北京大学
PEKING UNIVERSITY

Distributed Optimization in LLMs

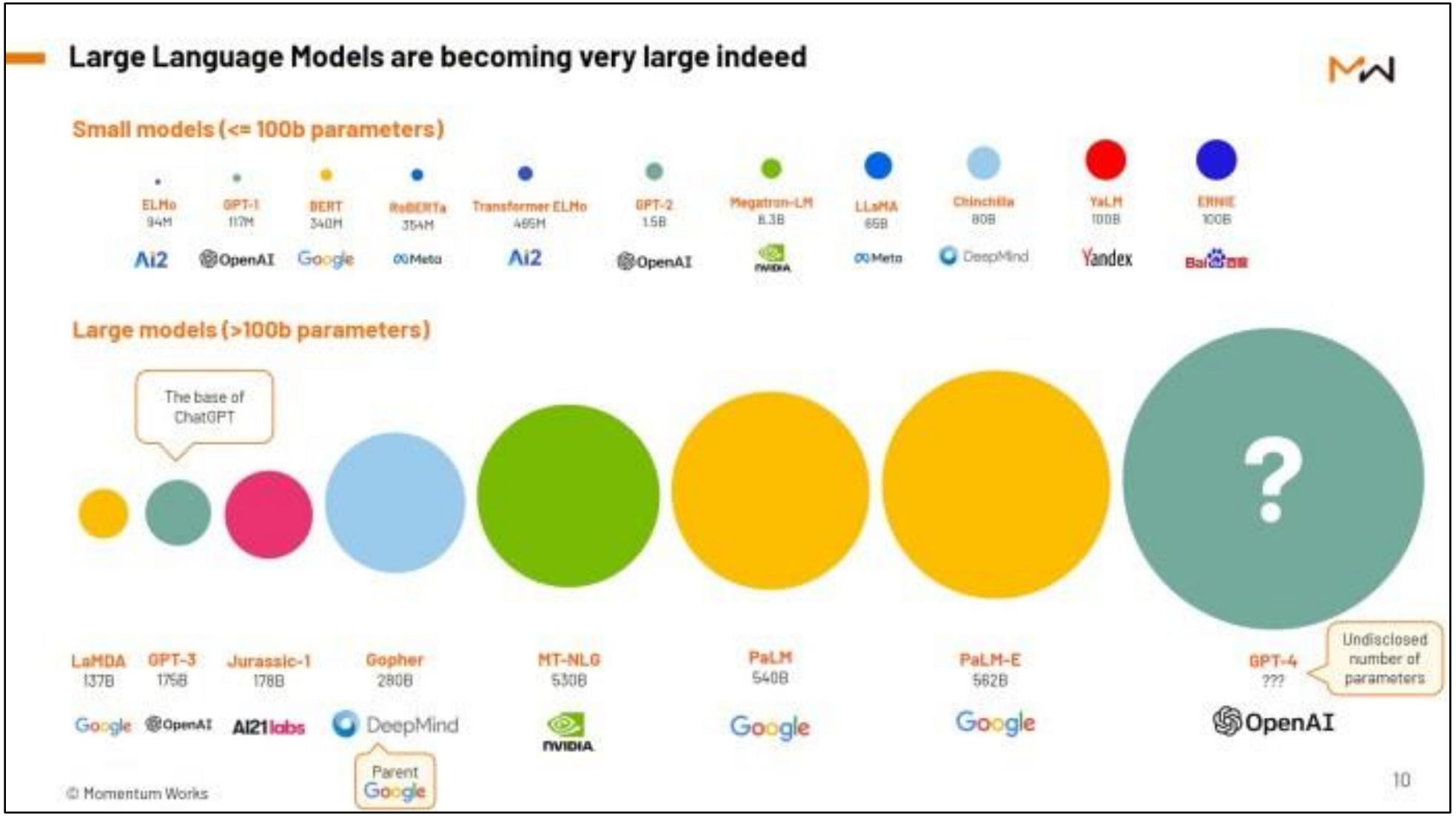
Kun Yuan

Center for Machine Learning Research @ Peking University

PART 01

LLM parameters and memories

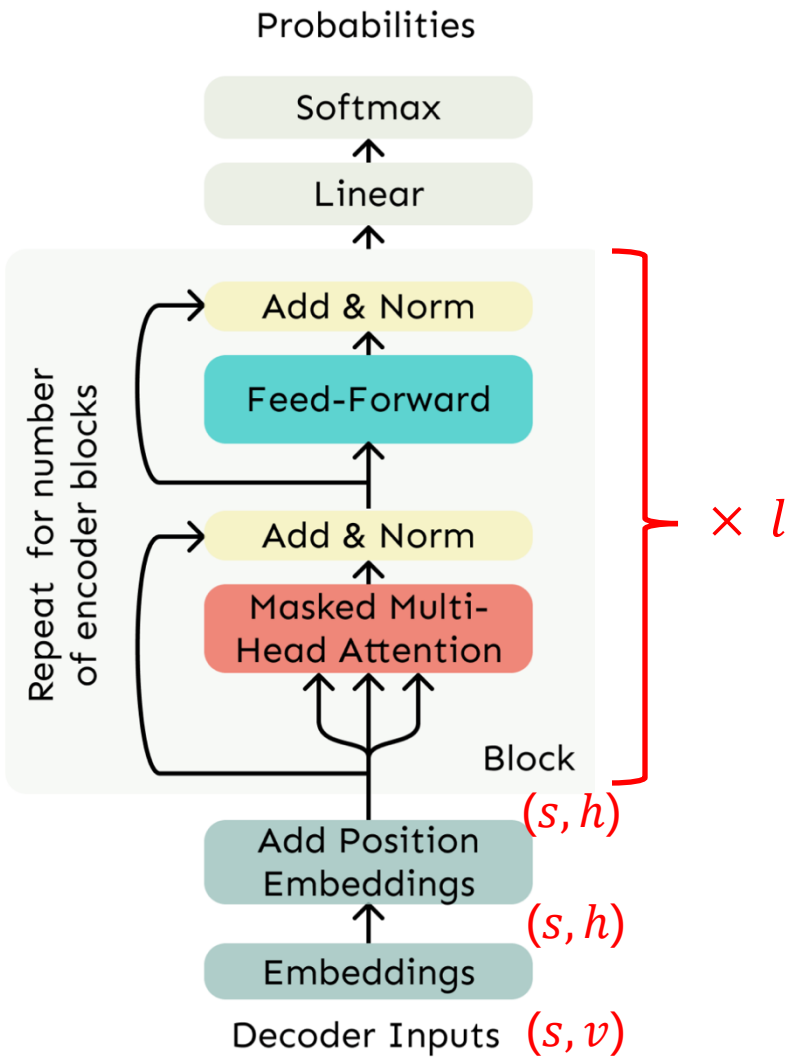
Large language models



[The emergence of Large Language Models (LLMs)]

GPTs are based on decoder-only transformers

- Number of the transformer layers: l
- Sequence length: s
- Vocabulary size: v
- Embedding representation dims: h

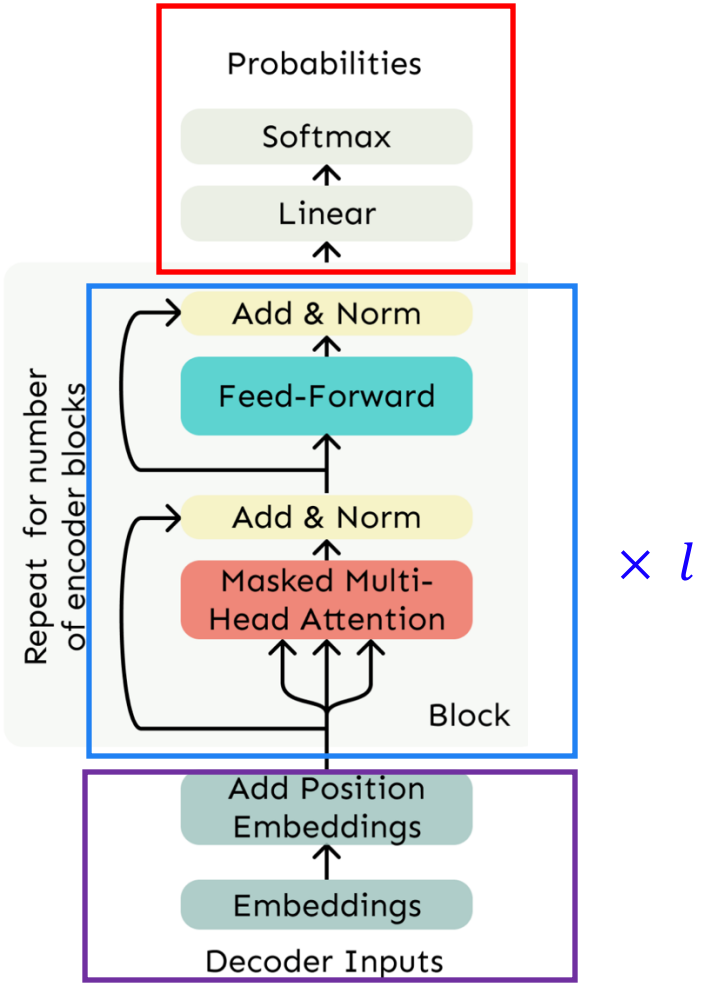


Total parameters

- Embeddings: vh
- Attention blocks: $12lh^2$
- Probability predictions: vh

Total parameters:

$$12\ell h^2 + 2vh$$



Example: LLaMA parameters

北京大学 PEKING UNIVERSITY

- Now we compare our theoretical evaluations with LLaMA model

- $12\ell h^2 + 2vh$ is a very accurate estimation

实际参数量	Embedding h	Attention层数l	Vocab大小v	预估参数量
6.7B	4096	32	32000	6,704,594,944
13.0B	5120	40	32000	12,910,592,000
32.5B	6656	60	32000	32,323,665,920
65.2B	8192	80	32000	64,948,797,440

Memory decomposition

北京大学 PEKING UNIVERSITY

Memory = Model + Gradient + Optimizer states + Activations

- Recall the Adam algorithm

$$
\begin{aligned}
g_k &= \nabla F(x_k; \xi_k) \\
m_k &= \beta_1 m_{k-1} + (1-\beta_1) g_k \\
s_k &= \beta_2 s_{k-1} + (1-\beta_2) g_k \odot g_k \\
x_{k+1} &= x_k - \frac{\gamma}{\sqrt{s_k} + \epsilon} \odot m_k
\end{aligned}
$$

- The optimizer states are twice the number of model parameters

Memory decomposition

Memory = Model + Gradient + Optimizer states + Activations

- Given a model with P parameters, gradient will consume P parameters, and Optimizer states will consume 2P parameters; **4P parameters in total.**
- When using FP32 to store parameters, each parameter takes **4** Bytes
- When using FP16 or BF16 to store parameters, each parameter takes **2** Bytes

Total memory

北京大学 PEKING UNIVERSITY

Memory = **Model + Gradient + Optimizer states** + **Activations**

$$(48\,l\,h^2 + bl(2s^2a + 14sh)) \times 4 \text{ Bytes}$$

- When hidden state **h** is large, the model parameters dominate the memory
- When batch-size **b** or sequence length **s** is large, the activation dominates the memory

Memory examples

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

GPT3 has 175B parameters; its model consumes $4 \times 175 \times 10^9$ Bytes = 700 GB

Its gradient takes 700 GB parameters; Optimizer states take 1.4 TB

GPT has sequence length s = 2048. When b=1, its activation takes 444 GB, 63% of the model

