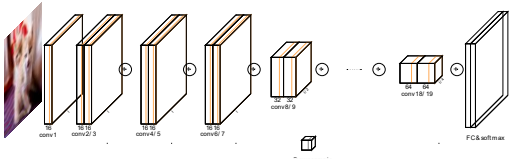
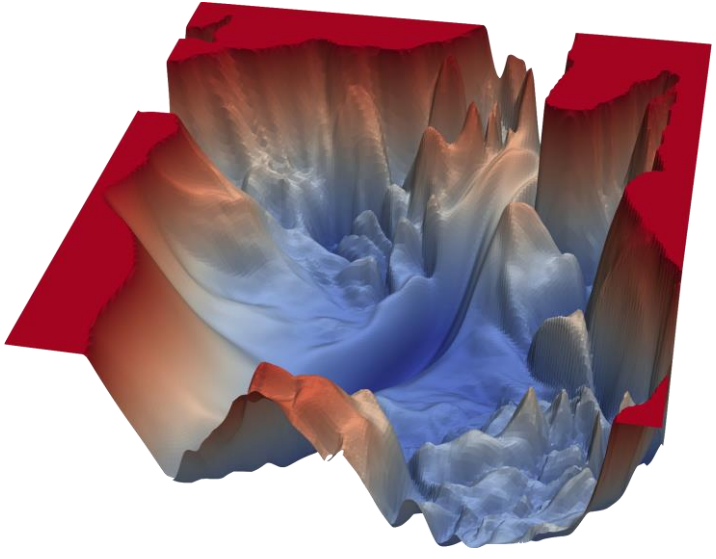
ACCELERATING DISTRIBUTED AI INFRASTRUCTURE SYSTEMS (II)

Review: Model Training Workload





Stochastic Gradient Descent (SGD)

$$\min_{w} \mathcal{J}(w) = \frac{1}{N} \sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \underbrace{\frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

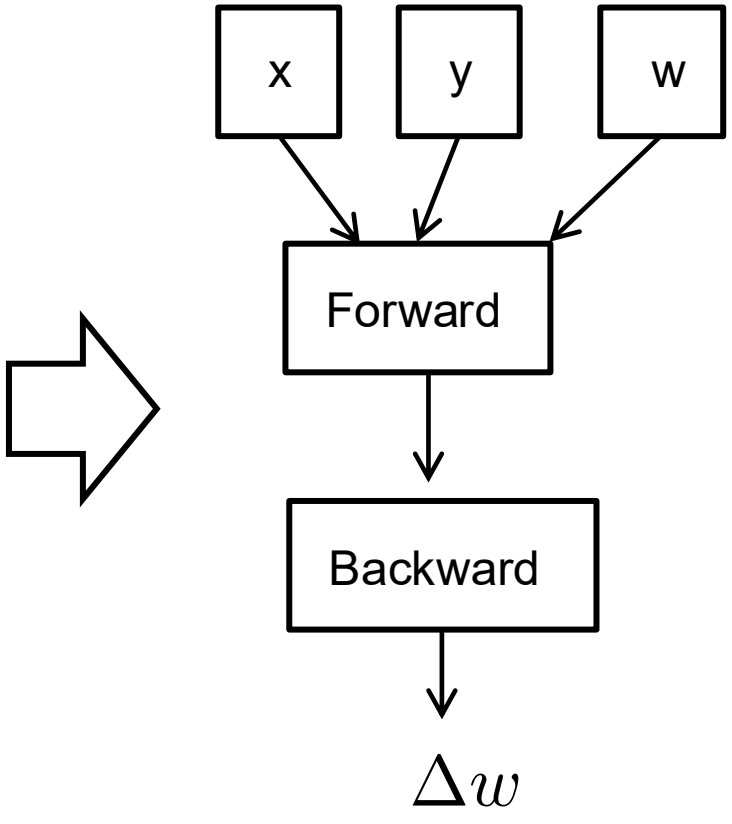
Review: Model Training into Computation Graph

X: input features

Y: labels in supervised learning

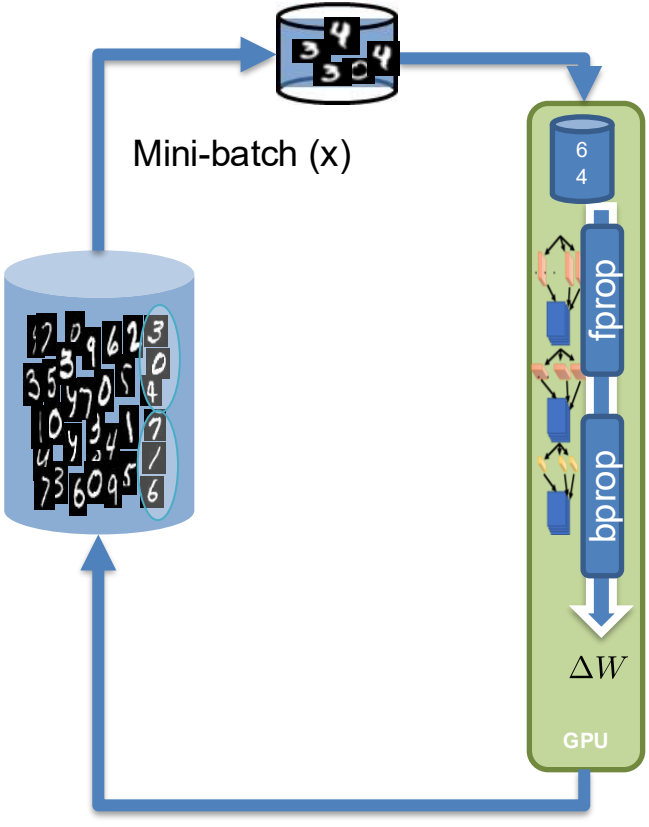
$$\min_{w} \mathcal{J}(w) = \frac{1}{N} \sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \underbrace{\frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$



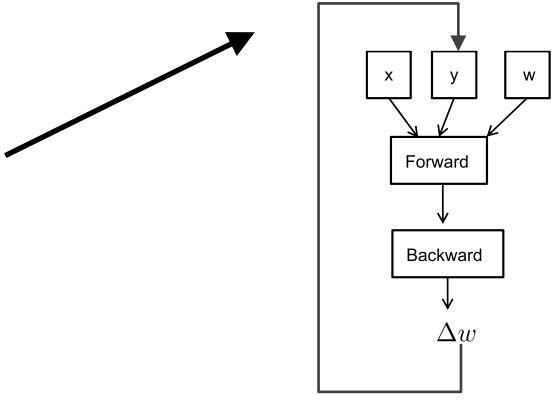
Review: Overview of Model Training in Action

In every iteration of SGD we load a random mini-batch of training data and compute gradient



$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \underbrace{\frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$



Review: Parallelization Opportunities

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Review: Data Parallelism

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

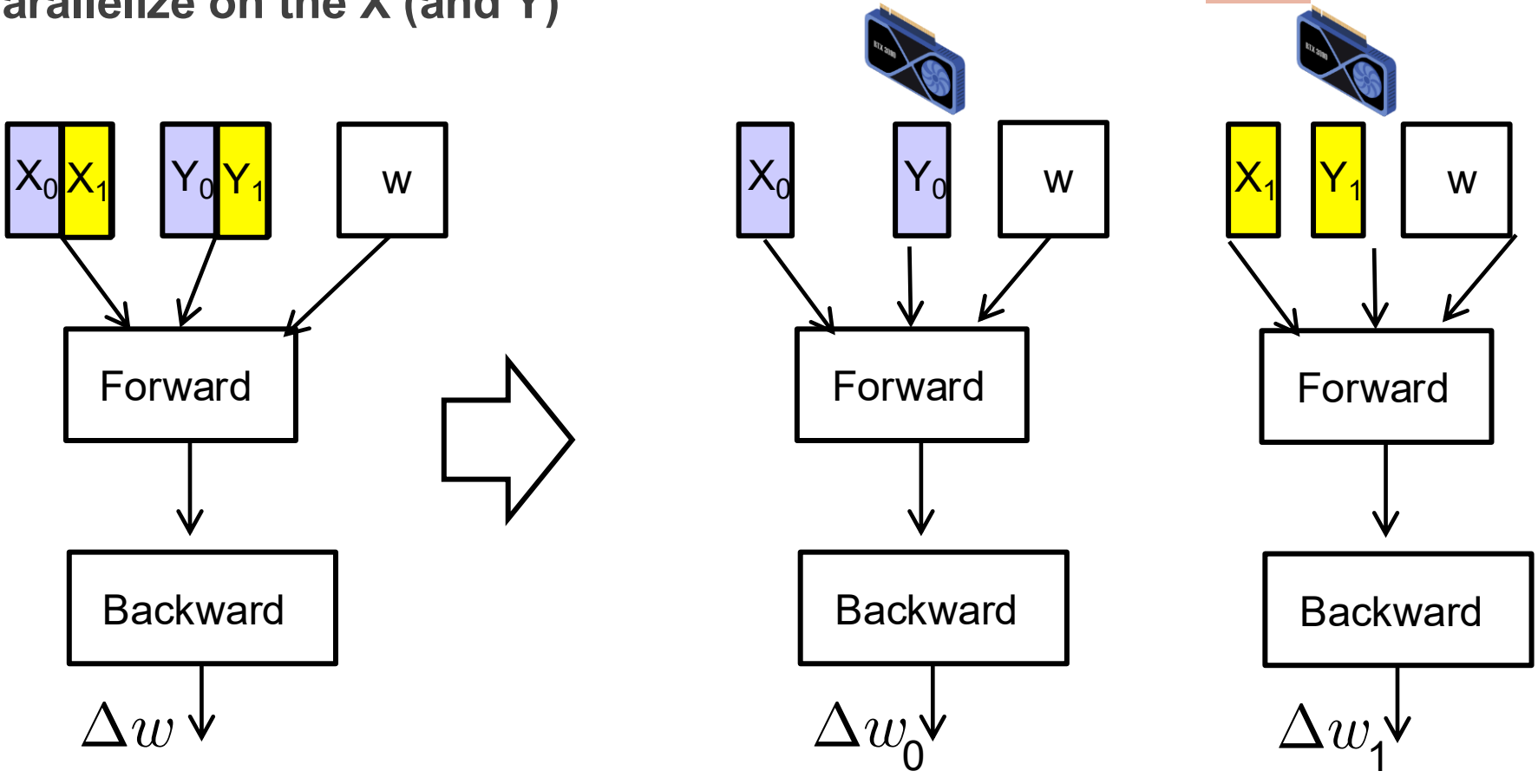
For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Review: Data Parallelism

