Tensor Parallelism in

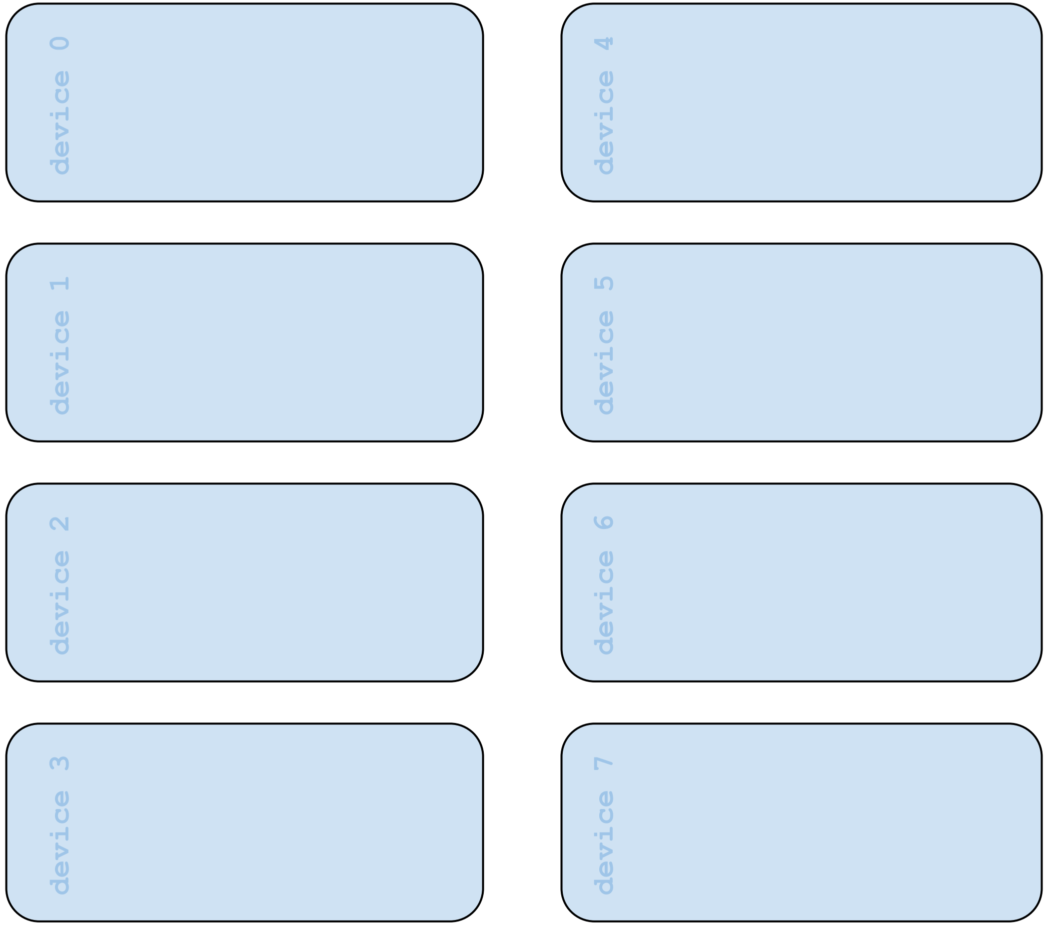
Eastern European Machine Learning Summer School
10-15 July 2023, Košice, Slovakia

Outline of tutorial

- Types of parallelism
- Refresher on matrix multiplication
- Megatron-like sharded MLP
- 2D parallelism

Types of parallelism

- Let's assume we have access to 8 interconnected devices that we can arrange in a grid.
- For example, the grid can be (8, 1) or (4, 2)

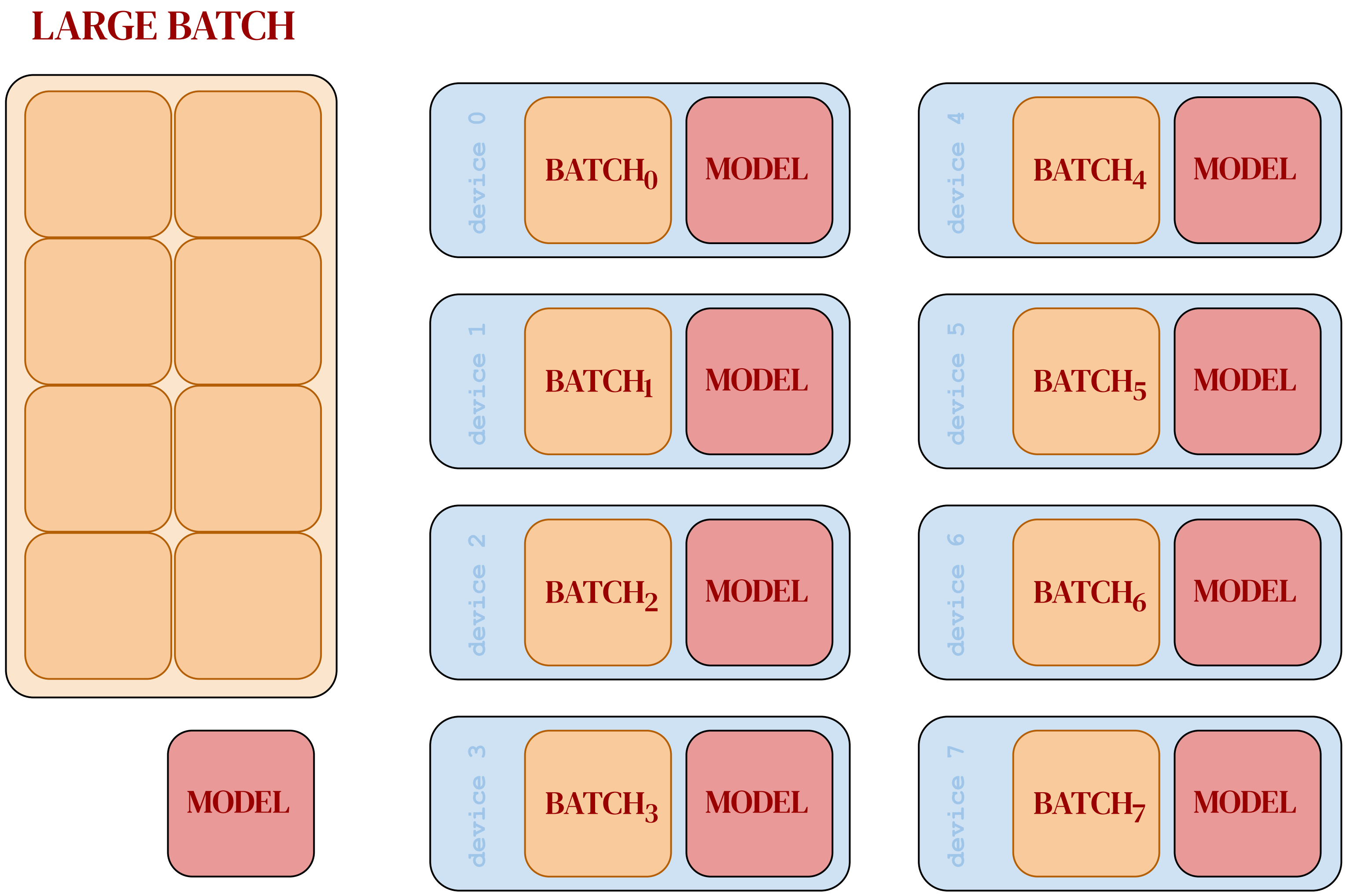


Data parallelism

In this paradigm, we have a model that fits in one device, but we want to speed up training by using larger batches.

To do so, we replicate model across every device and chunk global batch into smaller units.

The same model (forward and backward pass) is executed on each devices, but to keep params in sync, we need to do synchronization step before update.



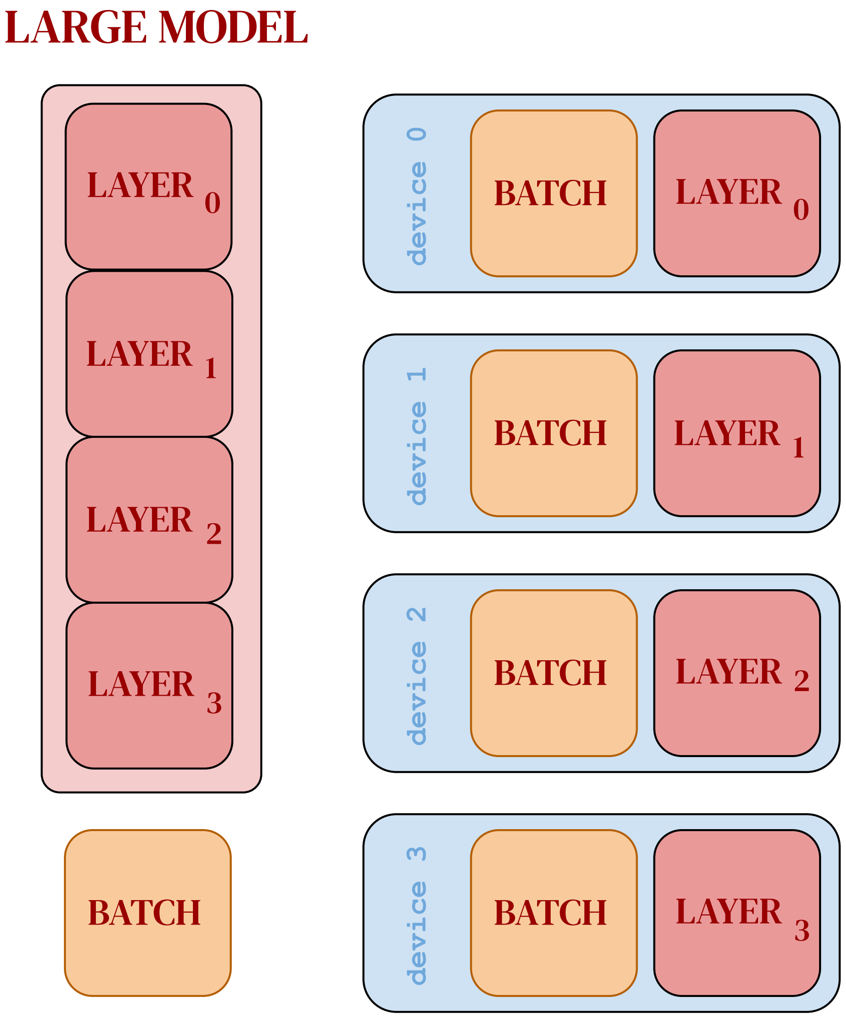
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



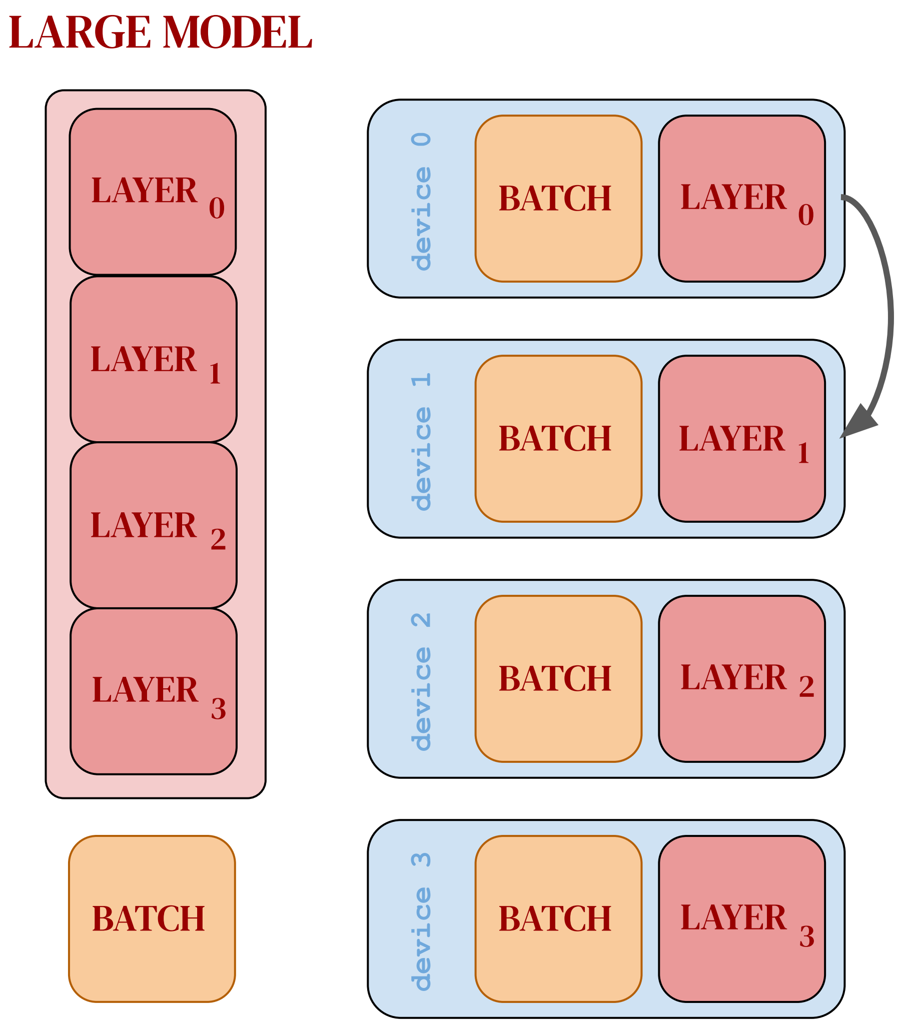
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



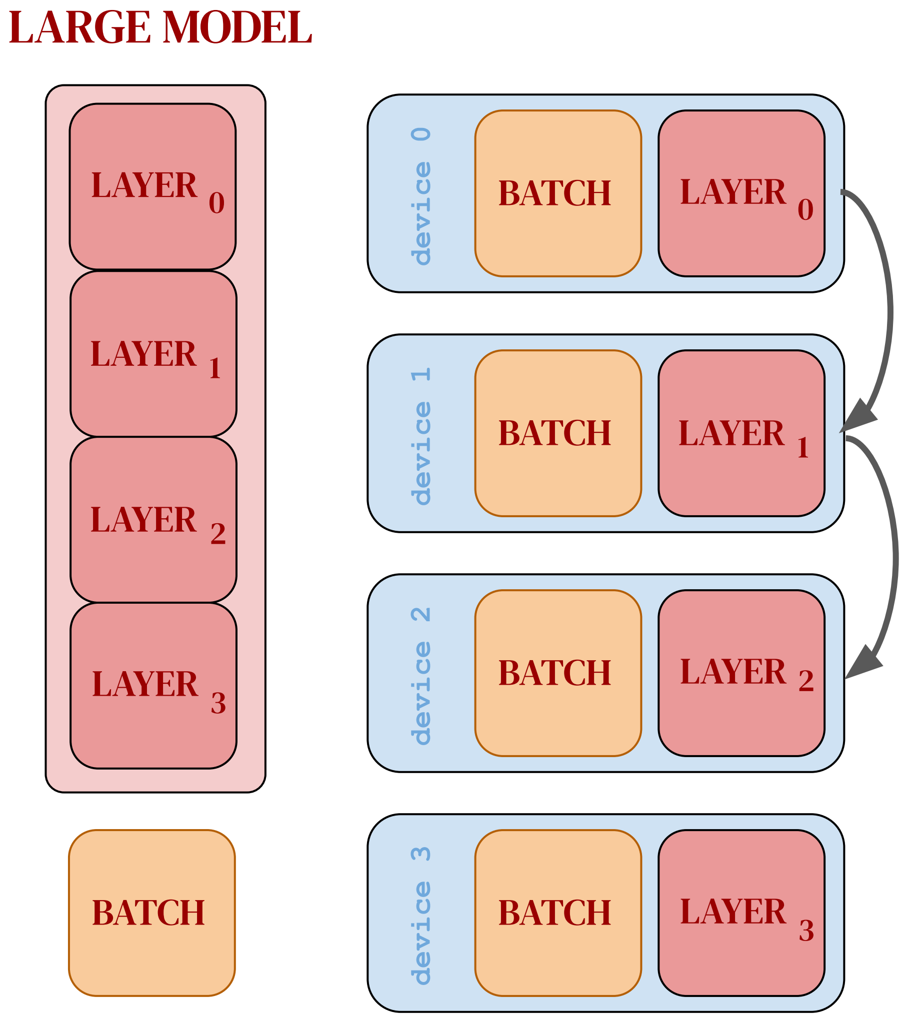
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



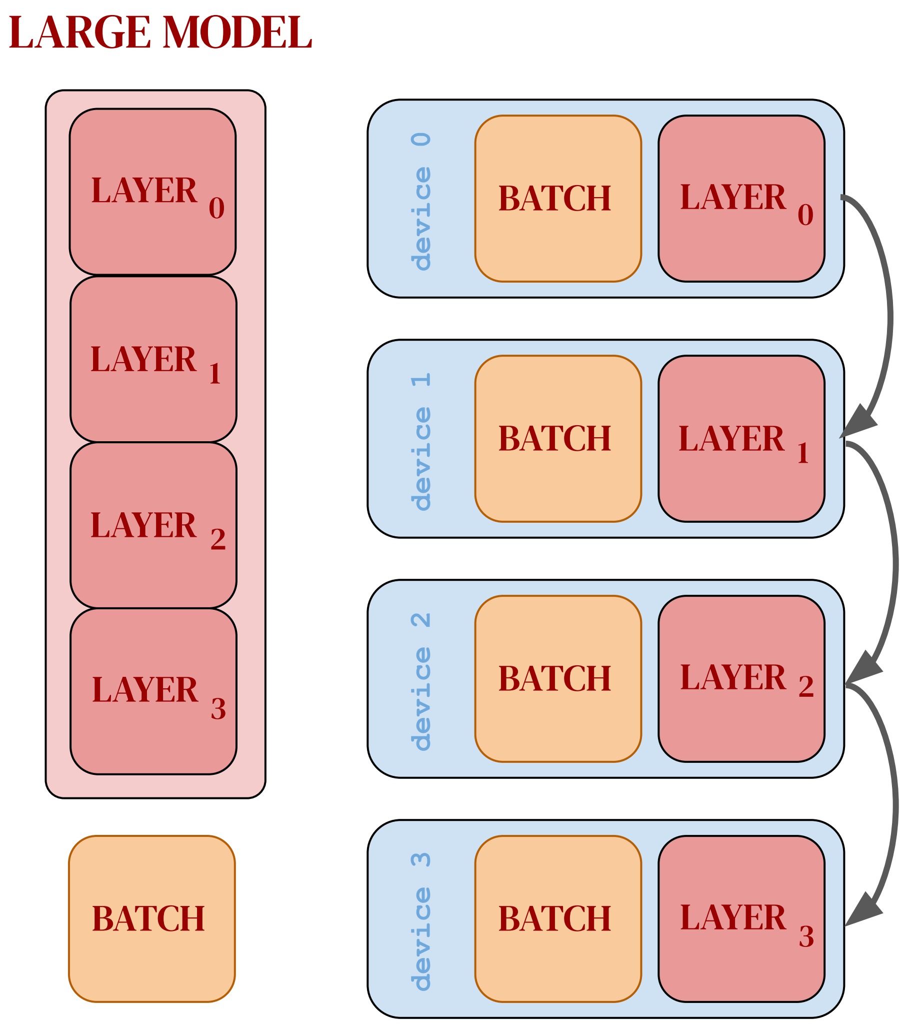
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



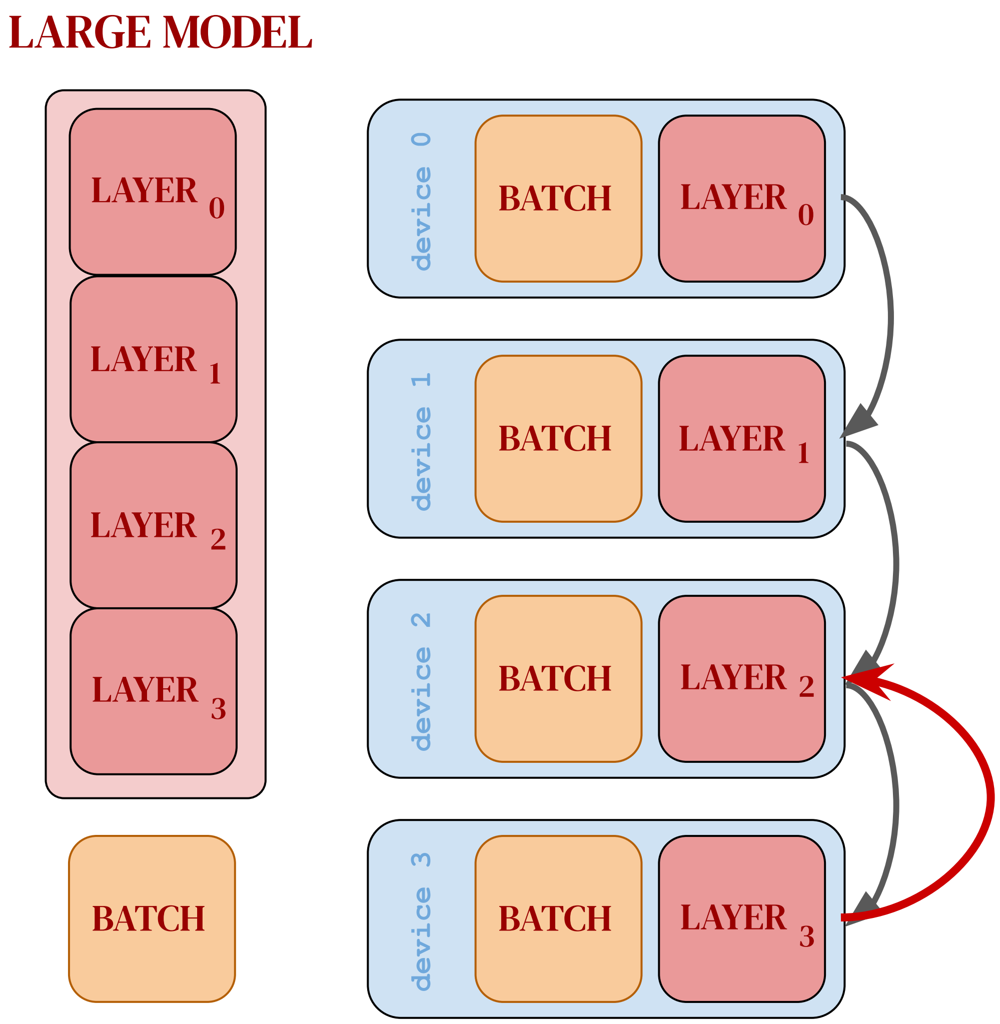
Model parallelism

If or model doesn’t fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



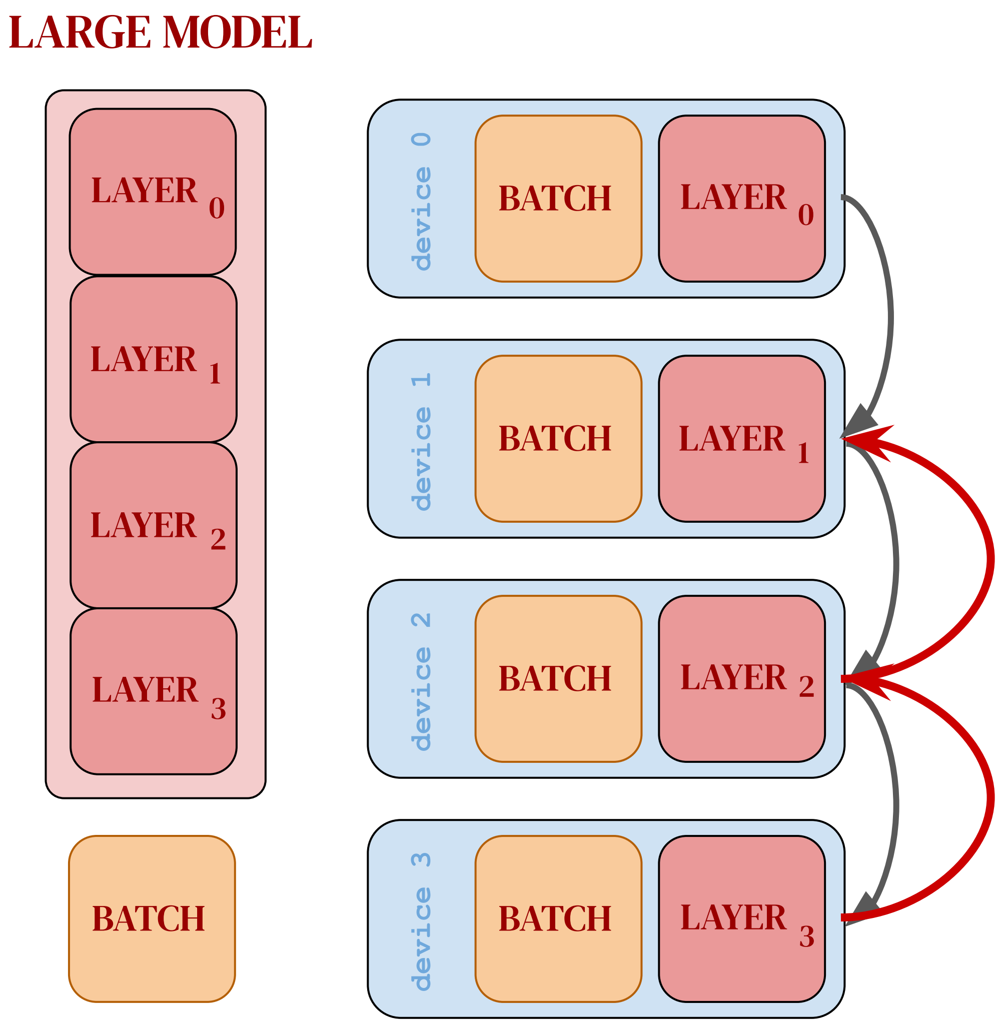
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



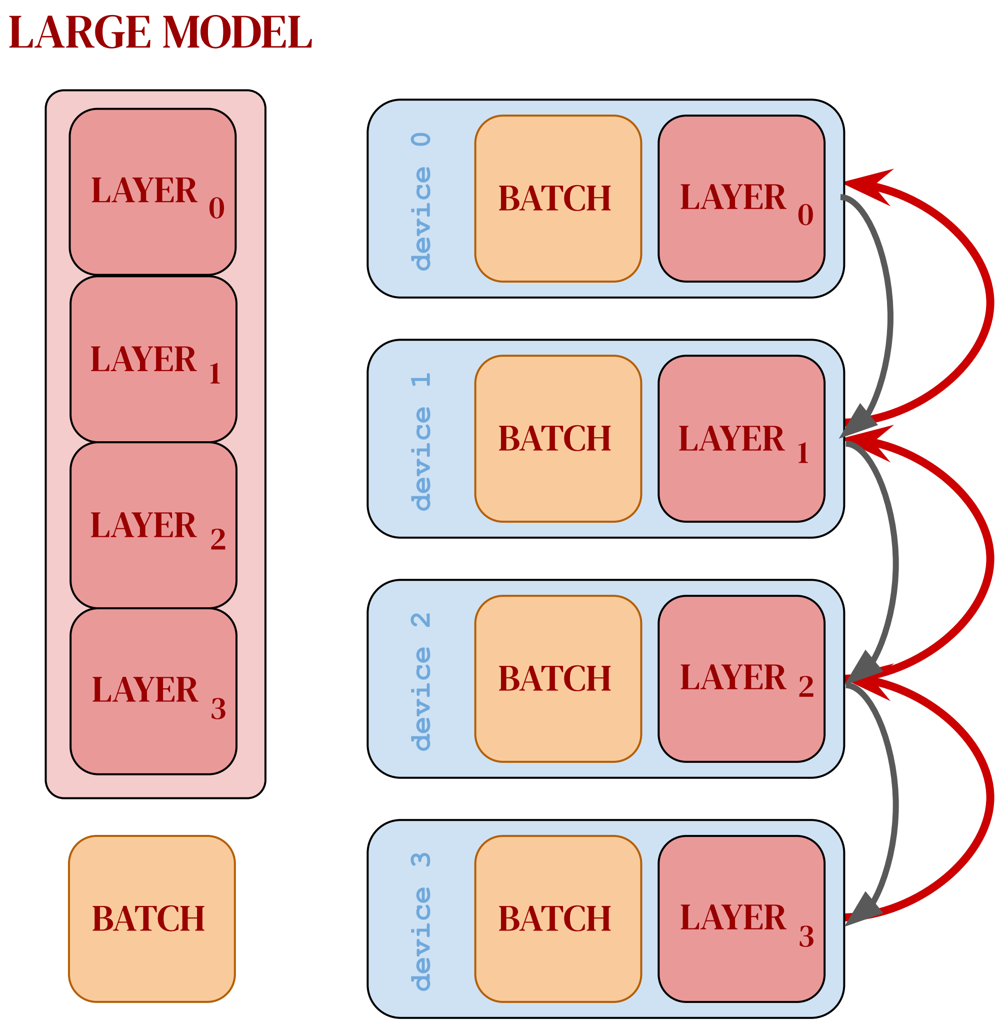
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Pipeline parallelism:** different sub-sequences of layers live on separate accelerators, and the computation flows sequentially from device to device (example GPipe)

- The main difficulty if keeping devices busy.



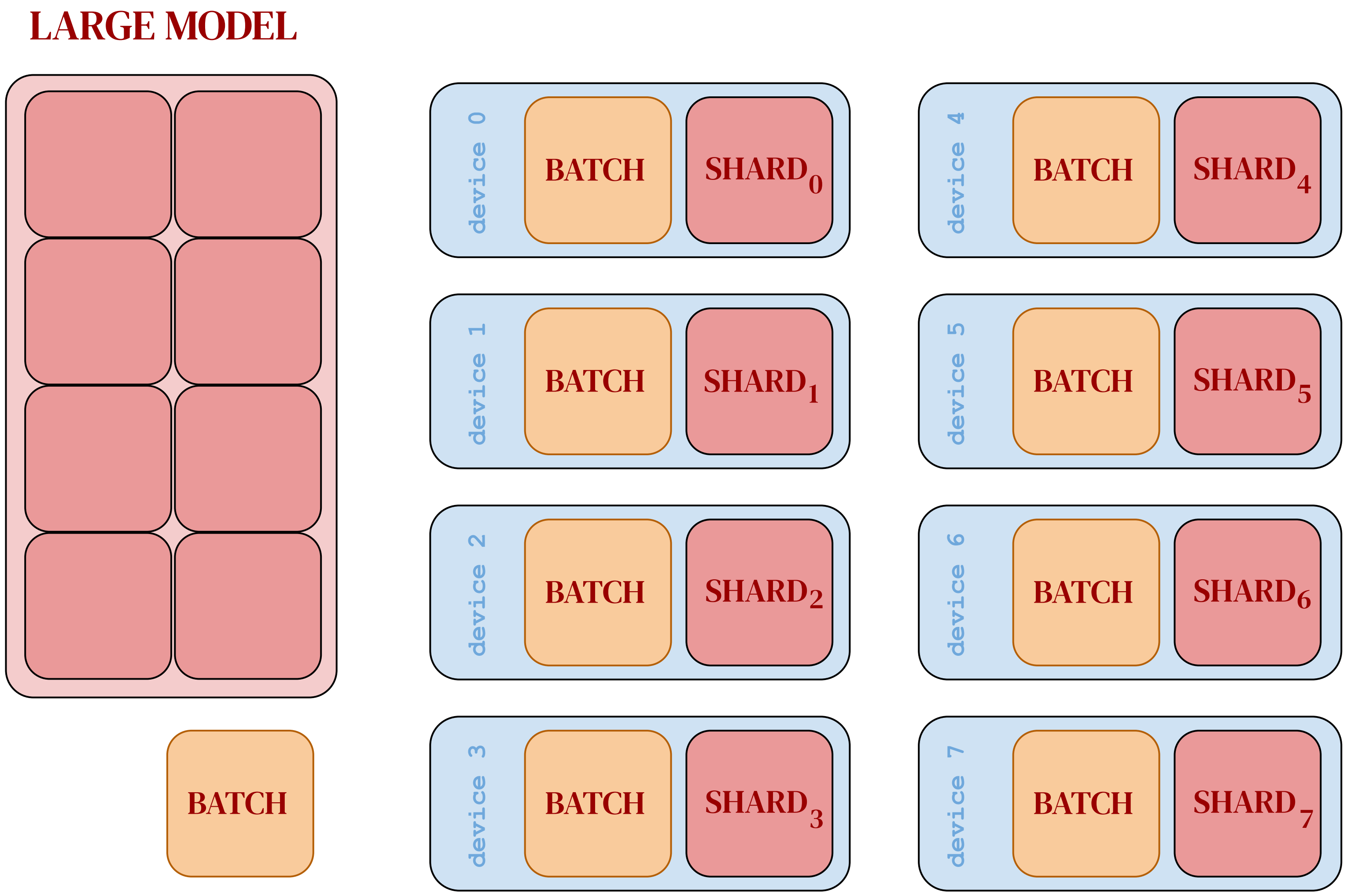
Model parallelism

If or model doesn't fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Tensor parallelism:** individual tensors (activations/weights) are split across devices (example: GSPMD)

- The main difficulty is keeping the communication volume down.



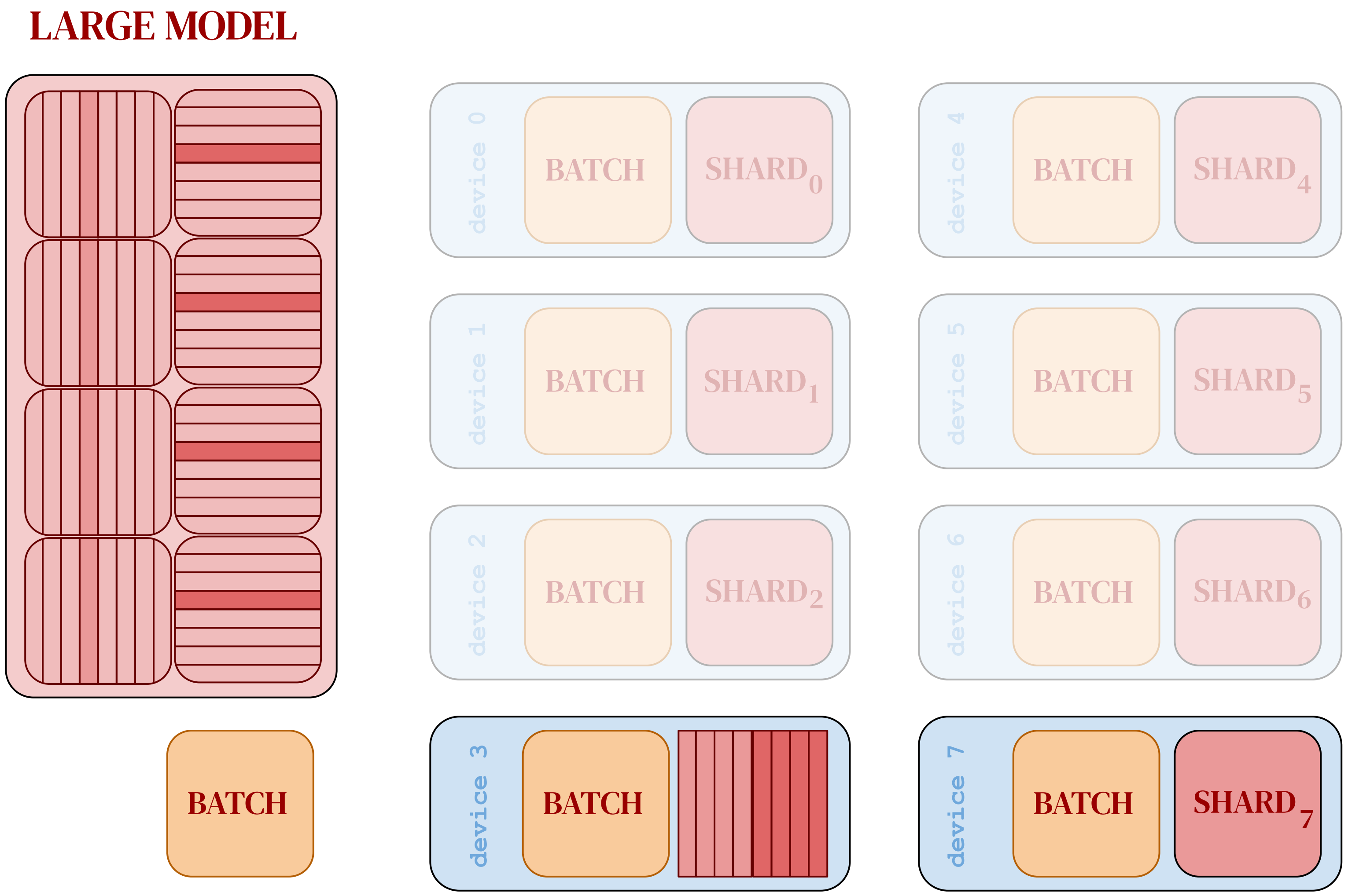
Model parallelism

If or model doesn’t fit on one devices, we can split computation across multiple devices.

There are different kinds of model parallelism

- **Tensor parallelism:** individual tensors (activations/weights) are split across devices (example: GSPMD)

- The main difficulty is keeping the communication volume down.

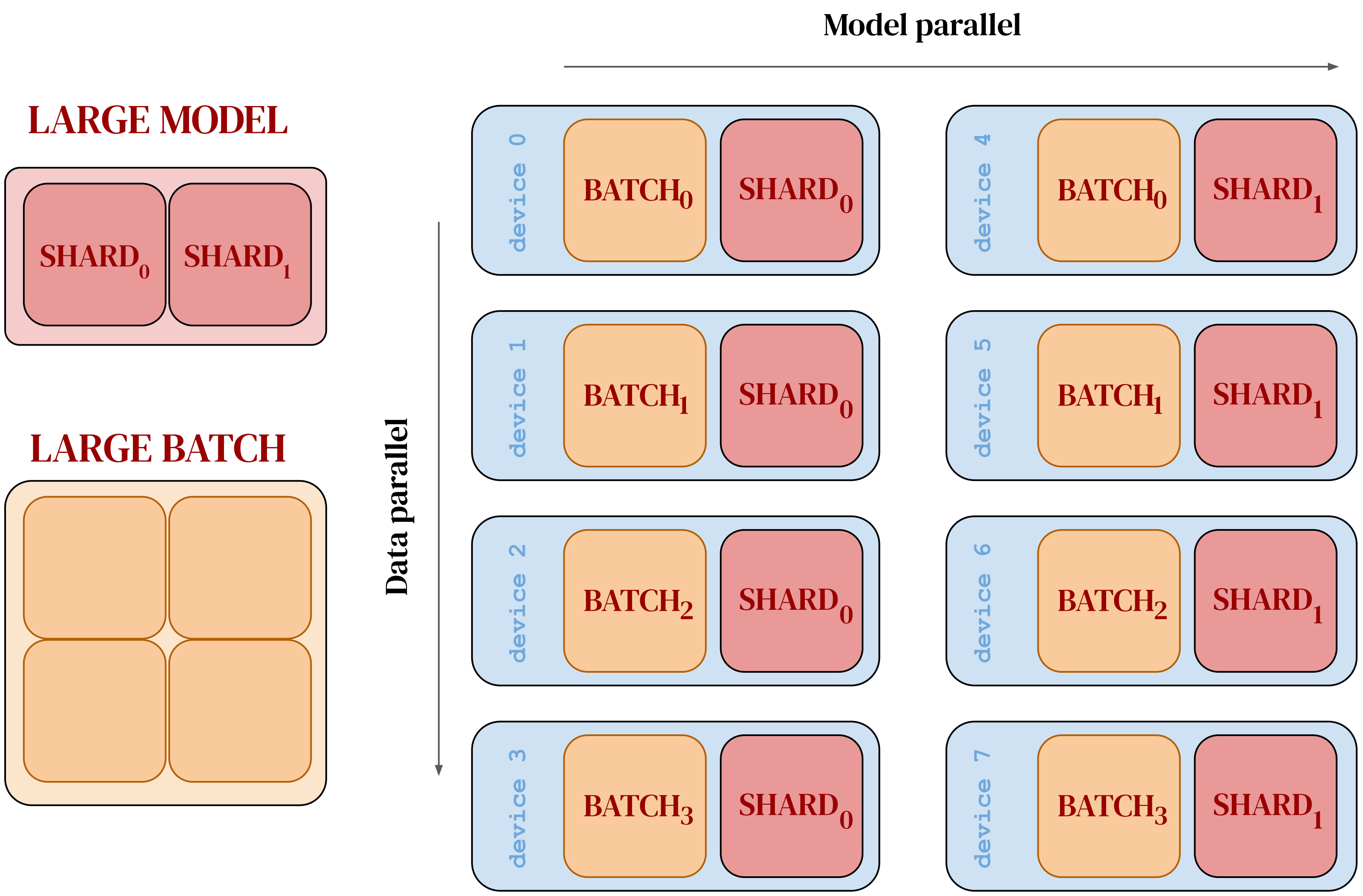


Batch and data parallelism

We can combine data and model parallelism, by arranging our devices in a 2D grid, where each axis represents different partitioning.

For example given (4, 2) grid:

- On the vertical axis we apply apply **batch parallelism** by splitting our large batch in 4
- On the horizontal axis we apply **model parallelism** by splitting the model in 2.



