IERG5050 AI Foundation Models, Systems and Applications
Spring 2025

Distributed/ Parallel Training of Foundation Models

Prof. Wing C. Lau
wclau@ie.cuhk.edu.hk
http://www.ie.cuhk.edu.hk/wclau

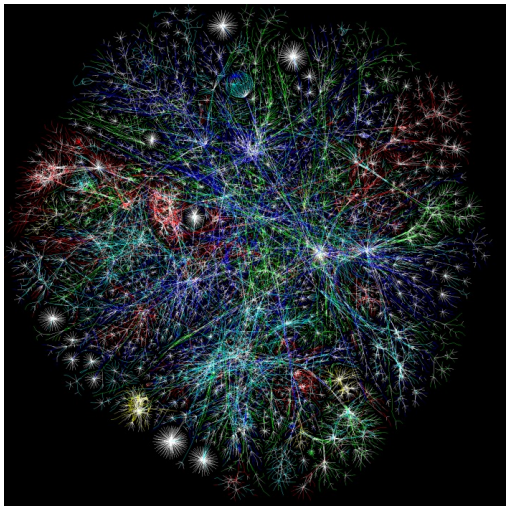
Acknowledgements

Many of the slides in this lecture are adapted from the sources below. Copyrights belong to the original authors.

- UC Berkeley CS294-162: AI-Systems (LLM Edition), Fall 2023, by Profs. Joseph E. Gonzalez and Matei Zaharia, https://learning-systems.notion.site/AI-Systems-LLM-Edition-294-162-Fall-2023-661887583bd340fa851e6a8da8e29abb
- UC Berkeley CS294: AI-Sys, Spring 2022, by Profs. Joseph E. Gonzalez and Amir Gholami, https://ucbrise.github.io/cs294-ai-sys-sp22/
- Stanford CS336: Language Modeling from Scratch, Spring 2024, by Profs. Tatsunori Hashimoto, Percy Liang, https://stanford-cs336.github.io/spring2024/
- MIT 6.5940: TinyML and Efficient Deep Learning Computing, Fall 2024, by Prof. Song Han, https://hanlab.mit.edu/courses/2024-fall-65940
- Stanford CS229S: Systems for Machine Learning, Fall 2023, by Profs. Azalia Mirhoseini, Simran Arora, https://cs229s.stanford.edu/fall2023/
- CMU 11-667: Large Language Models: Methods and Applications, Fall 2024, by Profs. Chenyan Xiong and Daphne Ippolito, https://cmu-llms.org
- Yann Dubois, “Introduction to Building LLMs,” Guest Lecture for Stanford CS229 Machine Learning, Aug 13, 2024, https://www.youtube.com/watch?v=9vM4p9NN0Ts ; https://drive.google.com/file/d/1B46VFrqFAPAEj3kaCrBAtQqeh2_Ztawl/view?usp=sharing
- UWaterloo CS886: Recent Advances on Foundation Models, Winter 2024, by Prof. Wenhu Chen, https://cs.uwaterloo.ca/~wenhuche/teaching/cs886/
- UPenn CIS7000: Large Language Models, Fall 2024, by Prof. Mayur Naik, https://llm-class.github.io/schedule.html
- CUHK-SZ CSC6203: Large Language Models, Fall 2024, by Prof. Benyou Wang, https://llm-course.github.io ; https://github.com/FreedomIntelligence/CSC6203-LLM
- Dr. Andrej Karpathy, Intro to LLMs, Nov. 2023, https://drive.google.com/file/d/1pxx_ZI7O-Nwl7ZLNk5hI3WzAsTLwvNU7
- CMU 15-442/15-642: Machine Learning Systems, Spring 2024, by Profs. Tianqi Chen and Zhihao Jia, https://mlsyscourse.org
- Stanford CS25: Transformer United V4, Spring 2024, https://web.stanford.edu/class/cs25/ by Instructors: Div Garg, Steven Feng, Seonghee Lee, Emily Bunnapradist, Faculty Advisor: Prof. Chris Manning, https://docs.google.com/presentation/d/1oXPs3LXtIVIsVbwTyGjAWj_aWvak9c1uNC4uhkS6glk/edit?usp=sharing

How to make (train) a LLM ?

Think of it like compressing the internet.



Chunk of the internet,
~10 TB of Data



6000 GPUs for 12 days, ~$2M
~1e24 FLOPS

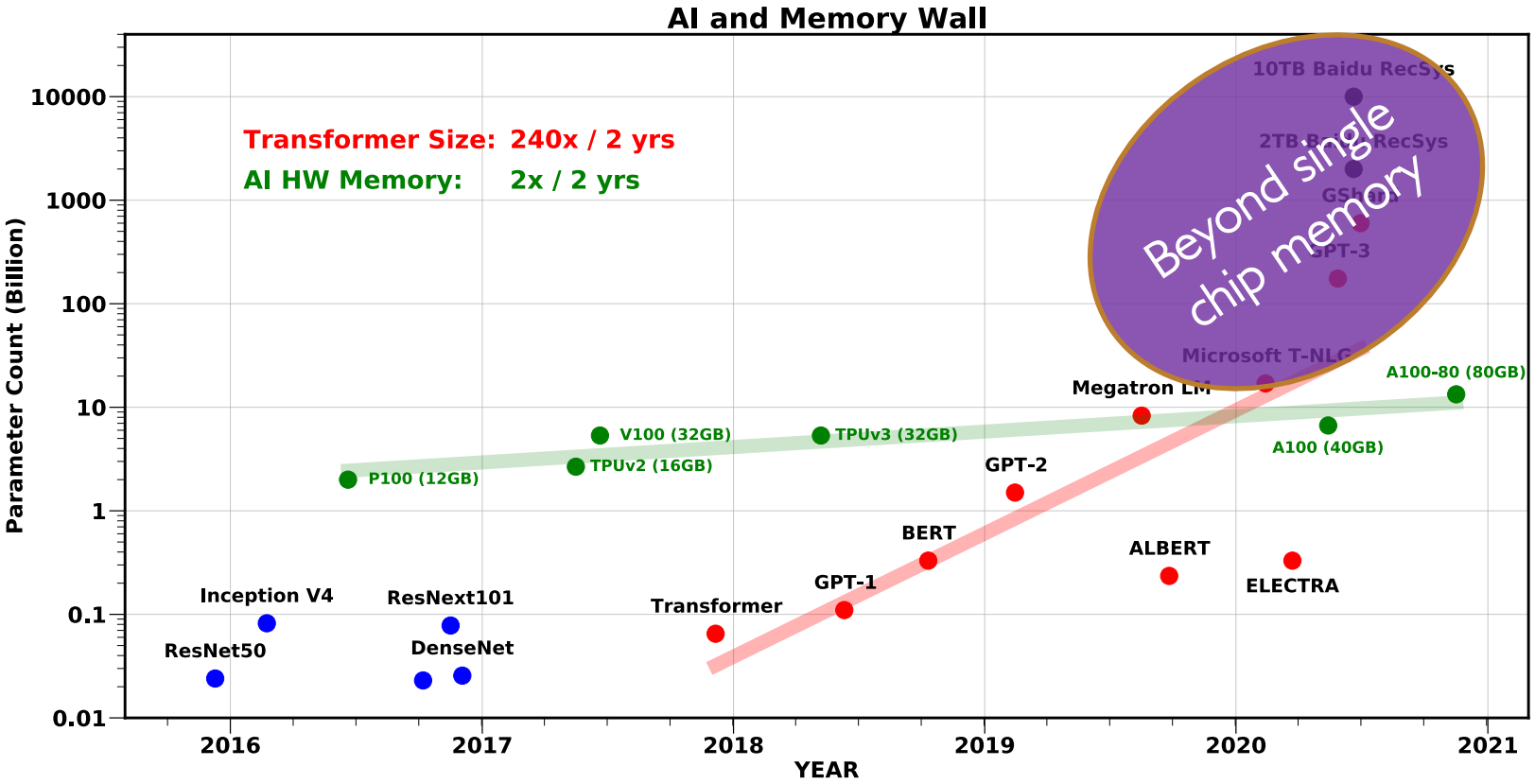


parameters.zip

~140GB file

*Numbers for Llama 2 70B ONLY

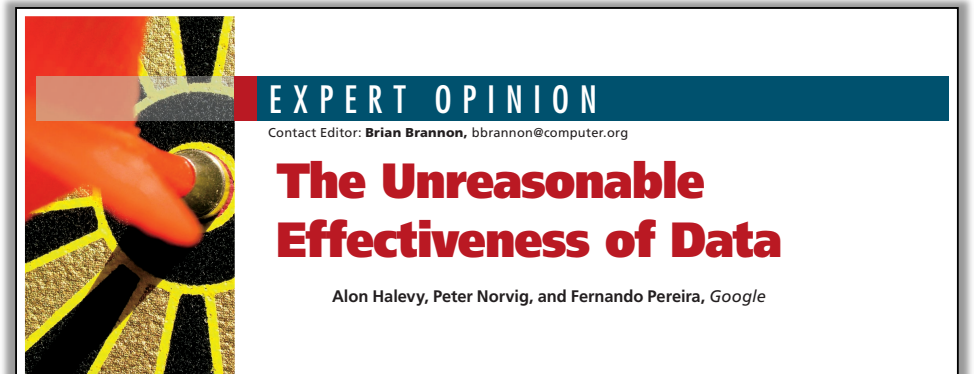
“Ancient” History of Training Large Models

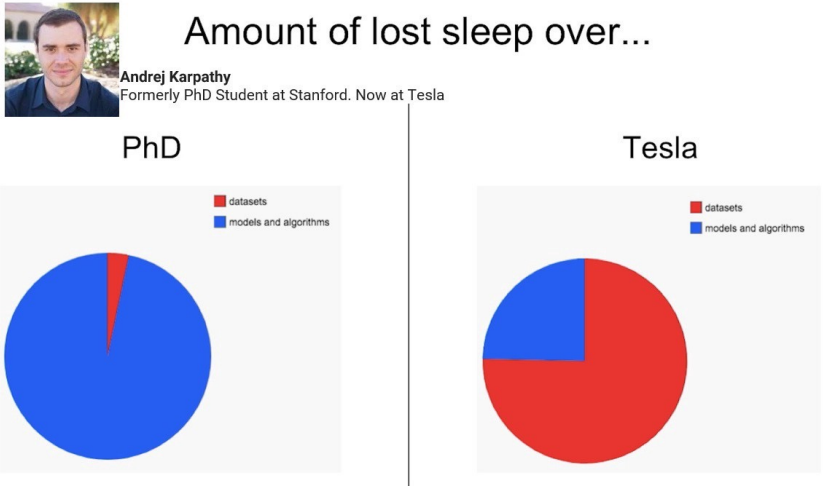


Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, Kurt Keutzer, AI and Memory Wall, Riselab Medium Blogpost, 2021.

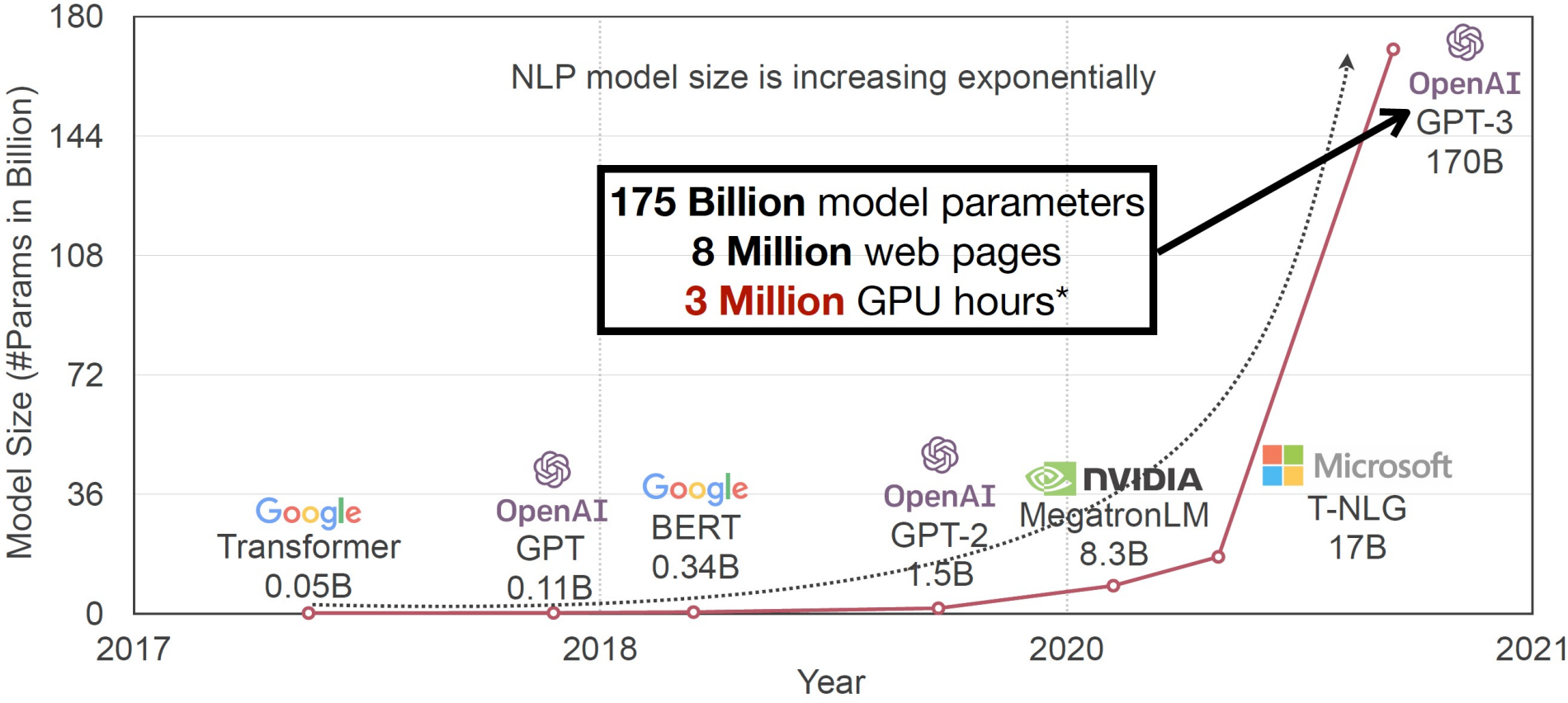
On Dataset Size and Learning

- Data is a resource! (e.g., like processors and memory)
  - Is having lots of processors a problem?
- You don't have to use all the data!
  - Though using more data can often help
- More data *often** dominates models and algorithms





Better (Bigger) models comes with Higher Computational Cost



Figures style adapted from Microsoft Turing Project

* Numbers referenced from https://lambdalabs.com/blog/demystifying-gpt-3

Better (Bigger) models Takes Longer Time to Train

Models	#Params (M)	Training Time (GPU Hours)
ResNet-50	26	31
ResNet-101	45	44
BERT-Base	108	84
Turing-NLG 17B	17,000	TBA
GPT-3 175B	175,000	3,100,000

Measured on Nvidia A100

If without distributed training, a single GPU would take **355 years** to finish GPT-3!

* Numbers referenced from https://lambdalabs.com/blog/demystifying-gpt-3

Distributed and Parallel Training for Foundation Models becomes a MUST !

- Developers/ Researchers' time are more valuable than Hardware
- If a training task takes 10 GPU days
  - Parallelize with Distributed Training
  - IDEALLY, 1024 GPUs can finish in 14 minutes
- The Development and Research Cycle for Foundation Models will be greatly boosted !

Compute Resource for Training LLMs (2024)

Source: https://docs.google.com/spreadsheets/d/1kc262HZSMAWI6FVsh0zJwbB-ooYvzhCHaHcNUiA0_hY

Model	Training end	Chip type	TFLOP/s (max)	Chip count	Wall clock time (days)	Total time (hours)	Total time (years)	Retail cost ($US)	MMLU ▼	Check (calculated column)
GPT-3	Apr/2020	V100	130	10,000	15 days	3,552,000	405 years	$9M	43.9	$8,808,960
Llama 1	Jan/2023	A100	312	2,048	21 days	1,032,192	118 years	$4M	63.4	$4,056,515
Llama 2	Jun/2023	A100	312	*2,048*	35 days	1,720,320	196 years	$7M	68.0	$6,760,858
Titan	Apr/2023	A100	312	13,760	48 days	11,558,400	1,319 years	$45M	70.4	$45,424,512
GPT-4	Aug/2022	A100	312	25,000	95 days	57,000,000	6,503 years	$224M	86.4	$224,010,000
Gemini	Nov/2023	TPUv4	275	57,000	100 days	136,800,000	15,606 years	$440M	90.0	$440,496,000
Llama 3 70B	Apr/2024	H100	989	24,576	*11 days*	6,300,000	719 years	$7M	82.0	$7,560,000
Llama 3 405B	Apr/2024	H100	989	24,576	50 days	29,491,200	3,364 years	$125M	88.6	$125,337,600
GPT-5	*Mar/2024*	H100	989	*50,000*	*120 days*	144,000,000	16,428 years	*$612M*		$612,000,000
Olympus	*Aug/2024*	H100	989							$0
Grok 2	*Jun/2024*	H100	989	*20,000*	*50 days*	57,600,000	6,571 years	*$245M*		$244,800,000
Gemini 2	*Nov/2024*	TPUv6	1847							$0
Grok 3	*Dec/2024*	H100	989	*100,000*	*50 days*	288,000,000	32,855 years	*$1.2B*		$1,224,000,000
										$0

Model	Source type (pr	Link	Quote
GPT-3	Primary	https://arxiv.c	"All models were trained on V10
	Analysis	https://arxiv.c	Note: this is a nearly-primary so
GPT-4	Analysis	https://www.s	"According to SemiAnalysis, GPT
Llama 1	Primary	https://arxiv.c	"When training a 65B-paramete
Llama 2	Primary	https://github	"Llama 2 70B: Time (GPU Hours
	Estimate only		2,048 chips is an assumption ba
Titan	Analysis	https://import	"200B dense model on 4T toke
	Primary	https://aws.a	"MMLU=70.4"
Gemini	Primary	https://arxiv.c	"Training Gemini Ultra used a large fleet of TPUv4 accelerators across multiple datacenters."
	Analysis	https://www.s	"The TPUv4 has a maximum performance of 275 TFLOP/s in bf16."
	Analysis	https://www.s	"According to SemiAnalysis... Ge
GPT-5	Estimate only	https://lifearcl	Alan's estimate based on comparison and extrapolating from Morgan Stanley research note and other sources. May be 2x-5x.
Llama 3	Primary	https://blogs.r	"Meta engineers trained Llama 3 on computer clusters packing 24,576 NVIDIA H100 Tensor Core GPUs"
	Primary	https://github	"Llama 3 70B6.4M [GPU hours]" extrap to 405B params
Olympus			
Grok 2	Primary	https://www.c	"In an interview with Norway wealth fund CEO Nicolai Tangen on Twitter/X spaces... Musk said training the Grok 2 model takes about 20,000 Nvidia H100 GPUs... training the Grok 3 model
Gemini 2			
Grok 3	Primary	https://www.c	"In an interview with Norway wealth fund CEO Nicolai Tangen on Twitter/X spaces... Musk said training the Grok 2 model takes about 20,000 Nvidia H100 GPUs... training the Grok 3 model
Other	1. Training end: Most training end dates assume 1 month before release for easy figures.		
	2. Total time (hours): column hidden in final report.		
	3. US$ estimate: Cost estimates using GCP in 2023: A100 @$3.93/hr, TPUv4 @$3.22/hr, H100 @$4.25/hr.		

Chip type	Pricing date	$ per chip-hour	Source	1M hours
V100	2020	$0.66	https://web.archiv	$660,000
V100	2020	$2.48	https://www.top5(	$2,480,000
A100	2023	$3.93	https://gpus.llm-u	$3,930,000
H100	2023	$4.25	https://web.archiv	$4,250,000
TPUv4	2023	$3.22	https://web.archiv	$3,220,000
TPUv5e	2024	$1.20	https://web.archiv	$1,200,000
TPUv5p	2024	$4.20	https://web.archiv	$4,200,000
TPUv6 (Trillium)	2024			$0

This sheet is owned and maintained by Dr Alan D. Thompson at LifeArchitect.ai.

Big pricing disparity between lambda and GCP

Using this number for GPT-3 in final table

Todo:

https://www.reddit.com/r/LocalLLaMA/comments

Example:
Scale is Google TPU's Primary Value Proposition



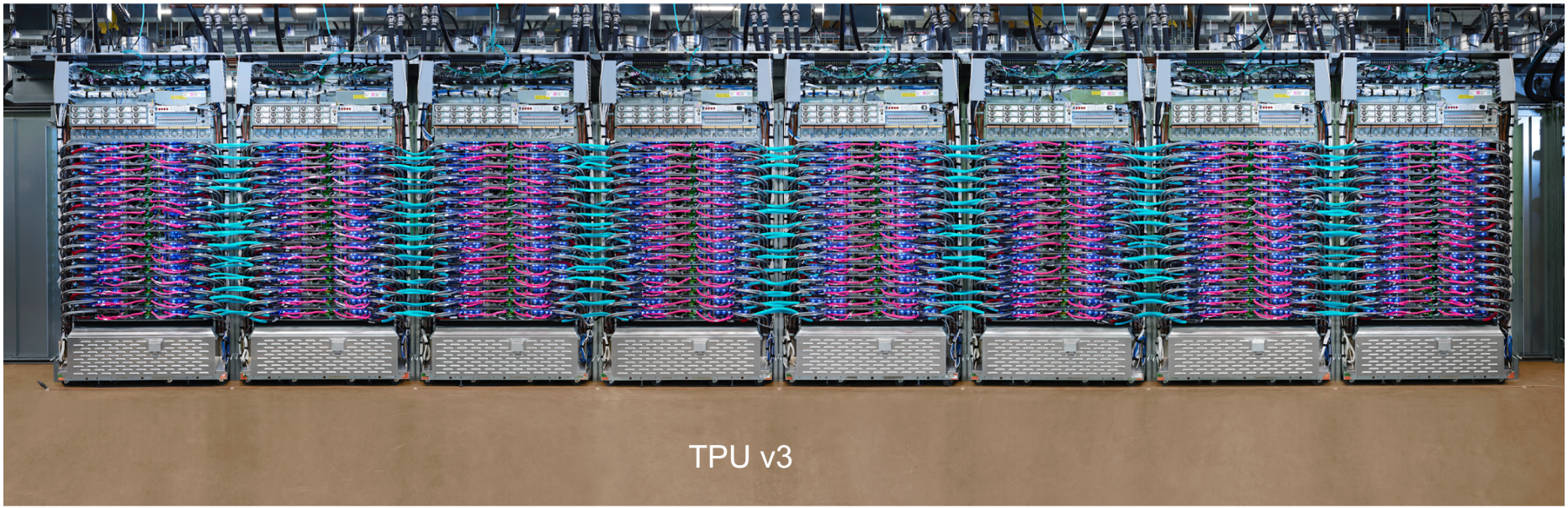
TPU Pod

64 2nd-gen TPUs

11.5 petaflops

4 terabytes of HBM memory

TPUv3



Distributed Training: What is it? & Why?

- **Distributed Training*** **~** Training across multiple devices
  - Different local and remote memory speeds / network
- Why do we need distributed training?
  - **Additional memory** (memory bandwidth) for larger model
    - “Need” to store weights + activations
  - Faster training by leveraging **parallel computation**
  - Reduce or eliminate **data movement**
    - Privacy → Federated Learning
    - Limited bandwidth to edge devices

*Very simplified definition.

Large-scale Distributed Training Example

SUMMIT Super Computer:



- CPU: 2 x 16 Core IBM POWER9 (connected via dual NVLINK bricks, 25GB/s each side)
- GPU: 6 x NVIDIA Tesla V100
- RAM: 512 GB DDR4 memory
- Storage: 250 PB
- Connection: Dual-rail EDR InfiniBand network of 200 Gbit/s

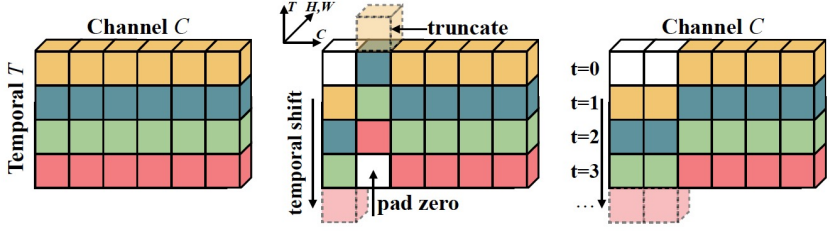
Large-scale Distributed Training Example

Train a Video Model (TSM) on 660 Hours of Videos

Models	Training Time	Accuracy
1 SUMMIT Nodes (6 GPUs)	49h 50min	74.1%
128 SUMMIT Nodes (768 GPUs)	28min	74.1%
256 SUMMIT Nodes (1536 GPUs)	14min (**211x**)	74.0%

Speedup the training by 200x, **from 2 days to 14minutes.**

- **Model setup**: 8-frame ResNet-50 TSM for video recognition
- **Dataset**: Kinetics (240k training videos) x 100 epoch

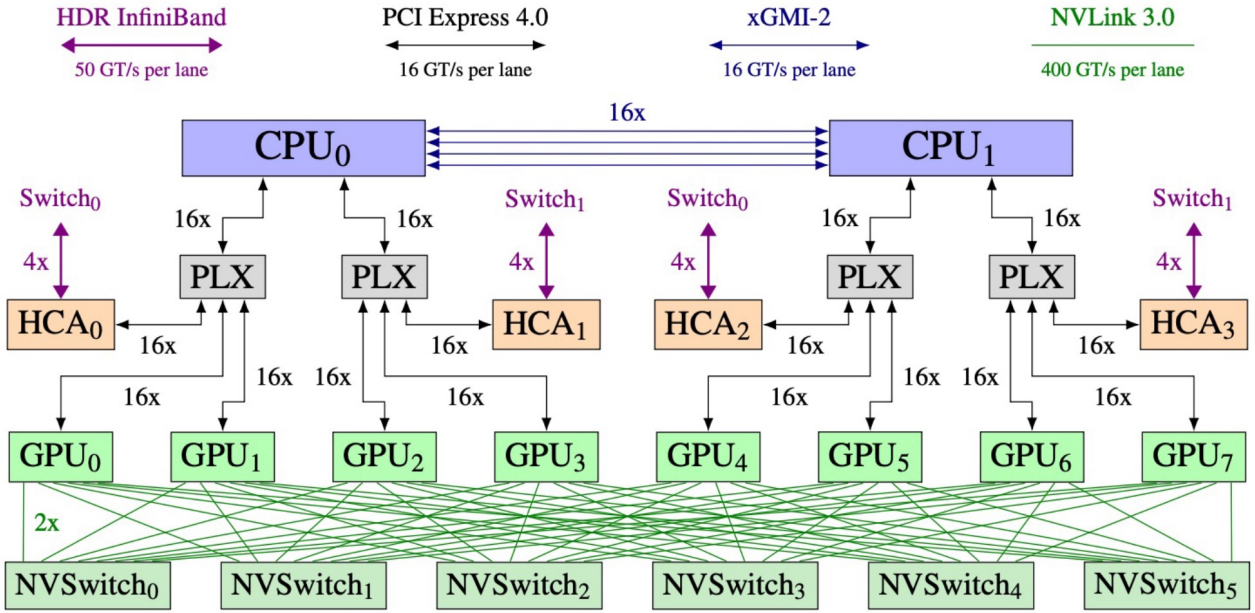


TSM: Temporal Shift Module for Efficient Video Understanding [Lin 2019]

Nvidia-based Multi-GPU + Multi-Machine Parallelism

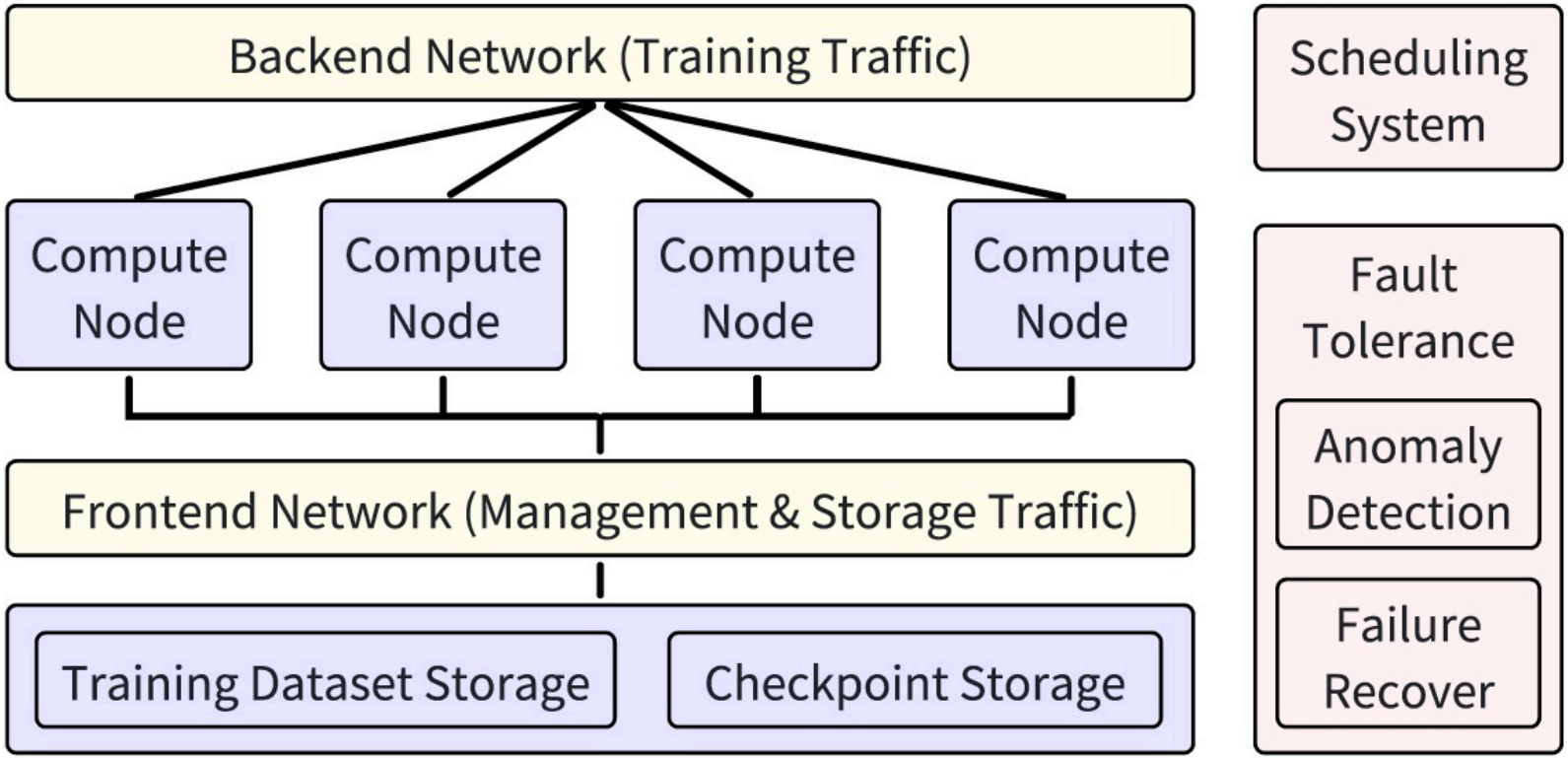
Intra-node parallelism
via high-speed interconnects

High-speed inter-node parallelism



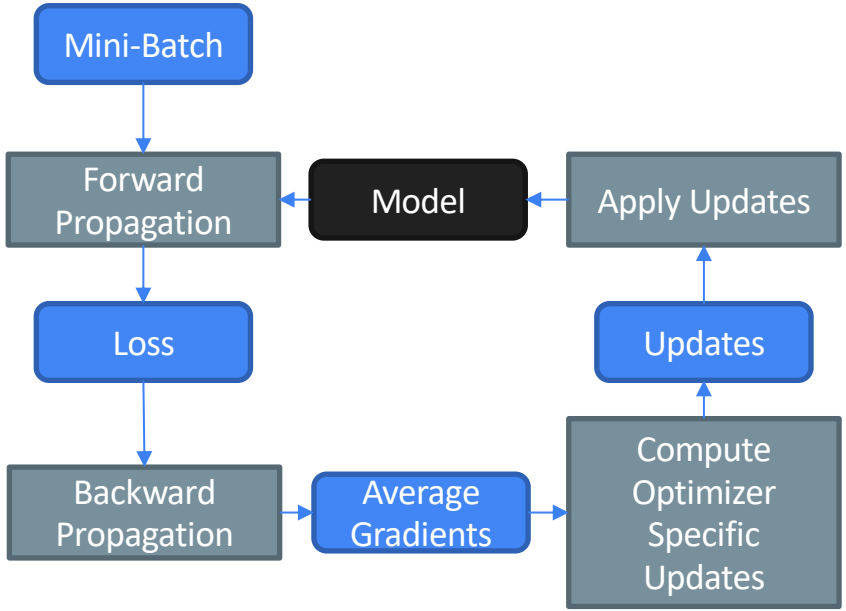
Split up memory and compute requirements across GPUs and machines

Infrastructure Overview for Distributed LLM Training

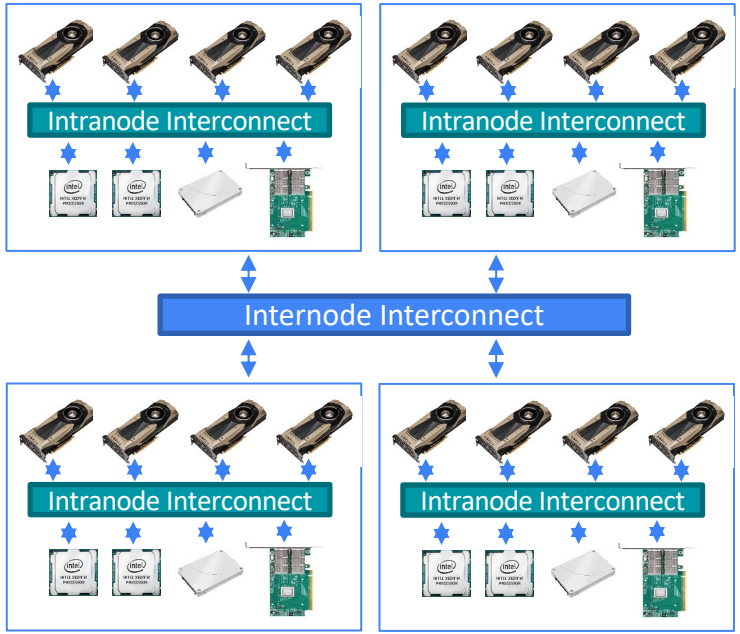


Source: Jiangfei Duan et al, “Efficient Training of LLMs on Distributed Infrastructures: A Survey”, July 2024

Distributed Data Parallel Training Overview



Data Parallel Training Loop



Distributed GPU Cluster

Parallelization Methods in Distributed Training for DNNs

Important Directions:

- Data Parallelism:
  - Naïve Data Parallel

- Further overcoming GPU Memory Bottleneck via Zero Redundancy Optimizer:
  - ZeRO Stage 1, 2, 3, ZeRO-Offload and ZeRO-Infinity (from Microsoft Research) implemented by:
    - DeepSpeed

    and
    - Fully Sharded Data Parallel (FSDP) in Pytorch (Meta)

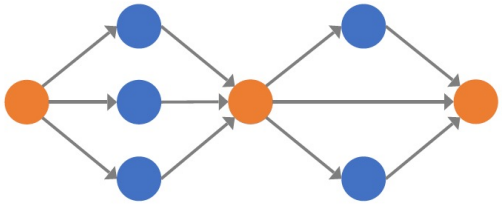
- Model Parallelism:
  - Pipeline Parallel: e.g. Gpipe, Pipedream, Nvidia's Megatron-PP
  - Tensor Parallel: e.g. Nvidia's Megatron-TP

- Sequence (aka Context) Parallelism:
  - Sequence Parallel

Plus:

- Activation Memory management: Selective Checkpointing and Recomputation
- Automatic/ Optimal Operator-Fusion and Model distribution support:
  - e.g. Alpa, Tensorflow (via XLA when training with TPU)

Data Parallelism in Distributed Training (1/3)

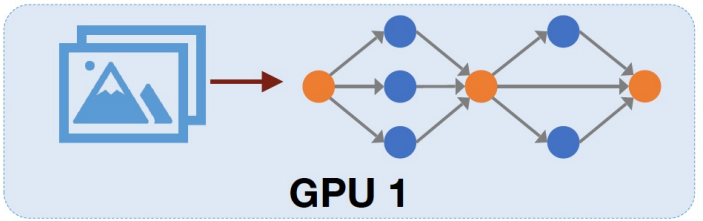


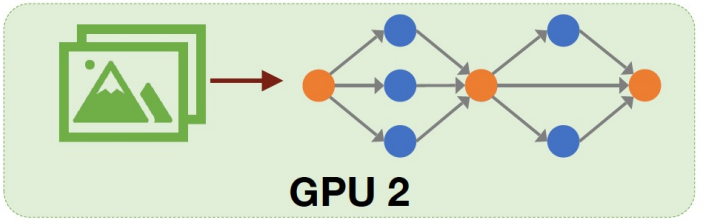
ML Model



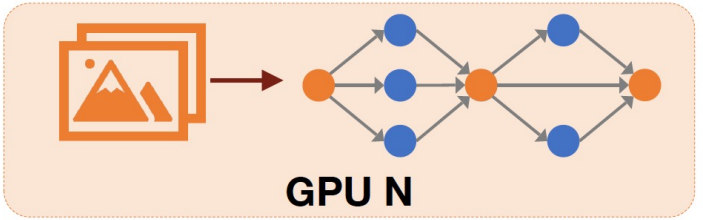
Training Dataset

Data Parallelism

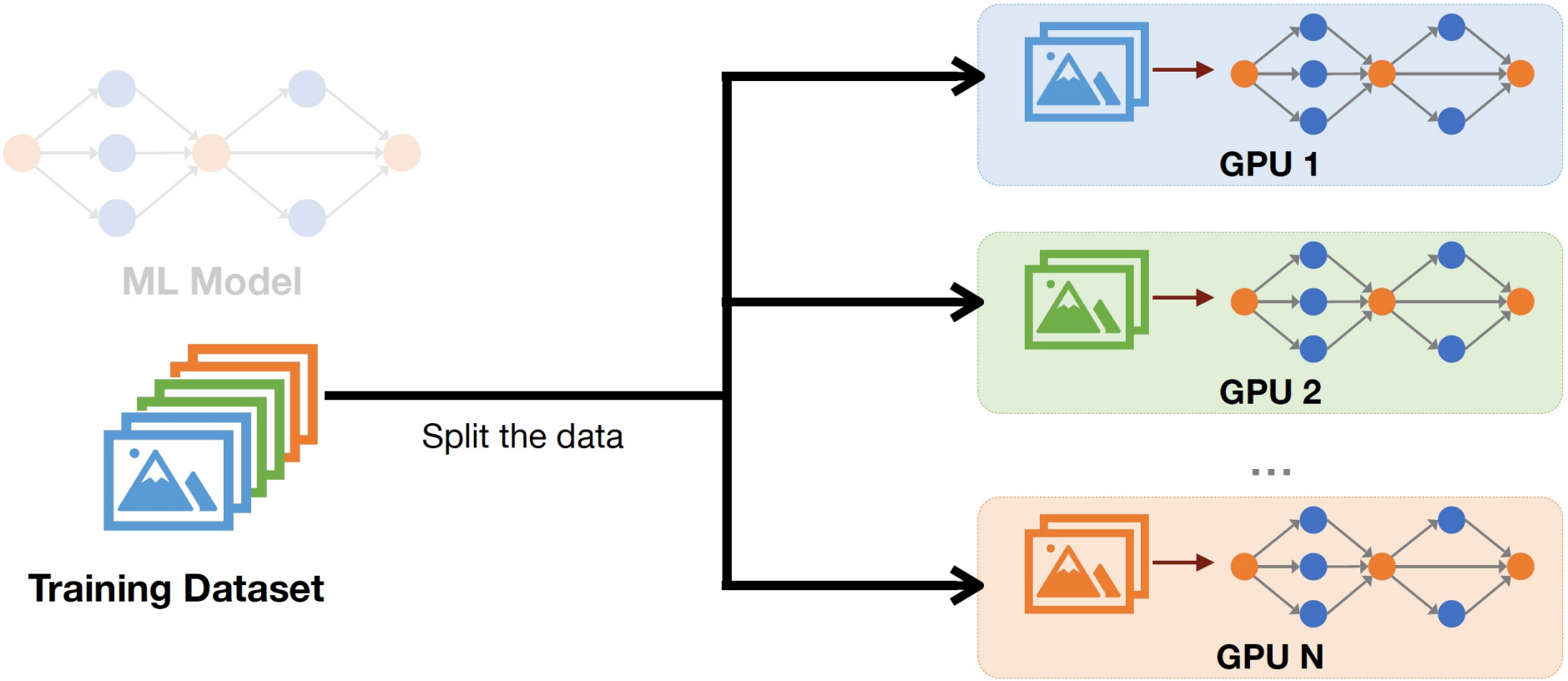




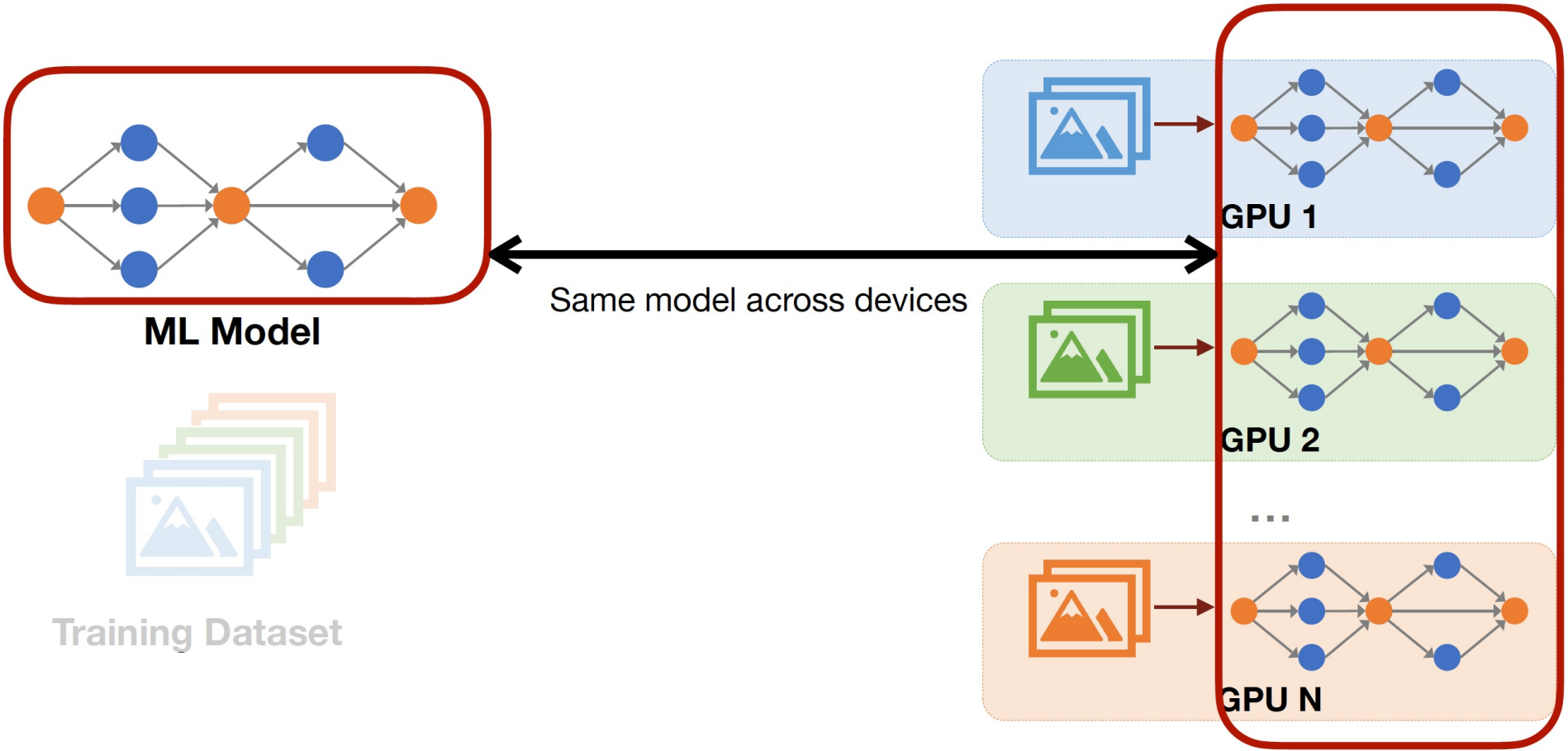
...



