

Lecture 23:

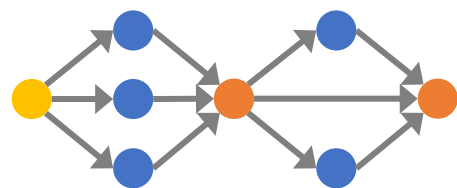
Parallel Deep Learning

(Model & Pipeline Parallelism)

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Fall 2026

Recap: Data Parallelism



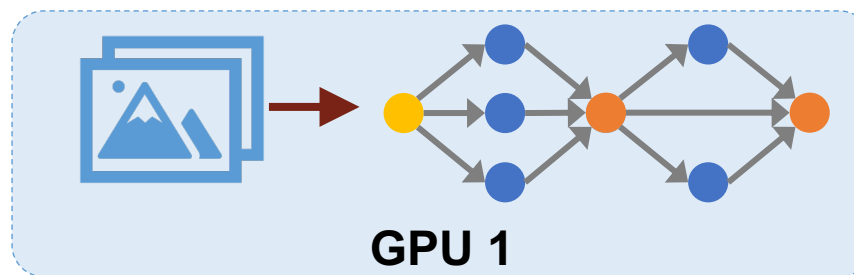
ML Model



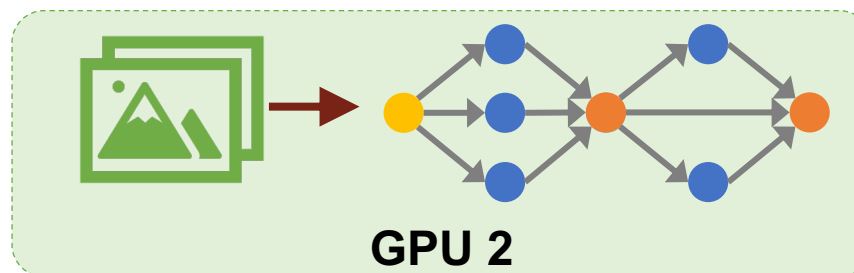
Training Dataset

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^n \nabla L_j(w_i)$$

1. Partition training data into batches

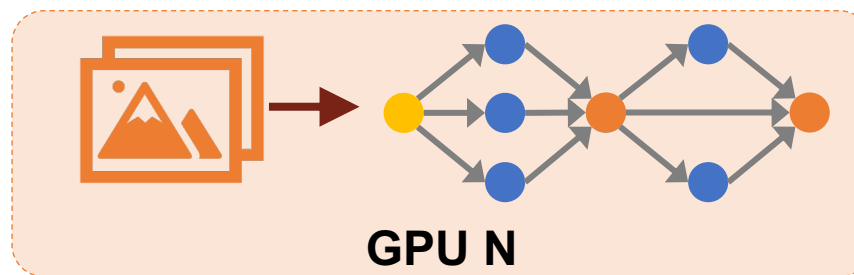


GPU 1



GPU 2

...



GPU N

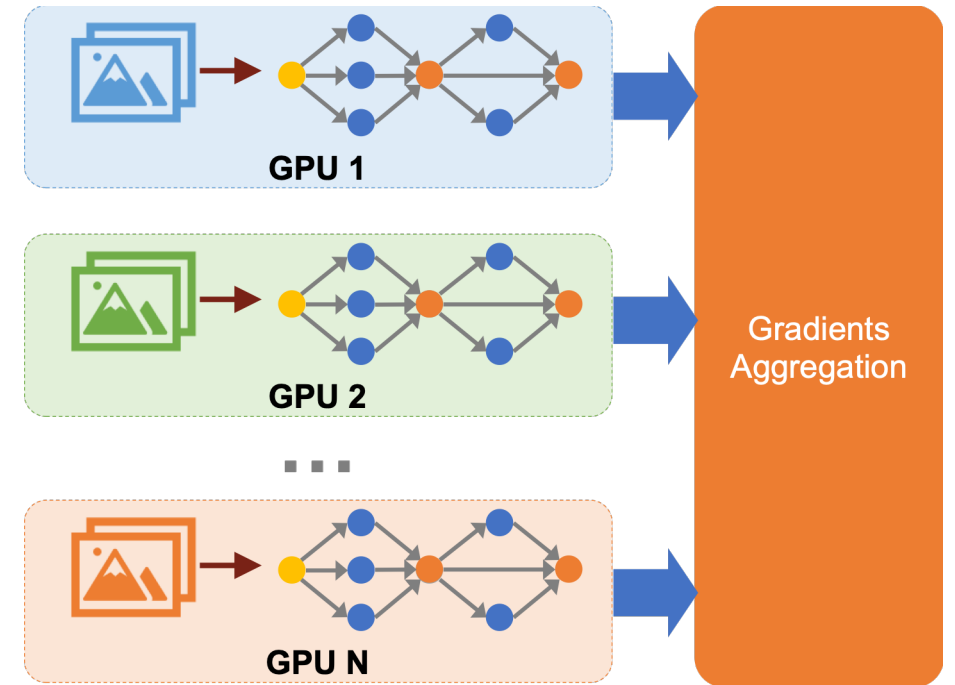
Gradients
Aggregation

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

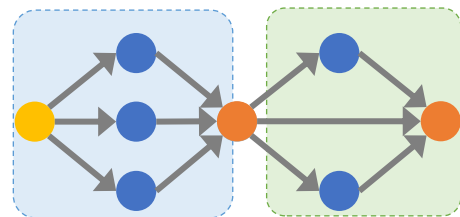
Recap: An Issue with Data Parallelism

- Each GPU saves a replica of the entire model
- Cannot train large models that exceed GPU device memory



Model Parallelism

- Split a model into multiple subgraphs and assign them to different devices

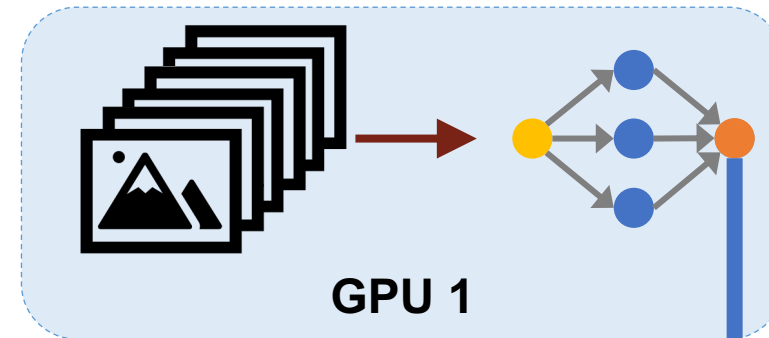


ML Model

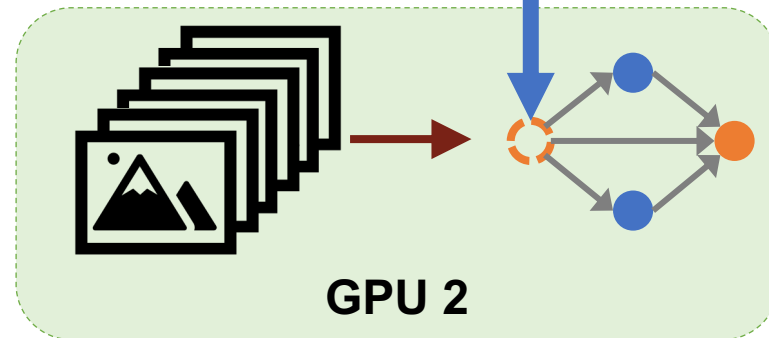


Training Dataset

**Model
Parallelism**



GPU 1



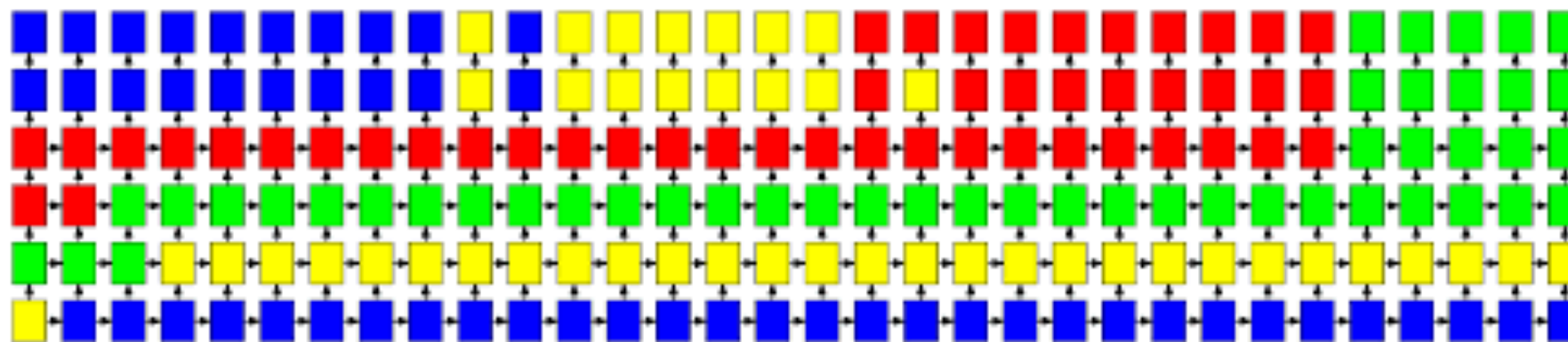
GPU 2

Transfer
intermediate
results
between
devices

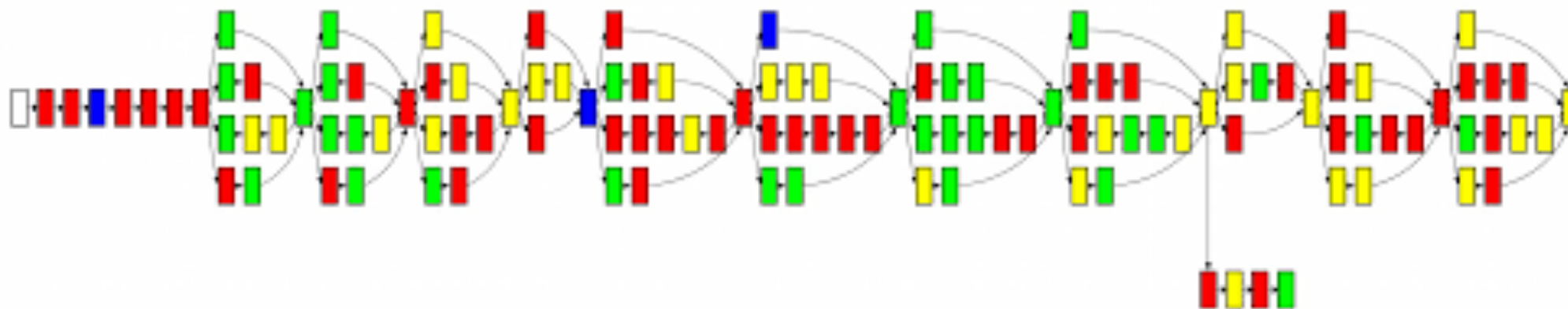
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^n \nabla L_j(w_i)$$

No synchronization of gradients

Device Placement for Model Parallelism is Challenging

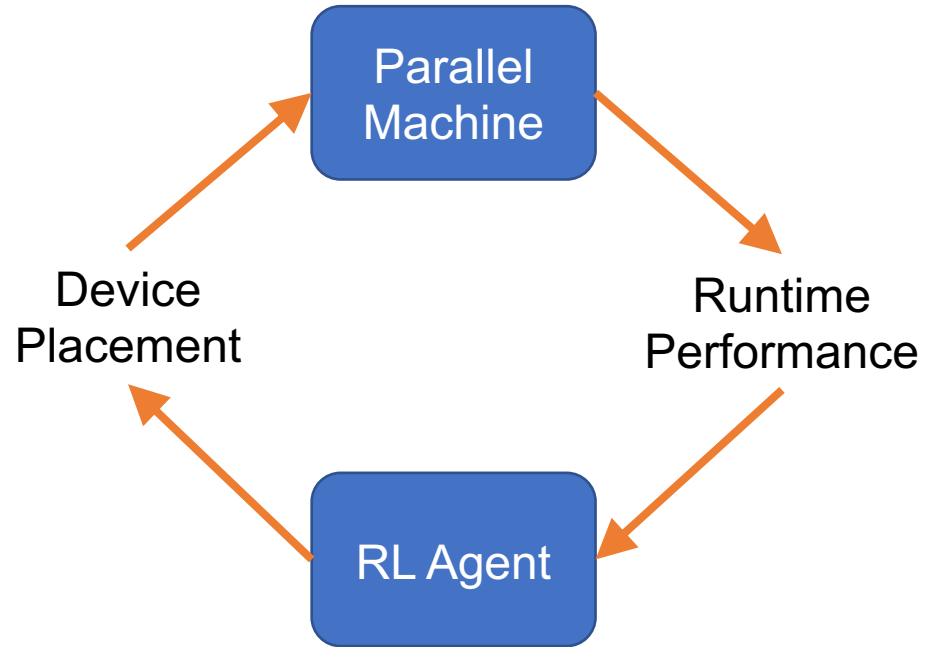
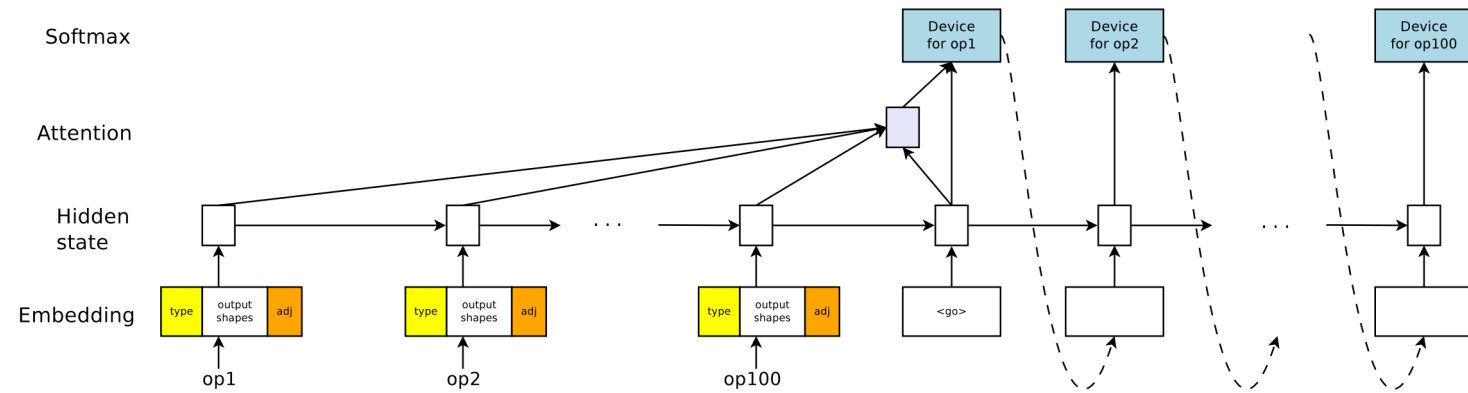