A Refresher on Matrix Multiplication

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} B_{11} \\ B_{21} \\ B_{31} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{13} \\ A_{21} \times B_{11} + A_{22} \times B_{21} + A_{23} \times B_{31} \end{bmatrix} = \begin{bmatrix} C_{11} \\ C_{21} \end{bmatrix}$$

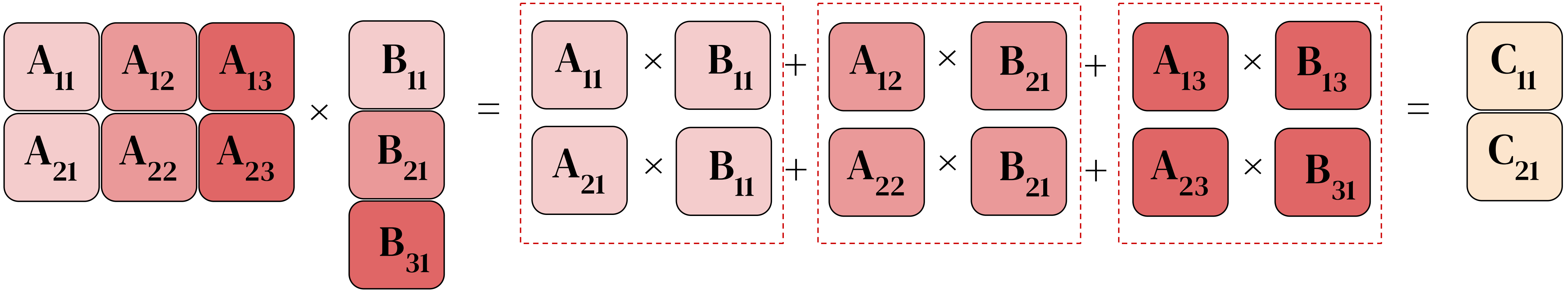
A Refresher on Matrix Multiplication

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} B_{11} \\ B_{21} \\ B_{31} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{13} \\ A_{21} \times B_{11} + A_{22} \times B_{21} + A_{23} \times B_{31} \end{bmatrix} = \begin{bmatrix} C_{11} \\ C_{21} \end{bmatrix}$$

A Refresher on Matrix Multiplication

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \times \begin{bmatrix} B_{11} \\ B_{21} \\ B_{31} \end{bmatrix} = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{13} \\ A_{21} \times B_{11} + A_{22} \times B_{21} + A_{23} \times B_{31} \end{bmatrix} = \begin{bmatrix} C_{11} \\ C_{21} \end{bmatrix}$$

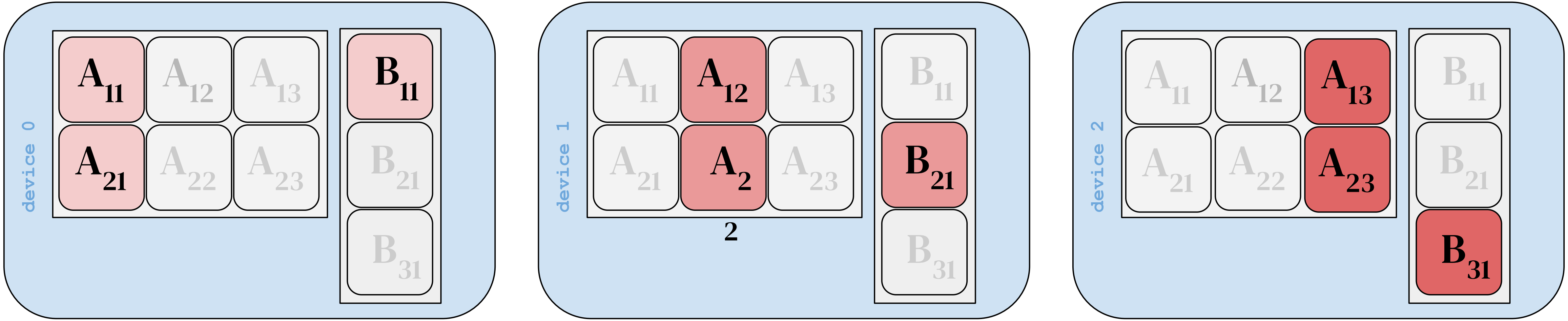
A Refresher on Matrix Multiplication



Those products can be calculated on separate devices

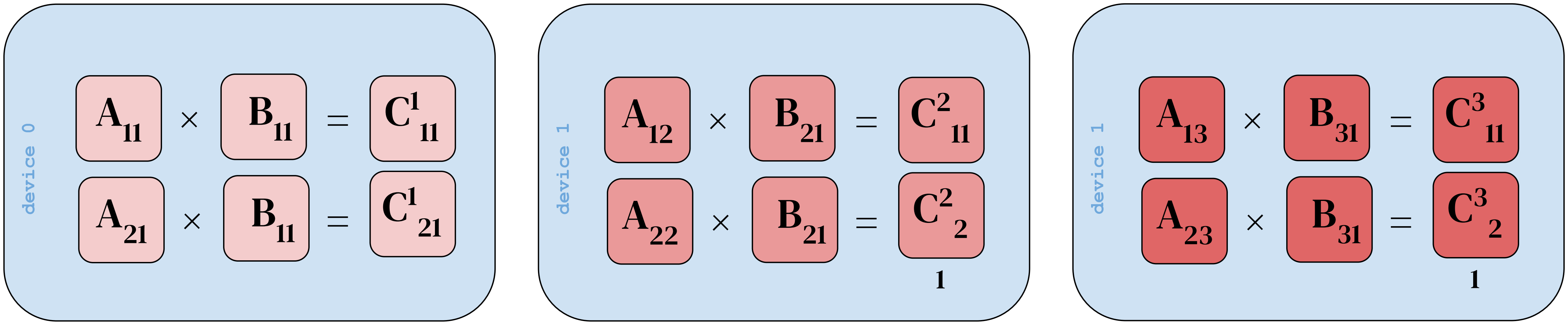
Distributed Matrix Multiplication

Step 1: Shard A on columns, B on rows



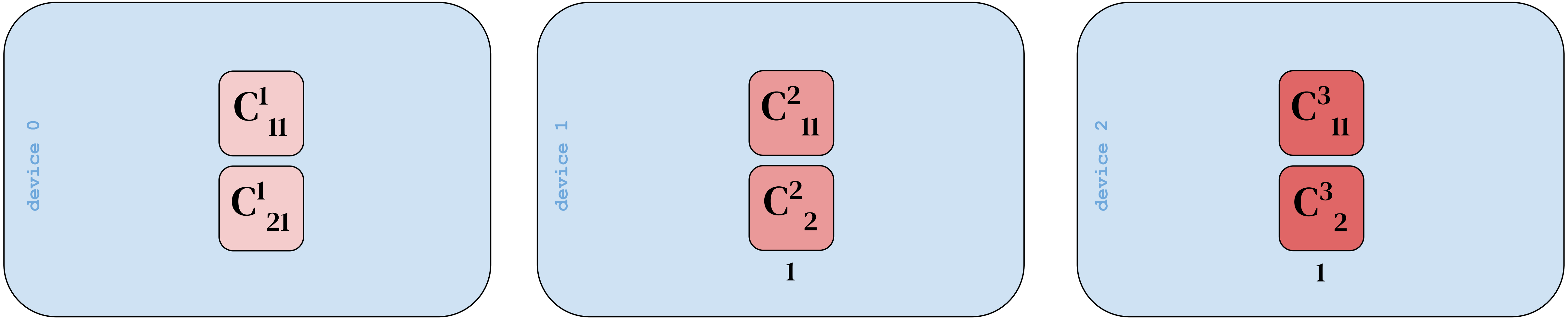
Distributed Matrix Multiplication

Step 2: Perform partial products on devices



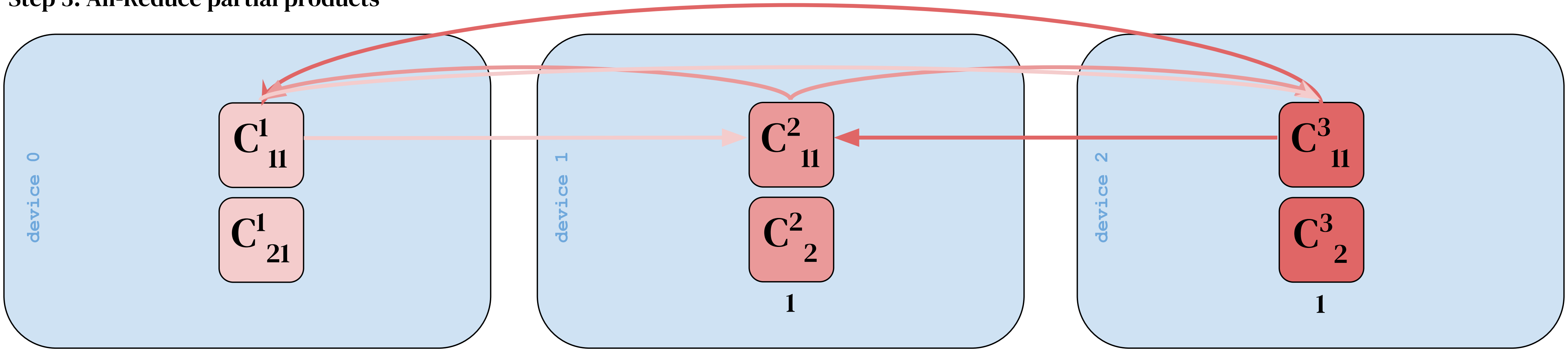
Distributed Matrix Multiplication

Step 3: All-Reduce partial products



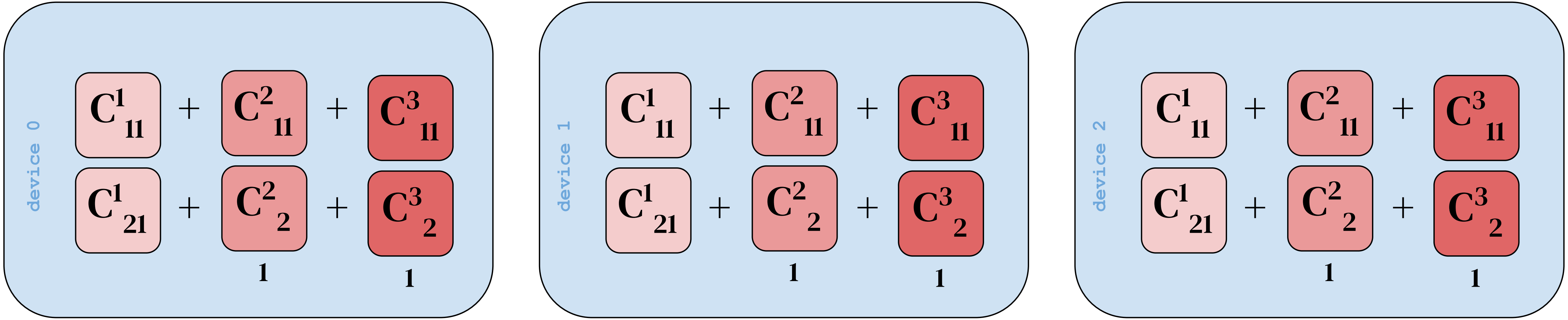
Distributed Matrix Multiplication

Step 3: All-Reduce partial products



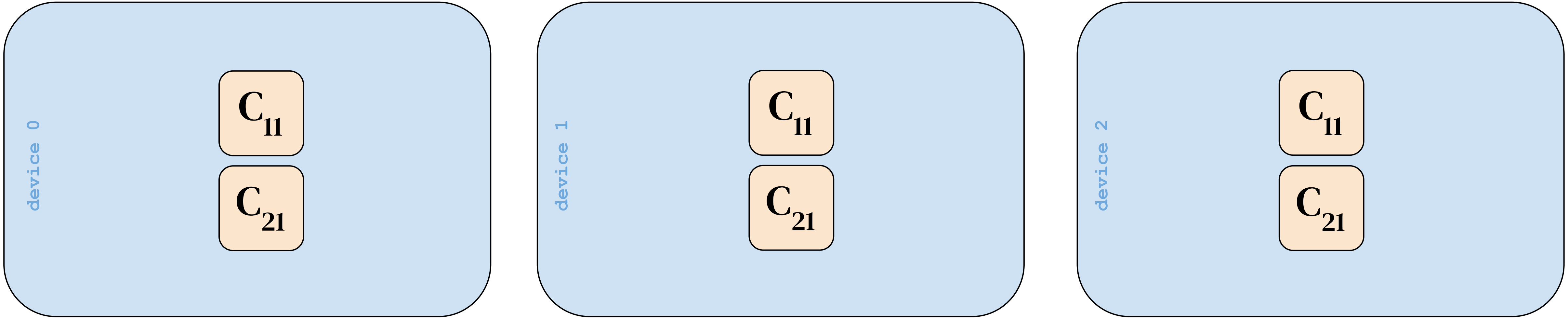
Distributed Matrix Multiplication

Step 3: All-Reduce partial products



Distributed Matrix Multiplication

Step 3: All-Reduce partial products



Distributed Matrix Multiplication - Summary

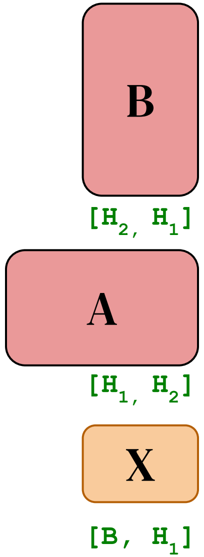
- When multiplying two matrices $\mathbf{A}$, $\mathbf{B}$ such that their reduction dimension is nicely divisible by number devices we can split the computation and perform the partial products $\mathbf{C_i}$ in parallel.
- To recover the unsharded $\mathbf{C} = \mathbf{AB}$ we need to all-reduce partial products incurring communication cost.
- This is a recurring pattern in tensor parallelism: we're trading device memory utilization for communication cost.

Sharding a 2-layer MLP

```
def mlp(
    x:  Float[Array,  "B H_1"],
    A: Float[Array, "H_1 H_2"],
    B: Float[Array, "H_2 H_1"]) -> Float[Array, "B H_1"]:
    """ Z = max(X * W_1, 0) * W_2 """
    y = jnp.dot(x, A)      # [B, H_1] @ [H_1, H_2] -> [B, H_2]
    u = jnp.maximum(y, 0) # [B, H_2]
    z = jnp.dot(u, B)      # [B, H_2] @ [H_2, H_1] -> [B, H_1]
    return z
```

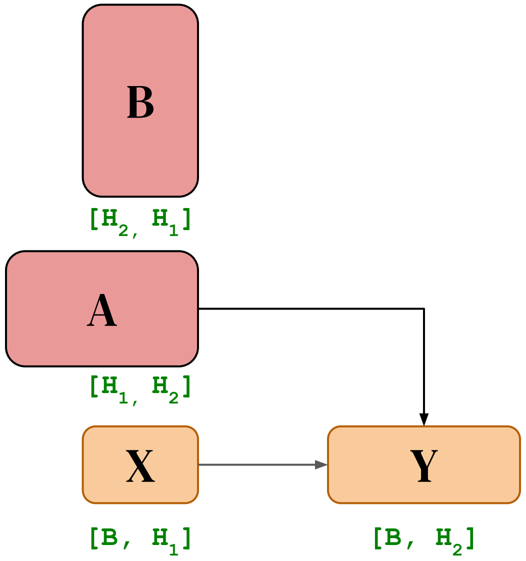
Sharding a 2-layer MLP

```
def mlp(
    x:  Float[Array,  "B H_1"],
    A: Float[Array, "H_1 H_2"],
    B: Float[Array, "H_2 H_1"]) -> Float[Array, "B H_1"]:
    """ Z = max(X * W_1, 0) * W_2 """
    y = jnp.dot(x, A)      # [B, H_1] @ [H_1, H_2] -> [B, H_2]
    u = jnp.maximum(y, 0) # [B, H_2]
    z = jnp.dot(u, B)      # [B, H_2] @ [H_2, H_1] -> [B, H_1]
    return z
```



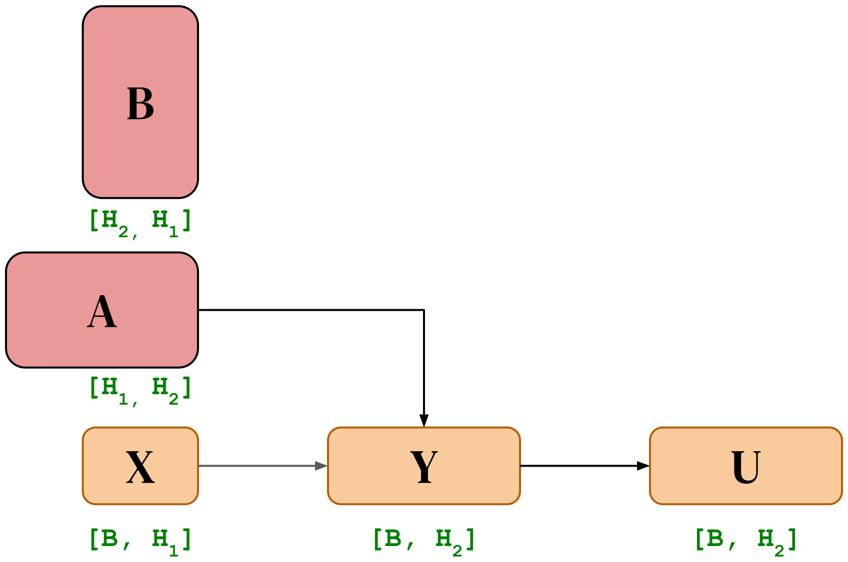
Sharding a 2-layer MLP

```python
def mlp(
    x:  Float[Array,  "B H_1"],
    A: Float[Array, "H_1 H_2"],
    B: Float[Array, "H_2 H_1"]) -> Float[Array, "B H_1"]:
    """ Z = max(X * W_1, 0) * W_2 """
    y = jnp.dot(x, A)      # [B, H_1] @ [H_1, H_2] -> [B, H_2]
    u = jnp.maximum(y, 0) # [B, H_2]
    z = jnp.dot(u, B)      # [B, H_2] @ [H_2, H_1] -> [B, H_1]
    return z
```



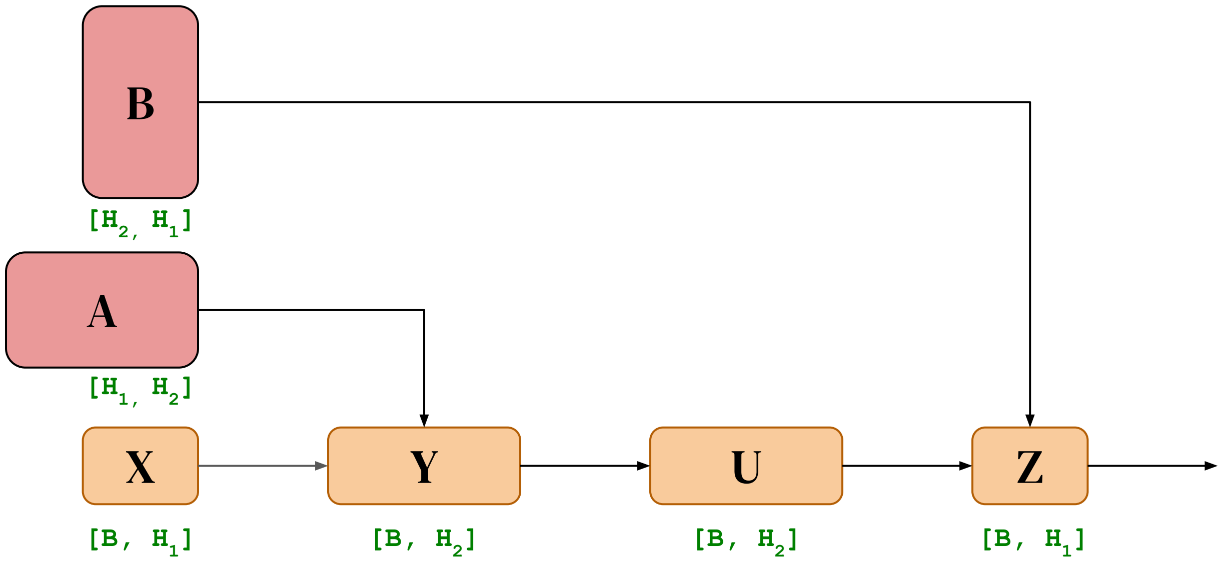
Sharding a 2-layer MLP

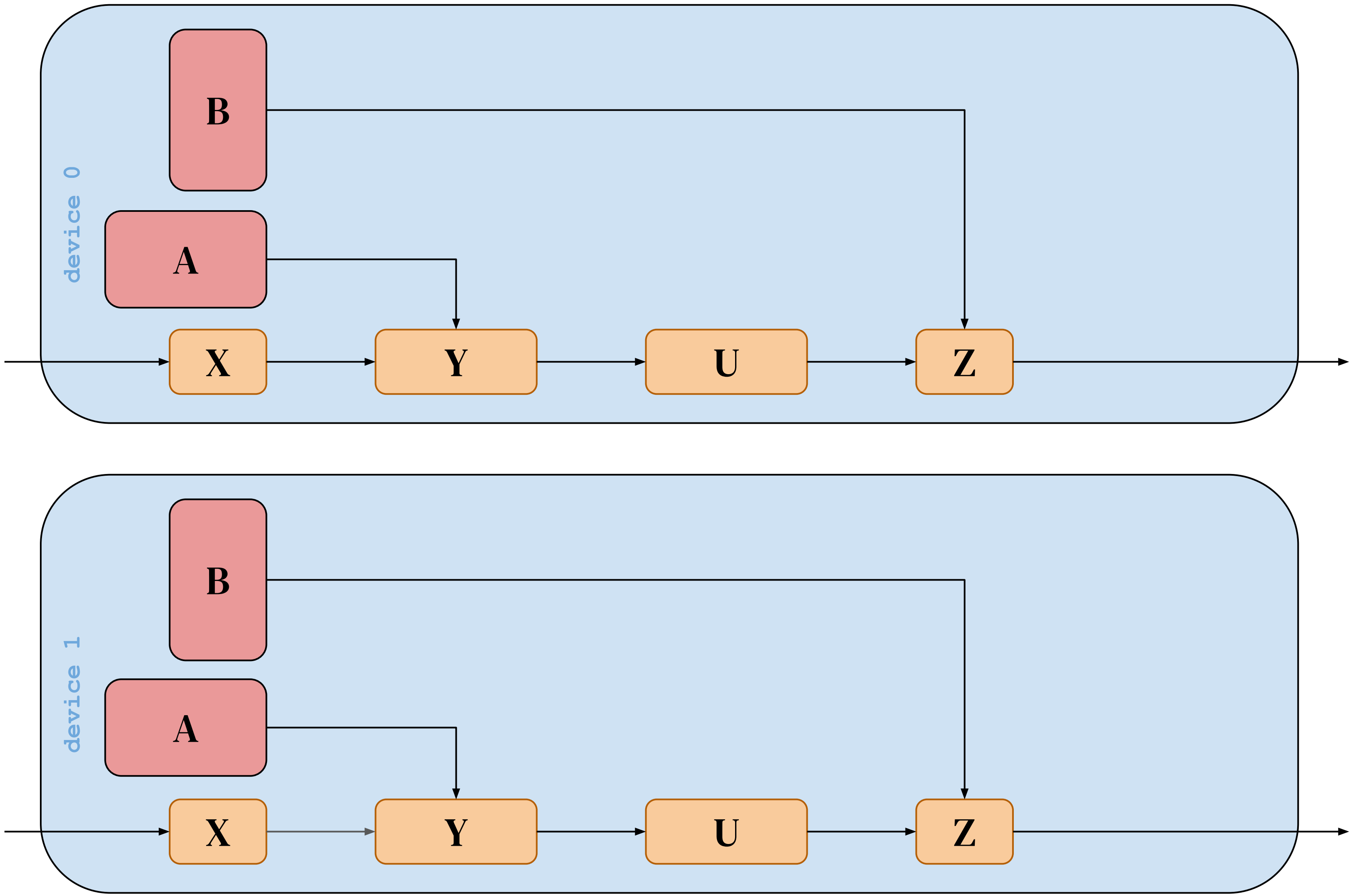
```
def mlp(
    x:  Float[Array,  "B H_1"],
    A: Float[Array, "H_1 H_2"],
    B: Float[Array, "H_2 H_1"]) -> Float[Array, "B H_1"]:
    """ Z = max(X * W_1, 0) * W_2 """
    y = jnp.dot(x, A)      # [B, H_1] @ [H_1, H_2] -> [B, H_2]
    u = jnp.maximum(y, 0) # [B, H_2]
    z = jnp.dot(u, B)      # [B, H_2] @ [H_2, H_1] -> [B, H_1]
    return z
```



Sharding a 2-layer MLP

```
def mlp(
    x:  Float[Array,  "B H_1"],
    A: Float[Array, "H_1 H_2"],
    B: Float[Array, "H_2 H_1"]) -> Float[Array, "B H_1"]:
    """ Z = max(X * W_1, 0) * W_2 """
    y = jnp.dot(x, A)      # [B, H_1] @ [H_1, H_2] -> [B, H_2]
    u = jnp.maximum(y, 0) # [B, H_2]
    z = jnp.dot(u, B)      # [B, H_2] @ [H_2, H_1] -> [B, H_1]
    return z
```

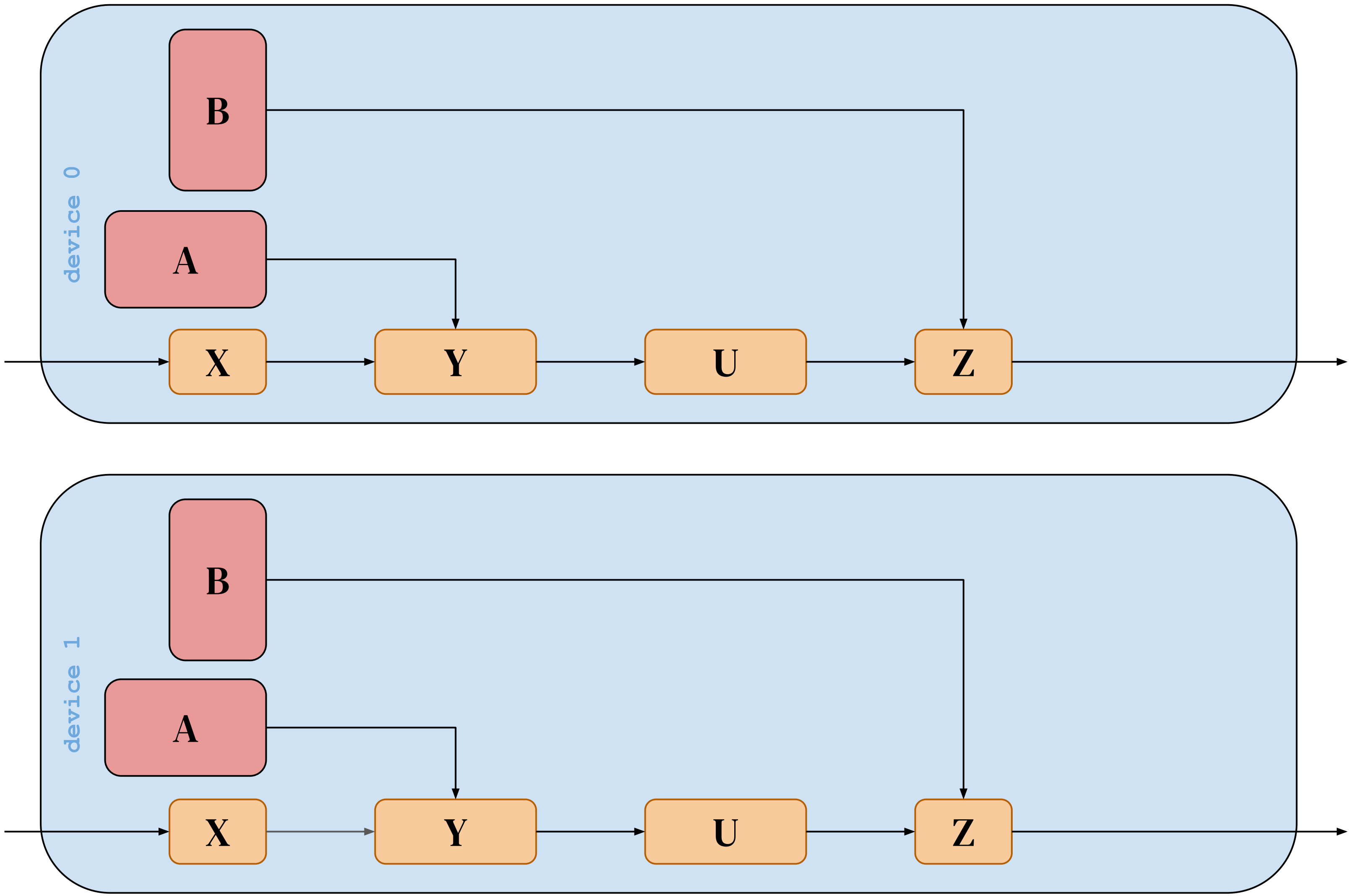


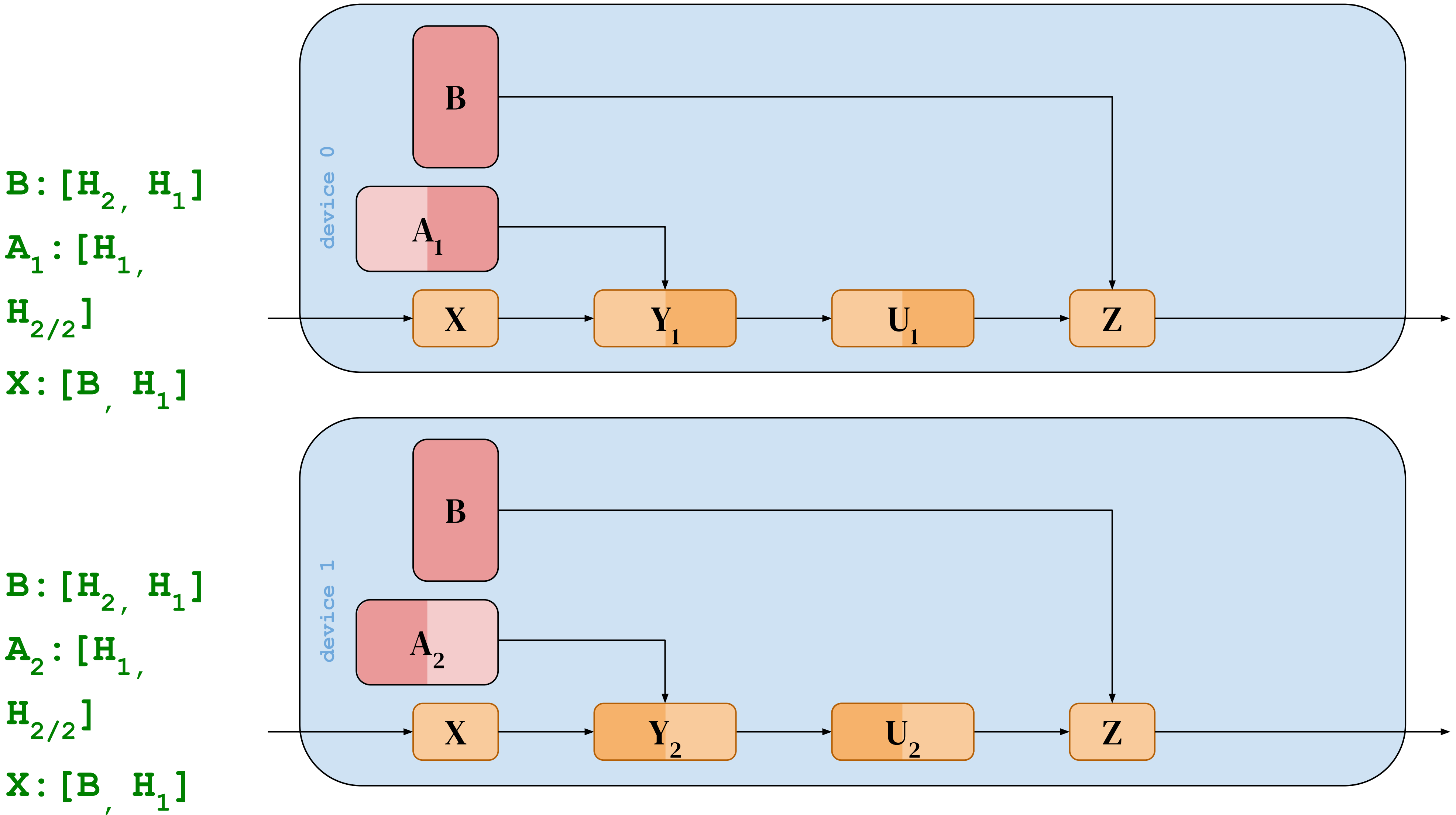

device 0
B
A
X
Y
U
Z
device 1
B
A
X
Y
U
Z

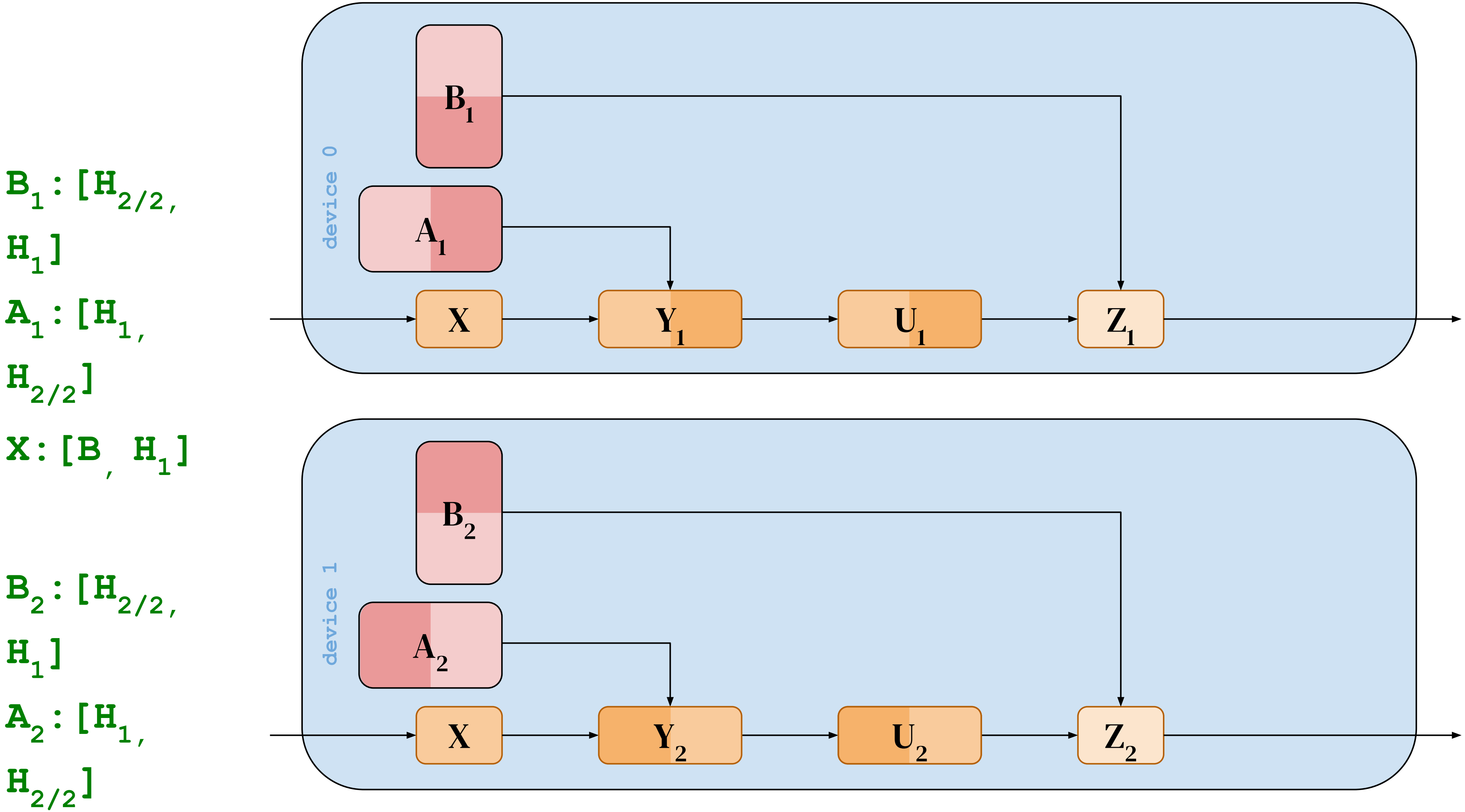
$B:[H_2, H_1]$

$A:[H_1, H_2]$

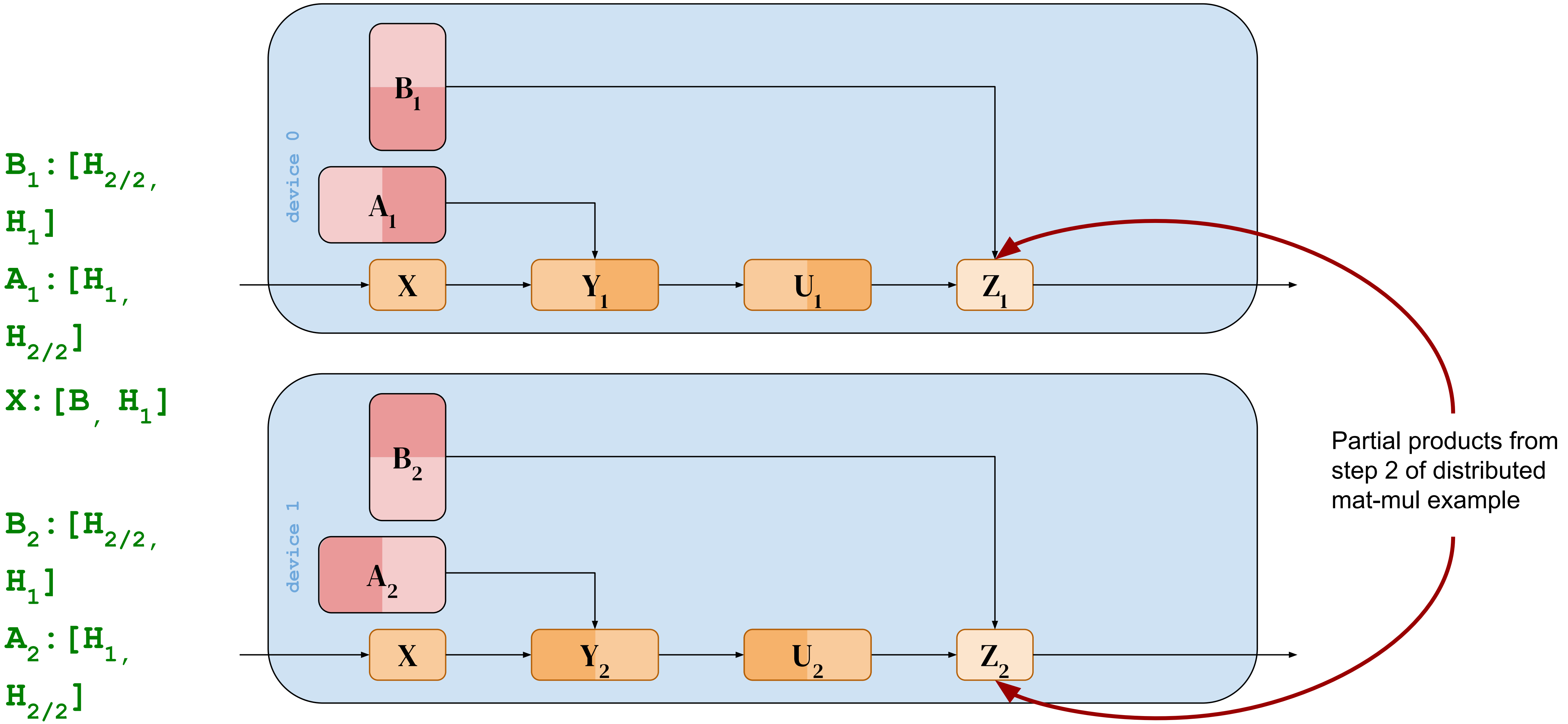
$X:[B, H_1]$

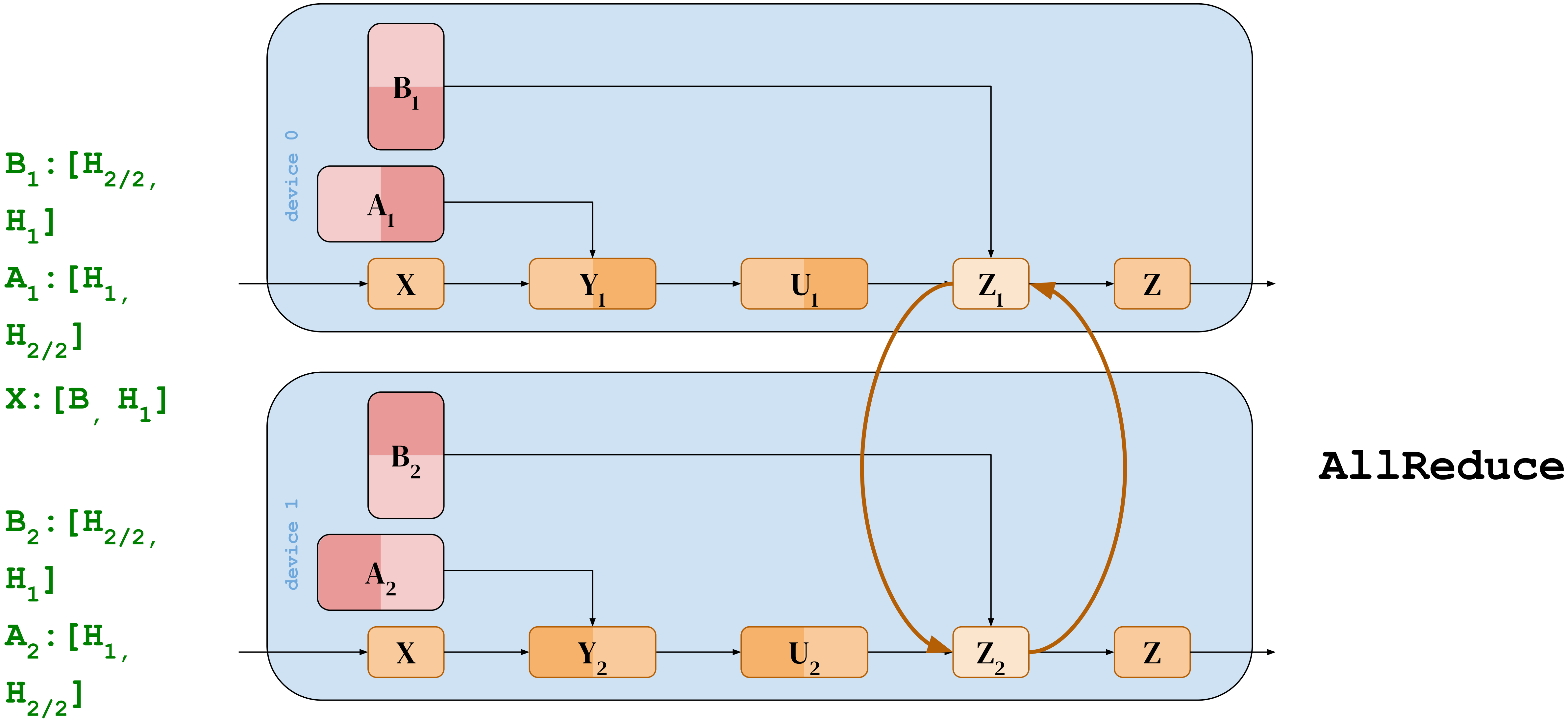



B:[H2, H1]
A1:[H1, H2/2]
X:[B, H1]
device 0
B
A1
X
Y1
U1
Z
B:[H2, H1]
A2:[H1, H2/2]
X:[B, H1]
device 1
B
A2
X
Y2
U2
Z



B1: [H2/2, H1]
A1: [H1, H2/2]
X: [B, H1]
B2: [H2/2, H1]
A2: [H1, H2/2]
device 0
device 1
B1
A1
X
Y1
U1
Z1
B2
A2
X
Y2
U2
Z2


B1: [H2/2, H1]
A1: [H1, H2/2]
X: [B, H1]
B2: [H2/2, H1]
A2: [H1, H2/2]
device 0
device 1
B1
A1
X
Y1
U1
Z1
B2
A2
X
Y2
U2
Z2
Partial products from step 2 of distributed mat-mul example

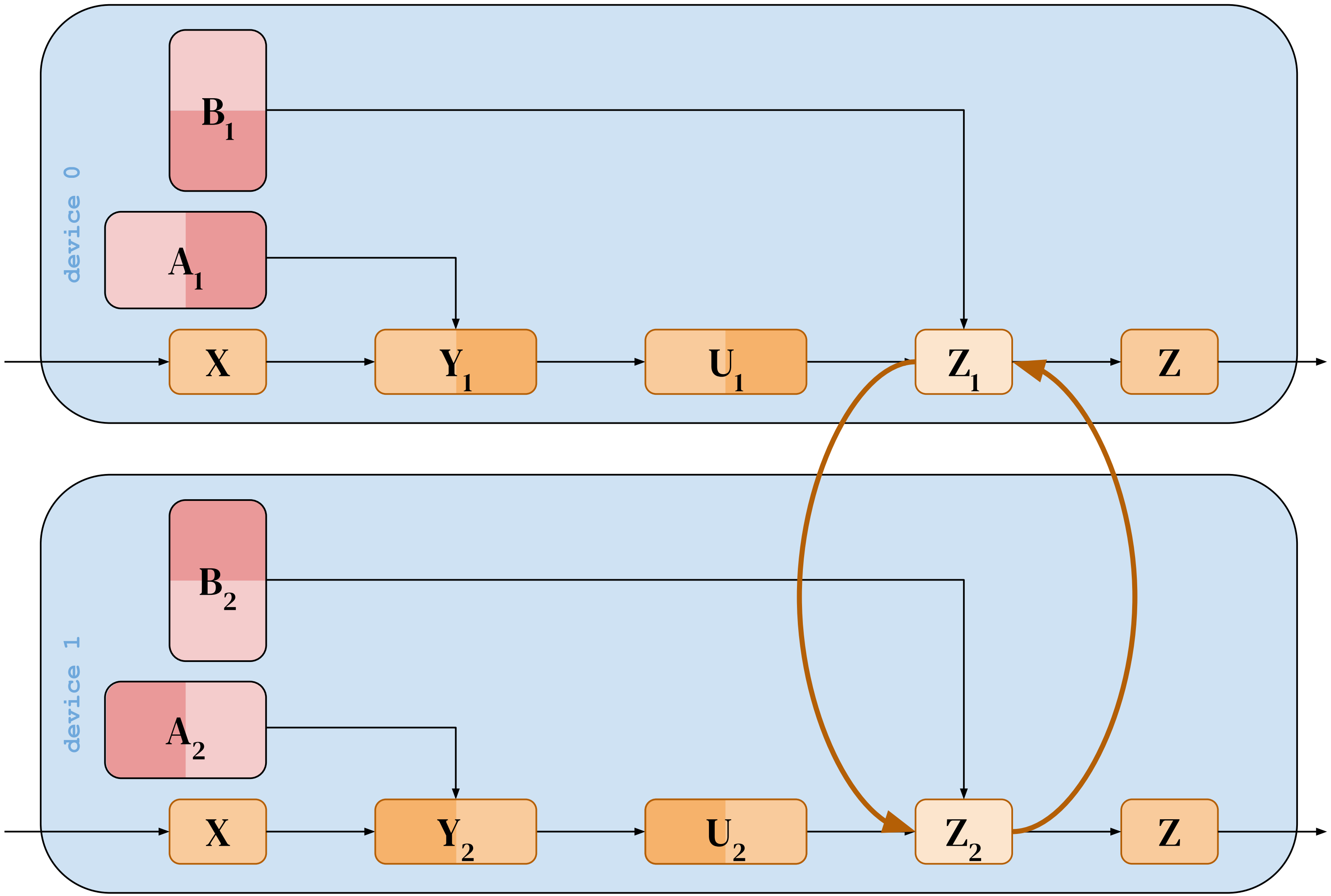


$B_1: [H_{2/2}, H_1]$
$A_1: [H_1, H_{2/2}]$
$X: [B, H_1]$
$B_2: [H_{2/2}, H_1]$
$A_2: [H_1, H_{2/2}]$
device 0
device 1
B_1
A_1
X
Y_1
U_1
Z_1
Z
B_2
A_2
X
Y_2
U_2
Z_2
Z
AllReduce

B_1: [$H_{2/2}$, H_1]

A_1: [H_1, $H_{2/2}$]

X: [B, H_1]



B_2: [$H_{2/2}$, H_1]

A_2: [H_1, $H_{2/2}$]

What about gradients?

