



Model Parallel Training

Lei Li

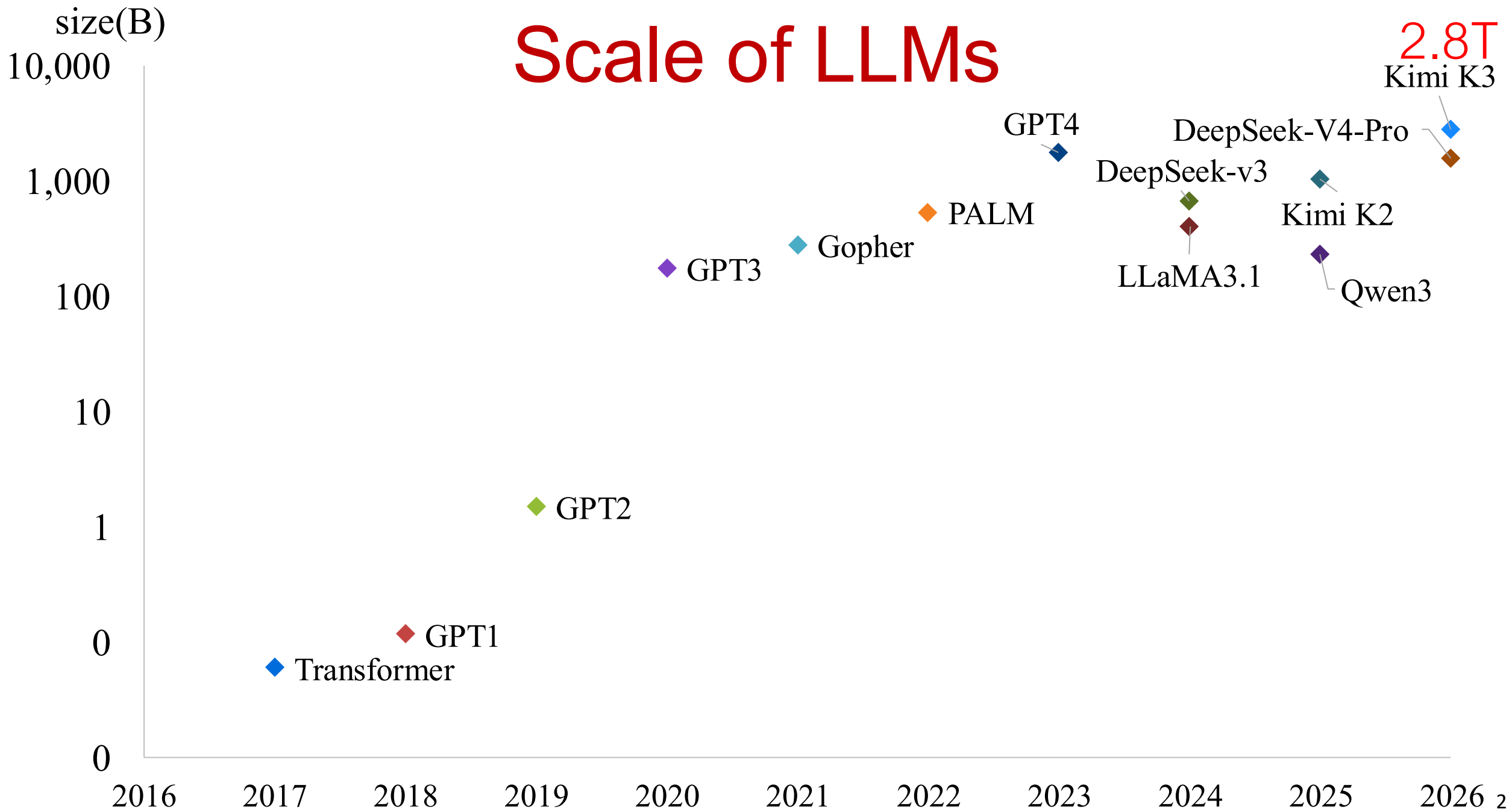


Language
Technologies
Institute

Carnegie Mellon University

School of Computer Science

Scale of LLMs

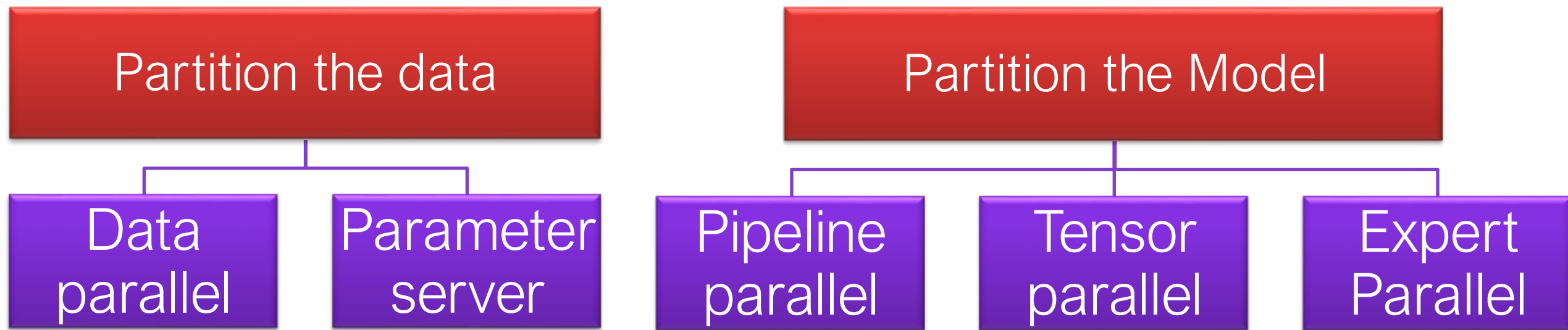


Memory Requirement for Training LLM

- How much memory is required to train Qwen 3.5 9B model? (using 16bit)
 - model parameters: 18GB
 - gradients: 18GB
 - optimizer states: roughly 108GB
 - activation memory (for all layers): 6GB for 8k context length, batch=1
 - total: 150GB
- H100 GPU memory: 80G

Strategies for Scalable Training

Motivation: LLM sizes increase exponentially. It is no longer possible to fit one LLM into a single GPU's memory.



