6.5930/1

Hardware Architectures for Deep Learning

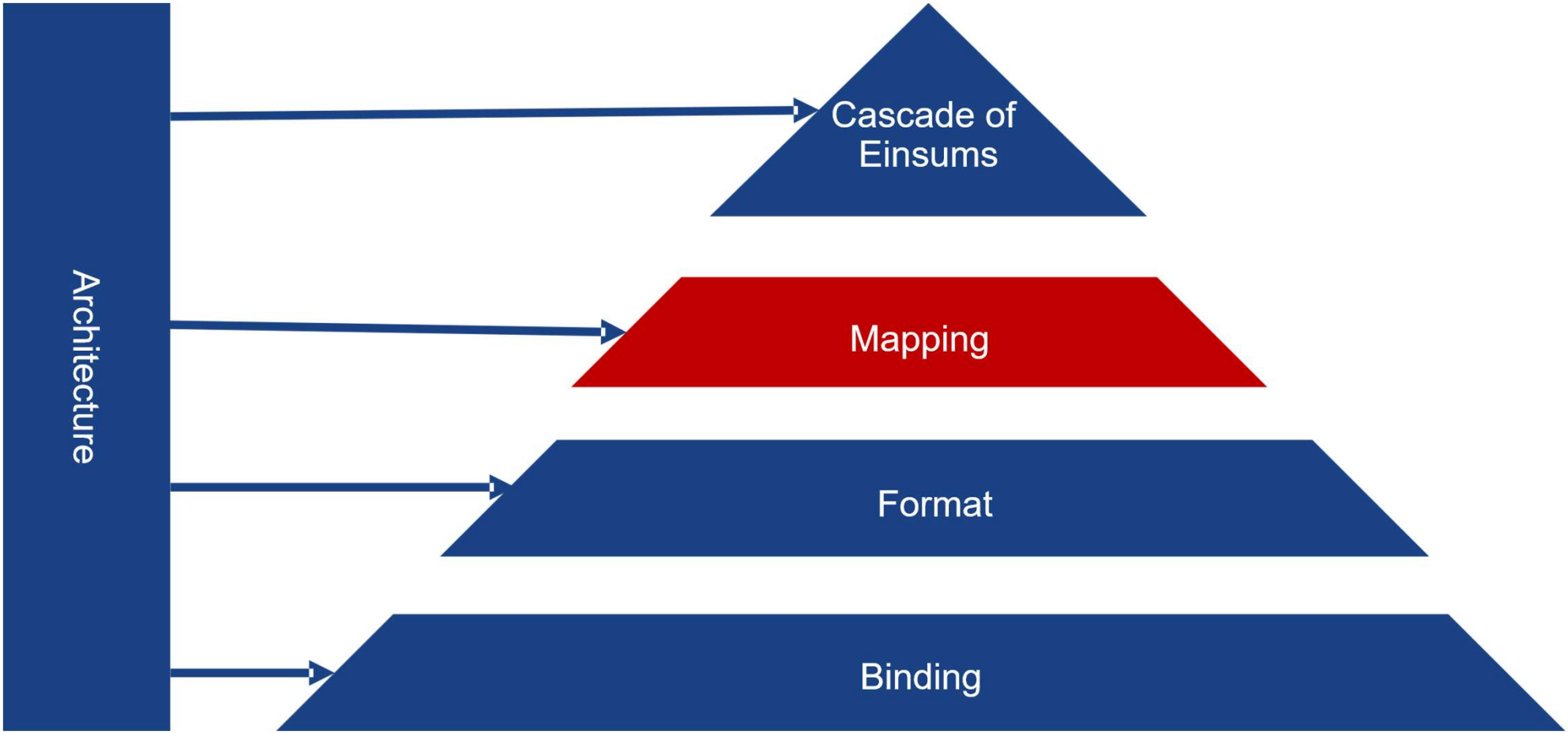
# Mapping - Dataflows

February 17, 2026

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

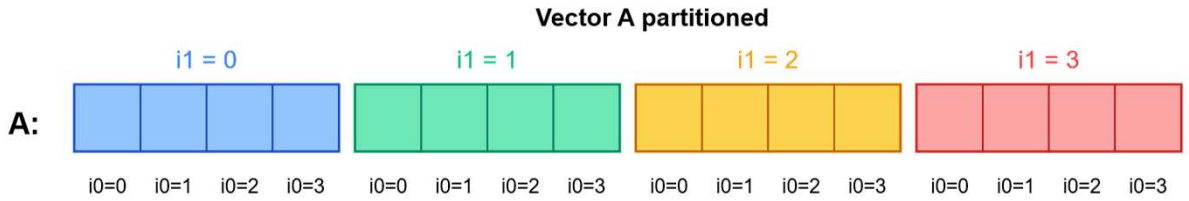
# Separate of Concerns - Mapping



# Aspects of Mapping

- **Partitioning**
  - **Goal:** Create chunks of data to enable control over temporal reuse in hierarchy of buffers and spatially for use by multiple processors
  - **Impact on loop nest:** Partition ranks add loops in the loop nest → each loop controls use (and reuse) of elements of the tensor in buffers or spatially

$$A_{\mathrm{i}} \rightarrow A_{i1,\mathrm{i0}}$$



# Aspects of Mapping

- **Dataflow**
  - **Goal:** Increase reuse for given data type (e.g., weight, input, output) by increasing stationarity. Align traversal order with storage order for concordant traversal (improves spatial locality); **(this class!)**
  - **Impact on loop nest:** Sets the order of the *for* loops. Data type whose ranks are the outer loops is the most stationary
- **Dataplacement**
  - **Goal:** Control placement of data in specific buffers to reduce data movement (including bypass)
  - **Impact on loop nest:** Augment information in the loop nest to show where data is held

# Aspects of Mapping

- **Compute Placement** (parallel or temporal loop)
  - **Goal:** Reduce cycles by processing multiple MACs in parallel
    - May also impact spatial data reuse (if same data reused across parallel MACs)
  - **Impact on loop nest:** Parallelism is indicated the use of a *parallel_for* loop (spatial partitioning)
- **Partition Sizing**
  - **Goal:** Determine the exact amounts of data to be processed both temporarily and spatially
  - **Impact on loop nest:** Sets the limits on the loop nests

# Goals of Today's Lecture

- Impact of data movement and memory hierarchy on energy consumption
- Taxonomy of dataflows for CNNs
  - Output Stationary
  - Weight Stationary
  - Input Stationary

# Background Reading

- **DNN Accelerators**
  - ***Efficient Processing of Deep Neural Networks***
    - Chapter 5 – thru 5.7.1
    - Chapter 5 – 5.8

*All these books and their online/e-book versions are available through MIT libraries.*

# Dataflow and Memory Hierarchy

# 1-D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

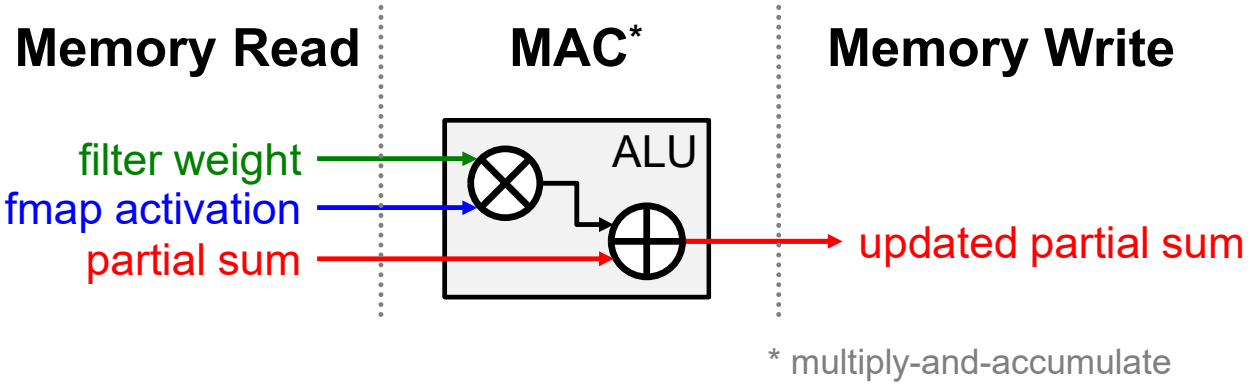
Operational definition of Einsums says traverse all valid values of “q” and “s.

Note how for a given value of q the I tensor will vary over a window of size S.

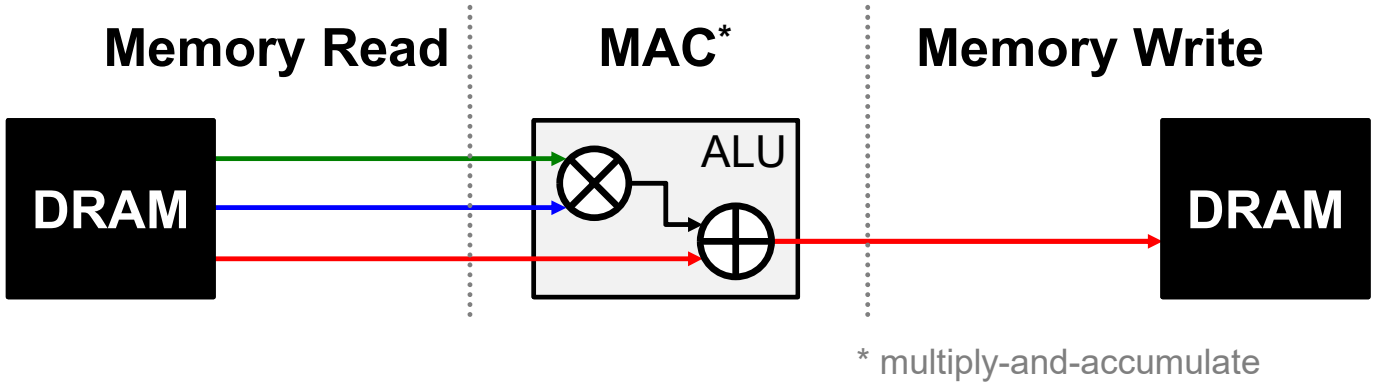
How much data needs to be moved to/from a single MAC unit to perform this computation?

4 *Q * S

# Memory Access is the Bottleneck



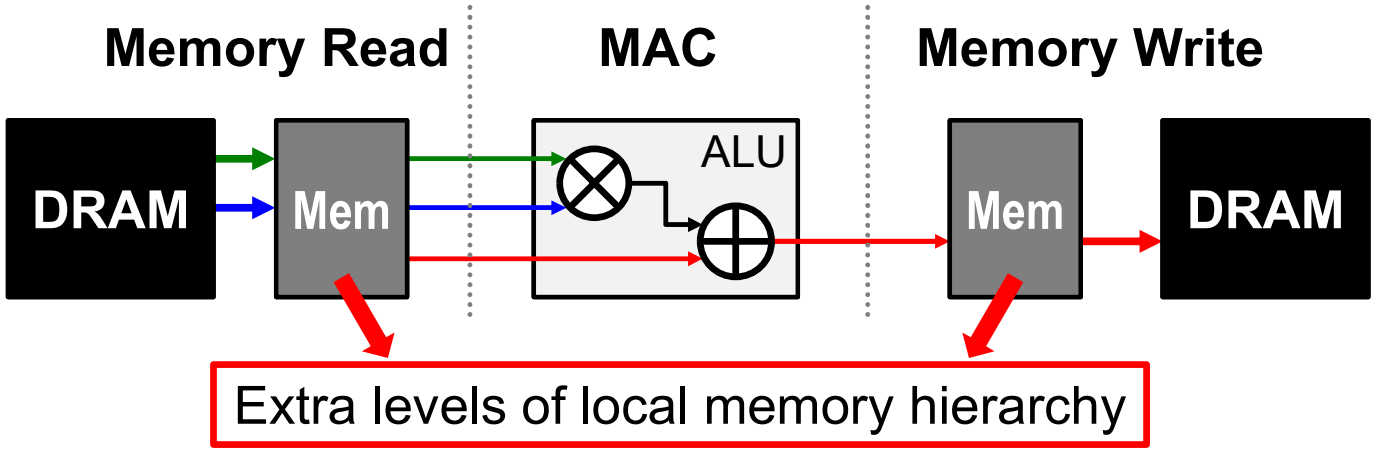
# Memory Access is the Bottleneck




Worst Case: all memory R/W are **DRAM** accesses

- Example: AlexNet [NeurIPS 2012] has **724M** MACs
  → **2896M** DRAM accesses required

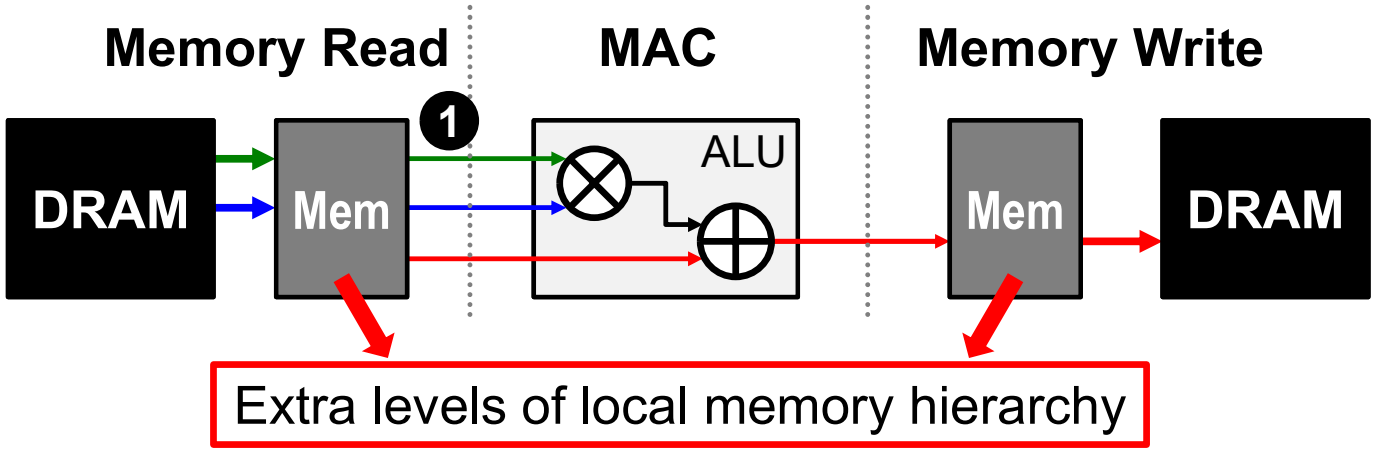
# Memory Access is the Bottleneck




Under what circumstances will these extra levels help?

Computational intensity > 1

# Memory Access is the Bottleneck

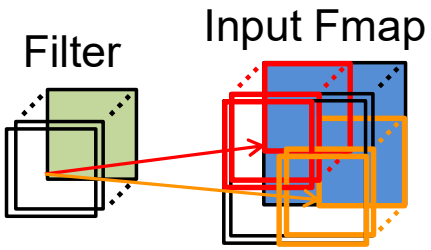



Opportunities: 1 **data reuse**

# Types of Data Reuse in DNN

**Convolutional Reuse**

CONV layers only
(sliding window)

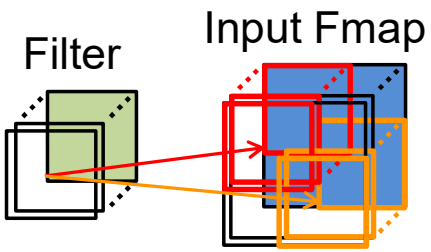



Reuse: Activations
Filter weights

# Types of Data Reuse in DNN

## Convolutional Reuse

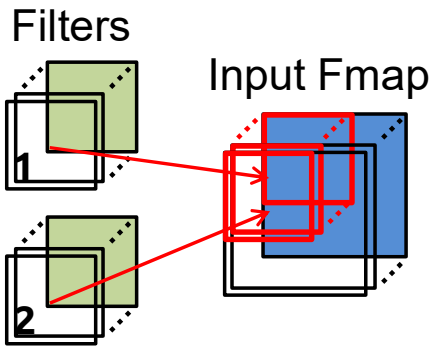
CONV layers only
(sliding window)




Reuse: Activations
Filter weights

## Fmap Reuse

CONV and FC layers

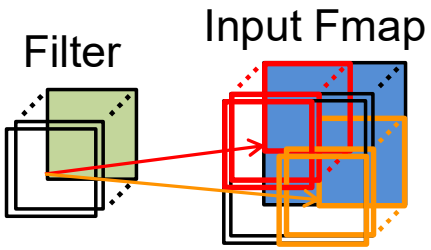



Reuse: Activations

# Types of Data Reuse in DNN

## Convolutional Reuse

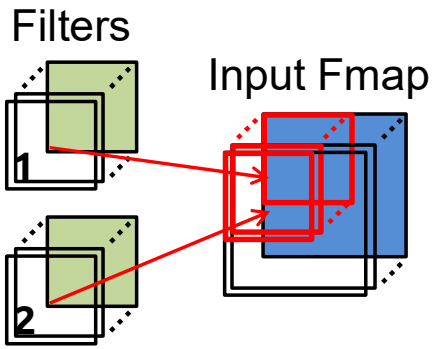
CONV layers only
(sliding window)




Reuse: Activations
Filter weights

## Fmap Reuse

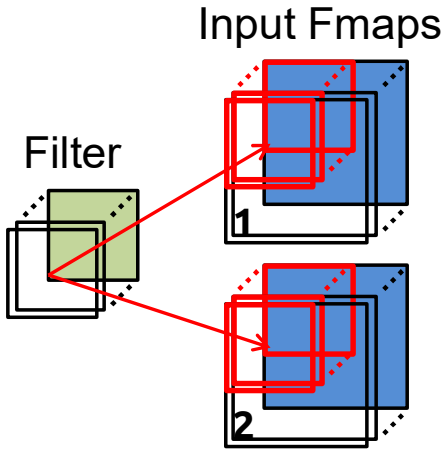
CONV and FC layers




Reuse: Activations

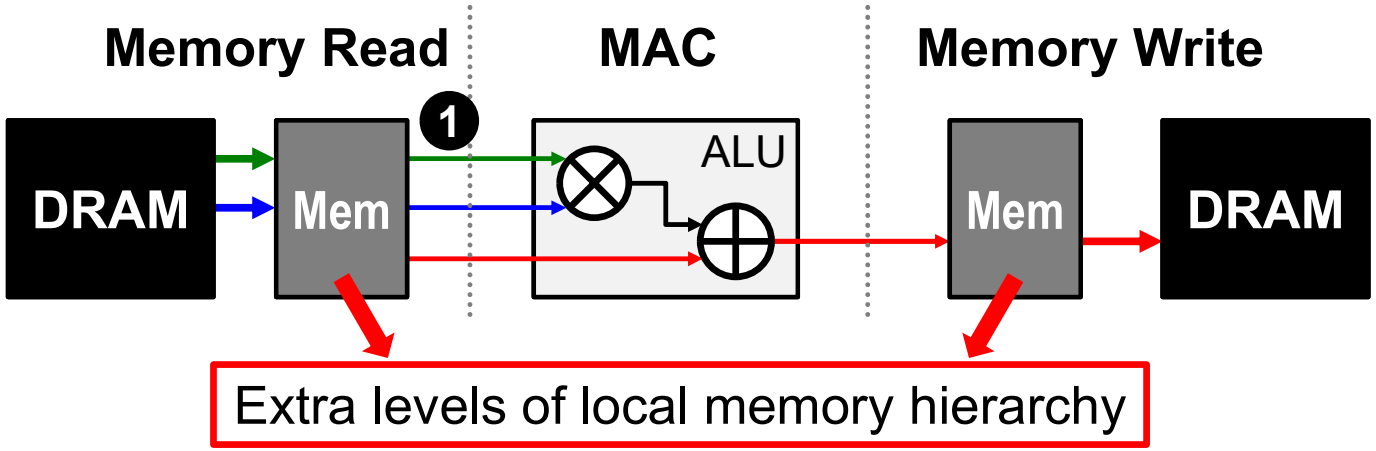
## Filter Reuse

CONV and FC layers
(batch size > 1)




