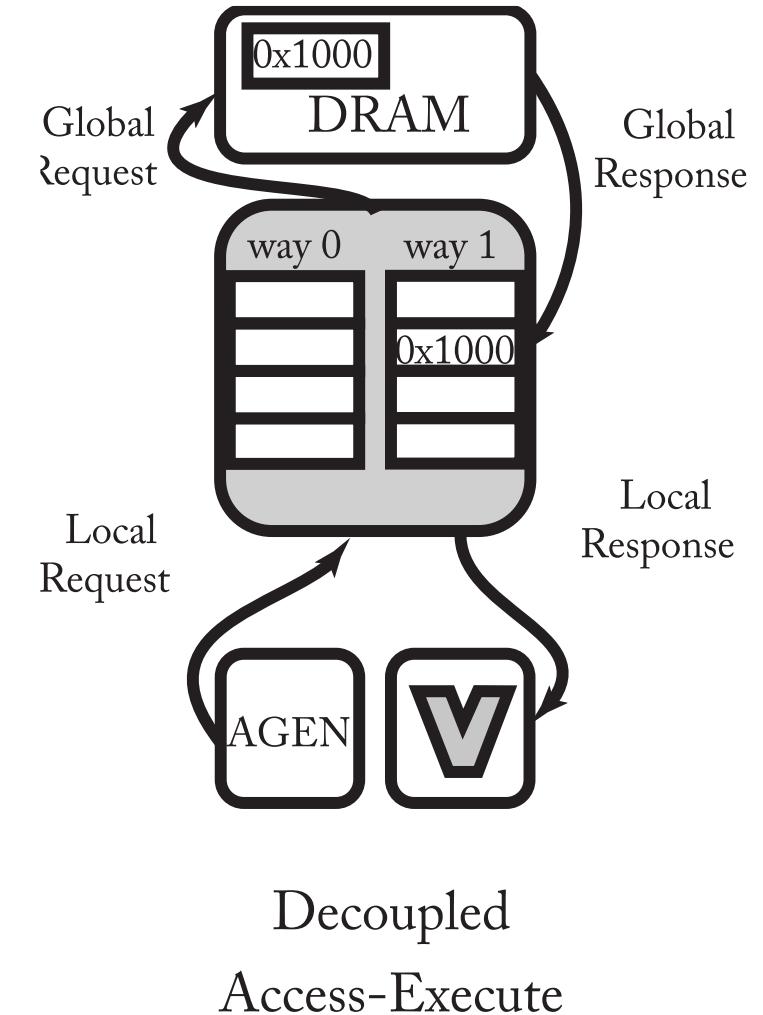
Implicit-Coupled Data Orchestration

- Cache
 - Widely used in general-purpose computing
 - Reusable, composable
 - High area and energy overhead
- Implicit:
 - Requestor, i.e., core, sends requests to L1 without knowing the exact location of data.
- Coupled:
 - Unified pipeline for mem and compute instructions.



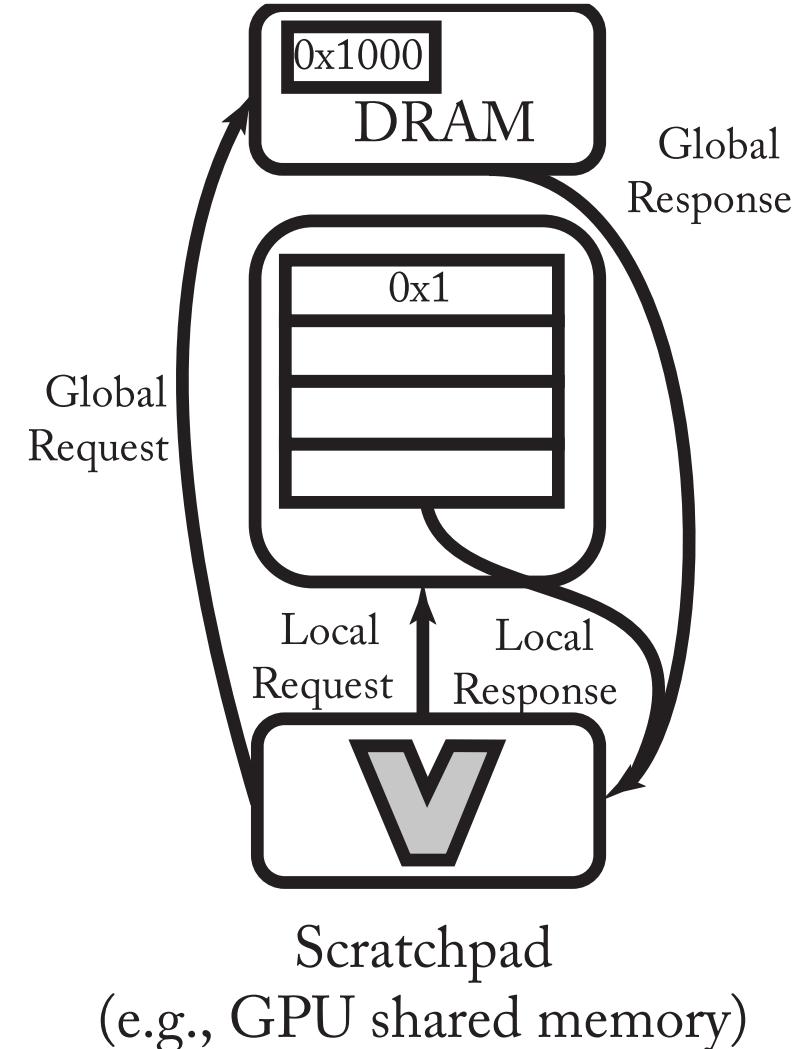
Implicit-Decoupled Data Orchestration

- Decoupled Access-Execute
 - Separating data requestor and consumer connected via hardware queues.
 - Targeting general-purpose computing.
 - Allowing both to proceed at their own rates.
- Implicit:
 - Requestor, or accessor, sends requests to L1 without knowing the exact location of data.
- Decoupled:
 - Separate requestor/accessor and consumer/executor



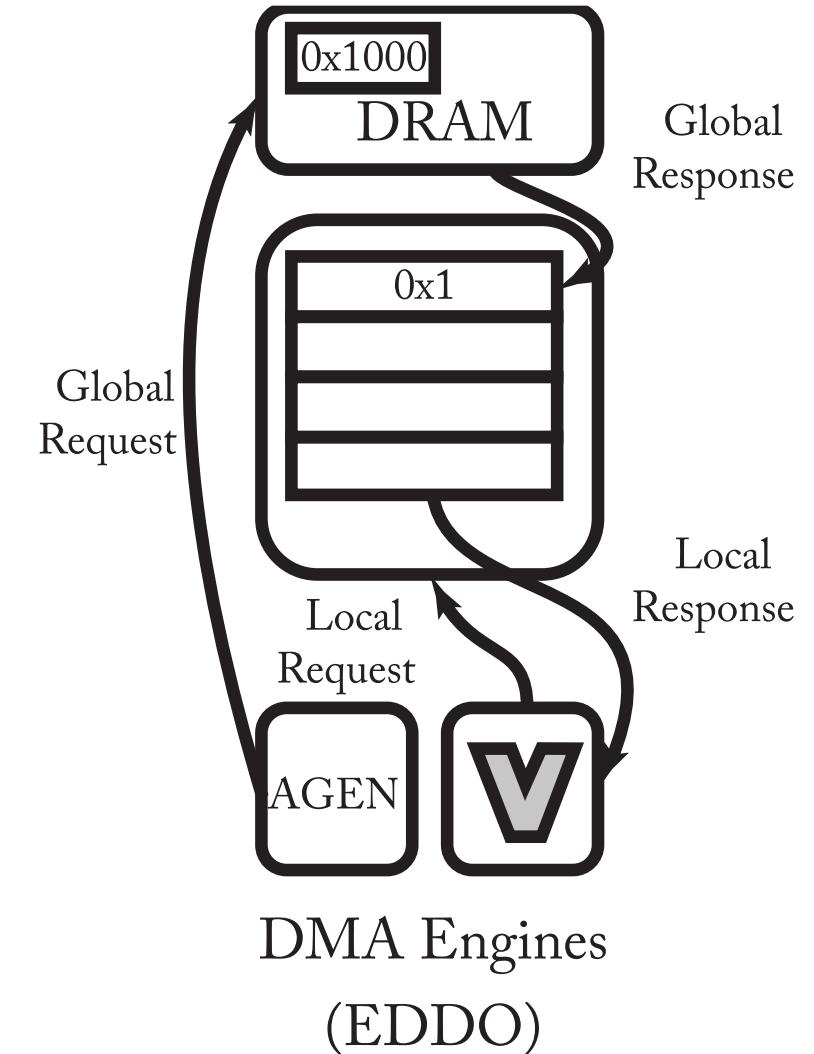
Explicit-Coupled Data Orchestration

- GPU Shared Memory
 - Scratchpad in GPU
 - Instructions to explicitly move data across memory hierarchy
- Explicit:
 - Core explicitly requests data from DRAM
- Coupled:
 - The same pipeline for data requests and consumption.



Explicit-Decoupled Data Orchestration (EDDO)

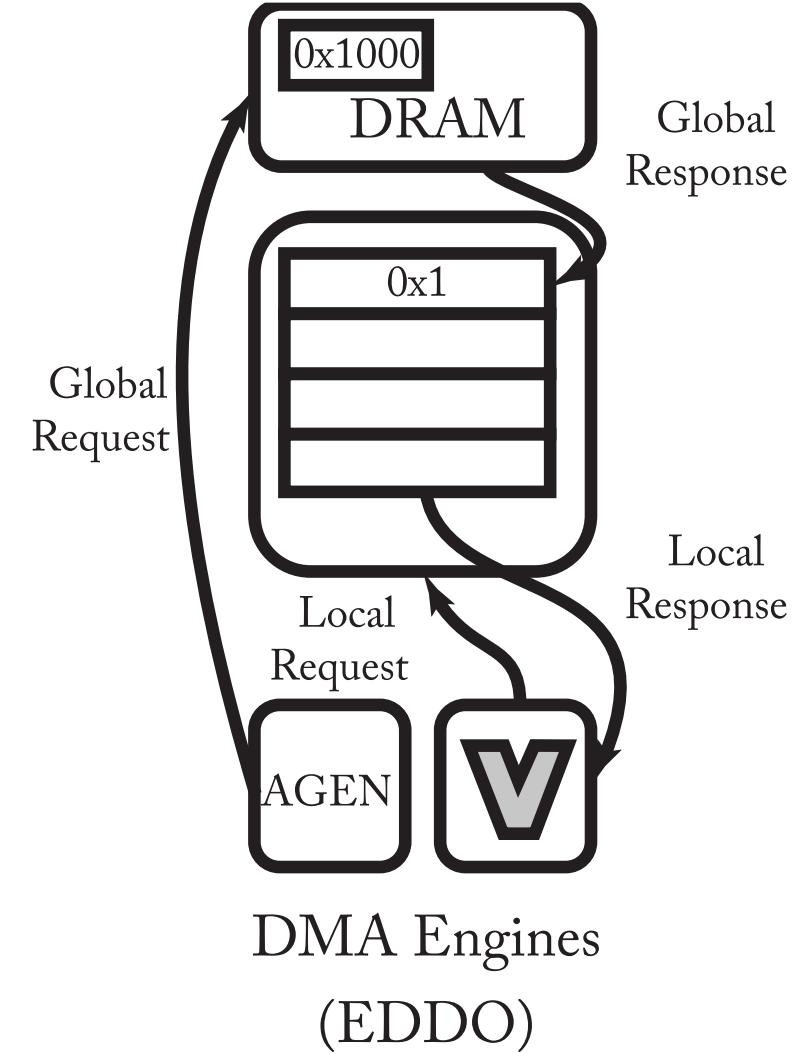
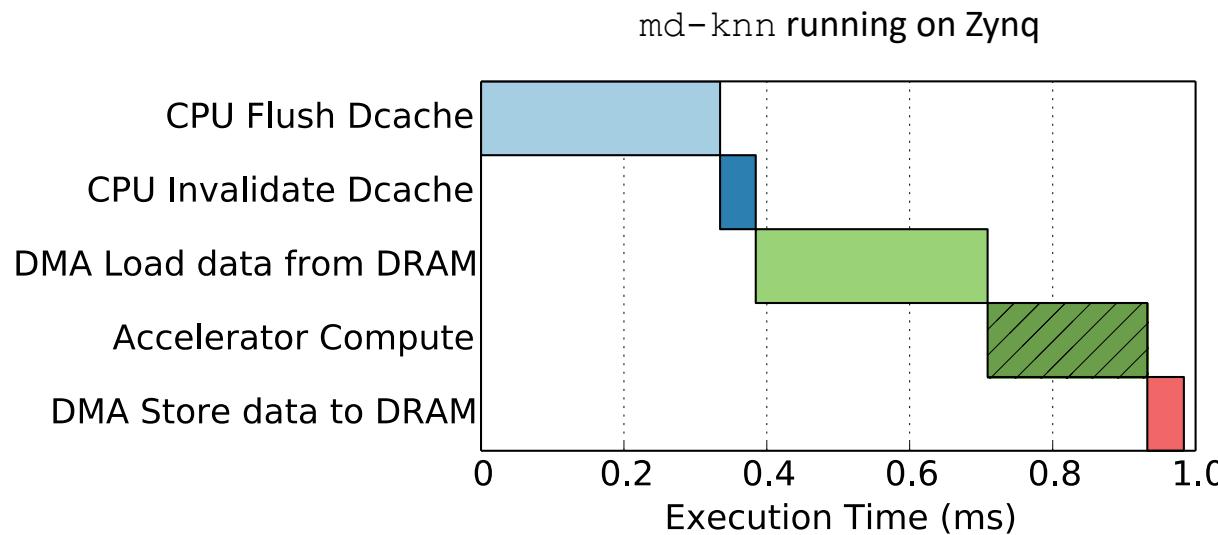
- DMA engines
 - Direct-Memory Access
 - Acts as an address (request) generator to DRAM
 - “Pushing” data to consumer
- Explicit:
 - Software-managed data placement
- Decoupled:
 - DMA only does data orchestration not compute.



EDDO for Domain-Specific Accelerators



- Benefits:
 - Leverage static workload knowledge
 - Efficient hardware implementations
- Challenges:
 - Synchronization in a decoupled system

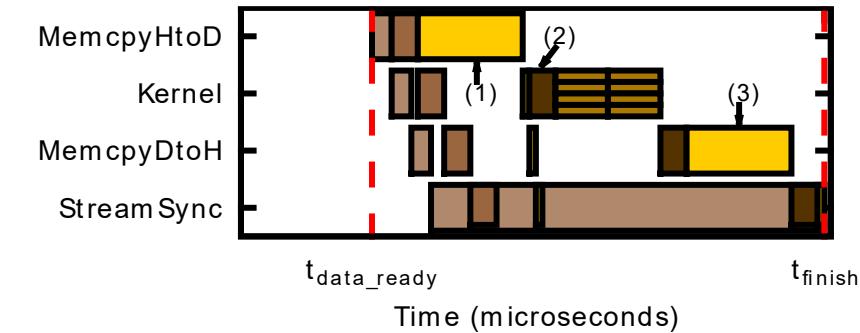


Types of EDDO Example

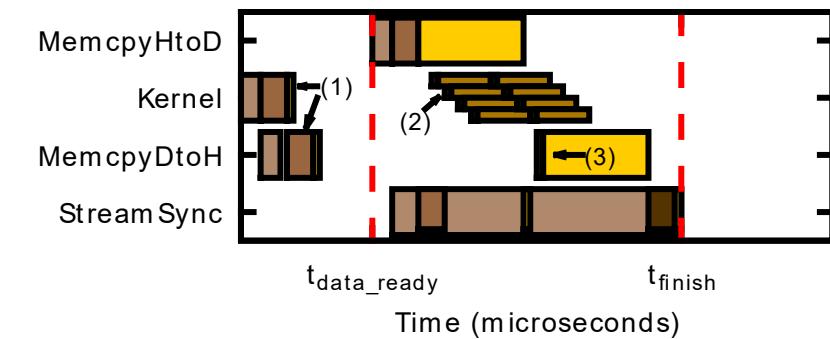
- Full/Empty Bits Synchronization
- FIFO-based Synchronization
- Buffet-based Synchronization

Full/Empty Bits Synchronization

- Use full-empty bits to track when regions of data have been transferred.
 - A full-empty bit per word
 - Producer writes only if the full-empty bit is empty, and sets it to full
 - Consumer reads only if the full-empty bit is set to full
- Pro:
 - Fine-grained synchronization
- Con:
 - Hardware cost



(a) Baseline. Labeled portions described in Section 2.1.



(b) Performance Improvement using the “full-overlap” scenario. Labeled portions described in Section 2.3.

Performance Improvement using the “full-overlap” scenario.
 (Heterogeneous Element Processor, Smith’1982, Reducing GPU Offload Latency via Fine-Grained CPU-GPU Synchronization, HPCA’2013).

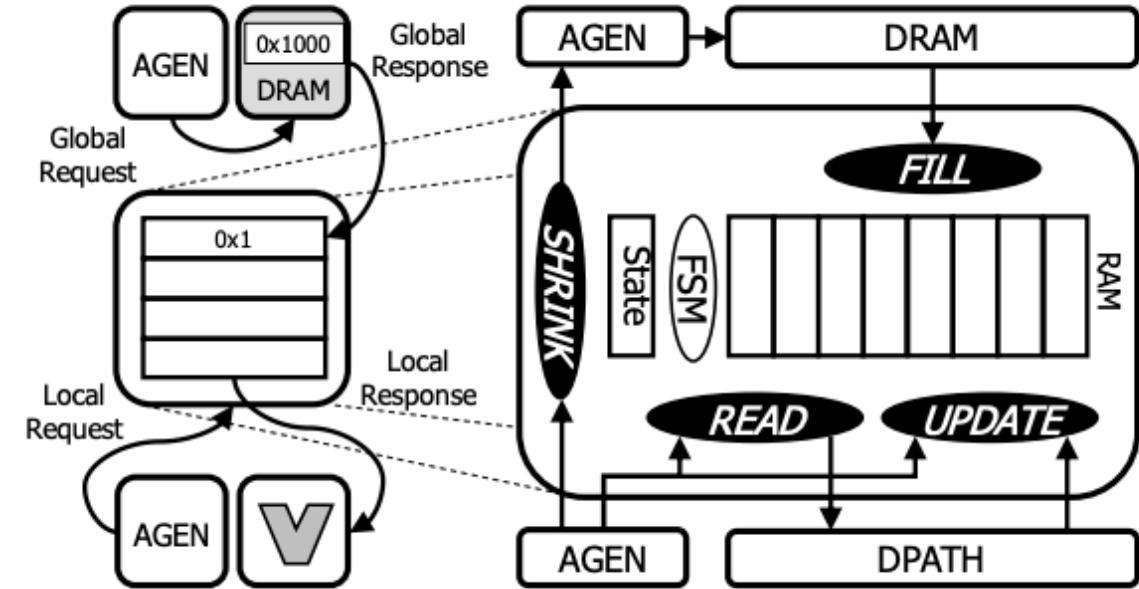
FIFO-based Synchronization

- FIFO is also EDDO.
 - Explicitly pushes/pops data
 - Decoupled requestor and consumer
- Features:
 - No need to have address accompanying the data
 - In-order fill
 - In-order read
 - Cannot do in-place update



Buffet-based Synchronization

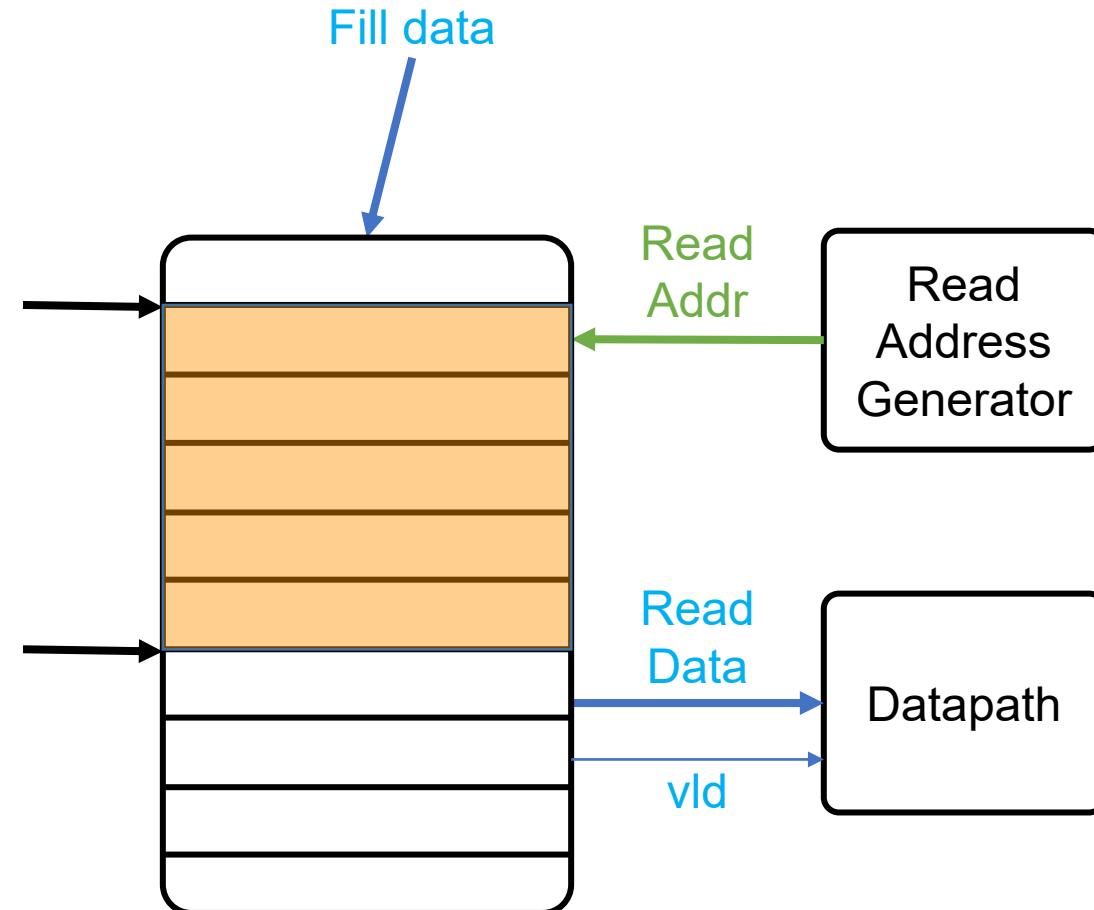
- Allows fine-grained overlap:
 - Fill-read overlap
- FIFO-like head-tail mechanisms together with:
 - Random read
 - In-place update
- Lifetime of staged data:



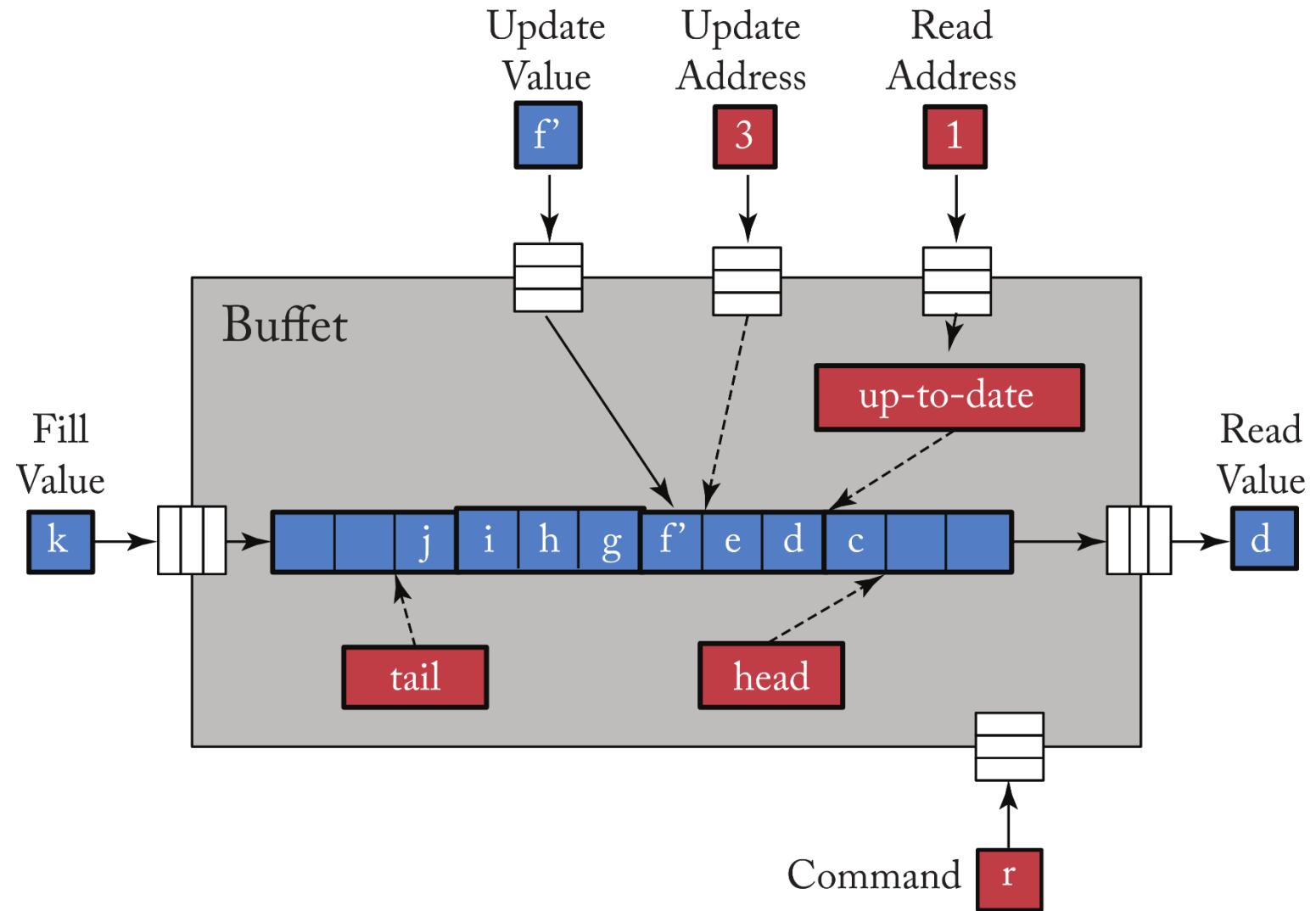
Fill → (Read → Update?)^{} → Read → Shrink*

Buffet-based Synchronization

- Fill-read synchronization

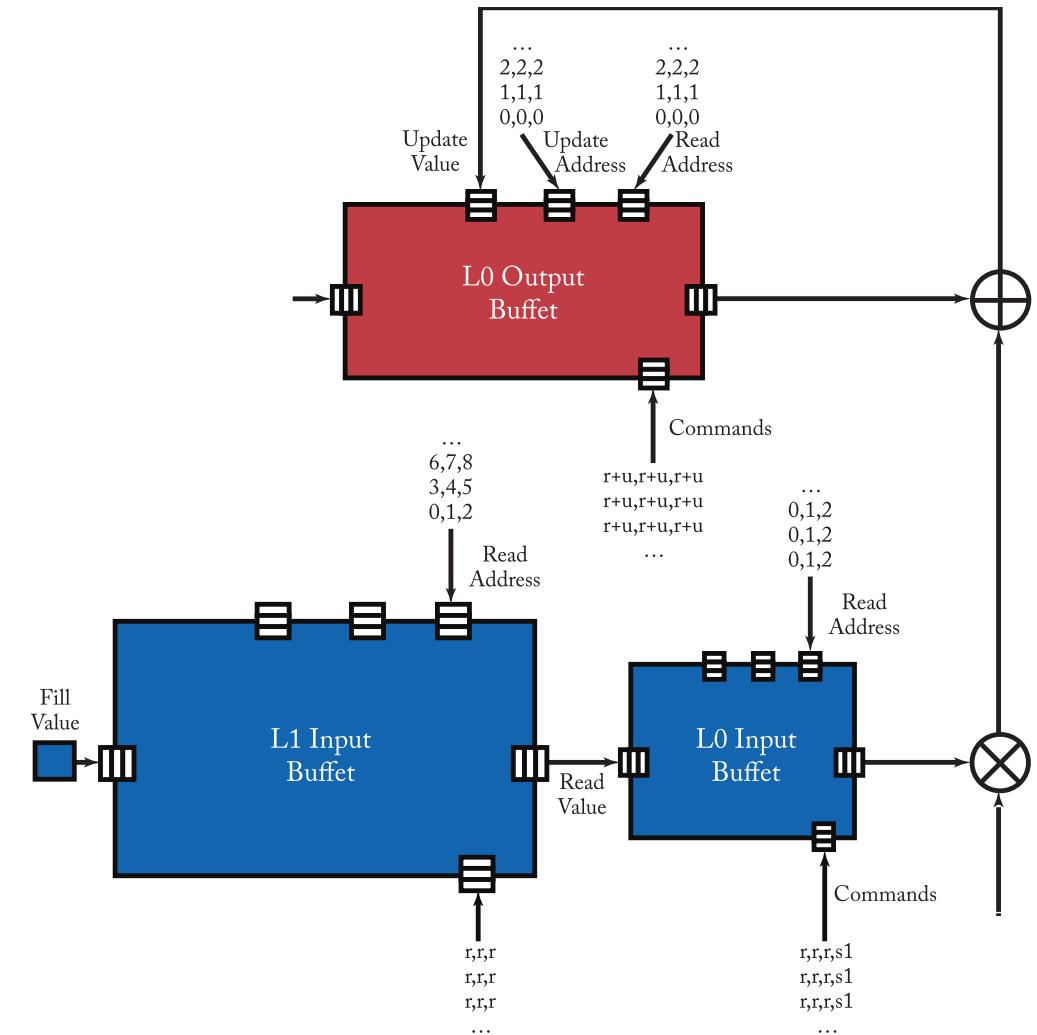


Buffet Block Diagram



Buffet Example

- Eyeriss-like global buffer and PE built with buffets
- Reads to the L1 Input Buffet fill the L0 Buffet
 - Performs reads that pass a sliding window of inputs to the multiplier
- The L0 Output Buffer performs a series of read+update commands to generate the partial sums.



*The weight buffet is not shown.

