EE-508: Hardware Foundations for Machine Learning Kernel Design

University of Southern California

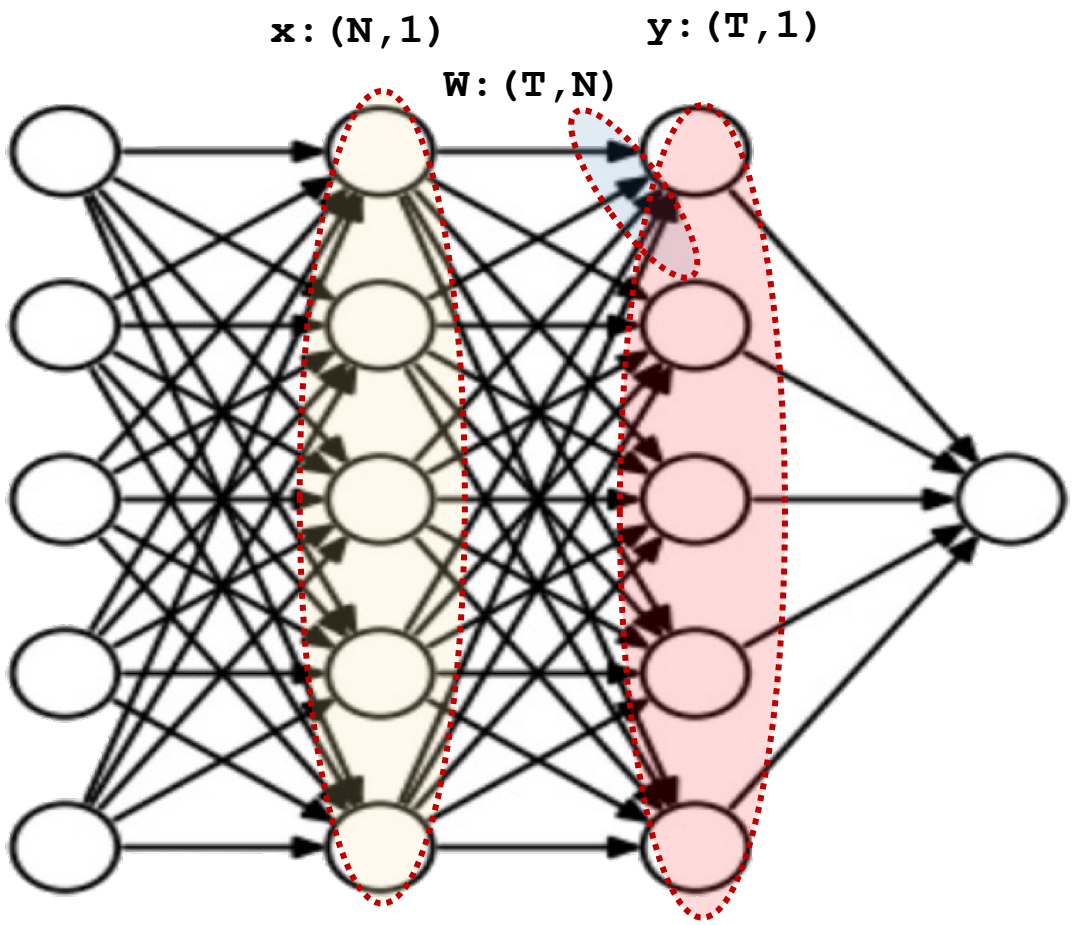
Ming Hsieh Department of Electrical and Computer Engineering

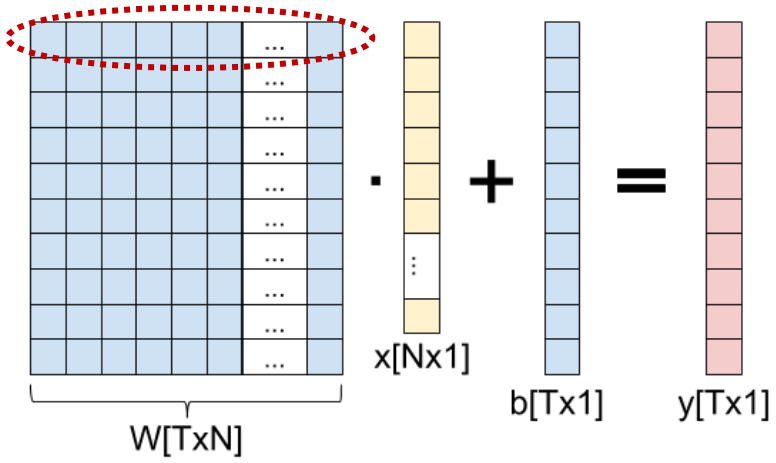
Instructor:
Arash Saifhashemi

Kernel Design

FC Layer

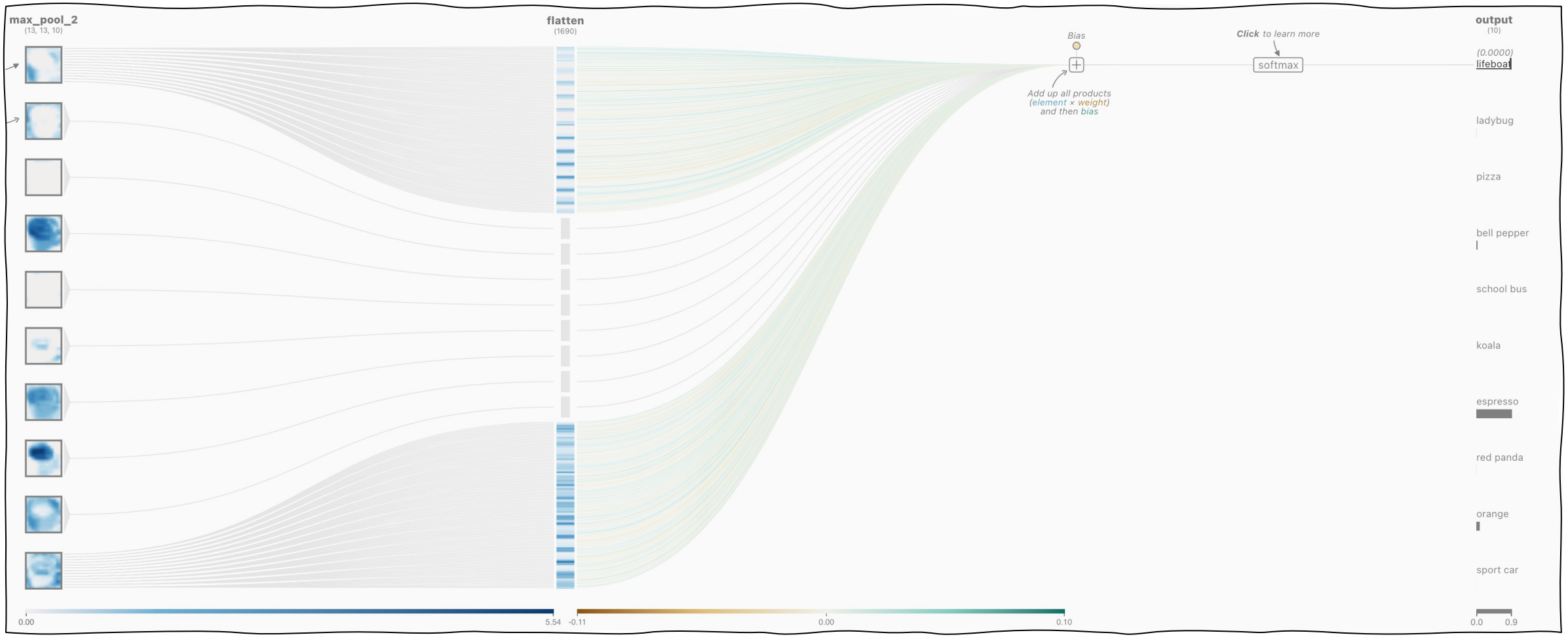
FC Layer



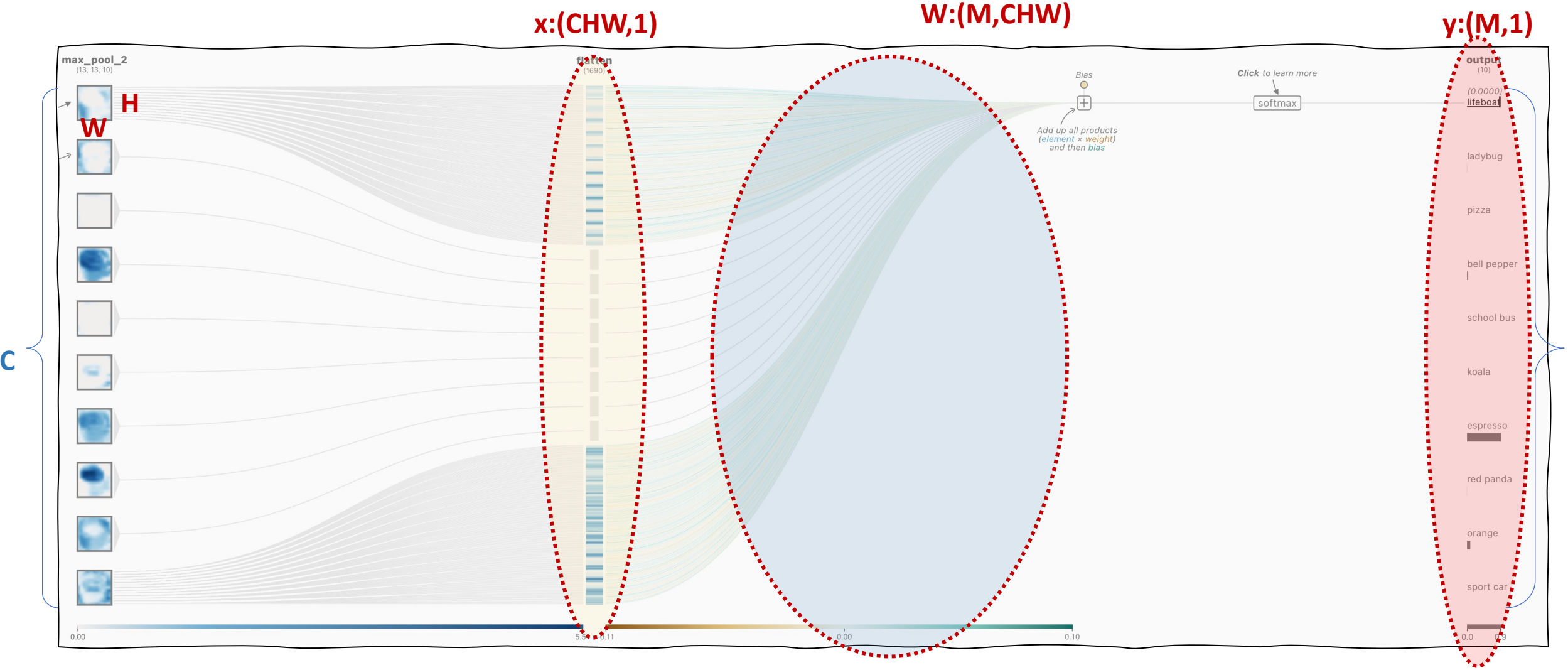


Source:
thegreenplace.net

$$y = Wx + b$$



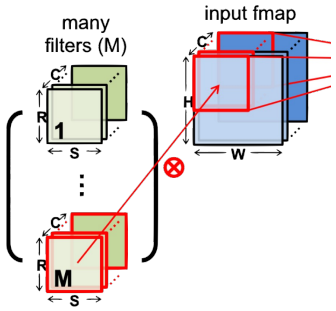
https://poloclub.github.io/cnn-explainer/



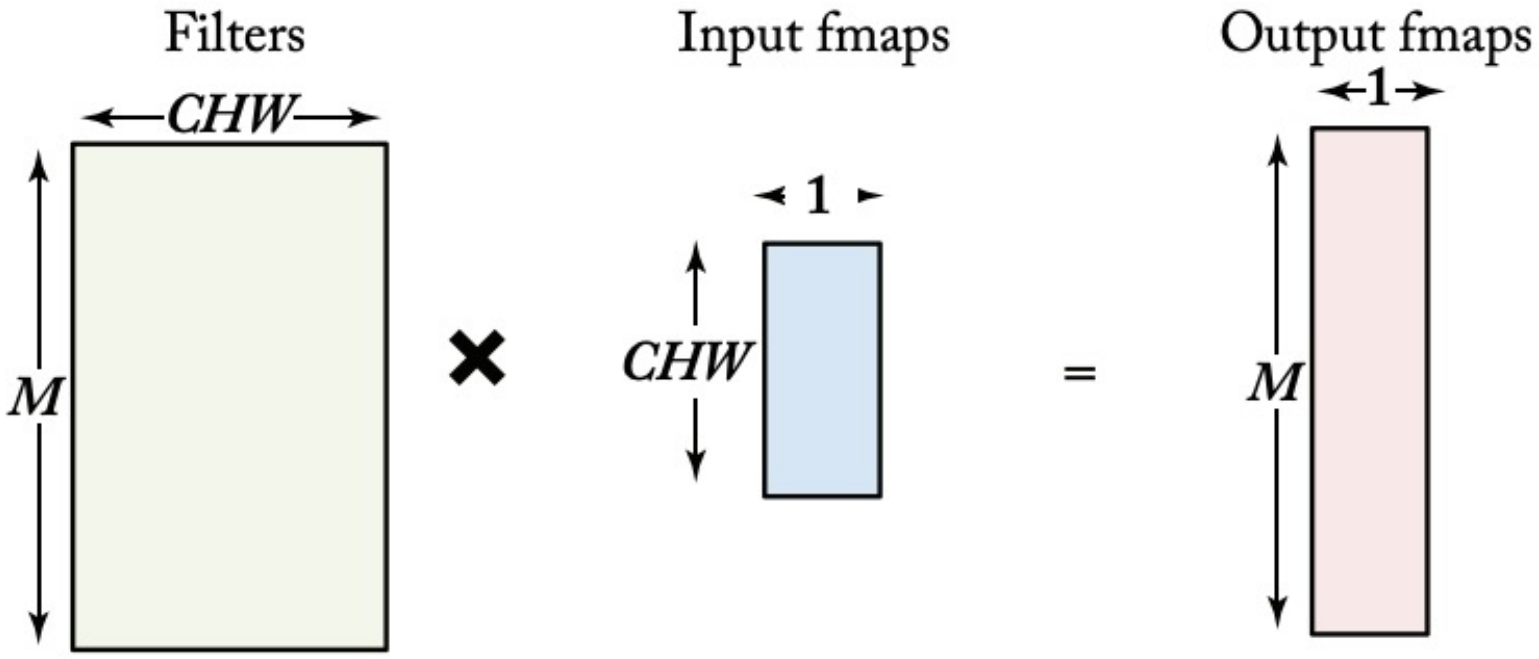
https://poloclub.github.io/cnn-explainer/

$$y = Wx + b$$

FC Layer in CNN with GEMM

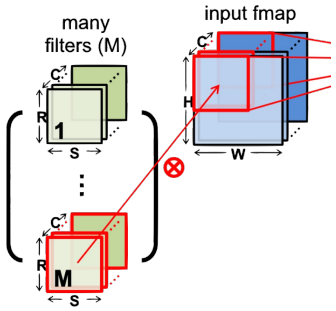


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels

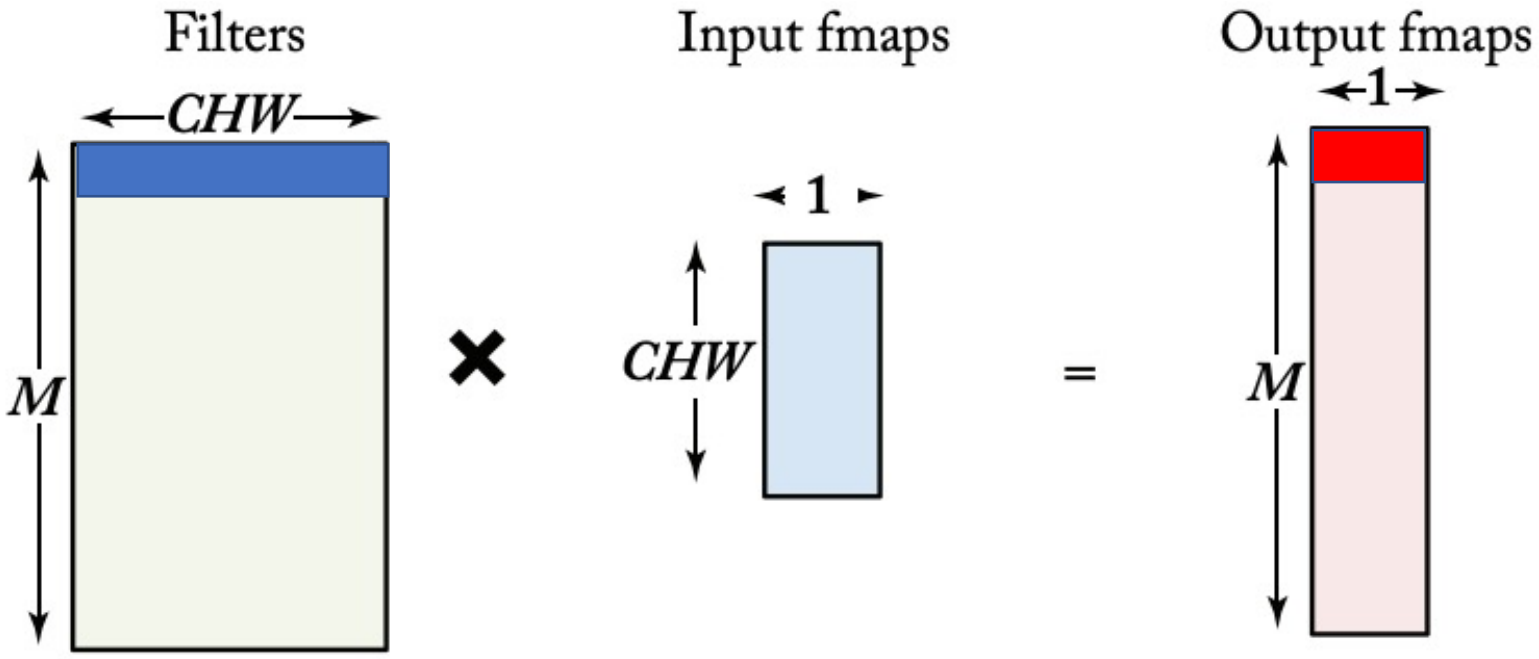


$$O_m = \sum_{c,h,w} I_{chw} \times F_{m,chw}$$

FC Layer in CNN with GEMM

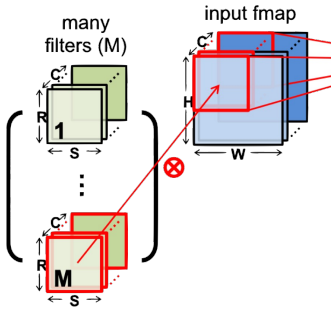


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels

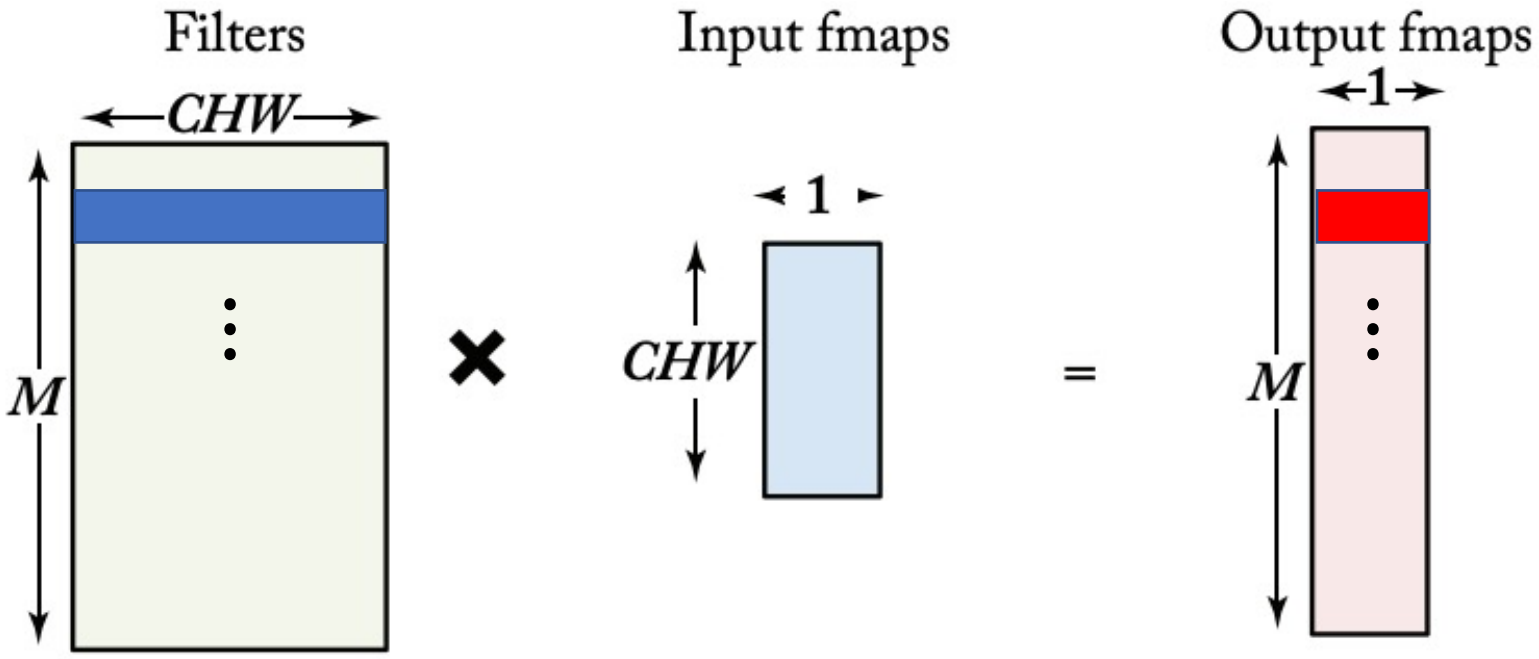


$$O_m = \sum_{c,h,w} I_{chw} \times F_{m,chw}$$

FC Layer in CNN with GEMM

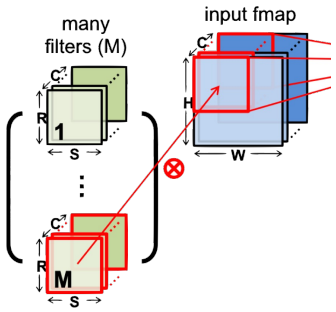


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels

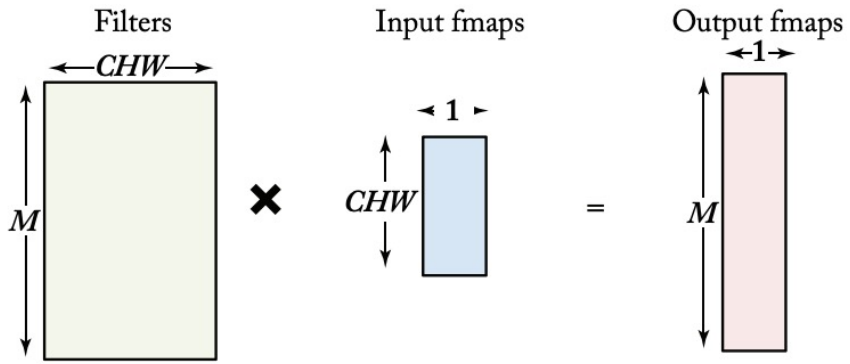


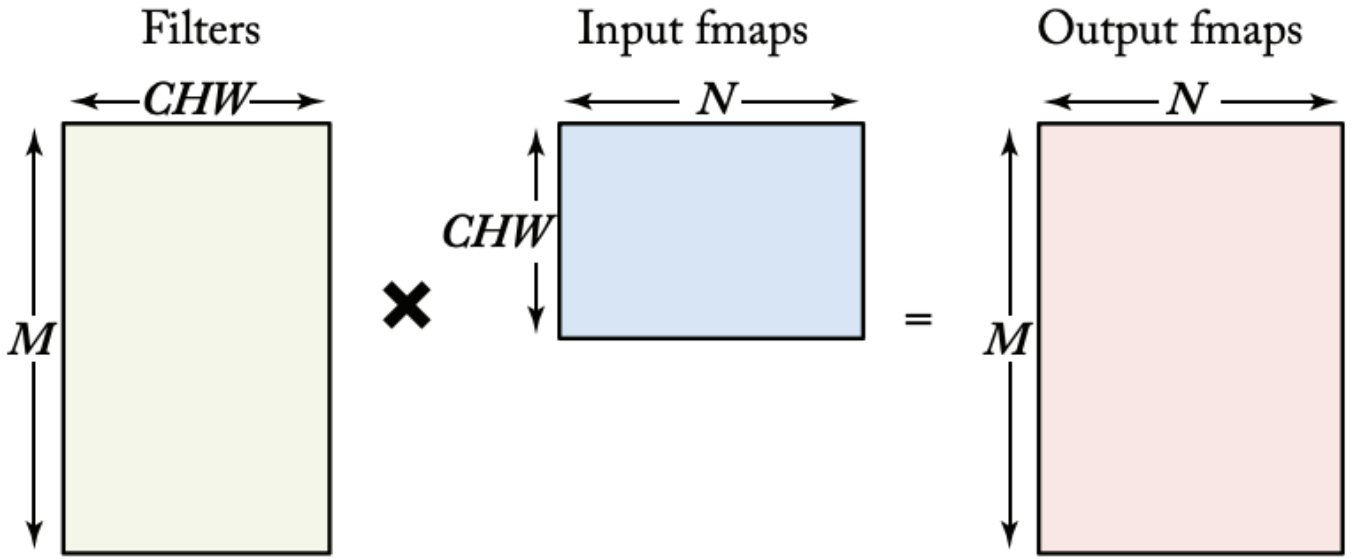
$$O_m = \sum_{c,h,w} I_{chw} \times F_{m,chw}$$

FC Layer in CNN with GEMM (Multiple Inputs)

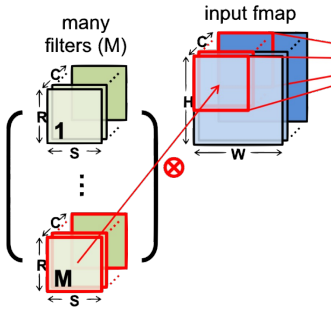


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels
N: Batch size

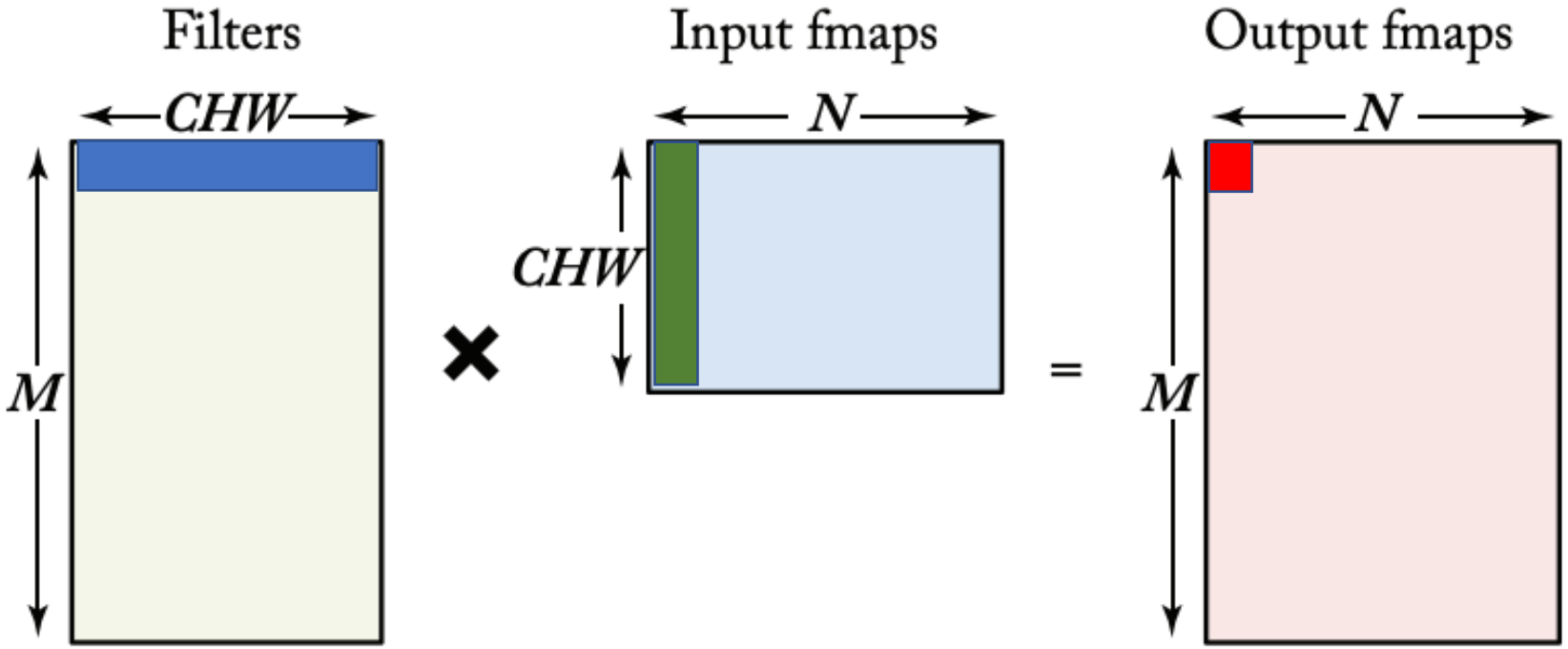




FC Layer in CNN with GEMM (Multiple Inputs)

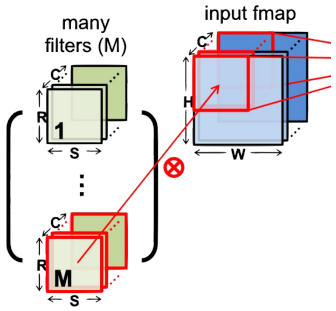


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels
N: Batch size

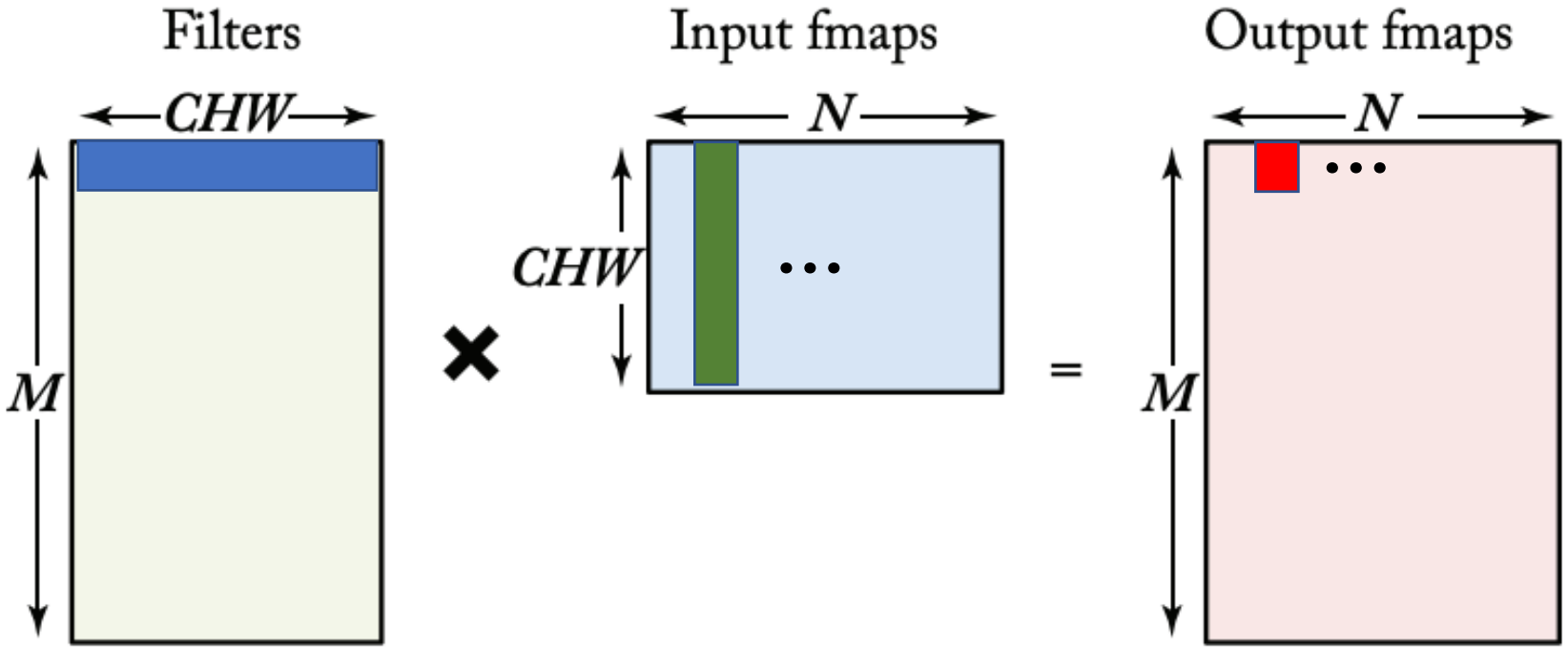


$$O_{m,n} = \sum_{c,h,w} I_{chw,n} \times F_{m,chw}$$

FC Layer in CNN with GEMM (Multiple Inputs)

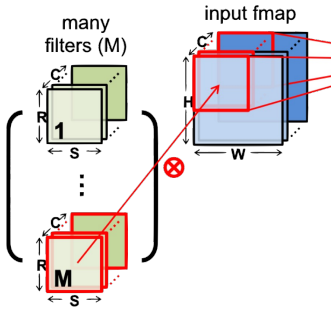


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels
N: Batch size

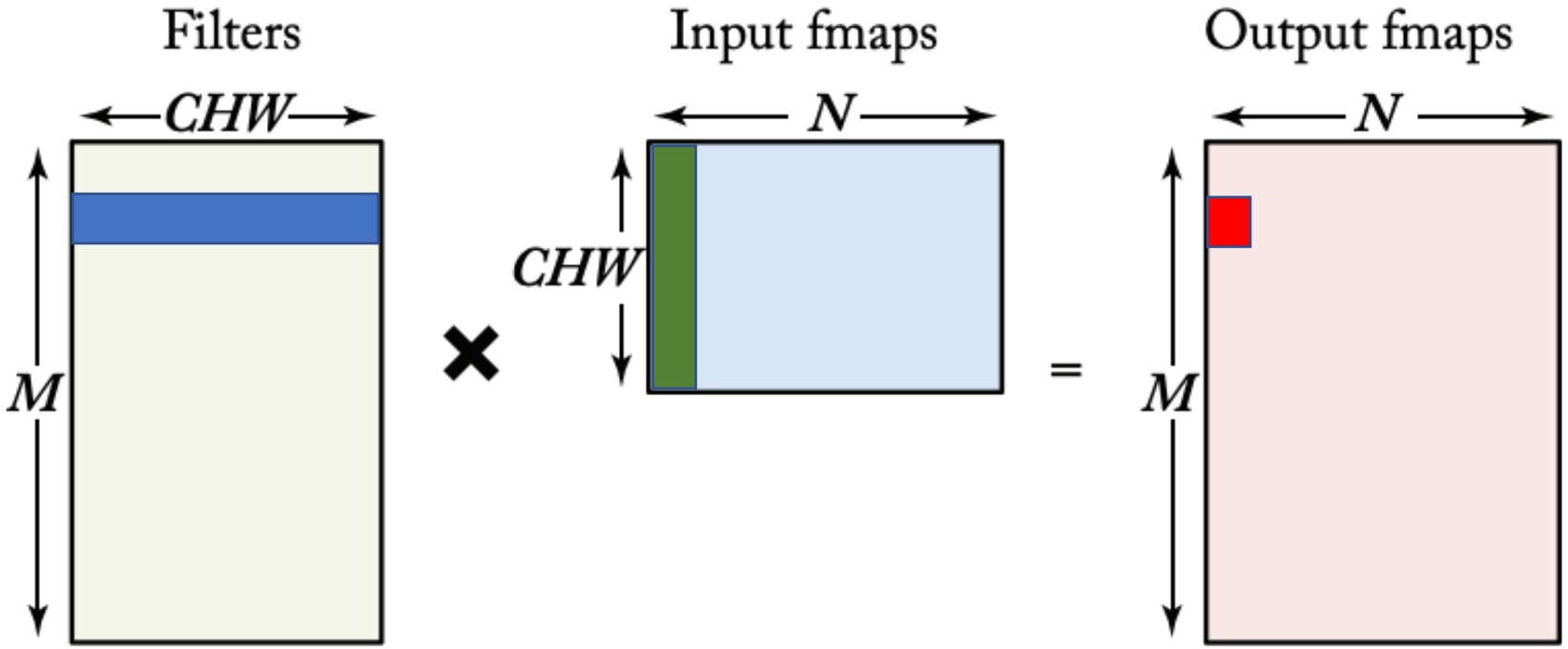


$$O_{m,n} = \sum_{c,h,w} I_{chw,n} \times F_{m,chw}$$

FC Layer in CNN with GEMM (Multiple Inputs)

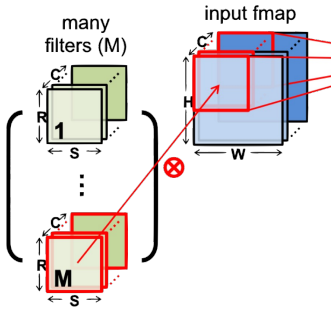


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels
N: Batch size

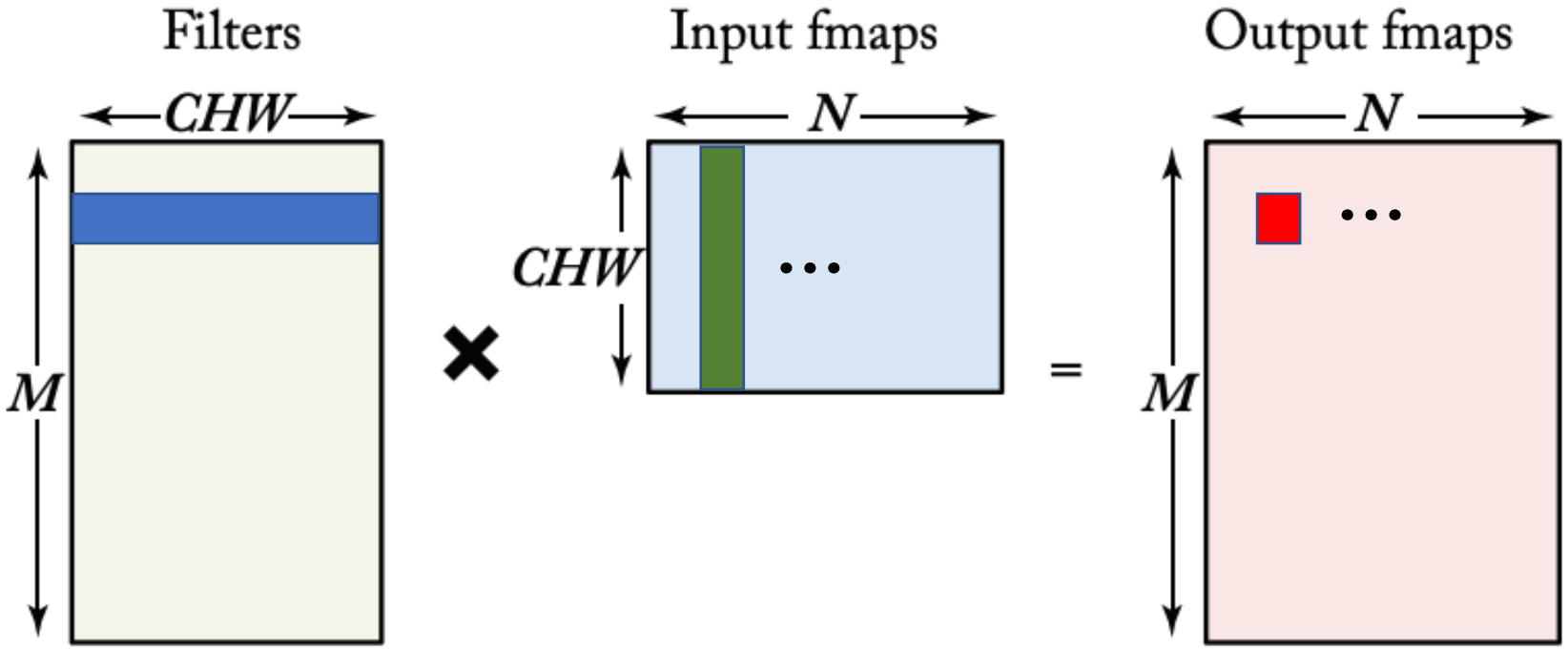


$$O_{m,n} = \sum_{c,h,w} I_{chw,n} \times F_{m,chw}$$

FC Layer in CNN with GEMM (Multiple Inputs)

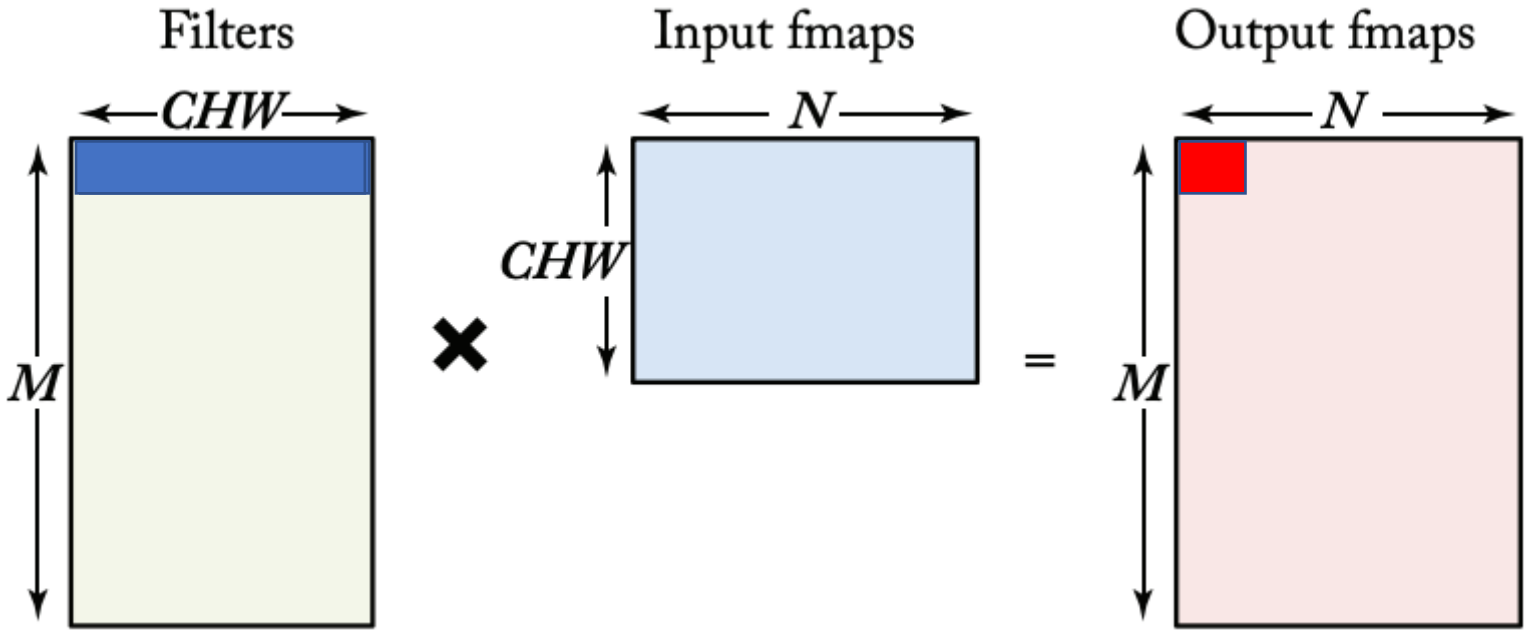


IFM: HxW
C: Number IFM channels
M: Number of OFM Channels
N: Batch size



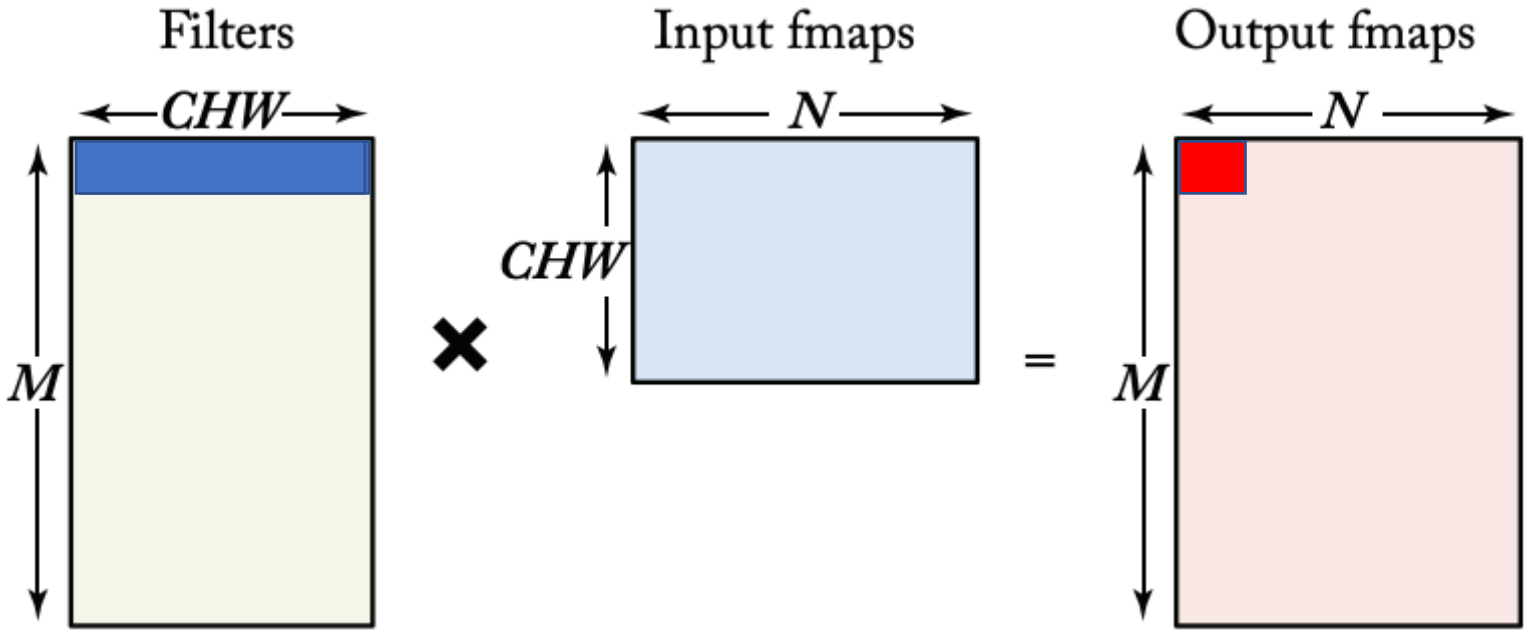
$$O_{m,n} = \sum_{c,h,w} I_{chw,n} \times F_{m,chw}$$

Reducing Memory Cost



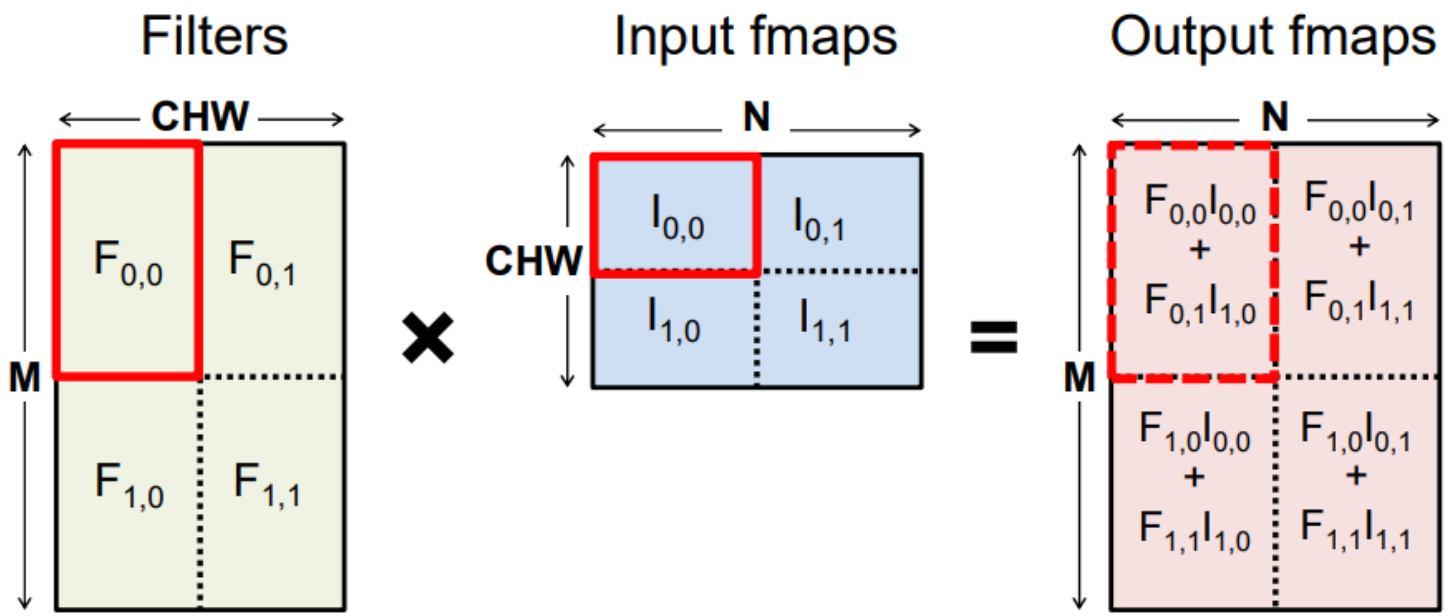
Are we doing reuse here?
(Memory is large enough)

Reducing Memory Cost



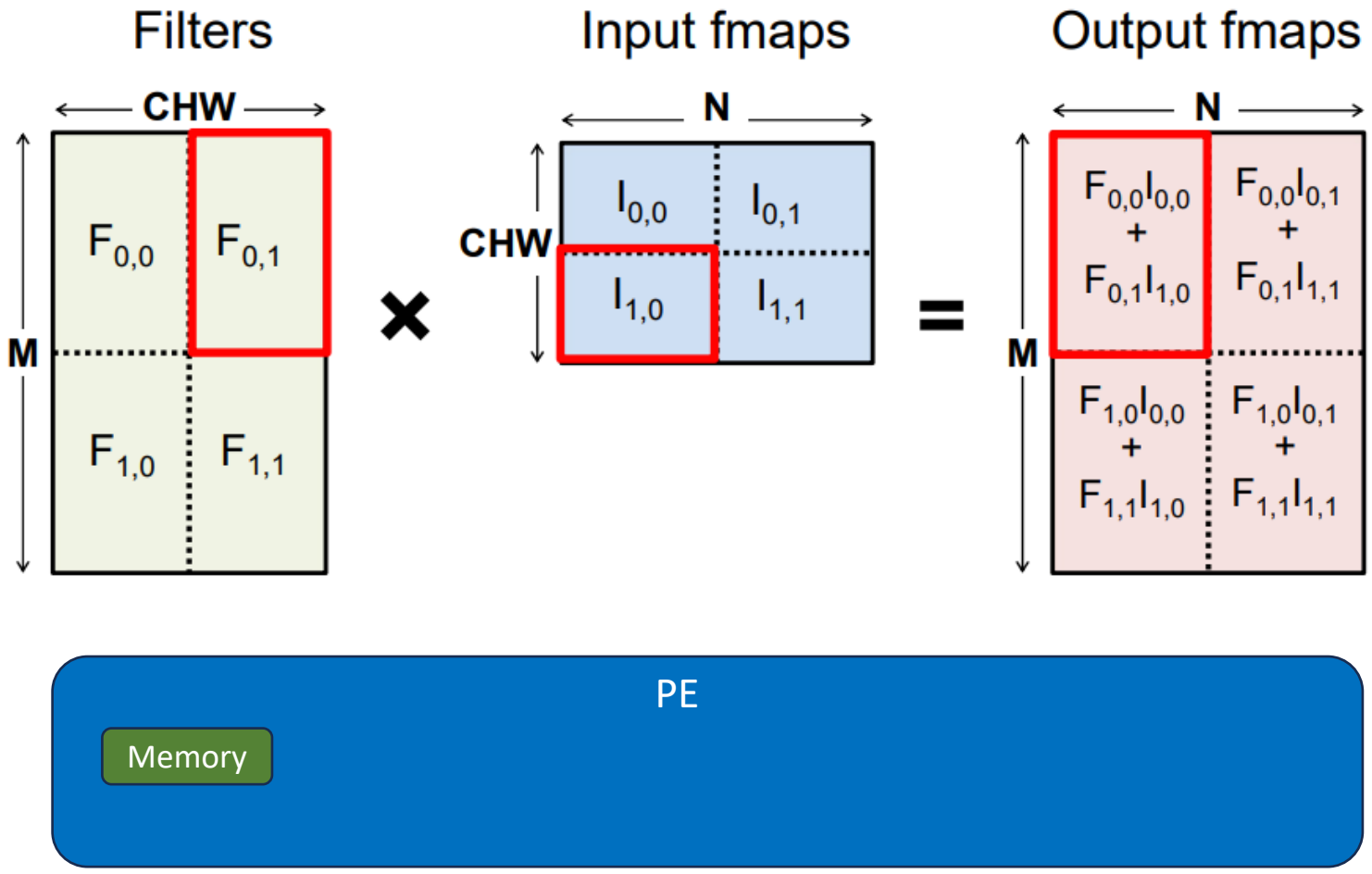
Are we doing reuse here?
(Memory is not large enough)

Tiling

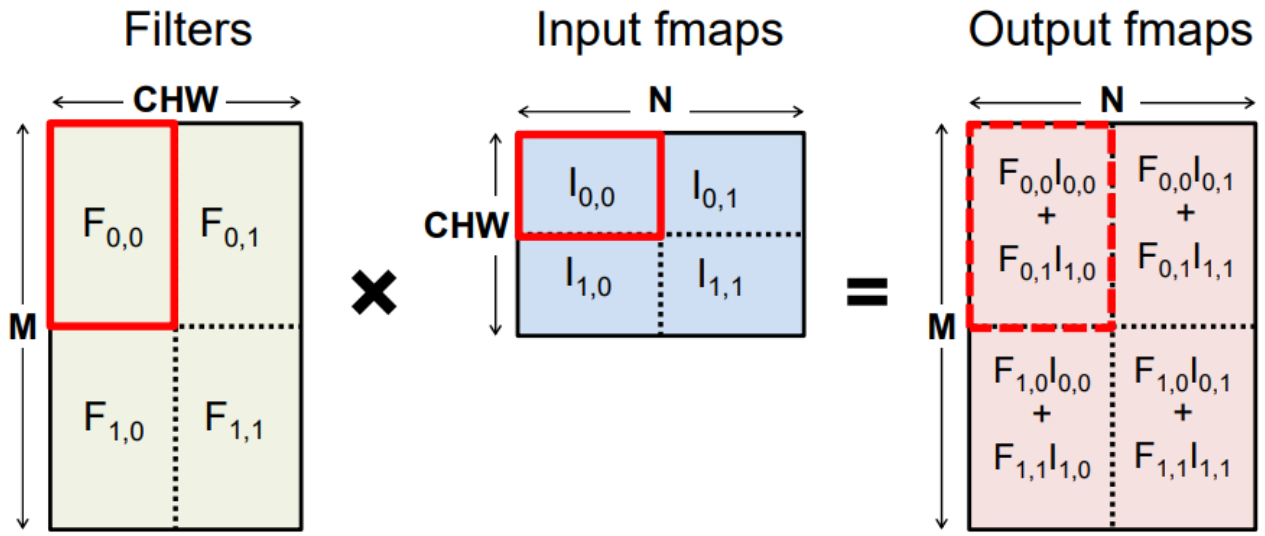

Filters
Input fmaps
Output fmaps
CHW
N
M
CHW
N
M
$F_{0,0}$
$F_{0,1}$
$F_{1,0}$
$F_{1,1}$
$I_{0,0}$
$I_{0,1}$
$I_{1,0}$
$I_{1,1}$
$F_{0,0}I_{0,0}$ + $F_{0,1}I_{1,0}$
$F_{0,0}I_{0,1}$ + $F_{0,1}I_{1,1}$
$F_{1,0}I_{0,0}$ + $F_{1,1}I_{1,0}$
$F_{1,0}I_{0,1}$ + $F_{1,1}I_{1,1}$


PE
Memory

Tiling

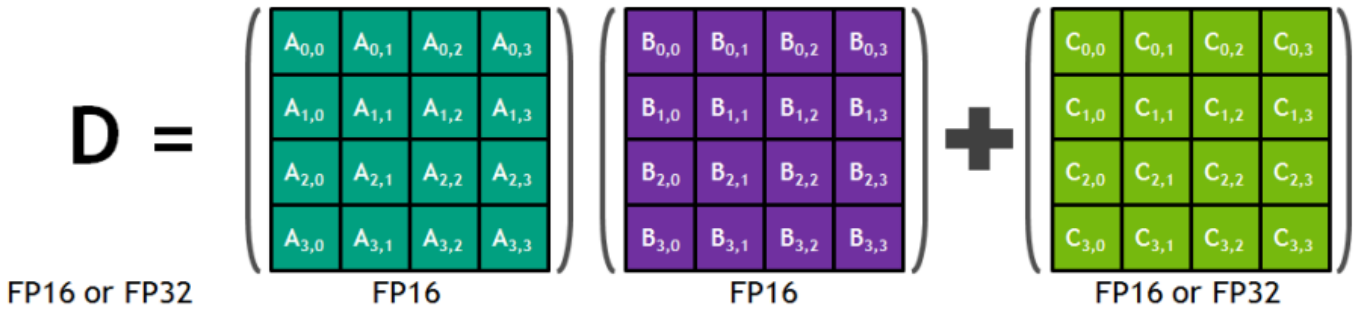

Filters
Input fmaps
Output fmaps
CHW
N
M
CHW
N
M
$F_{0,0}$
$F_{0,1}$
$F_{1,0}$
$F_{1,1}$
$I_{0,0}$
$I_{0,1}$
$I_{1,0}$
$I_{1,1}$
$F_{0,0}I_{0,0}$ + $F_{0,1}I_{1,0}$
$F_{0,0}I_{0,1}$ + $F_{0,1}I_{1,1}$
$F_{1,0}I_{0,0}$ + $F_{1,1}I_{1,0}$
$F_{1,0}I_{0,1}$ + $F_{1,1}I_{1,1}$
PE
Memory

How to Use Tiling (AKA Blocking)?