Reuse: Filter weights

# Memory Access is the Bottleneck

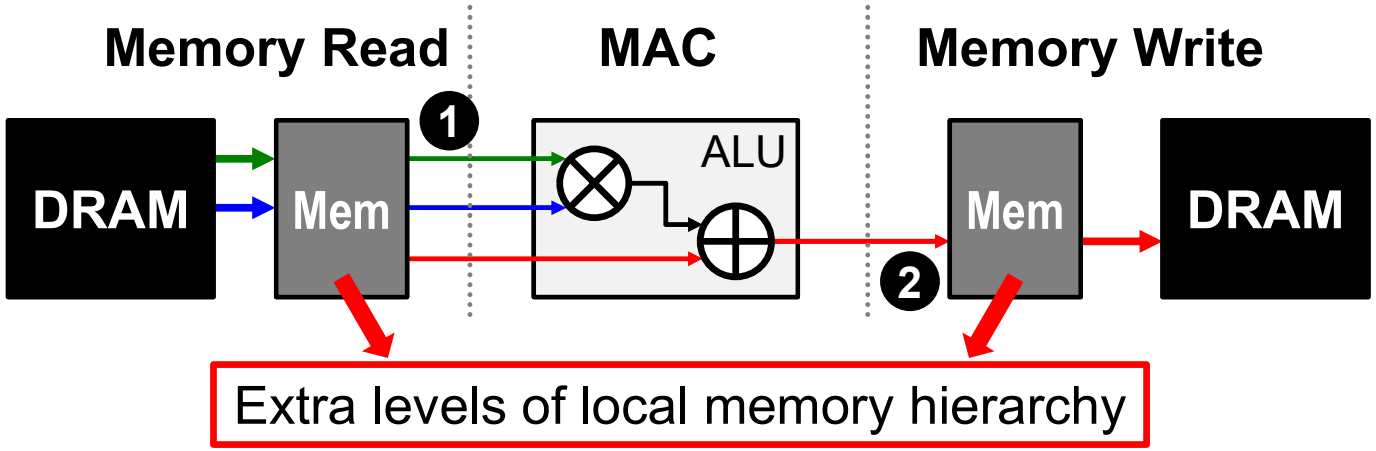



Opportunities: 1 **data reuse**

1 Can reduce DRAM reads of filter/fmap by up to **500×****

** AlexNet CONV layers

# Memory Access is the Bottleneck

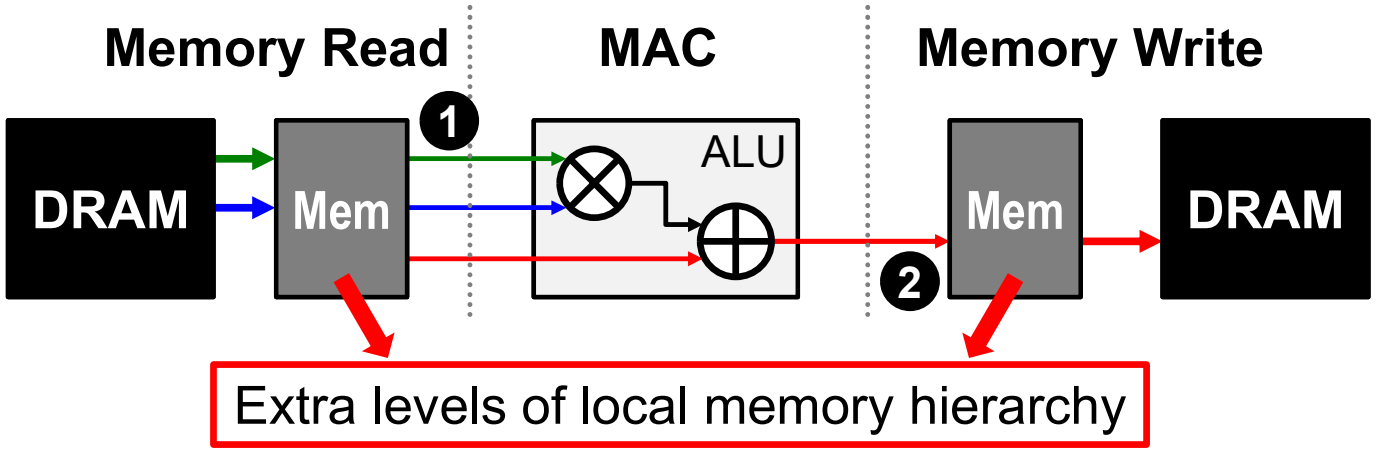



Opportunities: ❶ **data reuse** ❷ **local accumulation**

❶ Can reduce DRAM reads of filter/fmap by up to **500×**

❷ Partial sum accumulation does **NOT** have to access DRAM

# Memory Access is the Bottleneck



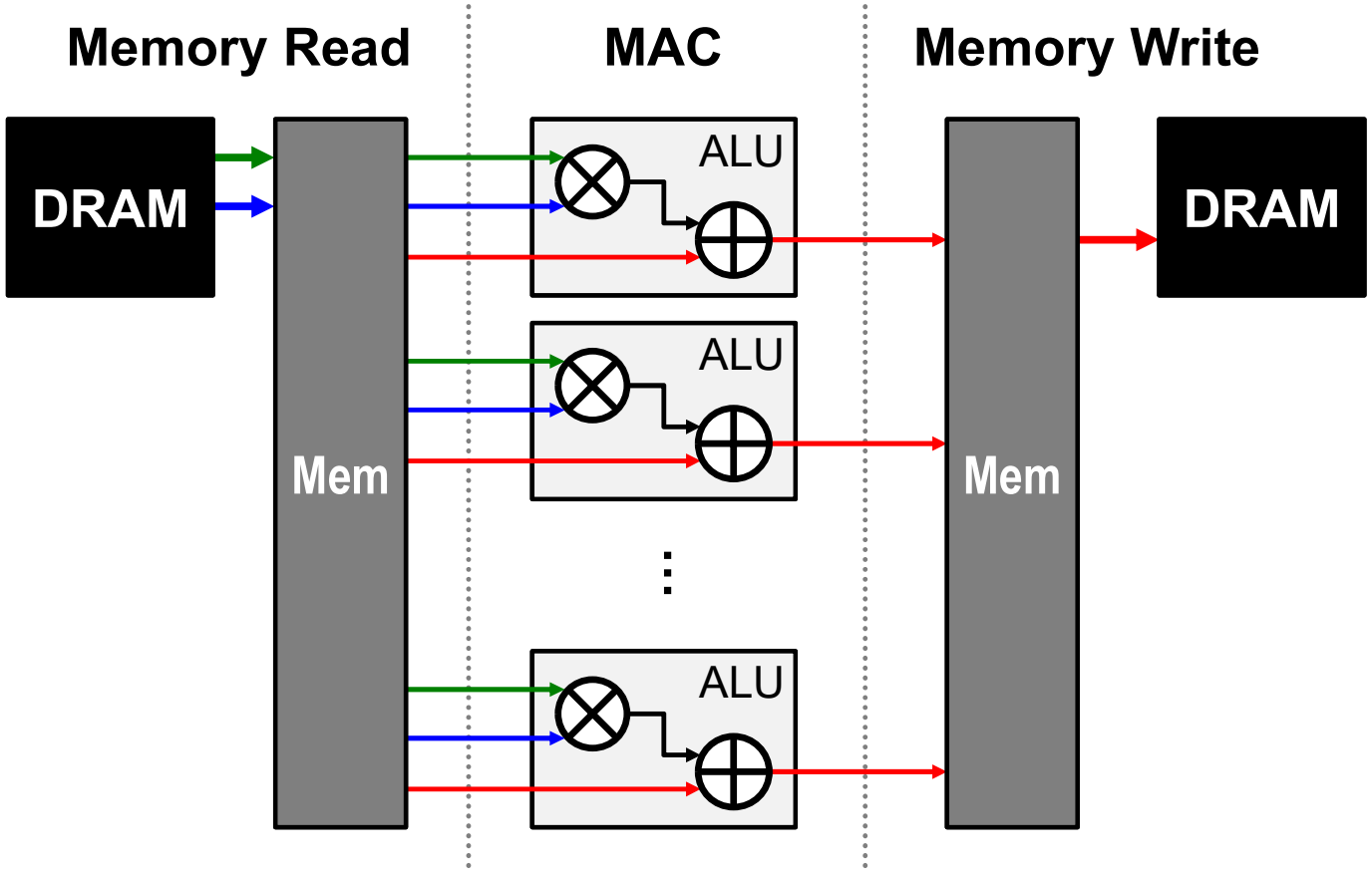

Opportunities: ❶ **data reuse** ❷ **local accumulation**

❶ Can reduce DRAM reads of filter/fmap by up to **500×**

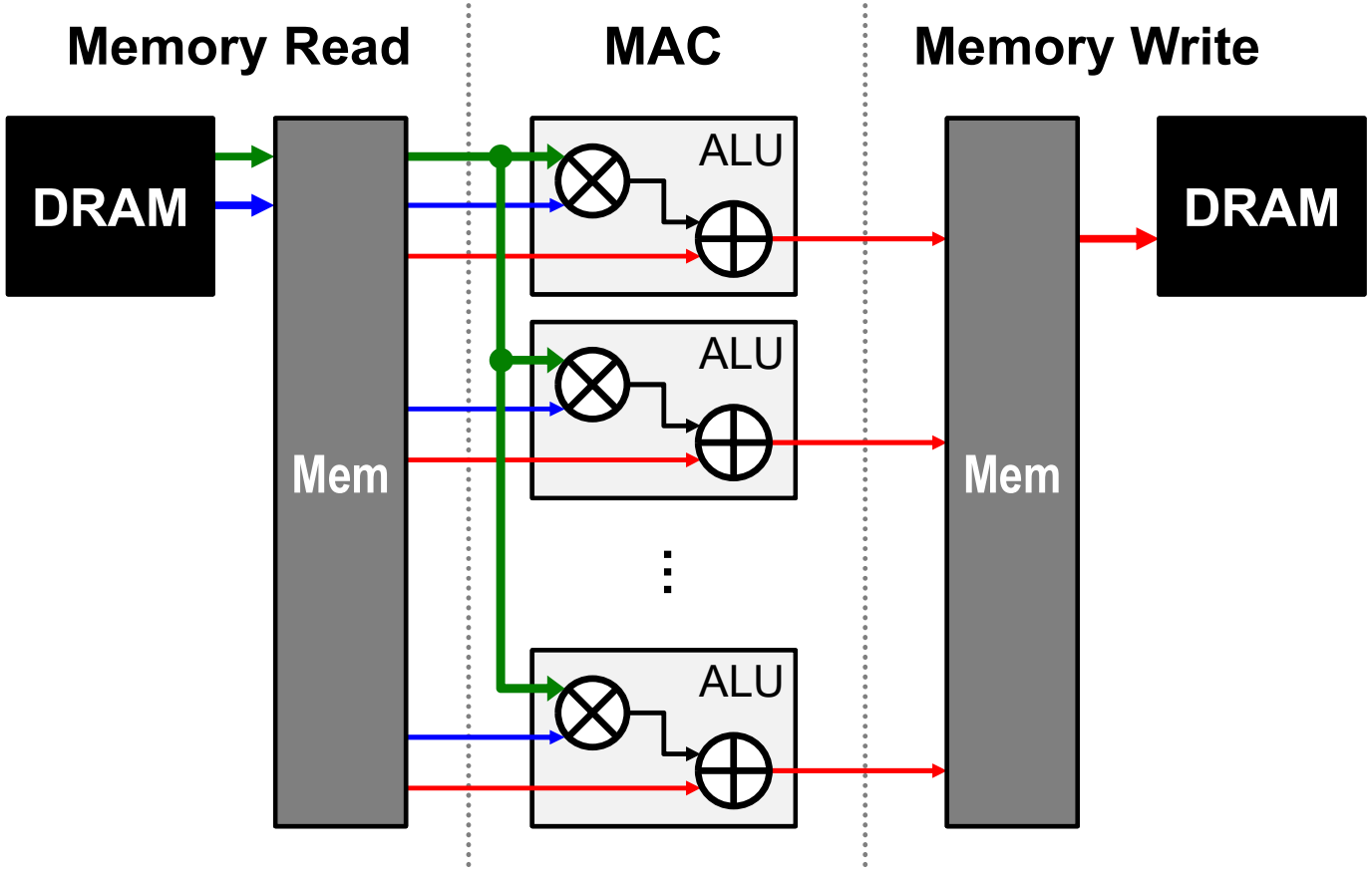
❷ Partial sum accumulation does **NOT** have to access DRAM

- Example: DRAM access in AlexNet can be reduced from **2896M** to **61M** (best case)

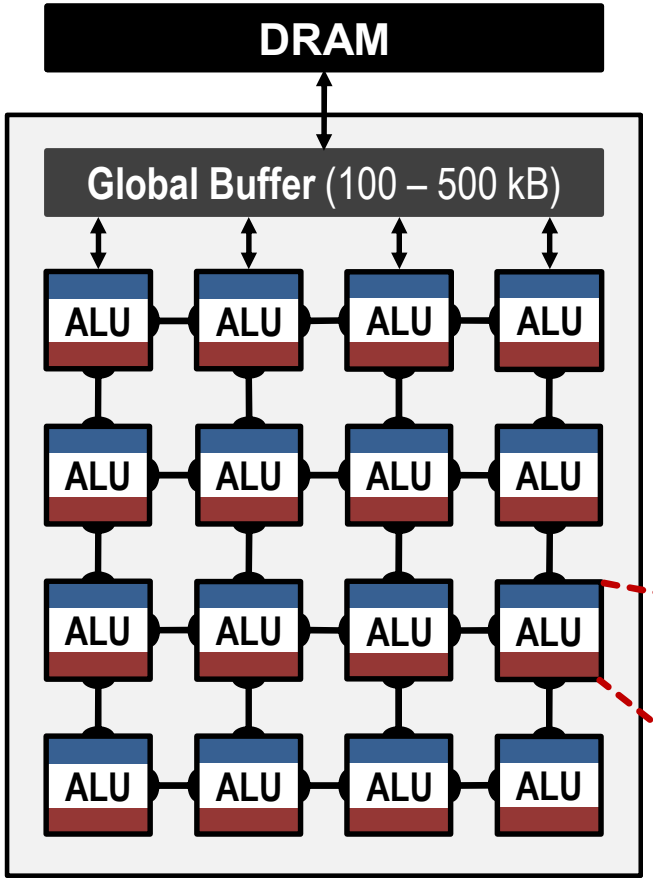
# Leverage Parallelism for Higher Performance



# Leverage Parallelism for *Spatial* Data Reuse



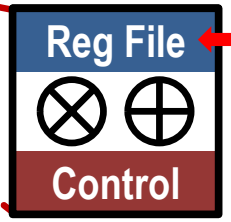
# Spatial Architecture for DNN




**Local Memory Hierarchy**

- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

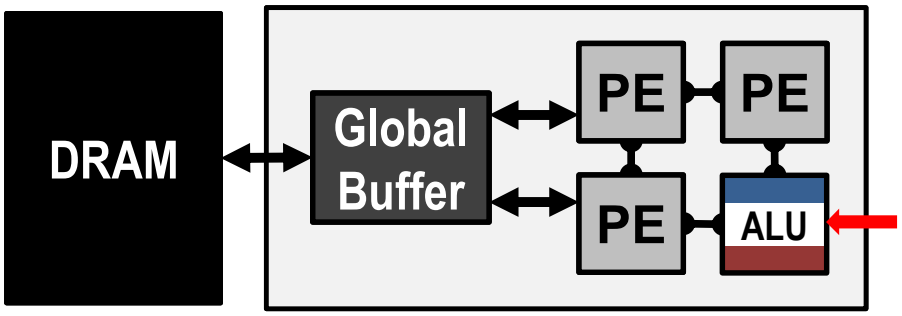
**Processing Element (PE)**

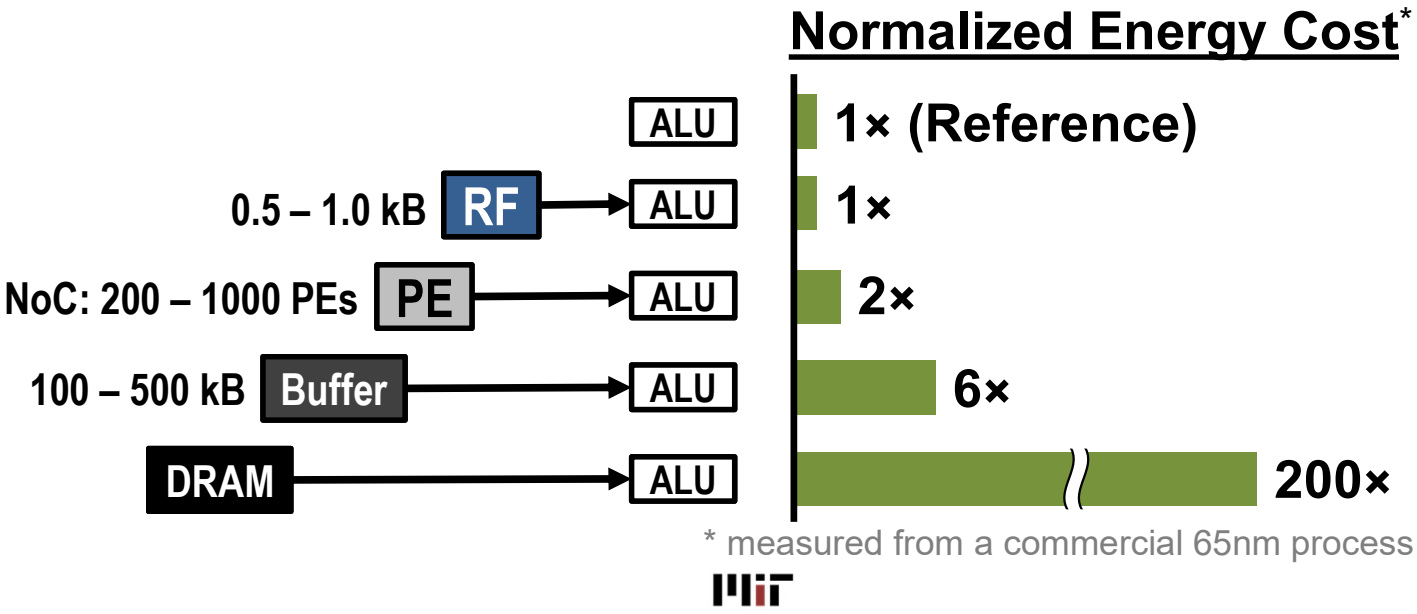



0.5 – 1.0 kB

*Note: Register File is also referred to as local buffer in Labs

# Low-Cost Local Data Access

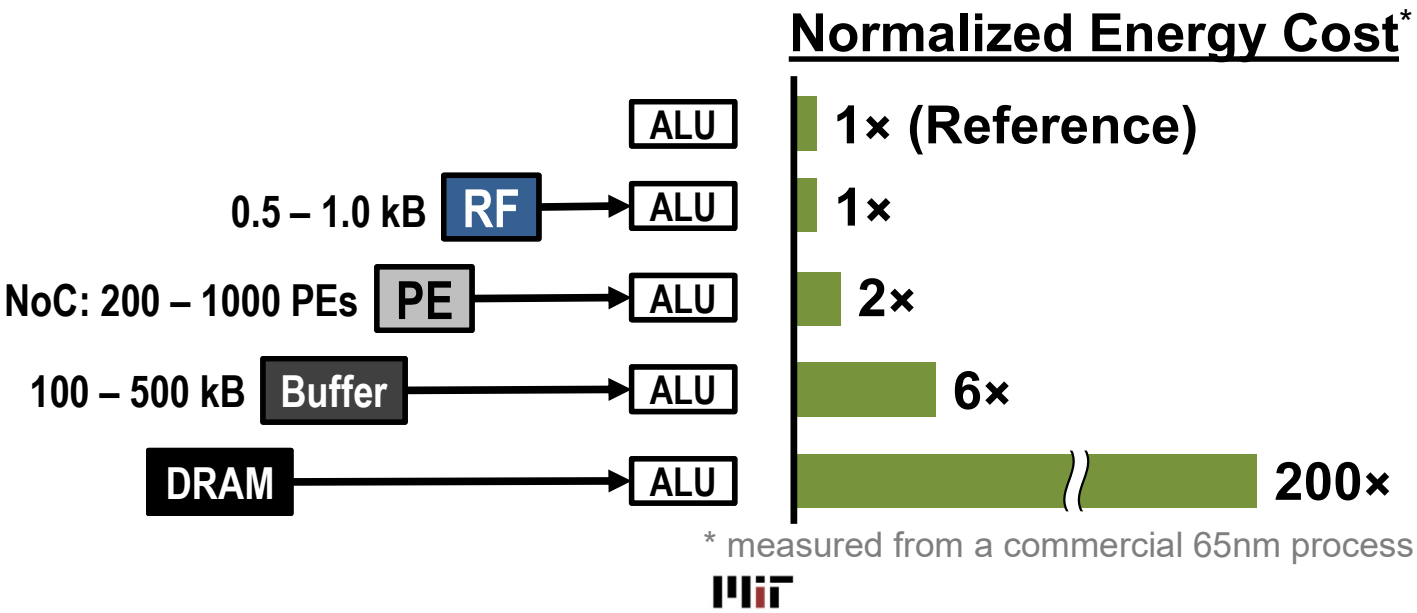



* measured from a commercial 65nm process

# Low-Cost Local Data Access

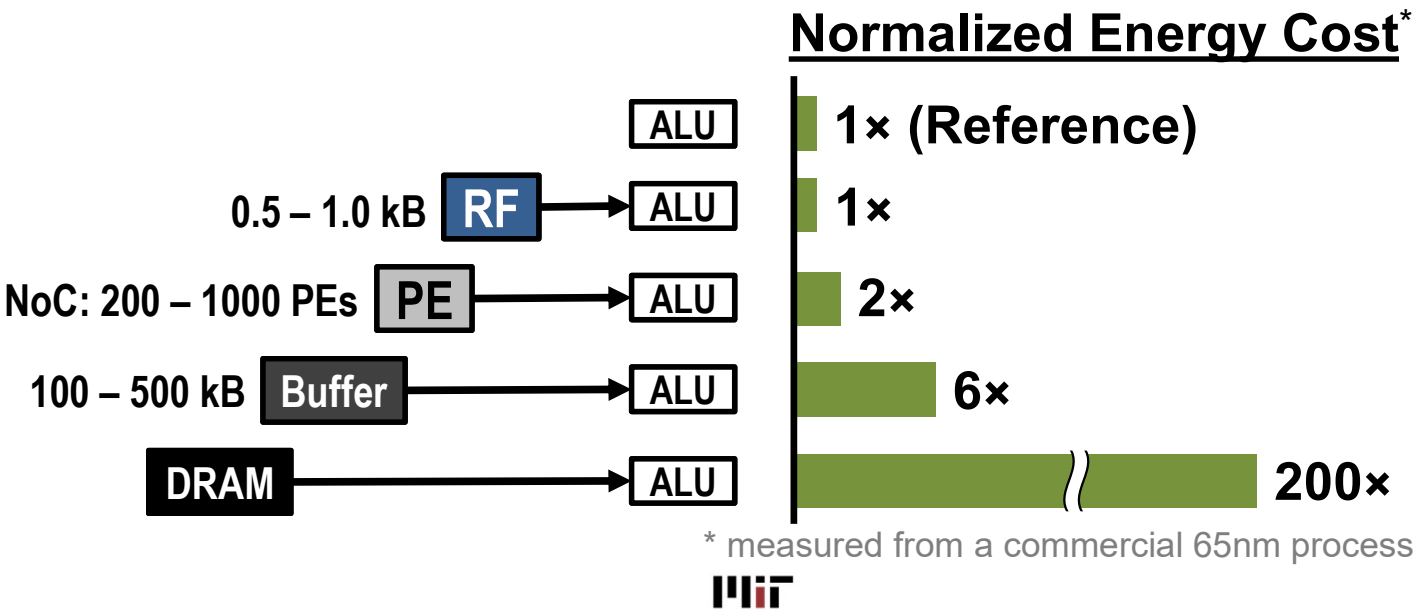
How to exploit ❶ **data reuse** and ❷ **local accumulation** with *limited* low-cost local storage?



# Low-Cost Local Data Access

How to exploit ❶ **data reuse** and ❷ **local accumulation** with *limited* low-cost local storage?

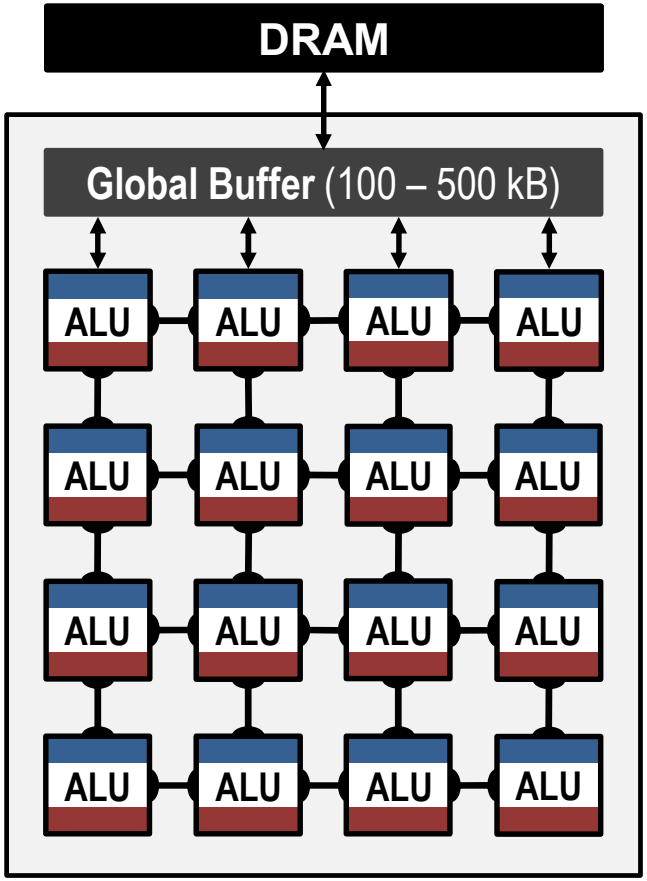
specialized **processing dataflow** required!




* measured from a commercial 65nm process

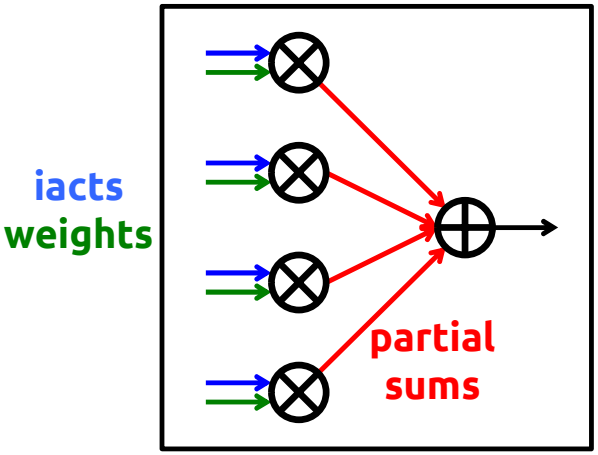
# Dataflow and Dataplacement

- Dataflow controls order of processing
  - Increase reuse/accumulation at low-cost levels of memory hierarchy
- Dataplacement of partitioned tensors enables the tensor to fit into different levels of the memory hierarchy to increase locality
  - e.g., partition to fit global buffer



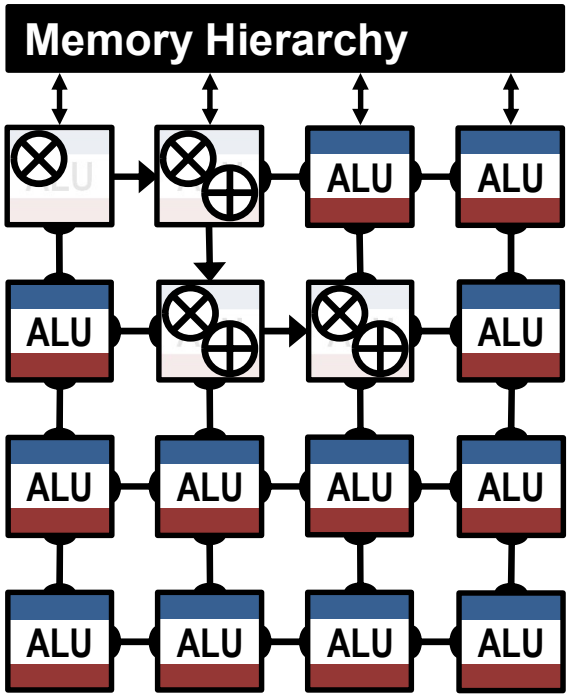
# How to Map the Dataflow?

CNN Convolution




?

Spatial Architecture
(Dataflow Processing)



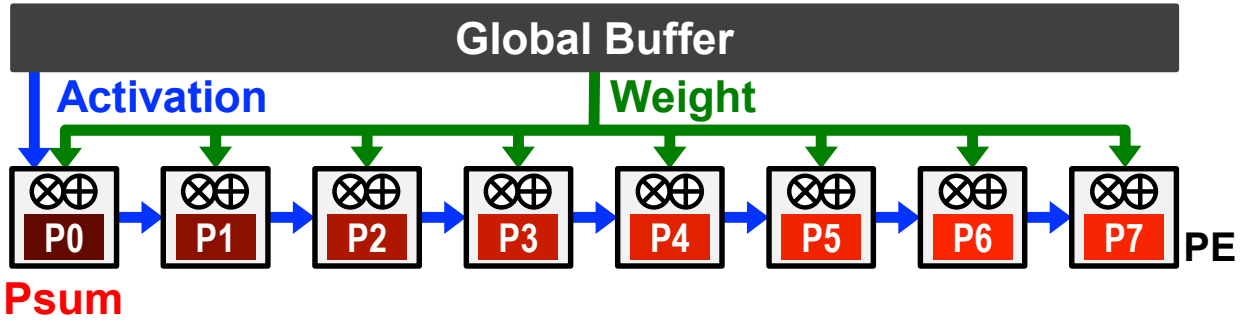

**Goal:** Increase reuse of input data (**input activations** and **weights**) and local **partial sums** accumulation

# Dataflow Taxonomy

- Output Stationary (OS)
- Weight Stationary (WS)
- Input Stationary (IS)

[**Chen**, *ISCA* 2016]

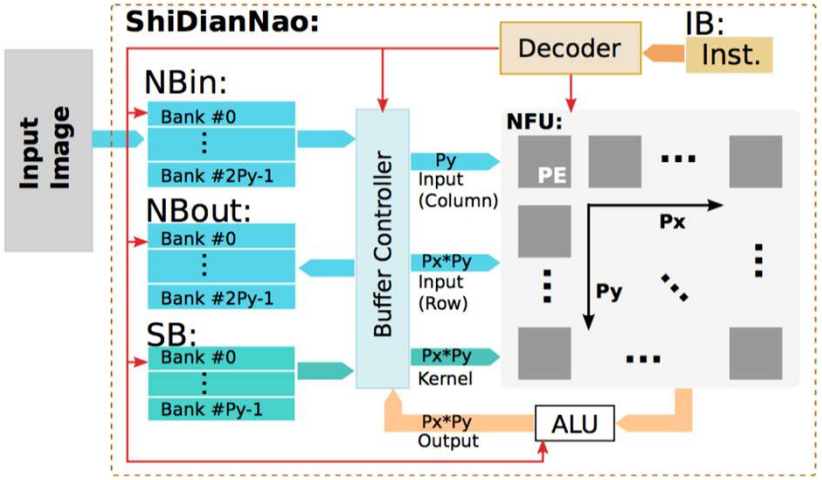
# Output Stationary (OS)



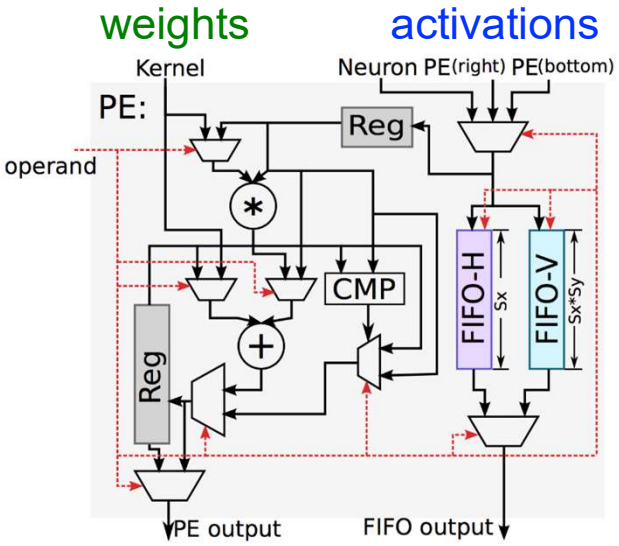

- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation
- **Broadcast/Multicast filter weights** and **reuse activations spatially** across the PE array