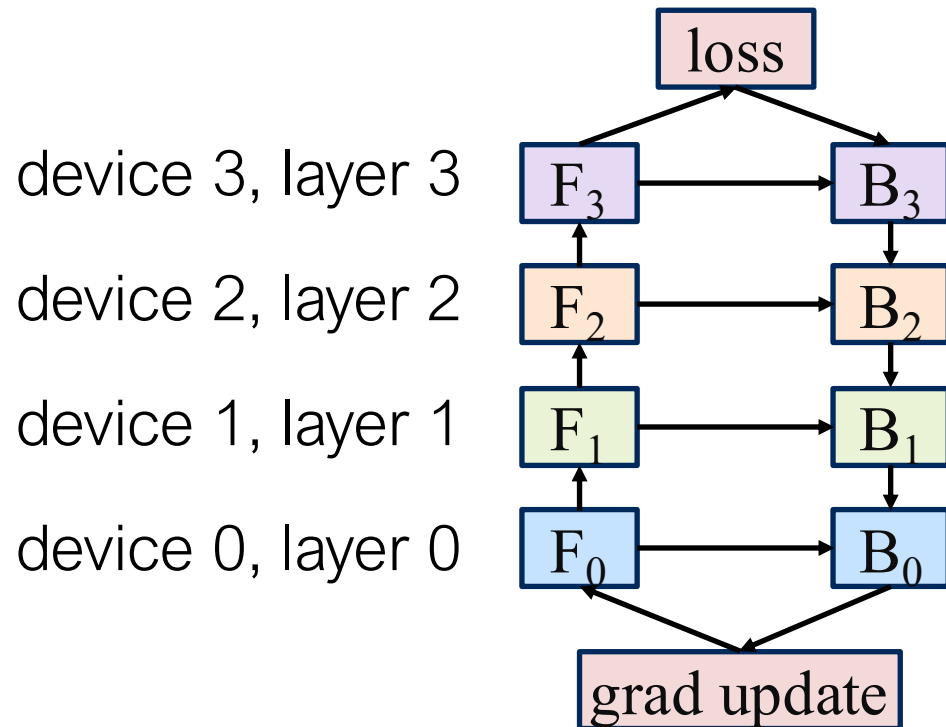
Today's Topic

- Pipeline Parallel Training
- Tensor Parallel Training

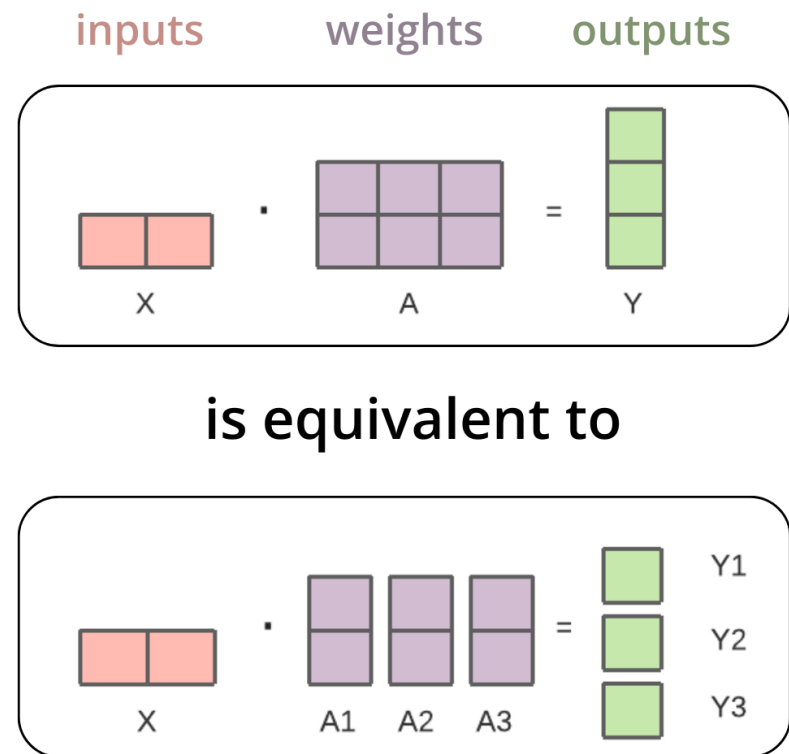
Model Parallel Training

Partition the model's computation (forward/backward/update) across multiple workers.

Distributed layer-wise computation



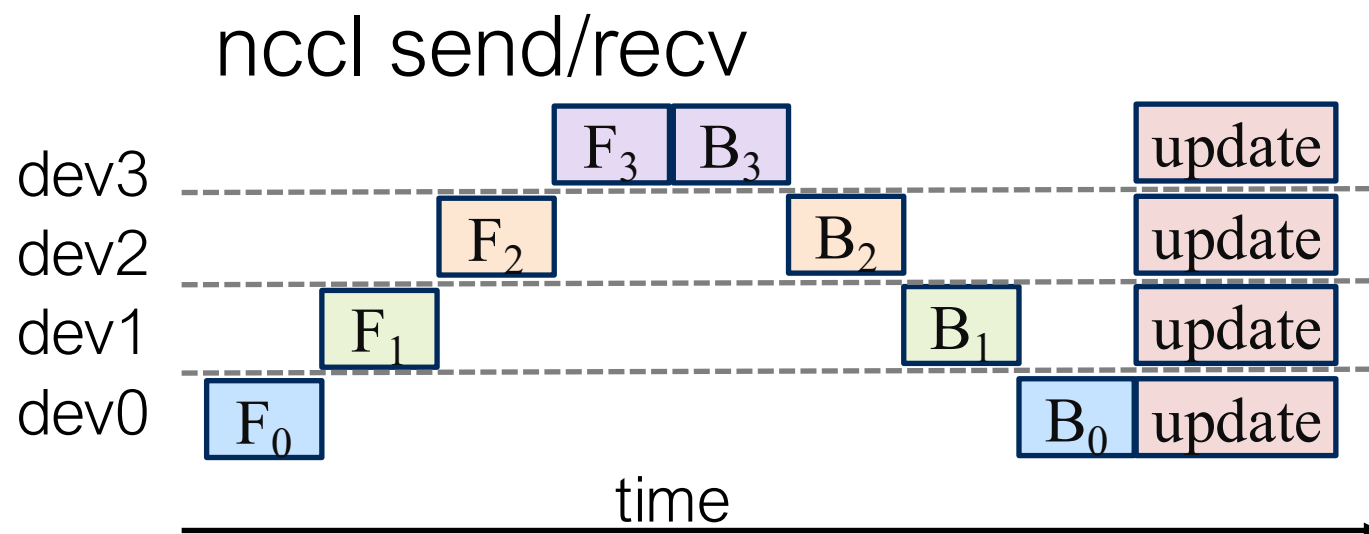
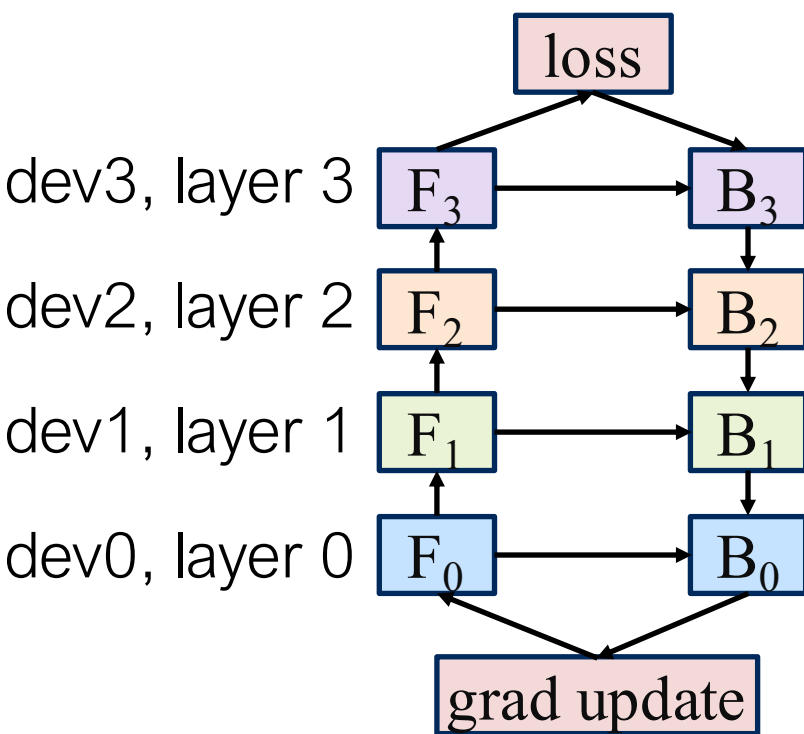
Distributed tensor computation



more advanced parallel training (expert parallelism) will be introduced later. 6

Pipeline Parallelism

Naïve Model Parallel: The model is distributed across multiple GPUs over layers.