- **Goal**: Optimize matrix operations for cache utilization.
- **Method**:
  - Divide large matrices into smaller submatrices or "tiles."
  - Perform operations on tiles to reduce cache misses.
- **Implementation**: Matrix Multiplication (GEMM)
  - CPU: OpenBLAS, Intel MKL, etc.
  - GPU: cuBLAS, cuDNN, etc.
- Optimization usually involves proper tiling to storage hierarchy

Example: HMMA Instruction in GV100 GPU



$$D = A \times B + C$$

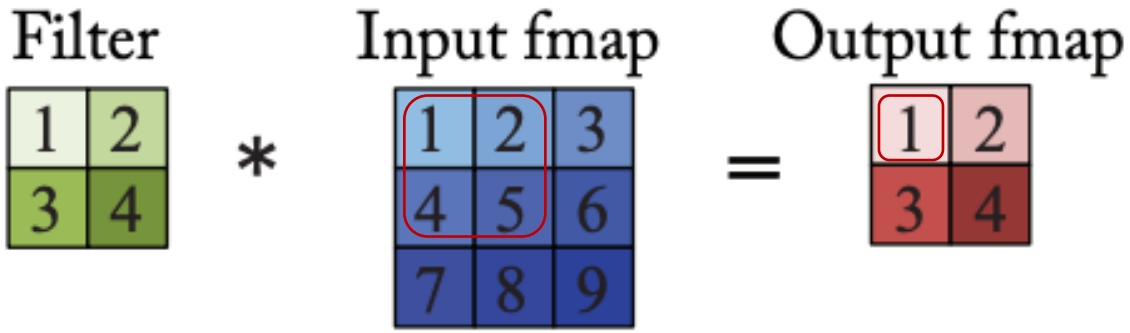
- ***A, B****:* 4×4 matrix of FP16 (half-precision) values.
- ***C****:* 4×4 matrix of FP32 (single-precision) values.
- ***D:*** 4×4 matrix

- ***Number of FP16 operands:*** *3*16 = 48 (Operands from A, B, C)*
- ***Number of multiplications:*** *4*4*4 = 64*
- ***Number of additions:*** *4*4*4 = 64 (Counting all additions for $A \times B + C$)*

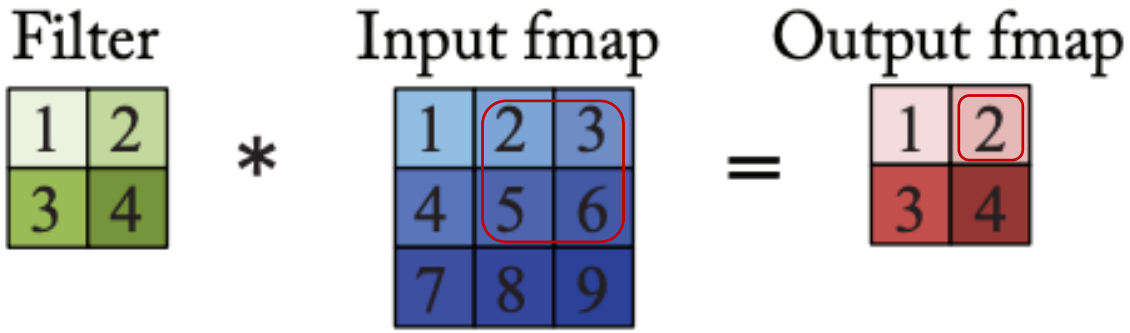
Kernel Design

Convolution Layer

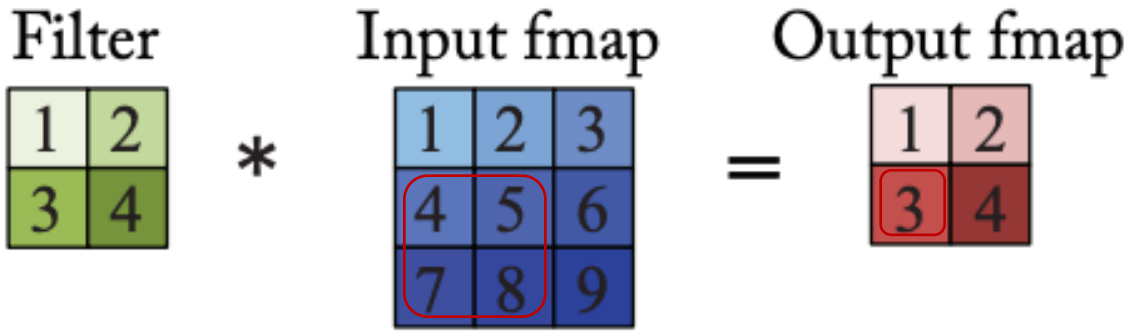
Convolution Layer



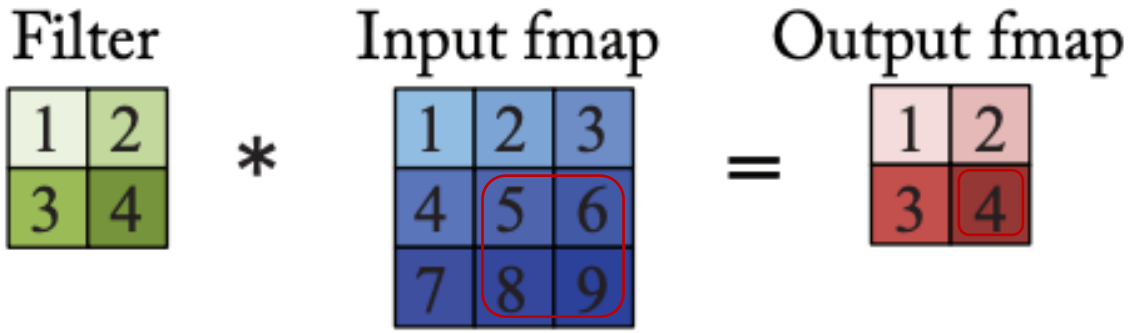
Convolution Layer



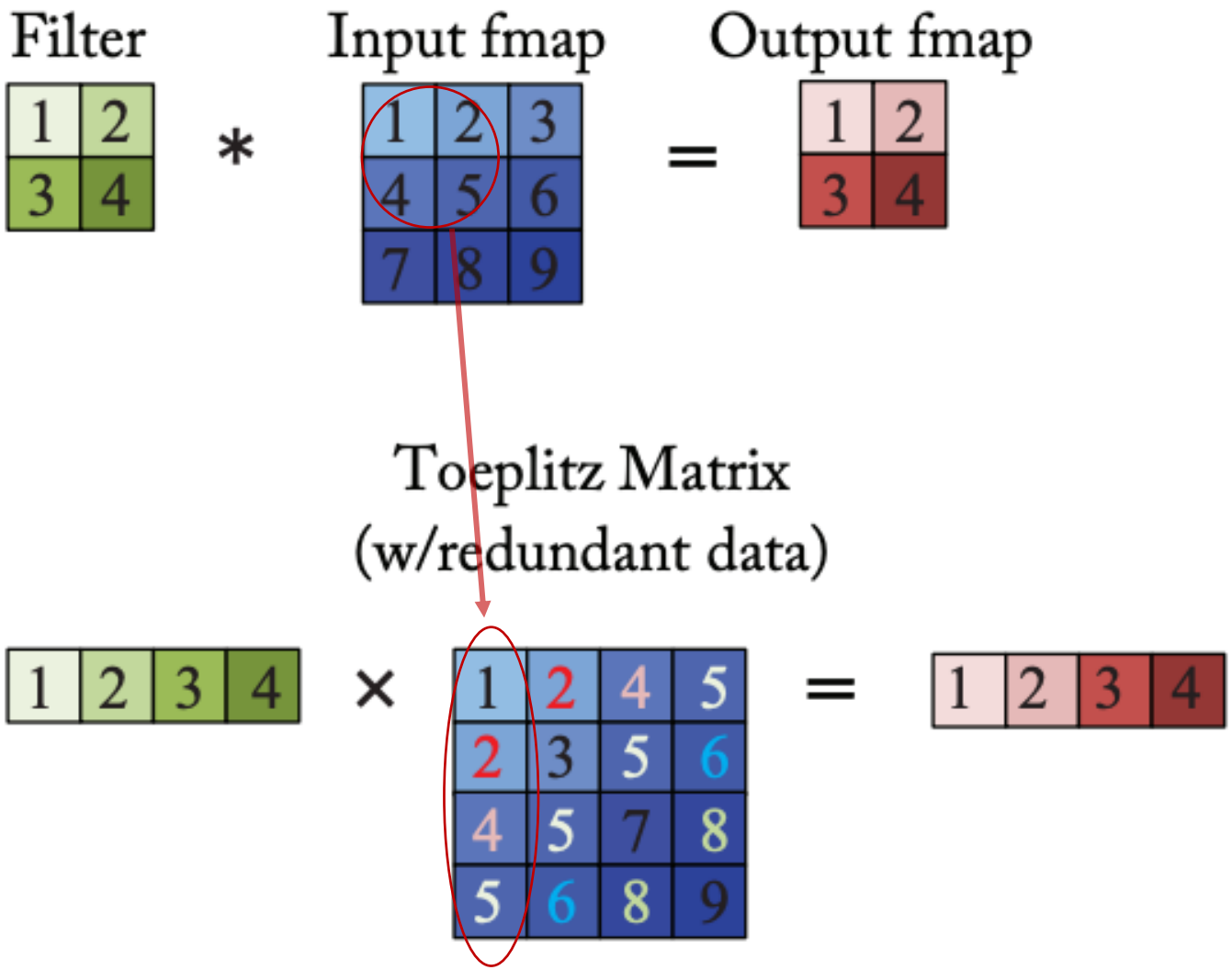
Convolution Layer



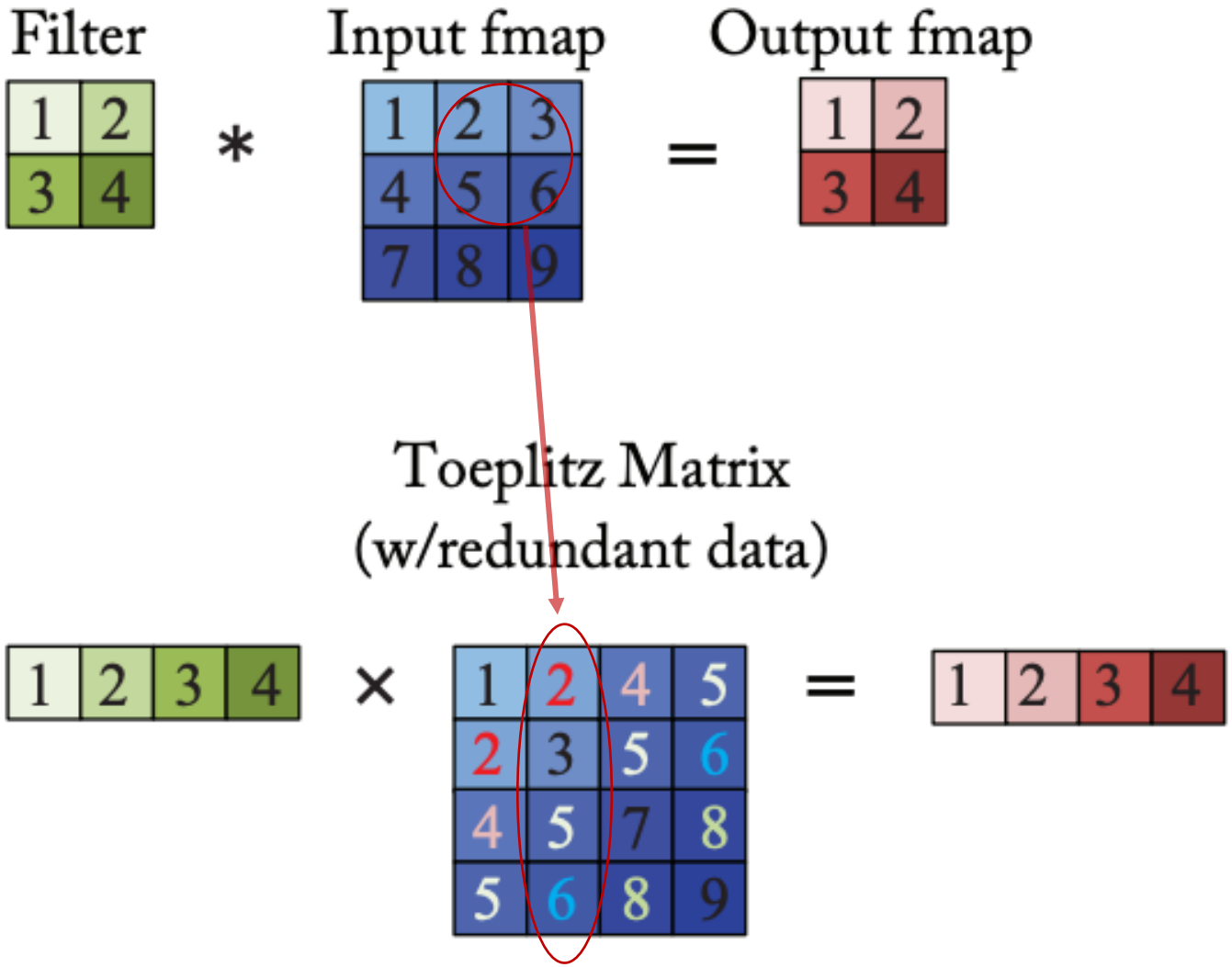
Convolution Layer



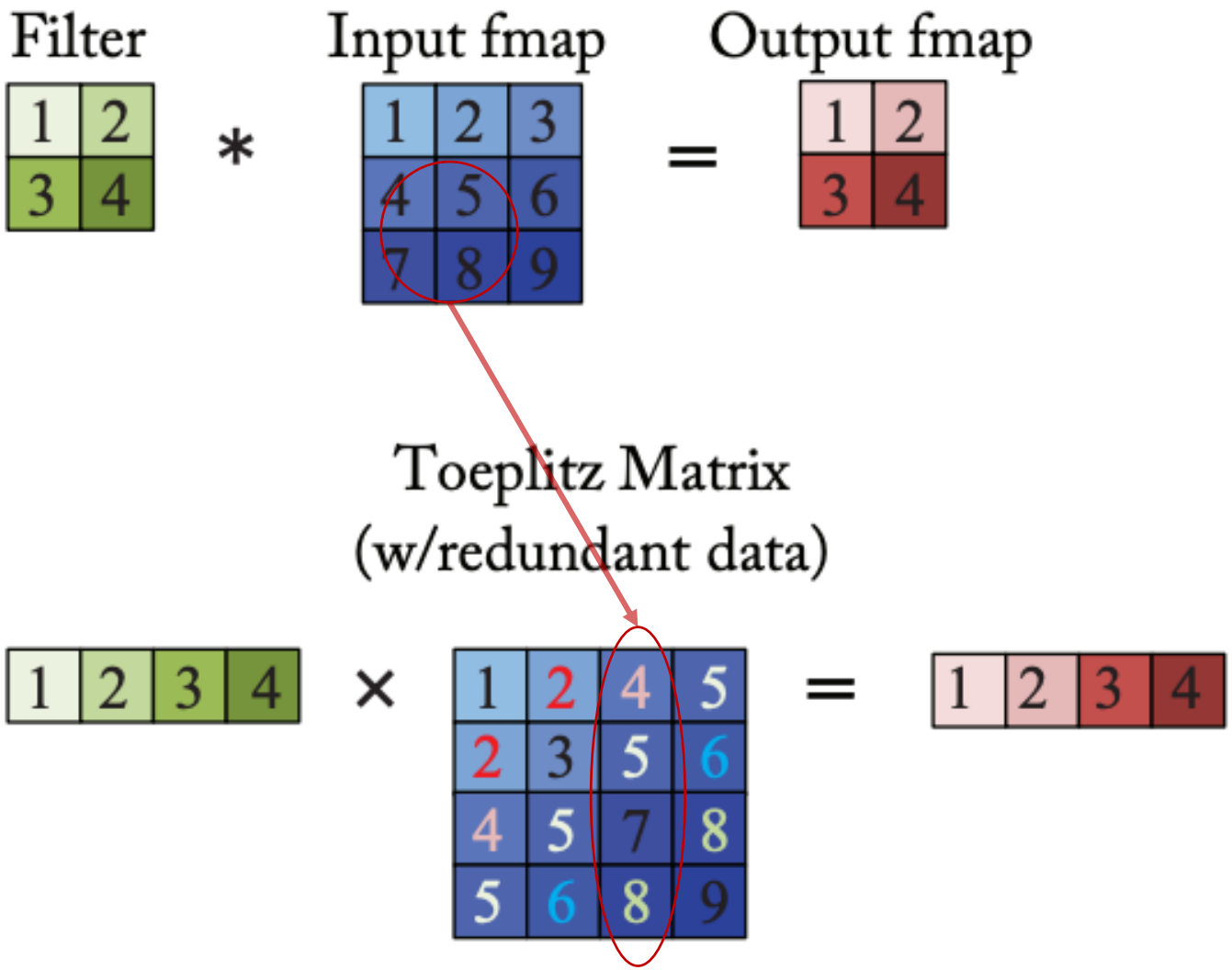
Convolution Layer



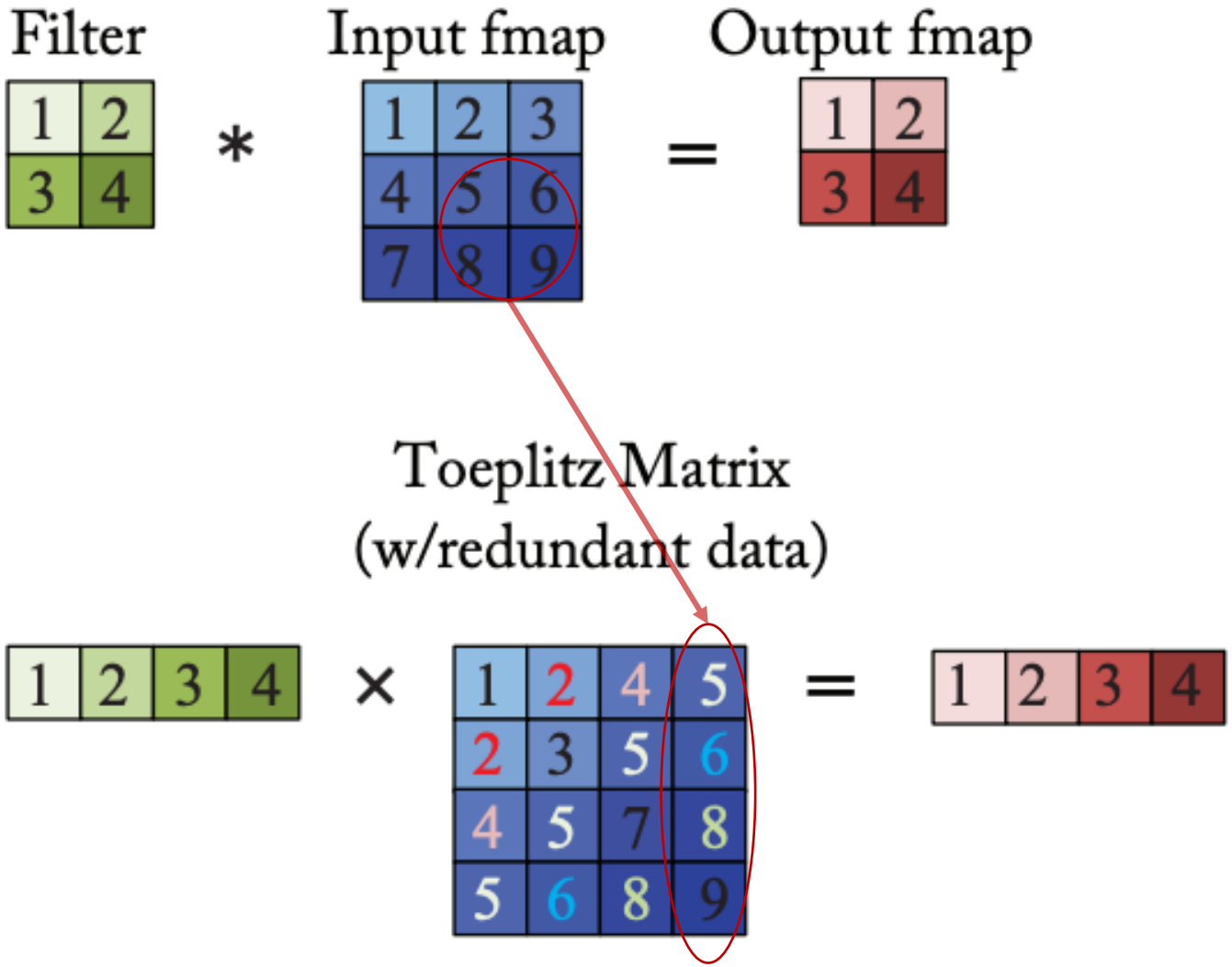
Convolution Layer



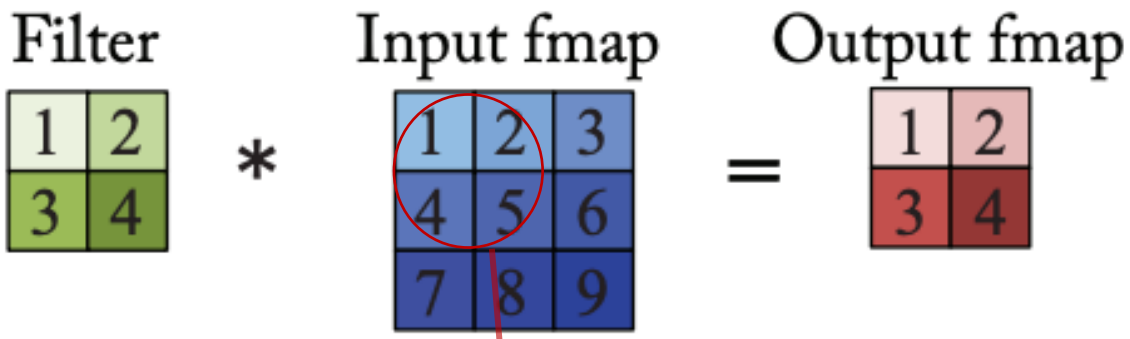
Convolution Layer



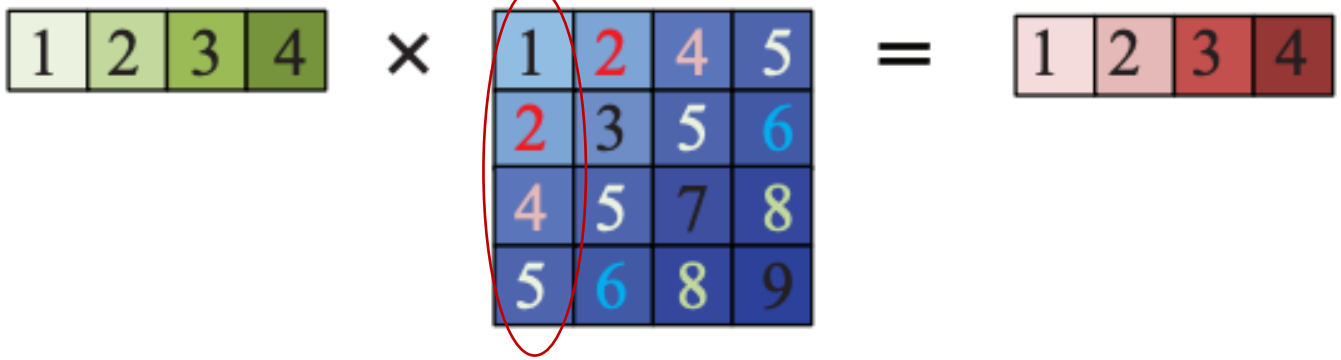
Convolution Layer



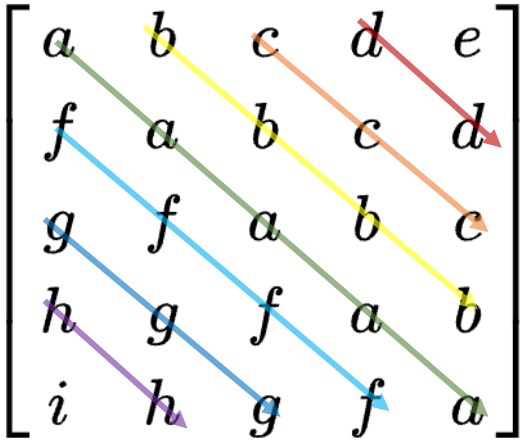
Convolution Layer



Toeplitz Matrix
(w/redundant data)



Note: this method is famously known as img2col

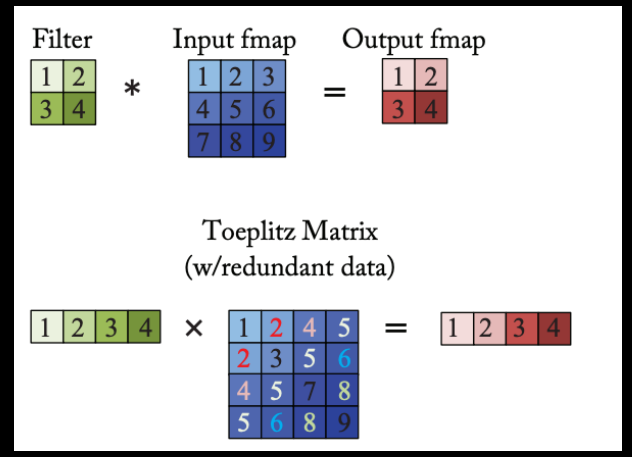


Toeplitz matrix: each descending diagonal from left to right is constant.

Note: The Toeplitz matrix used in CNN is a bit different than the original definition

Img2Col Algorithm

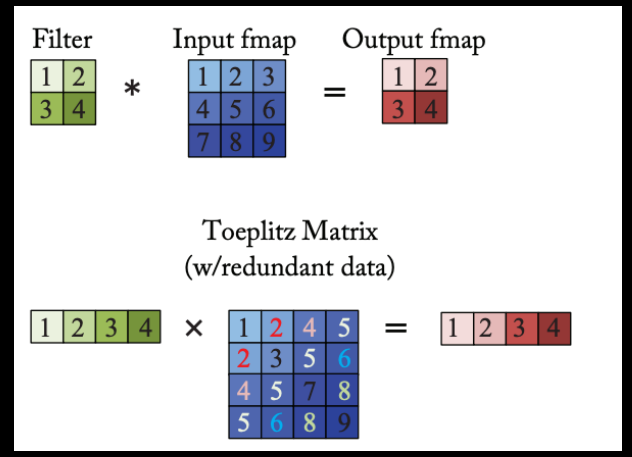
```
def img2col(image, block_size):
    # Get image dimensions
    rows, cols = image.shape
    # Calculate the output dimensions
    output_rows = rows - block_size + 1
    output_cols = cols - block_size + 1

    # Create an empty array to hold the columns
    output = np.zeros((block_size*block_size, output_rows*output_cols))

    # Fill the output array with each block
    col = 0
    for i in range(output_rows):
        for j in range(output_cols):
            block = image[i:i+block_size, j:j+block_size].flatten()
            output[:, col] = block
            col += 1

    return output
```



Runtime Complexity:

Img2Col Algorithm

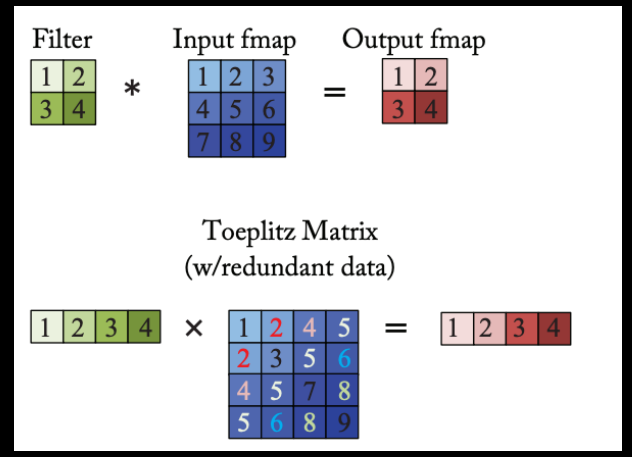
```python
def img2col(image, block_size):
    # Get image dimensions
    rows, cols = image.shape
    # Calculate the output dimensions
    output_rows = rows - block_size + 1
    output_cols = cols - block_size + 1

    # Create an empty array to hold the columns
    output = np.zeros((block_size*block_size, output_rows*output_cols))

    # Fill the output array with each block
    col = 0
    for i in range(output_rows):
        for j in range(output_cols):
            block = image[i:i+block_size, j:j+block_size].flatten()
            output[:, col] = block
            col += 1

    return output
```



Runtime Complexity: $O(block_{size}{}^2 \times rows \times col)$

Img2Col Algorithm

```
def img2col(image, block_size):
    # Get image dimensions
    rows, cols = image.shape
    # Calculate the output dimensions
    output_rows = rows - block_size + 1
    output_cols = cols - block_size + 1

    # Create an empty array to hold the columns
    output = np.zeros((block_size*block_size, output_rows*output_cols))

    # Fill the output array with each block
    col = 0
    for i in range(output_rows):
        for j in range(output_cols):
            block = image[i:i+block_size, j:j+block_size].flatten()
            output[:, col] = block
            col += 1

    return output
```



We can run each of these or every few of them in parallel

Img2Col Algorithm

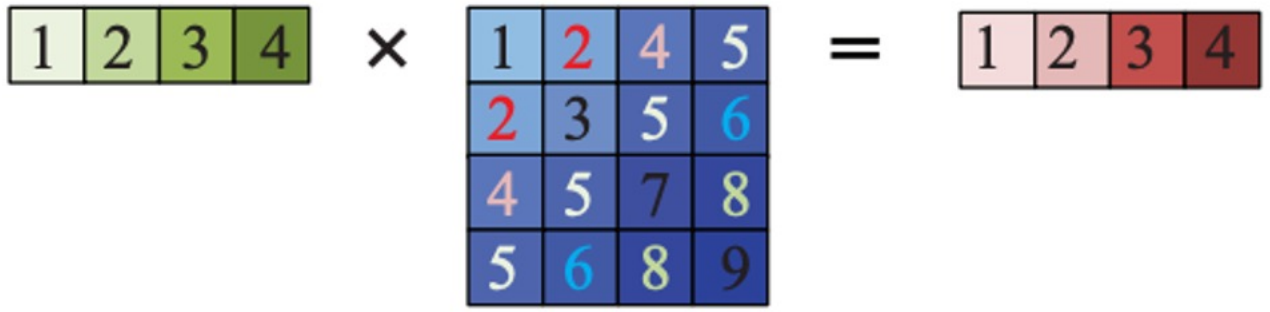
```python
def img2col_segment(image_segment, block_size):
    rows, cols = image_segment.shape
    output_rows = rows - block_size + 1
    output_cols = cols - block_size + 1
    output = np.zeros((block_size * block_size, output_rows * output_cols))

    col = 0
    for i in range(output_rows):
        for j in range(output_cols):
            block = image_segment[i:i+block_size, j:j+block_size].flatten()
            output[:, col] = block
            col += 1
    return output
```

Converts several rows

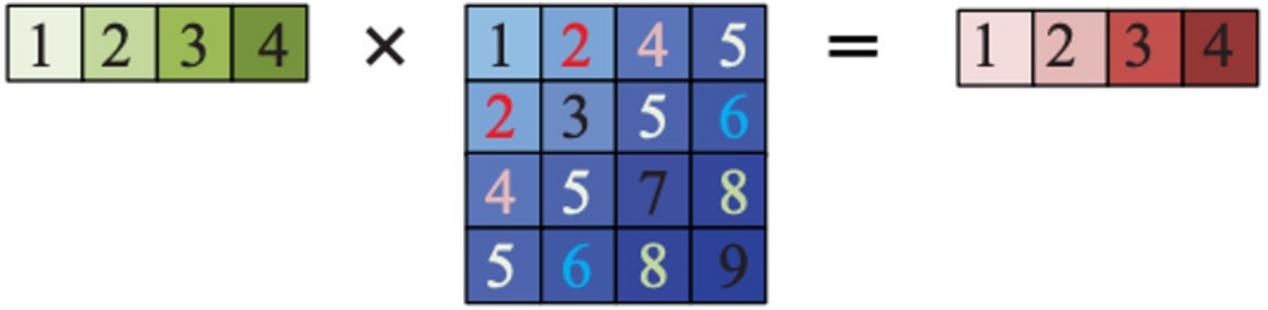
Spawns multiple img2col_segment

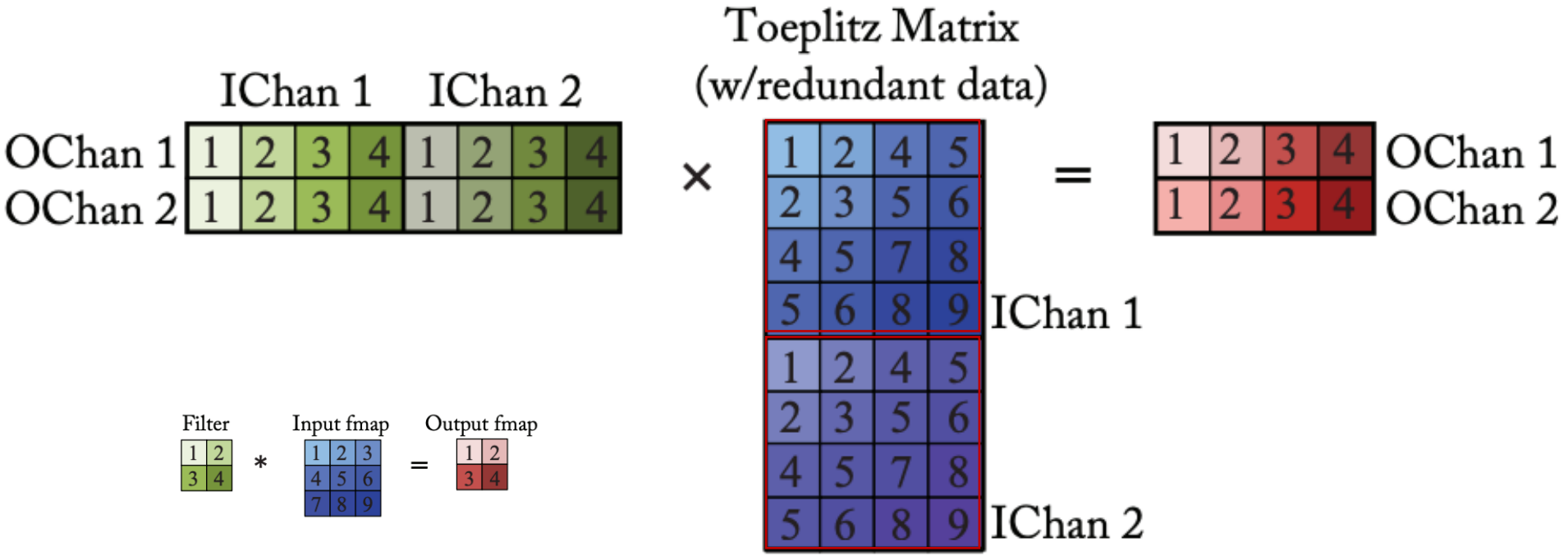
```python
def parallel_img2col(image, block_size, num_processes=None):
    rows, _ = image.shape
    segment_height = rows // num_processes

    # Split the image into horizontal segments
    segments = [image[i:i+segment_height]
    for i in range(0, rows, segment_height)]

    # Create a pool of processes
    with Pool(processes=num_processes) as pool:
        # Apply img2col to each segment in parallel
        results = pool.starmap(
        img2col_segment, [(segment, block_size) for segment in segments])

    # Combine the results from each process
    combined_result = np.hstack(results)
    return combined_result
```

Convolution Layer With Multiple Input/Output Channels

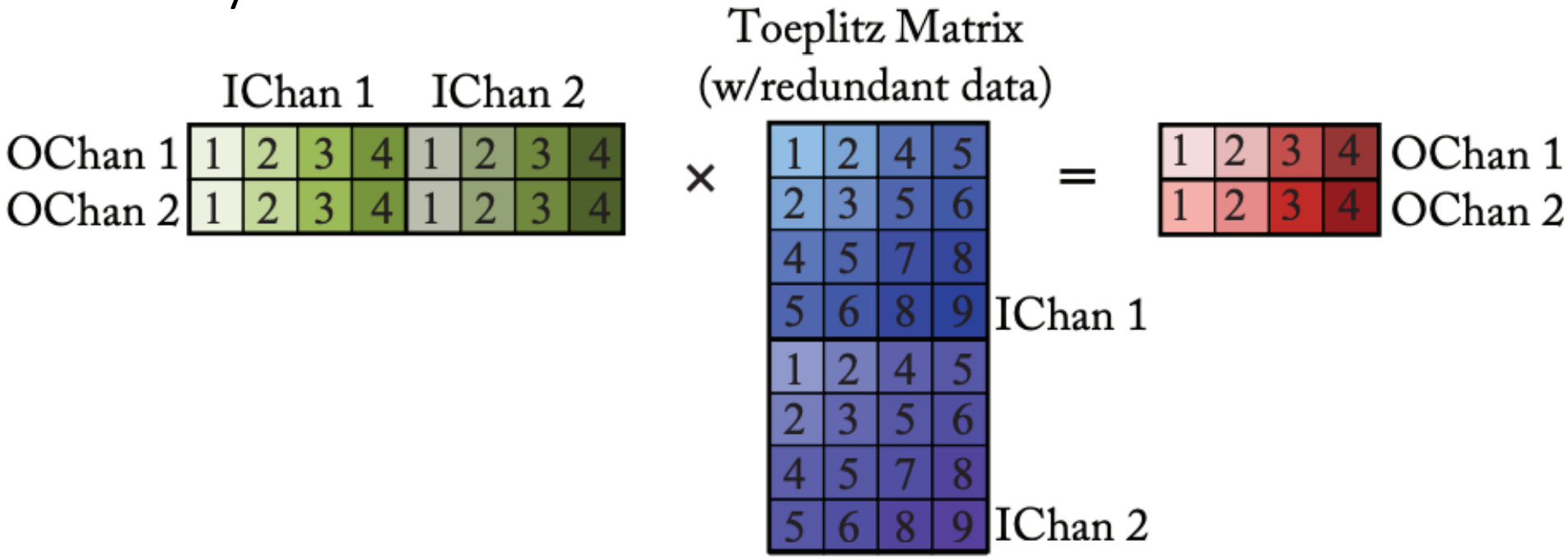


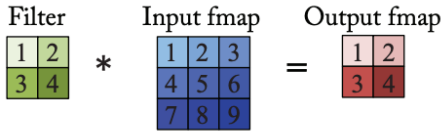
Convolution Layer With Multiple Input/Output Channels



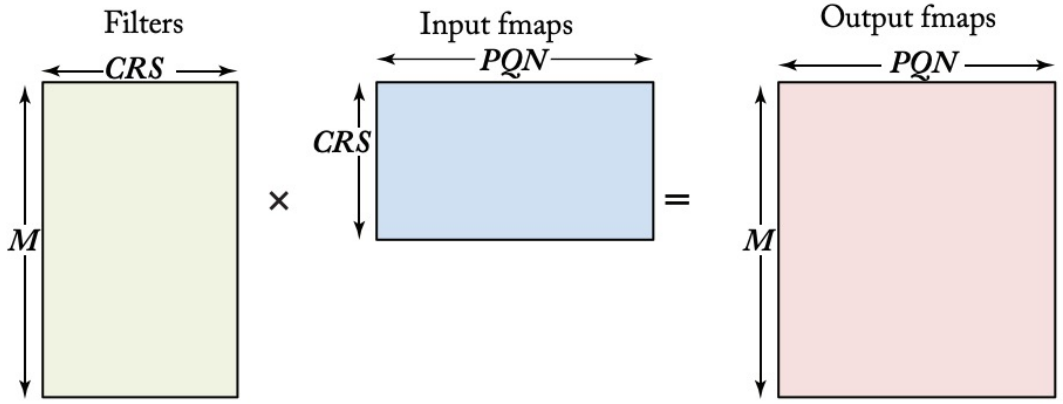


Convolution Layer



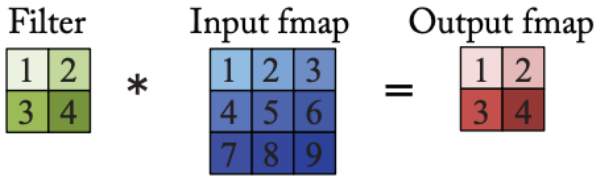


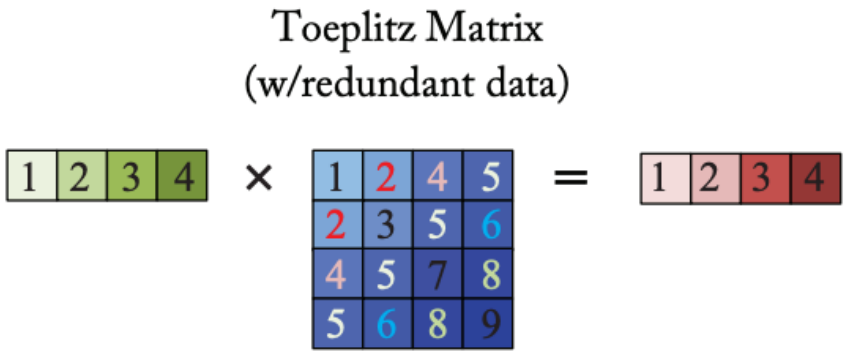
$$O_{n,m,p,q} = B_m + \sum_{c,r,s} I_{n,c,Up+r,Uq+s} \times F_{m,c,r,s}$$



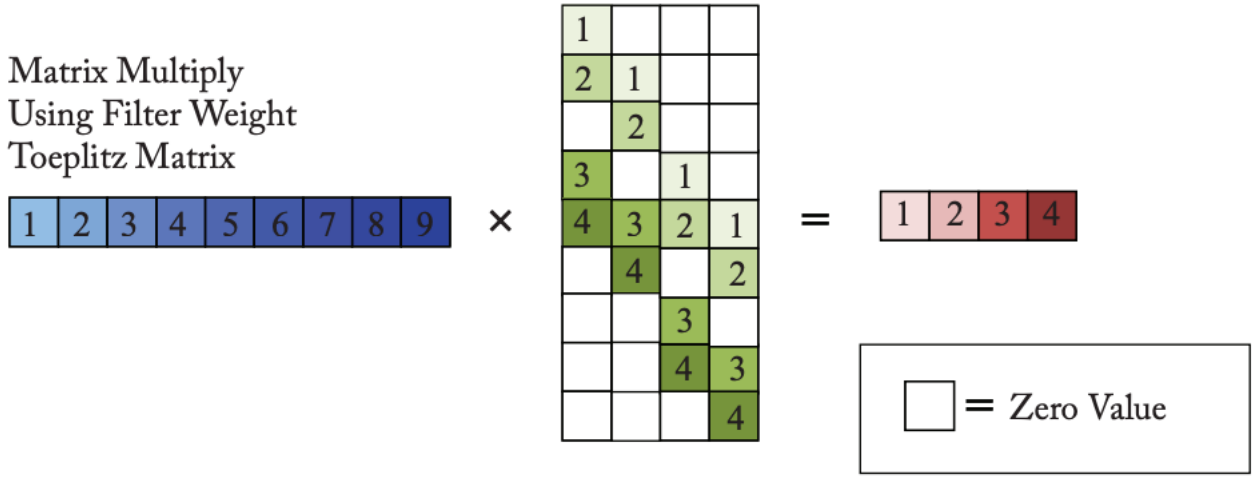
Alternative

$$A * B = B * A$$





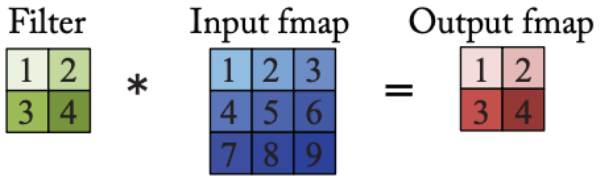
Img2Col



Note: In a **block Toeplitz matrix**, instead of each element being a scalar (like t0, t1, etc.), each is a block (matrix) and the same block is repeated along each diagonal.

Alternative

$$A * B = B * A$$

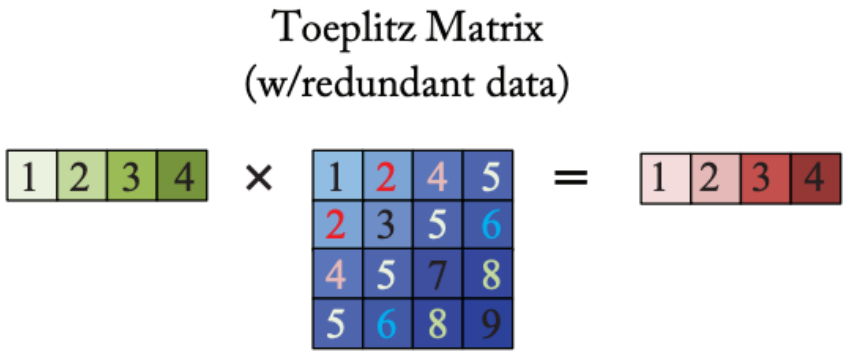


Example (2×2 blocks, 3×3 structure)

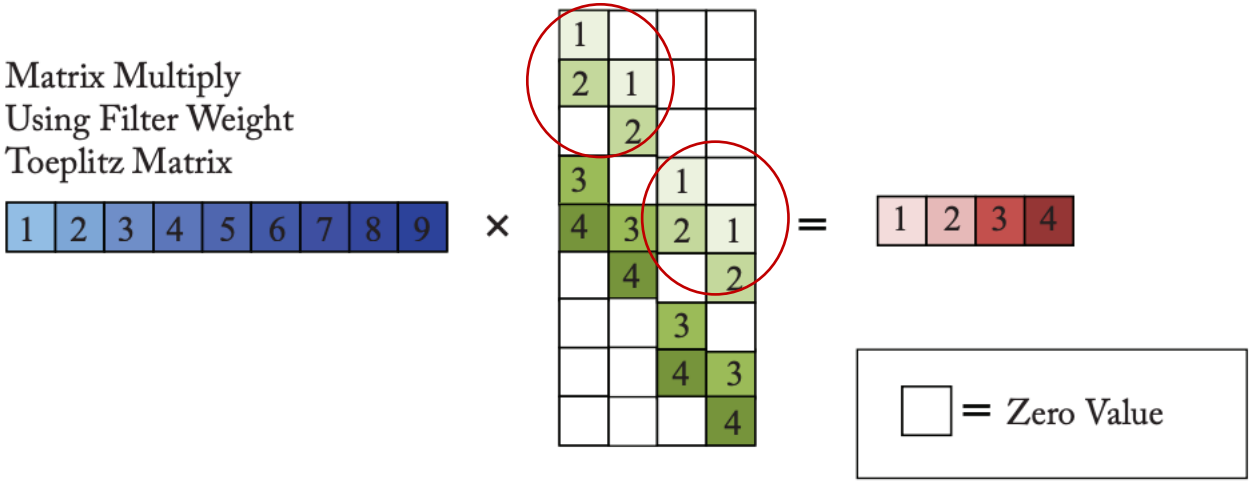
If each block is a 2×2 matrix, then a block Toeplitz matrix might look like:

$$T = \begin{bmatrix} A & B & C \\ D & A & B \\ E & D & A \end{bmatrix}$$

where each A, B, C, D, E are 2×2 matrices.

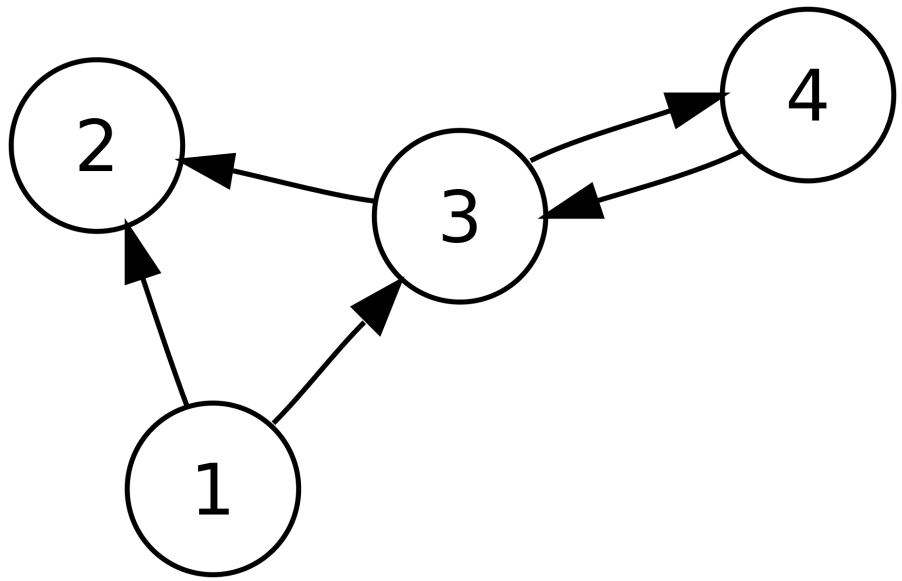


Img2Col



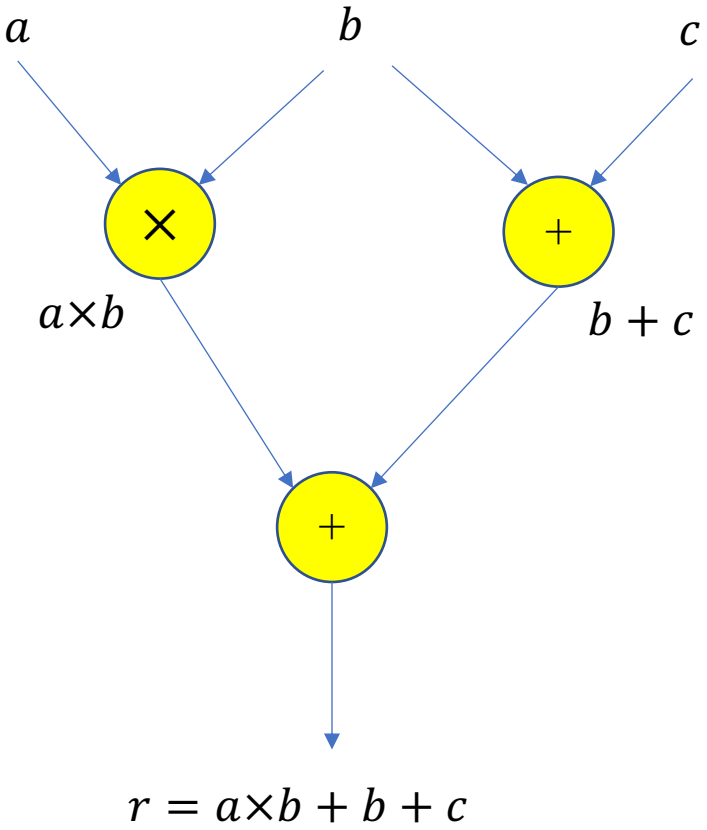
Note: In a **block Toeplitz matrix**, instead of each element being a scalar (like t0, t1, etc.), each is a block (matrix) and the same block is repeated along each diagonal.

Dataflow



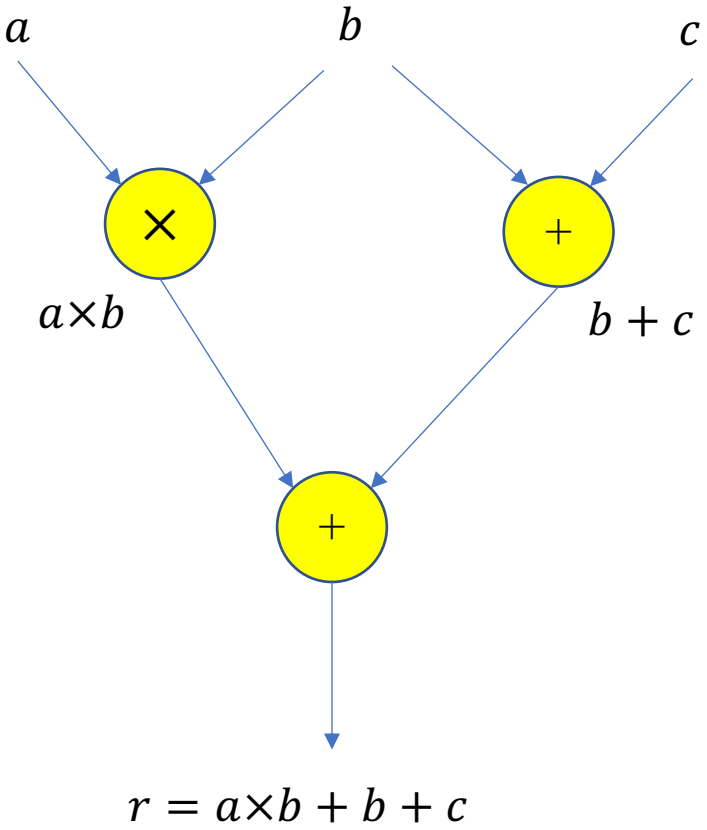
Dataflow

- **Data-Driven Execution**: Nodes operate when input data is ready.



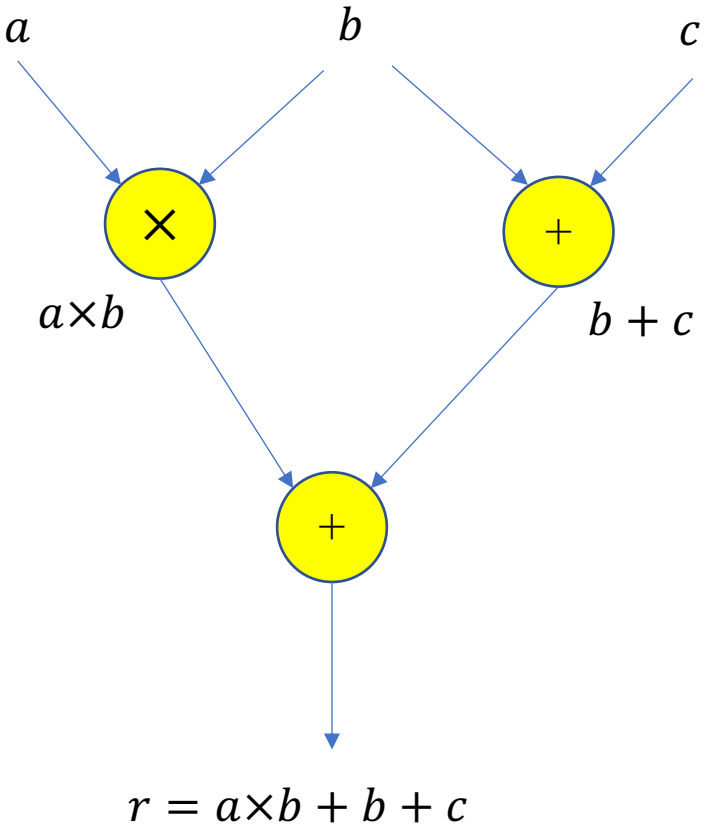
Dataflow

- **Data-Driven Execution**: Nodes operate when input data is ready.
- **Parallelism**: Multiple nodes can execute concurrently if their inputs are available.



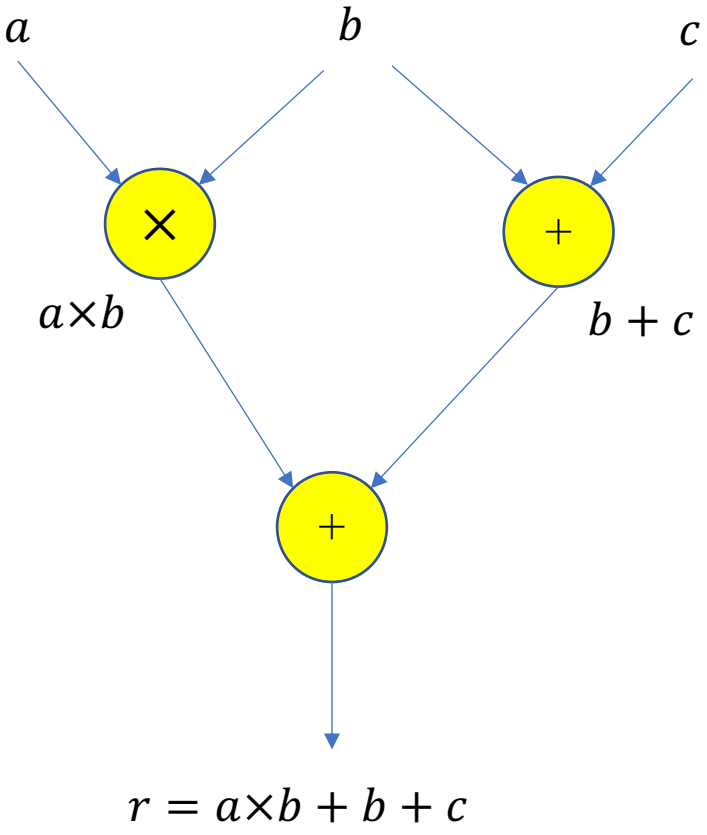
Dataflow

- **Data-Driven Execution**: Nodes operate when input data is ready.
- **Parallelism**: Multiple nodes can execute concurrently if their inputs are available.
- **Statelessness**: Nodes don’t store state information. They purely operate on inputs to produce outputs.



