Deep Learning

Sathish Vadhiyar

Introduction: Deep Neural Networks (DNNs)

- DNN composed of a sequence of tensors (generalized matrices with dynamical properties)
- The tensors are referred to as weights
- Input fed to DNN
- A series of tensor-matrix operations
  - Could be matrix-matrix multiplication, matrix-vector multiplication, FFT, non-linear transform
- Output obtained
- To get correct classification, need to get a set of working weights

Introduction: DNN Training

Weights need to be trained

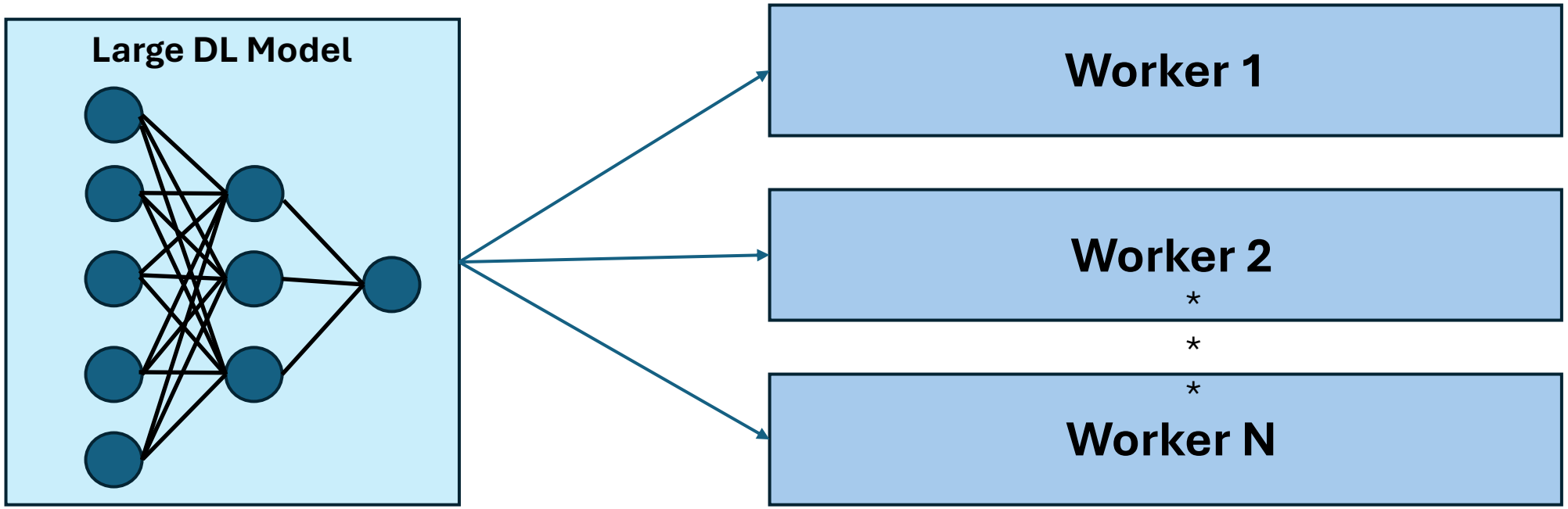
Training process consists of three steps:

1. Forward propagation: Input passed from first to last layer. Output is predicted
2. Backward propagation: Numerical prediction error passed from last to first layer and gradient of W, delta, obtained
3. Weight update: W = W-n.delta [n is the learning rate]

- Above three steps iterated until model is optimized
- Using stochastic gradient descent
- Randomly pick a batch of samples

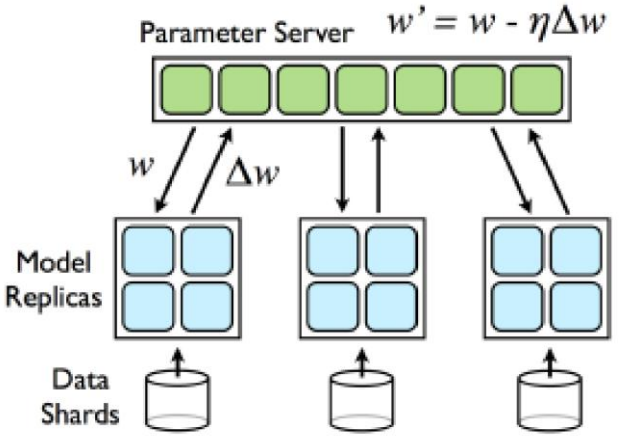
Introduction: Distributed and Parallel Training

Distributed and parallel Deep Learning involves utilizing high performance computing systems to train very large deep learning models



Parallel Algorithms

- Parameter server or asynchronous SGD
  - All workers complete their iteration step (local updates, sending to master, and receiving W from master) asynchronously
  - Master process uses lock to avoid weight update conflicts
  - One worker at a time
- Hogwild (lock-free)
  - Removes the above lock
  - Multiple workers at a time
- EAGSD (round-robin)



$$W_{t+1}^i = W_t^i - \eta(\Delta W_t^i + \rho(W_t^i - \bar{W}_t)) \quad (1)$$

Local updates by workers

$$\bar{W}_{t+1} = \bar{W}_t + \eta \sum_{i=1}^{P} \rho(W_t^i - \bar{W}_t) \quad (2)$$

