Efficient LLMs

Yatin Nandwani
Research Scientist, IBM Research

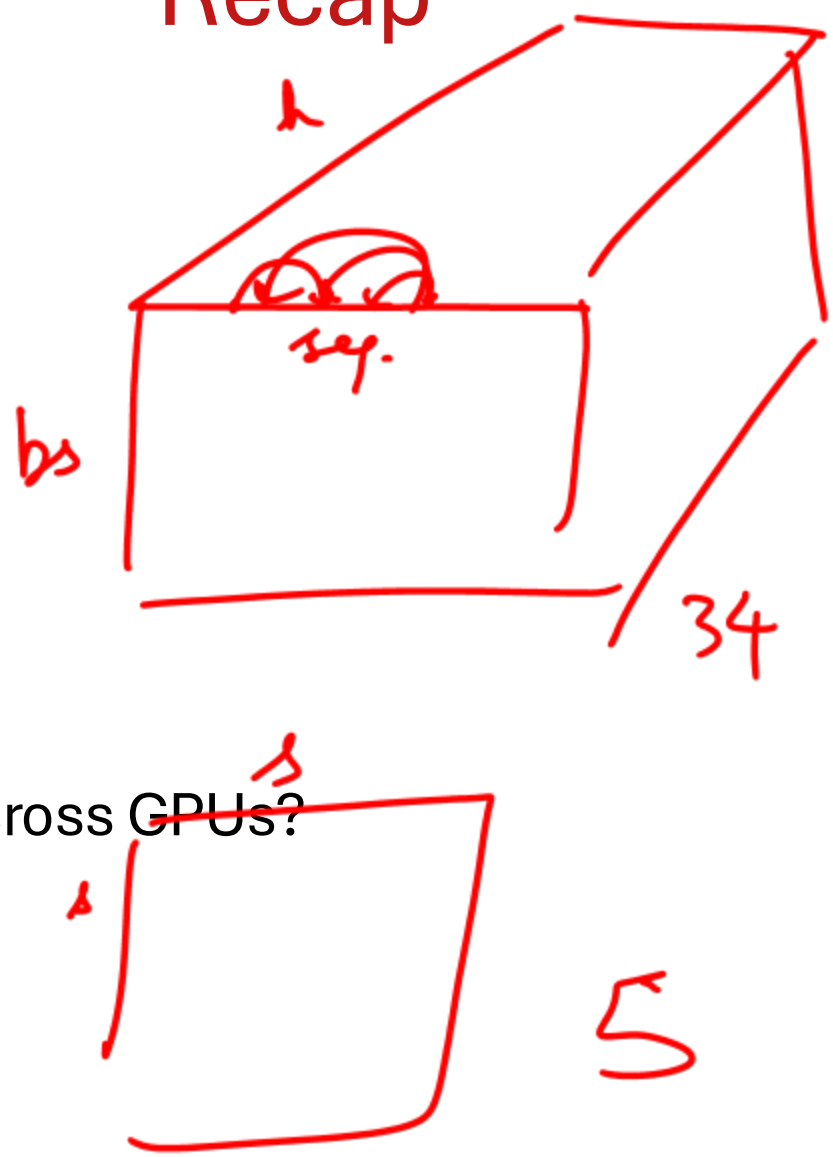
Large Language Models: Introduction and Recent Advances

Semester 1,
2025-2026

ELL881 · AIL821

Recap

- How to train big models on big data?
- What's in the GPU memory during training?
  i. Model weights ; ii. Param gradients iii. Optim states iv. Activations
- What is the size of params / grads / optim states?
- What is the size of activations?
- How to reduce activation memory?
  - Activation re-computation *aka* Gradient checkpointing
- How to increase batch size?
  - Gradient accumulation (run fwd / bwd *k* times before optim.step())
- Can we parallelize grad. Accumulation?
- Can we shard the optim. states, grads, and model params across GPUs?



Command: `torchrun --nproc_per_node 2 train.py`

```python
from torch.distributed.fsdp import fully_shard, FSDPModule
model = Transformer()
for layer in model.layers:
    fully_shard(layer)
fully_shard(model)

assert isinstance(model, Transformer)
assert isinstance(model, FSDPModule)
print(model)
#  FSDPTransformer(
#    (tok_embeddings): Embedding(...)
#    ...
#    (layers): 3 x FSDPTransformerBlock(...)
#    (output): Linear(...)
#  )
```

```python
for _ in range(epochs):
    x = torch.randint(0, vocab_size, (batch_size, seq_len), device=device
    loss = model(x).sum()
    loss.backward()
    optim.step()
    optim.zero_grad()
```

```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset,
    batch_size=32,
)
```

for batch in train_loader:

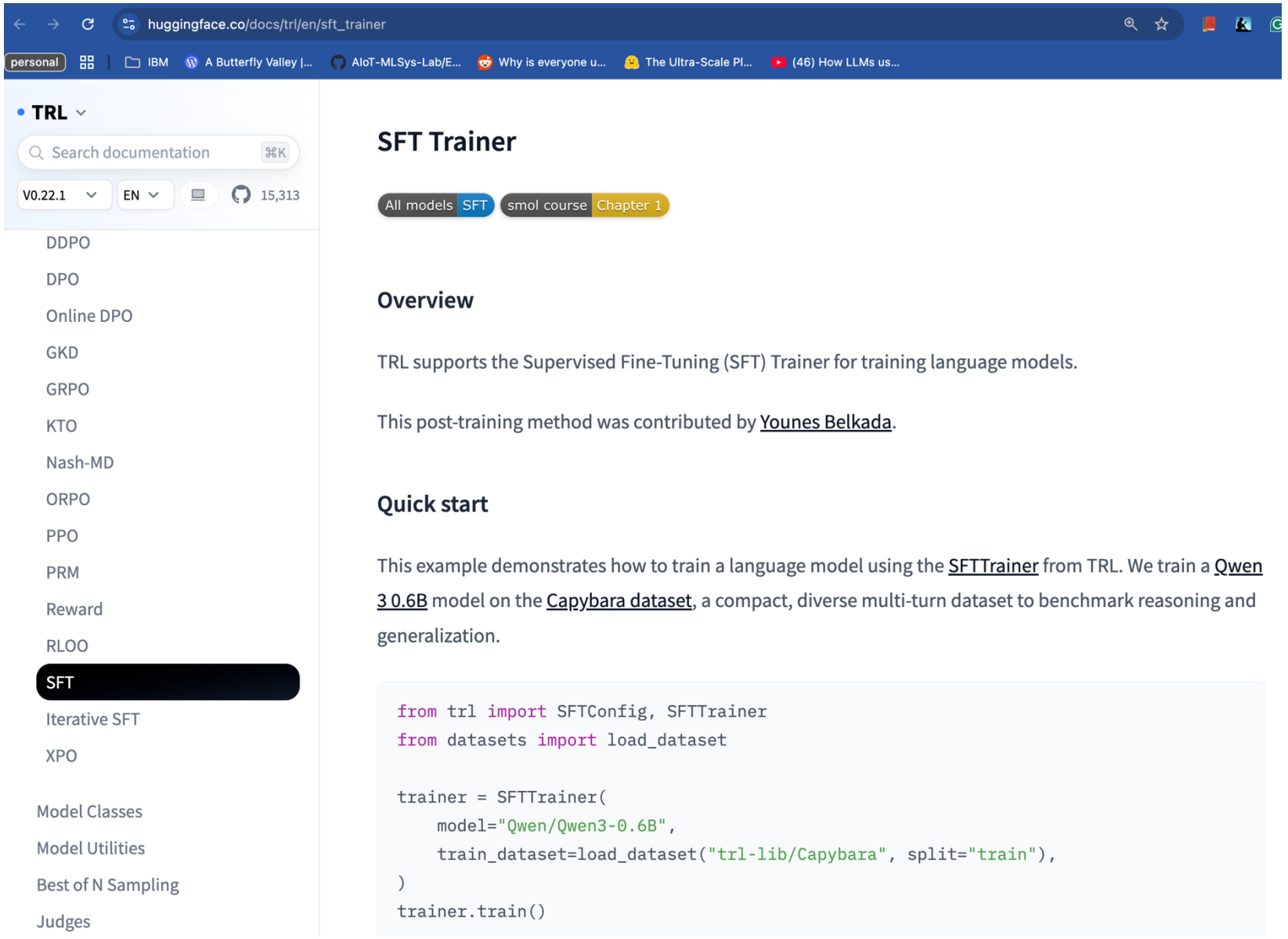
```python
from torch.utils.data import DataLoader

train_loader = DataLoader(
    dataset,
    batch_size=32,
)
```

```python
from torch.utils.data import DataLoader, DistributedSampler
import torch.distributed as dist

dist.init_process_group("nccl")
rank = dist.get_rank()
world_size = dist.get_world_size()

train_sampler = DistributedSampler(
    dataset, num_replicas=world_size, rank=rank,
    shuffle=True,           # shuffle at each epoch
)
train_loader = DataLoader(
    dataset,
    batch_size=local_batch_size,  # per-GPU batch size
    sampler=train_sampler,
)
```



trl

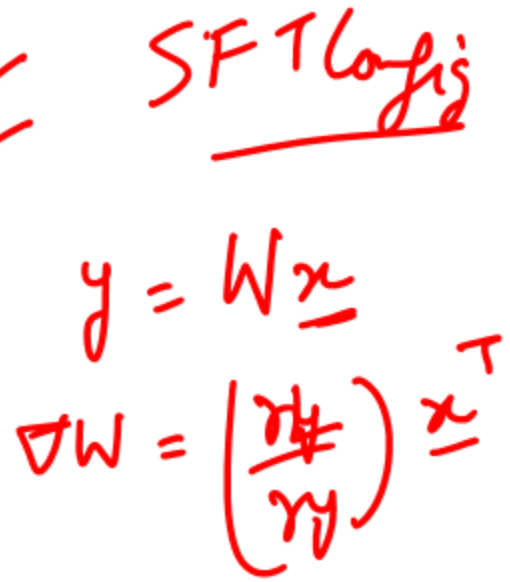
SFTTrainer

`torchrun` is low-level PyTorch-native

`accelerate` is high-level and automates much of the distributed setup.

https://huggingface.co/docs/trl/en/sft_trainer

```
accelerate launch --config_file <path/to/acc/config> trl/scripts/sft.py \
  --model_name_or_path Qwen/Qwen2-0.5B \
  --dataset_name trl-lib/Capybara \
  --learning_rate 2.0e-5 \
  --num_train_epochs 1 \
  --per_device_train_batch_size 2 \
  --gradient_accumulation_steps 8 \
  --gradient_checkpointing \
  --eos_token '<|im_end|>' \
  --eval_strategy steps \
  --eval_steps 100 \
  --output_dir Qwen2-0.5B-SFT \
```



SFTTrainer

torchrun is low-level PyTorch-native

accelerate is high-level and automates much of the distributed setup.

https://huggingface.co/docs/trl/en/sft_trainer

https://huggingface.co/docs/trl/main/en/distributing_training

https://github.com/huggingface/trl/blob/main/trl/scripts/sft.py

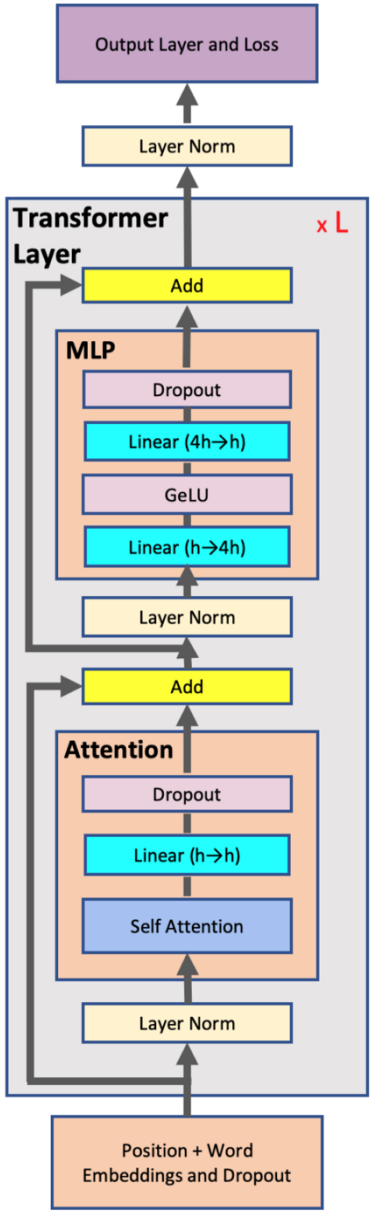
Recap

- How to train big models on big data?
- What's in the GPU memory during training?
  i. Model weights ; ii. Param gradients iii. Optim states iv. Activations
- What is the size of params / grads / optim states?
- What is the size of activations?
- How to reduce activation memory?
  - Activation re-computation *aka* Gradient checkpointing
- How to increase batch size?
  - Gradient accumulation (run fwd / bwd *k* times before optim.step())
- Can we parallelize grad. Accumulation?
- Can we shard the optim. states, grads, and model params across GPUs?

Recap

- How to train big models on big data?
- What's in the GPU memory during training?
  i. Model weights ; ii. Param gradients iii. Optim states iv. Activations
- What is the size of params / grads / optim states?
- What is the size of activations?
- How to reduce activation memory?
  - Activation re-computation *aka* Gradient checkpointing
- How to increase batch size?
  - Gradient accumulation (run fwd / bwd *k* times before optim.step())
- Can we parallelize grad. Accumulation?
- Can we shard the optim. states, grads, and model params across GPUs?
- Still not sufficient for big models (e.g 70B Llama). Can we split one input sequence across GPUs?

Let us revisit activation memory



Total	$\mathbf{34 * seq * bs * h + 5 * n_{heads} * seq^2 * bs}$
MLP Block	$\mathbf{19 * seq * bs * h}$
D/o mask	$1 * seq * bs * h$
Linear *(4h -> h)*	$8 * seq * bs * h$
GELU	$8 * seq * bs * h$
Linear *(h -> 4h)*	$2 * seq * bs * h$
Layer Norm	$\mathbf{2 * seq * bs * h}$
Attention Block	$\mathbf{11 * seq * bs * h + 5 * n_{heads} * seq^2 * bs}$
Layer Norm	$\mathbf{2 * seq * bs * h}$

Memory for Activations

$$m_{act} = L * \left(\begin{array}{c} 34 * seq * bs * h \\ + \\ 5 * n_{heads} * seq^2 * bs \end{array} \right)$$

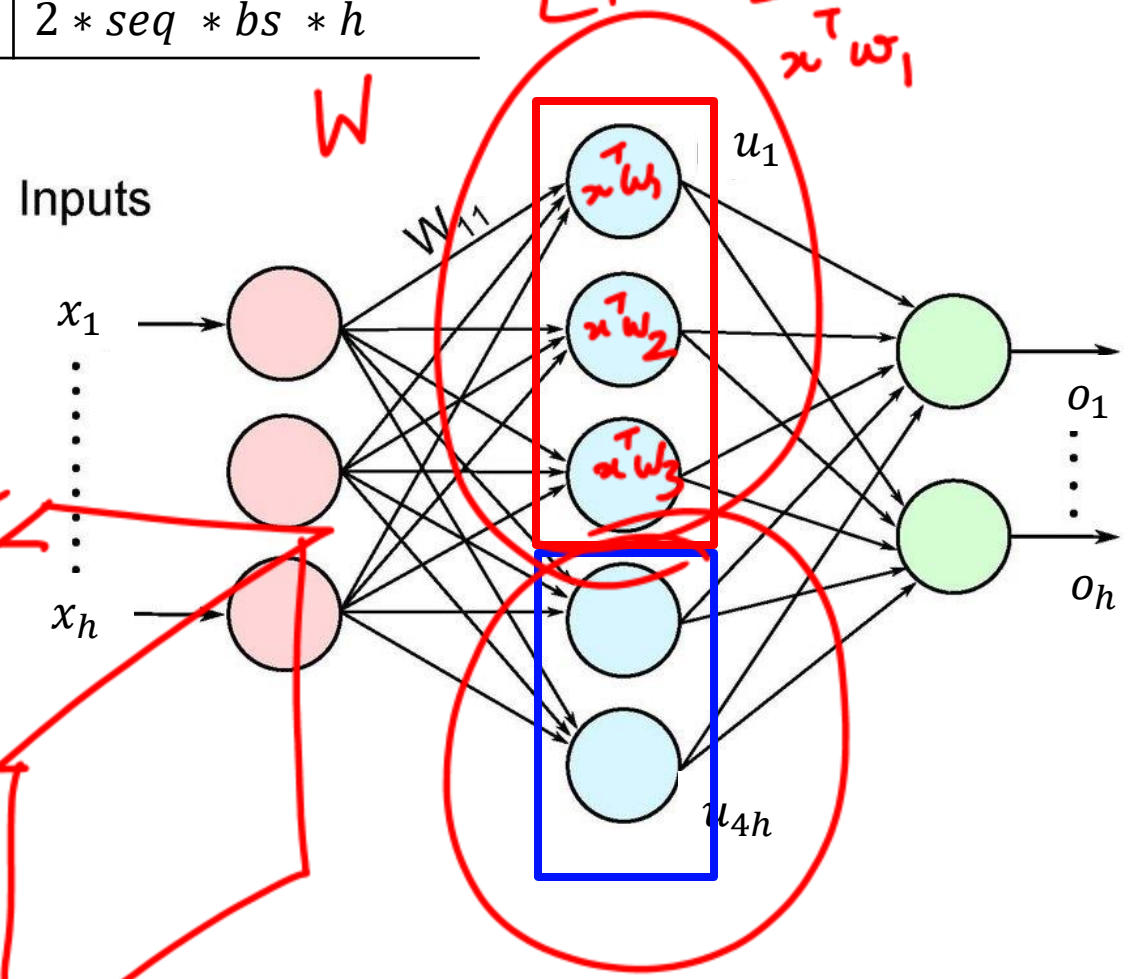
- Scales Linearly with batch size
- Quadratically with the sequence length

Korthikanti etal. 2022, Reducing Activation Recomputation in Large Transformer Models