Model parallelism: training a recurrent neural network on 4 GPUs



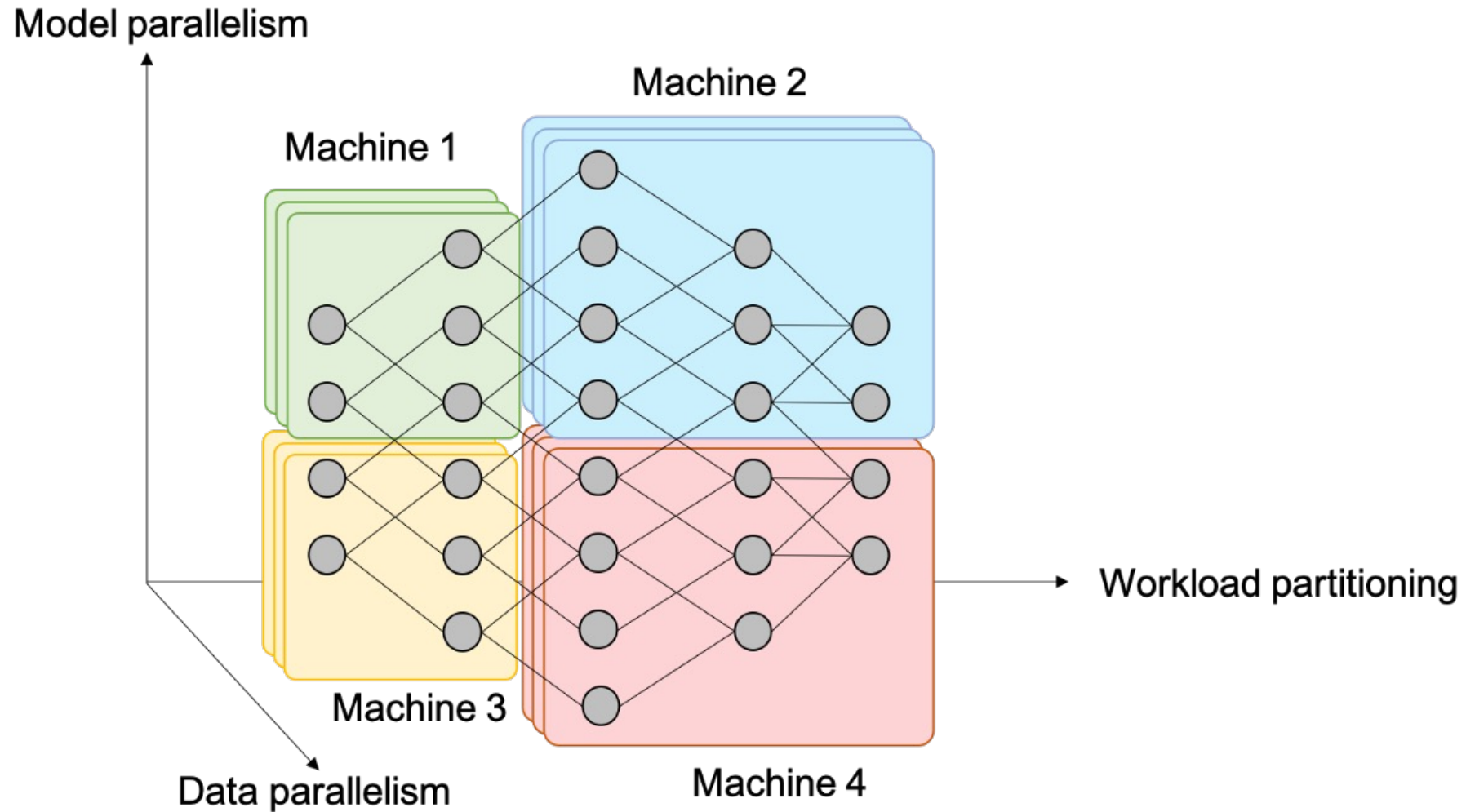
Model parallelism: training a conventional neural network on 4 GPUs

Using ML to Optimize Device Placement for ML



ColocRL's neural architecture

Combine Data and Model Parallelism



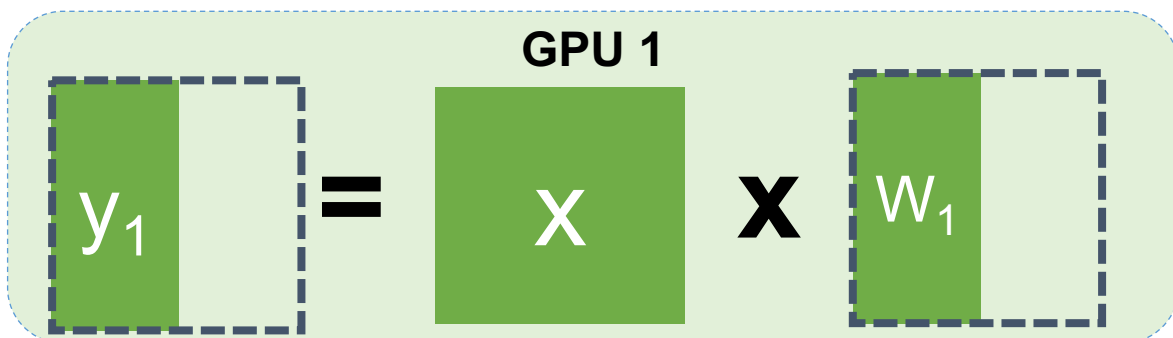
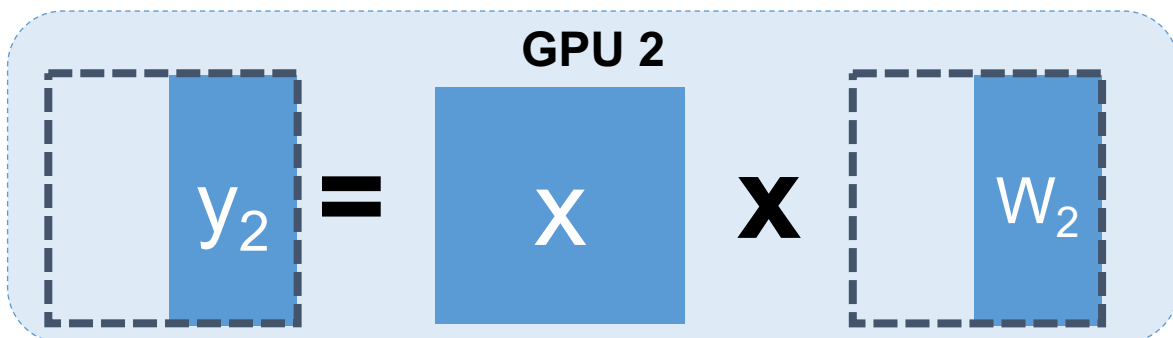
Tensor Model Parallelism

$$y = x \times W$$

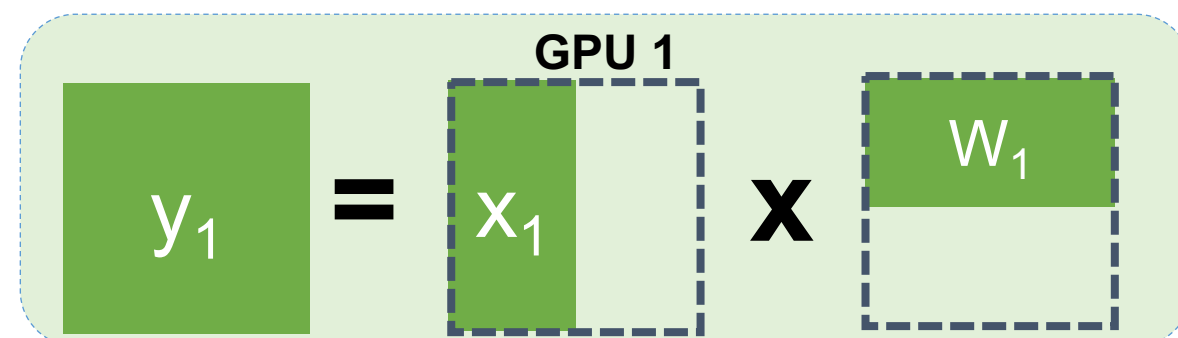
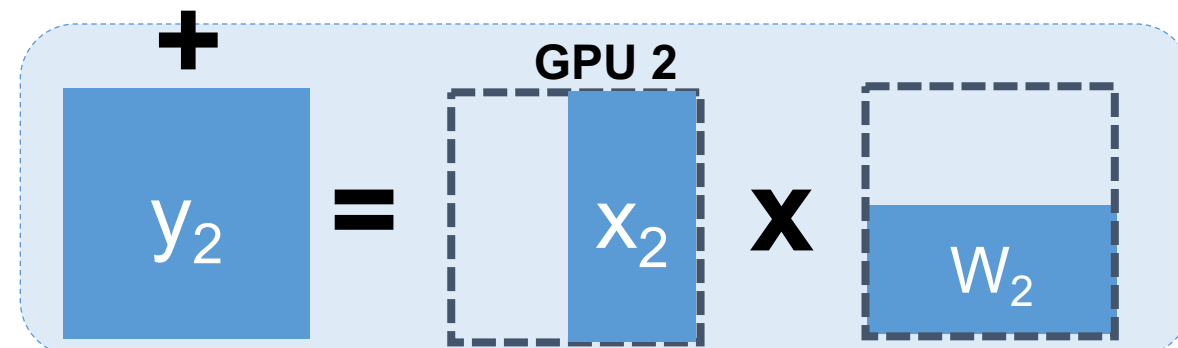
output input parameters

- Partition parameters/gradients *within* a layer/operator

Each GPU has the entire input



Tensor Model Parallelism (partition output)



Tensor Model Parallelism (reduce output)

$$y = y_1 + y_2$$

Comparing Data and Tensor Model Parallelism

$$\begin{matrix} B \\ \{ \end{matrix} \begin{matrix} \text{ } \\ y \end{matrix} = \begin{matrix} \text{ } \\ x \end{matrix} \times \begin{matrix} C_{out} \\ \{ \end{matrix} \begin{matrix} \text{ } \\ W \end{matrix} \begin{matrix} \text{ } \\ \} \end{matrix} \begin{matrix} C_{in} \end{matrix}$$

$$\begin{matrix} B \\ \{ \end{matrix} \begin{matrix} \text{ } \\ y_1 \end{matrix} = \begin{matrix} \text{ } \\ x_1 \end{matrix} \times \begin{matrix} C_{out} \\ \{ \end{matrix} \begin{matrix} \text{ } \\ W \end{matrix} \begin{matrix} \text{ } \\ \} \end{matrix} \begin{matrix} C_{in} \end{matrix}$$

GPU 1

$$\begin{matrix} \text{ } \\ y_2 \end{matrix} = \begin{matrix} \text{ } \\ x_2 \end{matrix} \times \begin{matrix} \text{ } \\ W \end{matrix}$$