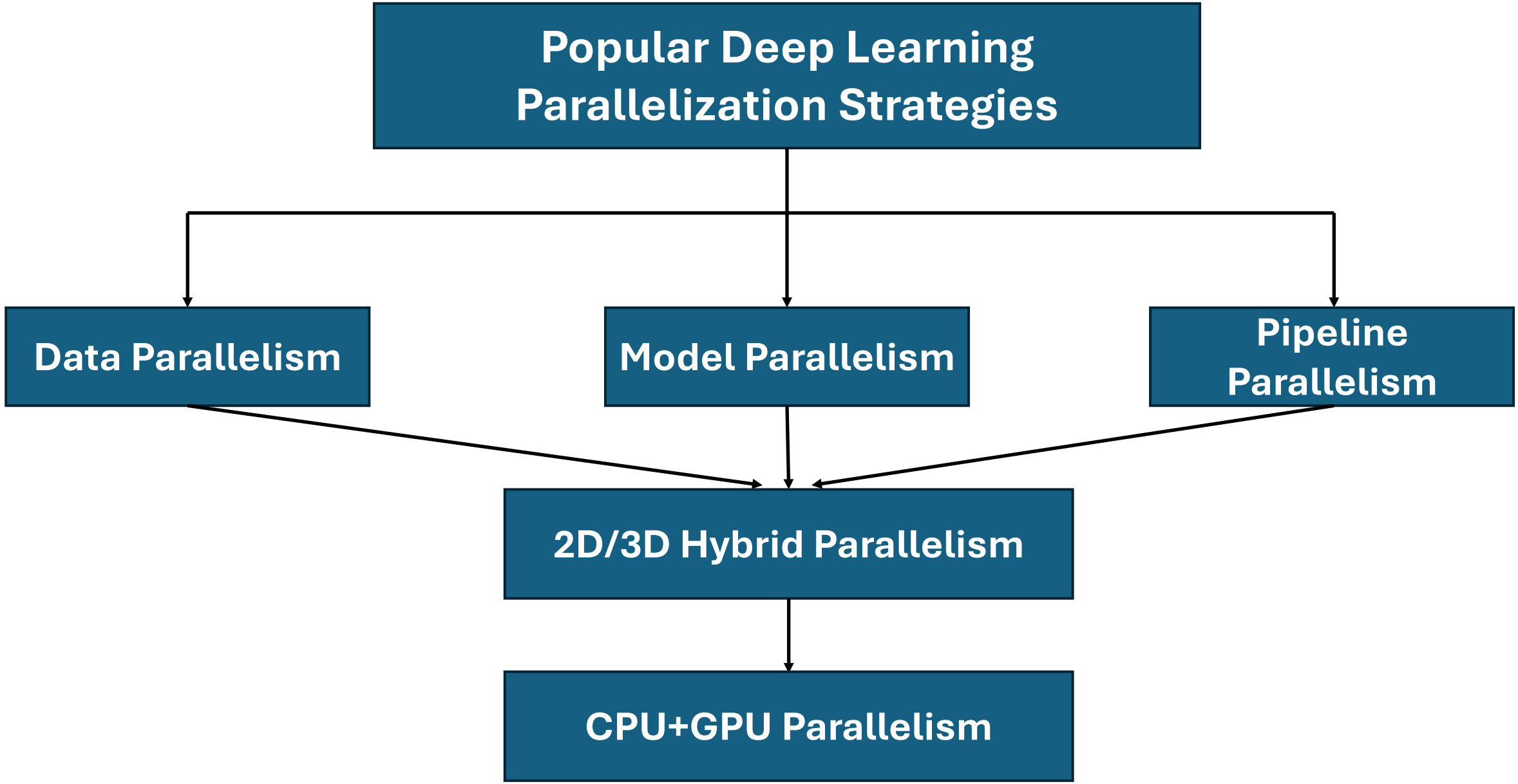
Global updates by master

Multi-GPU Implementation

Algorithm 1: Original EASGD on Multi-GPU system
master: CPU, workers: GPU_1, GPU_2, ..., GPU_P

Input: samples and labels: $\{X_i, y_i\}$ $i \in 1, \ldots, n$
#iterations: T, batch size: b, #GPUs: G
Output: model weight W

1 Normalize X on CPU by standard deviation: $E(X) = 0$ (mean) and $\sigma(X) = 1$ (variance)
2 Initialize W on CPU: random and Xavier weight filling
3 **for** $j = 1; j <= G; j{+}{+}$ **do**
4 create **local** weight W_j on j-th GPU, copy W to W_j
5 create **global** weight $\bar{W}_1$ on 0-th GPU, copy W to $\bar{W}_1$
6 **for** $t = 1; t <= T; t{+}{+}$ **do**
7 j = t mod G
8 CPU **randomly** picks b samples
9 CPU **asynchronously** copies b samples to j-th GPU
10 CPU sends $\bar{W}_t$ to j-th GPU
11 Forward and Backward Propagation on j-th GPU
12 CPU gets W_t^j from j-th GPU
13 j-th GPU updates W_t^j by Equation (1)
14 CPU updates $\bar{W}_t$ by $\bar{W}_{t+1} = \bar{W}_t + \eta\rho(W_t^j - \bar{W}_t)$