MLP Block

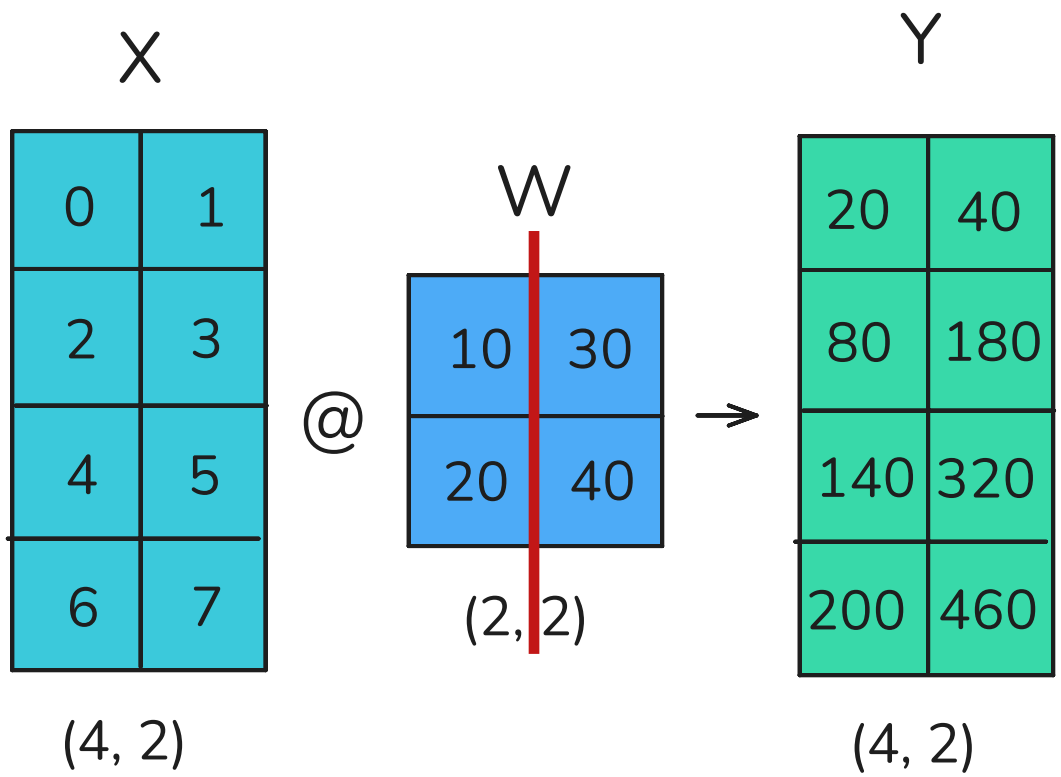
Linear *(4h -> h)*	$8 * seq * bs * h$
GELU	$8 * seq * bs * h$
Linear *(h -> 4h)*	$2 * seq * bs * h$

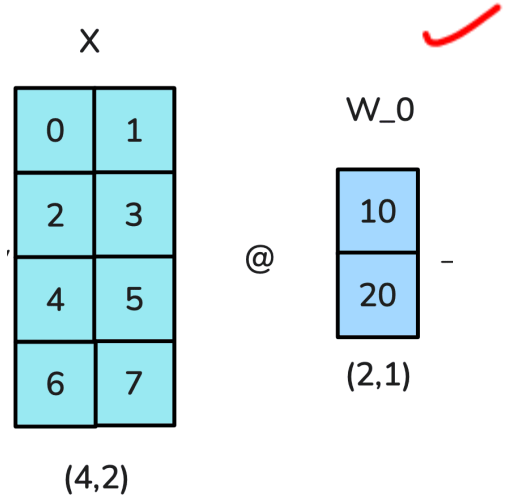
- neurons are independent of
- Can we compute them independently on different GPUs?
- How will it split the weight matrix?

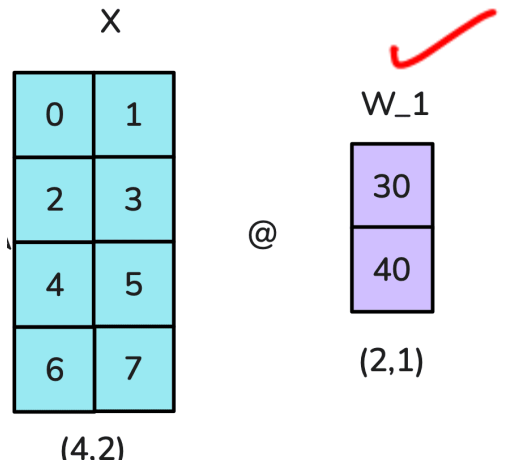


1. $A \cdot B = A \cdot [B_1 \quad B_2 \quad \cdots] = [AB_1 \quad AB_2 \quad \cdots]$

Column Linear

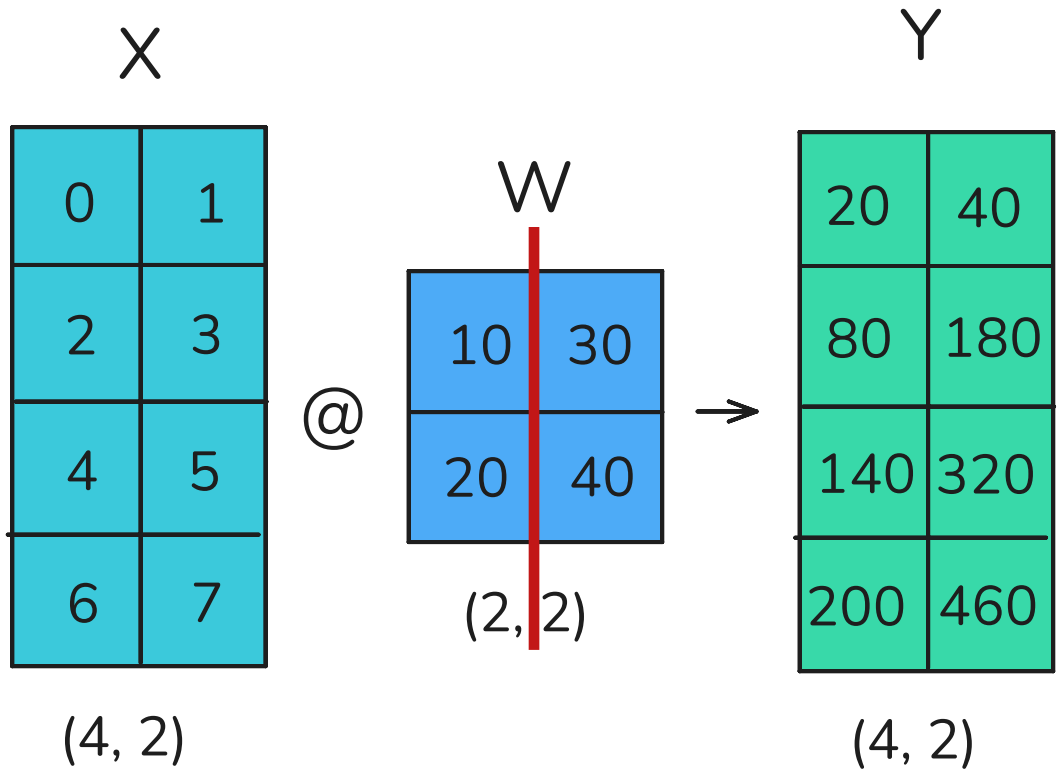


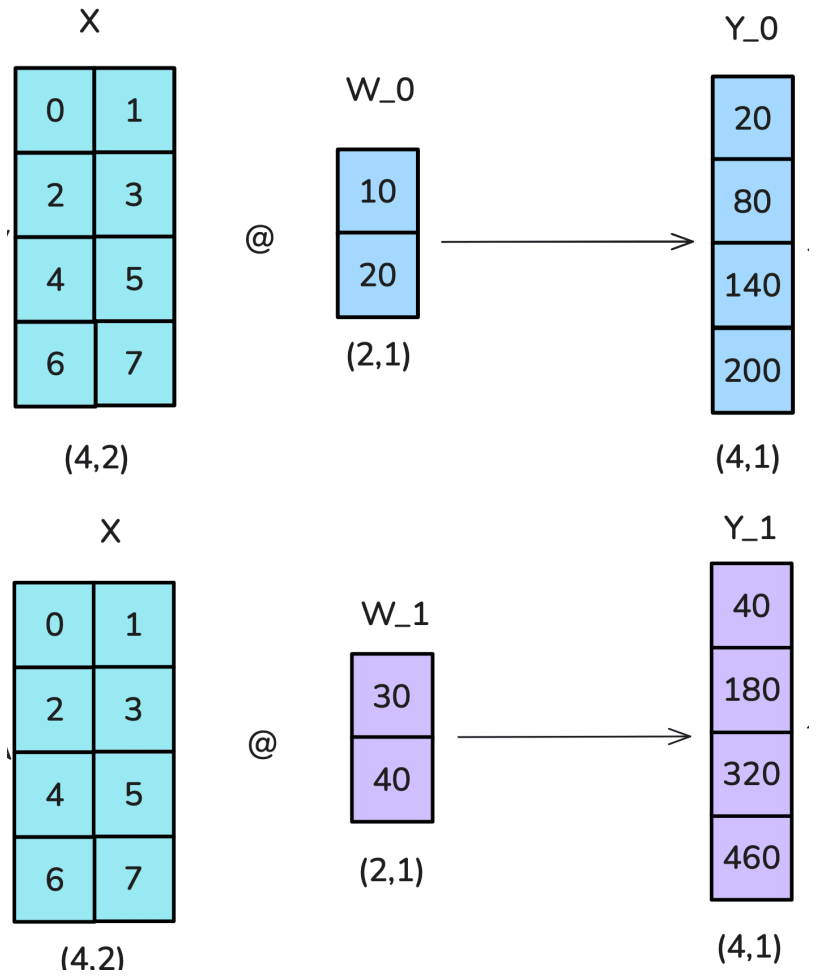




1. $A \cdot B = A \cdot \begin{bmatrix} B_1 & B_2 & \cdots \end{bmatrix} = \begin{bmatrix} AB_1 & AB_2 & \cdots \end{bmatrix}$

Column Linear



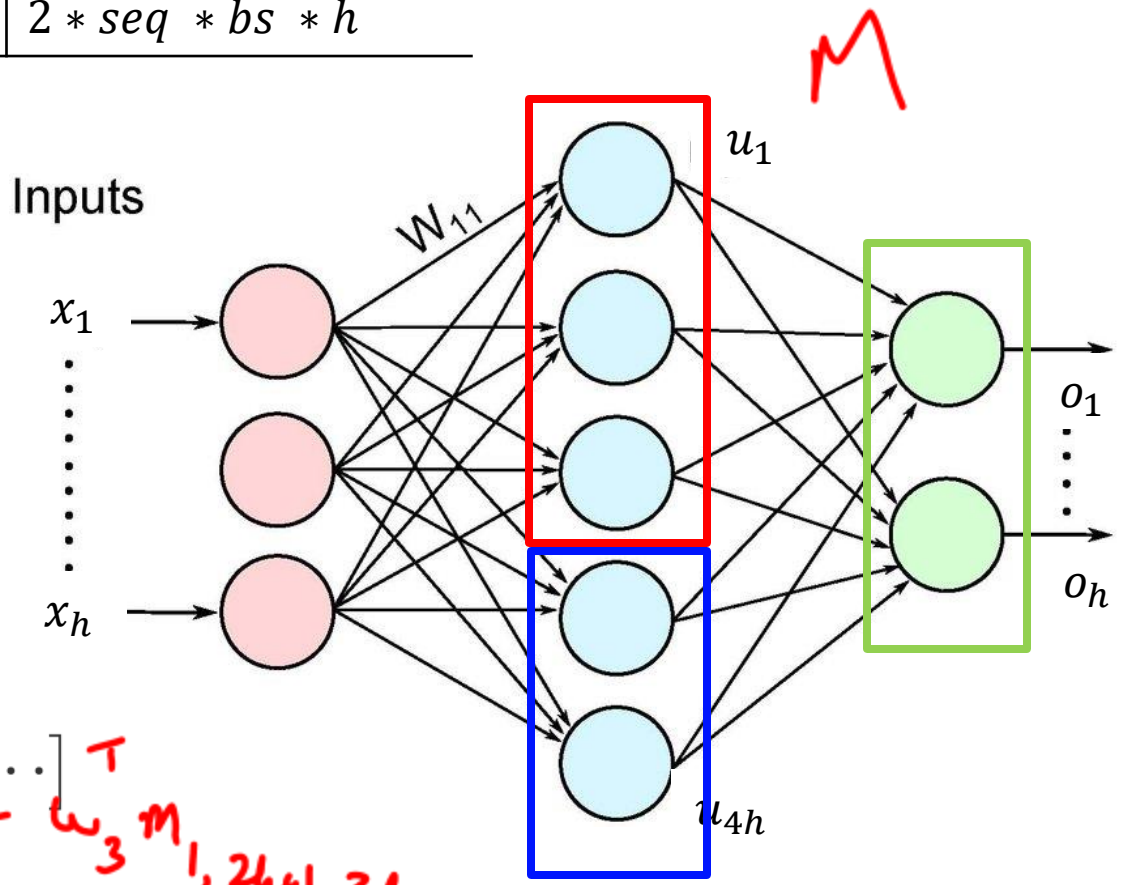


1. $A \cdot B = A \cdot \begin{bmatrix} B_1 & B_2 & \cdots \end{bmatrix} = \begin{bmatrix} AB_1 & AB_2 & \cdots \end{bmatrix}$

MLP Block

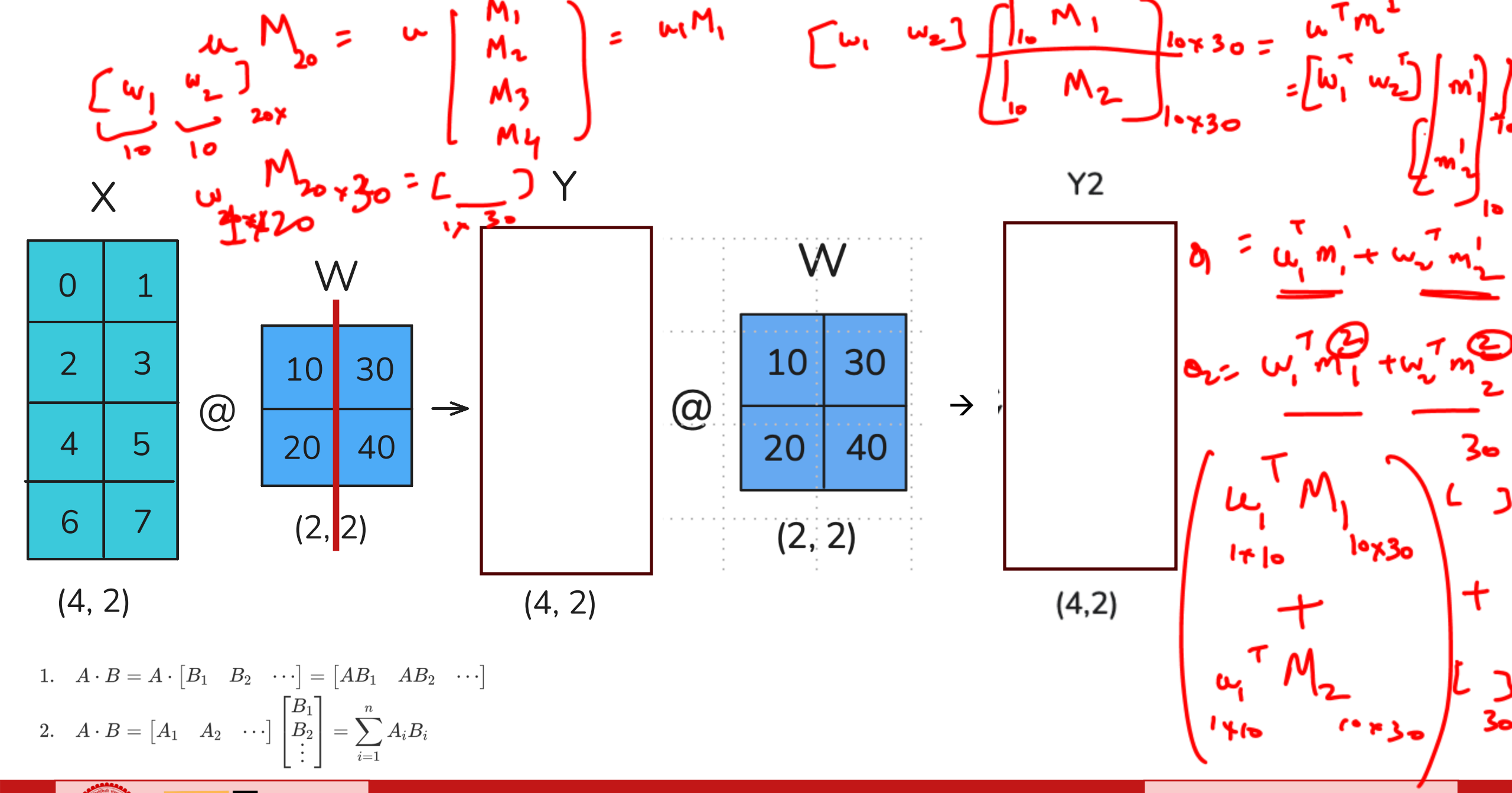
Linear *(4h -> h)*	$8 * seq * bs * h$
GELU	$8 * seq * bs * h$
Linear *(h -> 4h)*	$2 * seq * bs * h$

- Focus on the 2nd Linear Layer and output neurons
- Both [ ] and [ ] neurons contribute to all the output neurons
- Can we split the computation on two GPUs?

