CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

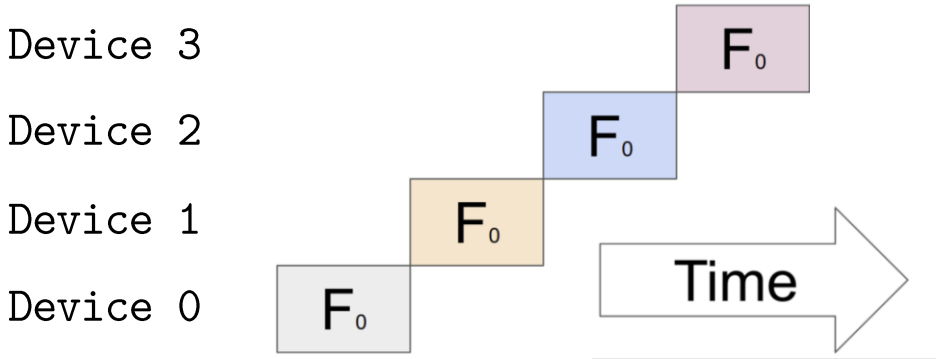
Lecture 12: Efficiency III

WORKING WITH LARGE MODELS

- NLP models benefit from increasing scale.
- GPT-3, for example, has ~175 billion parameters.
- Each parameter and activation is stored as a 16-bit floating point number.
- Thus, you need 350 GB of VRAM just to load the model parameters.
  - You need more memory to store (batched) activations and KV cache for inference.
  - You need a lot more to do any training or fine-tuning.
- The H200 GPU has 192 GB of VRAM.
  - Consumer GPUs have 16 or 24 GB.
- So in order to work with models that don't fit in one GPU's VRAM, we need to find ways to distribute inference/training across multiple GPUs.

DISTRIBUTED INFERENCE/TRAINING

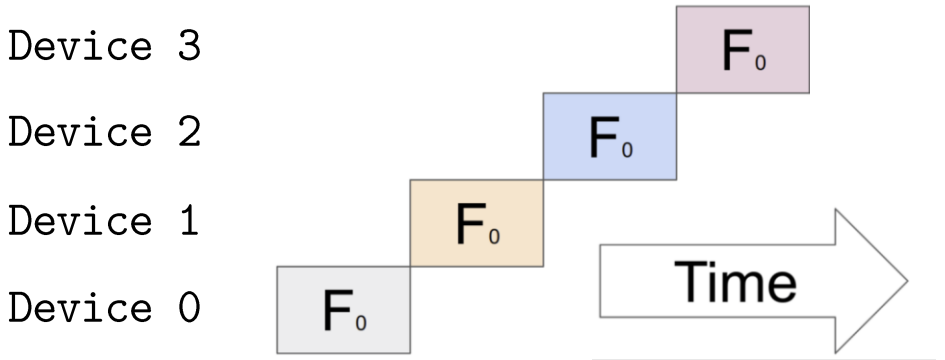
- If a model is too big to fit in a single GPU's VRAM, one idea is to split the model into smaller portions.
- Assign each portion of the model to a GPU.
  - This general idea is known as model parallelism.
- For example, if a model consists of many layers, we can divide the layers among the available GPUs.



[Huang et al., 2019]

PIPELINE PARALLELISM

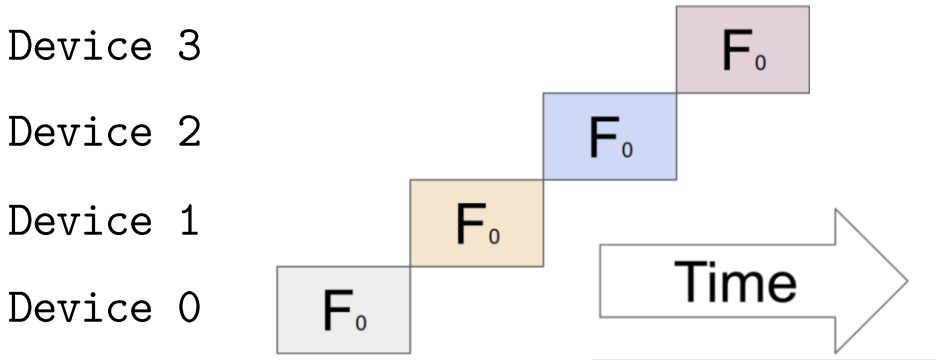
- For example, if our model has 100 layers, we assign 25 to each of 4 GPUs.
- In the forward pass, `device 0` first computes the activations after the first 25 layers.



[Huang et al., 2019]

PIPELINE PARALLELISM

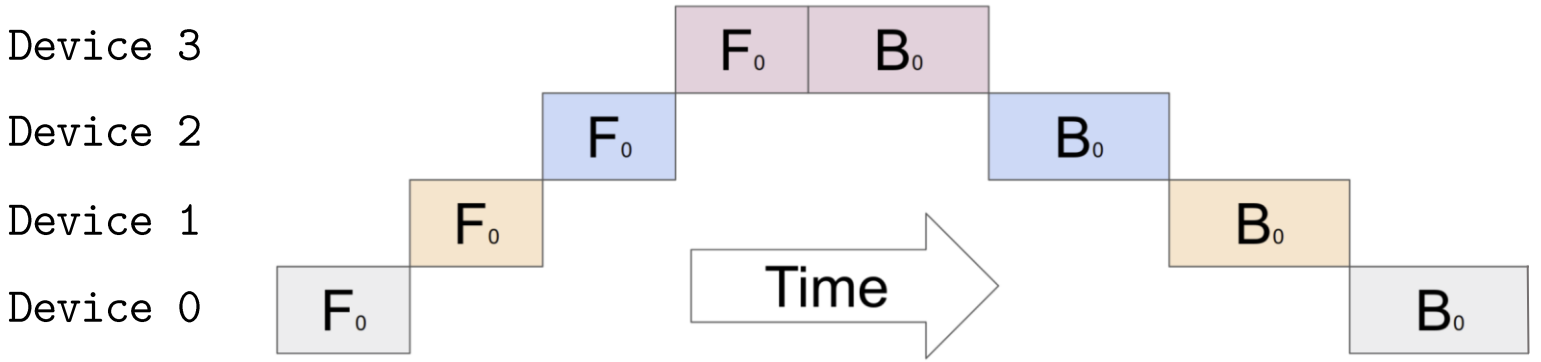
- The activations are then copied from `device 0` to `device 1`,
  - And `device 1` computes the activations after the first 50 layers.
- We repeat until `device 3` compute the activations after all 100 layers.
- Note that we can easily combine this approach with batching to improve throughput. (memory permitting)



[Huang et al., 2019]

PIPELINE PARALLELISM

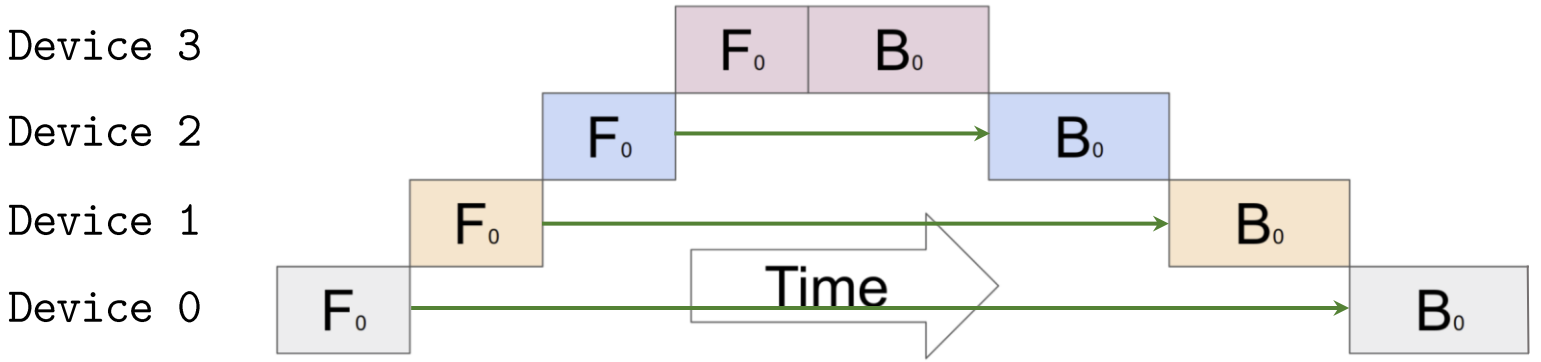
- The same idea can be used for training, too.
- After `device 3` finishes the forward pass, it performs a backward pass on layers 75-100.
- The gradients are copied from `device 3` to `device 2`, which then performs a backward pass on layers 50-75.
  - And so on.