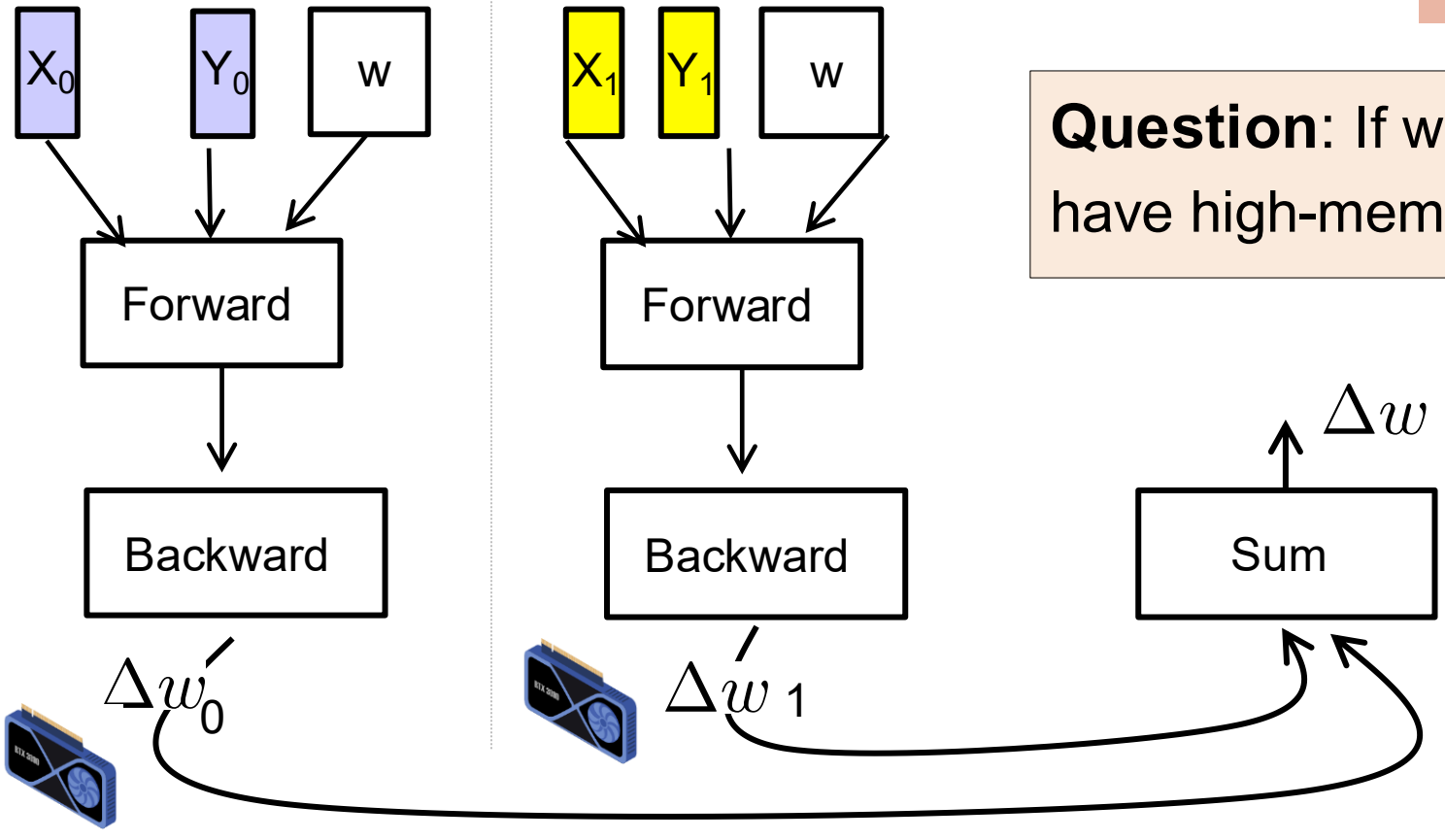
$$w^1 = w^0 - \frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Parallelize on the X (and Y)



Review: Data Parallelism

$$w^1 = w^0 - \frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}$$



Question: If we do not have high-memory GPUs?

Review: Model Parallelism

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

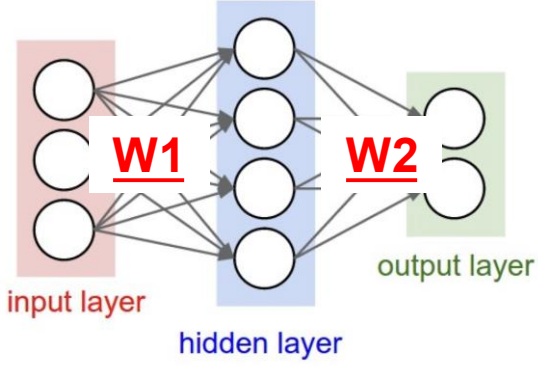
For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

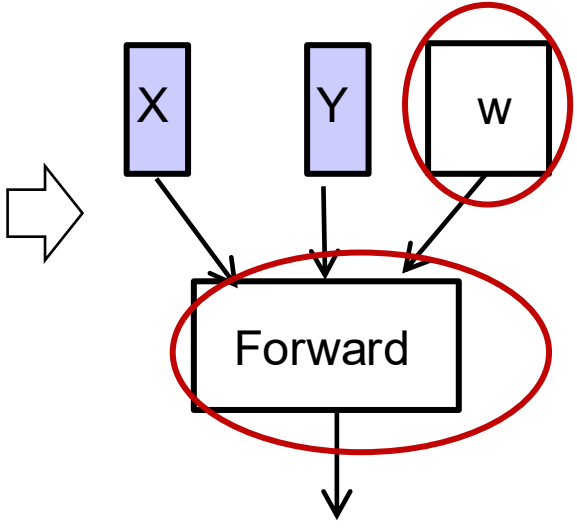
$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

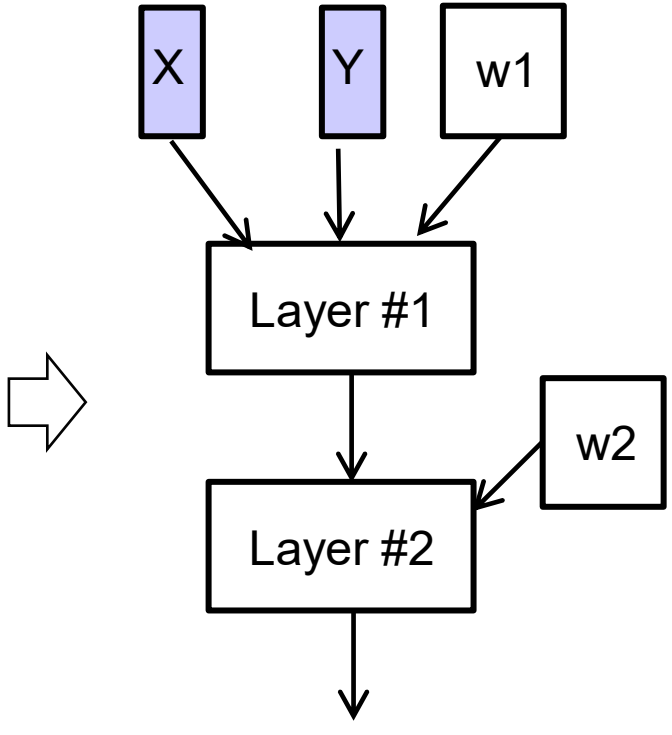
Review: Model Parallelism

Partition model parameters, typically done in two ways:

- **Partition on the W**: tensor parallelism
- **Partition on the layer**: pipeline parallelism



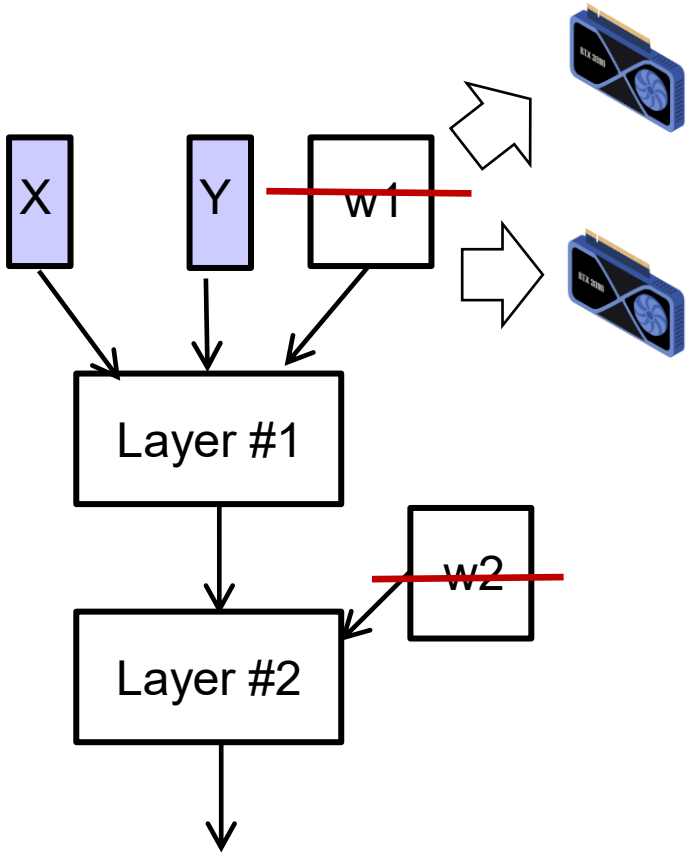




Review: Tensor Parallelism

Partition the parameters of a layer

- Each partition is deployed on a separate GPU
- **Reduced Memory Constraints**
- Allow for training **larger models** by leveraging multiple devices



Question: If we don't have high-memory GPUs and high-bandwidth interconnects?

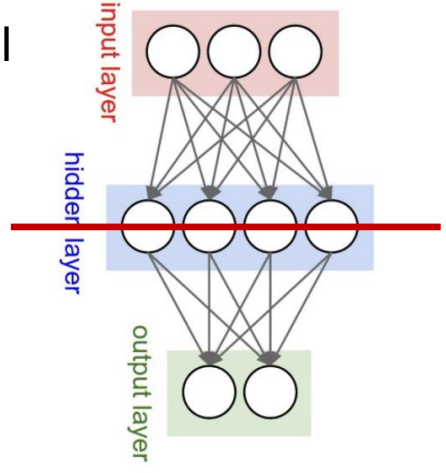
PIPELINE PARALLELISM

Pipeline Parallelism

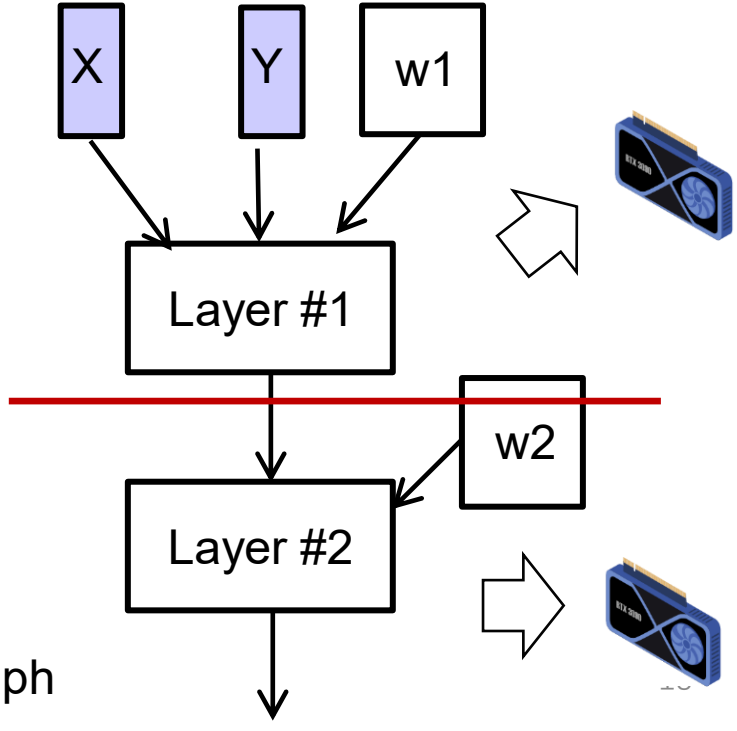
Partition the computation layer-by-layer, each partition is deployed on a device

- Layer's outputs are much smaller than weights
- Several layers can be grouped together

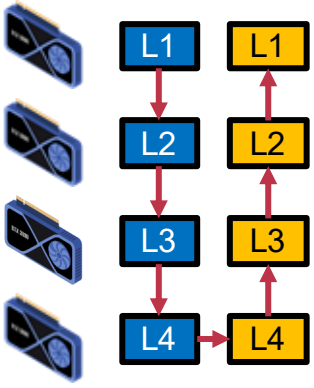
The model

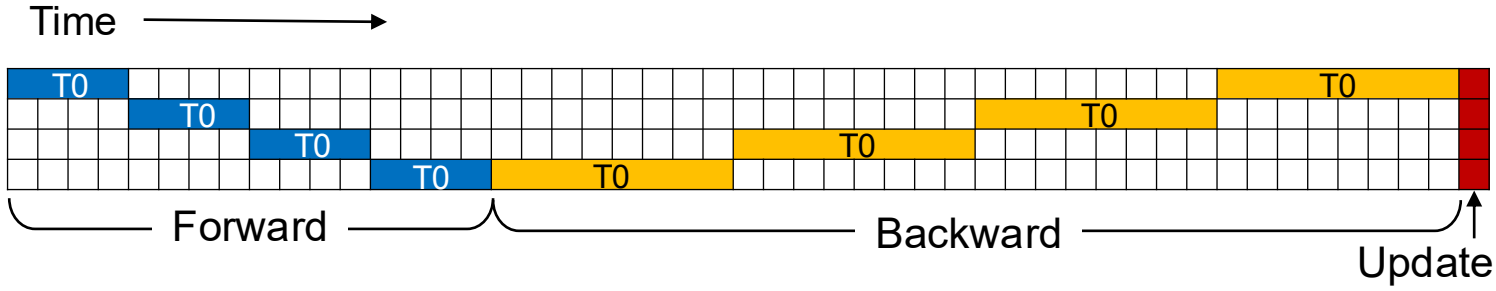


The partitioned computation graph

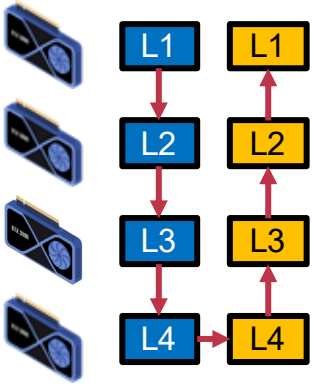


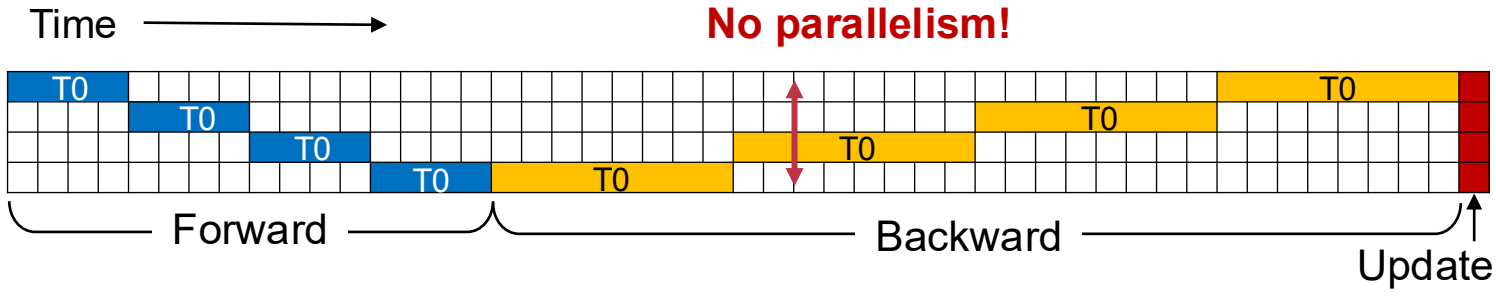
Computation in Layer-by-layer





Computation in Layer-by-layer





Question: How to parallelize?

