6.5930/1
Hardware Architectures for Deep Learning

Sparse Architectures – Part 2

April 8, 2024

Joel Emer and Vivienne Sze

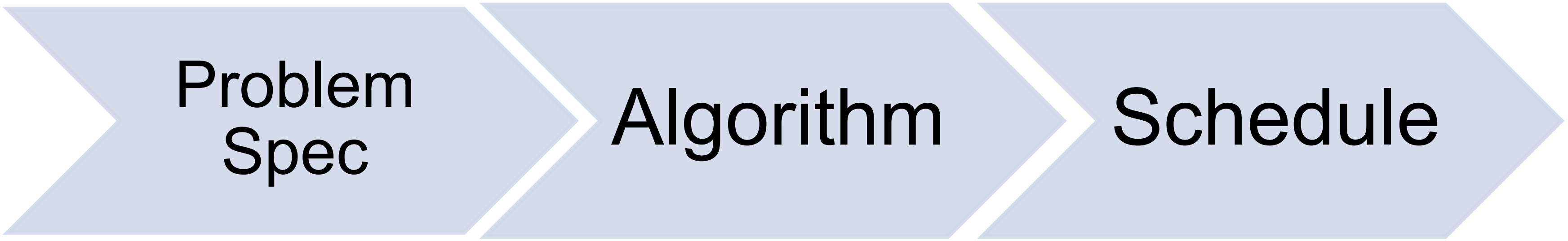
Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Goals of Today's Lecture

- Last lecture, we discussed an abstract representation for sparse tensors with ranks, fibers and fibertrees.
- Today, we will discuss how to translate sparsity into reductions in energy consumption and processing cycles through dataflows that exploit sparsity

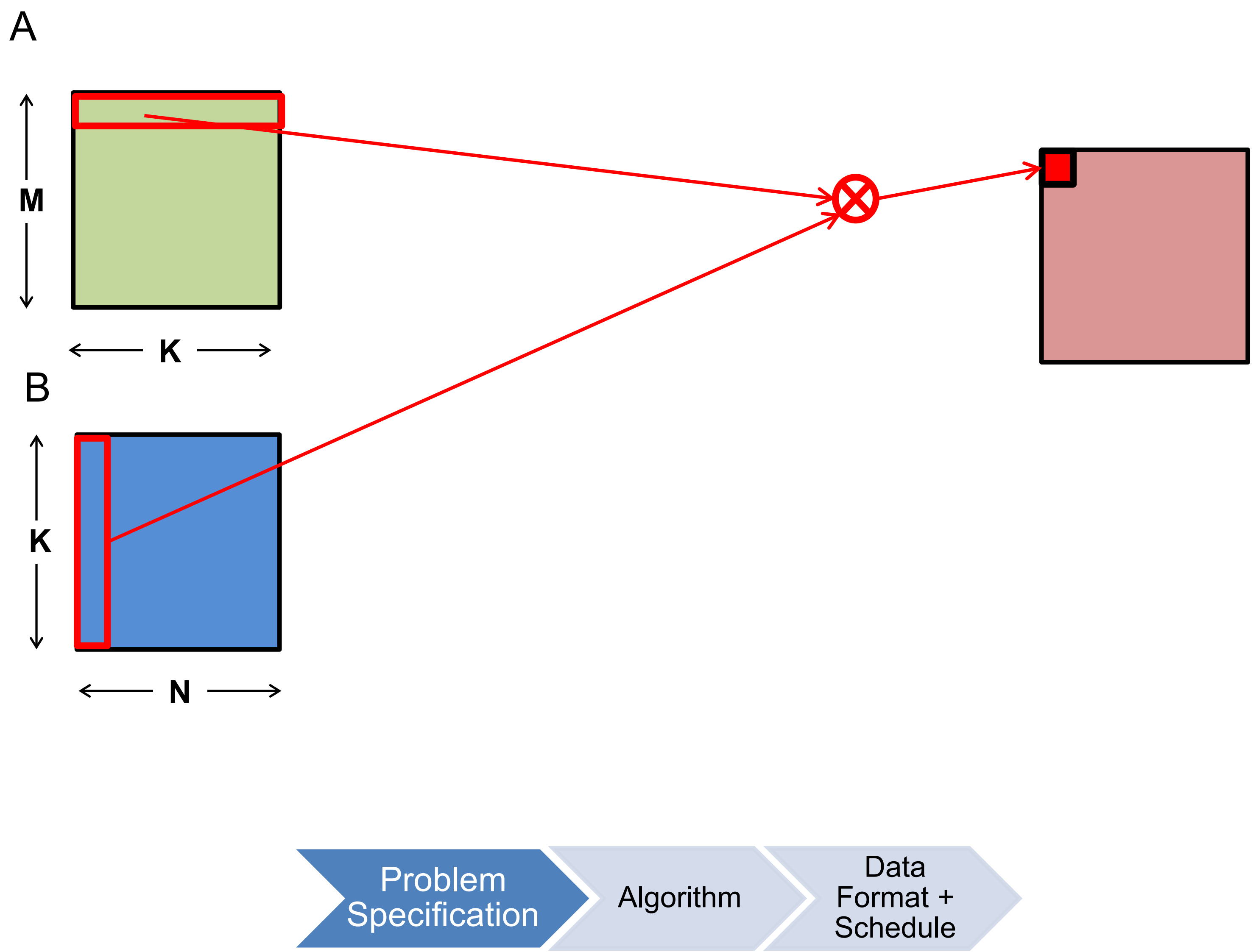
Resources: Course notes - Chapter 8.2 and 8.3

Design Steps

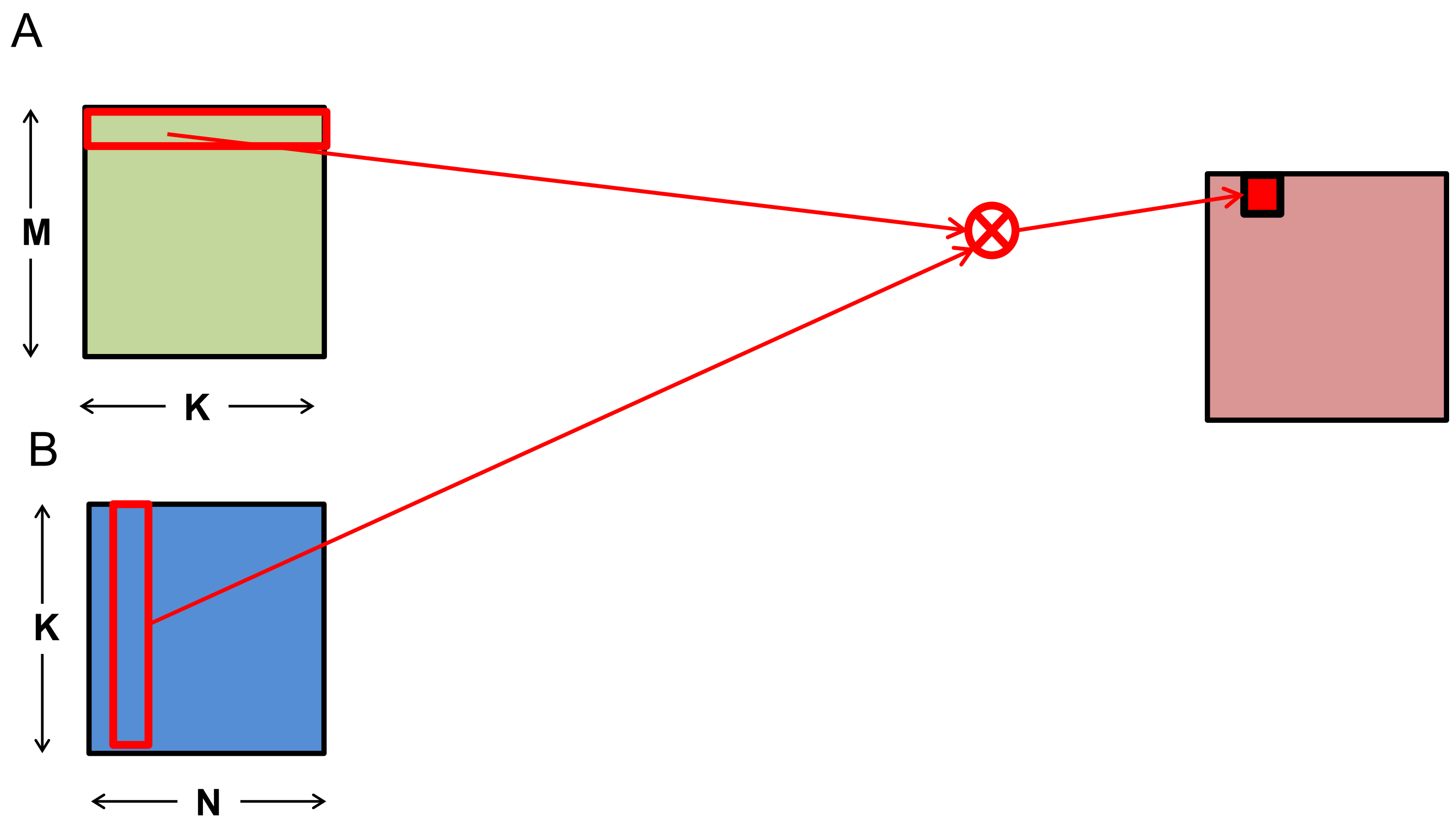


[Halide, Ragan-Kelly, et.al., PLDI 2013]

Matrix Multiply



Matrix Multiply



Problem Specification | Algorithm | Data Format + Schedule

Einsum – Matrix Multiply

$$Z_{m,n} = A_{m,k} \times B_{n,k}$$

Operational Definition for Einsums (ODE):

- Traverse all points in space of all legal index values (iteration space)
- At each point in iteration space:
  - Calculate value on right hand at specified indices for each operand
  - Assign value to operand at specified indices on left hand side
  - Unless that operand is non-zero, then reduce value into it

[Relativity, Einstein, Annelen de Physik, 1916]
[TACO, Kjolstad et.al., ASE 2017]
[Timeloop, Parashar et.al., ISPASS 2019]

Einsum – Matrix Multiply

$$Z_{m,n} = A_{m,k} \times B_{n,k}$$

- **Shared indices -> intersection**

Problem Specification
Algorithm
Data Format + Schedule

Einsum – Matrix Multiply

$$Z_{m,n} = A_{m,k} \times B_{n,k}$$

- **Shared indices -> intersection**
- **Contracted indices -> reduction**

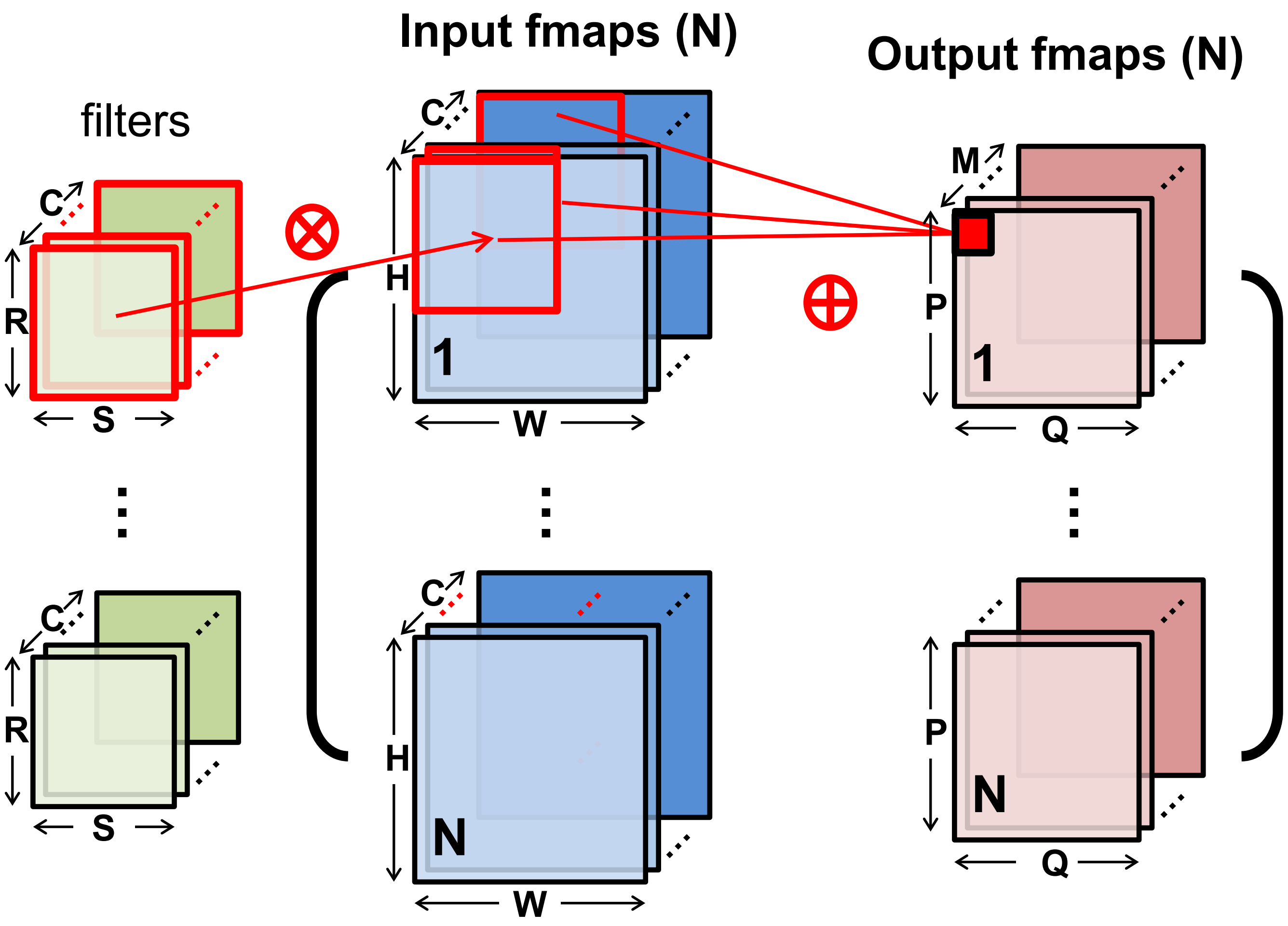
Einsum – Matrix Multiply

$$Z_{\color{red}{m,n}} = A_{\color{red}{m},k} \times B_{\color{red}{n},k}$$

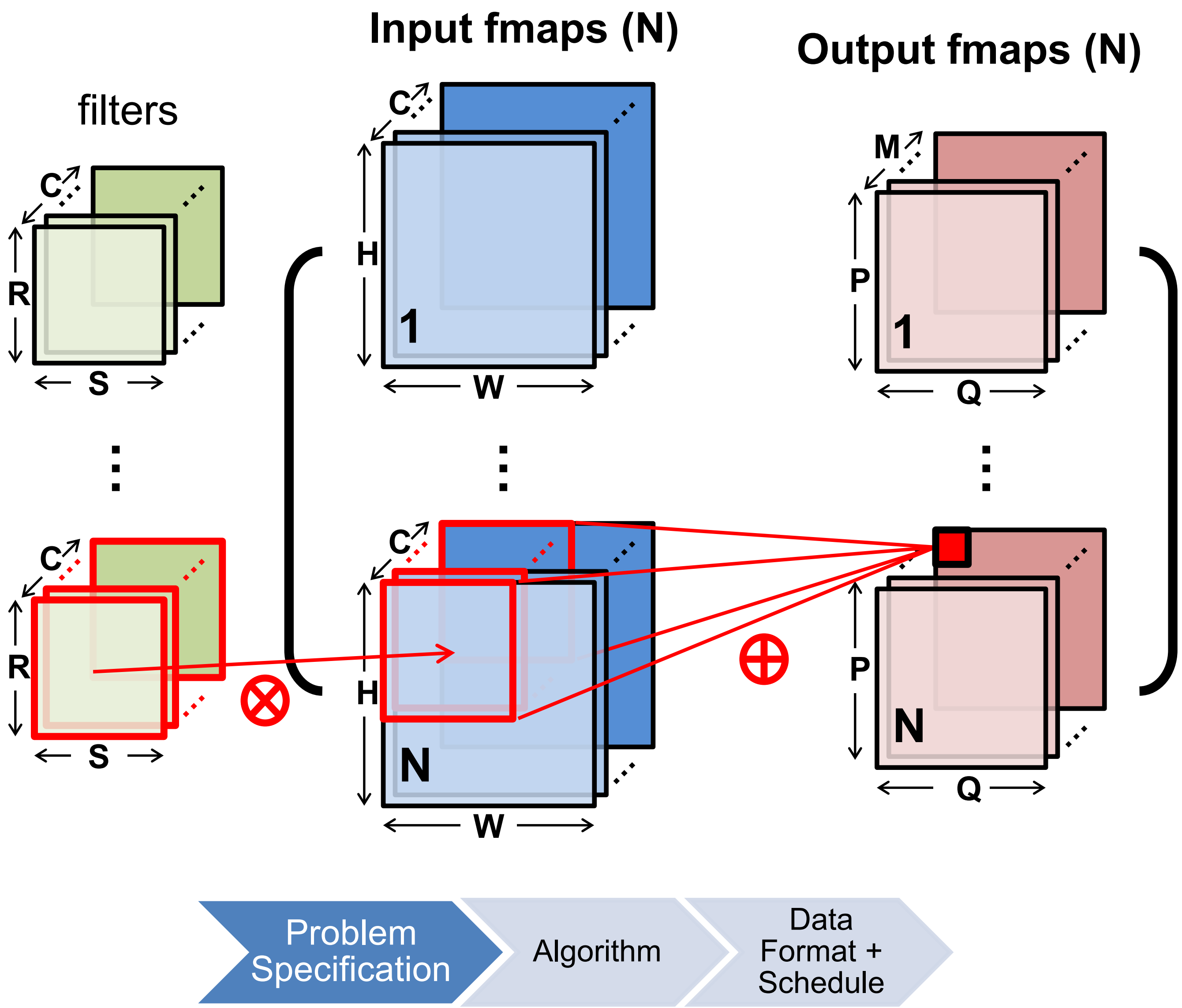
- **Shared indices -> intersection**
- **Contracted indices -> reduction**
- **Uncontracted indices -> populate output point**

Problem Specification → Algorithm → Data Format + Schedule

Convolution (CONV) Layer



Convolution (CONV) Layer



Einsum - Convolution

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

- **Shared indices -> intersection**
- **Contracted indices -> reduction**
- **Uncontracted indices -> populate output point**
- **Index arithmetic -> projection**

[Extensor, Hegde, et.al., MICRO 2019]

Problem Specification → Algorithm → Data Format + Schedule

Einsum - Convolution

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

- **Shared indices -> intersection**
- **Contracted indices -> reduction**
- **Uncontracted indices -> populate output point**
- **Index arithmetic -> projection**

[Extensor, Hegde, et.al., MICRO 2019]

Problem Specification → Algorithm → Data Format + Schedule

Aspects of Scheduling - Sparsity

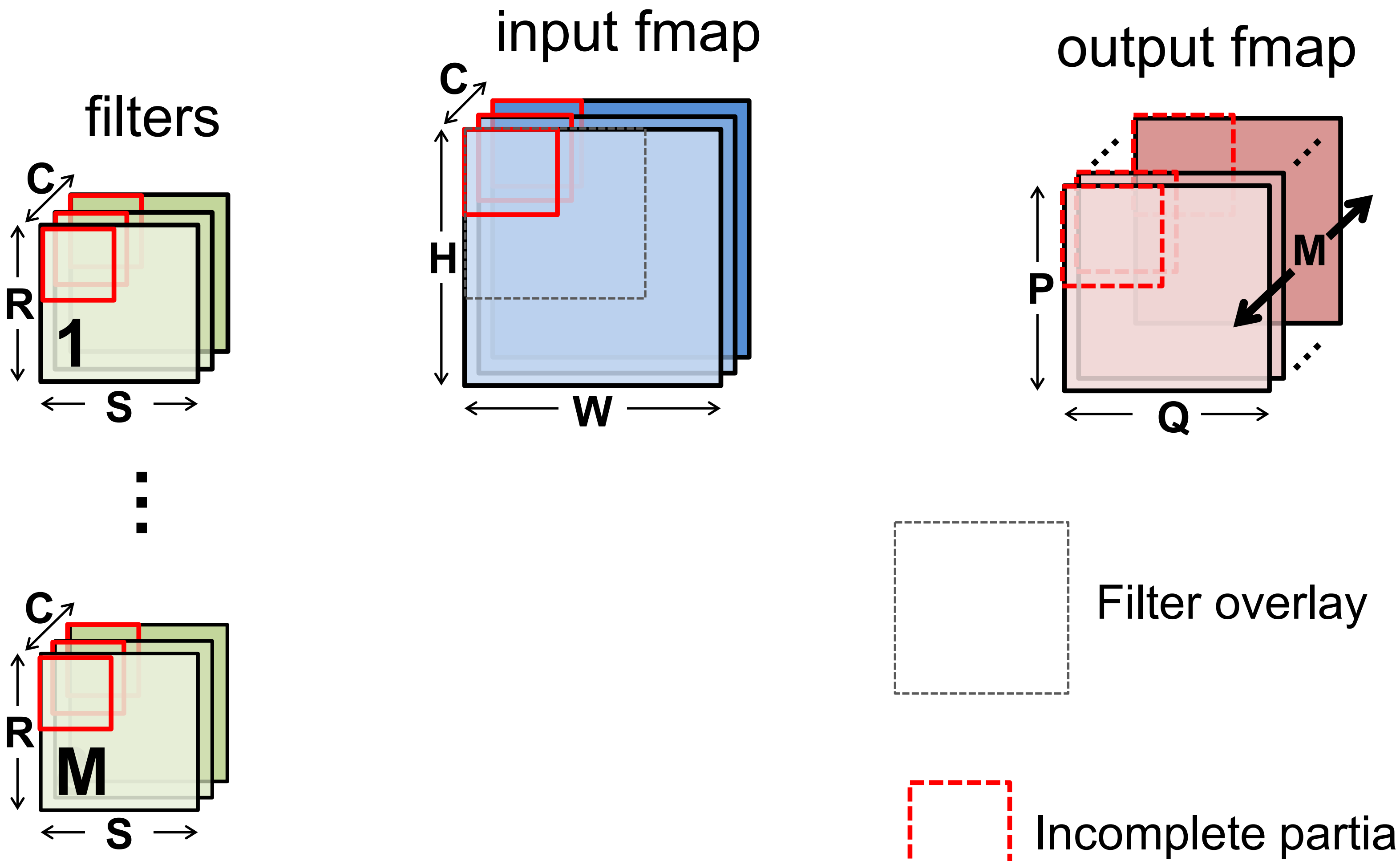
Format:
Choose tensor representations to save storage space and energy associated with zero accesses

Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

Skipping:
Explicitly eliminate ineffectual storage accesses and computes by skipping the cycle to save energy and time

CONV: Exploiting Sparse Weights

CONV Layer



Filter overlay

Incomplete partial sum

1-D Output-Stationary Convolution

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

```
i = Array(W)        # Input activations
f = Array(S)        # Filter weights
o = Array(Q)        # Output activations

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

† Assuming: 'valid' style convolution

1-D Output-Stationary Convolution


```
i = Array(W)        # Input activations
f = Array(S)        # Filter weights
o = Array(Q)        # Output activations

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

What opportunity(ies) exist if some of the values are zero?