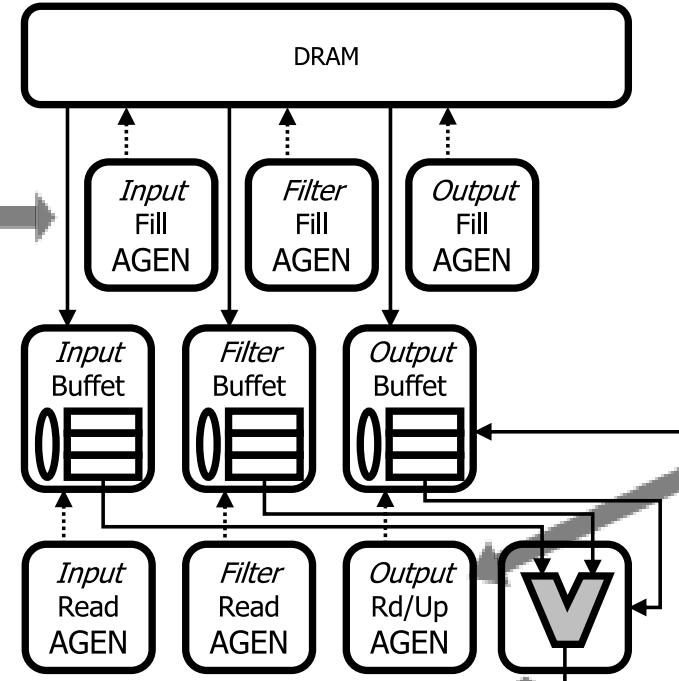
Example of A Buffet-based Accelerator

```

function FILLINPUT()
  halo_size = O_TileSize - F_TileSize + 1;
  for nf : [0..F_NumTiles)
    tile_base = nf * F_TileSize;
    TransferInputTile(tile_base, halo_size);
    for no : [0..O_NumTiles)
      TransferInputTile(tile_base + halo_size +
        no * O_TileSize, O_TileSize);

function FILLCFILTER()
  for nf : [0..F_NumTiles)
    TransferFilterTile(nf * F_TileSize, F_TileSize);

function FILLCOUTPUT()
  for nf : [0..F_NumTiles)
    for no : [0..O_NumTiles)
      TransferOutputTile(no * O_TileSize, O_TileSize);
  
```



```

function READINPUT() ③
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      for to : [0..O_TileSize)
        input_buffet.Read(tf+to);
        // Retain some overlap for next tile
        // (sliding window)
        input_buffet.Shrink(O_TileSize);

function READFILTER()
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      filter_buffet.Read(tf);
      filter_buffet.Shrink(F_TileSize);

function READANDUPDATEOUTPUT()
  for n : [0..F_NumTiles * O_NumTiles)
    for tf : [0..F_TileSize)
      for to : [0..O_TileSize)
        output_buffet.Read(to);
        output_buffet.update_idx_in.
          Send(to);
        // Drain of modified values omitted
        output_buffet.Shrink(O_TileSize);
  
```

All tile transfer functions above follow this form:

```

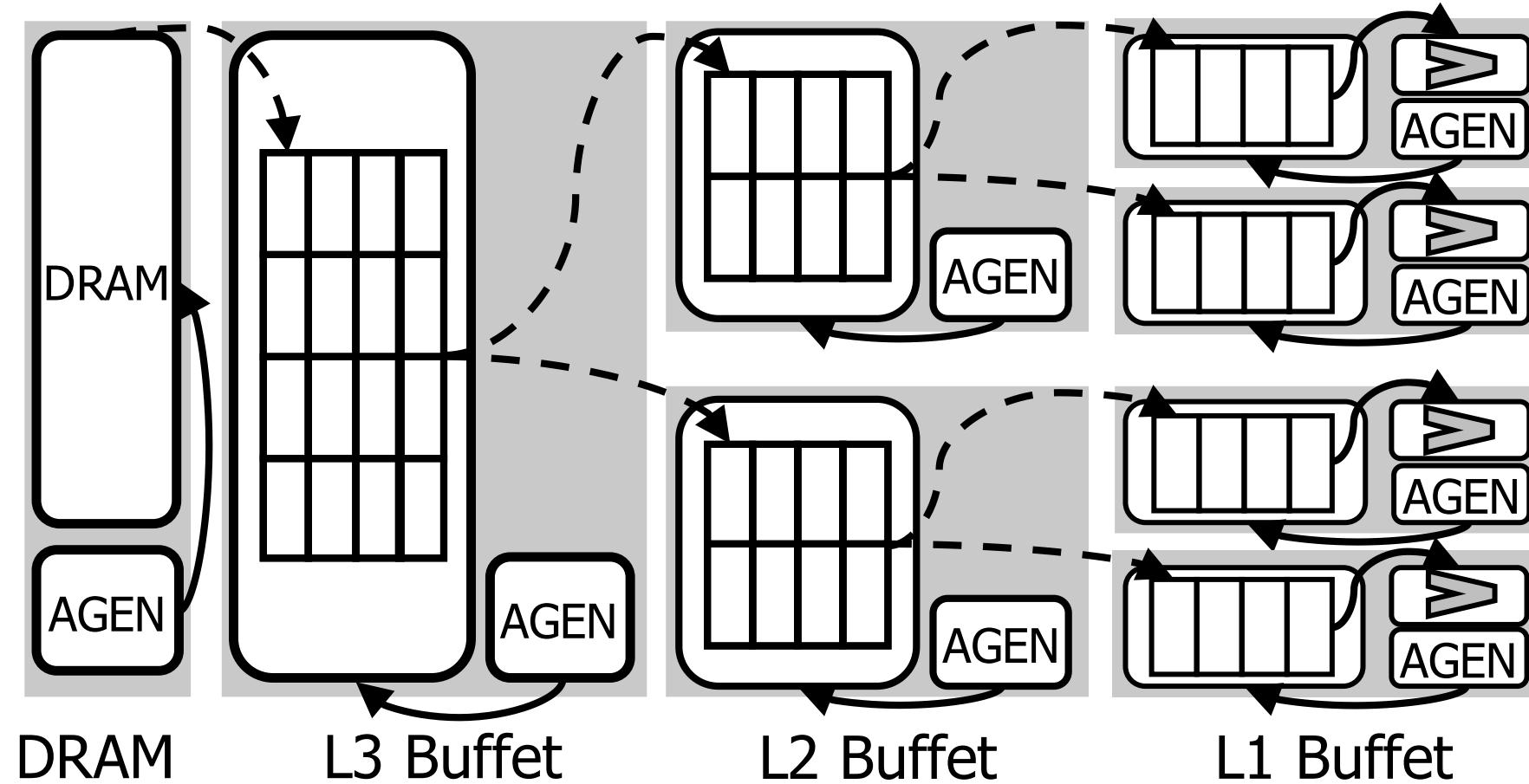
function TRANSFEROUTPUTTILE(base, size) ②
  wait _until(output_credit >= size);
  // This can be implemented as bulk transfer.
  for x : [0..size)
    output_buffet.Fill(output[base+x]);
  output_credit -= size;
  
```

```

function DATAPATH() ④
  inp = input_buffet.read_rsp_out.Recv();
  wt = filter_buffet.read_rsp_out.Recv();
  psum = output_buffet.read_rsp_out.Recv();
  psum += inp * wt;
  output_buffet.update_data_in.Send(psum);
  
```

Example of A Buffet-based Accelerator

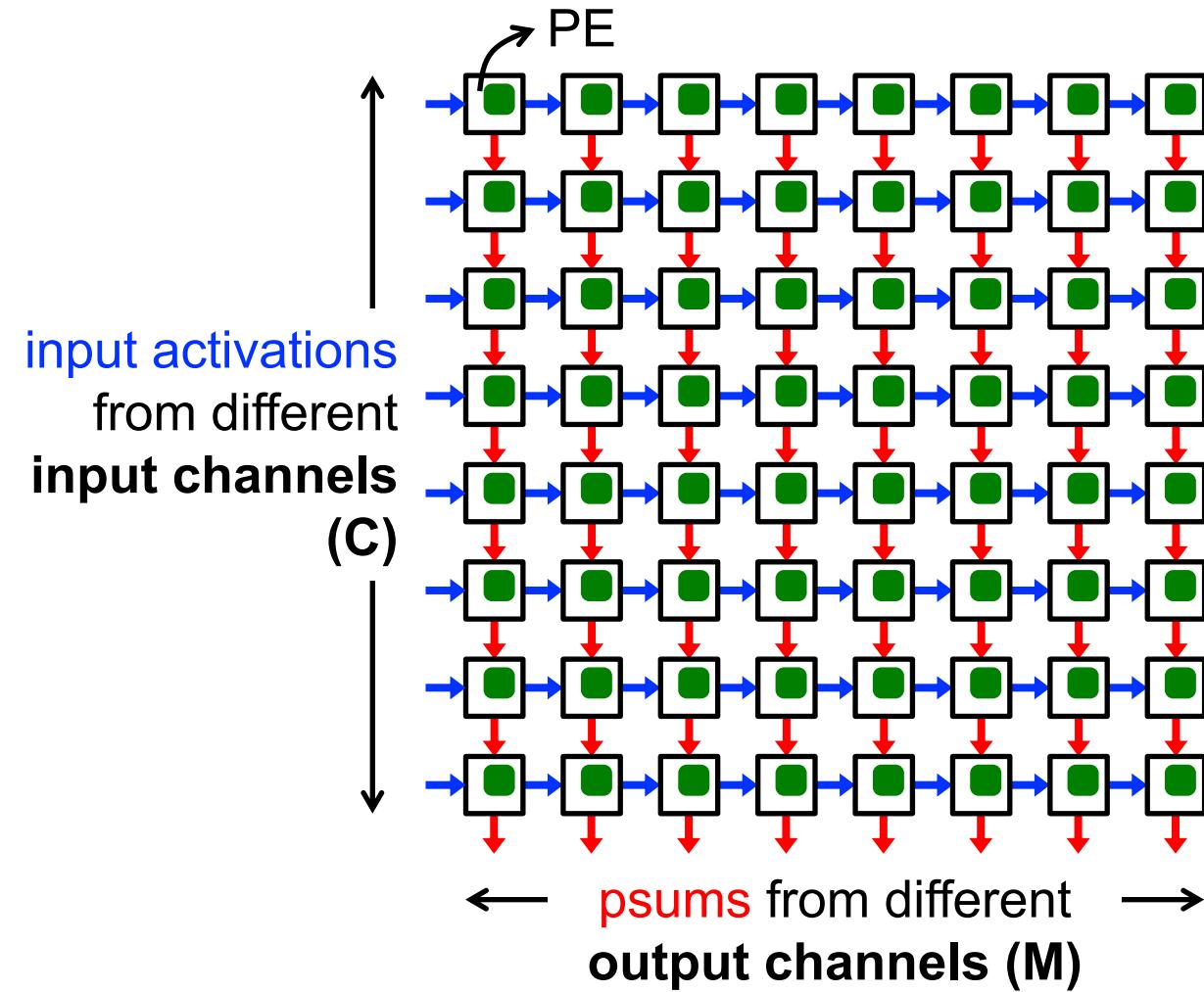
- Read response of Level x becomes the **Fill input to Level $x+1$**



Outline

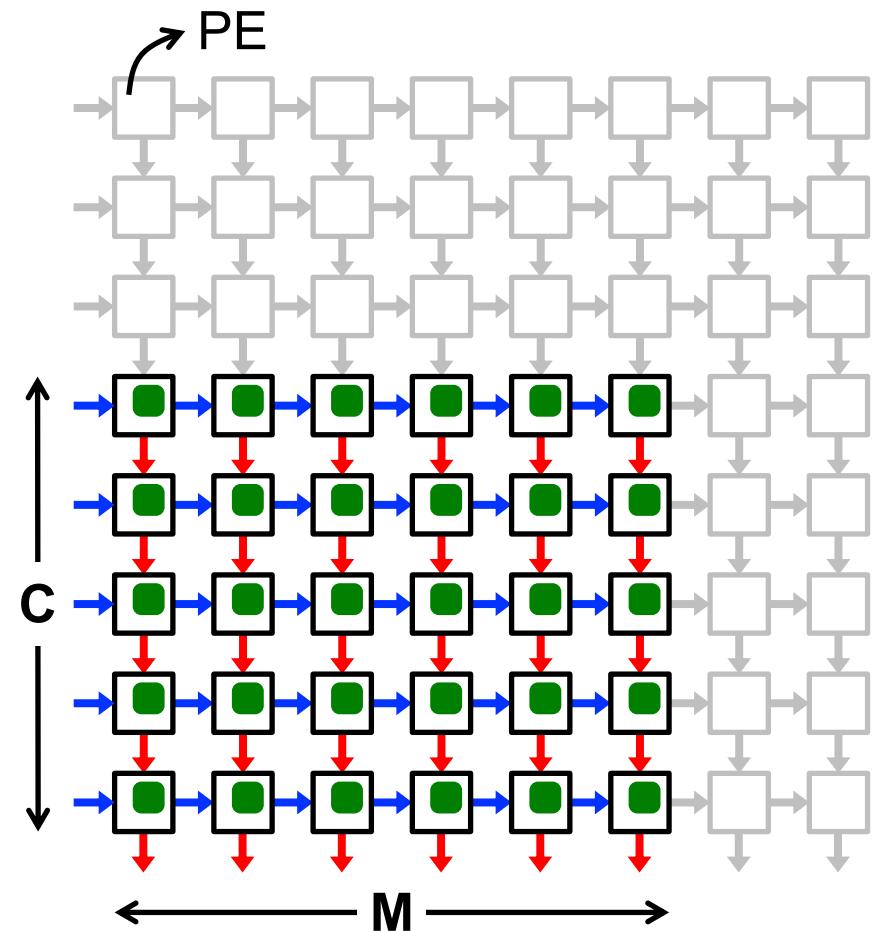
- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
 - Locality and Data Reuse
 - Dataflow Taxonomy
 - Data Orchestration
 - Network-on-Chip
 - Optimization
- Roofline Model
- Energy

A Common Design Pattern



A Common Design Pattern

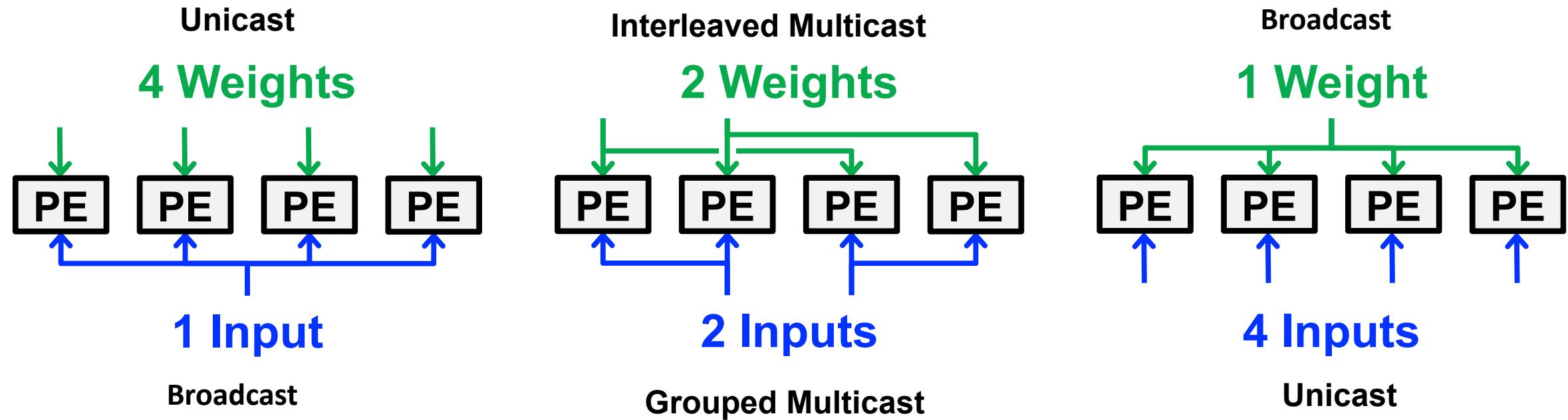
- PE array underutilized If there are fewer input and/or output channels than the array dimensions
- Effective data delivery BW also becomes lower
 - → further impact performance
- Not scalable
 - → utilization will be worse at larger scales



A More Flexible Data Delivery Strategy



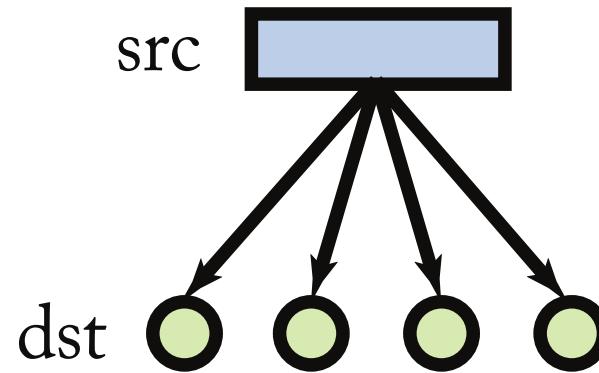
- Adapt to the reuse and bandwidth requirements



Common NoC Design



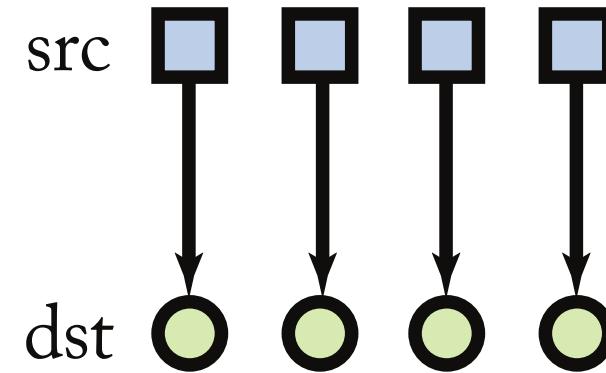
Broadcast Network



High Reuse

Low Bandwidth

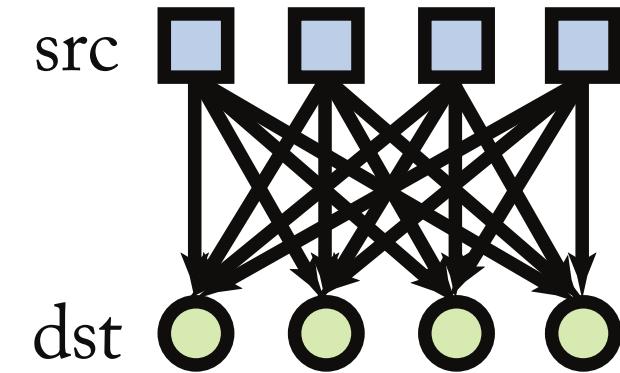
Unicast Network



Low Reuse

High Bandwidth

All-to-All Network

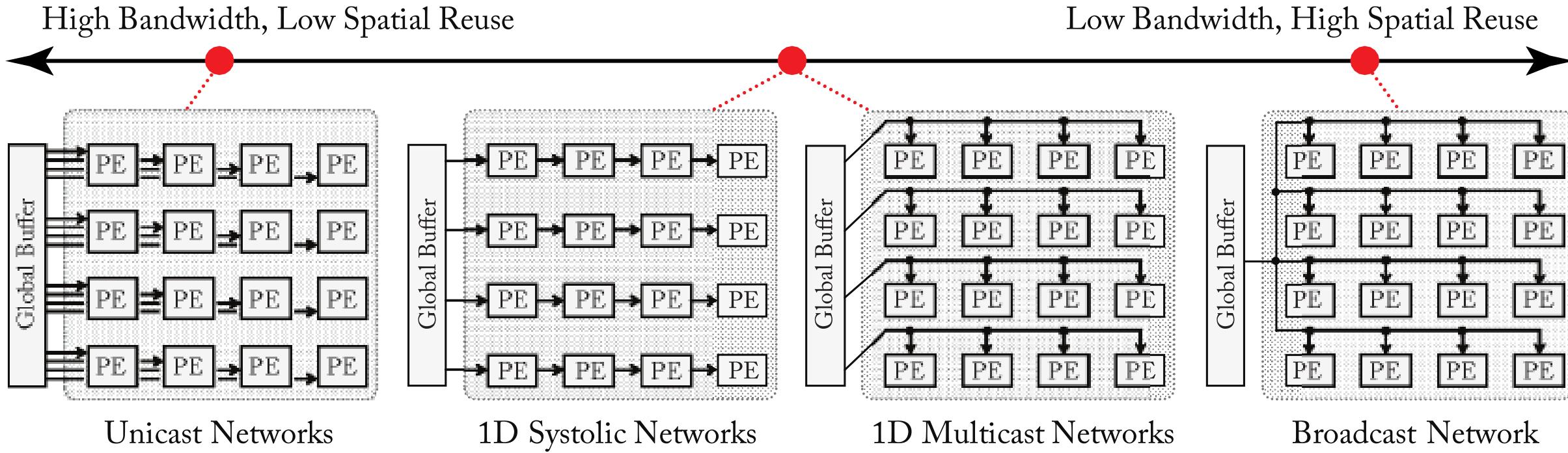


High Reuse

High Bandwidth

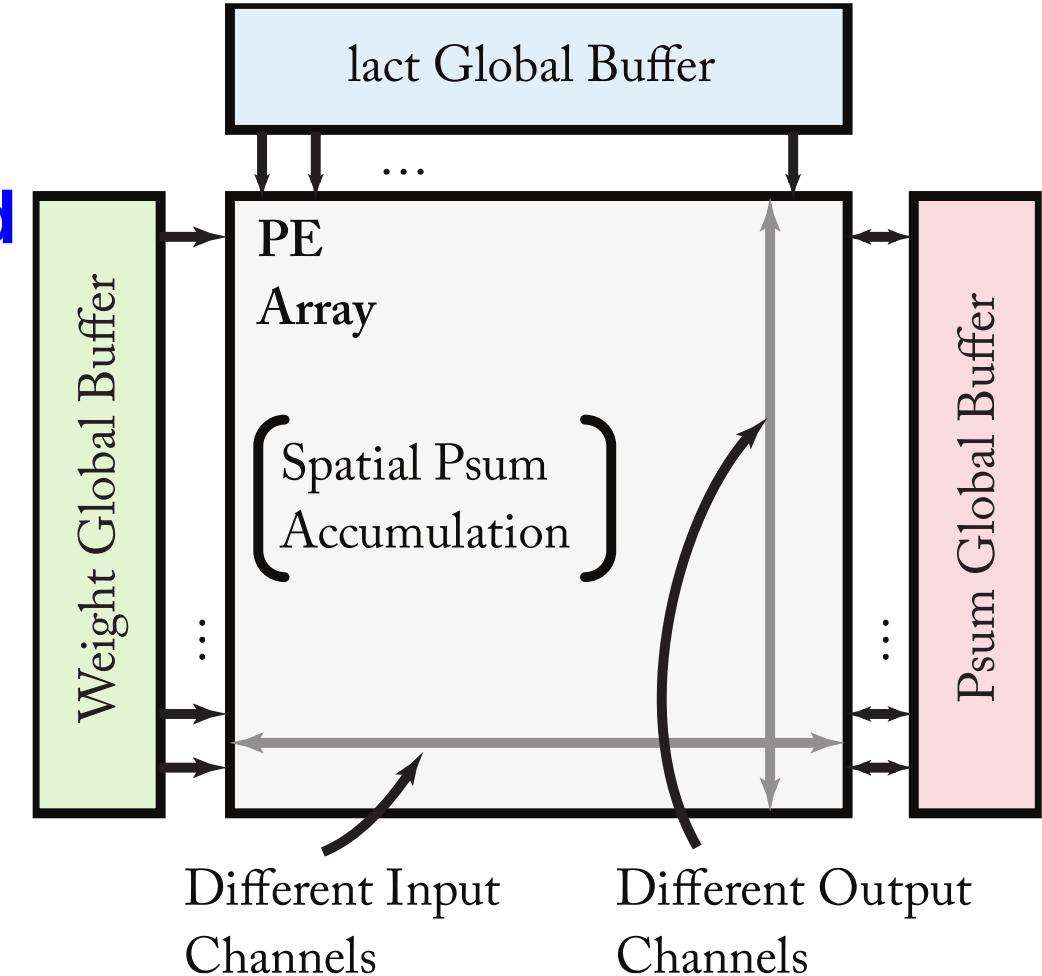
Hard to Scale

Common NoC Designs on DNNs



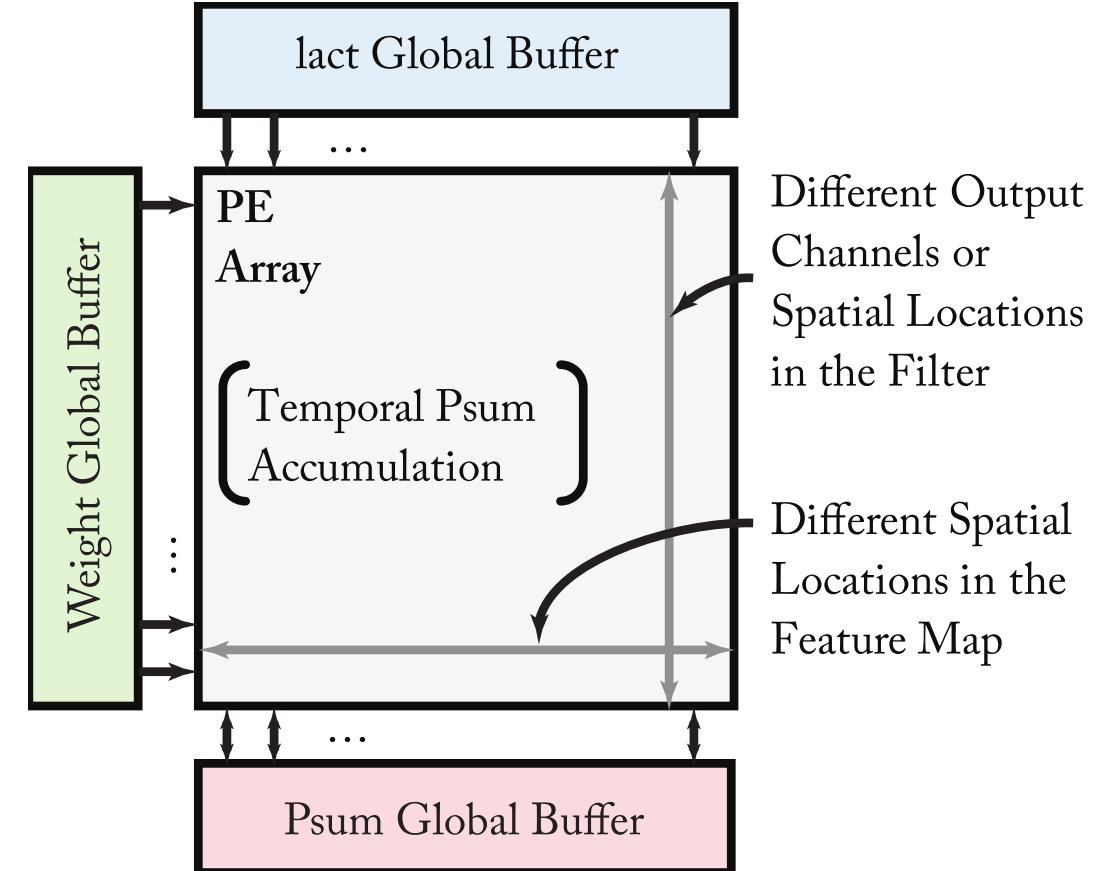
Spatial Accumulation DNN Accelerator Design

- **Input activations** (lacts) are **reused vertically**
- **Partial sums** (Psum) are **accumulated horizontally**
- Often used for a **weight-stationary** dataflow

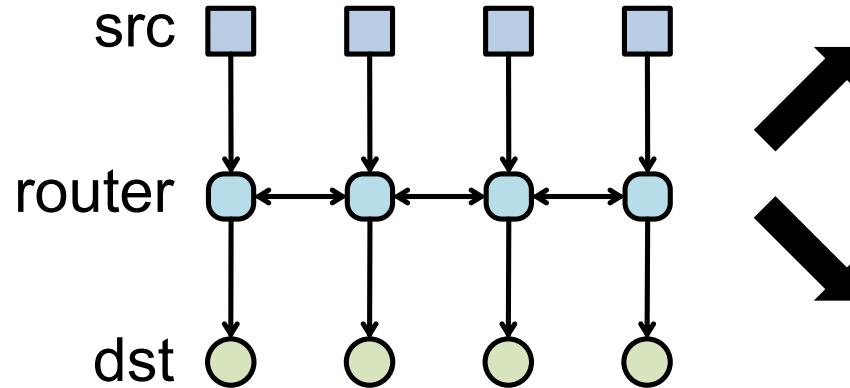


Temporal Accumulation DNN Accelerator Design

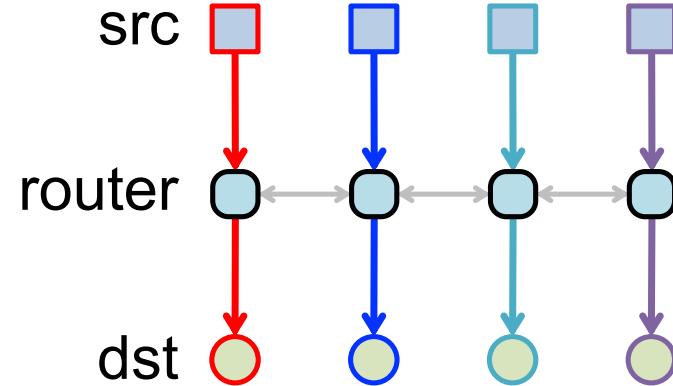
- **Input activations** (lacts) are **reused vertically**
- **Weights** are **reused horizontally**
- Often used for an **output-stationary** dataflow