When b=128, its activation is 81 times of the model size

The massive parameters and memory costs in LLM calls for distributed optimizers

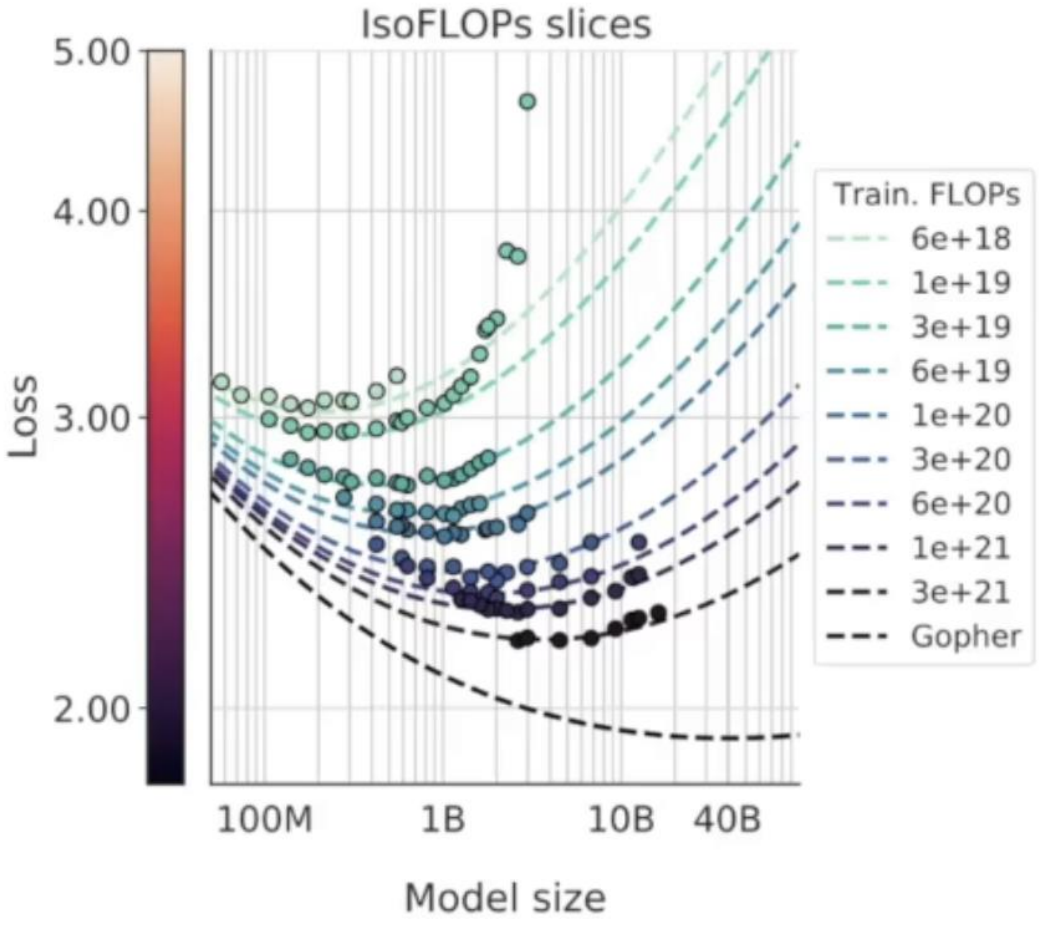
北京大学
PEKING UNIVERSITY

PART 02

LLM training systems and their scalability

Chinchilla law

[Training Compute-Optimal Large Language Models, 2022]



The “RGB” elements in LLM:

Larger dataset

+

Bigger model

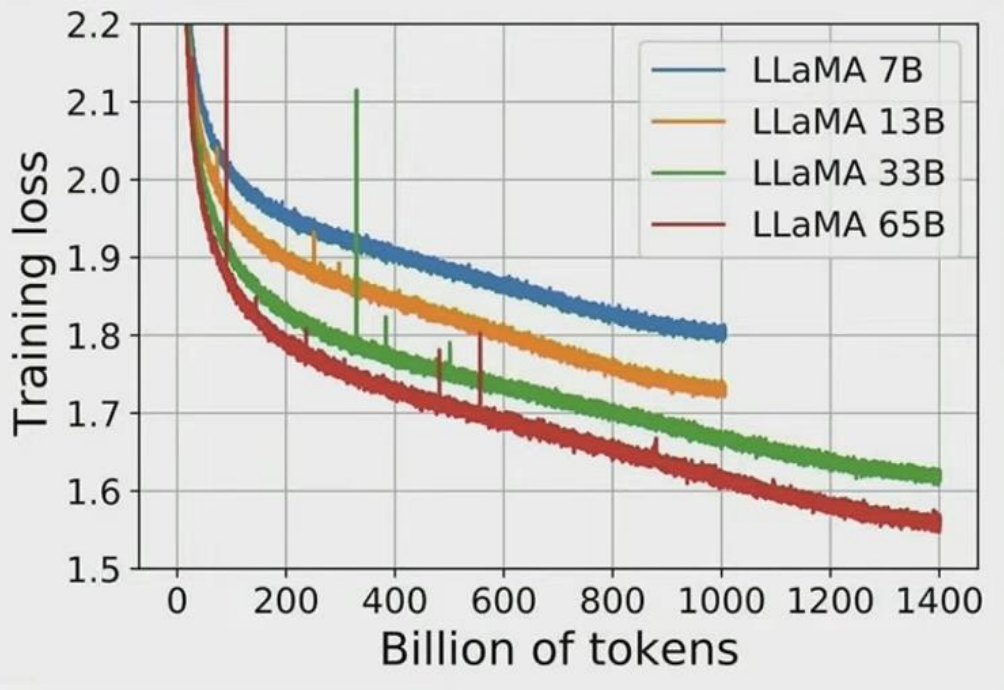
+

Longer training

= **Better LLM**

LLaMA follows Chinchilla law

[LLaMA: Open and Efficient Foundation Language Models,2023]



According to Chinchilla law:

- LLM model gets increasingly bigger
- Dataset gets increasingly larger
- Distributed training gets increasingly important

Thousands of GPUs are needed to train LLM

[State of GPT, 2023]

2 example models

GPT-3 (2020)

50,257 vocabulary size
2048 context length
175B parameters
Trained on 300B tokens

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

Training: (rough order of magnitude to have in mind)

- O(1,000 - 10,000) V100 GPUs
- O(1) month of training
- O(1-10) $M

LLaMA (2023)

32,000 vocabulary size
2048 context length
65B parameters
Trained on 1-1.4T tokens

params	dimension	n heads	n layers	learning rate	batch size	n tokens
6.7B	4096	32	32	$3.0e^{-4}$	4M	1.0T
13.0B	5120	40	40	$3.0e^{-4}$	4M	1.0T
32.5B	6656	52	60	$1.5e^{-4}$	4M	1.4T
65.2B	8192	64	80	$1.5e^{-4}$	4M	1.4T

Table 2: **Model sizes, architectures, and optimization hyper-parameters.**

Training for 65B model:

- 2,048 A100 GPUs
- 21 days of training
- $5M

Recent GPU clusters scale with more than 10000 GPUs

北京大学 PEKING UNIVERSITY

Feb. 23, 2024

MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs

Ziheng Jiang[1,*] Haibin Lin[1,*] Yinmin Zhong[2,*] Qi Huang[1] Yangrui Chen[1] Zhi Zhang[1]
Yanghua Peng[1] Xiang Li[1] Cong Xie[1] Shibiao Nong[1] Yulu Jia[1] Sun He[1] Hongmin Chen[1]
Zhihao Bai[1] Qi Hou[1] Shipeng Yan[1] Ding Zhou[1] Yiyao Sheng[1] Zhuo Jiang[1]
Haohan Xu[1] Haoran Wei[1] Zhang Zhang[1] Pengfei Nie[1] Leqi Zou[1] Sida Zhao[1]
Liang Xiang[1] Zherui Liu[1] Zhe Li[1] Xiaoying Jia[1] Jianxi Ye[1] Xin Jin[2,†] Xin Liu[1,†]

[1]*ByteDance* [2]*Peking University*

Abstract

We present the design, implementation and engineering experience in building and deploying MegaScale, a production system for training large language models (LLMs) at the scale of more than 10,000 GPUs. Training LLMs at this scale brings unprecedented challenges to training efficiency and stability. We take a full-stack approach that co-designs the algorithmic and system components across model block and optimizer design, computation and communication overlapping, operserving billions of users, we have been aggressively integrating AI into our products, and we are putting LLMs as a high priority to shape the future of our products.

Training LLMs is a daunting task that requires enormous computation resources. The scaling law [3] dictates that the model size and the training data size are critical factors that determine the model capability. To achieve state-of-the-art model capability, many efforts have been devoted to train large models with hundreds of billions or even trillions of parameters on hundreds of billions or even trillions of tokens. For example, GPT-3 [4] has 175 billion parameters and

Llama 3 uses 24000 GPUs in LLM training

北京大学 PEKING UNIVERSITY

Apr. 18, 2024

Build the future of AI with Meta Llama 3

To train our largest Llama 3 models, we combined three types of parallelization: data parallelization, model parallelization, and pipeline parallelization. Our most efficient implementation achieves a compute utilization of over 400 TFLOPS per GPU when trained on 16K GPUs simultaneously. We performed training runs on two custom-built 24K GPU clusters. To maximize GPU uptime, we developed an advanced new training stack that automates error detection, handling, and maintenance. We also greatly improved our

[Introducing Meta Llama 3: The most capable openly available LLM to date]

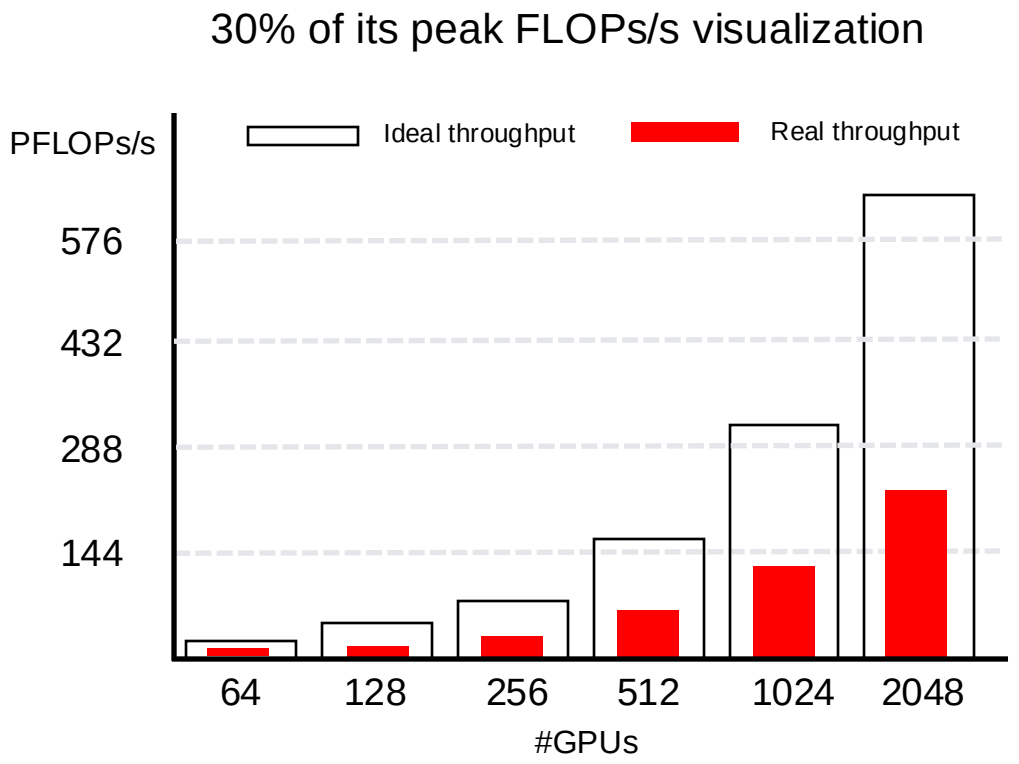
Distributed training over massive GPUs is extremely challenging

北京大学 PEKING UNIVERSITY

- Distributed training over thousands of GPUs is extremely challenging
- Two major challenges: **stability** and **scalability**
- Stability: very common that some GPU crashes during training LLMs
  - Meta OPT-175B: 175+ job restarts caused by hardware failures in 2 months
  - 书生大模型: Waste 41700 GPU hours due to training crashes (>80% are infrastructure failures)
- Stability in LLM training is very important, but this talk will not discuss it

Distributed training over massive GPUs is extremely challenging

- The **communication overhead** and **GPU idle time** severely hamper the scalability
- Each GPU can only achieve **30%~55%** of its peak FLOPs/s during LLM training
- When GPU achieves 30% of peak FLOP/s, we say the system achieves 30% scalability



Existing systems suffer from severe scalability issue

北京大学 PEKING UNIVERSITY

[Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, 2021]

Number of parameters (billion)	Attention heads	Hidden size	Number of layers	Tensor model-parallel size	Pipeline model-parallel size	Number of GPUs	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7	24	2304	24	1	1	32	512	137	44%	4.4
3.6	32	3072	30	2	1	64	512	138	44%	8.8
7.5	32	4096	36	4	1	128	512	142	46%	18.2
18.4	48	6144	40	8	1	256	1024	135	43%	34.6
39.1	64	8192	48	8	2	512	1536	138	44%	70.8
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0

