

Distributed Model Training

Yiying Zhang

Lecture 13

Distributed Training

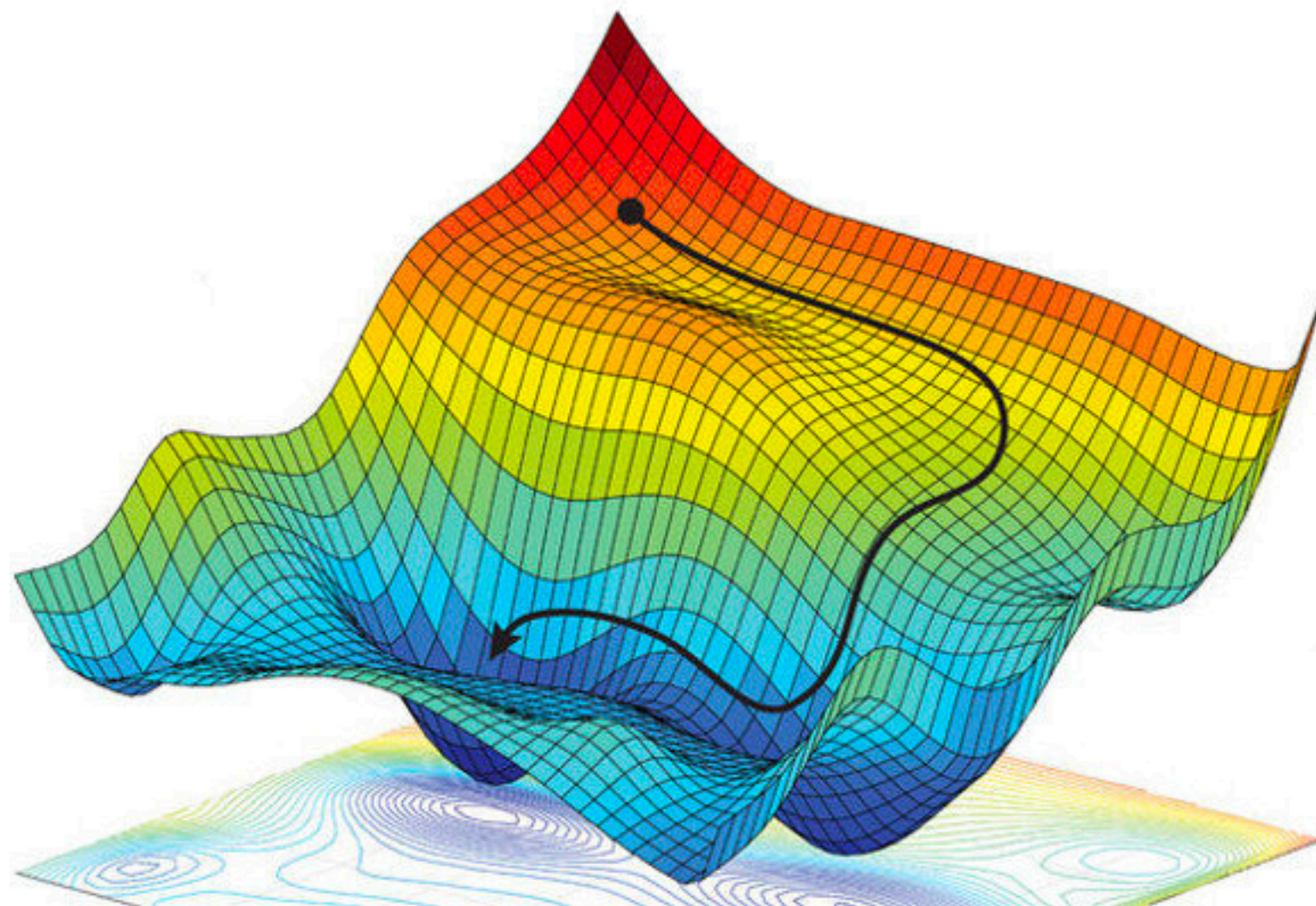
Song Han

songhan@mit.edu

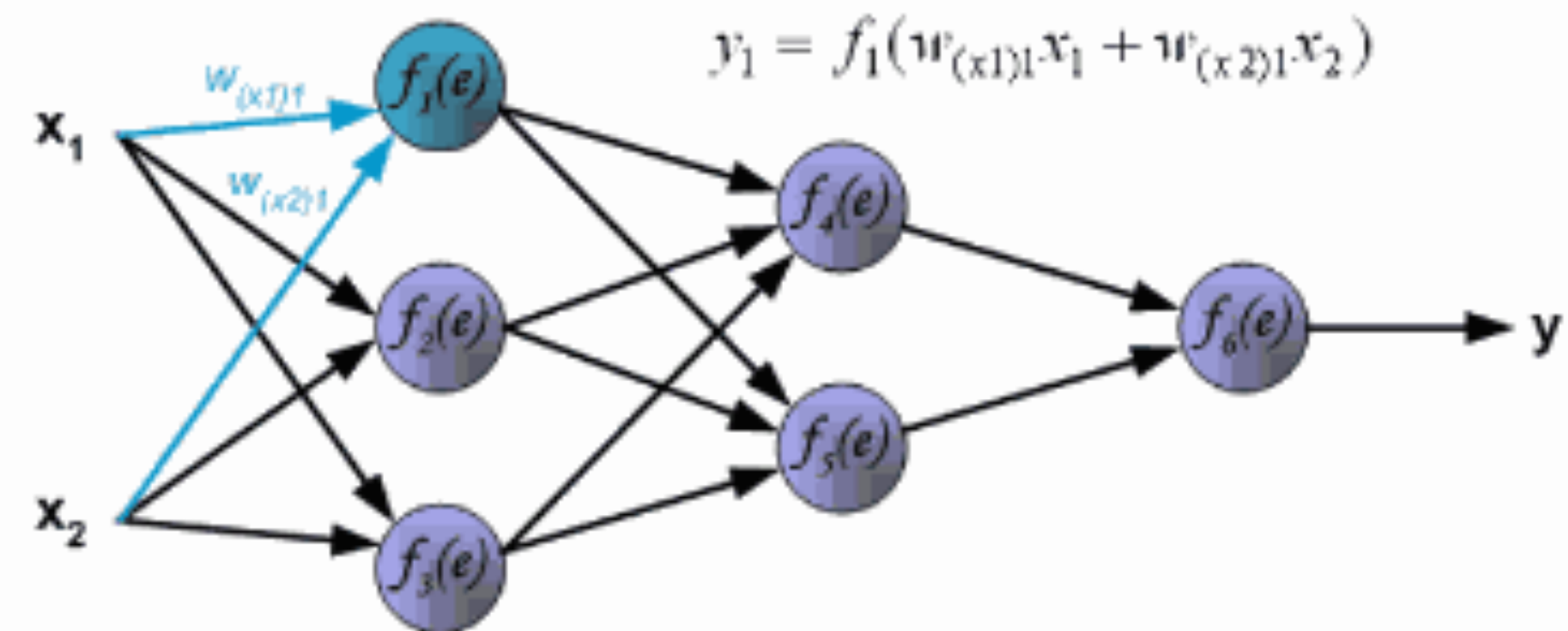


Neural Network Training

- Feedforward and back propagation gradient descent

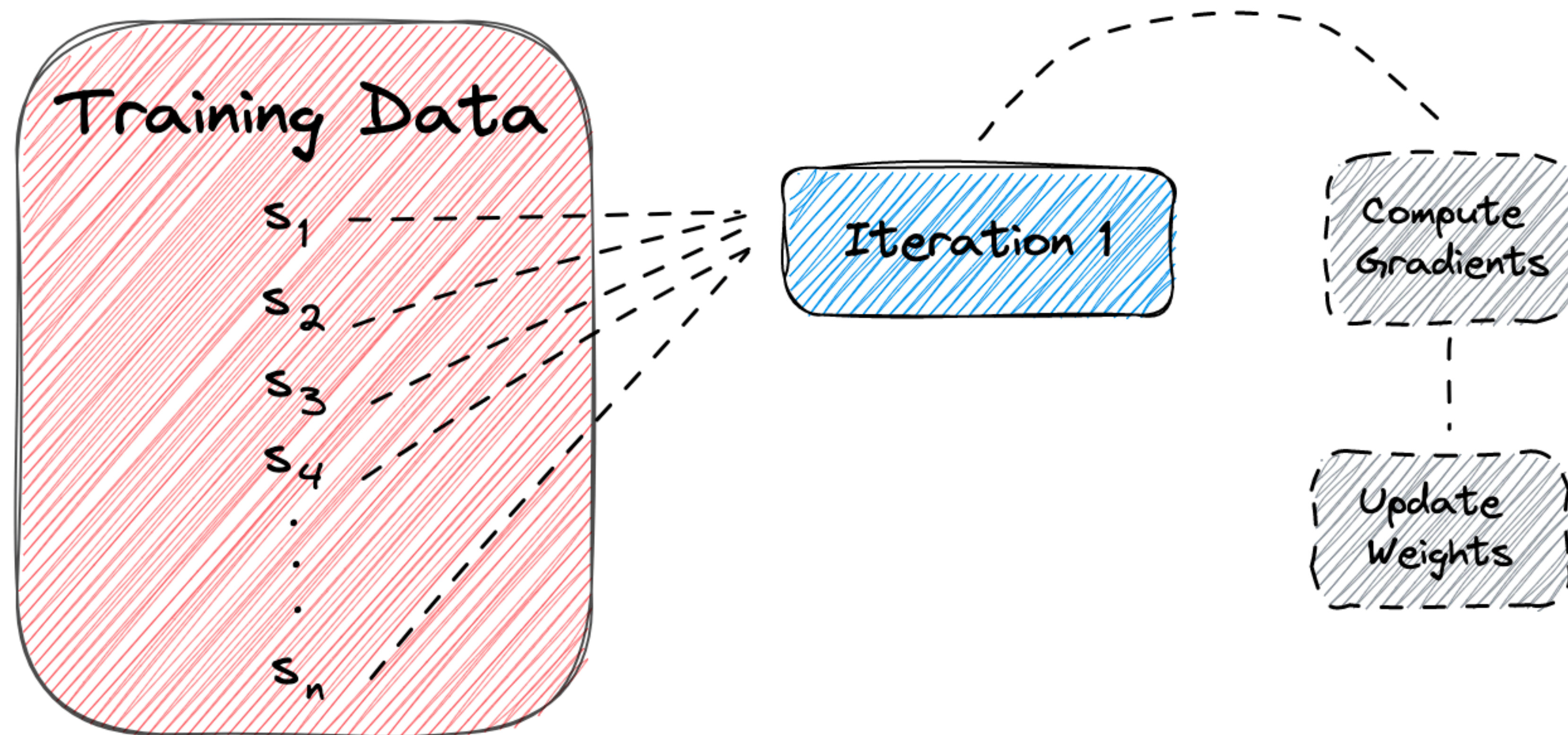


source: <https://arxiv.org/abs/1805.04829>



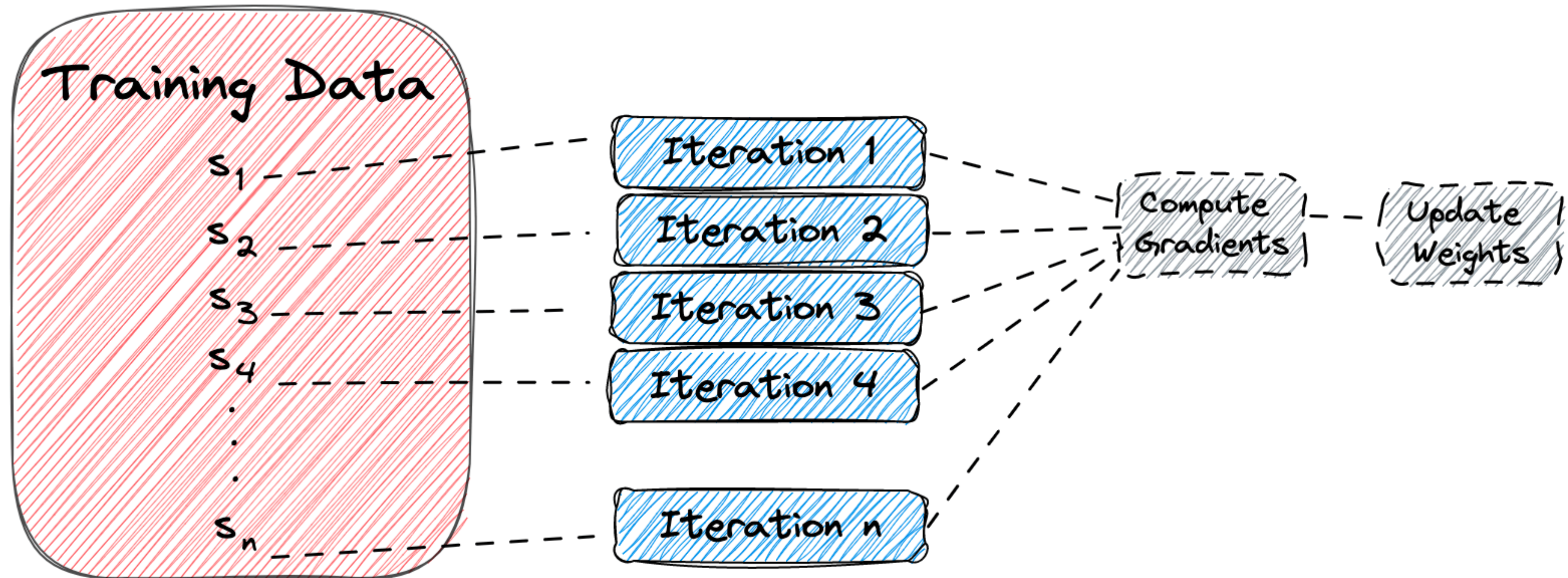
source: <https://www.programmersought.com/article/45312377380/>

Batch Gradient Descent



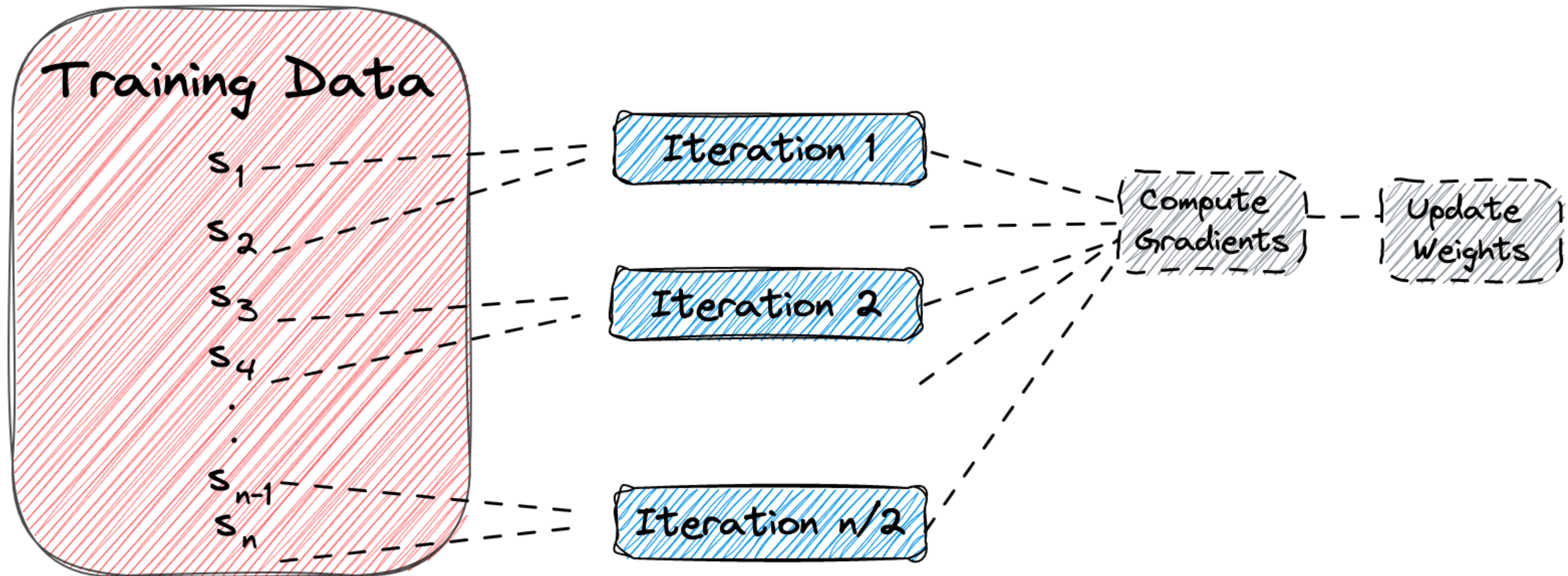
source: <https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch>

Stochastic Gradient Descent



source: <https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch>

Minibatch-Based SGD



source: <https://www.baeldung.com/cs/epoch-vs-batch-vs-mini-batch>

Model Training Cost

Model	Training end	Chip type	TFLOP/s (max)	Chip count	Wall clock time (days)	Total time (hours)	Total time (years)	Retail cost (\$US)
GPT-3	Apr/2020	V100	130	10,000	15 days	3,552,000	405 years	\$9M
Llama 1	Jan/2023	A100	312	2,048	21 days	1,032,192	118 years	\$4M
Llama 2	Jun/2023	A100	312	2,048	35 days	1,720,320	196 years	\$7M
Titan	Apr/2023	A100	312	13,760	48 days	11,558,400	1,319 years	\$45M
GPT-4	Aug/2022	A100	312	25,000	95 days	57,000,000	6,503 years	\$224M
Gemini	Nov/2023	TPUv4	275	57,000	100 days	136,800,000	15,606 years	\$440M
Llama 3 70B	Apr/2024	H100	989	24,576	11 days	6,300,000	719 years	\$7M
Llama 3 405B	Apr/2024	H100	989	24,576	50 days	29,491,200	3,364 years	\$125M
GPT-5	Mar/2024	H100	989	50,000	120 days	144,000,000	16,428 years	\$612M
Grok 2	Jun/2024	H100	989	20,000	50 days	57,600,000	6,571 years	\$245M
Grok 3	Dec/2024	H100	989	100,000	50 days	288,000,000	32,855 years	\$1.2B

Source data from [life architect.ai](#)

Distributed Training is Necessary

- Developers / Researchers' time **are more valuable** than hardware .
- If a training takes **10 GPU days**
 - Parallelize with distributed training
 - 1024 GPUs can finish in 14 minutes (ideally)!
- The develop and research cycle will be greatly boosted

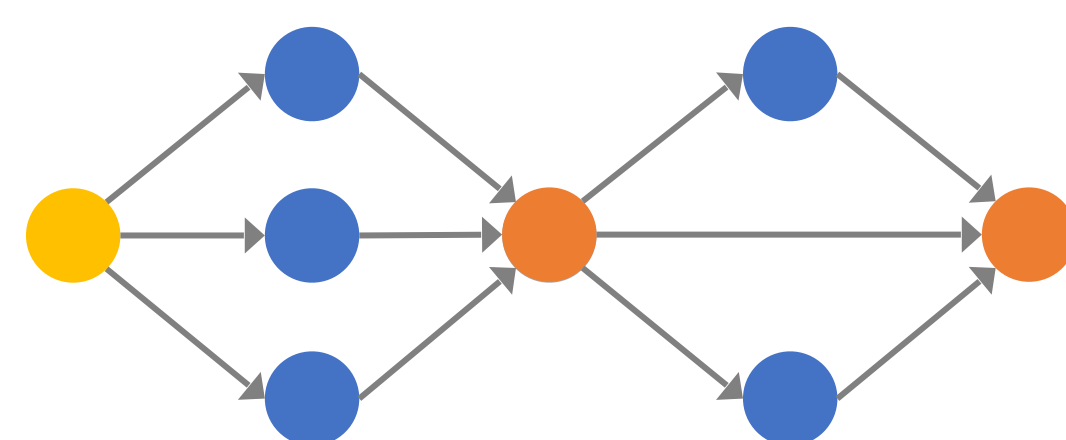
Let's see a use case of distributed training!