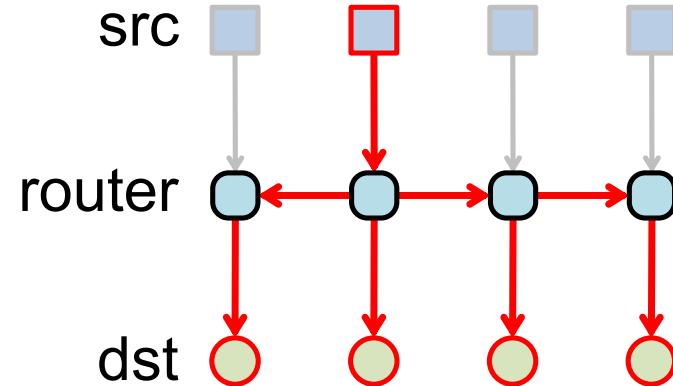
Mesh Network – Best of Both Worlds



High-Bandwidth Mode



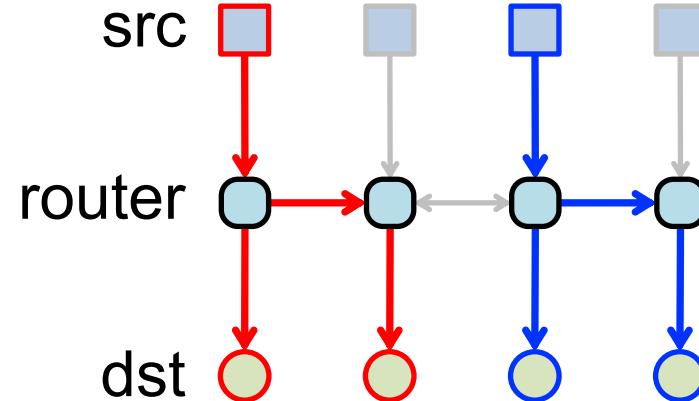
High-Reuse Mode



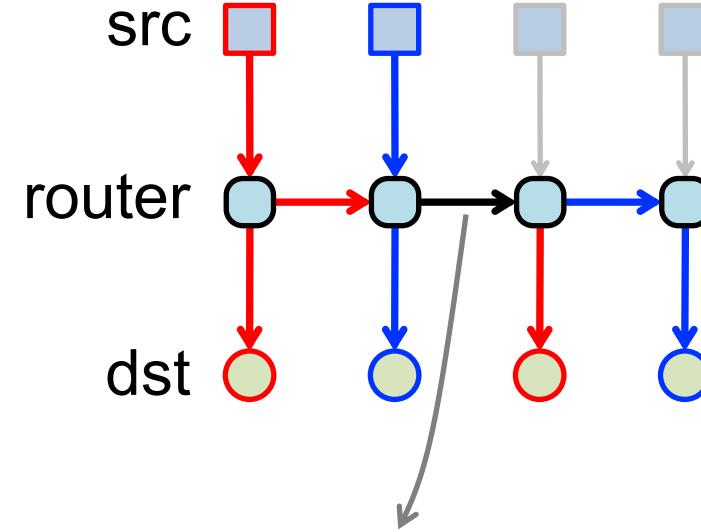
Mesh Network – More Complicated Cases



Grouped-Multicast Mode



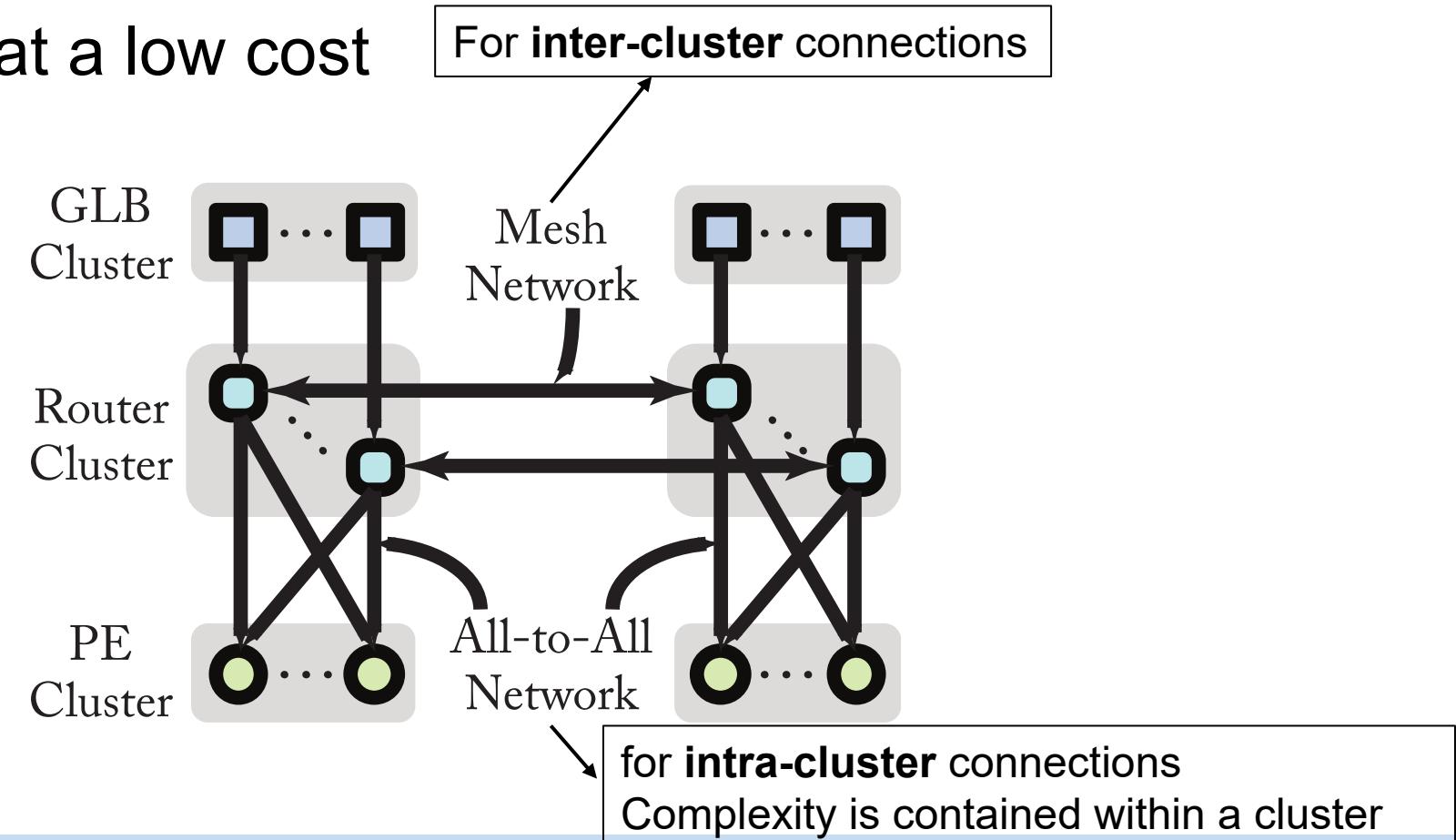
Interleaved-Multicast Mode



Bandwidth-limited route
(flow control required)

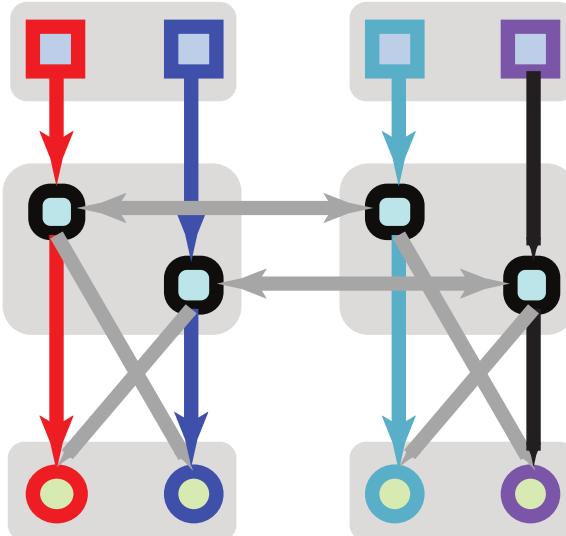
Hierarchical Mesh Network

- Flexible to support patterns ranging from high reuse to high bandwidth scenarios
- Can be easily scaled at a low cost

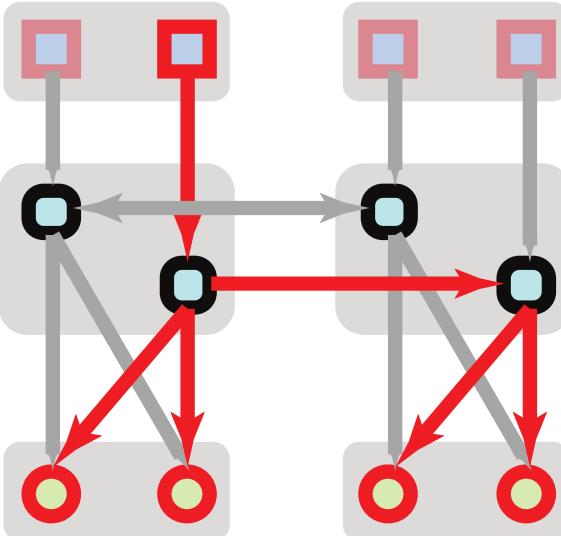


Different Operating Modes

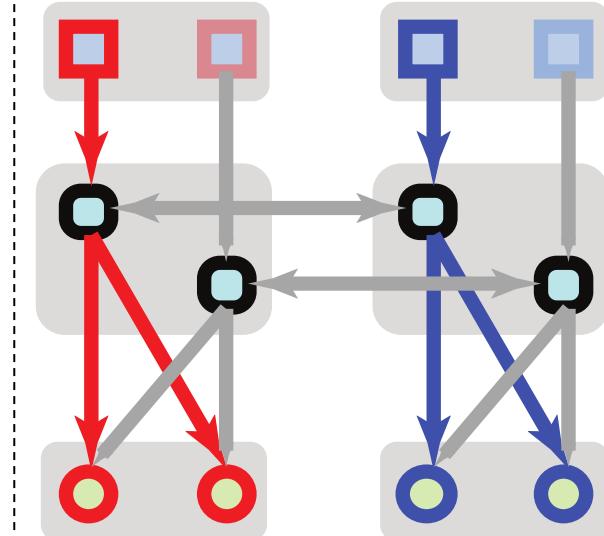
High-Bandwidth Mode



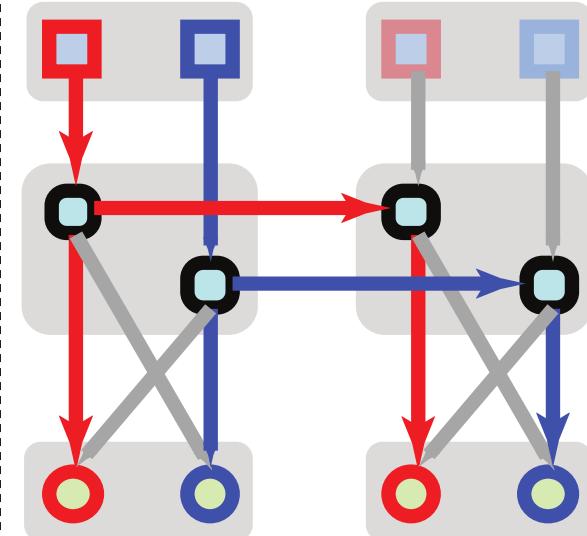
High-Reuse Mode



Grouped-Multicast Mode

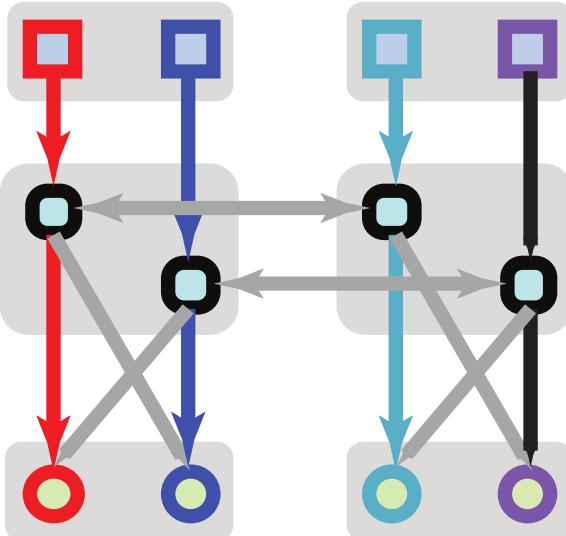


Interleaved-Multicast Mode

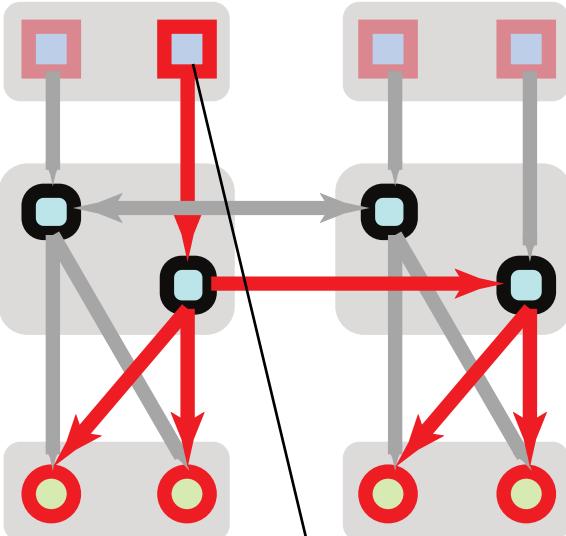


Different Operating Modes

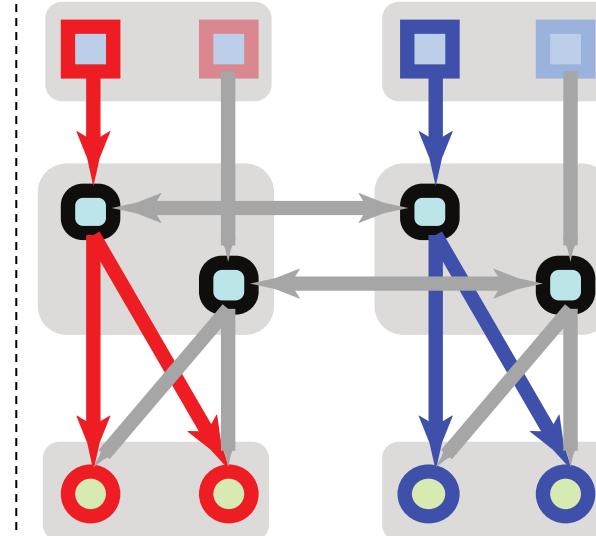
High-Bandwidth Mode



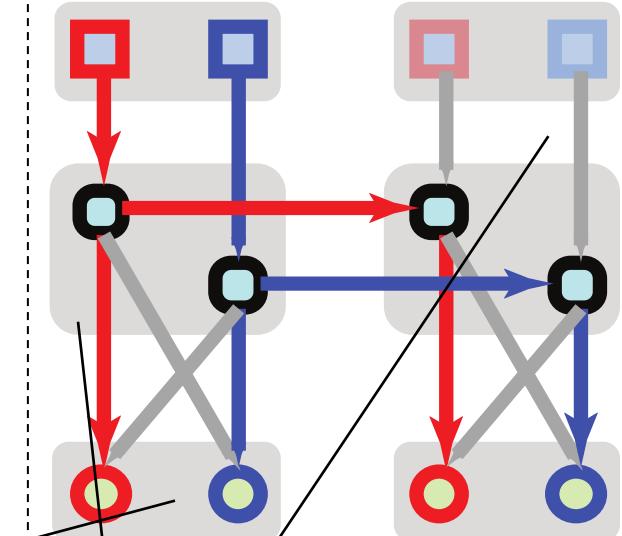
High-Reuse Mode



Grouped-Multicast Mode



Interleaved-Multicast Mode



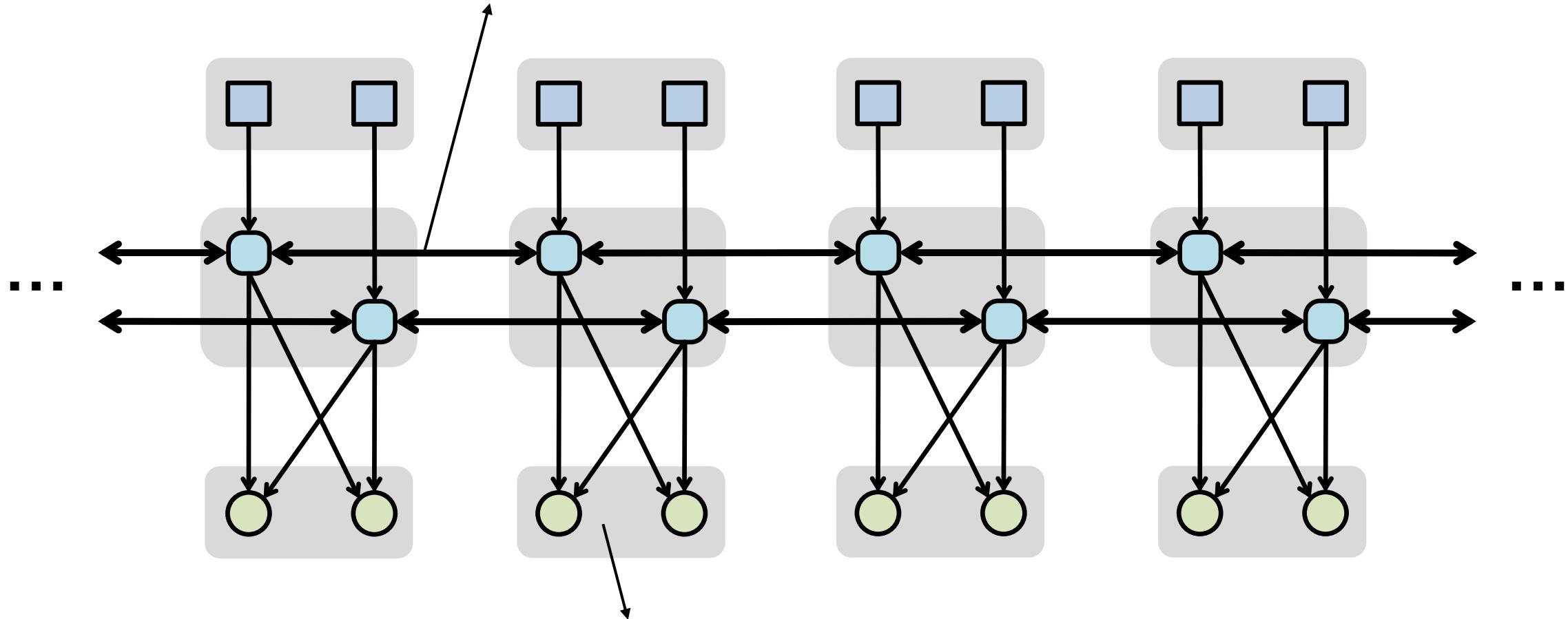
from any one src

Can interleave more by scaling up the cluster size

All routes are determined at configuration time
 → **Routers are circuit-switched** (only MUXes)

Scaling the Hierarchical Mesh Network

Cost of the mesh network scales **linearly**

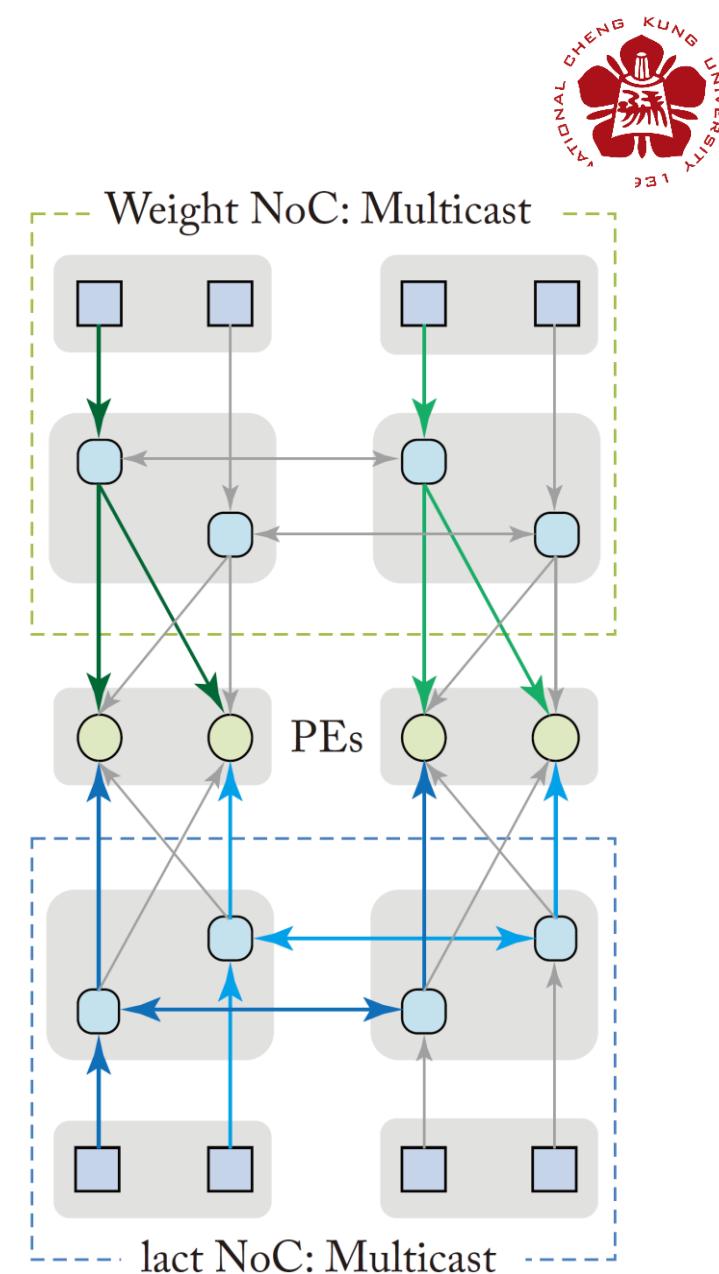


Cluster size can scale up depending on
tolerable cost of the all-to-all networks

Hierarchical Mesh Networks

CONV Layer

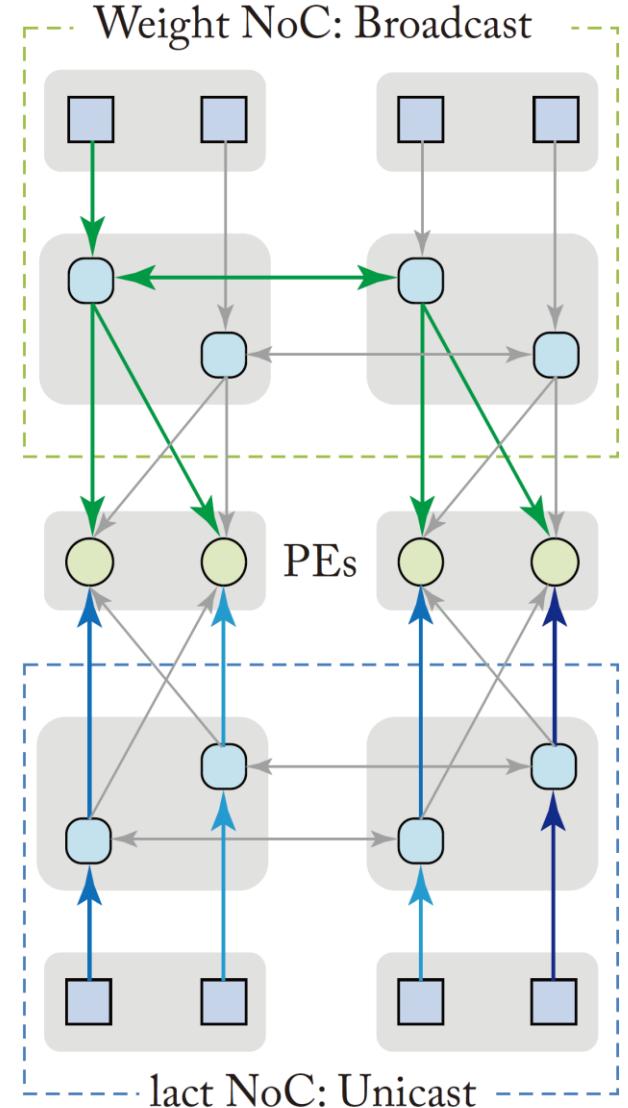
- Keep all four PE the lowest bandwidth requirement
 - 2 input activations
 - 2 weight
- HM-NoC Configurations (2 options)
 1. **Grouped-multicast** for **input activations**
Interleaved-multicast for **weights**
 2. **Grouped-multicast** for **weights**
Interleaved-multicast for **input activations**



Hierarchical Mesh Networks

Depth-wise CONV Layer

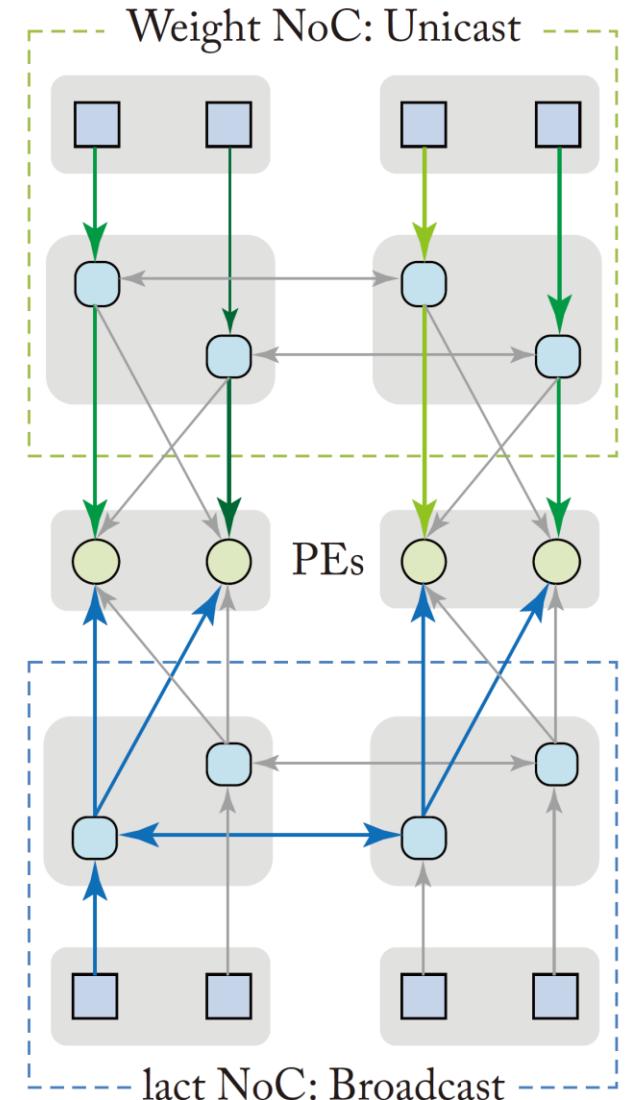
- Nearly no reuse for input activation
- Exploit the reuse of weights
- HM-NoC Configurations
 - **Broadcasting the weights** to all PEs
 - **Fetching unique input activation** for each PE



Hierarchical Mesh Networks

Fully-Connected Layer

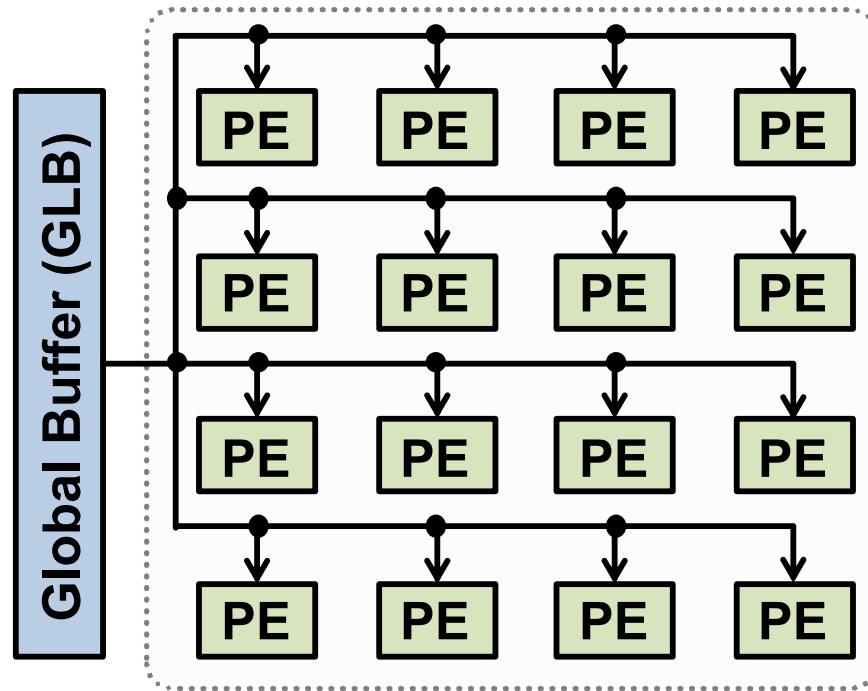
- Little reuse for weights
 - Especially when the batch size is limited
- HM-NoC Configurations
 - **Broadcast Input activations** to all PEs
 - **Unicast Weights** to the PEs