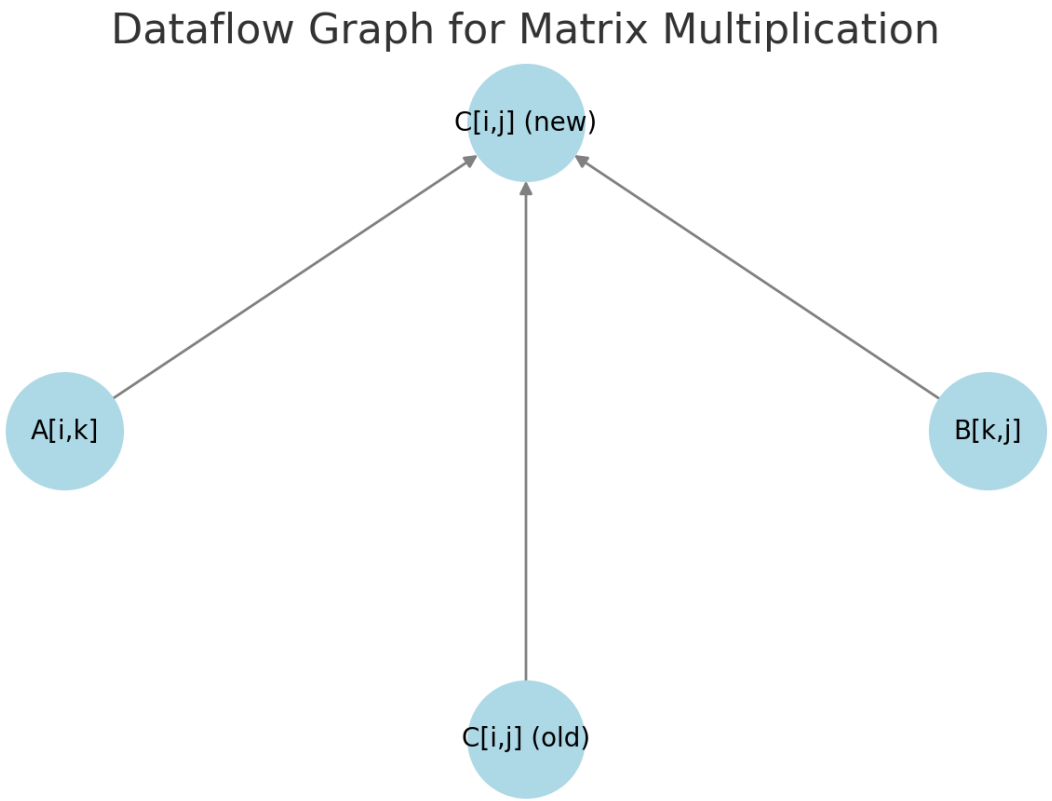
Dataflow

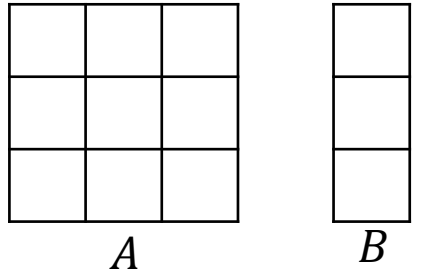
- **Data-Driven Execution**: Nodes operate when input data is ready.
- **Parallelism**: Multiple nodes can execute concurrently if their inputs are available.
- **Statelessness**: Nodes don’t store state information. They purely operate on inputs to produce outputs.
- **Dynamic Scheduling**: Execution order is not predetermined; it's dictated by data availability.



Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

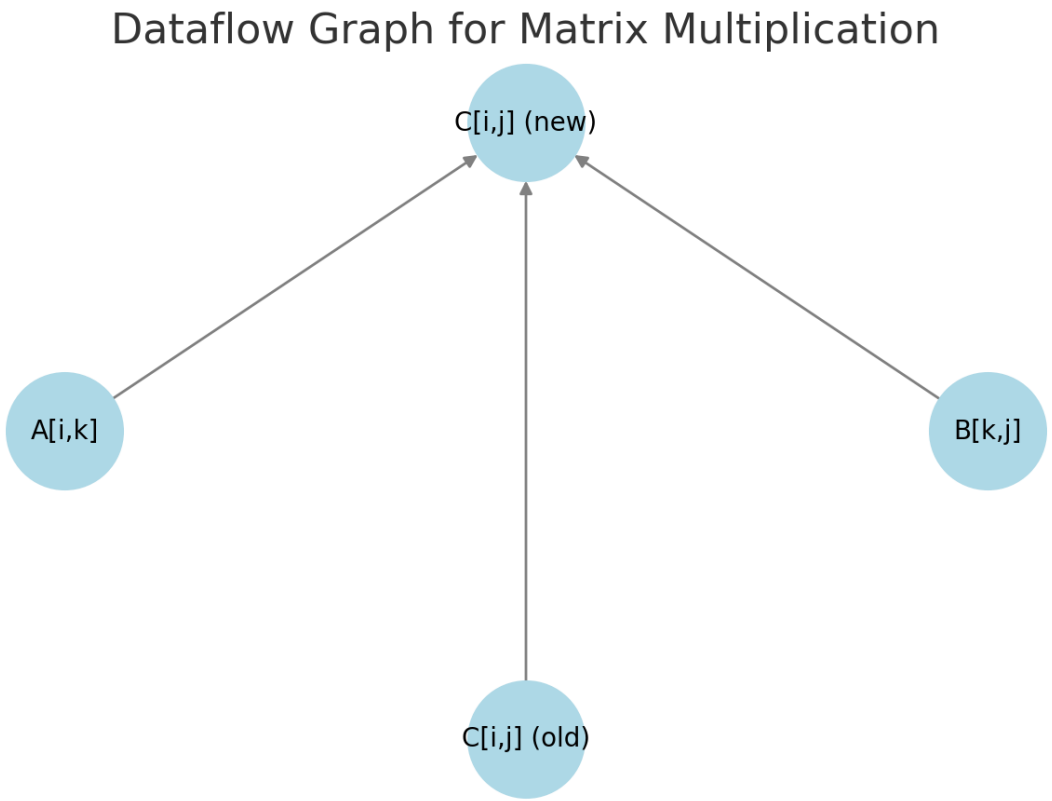


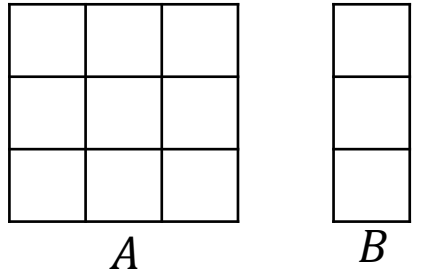



```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



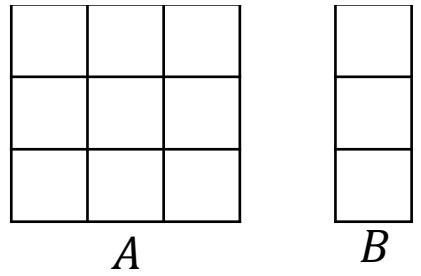


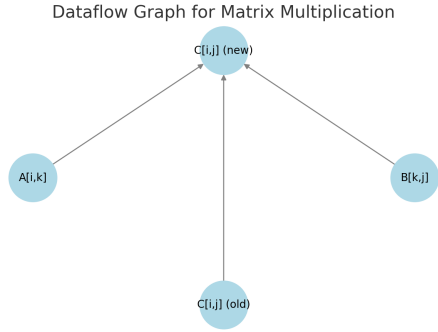

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

- **Why Representing Dataflow with Loop Nests?**
  - **Reordering for Parallelism**: Transform loop to expose parallel tasks, then represent as dataflow

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



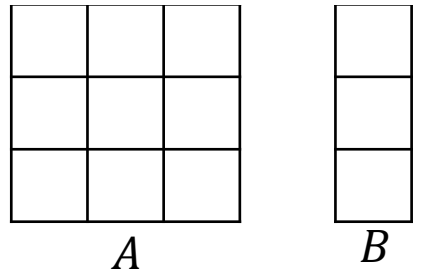


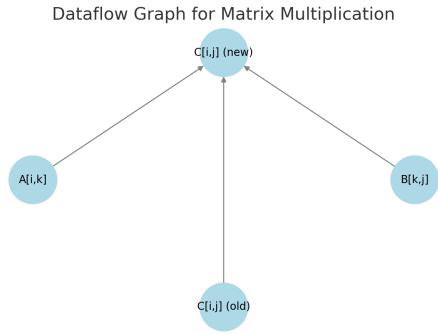

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

- **Why Representing Dataflow with Loop Nests?**
  - **Reordering for Parallelism**: Transform loop to expose parallel tasks, then represent as dataflow nodes.
  - **Mapping to Hardware**: Loop operations → Dataflow nodes for parallel architectures.

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



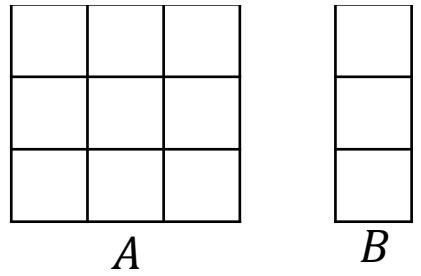


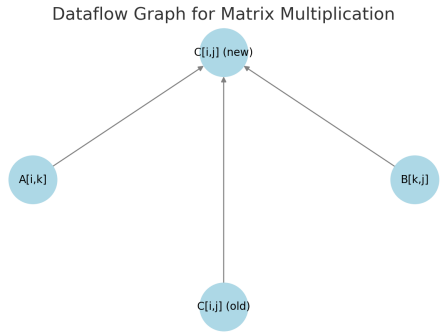

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

- **Why Representing Dataflow with Loop Nests?**
  - **Reordering for Parallelism**: Transform loop to expose parallel tasks, then represent as dataflow nodes.
  - **Mapping to Hardware**: Loop operations → Dataflow nodes for parallel architectures.
  - **Tiling**: Break loops into smaller chunks, map tiles to dataflow model.

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



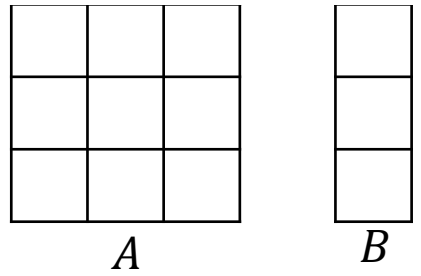


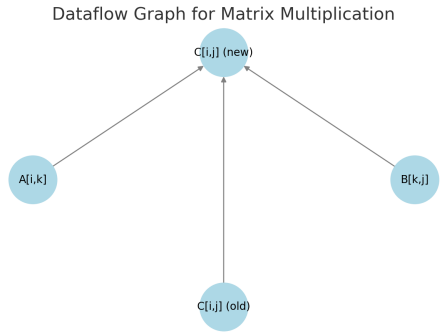

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

- **Why Representing Dataflow with Loop Nests?**
  - **Reordering for Parallelism**: Transform loop to expose parallel tasks, then represent as dataflow nodes.
  - **Mapping to Hardware**: Loop operations → Dataflow nodes for parallel architectures.
  - **Tiling**: Break loops into smaller chunks, map tiles to dataflow model.
  - **Loop Pipelining**: Different loop stages executed simultaneously, akin to dataflow pipelines.
    - *Example*: Different iterations of a loop being at fetch, process, and store stages simultaneously.

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



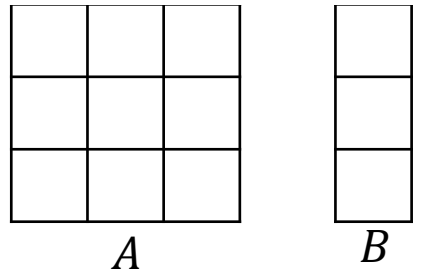


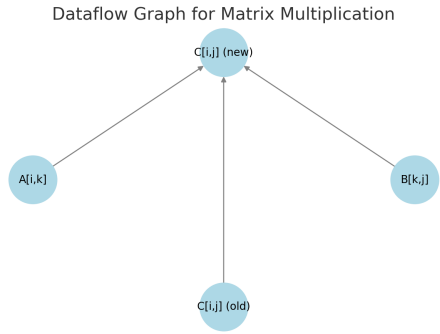

```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

Loop Nest

- **Why Representing Dataflow with Loop Nests?**
  - **Reordering for Parallelism**: Transform loop to expose parallel tasks, then represent as dataflow nodes.
  - **Mapping to Hardware**: Loop operations → Dataflow nodes for parallel architectures.
  - **Tiling**: Break loops into smaller chunks, map tiles to dataflow model.
  - **Loop Pipelining**: Different loop stages executed simultaneously, akin to dataflow pipelines.
    - *Example*: Different iterations of a loop being at fetch, process, and store stages simultaneously.
  - **Dependencies**: It still shows data dependency
    - *Example*: matrix cell (i, j) relying on cell (i-1, j).

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

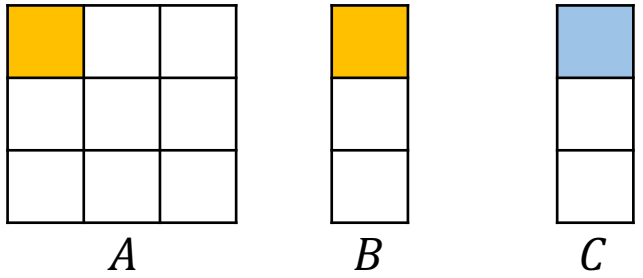





```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    C[i] += A[i][j] * B[j];
  }
}
```

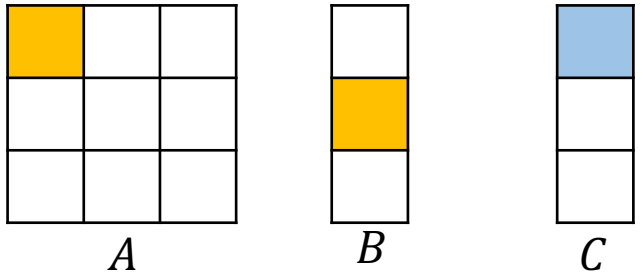
Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



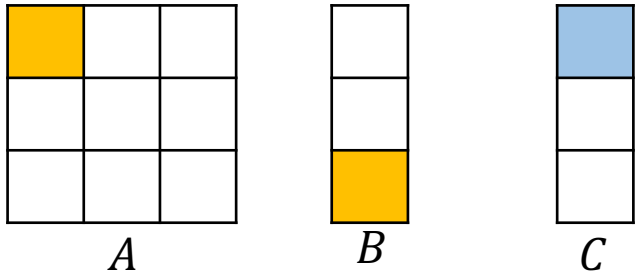
Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



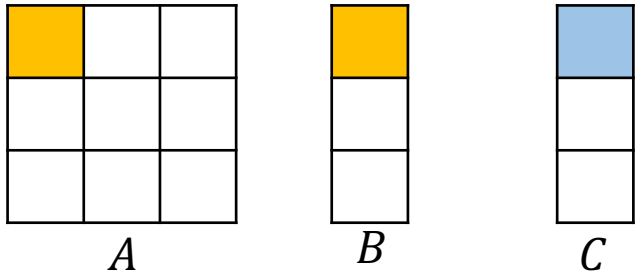
Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



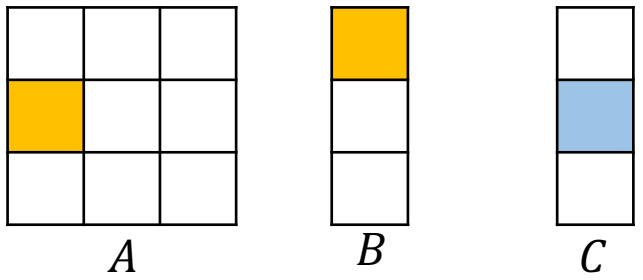
Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



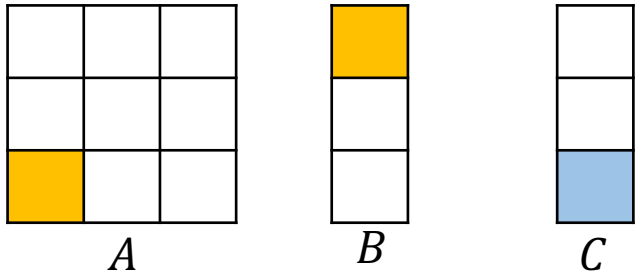
Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



Loop Nest

$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$



Loop Nest in Convolution

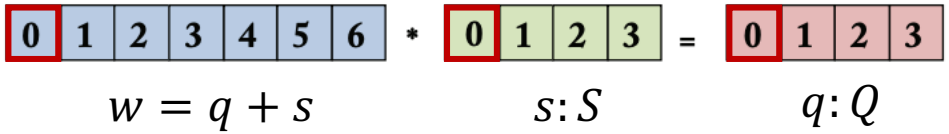
$$O_q = \sum_s I_{q+s} \times F_s$$

$w = q + s$ $s : S$ $q : Q$

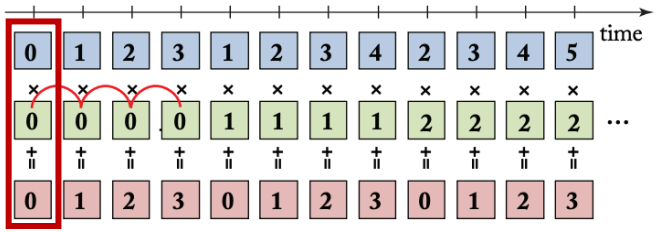
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

Loop Nest in Convolution – Weight Stationary

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



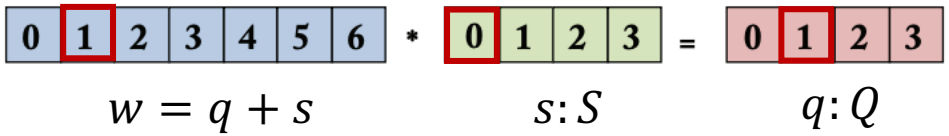

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



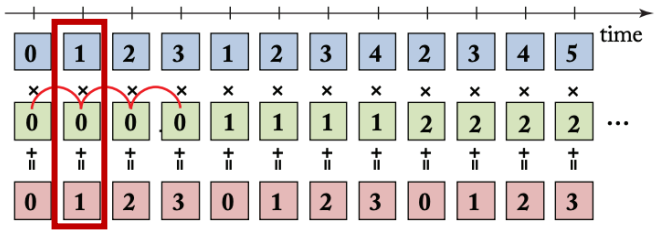
Stationary

Loop Nest in Convolution – Weight Stationary

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



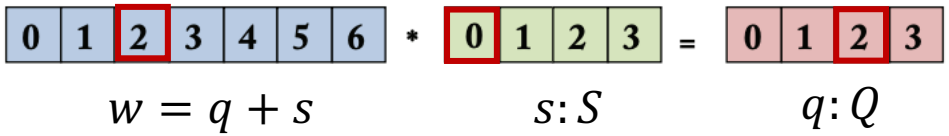

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



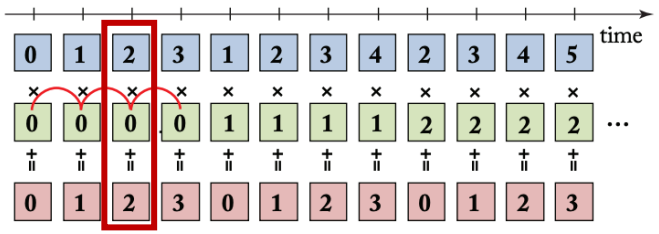
Stationary

Loop Nest in Convolution – Weight Stationary

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



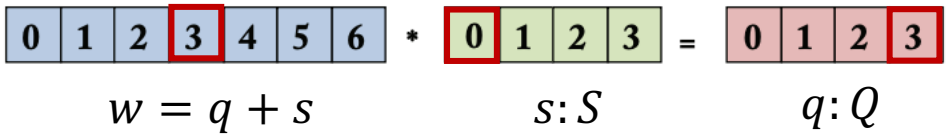

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



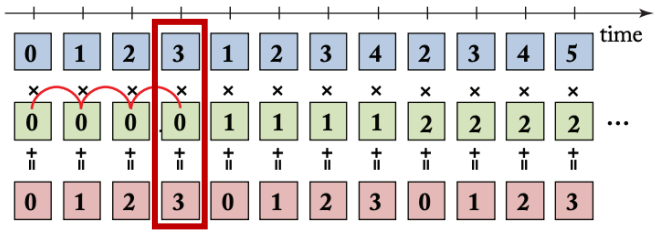
Stationary

Loop Nest in Convolution – Weight Stationary

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




```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



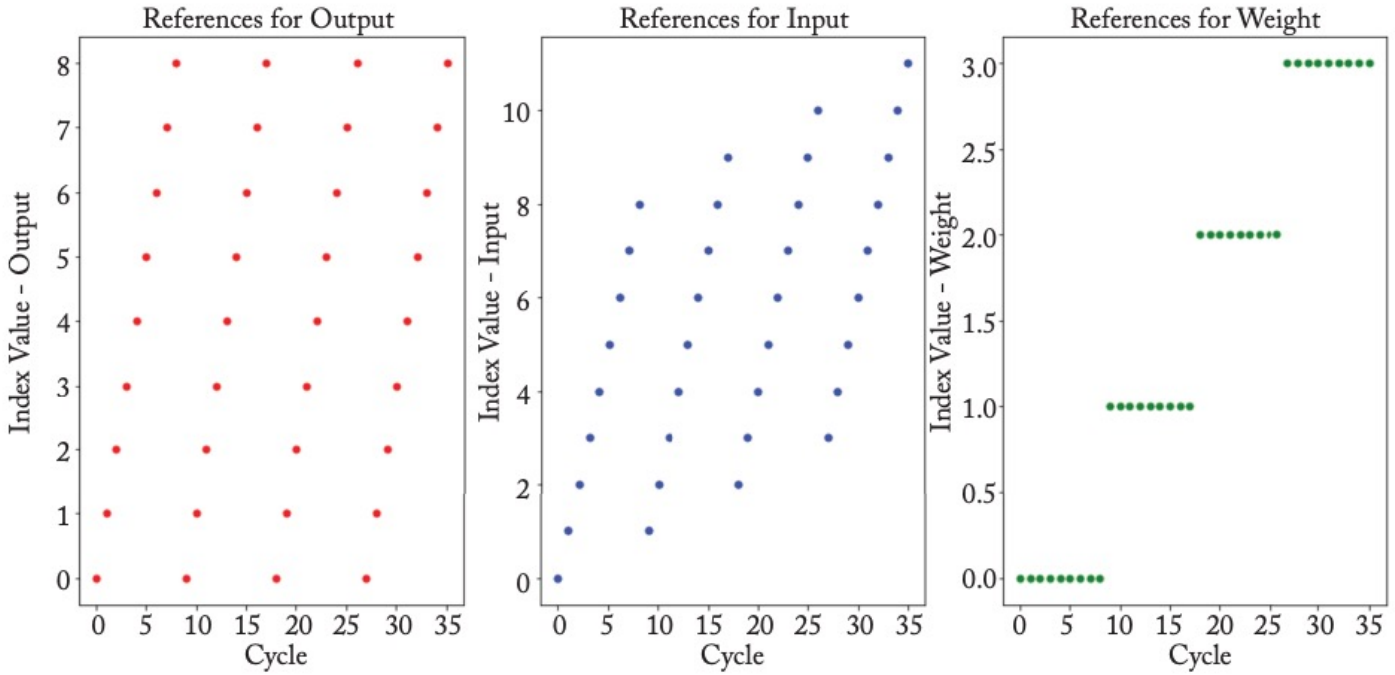
Stationary

Loop Nest in Convolution – Weight Stationary

What is the reuse distance?

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

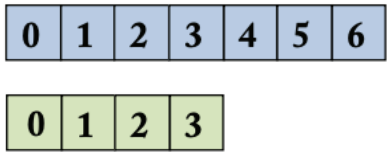
Stationary



Weight Stationary

Pros	Cons
Reduces memory traffic by keeping weights within the processing element.	Not suitable for large models where weights cannot be held in cache.
Power efficient due to less data movement.	Potential underutilization of processing elements due to data not being available on time.
Benefits parallelization at the weight matrix level.	

Tiled Weight Stationary



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

$w = q + s$ $\quad s : S$ $\quad q : Q$

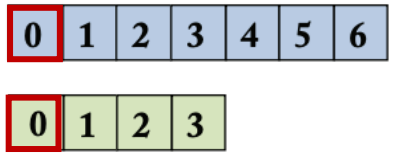
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

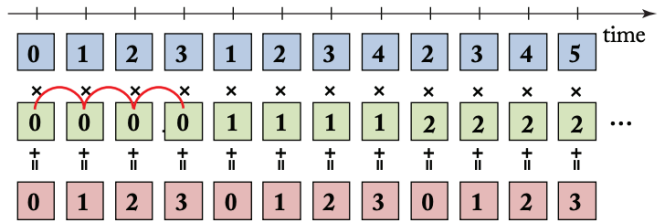
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

q1	s	q0	q	w
0	0	0	0	0

Tiled Weight Stationary



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



WS, no tiling

$w = q + s$ $\quad s : S$ $\quad q : Q$

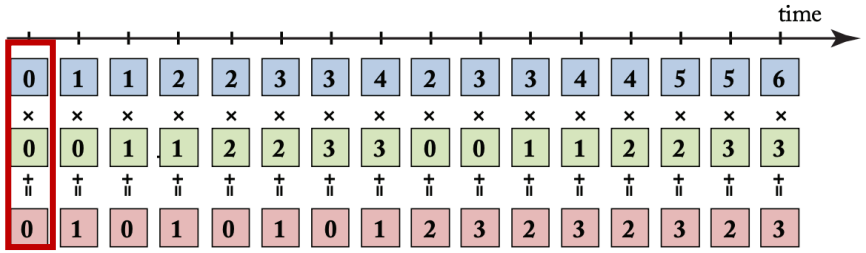
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

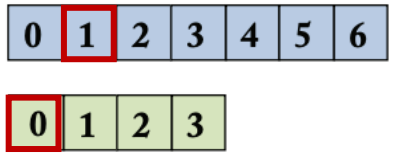
$$q = q_1 Q_0 + q_0$$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

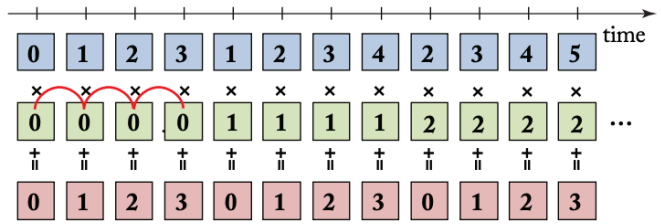
q1	s	q0	q	w
0	0	0	0	0



Tiled Weight Stationary



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



WS, no tiling

$w = q + s$ $\quad s : S$ $\quad q : Q$

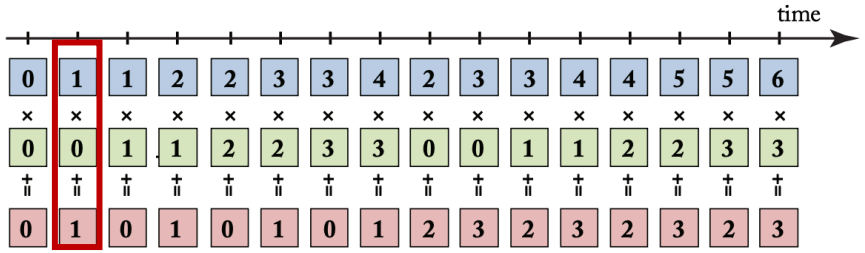
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

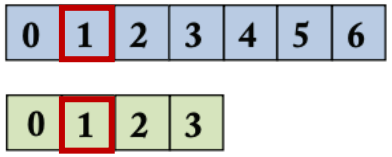
$$q = q_1 Q_0 + q_0$$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

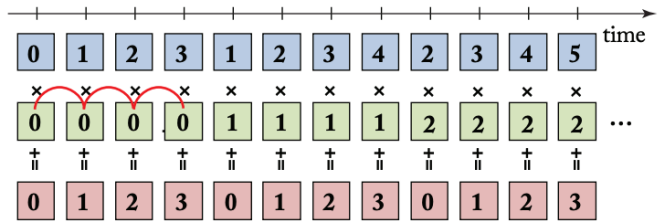
q1	s	q0	q	w
0	0	1	1	1



Tiled Weight Stationary



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



WS, no tiling

$w = q + s$ $\quad s : S$ $\quad q : Q$

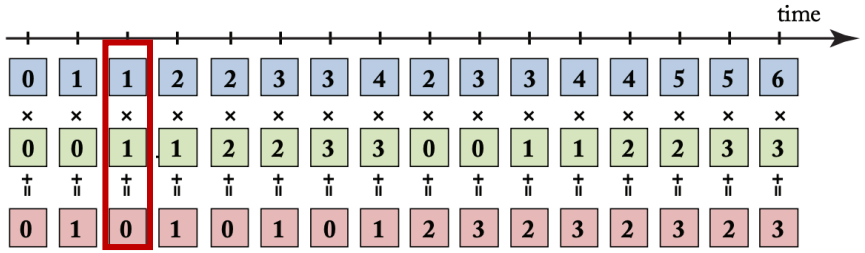
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

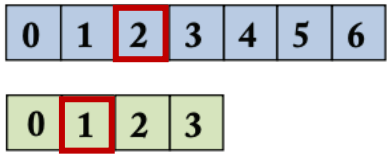
$$q = q_1 Q_0 + q_0$$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

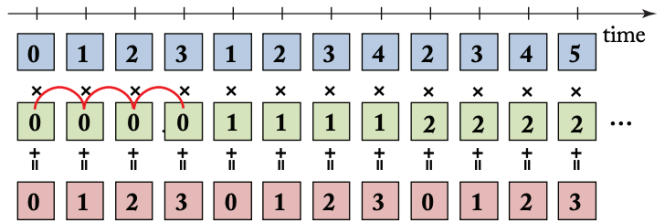
q1	s	q0	q	w
0	1	0	0	1



Tiled Weight Stationary



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



WS, no tiling

$w = q + s$ $\quad s : S$ $\quad q : Q$

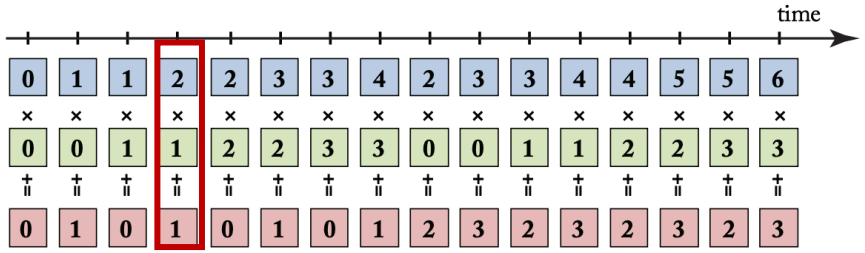
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

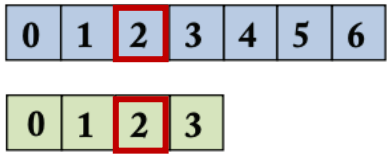
$$q = q_1 Q_0 + q_0$$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

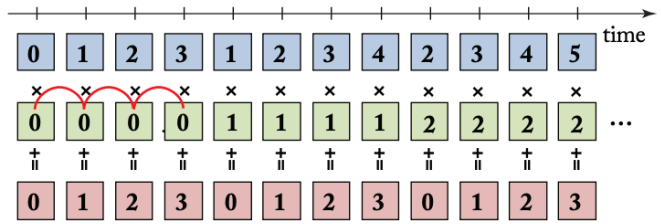
q1	s	q0	q	w
0	1	1	1	2



Tiled Weight Stationary



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



WS, no tiling

$w = q + s$ $\quad s : S$ $\quad q : Q$

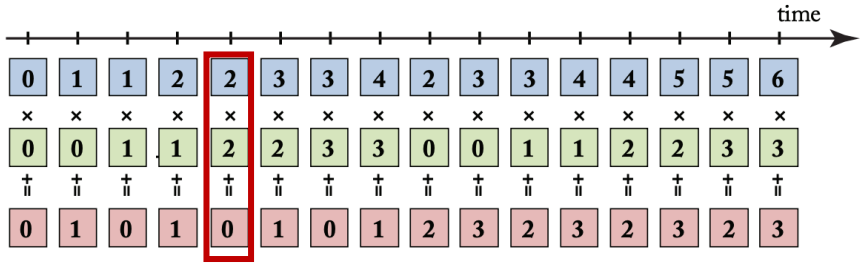
Tile size Number of Tiles

$$Q_0 = Q_1 = \frac{s}{2} = 2$$

$$q = q_1 Q_0 + q_0$$

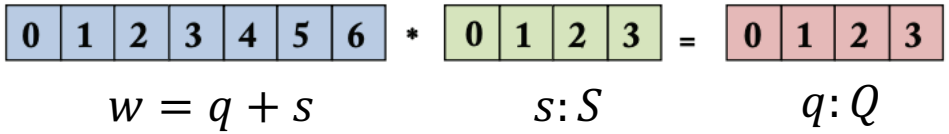
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for q1 in range(Q1):
  for s in range(S):
    for q0 in range(Q0):
      q = q1*Q0 + q0
      w = q+s
      o[q] += i[w] * f[s]
```

q1	s	q0	q	w
0	2	0	0	2



Loop Nest in Convolution – Output Stationary

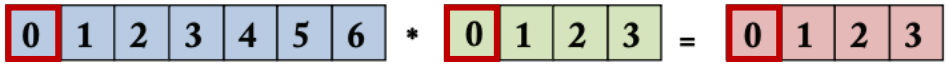
$$O_q = \sum_s I_{q+s} \times F_s$$




```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

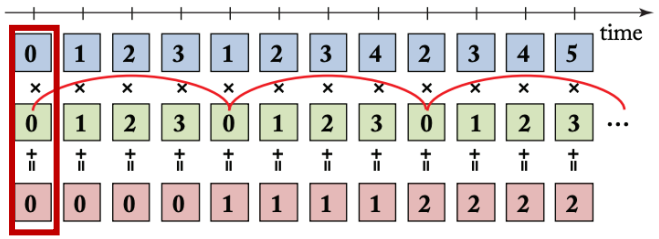
Loop Nest in Convolution – Output Stationary

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



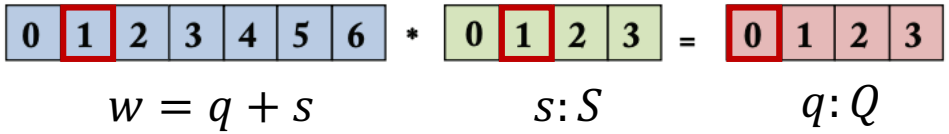
$w = q + s$ $s : S$ $q : Q$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

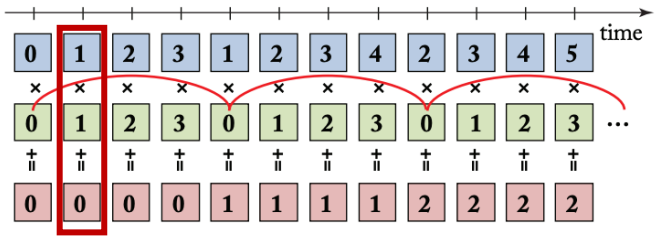


Loop Nest in Convolution – Output Stationary

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

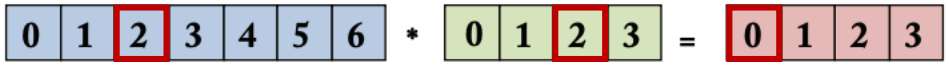



```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

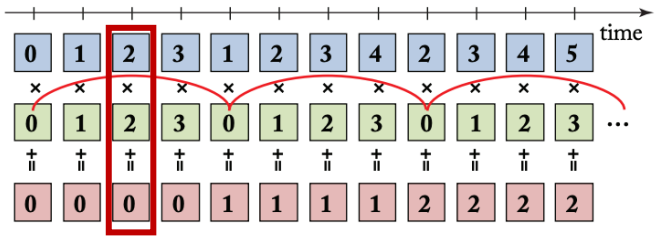


Loop Nest in Convolution – Output Stationary

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

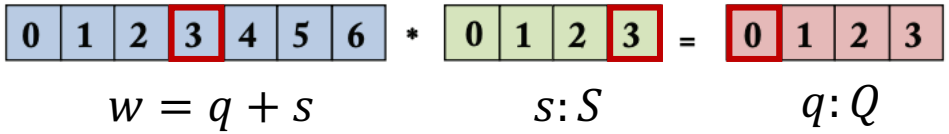



```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

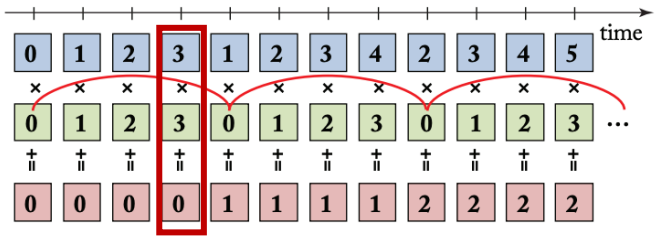


Loop Nest in Convolution – Output Stationary

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



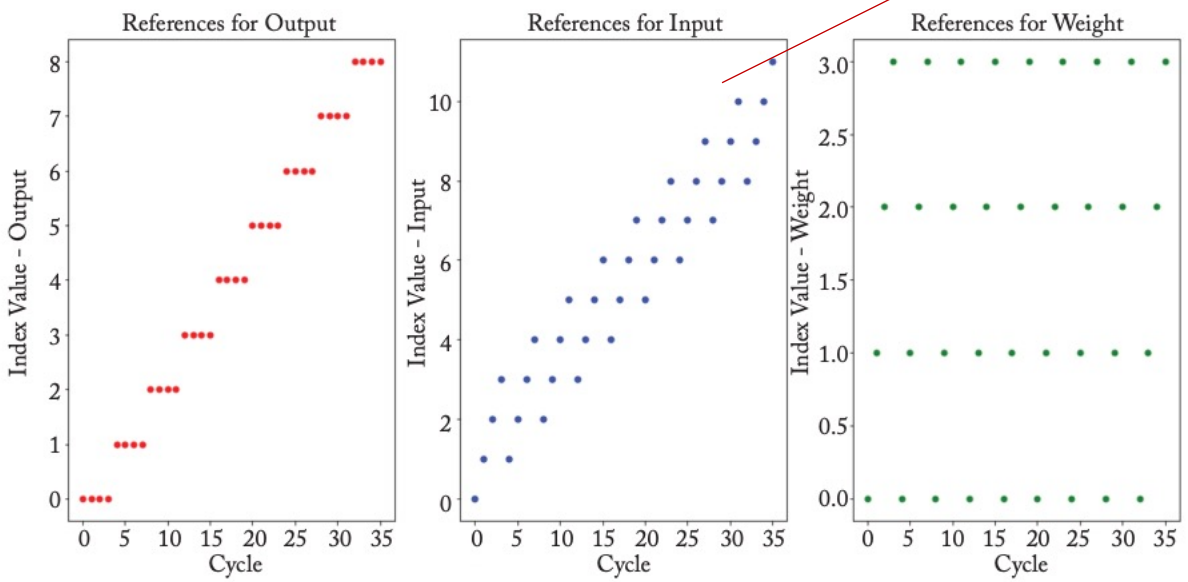

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

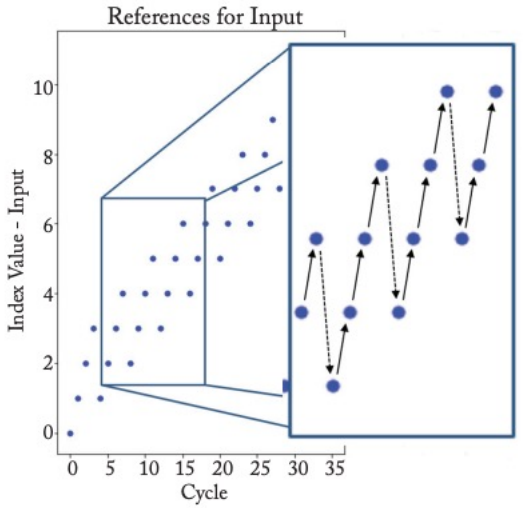


Loop Nest in Convolution – Output Stationary

What is the reuse distance?

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



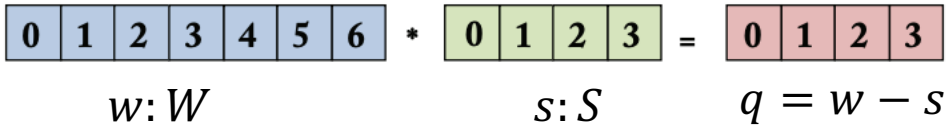


Output Stationary

Pros	Cons
Each processing element computes one element of the output feature map, so **partial sums can be accumulated locally without needing to be stored back to memory**, which reduces memory bandwidth requirements.	Can require significant data movement if the input feature maps are large, leading to high power consumption.
Ideal for architectures that emphasize throughput of generating output feature maps.	May lead to inefficiencies if the output size is small compared to the input size and filter size, as the same inputs are read multiple times.

Loop Nest in Convolution – Input Stationary

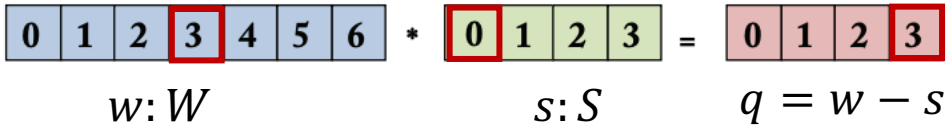
$$O_q = \sum_s I_{q+s} \times F_s$$




```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

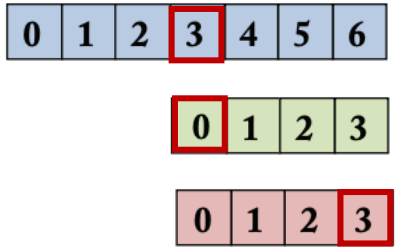
Loop Nest in Convolution – Input Stationary

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



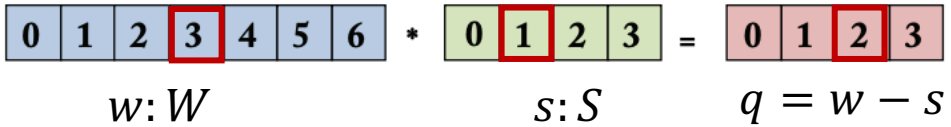

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

w	s	q
3	0	0



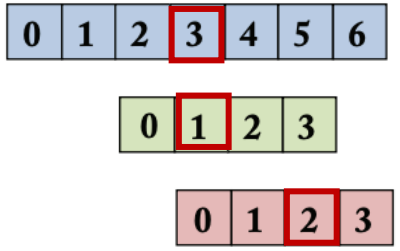
Loop Nest in Convolution – Input Stationary

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



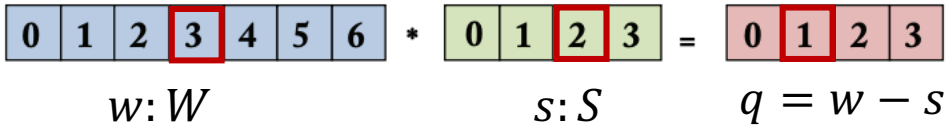

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

w	s	q
3	1	2



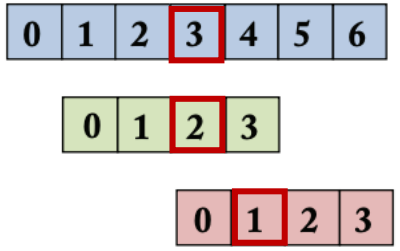
Loop Nest in Convolution – Input Stationary

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



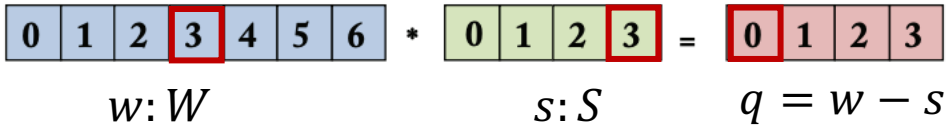

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

w	s	q
3	2	1



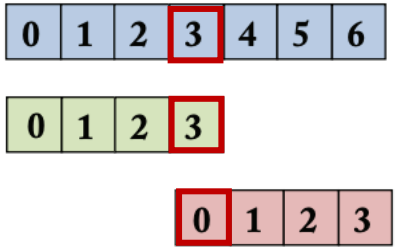
Loop Nest in Convolution – Input Stationary