GPU 2

Data parallelism

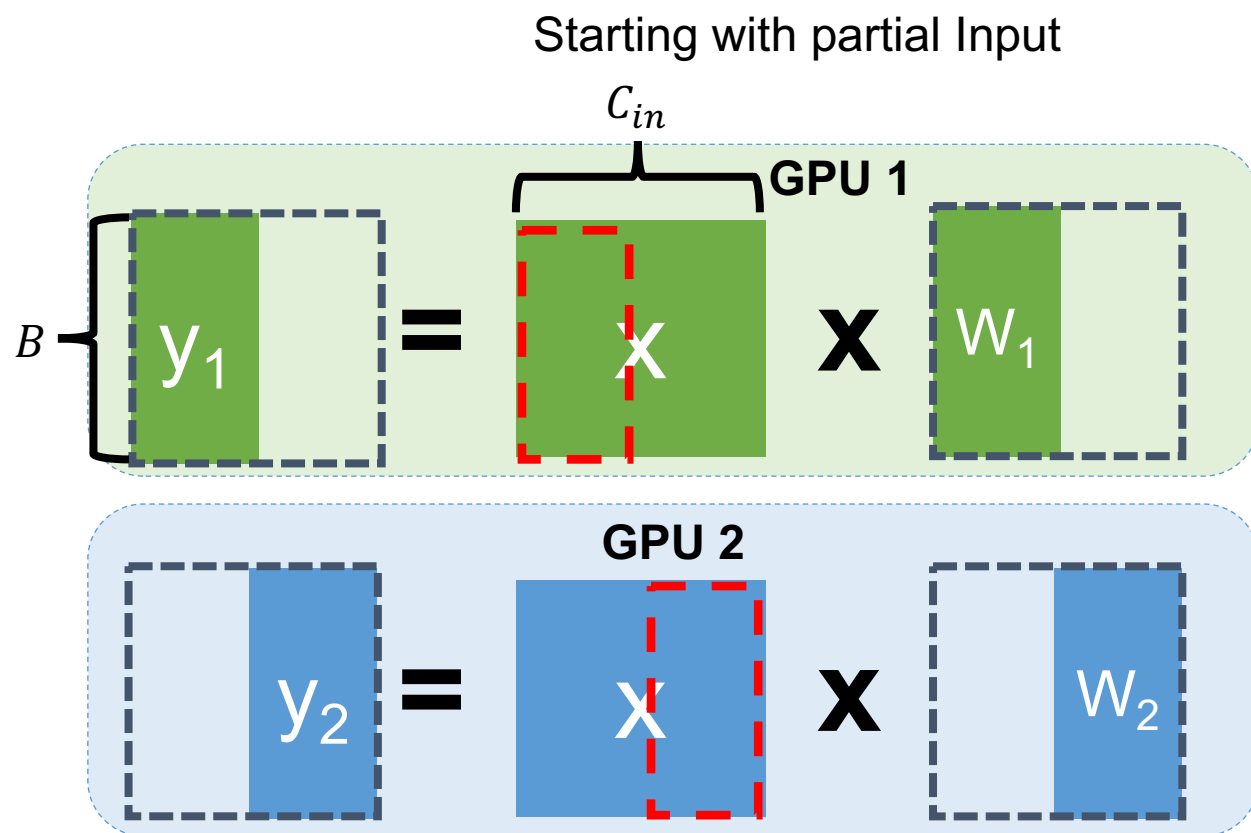
$$y = Wx$$

Forward Processing	Backward Propagation	Gradients Sync
0	0	$2 * C_{out} * C_{in}$

Communication Cost of Data Parallelism

Comparing Data and Tensor Model Parallelism

$$B \left\{ \begin{array}{c} \text{ } \\ y \end{array} \right\} = \begin{array}{c} \text{ } \\ x \end{array} \times \begin{array}{c} C_{out} \\ \left\{ \begin{array}{c} \text{ } \\ W \end{array} \right\} \\ C_{in} \end{array}$$



Tensor Model Parallelism (partition output)

Forward Processing	Backward Propagation	Gradients Sync
$B * C_{in}$	$B * C_{in}$	0

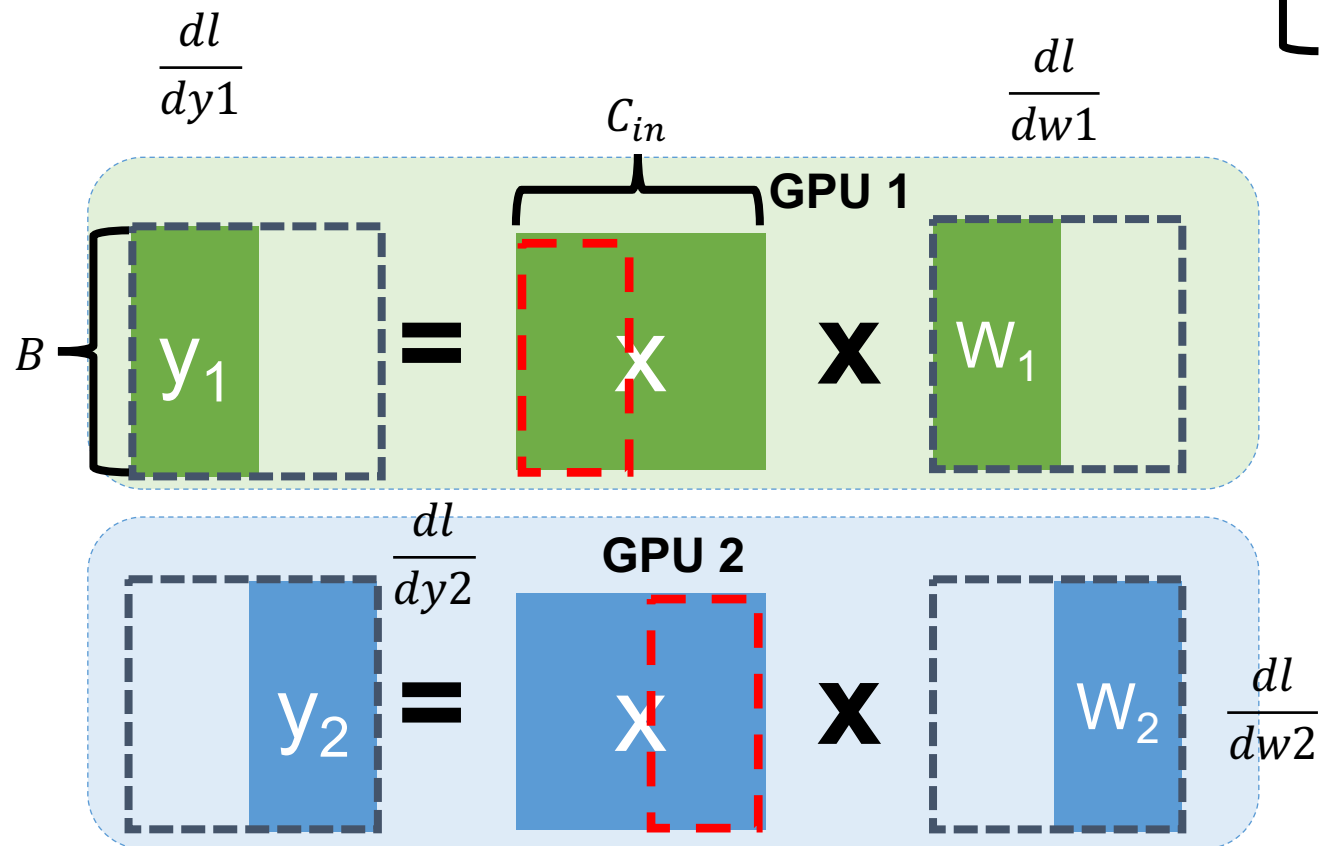
Communication Cost of Tensor Model Parallelism

*Assume each GPU has part of the input e.g., previous layer uses data parallelism

Comparing Data and Tensor Model Parallelism

$$\frac{dl}{dw1} = \mathbf{X} \frac{dl}{dly1}$$

$$B \left\{ \begin{array}{c} \text{y} \end{array} \right\} = \begin{array}{c} \text{x} \end{array} \times \begin{array}{c} \text{W} \end{array} \left\{ \begin{array}{c} C_{out} \\ C_{in} \end{array} \right\}$$



Tensor Model Parallelism (partition output)

$$\frac{dl}{dx} = \frac{dl}{dly1} + \frac{dl}{dly2}$$

Forward Processing	Backward Propagation	Gradients Sync
$B * C_{in}$	$B * C_{in}$	0

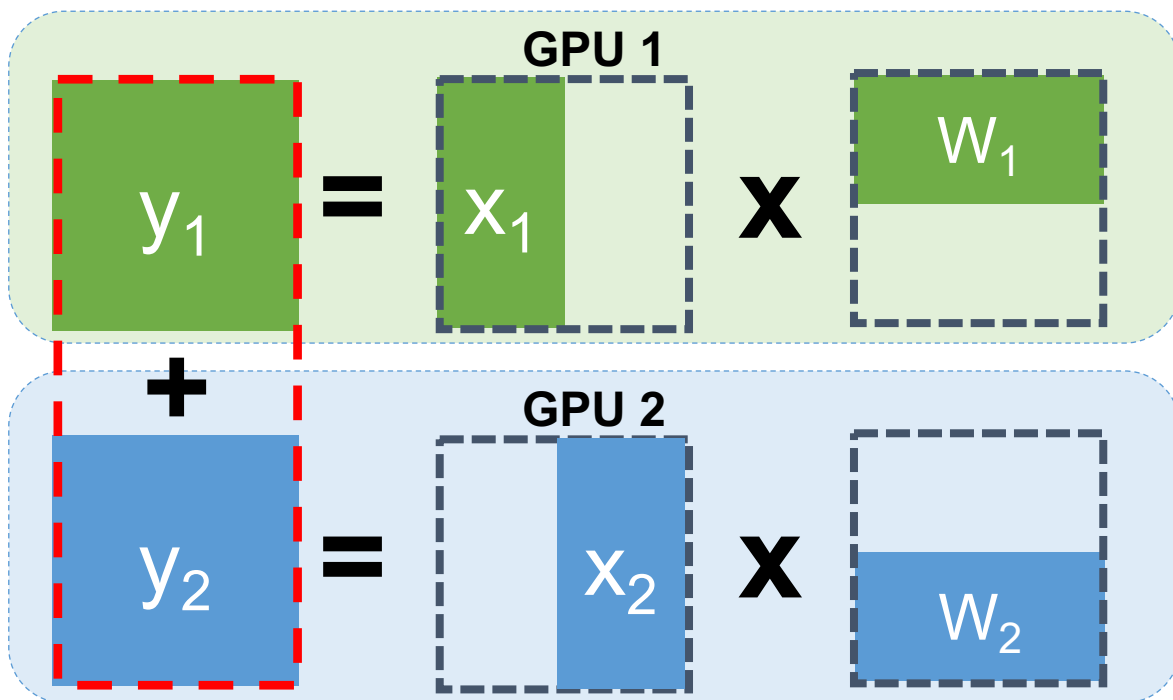
Communication Cost of Tensor Model Parallelism

*Assume each GPU has part of the input e.g., previous layer uses data parallelism

Comparing Data and Tensor Model Parallelism

$$\underbrace{B}_{\text{Batch Size}} \left[\begin{array}{c} \text{ } \\ y \\ \text{ } \end{array} \right] = \left[\begin{array}{c} \text{ } \\ x \\ \text{ } \end{array} \right] \times \left[\begin{array}{c} C_{out} \\ \text{ } \\ C_{in} \end{array} \right] W$$

Allreduce for y_1 and y_2



$$y = y_1 + y_2$$

Forward Processing	Backward Propagation	Gradients Sync
$2 * B * C_{out}$	0	0

Communication Cost of Tensor Model Parallelism

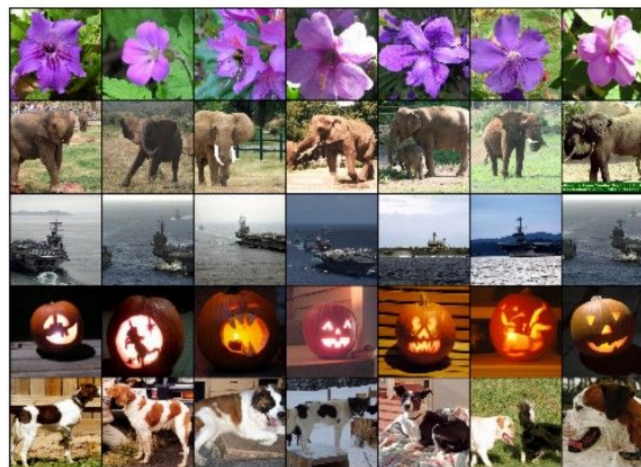
Comparing Data and Tensor Model Parallelism

- Data parallelism: $C_{out} * C_{in} \rightarrow$ Synchronize gradients
 - Tensor model parallelism (partition output): $B * C_{in}$
 - Tensor model parallelism (reduce output): $B * C_{out}$
- } $\rightarrow$ transfer activations*
- **The best strategy depends on the model and underlying machine**