Data Parallelism in Distributed Training (2/3)



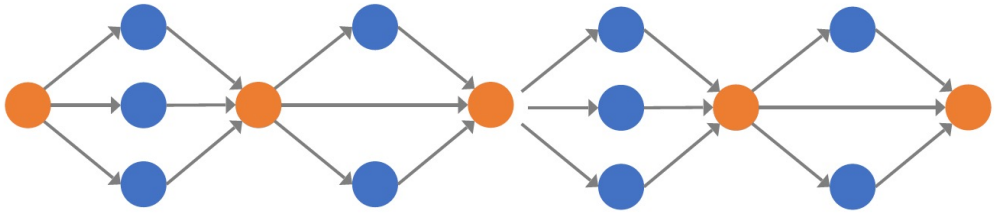
Data Parallelism in Distributed Training (3/3)



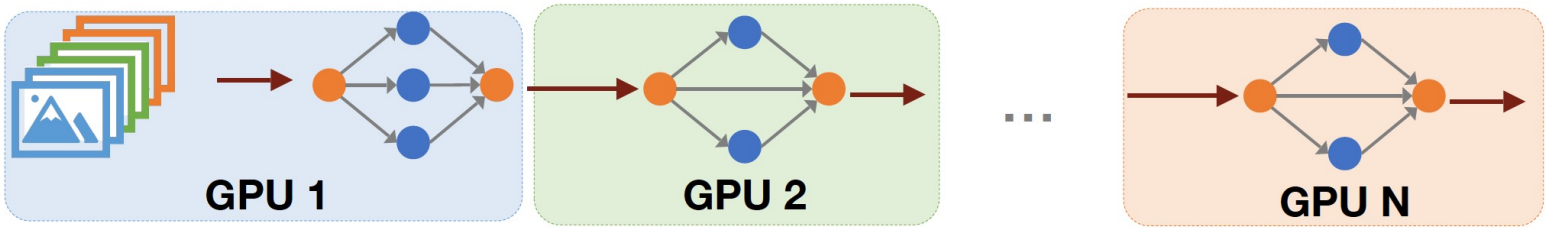
Pipeline Parallelism in Distributed Training (1/4)



Training Dataset

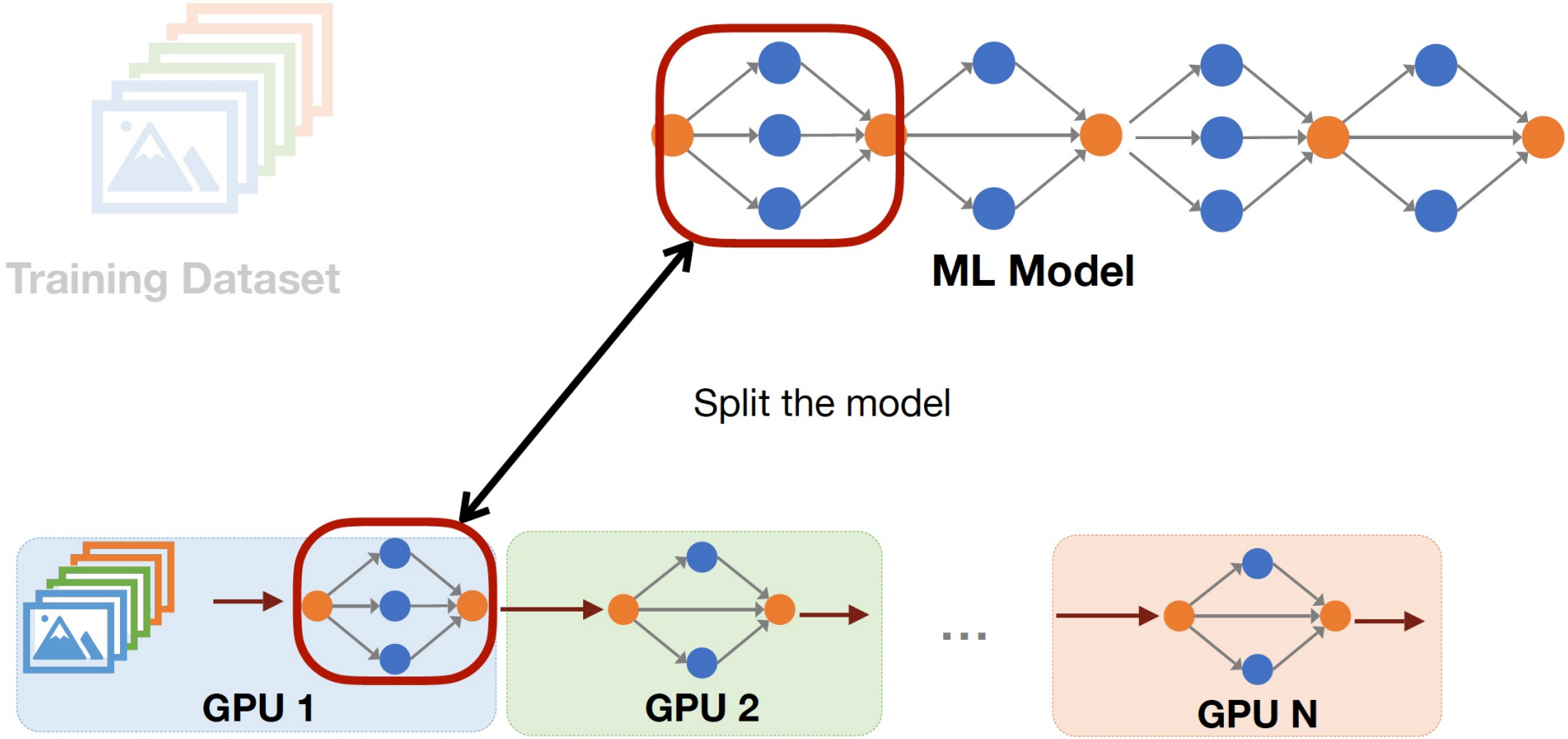


ML Model



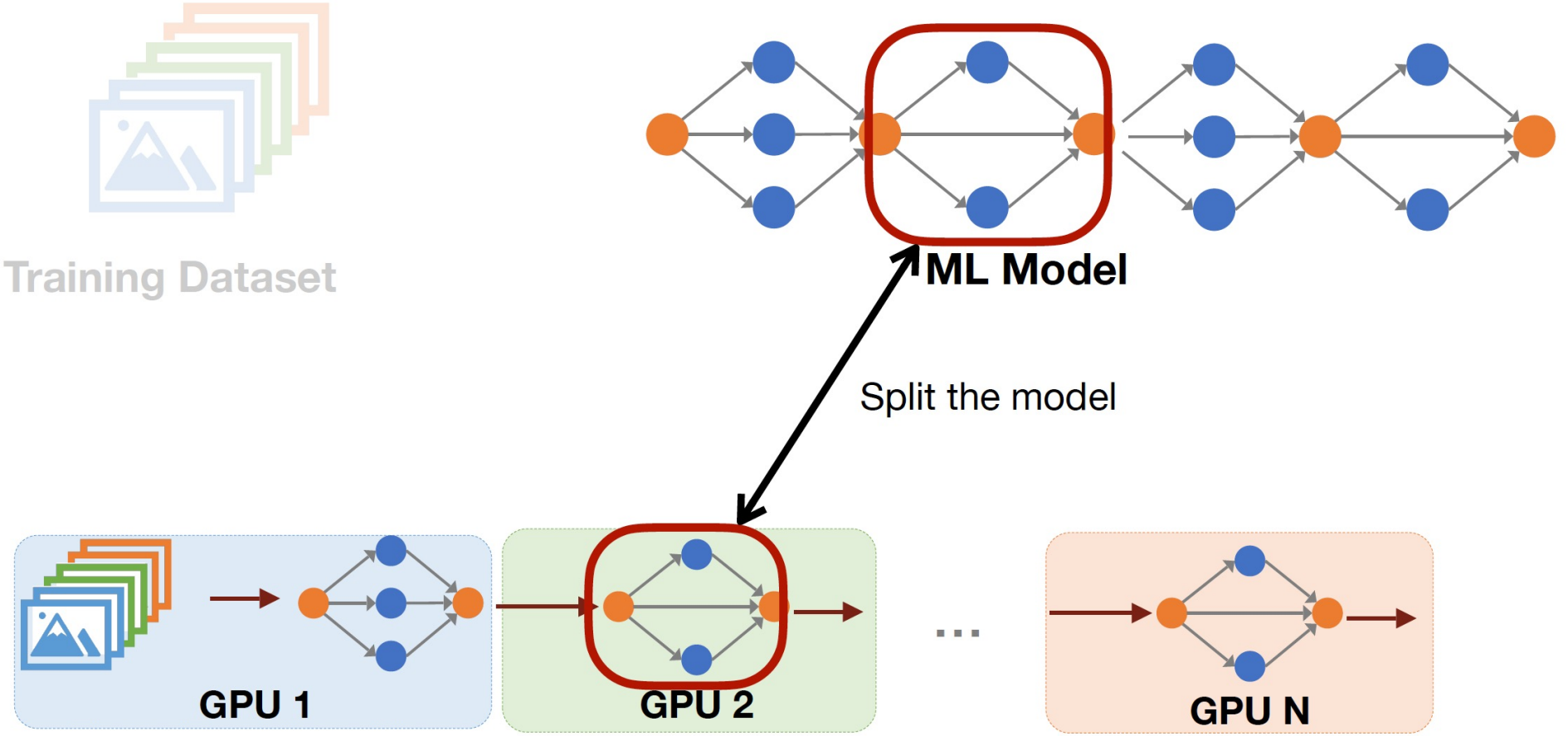
Figures credit: from CMU 15-849 [Jia 2022]

Pipeline Parallelism in Distributed Training (2/4)



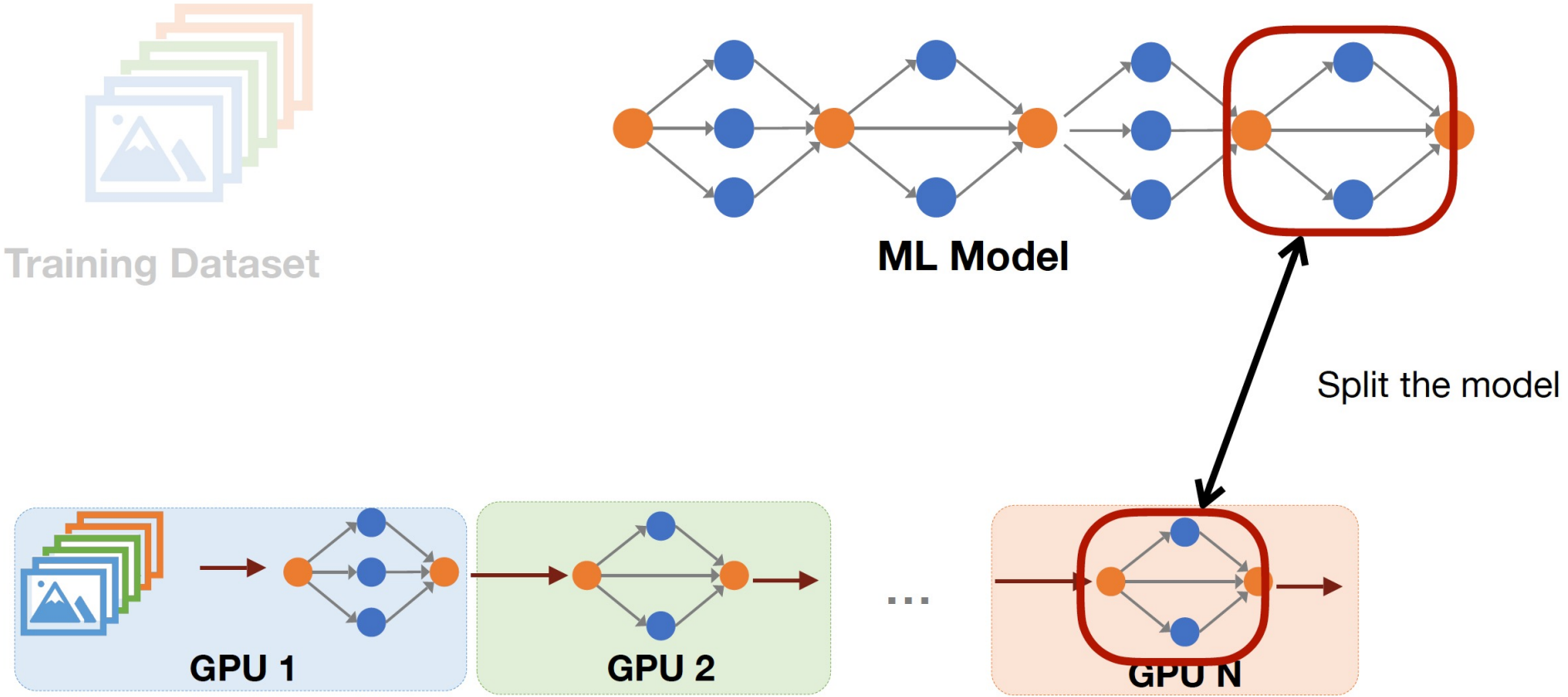
Figures credit: from CMU 15-849 [Jia 2022]

Pipeline Parallelism in Distributed Training (3/4)



Figures credit: from CMU 15-849 [Jia 2022]

Pipeline Parallelism in Distributed Training (4/4)

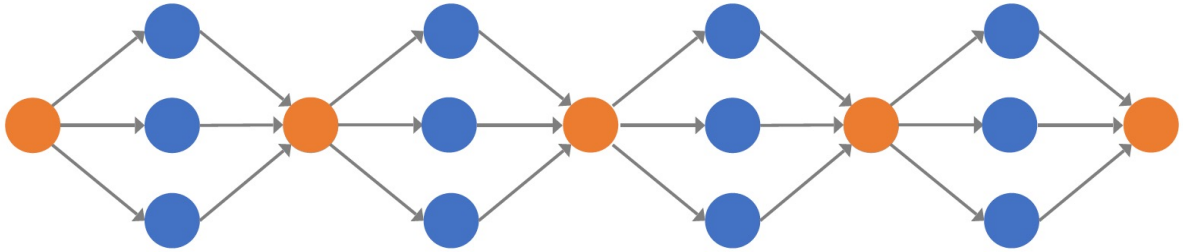


Figures credit: from CMU 15-849 [Jia 2022]

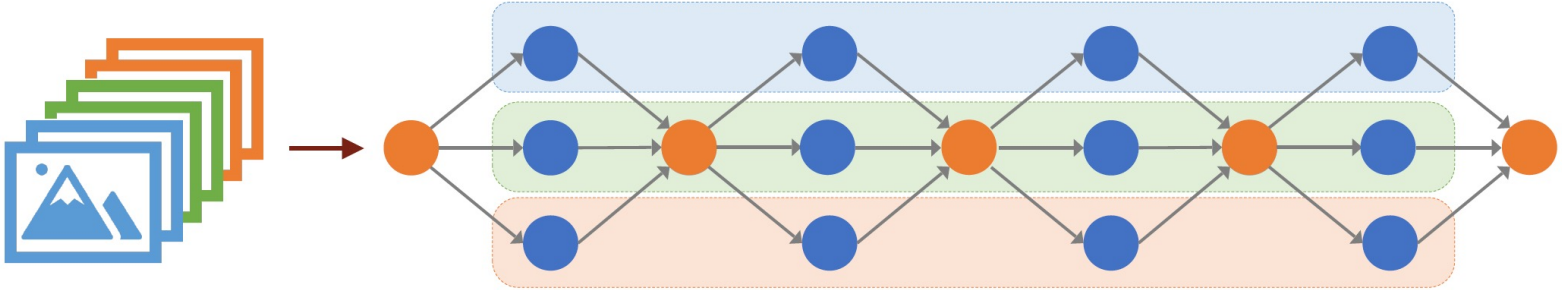
Tensor Parallelism in Distributed Training (1/6)



Training Dataset



ML Model



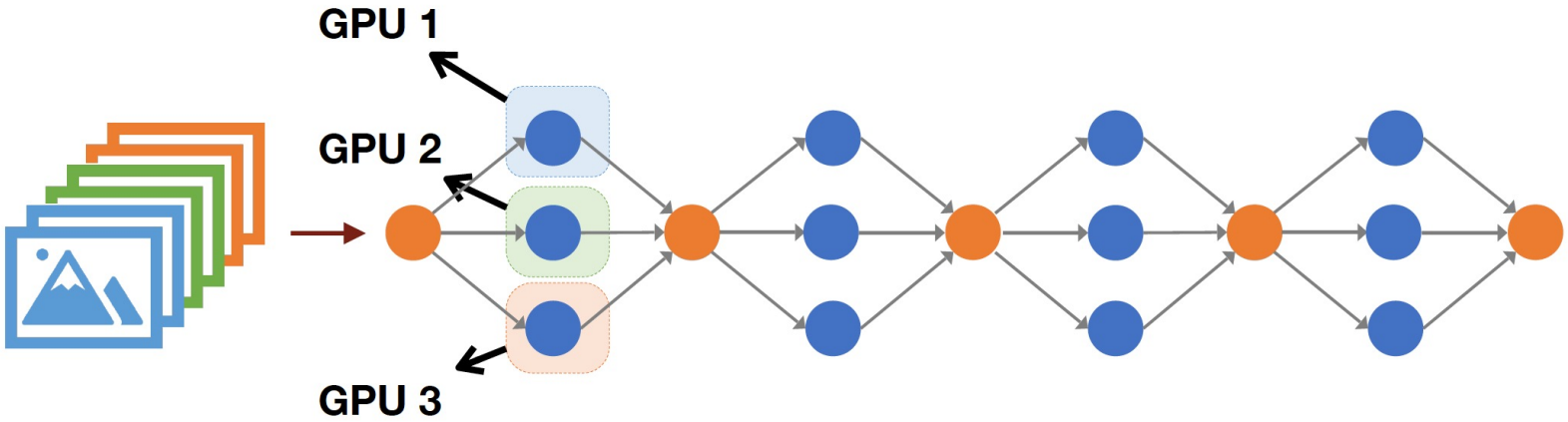
Figures credit: from CMU 15-849 [Jia 2022]

Tensor Parallelism in Distributed Training (2/6)

Training Dataset

ML Model

Split the activations then sync



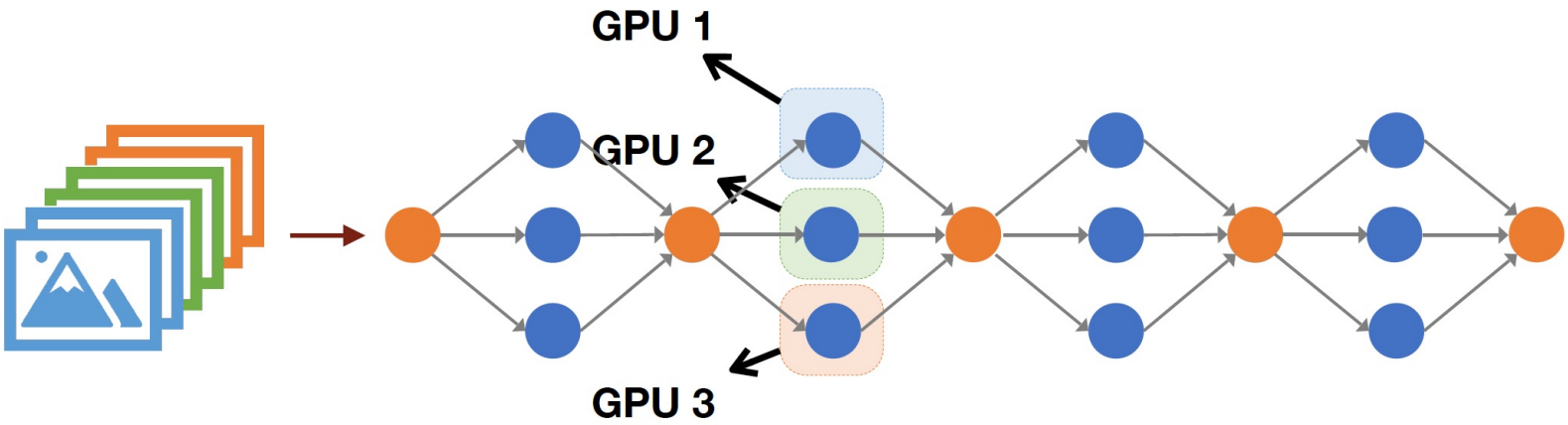
Figures credit: from CMU 15-849 [Jia 2022]

Tensor Parallelism in Distributed Training (3/6)

Training Dataset

ML Model

Split the activations then sync



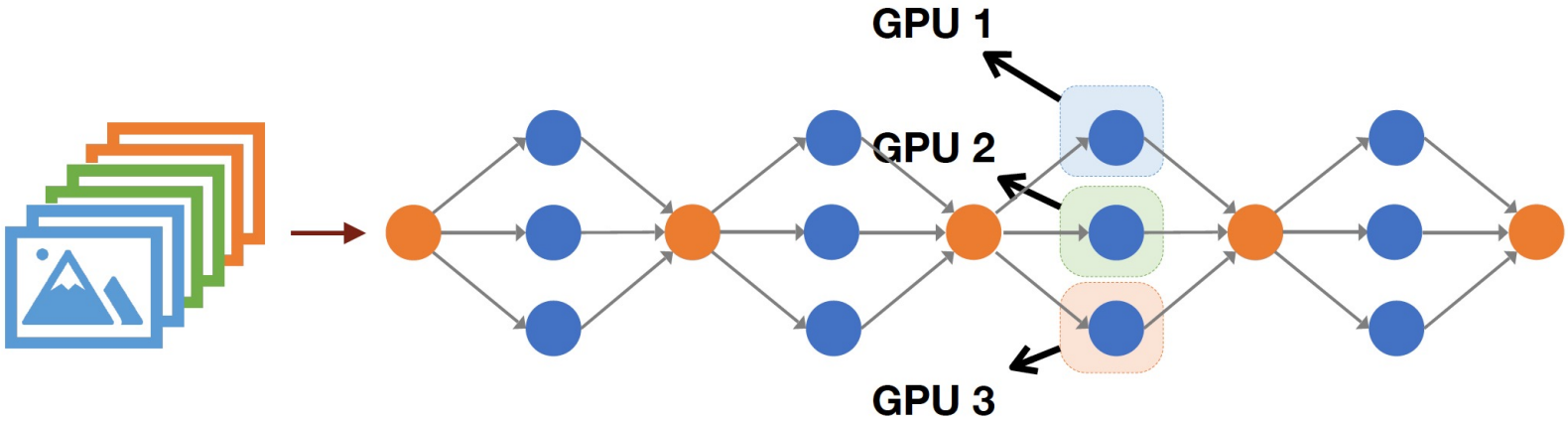
Figures credit: from CMU 15-849 [Jia 2022]

Tensor Parallelism in Distributed Training (4/6)

Training Dataset

ML Model

Split the activations then sync



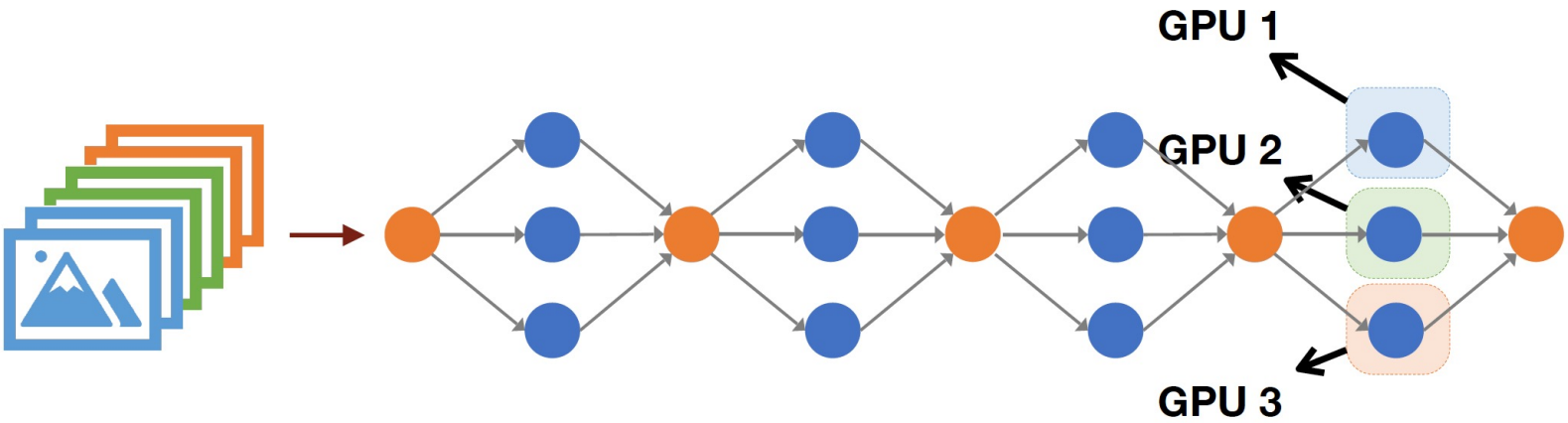
Figures credit: from CMU 15-849 [Jia 2022]

Tensor Parallelism in Distributed Training (5/6)

Training Dataset

ML Model

Split the activations then sync

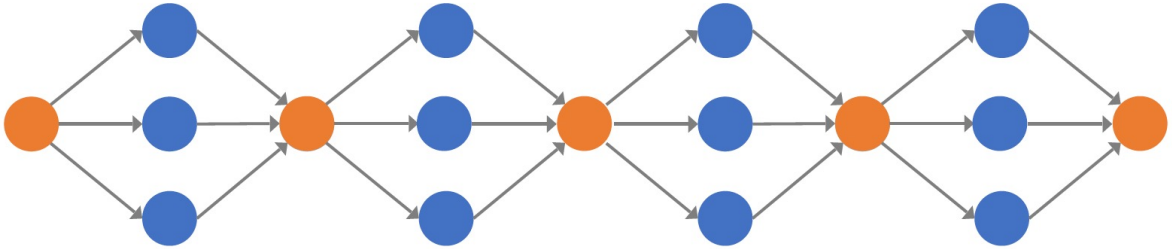


Figures credit: from CMU 15-849 [Jia 2022]

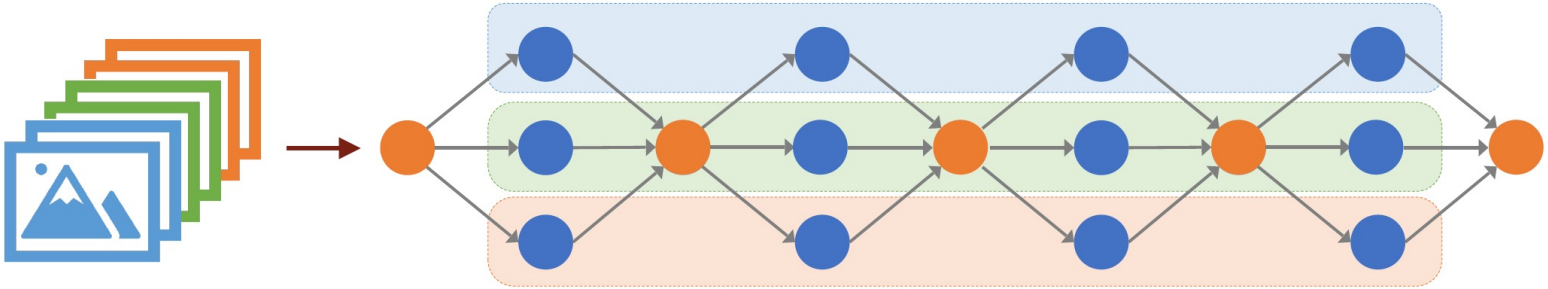
Tensor Parallelism in Distributed Training (6/6)



Training Dataset



ML Model



Figures credit: from CMU 15-849 [Jia 2022]

Sequence/ Context Parallelism in Distributed Training (1/2)

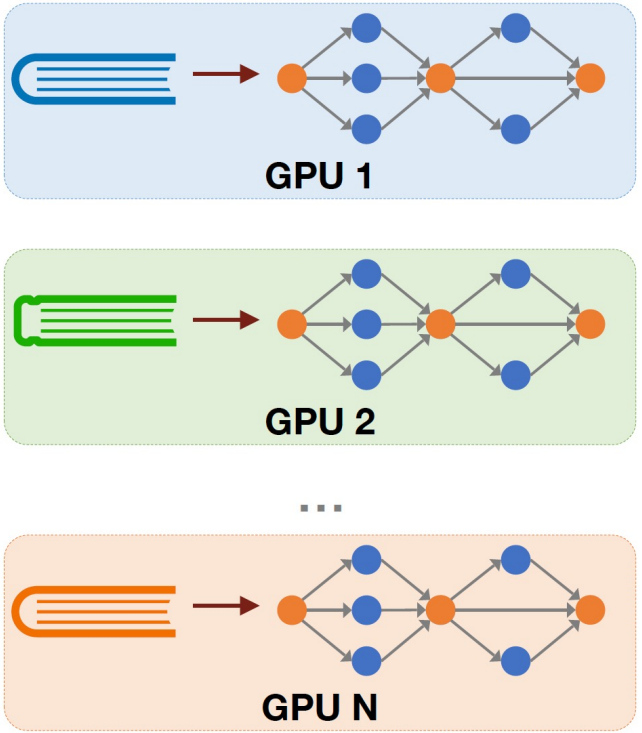


Long Text for Training

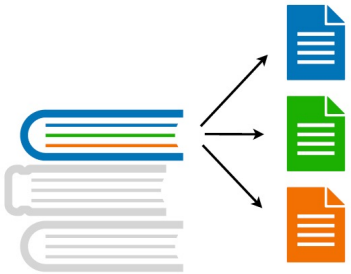
[**Batch Size**, Tokens, Hidden Dimension]

↑

Data Parallelism
Splits Batch



Sequence/ Context Parallelism in Distributed Training (2/2)



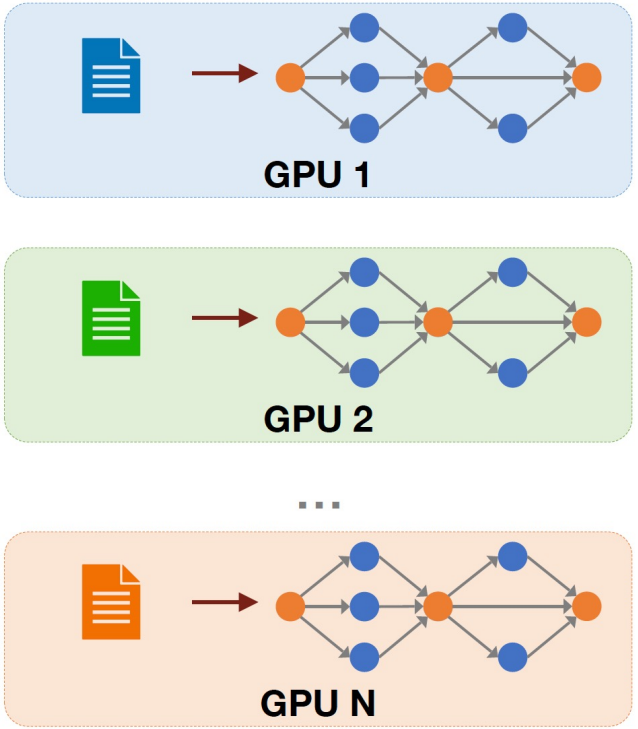
Long Text for Training

can be >100K

[Batch Size, **Tokens**, Hidden Dimension]

↑

Sequence Parallelism
Splits Tokens

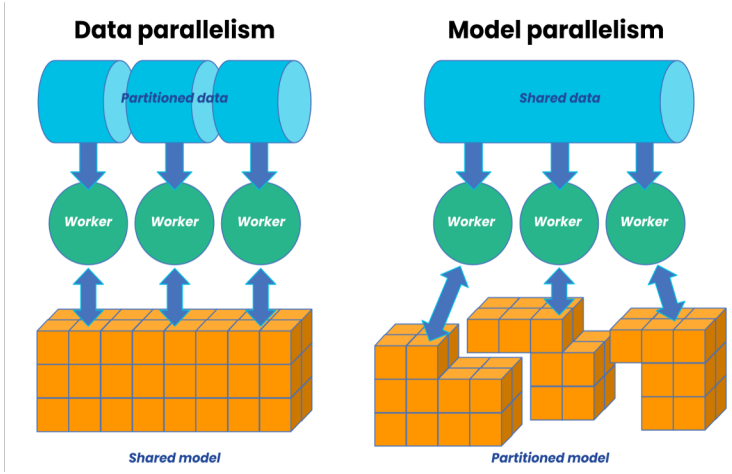


More Terminologies regarding Parallelism

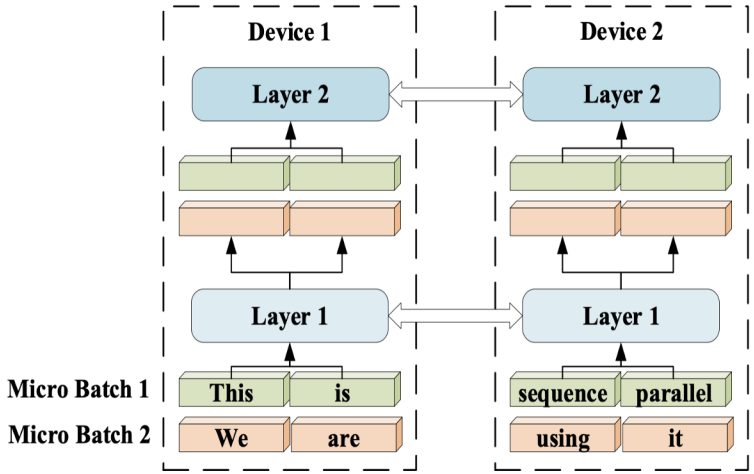
The so-called “4D” Parallelism

4D Parallelism to minimize bottlenecks and maximizes efficiency which combines: [1] Data, [2] Context, [3] Pipeline (Vertical), and [4] Tensor (Horizontal) Parallelism.

- **Data Parallelism** parallelizes tasks to speed up data processing and model iterations.
- **Context Parallelism** (aka Sequence Parallelism) splits input sequences into chunks to be processed separately.



K. Pijanowski and M. Galarnyk. What is Distributed Training? 2022.

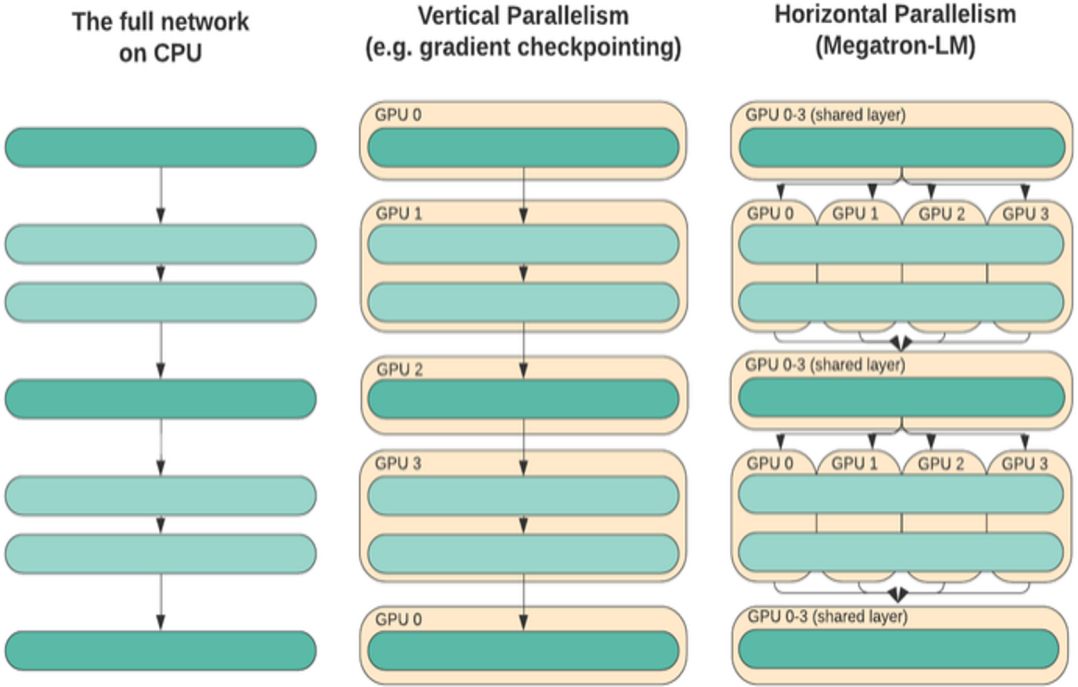


S. Li et al. Sequence Parallelism: Long Sequence Training from System Perspective. 2021.

More confusing terminology on Parallelism

Model Parallelism include:

- **Pipeline Parallelism** separates a model based on its layers, allowing higher throughput.
- **Tensor Parallelism** splits matrices across GPUs to reduce peak memory consumption.



Technical Details on

Distributed/ Parallel Training Approaches for Large Deep Neural Networks

Ideal Metric of Success for Efficient Training

$$\left(\frac{\text{“Learning”}}{\text{Second}}\right) = \left(\frac{\text{“Learning”}}{\text{Record}}\right) \times \left(\frac{\text{Record}}{\text{Second}}\right)$$

Convergence
Machine Learning
Property

Throughput
System
Property

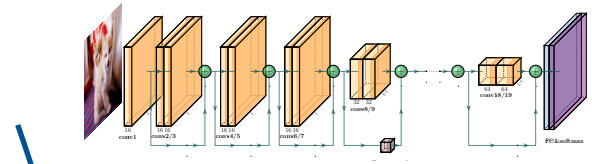
**Somewhat of a simplistic linear model. As we will later see there are many more moving parts to this*

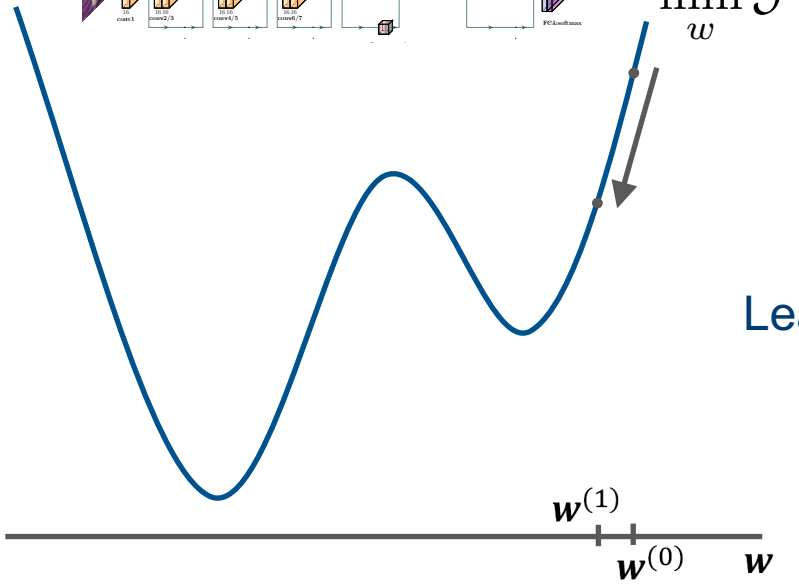
Metrics of Success

- Minimize training time to *“best model”*
  - Best model measured in terms of test error
- Other Concerns?
  - **Complexity**: *Does the approach introduce additional training complexity (e.g., hyper-parameters)*
  - **Stability**: *How consistently does the system train the model?*
  - **Cost:** *Will obtaining a faster solution cost more money (power)?*

Gradient Descent – Process ALL data-points per step

=> Batch-size = N (aka Full-Batched GD)





$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \alpha \underbrace{\frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

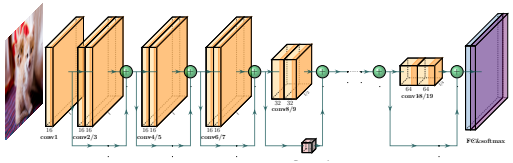
Learning rate

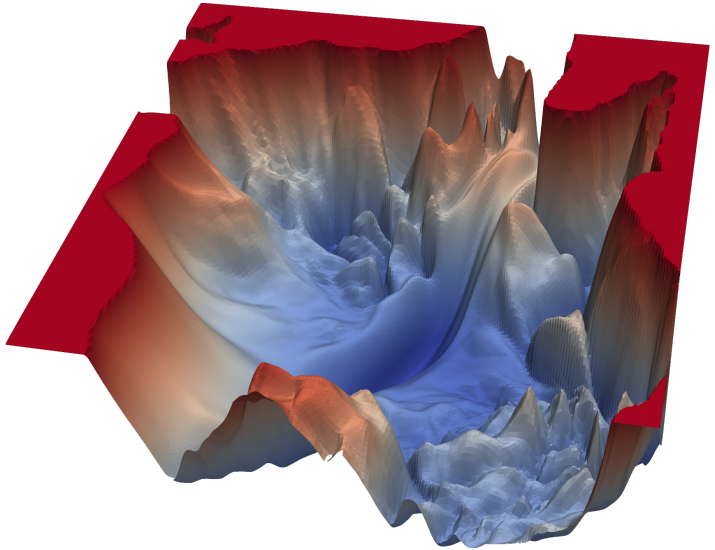
Two key elements:

- The computed gradient: the direction
- The learning rate: how big a step do we take?

Batched Gradient Descent (aka Mini-batched GD)
only process a subset of data points per step

(Typically B << N ; B=1 for the original “Stochastic GD”)





$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 - \underbrace{\frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

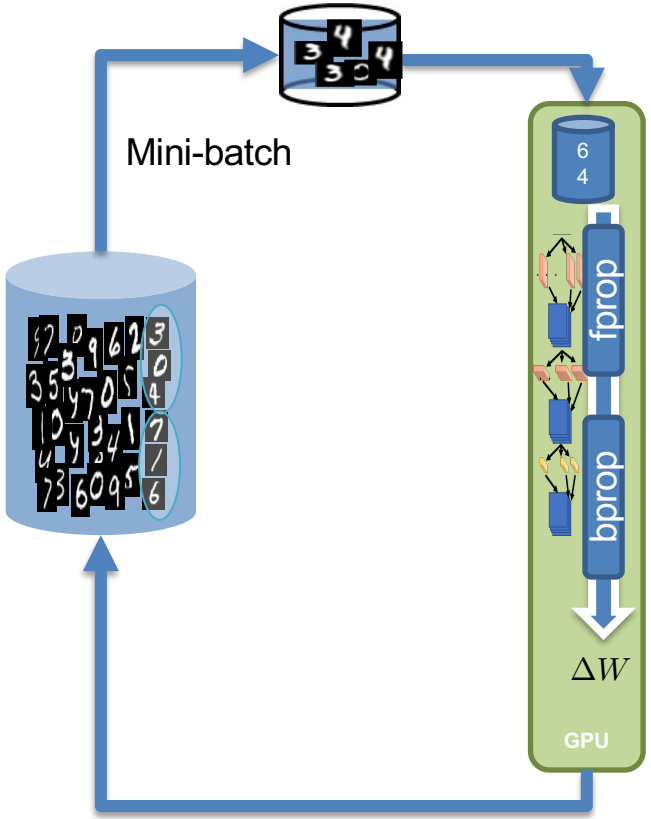
Learning rate

Two key elements:

- The computed gradient: the direction
- The learning rate: how big a step do we take?

Synchronous Stochastic Gradient Descent

In every iteration of SGD we load a **random mini-batch of training** data, and compute the gradient.



$$\min_w \mathcal{J}(w) = \frac{1}{N}\sum_{i=1}^{N} cost(w, x_i)$$

$$w^1 = w^0 \underbrace{- \frac{\alpha}{B}\sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}}_{\Delta w}$$

Parallelization Opportunities

Data Parallelism: *Distribute the processing of data to multiple Processing Elements (PEs).*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

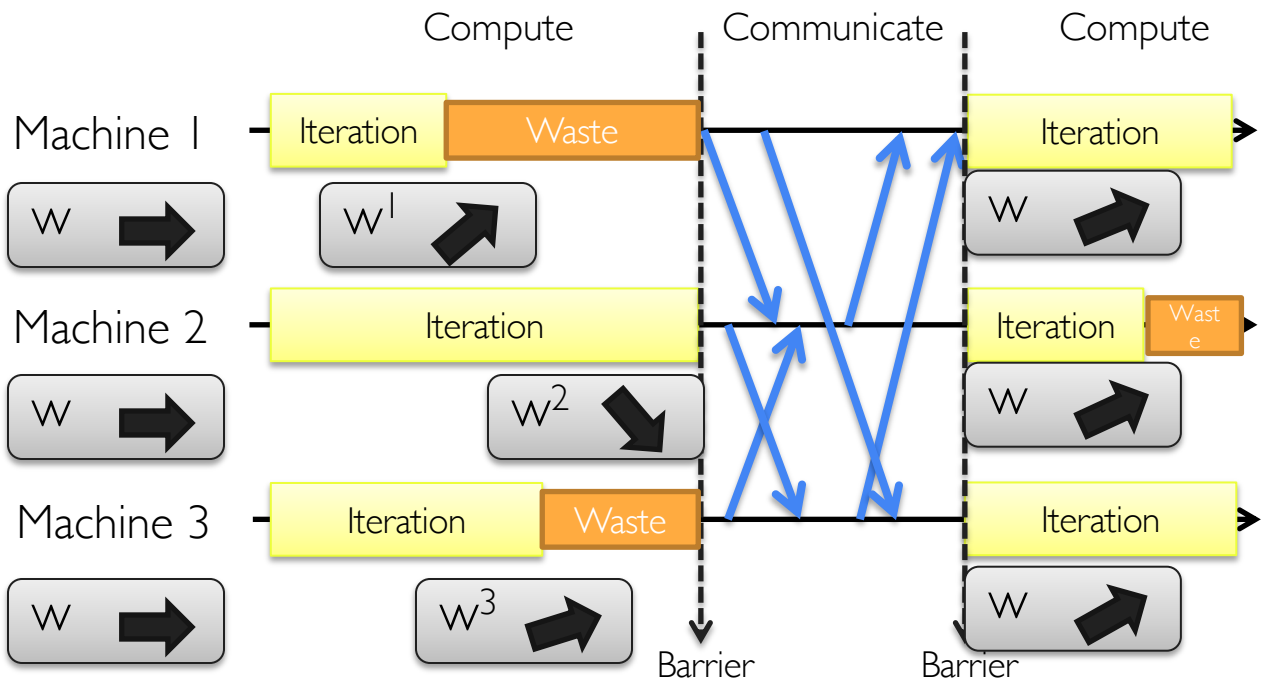
Model Parallelism: *Break the model and distribute processing of every layer to multiple PEs*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

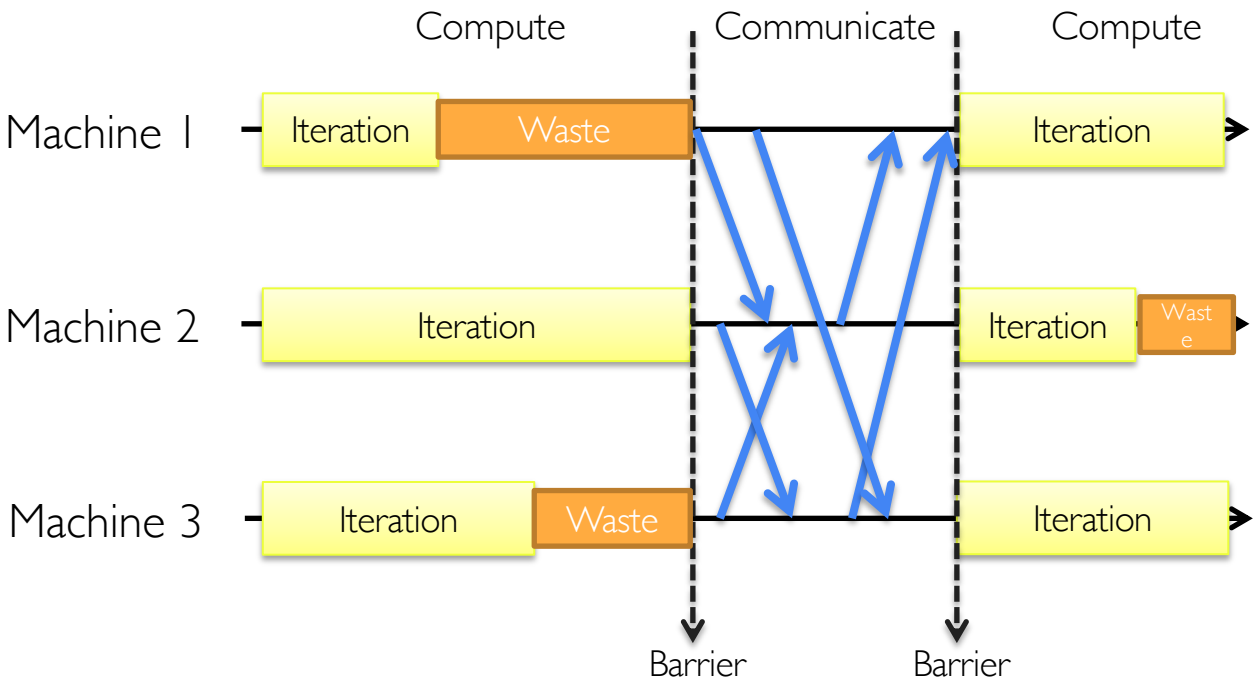
For either approach it is also possible to use ***synchronous*** *or* ***asynchronous*** *updates*

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$

Bulk Synchronous Parallel (BSP) Execution

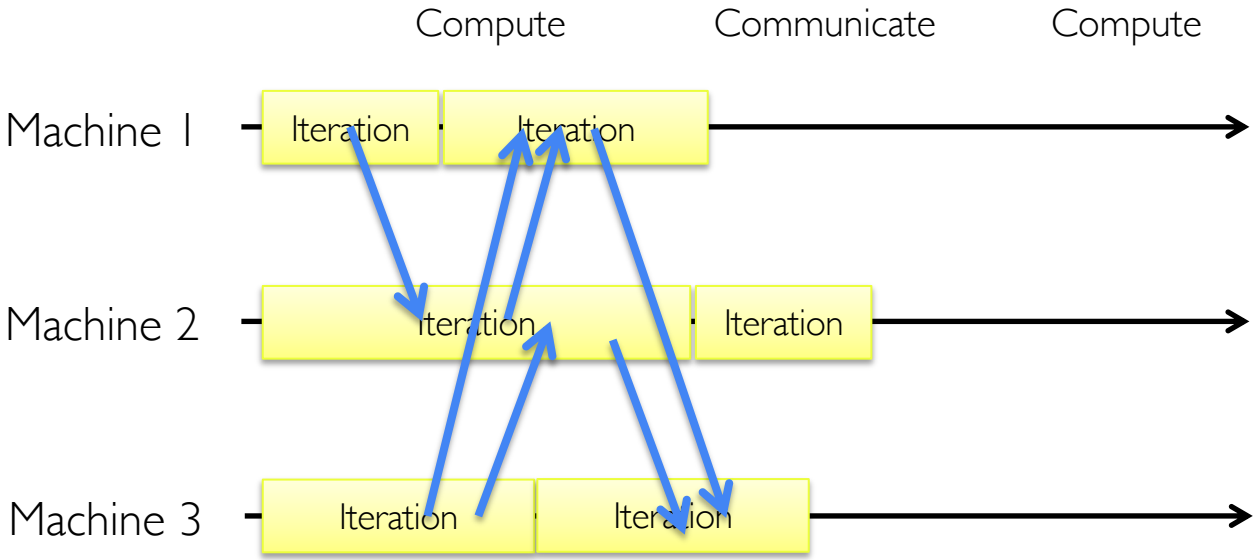


Bulk Synchronous Parallel (BSP) Execution



Enable more frequent coordination on parameter values

Asynchronous Execution



- Asynchronous Execution can enable more frequent coordination on parameter values, but often results in slower training loss convergence and worse generalization.