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




```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

w	s	q
3	3	0



Input Stationary

Pros	Cons
Keeps **input data in the PE**, which can be efficient if the same input is used to compute multiple outputs, as is common in CNNs with multiple filters.	May require a complex control mechanism to ensure that all necessary weights are available at the right time for the computation.
Reduces the bandwidth requirement for input feature maps.	Can lead to inefficiencies when filter weights are unavailable, resulting in idle processing elements.
Highly efficient for convolutional layers with a small number of input channels and many output channels.	

Weight Stationary – Parallel Processing

$$O_q = \sum_s I_{q+s} \times F_s \qquad S_0 = S_1 = \frac{S}{2} = 2$$

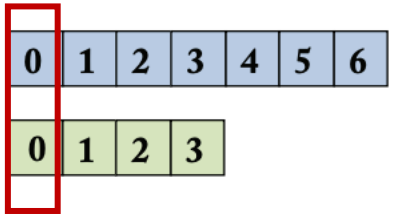
$w = q + s$ $\quad s : S$ $\quad q : Q$

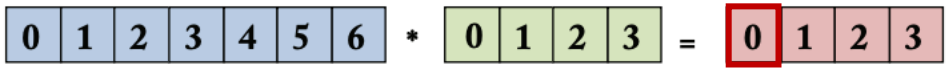
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```

Weight Stationary – Parallel Processing

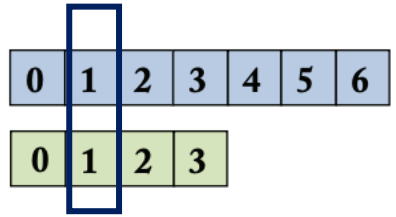
$$O_q = \sum_s I_{q+s} \times F_s$$

$$S_0 = S_1 = \frac{s}{2} = 2$$





$w = q + s$ $\quad s : S$ $\quad q : Q$



Temporal

Spatial

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```

$s_0 = 0$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

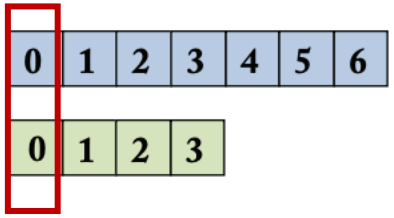
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

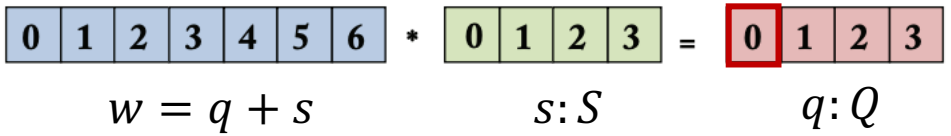
Weight Stationary – Parallel Processing

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

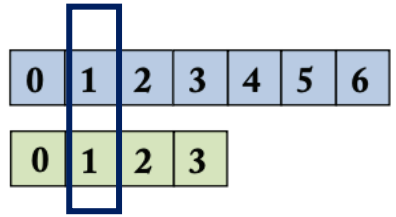
$$S_0 = S_1 = \frac{S}{2} = 2$$



$s_0 = 0$



$w = q + s$ $\quad s : S$ $\quad q : Q$



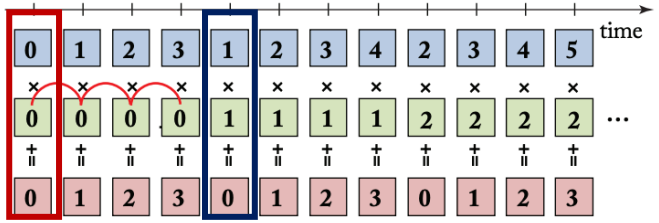
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
0	0	**0**	0

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

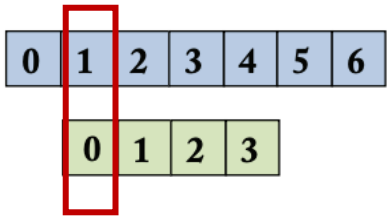
s1	q	s	w
0	0	**1**	1



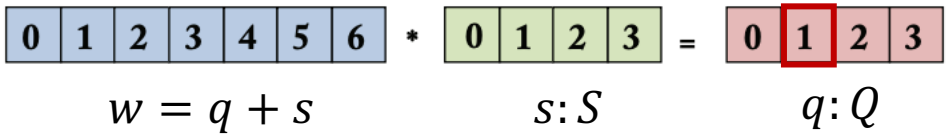
Weight Stationary – Parallel Processing

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

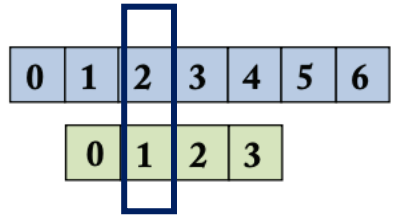
$$S_0 = S_1 = \frac{s}{2} = 2$$



$s_0 = 0$



$w = q + s$ $\quad s : S$ $\quad q : Q$



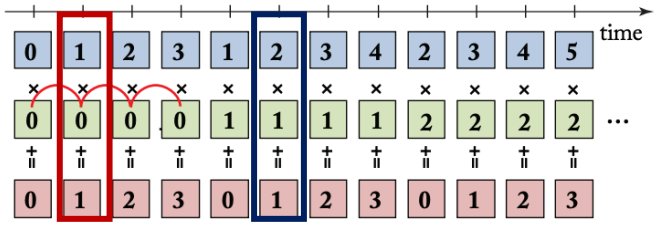
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
0	1	0	1

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

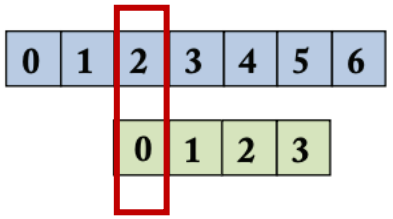
s1	q	s	w
0	1	1	2



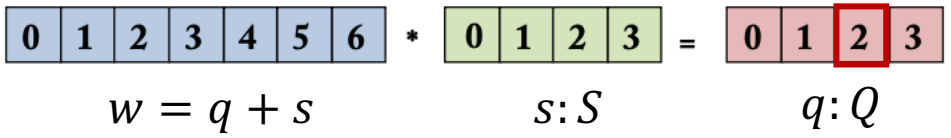
Weight Stationary – Parallel Processing

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

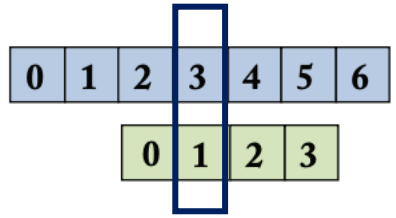
$$S_0 = S_1 = \frac{s}{2} = 2$$



$s_0 = 0$



$w = q + s$ $\quad s : S$ $\quad q : Q$



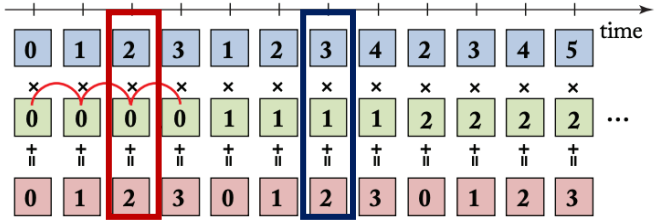
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
0	2	0	2

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

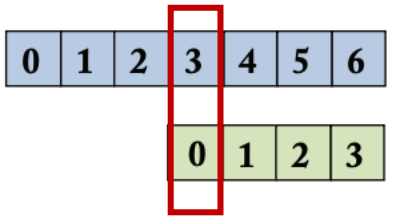
s1	q	s	w
0	2	1	3



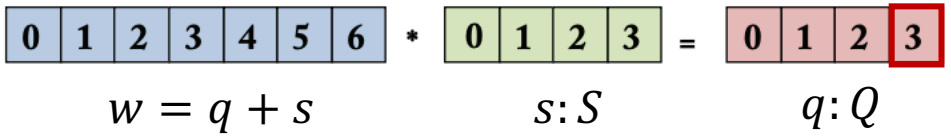
Weight Stationary – Parallel Processing

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

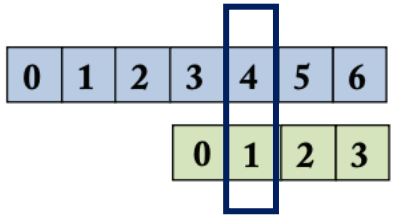
$$S_0 = S_1 = \frac{s}{2} = 2$$



$s_0 = 0$



$w = q + s$ $\quad s : S$ $\quad q : Q$



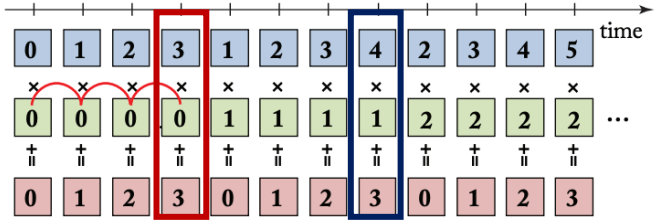
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
0	3	0	3

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

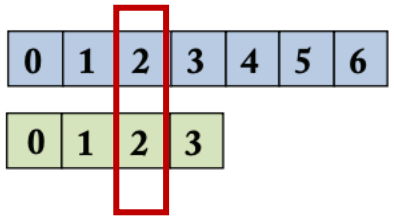
s1	q	s	w
0	3	1	4



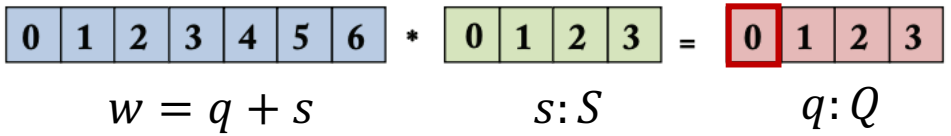
Weight Stationary – Parallel Processing

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

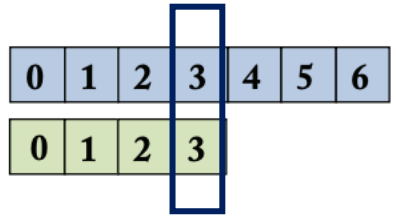
$$S_0 = S_1 = \frac{s}{2} = 2$$



$s_0 = 0$



$w = q + s$ $\quad s : S$ $\quad q : Q$



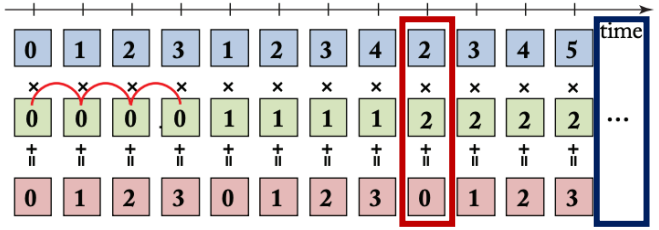
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
1	0	2	2

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

s1	q	s	w
1	0	3	3

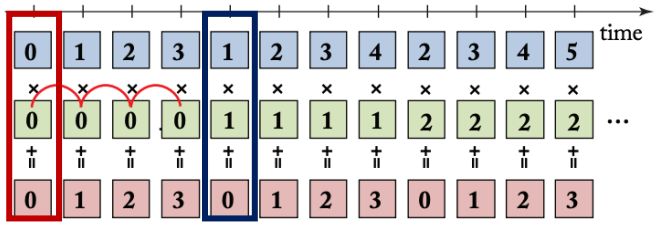


Weight Stationary – Parallel Processing

$s_0 = 0$

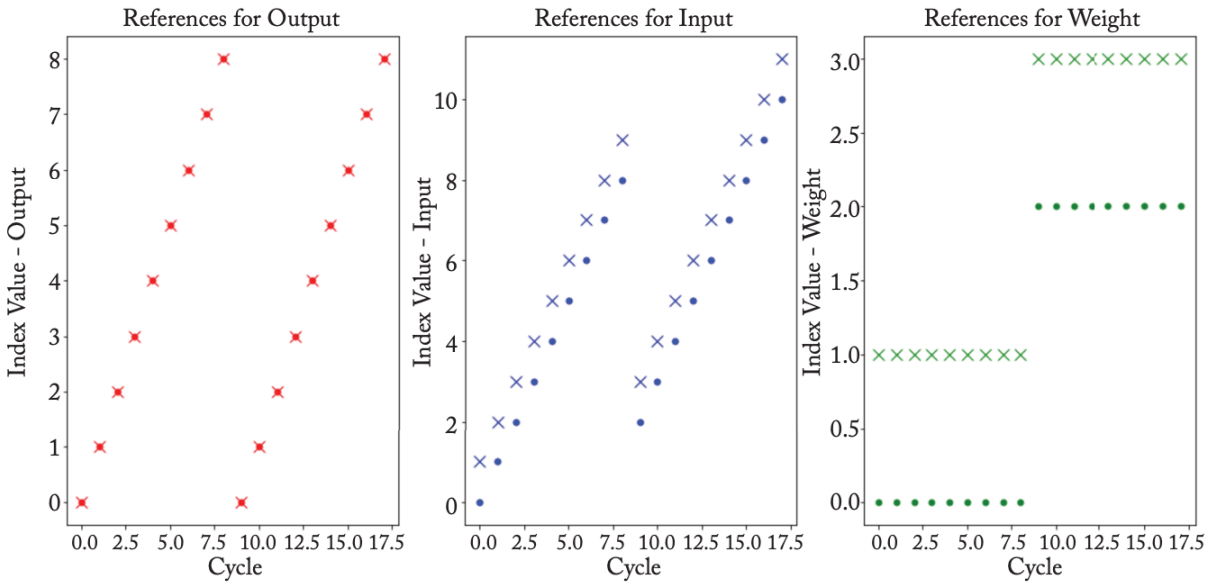
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```



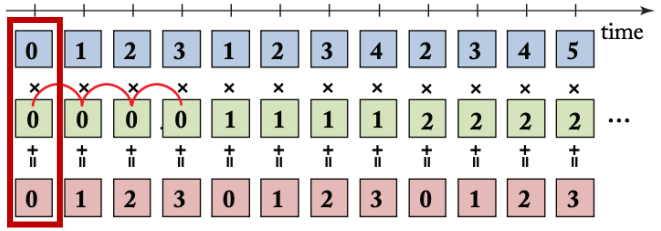
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```



Tensor Notation

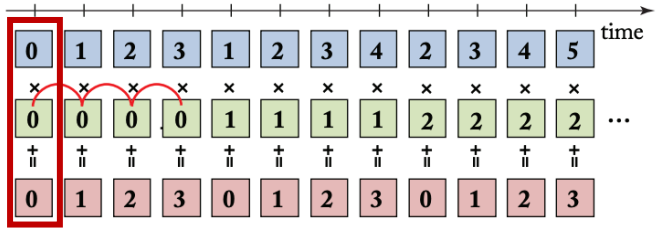
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



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

Tensor Notation

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



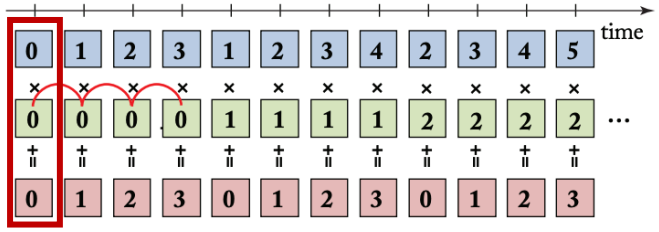
$$O_q = \sum_s I_{q+s} \times F_s$$

Traversal Order (fastest to slowest):

1. q
2. s

Tensor Notation

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

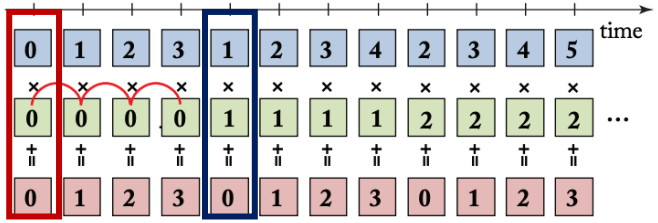


$$O_q = \sum_s I_{q+s} \times F_s$$

Traversal Order (fastest to slowest):

1. q
2. s

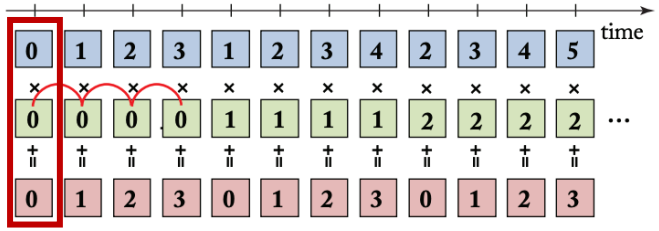
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```



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

Tensor Notation

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

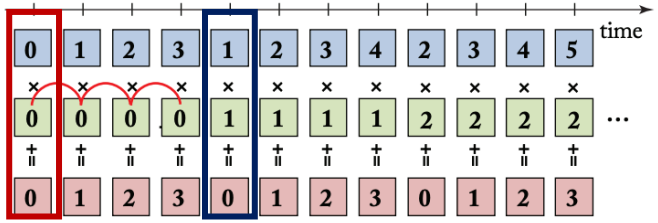


$$O_q = \sum_s I_{q+s} \times F_s$$

Traversal Order (fastest to slowest):

1. q
2. s

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```



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

Parallelize on s

Traversal Order (fastest to slowest):

1. q

Summary

- **Weight Stationary**
  - Best when a **significant amount of on-chip memory** or **cache** exists, allowing the weights to be stored close to the compute units to minimize memory reads.
  - Common in architectures designed to **maximize data reuse**, like **systolic arrays**, and is often used in **Google's TPU** (Tensor Processing Units).

Summary

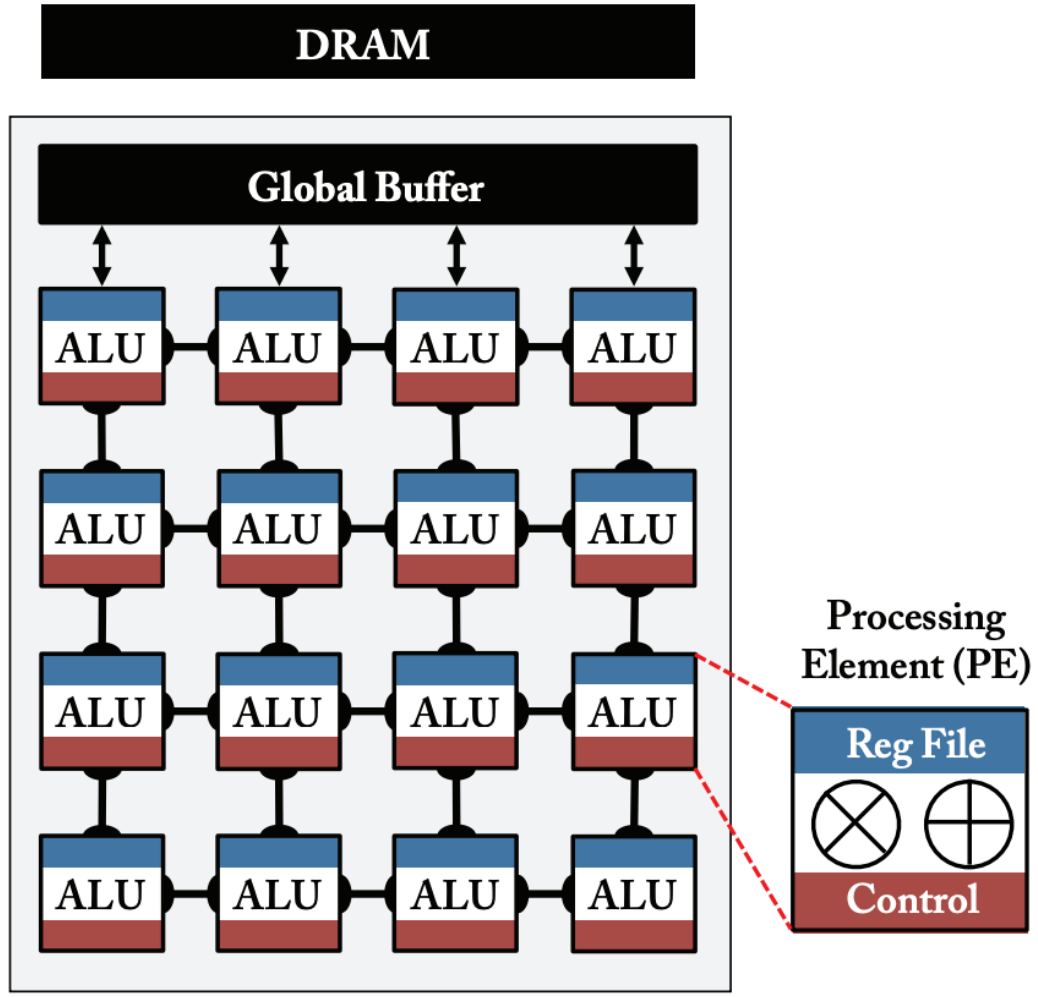
- **Weight Stationary**
  - Best when a **significant amount of on-chip memory** or **cache** exists, allowing the weights to be stored close to the compute units to minimize memory reads.
  - Common in architectures designed to **maximize data reuse**, like **systolic arrays**, and is often used in **Google's TPU** (Tensor Processing Units).
- **Output Stationary**
  - To **reduce the amount of data movement required for writing** back intermediate results to memory.
    - This is because it allows for the accumulation of partial sums directly within the compute unit, minimizing the need to access memory.
  - It can be a common approach in FPGAs and custom ASICs where designers have control over the data paths and can optimize for local storage of intermediate results.

Summary

- **Weight Stationary**
  - Best when a **significant amount of on-chip memory** or **cache** exists, allowing the weights to be stored close to the compute units to minimize memory reads.
  - Common in architectures designed to **maximize data reuse**, like **systolic arrays**, and is often used in **Google's TPU** (Tensor Processing Units).
- **Output Stationary**
  - To **reduce the amount of data movement required for writing** back intermediate results to memory.
    - This is because it allows for the accumulation of partial sums directly within the compute unit, minimizing the need to access memory.
  - It can be a common approach in FPGAs and custom ASICs where designers have control over the data paths and can optimize for local storage of intermediate results.
- **Input Stationary**
  - Used when **input data is large** and is used to **compute multiple outputs**
    - This is characteristic of many convolutional operations in deep learning.
  - Effective in reducing the bandwidth needed for input data, which can be advantageous in systems where input data movement is a bottleneck.

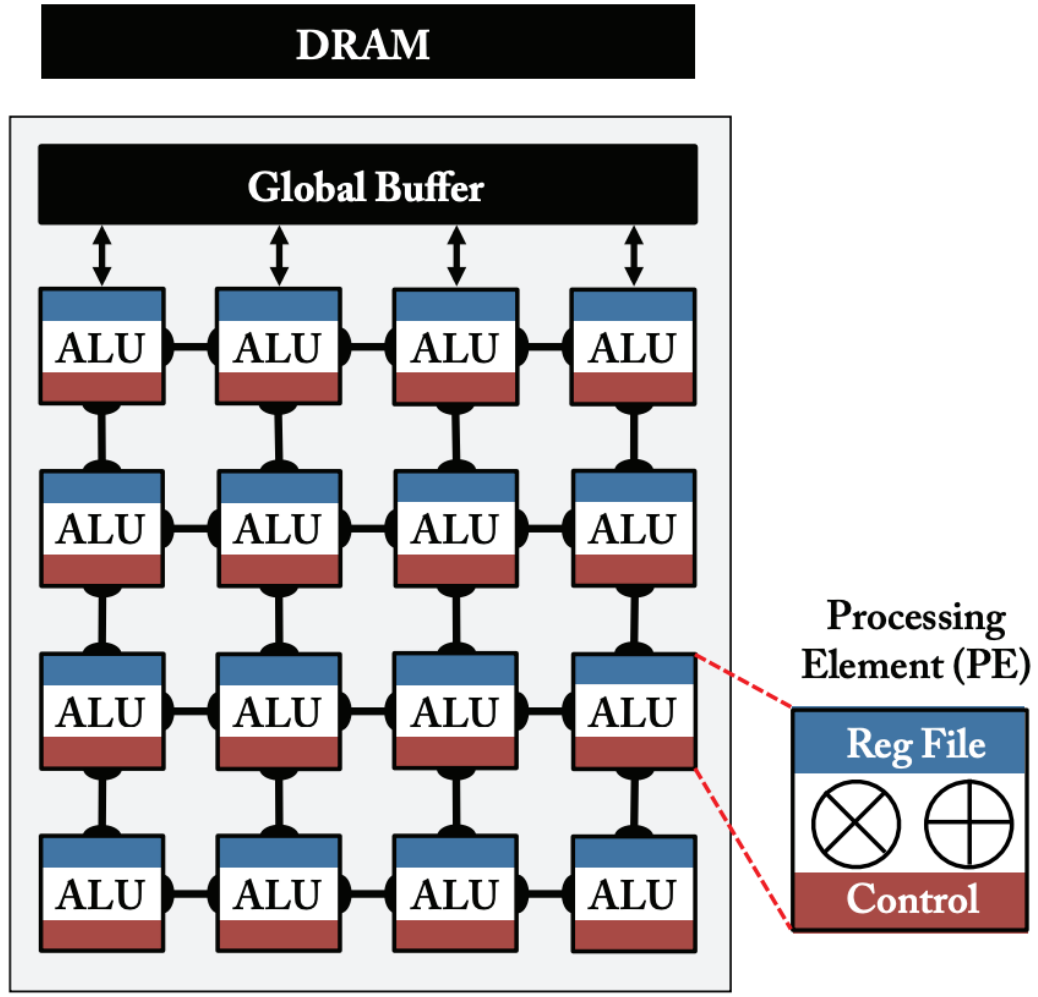
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity



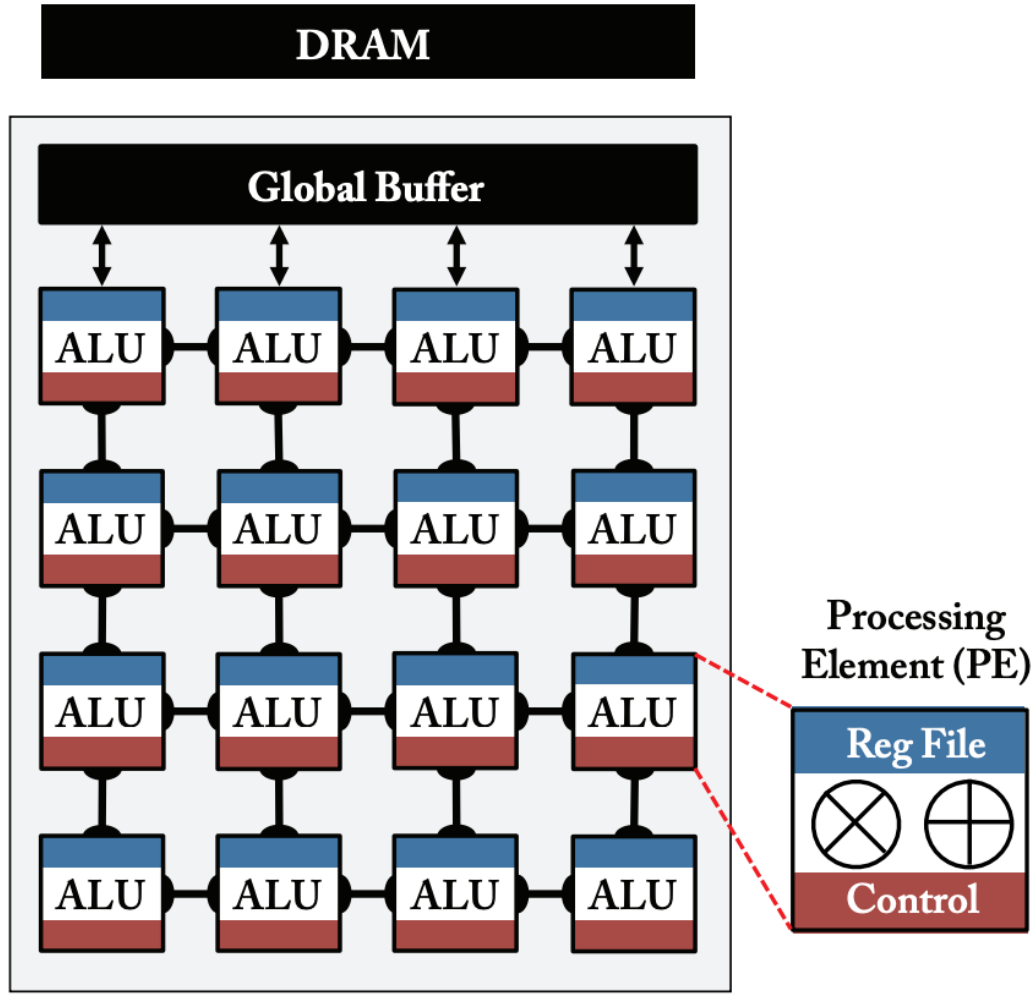
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity
- **Memory Hierarchy**



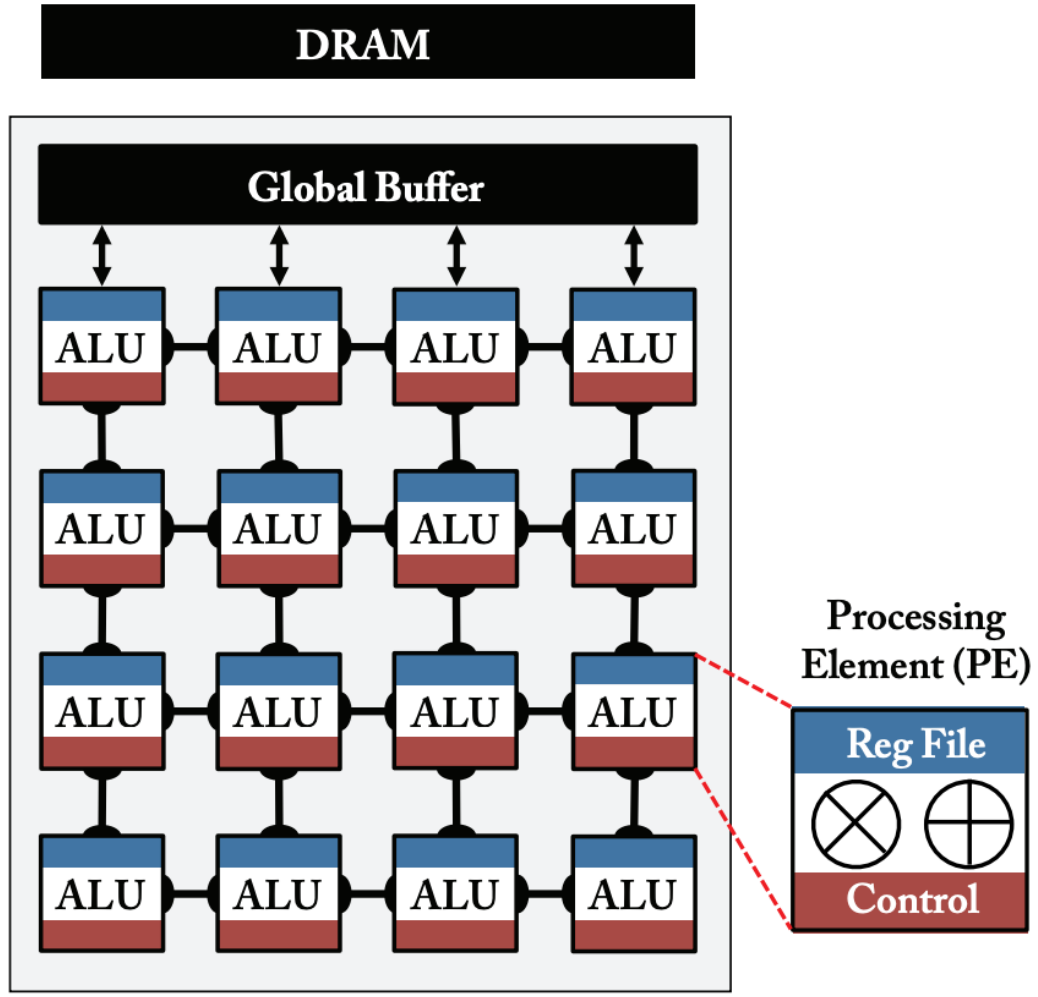
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity
- **Memory Hierarchy**
- **Dataflow Manager**
  - Implements data movement strategies like systolic arrays, weight-stationary, output-stationary, etc.
  - Manages the efficient use of the memory hierarchy to minimize off-chip memory access.



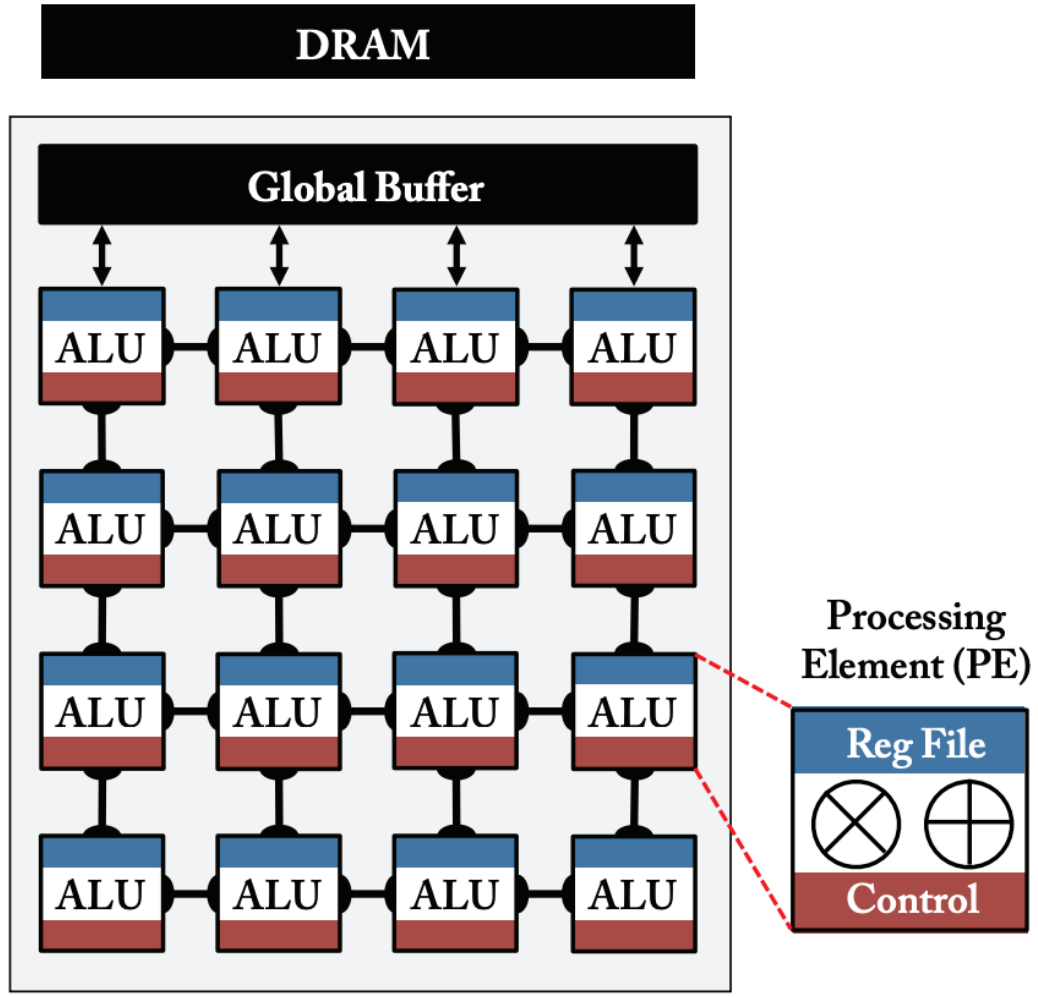
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity
- **Memory Hierarchy**
- **Dataflow Manager**
  - Implements data movement strategies like systolic arrays, weight-stationary, output-stationary, etc.
  - Manages the efficient use of the memory hierarchy to minimize off-chip memory access.
- **Pooling & Other Units**



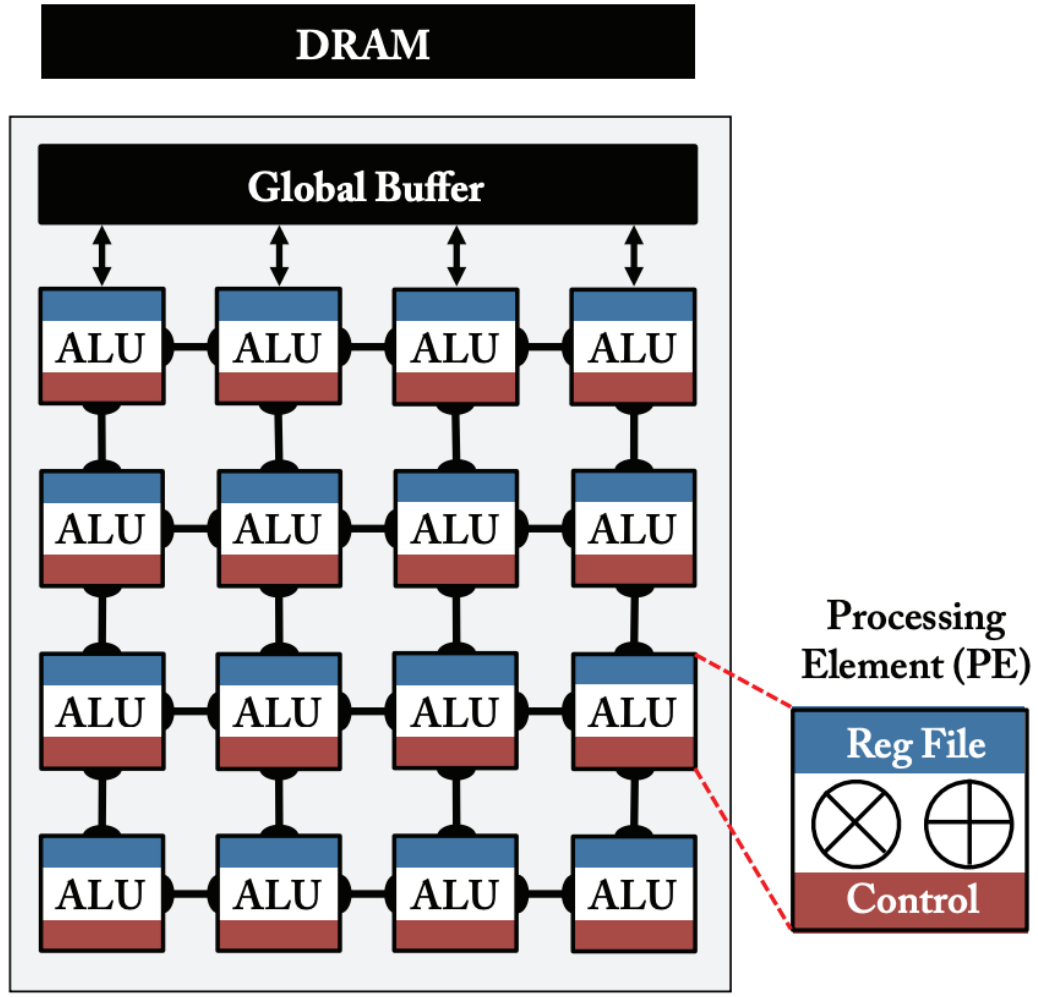
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity
- **Memory Hierarchy**
- **Dataflow Manager**
  - Implements data movement strategies like systolic arrays, weight-stationary, output-stationary, etc.
  - Manages the efficient use of the memory hierarchy to minimize off-chip memory access.
- **Pooling & Other Units**
- **Mechanism for Configuration and Programmability**

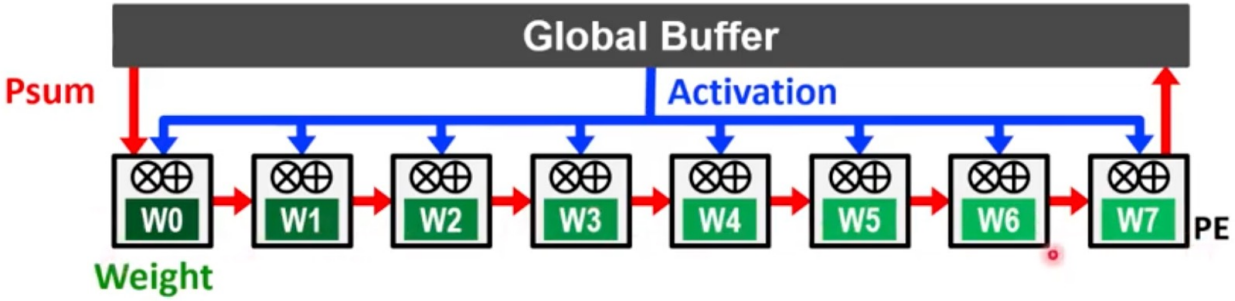


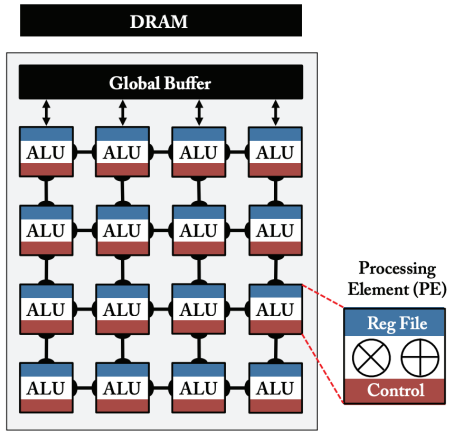
Generic DNN Accelerator

- **Processing Element (PE) Array**
  - Basic arithmetic operations like multiplication and accumulation
  - Non-linearity
- **Memory Hierarchy**
- **Dataflow Manager**
  - Implements data movement strategies like systolic arrays, weight-stationary, output-stationary, etc.
  - Manages the efficient use of the memory hierarchy to minimize off-chip memory access.
- **Pooling & Other Units**
- **Mechanism for Configuration and Programmability**
- **Interface**:
  - Communication interfaces to connect with CPUs, GPUs, or other system components. This can be PCI-E, AXI, or other standard interfaces.



Weight Stationary

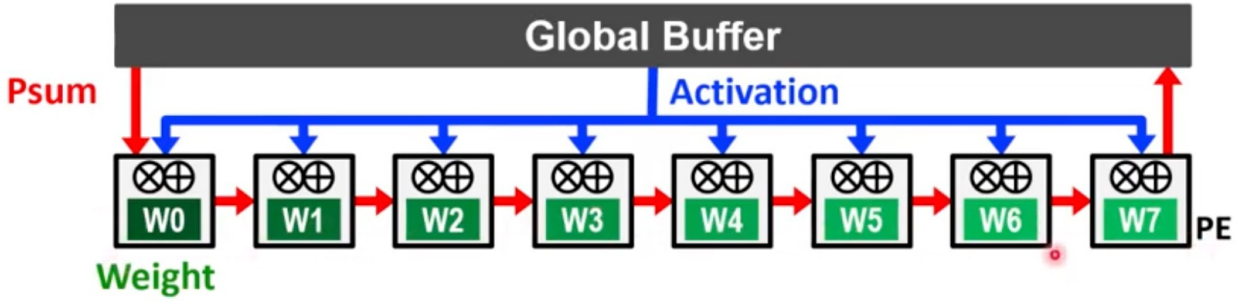


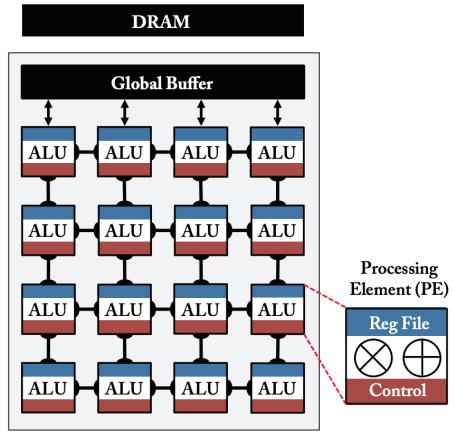


1.Goal:

1. Minimize energy consumption when reading weights.
2. Maximize weight reuse from the register file (RF) at each PE.

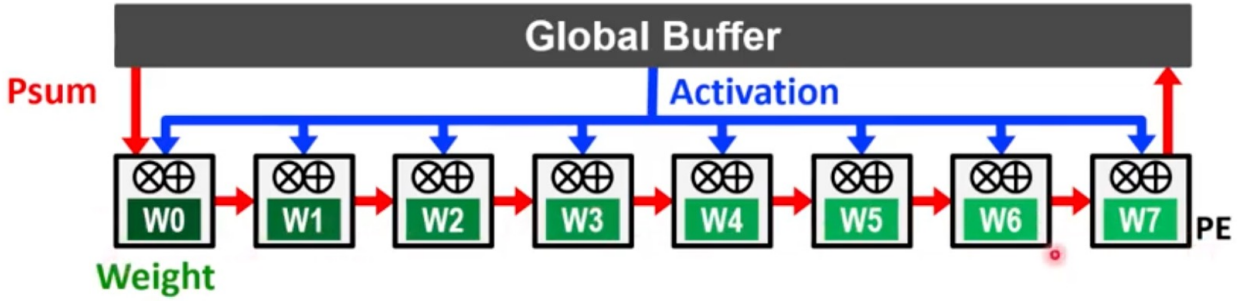
Weight Stationary

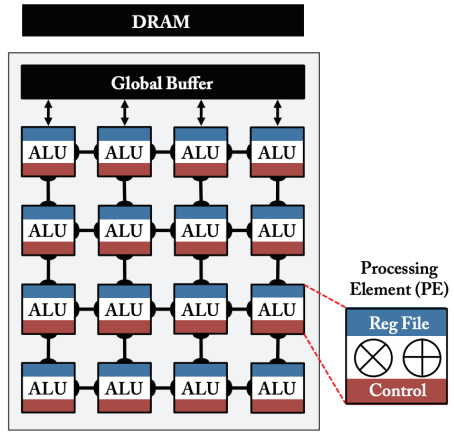




- **How it Works**:
  - Weights are loaded from DRAM into the **RF of each PE** and remain **stationary** for subsequent accesses.
  - Execute **as many MAC operations** using the same weight while it's in the RF.
  - Amplifies convolutional and filter reuse of weights.

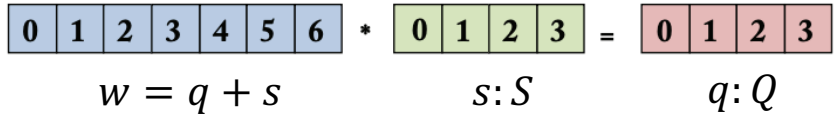
Weight Stationary

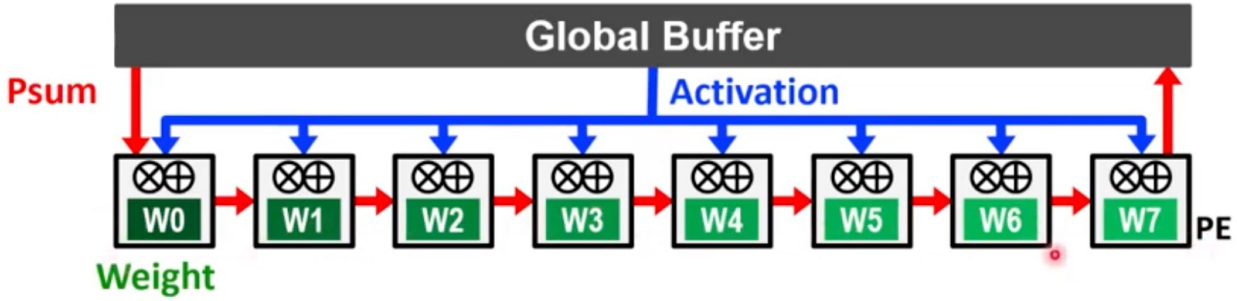


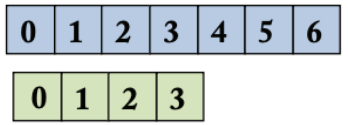


- **How it Works**:
  - Weights are loaded from DRAM into the **RF of each PE** and remain **stationary** for subsequent accesses.
  - Execute **as many MAC operations** using the same weight while it's in the RF.
  - Amplifies convolutional and filter reuse of weights.
- **Data Movement**:
  - **Input feature map** activations **broadcast** to all PEs.
  - **Partial sums move** through the spatial array and global buffer.
  - Subsequent spatial accumulation across the PE array.

Weight Stationary



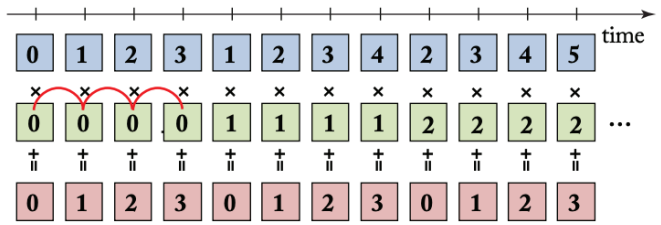




$s_0 = 0$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```

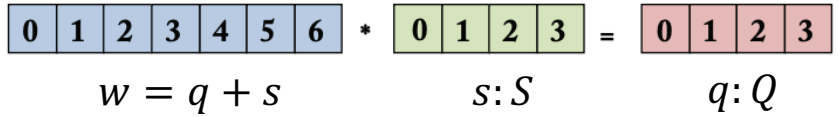


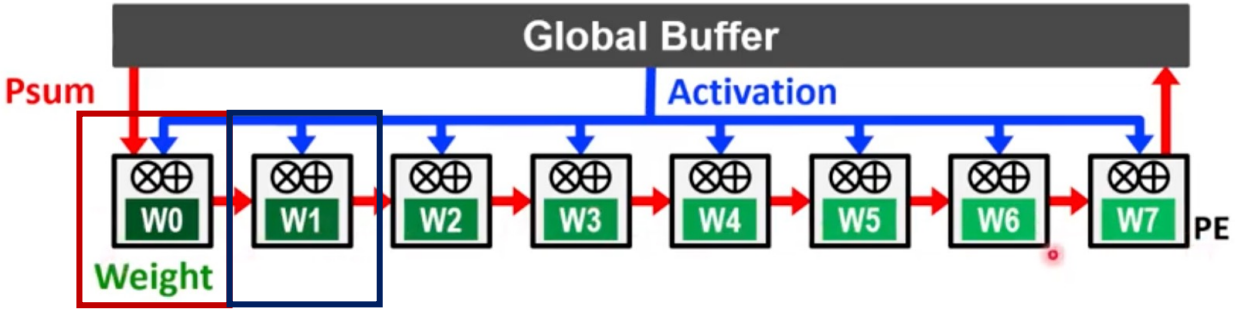
$s_0 = 1$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

- Each processor:
  - Works on one weight
  - All inputs
  - Partial output should be combined

Weight Stationary





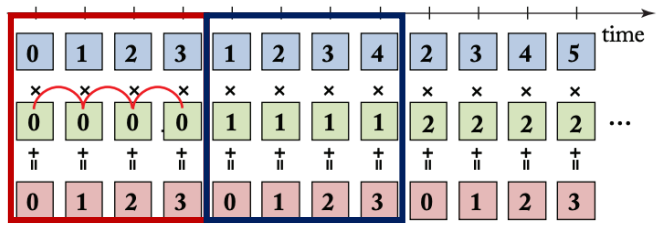

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

for s1 in range(S1):
  for q in range(Q):
    paarallel-for s0 in range(S0):
      s = s1*S0 + s0
      w = q+s
      o[q] += i[w] * f[s]
```

$s_0 = 0$

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 0
    w = q+s
    o[q] += i[w] * f[s]
```

$s_0 = 1$

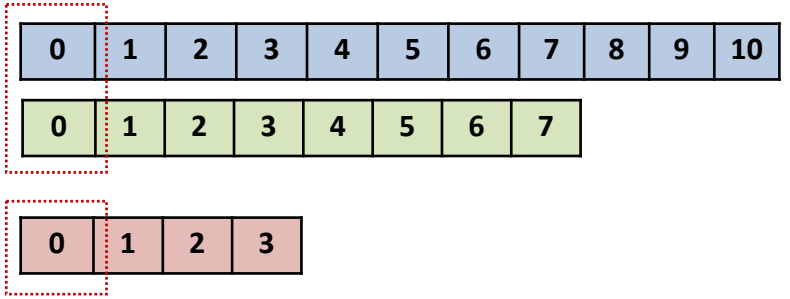
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations
for s1 in range(S1):
  for q in range(Q):
    s = s1*2 + 1
    w = q+s
    o[q] += i[w] * f[s]
```

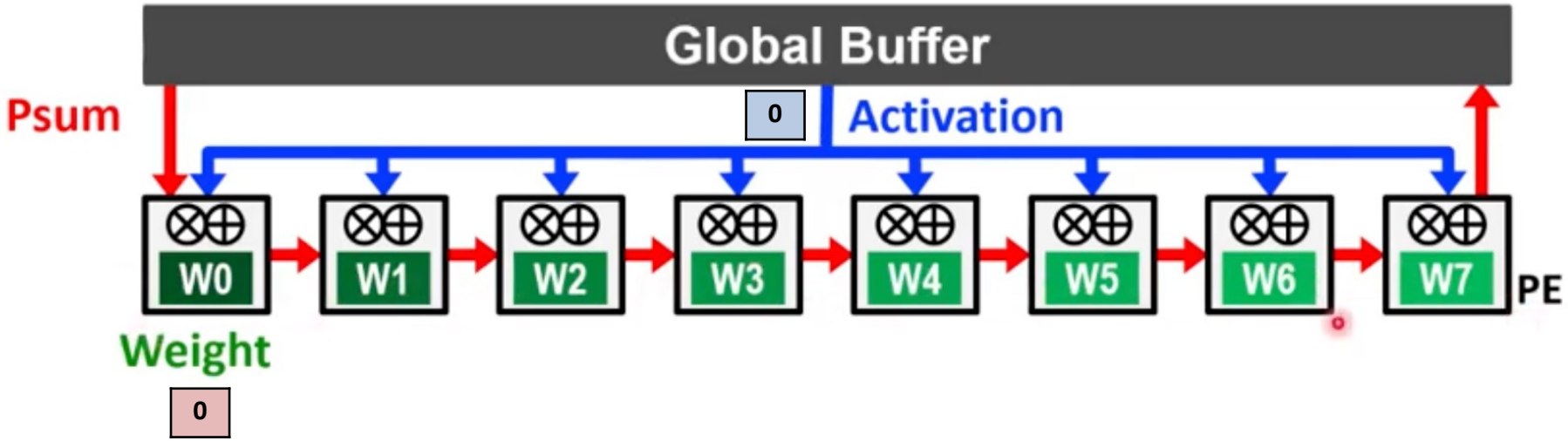


PE1 PE2

- Each processor:
  - Works on one weight
  - All inputs
  - Partial output should be combined