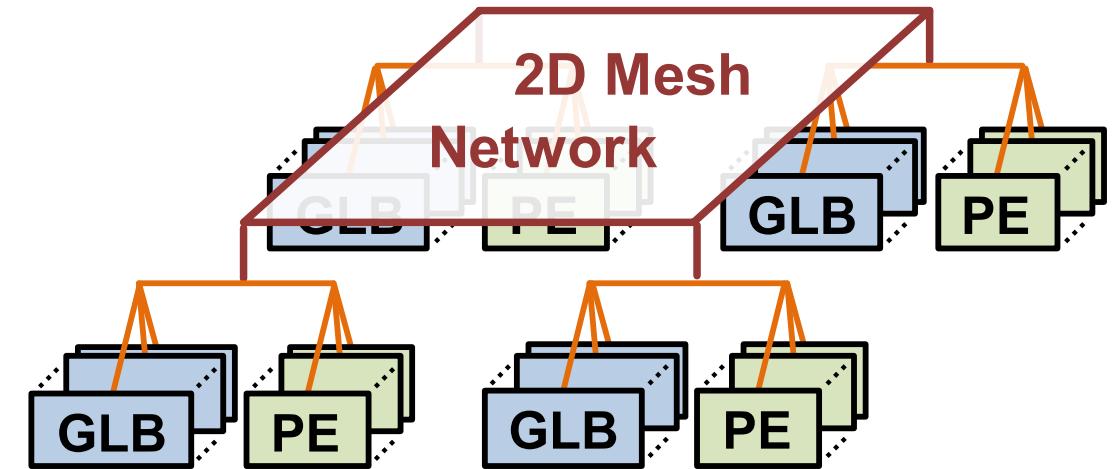
Eyeriss with Hierarchical Mesh Network



Eyeriss v1



Eyeriss v2



Outline

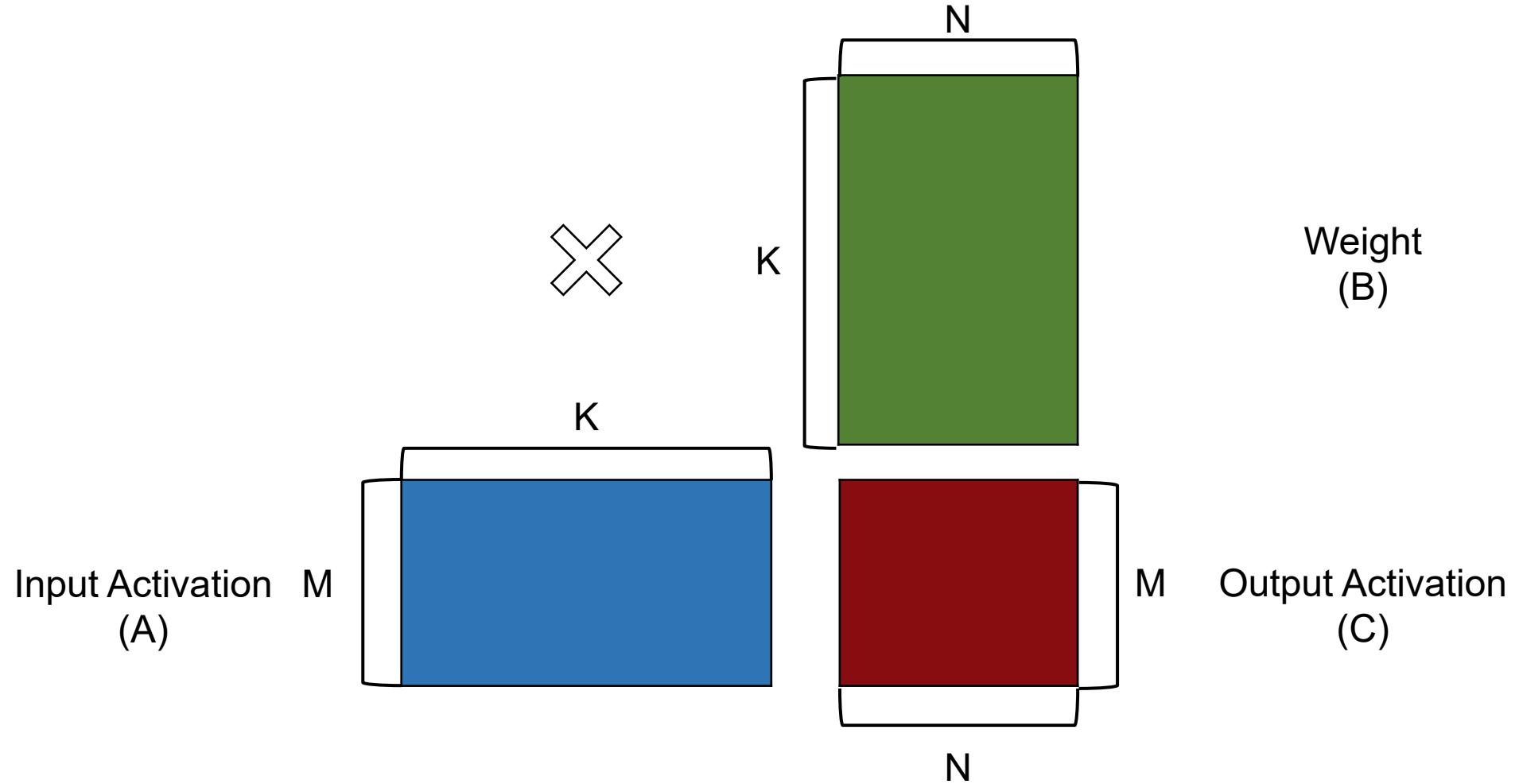
- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
 - Locality and Data Reuse
 - Dataflow Taxonomy
 - Data Orchestration
 - Network-on-Chip
 - Optimization
- Roofline Model
- Energy

Datapath Optimization

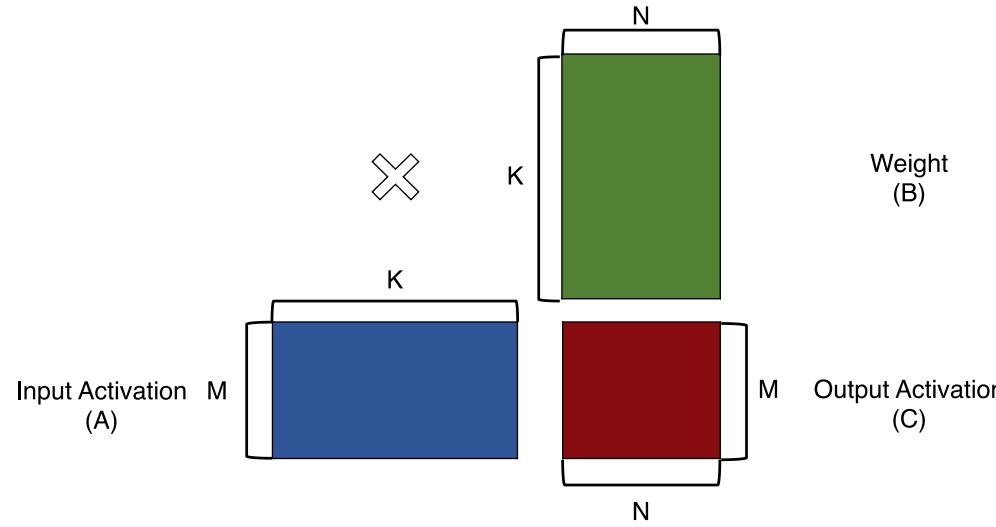


- **spatial_for** - to explore parallelism
 - Spatial-K: accumulation dimension
 - Adder tree/dot product
 - Systolic accumulation
 - Spatial-M/N: data reuse dimension
 - Direct-wiring multicast
 - Systolic multicast
- State-of-the-art accelerators typically use a combination of both at different levels of the hierarchy.

Matrix Multiply



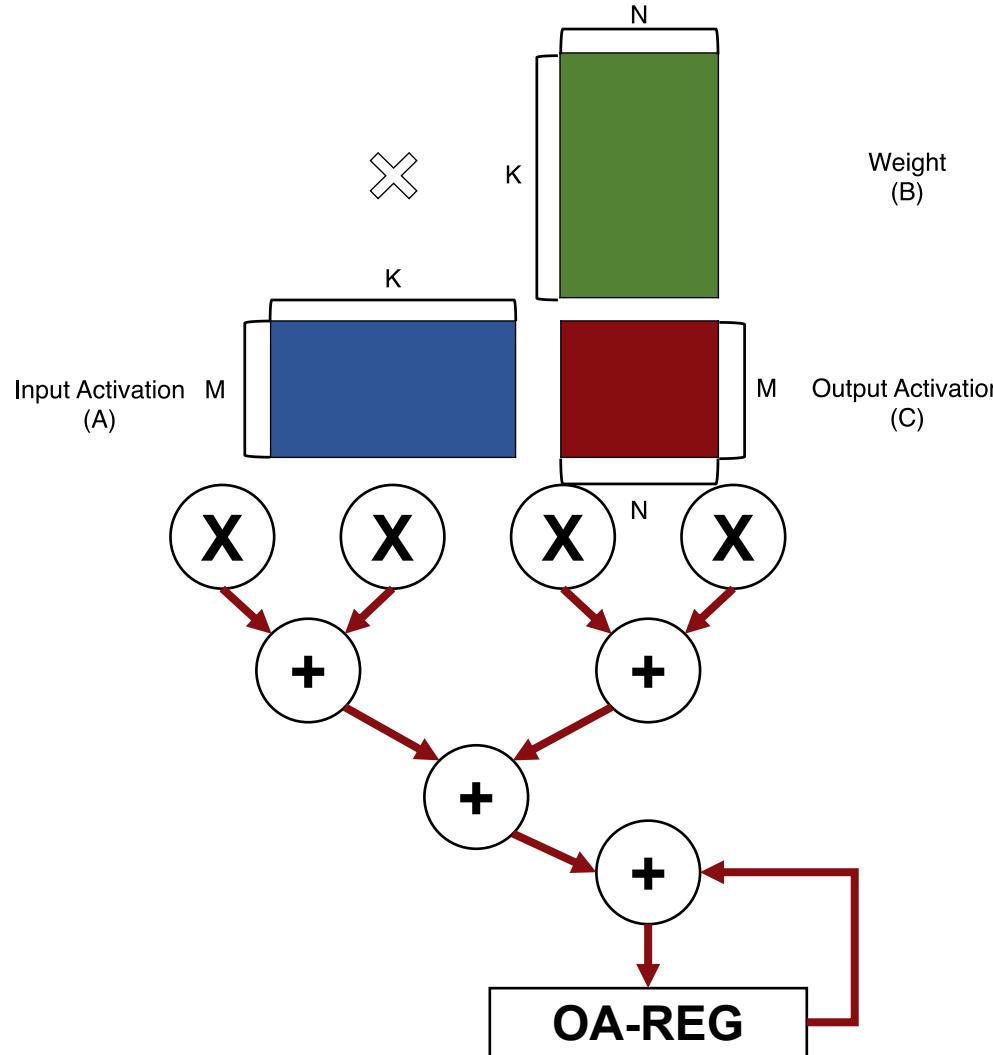
Matrix Multiply



```

for (m=0; m<M; m++) {
    for (n=0; n<N; n++) {
        OA[n,m] = 0;
        for (k=0; k<K; k++) {
            OA[n,m] += IA[m, k] * W[k, n];
        }
        OA[n,m] = Activation(OA[n,m]);
    }
}
  
```

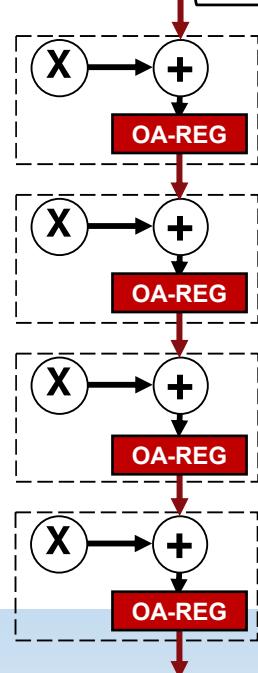
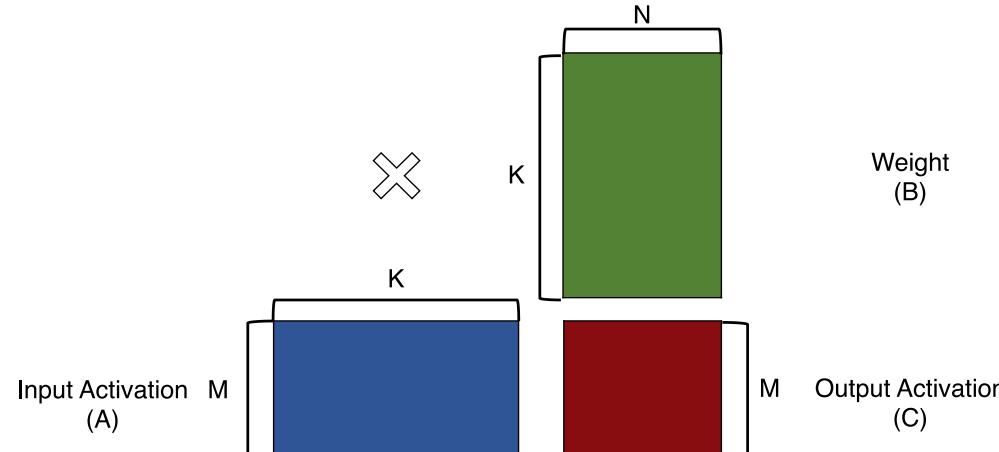
Datapath Optimization - Spatial-K



```
for (m=0; m<M; m++) {  
    for (n=0; n<N; n++) {  
        OA[n,m] = 0;  
        spatial_for (k=0; k<K; k++) {  
            OA[n,m] += IA[m, k] * W[k, n];  
        }  
        OA[n,m] = Activation(OA[n,m]);  
    }  
}
```

- Type 1: Adder Tree
- Example: NVDLA, DianNao
- Typical width: 8-64
- Applicable to any accumulation dimensions
 - E.g., R, S, C in convolution

Datapath Optimization - Spatial-K

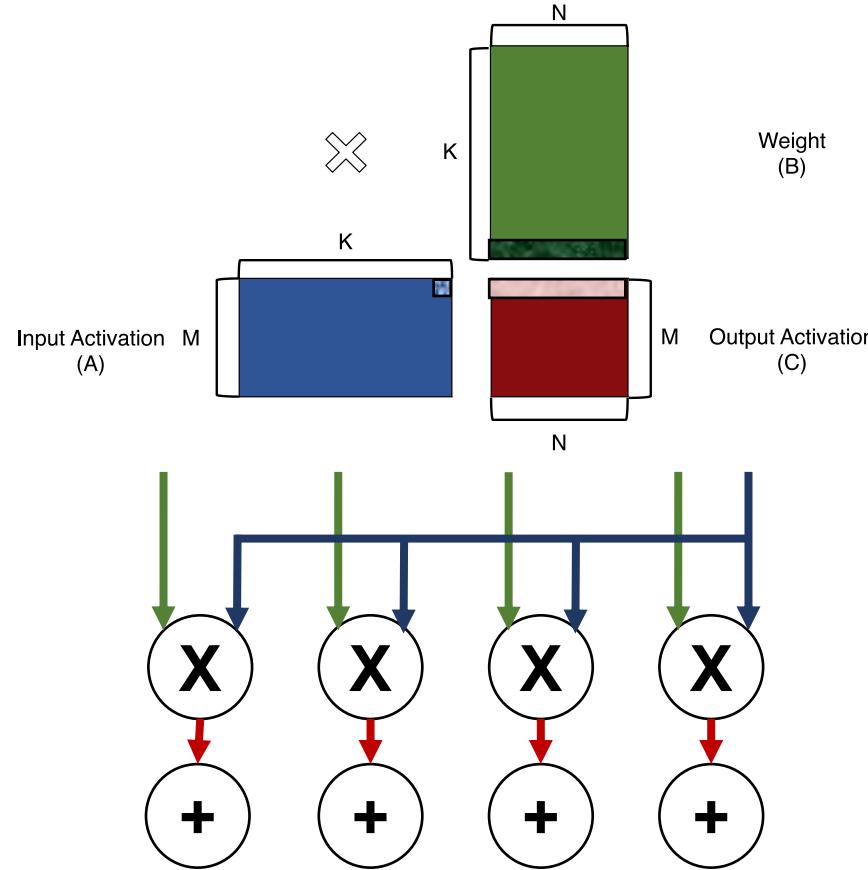


```

for (m=0; m<M; m++) {
    for (n=0; n<N; n++) {
        OA[n,m] = 0;
        spatial_for (k=0; k<K; k++) {
            OA[n,m] += IA[m, k] * W[k, n];
        }
        OA[n,m] = Activation(OA[n,m]);
    }
}
  
```

- Type 2: Systolic Accumulation
- Example: TPU, Gemmini
- Typical width: 8-256
- Applicable to any accumulation dimensions
 - E.g., R, S, C in convolution

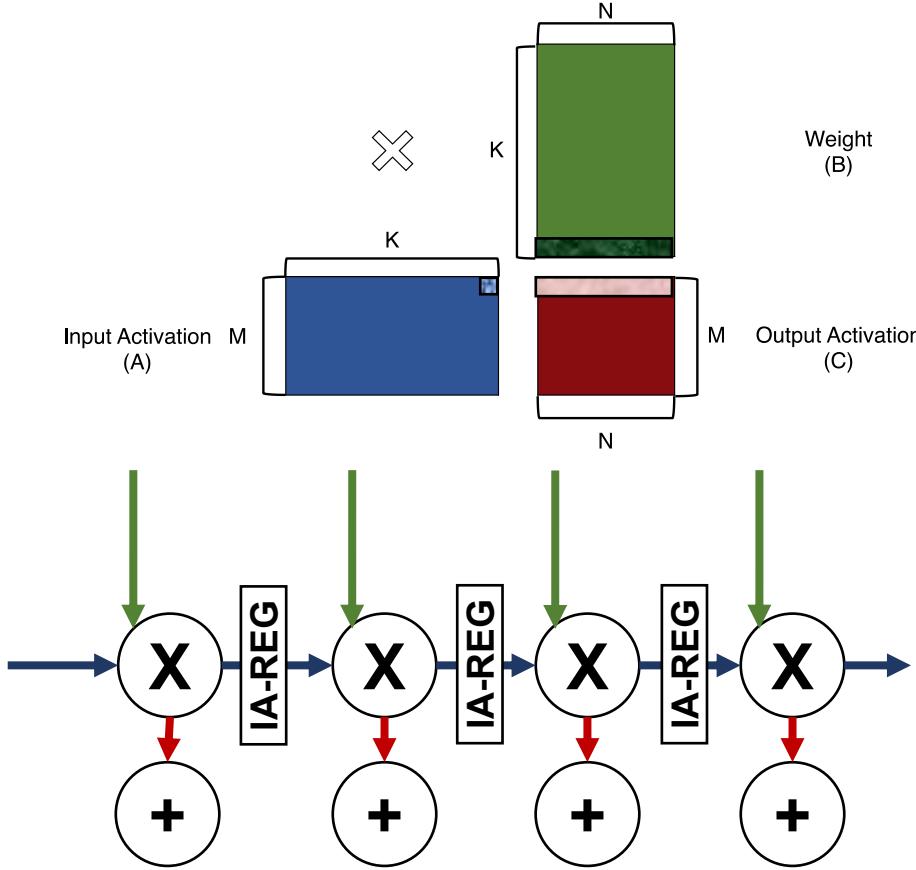
Datapath Optimization - Spatial-N



```
for (m=0; m<M; m++) {  
    spatial_for (n=0; n<N; n++) {  
        OA[n,m] = 0;  
        for (k=0; k<K; k++) {  
            OA[n,m] += IA[m, k] * W[k, n];  
        }  
        OA[n,m] = Activation(OA[n,m]);  
    }  
}
```

- Type 1: Direct-wiring multicast
- Example: NVDLA, DianNao
- Typical width: 8-16
- Applicable to any non-accumulation dimensions

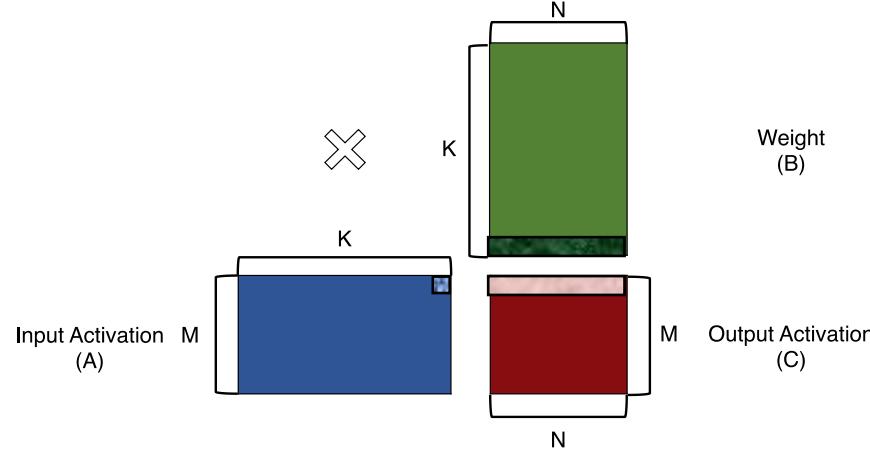
Datapath Optimization - Spatial-N



```
for (m=0; m<M; m++) {  
    spatial_for (n=0; n<N; n++) {  
        OA[n,m] = 0;  
        for (k=0; k<K; k++) {  
            OA[n,m] += IA[m, k] * W[k, n];  
        }  
        OA[n,m] = Activation(OA[n,m]);  
    }  
}
```

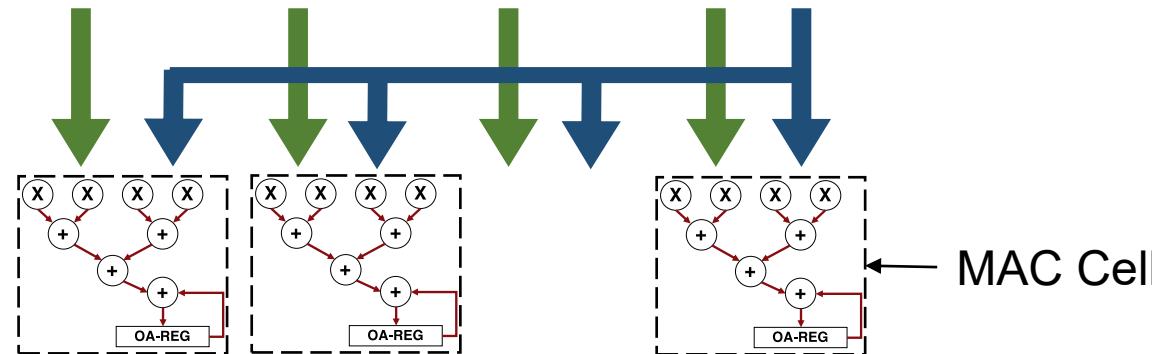
- Type 2: Systolic multicast
- Example: TPU, Gemmini
- Typical width: 8-256
- Applicable to any non-accumulation dimensions

Datapath Optimization Combined: NVDLA



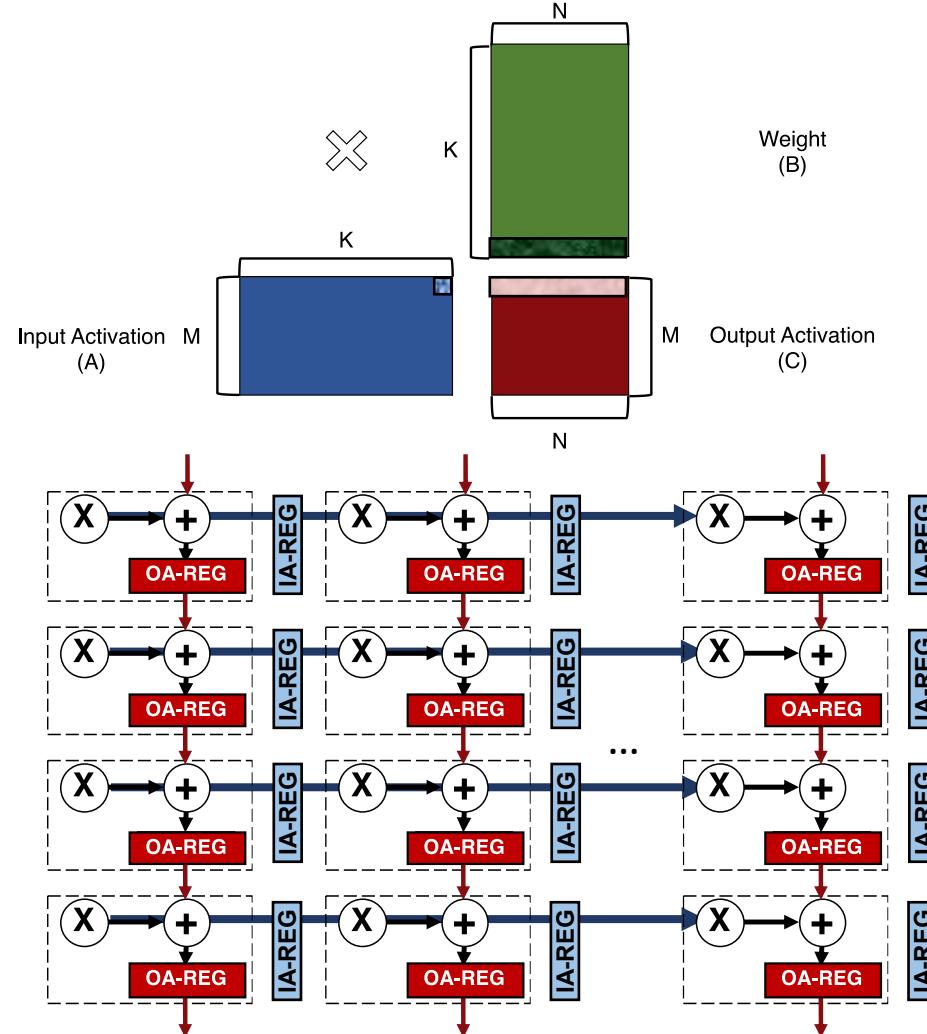
```

for (m=0; m<M; m++) {
    spatial_for (n=0; n<N; n++) {
        OA[n,m] = 0;
        spatial_for (k=0; k<K; k++) {
            OA[n,m] += IA[m, k] * W[k, n];
        }
        OA[n,m] = Activation(OA[n,m]);
    }
}
  
```



- Adder-tree accumulation
- Direct-wiring multicast

Datapath Optimization Combined: TPU



```

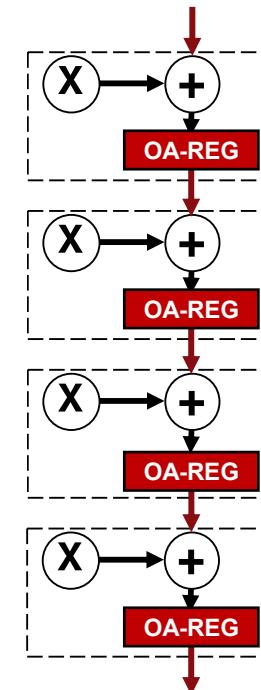
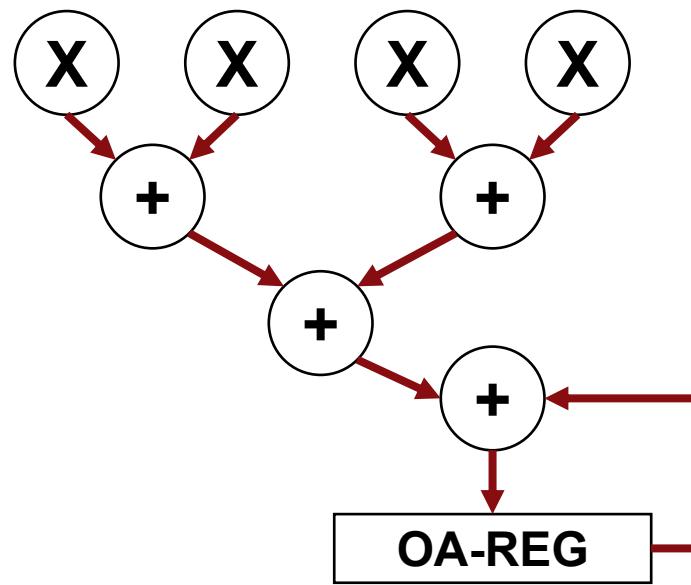
for (m=0; m<M; m++) {
    spatial_for (n=0; n<N; n++) {
        OA[n,m] = 0;
        spatial_for (k=0; k<K; k++) {
            OA[n,m] += IA[m, k] * W[k, n];
        }
        OA[n,m] = Activation(OA[n,m]);
    }
}
  
```

- Systolic accumulation
- Systolic multicast

Memory Optimization

Short-lived intermediate results

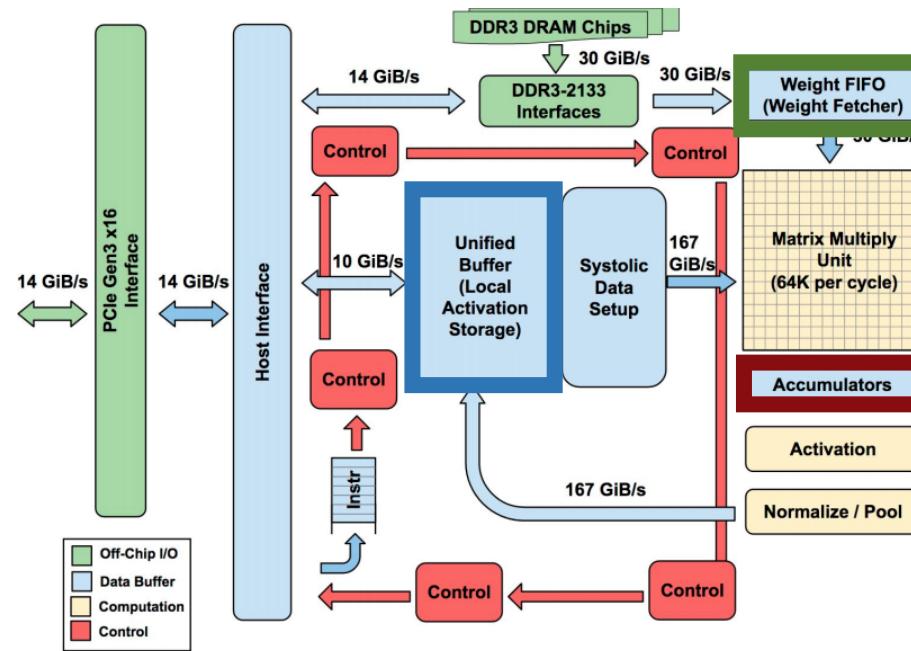
- Instead of accessing SRAM or a shared, large register file
 - Consume the intermediate results directly
- Example: adder tree and systolic accumulation