A Note on Asynchronous Methods for Distributed Training

- General advice: Training methods that adversely affect generalization are not adopted, unless there is a 10x speed improvement.

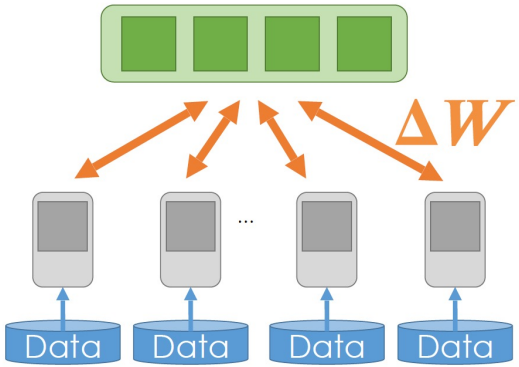
- As a result, Industrial-strength Frameworks to-date mostly adopt Synchronous Distributed Training.

- Otherwise, there are so many moving parts that can go wrong in training NNs, that most often practitioners stay away from async methods unless absolutely necessary:
  - For example of such exception: training very large Recommender Systems.

An Example: Realizing (Synchronized) Data Parallelism via the Parameter Server approach

Parameter Server

The central controller of the whole training process



Worker nodes

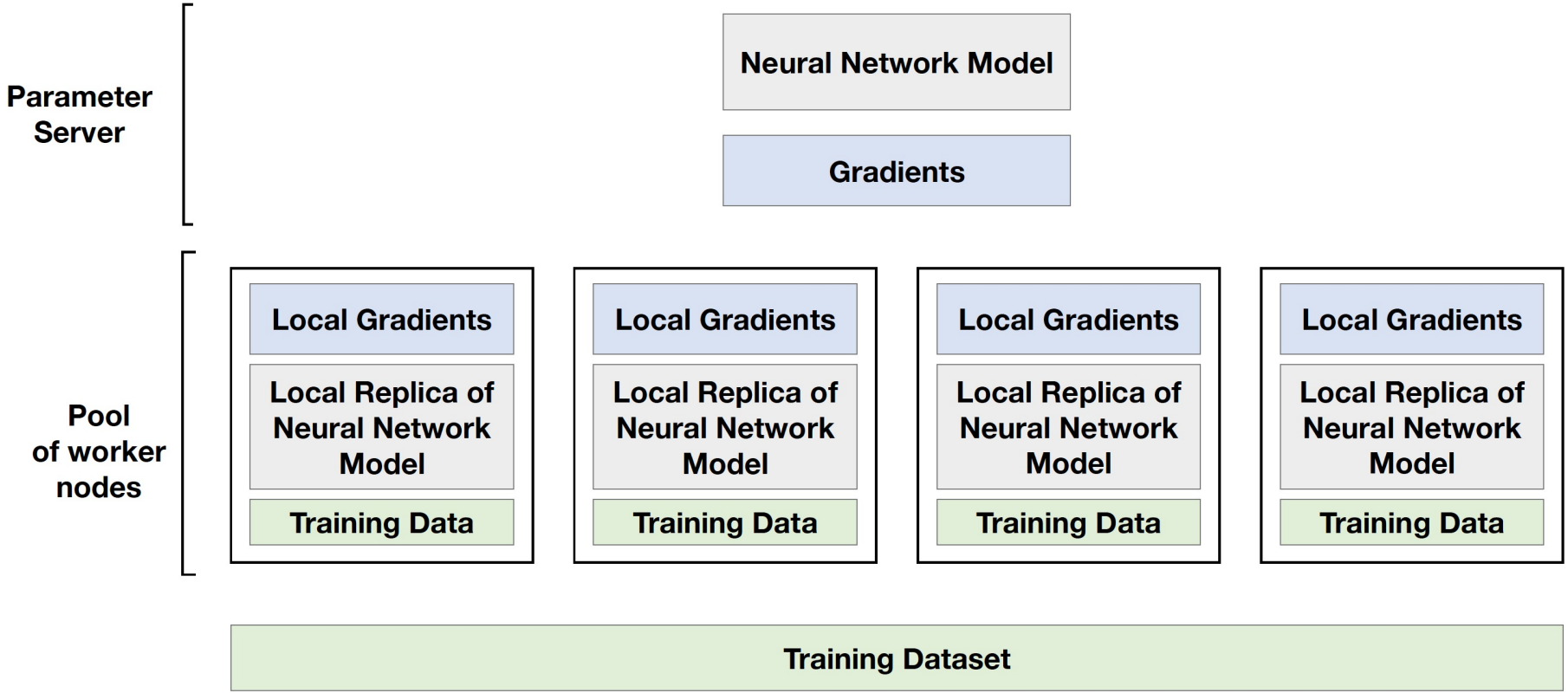
The hardware accelerators and dataset storage.

Two different roles in framework:

- Parameter Server: receive gradients from workers and send back the aggregated results
- Workers: compute gradients using splitted dataset and send to parameter server

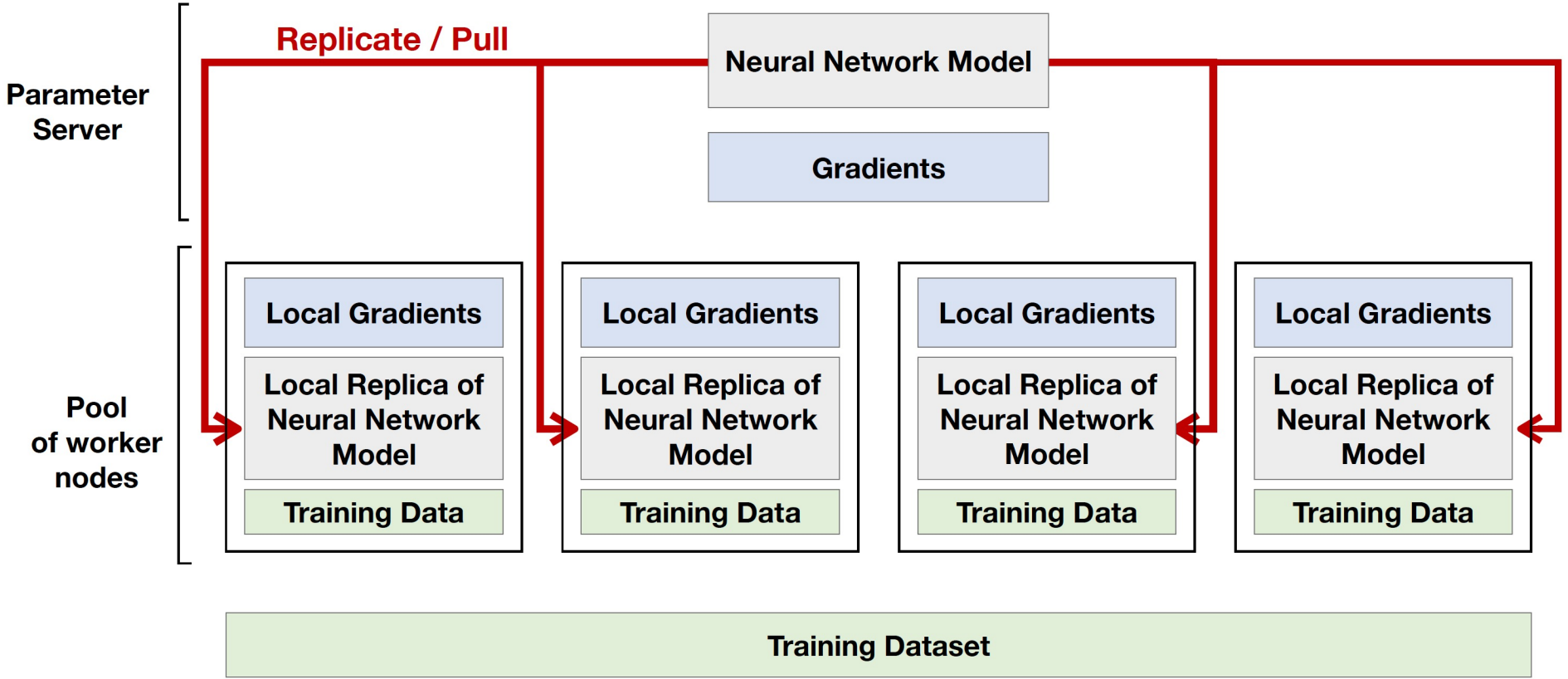
Scaling Distributed Machine Learning with the Parameter Server. Mu Li et al. 2014
Figure credits from: Deep Gradient Compression. Lin et al. 2018

Overview of the Infrastructure for the Parameter Server approach



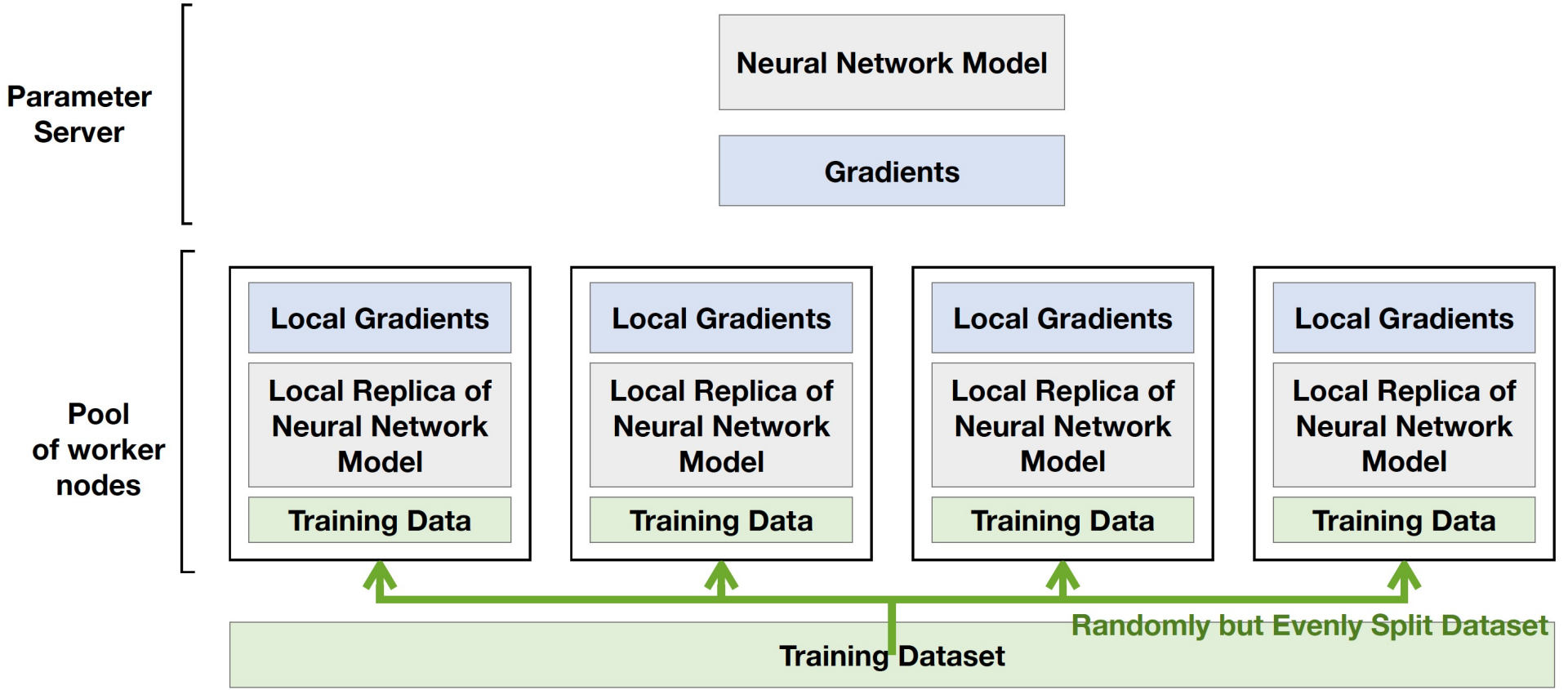
Synchronized Data Parallelism with the Parameter Server

1 - Replicate Models to Each Worker



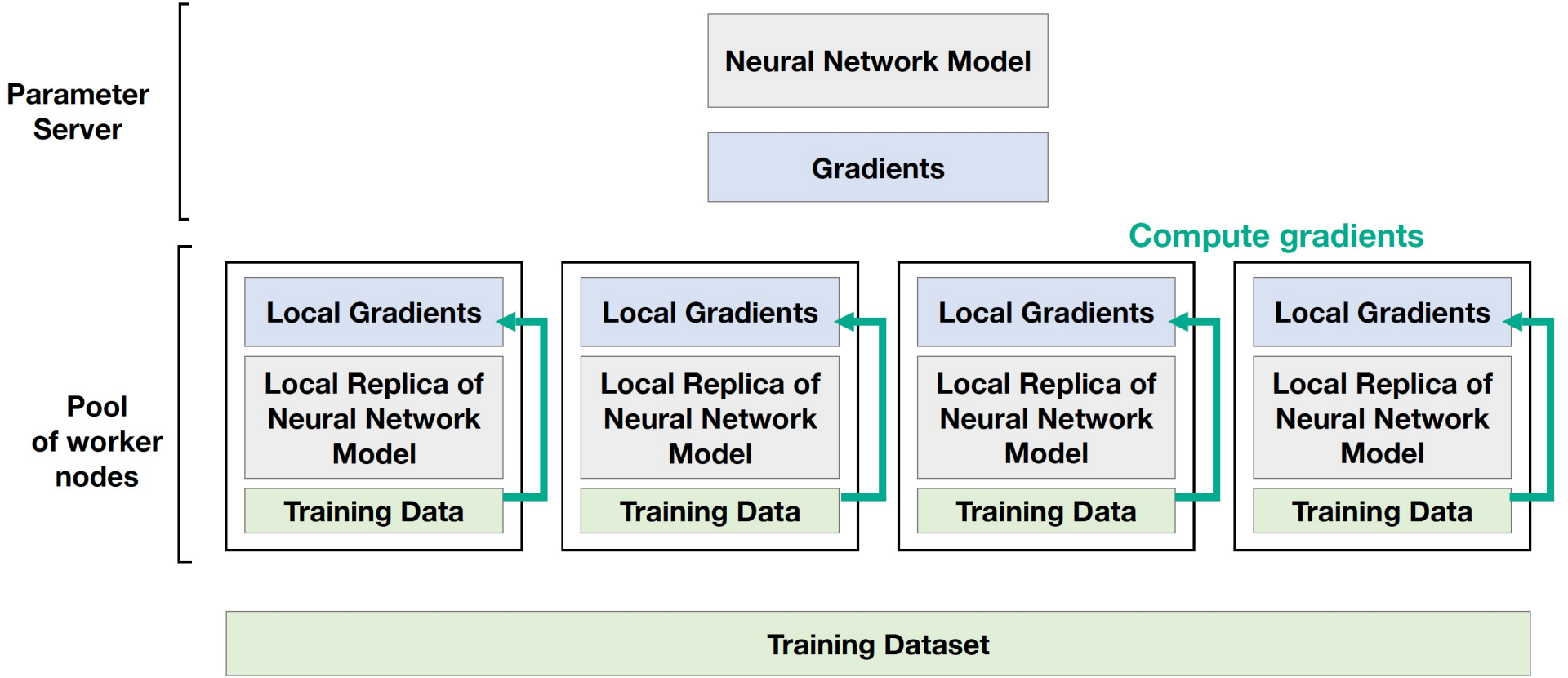
Synchronized Data Parallelism with the Parameter Server

2 - Split Training Data to Workers



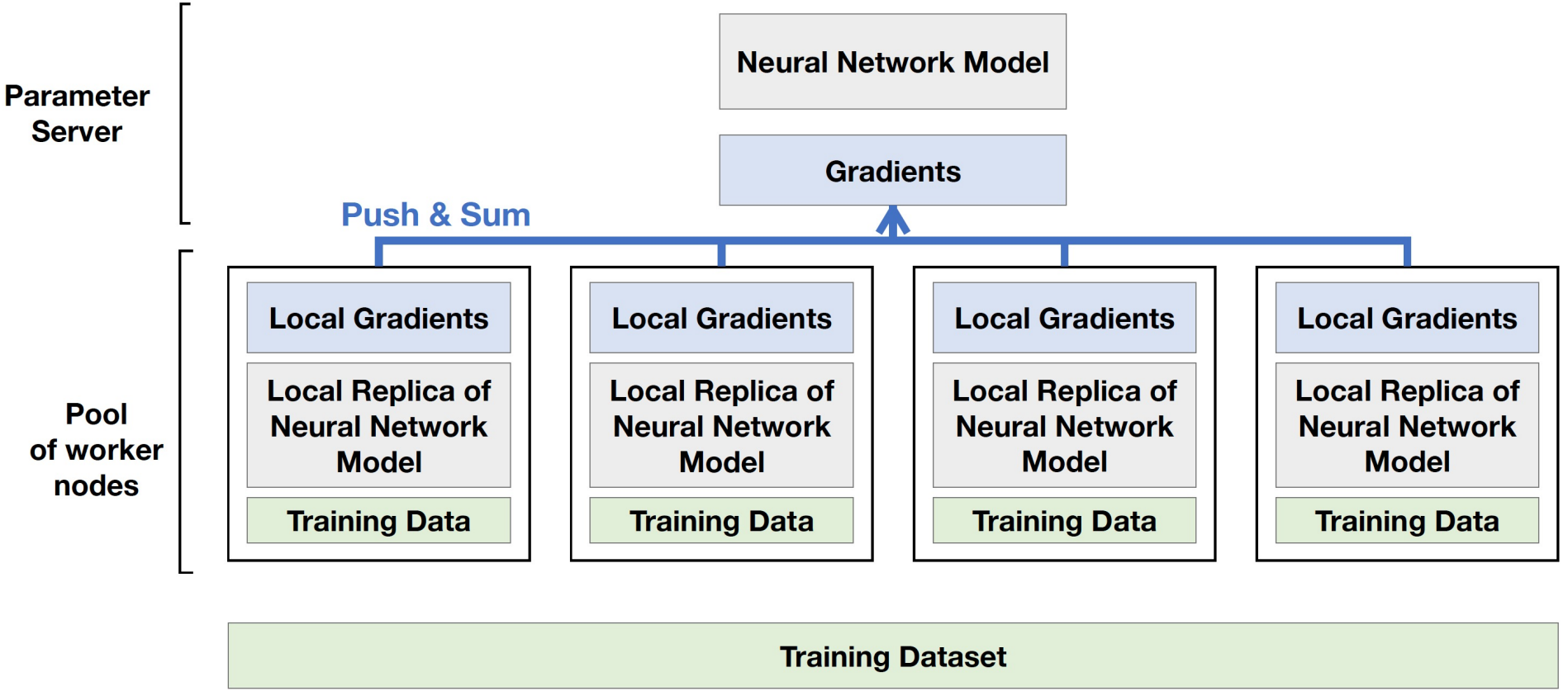
Synchronized Data Parallelism with the Parameter Server

3 - Compute gradients



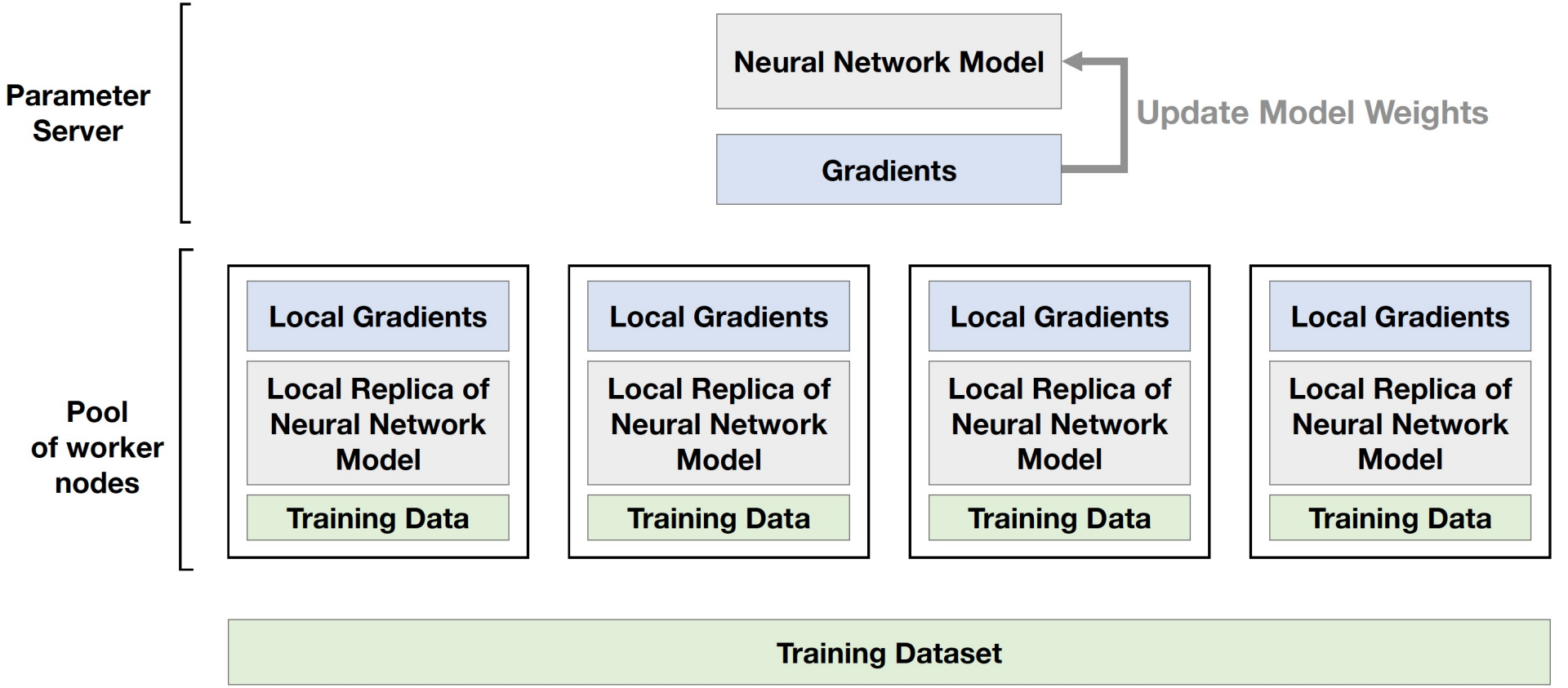
Synchronized Data Parallelism with the Parameter Server

4 - Aggregate and Synchronize Gradients

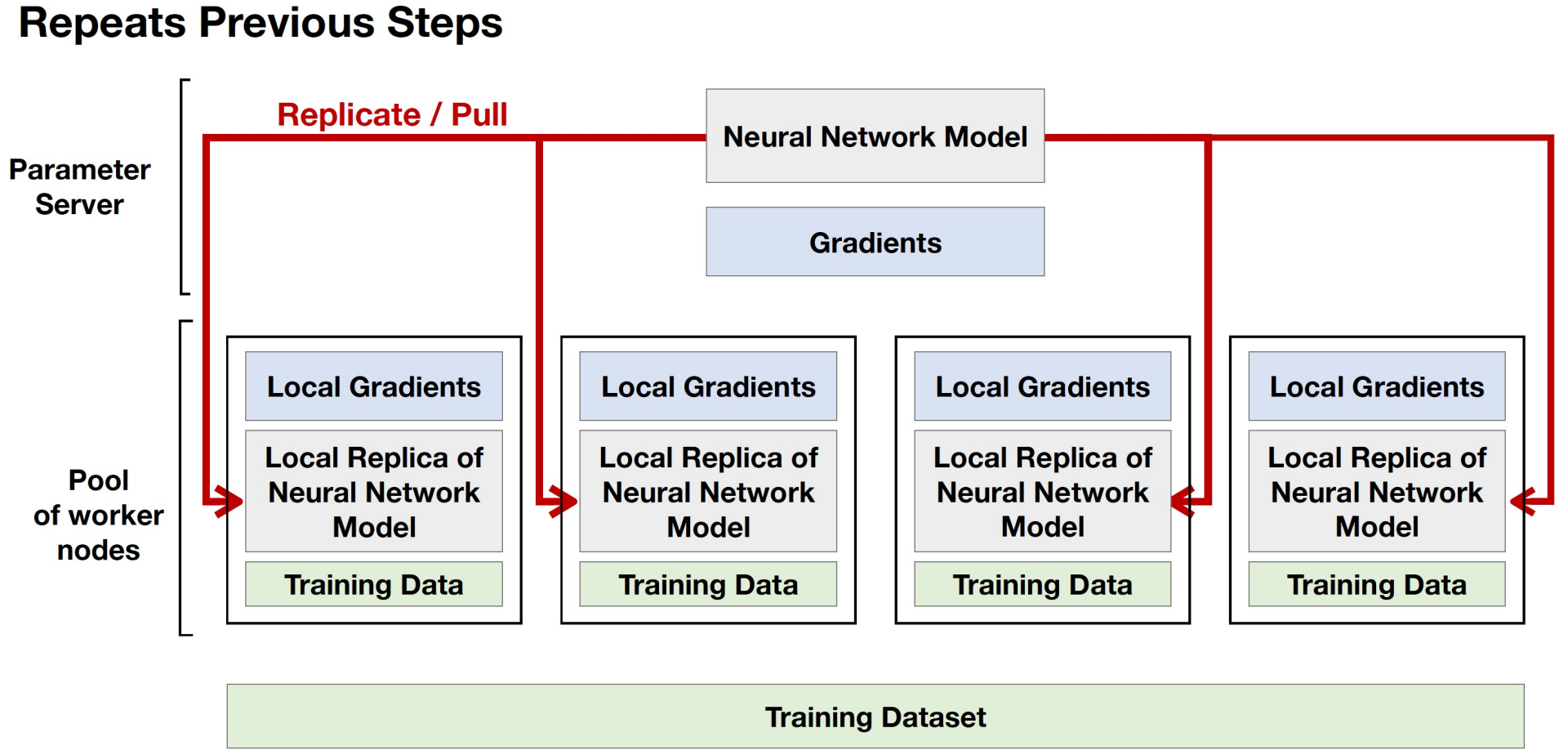


Synchronized Data Parallelism with the Parameter Server

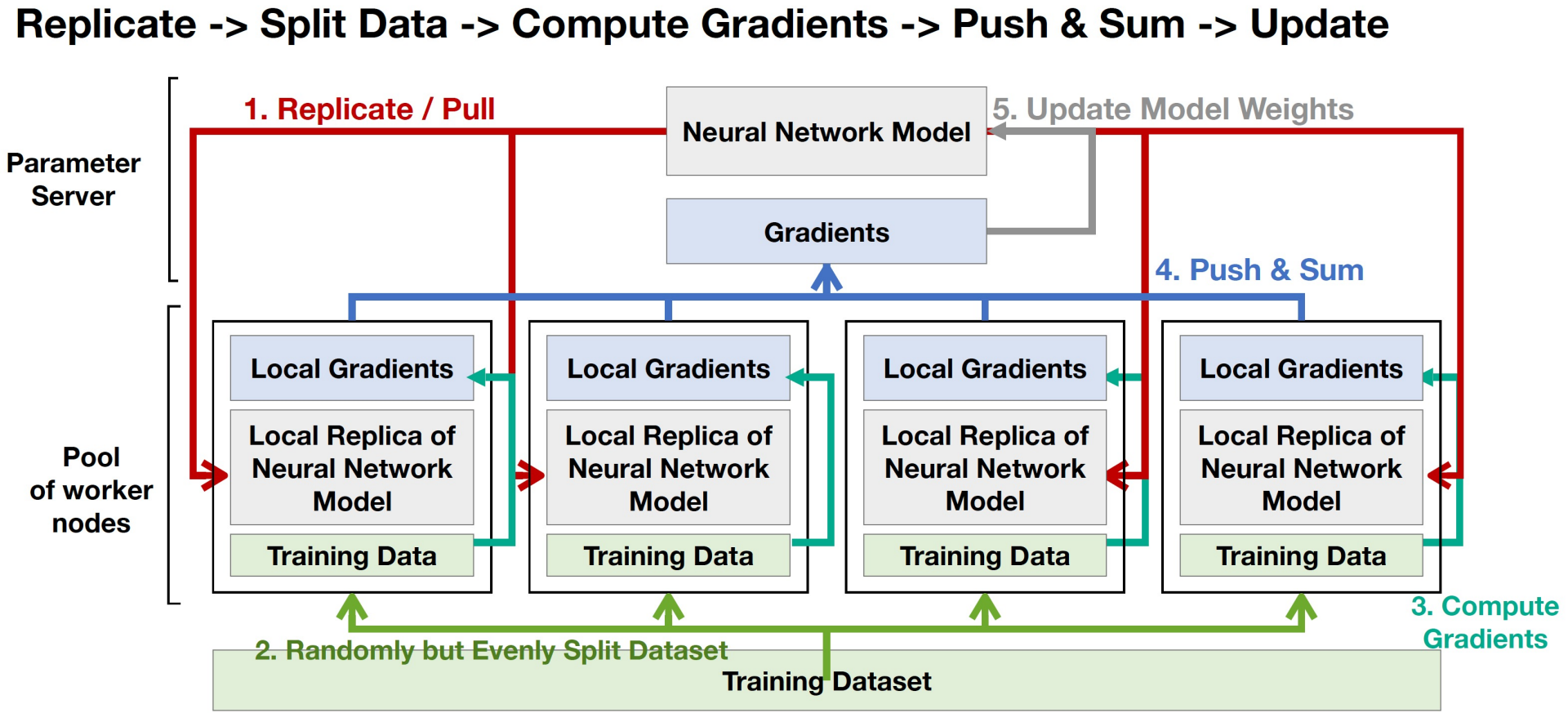
5 - Perform Gradient Step and Update Model Parameters



Synchronized Data Parallelism with the Parameter Server



Synchronized Data Parallelism with the Parameter Server



Comparison of Single Node vs. Distributed Training

Single Node Training:

For iteration i in 0 .. T,

1. Sample data from datasets $x_{(i)}, y_{(i)}$
2. Compute gradients $\nabla w_{(i)} = f(x_{(i)}, y_{(i)}; w_{(i)})$
3. Perform gradient step $w_{(i)} = w_{(i)} - \eta \nabla w_{(i)}$

Distributed Training:

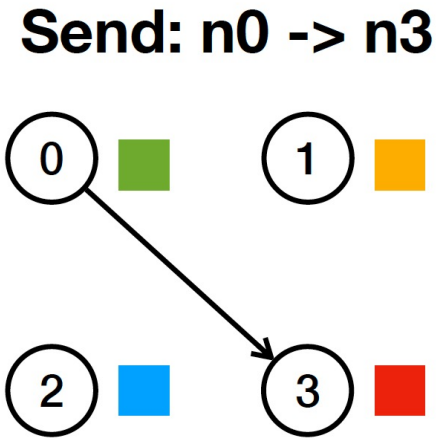
For iteration i in 0 .. T,

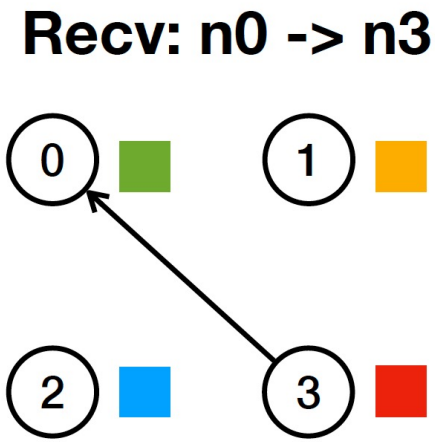
1. Replicate / pull gradients from parameter server
2. Sample data from datasets $x_{(i,j)}, y_{(i,j)}$
3. Compute gradients $\nabla w_{(i,j)} = f(x_{(i,j)}, y_{(i,j)}; w_{(i,j)})$
4. Push $\nabla w_{(i,j)}$ to parameter server and sum
5. Perform gradient step $w_{(i,j)} = w_{(i,j)} - \eta \overline{\nabla w_{(i)}}$ at parameter server.

two synchronization steps

Communication Primitives

One-to-One: Send and Recv



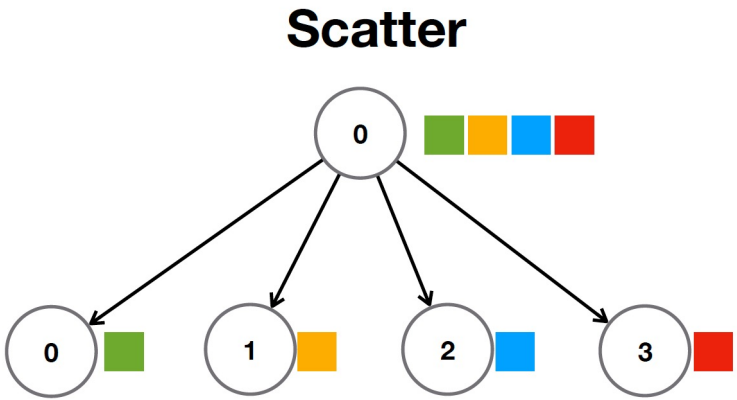


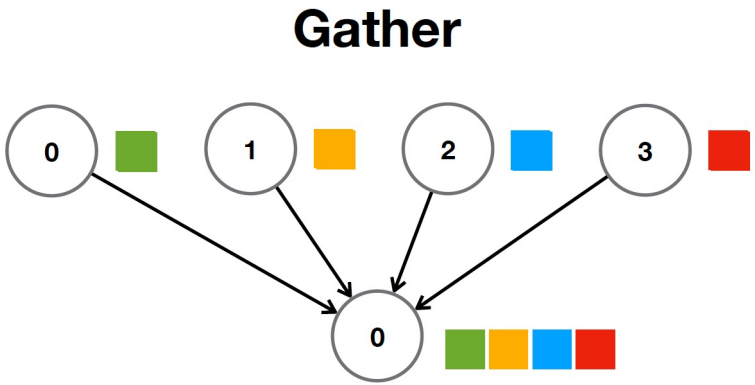
One-to-one communication: transfer data from one process to another

- Send & Receive are the most common distributed communication schemes.
- Implemented in Socket / MPI / Gloo / NCCL

Communication Primitives

One-to-Many: Scatter and Gather



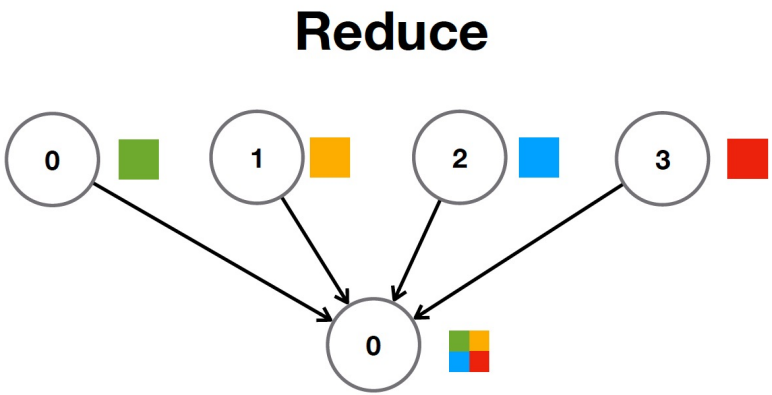


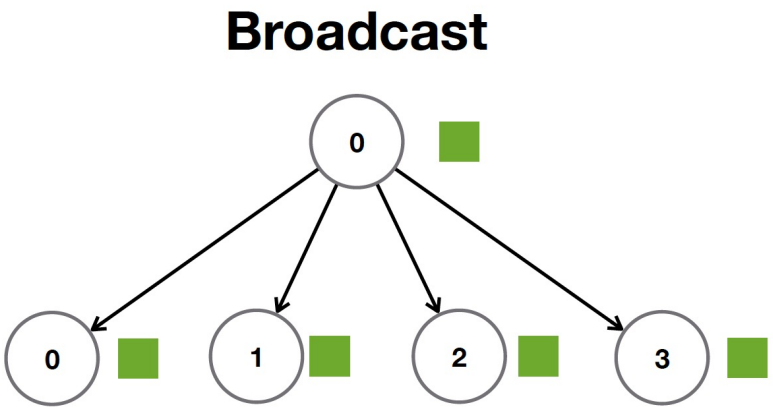
One-to-many communication: a type of operations performed across all workers.

- Scatter: send a tensor to all other workers
- Gather: receive a tensor from all other workers

Communication Primitives

One-to-Many: Reduce and Broadcast





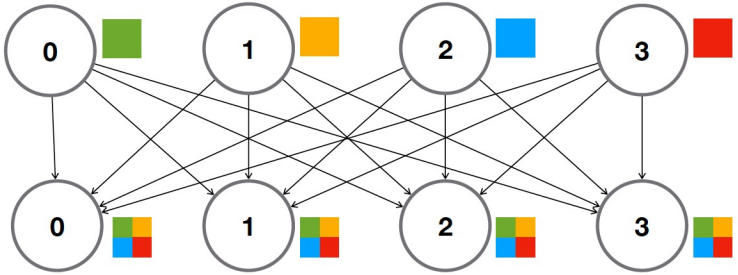
One-to-many communication: a type of operations performed across all workers.

- Reduce: similar to gather, but averaging / summing during aggregation.
  - [1] [2] [3] [4] —(gather)—> [1,2,3,4]
  - [1] [2] [3] [4] —(reduce)—> [1 + 2 + 3 + 4] = [10]
- Broadcast: send identical copies to all other workers.

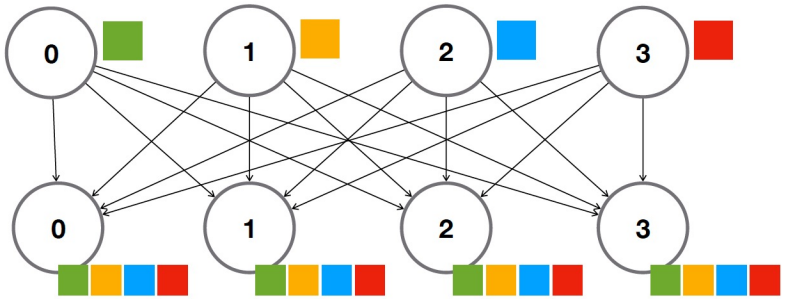
Communication Primitives

Many-to-Many: All Reduce and All Gather

All-Reduce



All-Gather

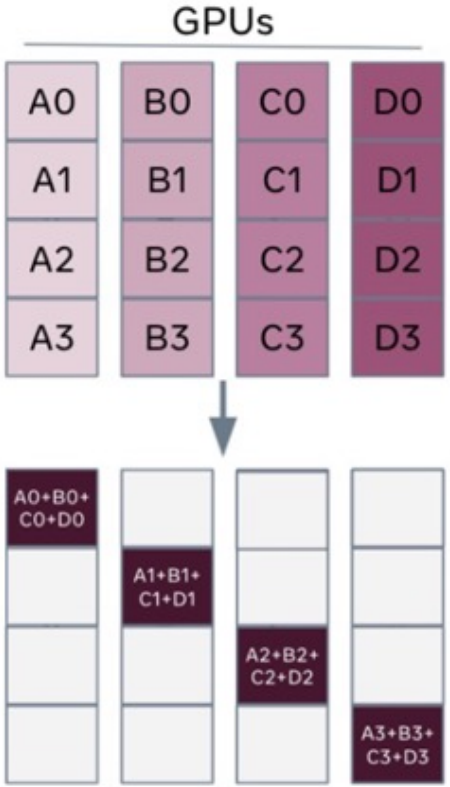


Many-to-many communication:

- All-Reduce: perform *Reduce* on all workers
- All-Gather: perform *Gather* on all workers

Communication Primitives

Many-to-Many: Reduce-Scatter



Perform the same operation as Reduce EXCEPT that the result is scattered in equal-sized blocks between ranks (i.e. GPUs) ;
Each rank (each GPU) getting a chunk of the data based on its rank-index.

In other words, input values are reduced across ranks (aka GPUs), with each rank receiving a subpart of the result.

Reduce-Scatter can be realized as:

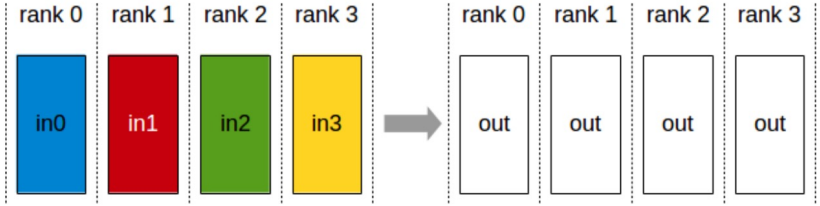
Step 1. Each worker performs Scatter on the data it carries locally, i.e.:

GPU0 scatters A, i.e., Keep A0 to self, Send A1 -> GPU1, A2 -> GPU2, A3 -> GPU3
Similarly,
GPU1 scatters B ;
GPU2 scatters C ;
GPU3 scatters D ;

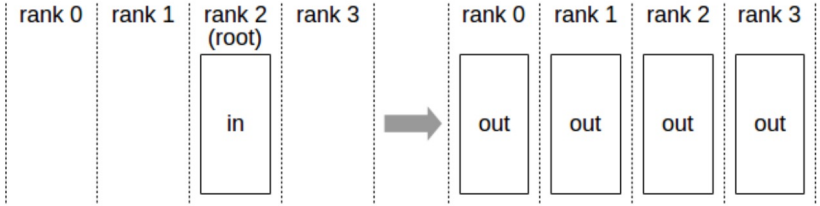
Step 2. Then each GPU sums up (or average) all data it receives.

Summary of Collective Communication Primitives

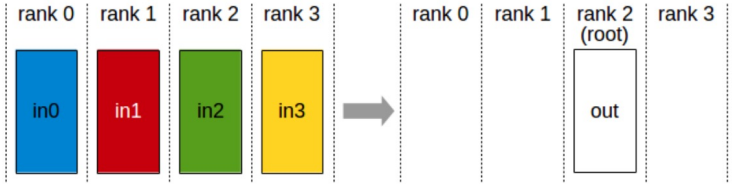
All reduce



Broadcast



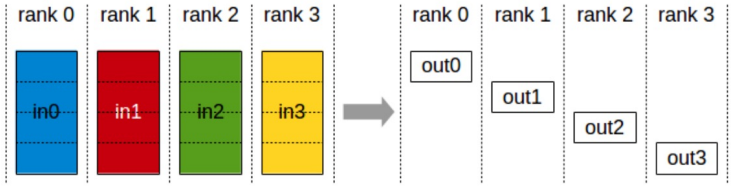
Reduce



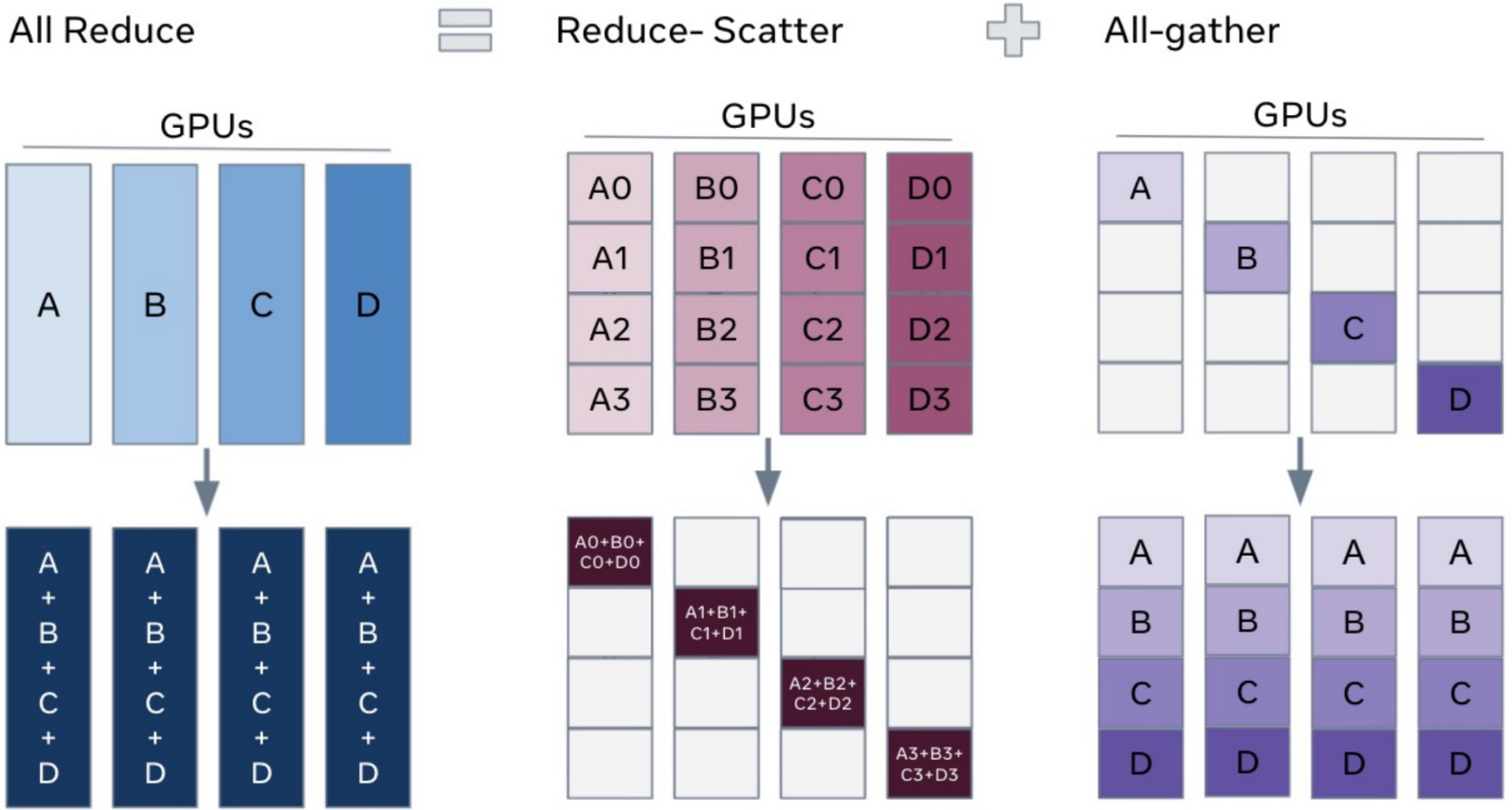
All Gather



Reduce Scatter

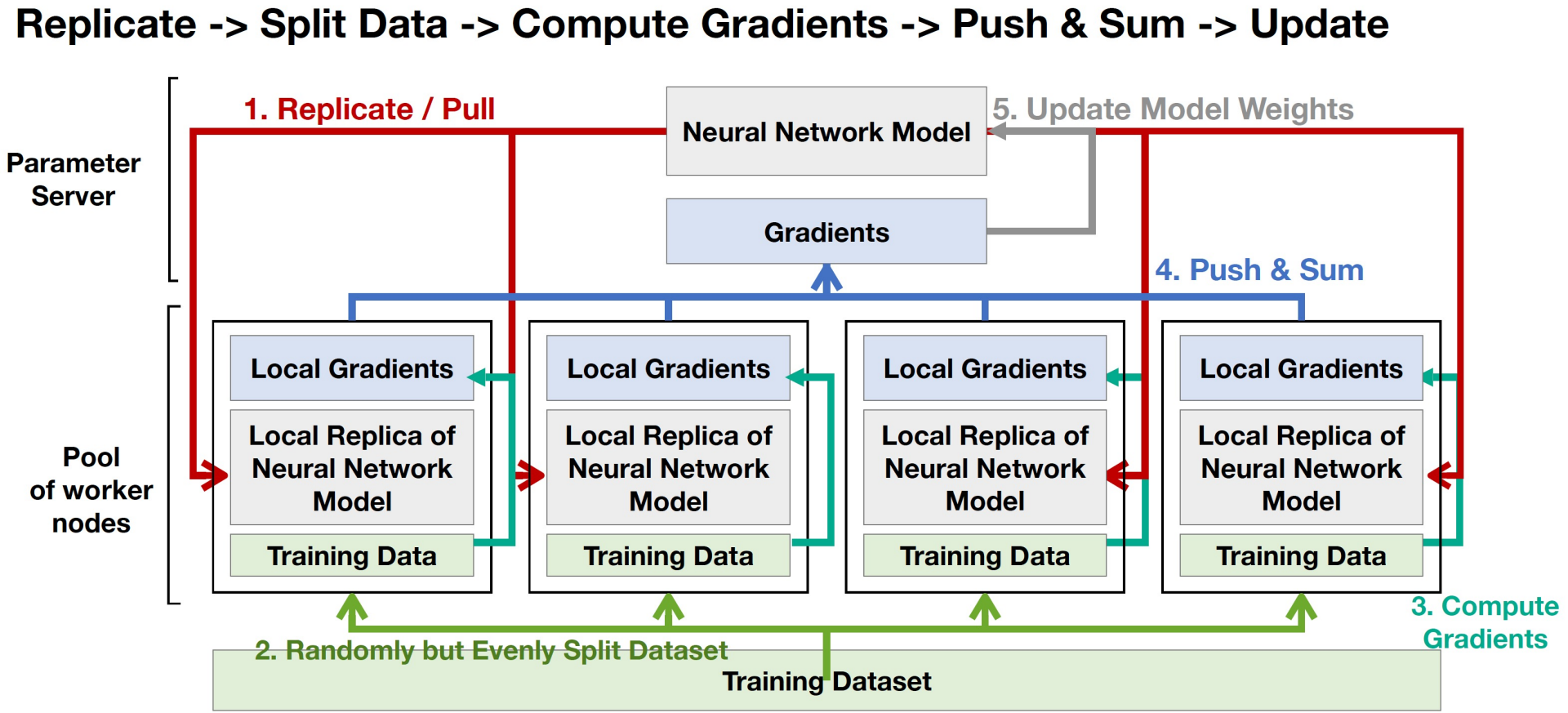


All Reduce realized in 2 Steps: Reduce-Scatter then All-Gather



- Under the Bandwidth-limited Regime, this is the Best we can do !