Parallelism in Distributed Training

- **Data Parallelism**
- Model Parallelism
- Compare the Advantages and Disadvantages of Two Parallelism

Introduction to Distributed Training

Data Parallelism

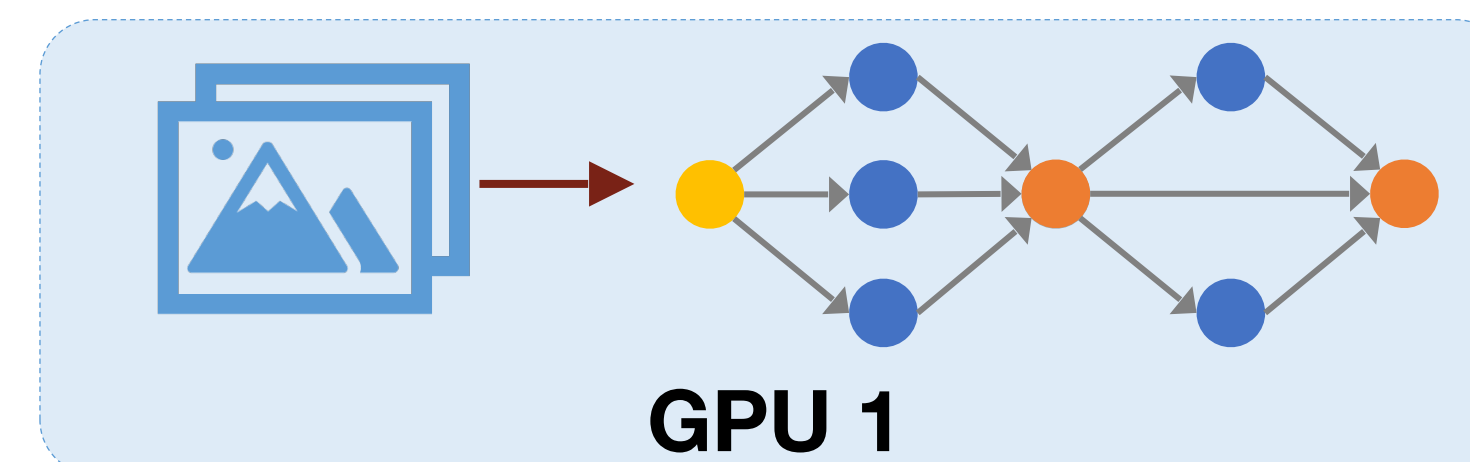


ML Model

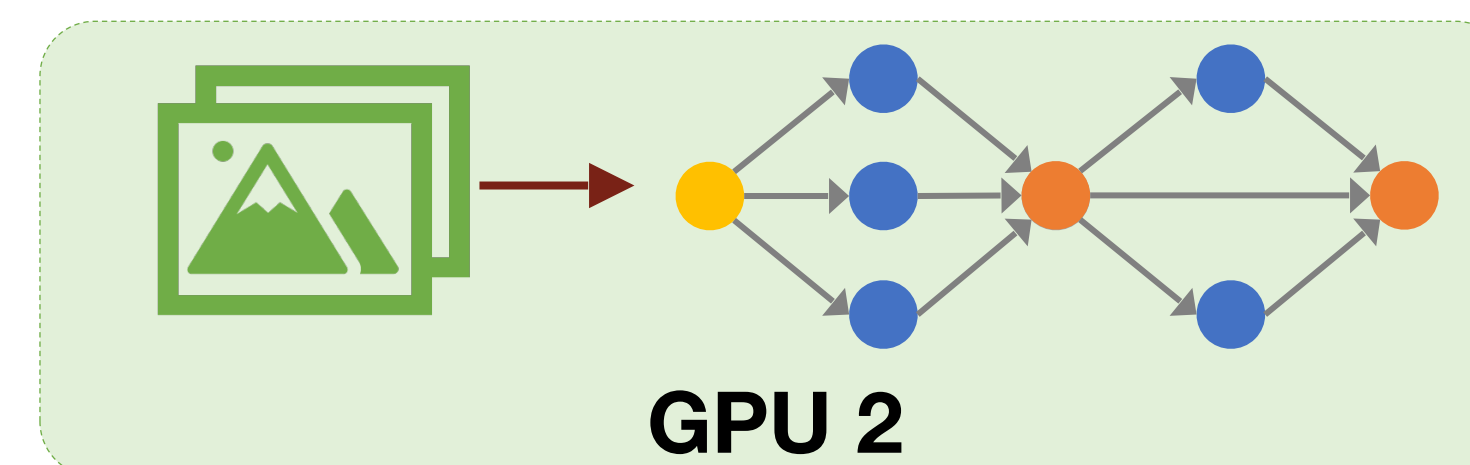


Training Dataset

Data Parallelism

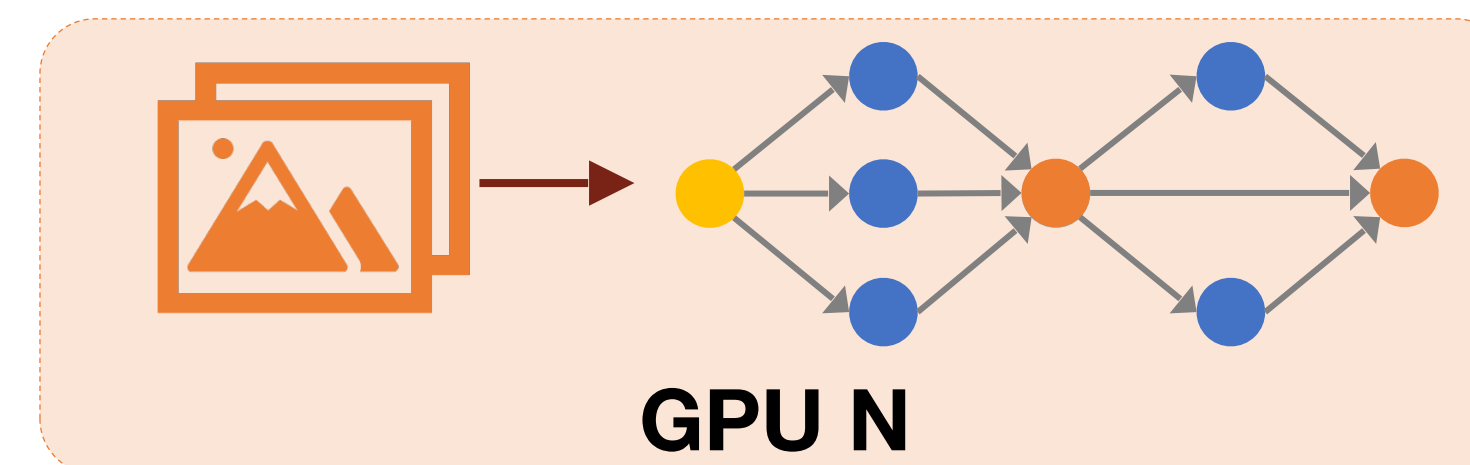


GPU 1



GPU 2

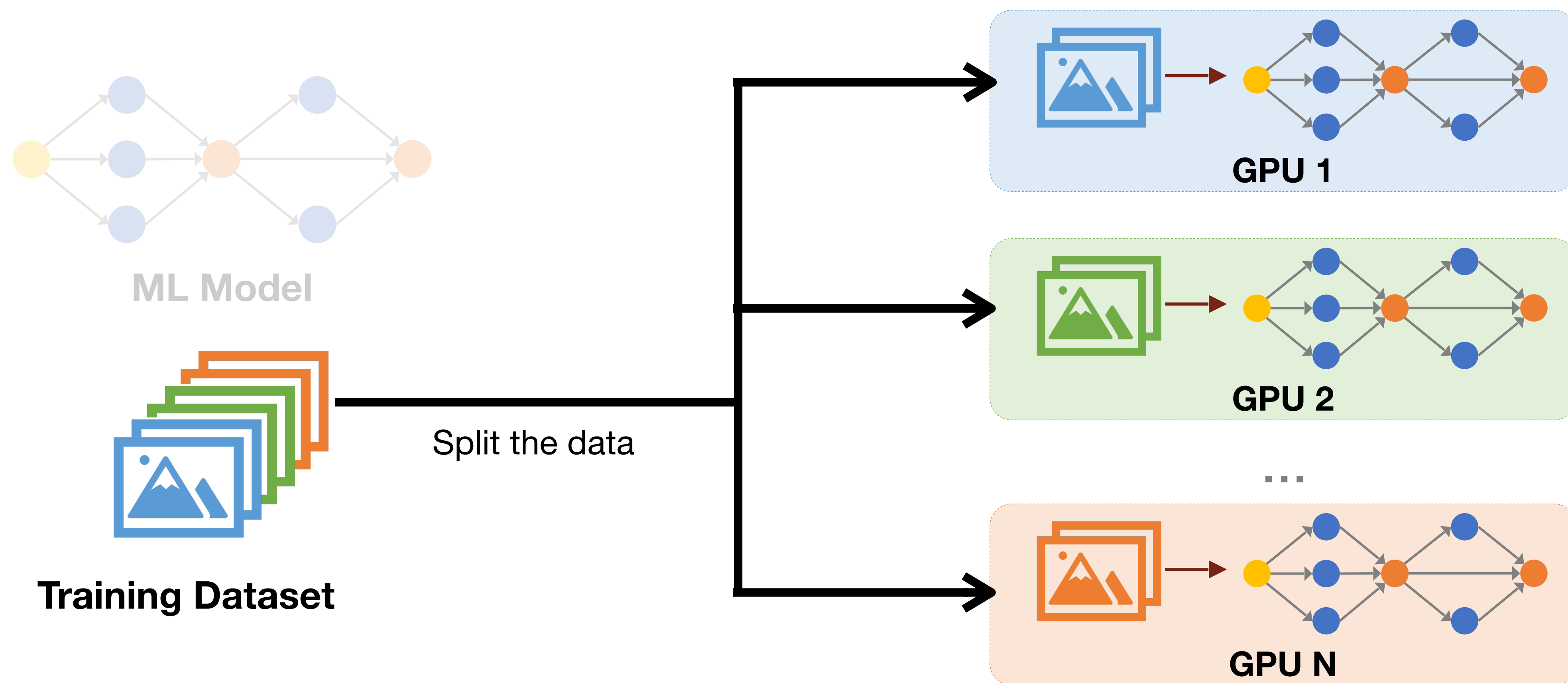
...



GPU N

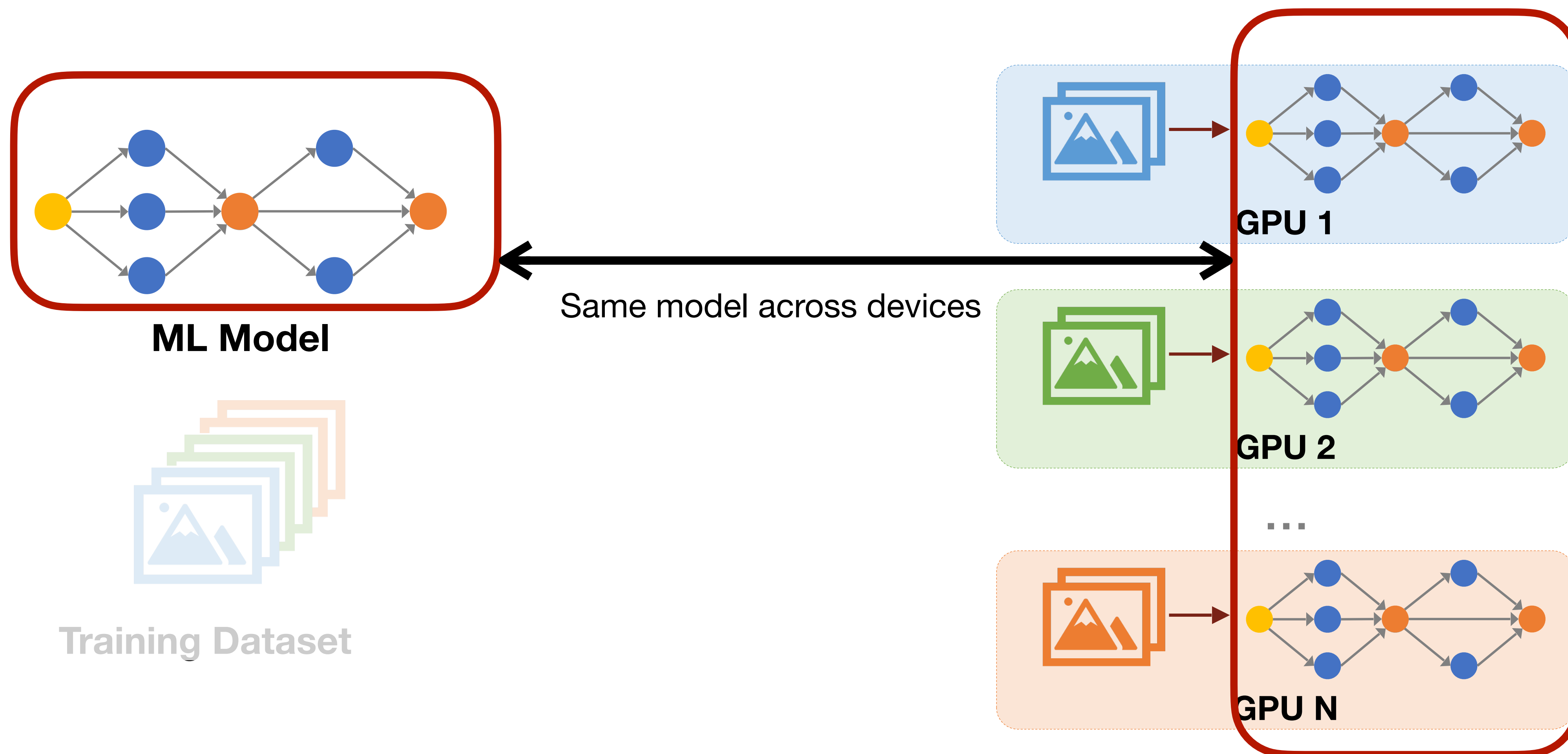
Introduction to Distributed Training

Data Parallelism



Introduction to Distributed Training

Data Parallelism

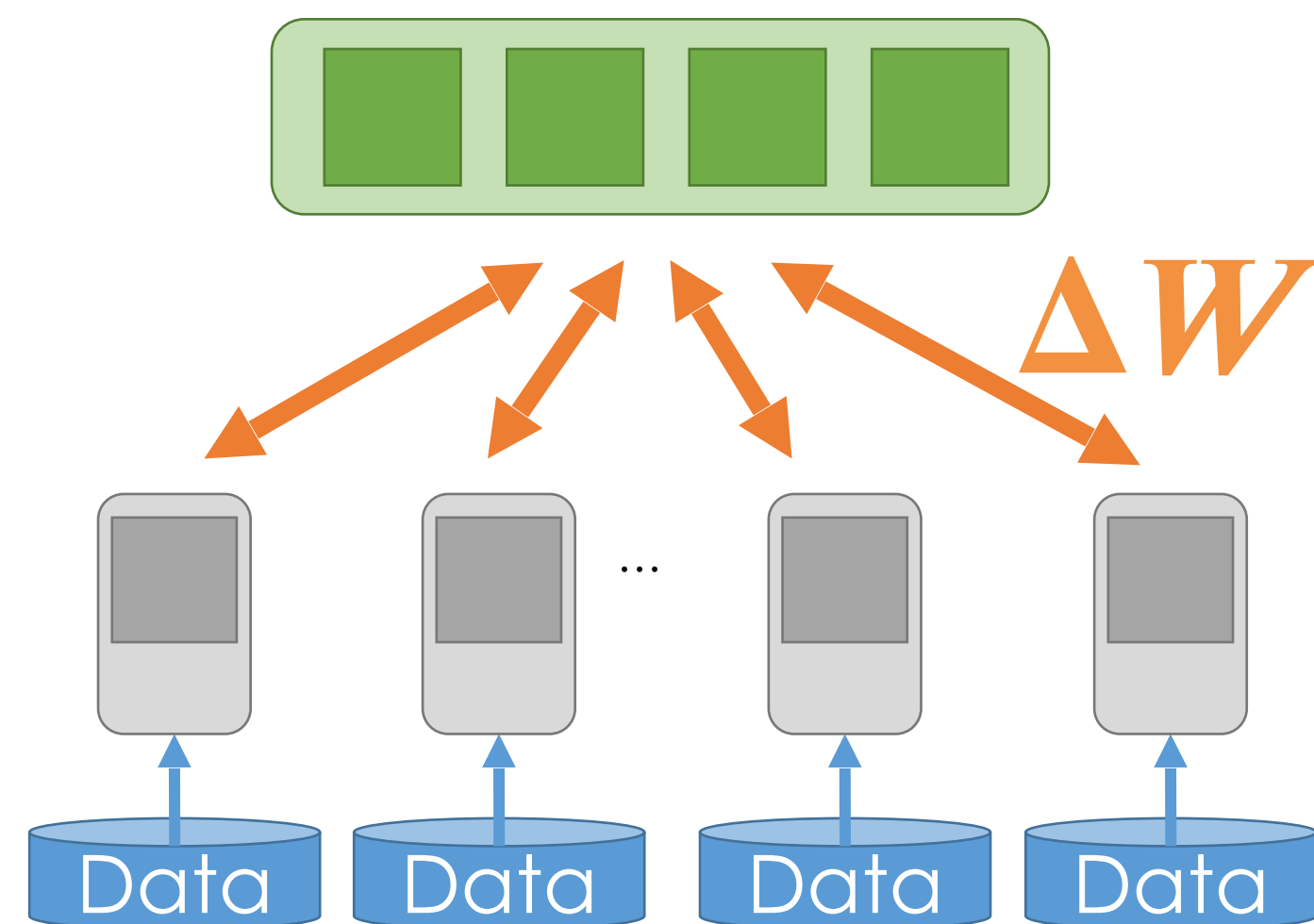


Dive into Data Parallelism

Scaling Distributed Machine Learning with the Parameter Server

Parameter Server

The central controller of the whole training process



Worker nodes

The hardware accelerators and dataset storage.

