LLM Sys

11868 LLM Systems Distributed Training – Model Parallelism

Lei Li

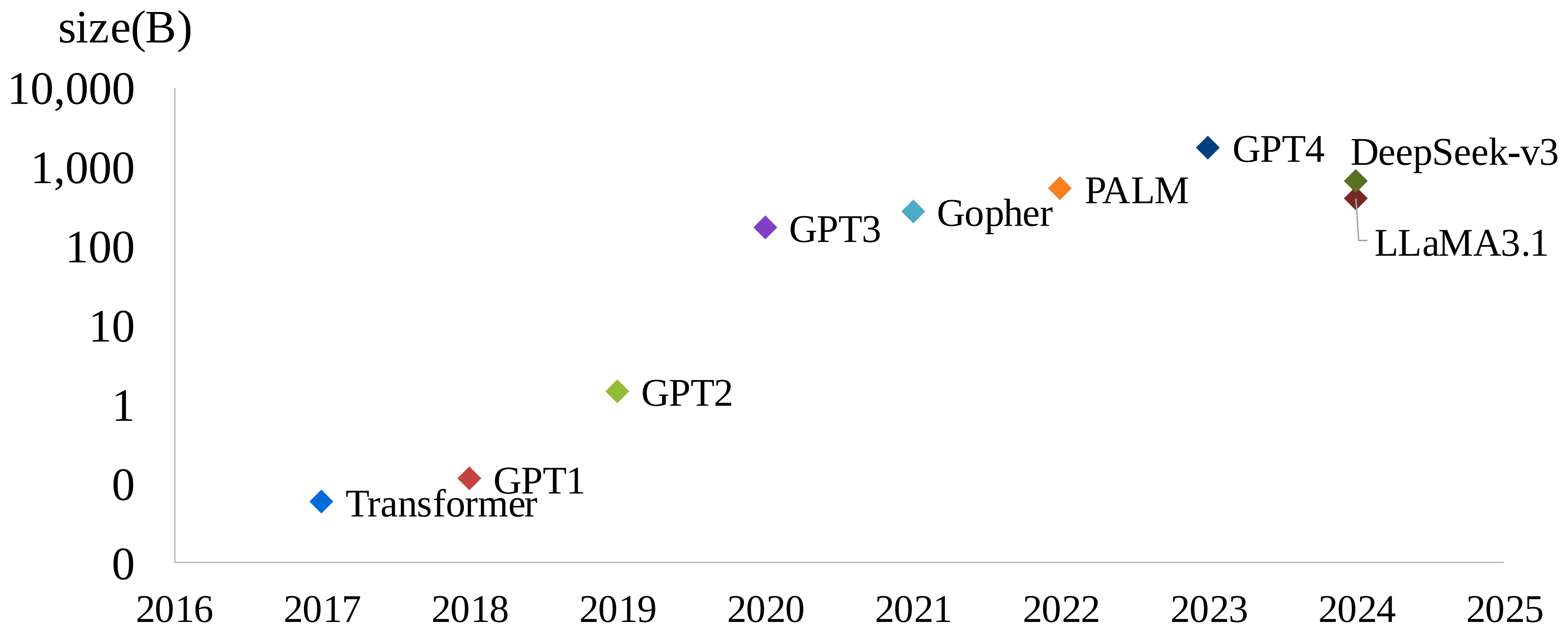
Carnegie Mellon University
Language Technologies Institute

Today’s Topic

- Model Parallel
- Pipeline Parallelism
- Tensor Parallelism

Model Parallelism

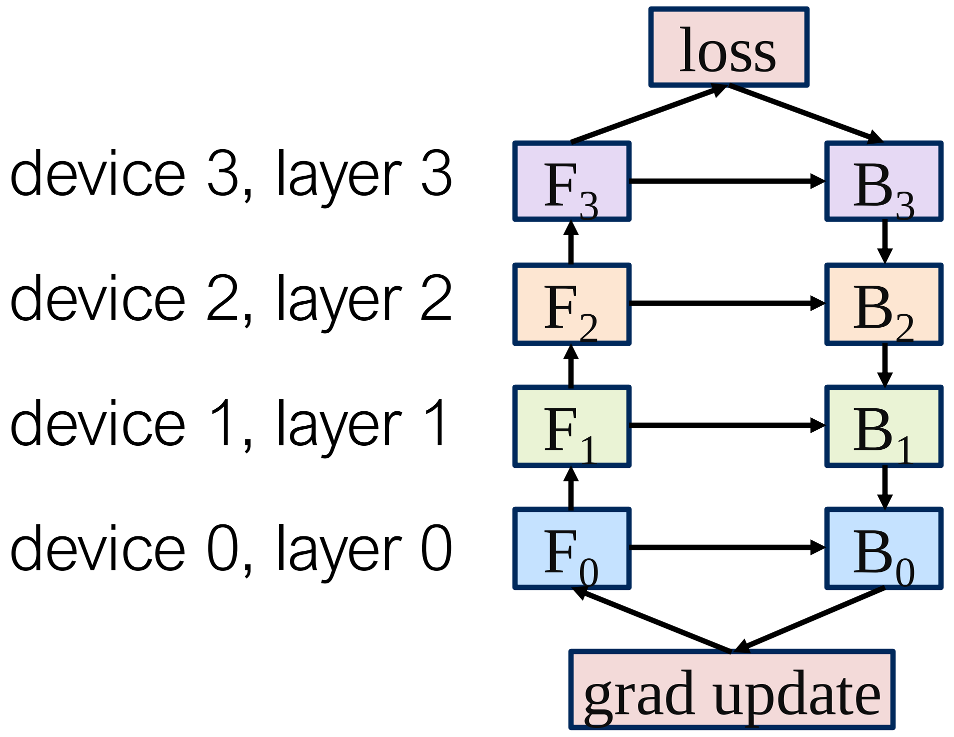
Motivation: The size of models increases exponentially fast and large. It is no longer possible to fit these large models into the memory of a single GPU.



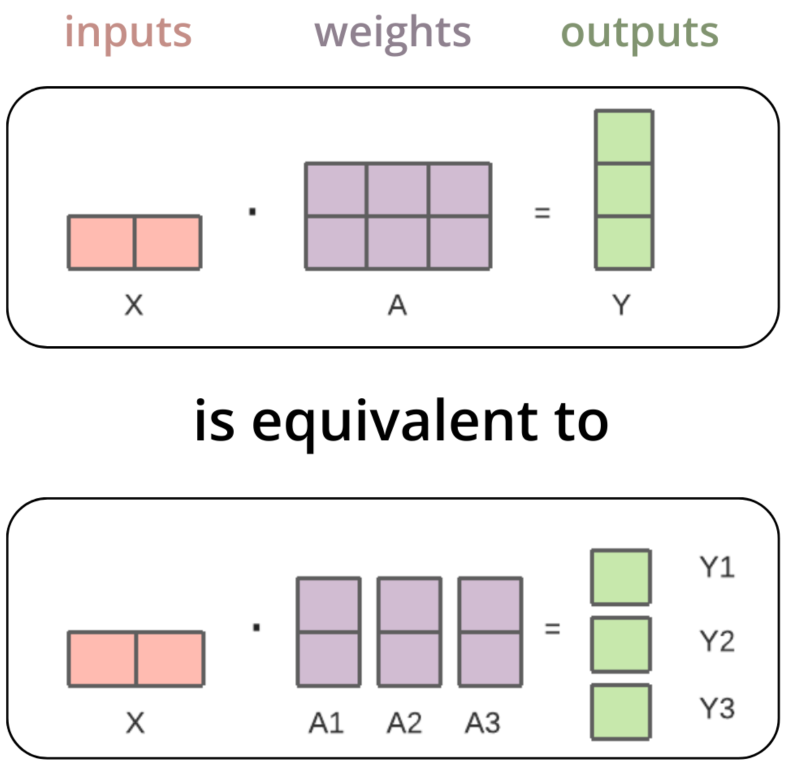
Model Parallel

Model Parallel: memory usage and computation of a model is distributed across multiple workers.