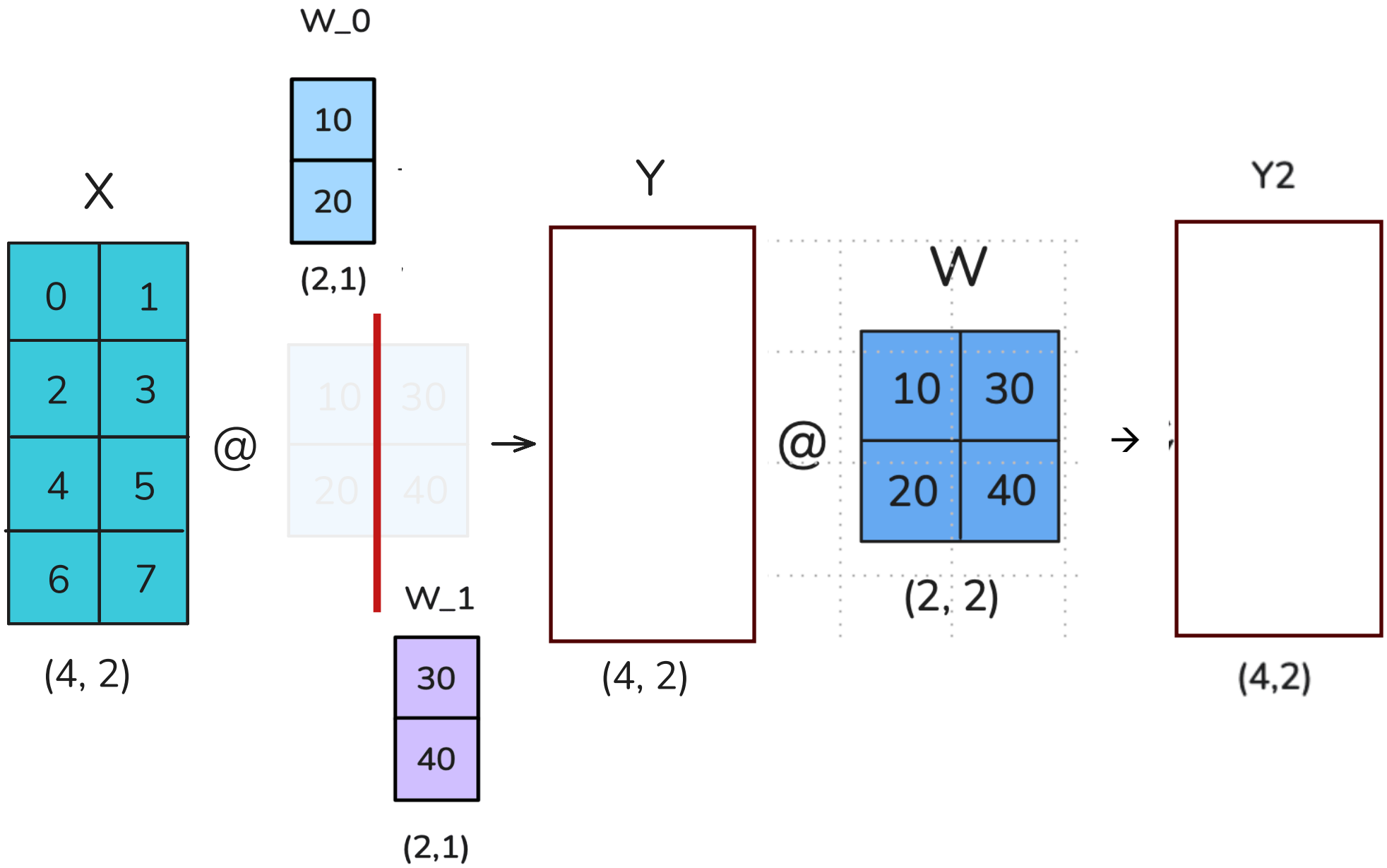


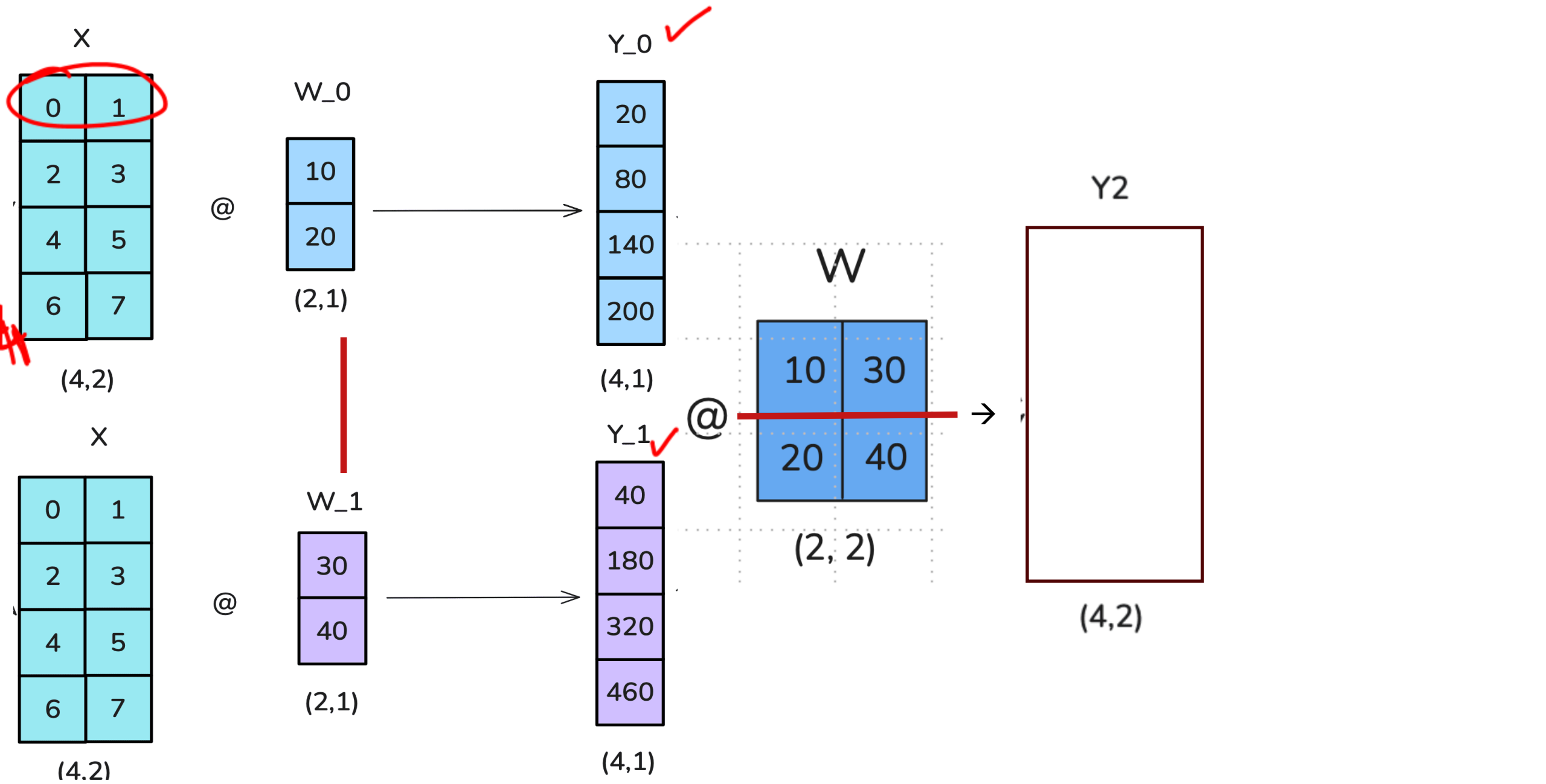
1. $A \cdot B = A \cdot [B_1 \; B_2 \; \ldots] = [AB_1 \; AB_2 \; \ldots]$

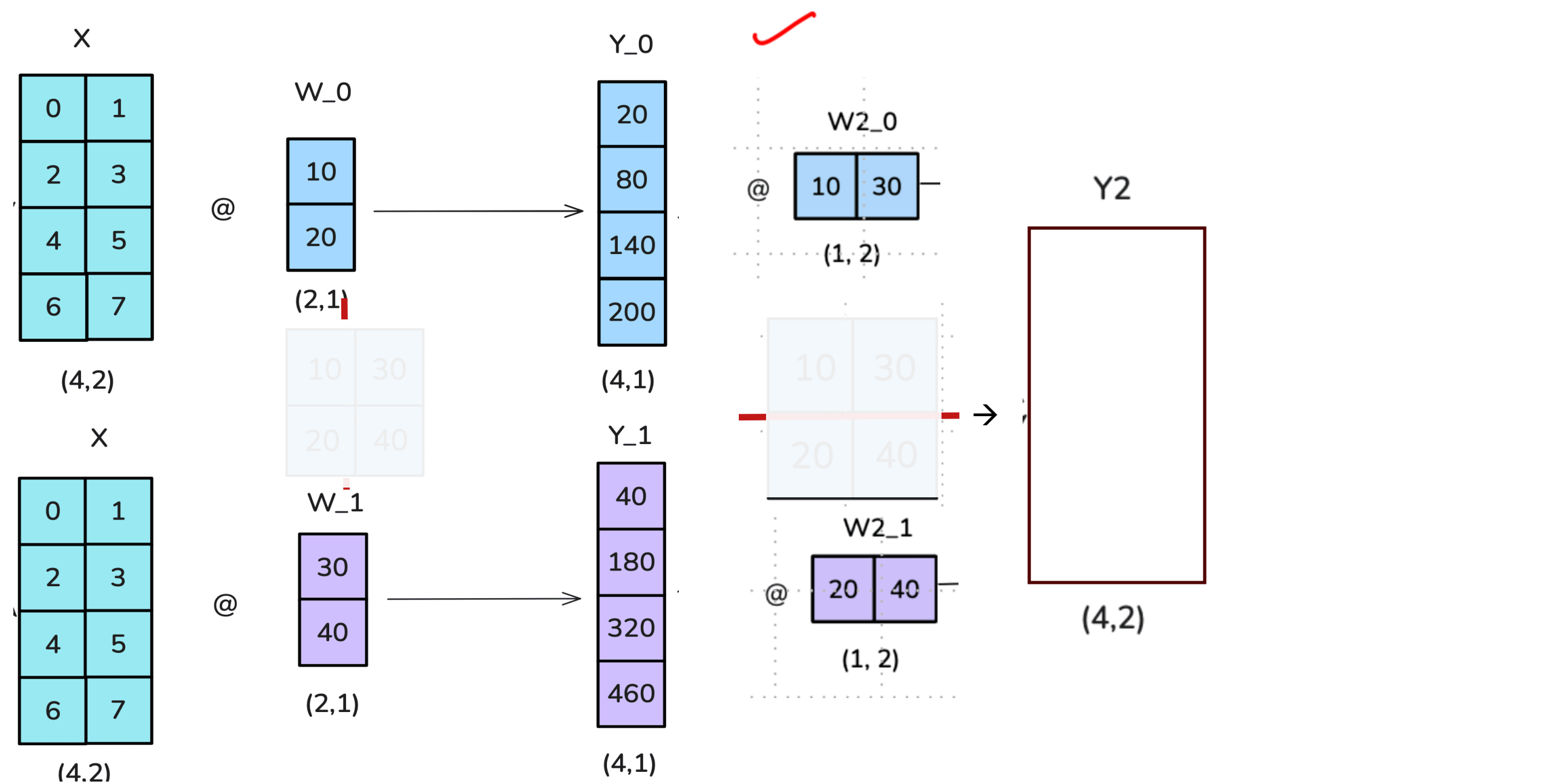
2. $A \cdot B = [A_1 \; A_2 \; \ldots] \begin{bmatrix} B_1 \\ B_2 \\ \vdots \end{bmatrix} = \sum_{i=1}^{n} A_i B_i$

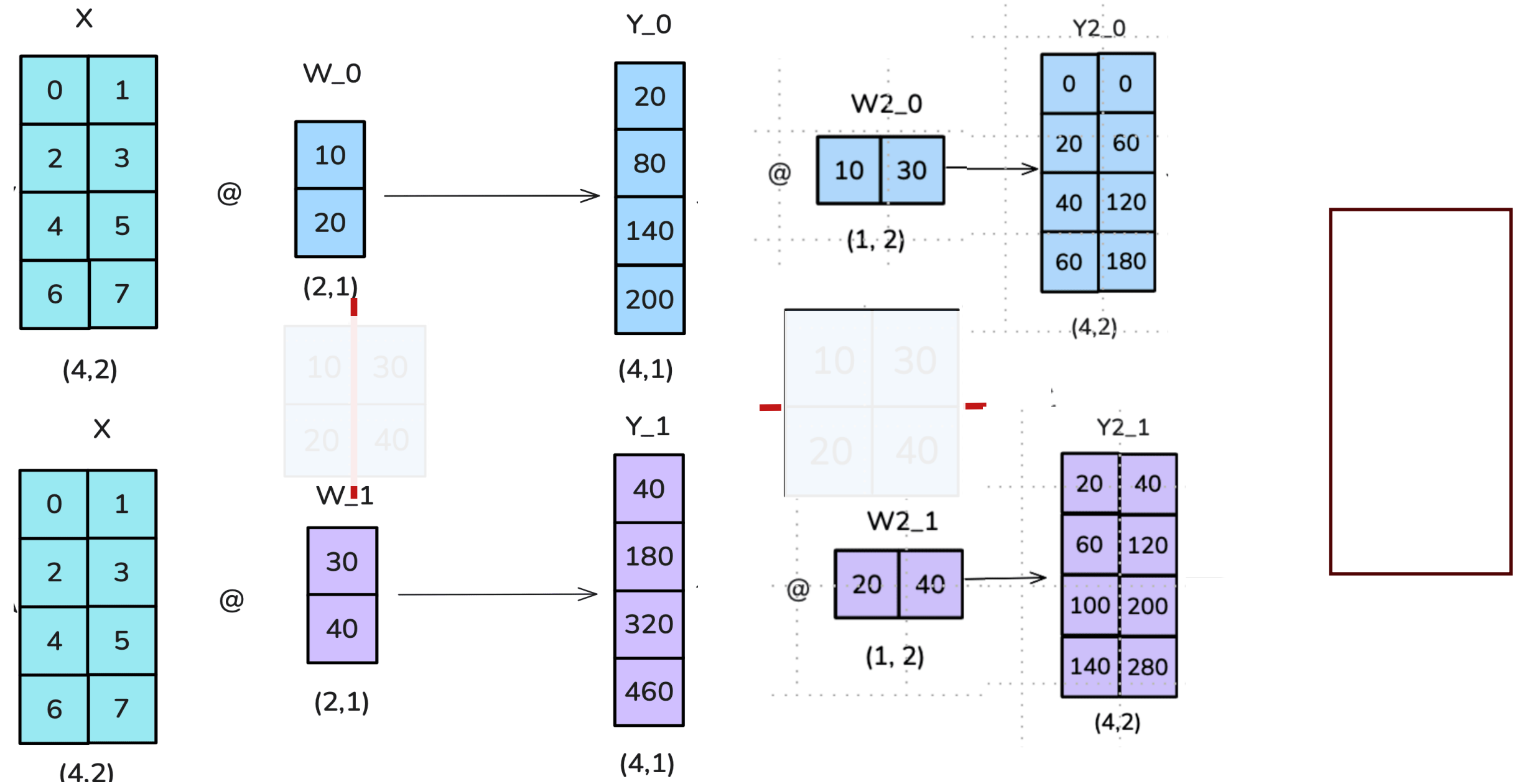


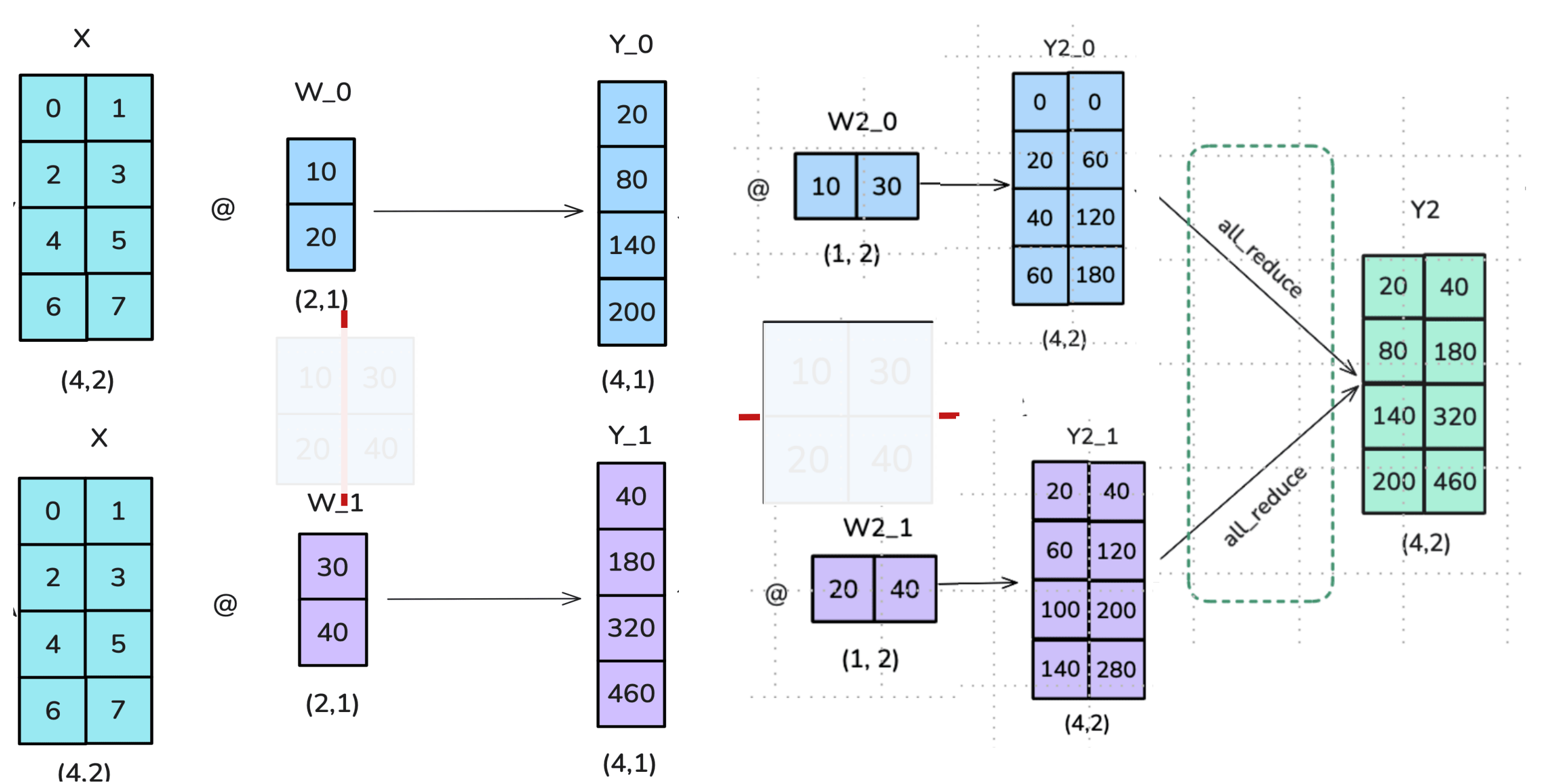
1. $A \cdot B = A \cdot [B_1 \quad B_2 \quad \cdots] = [AB_1 \quad AB_2 \quad \cdots]$
2. $A \cdot B = [A_1 \quad A_2 \quad \cdots] \begin{bmatrix} B_1 \\ B_2 \\ \vdots \end{bmatrix} = \sum_{i=1}^{n} A_i B_i$