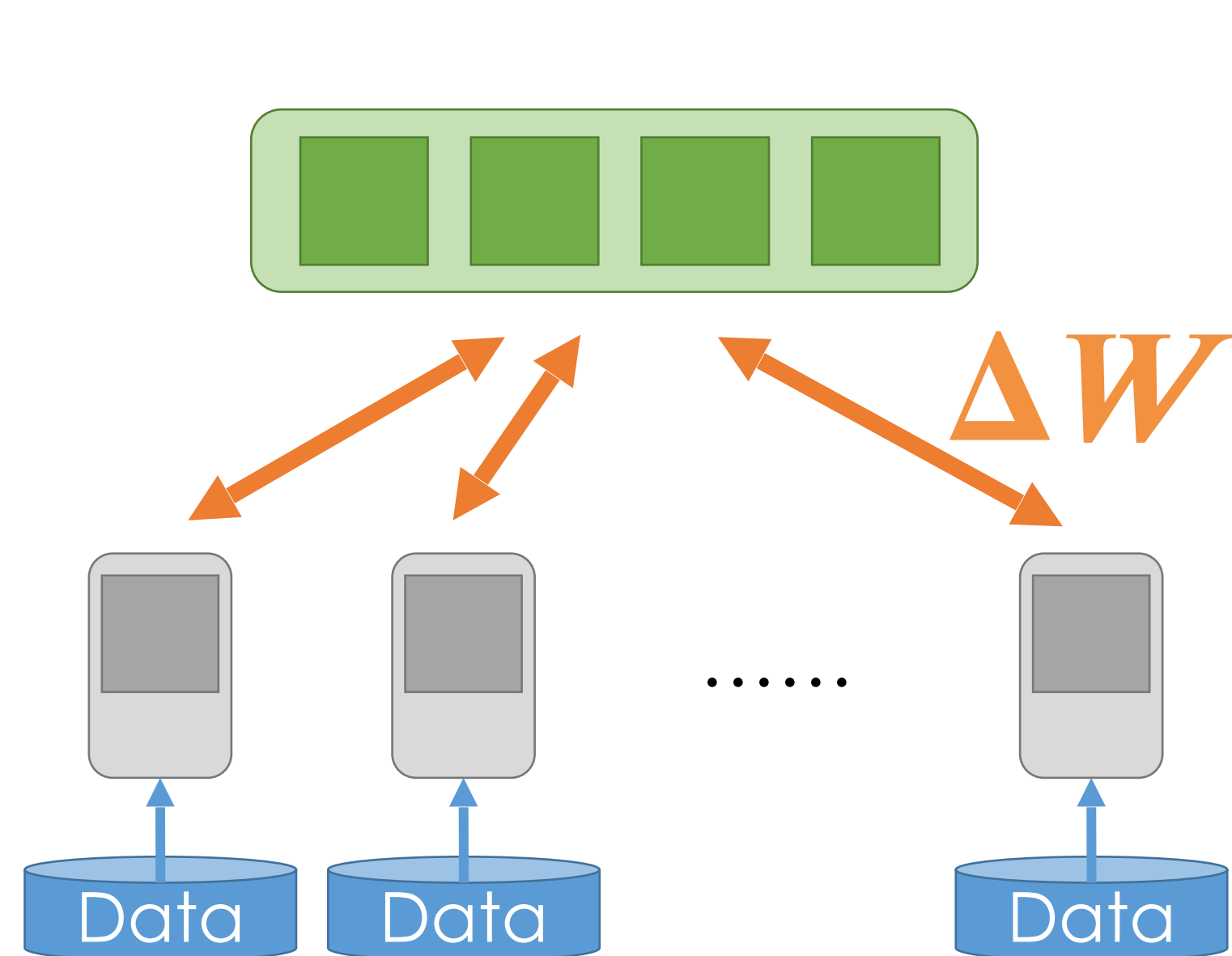
Two different roles in framework:

- **Parameter Server**: receive gradients from workers and send back the aggregated results
- **Workers**: compute gradients using splitted dataset and send to parameter server

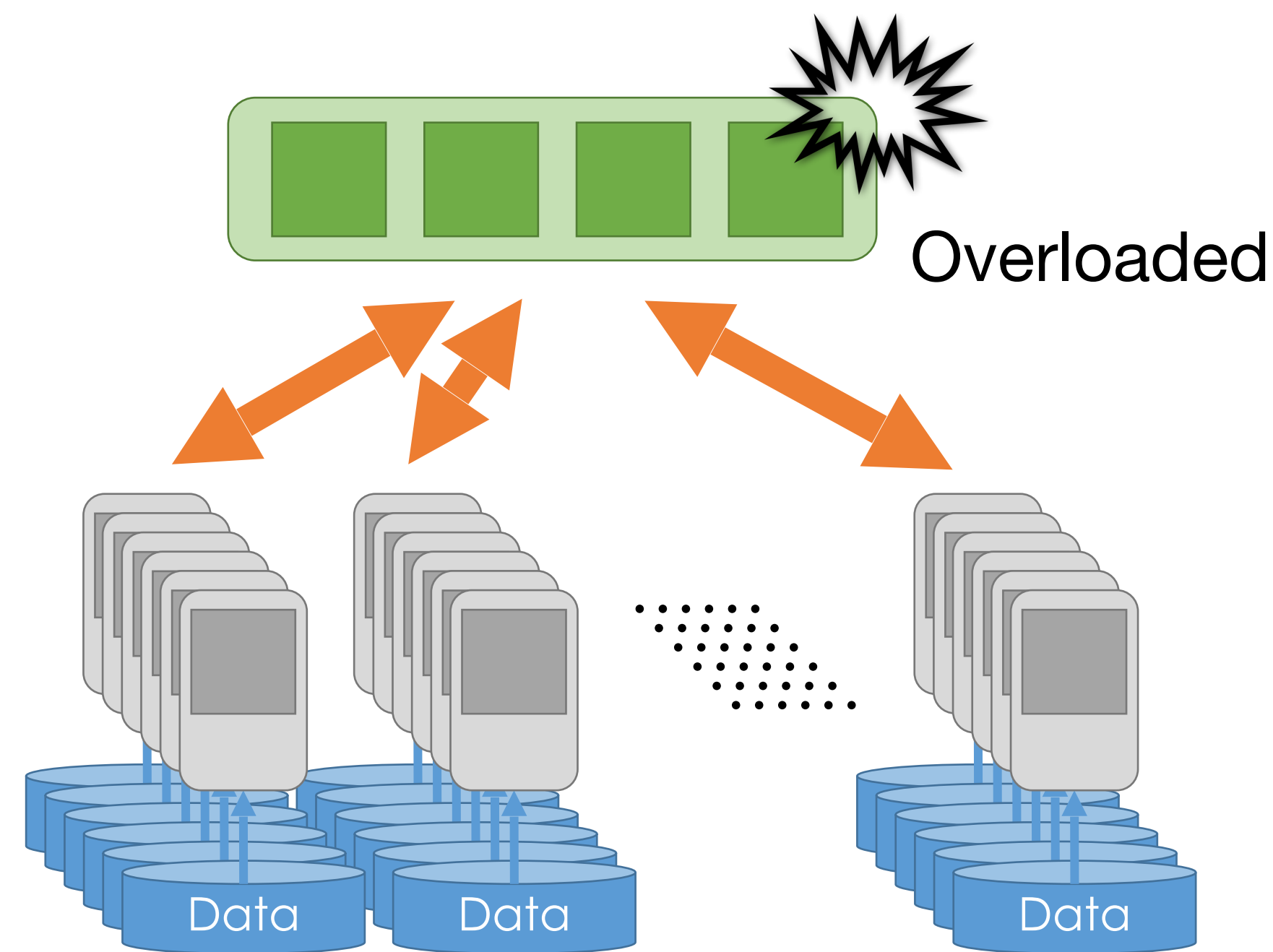
Scaling Distributed Machine Learning with the Parameter Server. Mu Li et al. 2014

Figure credits from: Deep Gradient Compression. Lin et al. 2018

Problems with Parameter Server



When number of workers is small 😊



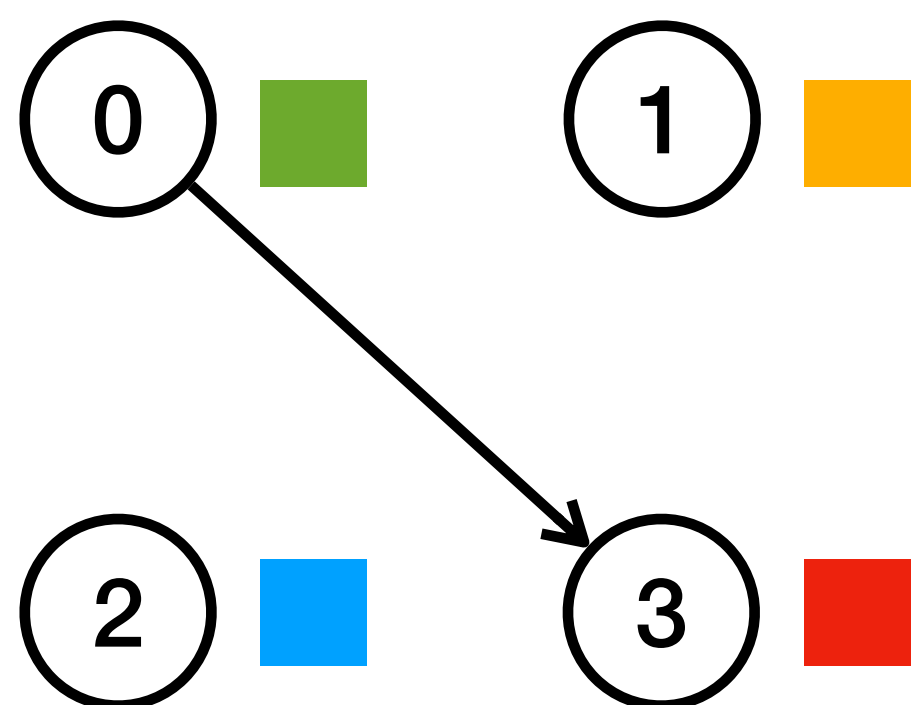
When number of workers is large 😭

The bandwidth requirement of parameter server grows linearly w.r.t number of workers.

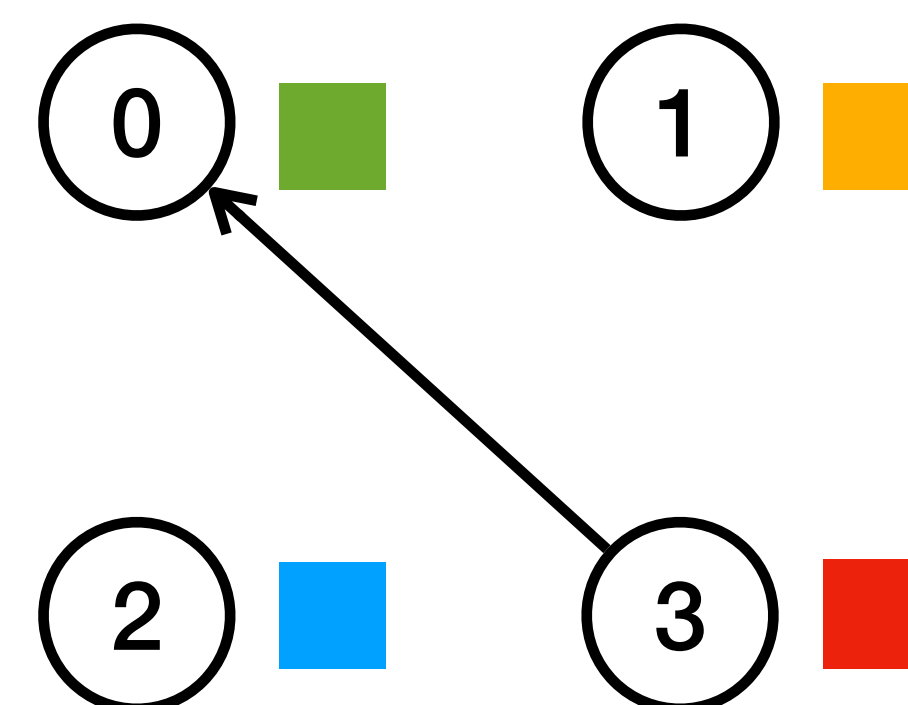
Distributed Communication

Point-to-Point: Send and Recv

Send: n0 -> n3



Recv: n0 -> n3



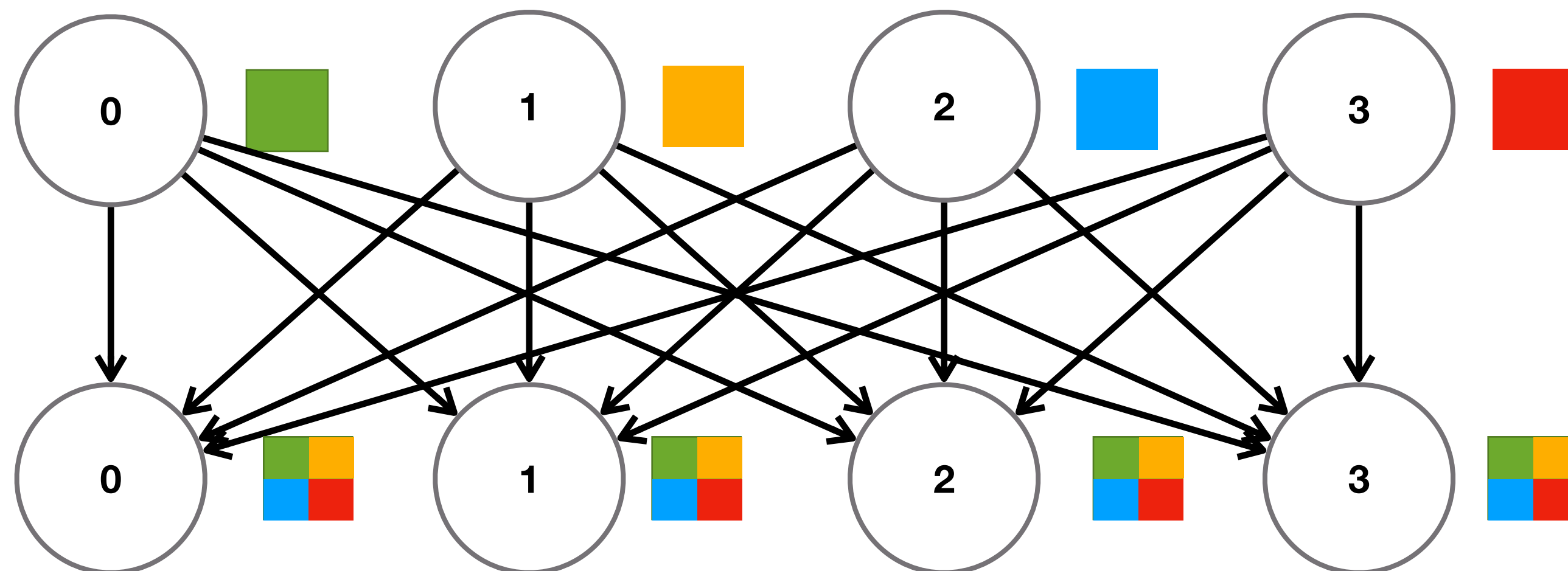
Point-to-point communication: transfer data from one process to another

- Send & Receive are the most common distributed communication schemes.
- Implemented in Socket / MPI / Gloo / NCCL

Distributed Communication

Naive All-Reduce Implementation - Parallel Reduce

Step 1



Pseudocode:

```
-----  
Parallel for i:=0 to N:  
  Allreduce(work[i])
```

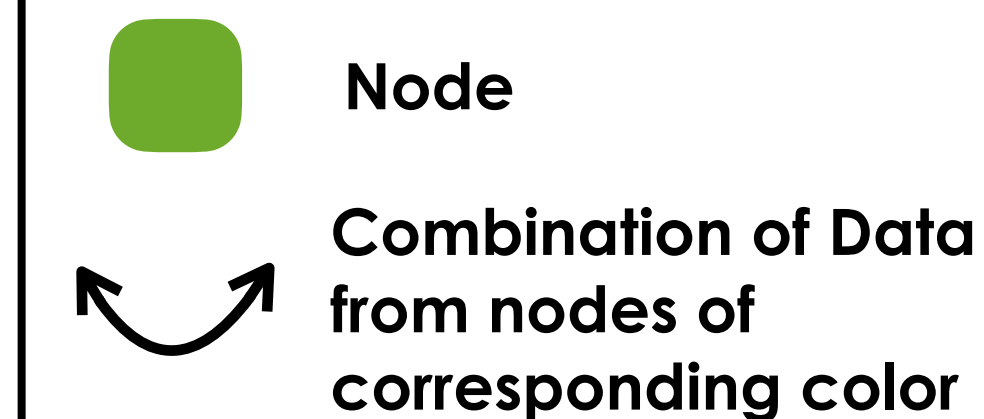
Time: $O(1)$ <- improved

Bandwidth: $O(N^2)$ <- worse

Perform **ALL** reduce operations simultaneously.