Distributed layer-wise computation

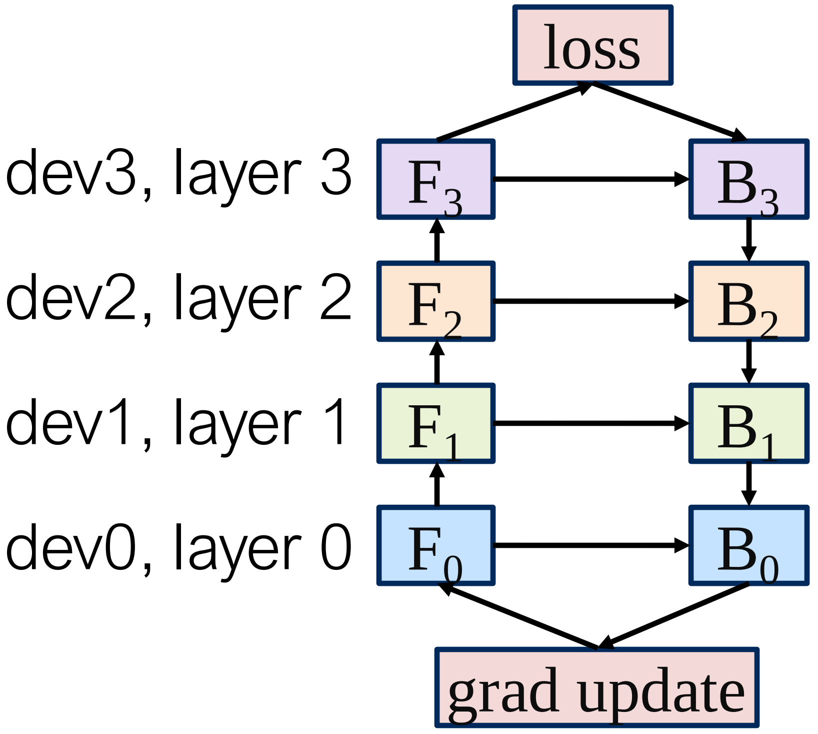


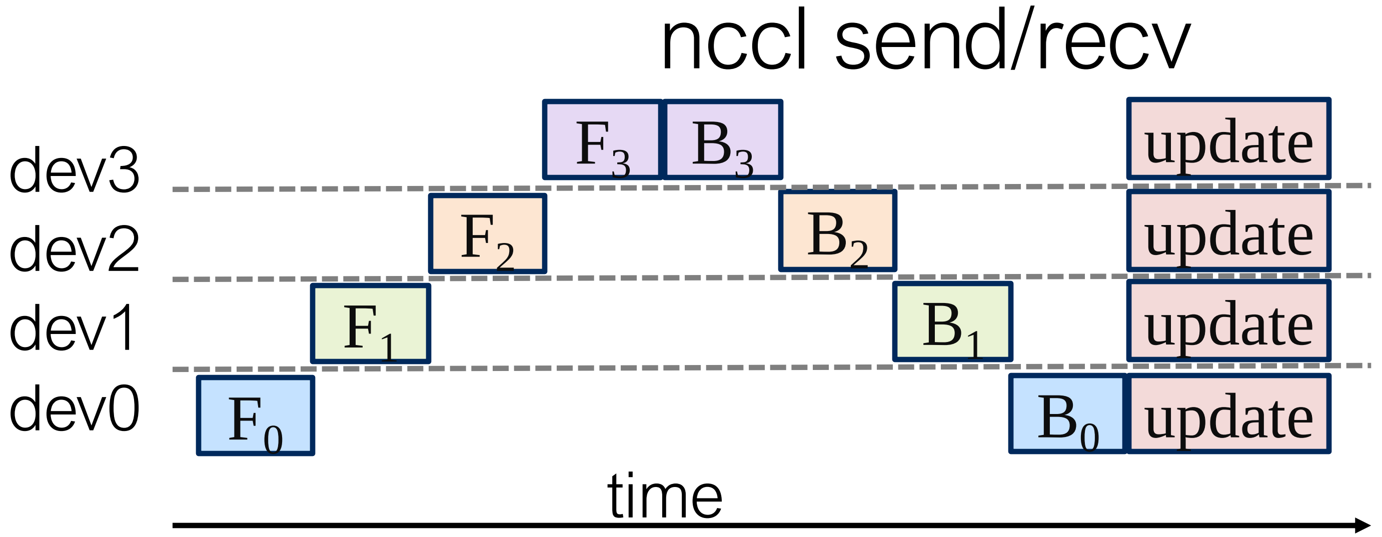
Distributed tensor computation



Pipeline Parallelism

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



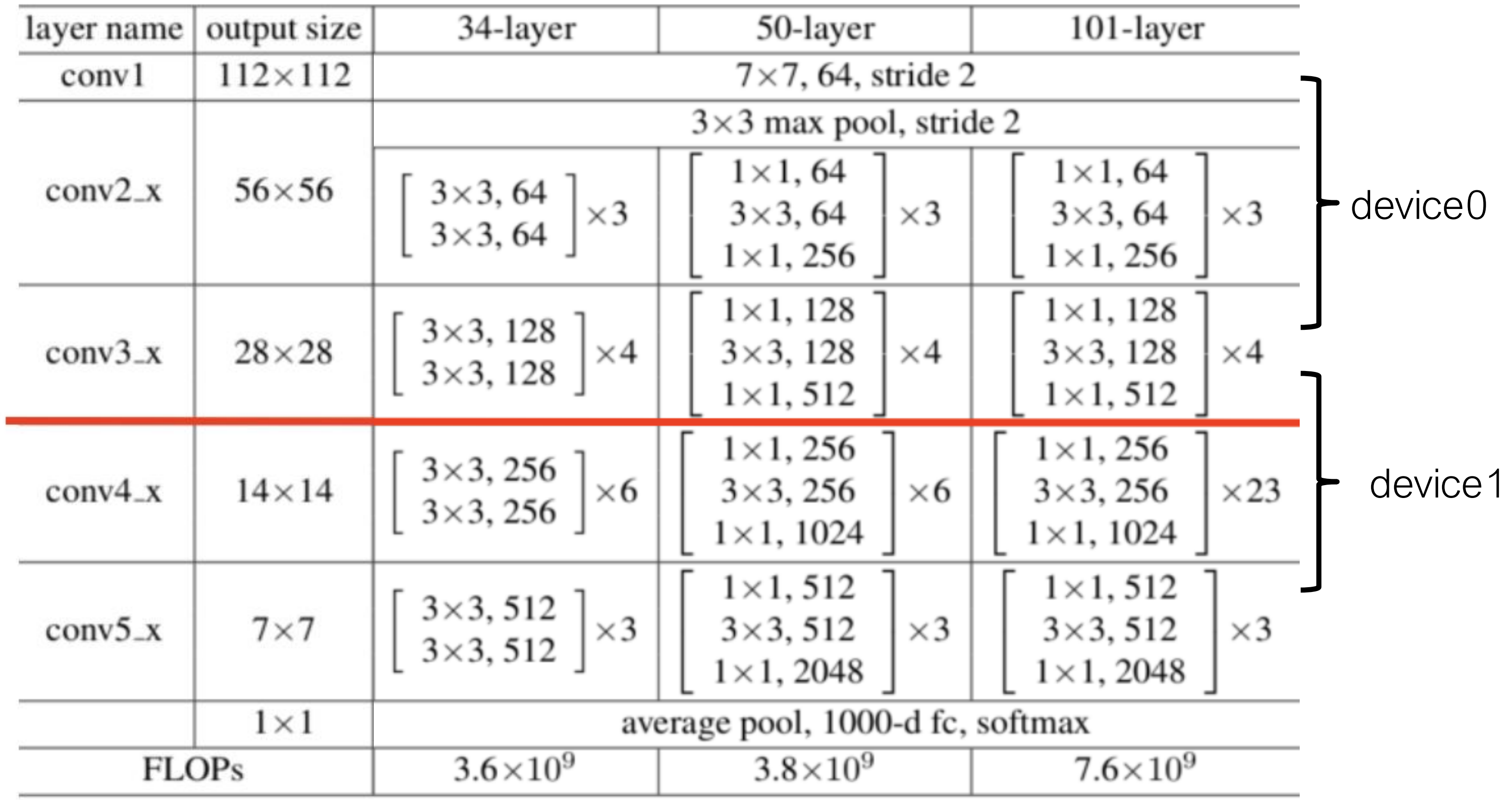


Any disadvantage?

all but one GPU is idle at any given moment!