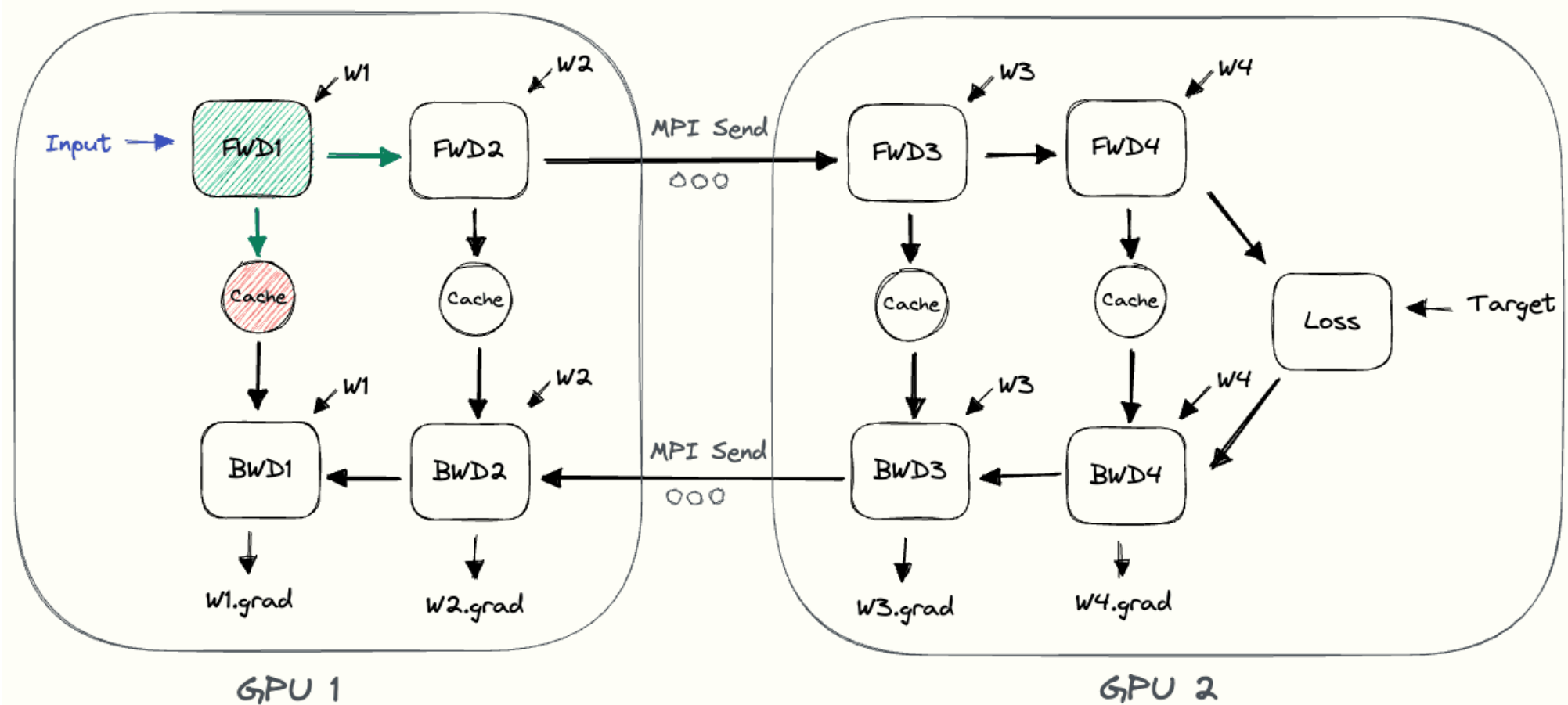


Any disadvantage?

all but one GPU is idle at any given moment!

Pipeline Parallelism Illustration

each device needs to calculate forward/backward,
memory: store params/grads/activations for the layers on each device



```

class ModelParallelResNet50(ResNet):
    def __init__(self, *args, **kwargs):
        super(ModelParallelResNet50, self).__init__(
            Bottleneck, [3, 4, 6, 3], num_classes=num_classes, *args, **kwargs)

        self.seq1 = nn.Sequential(
            self.conv1,
            self.bn1,
            self.relu,
            self.maxpool,

            self.layer1,
            self.layer2
        ).to('cuda:0')

        self.seq2 = nn.Sequential(
            self.layer3,
            self.layer4,
            self.avgpool,
        ).to('cuda:1')

        self.fc.to('cuda:1')

    def forward(self, x):
        x = self.seq2(self.seq1(x).to('cuda:1'))
        return self.fc(x.view(x.size(0), -1))

```

nccl send/recv

Limitations of Naïve Pipeline Parallel

- Low GPU utilization
 - at any point of time, only one device is working. others are idle.
- No interleaving of computation and communication
 - While sending intermediate results to next device, GPUs are idle.
- High memory demand
 - 1st GPU needs to store all activations until the whole batch completes.

Improving Pipeline Parallel: GPipe

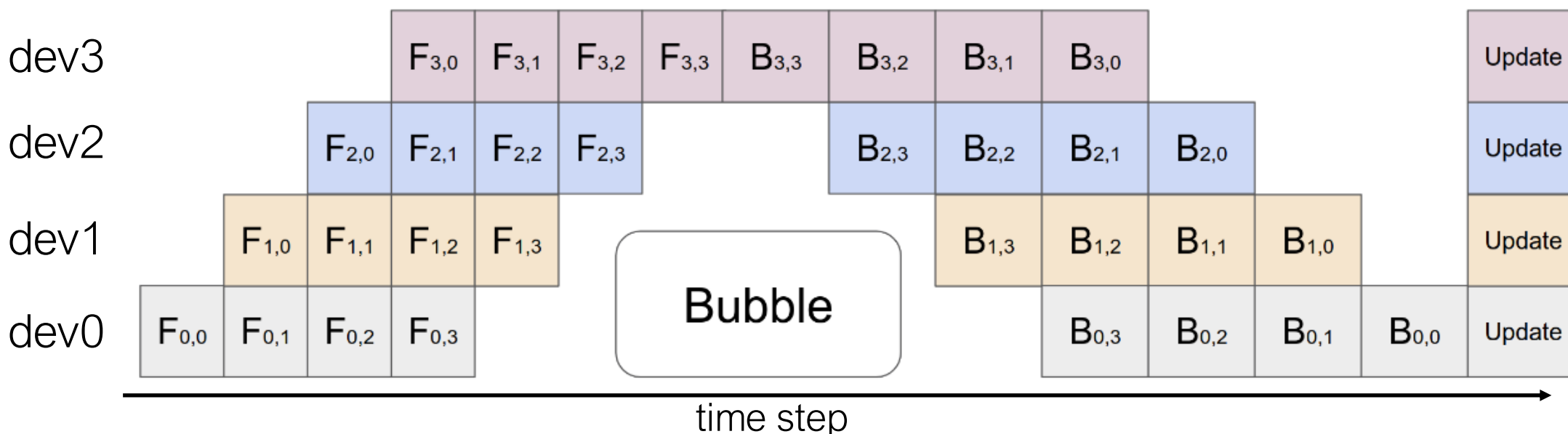
- Key idea: Divides input data batches into smaller micro-batches, pipelining microbatches.
- A batch is usually decided by GPU memory size and memory needed for one data sample's forward/backward
 - as large as possible to fill the GPU memory
- A micro-batch can be smaller

Pipeline Parallel via Micro-batching

GPipe: Divides input data mini-batches into smaller micro-batches.

Send each micro-batch results (layer output) immediately once completing (one small) forward.

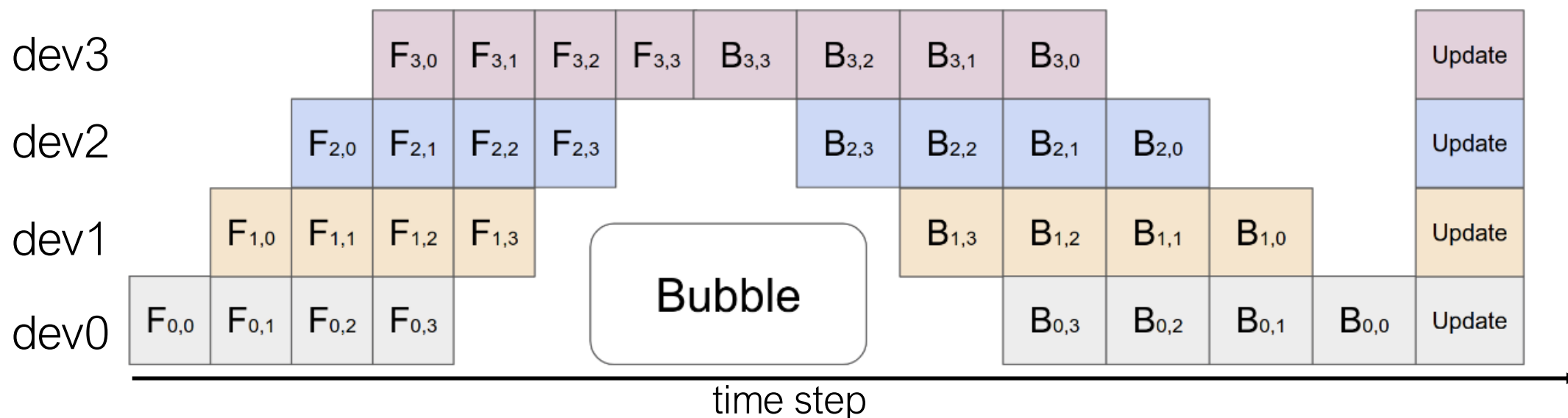
Finishing all forward before starting backward for micro-batches.



- (i) the number of model partitions K (*i.e.* number of devices)
- (ii) the number of micro-batches M
- (iii) the number of model layers: L

Microbatch Pipelining

GPipe: Divides input mini-batches into smaller micro-batches. During backward, recomputes forward, then send microbatch gradients immediately once ready.



Bubble overhead: $O(\frac{K-1}{M+K-1})$ could be negligible when $M > 4 \times K$

Communication overhead: transfer activation tensors at the partition boundaries

Peak activation memory: $O(N \times L) \rightarrow O(N + \frac{L}{K} \times \frac{N}{M})$ with gradient checkpoint (later)

GPipe Performance

Normalized training throughput using Gpipe with different # of partitions K and different # of micro-batches M on TPUs and GPUs without high-speed interconnect.

