6.5930/1

Hardware Architectures for Deep Learning

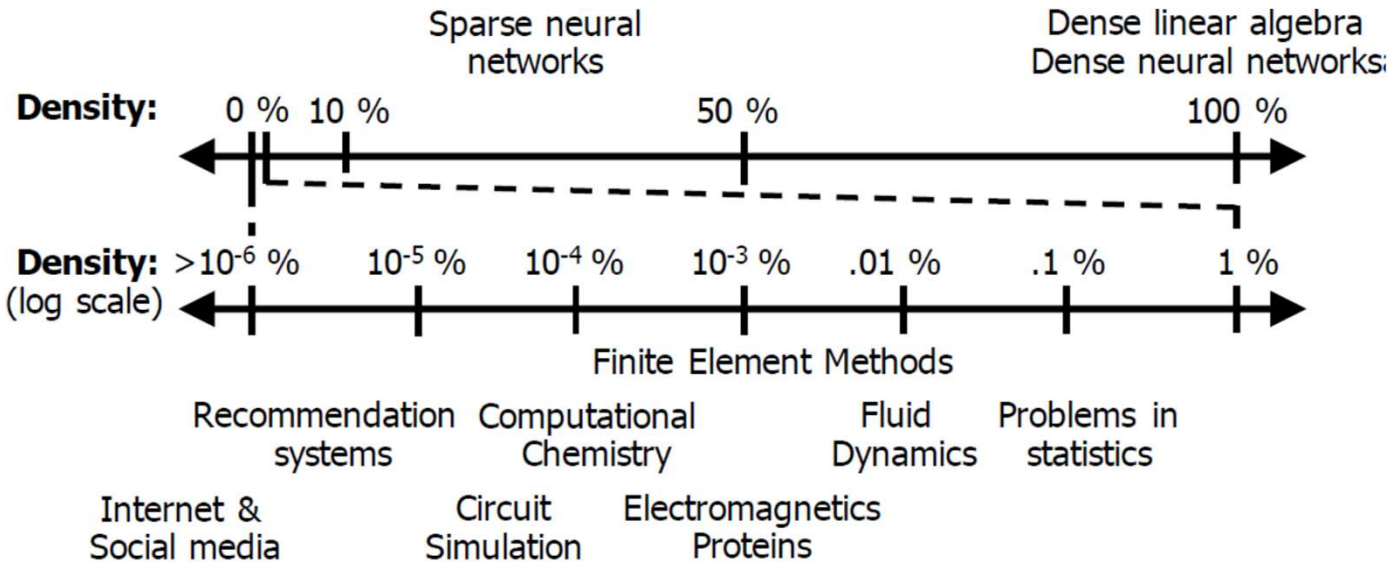
Sparse Architectures – Part 2

March 4, 2026

Joel Emer and Vivienne Sze

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

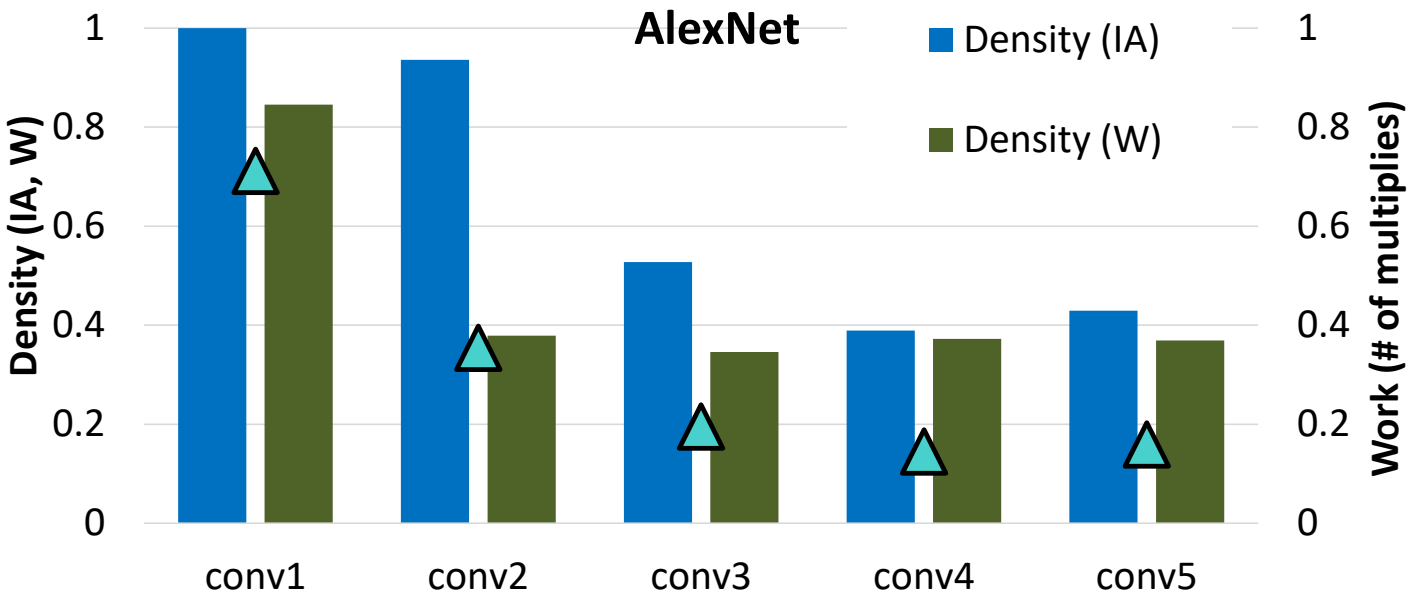
Many problems use Sparse Tensors



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

Motivation

- Leverage CNN sparsity to improve energy-efficiency



[Parashar, et.al., SCNN, ISCA 2017]

Aspects of Scheduling - Sparsity

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

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

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

Aspects of Scheduling - Sparsity

Gating:

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

Format:

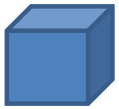
Choose tensor representations to save storage space and energy associated with zero accesses

Skipping:

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

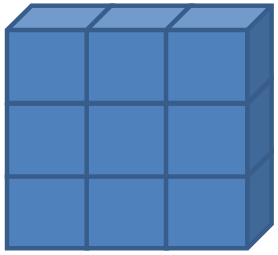
Tensors

Rank-0 - Scalar

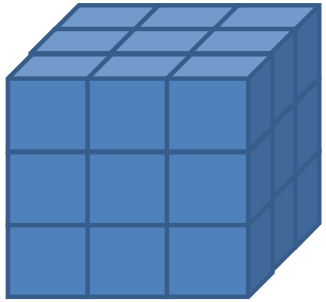


Rank-1 - Vector

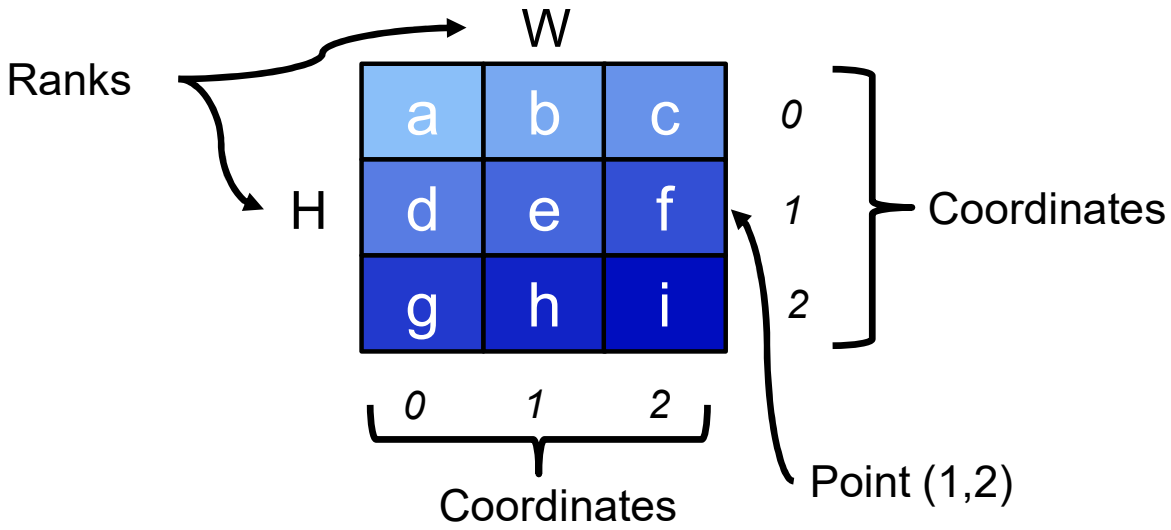
Rank-2 - Matrix



Rank-3 - Cube



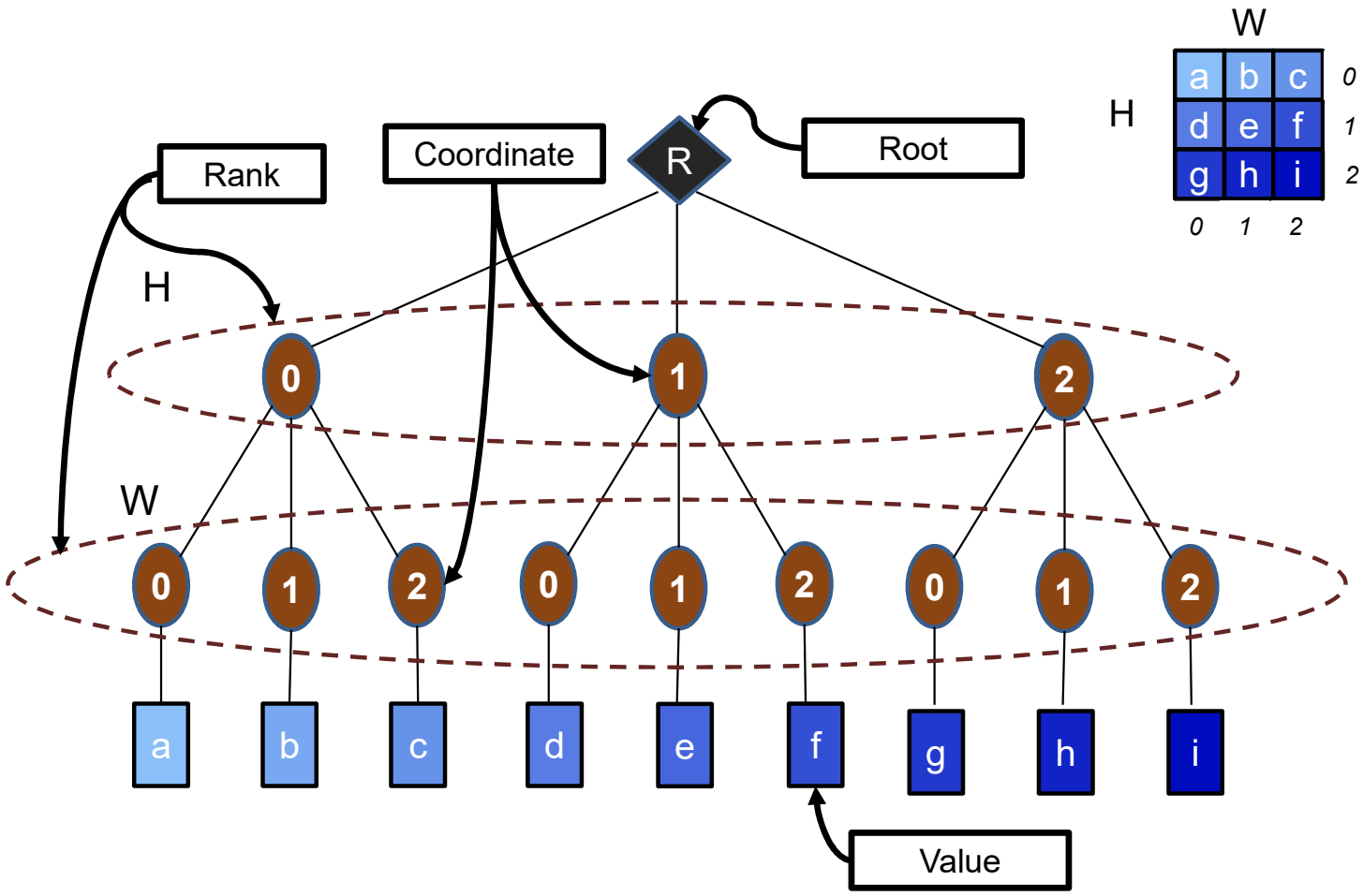
Tensor Data Terminology



- The elements of each “rank” (dimension) are identified by their “coordinates”, e.g., rank H has coordinates 0, 1, 2

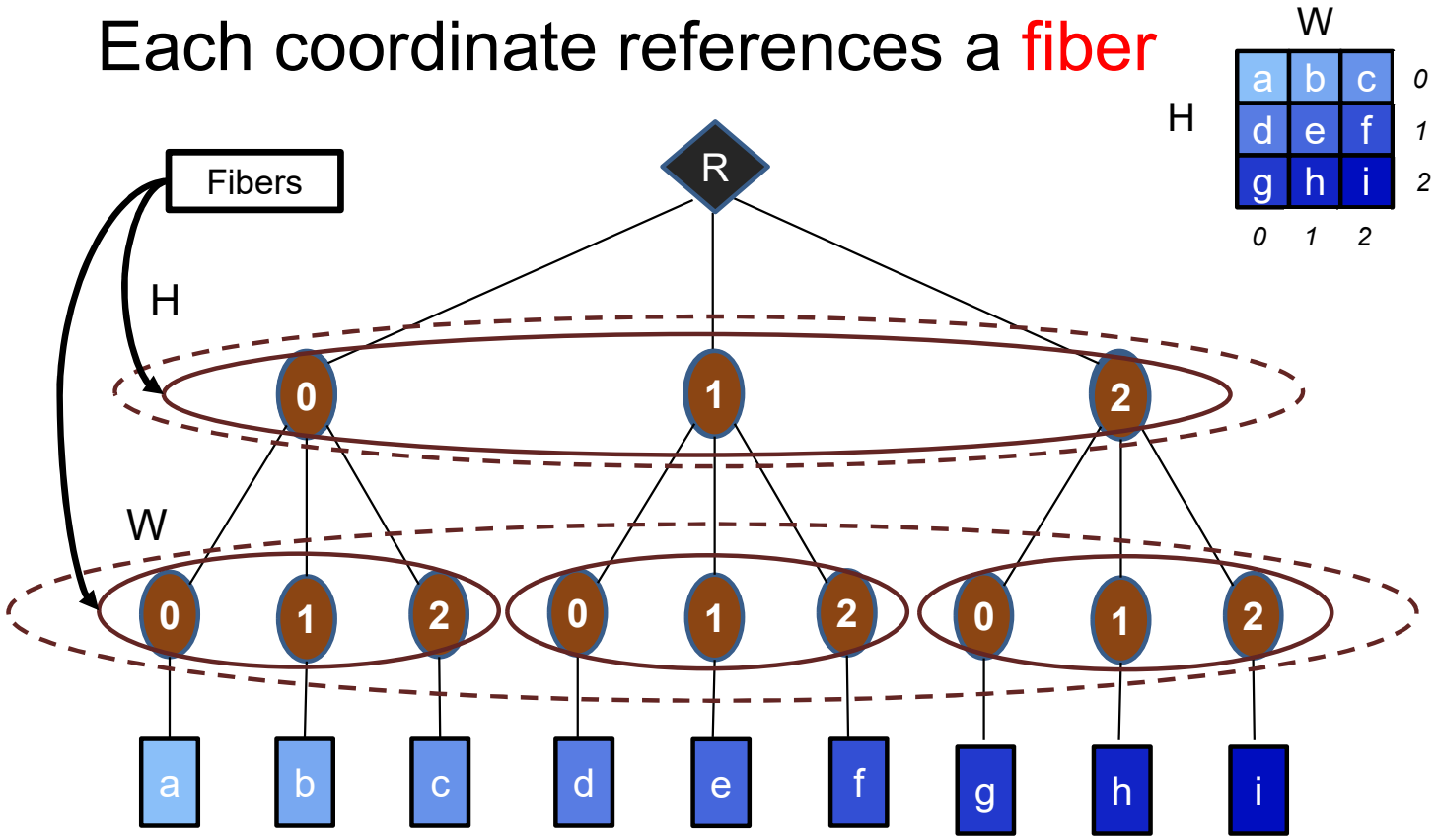
- Each element of the tensor is identified by the tuple of coordinates from each of its ranks, i.e., a “point”.
  So (1,2) -> “f”

Tree-based Tensor Abstraction



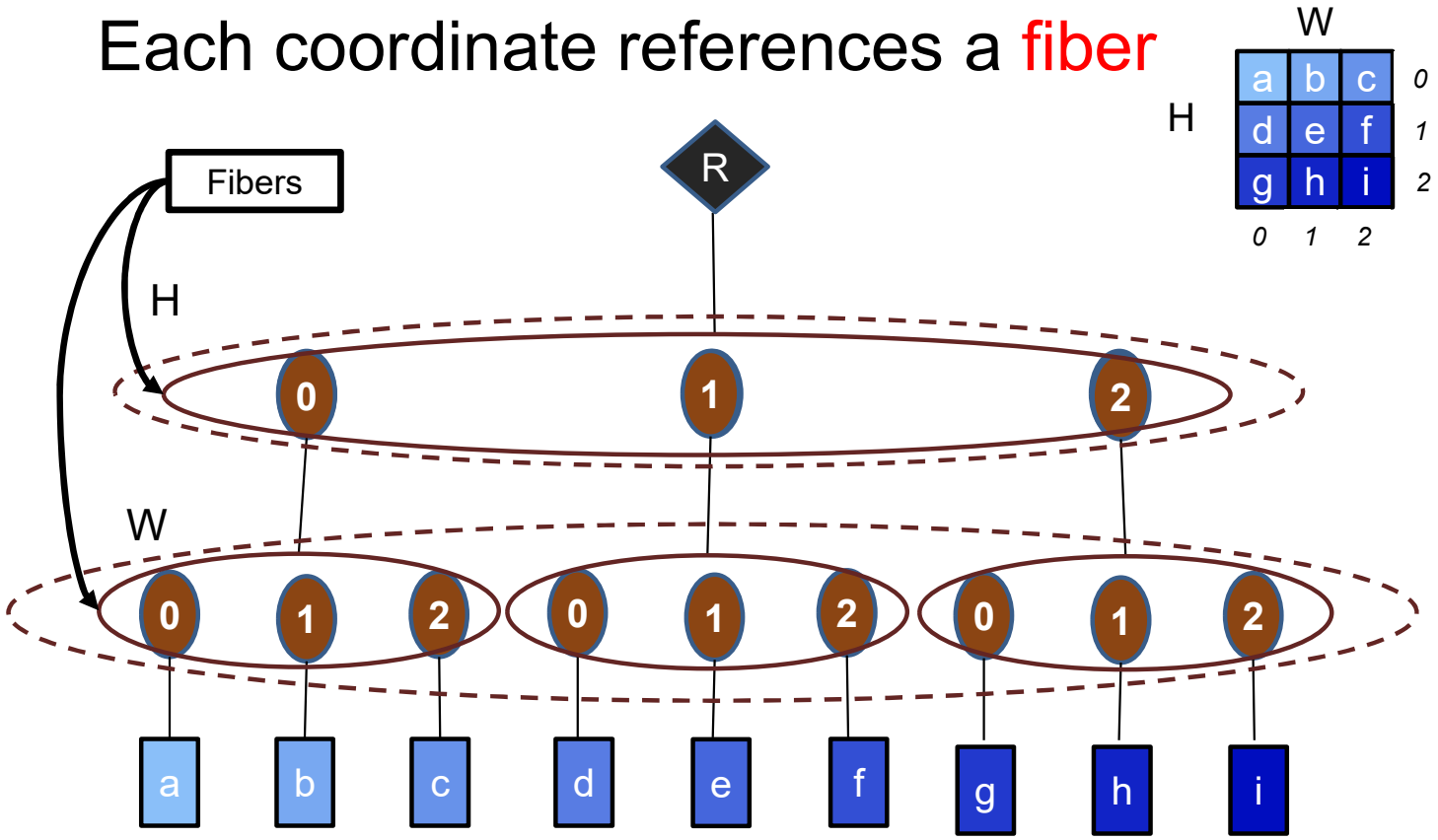
Tree-based Tensor Abstraction

Each coordinate references a fiber

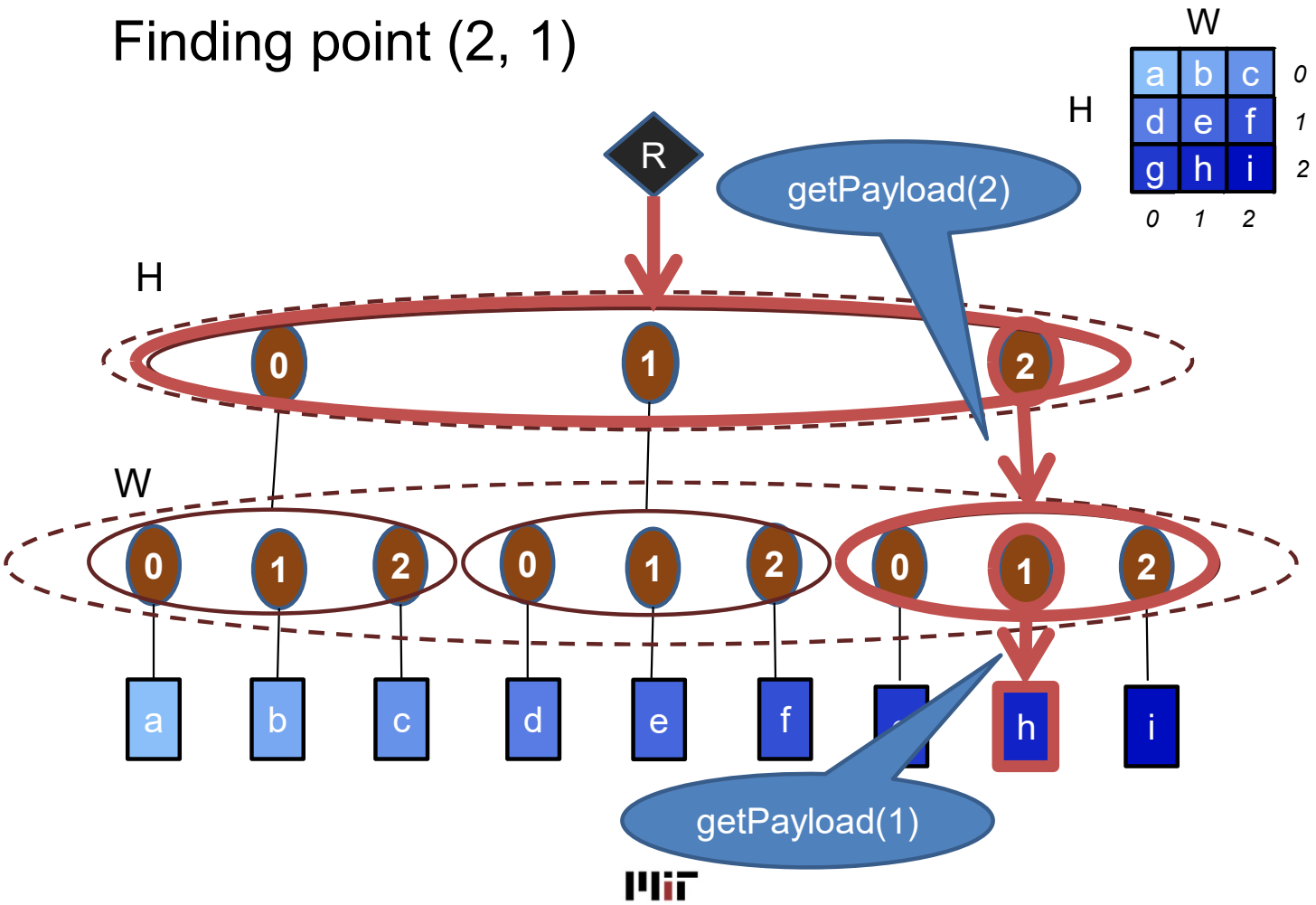


Fibertree Tensor Abstraction

Each coordinate references a fiber

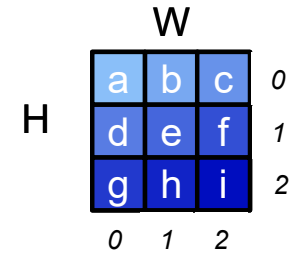


Fibertree Tensor Abstraction



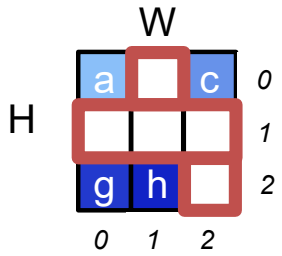
Fibertree Tensor Abstraction

What if tensor is sparse?



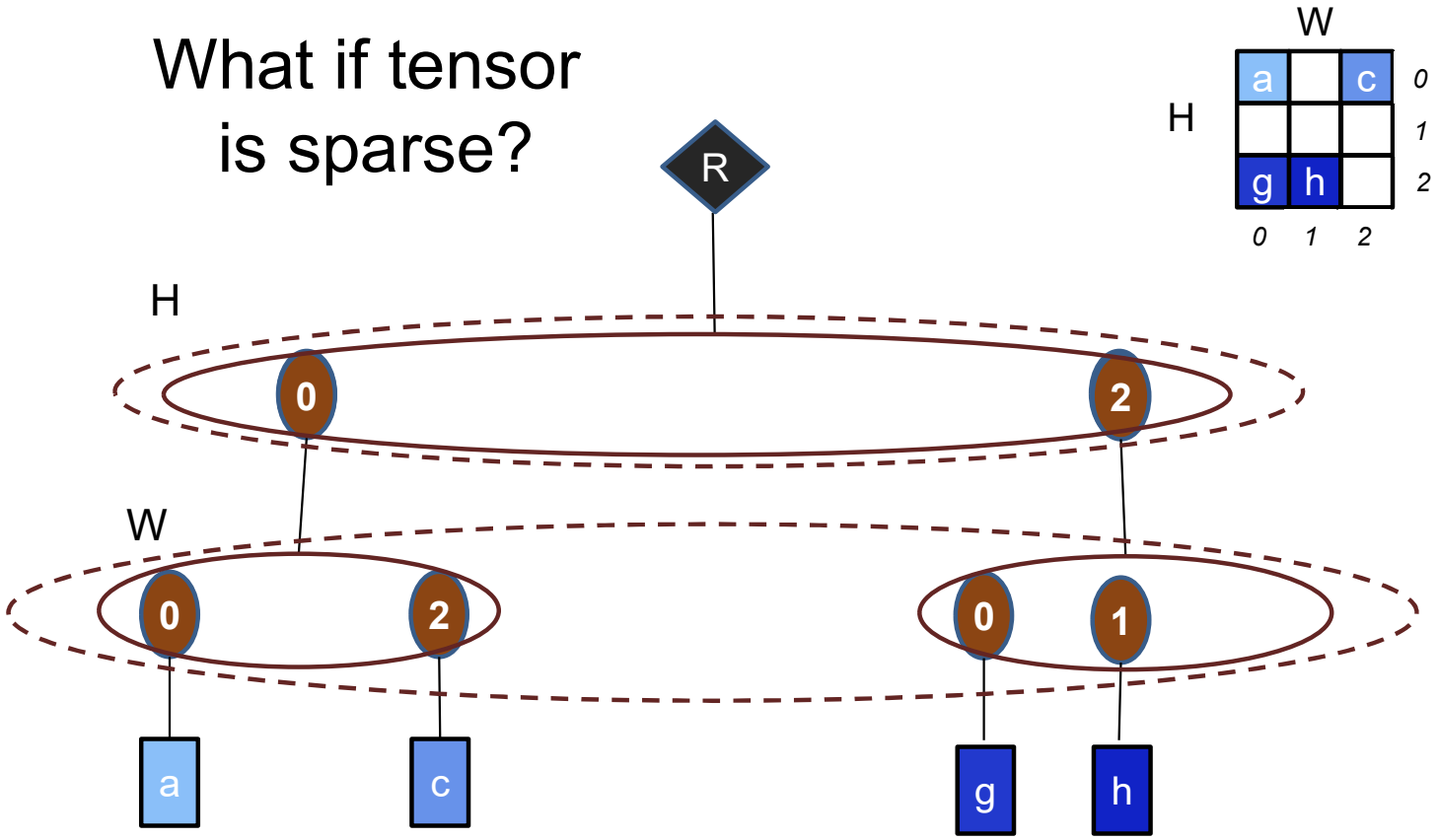
Fibertree Tensor Abstraction

What if tensor is sparse?

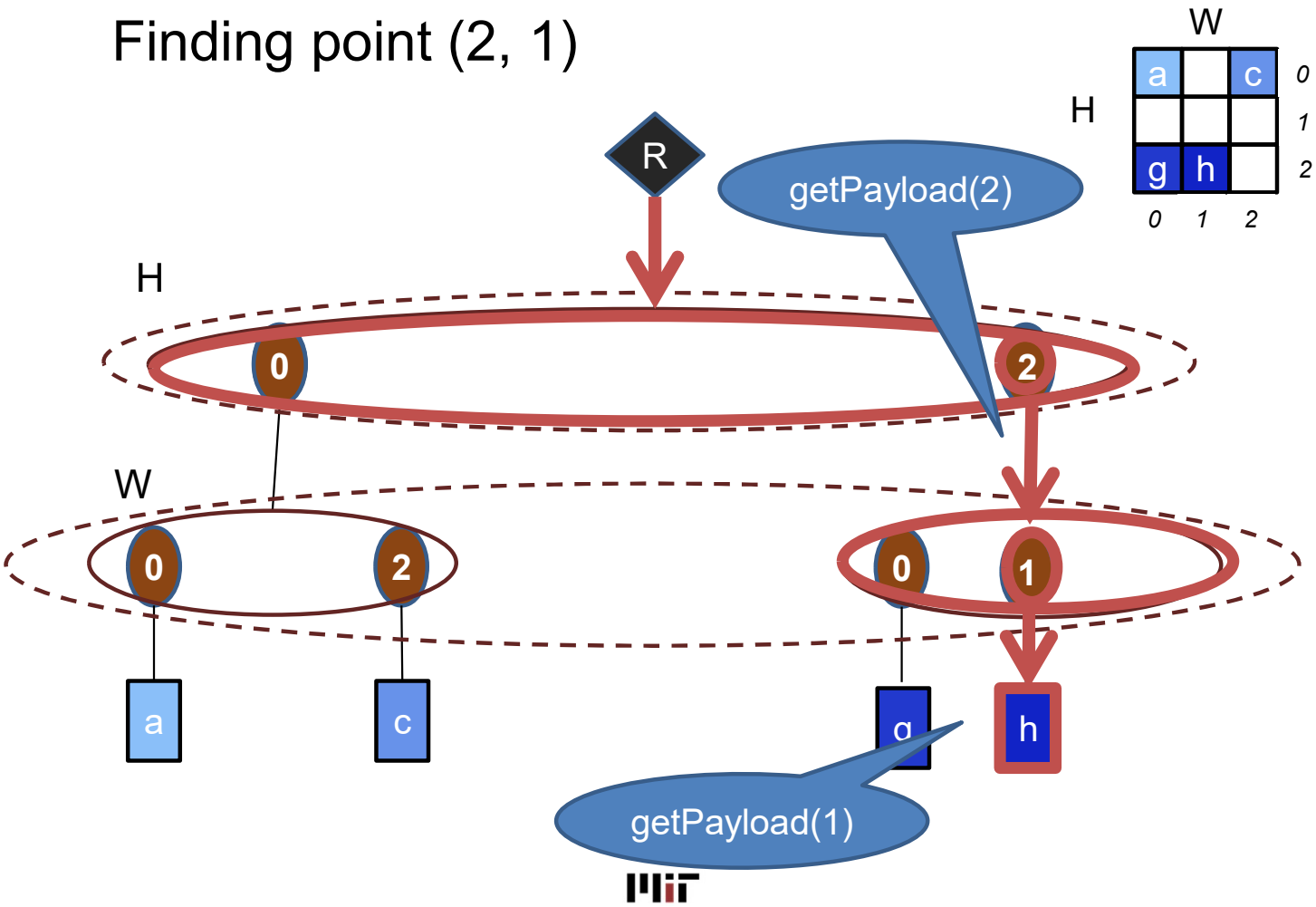


Fibertree Tensor Abstraction

What if tensor is sparse?