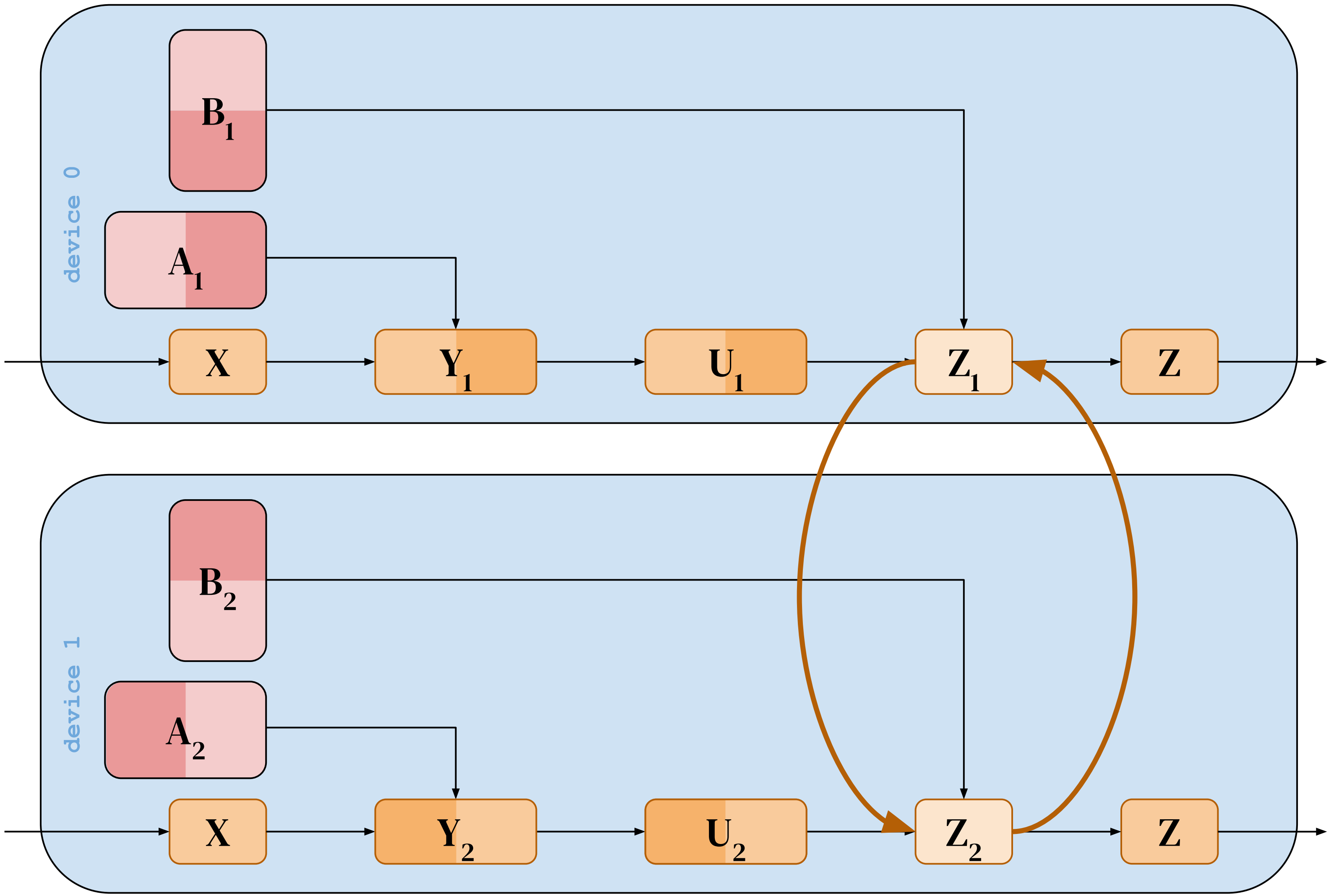
B_1: [$H_{2/2}$, H_1]

A_1: [H_1, $H_{2/2}$]

X: [B, H_1]

B_2: [$H_{2/2}$, H_1]

A_2: [H_1, $H_{2/2}$]



Let's forget about devices for a while

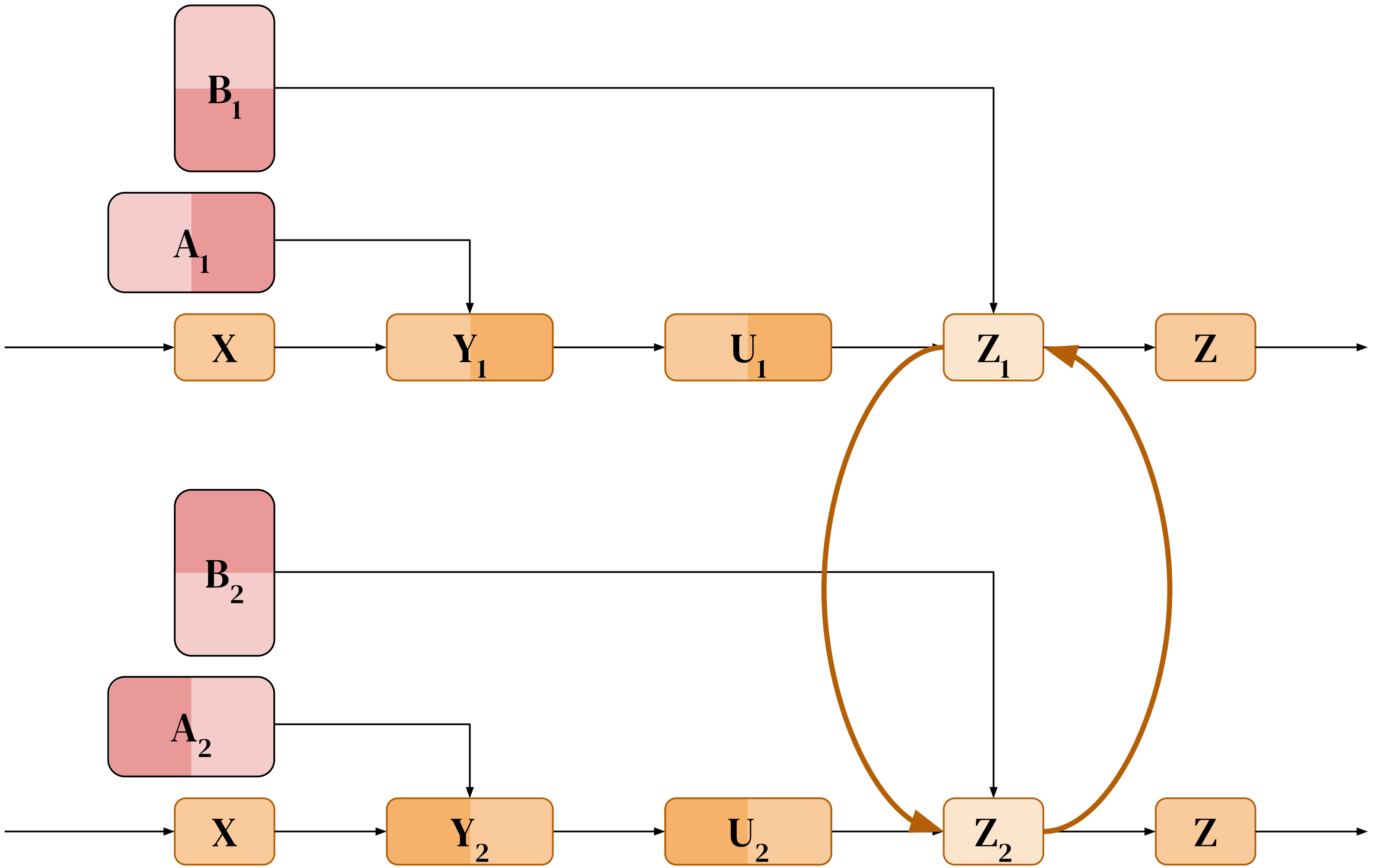
B_1: [$H_{2/2}$, H_1]

A_1: [H_1, $H_{2/2}$]

X: [$B_,$ H_1]

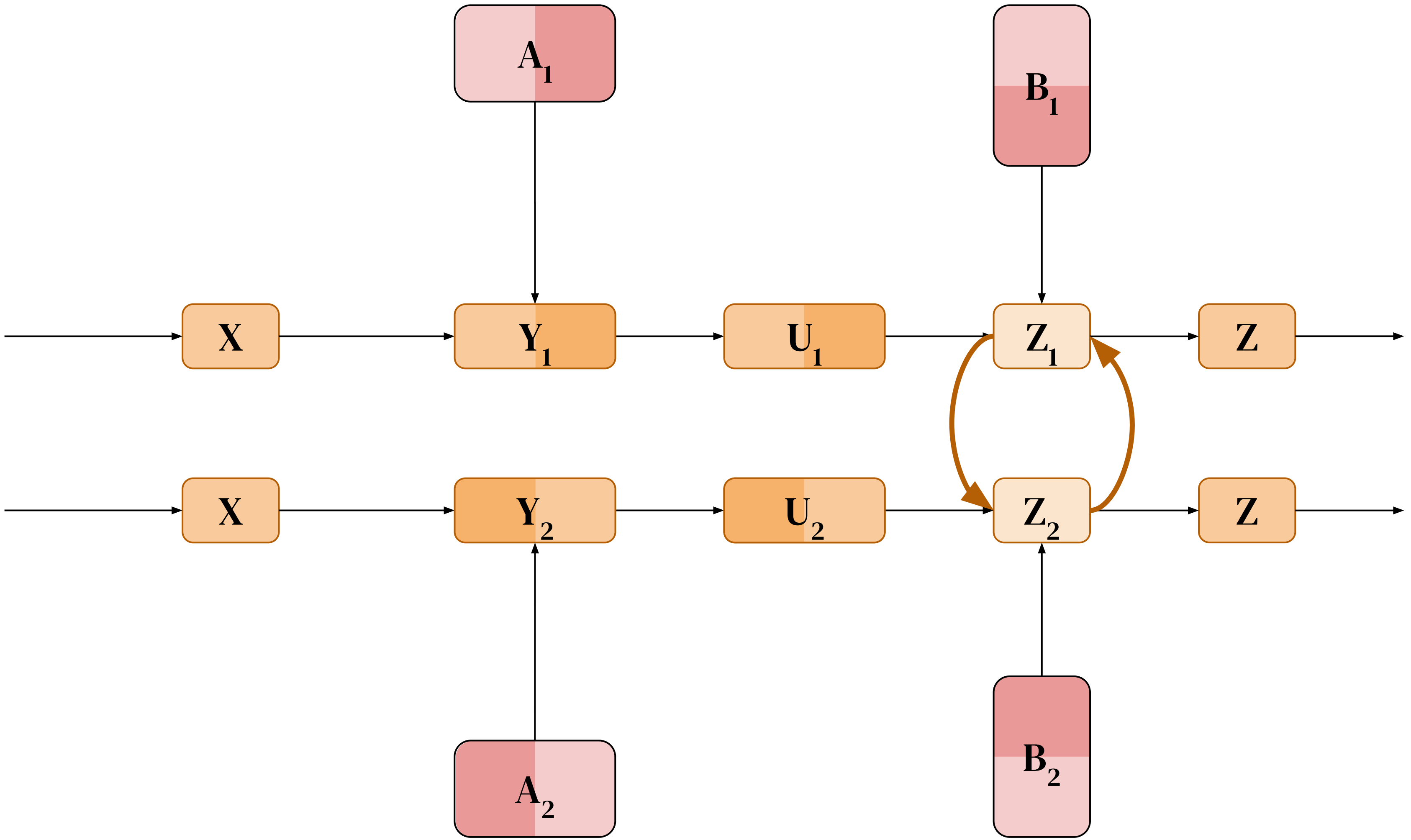
B_2: [$H_{2/2}$, H_1]

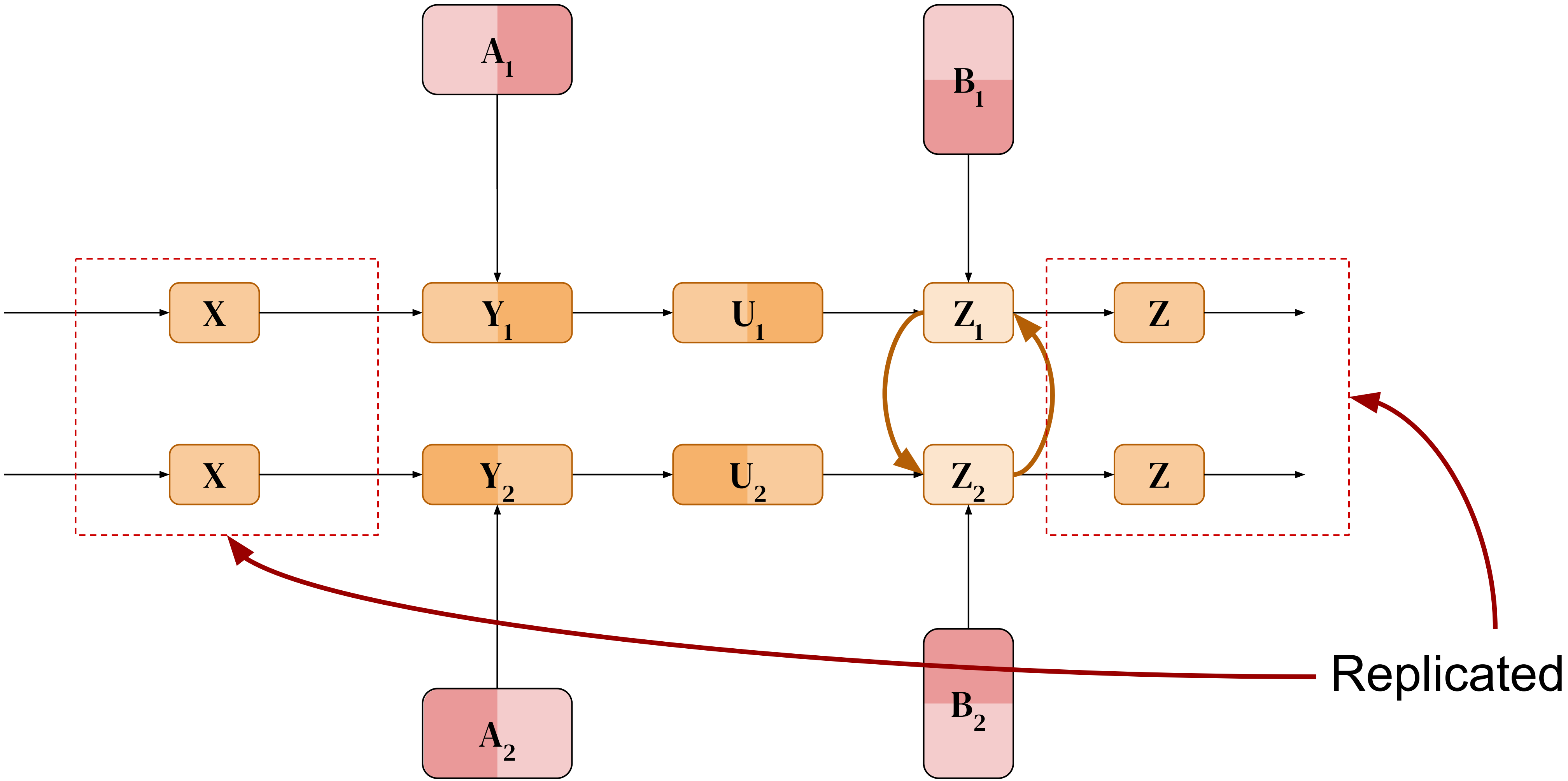
A_2: [H_1, $H_{2/2}$]



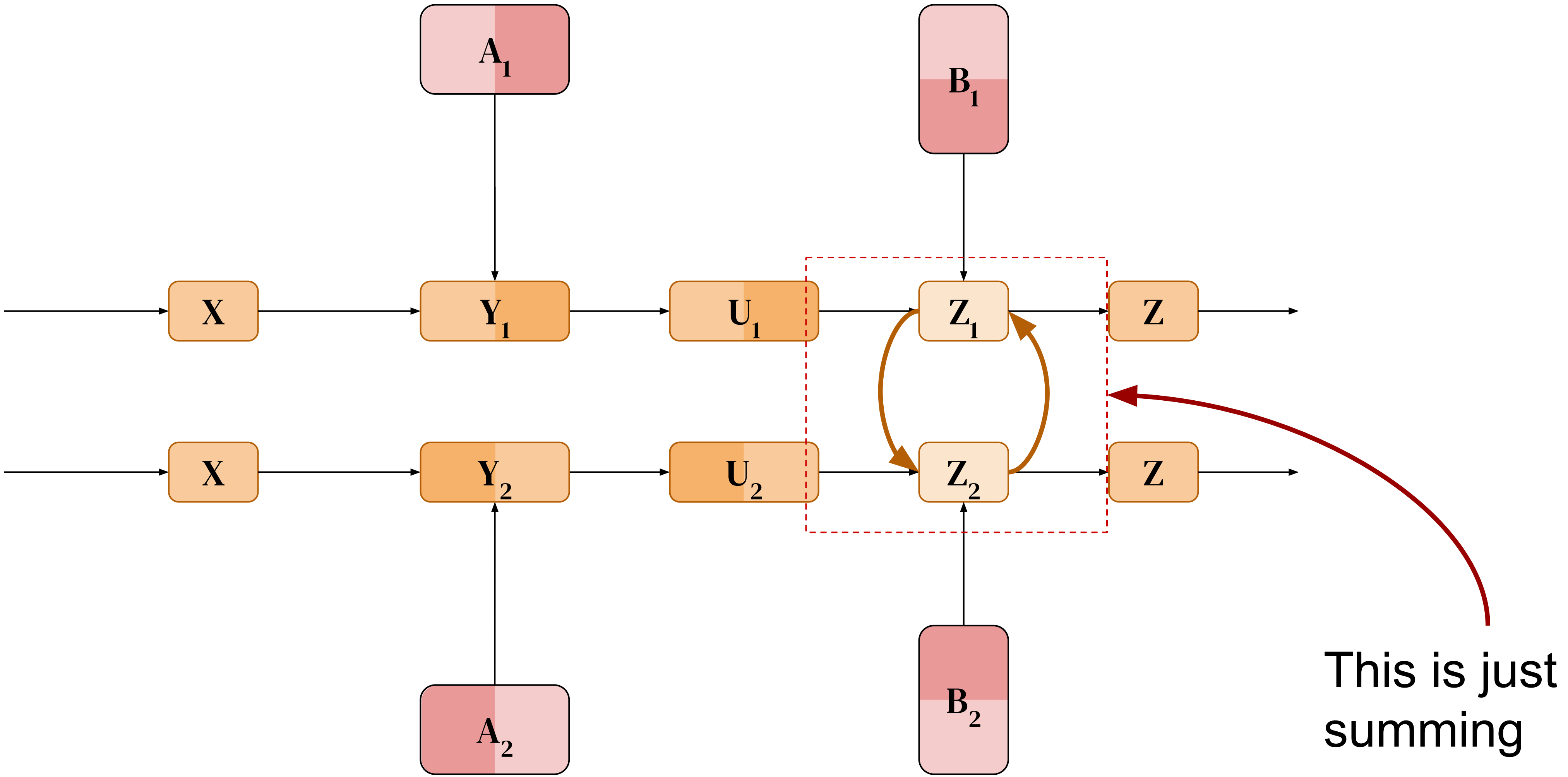
Let's forget about devices for a while

And rearrange the graph a bit

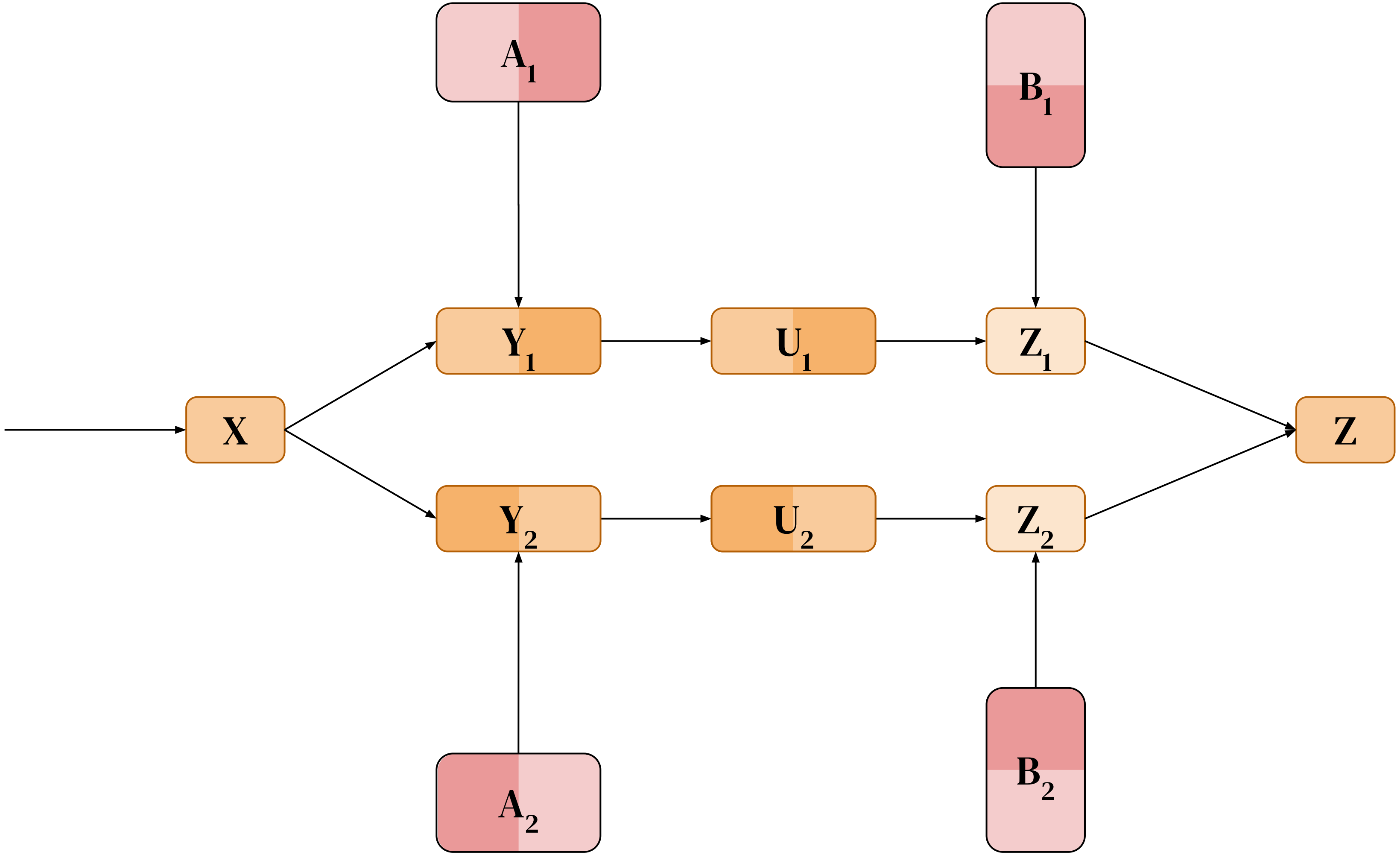




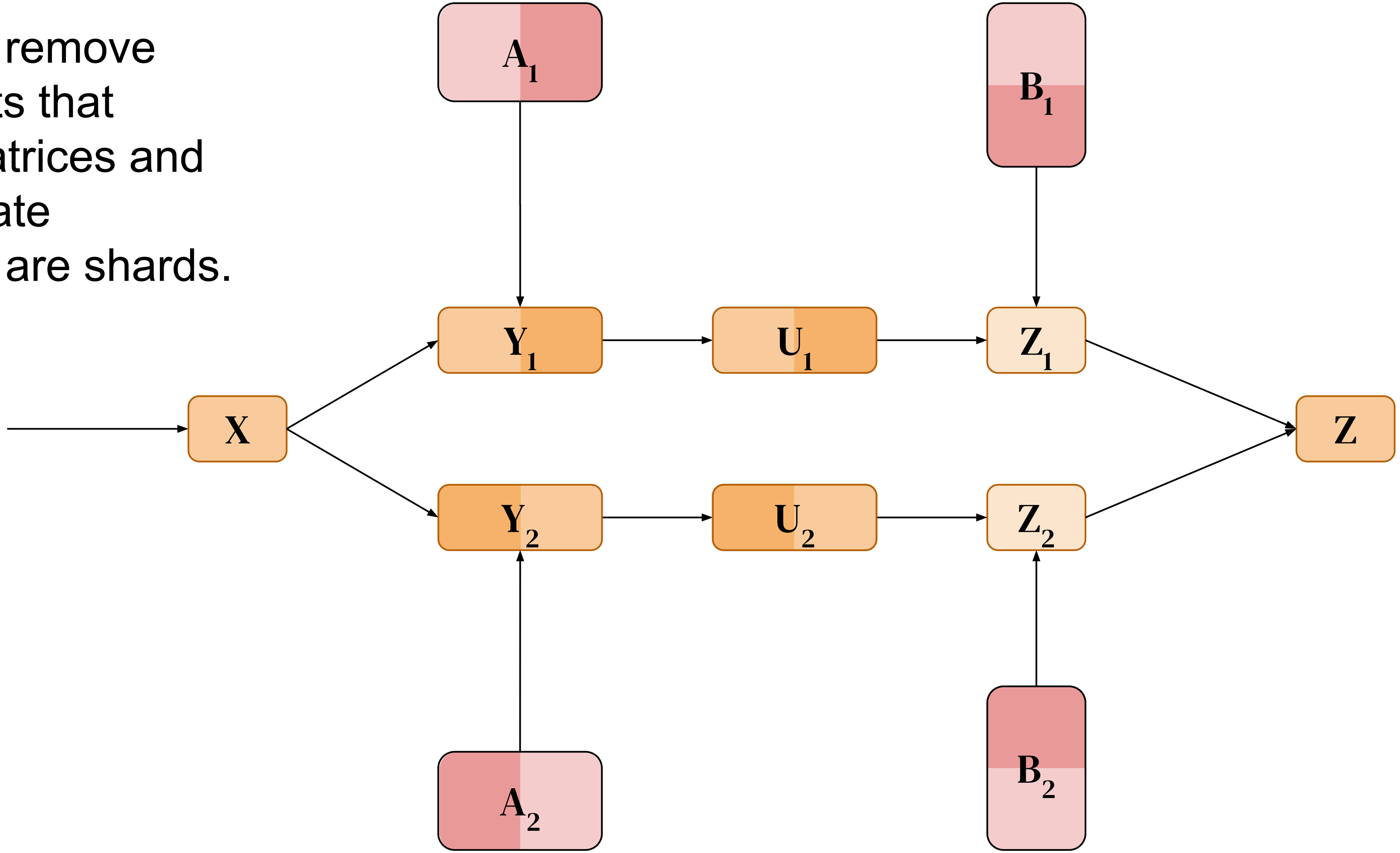
A1
B1
X
Y1
U1
Z1
Z
X
Y2
U2
Z2
Z
Replicated
A2
B2



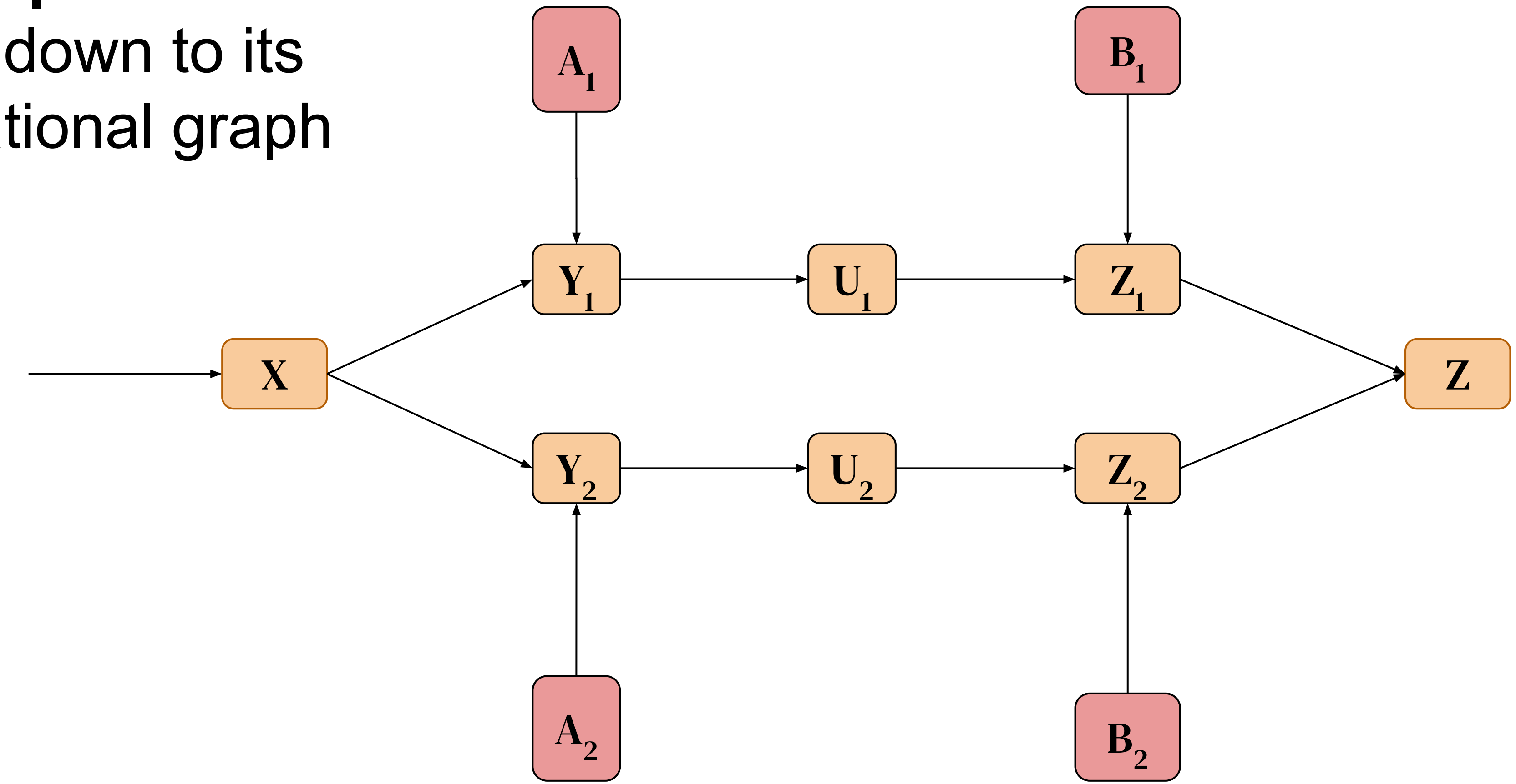
A_1
B_1
X
Y_1
U_1
Z_1
Z
X
Y_2
U_2
Z_2
Z
A_2
B_2
This is just summing

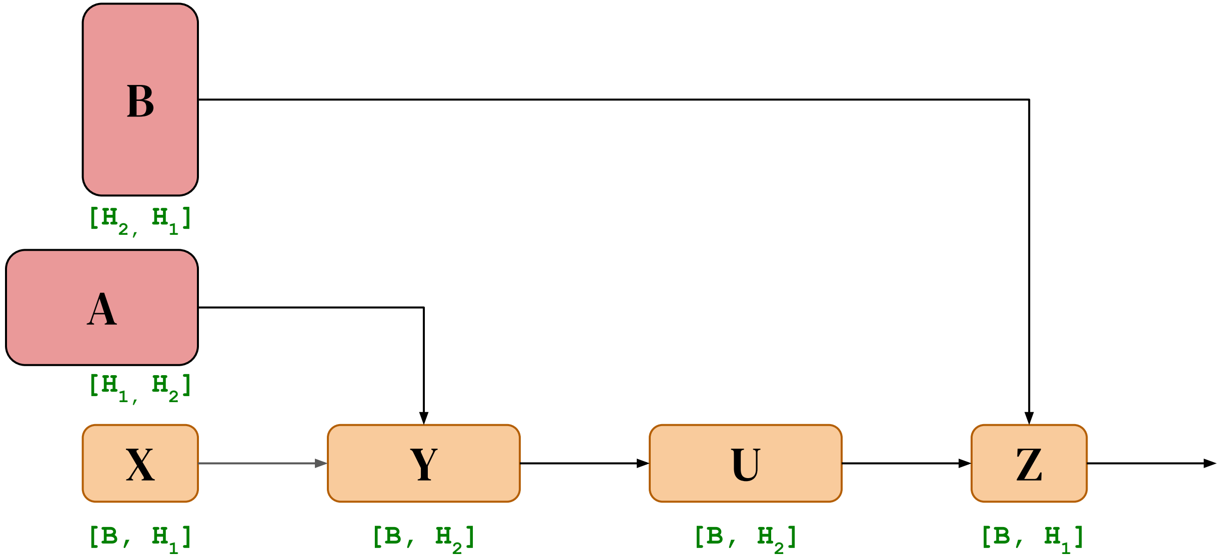

A1
B1
X
Y1
U1
Z1
Z
Y2
U2
Z2
A2
B2

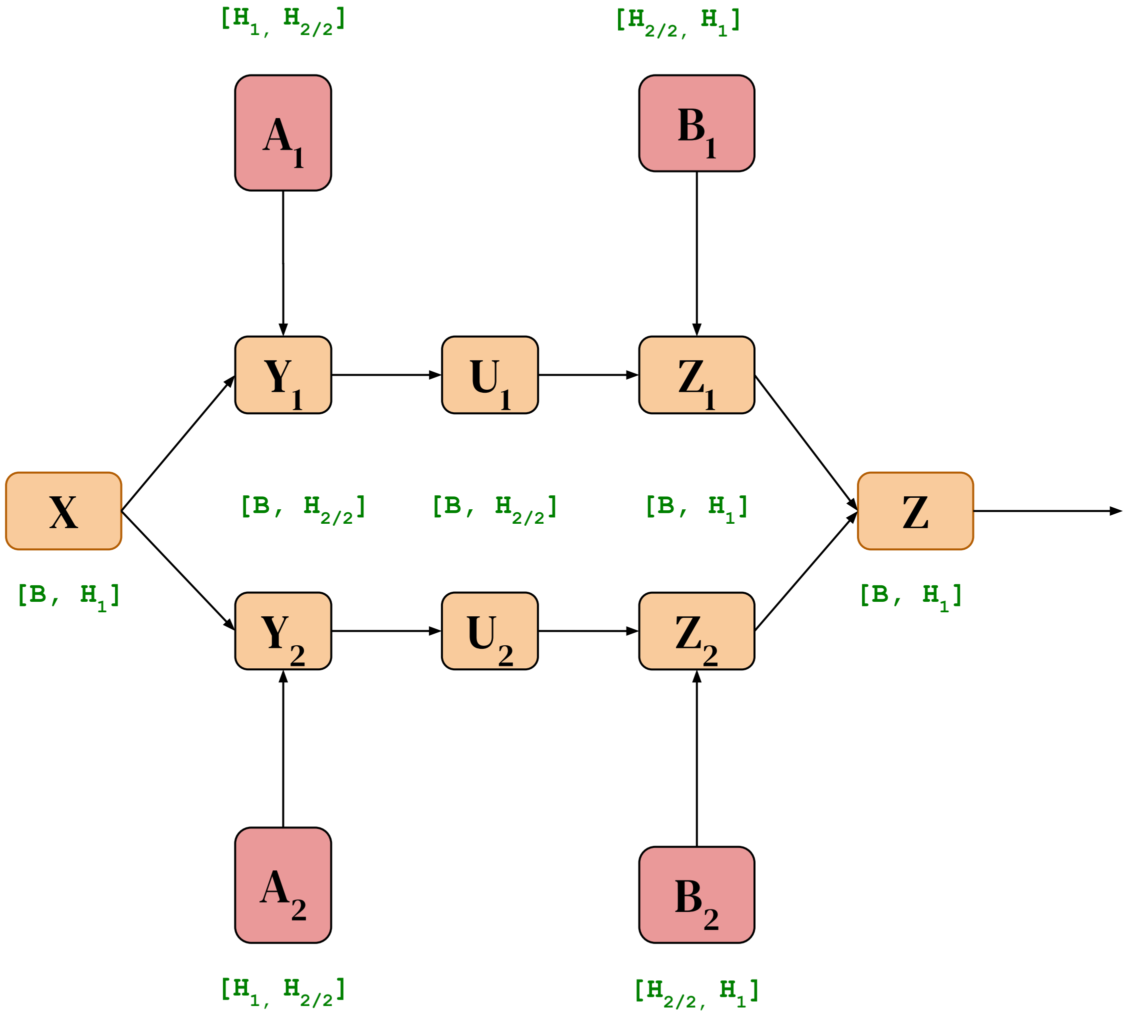
Let’s also remove visual hints that weight matrices and intermediate activation are shards.