Can avoid reading operands, doing multiply and updating output

† Assuming: 'valid' style convolution

1-D Output-Stationary Convolution

Weights		Inputs		Outputs
8 0 6	*		=	
S		W		Q = W-ceil(S/2)†

```
i = Array(W)         # Input activations
f = Array(S)         # Filter weights
o = Array(Q)         # Output activations

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

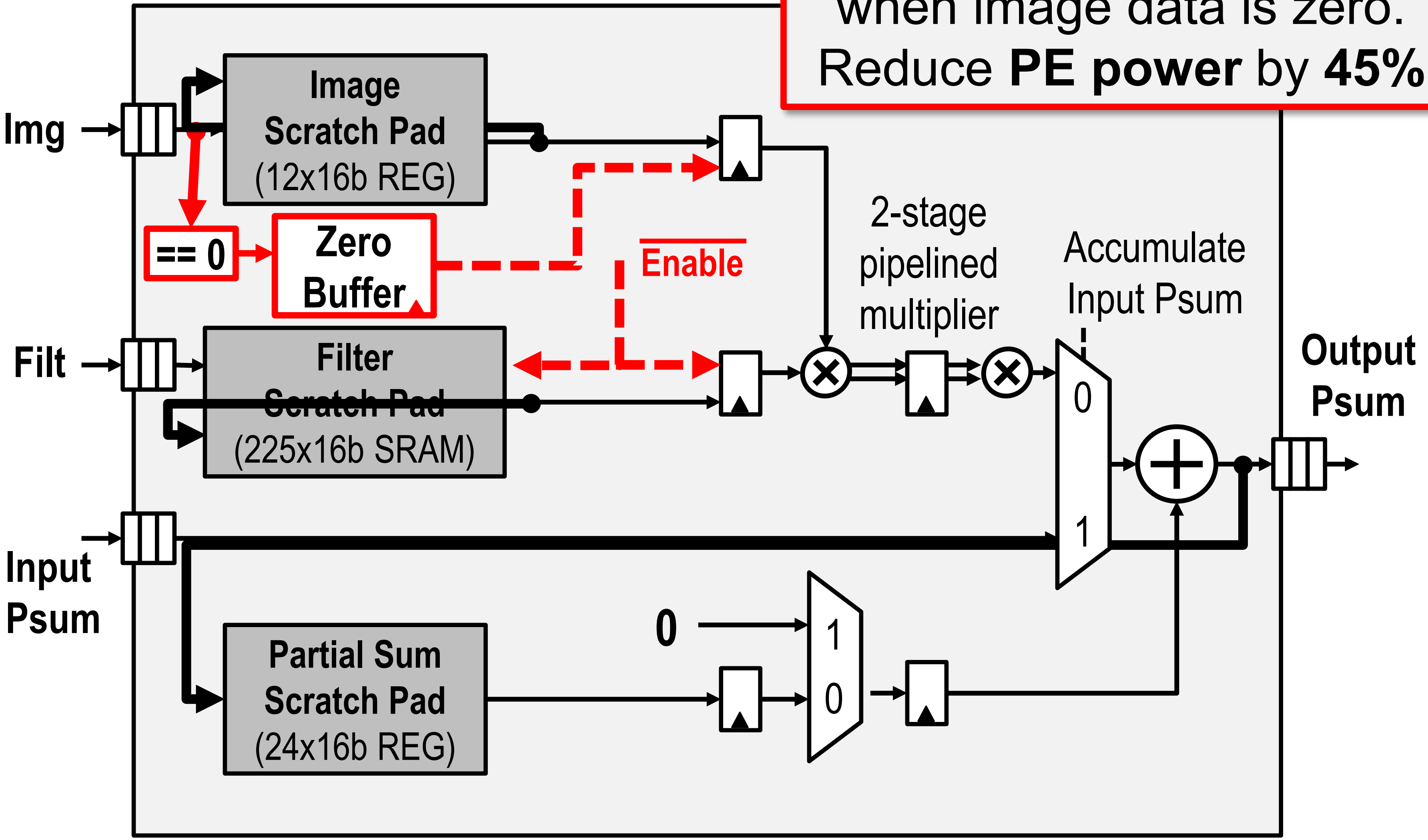
What did we save using the conditional execution? Energy

What didn't we save using the conditional execution? Time

† Assuming: 'valid' style convolution

Eyeriss – Clock Gating

Skip **mult** and **mem reads** when image data is zero. Reduce **PE power** by **45%**



Weight Stationary

```
i = Array(W)          # Input activations
f = Array(S)          # Filter weights
o = Array(Q)          # Output activations

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

Note: s, w are the coordinates of the desired elements of the tensor

Need to calculate position/coordinate in third tensor, i.e., do a **projection**

The variables "i" and "f" are? Tensors, Arrays and Fibers

What are the tensor representations of "i" and "f"? Uncompressed Implicit coordinate

The variables "s" and "w" are? Coordinates and Positions

Naïve Sparse Weight Stationary

```
i = Tensor(W)        # Input activations
f = Tensor(S)        # Filter weights
o = Array(Q)         # Output activations

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

The variables "i" and "f" are? Tensors, Fibers

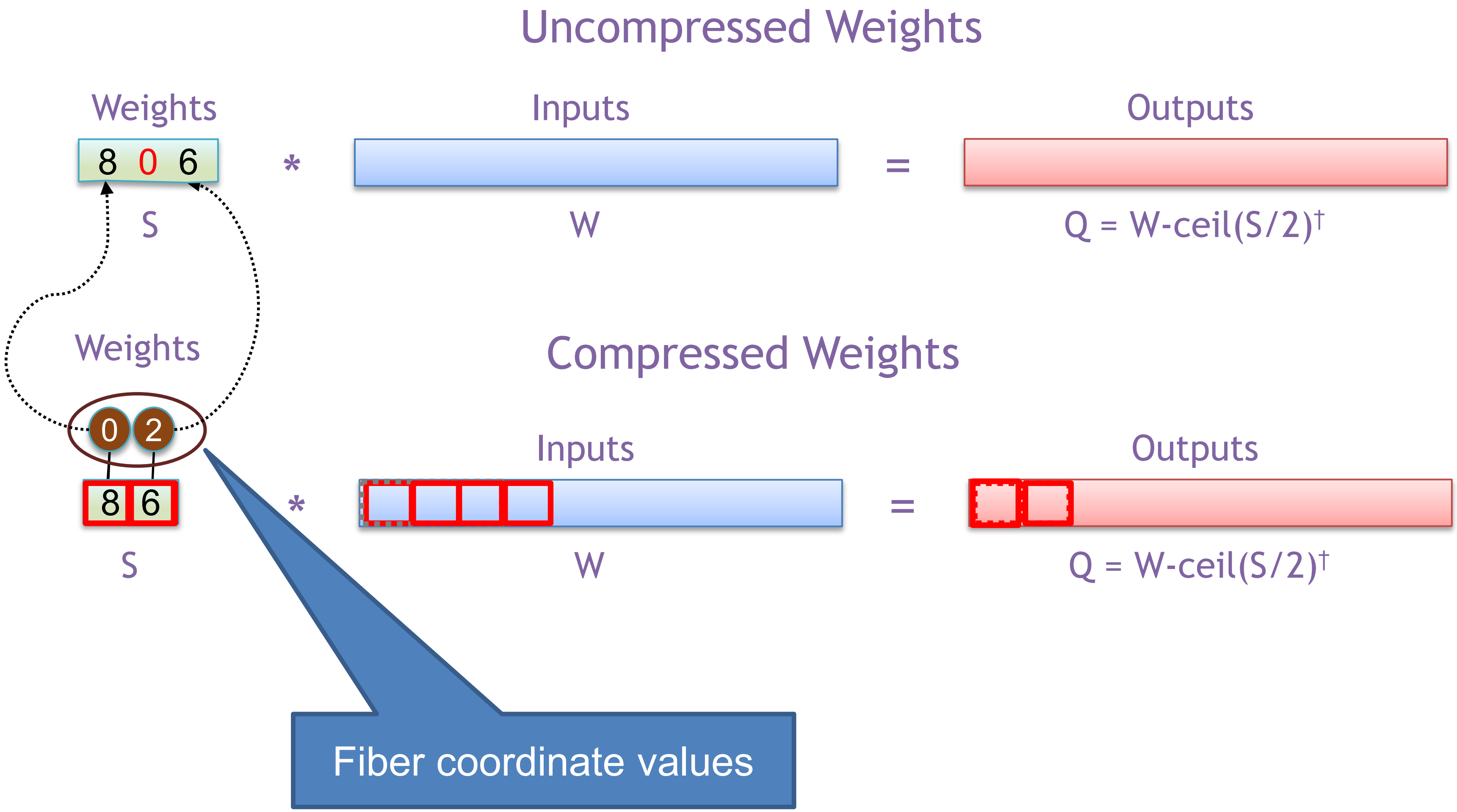
What are the tensor representations of "i" and "f"? Abstract

The variables "s" and "w" are? Coordinates

The variables "q" is? Coordinate and position

Why is this inefficient? No time savings --- ues getPayload()

Output Stationary – Sparse Weights



† Assuming: 'valid' style convolution

Output Stationary – Sparse Weights

```
i = Array(W)          # Input activations
f = Tensor(S)         # Filter weights
o = Array(Q)          # Output activations

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

Concordant traversal

What is “s”? Coordinate

What is “f_val”? Payload, Value

The traversal of “f” will be? Concordant

For sparser weights, this implementation will be? Faster

Output Stationary – Sparse Weights

Tensor: F[S, T]

Rank: T

Rank: S	0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0	0
2	3	3	3	3	3	3	3	3	3	3	3	3

Tensor: I[W, T]

Rank: T

Rank: W	0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	1	1	1	1	1
1	4	4	4	4	4	4	4	4	4	4	4	4
2	2	2	2	2	2	2	2	2	2	2	2	2
3	2	2	2	2	2	2	2	2	2	2	2	2
4	3	3	3	3	3	3	3	3	3	3	3	3
5	1	1	1	1	1	1	1	1	1	1	1	1
6	2	2	2	2	2	2	2	2	2	2	2	2
7	3	3	3	3	3	3	3	3	3	3	3	3

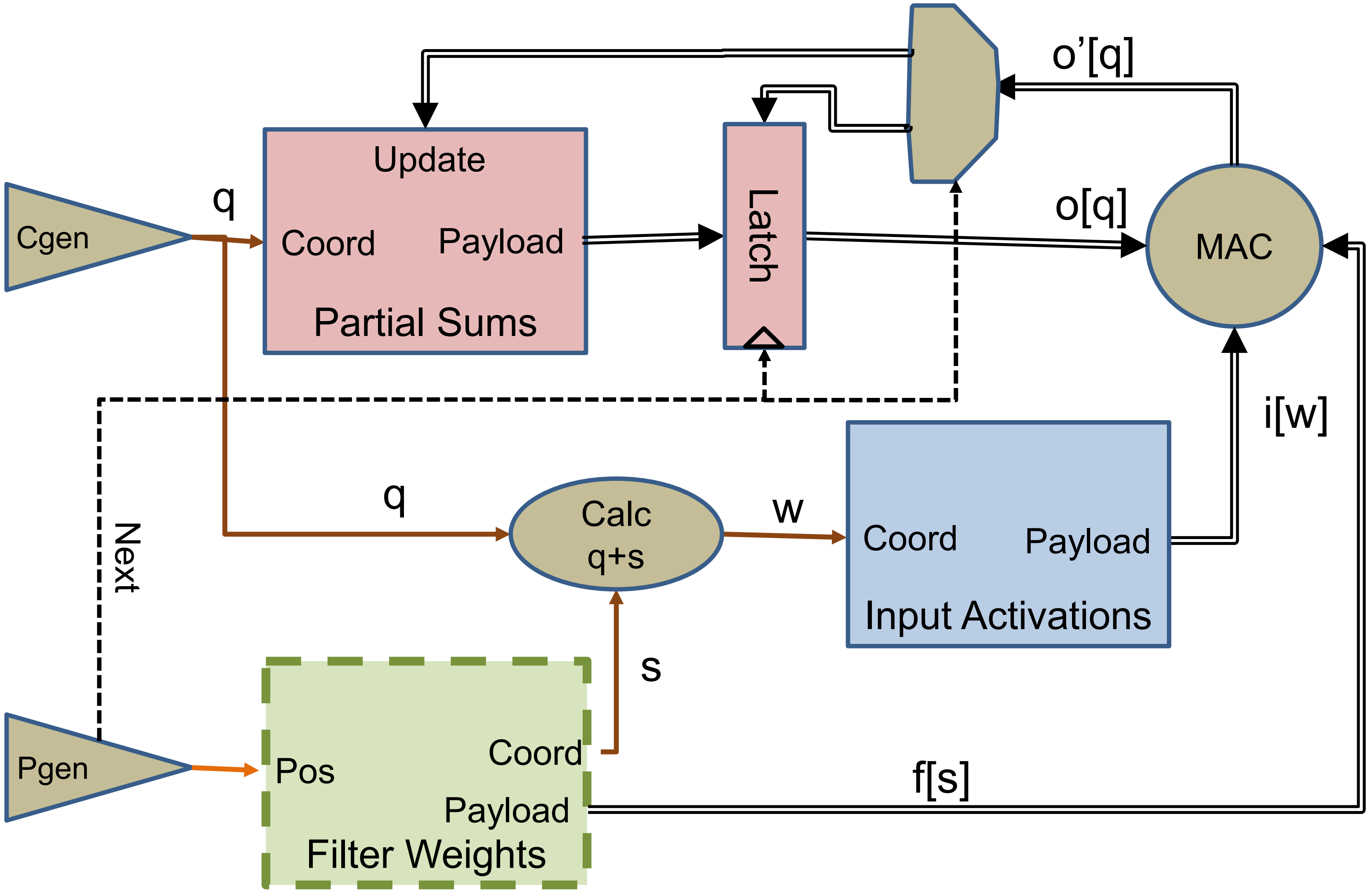
Tensor: O[Q, T]

Rank: T

Rank: Q	0	1	2	3	4	5	6	7	8	9	10	11
0	1	7	7	7	7	7	7	7	7	7	7	7
1	0	0	4	10	10	10	10	10	10	10	10	10
2	0	0	0	0	2	11	11	11	11	11	11	11
3	0	0	0	0	0	0	2	5	5	5	5	5
4	0	0	0	0	0	0	0	0	3	9	9	9
5	0	0	0	0	0	0	0	0	0	0	1	10