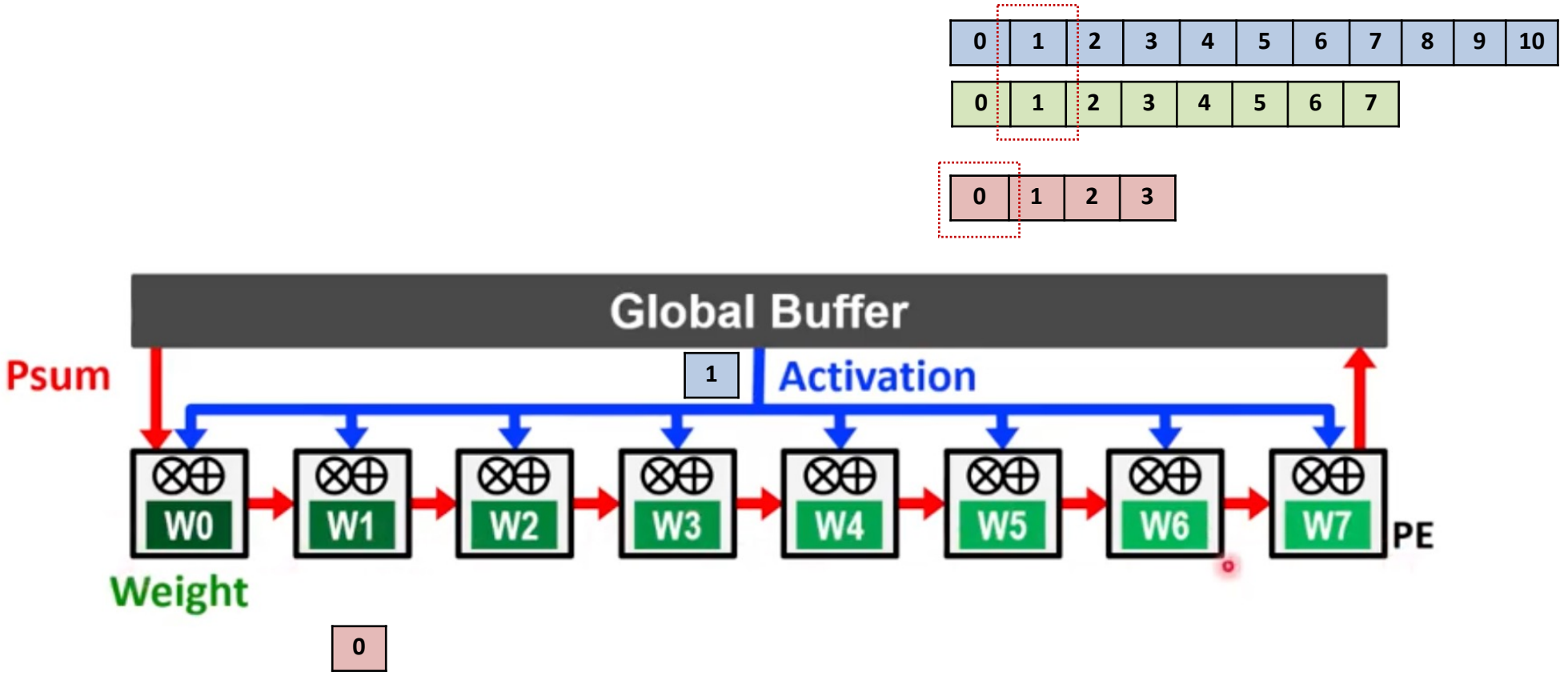
Weight Stationary





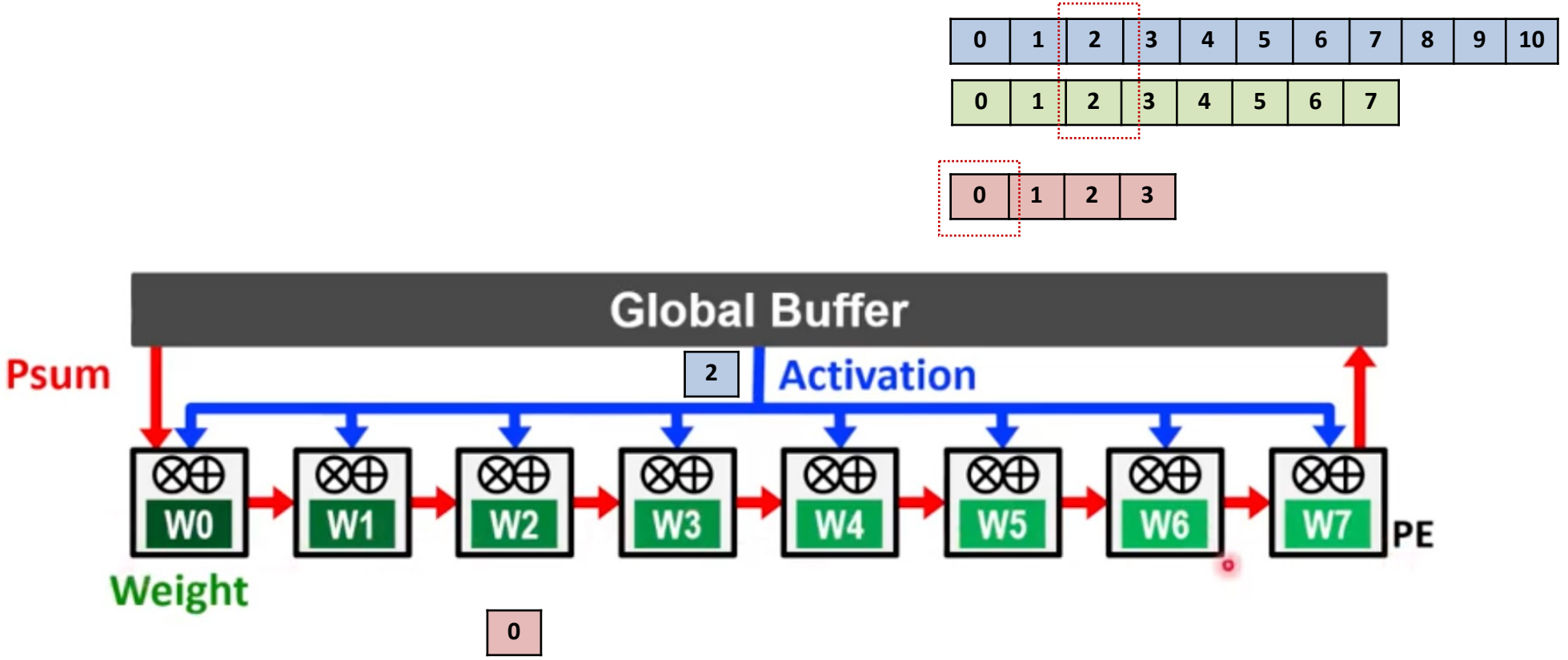
Focus on output 0

Weight Stationary



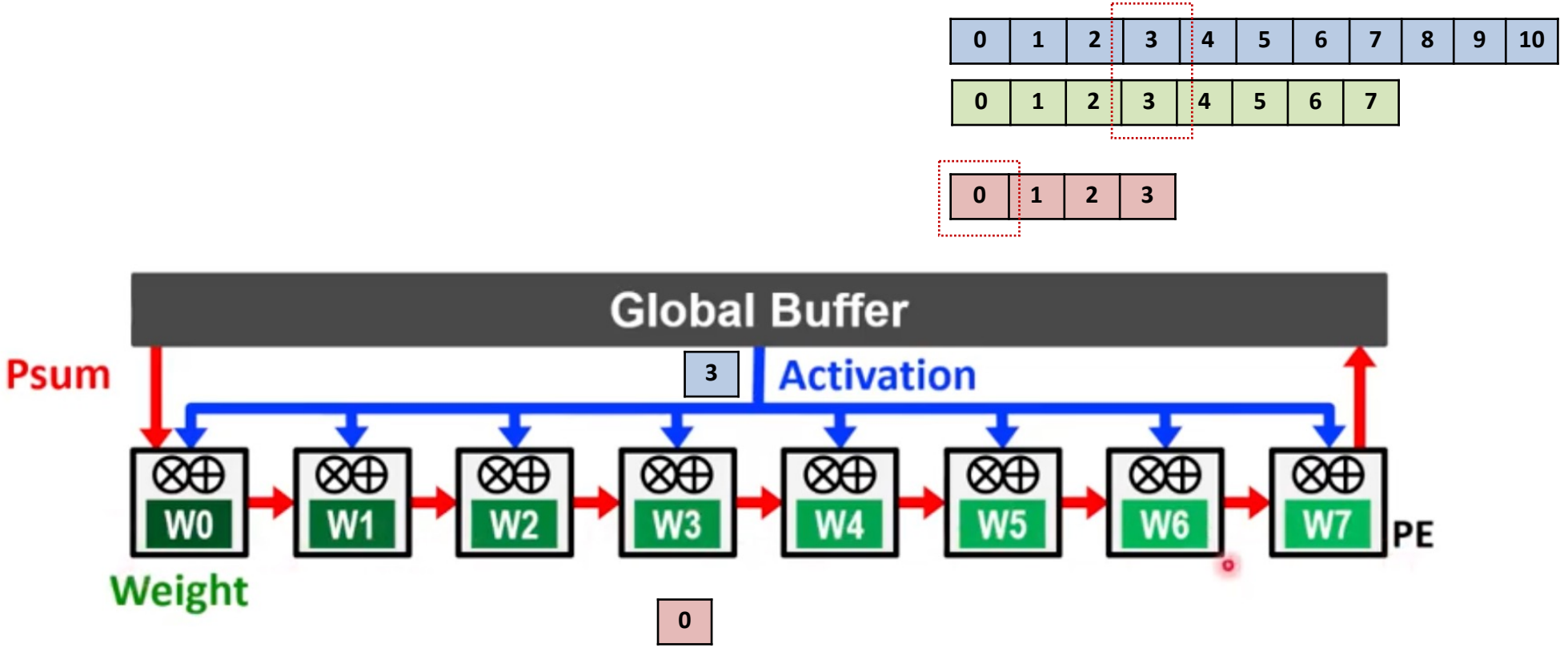
Focus on output 0

Weight Stationary



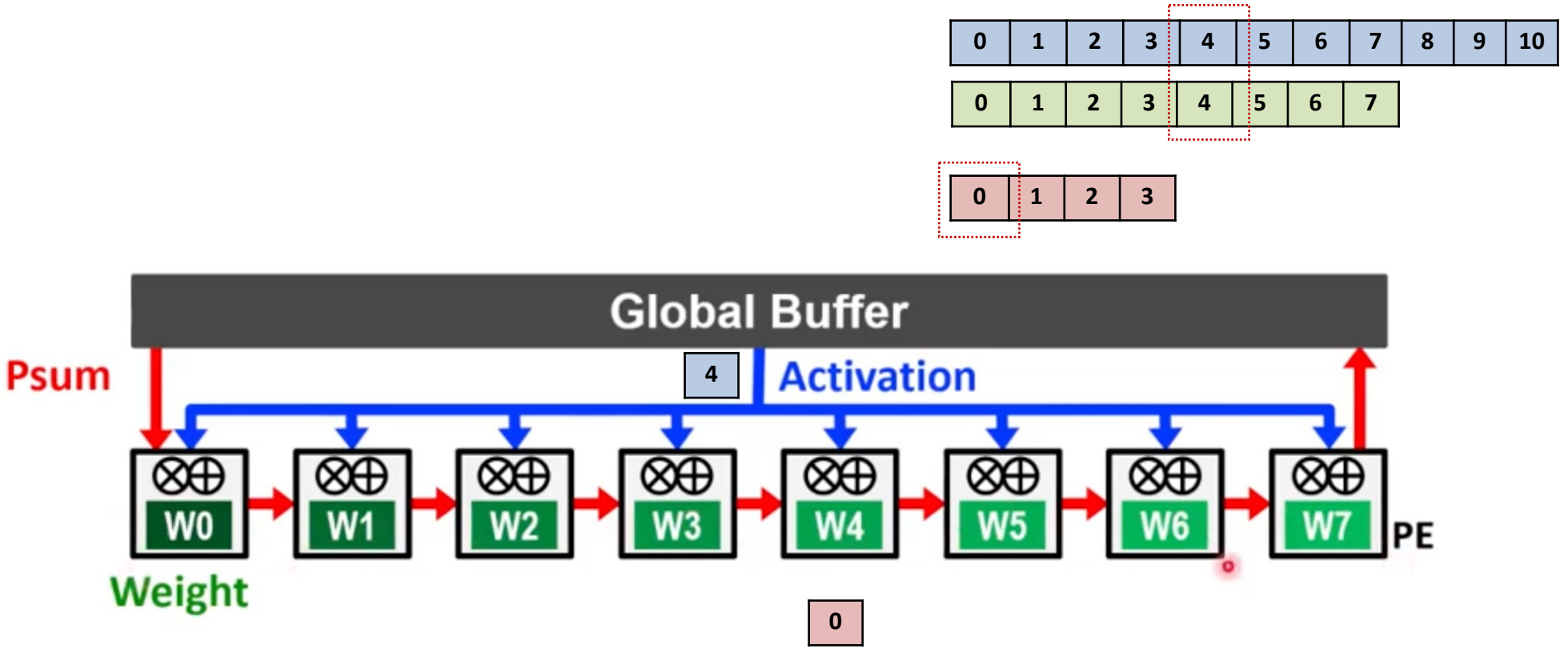
Focus on output 0

Weight Stationary



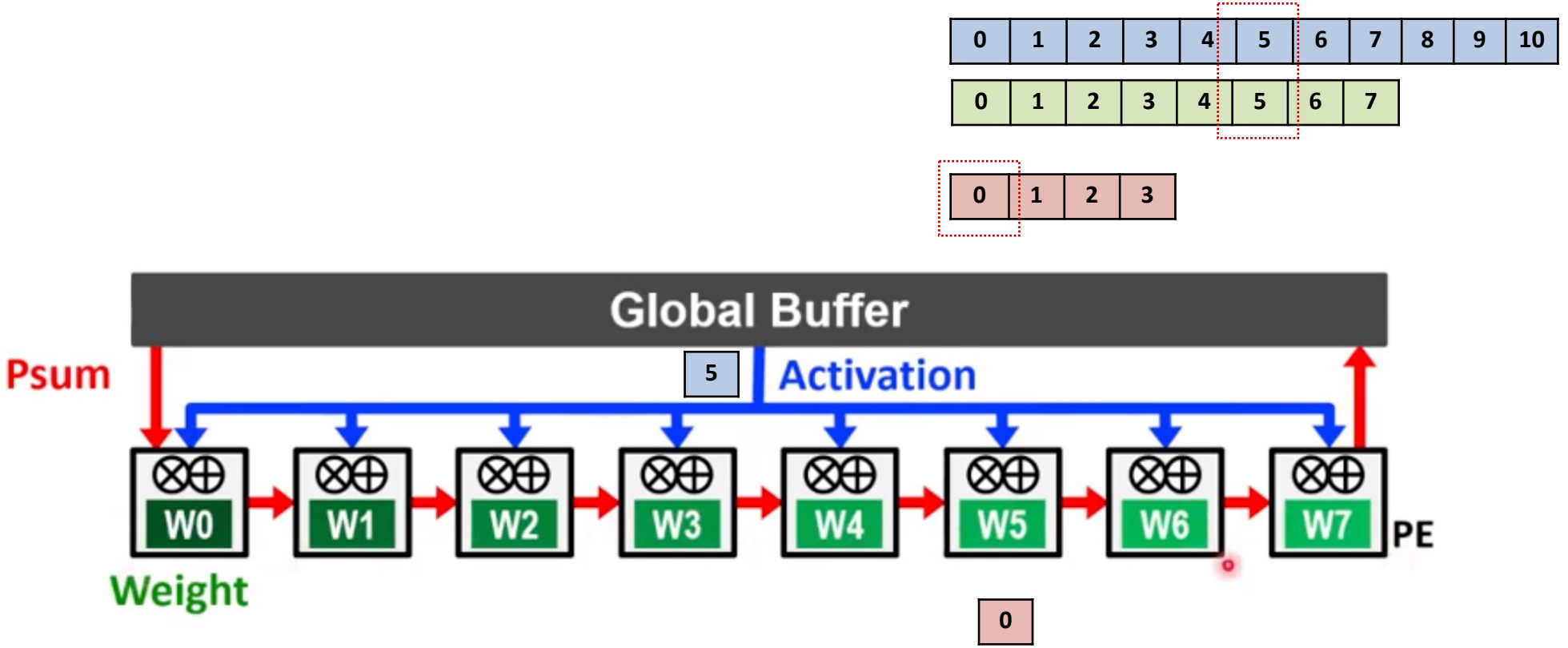
Focus on output 0

Weight Stationary



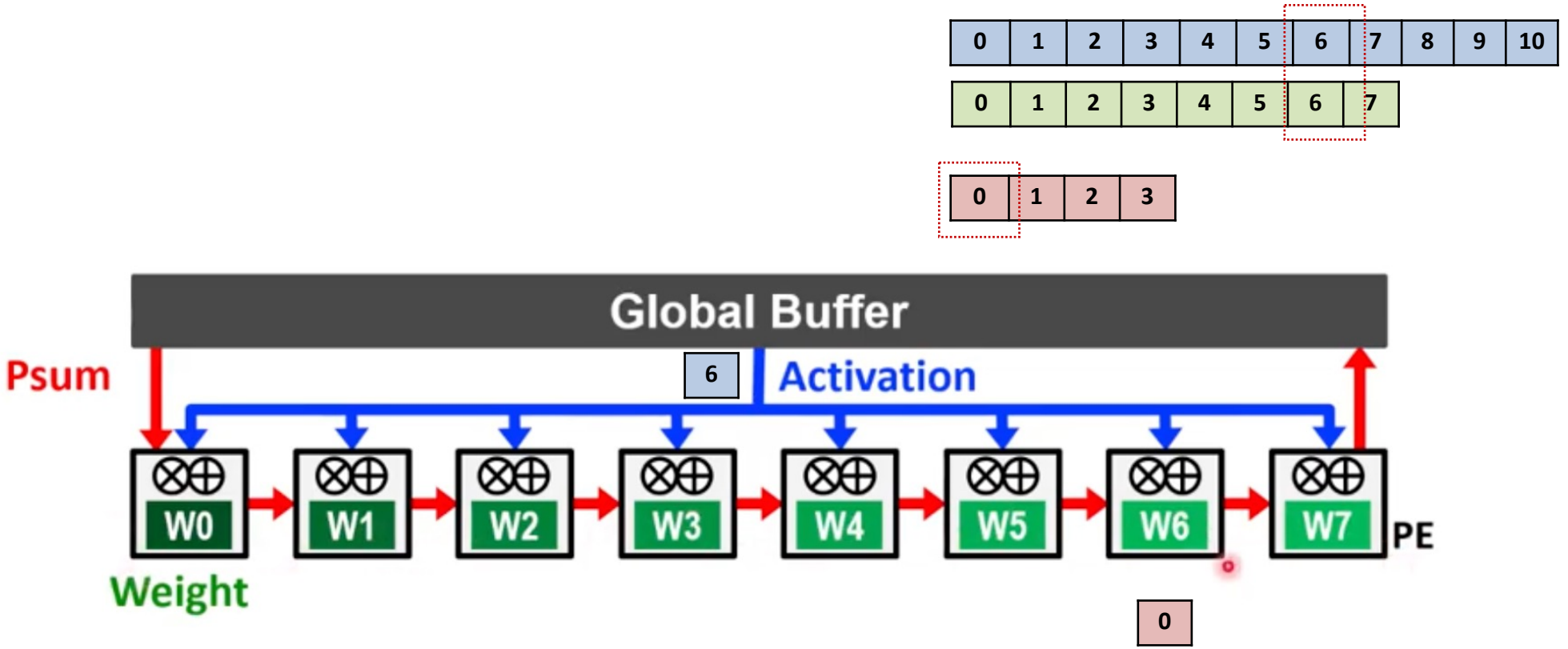
Focus on output 0

Weight Stationary



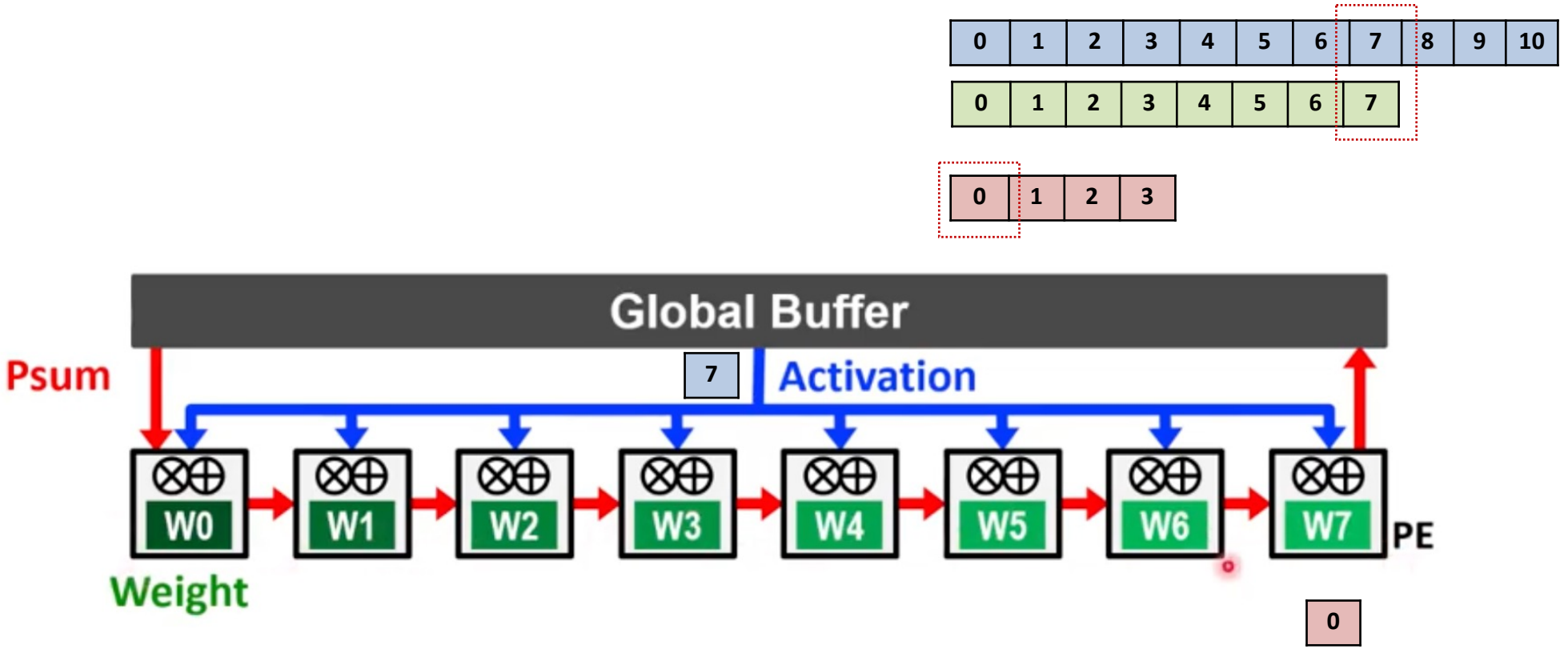
Focus on output 0

Weight Stationary



Focus on output 0

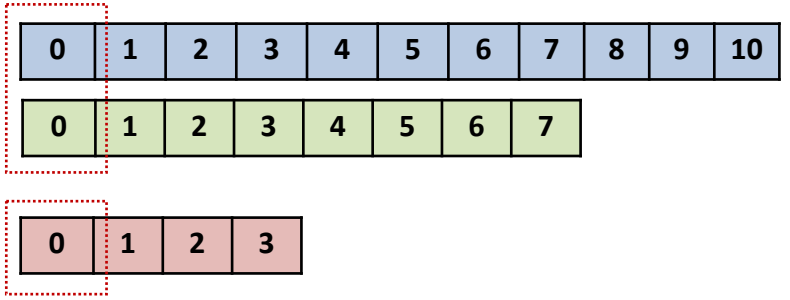
Weight Stationary

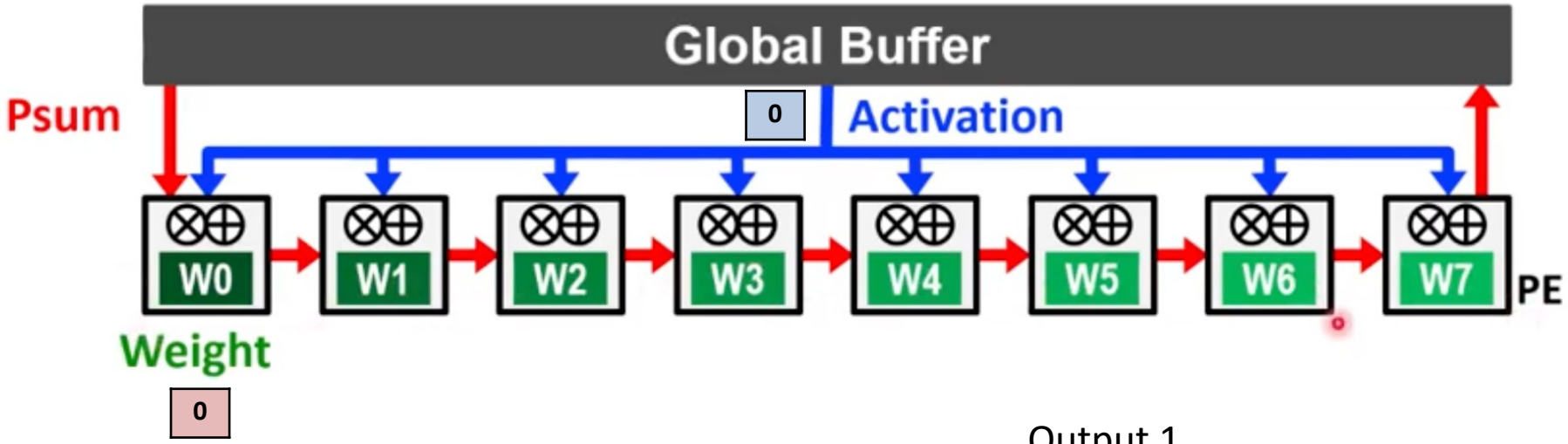


Focus on output 0

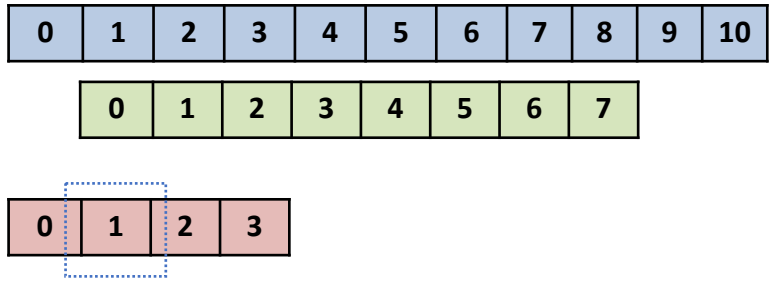
Weight Stationary

Output 0





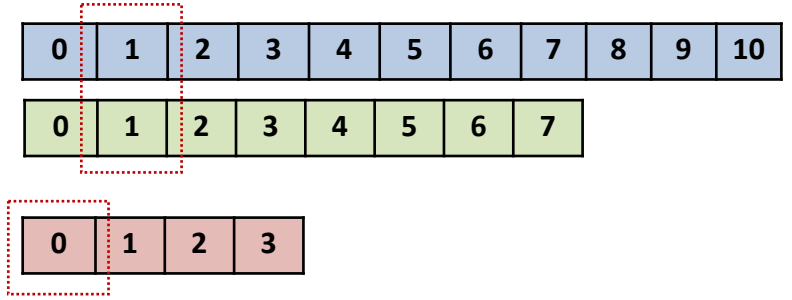
Output 1

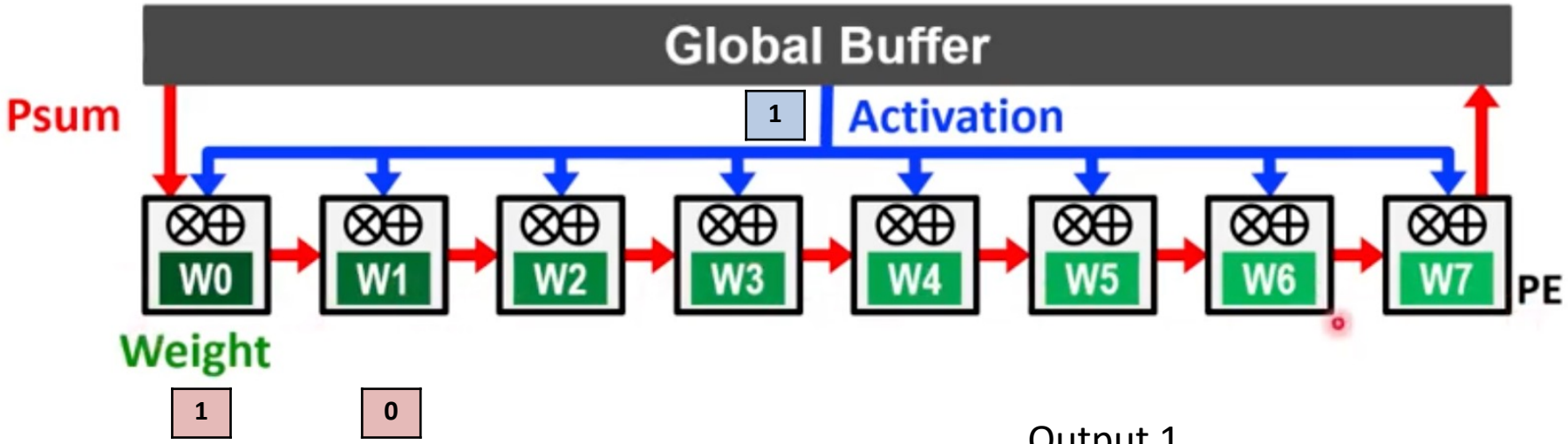


Let's see this again with focus on output 0 and 1

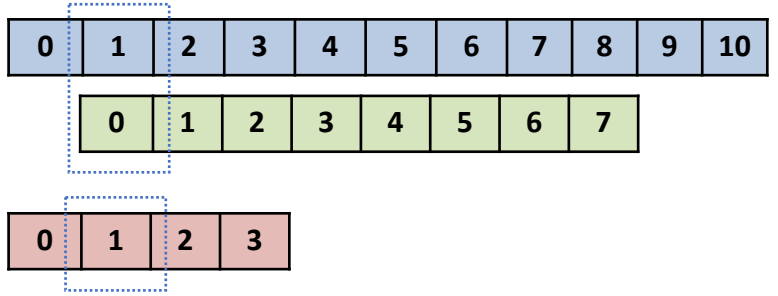
Weight Stationary

Output 0





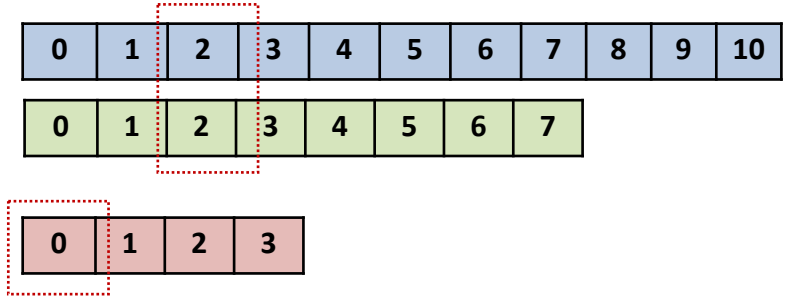
Output 1

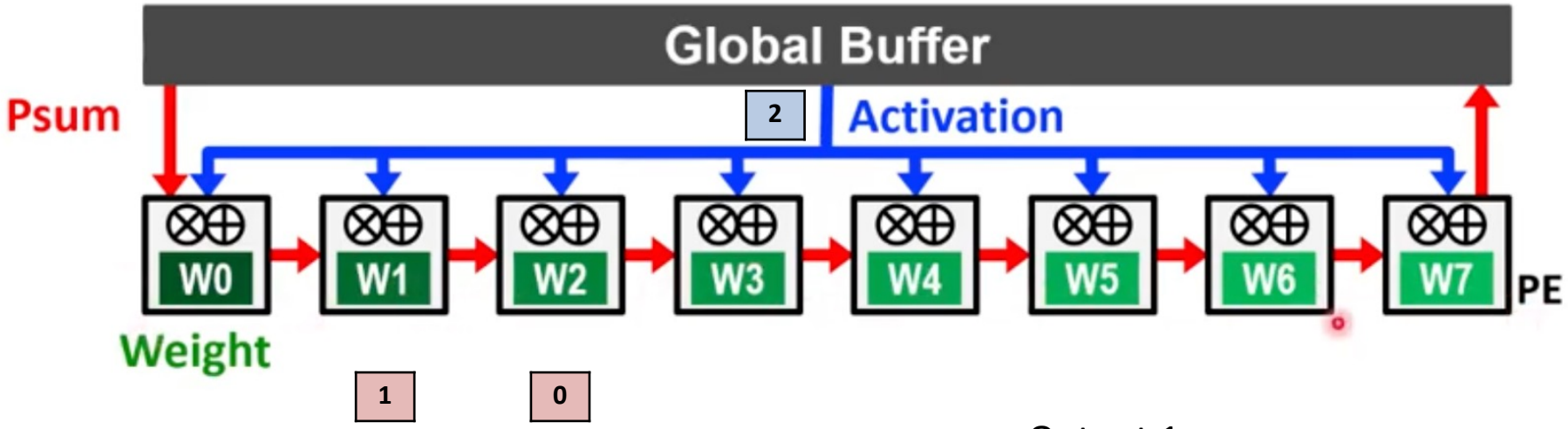


Focus on output 0 and 1

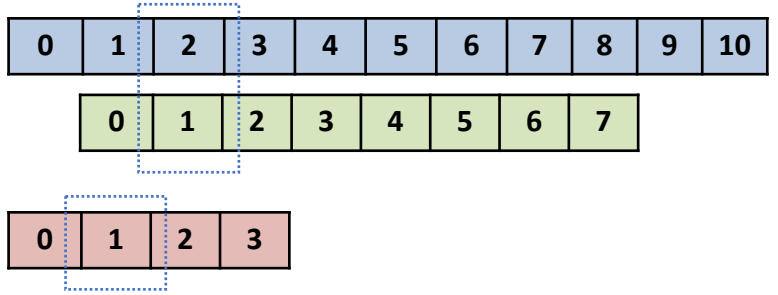
Weight Stationary

Output 0



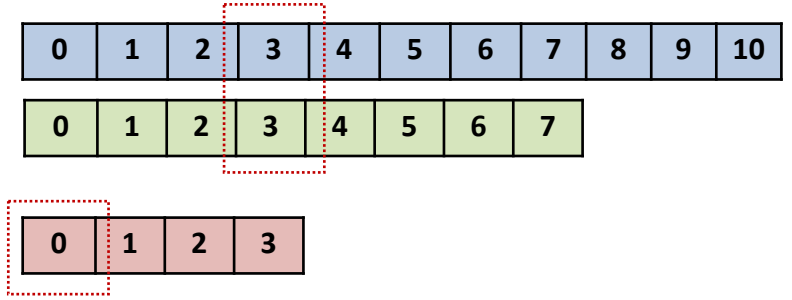


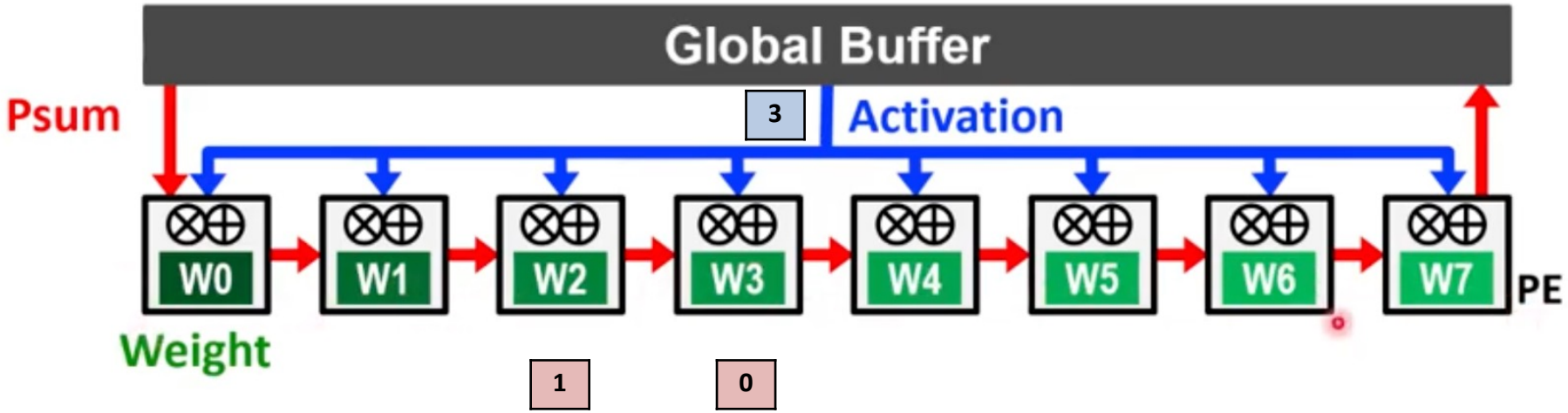
Output 1



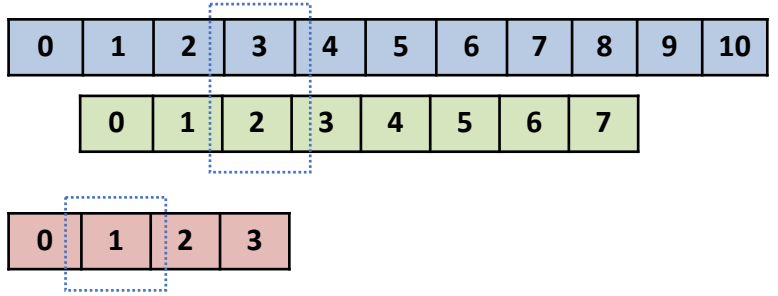
Weight Stationary

Output 0



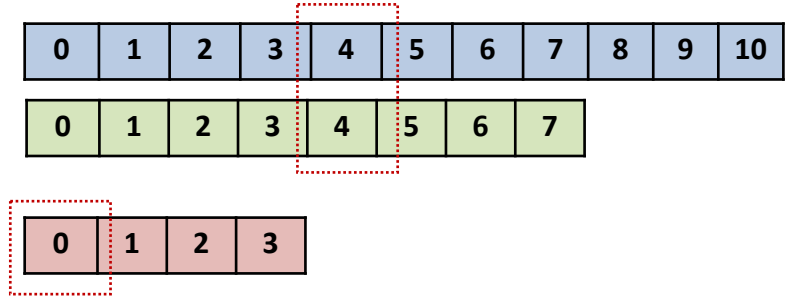


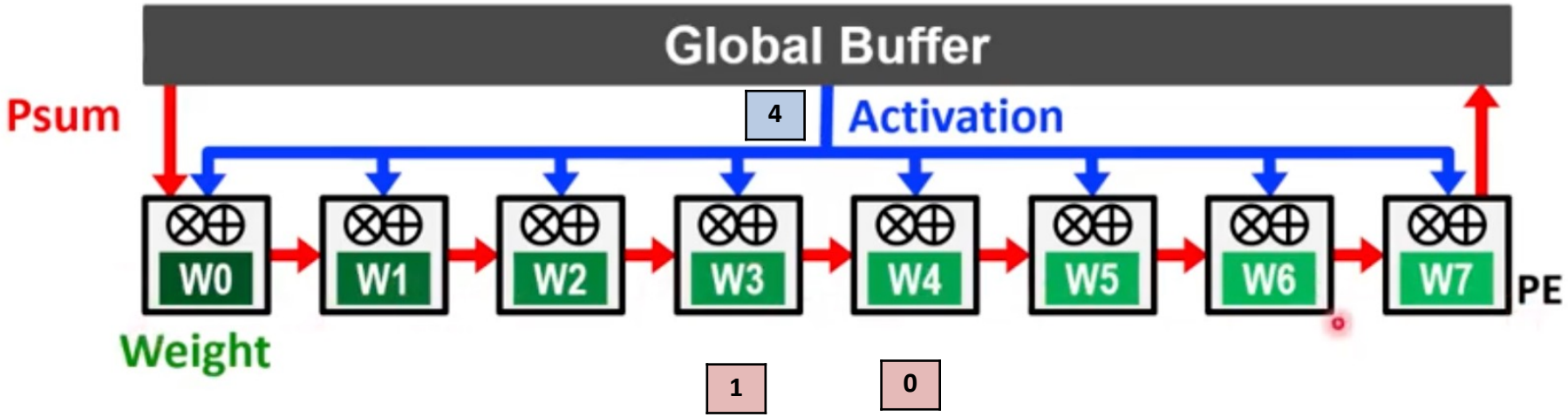
Output 1



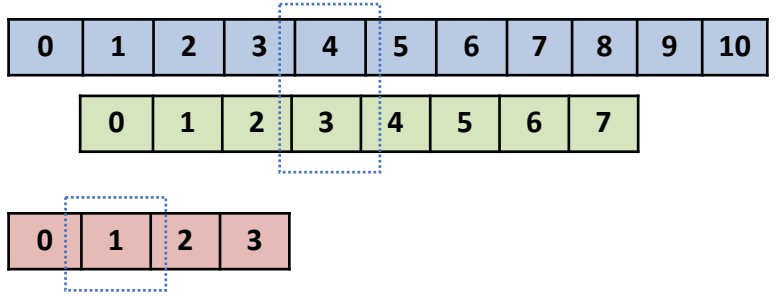
Weight Stationary

Output 0

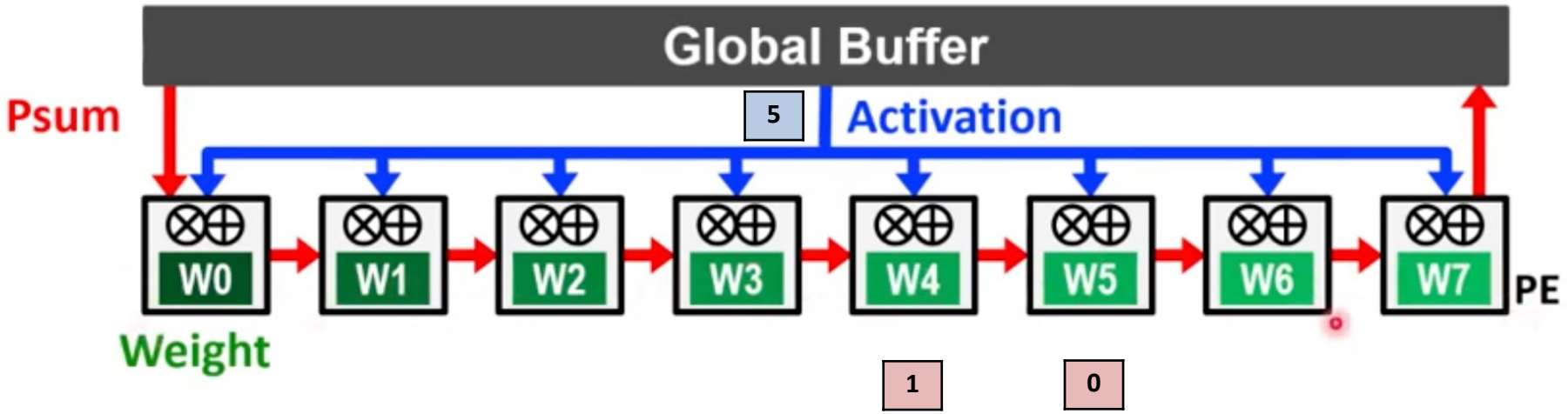




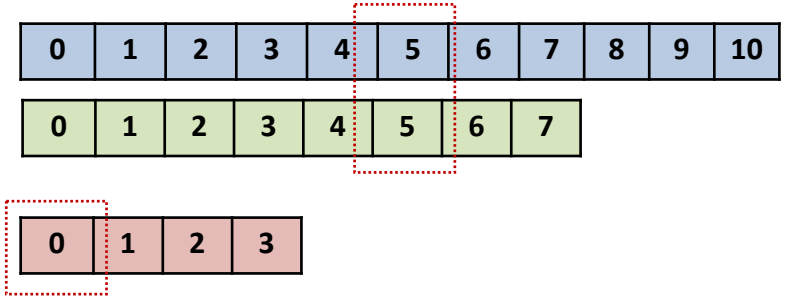
Output 1



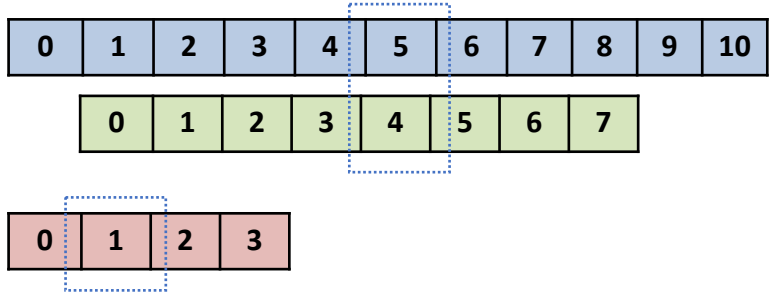
Weight Stationary



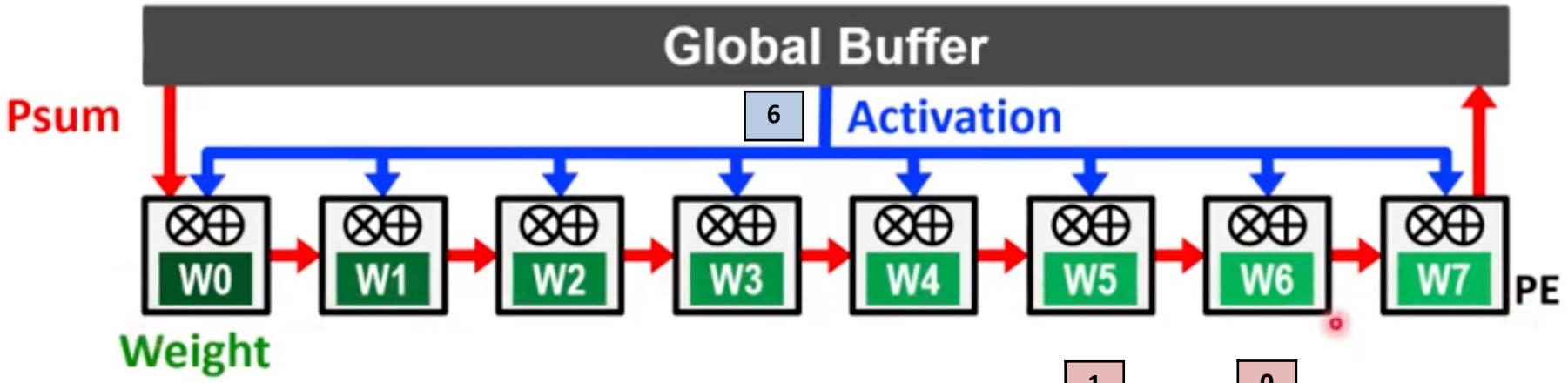
Output 0



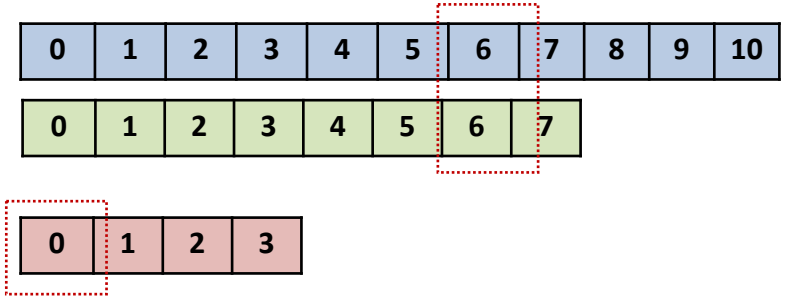
Output 1



Weight Stationary

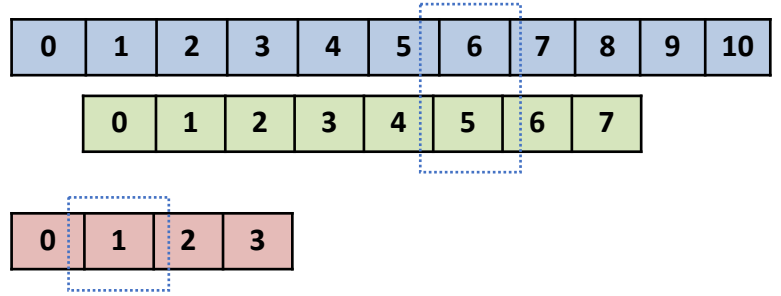


Output 0



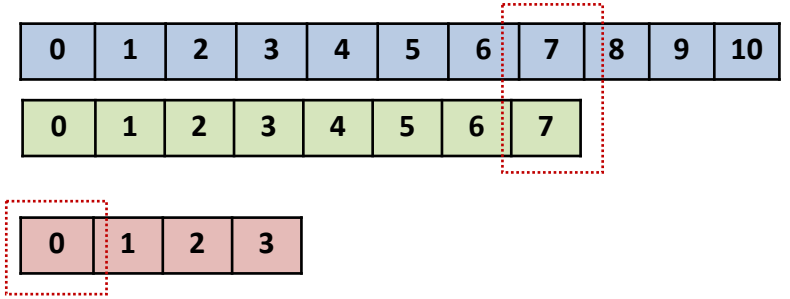
1 0

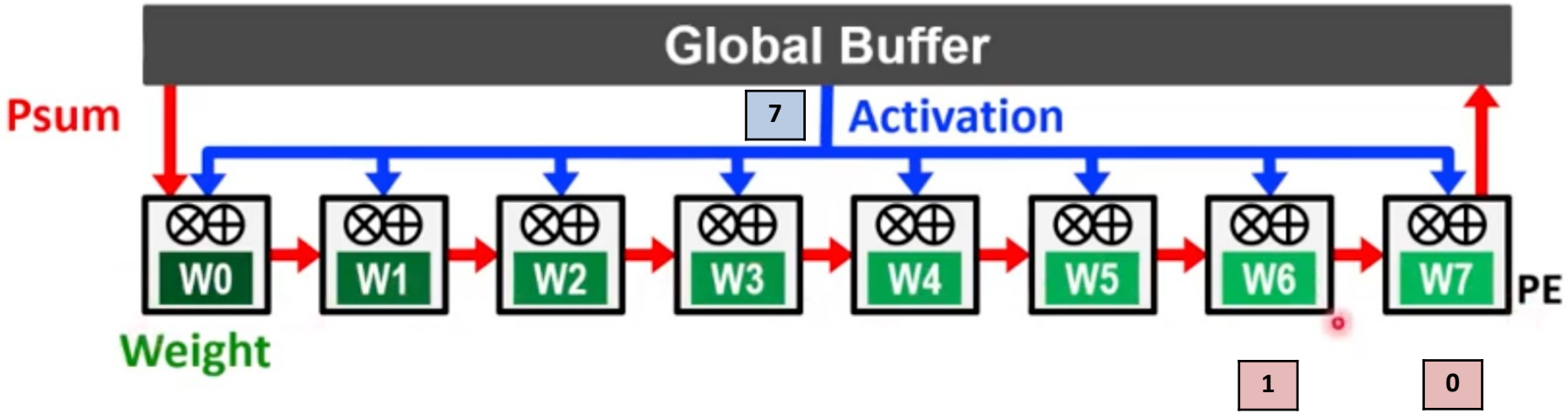
Output 1



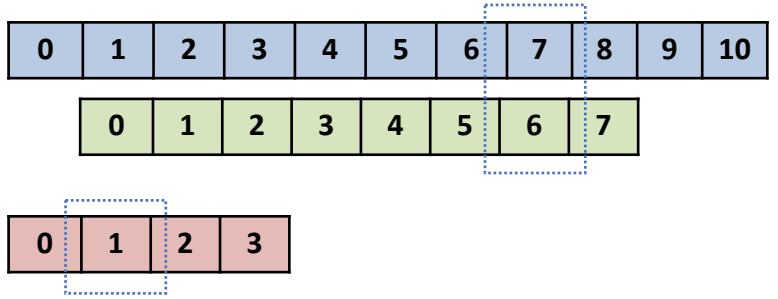
Weight Stationary

Output 0



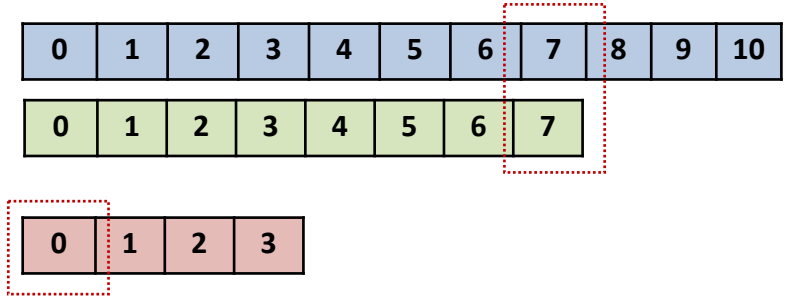


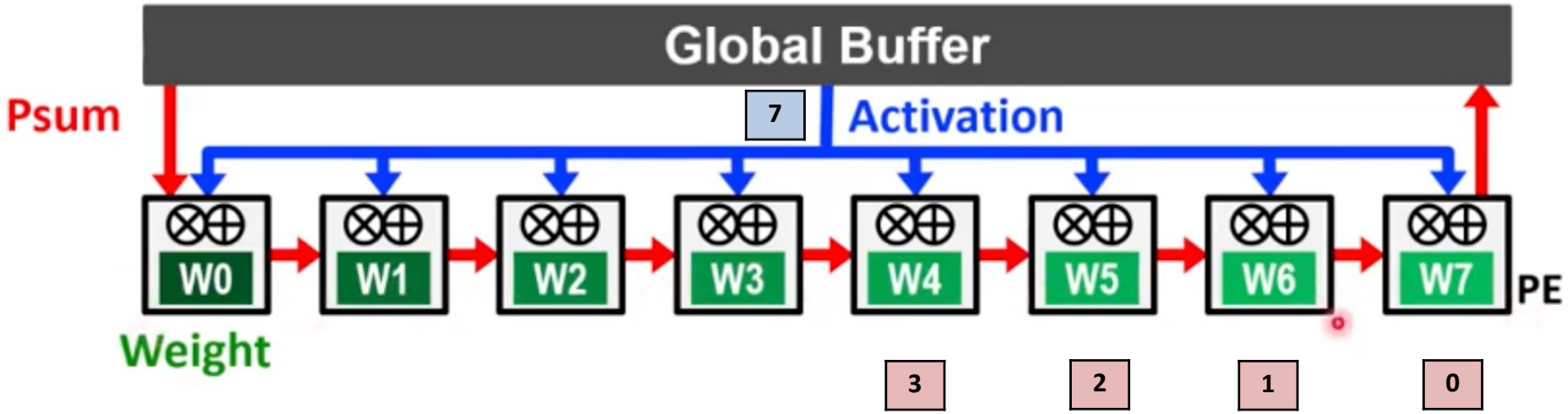
Output 1



Weight Stationary

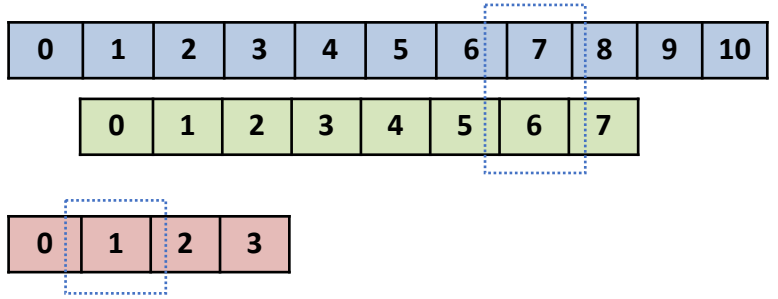
Output 0



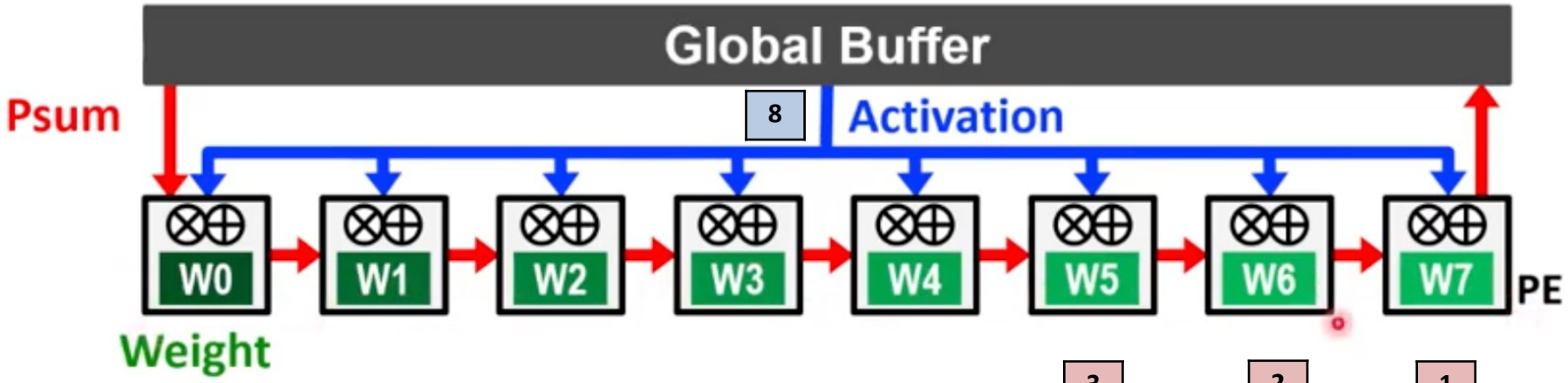


Output 1

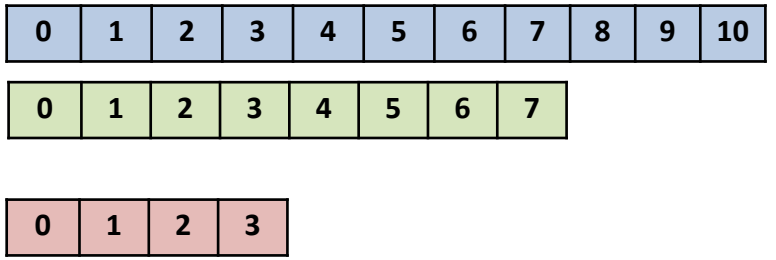
Other outputs following

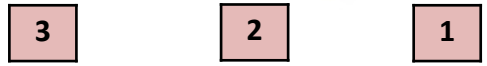


Weight Stationary



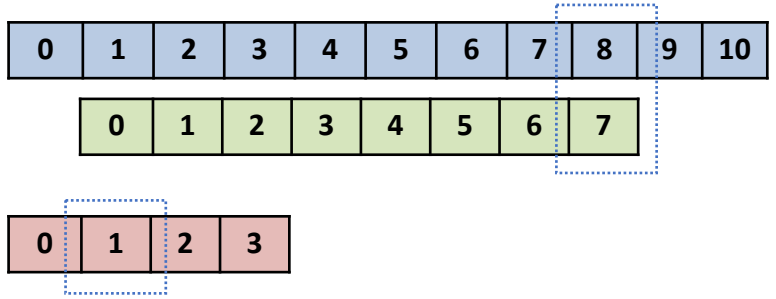
Output 0



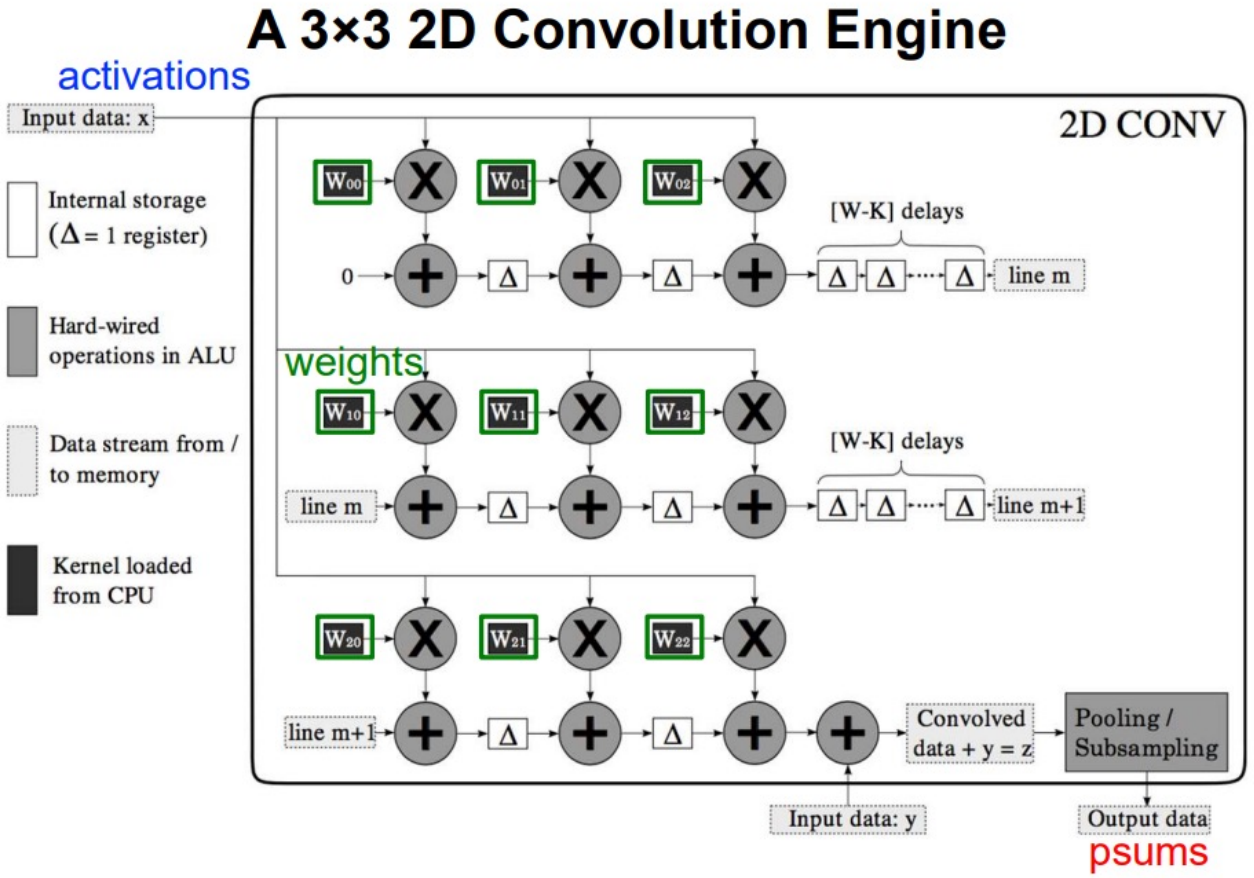


Output 1

Other outputs following



Weight Stationary



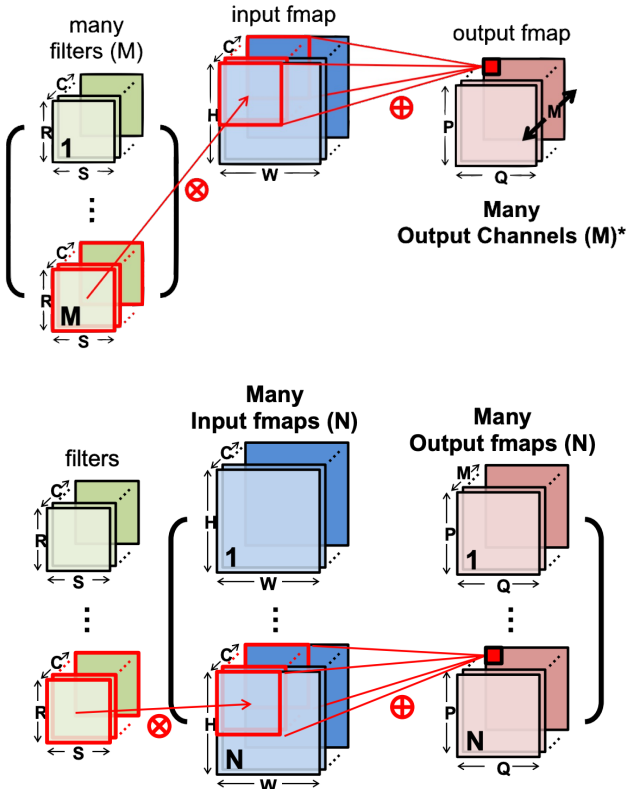
[Farabet et al., ICCV 2009]

WS dataflow as implemented in nn-X or neuFlow

Weight Stationary for Convolution Layer

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

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

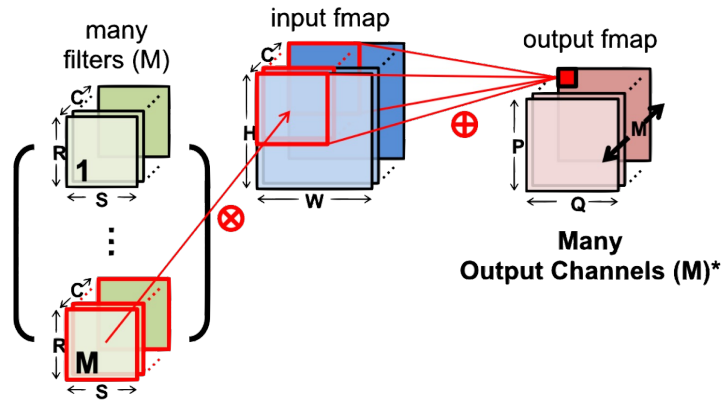


Weight Stationary for Convolution Layer

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

Assuming: stride of 1 and no batching

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

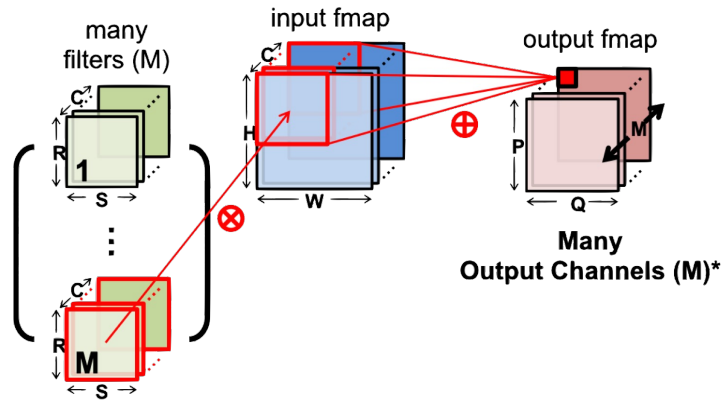


Weight Stationary for Convolution Layer

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

Assuming: stride of 1 and no batching

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

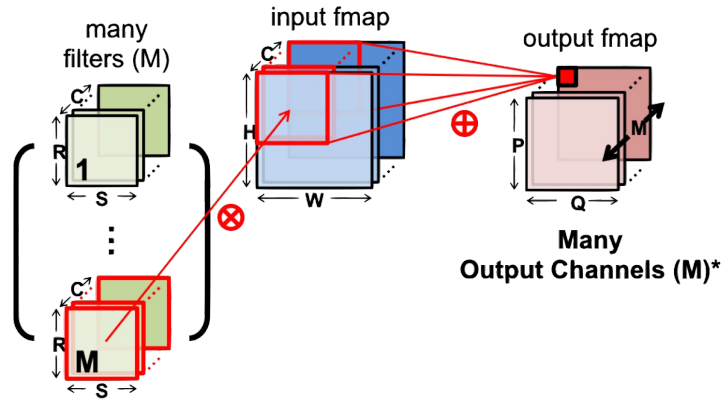


Weight Stationary for Convolution Layer

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

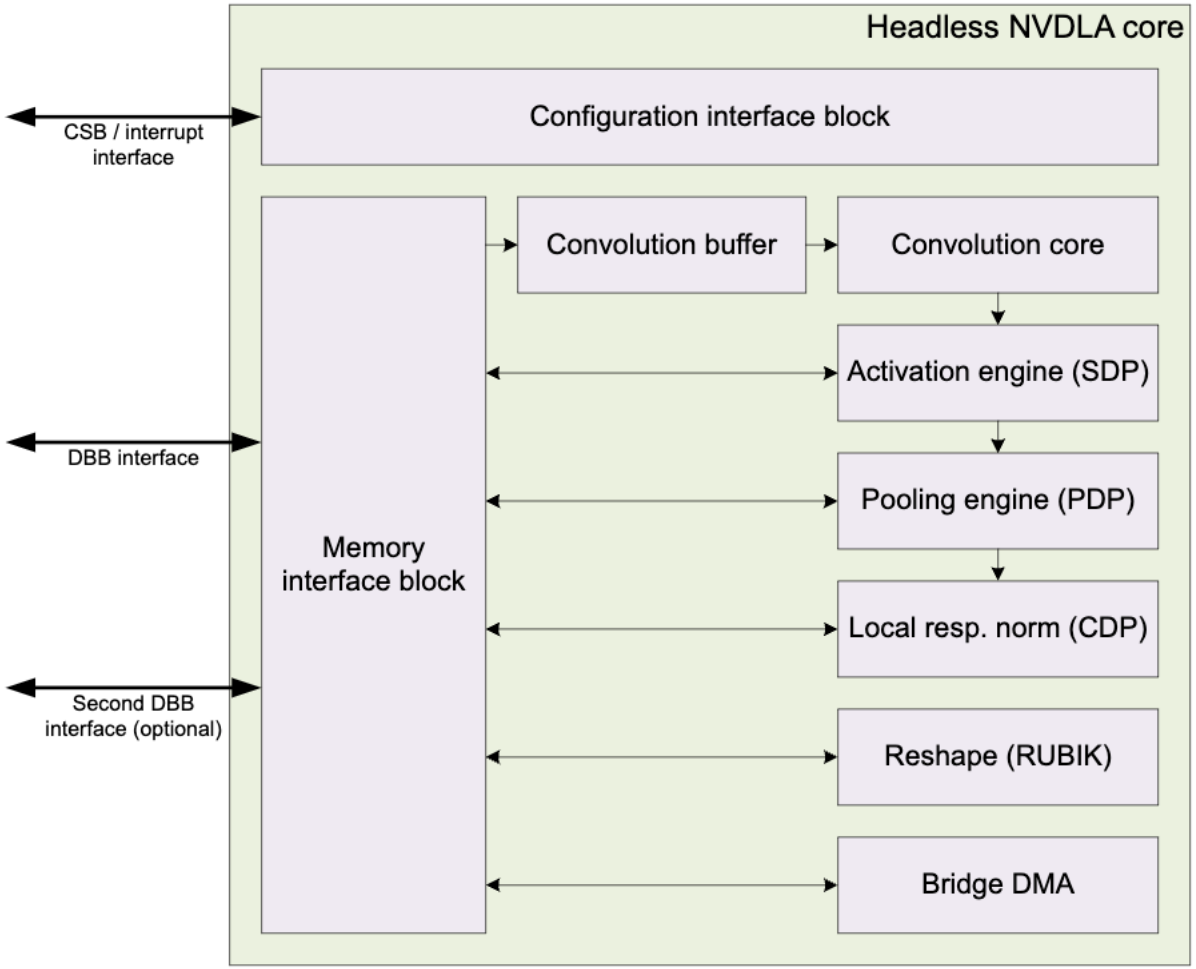
Assuming: stride of 1 and no batching

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



Traversal Order (fast to slow): p, q, r, s
Parallelize on C, M

WS Example: NVLDA

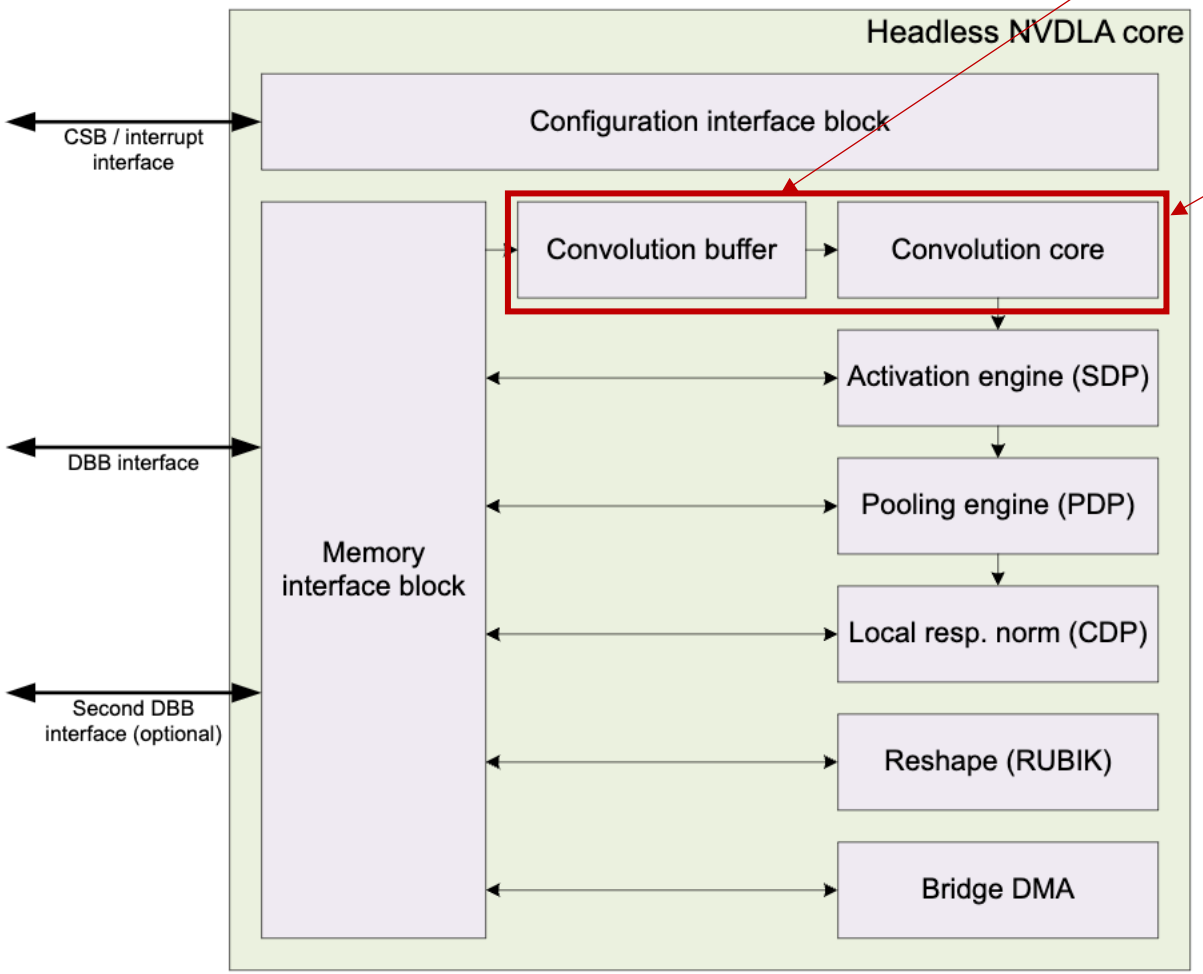


Headless NVDLA (NVIDIA Deep Learning Accelerator) core (Open Source)

https://github.com/nvdla/hw

WS Example: NVLDA

- Convolution buffer
  - Stores both weights and activations
- Optional compression support



M x C MACs:

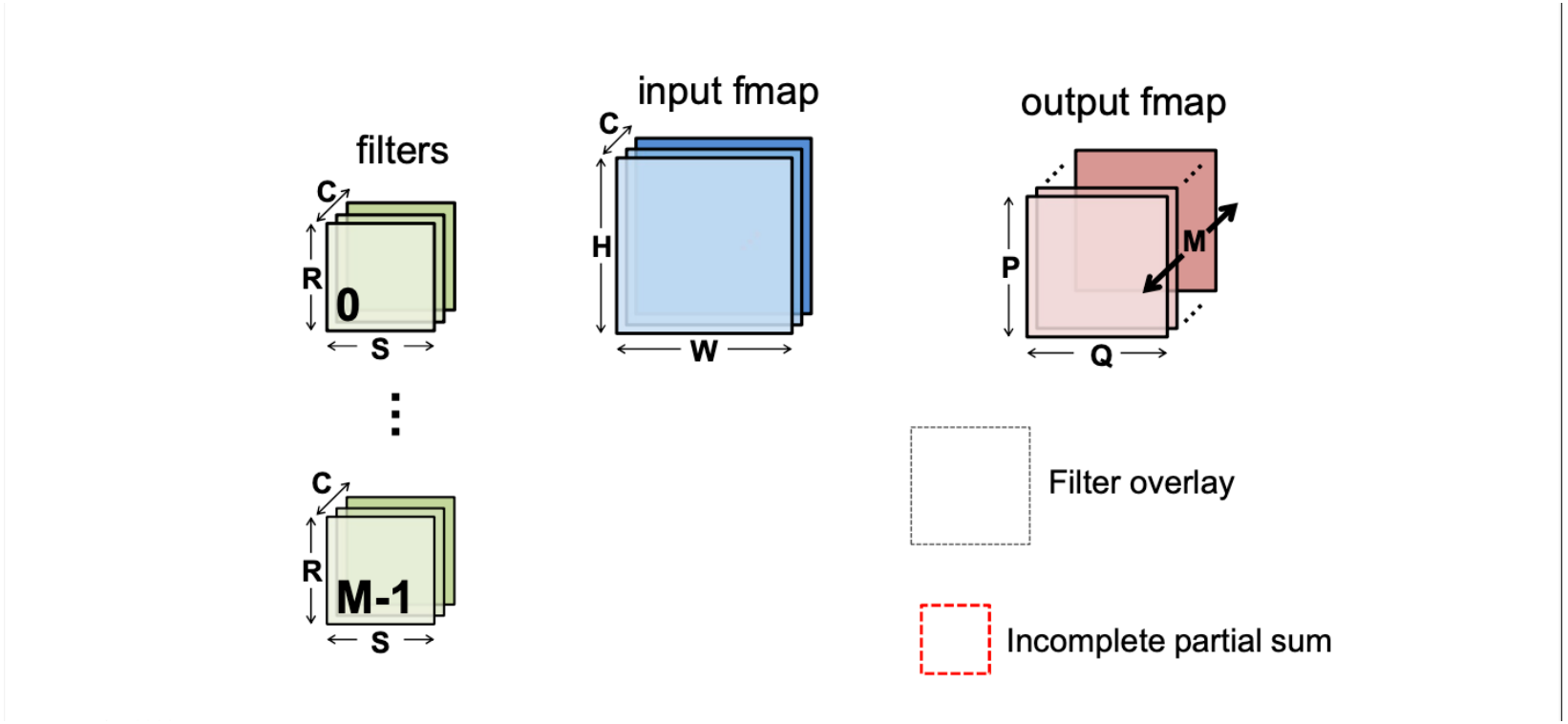
Range:
• C (16 to 128) • M(4to16)

Note: We use M for number of kernels (filters), while NVDLA uses K

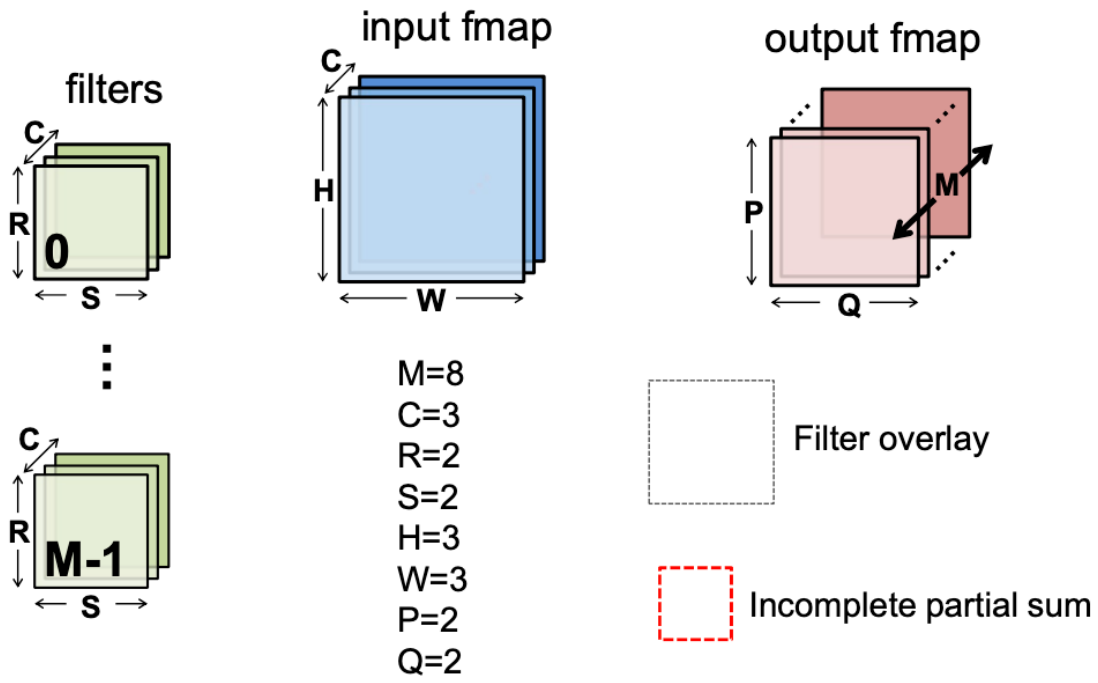
Headless NVDLA (NVIDIA Deep Learning Accelerator) core (Open Source)

Source: https://github.com/nvdla/hw
Descriptions adopted from Sze et al.