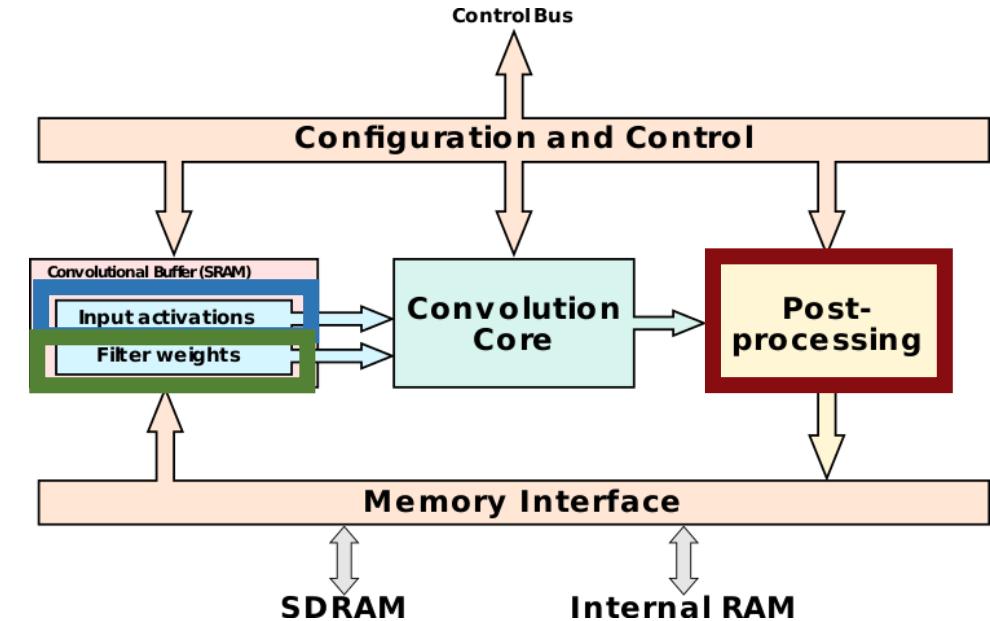
Memory Optimization

App-specific storage size/BW

- Dedicated storage for each operand



TPU



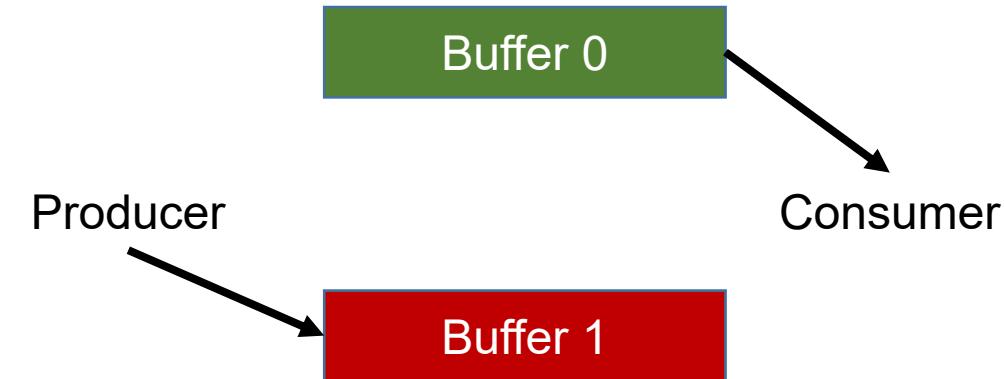
NVDLA

Memory Optimization



App-specific data delivery

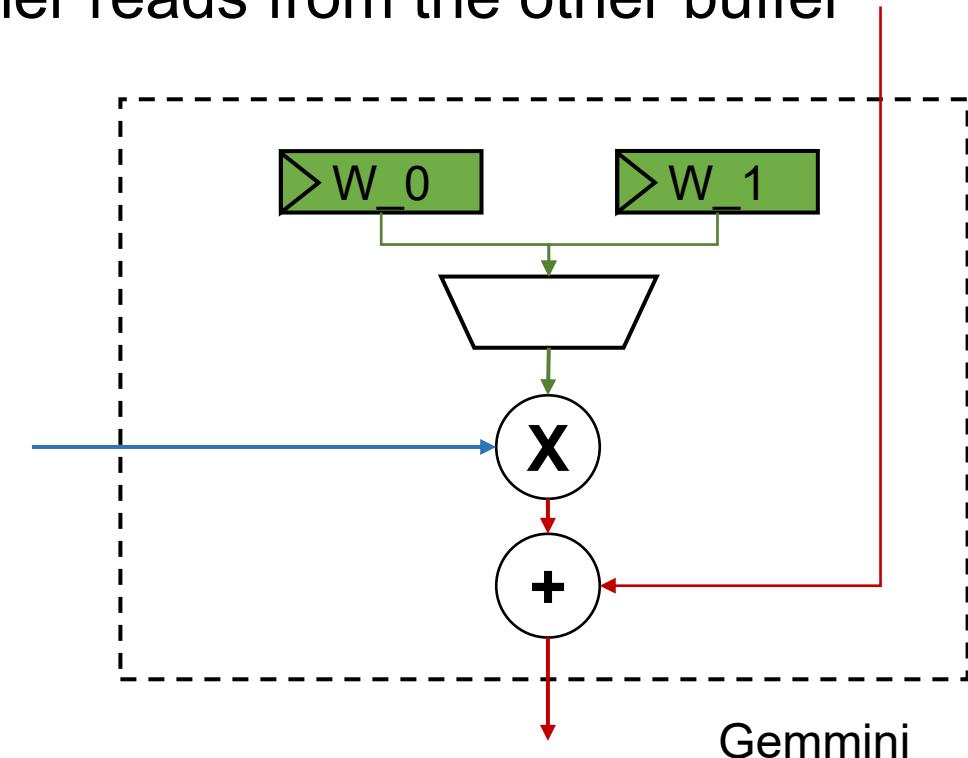
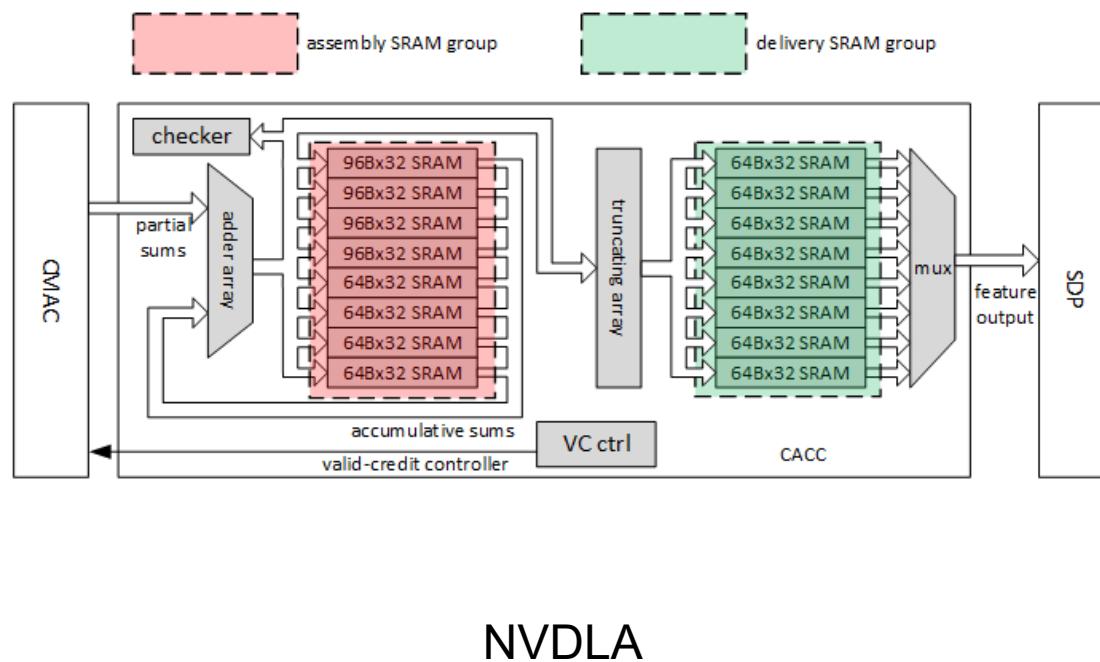
- Double-buffering
 - Goal: Overlap compute and communication by doubling the SRAM sizes.
 - Producer writes to a buffer while Consumer reads from the other buffer



Memory Optimization

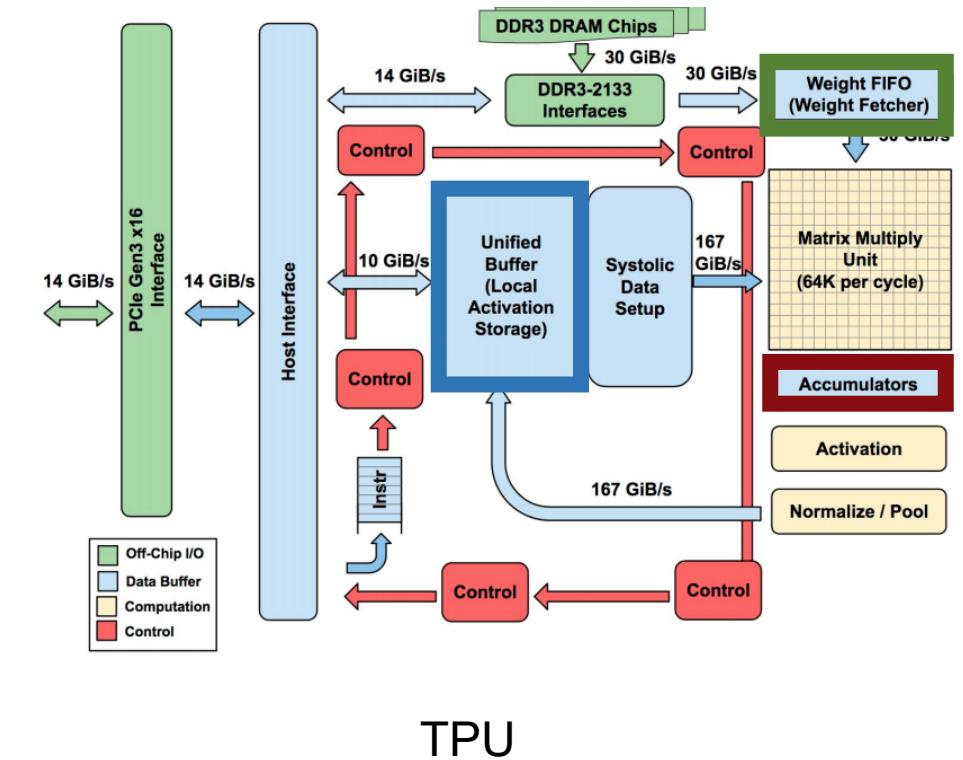
App-specific data delivery

- Double-buffering
 - Goal: Overlap compute and communication by doubling the SRAM sizes.
 - Producer writes to a buffer while Consumer reads from the other buffer



Tensor Processing Unit

- Datapath:
 - Spatial-K: Systolic Accumulation
 - Multi-cycle with registers
 - Spatial-N: Systolic multicast
 - Multi-cycle with registers
 - Better scalability
- Memory
 - Custom systolic registers
 - Dedicated accumulation and weight buffers
 - Double-buffered, weight-stationary dataflow



Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
- Roofline Model
- Energy

Performance Models / Simulators

- Historically, many performance models and simulators **tracked latencies to predict performance** (i.e. counting cycles)
- The last two decades saw a number of **latency-hiding techniques**
 - Out-of-order execution (hardware discovers parallelism to hide latency)
 - HW stream prefetching (hardware speculatively loads data)
 - Massive thread parallelism (independent threads satisfy the latency-bandwidth product)
- **Effectively latency hiding** has resulted in a shift **from a latency-limited computing regime to a throughput-limited computing regime**

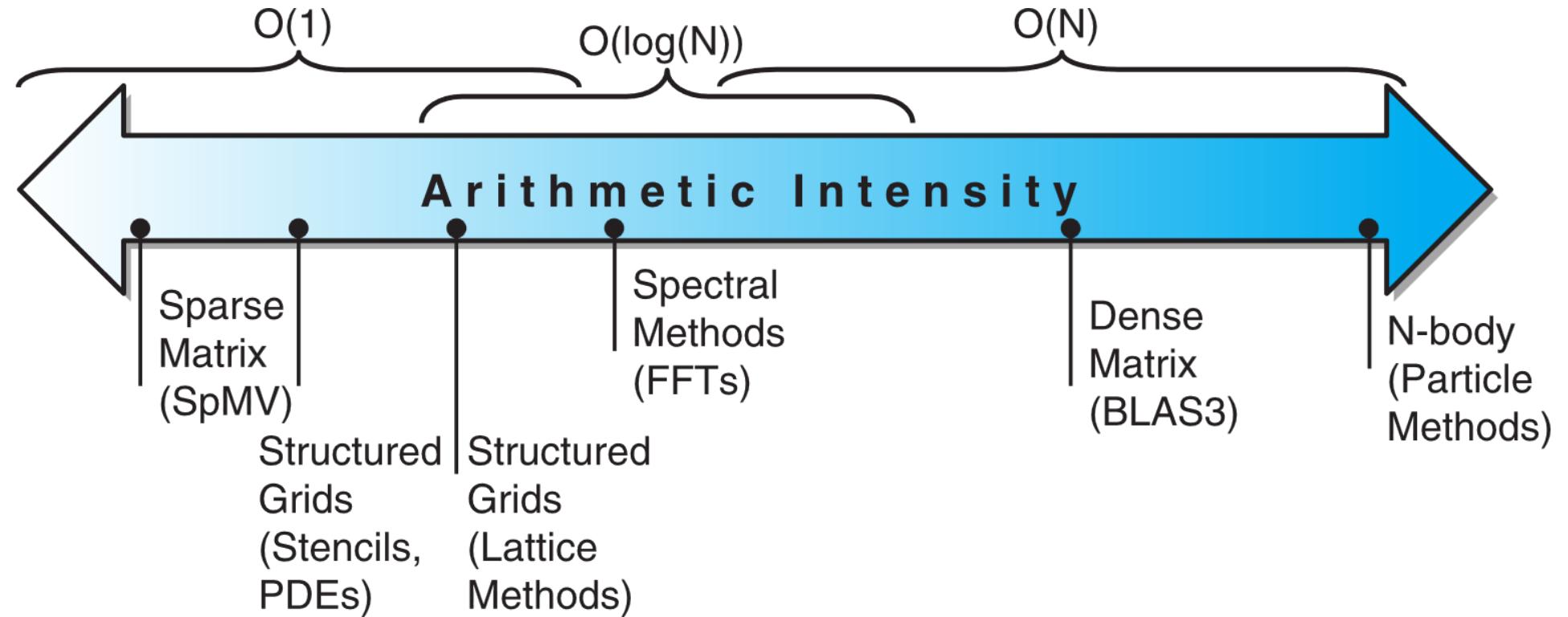
Roofline Model



- The Roofline Model is a **throughput-oriented performance model**
 - **Tracks rates** not times
 - Augmented with Little's Law
 - Concurrency = latency*bandwidth
 - **Independent of ISA and architecture**
 - Applies to CPUs, GPUs, Google TPUs, etc.
- Two Components
 - Machine Characterization
 - Realistic performance potential of the system
 - Theoretical Application Bounds
 - How well could my app perform with perfect compilers, caches, overlap, etc.

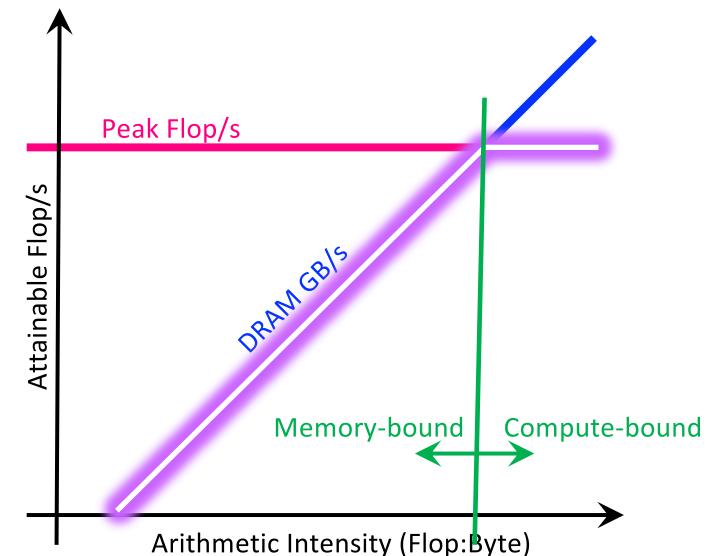
Application Bounds of Numerical Applications

- Arithmetic intensity of a kernel
 - FLOPs per byte of memory accessed
 - $\frac{FP \text{ ops}}{\text{Bytes accessed}}$



(DRAM) Roofline

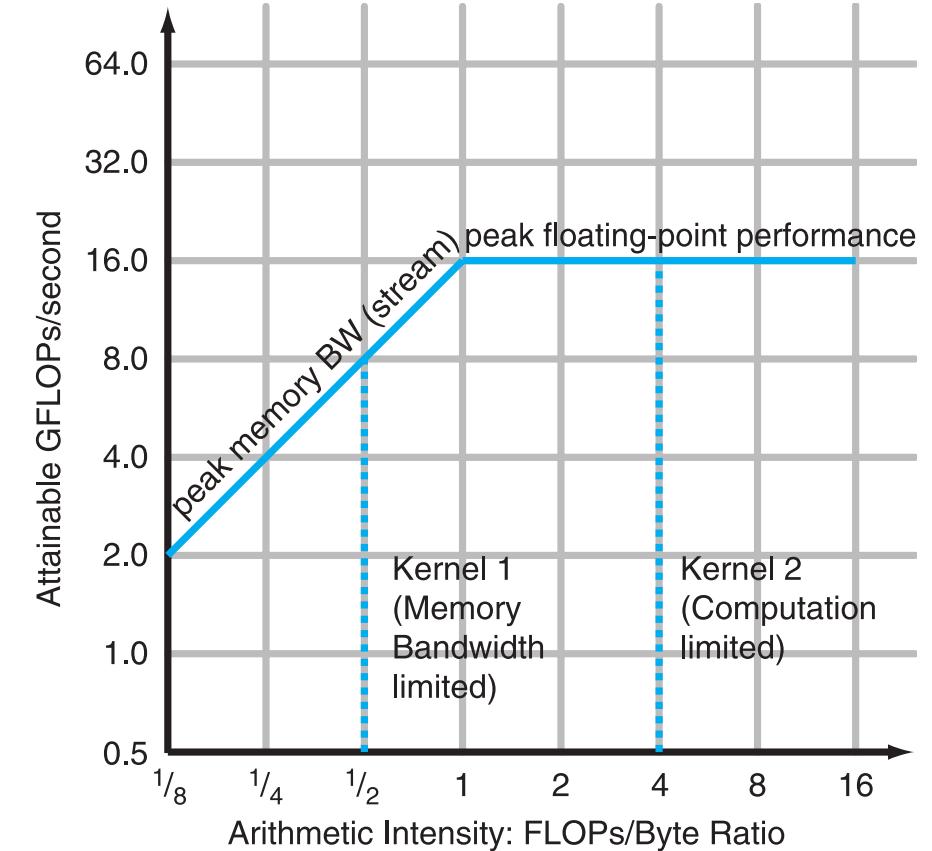
- One could hope to always attain peak performance (FLOP/s)
 - However, **finite locality (reuse) and bandwidth limit performance**
- Consider idealized processor/caches
- Plot the performance bound using Arithmetic Intensity (AI) as the x-axis
 - AI = Flops / Bytes presented to DRAM
 - **Attainable Flop/s = min(peak Flop/s, AI * peak GB/s)**
 - Log-log scale makes it easy to doodle, extrapolate performance along Moore's Law, etc.
 - Kernels with AI less than machine balance are ultimately DRAM bound



The Roofline Model – Simple Model



- For memory intensive FP kernels
 - Log-log scale
 - Data does not fit in cache
- Example
 - 16 GFLOPS peak
 - 16 GB/sec peak
- What does **ridge-point** indicate?



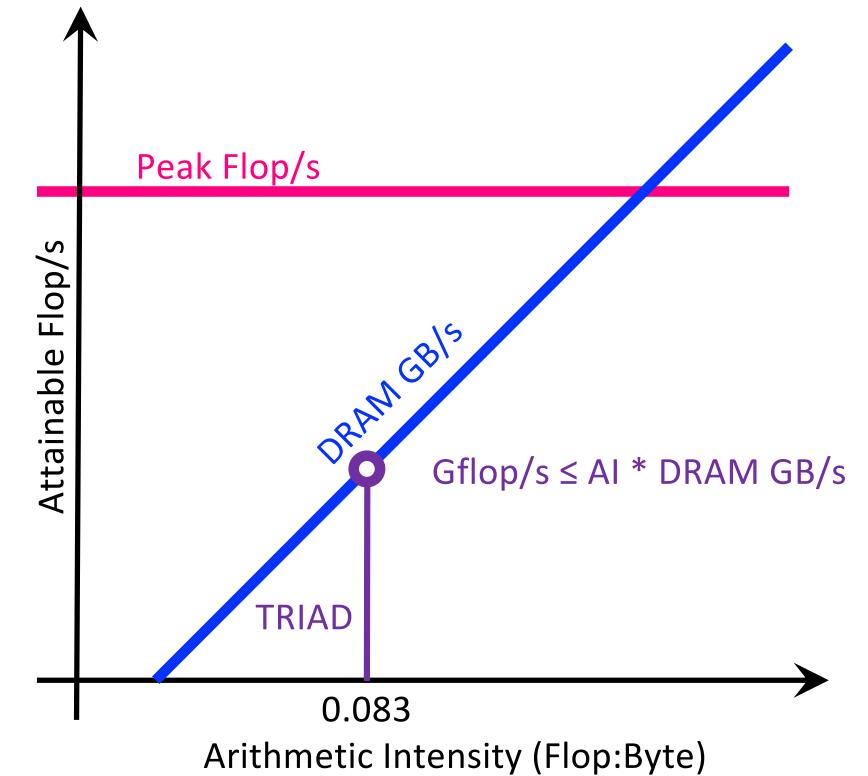
Attainable GPLOPs/sec = Min (Peak Memory BW × Arithmetic Intensity, Peak FP Performance)

Roofline Example

- Typical machine balance is 5-10 FLOPs per byte
 - 40-80 flops per double to exploit compute capability
 - Artifact of technology and money
 - Unlikely to improve**
- Consider STREAM Triad


```
for (i=0; i<N; i++){
    Z[i] = x[i] + α * y[i];
}
```

 - 2flops/iteration
 - Transfer 24 bytes per iteration
 - Read $x[i]$, $y[i]$, write $z[i]$
 - AI = 0.083flops per byte == Memory bound**



Roofline Example

- Conversely, 7-point constant coefficient stencil
 - 7 flops
 - 8 memory references (7 reads, 1 store) per point
 - Cache can filter all but 1 read and 1 write per point
 - AI = 0.44 flops per byte == memory bound**
 - but 5x the flop rate**

```

for {k=0; k<dim+1 k++}{  

    for {j=0; j<dim+1 j++}{  

        for {i=0; i<dim+1 i++}{  

            int ijk = i + j*Stride + k*Stride;  

            new[ijk] = -6.0 * old[ijk] \  

                +old[ijk-1] \  

                +old[ijk+1] \  

                +old[ijk-jStride] \  

                +old[ijk+jStride] \  

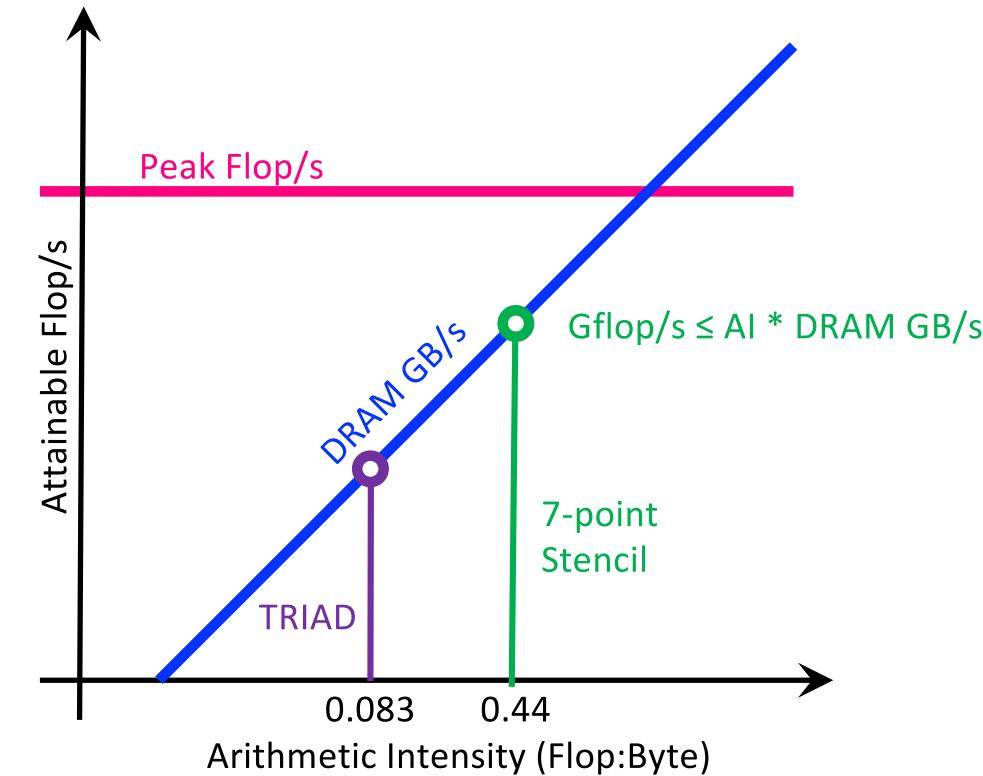
                +old[ijk-kStride] \  

                +old[ijk+kStride];  

        }  

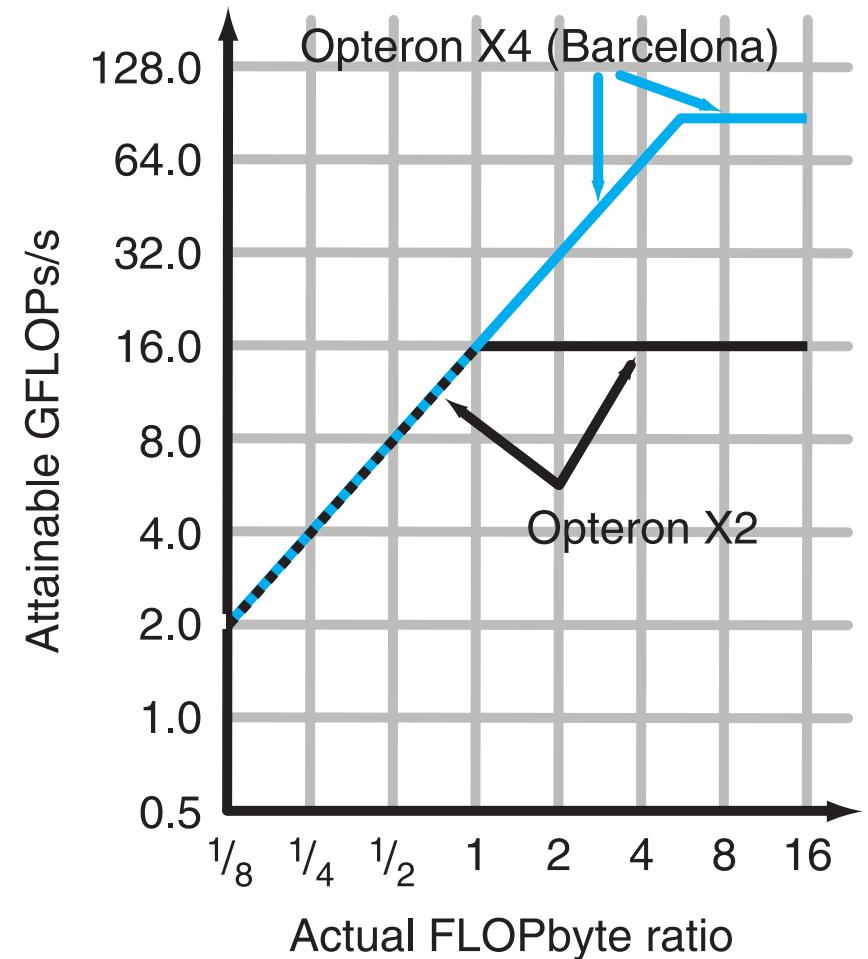
    }  

}
  
```



Comparing Systems

- Example: Opteron X2 vs. Opteron X4
 - 2-core vs. 4-core, $2 \times$ FP performance/core, 2.2GHz vs. 2.3GHz
 - Same memory system
- To get higher performance on X4 than X2
 - **Need higher arithmetic intensity**
 - Or working set must fit in X4's 2MB L-3 cache

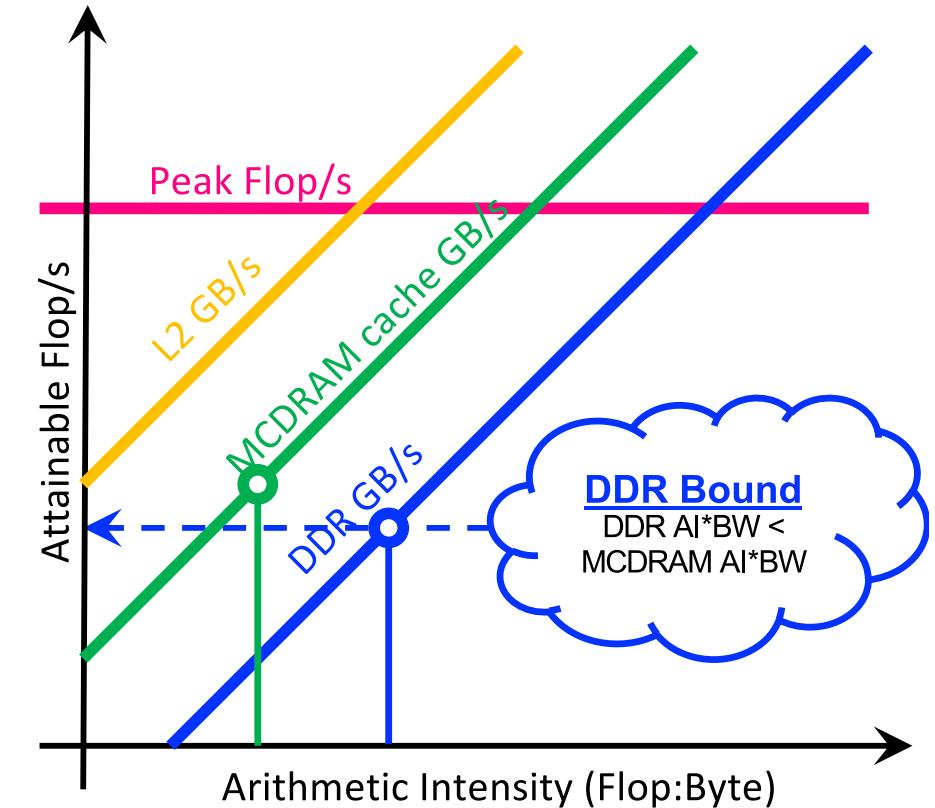


Hierarchical Roofline

- Real processors have multiple levels of memory
 - Registers
 - L1, L2, L3 cache
 - MCDRAM/HBM (KNL/GPU device memory)
 - DDR (main memory)
 - NVRAM (non-volatile memory)
- Applications can have locality in each level
 - Unique data movements imply unique AI's
 - Moreover, **each level will have a unique bandwidth**