Pipeline Parallelism

Naïve Model Parallel: The model is distributed across multiple GPUs over layers within one single node.


device0
device1

layer name	output size	34-layer	50-layer	101-layer
conv1	112×112	7×7, 64, stride 2		
conv2_x	56×56	3×3 max pool, stride 2		
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax		
FLOPs		3.6×10^9	3.8×10^9	7.6×10^9

```python
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

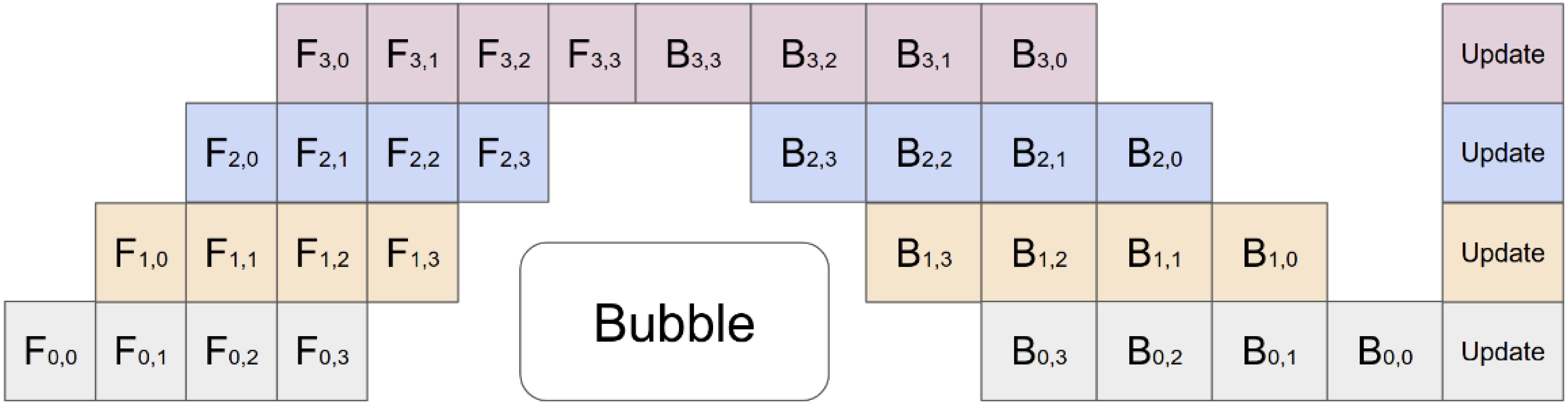
Pipeline Parallel

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

[1] Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." Advances in neural information processing systems 32 (2019).

Pipeline Parallelism

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

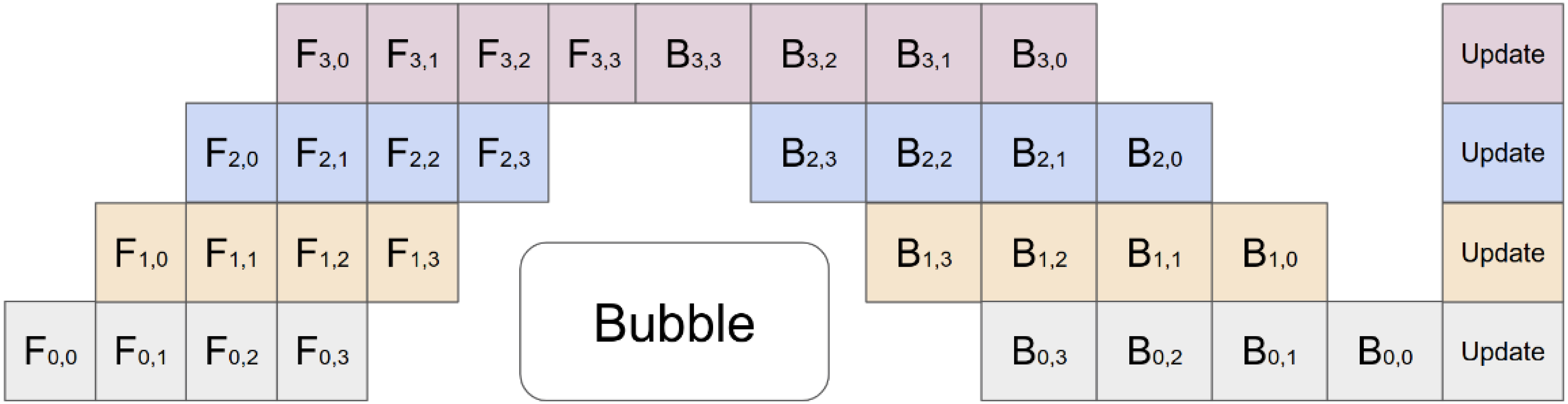

$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ Update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ Update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ Update
Bubble

(i) the number of model partitions *K*
(ii) the number of micro-batches *M*
(iii) the sequence and definitions of *L* layers that define the model

[1] Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." Advances in neural information processing systems 32 (2019).

Pipeline Parallelism

GPipe: Divides input mini-batches into smaller micro-batches. During backward, recomputes forward


$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ Update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ Update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ Update
Bubble

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: $O(N \times L)$ ➔ $O(N + \frac{L}{K} \times \frac{N}{M})$

Pipeline Parallelism

Pipeline Parallel: Split the inputs to reduce bubbles within one single node.

```python
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))
```

```python
class PipelineParallelResNet50(ModelParallelResNet50):
    def __init__(self, split_size=20, *args, **kwargs):
        super(PipelineParallelResNet50, self).__init__(*args, **kwargs)
        self.split_size = split_size

    def forward(self, x):
        splits = iter(x.split(self.split_size, dim=0))
        s_next = next(splits)
        s_prev = self.seq1(s_next).to('cuda:1')
        ret = []

        for s_next in splits:
            # A. ``s_prev`` runs on ``cuda:1``
            s_prev = self.seq2(s_prev)
            ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

            # B. ``s_next`` runs on ``cuda:0``, which can run concurrently with A
            s_prev = self.seq1(s_next).to('cuda:1')

        s_prev = self.seq2(s_prev)
        ret.append(self.fc(s_prev.view(s_prev.size(0), -1)))

        return torch.cat(ret)
```

Pytorch launches the GPUs asynchronously so that we can have `self.seq2(s_prev)` and `self.seq1(s_next)` run concurrently with different micro-batches of data.

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

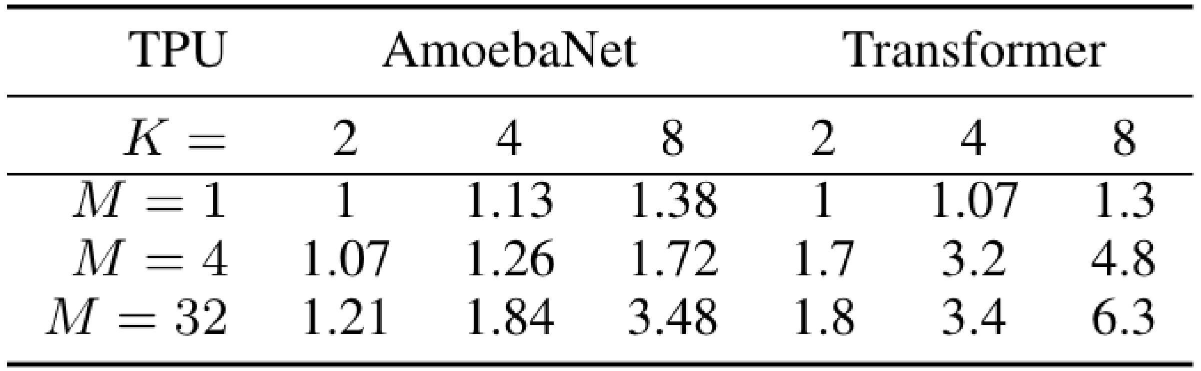
```
from torch.distributed.pipelining import PipelineStage

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)
```

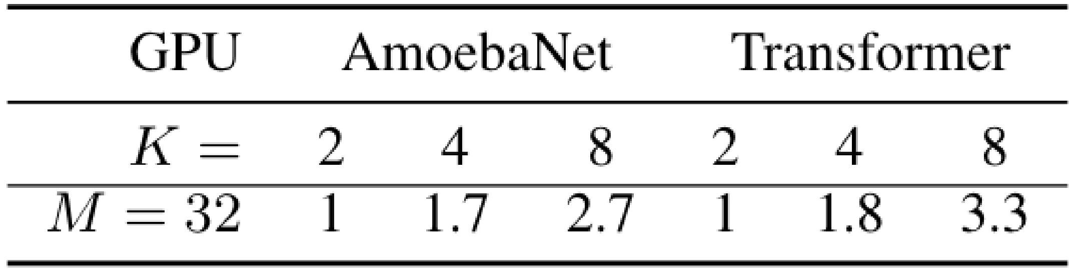
```
from torch.distributed.pipelining import ScheduleGPipe
# Create a schedule
schedule = ScheduleGPipe(stage, n_microbatches)
# Input data (whole batch)
x = torch.randn(batch_size, in_dim, device=device)
# Run the pipeline with input `x` # `x` will be divided into microbatches automatically
if rank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