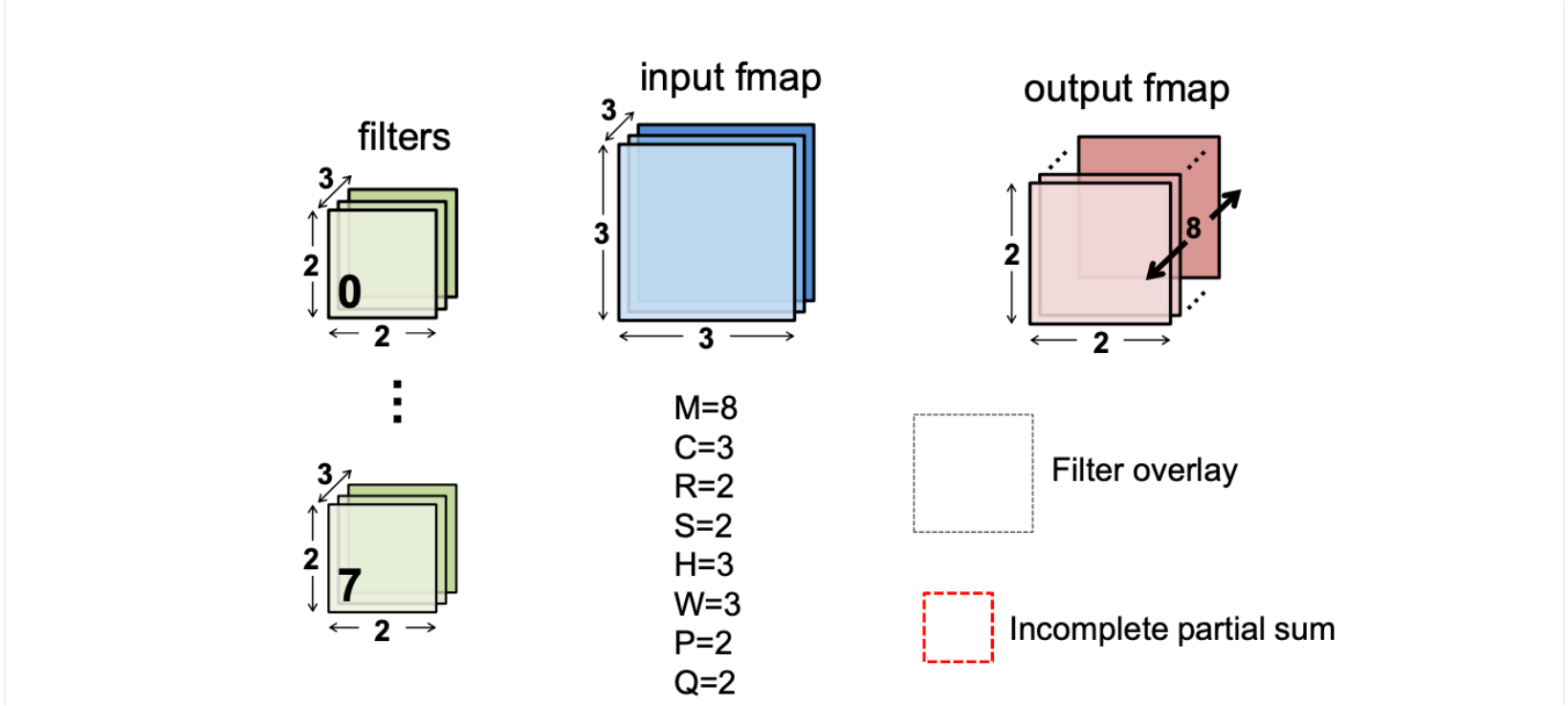
WS Example: Simplified NVLDA



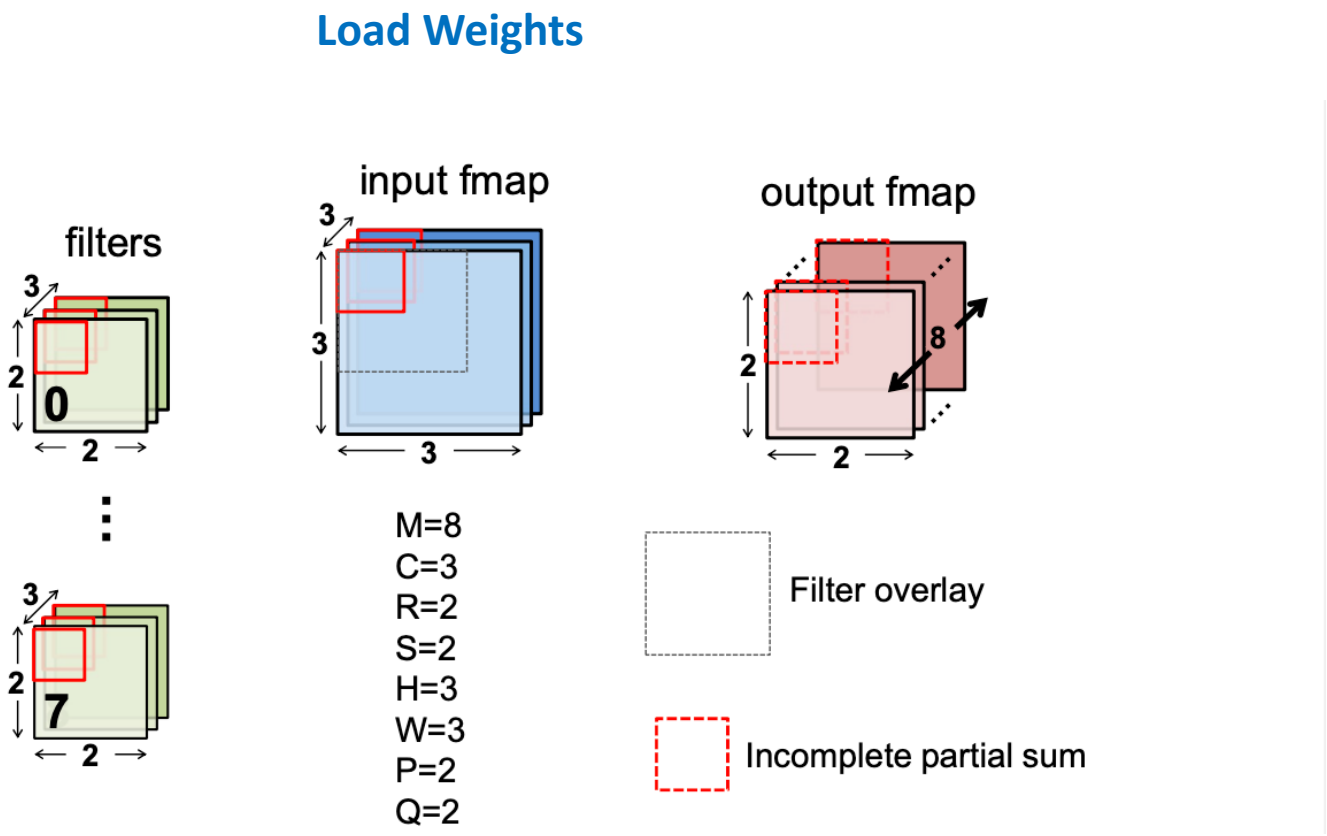
WS Example: Simplified NVLDA



WS Example: Simplified NVLDA

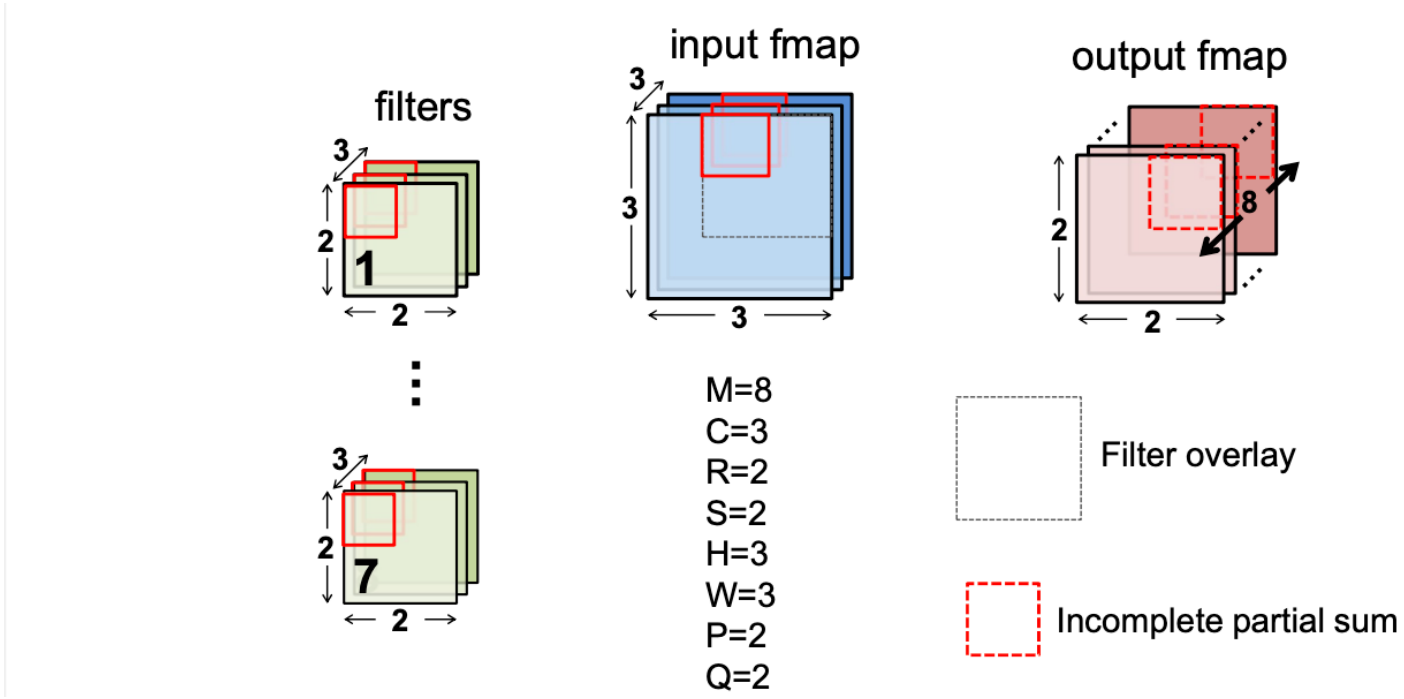


WS Example: Simplified NVLDA



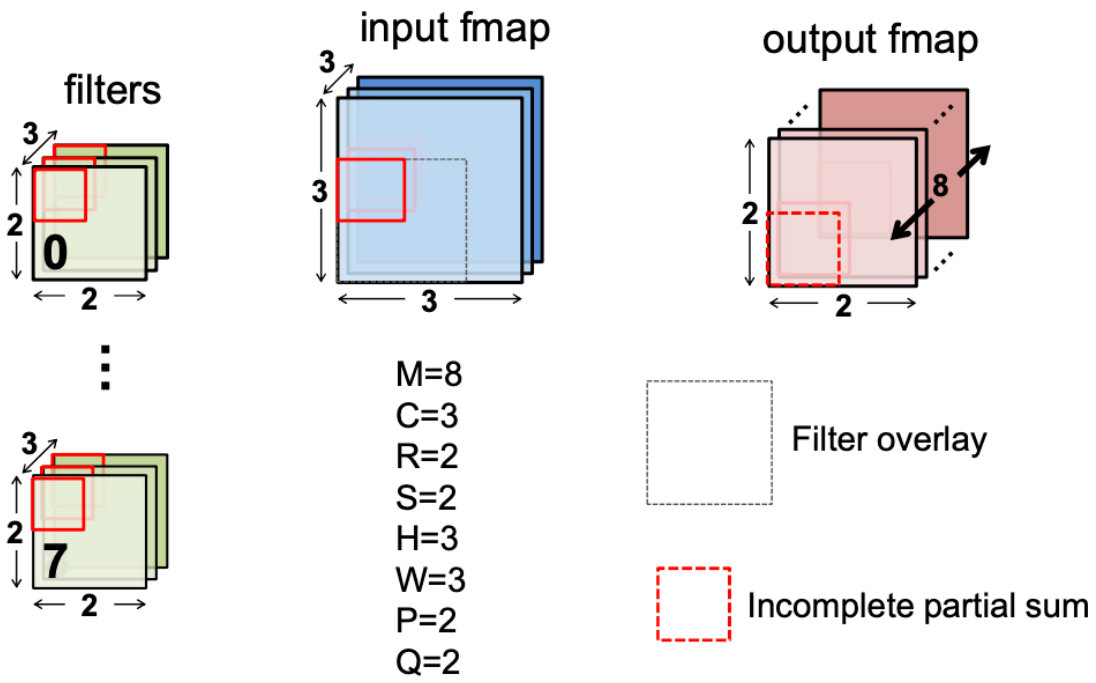
WS Example: Simplified NVLDA

Holding weights while Cycling through input and output feature maps



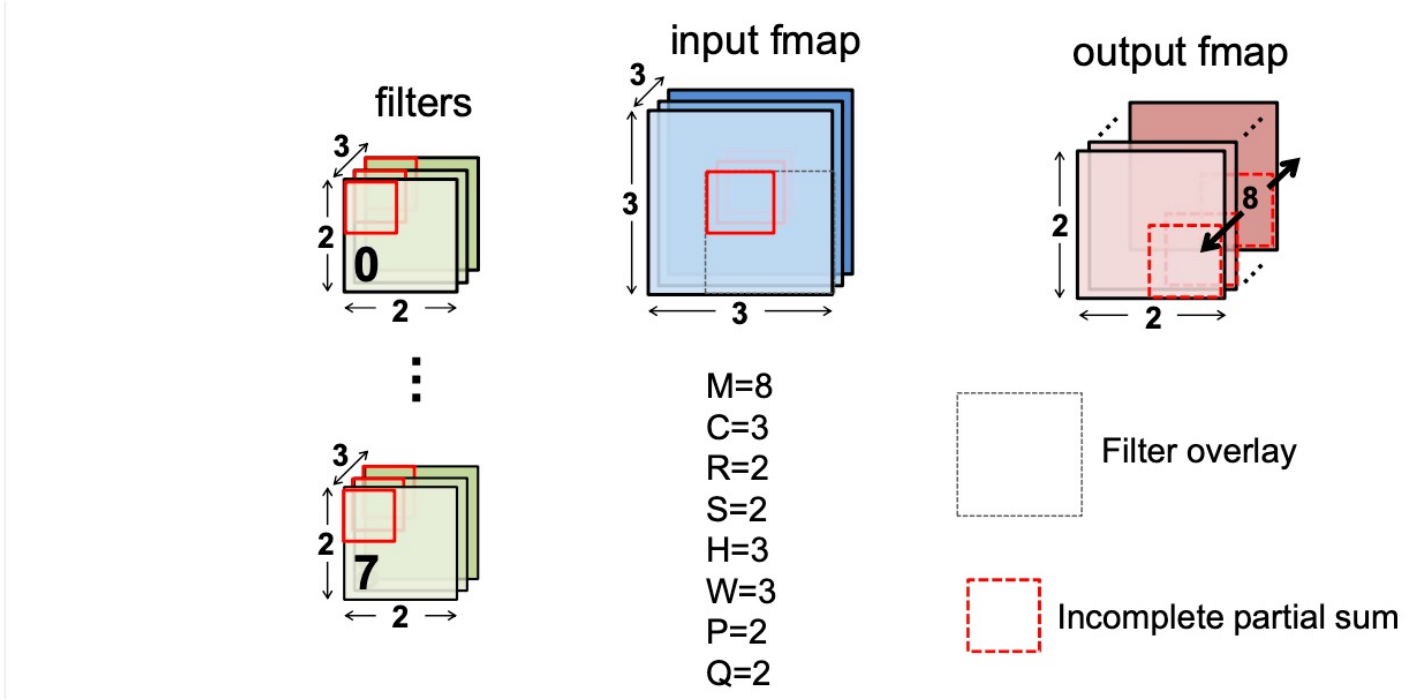
WS Example: Simplified NVLDA

Holding weights while Cycling through input and output feature maps



WS Example: Simplified NVLDA

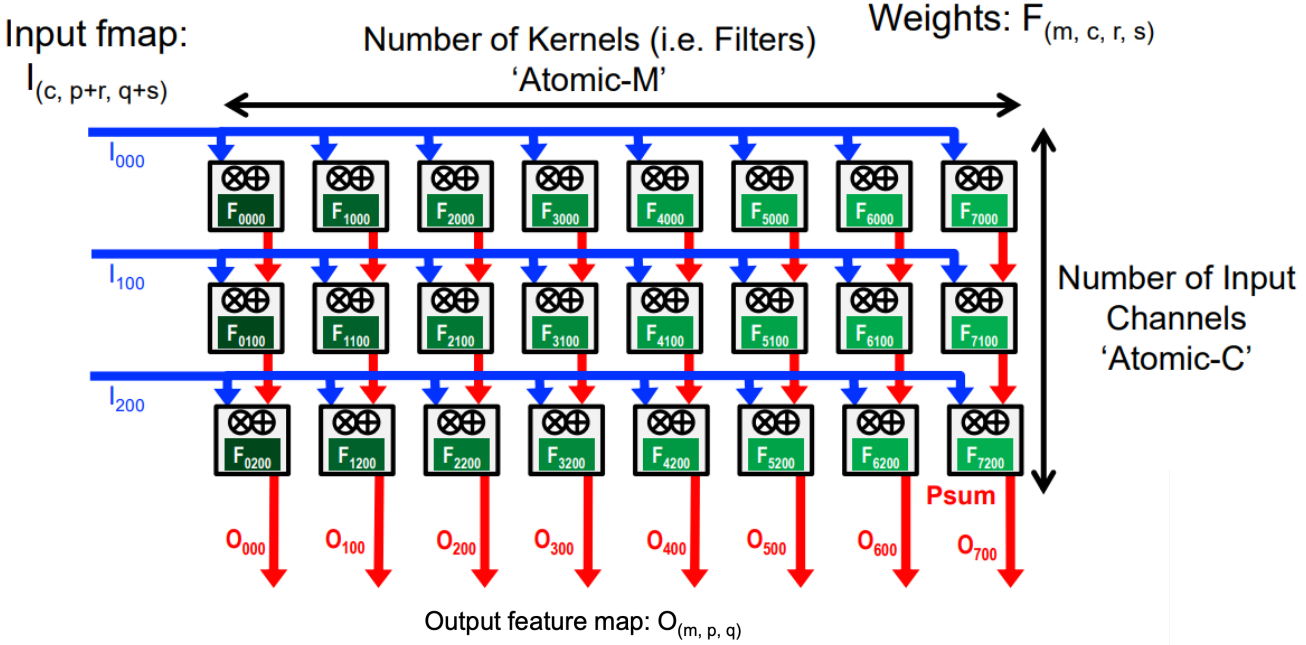
Holding weights while Cycling through input and output feature maps



WS Example: NVLDA (Simplified)

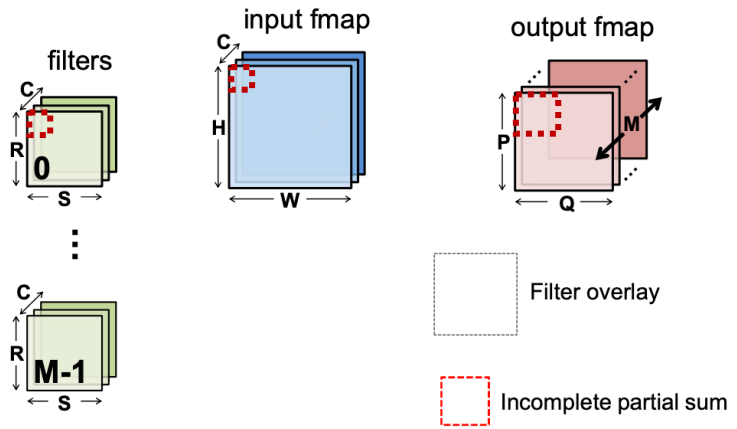
$$O_{m,p,q} = \sum_{c,r,s} I_{c,p+r,q+s} \times F_{m,c,r,s}$$

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



r	s	p	q
0	0	0	0

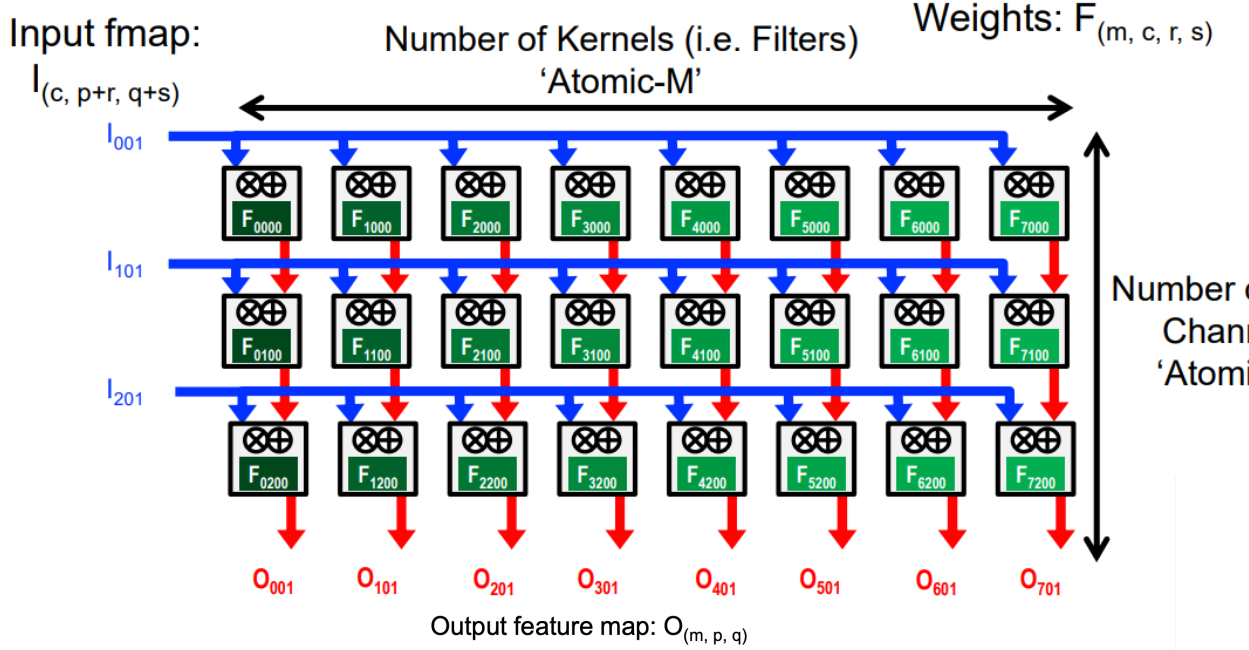
```
for r in range(R):
  for s in range(S):
    for p in range(P):
      for q in range(Q):
        parallel-for m in range(M):
          parallel-for c in range(C):
            output[m,p,q] += i[c,p+r,q+s] * f[m,c,r,s]
```



WS Example: NVLDA (Simplified)

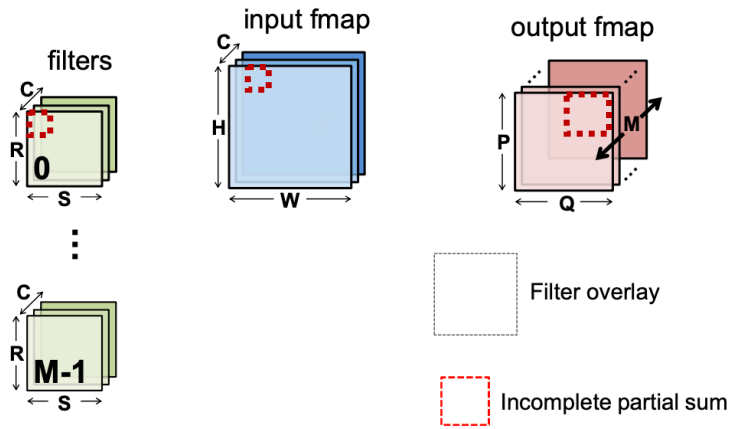
$$O_{m,p,q} = \sum_{c,r,s} I_{c,p+r,q+s} \times F_{m,c,r,s}$$

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



r	s	p	q
0	0	0	1

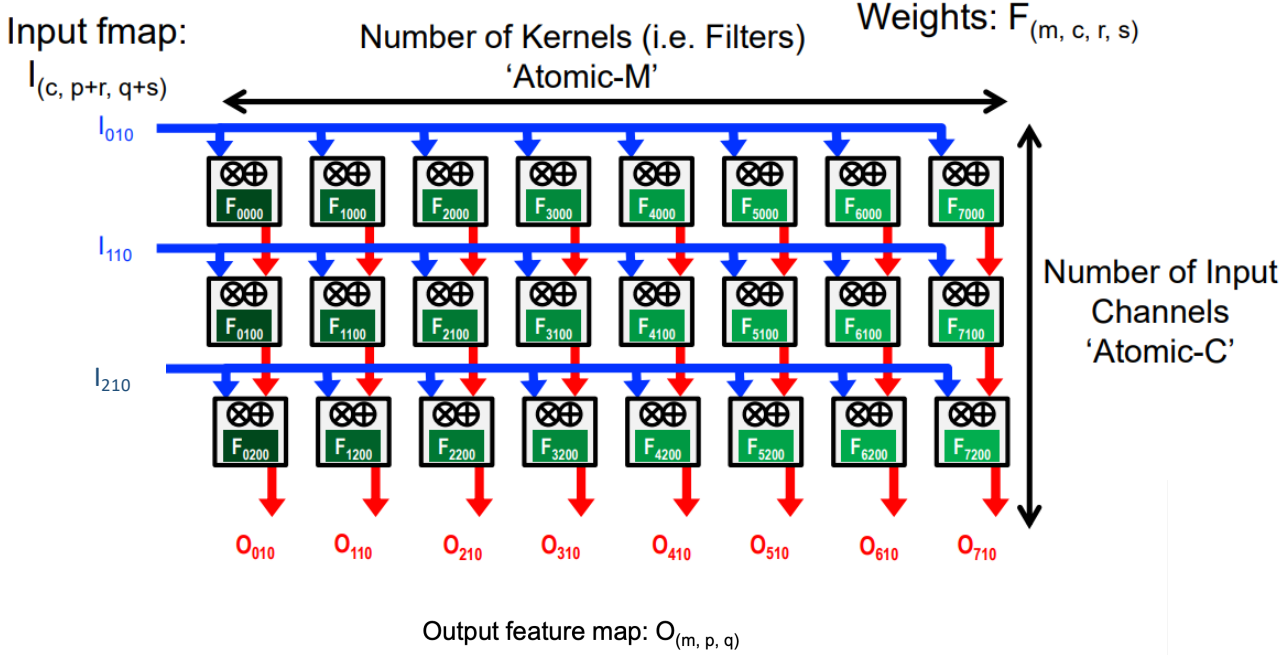
```
for r in range(R):
  for s in range(S):
    for p in range(P):
      for q in range(Q):
        parallel-for m in range(M):
          parallel-for c in range(C):
            output[m,p,q] += i[c,p+r,q+s] * f[m,c,r,s]
```



WS Example: NVLDA (Simplified)

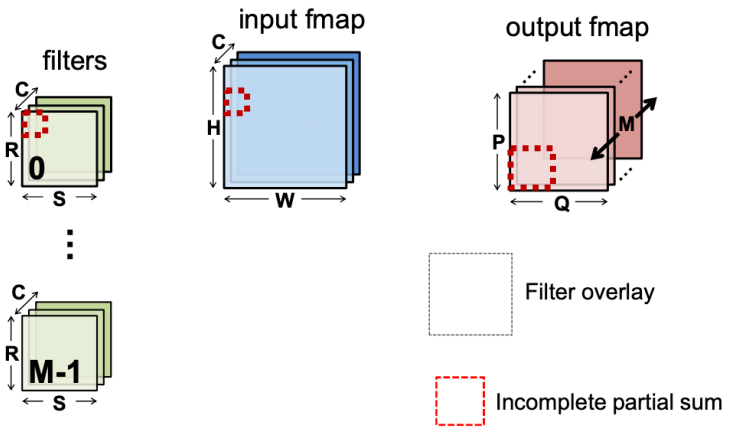
$$O_{m,p,q} = \sum_{c,r,s} I_{c,p+r,q+s} \times F_{m,c,r,s}$$

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



r	s	p	q
0	0	1	0

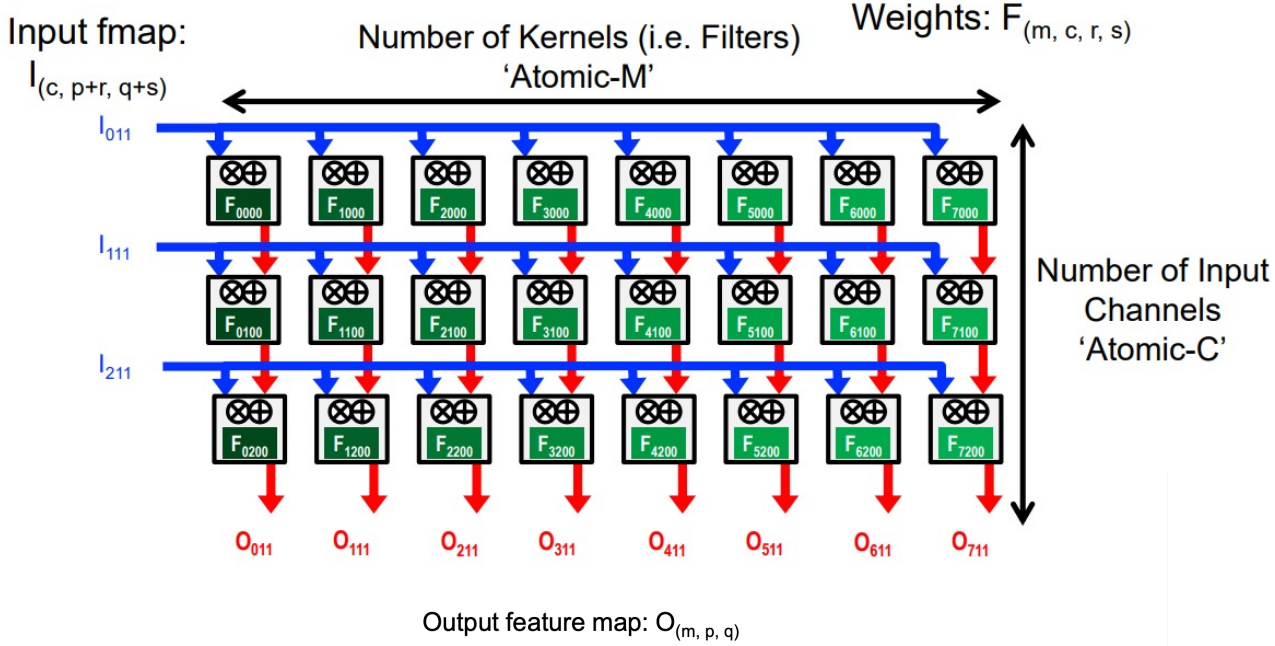
```
for r in range(R):
  for s in range(S):
    for p in range(P):
      for q in range(Q):
        parallel-for m in range(M):
          parallel-for c in range(C):
            output[m,p,q] += i[c,p+r,q+s] * f[m,c,r,s]
```



WS Example: NVLDA (Simplified)

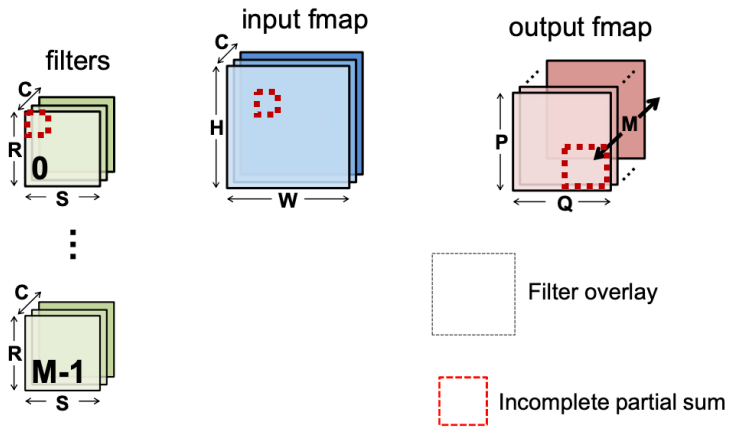
$$O_{m,p,q} = \sum_{c,r,s} I_{c,p+r,q+s} \times F_{m,c,r,s}$$

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



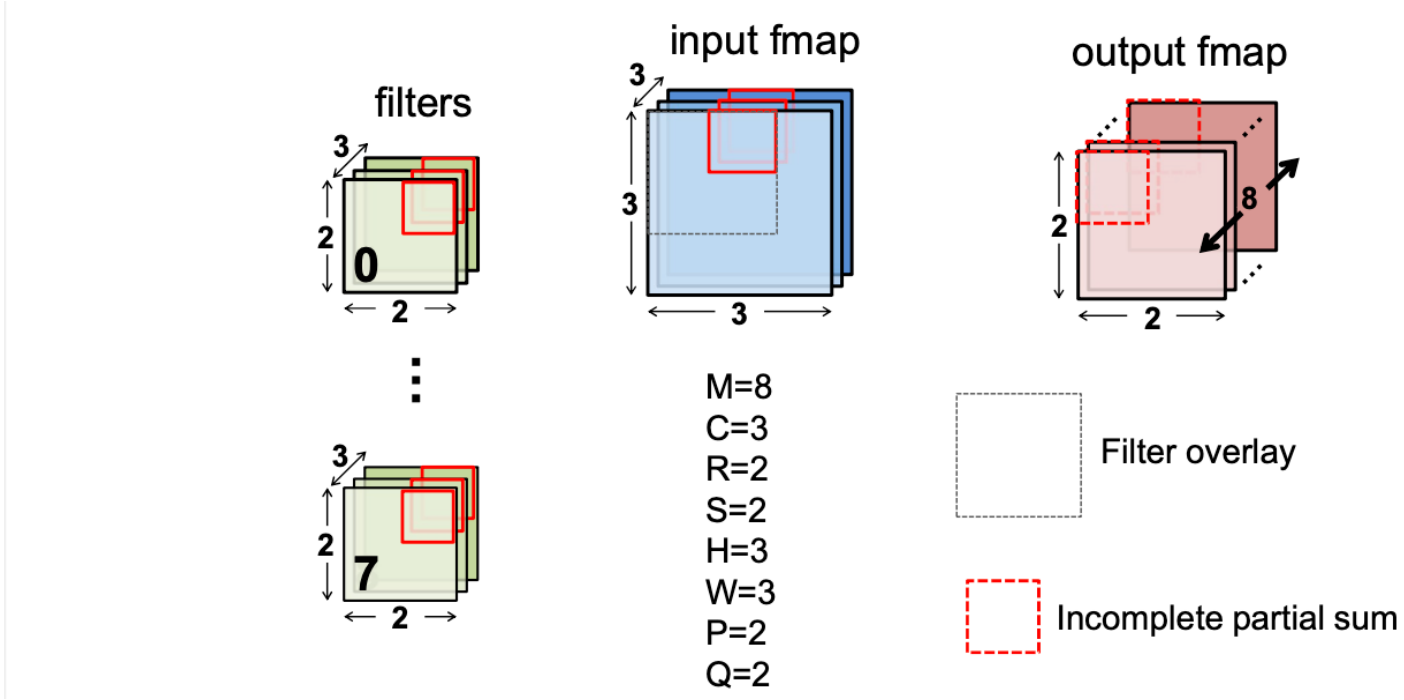
r	s	p	q
0	0	1	1

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



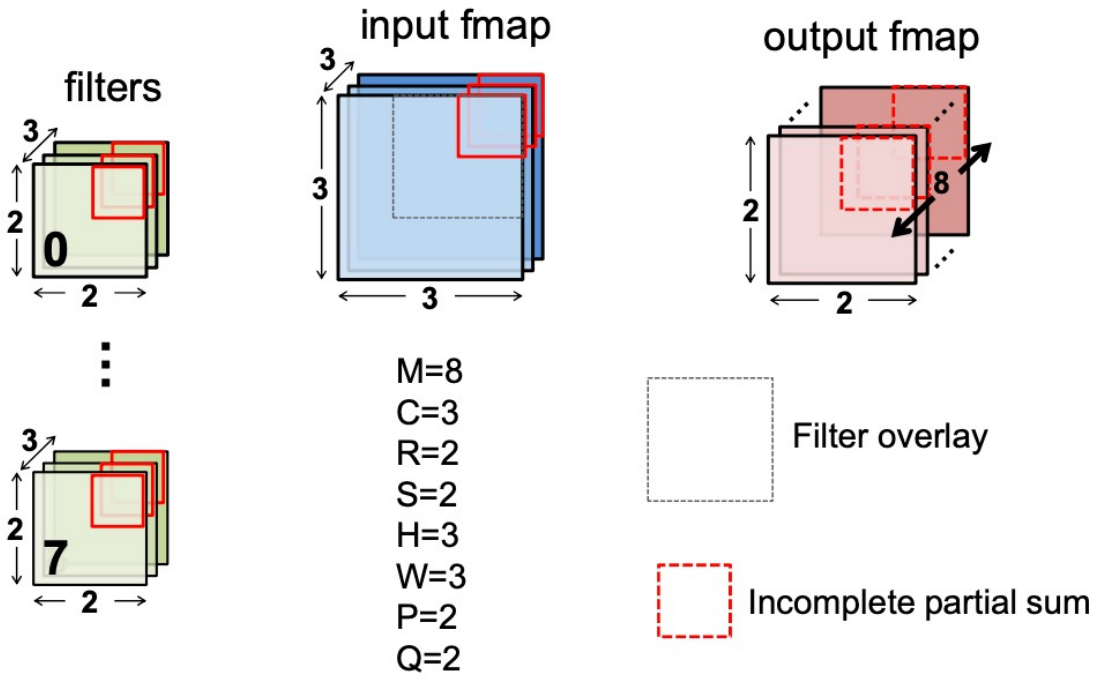
WS Example: Simplified NVLDA

Load new weights



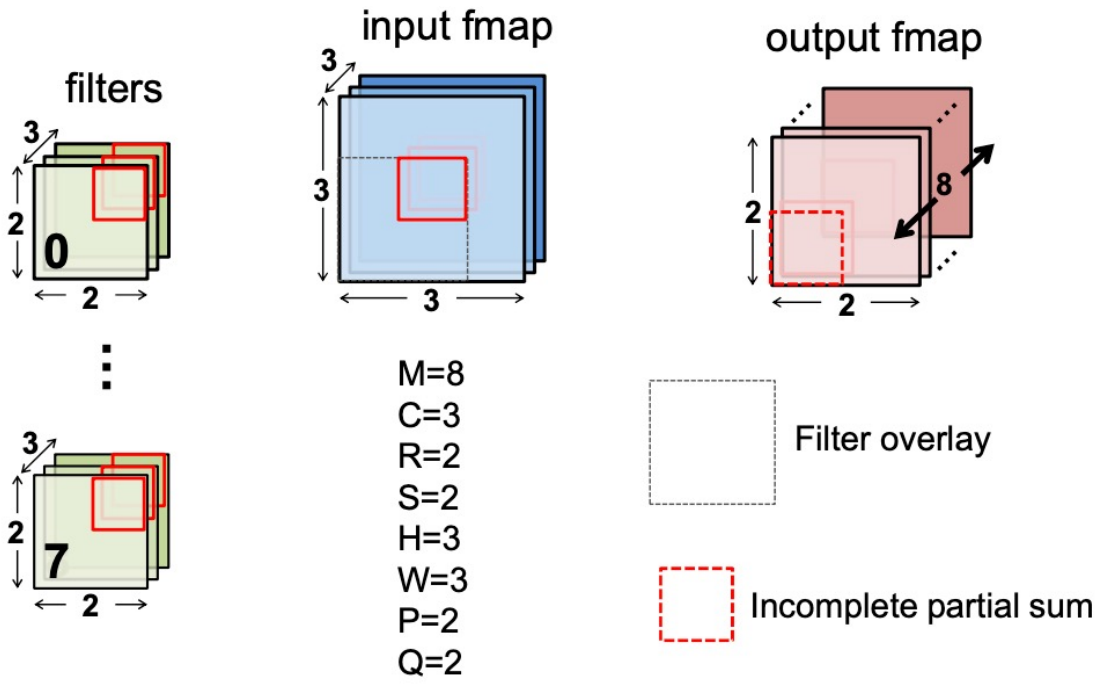
WS Example: Simplified NVLDA

Holding weights while Cycling through input and output feature maps



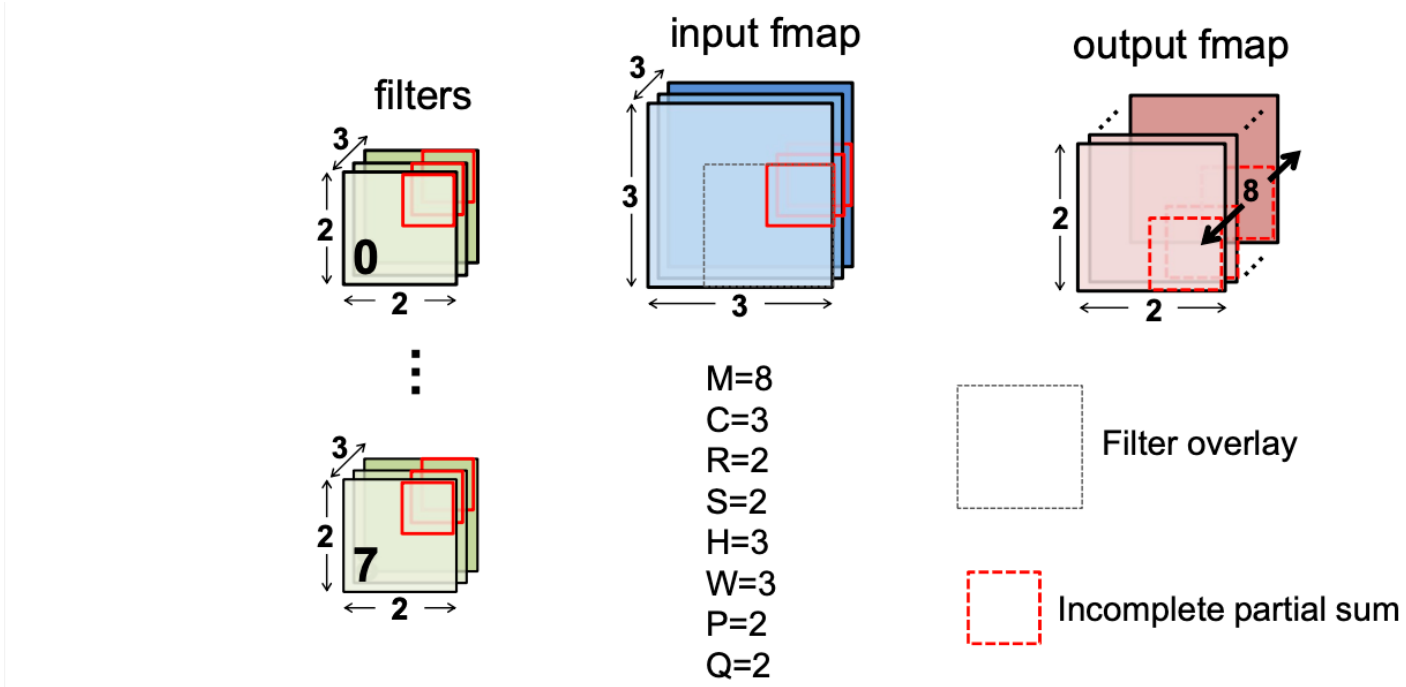
WS Example: Simplified NVLDA

Holding weights while Cycling through input and output feature maps



WS Example: Simplified NVLDA

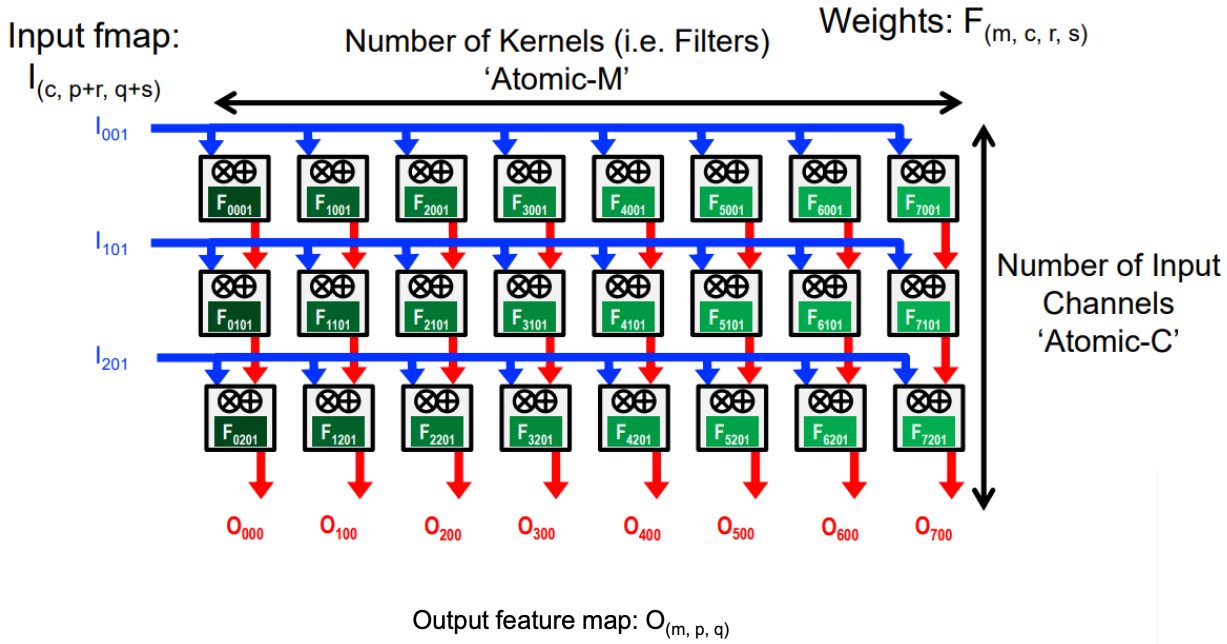
Holding weights while Cycling through input and output feature maps



WS Example: NVLDA (Simplified)

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

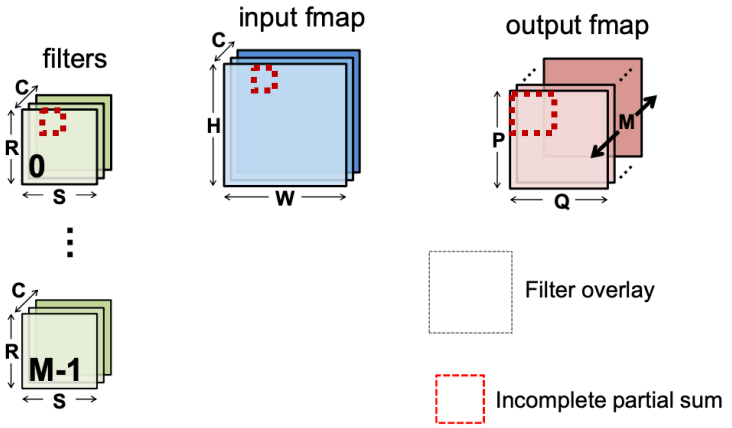
M=8
C=3
R=2
S=2
P=2
Q=2



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

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

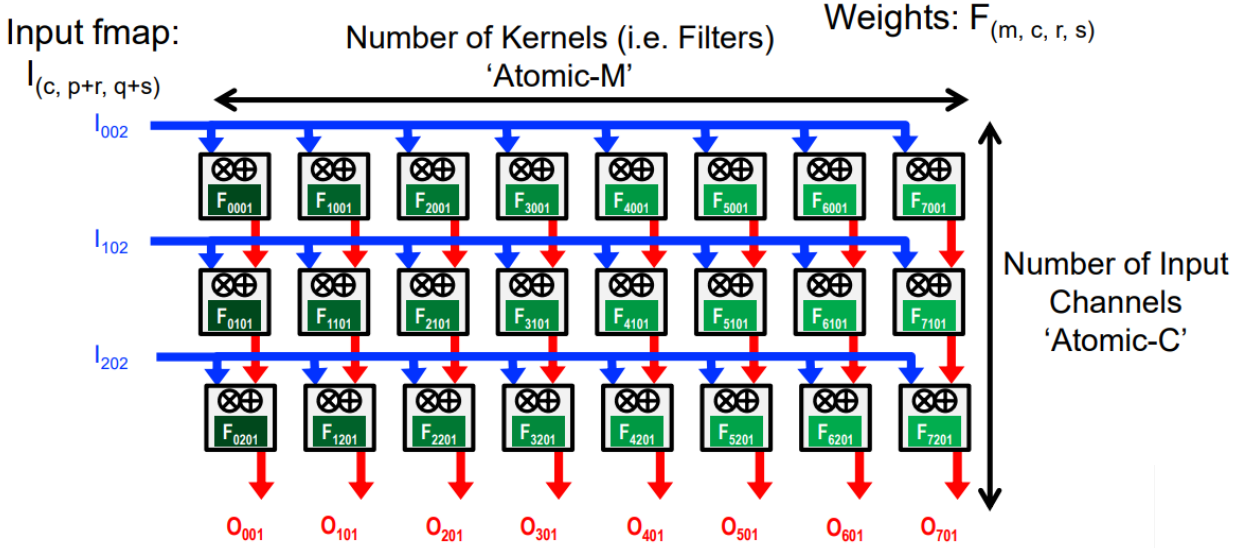
r	s	p	q
0	1	0	0



WS Example: NVLDA (Simplified)

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

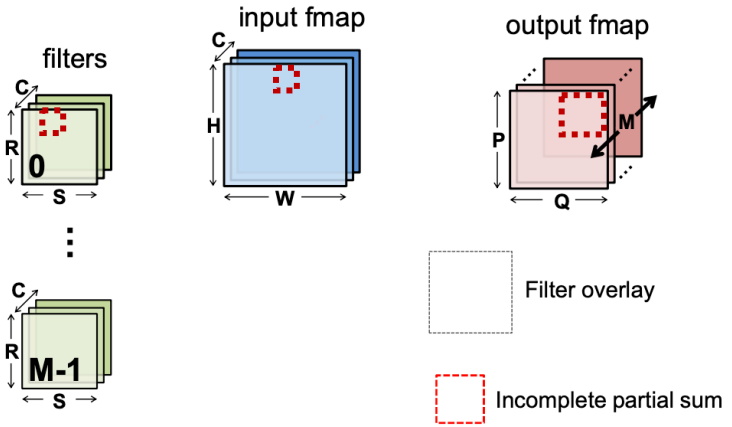
M=8
C=3
R=2
S=2
P=2
Q=2



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

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

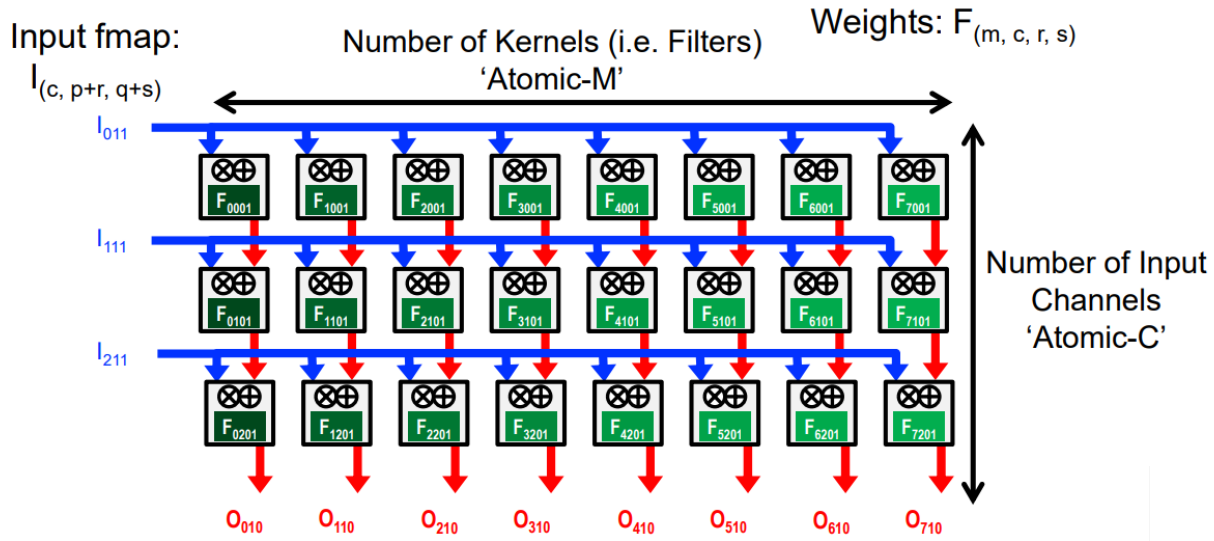
r	s	p	q
0	1	0	1



WS Example: NVLDA (Simplified)

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

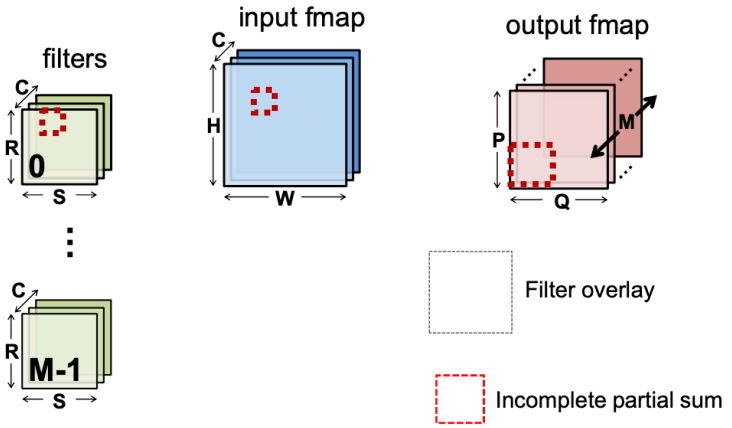
M=8
C=3
R=2
S=2
P=2
Q=2



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

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

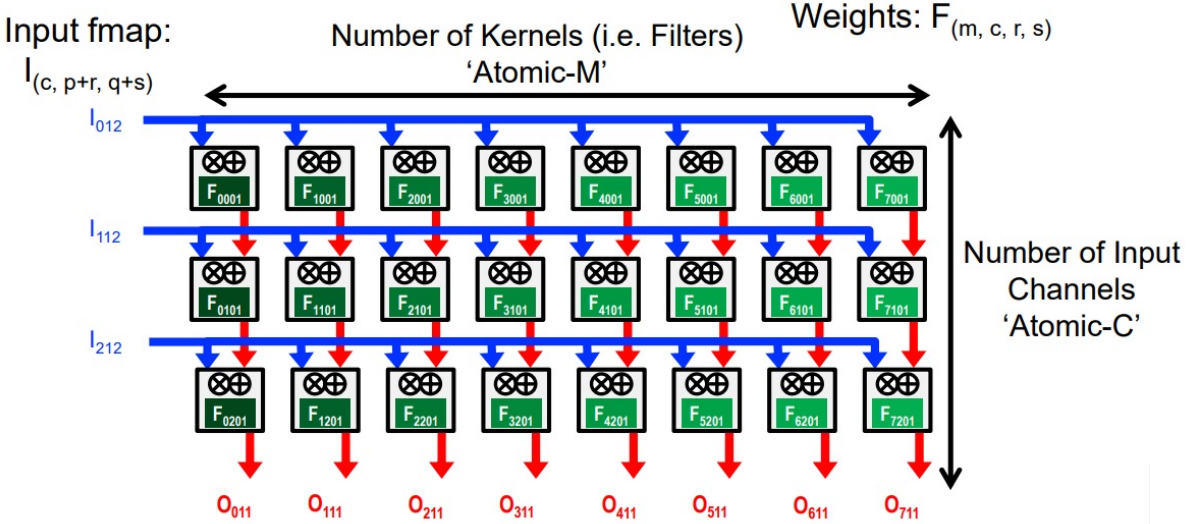
r	s	p	q
0	1	1	0



WS Example: NVLDA (Simplified)

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

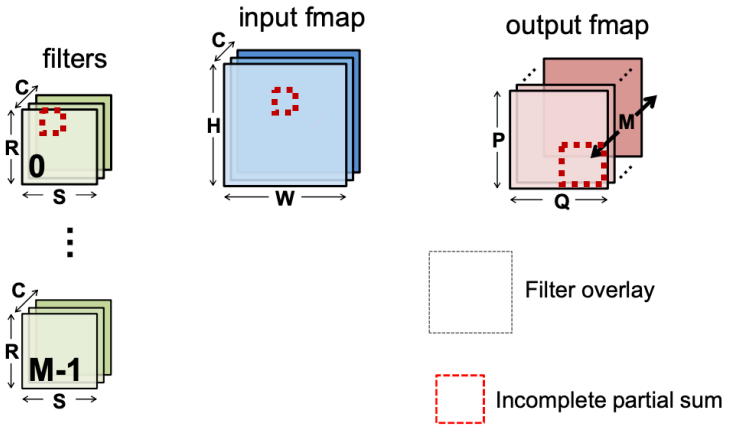
M=8
C=3
R=2
S=2
P=2
Q=2



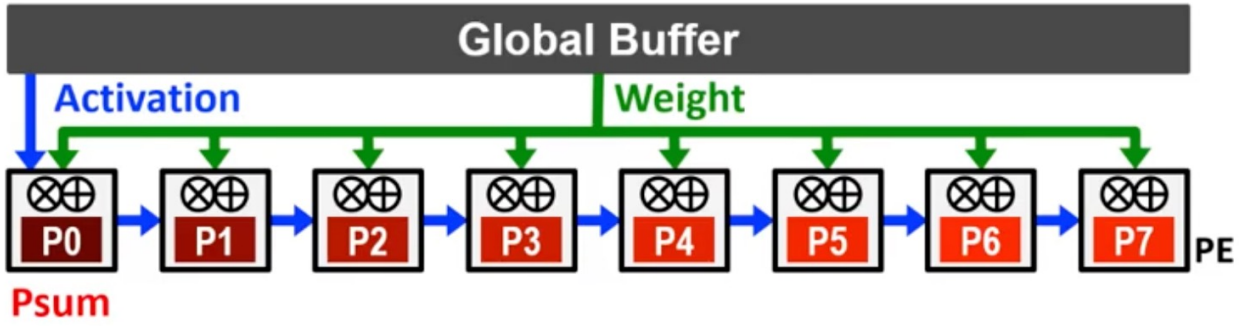
Output feature map: $O_{(m, p, q)}$

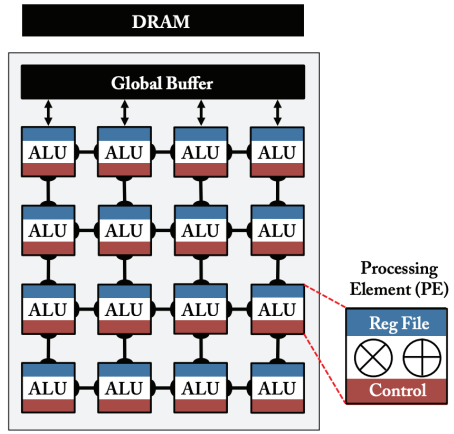
```
for r in range(R):
  for s in range(S):
    for p in range(P):
      for q in range(Q):
        parallel-for m in range(M):
          parallel-for c in range(C):
            output[m,p,q] += i[c,p+r,q+s] * f[m,c,r,s]
```

r	s	p	q
0	1	1	1



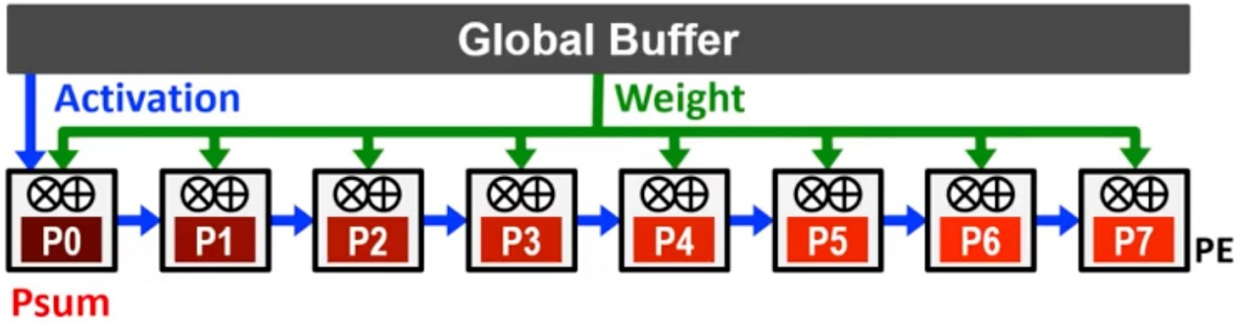
Output Stationary

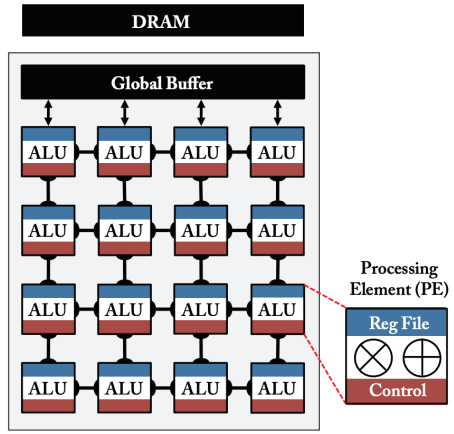




- **Goal:**
  - Minimize energy consumption by keeping the **partial output** results stationary.
  - Maximize **reuse of partial sums** (intermediate results) in each Processing Element (PE).

