香港城市大學(東莞)
City University of Hong Kong (Dongguan)

Celebrating the Establishment in Year 2024

Lecture 10: Efficient Training of LLMs

CS6493 Natural Language Processing

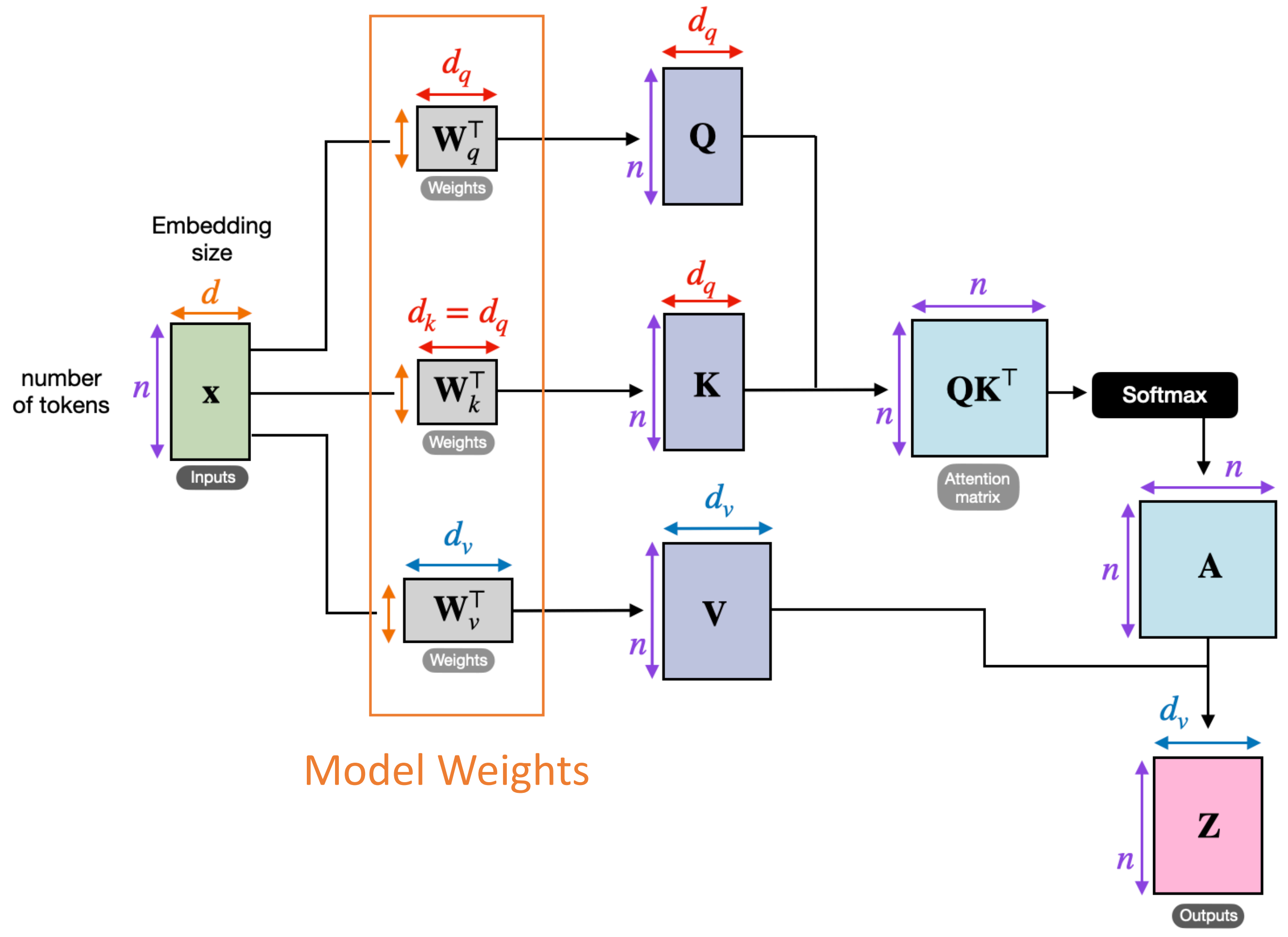
Instructor: Linqi Song

Outline

- Background
- Model Parallelism
- Data Parallelism
- Parameter-Efficient Fine-Tuning

Background- Model Weights

- Weights are the parameters of the model that are learned during training.
- They are adjusted by the optimizer based on the gradients to minimize the loss function.
- In transformers, weights include the parameters in self-attention, position-wise feed-forward networks, and layer normalization.



Background- Where did all the memory go?

Model weights take 2 bytes (16 bits) at fp16 and 4 bytes (32 bits) at fp32.

Consider a 1.5B parameter GPT-2 model requires 3GB of memory for its model weights in 16-bit precision.

It cannot be trained on a single GPU with 32GB memory using naïve Tensorflow or PyTorch due to the out-of-memory(OOM) issue.

Where did all the memory go?

Background mixed precision training

- To enabling the use of the high throughput tensor core units of GPUs, model weights are stored as fp16.
- During mixed-precision training, both the forward and backward propagation are performed using fp16.
- However, to effectively compute and apply the updates at the end of the backward propagation, an fp32 copy of the parameters and all the other optimizer states are kept.

Background: optimizer states

- Optimizer state refers to the internal data stored by the optimizer during the training process. This data is used to update the model's weights efficiently.
- Adam optimizer need to maintain one or two optimizer states (the time averaged momentum and variance of the gradients to compute the updates) per each parameter.
- As the model size grows, the memory consumed by optimizer states can be a dominating factor of memory consumption.

Memory consumption

- Consider a model with Ψ parameters.
- During fp16 training, model parameters need 2Ψbytes to store the model weights, 2Ψbytes for the corresponding gradients.
- Adam’s Optimizer States:
    - fp32 copy of parameters: 4Ψ
    - fp32 copy of momentum:4Ψ
    - fp32 copy of variance: 4Ψ
    - In total: $4\Psi + 4\Psi + 4\Psi = 12\Psi$

- Total consumption: $2\Psi + 2\Psi + 12\Psi = 16\Psi$

Background- Where did all the memory go?

Model weights take 2 bytes (16 bits) at fp16 and 4 bytes (32 bits) at fp32.

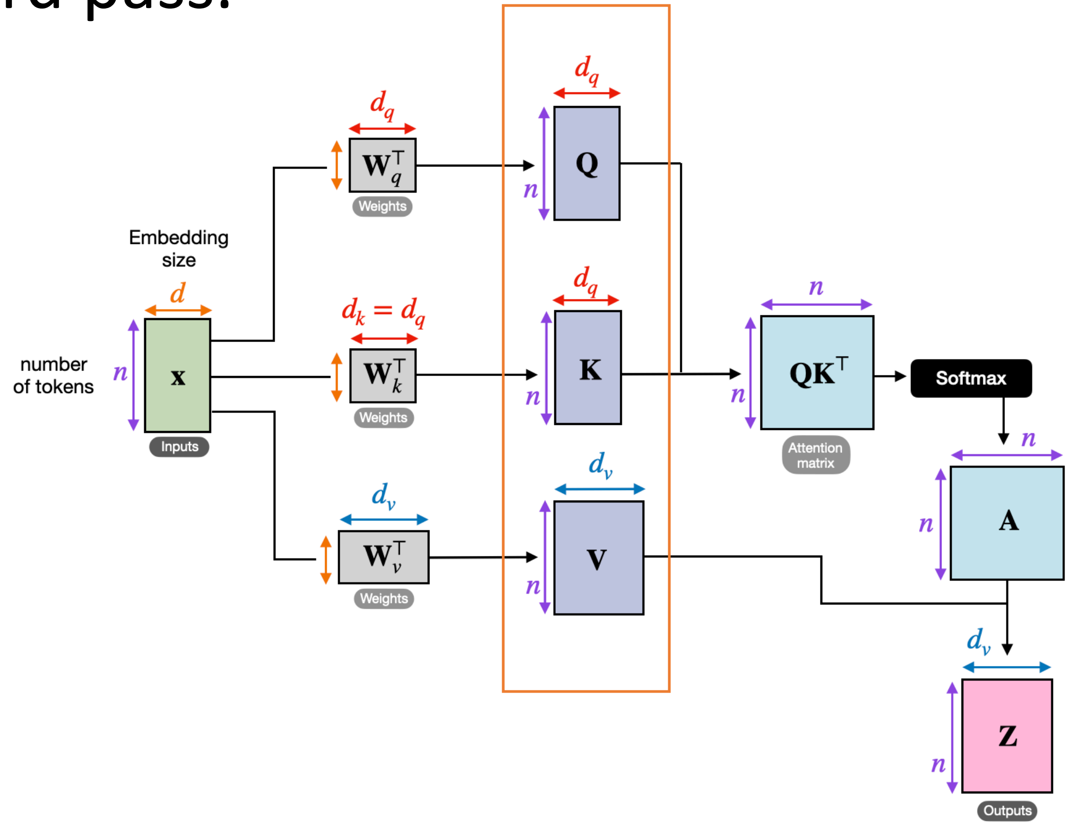
Consider a 1.5B parameter GPT-2 model requires 16 * 1.5 = 24GB of memory for its model weights, gradients, and optimizer states in 16-bit precision.

It cannot be trained on a single GPU with 32GB memory using naïve Tensorflow or PyTorch due to the out-of-memory(OOM) issue.

16 * 1.5 = 24GB <= 32GB, why OOM?

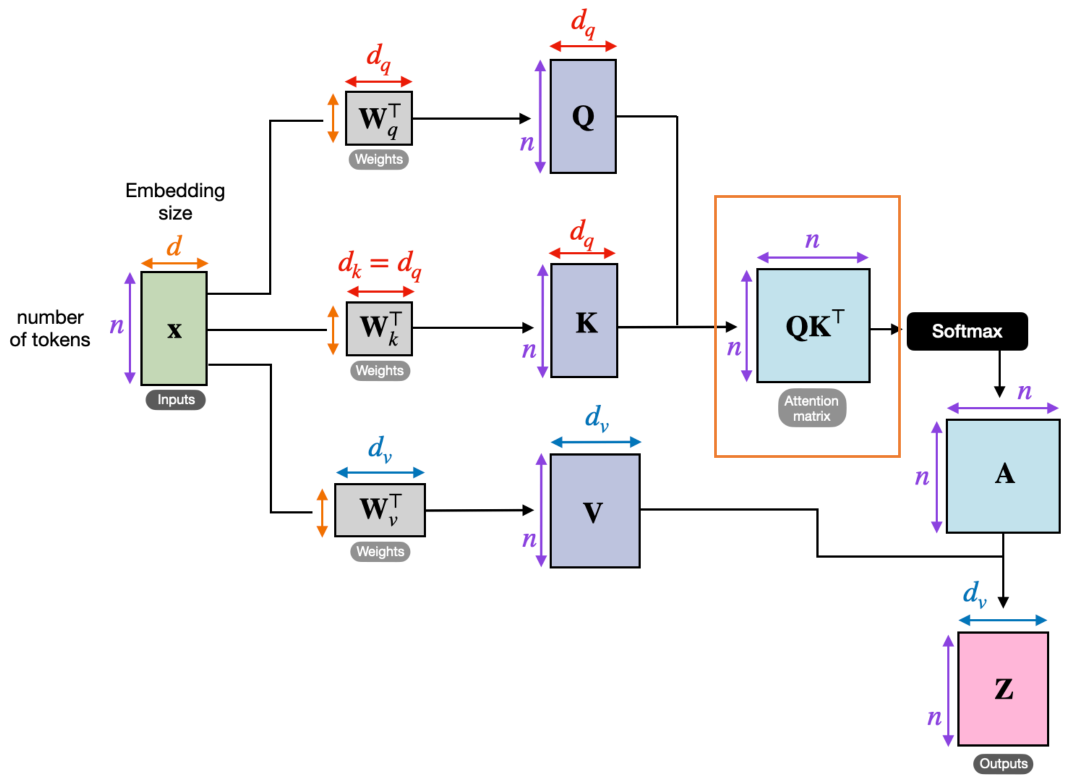
Background – Activation (intermediate results)

- The intermediate results stored from forward pass in order to perform backward pass.



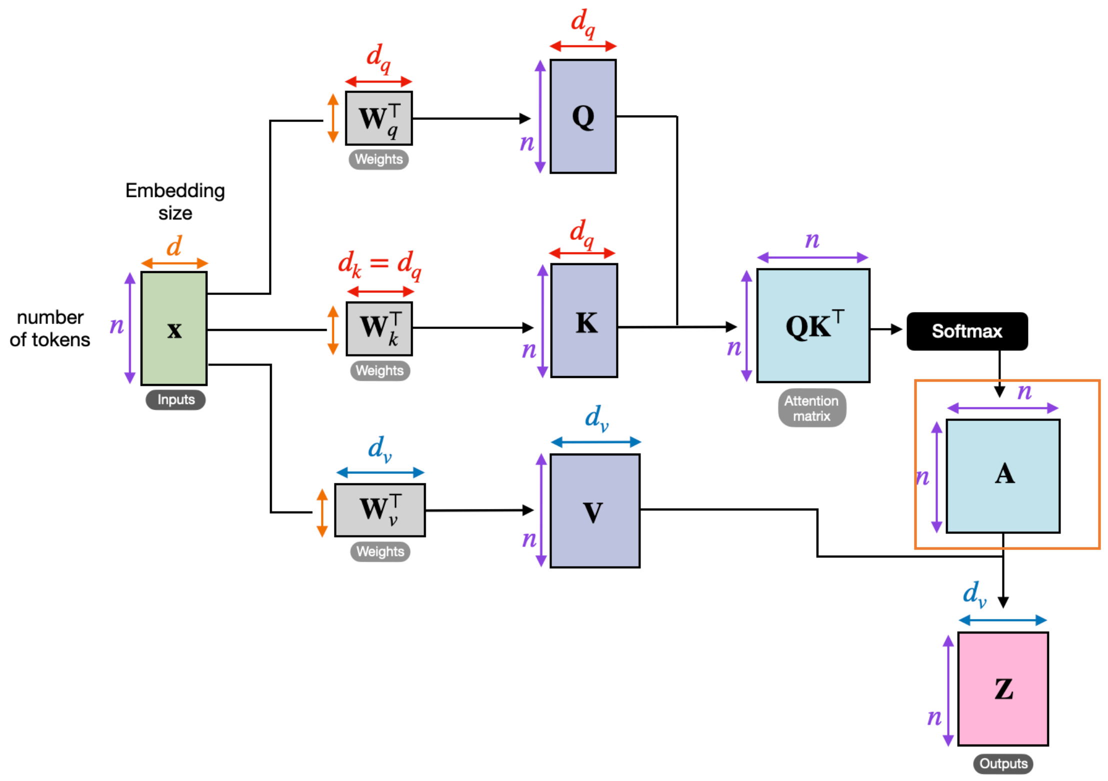
Background – Activation (intermediate results)

- The intermediate results stored from forward pass in order to perform backward pass.



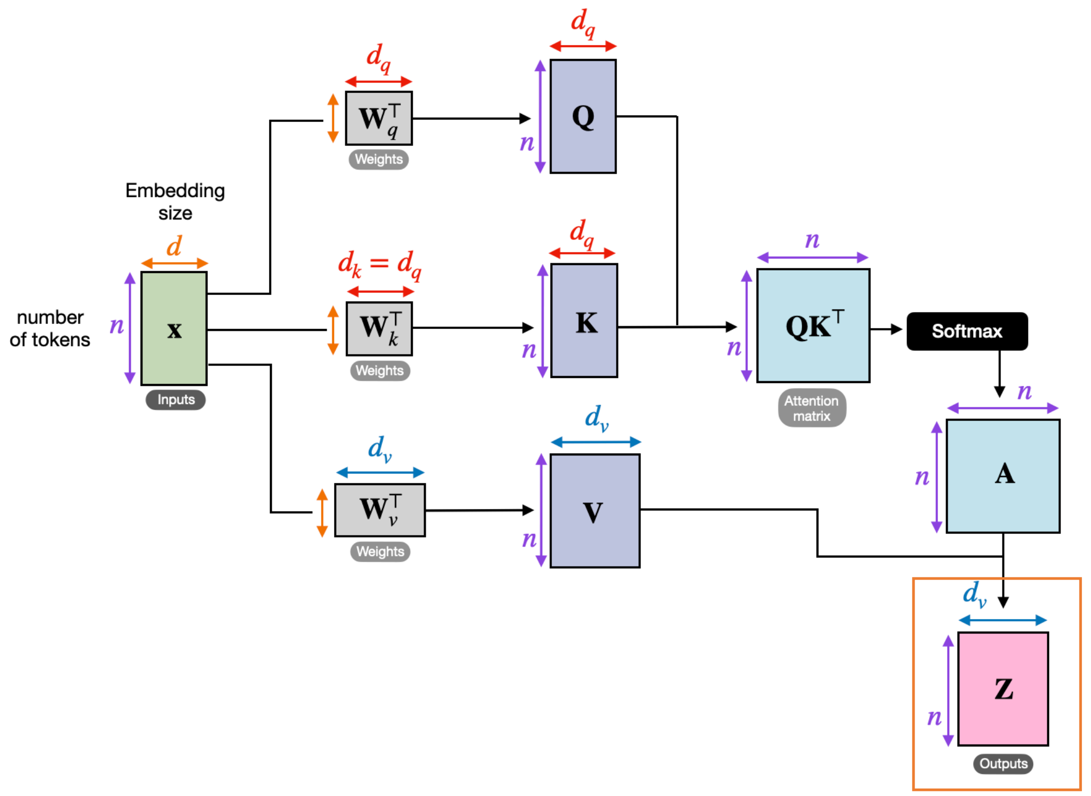
Background – Activation (intermediate results)

- The intermediate results stored from forward pass in order to perform backward pass.