Distributed Communication

Recursive Halving All Reduce



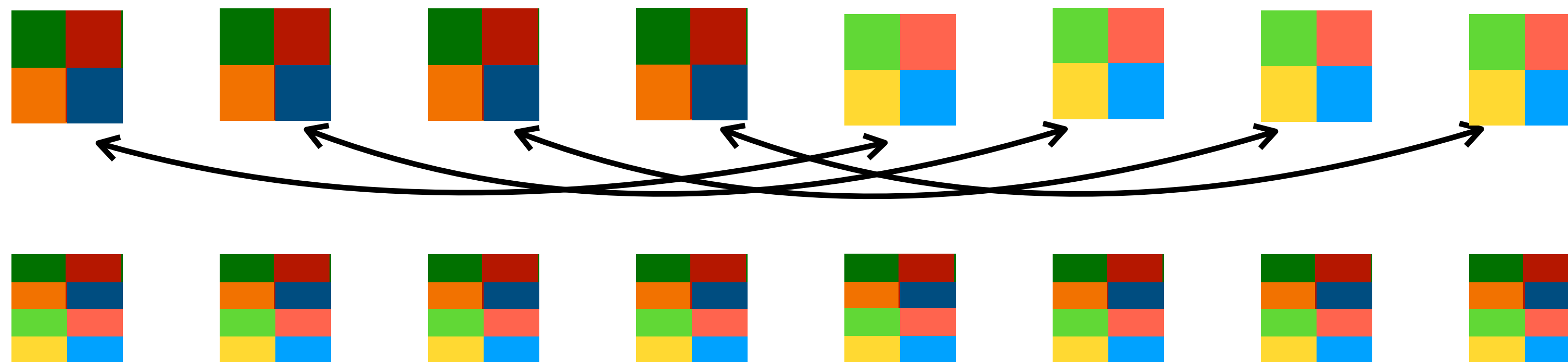
Step 1 - Each node exchanges with neighbors with offset 1



Step 2 - Each node exchanges with neighbors with offset 2



Step 3 - Each node exchanges with neighbors with offset 4



For N workers, AllReduce finish in **$\log(N)$** steps.

[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

Distributed Communication

All Reduce Implementations Comparison

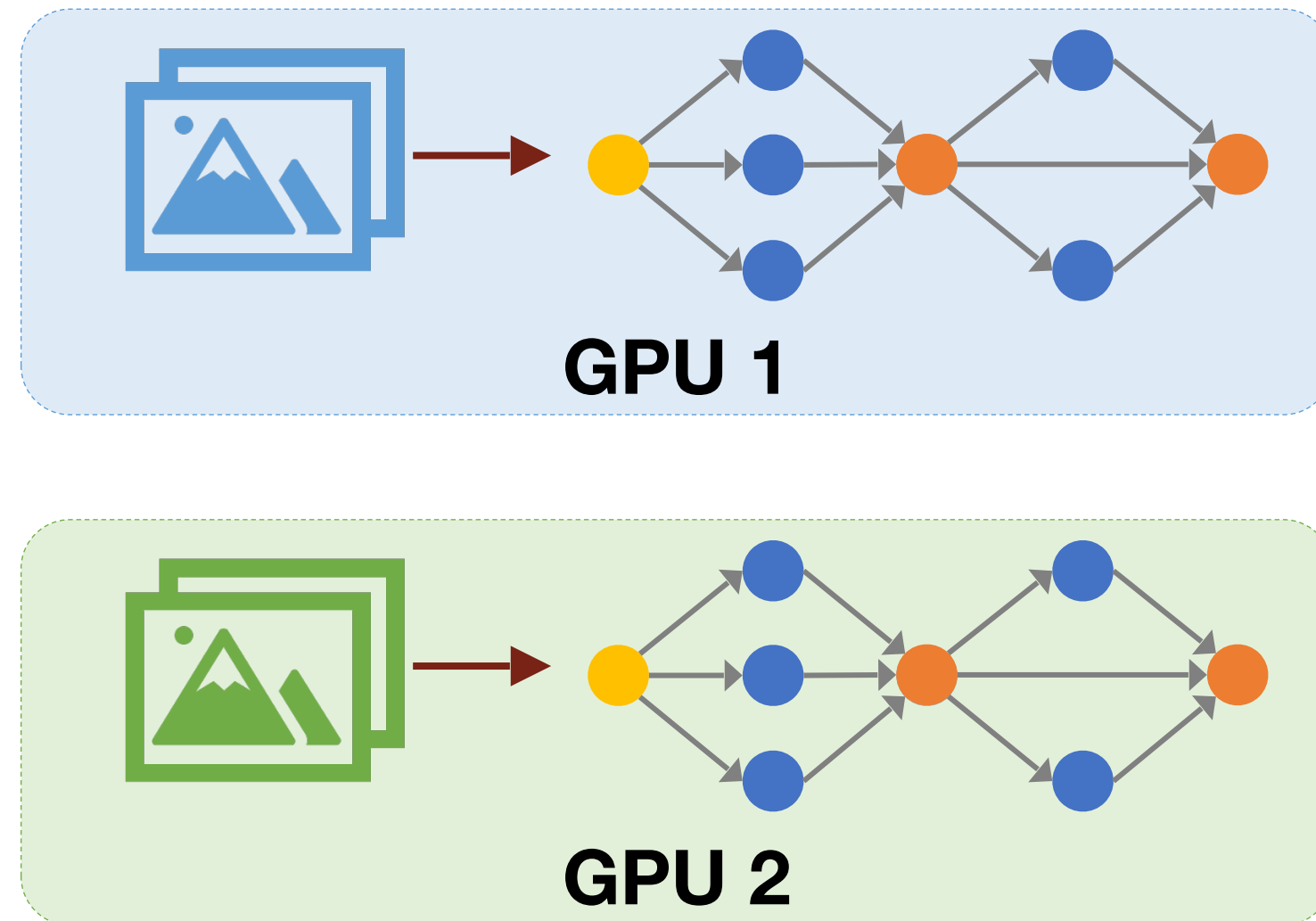
	Time	Peak Node Bandwidth	Total Bandwidth
Parameter Server	$O(1)$	$O(N)$	$O(N)$
All-Reduce - Sequential	$O(N)$	$O(N)$	$O(N)$
All-Reduce - Ring	$O(N)$	$O(1)$	$O(N)$
All-Reduce - Parallel	$O(1)$	$O(N)$	$O(N^2)$
All-Reduce - Recursive Halving	$O(\lg N)$	$O(1)$	$O(N)$

AllReduce with proper implementations reduce the peak bandwidth from $O(N)$ to $O(1)$ with little time overhead.

Parallelism in Distributed Training

- Data Parallelism
- **Model Parallelism**
- Compare the Advantages and Disadvantages of Two Parallelism

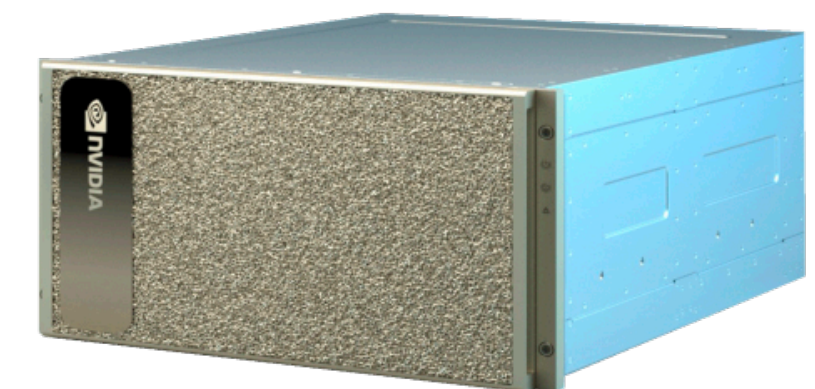
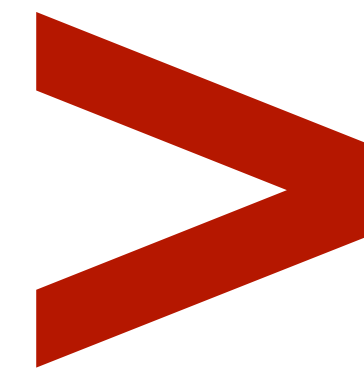
Data Parallelism Cannot Train Large Models



Though model parallelism has better device utilization, if train a super-large model (e.g., GPT-3)



175B * 16 Bits
= 350GB Memory



Nvidia A100 80GB

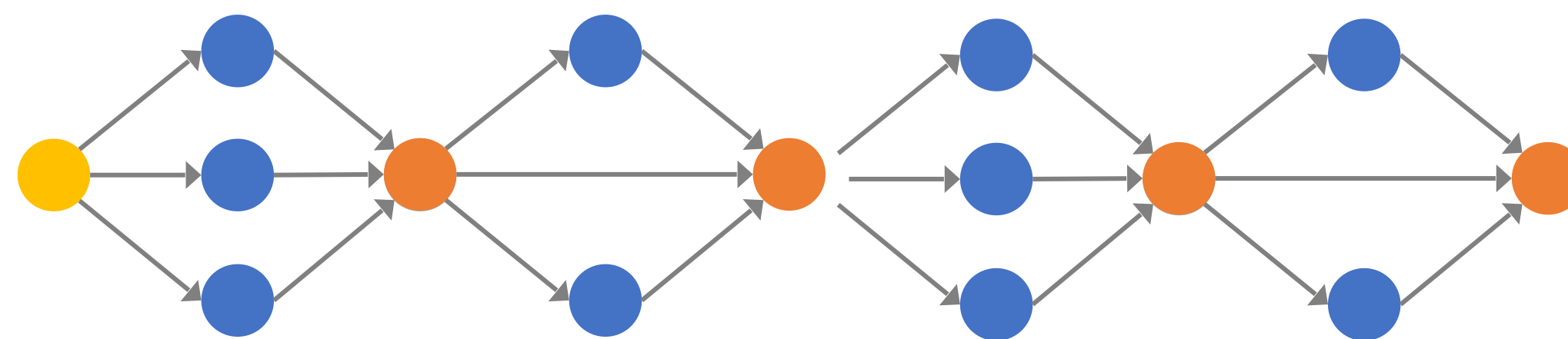
Even the best GPU **CANNOT** fit the model into memory!

Introduction to Distributed Training

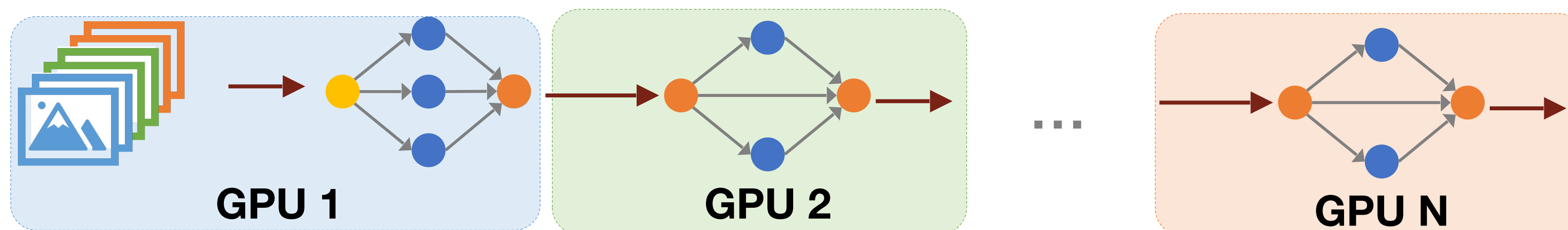
Model Parallelism



Training Dataset



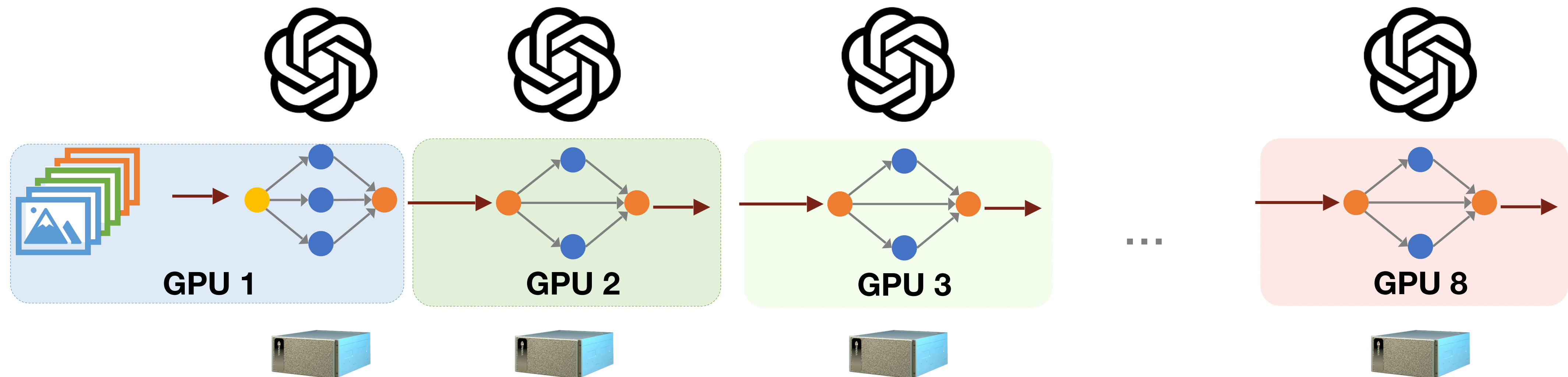
ML Model



Figures credit from CMU 15-849 [Jia 2022]

Model Parallelism Designed for Large Model Training

In order to fit training into hardware, instead of splitting the data, model parallelism split the model

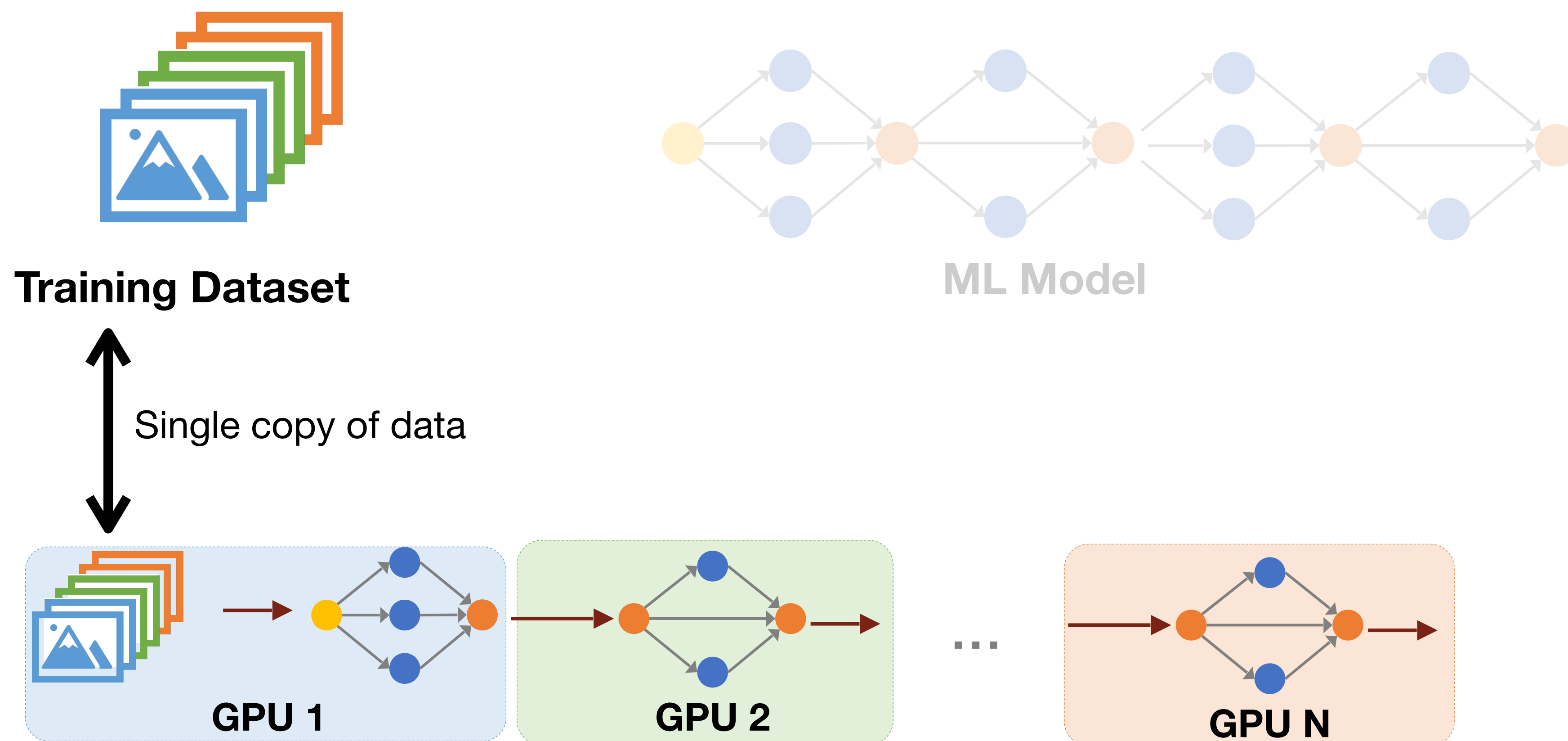


$$350\text{GB} / 8 \text{ cards} = 43.75\text{G} < 80\text{G}$$

With model parallelism, large ML models can be placed and trained on GPUs.