Forward pass
stripped down to its
computational graph





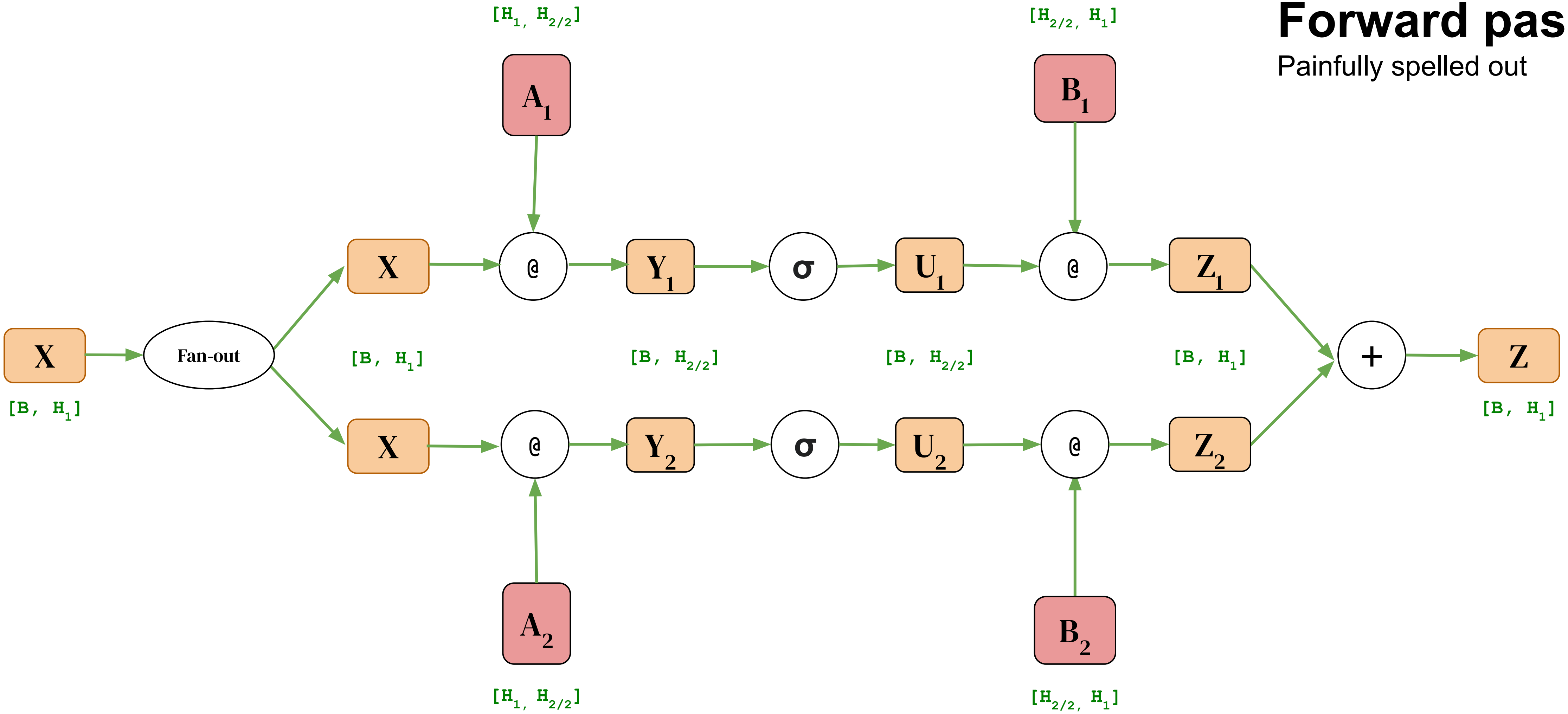


Forward pass

Two views on the computations in the forward pass.

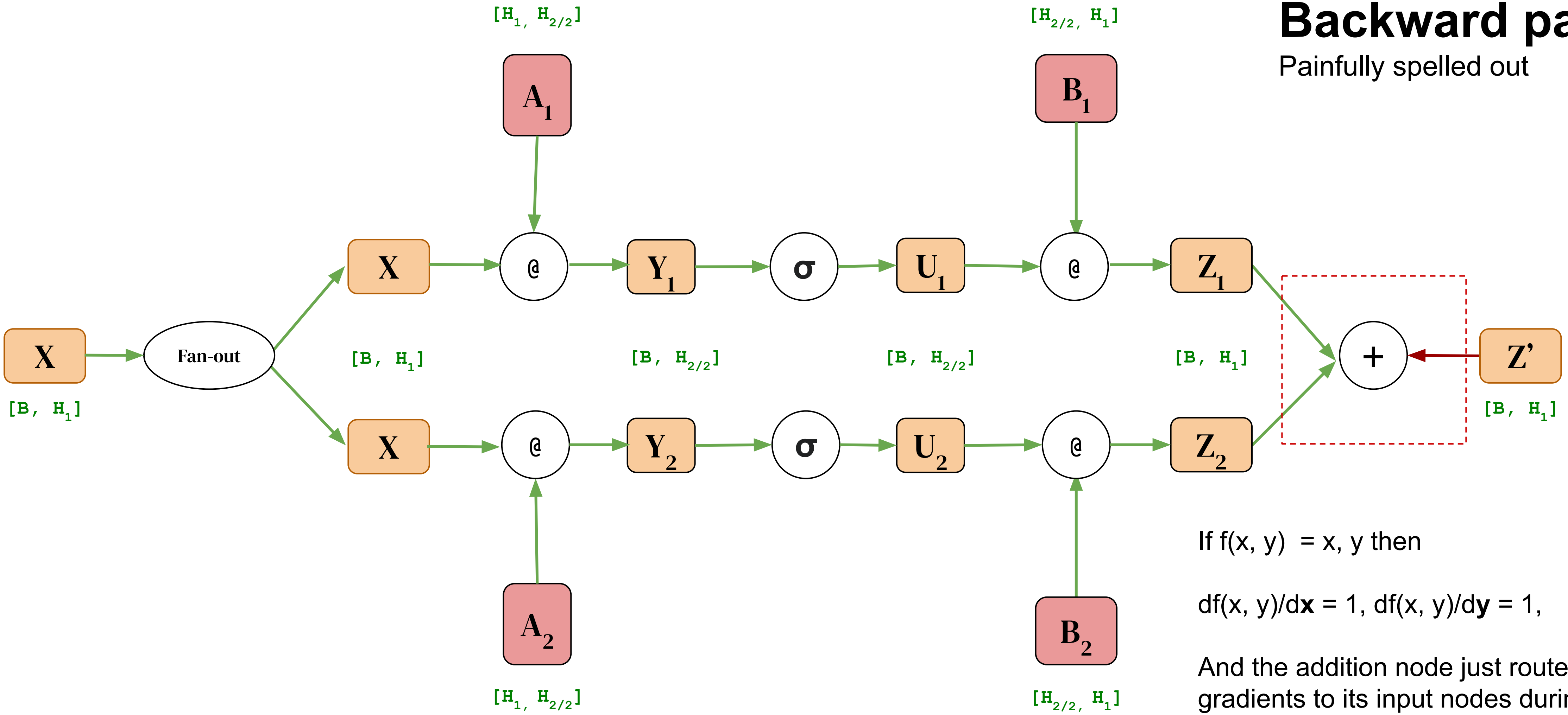
Forward pass

Painfully spelled out



Backward pass

Painfully spelled out



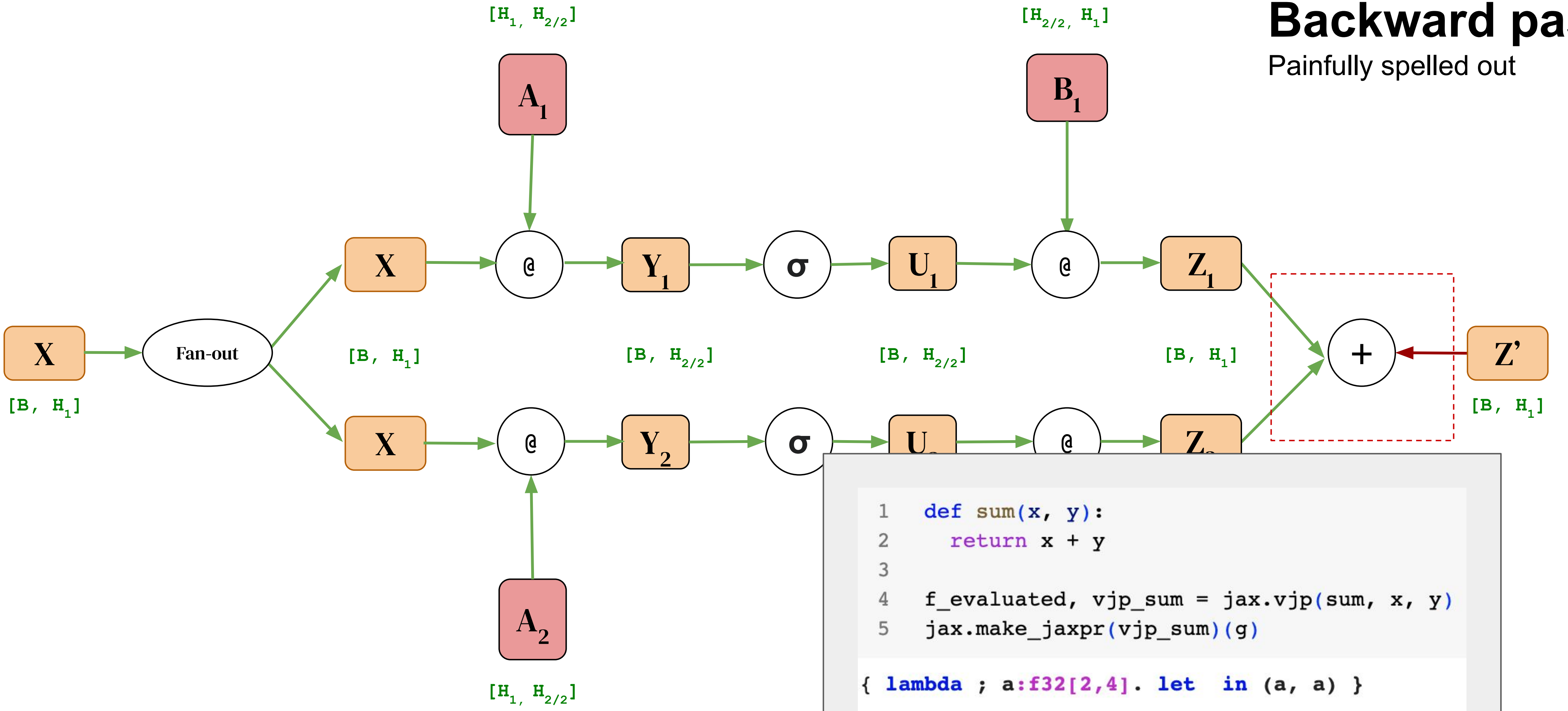
If f(x, y) = x, y then

df(x, y)/d**x** = 1, df(x, y)/d**y** = 1,

And the addition node just routes gradients to its input nodes during backward pass

Backward pass

Painfully spelled out



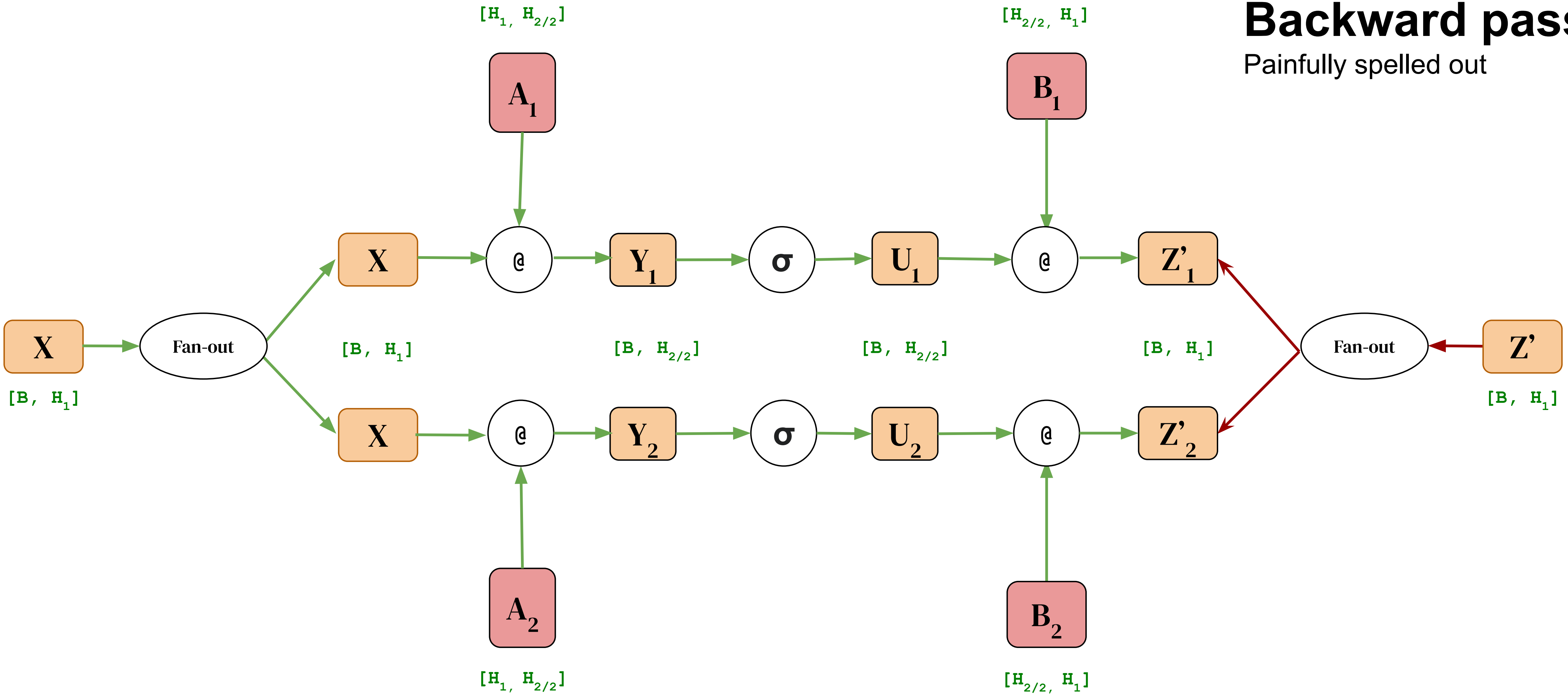

```python
def sum(x, y):
  return x + y

f_evaluated, vjp_sum = jax.vjp(sum, x, y)
jax.make_jaxpr(vjp_sum)(g)
```

```
{ lambda ; a:f32[2,4]. let  in (a, a) }
```

Backward pass

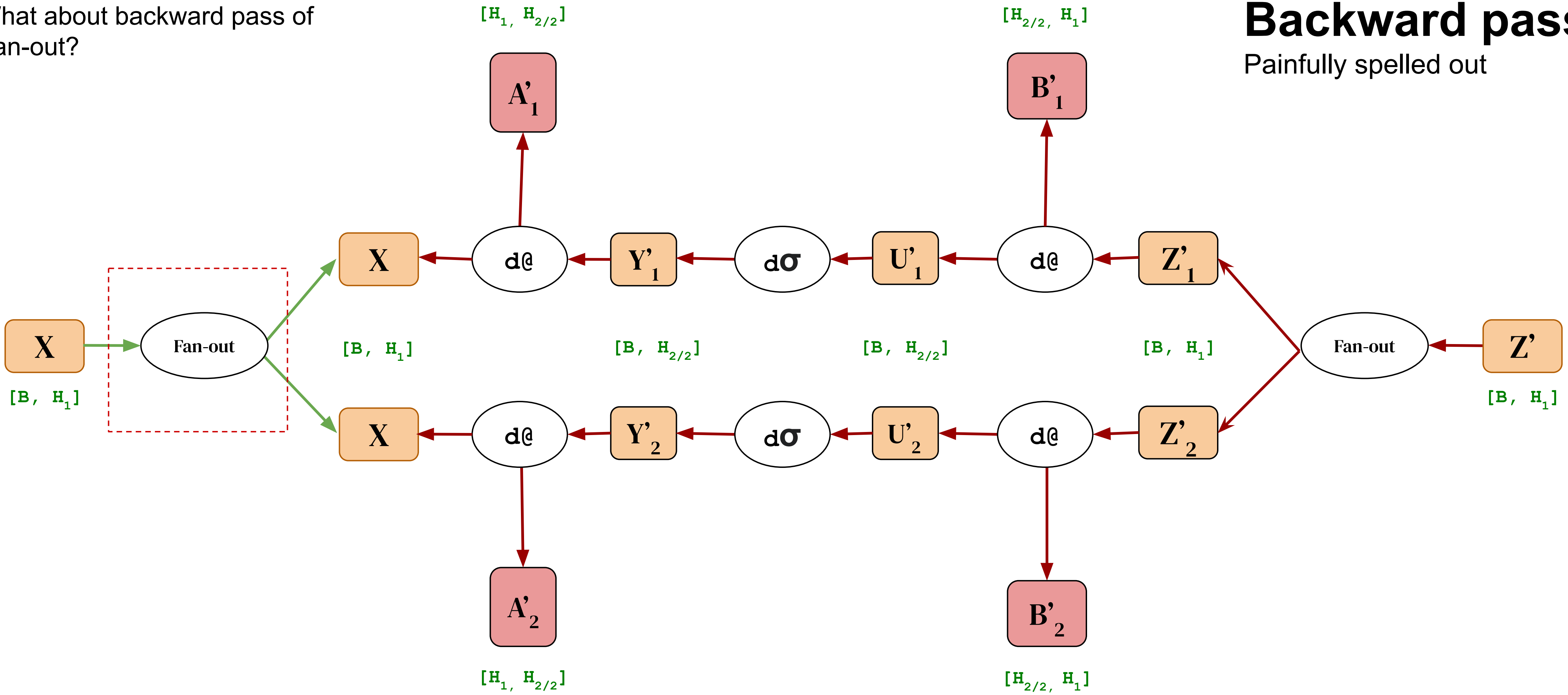
Painfully spelled out



What about backward pass of Fan-out?

Backward pass

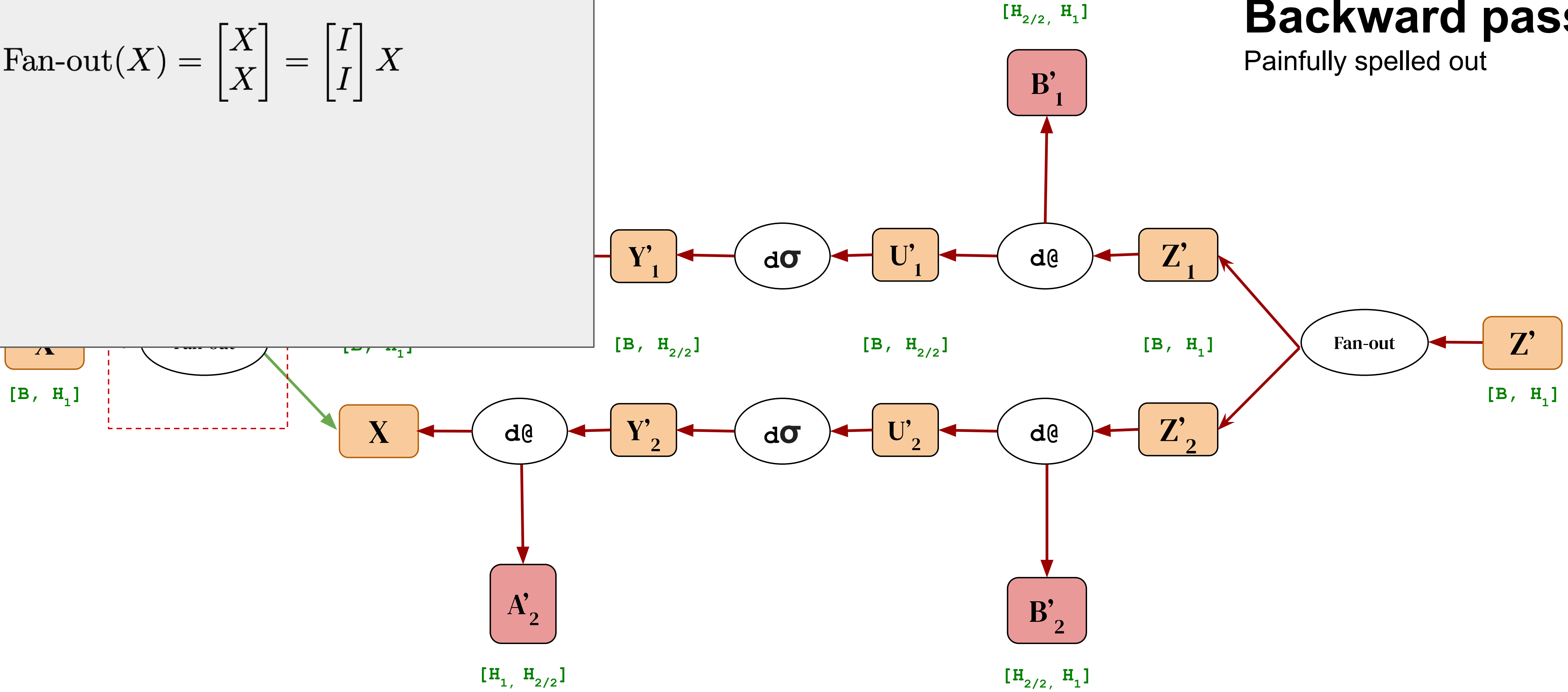
Painfully spelled out



Backward pass

Painfully spelled out

$$\text{Fan-out}(X) = \begin{bmatrix} X \\ X \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} X$$

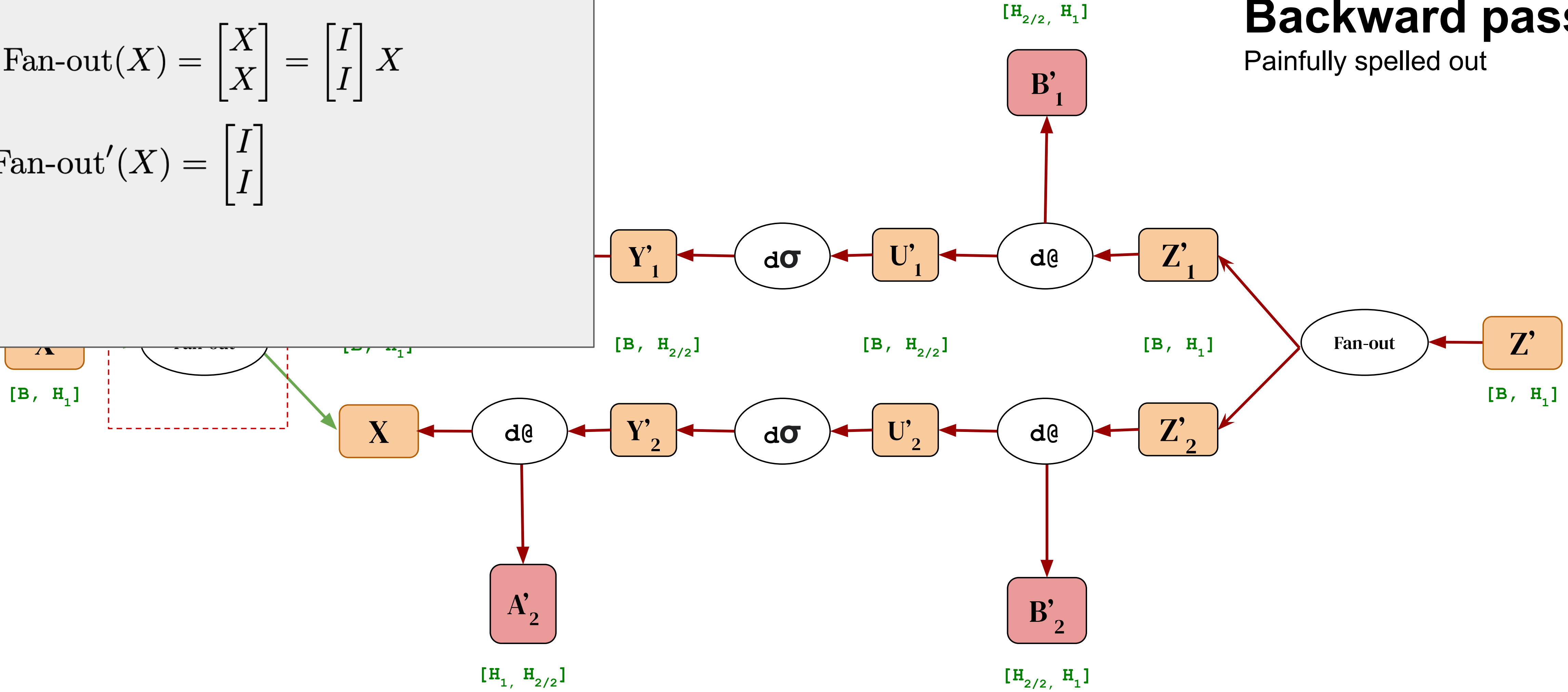


Backward pass

Painfully spelled out

$$\text{Fan-out}(X) = \begin{bmatrix} X \\ X \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} X$$

$$\text{Fan-out}'(X) = \begin{bmatrix} I \\ I \end{bmatrix}$$



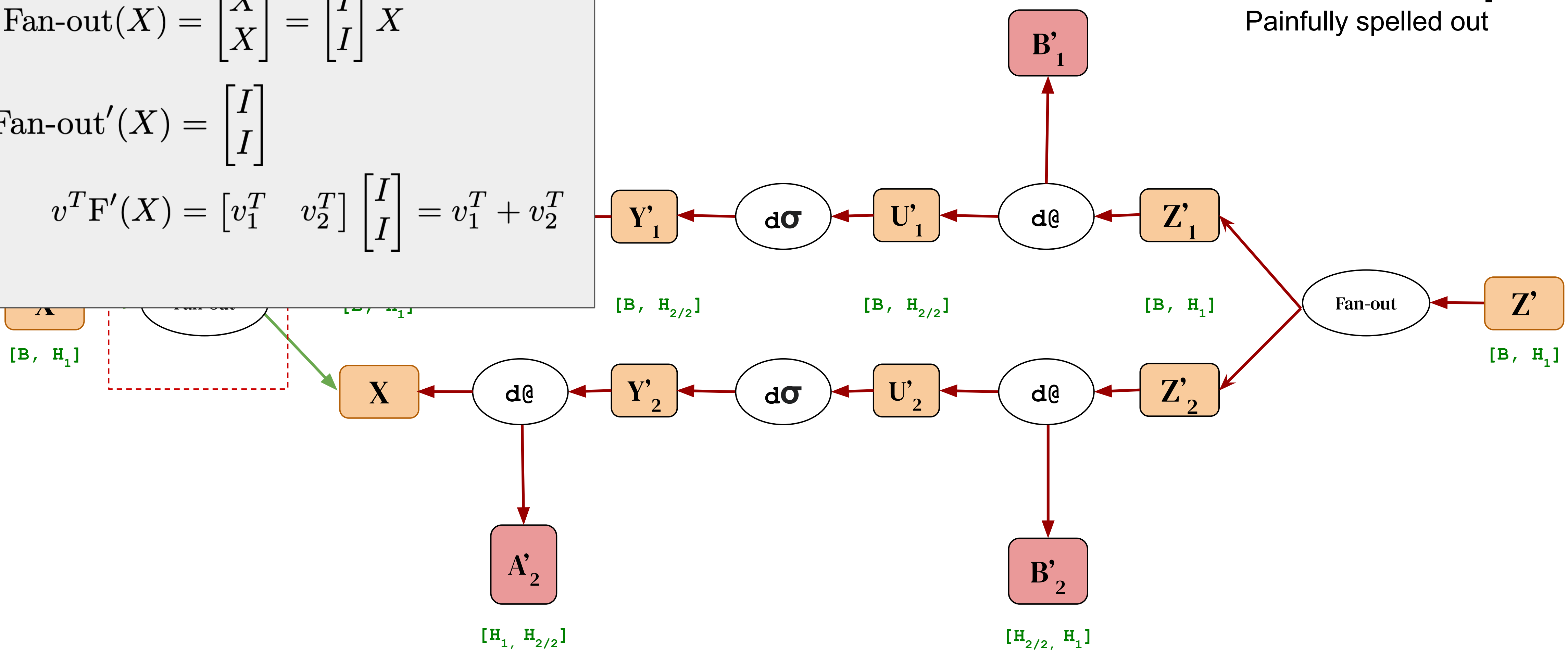
Backward pass

Painfully spelled out

$$\text{Fan-out}(X) = \begin{bmatrix} X \\ X \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} X$$

$$\text{Fan-out}'(X) = \begin{bmatrix} I \\ I \end{bmatrix}$$

$$v^T \mathrm{F}'(X) = \begin{bmatrix} v_1^T & v_2^T \end{bmatrix} \begin{bmatrix} I \\ I \end{bmatrix} = v_1^T + v_2^T$$



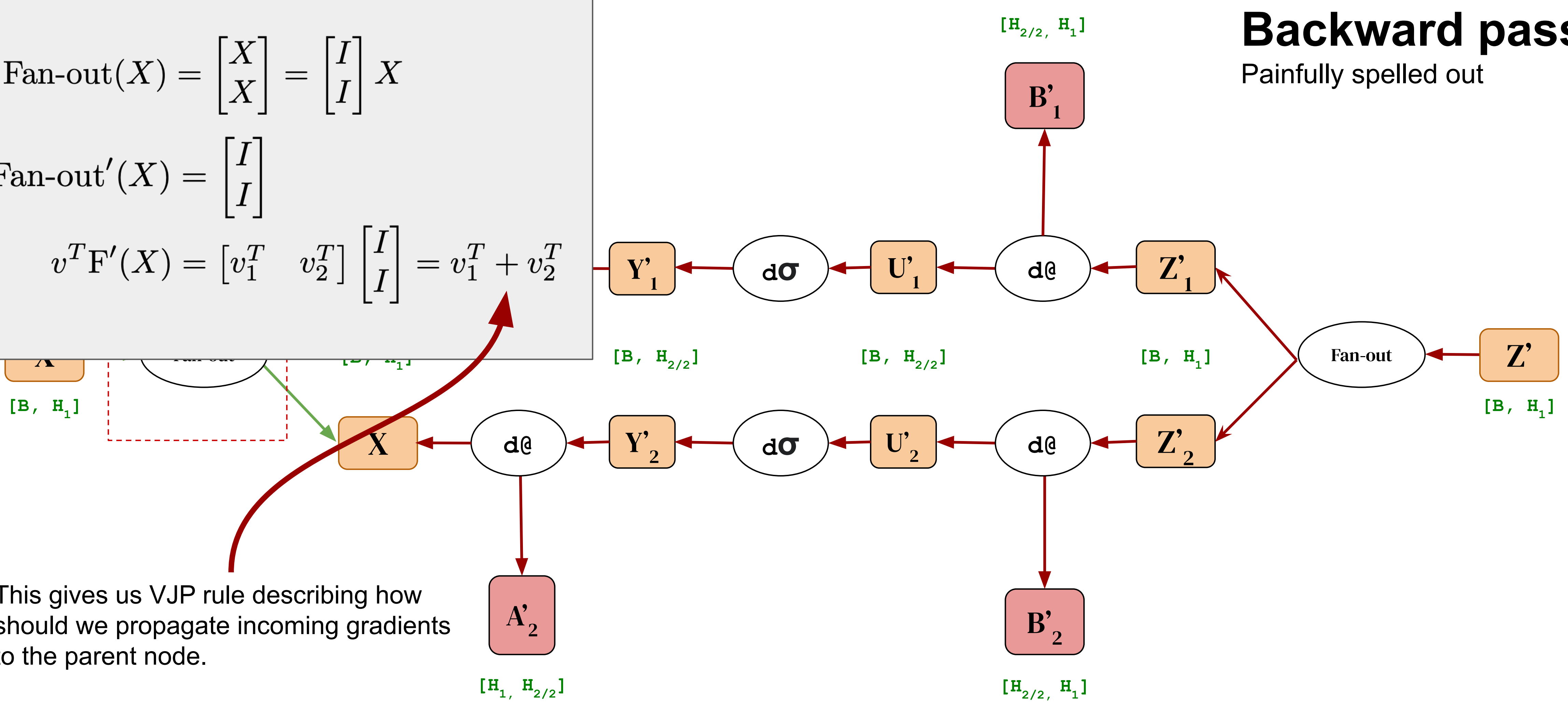
Backward pass

Painfully spelled out

$$\text{Fan-out}(X) = \begin{bmatrix} X \\ X \end{bmatrix} = \begin{bmatrix} I \\ I \end{bmatrix} X$$

$$\text{Fan-out}'(X) = \begin{bmatrix} I \\ I \end{bmatrix}$$

$$v^T \mathrm{F}'(X) = \begin{bmatrix} v_1^T & v_2^T \end{bmatrix} \begin{bmatrix} I \\ I \end{bmatrix} = v_1^T + v_2^T$$



This gives us VJP rule describing how should we propagate incoming gradients to the parent node.

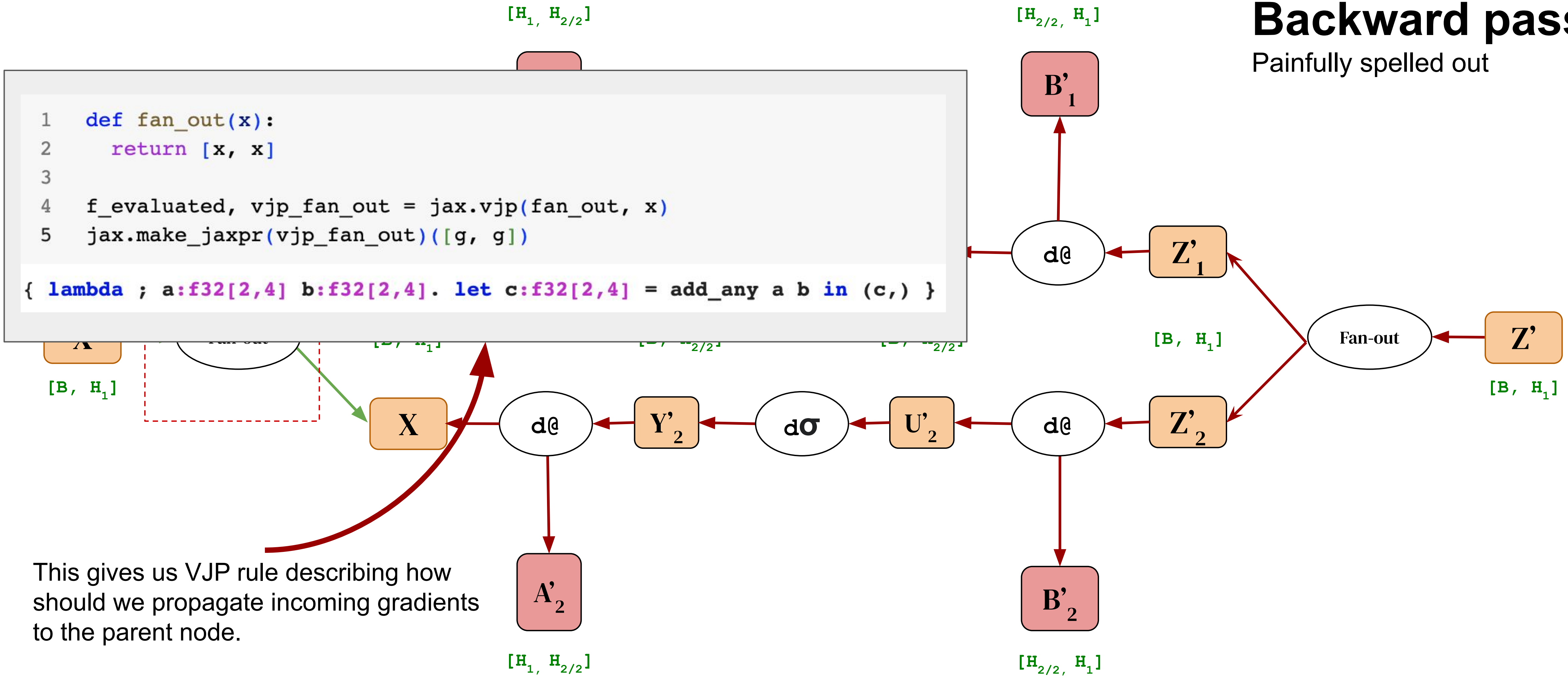
Backward pass

Painfully spelled out

```
def fan_out(x):
  return [x, x]

f_evaluated, vjp_fan_out = jax.vjp(fan_out, x)
jax.make_jaxpr(vjp_fan_out)([g, g])
```

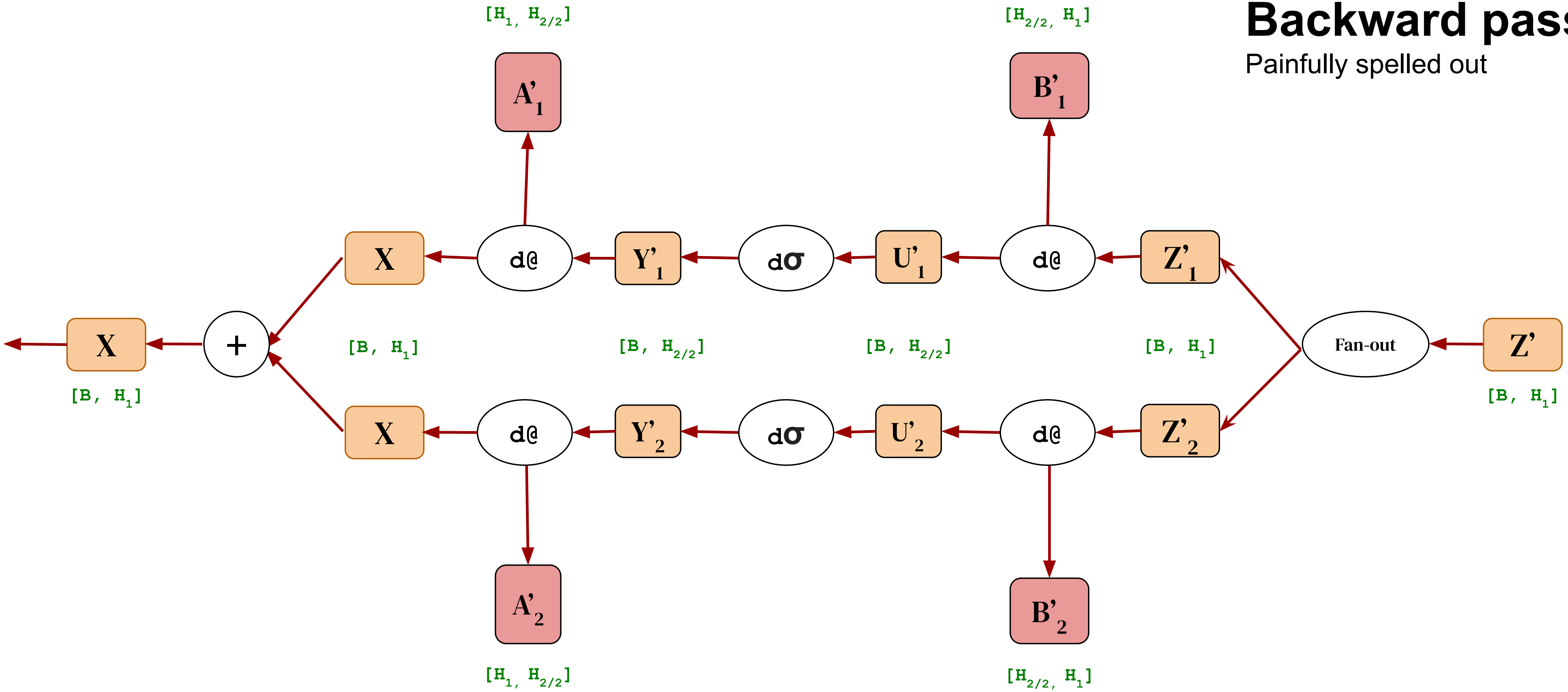
```
{ lambda ; a:f32[2,4] b:f32[2,4]. let c:f32[2,4] = add_any a b in (c,) }
```

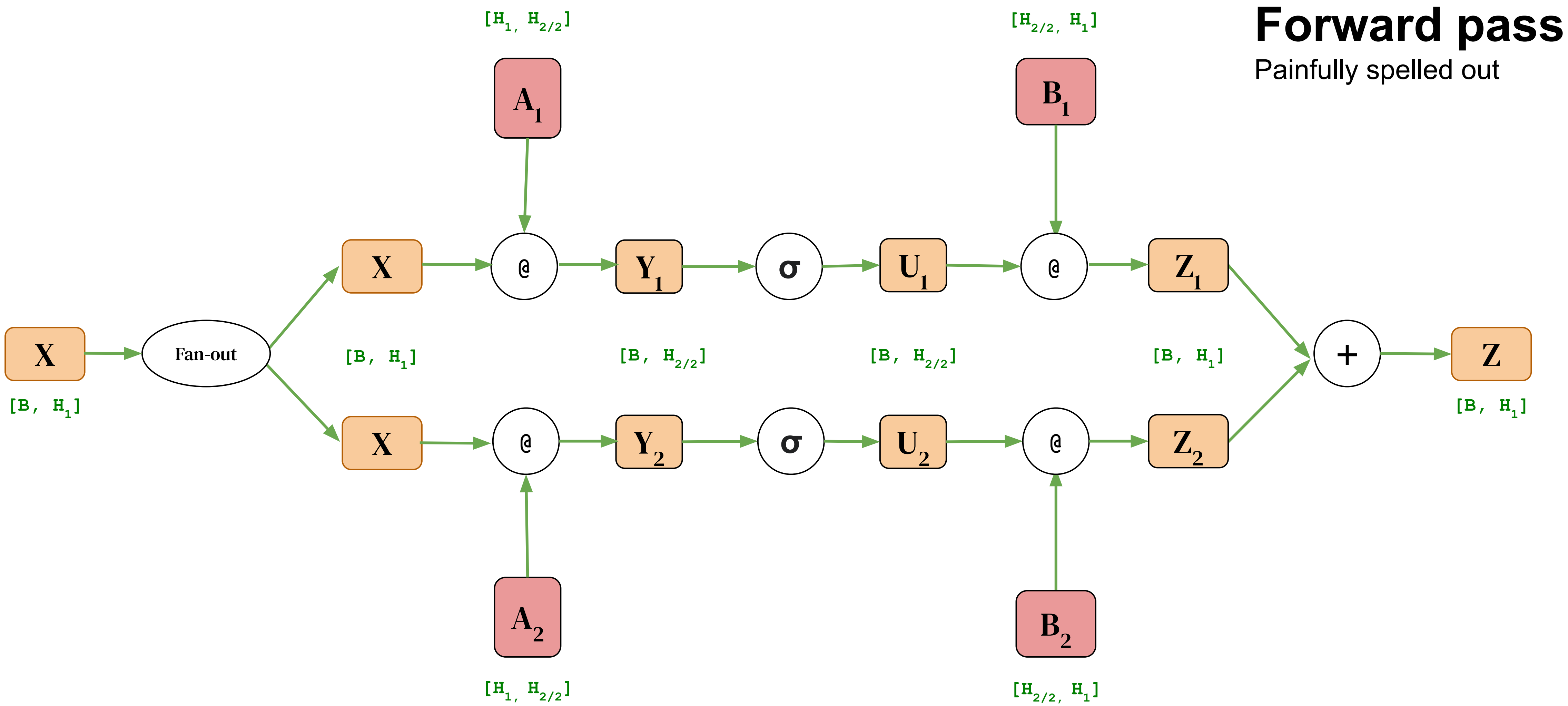


This gives us VJP rule describing how should we propagate incoming gradients to the parent node.