Background – Activation (intermediate results)

- The intermediate results stored from forward pass in order to perform backward pass.



Background – Activation (intermediate results)

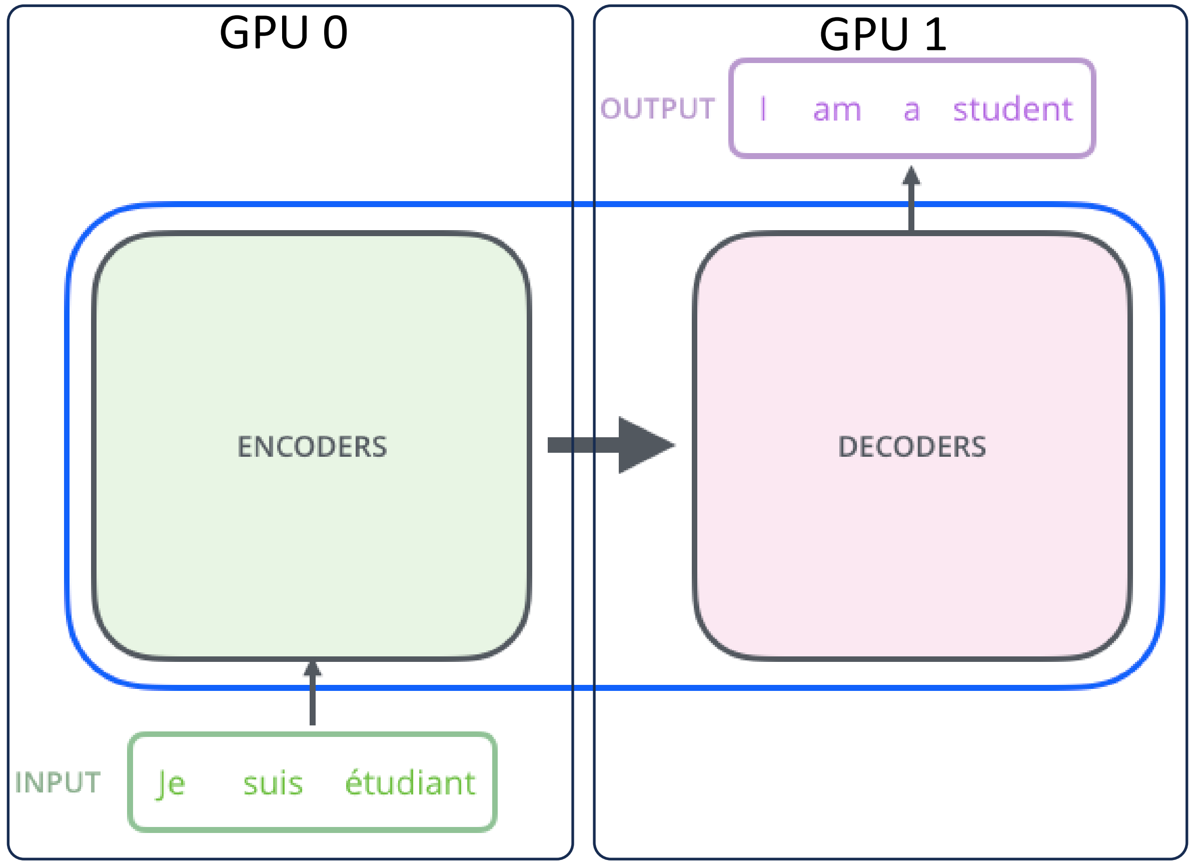
- The intermediate results stored from forward pass to perform backward pass.
- The activation memory of a transformer-based model is proportional to the number of

$$transformer\ layers \times\ hidden\ dimensions \times\ sequence\ length \times\ batch\ size.$$

- As a concrete example, the 1.5B parameter GPT-2 model trained with sequence length of 1K and batch size of 32 requires about 60GB of memory.

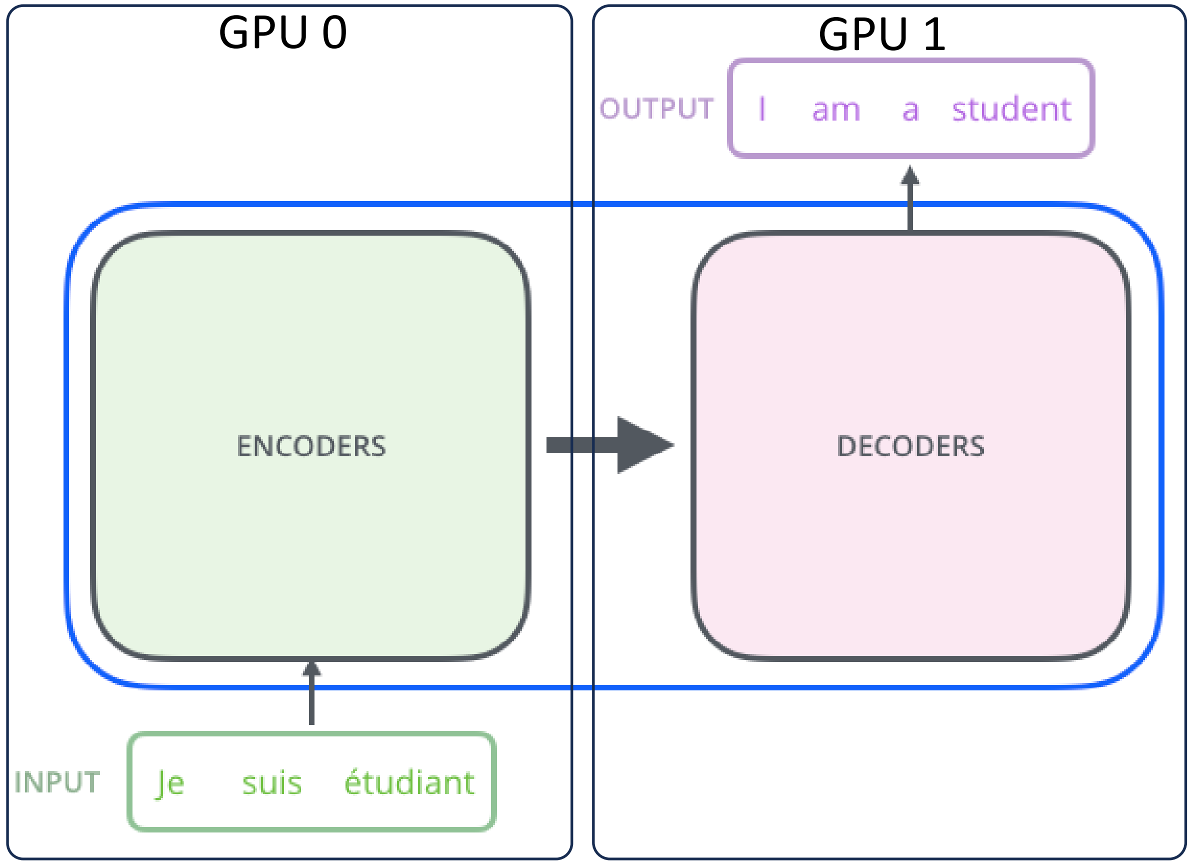
Model parallelism (MP)

- Naïve Model Parallelism (MP) – Vertical partition, spreads groups of model layers across multiple GPUs.



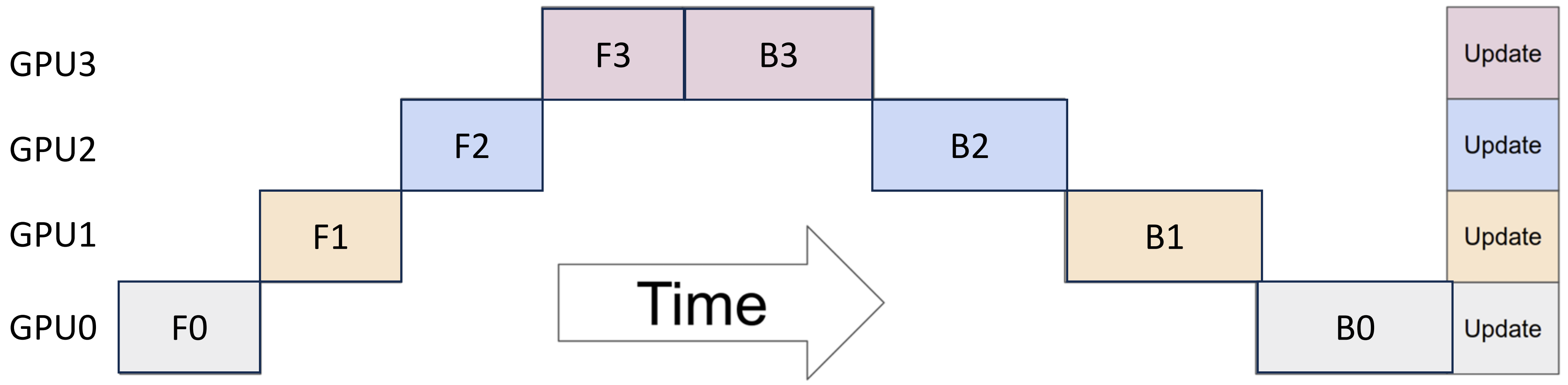
Naïve MP

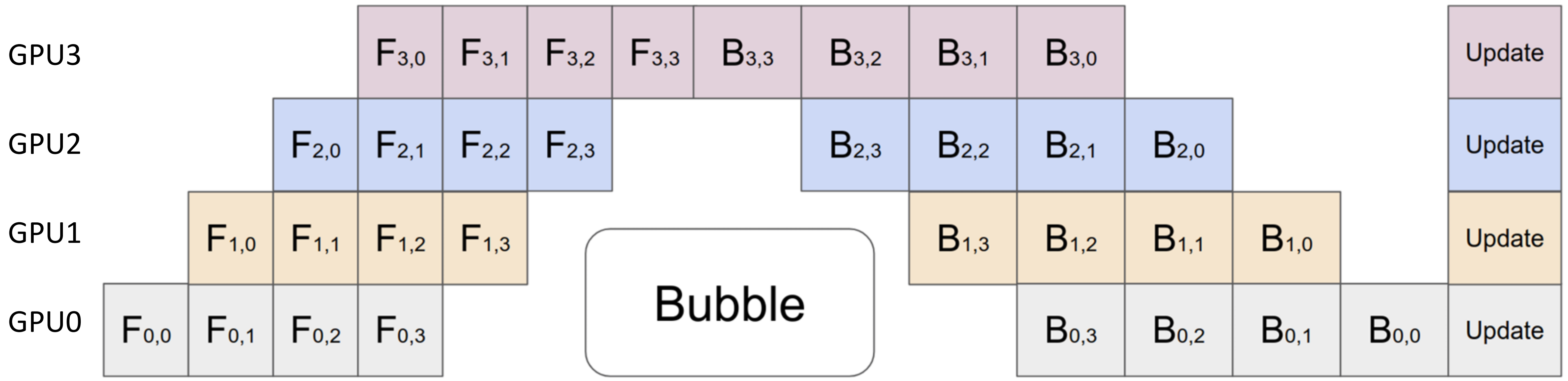
- While data travels from layer inside Encoders, this is just the normal model.
- But when data needs to pass from encoder to decoder it needs to travel from GPU0 to GPU1 which introduces a communication overhead.



Naïve MP

- Advantage:
  - fit very large models onto limited hardware
- Disadvantage:
  - GPU idle (Bubbles).

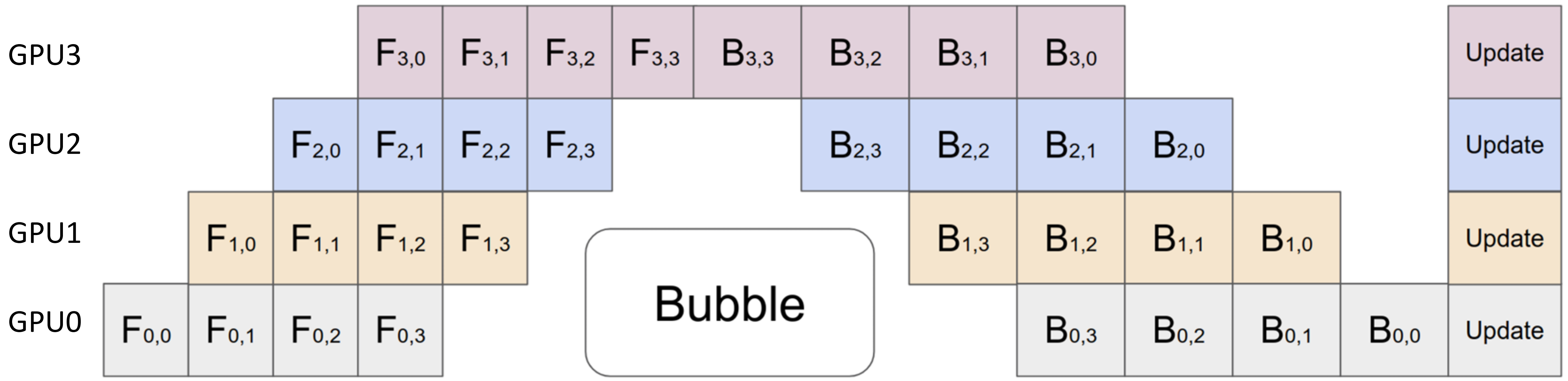




GPU3
GPU2
GPU1
GPU0
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ Update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ Update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ Update
Bubble

GPipe

- Split a batch of training examples into smaller micro-batches, then pipeline the execution of each set of micro-batches over cells.
- Each Cell is then placed on the GPUs.

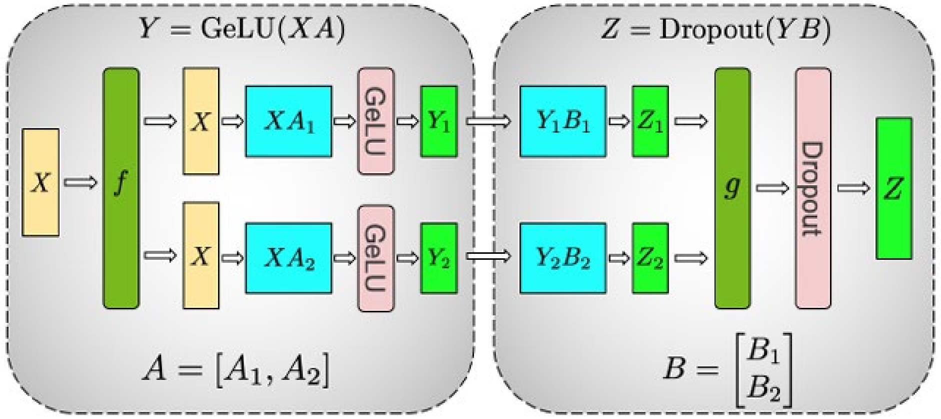

GPU3
GPU2
GPU1
GPU0
$F_{3,0}$ $F_{3,1}$ $F_{3,2}$ $F_{3,3}$ $B_{3,3}$ $B_{3,2}$ $B_{3,1}$ $B_{3,0}$ Update
$F_{2,0}$ $F_{2,1}$ $F_{2,2}$ $F_{2,3}$ $B_{2,3}$ $B_{2,2}$ $B_{2,1}$ $B_{2,0}$ Update
$F_{1,0}$ $F_{1,1}$ $F_{1,2}$ $F_{1,3}$ $B_{1,3}$ $B_{1,2}$ $B_{1,1}$ $B_{1,0}$ Update
$F_{0,0}$ $F_{0,1}$ $F_{0,2}$ $F_{0,3}$ $B_{0,3}$ $B_{0,2}$ $B_{0,1}$ $B_{0,0}$ Update
Bubble

GPipe

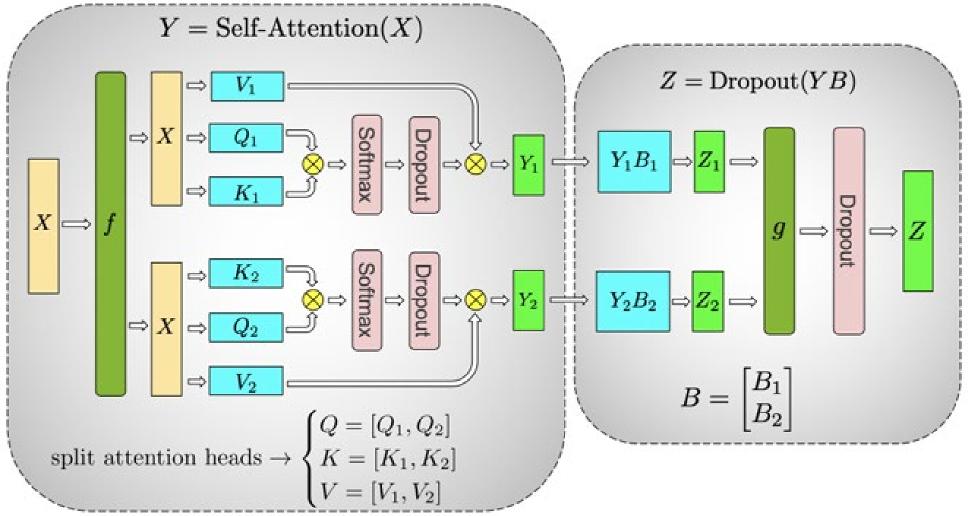
- Advantage:
  - Less bubble, make the most use of GPUs.
- Disadvantage:
  - Need to experiment to find the size of micro-batch that leads to the highest efficient utilization of the GPUs.