[Huang et al., 2019]

PIPELINE PARALLELISM

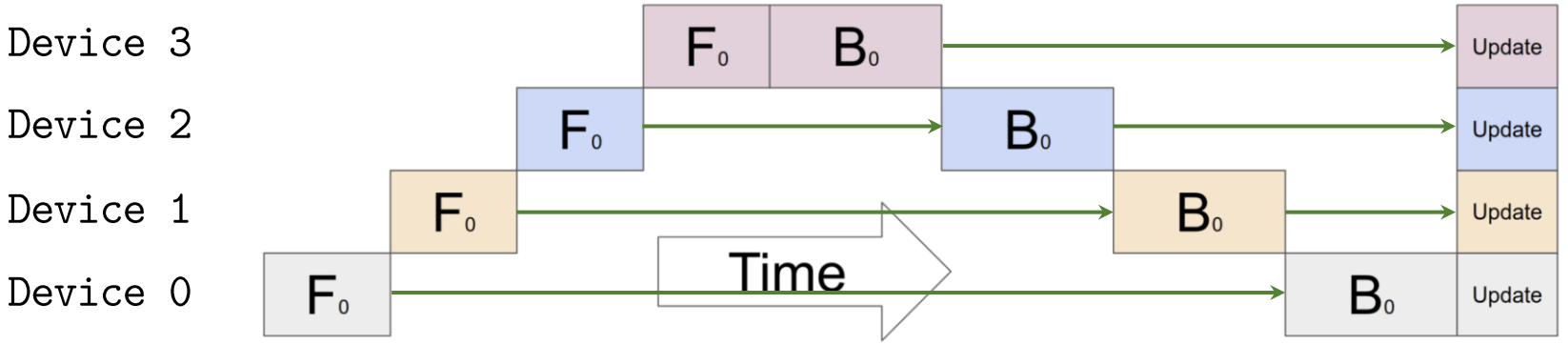
- In order to perform the backward pass, each device needs to keep the activations from the forward pass in memory.
- In each backward pass, each device computes the gradients for only the weights of the layers that are assigned to that device.



[Huang et al., 2019]

PIPELINE PARALLELISM

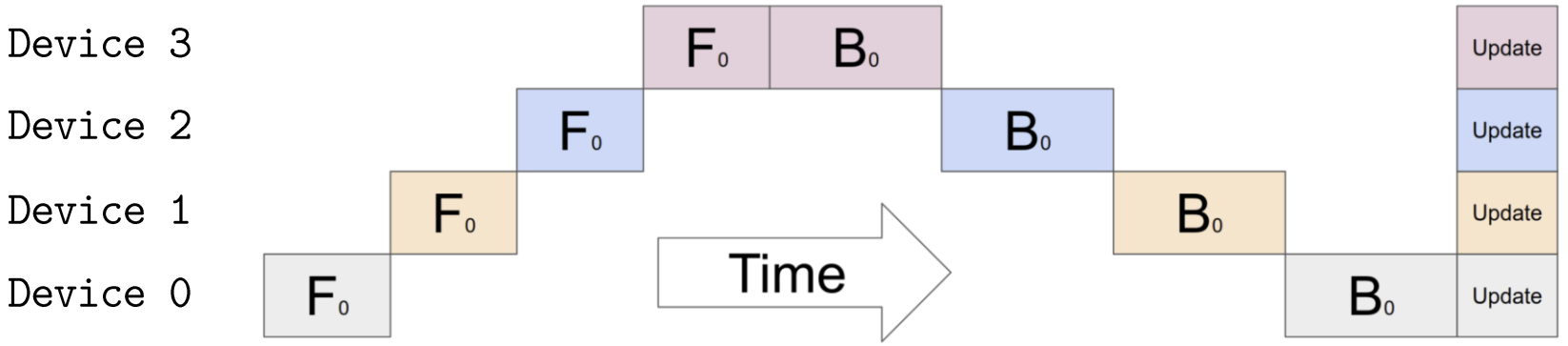
- In order to perform the backward pass, each device needs to keep the activations from the forward pass in memory.
- In each backward pass, each device computes the gradients for only the weights of the layers that are assigned to that device.
- Once we have computed the gradients on all devices, the gradient update step is performed simultaneously.



[Huang et al., 2019]

PIPELINE PARALLELISM

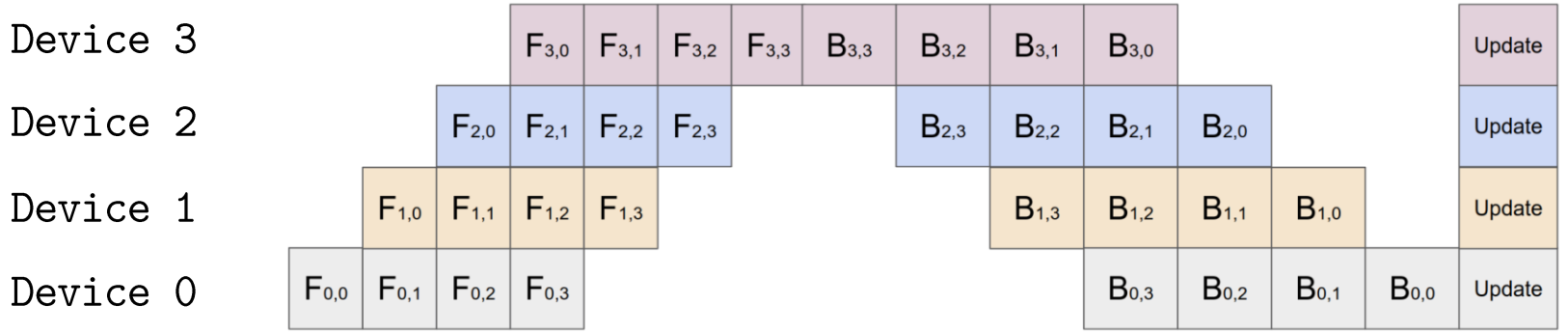
- One disadvantage of this approach is idle time:
  - Most devices are spending most of their time doing nothing.
  - Hardware utilization is low.
- An idea to address this is to process multiple inputs simultaneously.
  - If using batches, we would process multiple input batches simultaneously.



[Huang et al., 2019]

PIPELINE PARALLELISM

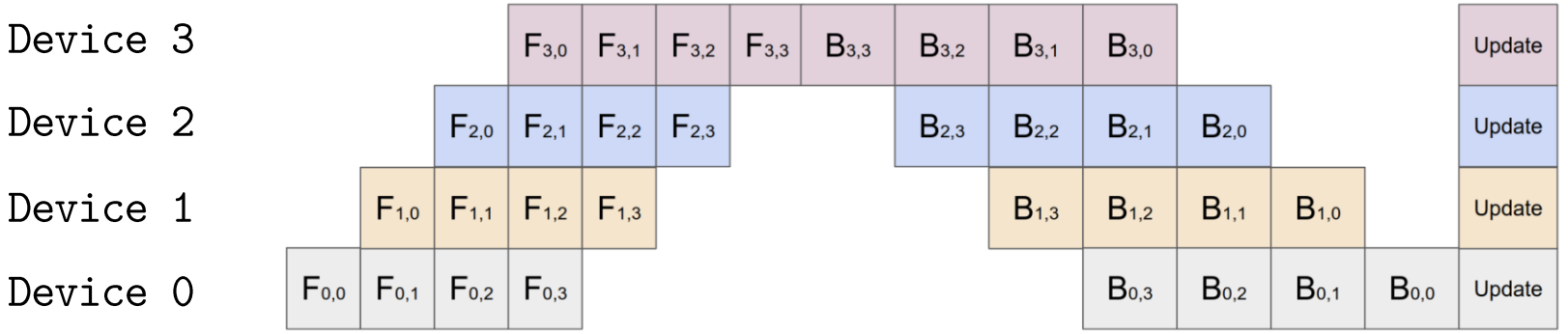
- `Device 0` processes `batch 0` first, then passes the activations to `device 1`, and immediately begins processing `batch 1`, etc.
- This is called pipeline parallelism or inter-layer parallelism.
  - Since we can apply it to any model that contains sequential computation (i.e., a pipeline), such as the layers of a transformer.