Nvidia Megatron: the most popular distributed training framework

Existing systems suffer from severe scalability issue

- Open AI and Meta does not disclose its training scalability, but we can infer them

- End-to-end training time can be estimated by

$$\text{Training time} = \frac{\text{Total FLOPS to train the model}}{\text{FLOPs/s of the GPU cluster}} \approx \frac{8\,T\,P}{N\,F\,U}$$

T: the number of tokens

P: the number of parameters

N: the number of GPUs

F: Peak FLOP/s per GPU

U: Utilization

[Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, 2021]

Existing systems suffer from severe scalability issue

- **Open AI GPT3-175B**: 1024 A100 GPUs, 300B tokens, training time is 35 days

$$U \approx \frac{8 \times (300 \times 10^9) \times (175 \times 10^9)}{1024 \times (312 \times 10^{12}) \times (35 \times 24 \times 3600)} = \mathbf{0.44}$$

  The scalability is around 0.44

- **Meta LLaMA-65B**: 2048 A100 GPUs, 1.4TB tokens, training time is 21 days

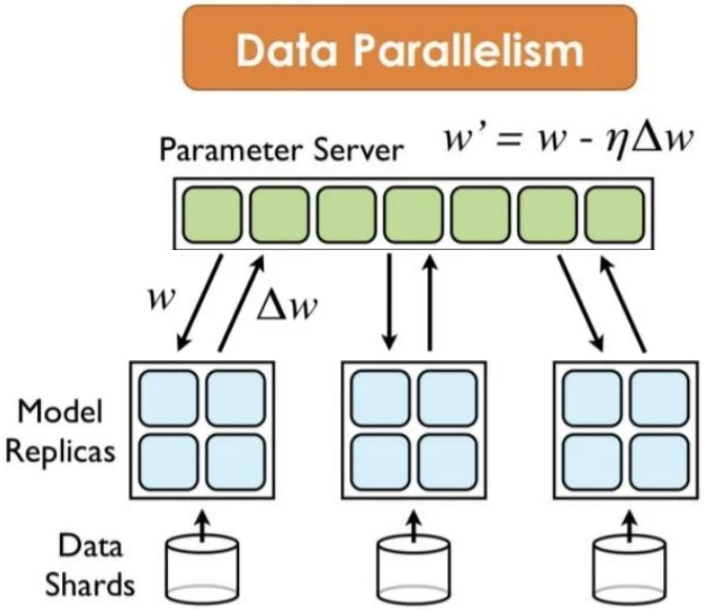
$$U \approx \mathbf{0.3}$$

  The scalability is around 0.3

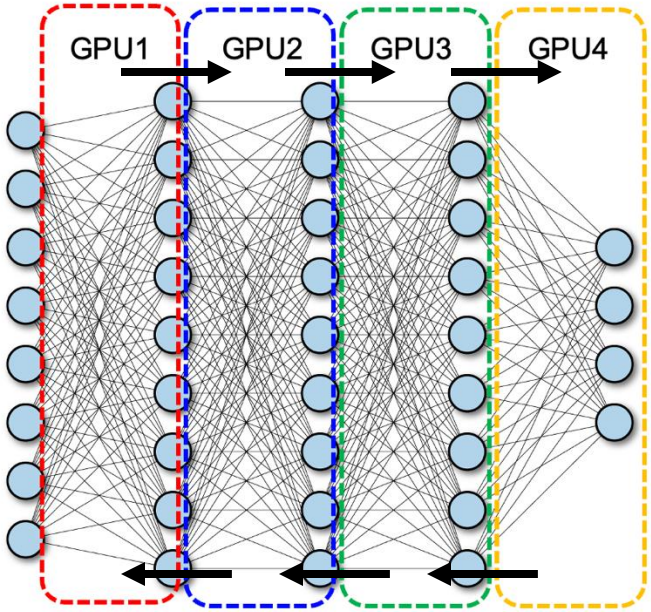
- We find even Nvidia, Open AI, and Meta **cannot** achieve strong scalability

Communication overhead is the top factor hampering the scalability

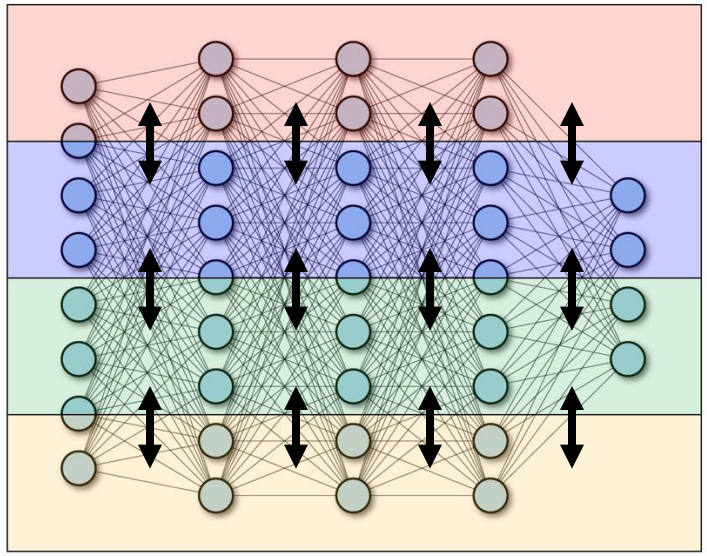
- 3D parallelism indicates 3 orthogonal parallel techniques used to train LLM



Data parallelism



Pipeline parallelism



Tensor parallelism

- This talk mainly focuses on **data parallelism**. → communication

北京大学
PEKING UNIVERSITY

PART 03

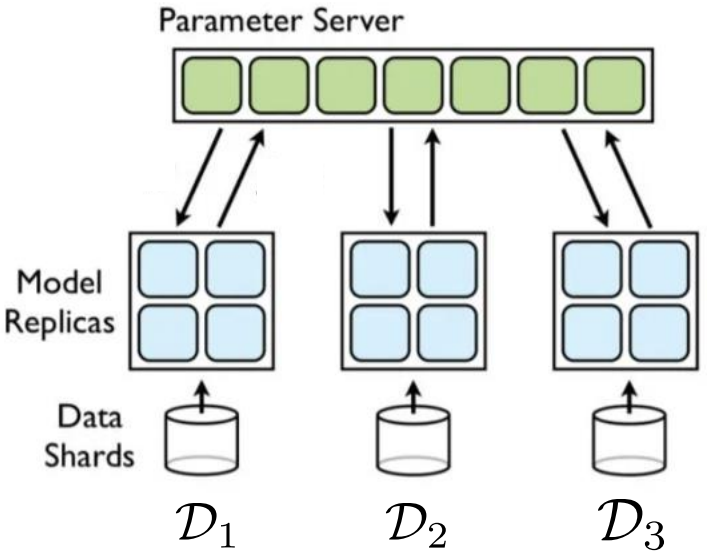
Data Parallelism

Data parallelism

A network of n nodes (devices such as GPUs) collaborate to solve the problem:

$$\min_{x\in\mathbb{R}^d} \quad f(x)=\frac{1}{n}\sum_{i=1}^{n} f_i(x), \quad \text{where} \quad f_i(x)=\mathbb{E}_{\xi_i\sim D_i} F(x;\xi_i)$$

- Each component $f_i : \mathbb{R}^d \to \mathbb{R}$ is local and private to node i
- Random variable ξ_i denotes local data that follows distribution D_i
- Each local distribution D_i is different; data heterogeneity exists



Vanilla parallel stochastic gradient descent (PSGD)

$$\min_{x\in\mathbb{R}^d} \quad f(x)=\frac{1}{n}\sum_{i=1}^{n} f_i(x), \quad \text{where} \quad f_i(x)=\mathbb{E}_{\xi_i\sim D_i}F(x;\xi_i).$$

PSGD

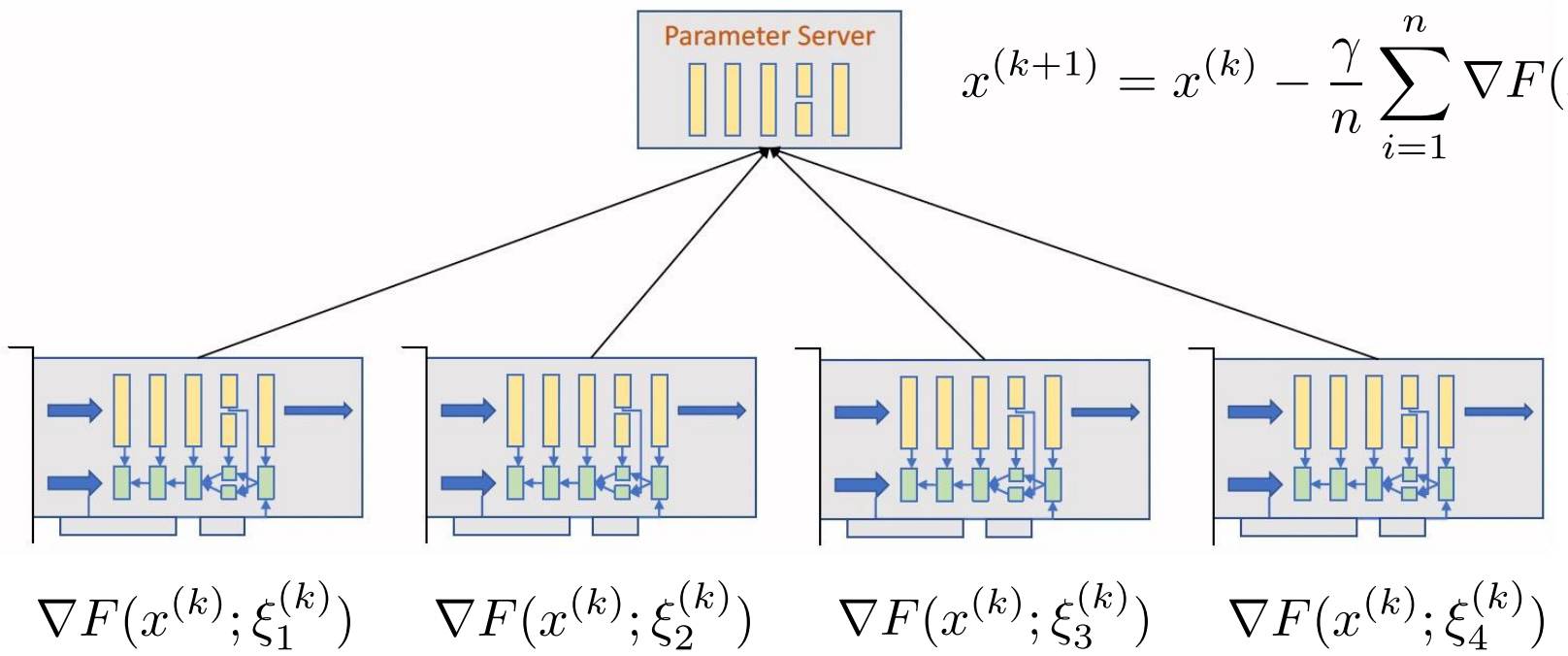
$$g_i^{(k)} = \nabla F(x^{(k)};\xi_i^{(k)}) \qquad \text{(Local compt.)}$$

$$x^{(k+1)} = x^{(k)} - \frac{\gamma}{n}\sum_{i=1}^{n} g_i^{(k)} \qquad \text{(Global comm.)}$$

- Each node i samples data $\xi_i^{(k)}$ and computes gradient $\nabla F(x^{(k)};\xi_i^{(k)})$
- All nodes synchronize (i.e. globally average) to update model x per iteration

Vanilla parallel stochastic gradient descent (PSGD)

- Federated learning typically implements PSGD using parameter server



- LLM training within data-centers implements PSGD using Ring-Allreduce

PSGD convergence rate

北京大学 PEKING UNIVERSITY

Assumption [PSGD assumption]

(1) Each $f_i(x)$ is L-smooth, i.e., $\|\nabla f_i(x) - \nabla f_i(y)\| \leq L\|x - y\|$ for any x, y;

(2) Each stochastic gradient is unbiased and has bounded variance, i.e.,

$$\mathbb{E}[\nabla F(x;\xi_i)] = \nabla f_i(x), \quad \mathbb{E}\|\nabla F(x;\xi_i) - \nabla f_i(x)\|^2 \leq \sigma^2$$