Modern MP – MegatronLM (Tensor Parallelism)

- In Tensor Parallelism each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.
- It parallize the tensor in column- and row-wise methods.



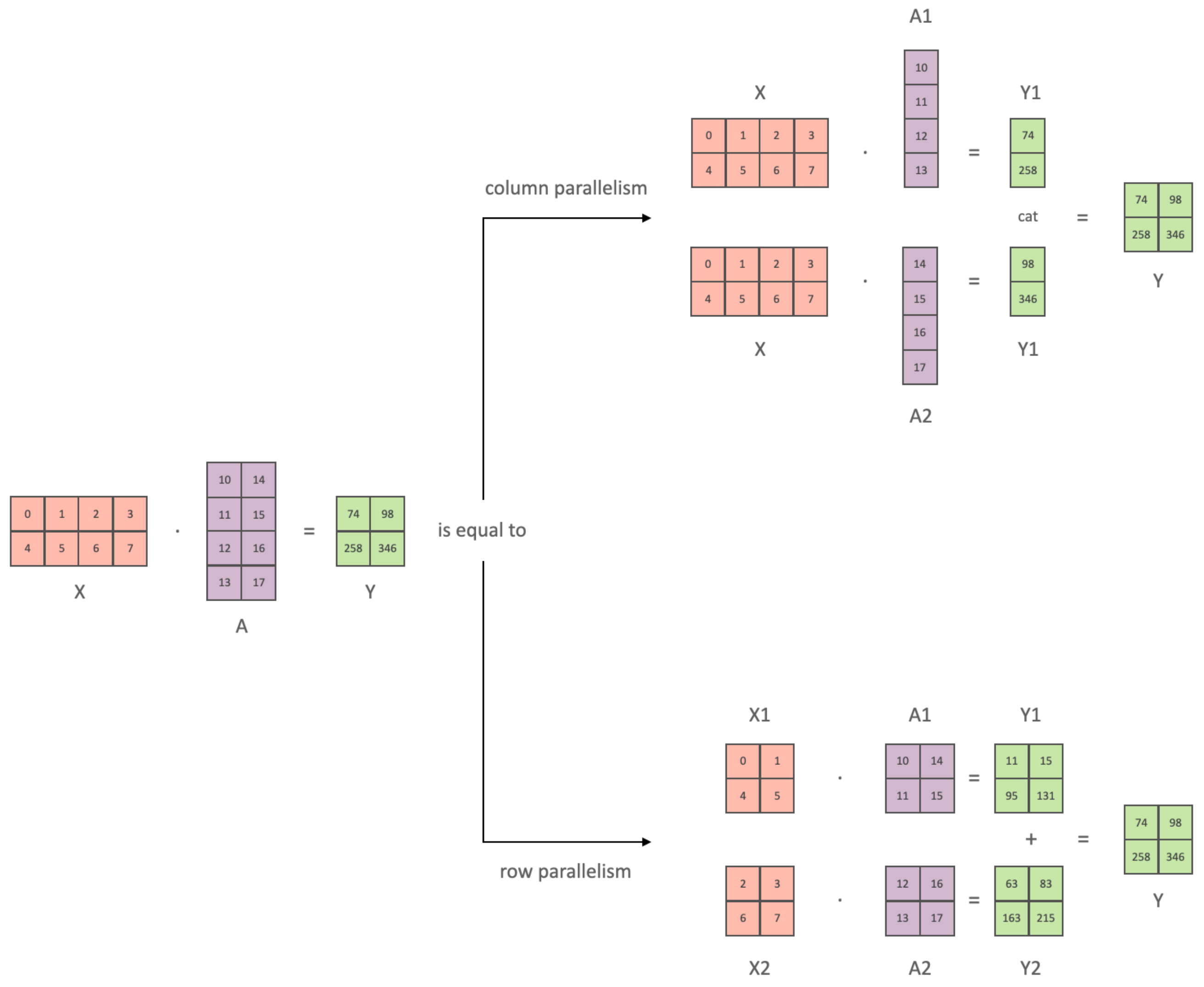
(a) MLP


$Y = \text{Self-Attention}(X)$
$Z = \text{Dropout}(YB)$
X
f
V_1
Q_1
K_1
K_2
Q_2
V_2
Softmax
Dropout
Y_1
Y_2
Y_1B_1
Y_2B_2
Z_1
Z_2
g
Z
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
split attention heads $\rightarrow$ $\begin{cases} Q = [Q_1, Q_2] \\ K = [K_1, K_2] \\ V = [V_1, V_2] \end{cases}$

(b) Self-Attention

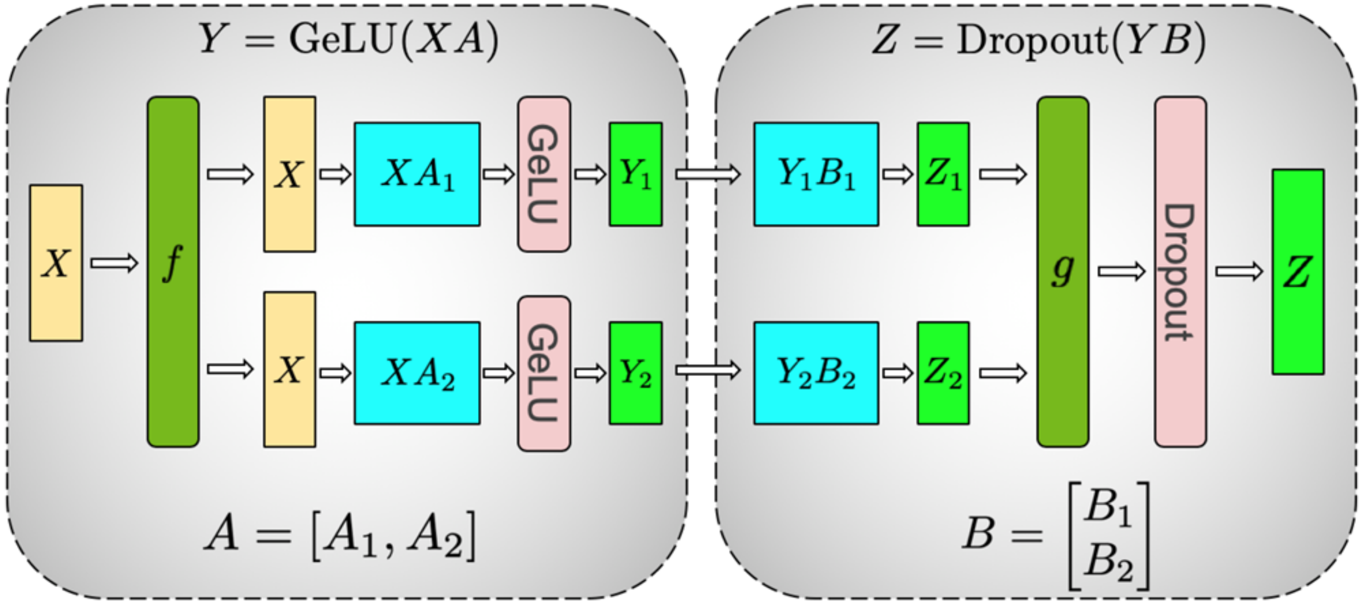
Tensor parallelism (TP)

- Core idea: matrix multiplication can be split between multiple GPUs



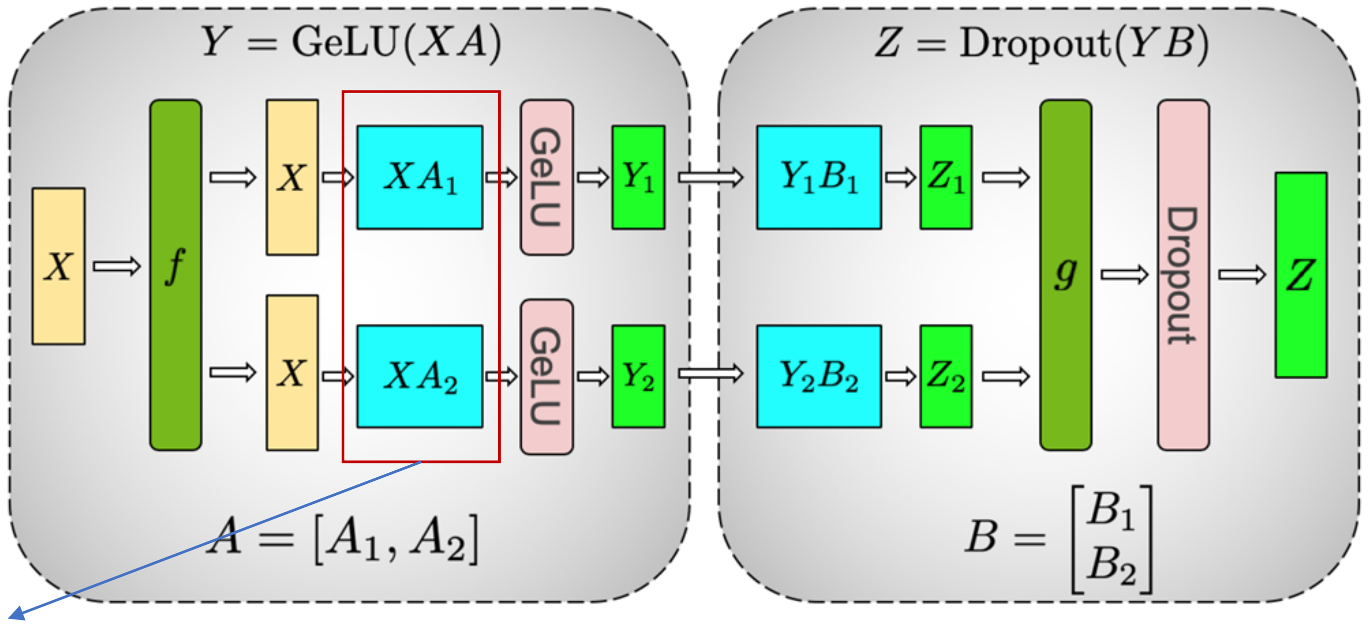
TP for MLP layer

- we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end, where we need to reconstruct the output vector from shards.



TP for MLP layer

- we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end, where we need to reconstruct the output vector from shards.

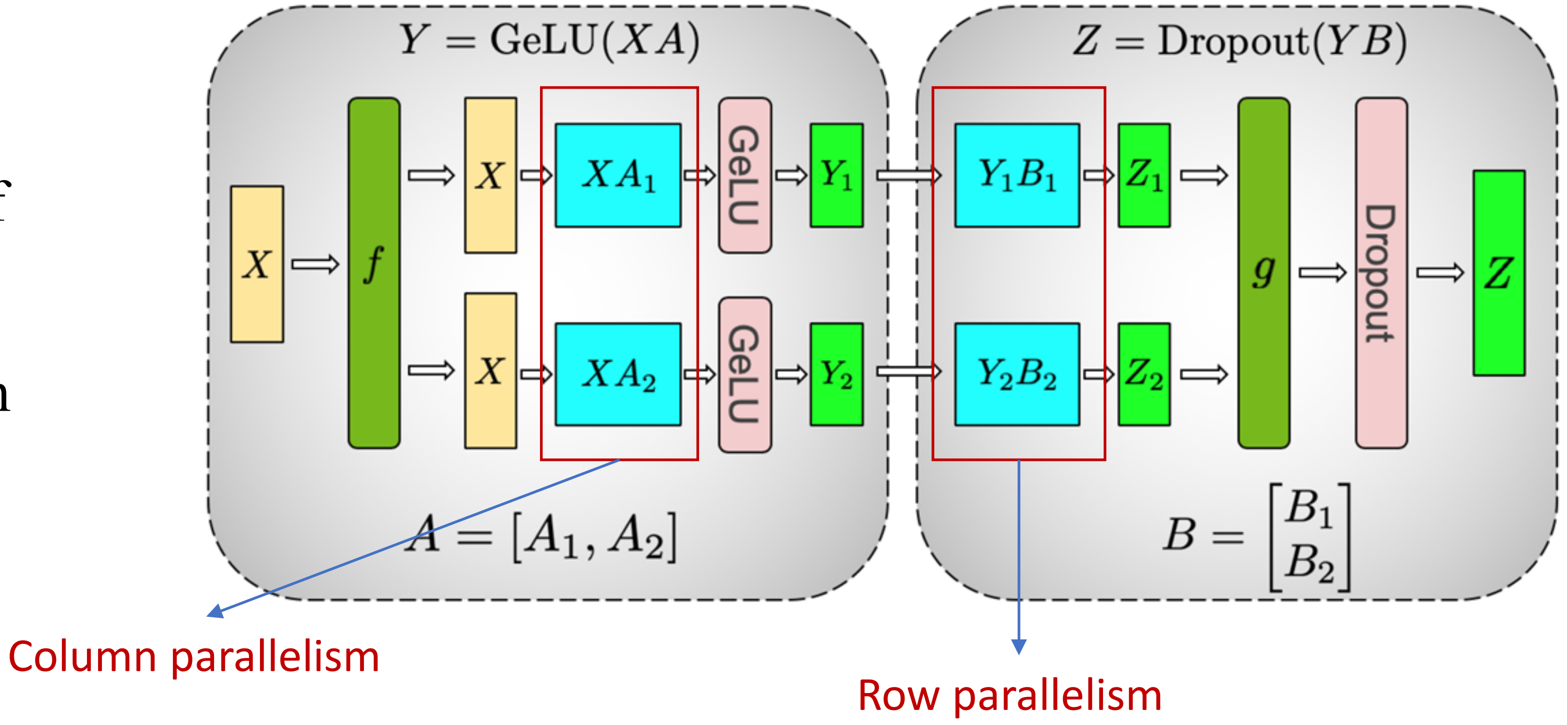


Column parallelism

f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

TP for MLP layer

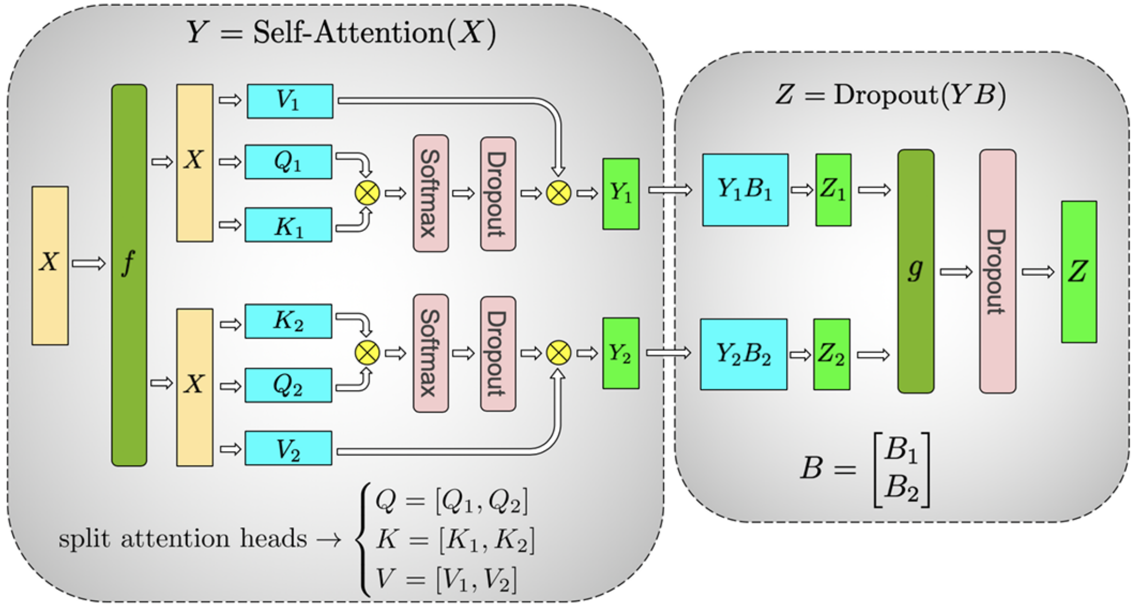
- we can update an MLP of arbitrary depth, without the need for any synchronization between GPUs until the very end, where we need to reconstruct the output vector from shards.