[Huang et al., 2019]

PIPELINE PARALLELISM

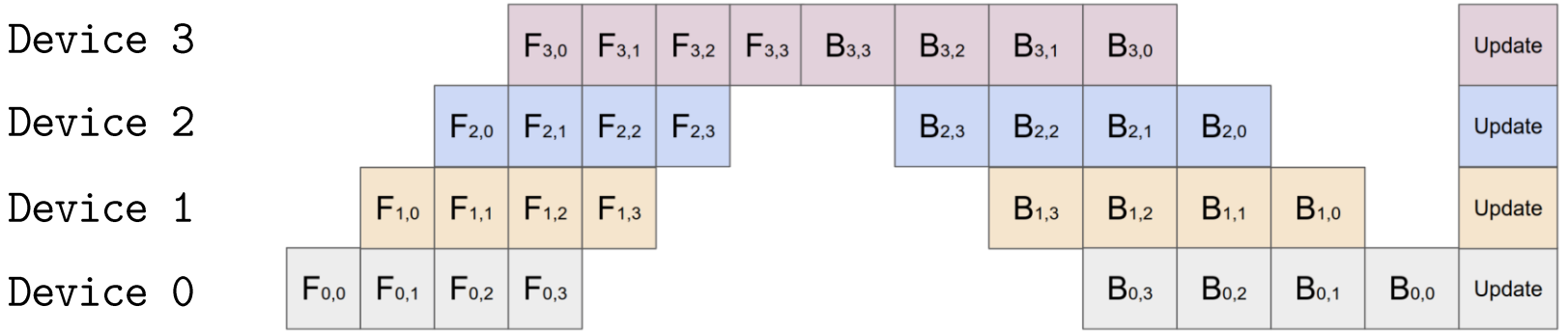
- During training, in the example below, the training batch is split into 4 “minibatches.”
- Gradients are accumulated across the backward passes, so that we correctly compute the gradient over the full training batch.
- The gradient update is performed with the accumulated gradients.



[Huang et al., 2019]

PIPELINE PARALLELISM

- Even with pipelining, there is still considerable idle time for each device,
  - Especially during training when `device 0` has to wait for all other devices to finish their forward and backward passes.
- What if we divide each individual layer across GPUs?



[Huang et al., 2019]

TENSOR PARALLELISM

- Let's take a closer look at the feedforward layer in the transformer:
  - $X_{out} = f(X W_1^T + b_1) W_2^T + b_2$
- Recall that X has dimension $n \times d$, where n is the sequence length and d is the model dimension.
- What if we divided the weight matrix W_1 and X along their columns?
- For simplicity, let's assume we just have two machines/GPUs.

$$X = [X^{(1)} \quad X^{(2)}] \text{ and } W_1 = \begin{bmatrix} W_1^{(1)} & W_1^{(2)} \end{bmatrix}$$

- Then $f(X W_1^T + b_1) = f(X^{(1)} W_1^{(1)T} + X^{(2)} W_1^{(2)T} + b_1)$.

TENSOR PARALLELISM

$$f(X W_1^T + b_1) = f(X^{(1)} W_1^{(1)T} + X^{(2)} W_1^{(2)T} + b_1)$$

- So we can imagine giving the left $d/2$ columns of W_1 and X to the first machine, and give the remaining $d/2$ columns to the second machine.
- The first machine computes $X^{(1)} W_1^{(1)T}$ and the second computes $X^{(2)} W_1^{(2)T}$.
  - But the problem is the nonlinear activation function $f(\ldots)$.
  - We can't simply apply f to each portion of the product, since it's not linear.

$$f(X^{(1)} W_1^{(1)T} + X^{(2)} W_1^{(2)T} + b_1) \neq f(X^{(1)} W_1^{(1)T}) + f(X^{(2)} W_1^{(2)T} + b_1)$$

  - Instead, we would need to synchronize the machines:
    - Have them share information about the first matrix product before computing the activation function.

TENSOR PARALLELISM

- So let's try the other approach: Divide W_1 along its rows.
  - The first machine has the first $d_{ff}/2$ rows of W_1.
  - The second machine has the last $d_{ff}/2$ rows of W_1.

$$W_1 = \begin{bmatrix} W_1^{(1)} \\ W_1^{(2)} \end{bmatrix}$$

- Then $f(X W_1^T + b_1) = f\left(\left[X W_1^{(1)T} + b_1^{(1)}, \quad X W_1^{(2)T} + b_1^{(2)}\right]\right),$

$$= \left[f(X W_1^{(1)T} + b_1^{(1)}), \quad f(X W_1^{(2)T} + b_1^{(2)})\right].$$

TENSOR PARALLELISM

- To compute the linear layer, we divide W_2 along its *columns*:
  - The first machine has the first $d/2$ columns of W_2.
  - The second machine has the last $d/2$ columns of W_2.

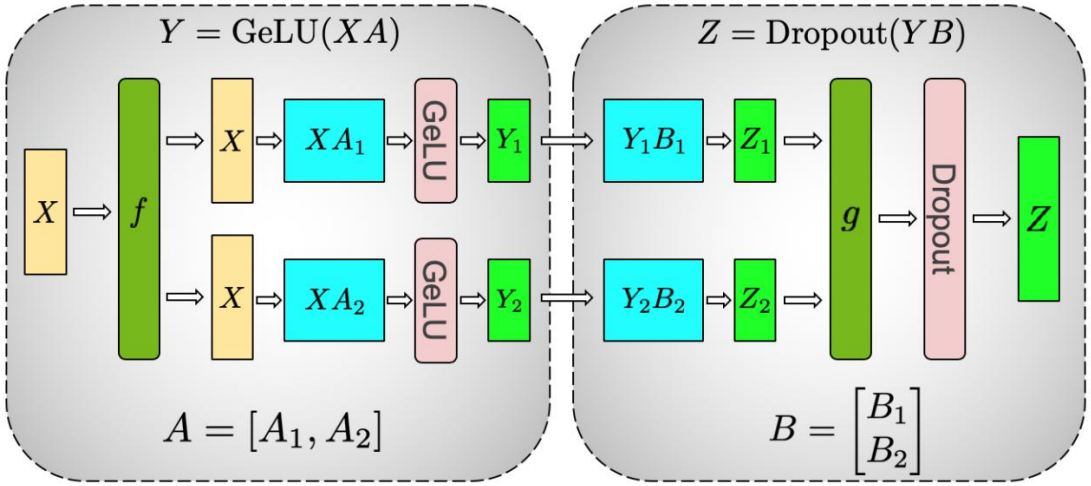
$$W_2 = \begin{bmatrix} W_2^{(1)} & W_2^{(2)} \end{bmatrix}$$

- Then $\left[f(X W_1^{(1)T} + b_1^{(1)}), \quad f(X W_1^{(2)T} + b_1^{(2)}) \right] W_2^T + b_2$

  $= f(X W_1^{(1)T} + b_1^{(1)}) W_2^{(1)T} + f(X W_1^{(2)T} + b_1^{(2)}) W_2^{(2)T} + b_2.$
- So each machine computes one term of this sum.
- We simply need to sum them together to get the correct output.

TENSOR PARALLELISM

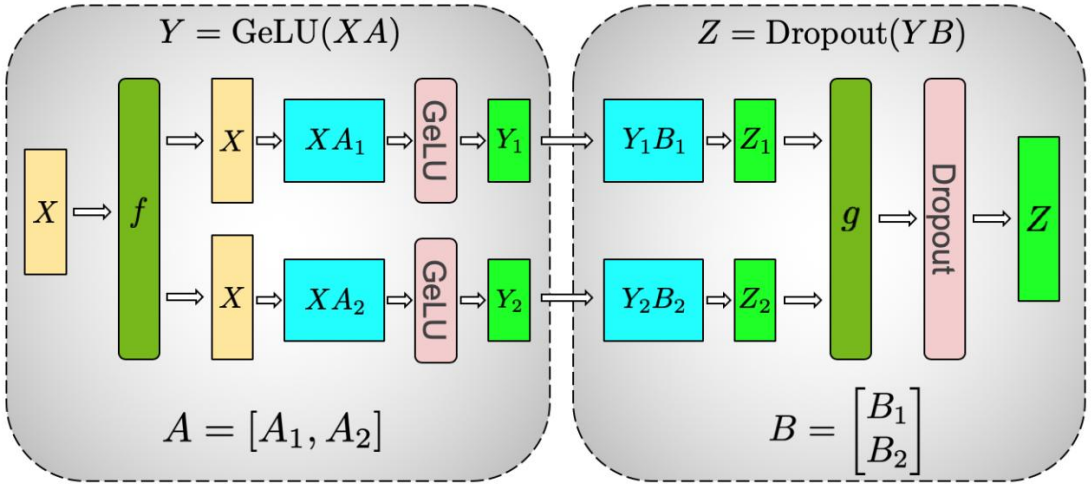
- Below is an example where the activation function is GeLU.
- The f block is an identity in the forward pass, whereas the g block is an all-reduce operation.
- All-reduce involves all machines sharing their tensor portions with each other, so each machine can reconstruct the full tensor.