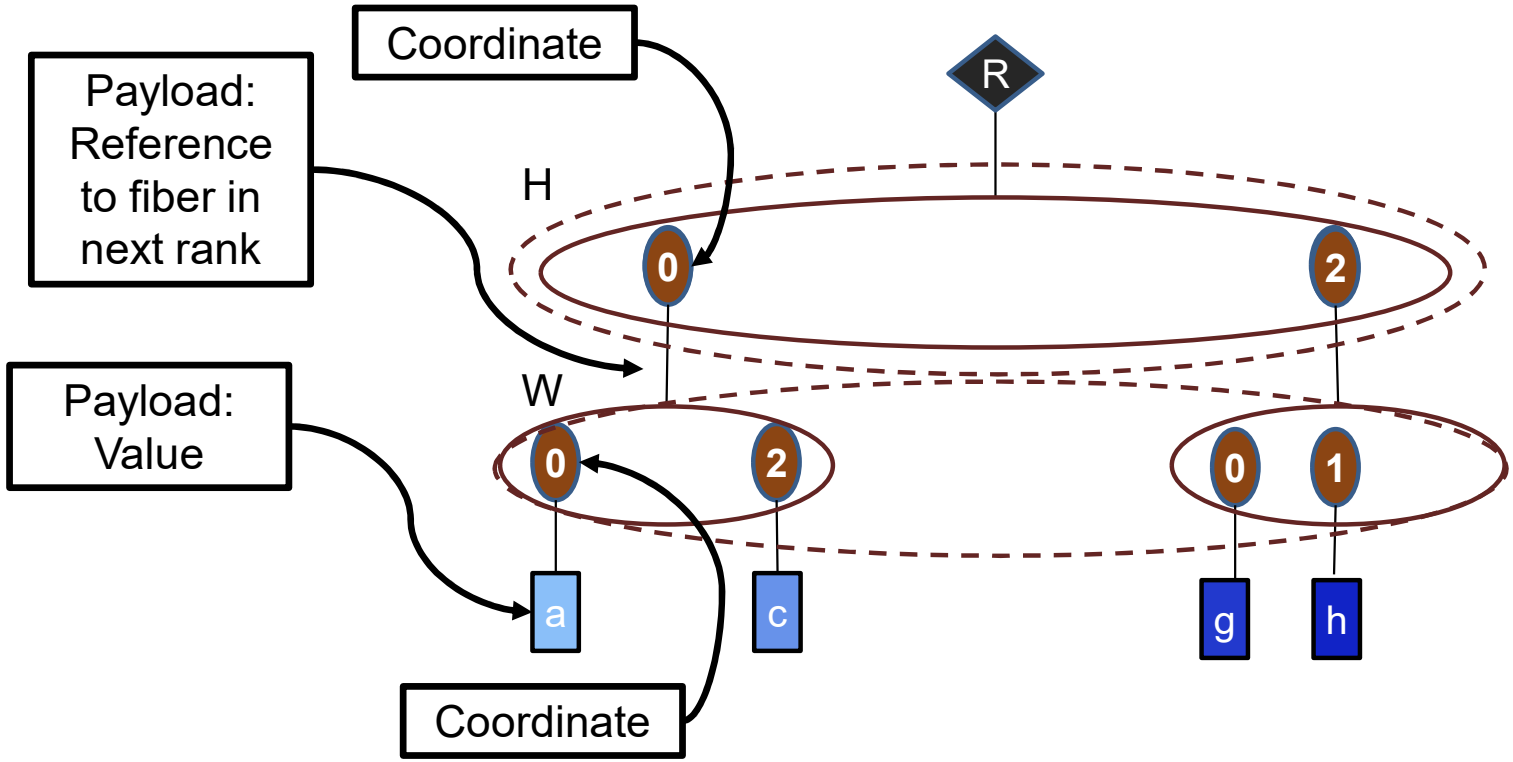


Fibertree Tensor Abstraction



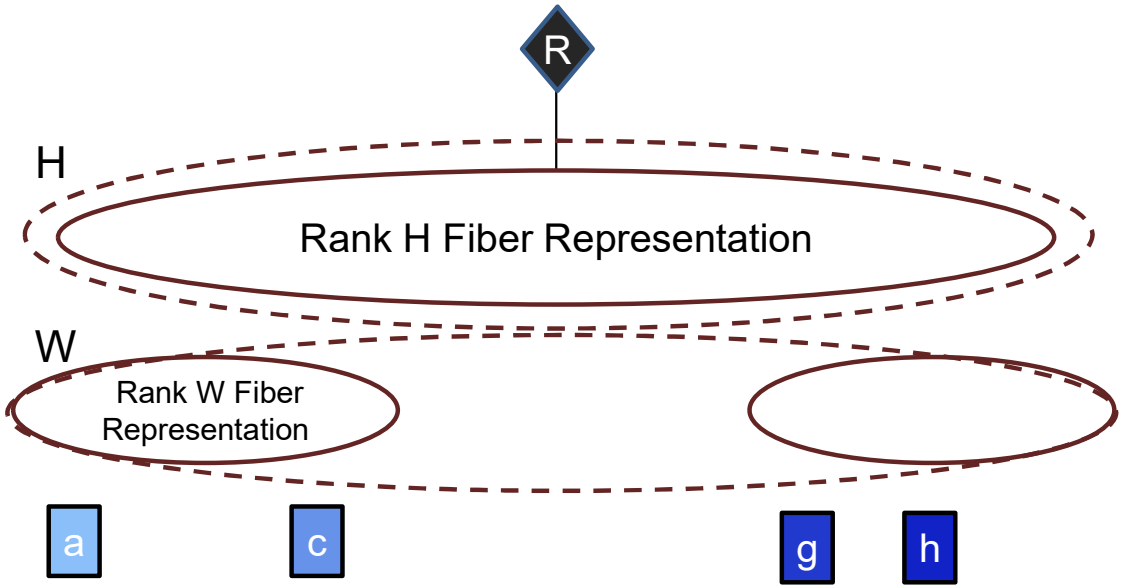
Information in a Fiber

- Each fiber has a set of (coordinate, “payload”) tuples

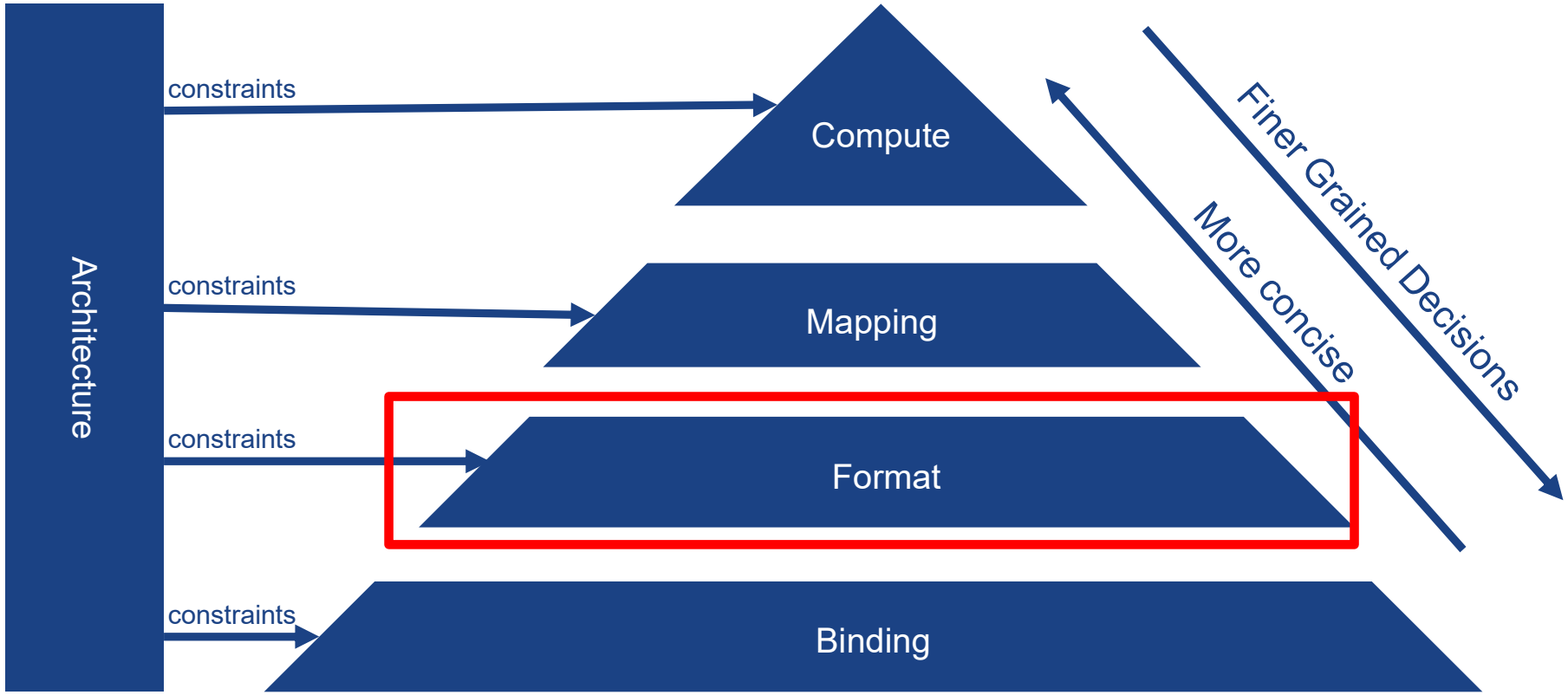


Information in a Fiber

Method: maybe(payload) = fiber.getPayload(coordinate)



TeAAL Pyramid of Concerns

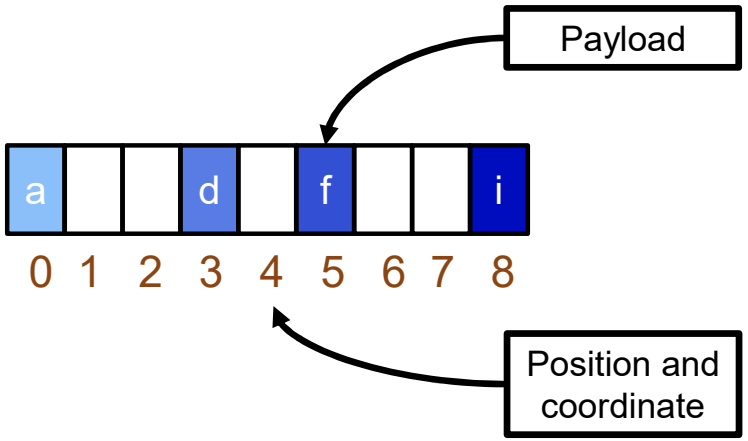


[TeAAL, Nayak et.a. MICRO 2023]

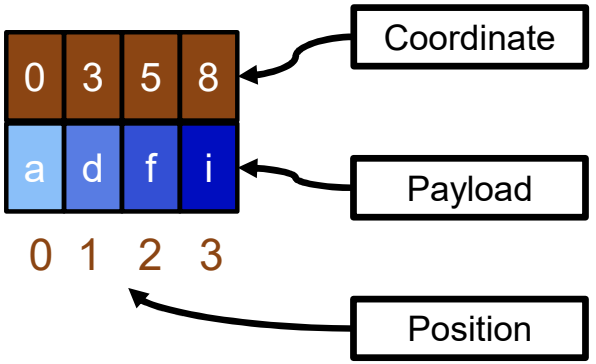
Example Fiber Representations

Each fiber has a set of (coordinate, “payload”) tuples

Array



Coordinate/Payload List



Data in a fiber is accessed by its position or offset in memory

Fiber Representation Choices

- Implicit Coordinates
  - Uncompressed (no metadata required)
  - Compressed – e.g., run length encoded
- Explicit Coordinates
  - E.g., coordinate/payload list
- Compressed vs Uncompressed
  - Compressed/uncompressed is an attribute of the representation*.
  - Uncompressed means size is proportional to maximum coordinate value
  - Compressed formats will have metadata overhead relative to uncompressed formats. For dense data, this may cost more than just using an uncompressed format.
  - Space efficiency of a representation depends on sparsity

*Note: sparsity/density is an attribute of the data.

Compressed Implicit Coordinate Representations

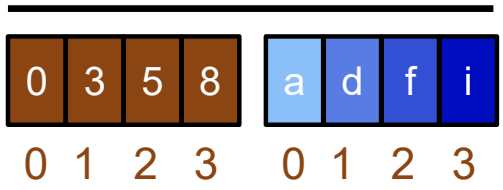
- "Empty" coordinate compression via zero-run encoding
  - Run-length coding (RLE)
    - (run-length of zeros, non-zero payload)…
  - Significance map encoding
    - (flag to indicate if non-zero, non-zero payload)…
- Payload encoding
  - Fixed length payload
  - Variable length payload
    - E..g., Huffman coding
- Efficiency of different traversal patterns through the tensor is affected by encoding, e.g., finding the payload for a particular coordinate…

Compressed Explicit Coordinate Representations

- Coordinate list representation
  - Struct of arrays form

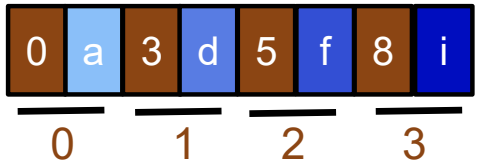
    (coordinate of non-zero value)…

    (non-zero payload)…



  - Array of structs form

    (coordinate of non-zero value, non-zero payload)…



Black bar show scope of struct

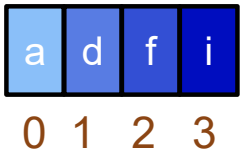
- Payload encoding
  - Explicit
    - Immediate value
    - Pointer
  - Implicit
    - Offset of coordinate is offset of payload

More Explicit Coordinate Representations

- Coordinate Bitmask

Any complexity with lookupPayload()?

May require a population count of the coordinate array.

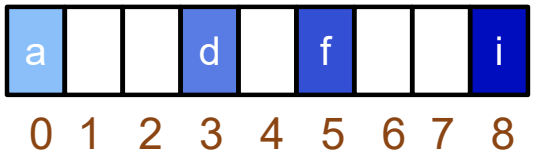


Have we seen a representation like this?

Yes, the Eyeriss input activations used for gating were sort of like that….

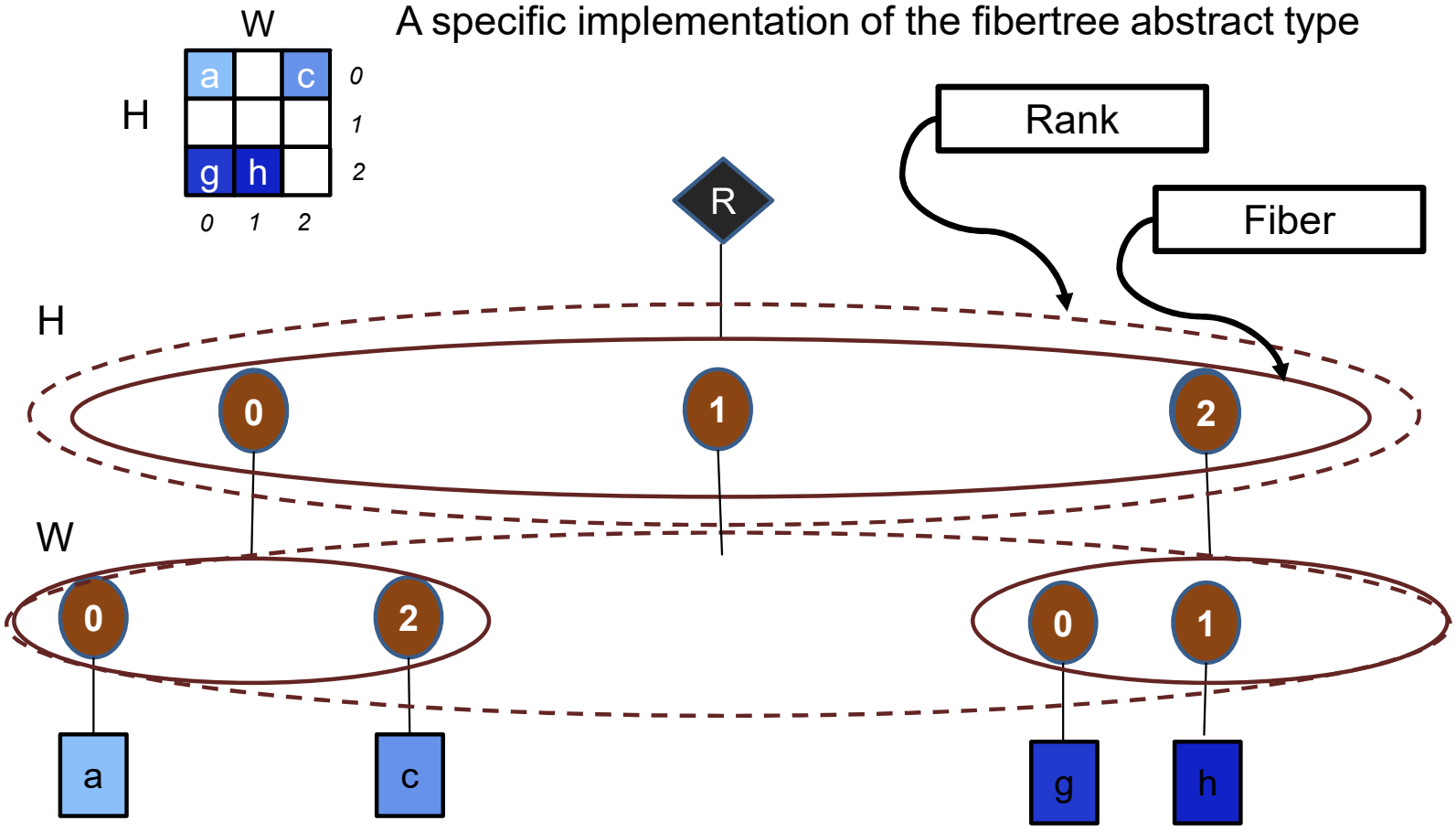
Is this useful even with no compression?

Yes, cheap check for zeros.



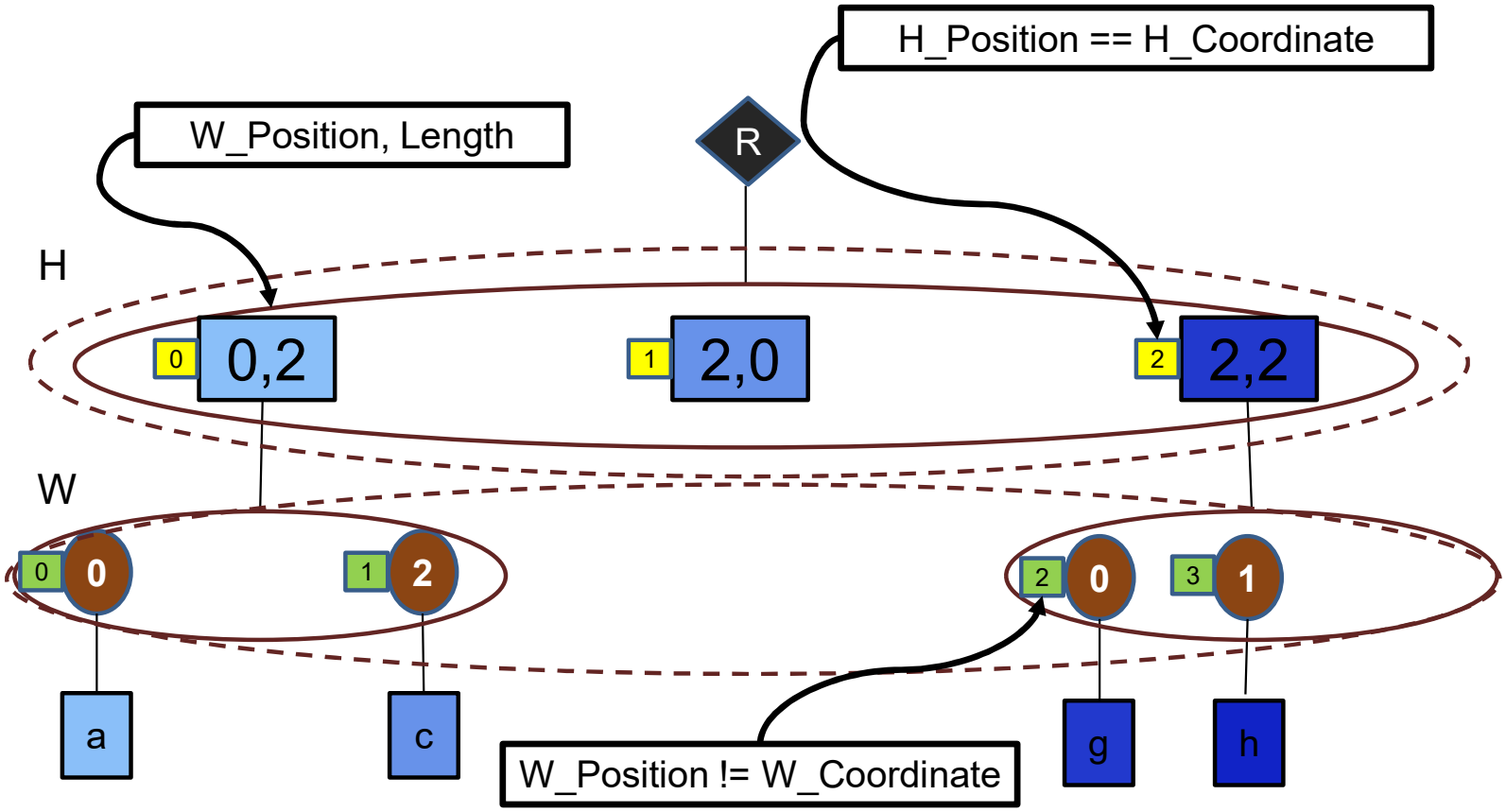
Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



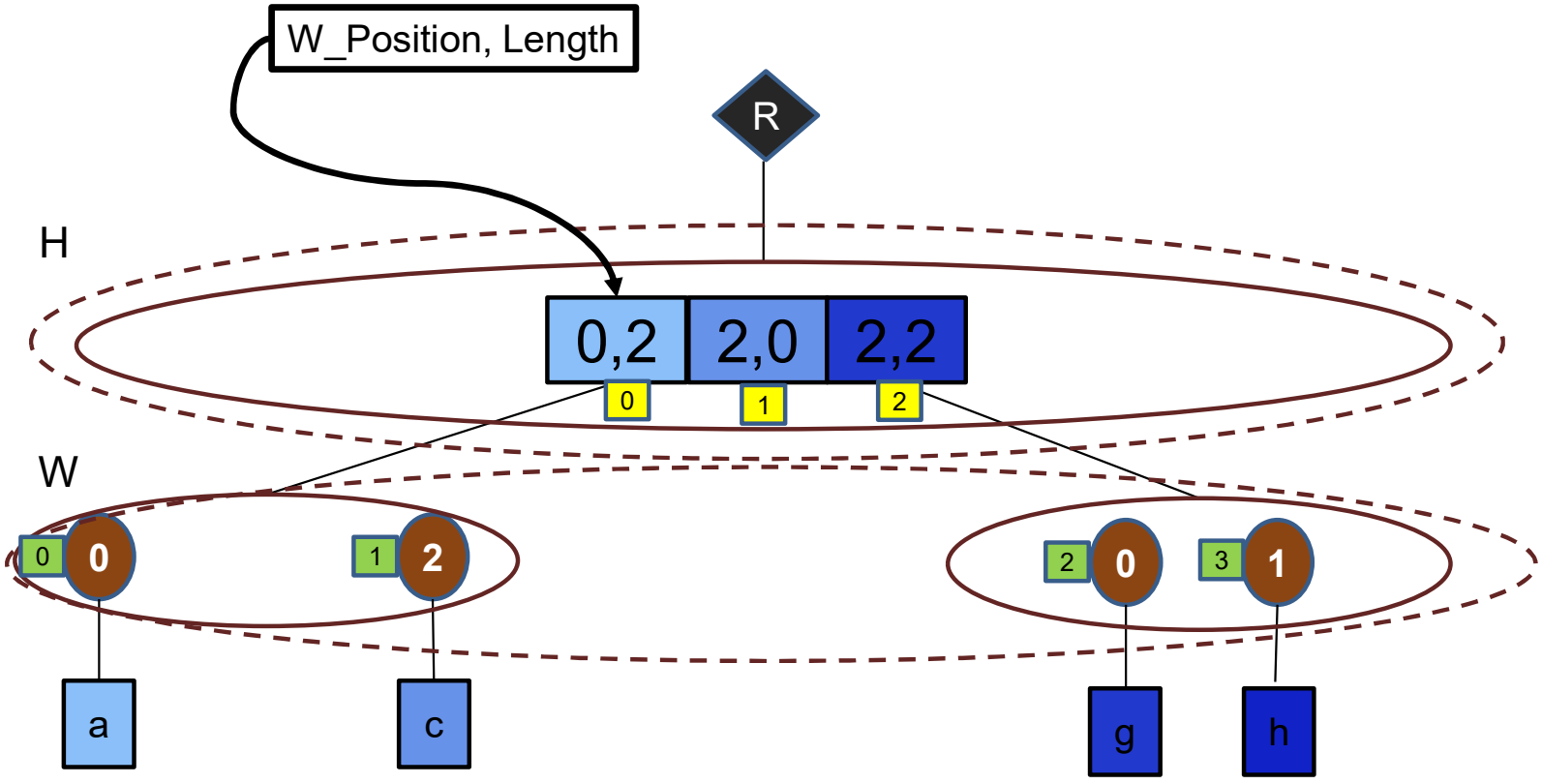
Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



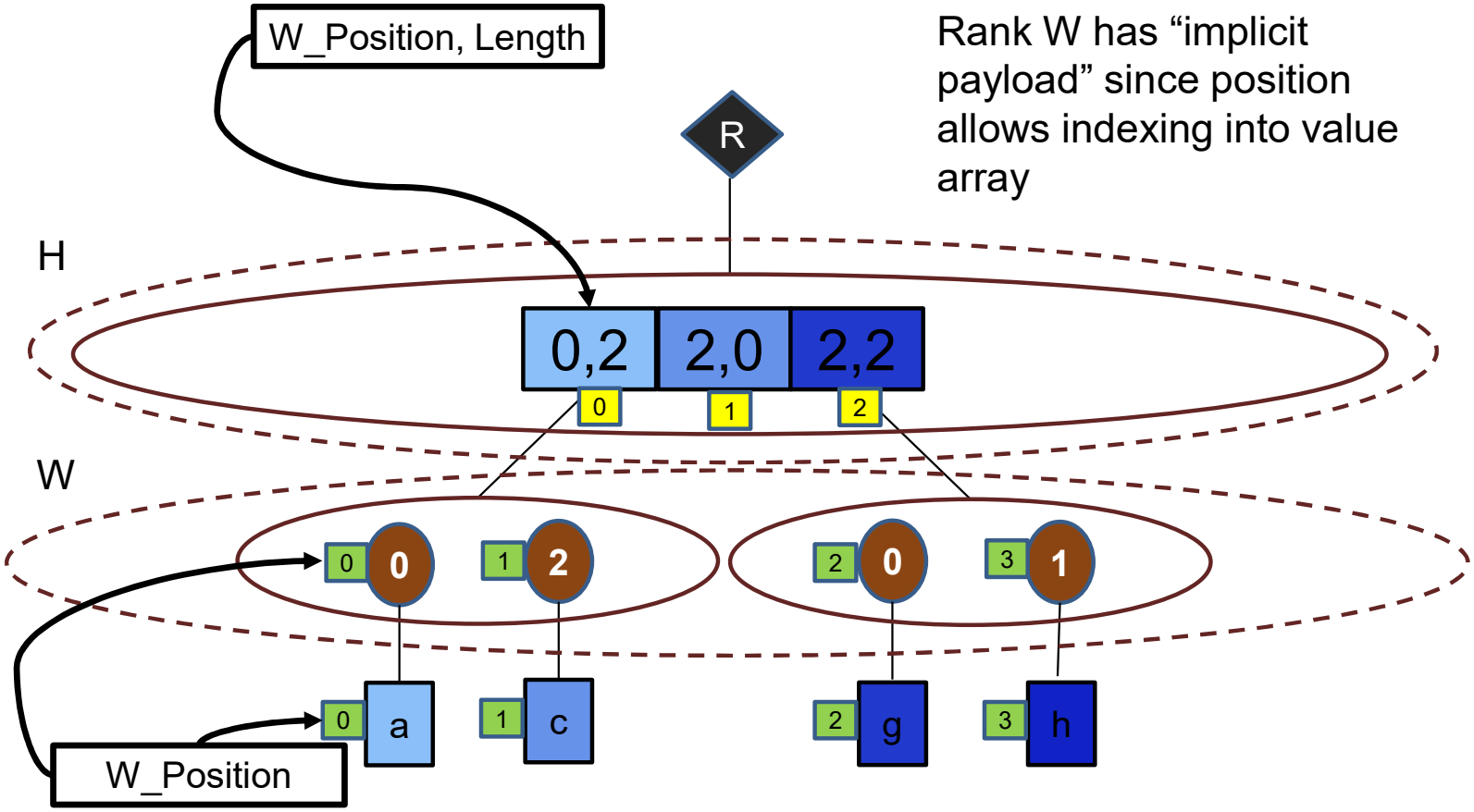
Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



Uncompressed/Compressed Representation

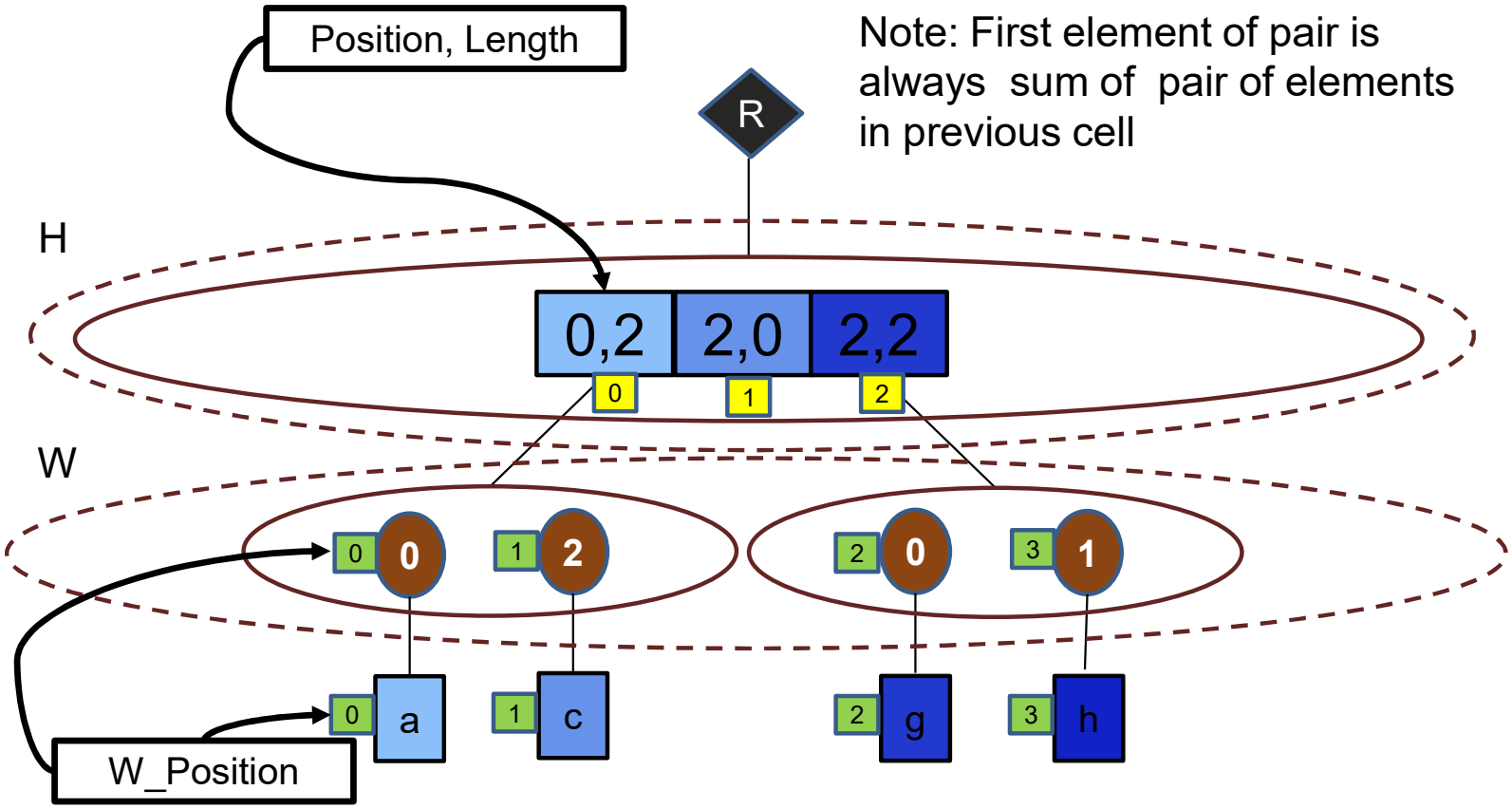
A specific implementation of the fibertree abstract type



Rank W has “implicit payload” since position allows indexing into value array

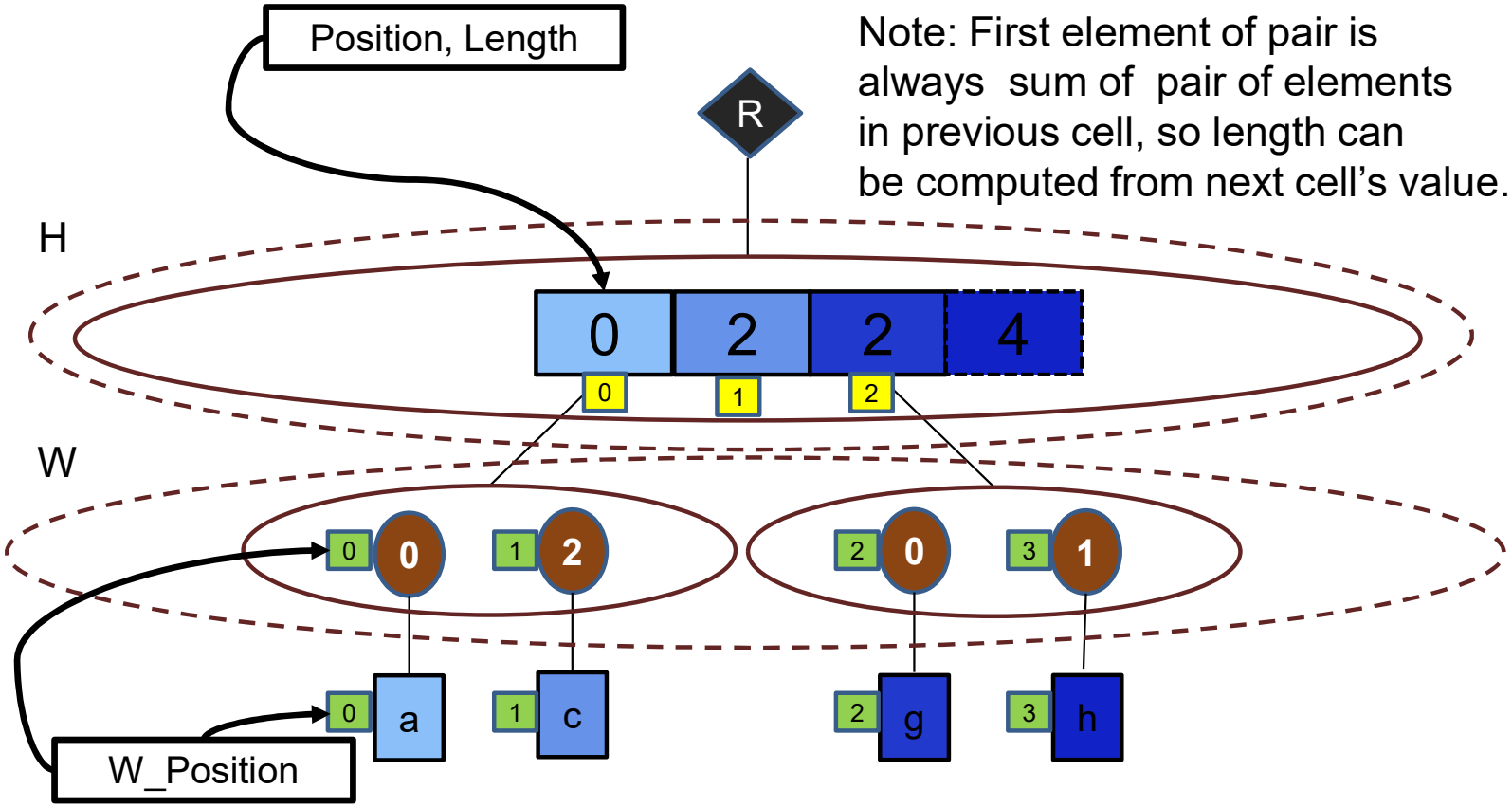
Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



Uncompressed/Compressed Representation

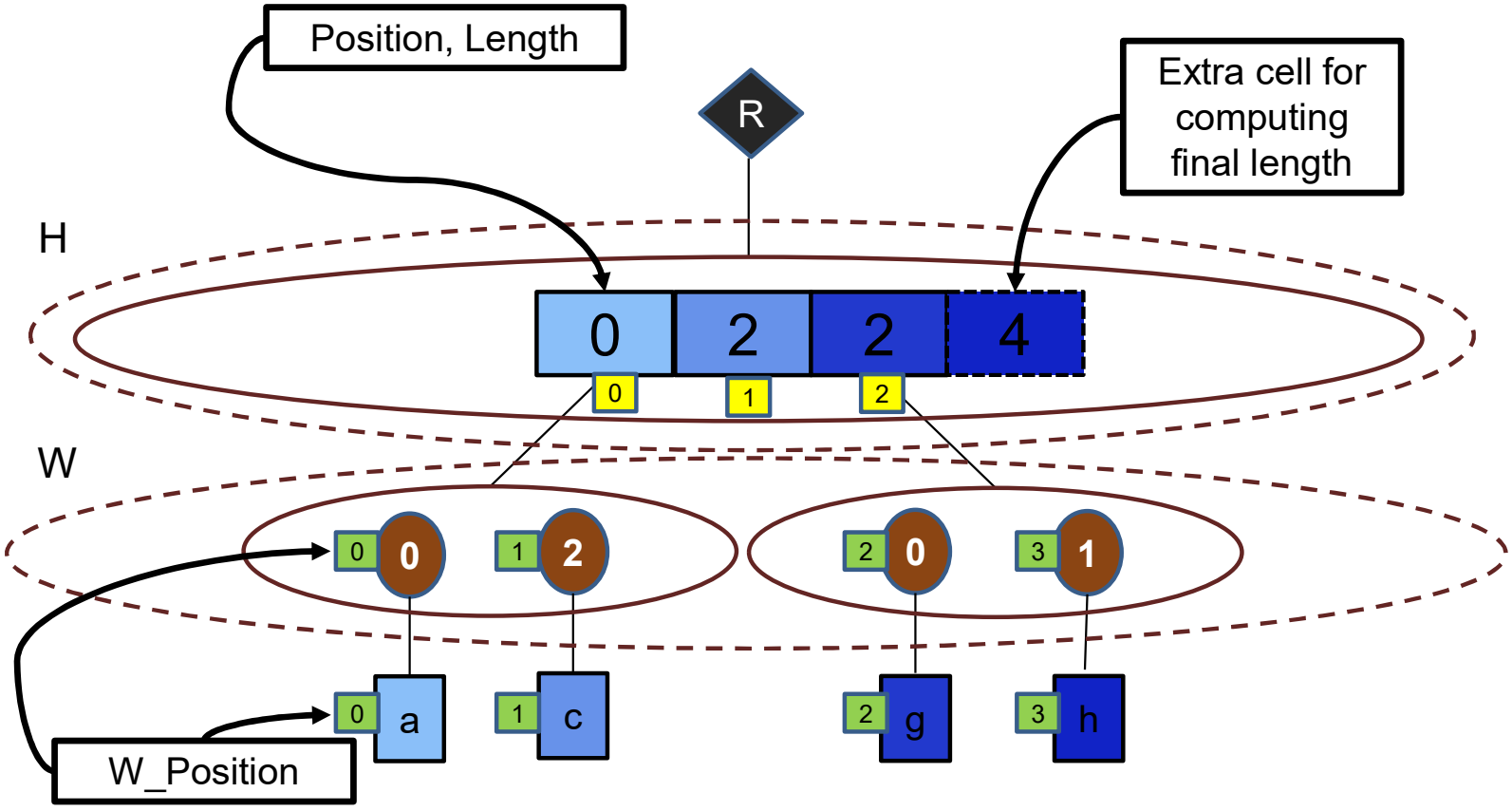
A specific implementation of the fibertree abstract type



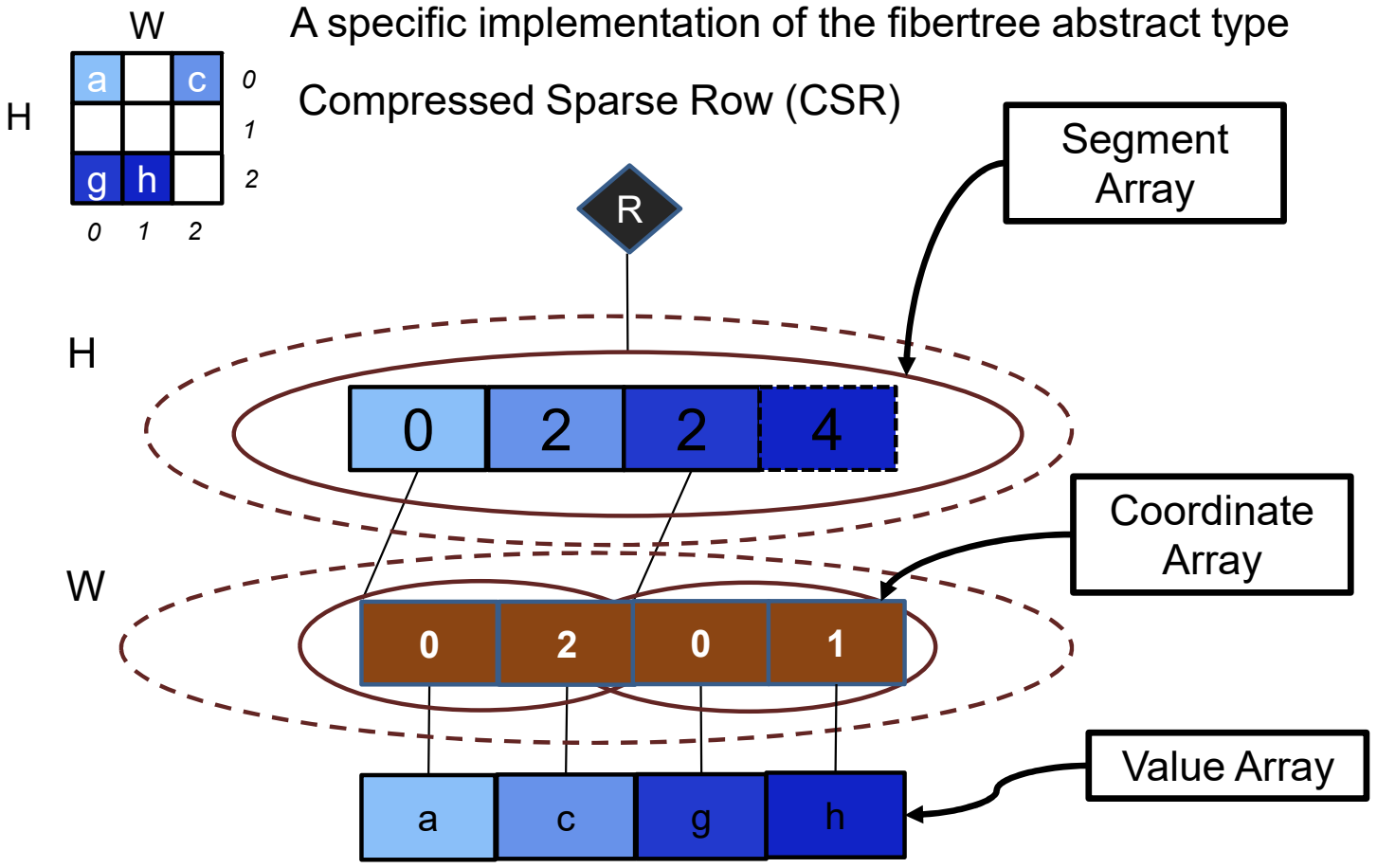
Note: First element of pair is always sum of pair of elements in previous cell, so length can be computed from next cell's value.

Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



Uncompressed/Compressed Representation



Explicit Coordinate Representations

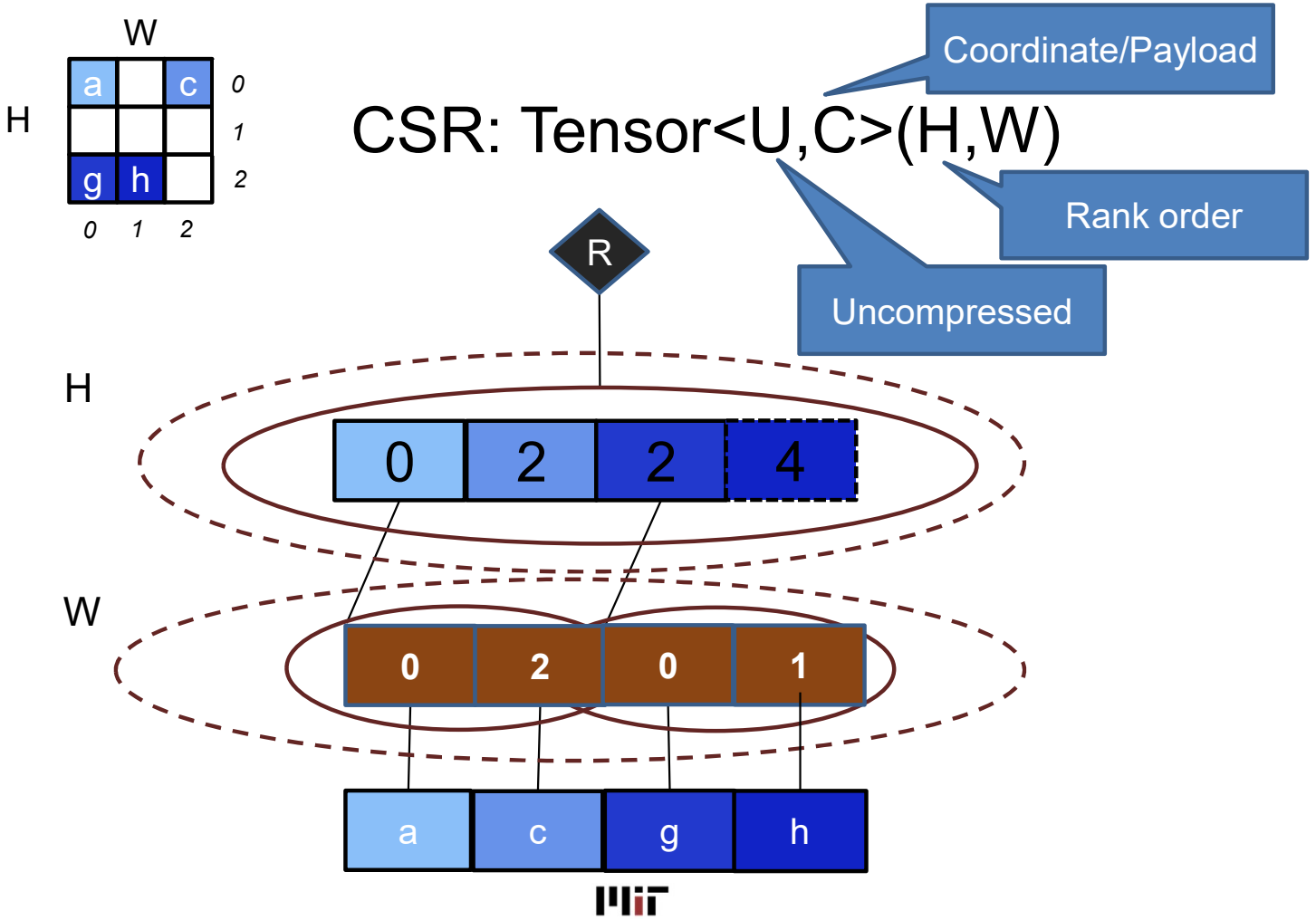
- Coordinate/Payload list
  - (coordinate, non-zero payload)… (array of structs)
  - (coordinate)… , (non-zero payload)… (struct of arrays)
- Hash table (per fiber)
  - (coordinate -> payload) mapping
- Hash table (per rank)
  - (fiber_id, coordinate -> payload) mapping
- Bit vector of non-zero coordinates
  - Compressed or uncompressed payload

Per Rank Tensor Representations

- Uncompressed [U]
  -
- Run-length Encoded [R]
  -
- Coordinate/Payload List [C]
  -
- Hash Table (per rank) [H_r]
- Hash Table (per fiber) [H_f]
- Tagged union of any combination of previous types

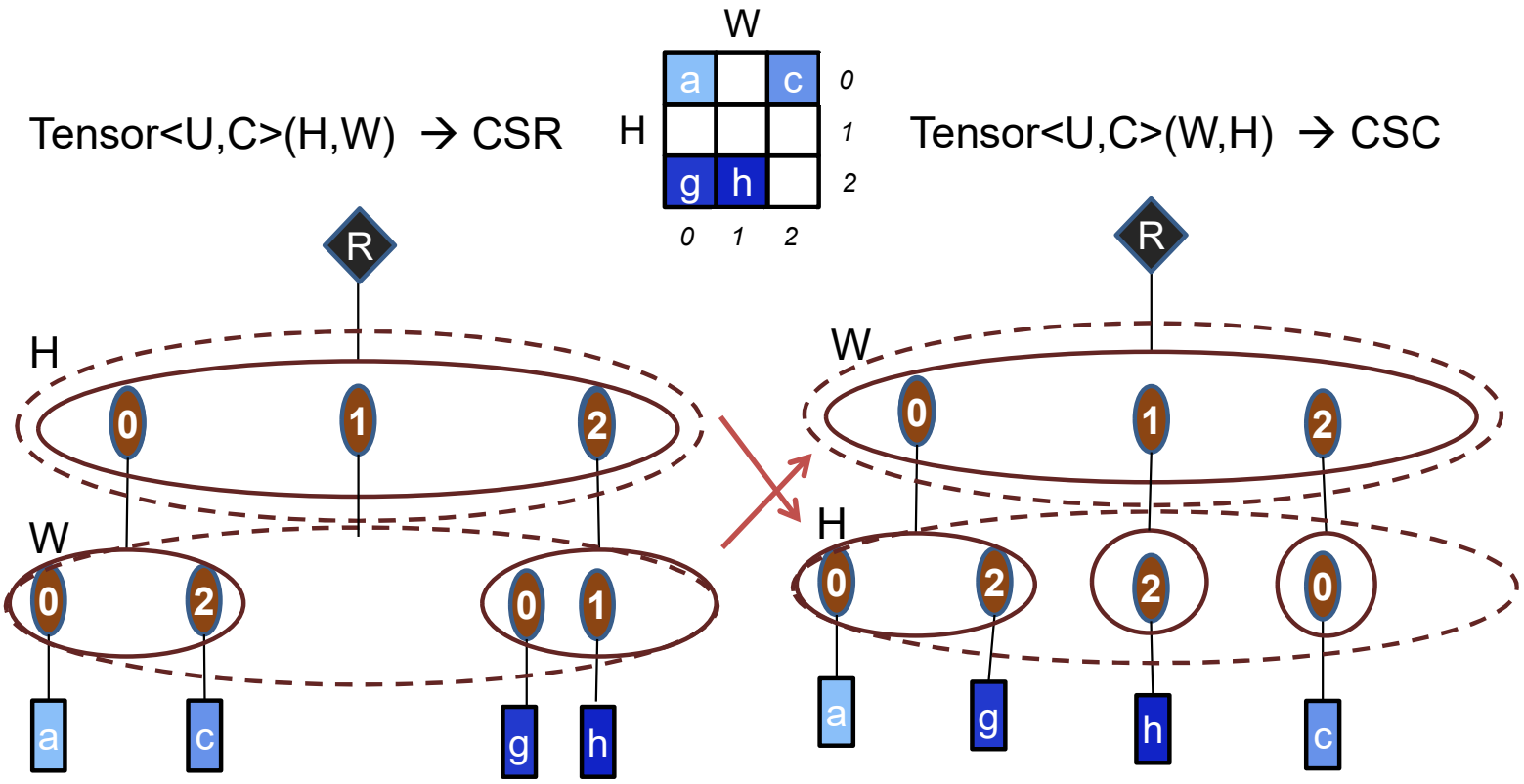
Inspired by collaboration with Kjolstad
in [Kjolstad, OOPSLA17], [Chou, OOPSLA18]

Notation for CSR



Representation of Order of Ranks

Differentiating CSR and CSC



Can be though of as a rank swap

Traversal Efficiency

Efficiency of different traversal patterns through the tensor is affected by representation, e.g., finding the payload for a particular coordinate…

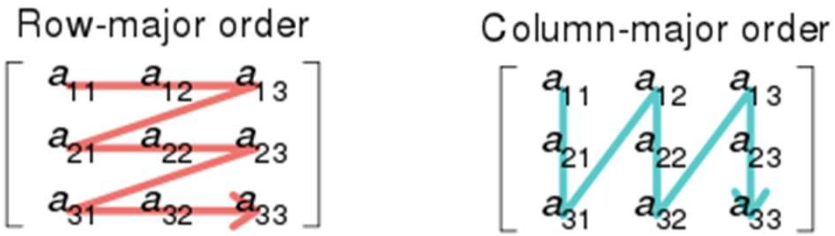
- Operations:
  - maybe(payload) = Fiber.getPayload(coordinate)
  - (coordinate, payload) = Fiber.getNext(rank_traversal_order)

Fiber.getNext() is a useful iterator and its efficiency is highly dependent on representation, both order of ranks and representation of each rank….

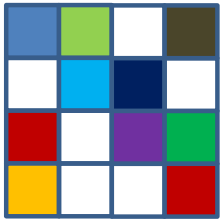
Concordant traversal orders

CSR and CSC each has a natural (or "concordant"*) traversal order

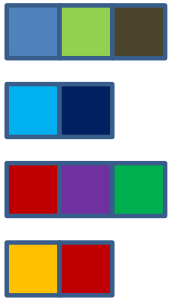
Processing Order



Original Matrix



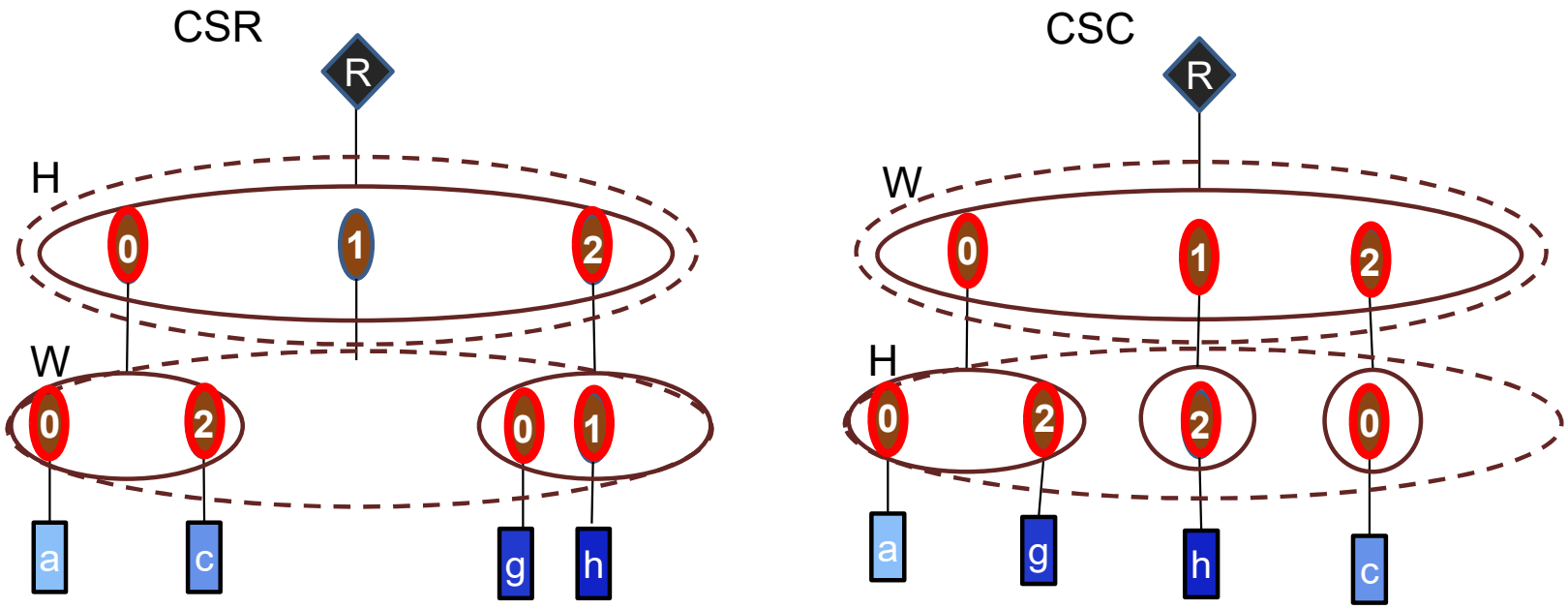
Compressed Sparse Row (CSR)



Compressed Sparse Column (CSC)

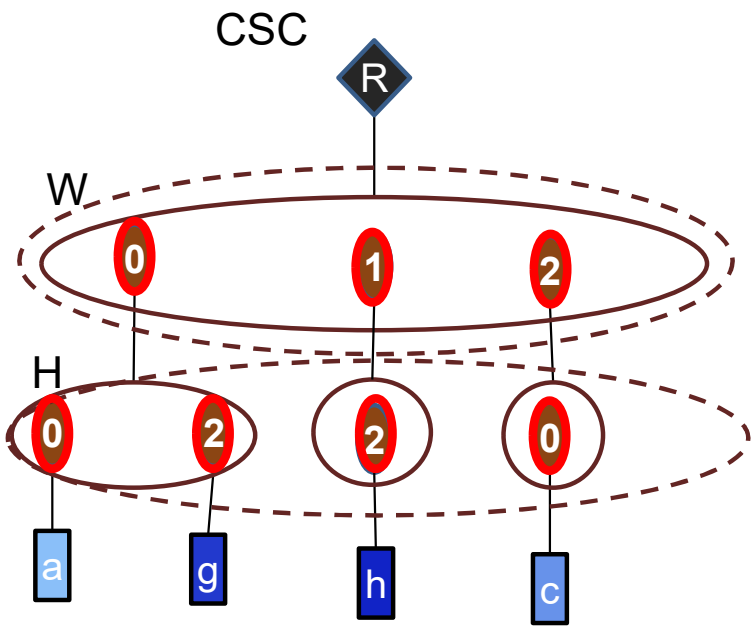
* Term from Michael Pellauer

Fibertree Concordant Traversal

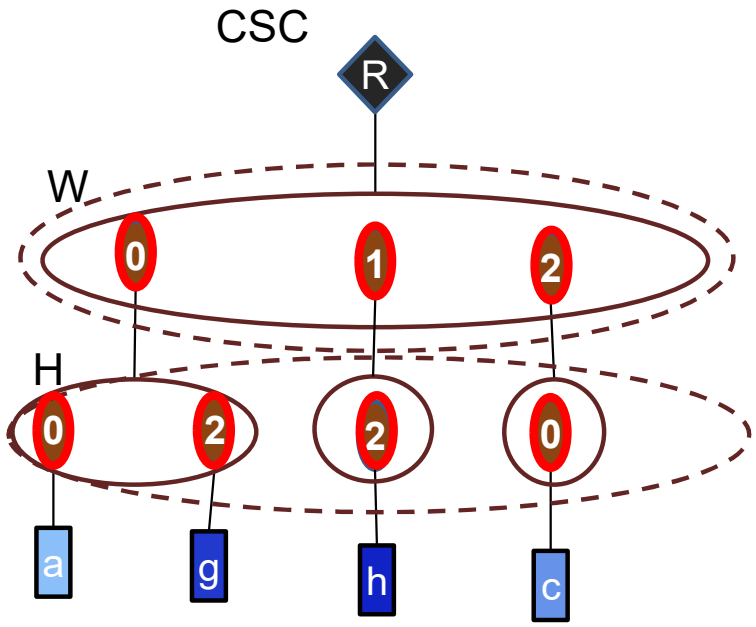


Fibertree Discordant Traversal

Discordant traversal - by row in CSC



Partially discordant traversal - W in [1,0, 2]



Example Traversal Efficiency

- Efficiency of getPayload():
  - Uncompressed – direct reference - O(1)
  - Run length encoded – linear search – O(n)
  - Hash table – multiple references and compute – O(1)
  - Coordinate/Payload list – binary search – O(log n)

- Efficiency of getNext() - (concordant traversal)
  - Uncompressed – sequential reference, good spatial locality - O(1)
  - Run length encoded – sequential reference – O(1)
  - Coordinate/Payload list - same as uncompressed

- Efficiency of getNext() (discordant traversal)
  - Essentially as good (or bad) as getPayload-method….

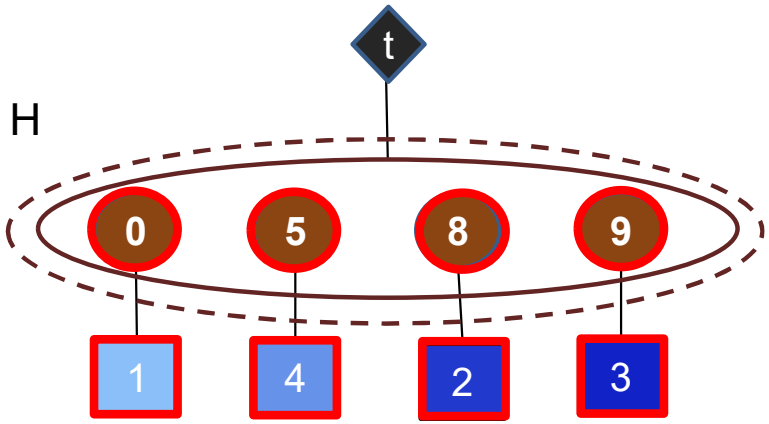
Traversing a Sparse Tensor

$$Z = T_h$$

```
# 1-D Tensor Traversal

t = Tensor(H)

z = 0
for (h, t_val) in t:
    z += t_val
```

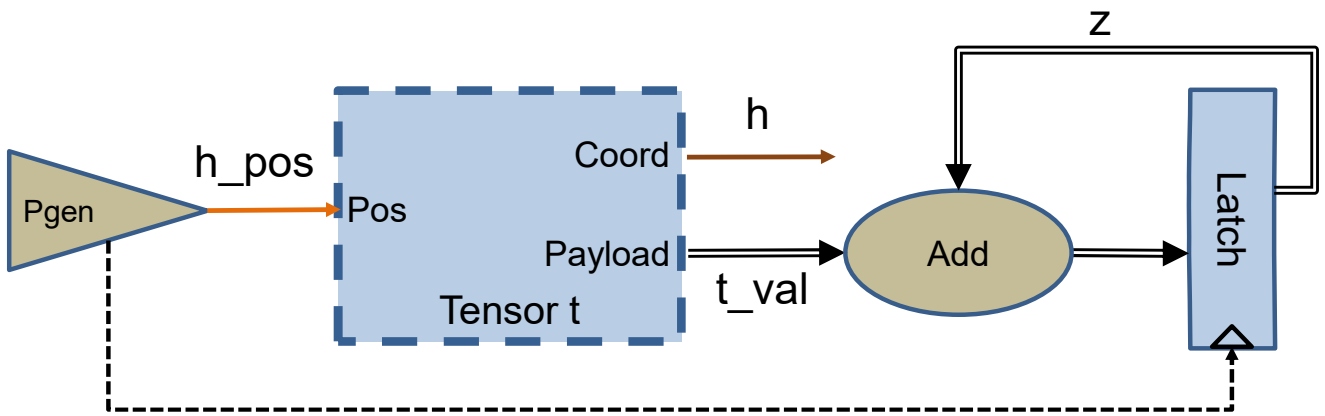
Each iteration returns a (coordinate, payload) tuple

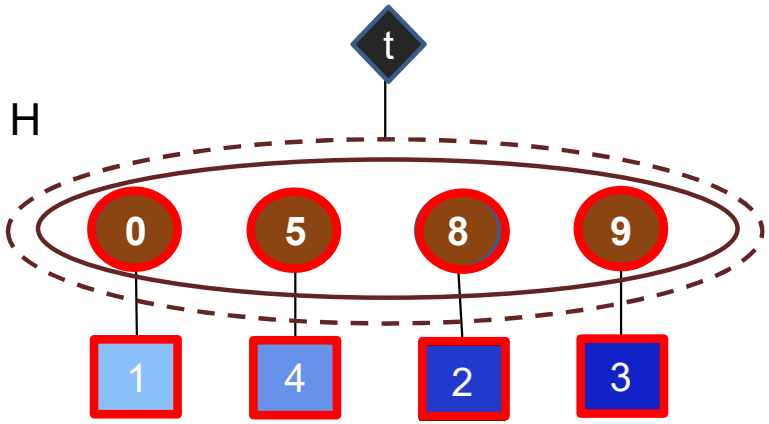
Iteration generated by repeated calls to getNext()



t_pos	h	t_val	z
0	0	1	1
1	5	4	5
2	8	2	7
3	9	3	10

Traversing a Sparse Tensor





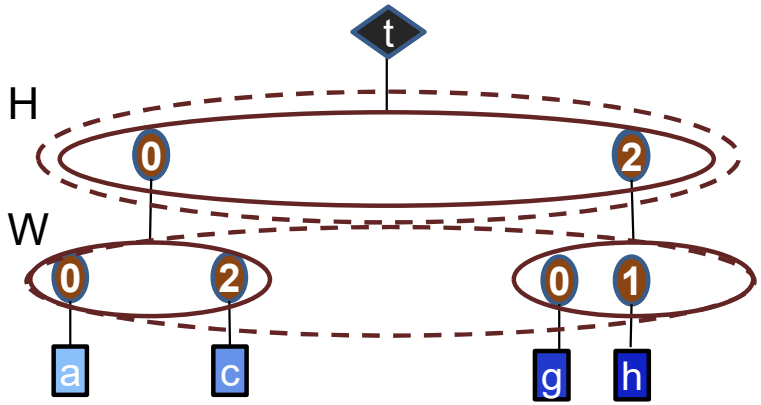
t_pos	h	t_val	z
0	0	1	1
1	5	4	5
2	8	2	7
3	9	3	10

Tensor Traversal (2-D)

$$Z = T_{h,w}$$

```
# 2-D Tensor Traversal

t = Tensor(H,W)

z = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
      z += t_val
```

Each iteration returns a (coordinate, payload) tuple

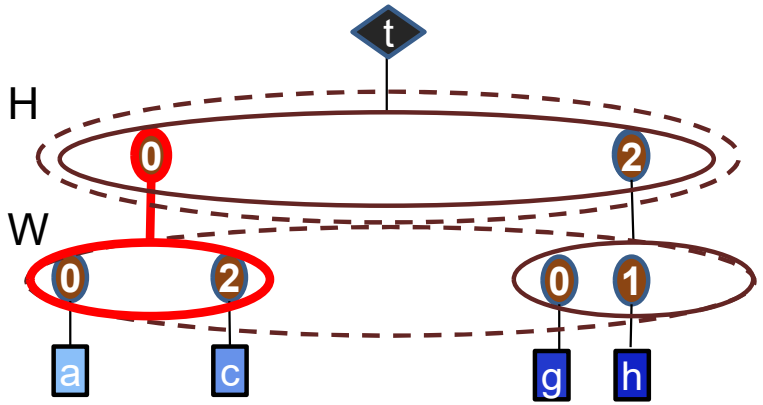


t_pos	h	t_h_pos	w	t_val
0	0	?	?	?
0	0	0	0	a
0	0	1	2	c
1	2	?	?	?
…	…	…	…	…

Tensor Traversal (2-D)

$$Z = T_{h,w}$$

```
# 2-D Tensor Traversal

t = Tensor(H,W)

z = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
      z += t_val
```

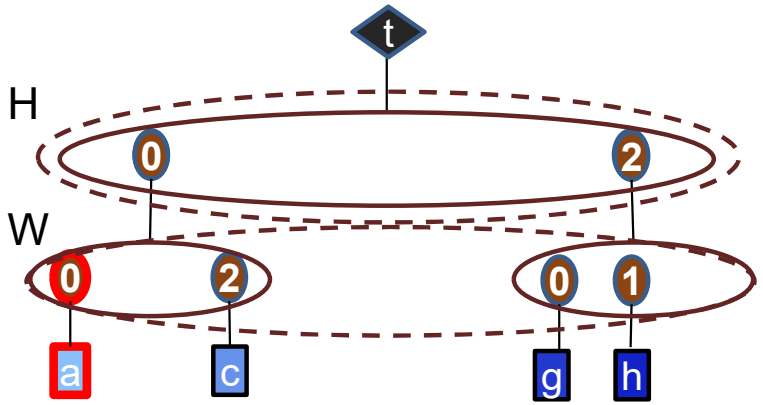


t_pos	h	t_h_pos	w	t_val
0	0	?	?	?
0	0	0	0	a
0	0	1	2	c
1	2	?	?	?
…	…	…	…	…

Tensor Traversal (2-D)

$$Z = T_{h,w}$$

```
# 2-D Tensor Traversal

t = Tensor(H,W)

z = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
      z += t_val
```

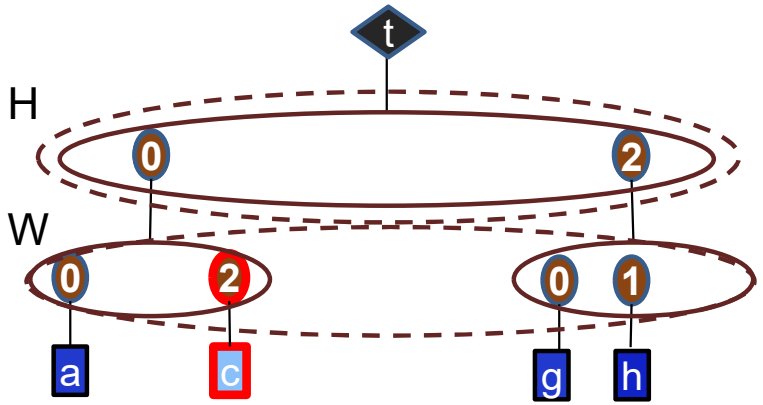


t_pos	h	t_h_pos	w	t_val
0	0	?	?	?
0	0	0	0	a
0	0	1	2	c
1	2	?	?	?
…	…	…	…	…

Tensor Traversal (2-D)

$$Z = T_{h,w}$$

```
# 2-D Tensor Traversal

t = Tensor(H,W)

z = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
      z += t_val
```

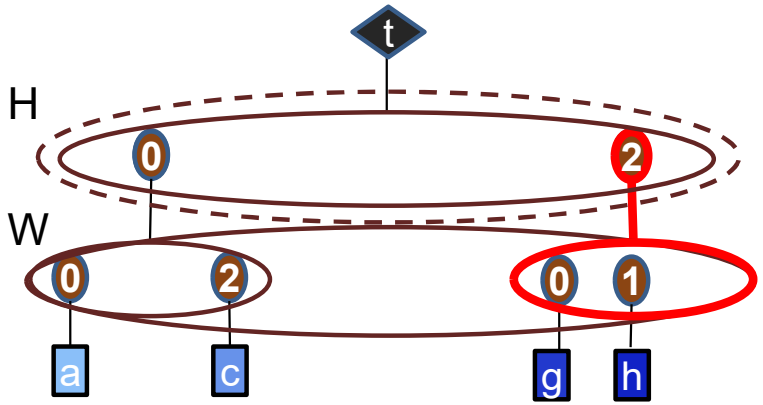


t_pos	h	t_h_pos	w	t_val
0	0	?	?	?
0	0	0	0	a
0	0	1	2	c
1	2	?	?	?
…	…	…	…	…

Tensor Traversal (2-D)

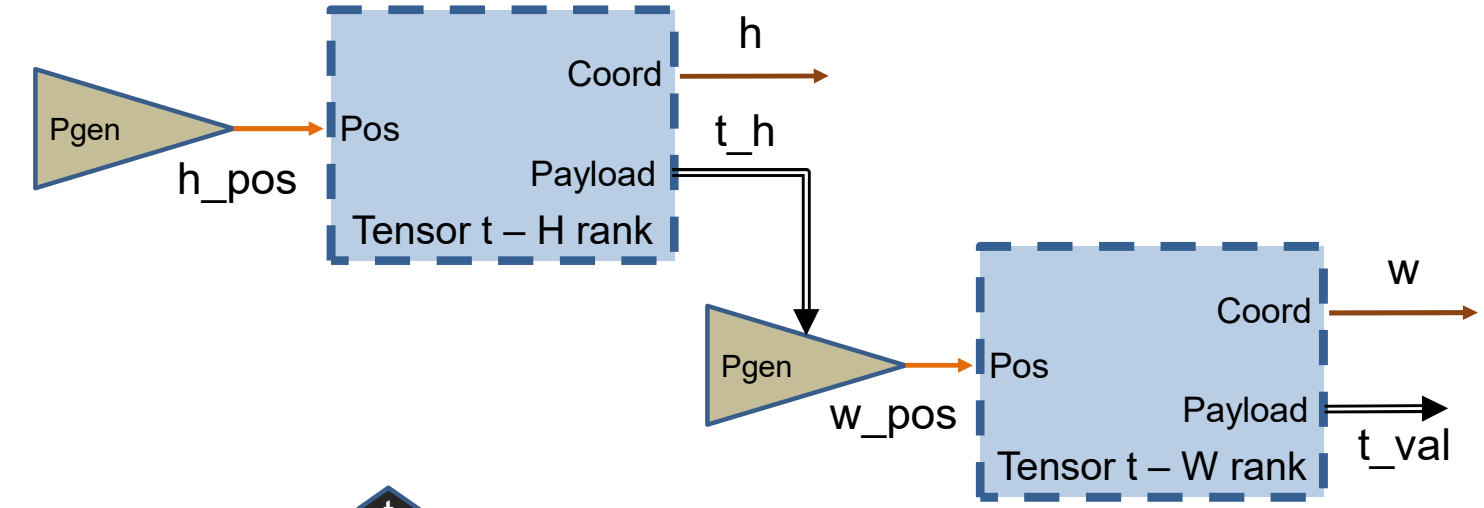
$$Z = T_{h,w}$$

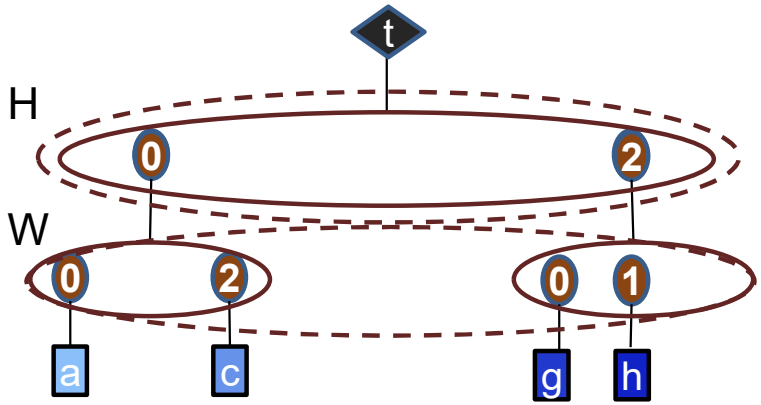
```
# 2-D Tensor Traversal

t = Tensor(H,W)

z = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
     z += t_val
```



t_pos	h	t_h_pos	w	t_val
0	0	?	?	?
0	0	0	0	a
0	0	1	2	c
1	2	?	?	?
…	…	…	…	…

Tensor Traversal (2-D)





Concordant Traversal

Abstraction versus Implementation

- Abstraction
  - An interface and semantics
    - Attributes: No implementation, data layout or timing
    - Use: implementation-agnostic understanding
    - Examples:
      - Fibers
      - Fibertree

- Implementation
  - Specific implementation of an abstract spec
    - Attributes: Concrete implementation, data layout and timing
    - Examples:
      - Fibers → uncompressed array, coordinate/payload list
      - Fiber-tree → CSR, CSC, CSF, COO…

Tensor Traversal (CSR Style)

```
# 2-D Tensor Traversal (CSR)

t_segs = Array(H)
t_coords = Array(W)
t_vals = Array(W)

sum = 0
for t_h_pos in [0,H):
  h = t_h_pos
  t_w_start = t_segs[t_h_pos]
  t_w_len = t_segs[t_h_pos+1]-t_w_start
  for t_w_pos in [t_w_start, t_w_len):
     h = t_coords[t_w_pos]
     t_val = t_vals[t_w_pos]
     sum += t_val
```

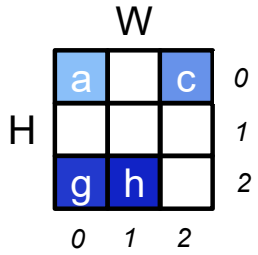
For uncompressed rank coordinate equals position

Coordinates not actually used in this example

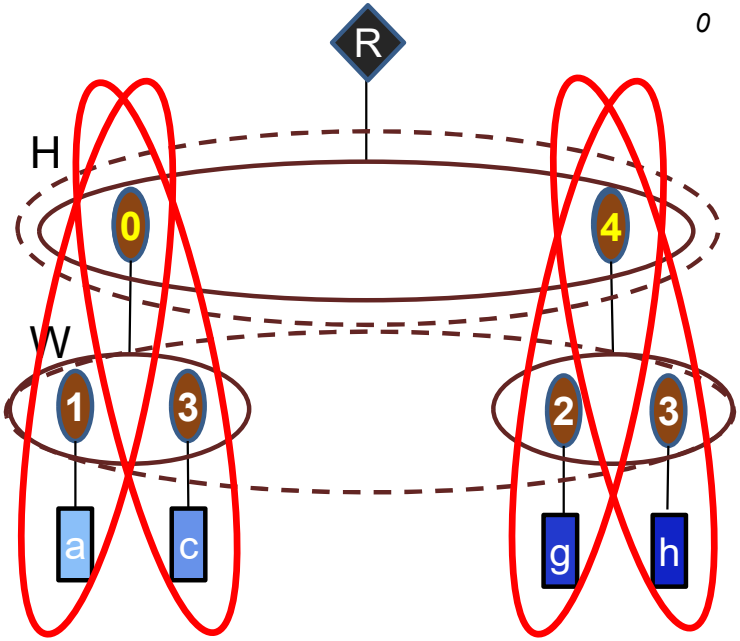
Merging Ranks

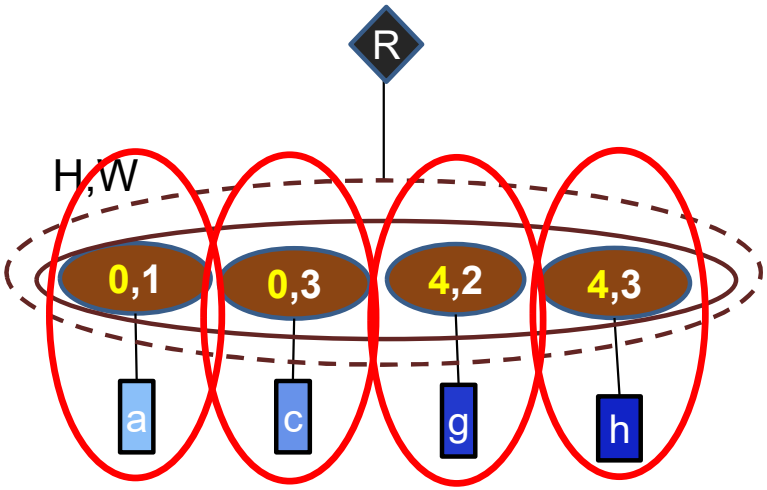
For efficiency one can form new representations where the data structure for two or more ranks are combined.