Popular Deep Learning Parallelization Strategies
Data Parallelism
Model Parallelism
Pipeline Parallelism
2D/3D Hybrid Parallelism
CPU+GPU Parallelism

Data Parallelism

Splits data across compute nodes

Each machine maintains an identical copy of the model

During the backward step, after every batch, all the subgradients are collected from all the processors

Using the sum of all the subgradients, delta, the weights are updated

Using the same gradients to update the models results in identical model weights being maintained across the nodes

Pros:

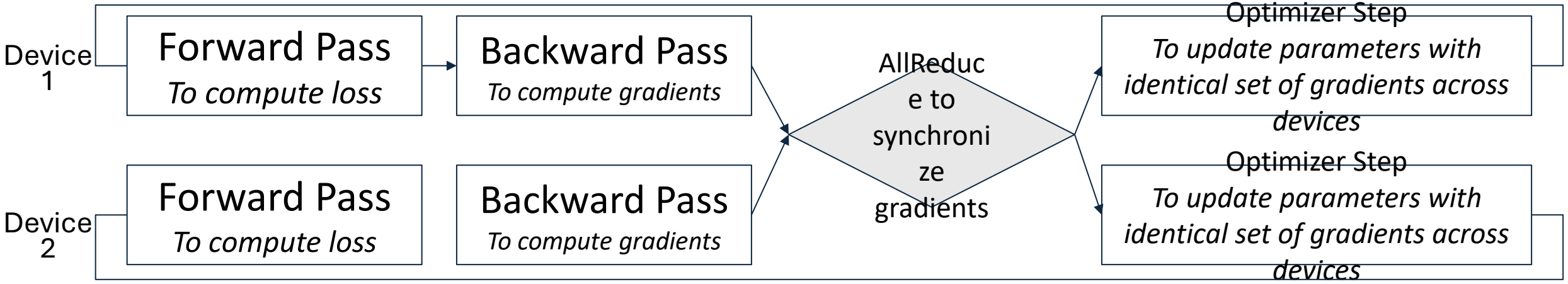
- Reduces training time significantly compared to single node training

Cons:

- Unable to train large models in GPU memory

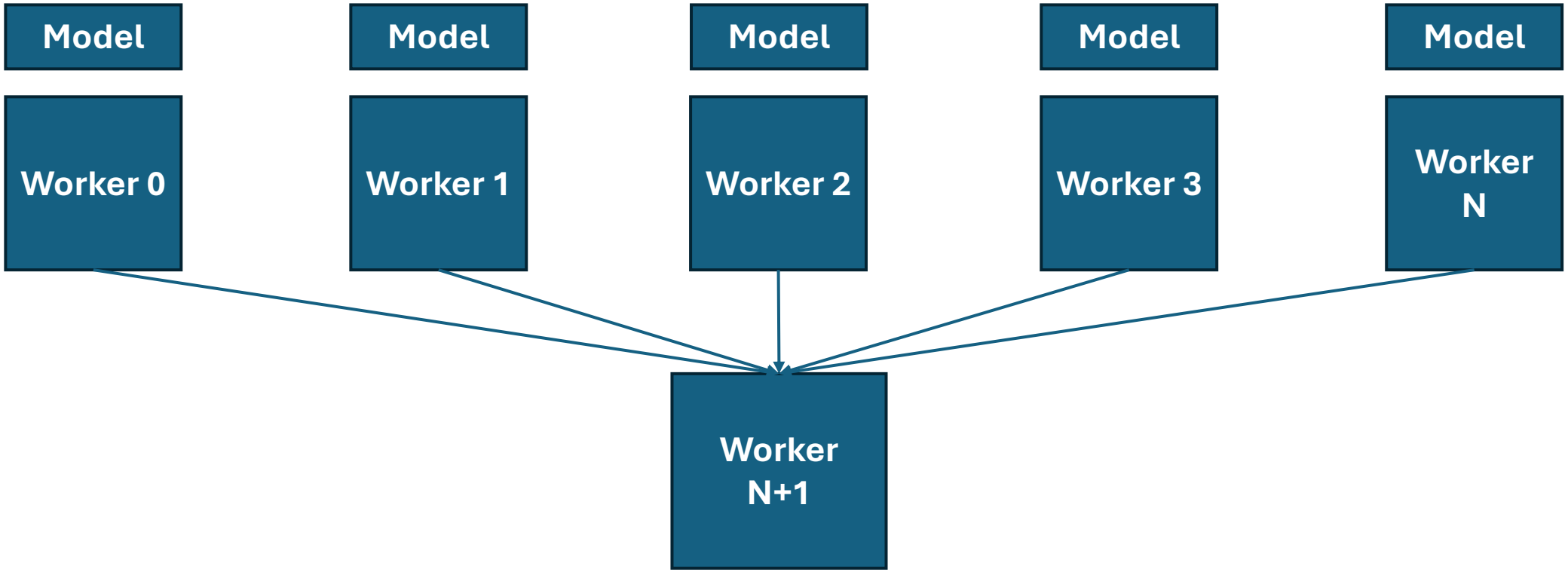
Example of Data Parallelism:

Traditional single machine DNN training



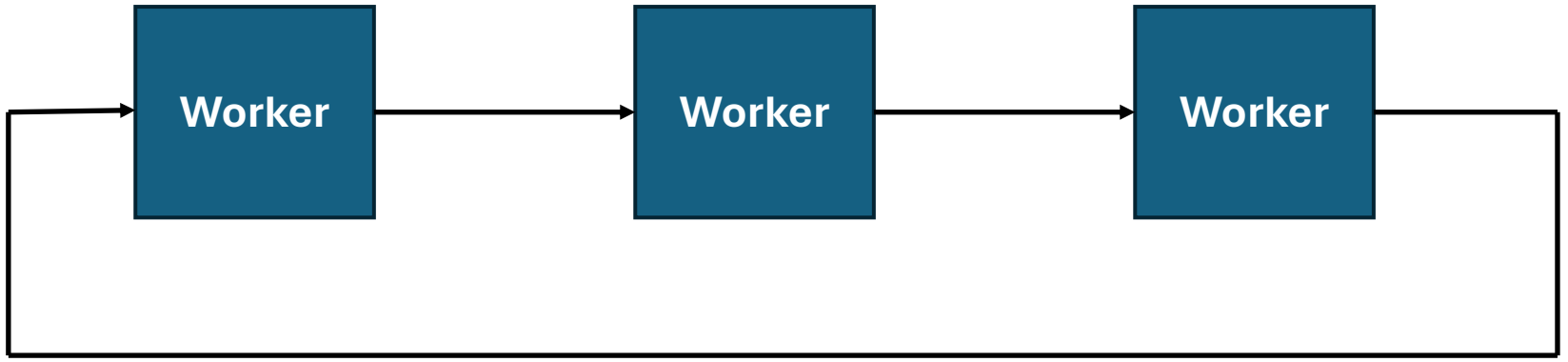
Data Parallelism – Parameter Server

Each worker holds the entire copy of the model and it trains its copy of the model on an assigned subset of the dataset.



The model is then synchronized through a worker called as a Parameter Server.

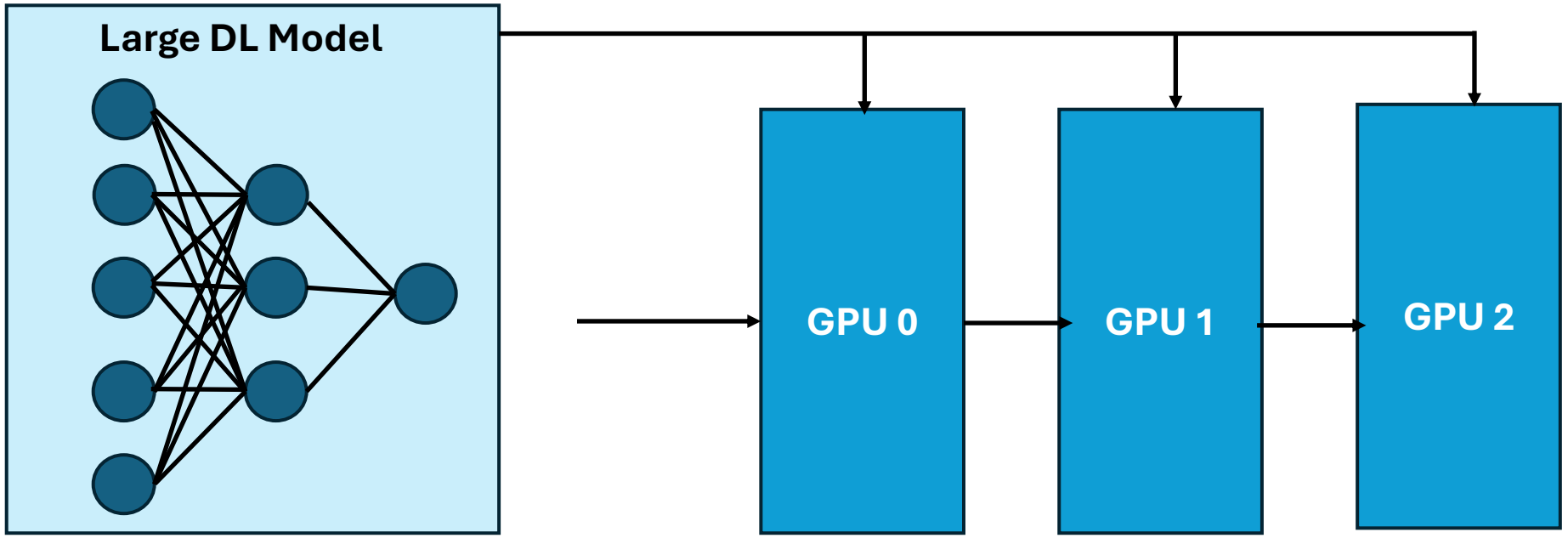
Data Parallelism – DDP (PyTorch Implementation)