Real-World Pipelines: Car Washes

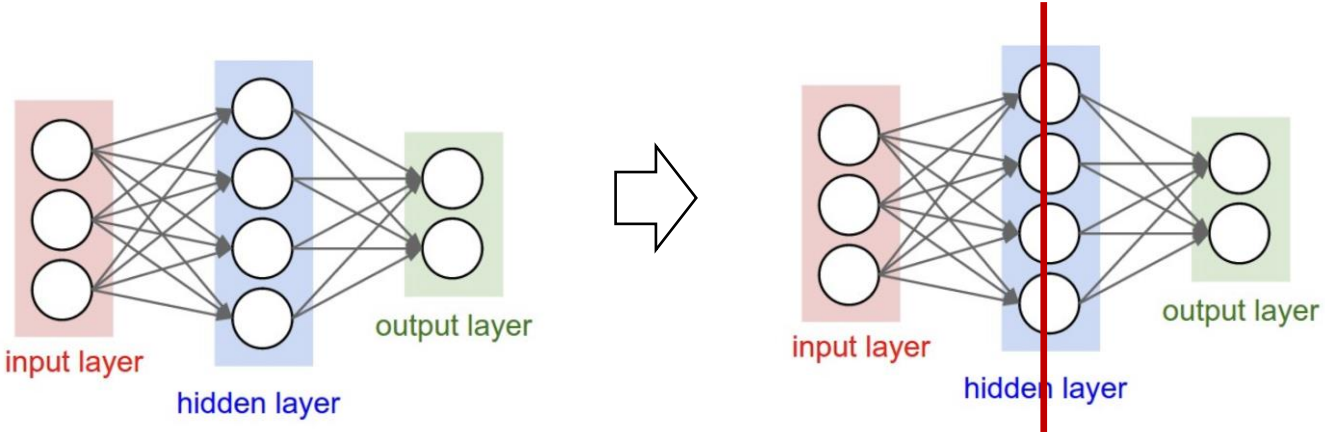
Pipelining

- Divide process into **independent** stages
- Move objects through stages in **sequence**
- At any given times, **multiple** objects being processed

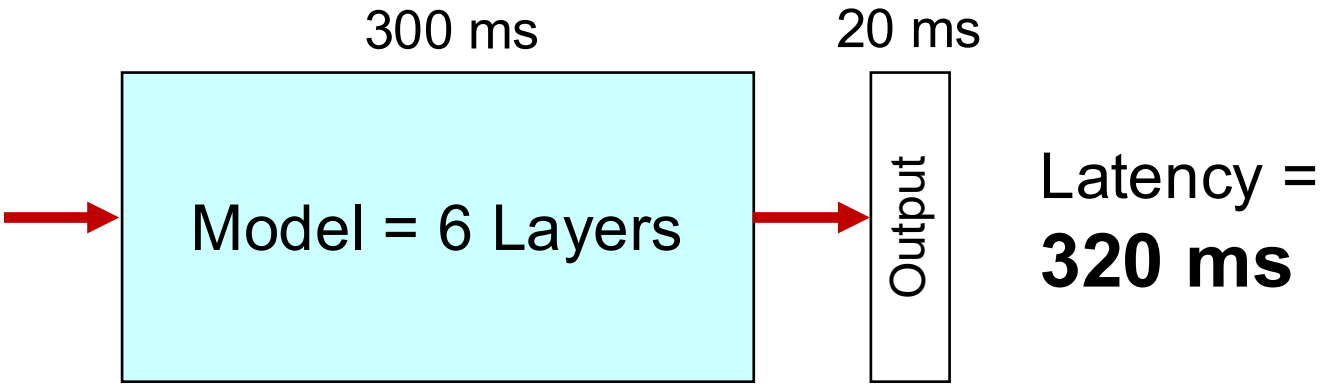
Pipelining Model Training

Pipelining

- Divide ~~process~~ **model** into independent ~~stages~~ **layers**
- Move ~~objects~~ **batches** through stages in sequence
- At any given times, multiple ~~objects~~ **batches** being processed