Tensor-<C,C>(H,W)



Tensor-<C²>(H,W)



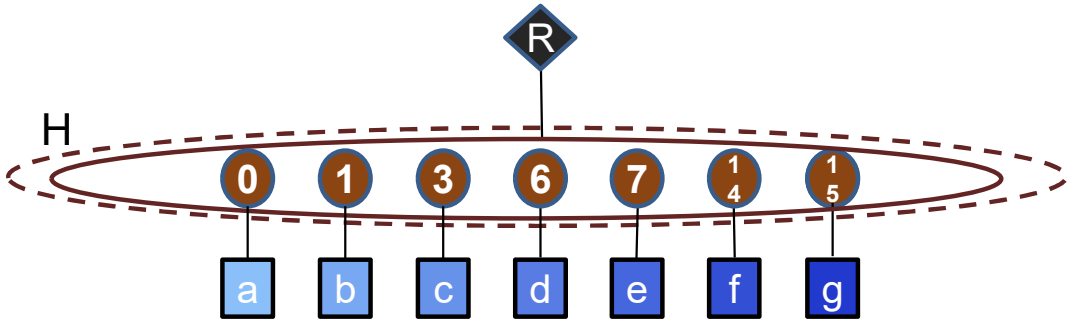


Merging Ranks

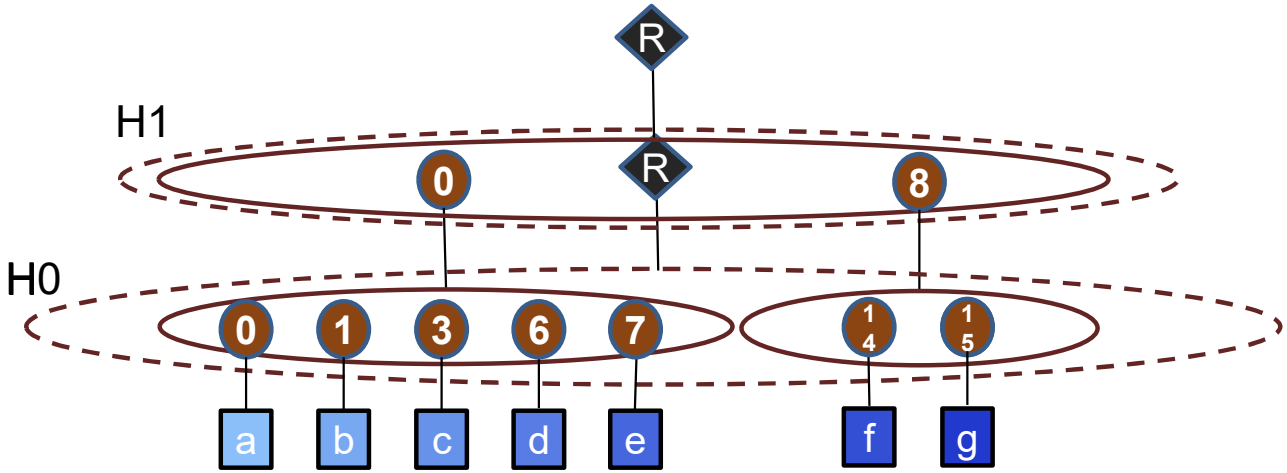
- For efficiency one can form new representations where the data structure for two or more ranks are combined:

- Examples:
  - Tensor-(C^2)
    List of (coordinate tuple,payload) - COO
  - Tensor-(H^2)
    - Hash table with coordinate tuple as key
  - Tensor-(U^2)
    - Flattened array
    - Coordinates can be recovered with modulo arithmetic on “position”
  - Tensor-(R^2)
    - Flattened run-length encoded sequence

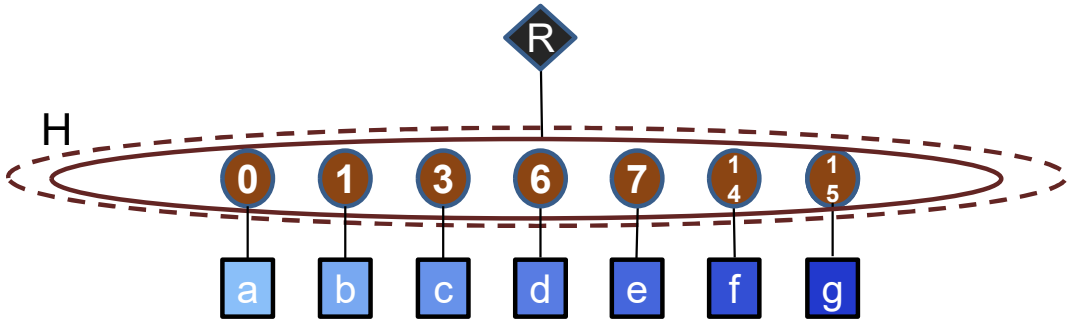
Splitting Fibers – Coordinate Space



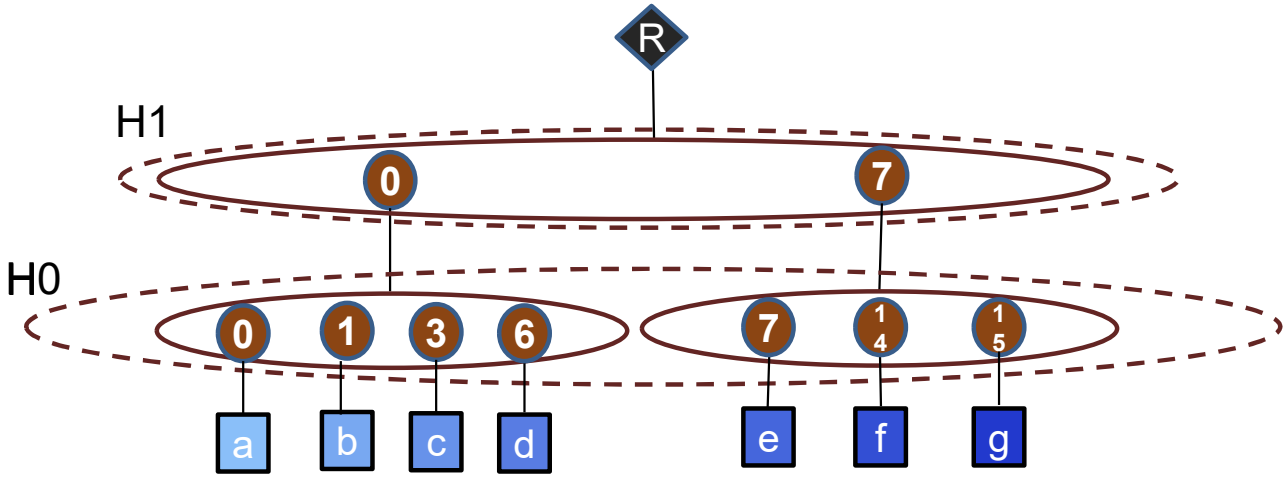
Split uniformly by coordinates (groups of 8 coordinates)



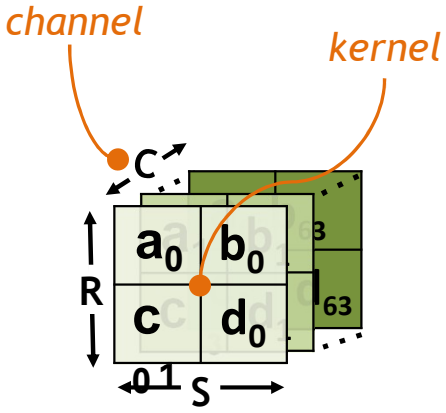
Splitting Fibers – Position Space



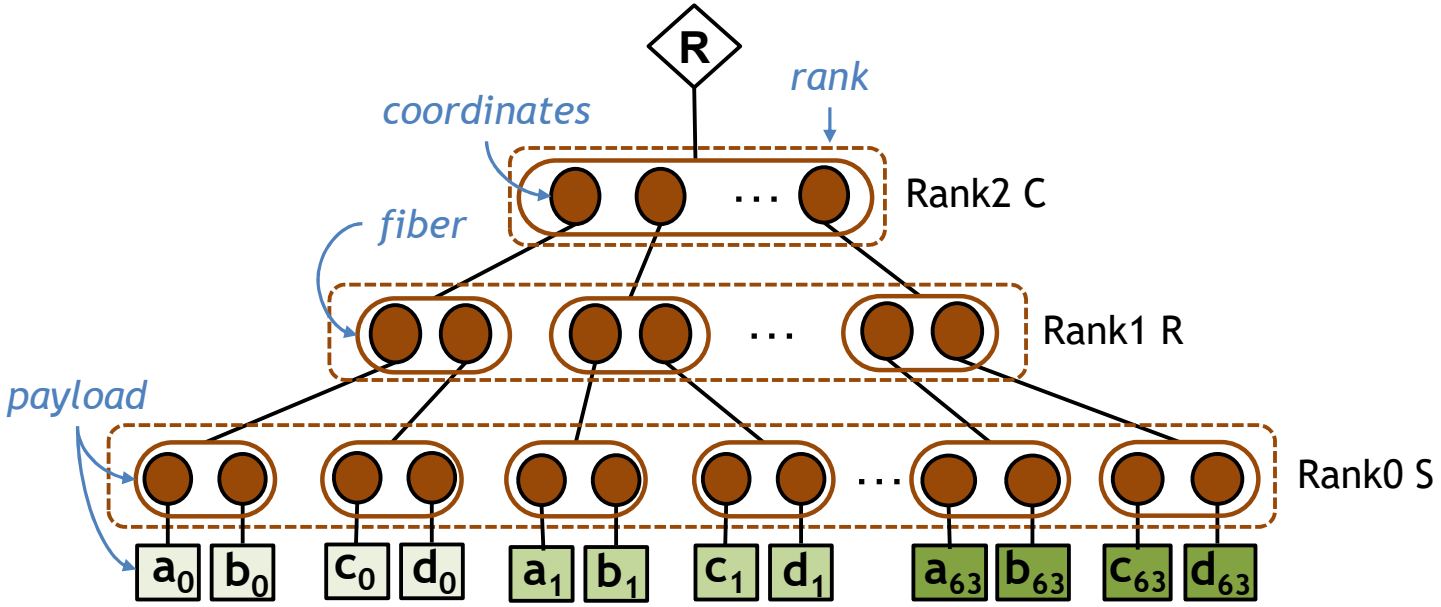
Split evenly by occupancy (groups of 4)



Fibertree Representation of Weight Pruning



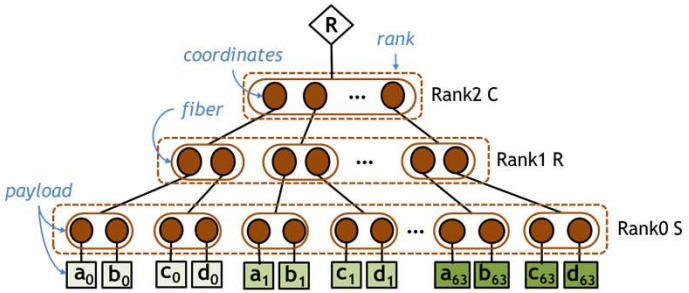
Example 3D Weight Tensor



Fibertree Representation

Each dimension in the original tensor is represented as a rank in the tree

Specification of Channel-based Sparsity

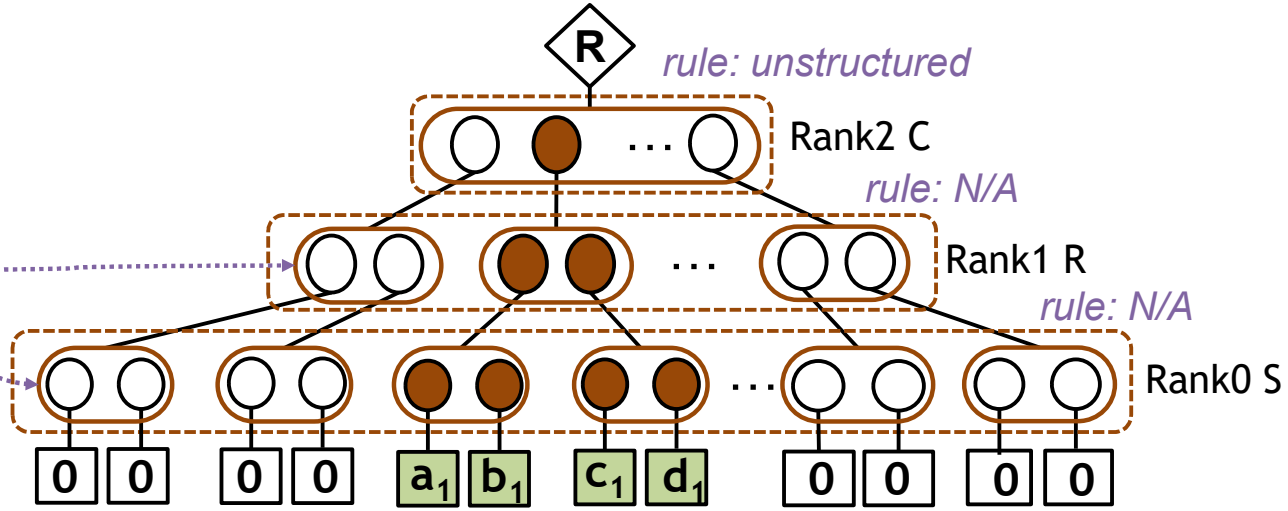


Fibertree Representation of Dense Tensor

Apply Per-rank Pruning Rule

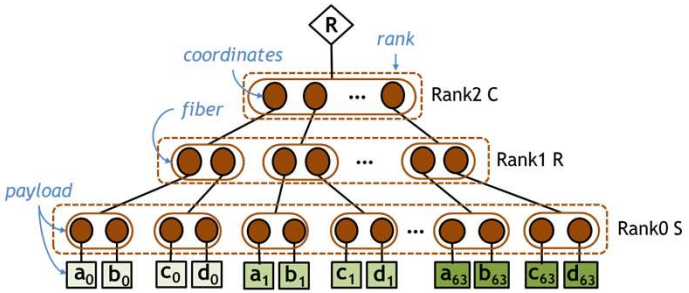
(identical rules are applied to all fibers in each rank)

Implicit pruning of the coordinates in the payload



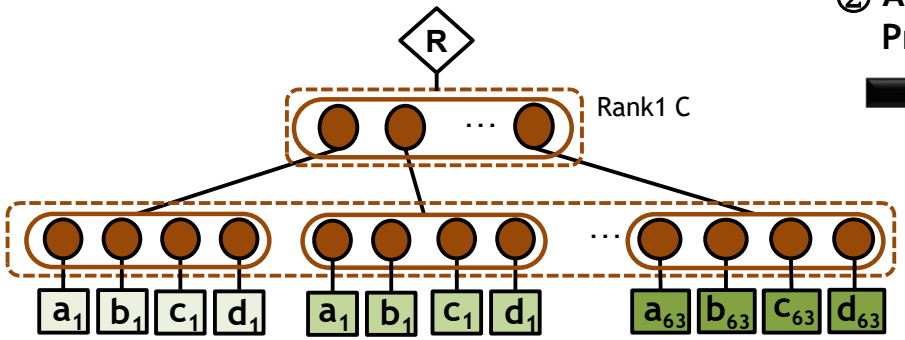
$C(unstructured) \rightarrow R \rightarrow S$

Flattening: Specification of Sub-kernel Sparsity

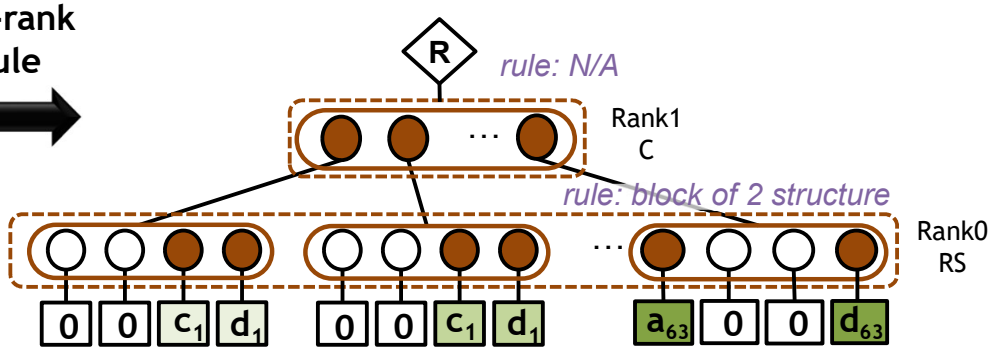


Fibertree Representation of Dense Tensor

① Flatten a Subset of Ranks

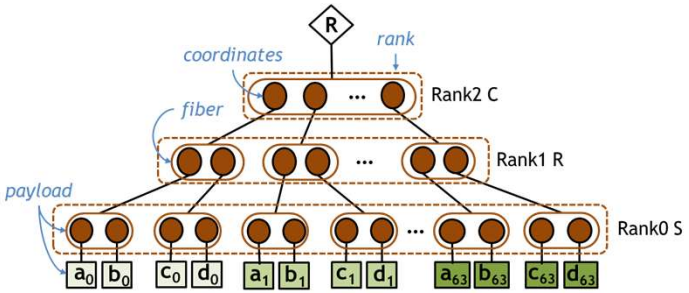


② Apply Per-rank Pruning Rule



$C \rightarrow RS\ (2\ of\ 4)$

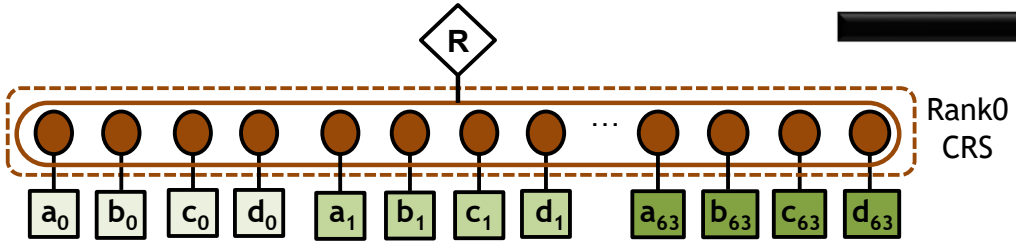
Flattening: Specification of Unstructured Sparsity

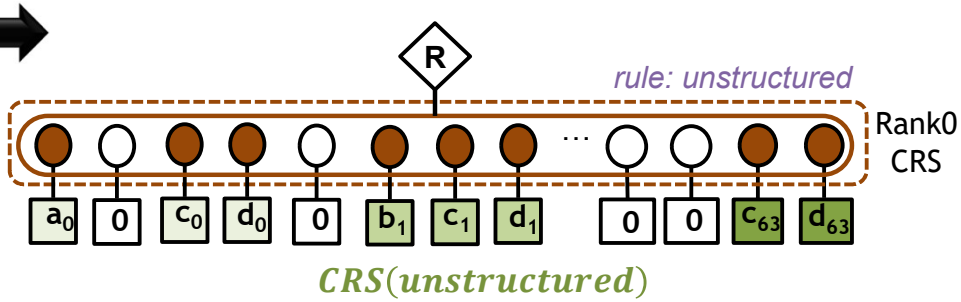


Fibertree Representation of Dense Tensor

① Flatten All Ranks

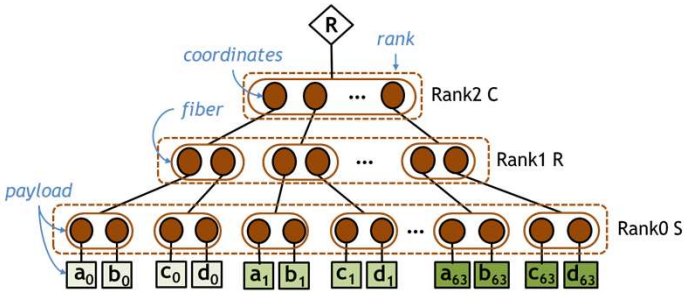
② Apply Per-rank Pruning Rule





$CRS(unstructured)$

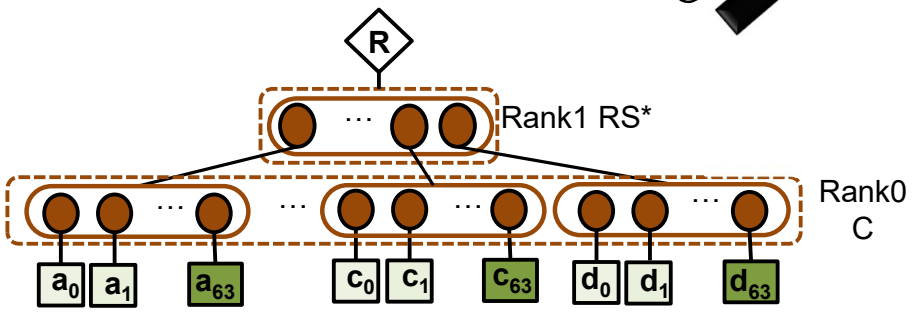
Reordering & Partitioning: Specification of 2:4 Sparsity

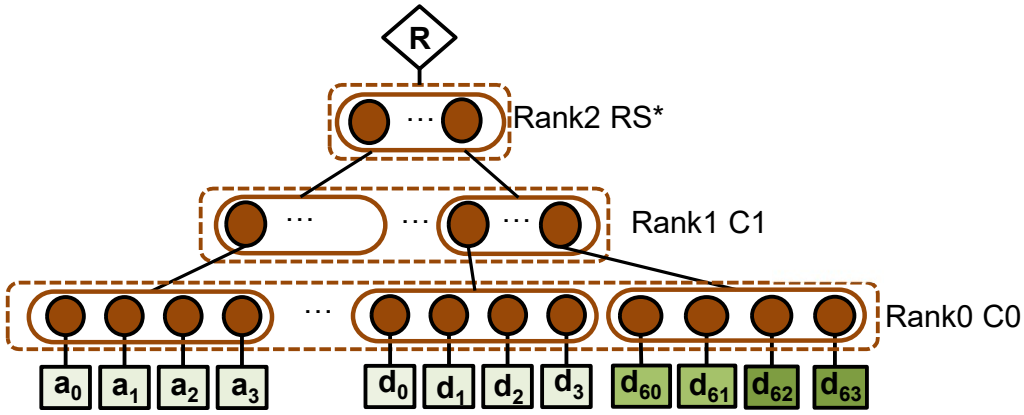


Fibertree Representation of Dense Tensor

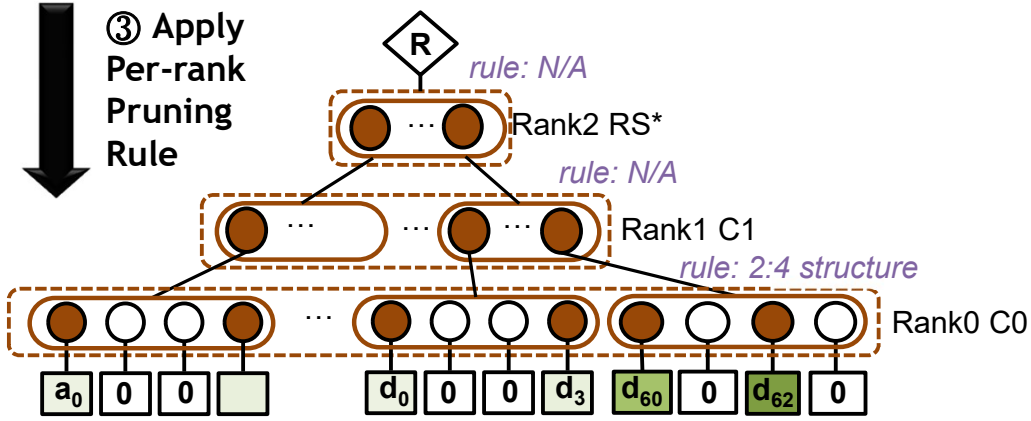
① Reorder Ranks

② Partition Rank C





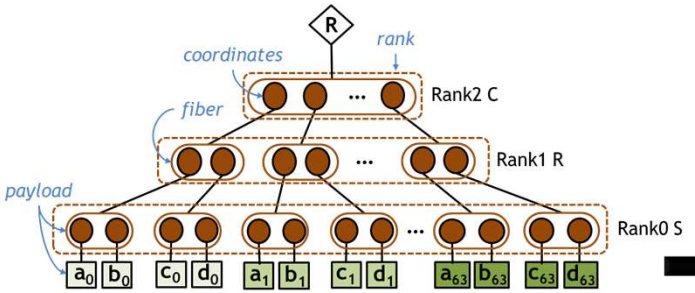
③ Apply Per-rank Pruning Rule



$$RS \rightarrow C_1 \rightarrow C_0(2{:}4)$$

*flattened for ease of presentation, not required

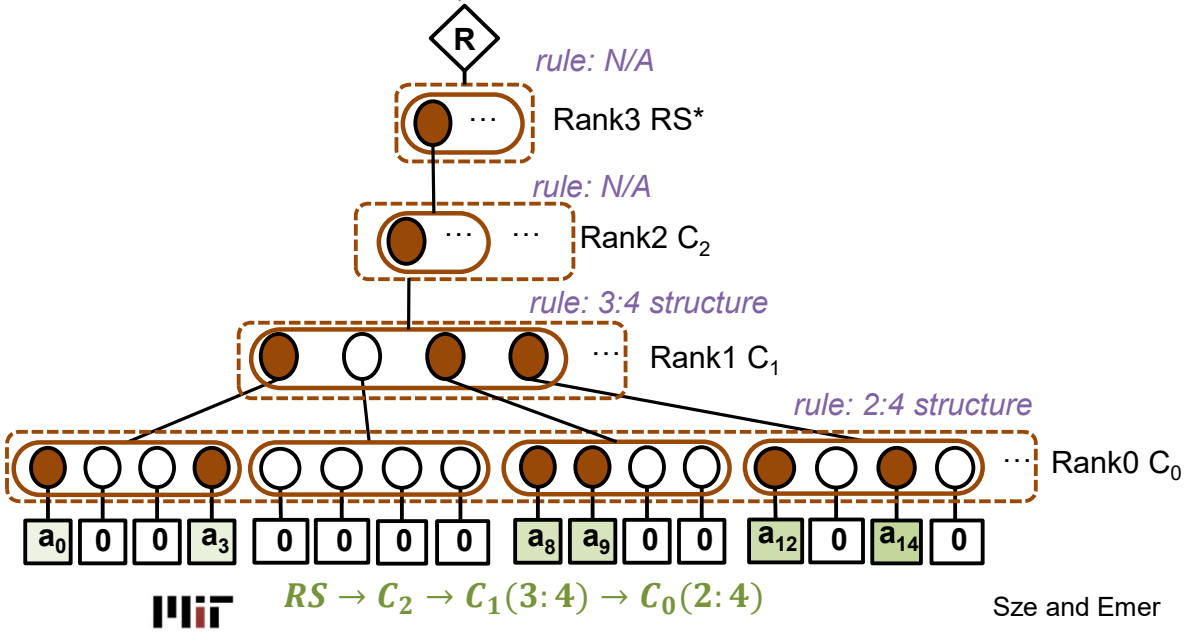
Hierarchical Structured Sparsity (HSS)



Fibertree Representation of Dense Tensor

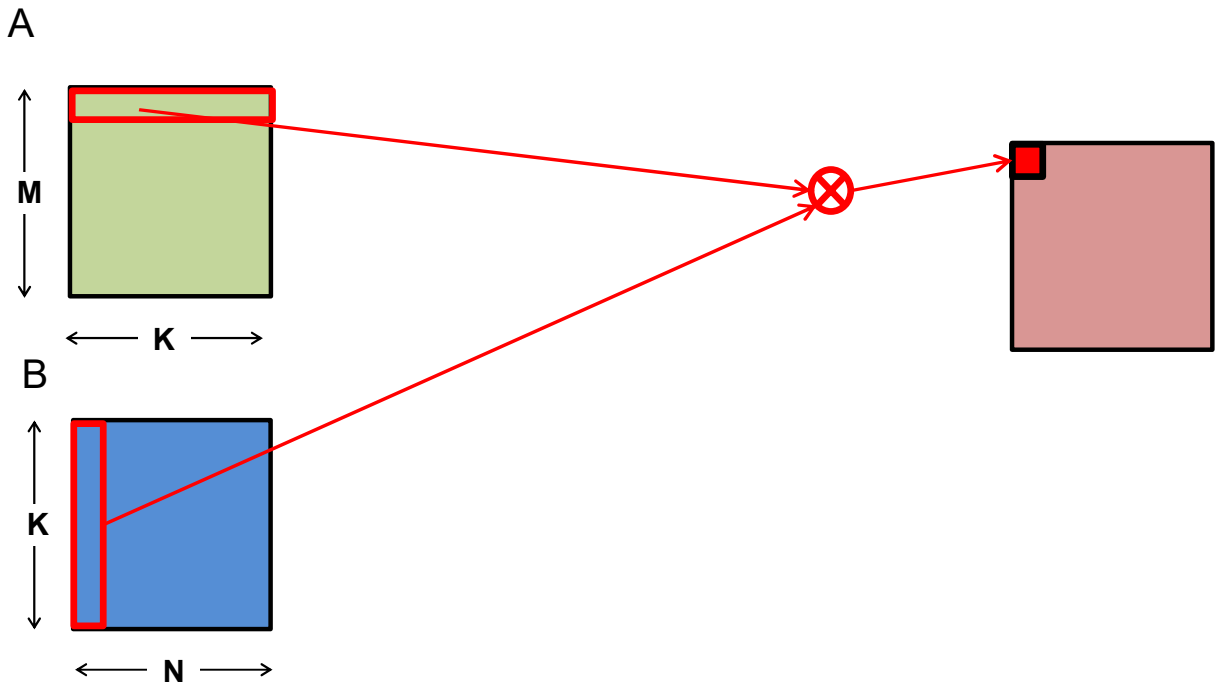
① Reorder Ranks
② Partition Rank C into N ranks (N>=2) , e.g., N=3 as shown below
③ Apply Per-rank Pruning Rule

- N-1 rank HSS defined as
  - $RS \rightarrow C_{N-1} \rightarrow C_{N-2}(G_{N-2}: H_{N-2}) \rightarrow \ldots \rightarrow C_1(3:4) \rightarrow C_0(2:4)$
- HSS qualitative difference: allows pruning rules for more thank one ranks
- HSS provides a systematic and modularized way to represent a large number of sparsity degrees



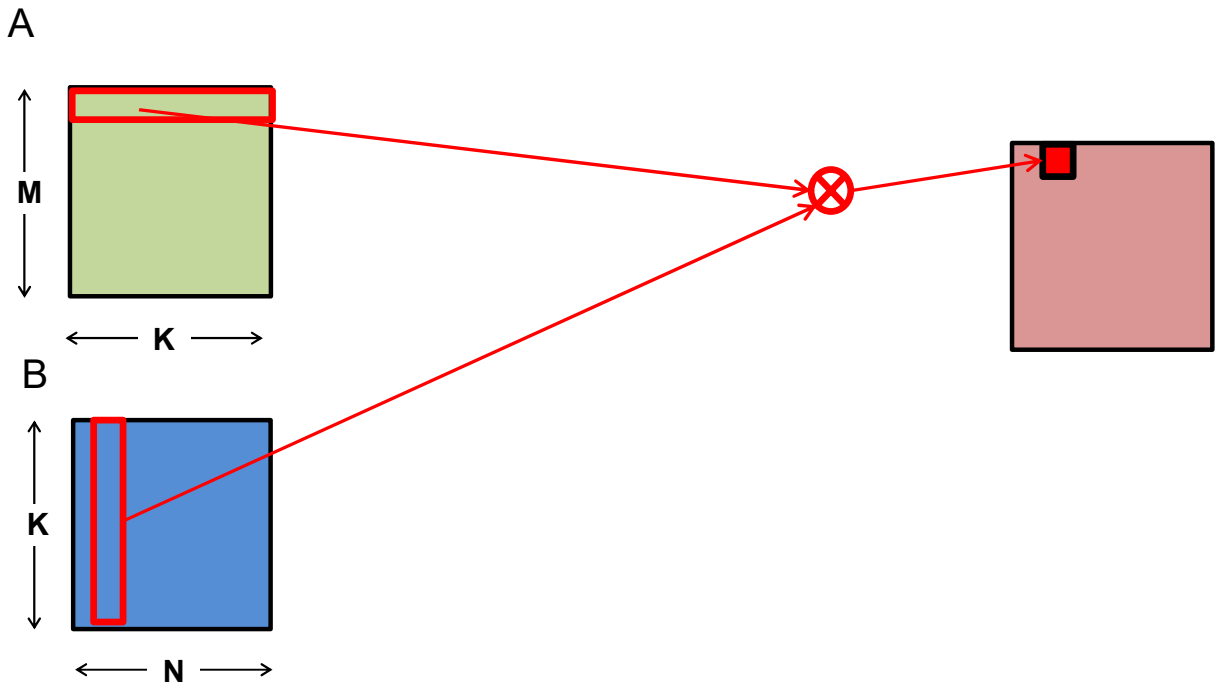
*flattened for ease of presentation, not required