[Shoeybi et al., 2020]

TENSOR PARALLELISM

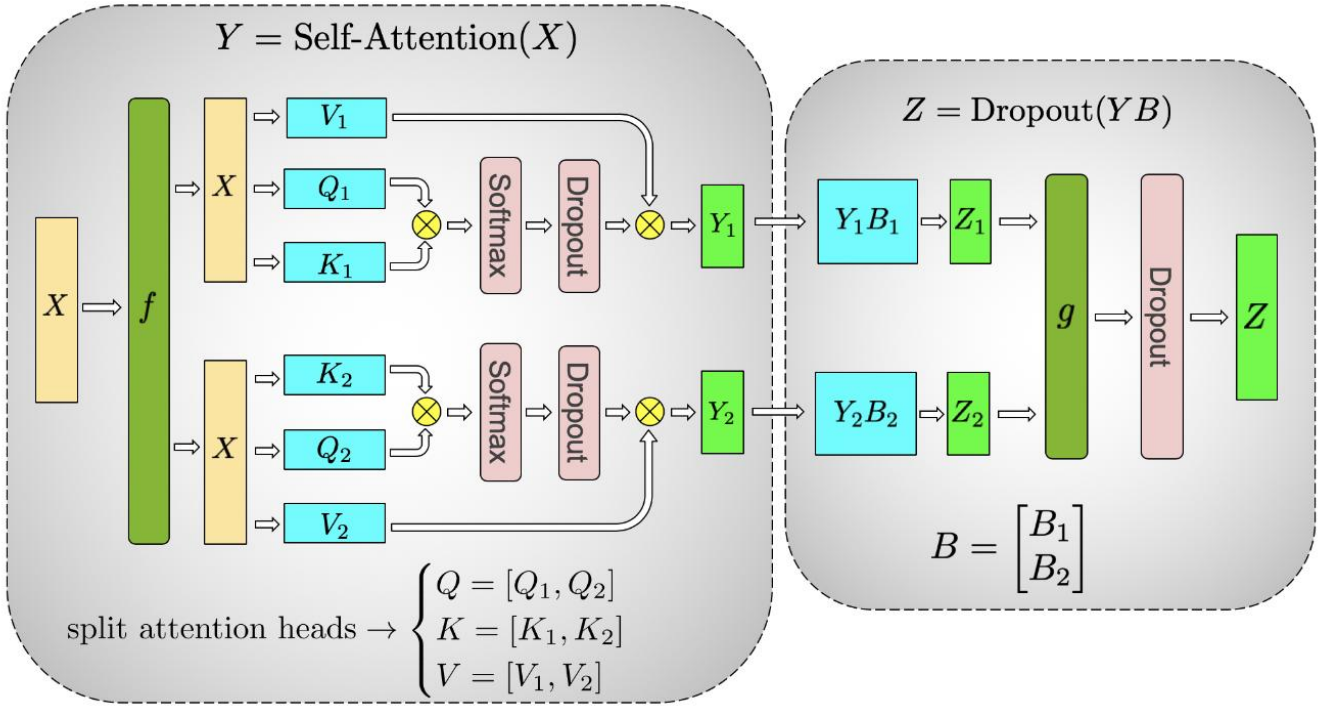
- All-reduce operations are required if the next operation needs the full tensor (e.g., LayerNorm, RMSNorm, softmax, dropout, etc).
- In the backward pass, the g block is an identity, whereas the g block is an all-reduce operation.



[Shoeybi et al., 2020]

TENSOR PARALLELISM

- What about the attention layer?
- Luckily, the attention heads provide a natural way to divide the work across multiple machines.



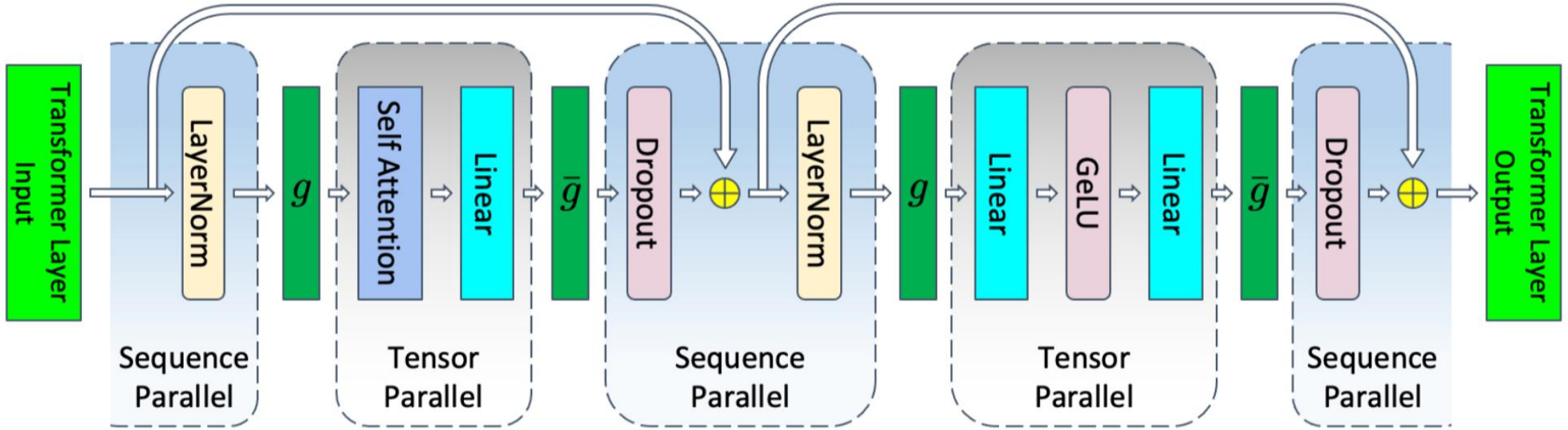
[Shoeybi et al., 2020]

TENSOR PARALLELISM

- This approach is called tensor parallelism, as we are dividing the weight and activation tensors across available GPUs.
  - This is a type of intra-layer model parallelism.
- But there are some operations that require the full input tensor, rather than one term in a sum:
  - E.g., LayerNorm, RMSNorm, softmax, dropout.
- Notice that these operations are all applied on each row of the input.
- So we could divide the rows of the input across machines, and each machine would compute the operation only on its assigned rows.
- This approach is called sequence parallelism (Korthikanti et al., 2022).
  - Since the rows correspond to tokens in the input sequence.

SEQUENCE PARALLELISM

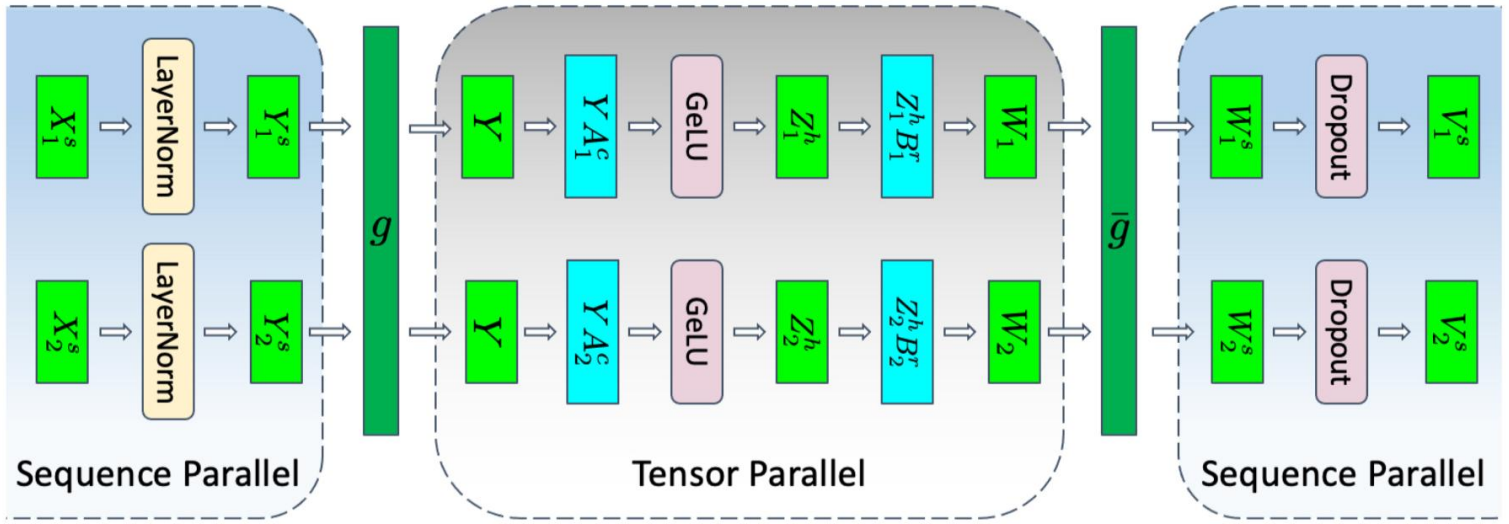
- This approach is often combined with tensor parallelism:



[Korthikanti et al., 2020]