Synchronized Data Parallelism with the Parameter Server



Comparison of Single Node vs. Distributed Training

Single Node Training:

For iteration i in 0 .. T,

1. Sample data from datasets $x_{(i)}, y_{(i)}$
2. Compute gradients $\nabla w_{(i)} = f(x_{(i)}, y_{(i)}; w_{(i)})$
3. Perform gradient step $w_{(i)} = w_{(i)} - \eta \nabla w_{(i)}$

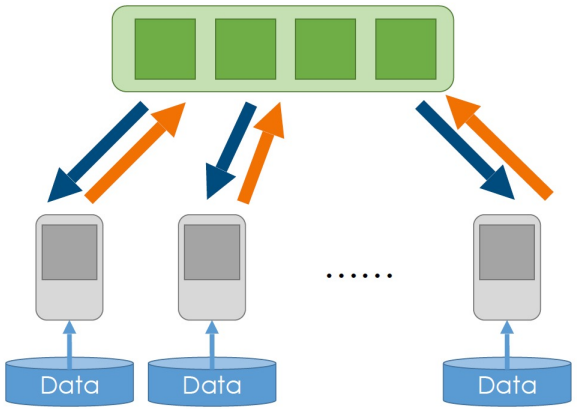
Distributed Training:

For iteration i in 0 .. T,

1. Replicate / pull gradients from parameter server
2. Sample data from datasets $x_{(i,j)}, y_{(i,j)}$
3. Compute gradients $\nabla w_{(i,j)} = f(x_{(i,j)}, y_{(i,j)}; w_{(i,j)})$
4. Push $\nabla w_{(i,j)}$ to parameter server and sum
5. Perform gradient step $w_{(i,j)} = w_{(i,j)} - \eta \overline{\nabla w_{(i)}}$ at parameter server.

two synchronization steps

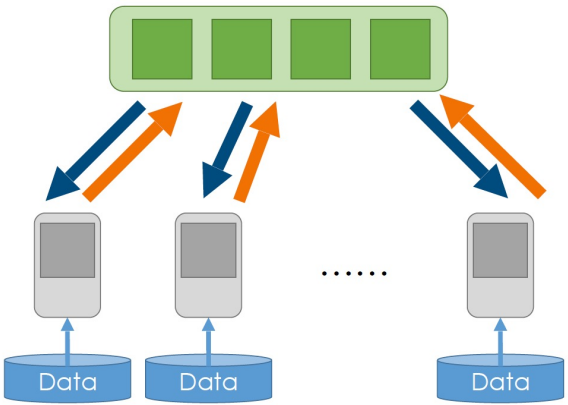
Communication Schemes used in Parameter Server



- Which type of communication scheme is used?
  - **Replicate & Pull**: Broadcast
    - Parameter Server to all workers
  - **Push & Sum**: Reduce
    - Average from all workers to Parameter Server
- What is the bandwidth requirements on each node?
  - **Replicate & Pull**:
    - Worker: O(1)
    - Parameter Server: O(P)
  - **Push & Sum**:
    - Worker: O(1)
    - Parameter Server: O(P)

Note: P is the number of training workers.

Communication Schemes used in Parameter Server



- **Replicate & Pull**:
  - Worker: O(1)
  - Parameter Server: O(P)

- **Push & Sum:**
  - Worker: O(1)
  - Parameter Server: O(P)

The bandwidth of central parameter server grows linearly w.r.t #num of workers, which can be a **bottleneck** when there are more machines.

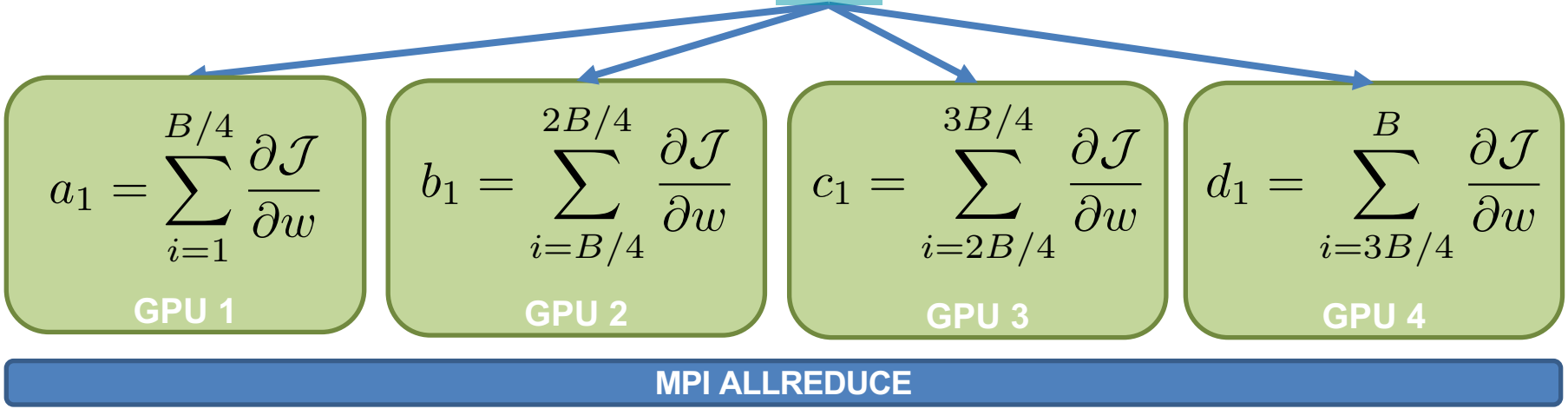
Can we perform the aggregation without a central server?

All-Reduce!

Note: P is the number of training workers.

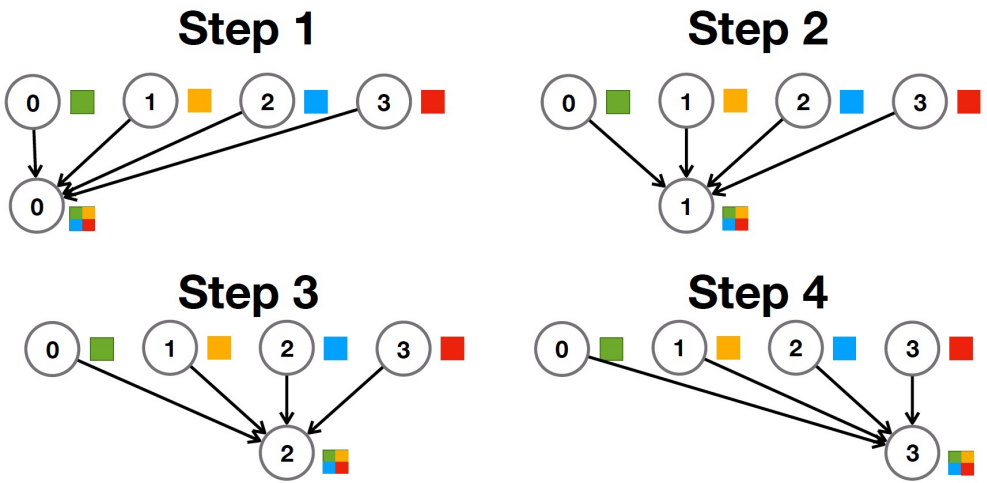
All Reduce

$$w^1 = w^0 - \frac{\alpha}{B} \sum_{i=1}^{B} \frac{\partial \mathcal{J}(w^0)}{\partial w}$$



$$\sum_{i=1}^{B} \frac{\partial \mathcal{J}}{\partial w} = a_1 + b_1 + c_1 + d_1$$

Naïve All-Reduce Implementation - Sequential




```
Pseudocode:
_______________
For i:=0 to P:
   Allreduce(work[i])
```

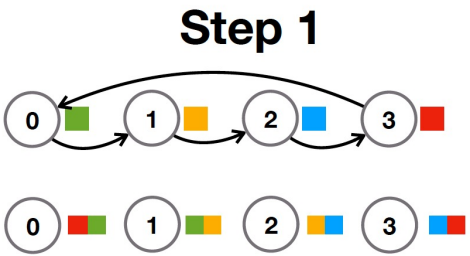
Time: O(P)

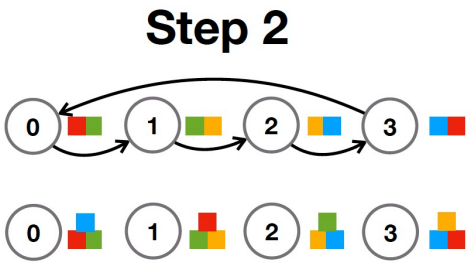
Bandwidth: O(P)

Each step performs a **SINGLE** Reduce operation

Note: P is the number of training workers.

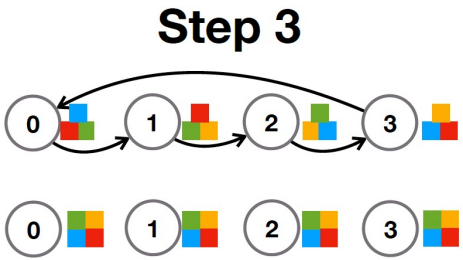
Better All-Reduce Implementation - Ring





Pseudocode:

```
For i:=0 to P:
   send(src=i, dst=(i+1)%P, data=worker[i])
   obj = recv(src=(i+1)%P, dst=i)
   worker[i] = merge(obj, worker[i])
```

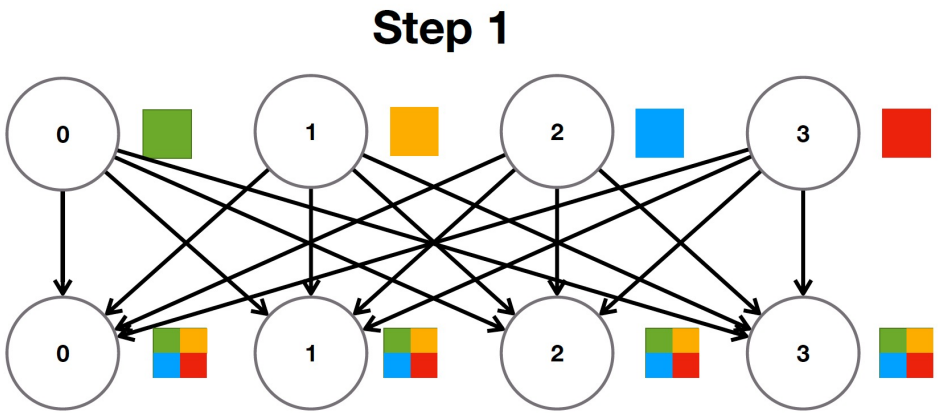


Time: O(P)

Bandwidth: O(1) **<- reduced**

Each step performs a **SINGLE** send and merge.

Naïve All-Reduce Implementation – Parallel Reduce




```
Pseudocode:
------------------
Parallel for i:=0 to P:
   Allreduce(work[i])
```

Time: O(1) **<- improved**

Bandwidth: O(P^2) **<- worse**

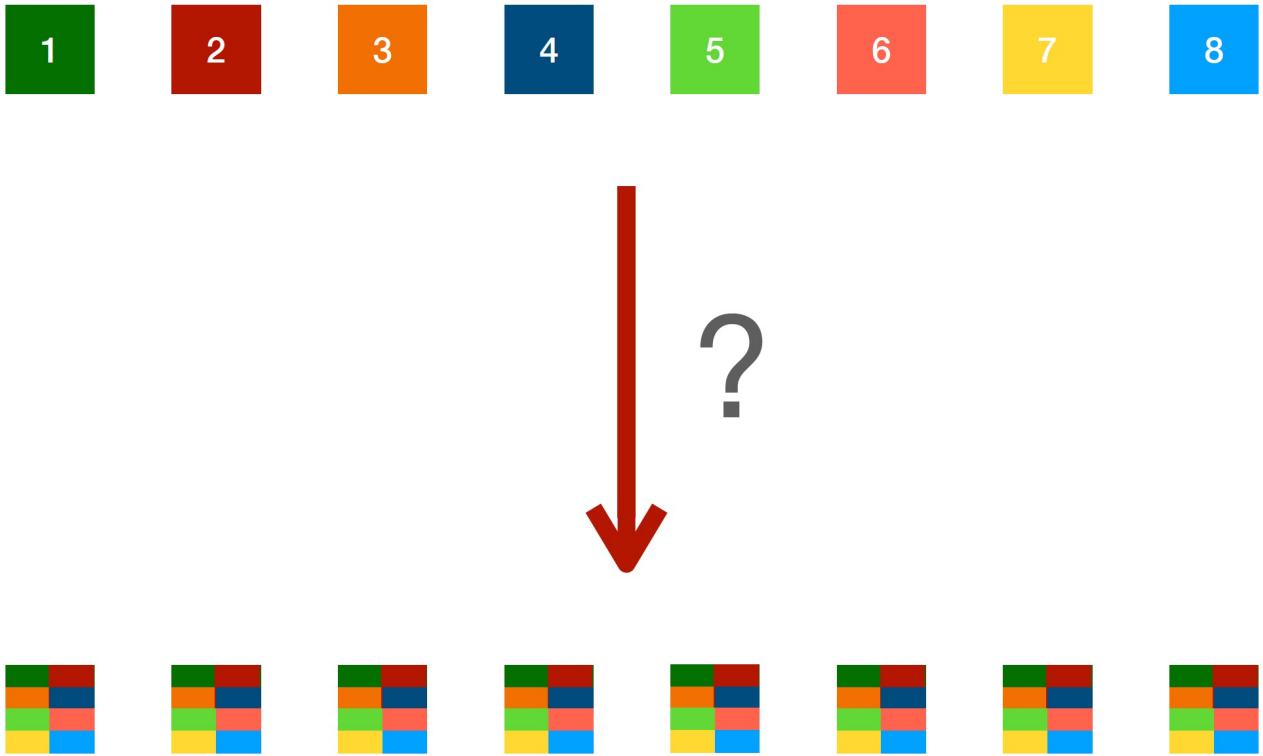
Perform **ALL** reduce operations simultaneously.

Comparison b/w Different Distributed Aggregation schemes

	Time	Peak Node Bandwidth	Total Bandwidth
Parameter Server	O(1)	O(P)	O(P)
All-Reduce - Sequential	O(P)	O(P)	O(P)
All-Reduce - Ring	O(P)	O(1)	O(P)
All-Reduce - Parallel	O(1)	O(P)	O(P^2)

Can we combine the advantages of Ring and Parallel?

Recursive Halving All Reduce

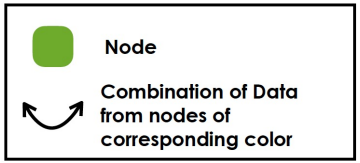


[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

Recursive Halving All Reduce

Step 1 - Each node exchanges with neighbors with offset 1

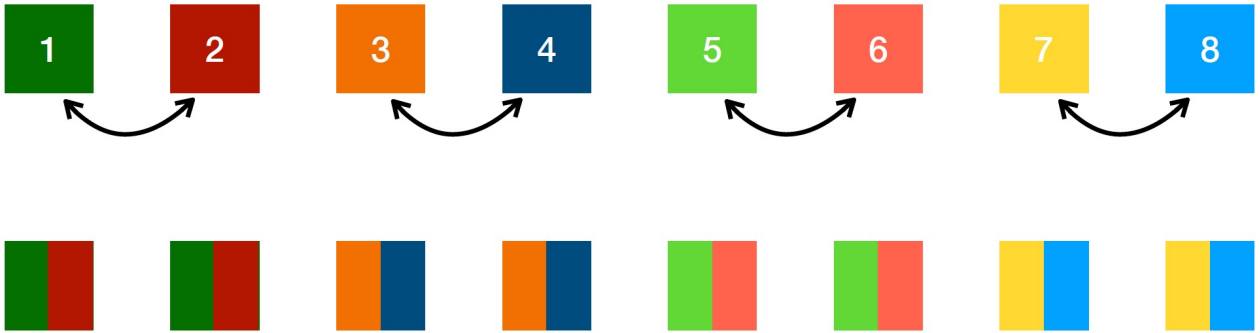


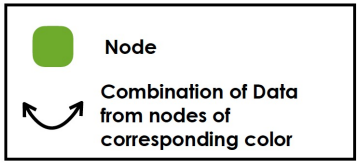


[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

Recursive Halving All Reduce

Step 1 - Each node exchanges with neighbors with offset 1





[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

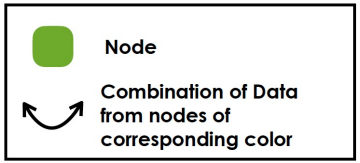
Recursive Halving All Reduce

Step 1 - Each node exchanges with neighbors with offset 1



Step 2 - Each node exchanges with neighbors with offset 2





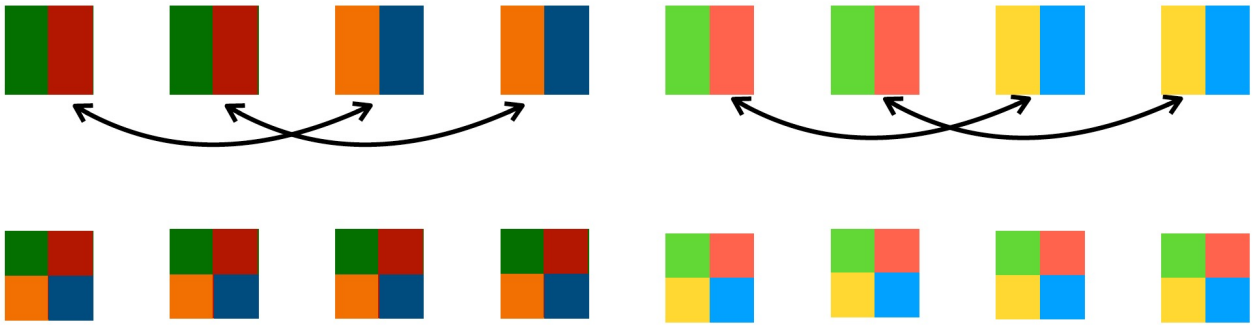
[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

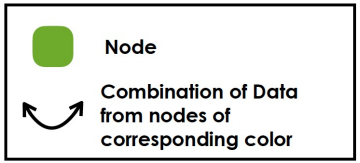
Recursive Halving All Reduce

Step 1 - Each node exchanges with neighbors with offset 1



Step 2 - Each node exchanges with neighbors with offset 2





[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

Recursive Halving All Reduce

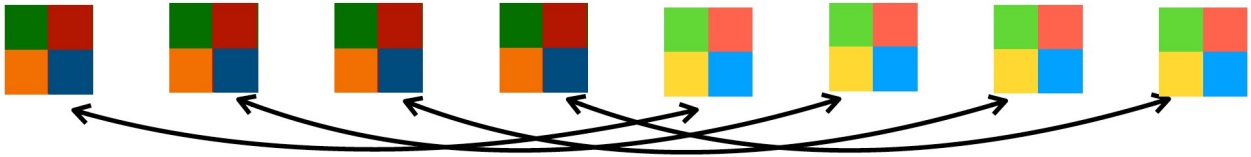
Step 1 - Each node exchanges with neighbors with offset 1

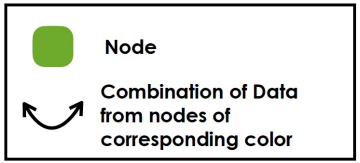


Step 2 - Each node exchanges with neighbors with offset 2



Step 3 - Each node exchanges with neighbors with offset 4





[1] Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. "Optimization of collective communication operations in MPICH."

Recursive Halving All Reduce

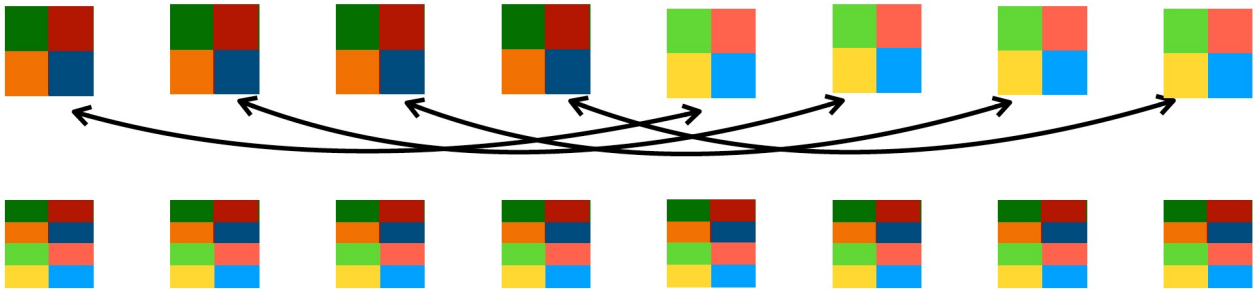
Step 1 - Each node exchanges with neighbors with offset 1

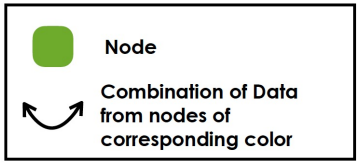


Step 2 - Each node exchanges with neighbors with offset 2



Step 3 - Each node exchanges with neighbors with offset 4

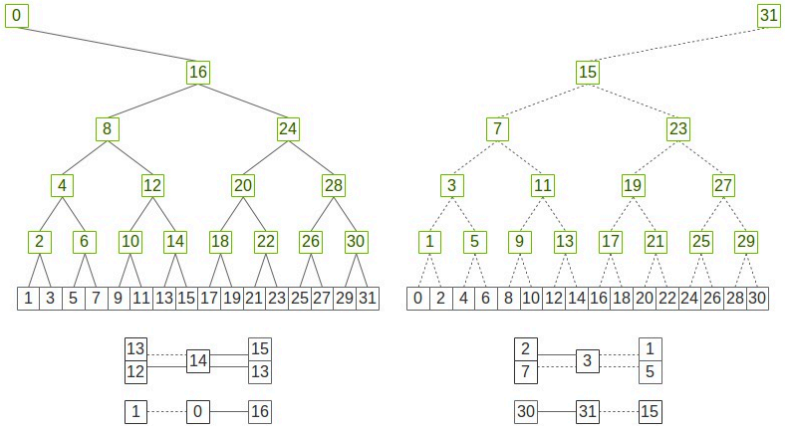


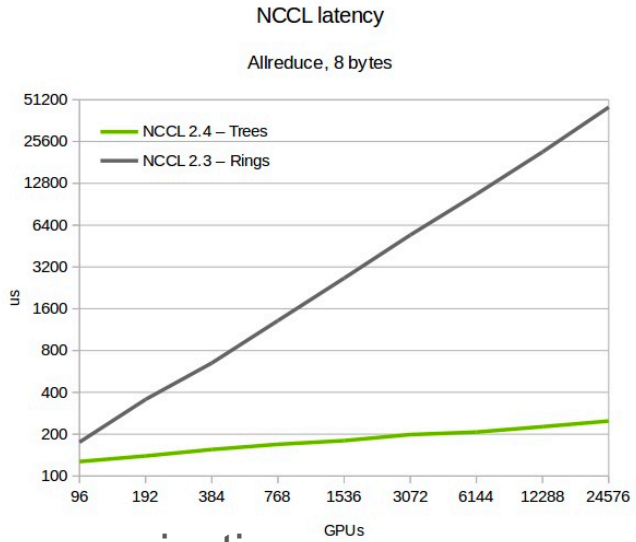


For N workers, AllReduce finish in **log(P)** steps.

Double Binary Tree All-Reduce

- Two overlaid binary reduction trees





- Double the fan-in → Log(p) rounds of communication
  - Currently used on Summit super-computer and latest NCCL

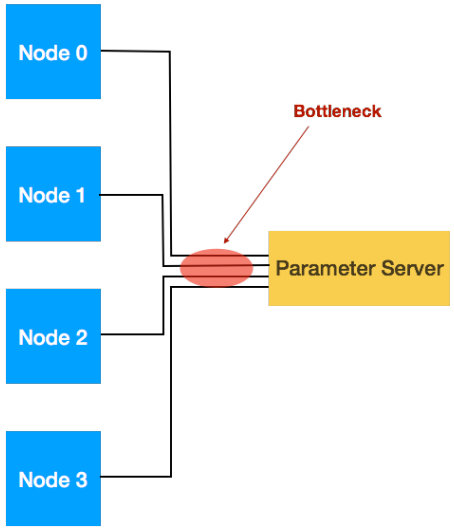
https://devblogs.nvidia.com/massively-scale-deep-learning-training-nccl-2-4/

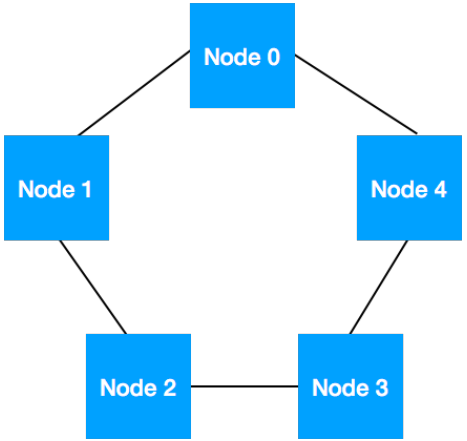
Communication Complexity Summary

$$T_{comm} = (\alpha + PN\beta)$$

$$T_{comm} = 2((P-1)\alpha + \frac{P-1}{P}N\beta)$$

α fixed latency
β = 1/ bandwidth
N message size
P #processes





Parameter Server

Ring All-reduce

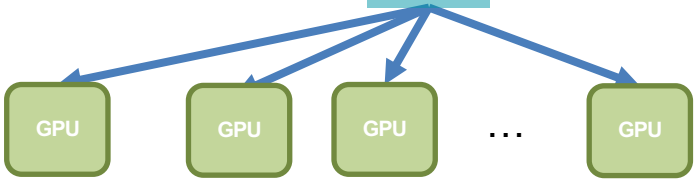
Great Reference: T. Rajeev, R. Rabenseifner, and W. Gropp. "Optimization of collective communication operations in MPICH." ***The International Journal of High Performance Computing Applications,*** **2005.**

Data Parallel Training Complexity Analysis

- Question: Communication time of Ring AllReduce is independent of the number of processors. So, what limits scalability (i.e. prevents us from increasing *P* indefinitely ?)

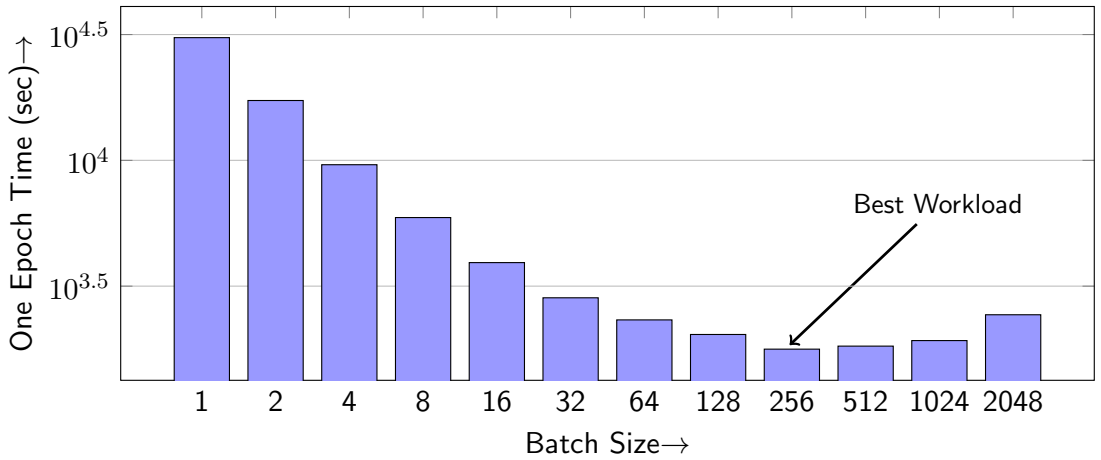
$$T_{comm}(batch) = 2\sum_{i=0}^{L}\left(\alpha(P-1) + \beta\frac{P-1}{P}|W_i|\right)$$

$$w^1 = w^0 - \frac{\alpha}{B}\sum_{i=1}^{B}\frac{\partial\mathcal{J}(w^0)}{\partial w}$$



Limits of Data Parallel Scaling

- The maximum limit of processors that you can use is P=B
- But this often leads to very low utilization of the hardware and would not yield any speed up

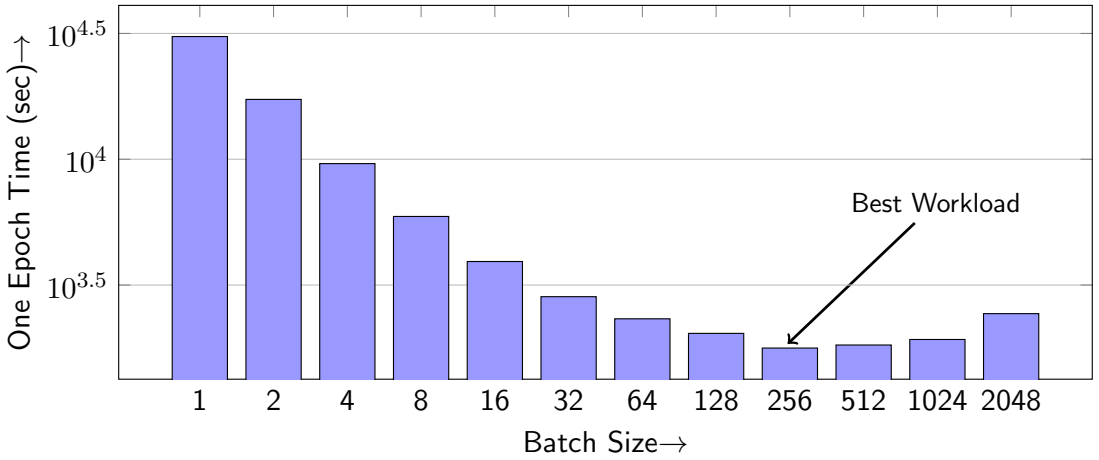


- Why does this happen?
  - Roofline model

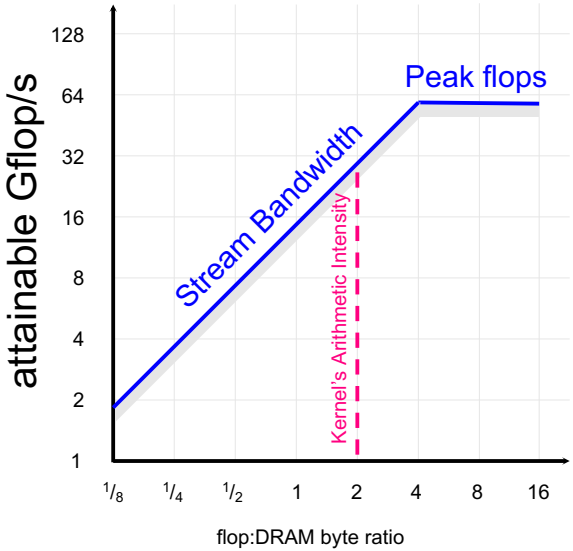
One epoch training time of AlexNet computed on an Intel KNL system

Limits of Data Parallel Scaling

- The maximum limit of processors that you can use is P=B
- But this often leads to very low utilization of the hardware and would not yield any speed up

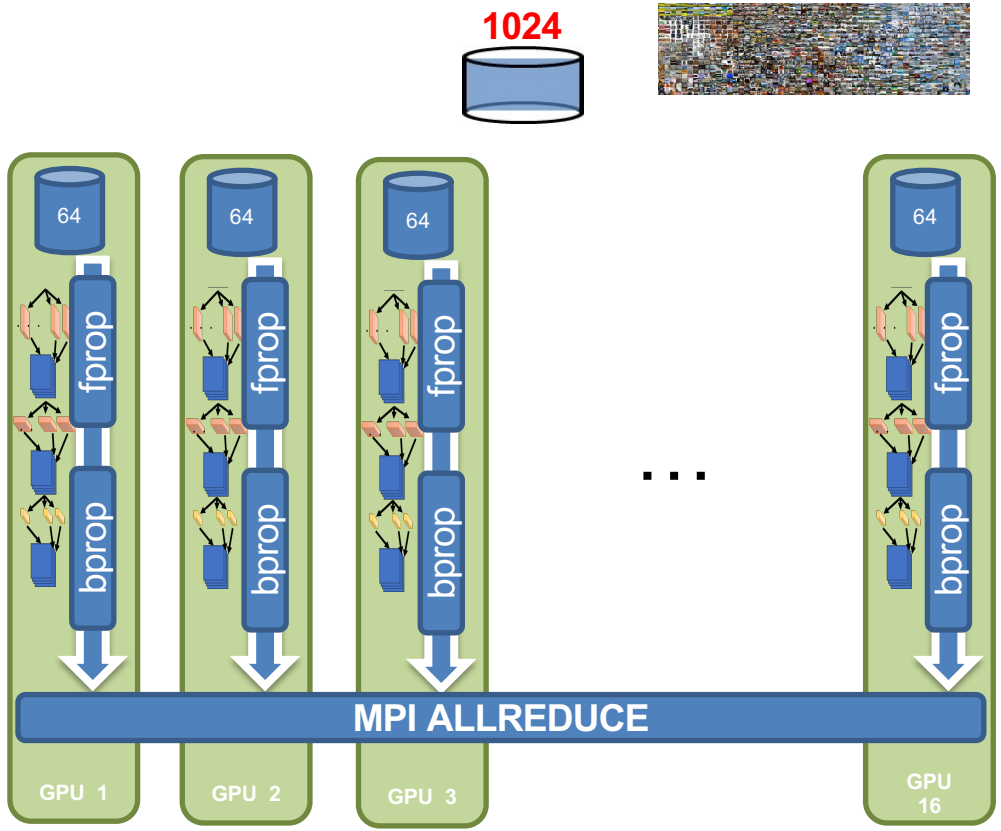


One epoch training time of AlexNet computed on an Intel KNL system

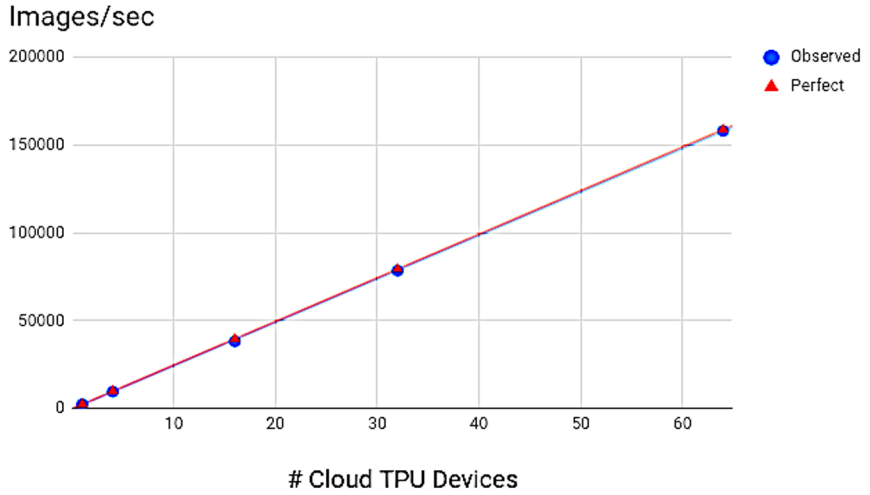


Scaling Data Parallel Training

If we want to keep scaling synchronous SGD then we have to keep increasing the batch size.



Naively increasing Batch size leads to perfect results but ...



$$\left(\frac{\text{“Learning”}}{\text{Second}}\right) = \left(\frac{\text{“Learning”}}{\text{Record}}\right) \times \left(\frac{\text{Record}}{\text{Second}}\right)$$

Convergence
Machine Learning
Property

Throughput
System
Property

Bigger BATCH isn't Always Better

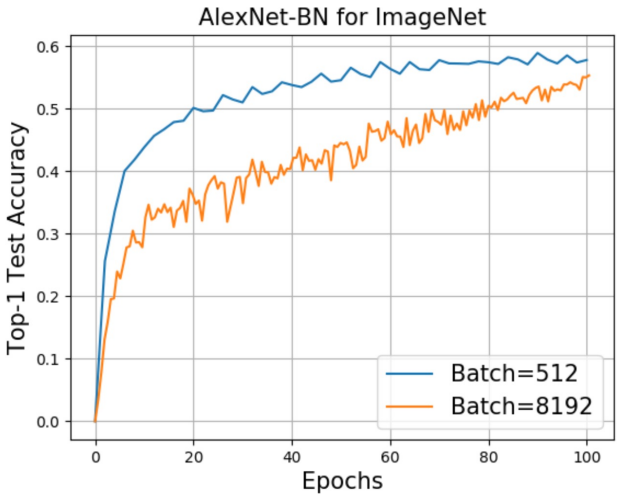
- Motivation for larger batch sizes
  - More opportunities for parallelism → but is it useful?
  - Recall (1/n variance reduction):

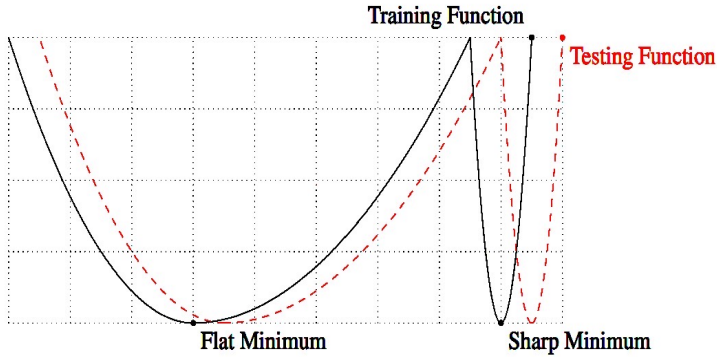
$$\frac{1}{n}\sum_{i=1}\nabla_\theta \mathbf{L}(y_i, f(x_i;\theta)) \approx \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\nabla_\theta \mathbf{L}(y_i, f(x_i;\theta))$$

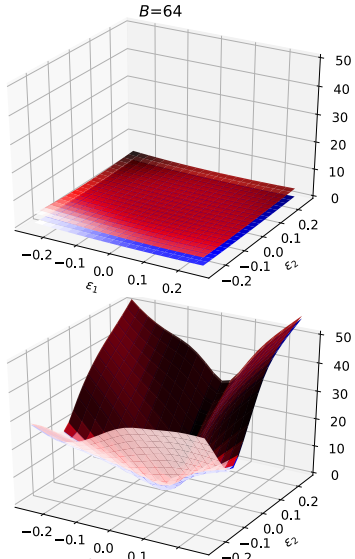
- Is a variance reduction helpful?
  - Only if it let's you take bigger steps (move faster)
  - Does it affect the final prediction accuracy?

Problems with Large Batch Training

- Larger Batch leads to sub-optimal generalization
- A common belief is that large batch training gets attracted to "sharp minimas"

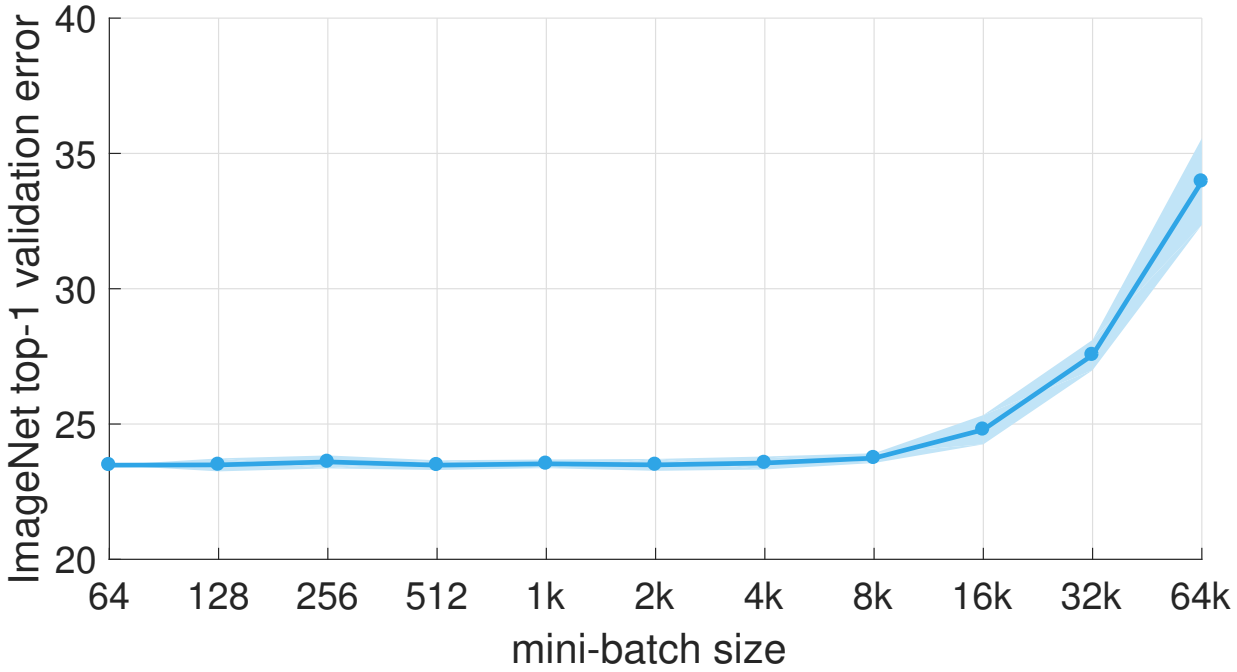






Keskar et al., On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima, ICLR'16.
Z. Yao, A. Gholami, Q. Lei, K. Keutzer, M. Mahoney. Hessian-based Analysis of Large Batch Training and Robustness to Adversaries, NeurIPS'18.
Ginsburg, Boris, Igor Gitman, and Yang You. "Large Batch Training of Convolutional Networks with LARS." arXiv:1708.03888, 2018.

Generalization Gap Problem



Larger batch sizes harm generalization performance.

