# Sparse Tensor Accelerators: Abstraction and Modeling

Background Lecture Part 1

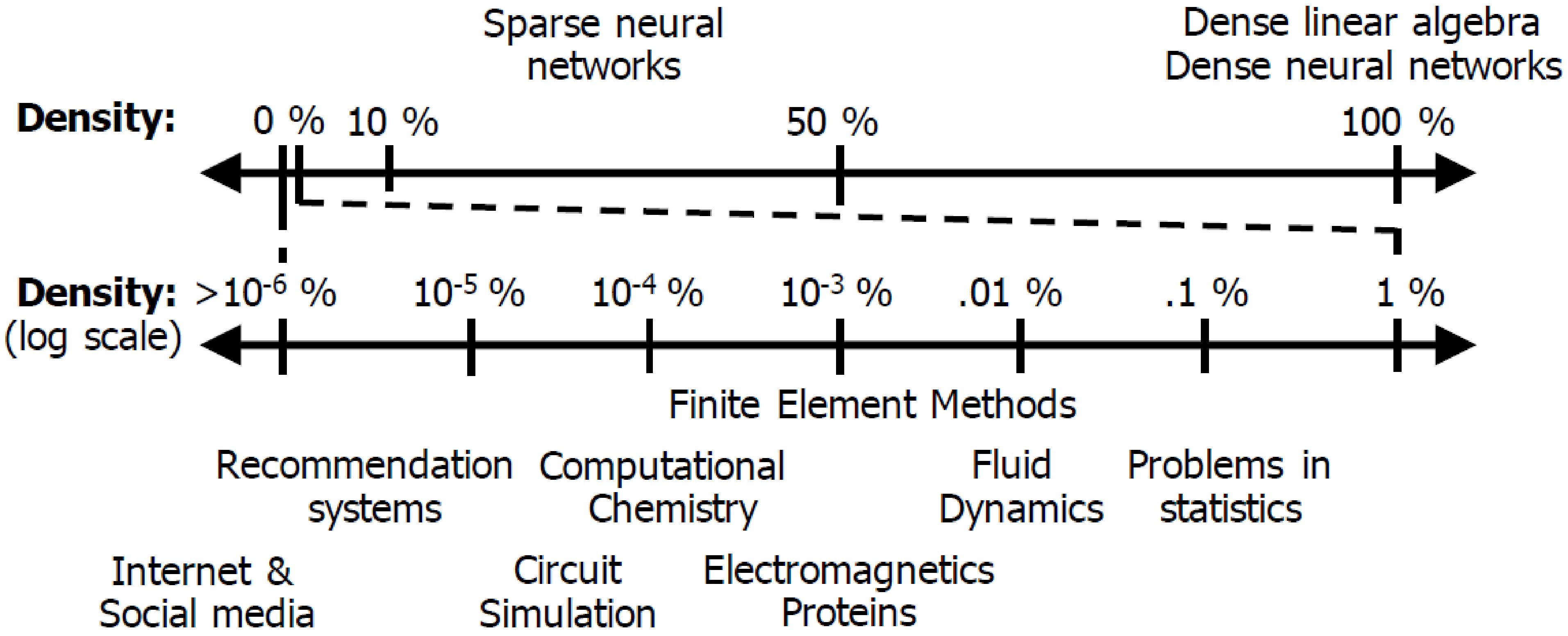
Joel Emer
Angshuman Parashar
Vivienne Sze
Po-An Tsai
Nellie Wu

ISCA Tutorial

June 2021

NVIDIA
Massachusetts Institute of Technology

# Many problems use Sparse Tensors




[Hegde, et.al., MICRO 2019]

# Exploiting Sparsity

Sparse data can be compressed

Can save space and energy by avoiding manipulation of zero values

$$\boldsymbol{anything} \times \mathbf{0} = \mathbf{0}$$

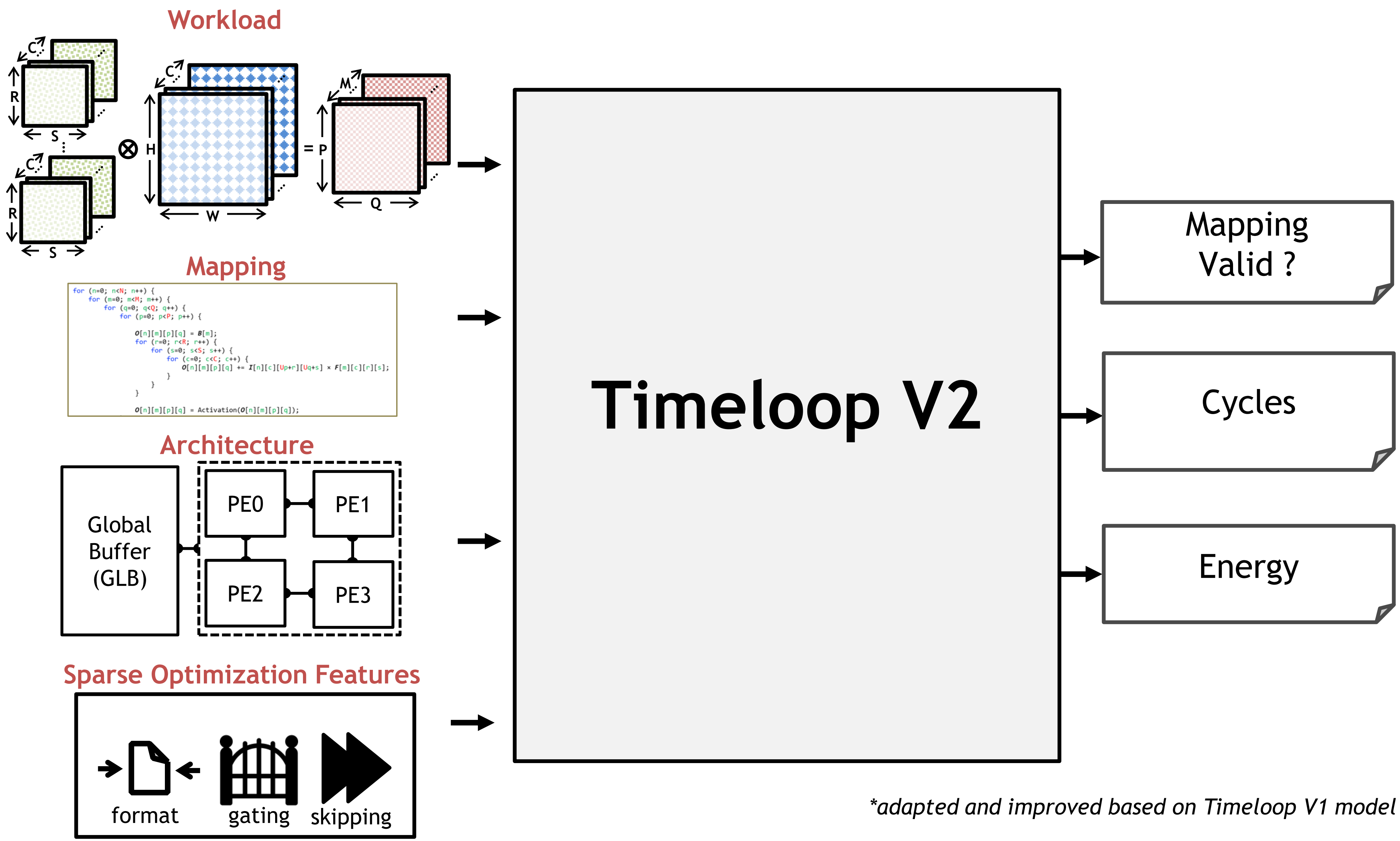
$$\boldsymbol{anything} + \mathbf{0} = \boldsymbol{anything}$$

Can save time and energy by avoiding fetching unnecessary operands and avoiding computations

# Outline

- Problem specification and motivation
- Specifying scheduling of computations on dense data
- Abstracting the representation of sparse tensors
- Specifying scheduling of computations on sparse data
- Present simple example architectures that exploit sparsity
- Architectural features for exploiting sparsity
- Workload specification for sparse computations
- Modeling of impact of sparse optimization features

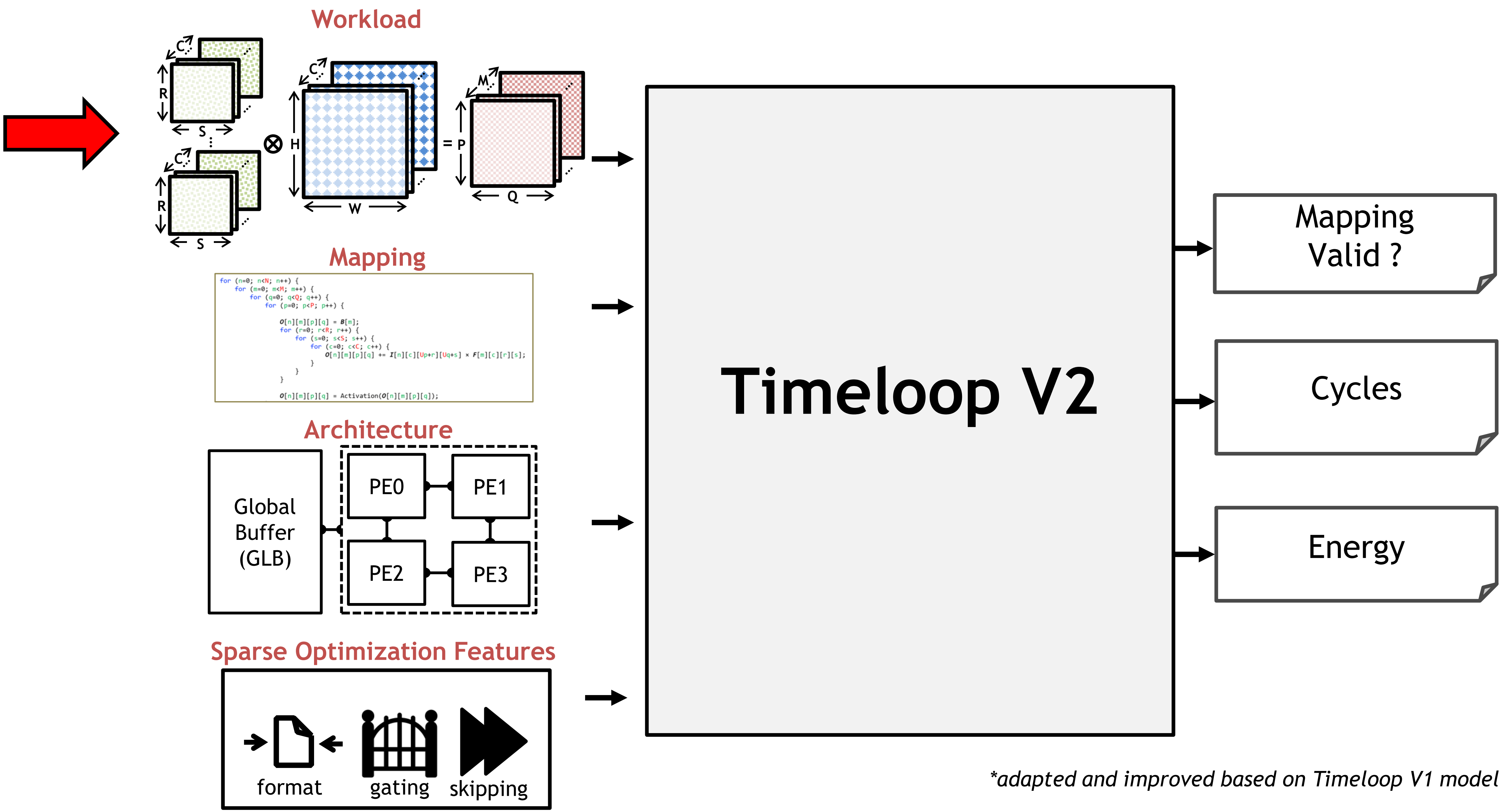
# Modeling Overview




*adapted and improved based on Timeloop V1 model

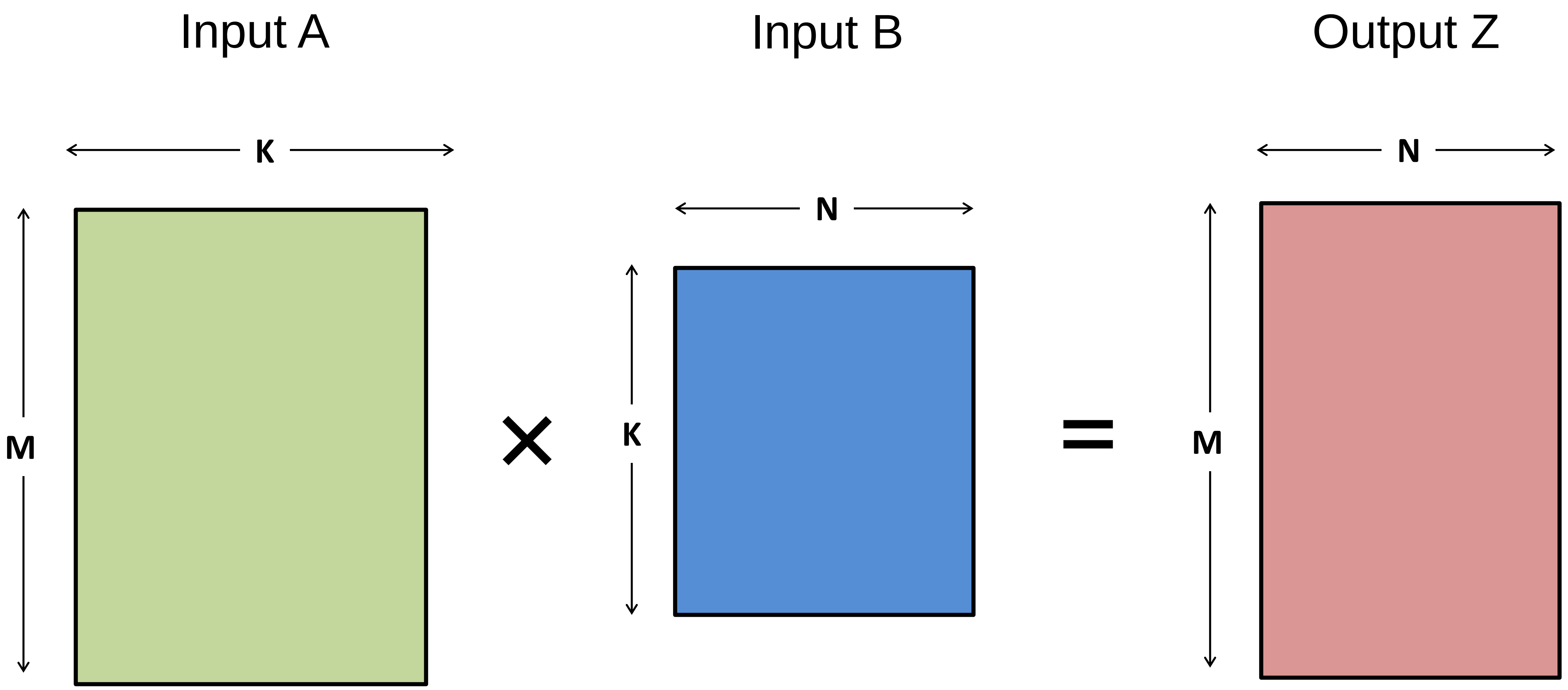
# Problem Specification

# Modeling Overview

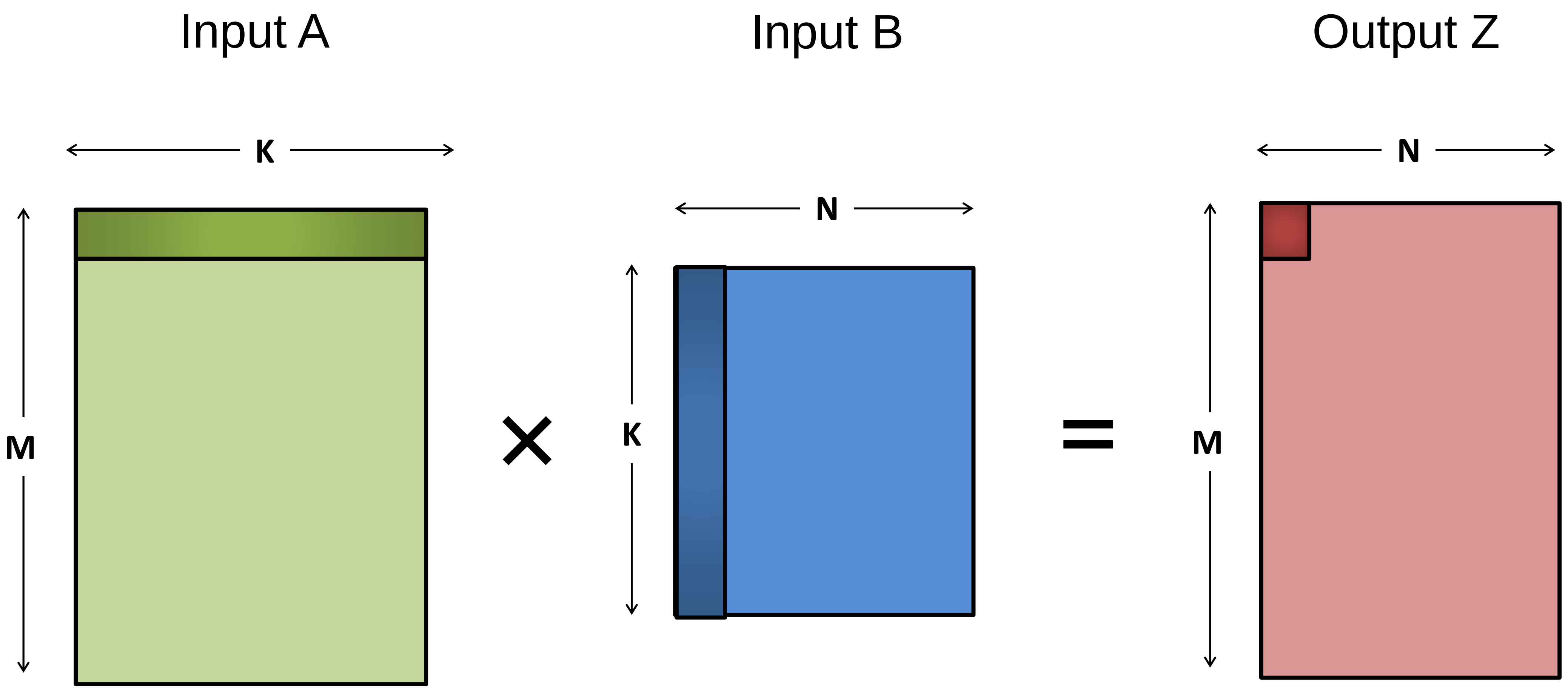



**adapted and improved based on Timeloop V1 model*

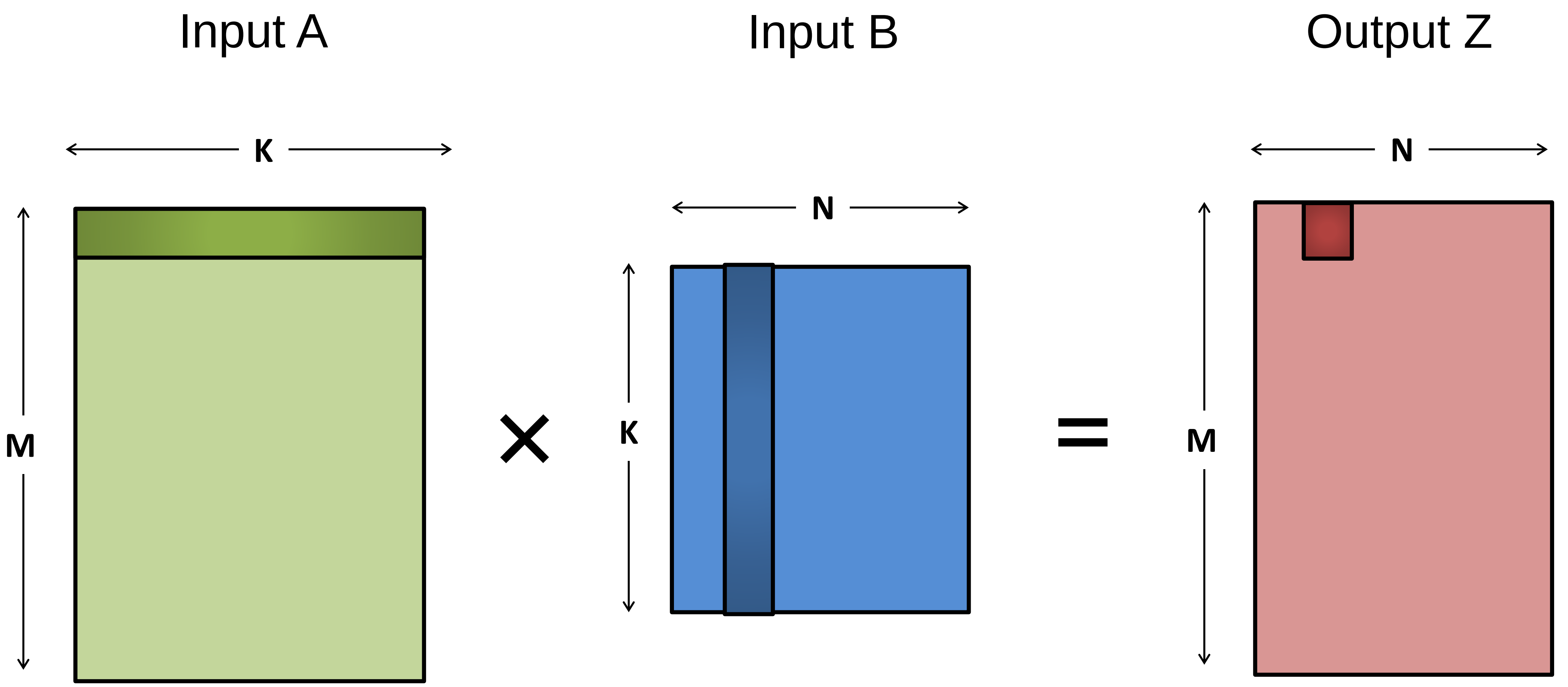
# Tensor Matrix Multiplication



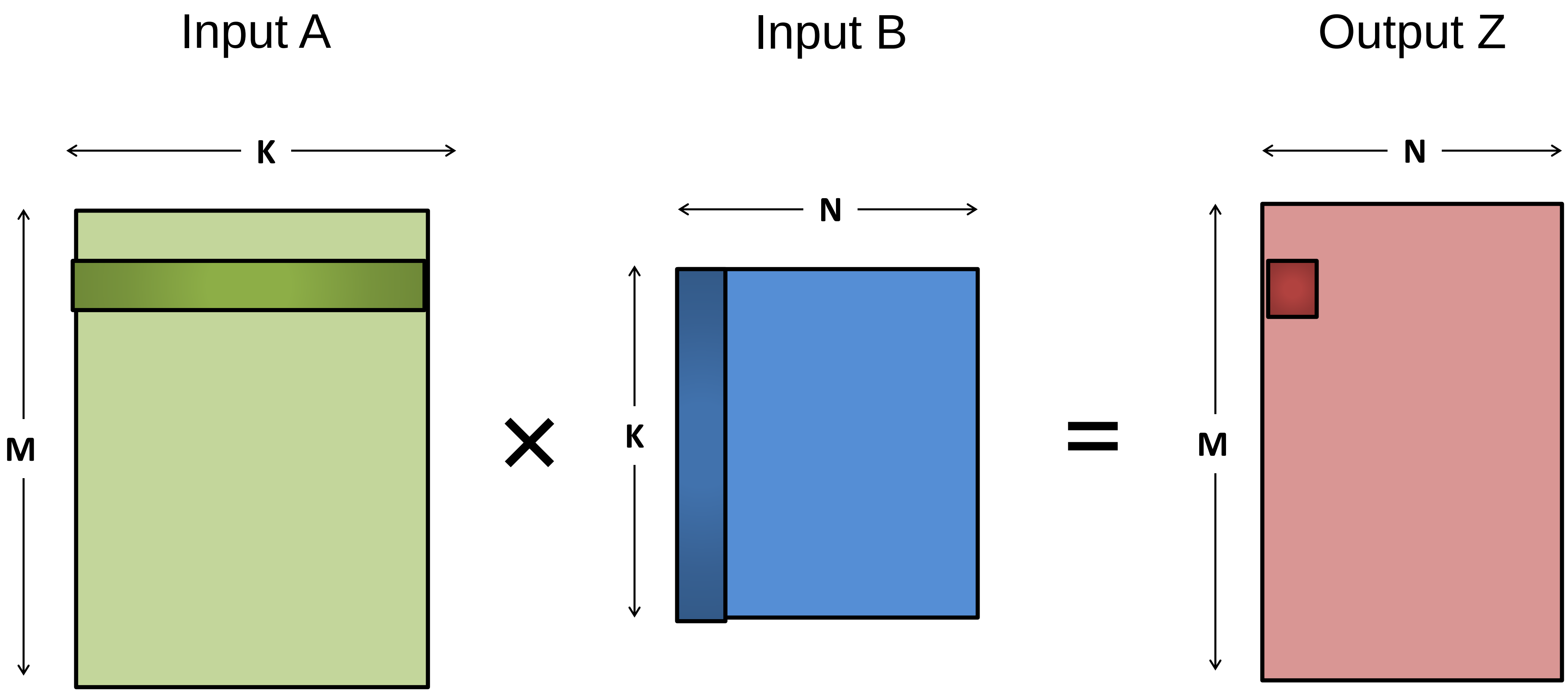
# Matrix Multiplication Tensor Computation



# Matrix Multiplication Tensor Computation



# Matrix Multiplication Tensor Computation



# Matrix Multiply - Loop Nest

```
for m in [0, M):
    for n in [0, N):
        for k in [0, K):
            Z[m][n] += A[m][k] × B[k][n]
```

# Matrix Multiply - Einsum

```
for m in [0, M):
    for n in [0, N):
        for k in [0, K):
            Z[m][n] += A[m][k] × B[k][n]
```

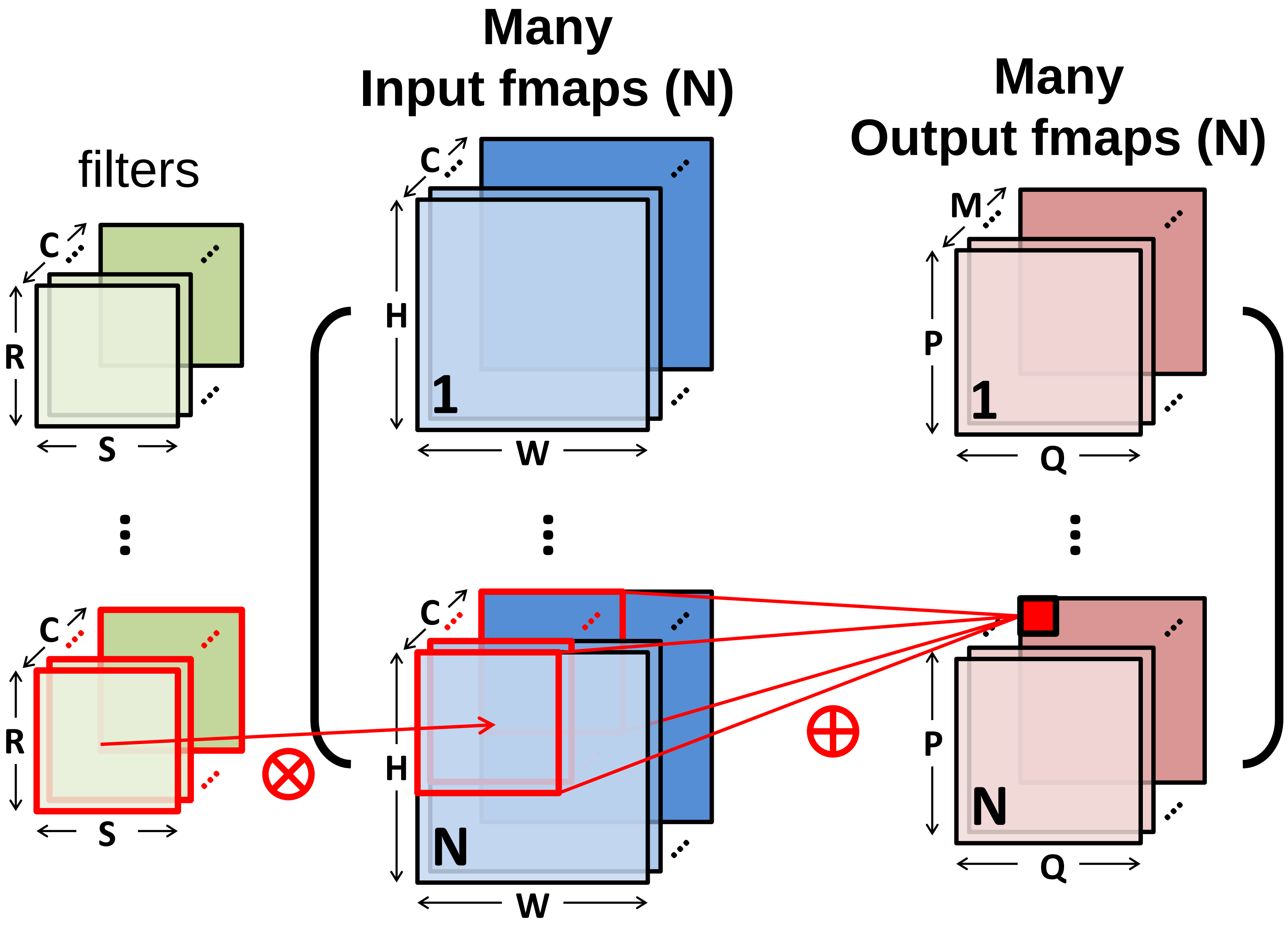
$$Z_{m,n} = A_{m,k} \times B_{k,n}$$

m, n – uncontracted dimensions
k – contracted dimension

# Matrix Multiply - Instance Specification

- M - Height of input A and output Z, e.g., M=256
- K - Width of input A and height of input B, e.g., K=128
- N - Width of input B and output Z, e.g., N=64