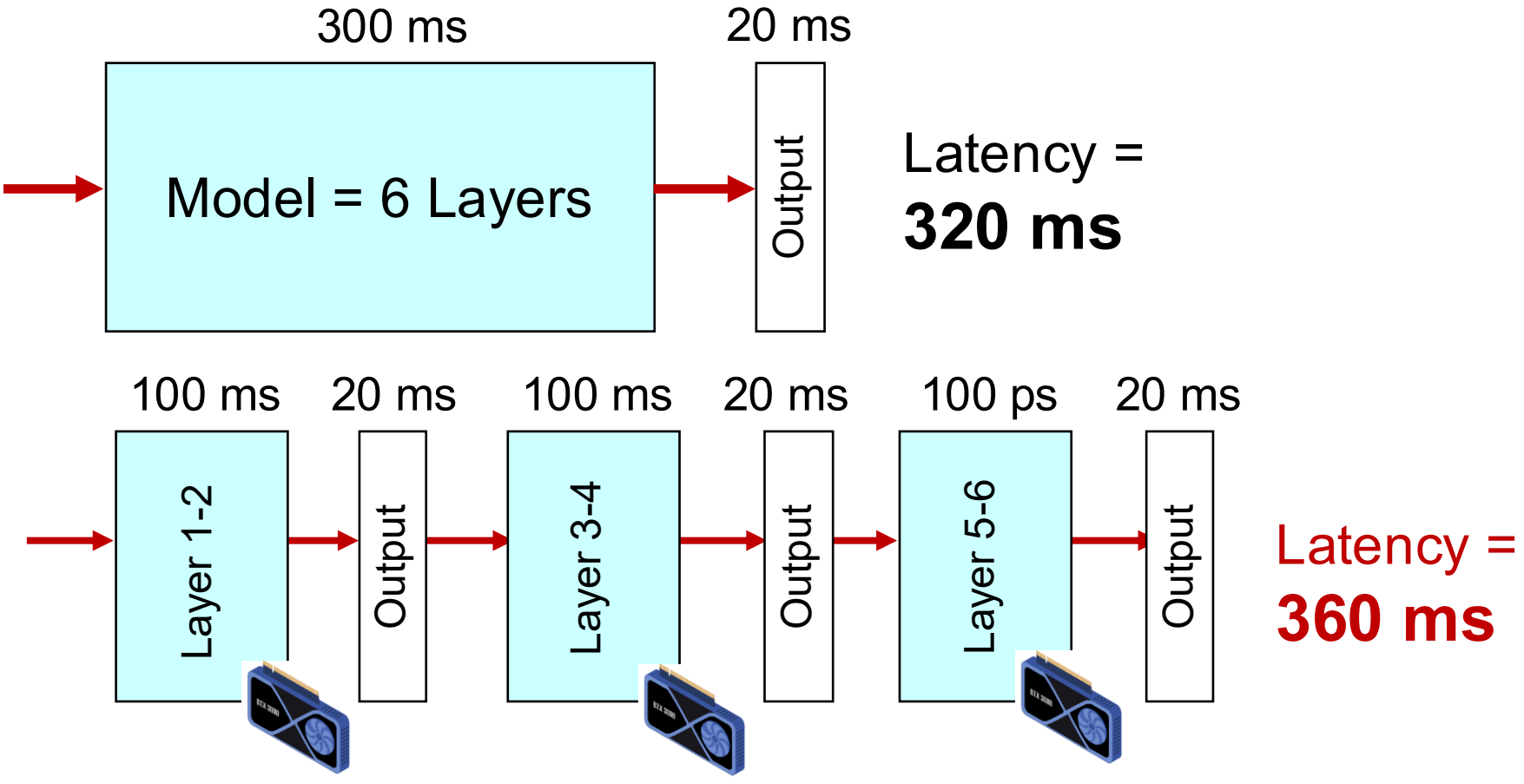


Pipeline Parallelism

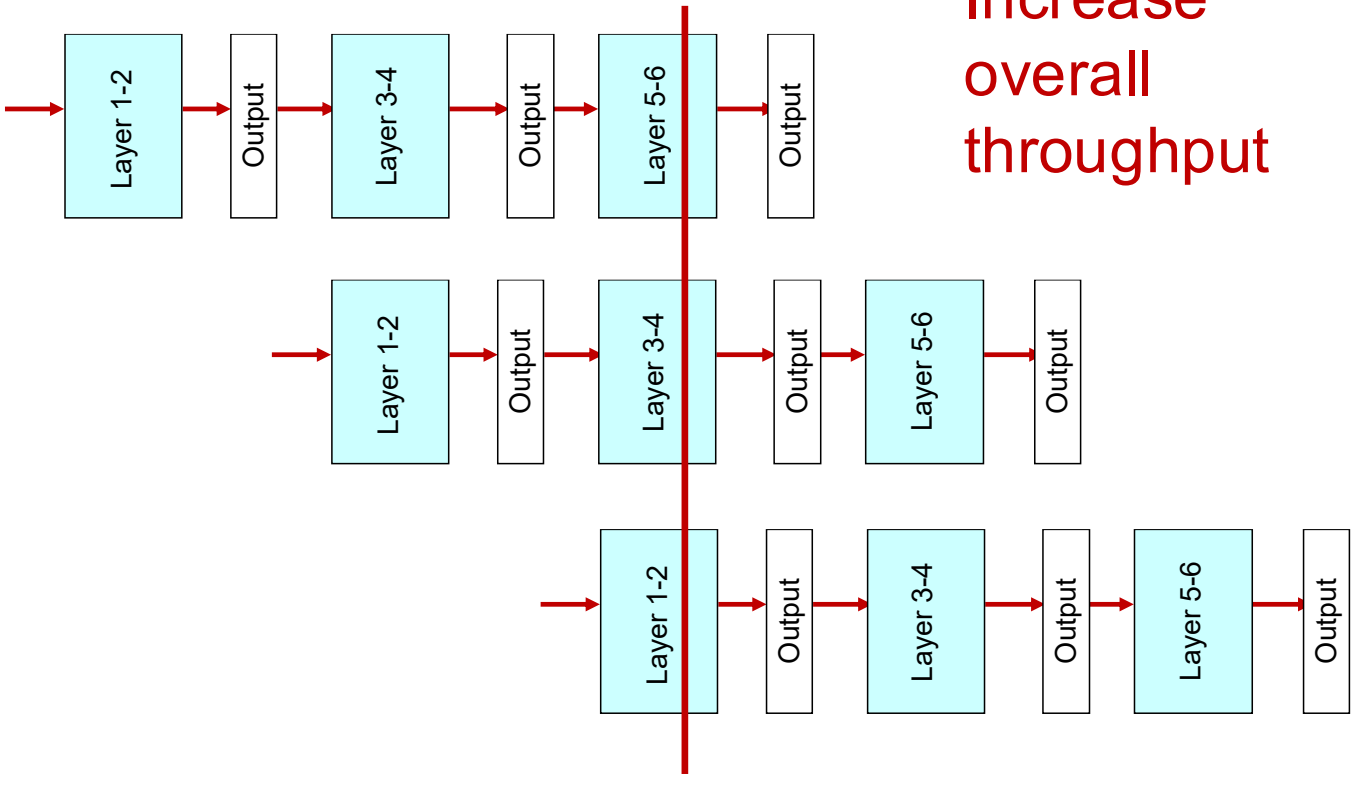


Latency =
320 ms

Pipeline Parallelism



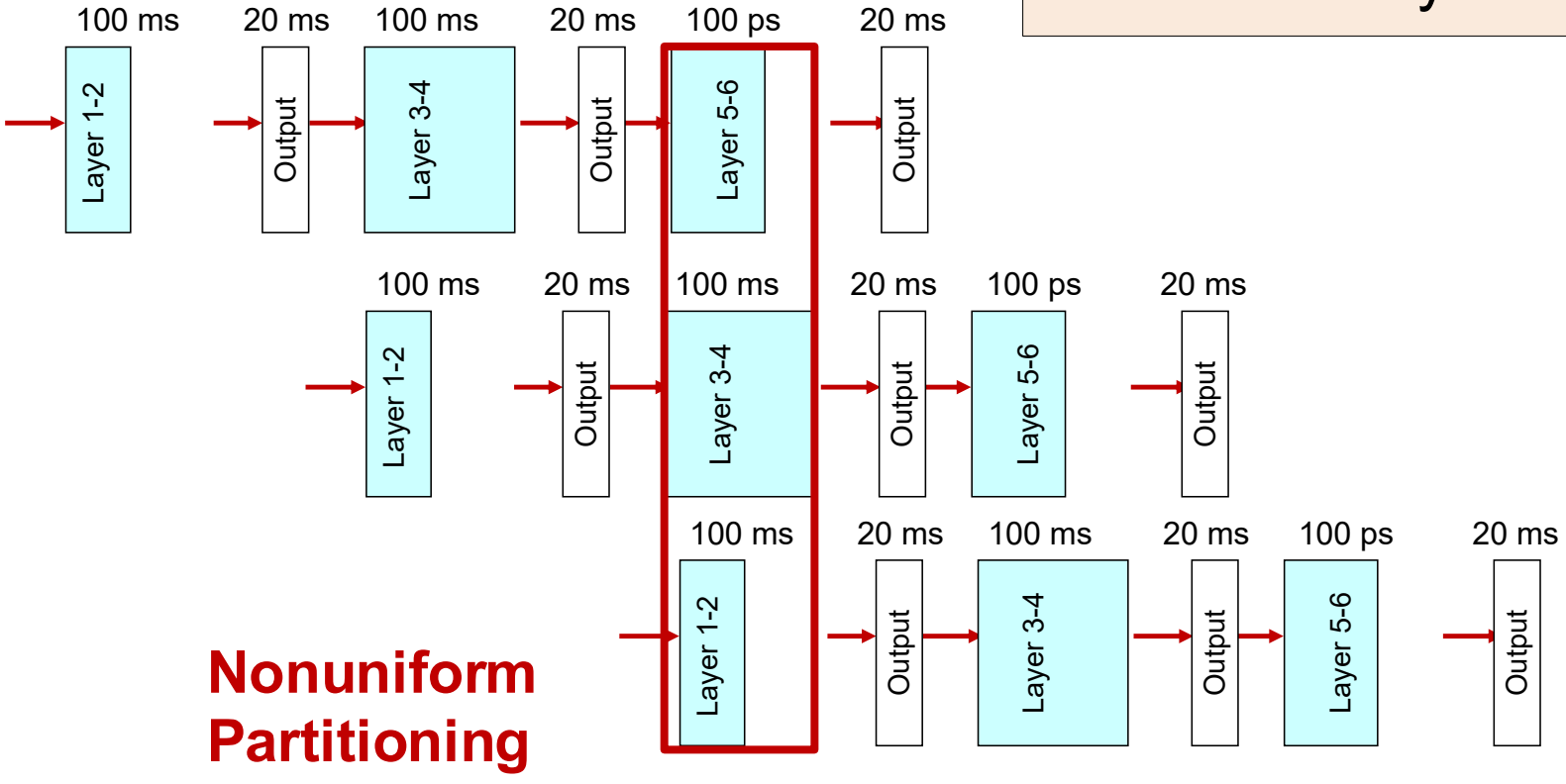
Pipeline Parallelism



Increase overall throughput

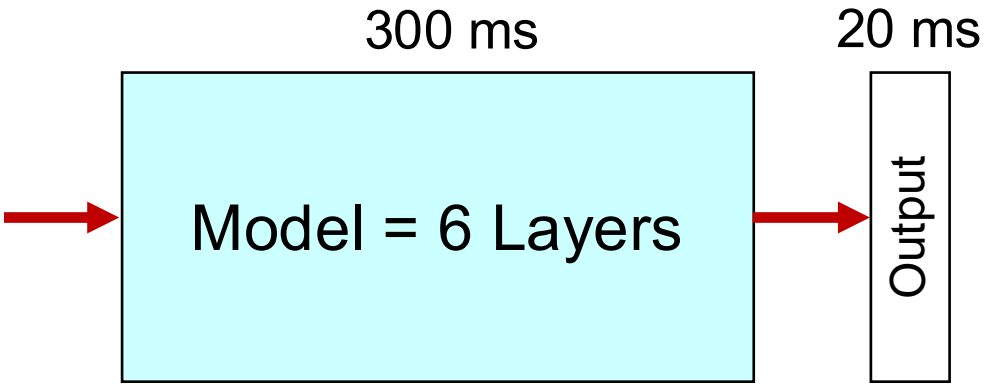
Traditional Issues of Pipelining

Question: increase overall latency

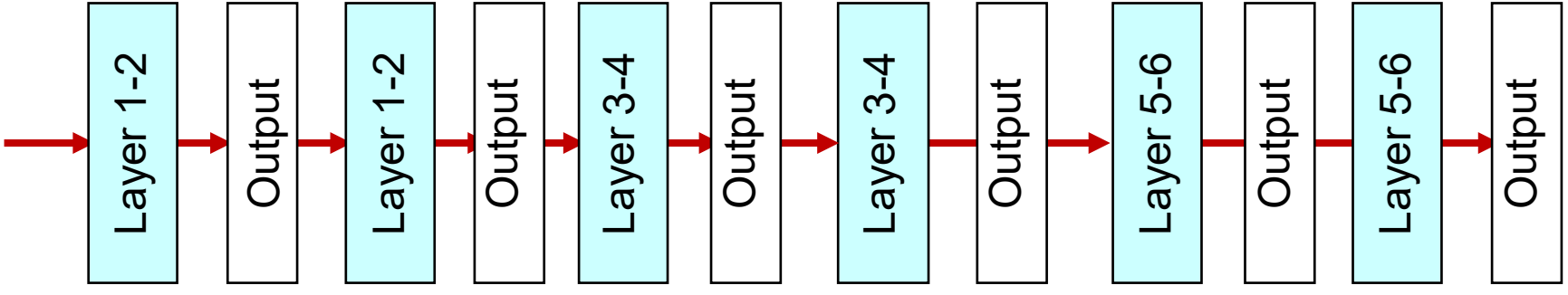


Traditional Issues of Pipelining

Question: increase overall latency



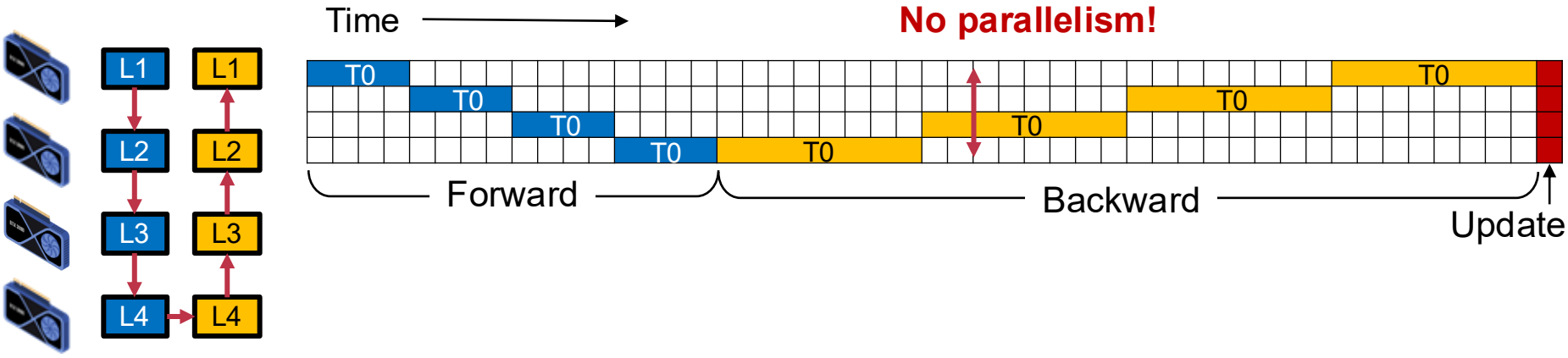
Latency = **320 ms**



Latency = **420 ms**

Communication Overhead

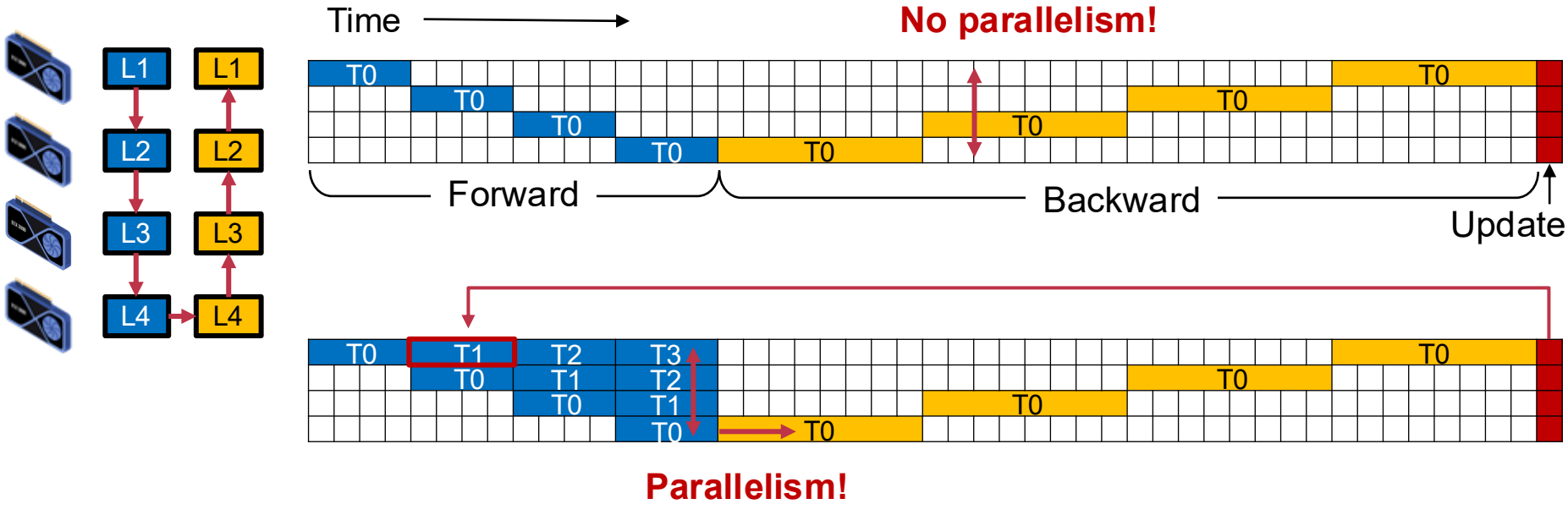
Pipeline Parallelism of Model Training (One-by-one)



Model Training is bi-directional

- A **forward pass** goes through the computation graph
- A **backward pass** uses state and intermediate data computed during the forward pass