f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

TP for self-attn

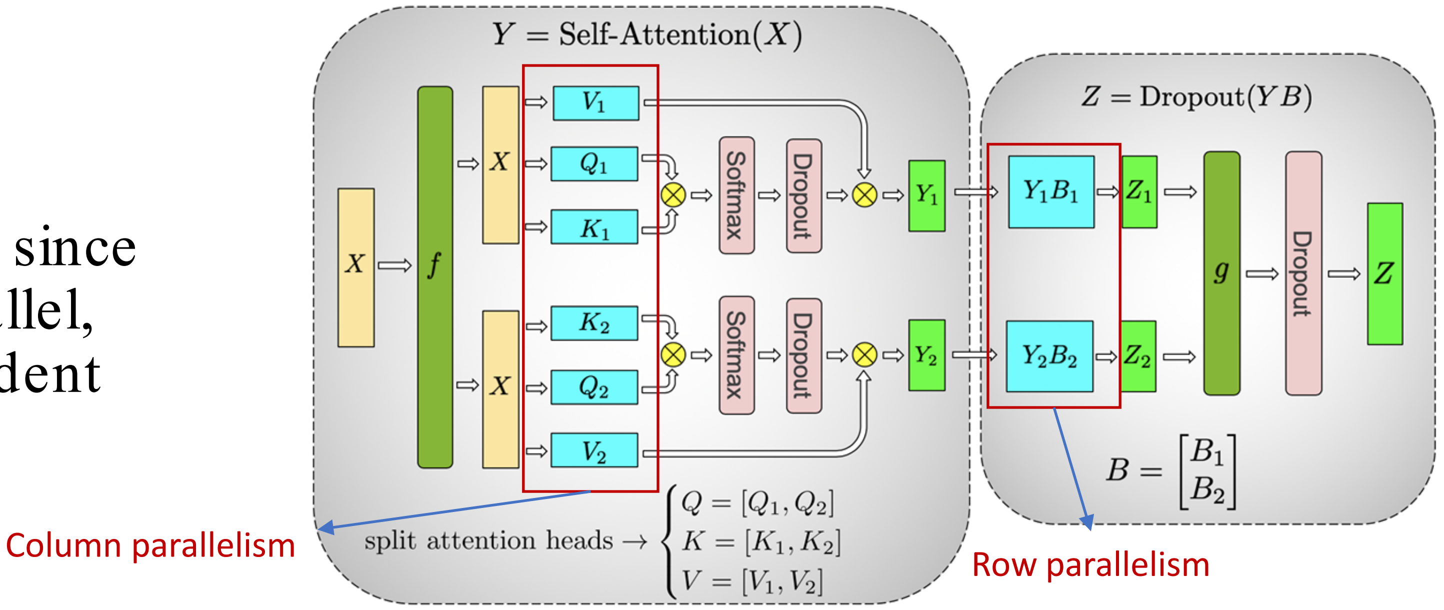
- Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads



f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

TP for self-attn

- Parallelizing the multi-headed attention layers is even simpler, since they are already inherently parallel, due to having multiple independent heads

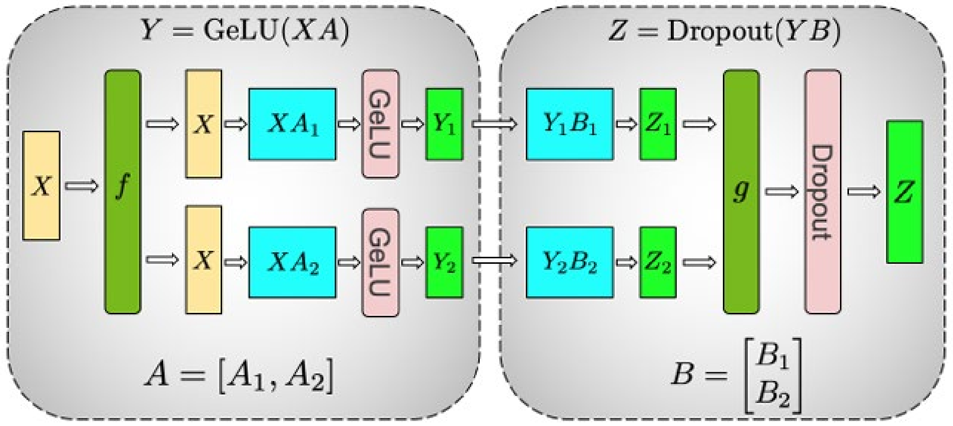


$Y = \text{Self-Attention}(X)$
$Z = \text{Dropout}(YB)$
X
f
V_1
Q_1
K_1
K_2
Q_2
V_2
Softmax
Dropout
Y_1
Y_2
Y_1B_1
Y_2B_2
Z_1
Z_2
g
Z
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
split attention heads $\rightarrow \begin{cases} Q = [Q_1, Q_2] \\ K = [K_1, K_2] \\ V = [V_1, V_2] \end{cases}$
Column parallelism
Row parallelism

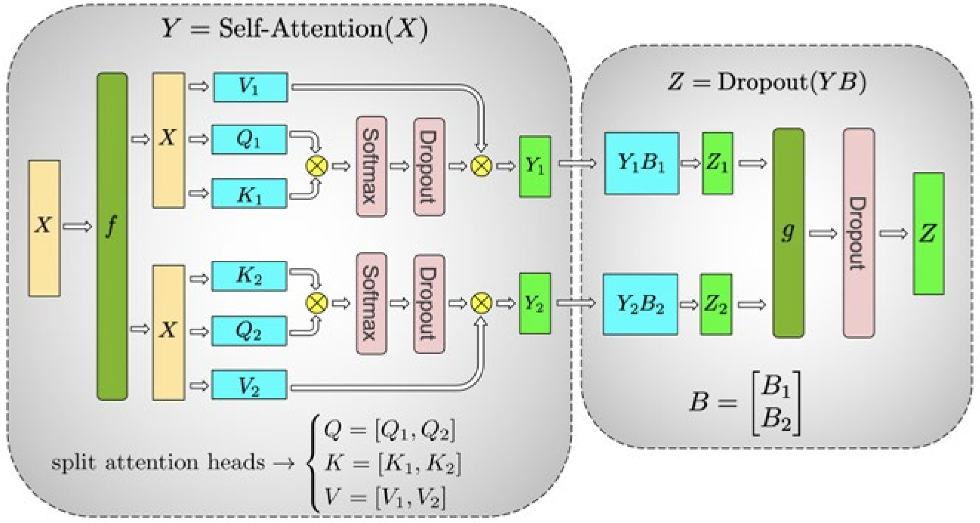
f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

Modern MP – MegatronLM (TP)

- Advantage:
  - a simple and efficient model parallel approach by making only a few targeted modifications to an existing PyTorch transformer implementation.
  - demonstrate up to 76% scaling efficiency using 512 GPUs.
  - Can scaling up the model size with limited hardware resources.
  - The scaled MegatronLM achieve state of the art results on test sets: perplexity on WikiText103 (10.8 ppl), accuracy on LAMBADA (66.5%), and accuracy on RACE(90.9%)
- Disadvantage:
  - TP requires very fast network, and therefore it's not advisable to do TP across more than one node.
  - The parallelism degree of TP is highly dependent on the number of devices. if a node has 4 GPUs, the highest TP degree is therefore 4. If you need a TP degree of 8, you need to use nodes that have at least 8 GPUs.



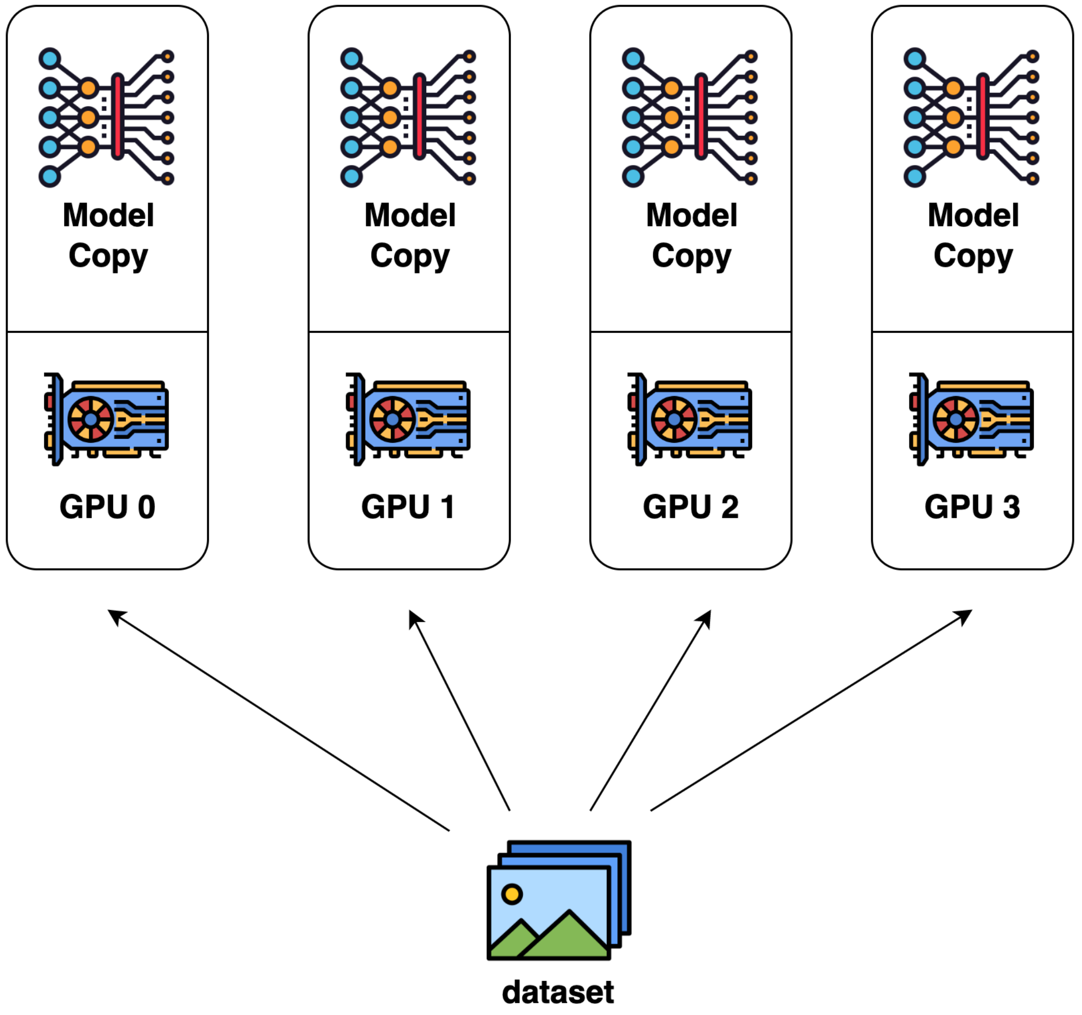
(a) MLP



(b) Self-Attention

Data parallelism (DP) – Naïve DP

- In data parallel training, the dataset is split into several shards, each shard is allocated to a device.
- Each device will hold a full copy of the model replica and trains on the dataset shard allocated.

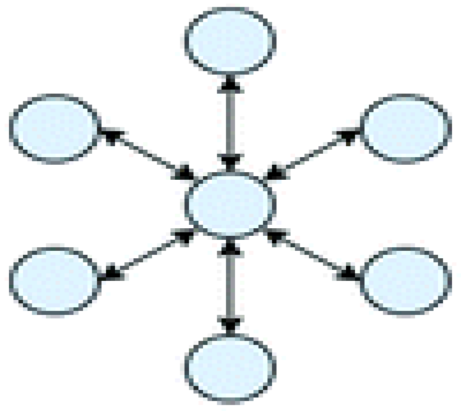


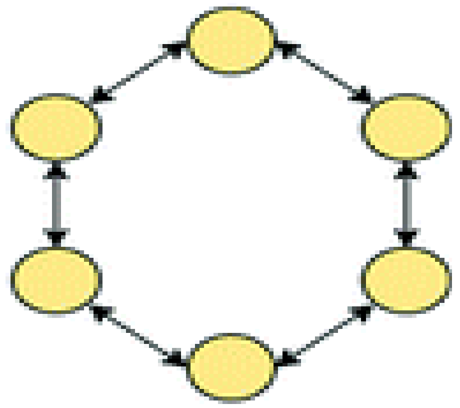
Naïve DP vs distributed data parallel (DDP)

- DataParallel is single-process, multi-thread, and only works on a single machine. (Threads-based)
- In DP, GPU0 needs to
  - reads the batch of data and then split them into mini-batch, and then send them to each GPU.
  - Receive output from each GPU, computes loss
  - scatters loss to each GPU, run backward
  - Receive gradients from each GPU, average those.
- GPU 0 in DP mode performs a lot more work than the rest of the GPUs, thus resulting in under-utilization of other GPUs and OOM in GPU 0.

Naïve DP vs DDP

- DistributedDataParallel is multi-process and works for both single- and multi- machine training. (multiprocess-based)
- In DDP, each GPU consumes each own mini-batch of data directly
- During backward, once the local gradients are ready, they are then averaged across all processes.
- DDP is more “balance” than DP, which is also faster and memory-friendly.





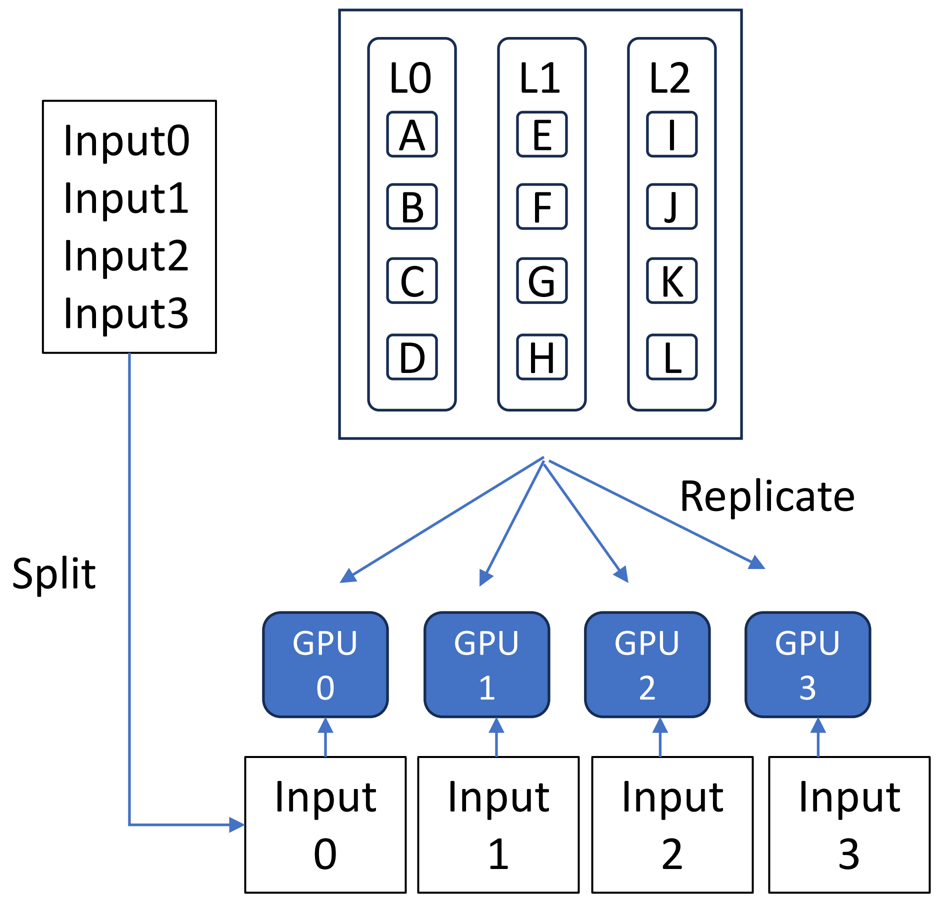
DP vs DDP

How can we overcome the limitations of existing solutions and train large models more efficiently?

- Basic data parallelism (DP) does not reduce memory per device and runs OOM for models with more than 1.4B parameters on GPUs with 32GB memory.
- Other existing solutions such as Pipeline Parallelism (PP), Model Parallelism (MP), make tradeoffs between functionality, usability, as well as memory and compute/communication efficiency.
- TP works well within a single node where the inter-GPU communication bandwidth is high, but the efficiency degrades quickly beyond a single node.