TPU	AmoebaNet			Transformer		
$K =$	2	4	8	2	4	8
$M = 1$	1	1.13	1.38	1	1.07	1.3
$M = 4$	1.07	1.26	1.72	1.7	3.2	4.8
$M = 32$	1.21	1.84	3.48	1.8	3.4	6.3

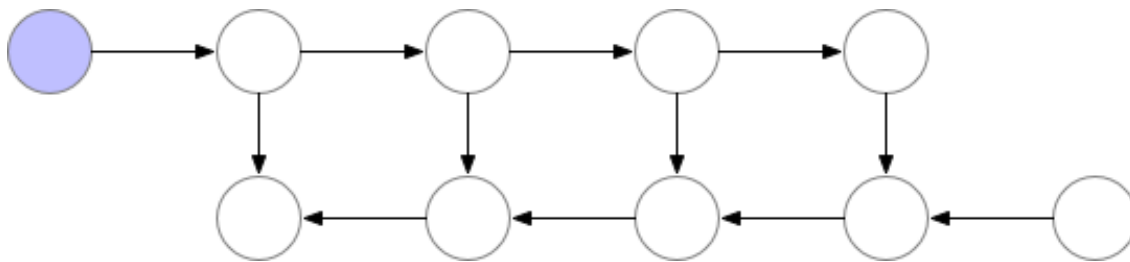
GPU	AmoebaNet			Transformer		
$K =$	2	4	8	2	4	8
$M = 32$	1	1.7	2.7	1	1.8	3.3

Reduce PP Memory Cost: Gradient Checkpointing

Re-materialization

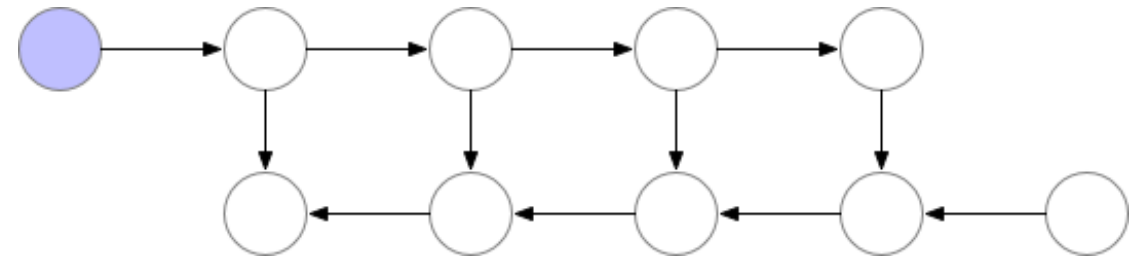
- Forward pass: each accelerator only stores output activations
- Backward pass: the k -th accelerator recomputes the composite forward function F_k

Vanilla backprop



- Memory for activations: $O(n)$
- Node computation: $O(n)$

Memory poor backprop



- Memory for activations: $O(1)$
- Node computation: $O(n^2)$

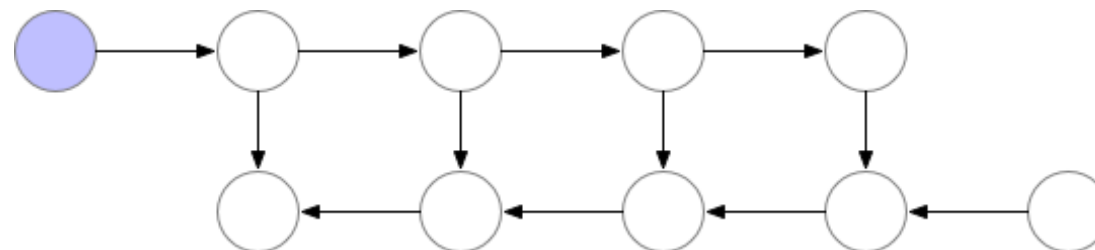
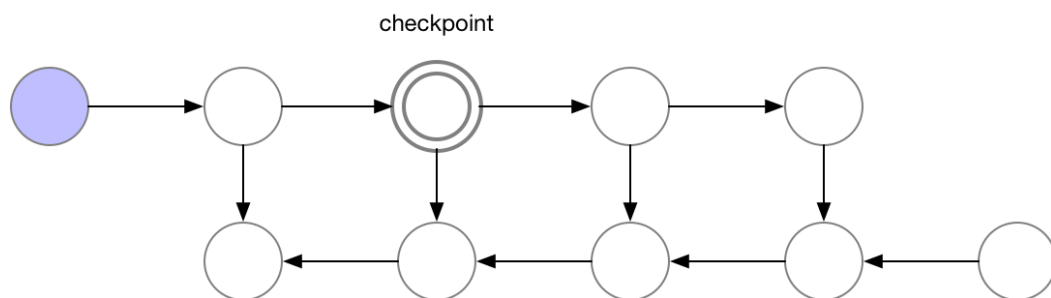
[1] Chen, Tianqi, et al. "Training deep nets with sublinear memory cost." arXiv preprint arXiv:1604.06174 (2016).

[2] <https://github.com/cybertronai/gradient-checkpointing>

Gradient Checkpointing

Gradient checkpoint

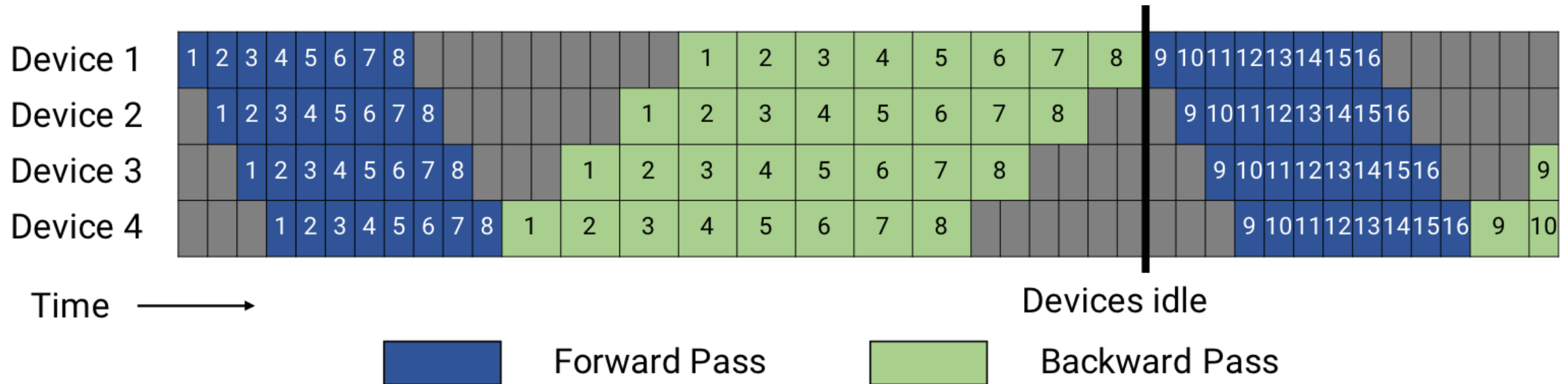
- Cash the activations of every $\sqrt{n}$ layers
- Memory for activations: $O(n)$
- *Node computation*: $O(\sqrt{n} * \sqrt{n}) = O(n)$



[1] Chen, Tianqi, et al. "Training deep nets with sublinear memory cost." arXiv preprint arXiv:1604.06174 (2016).

[2] <https://github.com/cybertronai/gradient-checkpointing>

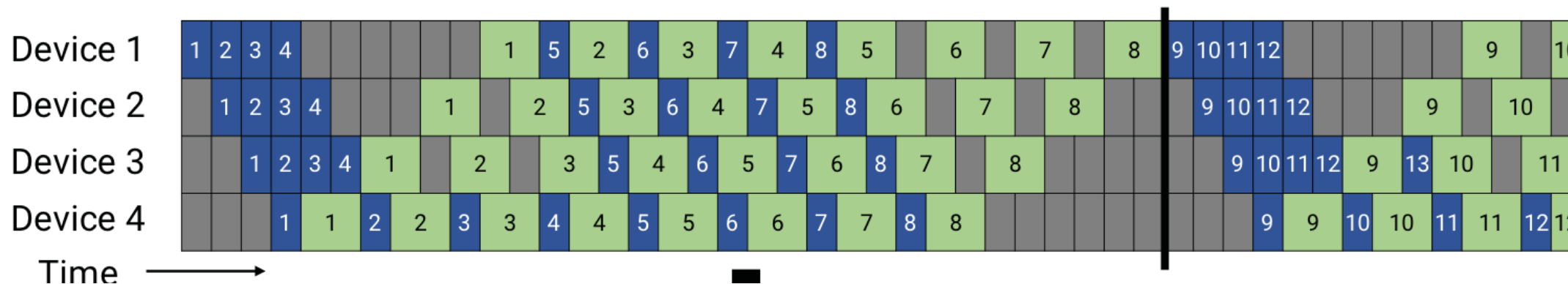
Limitations of Pipeline Parallel



finishing all forward before backward on micro-batches.
number of micro-batches in-flight (completed forward but not backward): 8 in this example,
need to store all the activations for these micro-batches

Improving Pipeline Parallel with 1F1B flush

- PipeDream-Flush 1F1B – start backward as soon as possible



- benefit: reduce memory cost for storing the activations
- number of micro-batches in flight (completed forward but not backward): at most 4 (assuming 1B=2F) vs 8 in GPipe.