# Tensor Convolution




Batch Size (N)

# Convolution - Loop Nest

```
for n in [0, N):
    for  m in [0, M):
        for q in [0, Q):
            for p in [0, P):
                O[n][m][p][q] = B[m]

                for r in [0, R):
                    for s in [0, S):
                        for c in [0,C):
                            O[n][m][p][q] += I[n][c][Up+r][Uq+s] ×
                                             F[m][c][r][s]

                O[n][m][p][q] = Activation(O[n][m][p][q]);
```

For each output fmap value

Convolve a window and apply activation

# Convolution - Einsum

```
for n in [0, N):
     for  m in [0, M):
          for q in [0, Q):
               for p in [0, P):
                    O[n][m][p][q] = B[m]

                    for r in [0, R):
                         for s in [0, S):
                              for c in [0,C):
                                   O[n][m][p][q] += I[n][c][Up+r][Uq+s] ×
                                                    F[m][c][r][s]

                    O[n][m][p][q] = Activation(O[n][m][p][q]);
```

For each output fmap value

Convolve a window and apply activation

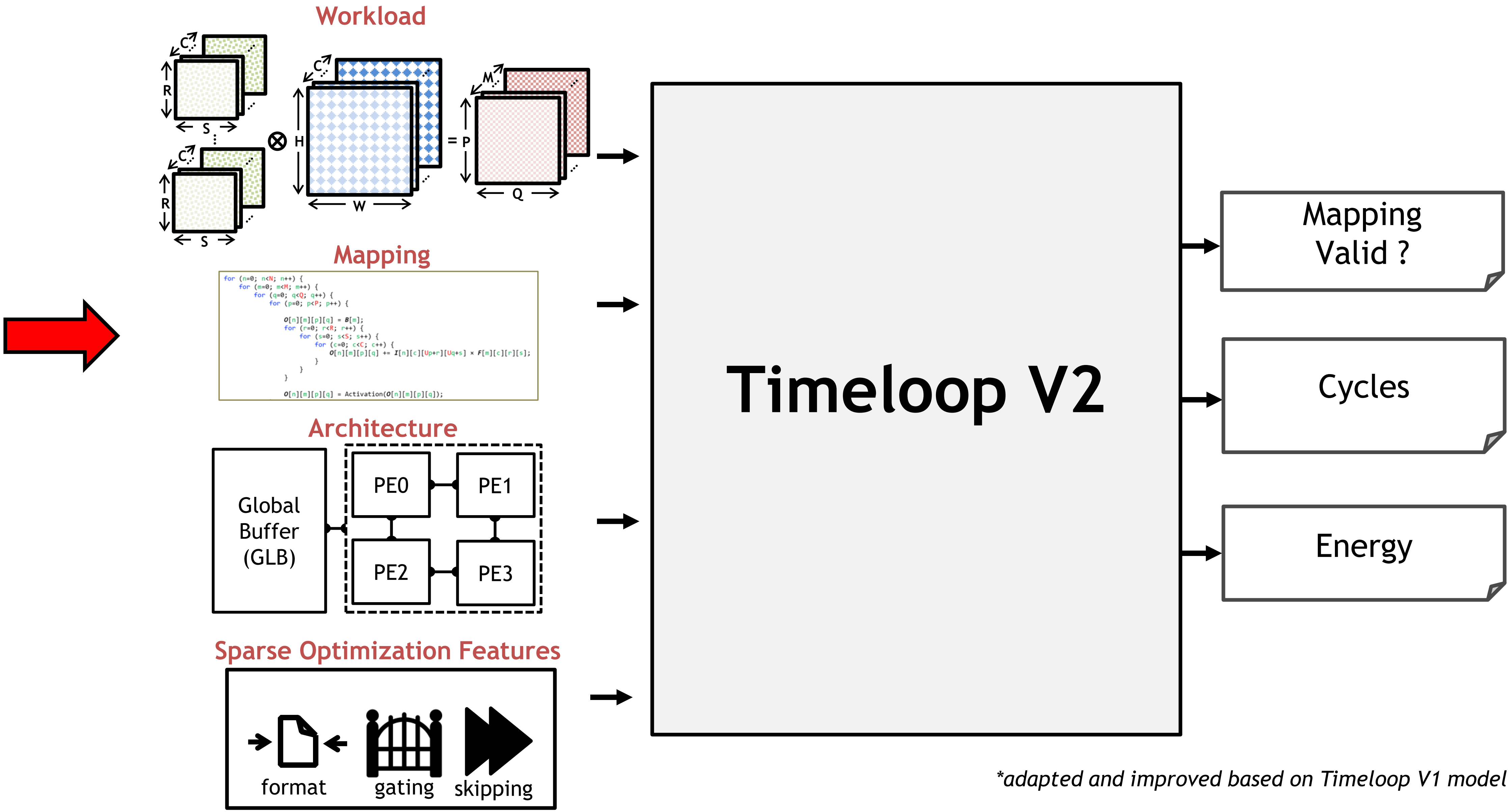
$$O_{n,m,p,q} = I_{n,c,(p+r),(q+s)} \times F_{m,c,r,s}$$

# Convolution - Instance Specification

- N - Number of input fmaps/output fmaps (batch size)
- C - Number of channels in input fmaps (activations) & filters (weights)
- H - Height of input fmap (activations)
- W - Width of input fmap (activations)
- R - Height of filter (weights)
- S - Width of filter (weights)
- M - Number of channels in output fmaps (activations)
- P - Height of output fmap (activations)
- Q - Width of output fmap (activations)
- U - Stride of convolution

# Schedule Specification

# Modeling Overview

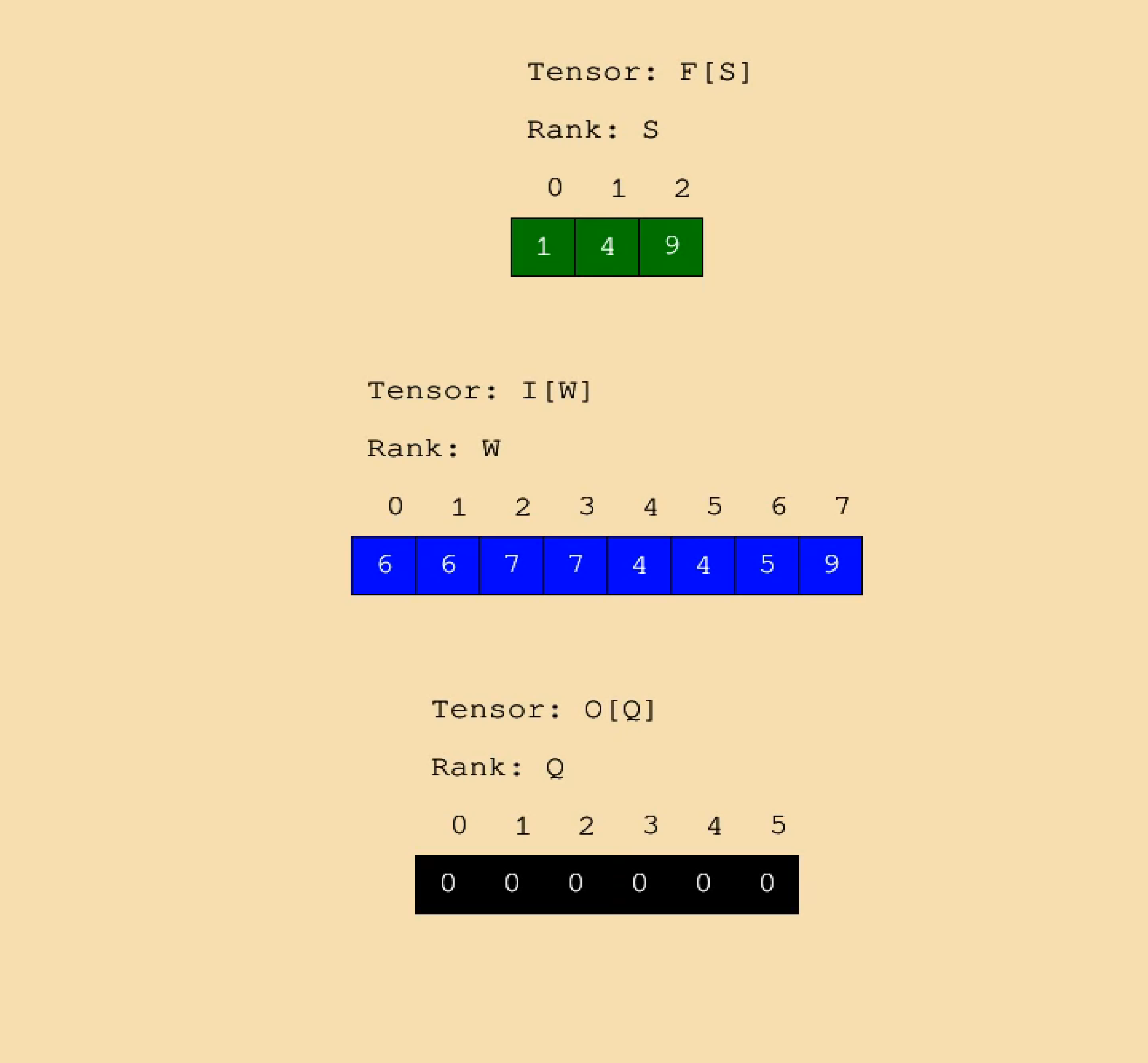



*adapted and improved based on Timeloop V1 model

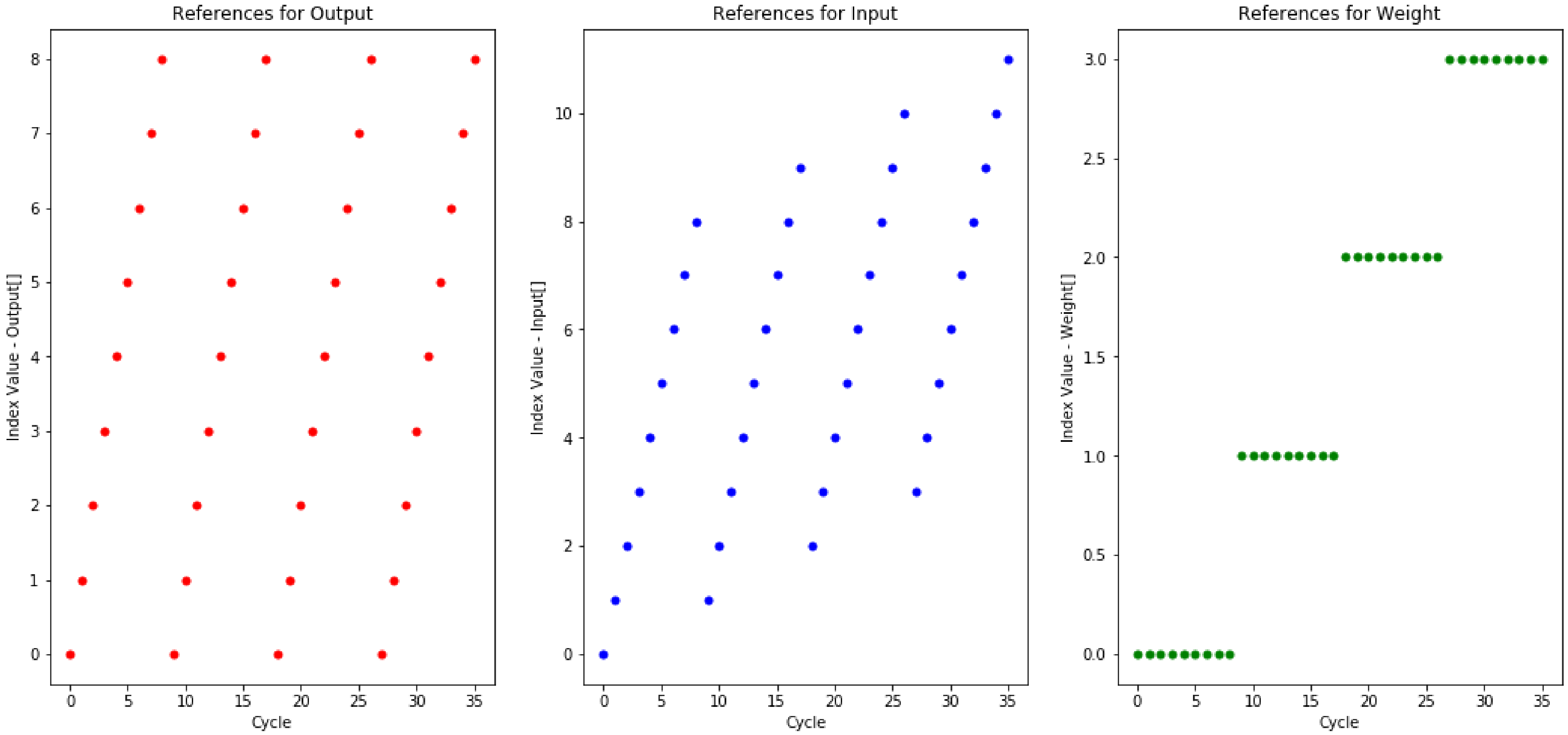
# Weight stationary dataflow

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

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

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



# Weight Stationary - Reference Pattern



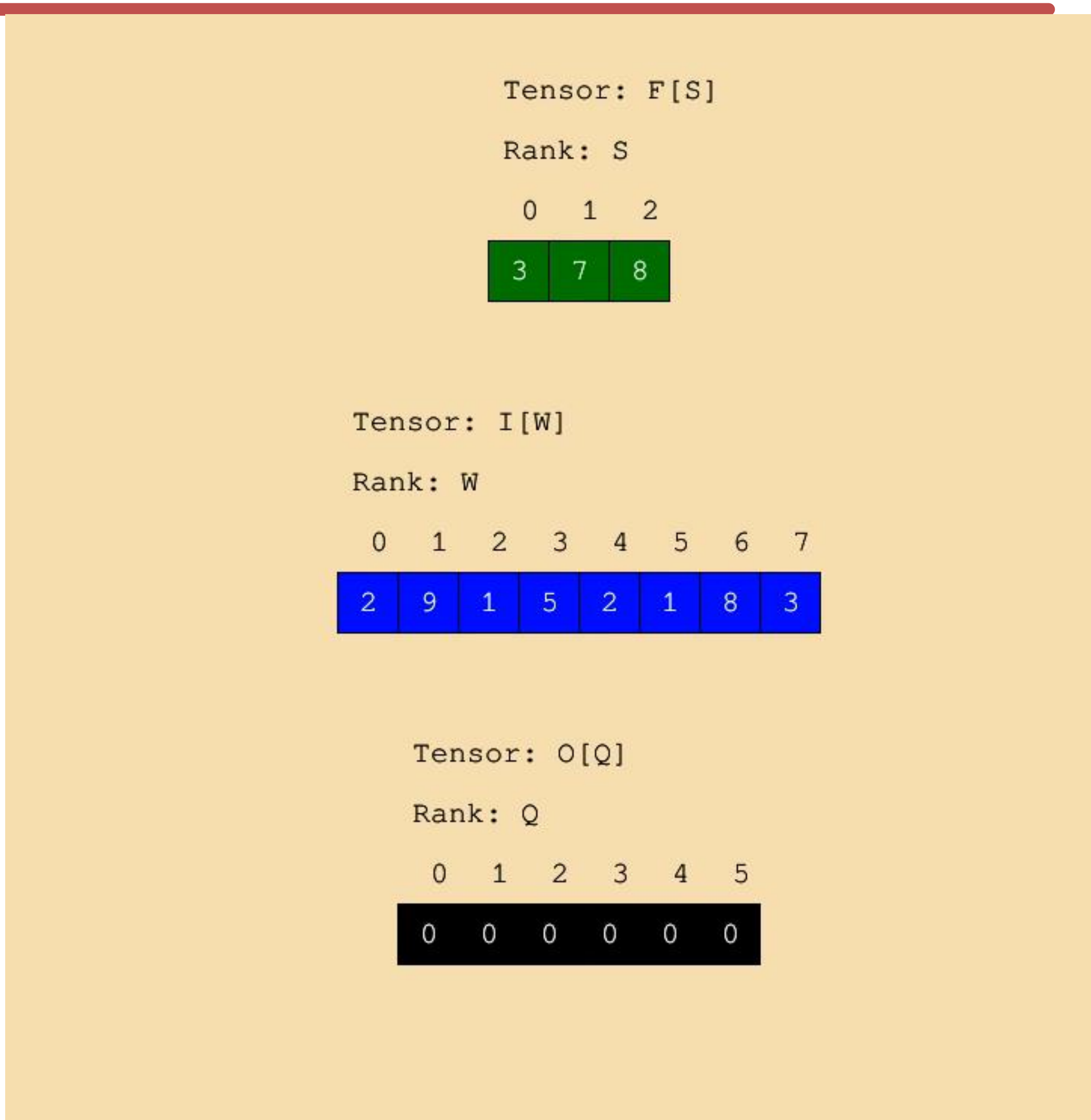

Observations:

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

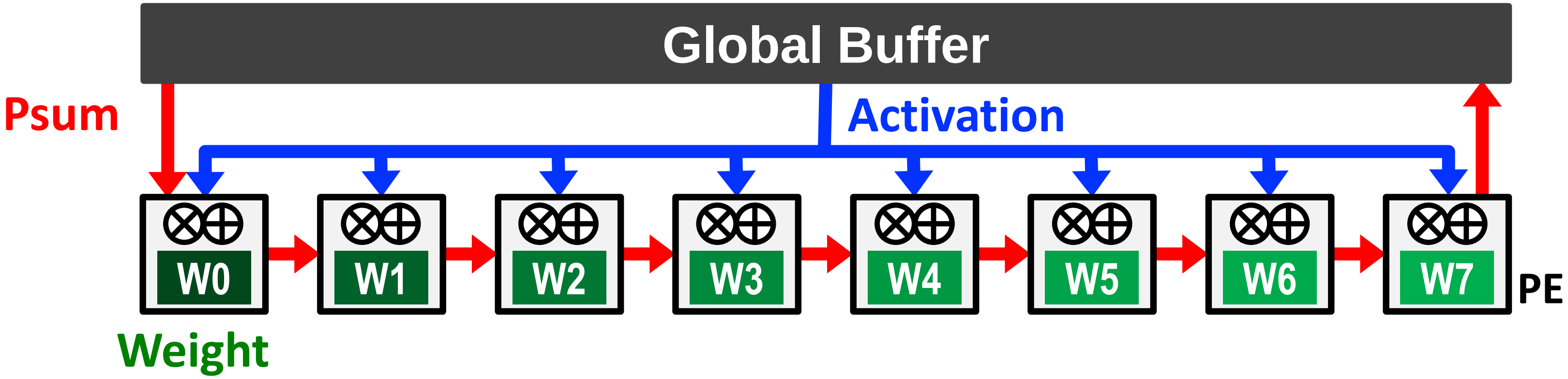
# Parallel Weight Stationary - Animation

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

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

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



# Weight Stationary (WS)



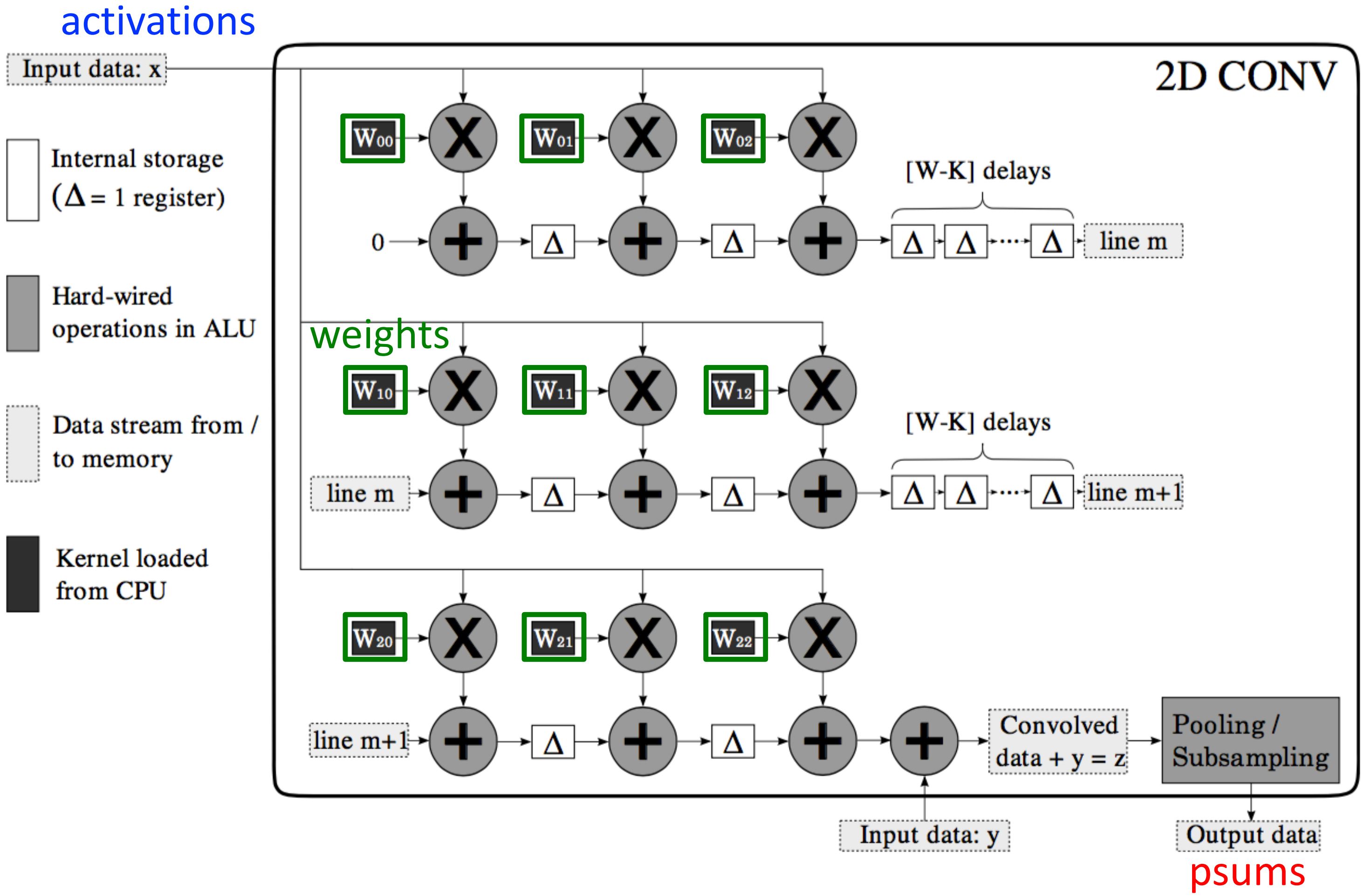

- **Minimize weight** read energy consumption
  - maximize convolutional and filter reuse of weights

- **Broadcast activations** and **accumulate psums spatially** across the PE array.

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

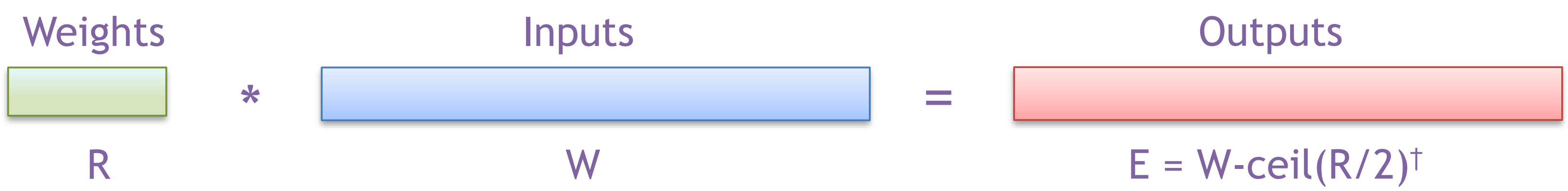
# WS Example: nn-X (NeuFlow)

## A 3×3 2D Convolution Engine




[Farabet et al., ICCV 2009]

# 1-D Convolution - Weight Stationary




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

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

No constraints on loop permutations!

† Assuming: ‘valid’ style convolution

# 1-D Convolution

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

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

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

Tensor: F[S]

Rank: S

| 0 | 1 | 2 |
|---|---|---|
| 1 | 4 | 9 |

Tensor: I[W]

Rank: W

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 6 | 7 | 7 | 4 | 4 | 5 | 9 |

Tensor: O[Q]

Rank: Q

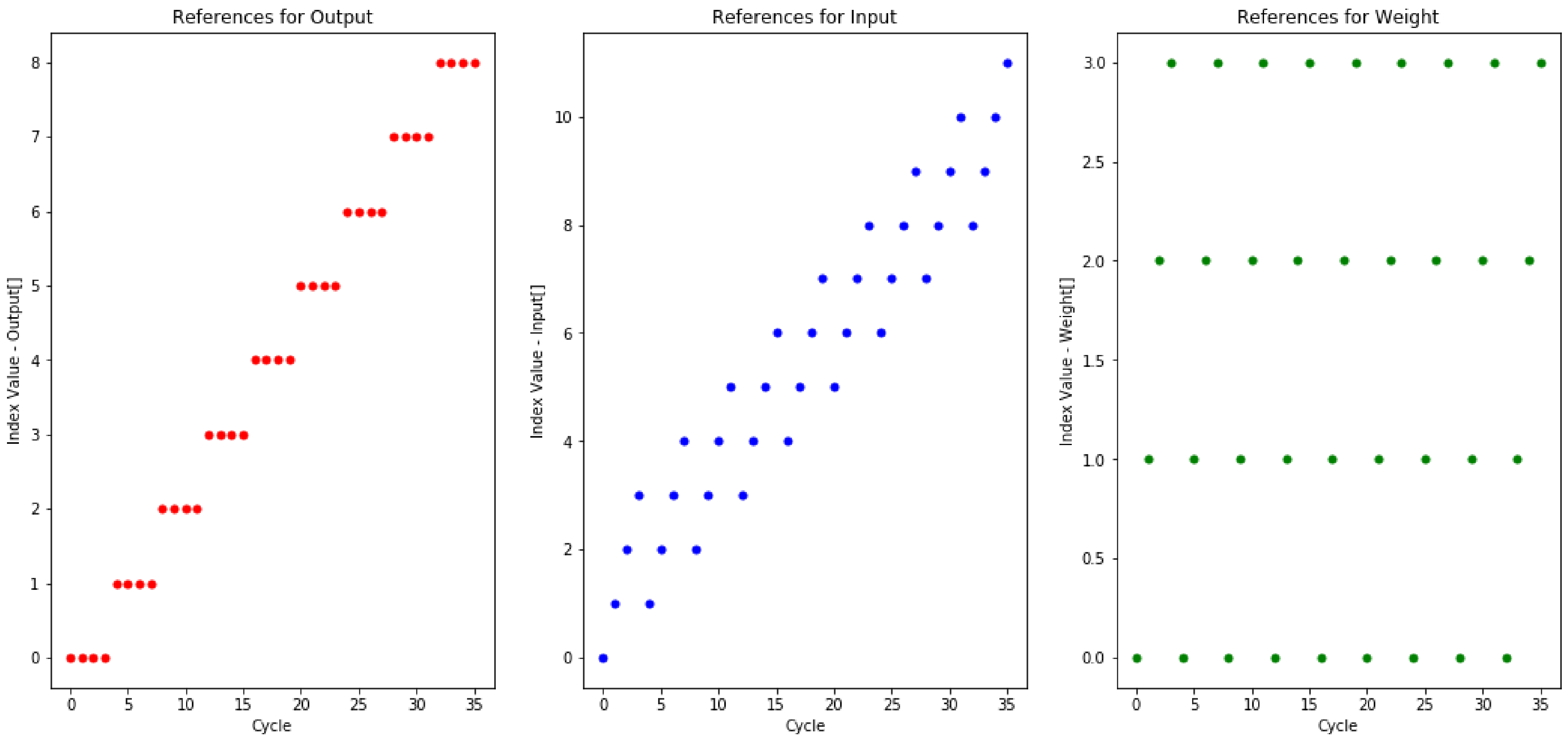
| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |

# Output Stationary - Reference Pattern

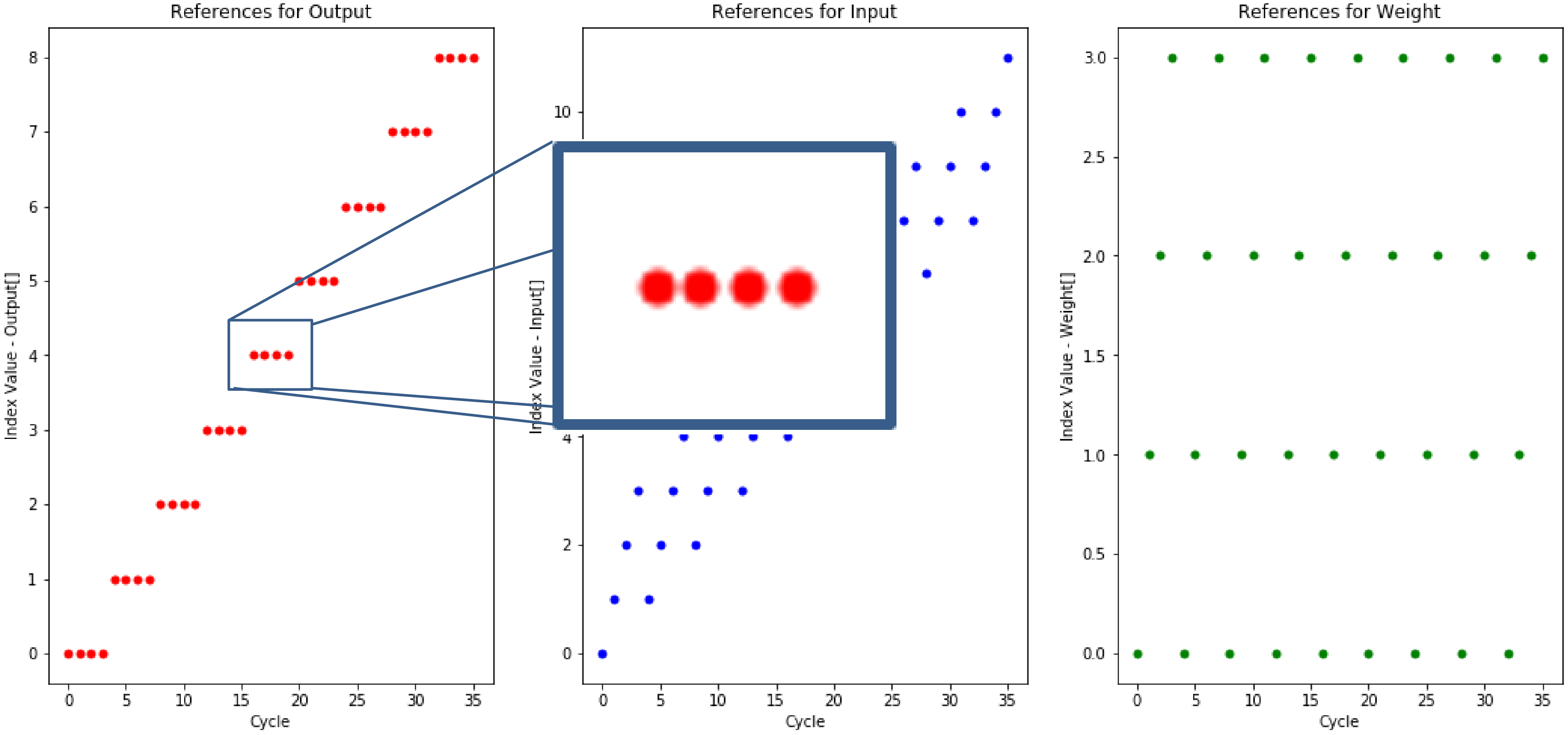
```
for q in [0, Q):
        for s in [0, S):
            o[q] += i[q+s]*f[s]
```

Instance:

- S = 4
- Q = 9
- W = 12

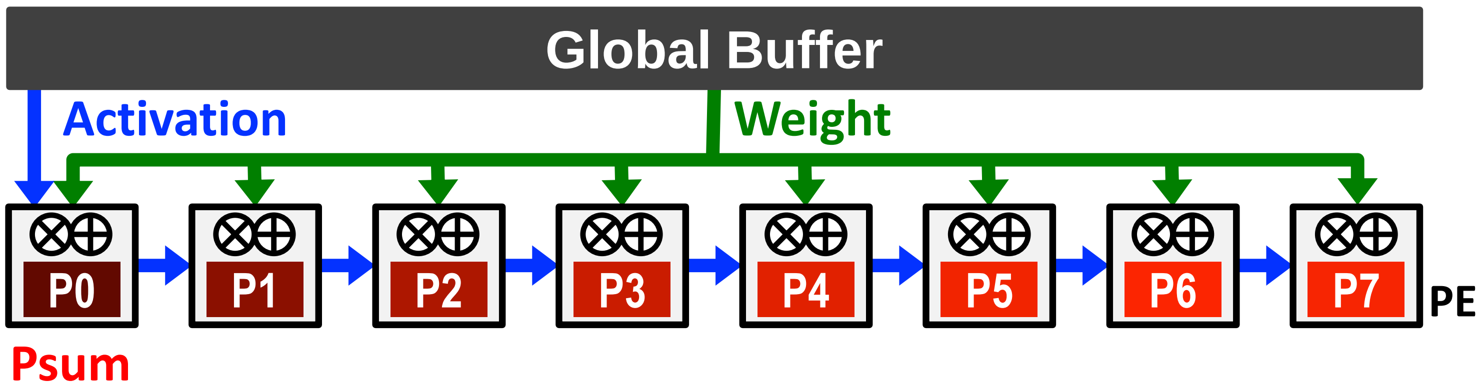


# Output Stationary - Reference Pattern




Observations: - Single output is reused many times (S)

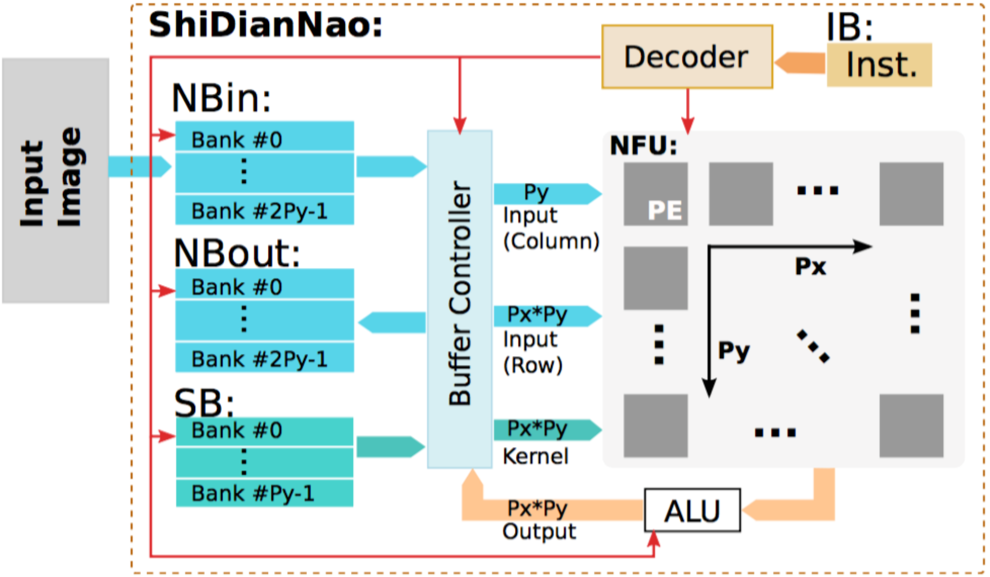
# Output Stationary (OS)



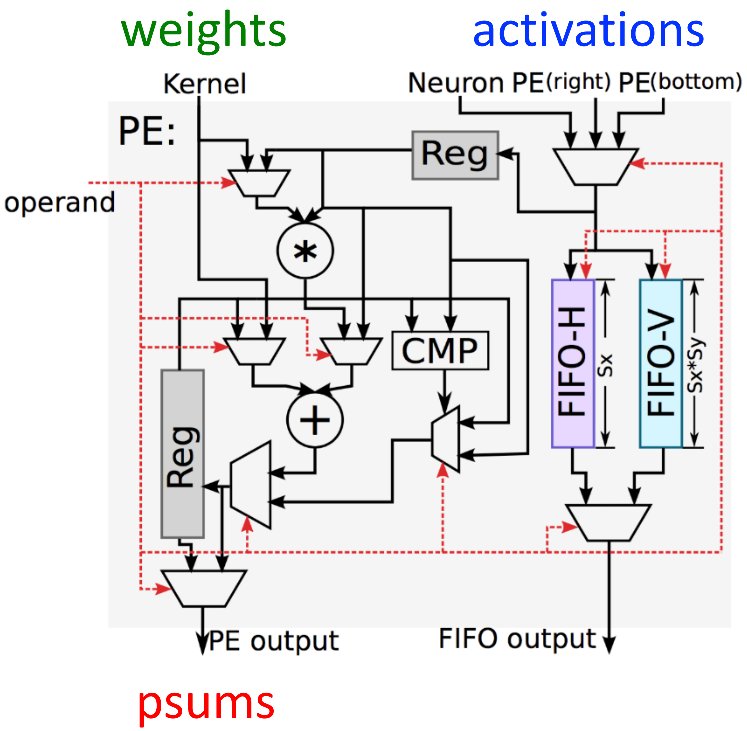

- **Minimize** **partial sum** R/W energy consumption
  - maximize local accumulation
- **Broadcast/Multicast** **filter weights** and **reuse** **activations** **spatially** across the PE array

# OS Example: ShiDianNao

## Top-Level Architecture




## PE Architecture

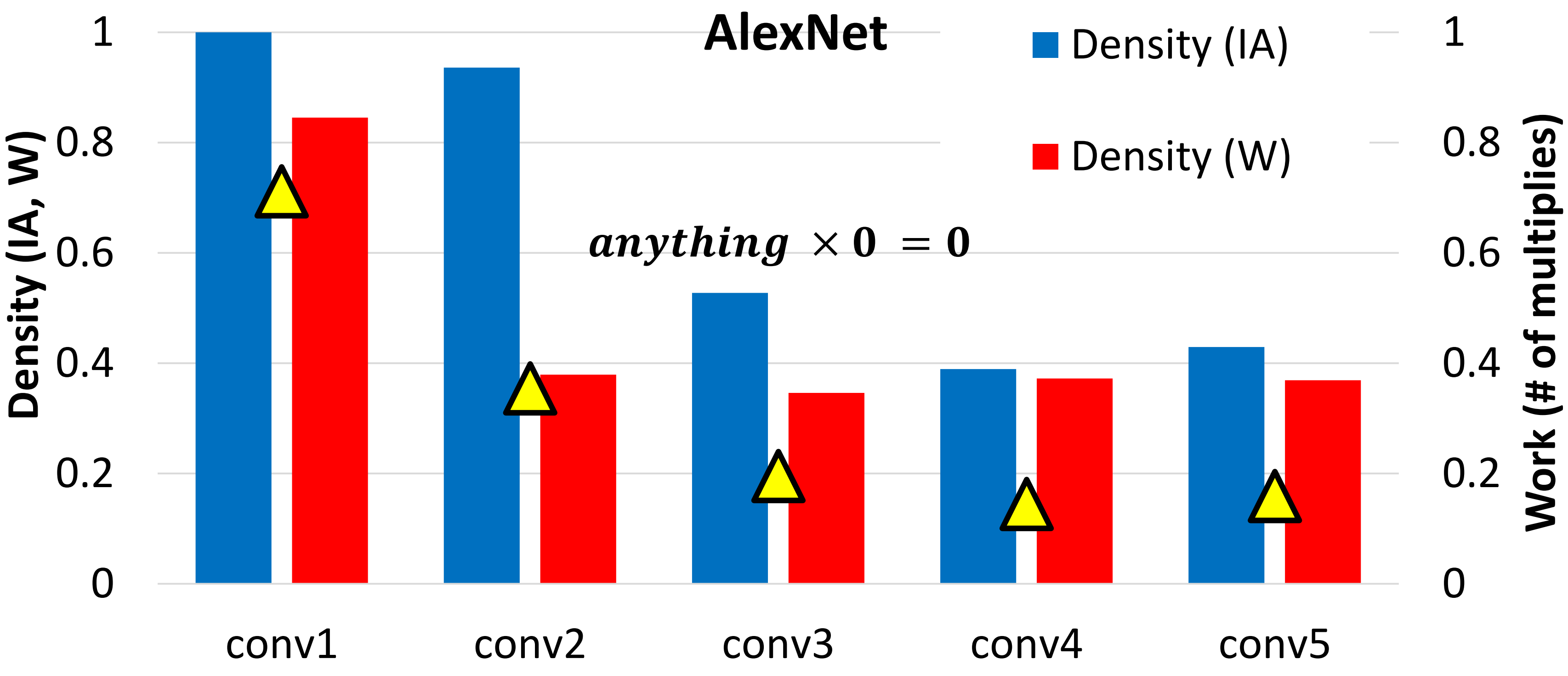



- Inputs streamed through array
- Weights broadcast
- Partial sums accumulated in PE and streamed out

[Du et al., ISCA 2015]

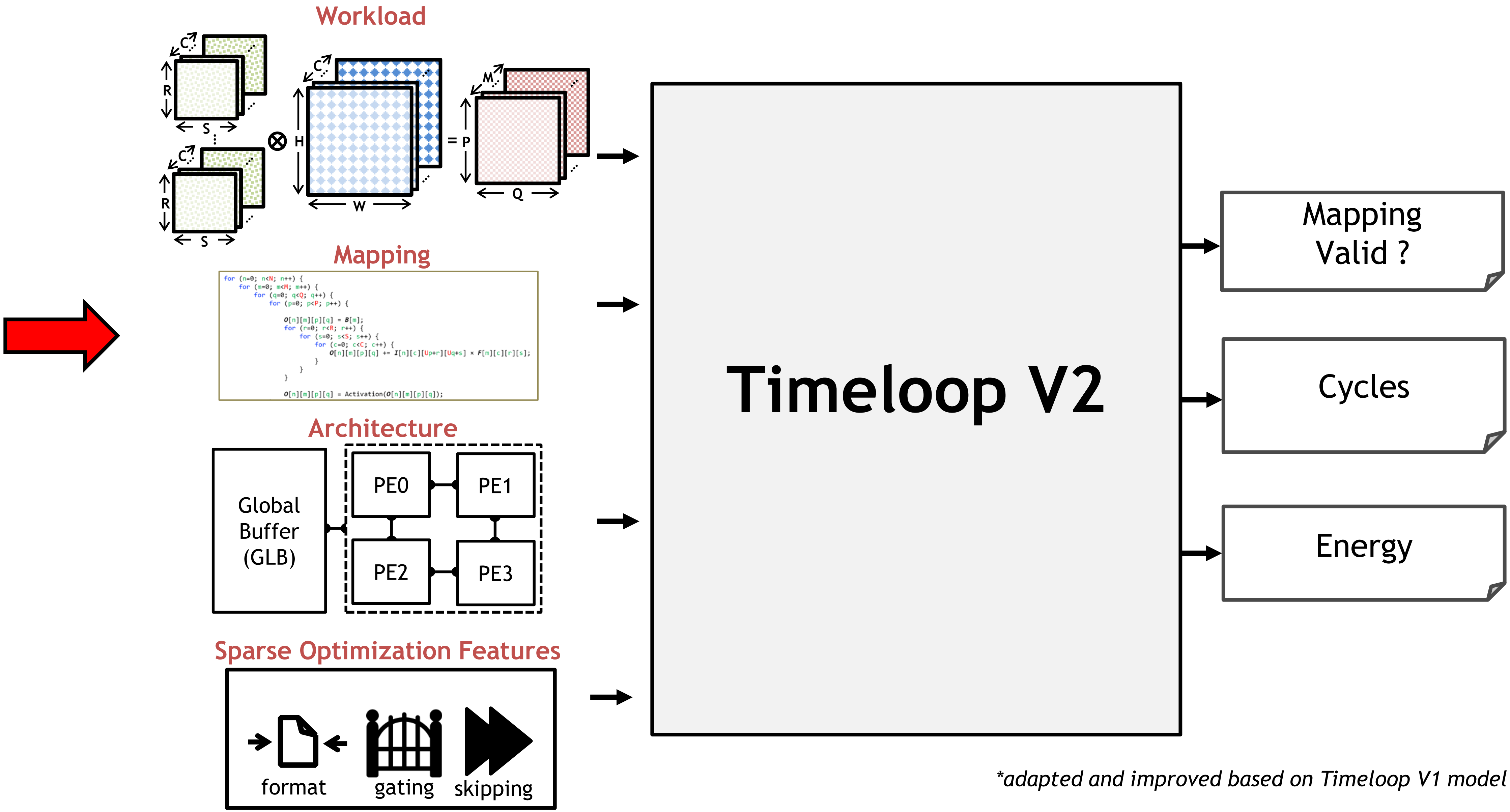
# Motivation

- Leverage CNN sparsity to improve energy-efficiency



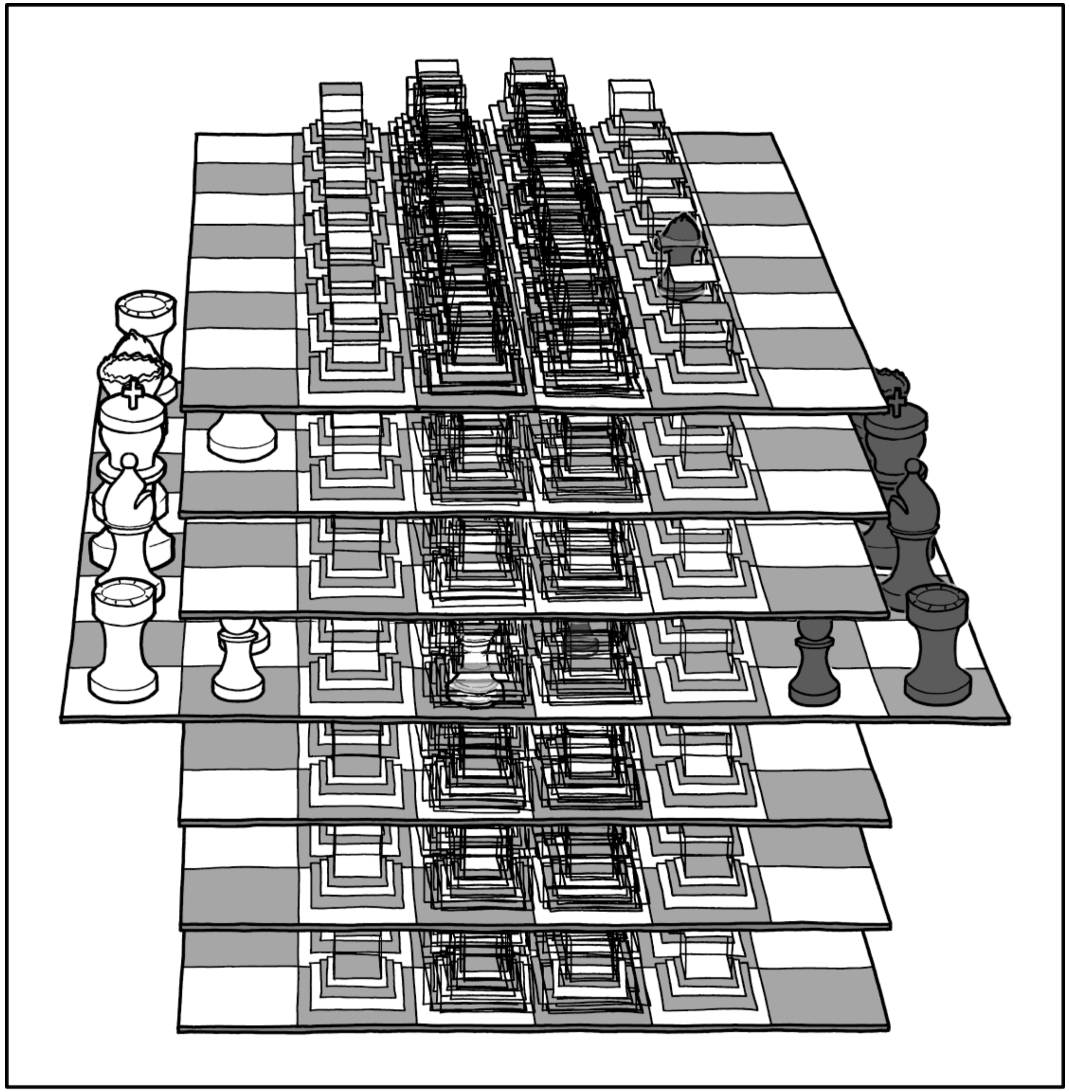
# Tensor Abstraction

# Modeling Overview




**adapted and improved based on Timeloop V1 model*

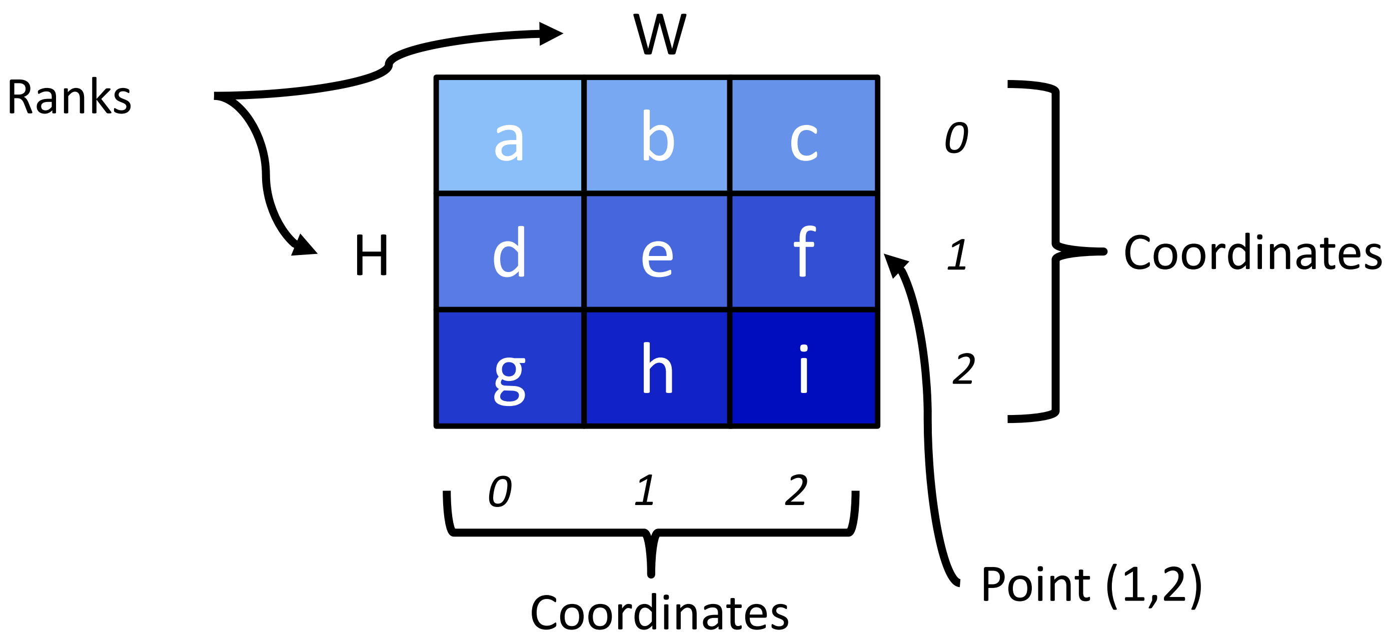
# "In Dimensional Chess, every move is annotated '?!'."



THE PROBLEM WITH N-DIMENSIONAL CHESS IS THAT N IS A CONSTANT ACROSS THE BOARD. IN MY NEW VARIANT, EVERY ROW HAS ONE MORE DIMENSION THAN THE ONE BEHIND IT.

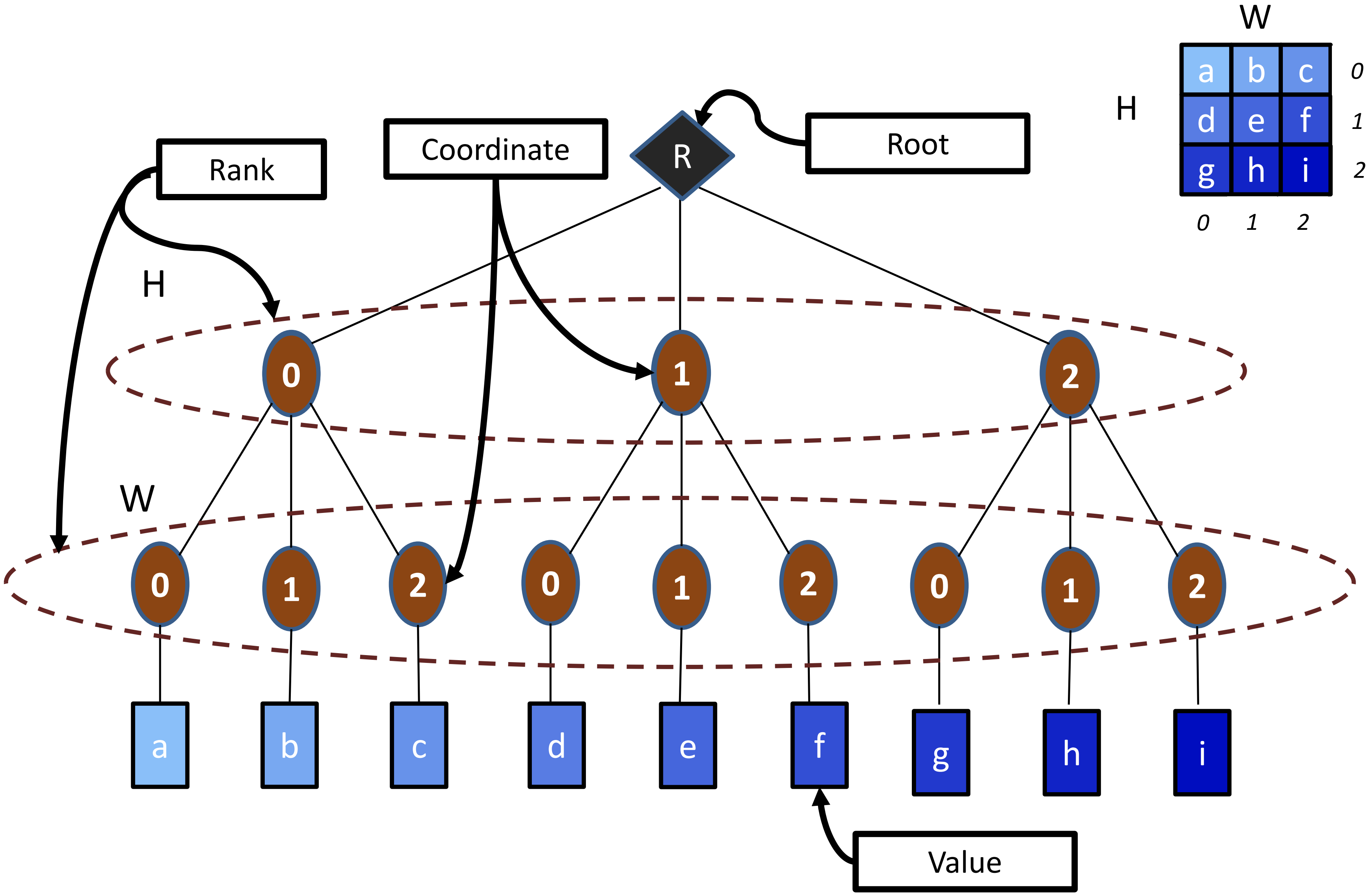
Source: XKCD/2465

# Tensor Data Terminology



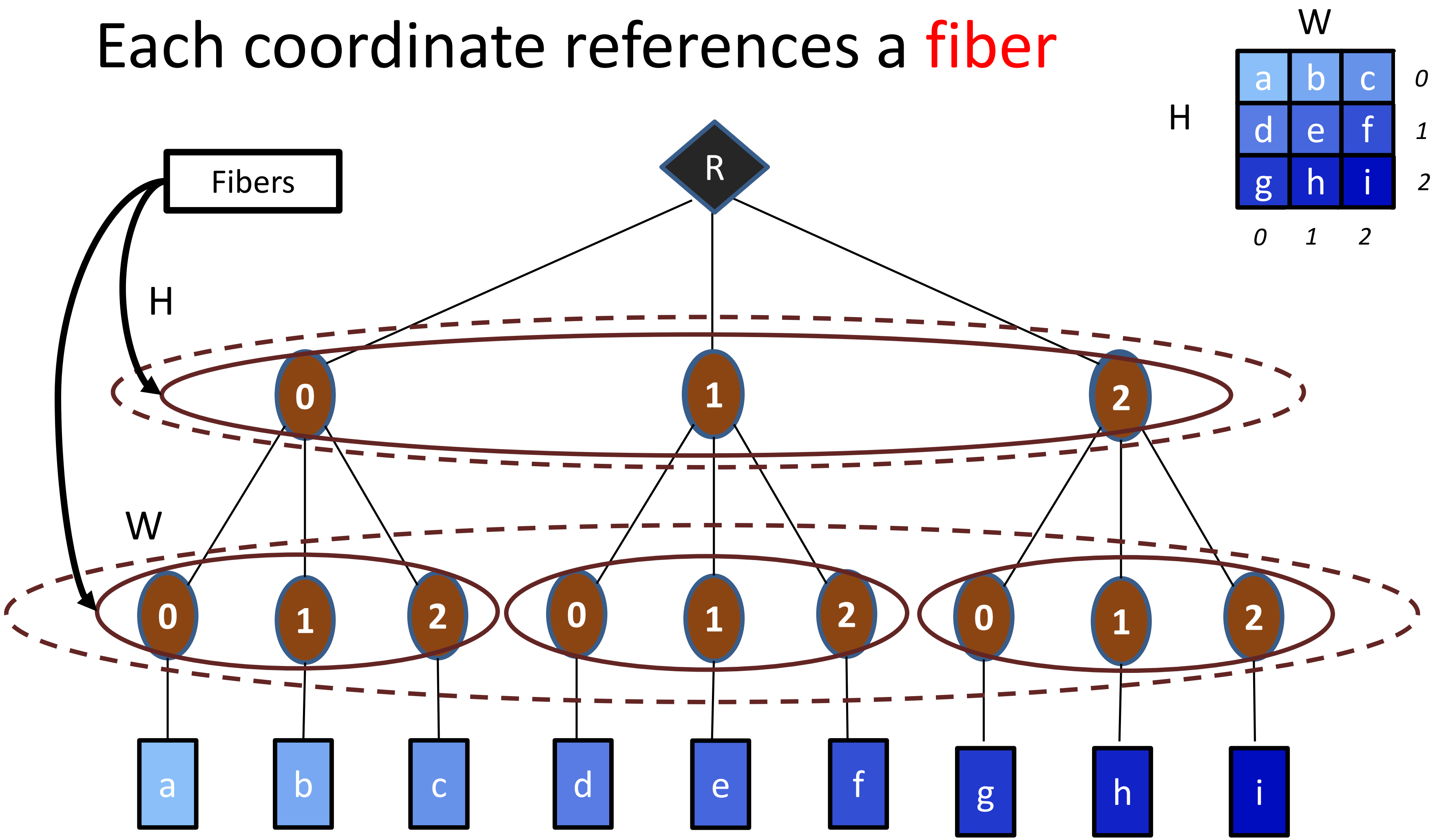

- The elements of each "rank" (dimension) are identified by their "coordinates", e.g., rank H has coordinates 0, 1, 2
- Each element of the tensor is identified by the tuple of coordinates from each of its ranks, i.e., a "point".
  So (1,2) -> "f"