SEQUENCE PARALLELISM

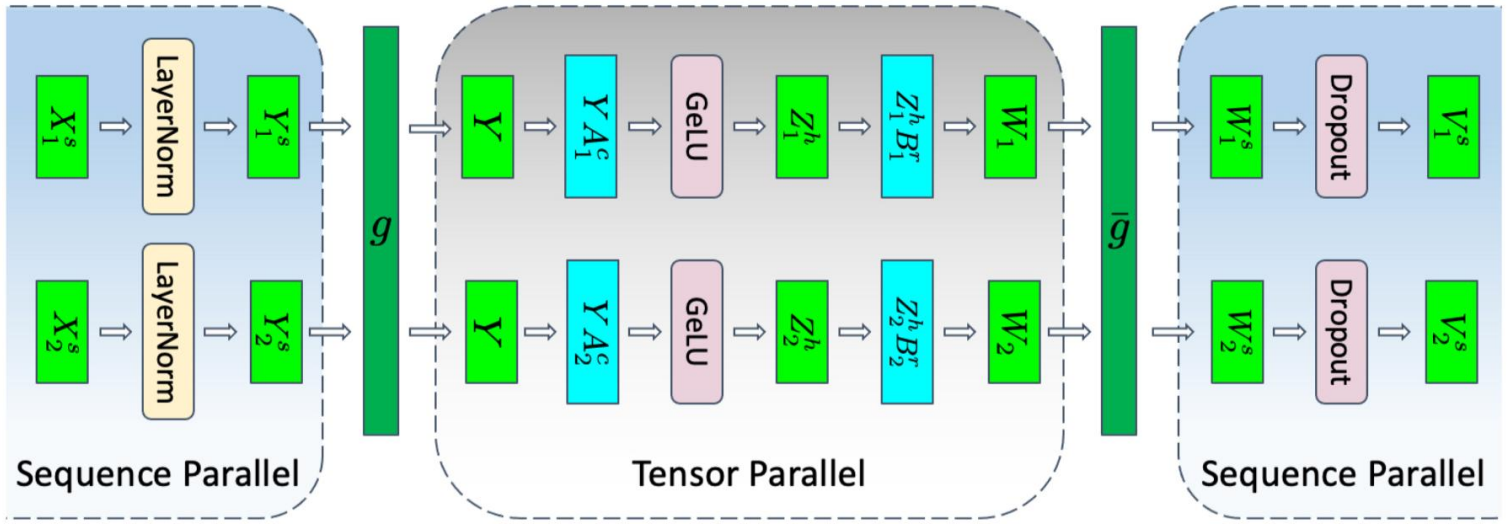
- Zooming into the FF (i.e., MLP) layer:
- Notice that g needs to convert from sequence to tensor parallelism.
  - It performs an all-reduce to reconstruct the full input to the FF layer.



[Korthikanti et al., 2020]

SEQUENCE PARALLELISM

- $\bar{g}$ needs to convert from tensor to sequence parallelism:
  - Perform an all-reduce to compute the sum of the terms from each machine.
  - Then “scatter” the rows of the resulting matrix across machines.



[Korthikanti et al., 2020]

TENSOR/SEQUENCE PARALLELISM

- Korthikanti et al., 2022, combined tensor and sequence parallelism with pipeline parallelism and were able to fit a 1T-parameter model on 512 GPUs.

Model Size	Attention Heads	Hidden Size	Layers	Tensor Parallel Size	Pipeline Parallel Size	Number of GPUs	Global Batch Size	Micro Batch Size
22B	64	6144	48	8	1	8	4	4
175B (GPT-3)	96	12288	96	8	8	64	64	1
530B (MT-NLG)	128	20480	105	8	35	280	280	1
1T	160	25600	128	8	64	512	512	1

- Compared to pipeline parallelism, there is much less idle time in tensor/sequence parallelism.
- However, the all-reduce operations require a lot of inter-device communication, which can be expensive.

INTER-DEVICE COMMUNICATION

- A node is a machine that has one motherboard, CPU, etc.
- Each node can have multiple GPUs
  - Which are connected to each other with some GPU interconnect.
  - PCIe is the most typical interconnect, but there are others.
- PCIe bandwidth is shared across all PCIe devices.
  - E.g., PCIe 6.0 has a total bandwidth of 242 GB/s.
  - So if GPU 1 is transferring data to GPU 2 at a rate of 200 GB/s, (hypothetically)
  - There will only be 42 GB/s of bandwidth remaining for any other transfers.

INTER-DEVICE COMMUNICATION

- NVLink is a proprietary interconnect developed by Nvidia.
- Infinity Fabric is a competing interconnect by AMD.
- Both technologies enable point-to-point communication between devices, and so bandwidth doesn't need to be shared.
- As you might expect, it is faster to read/write data on the GPU's high-bandwidth memory than it is to read/write data on another GPU.
  - E.g., For a P100 GPU, HBM bandwidth is 720 GB/s.
  - NVLink 1.0 has a peak theoretical bandwidth of 80 GB/s.

INTER-DEVICE COMMUNICATION

- Benchmarks of inter- vs intra-GPU bandwidth and latency: (4x P100 GPUs)

NVLink

```
Unidirectional P2P=Enabled Bandwidth Matrix (GB/s)
   D\D     0      1      2      3
     0 457.93  35.30  20.37  20.40
     1  35.30 454.78  20.16  20.14
     2  20.19  20.16 454.56  35.29
     3  18.36  18.42  35.29 454.07
```

PCIe

```
Unidirectional P2P=Enabled Bandwidth Matrix (GB/s)
   D\D     0      1      2      3
     0 452.19  10.19  10.73  10.74
     1  10.19 450.04  10.76  10.75
     2  10.91  10.90 450.94  10.21
     3  10.90  10.91  10.18 450.95
```

[Eshelman, Comparing NVLink vs PCI-E with NVIDIA Tesla P100 GPUs on OpenPOWER Servers, 2017]

INTER-DEVICE COMMUNICATION

- Benchmarks of inter- vs intra-GPU bandwidth and latency: (4x P100 GPUs)

NVLink

```
Unidirectional P2P=Enabled Bandwidth Matrix (GB/s)
   D\D     0      1      2      3
     0 457.93  35.30  20.37  20.40
     1  35.30 454.78  20.16  20.14
     2  20.19  20.16 454.56  35.29
     3  18.36  18.42  35.29 454.07

...

P2P=Enabled Latency Matrix (us)
   D\D     0      1      2      3
     0   4.99   7.92  15.56  15.43
     1   8.06   5.00  15.40  15.40
     2  15.47  15.52   5.04   8.07
     3  15.43  15.49   8.04   4.97
```

PCIe

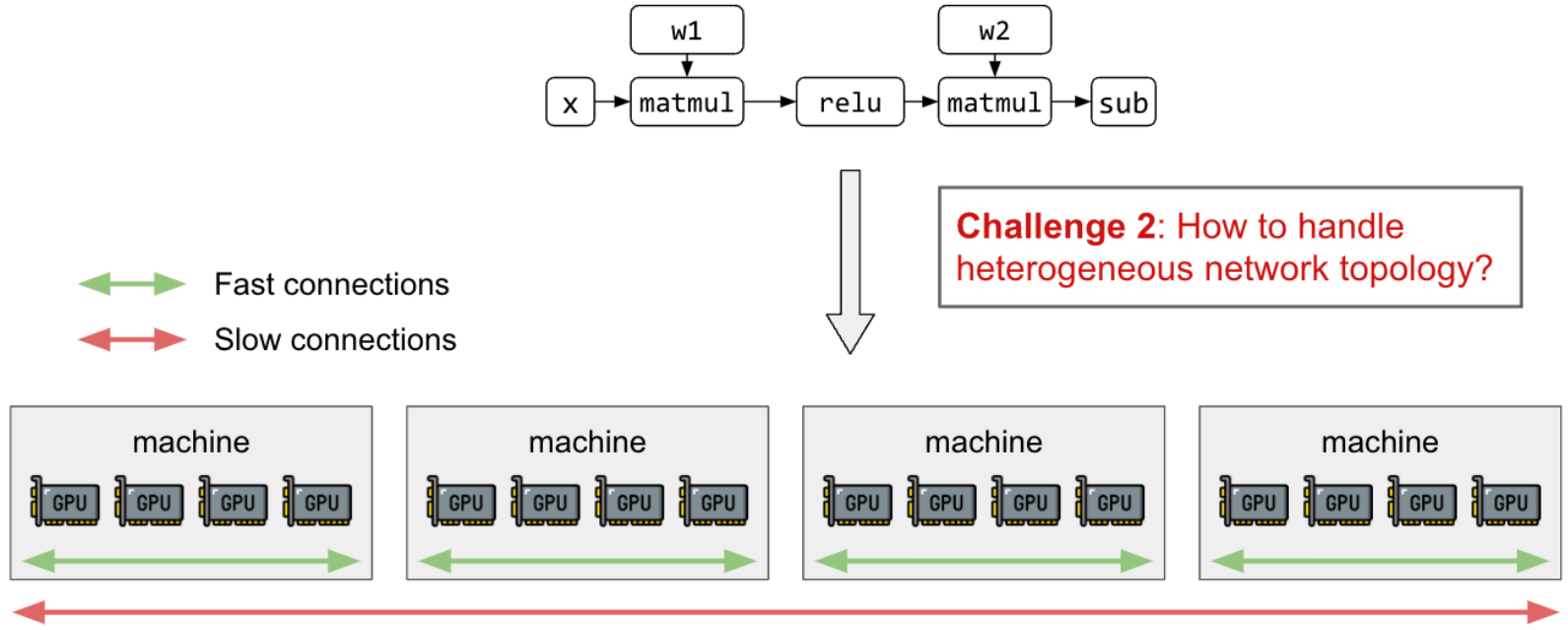
```
Unidirectional P2P=Enabled Bandwidth Matrix (GB/s)
   D\D     0      1      2      3
     0 452.19  10.19  10.73  10.74
     1  10.19 450.04  10.76  10.75
     2  10.91  10.90 450.94  10.21
     3  10.90  10.91  10.18 450.95

...

P2P=Enabled Latency Matrix (us)
   D\D     0      1      2      3
     0   3.22   7.86  16.90  17.05
     1   7.85   3.21  17.08  17.22
     2  16.32  16.37   3.07   7.85
     3  16.26  16.35   7.84   3.07
```