Output Stationary – Sparse Weights

Output Stationary – Sparse Weights



Weight Stationary - Sparse Weights

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

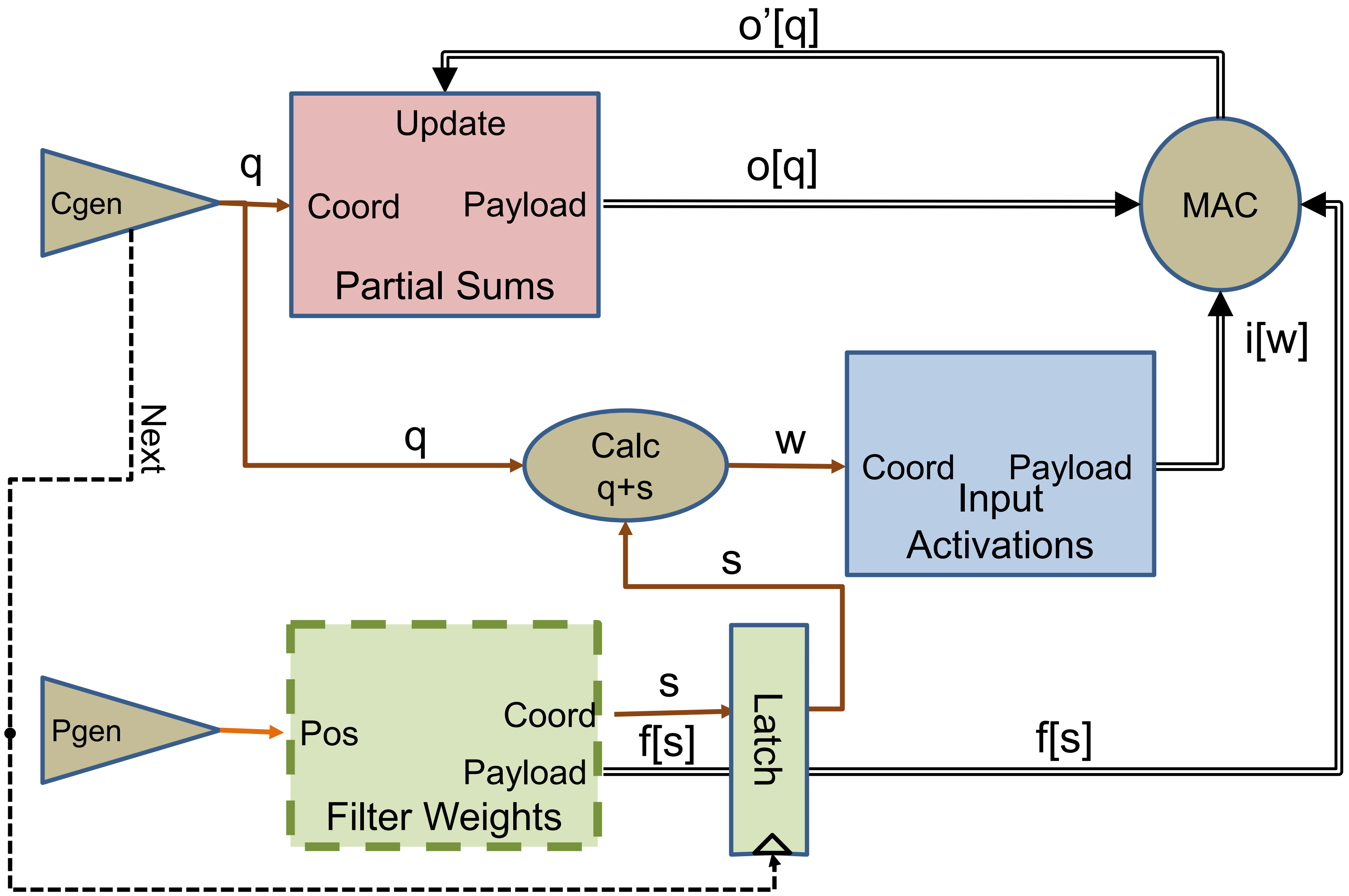
```
i = Array(W)        # Input activations
f = Tensor(S)       # Filter weights
o = Array(Q)        # Output activations

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

Concordant traversal

What dataflow is this?

Weight stationary

Weight Stationary - Sparse Weights



To Extend to Other Dimensions of DNN

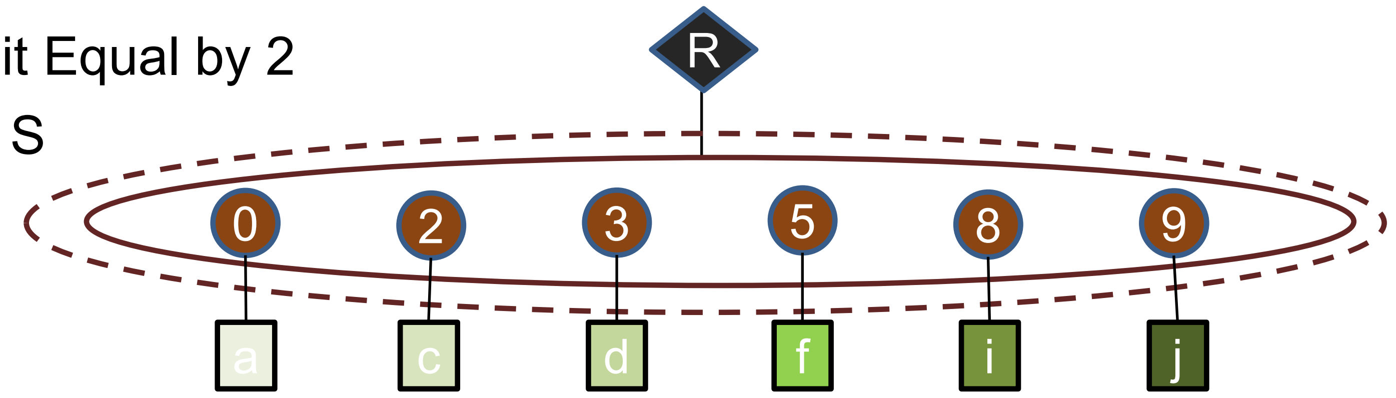
- Need to add loop nests for:
  - 2-D input activations and filters
  - Multiple input channels
  - Multiple output channels

- Add parallelism…

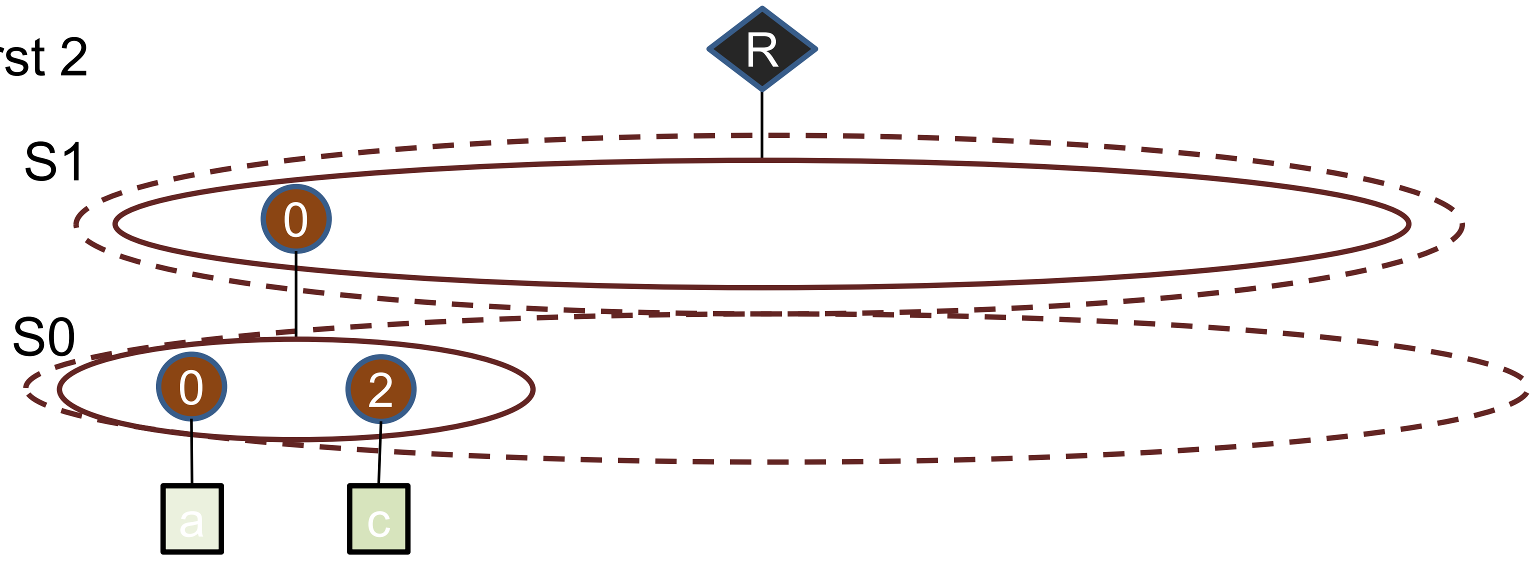
Consider working on two weights at a time

Fiber Splitting Equally in Position Space

Before Split Equal by 2

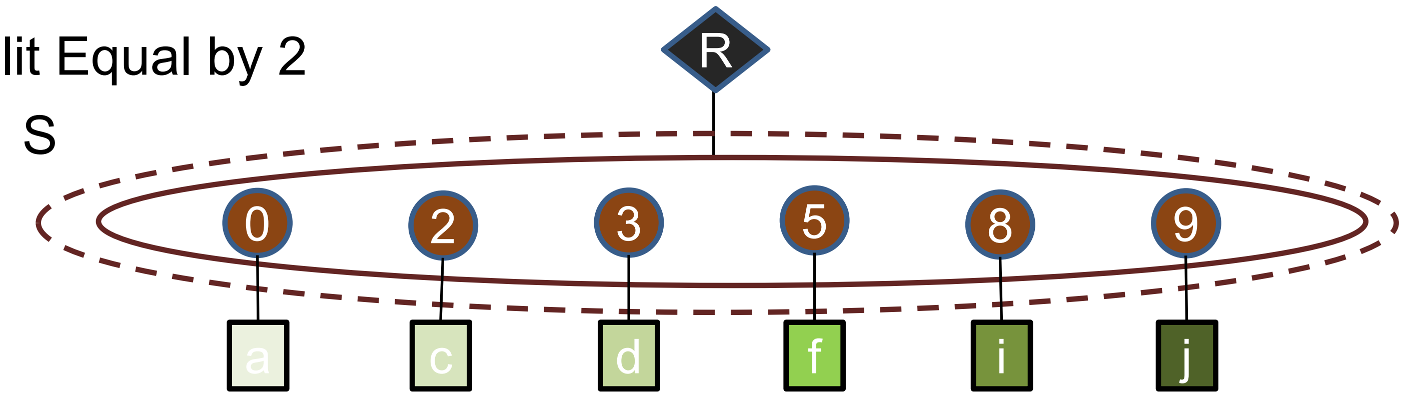


Grab first 2

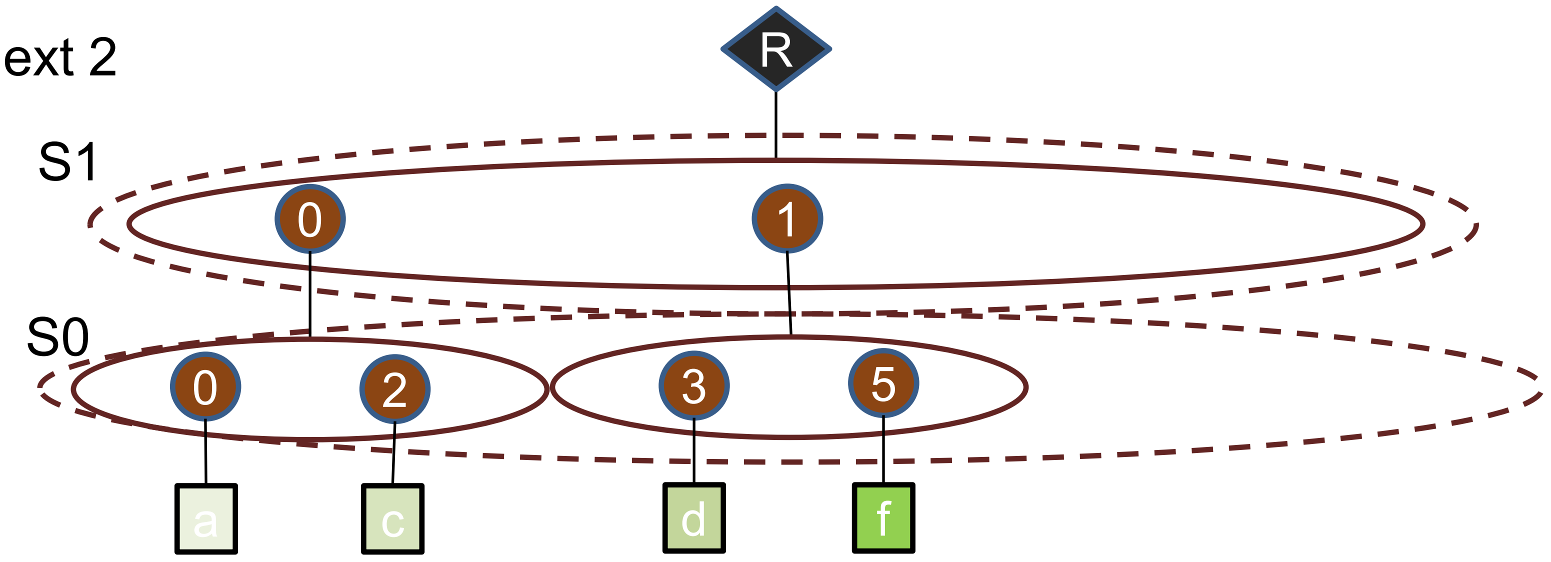


Fiber Splitting Equally in Position Space

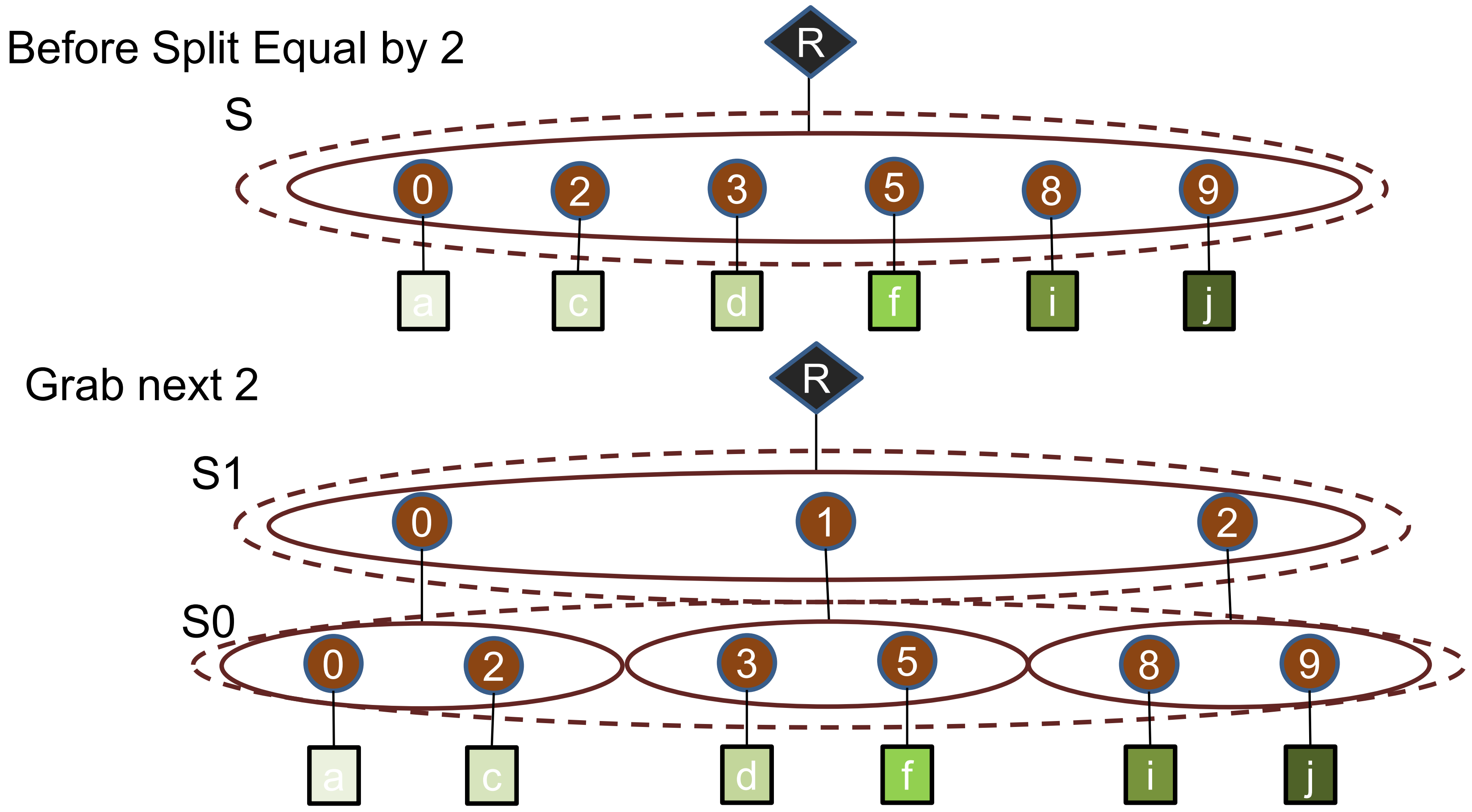
Before Split Equal by 2



Grab next 2



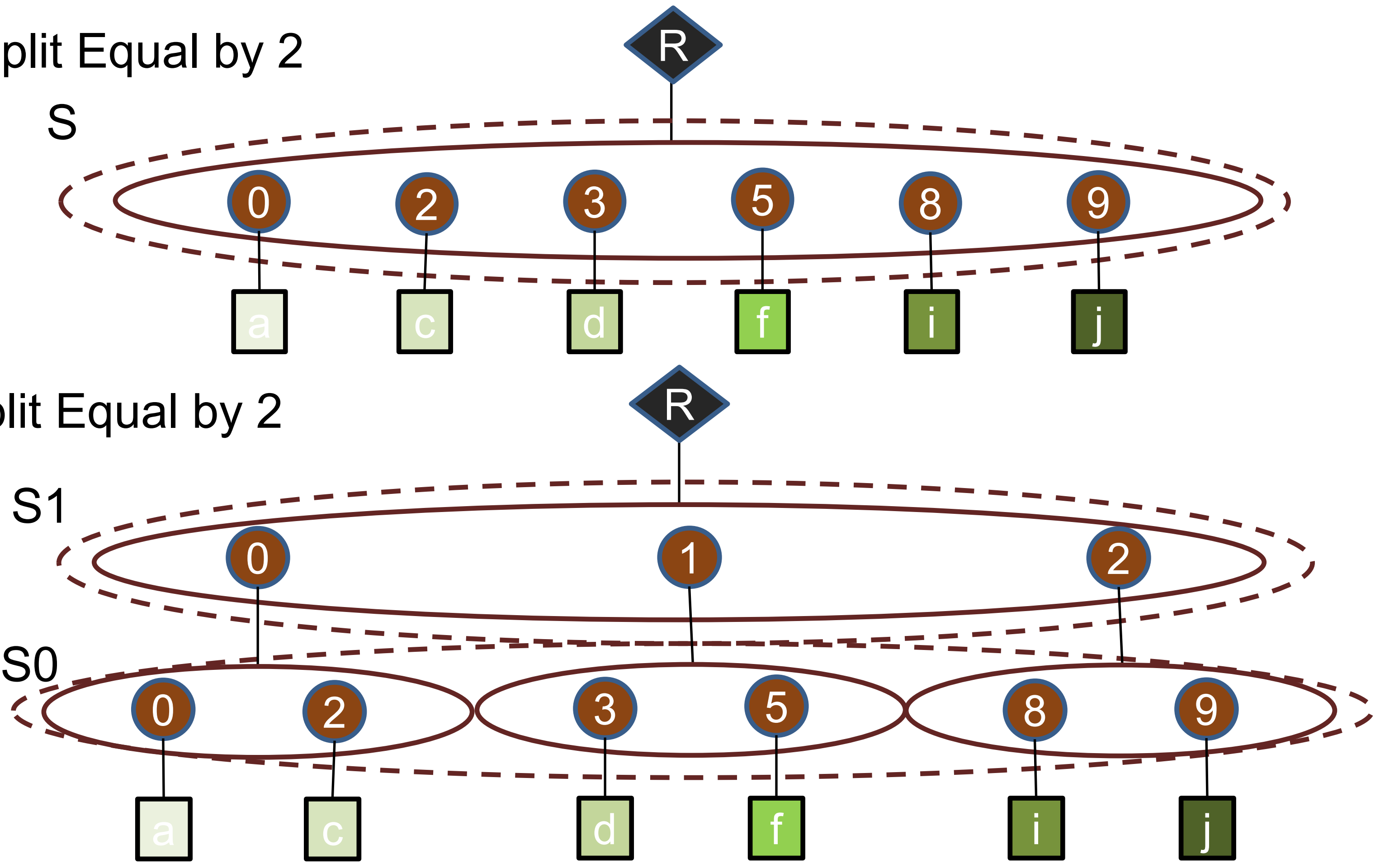
Fiber Splitting Equally in Position Space



Fiber Splitting Equally in Position Space

Before Split Equal by 2

After Split Equal by 2



Complexity for uncompressed fiber?

Low, but doesn't exploit sparsity

… for coordinate/payload list fiber?

Also low, but exploits sparsity

Parallel Weight Stationary - Sparse Weights