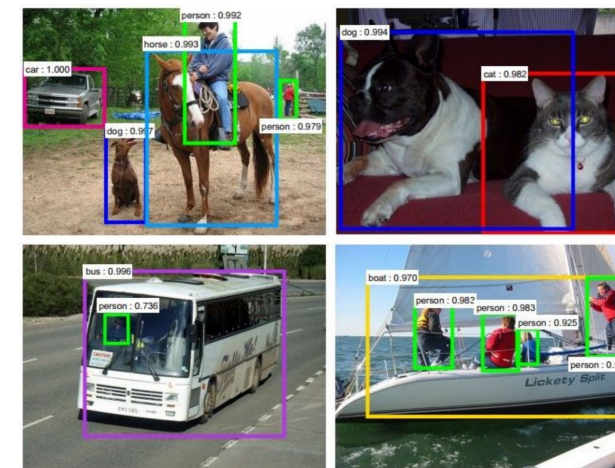
Example: Convolutional Neural Networks



Classification



Retrieval



Detection



Segmentation



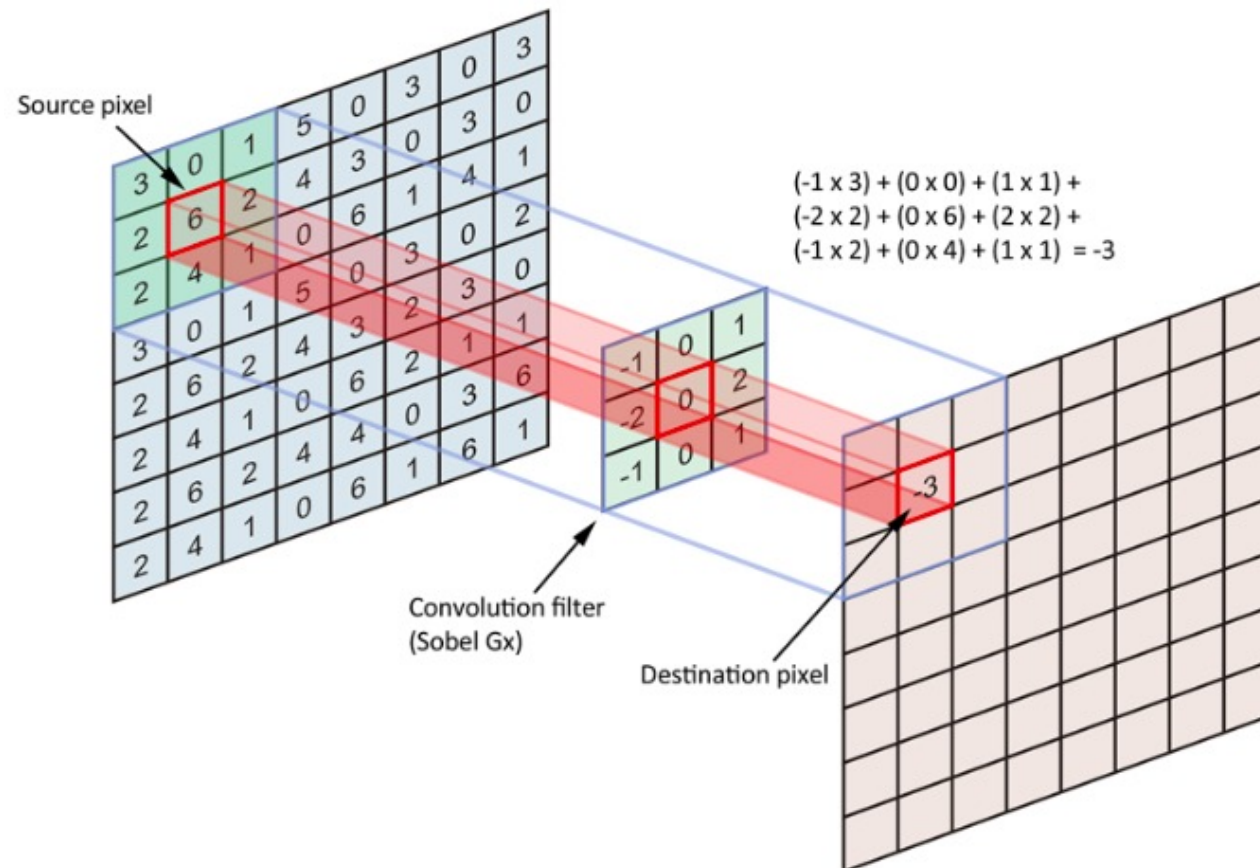
Self-Driving



Synthesis

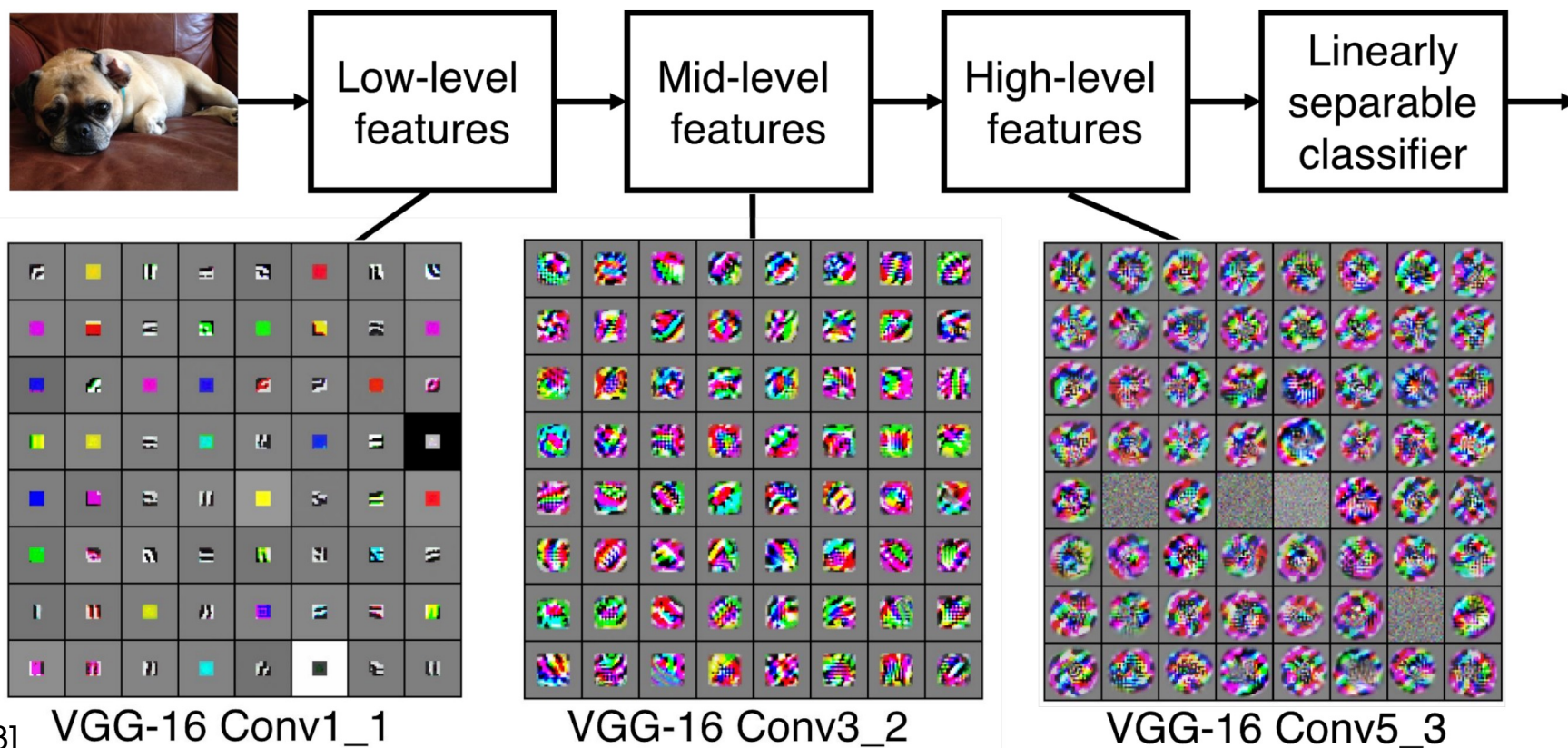
Convolution

- Convolve the filter with the image: slide over the image spatially and compute dot products



CNNs

- A sequence of convolutional layers, interspersed by pooling, normalization, and activation functions



Parallelizing Convolutional Neural Networks

- Convolutional layers
 - 90-95% of the computation
 - 5% of the parameters
 - Very large intermediate activations

Data parallelism

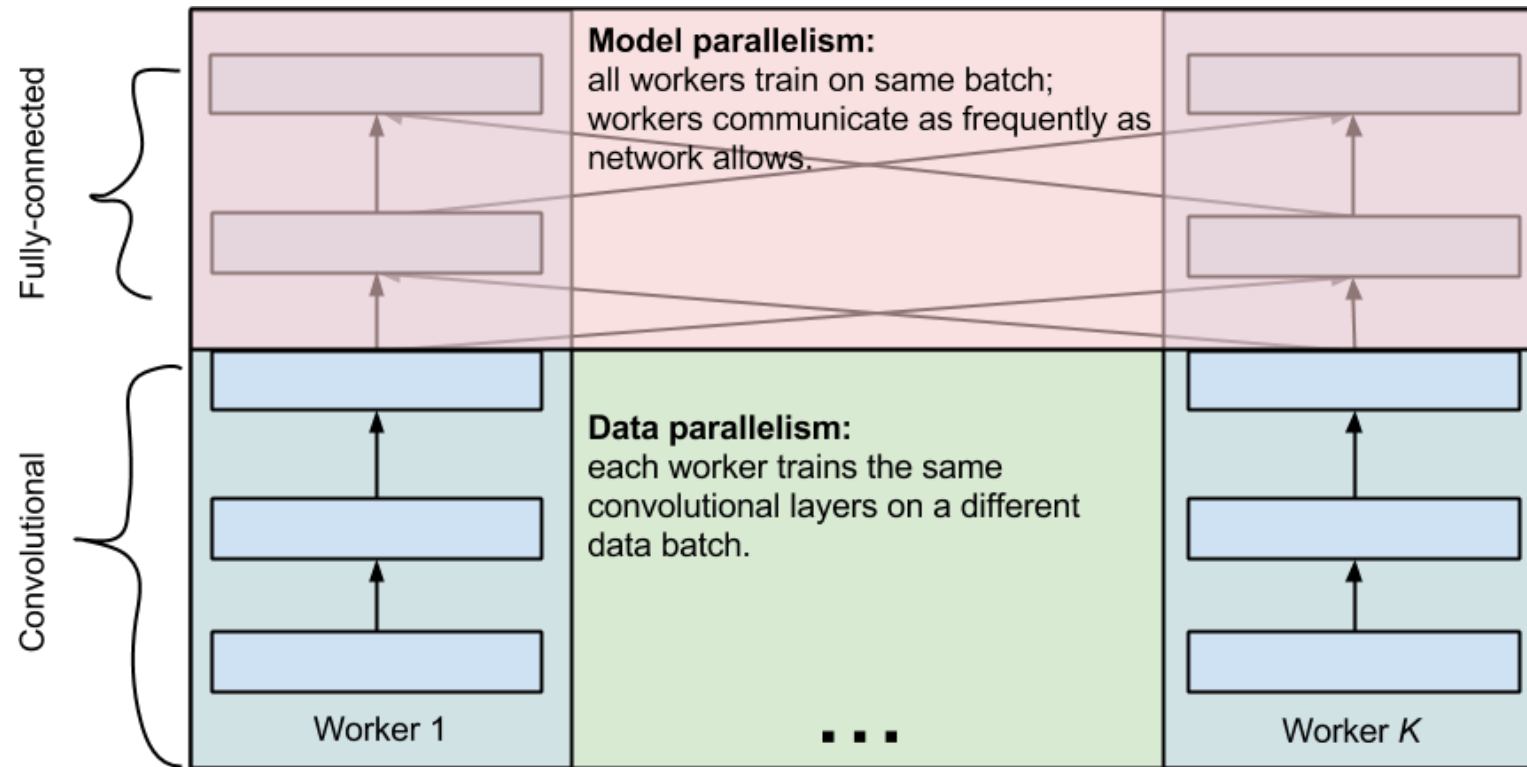
- Fully-connected layers
 - 5-10% of the computation
 - 95% of the parameters
 - Small intermediate activations

Tensor model parallelism

- **Discussion: how to parallelize CNNs?**

Parallelizing Convolutional Neural Networks

- Data parallelism for convolutional layers
- Tensor model parallelism for fully-connected layers