# OS Example: ShiDianNao

## Top-Level Architecture



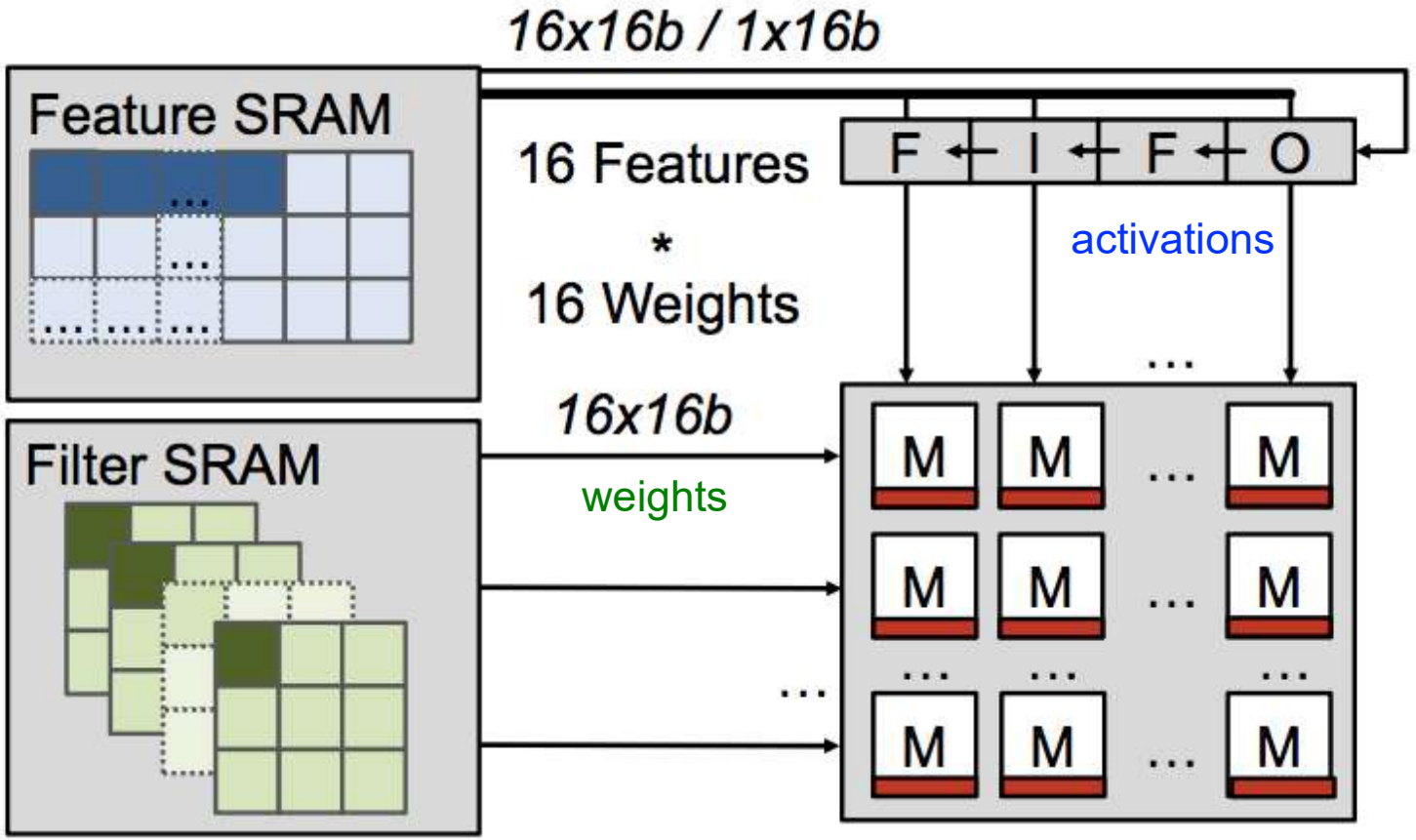

## PE Architecture




- Inputs streamed through array
- Weights broadcast
- Partial sums accumulated in PE and streamed out

[**Du**, *ISCA* 2015]

# OS Example: KU Leuven




[**Moons**, *VLSI* 2016], [**Moons**, *ISSCC* 2017]

# 1-D Convolution Einsum

$$O_q = I_{q+s} \times F_s$$

Operational definition of Einsum says traverse all valid values of “q” and “s”… but in what order….

Traversal order (fastest to slowest): S, Q

Which “for” loop is outermost? Q

# 1-D Convolution

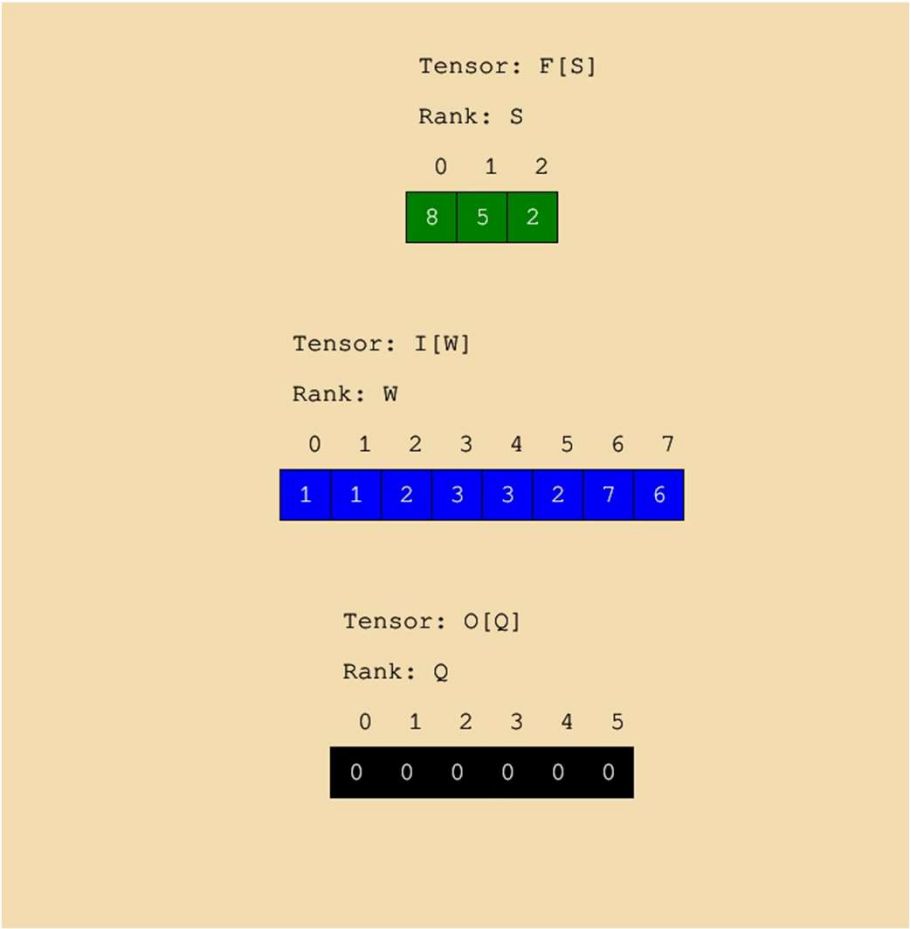

```
int i[W];     # Input activations
int f[S];     # Filter weights
int o[Q];     # Output activations

for q in [0, Q):
      for s in [0, S):
         o[q] += i[q+s]*f[s]
```

What dataflow is this? Output stationary

Is it easy to tell dataflow from the loop nest? Yes, from outermost loop index

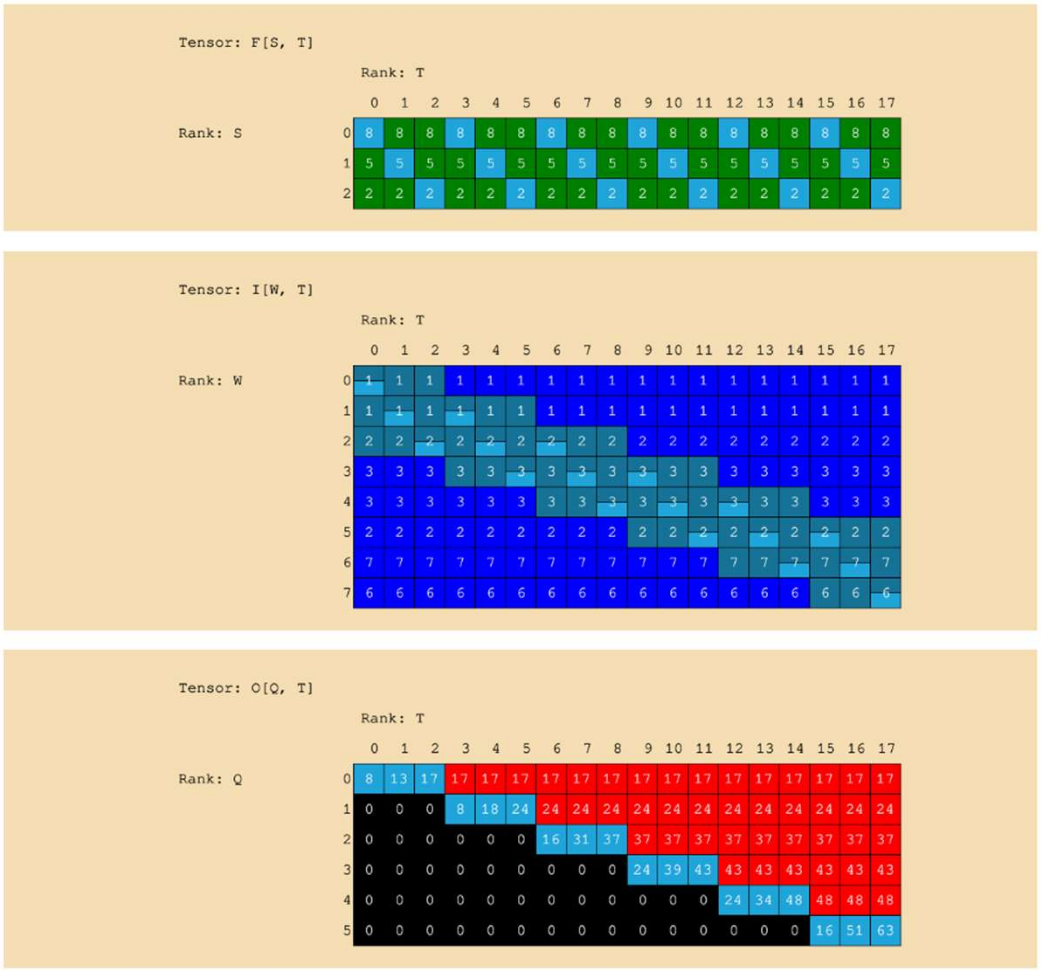
# Output Stationary - Animation

```
int i[W];     # Input activations
int f[S];     # Filter weights
int o[Q];     # Output activations

for q in [0, Q):
       for s in [0, S):
          o[q] += i[q+s]*f[s]
```




Full animation at https://csg.csail.mit.edu/6.5930/lectures/slidev/l05/#2

# Output Stationary – Spacetime View




Note: ‘Rank T’ here just meant time in cycles, not an actual rank in tensor

# CONV-layer Einsum

$$O_{m,p,q} = I_{c,p+r,q+s} \times F_{m,c,r,s}$$

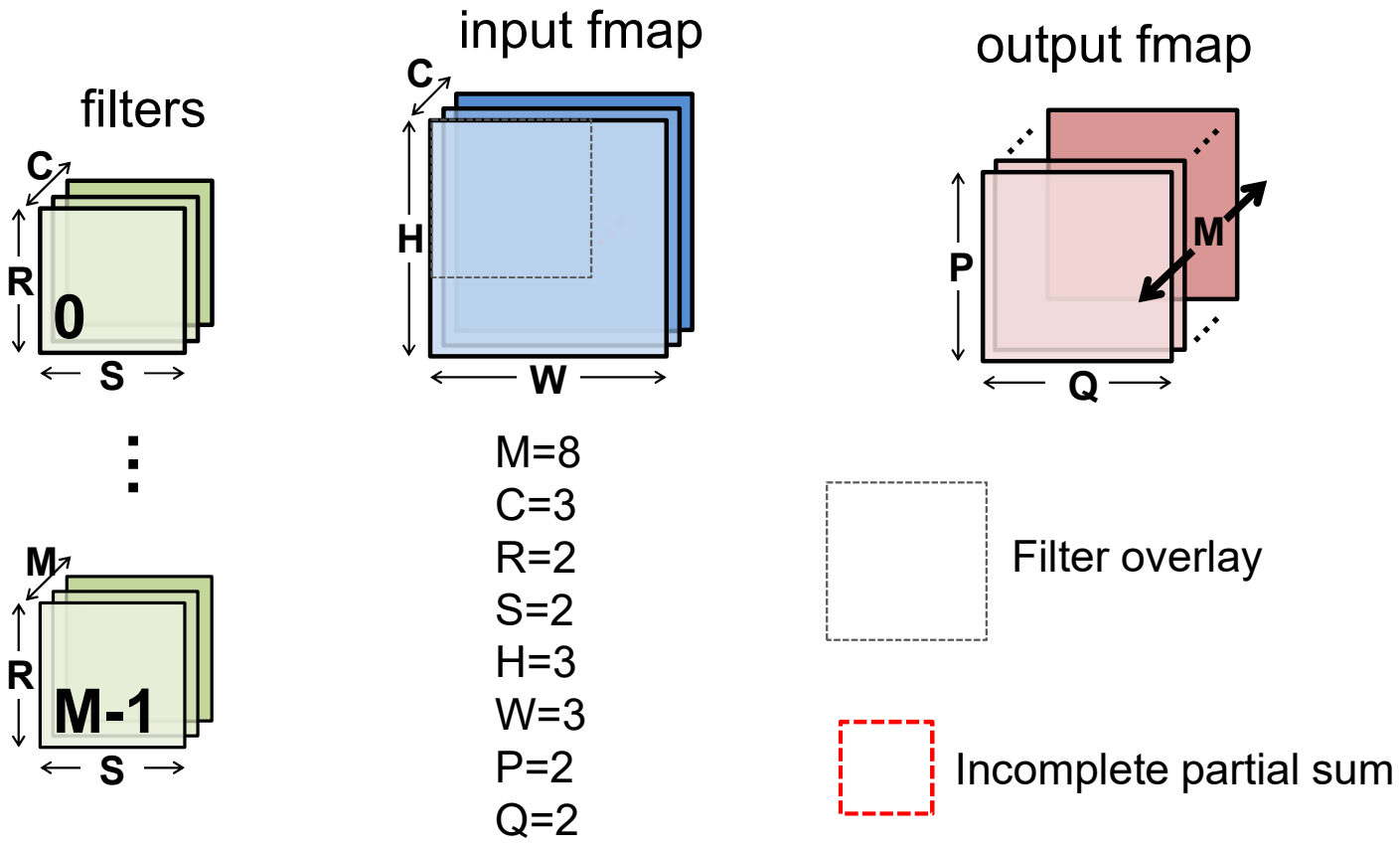
Traversal order (fastest to slowest): S, R, Q, P