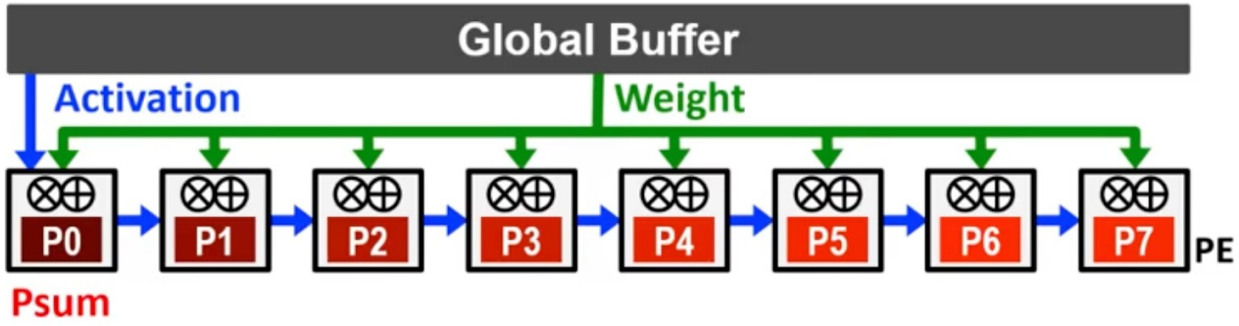
Output Stationary

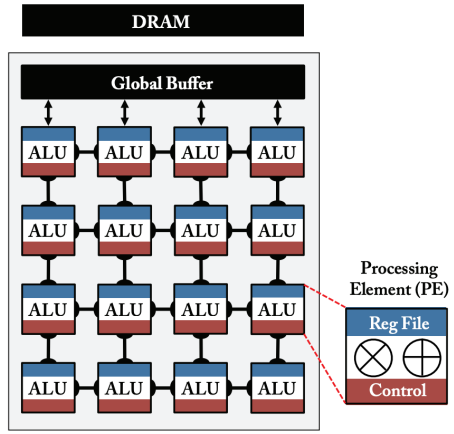




- **How it Works**:
  - Partial results (sums) of MAC operations remain in the same location (typically the RF of a PE).
  - Weights and input activations flow through the system and interact with these stationary partial sums, continually updating them.

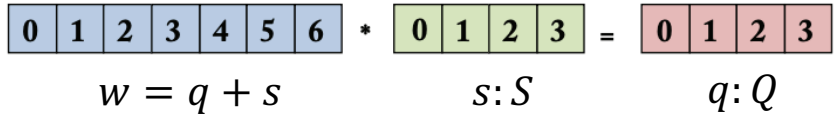
Output Stationary

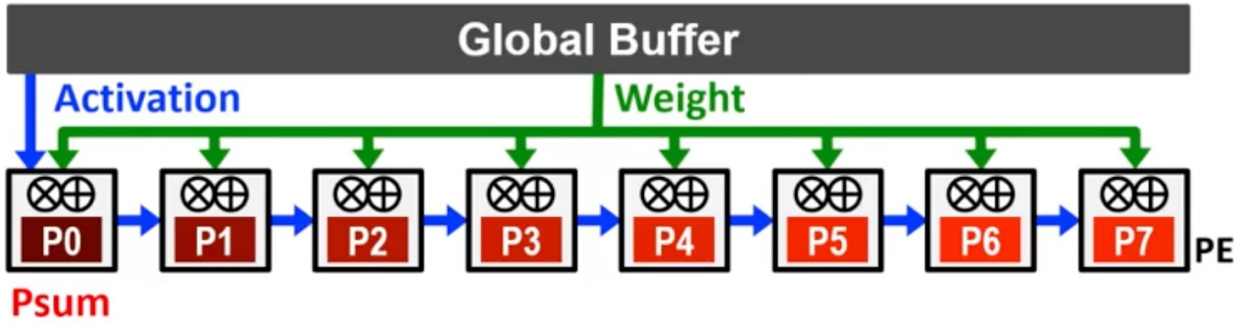


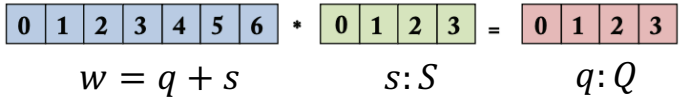


- **Data Movement**:
  - Weights and input feature map activations are moved and broadcast to the PEs.
  - Partial sums (output data) remain fixed in their PEs, reducing movement-related energy costs.
  - Once computation is complete, the consolidated results can be read from each PE.

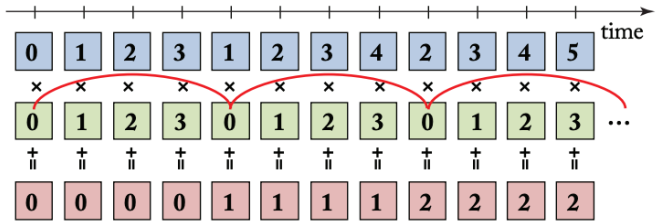
Output Stationary

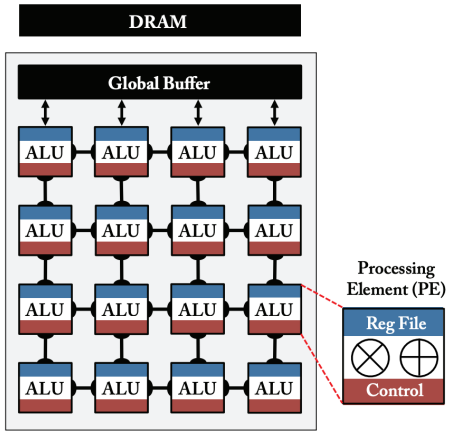




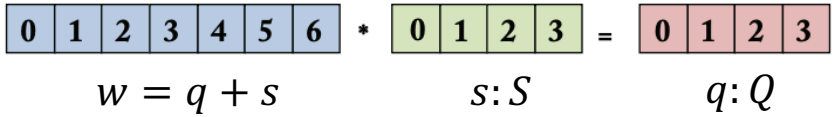


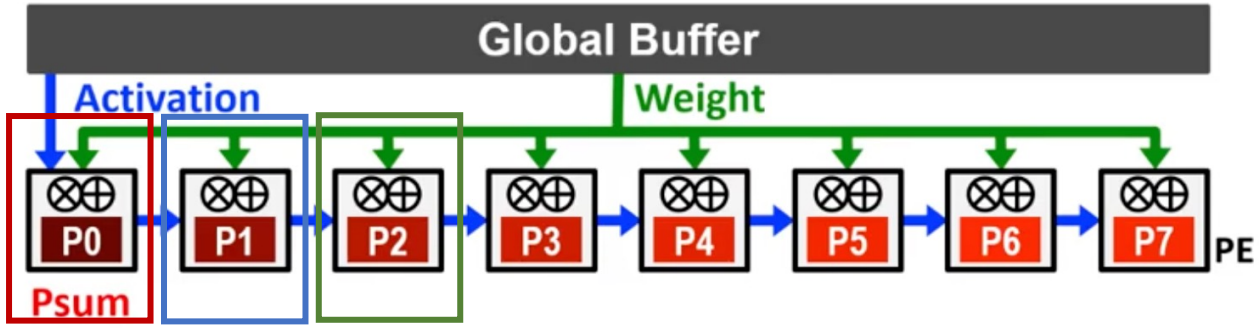

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

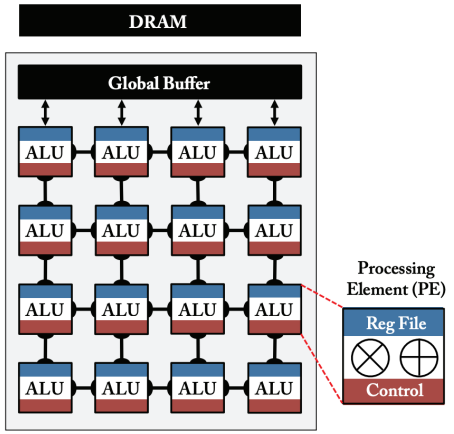




Output Stationary



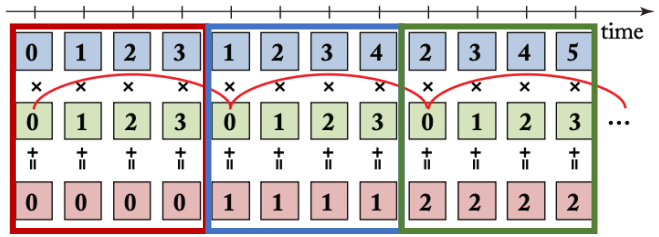




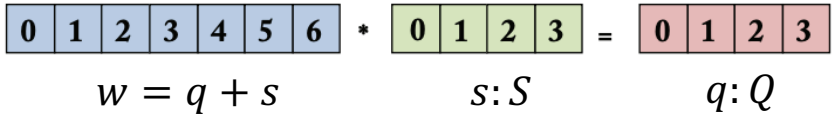
Broadcast **Weights**
Reuse **Activations** by passing them sequentially

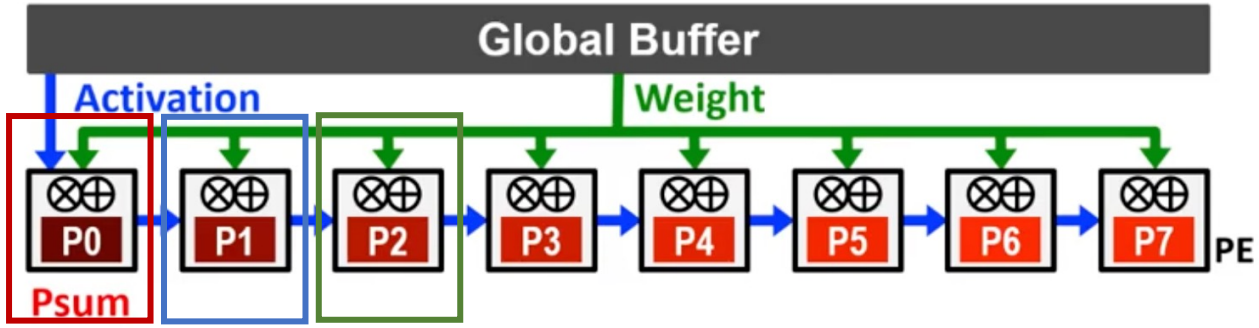
```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

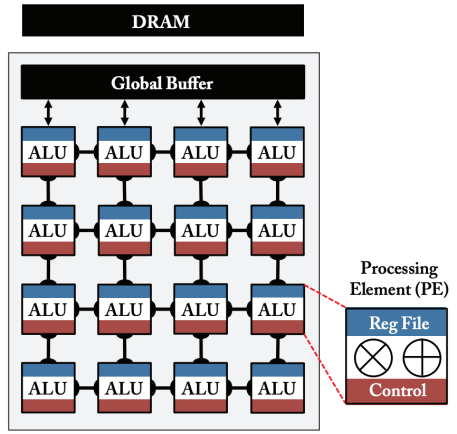
$$O_q = \sum_s I_{q+s} \times F_s$$



Output Stationary



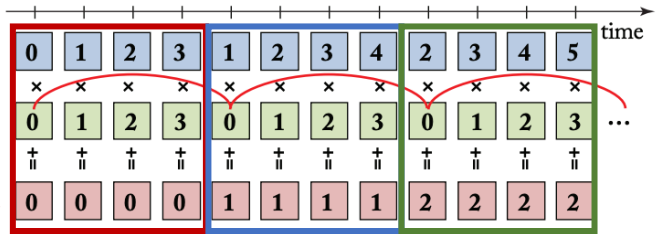




Broadcast **Weights**
Reuse **Activations** by passing them sequentially

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

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



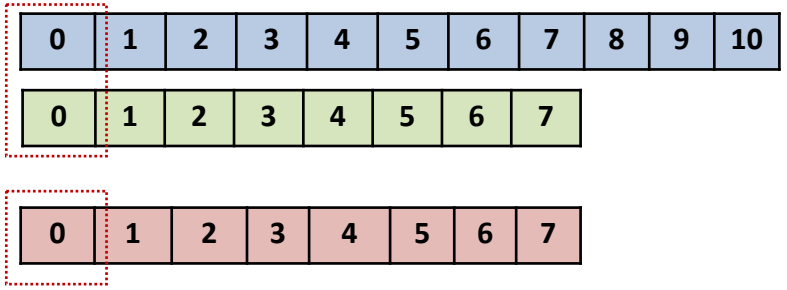
This can be in register file

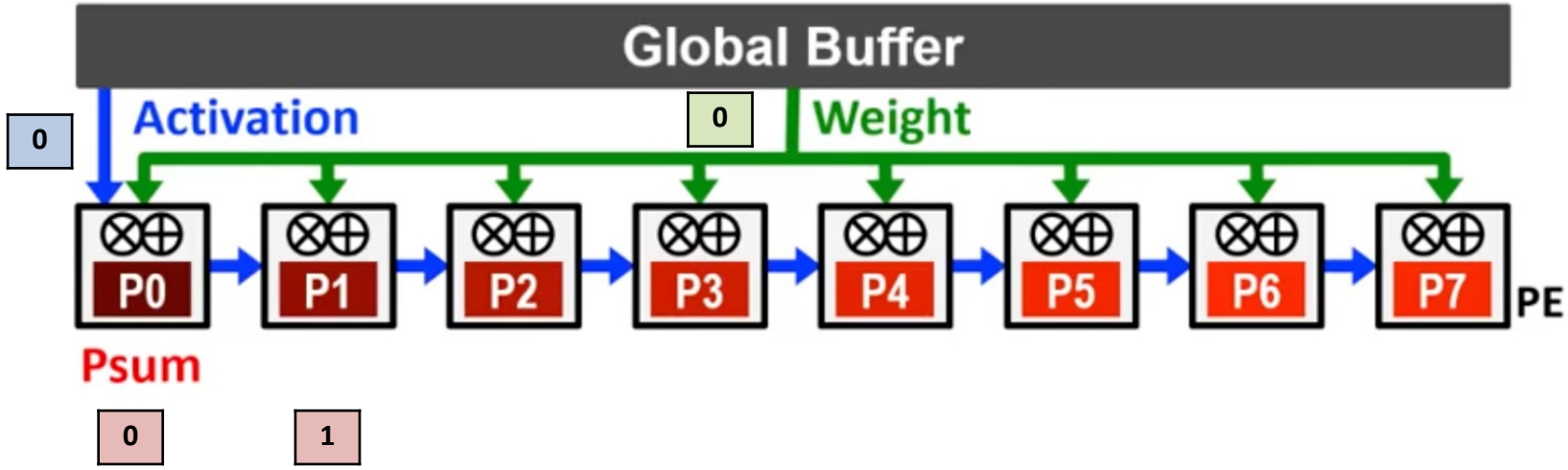
Traversal Order (fastest to slowest):

1. s
2. q

Output Stationary

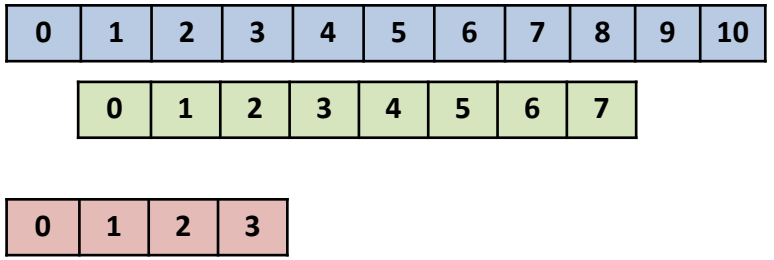
Output 0





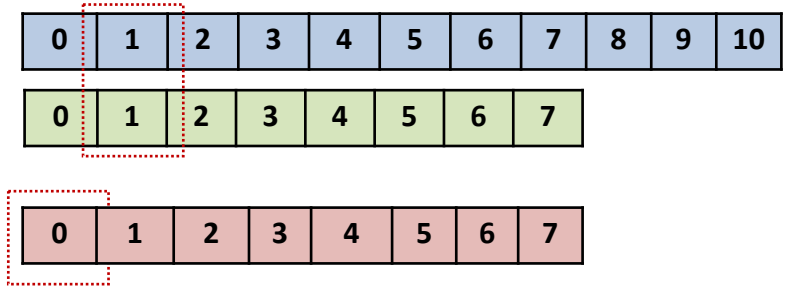
Focus on Output 0, 1

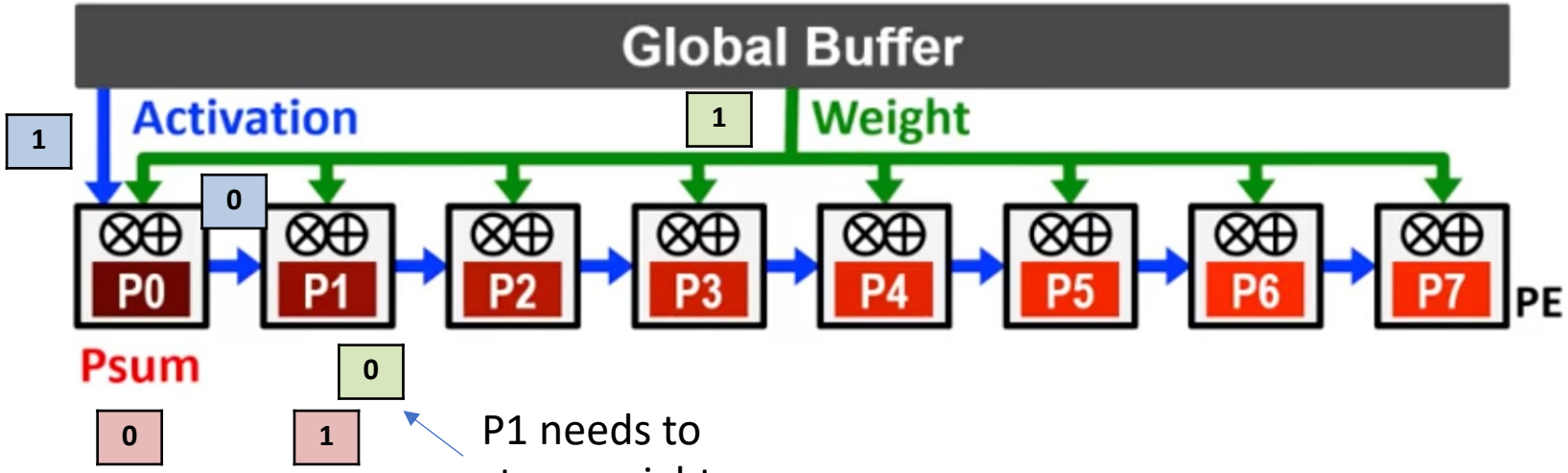
Output 1



Output Stationary

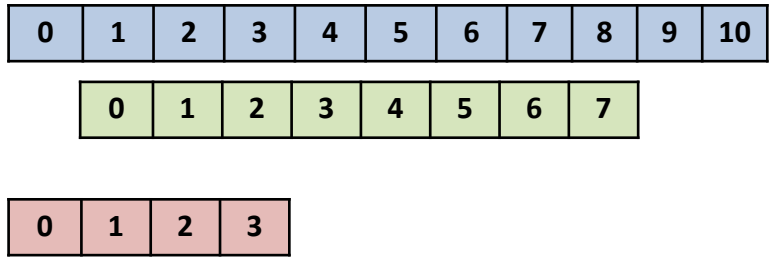
Output 0



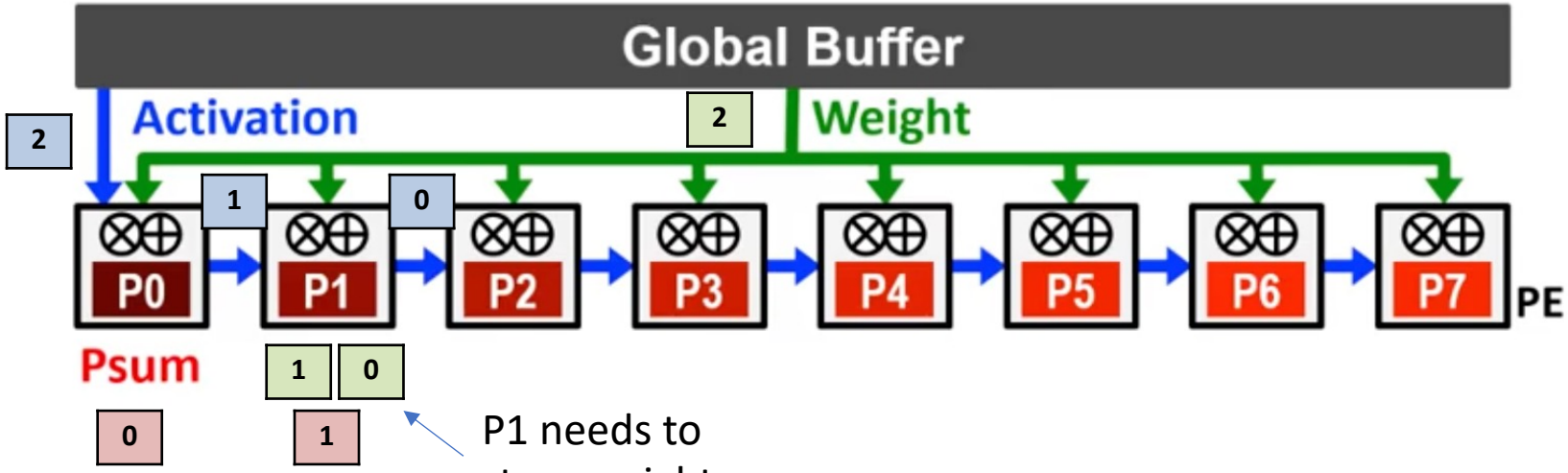


Focus on Output 0, 1

Output 1

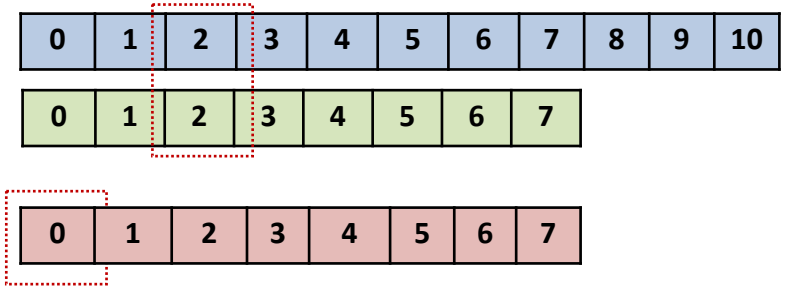


Output Stationary

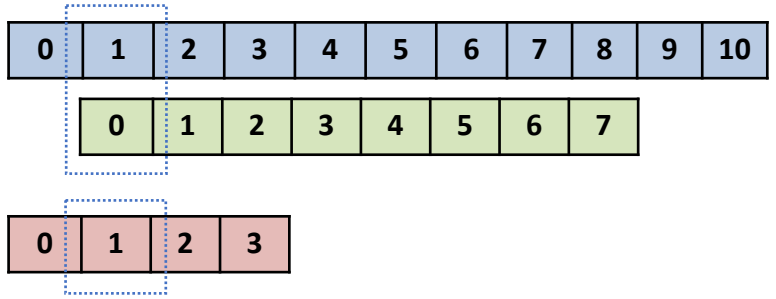


Focus on Output 0, 1

Output 0

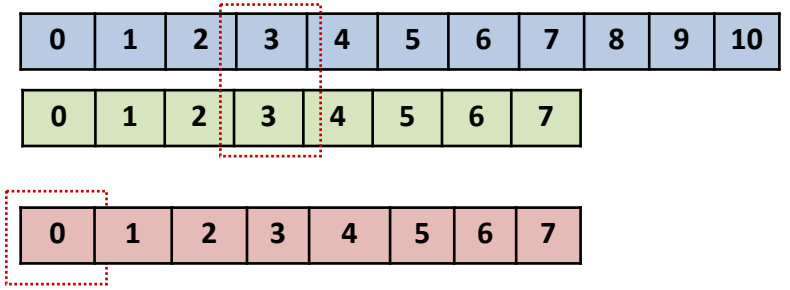


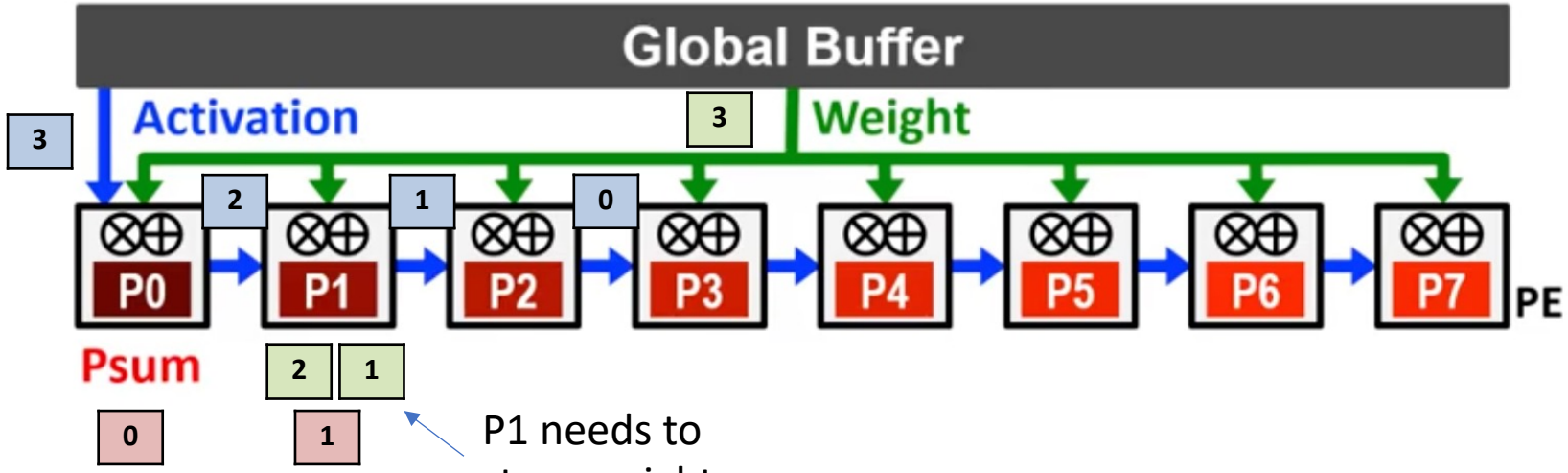
Output 1



Output Stationary

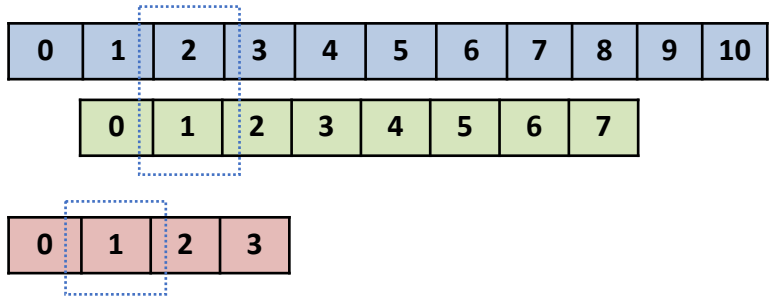
Output 0





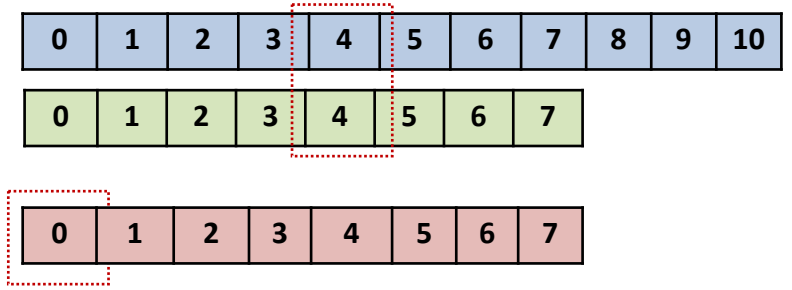
Focus on Output 0, 1

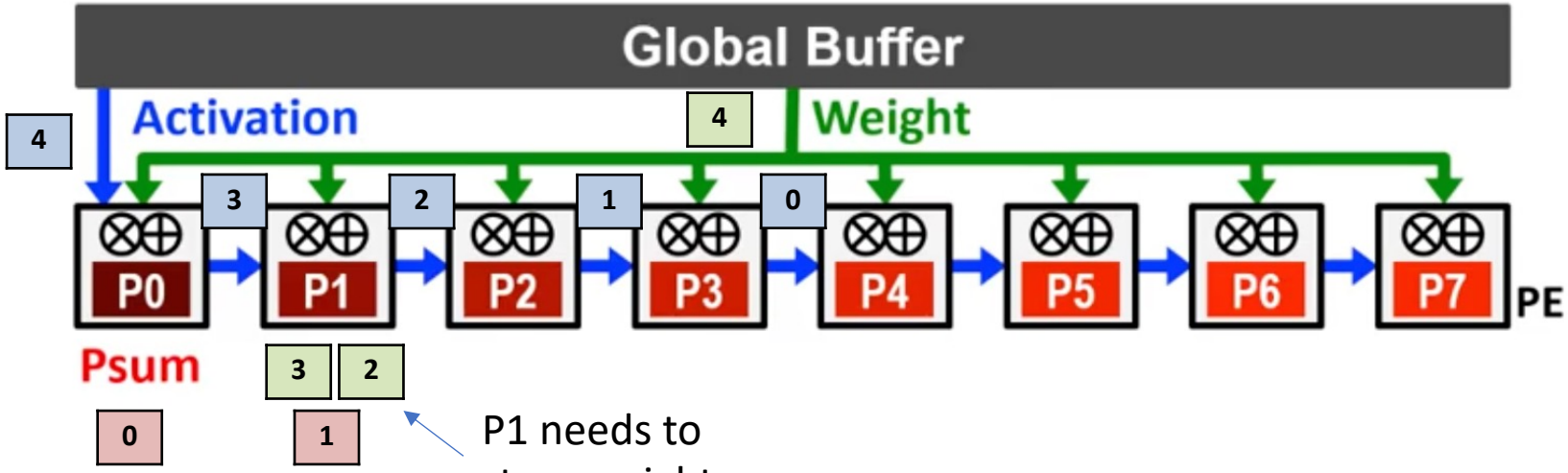
Output 1



Output Stationary

Output 0

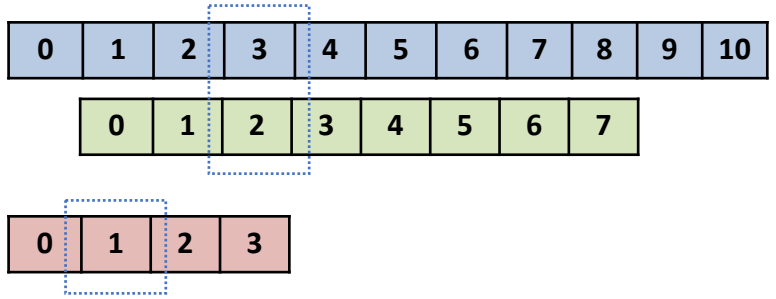




P1 needs to store weights

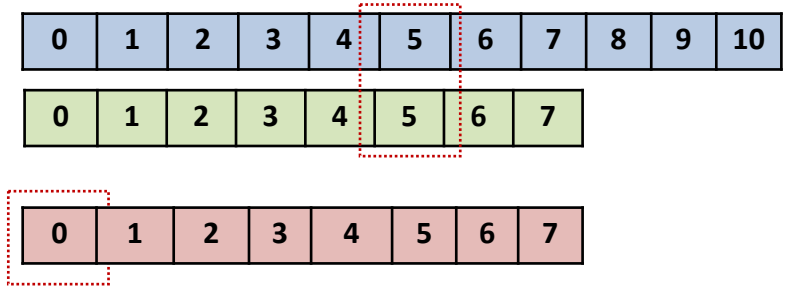
Focus on Output 0, 1

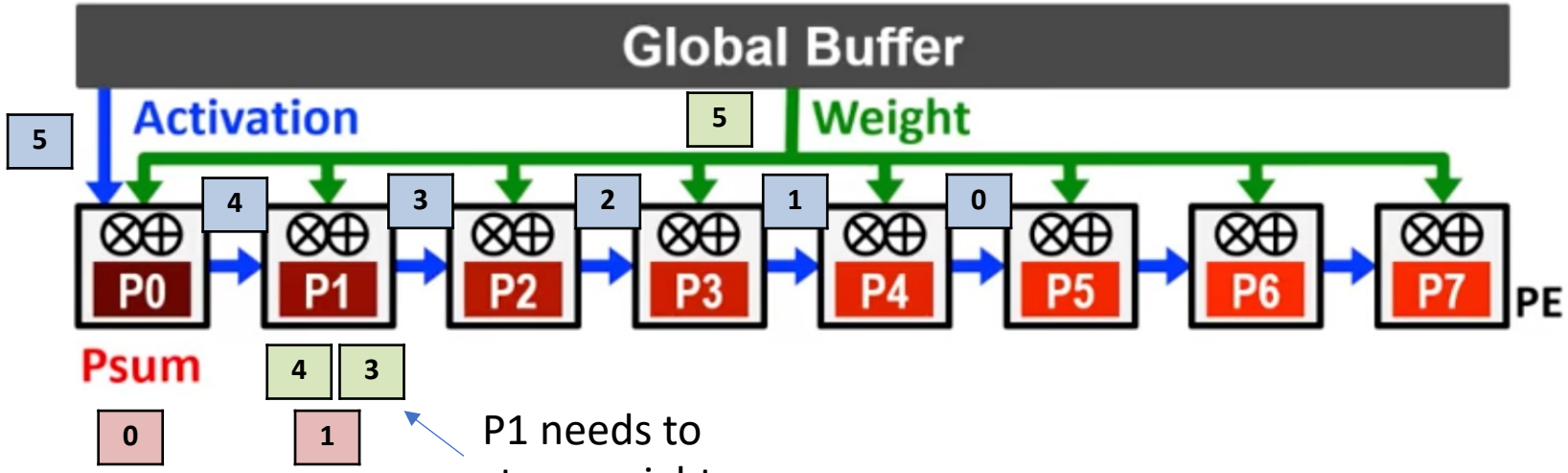
Output 1



Output Stationary

Output 0

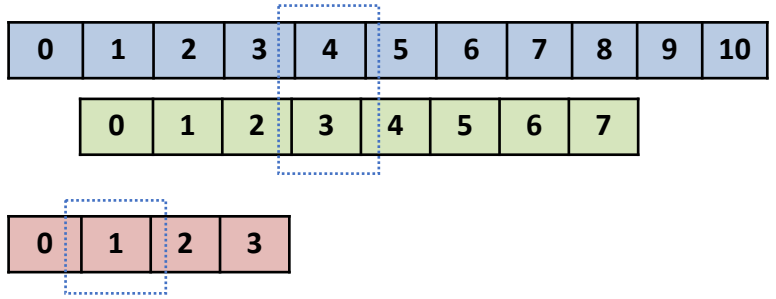




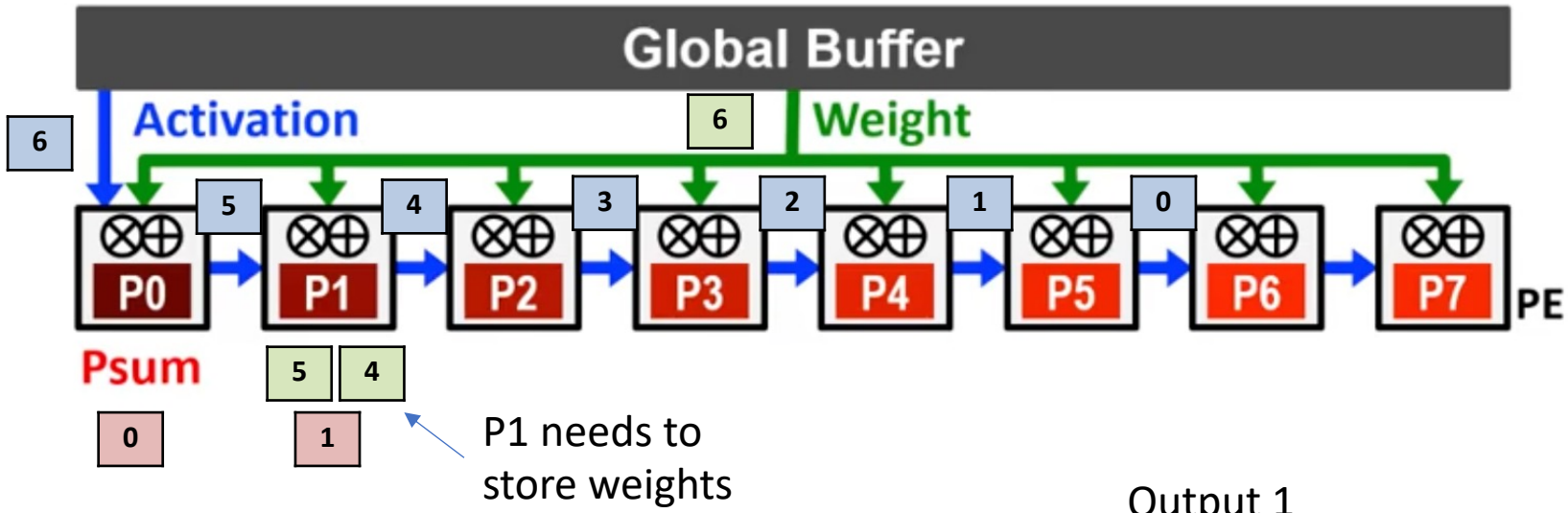
P1 needs to store weights

Focus on Output 0, 1

Output 1

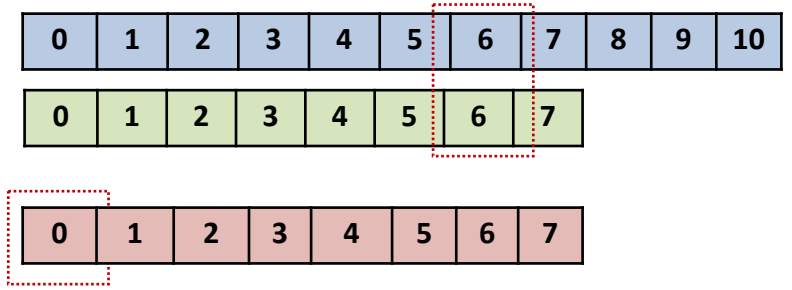


Output Stationary

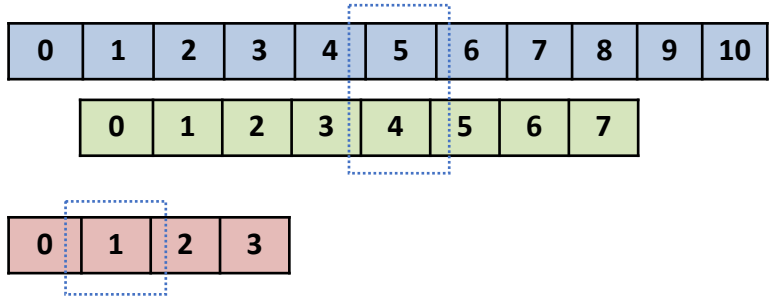


Focus on Output 0, 1

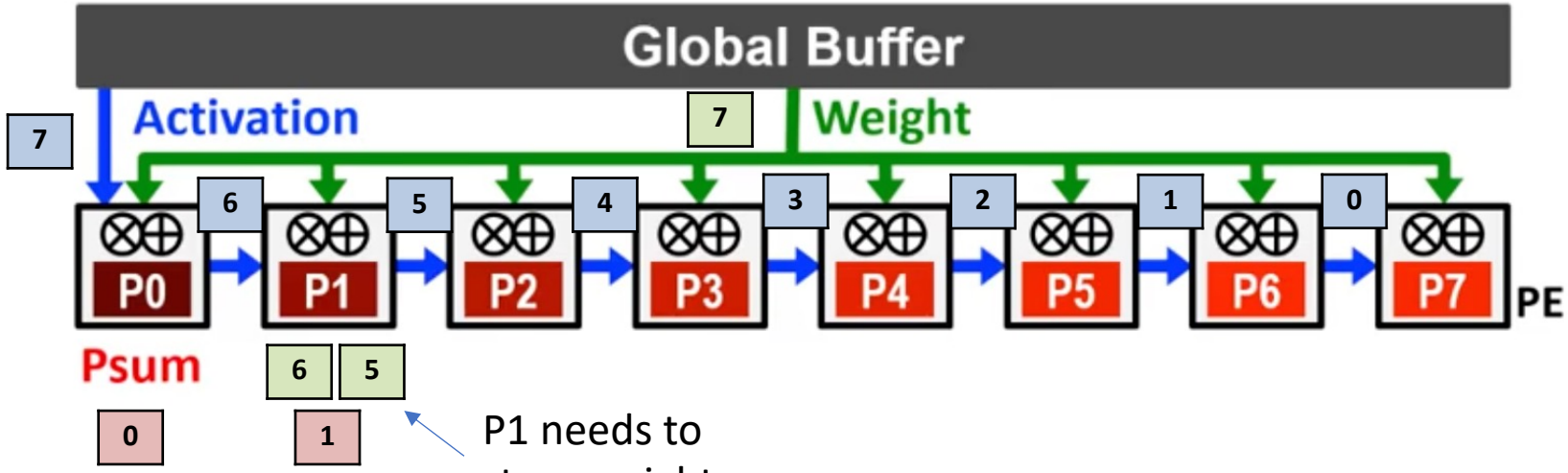
Output 0



Output 1



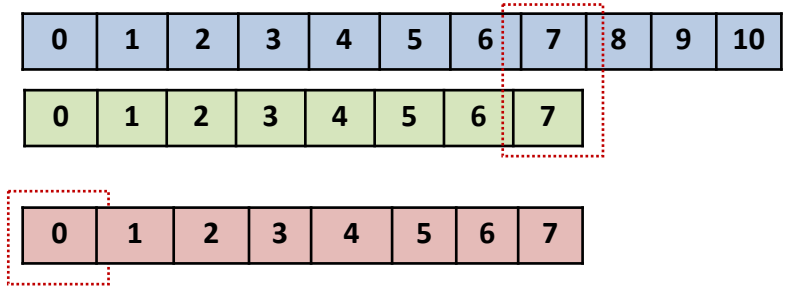
Output Stationary



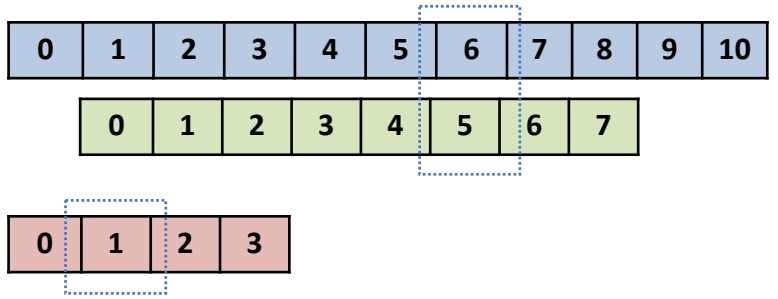
P1 needs to store weights

Focus on Output 0, 1

Output 0



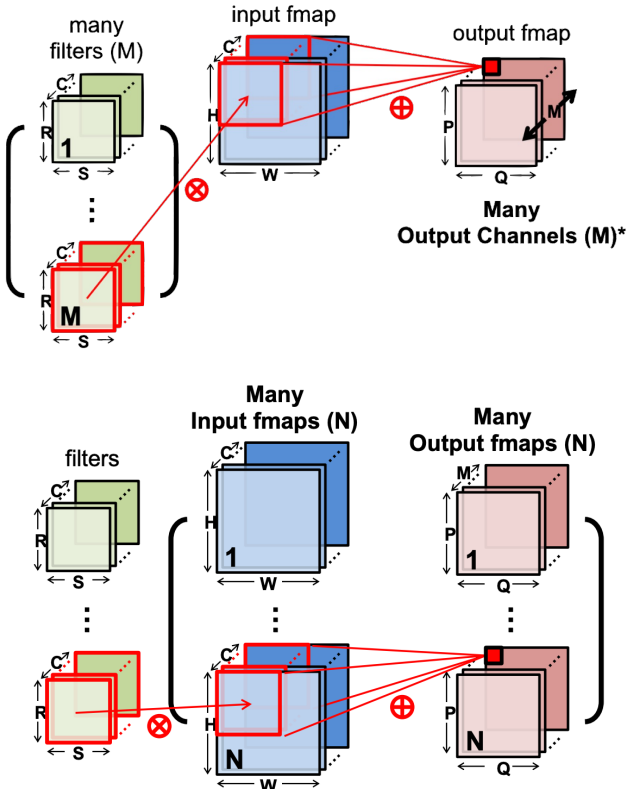
Output 1



Output Stationary for Convolution Layer

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

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

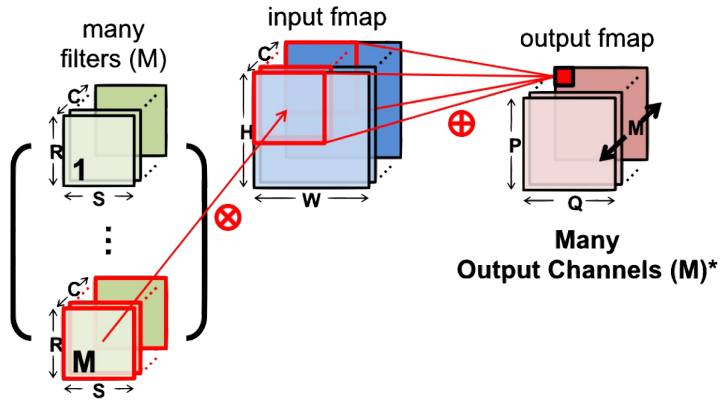


Output Stationary for Convolution Layer

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

Assuming: stride of 1 and no batching

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

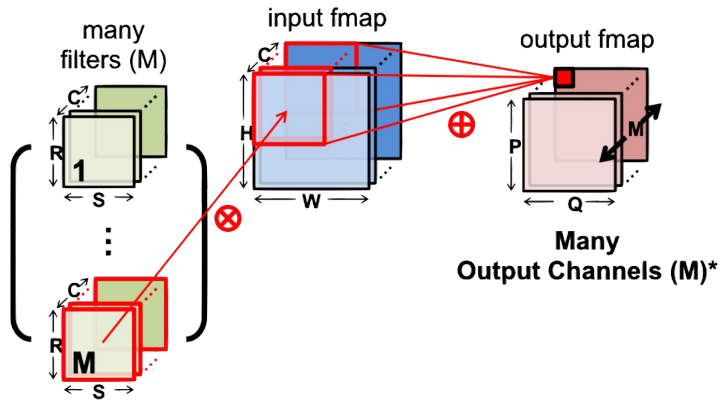


Output Stationary for Convolution Layer

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

Assuming: stride of 1 and no batching

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

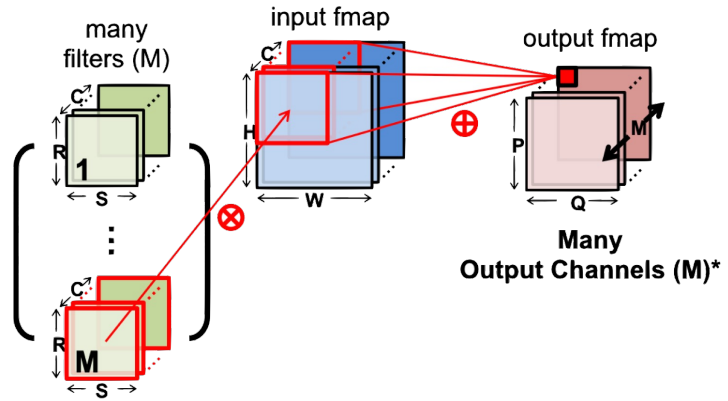


Output Stationary for Convolution Layer

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

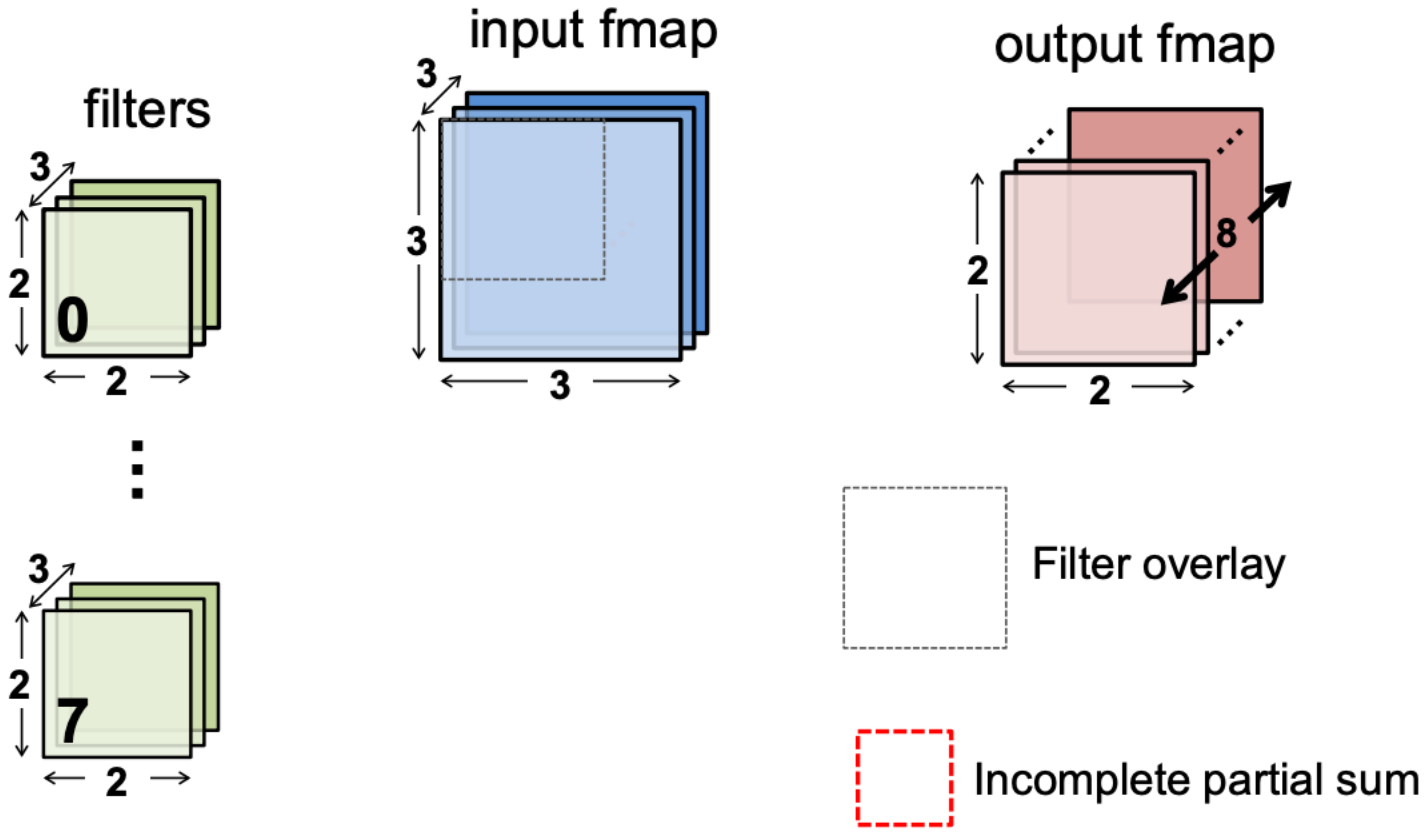
Assuming: stride of 1 and no batching

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

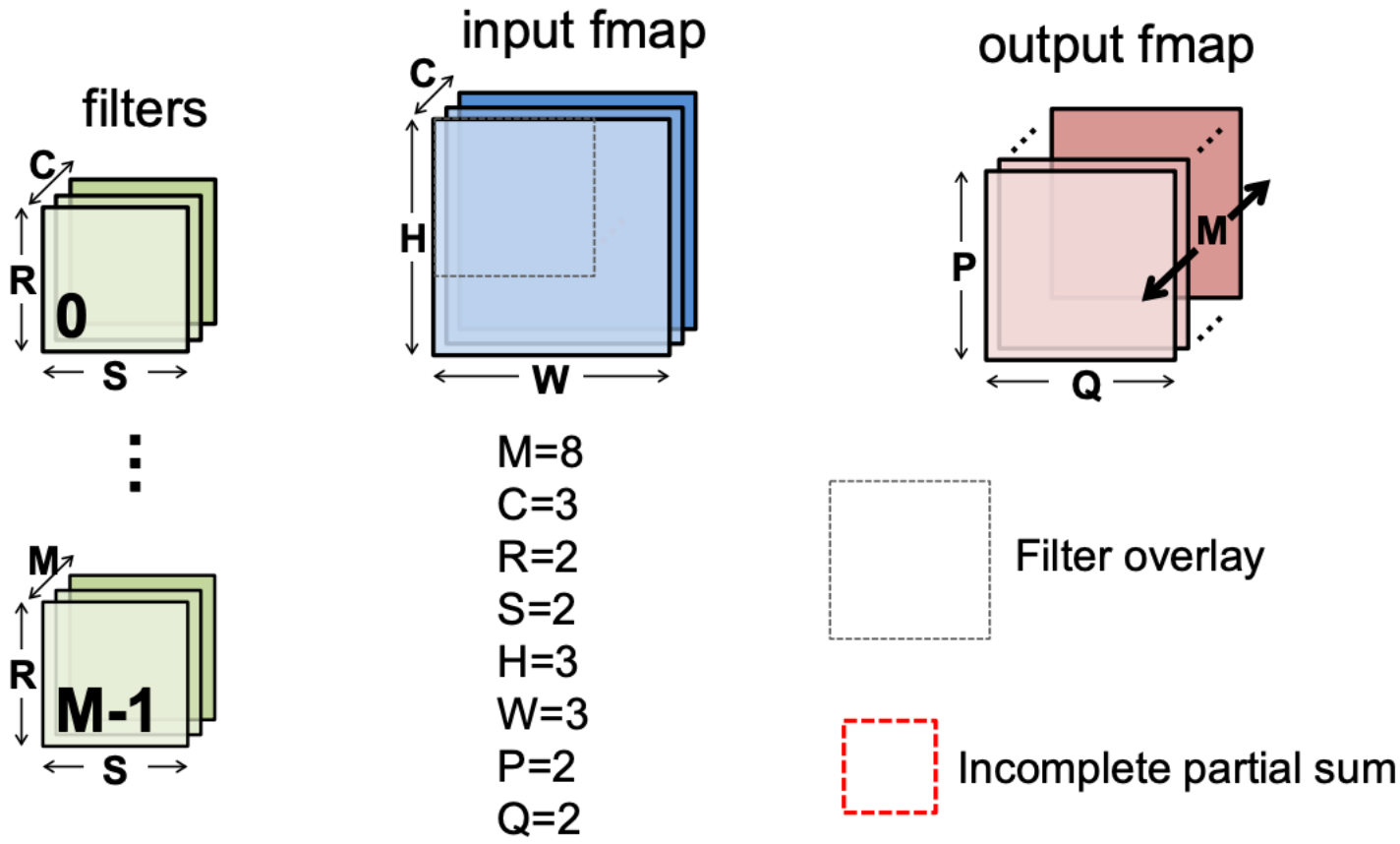


Traversal Order (fast to slow): s, r, q, p
Parallelize on C, M

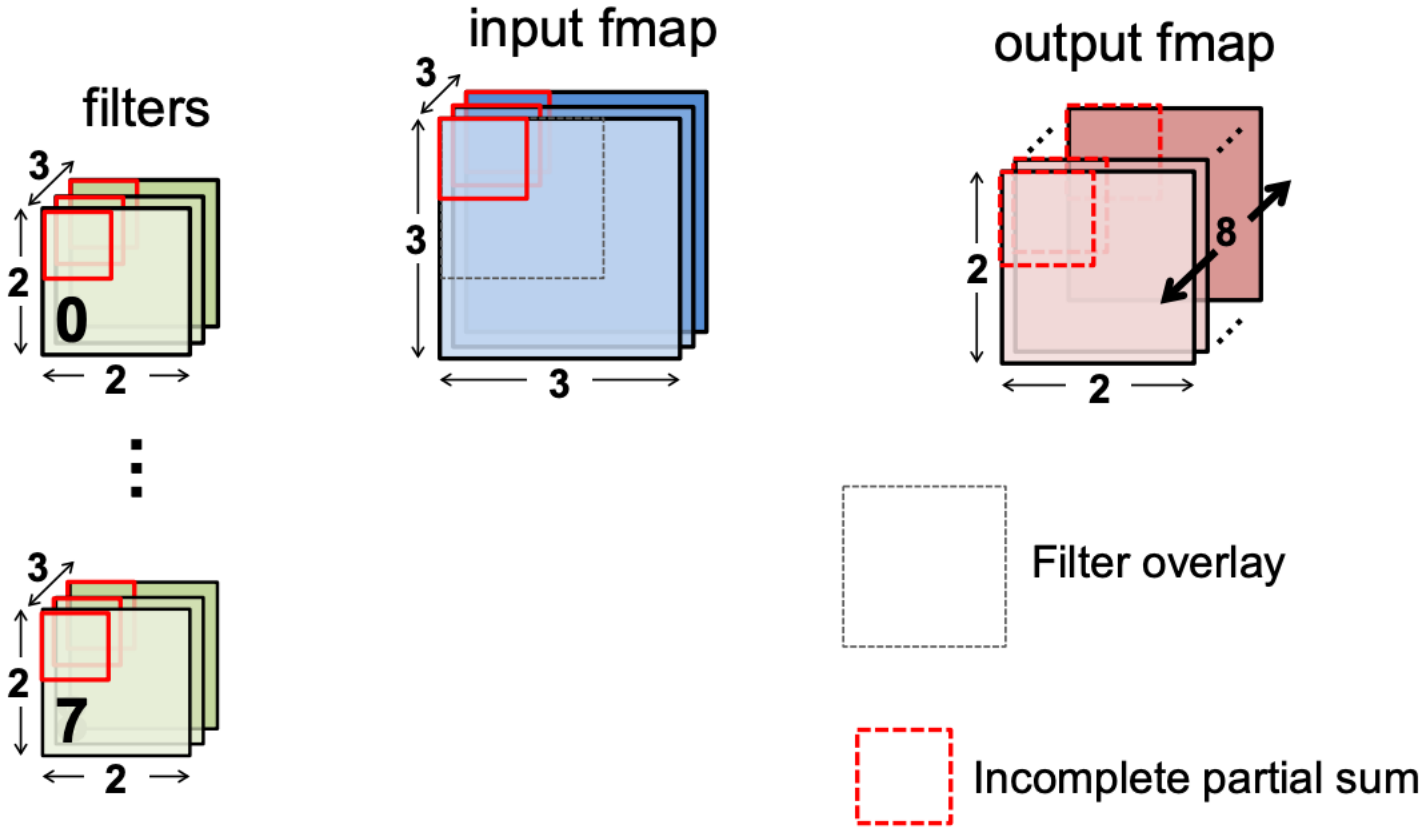
Output Stationary for Convolution Layer



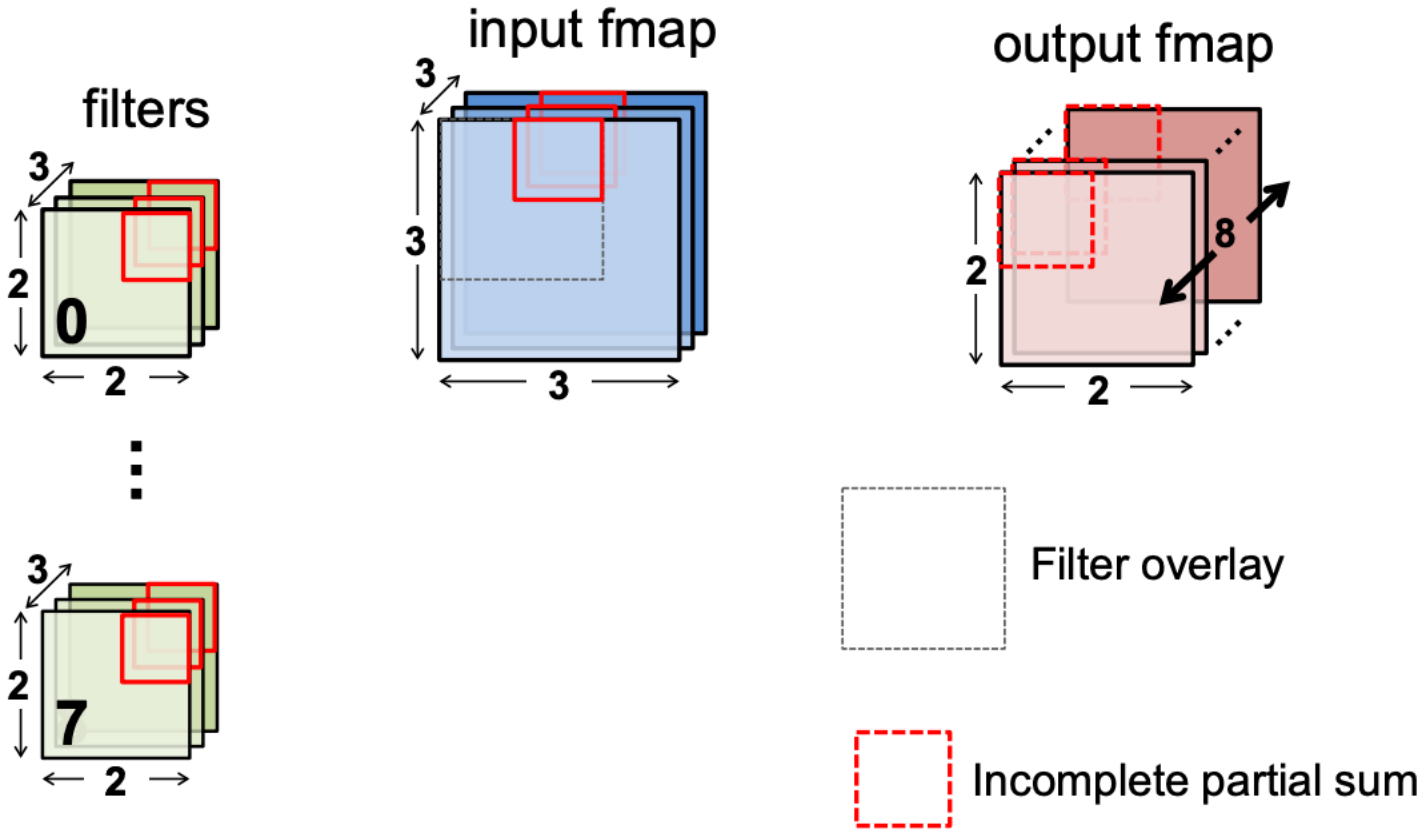
Output Stationary for Convolution Layer



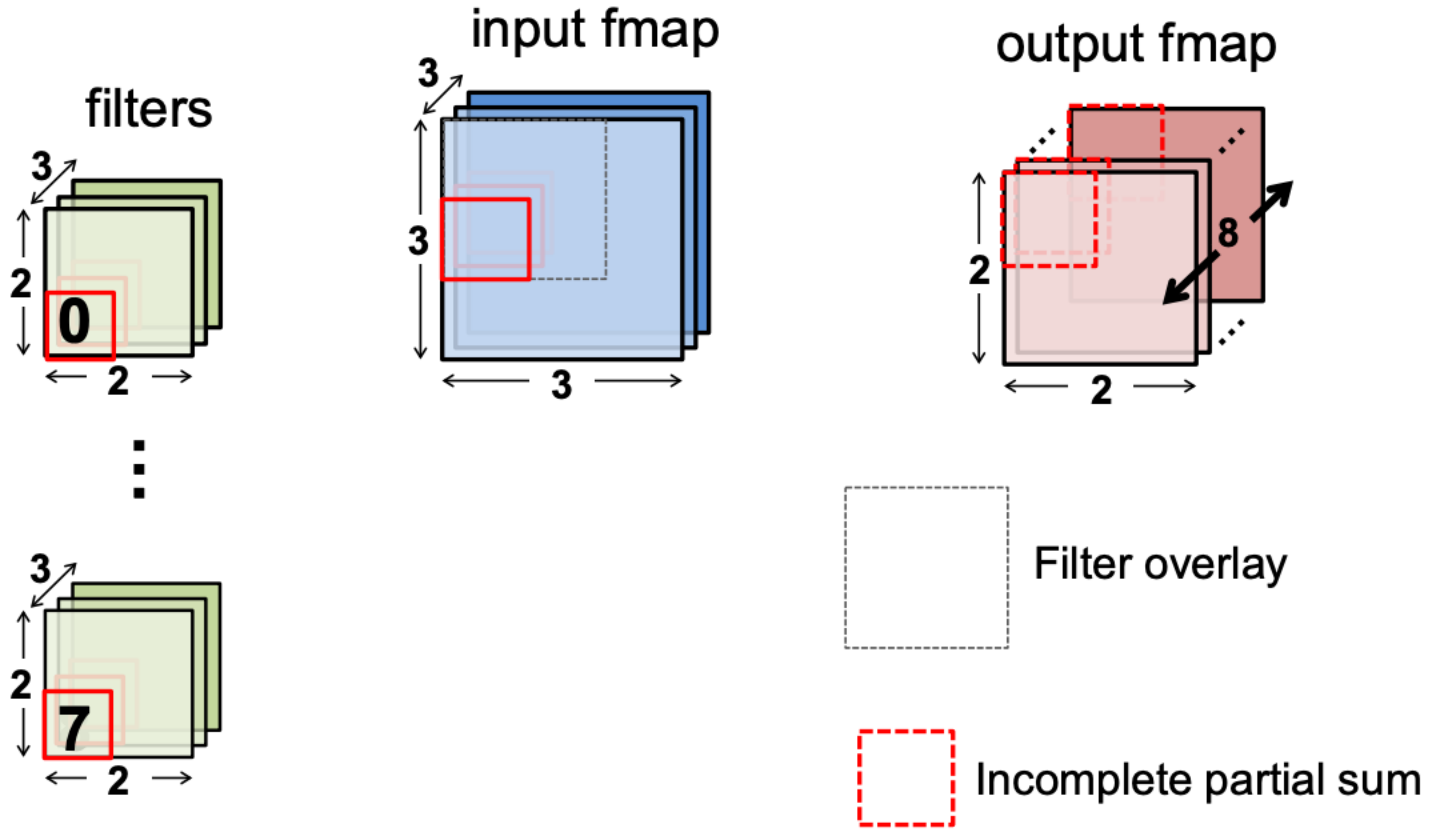
Output Stationary for Convolution Layer



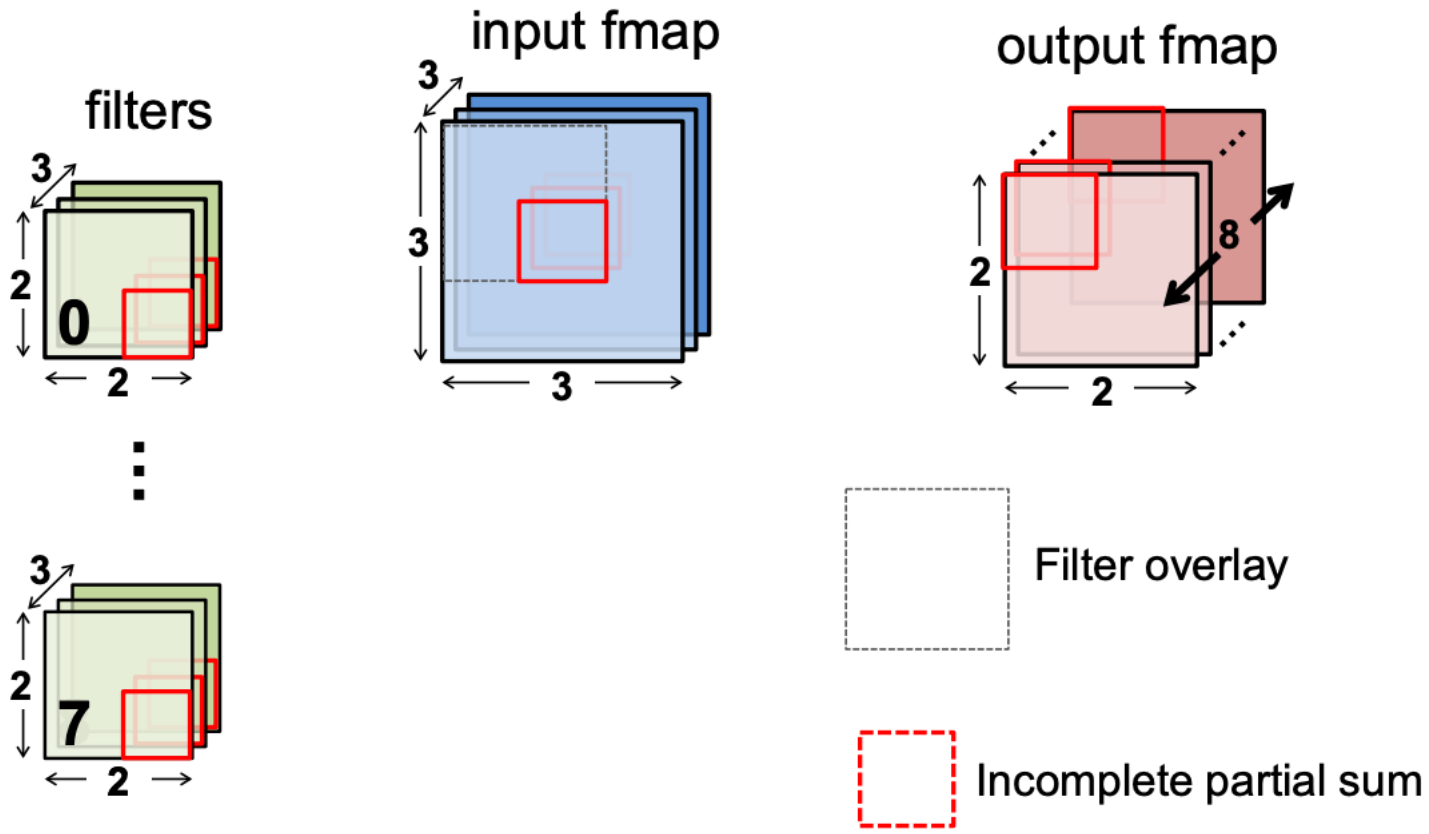
Output Stationary for Convolution Layer



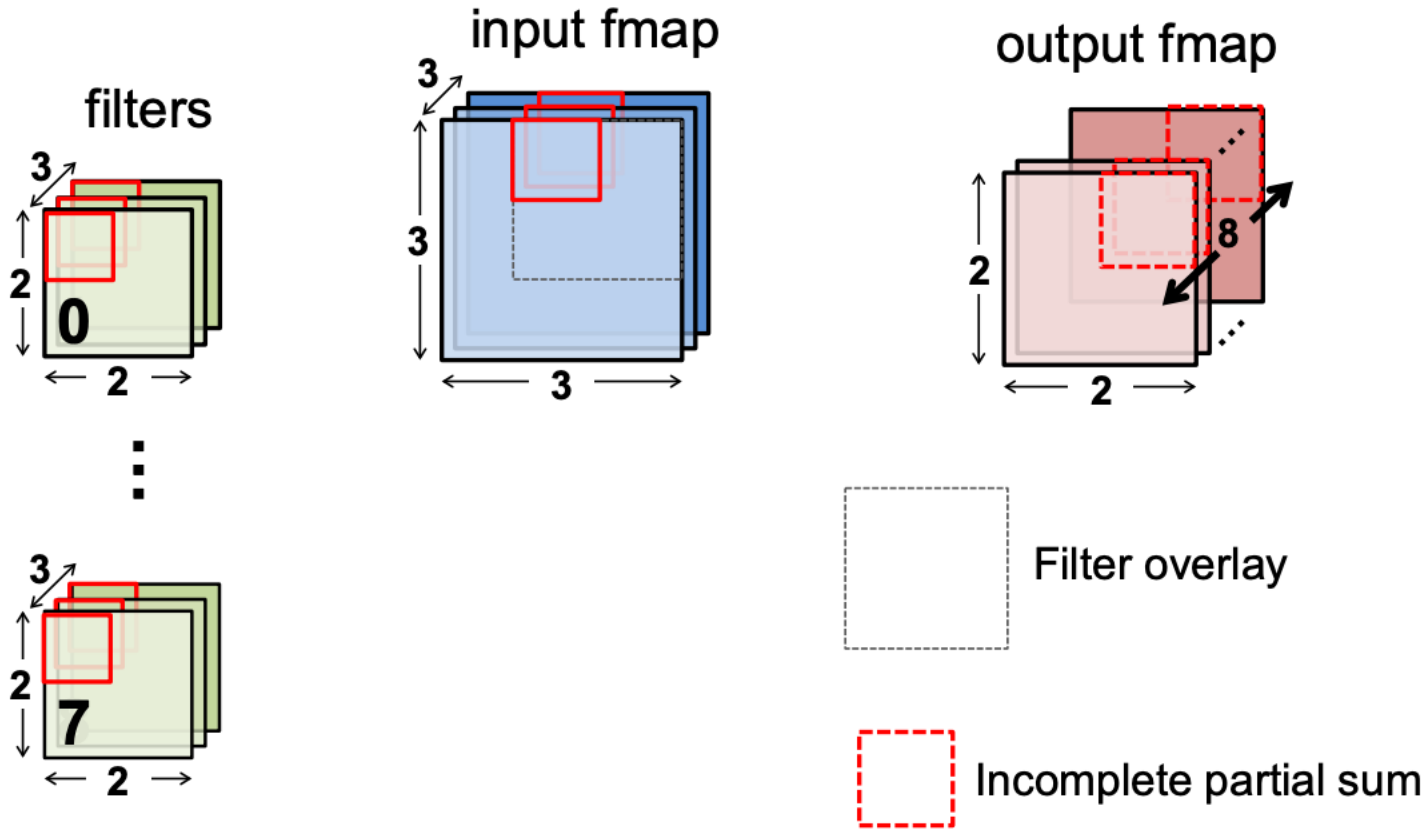
Output Stationary for Convolution Layer



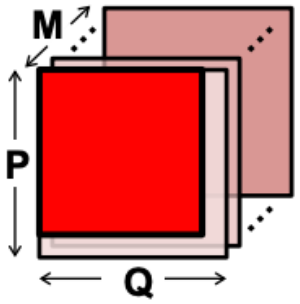
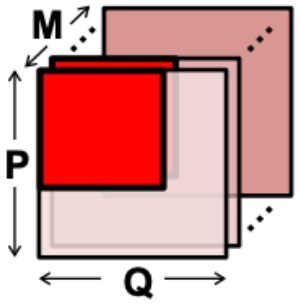
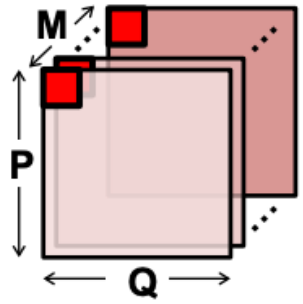
Output Stationary for Convolution Layer



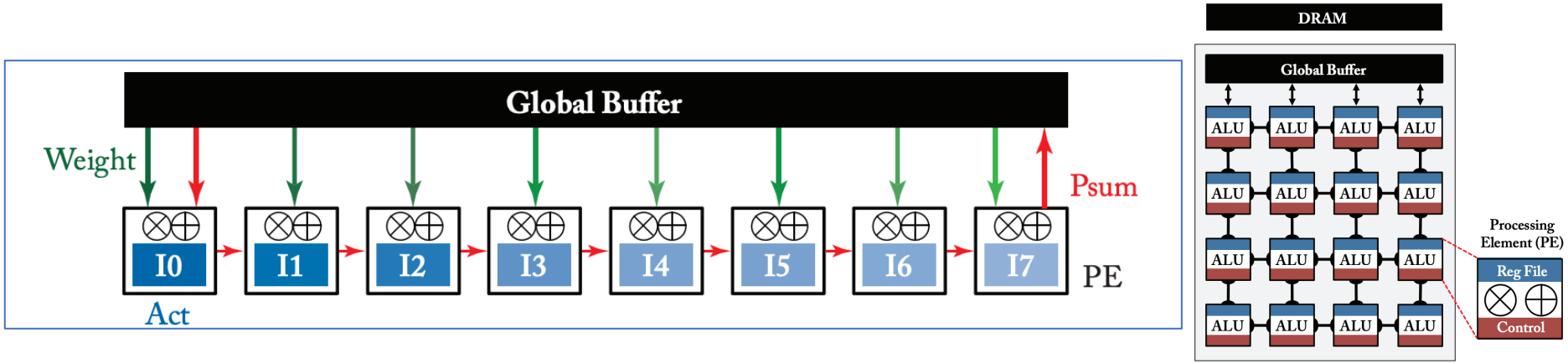
Output Stationary for Convolution Layer



Options for Parallelism

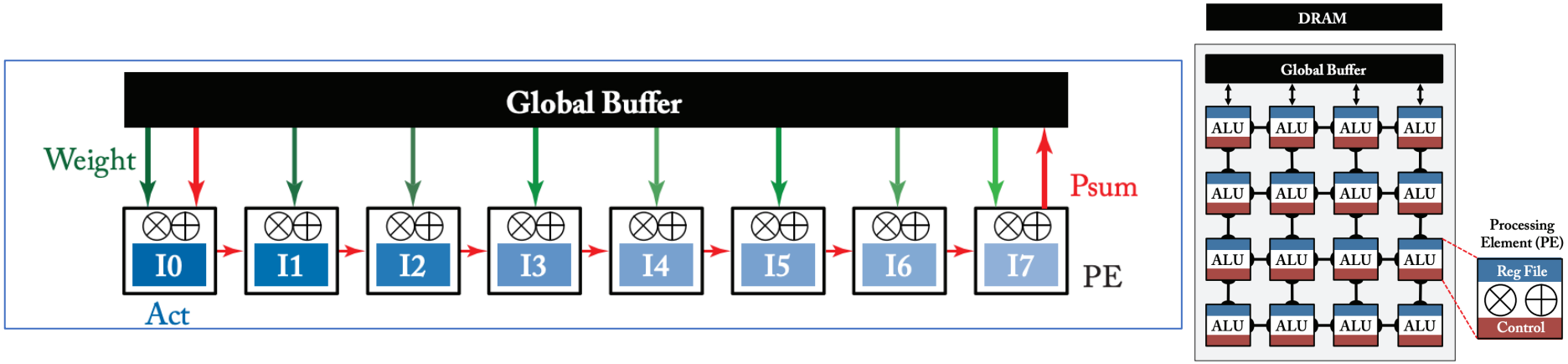
	OS_A	OS_B	OS_C
Parallel Output Region			
# Output Channels	Single	Multiple	Multiple
# Output Activations	Multiple	Multiple	Single
Notes	Targeting **CONV** layers		Targeting **FC** layers

Input Stationary



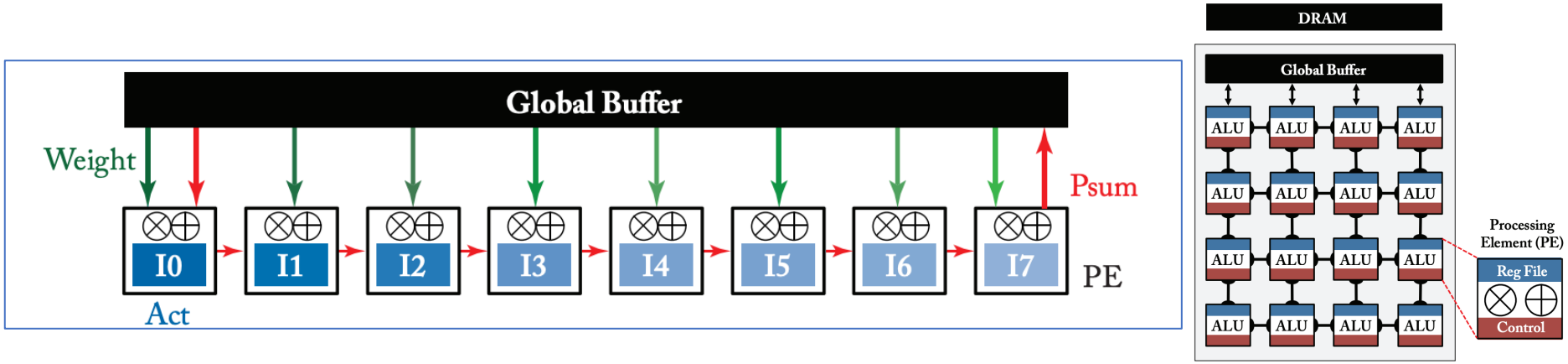
- **Energy Efficiency:**
  - Retain input activations in a fixed location to conserve energy.

Input Stationary



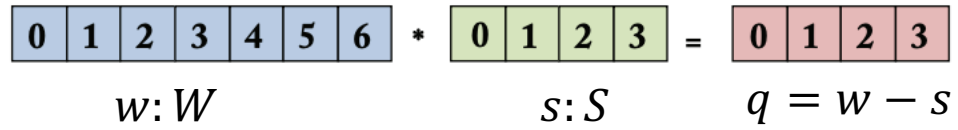
- **Optimized Reuse**:
  - Amplify the reuse rate of input activations.

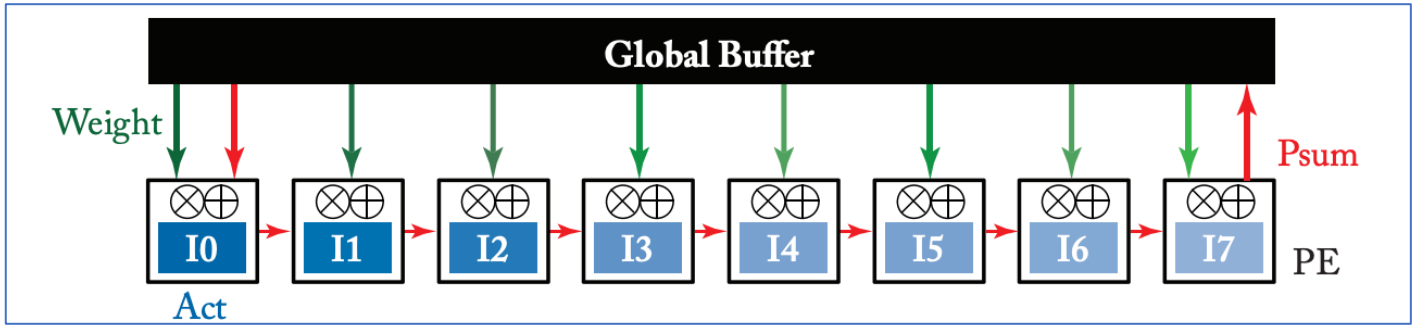
Input Stationary



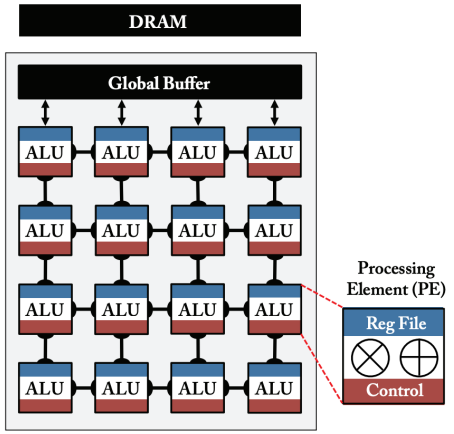
- **Mechanism:**
  - Keep input activations at their original positions, usually within a PE's Register File (RF).
  - Allow weights and partial sums to traverse the system.

Input Stationary



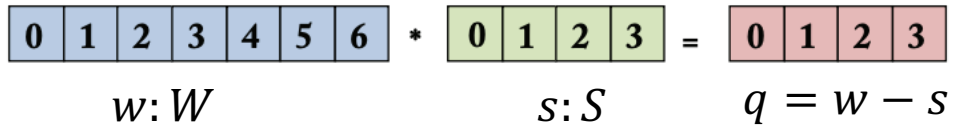


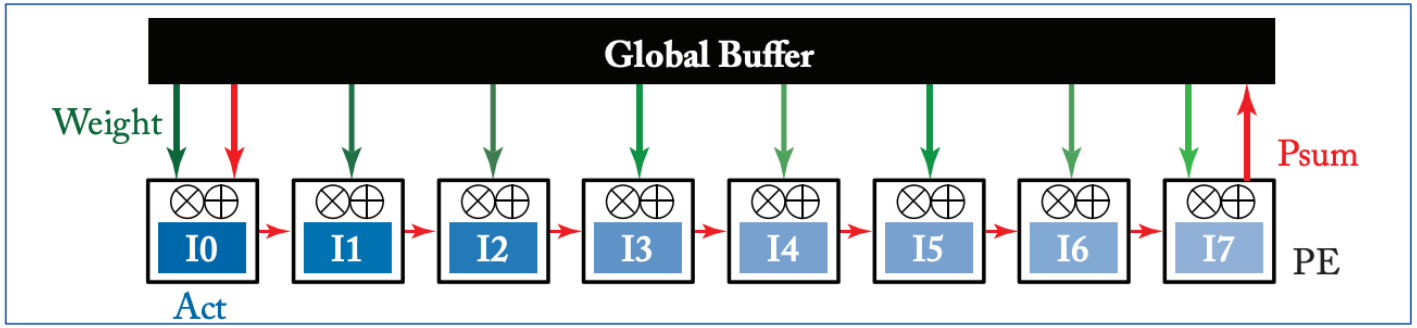

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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



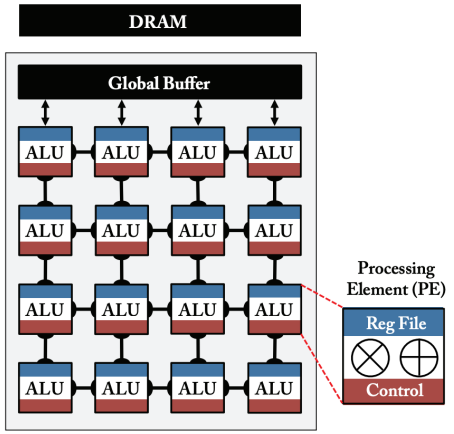
$$O_q = \sum_s I_{q+s} \times F_s$$

Input Stationary