Why? Large Batch Reduces Noise and may Get Trapped in Local Minima

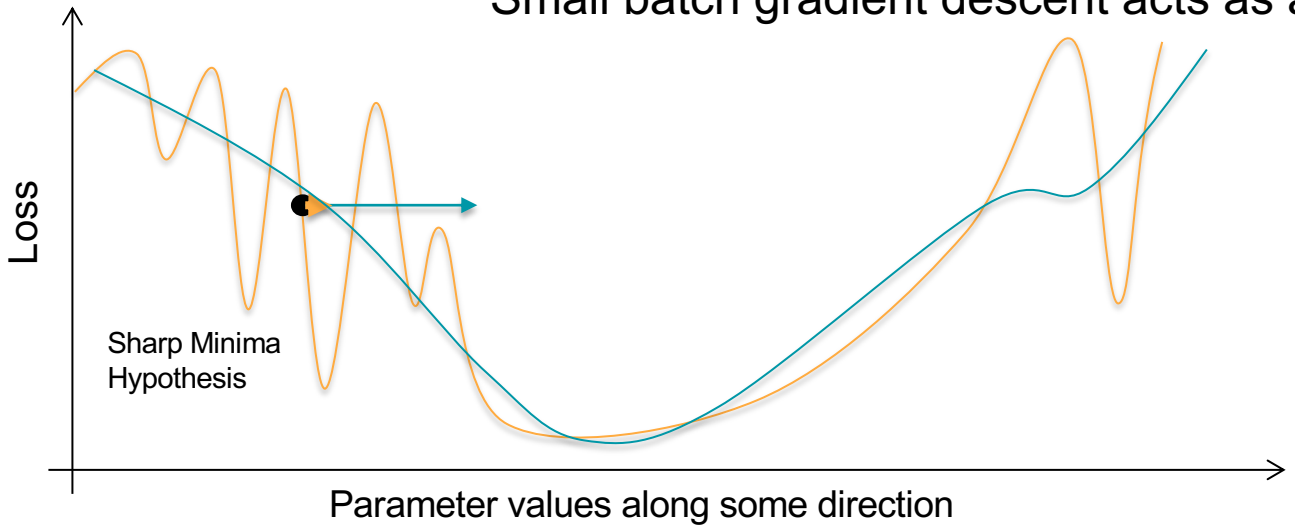
Objective function

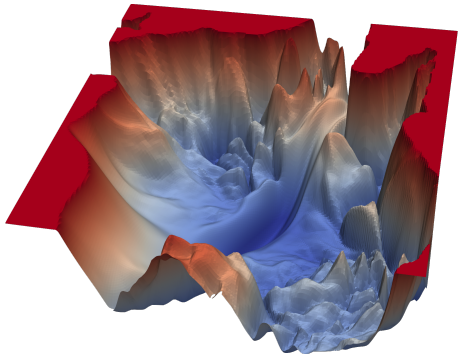
$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i, \theta)$$

Update rule

$$\theta_{t+1} = \theta_t - \eta_t \frac{1}{|B|} \sum_{(x,y)\in B} \nabla_\theta l(x, y, \theta_t)$$

Small batch gradient descent acts as a **regularizer**





Active Research problem: *Addressing the generalization gap for large batch sizes.*

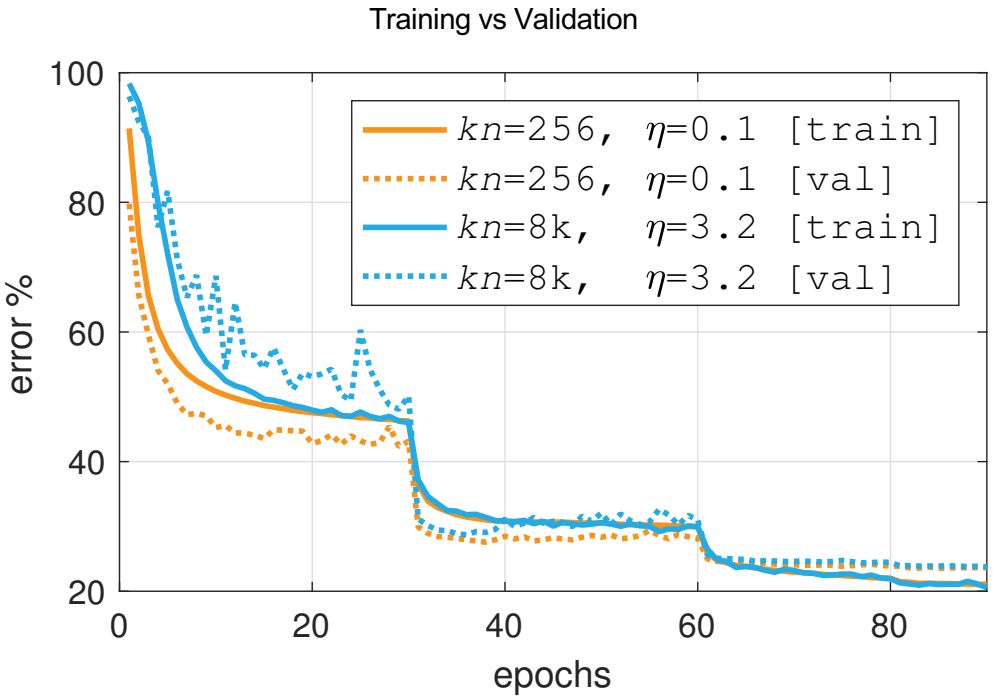
Solution: Linear Scaling Rule

- Scale the learning rate linearly with the batch size

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \hat{\eta} \overbrace{= \eta k}^{} \left(\frac{1}{k} \sum_{j=1}^{k} \frac{1}{|\mathcal{B}_j|} \sum_{i \in \mathcal{B}_j} \nabla_\theta \mathbf{L}(y_i, f(x_i; \theta)) \Big|_{\theta=\theta^{(t)}} \right)$$

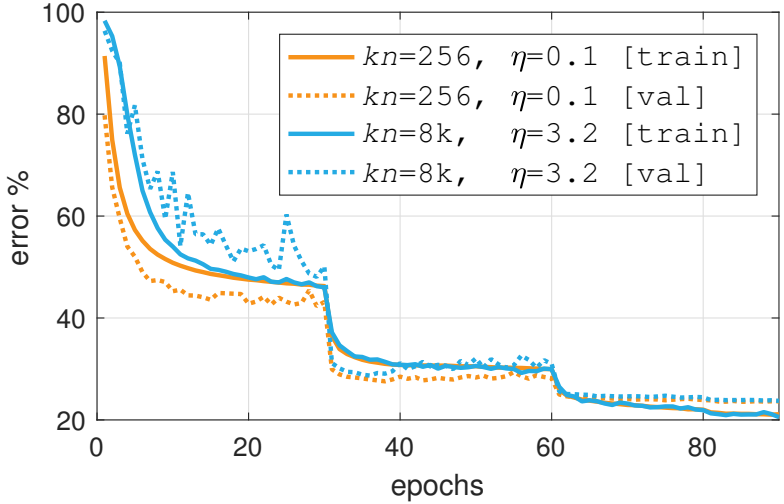
- Addresses generalization performance by **taking larger steps** (also improves training convergence)
- **Sub-problem:** *Large learning rates can be destabilizing in the beginning.* Why?
  - **Gradual warmup solution:** increase learning rate scaling from constant to linear in first few epochs
  - Doesn't help for very large k…

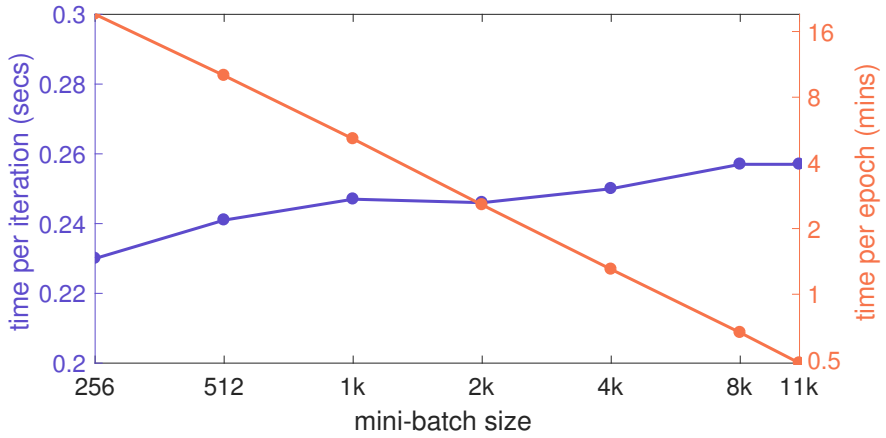
Key Results



All curves closely match using the linear scaling rule. Note learning rate schedule drops.

Key Results





$$\left(\frac{\text{“Learning”}}{\text{Epoch}}\right)$$

Machine Learning

$$\left(\frac{\text{Epoch}}{\text{Second}}\right)$$

System

Key Results

- Train ResNet-50 to state-of-the-art on 256 GPUs in 1 hour
  - 90% scaling efficiency

- Fairly careful study of the linear scaling rule
  - Observed limits to linear scaling do not depend on dataset size
  - But what is the limit?
    - You cannot indefinitely scale the learning rate …

Since then there has been a race to train ImageNet faster and several new large batch training methods have been developed (some with good foundation and some heuristics)

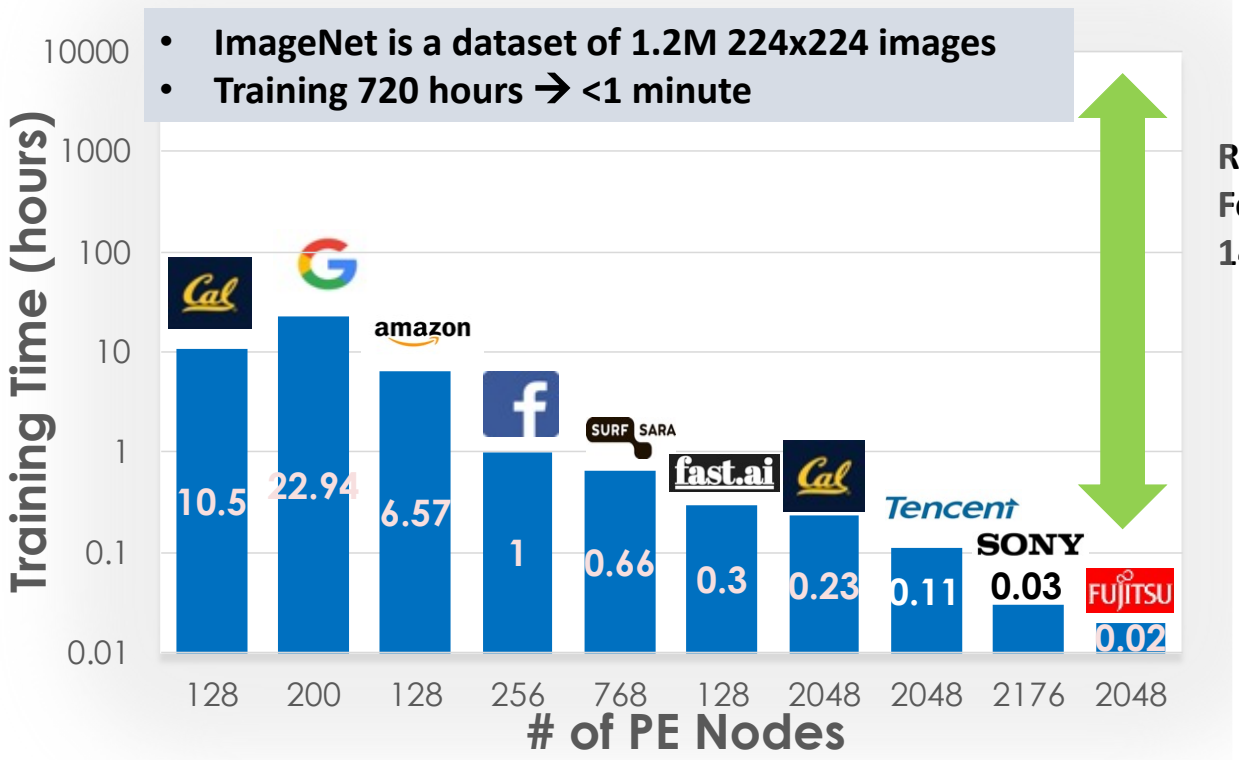
ImageNet Training Competition!



Iandola



Yang You



Record as of Feb 2022: 14 seconds / TPUv4

- Iandola FN, Moskewicz MW, Ashraf K, Keutzer K. **FireCaffe**: near-linear acceleration of deep neural network training on compute clusters. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition 2016 (pp. 2592-2600).
- You Y, Zhang Z, Hsieh CJ, Demmel J, Keutzer K. **Imagenet training in minutes**. In Proceedings of the 47th International Conference on Parallel Processing 2018 Aug 13 (p. 1) ACM **(Best Paper Award)**

More Resources of this Active area of Research

- Fabian Pedregosa of Google AI, "A birds-eye view of optimization algorithms," Nov. 2018, https://fa.bianp.net/teaching/2018/eecs227at/
  This webpage provides nice, interactive visualization of how GD and SGD behave under different settings, e.g. learning rate/ step-size, etc"
  https://fa.bianp.net/teaching/2018/eecs227at/gradient_descent.html
  https://fa.bianp.net/teaching/2018/eecs227at/stochastic_gradient.html

- An insightful guess lecture by Prof. Suvrit Sra on Stochastic Gradient Descent (Lecture #25) in the MIT course titled "Matrix Methods in Data Analysis, Signal Processing and Machine Learning" by Prof. Gilbert Strang:
  - https://ocw.mit.edu/courses/18-065-matrix-methods-in-data-analysis-signal-processing-and-machine-learning-spring-2018/resources/lecture-25-stochastic-gradient-descent/
  - In particular, Prof. Sra gave a nice one-dimension example (starting at: 22:50/53:03) to illustrate why SGD works so well at the beginning stage in terms of moving along the right direction towards the optimal point even though only ONE random data-point is used to compute the gradient. Furthermore, it also shows that, once reaching the "neighborhood" of the optimal region, SGD actually may not zoom into the "ultimate optimal" point but wandering around within the neighborhood. This behavior can help to provide a more generalizable solution !

- Golmant, Noah, et al. "On the computational inefficiency of large batch sizes for stochastic gradient descent" **(UC Berkeley)**

- Shallue et al. "Measuring the Effects of Data Parallelism on Neural Network Training" **(Google)**

- You, Yang, et al. "Large batch optimization for deep learning: Training BERT in 76 minutes." **(UC Berkeley)**

Recap: Data Parallelism - Summary so far

- An efficient parallel training method where the comm time is independent of processors with Ring AllReduce
- Very easy to implement. Only requires AllReduce operation before updating parameters
- Very challenging to scale. Using large batch training is not an option as it hurts generalization performance.
  - Existing solutions often require a lot of tuning (outside of ResNet-50 on ImageNet)
- Processes are never idle

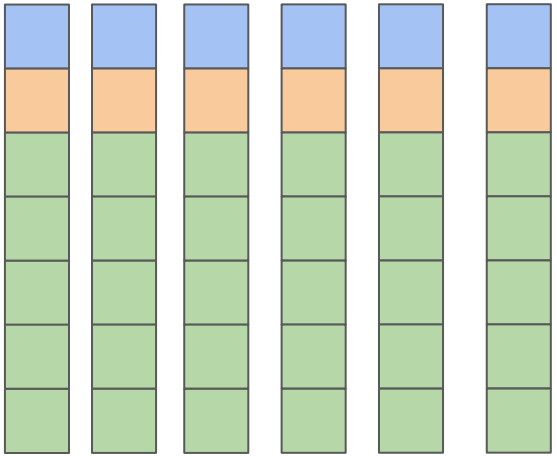
BUT

- Does not work for large models such as GPT-3 (w/ 175B parameters) which are too large to fit in the memory of a Single, even the most powerful, GPU

Reducing GPU Memory Requirement for Data Parallelism with ZeRO

What's Wrong with Naïve Data Parallelism ?
GPU Memory Requirement

- GPU Memory required to keep track of the Model states:



Model Parameters

Gradients

Optimizer States:

e.g. Momentum, variance, etc for the Adam

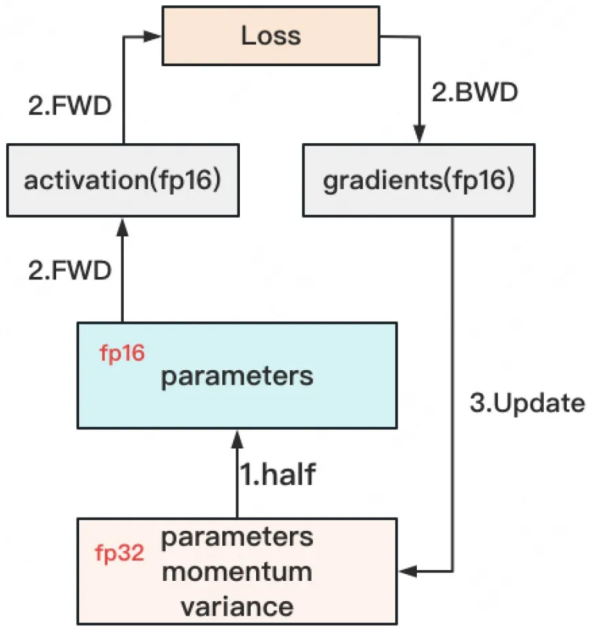
Depending on our precision..

We need 5 copies of weights and 16 bytes per param!

- 2 bytes for FP/BF 16 model parameters
- 2 bytes for FP/BF 16 gradients
- 4 bytes for FP32 master weights (the thing you accumulate into in SGD)
- 4 (or 2) bytes for FP32/BF16 Adam first moment estimates
- 4 (or 2) bytes for FP32/BF16 Adam second moment estimates

"Optimizer state"

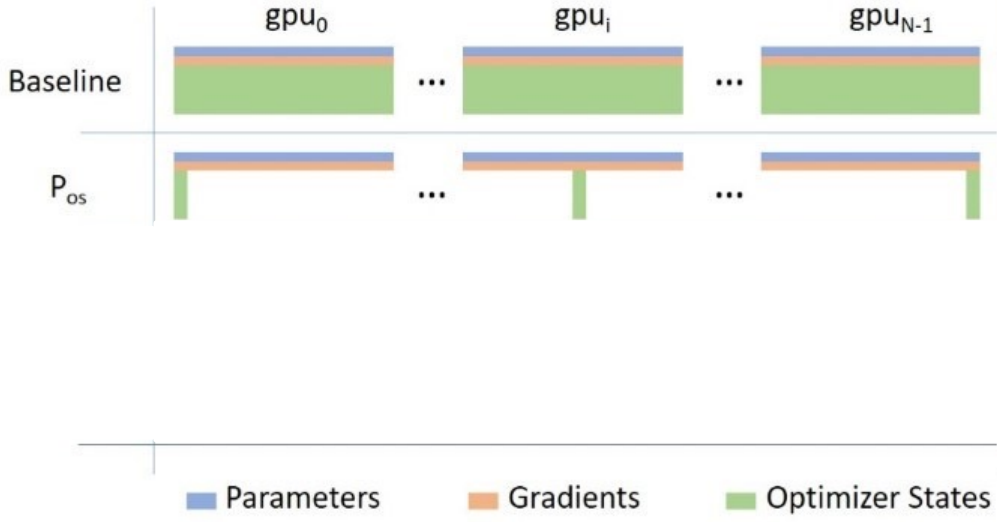
Motivation for ZeRO: Zero Redundancy Optimizer



- Maybe surprising to many people, Optimizer consumes the most memory, especially with mixed-precision training
  - Consider fp16/32, use Adam as an example.
  - Consider a model with Ψ parameters
    - fp16 copy of parameters and gradients
      - 2Ψ bytes + 2Ψ bytes
    - Optimizer states: fp32 copy of parameters, momentum and variance
      - 4Ψ bytes + 4Ψ + 4Ψ bytes
    - Total 16Ψ bytes
  - 1.5B model
    - Memory requirements of 24GB

ZeRO-DP: ZeRO powered Data Parallelism

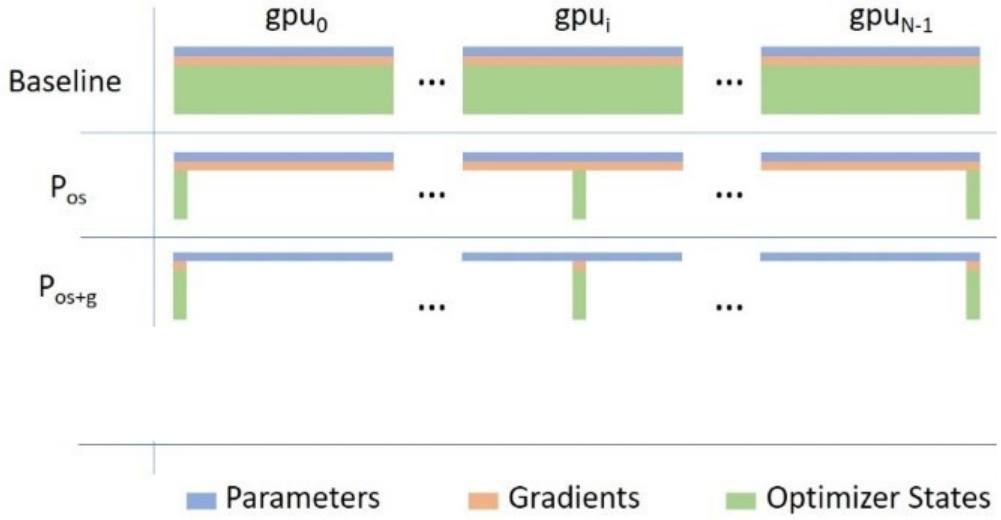
- **Core Idea:**
  - To Split up the expensive parts (Model states) and use Reduce-Scatter equivalence
    => ZeRO removes the redundancy across data parallel processes
- Partition optimizer states, gradients and parameters (was implemented in 3 stages)



Stage 1 (P_{os})

ZeRO-DP: ZeRO powered Data Parallelism

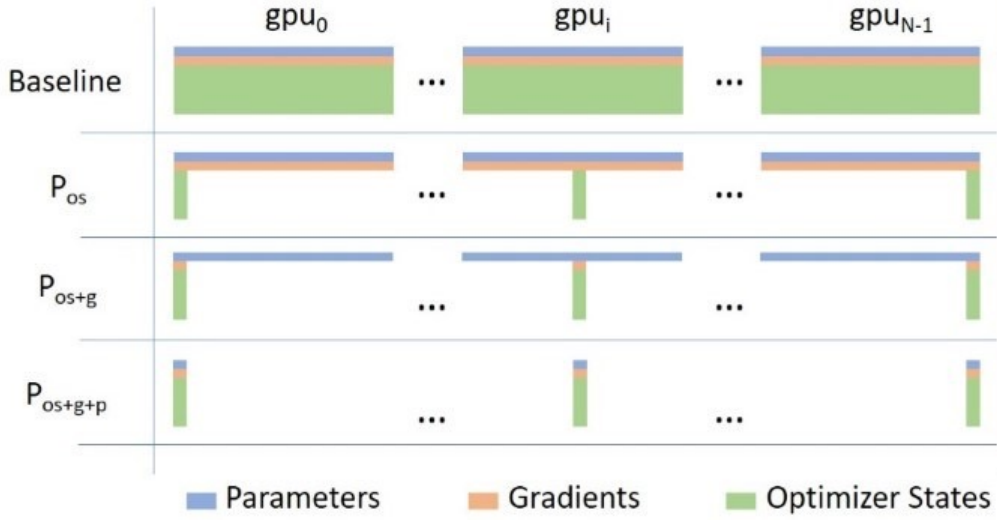
- **Core Idea:**
  - To Split up the expensive parts (Model states) and use Reduce-Scatter equivalence
    => ZeRO removes the redundancy across data parallel processes
- Partition optimizer states, gradients and parameters (was implemented in 3 stages)



Stage 2 (P_{os+g})

ZeRO-DP: ZeRO powered Data Parallelism

- **Core Idea:**
  - To Split up the expensive parts (Model states) and use Reduce-Scatter equivalence
    => ZeRO removes the redundancy across data parallel processes
- Partition optimizer states, gradients and parameters (was implemented in 3 stages)

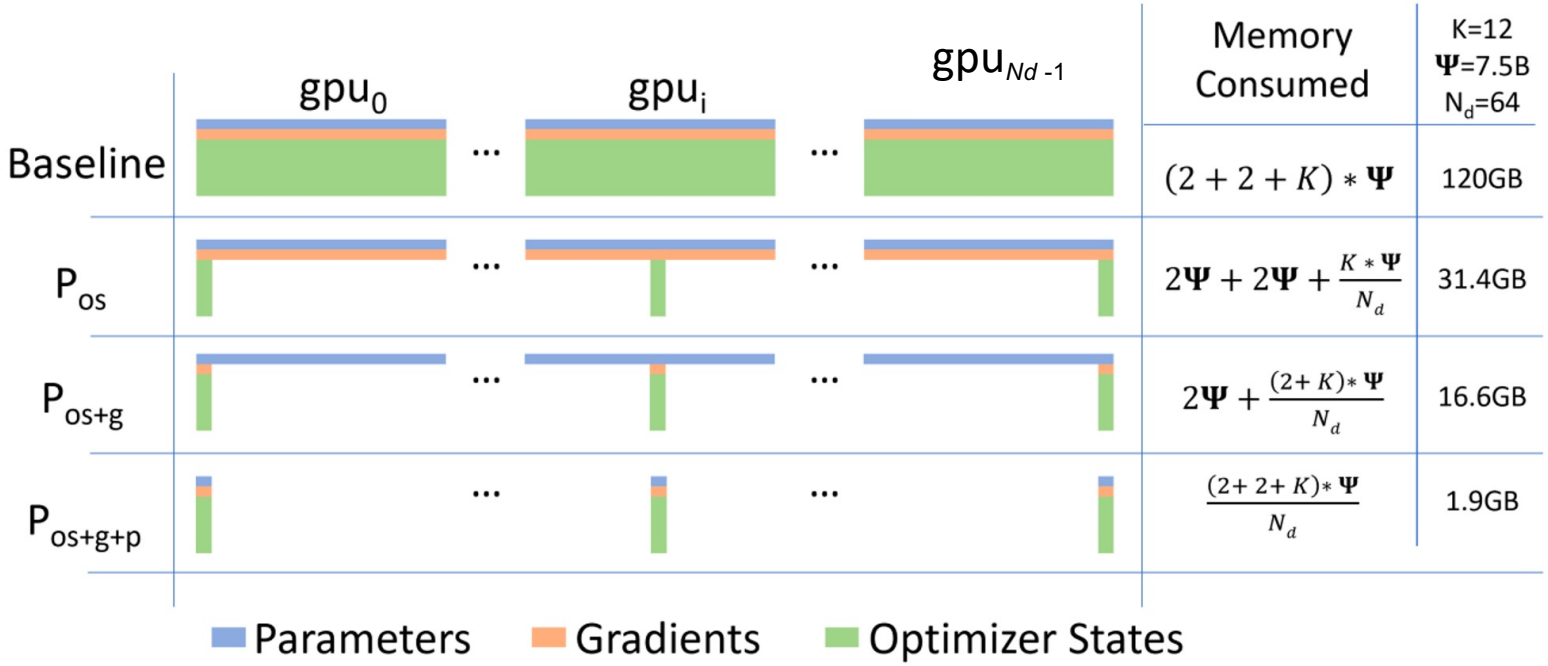


Stage 3 (P_{os+g+p})

ZeRO-DP = ZeRO Stage 3 (aka FSDP – Fully Shard DP)

Core Idea:

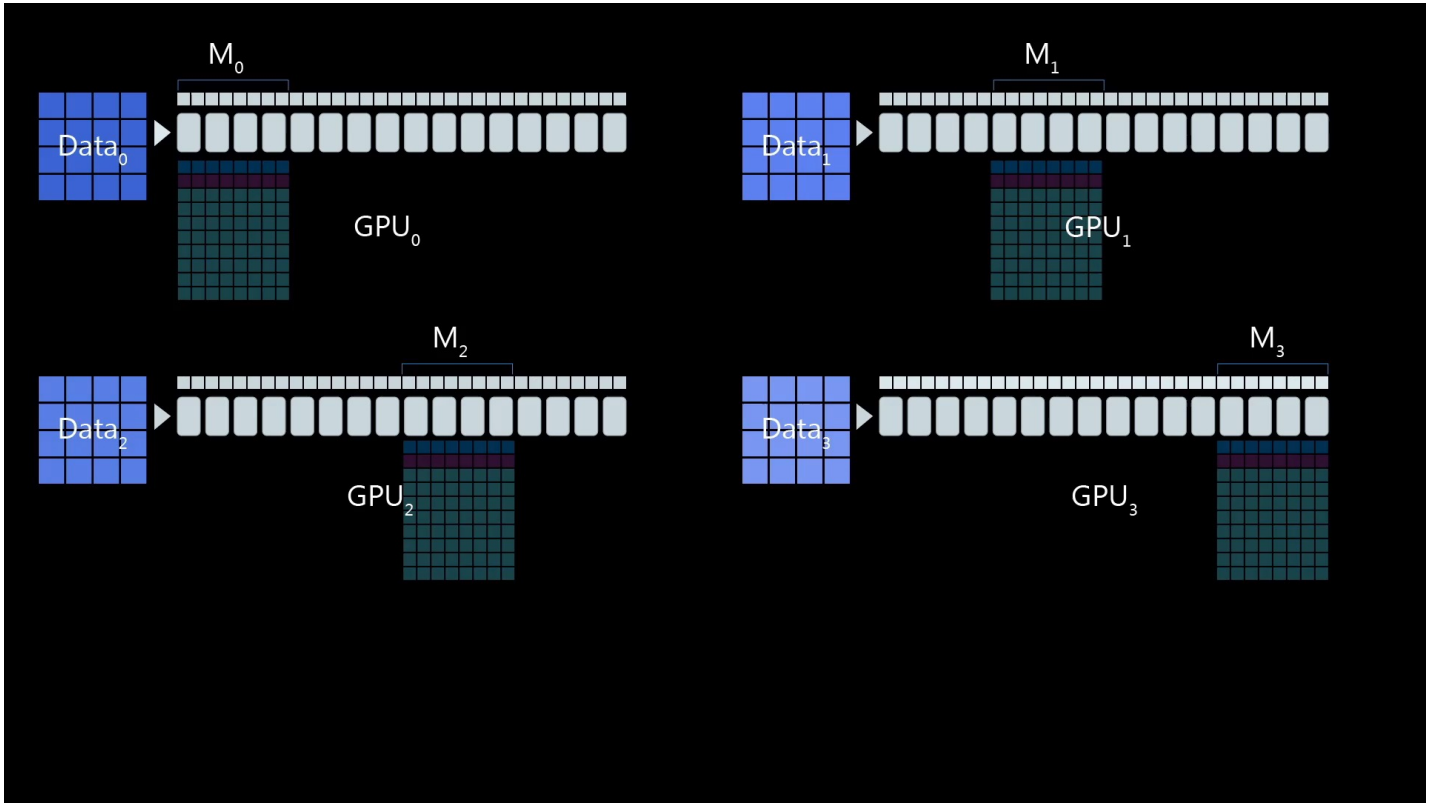
- To Split up the expensive parts (Model states) and use Reduce-Scatter equivalence

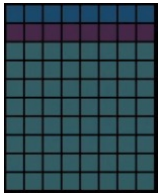


ZeRO-DP in Action

https://youtu.be/czgA-MbAdvA?si=dppN5q8KjZFOK08F&t=2455

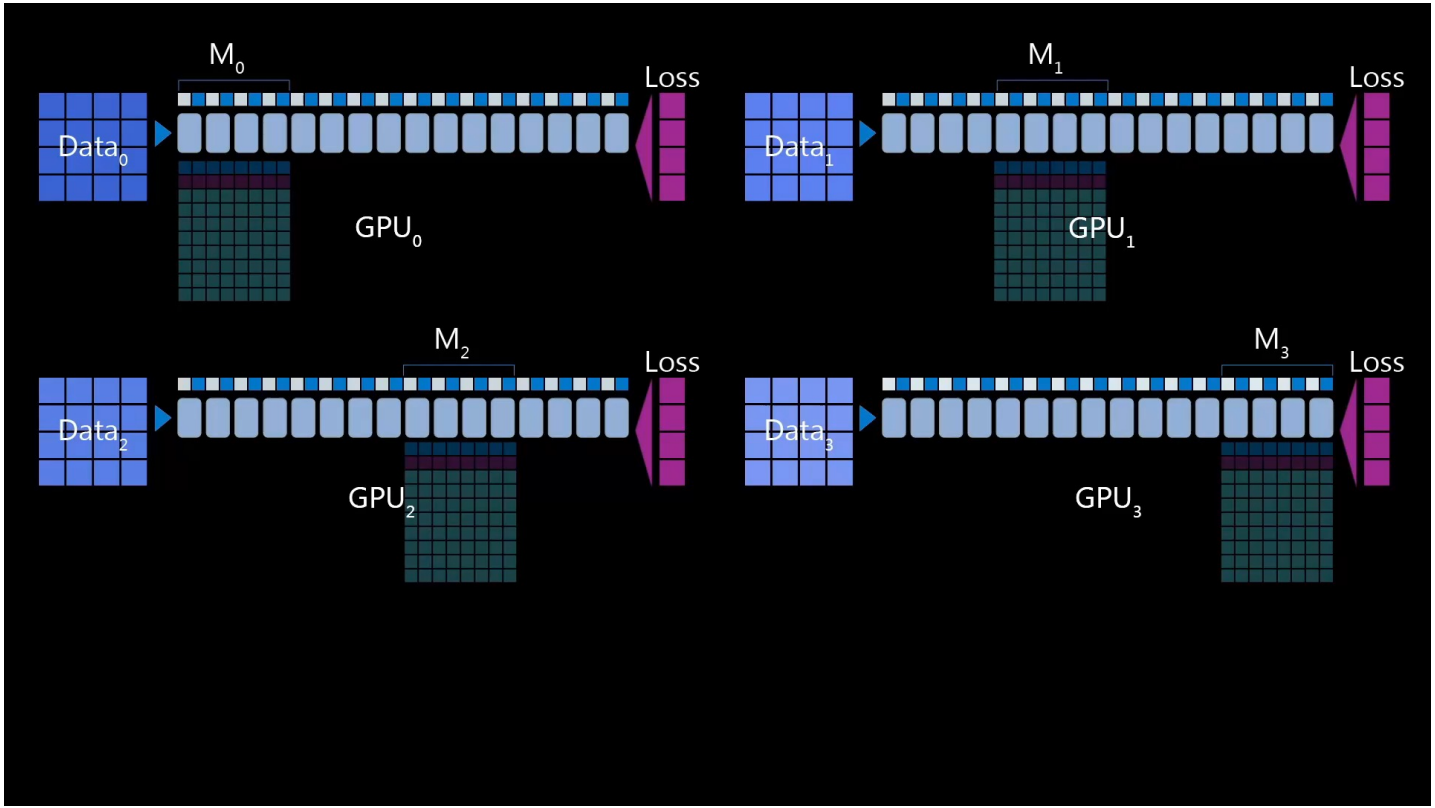
ZeRO-DP : Stage 3 Forward Propagation

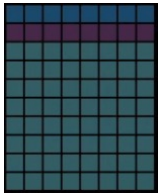




fp16 params
fp16 grads
fp32 grads
fp32 momentum
fp32 variance
fp32 params

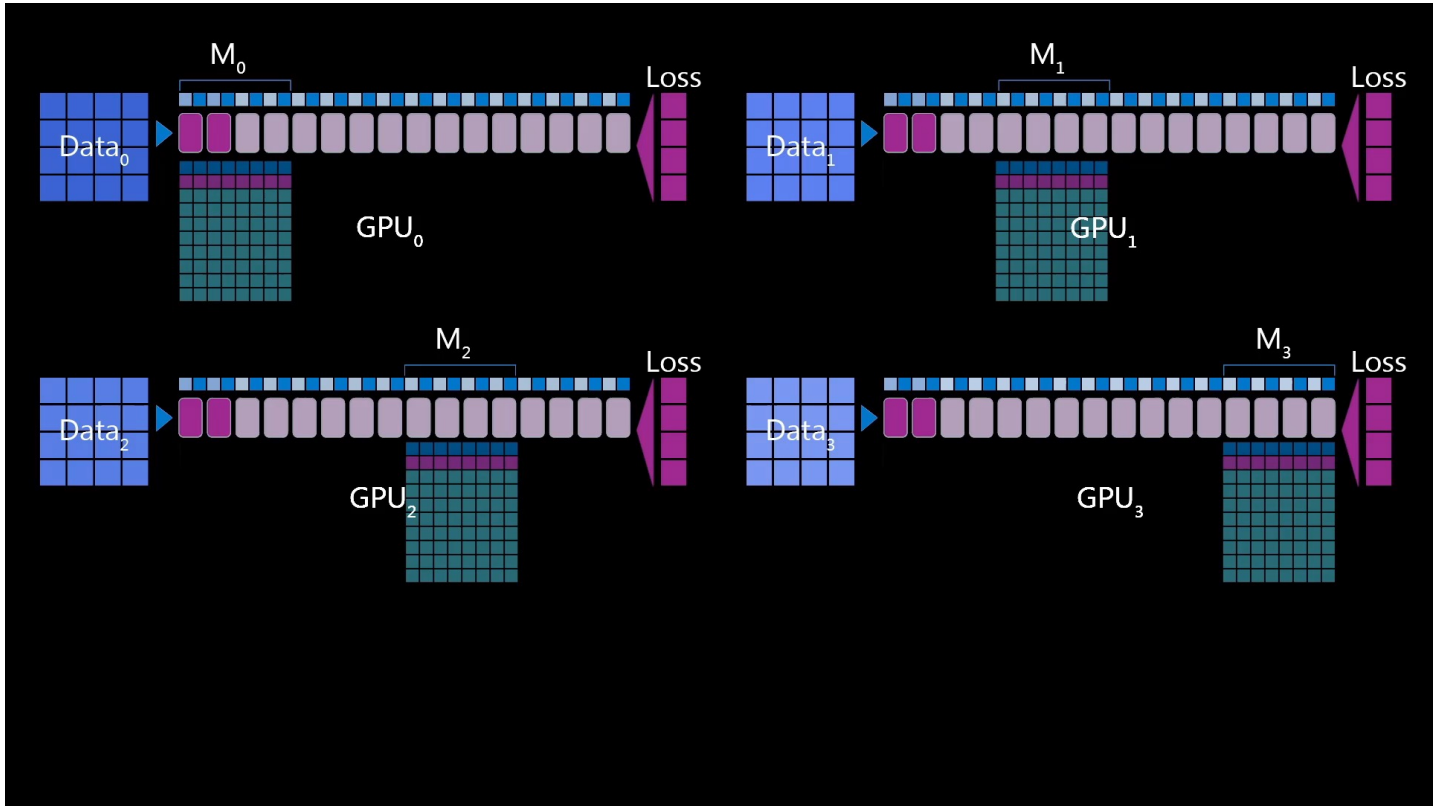
ZeRO-DP : Stage 3 Backward Propagation

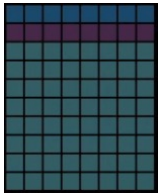




fp16 params
fp16 grads
fp32 grads
fp32 momentum
fp32 variance
fp32 params

ZeRO-DP : Stage 3 Optimizer Step



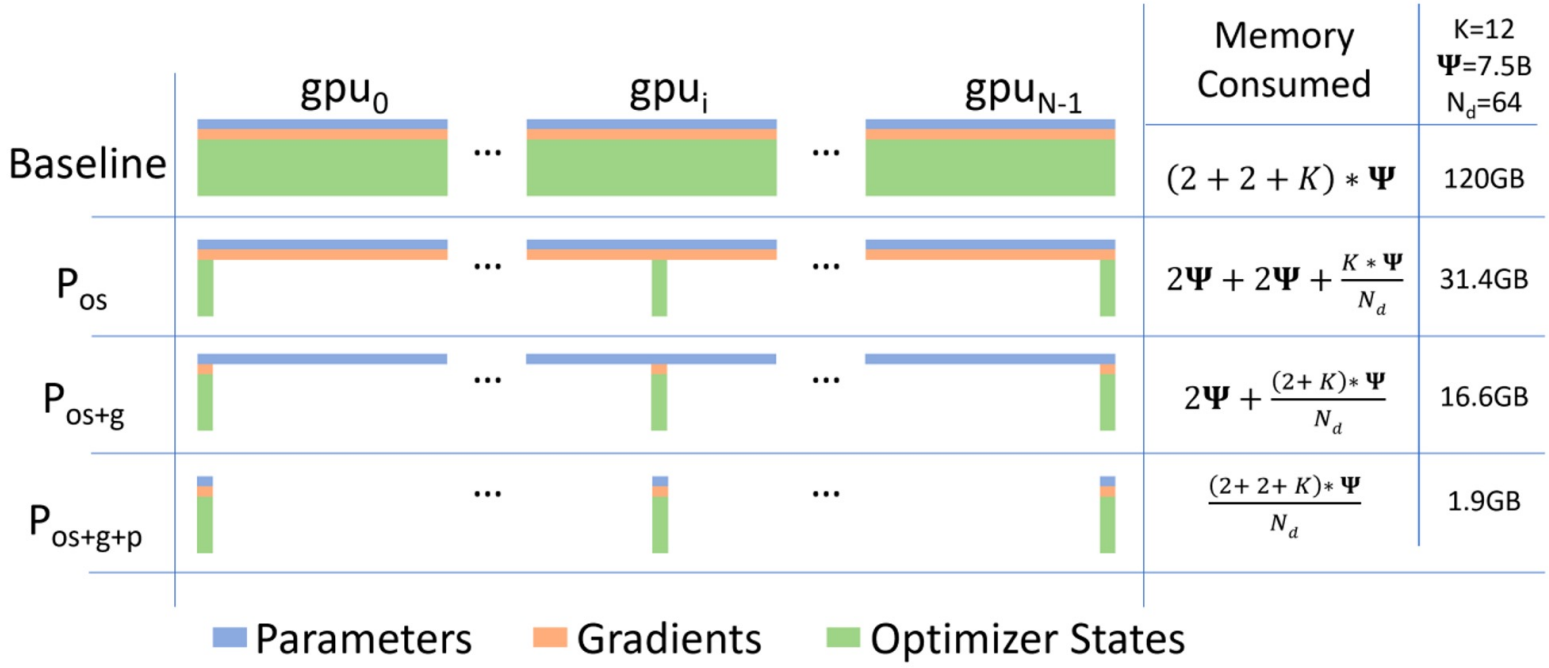


fp16 params
fp16 grads
fp32 grads
fp32 momentum
fp32 variance
fp32 params

Summary of ZeRO-DP:

Core Idea:

- To Split up the expensive parts (Model states) and use Reduce-Scatter equivalence



ZeRO-DP: Memory Savings vs. Communication Volume

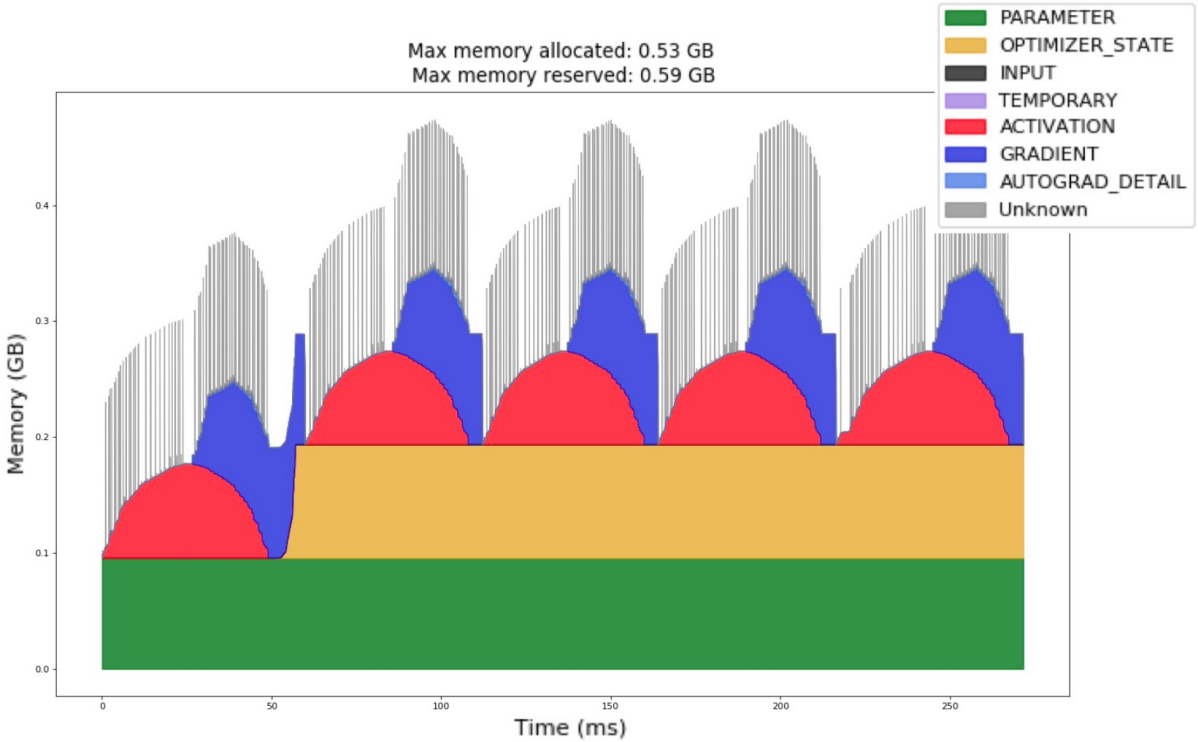
	Memory Reduction with N GPUs	Max params with ZeRO only (in billions)	Max params with ZeRO + Model Parallelism (in billions)	Comm Volume
Data Parallel	1x	1.2	20	1x
ZeRO Stage 1 (P_{os})	4x	6	100	1x
ZeRO Stage 2 (P_{os+g})	8x	13	200	1x
ZeRO Stage 3 (P_{os+g+p})	Nx	>1000	>1000	1.5x

- That's how ZeRO Stage-3 == Zero-DP optimizes model states

*Mixed precision Adam on Cluster of DGX-2 with NVIDIA 32 GB V100 GPUs

More Complexity: GPU memory need is Dynamic

- Memory is NOT just the static bits, but also Activations:



Next: Optimizing Residual States Memory with ZeRO-R

- Let’s talk about Residual States Memory with Zero-R
  - Activation
  - Temporary buffer resizes
  - Segmented memory

ZeRO-R

- Activation
  - 1.5B GPT-2, sequence length = 1K, batch size = 32
  - **60 GB memory**
  - Activation checkpointing
    - Clearing activation of some layers and recompute them in backward pass.
    - 8GB memory + 33% re-computation
- Solution
  - Intelligently pick which activations to store.
  - Partition activation across memory-parallel processes
    - All-gather
  - Memory offload when limited

ZeRO-R

- Temporary Buffers
  - Store intermediate results
  - Eg: Fuse all gradients into a single flattened buffer
    - Before applying operations
  - Usually fp32 tensor
  - 1.5B model, fp32 flattened buffer -> 6GB memory
- Solution:
  - Use a constant size, performance efficient fused buffer
  - Trade-off: smaller buffer size, slightly slower computation.

ZeRO-R

- Memory fragmentation
  - Memory fragmentation common for training LLMs
  - Occurs as a result of activation checkpointing
    - Store selected activations for backpropagation
    - Most can be recomputed during back propagation
    - Interleaving discarded activations and other activations
  - Requests can fail when no contiguous memory available.
  - 30% memory available yet OOM.
- Solution:
  - Classic system techniques:
    - Perform defragmentation on the fly
    - Preallocate contiguous memory chunks

Actual Picture of How FSDP / ZeRO Stage 3 works

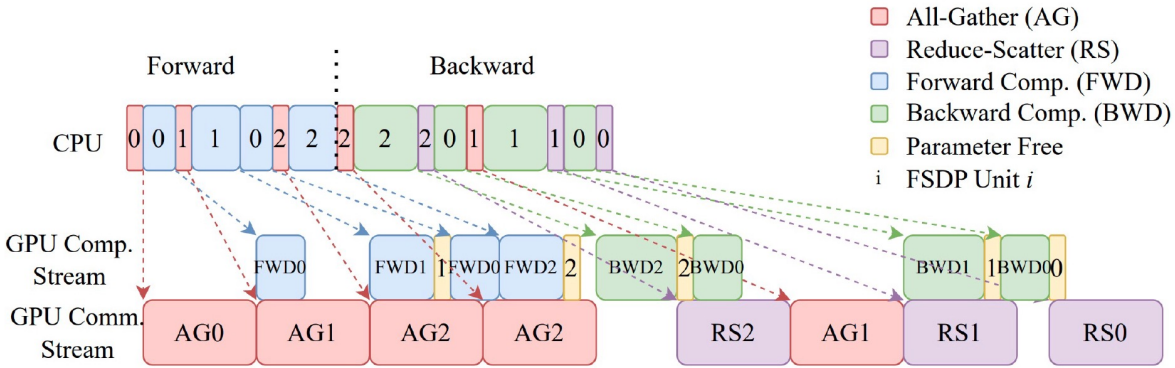
Let's walk through a FSDP example to see some important ideas

Incremental computation / communication

- Parameters / gradients are requested / sent and then immediately freed

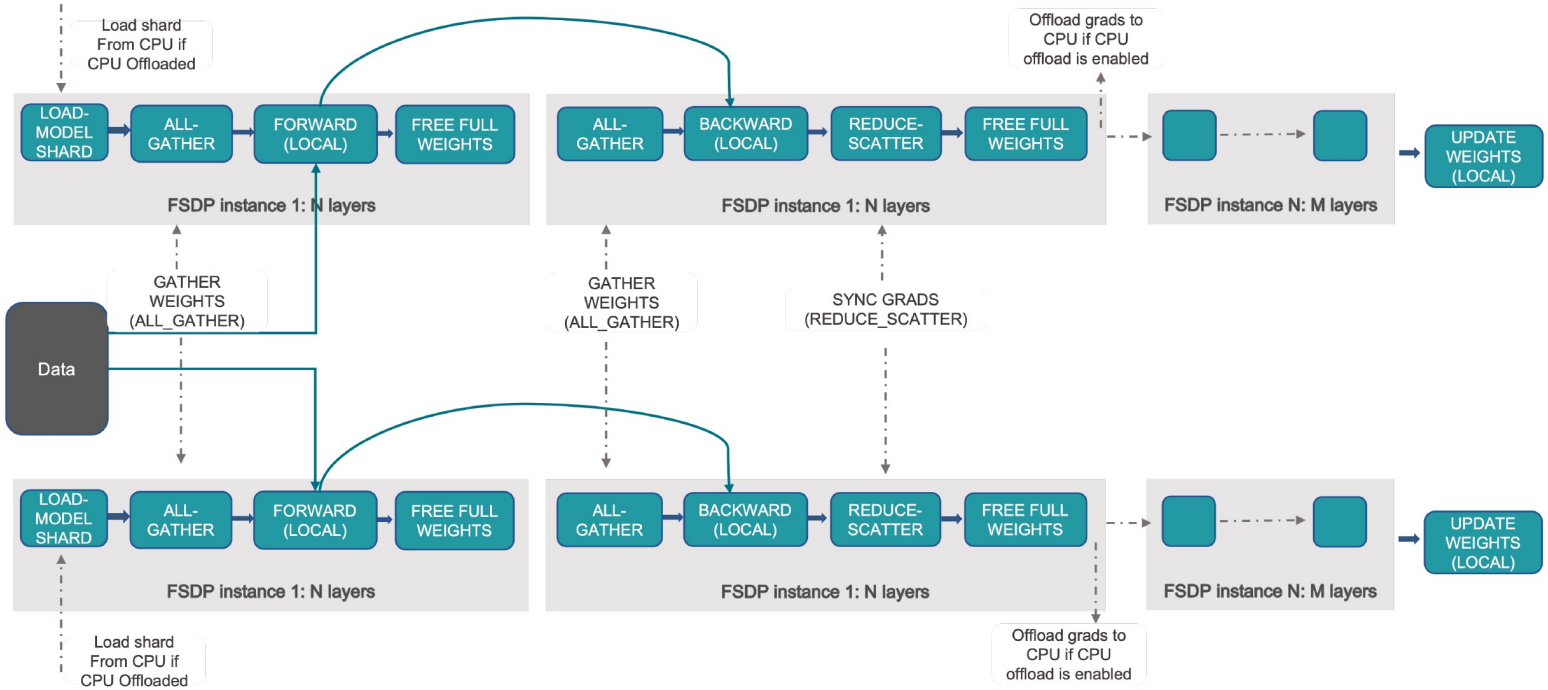
Overlapping communication and computation

- The all-gathers happen all at once while forward happens, masking the comm cost.



https://arxiv.org/pdf/2304.11277.pdf

How ZeRO Stage 3 (aka FSDP – Fully Shard DP) works ?