Matrix Multiply



Problem Specification → Algorithm → Data Format + Schedule

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

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

Indices are coordinates

Einsum – Matrix Multiply

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

- **Shared indices -> intersection**

Einsum – Matrix Multiply

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

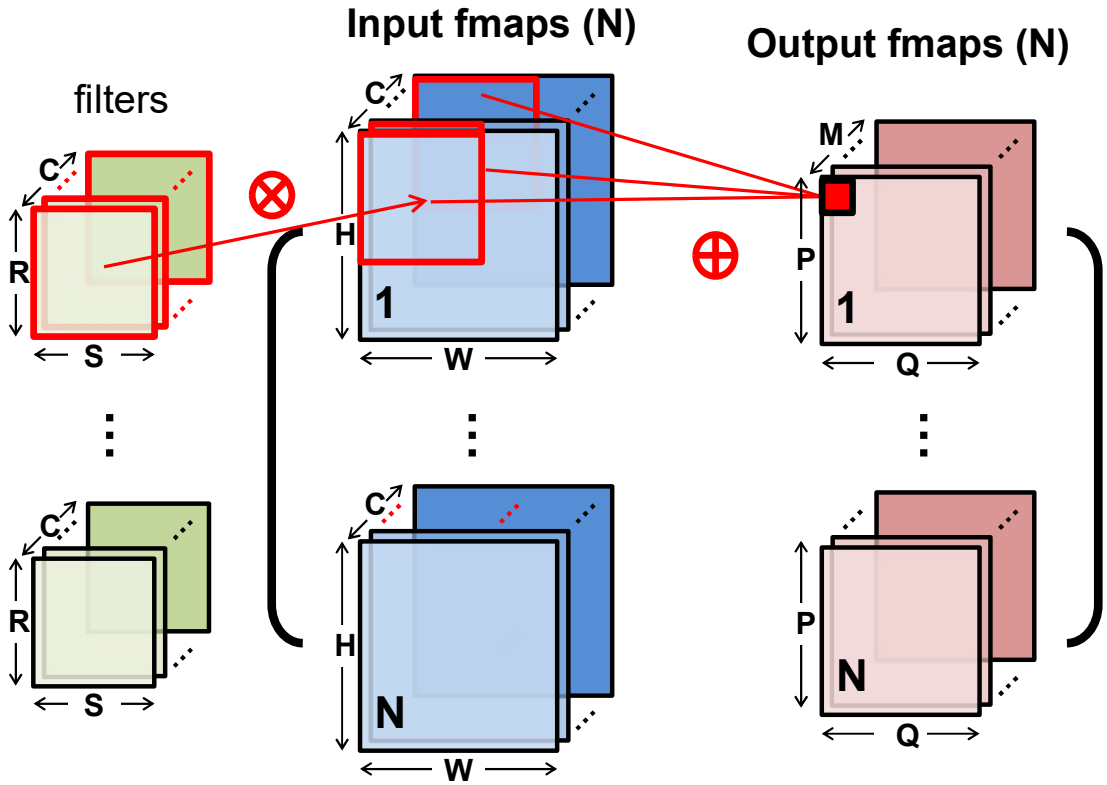
- **Shared indices -> intersection**
- **Contracted indices -> reduction**

Einsum – Matrix Multiply

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

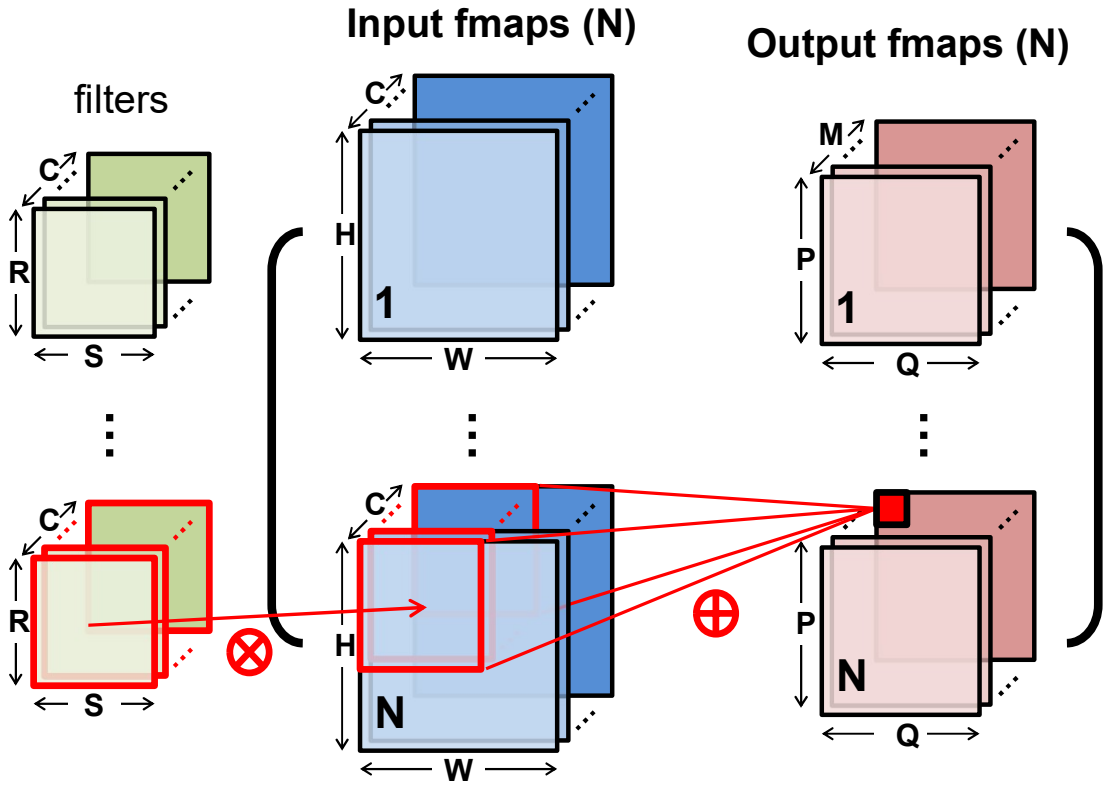
- **Shared indices -> intersection**
- **Contracted indices -> reduction**
- **Uncontracted indices -> populate output point**

Convolution (CONV) Layer



Problem Specification → Algorithm → Data Format + Schedule

Convolution (CONV) Layer



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]

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]

Aspects of Scheduling - Sparsity

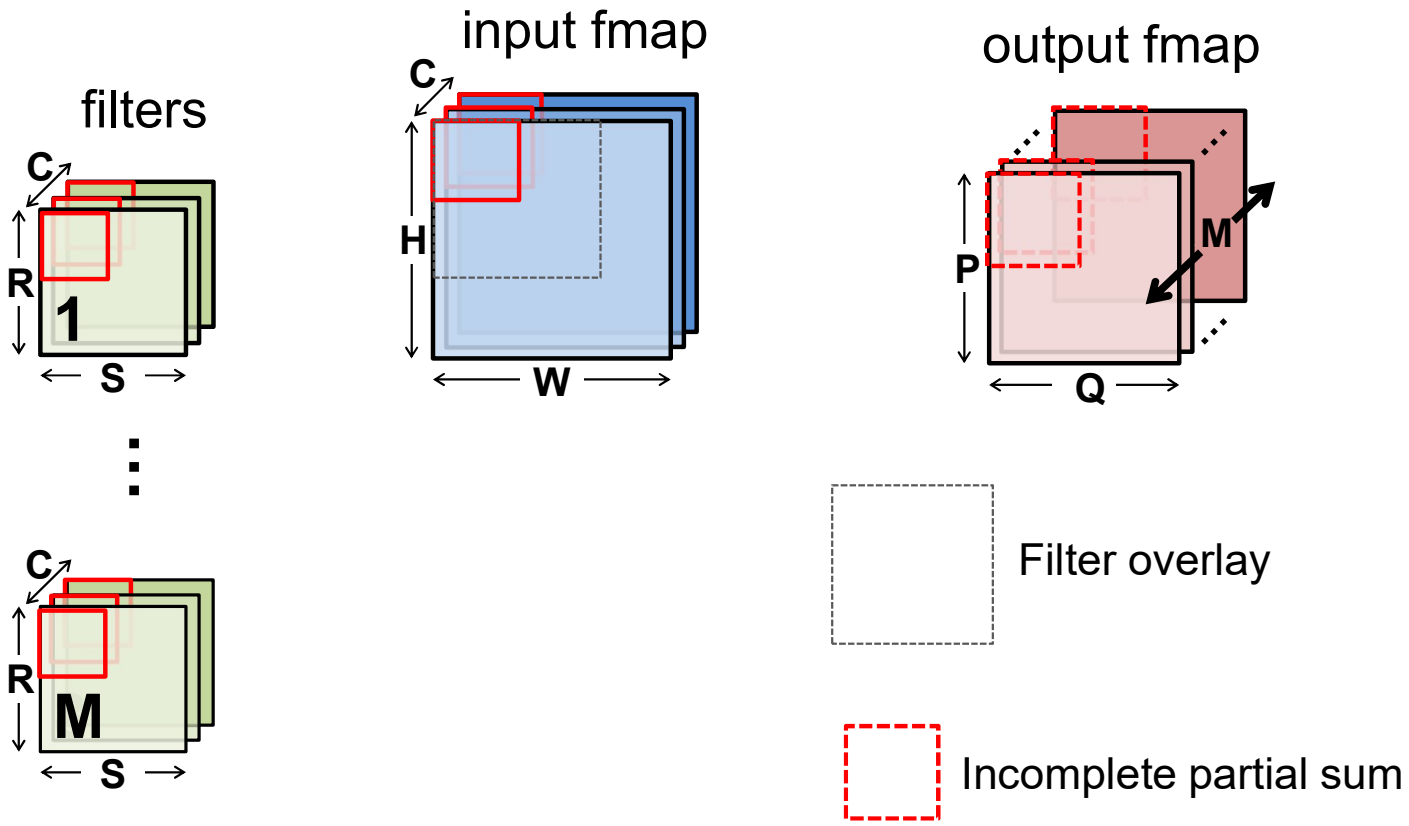
Format:
Choose tensor representations to save storage space and energy associated with zero accesses

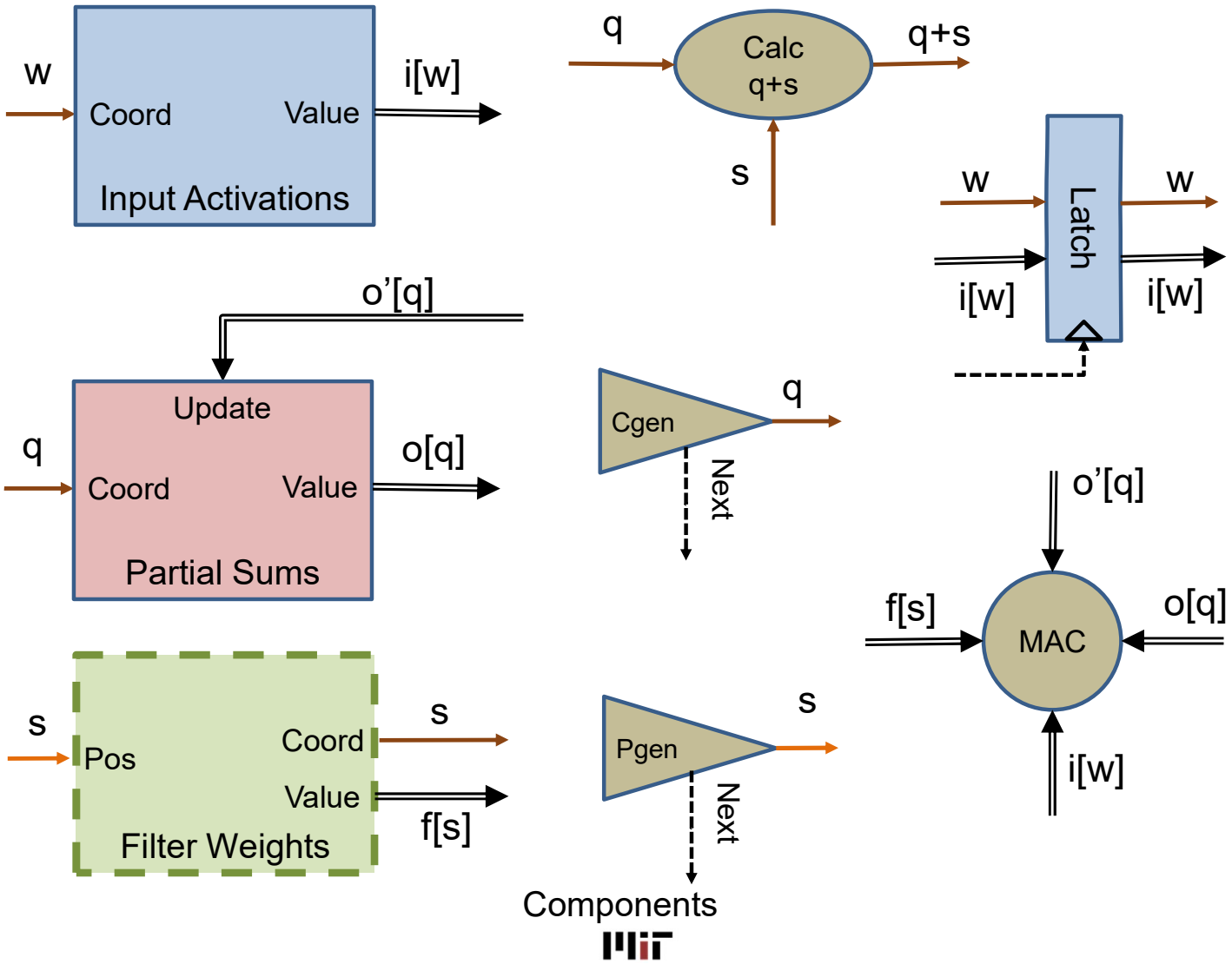
Gating:
Explicitly eliminate ineffectual storage accesses and computes by letting the hardware unit staying idle for the cycle to save energy

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

CONV: Exploiting Sparse Weights

CONV Layer


filters
C
R
1
S
C
R
M
S
input fmap
C
H
W
output fmap
M
P
Q
Filter overlay
Incomplete partial sum


w
Coord
Value
i[w]
Input Activations
q
Calc
q+s
q+s
s
w
Latch
w
i[w]
i[w]
o'[q]
Update
q
Coord
Value
o[q]
Partial Sums
Cgen
q
Next
o'[q]
f[s]
MAC
o[q]
i[w]
s
Pos
Coord
Value
s
f[s]
Filter Weights
Pgen
s
Next
Components