```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

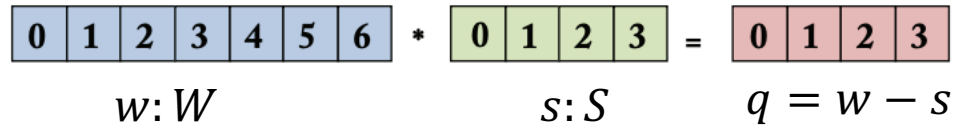


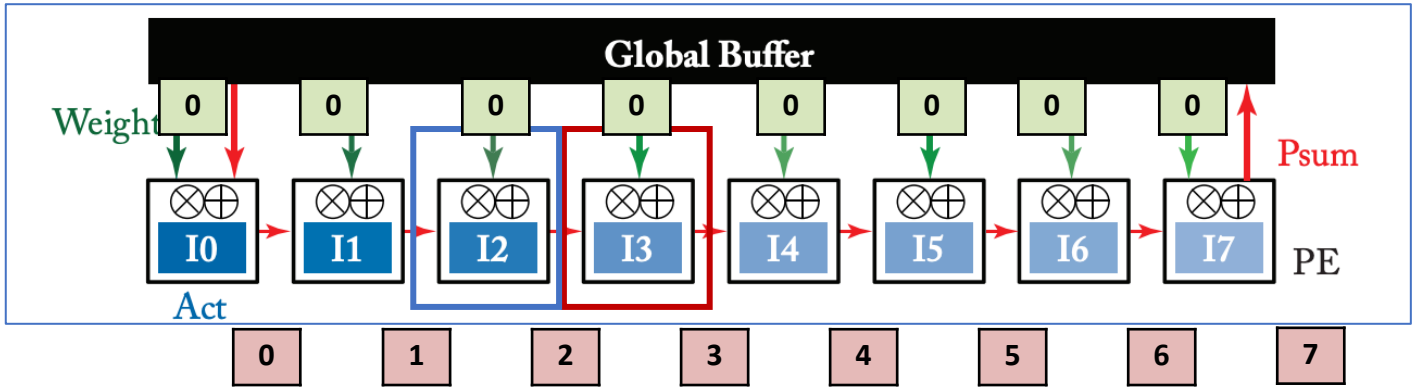
$$O_q = \sum_s I_{q+s} \times F_s$$

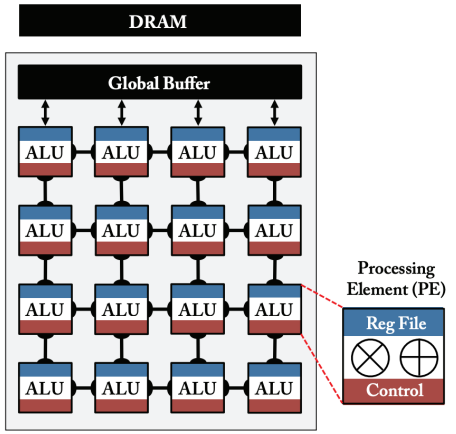
Traversal Order (fastest to slowest):

1. s
2. w

Input Stationary







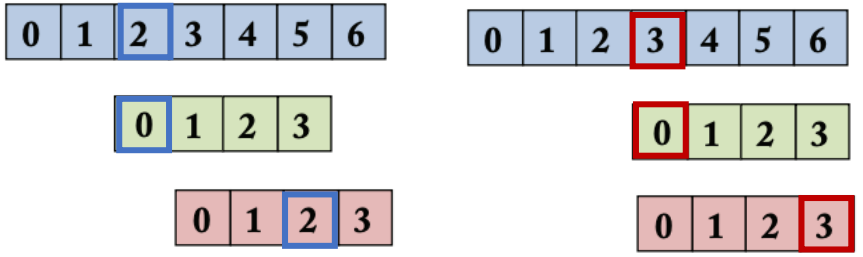

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

Reuse **Activations** by storing them in RF

Unicast/Broadcast **Weights**

Partial Sums are moved from one PE to another.

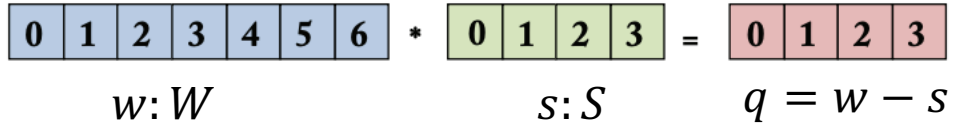
$$O_q = \sum_s I_{q+s} \times F_s$$

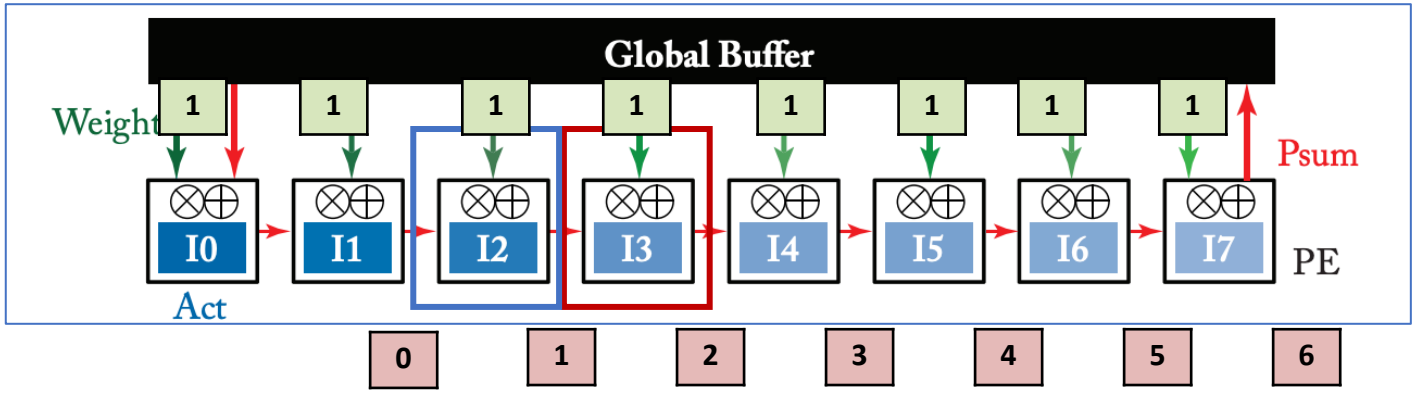


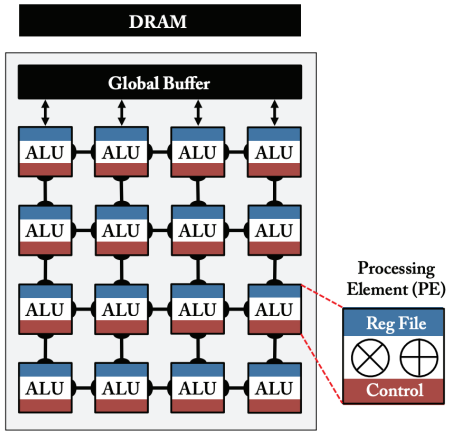
Traversal Order (fastest to slowest):

1. s
2. w

Input Stationary







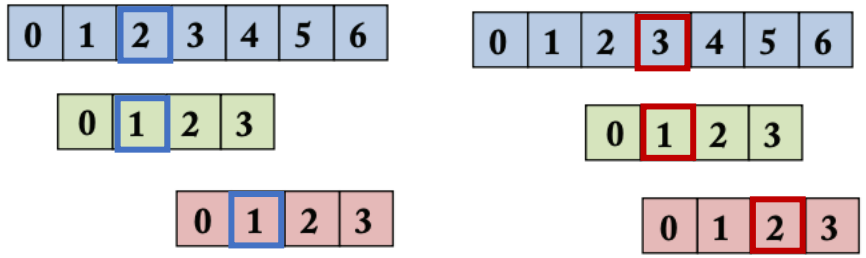

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

Reuse **Activations** by storing them in RF

Unicast/Broadcast **Weights**

Partial Sums are moved from one PE to another.

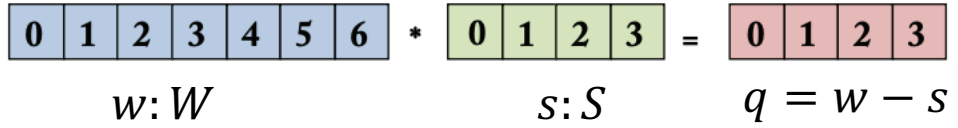
$$O_q = \sum_s I_{q+s} \times F_s$$

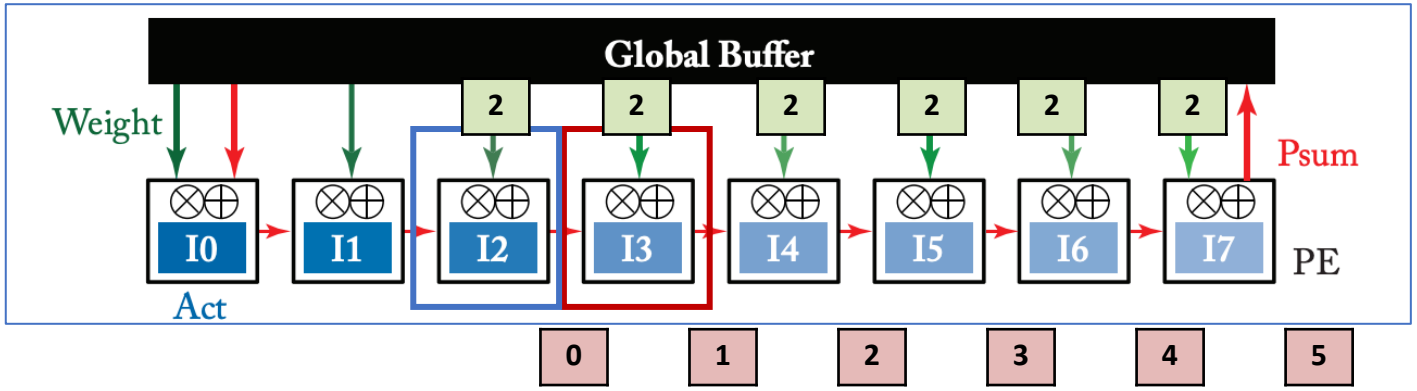


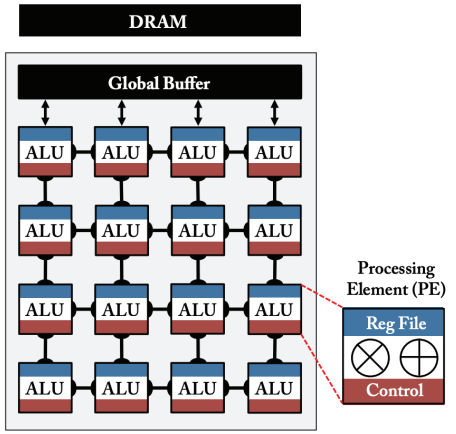
Traversal Order (fastest to slowest):

1. s
2. w

Input Stationary







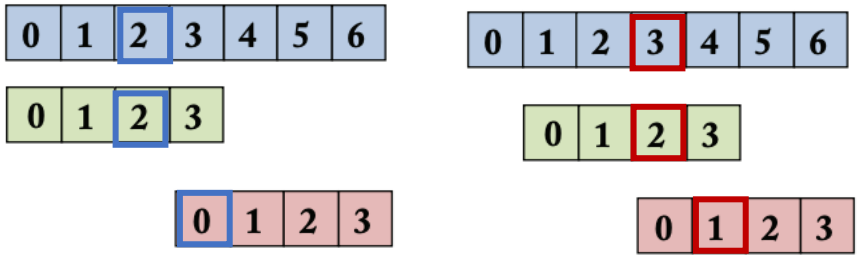

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

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

Reuse **Activations** by storing them in RF

Unicast/Broadcast **Weights**

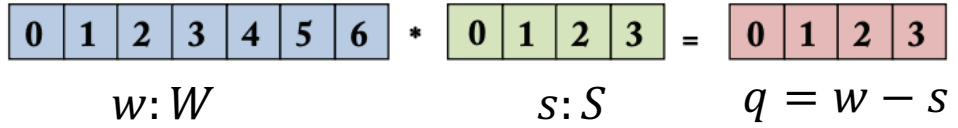
Partial Sums are moved from one PE to another.

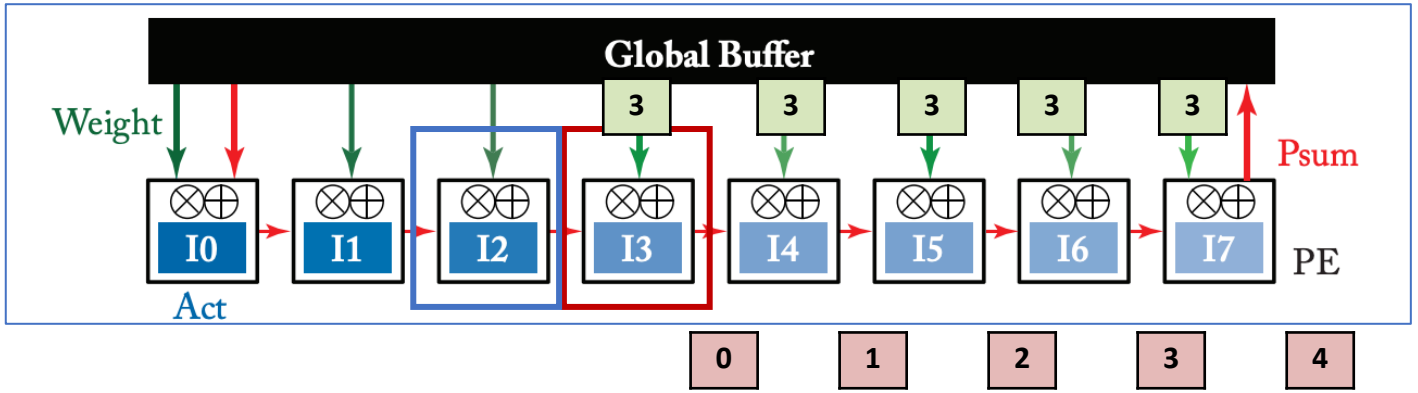


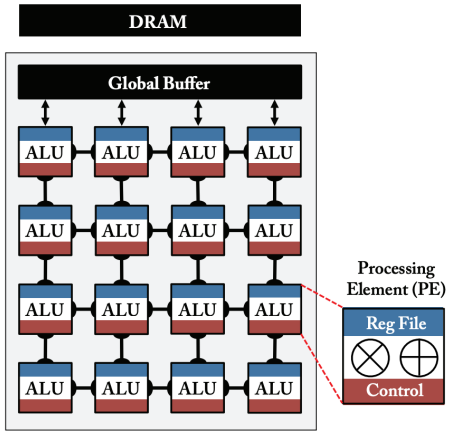
Traversal Order (fastest to slowest):

1. s
2. w

Input Stationary







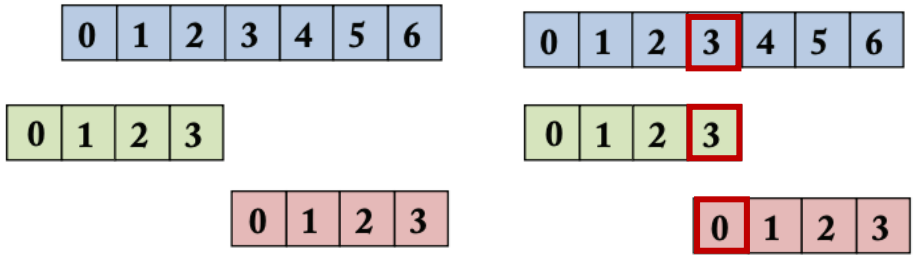

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

Reuse **Activations** by storing them in RF

Unicast/Broadcast **Weights**

Partial Sums are moved from one PE to another.

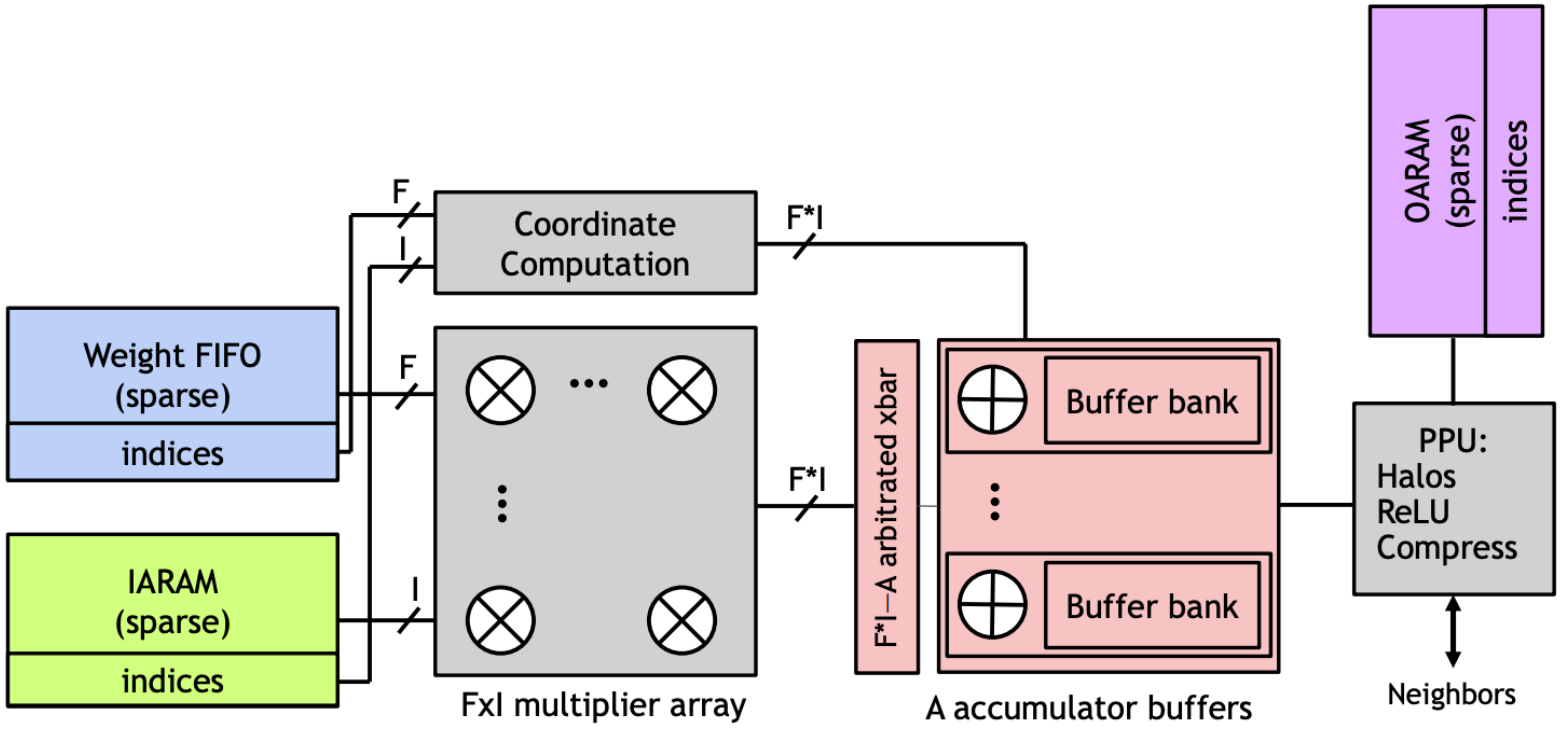
$$O_q = \sum_s I_{q+s} \times F_s$$



Traversal Order (fastest to slowest):

1. s
2. w

Input Stationary Example



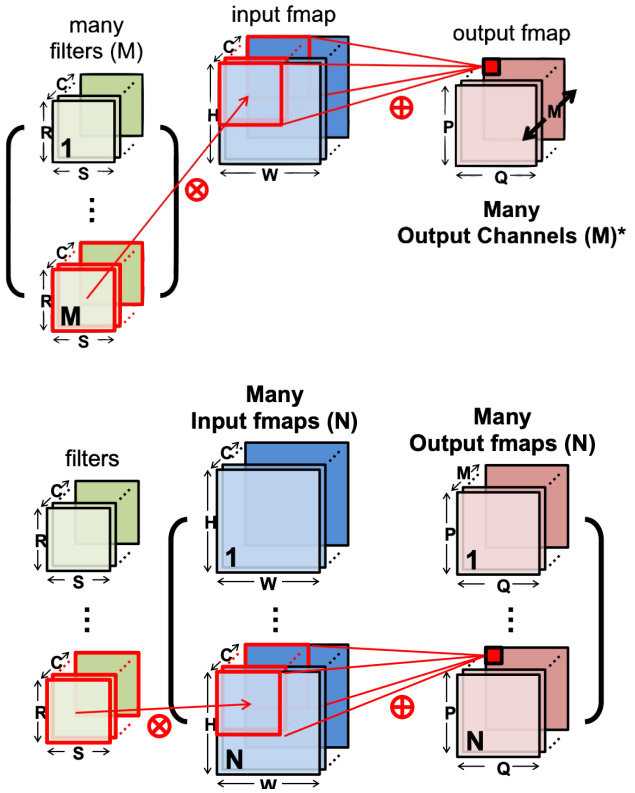
SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks

Source: SCNN, Angshuman Parashar et al.

Input Stationary for Convolution Layer

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

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

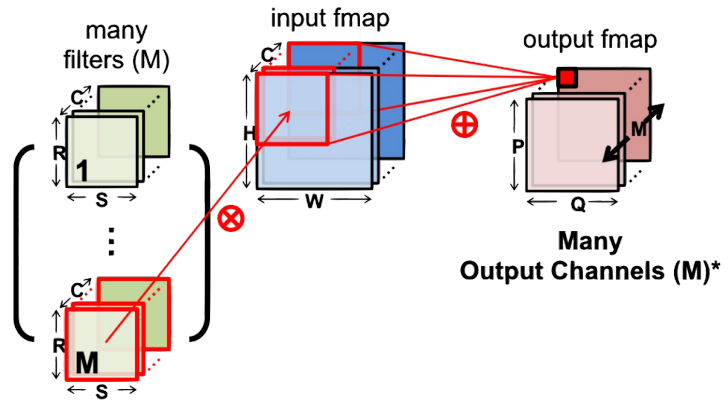


Input Stationary for Convolution Layer

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

Assuming: stride of 1 and no batching

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

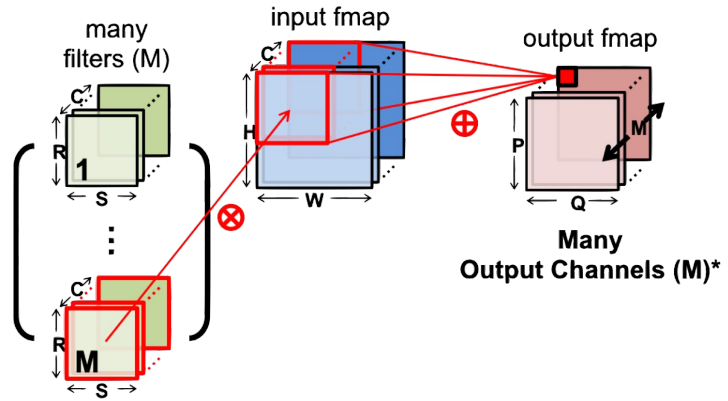


Input Stationary for Convolution Layer

$$O_{m,h-r,w-s} = \sum_{c,r,s} I_{c,h,w} \times F_{m,c,r,s}$$

Assuming: stride of 1 and no batching

```
for h in range(H):
  for w in range(W):
    for r in range(R):
      for s in range(S):
        q = w - s
        p = h - r
        for m in range(M):
          for c in range(C):
            output[m,p,q] += i[c,h,w] * f[m,c,r,s]
```



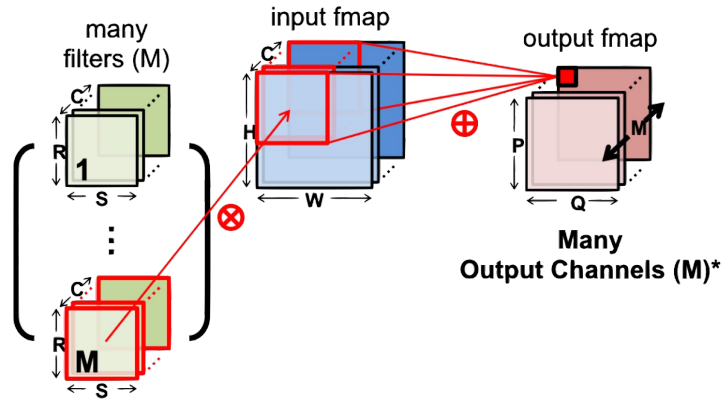

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

Input Stationary for Convolution Layer

$$O_{m,h-r,w-s} = \sum_{c,r,s} I_{c,h,w} \times F_{m,c,r,s}$$

Assuming: stride of 1 and no batching

```
for h in range(H):
  for w in range(W):
    for r in range(R):
      for s in range(S):
        q = w - s
        p = r - h
        parallel-for m in range(M):
          parallel-for c in range(C):
            output[m,p,q] += i[c,p,w] * f[m,c,r,s]
```



Traversal Order (fast to slow): s, r, w, h
Parallelize on C, M

```
# i[W] - input activations
# f[S] - filter weights
# o[Q] - output activations

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

References

- Many images were taken from
  - Efficient Processing of Deep Neural Networks (Synthesis Lectures on Computer Architecture) by Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang , Joel S. Emer