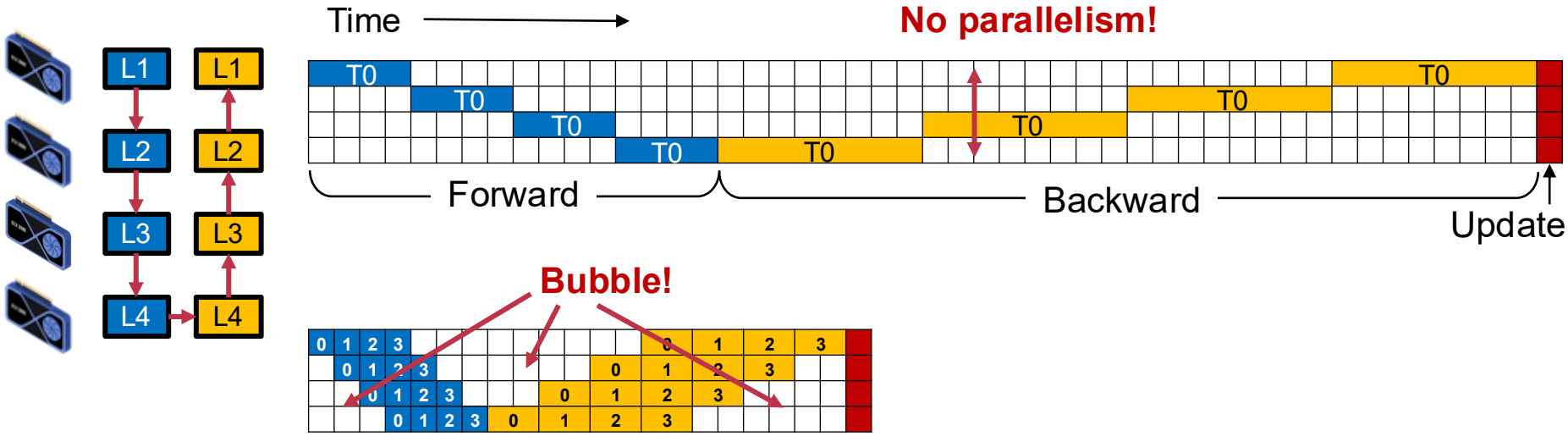
Pipeline Parallelism of Model Training (Multi-batching)



Pipelining by concurrently training multiple batches

- Process batches on **a stale model state**
- Delay **backward operations**

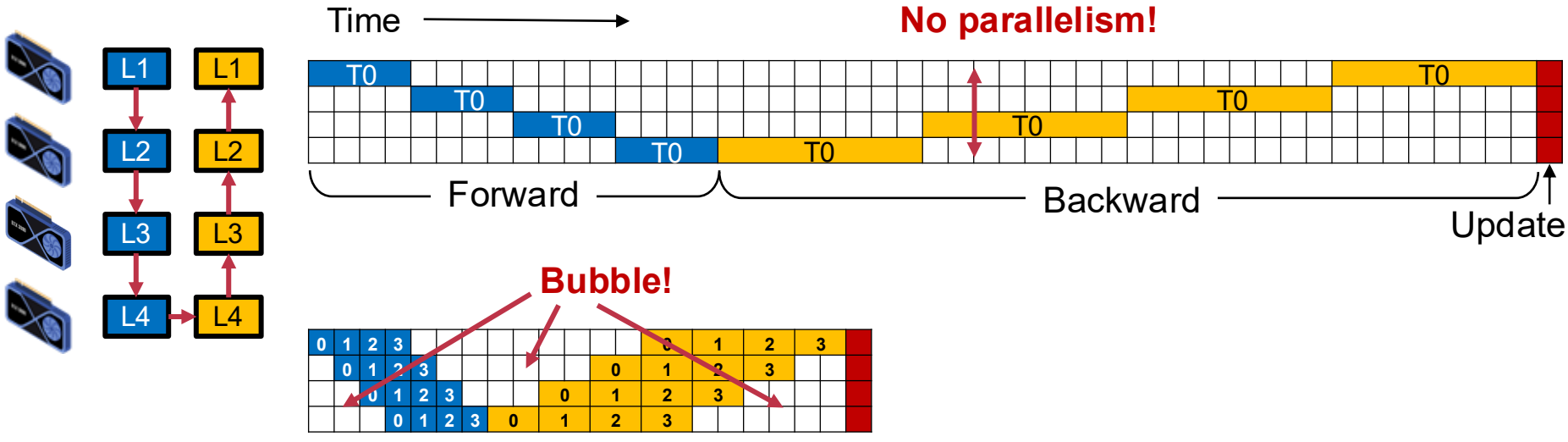
Pipeline Parallelism (Micro-batching)



Pipelining the batch by breaking the batch size into smaller pieces

- An easy way to implement
- Reduce the idle time (bubble) of the processes

Pipeline Parallelism (Micro-batching)



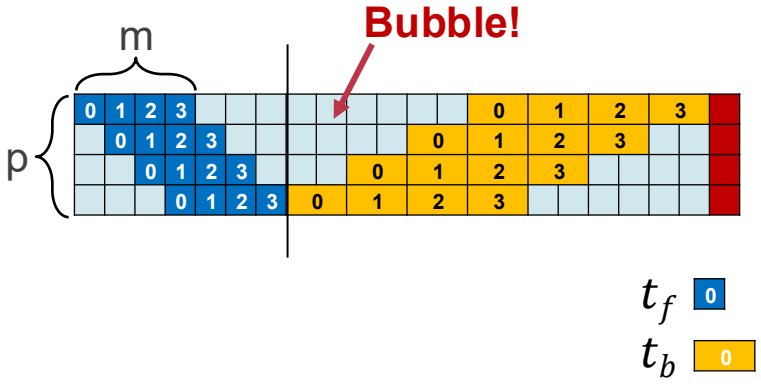
Question: to what extent can micro-batching improves performance?

- The **bubble size** depends on the **#stages** (several layers)
- The ratio bubble overhead depends on the overall **pipeline numbers**

Overhead of Pipeline Parallelism

Question: what is the bubble size ?

- p – 1 (p == #devices)



Question: what is the overhead?

- t_f : time of a forward of a micro-batch (m)
- t_b : time of a backward of a micro-batch (m)
- Ideal process time: $m * (t_f + tb)$
- Wasted time of bubble: $(p - 1) * (t_f + tb)$
- **Bubble time fraction: $\frac{p-1}{m}$**

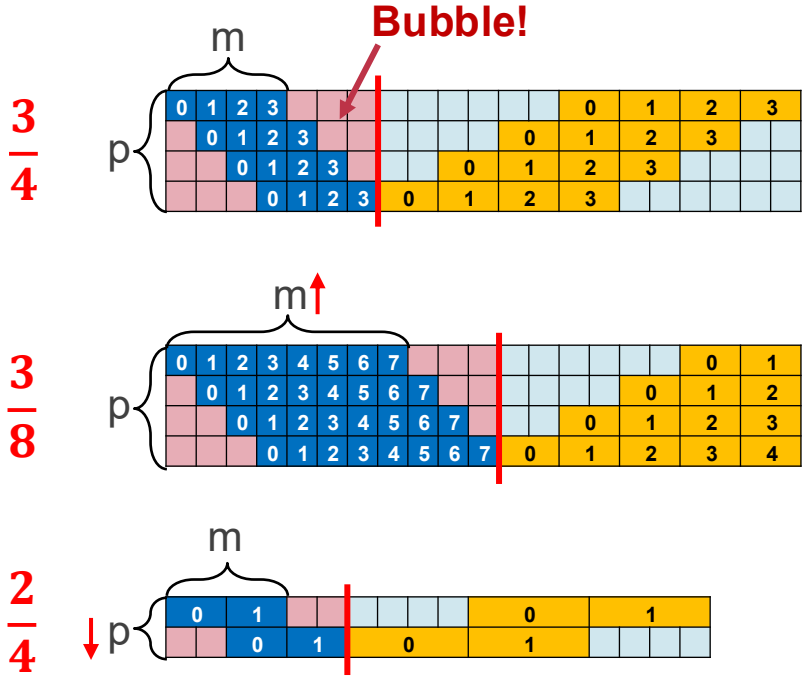
Overhead of Pipeline Parallelism

Question: what is the overhead of pipeline parallelism?

– Bubble time fraction: $\frac{p-1}{m}$

Optimization directions

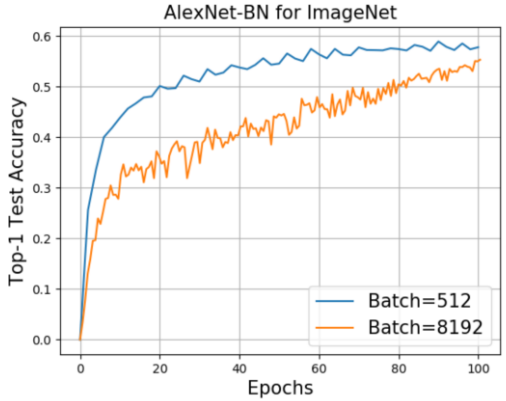
1. increase m (e.g., increase batch size)
2. reduce p (i.e., reduce #partitions)

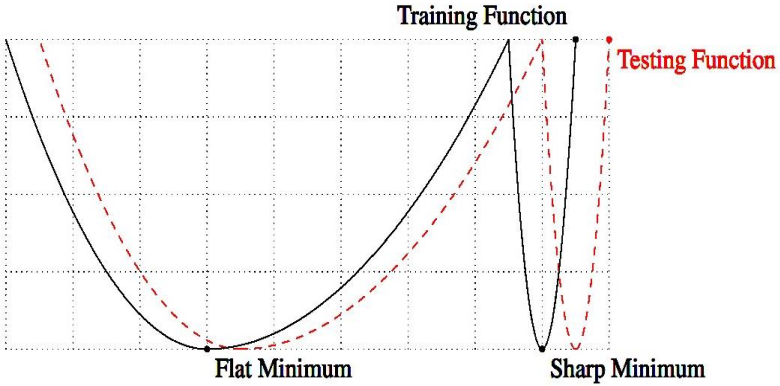


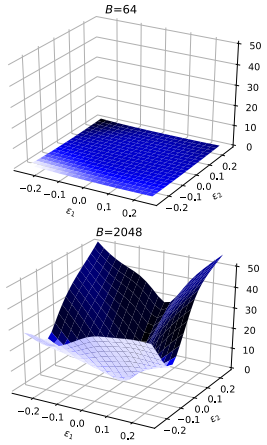
Problems with Large Batch Size

Larger batch leads to **sub-optimal generalization**

A common belief is that large-batch training gets attracted to **“sharp minimus”**







Key takeaway: improving system should consider application properties

Keskar et al., On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, ICLR’16.
Z. Yao, A. Gholami, Q. Lei, K. Keutzer, M. Mahoney. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries, NeurIPS’18.
Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with LARS." arXiv:1708.03888, 2018.

Problems with Large Batch Size

P#0. Decreased accuracy

P#1. Increased memory at each device

- Memory used for storing the activation memory

Though some optimizations exist

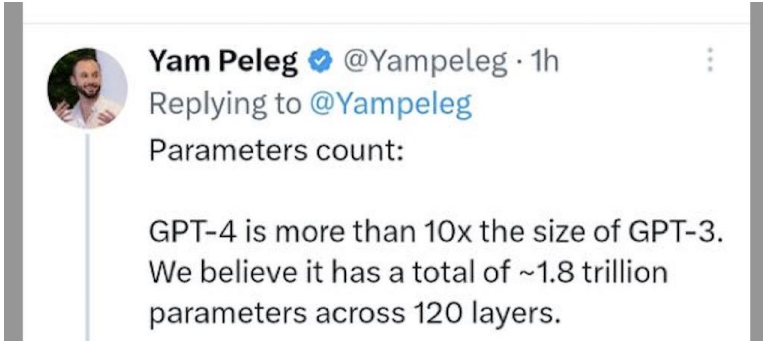
- For example, **Re-computation**, throw away the intermedia activations generated during the forward path, re-compute it during the backward path
  - **Cons**: trades performance for memory usage

Can We Reduce #Partitions (p)?