INTER-NODE COMMUNICATION

- Each node/machine can only be fitted with a limited number of GPUs.
  - It is difficult to effectively cool a machine if there are too many GPUs, each running at full utilization.
  - It is expensive to implement a high-performance interconnect if there are many GPUs on the same node.
  - (4 GPUs per node is typical)
- But we need hundreds or thousands of GPUs to train very large-scale models.
  - So we need a fast interconnect between nodes.
  - Ethernet is the most well-known, but there are other interconnects that have been developed for high-performance computing:
  - E.g., InfiniBand

INTER-NODE COMMUNICATION

- Inter-node communication is slower than intra-node communication.
- So we want to avoid frequent all-reduce operations that require inter-node communication.



AUTOMATIC PARALLELIZATION

- A better strategy could be to use tensor/sequence parallelism on individual nodes, and pipeline parallelism across nodes.
  - Assign layers to nodes.
  - Split the tensors in each layer across the GPUs in the same node.
- Software tools have been developed to automatically apply different parallelism techniques, considering the topology of the GPU cluster.
  - E.g., XLA auto sharding,
  - GSPMD (Xu et al., 2021)
  - PyTorch FSDP (Zhao et al., 2023)

DATA PARALLELISM

- Another form of parallelism that can be easily combined with the previous types of parallelism is called data parallelism.
- Suppose we have a parallelized model that runs with a batch size of 1 on n nodes.
- If we have Bn nodes available, we can simply increase the batch size linearly to B,
  - By dividing the batch into minibatches, where each minibatch is given to a group of n nodes.
- This is necessary for training since larger models require a larger batch size for compute-optimal training.
- Data parallelism is very useful for increasing throughput during inference.

WHAT IF WE DON'T HAVE 1000 GPUS?

- We don't all have access to massive clusters with thousands of GPUs.
- How do we train or run inference on models that are too large for 1 or 4 GPUs?
- We can use approximation to reduce the memory footprint of the model.
- Approximation may lead to a cost in accuracy.
- Suppose we have a model that is small enough to fit in our memory for inference, but too large for training.
  - (recall the memory cost of training is significantly larger than that for inference)

APPROXIMATING MATRIX PRODUCTS

- The weight matrices of the linear layers are the largest contributors to a model's memory footprint.
- During training, in a linear layer, we compute the forward pass:

$$f(X) = XW^T + b.$$

- In the backward pass, we compute the gradient of the loss with respect to the parameters W and b.
- Then we update the values of W and b according to the step size γ.

$$W_{new} = W - \gamma \frac{\partial L}{\partial W}.$$

APPROXIMATING MATRIX PRODUCTS

- We can gather all the gradient updates into a single ΔW term:

$$W_{new} = W + \Delta W.$$

- This way, during training, we keep W unchanged and only keep track of ΔW.
- So the forward pass becomes:

$$f(X) = X(W + \Delta W)^T + b.$$

- Let's try to approximate ΔW using a product of smaller matrices:

$$\Delta W = AB,$$

- Where ΔW has dimension $d \times d$, A has dimension $d \times r$, and B has dimension $r \times d$, where r is much smaller than d.

APPROXIMATING MATRIX PRODUCTS

- Thus our forward pass is now:

$$f(X) = X(W + AB)^T + b$$

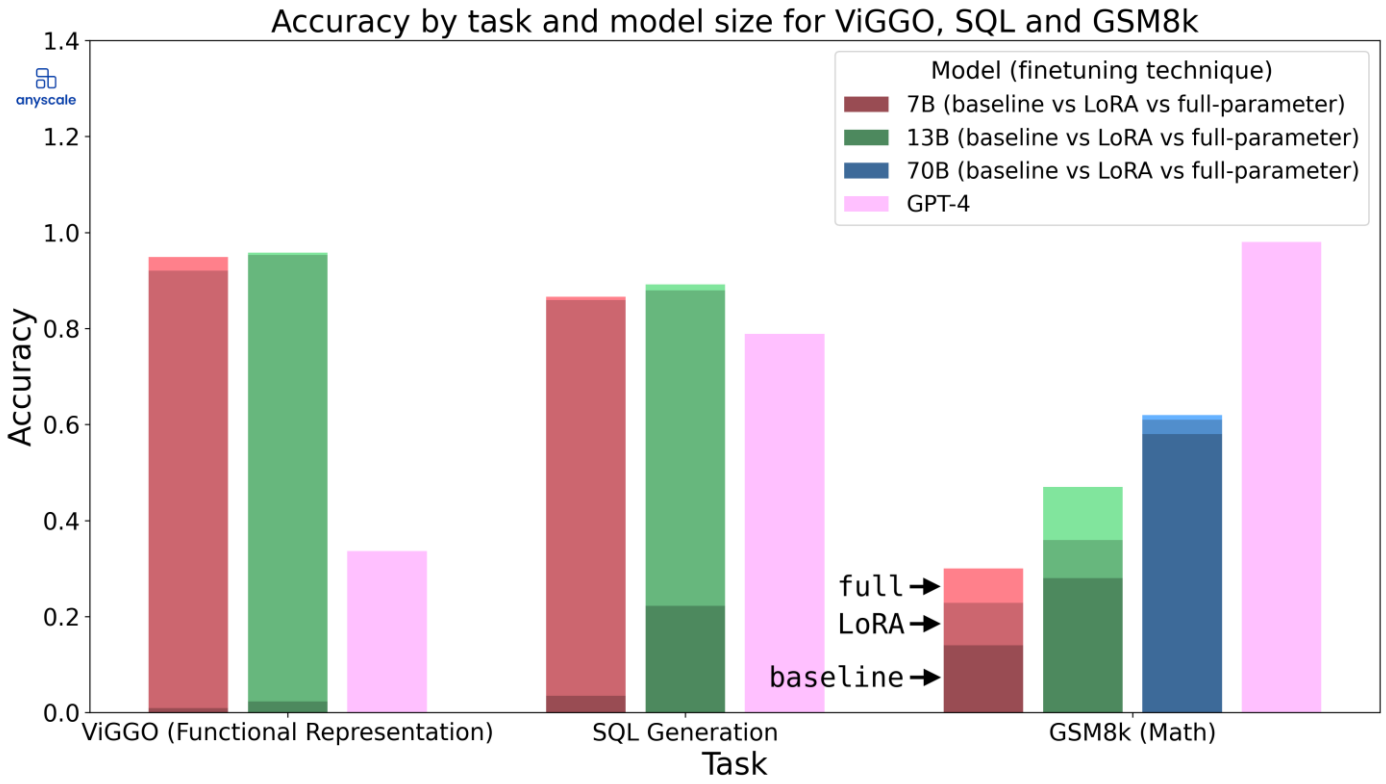
- Where W is a constant and only A and B are learnable parameters.
- So how many training parameters do we have?
  - Previously, we had $d^2 + d$ per linear layer.
  - Now, we have $2dr + d$, which can be much smaller when r is small.

LORA

- This approach is called Low-Rank Adaptation or LoRA (Hu and Shen et al., 2021).
- It works well if the changes to the weight matrix have low rank, which is often true in fine-tuning.
  - But this is not true in pretraining, where the learned weight matrices have high rank.
  - Thus LoRA is only used for fine-tuning.