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



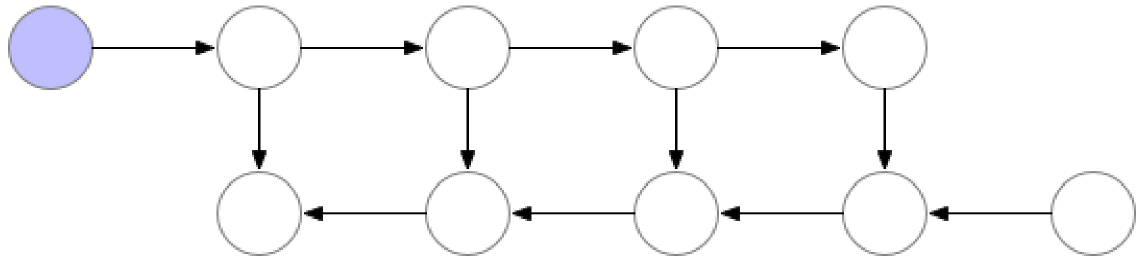
GPU	AmoebaNet			Transformer		
$K =$	2	4	8	2	4	8
$M = 32$	1	1.7	2.7	1	1.8	3.3

Gradient Checkpointing

Re-materialization

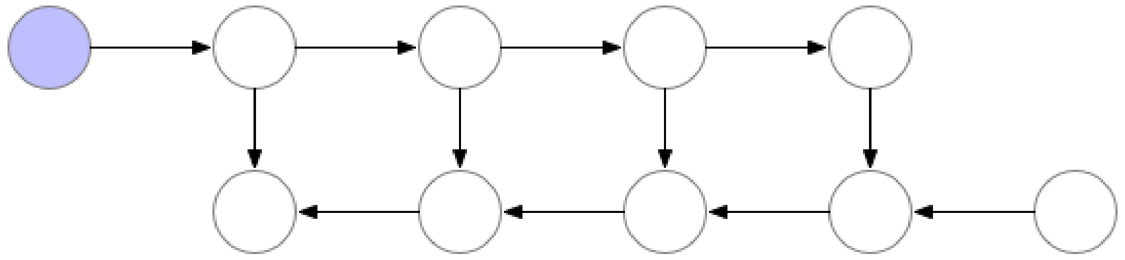
- Forward pass: each accelerator only stores output activations
- Backward pass: the k–th accelerator recomputes the composite forward function F_k

Vanilla backprop



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

Memory poor backprop



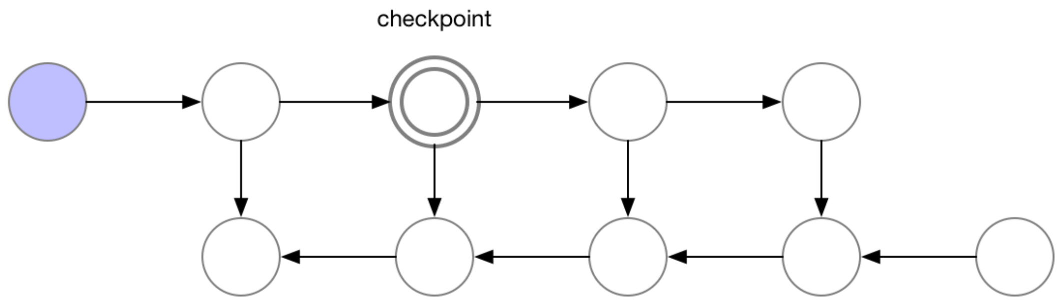
- Memory for activations: *O(1)*
- Node computation: $O(n^2)$

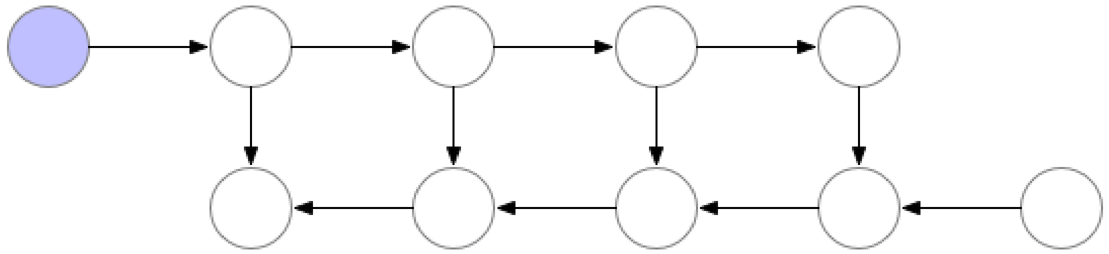
[1] Chen, Tianqi, et al. "Training deep nets with sublinear memory cost." arXiv preprint arXiv:1604.06174 (2016).
[2] https://github.com/cybertronai/gradient-checkpointing

Gradient Checkpointing

Gradient checkpoint

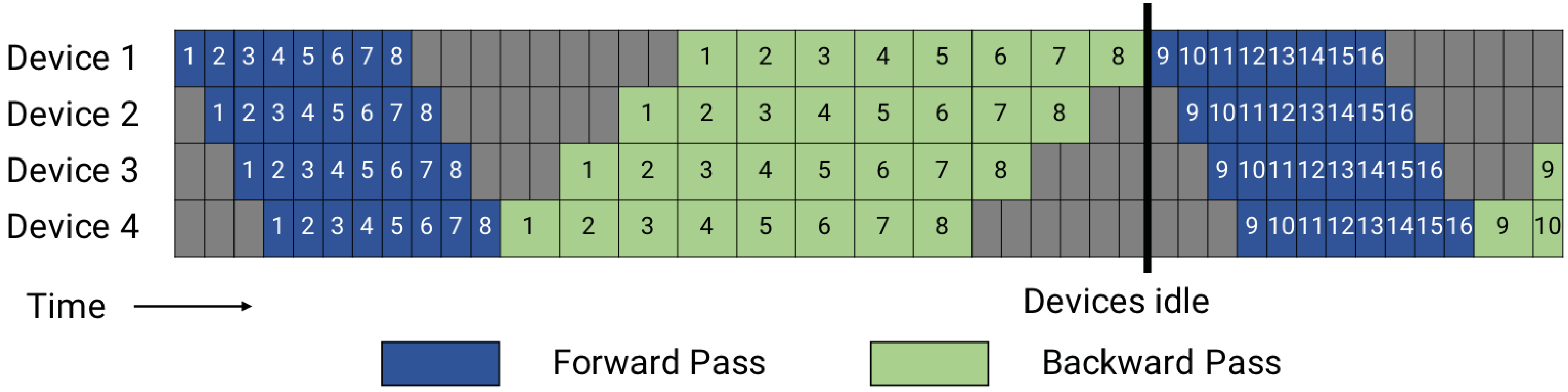
- Cash the activations of every sqrt(n) layers
- Memory for activations: *O(n)*
- *Node computation: O(sqrt(n) * sqrt(n)) = O(n)*





[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

Standard Pipeline Model Parallel

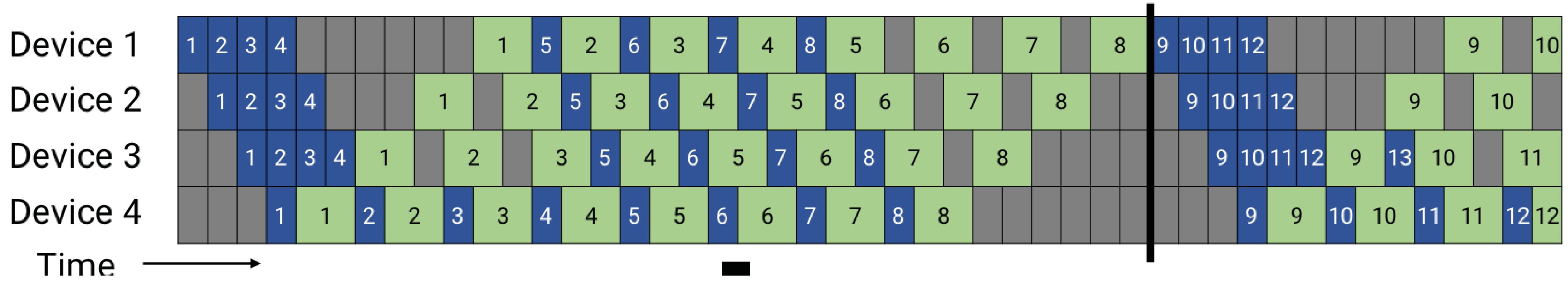


number of micro-batches in a batch: *m*
number of pipeline stages (number of devices used for pp): *p*
ideal time per iteration: t_{id} , forward pass for single micro-batch: t_f , backward pass: t_b

bubble time fraction (pipeline bubble size): $\frac{t_{pb}}{t_{id}} = \frac{(p-1)\cdot(t_f+t_b)}{m\cdot(t_f+t_b)} = \frac{p-1}{m}$