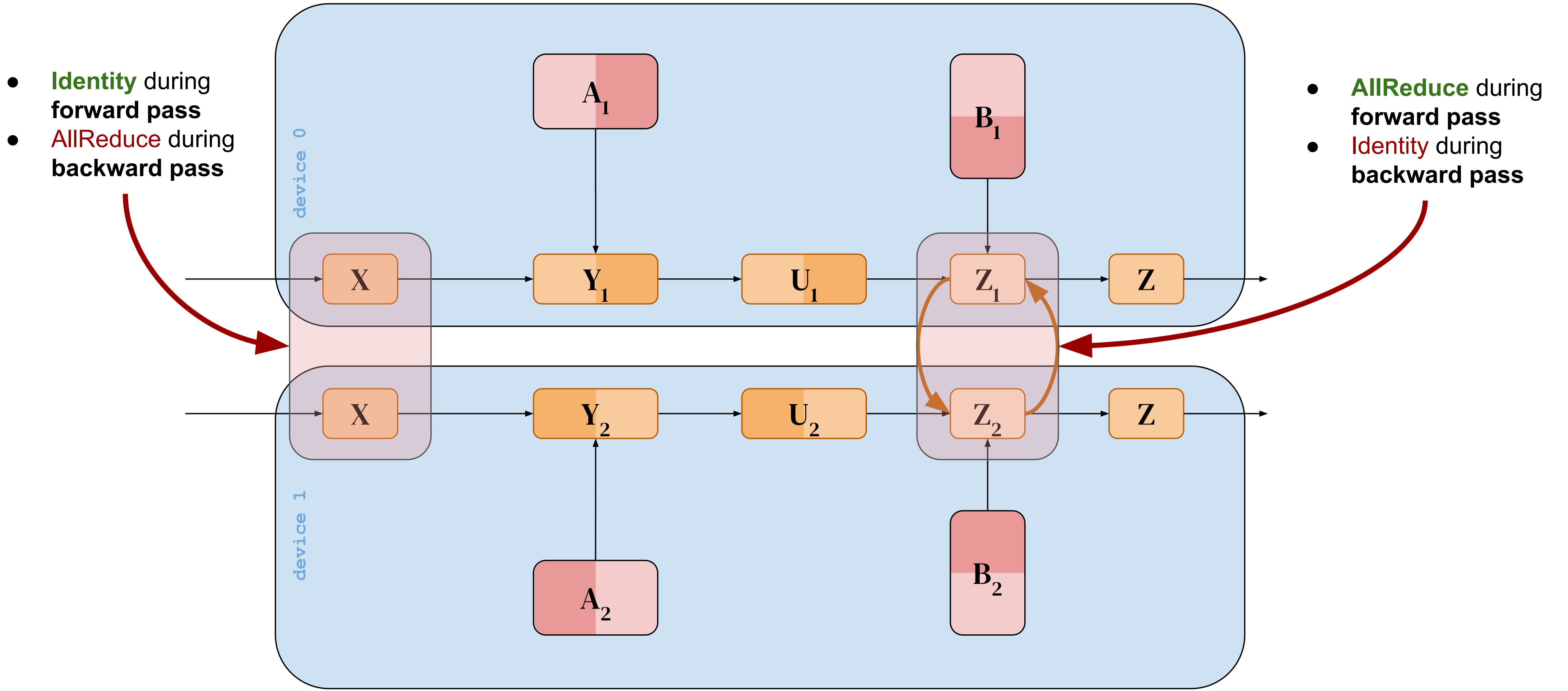
Backward pass

Painfully spelled out





Forward pass
Painfully spelled out
[H1, H2/2]
A1
[H2/2, H1]
B1
X
Fan-out
[B, H1]
X
@
Y1
σ
U1
@
Z1
+
Z
[B, H1]
[B, H2/2]
[B, H2/2]
[B, H1]
[B, H1]
X
@
Y2
σ
U2
@
Z2
A2
[H1, H2/2]
B2
[H2/2, H1]

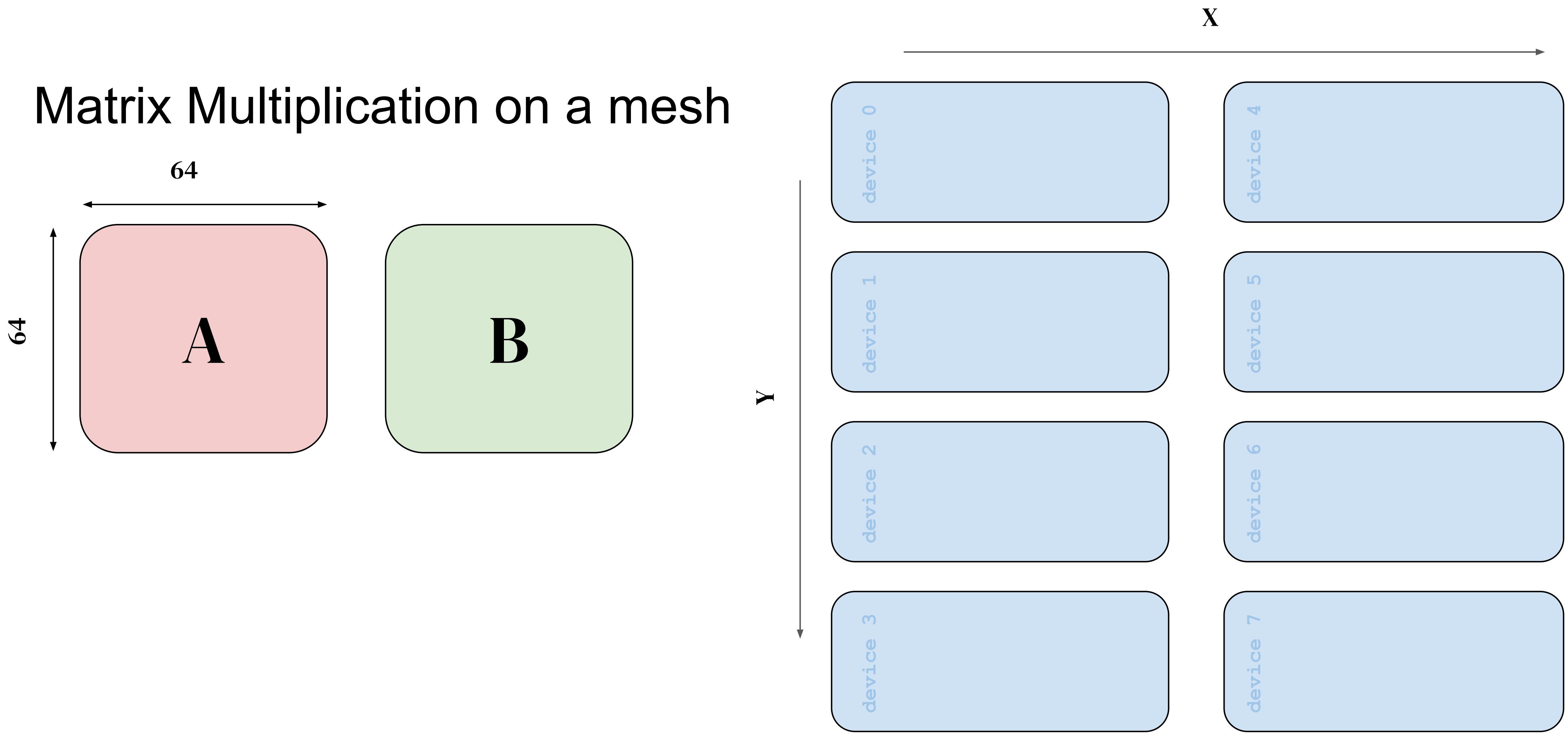


Identity during forward pass
AllReduce during backward pass
AllReduce during forward pass
Identity during backward pass
device 0
device 1
X
A_1
Y_1
U_1
B_1
Z_1
Z
X
A_2
Y_2
U_2
B_2
Z_2
Z

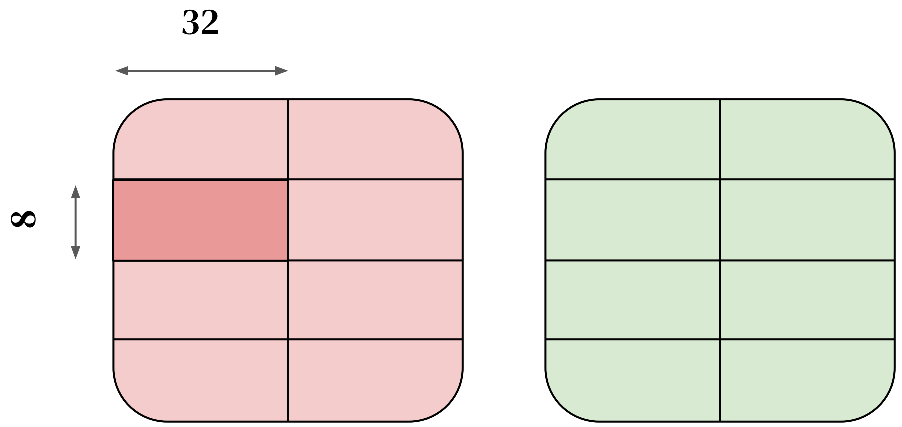
Sharded MLP - Summary

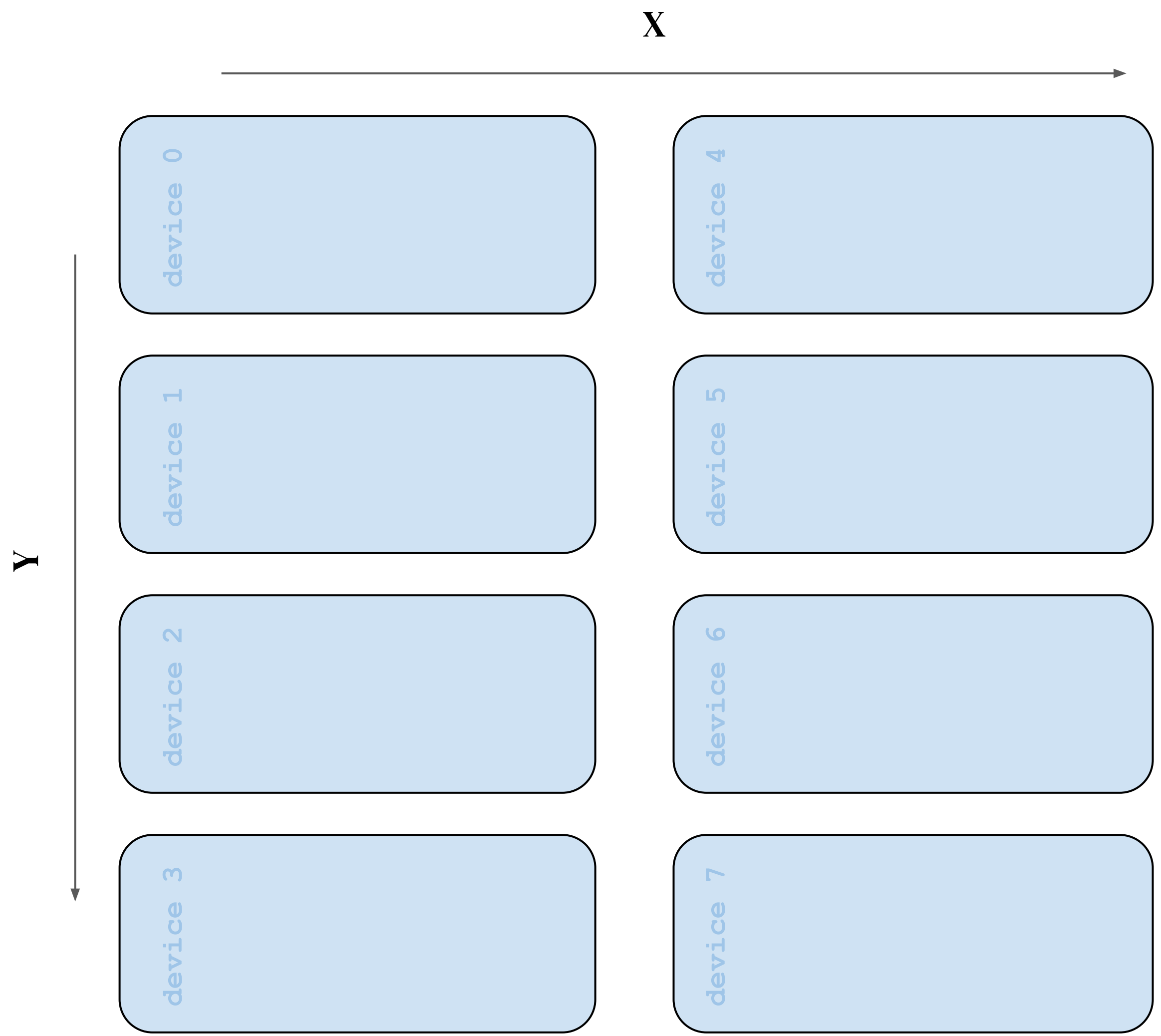
- Megatron sharding is a tensor parallelism technique, that avoids unnecessary communication between **N** devices.
- It can be applied to shard attention layers (splitting Q,K,V on heads).
- Reference: Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

Matrix Multiplication on a mesh

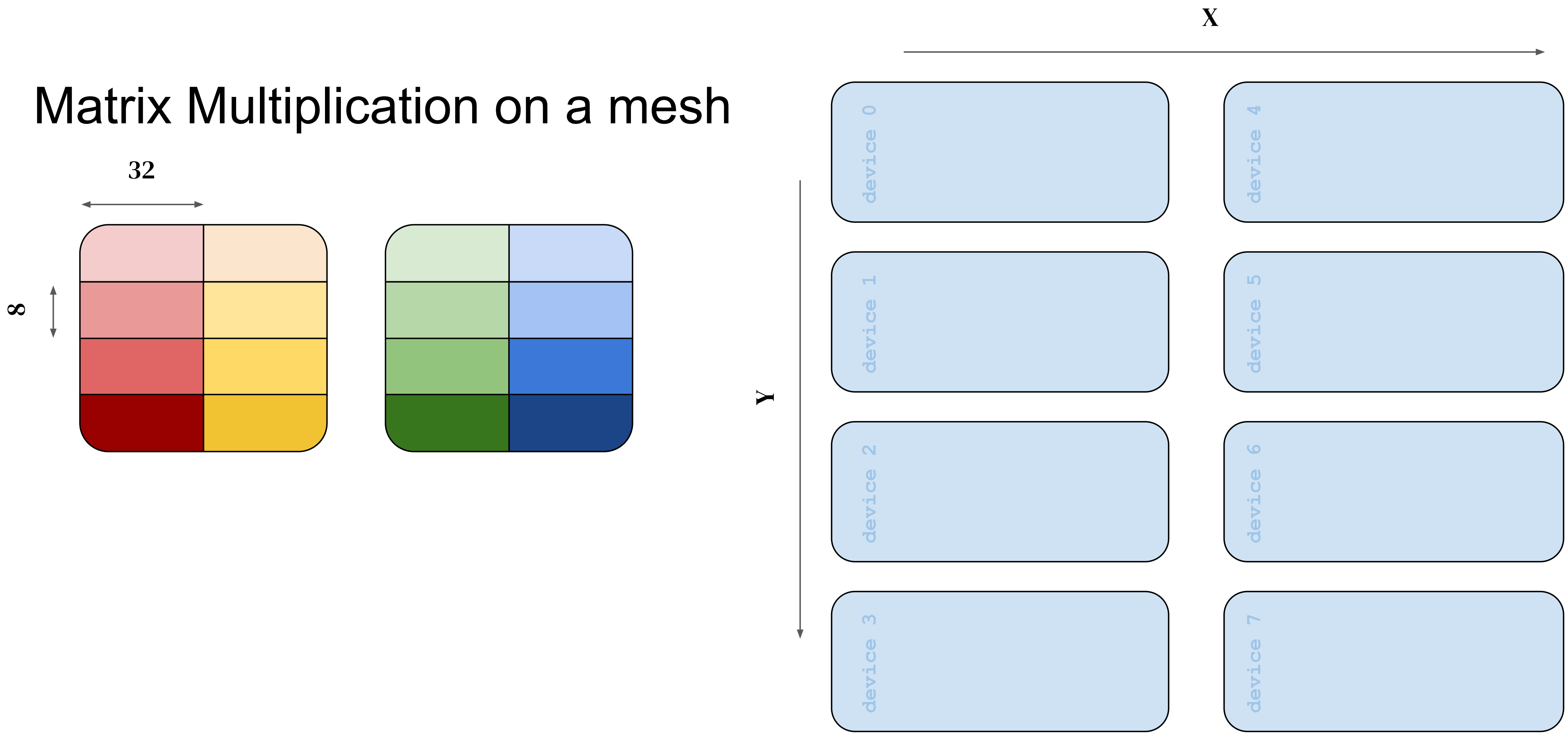


Matrix Multiplication on a mesh

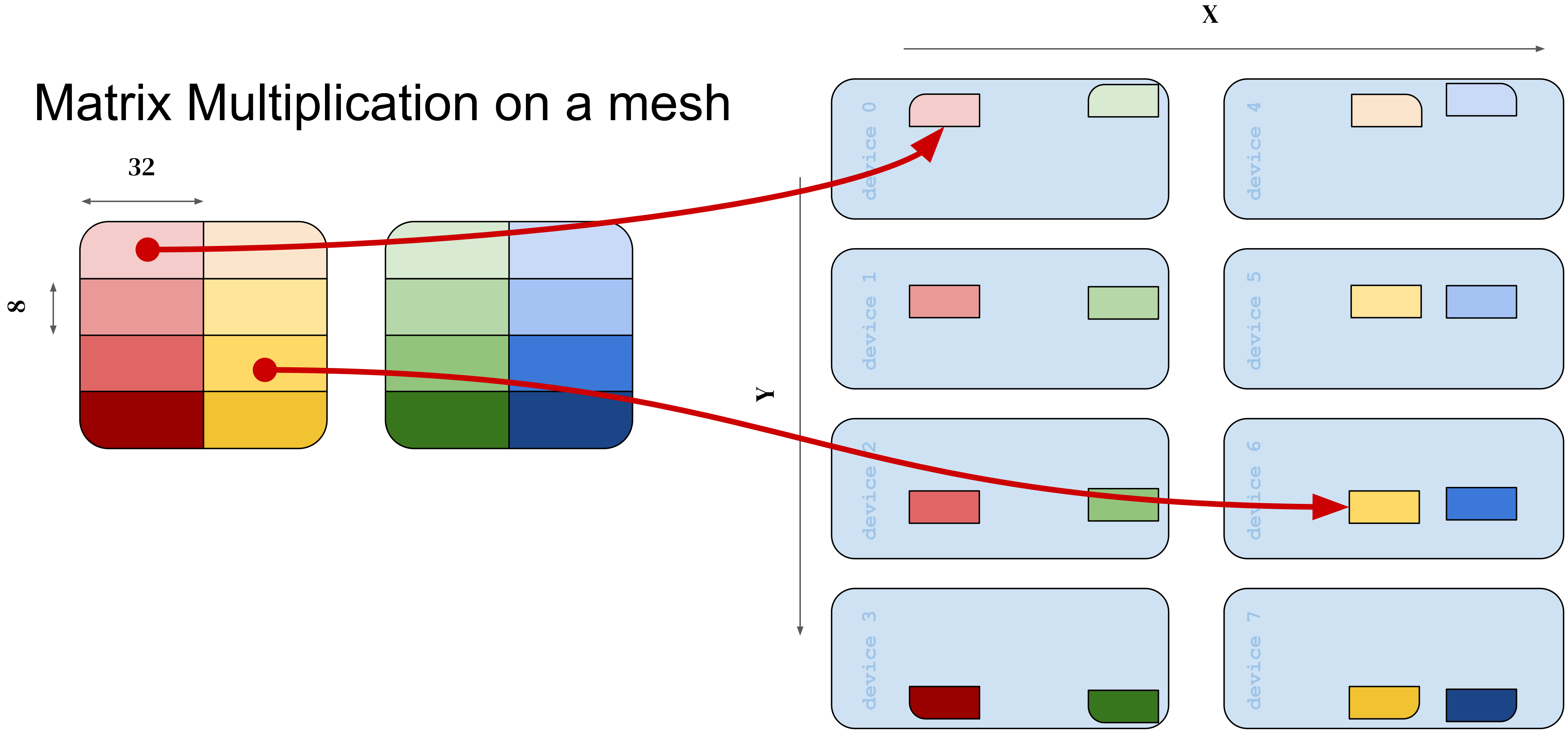




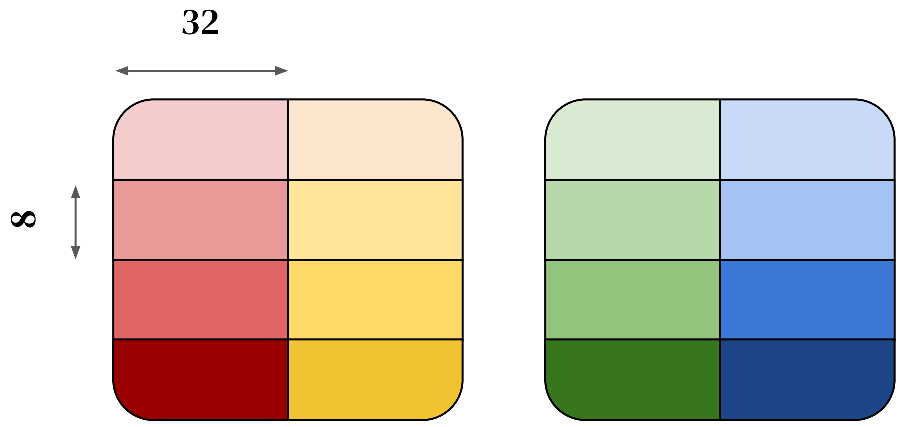
Matrix Multiplication on a mesh

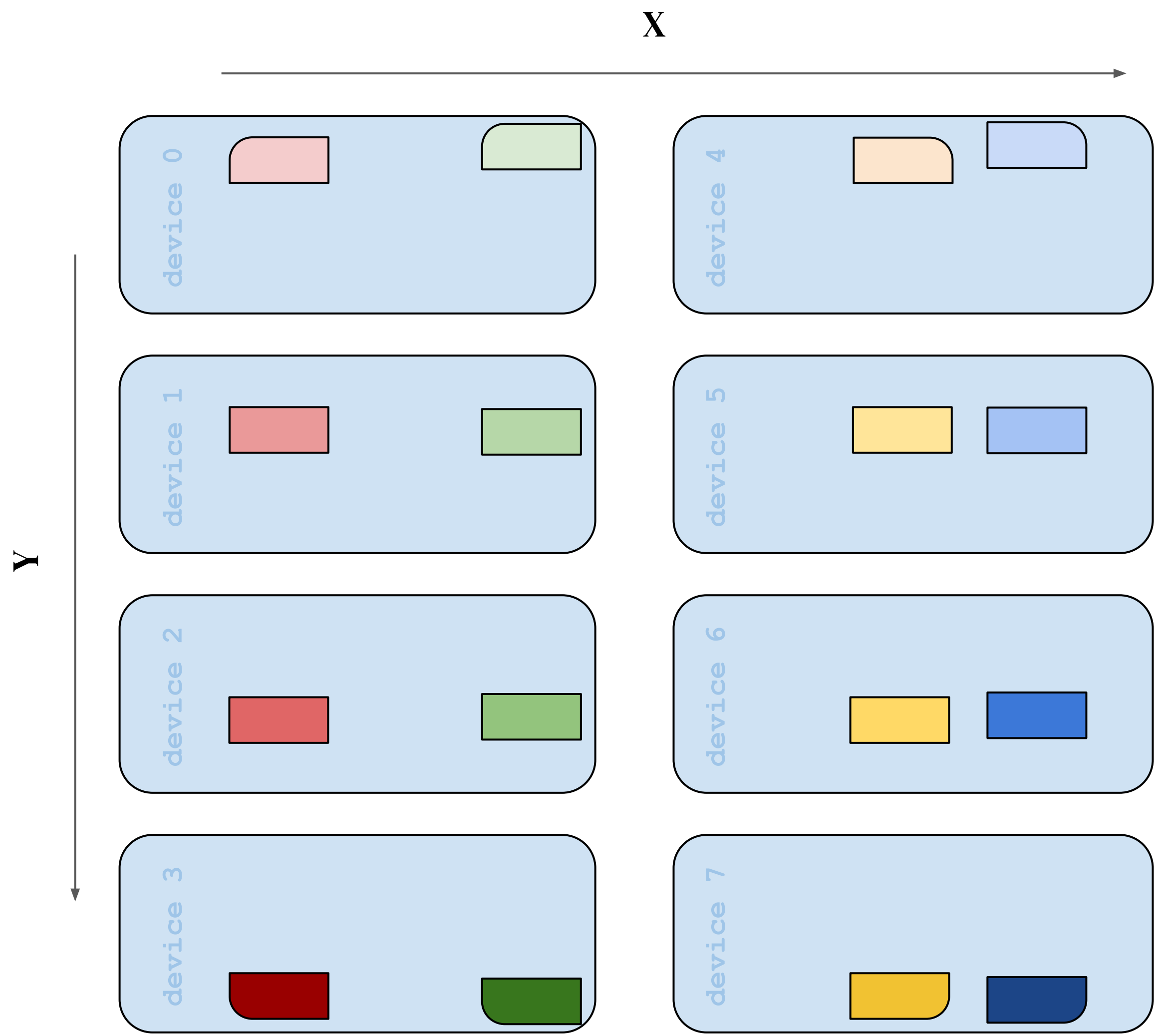


Matrix Multiplication on a mesh

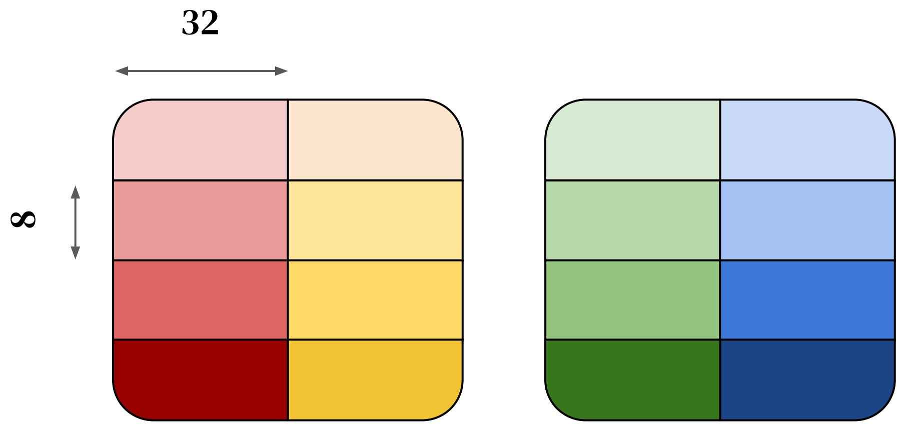


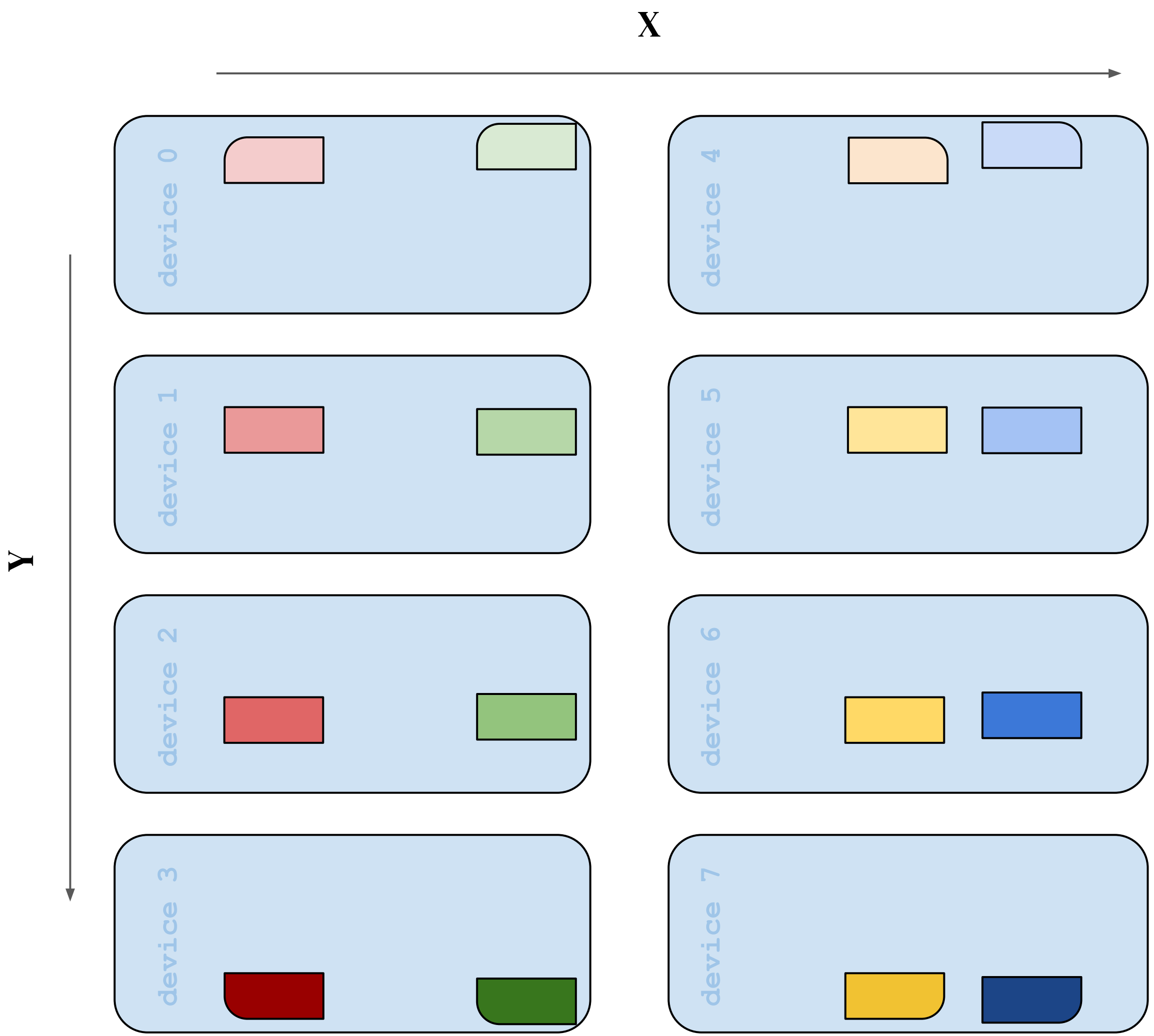
… but how do we multiply?



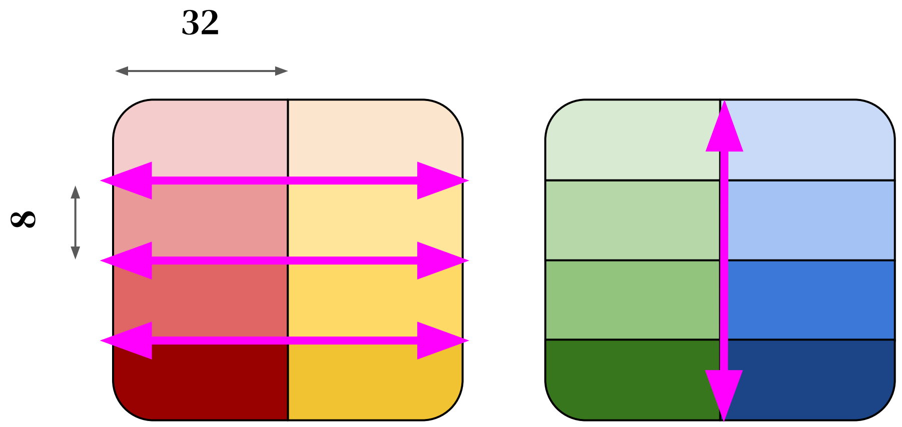


AllGather to the rescue

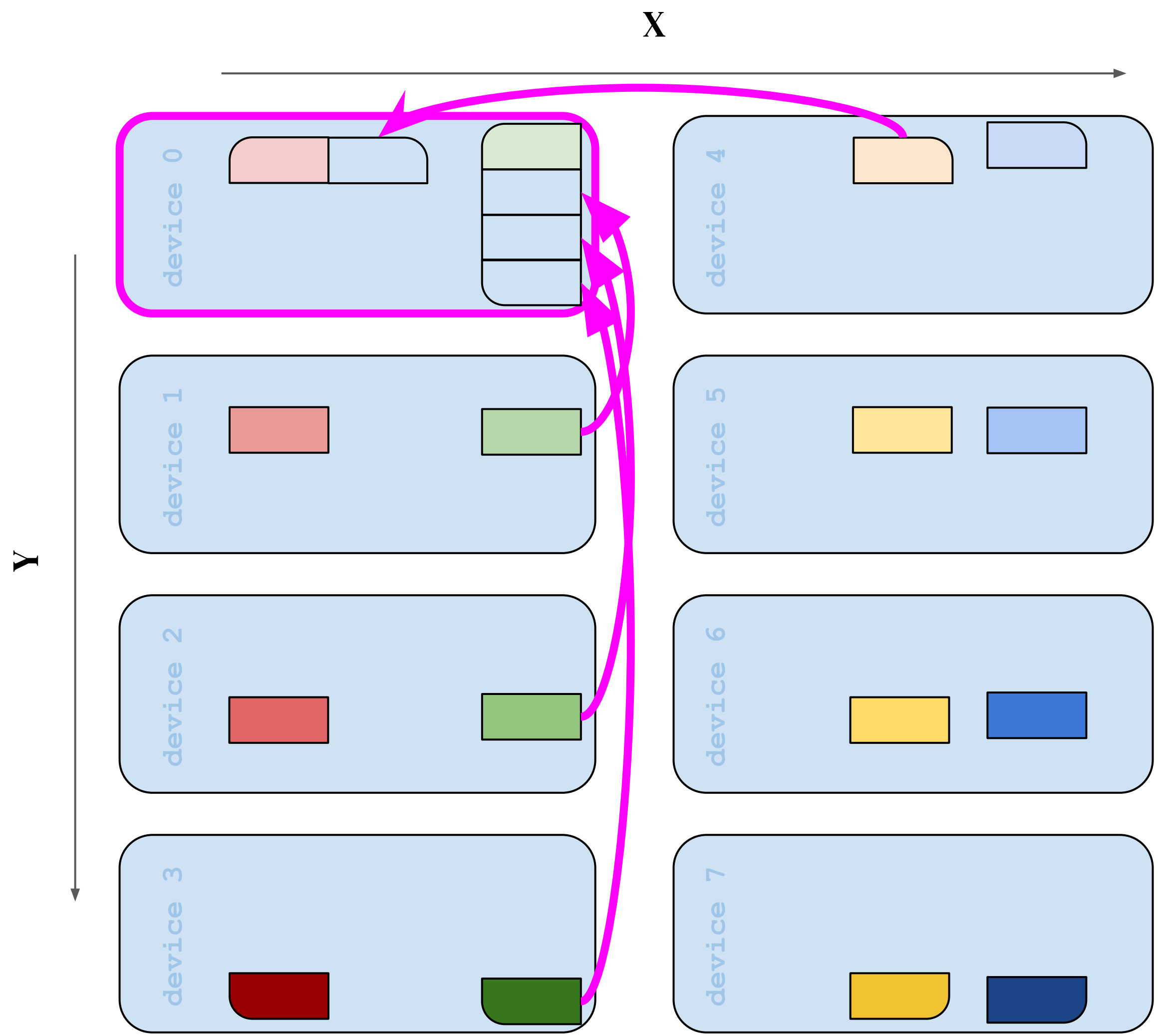




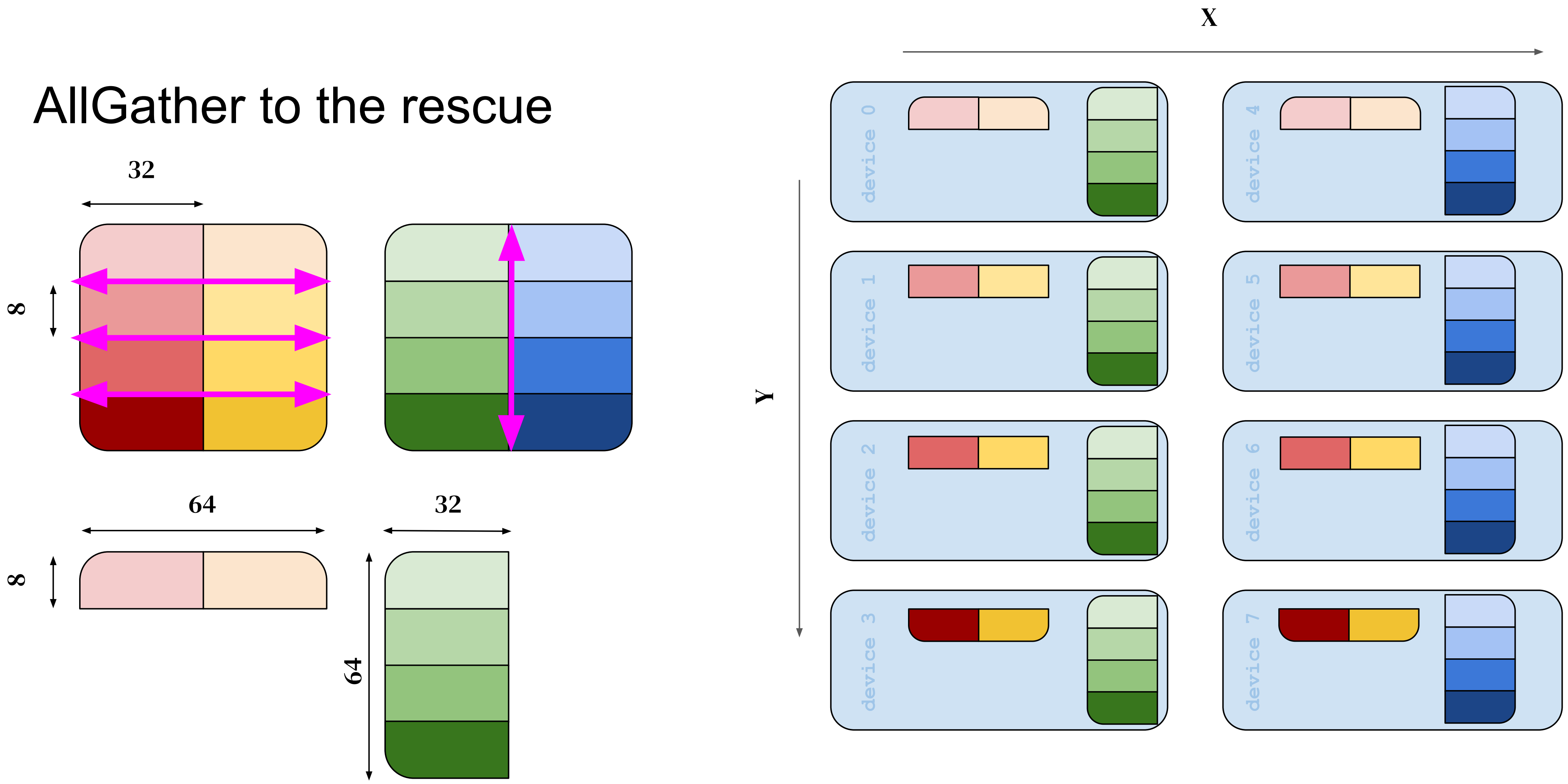
AllGather to the rescue



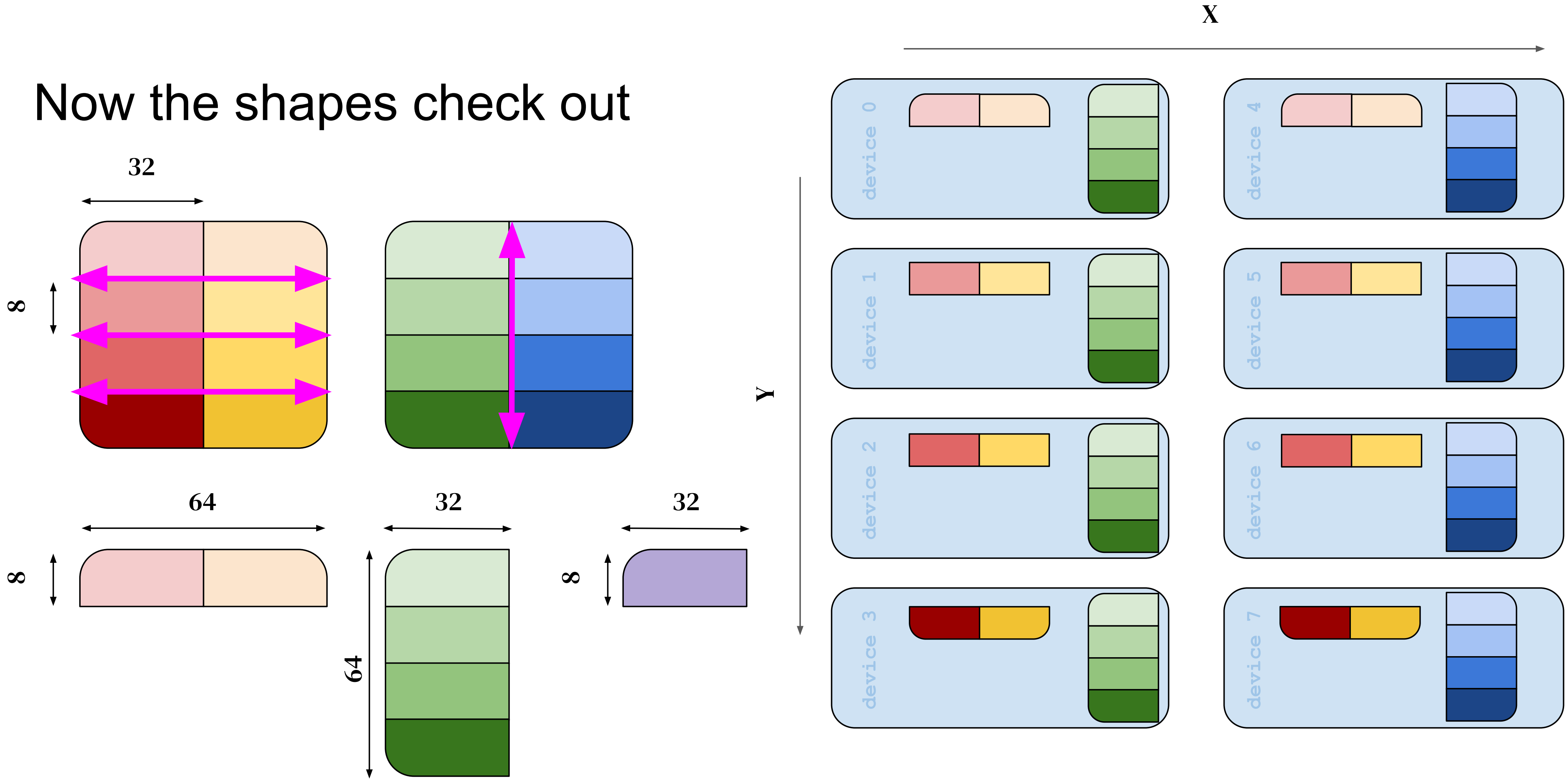
We’re focusing on device 0, but this happens simultaneously on all devices.