LORA

- Fine-tuning vs LoRA experiments on Llama-2: ($r = 8$)



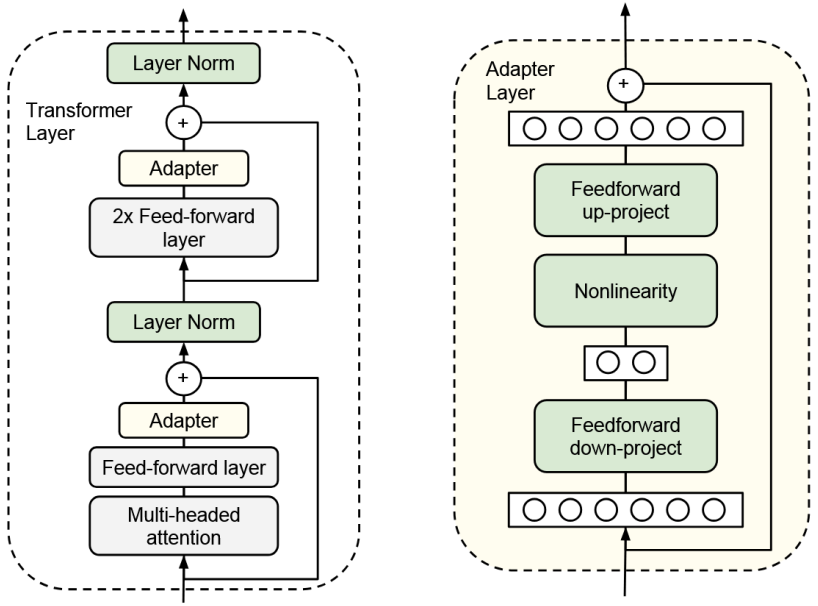
[Niederfahrenhorst et al., Fine-Tuning LLMs: LoRA or Full-Parameter? An in-depth Analysis with Llama 2, 2023]

PEFT

- LoRA is an example of parameter-efficient fine-tuning (PEFT).
- There are many other PEFT methods, and research into new methods is ongoing.
- One simple PEFT approach is to freeze all layers of the model except for the last layer.
  - This was a typical approach for fine-tuning BERT.
  - A simple extension is to fine-tune the last k layers.

PEFT

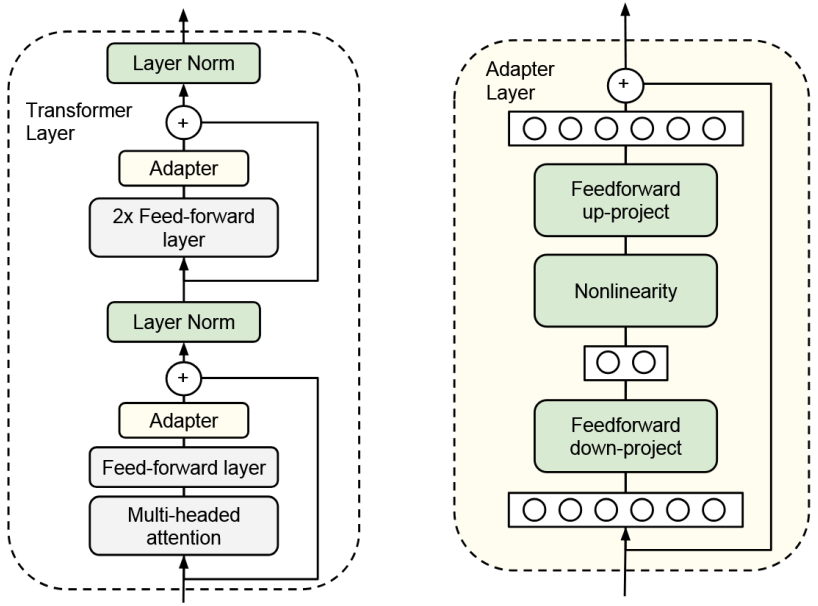
- LoRA is an example of parameter-efficient fine-tuning (PEFT).
- There are many other PEFT methods, and research into new methods is ongoing.
- Another class of PEFT methods are called adapter methods.



[Houlsby et al., 2019]

ADAPTER METHODS

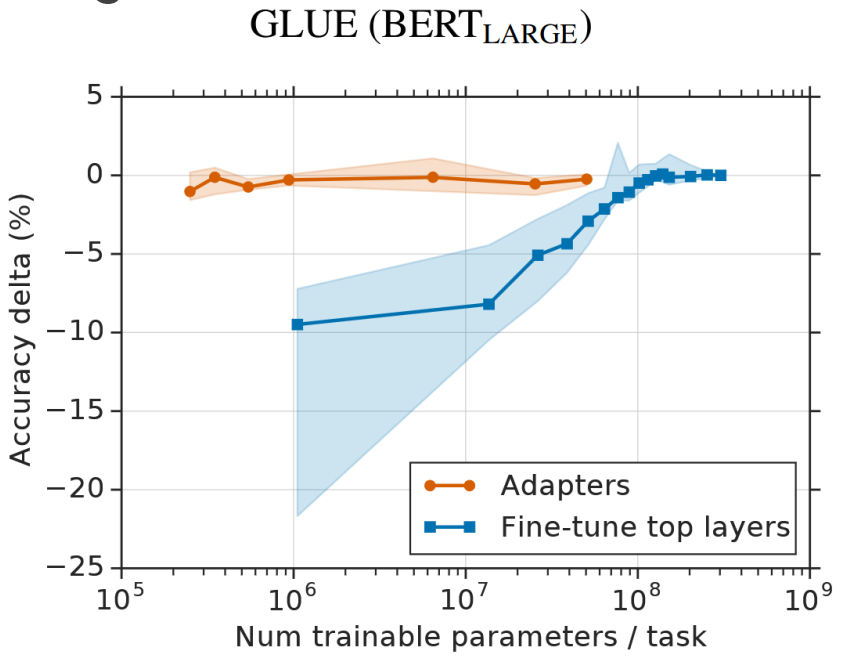
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

ADAPTER METHODS

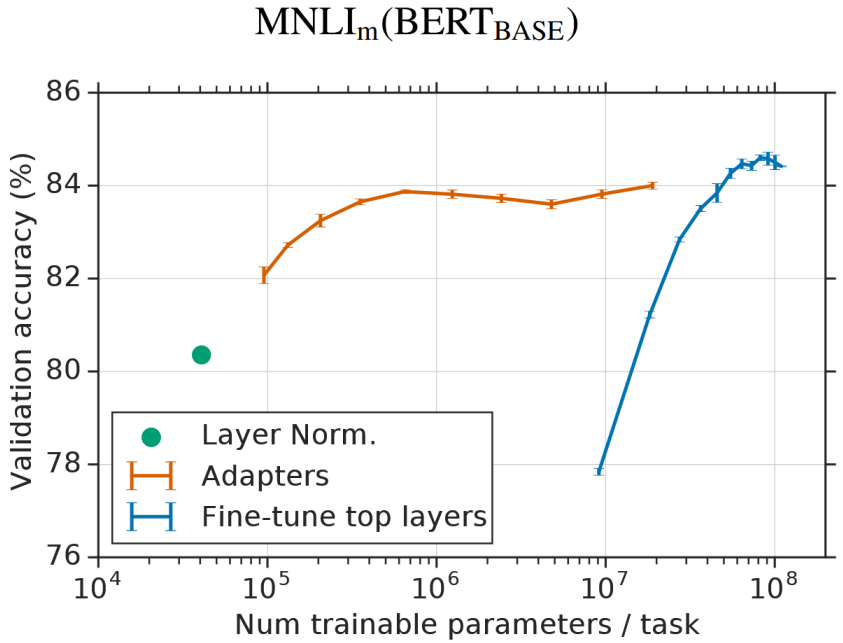
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

ADAPTER METHODS

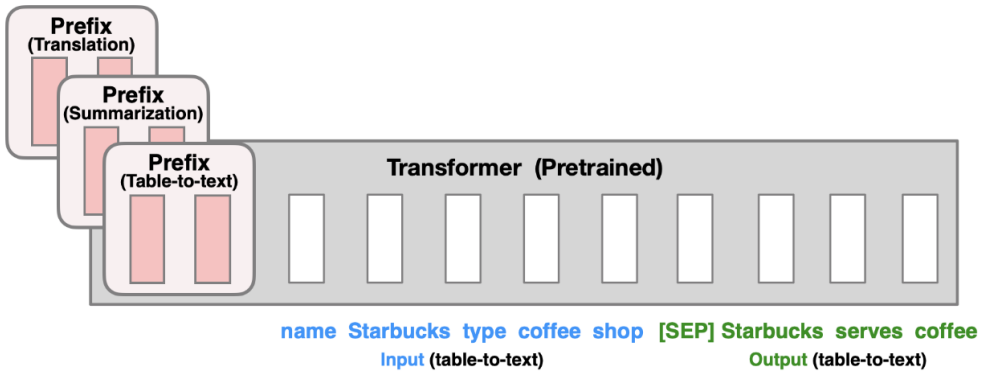
- The idea is to add a small network inside the attention and FF layers.
  - Called the “adapter.”
- Keep all model parameters fixed except for those in the adapter, which is much smaller than the original model.