```
i = Array(W)          # Input activations
f = Tensor(S)         # Filter weights
o = Array(Q)          # Output activations

for (s1, f_split) in f.splitEqual(2):
  for q1 in [0, Q/4):
     parallel-for (s, f_val) in f_split:
       parallel-for q0 in [0, 4):
         q = q1*4 + q0
         w = q + s
         o[q] += i[w] * f_val
```

Get groups of two weights

Work on two weights in parallel

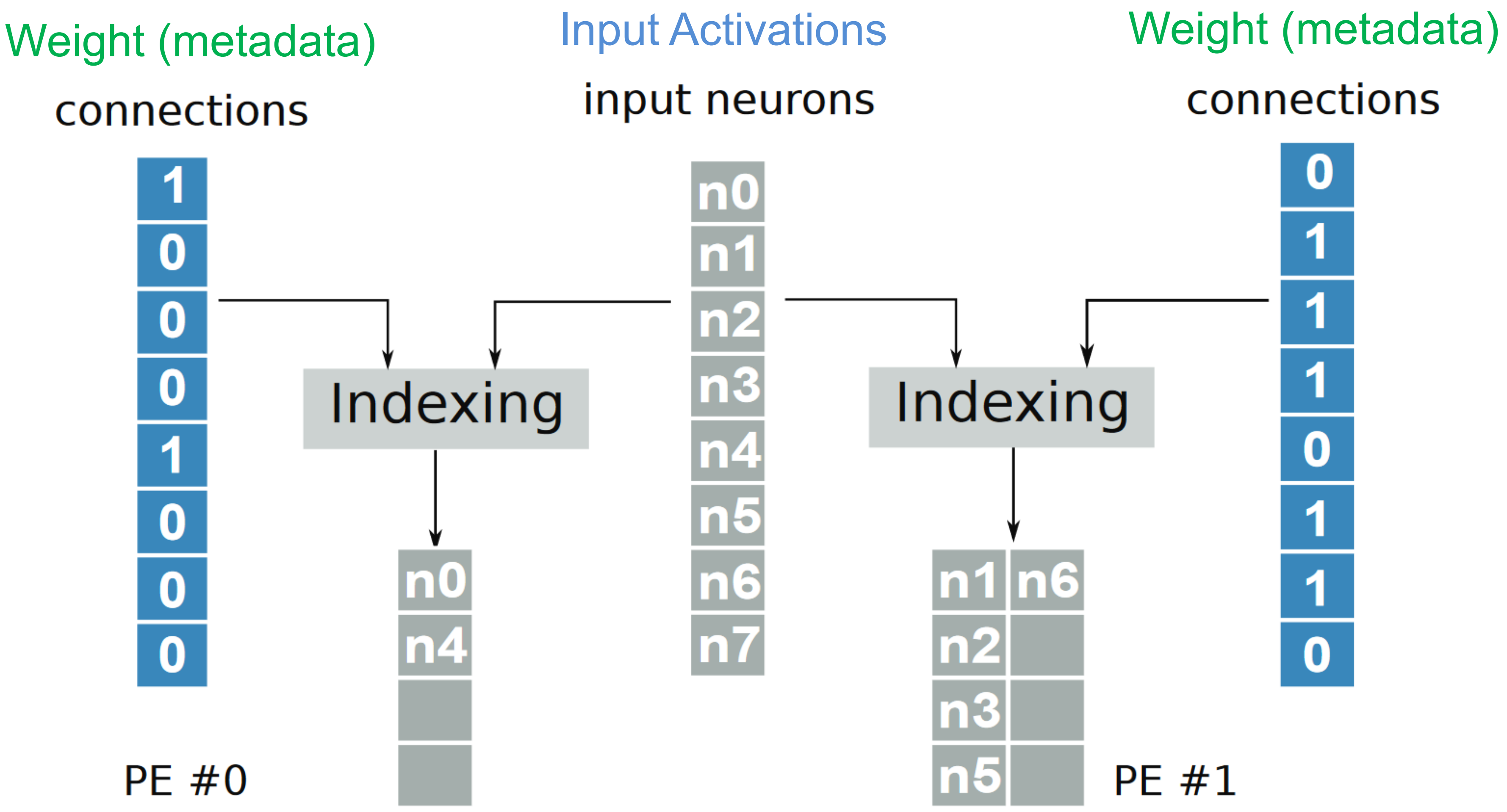
Work on four outputs at once

Calculate coordinates

Accumulate multiple outputs each spatially

Look up input activation

Cambricon-X – Activation Access



Cambricon-X – Zhang et.al., Micro 2016

Parallel Weight Stationary - Sparse Weights

CONV: Exploiting Sparse Inputs

Weight Stationary - Sparse Inputs

$$O_q = I_{q+s} \times F_s \Rightarrow O_{w-s} = I_w \times F_s$$

```
i = Tensor(W)         # Input activations
f = Array(S)          # Filter weights
o = Array(Q)          # Output activations

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

Skipping traversal

Need to restrict input coordinates for the current weight coordinate

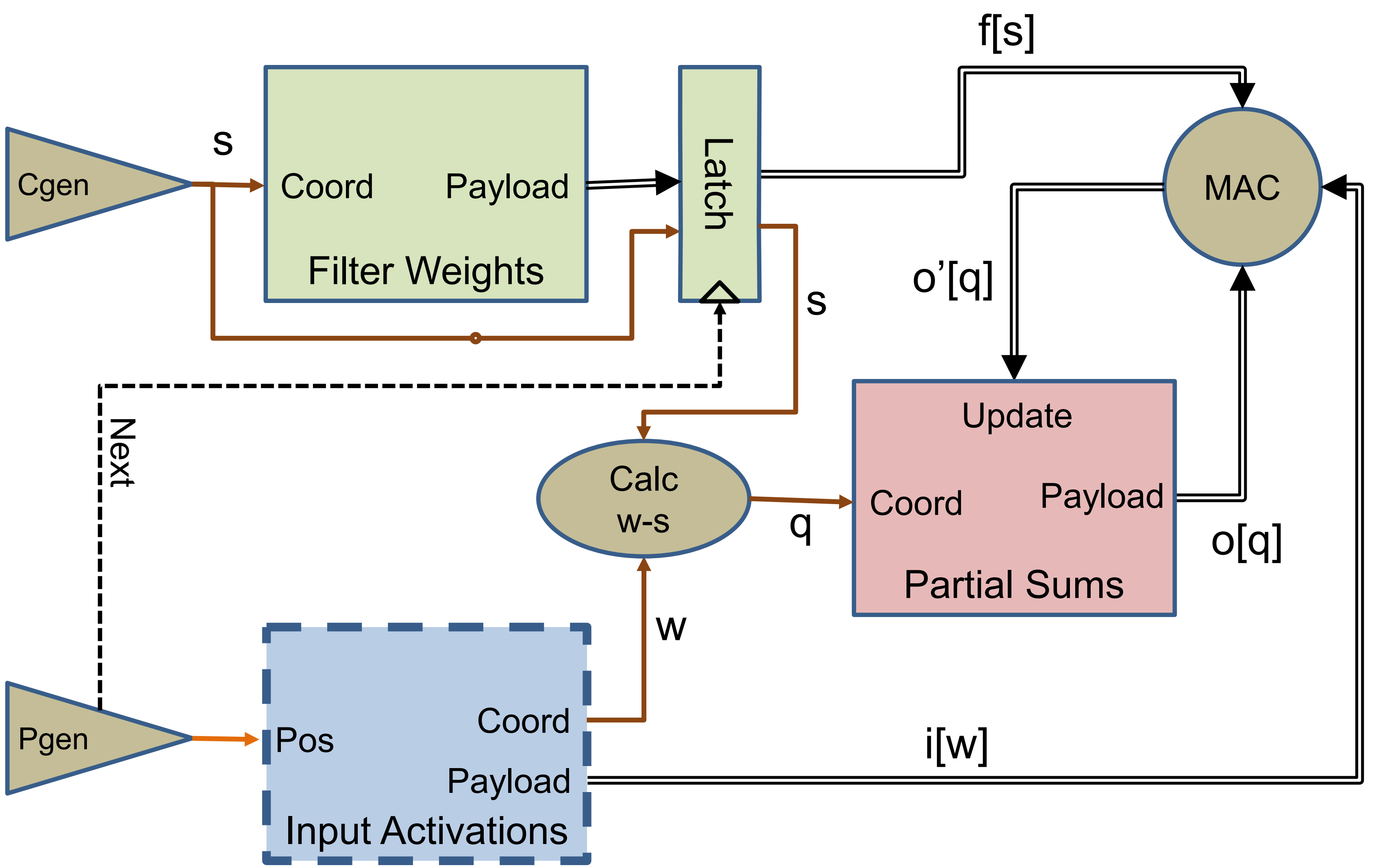
Projection of w and s

Populate

Reduction

Can look up this weight once since it is stationary.