Theorem [PSGD convergence]

Under the above assumptions and with proper γ, PSGD converges as follows

$$\frac{1}{T}\sum_{k=1}^{T}\mathbb{E}\|\nabla f(x^{(k)})\|^2 = \mathcal{O}\left(\frac{\sigma}{\sqrt{nT}}\right)$$

[K. Yuan, Lecture 6: stochastic gradient descent, PKU Class “Optimization for deep learning”, check Kun Yuan’s website]

Linear speedup in PSGD

北京大学 PEKING UNIVERSITY

Theorem [PSGD convergence]

Under the above assumptions and with proper γ, PSGD converges as follows

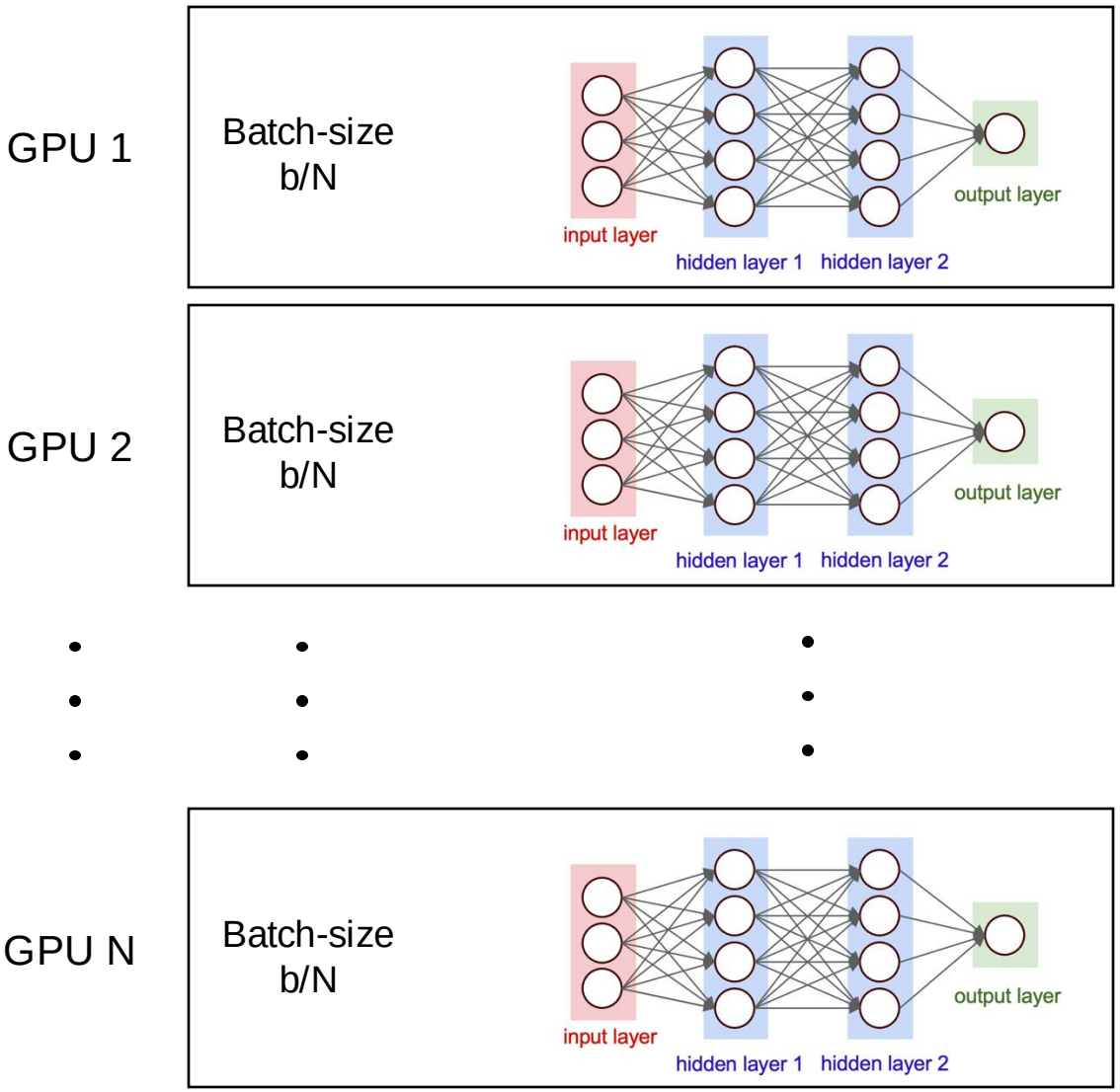
$$\frac{1}{T}\sum_{k=1}^{T}\mathbb{E}\|\nabla f(x^{(k)})\|^2 = \mathcal{O}\left(\frac{\sigma}{\sqrt{nT}}\right)$$

- This implies that to achieve an ϵ-accurate solution, PSGD needs

$$\frac{\sigma}{\sqrt{nT}} \leq \epsilon \quad \Longrightarrow \quad T \geq \frac{\sigma^2}{n\epsilon^2}$$

iterations, which decreases linearly with number of nodes n; this is called **linear speedup**

Data parallelism: memory and communication



Pros:

- Cuts the computation time by N folds
- Reduce the activation memory by N folds

$$2\left(4\Phi + \frac{b}{N}\Theta\right)$$

FP16 M+g+os = 4 # parameters # activations

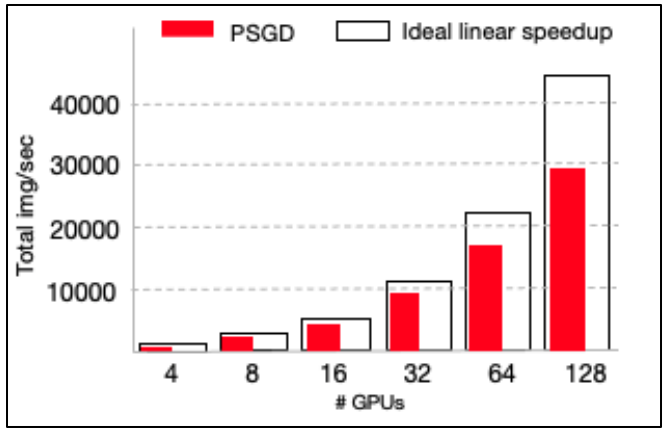
- Memory-efficient when $\Theta \gg \Phi$

Cons:

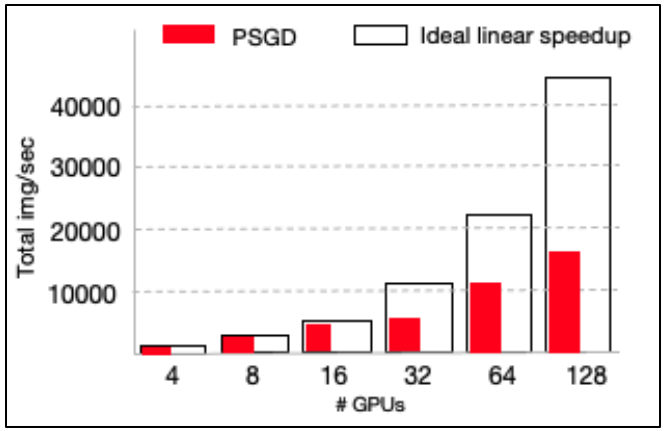
- Cannot save memory significantly if $\Phi \gg \Theta$
- Incurs 2Φ Bytes to communicate (extra time)

PSGD cannot achieve ideal scalability due to comm. overhead

- PSGD **cannot** achieve ideal linear speedup in throughput due to comm. overhead
- Larger comm-to-compt ratio leads to worse performance in PSGD



Small comm.-to-compt. ratio



Large comm.-to-compt. ratio

Ring-Allreduce is used in each experiment

- How can we reduce the communication overhead in PSGD?

B. Ying, K. Yuan, H. Hu, Y. Chen and W. Yin, “BlueFog: Make decentralized algorithms practical for optimization and deep learning”, arXiv: 2111. 04287, 2021

Methodologies to save communication

北京大学 PEKING UNIVERSITY

- Global average incurs $O(n)$ comm. overhead; proportional to network size n

  [Decentralized communication]

- Each node interacts with the server at every iteration; proportional to iteration numbers

  [Lazy communication]

- Each node sends a full model (or gradient) to the server; proportional to dimension d

  [Compressed communication]

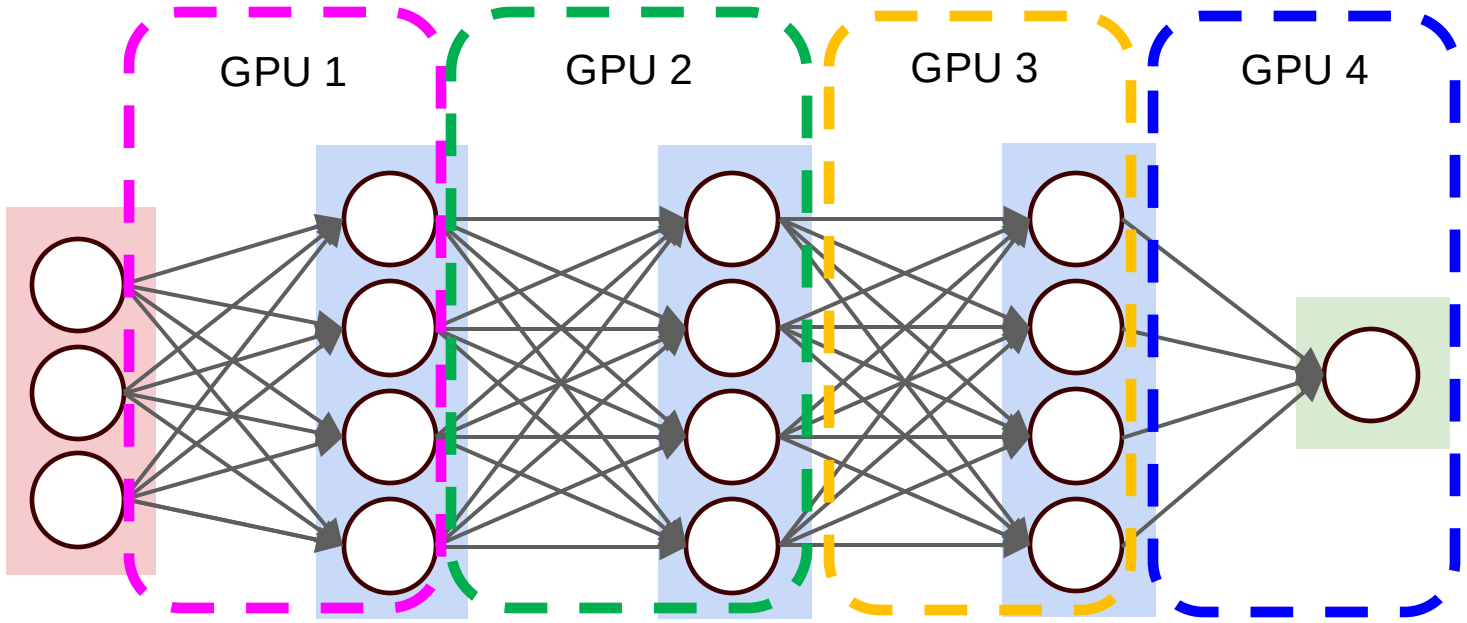
- and more (asynchronous communication; robust communication against Byzantine nodes, etc.)