[Houlsby et al., 2019]

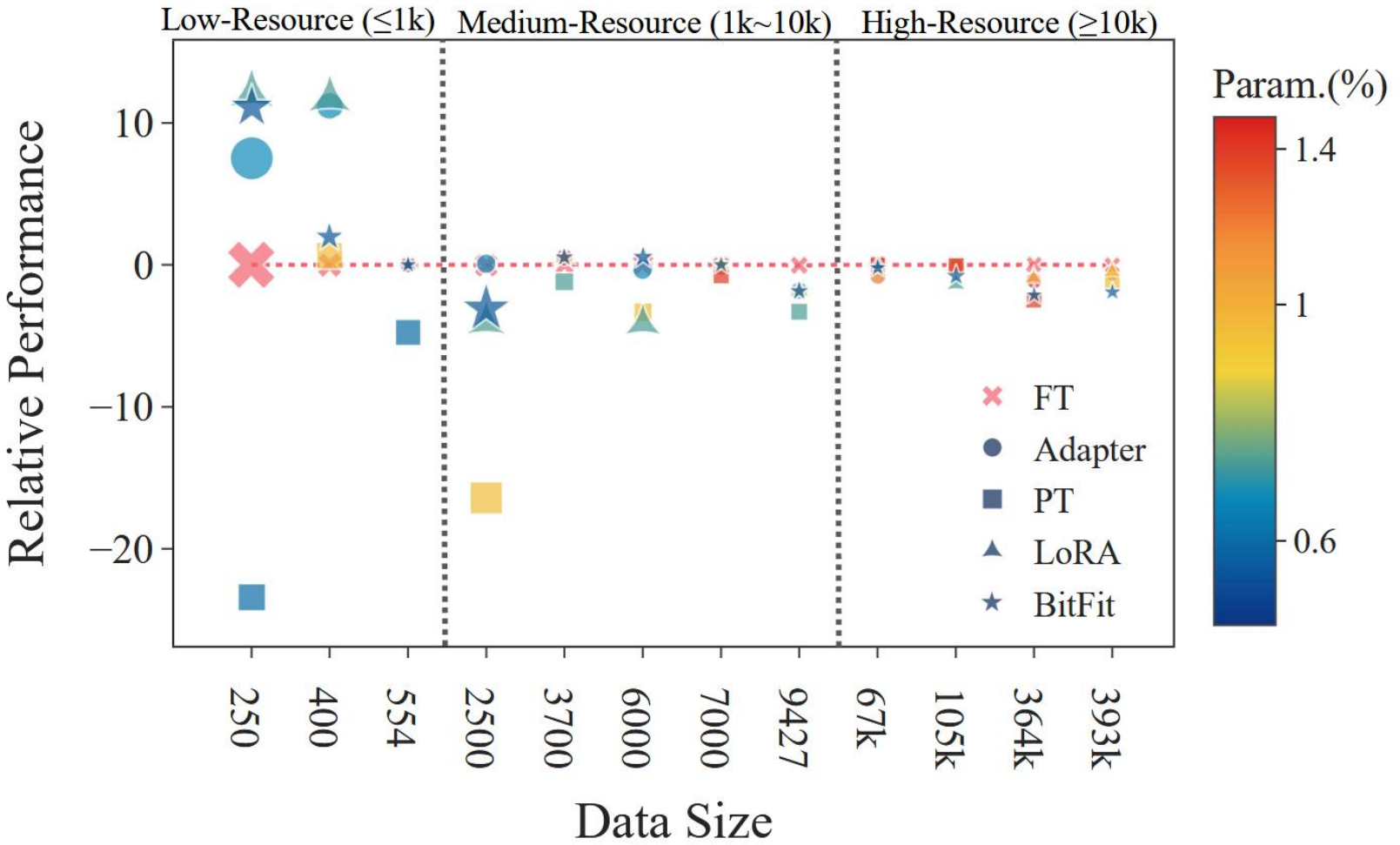
PREFIX TUNING

- Adapter methods and LoRA are similar in that they freeze the original model's parameters
  - And instead added a small number of trainable parameters.
- Prefix tuning is another PEFT method where new tokens are added to the beginning of the input prompt.
  - Unlike regular text tokens, these added tokens are continuous.
  - Their embeddings are the only trainable parameters in the model.



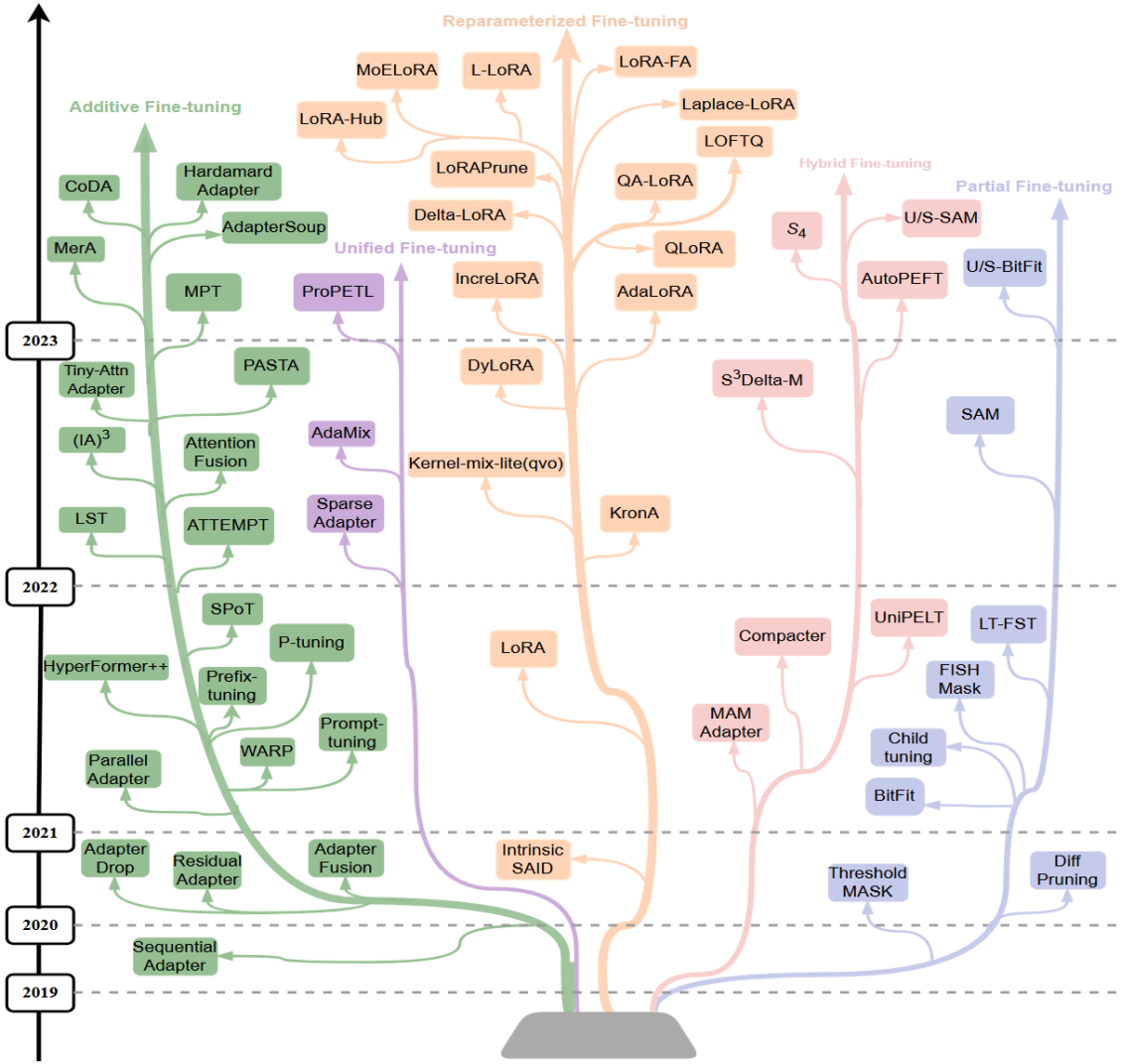
[Li and Liang, 2021]

PEFT



[Chen et al., 2022]

PEFT



[Xu et al., 2023]

NEXT TIME

- Reinforcement learning
  - Can we provide more informative supervision during post-training?
- After: What if the original model is too large to fit in memory even for inference alone?
- Can we make the model smaller?
  - Quantization: Reduce the precision of the floating-point numbers in the model.
    - How can we reduce the precision without adversely affecting the model's accuracy?
    - It's not so simple, especially with very low precision.
  - Distillation: Use a larger model to train a small model.

QUESTIONS?