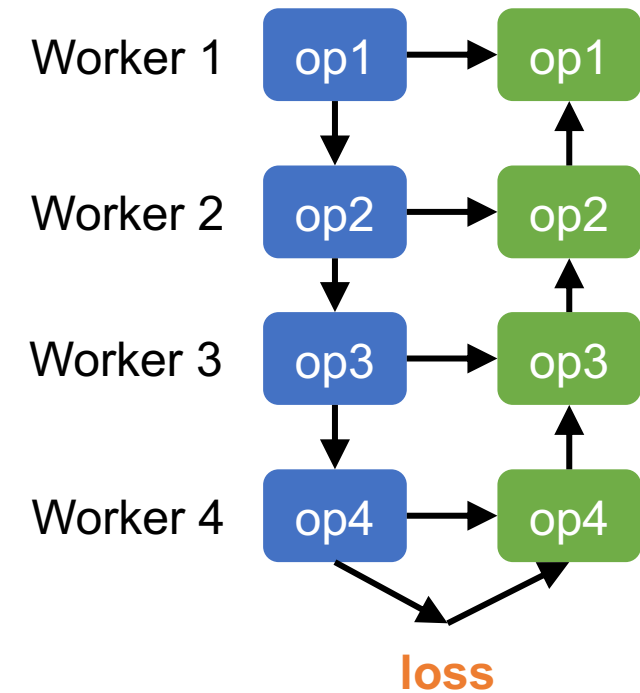
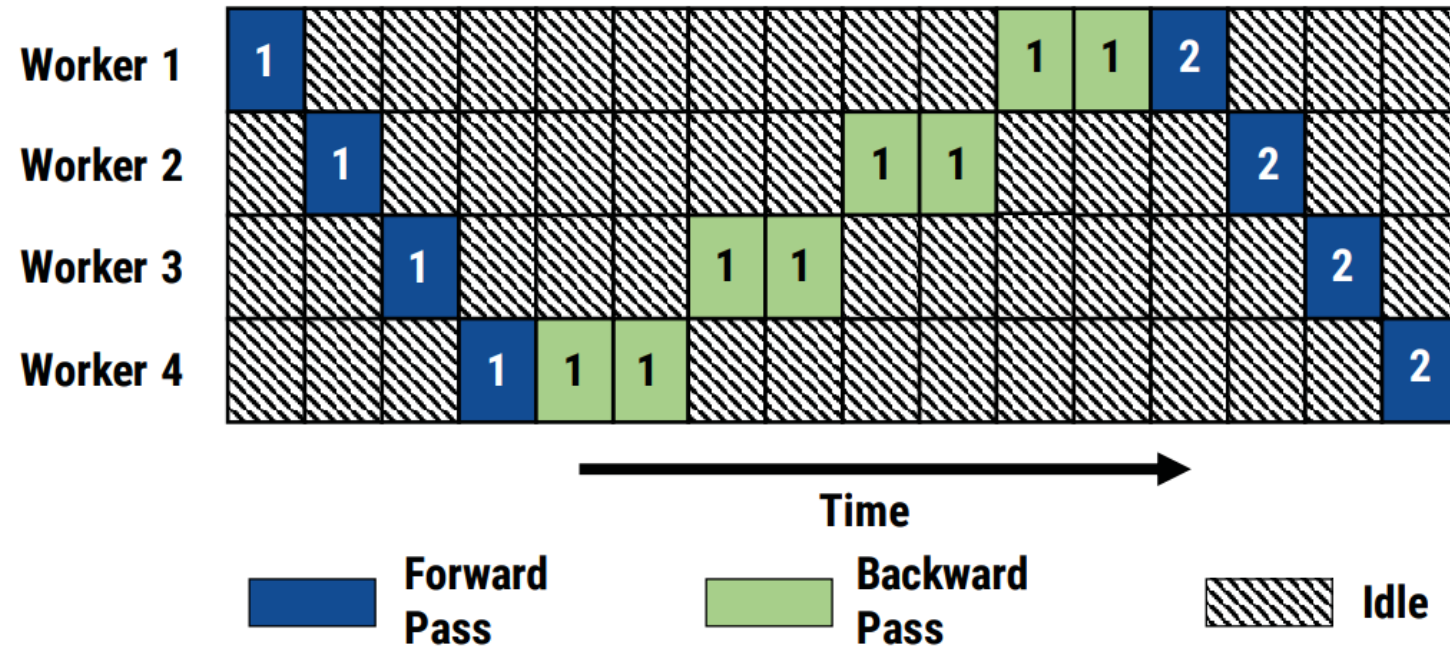


How to parallelize DNN Training?

- Data parallelism
- Model parallelism
- Tensor model parallelism
- **Pipeline model parallelism**

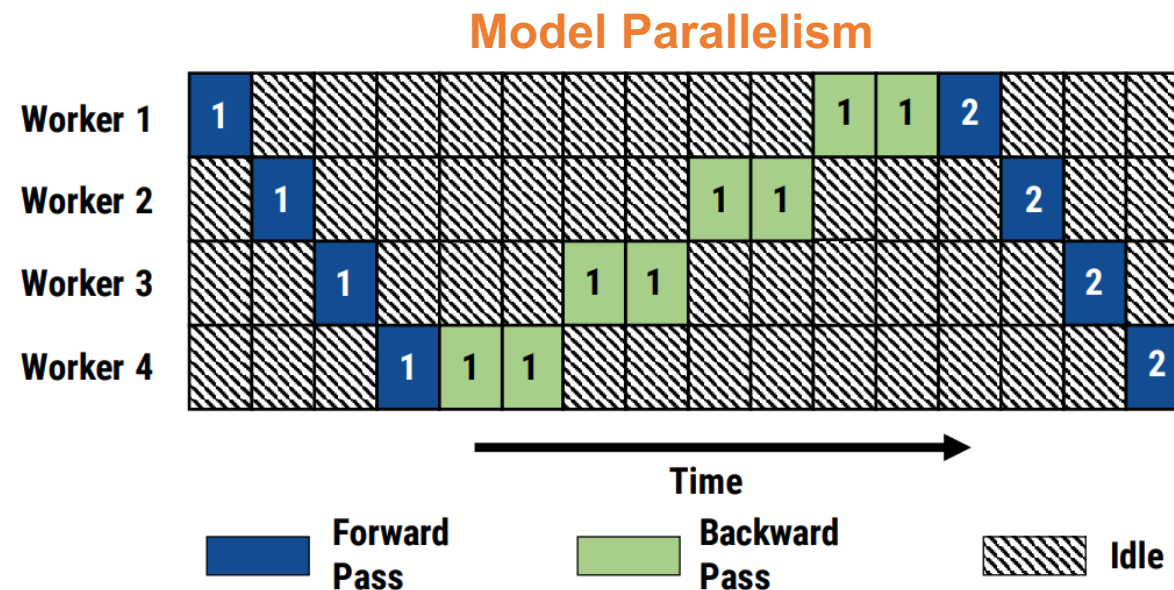
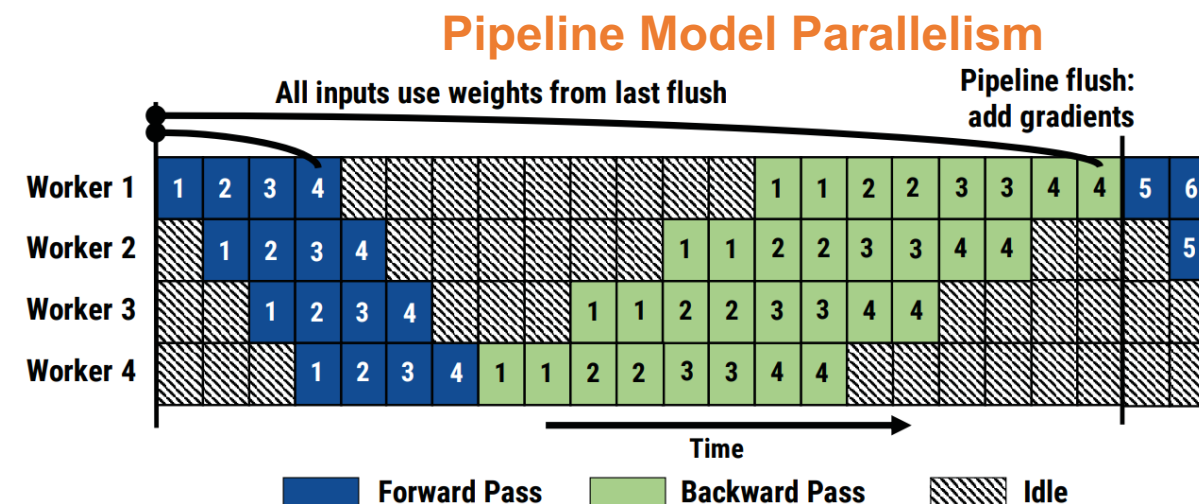
An Issue with Model Parallelism

- Under-utilization of compute resources
- Low overall throughput due to resource utilization



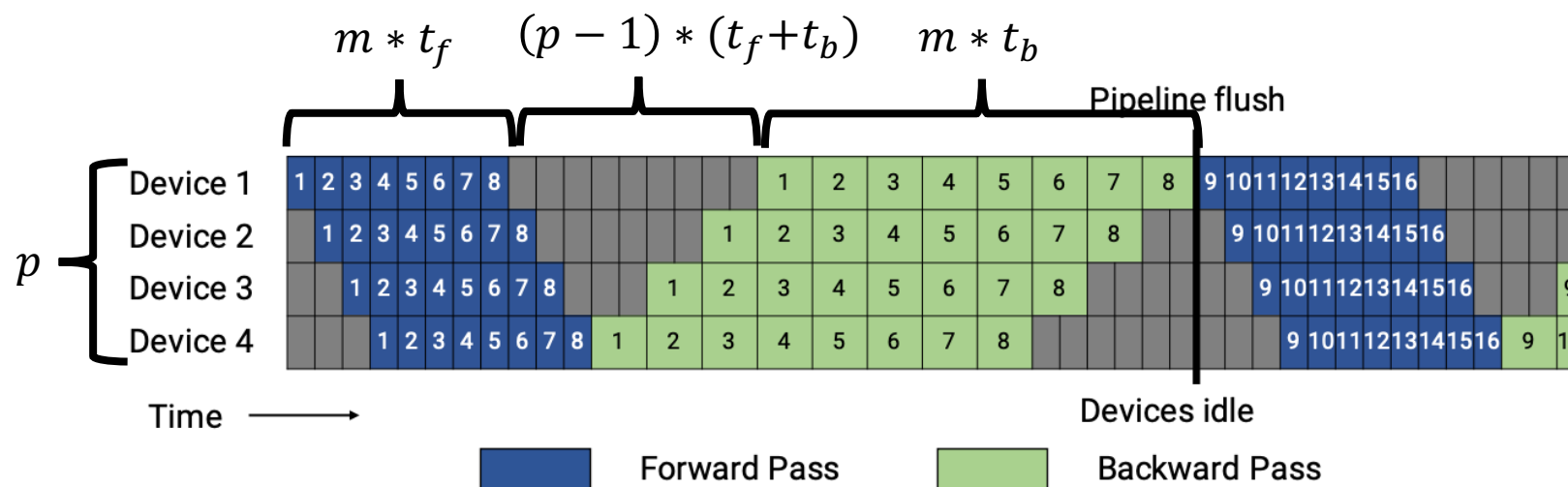
Pipeline Model Parallelism

- **Mini-batch**: the number of samples processed in each iteration
- Divide a mini-batch into multiple **micro-batches**
- Pipeline the forward and backward computations across micro-batches



Pipeline Model Parallelism: Device Utilization

- m : micro-batches in a mini-batch
- p : number of pipeline stages
- All stages take t_f/ t_b to process a forward (backward) micro-batch

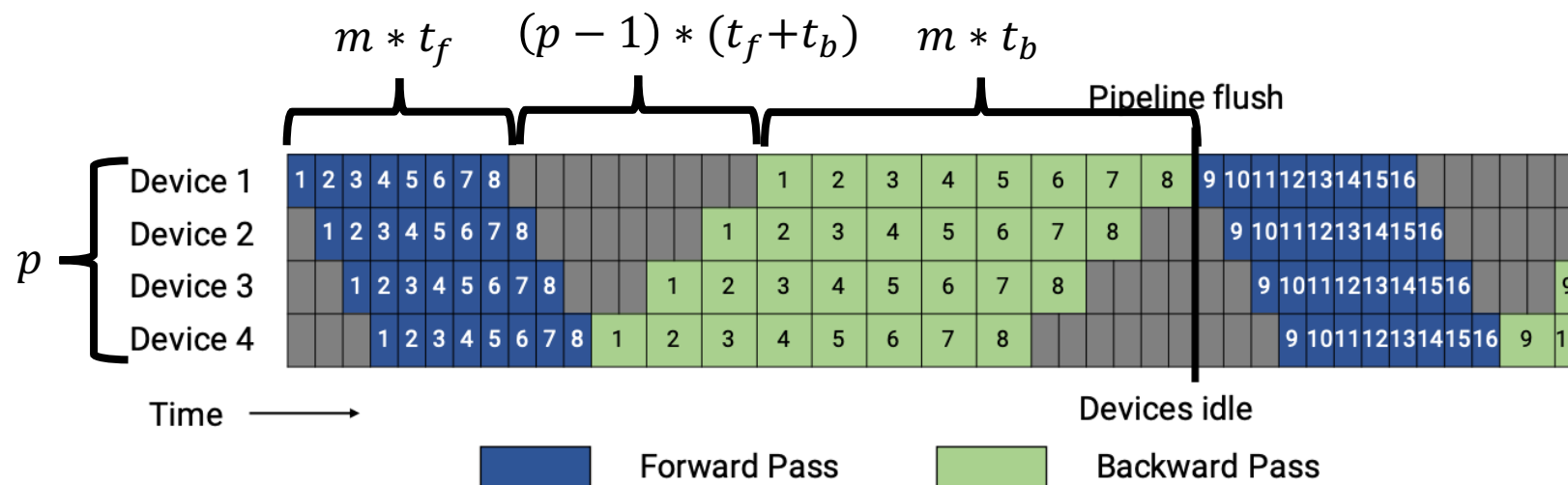


$$\text{BubbleFraction} = \frac{(p-1) * (t_f + t_b)}{m * t_f + m * t_b} = \frac{p-1}{m}$$