- **Parameter server approach is not feasible when it comes to scalability, where the worker hosting parameter server becomes bottleneck**
- **DDP (Distributed Data Parallel) overcomes this issue by using asynchronous ring all reduce strategy, in which each worker transmits and receives reduced gradients from his immediate neighbor.**

Model Parallelism

- **As the model size is increasing rapidly, entire model cannot fit in single worker (GPU). Therefore, Model parallelism is proposed through which model layers are divided among workers and gradients/ activations are communicated among workers**



Model Parallelism: Pros and Cons

Pros:

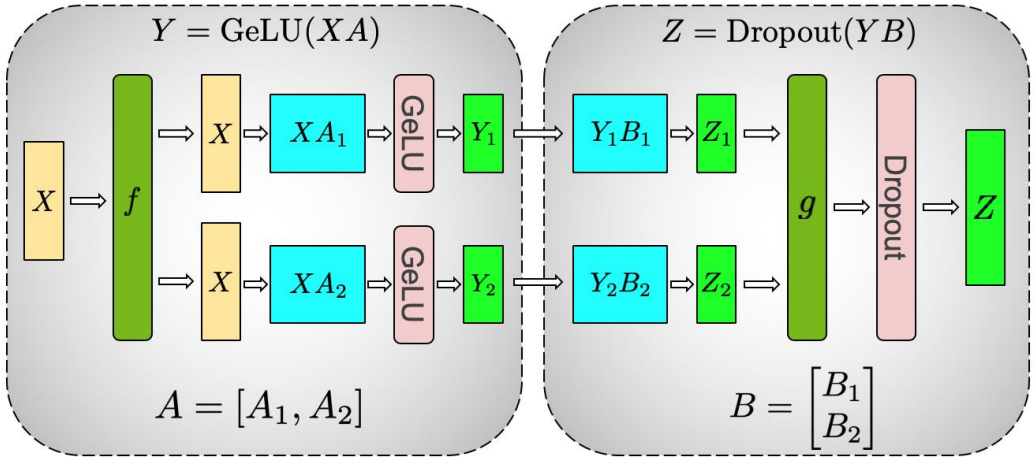
- Able to train large models in GPU memory by partitioning

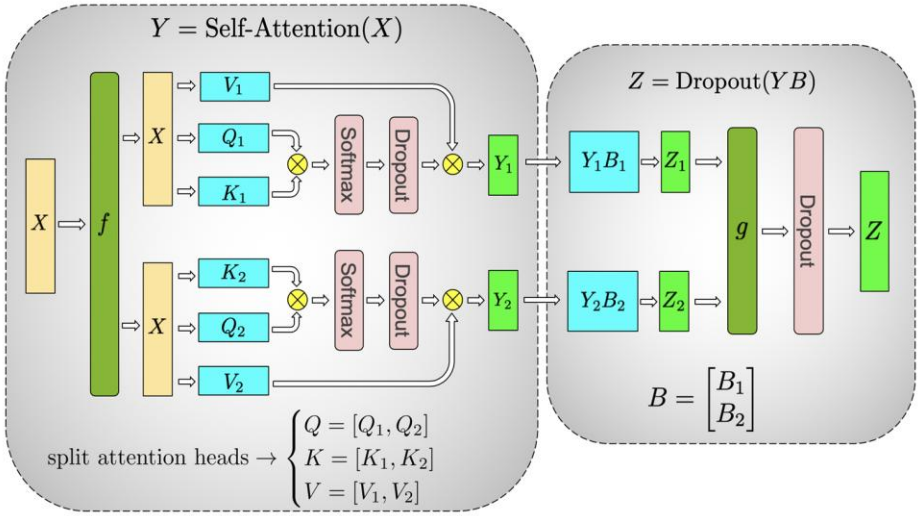
Cons:

- Requires large amount of communication of computations between layers
- High degree of model parallelism can create small matrix multiplications (GEMMs), potentially decreasing GPU utilization

Example of Model Parallelism:

Megatron-LM





Used to parallelize LLMs

Figure on the left shows their approach for parallelizing Fully Connected Layers; Splitting weights along the columns

Figure on the right shows their approach for parallelizing Self-Attention Layers; Splitting Query, Key, Value matrices along the columns

Pipeline Parallelism

Pipeline different stages of training

Layers of a model are striped over multiple GPUs

A batch is split into smaller micro-batches, and execution is pipelined across these microbatches.

Layers can be assigned to workers in various ways, and various schedules for the forward and backward passes of inputs can be used. The later assignment and scheduling strategy results in different performance tradeoffs.