Introduction to Distributed Training

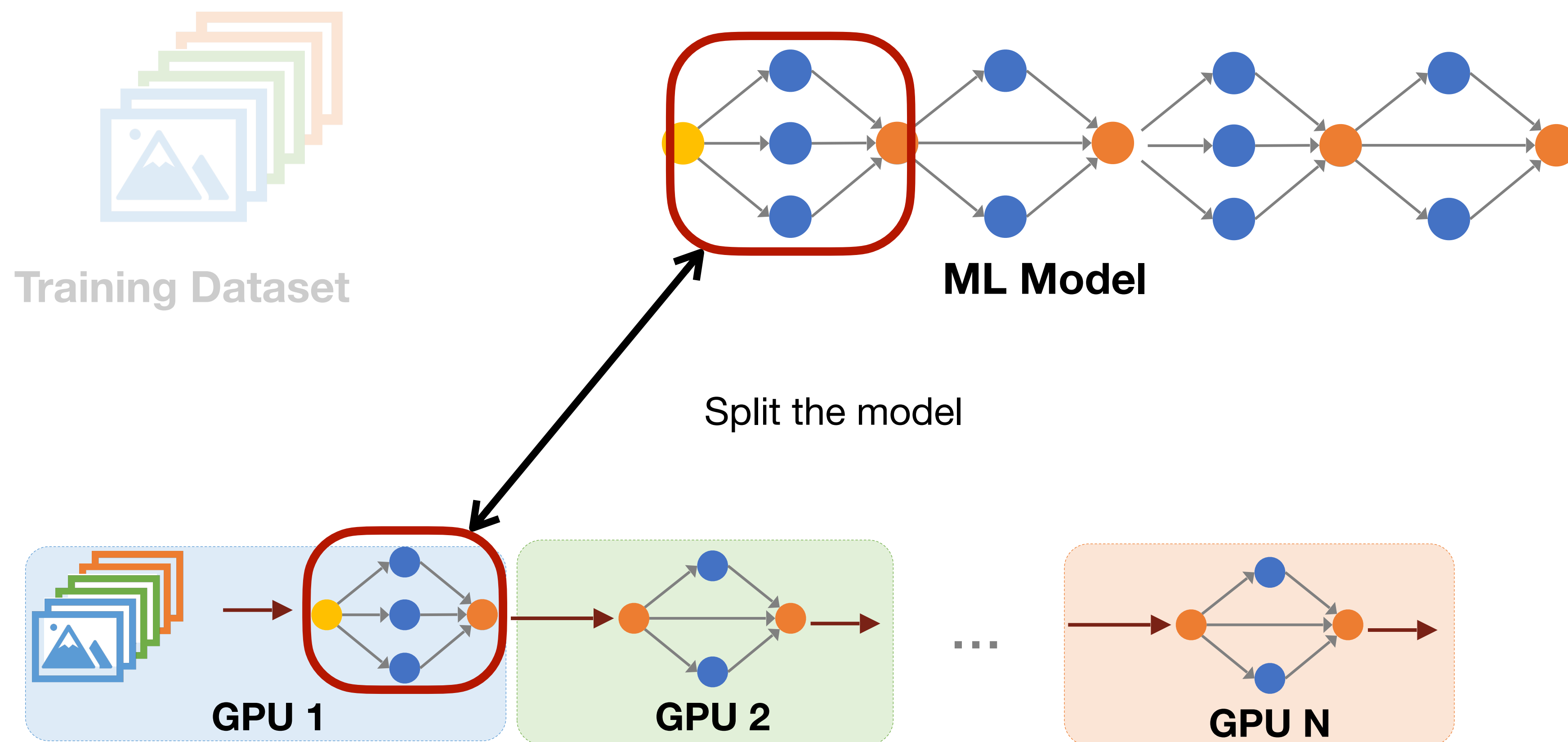
Model Parallelism



Figures credit from CMU 15-849 [Jia 2022]

Introduction to Distributed Training

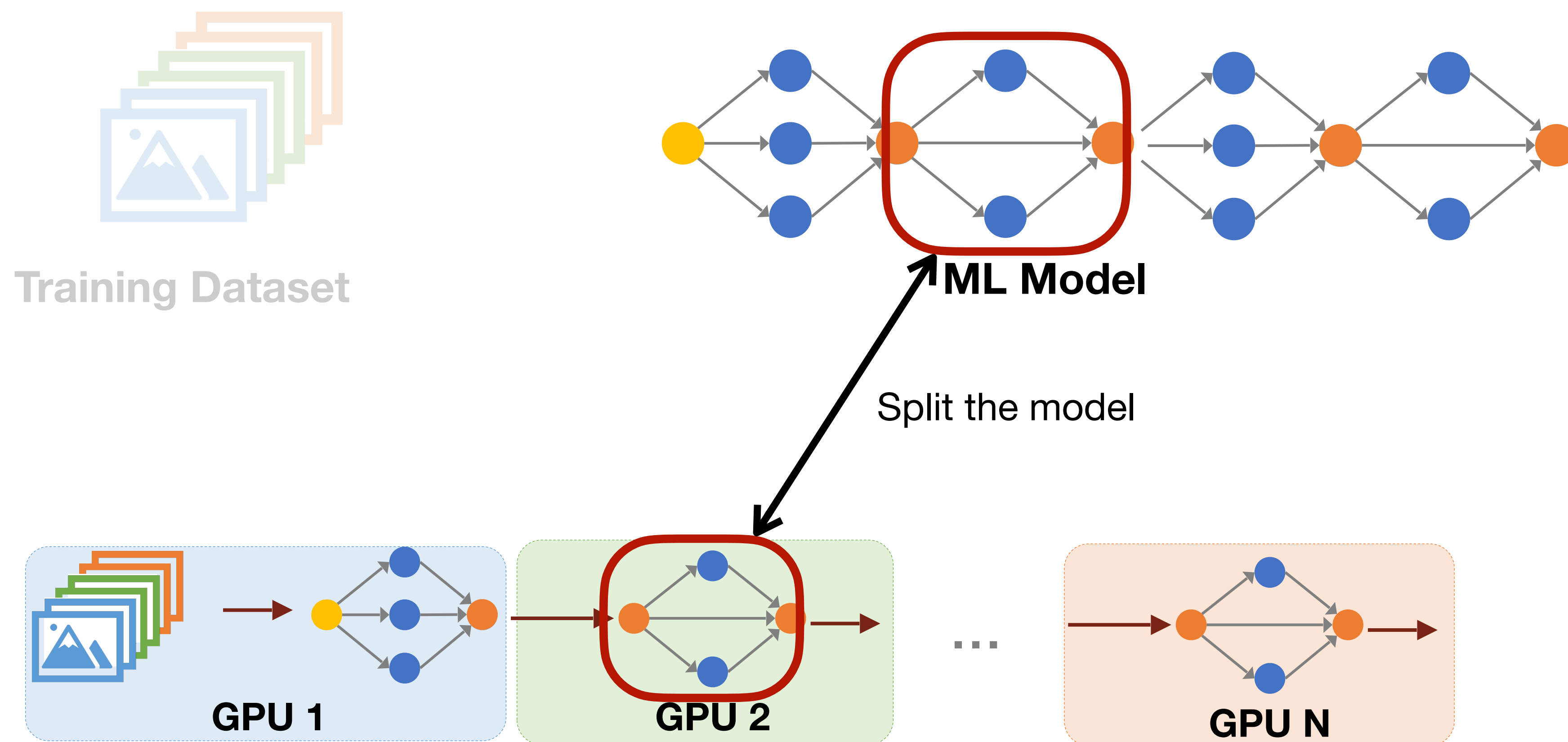
Model Parallelism



Figures credit from CMU 15-849 [Jia 2022]

Introduction to Distributed Training

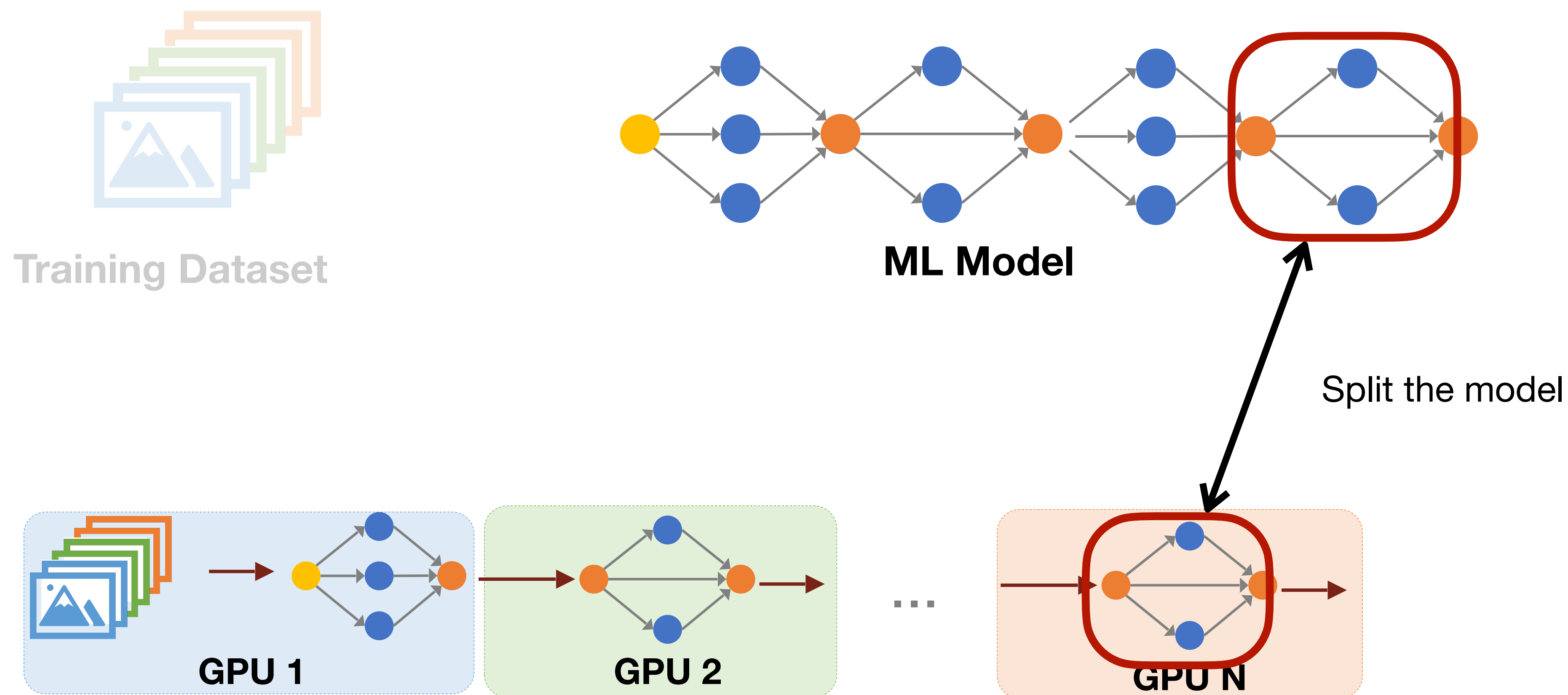
Model Parallelism



Figures credit from CMU 15-849 [Jia 2022]

Introduction to Distributed Training

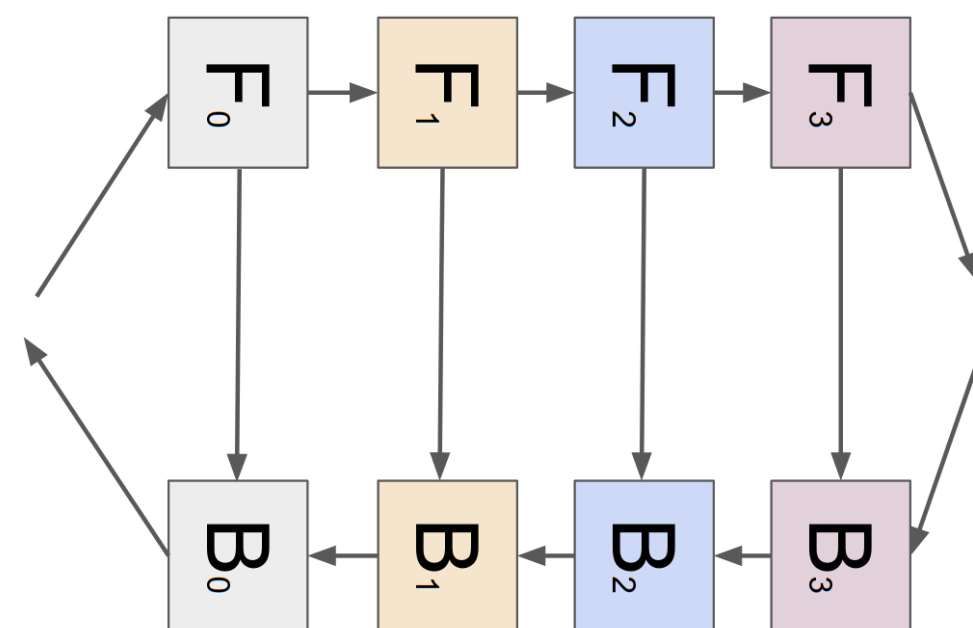
Model Parallelism



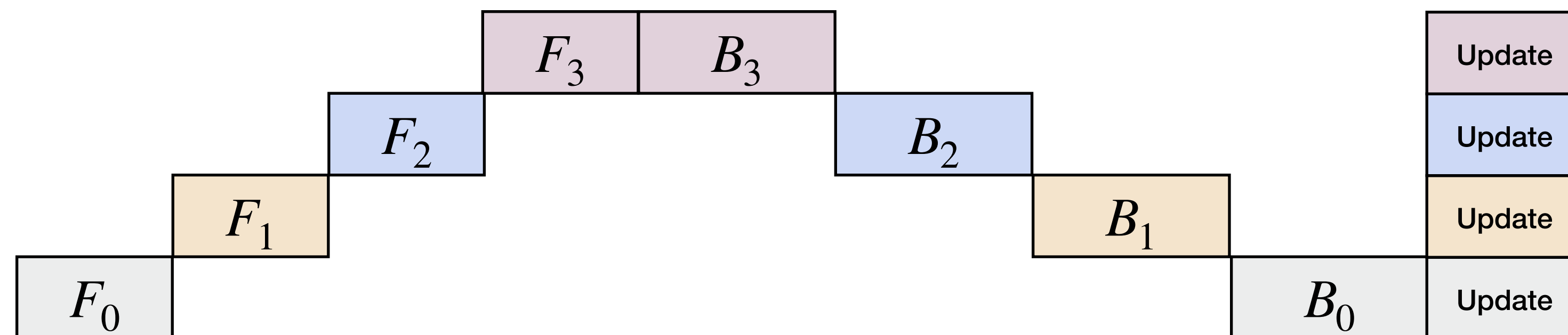
Figures credit from CMU 15-849 [Jia 2022]

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



(b). Training timeline

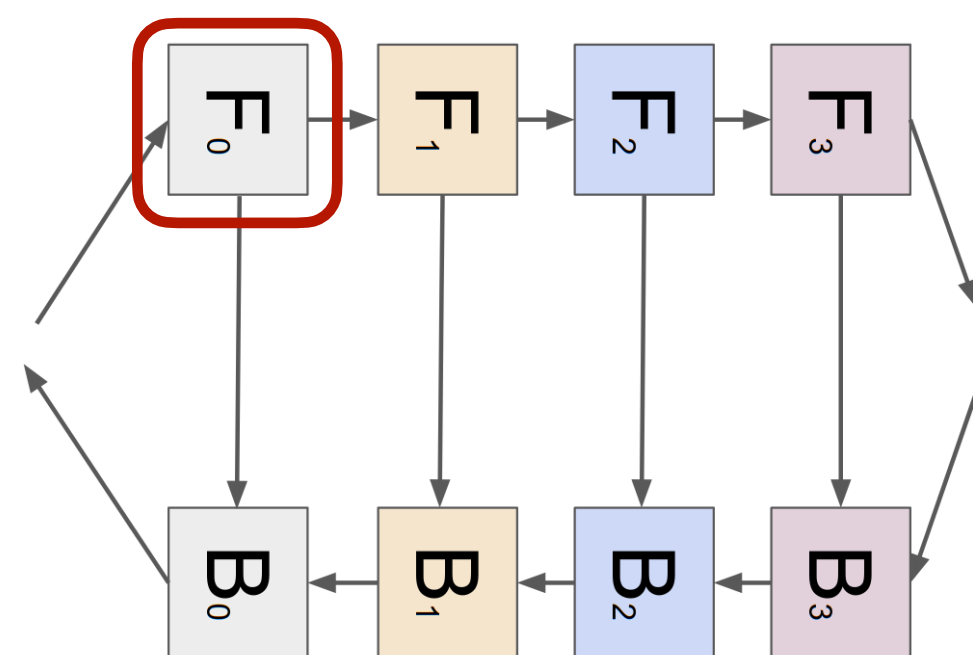
F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

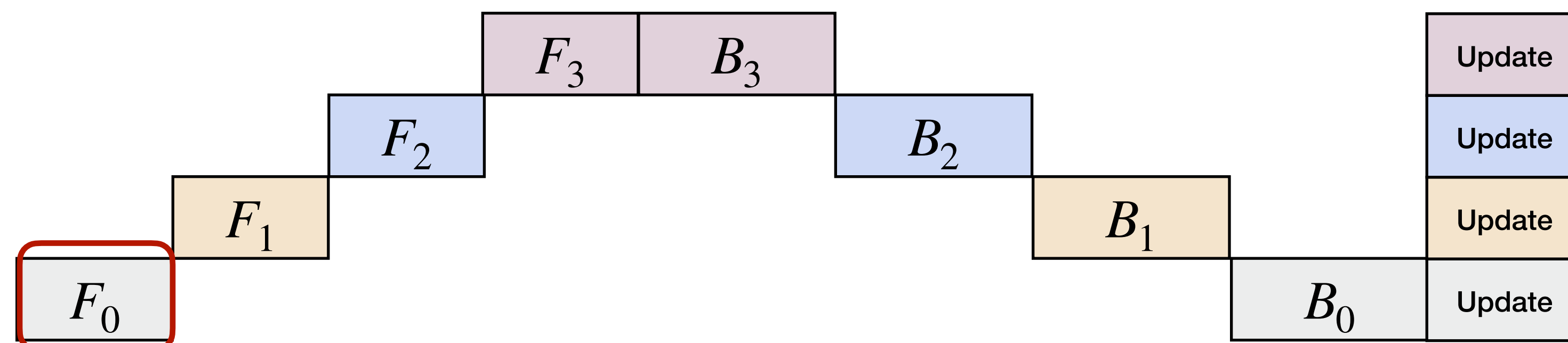
GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism [Huang et al. 2018]