Parallel Ranks: C, M

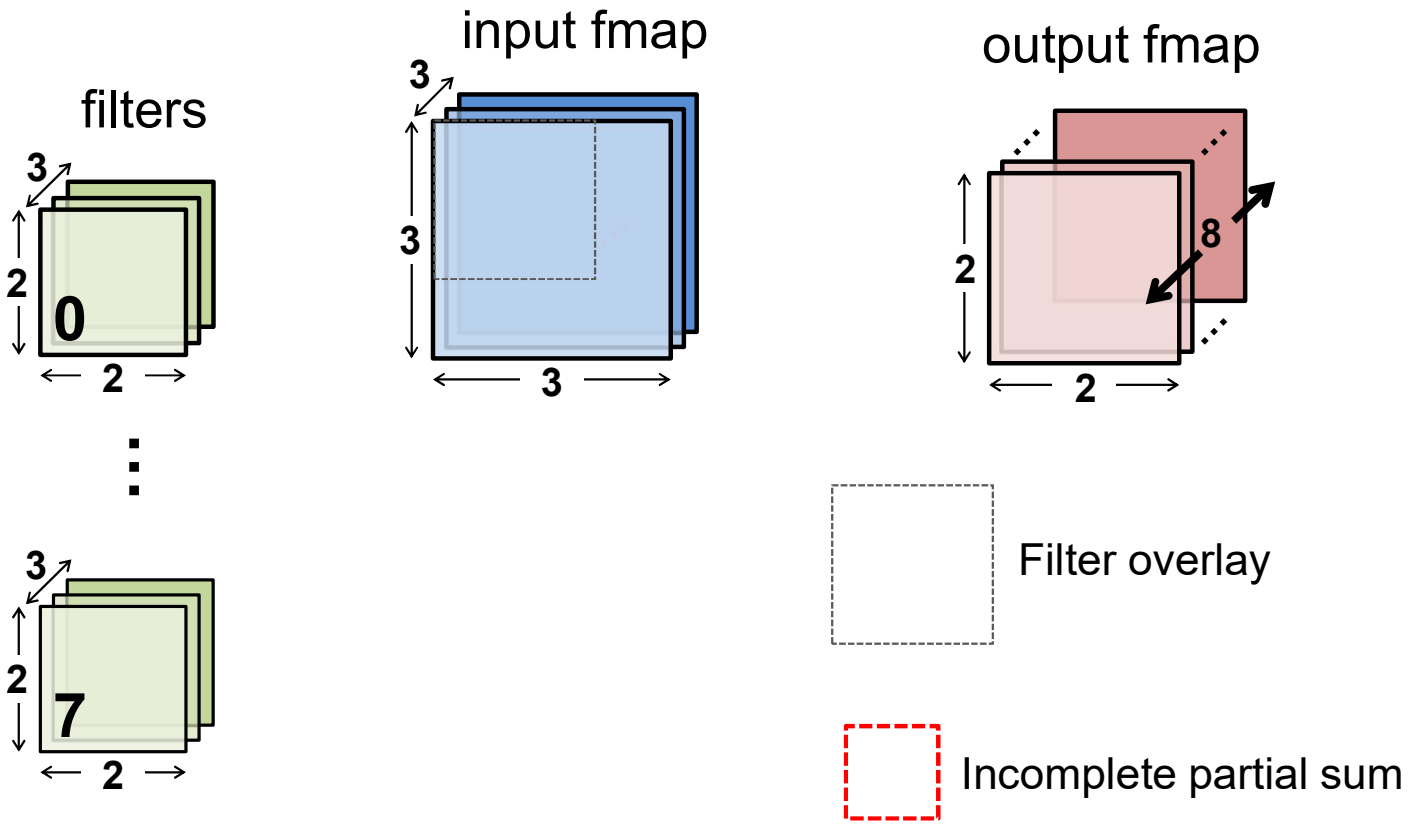
# CONV Layer OS Loop Nest

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

for p in [0, P):
  for q in [0, Q):
    for r in [0, R):
      for s in [0, S):
        parallel-for c in [0, C):
          parallel-for m in [0, M):
            o[m][p][q] += i[c][p+r][q+s]*f[m][c][r][s]
```

# CONV Layer OS Dataflow

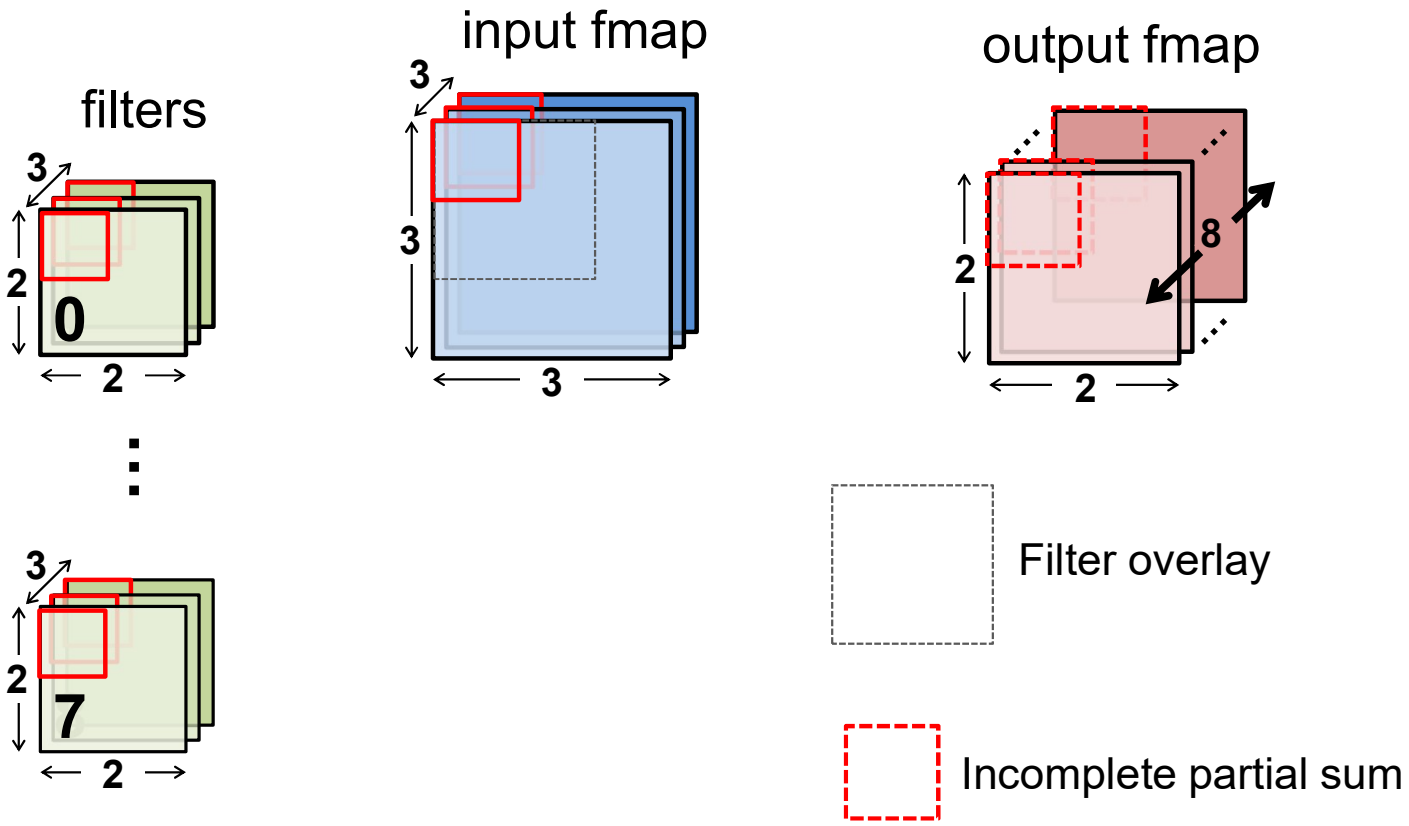


# CONV Layer OS Dataflow



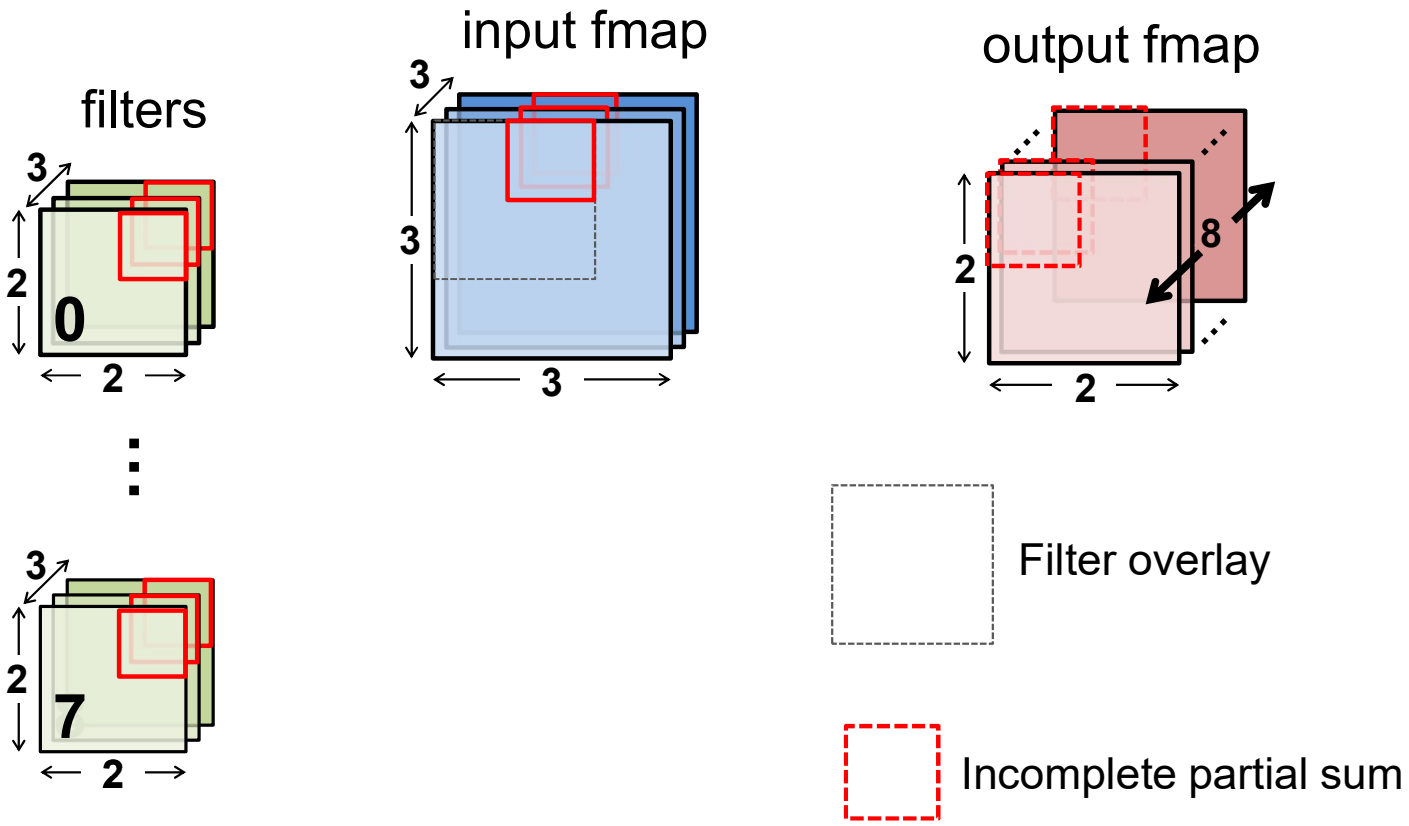
# CONV Layer OS Dataflow

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



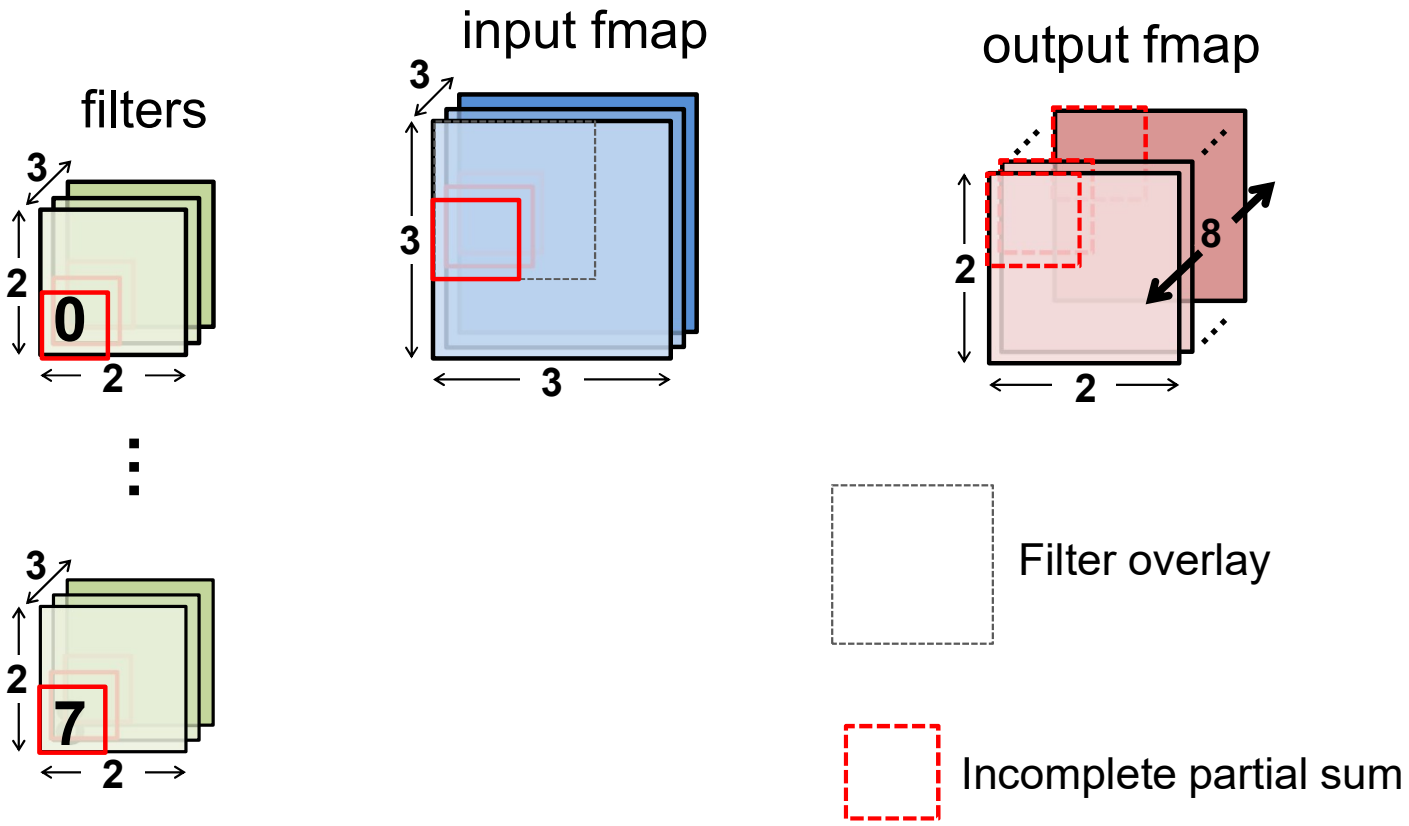
# CONV Layer OS Dataflow

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



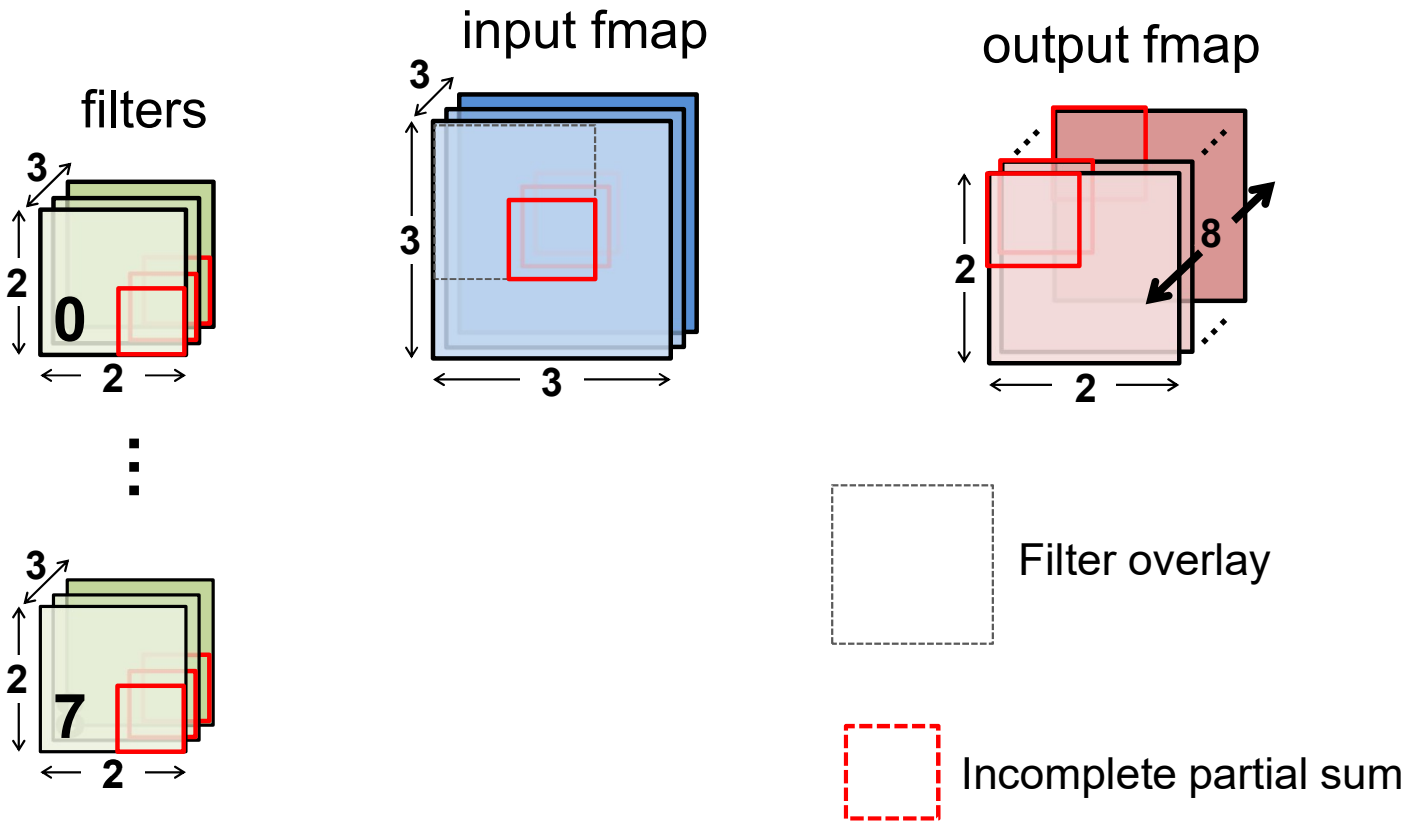
# CONV Layer OS Dataflow

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



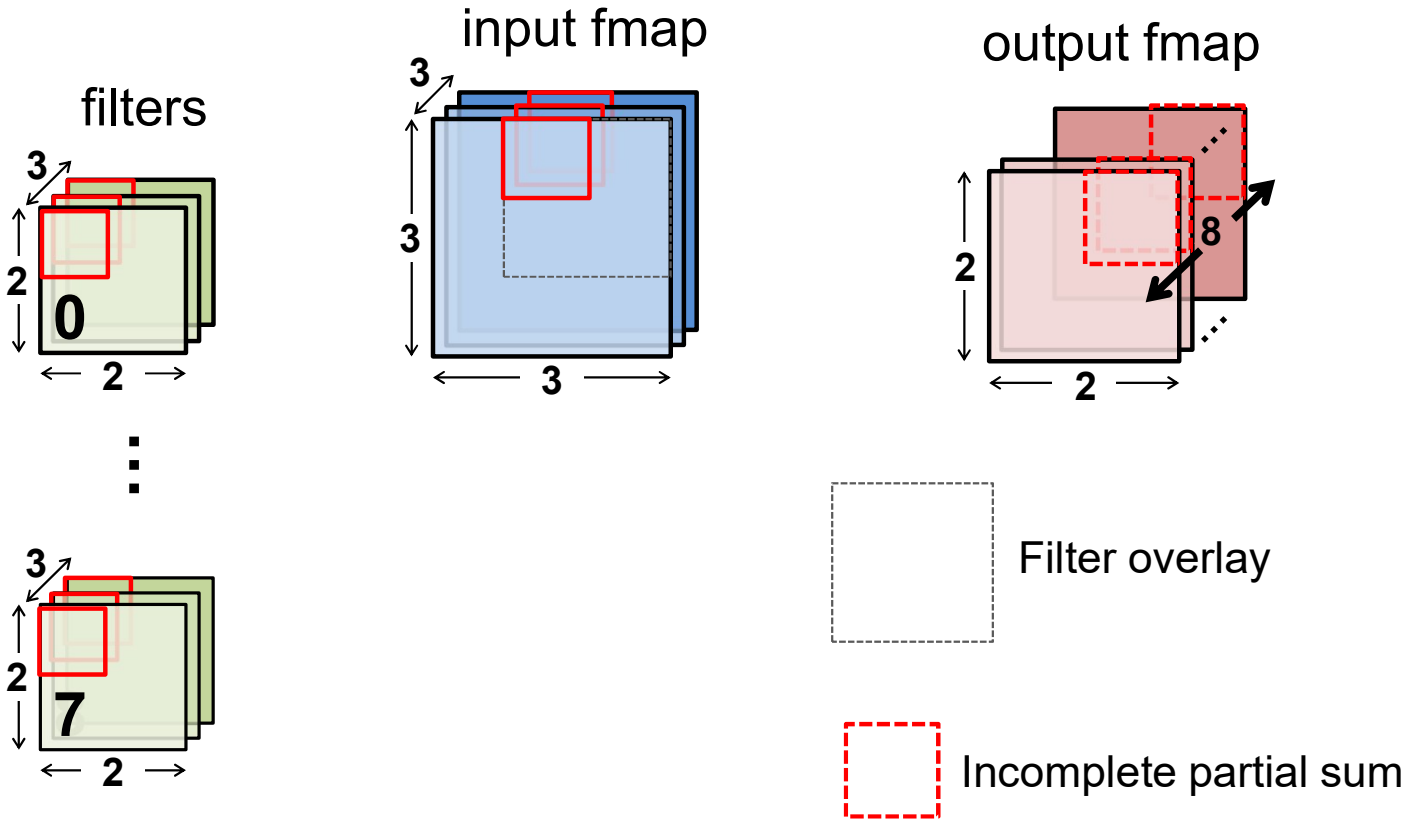
# CONV Layer OS Dataflow

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



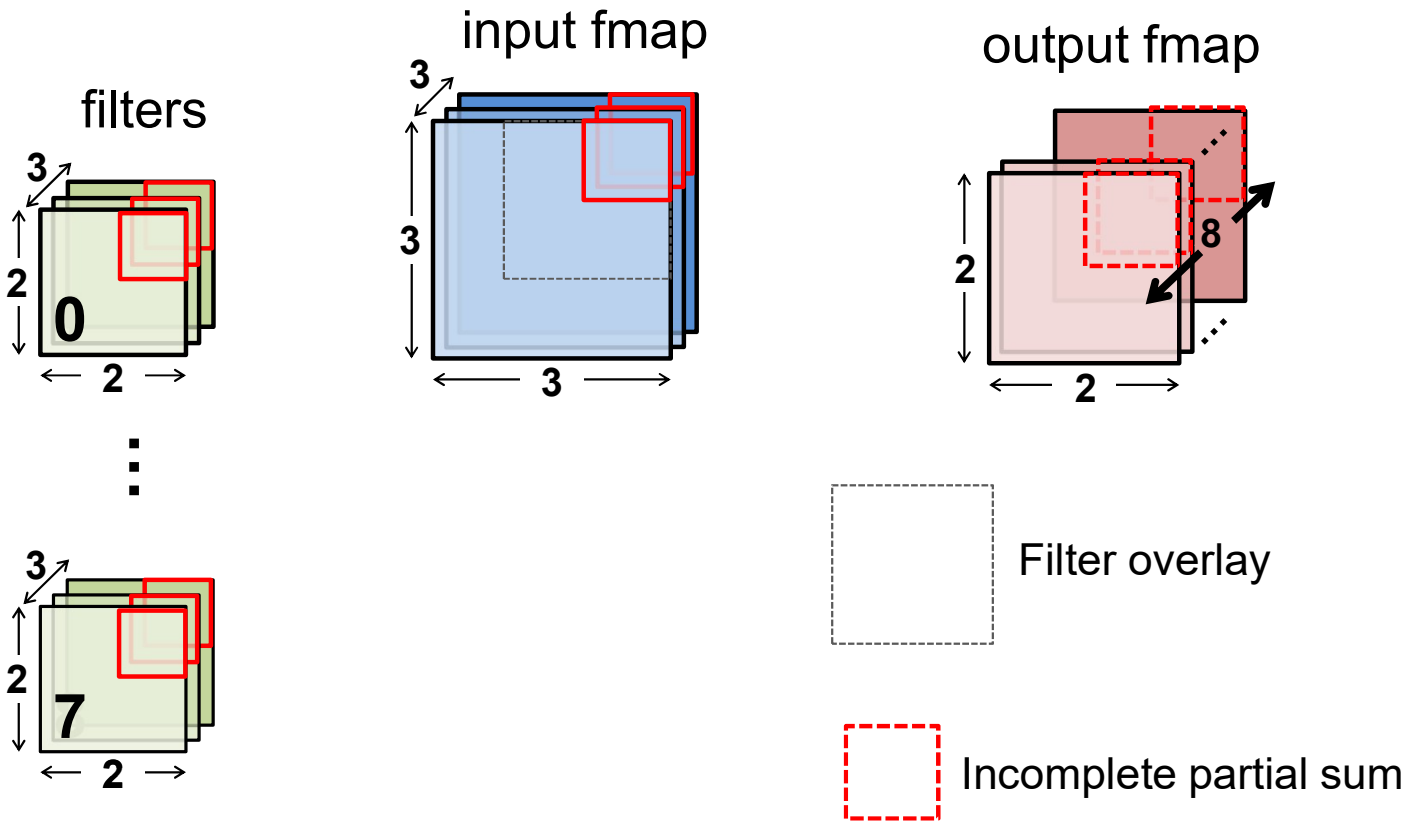
# CONV Layer OS Dataflow

Start processing next output feature activations



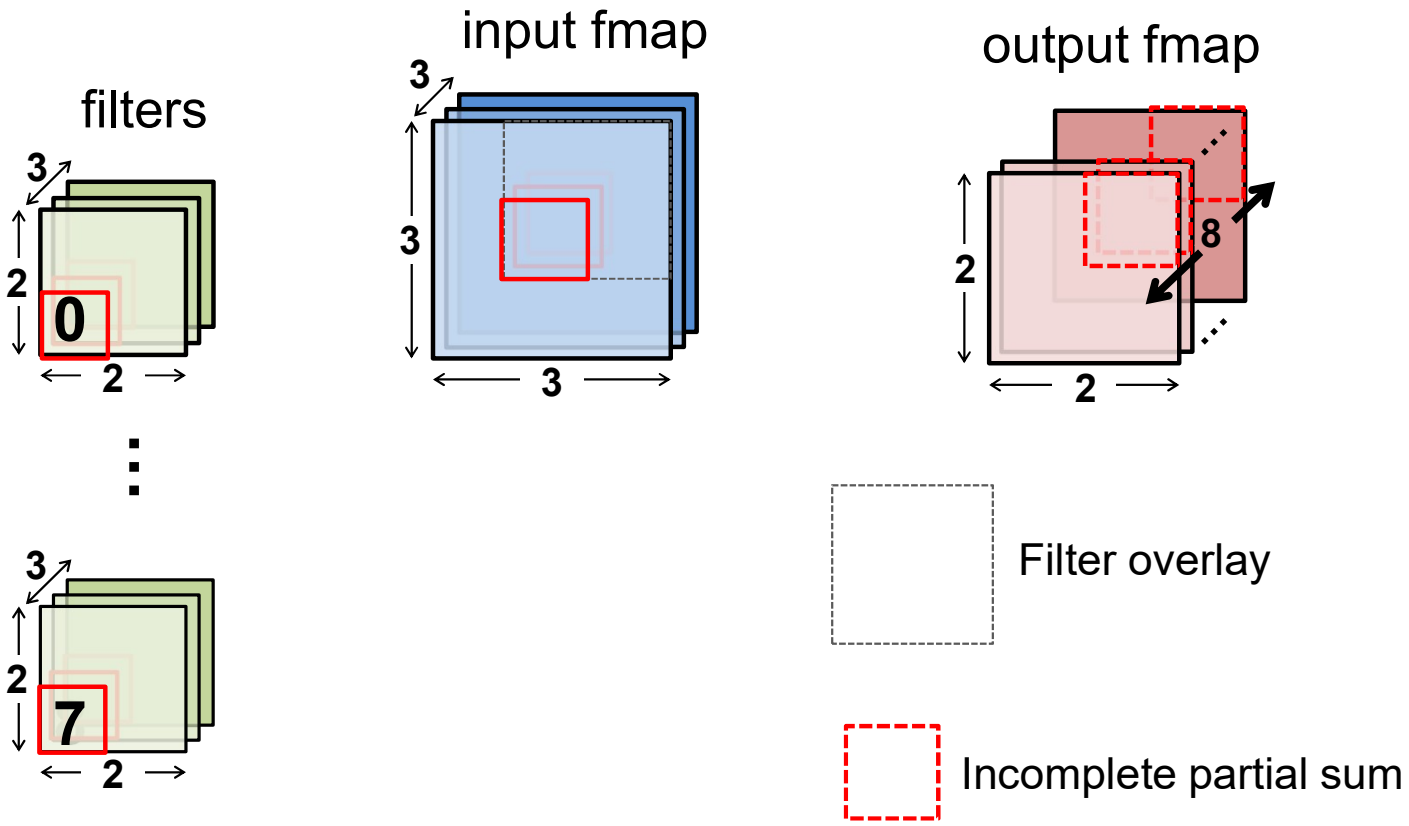
# CONV Layer OS Dataflow

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



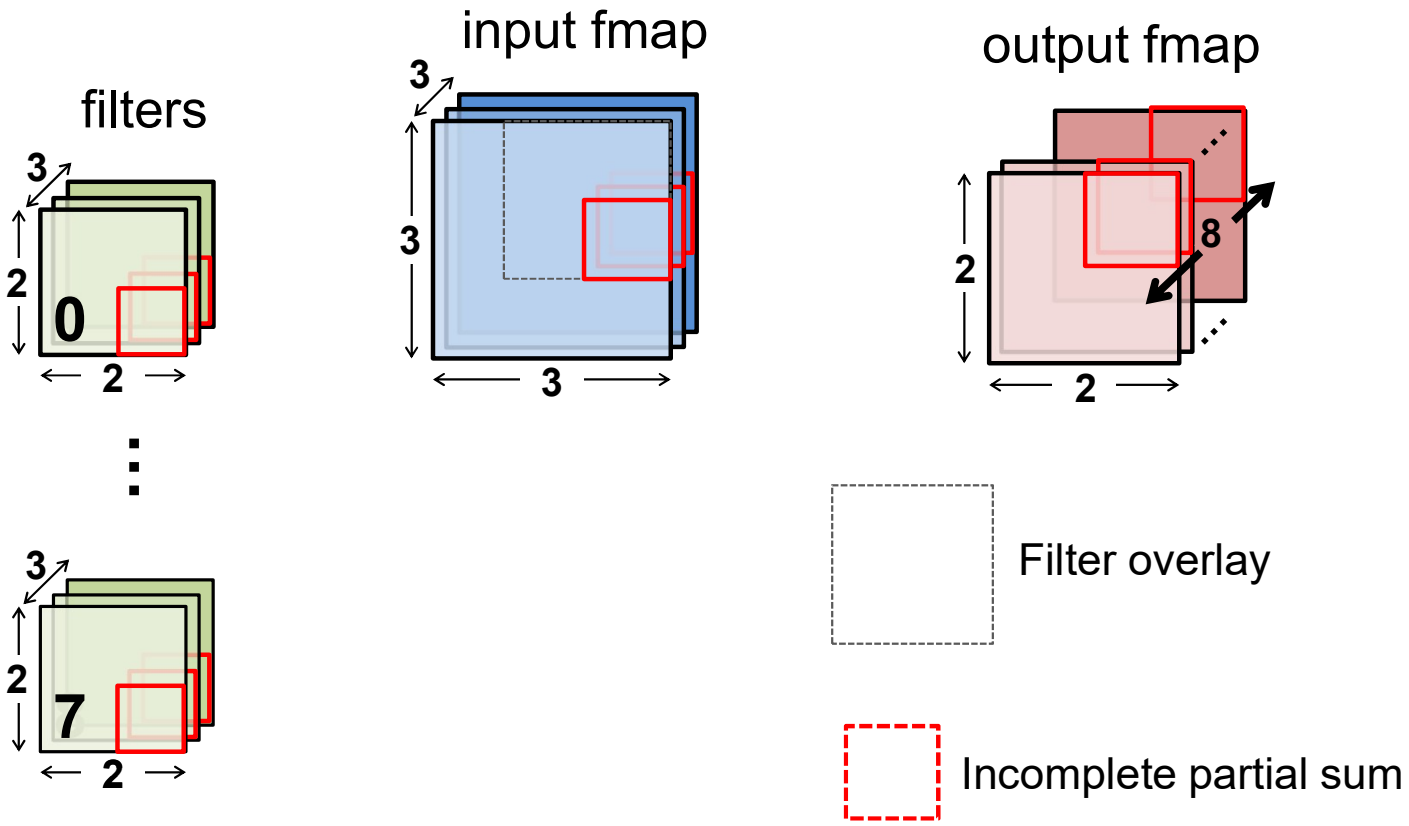
# CONV Layer OS Dataflow

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



# CONV Layer OS Dataflow

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



# Weight Stationary

# 1-D Convolution – Output Stationary


```
int i[W];     # Input activations
int f[S];     # Filter weights
int o[Q];     # Output activations

for q in [0, Q):
    for s in [0, S):
        o[q] += i[q+s]*f[s]
```

No constraints on loop permutations!

† Assuming: 'valid' style convolution

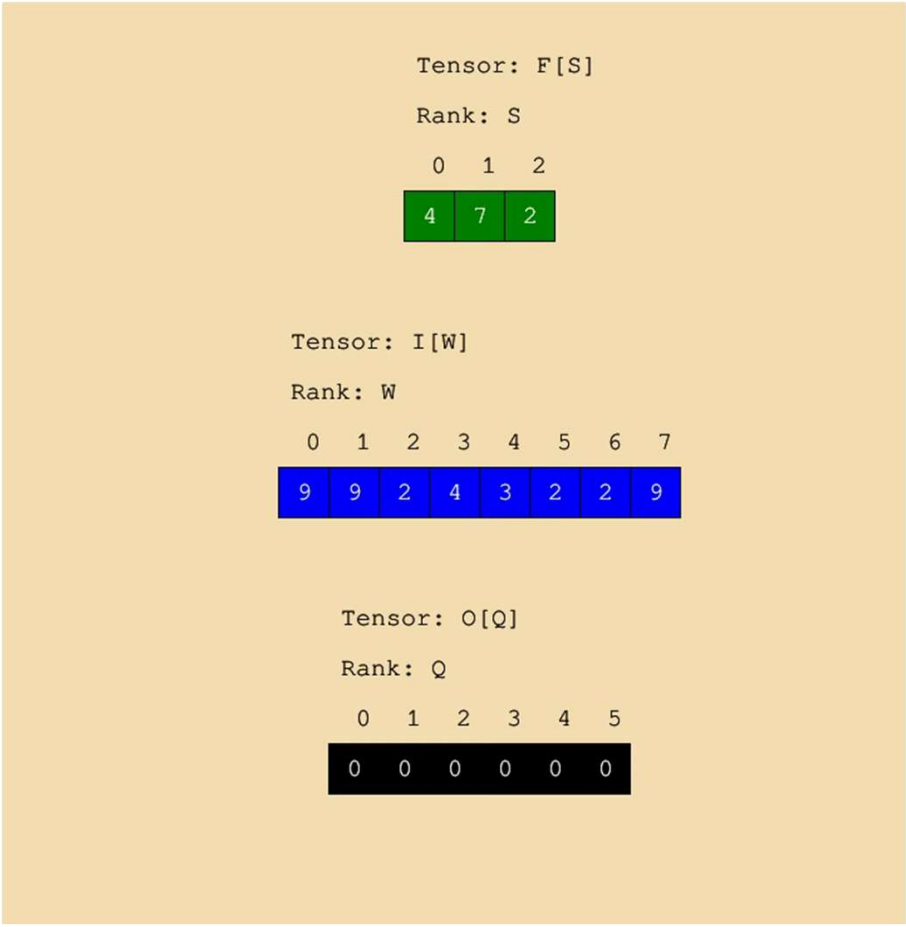
# 1-D Convolution


```
int i[W];     # Input activations
int f[S];     # Filter weights
int o[Q];     # Output activations

for s in [0, S):
      for q in [0, Q):
         o[q] += i[q+s]*f[s]
```

What dataflow is this? Weight stationary

# Weight Stationary - Animation