Further Improving Pipeline Parallel by Chunking Model Layers and Interleaving Stages

	Chunk 1 Layers	Chunk 2 Layers	
Device 1	1, 2	9, 10	
Device 2	3, 4	11, 12	
Device 3	5, 6	13, 14	
Device 4	7, 8	15, 16	

Complete smaller computation for each chunk and 2 stages of F/B



Implementing GPipe Parallelism

```
def minibatch_steps(self):  
    yield [ZeroGrad()]  
  
    # STAGE 1: First, we FWD all microbatches  
    for microbatch_id in range(self.num_micro_batches):  
        yield self.steps_FWD_microbatch(microbatch_id)  
  
    # at this position, all microbatches are in flight and  
    # memory demand is highest  
  
    # STAGE 2: Then, we BWD all microbatches  
    for microbatch_id in reversed(range(self.num_micro_batches)):  
        yield from self.steps_BWD_microbatch(microbatch_id)  
  
    # updating the weights is the last step of processing any batch  
    yield [OptimizerStep()]
```

full code in <https://github.com/siboehm/shallowspeed>

```
def steps_FWD_microbatch(self, microbatch_id):
    cmds = []
    if self.is_first_stage:
        # first pipeline stage loads data from disk
        cmds.append(LoadMicroBatchInput(microbatch_id=microbatch_id))
    else:
        # all other stages receive activations from prev pipeline stage
        cmds.append(RecvActivations())

    cmds.append(Forward(microbatch_id=microbatch_id))

    if not self.is_last_stage:
        # all but the last pipeline stage send their output to next stage
        cmds.append(SendActivations())
    return cmds
```

non-blocking

```

def steps_BWD_microbatch(self, microbatch_id):
    cmds = []
    if self.is_last_stage:
        # last pipeline stage loads data from disk
        cmds.append(LoadMicroBatchTarget(microbatch_id=microbatch_id))
    else:
        # all other stages wait to receive grad from prev stage
        cmds.append(RecvOutputGrad())

    # the first microBatch is the lastest one that goes through backward pass
    if self.is_first_microbatch(microbatch_id):
        # interleaved backprop and AllReduce during last microBatch of BWD
        cmds.append(BackwardGradAllReduce(microbatch_id=microbatch_id))
    else:
        cmds.append(BackwardGradAcc(microbatch_id=microbatch_id))

    if not self.is_first_stage:
        # all but last pipeline stage send their input grad to prev stage
        cmds.append(SendInputGrad())
    yield cmds

```

Pipeline Parallelism in pytorch

`torch.distributed.pipelining`

- It consists of two stages
 - build PipelineStage
 - manually splitting the model
 - splitting model automatically
 - use PipelineSchedule for execution

```

class Transformer(nn.Module):
    def __init__(self, model_args: ModelArgs):
        super().__init__()
        self.tok_embeddings = nn.Embedding(...)
        # Using a ModuleDict lets us delete layers without affecting names, ensuring checkpoints will correctly save and load.
        self.layers = torch.nn.ModuleDict()
        for layer_id in range(model_args.n_layers):
            self.layers[str(layer_id)] = TransformerBlock(...)
        self.output = nn.Linear(...)

    def forward(self, tokens: torch.Tensor):
        # Handling layers being 'None' at runtime enables easy pipeline splitting
        h = self.tok_embeddings(tokens) if self.tok_embeddings else tokens
        for layer in self.layers.values():
            h = layer(h, self.freqs_cis)
        h = self.norm(h) if self.norm else h
        output = self.output(h).float() if self.output else h
        return output

```

<https://pytorch.org/docs/main/distributed.pipelining.html>

```
from torch.distributed.pipelining import PipelineStage
```

```
with torch.device("meta"):
```

```
    assert num_stages == 2, "This is a simple 2-stage example"
```

```
    # we construct the entire model, then delete the parts we do not need for this stage # in practice, this can  
be done using a helper function that automatically divides up layers across stages.
```

```
    model = Transformer()
```

```
    if stage_index == 0: # prepare the first stage model
```

```
        del model.layers["1"]
```

```
        model.norm = None
```

```
        model.output = None
```

```
    elif stage_index == 1: # prepare the second stage model
```

```
        model.tok_embeddings = None
```

```
        del model.layers["0"]
```

```
    stage = PipelineStage(model, stage_index, num_stages, device)
```

```
from torch.distributed.pipelining import ScheduleGPipe  
# Create a schedule  
schedule = ScheduleGPipe(stage, n_microbatches)  
# Input data (whole batch)  
x = torch.randn(batch_size, in_dim, device=device)  
# Run the pipeline with input `x` # `x` will be divided into microbatches automatically  
if rank == 0:  
    schedule.step(x)  
else:  
    output = schedule.step()
```