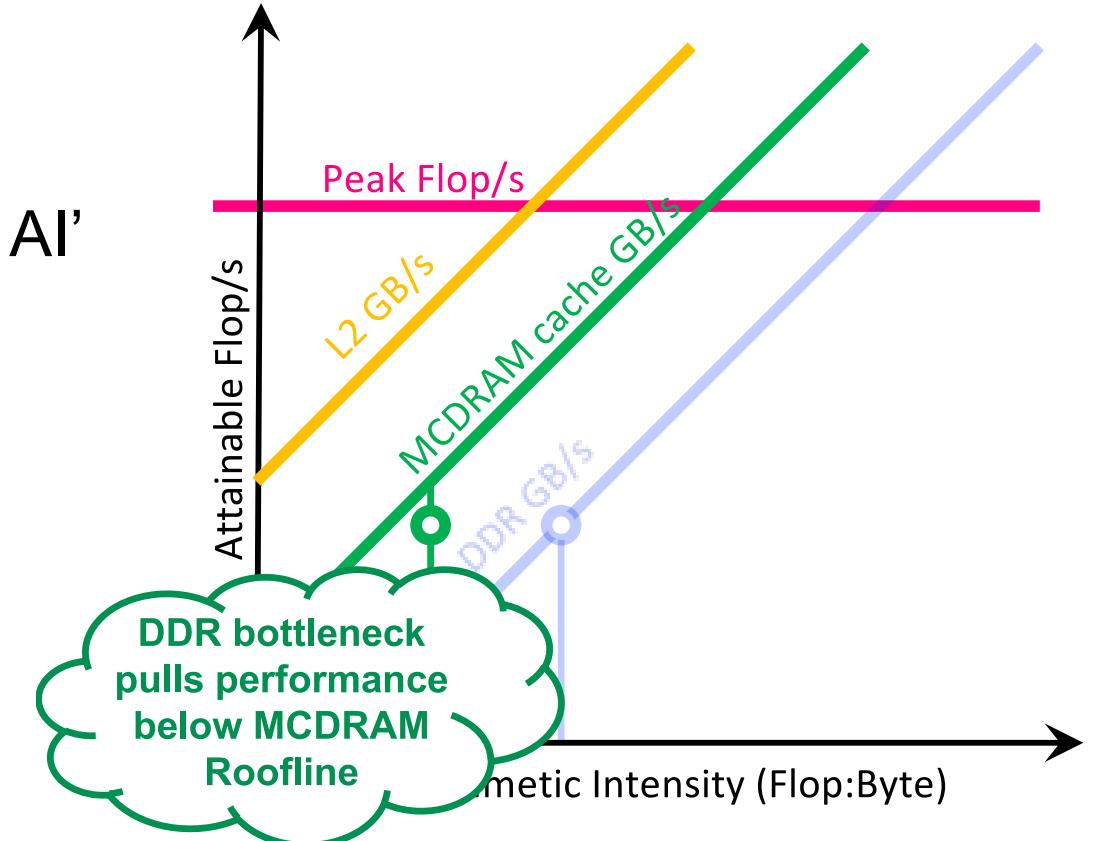
Hierarchical Roofline

- Construct superposition of Rooflines
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although a loop nest may have multiple AI's and multiple bounds (flops, L1, L2, ... DRAM)
- **Performance is bound by the minimum**



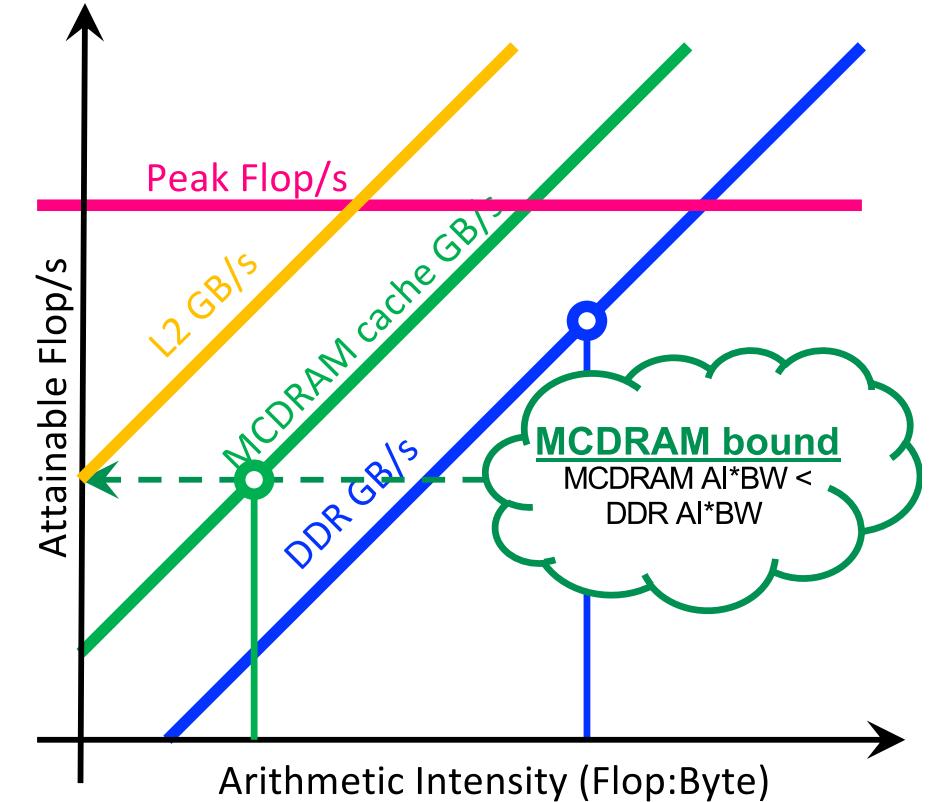
Hierarchical Roofline

- Construct superposition of Rooflines
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although a loop nest may have multiple AI' and multiple bounds (flops, L1, L2, ... DRAM)
- **Performance is bound by the minimum**



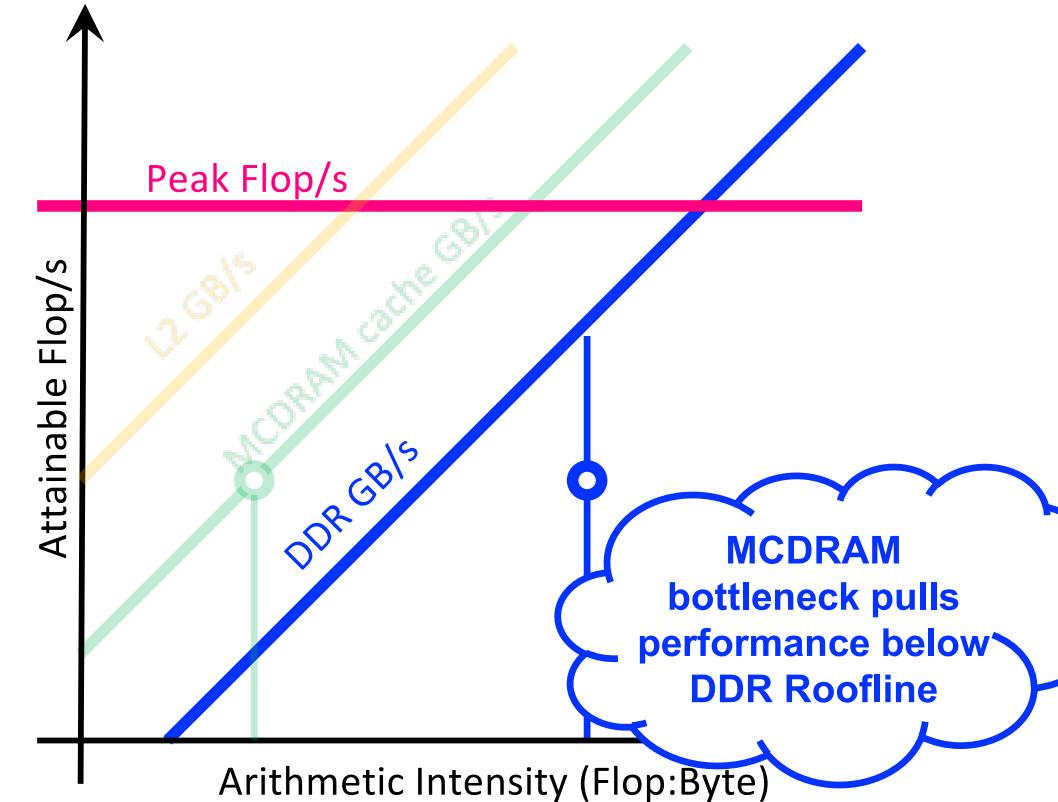
Hierarchical Roofline

- Construct superposition of Rooflines
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although a loop nest may have multiple AI' and multiple bounds (flops, L1, L2, ... DRAM)
- **Performance is bound by the minimum**



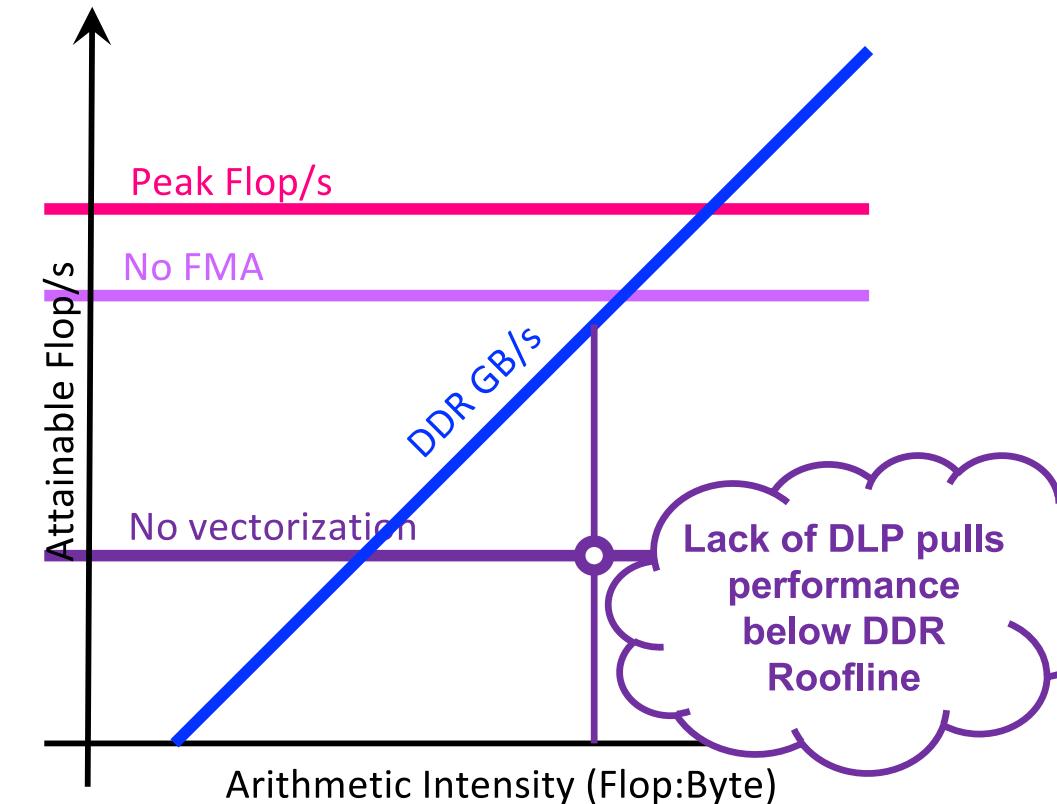
Hierarchical Roofline

- Construct superposition of Rooflines
 - Measure a bandwidth
 - Measure AI for each level of memory
 - Although a loop nest may have multiple AI' and multiple bounds (flops, L1, L2, ... DRAM)
- **Performance is bound by the minimum**



Data, Instruction, Thread-Level Parallelism

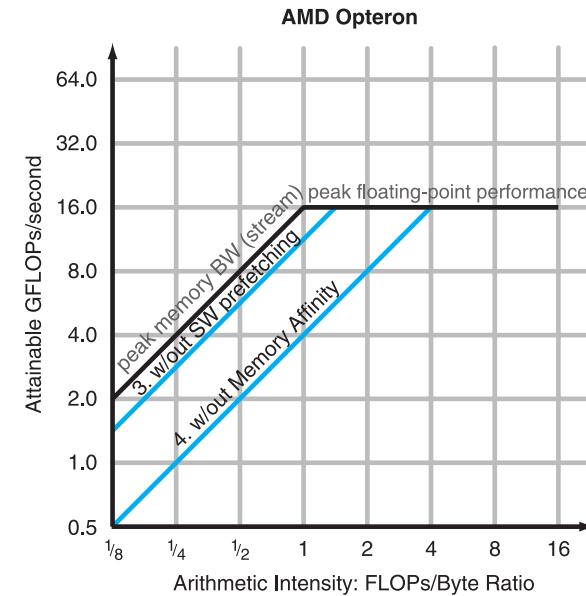
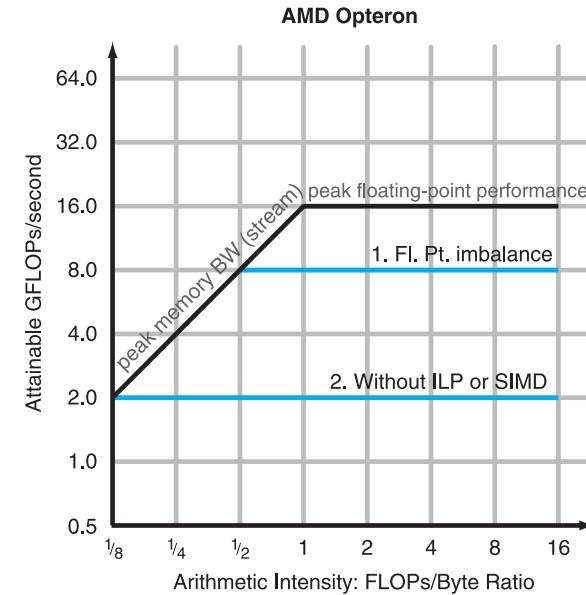
- We have assumed one can attain peak flops
- In reality, this is premised on sufficient ...
 - Use special instructions (e.g. fused multiply-add)
 - Vectorization/SIMD (16 flops per instruction)
 - Unrolling, out-of-order execution (hide FPU latency)
 - OpenMP across multiple cores
- Without these
 - Peak performance is not attainable
 - Some kernels can transition from memory-bound to compute-bound
 - n.b. in reality, DRAM bandwidth is often tied to DLP and TLP (single core can't saturate BW w/scalar code)



Optimizing Performance

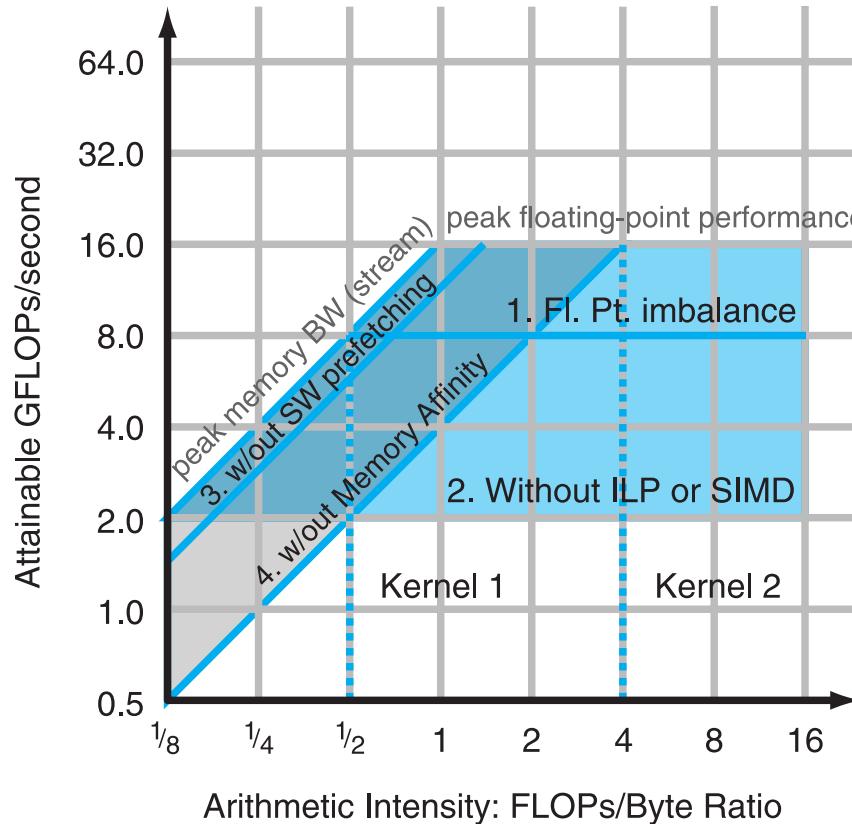


- Increase thread parallelism
- Optimize FP performance
 - Balance adds & multiplies
 - Improve superscalar ILP and use of SIMD instructions
 - Loop unrolling, software pipelining
- Optimize memory usage
 - Software prefetch (SW pipelining)
 - Avoid load stalls
 - Memory affinity
 - NUMA(Non-Uniform Memory Access)
 - Avoid non-local data accesses



Optimizing Performance

- Choice of optimization depends on arithmetic intensity of code (Kernel 1 vs. Kernel 2)



- Arithmetic intensity is not always fixed
 - May increase with problem size
 - Dense matrix, N-body
 - Weak scaling advantage
 - Caching reduces memory accesses
 - Increases arithmetic intensity
 - Blocking
 - Use hierarchical roofline

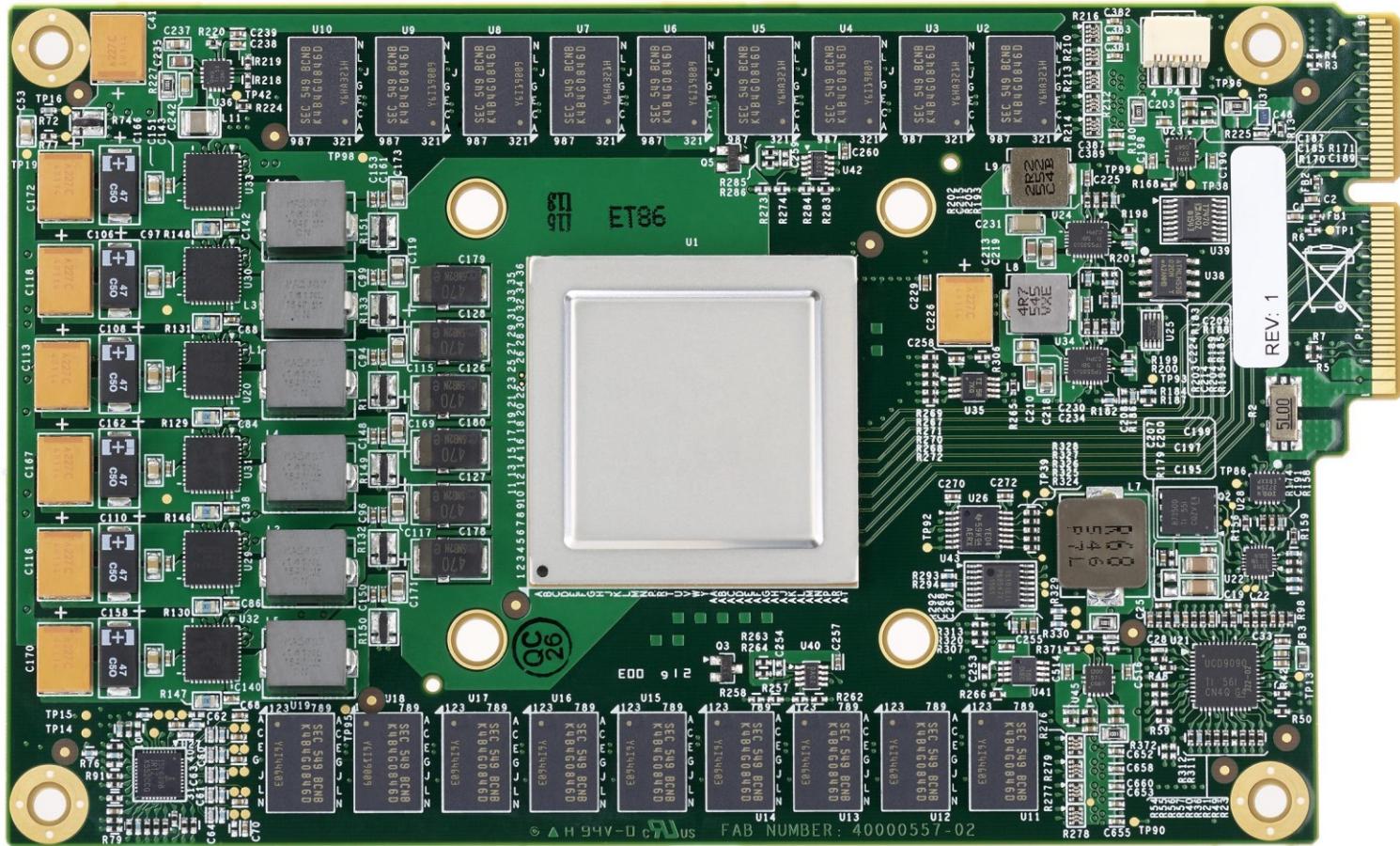
TPU: Avoid Success/Disaster for NN Apps



- 2013: NNs apps would require 2x–3x CPUs
- Custom hardware to reduce TCO of NN
- Improve inference by 10x vs. GPUs
- Short development cycle
 - Start in 2014 deployed 15 months later in datacenter

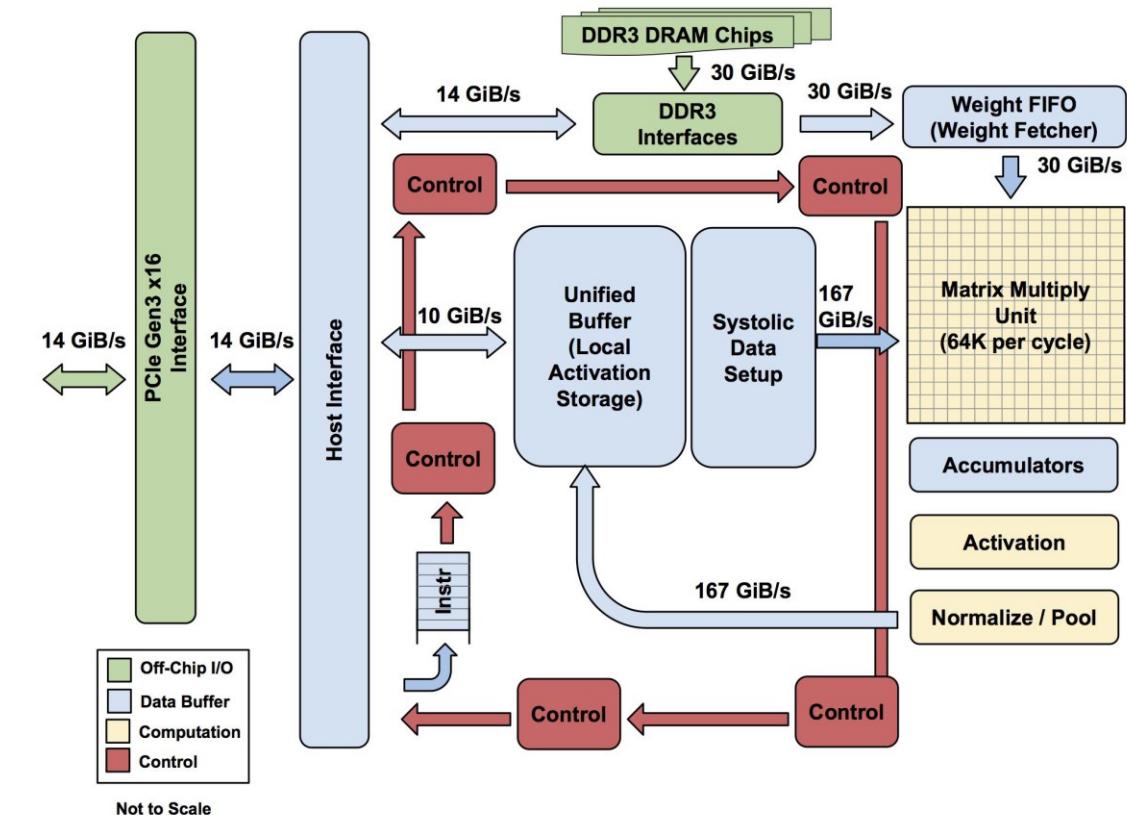
TPU Card

- Up to 4 cards per server



TPU High-level Chip Architecture

- The Matrix Unit
 - 256x256(65,536) 8-bit MAC units
 - Systolic array
- 700Mhz clock rate
- Peak: 92T operations/seconds
 - $65,536 \times 2 \times 700M$
- 4Mb of on-chip Accumulator memory
- 24MB of on-chip Unified Buffer
 - for activation
- 2 x 2133Mhz DDR3 DRAM channel
- 8GB of off-chip weights DRAM memory
- V.s. GPU and CPU
 - >25x as many MACs vs. GPU
 - >100x as many MACs vs. CPU



TPU Programmer's View



- Five key CISC instructions (CPI > 10)
 - Read_Host_Memory
 - Write_Host_Memory
 - Read_Weights
 - MatrixMultiply/Convolve
 - Activate(ReLU,Sigmoid,Maxpool,LRN,...)
- Complexity in software
 - No branches
 - In-order issue
 - Software controlled buffers
 - Software controlled pipeline synchronization

2016 NN Datacenter Workload



Name	LOC	Layers					<i>Nonlinear function</i>	Weights	<i>TPU Ops / Weight Byte</i>	<i>TPU Batch Size</i>	% Deployed
		FC	Conv	Vector	Pool	Total					
MLP0	0.1k	5				5	ReLU	20M	200	200	61%
MLP1	1k	4				4	ReLU	5M	168	168	
LSTM0	1k	24		34		58	sigmoid, tanh	52M	64	64	29%
LSTM1	1.5k	37		19		56	sigmoid, tanh	34M	96	96	
CNN0	1k		16			16	ReLU	8M	2888	8	5%
CNN1	1k	4	72		13	89	ReLU	100M	1750	32	

Three Contemporary Chips

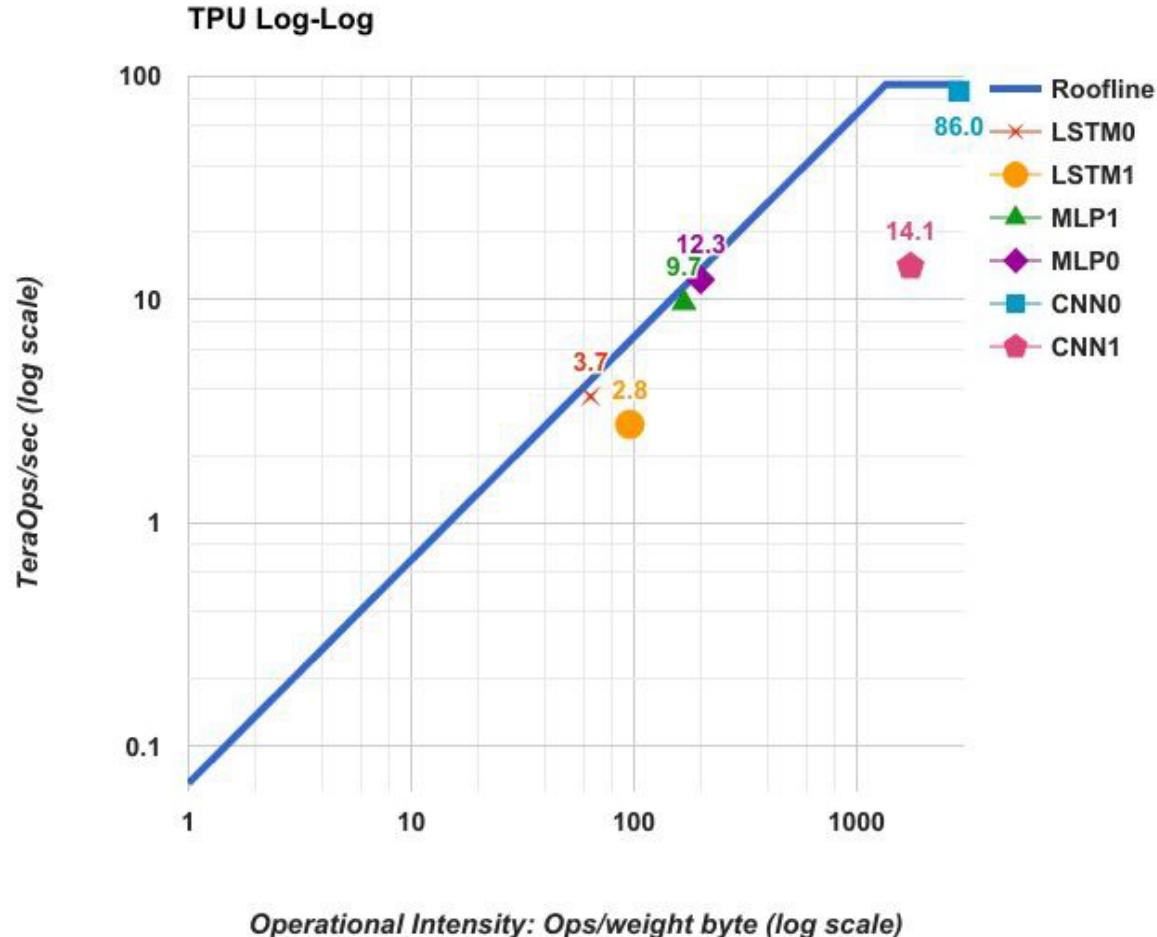
Processor	mm^2	Clock MHz	TDP Watts	Idle Watts	Memory GB/sec	Peak TOPS/chip	
						8b int.	32b FP
CPU: Haswell (18 core)	662	2300	145	41	51	2.6	1.3
GPU: Nvidia K80 (2 / card)	561	560	150	25	160	--	2.8
TPU	<331*	700	75	28	34	91.8	--