PART 04

Pipeline Parallelism

Pipeline parallelism: architecture

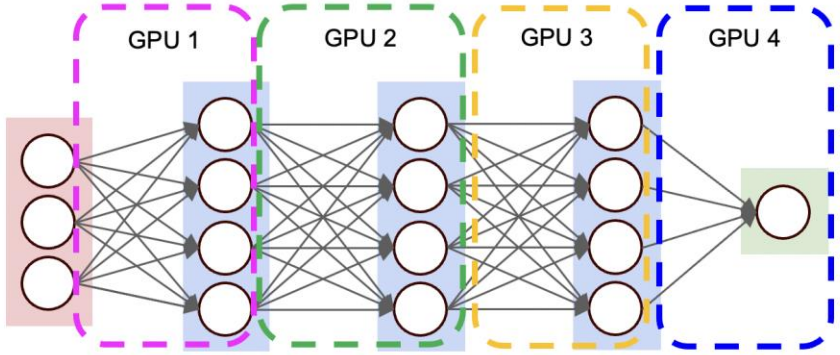
- Pipeline parallelism partitions layers and assigns them to different GPUs

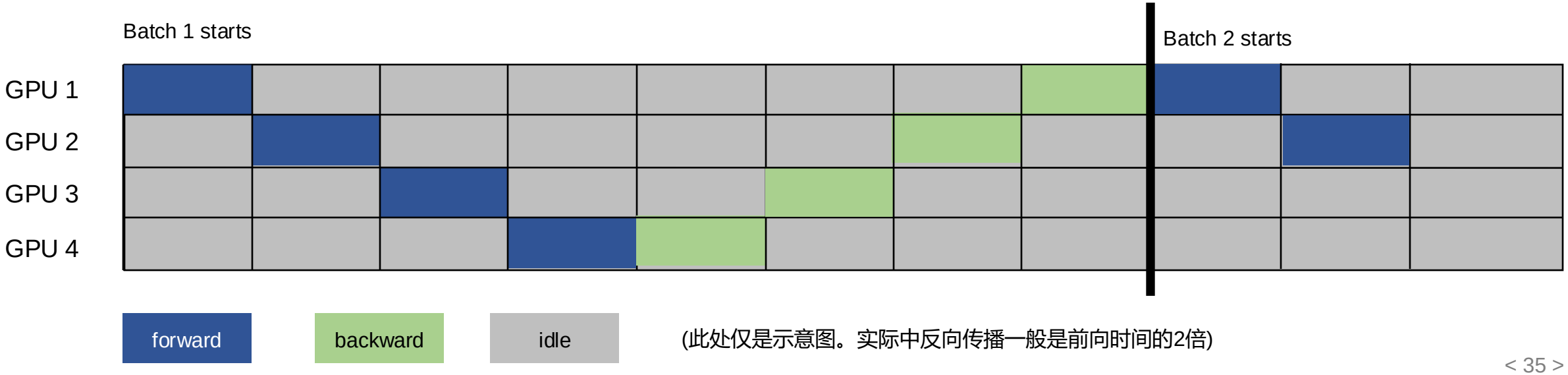


- If models are partitioned uniformly, the parameters (memories) are cut down by N folds

Model parallelism: no pipeline

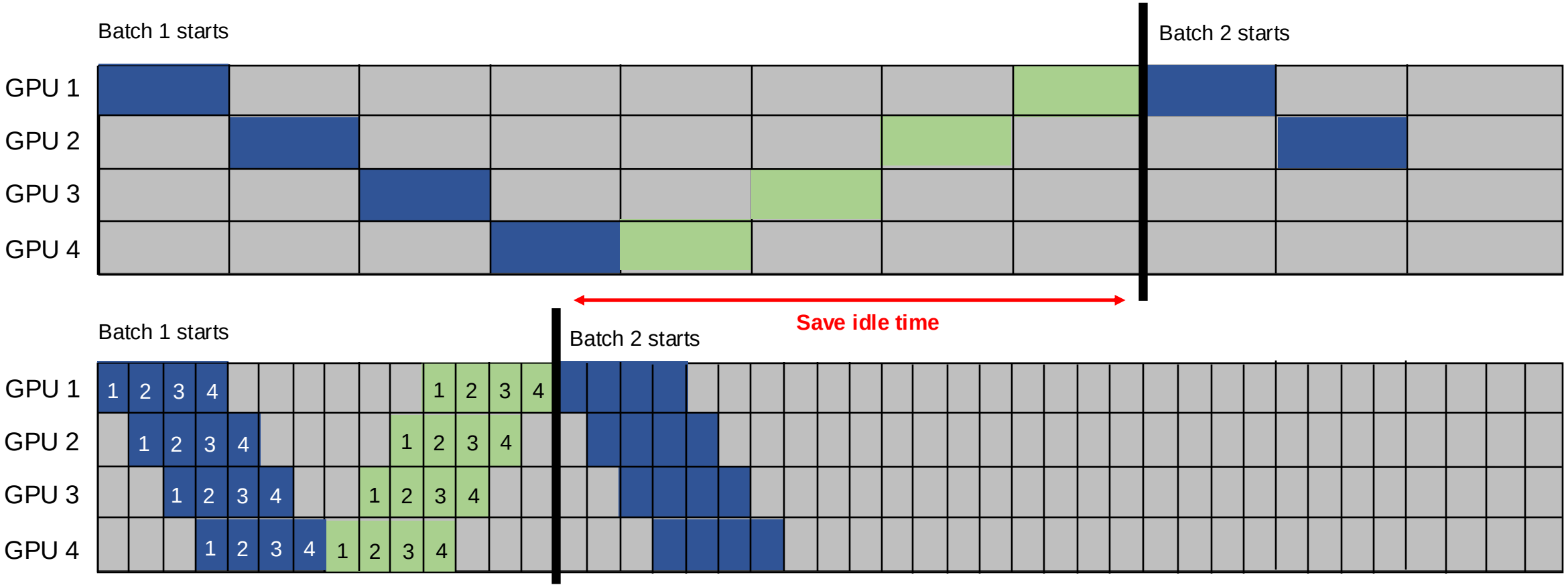
- Naïve model parallelism introduces substantial idle time. GPUs are not parallel.





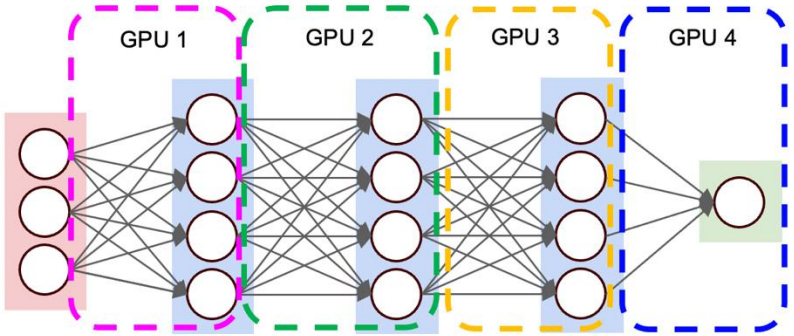
Pipeline parallelism: pipeline reduces idle time

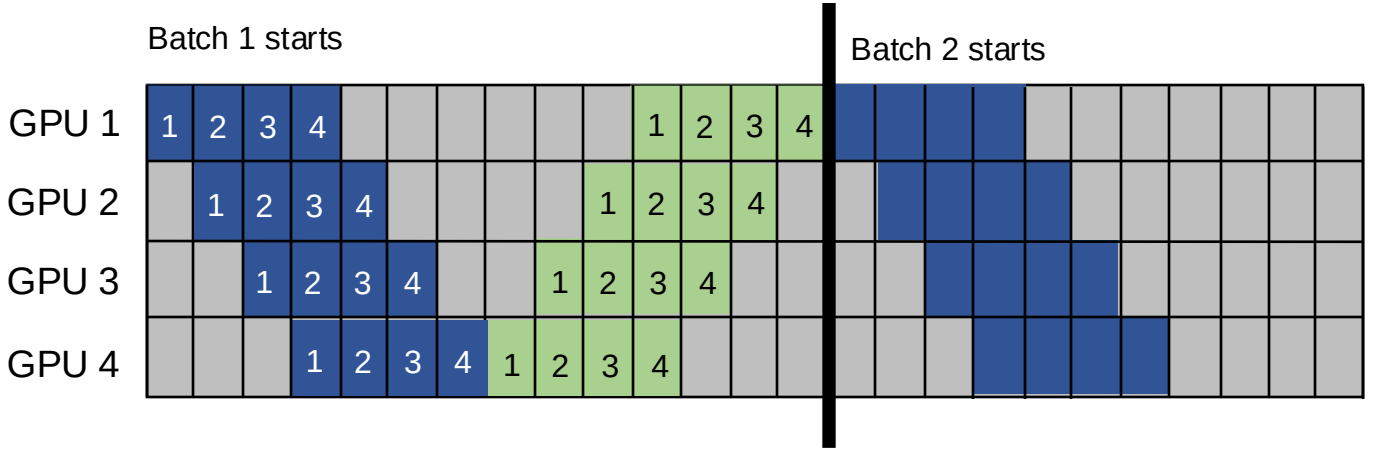
- Split one batch into a set of **micro-batches**



[GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism]

Pipeline parallelism: idle time



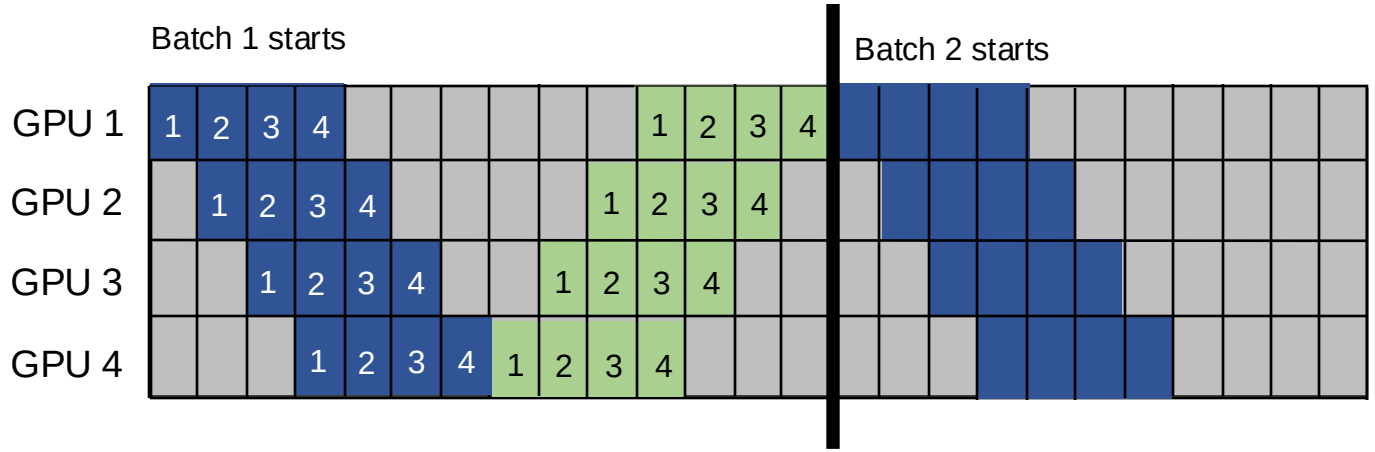


- While idle time is reduced, it cannot be fully eliminated
- If each partition is of similar size, the idle time fraction is $\frac{N-1}{m}$ N: # GPUs m: # micro-batches
- More micro-batches, less idle time (in line with the intuition)

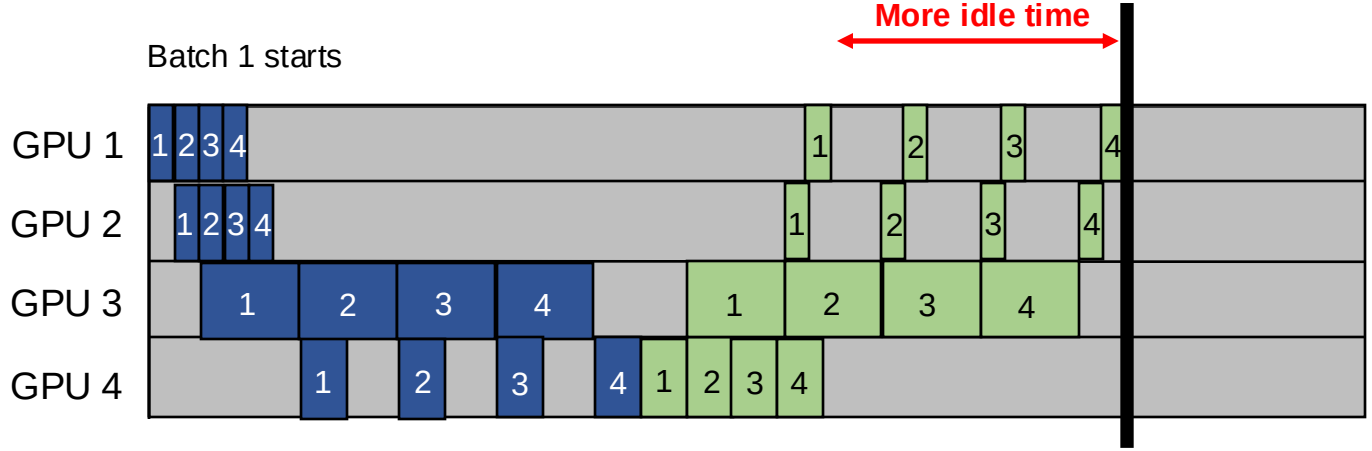
Pipeline parallelism: idle time

- If each partition has very different parameter size, the idle time fraction will be enlarged