Weight Stationary - Sparse Inputs



Output Stationary - Sparse Inputs

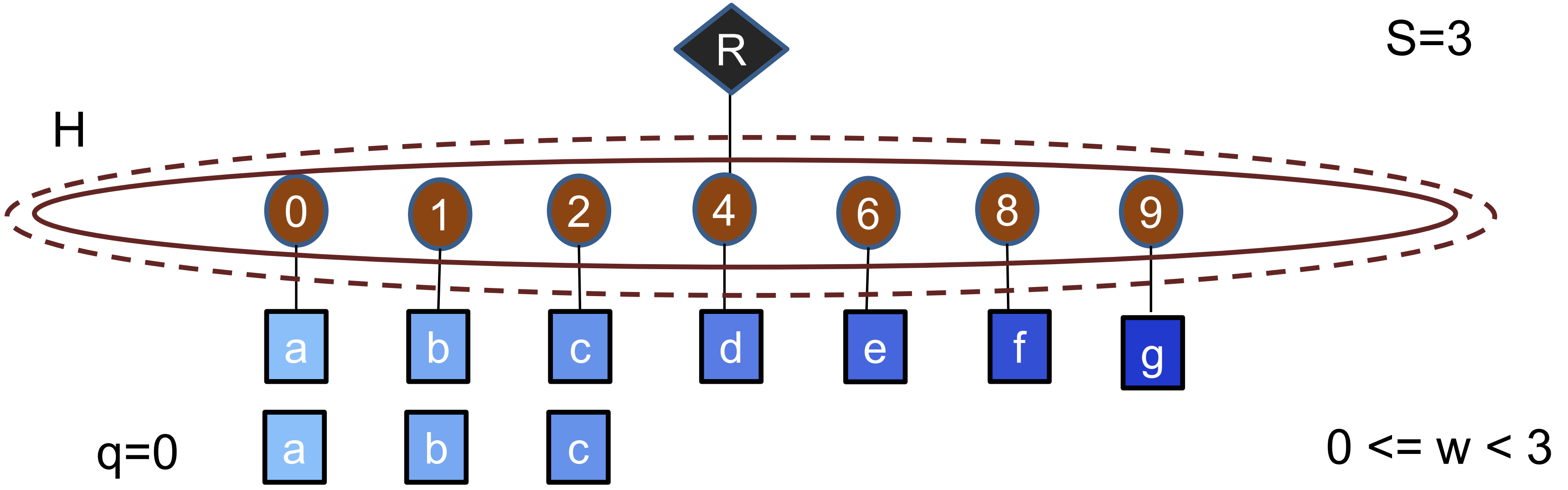
$$O_q = I_{q+s} \times F_s \Rightarrow O_q = I_w \times F_{w-q}$$

```
i = Tensor(W)           # Input activations
f = Array(S)            # Filter weights
o = Array(Q)            # Output activations

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

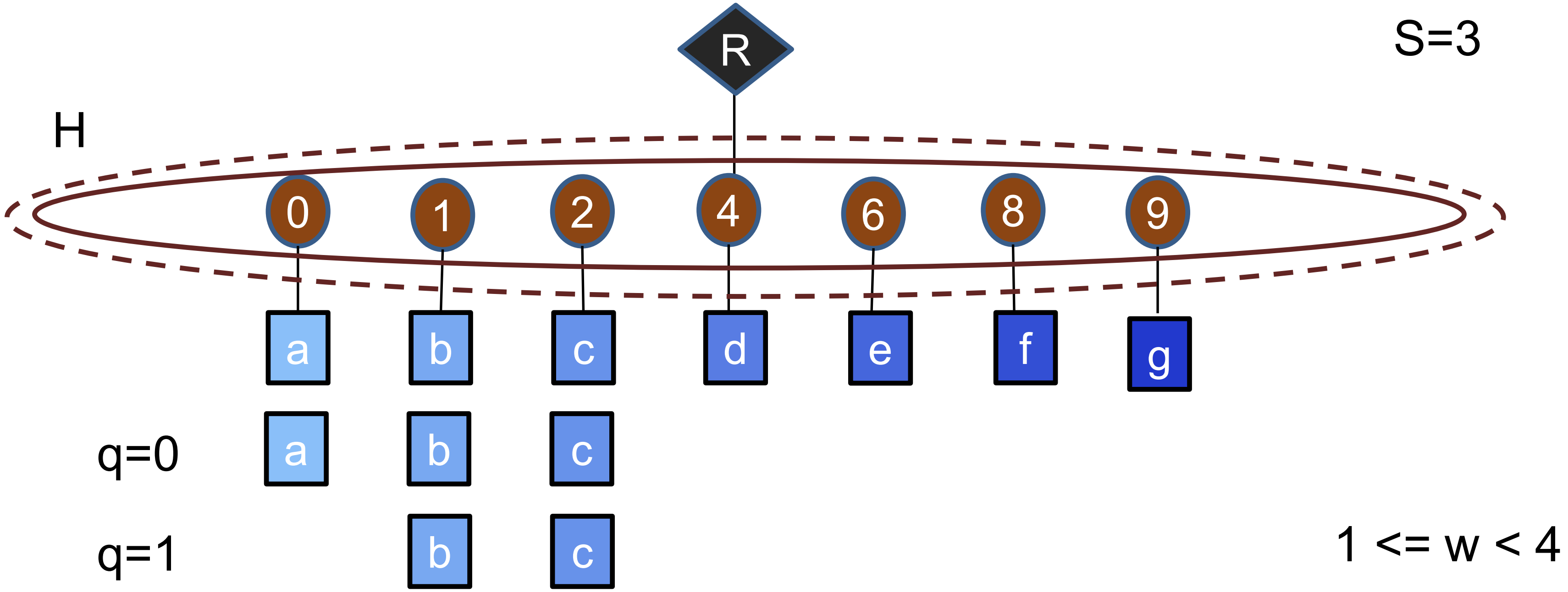
Need to look up a filter weight for each input

Need to restrict input coordinates to the active outputs

Sparse Sliding Window

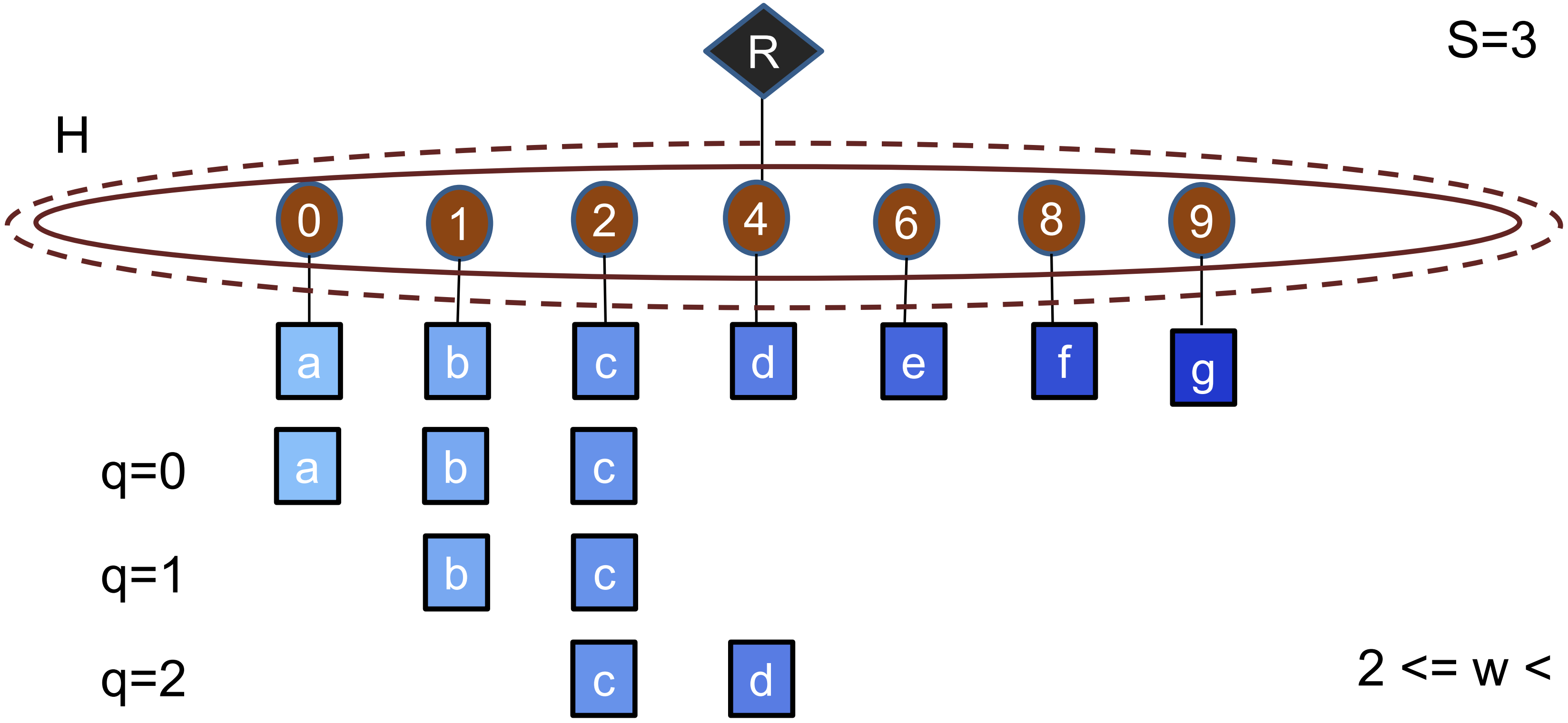


Sparse Sliding Window



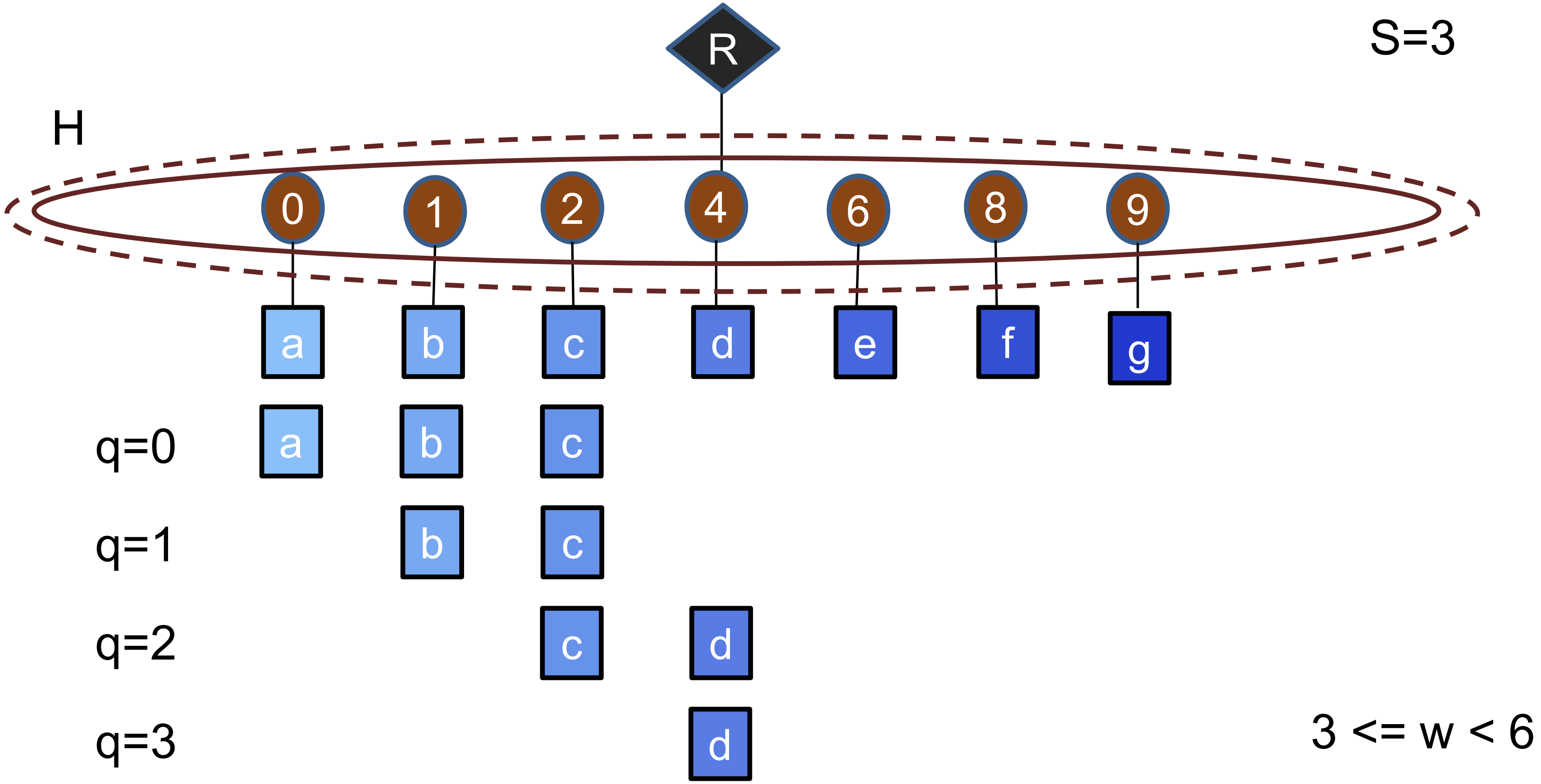
Sparse Sliding Window

S=3

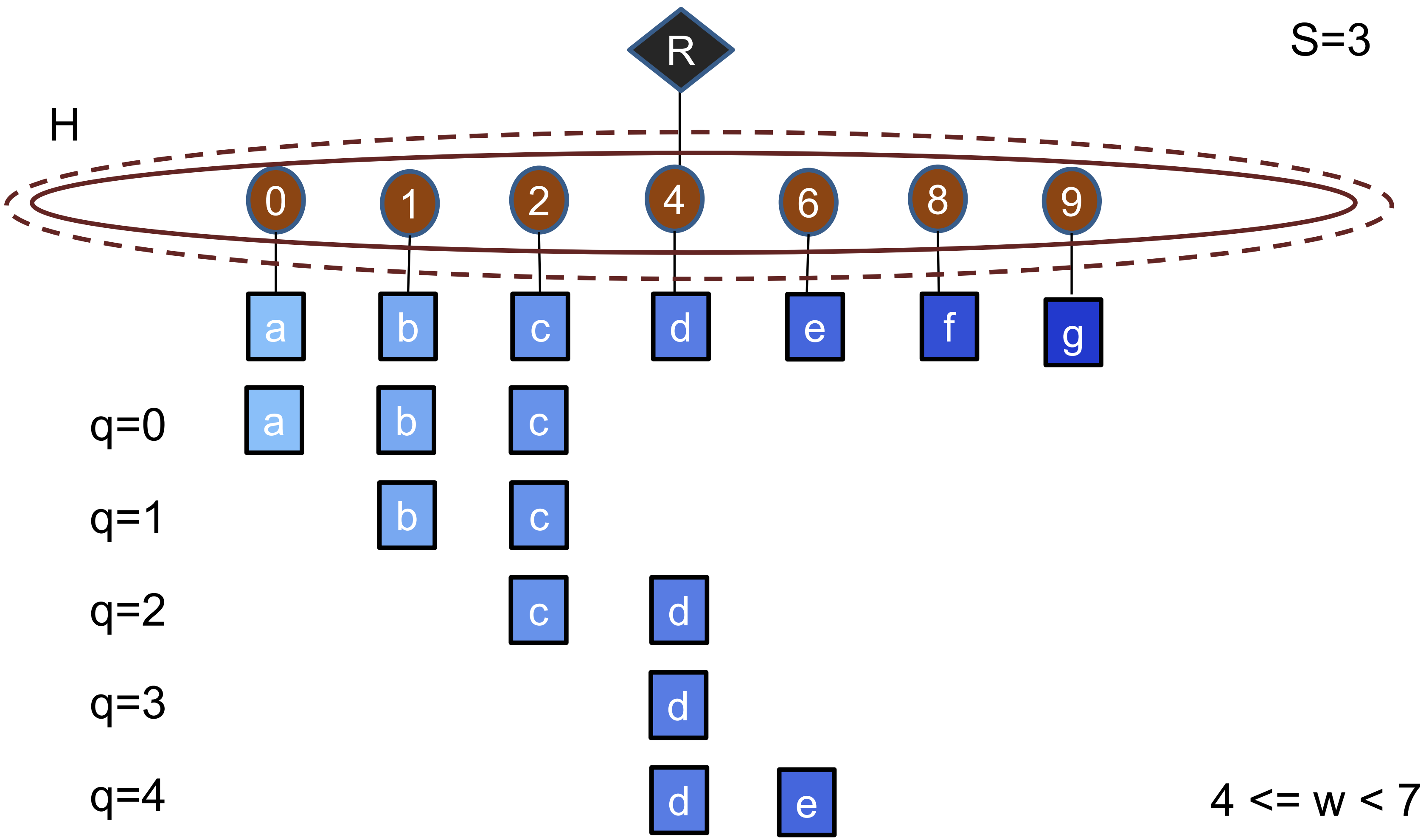


2 <= w < 5

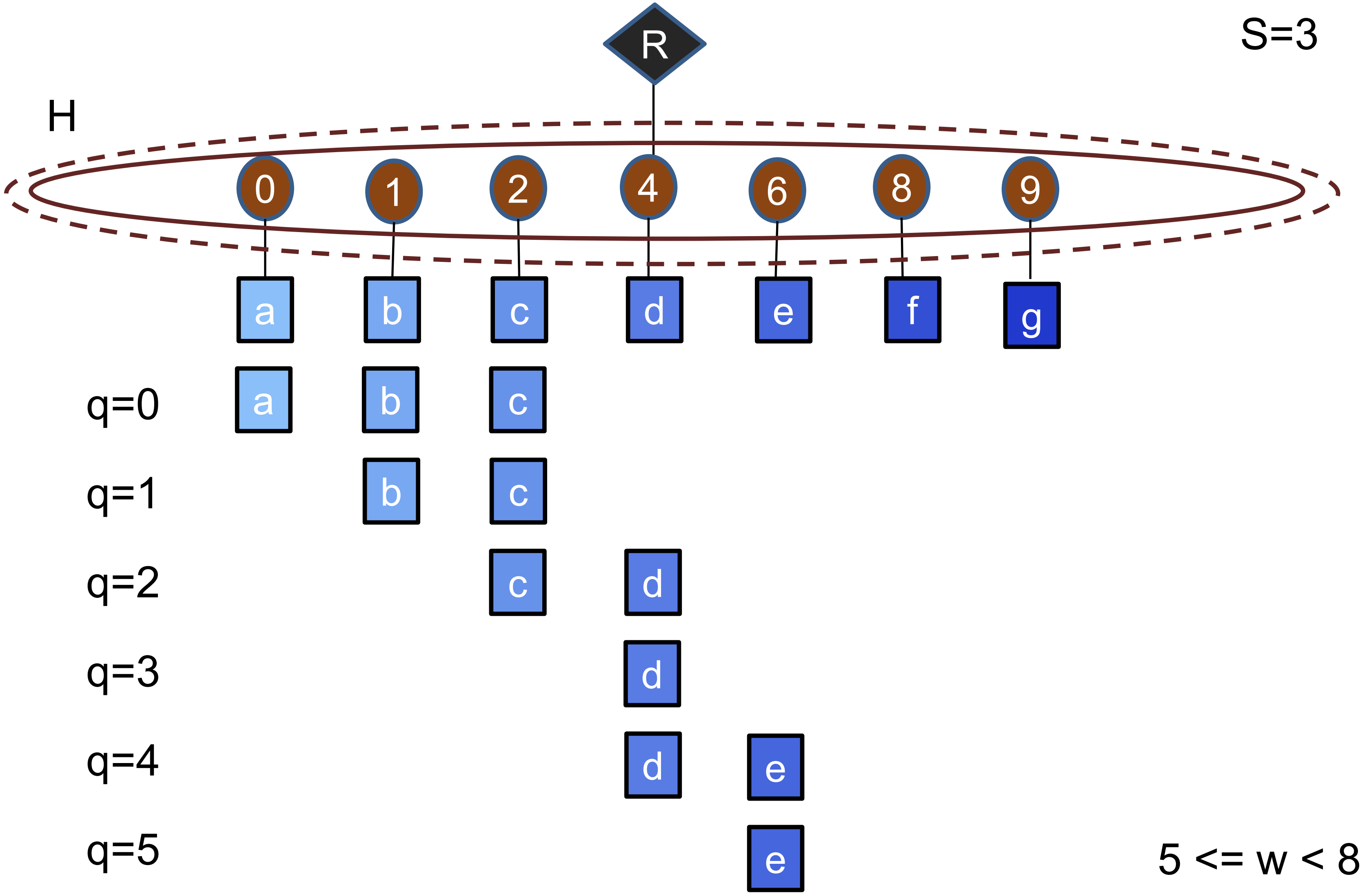
Sparse Sliding Window



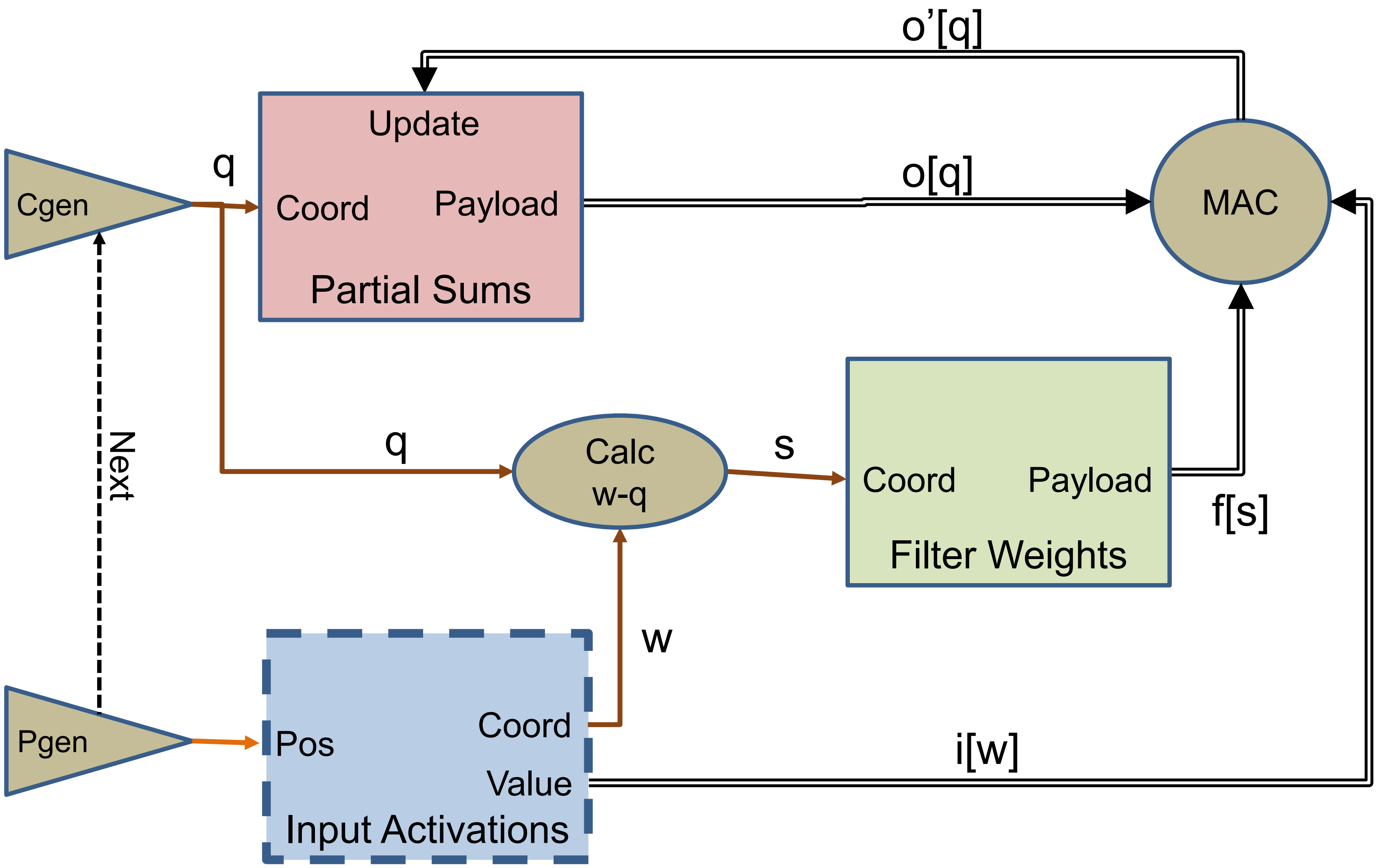
Sparse Sliding Window



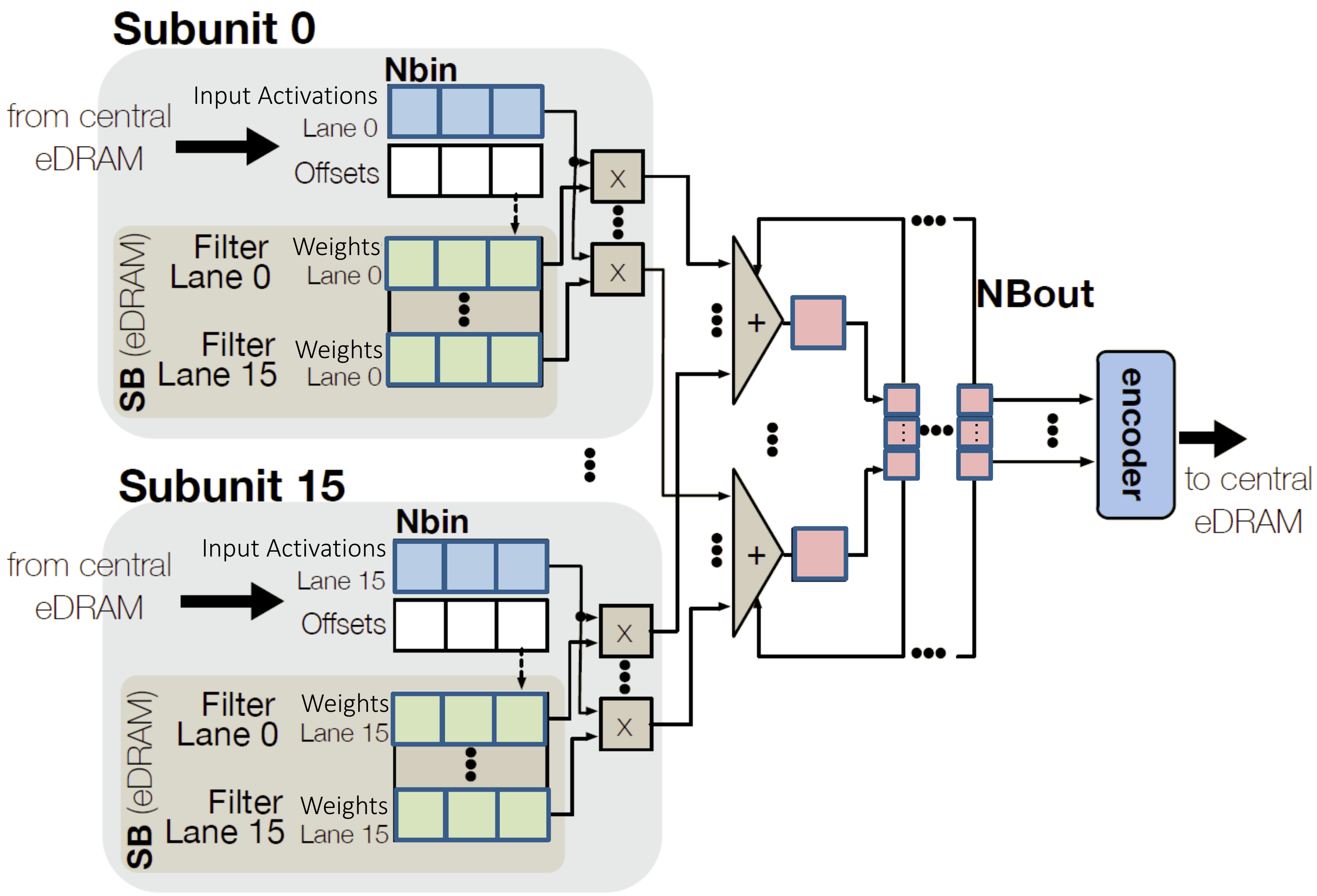
Sparse Sliding Window



Output Stationary - Sparse Inputs



Cnvlutin



Source: CNVLUTIN: Ineffectual-neuron-free DNN computing

Serial Cnvlutin Loop Nest

```
i = Tensor(C,W)        # Input activations
f = Tensor(M,C,S)      # Filter weights
o = Array(Q,M)         # Output activations

for q  in [0, Q]:
  for m, f_c in f:
    for (c, (f_s, i_w)) in f_c & i_c:
      for (w, i_val) in getWindow(i_w, q, S):
        s = w - q
        o[m, q] += i_val * f_s.getPayload(s)
```

Output stationary

Implicit intersection

Irregular sliding window

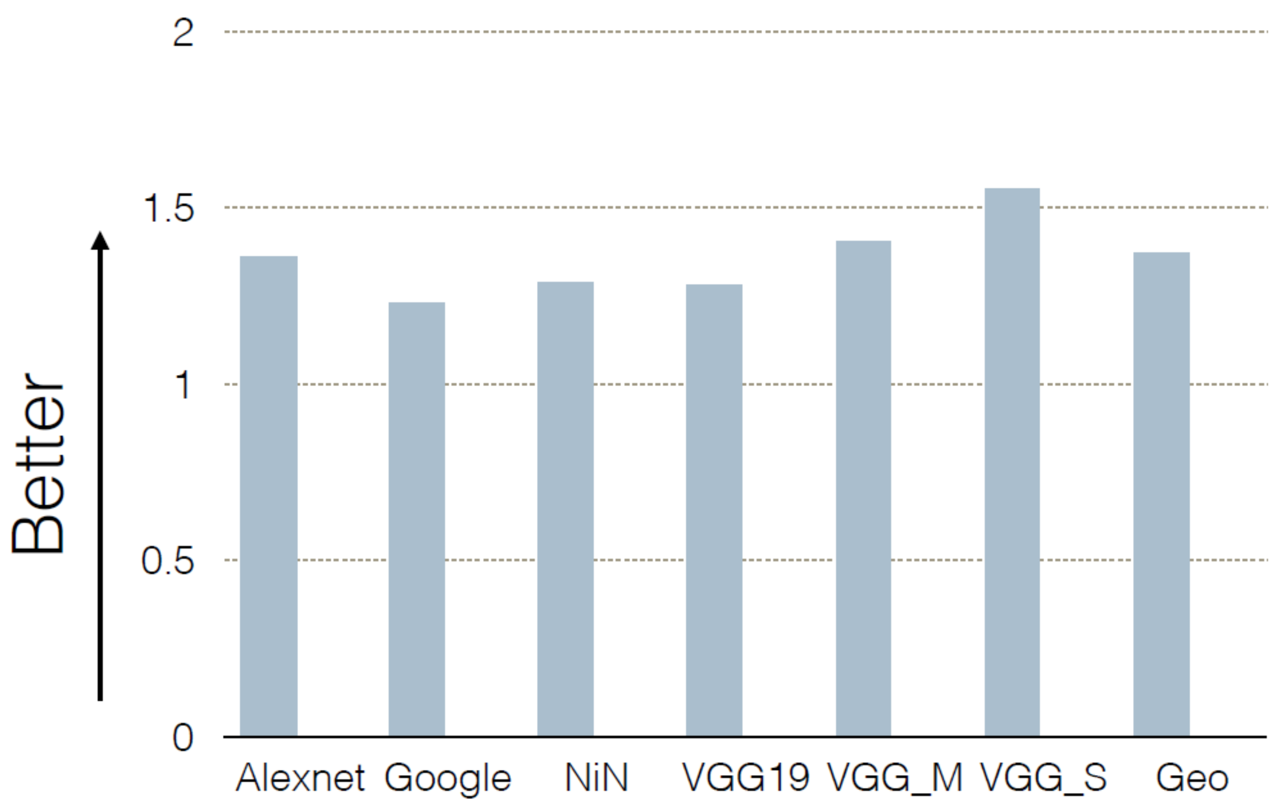
Corresponds to lookup of weight based on current input (and output)

How do we make the getPayload() cheap?

Use uncompressed

More loops needed to show parallel processing of input and output channels

CNVLUTIN - Speedup



Compressing zero activations

Source: CNVLUTIN: Ineffectual-neuron-free DNN computing

Input Stationary - Sparse Weights & Inputs

$$O_q = I_{q+s} \times F_s \quad \Rightarrow \quad O_{w-s} = I_w \times F_s$$