PipeDream-Flush

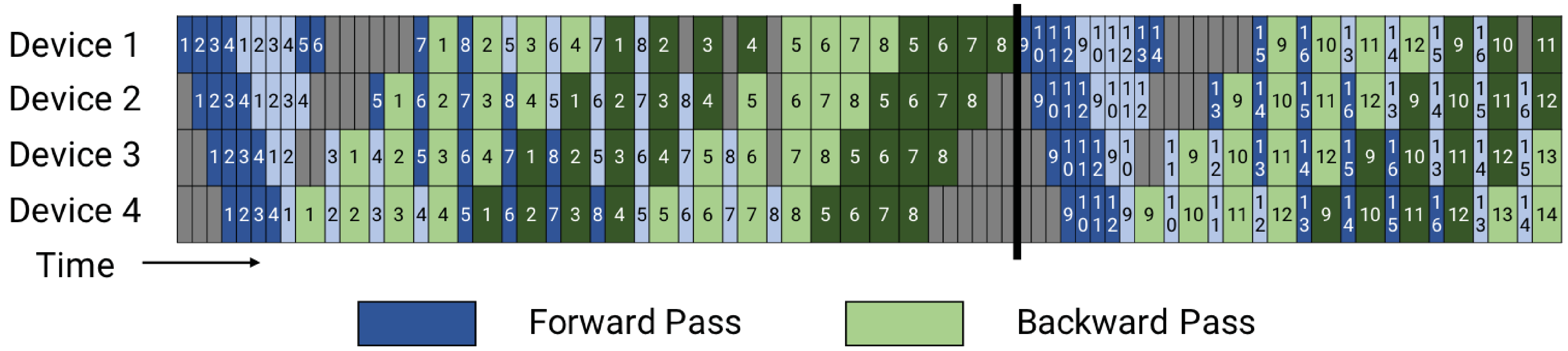
- PipeDream-Flush – start backward as soon as possible



Memory-Efficient Pipeline-Parallel DNN Training. Narayanan et al ICML 2021.

Interleaved Pipeline Parallel

- Schedule with Interleaved Stages



number of micro-batches in a batch: *m*
number of pipeline stages (number of devices used for pp): *p*

model chunks: *v* , pipeline bubble time: $t_{pb}^{\text{int.}} = \frac{(p-1)\cdot(t_f+t_b)}{v}$

bubble time fraction (pipeline bubble size): $\frac{t_{pb}^{\text{int.}}}{t_{id}} = \frac{1}{v} \cdot \frac{p-1}{m}$

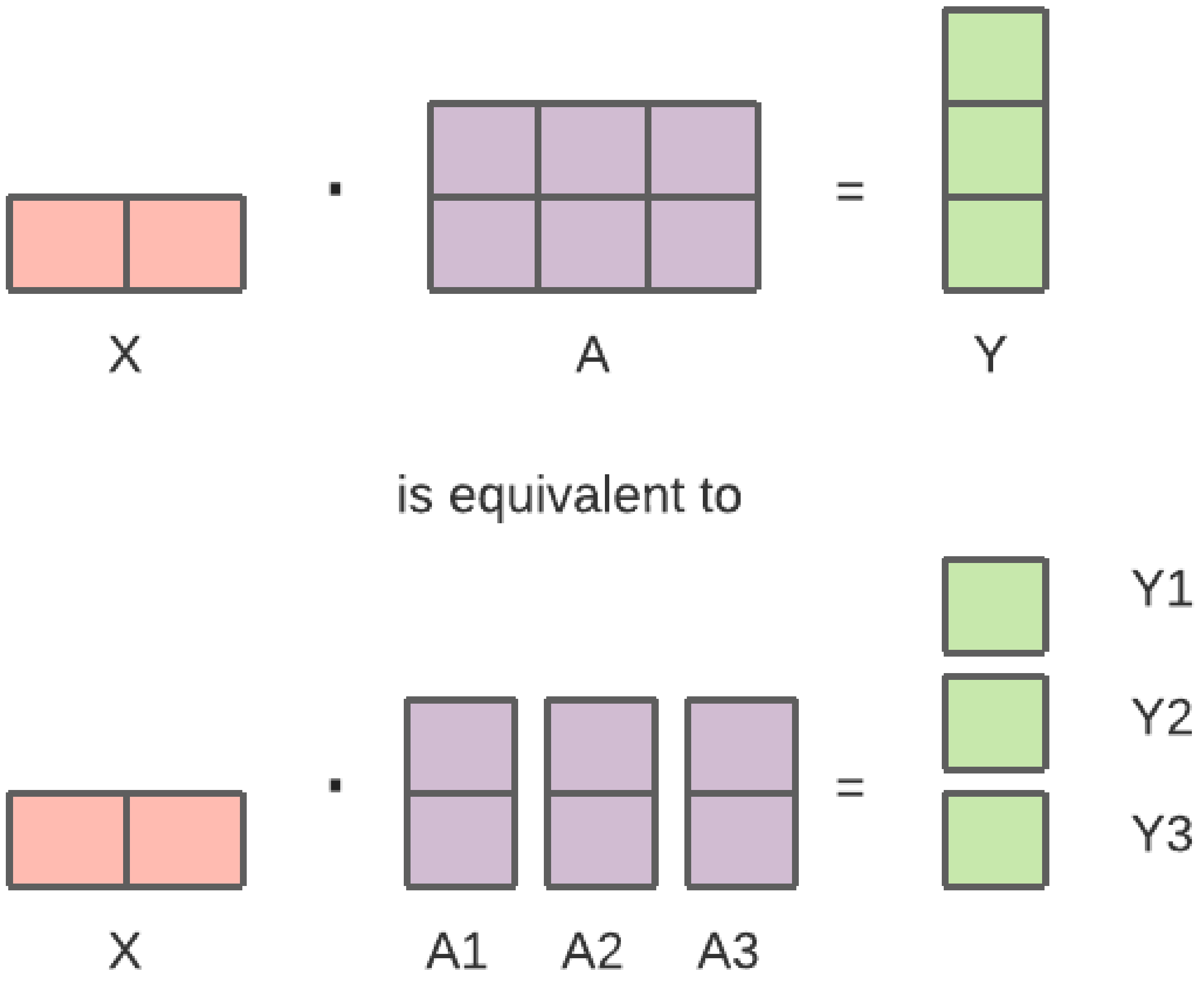
Megatron-LM. Narayanan et al 2021.