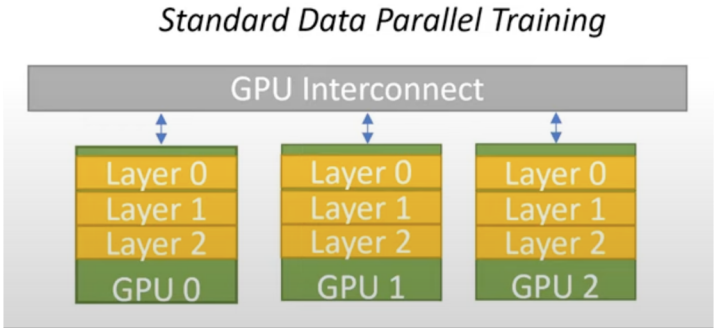


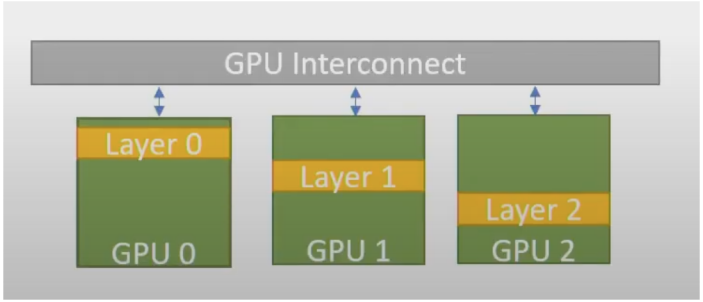
Communication cost – 2 all gather (#param), 1 reduce-scatter (#param).

https://pytorch.org/tutorials/intermediate/FSDP_tutorial.html

Summary of ZeRO so far:

- Zero-DP: optimizes model states
  - Existing DP replicates model states at each device.
    - Significant memory overhead
    - OOM on 32GB for 1.4B model
- Zero-R: reduces residual memory consumption.
  - Activation
    - Activation checkpointing
  - Temporary buffer resizes
  - Segmented memory





ZeRO in Practice: Will it fit ?

In 2024 – pure BF16 training (with Kahan summation), is viable optimizer states are less beefy. Let's say BF16 for everything but the master weights – 12 bytes per param

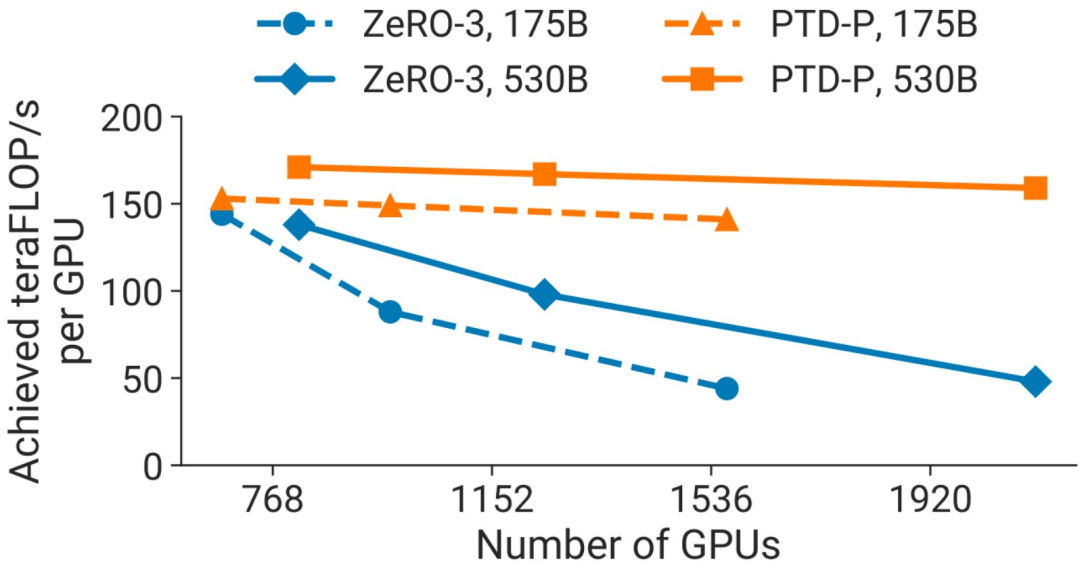
On a 8X A100 80G..

	Max size (params)	Formula for B/param
Baseline	6.66.. B	12
Zero stage 1	16 B	5
Zero stage 2	24.62 B	2 (param) + (10 (grad+state))/8)
Zero stage 3	53.33 B	12/8

Issues remain with Data Parallel – Large Models don't Fit

Zero stages 1 and 2 don't let you scale memory

Zero stage 3 is nice in principle, but slow in practice



Better ways to split up the model is needed…

Beyond Data Parallel – Model Parallelism

Scaling up in memory (without changing batch size) with model parallelism

What model parallelism is..

- It splits up the parameters across GPUs (like zero3)..
- But communicate activations (while zero3 sends params).

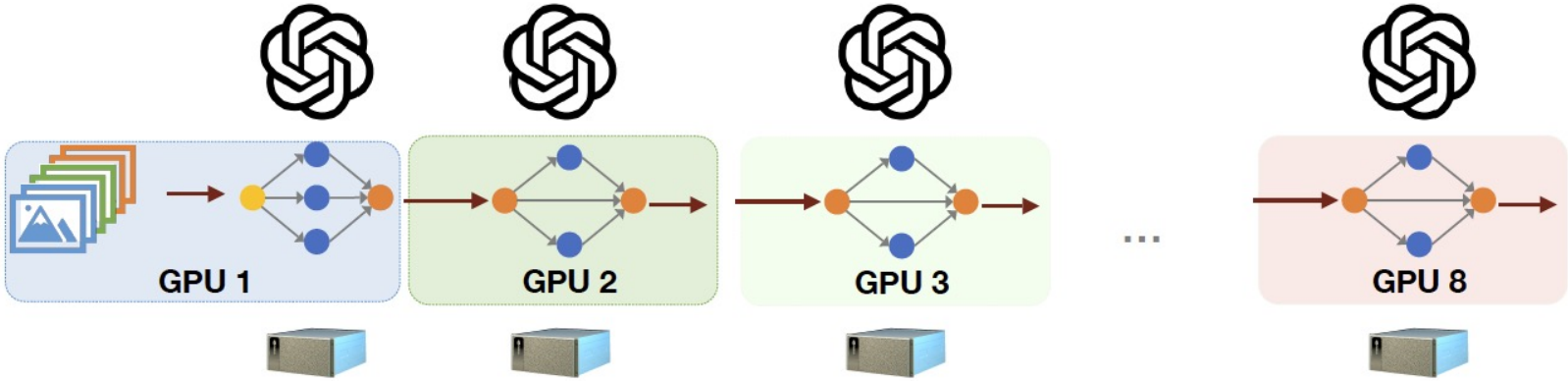
We cover 3 different types of parallelism

1. Pipeline parallel
2. Tensor parallel
3. Sequence parallel

These correspond to 3 different ways of cutting up the model.

Pipeline Parallelism

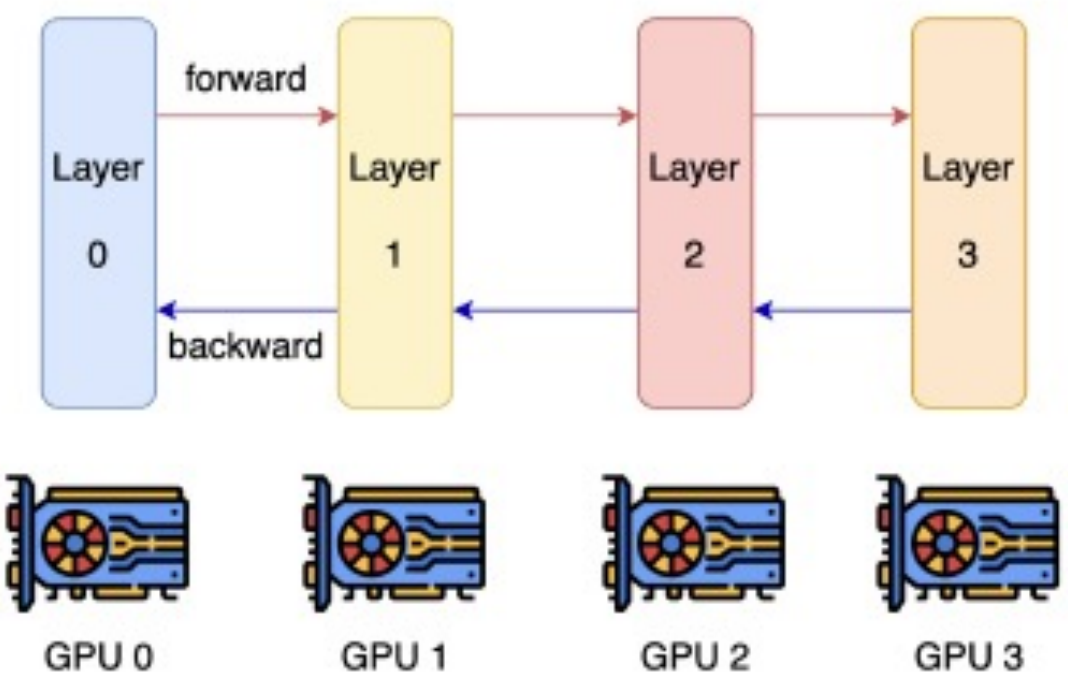
Instead of splitting the data, pipeline parallelism splits the model



350GB / 8 cards = 43.75G < 80G

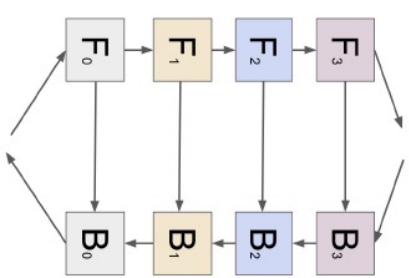
With Pipeline Parallelism, large ML models can be placed and trained on GPUs.

Pipeline Parallelism == Layer-wise Parallel

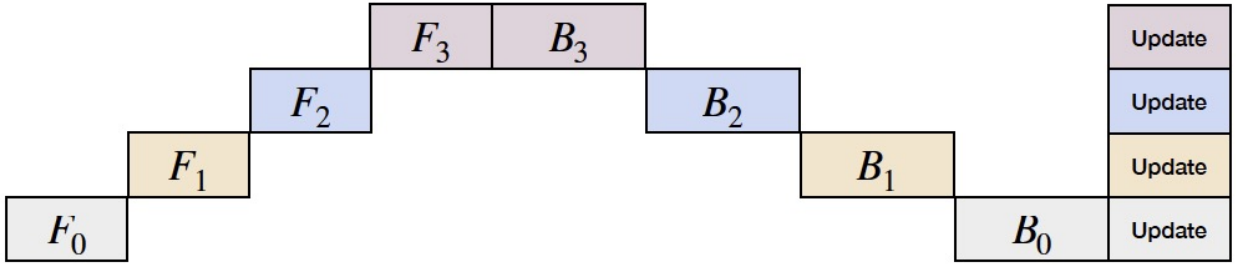


- Layer-wise Parallel (aka vertical parallelism) partition the layers of the model ; assign some subset to each GPU
- Activations and Partial Gradients are passed forth and back

Naïve Implementation of Pipeline Parallelism Workflow



(a). Training data flow



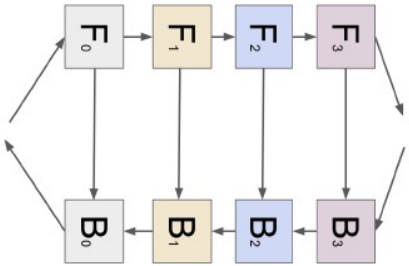
(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with Pipeline Parallelism.

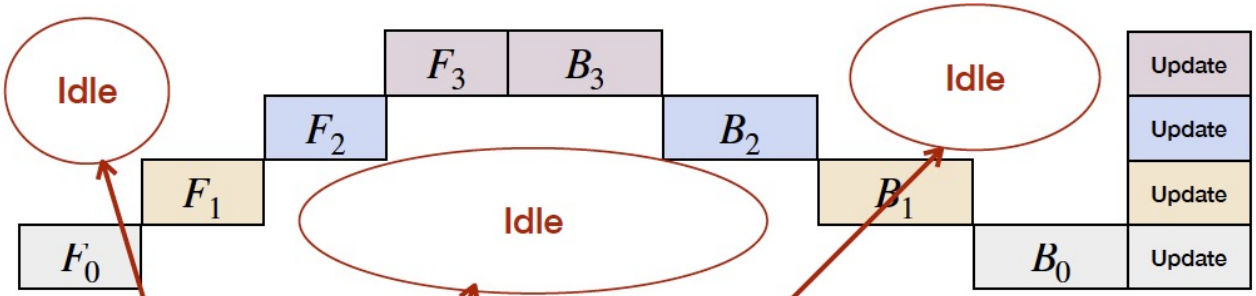
Pipeline Parallelism is needed for training a bigger DNN model by dividing the model into partitions and assigning different partitions to different accelerators.

GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism [Huang et al. 2018]

Naïve Implementation of Pipeline Parallelism Workflow



(a). Training data flow



(b). Training timeline

F: Forward B: Backward. Train a 4 layer network with Pipeline Parallelism.

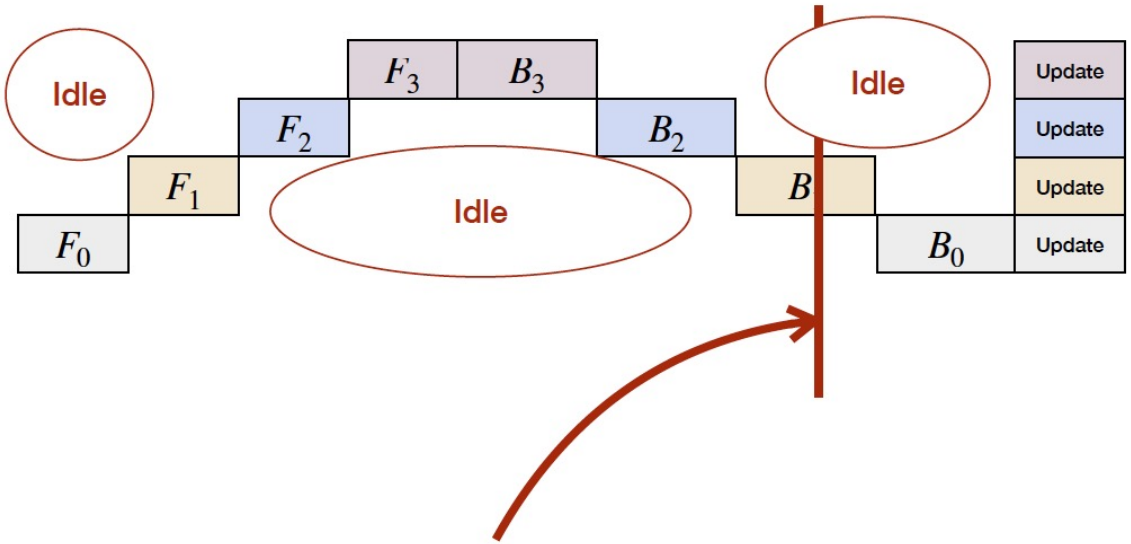
Pipeline Parallelism is needed for training a bigger DNN model by dividing the model into partitions and assigning different partitions to different accelerators.

But accelerators are significantly **under-utilized** during Pipeline Parallelism!

Terrible Scaling:

With P GPUs, each GPU is only active 1/P of the time ; mostly waiting for the Backward Pass to propagate back !

Naïve Pipeline Parallelism suffers from Utilization



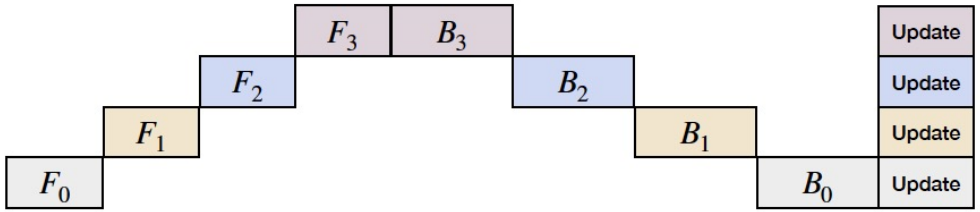
Only one device is computing at a time and others are waiting for it.

Theoretical utilization (in this 4 layer NN): 25% (**low!**)

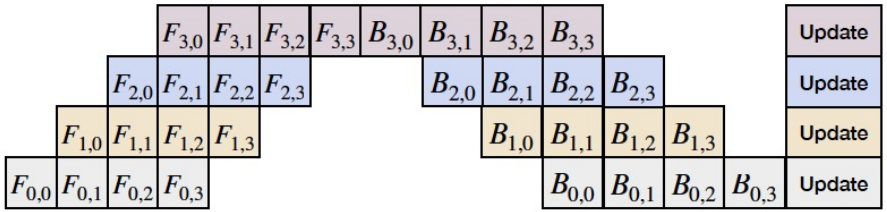
Terrible Scaling:

With P GPUs, each GPU is only active 1/P of the time ; mostly waiting for the Backward Pass to propagate back !

GPipe: Simple Scaling with Micro-batch Pipeline Parallelism



(a). Naive Pipeline Parallelism

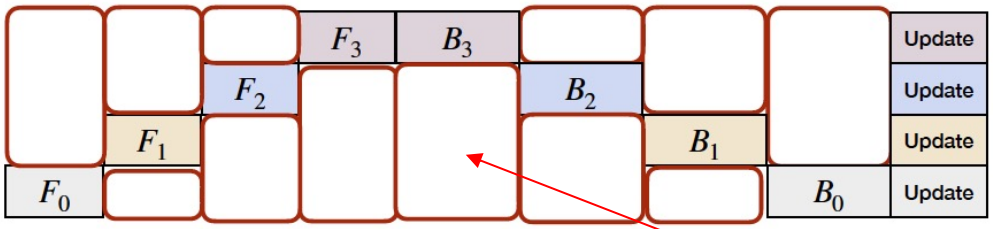


(b). Improved Pipeline parallelism

- Split a single batch to micro batches
  - [16, 10, 512] ->
    - [4, 10, 512]
    - [4, 10, 512]
    - [4, 10, 512]
    - [4, 10, 512]
- Motivation: model parameters are not changed during computation within a batch, thus we can pipeline computation and communication

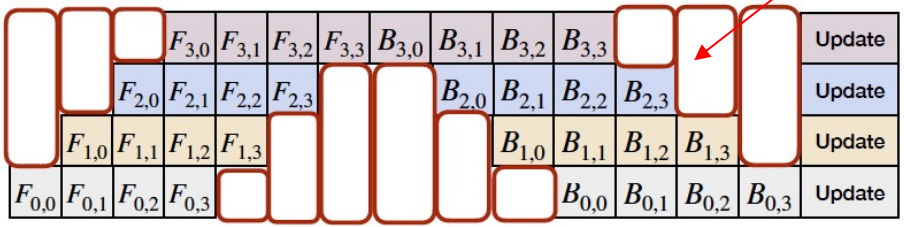
Process 'micro-batches' (in this case, 4).
Send off the first microbatch and start computing the second.

Simple Micro-batching can improve Devices Utilization



(a). Naive Pipeline Parallelism

Bubbles !



(b). Pipeline parallelism

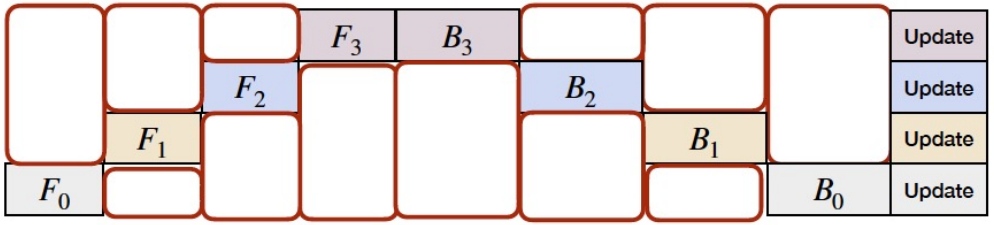
Utilization: (25%)

The more chunks (#num of micro batches), the higher of device utilization.

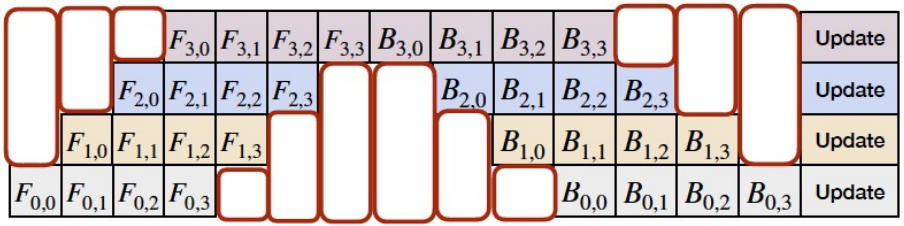
Utilization: **57%** (2.5x improvement)

The ratio of bubble time to useful compute is .. $\frac{n_{stages}-1}{n_{micro}}$ so we need a big batch size!

Micro-batching improves Devices Utilization, but still…



(a). Naive Pipeline Parallelism



(b). Pipeline parallelism

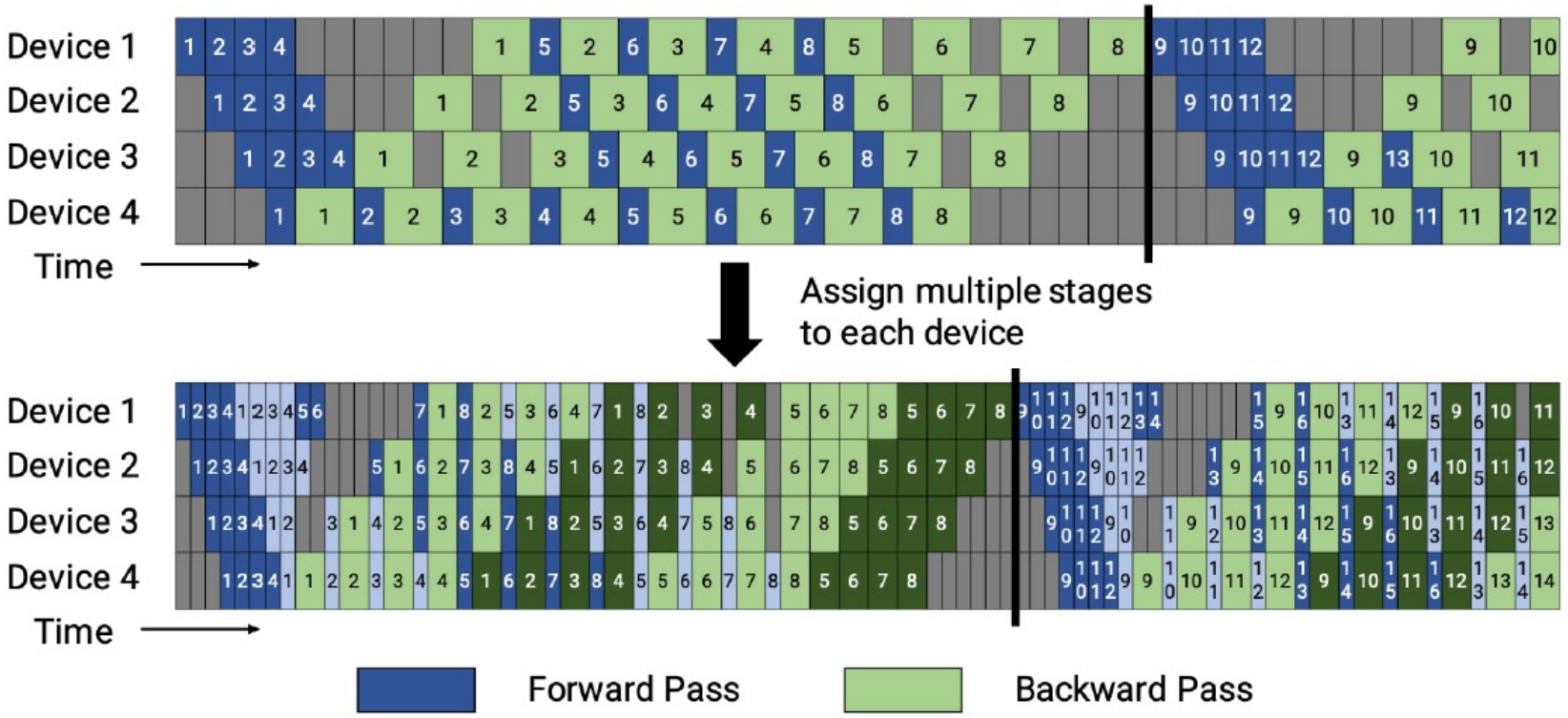
Utilization: (25%)

The more chunks (#num of micro batches), the higher of device utilization.

Utilization: **57%** (2.5x improvement)

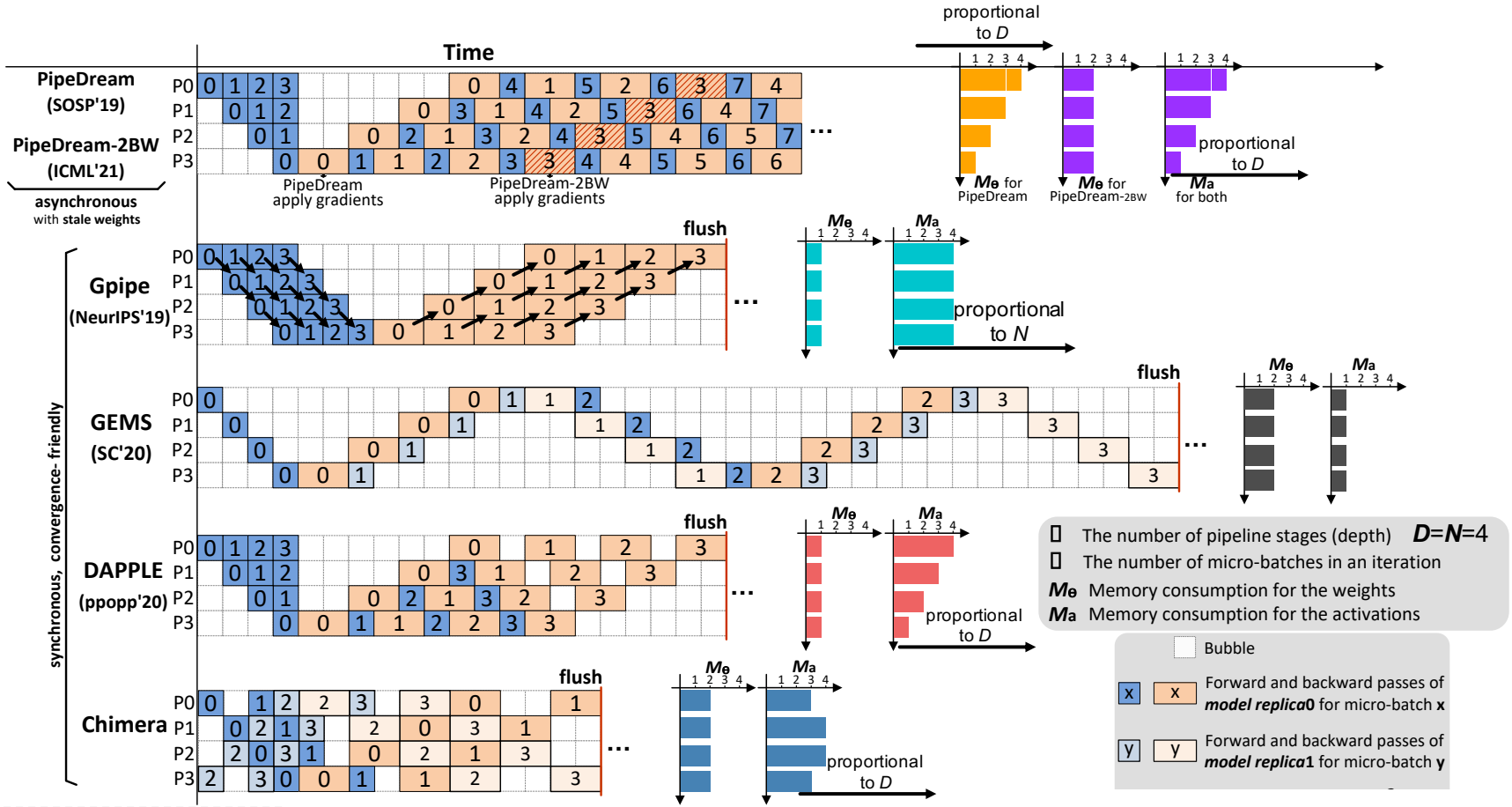
GPU Memory is still a concern: Activations still occupy GPU memory while waiting for Backward Pass to be back => Can be dealt with by checkpointing and/or freeing Activation memory and reload and/or recompute them later at the expense of extra loading delay/ computation!

Trading Communication Bandwidth for GPU utilization



More Complicated Pipeline patterns, e.g. PipeDream [SOSP 2019]
can improve utilization but at the expense of Communication Bandwidth and Complexity !

Summary on Pipeline Parallelism Research Schemes

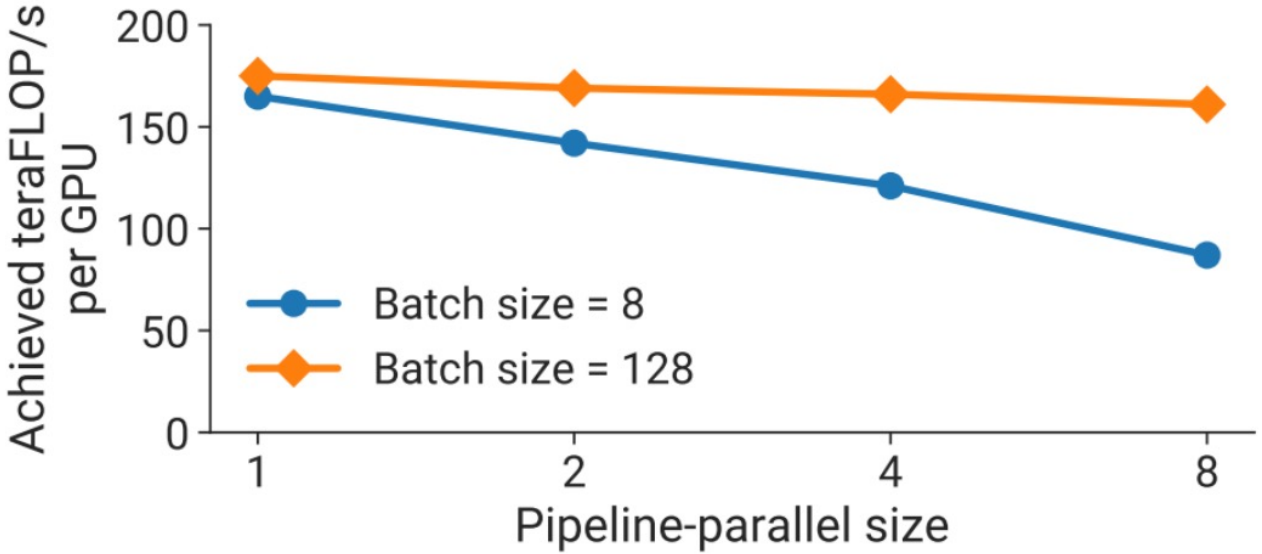


Slide: Courtesy of Shigang Li

PipeDream

- PipeDream provides a more flexible and efficient pipeline parallelism approach compared to GPipe.
- PipeDream combines model parallelism and data parallelism in a pipelined fashion to partition and schedule the training job into stages over a network of devices.
- To support pipelining for DNN jobs, PipeDream resolves 3 challenges that traditional hardware pipeline doesn't see.
    - Dynamic Programming algorithm to partition workload evenly
    - Static scheduling policy to coordinate forward and backward pass
    - Weight stashing to improve statistical efficiency
- Able to achieve around 2x higher throughput than GPipe; 1.9x faster than FlexFlow on AlexNet

Pipeline Parallelism Performance Highly Dependent on Batch Size



Batch sizes are key to hiding the bubble – otherwise pipeline rapidly degrades perf

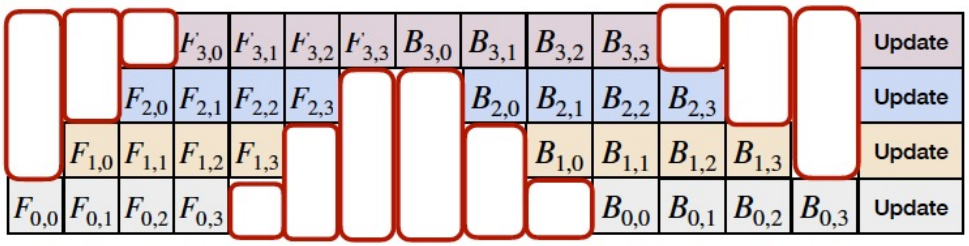
Why do we still do Pipeline Parallelism ?

- Pipelines save GPU memory (compared to Data Parallelism)
- Pipelines can have good communication properties (compared to Tensor Parallelism and Fully Sharded Data Parallel (FSDP) as it depends only on Activations and is Point-to-Point !
- Require special handling of bubble that results in idle processes

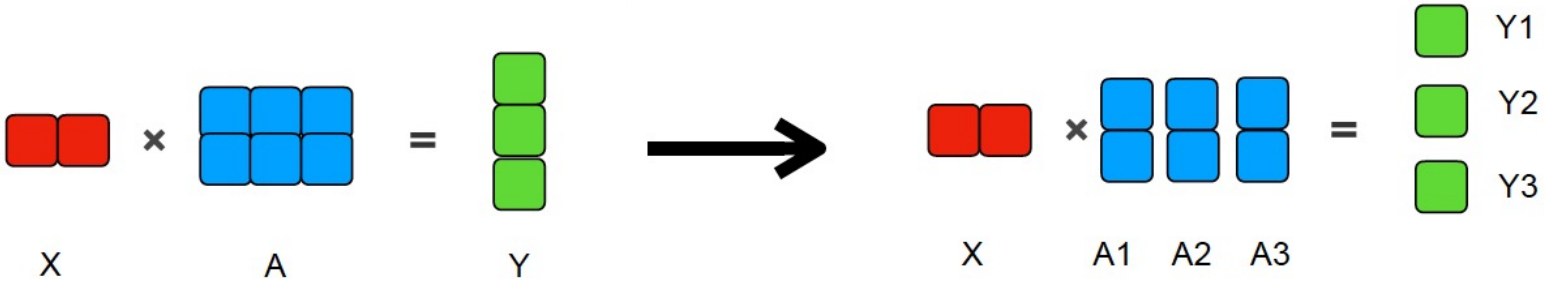
=>

Generally, Pipeline Parallelism is used on Slower Network Links (i.e. Inter-node) as a way to get better memory-wise scaling !

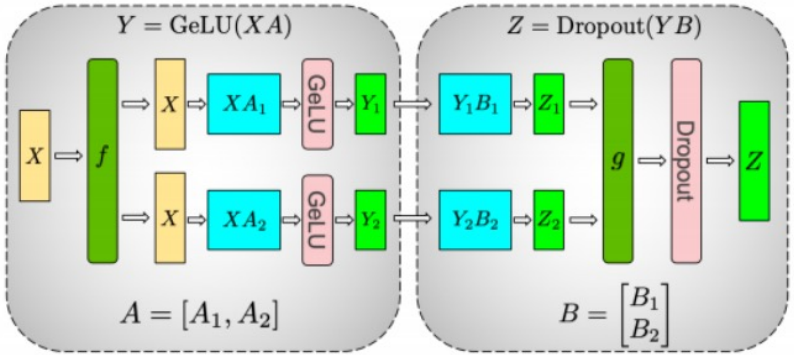
Tensor Parallelism



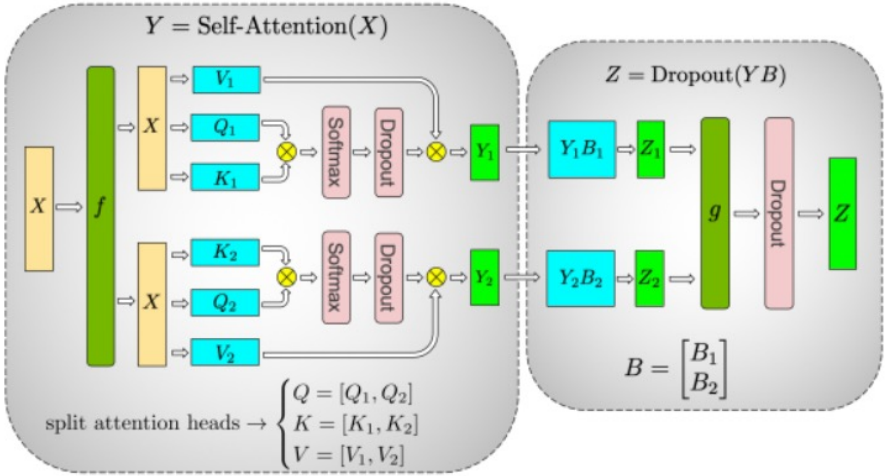
- The optimized pipeline parallelism still leads to idle time of GPUs, can we further improve it (while keeping memory usage low)?
- Make the model partition ***more fine-grained***!



Tensor Parallelism



(a) MLP

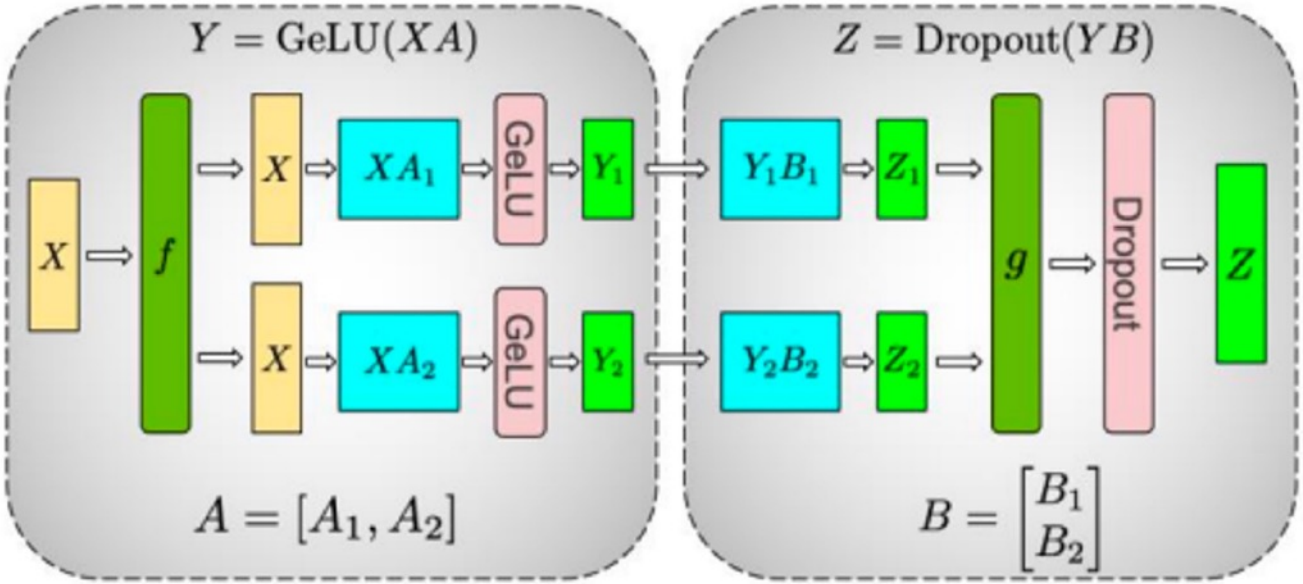


(b) Self-Attention

- Tensor parallelism: split a weight tensor into N chunks and parallelize
- f and g denote the AllReduce / Identity operations to synchronize.
- The params and activations (blue part) are spliced across different GPUs.

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

Tensor Parallelism – GPUs carry Submatrices

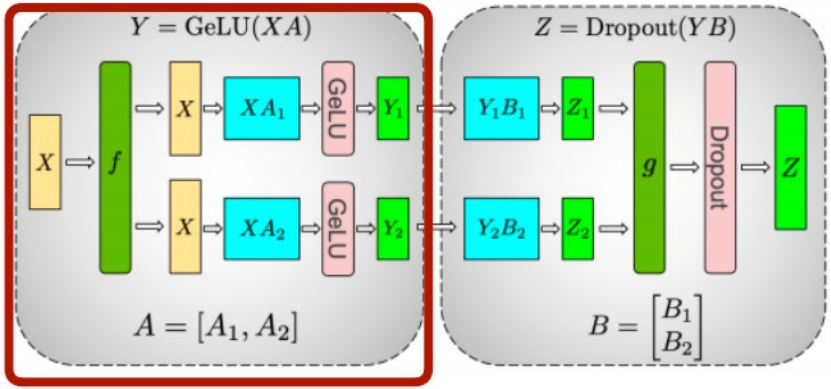


Assign columns (A1, A2) and rows (B1, B2) to separate GPUs.

- In the forward pass, f is the identity, and g is an all-reduce.
- In the backward pass, f is an all-reduce, g is the identity.

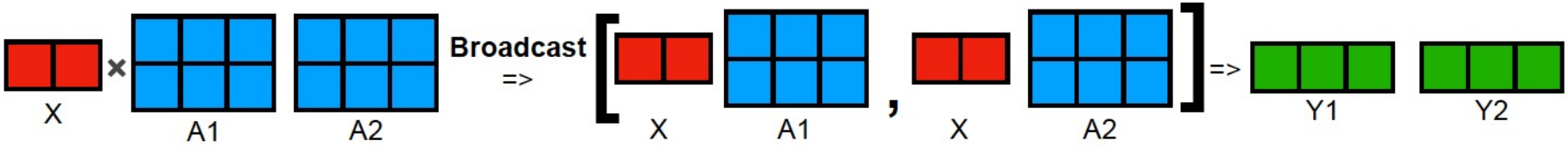
Tensor Parallelism

Partition in First FFN Layer



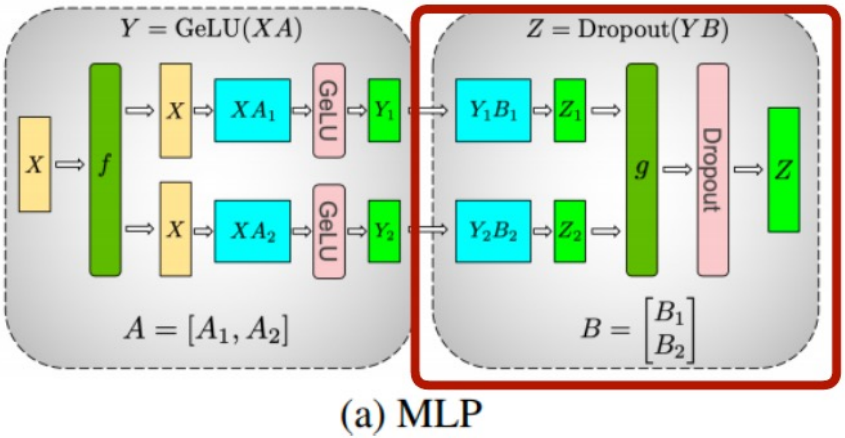
(a) MLP

- Take [1, 2] @ [2, 6] @ [6, 2] as an example
- The first Linear layer is partitioned in **column** parallel fashion.
- We use **Broadcast to duplicate input x** to all devices.



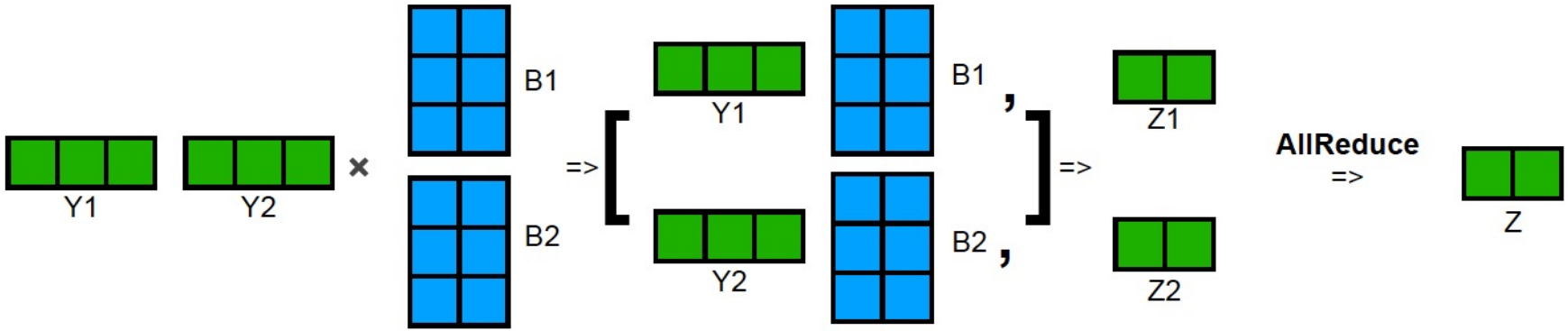
Tensor Parallelism

Partition in Second FFN Layer



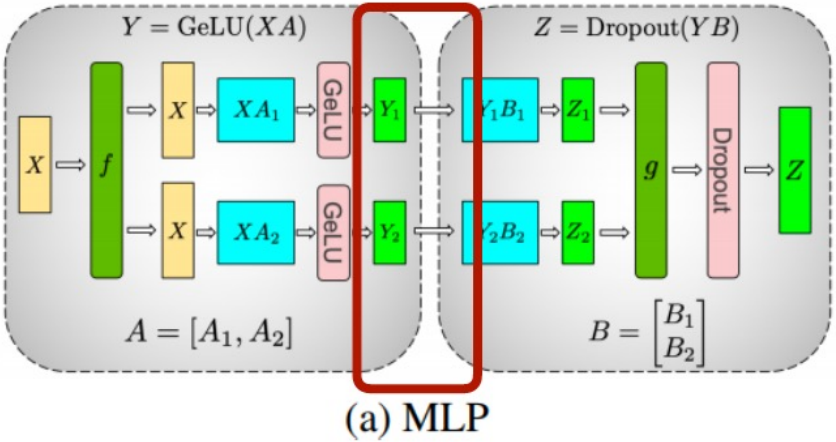
(a) MLP

- Take [1, 2] @ [2, 6] @ [6, 2] as an example
- The second Linear layer is partitioned in **row** parallel fashion.
- We **AllReduce** the output to gather outputs.