# Tree-based Tensor Abstraction



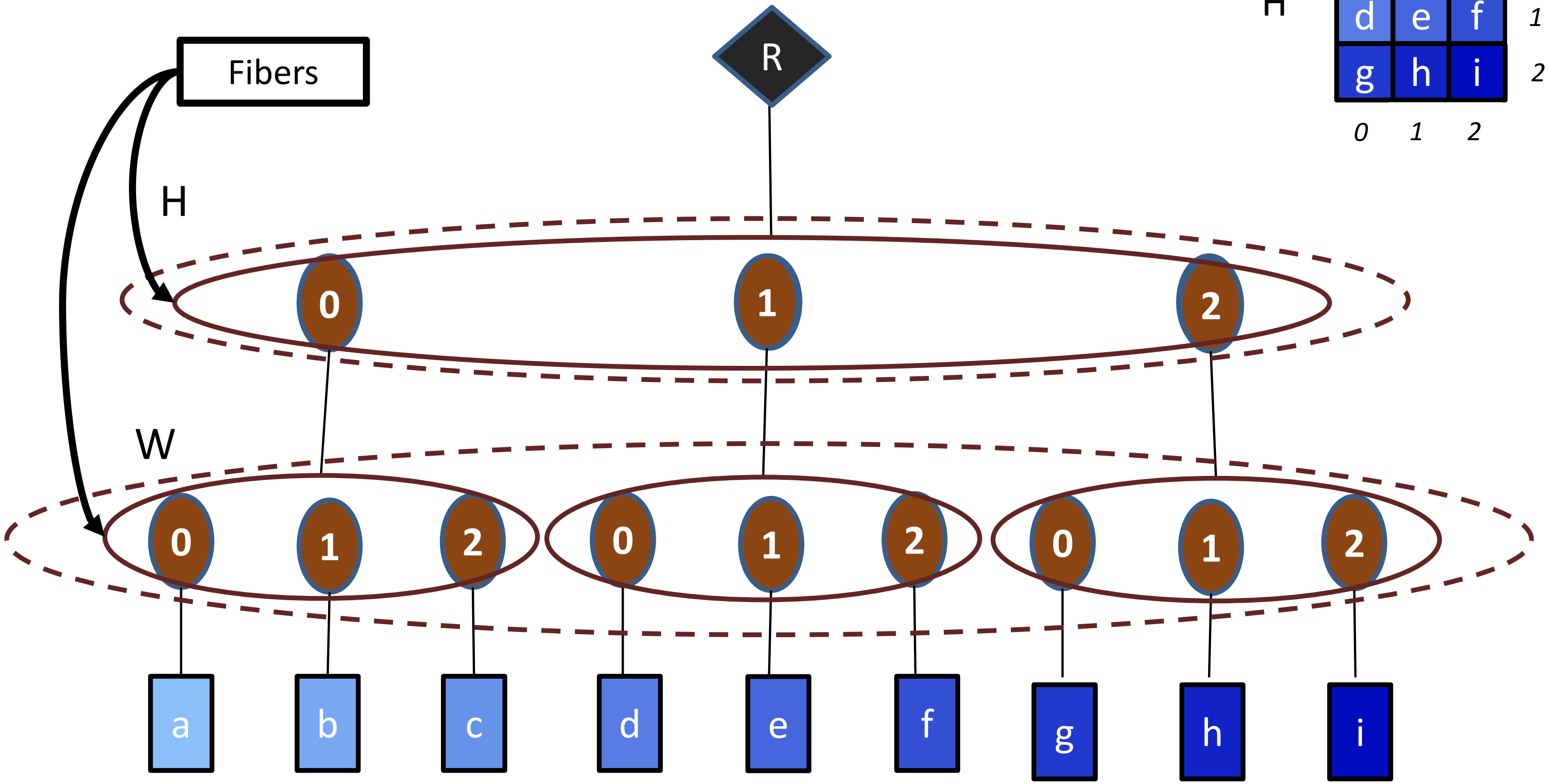
# Tree-based Tensor Abstraction

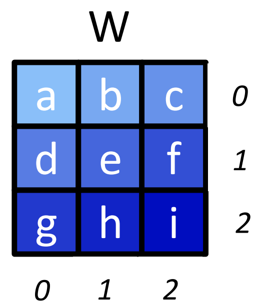
## Each coordinate references a fiber



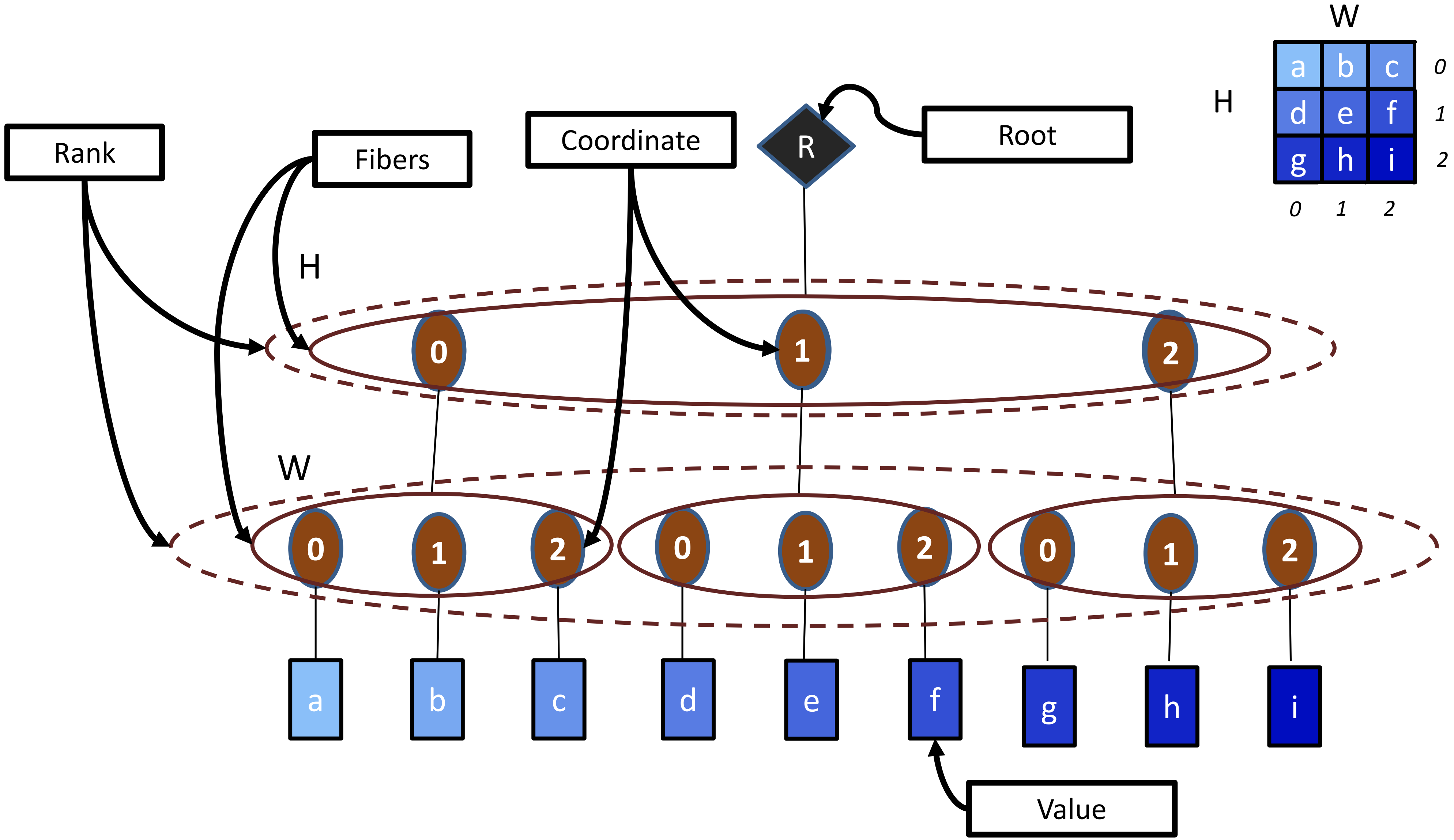
# Fiber-Tree Tensor Abstraction

Each coordinate references a fiber



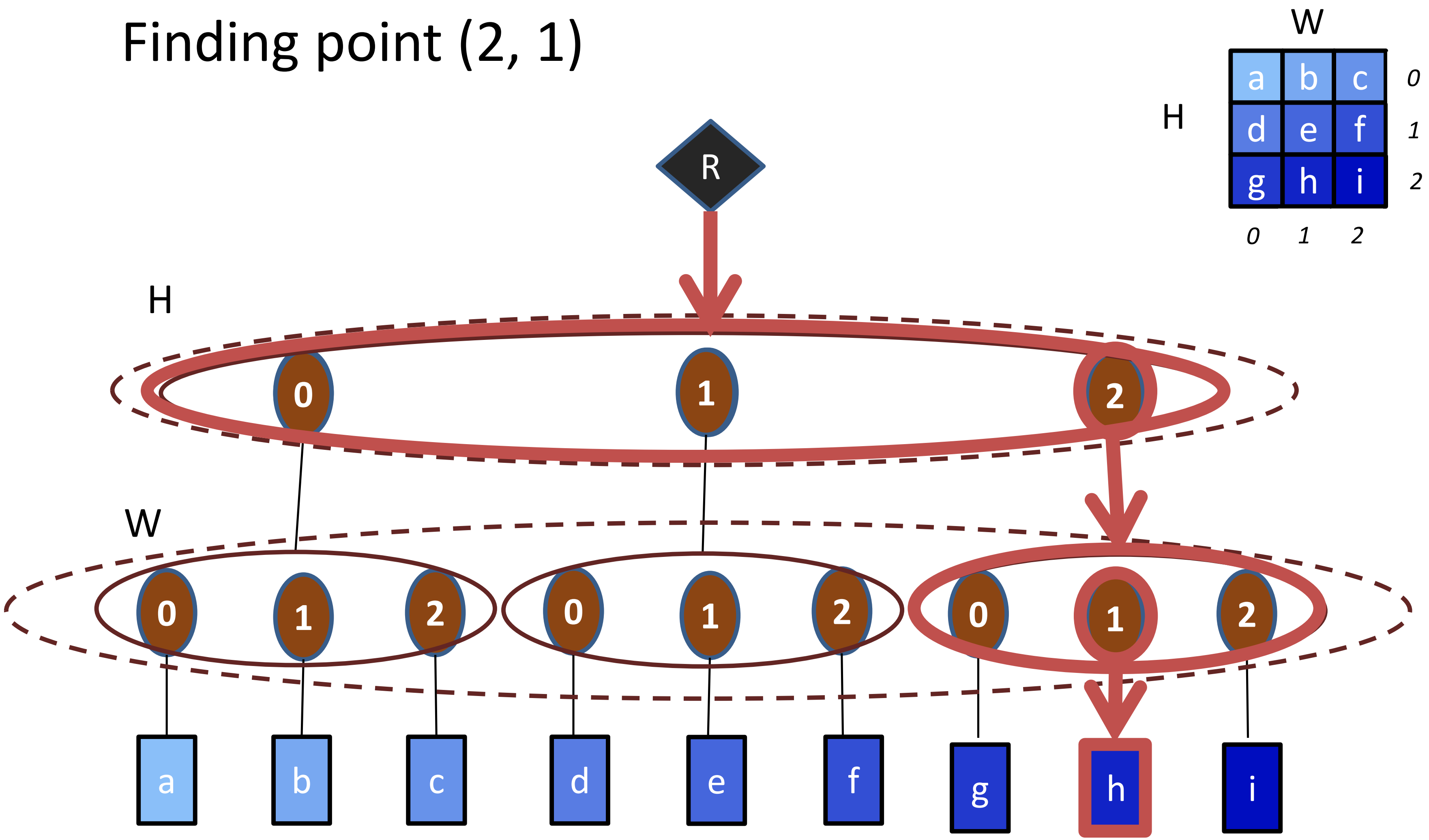



# Fiber-Tree Tensor Abstraction



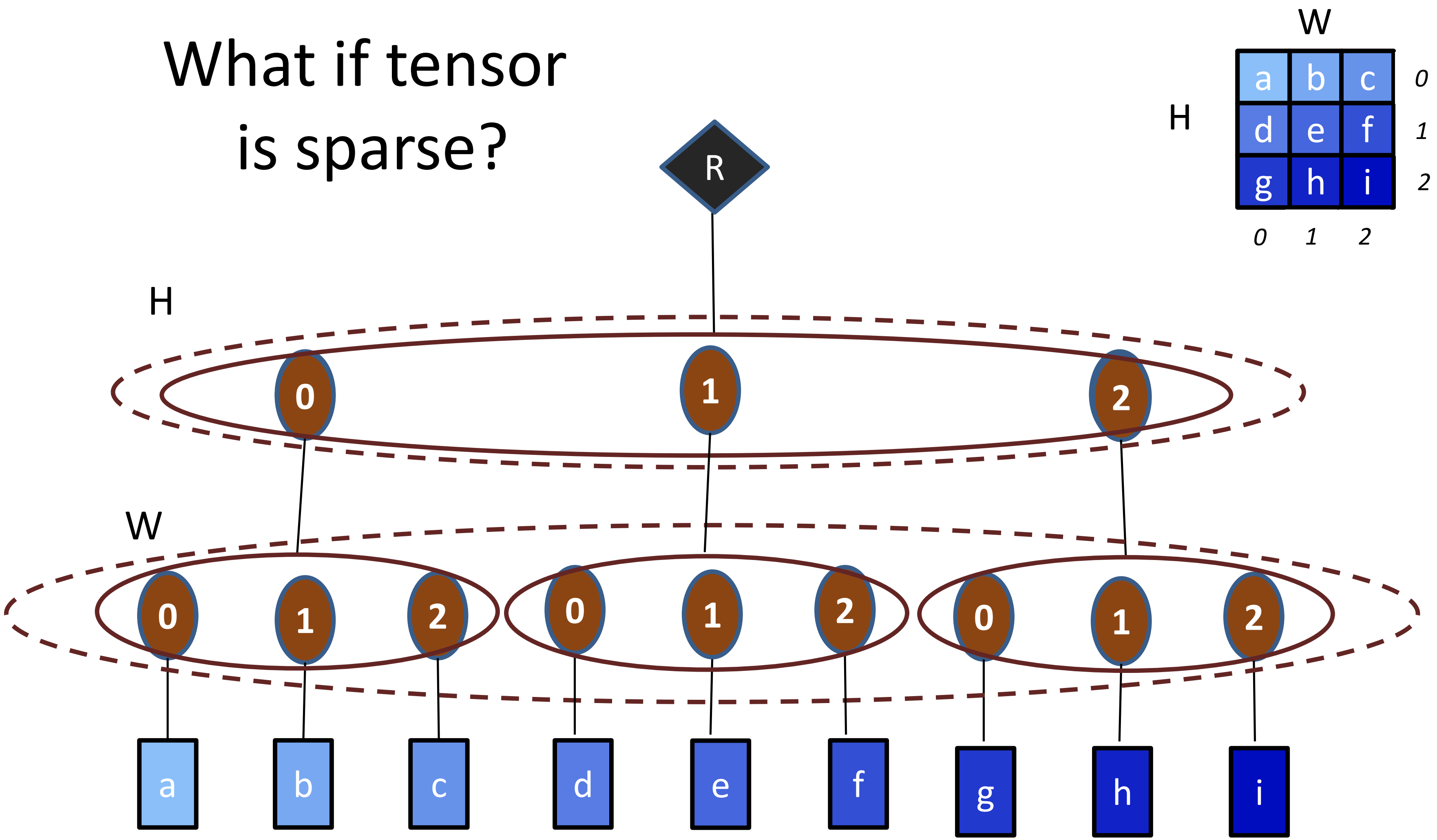
# Fiber-Tree Tensor Abstraction

Finding point (2, 1)



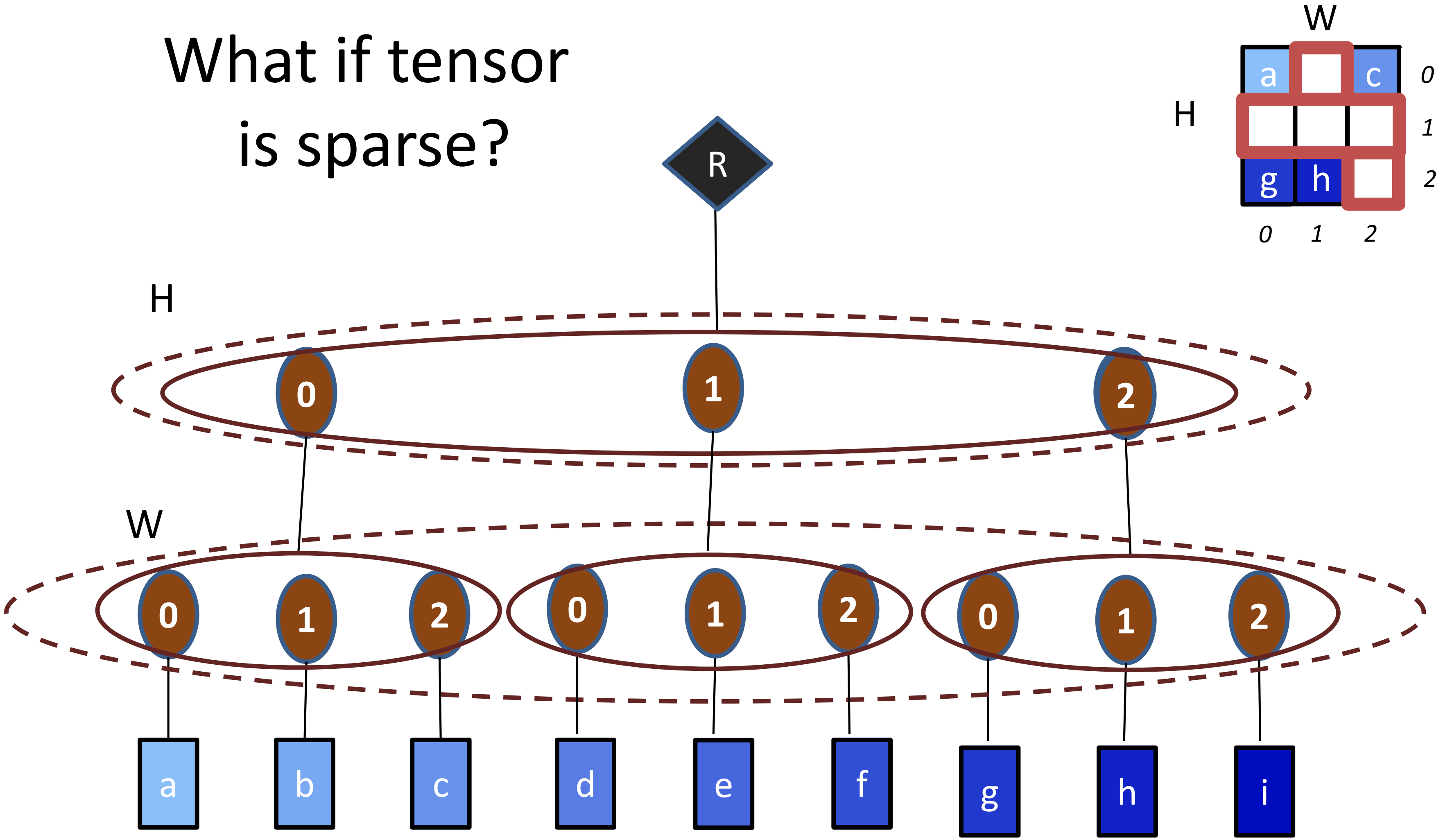
# Tree-based Tensor Abstraction

What if tensor
is sparse?



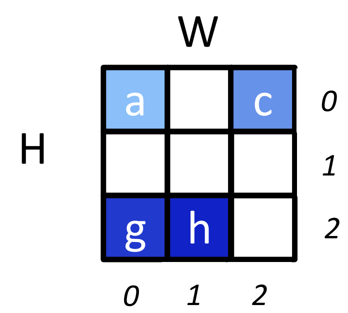
# Tree-based Tensor Abstraction

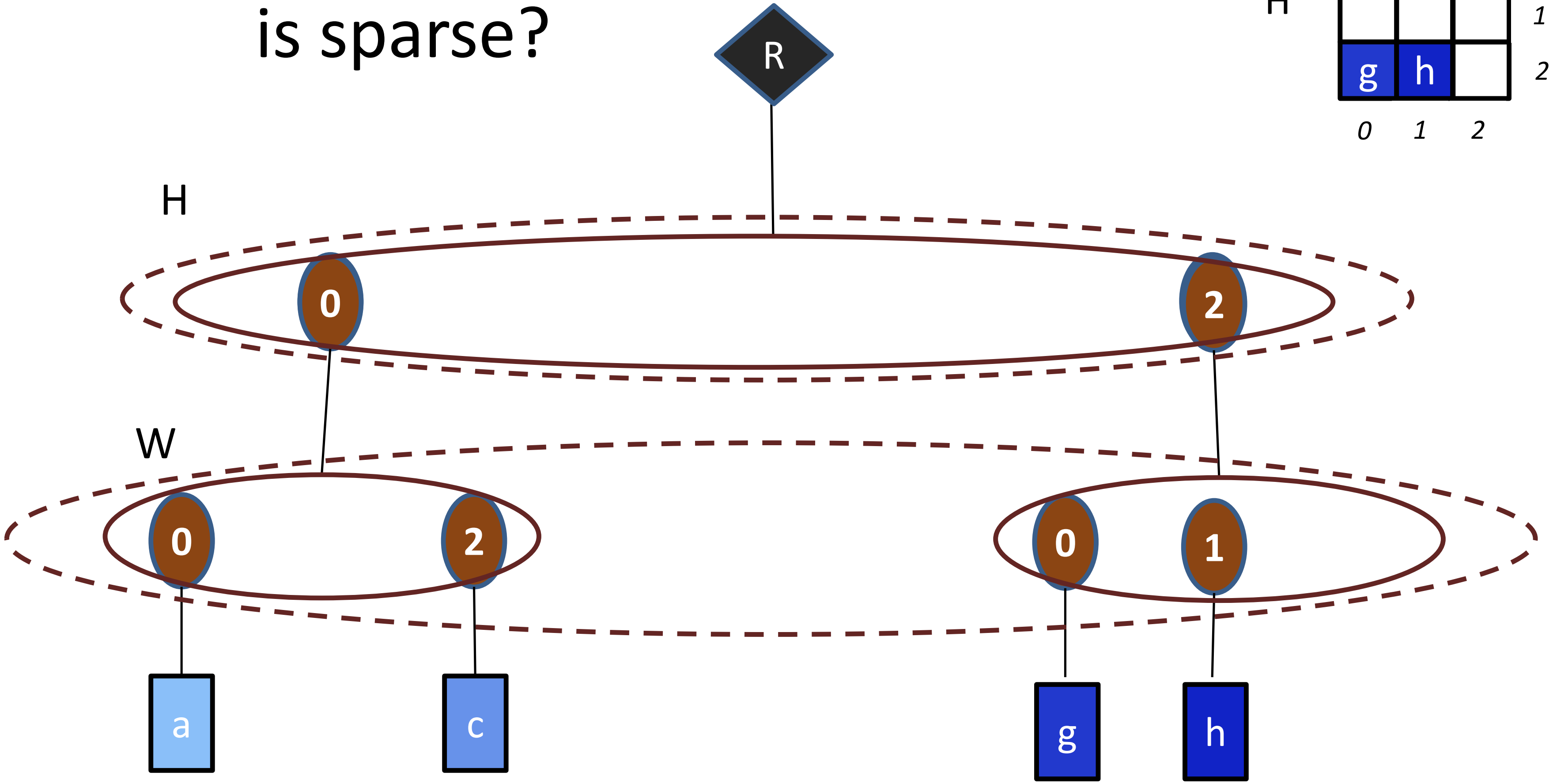
What if tensor is sparse?



# Tree-based Tensor Abstraction

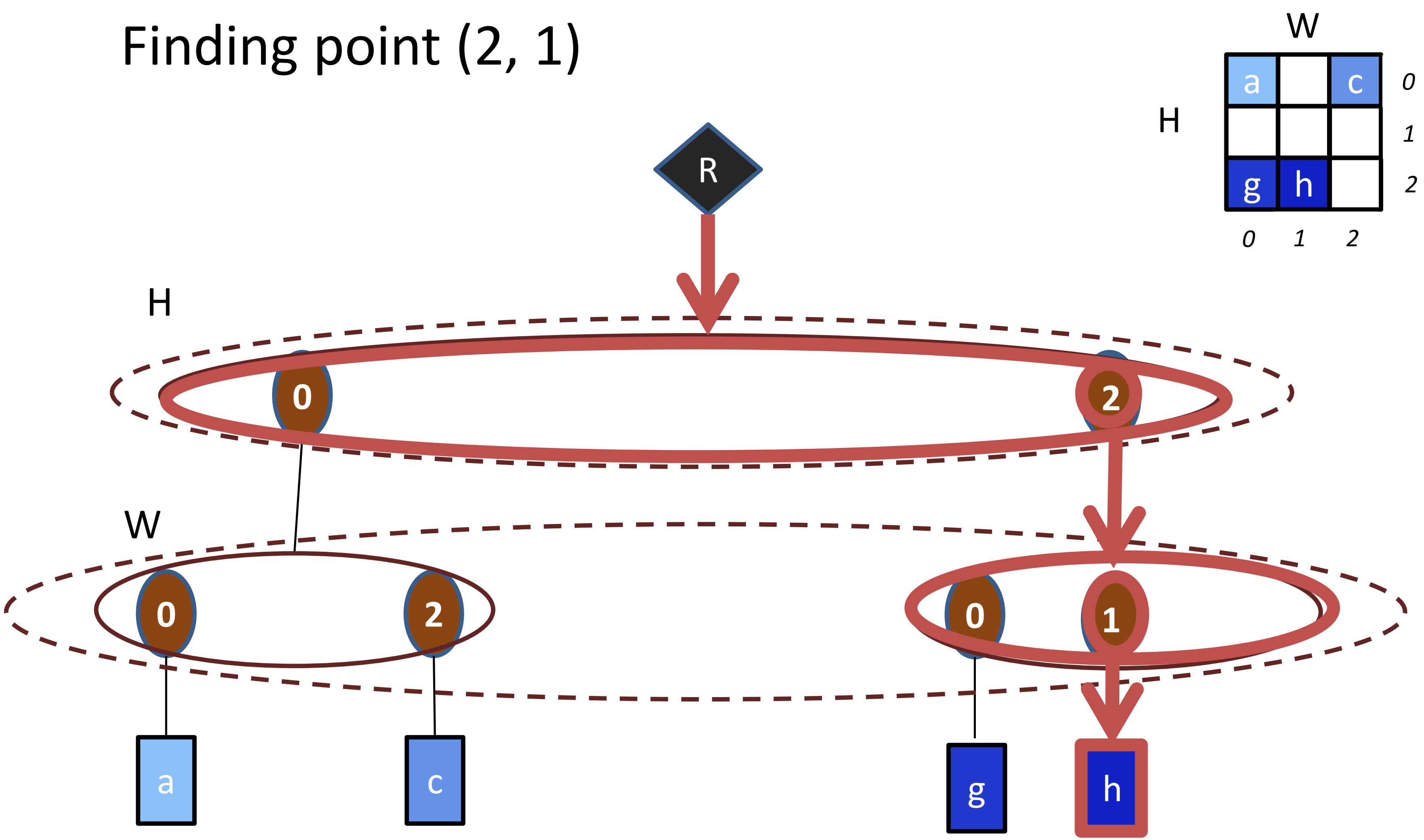
What if tensor is sparse?