TPU is less than half die size of the Intel Haswell processor

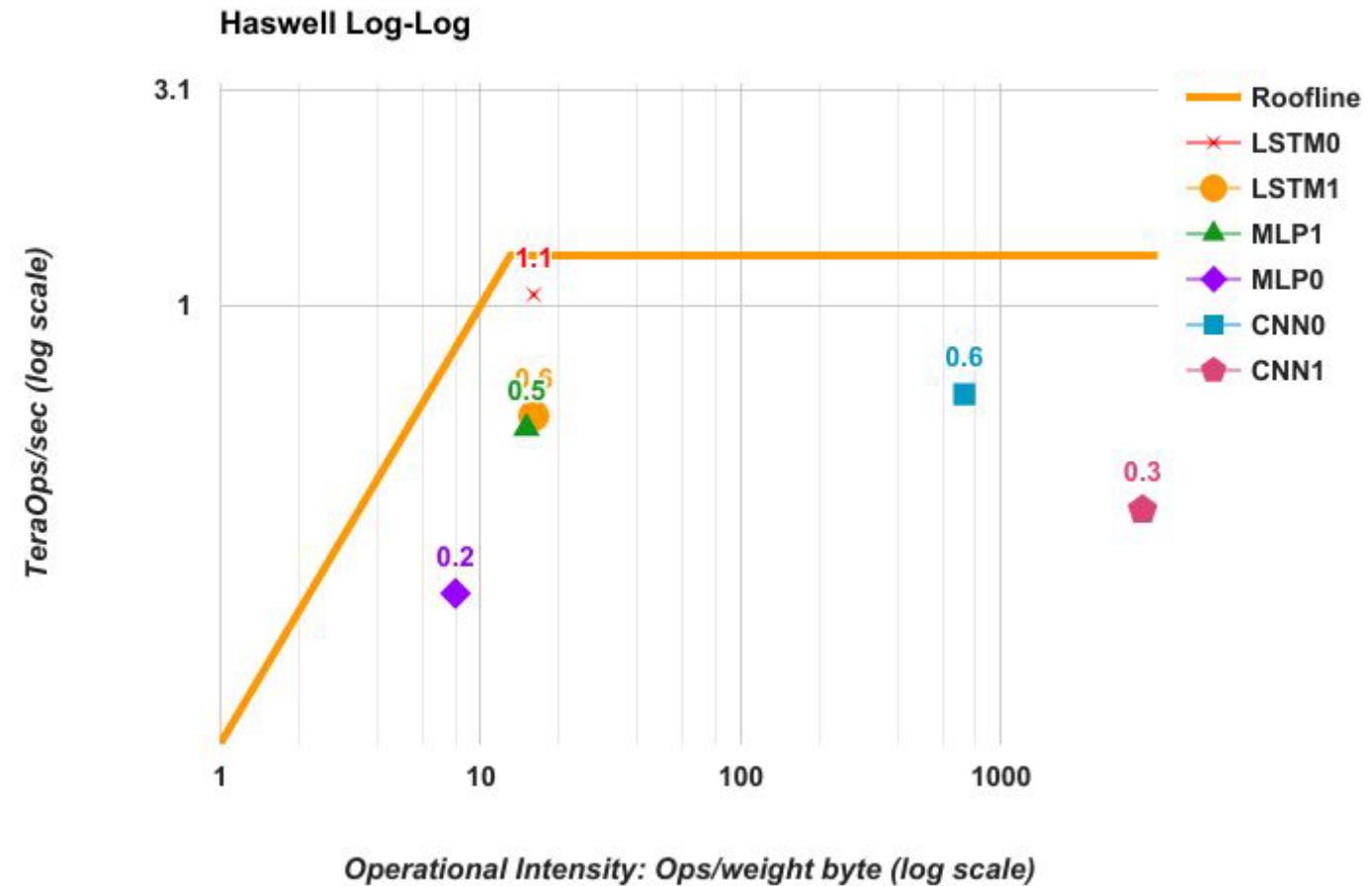
K80 and TPU in 28 nm process; Haswell fabbed in Intel 22 nm process

These chips and platforms chosen for comparison because widely deployed in Google data centers

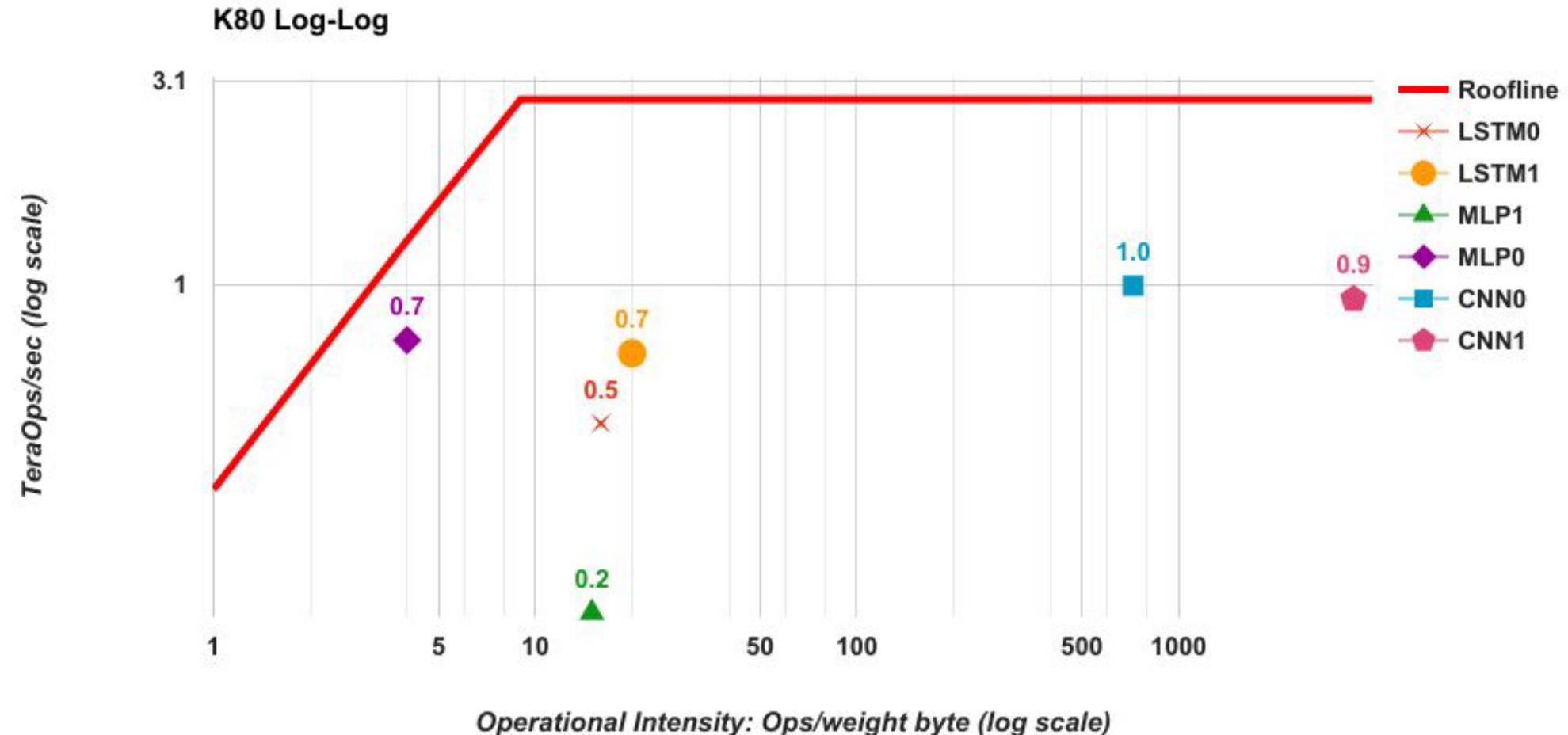
TPU Roofline



CPU (Haswell) Roofline



GPU (K80) Roofline



Why Below Rooflines (MLP0)

<i>Type</i>	<i>Batch</i>	<i>99th% Response</i>	<i>Inf/s (IPS)</i>	<i>% Max IPS</i>
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%

Performance of TPU & GPU Relative to CPU

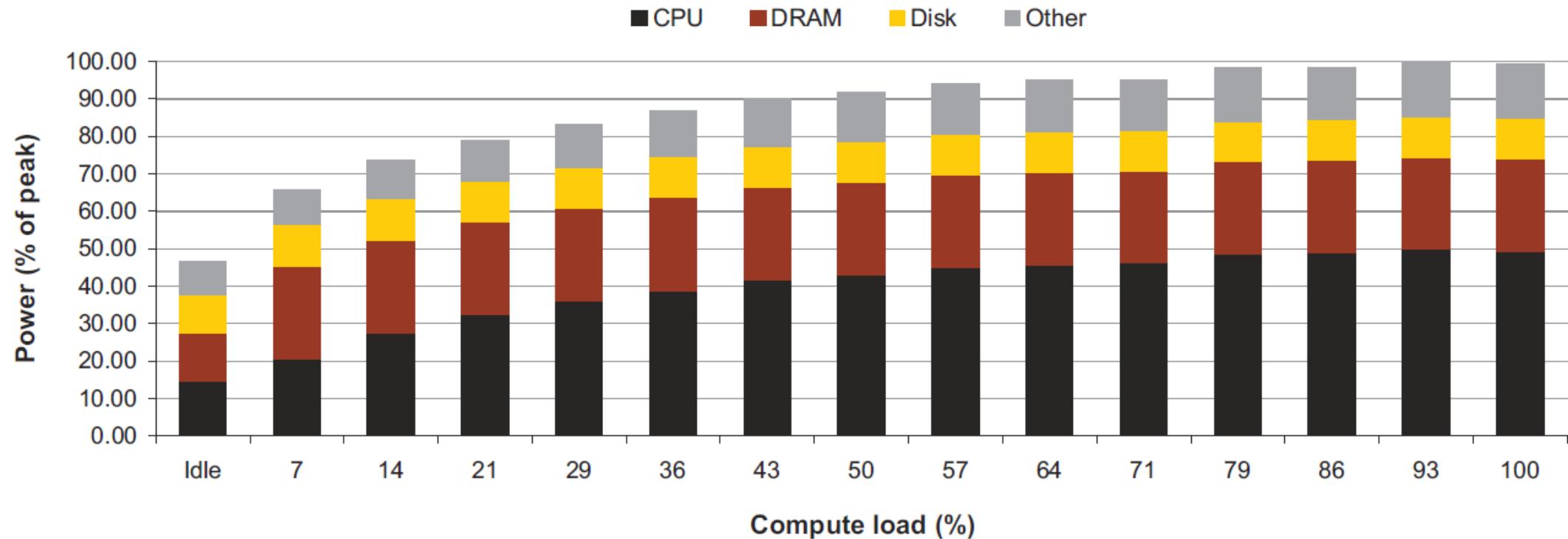


Type	MLP		LSTM		CNN		Weighted Mean
	0	1	0	1	0	1	
GPU	2.5	0.3	0.4	1.2	1.6	2.7	1.9
TPU	41.0	18.5	3.5	1.2	40.3	71.0	29.2
Ratio	16.7	60.0	8.0	1.0	25.4	26.3	15.3

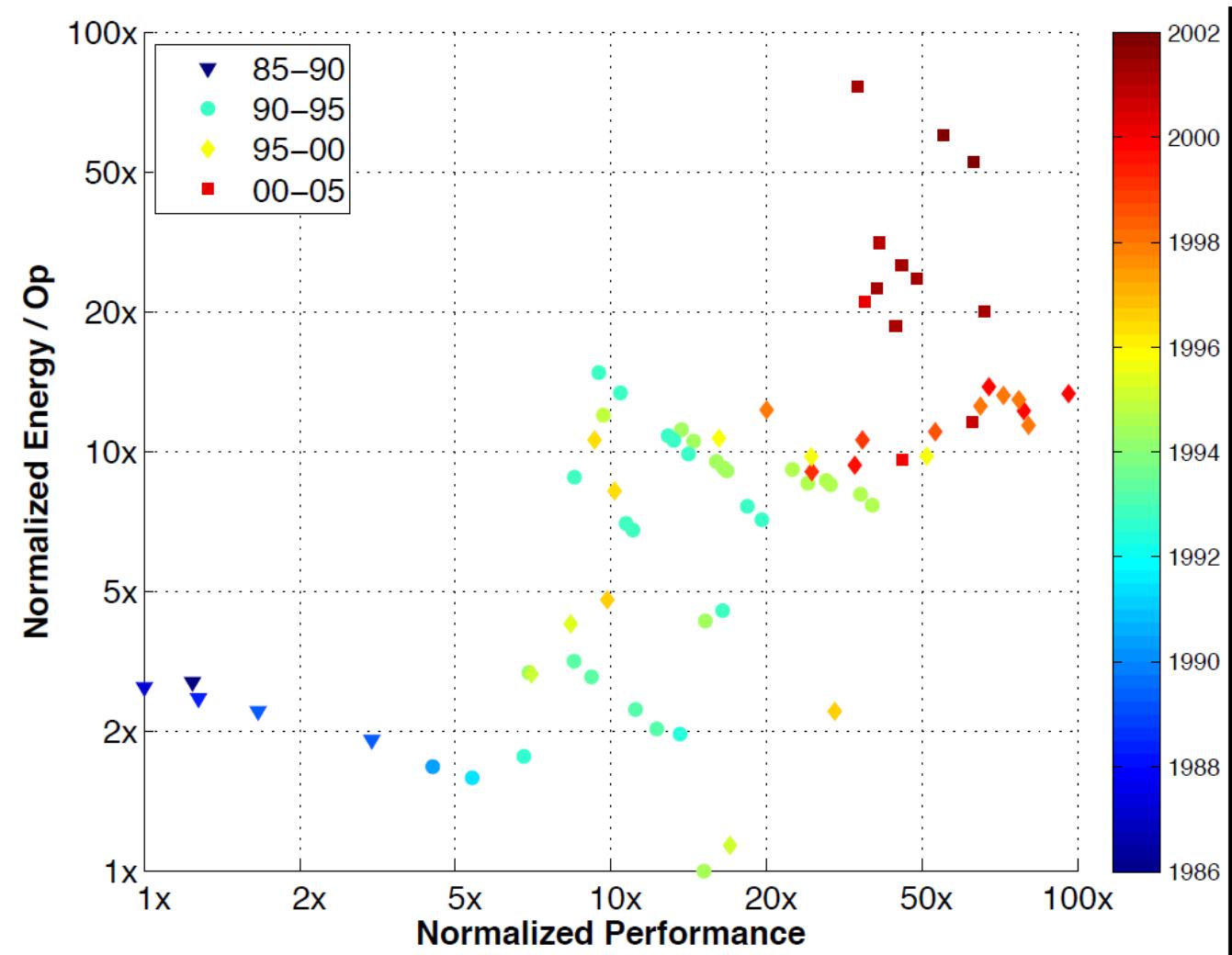
Outline

- Benchmarking Metrics
- GEMM Accelerator Design
- DNN Accelerator Design
- Roofline Model
- Energy

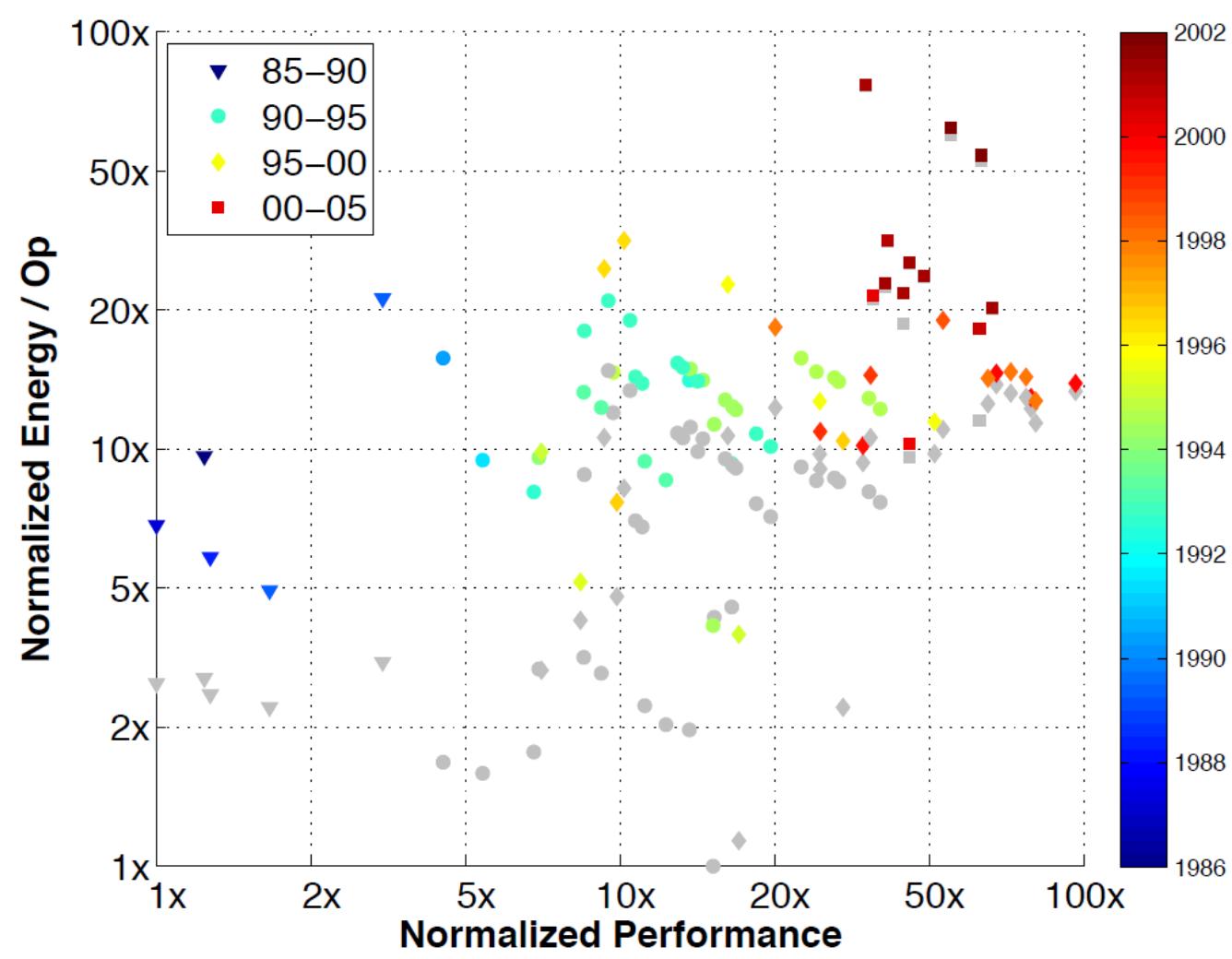
Don't Forge Memory System Energy



Don't Forget Cache Energy

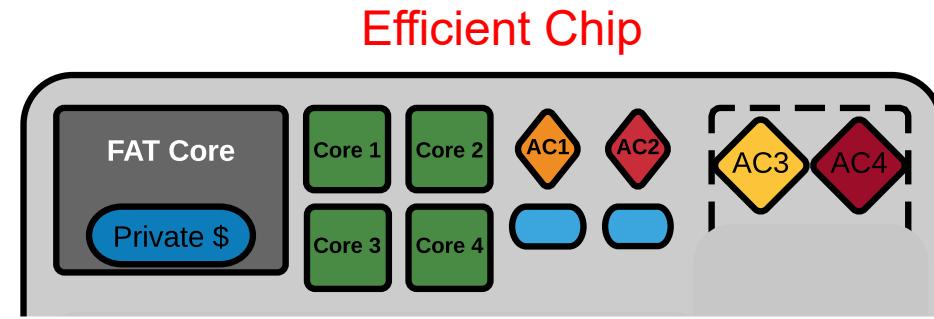


Energy with Corrected Cache Size



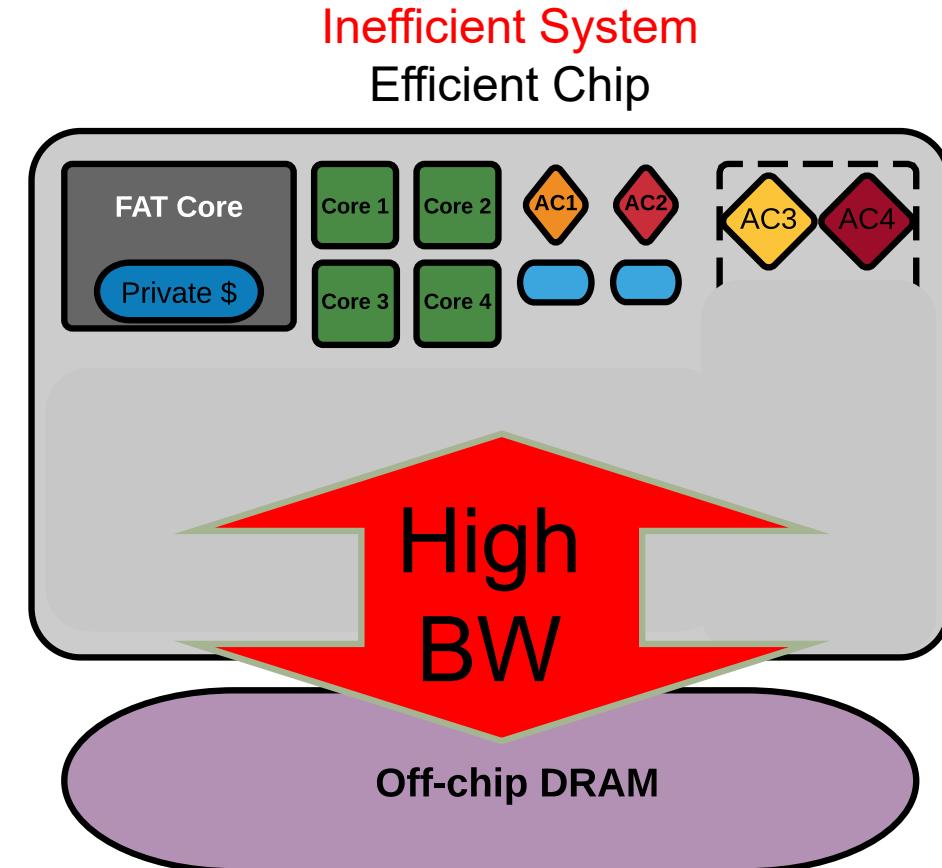
Dark Silicon View

- Efficient Accelerators
- Increase Chip's
 - Energy Efficiency
 - Area Efficiency
- Report 10~1000x improvements



Real Perspectives

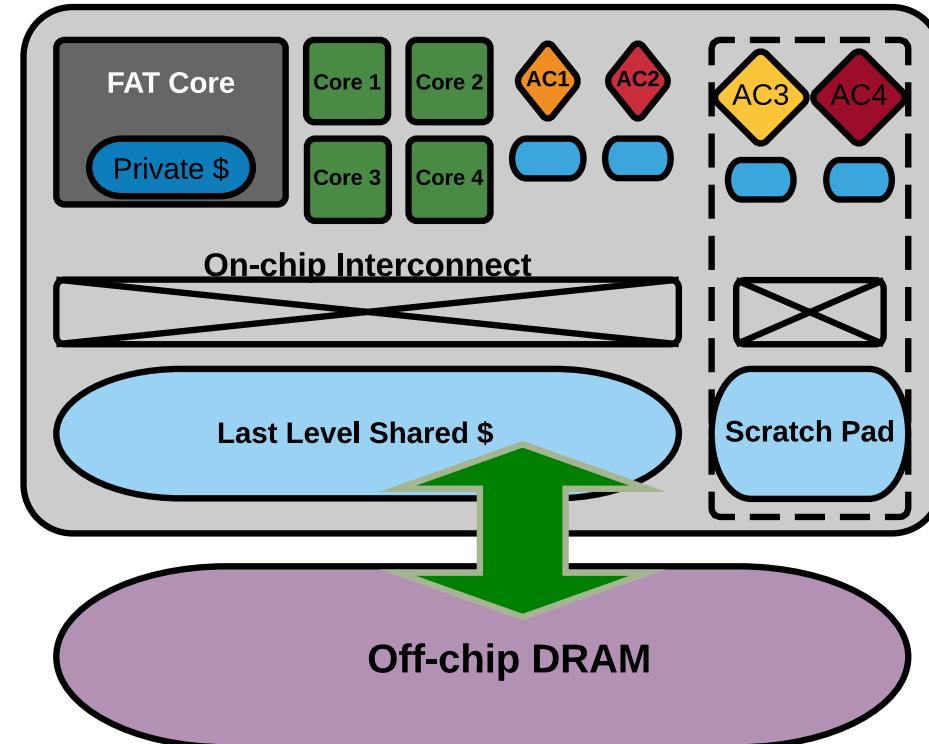
- Efficient Accelerators
- Increase ~~Chip's~~
 - Energy Efficiency
 - Area Efficiency
- Report ~~10~1000x~~ improvements



Dark Memory Perspectives

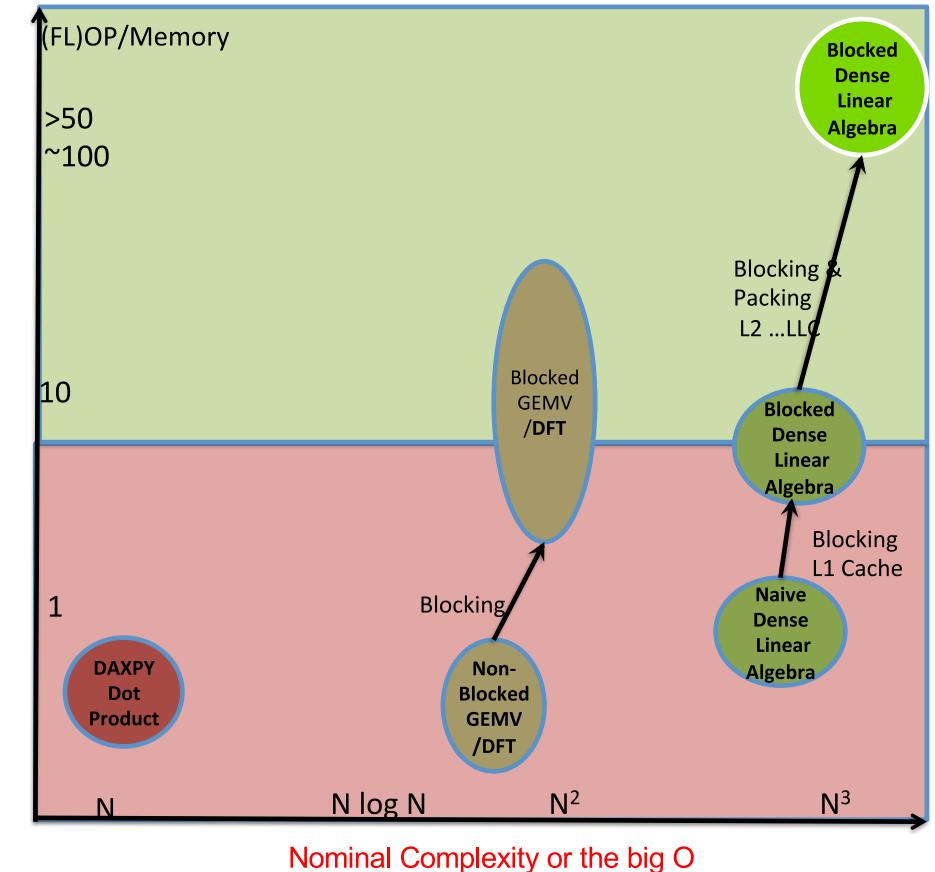
- Efficient System
- On-chip buffers
 - Less efficient chip
- Lower Off-chip Memory Bandwidth

Efficient System
Locality

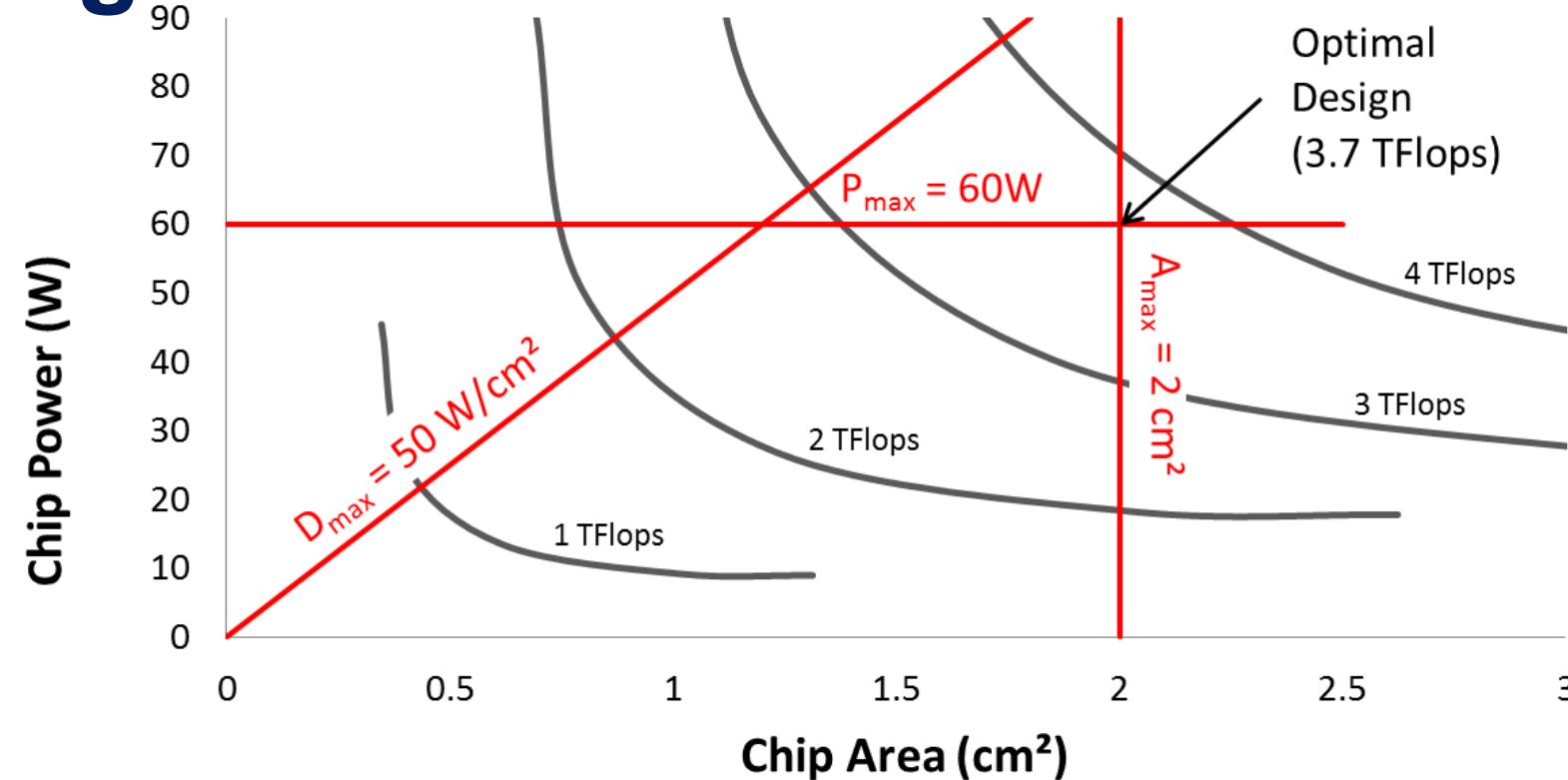


Dark Memory

- Keep DRAM and Memory Hierarchy Dark
- First Focus on Memory Hierarchy
- Algorithm: minimize DRAM access
- High chip level locality when parallelized