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



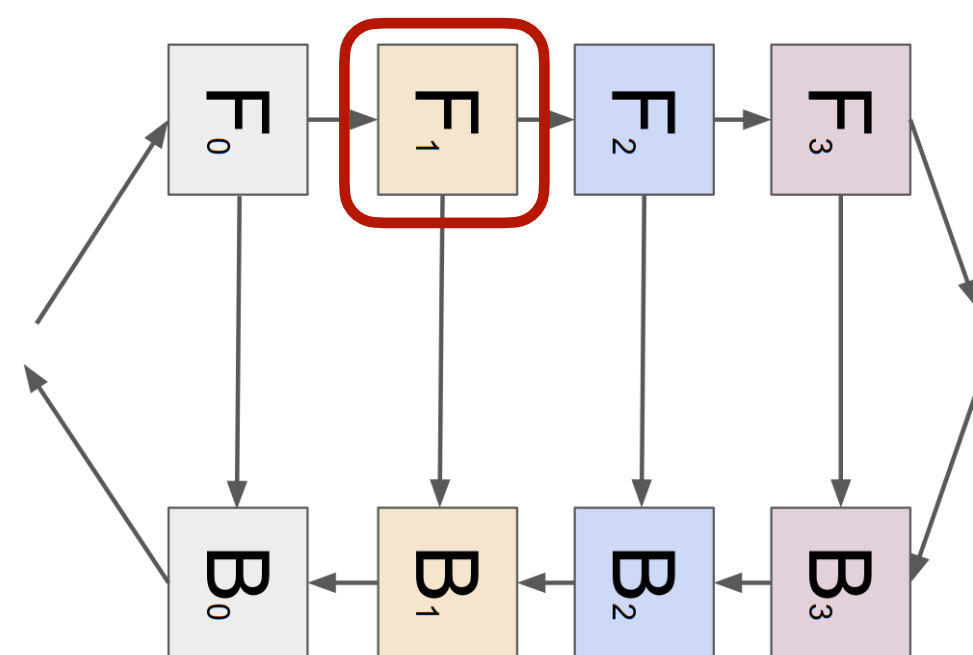
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with model parallelism.

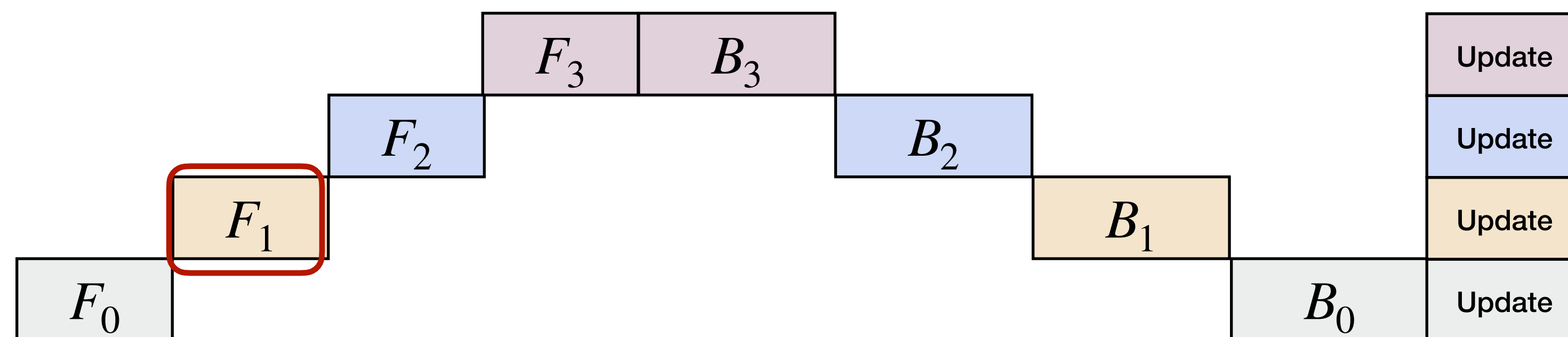
Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



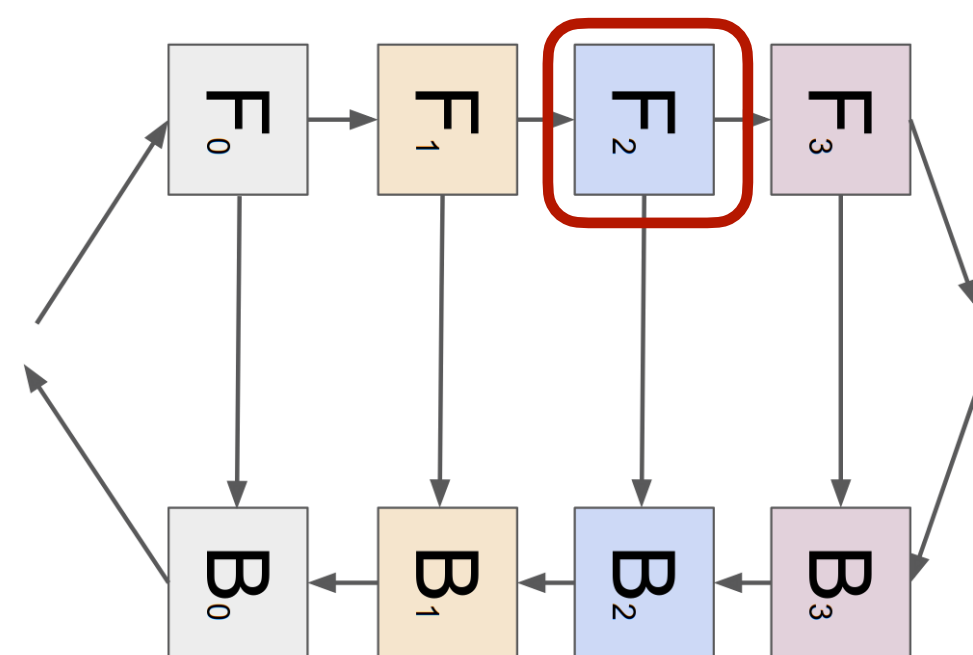
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with model parallelism.

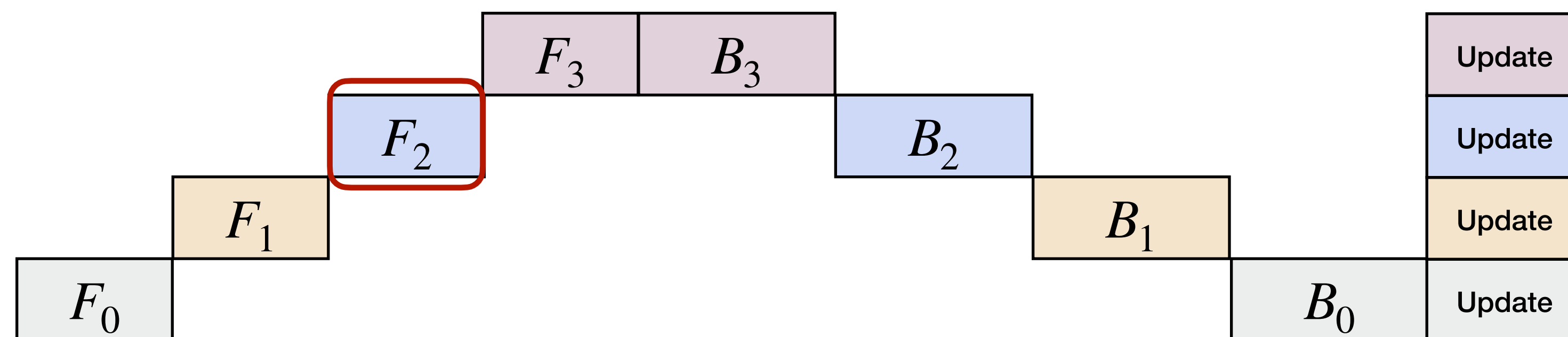
Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



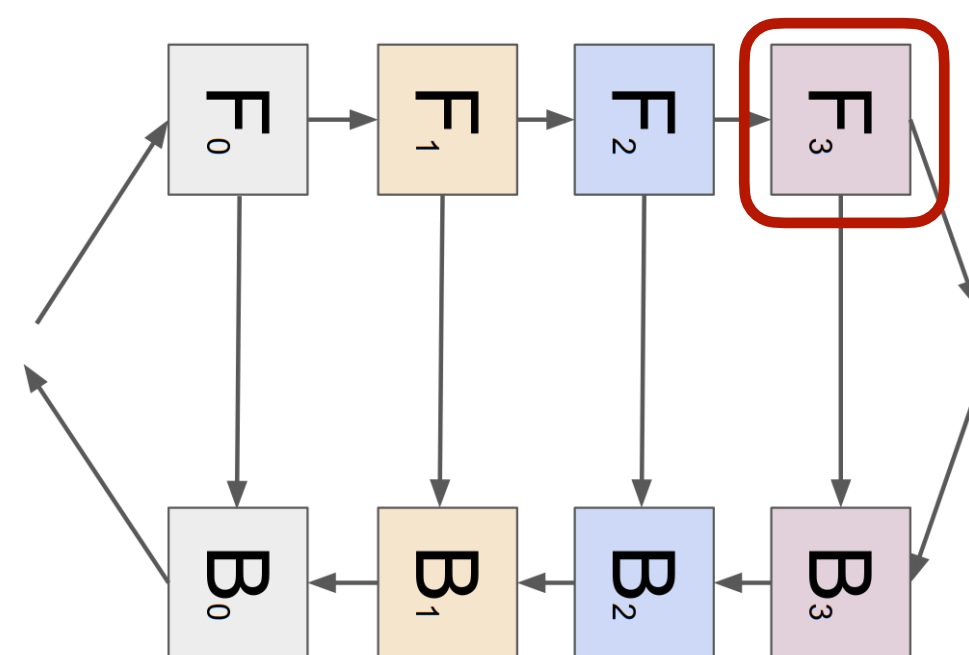
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with model parallelism.

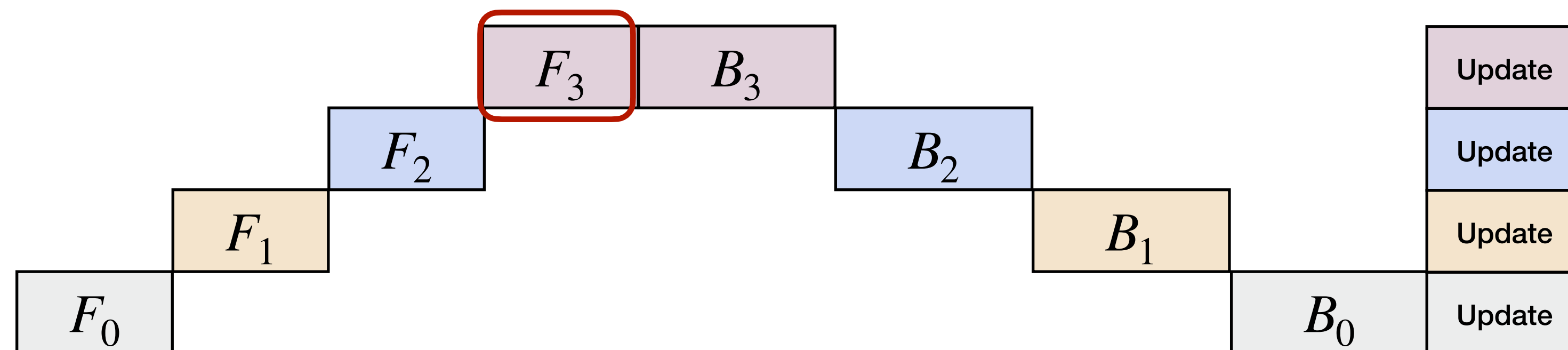
Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



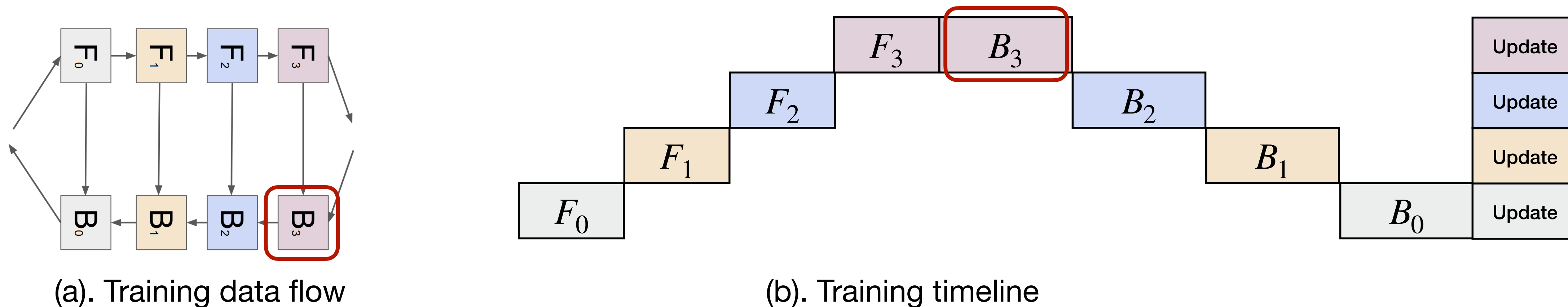
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation

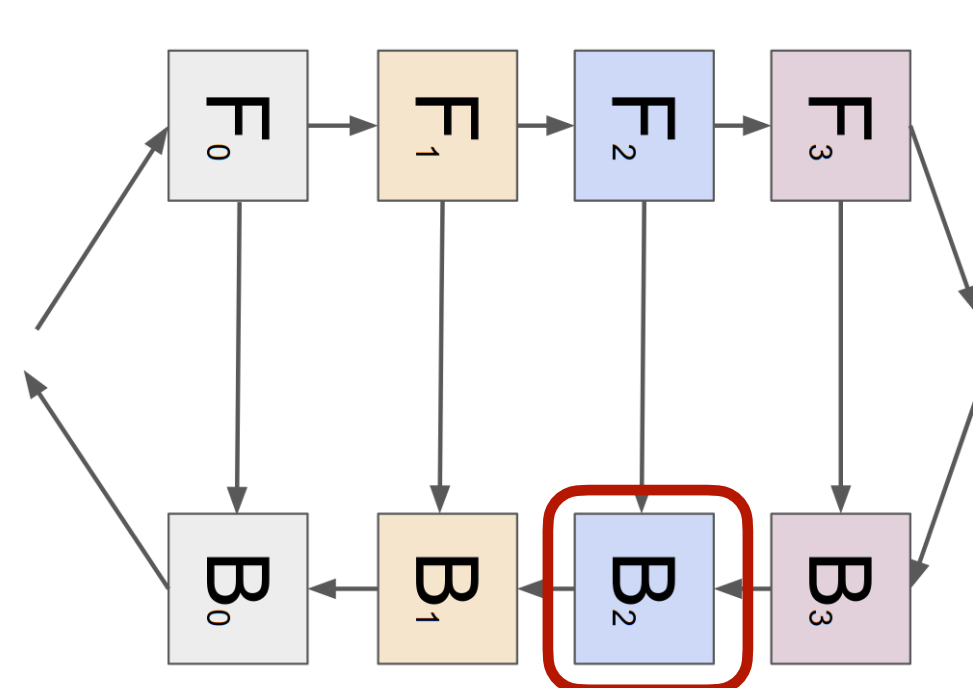


F: Forward B: Backward. Train a 4 layer network with model parallelism.