W_0
10
20
(2,1)
X
0
1
2
3
4
5
6
7
@
10
30
20
40
W_1
30
40
(2,1)
(4, 2)
Y
(4, 2)
@
W
10
30
20
40
(2, 2)
Y2
(4,2)

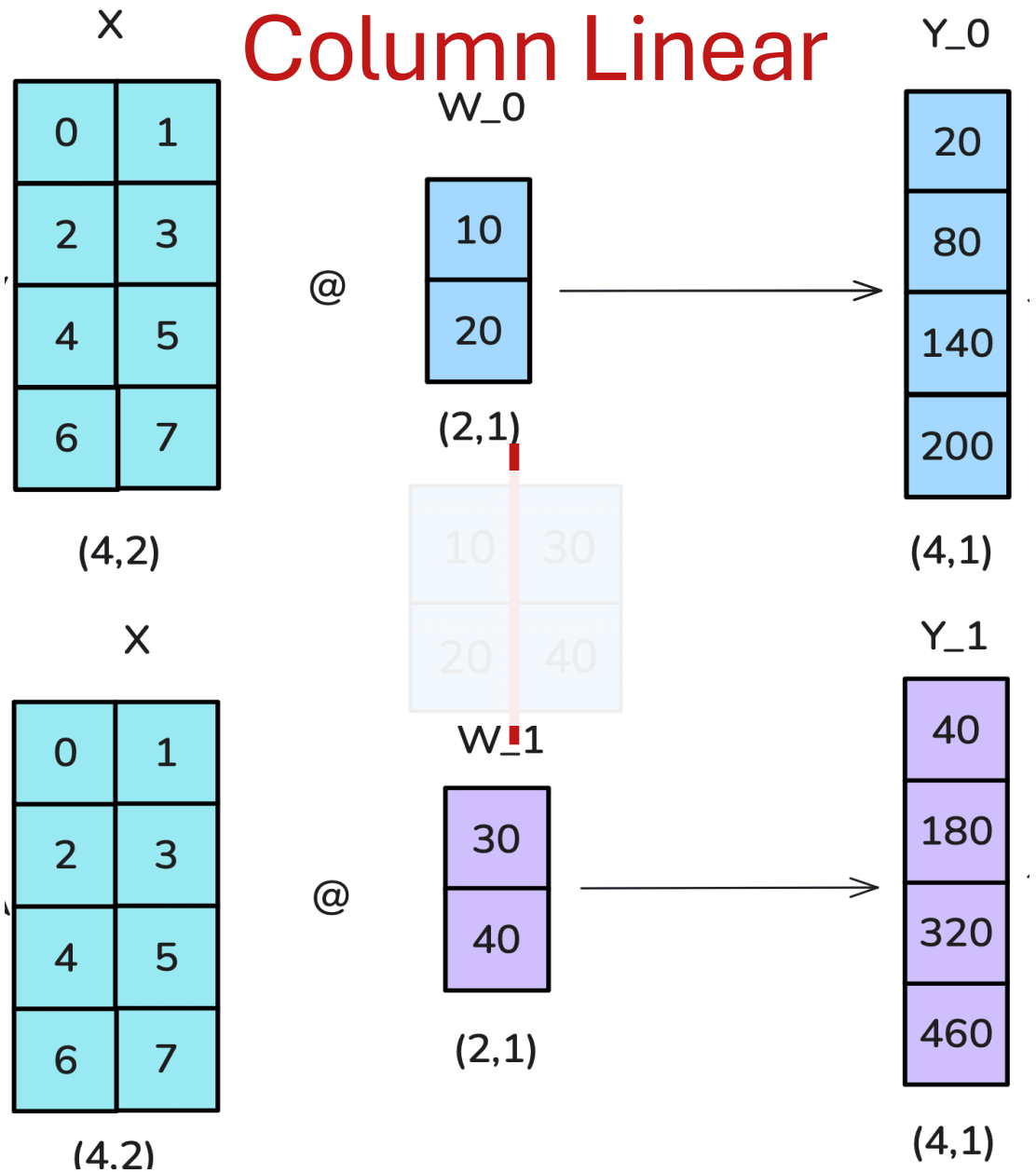

X
0
1
2
3
4
5
6
7
(4,2)
W_0
10
20
(2,1)
@
Y_0
20
80
140
200
(4,1)
W_1
30
40
(2,1)
@
Y_1
40
180
320
460
(4,1)
X
0
1
2
3
4
5
6
7
(4,2)
W
10
30
20
40
(2, 2)
@
Y2
(4,2)


X
0 1
2 3
4 5
6 7
(4,2)
@
W_0
10
20
(2,1)
Y_0
20
80
140
200
(4,1)
W2_0
@
10 30
(1, 2)
Y2
(4,2)
10 30
20 40
X
0 1
2 3
4 5
6 7
(4,2)
@
W_1
30
40
(2,1)
Y_1
40
180
320
460
(4,1)
W2_1
@
20 40
(1, 2)

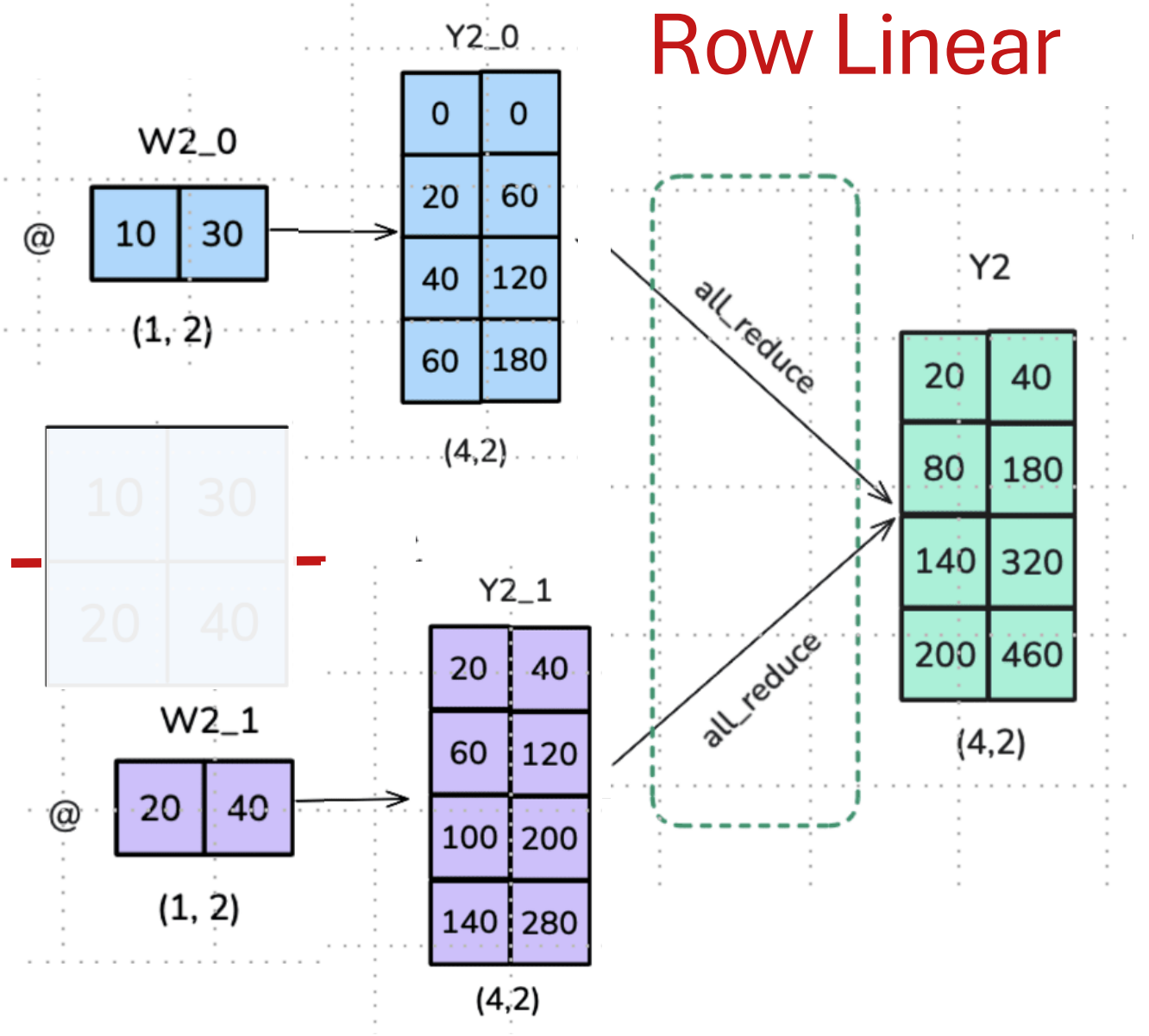

X
0 1
2 3
4 5
6 7
(4,2)
@
W_0
10
20
(2,1)
Y_0
20
80
140
200
(4,1)
Y2_0
@
W2_0
10 30
(1, 2)
0 0
20 60
40 120
60 180
(4,2)
10 30
20 40
X
0 1
2 3
4 5
6 7
(4,2)
@
W_1
30
40
(2,1)
Y_1
40
180
320
460
(4,1)
Y2_1
@
W2_1
20 40
(1, 2)
20 40
60 120
100 200
140 280
(4,2)


X
0 1
2 3
4 5
6 7
(4,2)
@
W_0
10
20
(2,1)
Y_0
20
80
140
200
(4,1)
X
0 1
2 3
4 5
6 7
(4,2)
@
W_1
30
40
(2,1)
Y_1
40
180
320
460
(4,1)
W2_0
@
10 30
(1, 2)
Y2_0
0 0
20 60
40 120
60 180
(4,2)
W2_1
@
20 40
(1, 2)
Y2_1
20 40
60 120
100 200
140 280
(4,2)
all_reduce
all_reduce
Y2
20 40
80 180
140 320
200 460
(4,2)

Column Linear

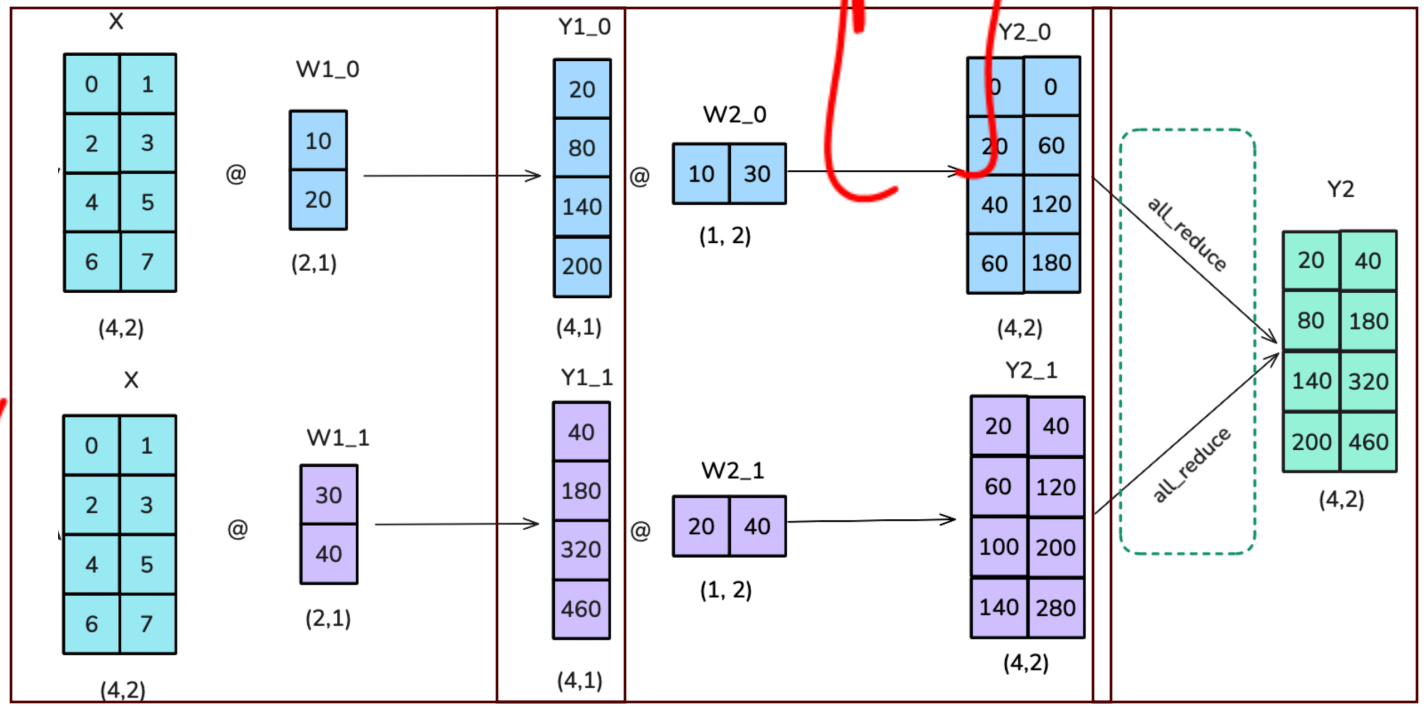


Row Linear



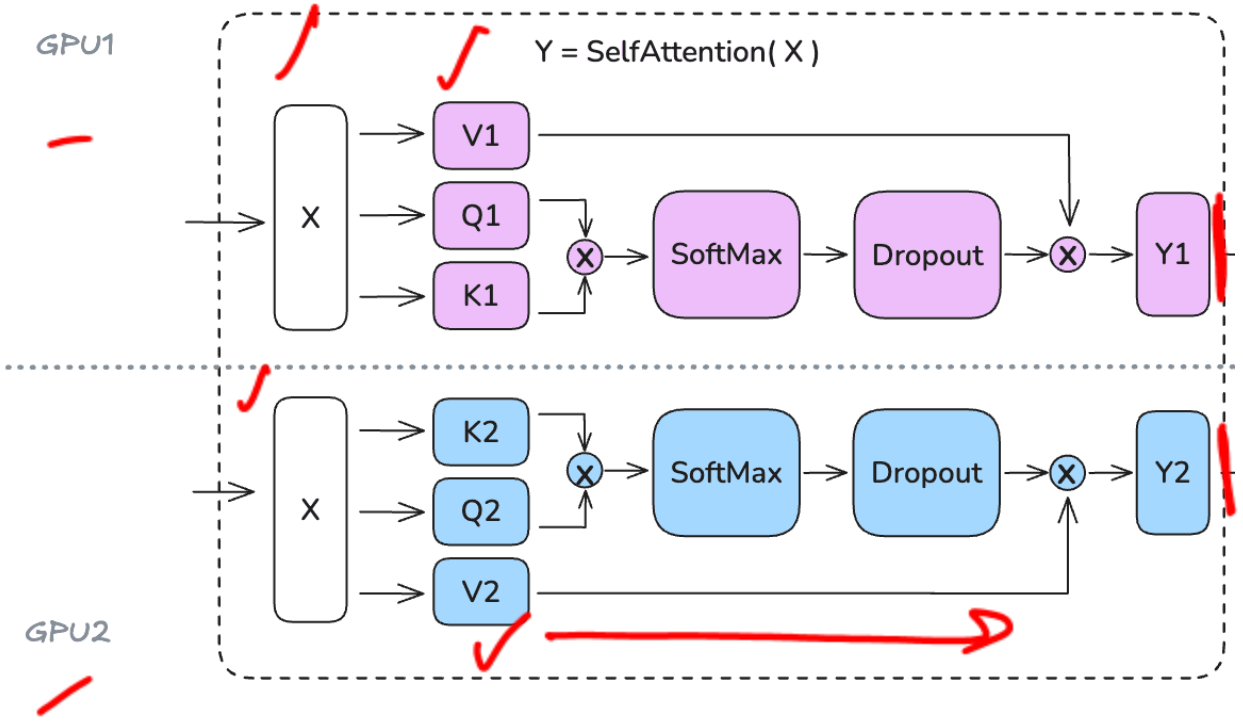
Tensor Parallelism in MLP Block

- Use *column-linear* to split the 1st layer:
  - compute different neurons on different GPUs
  - Each GPU compute $\frac{4h}{TP}$ activations
- Use *row-linear* to split the 2nd layer:
  - Each GPU acts on different neurons and computes partial output
  - No need to communicate the intermediate activations across GPUs → reduction in activation memory!
- Use *all_reduce* to communicate the partial outputs