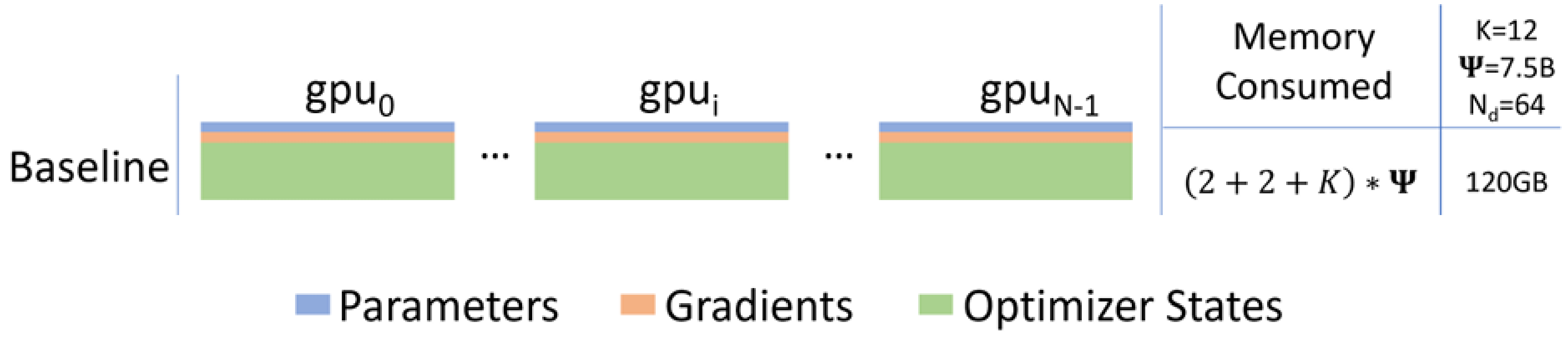
Memory consumption of normal DP

- Each GPU in DP mode holds different slice of data samples.
- DP requires each GPU to replicate the full model params, gradients and optimizer states. Then each GPU can process their own data samples with these replicated items.



Memory consumption - redundancy

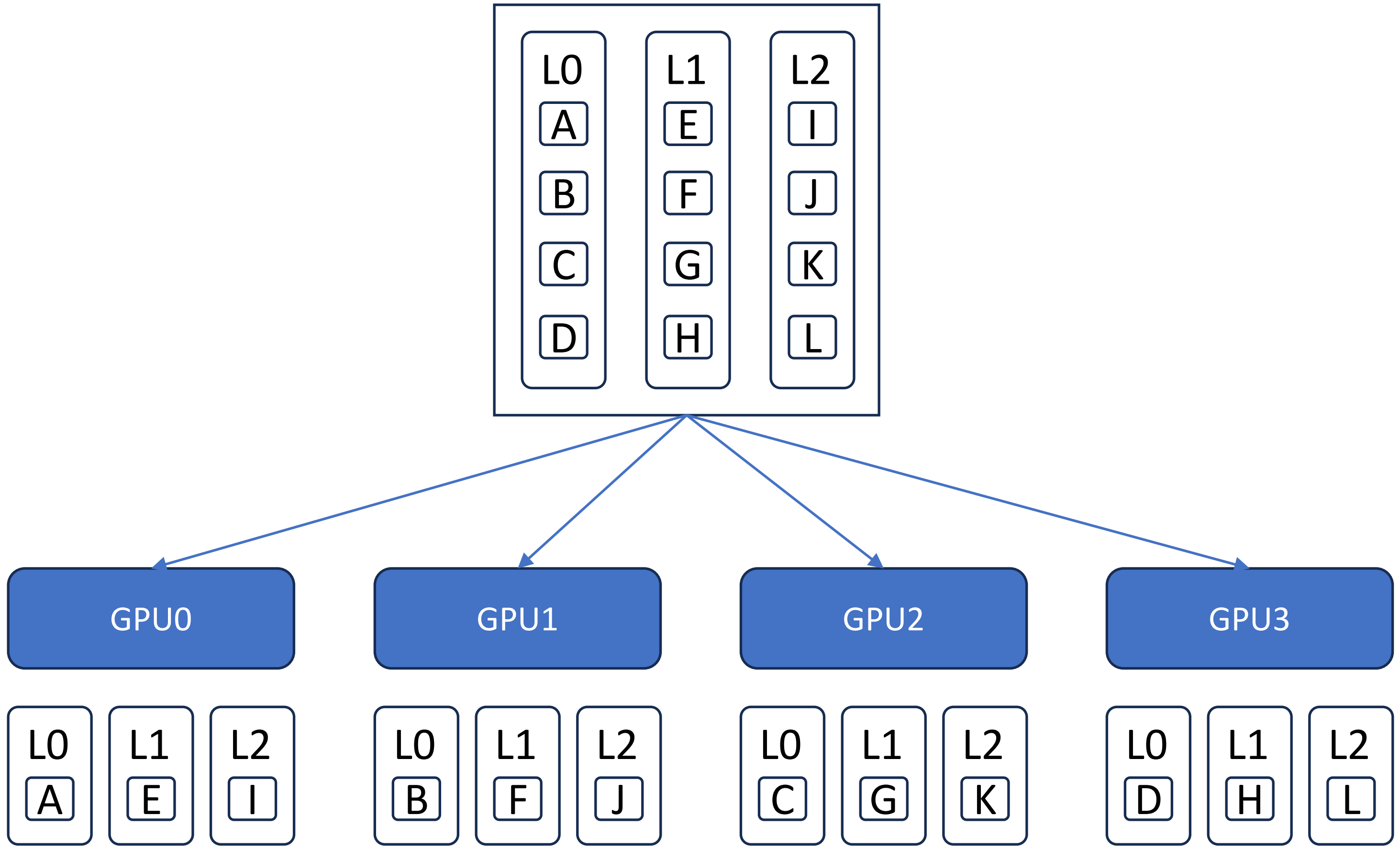
- Each GPU in DP mode holds different slice of data samples.
- DP requires each GPU to replicate the full model params, gradients and optimizer states. Then each GPU can process their own data samples with these replicated items.
- For large models, when training with DP, the majority of the memory is occupied by model states which include the optimizer states (such as momentum and variances in Adam), gradients, and parameters.



Basic data parallelism (DP) does not reduce memory per device. A 64-way DP training of a 7.5B LLM consumes (2 + 2 + 12) * 7.5 = 120GB memory

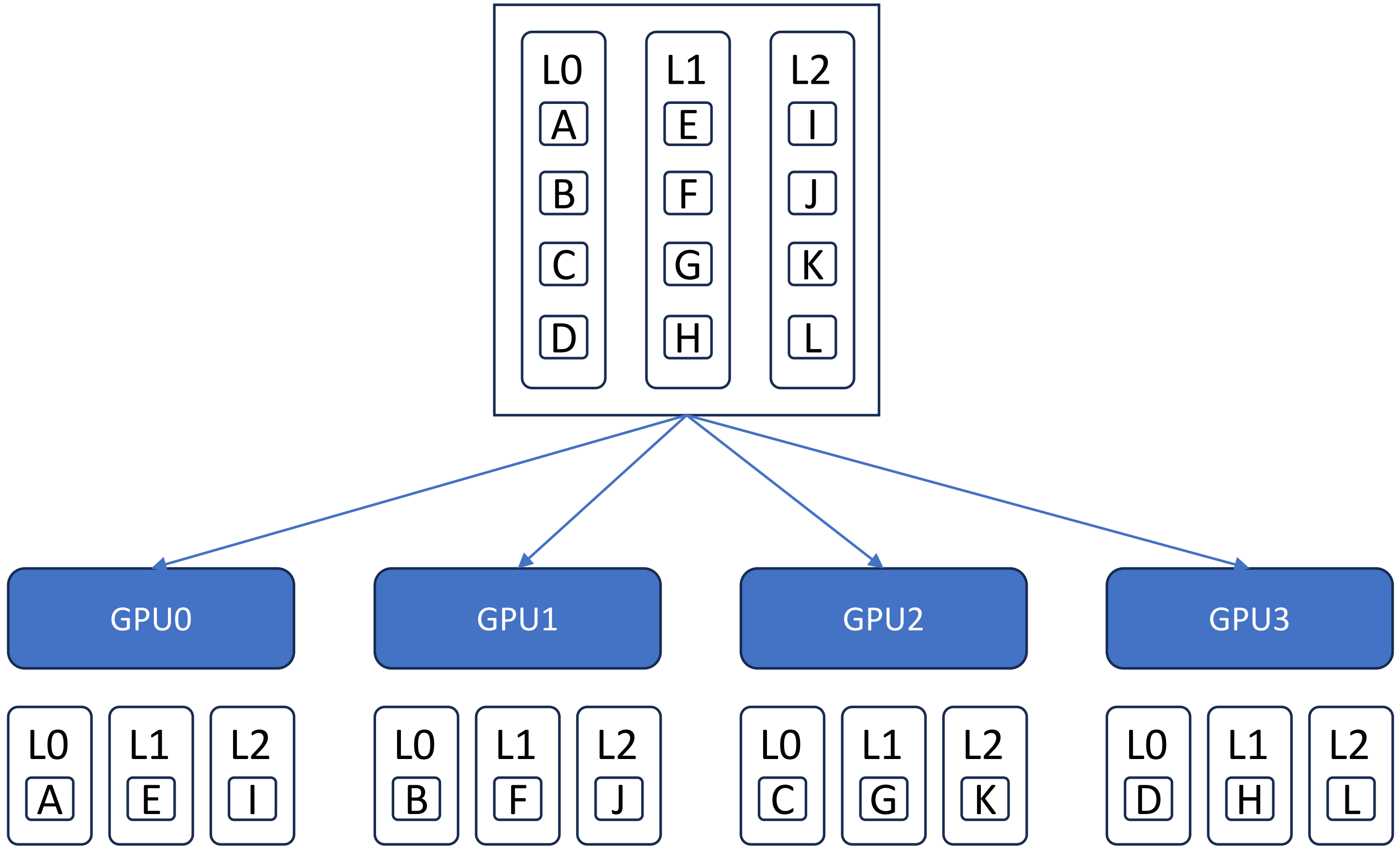
Solution: DeepSpeed ZeRO

- Instead of replicating the full model params, gradients and optimizer states, in DeepSpeed ZeRO, each GPU stores only a slice of these items.
- At run-time when the full layer params are needed just for the given layer, all GPUs synchronize to give each other parts that they miss.



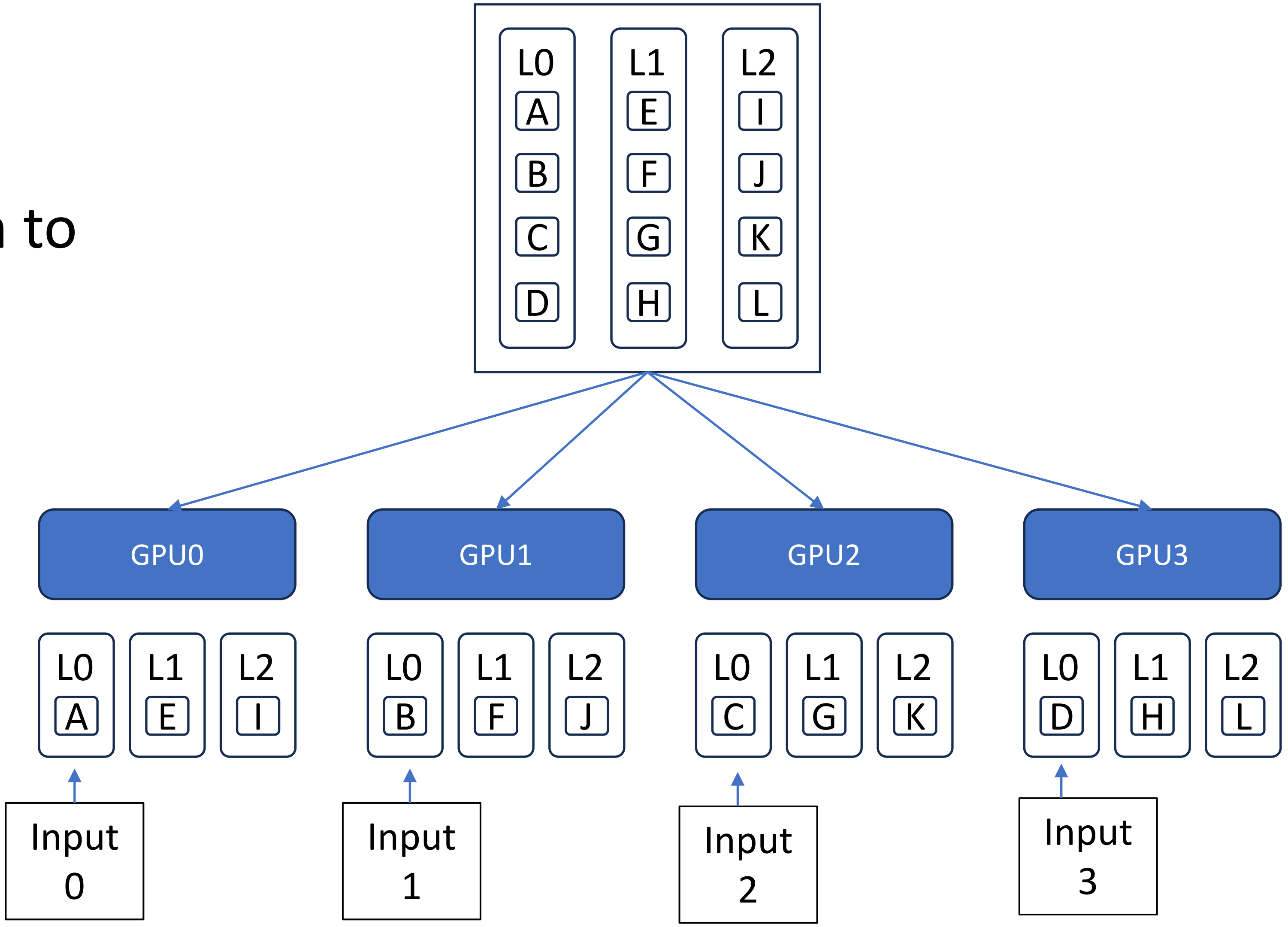
Solution: DeepSpeed ZeRO

- Considering a LM with only 3 layers, and each layer contains only 4 parameters. Given a node with 4 GPUs, take the partition of model parameter as an example, DeepSpeed ZeRO first assign the model parameters to these 4 GPUs.



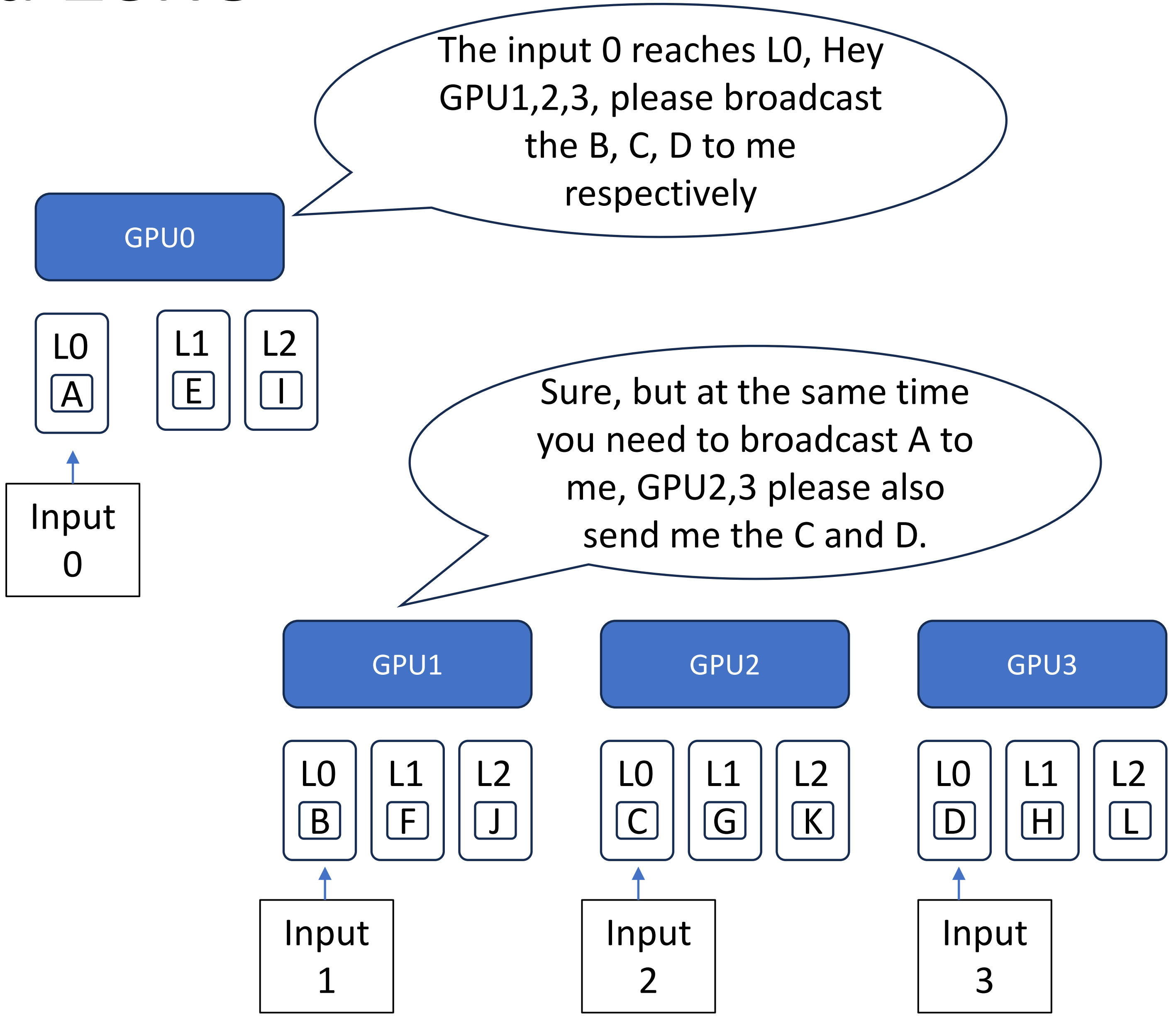
Solution: DeepSpeed ZeRO

- Same as the original DP, DeepSpeed split the input into 4 slices and feed them to 4 different GPUs.