Improving Pipeline Parallelism Efficiency

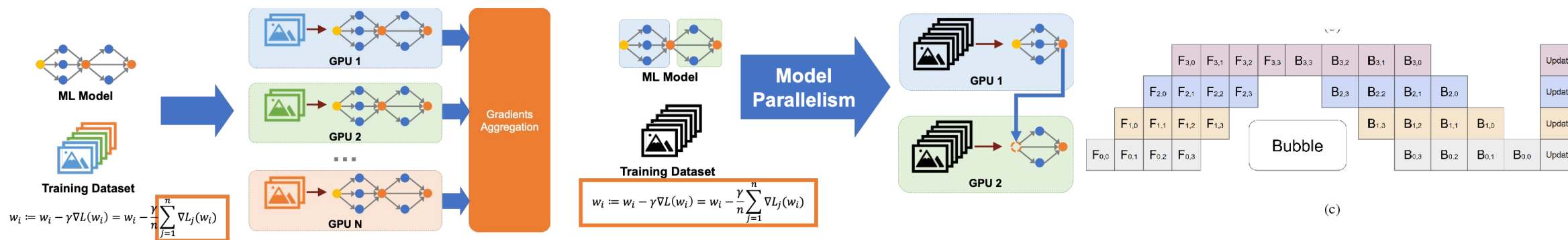
- m : number of micro-batches in a mini-batch
 - Increase mini-batch size or reduce micro-batch size
 - Caveat: large mini-batch sizes can lead to accuracy loss; small micro-batch sizes reduce GPU utilization
- p : number of pipeline stages
 - Decrease pipeline depth
 - Caveat: increase stage size

$$m = \frac{\text{minibatch}}{\text{microbatch}}$$



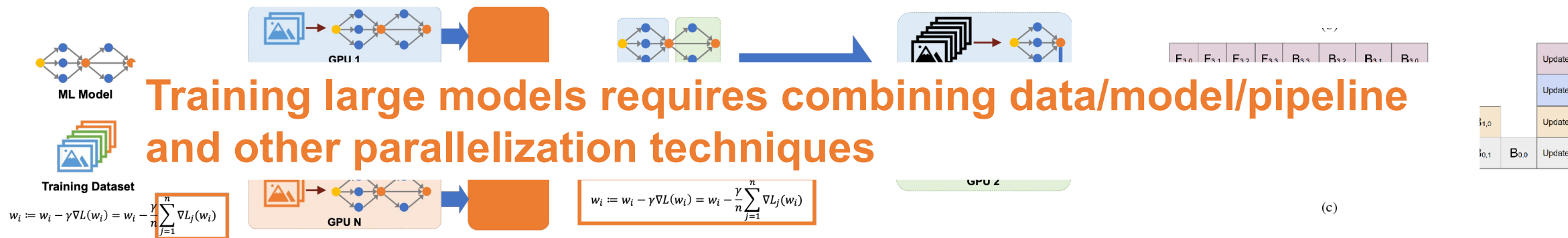
$$\text{BubbleFraction} = \frac{(p - 1) * (t_f + t_b)}{m * t_f + m * t_b} = \frac{p - 1}{m}$$

Summary: Comparing Data/Model/Pipeline Parallelism



	Data Parallelism	Model Parallelism	Pipeline Parallelism
Pros	<ul style="list-style-type: none"> ✓ Massively parallelizable ✓ Require no communication during forward/backward 	<ul style="list-style-type: none"> ✓ Support training large models ✓ Efficient for models with large numbers of parameters 	<ul style="list-style-type: none"> ✓ Support large-batch training ✓ Efficient for deep models
Cons	<ul style="list-style-type: none"> ❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters 	<ul style="list-style-type: none"> ❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward 	<ul style="list-style-type: none"> ❖ Limited utilization: bubbles in forward/backward

Summary: Data/Model/Pipeline Parallelism



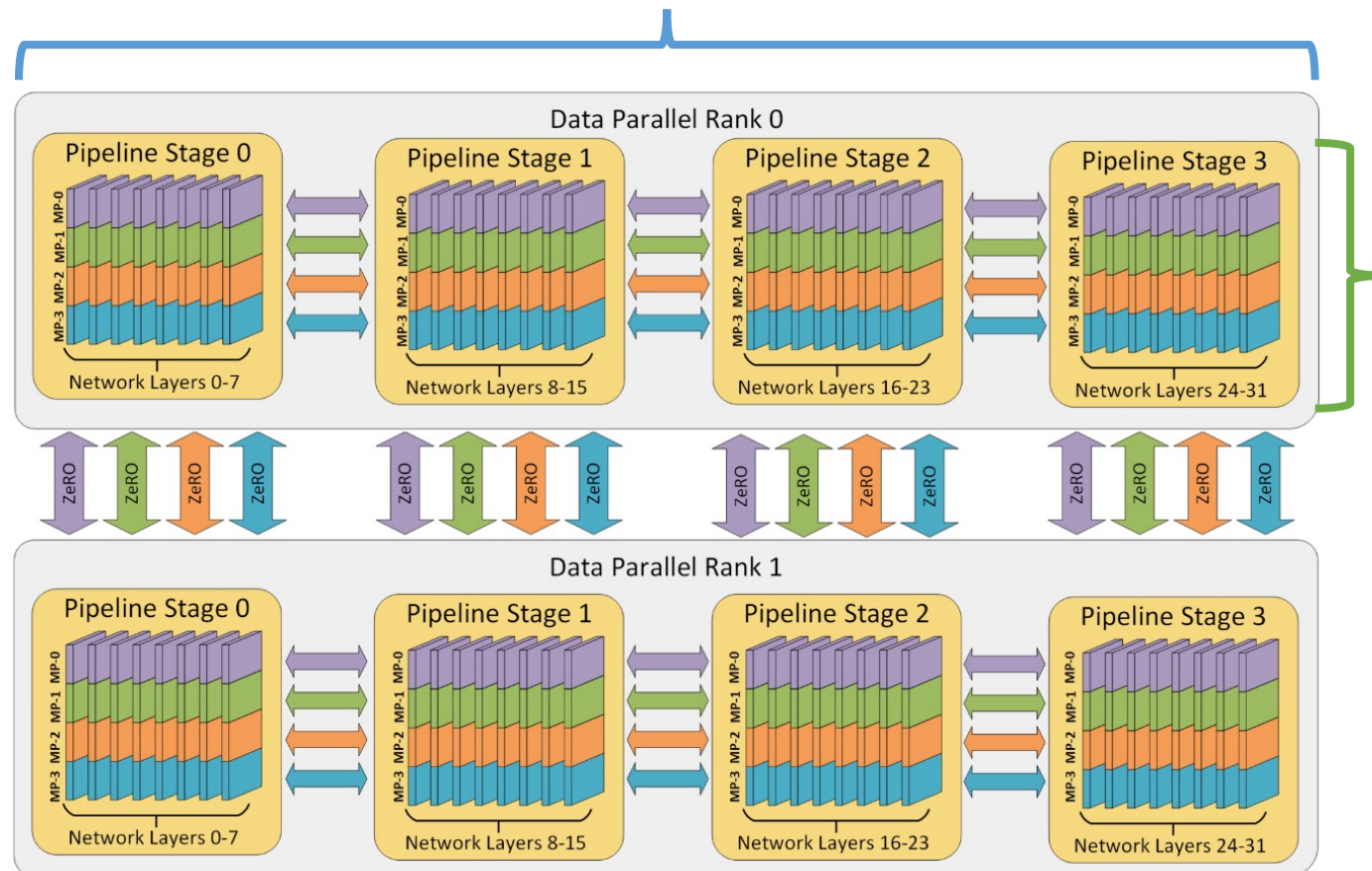
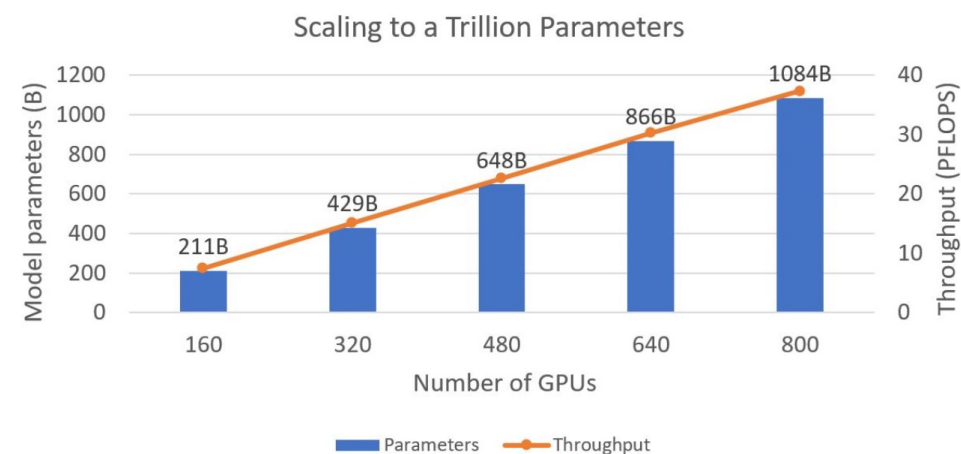
	Data Parallelism	Model Parallelism	Pipeline Parallelism
Pros	<ul style="list-style-type: none"> ✓ Massively parallelizable ✓ Require no communication during forward/backward 	<ul style="list-style-type: none"> ✓ Support training large models ✓ Efficient for models with large numbers of parameters 	<ul style="list-style-type: none"> ✓ Support large-batch training ✓ Efficient for deep models
Cons	<ul style="list-style-type: none"> ❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters 	<ul style="list-style-type: none"> ❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward 	<ul style="list-style-type: none"> ❖ Limited utilization: bubbles in forward/backward

Example: 3D parallelism in DeepSpeed

Pipeline Model Parallelism

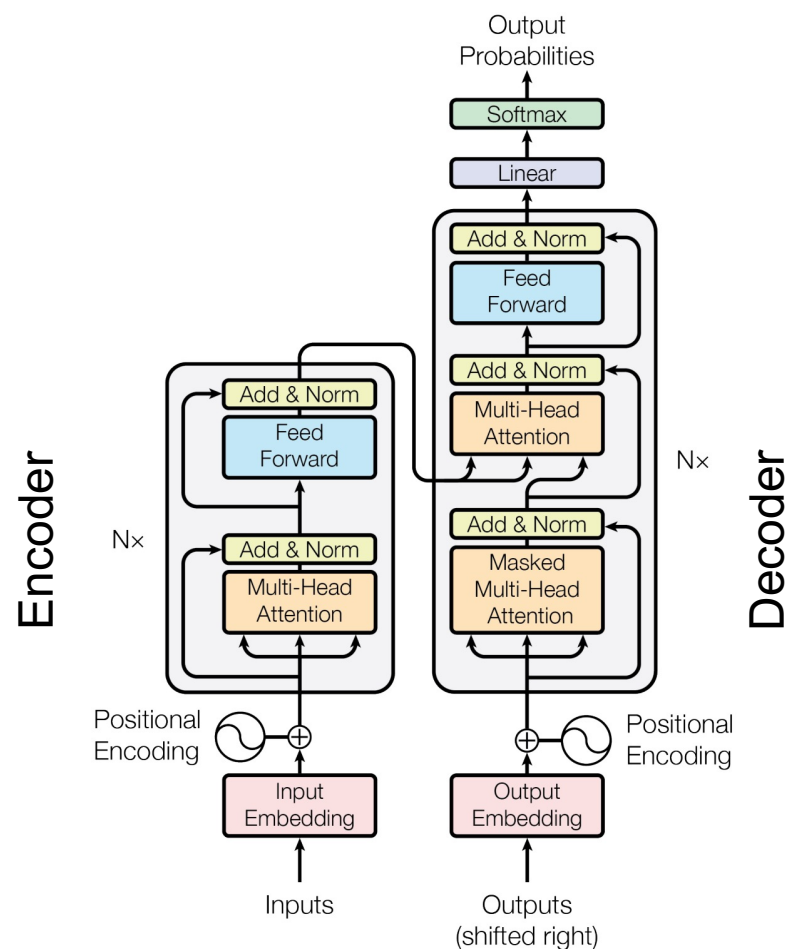
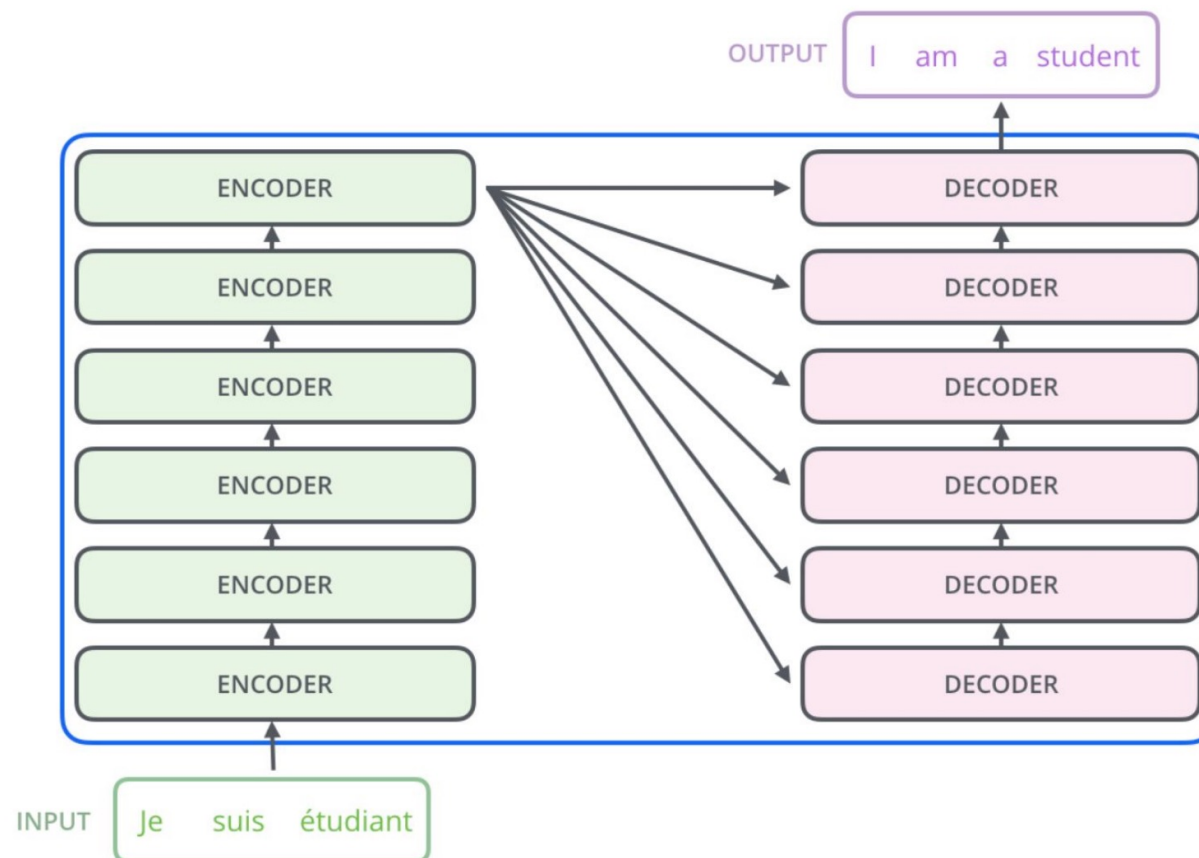
Data Parallelism