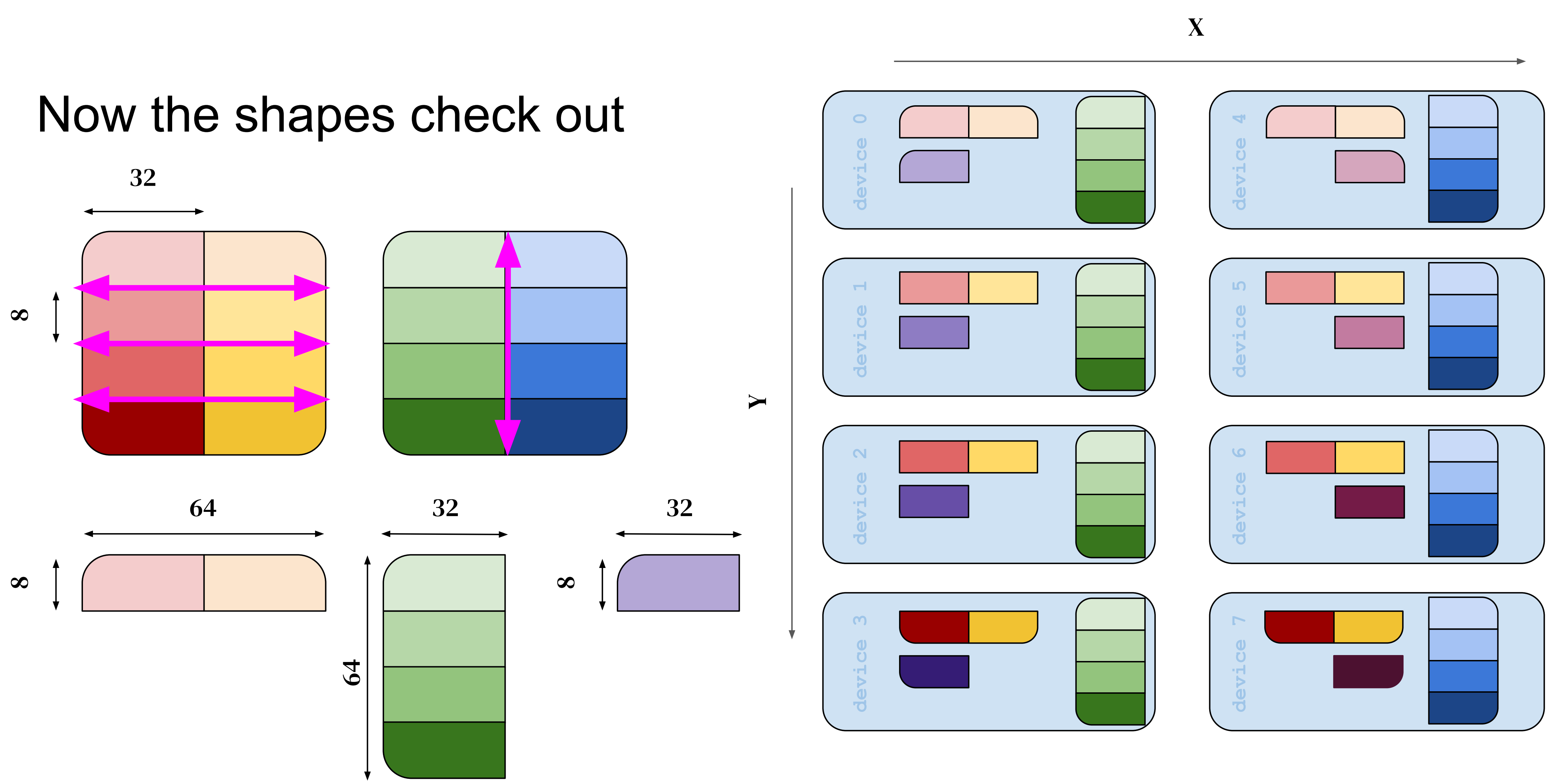
AllGather to the rescue



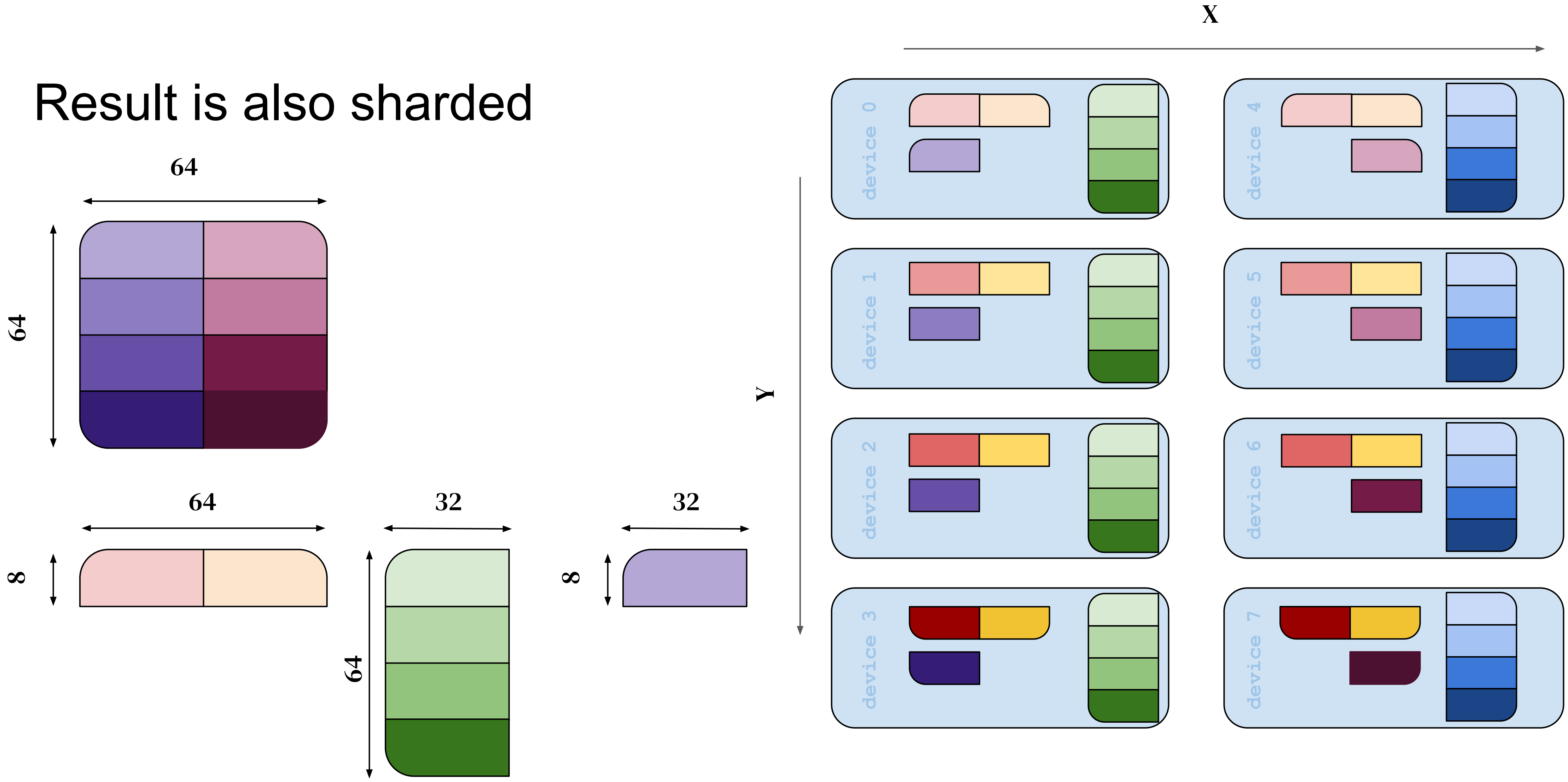
Now the shapes check out



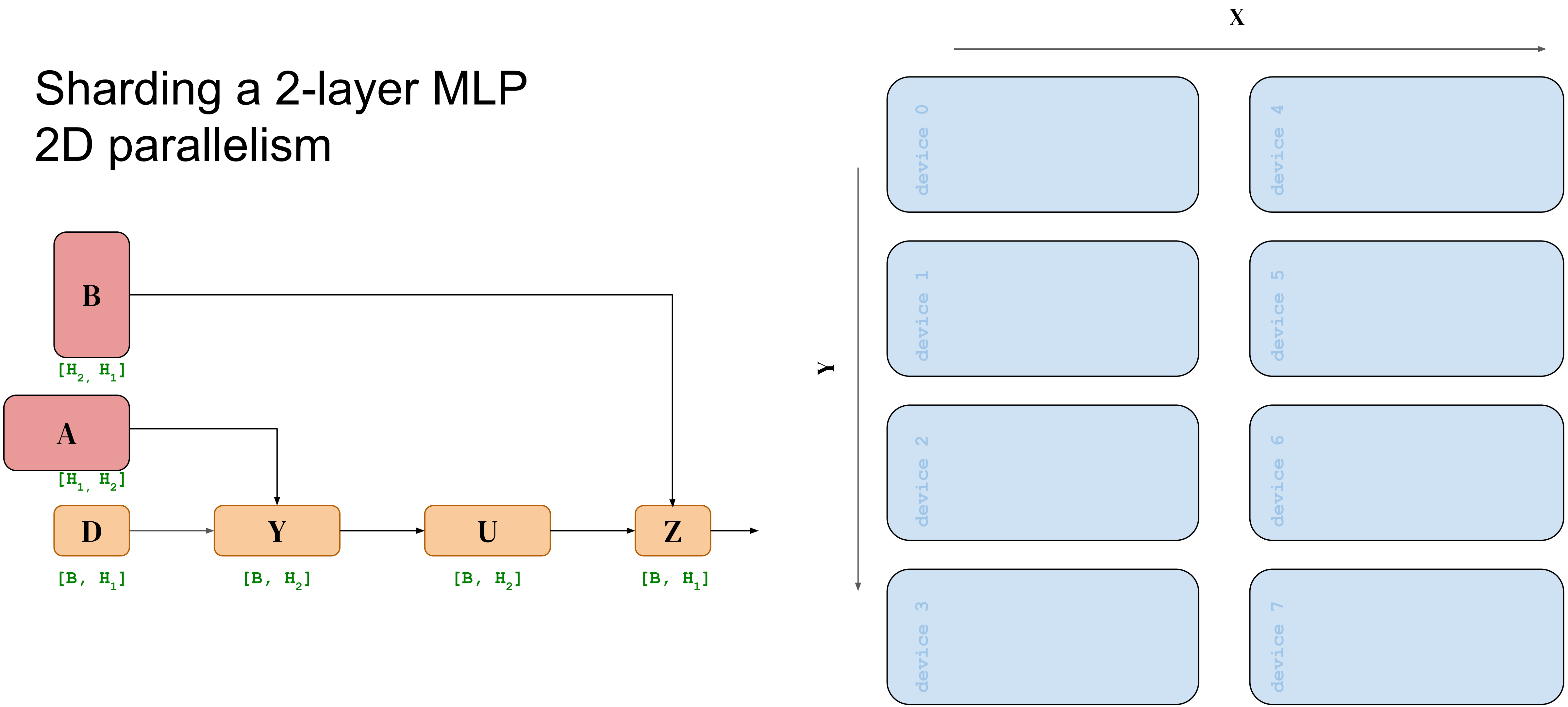
Now the shapes check out

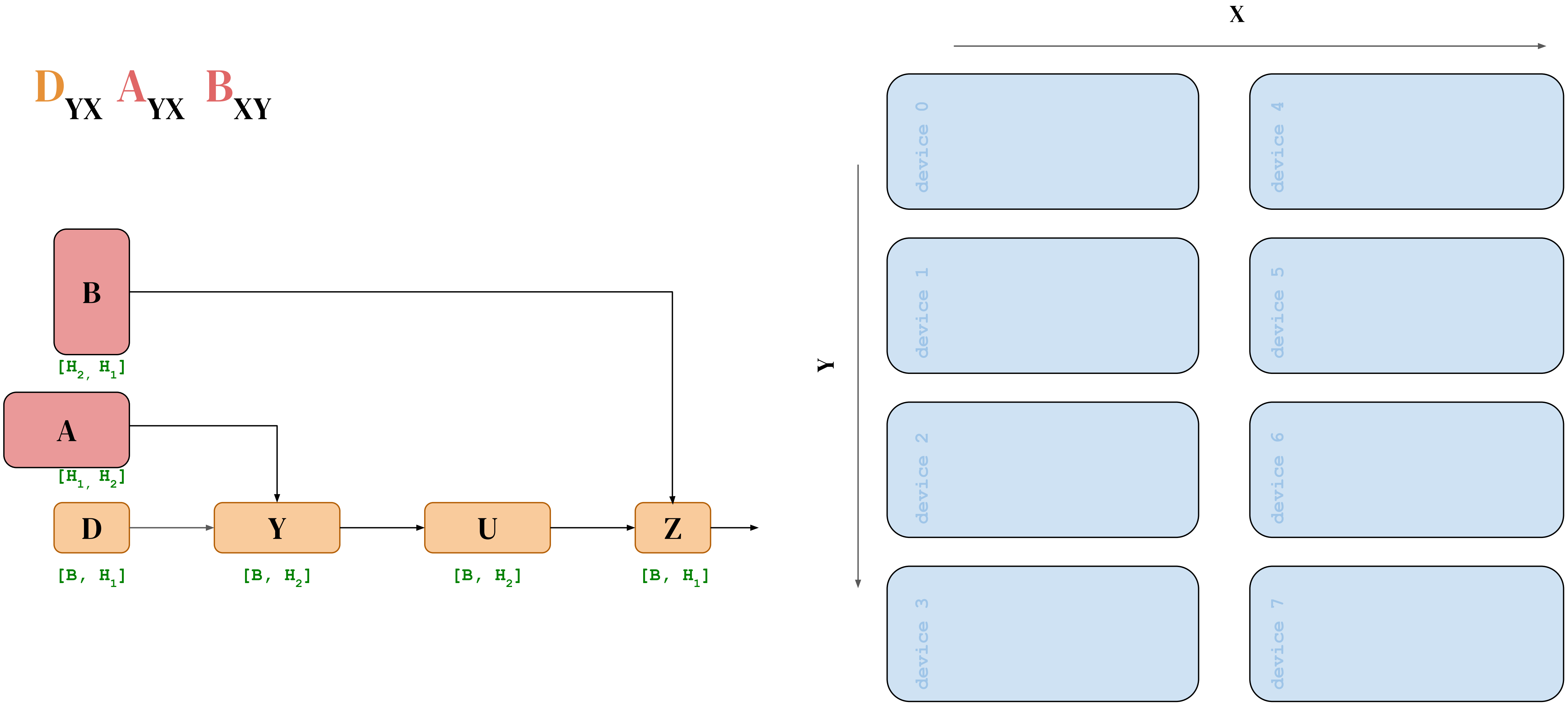


Result is also sharded

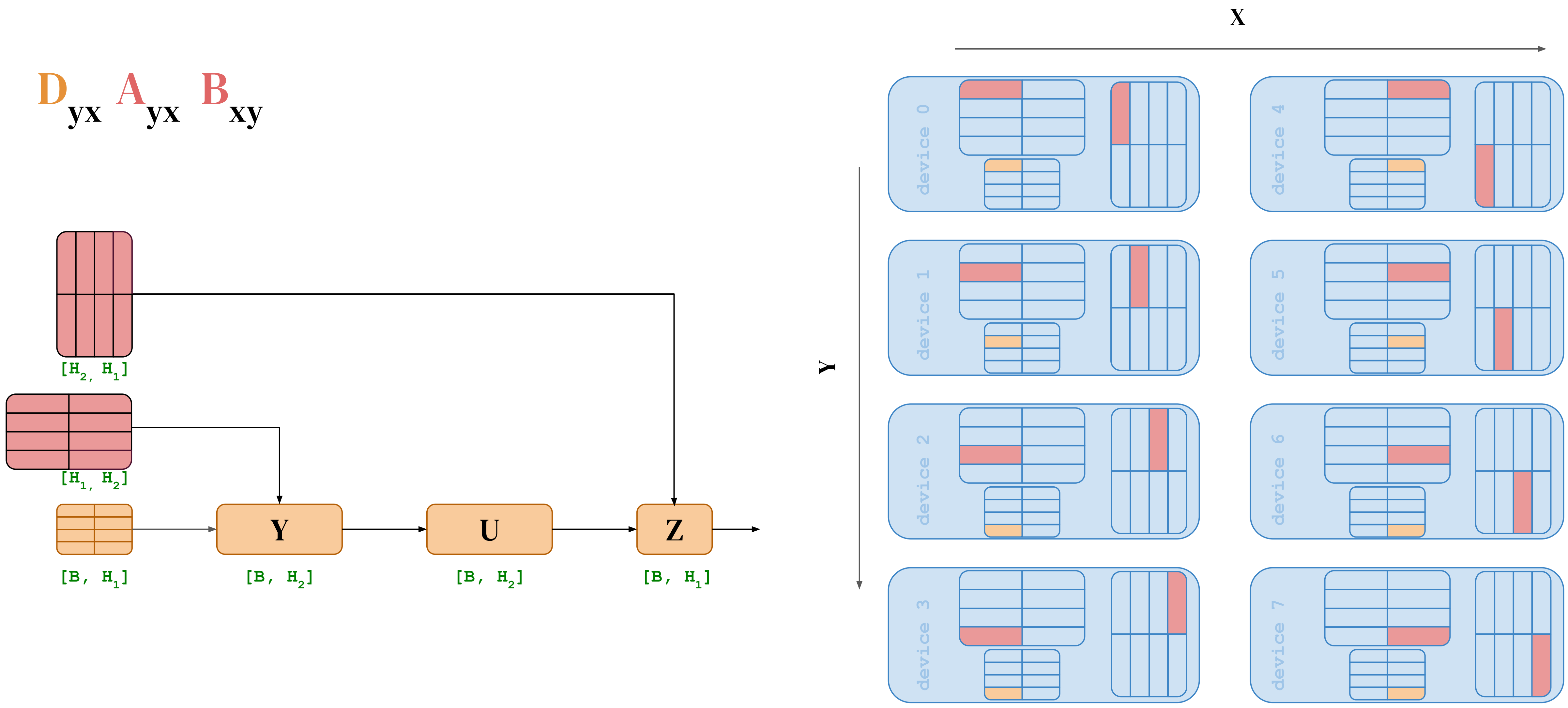


Sharding a 2-layer MLP
2D parallelism

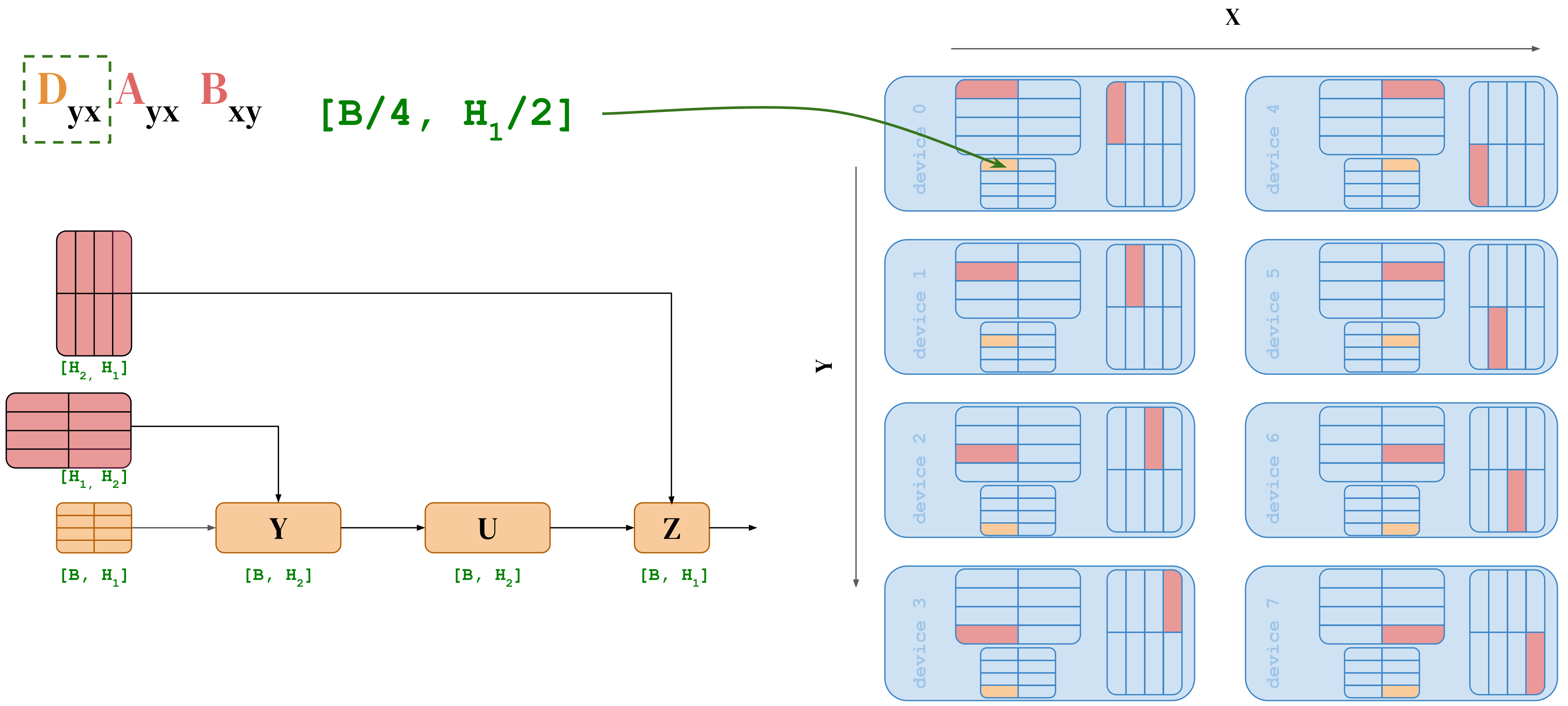




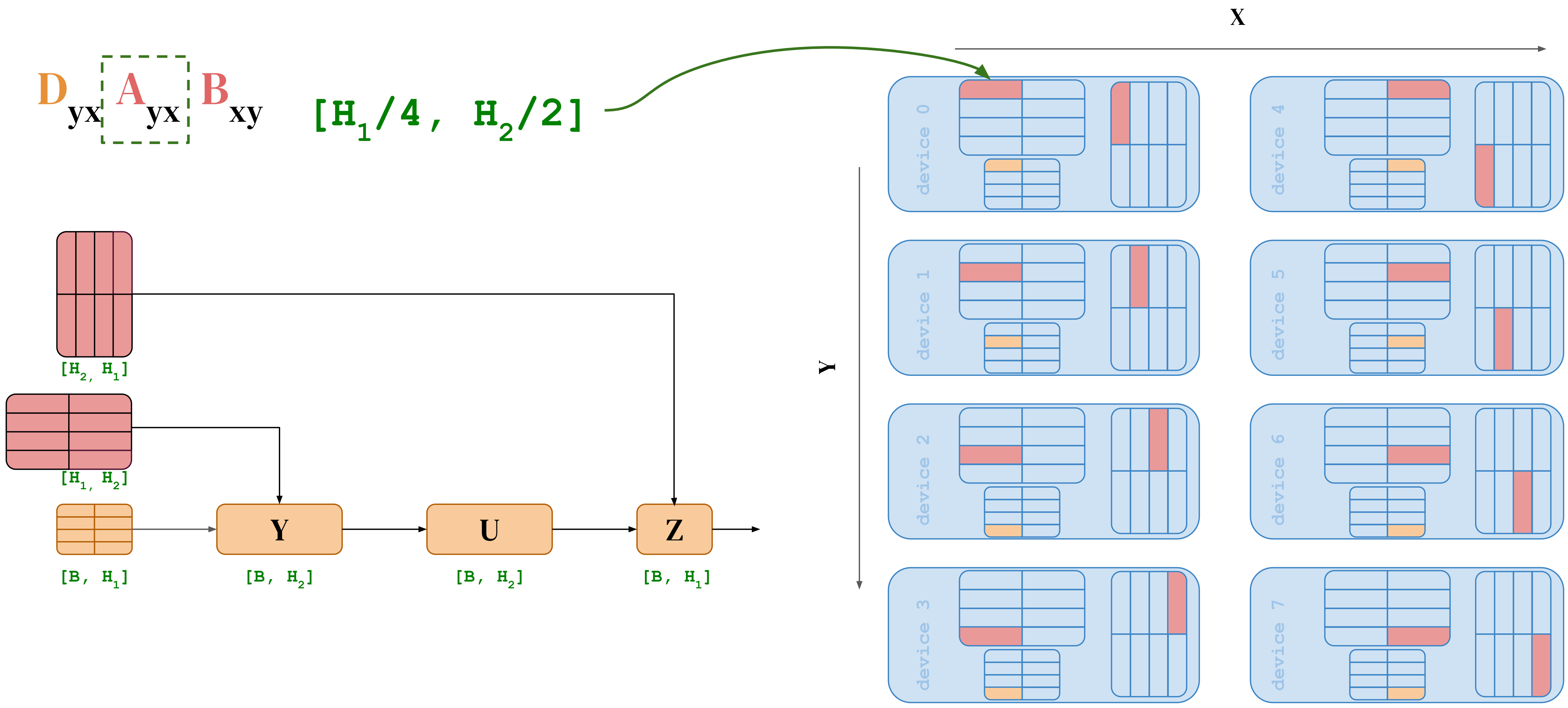
D_{YX} A_{YX} B_{XY}
B
[H_2, H_1]
A
[H_1, H_2]
D
[B, H_1]
Y
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



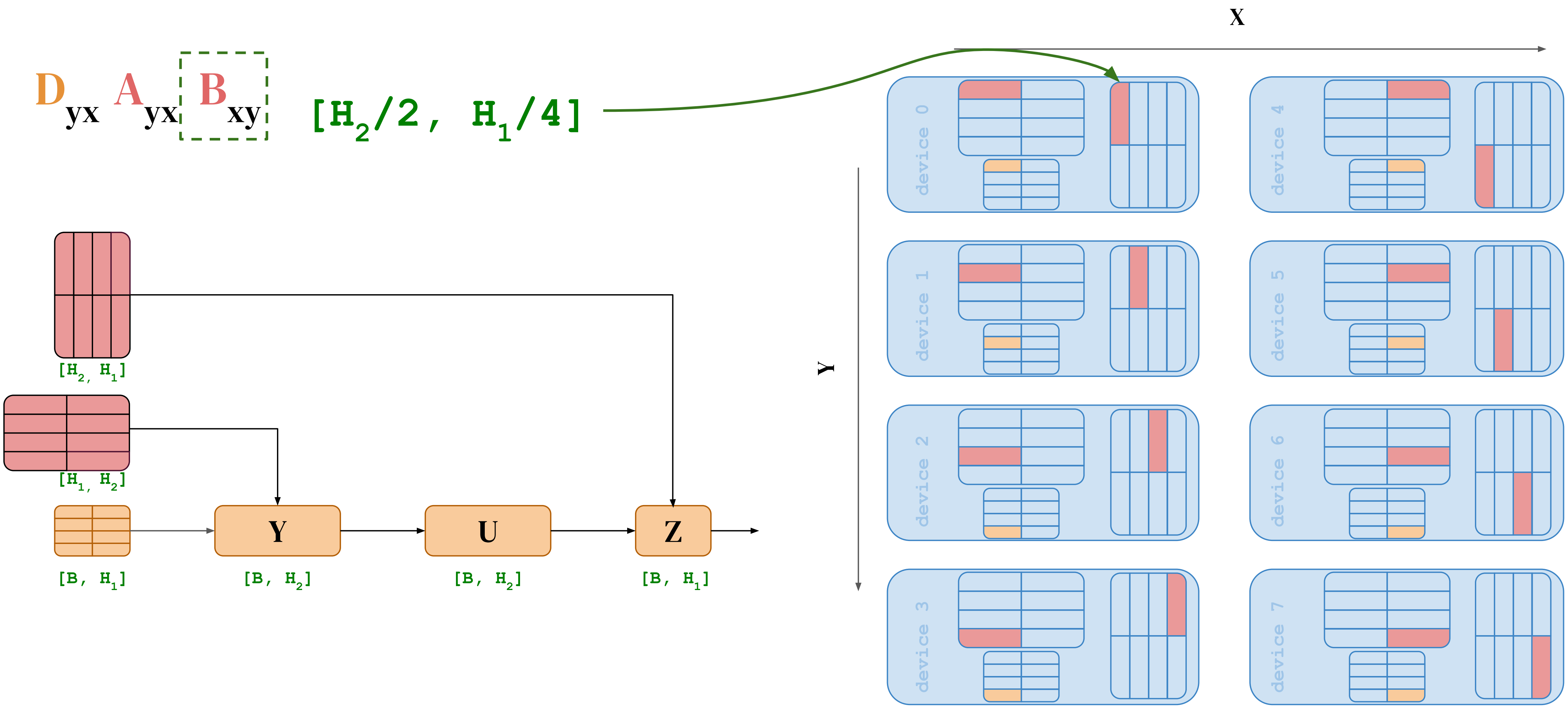
D_{yx} A_{yx} B_{xy}
[H2, H1]
[H1, H2]
[B, H1]
Y
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



D_{yx} A_{yx} B_{xy}
[B/4, H_1/2]
X
Y
[H_2, H_1]
[H_1, H_2]
[B, H_1]
Y
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

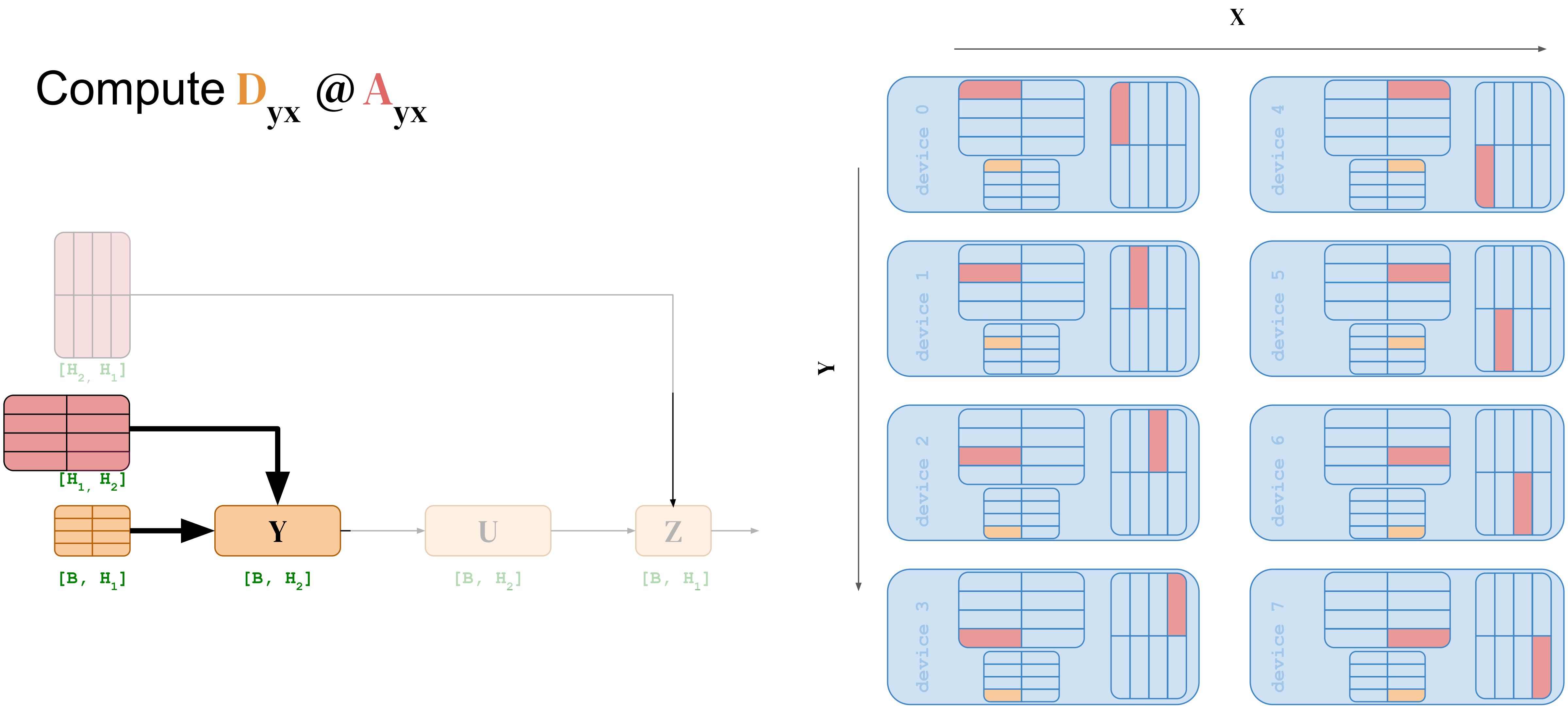


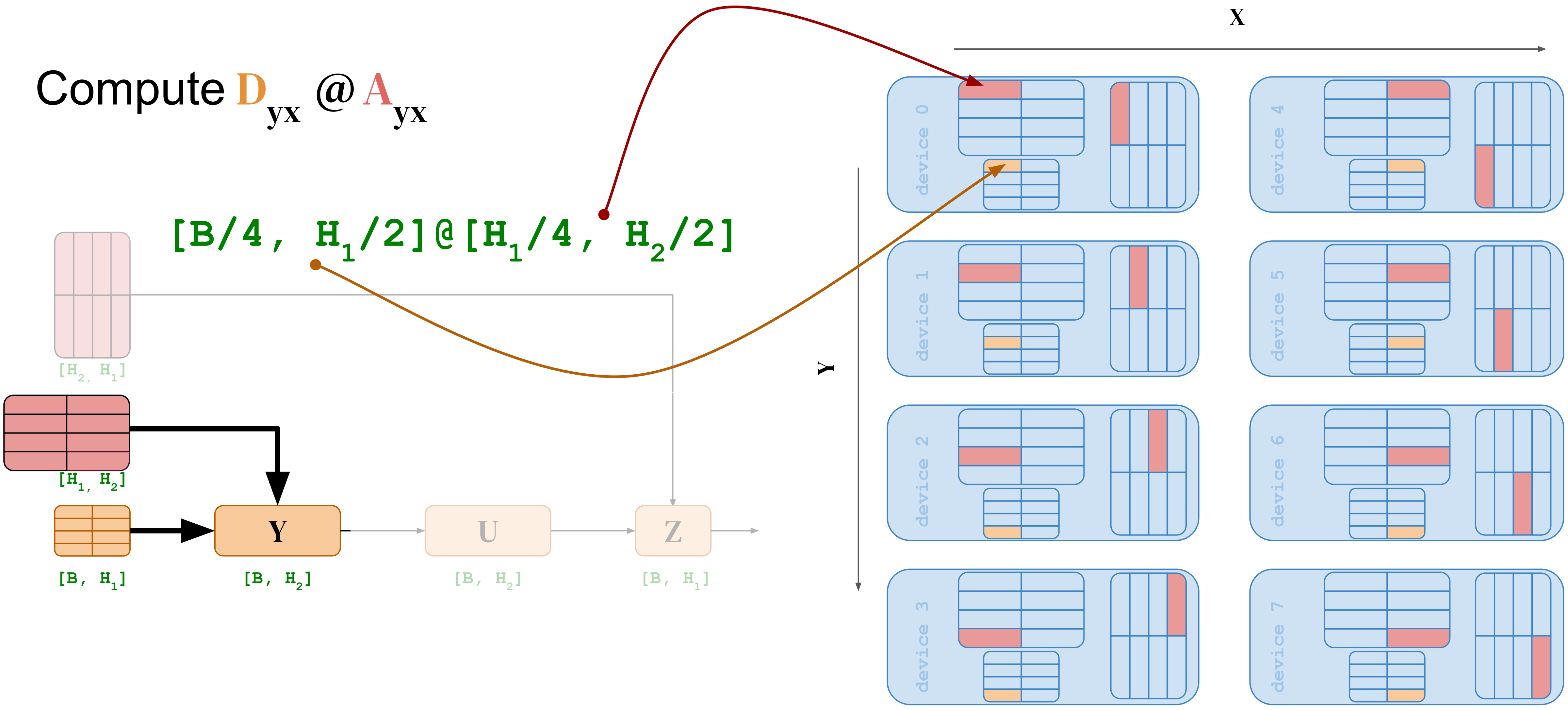
$D_{yx} A_{yx} B_{xy}$
$[H_1/4,\ H_2/2]$
X
Y
$[H_2,\ H_1]$
$[H_1,\ H_2]$
$[B,\ H_1]$
Y
$[B,\ H_2]$
U
$[B,\ H_2]$
Z
$[B,\ H_1]$
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



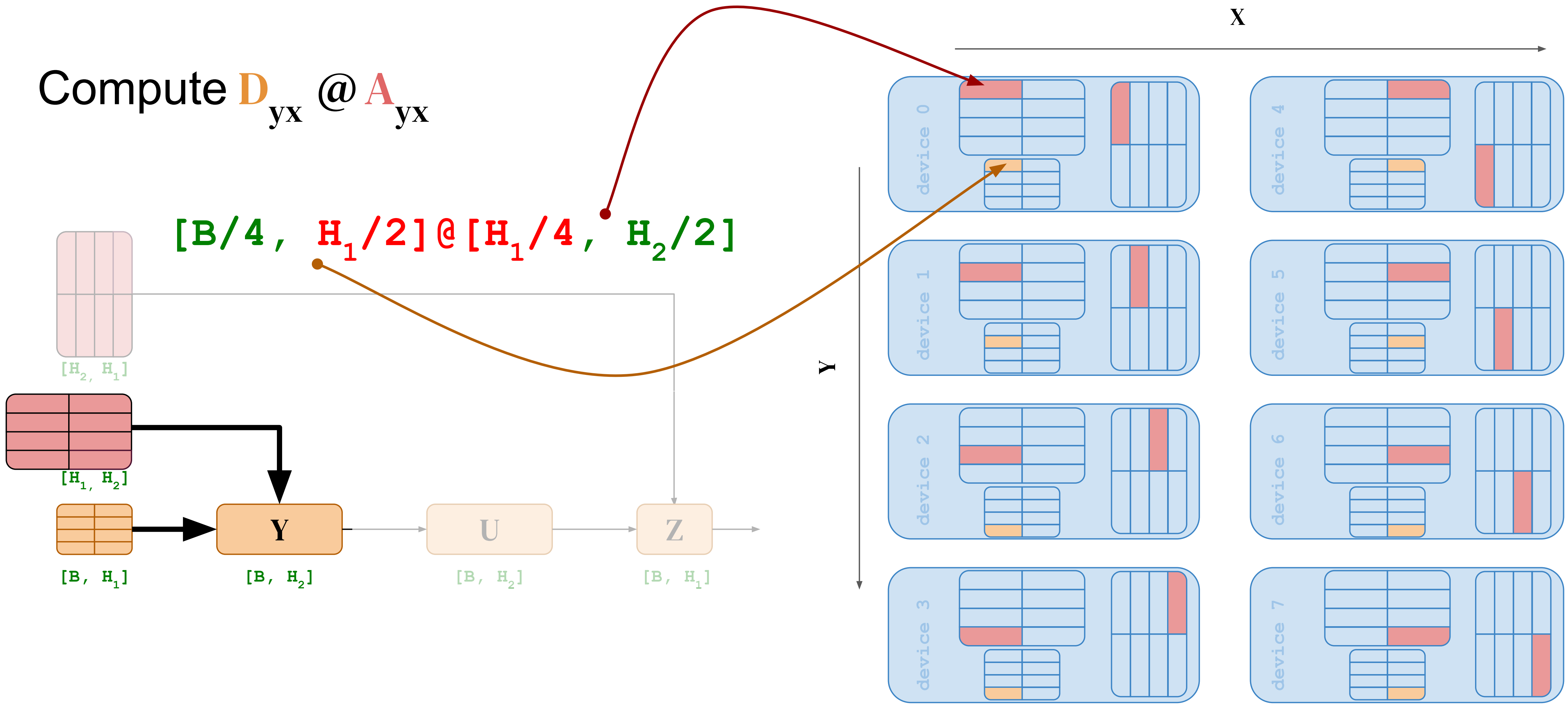
$D_{yx}\ A_{yx}\ B_{xy}$
$[H_2/2,\ H_1/4]$
X
Y
$[H_2,\ H_1]$
$[H_1,\ H_2]$
$[B,\ H_1]$
Y
$[B,\ H_2]$
U
$[B,\ H_2]$
Z
$[B,\ H_1]$
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