Typically, also challenging because AI models are really really large

Example: GPT-4

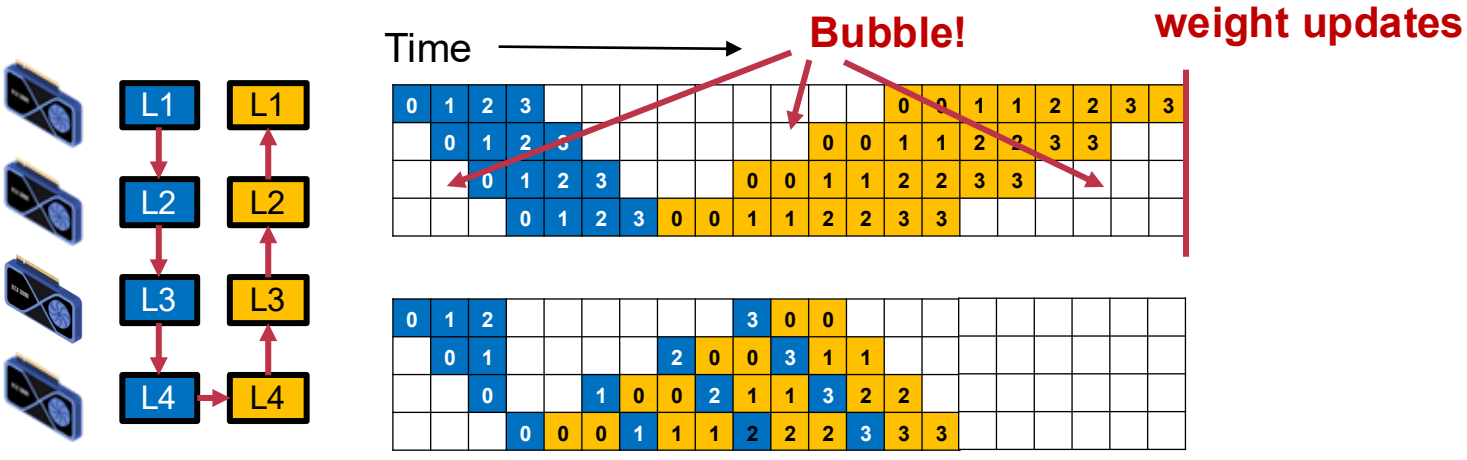
- Setup: 1.8 trillion parameters
- 120 layers
- Assuming 4B for each parameter, we have 60 GB per-layer



What are the memory capacity of GPU?

- 80GB for H100, 140GB for H200
- Note that we also need to store activations & input data (& optimization state)

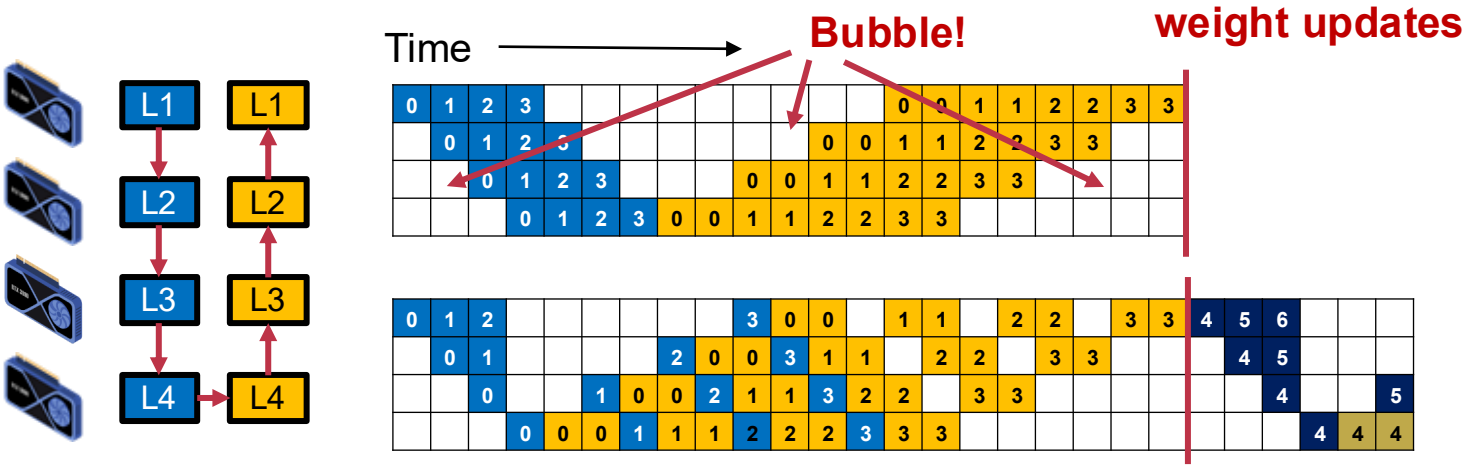
Advanced Pipeline Parallelism



Key question: the schedule of pipelined computation

- Dynamically choose which micro-batches or layers to execute at each step
- Impact the pipeline bubble size, accuracy, and memory consumption

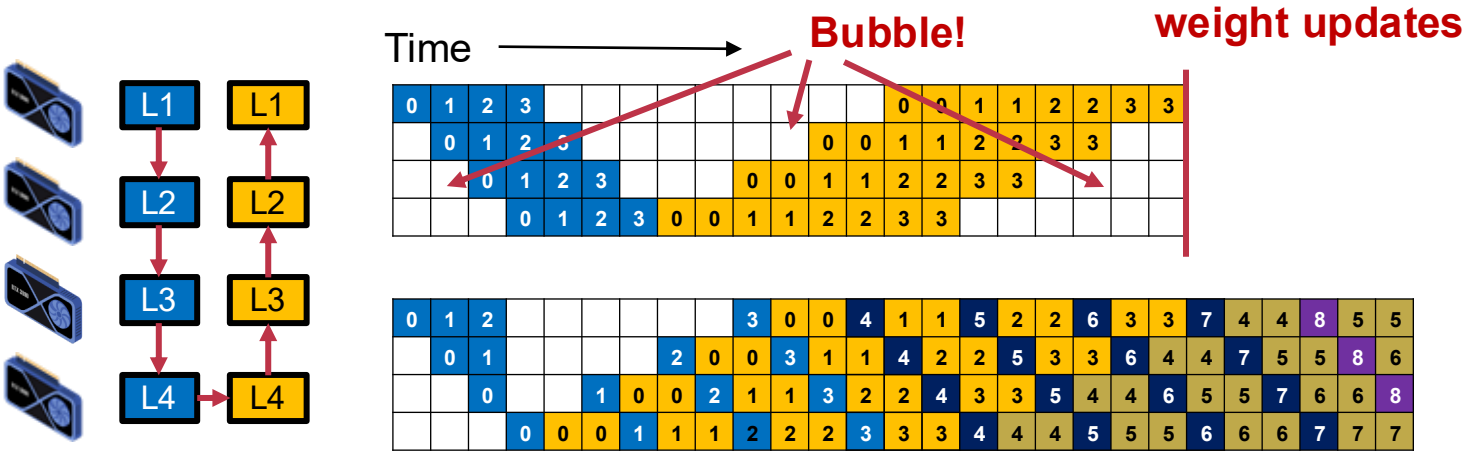
Advanced Pipeline Parallelism



Naïve pipelining (GPipe)

- Apply weight updates every m minibatches
- Use synchronous weight updates

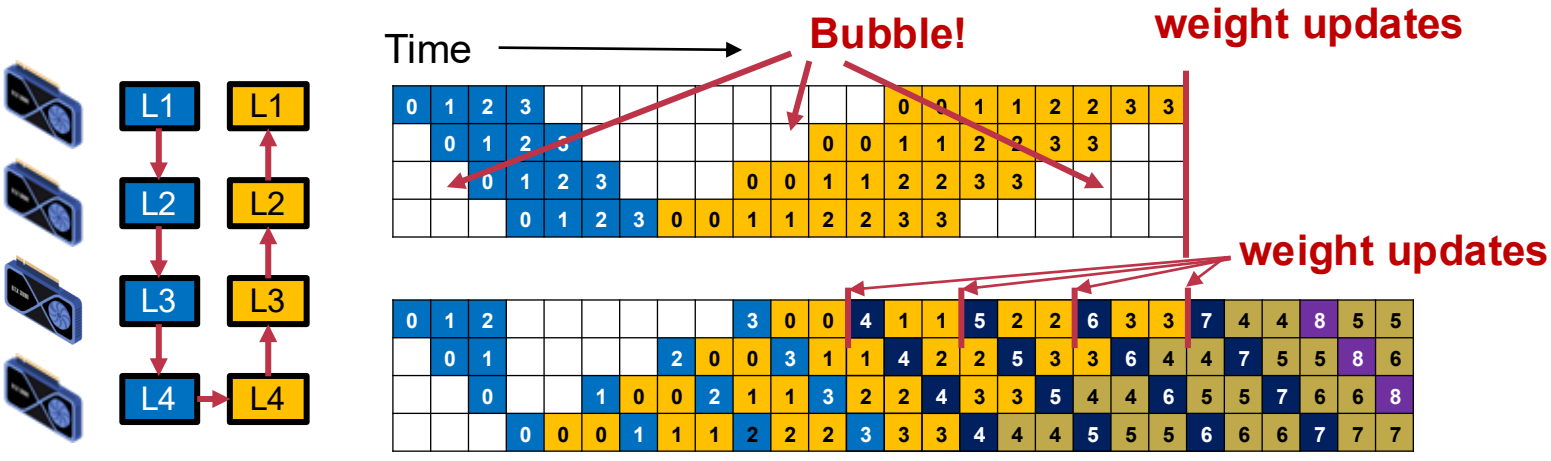
Advanced Pipeline Parallelism



1F1B pipelining

- Keep hardware well-utilized (similar to data parallelism)
- **No pipeline stalls** in steady state

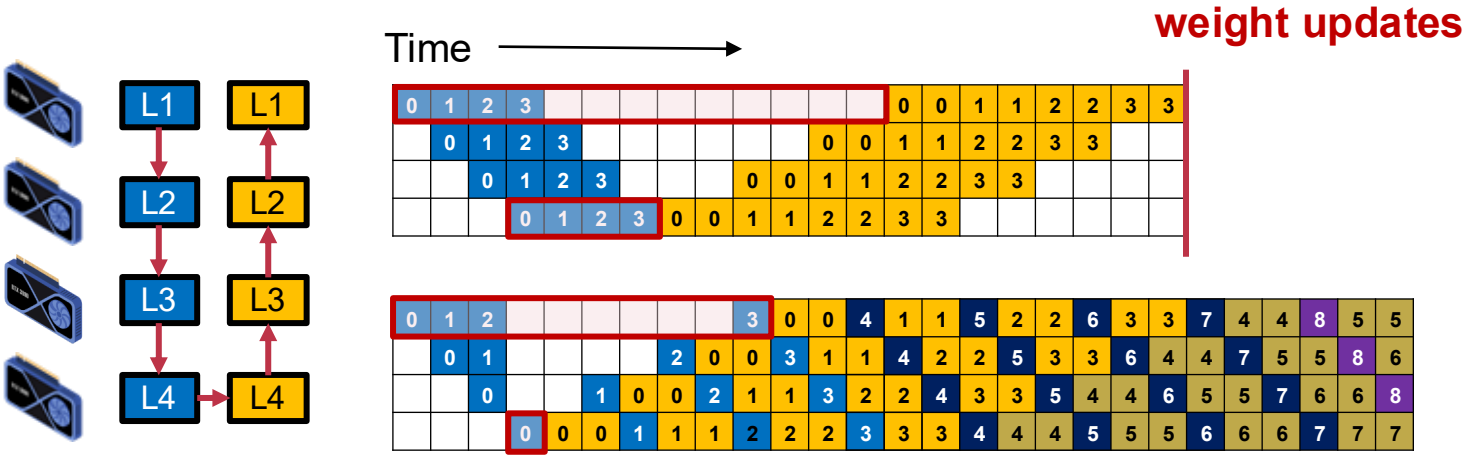
Advanced Pipeline Parallelism



Asynchronous weight updates

- **Each backward pass** results in weight updates; the next forward pass uses the latest version of weights available
- Forward pass will not see updates from **incomplete in-flight** mini-batches