```
i = Tensor(W)          # Input activations
f = Tensor(S)          # Filter weights
o = Array(Q)           # Output activations

for (w, i_val) in i:
  for (s, f_val) in f if w-Q <= s < w:
    q = w - s
    o[q] += i_val * f_val
```

Need to restrict weight coordinates to those relevant to the current input

What dataflow is this?

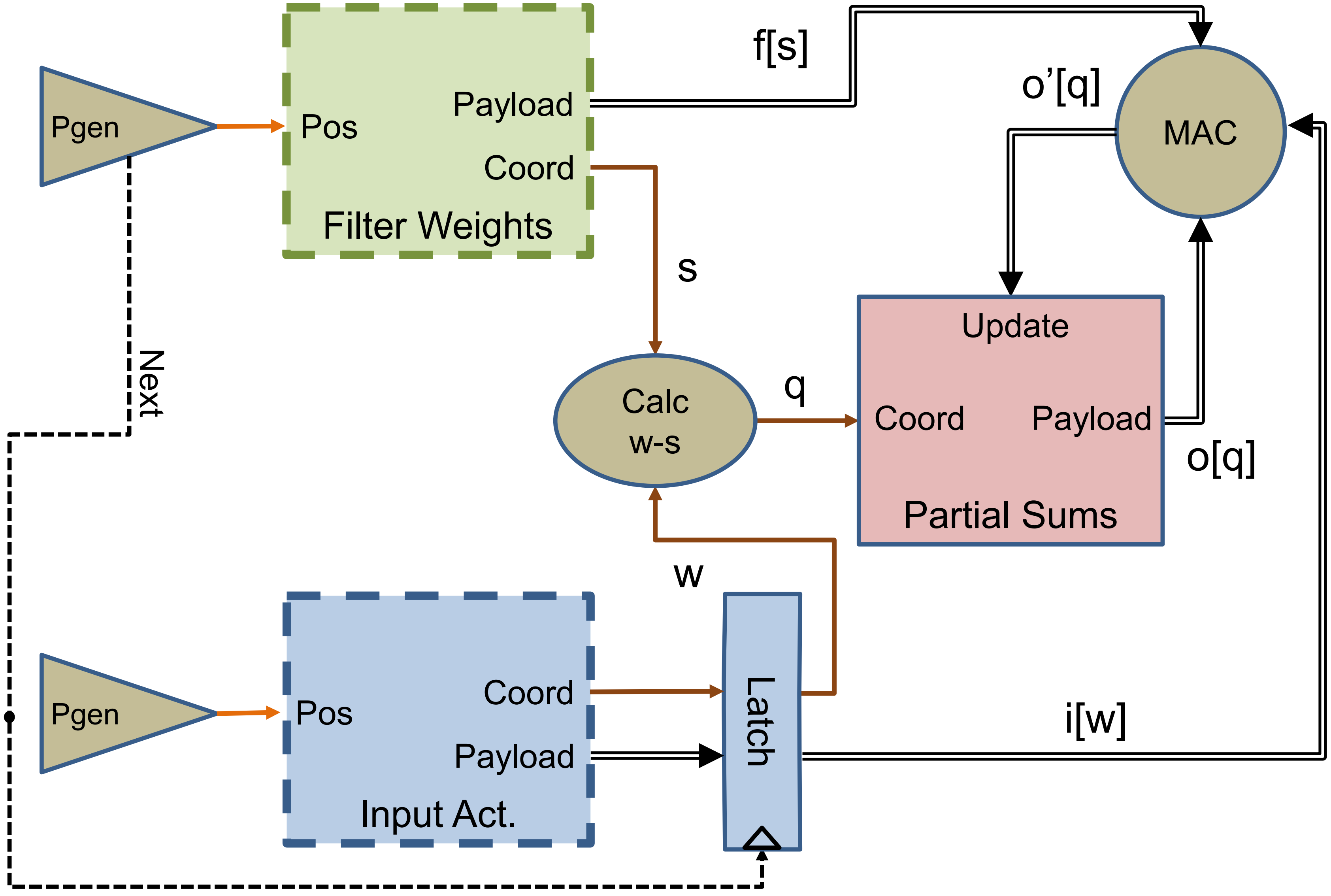
Input stationary

What sparsity can it exploit?

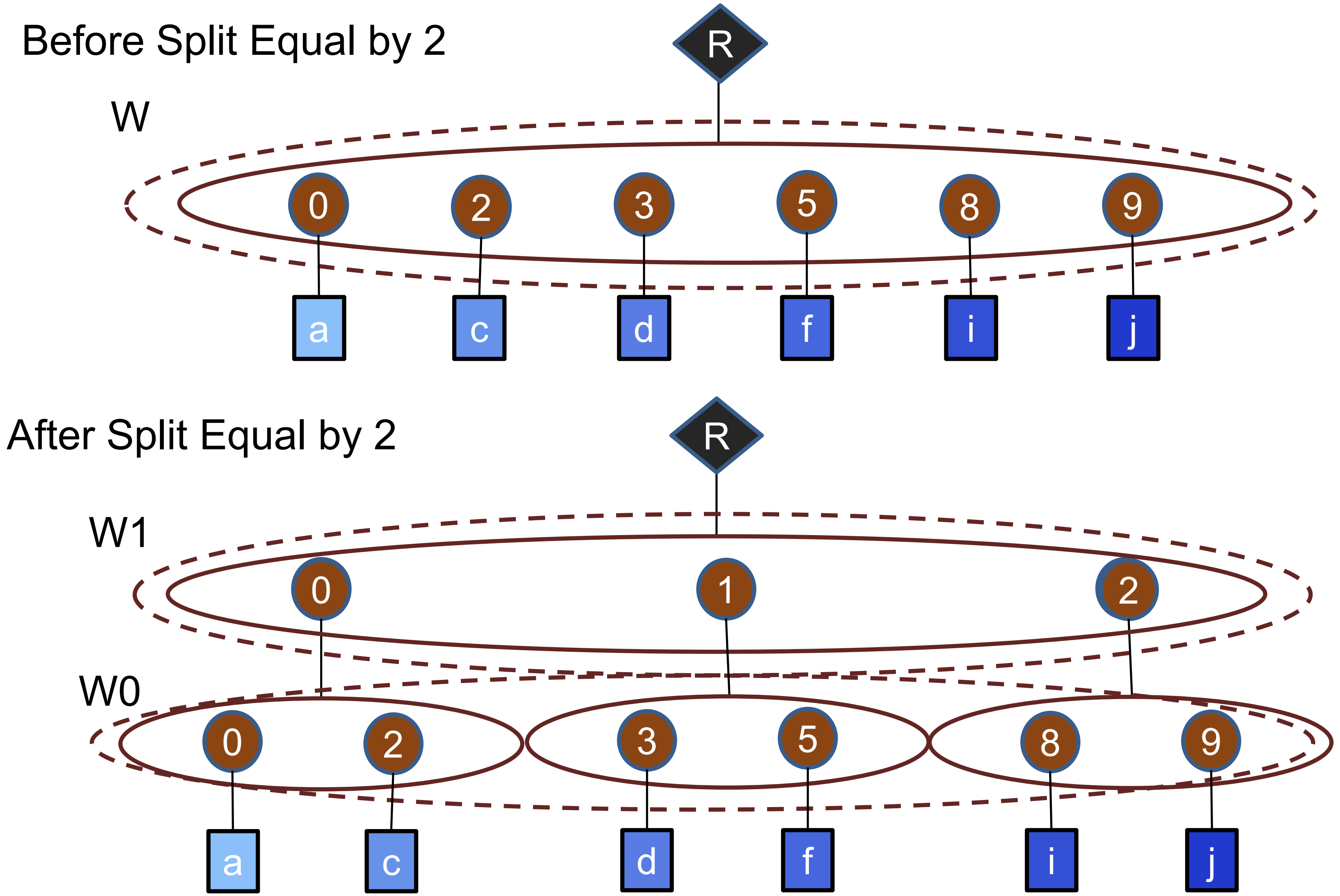
Inputs and Weights

CONV: Exploiting Sparse Inputs & Sparse Weights

Input Stationary - Sparse Weights & Inputs



Fiber Splitting Equally in Position Space



Input Stationary - Sparse Weights & Inputs

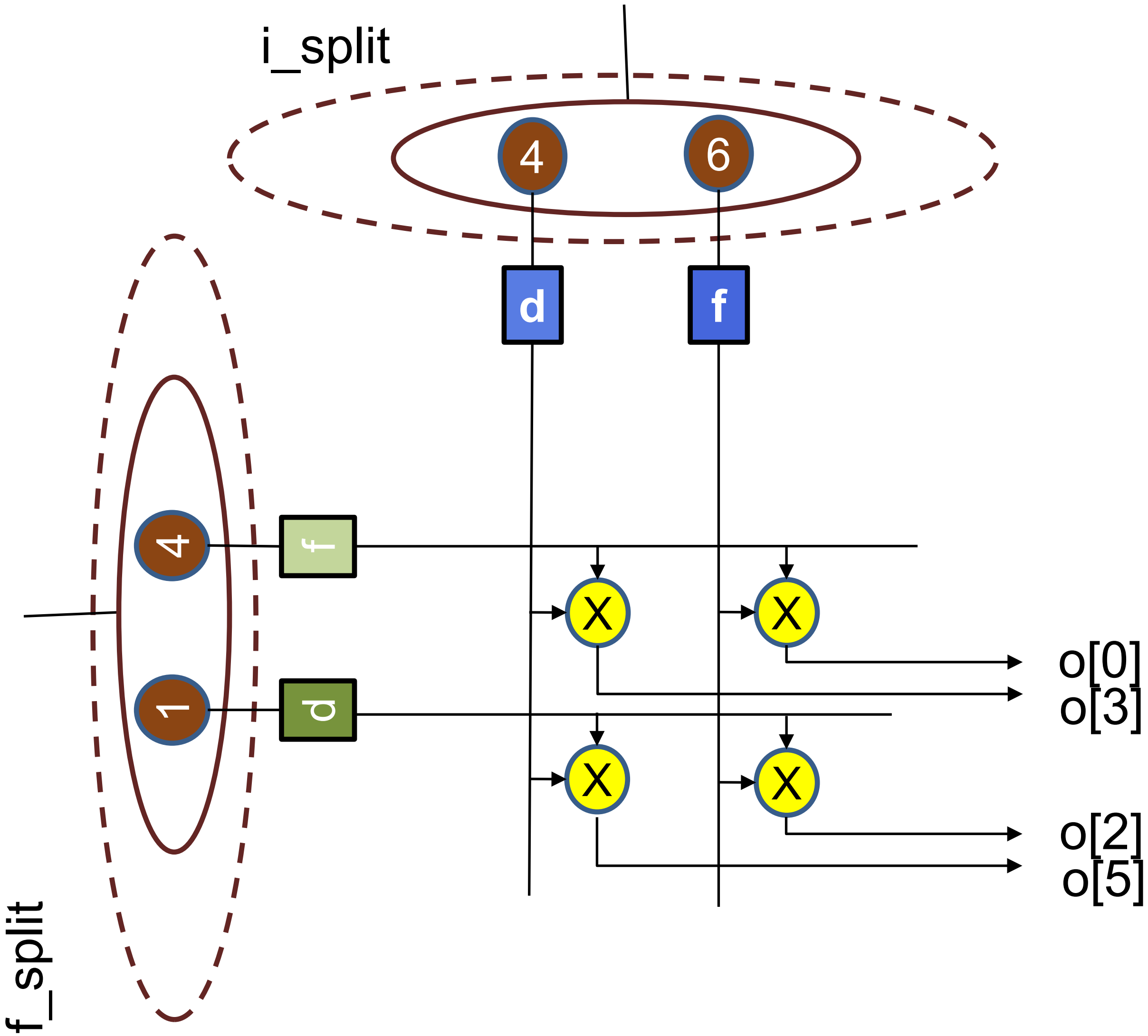
```
i = Tensor(W)         # Input activations
f = Tensor(S)         # Filter weights
o = Array(Q)          # Output activations

for (w1, i_split) in i.splitEqual(2):
  for (s1, f_split) in f.splitEqual(2):
    parallel-for (w0, i_val) in i_split:
      parallel-for (s0, f_val) in f_split if w0-Q <= s0 < w0
      w = w0
      s = s0
      q = w - s
      o[q] += i_val * f_val
```

How many multipliers in this design? 4

Is there a nice pattern to the multipliers' input operands? Yes

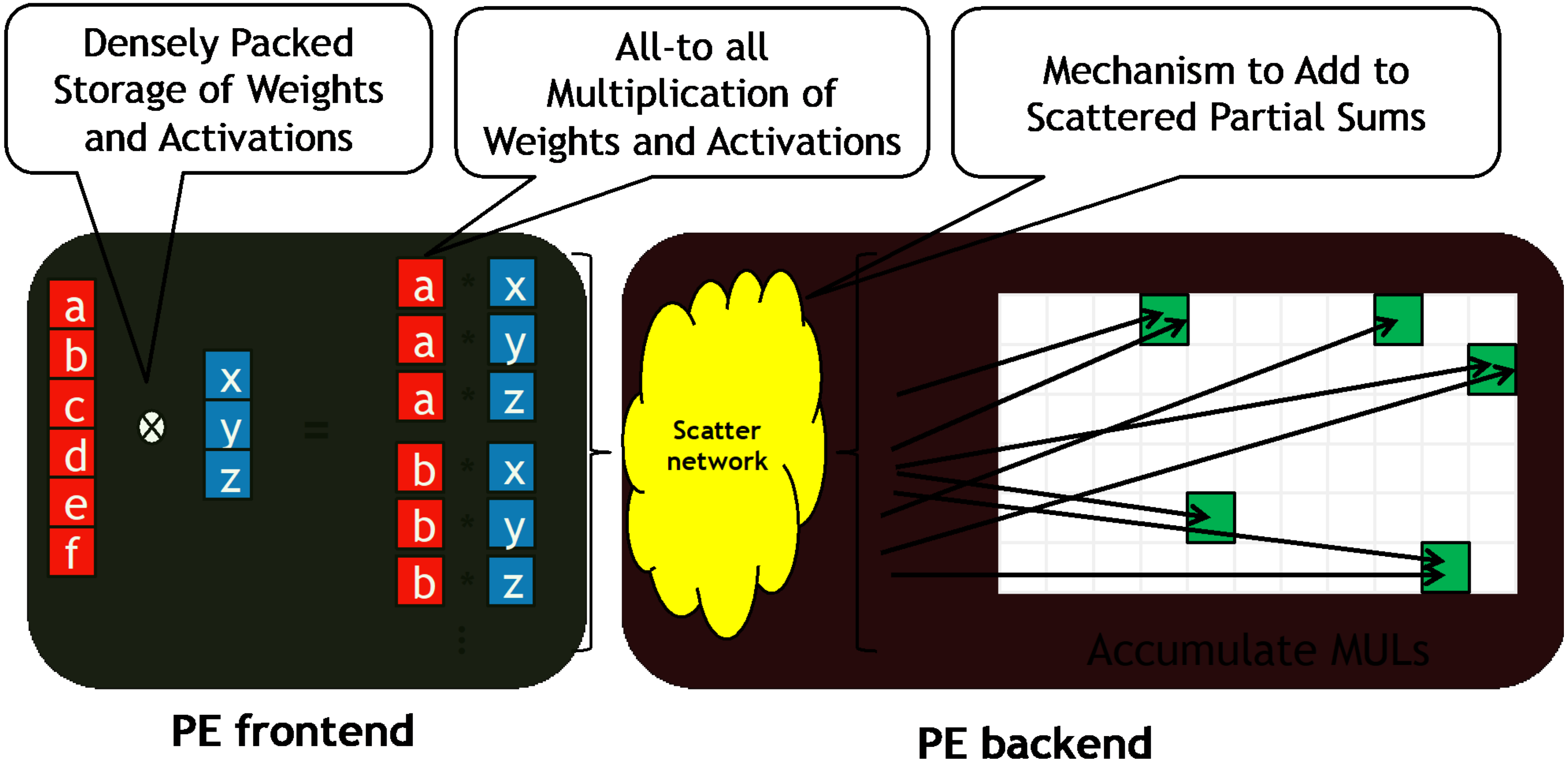
Is there a nice pattern to the multiplier outputs? No

Cartesian Product



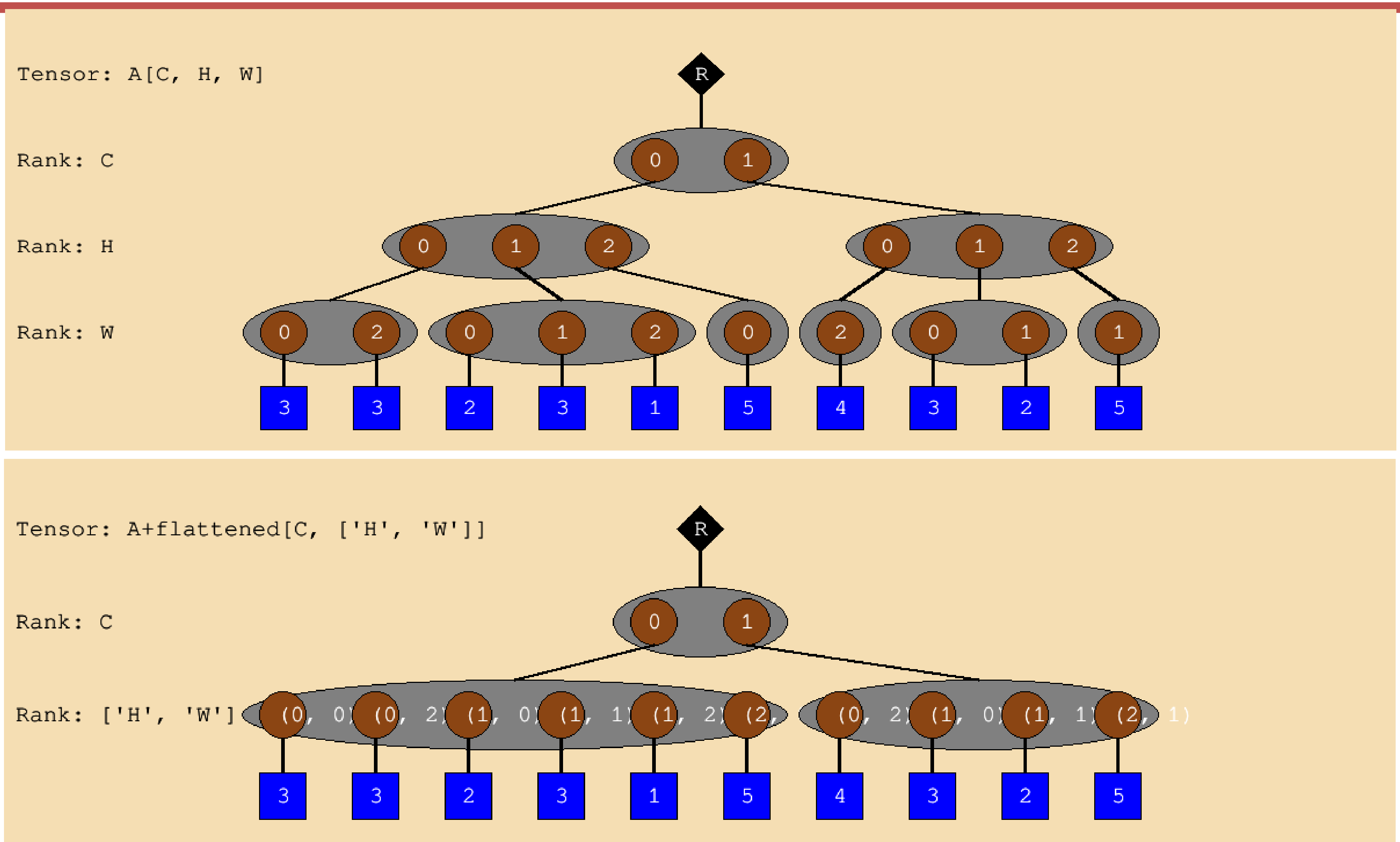
Sparse CNN (SCNN)

Supports Convolutional Layers



Input Stationary Dataflow [Parashar et al., SCNN, ISCA 2017]

Flattening


Tensor: A[C, H, W]
R
Rank: C
Rank: H
Rank: W
Tensor: A+flattened[C, ['H', 'W']]
R
Rank: C
Rank: ['H', 'W']

SCNN Tile – one channel

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

Rearrange indices

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

Flatten

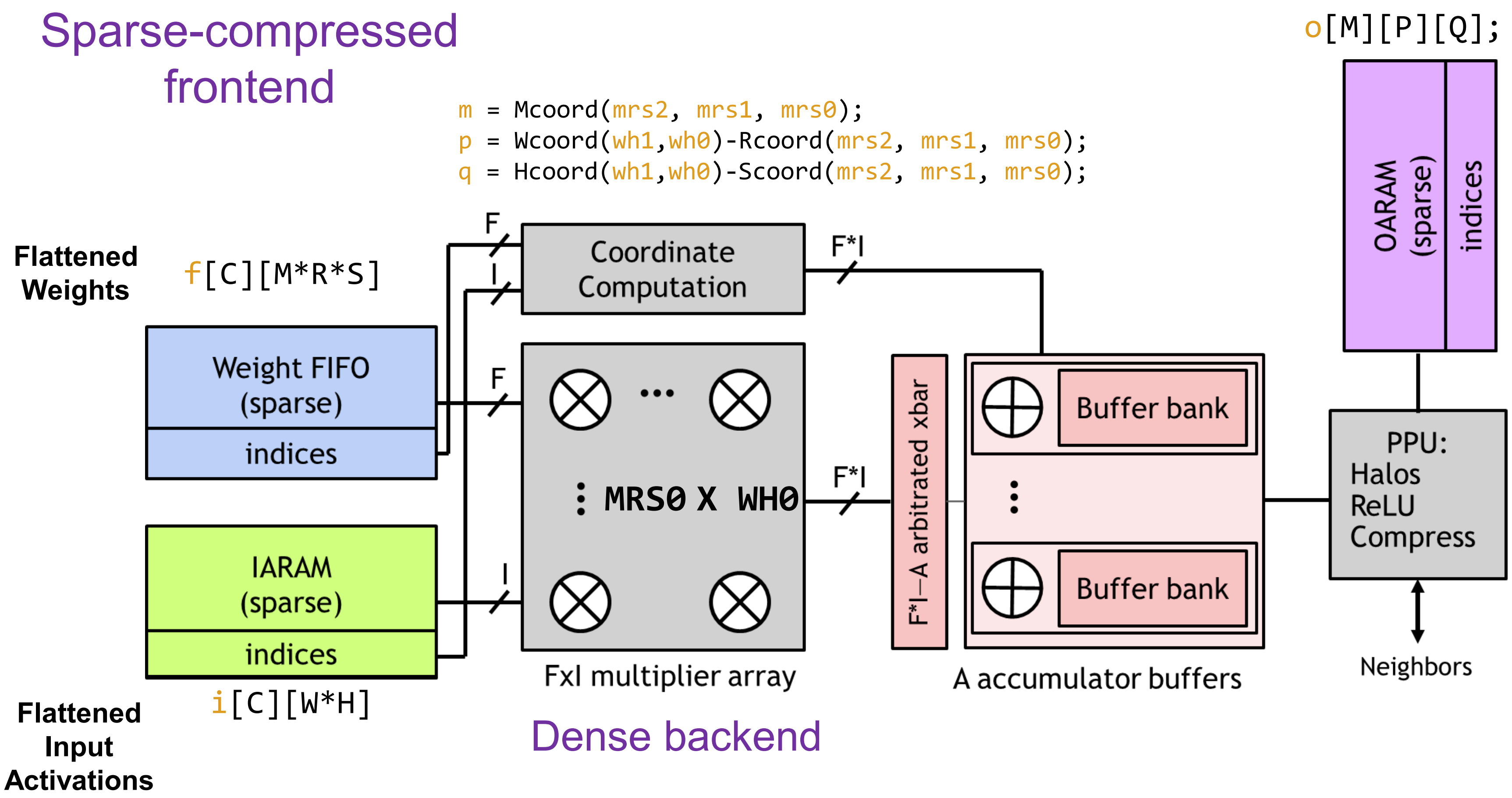
$$O_{m,h-r,w-s} = I_{hw} \times F_{mrs}$$

SCNN Tile – one channel

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

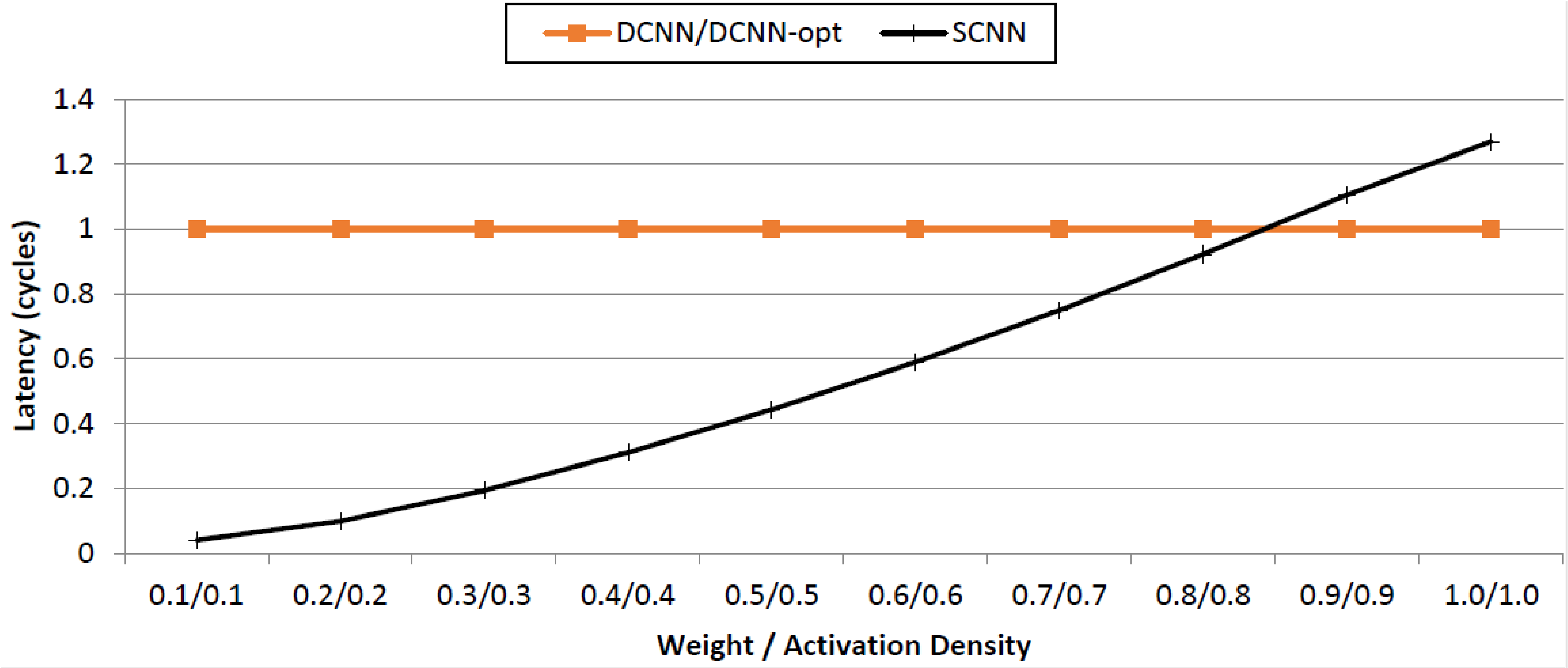
```
i = Tensor(HW)       # Input activations
f = Tensor(MRS)      # Filter weights
o = Array(M,P,Q)     # Output activations