# Tree-based Tensor Abstraction

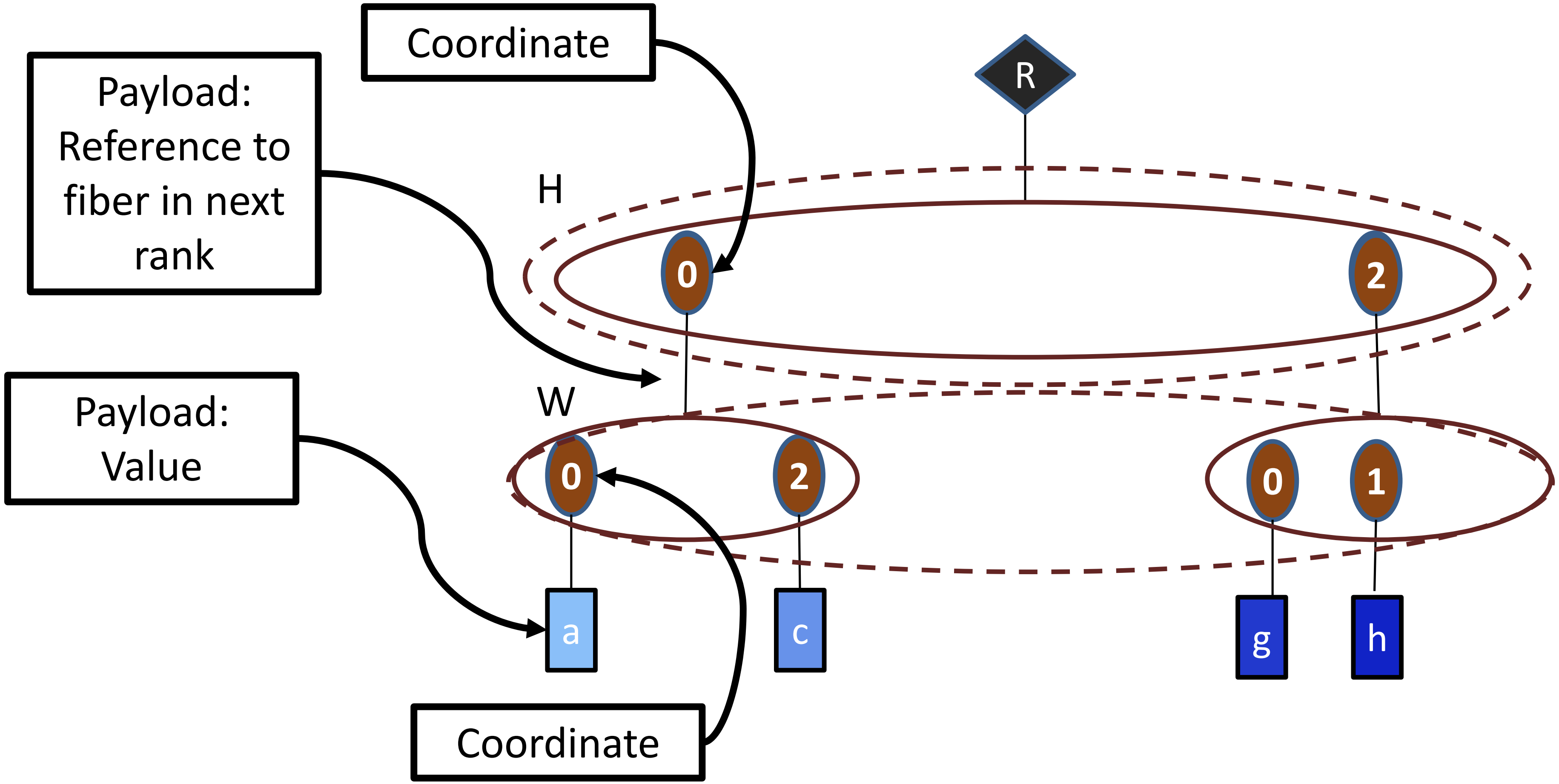
Finding point (2, 1)



# Concrete Fiber  Implementations

# Information in a Fiber

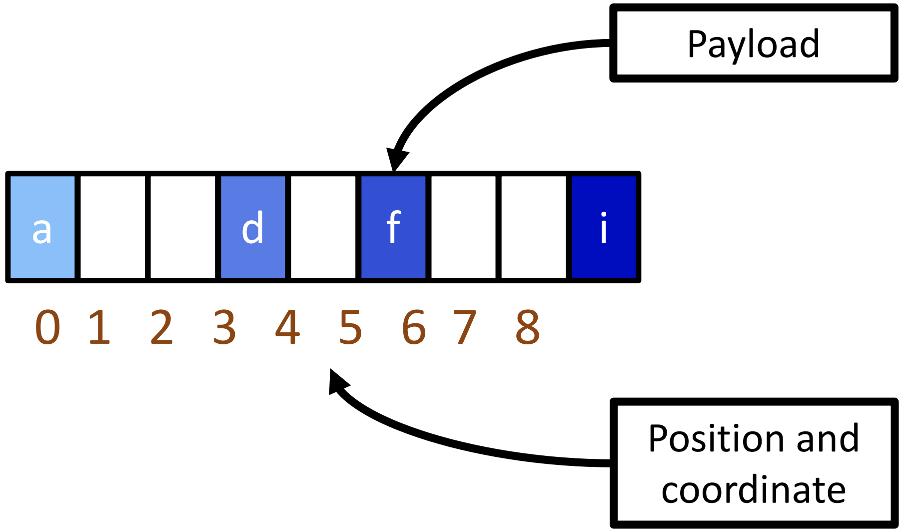
- Each fiber has a set of (coordinate, “payload”) tuples



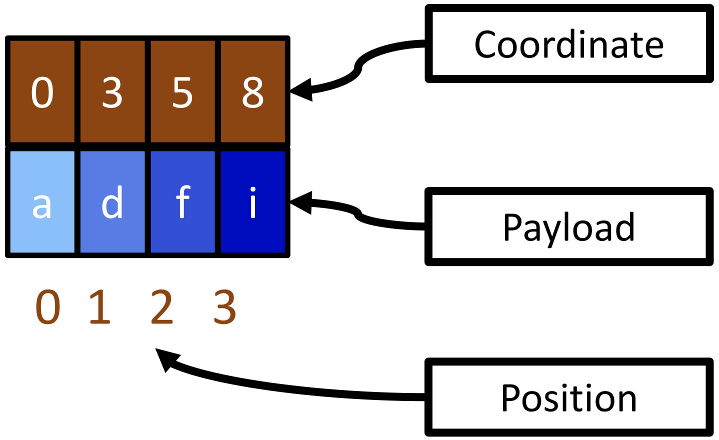
# Example Fiber Representations

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 coding
    - (flag to indicate if non-zero, non-zero payload)…

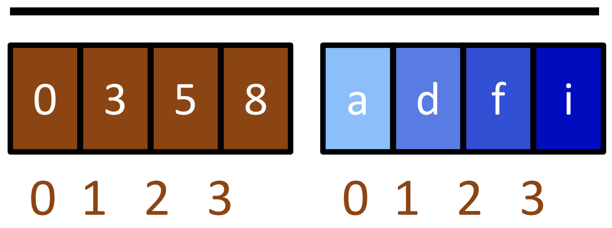
- Payload encoding
  - Fixed length payload
  - Variable length payload
    - E..g., Huffman coding

# Compressed Explicit Coordinate Representations

- Coordinate list representation
  - Struct of arrays form

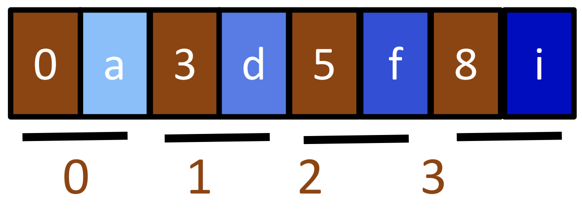
    (coordinate of non-zero value)…

    (non-zero payload)…




  - Array of structs form

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

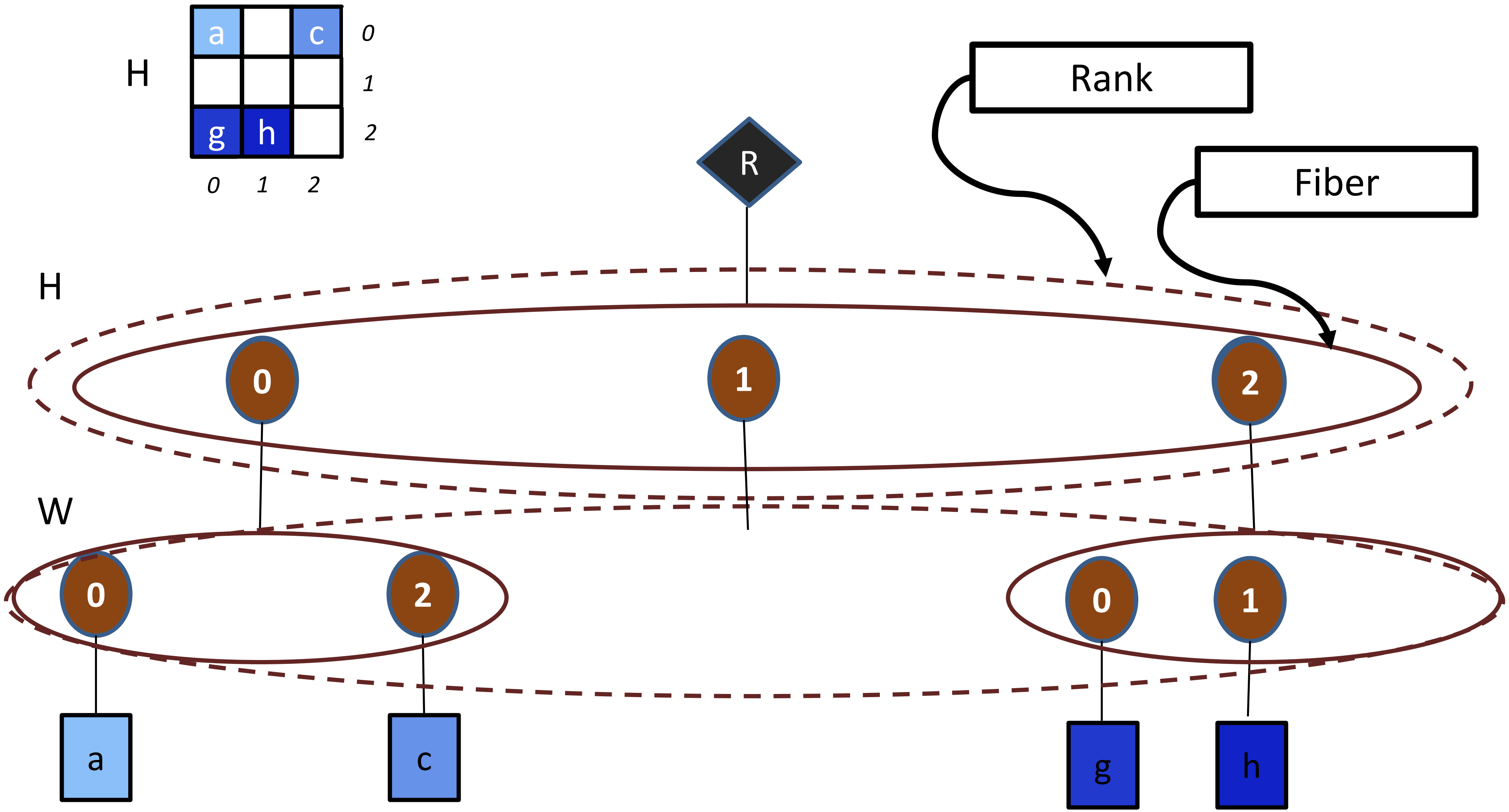



Black bar show scope of struct

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

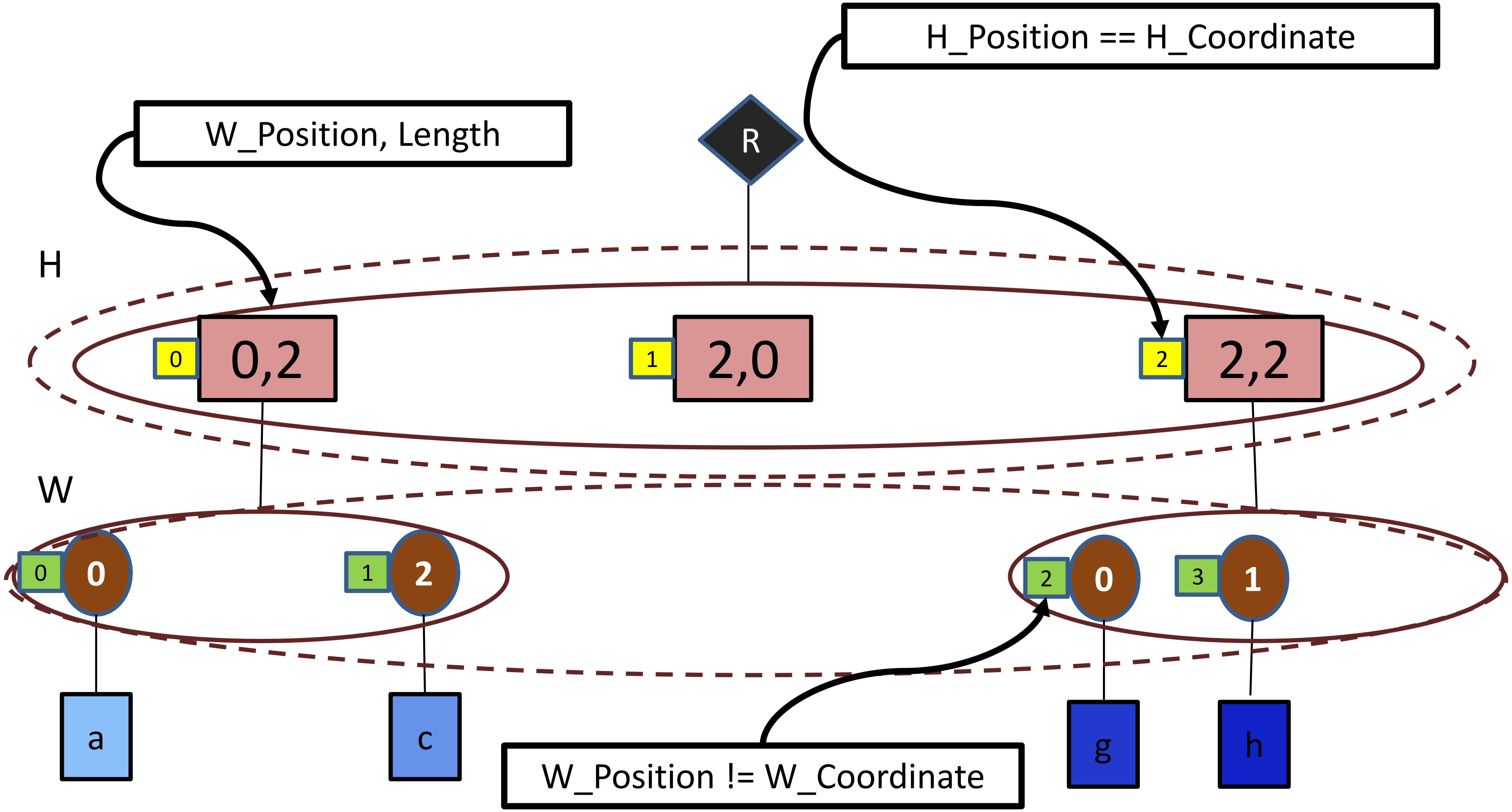
# Uncompressed/Compressed Representation

A specific implementation of the fibertree abstract type



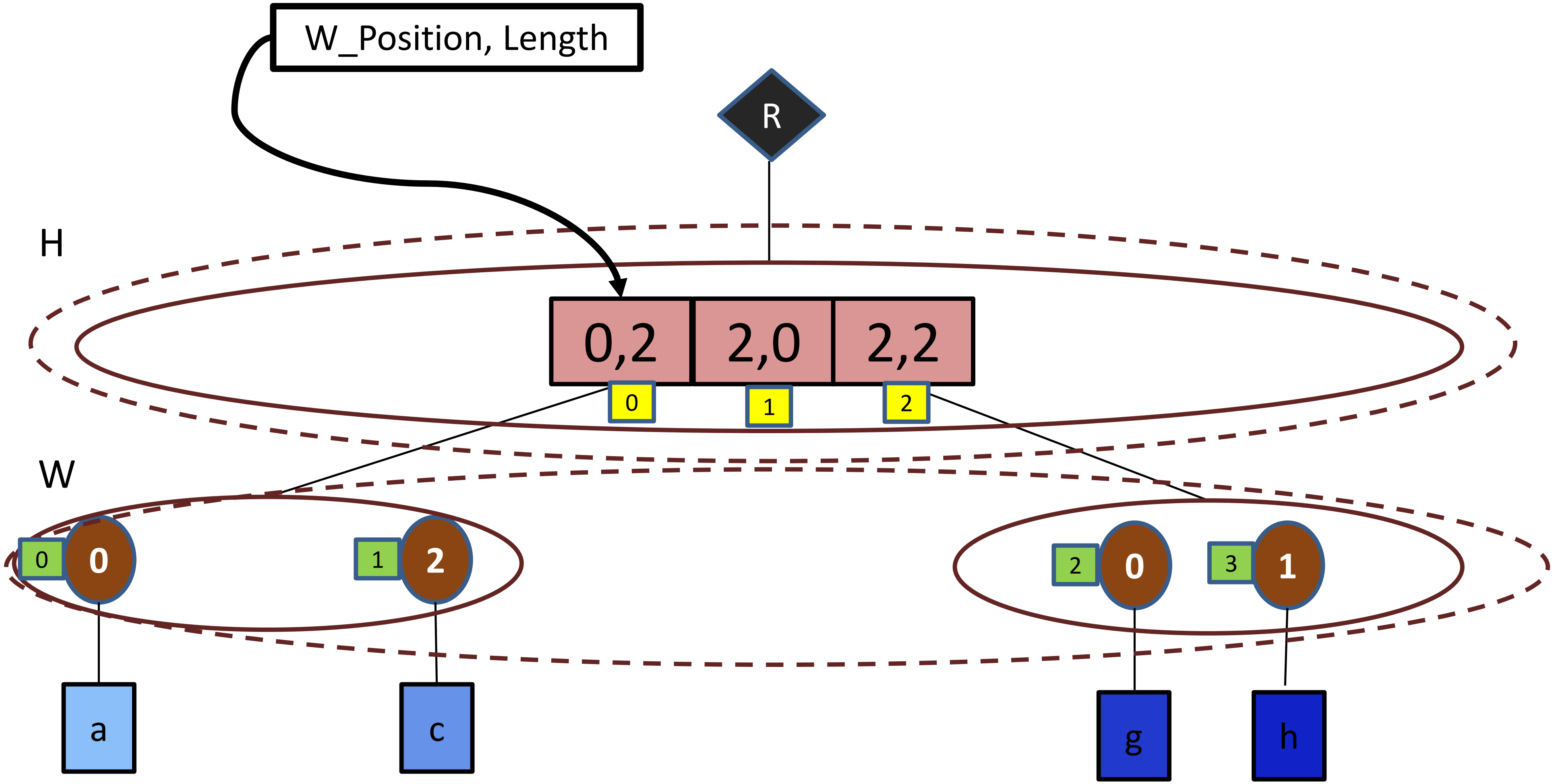
# Uncompressed/Compressed Representation

A specific implementation of the fiber-tree abstract type



# Uncompressed/Compressed Representation

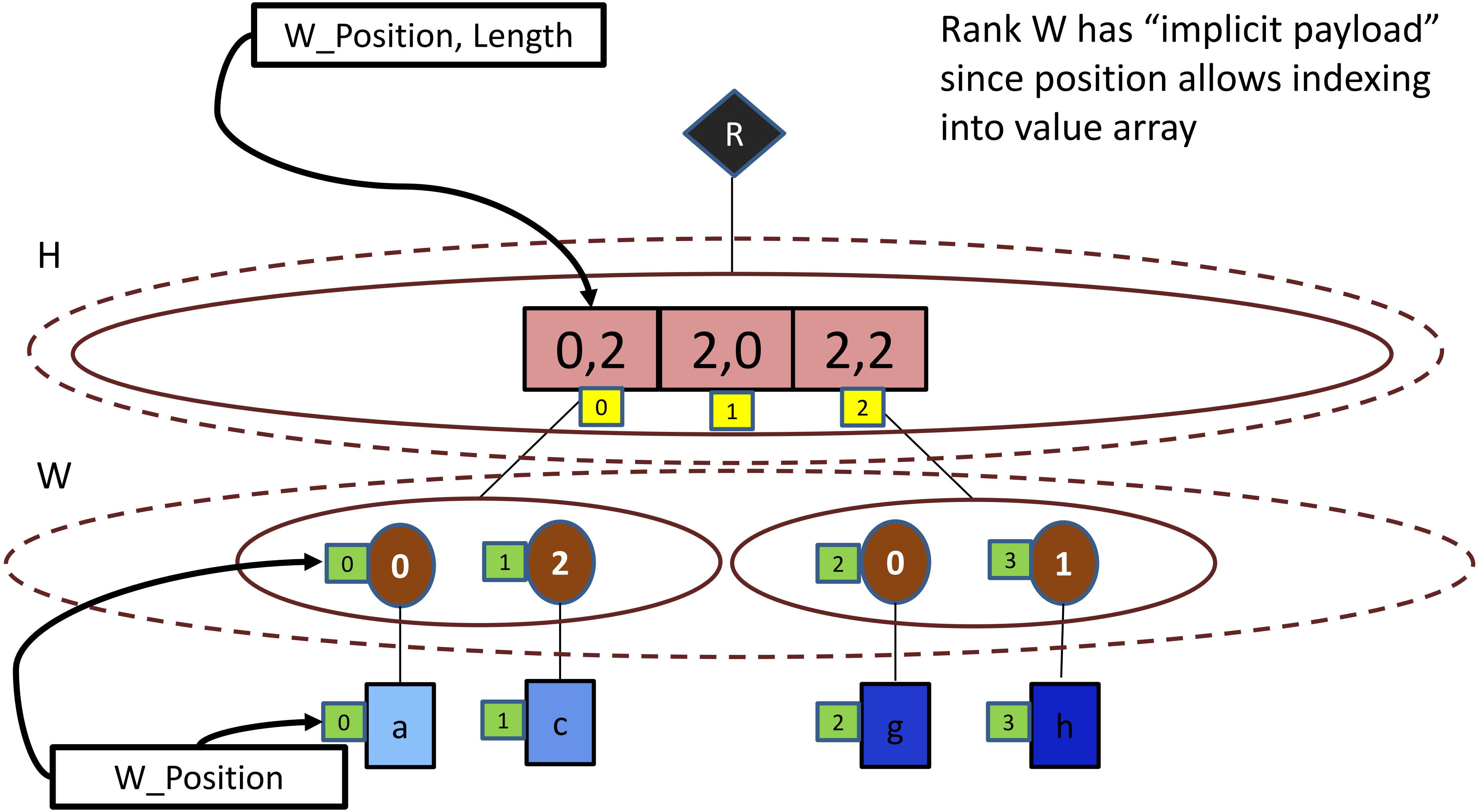
A specific implementation of the fiber-tree abstract type



# Uncompressed/Compressed Representation

A specific implementation of the fiber-tree abstract type

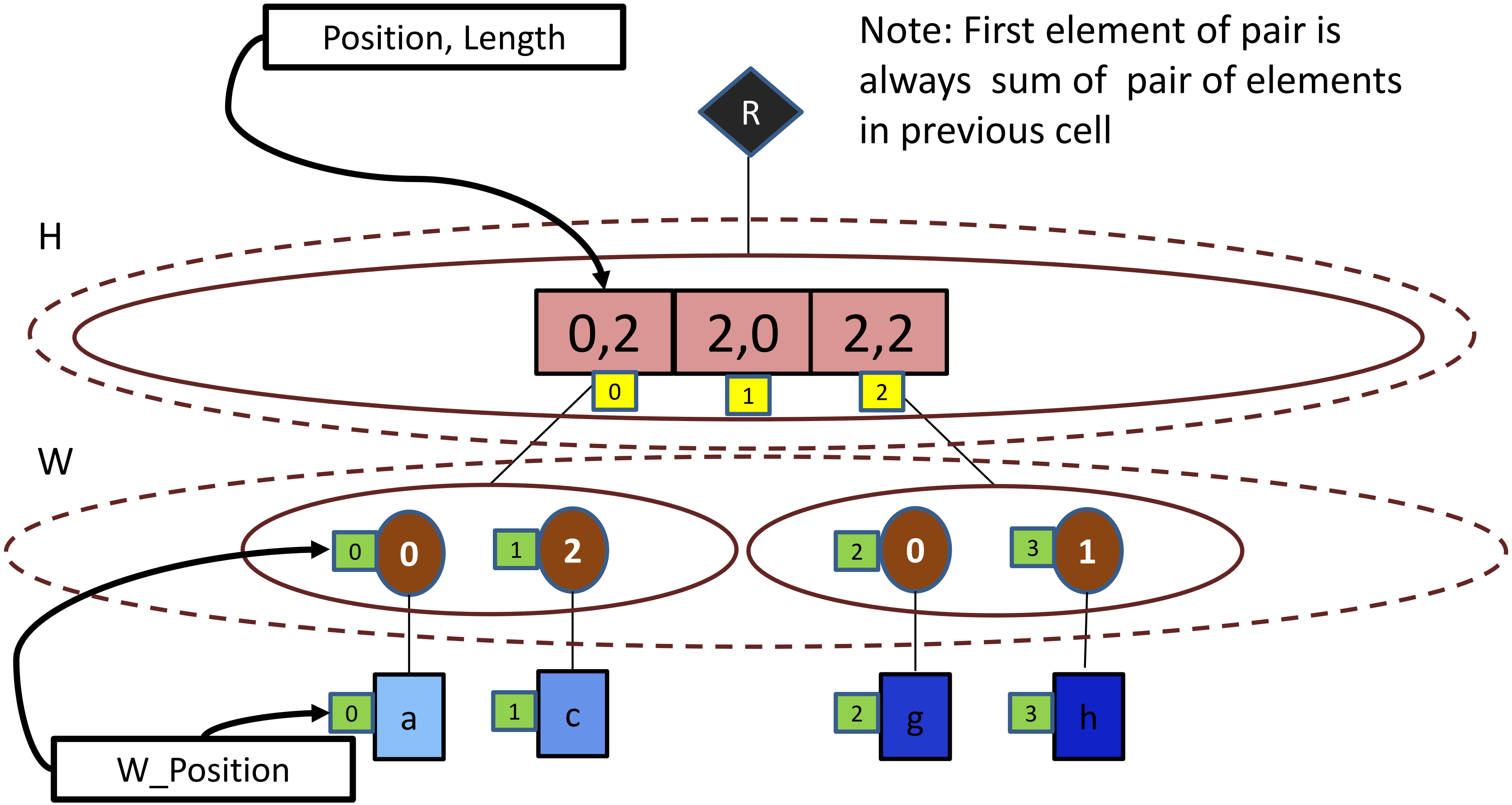
Rank W has “implicit payload” since position allows indexing into value array



# Uncompressed/Compressed Representation

A specific implementation of the fiber-tree abstract type

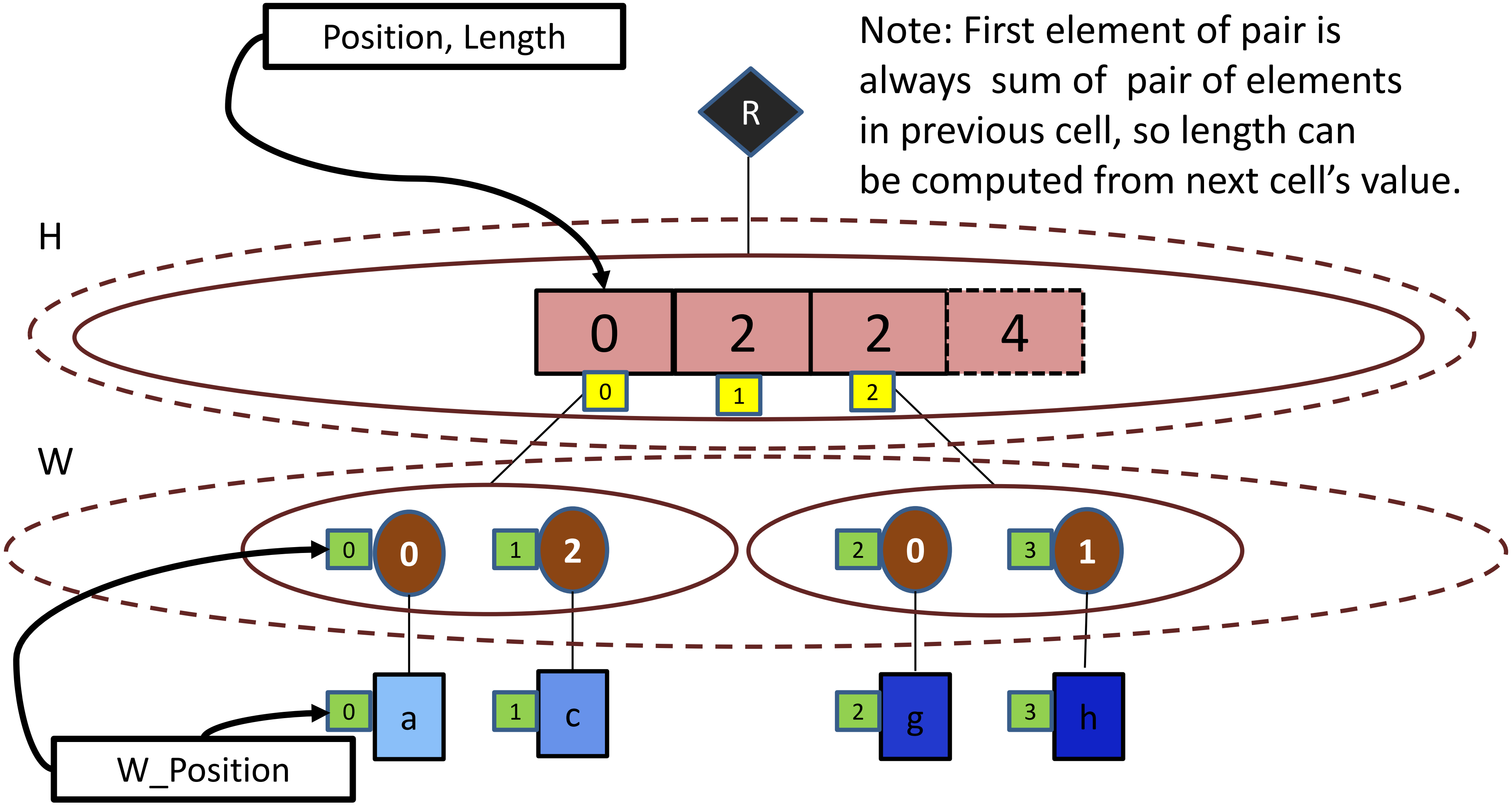
Note: First element of pair is always sum of pair of elements in previous cell