Pros:

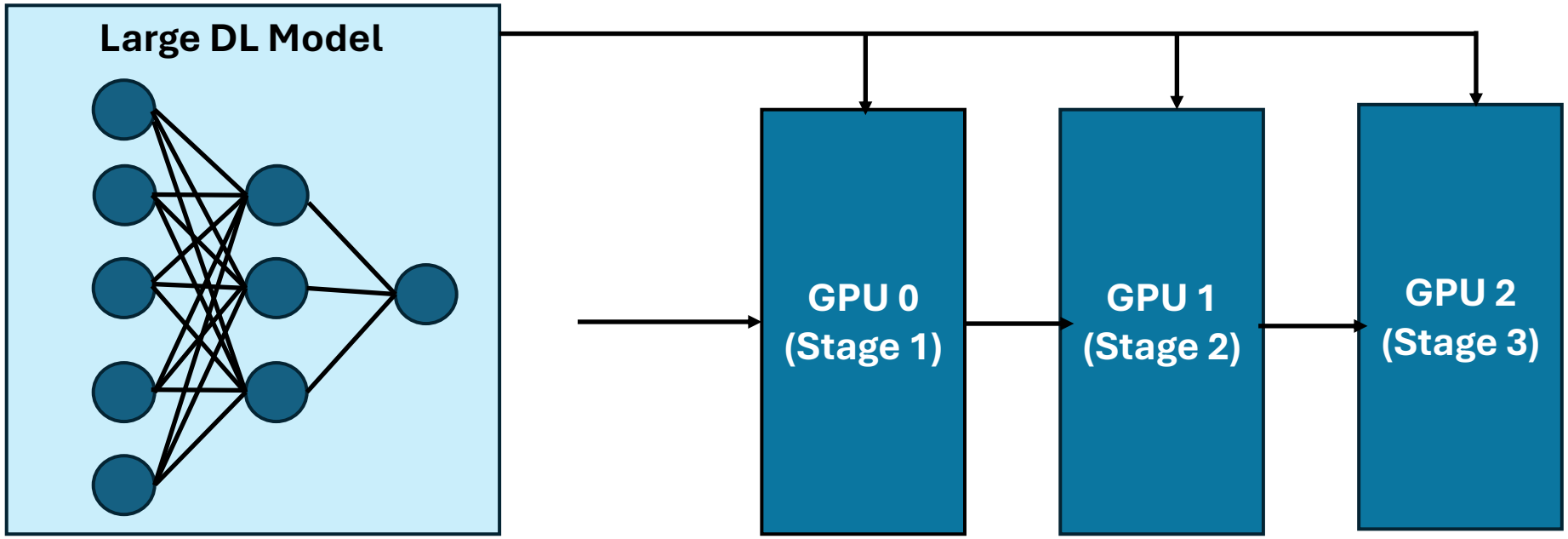
- Utilizes the compute resource completely during the training

Cons:

- Large amount of communication and synchronization

Pipeline Parallelism

- **Model Parallelism fails to utilize computing resources efficiently where only one GPU is actively computing at a given instance of time. There fore, a mini batch is further divided into micro batches and these micro batches are pipelined through GPU stages of the model.**



Different Pipeline Scheduling Strategies

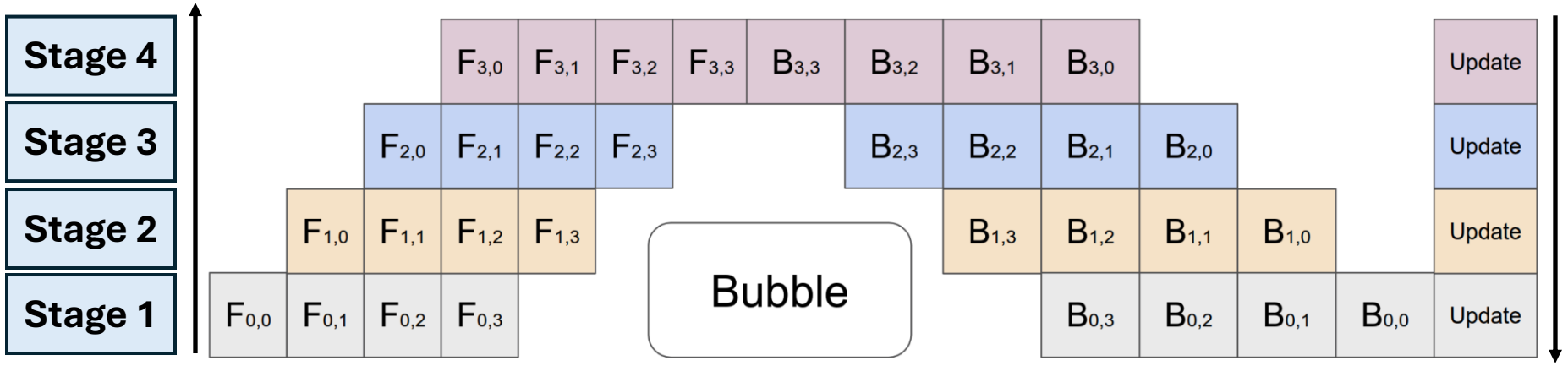


Figure – 1: (Gpipe) Micro batches are pipelined and FWD + BWD pass is conducted. Later, parameters are updated in each stage