Compute D_{yx} @ A_{yx}

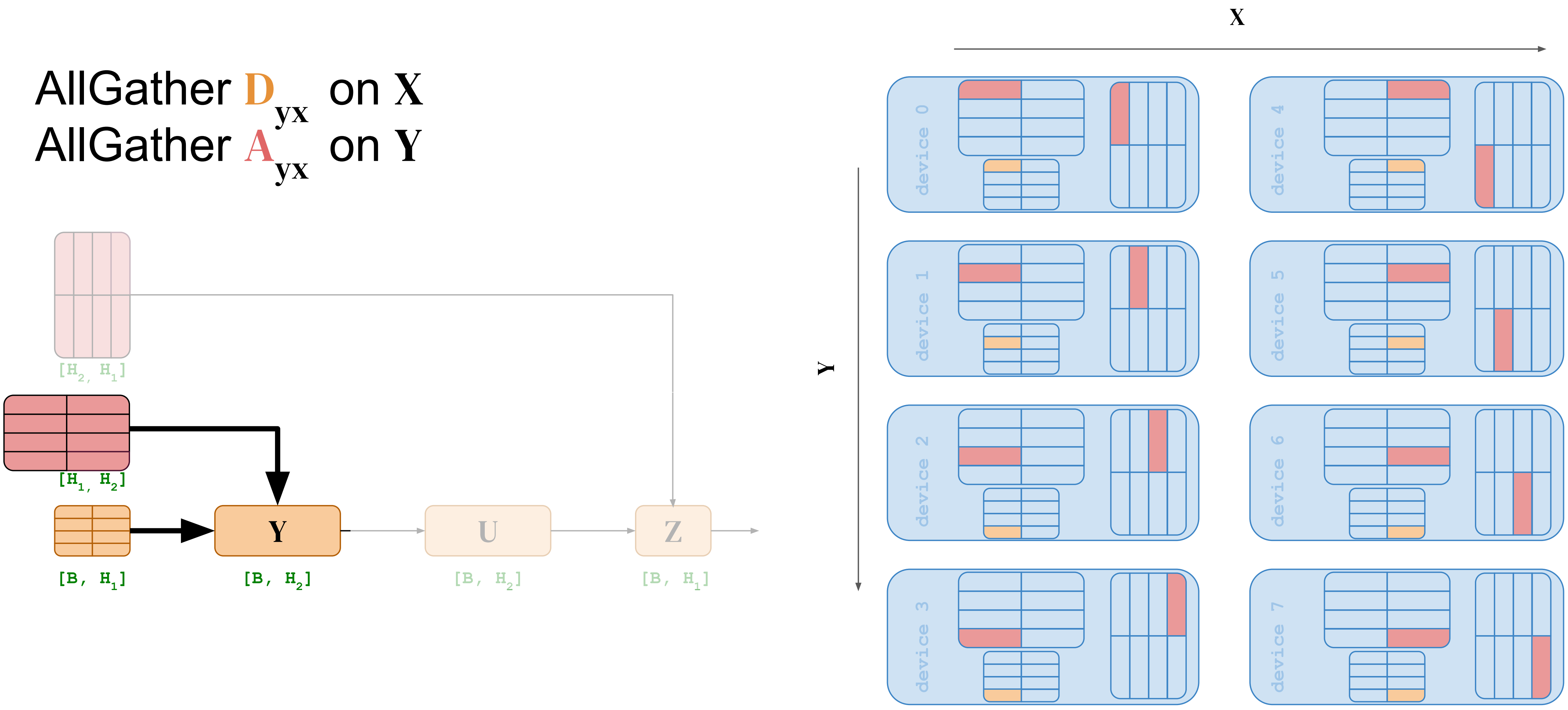


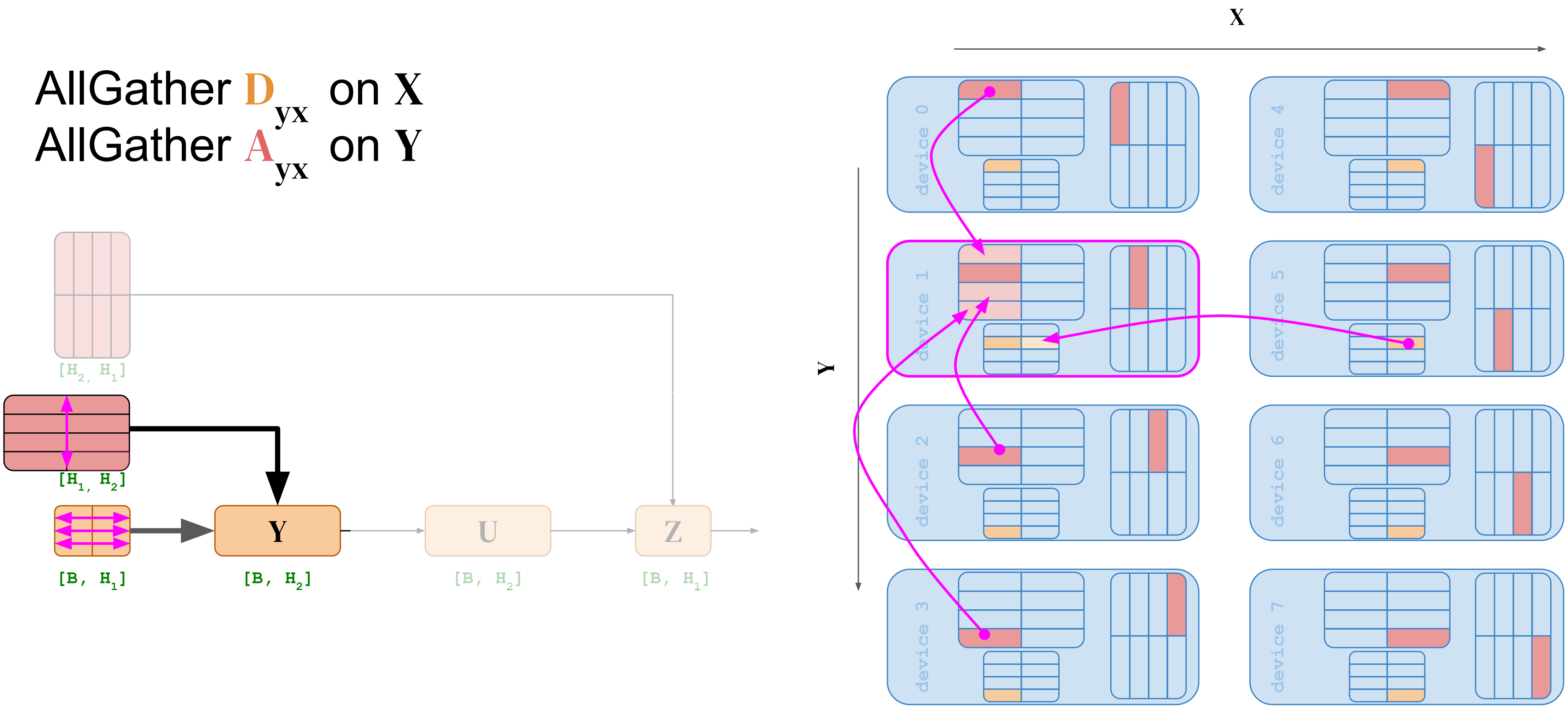


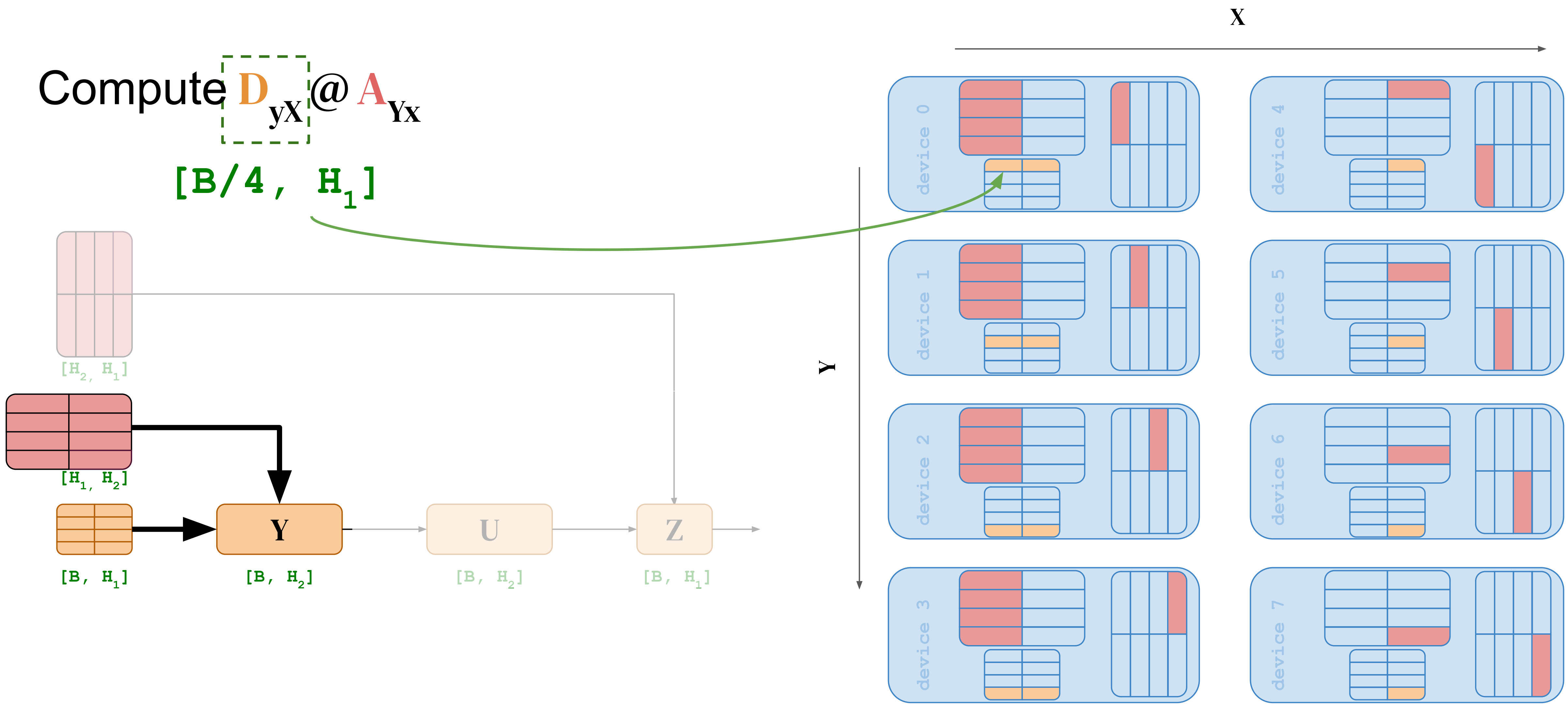
Compute D yx @ A yx
[B/4, H1/2]@[H1/4, H2/2]
[H2, H1]
[H1, H2]
[B, H1]
Y
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



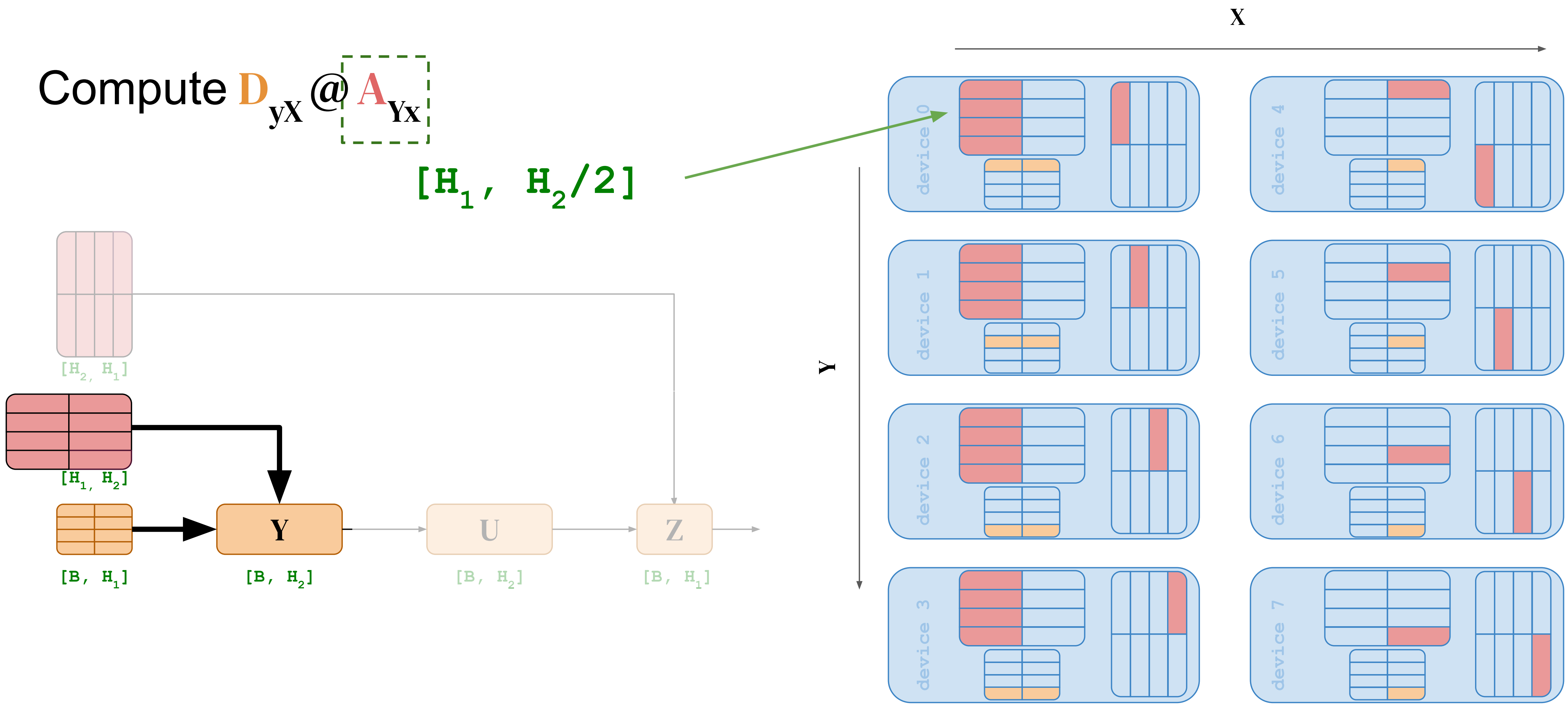
Compute D yx @ A yx
[B/4, H1/2]@[H1/4, H2/2]
[H2, H1]
[H1, H2]
[B, H1]
Y
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7


AllGather D yx on X
AllGather A yx on Y
[H2, H1]
[H1, H2]
[B, H1]
Y
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

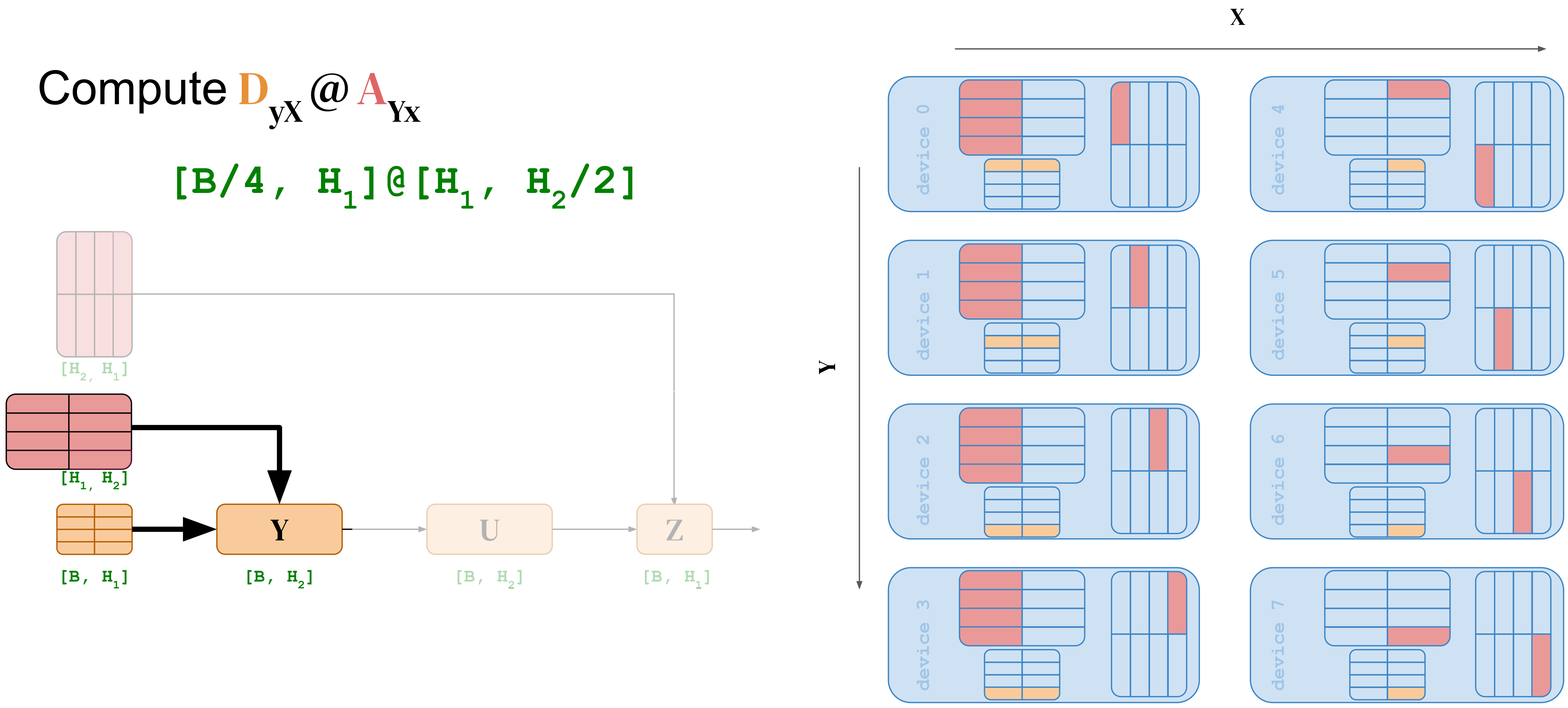

AllGather D_{yx} on X
AllGather A_{yx} on Y
[H_2, H_1]
[H_1, H_2]
[B, H_1]
Y
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



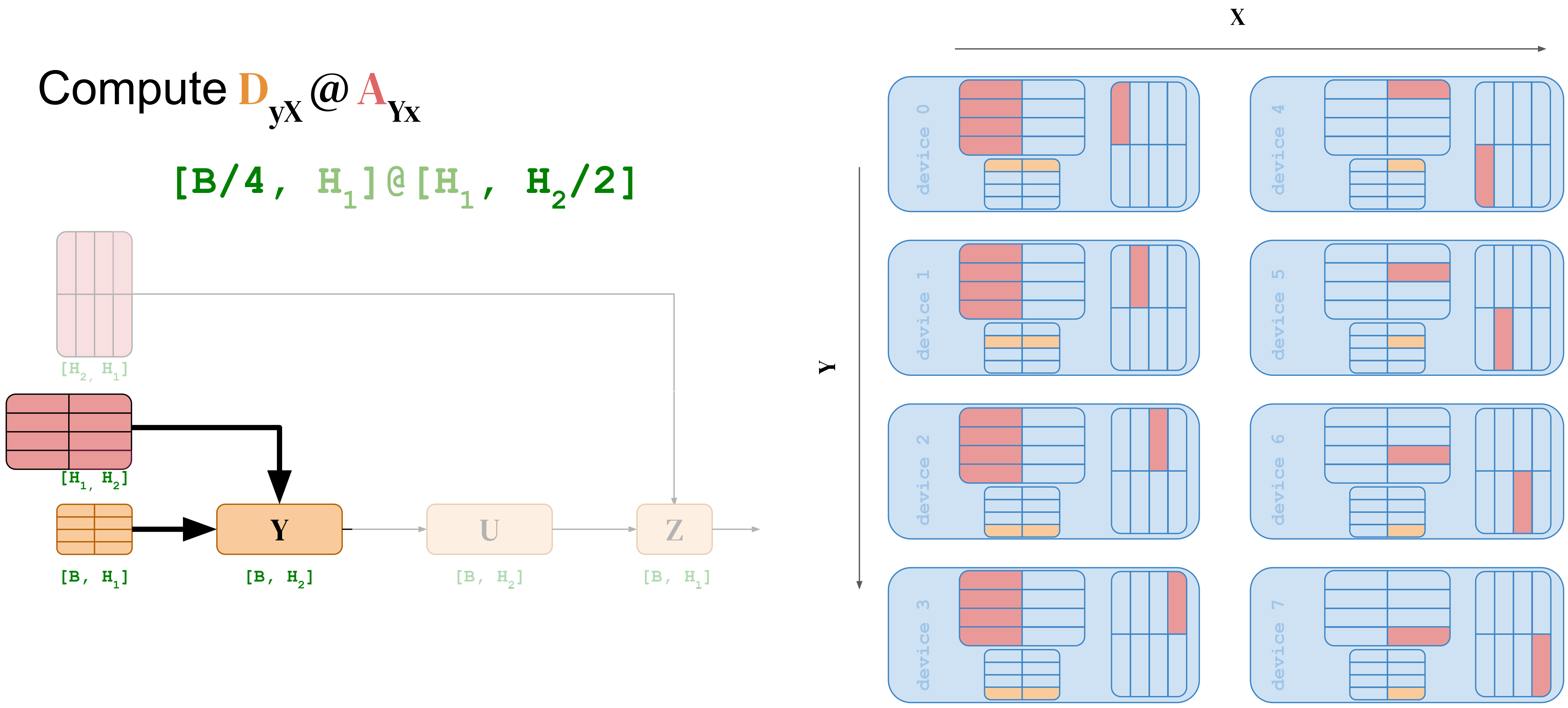
Compute D_{yX} @ A_{Yx}
[B/4, H_1]
[H_2, H_1]
[H_1, H_2]
[B, H_1]
Y
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



Compute D_{yX} @ A_{Yx}
$[H_1, H_2/2]$
X
Y
$[H_2, H_1]$
$[H_1, H_2]$
Y
U
Z
$[B, H_1]$
$[B, H_2]$
$[B, H_2]$
$[B, H_1]$
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

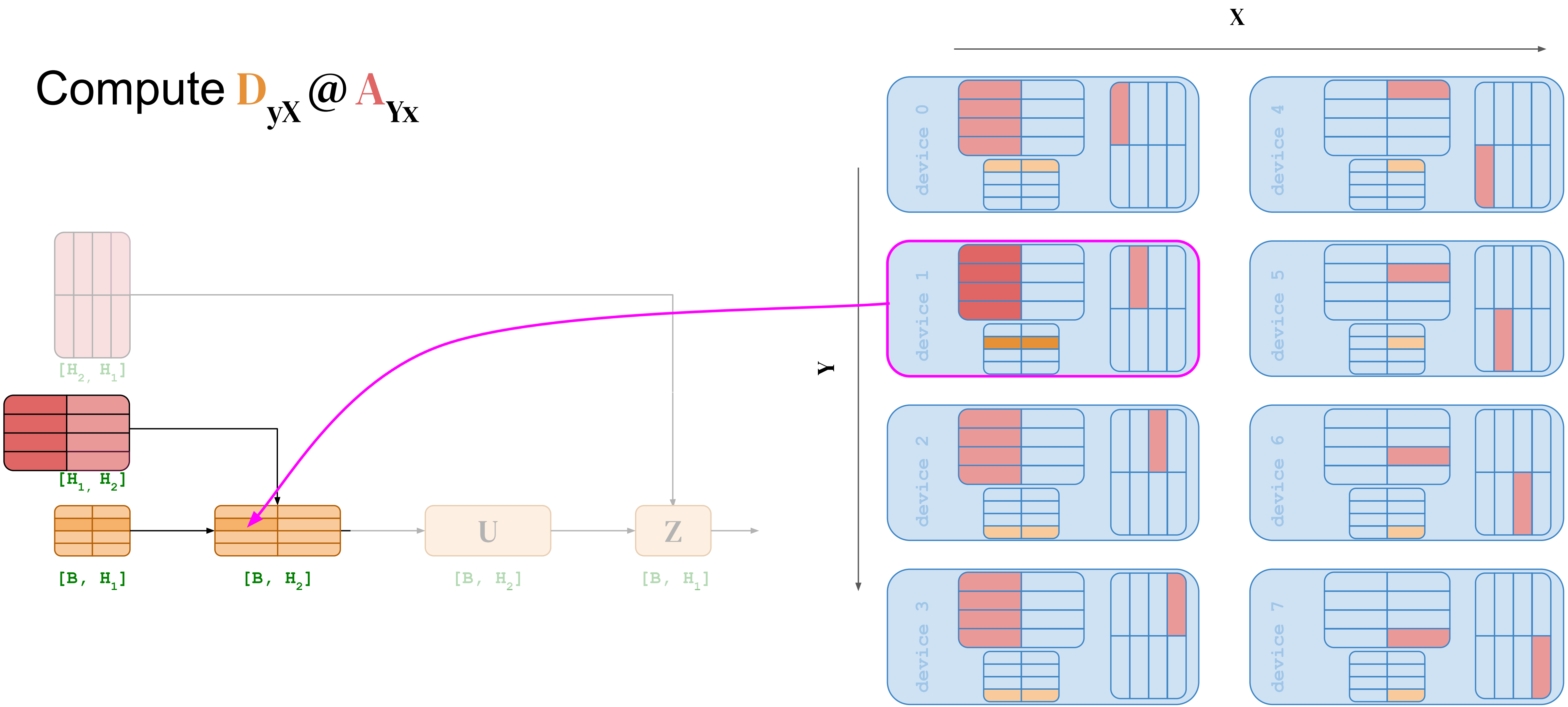


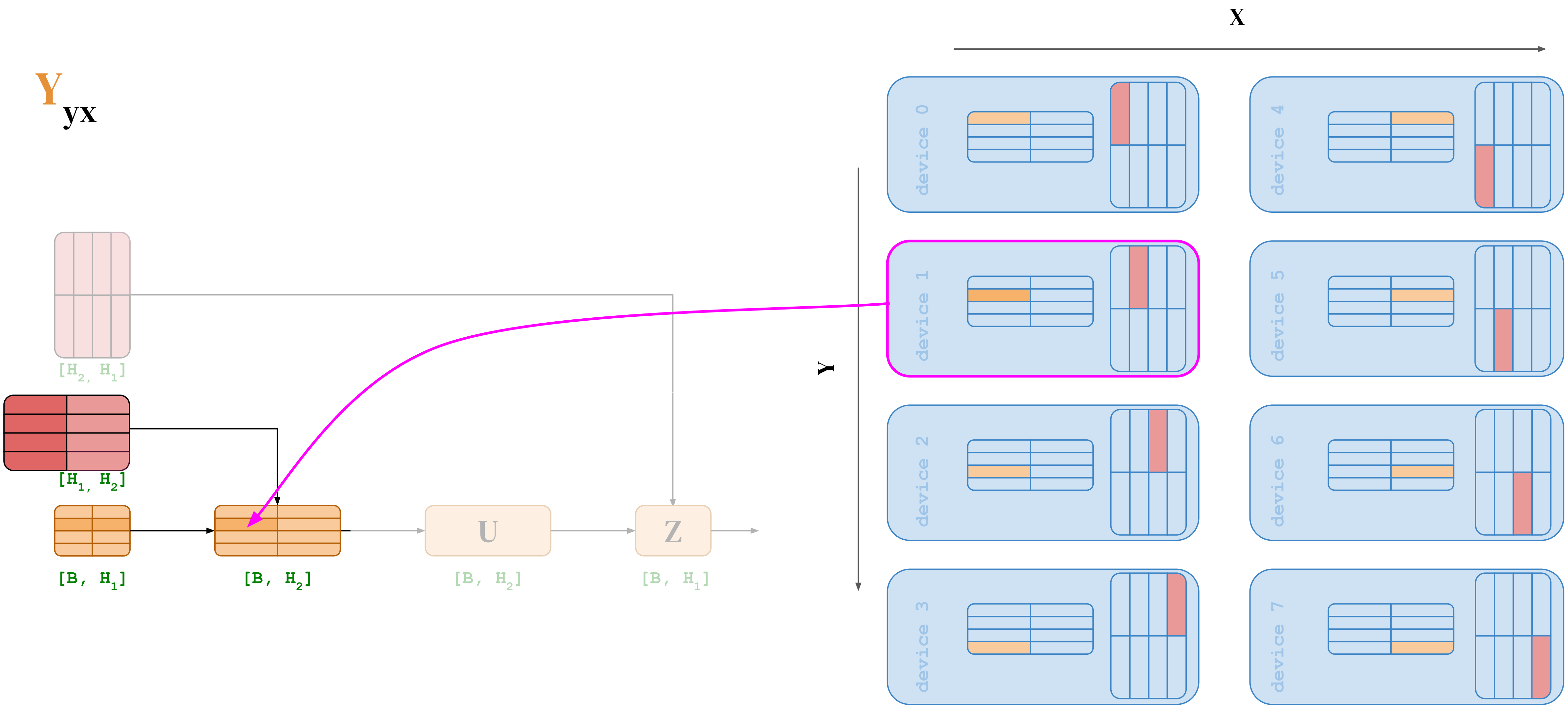
Compute D_{yX} @ A_{Yx}
[B/4, H_1]@[H_1, H_2/2]
[H_2, H_1]
[H_1, H_2]
[B, H_1]
Y
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



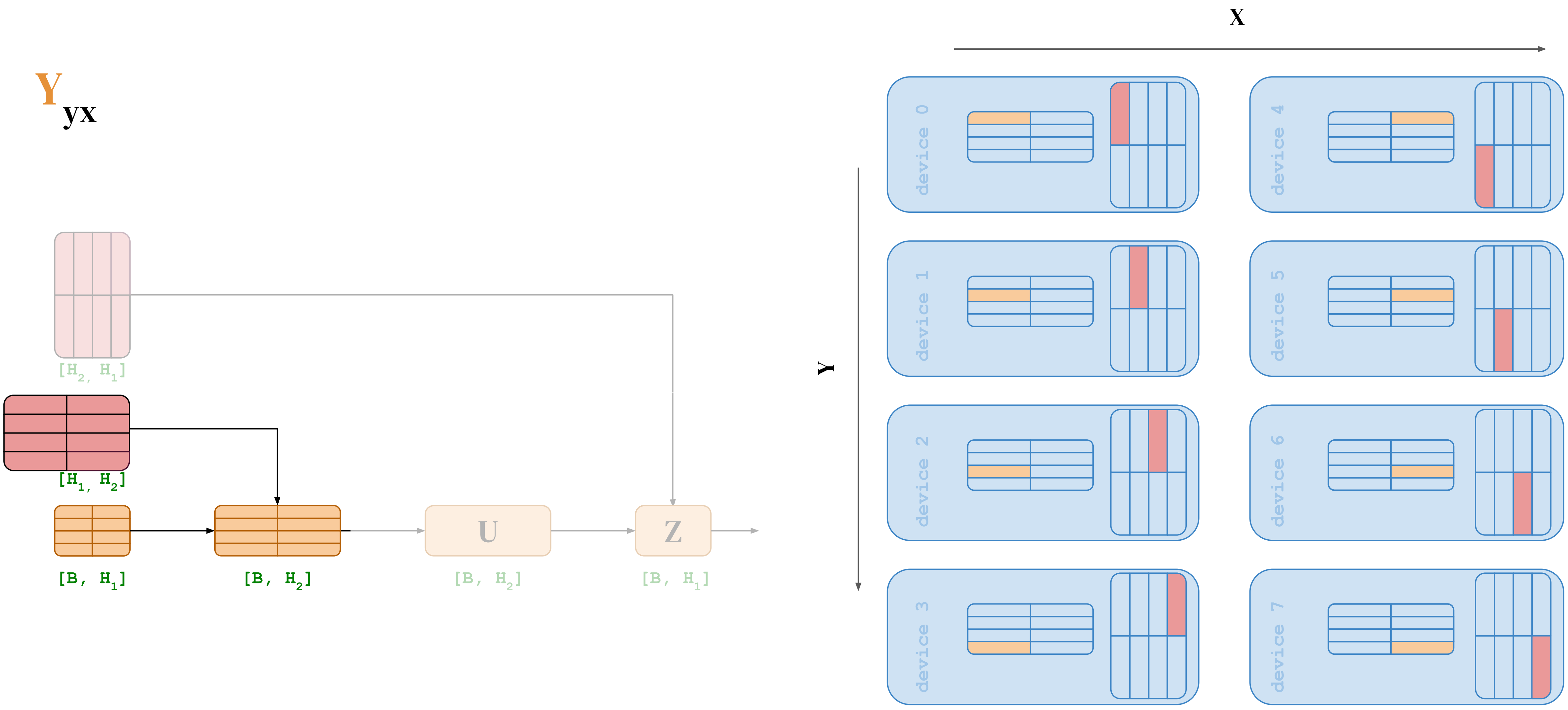
Compute D yX @ A Yx
[B/4, H1]@[H1, H2/2]
[H2, H1]
[H1, H2]
[B, H1]
Y
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

Compute D_{yX} @ A_{Yx}

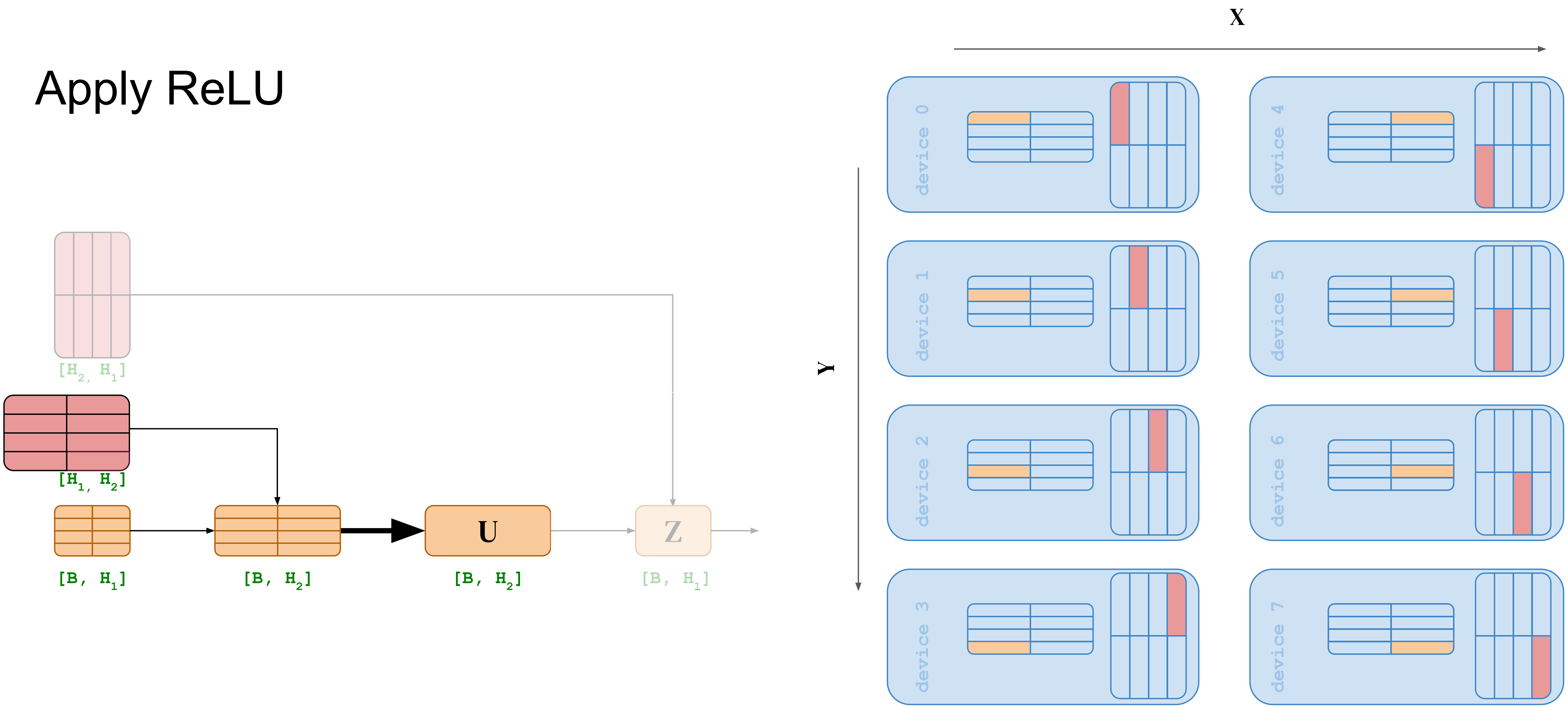




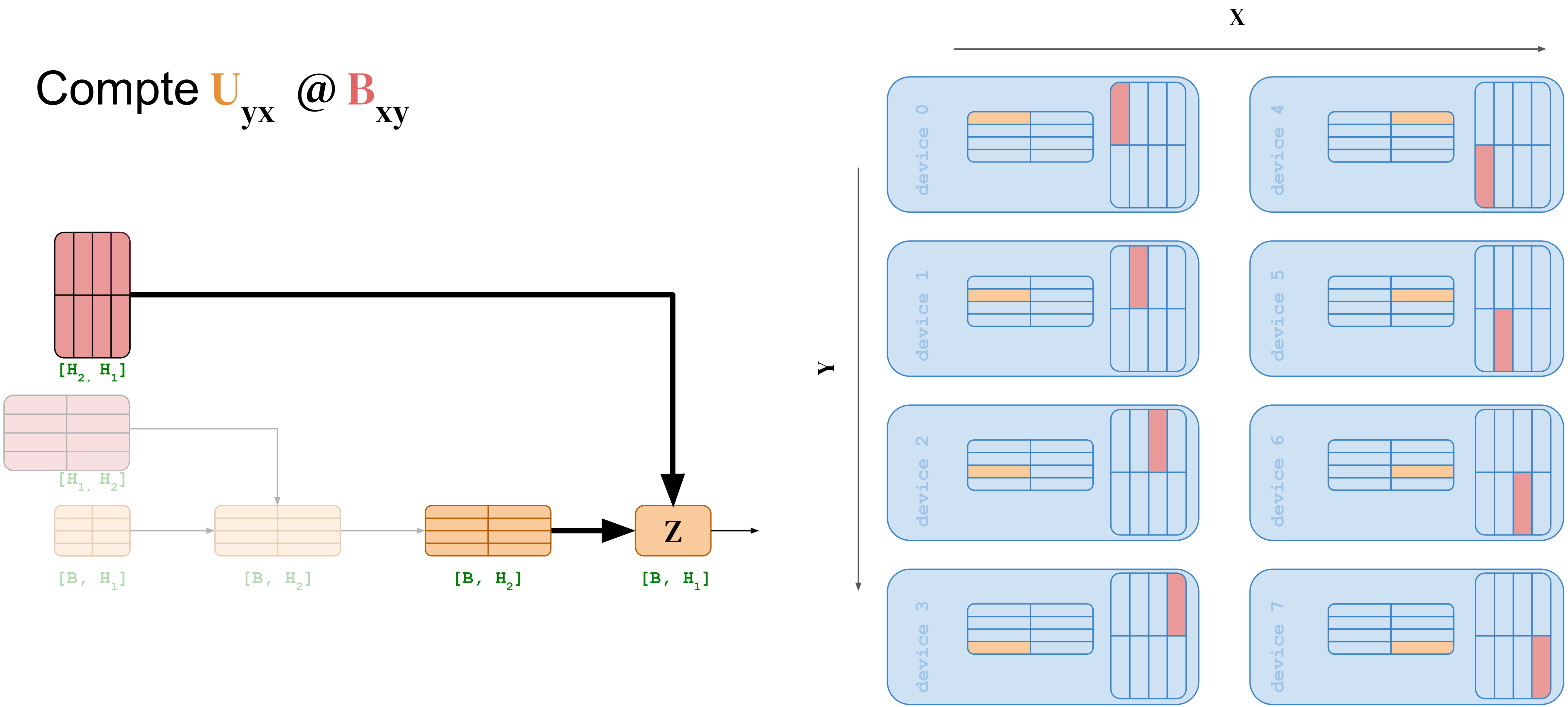
Y_{yx}
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
U
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

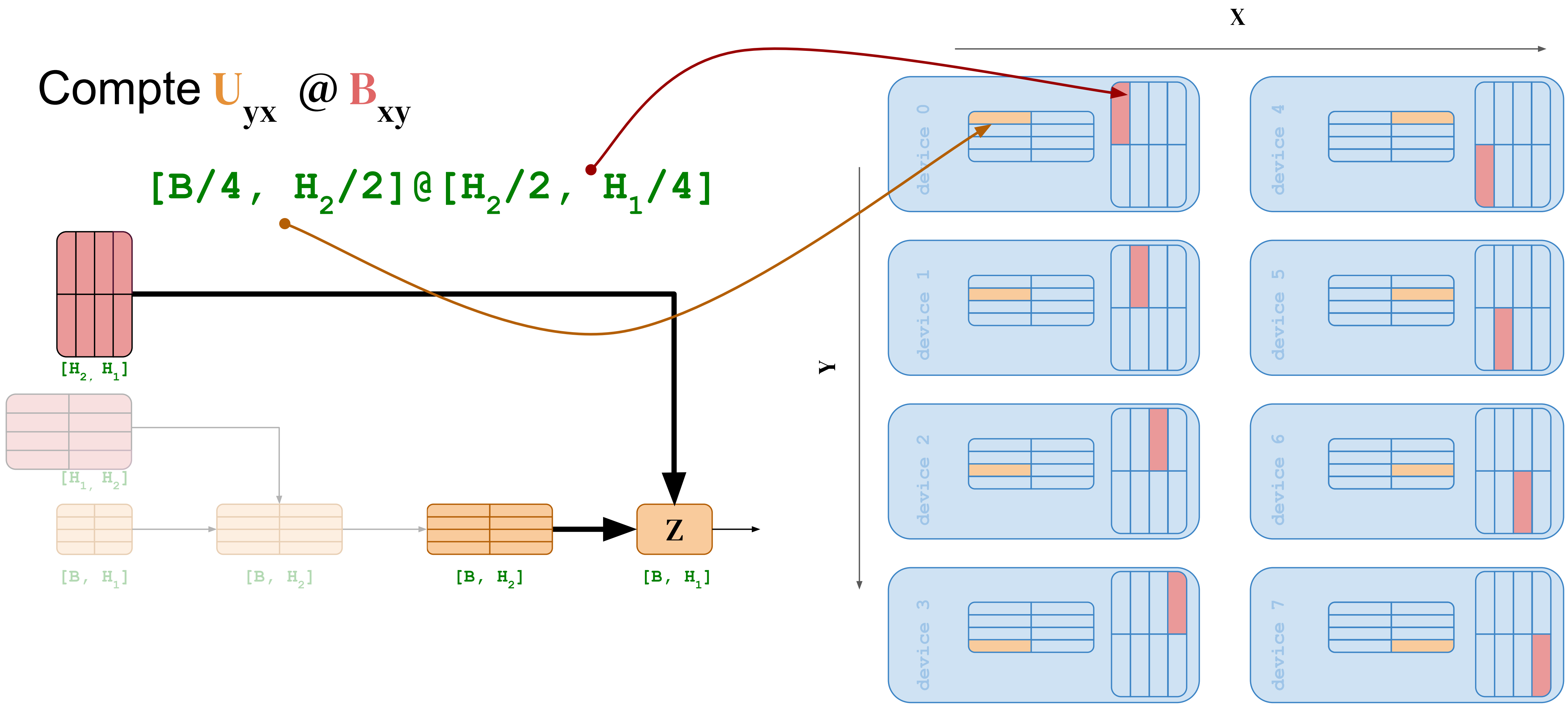

Y_{yx}
$[H_2, H_1]$
$[H_1, H_2]$
$[B, H_1]$
$[B, H_2]$
U
$[B, H_2]$
Z
$[B, H_1]$
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