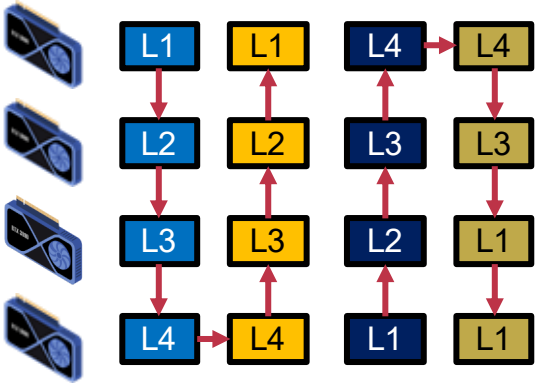
Advanced Pipeline Parallelism



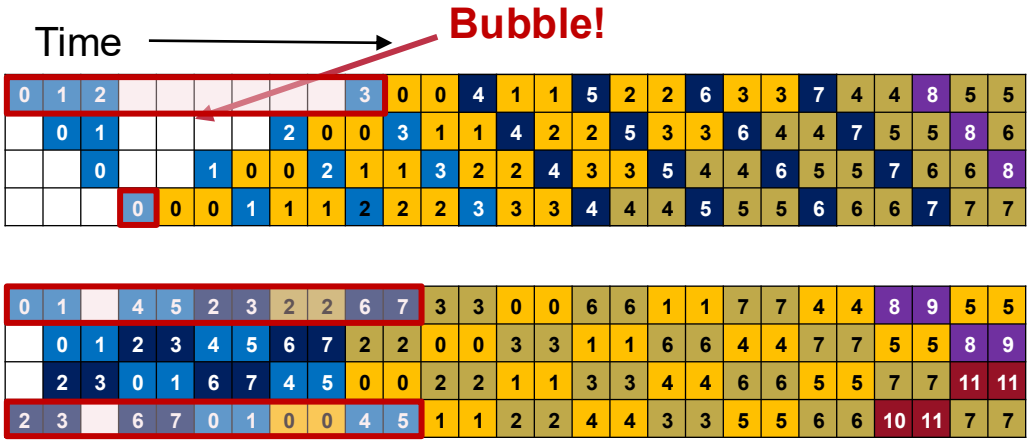
Imbalance memory usage in F1B1

- **Memory usage** on each device are different (activation, output)
- Optimizations: offloading (host memory), re-computation

Advanced Pipeline Parallelism



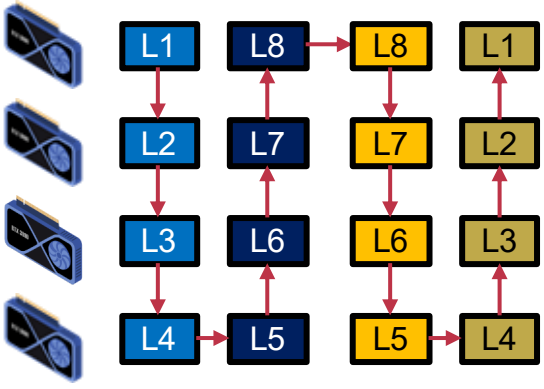
Bi-direction pipeline



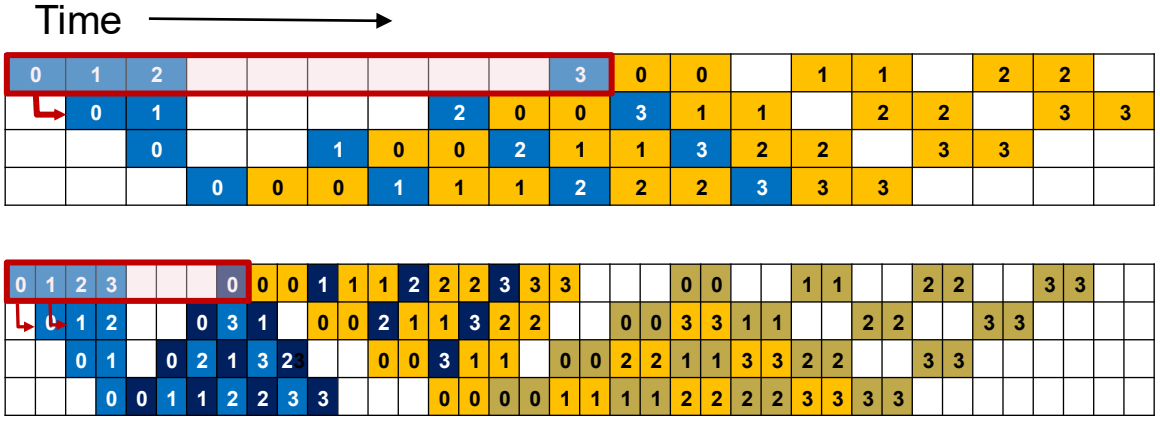
Chimera: bi-direction pipeline

- Reduce **startup bubbles**
- Higher peak memory, but more **balance**

Advanced Pipeline Parallelism



Wave-like pipeline



Hanayo: wave-like pipeline

- Less **startup bubbles**, fine-grained **communication**
- Less **peak memory** (earlier memory release)

Summary: Model Parallelism

Two forms: Pipeline Parallelism + Tensor Parallelism

Pros	Cons
Memory Efficiency	Difficult to Implement

Tensor parallelism: partition the parameters of a layer (or a set of layers)

- Pros: better support for large models
- Cons: high communication cost

Pipeline parallelism: partition the computation graph by layers

- Pros: reduced communication
- Cons: bubbles

SYNCING GREDIANTS

Parallelization Opportunities

Data Parallelism: *Distribute the processing of data to multiple PEs.*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

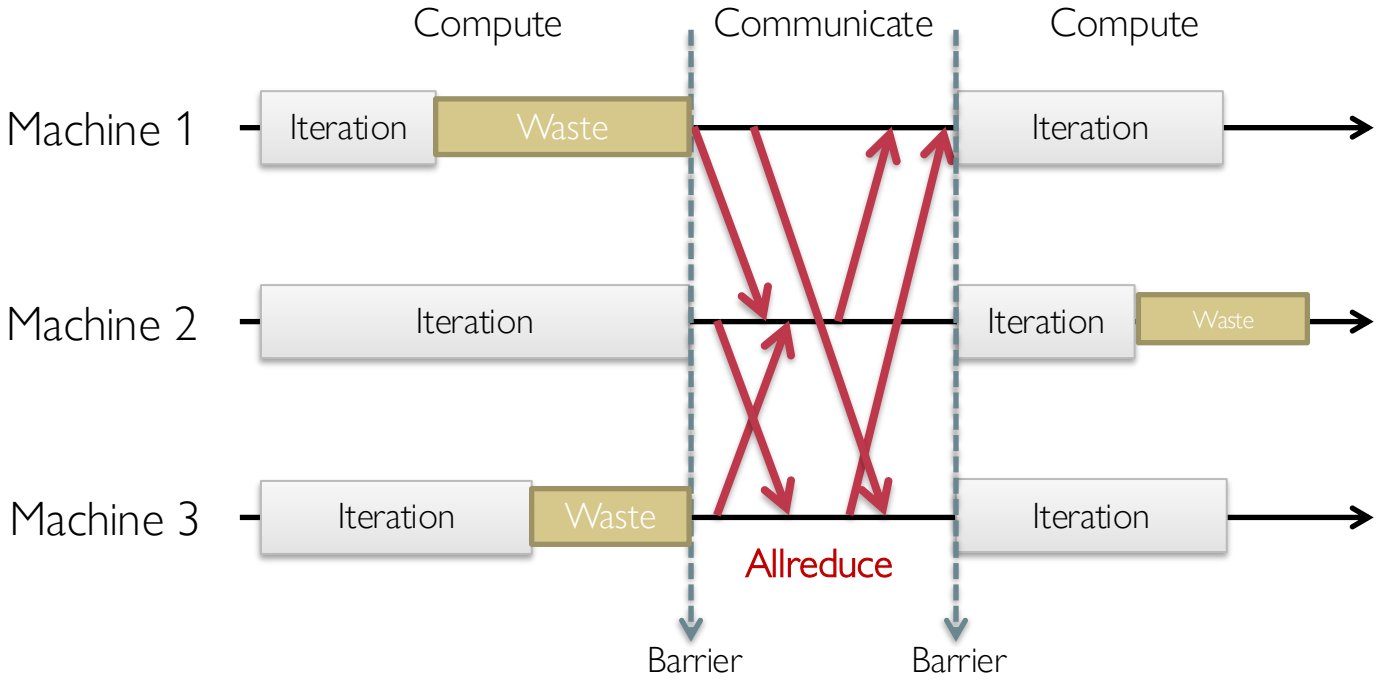
$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Recall: BSP for data parallel execution

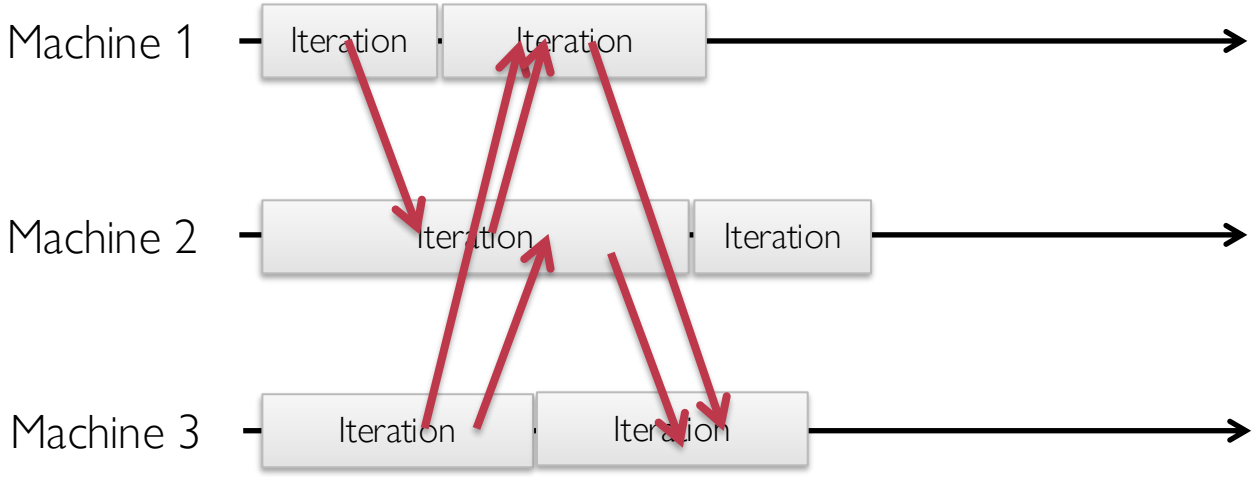
Problem: may have idle time at each process



Asynchronous Execution

Key idea: each process does not wait for others

– Update the gradients upon receive a messages, no iteration barrier



Async vs. Sync Execution

Common: nearly all training adopts a Sync. approach

Async

- Pros: efficiency
- Cons: trade accuracy for performance

Sync

- Pros: preserve the SGD training property
- Cons: poor performance

Which is better?

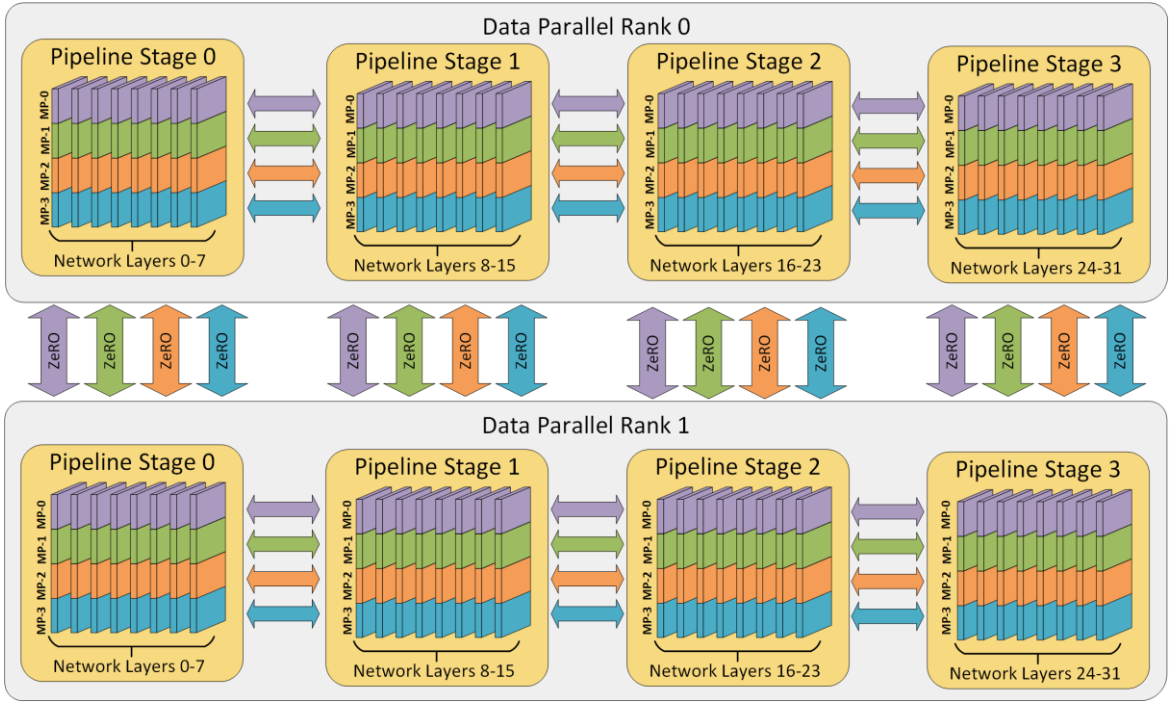
- Hard to tell. But, as a system designer, preserving the algorithms property is **very important**. Any design that alter the training property should be justified