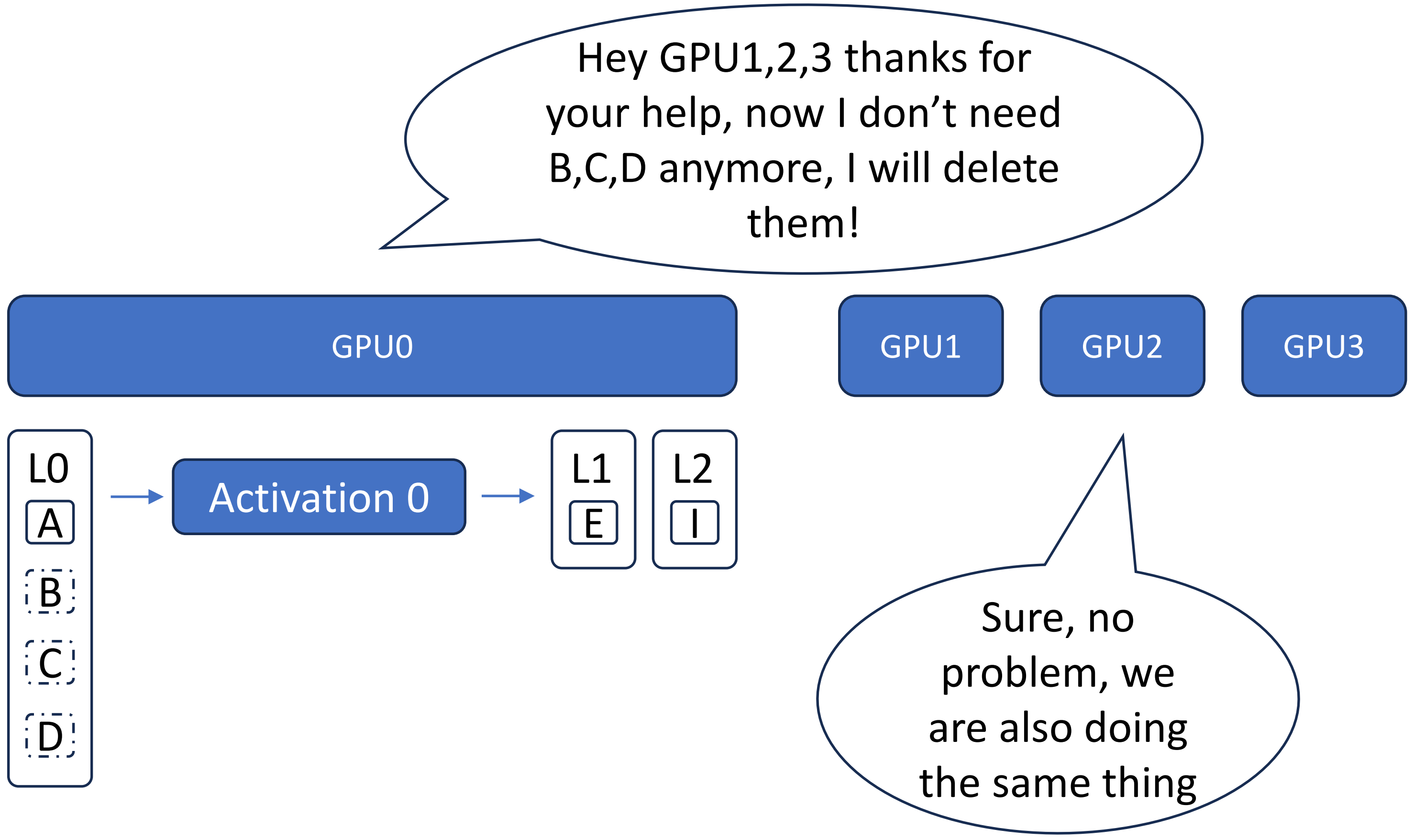
Solution: DeepSpeed ZeRO

- However, different from the original DP, each GPU in DeepSpeed ZeRO only holds a small fraction of the model parameters instead of the replicated entire model.
- To finish the forward pass, each GPU need to ‘borrow’ parameters from other GPUs


The input 0 reaches L0, Hey GPU1,2,3, please broadcast the B, C, D to me respectively
GPU0
L0 A
L1 E
L2 I
Input 0
Sure, but at the same time you need to broadcast A to me, GPU2,3 please also send me the C and D.
GPU1
L0 B
L1 F
L2 J
Input 1
GPU2
L0 C
L1 G
L2 K
Input 2
GPU3
L0 D
L1 H
L2 L
Input 3

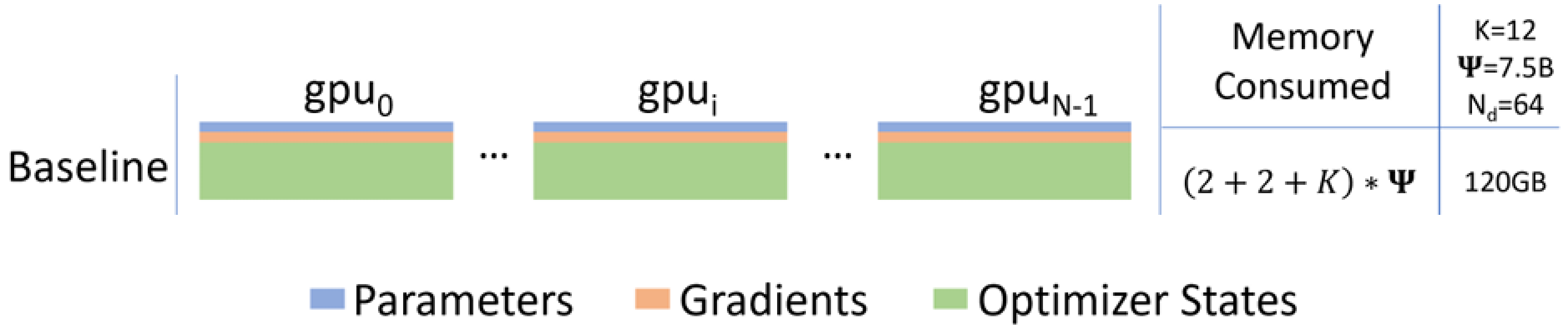
Solution: DeepSpeed ZeRO

- After each GPU makes the use of the borrowed parameters, it will delete those parameters to save the memory.


Hey GPU1,2,3 thanks for your help, now I don't need B,C,D anymore, I will delete them!
GPU0
GPU1
GPU2
GPU3
L0
A
B
C
D
Activation 0
L1
E
L2
I
Sure, no problem, we are also doing the same thing

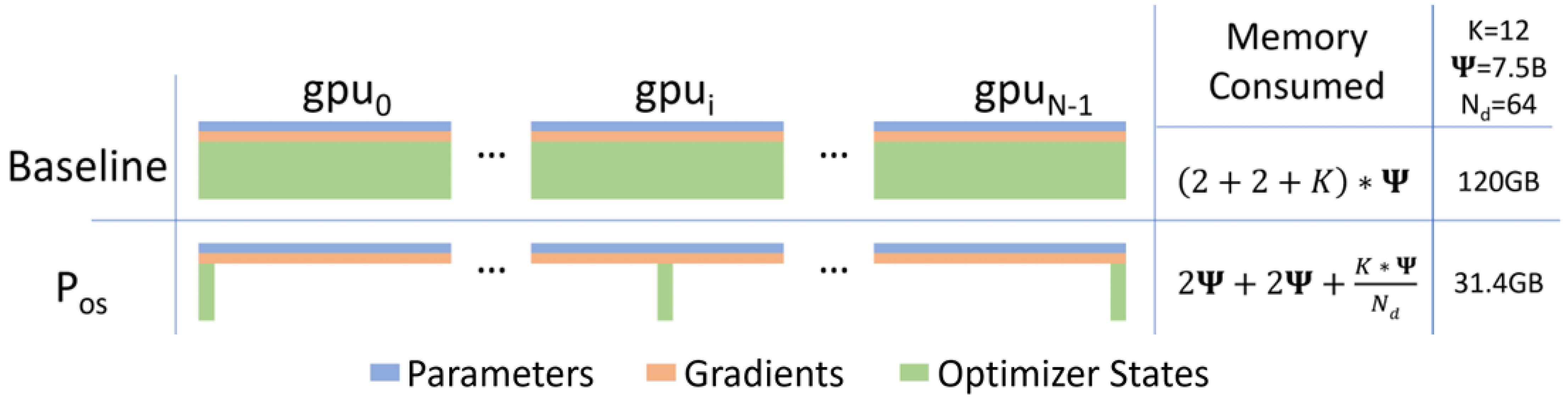
DeepSpeed ZeRO

- As soon as the calculation is done, the items (model parameters, optimizer states, gradient) that is no longer needed gets dropped.



DeepSpeed ZeRO

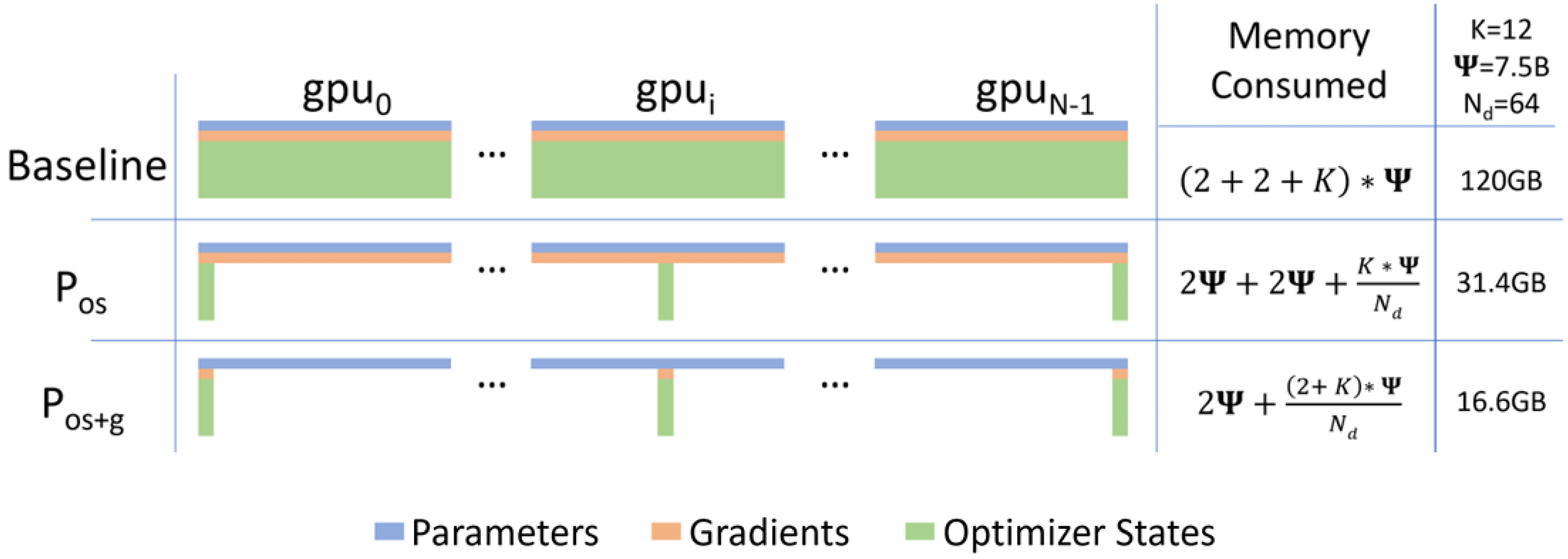
- 1) Optimizer State Partitioning (P_{os}): 4x memory reduction, same communication volume as DP;



	Memory Consumed	K=12 $\boldsymbol{\Psi}$=7.5B N_d=64
Baseline	$(2+2+K)*\boldsymbol{\Psi}$	120GB
P_{os}	$2\boldsymbol{\Psi}+2\boldsymbol{\Psi}+\frac{K*\boldsymbol{\Psi}}{N_d}$	31.4GB

DeepSpeed ZeRO

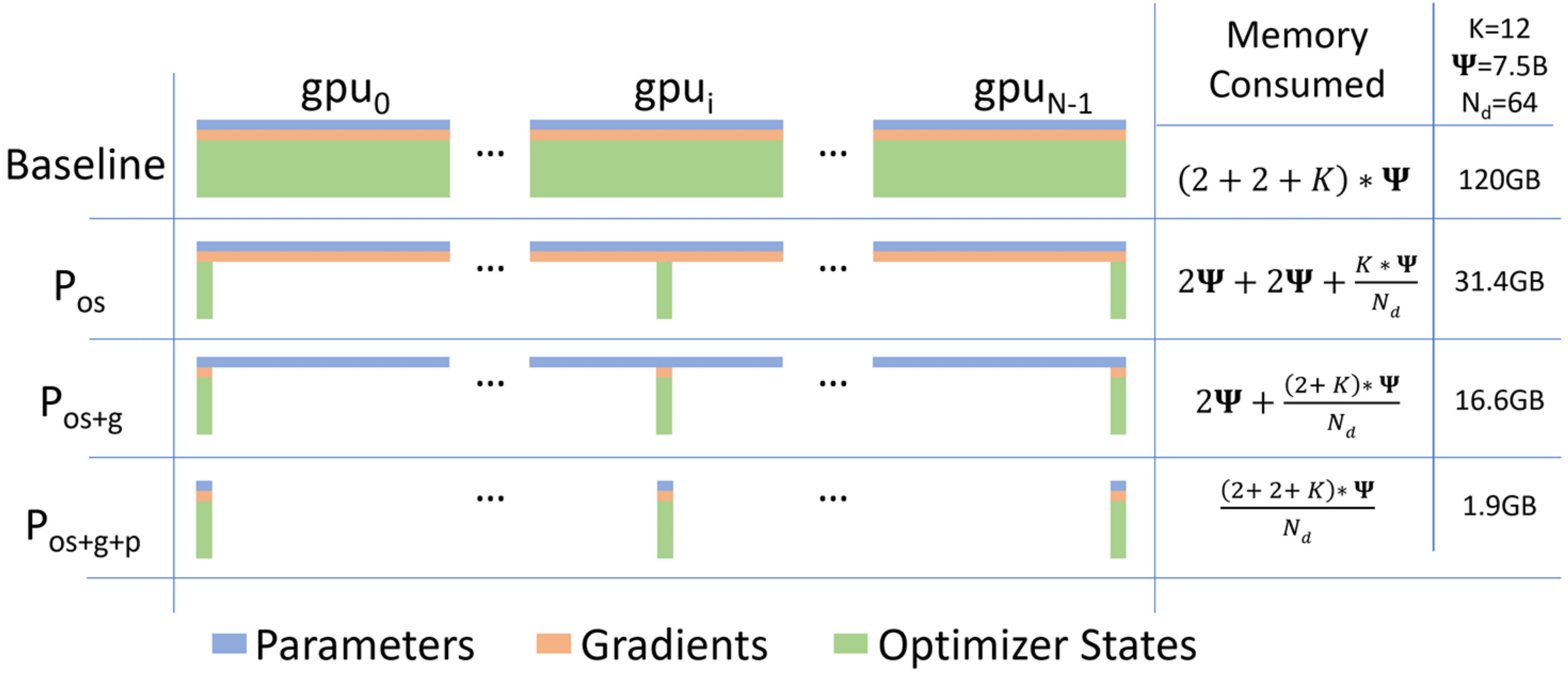
- 2) Add Gradient Partitioning (P_{os+g}): 8x memory reduction, same communication volume as DP;



	Memory Consumed	K=12 Ψ=7.5B N_d=64
Baseline	$(2+2+K)*\Psi$	120GB
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB

DeepSpeed ZeRO

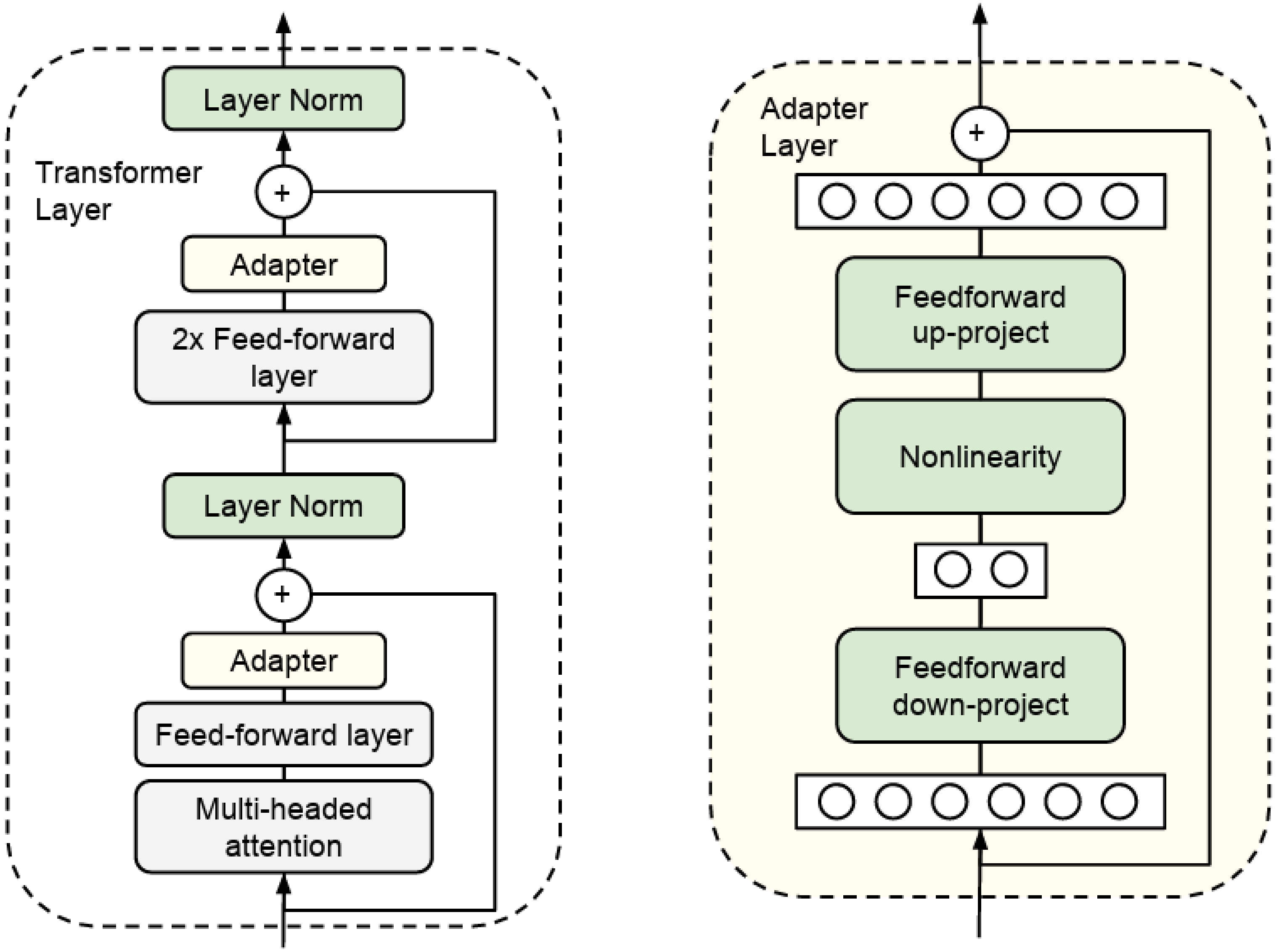
- 3) Add Parameter Partitioning (P_{os+g+p}): Memory reduction is linear with DP degree Nd. For example, splitting across 64 GPUs ($N_d = 64$) will yield a 64x memory reduction. There is a modest 50% increase in communication volume.



	Memory Consumed	K=12 Ψ=7.5B N_d=64
Baseline	$(2+2+K)*\Psi$	120GB
P_{os}	$2\Psi + 2\Psi + \frac{K*\Psi}{N_d}$	31.4GB
P_{os+g}	$2\Psi + \frac{(2+K)*\Psi}{N_d}$	16.6GB
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB

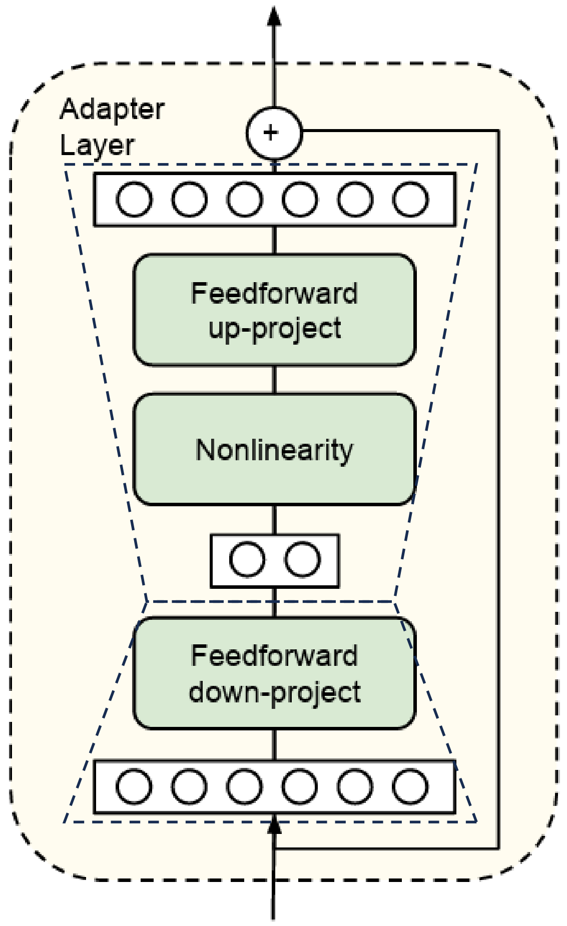
Parameter-efficient fine-tuning (PEFT) - Adapter

- Adapter-based methods add extra trainable parameters after the attention and fully-connected layers of a frozen pretrained model to reduce memory-usage and speed up training.
- The adapters are typically small but demonstrate comparable performance to a fully finetuned model and enable training larger models with fewer resources.



Adapter

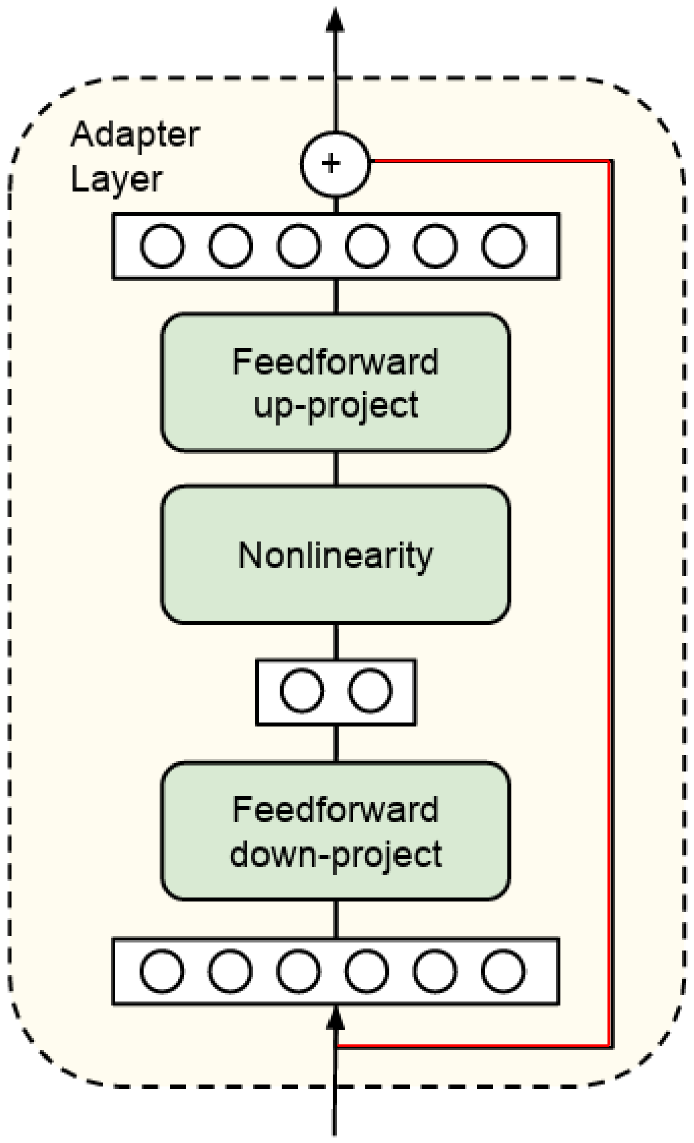
- The adapter consists of a bottleneck which contains few parameters relative to the attention and feedforward layers in the original model.



Bottleneck: The adapters first project the original d-dimensional features into a smaller dimension, m, apply a nonlinearity, then project back to d dimensions. The total number of parameters added per layer, including biases, $is\ 2md\ + d + m$. By setting $m \ll d$, the number of parameters added is limited.

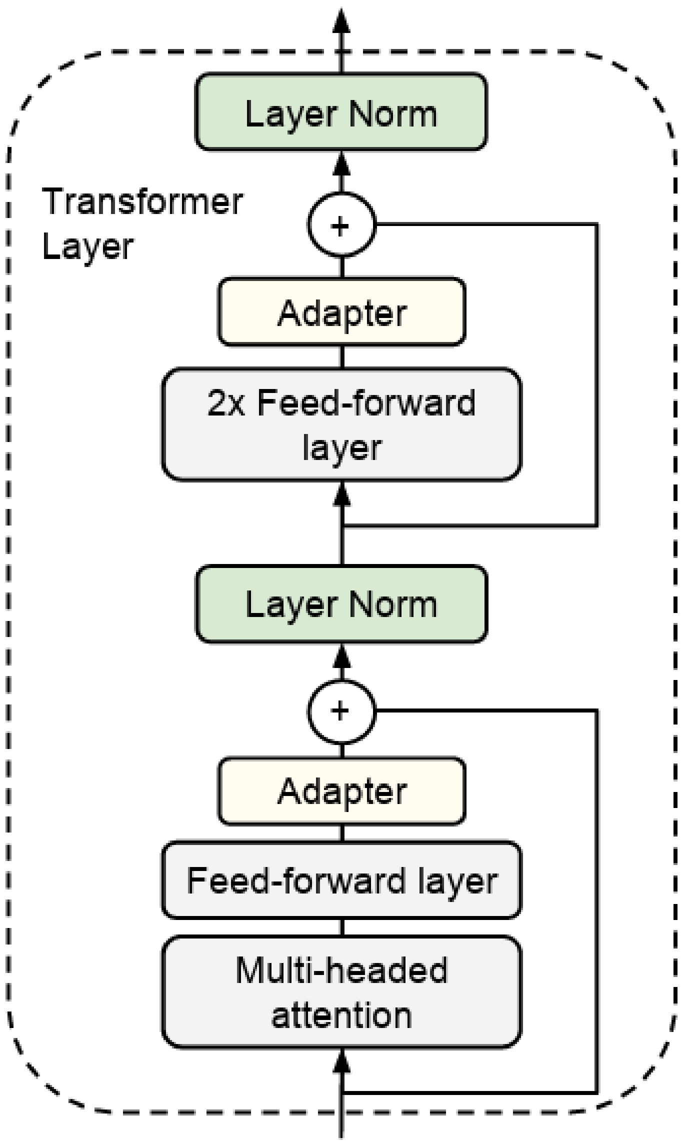
Adapter

- The adapter also contains a skip-connection.



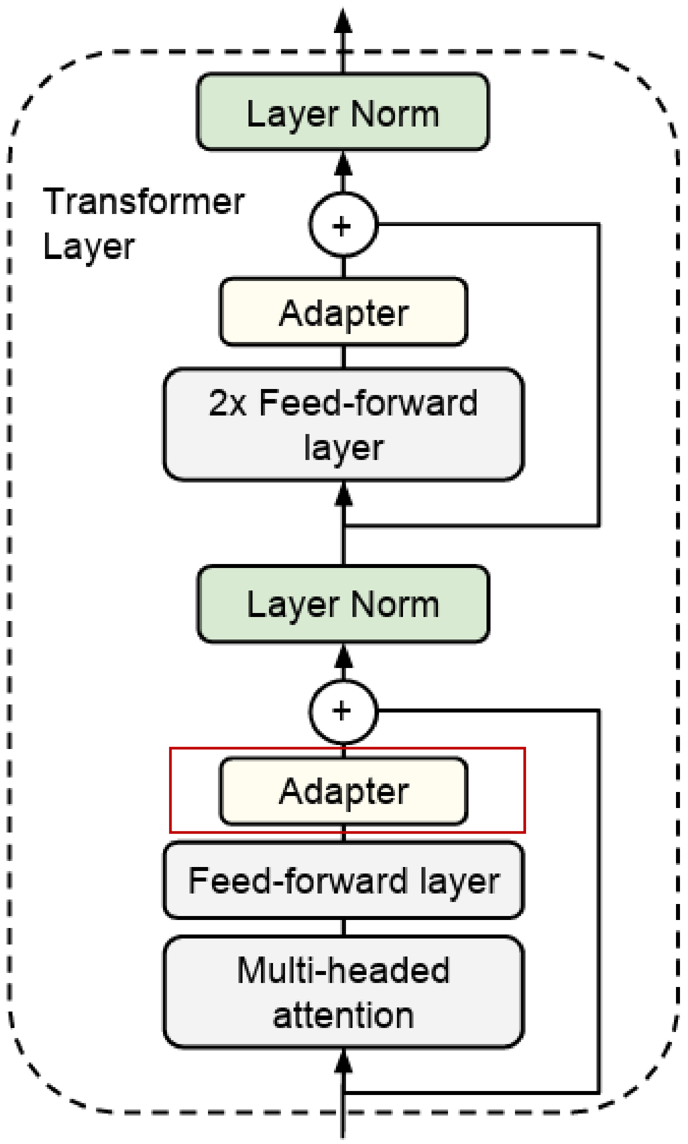
Adapter

- The adapter module are added twice to each Transformer layer



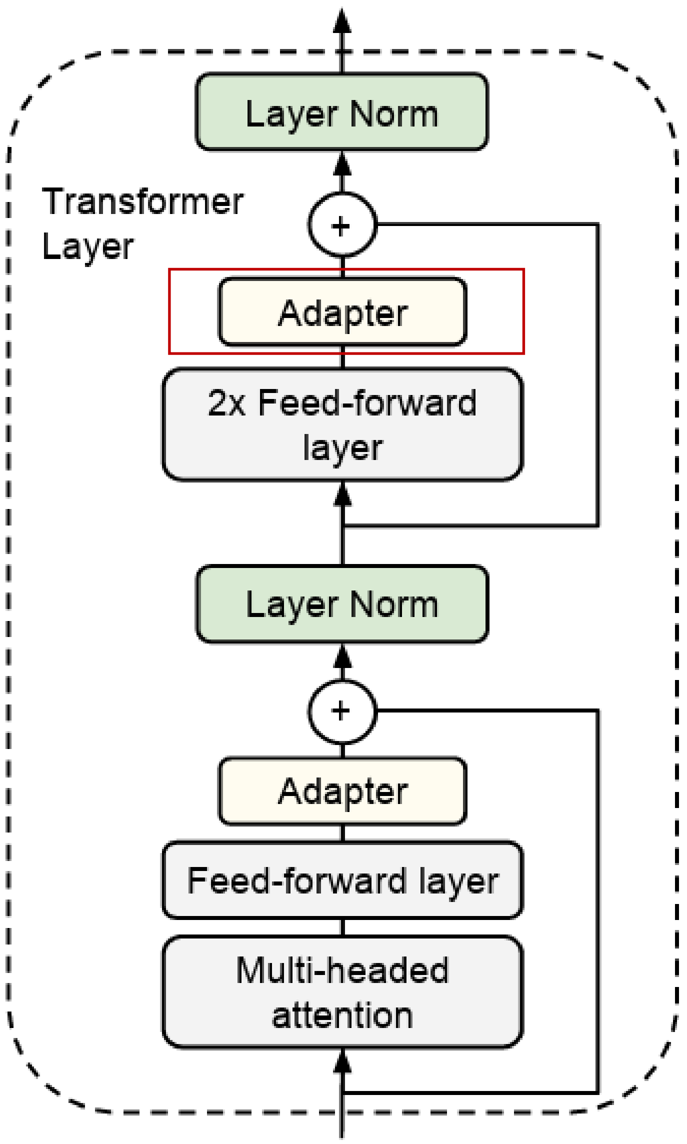
Adapter

- The adapter module are added twice to each Transformer layer: after the projection following multi-headed attention



Adapter

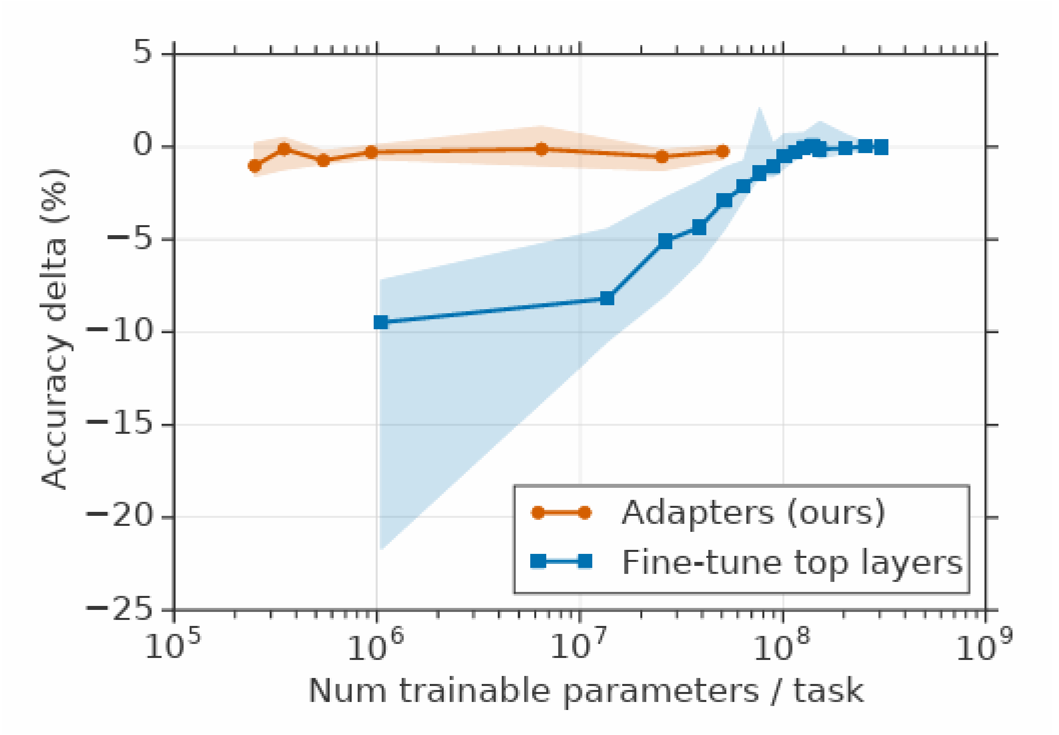
- The adapter module are added twice to each Transformer layer: after the two feed-forward layers