```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

for s in [0, S):
       for q in [0, Q):
          o[q] += i[q+s]*f[s]
```




Full animation at: https://csg.csail.mit.edu/6.5930/lectures/slidev/l05/#3

# Weight Stationary - Spacetime

Tensor: F[S, T]

Rank: T

| Rank: S | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 1 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |

Tensor: I[W, T]

Rank: T

| Rank: W | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 1 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 5 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 6 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 7 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |

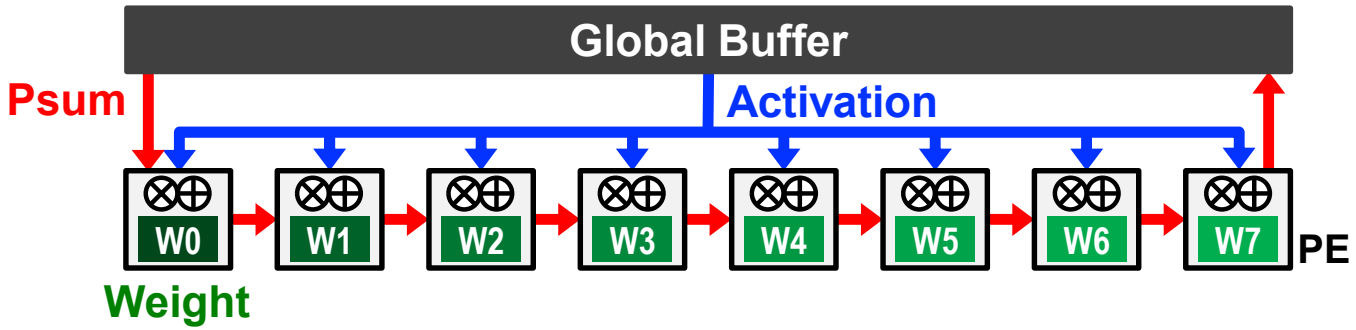
Tensor: O[Q, T]

Rank: T

| Rank: Q | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 36 | 36 | 36 | 36 | 36 | 36 | 99 | 99 | 99 | 99 | 99 | 99 | 103 | 103 | 103 | 103 | 103 | 103 |
| 1 | 0 | 36 | 36 | 36 | 36 | 36 | 36 | 50 | 50 | 50 | 50 | 50 | 50 | 58 | 58 | 58 | 58 | 58 |
| 2 | 0 | 0 | 8 | 8 | 8 | 8 | 8 | 8 | 36 | 36 | 36 | 36 | 36 | 36 | 42 | 42 | 42 | 42 |
| 3 | 0 | 0 | 0 | 16 | 16 | 16 | 16 | 16 | 16 | 37 | 37 | 37 | 37 | 37 | 37 | 41 | 41 | 41 |
| 4 | 0 | 0 | 0 | 0 | 12 | 12 | 12 | 12 | 12 | 12 | 26 | 26 | 26 | 26 | 26 | 26 | 30 | 30 |
| 5 | 0 | 0 | 0 | 0 | 0 | 8 | 8 | 8 | 8 | 8 | 8 | 22 | 22 | 22 | 22 | 22 | 22 | 40 |

Note: ‘Rank T’ here just meant time in cycles, not an actual rank in tensor

# 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.

***Weights are in the outer loop***

# 1-D Convolution Einsum + WS

Serial WS design

Einsum: $O_q = I_{q+s} \times F_s$

Traversal order (fastest to slowest): Q, S

---

WS design for parallel weights

Einsum: $O_q = I_{q+s} \times F_s$

Parallel Ranks: S

Traversal order (fastest to slowest): Q

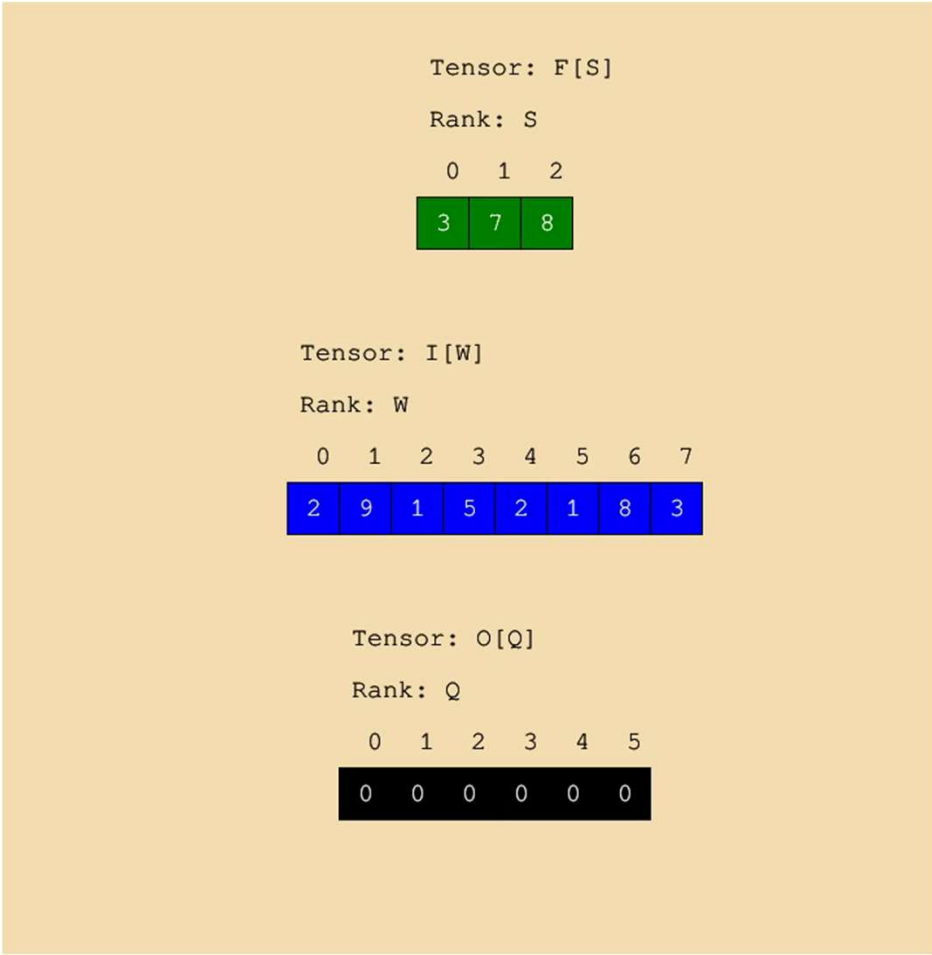
# Parallel Weight Stationary - Animation

Note: Different colored highlights are different parallel PEs