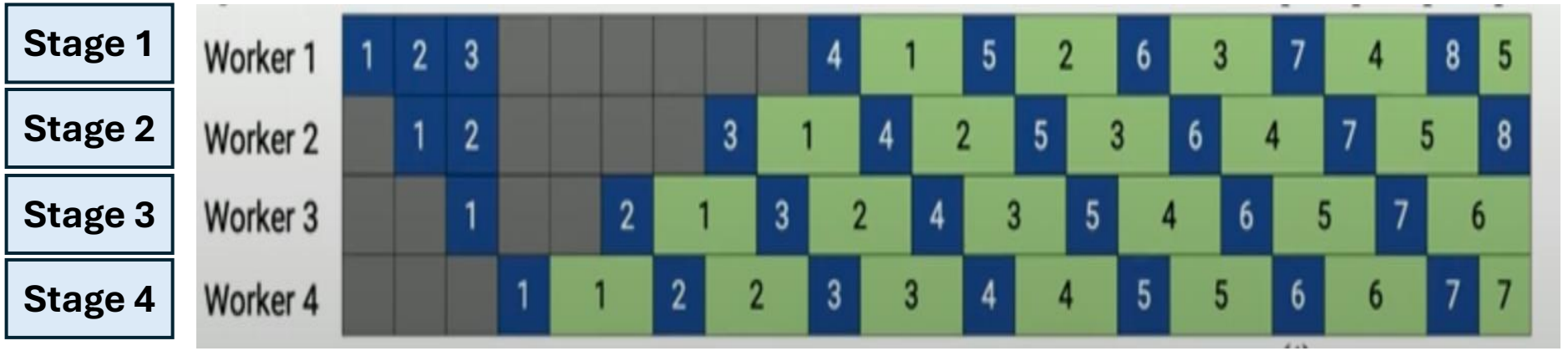
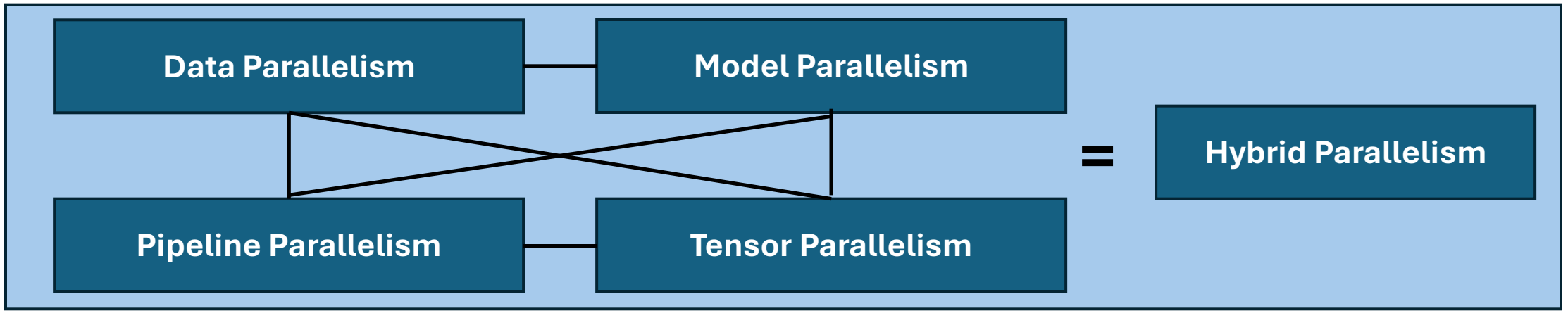
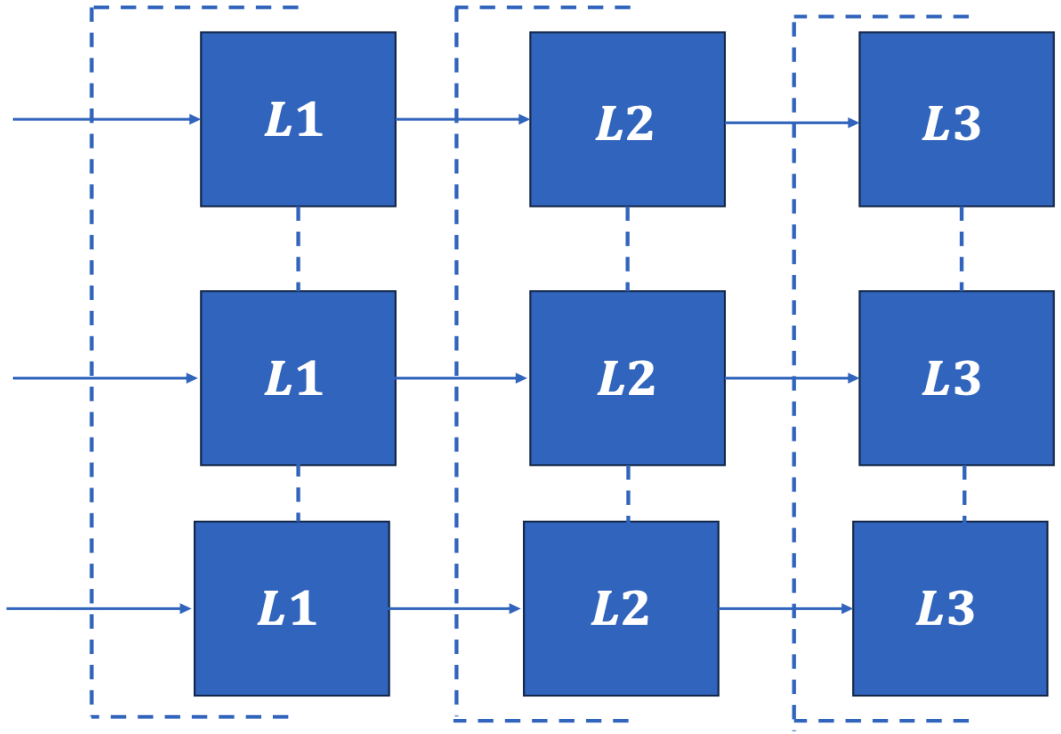


Figure – 2: (PipeDream) Micro batches are pipelined and FWD + BWD passes are scheduled through which FWD and BWD are pipelined.