# Uncompressed/Compressed Representation

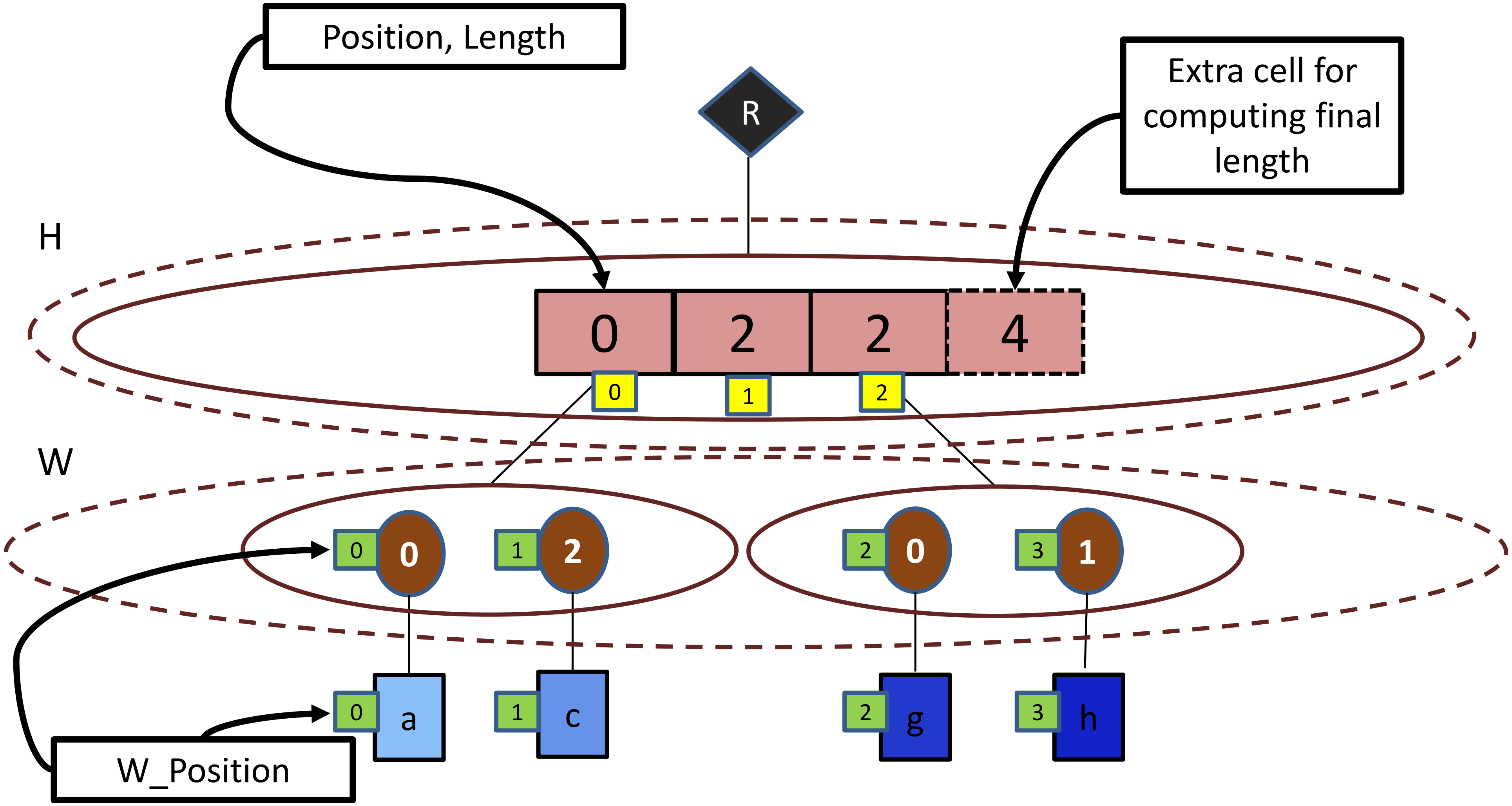
A specific implementation of the fiber-tree abstract type

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

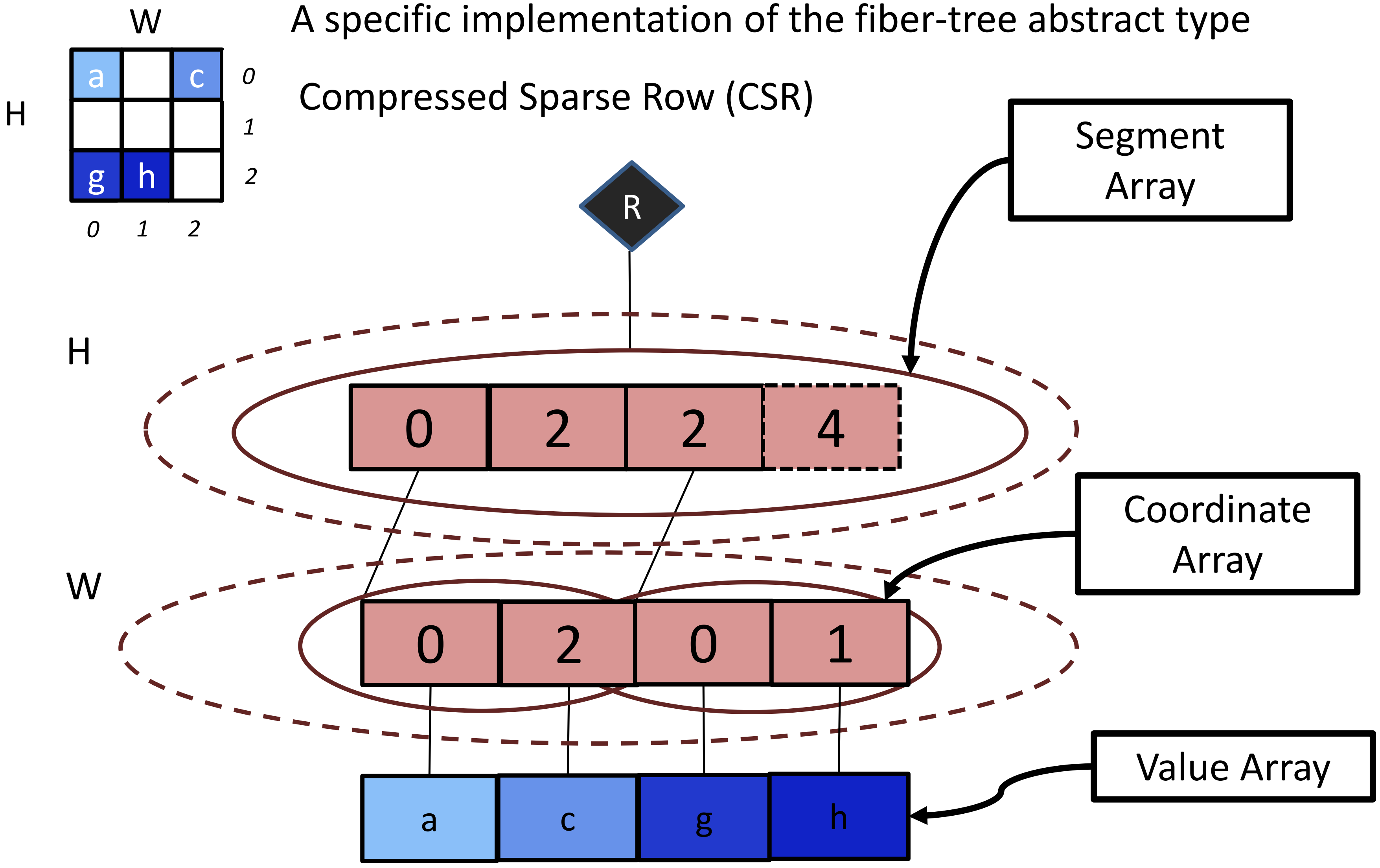
A specific implementation of the fiber-tree abstract type



# Uncompressed/Compressed Representation

A specific implementation of the fiber-tree abstract type

Compressed Sparse Row (CSR)



# Explicit Coordinate Representations

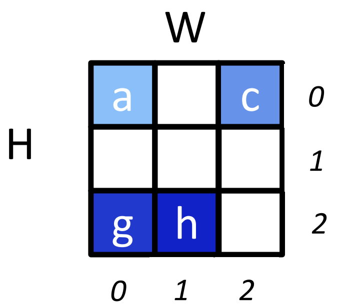
- Coordinate/Payload list
  - (coordinate, non-zero payload)… (array of structs)
  - (coordinate)… , (non-zero payload)… (struct of arrays)
- Hash table (per fiber)
  - (coordinate -> payload) mapping
- Hash table (per rank)
  - (fiber_id, coordinate -> payload) mapping
- Bit vector of non-zero coordinates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

# Per Rank Tensor Representations

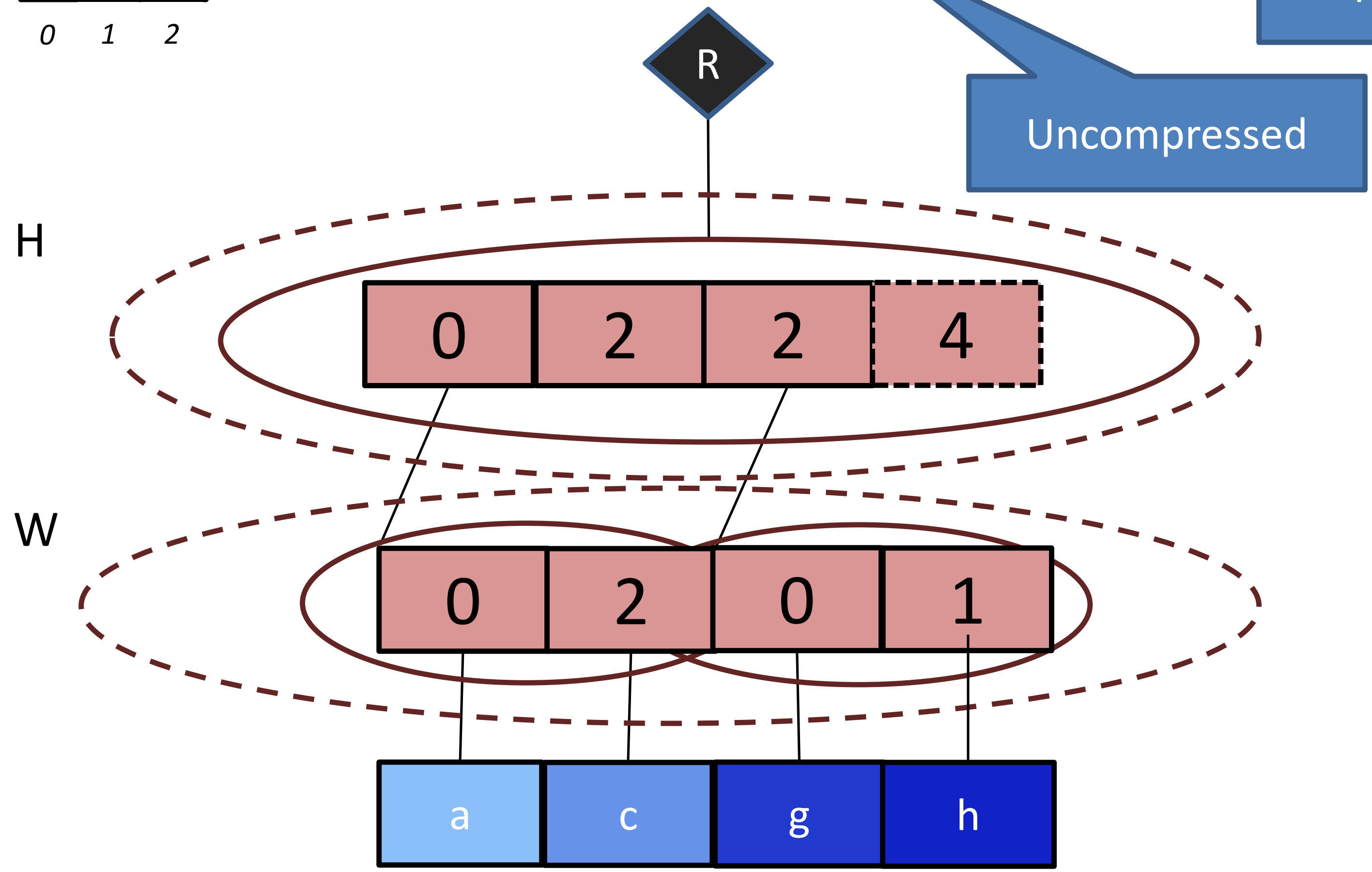
- Uncompressed [U]
  - 
- Run-length Encoded [R]
  - 
- Coordinate/Payload List [C]
  - 
- Hash Table (per rank) [$H_r$]
- Hash Table (per fiber) [$H_f$]
- Tagged union of any combination of previous types

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

# Notation for CSR



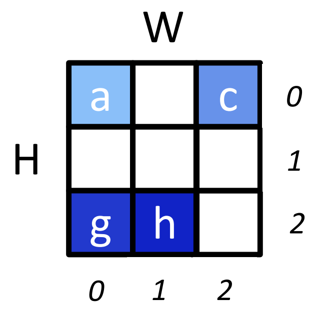

CSR: Tensor<U,C>(H,W)



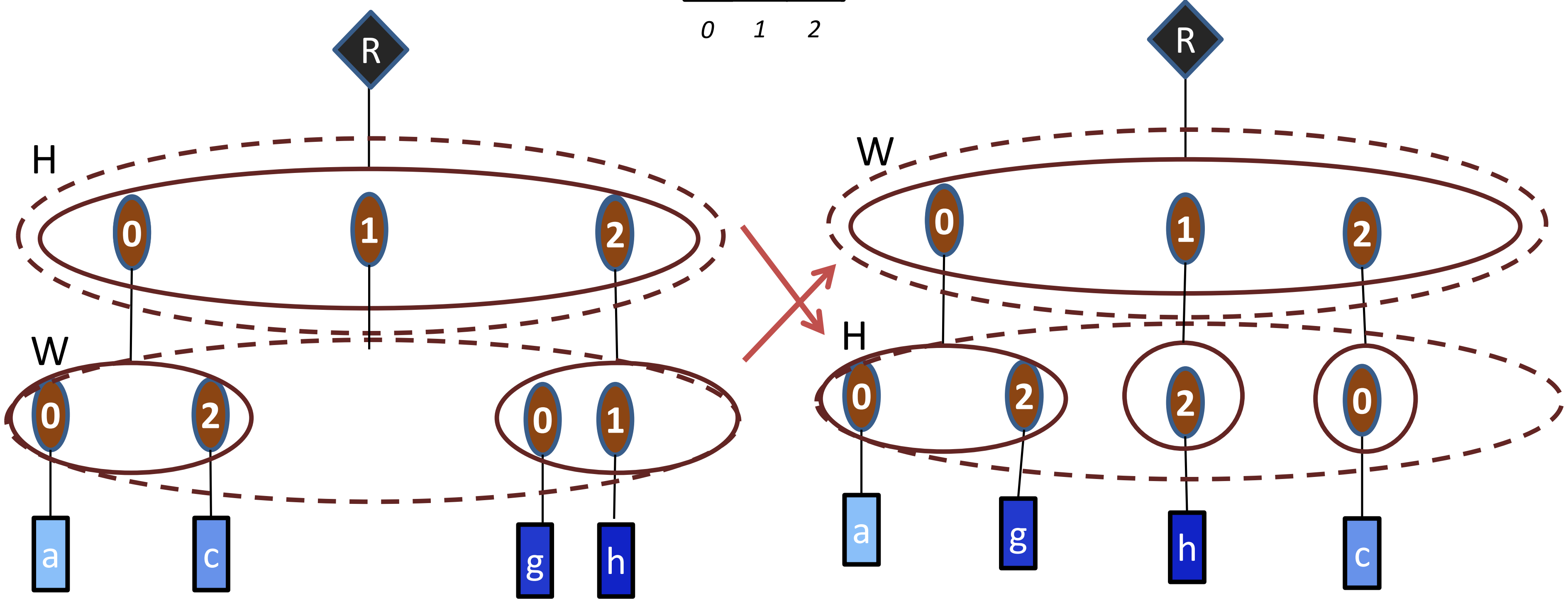
# Representation of Order of Ranks

## Differentiating CSR and CSC

Tensor<UC>(HW) → CSR

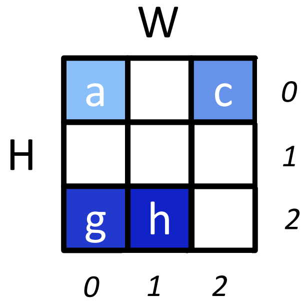



Tensor<UC>(WH) → CSC

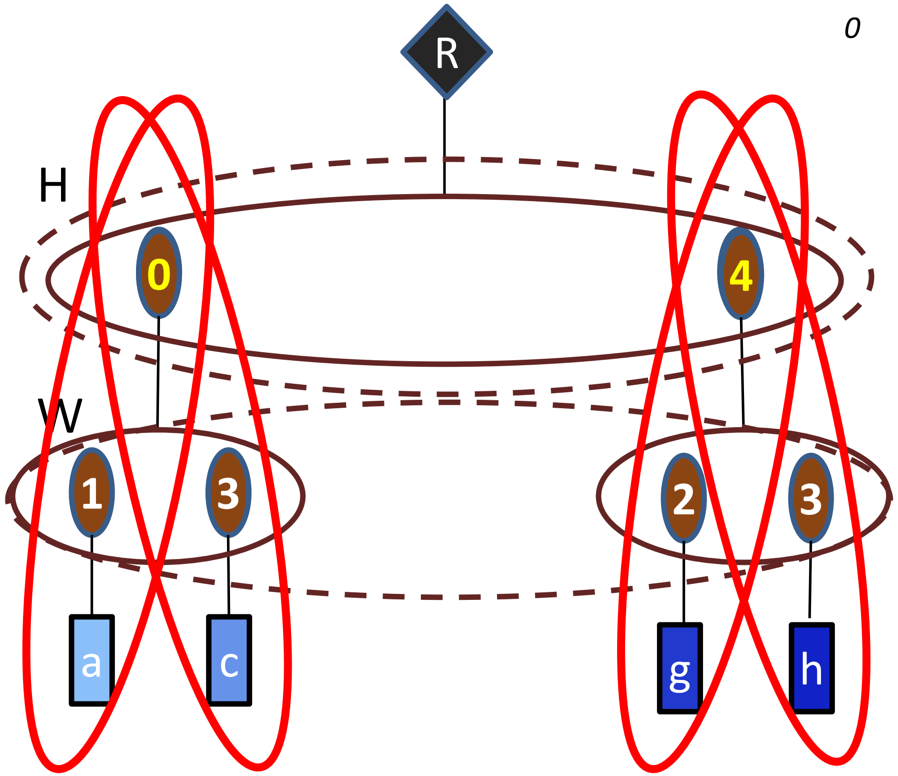


# Merging Ranks

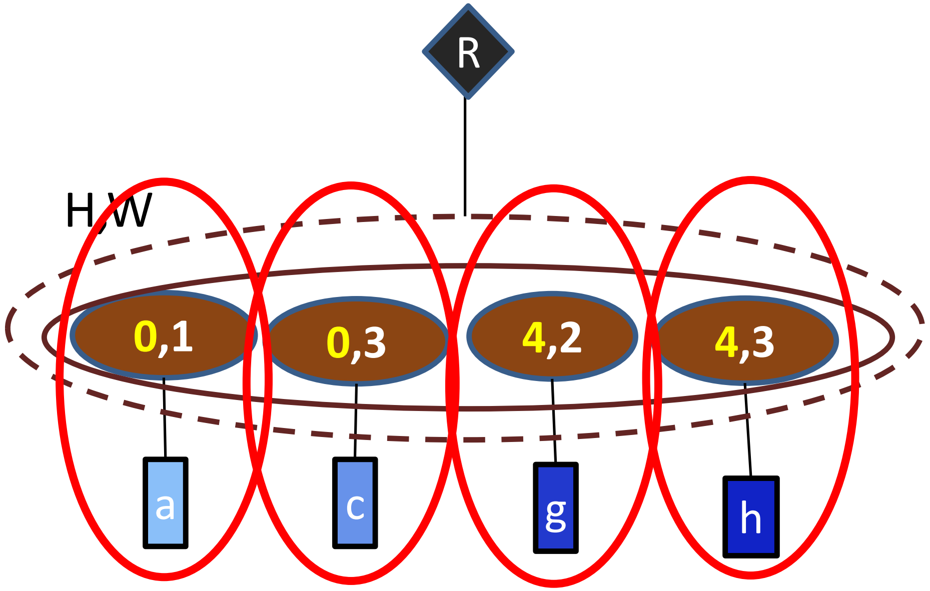
For efficiency one can form new representations where the data structure for two or more ranks are combined.




Tensor-<CC>(HW)




Tensor-<$C^2$>(HW)



# 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

# Traversal Efficiency

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

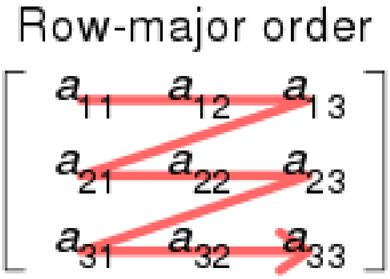
- Operations:
  - maybe(payload) = Fiber.getPayload(coordinate)
  - (coordinate, payload) = Fiber.getNext(rank_traversal_order)

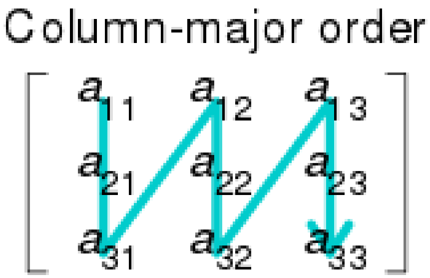
Fiber.next() is a useful iterator and its efficiency is highly dependent on representation, both order of ranks and representation of each rank….

# Concordant traversal orders

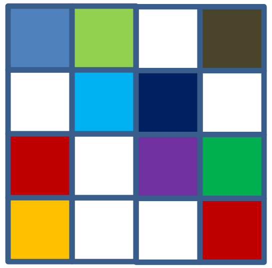
CSR and CSC each has a natural (or "concordant"*) traversal order

Processing Order

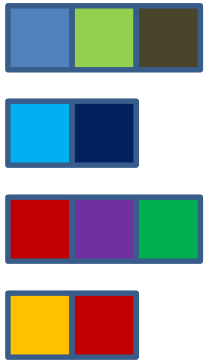


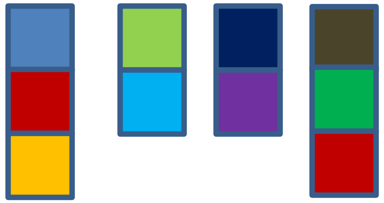

Original Matrix



Compressed Sparse Row (CSR)



Compressed Sparse Column (CSC)



* Term from Michael Pellauer

# 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 payload-method….

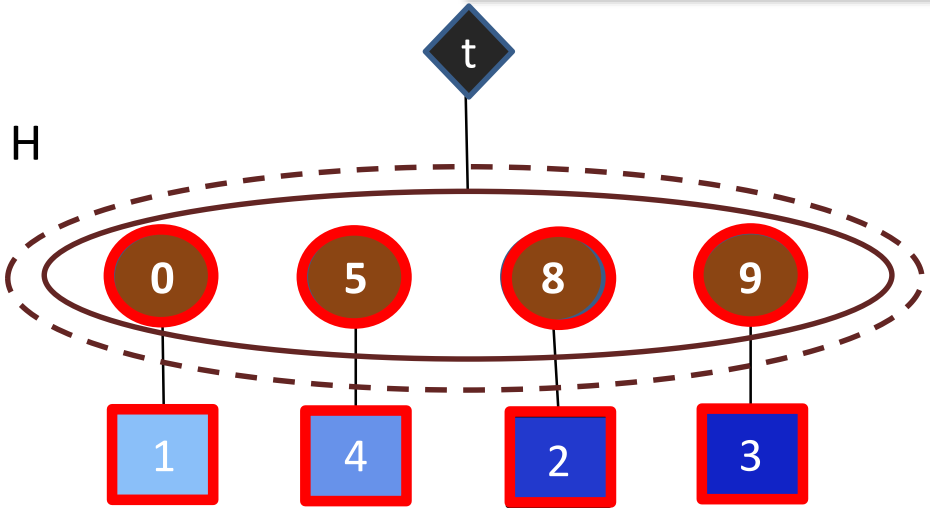
# Tensor Traversal Scheduling

# Traversing a Sparse Tensor

```
# 1-D Tensor Traversal

t = Tensor(H)

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

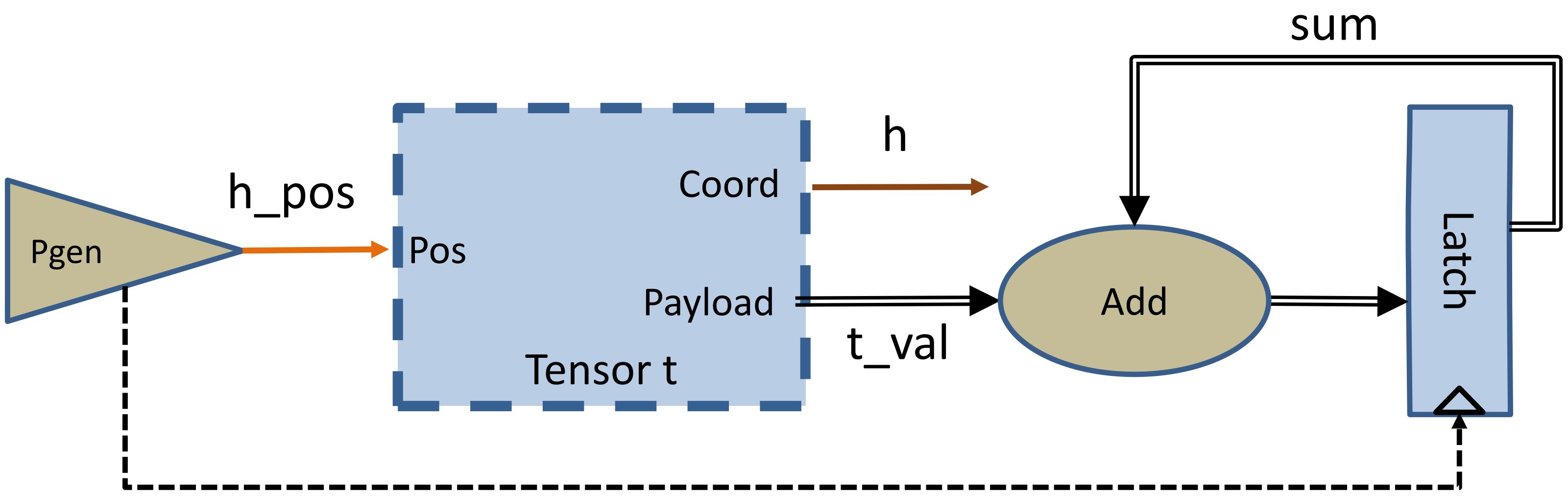
Each iteration returns a (coordinate, payload) tuple

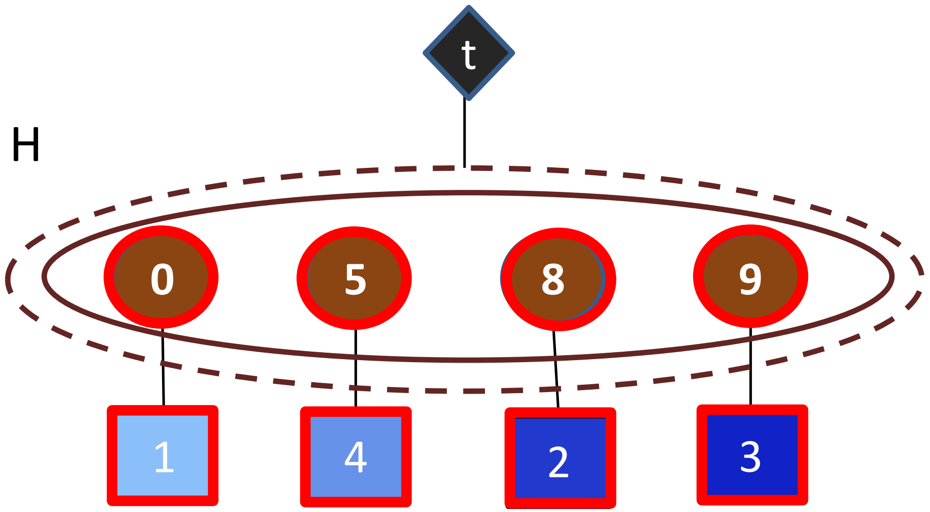
Iteration generated by repeated calls to getNext()




| t_pos | h | t_val | sum |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 5 | 4 | 5 |
| 2 | 8 | 2 | 7 |
| 3 | 9 | 3 | 10 |

# Traversing a Sparse Tensor



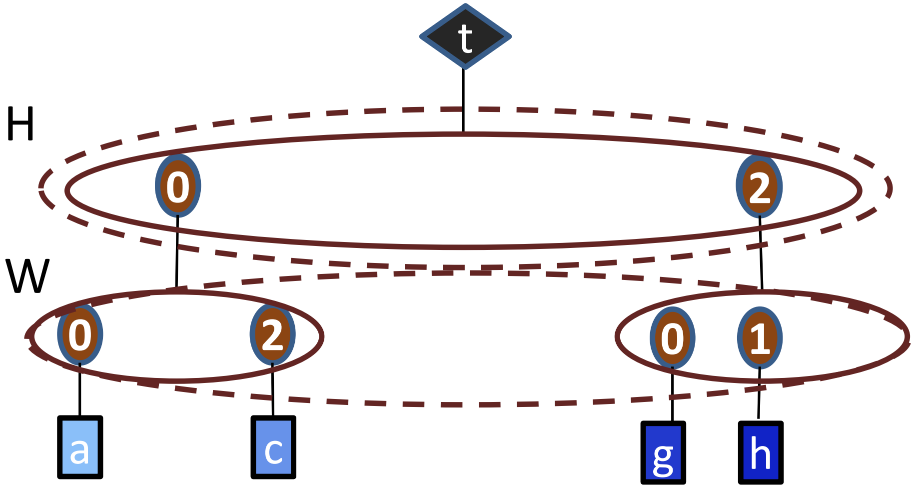




| t_pos | h | t_val | sum |
|---|---|---|---|
| 0 | 0 | 1 | 1 |
| 1 | 5 | 4 | 5 |
| 2 | 8 | 2 | 7 |
| 3 | 9 | 3 | 10 |

# Tensor Traversal (2-D)