Tensor Parallelism

Partition in FFN Layers



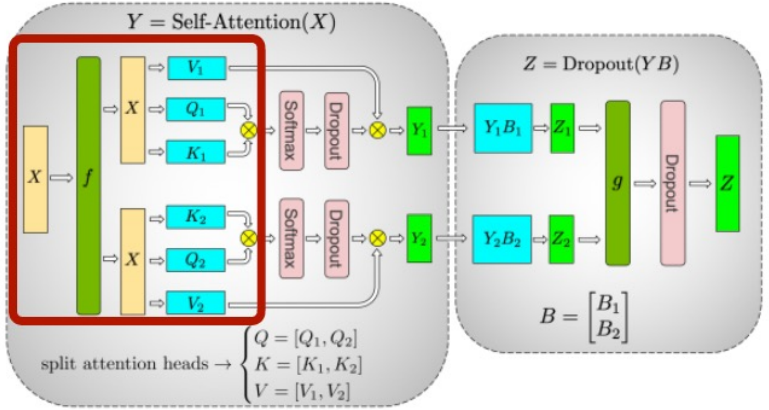
(a) MLP

- Partitioning the **first** GEMM in **column** and split the **second** GEMM along **rows**
- Thus it takes the output directly without requiring any communication
- As long as the communication is not bottleneck, GPUs can be fully utilized.

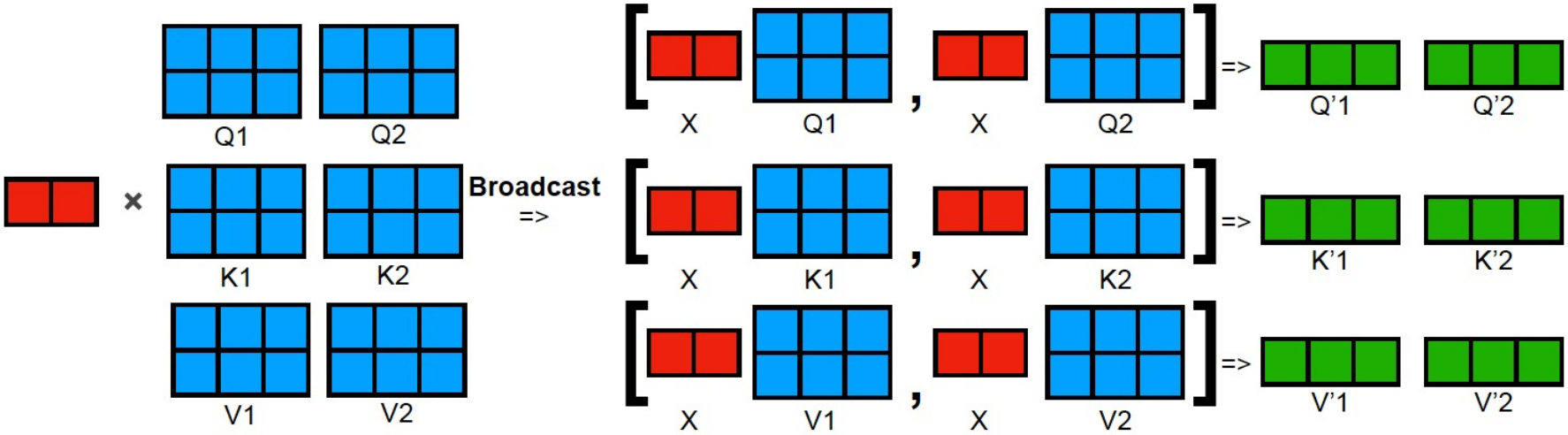
Tensor Parallelism

Partition Attention Layers: QKV Projection

- Split the QKV projection layer via **columns**.
- Input X is **Broadcasted** to all devices.



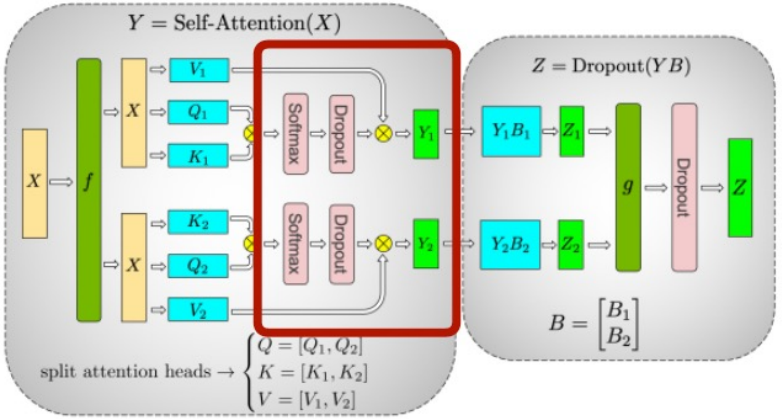
(b) Self-Attention



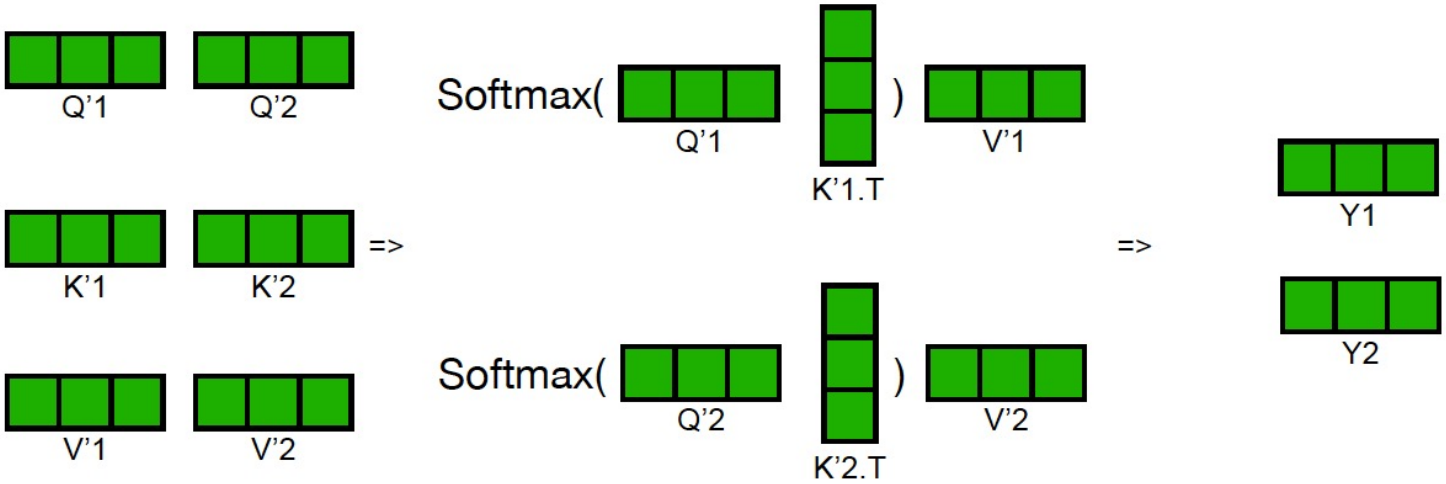
Tensor Parallelism

Partition Attention Layers: Attention

- Soft(Q@K.T)@V is performed locally with no extra communication



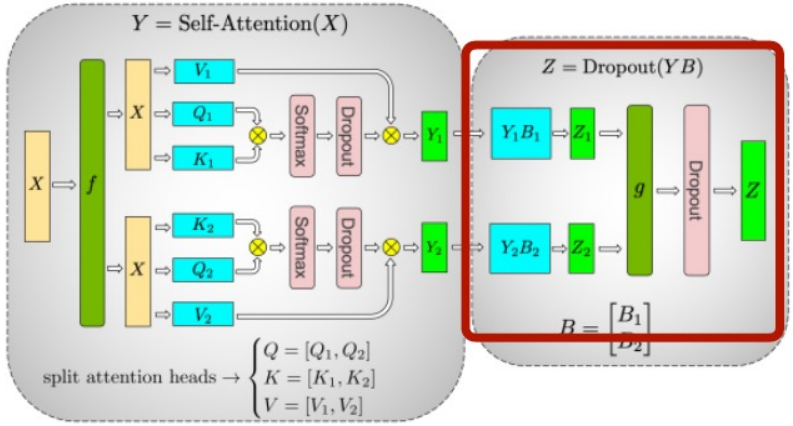
(b) Self-Attention



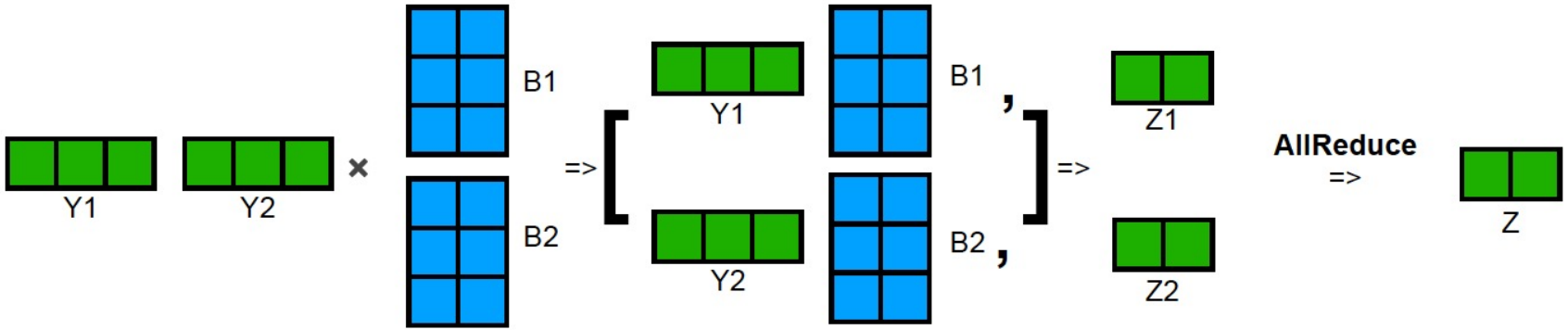
Tensor Parallelism

Partition Attention Layers: Out Projection

- OutputProjection layer is splitted by **rows**.
- The computation can be done without synchronization.
- The outputs is then collected with **AllReduce**.



(b) Self-Attention



Tensor Parallel vs. Pipeline Parallelism
Pros and Cons

Pros:

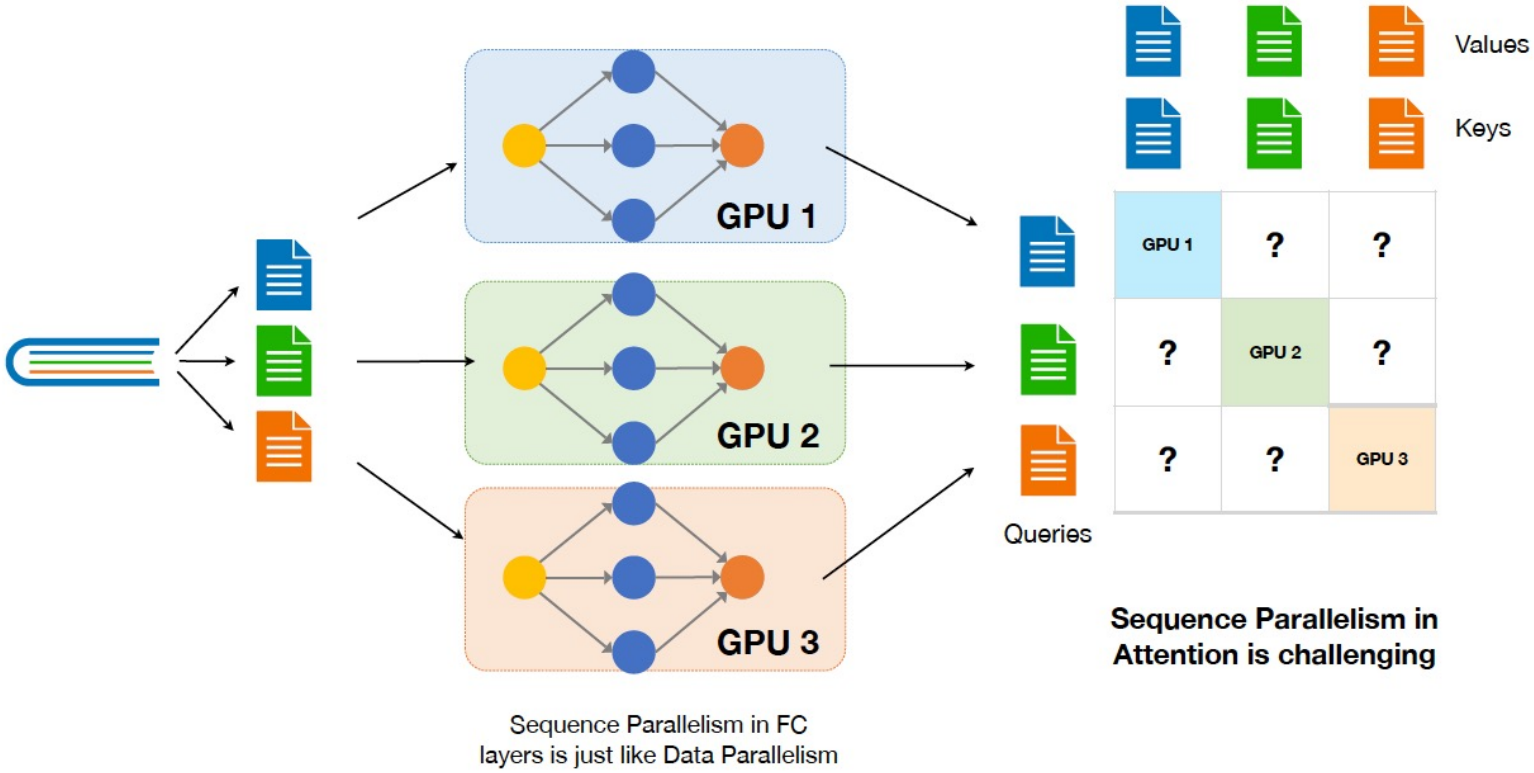
- No bubble. If the “network connection” between the GPUs is fast enough, there is no waiting for others.
- Does not need large batch sizes to work well

Cons:

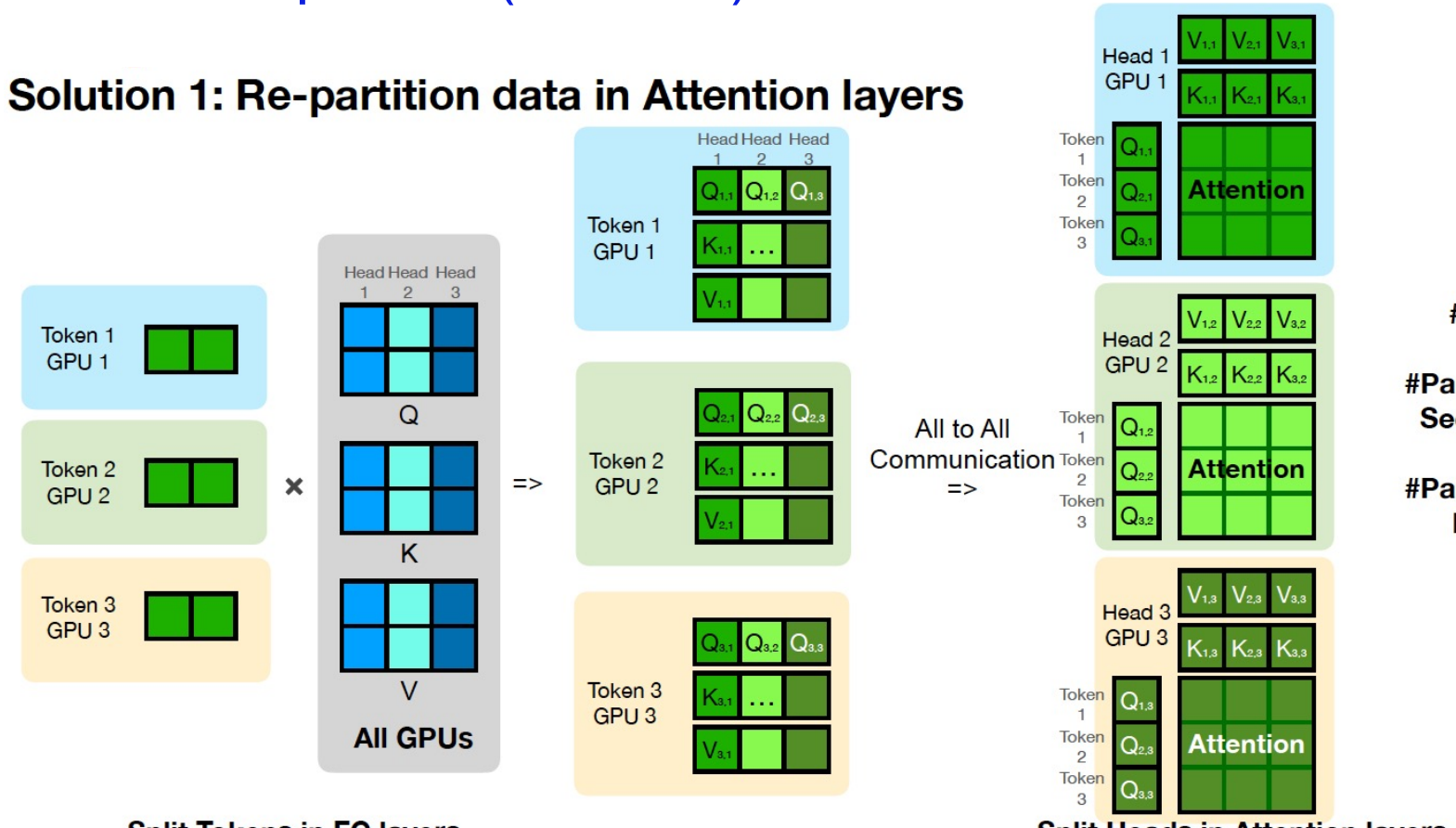
- Much higher communication demand (i.e. network bandwidth) than Pipeline Parallel
  - Pipeline //: point-to-point communication volume = *bsh* per micro-batch
  - Tensor //: 8 *bsh* * (P-1)/ P *per layer* + *AllReduce communication*

Thus, do Tensor Parallel whenever Low-latency, High-bandwidth Interconnects b/w GPUs are available, e.g. NVLink among multiple GPUs in a server (node) under the Nvidia framework !

Sequence (Context) Parallelism



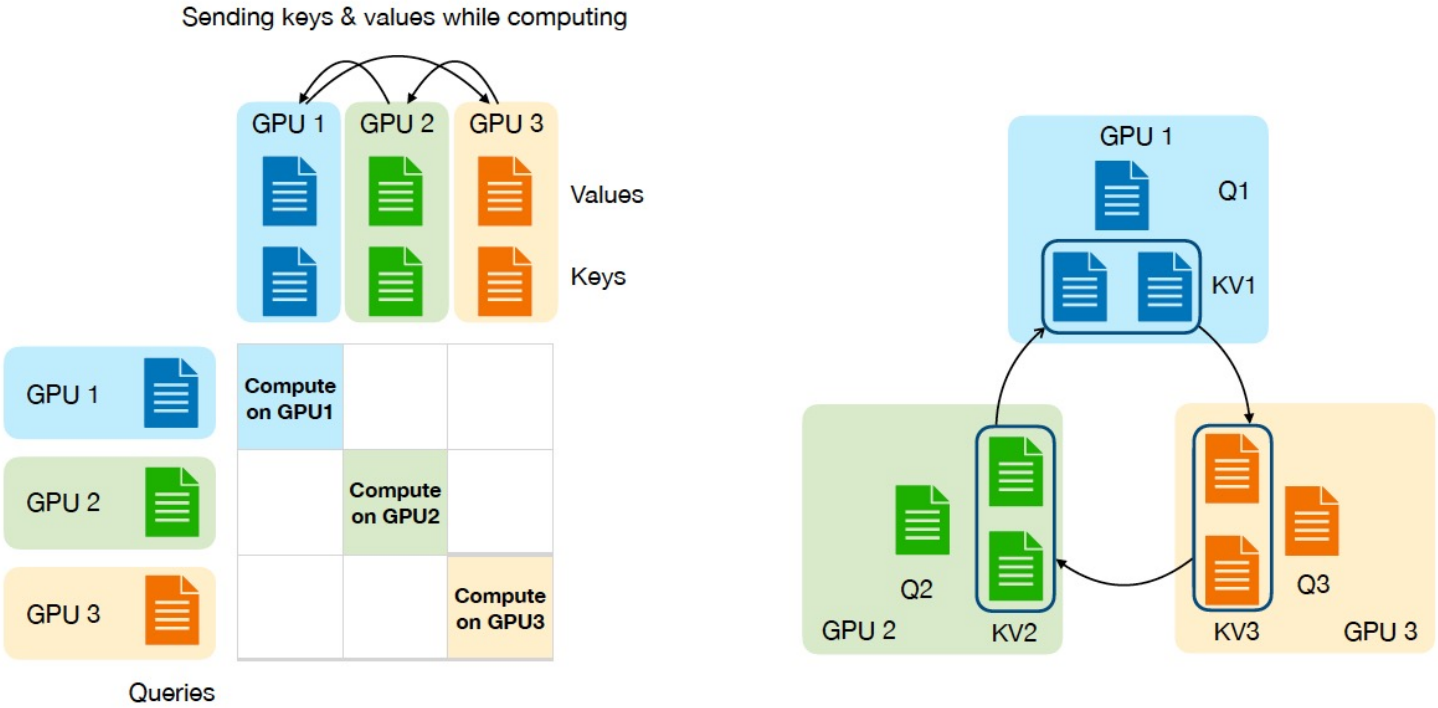
Sequence (Context) Parallelism



DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models

Sequence (Context) Parallelism

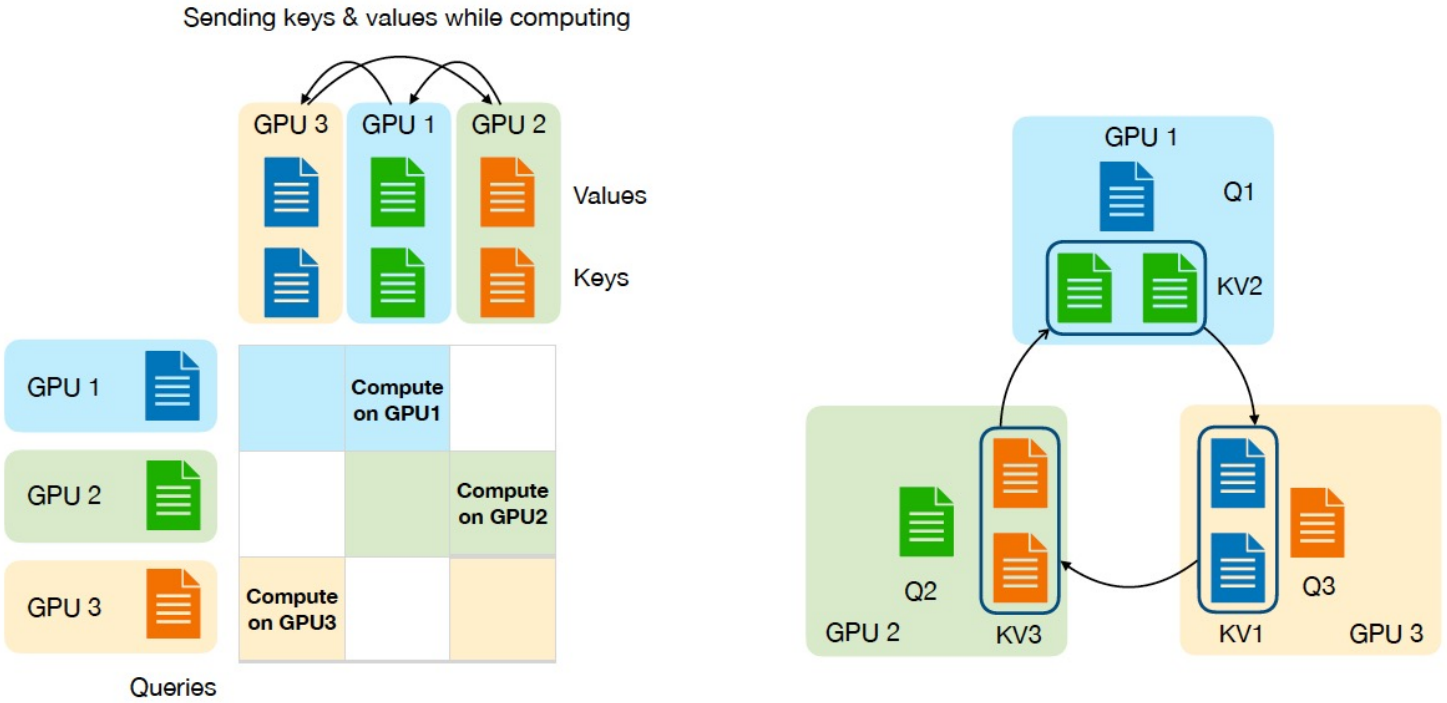
Solution 2: Ring Attention



Ring Attention with Blockwise Transformers for Near-Infinite Context

Sequence (Context) Parallelism

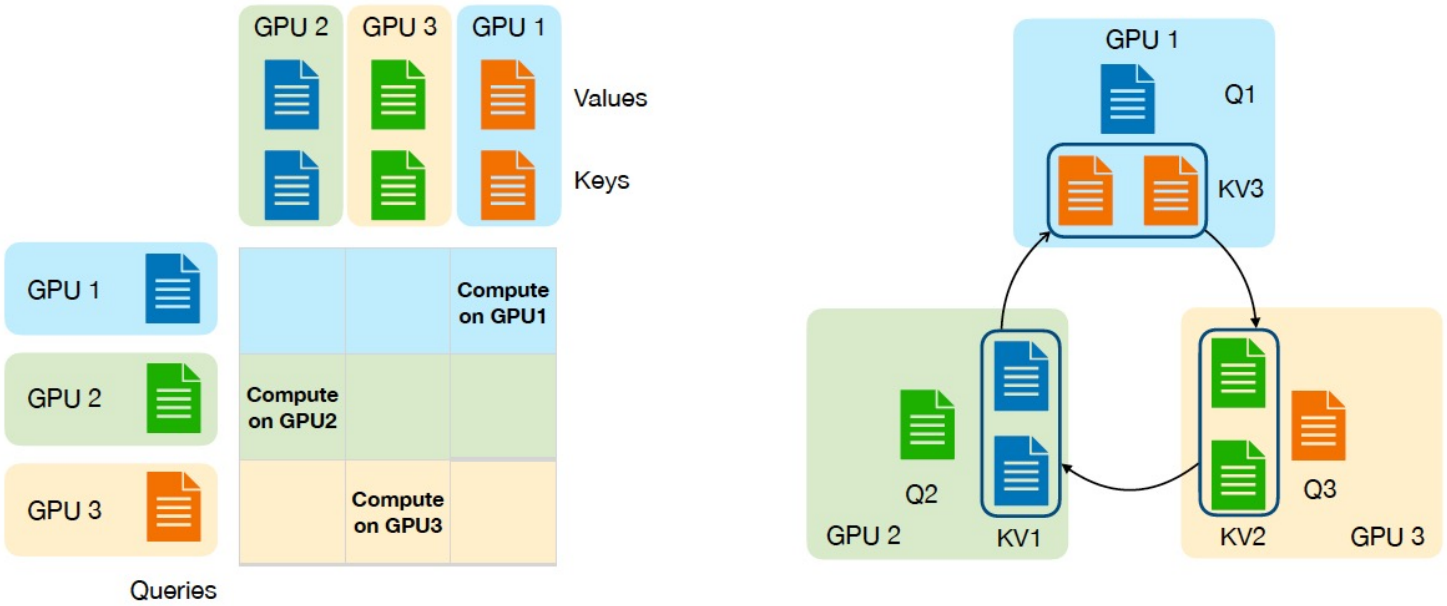
Solution 2: Ring Attention



Ring Attention with Blockwise Transformers for Near-Infinite Context

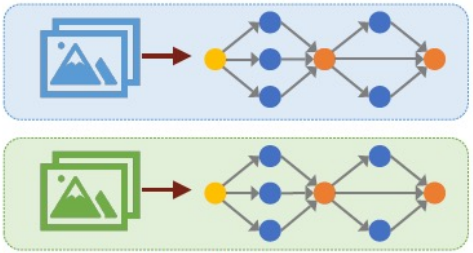
Sequence (Context) Parallelism

Solution 2: Ring Attention



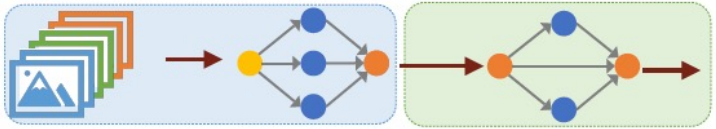
Ring Attention with Blockwise Transformers for Near-Infinite Context

Summary of Different Parallelization Methods



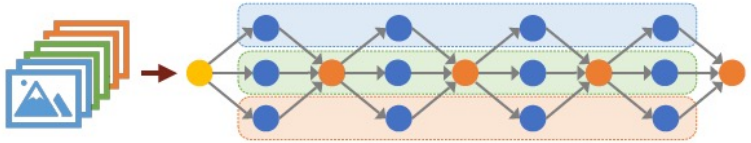
1. Data Parallelism:

- Split the data
- N copies of model
- high utilization, high memory cost, low communication
- Optimization: ZeRO 1/2/3, FSDP



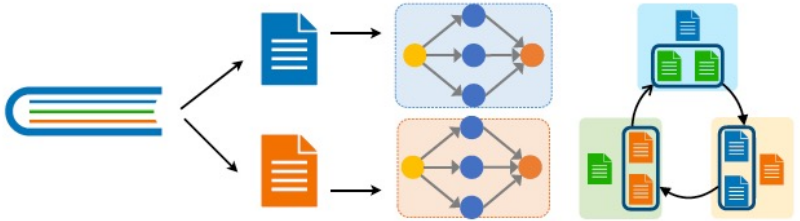
2. Pipeline Parallelism:

- Split the model by layer
- Single copy of model
- low utilization, low memory cost, medium communication



3. Tensor Parallelism:

- Split the model by tensor
- Single-copy of model
- high utilization, low memory cost, high communication



4. Sequence Parallelism:

- Split the data by tokens
- Extra communication in Attention layers

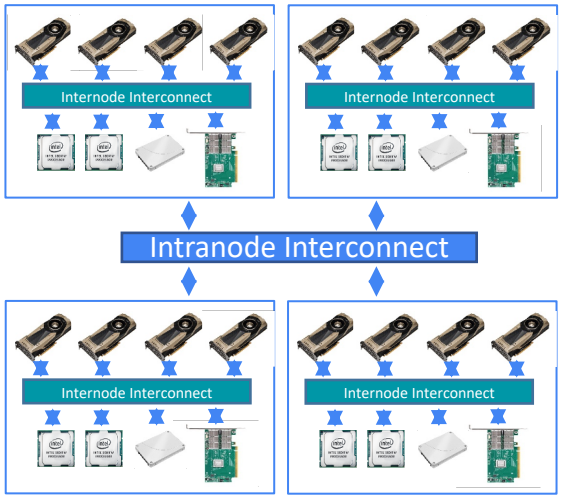
Figures credit from CMU 15-849 [Jia 2022]

References

- TSM: Temporal Shift Module for Efficient Video Understanding [Lin et al. 2019]
- Scaling Distributed Machine Learning with the Parameter Server. [Li et al. 2014]
- Optimization of collective communication operations in [Rajeev et al. 2005]
- GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism [Huang et al. 2018]
- Colossal-AI: A Unified Deep Learning System For Large-Scale Parallel Training [Li et al. 2021]
- Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism [Shoeybi, et al, 2019]
- ZeRO: Memory Optimizations Toward Training Trillion Parameter Models [Rajbhandari, et al, 2019]
- DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models [Jacobs, et al, 2023]
- Ring Attention with Blockwise Transformers for Near-Infinite Context [Liu, et al, 2023]

Billon-Scale Model Training - Scale Out Large Model Training

- Model parallelism (Megatron-LM)
  - Partition the model states vertically across multiple GPUs.
- Pipeline parallelism (PipeDream, SOSP'19)
  - Partition the model states horizontally across layers.
- ZeRO: Zero Redundancy Optimizer (ZeRO, SC'20)
  - Split the training batch across multiple GPUs without model states duplication.



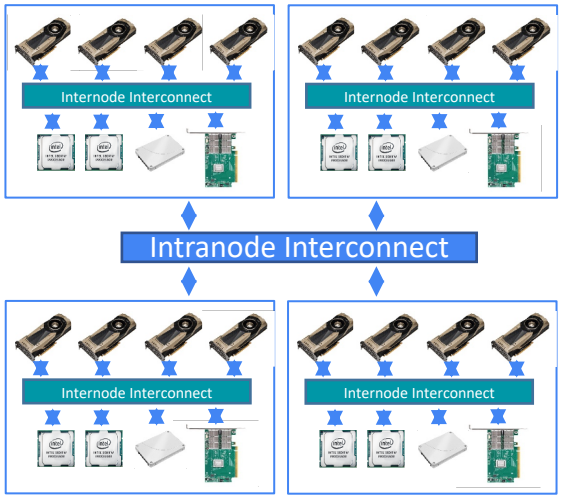
Distributed GPU Cluster

Billon-Scale Model Training - Scale Out Large Model Training

- Model parallelism (Megatron-LM)
  - Partition the model states vertically across multiple GPUs.

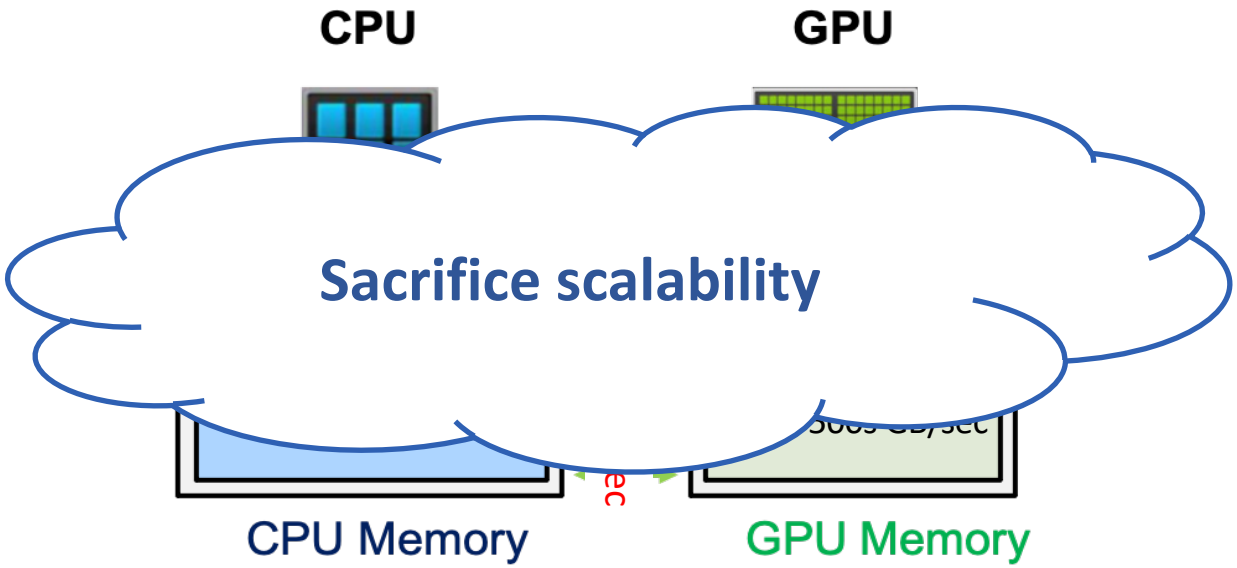
Require multiple GPUs resources

  - Split the training batch across multiple GPUs without model states duplication.



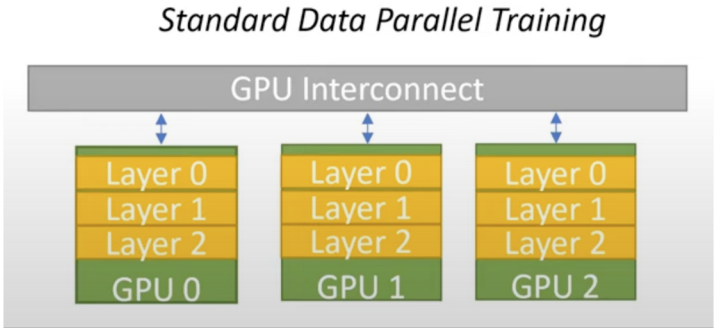
Distributed GPU Cluster

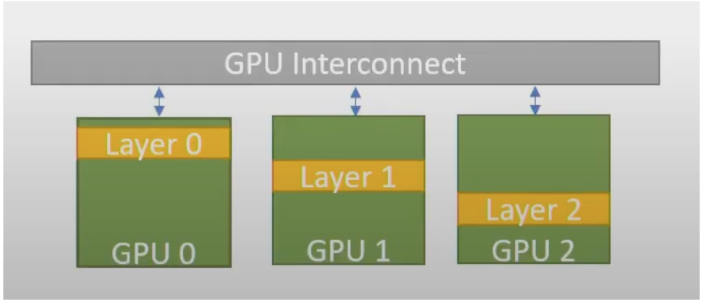
Billon-Scale Model Training - Scale Out Large Model Training



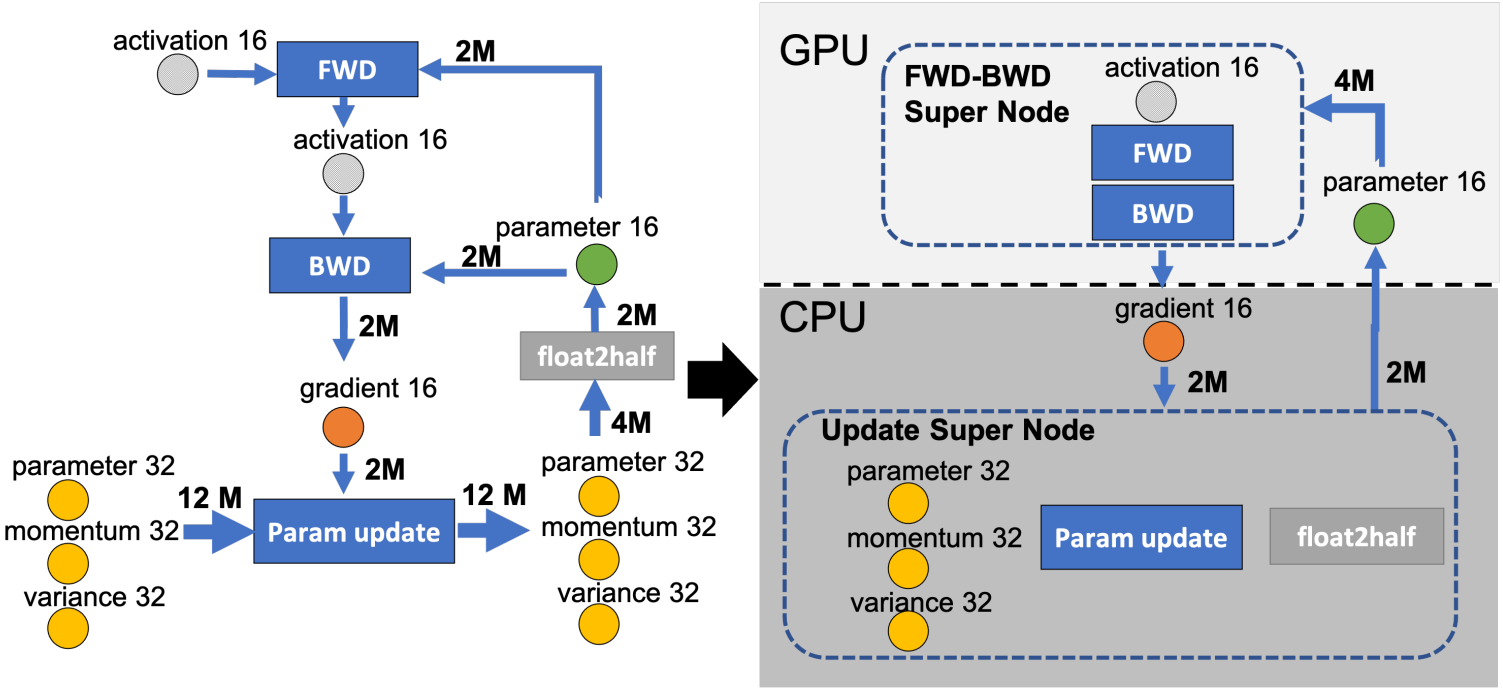
Recap: ZeRO so far:

- Zero-DP: optimizes model states
  - Existing DP replicates model states at each device.
    - Significant memory overhead
    - OOM on 32GB for 1.4B model
- Zero-R: reduces residual memory consumption.
  - Activation
    - Activation checkpointing
  - Temporary buffer resizes
  - Segmented memory
- Adam and mixed precision training are typical



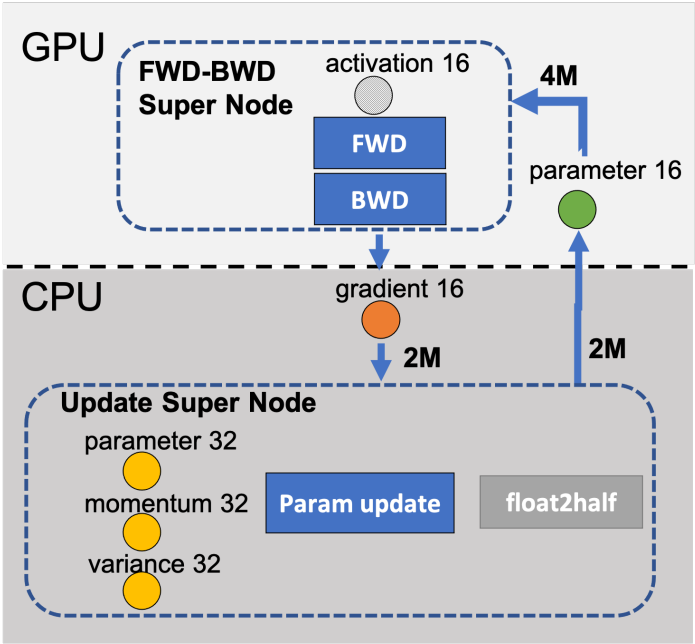


ZeRO-Offload [ATC 21] enables large model training by offloading data and compute to CPU



Offloading fp16 gradients and Update Super Node to CPU

Unique Optimal Offload Strategy



FWD-BWD	param16	gradient16	Update	Memory	Reduction
GPU	GPU	GPU	GPU	16M	1x(baseline)
GPU	GPU	CPU	GPU	14M	1.14x
GPU	GPU	GPU	CPU	4M	4x
GPU	GPU	CPU	CPU	2M	8x

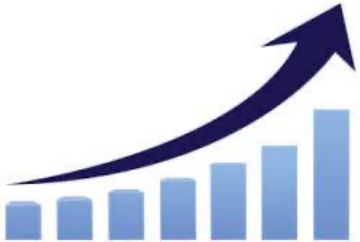
Memory savings for offload strategies that minimize communication volume compared to the baseline.

ZeRO-Offload: Democratizing Billion-Scale Model Training



Efficiency

- Enable 13B parameters model training on a single NVIDIA V100 GPU at 40 TFLOPS.



Scalability

- Achieve near perfect linear speedup with multiple GPUs.

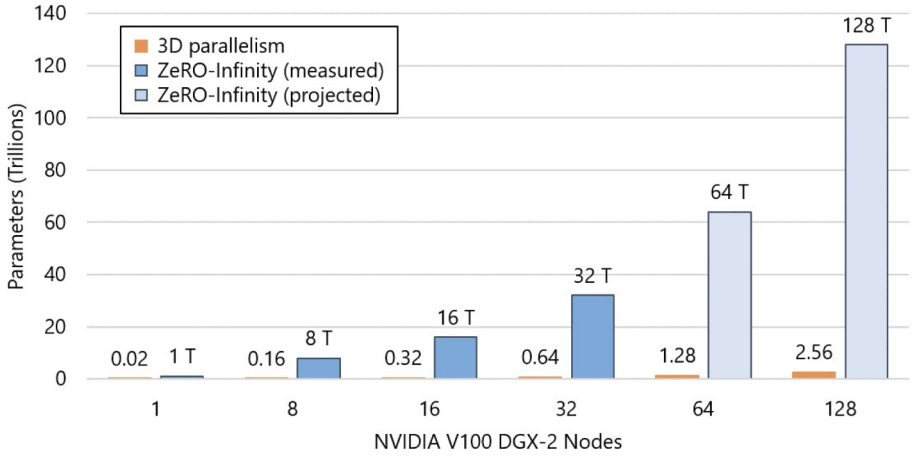


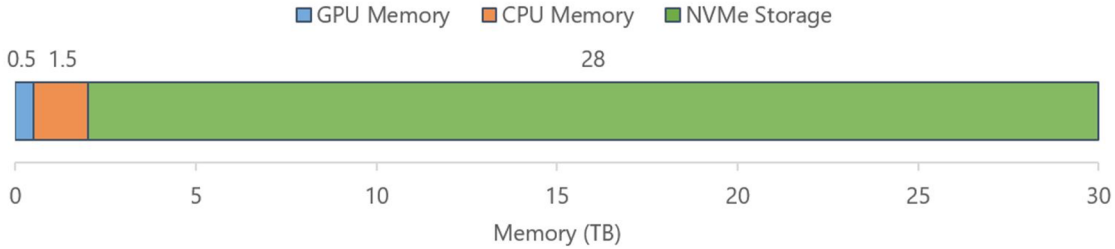
Usability

- Require no model refactoring.

Next: ZeRO-Infinity

- Breaking the GPU Memory Wall for Extreme Scale Deep Learning
- SC 21
- How to extend existing parallel training techniques for CPU/NVMe?





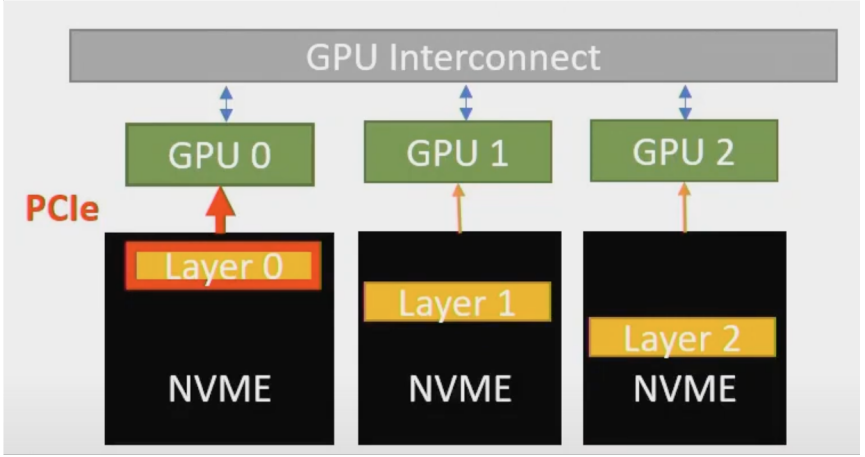
ZeRO-Infinity

- ZeRO with CPU and NVMe offload.

Params/grads stay in GPUs; Inter-GPU BW better
Optimizer states: updated across all nodes in ZeRO. Required data movement can occur in parallel across all nodes. Achieve an aggregate NVMe bandwidth of 1.7TB/s across 64 nodes.
Activation: each *GPU* can read and write to *CPU* memory in parallel for about 4GB/s. Checkpoint.