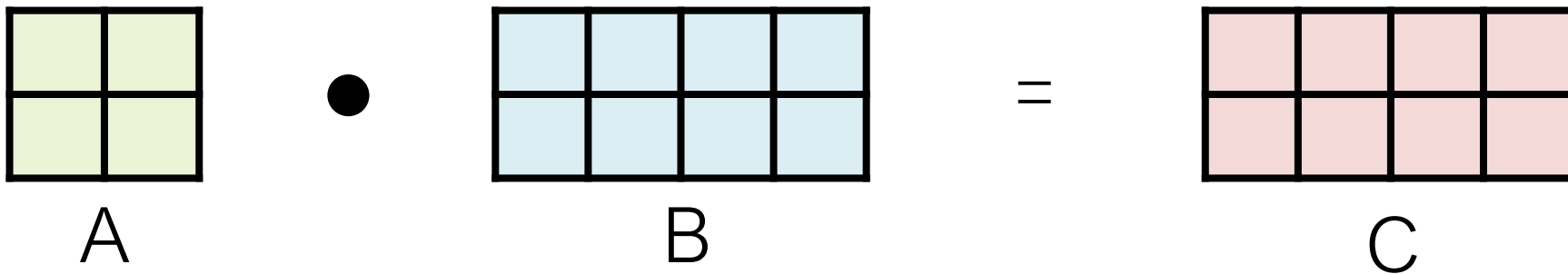
Today's Topic

- Pipeline Parallel Training

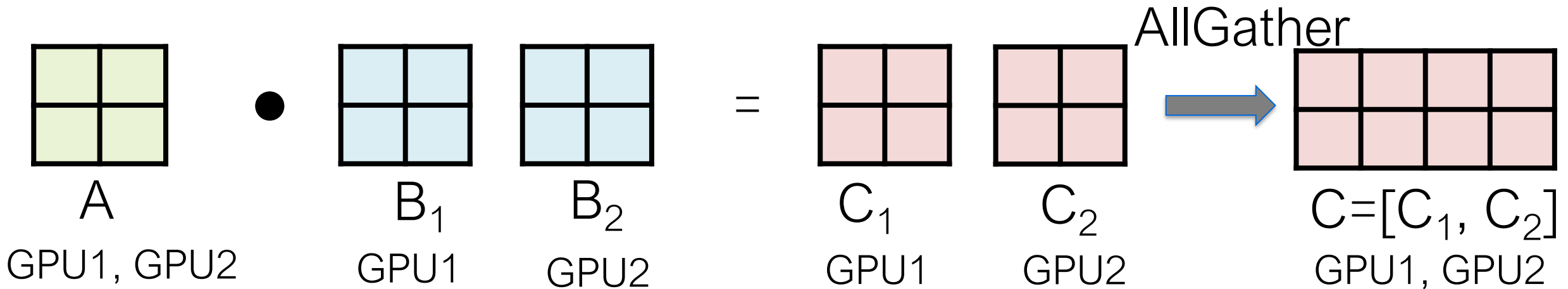
-  • Tensor Parallel Training

Tensor Parallelism – splitting the matrix computation

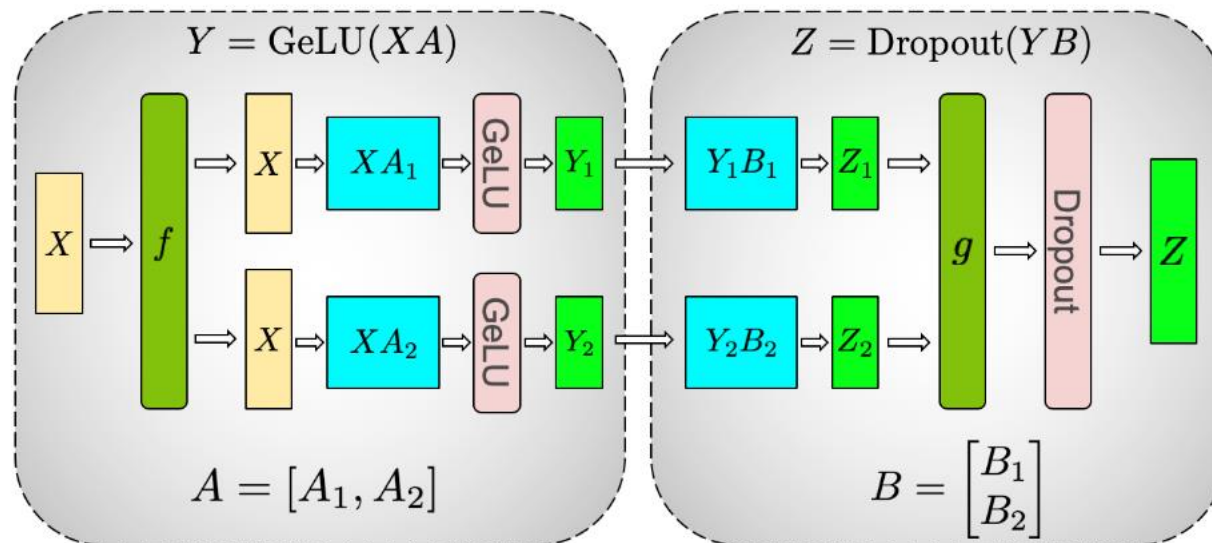
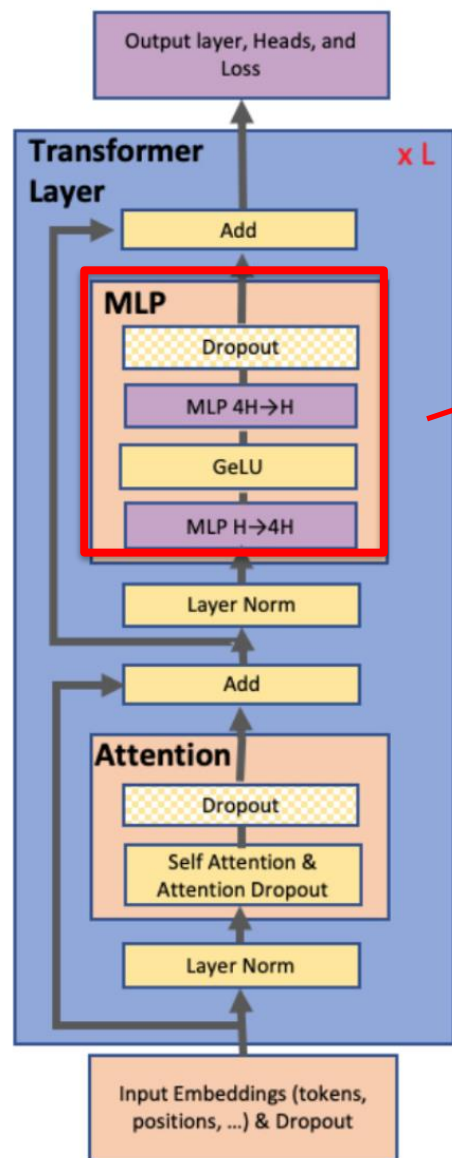
Matrix multiplication on 1 GPU



Matrix multiplication on 2 GPUs



Tensor Parallelism for FFN (big mat mul)



$$Y = \text{GeLU}(XA)$$

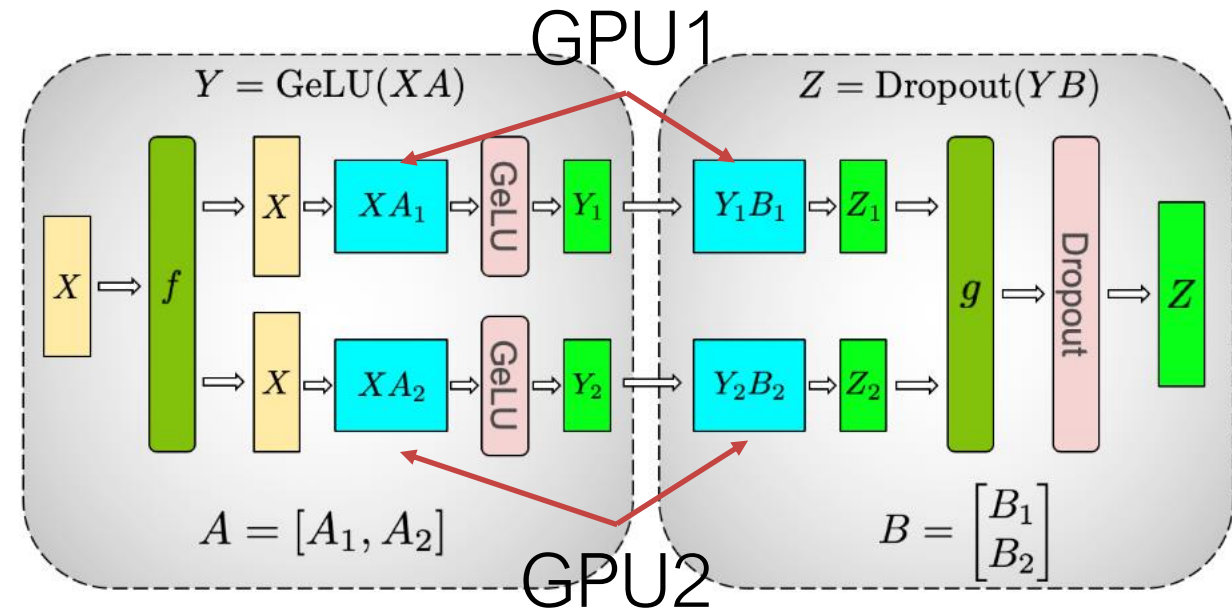
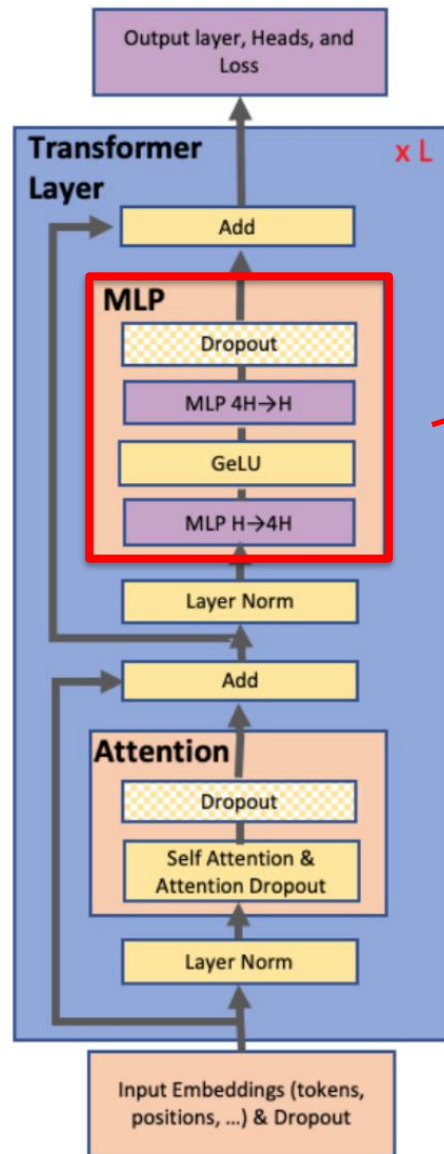
Splitting X and A

$$X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}, A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \quad Y = \text{GeLU}(X_1 A_1 + X_2 A_2)$$

$$\text{GeLU}(X_1 A_1 + X_2 A_2) \neq \text{GeLU}(X_1 A_1) + \text{GeLU}(X_2 A_2)$$

All-reduce (to compute XA) is needed !