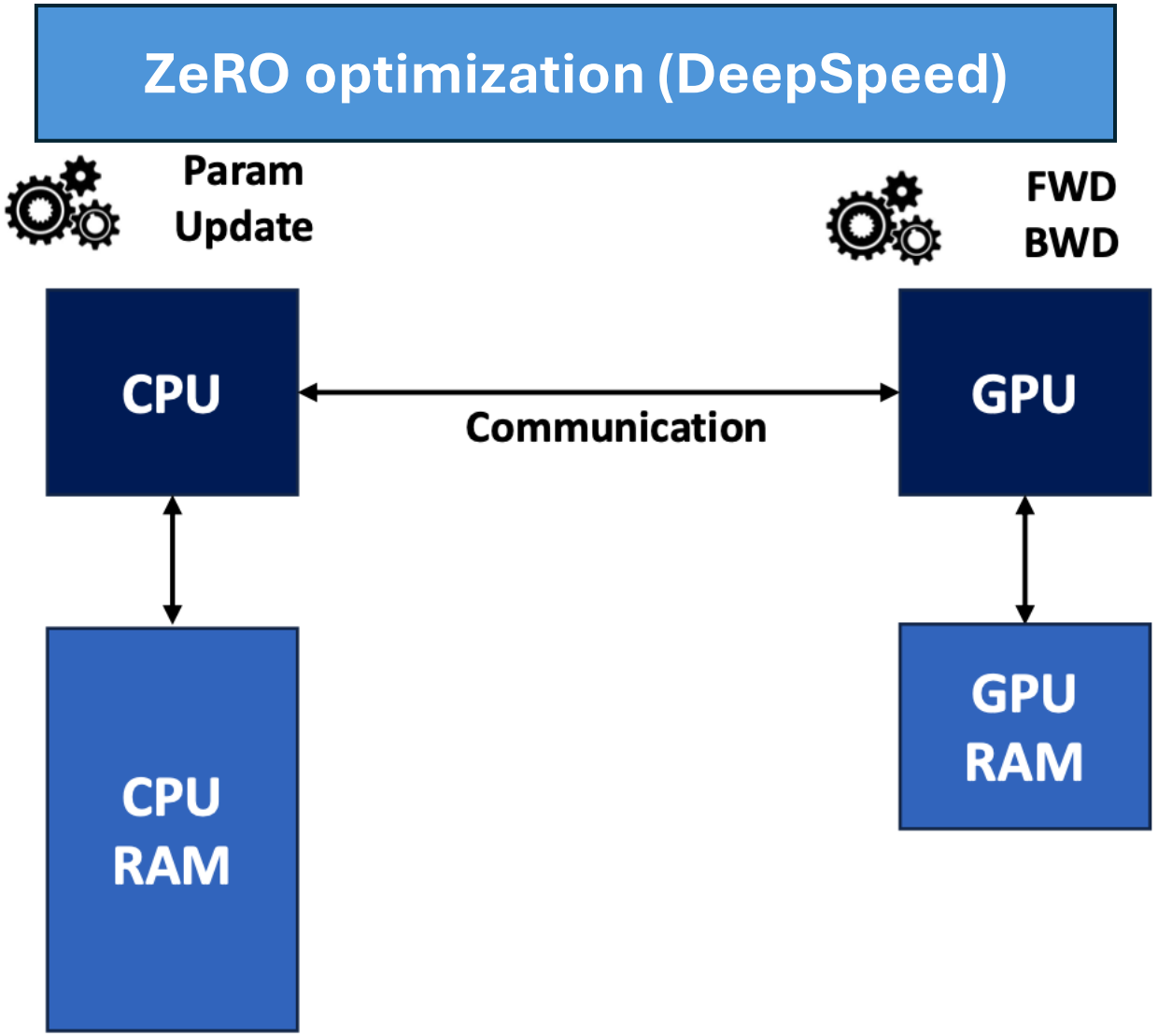
Hybrid Parallelization Strategies





As we see in the figure (to the left) model pipelined (from left to right) to implement pipeline parallelism . This setup is replicated to achieve data parallelism (top to bottom) and synchronization

CPU + GPU Parallelization Strategies



1) With rapid growth in model sizes, It was challenging to hold entire model on a stand alone GPU, to deal with this problem, DeepSpeed (Microsoft) proposed to hold all the parameters and optimizer states on the CPU's Large RAM.

2) The necessary parameters are sent to the GPU when required. GPU computes intensive operations like FWD and BWD passes involving Matrix Multiplications, and CPU handles communication and Parameter update of the parameters and optimizer states (CPU and GPU operations are parallelized).