for (hw1, i_split) in i.splitEqual(4):
  for (mrs1, f_split) in f.splitEqual(4):
    parallel-for (((h,w), i_val) in i_split:
      parallel-for ((m,r,s), f_val) in f_split if “legal”
      p = h - r
      q = w - s
      o[m,p,q] += i_val * f_val
```

SCNN PE microarchitecture



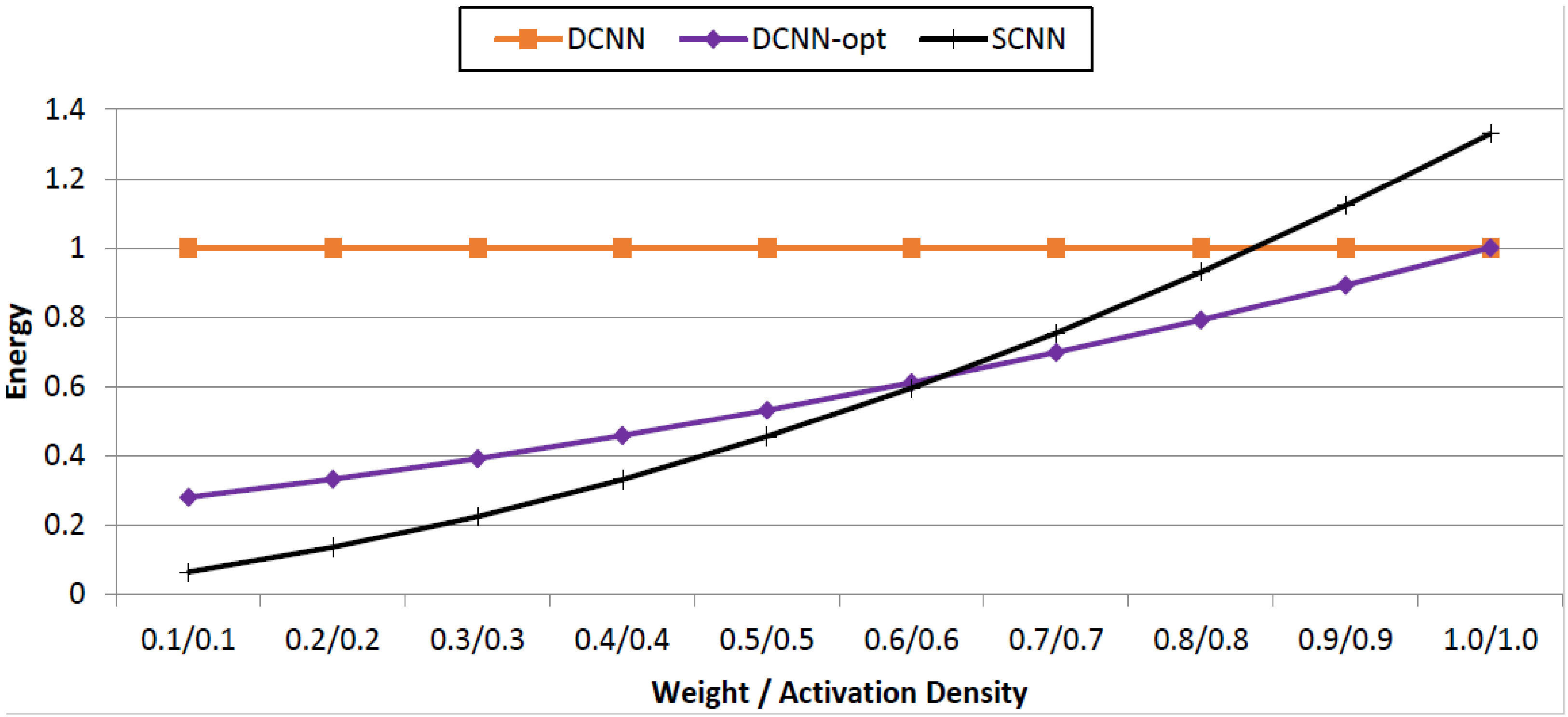
[Parashar et al., SCNN, ISCA 2017]

SCNN Latency Versus Density



[Parashar et al., SCNN, ISCA 2017]

SCNN Energy Versus Density



[Parashar et al., SCNN, ISCA 2017]

Weight Stationary - Sparse Weights & Inputs

```
i = Tensor(W)          # Input activations
f = Tensor(S)          # Filter weights
o = Array(Q)           # Output activations

for (s1, f_split) in f.splitEqual(2):
 for (w1, i_split) in i.splitEqual(2):
   parallel-for (w0, i_val) in i_split:
      parallel-for (s0, f_val) in f_split if w0-Q <= s0 < w0
      w = w0
      s = s0
      q = w - s
      o[q] += i_val * f_val
```

Loops reversed

Do you see any disadvantage to this design?

Yes, more frequent read from larger buffer

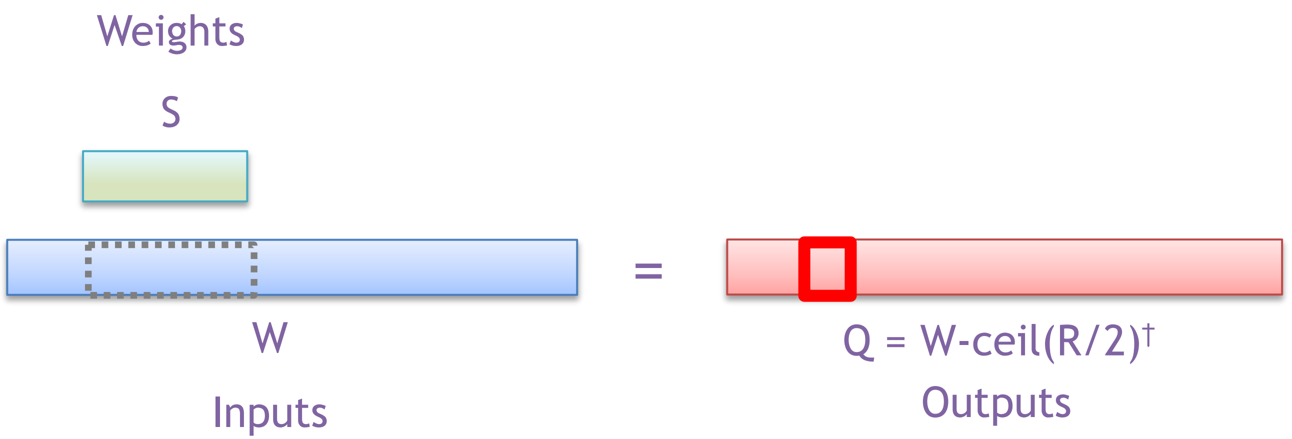
Output Stationary - Sparse Weights & Inputs

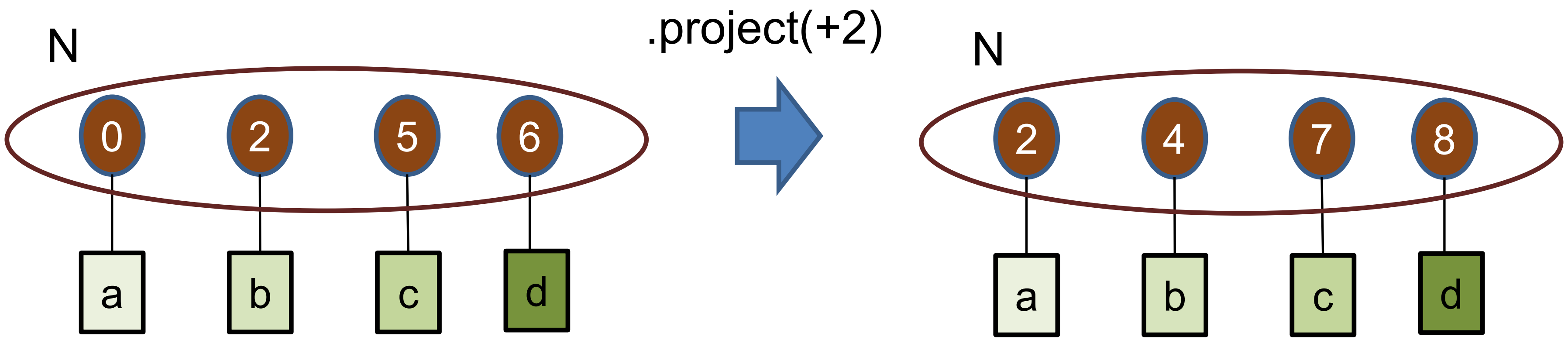
$$O_q = I_{q+s} \times F_s$$

```
i = Tensor(W)          # Input activations
f = Tensor(S)          # Filter weights
o = Array(Q)           # Output activations

for q in [0,Q):
  for (s, (f_val, i_val)) in f.project(+q) & i:
    o[q] += i_val * f_val
```

Need to work on a series of pairs of weights and inputs

Fiber Coordinate Projection

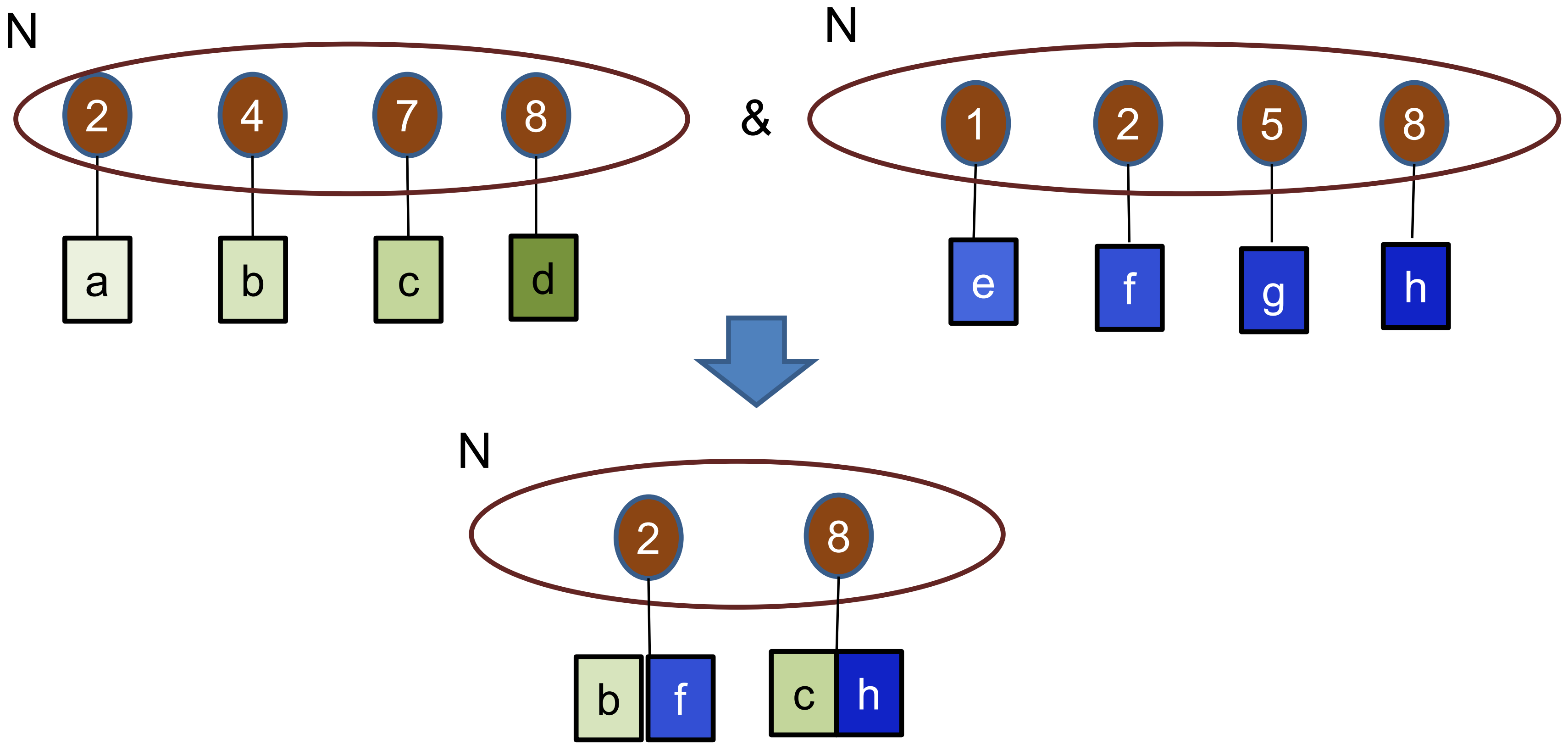




Does projection require complex hardware?

Representation dependent

Fiber Intersection



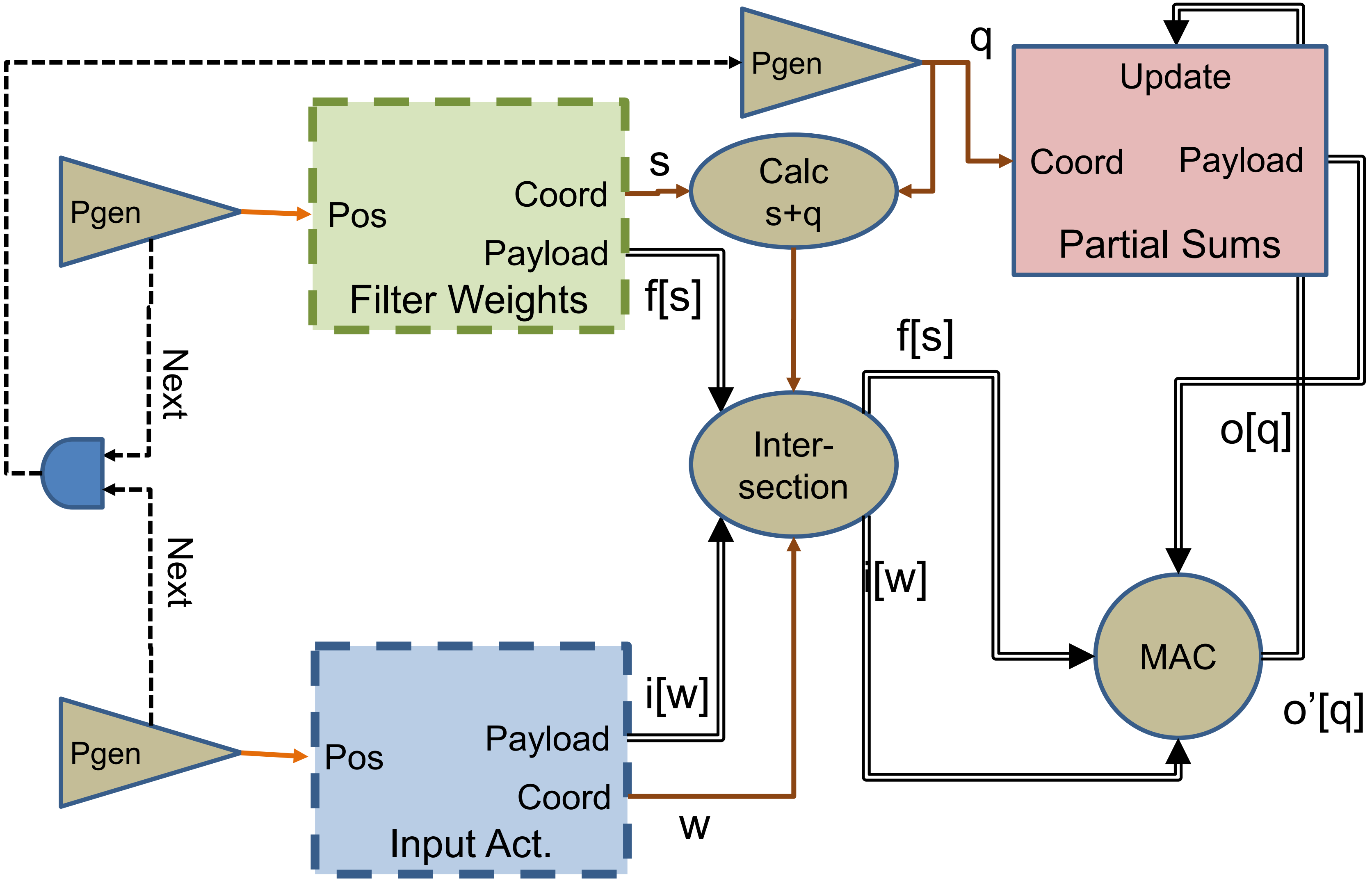
Does intersection require complex hardware?

Representation dependent

What representations would be good?

Uncompressed, bit-mask, maybe coordinate/payload

Output Stationary - Sparse Weights & Inputs



IS-OS Dataflow Einsums (K=1)

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

Substituting h=p+r, p=h-r and w=q+s, q=w-s

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

Split into multiple steps

$$T_{h,r,w-s} = I_{c,h,w} \times F_{c,r,s}$$

$$O_{h-r,w-s} = T_{h,r,w-s}$$

Reverse-substituting p=h-r, h=p+r and q=w-s into the second step

$$T_{h,r,w-s} = I_{c,h,w} \times F_{c,r,s}$$

$$O_{\underline{p,q}} = T_{\underline{p+r},r,\underline{q}}$$

IS-OS Dataflow – Step 1

$$T_{h,r,w-s} = I_{c,h,w} \times F_{c,r,s}$$

Order: h -> w -> c -> r -> s