Apply ReLU

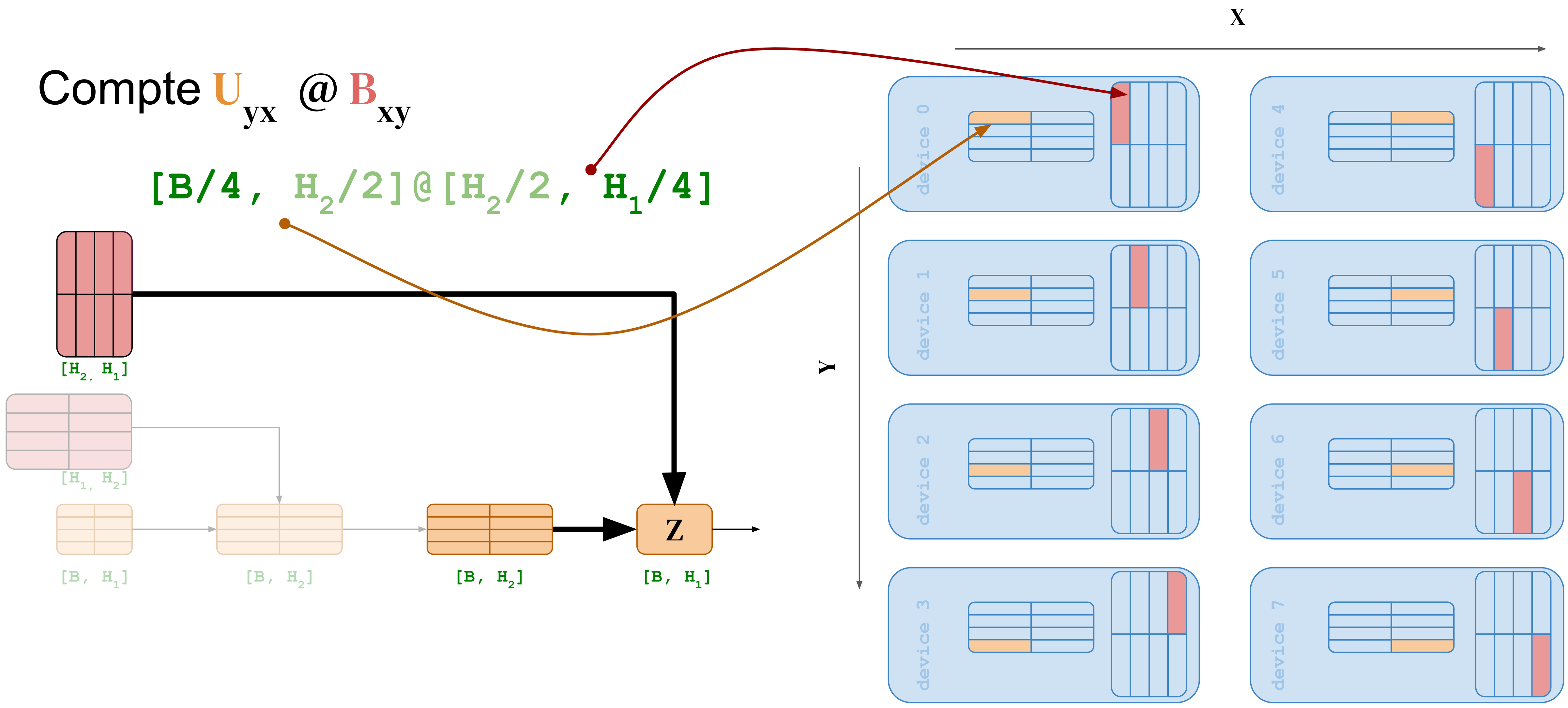


[H_2, H_1]
[H_1, H_2]
[B, H_1]
[B, H_2]
U
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

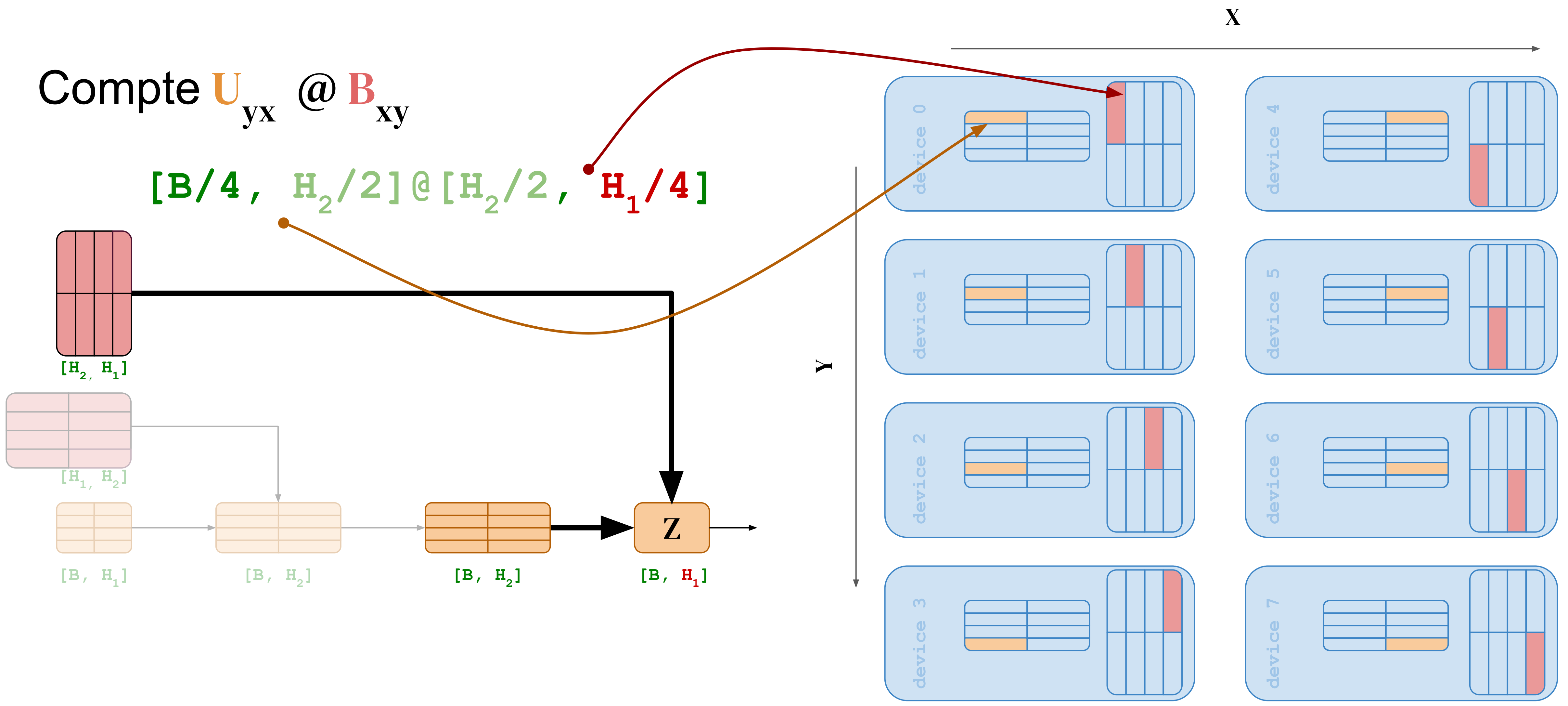

Compte U yx @ B xy
X
Y
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
[B, H2]
Z
[B, H1]
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



Compte U_{yx} @ B_{xy}
[B/4, H_2/2]@[H_2/2, H_1/4]
[H_2, H_1]
[H_1, H_2]
[B, H_1]
[B, H_2]
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

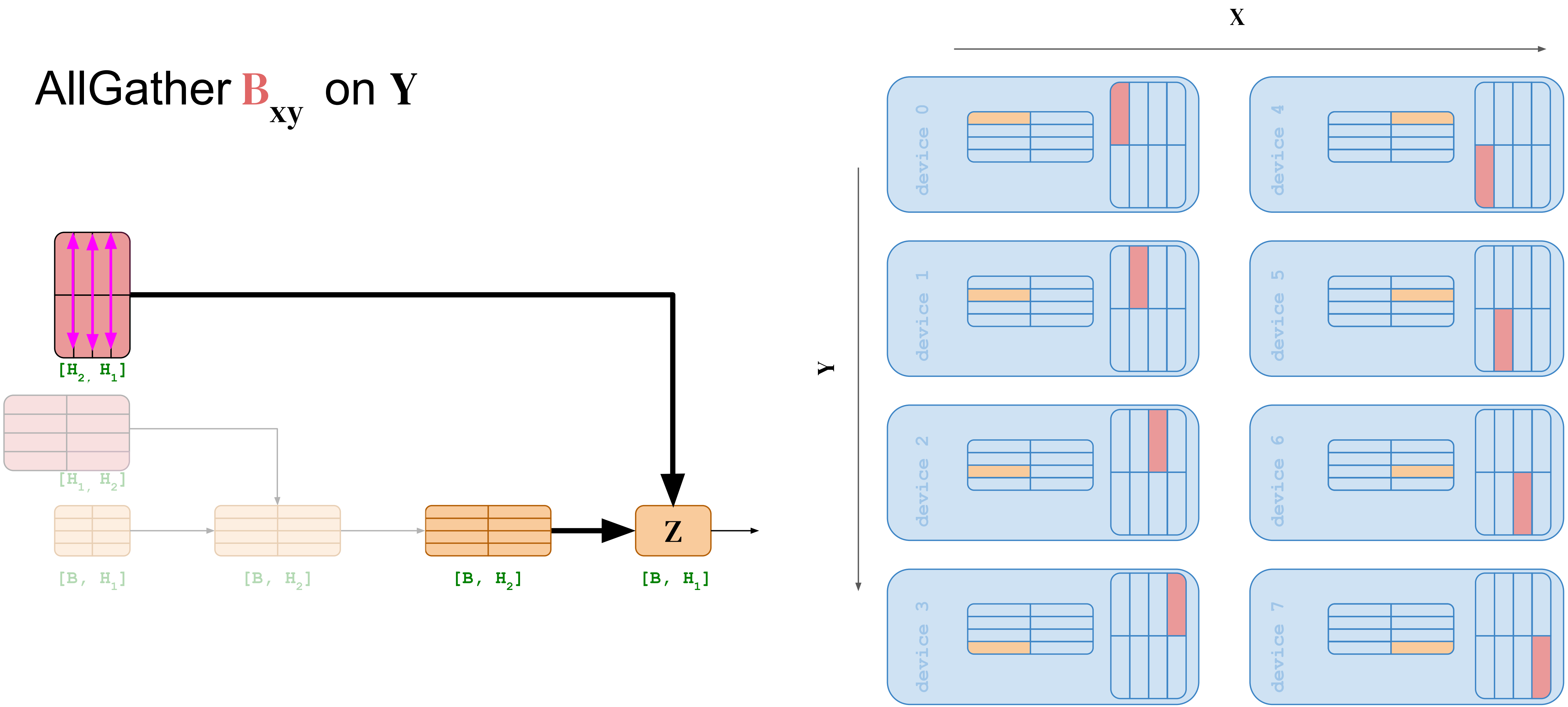


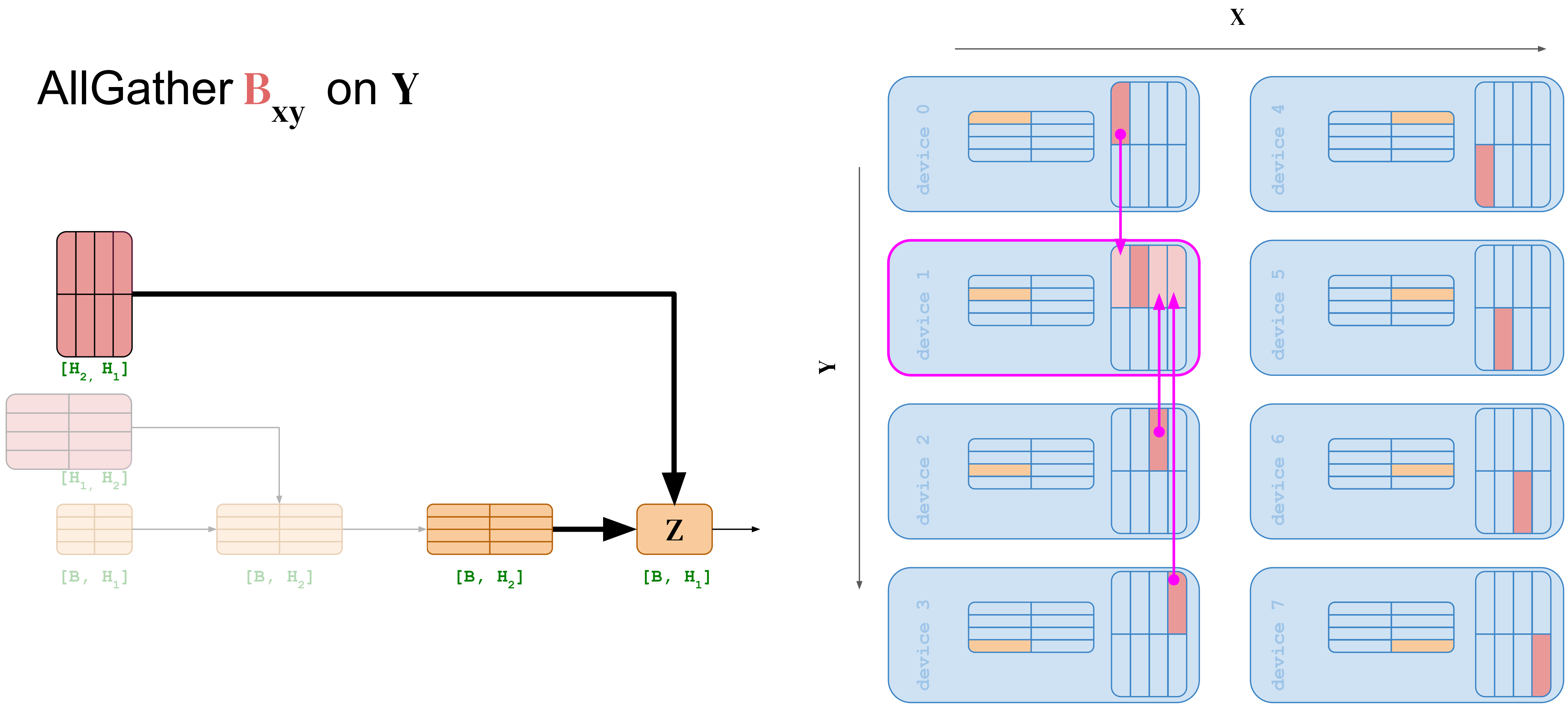
Compte U_{yx} @ B_{xy}
$[B/4,\ H_2/2]@[H_2/2,\ H_1/4]$
$[H_2,\ H_1]$
$[H_1,\ H_2]$
$[B,\ H_1]$
$[B,\ H_2]$
$[B,\ H_2]$
Z
$[B,\ H_1]$
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7



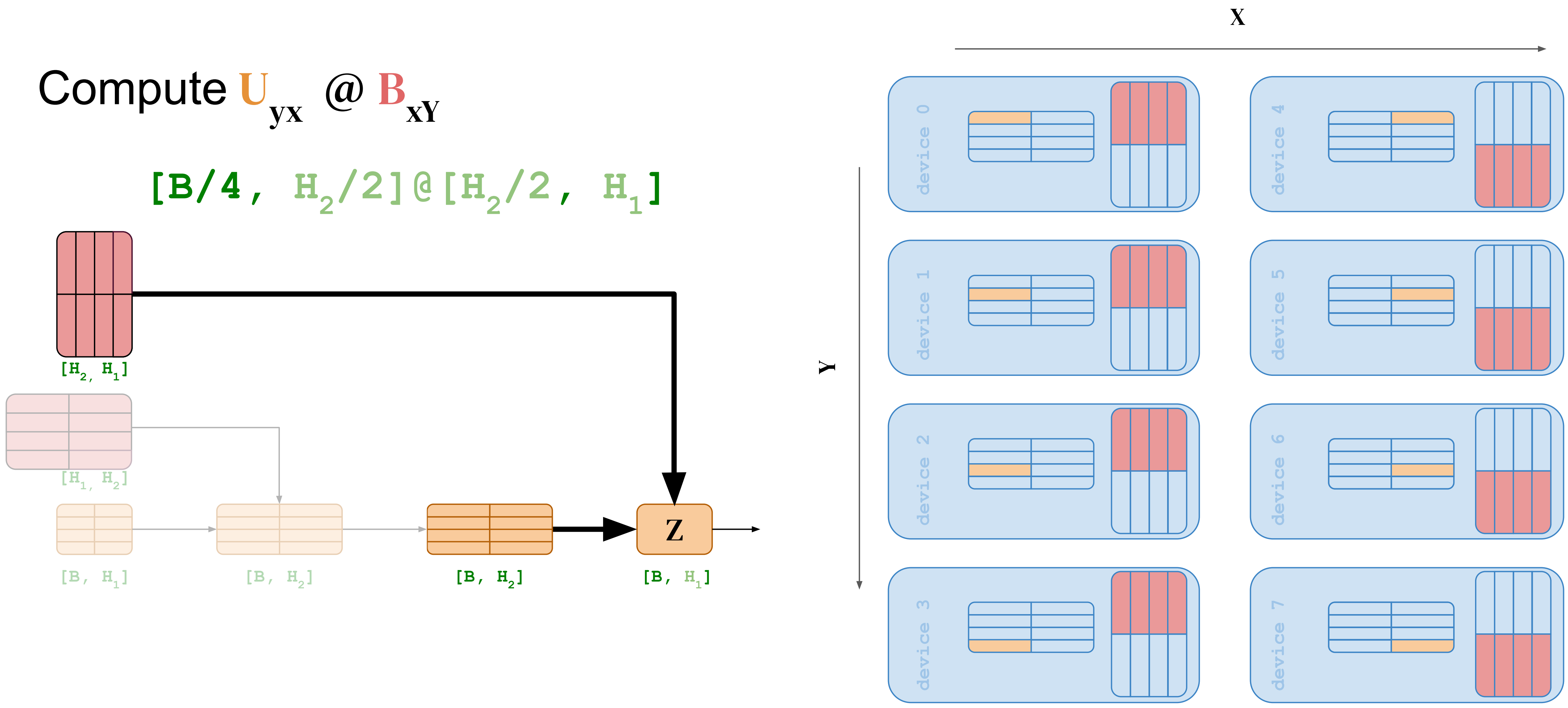
Compte U_{yx} @ B_{xy}
[B/4, H_2/2]@[H_2/2, H_1/4]
[H_2, H_1]
[H_1, H_2]
[B, H_1]
[B, H_2]
[B, H_2]
Z
[B, H_1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

AllGather B_{xy} on Y

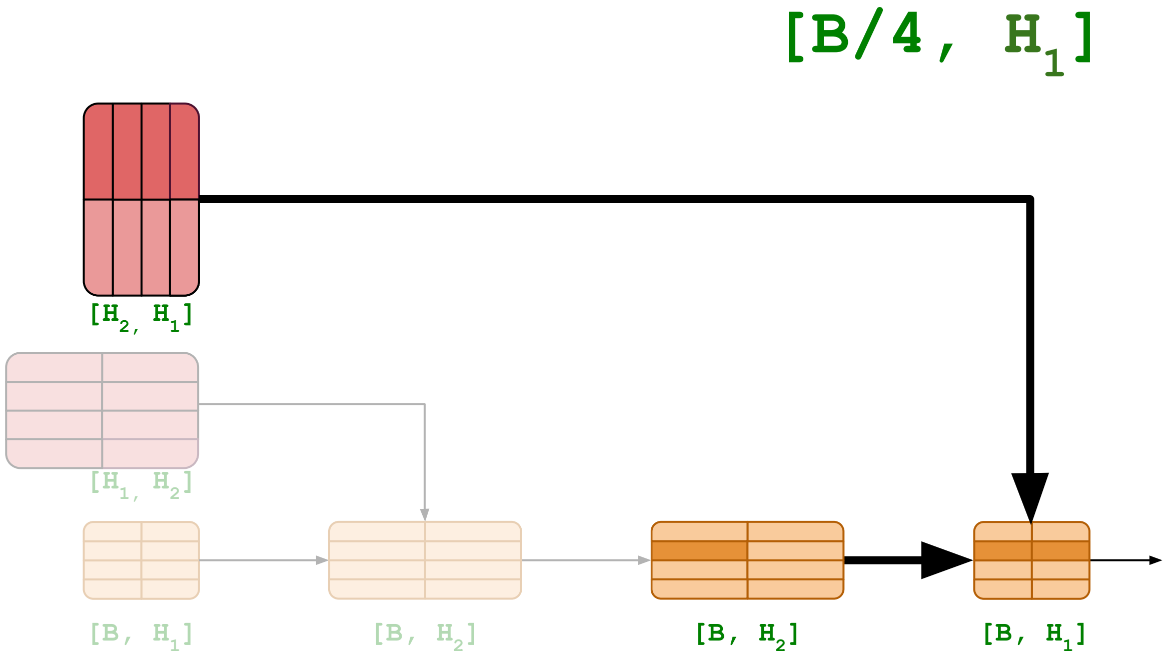


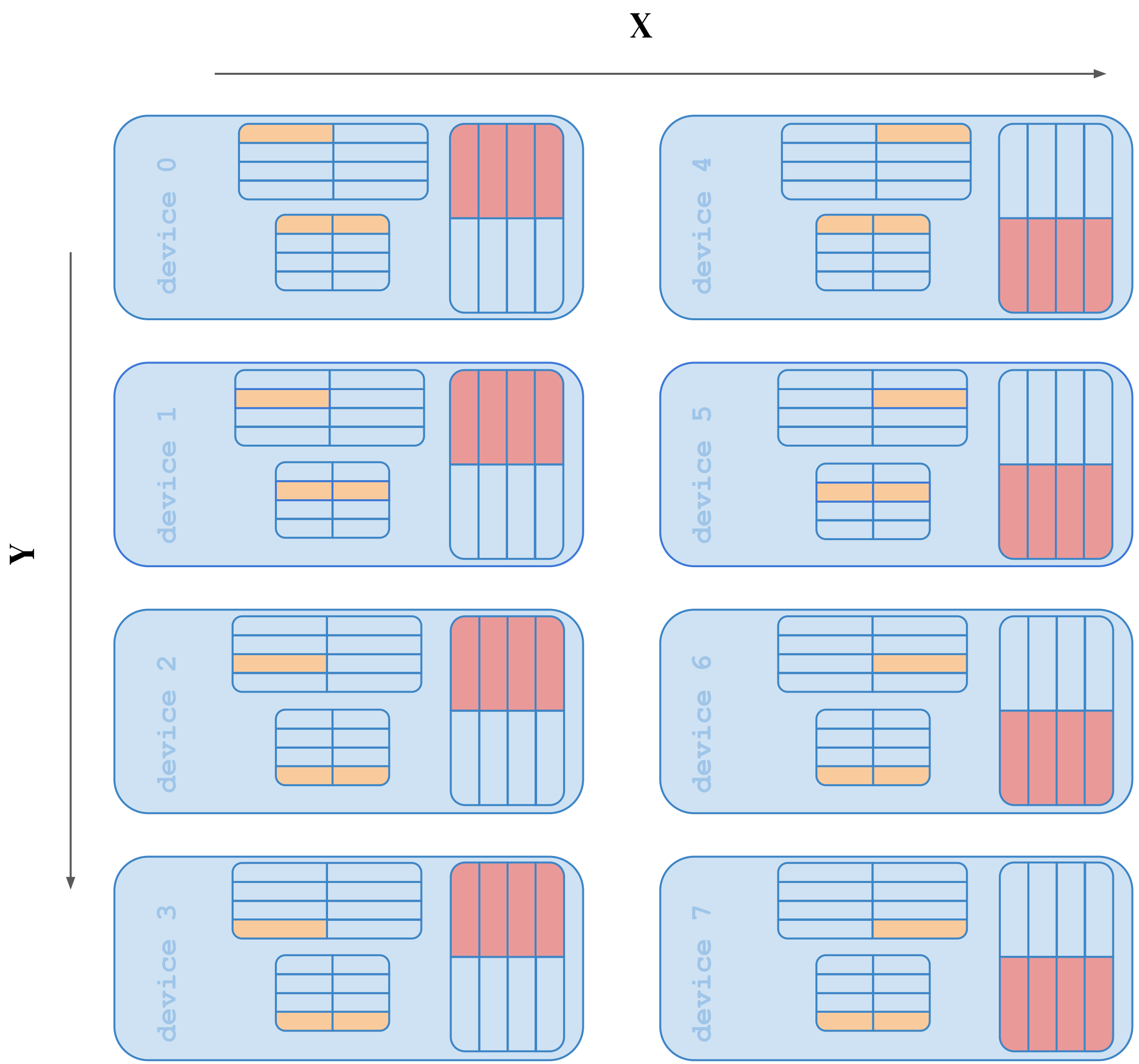


AllGather B xy on Y
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

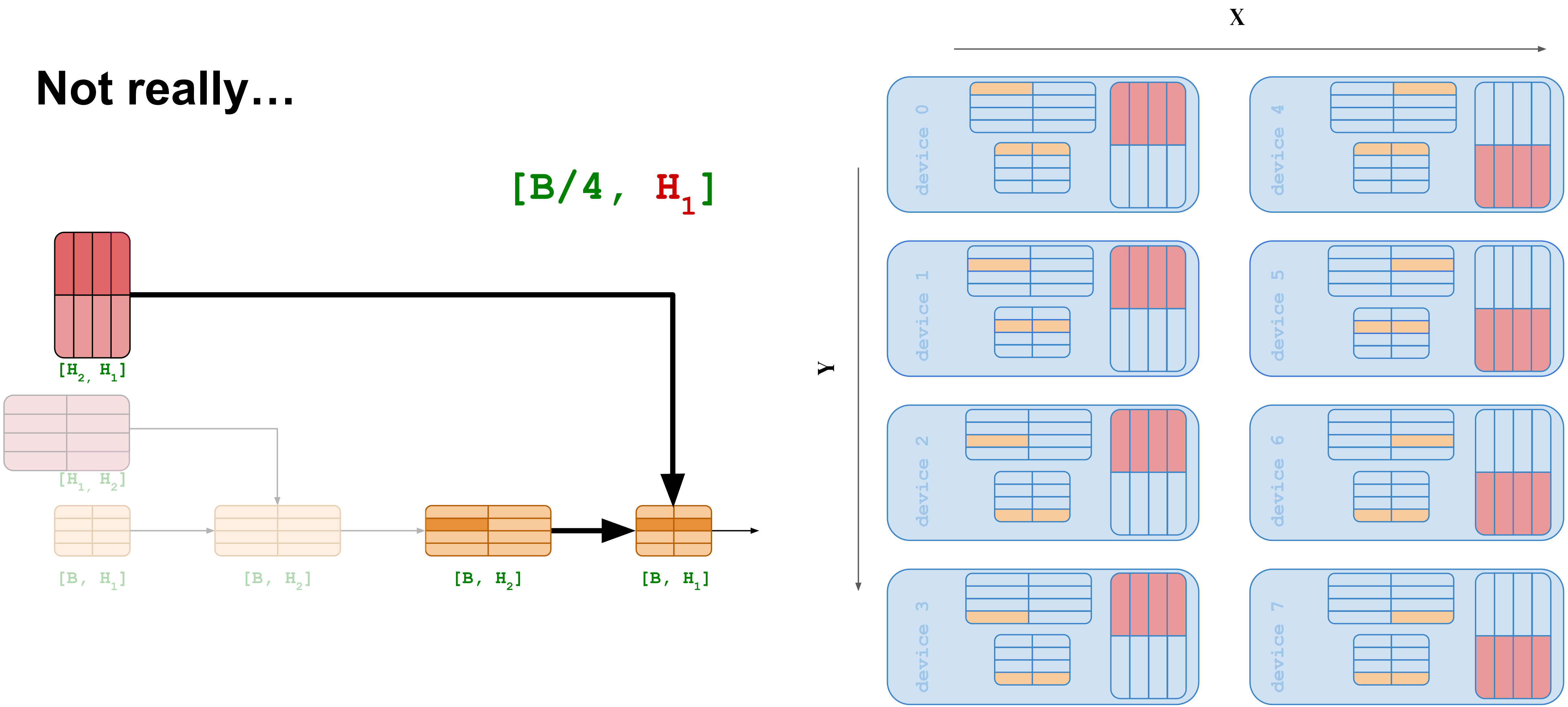

Compute U yx @ B xY
[B/4, H2/2]@[H2/2, H1]
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
[B, H2]
Z
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

Are we done?


[B/4, H1]
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
[B, H2]
[B, H1]


X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

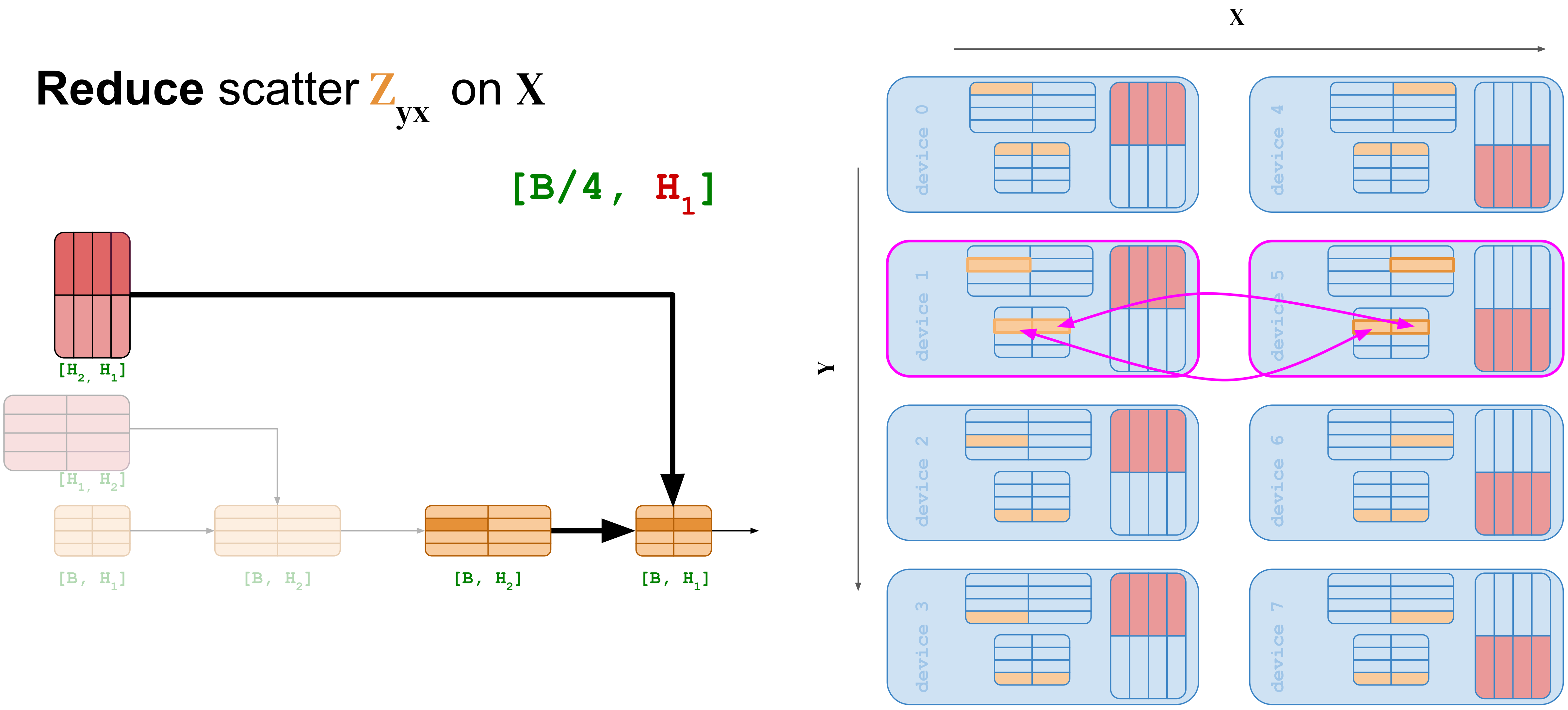
Not really…



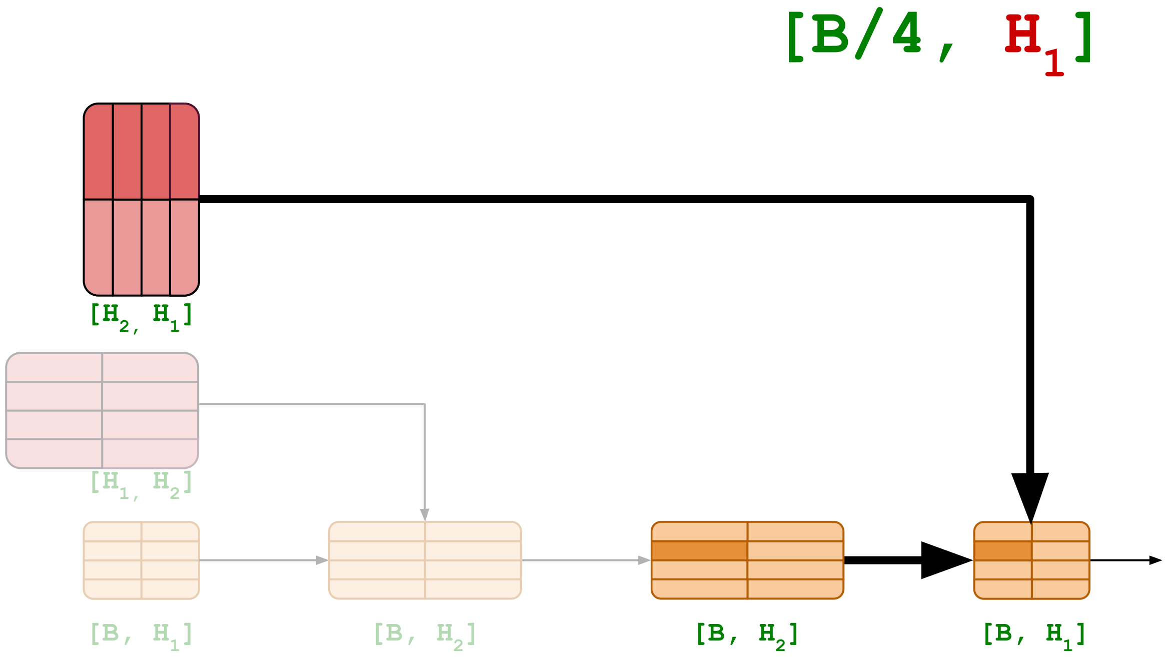
[B/4, H1]
[H2, H1]
[H1, H2]
[B, H1]
[B, H2]
[B, H2]
[B, H1]
X
Y
device 0
device 1
device 2
device 3
device 4
device 5
device 6
device 7

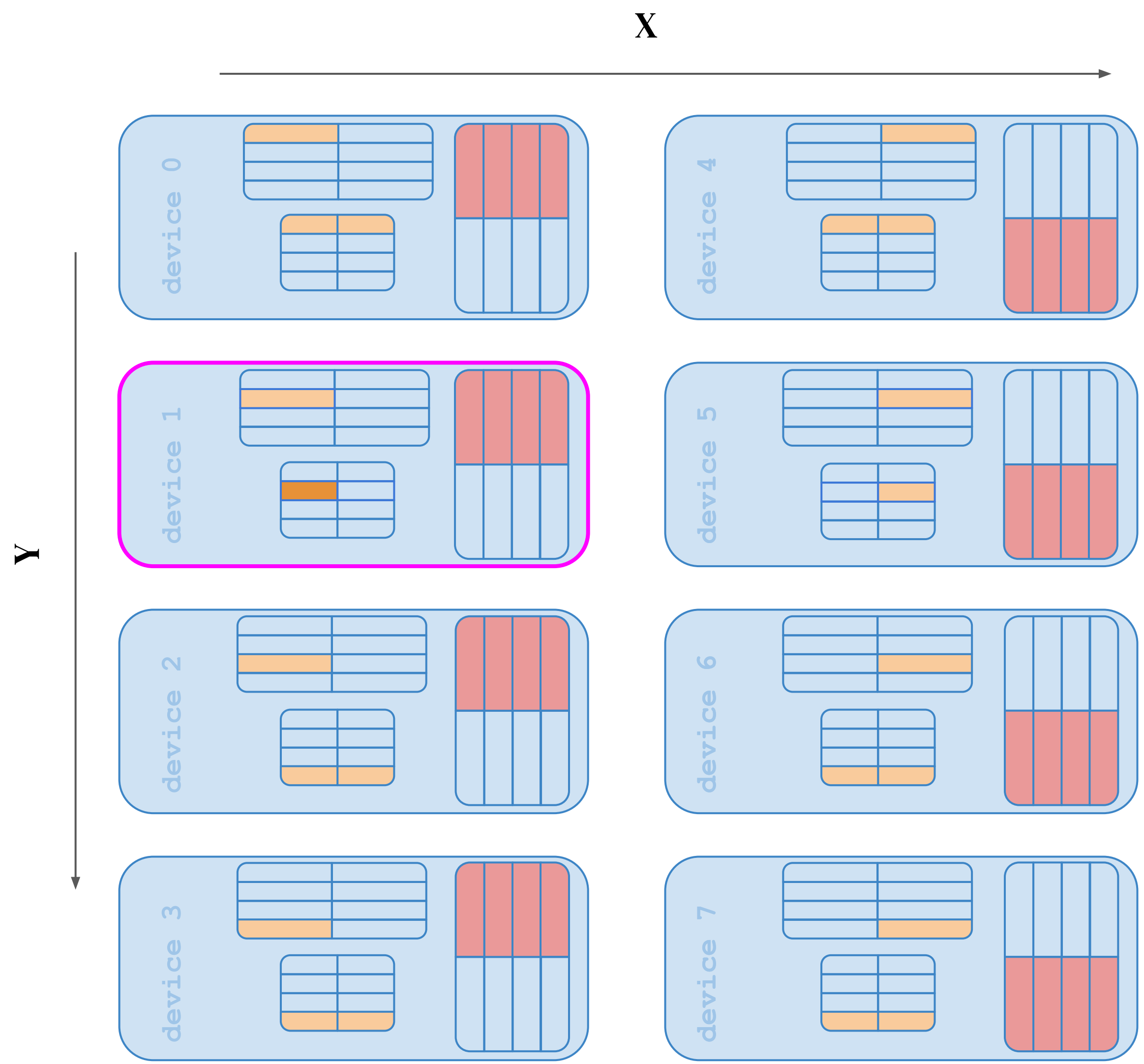
Reduce scatter Z_{yx} on X

[B/4, H_1]

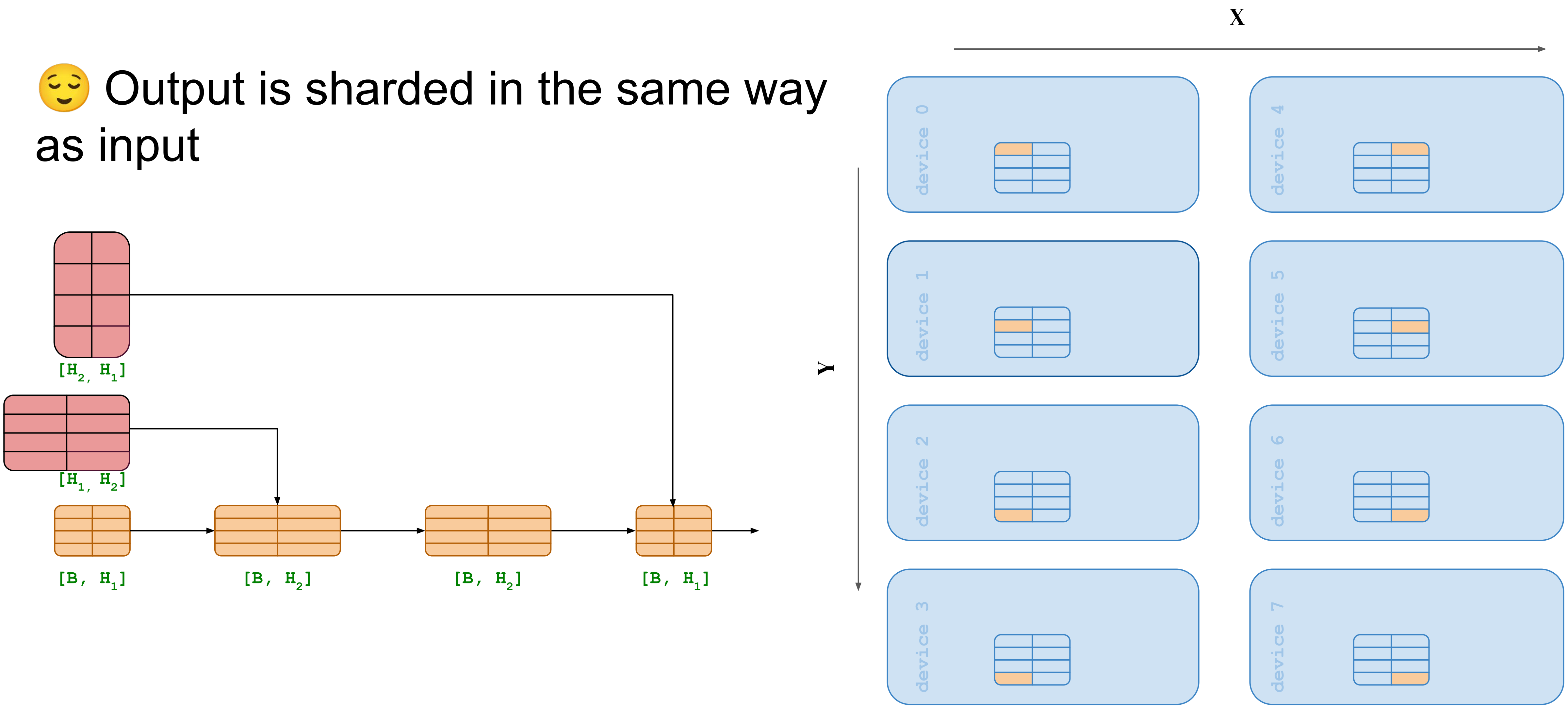


Reduce **scatter** Z_{yx} on Y





😌 Output is sharded in the same way as input



Summary

- We’ve described two main approaches to tenor parallelism
  - Megatron sharding
  - 2D parallelism on a mesh
- If you’re interested in a deep dive how one would shard a transformer model on a mesh GSPMD: General and Scalable Parallelization for ML Computation Graphs paper is an excellent introduction
- Specific kind of employed parallelism in dependent on what hardware we’re using
  - TPU pods?
    - 2D tensor parallelism works great due to fast interconnect
  - GPU clusters?
    - Combination of pipelining and tensor parallelism

What tools does JAX provide?

- **`pmap`**
  - Explicit parallel multi-device programming with leading device dimension and manually calling collective ops on reduction axes (JAX documentation has an excellent introduction to `pmap`).
- **`jax.jit`** with **`jax.Array`**
  - Implicit, compiler-based parallelization system based on user annotation and constraint propagation. Calls GSPMD (paper, blogpost) under the hood, which rewrites the computation and inserts collective ops.
  - Compiler can will try to be smart, but one can enforce strict constraints with `jax.lax.with_sharding_constraint` (official JAX tutorial)

pmap vs GSPMD

```
A_pmap = shard_on_columns(A, 8)
B_pmap = shard_on_rows(B, 8)

@partial(jax.pmap, axis_name="pmap_axis")
def dot_pmap(x, y):
    return jax.lax.psum(x @ y, 'pmap_axis')
```

```
mesh = jax.sharding.Mesh(
    mesh_utils.create_device_mesh((8, ),
    axis_names=("x", ))

dot_explicit = jax.jit(
    lambda x, y: jnp.dot(x,y),
    in_shardings=(
        NamedSharding(mesh, PartitionSpec( None, "x")),
        NamedSharding(mesh, PartitionSpec( "x", None))
    )
)
```