```
# 2-D Tensor Traversal

t = Tensor(H,W)

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

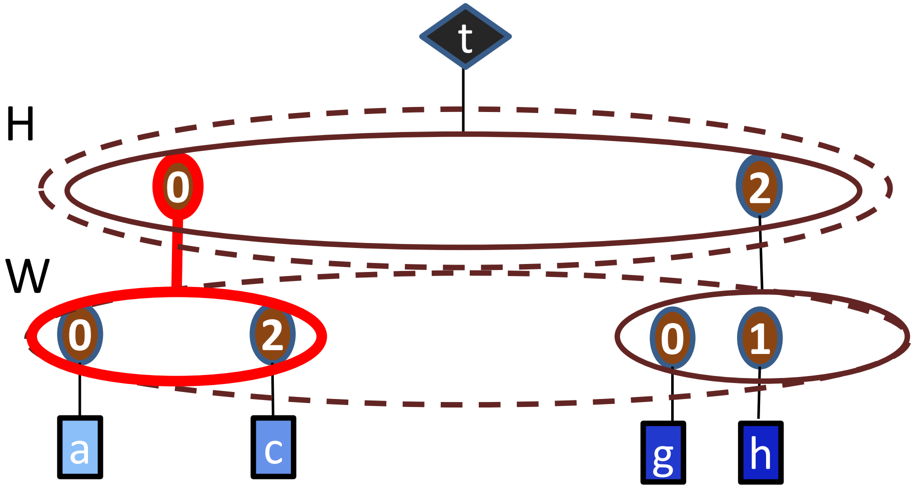
Each iteration returns a (coordinate, payload) tuple




| 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)

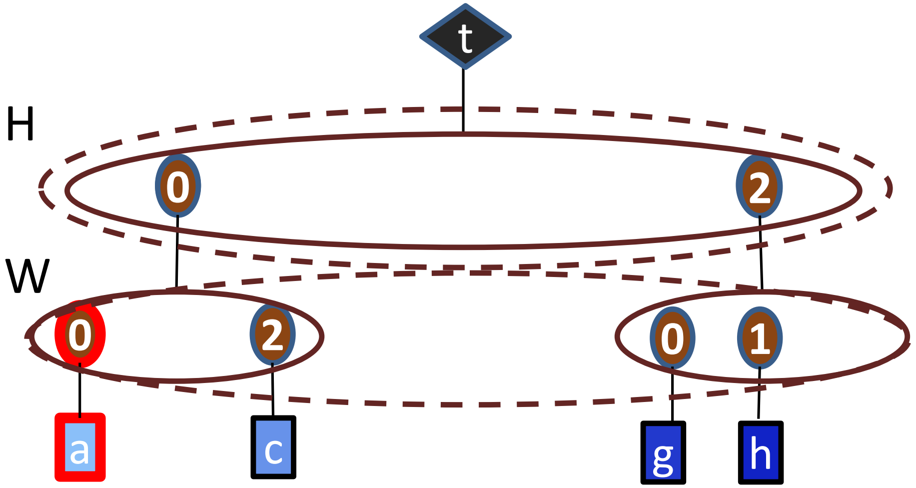
```
# 2-D Tensor Traversal

t = Tensor(H,W)

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

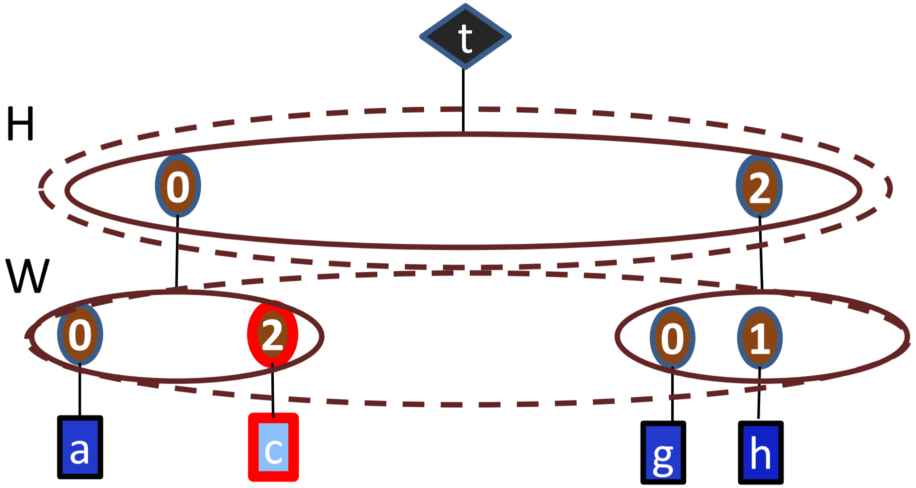
```
# 2-D Tensor Traversal

t = Tensor(H,W)

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

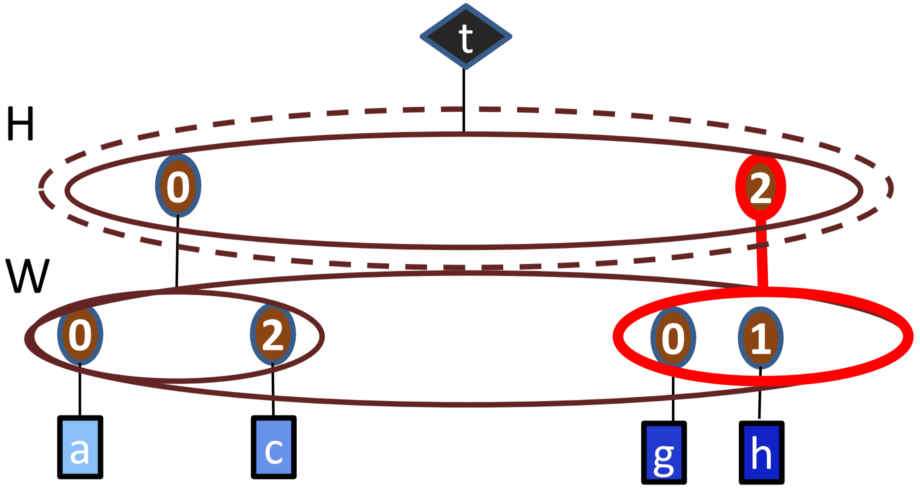
```
# 2-D Tensor Traversal

t = Tensor(H,W)

sum = 0
for (h, t_h) in t:
  for (w, t_val) in t_h:
     sum += 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 | ? | ? | ? |
| … | … | … | … | … |

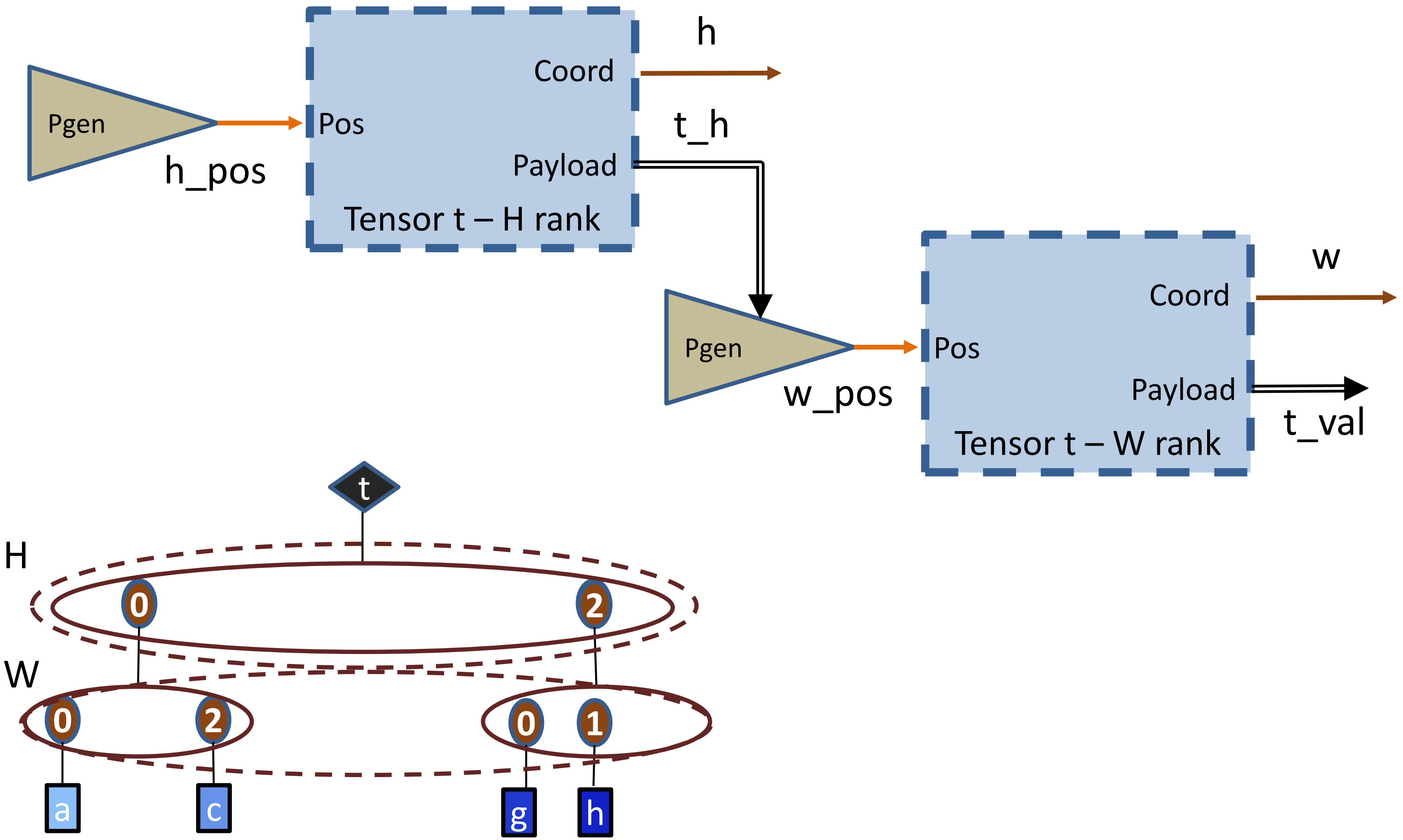
# Tensor Traversal (2-D)

```
# 2-D Tensor Traversal

t = Tensor(H,W)

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



# Tensor Traversal (CSR Style)

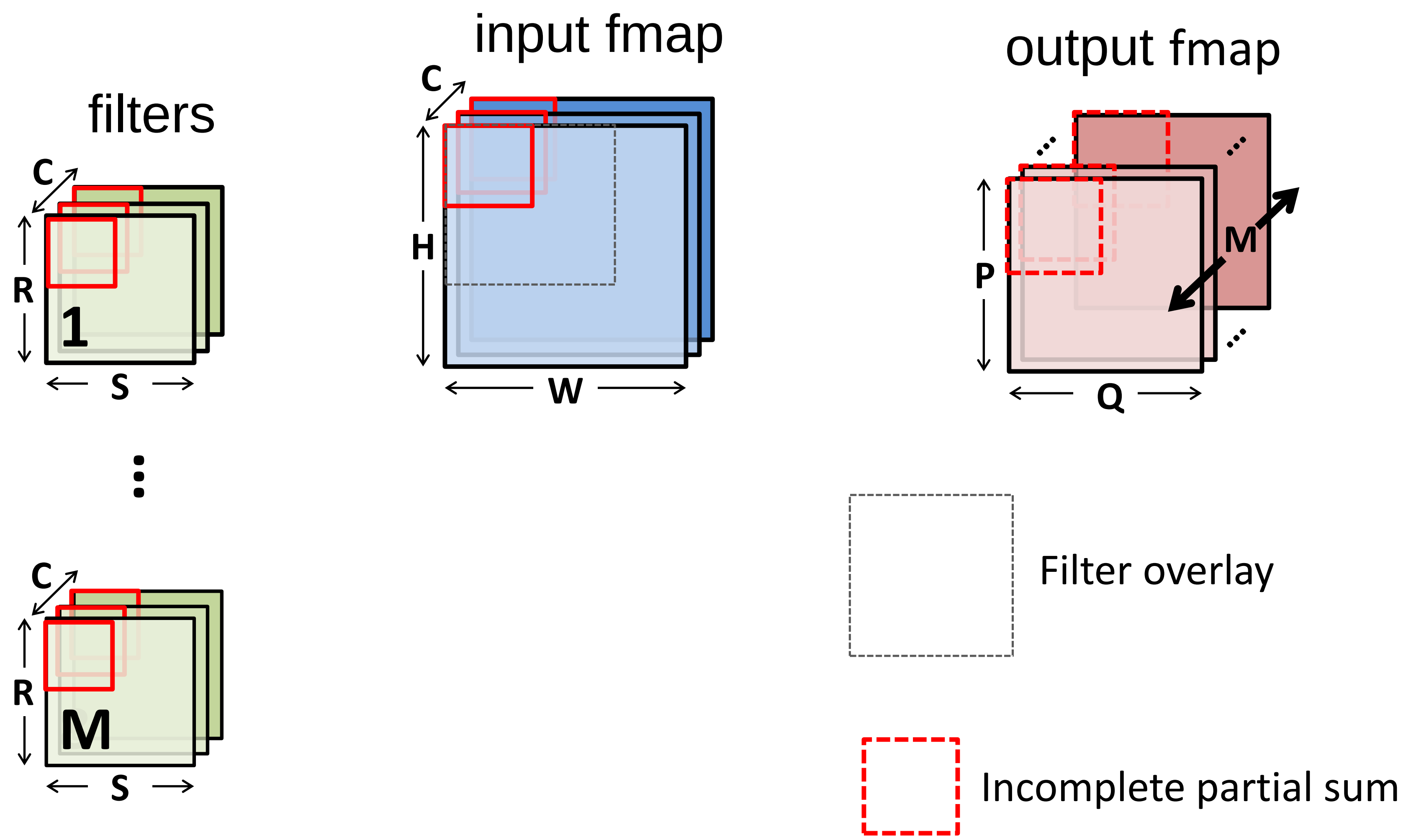
```
# 2-D Tensor Traversal (CSR)

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

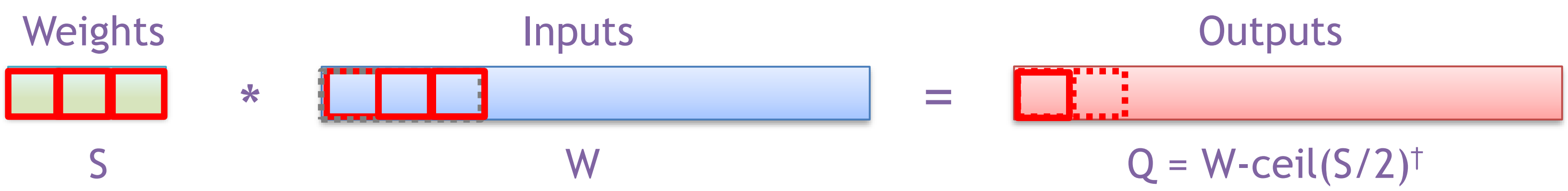
# Sparse Tensor Computations

# CONV Layer



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

# Output Stationary - Uncompressed

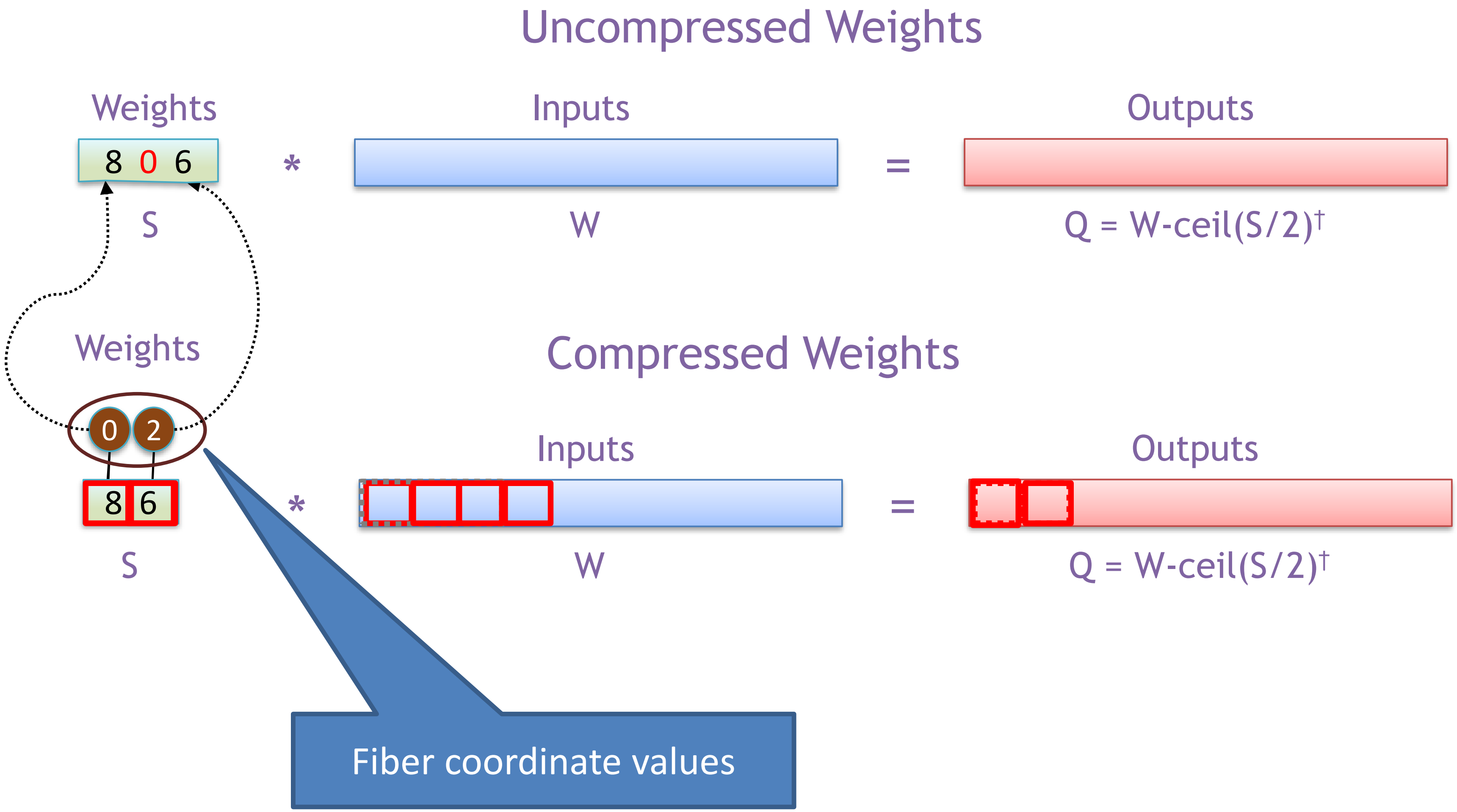



```
i = Array(W)         # Input activations
f = Array(S)         # Filter weights
o = Array(Q)         # Output activations

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

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

Need to calculate position/coordinate in third tensor

# Output Stationary - Sparse Weights




† Assuming: 'valid' style convolution

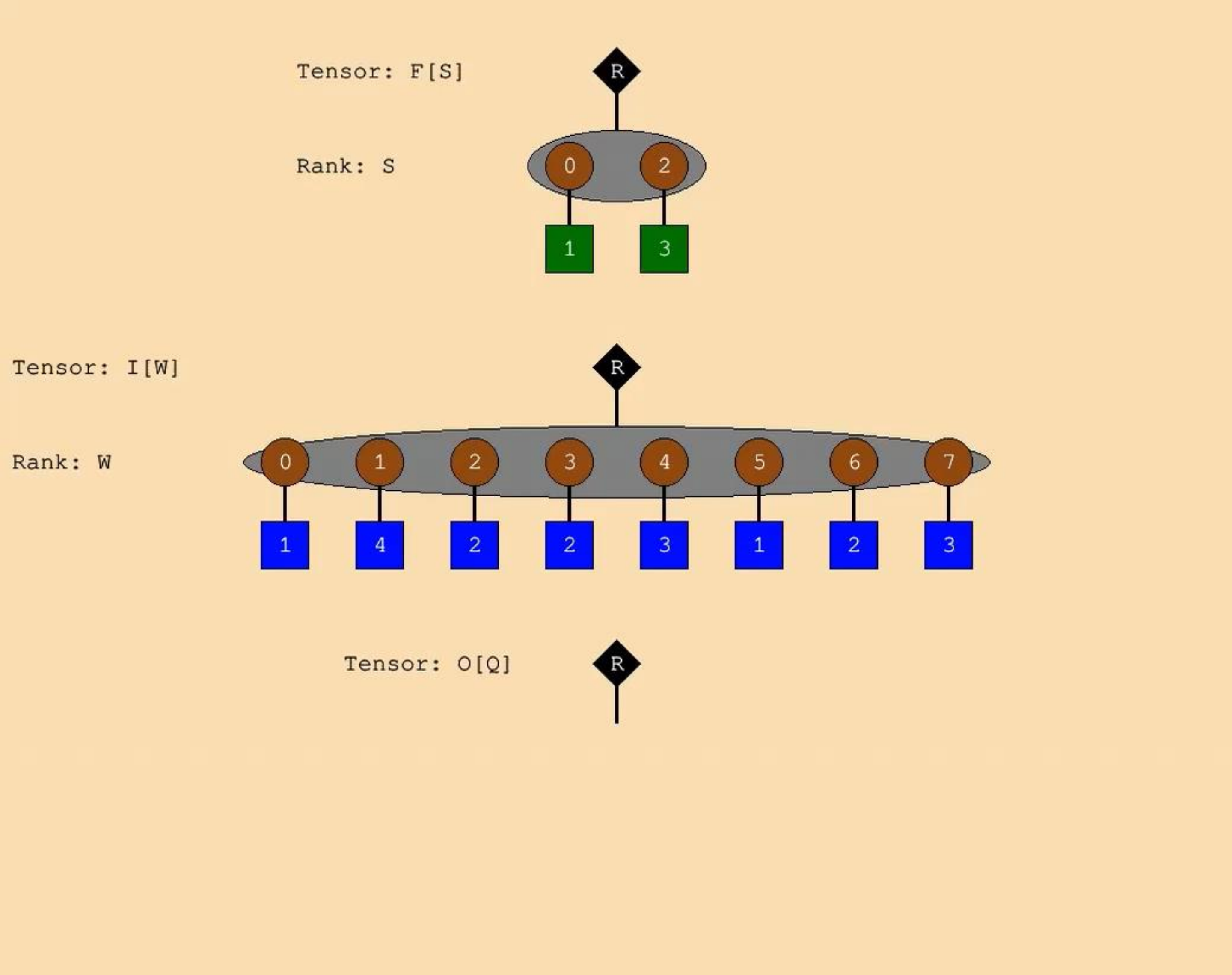
# Output Stationary - Sparse Weights

```
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]
```

```
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

# Output Stationary – Sparse Weights



# 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

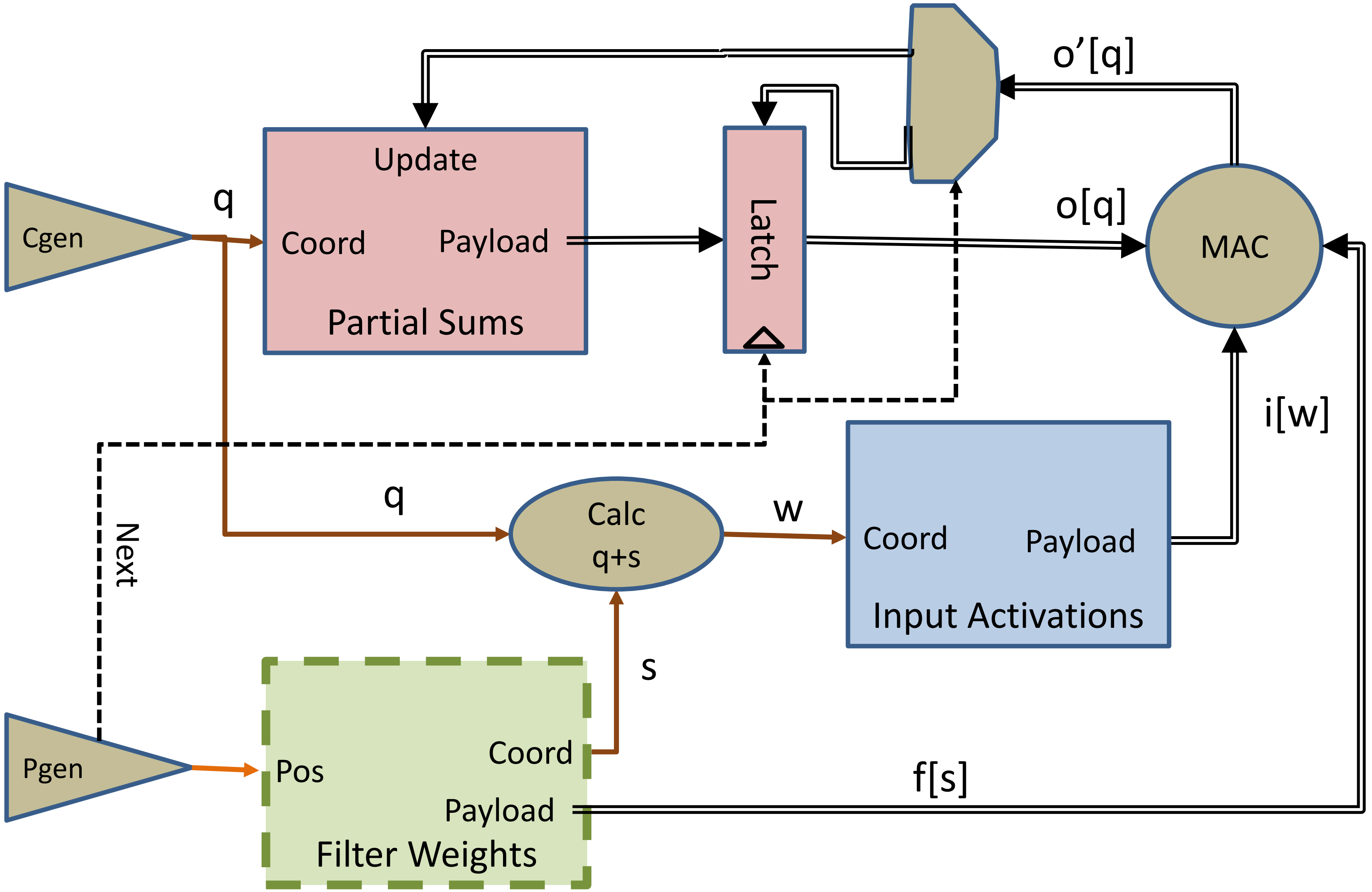
| Rank: W | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 2 | 2 | 2 | 2 | 2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

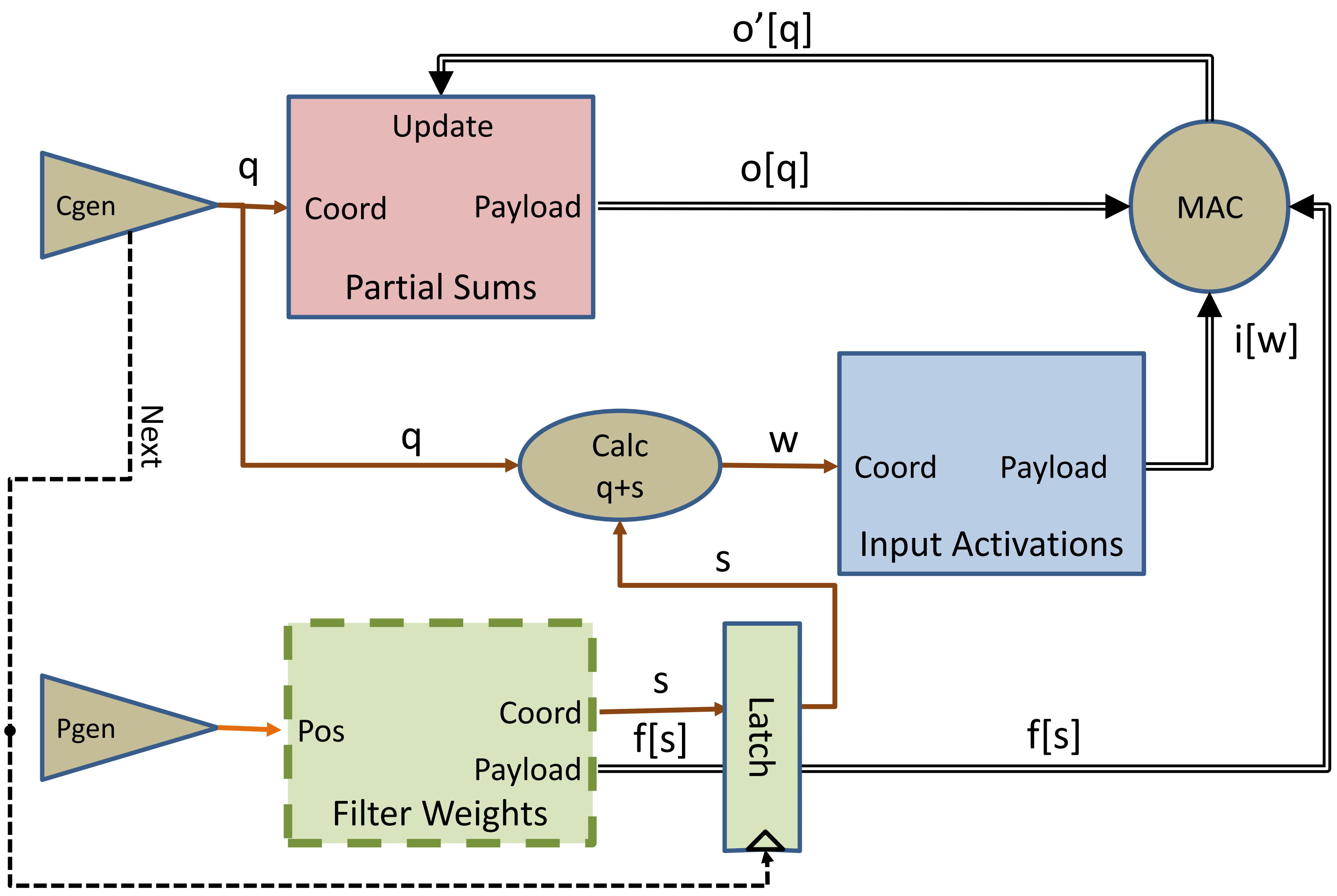


# Weight Stationary - Sparse Weights