Tensor Model Parallelism



Example: Parallelizing Transformers

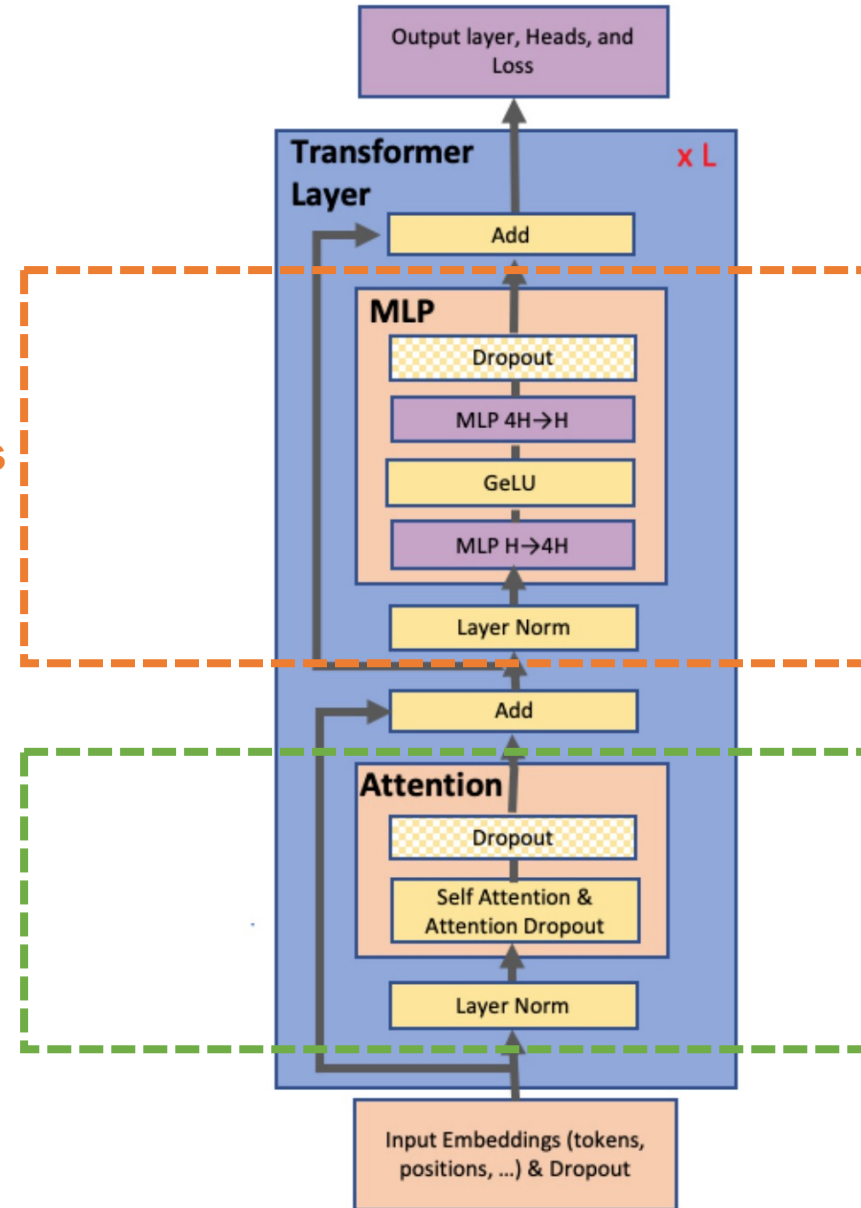
- Transformer: attention mechanism for language understanding



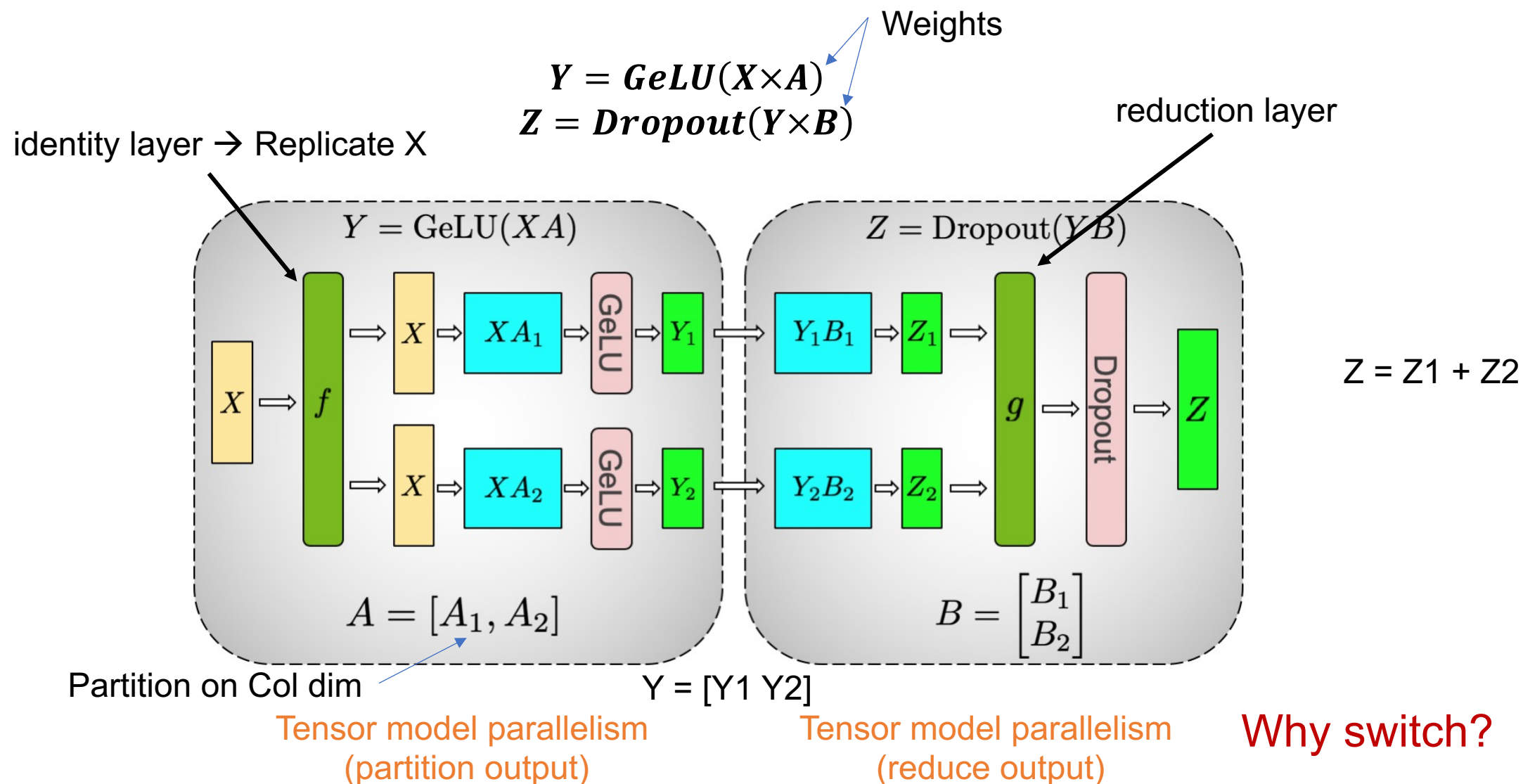
A Single Transformer Layer

Fully-Connected Layers

Self-Attention Layers



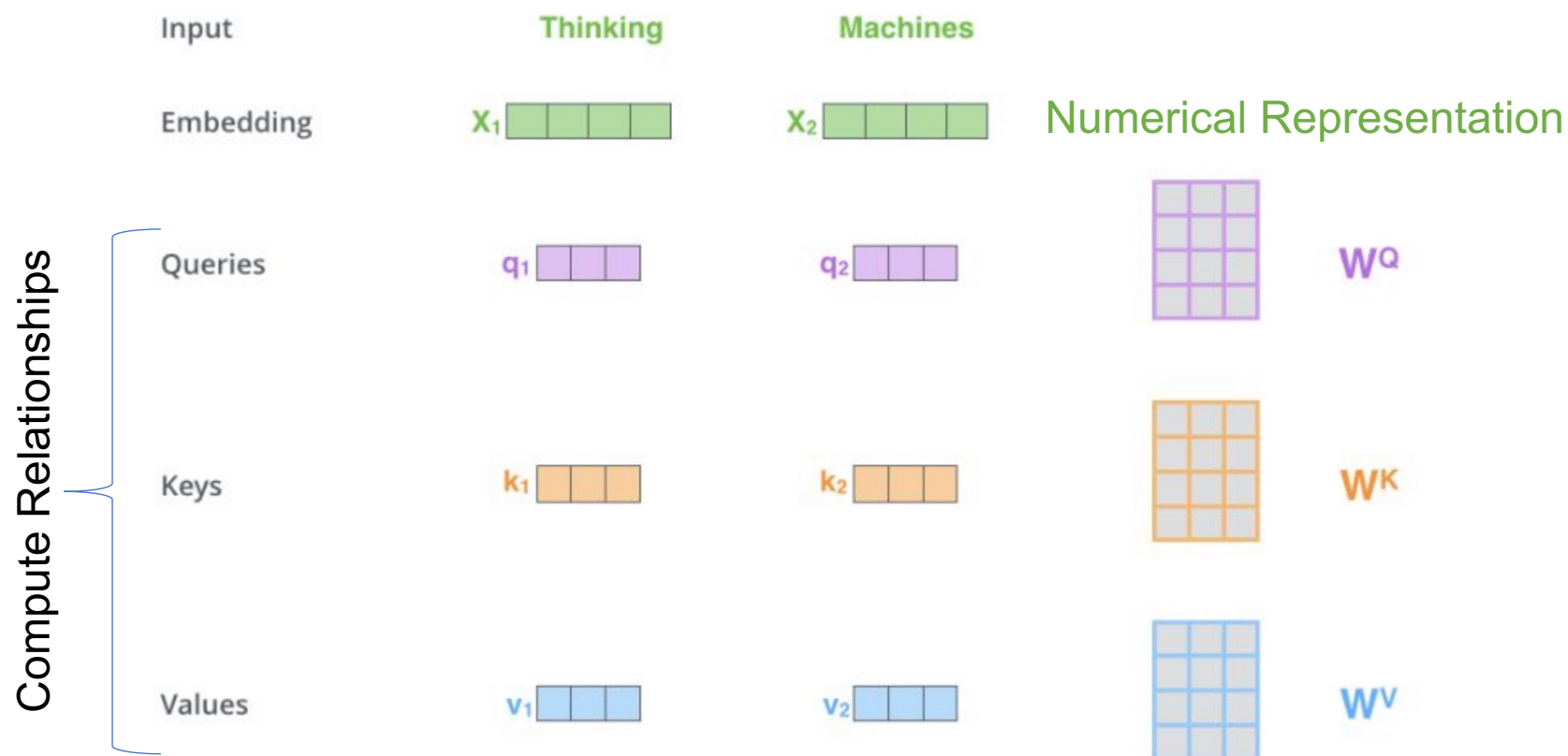
Parallelizing Fully-Connected Layers in Transformers