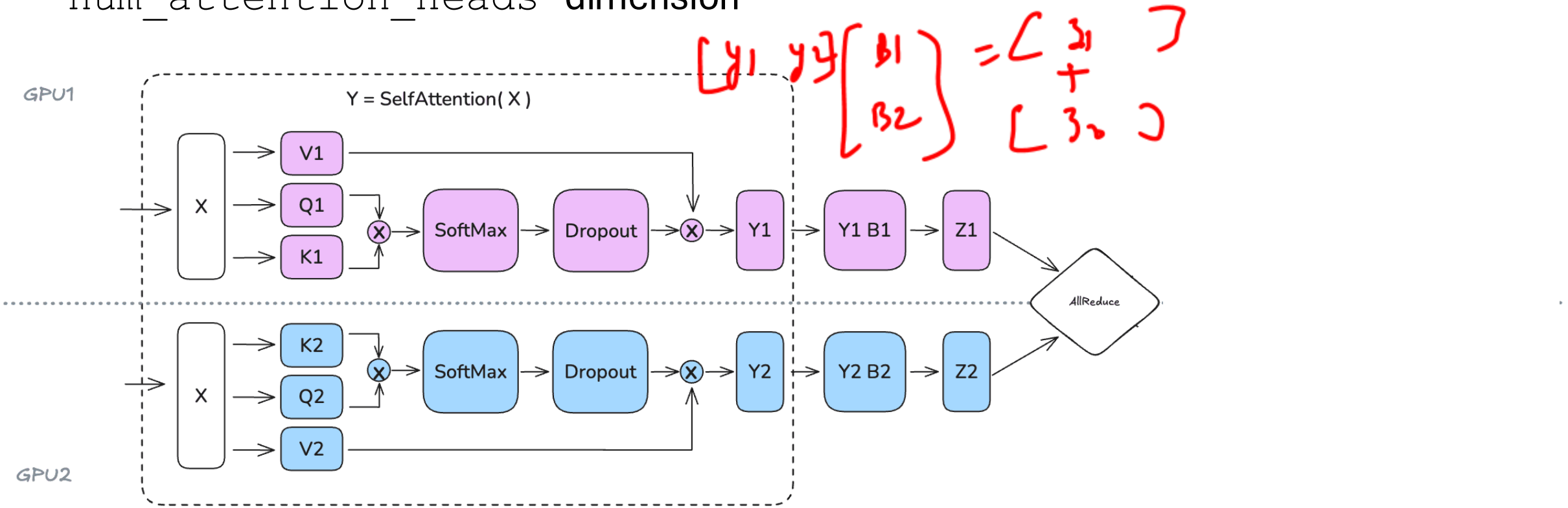
Tensor Parallelism in MHA Block

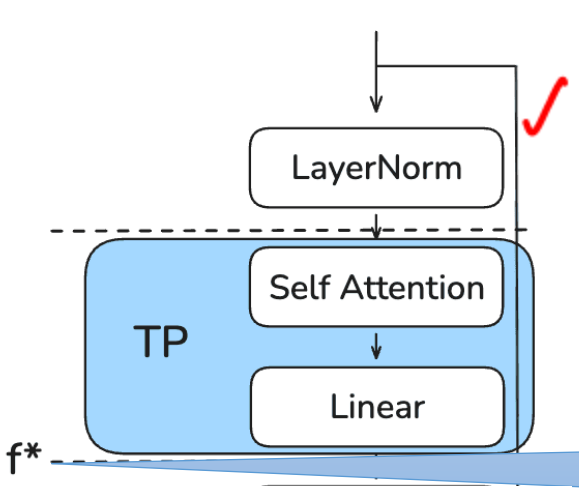
- Parallelize different heads on different GPUs – i.e. along `num_attention_heads` dimension
- Same as splitting *K*, *Q*, *V* matrices in column-parallel



Tensor Parallelism in Attention Block

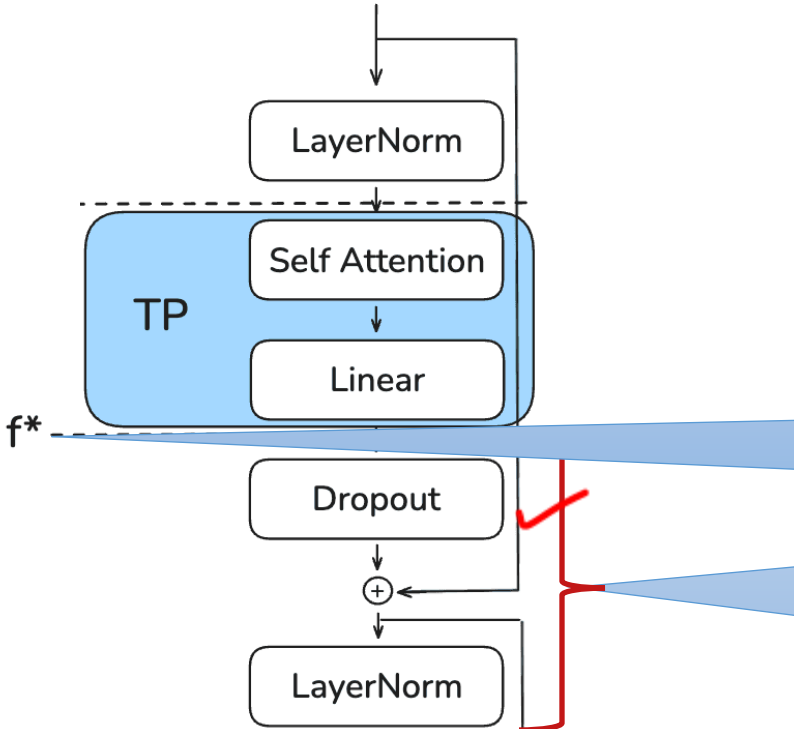
- Parallelize different heads on different GPUs – i.e. along `num_attention_heads` dimension
- Same as splitting *K*, *Q*, *V* matrices in column-parallel





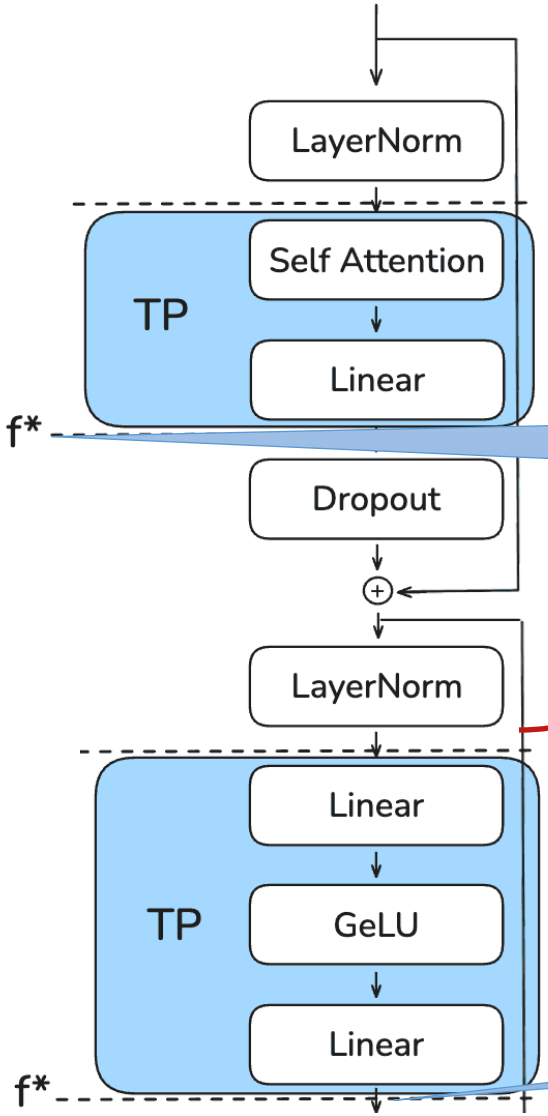
*f** : all-reduce to synchronize activations

1. Synchronization not overlapping with computation
2. *"Exposed communication" overhead* is necessary to combine partial results across tensor-parallel ranks before the final LayerNorm can be applied.



*f** : all-reduce to synchronize activations

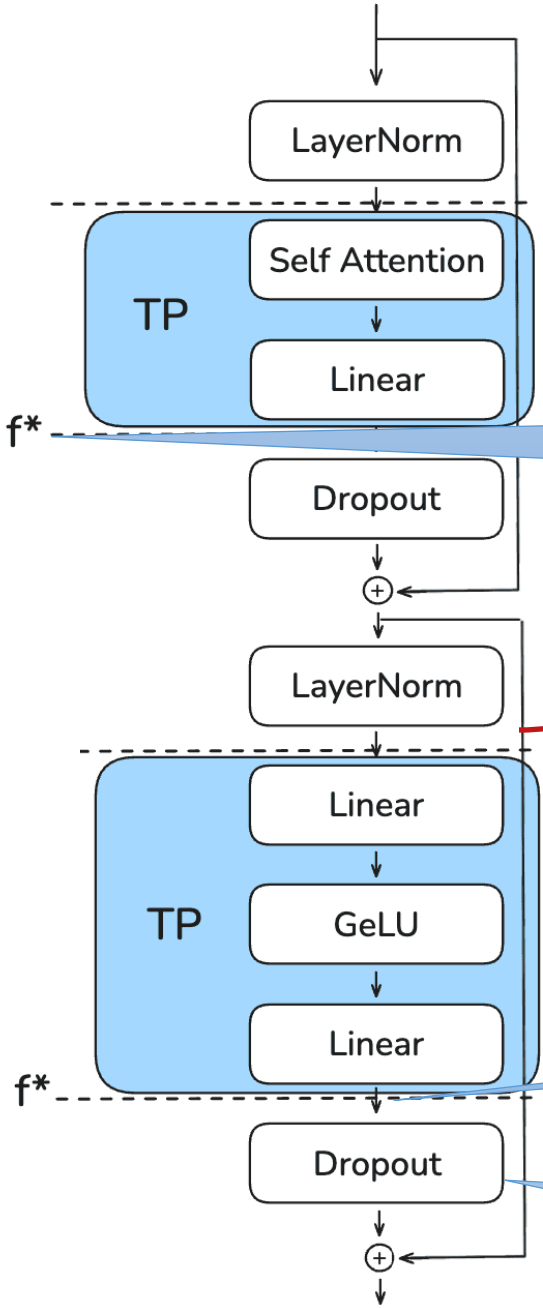
Dropout & LayerNorm: exactly same operations replicated on exactly same data



*f** : all-reduce to synchronize activations

Dropout & LayerNorm: exactly same operations replicated on exactly same data

*f** : all-reduce to synchronize activations

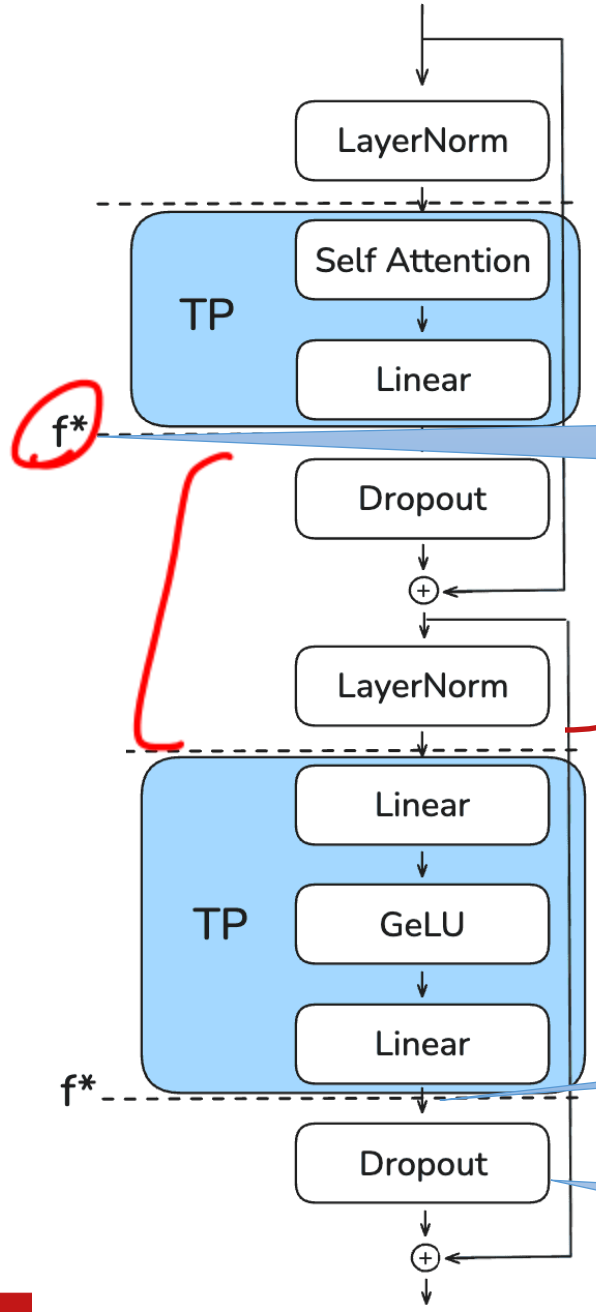


*f** : all-reduce to synchronize activations

Dropout & LayerNorm: exactly same operations replicated on exactly same data

*f** : all-reduce to synchronize activations

Dropout: same operation on same data!



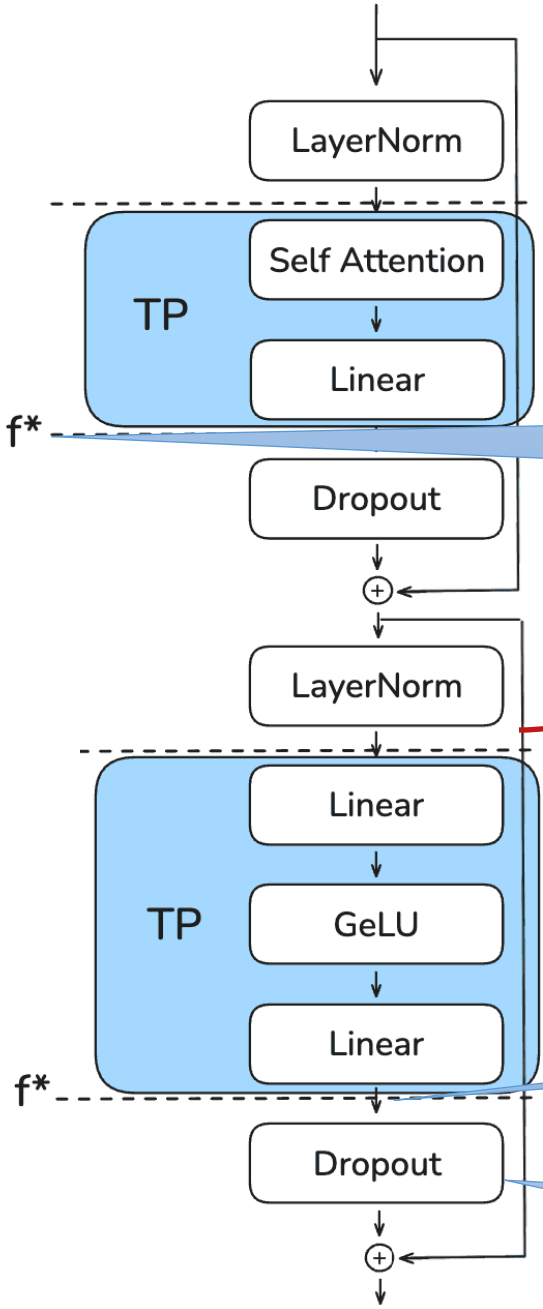
Dropout and LayerNorm – doing same operation on same data on all TP GPUs!

*f** : all-reduce to synchronize activations

Dropout & LayerNorm: exactly same operations replicated on exactly same data

*f** : all-reduce to synchronize activations

Dropout: same operation on same data!



Can we parallelize dropout and LayerNorm?

*f** : all-reduce to synchronize activations

Dropout & LayerNorm: exactly same operations replicated on exactly same data

*f** : all-reduce to synchronize activations

Dropout: same operation on same data!

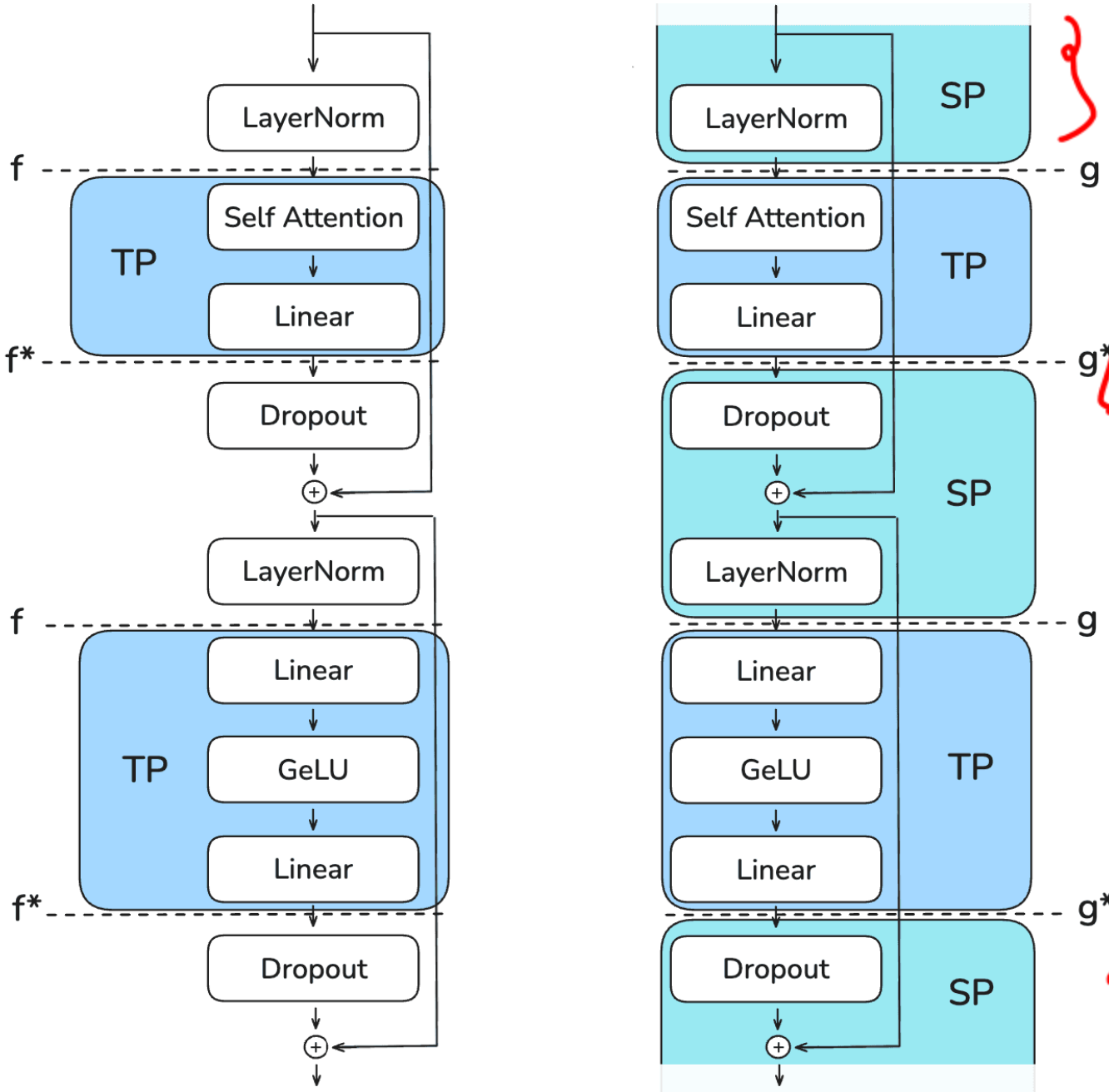
Sequence Parallel – parallelizing dropout & LayerNorm