```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

parallel_for s in [0, S):
   for q in [0, Q):
         o[q] += i[q+s]*f[s]
```

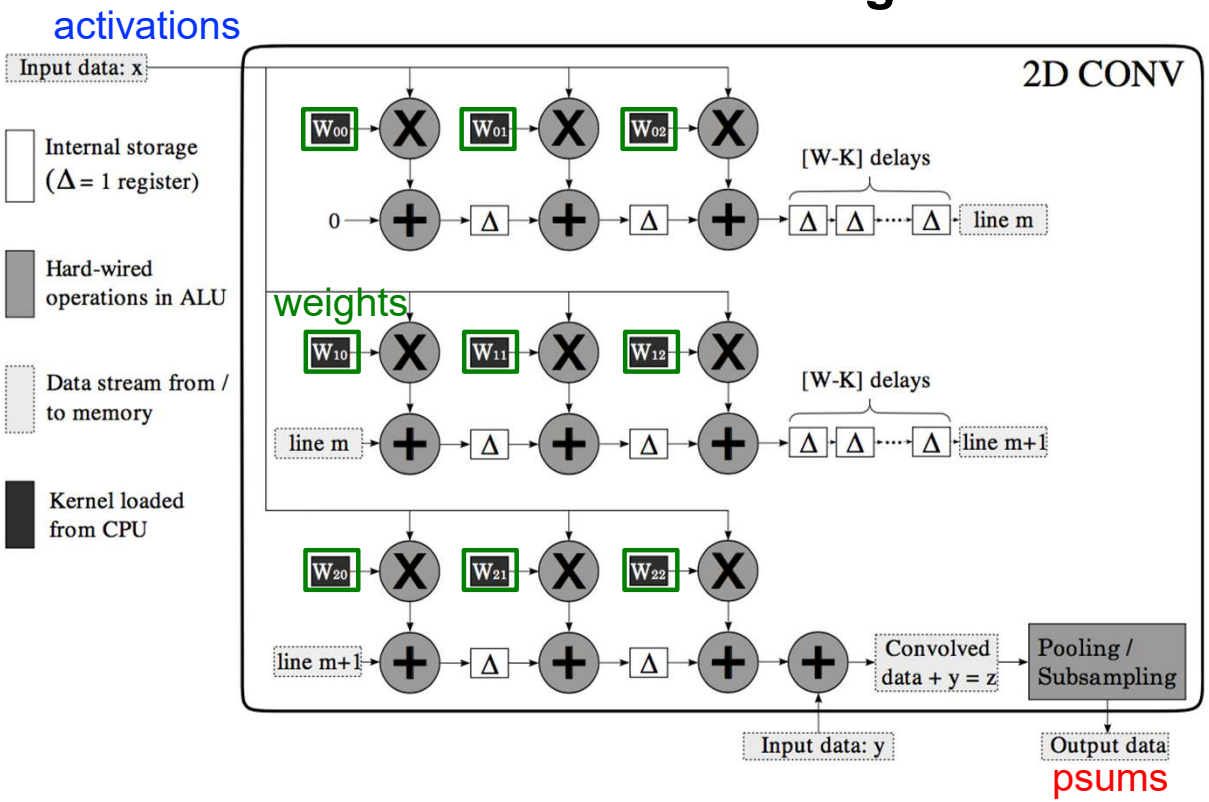
- All weights used
- Activations are multicast to all PEs
- Multiple outputs are worked on and then they move to the next PE

Full animation at https://csg.csail.mit.edu/6.5930/lectures/slidev/l05/#4



# WS Example: nn-X (NeuFlow)

## A 3×3 2D Convolution Engine

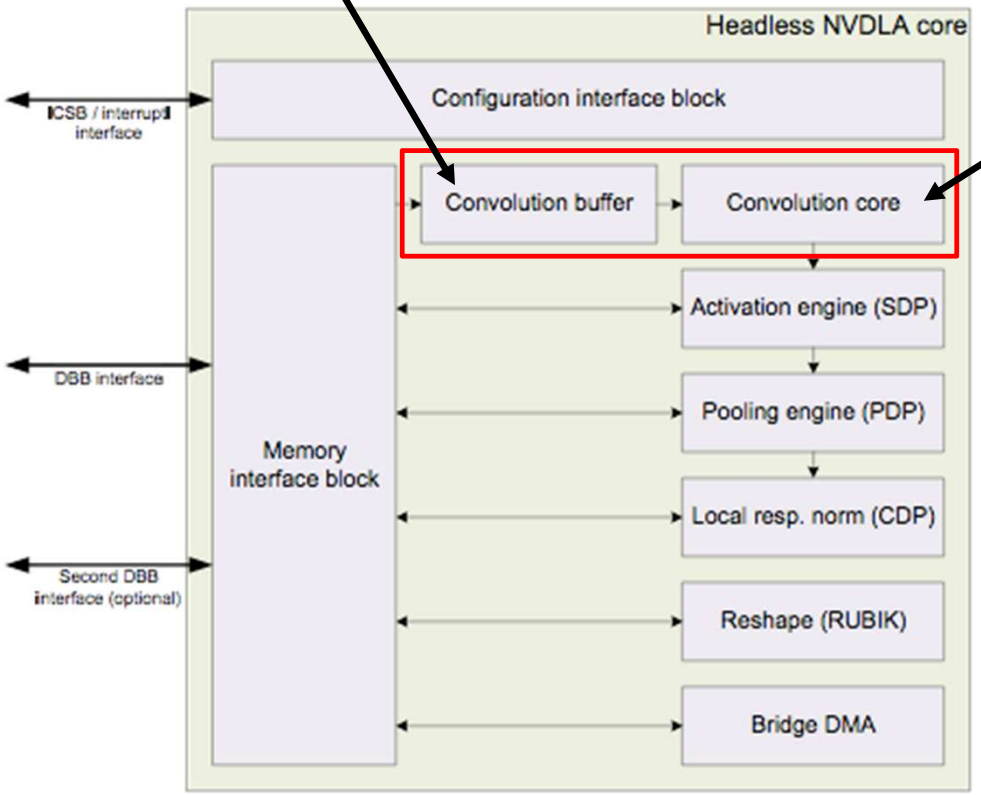



[Farabet et al., ICCV 2009]

# WS Example: NVDLA (simplified)

Global Buffer

Released Sept 29, 2017




PE Array

*M x C MACs:*
The number of input and output channels must match array, otherwise MACs idle
Range:

- C (16 to 128)
- M (4 to 16)

*Note: In this class we use M for number of kernels (filters), while NVDLA uses K

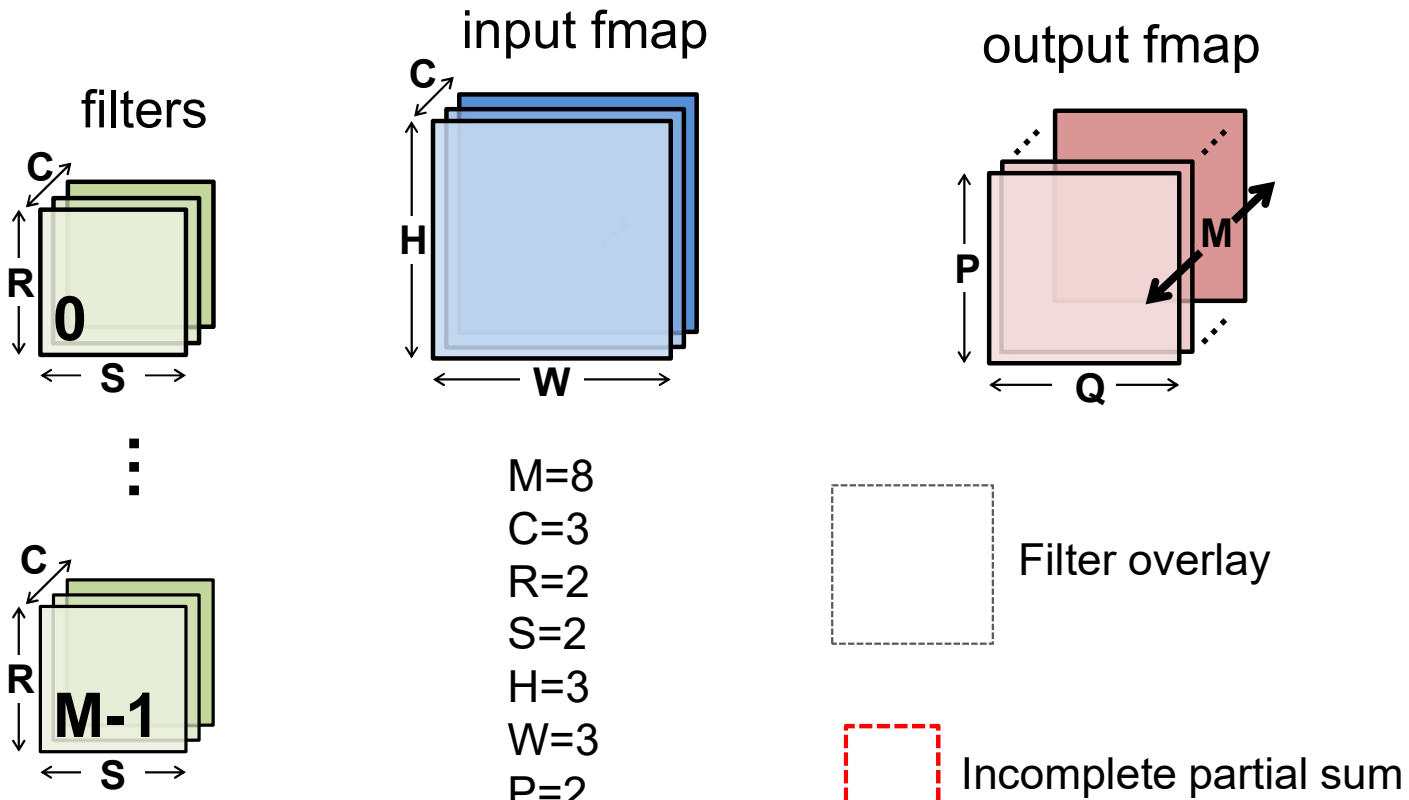
Image Source: Nvidia

http://nvdla.org

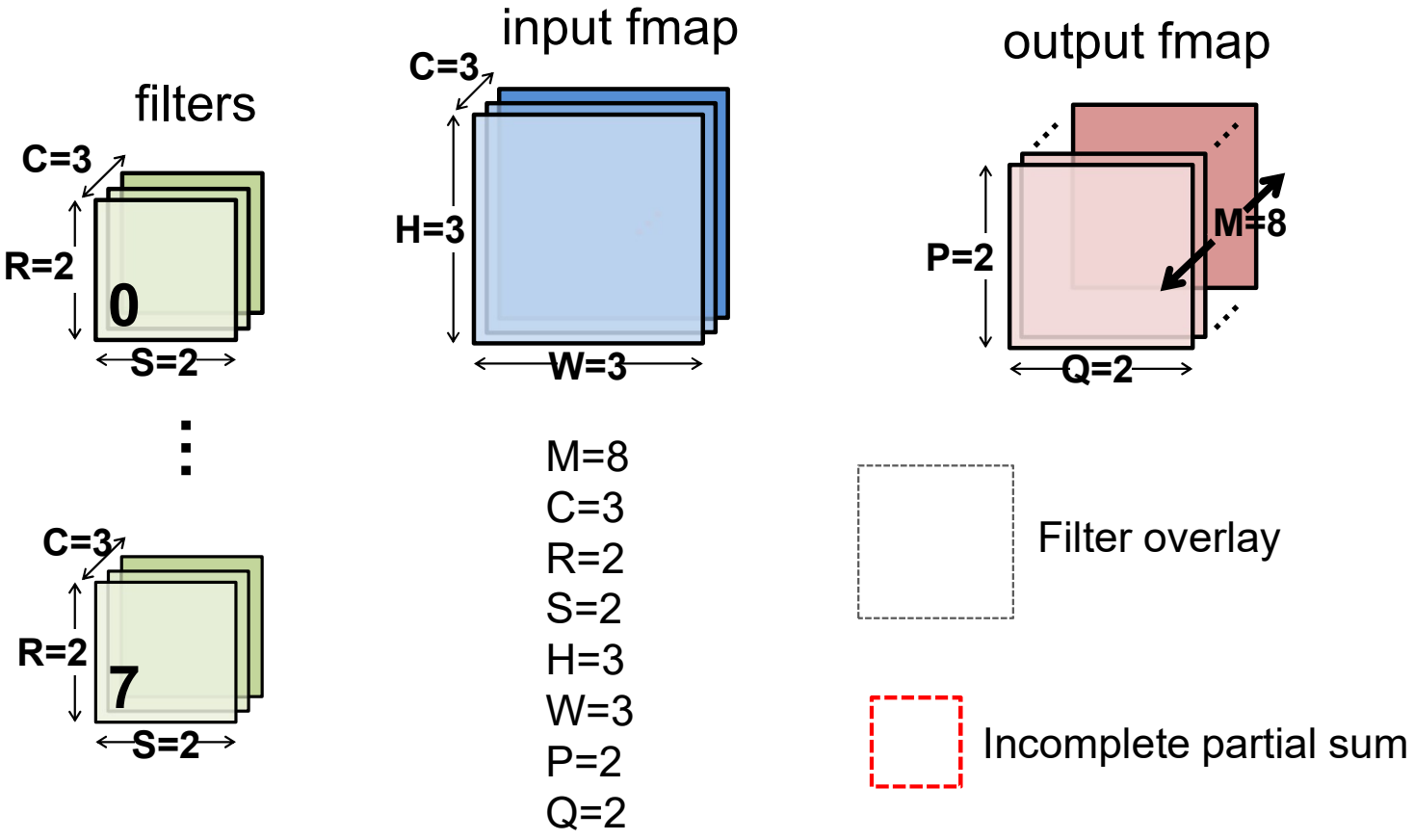
# WS Example: NVDLA

- **Convolution Buffer**
  - Stores both weights and activations
  - Ratio of weights and activation data varies across layers
  - Four access ports: R+W activations & R+W weights
    - Convolution read bandwidth determine size of read port (C*datasize)
  - Optional compression support
  - Prefer to either store all weights or all activations

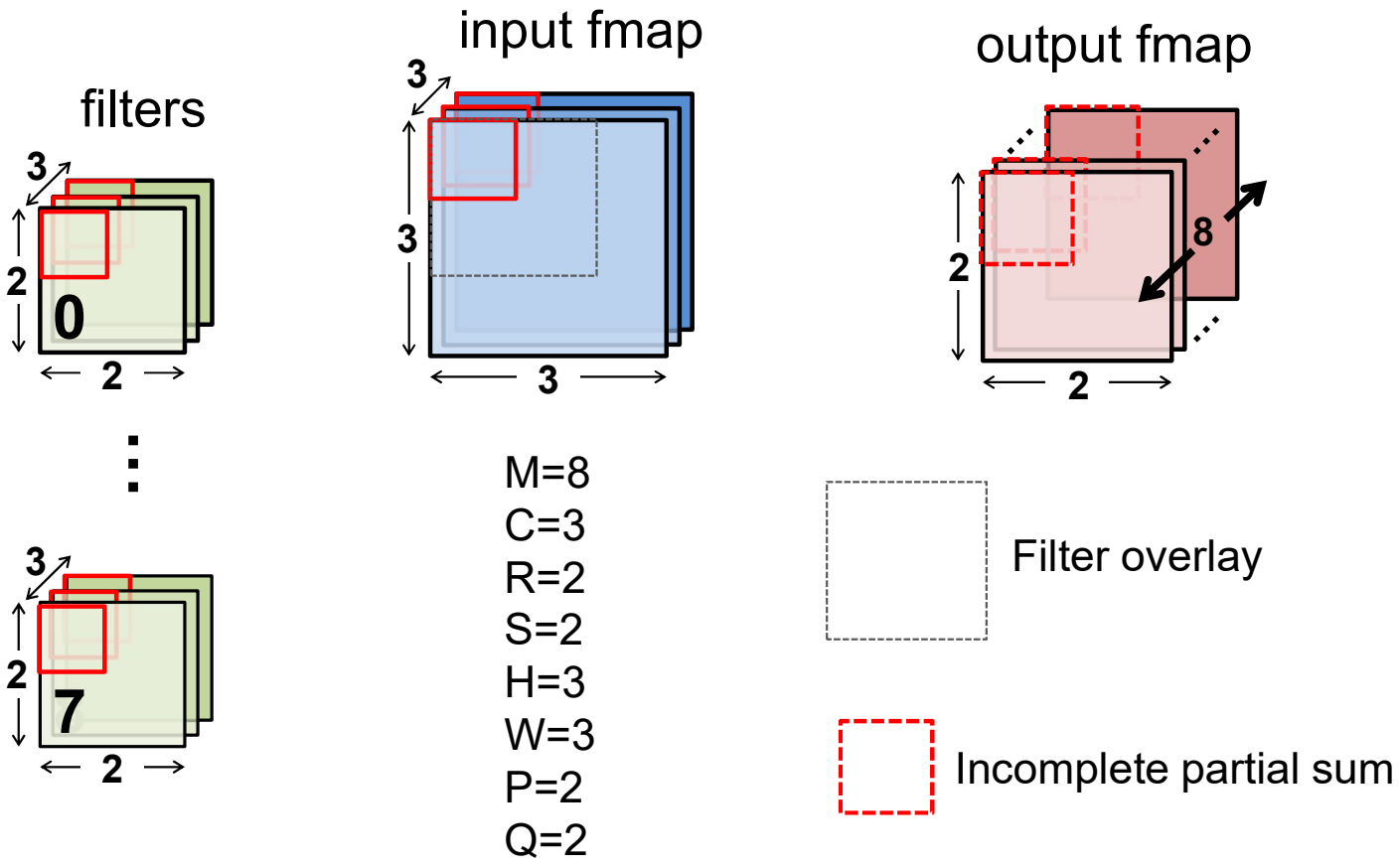
# WS Example: NVDLA (simplified)



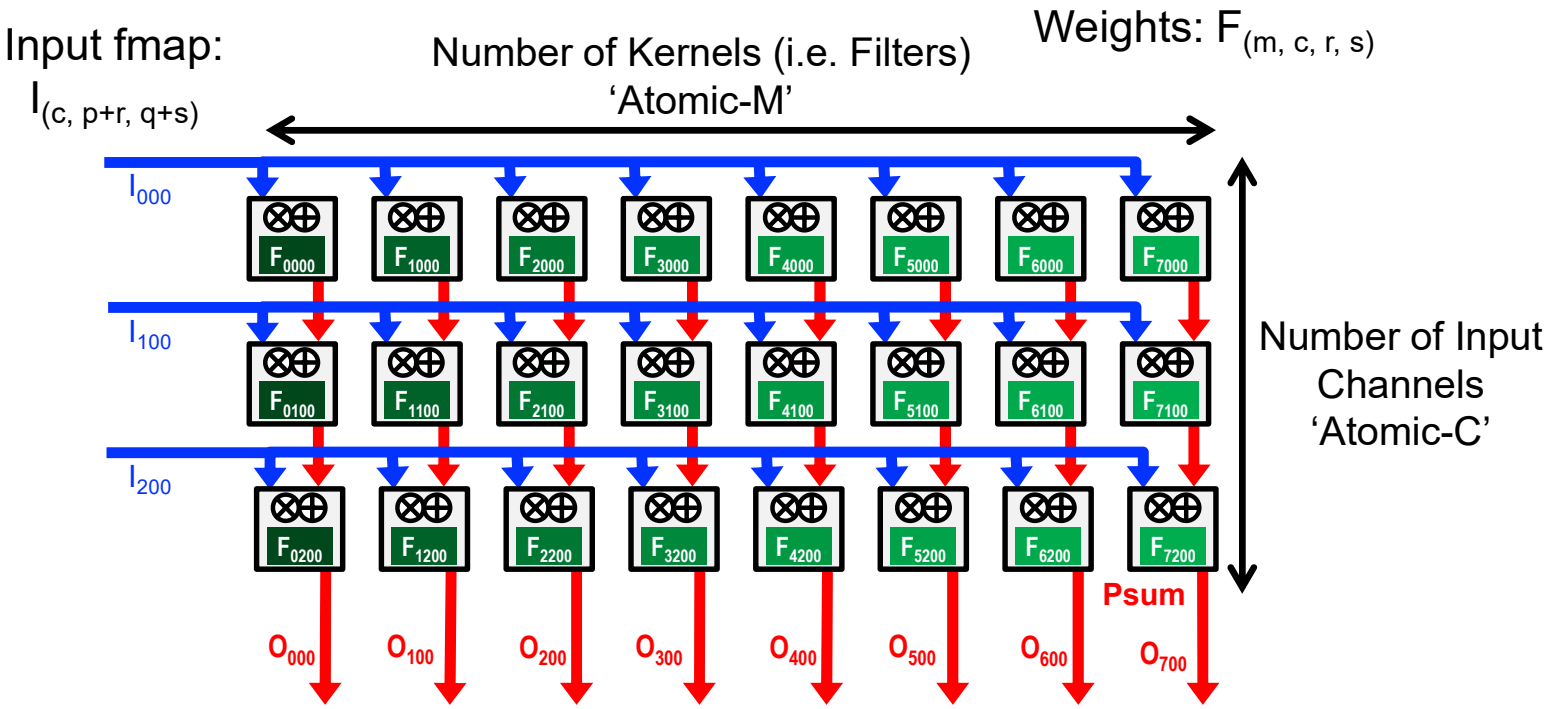
# WS Example: NVDLA (simplified)



# WS Example: NVDLA (simplified)

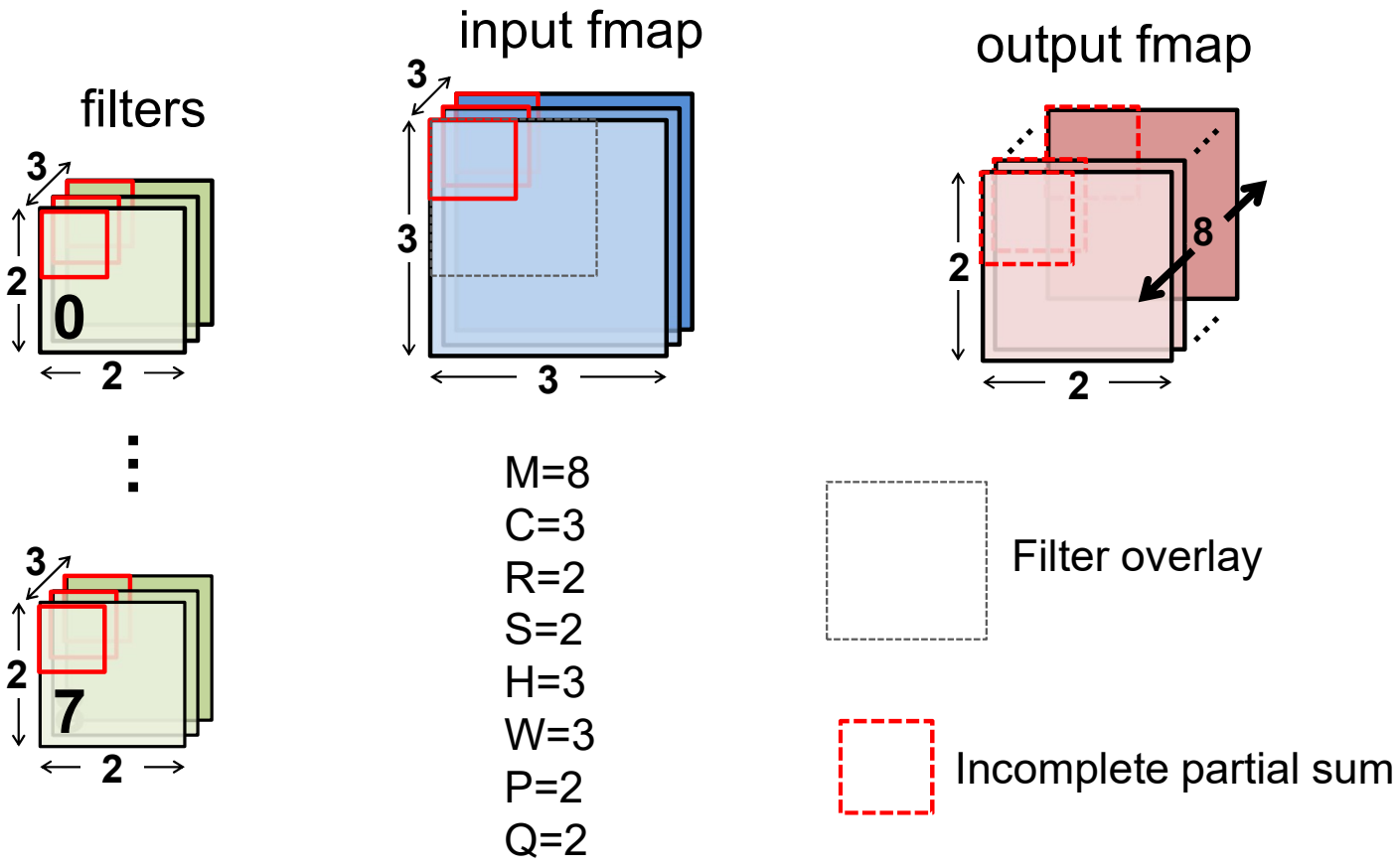


# WS Example: NVDLA (simplified)



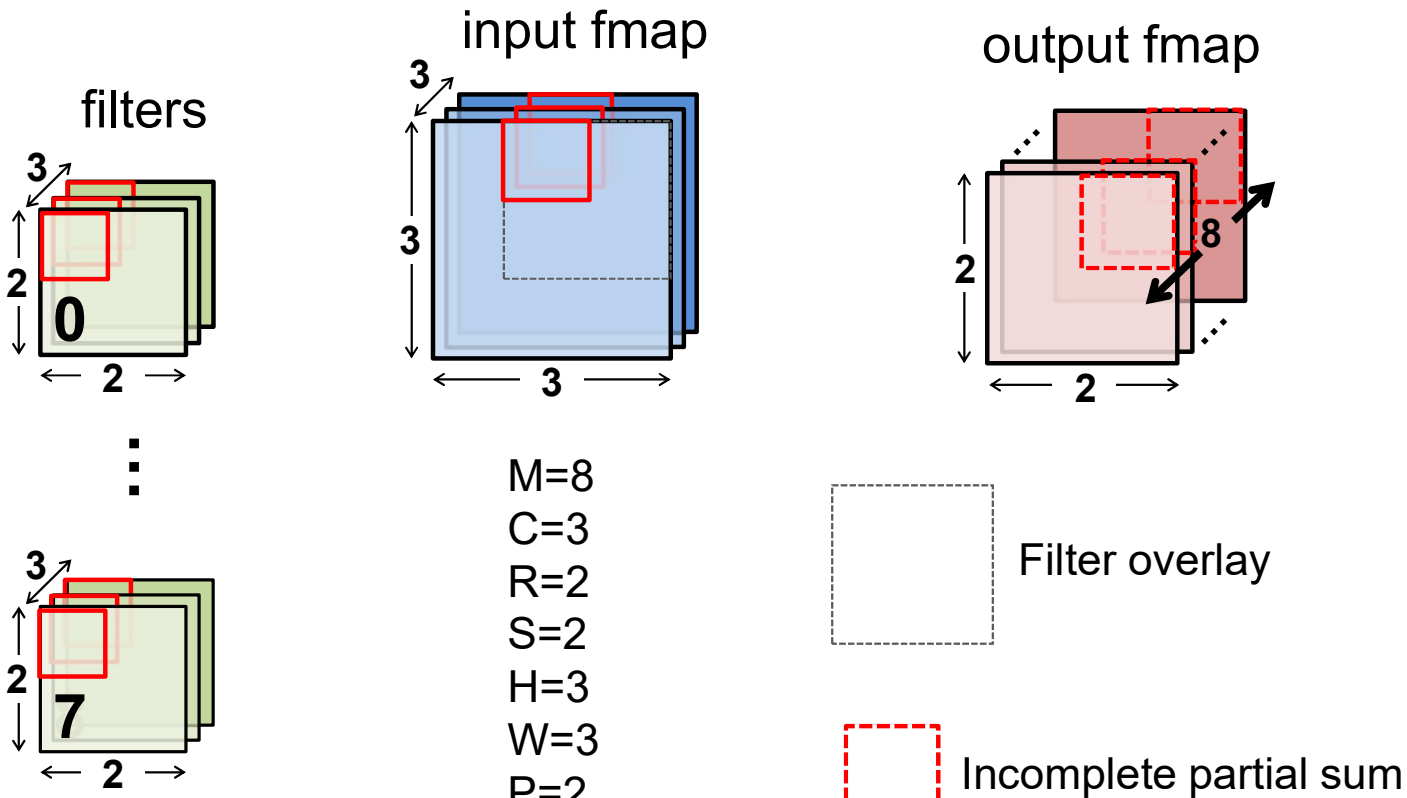

Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)



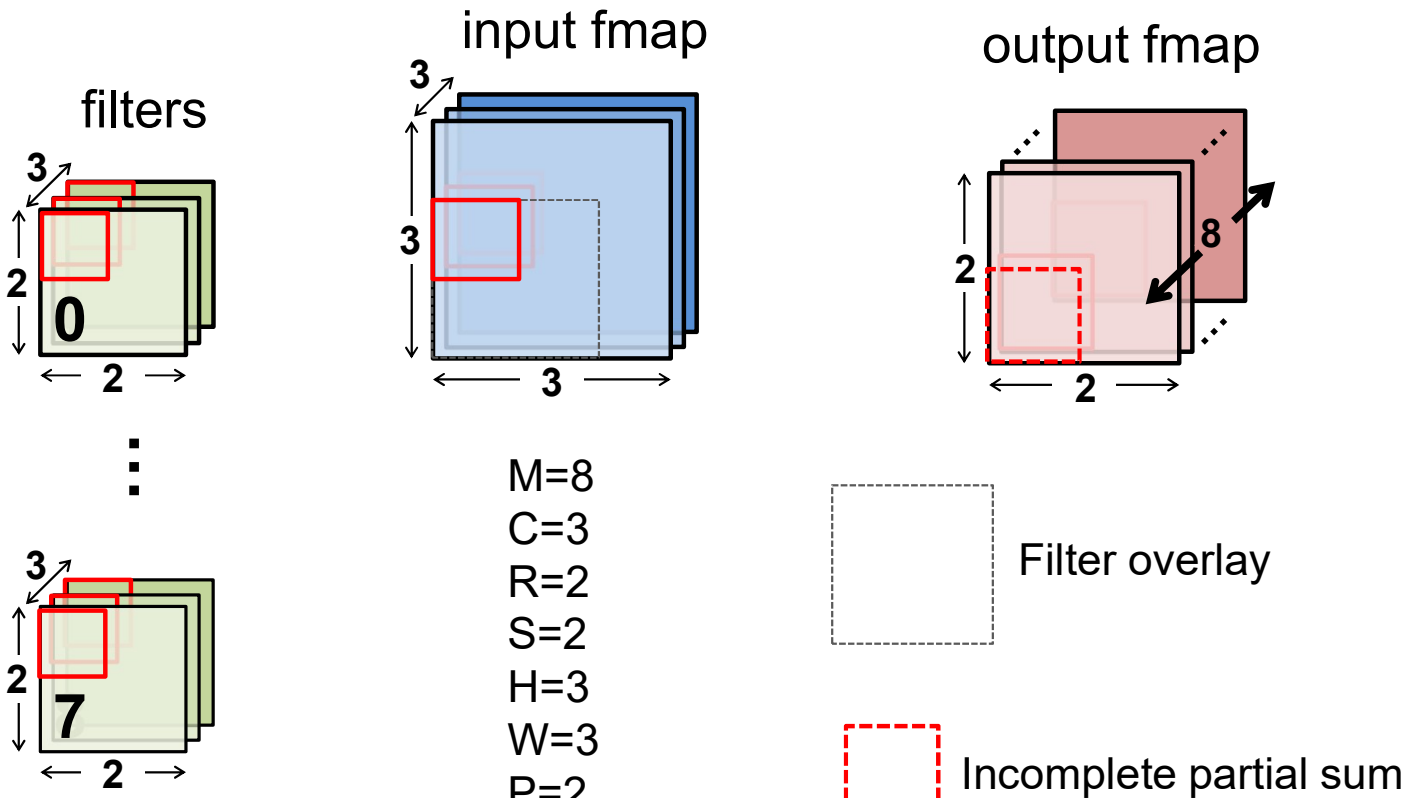
# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weight)



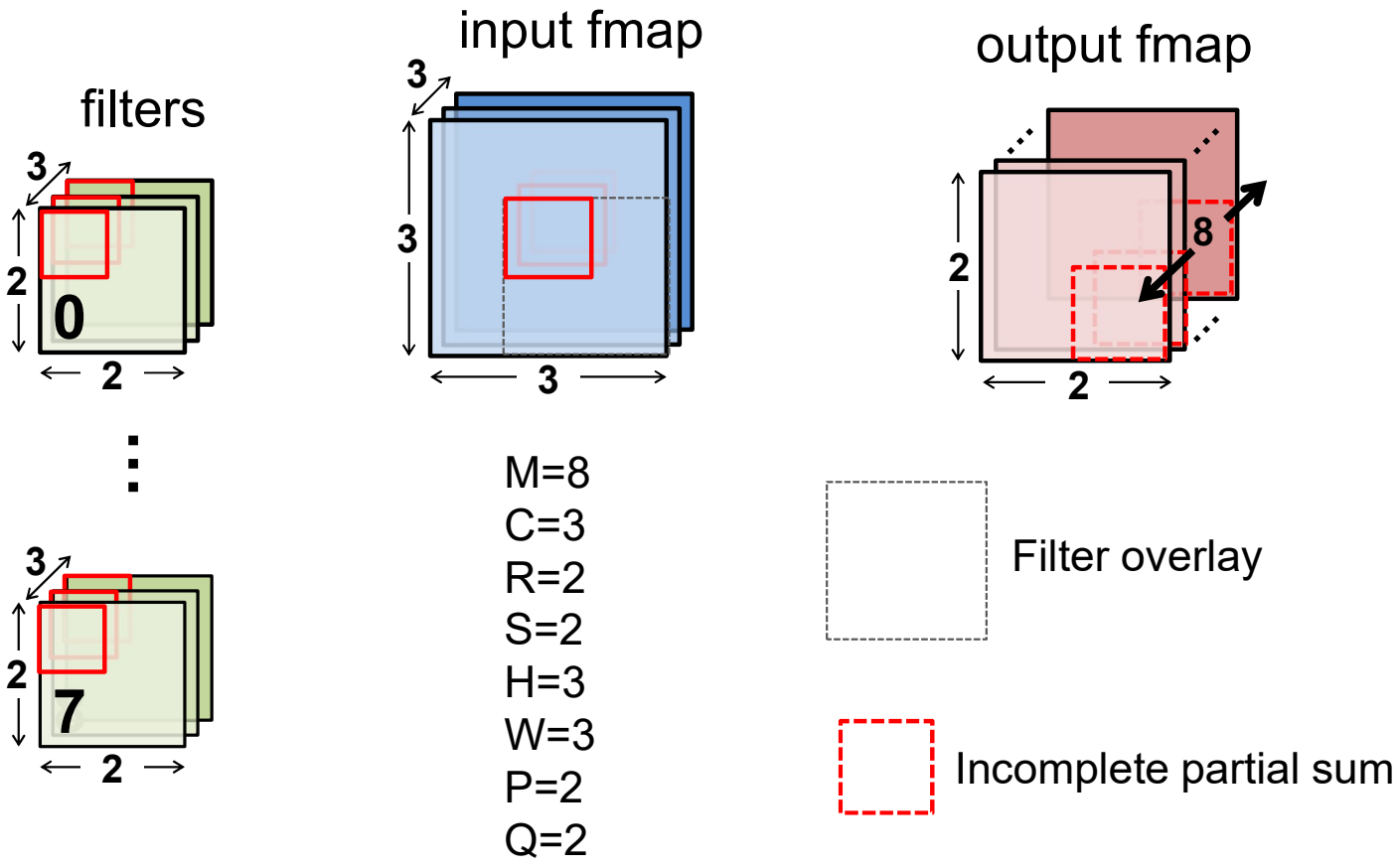
# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weight)

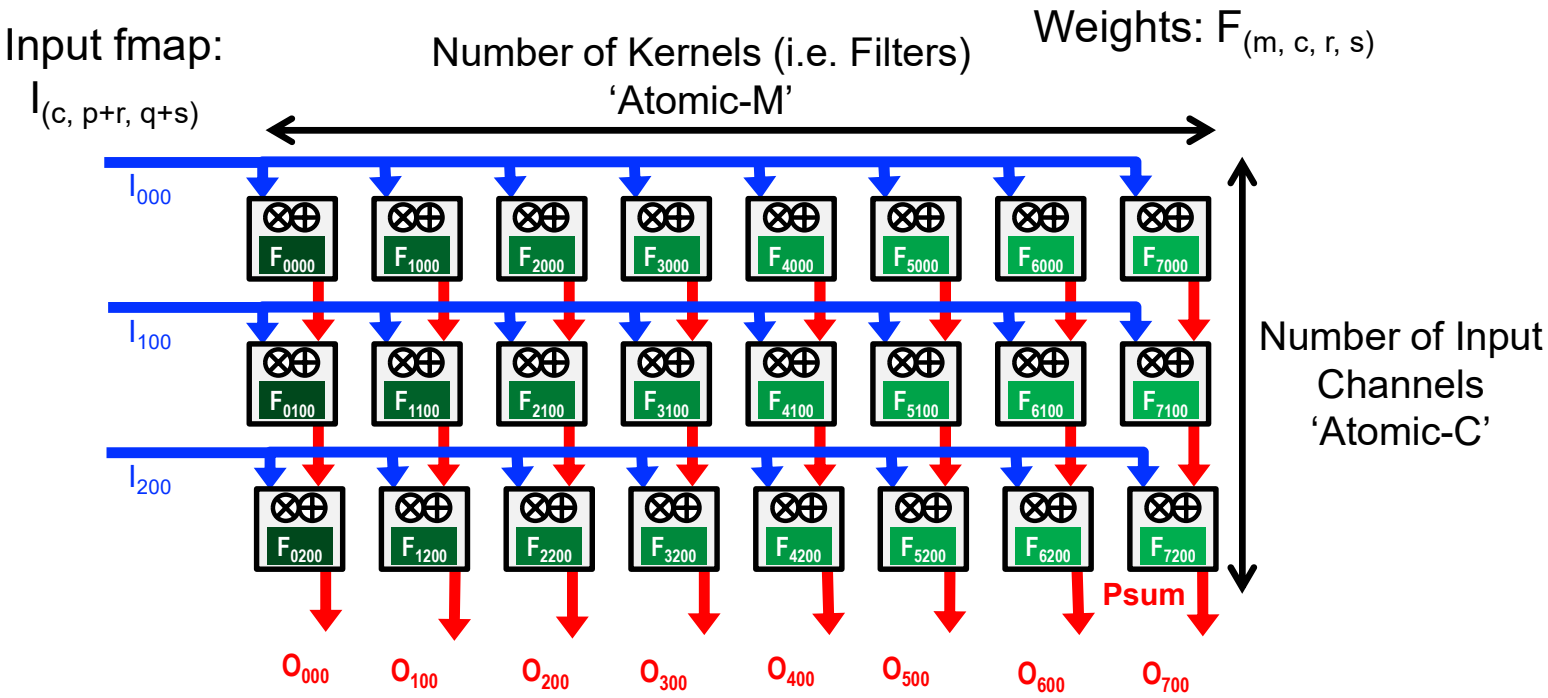


# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weights)

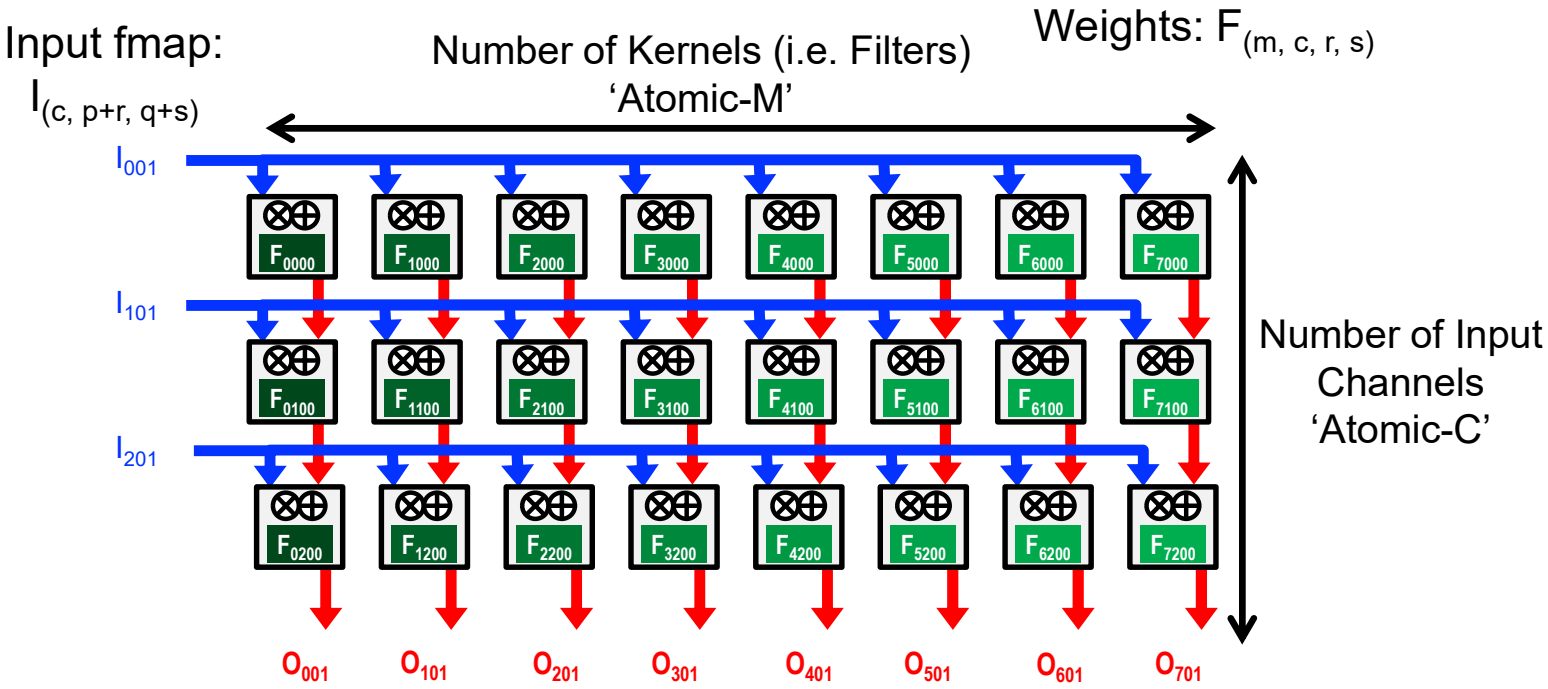


# WS Example: NVDLA (simplified)



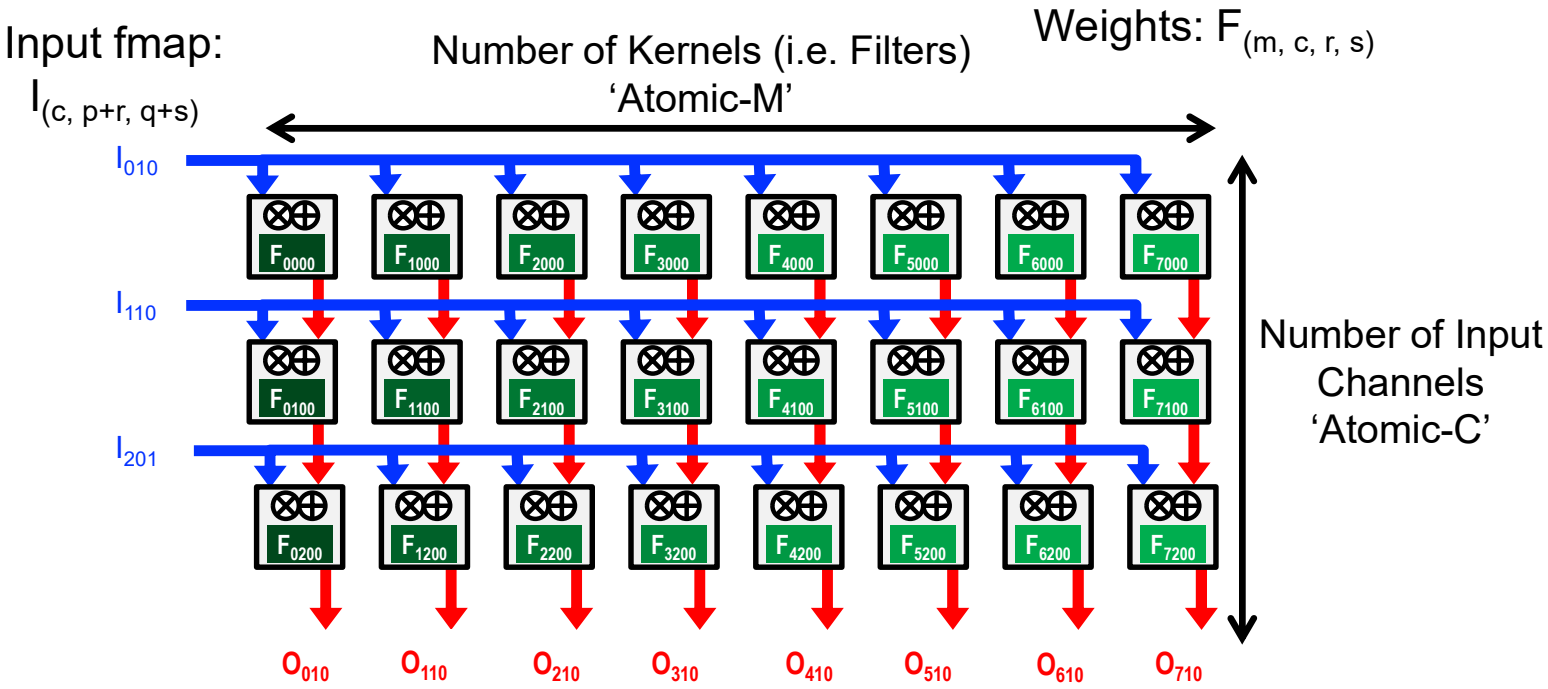

Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)



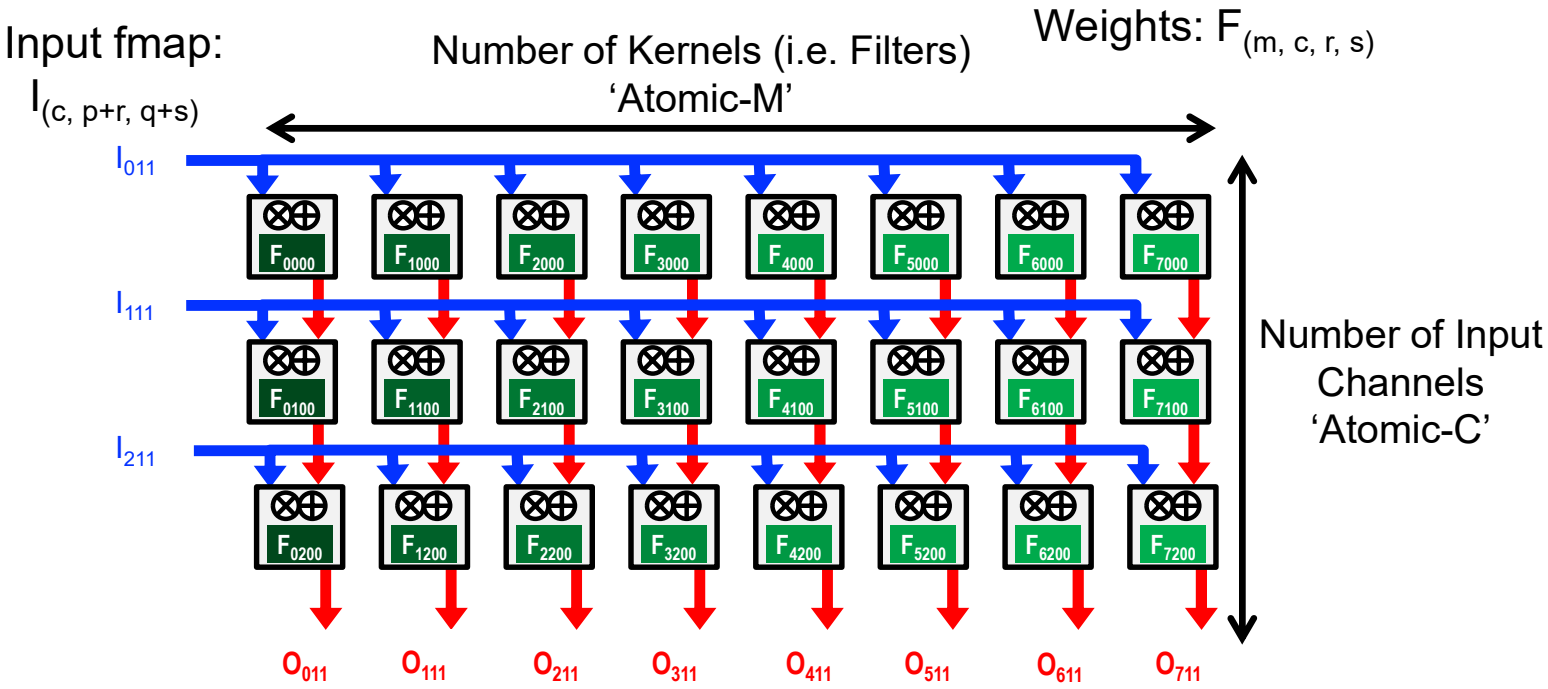

Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)




Output feature map: $O_{(m, p, q)}$

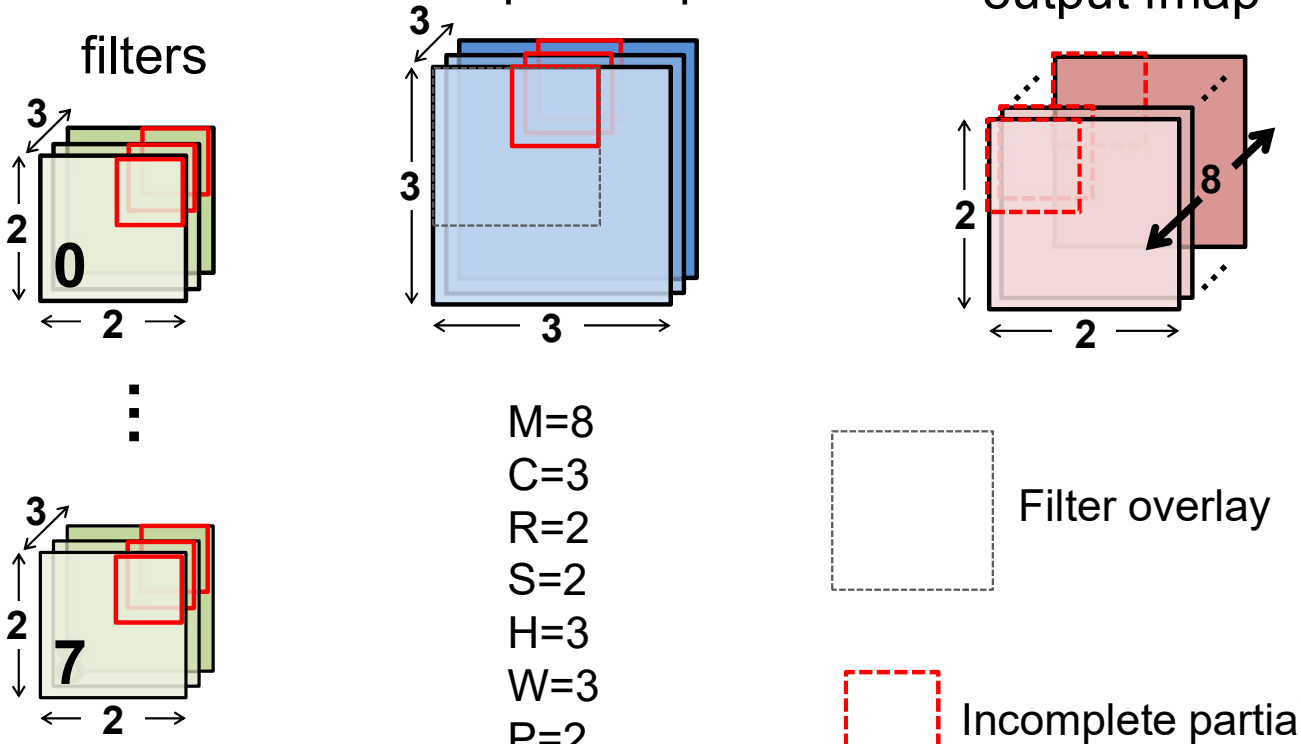
# WS Example: NVDLA (simplified)




Output feature map: $O_{(m, p, q)}$

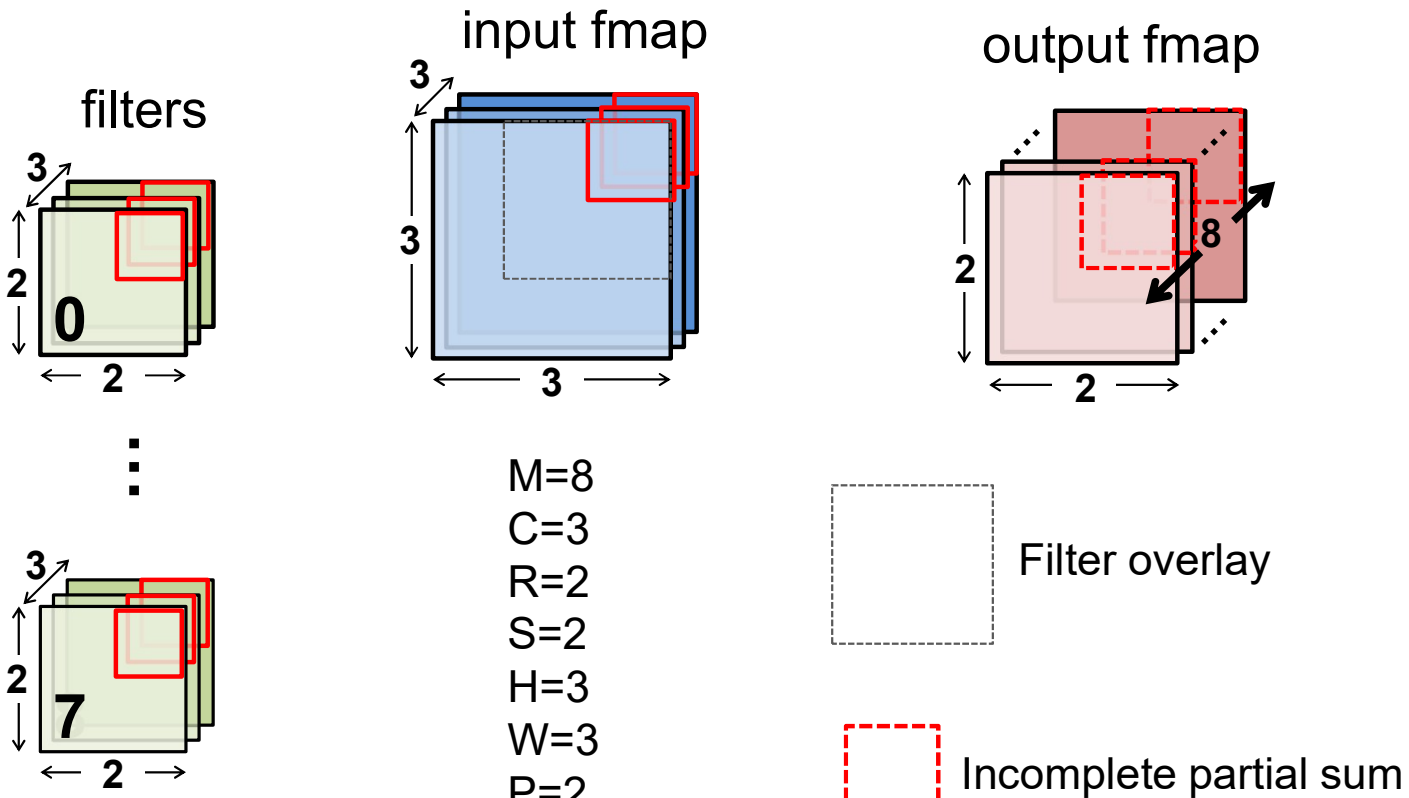
# WS Example: NVDLA (simplified)

Load new weights



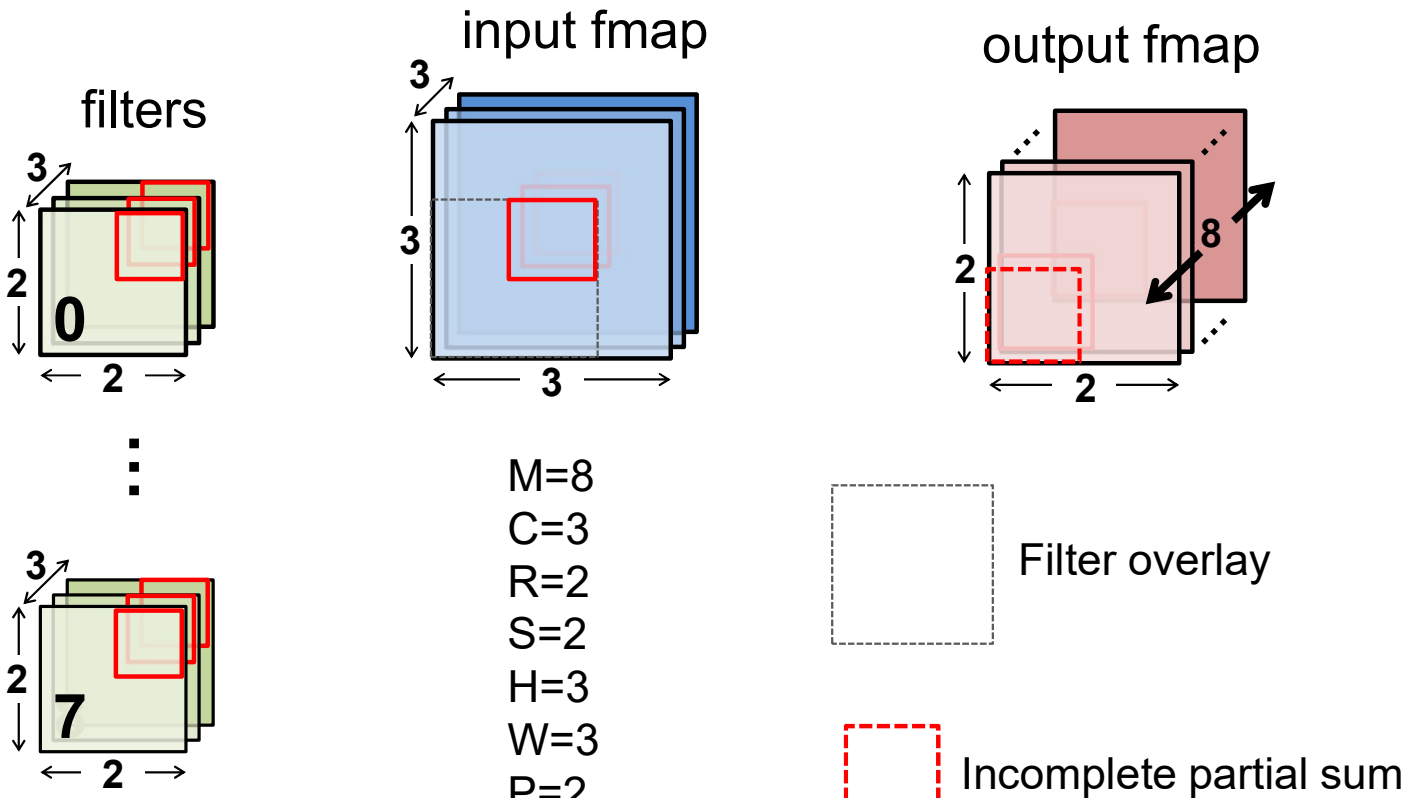
# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weights)



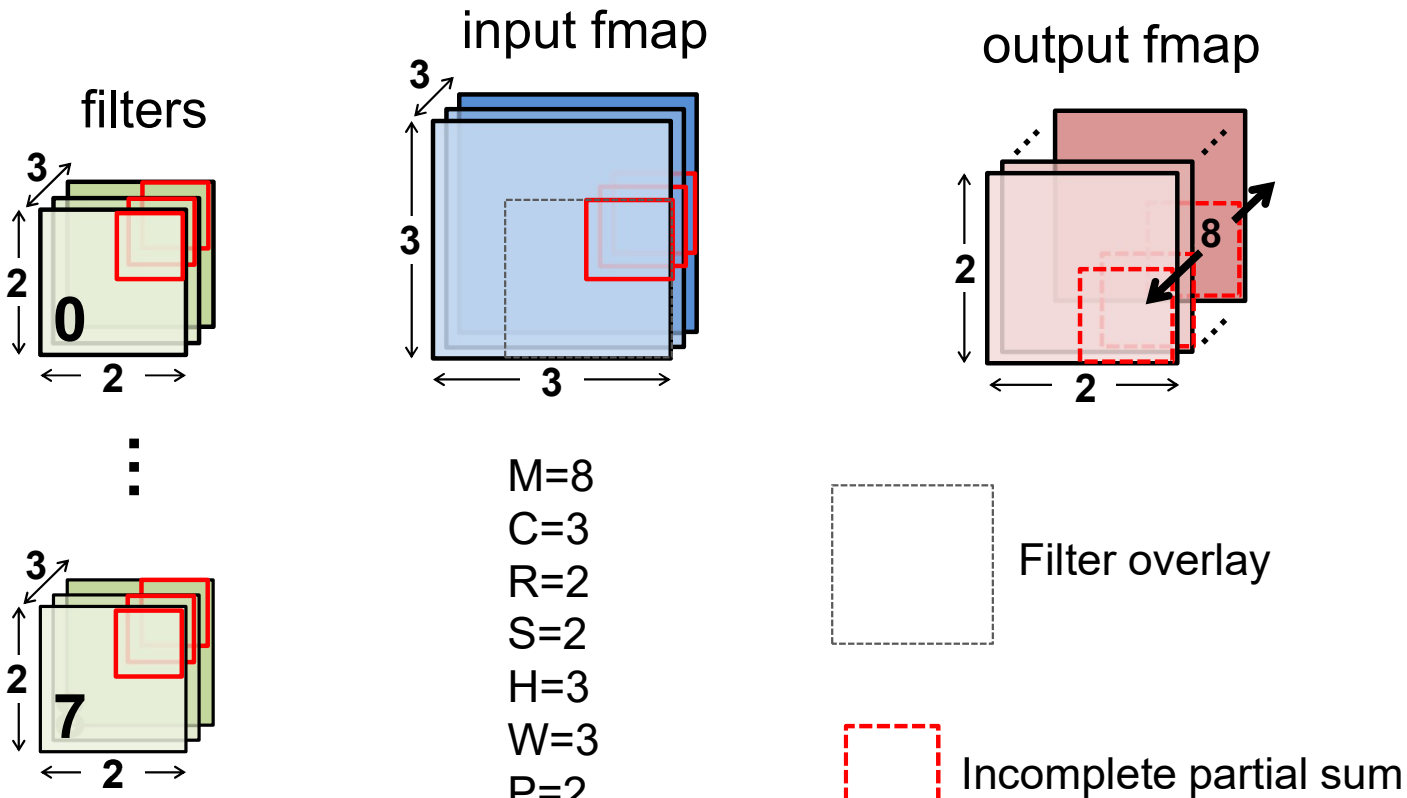
# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weights)

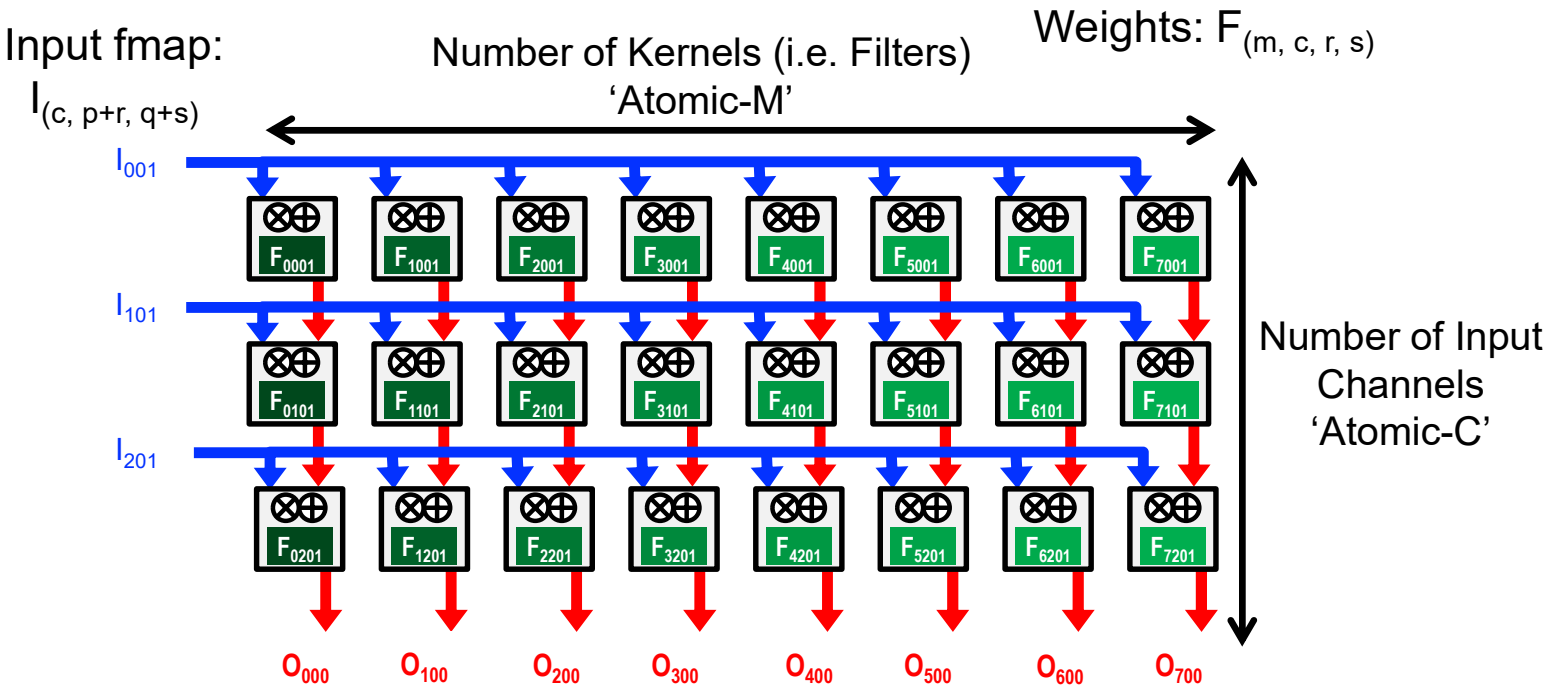


# WS Example: NVDLA (simplified)

Cycle through input and output fmap (hold weights)

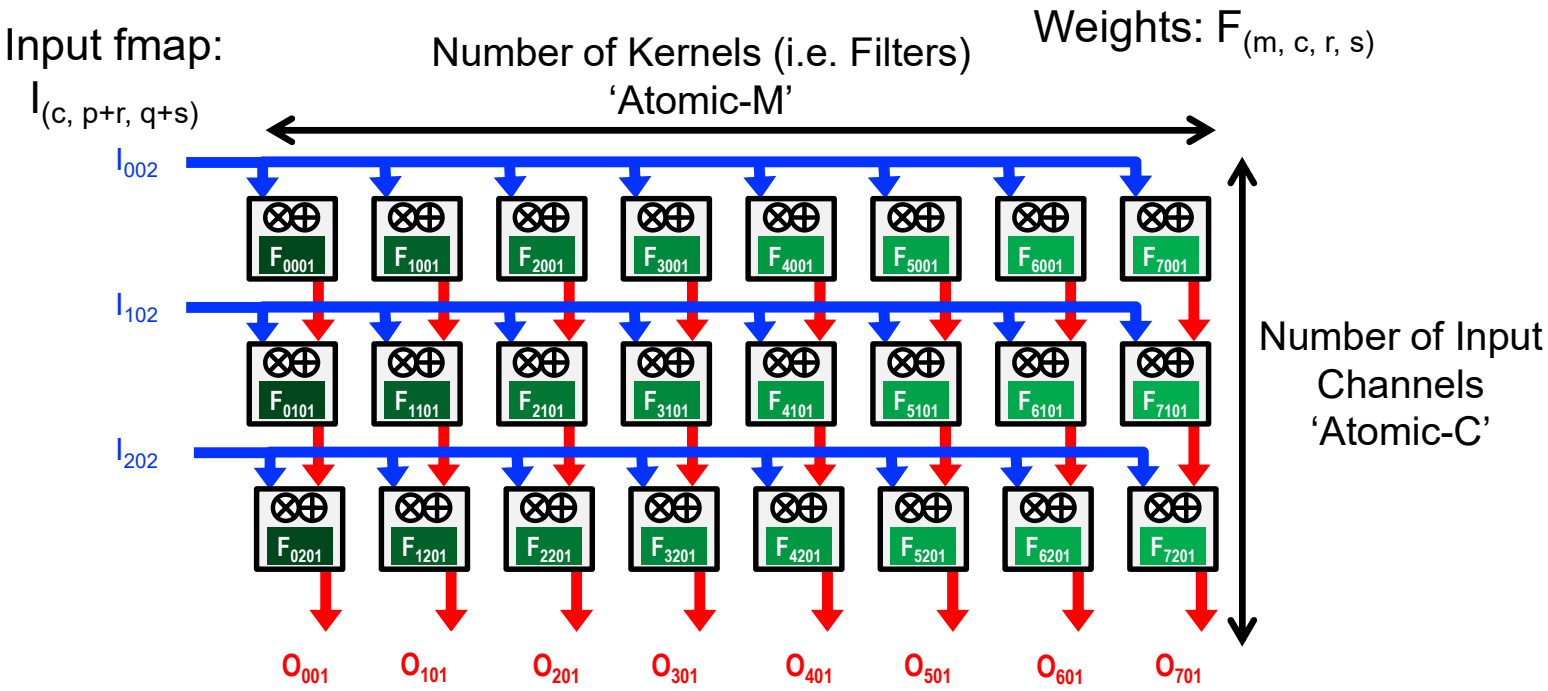


# WS Example: NVDLA (simplified)



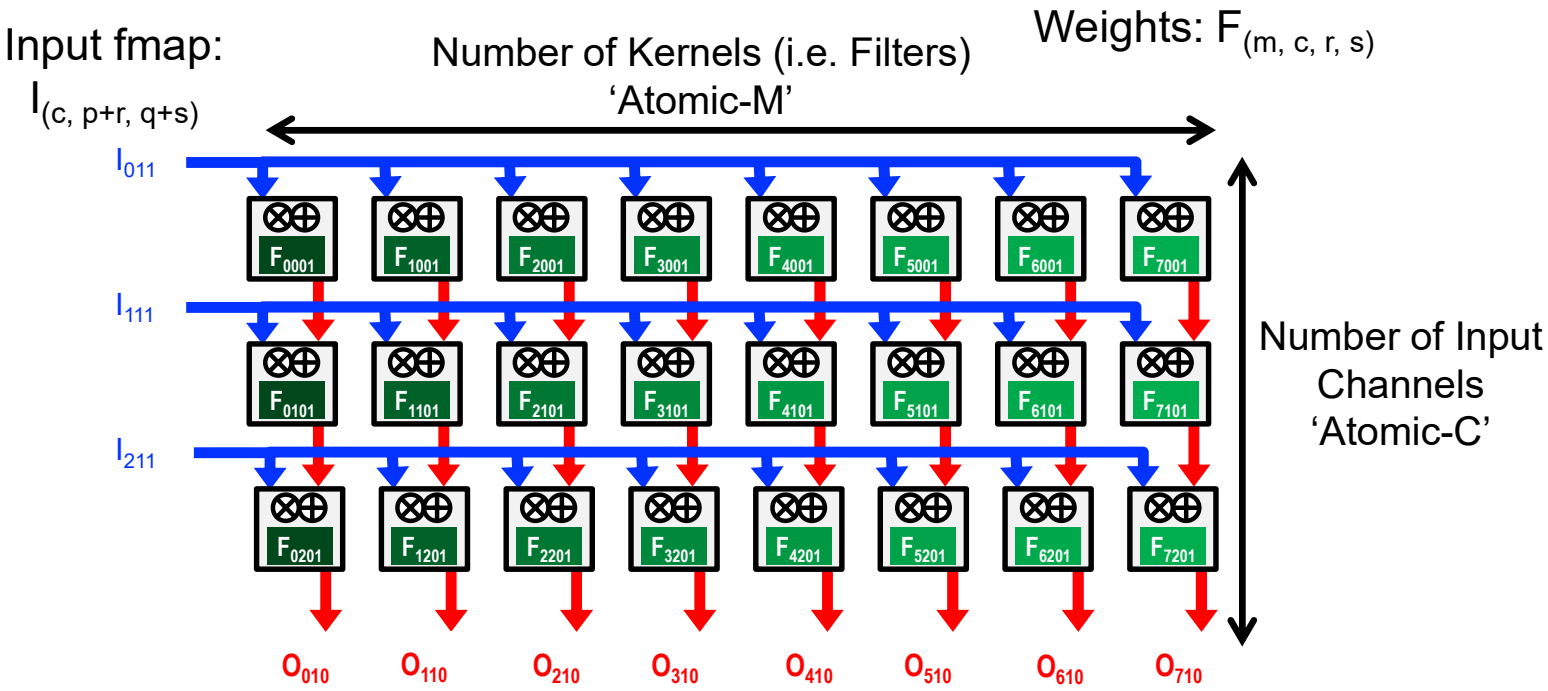

Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)



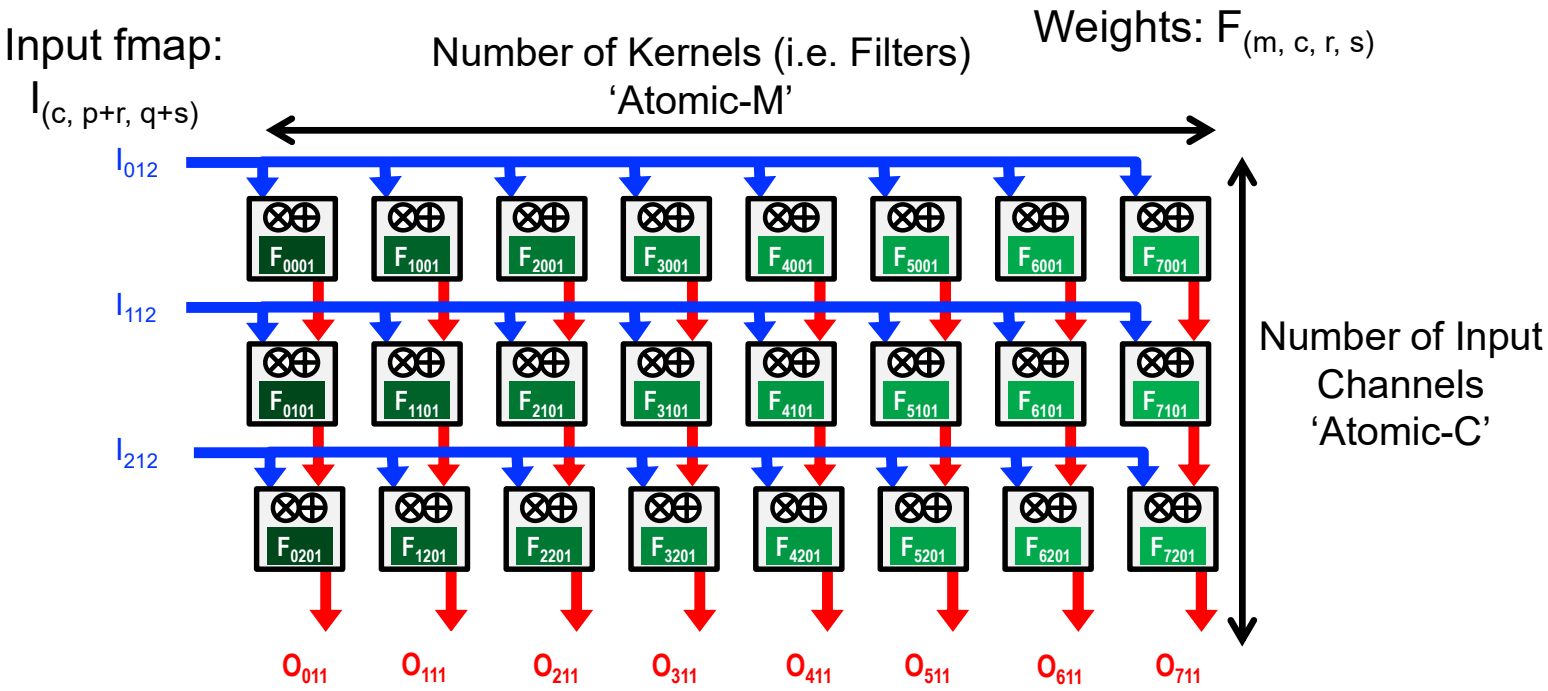

Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)




Output feature map: $O_{(m, p, q)}$

# WS Example: NVDLA (simplified)




Output feature map: $O_{(m, p, q)}$

# Loop Nest: NVDLA (simplified)