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

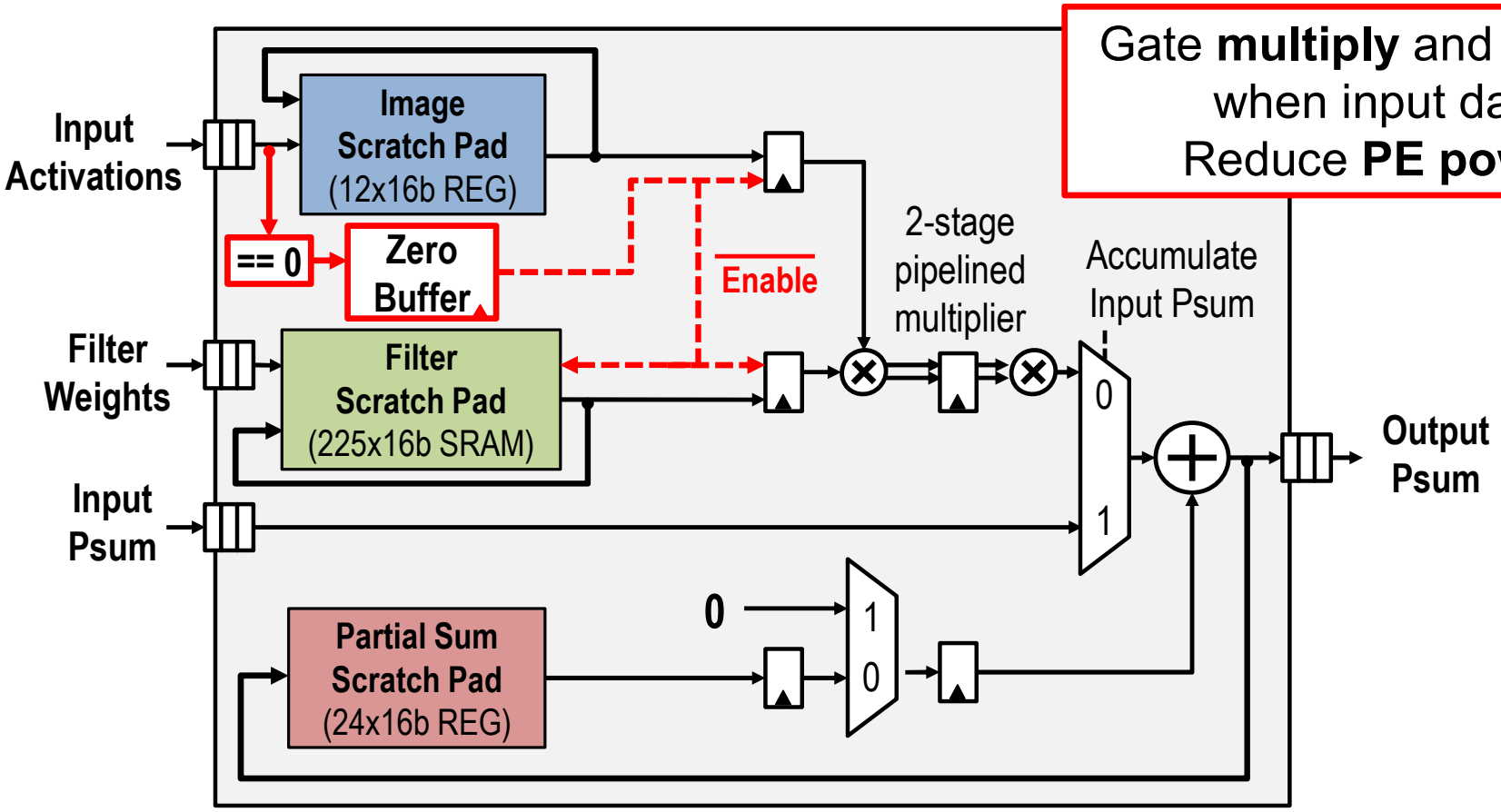

```
i = Array(W)        # Input activations
f = Array(S)        # Filter weights
o = Array(Q)        # Output activations

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 – Gating



Gate **multiply** and **memory reads** when input data is zero.
Reduce **PE power** by **45%**

[**Chen**, *ISSCC* 2016]

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

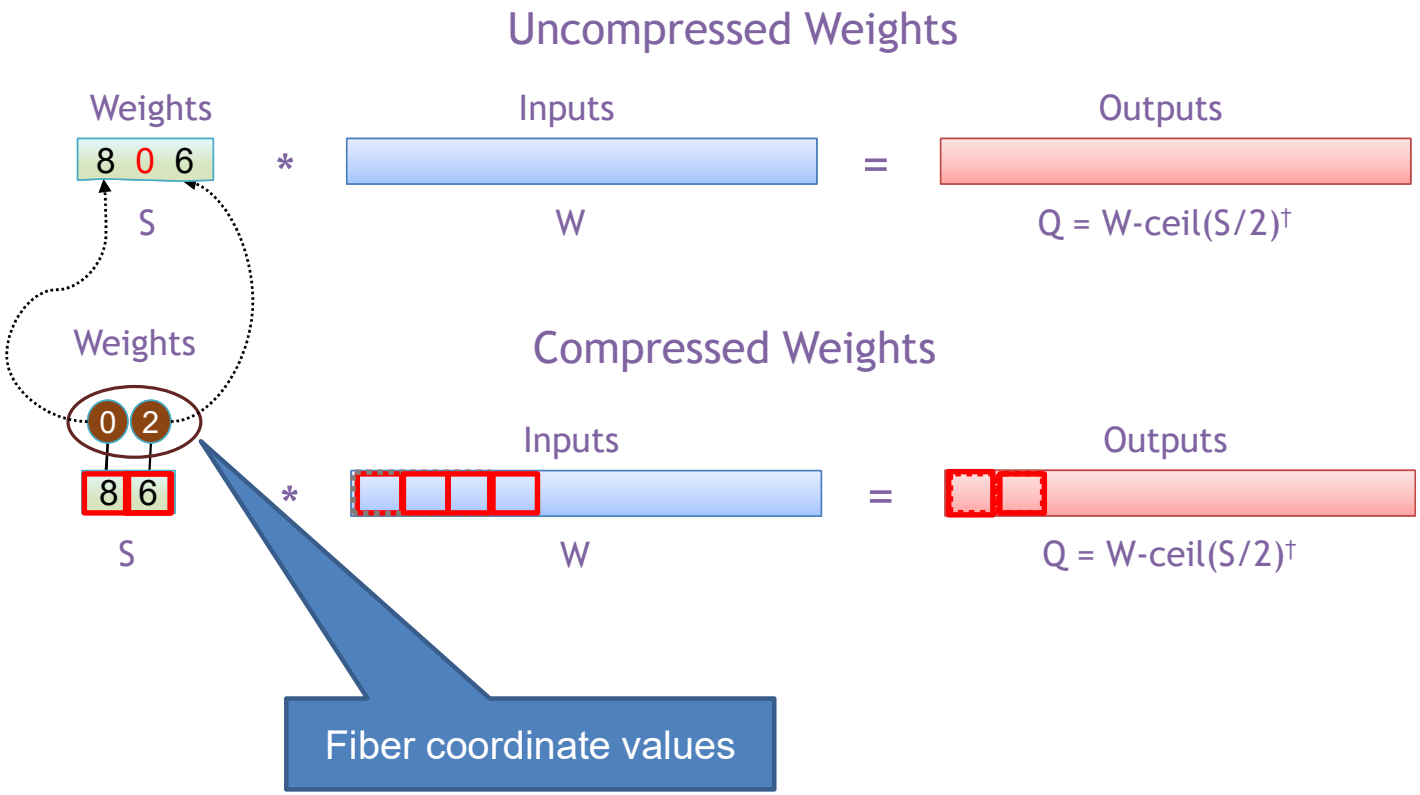
What are the tensor representations of "i" and "f"? Abstract, Fibertree

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

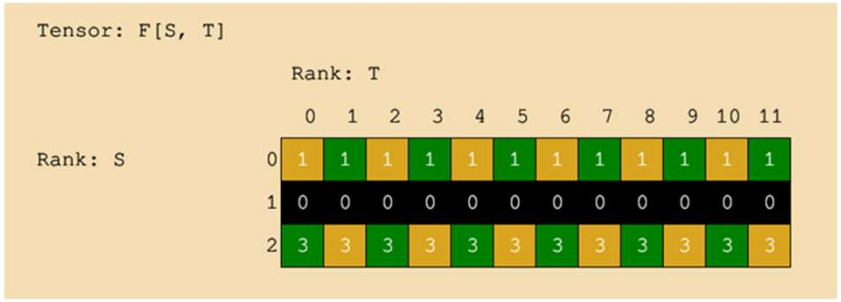
What is "s"? Coordinate

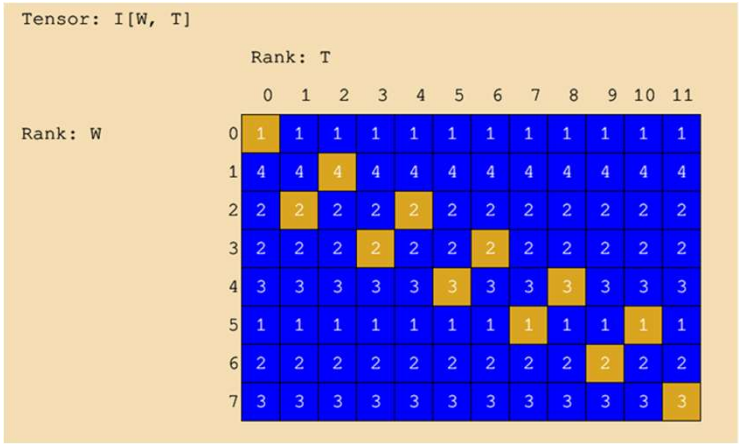
What is "f_val"? Payload, Value

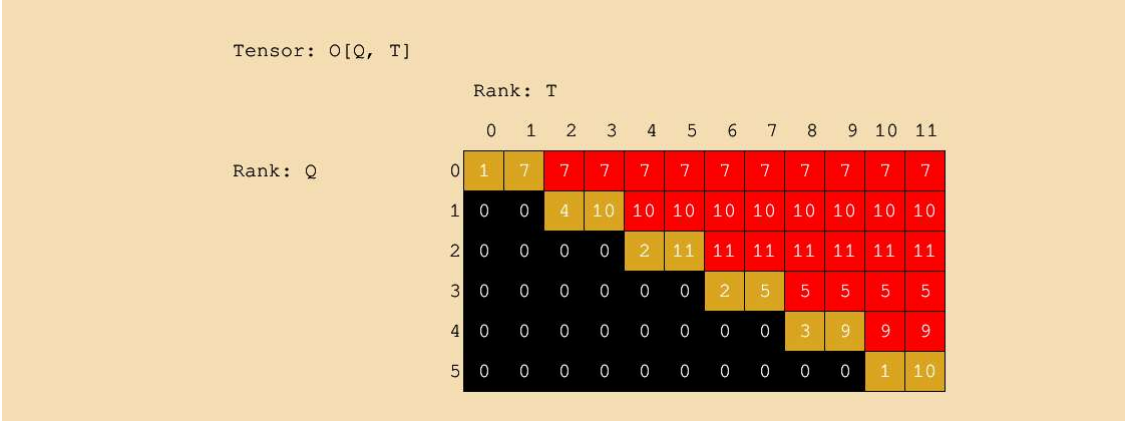
The traversal of "f" will be? Concordant

For sparser weights, this implementation will be? Faster

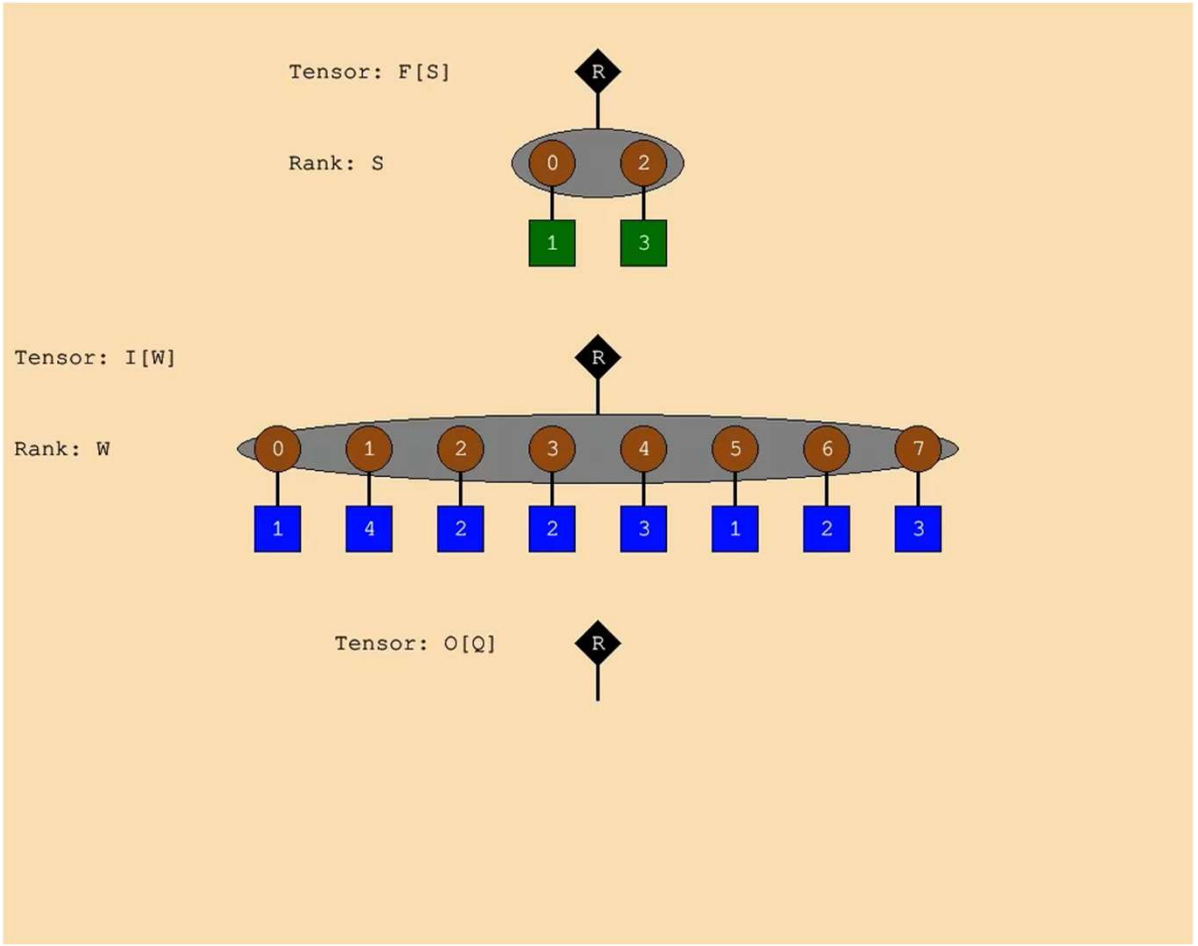
Output Stationary – Sparse Weights



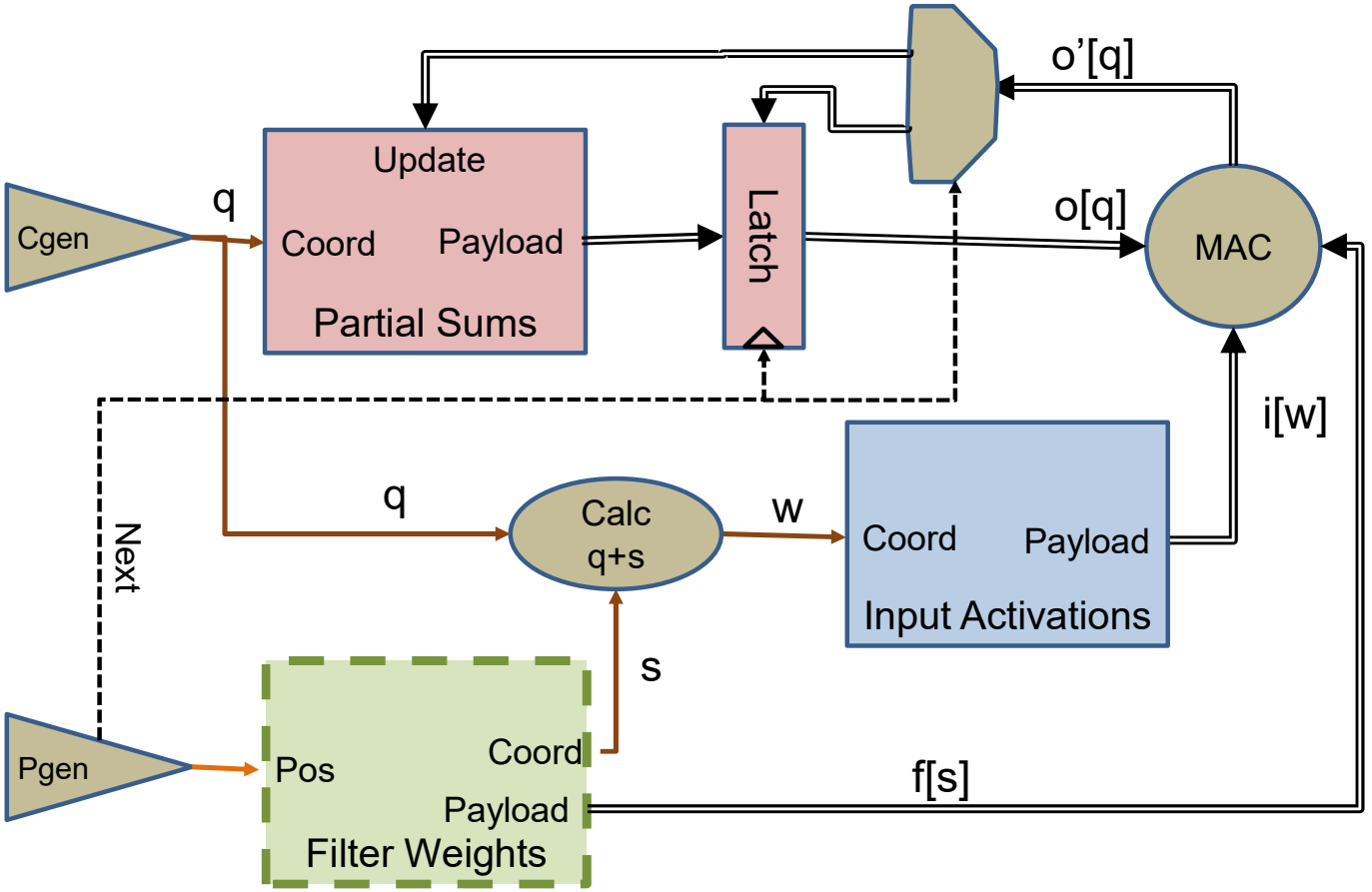




Output Stationary – Sparse Weights



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

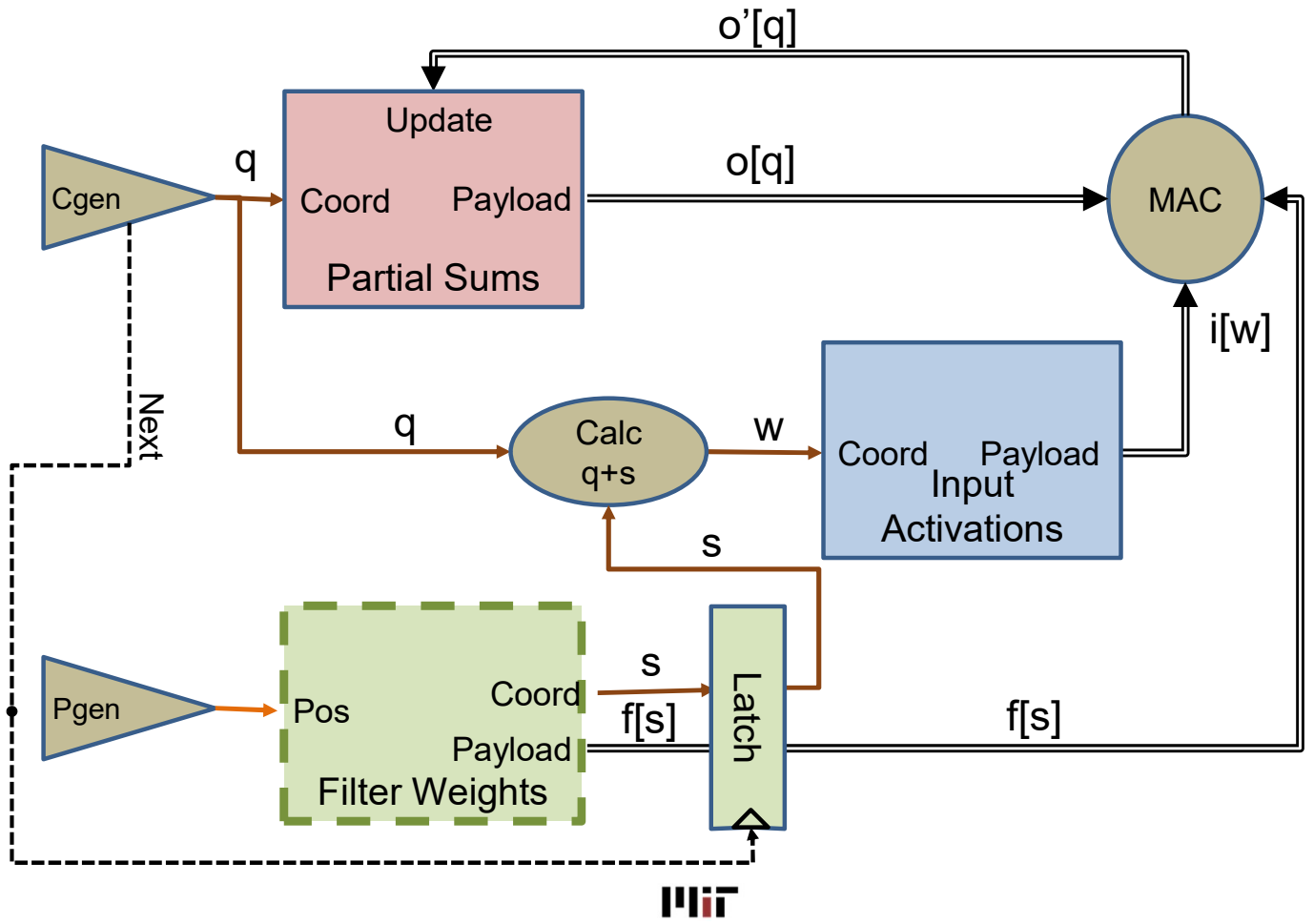
What dataflow is this?

Weight stationary

Where does it exploit sparsity?

Weights

Weight Stationary - Sparse Weights



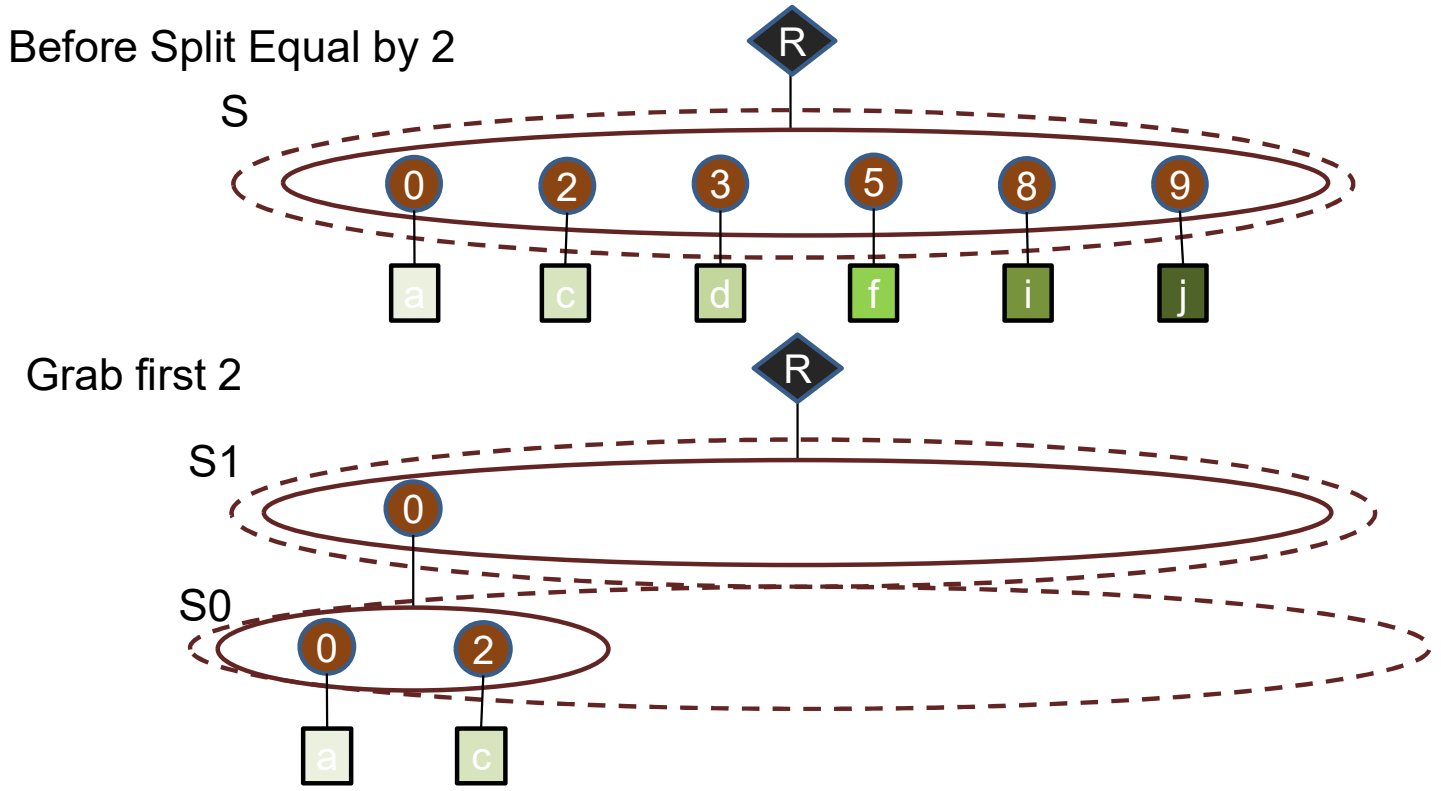
To Extend to Other Dimensions of DNN

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

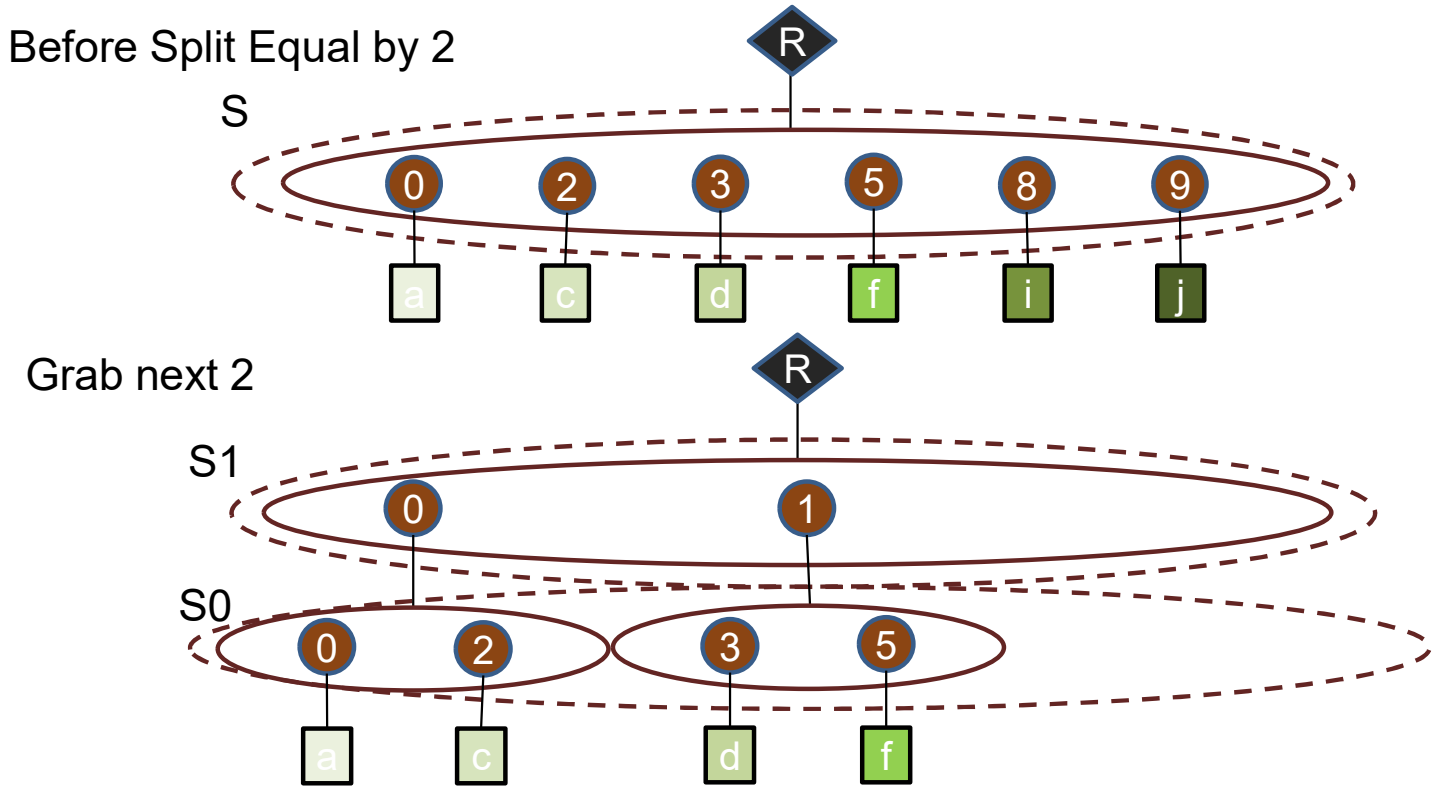
- **Add parallelism…**

Consider working on two weights at a time

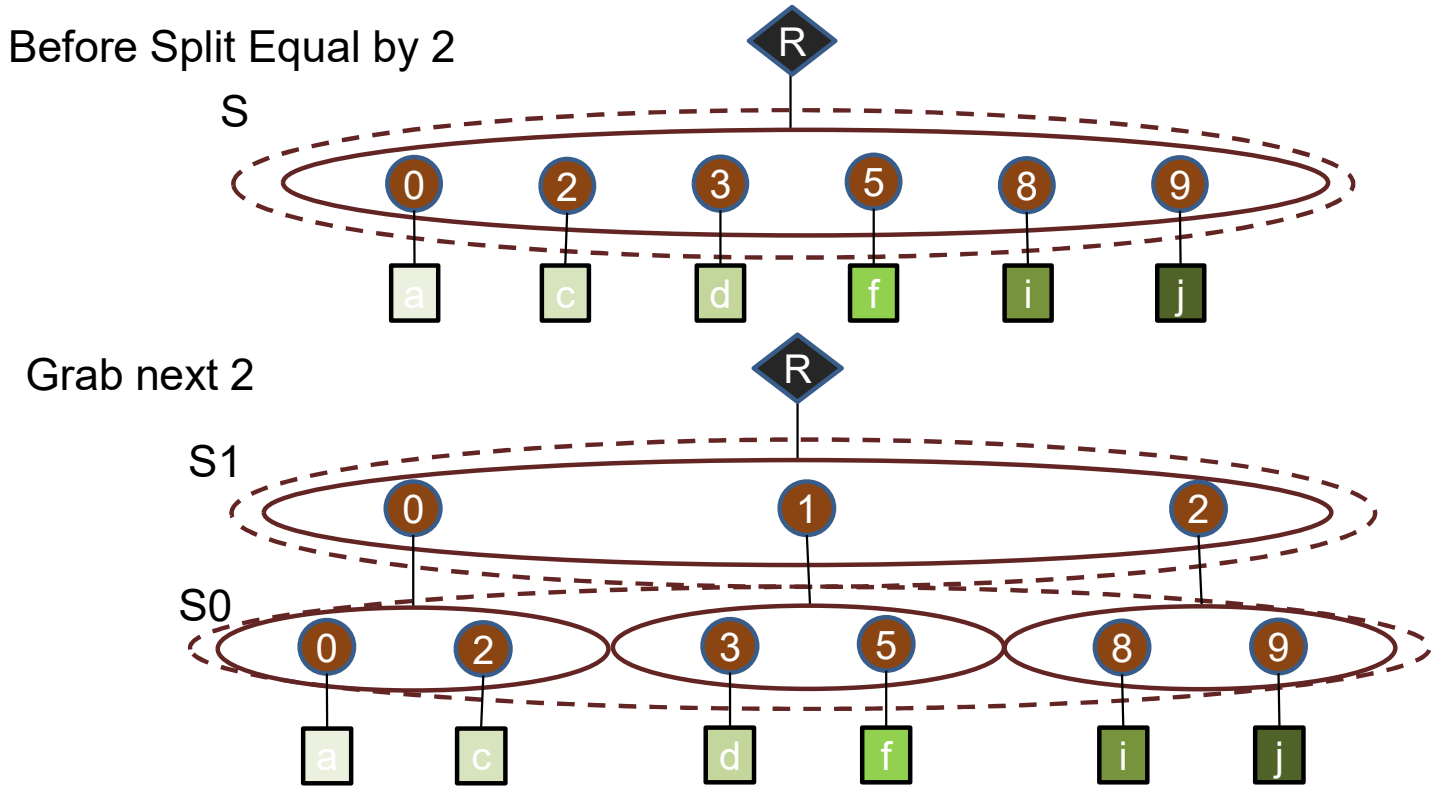
Fiber Splitting Equally in Position Space



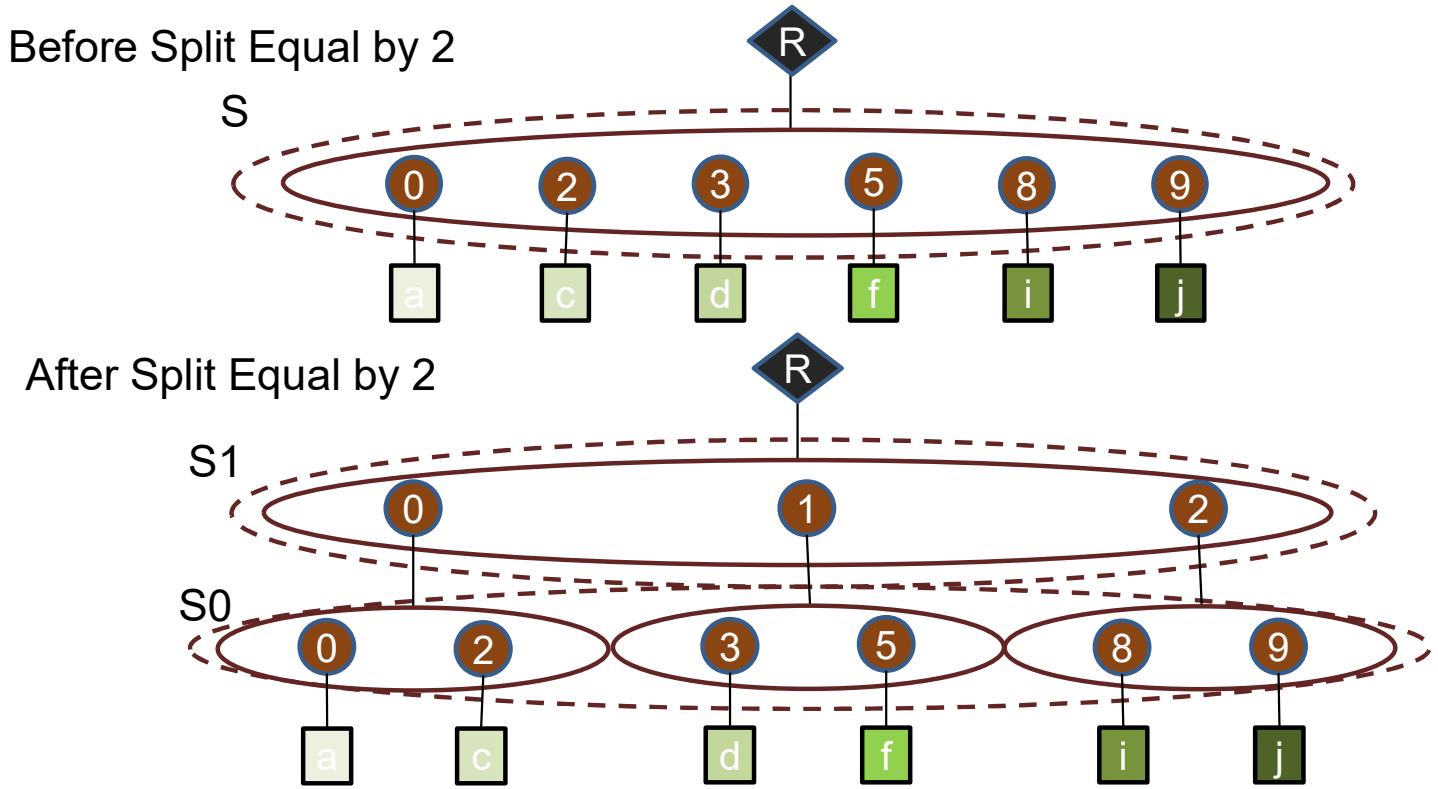
Fiber Splitting Equally in Position Space



Fiber Splitting Equally in Position Space



Fiber Splitting Equally in Position Space



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

Uncompressed tiling

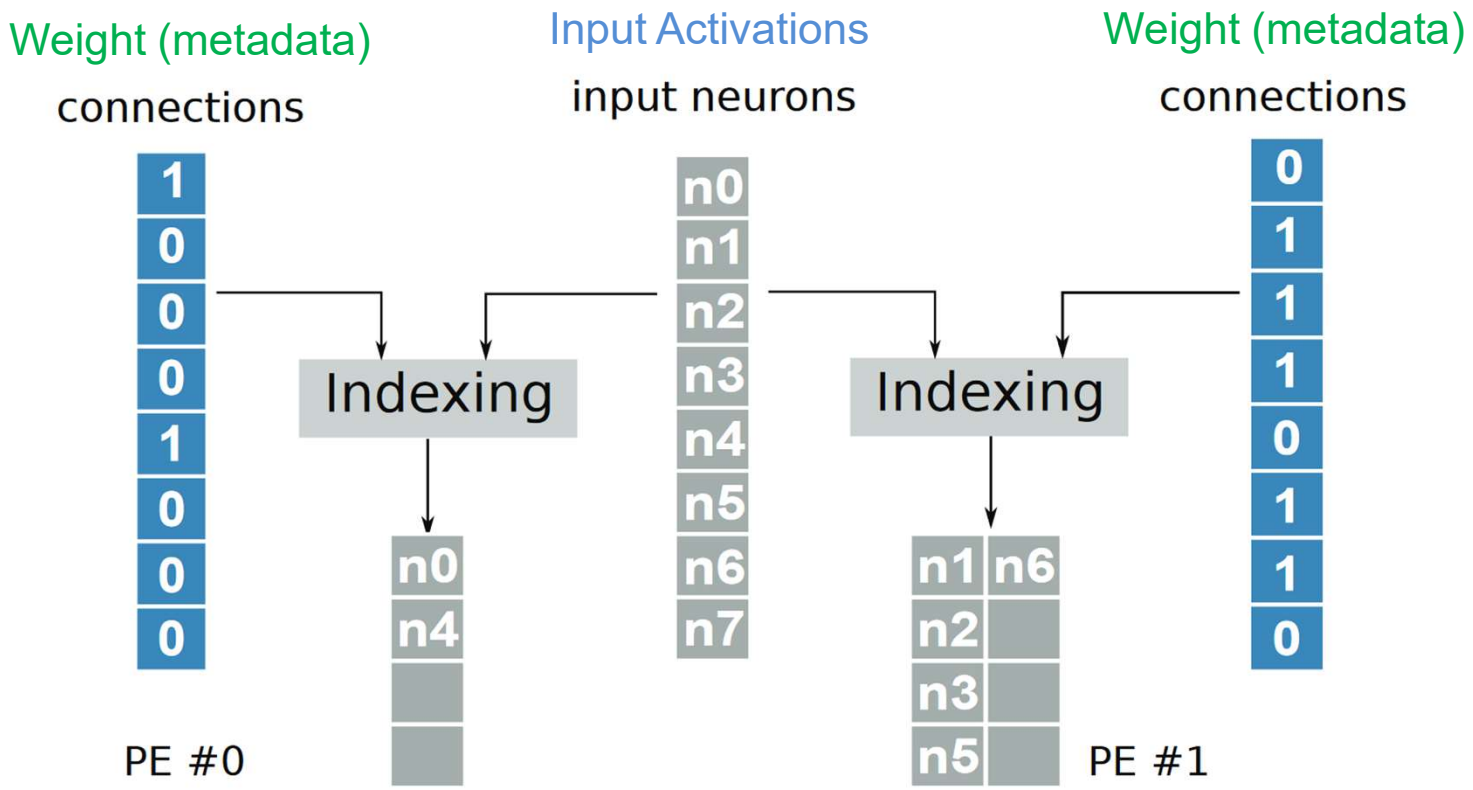
Work on four outputs at once

Calculate coordinates

Accumulate multiple outputs each spatially

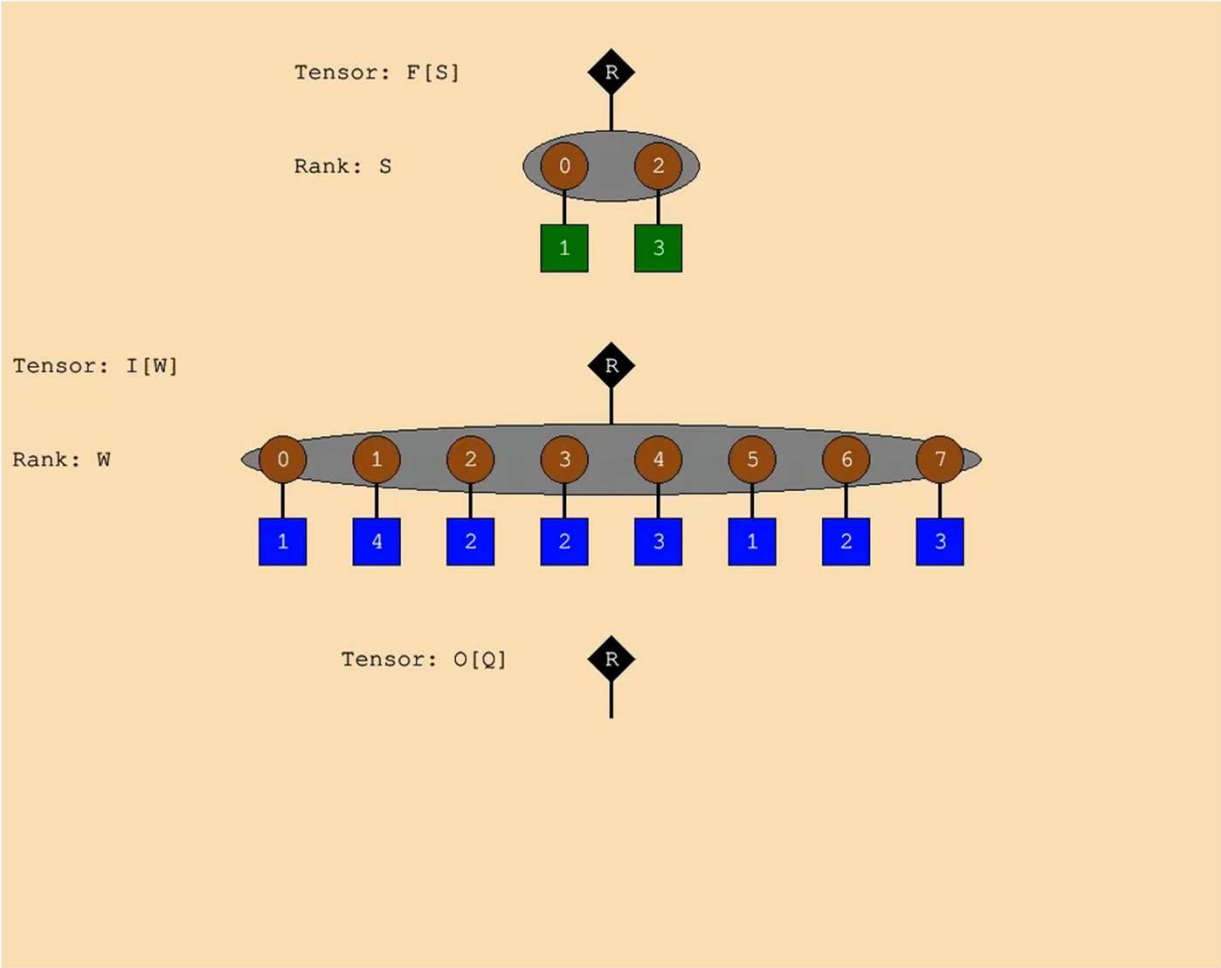
Look up input activation

Cambricon-X – Activation Access



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

Parallel Weight Stationary - Sparse Weights



CONV: Exploiting Sparse Inputs

Weight Stationary - Sparse Inputs

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

```
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

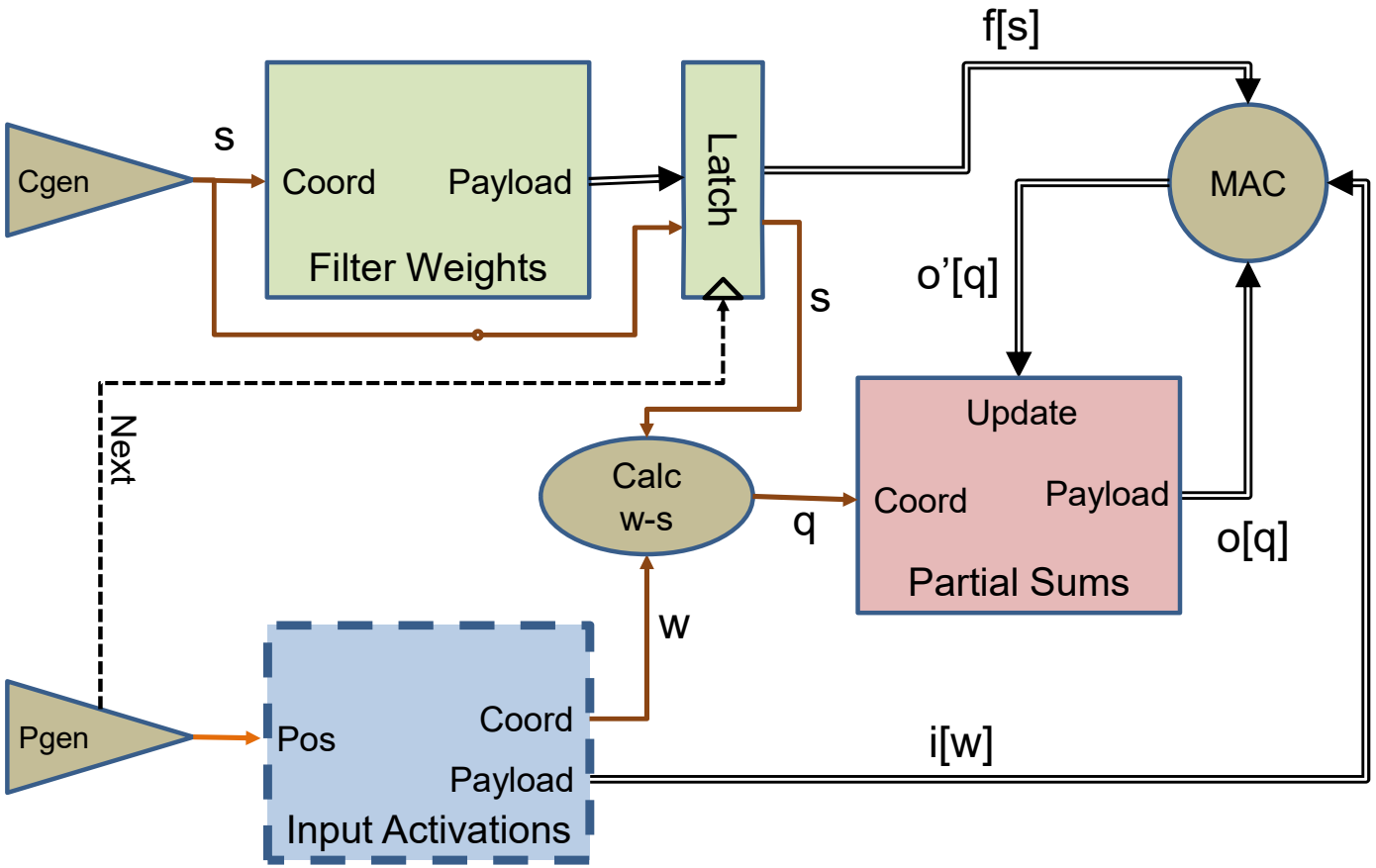
Projection of w and s

Populate

Reduction

Can look up this weight once since it is stationary.

Weight Stationary - Sparse Inputs



Output Stationary - Sparse Inputs

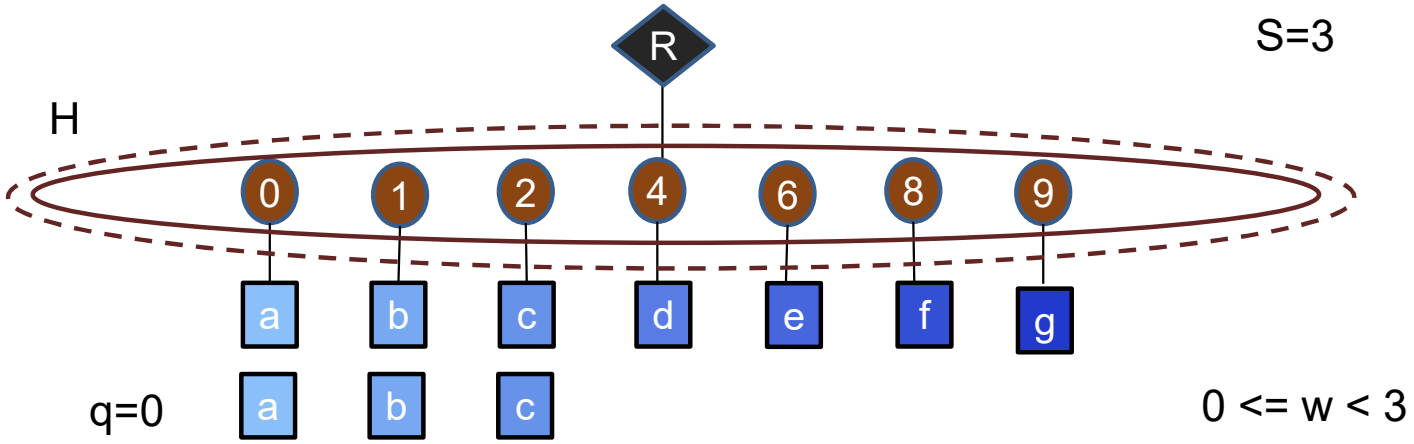
$$O_q = I_{q+s} \times F_s \quad \Rightarrow \quad O_q = I_w \times F_{w-q}$$

```
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]
```

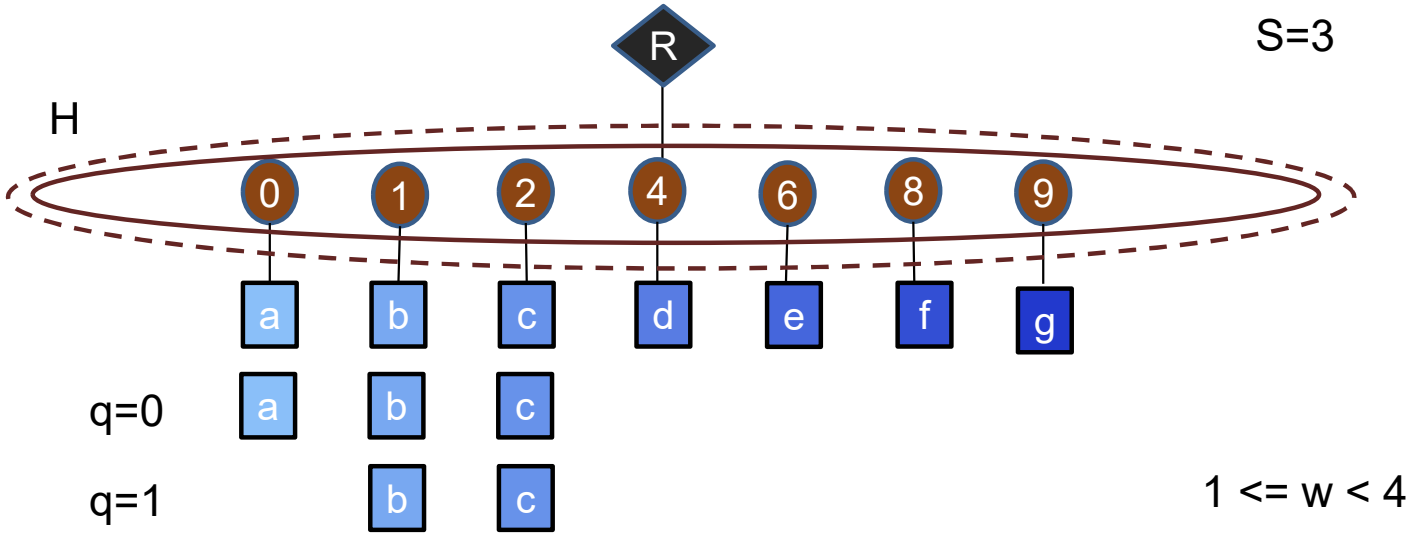
Need to look up a filter weight for each input