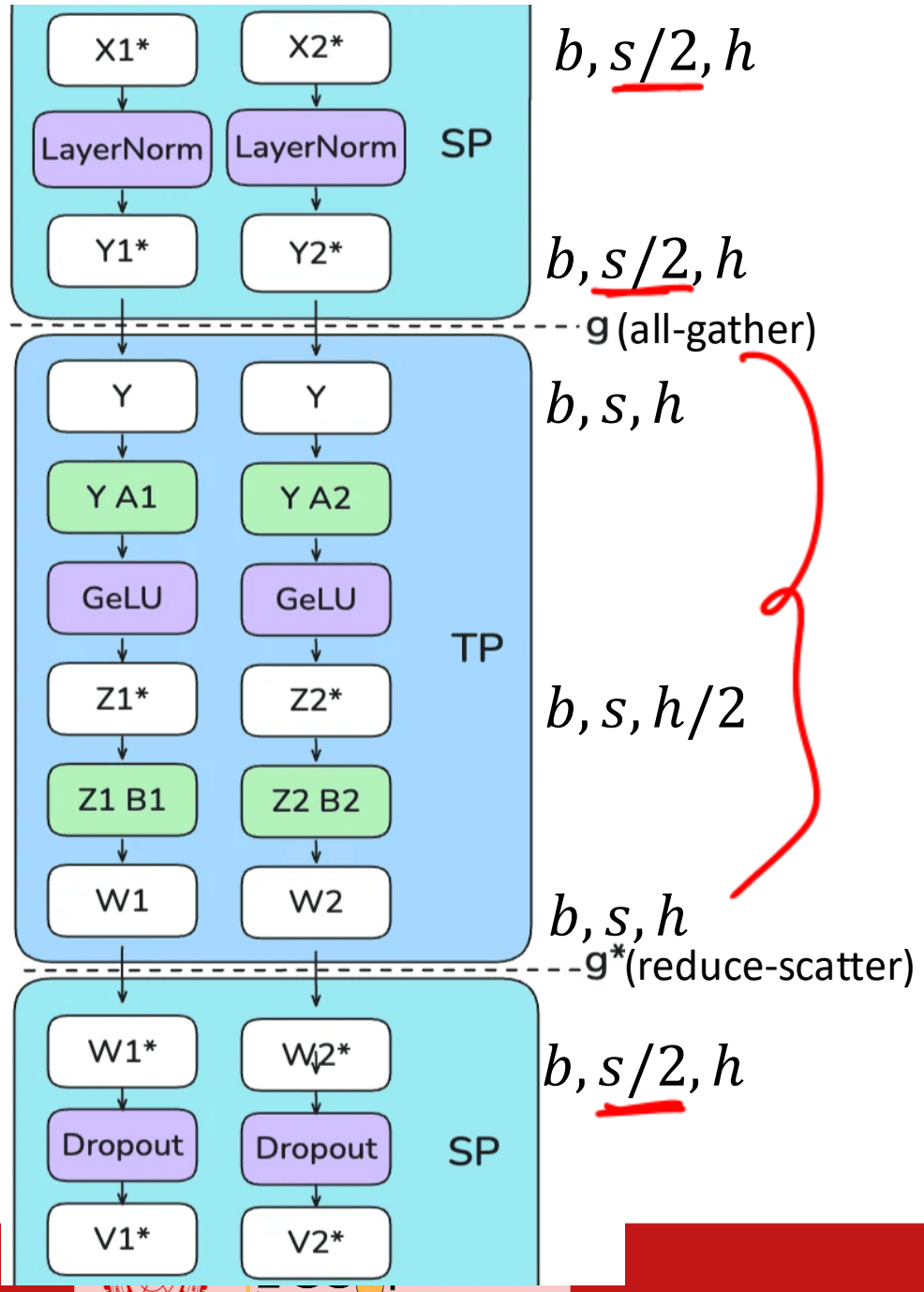
Total	$\mathbf{34 * seq * bs * h + 5 * n_{heads} * seq^2 * bs}$
MLP Block	$\mathbf{19 * seq * bs * h}$
D/o mask	$1 * seq * bs * h$
Linear *(4h -> h)*	$8 * seq * bs * h$
GELU	$8 * seq * bs * h$
Linear *(h -> 4h)*	$2 * seq * bs * h$
Layer Norm	$\mathbf{2 * seq * bs * h}$

Attention Block	$\mathbf{11 * seq * bs * h + 5 * n_{heads} * seq^2 * bs}$

Layer Norm	$\mathbf{2 * seq * bs * h}$

- In DP, we parallelize along the "batch dim" ($\boldsymbol{bs}$)
- In TP, we parallelize along the "hidden dim" ($\boldsymbol{h}$)
- In SP, we parallelize along the input sequence dimension ($\boldsymbol{seq}$)


LayerNorm
f
Self Attention
TP
Linear
f*
Dropout
LayerNorm
f
Linear
TP
GeLU
Linear
f*
Dropout
SP
LayerNorm
g
Self Attention
TP
Linear
g*
Dropout
SP
LayerNorm
g
Linear
TP
GeLU
Linear
g*
Dropout
SP
(all-gather)
(reduce-scatter)
(all-gather)
(reduce-scatter)



Initial LayerNorm layer (SP region)

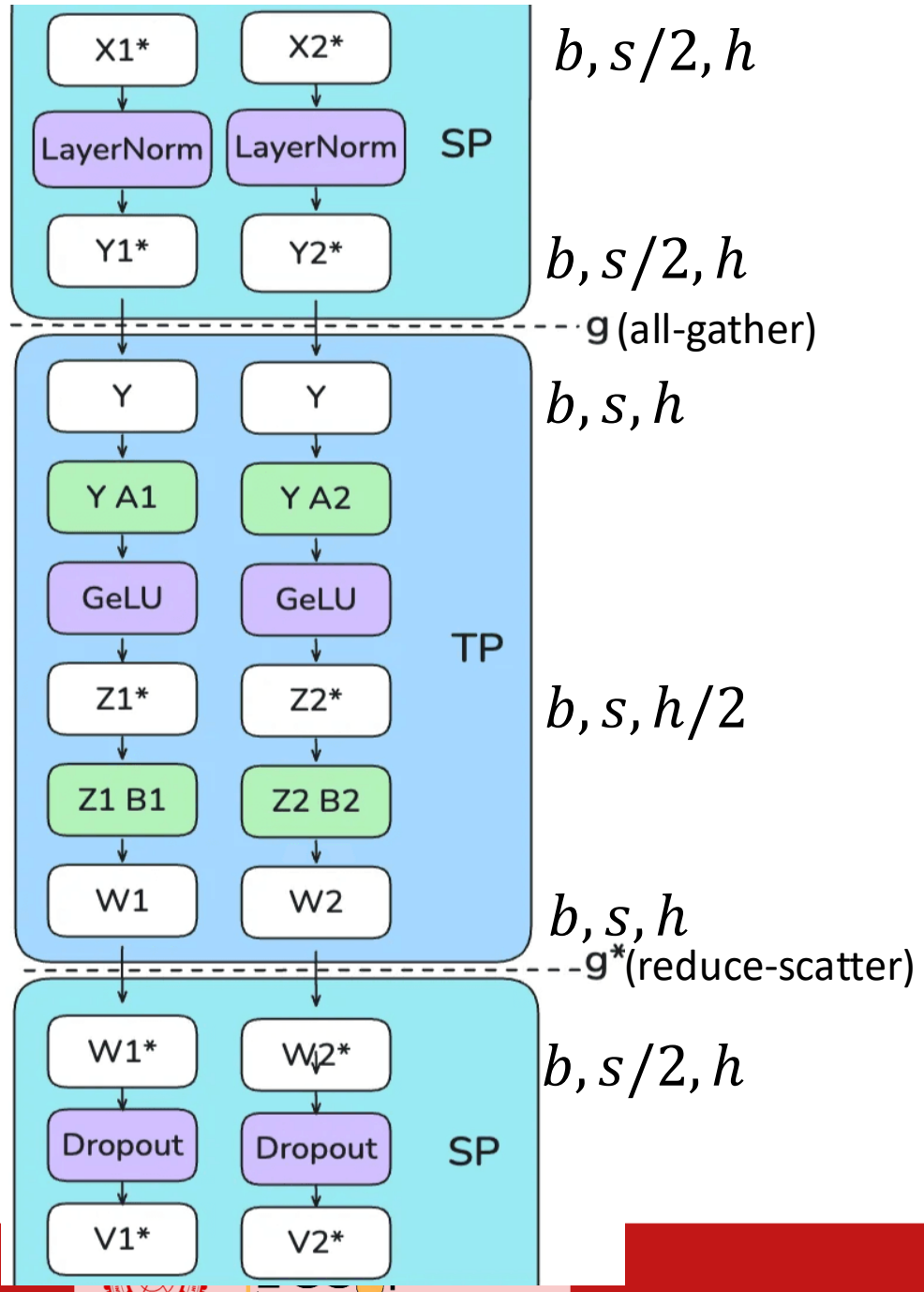
- Input tensors *X1** and *X2** $(b, s, /2h)$ enter, already split across the sequence dimension.
- Each GPU computes LayerNorm independently on its sequence chunk, giving *Y1** and *Y2**.

First transition (SP → TP)

- g operation (all-gather) combines *Y1* and *Y2* back to full sequence length.
- Restores *Y* (b, s, h) since column-linear layers need the full hidden dimension h.

First linear layer (TP region)

- *A1* and *A2* are column-linear layers, so they split *Y* along the hidden dimension.
- GELU is applied independently on each GPU.
- *Z1** and *Z2** are $(b, s, h/2)$.



$b, s/2, h$

$b, s/2, h$

b, s, h

$b, s, h/2$

b, s, h

$b, s/2, h$

Initial LayerNorm layer (SP region)

- Input tensors *X1** and *X2** $(b, s, /2h)$ enter, already split across the sequence dimension.
- Each GPU computes LayerNorm independently on its sequence chunk, giving *Y1** and *Y2**.

First transition (SP → TP)

- g operation (all-gather) combines *Y1* and *Y2* back to full sequence length.
- Restores *Y* (b, s, h) since column-linear layers need the full hidden dimension h.

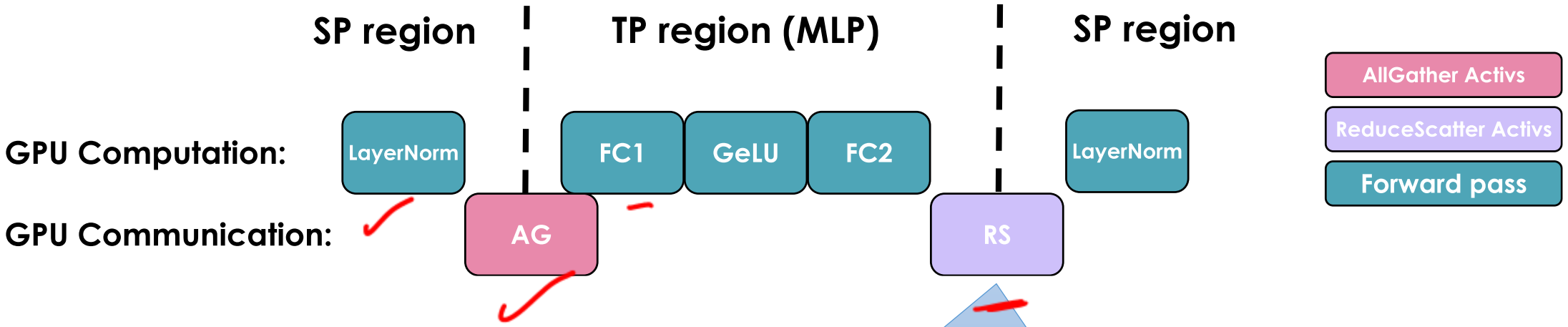
Second linear layer (TP region)

- *B1* and *B2* are row-linear layers, so they restore the hidden dimension.
- *W1* and *W2* are (b, s, h) that need to be summed together.

Final transition (TP → SP)

- g^* operation (reduce-scatter) reduces for previous row-linear correctness while scattering along the sequence dimension.
- *W1** and *W2** are $(b, s, /2h)$

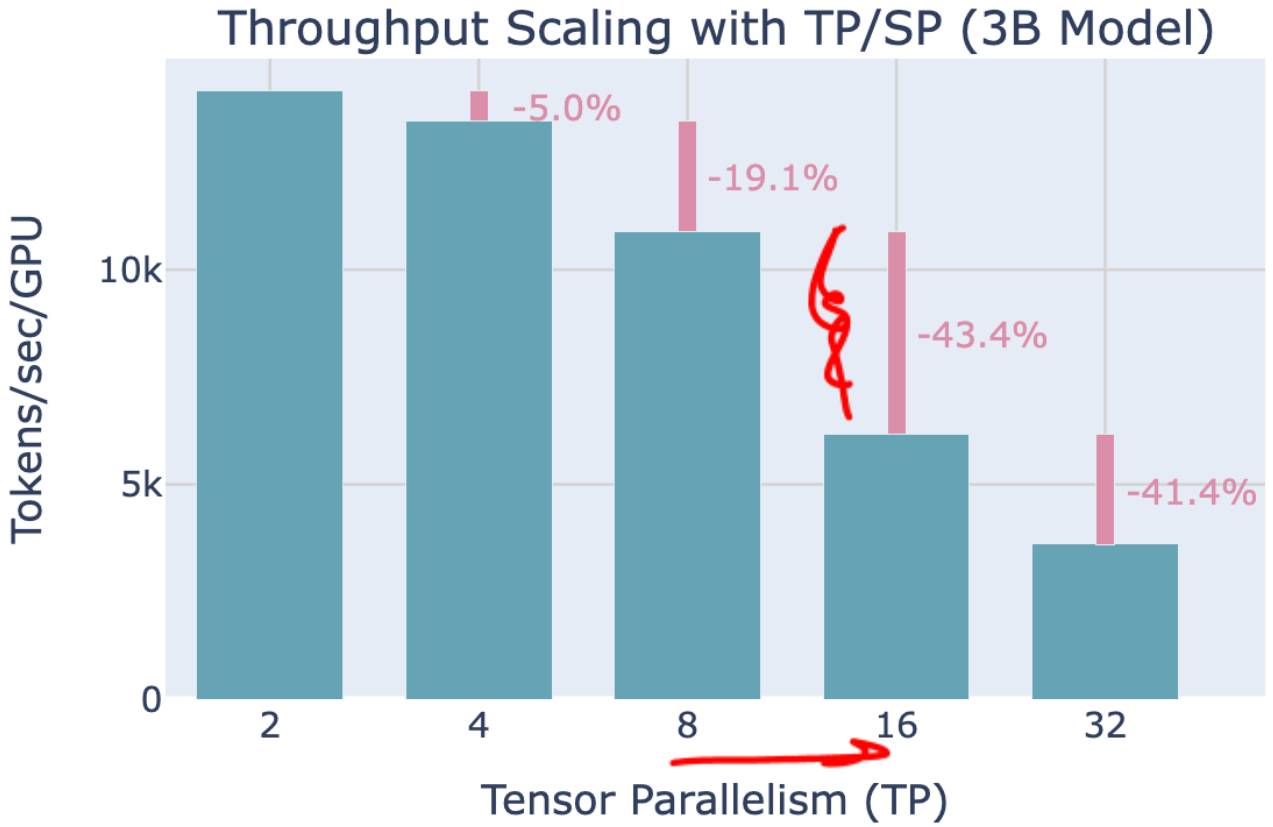
Computation-communication timeline for MLP Layer



1. Synchronization not overlapping with computation
2. *"Exposed communication" overhead* is necessary to combine partial results across tensor-parallel ranks before the LayerNorm can be applied.

Tensor+Sequence Parallelism - Tradeoffs

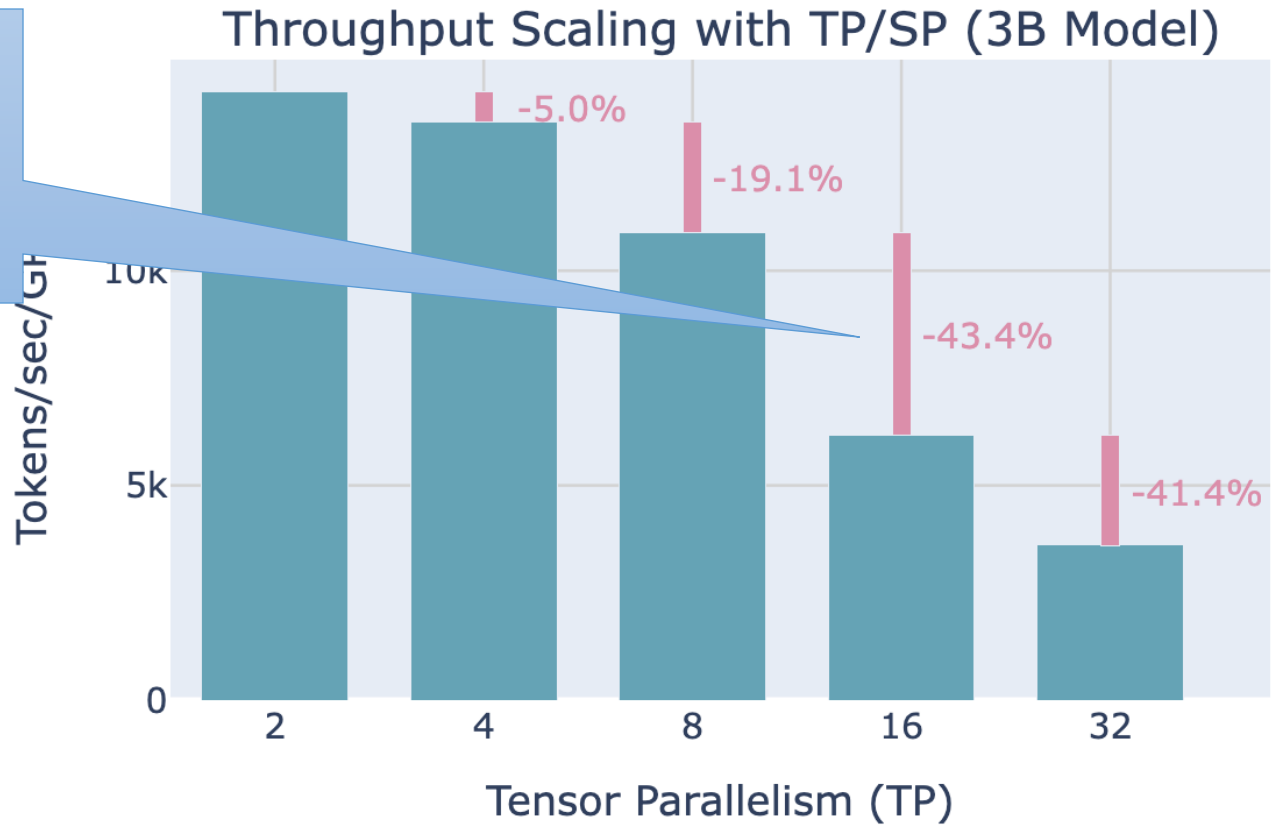
- Intermediate activations sharded across GPUs.
- **TP**: Reduces activation memory for matrix multiplication
- **SP**: Reduces activation memory for LayerNorm & dropout
- Need to gather full activations for LayerNorm
- Introduces significant communication overhead
- Introduces "*exposed communication*"