Self-Attention

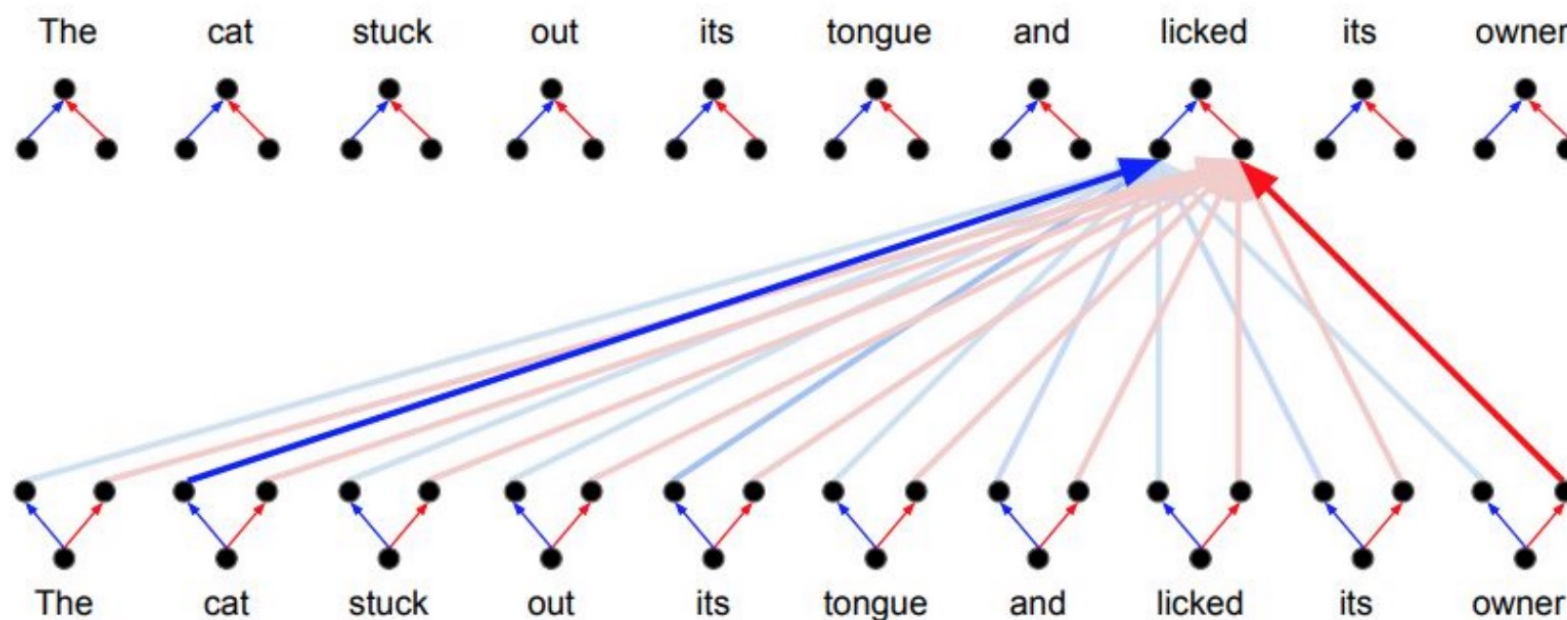
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

- Mapping a query and a set of key-value pairs to an output



Multi-Head Self-Attention

- Parallelize attention layers with different linear transformations on input and output
- **Benefits: more parallelism, reduced computation cost**

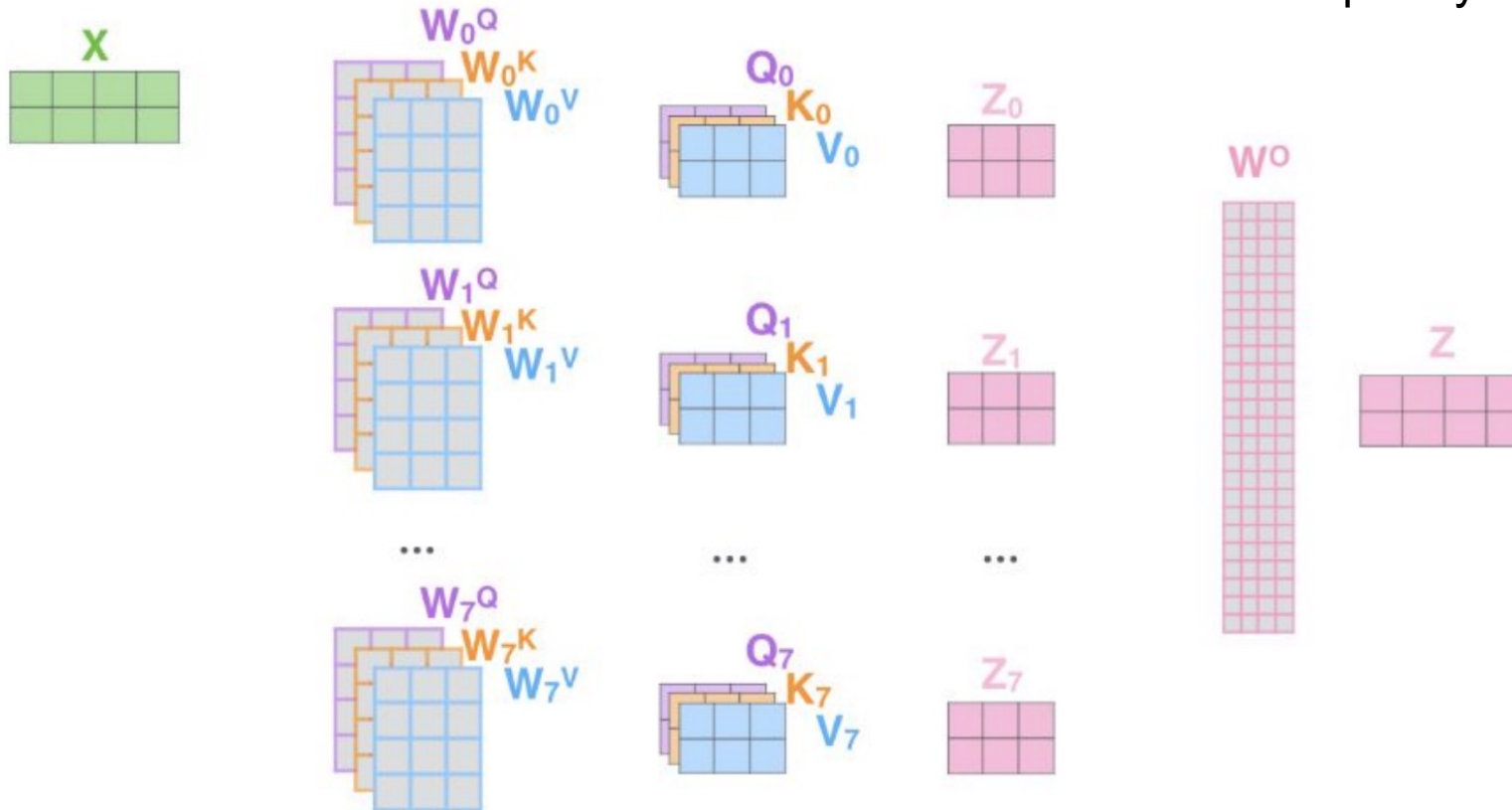


Multi-Head Self-Attention

Why multi-head attention?

→ Reduce complexity of matrix multiply!

Thinking
Machines



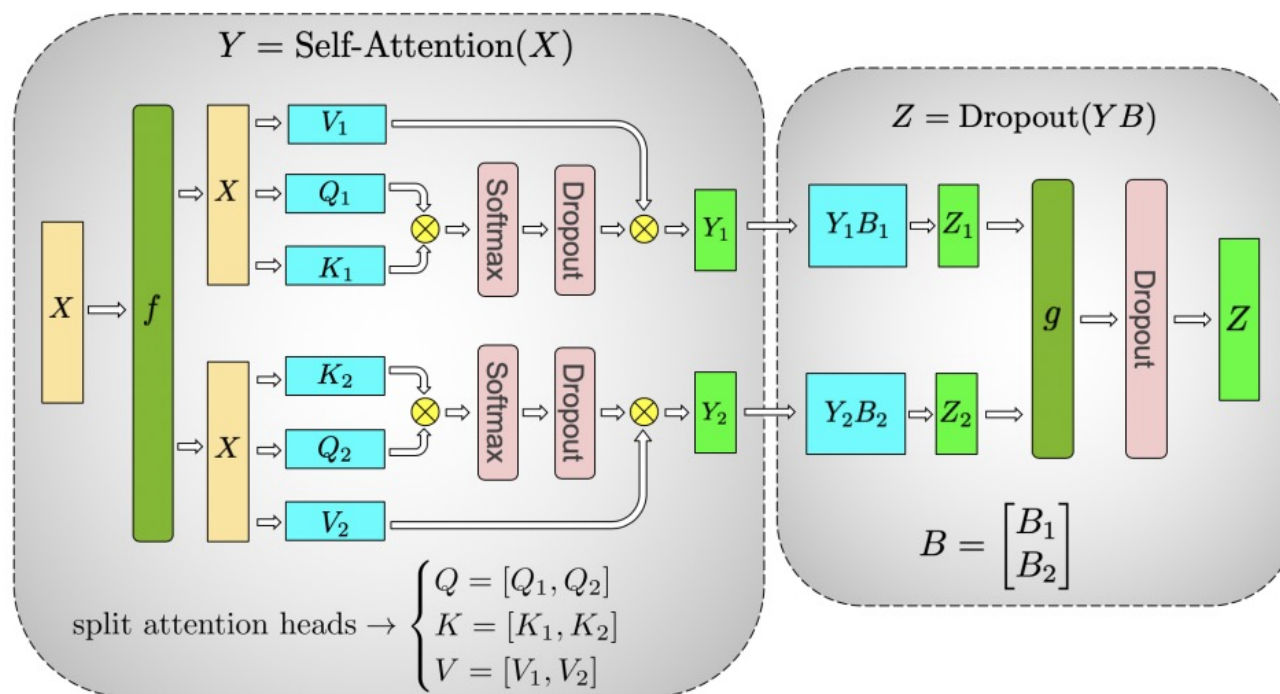
$$Z_i = A(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d}}\right) V_i$$

$$Z = \text{MultiHead}(Q, K, V) = \text{Concat}(Z_0, \dots, Z_7) W^O$$

Parallelizing Self-Attention Layers in Transformers

$$Y_i = A(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d}}\right) V_i$$

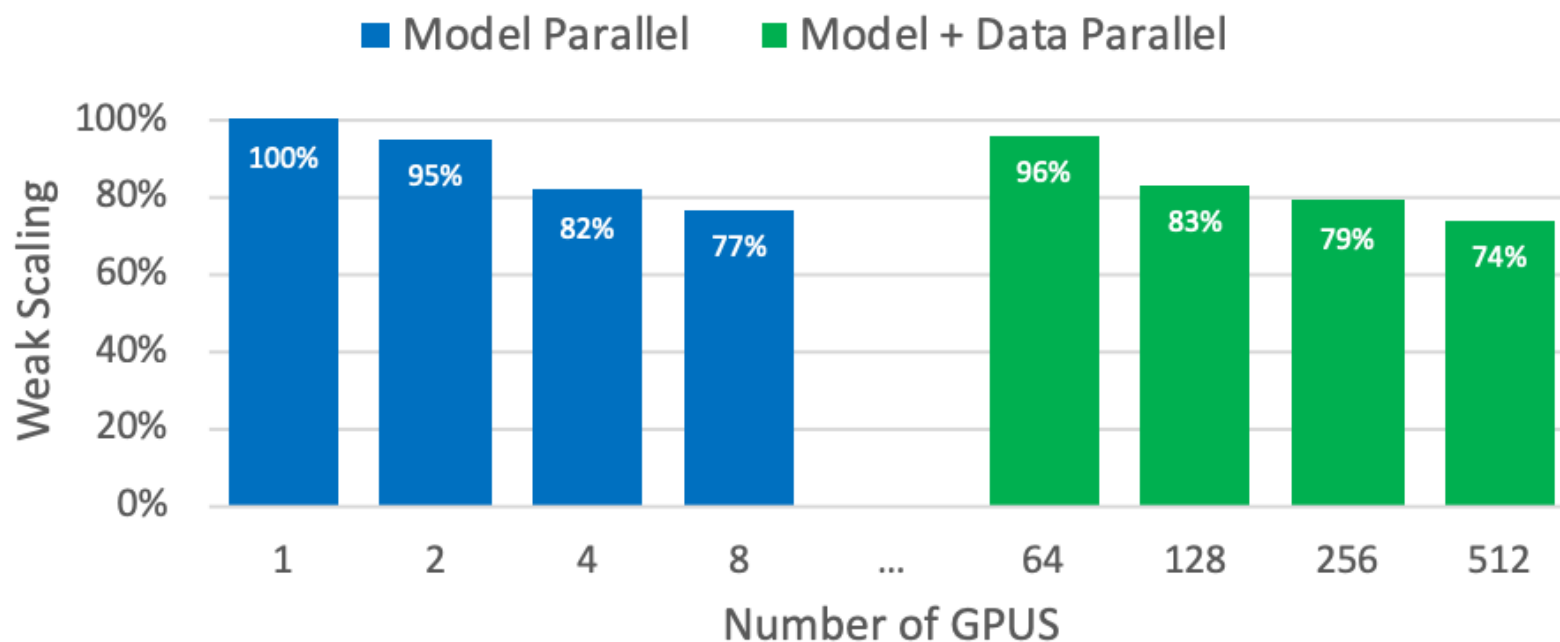
$$Z = \text{MultiHead}(Q, K, V) = \text{Concat}(Y_0, \dots, Y_h) W^o$$



**Parallelizing across
attention heads**

**Tensor model parallelism
(reduce output)**

Parallelizing Transformers



Scale to 512 GPUs by combining data and model parallelism