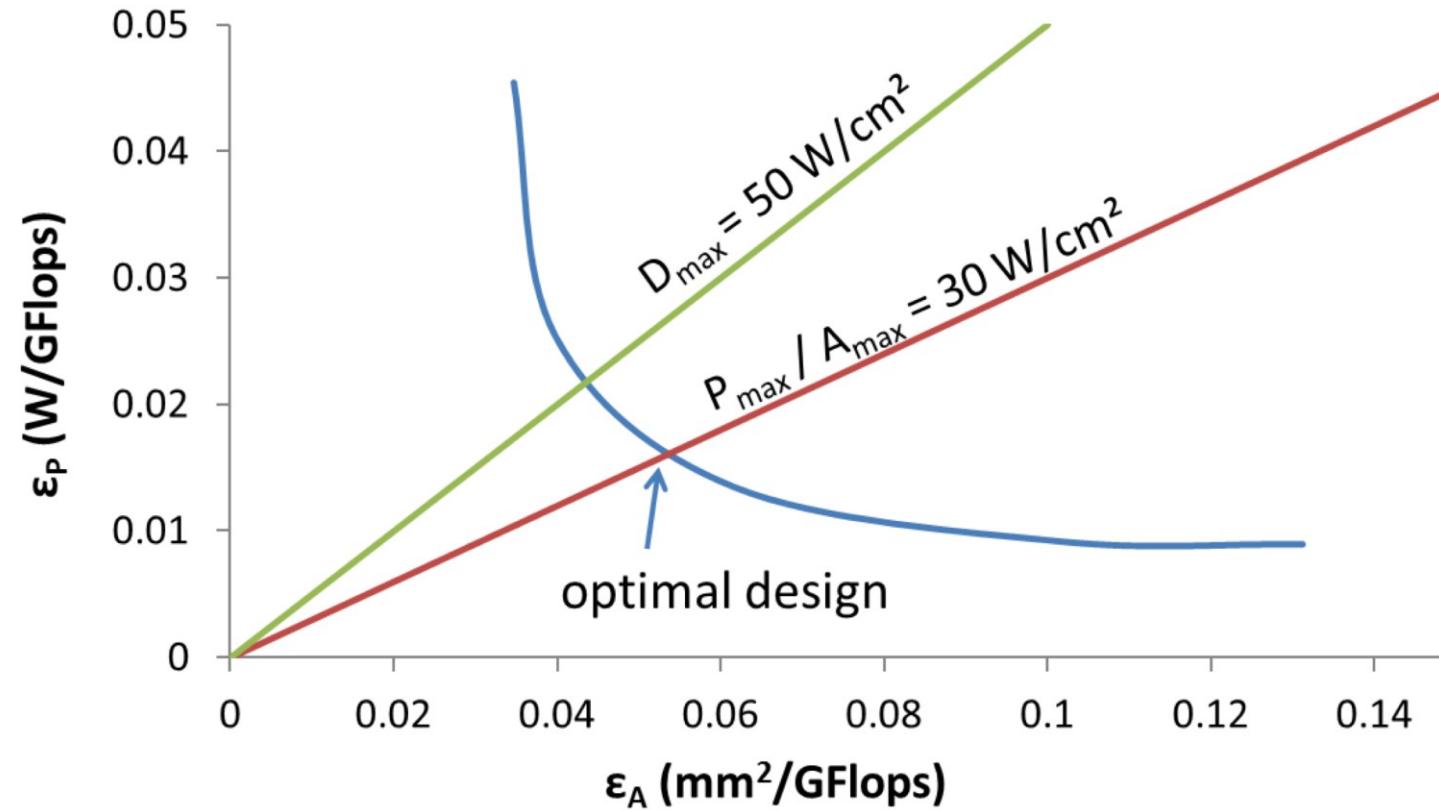


Metrics for Resource Constrained Computing



- Data Parallel Space
- More Performance
- More Hardware
 - More Power
 - More Area

Scale by Performance

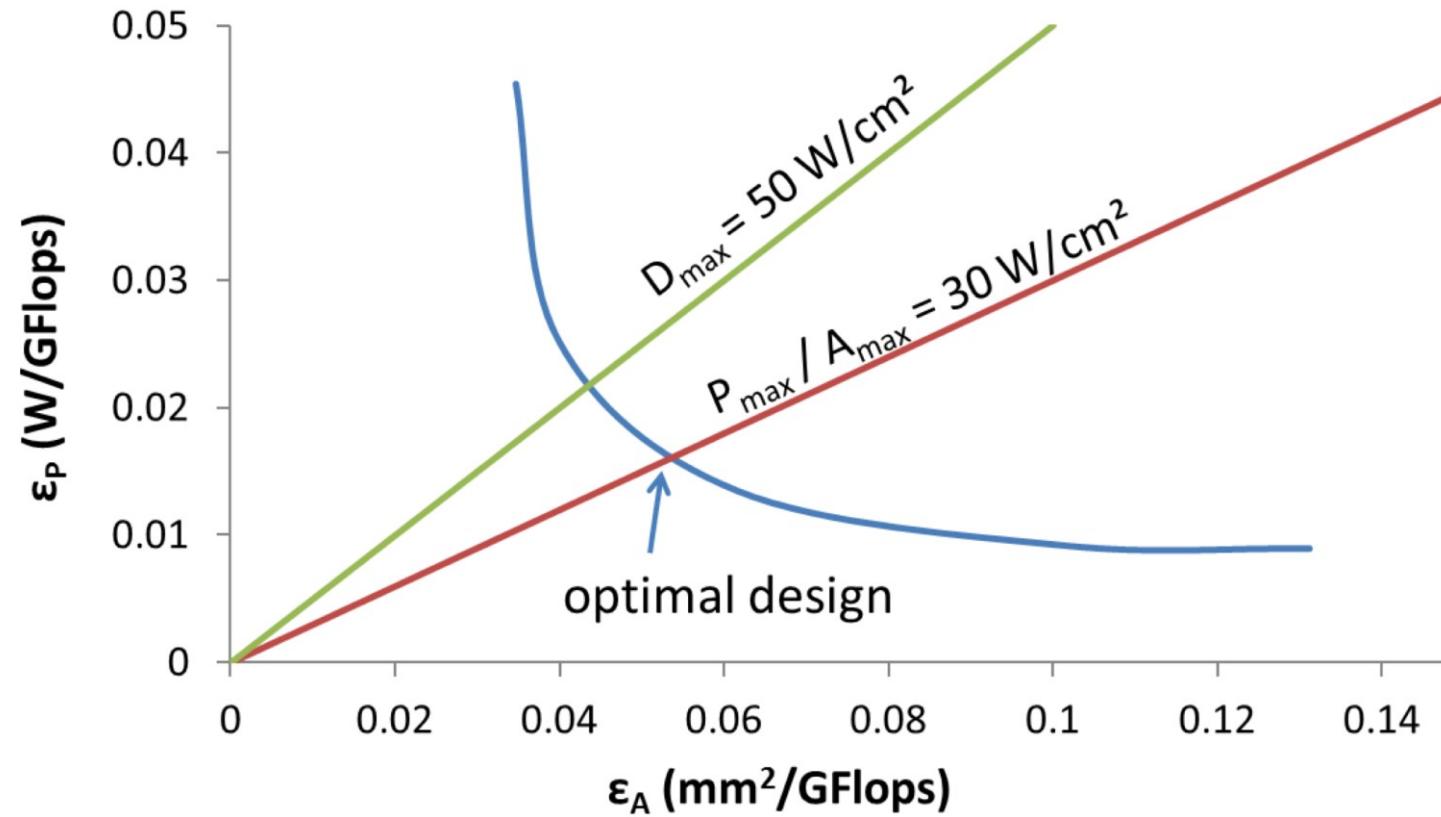


- Energy/OP
- Area/OP
- One Pareto Optimal
- Diming the chip

Always (FL)OPs?

- Define op
 - **Invariant** across the different implementations
- DFT and FFT
 - Op is one Transform
- (Convolutional) Neural Networks
 - Op is one Inference
- Dense and Sparse Solvers
 - Op is a number of solutions

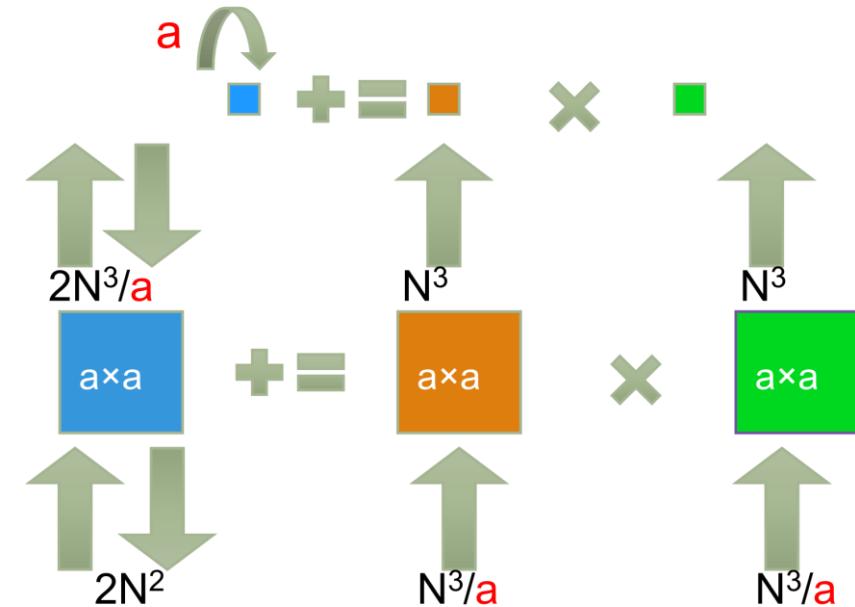
System Design Optimization



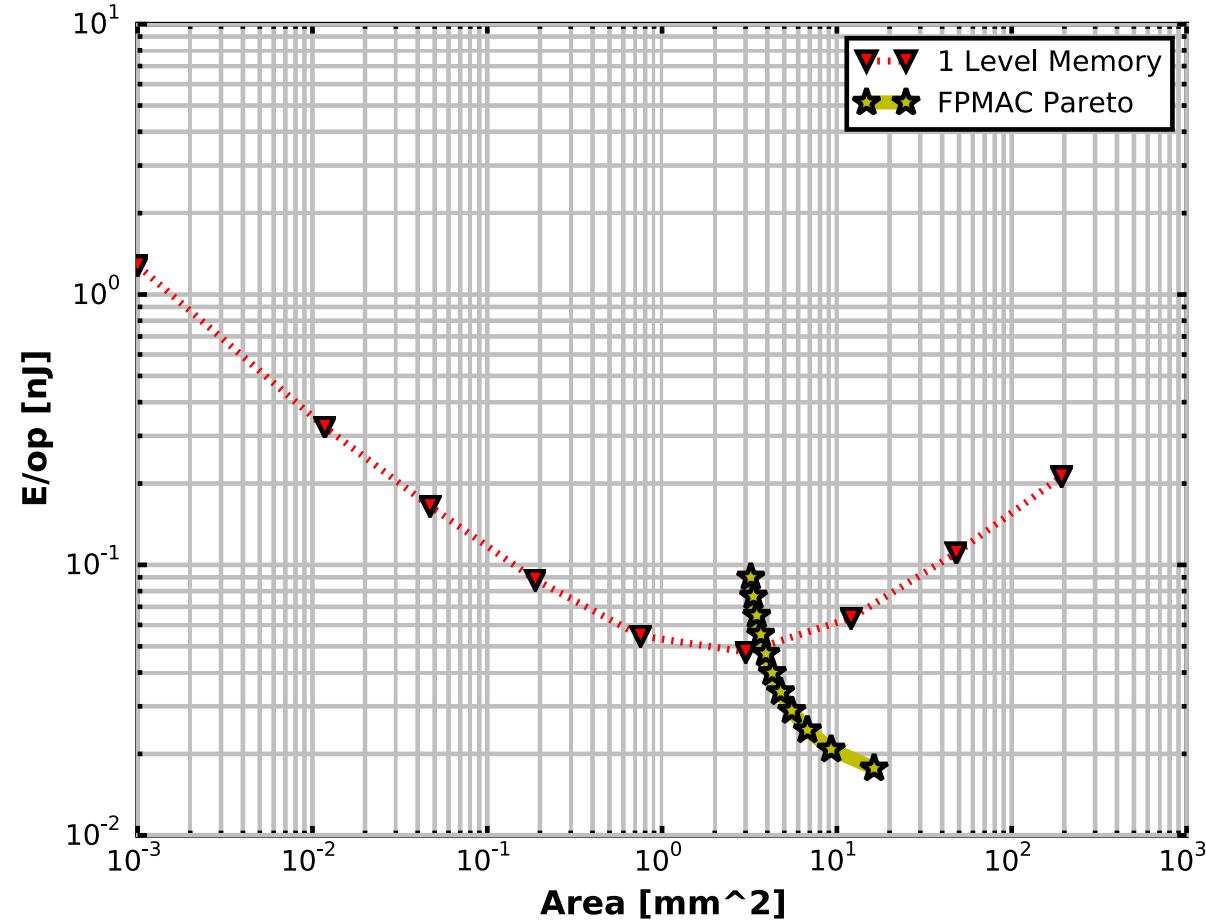
- Sensitivity study along the Pareto Curves (Puzzle)
- Trade Area/Performance of one IP with another
- Both Scalable and Non-scalable

Example: Blocked GEMM

- A or B
 - From axa buffer
 - N^3/a
- C
 - Access for last level cache is $2N^2$
 - Highest level always suffers from constant N^3 accesses to A and B



One Level Caching DP-FP

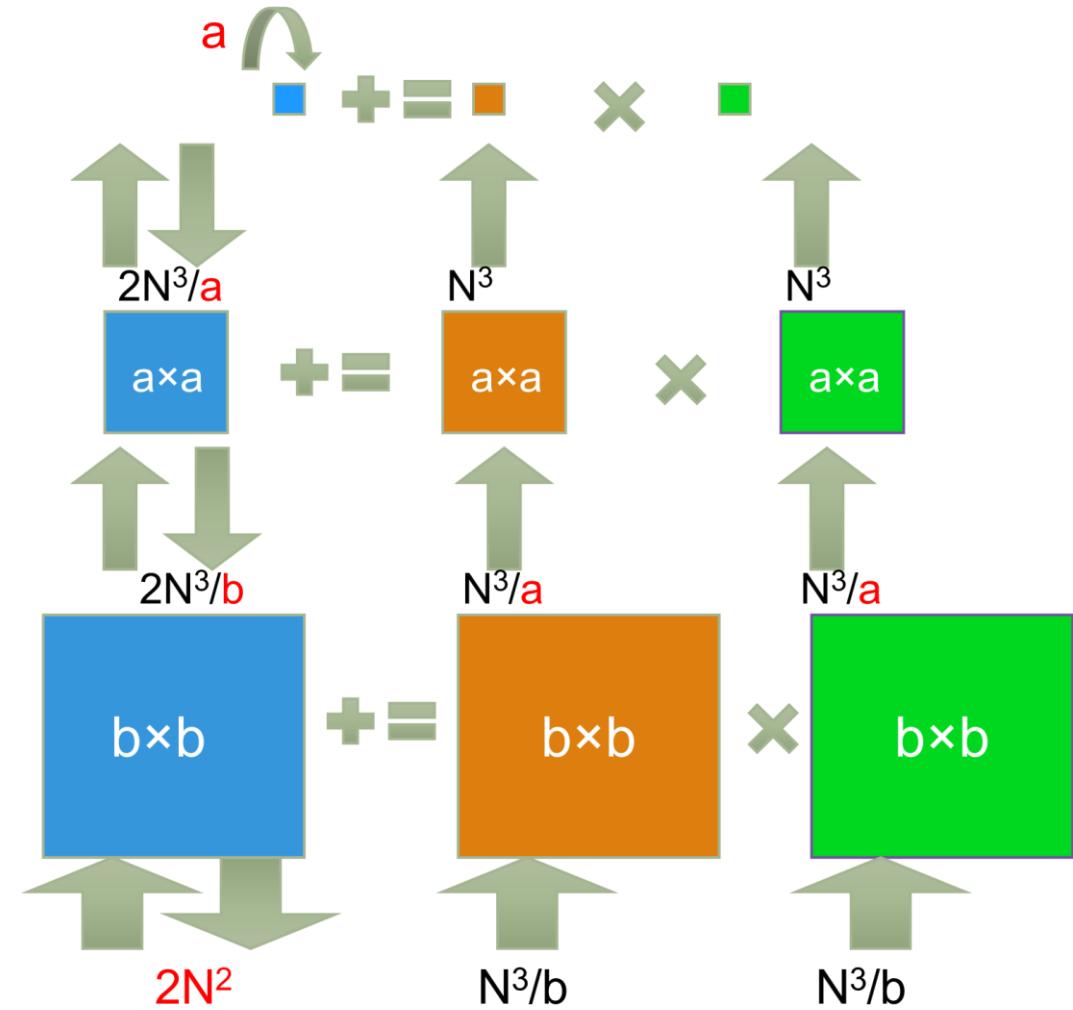


The Energy vs. Size of Top Level

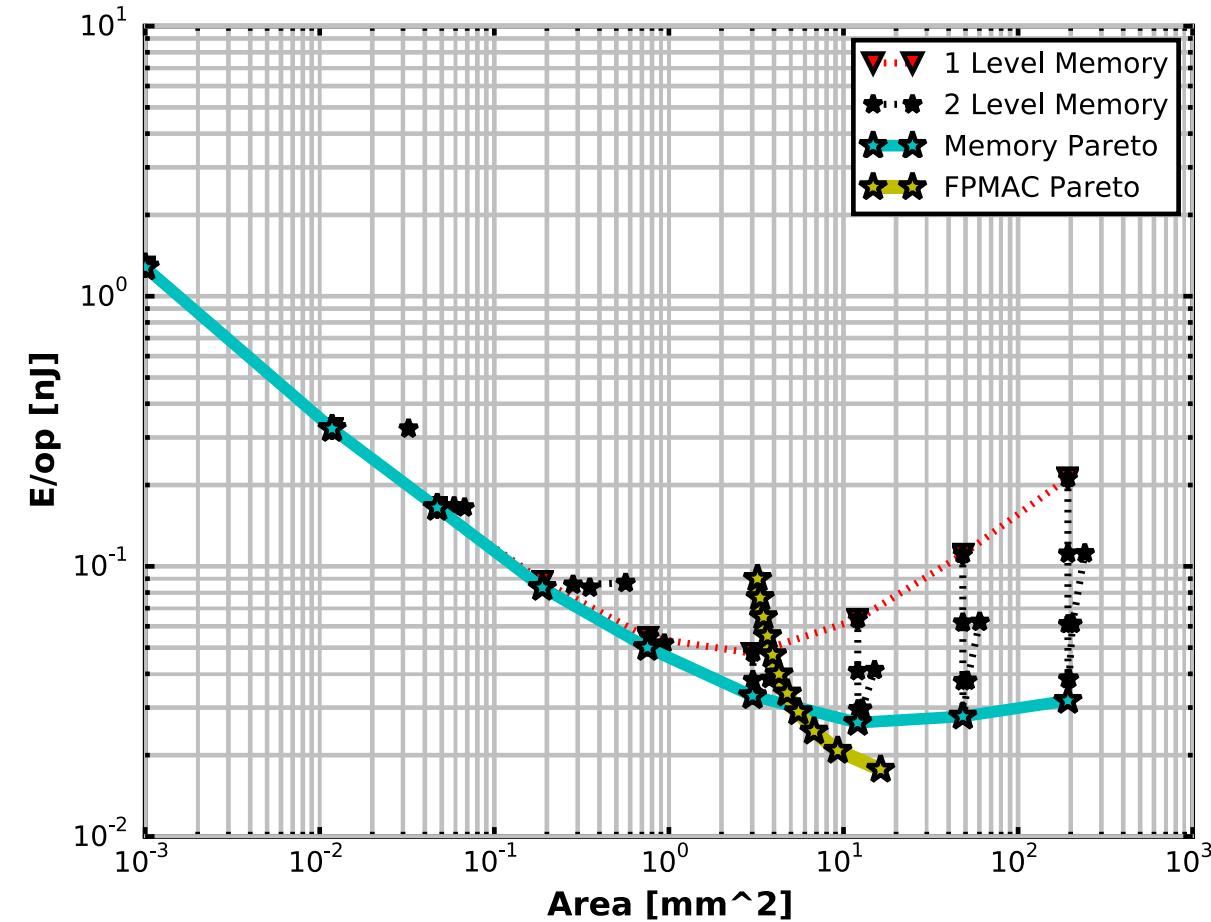
- Energy Consumption
 - Boosts significantly as size grows
- Memory Size:
 - Does not compensate for the access savings

Let's Add Another Layer

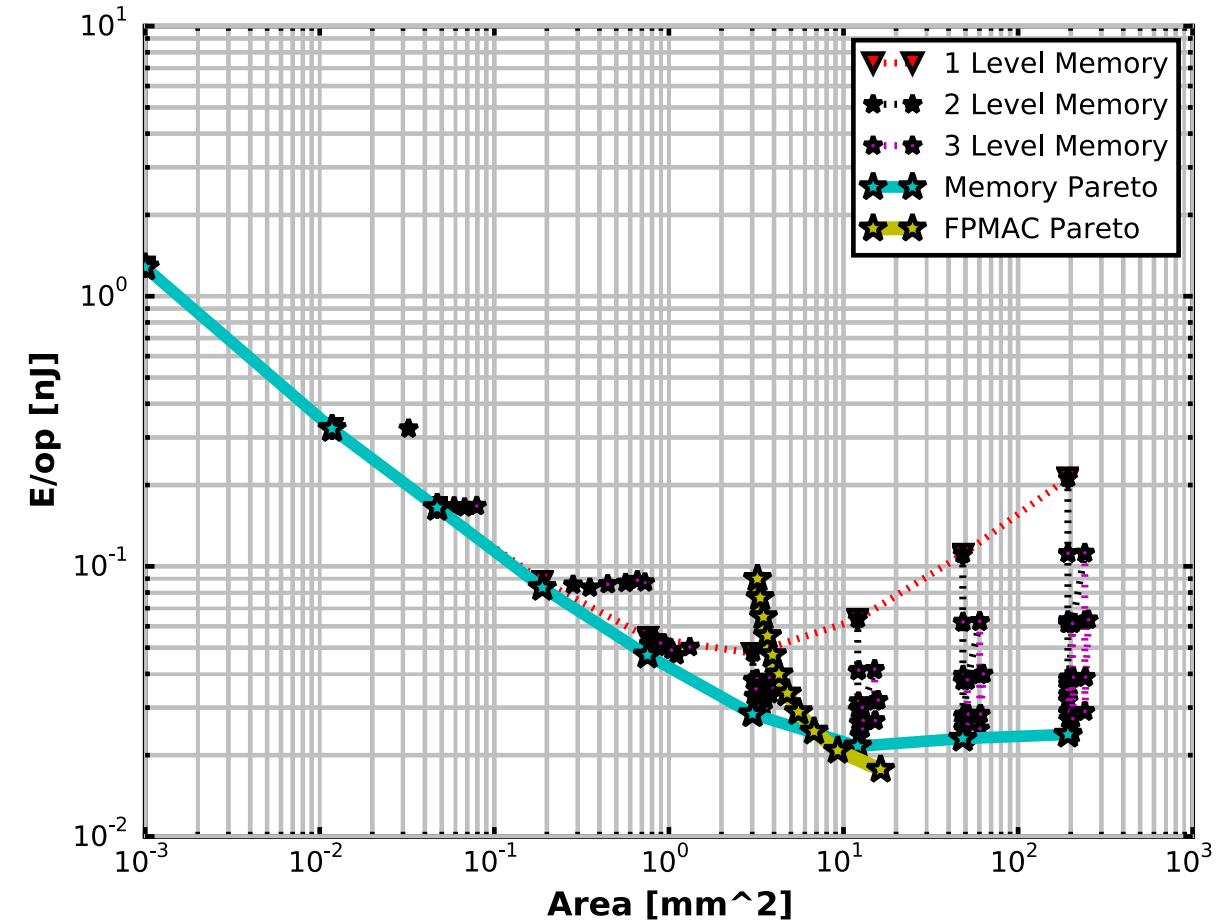
- A or B
 - From $b \times b$ buffer
 - N^3/b
- C
 - Access for last level cache is $2N^2$



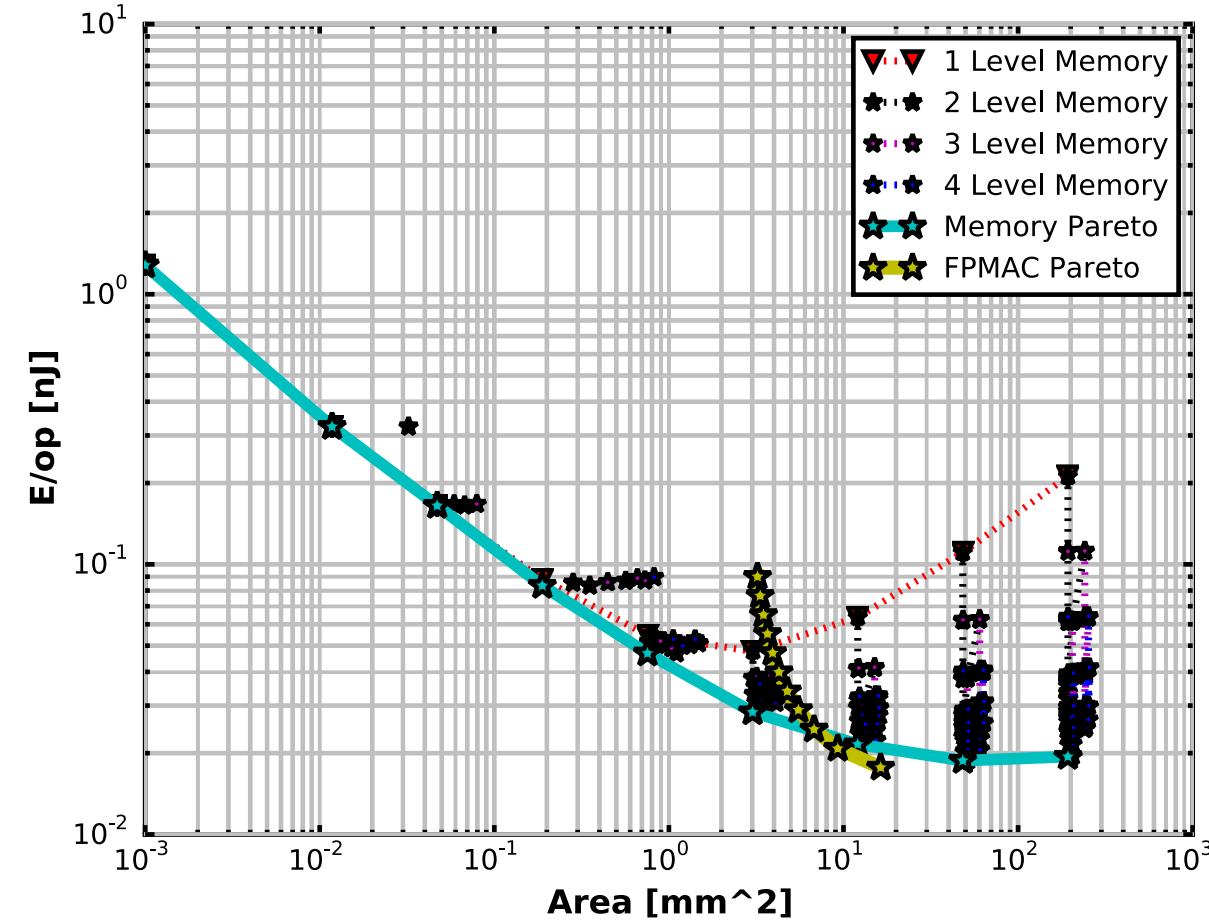
Second-level Caching



Three Levels

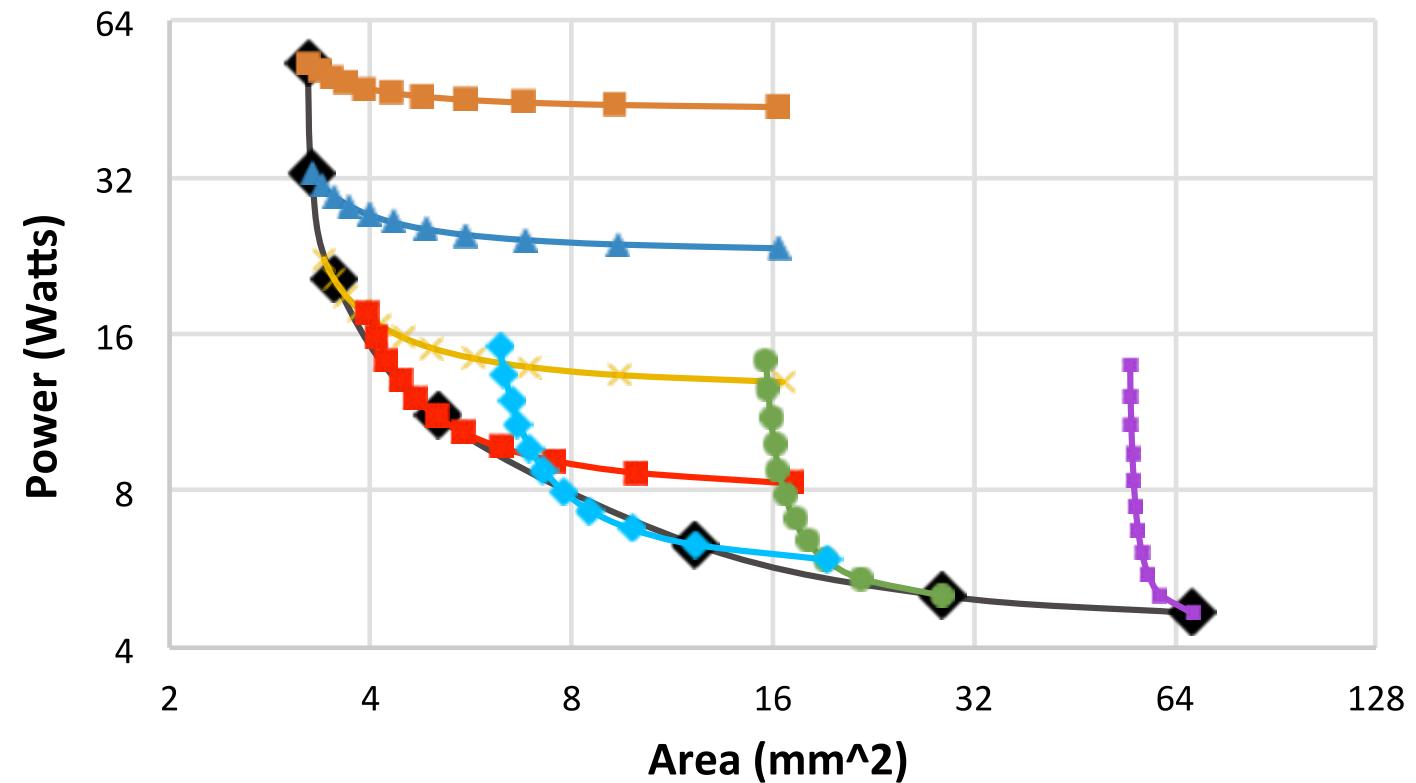


Memory Subsystem = Half Energy



Putting It All Together

- Merging FPU Pareto with Memory Pareto



Algorithm/Architecture Co-design Methodology