Adapter

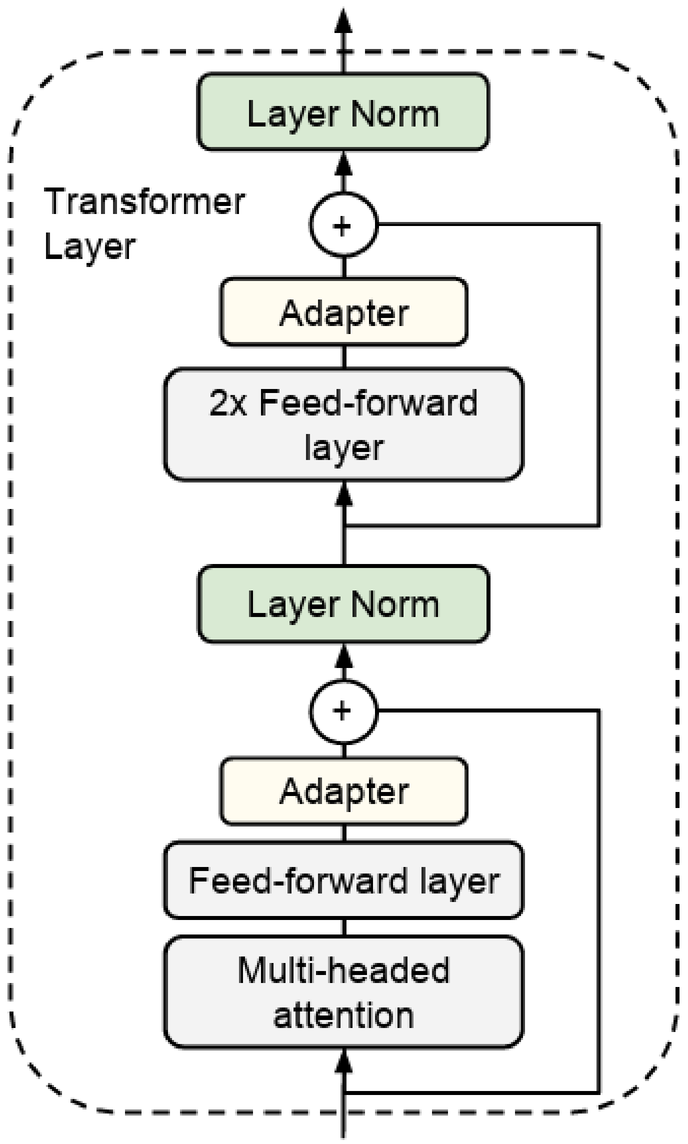
- On GLUE, performance decreases dramatically when fewer layers are fine-tuned.

- Adapters yield good performance across a range of sizes two orders of magnitude fewer than fine-tuning.



Adapter

- Disadvantage:
  - large neural networks rely on hardware parallelism to keep the latency low, and adapter layers have to be processed sequentially.



Adapter

- In a generic scenario without model parallelism, such as running inference on GPT-2 medium on a single GPU, we see a noticeable increase in latency when using adapters, even with a very small bottleneck dimension.
- This problem gets worse when we need to shard the model, because the additional depth requires more synchronous GPU operations such as AllReduce and Broadcast

Batch Size	32	16	1
Sequence Length	512	256	128
$\|\Theta\|$	0.5M	11M	11M
Fine-Tune	1449.4±0.8	338.0±0.6	19.8±2.7
AdapterL	1482.0±1.0 (+2.2%)	354.8±0.5 (+5.0%)	23.9±2.1 (+20.7%)
AdapterH	1492.2±1.0 (+3.0%)	366.3±0.5 (+8.4%)	25.8±2.2 (+30.3%)

Inference latency of a single forward pass in GPT-2 medium measured in milliseconds, "$|\Theta|$" denotes the number of trainable parameters in adapter layers. AdapterL and AdapterH are two variants of adapter tuning

LoRA

- Consider the weight matrix, **W0**, which measures **d** by **d** in size and is kept unchanged during the training procedure. The updated weights, ΔW, also measure **d** by **d**.

- Directly update weight matrix by training the **d** by **d** weights consumes memory and time.

- Training with a sequential adapter add latency during inference time.

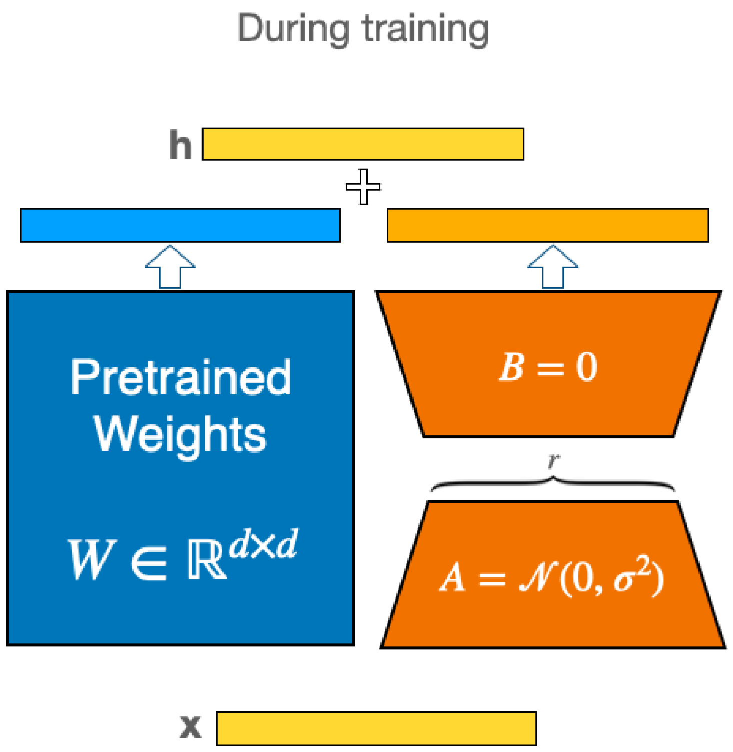
LoRA

- Consider the weight matrix, **W0**, which measures **d** by **d** in size and is kept unchanged during the training procedure. The updated weights, ΔW, also measure **d** by **d**.

- Directly update weight matrix by training the **d** by **d** weights consumes memory and time.

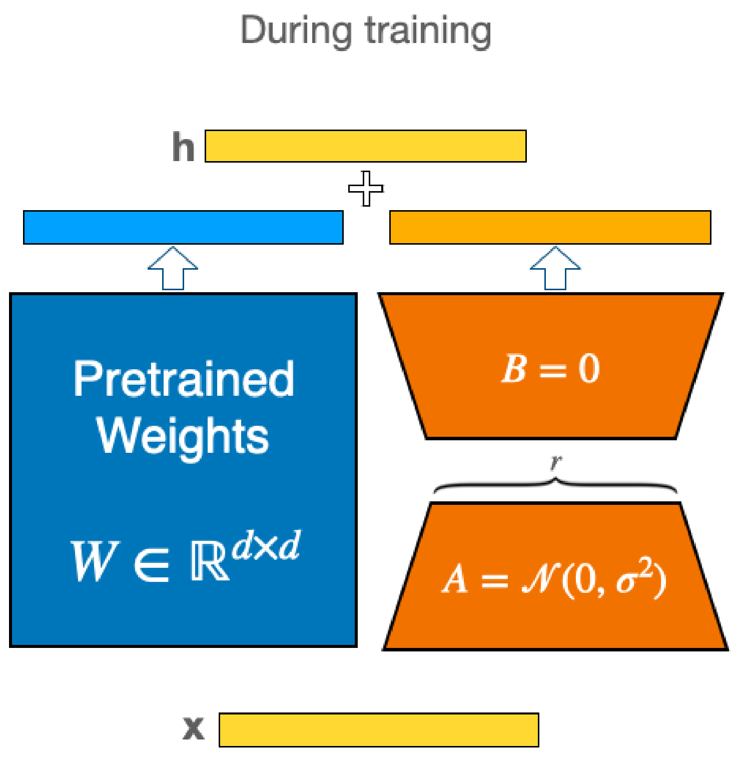
- Training with a sequential adapter add latency during inference time.

- Similar to the adapter, in LoRA, a parameter **r** is introduced which reduces the size of the matrix. The smaller matrices, A and B, are defined with a reduced size of **r** by **d**, and **d** by **r**. (bottleneck)


During training
h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
x

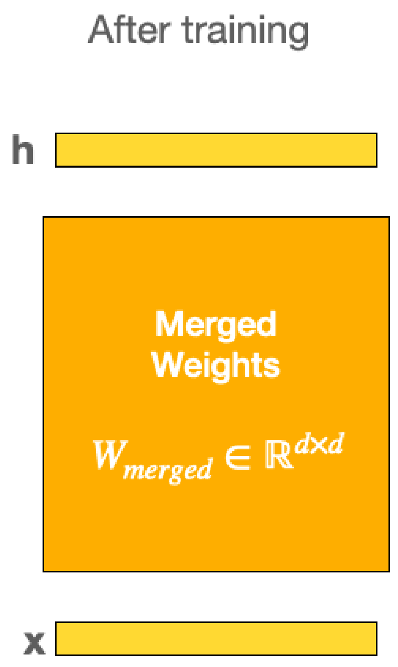
LoRA

- Different from the adapter, instead of sequentially added to the transformer layers, LoRA is added parallel to the pretrained weights.
- After training, LoRA weights can be merged into the model weights, so there is no extra inference latency.

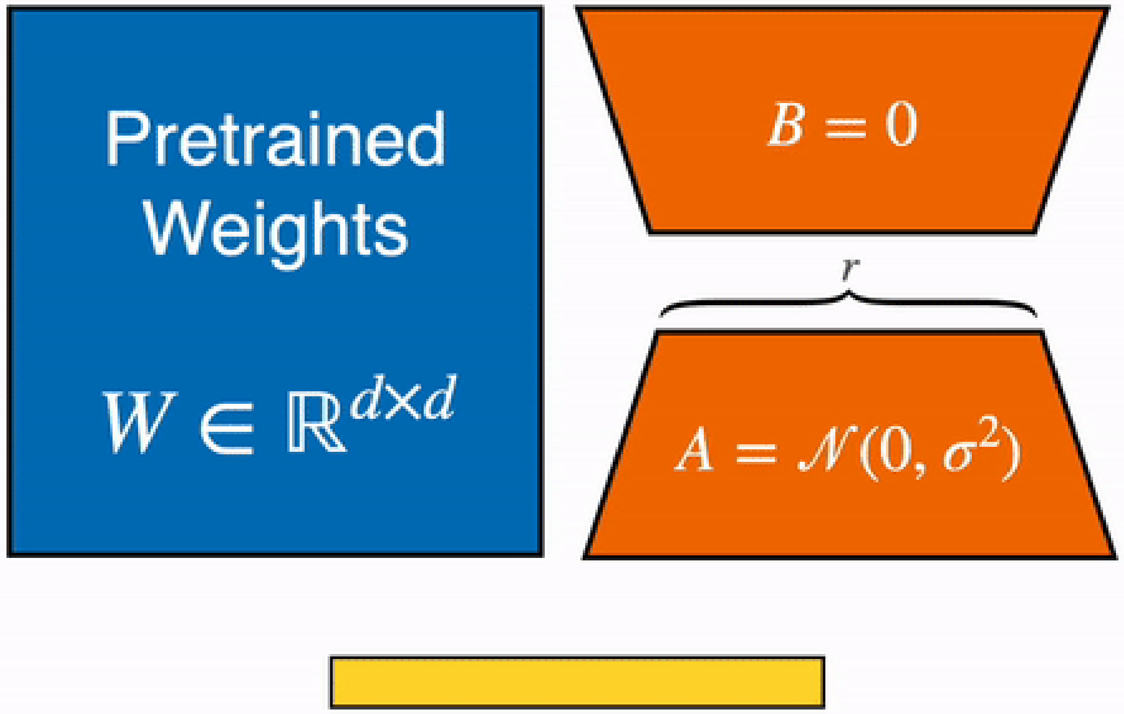

During training
h
Pretrained Weights
$W \in \mathbb{R}^{d\times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
x

$$h = Wx + BAx$$

$$h = \underbrace{(W + BA)}_{W_{merged}}x$$



LoRA


Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$

LoRA – efficient and also effective

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoBbase (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoBbase (AdptD)*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoBbase (AdptD)*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoBbase (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
RoBlarge (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
RoBlarge (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**
RoBlarge (AdptP)†	3.0M	$90.2_{\pm.3}$	$96.1_{\pm.3}$	$90.2_{\pm.7}$	$\mathbf{68.3}_{\pm1.0}$	$\mathbf{94.8}_{\pm.2}$	$\mathbf{91.9}_{\pm.1}$	$83.8_{\pm2.9}$	$92.1_{\pm.7}$	88.4
RoBlarge (AdptP)†	0.8M	$\mathbf{90.5}_{\pm.3}$	$\mathbf{96.6}_{\pm.2}$	$89.7_{\pm1.2}$	$67.8_{\pm2.5}$	$\mathbf{94.8}_{\pm.3}$	$91.7_{\pm.2}$	$80.1_{\pm2.9}$	$91.9_{\pm.4}$	87.9
RoBlarge (AdptH)†	6.0M	$89.9_{\pm.5}$	$96.2_{\pm.3}$	$88.7_{\pm2.9}$	$66.5_{\pm4.4}$	$94.7_{\pm.2}$	$92.1_{\pm.1}$	$83.4_{\pm1.1}$	$91.0_{\pm1.7}$	87.8
RoBlarge (AdptH)†	0.8M	$90.3_{\pm.3}$	$96.3_{\pm.5}$	$87.7_{\pm1.7}$	$66.3_{\pm2.0}$	$94.7_{\pm.2}$	$91.5_{\pm.1}$	$72.9_{\pm2.9}$	$91.5_{\pm.5}$	86.4
RoBlarge (LoRA)†	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.2}_{\pm1.0}$	$68.2_{\pm1.9}$	$\mathbf{94.8}_{\pm.3}$	$91.6_{\pm.2}$	$\mathbf{85.2}_{\pm1.1}$	$\mathbf{92.3}_{\pm.5}$	**88.6**
DeBXXL (FT)*	1500.0M	91.8	**97.2**	92.0	72.0	**96.0**	92.7	93.9	92.9	91.1
DeBXXL (LoRA)	4.7M	$\mathbf{91.9}_{\pm.2}$	$96.9_{\pm.2}$	$\mathbf{92.6}_{\pm.6}$	$\mathbf{72.4}_{\pm1.1}$	$\mathbf{96.0}_{\pm.1}$	$\mathbf{92.9}_{\pm.1}$	$\mathbf{94.9}_{\pm.4}$	$\mathbf{93.0}_{\pm.2}$	**91.3**

References

- Rasley, Jeff, et al. "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters." *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2020.
- Shoeybi, Mohammad, et al. "Megatron-lm: Training multi-billion parameter language models using model parallelism." *arXiv preprint arXiv:1909.08053* (2019).
- Huang, Yanping, et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism." *Advances in neural information processing systems* 32 (2019).
- Hillis, W. Daniel, and Guy L. Steele Jr. "Data parallel algorithms." *Communications of the ACM* 29.12 (1986): 1170-1183.
- Narayanan, Deepak, et al. "PipeDream: generalized pipeline parallelism for DNN training." *Proceedings of the 27th ACM symposium on operating systems principles*. 2019.
- Rajbhandari, Samyam, et al. "Zero: Memory optimizations toward training trillion parameter models." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.
- Rajbhandari, Samyam, et al. "Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning." *Proceedings of the international conference for high performance computing, networking, storage and analysis*. 2021.
- Hu, Edward J., et al. "Lora: Low-rank adaptation of large language models." *arXiv preprint arXiv:2106.09685* (2021).
- Houlsby, Neil, et al. "Parameter-efficient transfer learning for NLP." *International conference on machine learning*. PMLR, 2019.