Tensor Parallelism for FFN



Splitting weight A

$$A = [A_1, A_2],$$

weight B

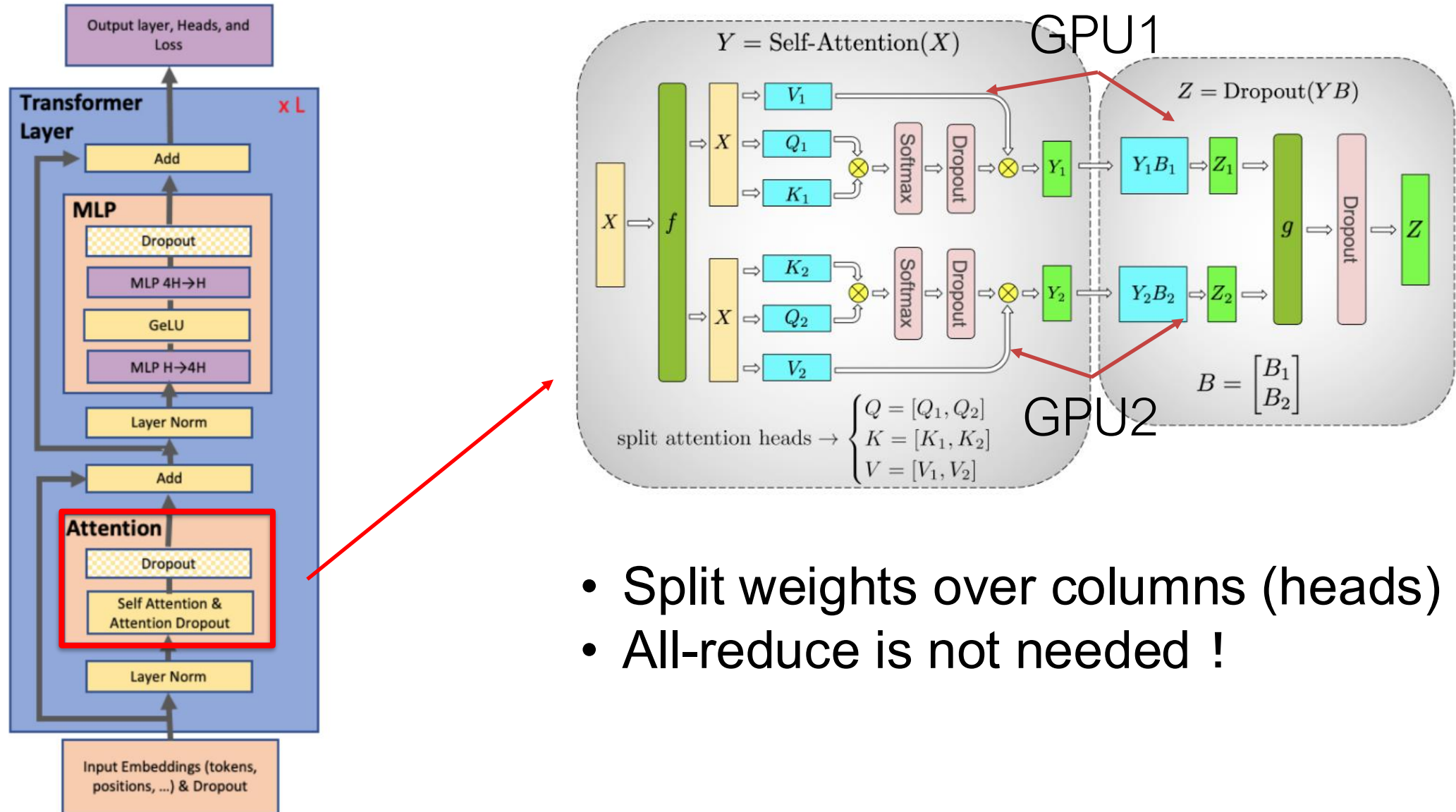
$$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix},$$

$$[Y_1, Y_2] = [\text{GeLU}(X \cdot A_1), \text{GeLU}(X \cdot A_2)],$$

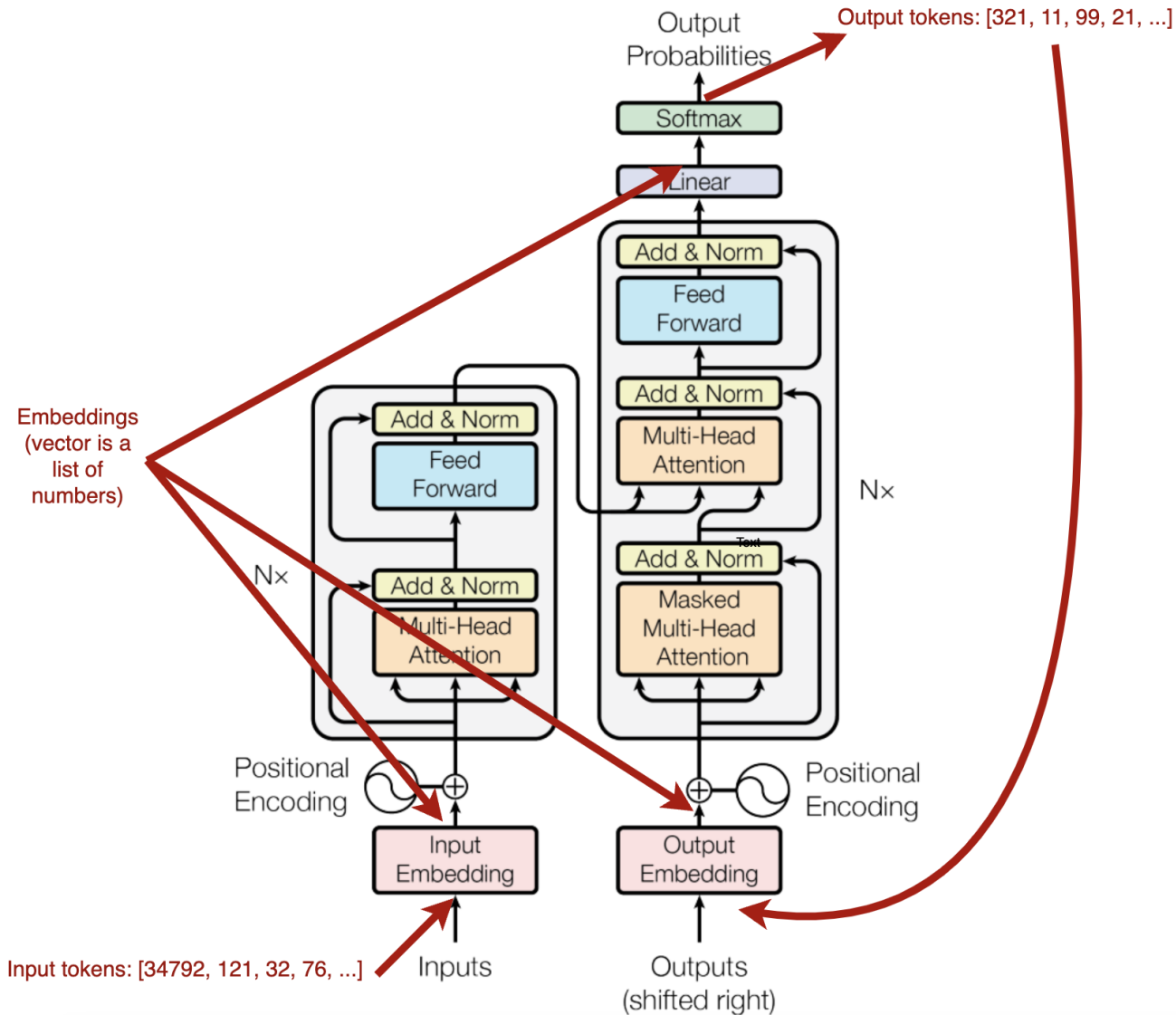
All-reduce is **not** needed (for Y) !