Quiz 8

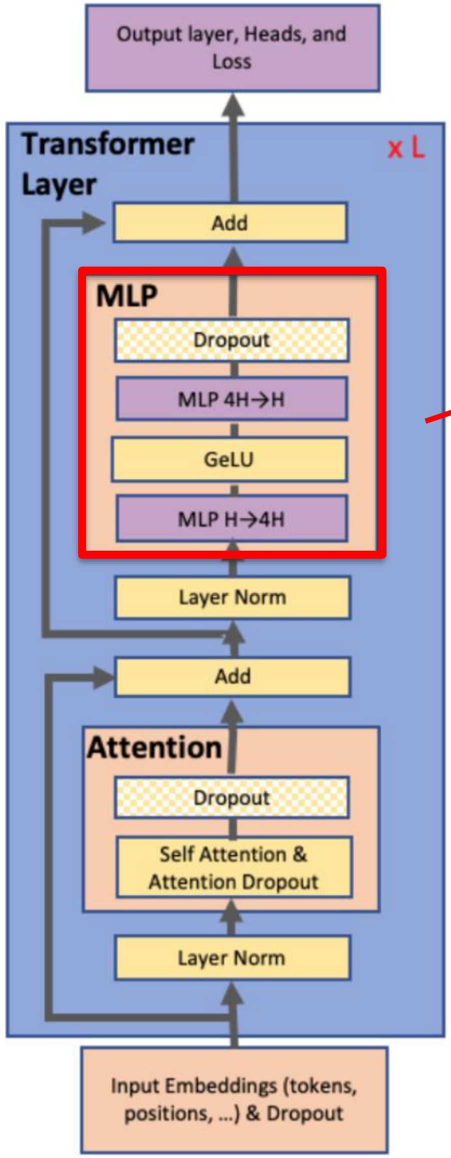
- on canvas

Tensor Parallelism

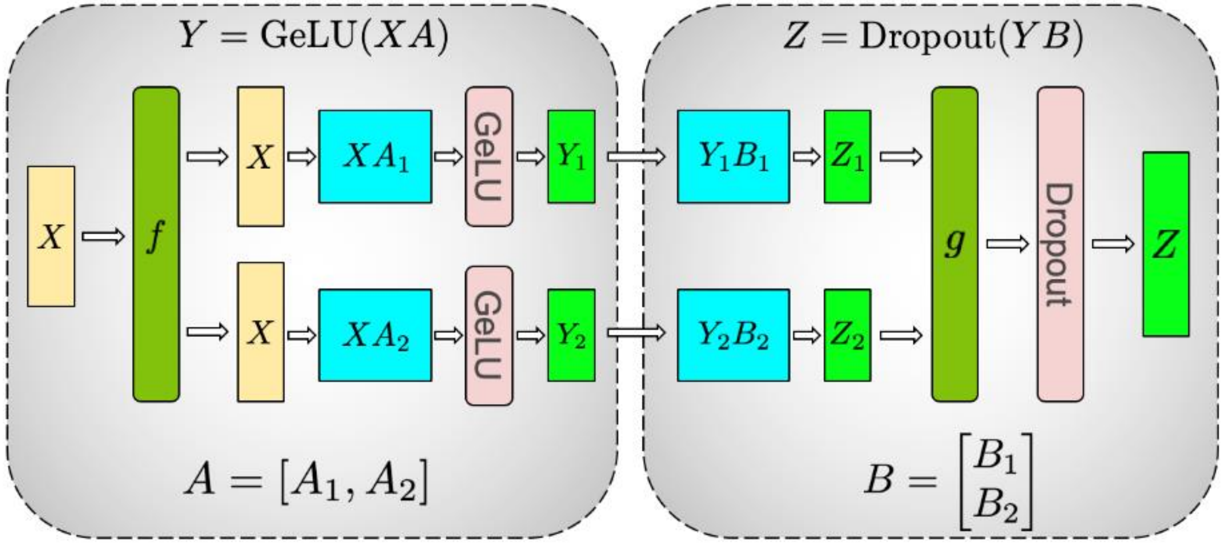
Tensor Parallelism



Tensor Parallelism for FFN



Output layer, Heads, and Loss
Transformer Layer
x L
Add
MLP
Dropout
MLP 4H→H
GeLU
MLP H→4H
Layer Norm
Add
Attention
Dropout
Self Attention & Attention Dropout
Layer Norm
Input Embeddings (tokens, positions, ...) & Dropout



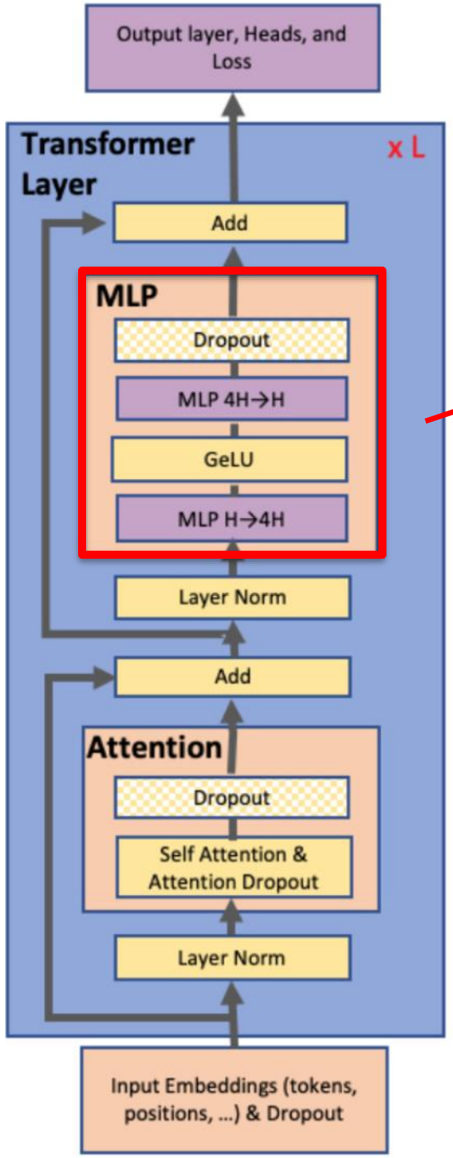
$$Y = GeLU(XA)$$

$$X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}, A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \qquad Y = GeLU(X_1A_1 + X_2A_2)$$

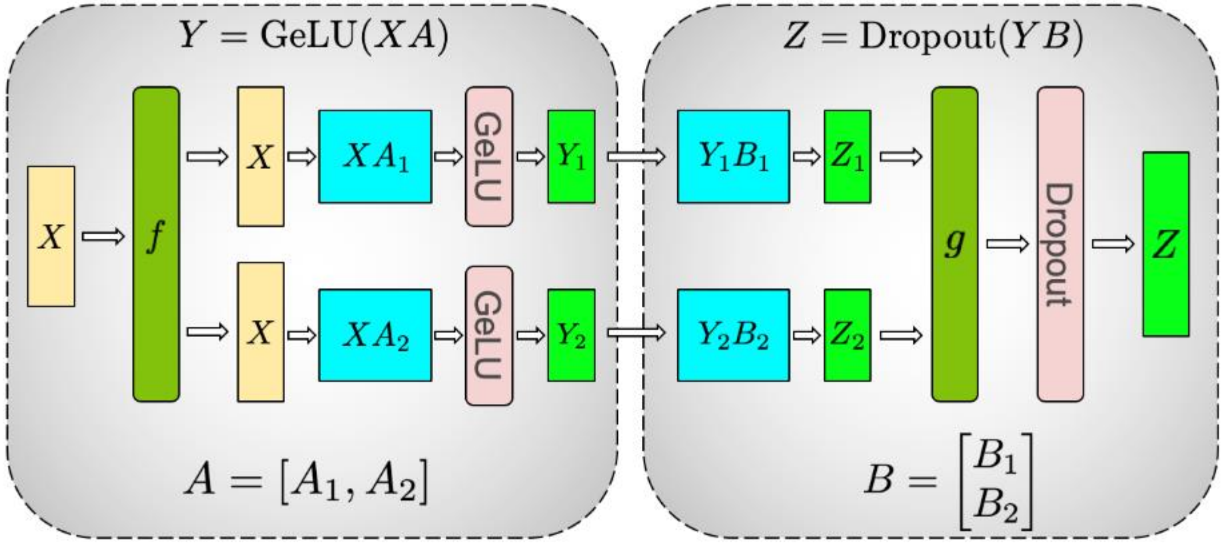
$$GeLU(X_1A_1 + X_2A_2) \neq GeLU(X_1A_1) + GeLU(X_2A_2)$$

All-reduce is needed！

Tensor Parallelism for FFN



Output layer, Heads, and Loss
Transformer Layer
x L
Add
MLP
Dropout
MLP 4H→H
GeLU
MLP H→4H
Layer Norm
Add
Attention
Dropout
Self Attention & Attention Dropout
Layer Norm
Input Embeddings (tokens, positions, ...) & Dropout