For Sequence Length 4096

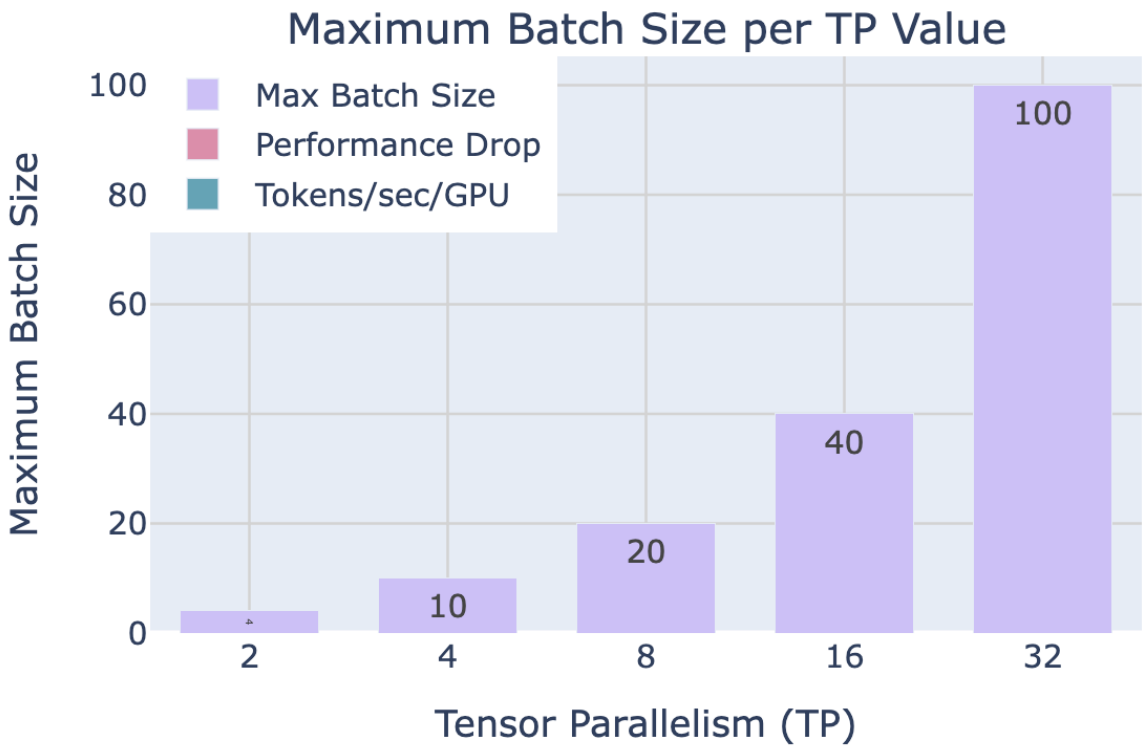
Tensor+Sequence Parallelism - Tradeoffs

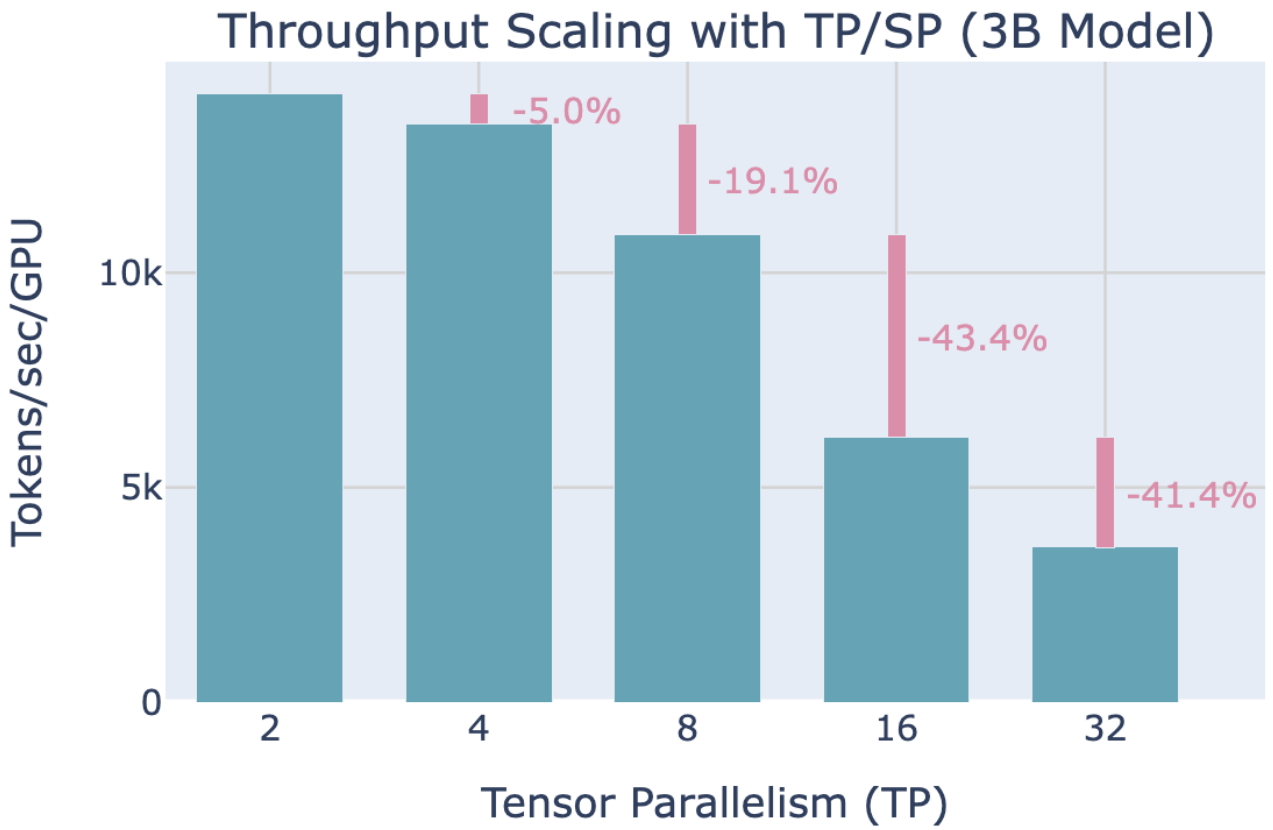
- TP leverages fast NVLink interconnects within a node.
- Slower network connections across nodes results in huge throughput drop.



For Sequence Length 4096

Tensor+Sequence Parallelism - Tradeoffs



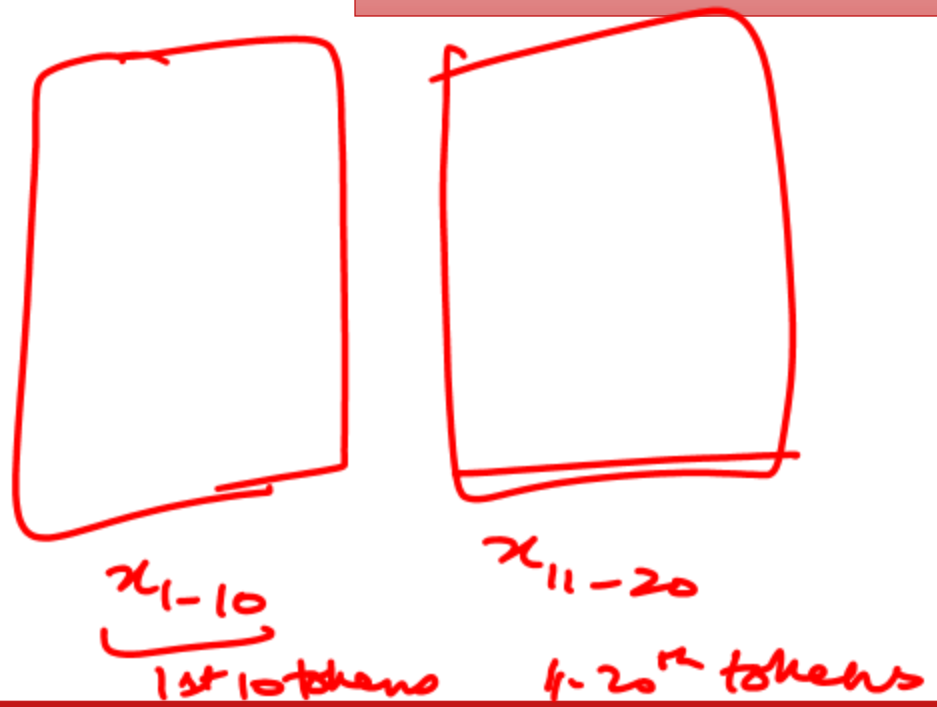


For Sequence Length 4096

Tensor+Sequence Parallelism - Limitations

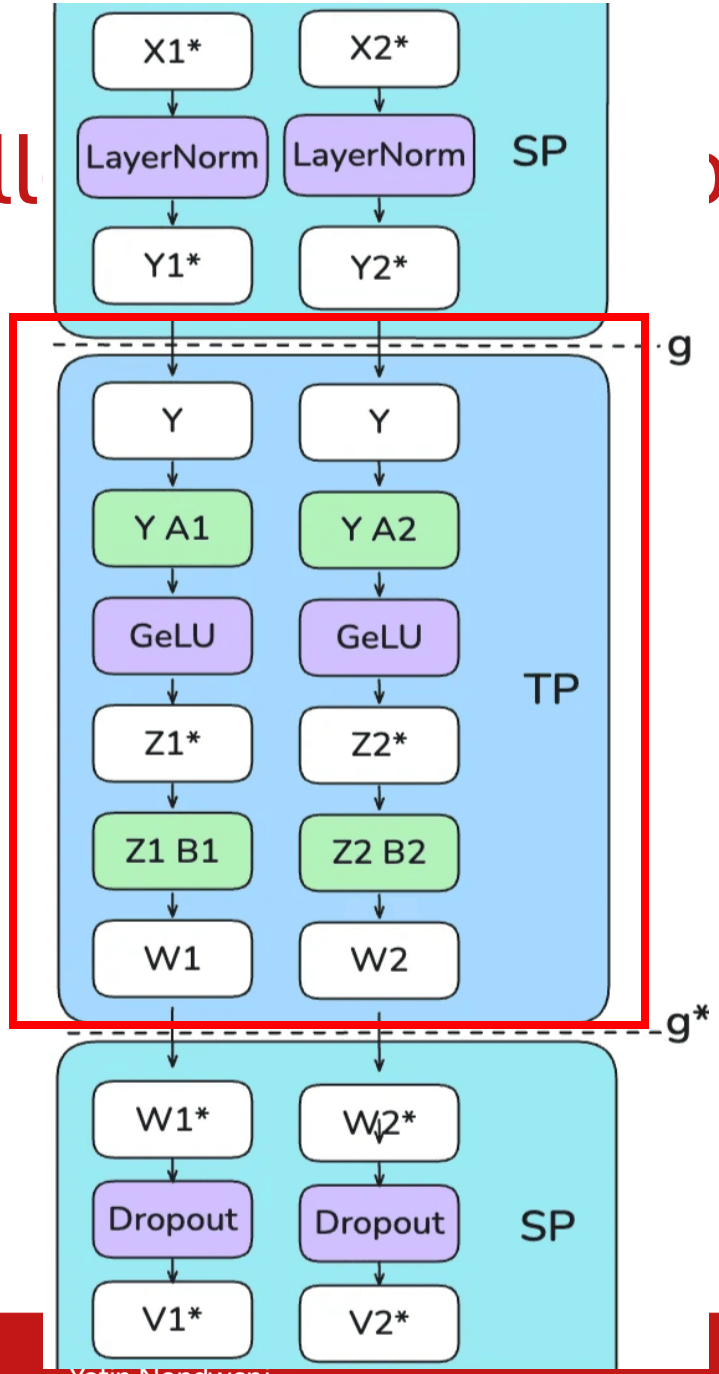
1. If we scale the sequence length the activation memory will still blow up in the TP region

- In SP, we split one sequence into chunks and process each chunk in parallel.
- Can we do the same for MLP Layer?
- How about Attention Layer?



Tensor+Sequence Parall ons

1. If we scale the sequence length the activation memory will still blow up in the TP region



Context Parallelism

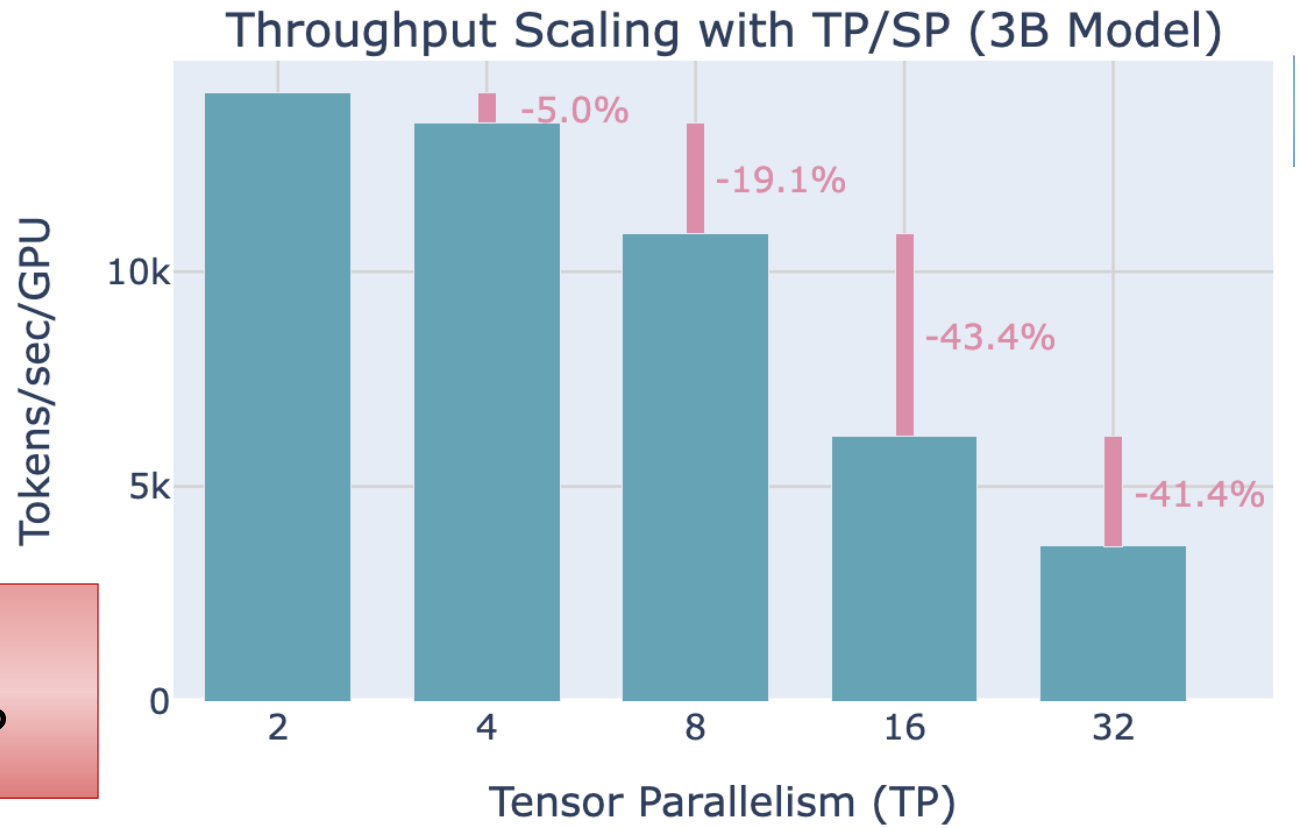
Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

2. If the model is too big to fit with TP=8 we will see a massive slowdown due to the inter-node connectivity.

- In TP, we split a “Tensor” across GPUs.
- How about splitting layers across GPUs?

Pipeline Parallelism



Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

Context Parallelism

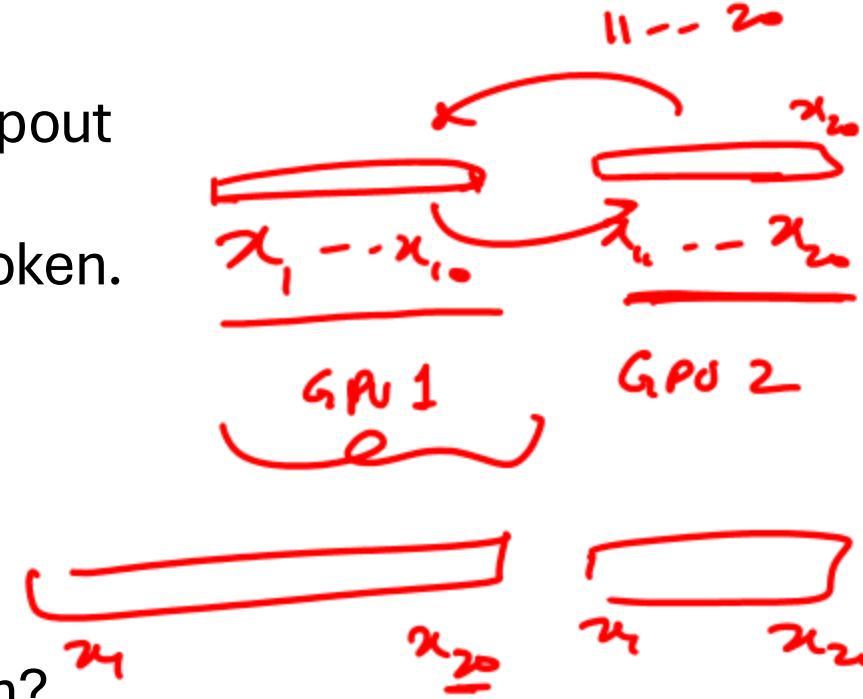
2. If the model is too big to fit with TP=8 we will see a massive slowdown due to the inter-node connectivity.