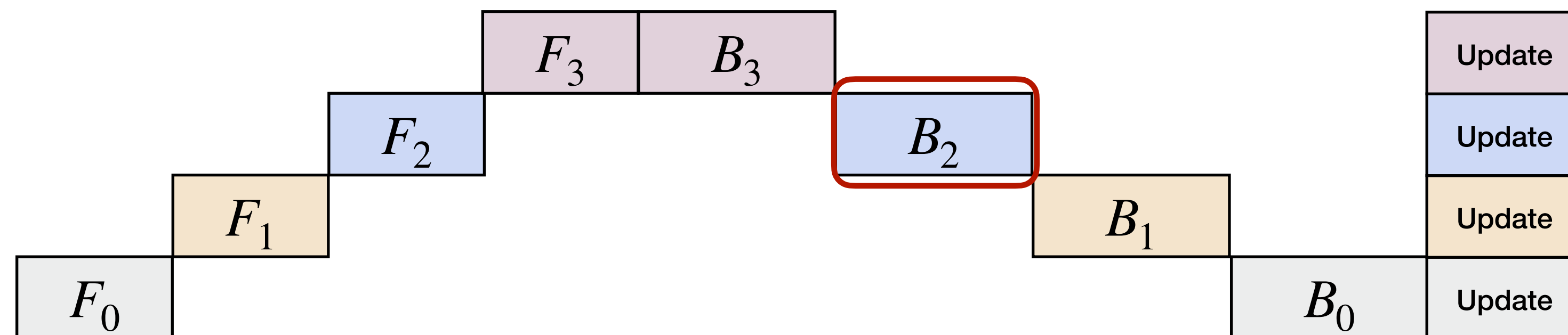
Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation



(a). Training data flow



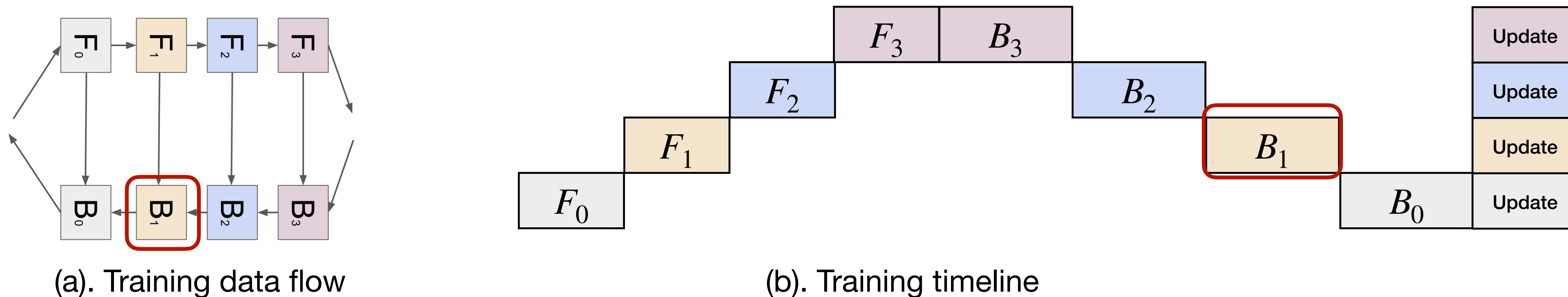
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

Naive Implementation

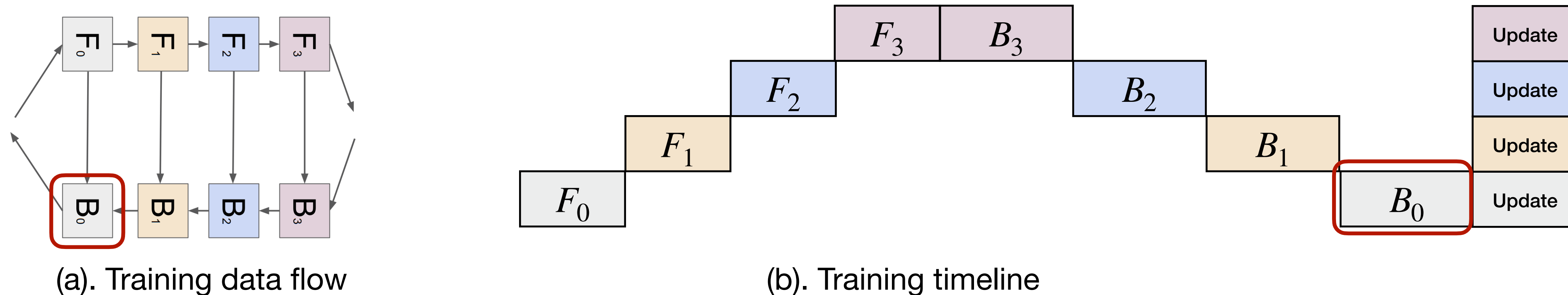


F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow

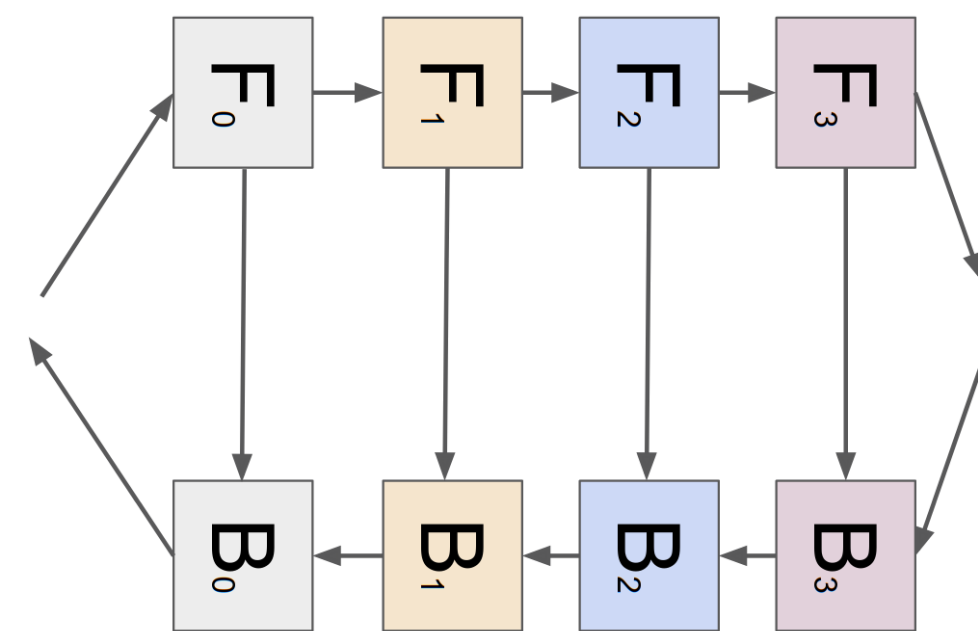
Naive Implementation



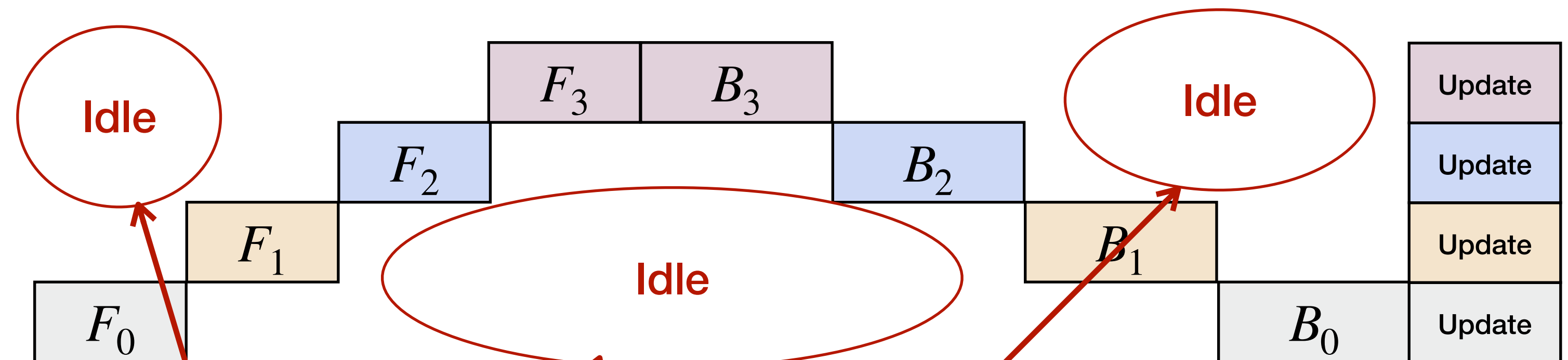
F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

Model Parallelism Workflow



(a). Training data flow



(b). Training timeline

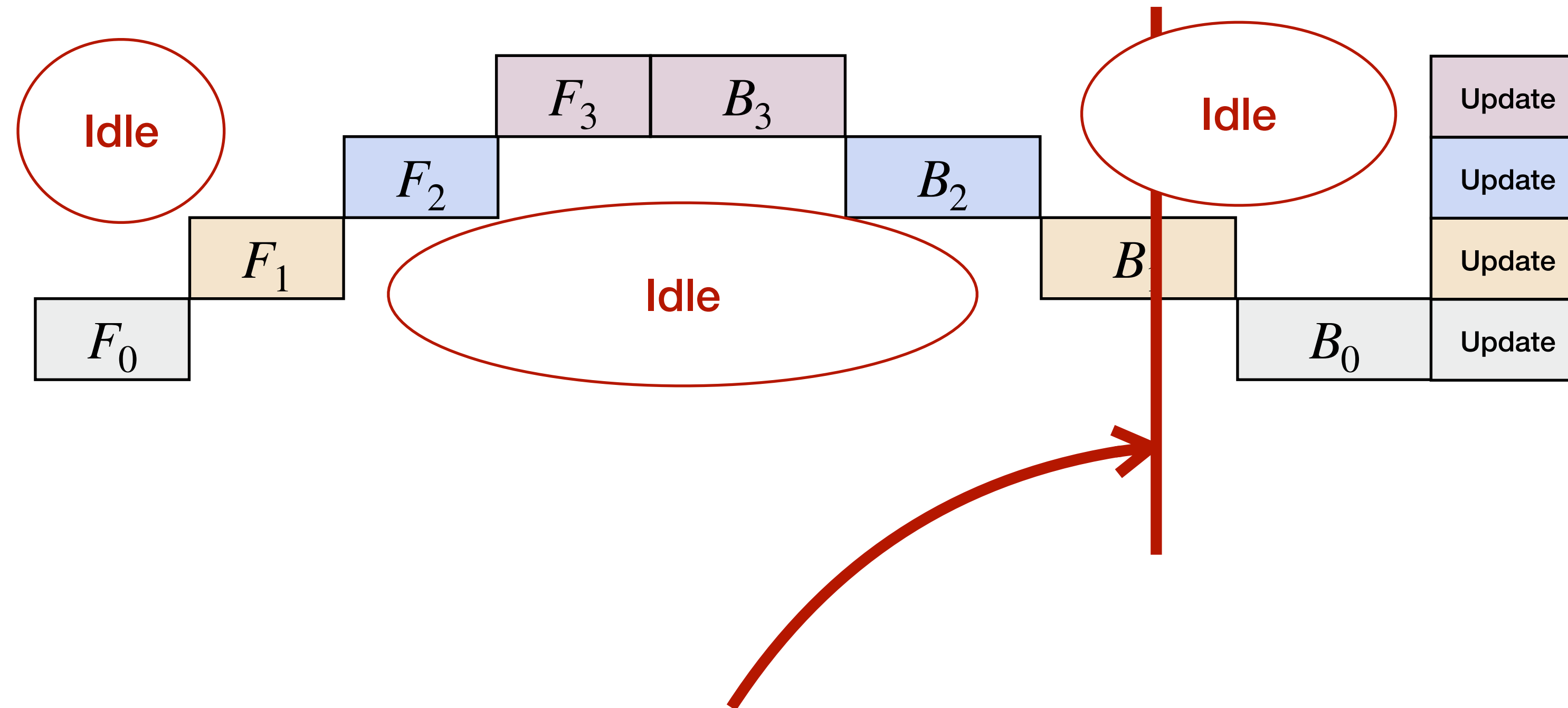
F: Forward B: Backward. Train a 4 layer network with model parallelism.

Model parallelism is needed for training a bigger DNN model on accelerators by dividing the model into partitions and assigning different partitions to different accelerators.

But accelerators are significantly **under-utilized** during model parallelism!

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism [Huang et al. 2018]

Naive Model Parallelism Suffers from Utilization



Only one device is computing at a time and others are waiting for it.

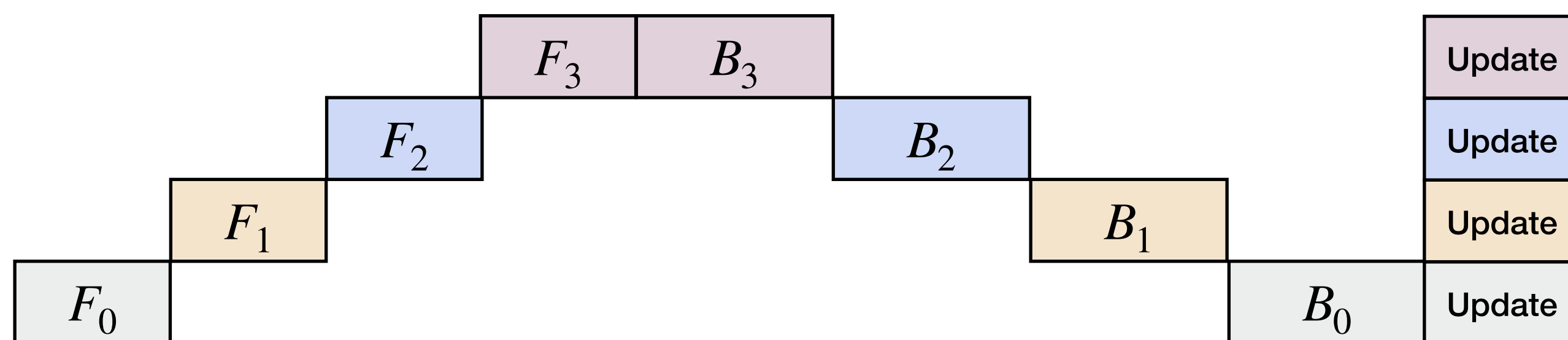
Theoretical utilization: 25% (**low!**)

Usual data parallelism utilization: ~75%

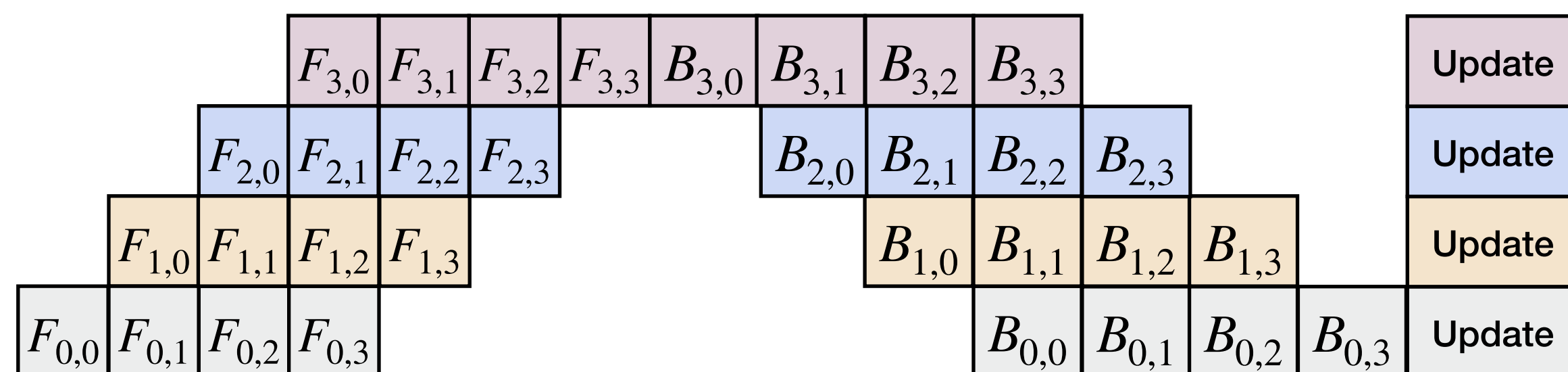
GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism [Huang et al. 2018]

Pipeline Parallelism

Gpipe: *Easy Scaling with Micro-Batch Pipeline Parallelism*



(a). Naive model parallelism

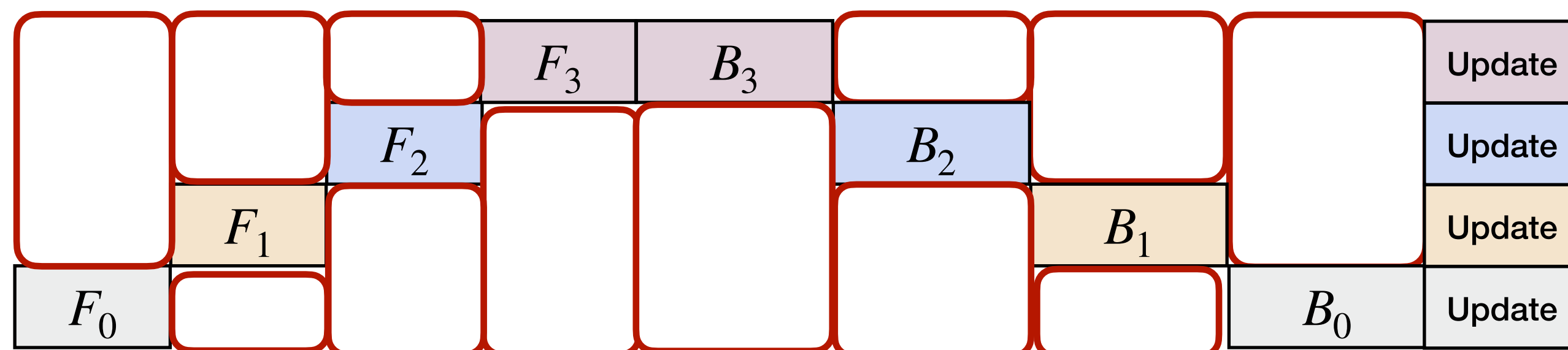


(b). Pipeline parallelism

- Split a single batch to micro batches
 - $[16, 10, 512] \rightarrow$
 - $[4, 10, 512]$
 - $[4, 10, 512]$
 - $[4, 10, 512]$
 - $[4, 10, 512]$
- Motivation: model parameters are not changed during computation within a batch, thus we can pipeline computation and communication

Pipeline Parallelism

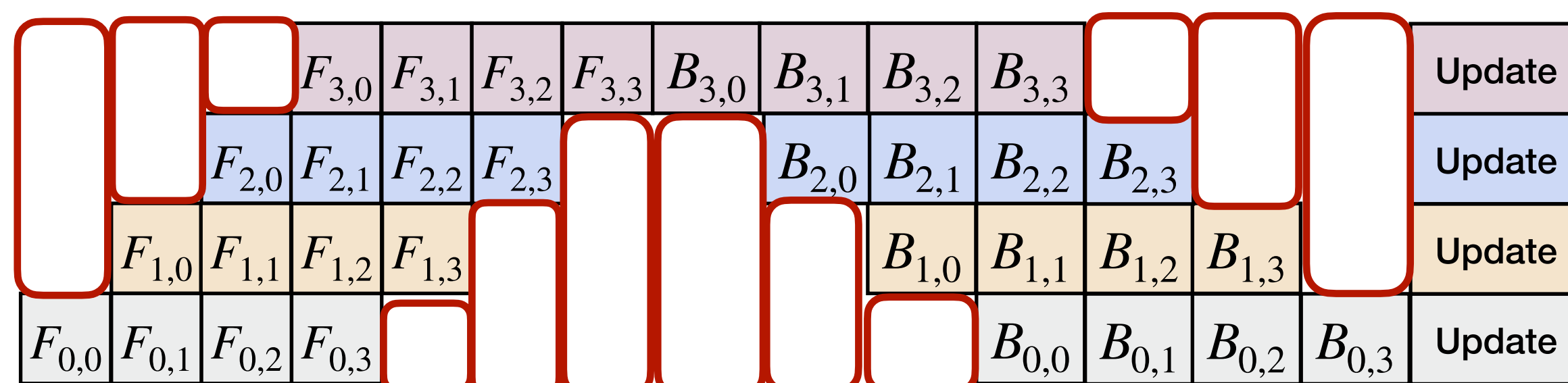
Micro-batch improves the device utilization



(a). Naive model parallelism

Utilization: (25%)

The more chunks (#num of micro batches), the higher of device utilization.