$$Z = Y_1 \cdot B_1 + Y_2 \cdot B_2$$

Tensor Parallelism for Self-Attention

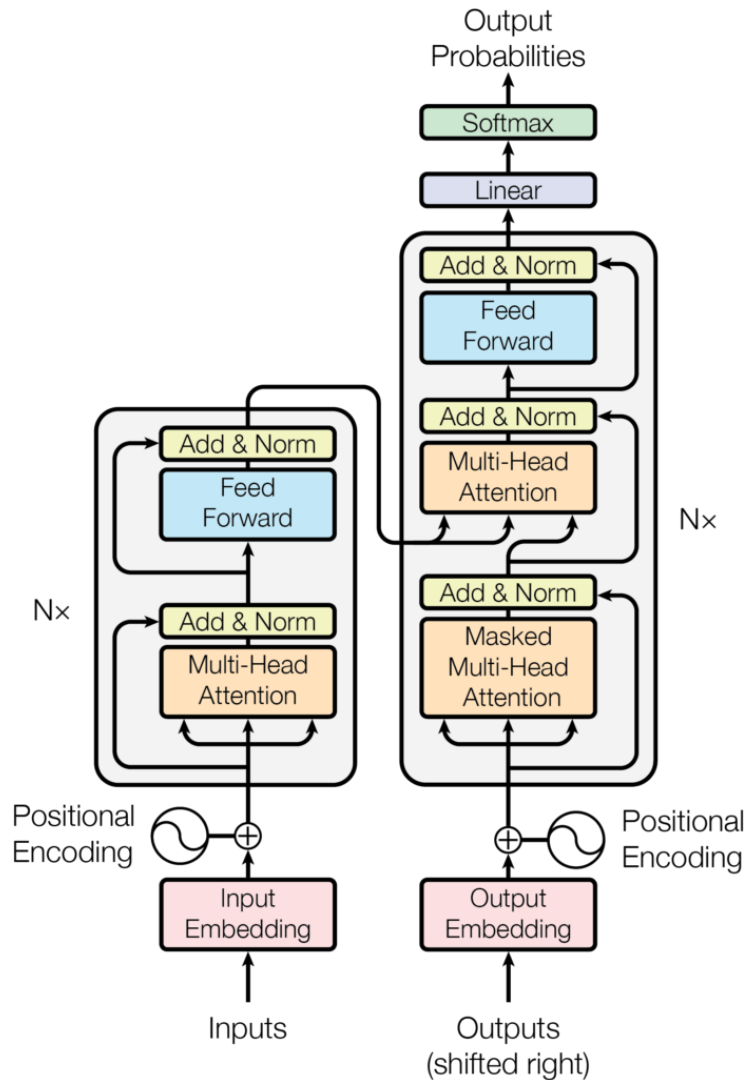


Tensor Parallelism - Embeddings



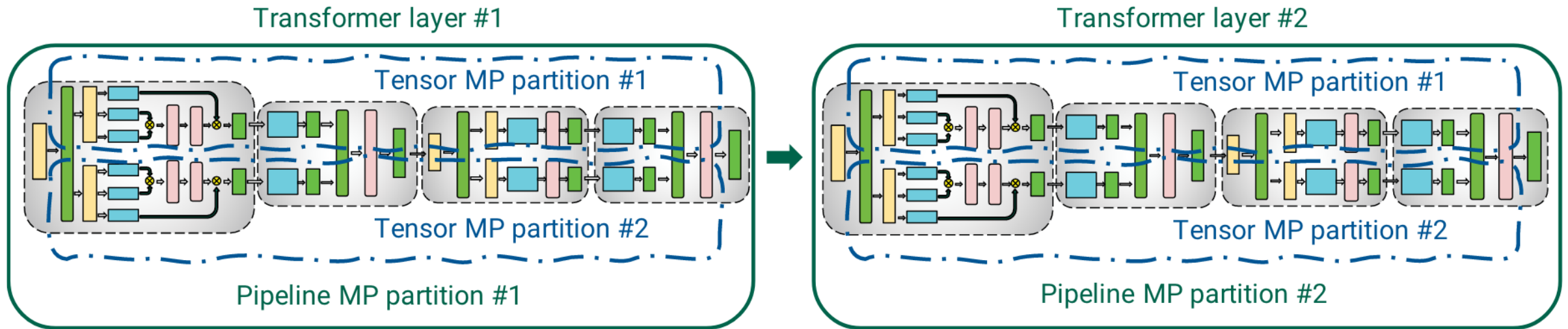
- Input embedding
 - Split over columns
$$E = [E_1, E_2] \text{ (column-wise)}$$
 - all-reduce is required
- Output embedding
 - Split over columns
$$\text{GEMM } [Y_1, Y_2] = [X E_1, X E_2]$$
 - Fuse outputs with cross-entropy loss (huge reduction in communication)
 - all-gather is needed

Tensor Parallelism



- Layer normalization, dropout, residual connections
 - Duplicate across GPUs
- Each model parallel worker optimizes its own set of parameters

Combination of Pipeline and Tensor Model Parallelism



#PP and #TP depend on model architecture and GPU server config

Combination of Pipeline and Tensor Model Parallelism

- Takeaway #1: When considering different forms of model parallelism, tensor model parallelism should generally be used up to degree g when using g -GPU servers, and then pipeline model parallelism can be used to scale up to larger models across servers → TP for in-node parallel computing

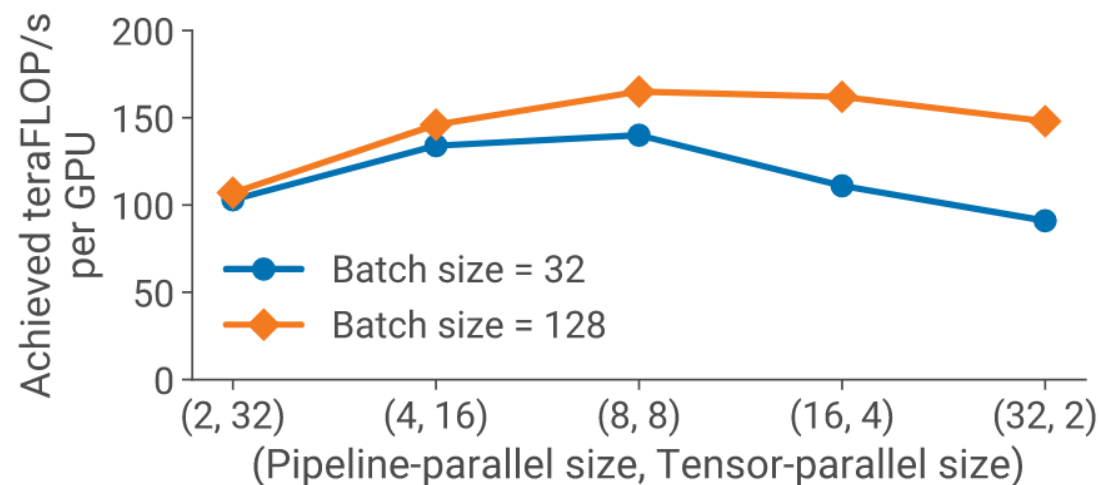


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

Model Parallel + Data Parallel

- Takeaway #2: When using data and model parallelism, a total model-parallel size of $M = t \cdot p$ should be used so that the model's parameters and intermediate metadata fit in GPU memory; data parallelism can be used to scale up training to more GPUs.

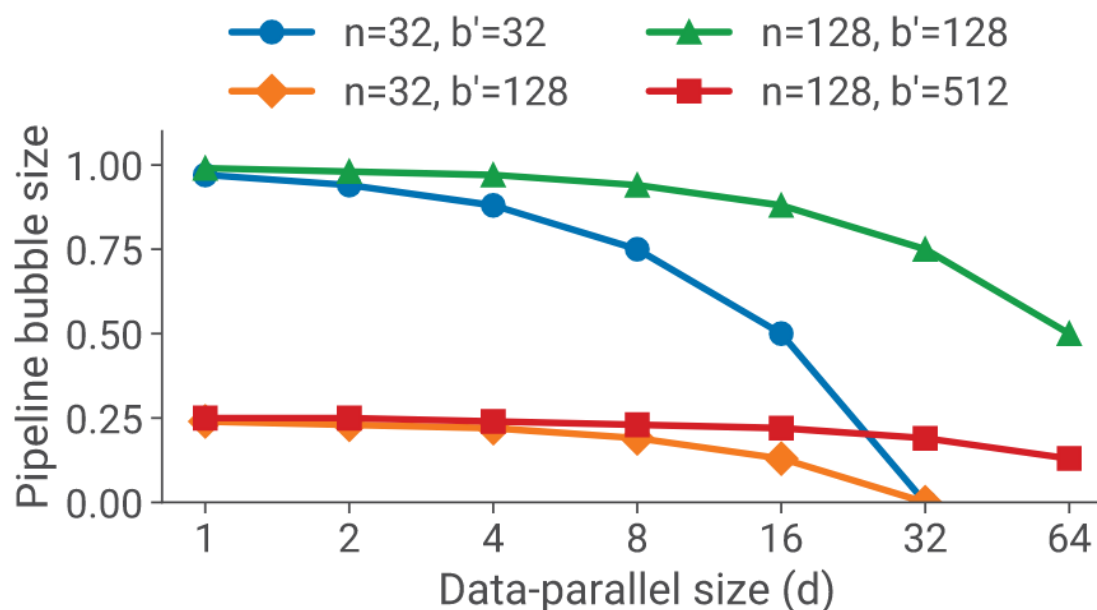


Figure 6: Fraction of time spent idling due to pipeline flush (pipeline bubble size) versus data-parallel size (d), for different numbers of GPUs (n) and ratio of batch size to microbatch size ($b' = B/b$).

Summary of Model Parallel Training

- Pipeline Parallelism
 - split by layers (horizontal split)
 - eliminate the bubbles (idle)
 - interleaving forward/backward
 - chunking layers and interleaving stages
- Tensor Parallelism
 - split the matrix computation over multiple GPUs
 - usually within a single node

Quiz

Reading for Next Lecture

- Lepikhin et al. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. ICLR 2021.
- Fedus et al. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. JMLR 2022.
- Dai et al. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models, ACL 2024.