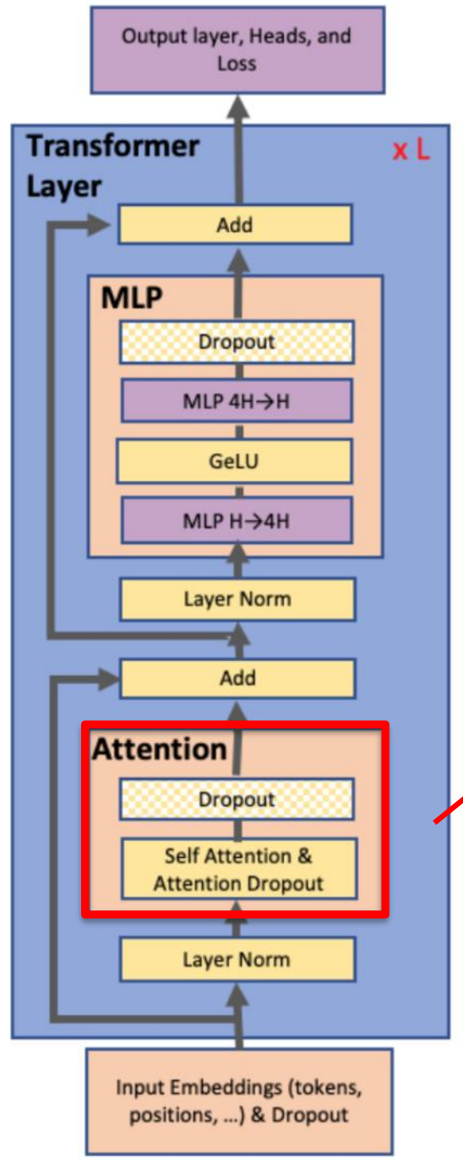
$$Y = GeLU(XA)$$

$$A = [A_1, A_2]$$

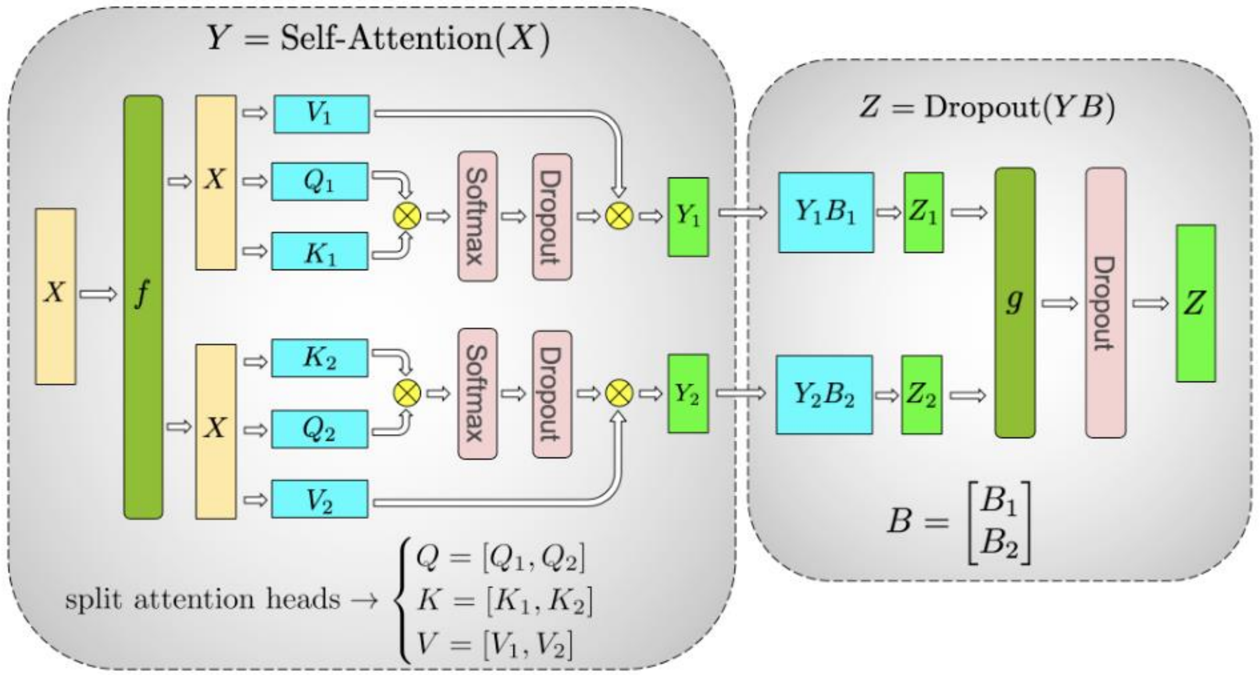
$$[Y_1 \quad Y_2] = [GeLU(XA_1), GeLU(XA_2)]$$

All-reduce is not needed !

Tensor Parallelism for Self-Attention

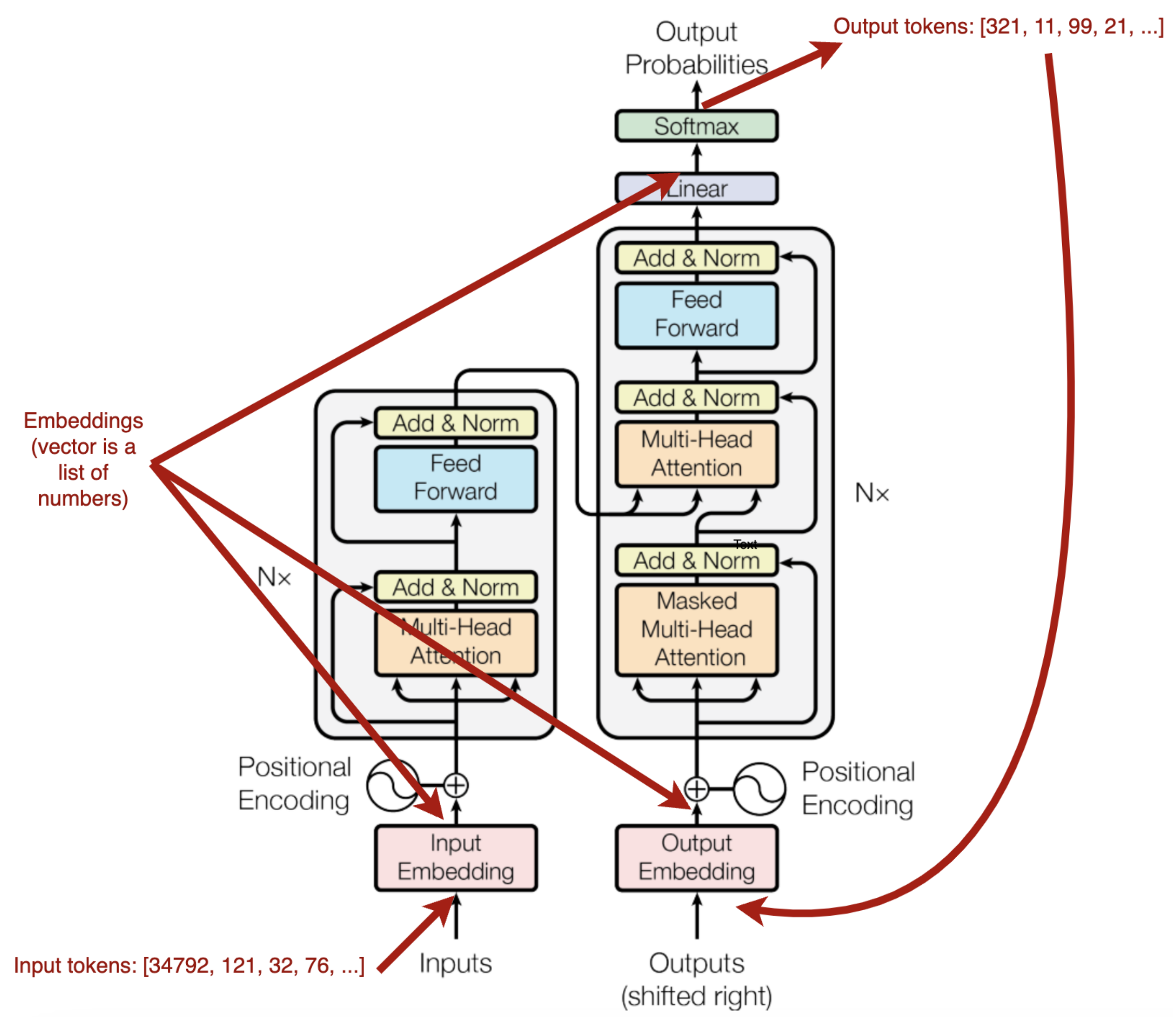


Output layer, Heads, and Loss
Transformer Layer
x L
Add
MLP
Dropout
MLP 4H→H
GeLU
MLP H→4H
Layer Norm
Add
Attention
Dropout
Self Attention & Attention Dropout
Layer Norm
Input Embeddings (tokens, positions, ...) & Dropout