Need to restrict input coordinates to the active outputs

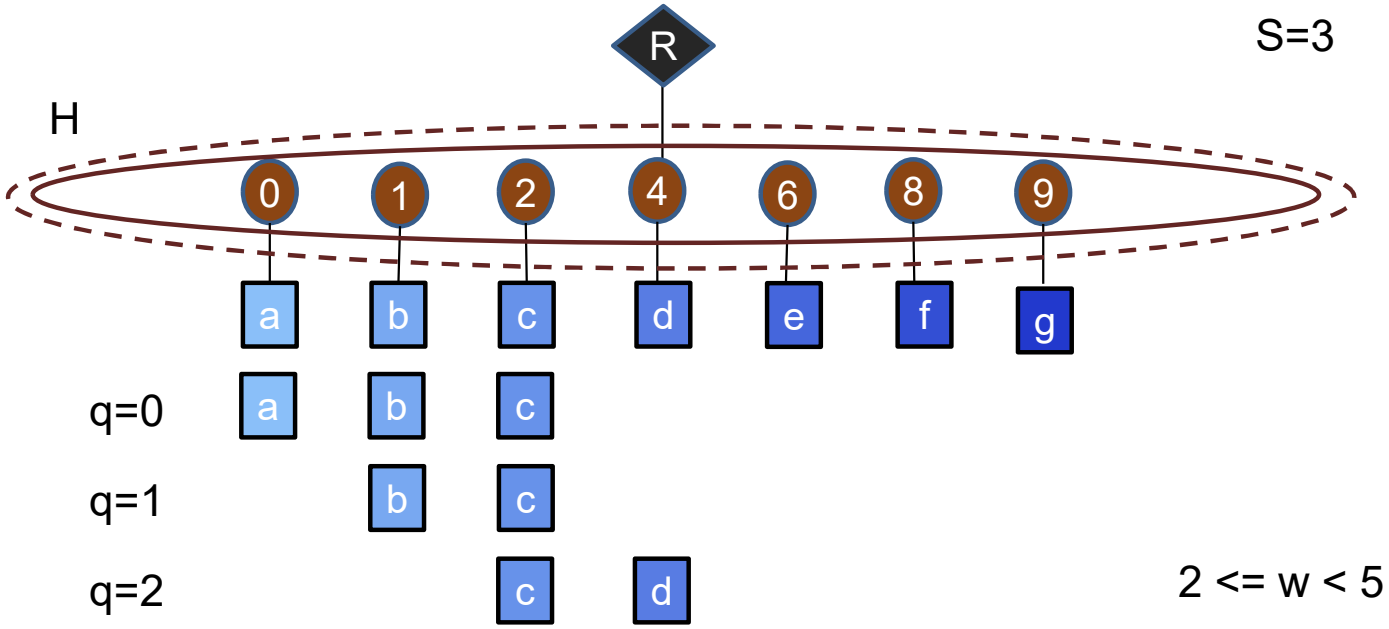
Sparse Sliding Window



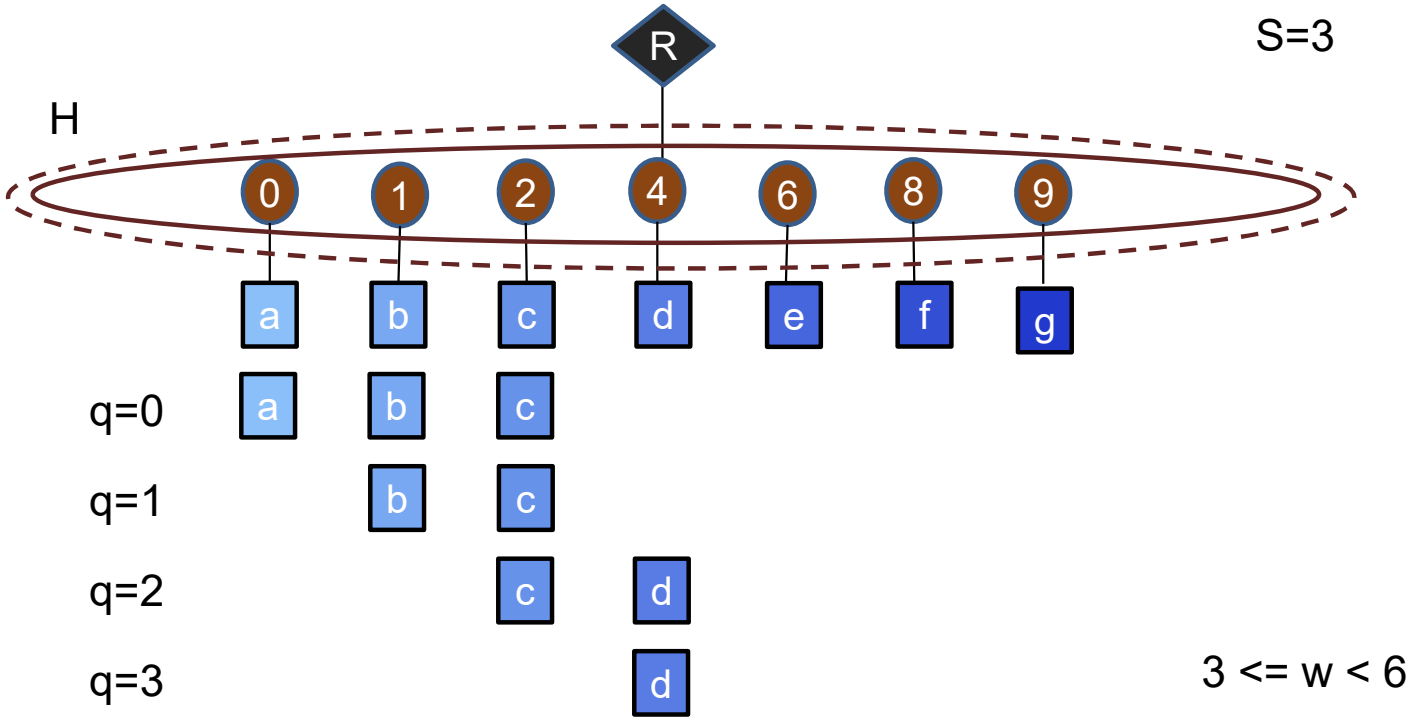
Sparse Sliding Window



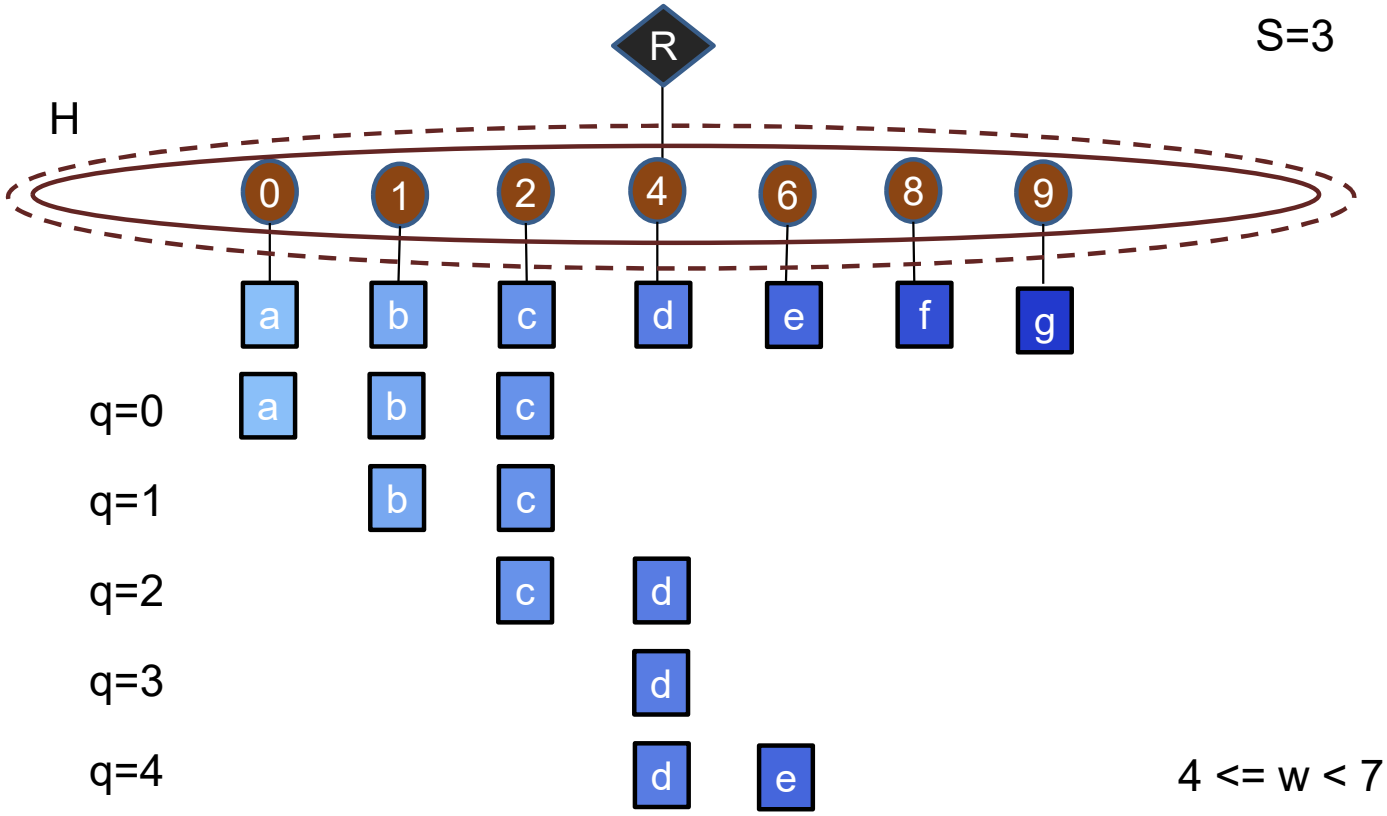
Sparse Sliding Window



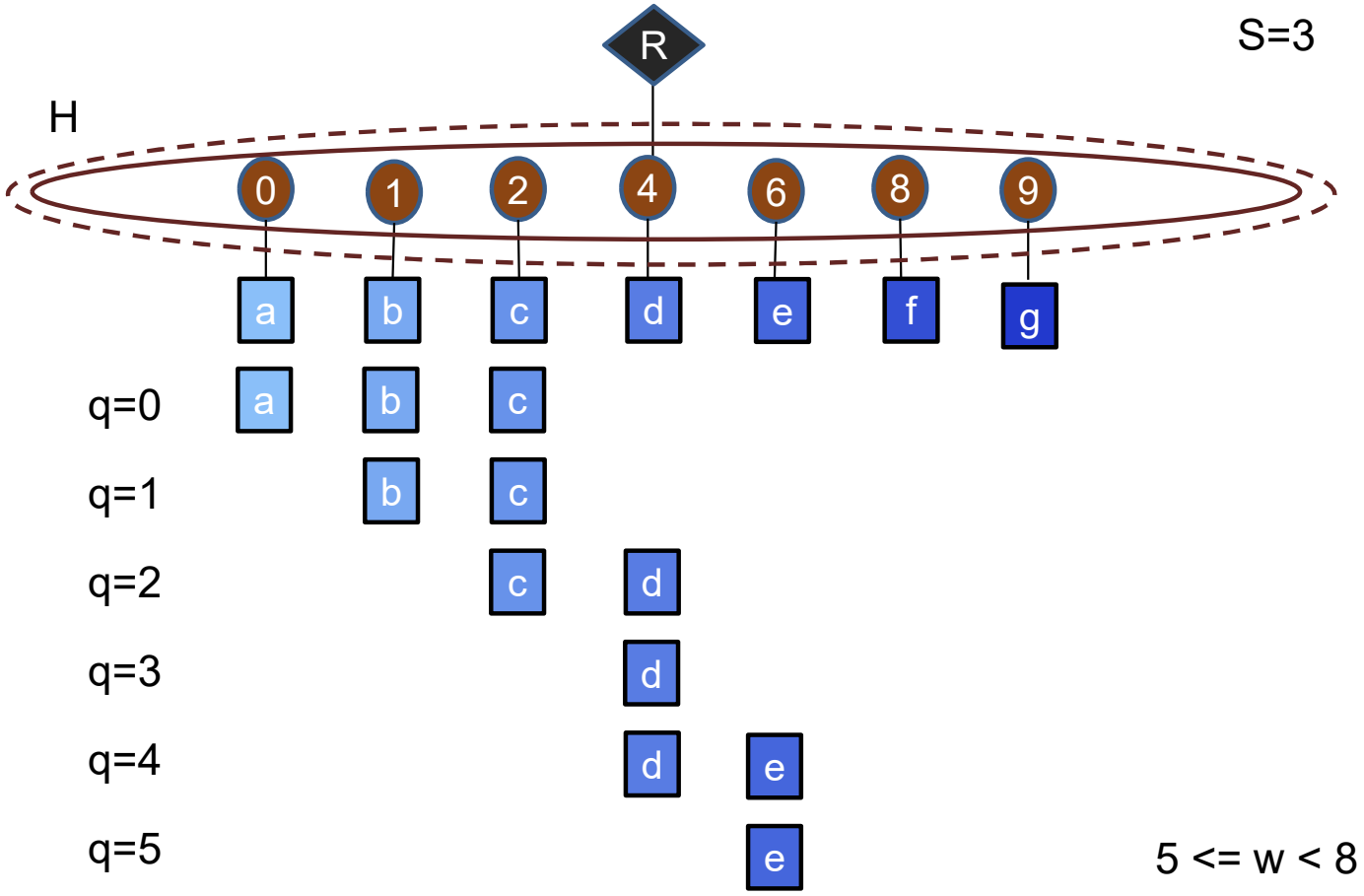
Sparse Sliding Window



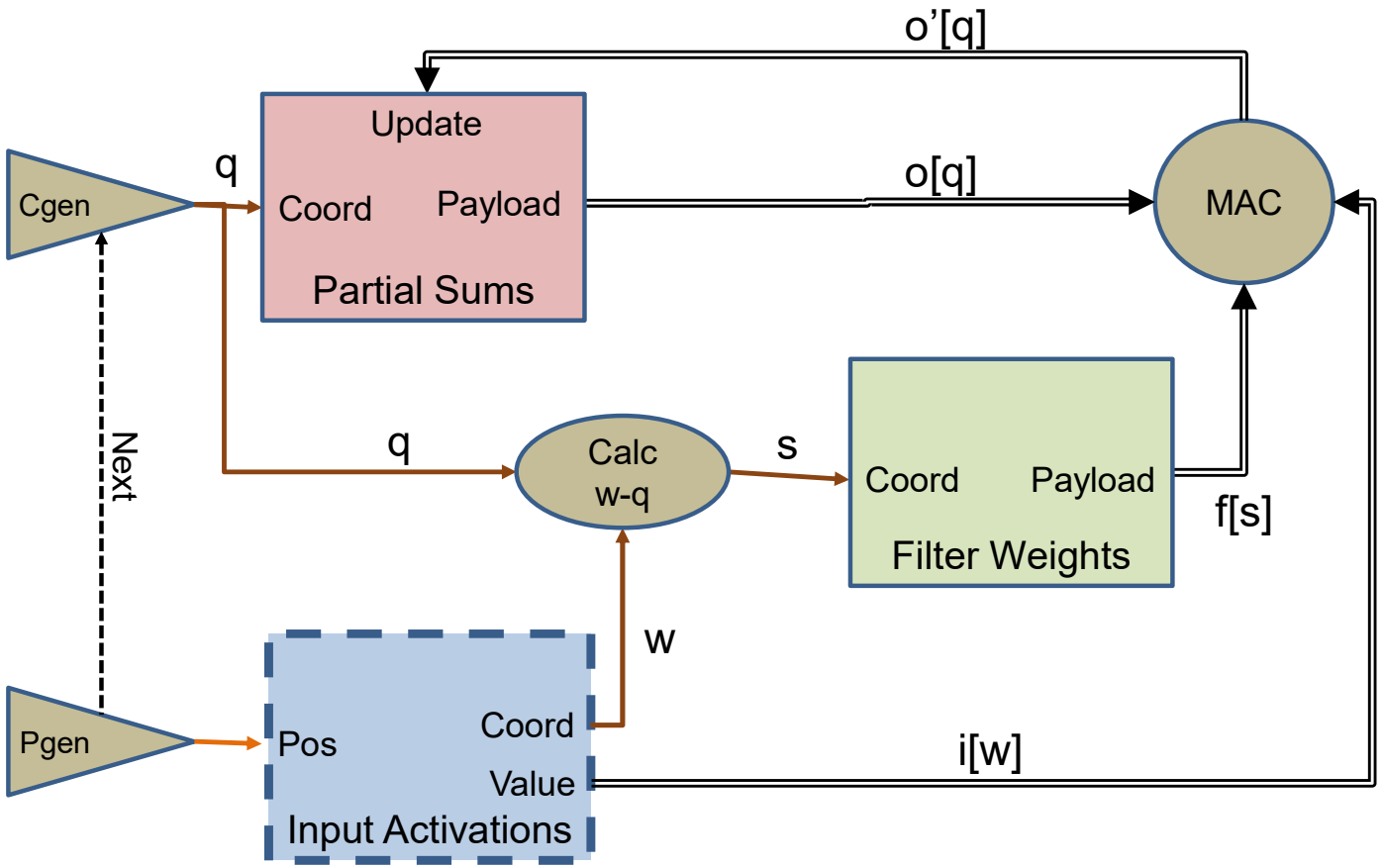
Sparse Sliding Window



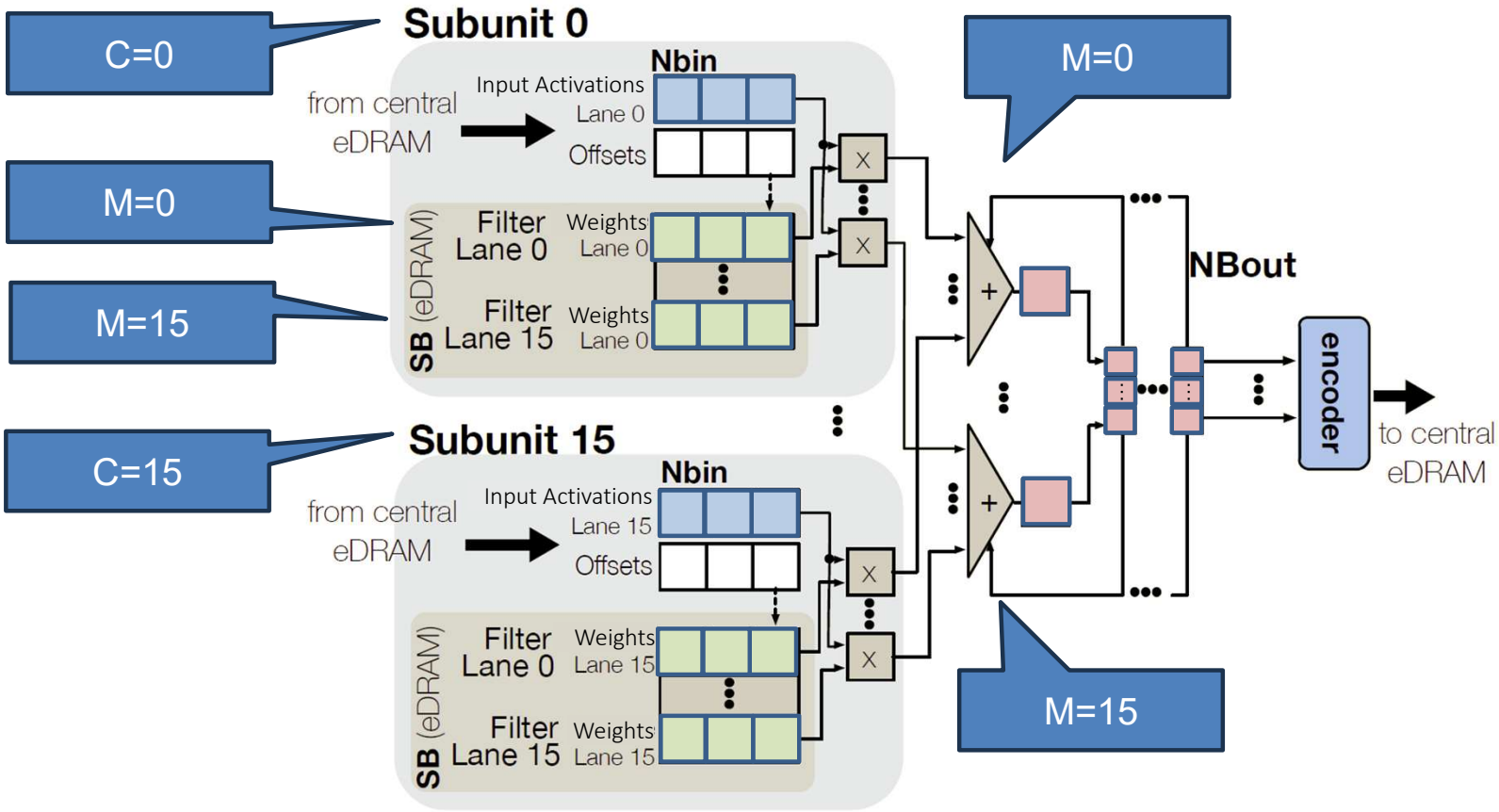
Sparse Sliding Window



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

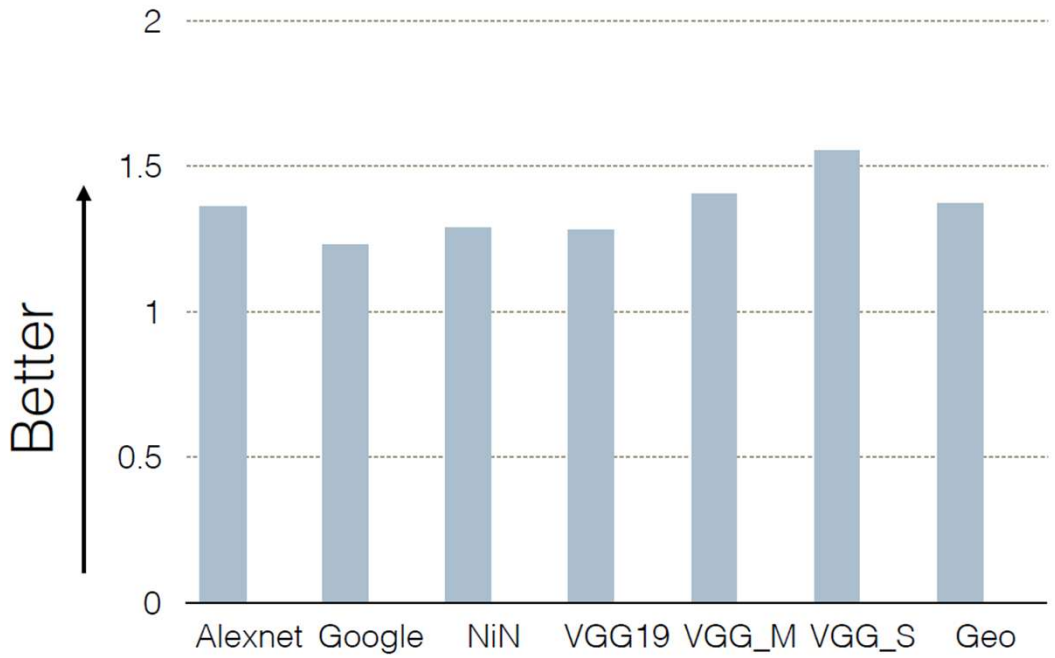
How do we make the getPayload() cheap?

Use uncompressed

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

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?

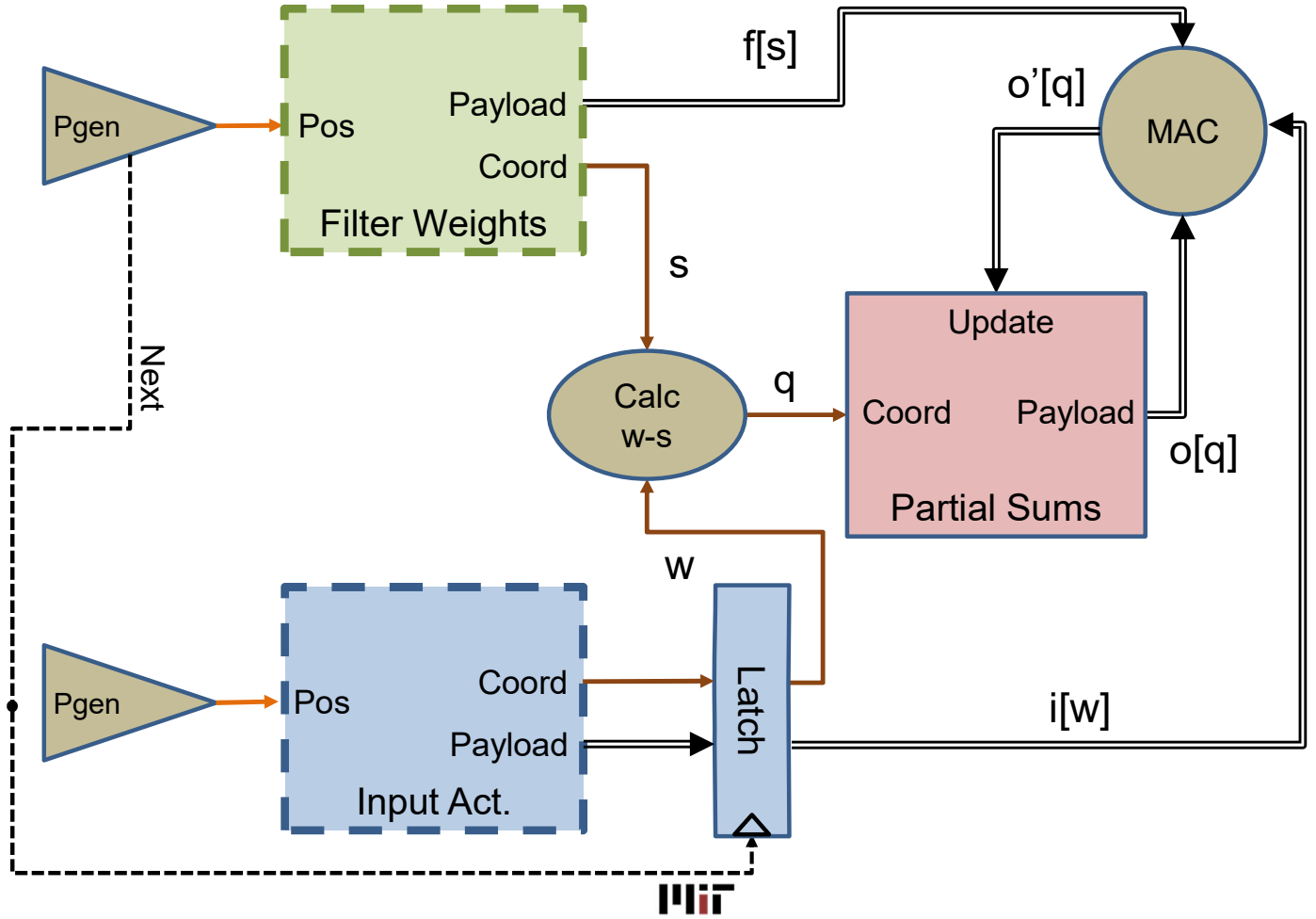
Input stationary

What sparsity can it exploit?