```
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

# Weight Stationary - Sparse Weights



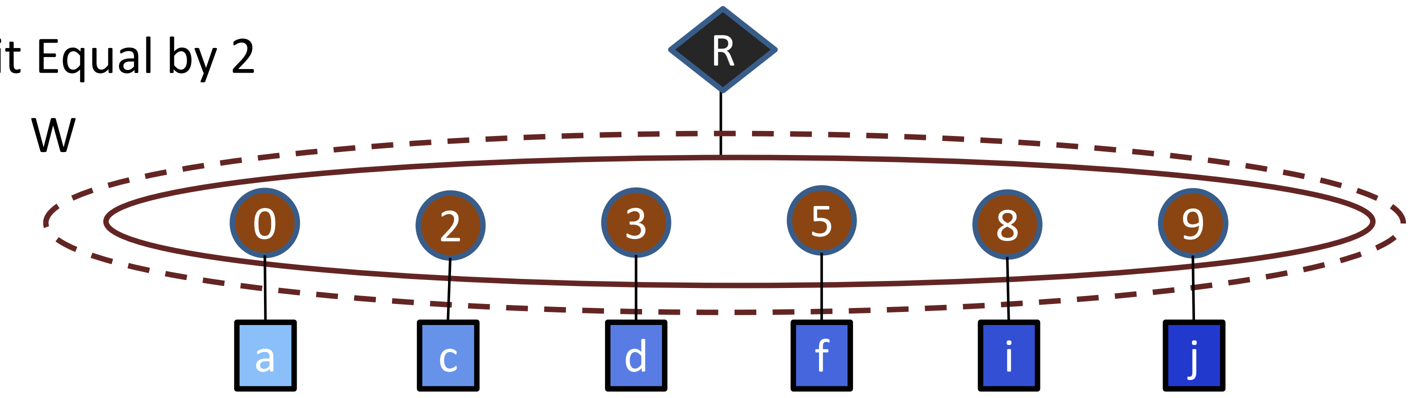
# To Extend to Other Dimensions of DNN

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

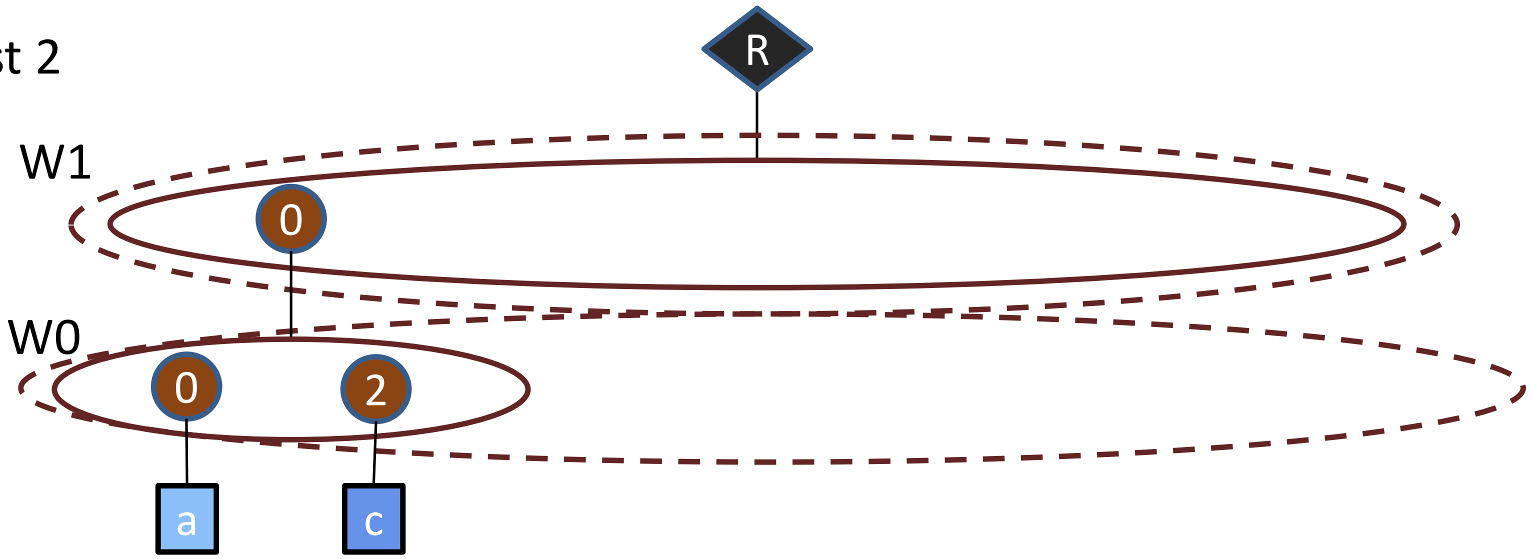
- Add parallelism…

# Fiber Splitting Equally in Position Space

Before Split Equal by 2

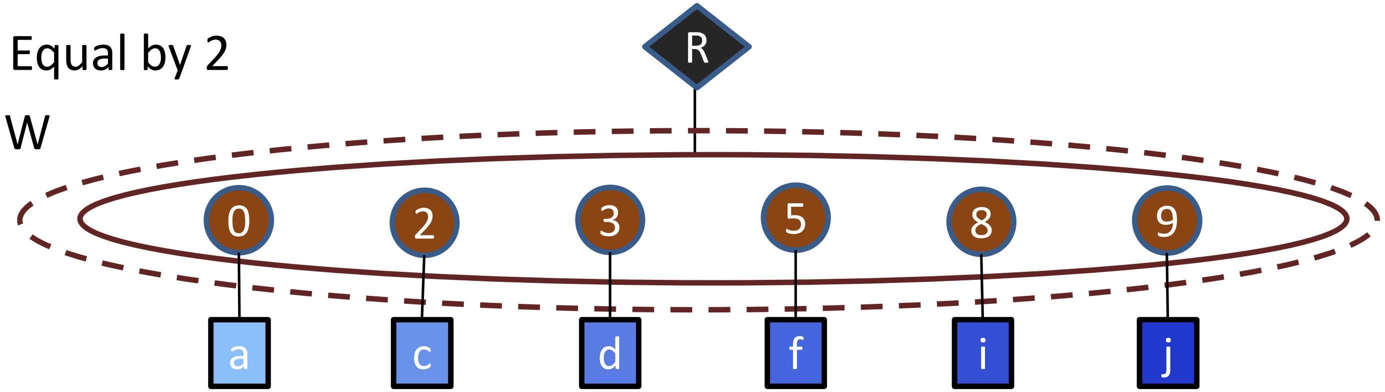



Grab first 2

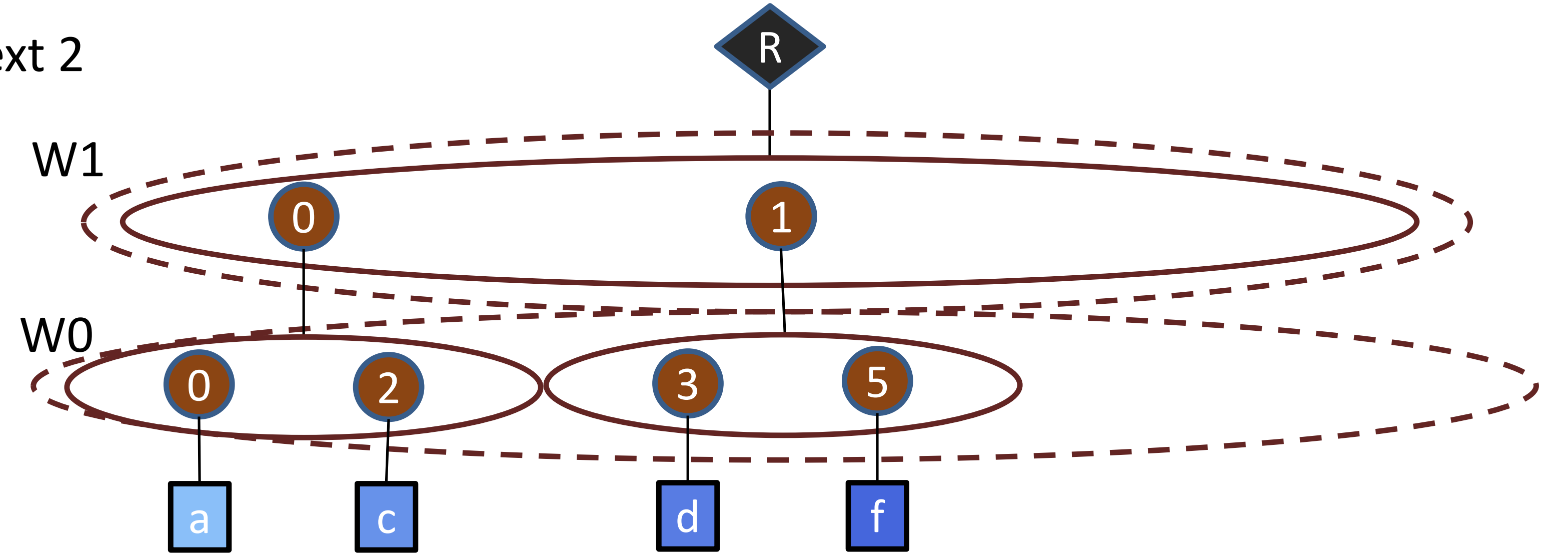


# Fiber Splitting Equally in Position Space

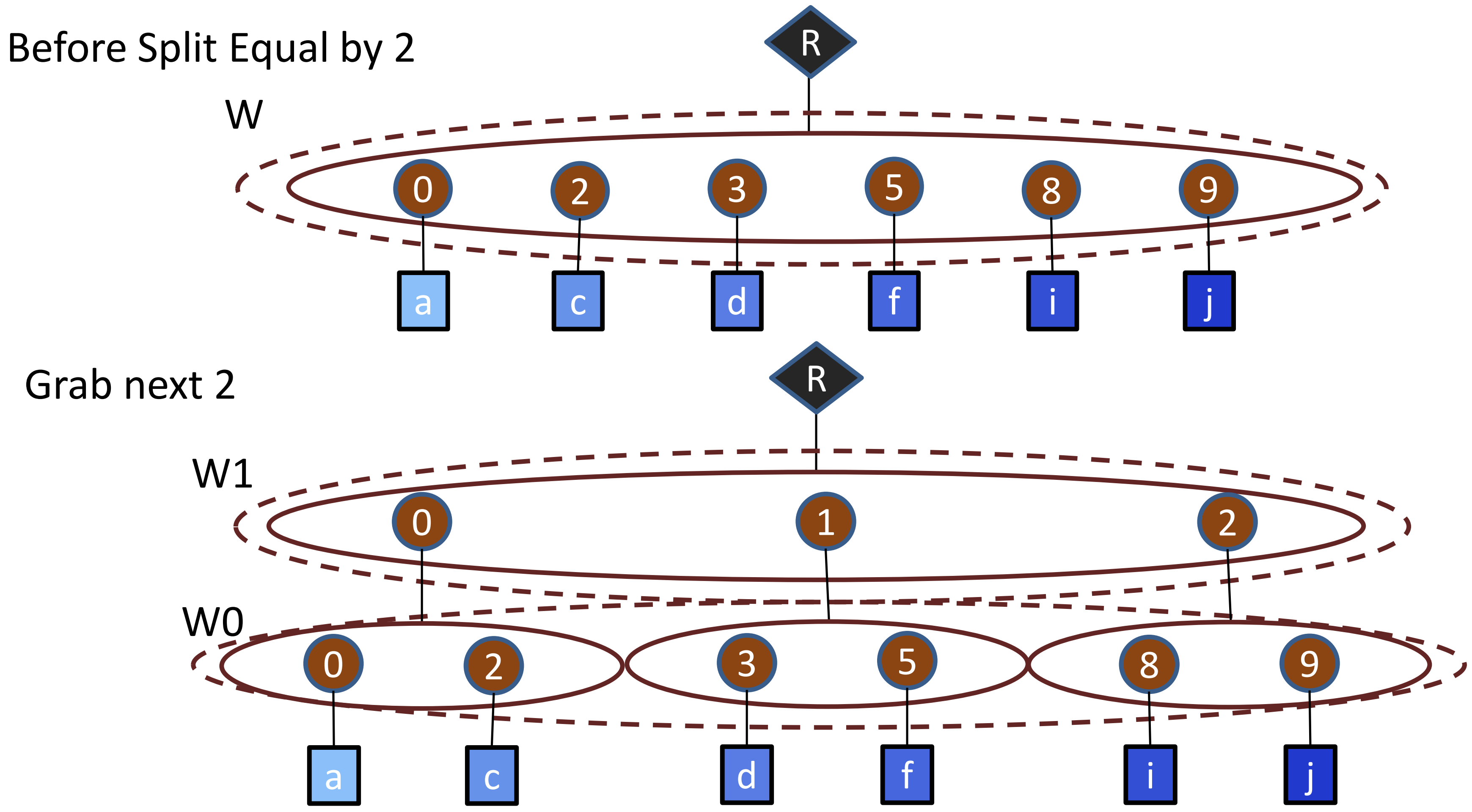
Before Split Equal by 2




Grab next 2

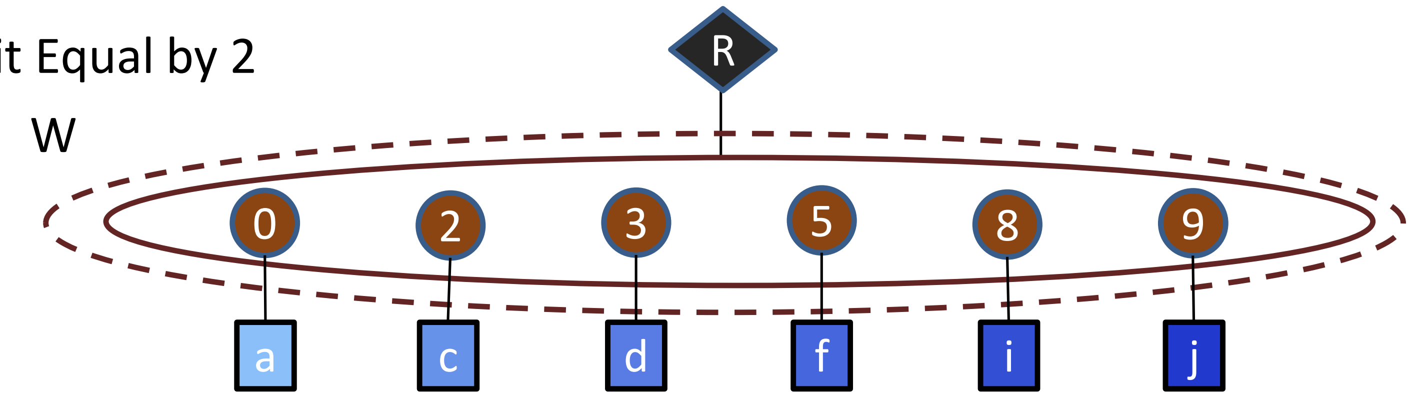


# Fiber Splitting Equally in Position Space

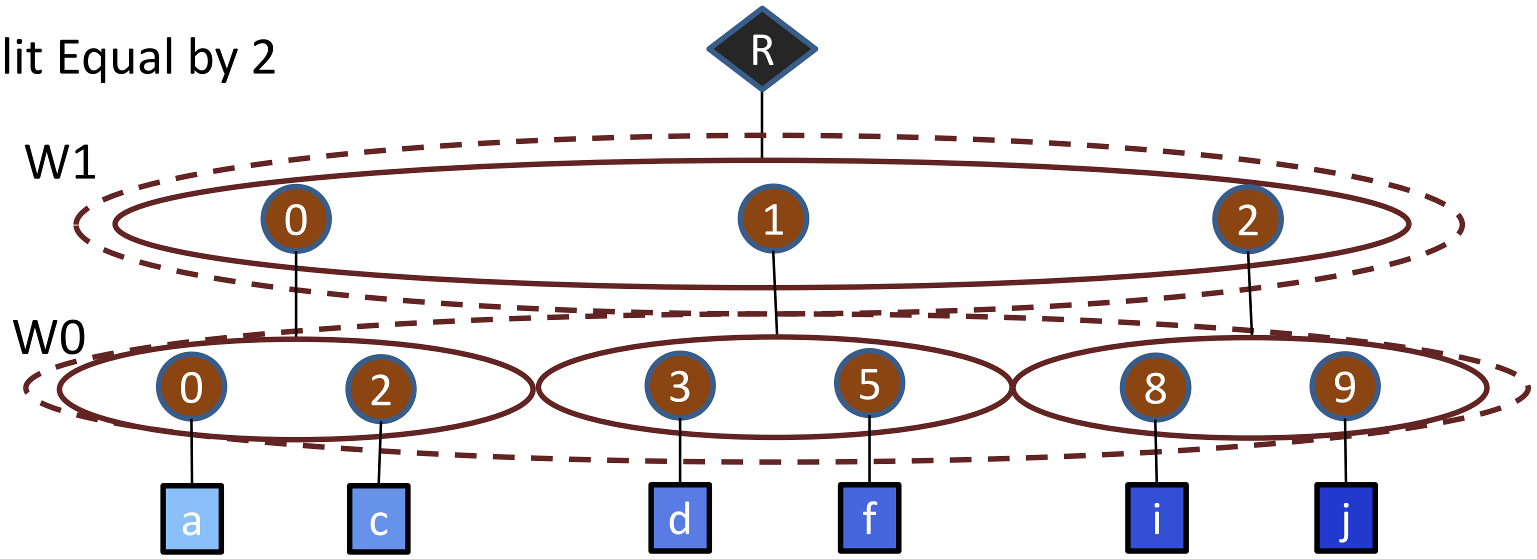


# Fiber Splitting Equally in Position Space

Before Split Equal by 2




After Split Equal by 2



# 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 (s0, 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

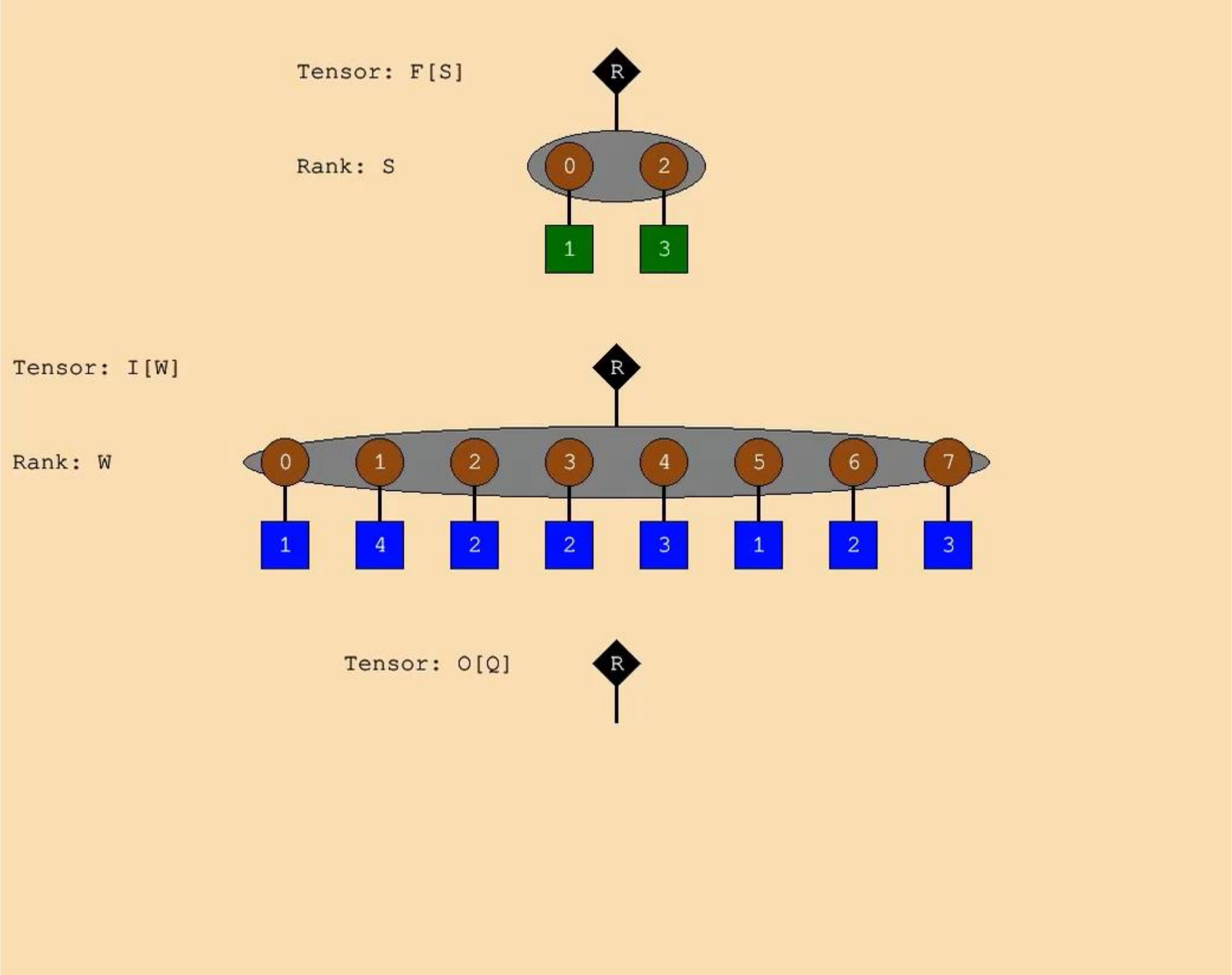
Work on four outputs at once

Calculate coordinates

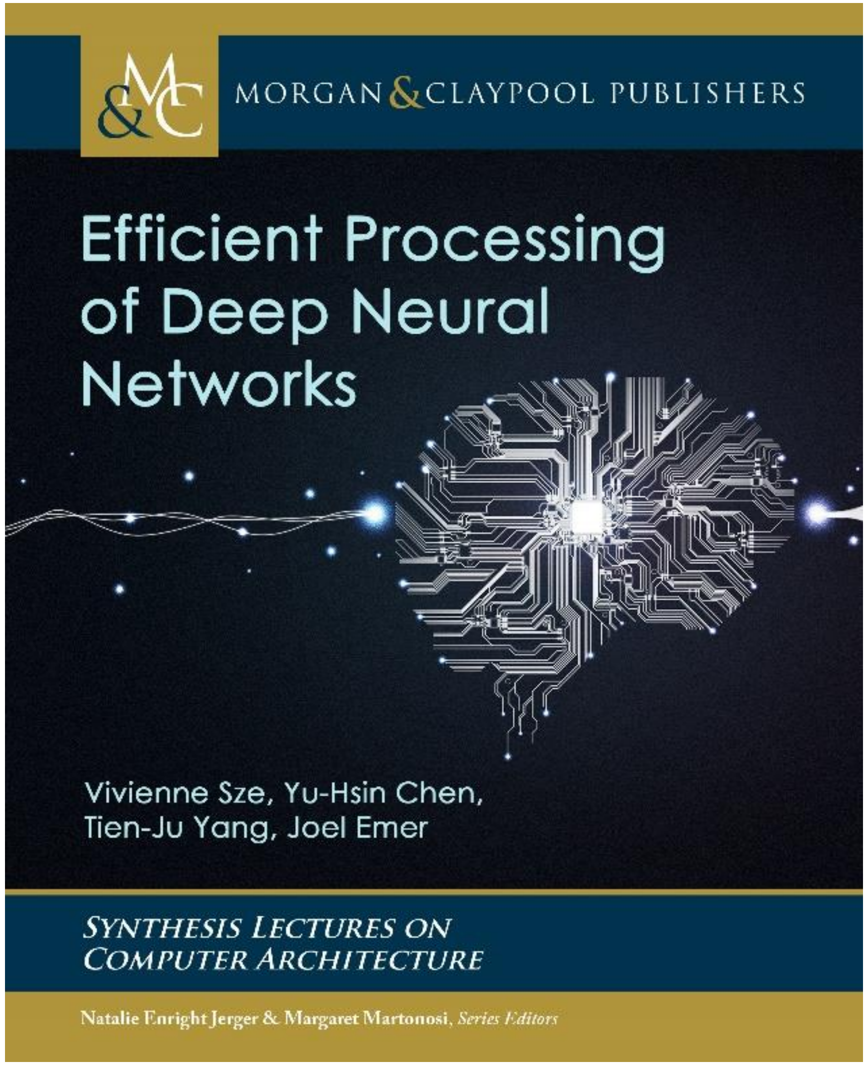
Look up input activation

Accumulate multiple outputs each spatially

# Parallel Weight Stationary - Sparse Weights



# Book on Efficient Processing of DNNs




***Part I Understanding Deep Neural Networks***
*Introduction*
*Overview of Deep Neural Networks*

***Part II Design of Hardware for Processing DNNs***
*Key Metrics and Design Objectives*
*Kernel Computation*
*Designing DNN Accelerators*
*Operation Mapping on Specialized Hardware*

***Part III Co-Design of DNN Hardware and Algorithms***
*Reducing Precision*
*Exploiting Sparsity*
*Designing Efficient DNN Models*
*Advanced Technologies*

https://tinyurl.com/EfficientDNNBook