(b). Pipeline parallelism

Utilization: **57%** (2.5x improvement)

GPipe Pipeline Bubbles

- GPipe synchronizes weights at each mini batch
- Pro: maintain the original maths => potentially more efficient SGD
- Con: pipeline bubble => inefficient GPU use

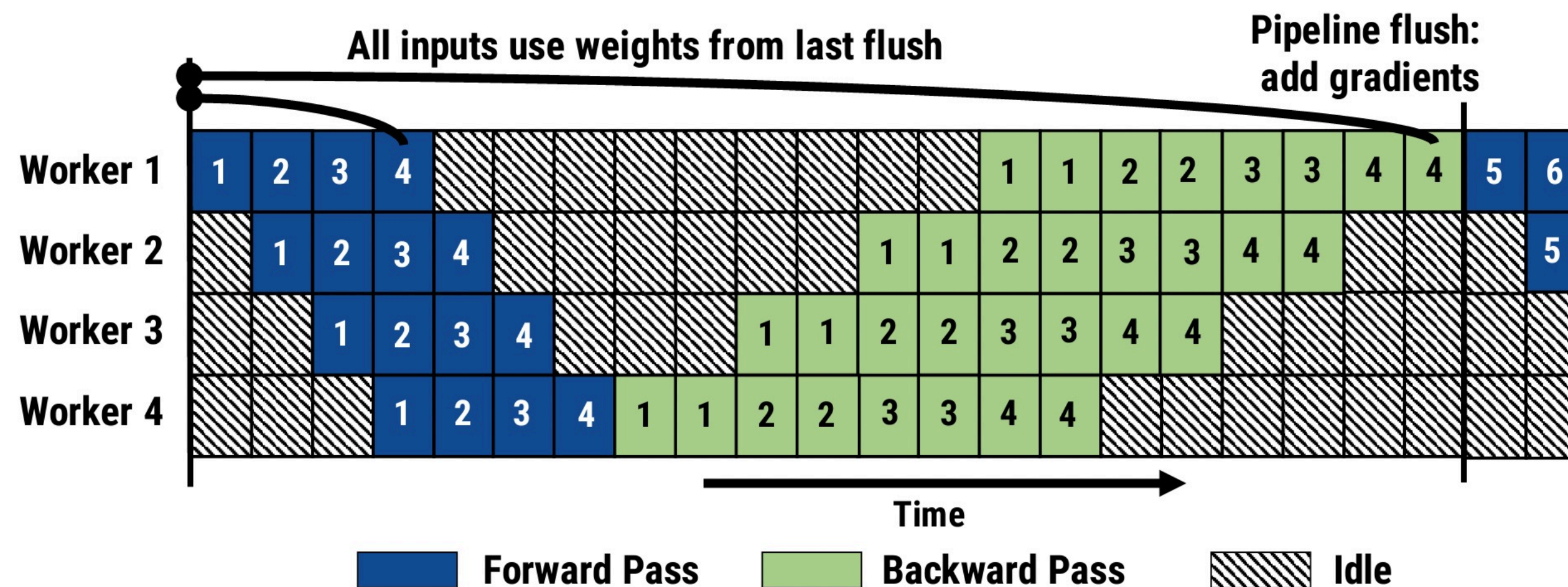
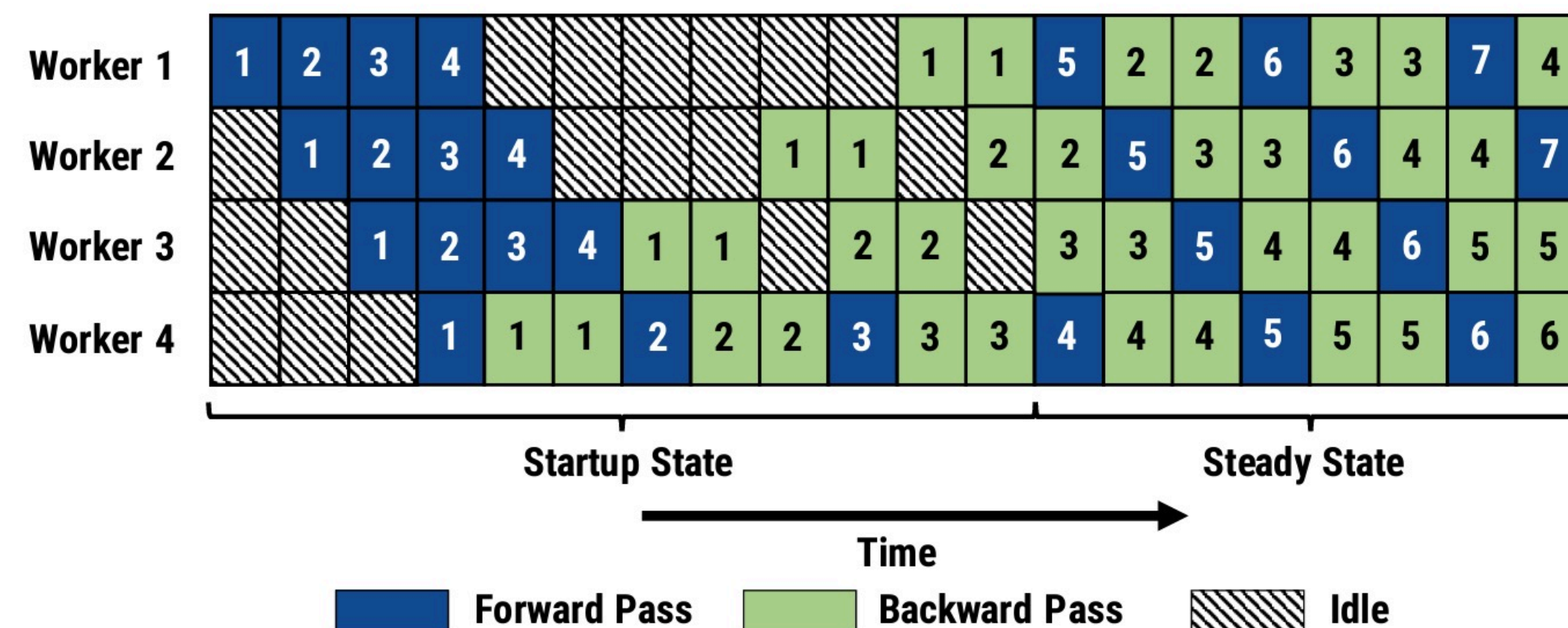


Figure 3: GPipe’s inter-batch parallelism approach. Frequent pipeline flushes lead to increased idle time.

Image from Narayanan eval. “PipeDream: Generalized Pipeline Parallelism for DNN Training” [SOSP 2019]

PipeDream: Basic Idea

- Partition layers into multi stages and process mini-batches (like GPipe)
- No synchronization points (unlike GPipe)
- Workers alternate between forward and backward passes on minibatches
- Gradients used to update model immediately



Challenge 1: Stage Partitioning

- How to partition model layers into the stages evenly?
- Throughput depends on the slowest stage in pipeline
- Solution:
 - Profile layer-wise perf and comm perf
 - Allows a stage to be replicated (DP)
 - Uses dynamic programming to find optimal partition and layer replication

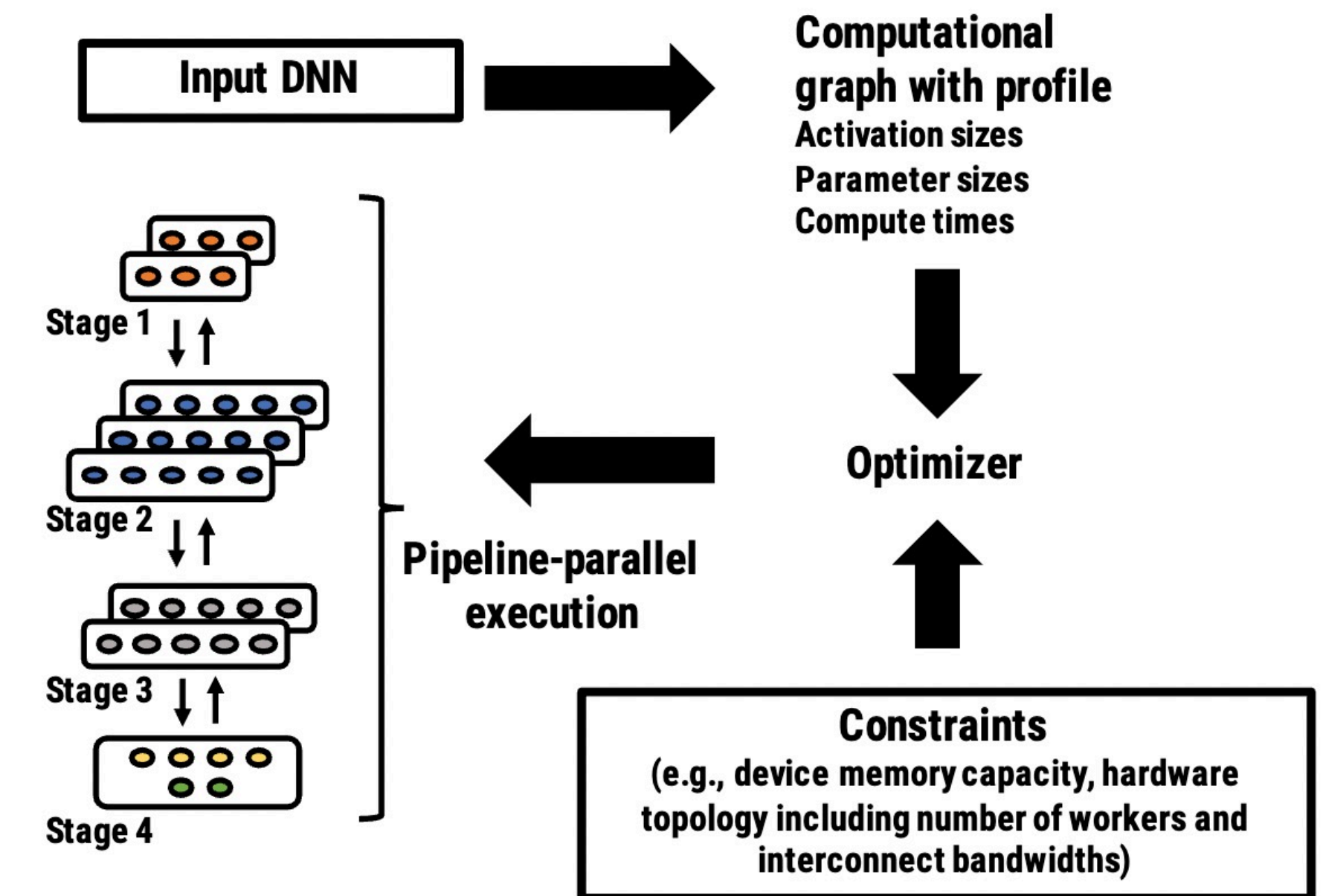


Figure 6: PipeDream's automated mechanism to partition DNN layers into stages. PipeDream first profiles the input DNN, to get estimates for each layer's compute time and output size. Using these estimates, PipeDream's optimizer partitions layers across available machines, which is then executed by PipeDream's runtime.

Challenge 2: Work Scheduling

- How to schedule forward and backward computation on a worker?
- Solution: 1F1B-RR
- Run one forward and one backward
- Round-robin across replicated stages

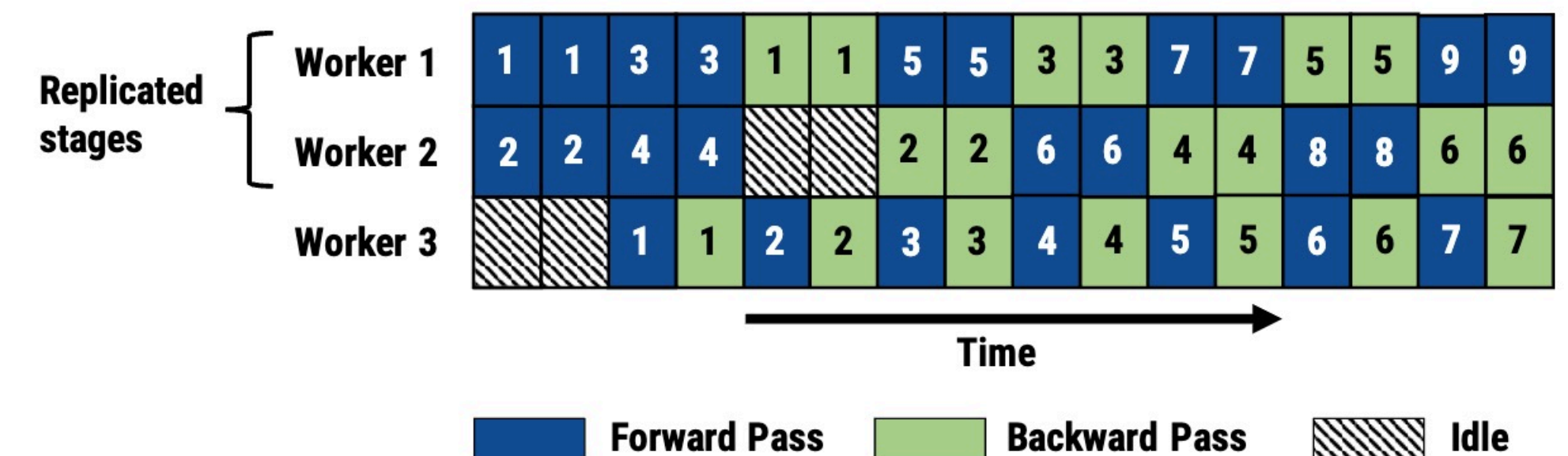
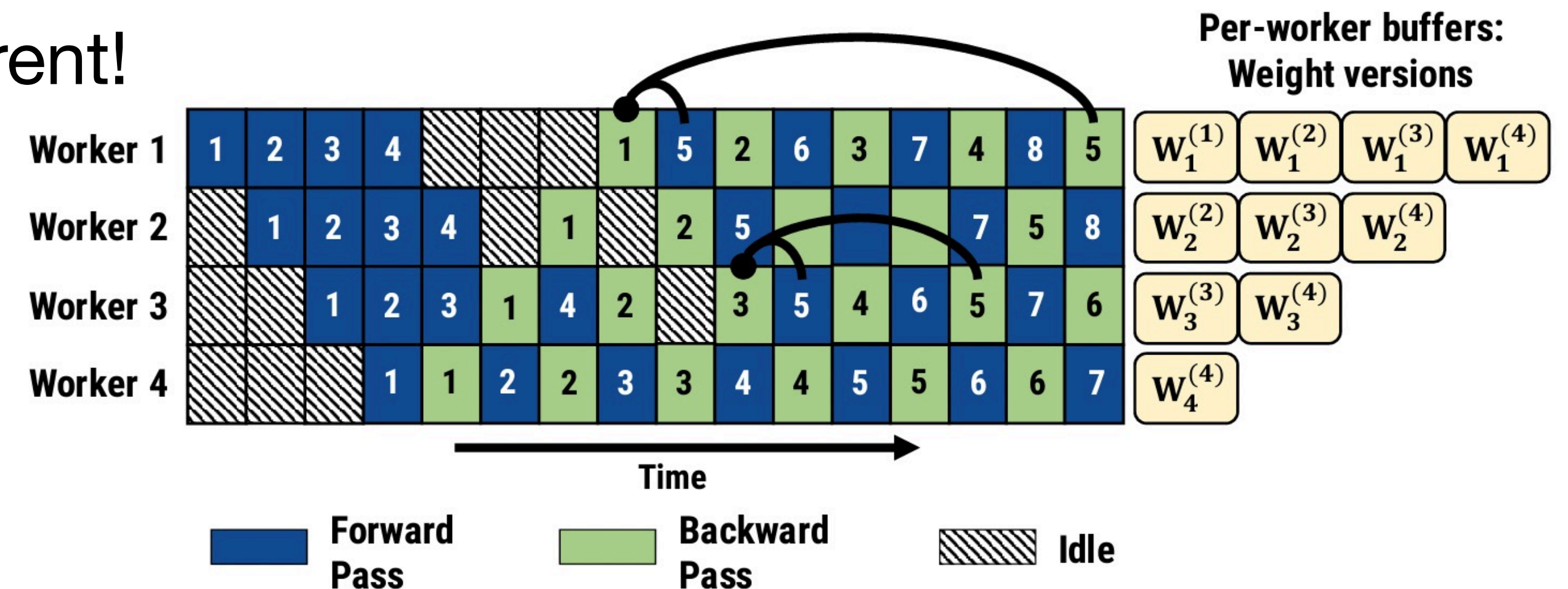


Figure 8: An example PipeDream pipeline with 3 workers and 2 stages. We assume that forward and backward passes in the first stage take two time units, while forward and backward passes in the second stage take only a single time unit. The first stage in this pipeline is replicated twice so that each stage sustains roughly the same throughput. Here, we assume that forward and backward passes take equal time, but this is not a requirement of our approach.

Challenge 3: Weight Versioning

- How to ensure the same minibatch uses the same weight version across workers for forward and backward?
- Otherwise computation will be far off and training not able to converge
- Solution: Store multiple weight versions so that the backward and forward of the same minibatch
- Weights across workers can be different!



PipeDream Pros and Cons

- Pros
 - no pipeline bubble
 - very efficient GPU utilization
- Cons
 - need to store multiple versions of weights (and intermediate results)
 - lossy computation (not the original maths)

Kimi K3: Training Beyond Static Pipelines

- Dynamic expert placement balances MoE work
- Partial rollouts pause stragglers across training iterations
- External CPU memory retains idle KV state
- Throttle rollout requests as queueing and KV pressure grow
- Discussion: optimize tokens per second, or completed useful trajectories?