Uniform partitions



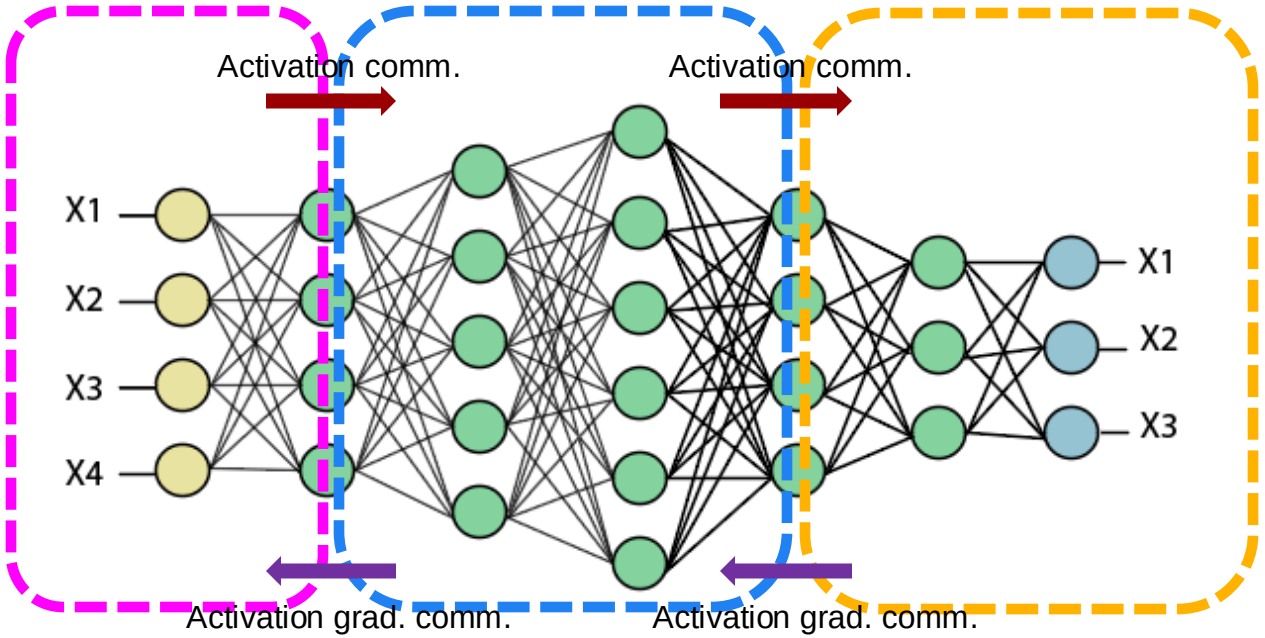
Non-uniform partitions



(idle time can also be estimated)

Pipeline parallelism: communication time

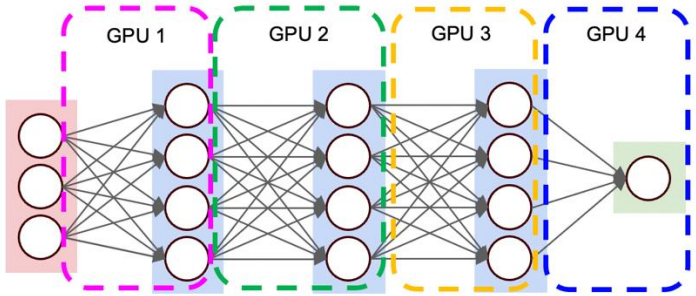
- Pipeline parallelism incurs additional communication of activation and activation gradients
- Not all activations are to be communicated, but those at the partition points



Micro-batch's communicated activation cost $\leq$ Pipeline comm. cost $\leq$ Full-batch's communicated activation cost

(can be modeled precisely)

Pipeline parallelism: total time



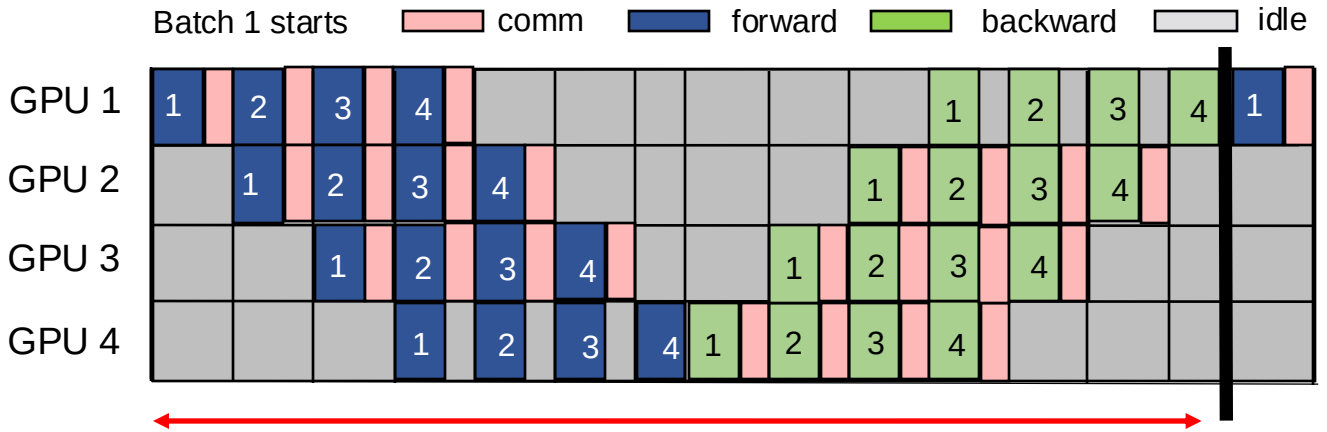
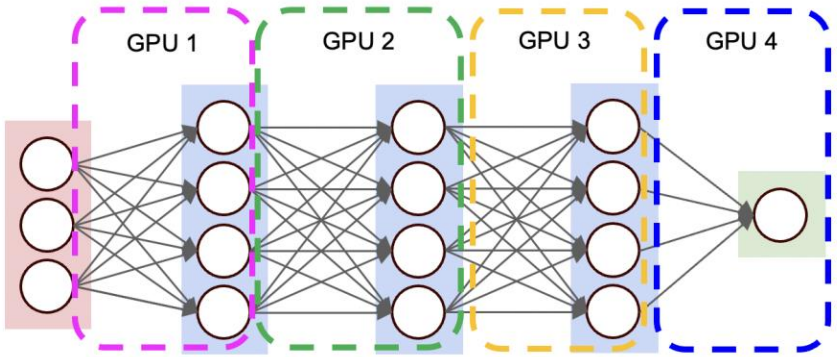


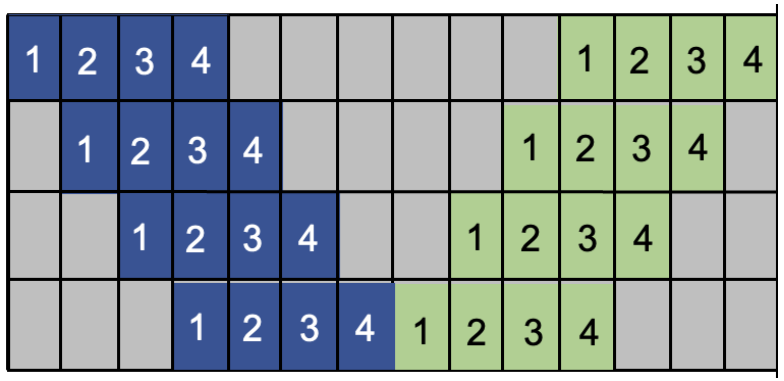
Illustration of the total time

- The above figure illustrates the very ideal scenario. While practical scenarios are more complicated (non-uniform partition), they share the same gist.

- Since each frame of comm., forward, backward, and idle time can be calculated, the total time can also be calculated

Pipeline parallelism: summary





Pros:

- Cuts the memory by N folds (if partition uniformly)
- Accelerates training due to micro-batch parallelism
- Less comm. overhead due to micro-batch (not full-batch) activation (not model) communication at partitioned layers (not all layers)

Cons:

- Introduce idle time which cannot be fully removed
- Difficult to partition layers uniformly

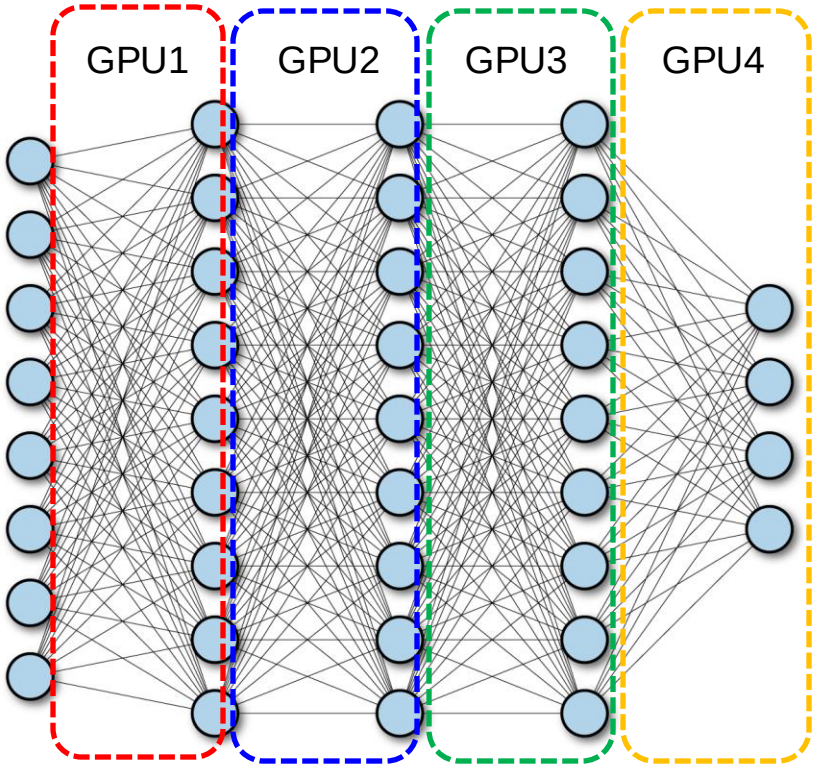
Performance can be fully computed

PART 05

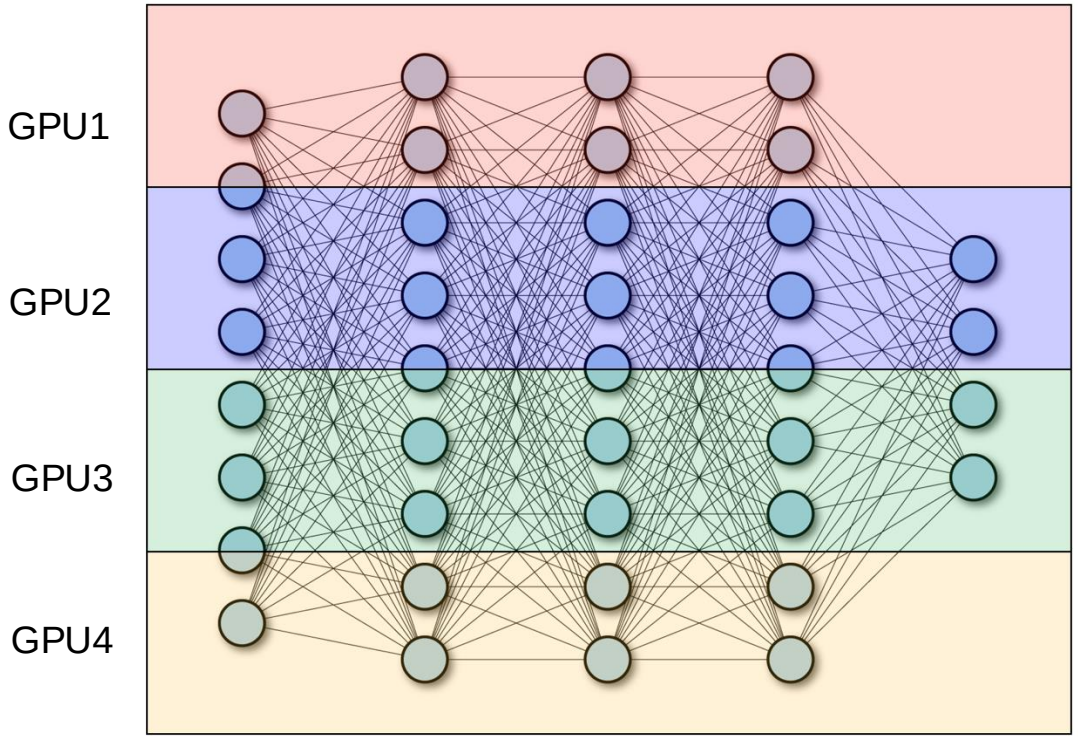
Tensor Parallelism

Tensor parallelism

- Tensor parallelism partitions tensors and assigns them to different GPUs (save memory)