PARALLELISM COMBINATION

Credit: “System for Machine Learning”, CS229s, Stanford

Put It All Together

It is practical/common to use DP, TP, and PP together (3D or PTD parallelism) to scale to thousands of GPUs

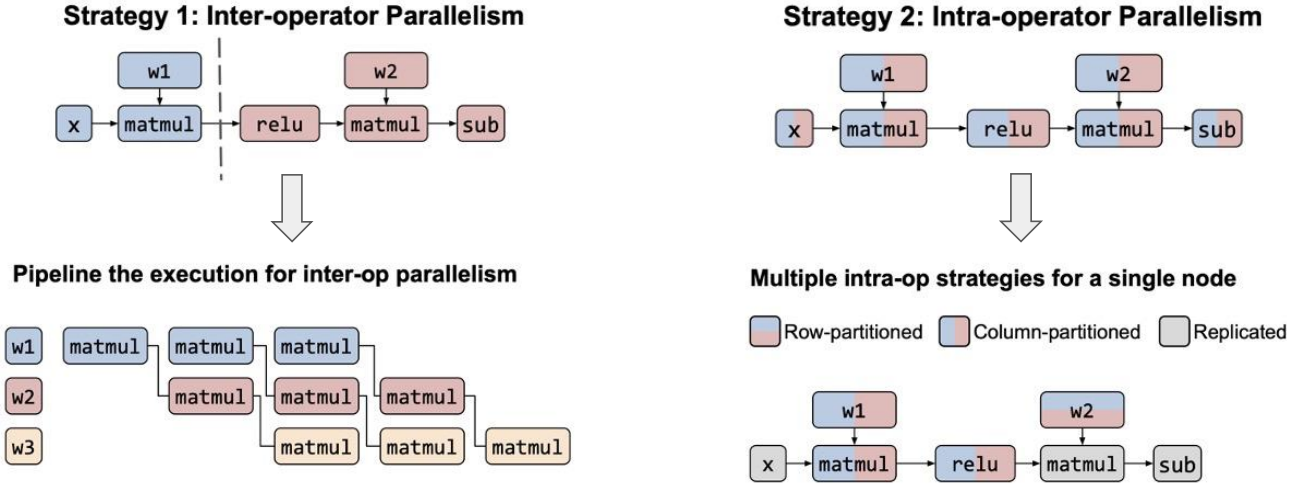


Credit: “System for Machine Learning”, CS229s, Stanford

Automatic Parallelization with Alpa

Alpa: automatically select and execute the optimal parallelism strategy for a given model and cluster

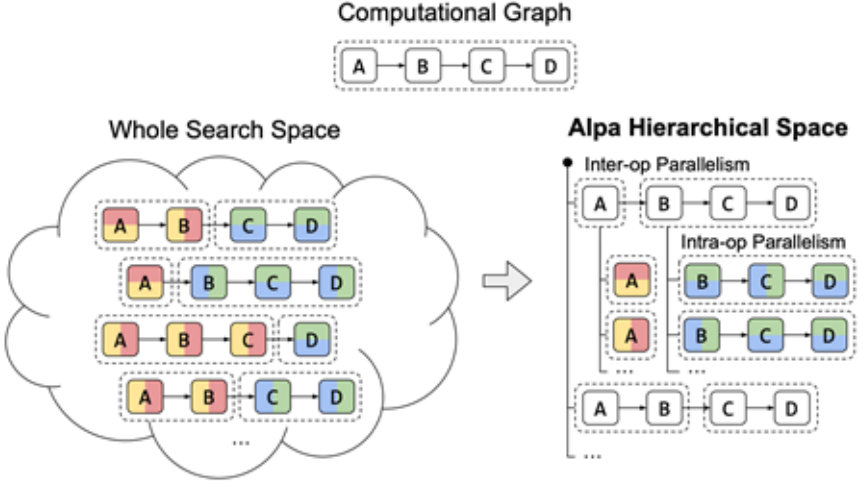
– Consider both **inter-operator parallelism** (i.e. pipeline parallelism) and **intra-operator parallelism** (i.e. data and tensor model parallelism)

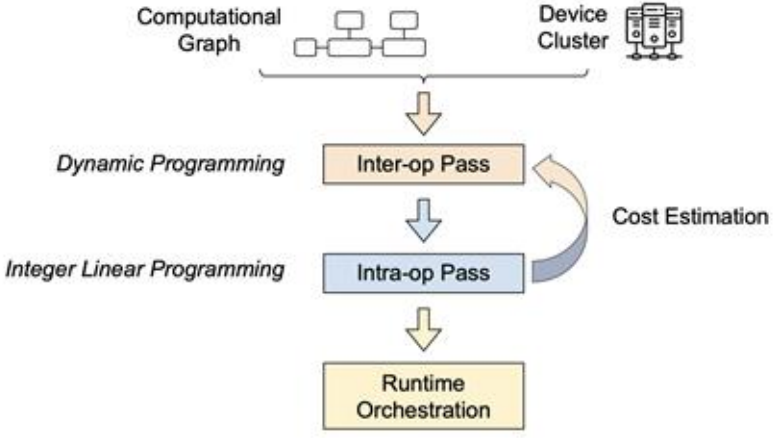


Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022.

Automatic Parallelization with Alpa

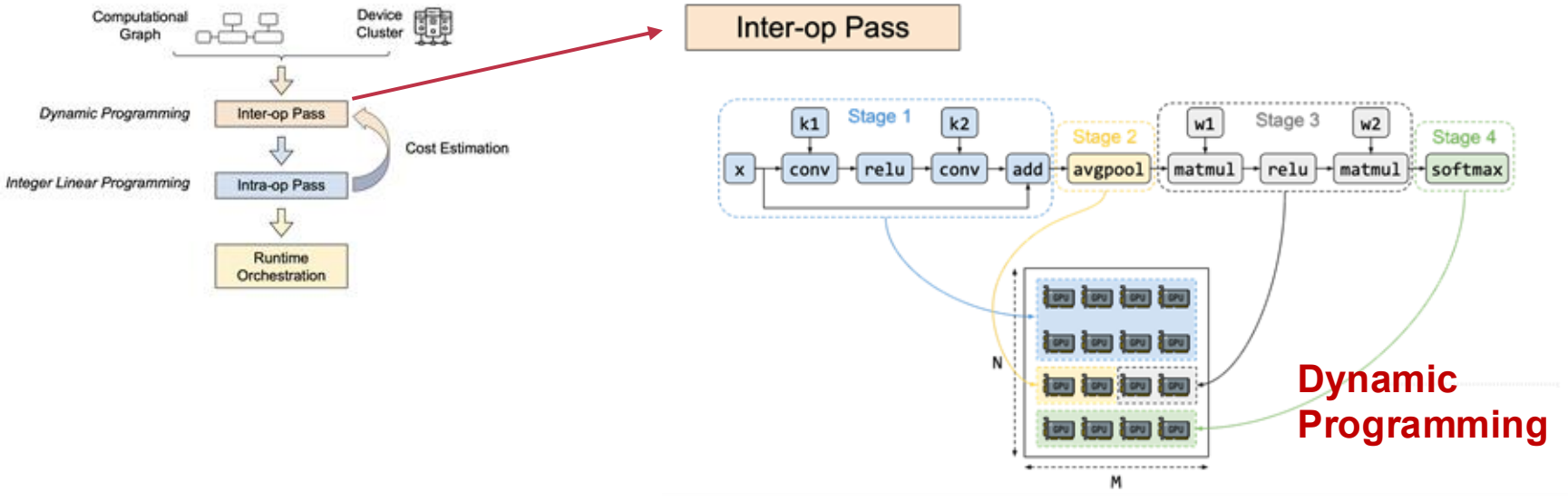
1. Automatically **search over** the space of possible parallelism strategies for a given graph
2. Choose the **optimal combination** of inter-op and intra-op parallelism





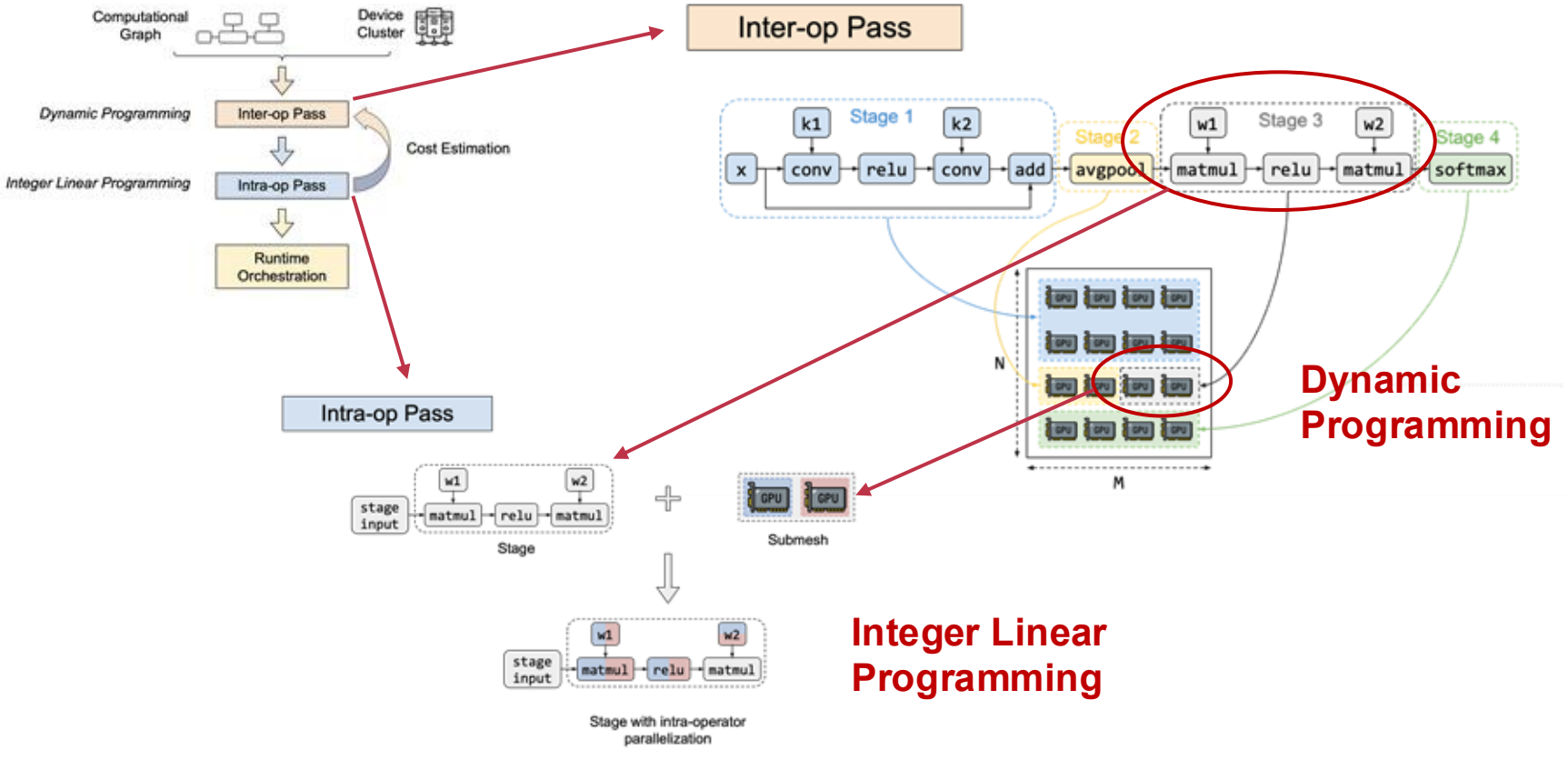
Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022.

Automatic Parallelization with Alpa



Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022.

Automatic Parallelization with Alpa



Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. OSDI 2022.

Summary

We need to **parallelize data and models** to train and serve models at scale

The most widely used forms of parallelism are **Data Parallelism**, **Tensor Parallelism**, and **Pipeline Parallelism**

Each of these parallelization strategies has **trade-offs** depending on the amount of **computation efficiency**, **communication overhead** and **memory usage**

A system for **automating** the decision of which parallelization strategy or strategies to use is helpful and important

Credit: “System for Machine Learning”, CS229s, Stanford