Pipeline Parallelism

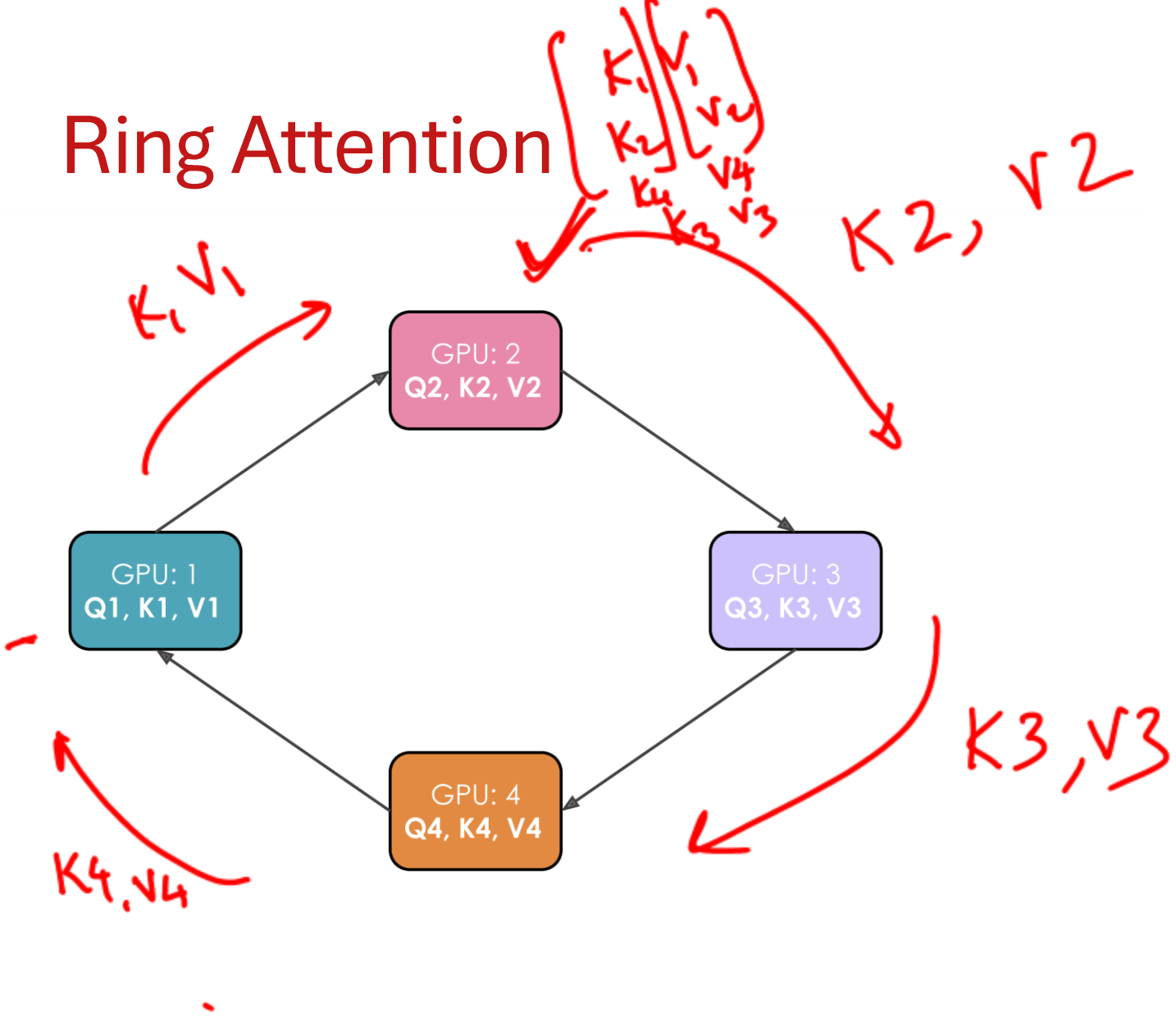
Context Parallelism – partition along sequence length

- For MLP layers, its exactly same as SP for LayerNorm & Dropout
- In Attention Layer, each token has to attend on every other token.
- But tokens in a different chunk are on a different GPU!
- Full communication b/w GPUs?
- Can we somehow overlap computation with communication?

Ring Attention - online softmax computation + overlapped computation

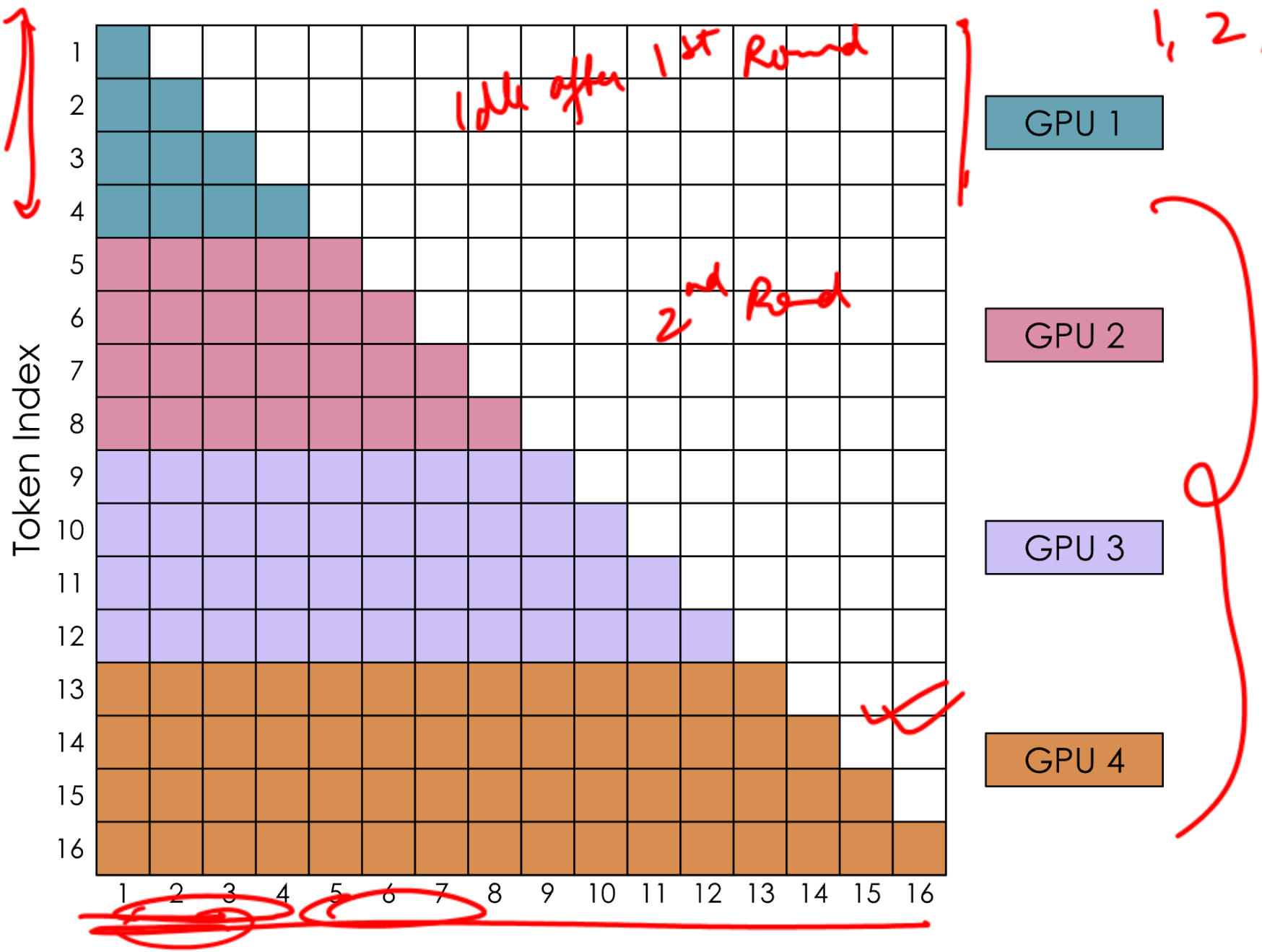


Ring Attention

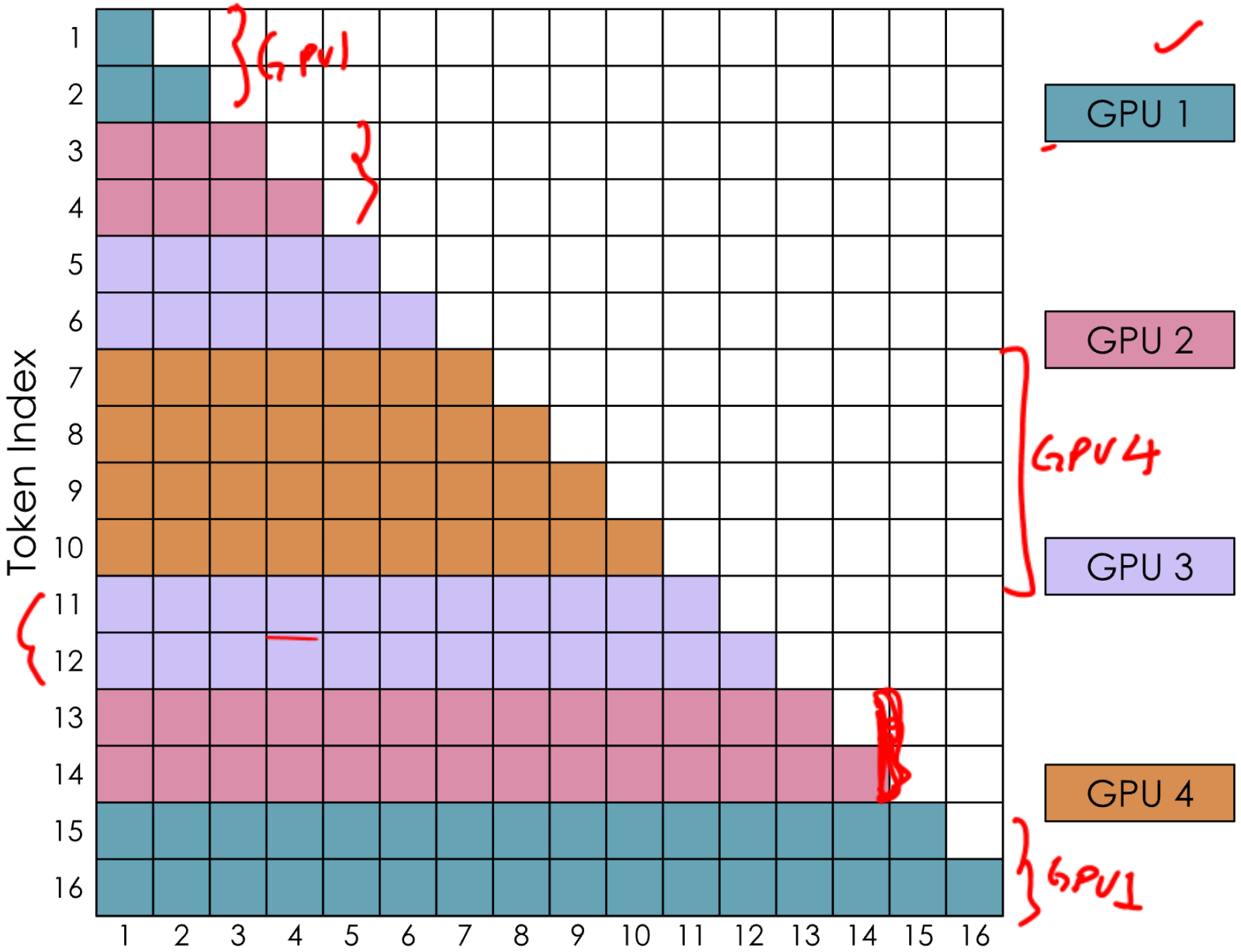


Ring Attention with Blockwise Transformers for Near-Infinite Context H. Liu, M. Zaharia, P. Abbeel. 2023 [PDF]

Ring Attention



Ring Attention

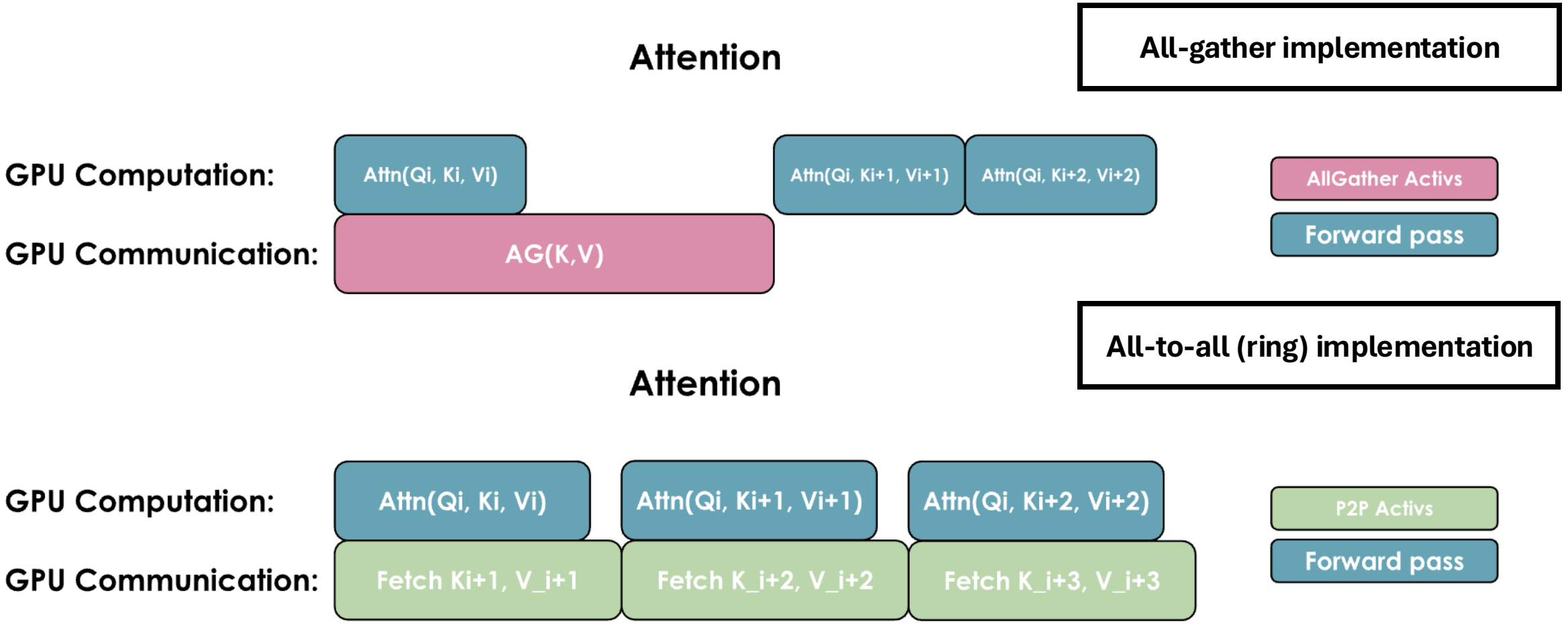


Computation-communication timeline

All-to-all (ring) implementation:

- GPUs exchange K/V pairs in a ring-like pattern, one chunk at a time.
- More memory-efficient, as each GPU only needs to store one additional chunk temporarily.
- Communication is spread out and overlapped with computation, though with some additional base latency overhead from multiple communication steps.

Comparing with Naïve *all-gather* Implementation



Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

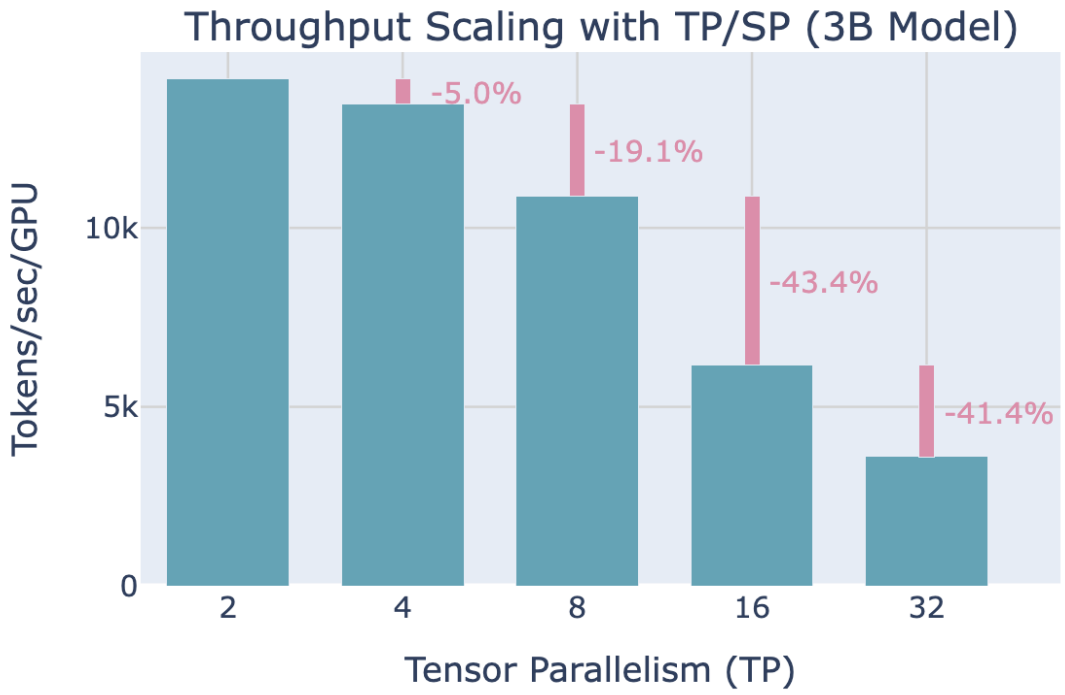
Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region



Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes
- How about splitting layers across GPUs?

Tensor+Sequence Parallelism - Limitations

1. If we scale the sequence length the activation memory will still blow up in the TP region

2. If the model is too big to fit with TP=8 we will see a massive slowdown due to the inter-node connectivity.

Pipeline Parallelism

Context Parallelism

- TP: Split a model across one node to tame large models
- CP: Tame the activation explosion with long sequences.

- TP: doesn't scale well across nodes
- How about splitting layers across GPUs?