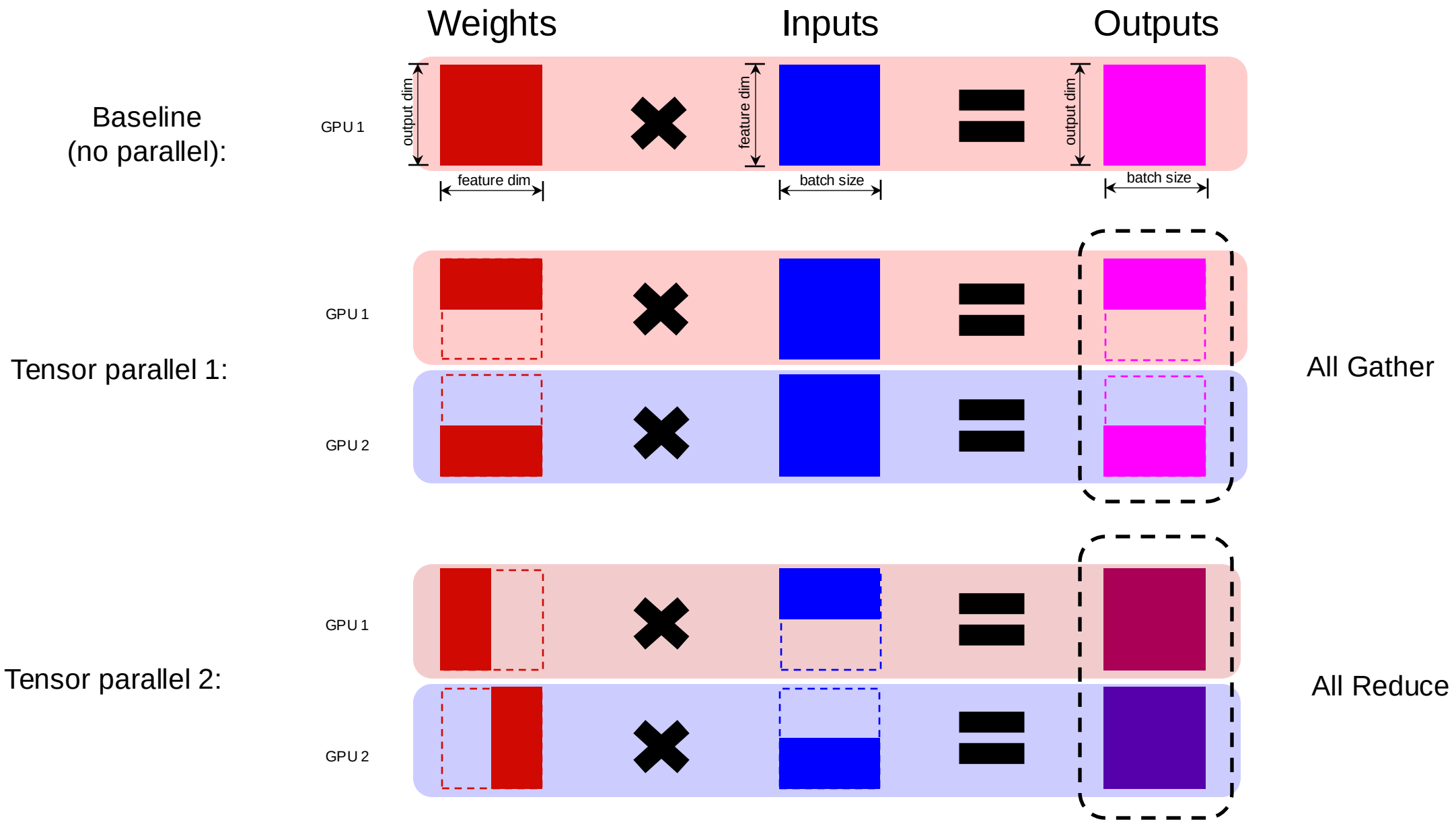


Pipeline parallelism



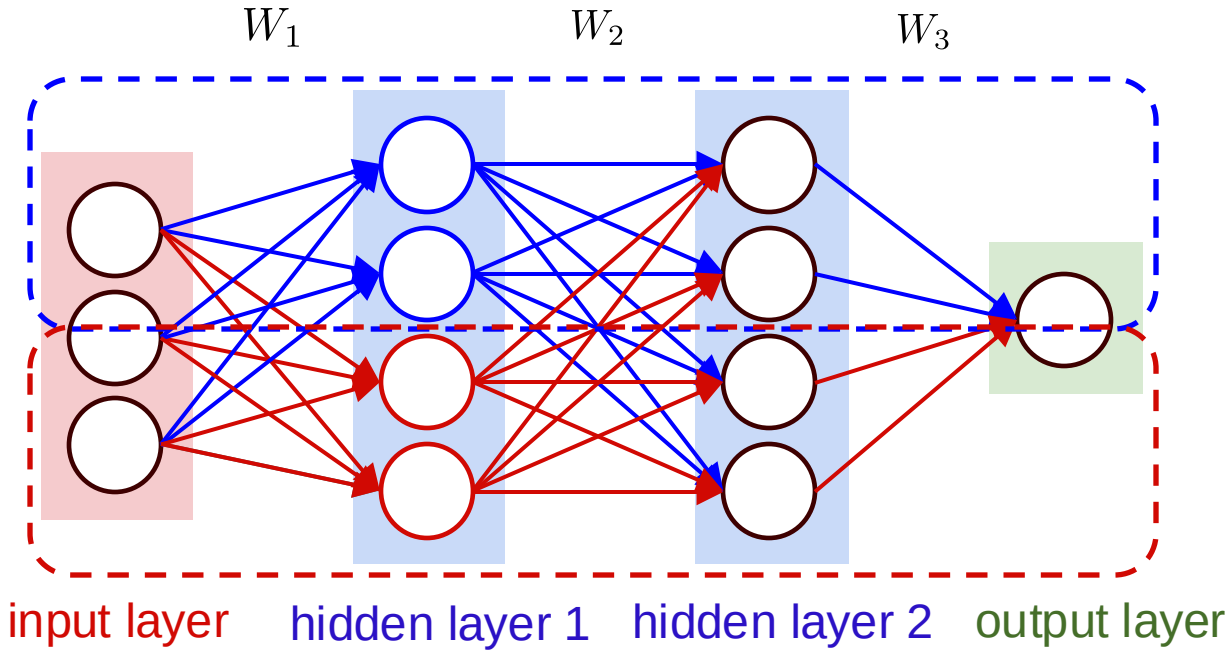
Tensor parallelism

Tensor parallelism is very flexible in partition manners

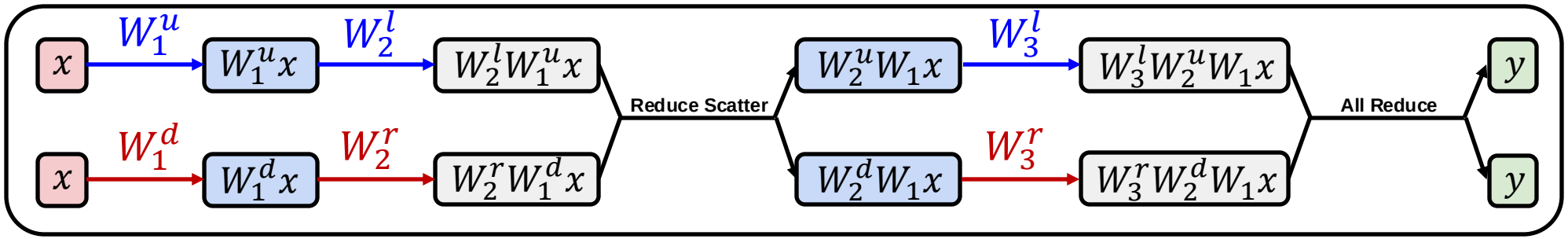


Tensor parallelism: an example

$$y = W_3 W_2 W_1 x$$



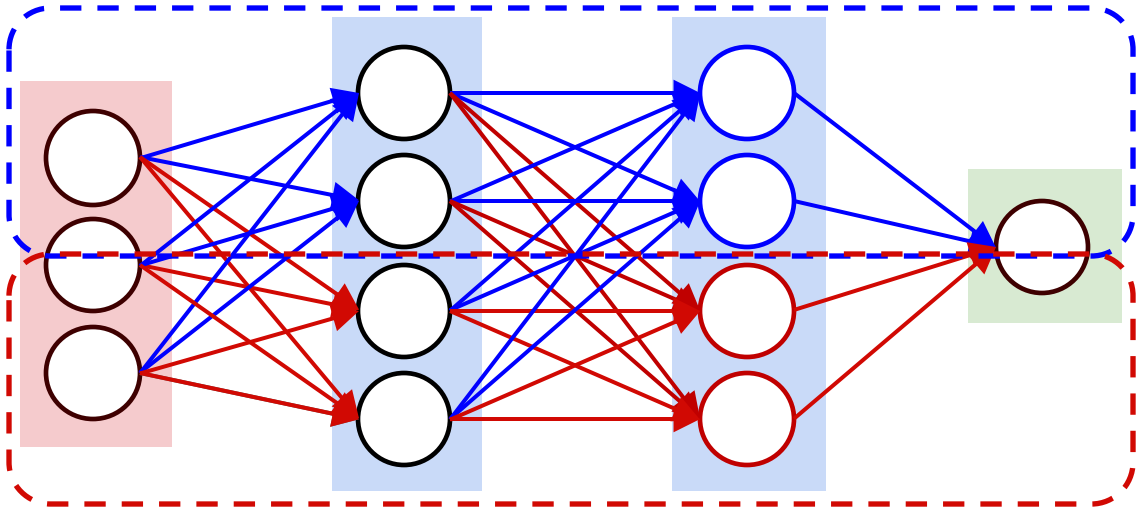
Partition 1: $\begin{bmatrix} W_1^u \\ W_1^d \end{bmatrix}$ $[W_2^\ell \;\; W_2^r]$ $[W_3^\ell \;\; W_3^r]$



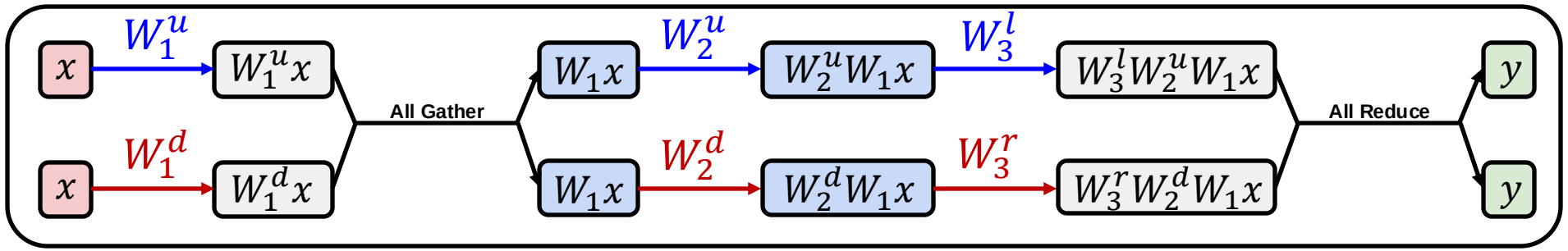
No idle time; only comm. and compt. time exist; almost each layer incurs comm.