```
parallel-for h, (t_r, i_w) in t_h << i_h:
   for w, i_val in i_w:
      for c, (i_w, f_r) in i_c & f_c:
         for r, (t_q, f_s) in t_r << f_r:
            parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
               t_ref += i_val * f_val
```

Project \`t_q\` to \`s\` using \`s = w-q\`

The fiber \`t_q\` is from the \`w-s\` rank of T

IS-OS Dataflow – Step 2

$$O_{p,q} = T_{p+r,r,q}$$

Order: p -> q -> r -> p+r

```
parallel-for p, o_q in o_p:
   for q, (o_ref, t_val) in o_q << t_q:
     for r, t_h in t_r:
       t_val = t_h.getPayload(p+r):
       o_ref += t_val
```

Pathological iteration over rank, since it is constrained by known `p` and `r`

IS-OS Dataflow

```
parallel-for h, (t_r, i_w) in t_h << i_h:
  for w, i_val in i_w:
    for c, (i_w, f_r) in i_c & f_c:
      for r, (t_q, f_s) in t_r << f_r:
        parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
          t_ref += i_val * f_val

                [“H”, “R”, “Q”] -> [“Q”, “R”, “H”]

parallel-for p, o_q in o_p:
  for q, (o_ref, t_r) in o_q << t_q:
    for r, t_h in t_r:
      t_val = t_h.getPayload(p+r):
      o_ref += t_val
```

IS-OS Dataflow

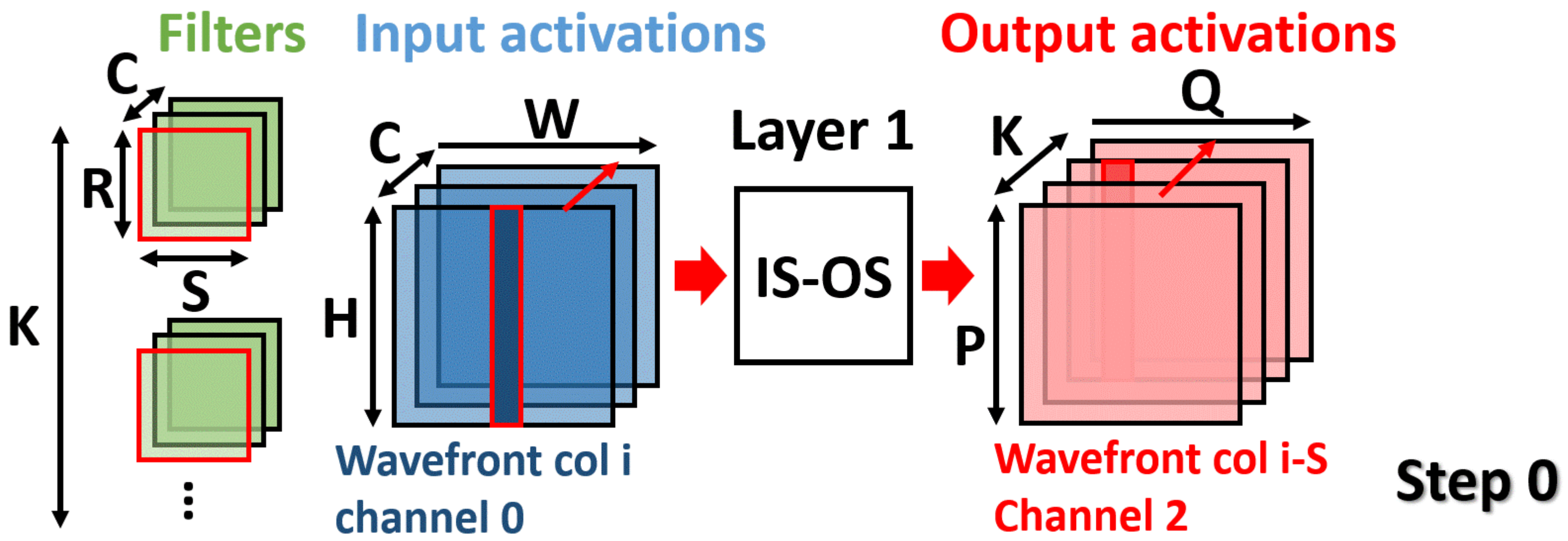
```
parallel-for h, (t_r, i_w) in t_h << i_h:
  for w, i_val in i_w:
    for c, (i_w, f_r) in i_c & f_c:
      for r, (t_q, f_s) in t_r << f_r:
        parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
          t_ref += i_val * f_val

                    ["H", "R", "Q"] -> ["Q", "R", "H"]

parallel-for p, o_q in o_p:
  for q, (o_ref, t_r) in o_q << t_q:
    for r, t_h in t_r:
      t_val = t_h.getPayload(p+r):
      o_ref += t_val
```

T is traversed in a discordant order

IS-OS Dataflow

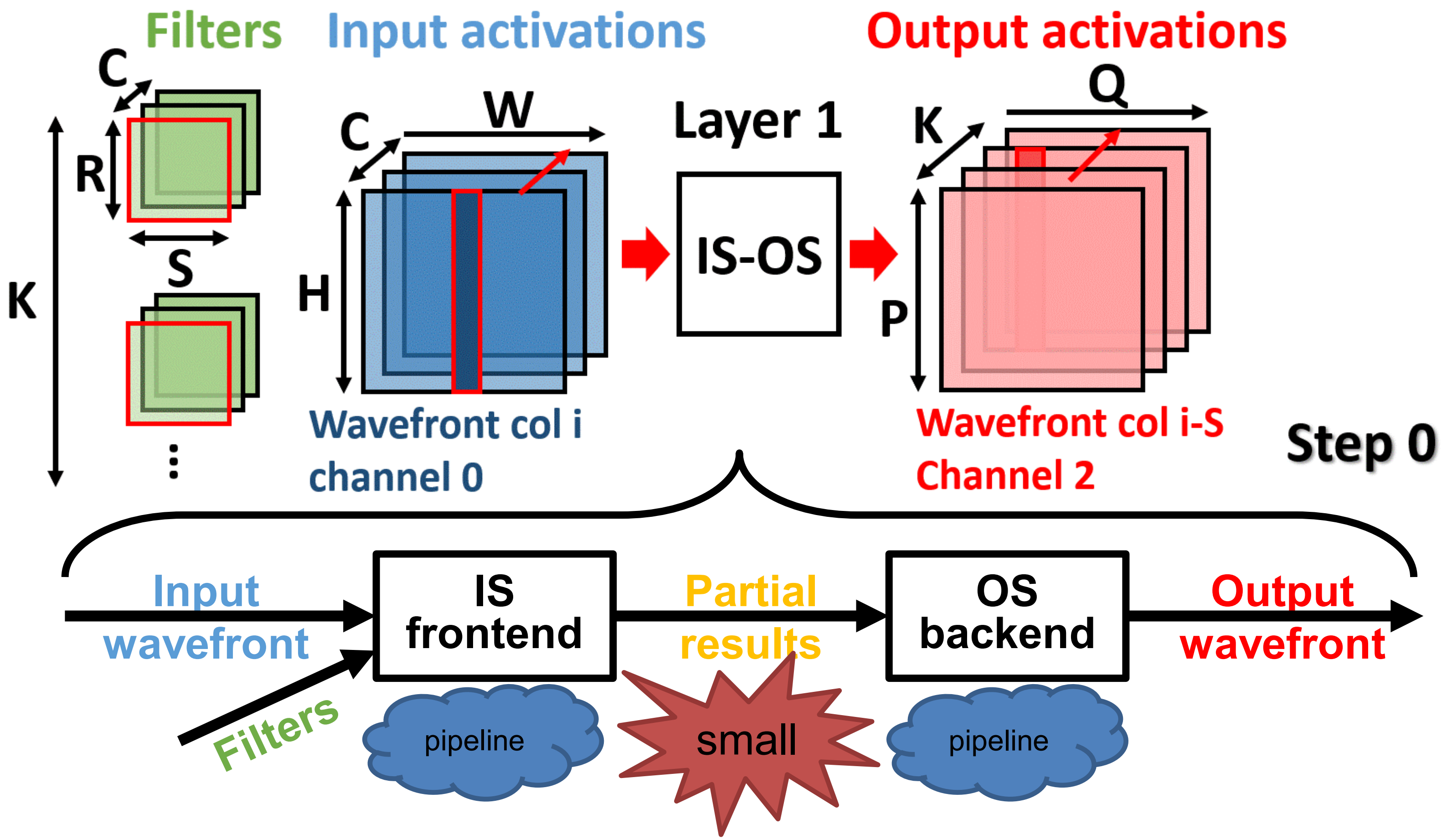
```
parallel-for h, (t_r, i_w) in t_h << i_h:
  for w, i_val in i_w:
    for c, (i_w, f_r) in i_c & f_c:
      for r, (t_q, f_s) in t_r << f_r:
        parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
          t_ref += i_val * f_val

t = t.swizzleRanks([“H”, “R”, “Q”] -> [“Q”, “R”, “H”])

parallel-for p, o_q in o_p:
  for q, (o_ref, t_r) in o_q << t_q:
    for r, t_h in t_r:
      t_val = t_h.getPayload(p+r):
      o_ref += t_val
```

IS-OS dataflow breakdown



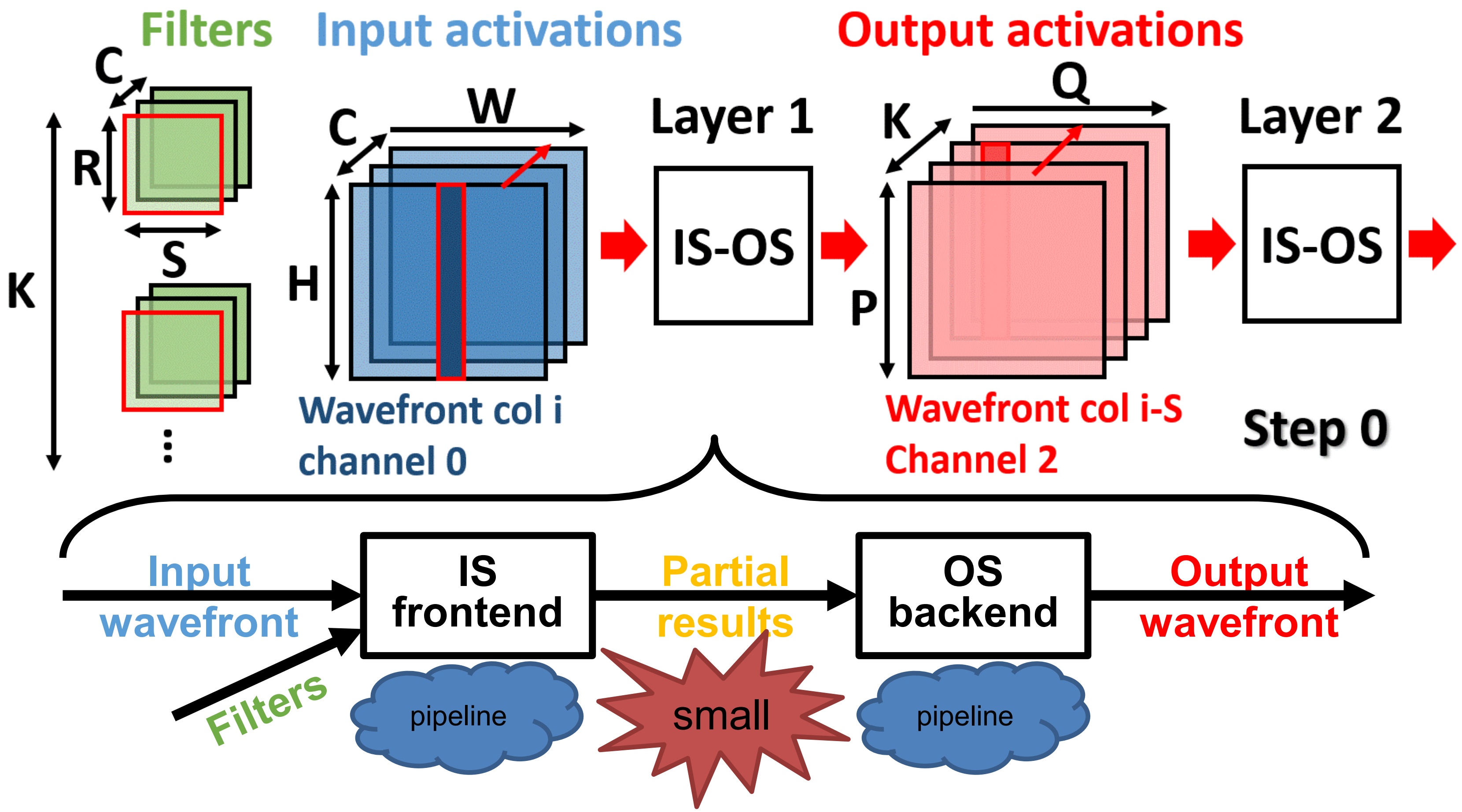
[Yang et al., ISOSceles, HPCA 2023]

IS-OS dataflow breakdown



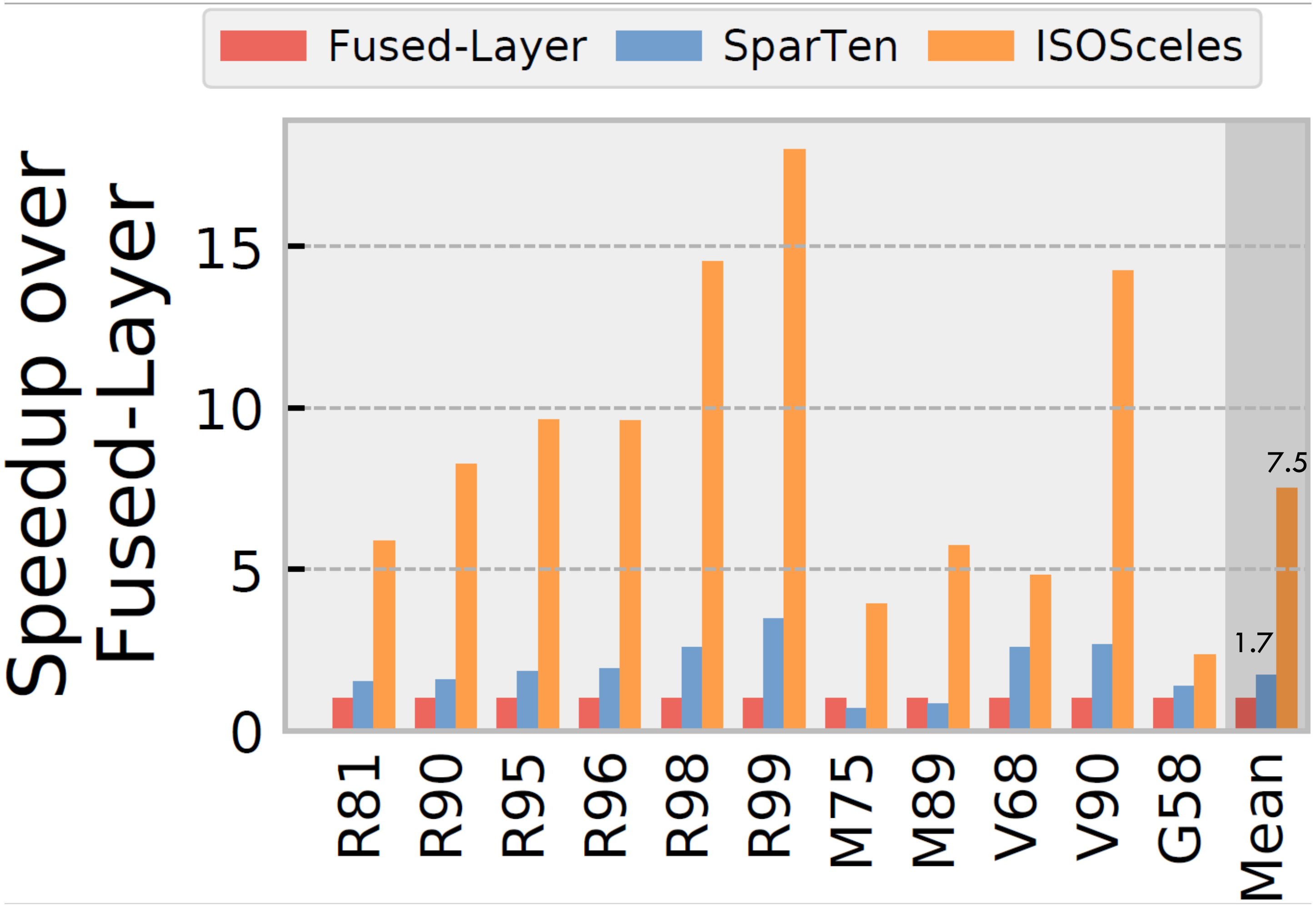
[Yang et al., ISOSceles, HPCA 2023]

IS-OS dataflow breakdown



[Yang et al., ISOSceles, HPCA 2023]

ISOSceles Speedup



[Yang et al., ISOSceles, HPCA 2023]

Next Lecture:
Sparse Multiplication