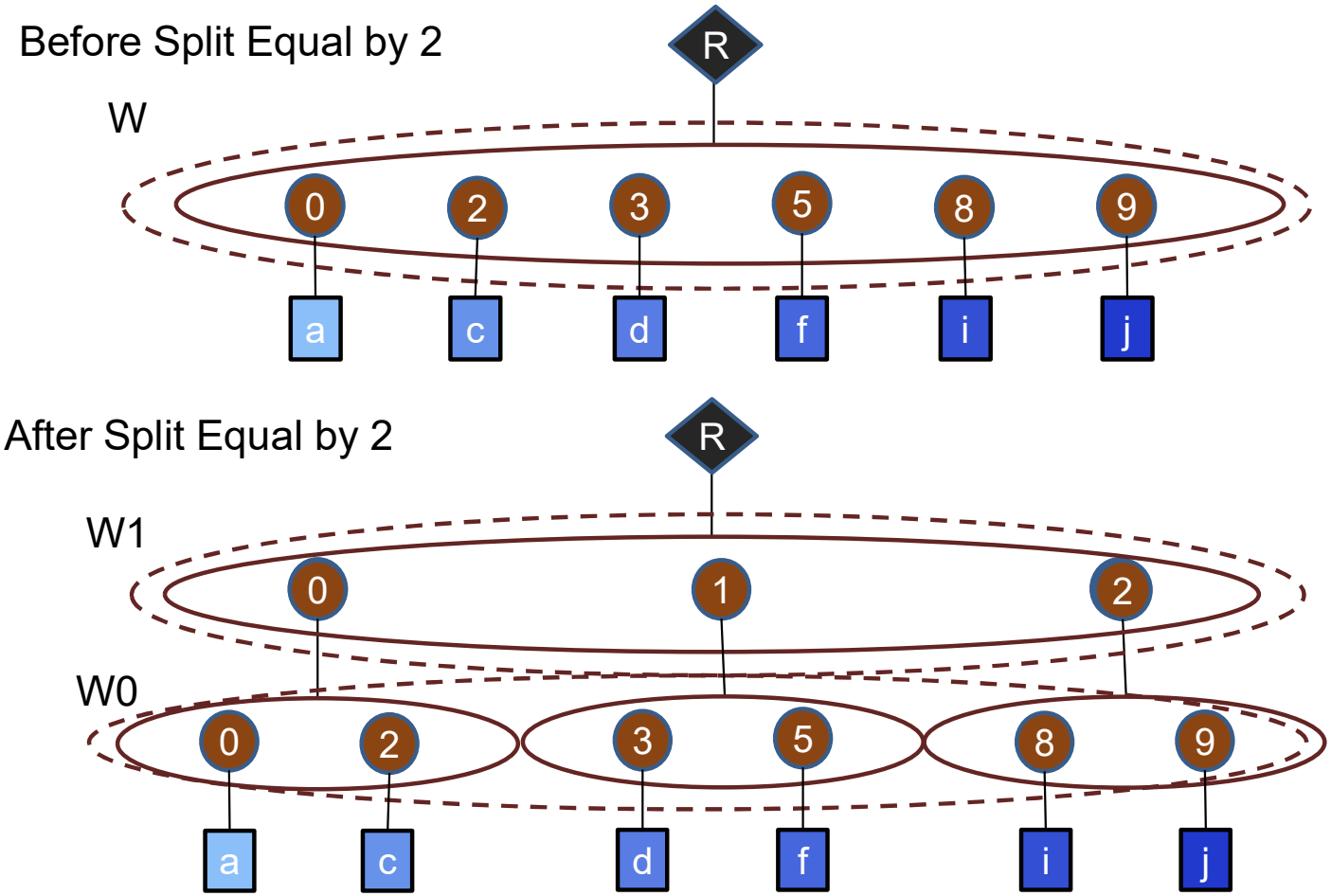
Inputs and Weights

CONV: Exploiting Sparse Inputs & Sparse Weights

Input Stationary - Sparse Weights & Inputs



Fiber Splitting Equally in Position Space



Input Stationary - Sparse Weights & Inputs

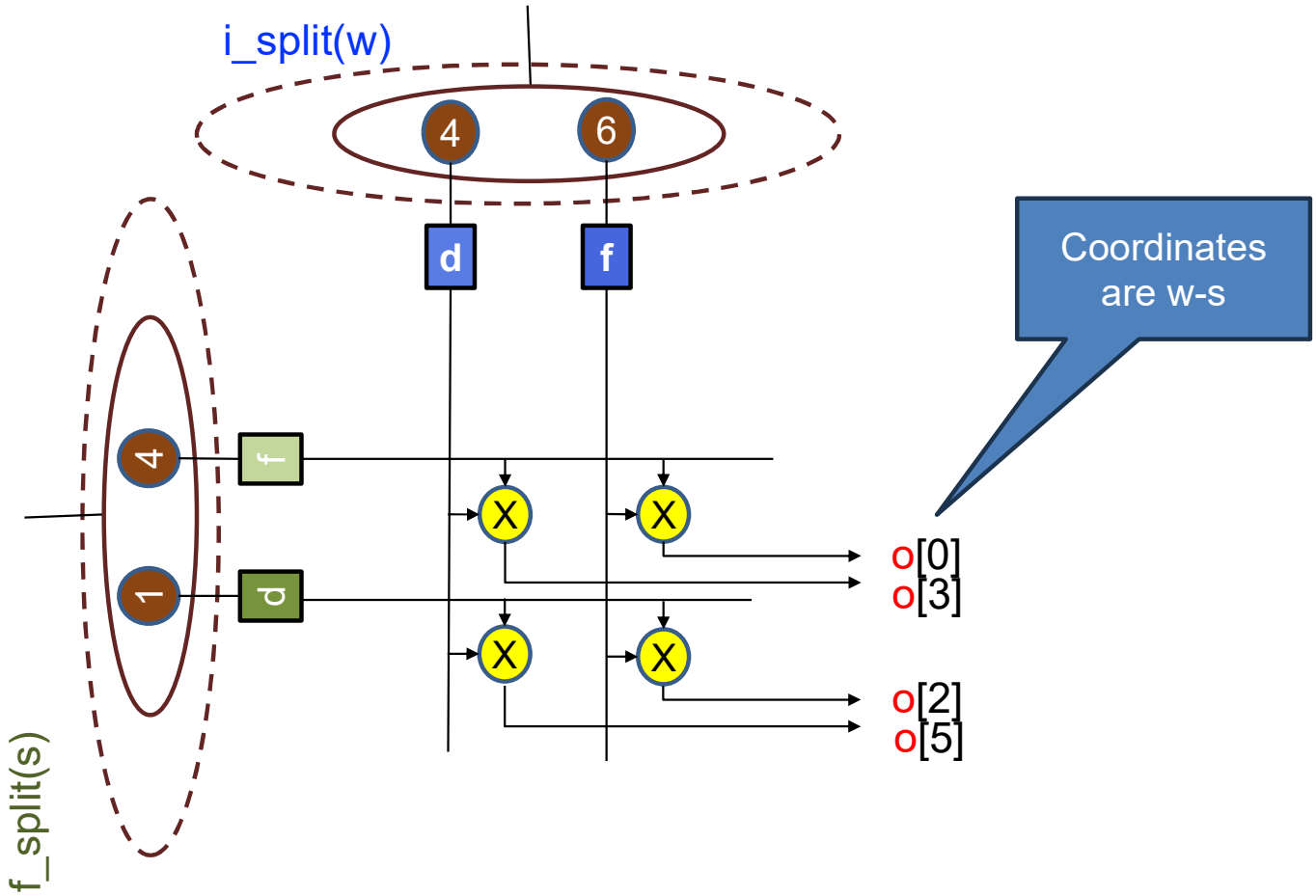
```
i = Tensor(W)          # Input activations
f = Tensor(S)          # Filter weights
o = Array(Q)           # Output activations

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

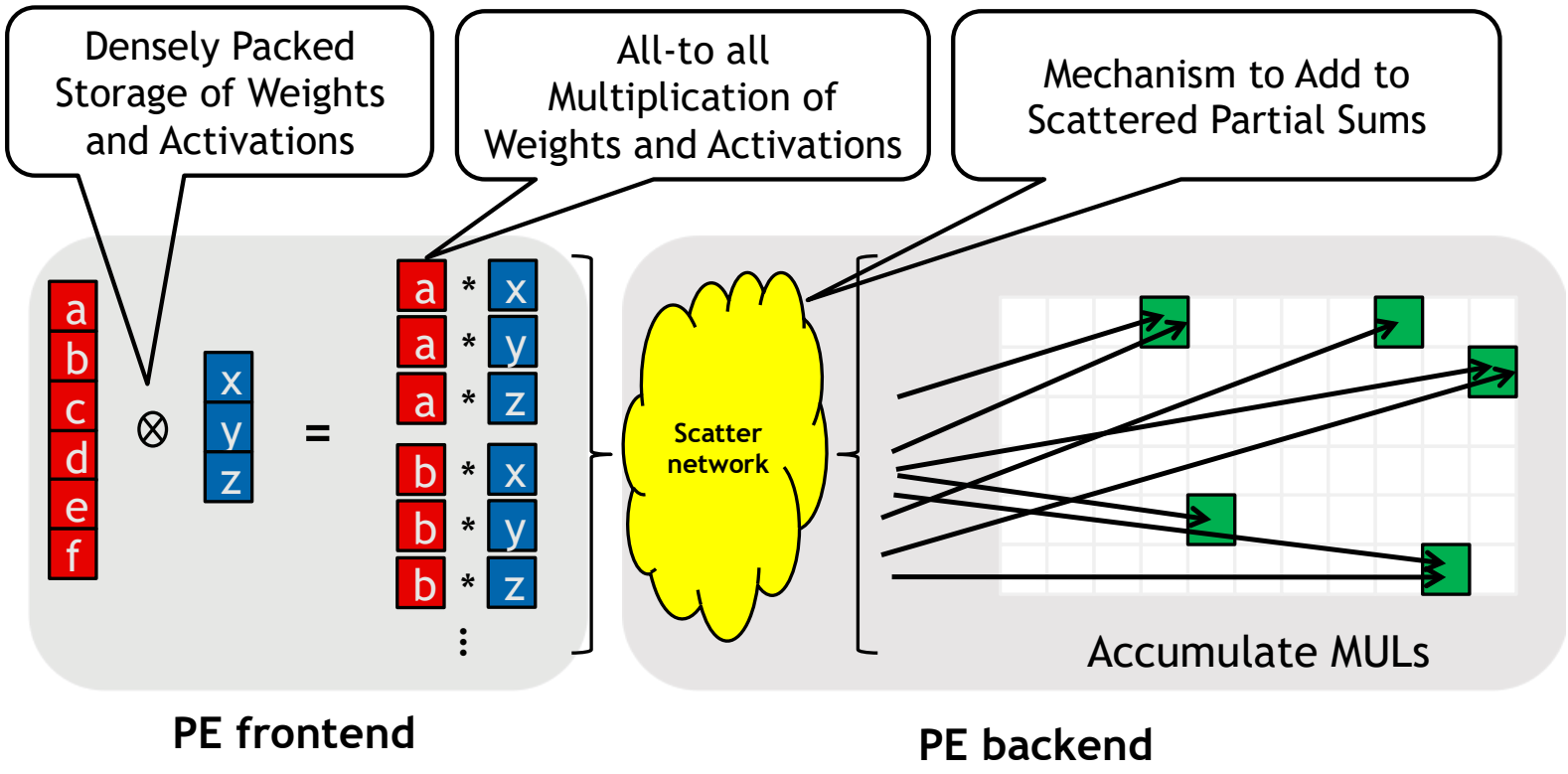
Is there a nice pattern to the multiplier outputs? No

Cartesian Product Multiplication



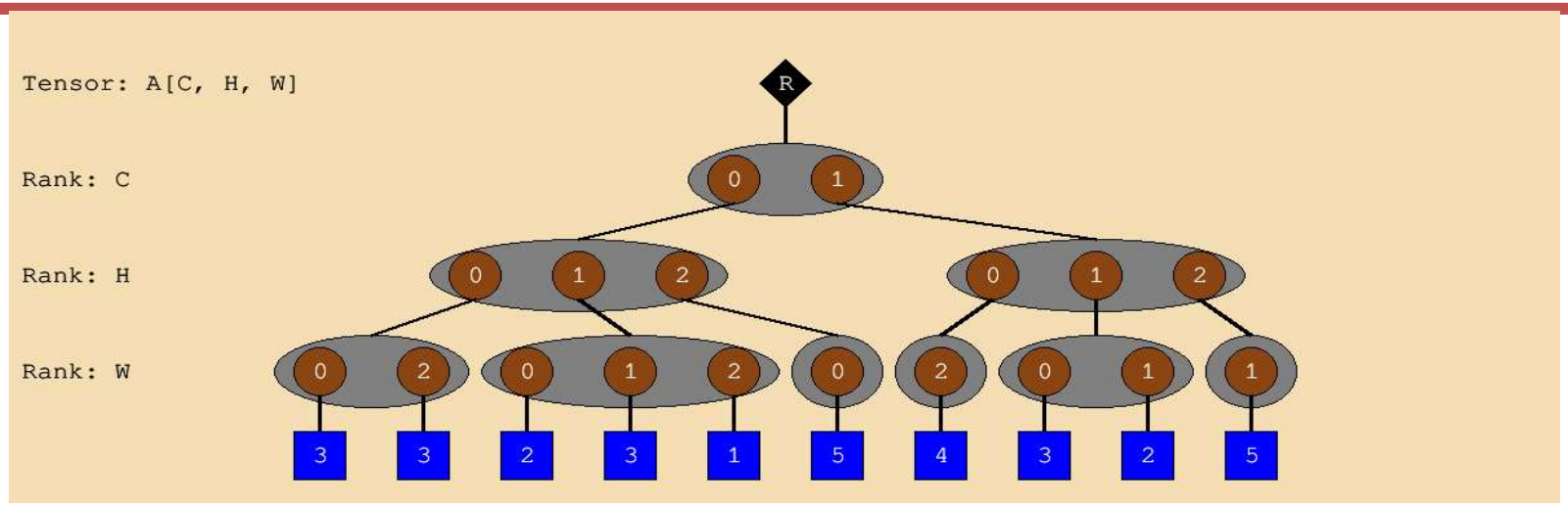
Sparse CNN (SCNN)

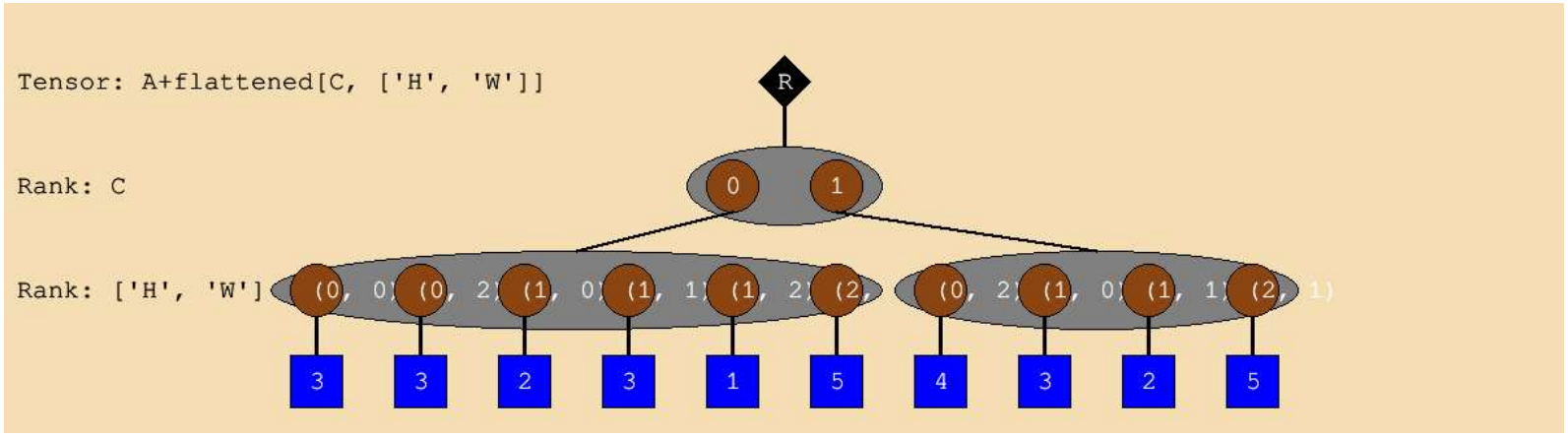
Supports Convolutional Layers



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

Flattening


Tensor: A[C, H, W]
Rank: C
Rank: H
Rank: W
R
0 1
0 1 2
0 2
0 1 2
0
0 1 2
2
0 1
1
3 3 2 3 1 5 4 3 2 5


Tensor: A+flattened[C, ['H', 'W']]
Rank: C
Rank: ['H', 'W']
R
0 1
(0, 0) (0, 2) (1, 0) (1, 1) (1, 2) (2, 0)
(0, 2) (1, 0) (1, 1) (2, 1)
3 3 2 3 1 5 4 3 2 5

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

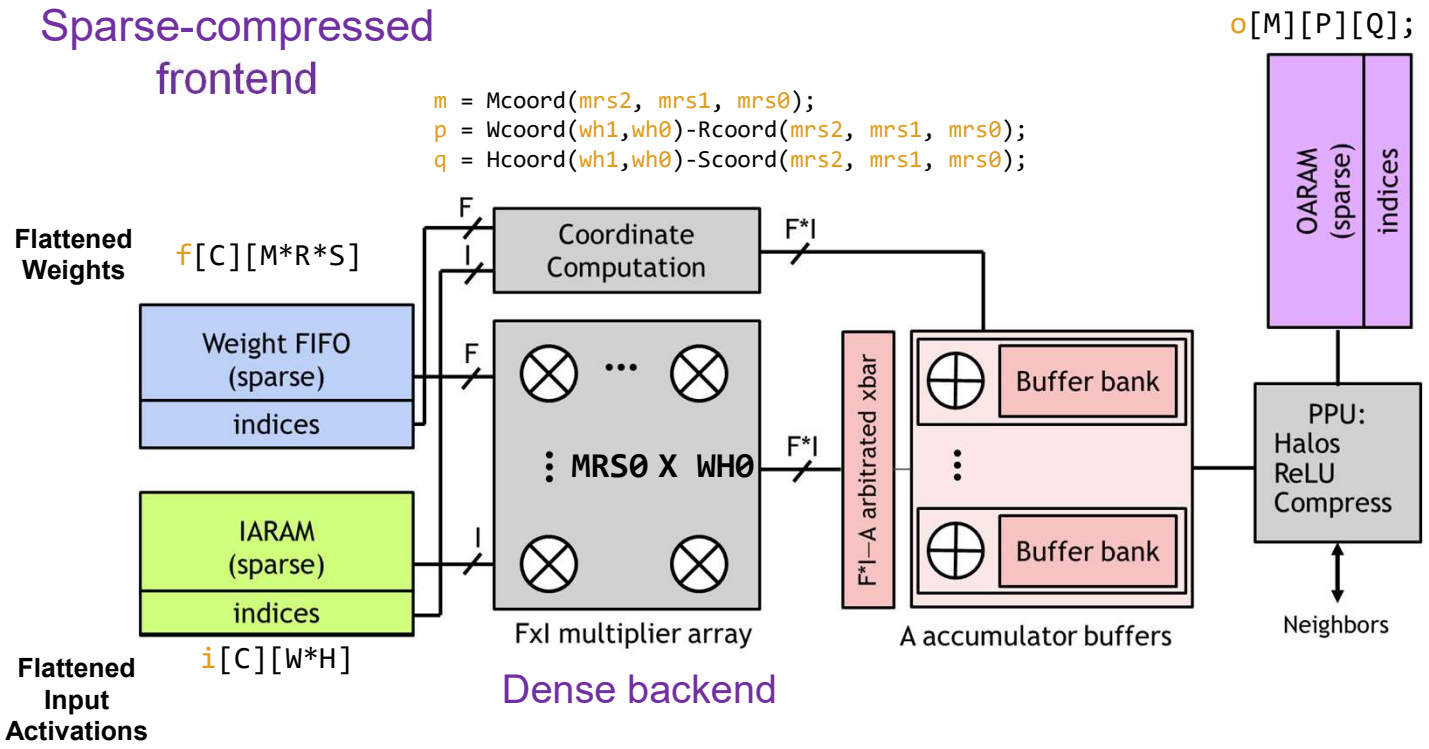
$$O_{m,h-r,w-s} = I_{hw} \times F_{mrs}$$

SCNN Tile – one channel

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

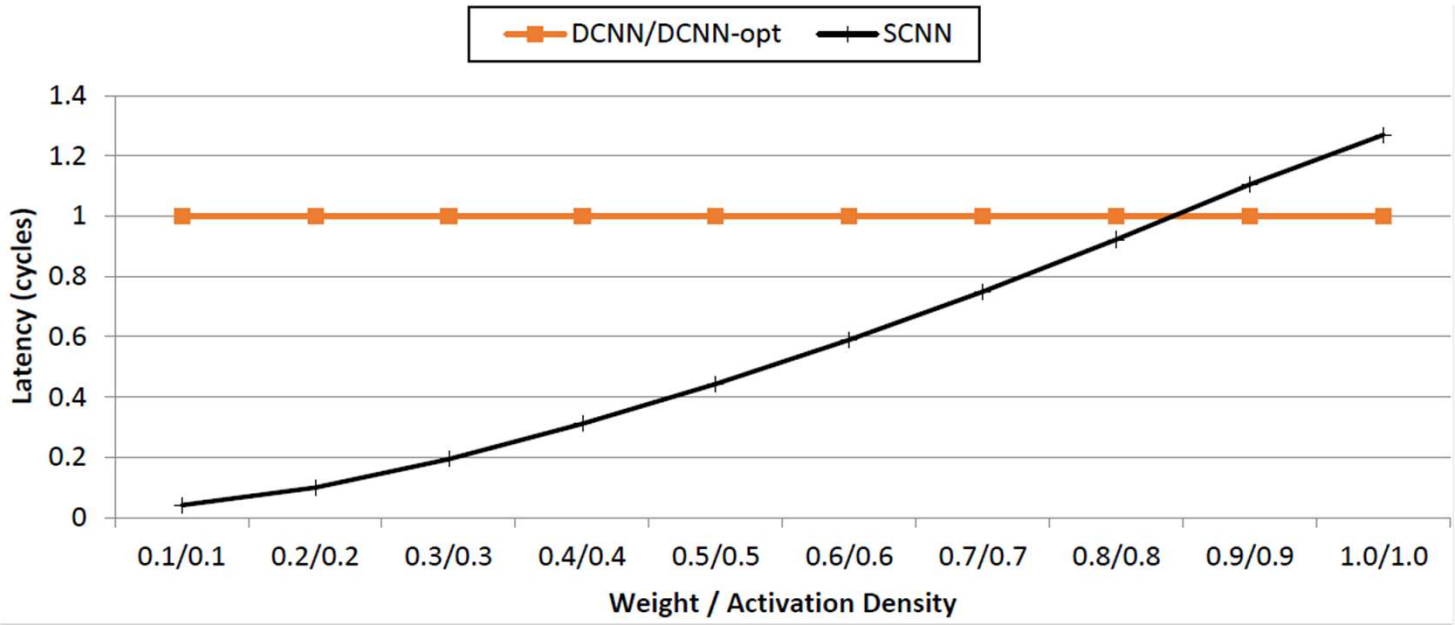
```
i = Tensor(HW)       # Input activations
f = Tensor(MRS)      # Filter weights
o = Array(M,P,Q)     # Output activations

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



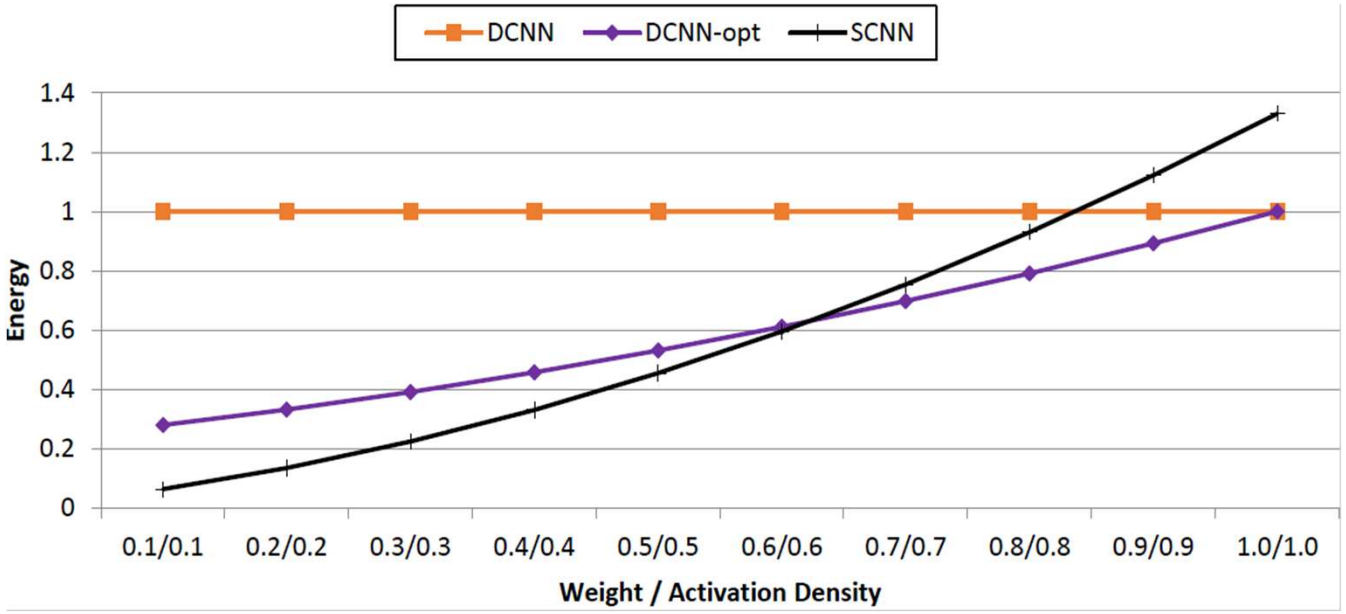
[Parashar et al., SCNN, ISCA 2017]

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

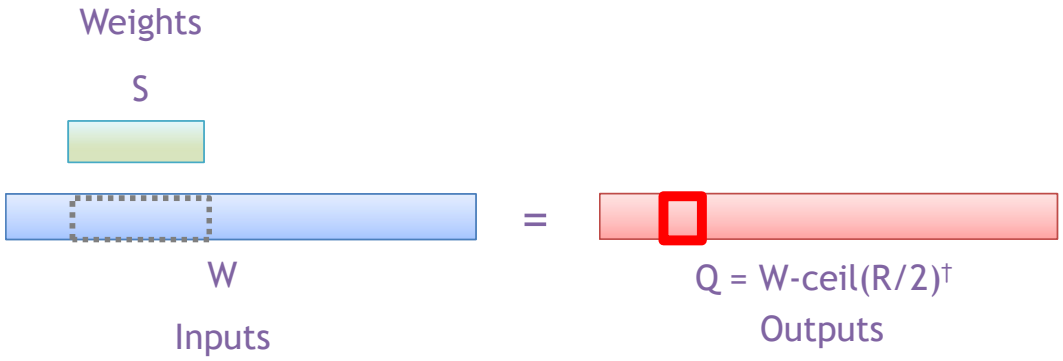
Output Stationary - Sparse Weights & Inputs

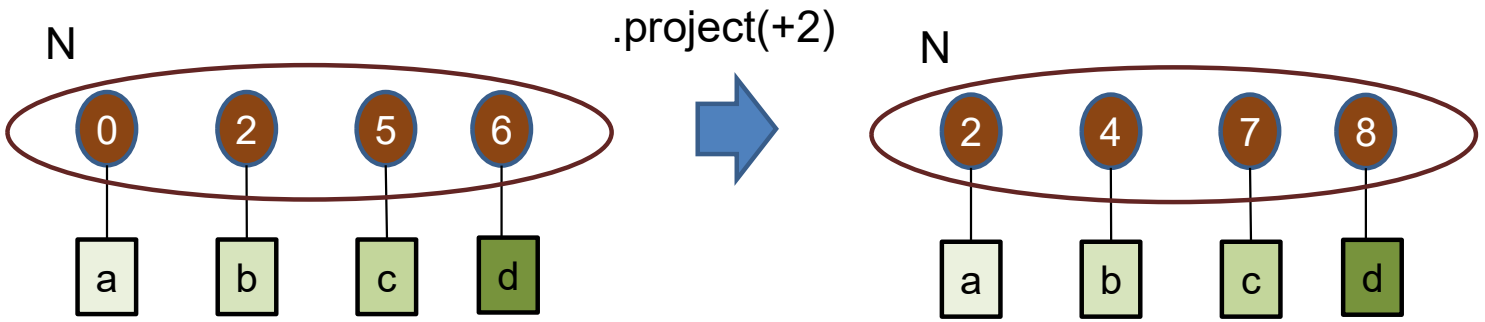
$$O_q = I_{q+s} \times F_s$$

```
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

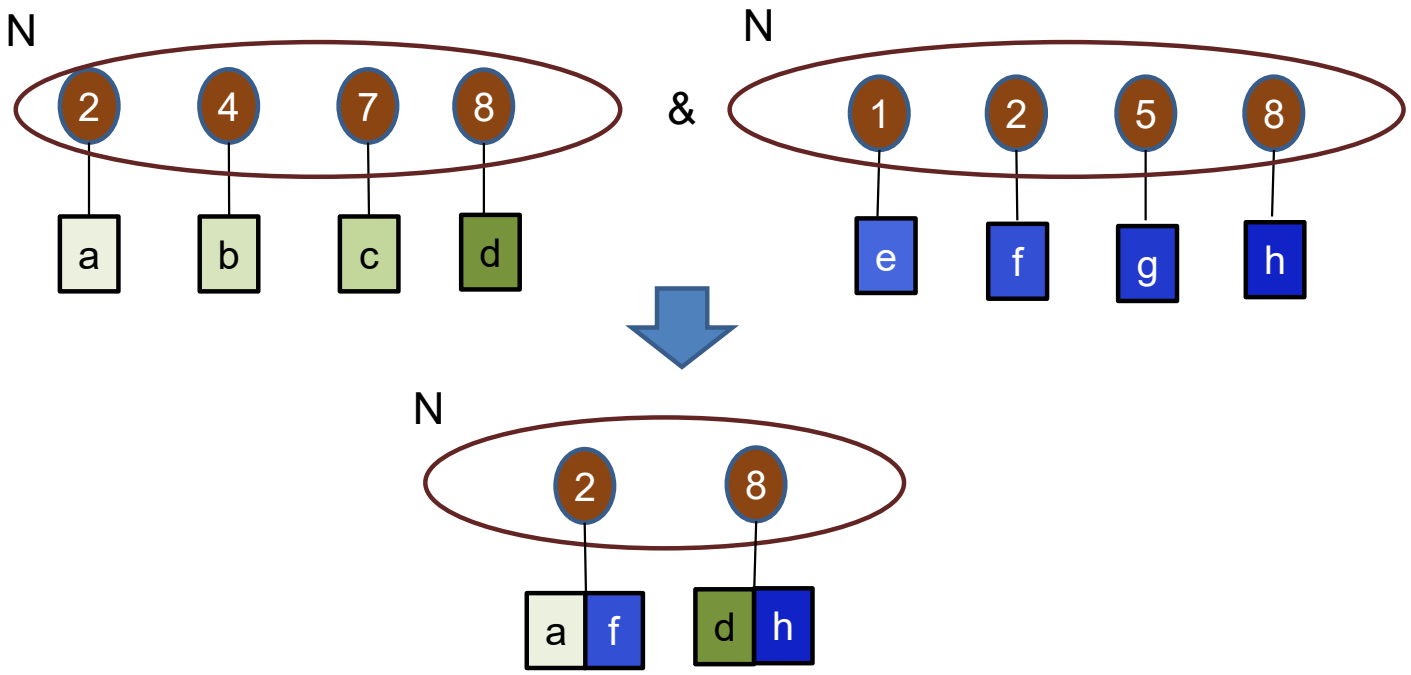




Does projection require complex hardware?

Representation dependent

Fiber Intersection



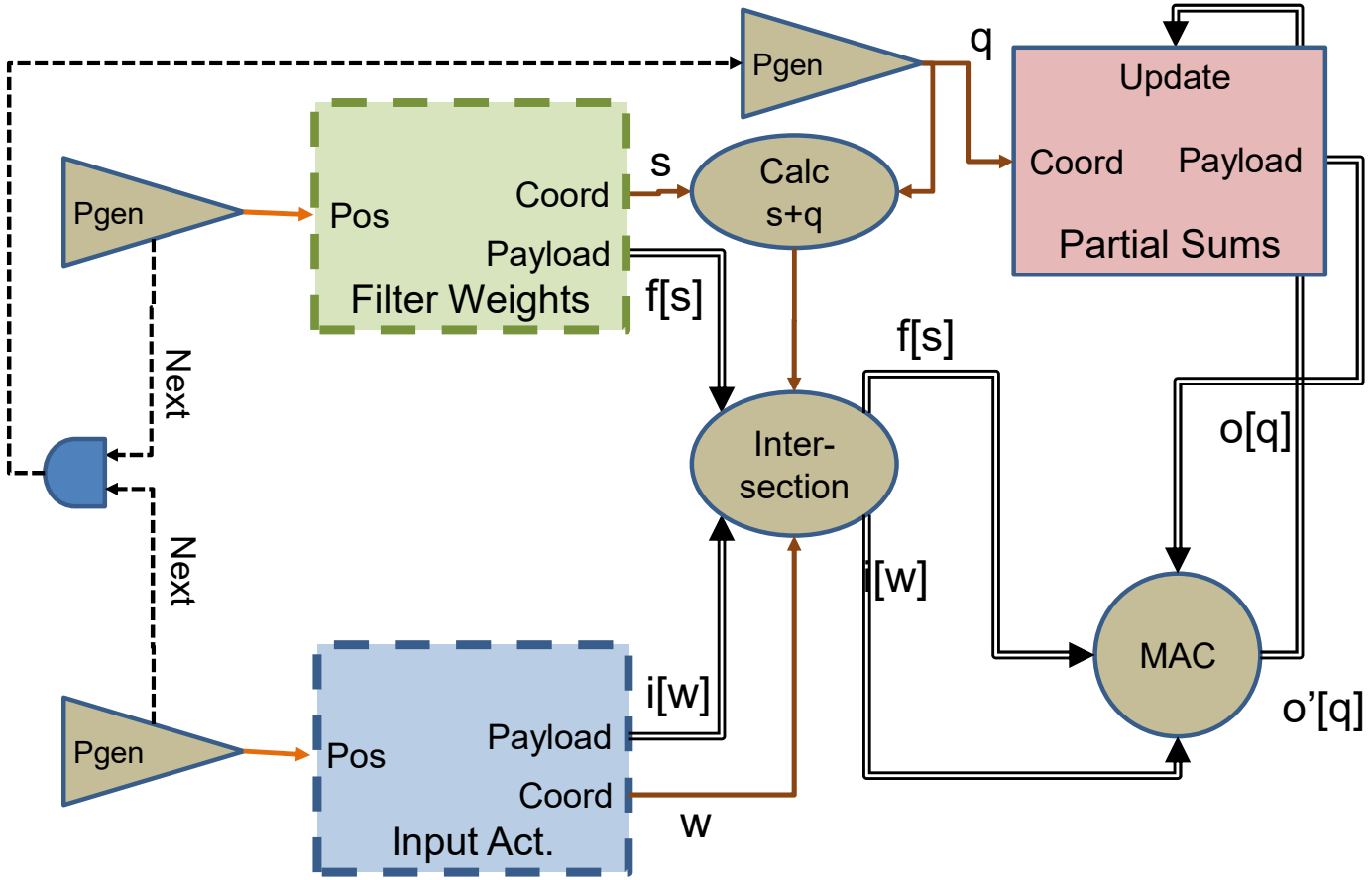
Does intersection require complex hardware?

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

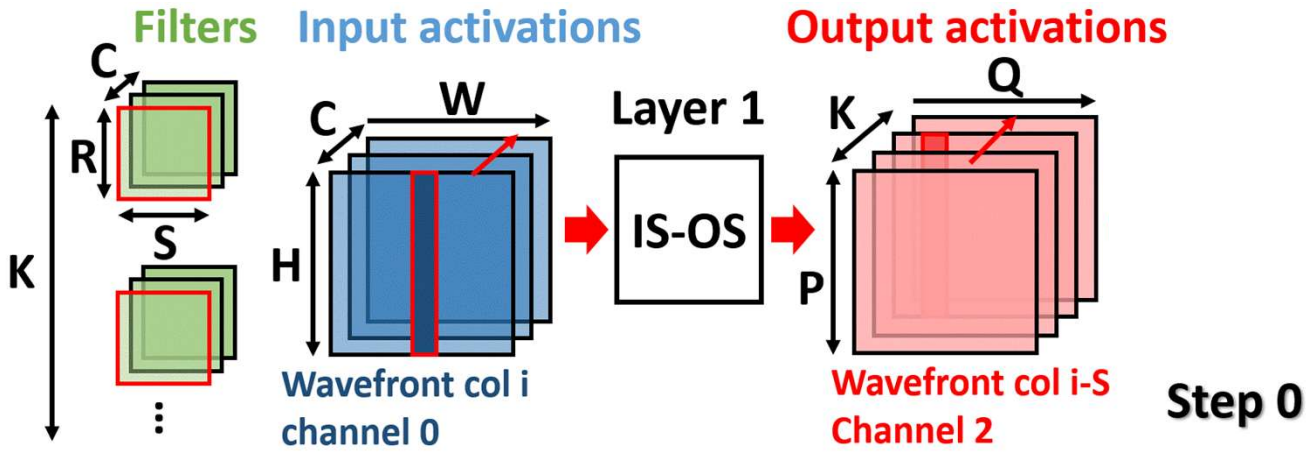
```
parallel-for h, (t_r, i_w) in t_h << i_h:
  for w, i_val in i_w:
    for c, (i_w, f_r) in i_c & f_c:
      for r, (t_q, f_s) in t_r << f_r:
        parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
          t_ref += i_val * f_val

                  ["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

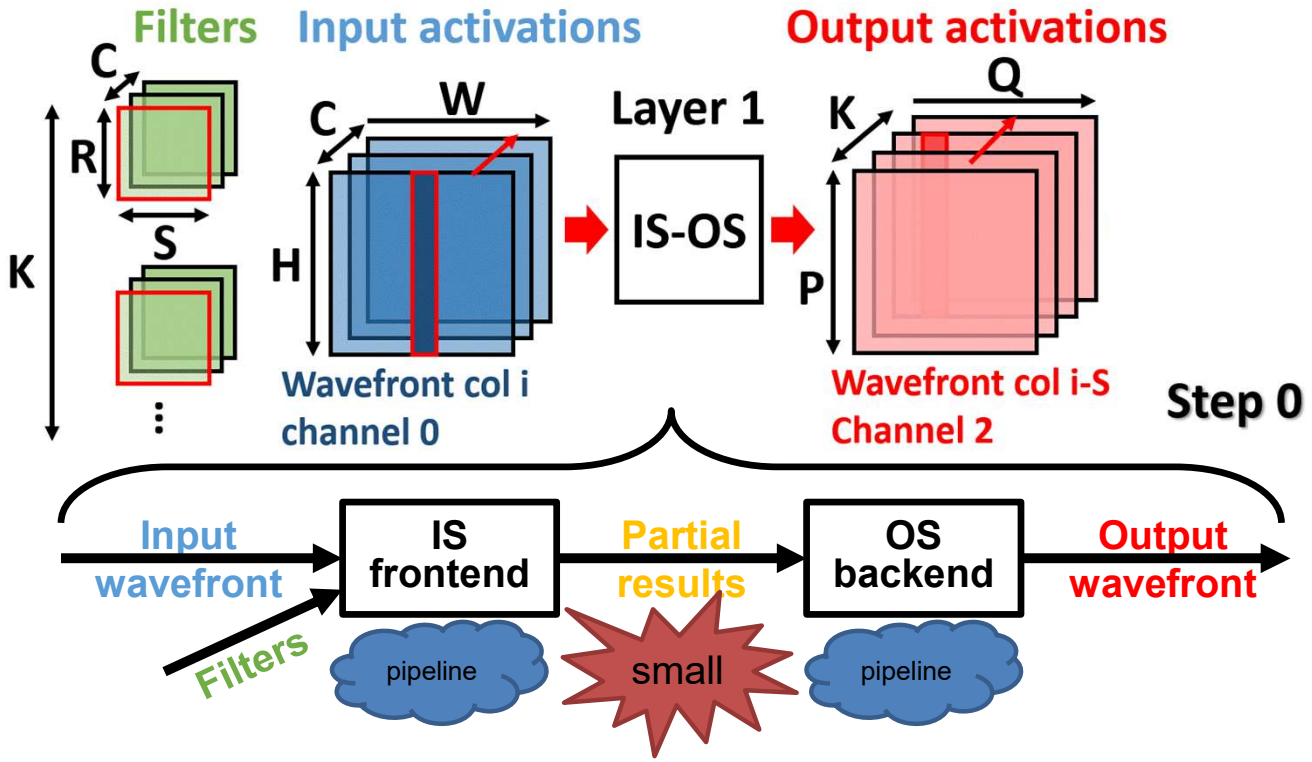
```
parallel-for h, (t_r, i_w) in t_h << i_h:
  for w, i_val in i_w:
    for c, (i_w, f_r) in i_c & f_c:
      for r, (t_q, f_s) in t_r << f_r:
        parallel-for s, (t_ref, f_val) in t_q.project(w-q) << f_s
          t_ref += i_val * f_val

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



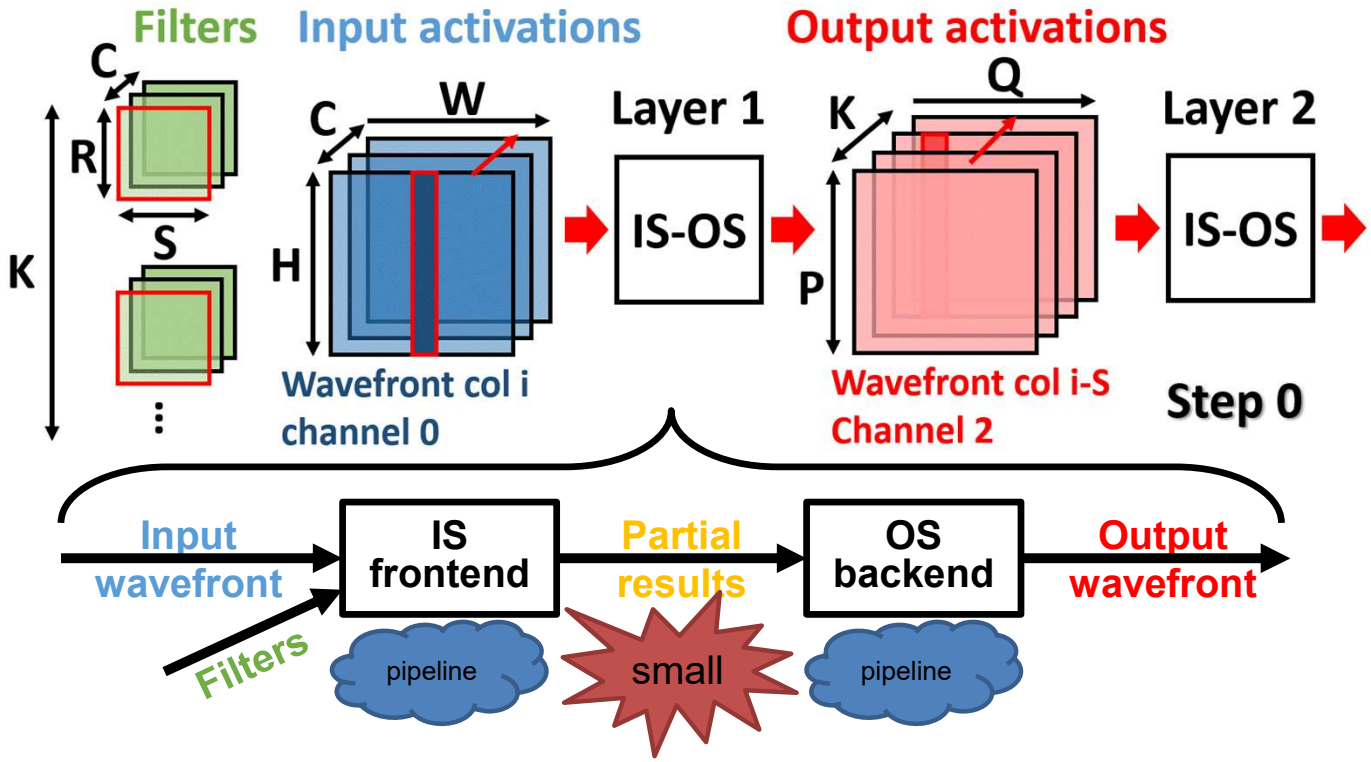
[Yang et al., ISOSceles, HPCA 2023]

IS-OS dataflow breakdown



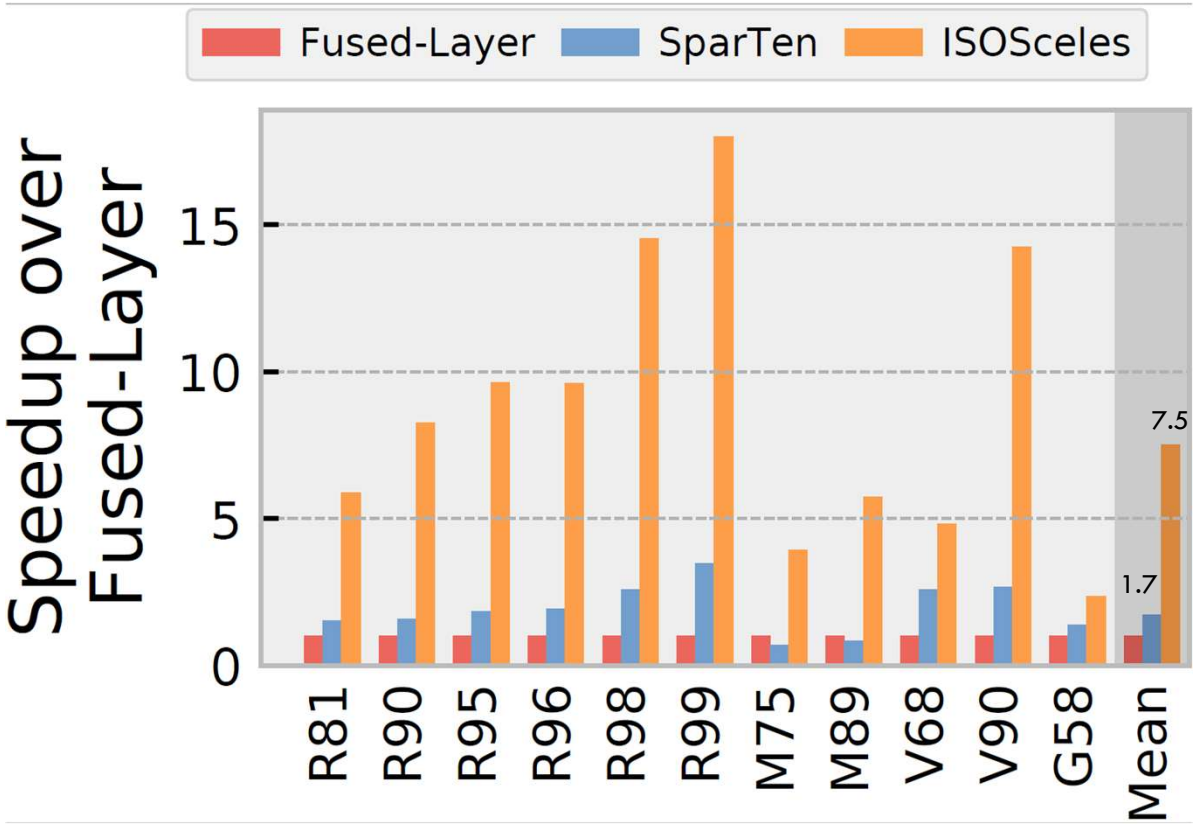
[Yang et al., ISOSceles, HPCA 2023]

IS-OS dataflow breakdown



[Yang et al., ISOSceles, HPCA 2023]

ISOSceles Speedup



[Yang et al., ISOSceles, HPCA 2023]