Tensor parallelism: an example

$$y = W_3 W_2 W_1 x$$

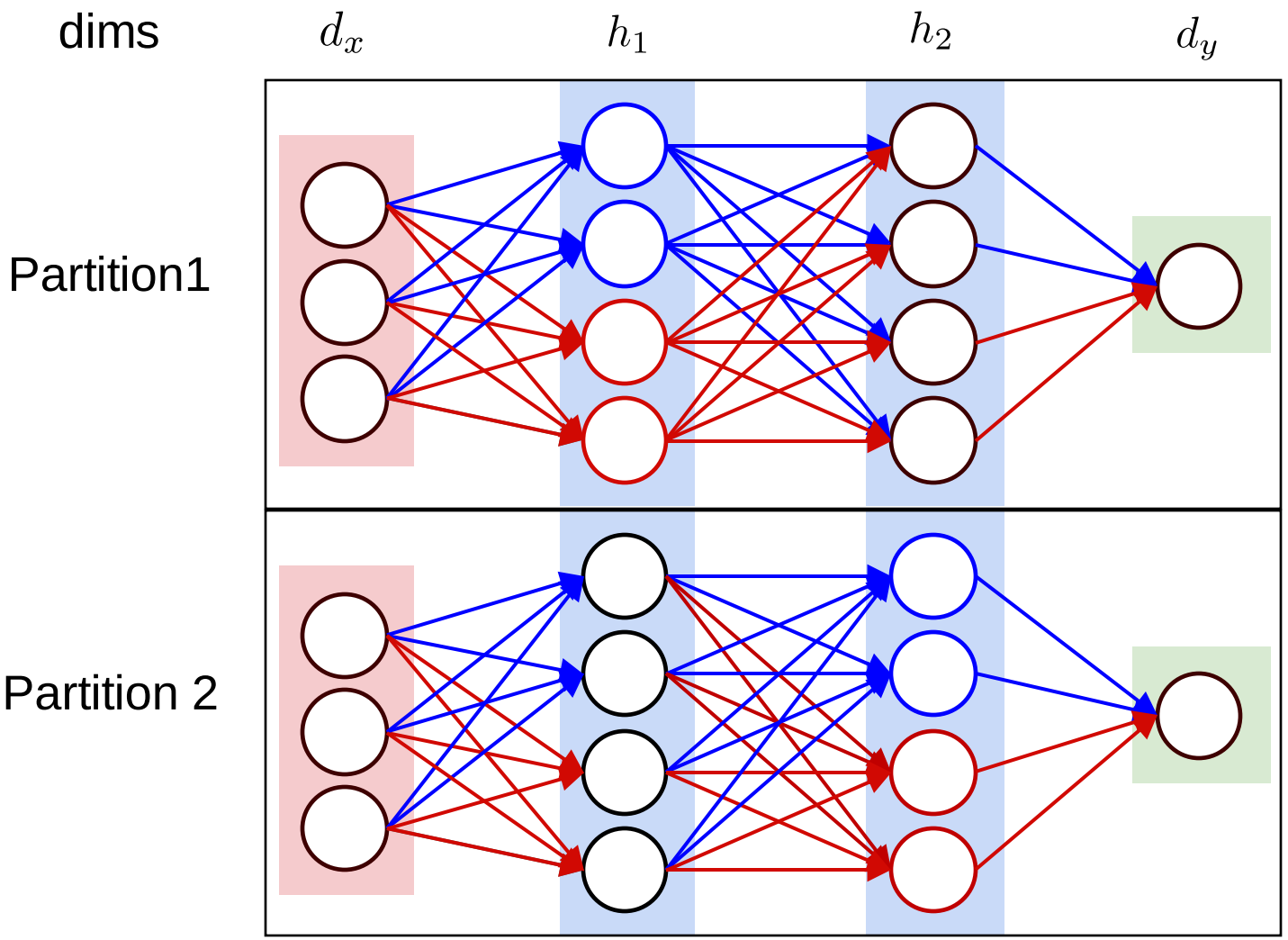


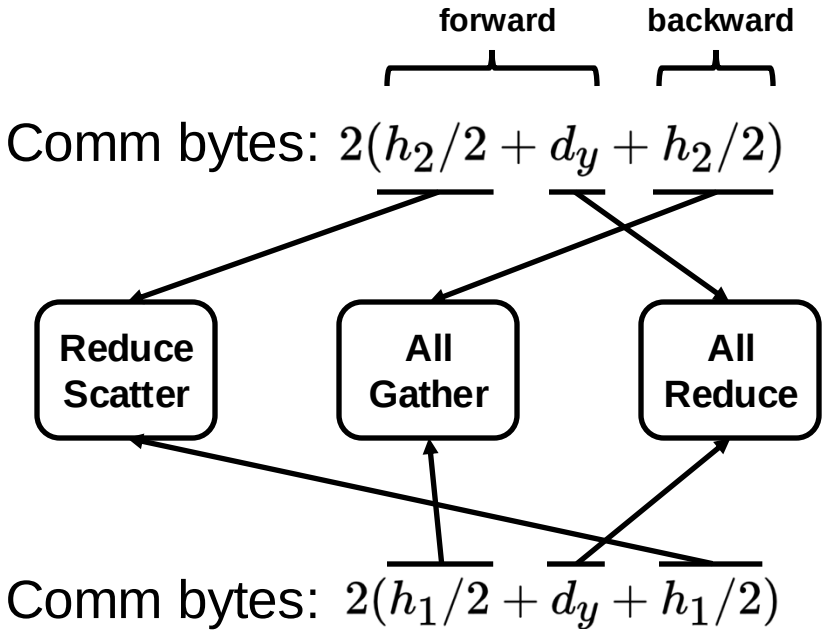
Partition 2: $\begin{bmatrix} W_1^u \\ W_1^d \end{bmatrix}$ $\begin{bmatrix} W_2^u \\ W_2^d \end{bmatrix}$ $[W_3^\ell \quad W_3^r]$



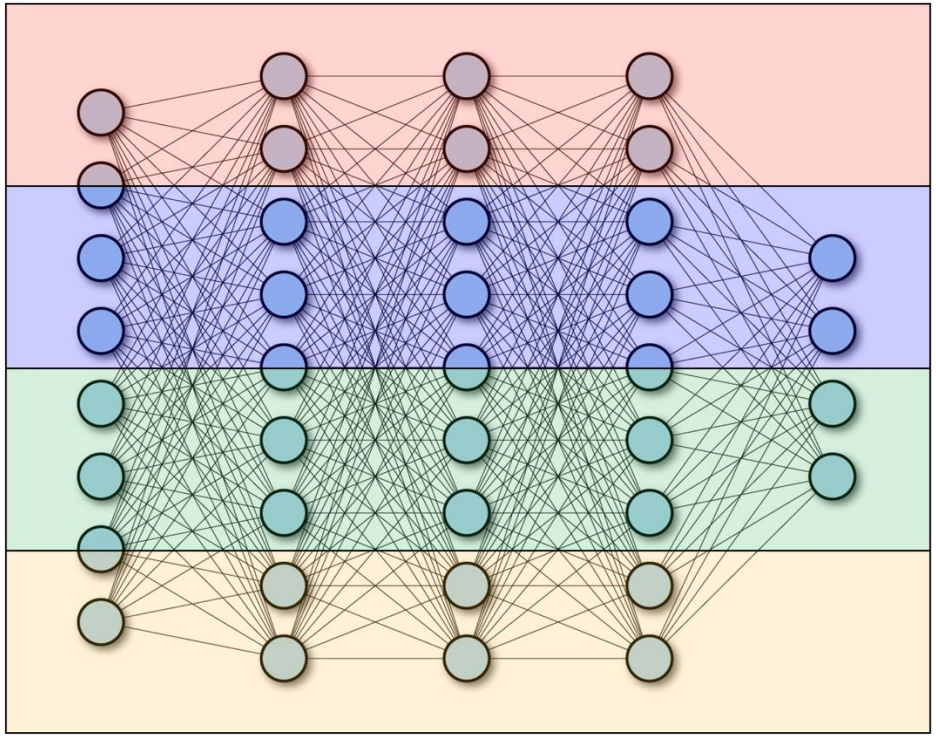
No idle time; only comm. and compt. time exist; almost each layer incurs comm.

Different partitions incurs different communication cost





Tensor parallelism: summary



Pros:

- Cuts the memory by N folds (if partition uniformly)
- Accelerates training due to parallelism
- No idle time

Cons:

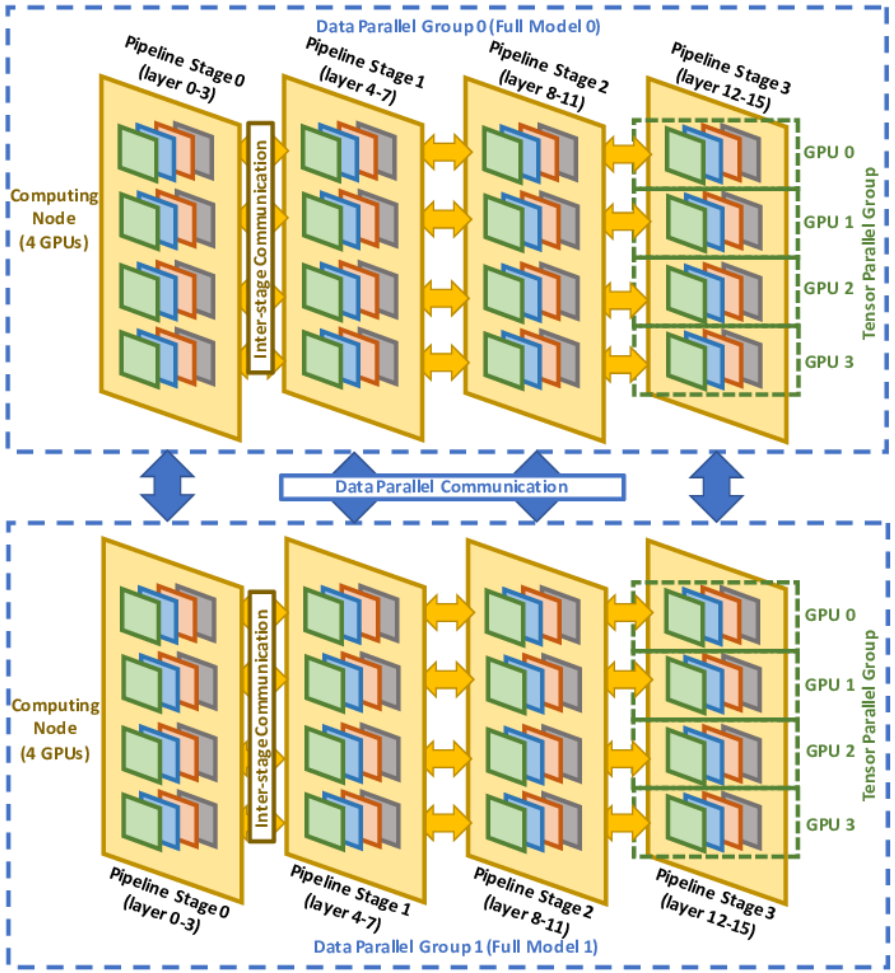
- Introduce more comm. than pipeline parallelism
- Partition affects comm.; non-trivial to find good partition

The training time can be fully computed since both computation and communication cost are clear

PART 06

3D Parallelism

3D parallelism



- 3D parallelism is the most common approach to organize GPUs
- It is critical to determine how to combine data/pipeline/tensor parallelism
- Different combination will incur very different communication cost

3D parallelism: summary

- Data parallelism reduces token cost but maintain the big model
- Pipeline parallelism reduces memory cost; incurs less comm. cost; introduces idle time; sensitive to how layers are partitioned
- Tensor parallelism reduces memory cost; incurs big comm. cost; does not introduce time; sensitive to how tensors are partitioned
- Different 3D parallel combination will result in different LLM training performance
- Each parallelism’s mechanism is very clear; training time can be fully computed

Search for better 3D parallel policy

- Given a cluster of GPUs, a LLM task, and a specific 3D parallel policy, we can evaluate whether the policy is feasible, and what the performance (throughput, scalability, etc.) is

- In fact, we have a performance simulator!

- Given the simulator, we can easily search for a better 3D parallel policy using
  - Reinforcement learning
  - Integer programming
  - Dynamic programming
  - Zeroth-order optimization

北京大学
PEKING UNIVERSITY

Thank you!

Kun Yuan homepage: https://kunyuan827.github.io/