```
M = 8; C = 3;
R = 2; S = 2
P = 2; Q = 2

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

for r in [0,R):
  for s in [0,S):
    for p in [0,P):
      for q in [0,Q):
        parallel-for m in [0, M):
          parallel-for c in [0, C):
            o[m][p][q] += i[c][p+r][q+s] * f[m][c][r][s]
```

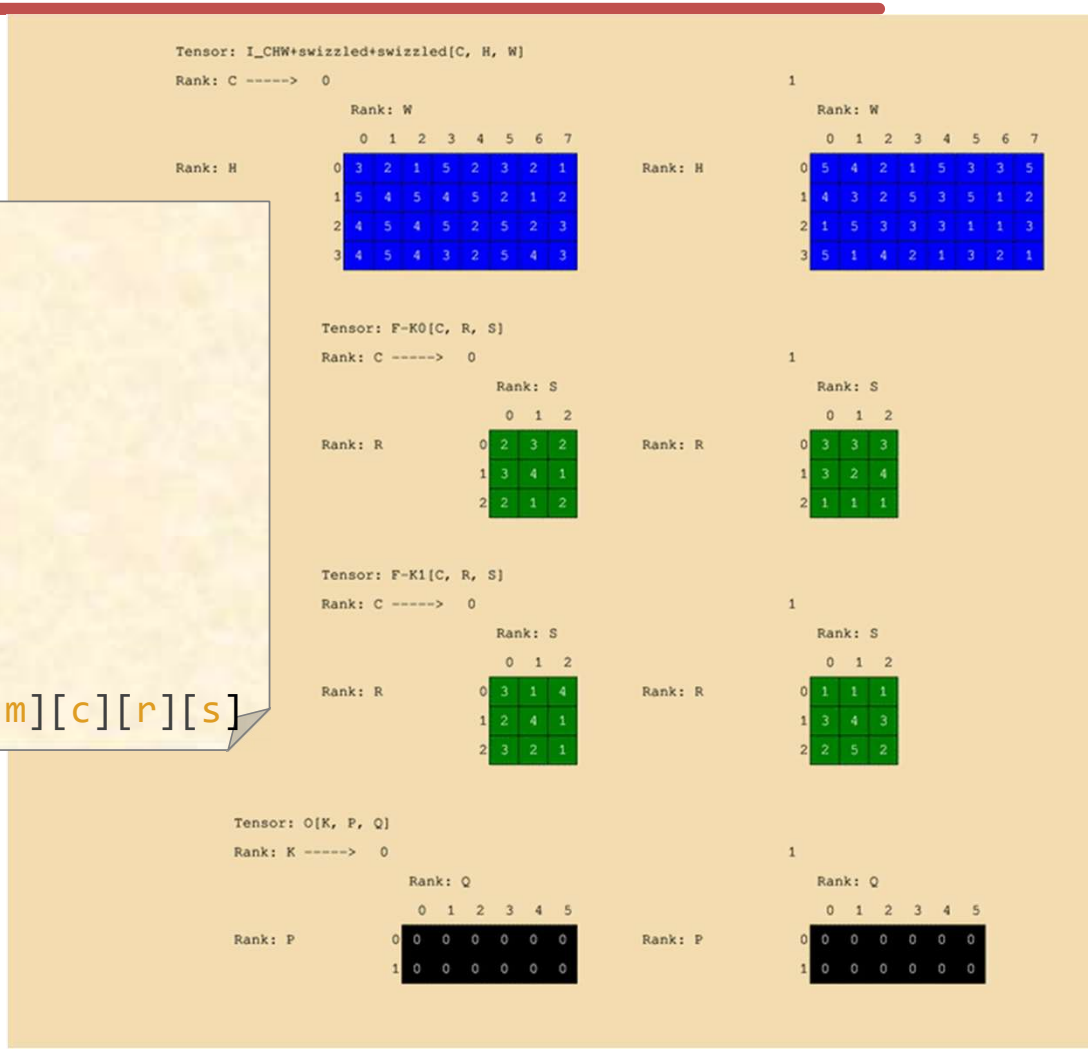
How can we tell this is weight stationary?

Top loops are r and s

# NVDLA (Simplified) - Animation

Note: Different colored highlights are different parallel PEs

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

for r in [0,R):
  for s in [0,S):
    for p in [0,P):
      for q in [0,Q):
        parallel-for m in [0, M):
          parallel-for c in [0, C):
            o[m][p][q] += i[c][p+r][q+s] * f[m][c][r][s]
```




M=K=2 H = 4
C = 2 W = 8
R = 3 P = 2
S = 3 Q = 6

Full animation at: https://csg.csail.mit.edu/6.5930/lectures/slidev/l05/#5

# CONV-layer Einsum

$$O_{m,p,q} = I_{c,p+r,q+s} \times F_{m,c,r,s}$$

Traversal order (fastest to slowest): Q, P, S, R

Parallel Ranks: C, M

# WS Example: NVDLA (simplified)

Global Buffer

Released Sept 29, 2017

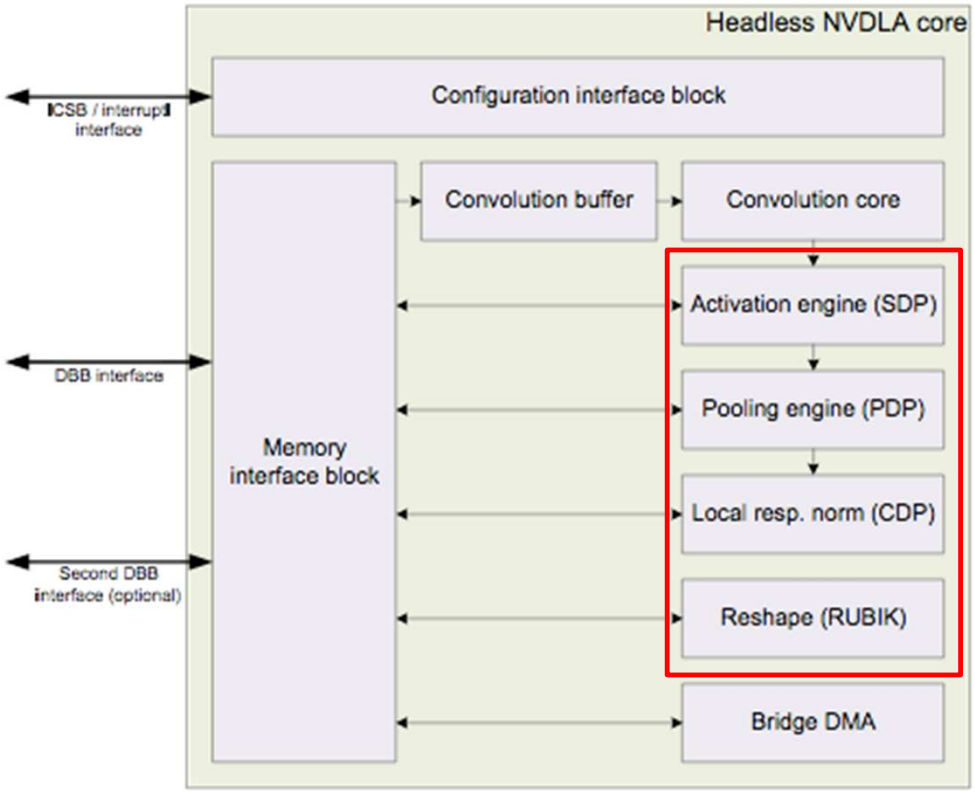



Image Source: Nvidia

http://nvdla.org

# WS Example: Nvidia DLA

- Single Data Point Operations
  - Immediately after convolution
  - Linear functions
    - e.g., bias addition, precision scaling (before writing to memory), batch normalization, element-wise operation
    - Scaling: all values, per channel, per pixel
  - Non-linear functions use LUT
    - e.g., ReLU, PReLU, sigmod, tanh
- Planar Data Operations
  - For pooling: max, min, average
- Multi-Plane Operation
  - Cross-channel processing for normalization

# Taxonomy: More Examples

- **Output Stationary (OS)**

  [**Peemen**, *ICCD* 2013] [**ShiDianNao**, *ISCA* 2015]
  [**Gupta**, *ICML* 2015] [**Moons**, *VLSI* 2016] [**Thinker**, *VLSI* 2017]

- **Weight Stationary (WS)**

  [**Chakradhar**, *ISCA* 2010] [**nn-X (NeuFlow)**, *CVPRW* 2014]
  [**Park**, *ISSCC* 2015] [**ISAAC**, *ISCA* 2016] [**PRIME**, *ISCA* 2016]
  [**TPU**, *ISCA* 2017]

What other dataflows exists? Input stationary

# Input Stationary

# Input Stationary

- Hold inputs stationary rather than outputs or weights
- Used for sparse CNNs [**Parashar**, SCNN, *ISCA* 2017]
  - Sparse CNN is where many weights are zeros
  - Advantage is the inputs are larger than weights thus require larger memory

    → reduce reads to larger memory
- Not analyzed for dense

# 1-D Convolution – Output Stationary


```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

for q in [0, Q):
      for s in [0, S):
         w = q + s
         o[q] += i[w]*f[s]
```

How can we implement input stationary with no input index?

† Assuming: 'valid' style convolution

# 1-D Convolution – Input Stationary


```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

for w in [0, W):
      for s in [0, S):
          q = w - s
          o[q] += i[w]*f[s]
```

Beware q must be >= 0 and <Q

† Assuming: 'valid' style convolution

# 1-D Convolution Einsum + IS

$$O_q = I_{q+s} \times F_s$$

$$O_q^Q = I_{q+s}^W \times F_s^S$$

Add superscript to indicate name of rank

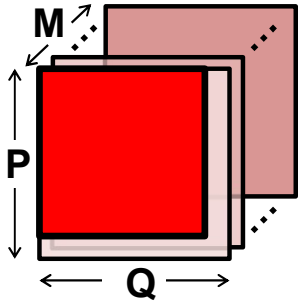
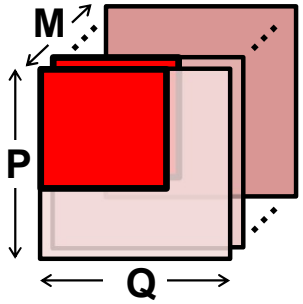
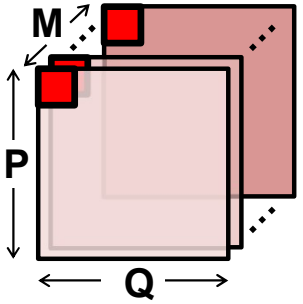
Note: w = q+s, so q = w-s, and therefore

$$O_{w-s}^Q = I_w^W \times F_s^S$$

Traversal order (fastest to slowest): S, W

# Output Stationary (revisited)

# Variants of Output Stationary (Different Tiling)

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

# Einsums for Different OS Tiling

$$O_{m,p,q} = I_{c,p+r,q+s,} \times F_{m,c,r,s}$$

**$OS_A$**

$$O_{m,p1,q1,p0,q0} = I_{c,p1,q1,p0+r,q0+s,} \times F_{m,c,r,s}$$

Parallel: P0, Q0

**$OS_B$**

$$O_{m1,m0,p1,q1,p0,q0} = I_{c,p1,q1,p0+r,q0+s,} \times F_{m1,m0,c,r,s}$$

Parallel: M0, P0, Q0

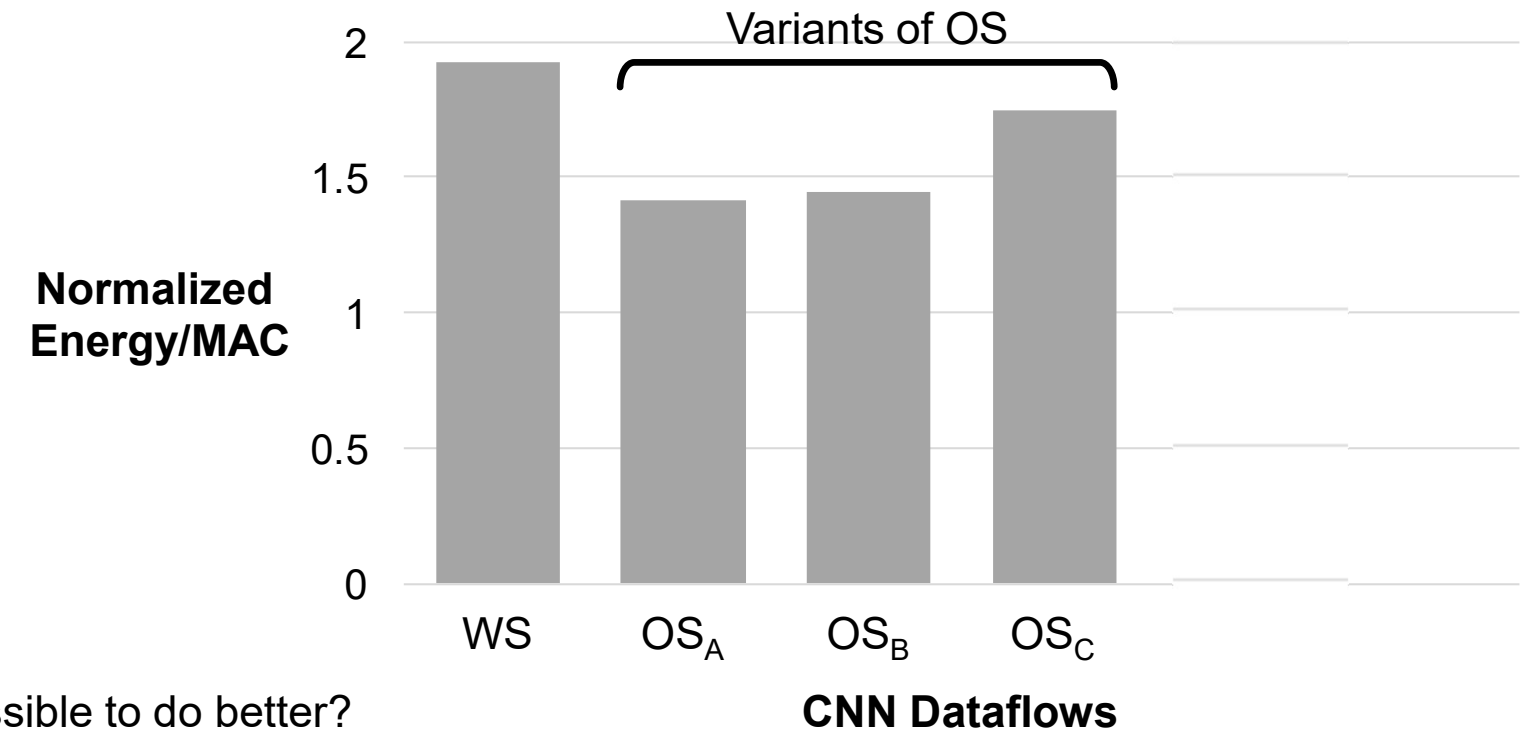
**$OS_C$**

$$O_{m1,m0,p,q} = I_{c,p+r,q+s,} \times F_{m1,m0,c,r,s}$$

Parallel: M0

# Energy Efficiency Comparison

- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16




Is it possible to do better?

[Chen et al., ISCA 2016]

# Summary of Processing Order Optimizations

- **Partitioning**
  - **Goal:** Create chunks of data to enable control over temporal reuse in hierarchy of buffers and spatially for use by multiple processing units.
- **Dataflow** (loop order)
  - **Goal:** Increase reuse for given data type (weight, input, output) by increasing stationarity. Align traversal with storage order for concordant traversal (improve spatial locality); **(this class!)**
- **Dataplacement**
  - **Goal:** Control placement of data in specific buffers to reduce data movement (including bypass)
- **Compute Placement** (parallel or temporal loop)
  - **Goal:** Reduce cycles by processing multiple MACs in parallel
- **Partition Sizing**
  - **Goal:** Determine the exact amounts of data processed temporarily and spatially

# Looptree – Dataflow and Dataplacement

# Example Workload: Matrix Multiplication

Matrix-matrix multiplication in *Einsum notation*:

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

# Example Workload: Matrix Multiplication

Matrix-matrix multiplication in *Einsum notation*:

Note specifically:

- *Ranks* of tensors (*e.g.*, $A$ has ranks $M$ and $NA$)

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

# Example Workload: Matrix Multiplication

Matrix-matrix multiplication in *Einsum notation*:

Note specifically:

- *Ranks* of tensors (*e.g.*, $A$ has ranks $M$ and $NA$)
- Binary operation (*e.g.*, multiplication)

$$A_{m,na} = I_{m,ni} \times W A_{ni,na}$$

# Example Workload: Matrix Multiplication

Matrix-matrix multiplication in *Einsum notation*:

Note specifically:

- *Ranks* of tensors (*e.g.*, $A$ has ranks $M$ and $NA$)
- Binary operation (*e.g.*, multiplication)
- Implicit summation over right-hand-side-only rank variable (*e.g.*, $ni$)

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

# Example Workload: Matrix Multiplication

Matrix-matrix multiplication in *Einsum notation*:

Note specifically:

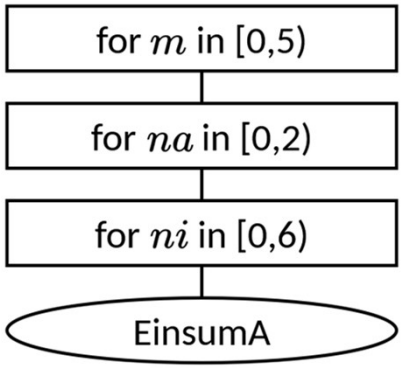
- *Ranks* of tensors (*e.g.*, $A$ has ranks $M$ and $NA$)
- Binary operation (*e.g.*, multiplication)
- Implicit summation over right-hand-side-only rank variable (*e.g.*, $ni$)
- Does not assume an order for operations

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

# Choosing a Dataflow

In the LoopTree notation,

*dataflow* is the order of *loop nodes*



# Partitioning: …

Original Einsum:

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

# Partitioning a Rank + …

Original Einsum:

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

Partition rank $NI$:

$$A_{m,na} = I_{m,ni1,ni0} \times WA_{ni1,ni0,na}$$

# Partitioning a Rank + Swizzling Ranks

Original Einsum:

$$A_{m,na} = I_{m,ni} \times WA_{ni,na}$$

Partition rank $NI$:

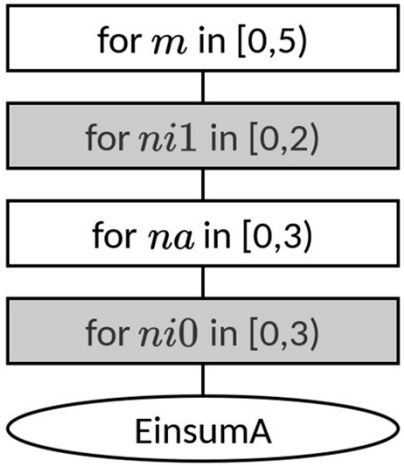
$$A_{m,na} = I_{m,ni1,ni0} \times WA_{ni1,ni0,na}$$

Swizzling ranks:

$$A_{m,na} = I_{m,ni1,ni0} \times WA_{ni1,na,ni0}$$

# Specifying a Dataflow with Partitioned Ranks

Dataflow in LoopTree is post-partitioning:



# Specifying a Dataplacment

*Dataplacement*

A specification of how capacity of a buffer should be utilized for various tensors.

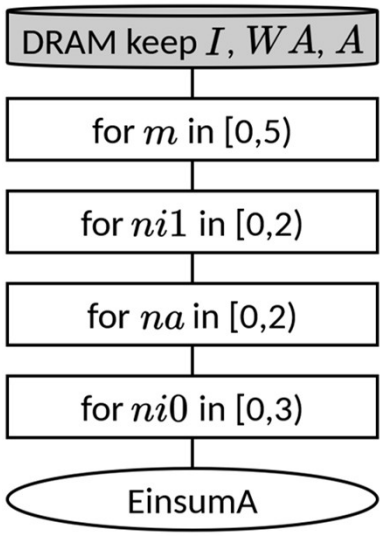
# Specifying a Storageplan

In LoopTree, add a dataplacement by placing storage nodes:

# Specifying a Storageplan

In LoopTree, add a dataplacement by placing storage nodes:

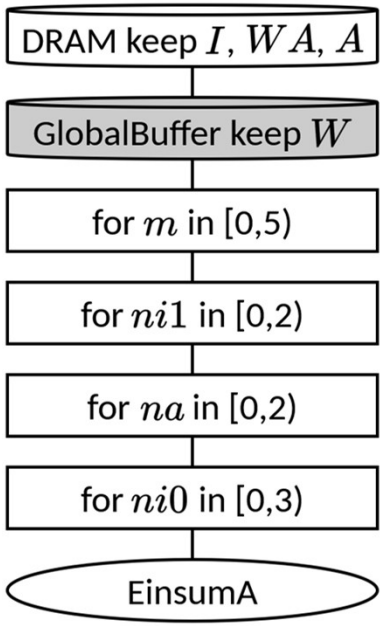
- DRAM, as backing storage, keeps all tensors.



# Specifying a Storageplan

In LoopTree, add a dataplacement by placing storage nodes:

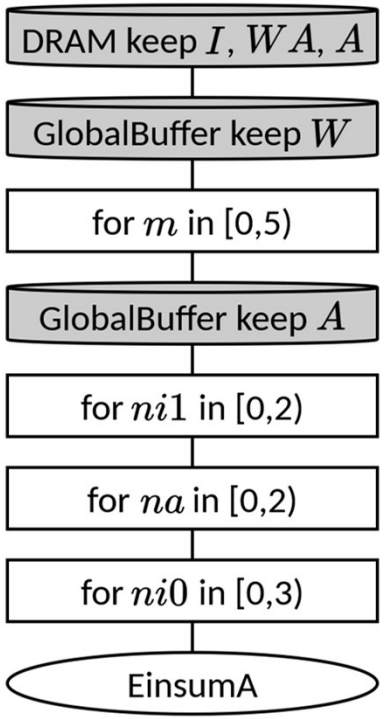
- DRAM, as backing storage, keeps all tensors.
- Fetch all weights *WA* into on-chip GlobalBuffer



# Specifying a Storageplan

In LoopTree, add a dataplacement by placing storage nodes:

- DRAM, as backing storage, keeps all tensors.
- Fetch all weights *WA* into on-chip GlobalBuffer
- Every iteration of "for *m*" fetch a chunk of *A* into GlobalBuffer



# Specifying a Storageplan

In LoopTree, add a dataplacement by placing storage nodes:

- DRAM, as backing storage, keeps all tensors.
- Fetch all weights *WA* into on-chip GlobalBuffer
- Every iteration of "for *m*" fetch a chunk of *A* into GlobalBuffer
- Every iteration of "for ni1" fetch a chunk of I into GlobalBuffer