- Split weights over columns (heads)
- All-reduce is not needed！

Tensor Parallelism - Embeddings



- Input embedding
  - Split over columns

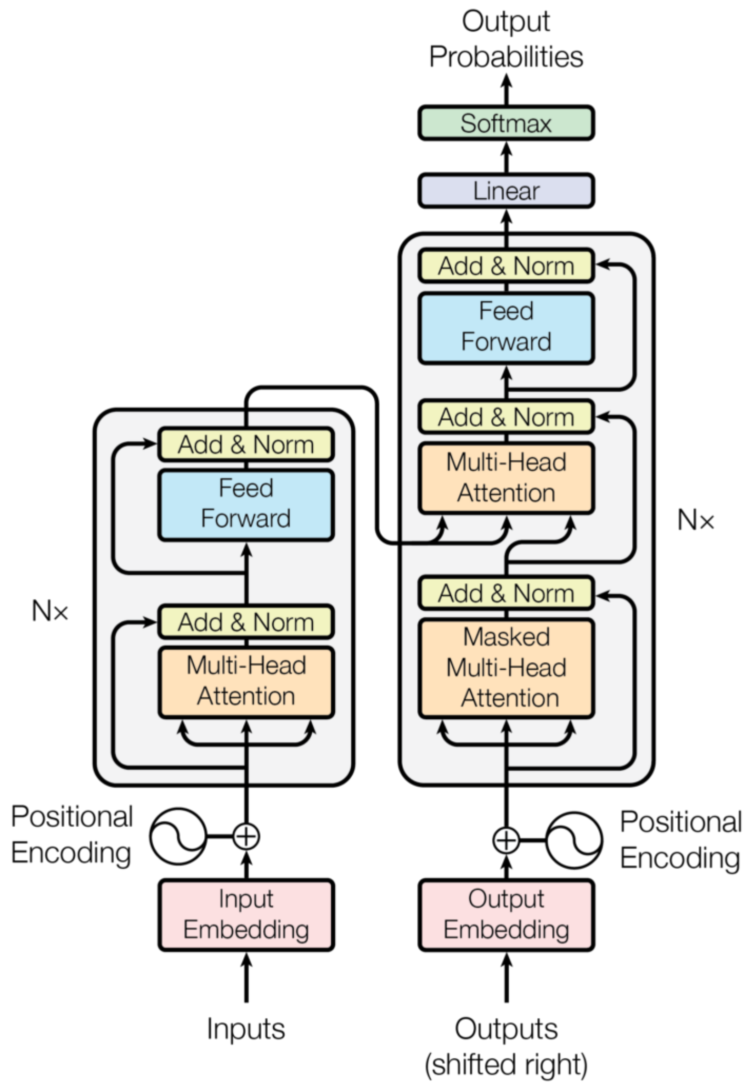
    $$E = [E_1, E_2] \text{ (column-wise)}$$

  - all-reduce is required
- Output embedding
  - Split over columns

    $$\text{GEMM } [Y_1, Y_2] = [XE_1, XE_2]$$

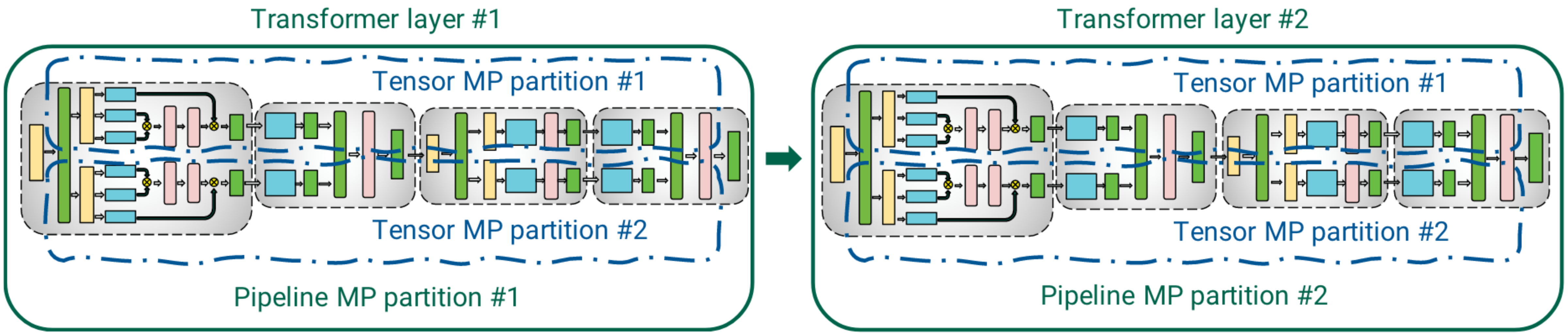
  - Fuse outputs with cross-entropy loss (huge reduction in communication)
  - all-gather is needed

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



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

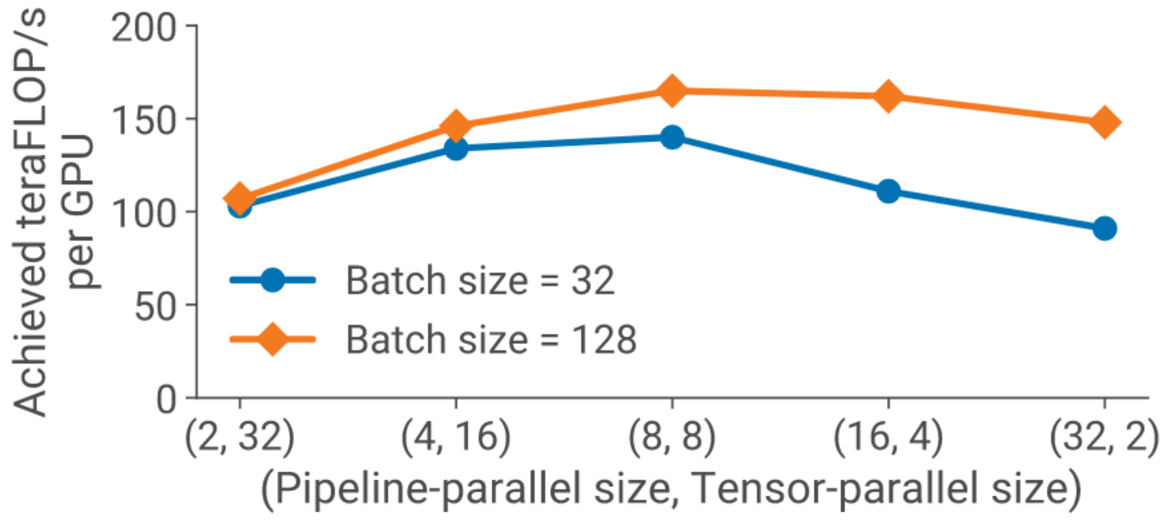


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

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.

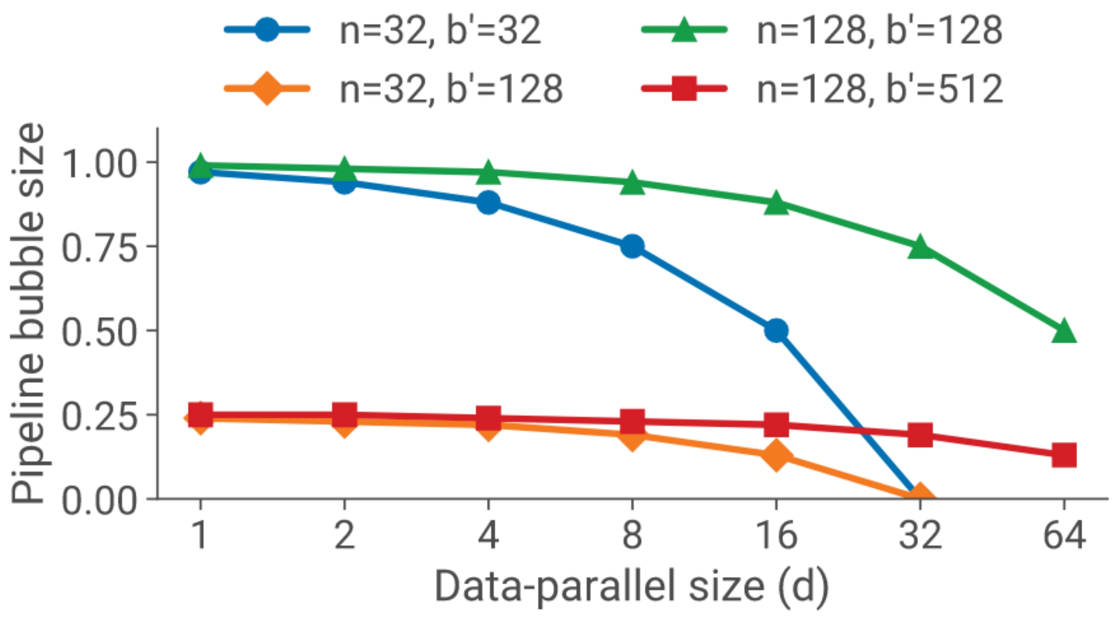


Figure 6: Fraction of time spent idling due to pipeline flush (pipeline bubble size) versus data-parallel size (d), for different numbers of GPUs (n) and ratio of batch size to microbatch size ($b' = B/b$).

Summary

- Pipeline Parallelism
  - split by layers (horizonal split)
  - eliminate the bubbles (idle)
  - interleaving forward/backward
- Tensor Parallelism
  - split the matrix computation