ZeRO with non-GPU memory

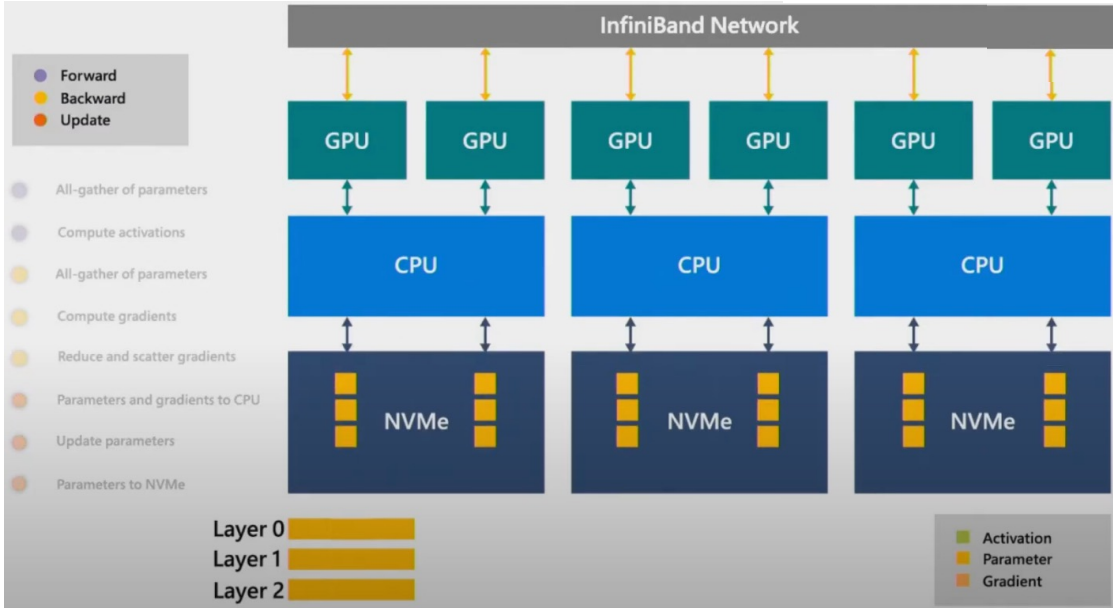
GPUs	Data Type	Required	NVMe memory	CPU Memory
1024	Params/Grads	60 GB/s	12 GB/s	12 GB/s
1024	Optimizer States	1500 GB/s	1792 GB/s	4096 GB/s
1024	Activations	4 GB/s	1.75GB/s	4GB/s



Since params/grads belong to each GPU, bandwidth is limited by a single PCIe link between NVMe and GPU.

ZeRO-Infinity

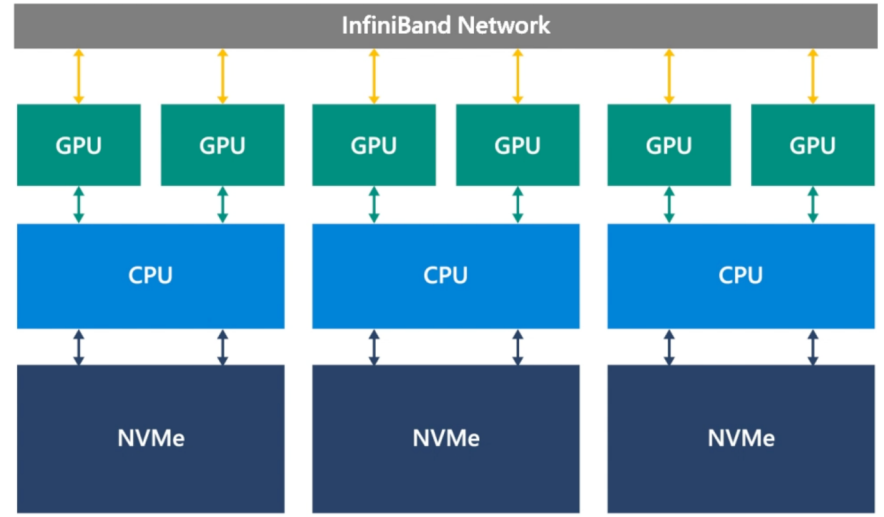
- Forward
  - Each GPU fetches a partition of model layer from NVMes, do an all gather in GPU infiniband => full layer.
  - Generate activations.
- Backward pass
  - Layer fetched in similar fashion
  - Generate gradients
  - Gradients reduced across the ranks, scattered to be written out to NVMes.
- Optimizer:
  - Gradient and parameters fetched into CPU memory from NVMes
  - Update the model parameters, return back to NVMes.



ZeRO-Infinity in Action:
https://youtu.be/YEdtLCjSZFY?si=P-8zTpGaMXdySLsh&t=598

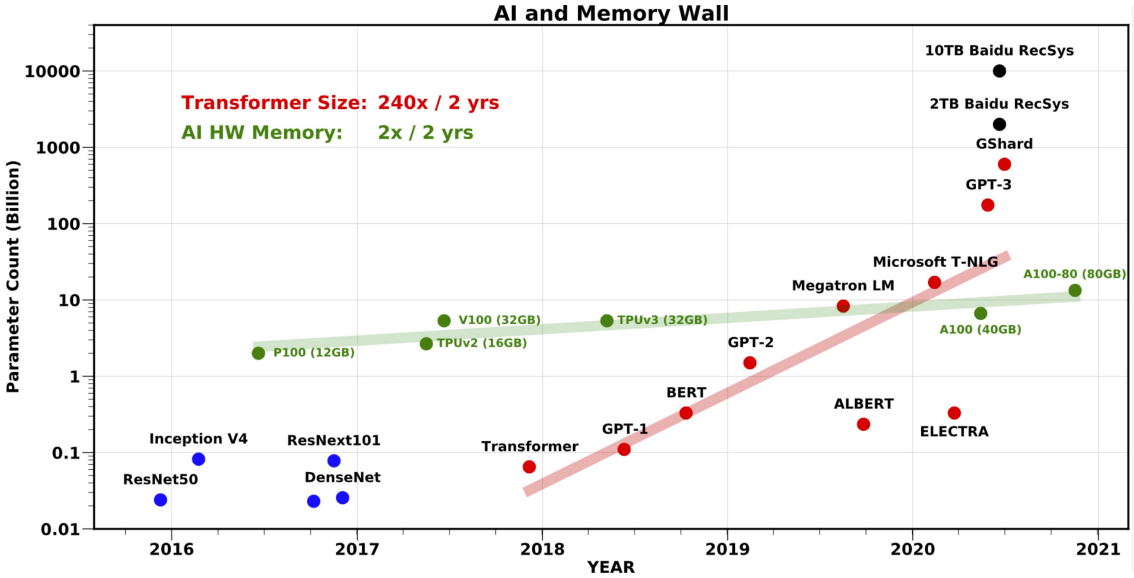
Summary of ZeRO-Infinity

- ZeRO with CPU and NVMe offload.
- Overlap centric design:
  - GPU computation
  - GPU <-> CPU, CPU <-> NVMe communications
  - GPU <-> GPU communications.



Large model training landscape BEFORE ZeRO-Infinity

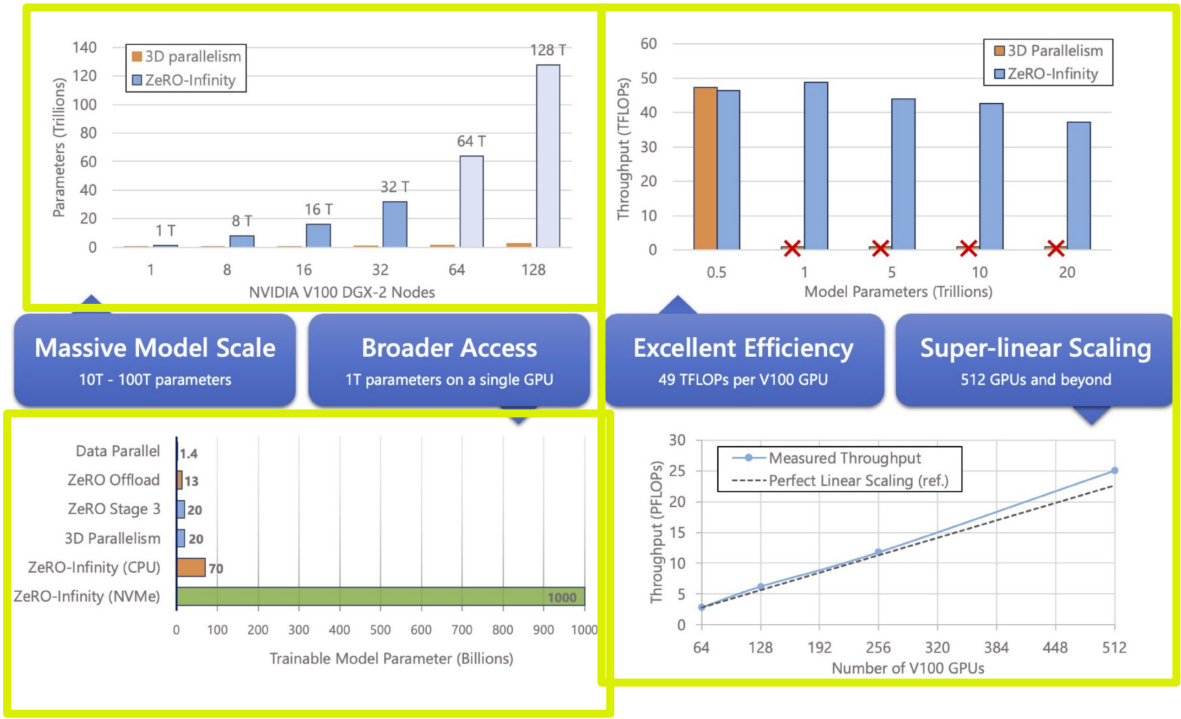
- **GPU Memory Wall**
  - 1T (10T) params: 800 (8K) V100 GPUs
  - How do we support the growth in model size?
- **Accessibility to large model training**
  - 256 GPUs to fine-tune GPT-3
  - Limited access to such resources
- **Model code refactoring**
  - Re-writing the model using 3D parallelism (tensor-slicing + pipeline parallelism)
  - Painful and error prone



*AI and Memory Wall. (This blogpost has been written in... | by Amir Gholami | riselab | Medium

Redefining the landscape with ZeRO-Infinity

- Beyond GPU Memory
  - 50x larger models
  - 32T params on 512 GPUs (instead of 25K)
- Broader access to large model training
  - GPT-3 sized fine-tuning on a single node/GPU (instead of 16 nodes)
- Excellent Throughput and Scalability
  - Comparable to 3D-parallelism
- Ease of Use
  - No model refactoring necessary



DeepSpeed

- ZeRO is implemented as part of DeepSpeed, compatible with Pytorch

```
import torch
...
...
model = BuildModel(config)
optimizer = Optimizer(model)
...
...
...
for batch in batches:
    loss = model(batch)
    loss.backward()
    optimizer.step()
```

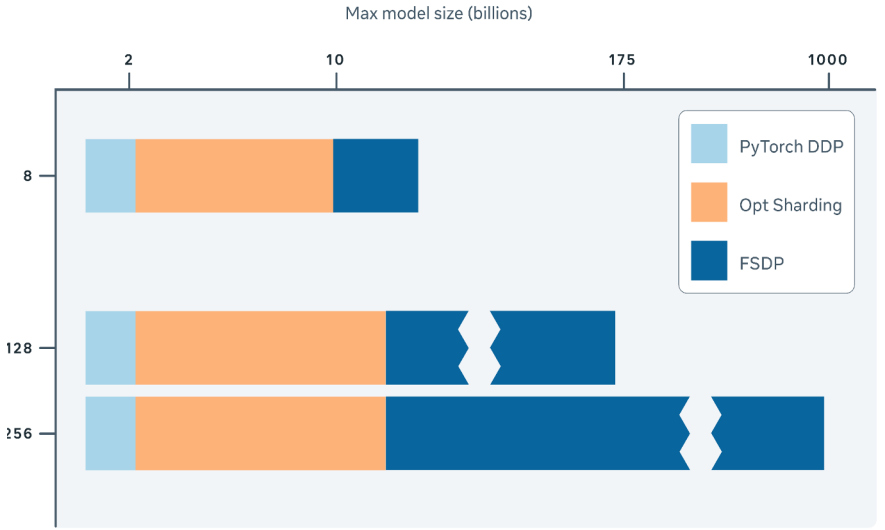
```
import torch
import deepspeed
...
model = BuildModel(config)
optimizer = Optimizer(model)
model = deepspeed.initialize(model,
                             optimizer
...
...
for batch in batches:
    loss = model(batch)
    model.backward(loss)
    model.step()
```

Pytorch FSDP (Fully Sharded Data Parallel)

- Initially by Fairscale.
- Meta's version
  - Partially inspired by DeepSpeed
  - Very similar to ZeRO3
- Supported in PyTorch June 2023
- The PyTorch FSDP (Fully Sharded Data Parallel) API implements data parallelism with ZeRO optimizations

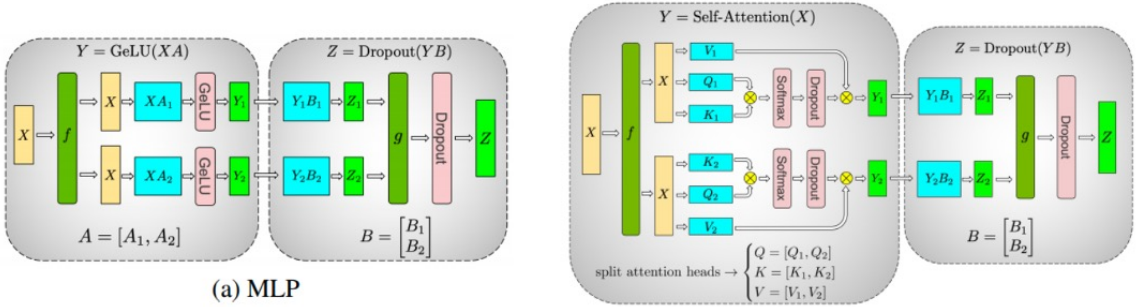
Example usage:

```
import torch
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP

torch.cuda.set_device(device_id)

module = ... # Module definition

sharded_module = FSDP(module)
optim = torch.optim.Adam(sharded_module.parameters(), lr=0.0001)

x = ... # Input data
y = sharded_module(x)

loss = y.sum()
loss.backward()
```

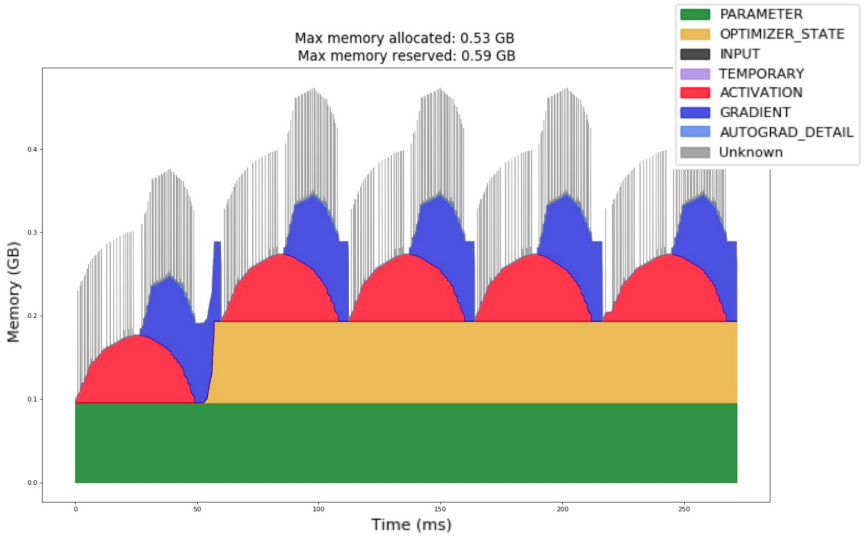


One more thing: GPU memory need is Dynamic

- Memory is NOT just the static bits, but also Activations (light-blue in the fig. below):

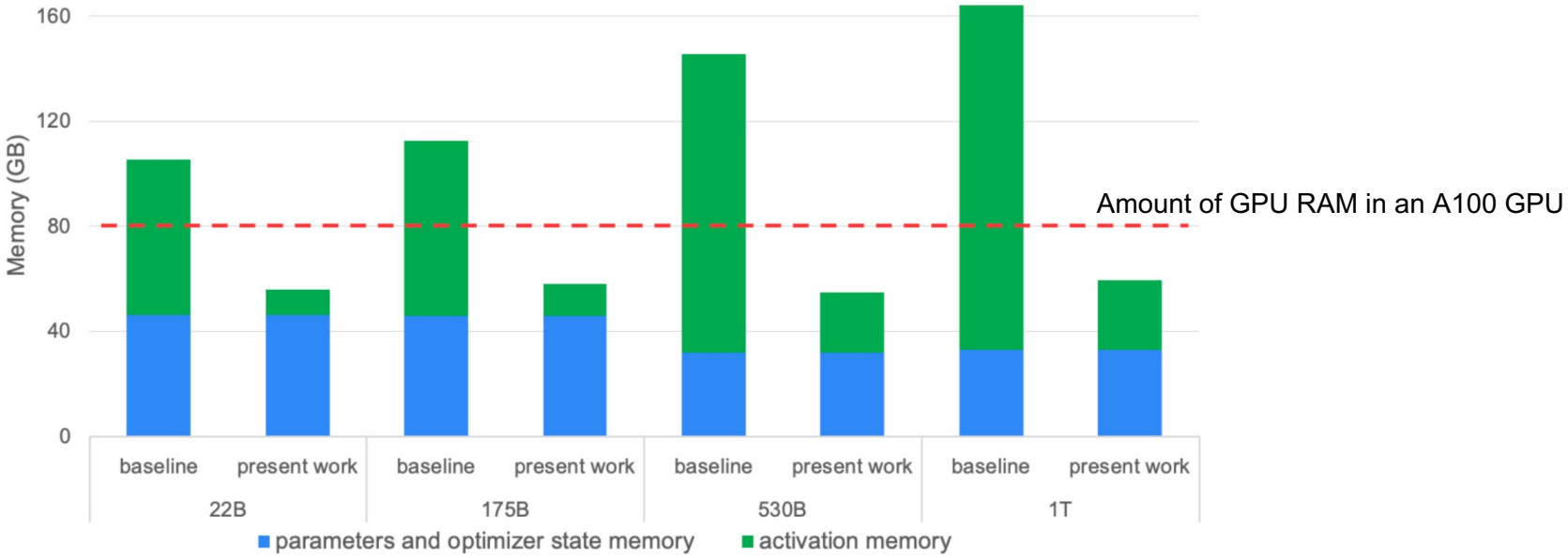


(a) MLP



One more thing: Activation Memory

- So far, we have mainly focused on the memory for Parameters
- Tensor // and Pipeline // can linearly reduce those BUT what about Activation Memory ?



[Korthikanti et al of Nvidia, 2022]

Activation Memory Requirement under Different schemes

[Korthikanti et al of Nvidia, 2022]

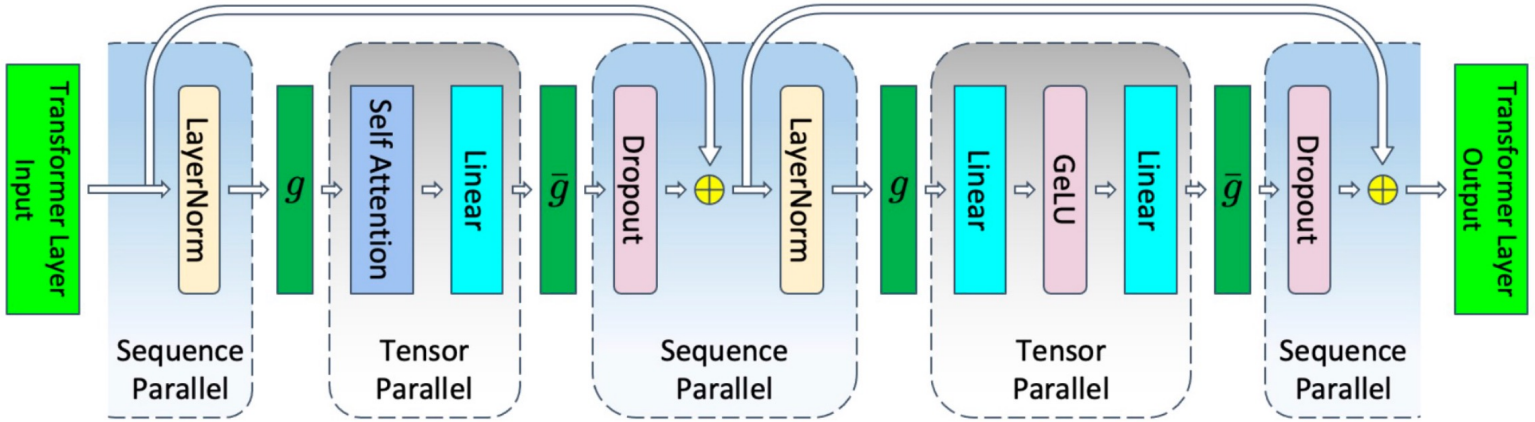
a	number of attention heads	p	pipeline parallel size
b	microbatch size	s	sequence length
h	hidden dimension size	t	tensor parallel size
L	number of transformer layers	v	vocabulary size

Table 1: Variable names.

Configuration	Activations Memory Per Transformer Layer
no parallelism	$sbh\left(34+5\frac{as}{h}\right)$
tensor parallel (baseline)	$sbh\left(10+\frac{24}{t}+5\frac{as}{ht}\right)$
tensor + sequence parallel	$sbh\left(\frac{34}{t}+5\frac{as}{ht}\right)$
tensor parallel + selective activation recomputation	$sbh\left(10+\frac{24}{t}\right)$
tensor parallel + sequence parallel + selective activation recomputation	$sbh(\frac{34}{t})$
full activation recomputation	$sbh(2)$

Table 2: Activations memory, in bytes, per transformer layer for different techniques.

Achieve True Linear Memory Scaling via Sequence Parallel



Observation: all the 10sbh terms are pointwise ops over the sequence

… so split up the layer norm/dropout layers along the sequence axis.

- In the forward pass, 'g' is an all gather, '$\bar{g}$' is reduce-scatter
- In the backward pass, the two are reversed.

Reducing Activation Memory Requirement via Sequence Parallelism and Selective Activation Recomputation

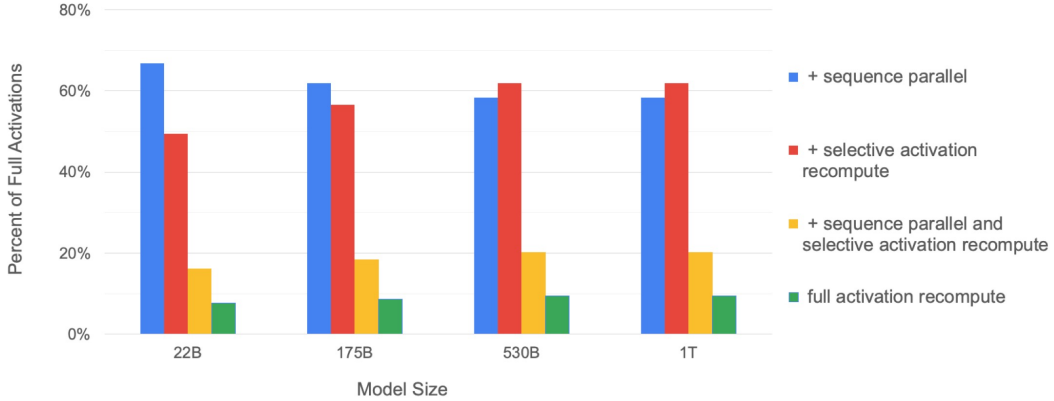


Figure 7: Percentage of required memory compared to the tensor-level parallel baseline. As the model size increases, both sequence parallelism and selective activation recomputation have similar memory savings and together they reduce the memory required by $\sim 5\times$.

Experiment	Forward (ms)	Backward (ms)	Combined (ms)	Overhead (%)
Baseline no recompute	7.7	11.9	19.6	–
Sequence Parallelism	7.2	11.8	19.0	−3%
Baseline with recompute	7.7	19.5	27.2	39%
Selective Recompute	7.7	13.2	20.9	7%
Selective + Sequence	7.2	13.1	20.3	4%

Table 4: Time to complete the forward and backward pass of a single transformer layer of the 22B model.

[Korthikanti et al, 2022]

Selective Activation Recomputation:
Judiciously choose to checkpoint and recompute only parts of each transformer layer that take up considerable amount of memory BUT not computationally expensive to recompute

Trade-offs of Distributed/ Parallelized Training Schemes

What are each of the parallelism primitives good for?

	Sync overhead	Memory	Bandwidth	Batch size	Easy to use?
DDP/ZeRO1	Per-batch	No scaling	2* # param	Linear	Very
FSDP (ZeRO3)	3x Per-FSDP block	Linear	3 * # param	Linear	Very
Pipeline	Per-pipeline	Linear	Activations	Linear	No
Tensor+seq	2x transformer block	Linear	8*activations per-layer all-reduce	No impact	No

Have to balance limited resource – memory, bandwidth, batch size

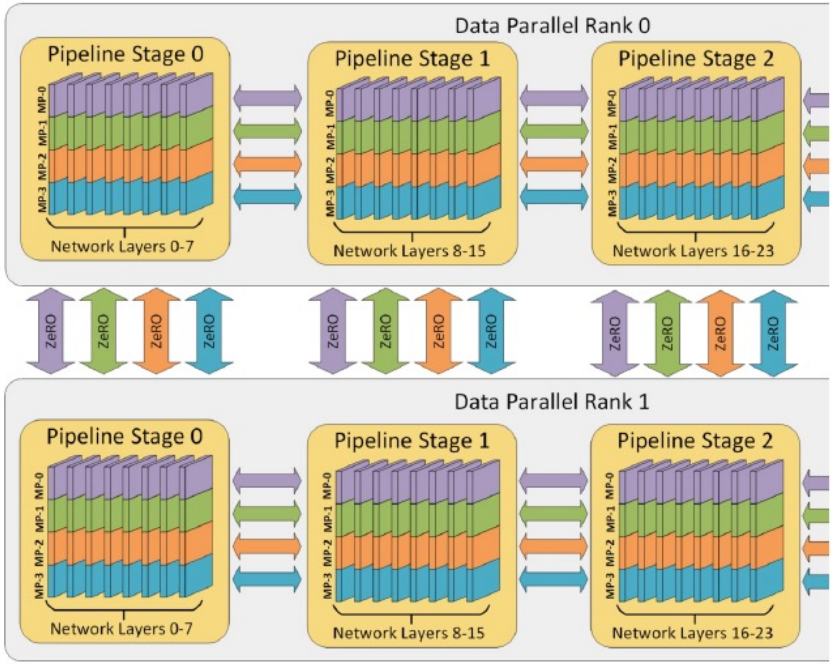
Putting it All Together: “3D Parallelism”

Simple rules of thumb from the literature.

1. Until your model fits in memory..
   - Tensor parallel up to GPUs / machine
   - Pipeline parallel across machines

2. Then until you run out of GPUs
   - Scale the rest of the way with data parallel

If your batch size is small.. gradient accumulate to trade higher batch sizes for better communication efficiency.



Megatron-LM: PTD-P Results

Scheme	Number of parameters (billion)	Model-parallel size	Batch size	Number of GPUs	Microbatch size	Achieved teraFlOP/s per GPU	Training time for 300B tokens (days)
ZeRO-3 without Model Parallelism	174.6	1	1536	384	4	144	90
				768	2	88	74
				1536	1	44	74
	529.6	1	2560*	640	4	138	169
			2240	1120	2	98	137
				2240	1	48	140
PTD Parallelism	174.6	96	1536	384	1	153	84
				768	1	149	43
				1536	1	141	23
	529.6	280	2240	560	1	171	156
				1120	1	167	80
				2240	1	159	42

Linear Scaling (w.r.t. #GPUs) is achievable via 3D-Parallelism

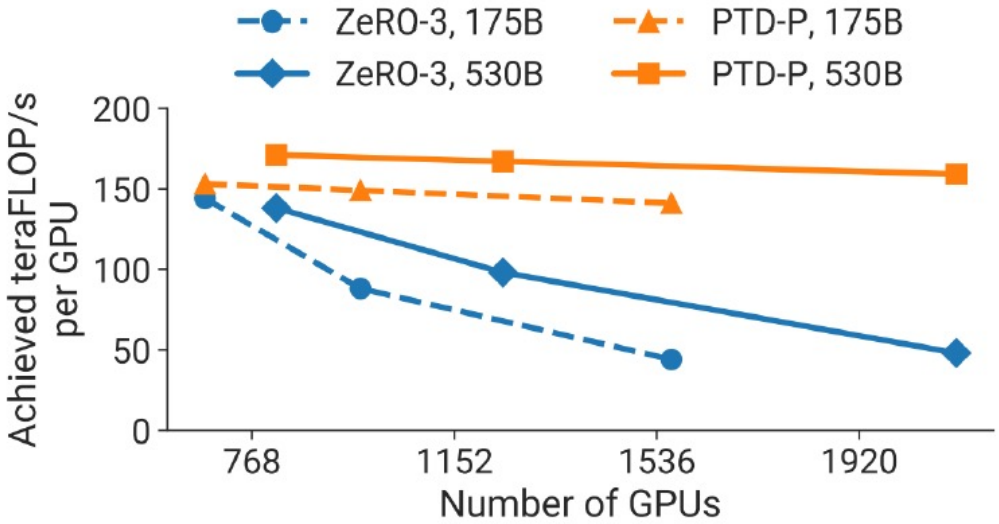


Figure 10: Throughput per GPU of PTD-P and ZeRO-3 for two different GPT models (the 175B GPT-3 model is shown with dotted lines, and the 530B model is shown with solid lines). Global batch sizes are fixed and ZeRO-3 is used without any model parallelism.

More GPUS, same, flat utilization!

Scaling Strategies as reported in [D. Narayanan et al 2021]

Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM

Deepak Narayanan[†*], Mohammad Shoeybi[†], Jared Casper[†], Patrick LeGresley[†], Mostofa Patwary[†], Vijay Korthikanti[†], Dmitri Vainbrand[†], Prethvi Kashinkunti[†], Julie Bernauer[†], Bryan Catanzaro[†], Amar Phanishayee[*], Matei Zaharia[‡]
[†]NVIDIA [‡]Stanford University [*]Microsoft Research

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	DP size
1.7	24	2304	24	1	1	32	512	137	44%	4.4	32
3.6	32	3072	30	2	1	64	512	138	44%	8.8	32
7.5	32	4096	36	4	1	128	512	142	46%	18.2	32
18.4	48	6144	40	8	1	256	1024	135	43%	34.6	32
39.1	64	8192	48	8	2	512	1536	138	44%	70.8	32
76.1	80	10240	60	8	4	1024	1792	140	45%	143.8	32
145.6	96	12288	80	8	8	1536	2304	148	47%	227.1	24
310.1	128	16384	96	8	16	1920	2160	155	50%	297.4	15
529.6	128	20480	105	8	35	2520	2520	163	52%	410.2	9
1008.0	160	25600	128	8	64	3072	3072	163	52%	502.0	6

Notes

- Tensor parallel first up to 8, then caps out at 8.
- Pipeline parallel goes up to make the model fit.
- Dara parallel gradually decreases with scale, with the largest model having DP=6

For Tensor Parallelism, P = 8 is often Optimal Why ?

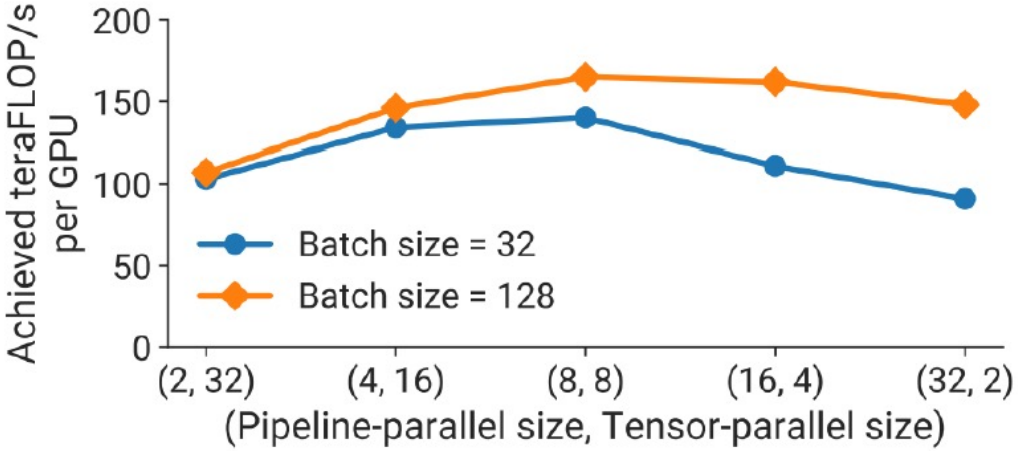


Figure 13: Throughput per GPU of various parallel configurations that combine pipeline and tensor model parallelism using a GPT model with 162.2 billion parameters and 64 A100 GPUs.

When parallelizing across 64 machines – it's best to use a 8 x 8 configuration.

What did recent LLM trainers do ?

DeepSeek

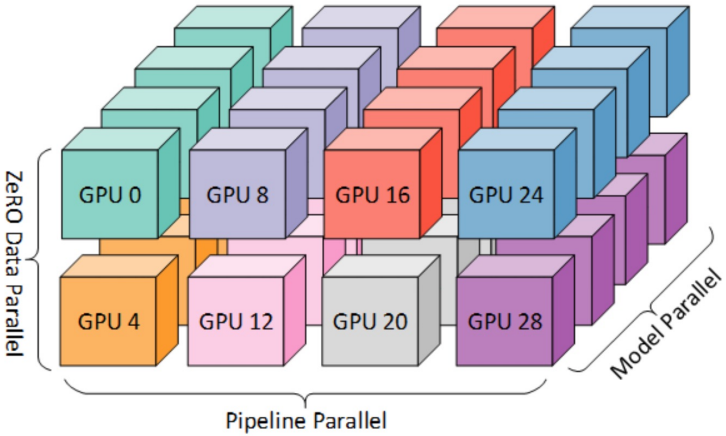
2.4. Infrastructures

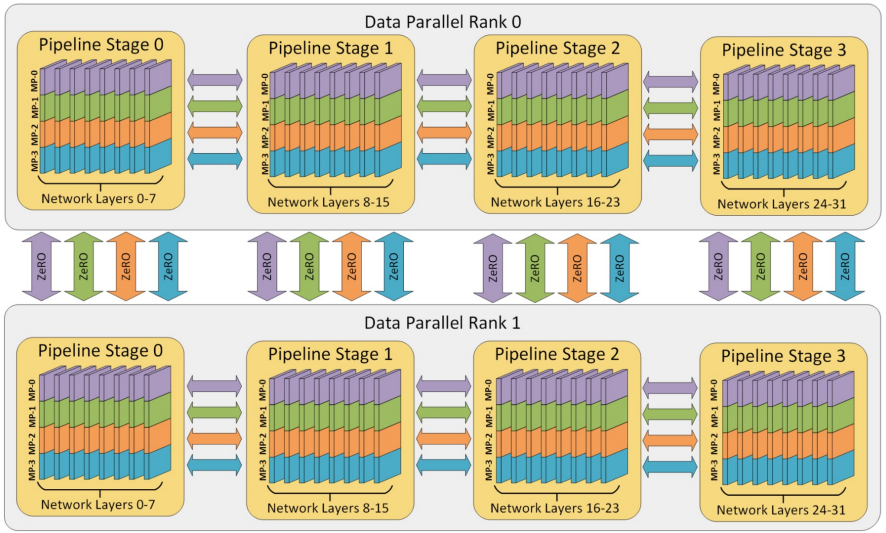
We use an efficient and light-weight training framework named HAI-LLM (High-flyer, 2023) to train and evaluate large language models. Data parallelism, tensor parallelism, sequence parallelism, and 1F1B pipeline parallelism are integrated into this framework as done in Megatron (Korthikanti et al., 2023; Narayanan et al., 2021; Shoeybi et al., 2019). We also leverage the flash attention (Dao, 2023; Dao et al., 2022) technique to improve hardware utilization. ZeRO-1 (Rajbhandari et al., 2020) is exploited to partition optimizer states over data parallel ranks. Efforts are also made to overlap computation and communication to minimize additional waiting overhead, including the backward procedure of the last micro-batch and reduce-scatter operation in ZeRO-1, and GEMM computation and all-gather/reduce-scatter in sequence parallel. Some layers/operators are fused to speed up training, including LayerNorm, GEMM whenever possible, and Adam updates. To improve model training stability, we train the model in bf16 precision but accumulate gradients in fp32 precision. In-place cross-entropy is performed to reduce GPU memory consumption, i.e.: we convert bf16 logits to fp32 precision on the fly in the cross-entropy CUDA kernel (instead of converting it beforehand in HBM), calculate the corresponding bf16 gradient, and overwrite logits with its gradient.

DeepSeek – ZeRO stage 1 with Tensor, Sequence, and Pipeline parallel

deepspeed

- ZeRO + 3D (aka Pipeline-Tensor-Data PTD) Parallelism and Much More !
- Can scale out to 1000's of GPUs





https://www.deepspeed.ai/training/

DeepSpeed: Extreme-scale model training for everyone. DeepSpeed Team, Majumder, and Wang. https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/
Efficient Large-Scale Language Model Training on GPU Clusters using Megatron-LM. Narayanan, et al. https://dl.acm.org/doi/pdf/10.1145/3458817.3476209



deepspeed

- Open-source DL library accelerating large scale training and inference on-top of PyTorch
  - https://github.com/microsoft/DeepSpeed

- **ZeRO**
- 3D Parallelism
- MoE training + inference
- Optimized CUDA/ROCm/CPU kernels
  - Transformer, Adam, LAMB, AdaGrad, MoE
- Optimized communication libraries
- 1-bit Adam/LAMB

- Mixed precision training
- Profiling and Auto-tuning
- Sparse Attention
- Mixture of Quantization (MoQ)
- Progressive Layer Drop (PLD)
- Curriculum Learning
- Elastic training
- Advanced learning rate schedules

An Incomplete list of Configurations of DeepSpeed

- DeepSpeed ZeRO Stage 1 - **Shard optimizer states**, remains at speed parity with DDP whilst providing memory improvement
- DeepSpeed ZeRO Stage 2 - **Shard optimizer states and gradients**, remains at speed parity with DDP whilst providing even more memory improvement
- DeepSpeed ZeRO Stage 2 Offload - **Offload optimizer states and gradients to CPU**. Increases distributed communication volume and GPU-CPU device transfer, but provides significant memory improvement
- DeepSpeed ZeRO Stage 3 - **Shard optimizer states, gradients, parameters and optionally activations**. Increases distributed communication volume, but provides even more memory improvement
- DeepSpeed ZeRO Stage 3 Offload - **Offload optimizer states, gradients, parameters and optionally activations to CPU**. Increases distributed communication volume and GPU-CPU device transfer, but even more significant memory improvement.
- DeepSpeed Activation Checkpointing - **Free activations after forward pass**. Increases computation, but provides memory improvement for all stages.

DeepSpeed Usability

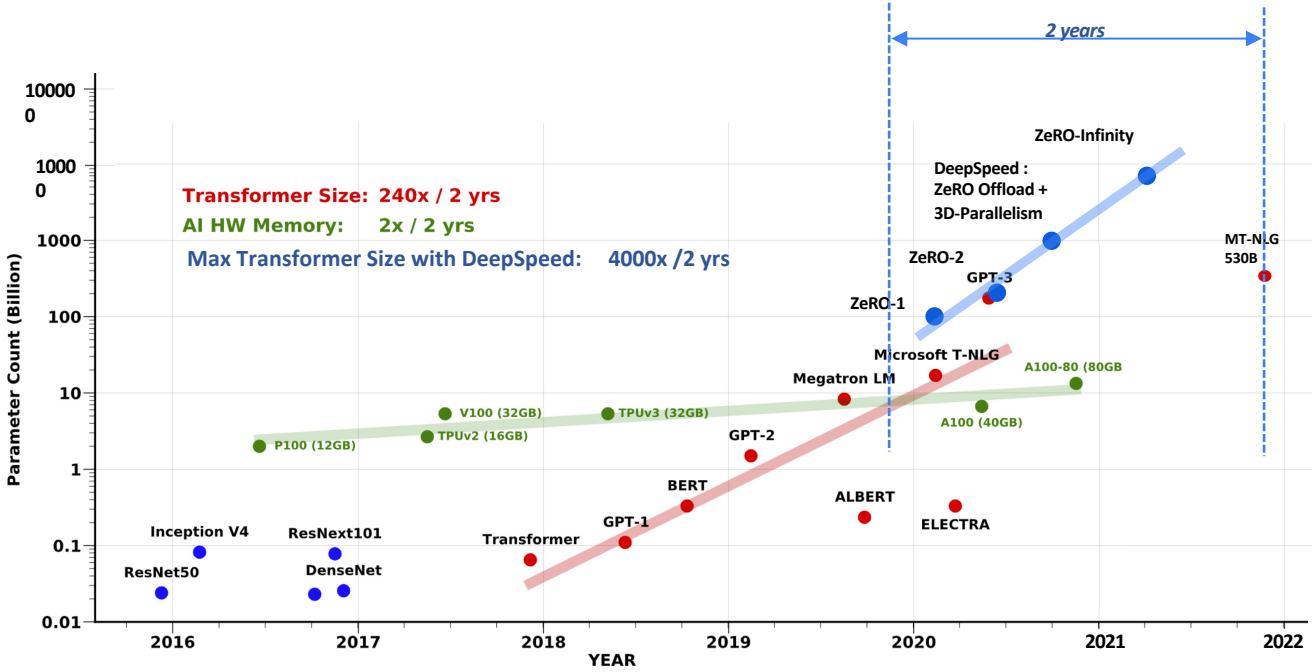
```python
# construct torch.nn.Module
model = MyModel()

# wrap w. DeepSpeed engine
engine, *_ = deepspeed.initialize(
    model=model,
    config=ds_config

# training-loop w.r.t. engine
for batch in data_loader:
  loss = engine(batch)
  engine.backward(loss)
  engine.step()
```

```python
ds_config = {
  "optimizer": {
    "type": "Adam",
    "params": {"lr": 0.001}
  },
  "zero": {
    "stage": 3,
    "offload_optimizer": {
        "device": "[cpu|nvme]"
    },
    "offload_param": {
        "device": "[cpu|nvme]"
    }
  }
}
```

Large model training landscape with DeepSpeed

- DeepSpeed Integration:
  - Huggingface
  - Pytorch Lightening
  - Catalyst

- DeepSpeed Powered Massive Models:
  - Microsoft-Turing NLG **17B**
  - GPT Neo-X **20B**
  - Jurrasic-1 **178B**
  - Big Science **200B** (ongoing)
  - Megatron-Turing NLG **530B**



*max model size efficiently trainable on 64 DGX-2 SuperPod, 1024 V100 (32GB) with DeepSpeed

*AI and Memory Wall. (This blogpost has been written in... | by Amir Gholami | riselab | Medium

Extra reading/resources

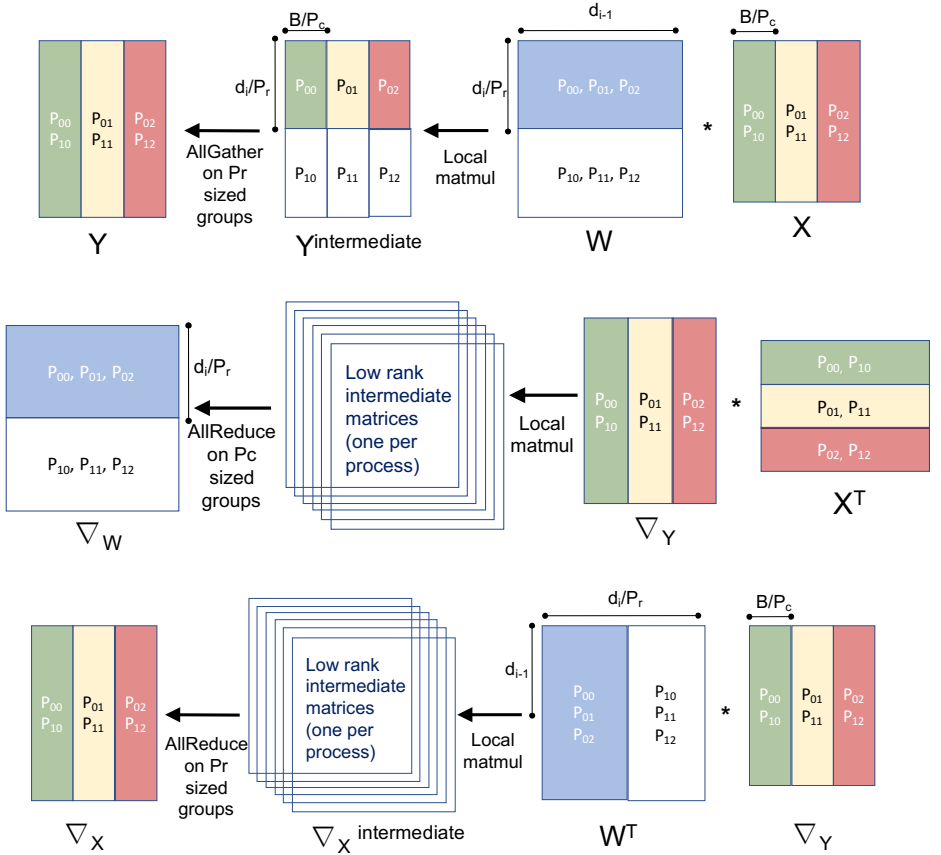
- Blogs
  - ZeRO & DeepSpeed: New system optimizations enable training models with over 100 billion parameters - Microsoft Research
  - Turing-NLG: A 17-billion-parameter language model by Microsoft - Microsoft Research
  - ZeRO-2 & DeepSpeed: Shattering barriers of deep learning speed & scale - Microsoft Research
  - DeepSpeed: Extreme-scale model training for everyone - Microsoft Research
  - ZeRO-Infinity and DeepSpeed: Unlocking unprecedented model scale for deep learning training - Microsoft Research
  - DeepSpeed: Accelerating large-scale model inference and training via system optimizations and compression - Microsoft Research
  - DeepSpeed: Advancing MoE inference and training to power next-generation AI scale - Microsoft Research
  - Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model - Microsoft Research
- Papers
  - ZeRO: Memory Optimizations Toward Training Trillion Parameter Models - Microsoft Research
  - ZeRO: Offload 2101.06840v1.pdf (arxiv.org)
  - ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning - Microsoft Research
  - Scalable and Efficient MoE Training for Multitask Multilingual Models - Microsoft Research
  - 1-bit LAMB: Communication Efficient Large-Scale Large-Batch Training with LAMB's Convergence Speed - Microsoft Research
  - 1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed - Microsoft Research
  - DeepSpeed-MoE: Advancing Mixture-of-Experts Inference and Training to Power Next-Generation AI Scale - Microsoft Research
  - [2201.11990v1] Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model (arxiv.org)
  - Large batch training with SimiGrad: 2021_NIPS_SimiGrad.pdf (unr.edu)
  - Accelerating Training of Transformer-Based Language Models with Progressive Layer Dropping - Microsoft Research

Practical Challenges When Deploying Parallelism

- Practical challenges when trying to deploy parallelism:
  - The space of possible strategies grows combinatorially large
  - Experimenting with different parallelization strategies is slow / expensive at the scale of hundreds or thousands of GPUs
- Can we *automate* parallelization for a given model and cluster?

Towards Automated Integration of Model and Data Parallelism

For a linear graph we can find the optimal hybrid method for analyzing the communication complexity, coupled with hardware utilization [1]



Processes are 2D indexed:
P= P_r x P_c

[1] Gholami, Amir, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. "Integrated model, batch, and domain parallelism in training neural networks." SPAA, 2018.

General Hybrid Methods

For a general computational graph we need to decide on:

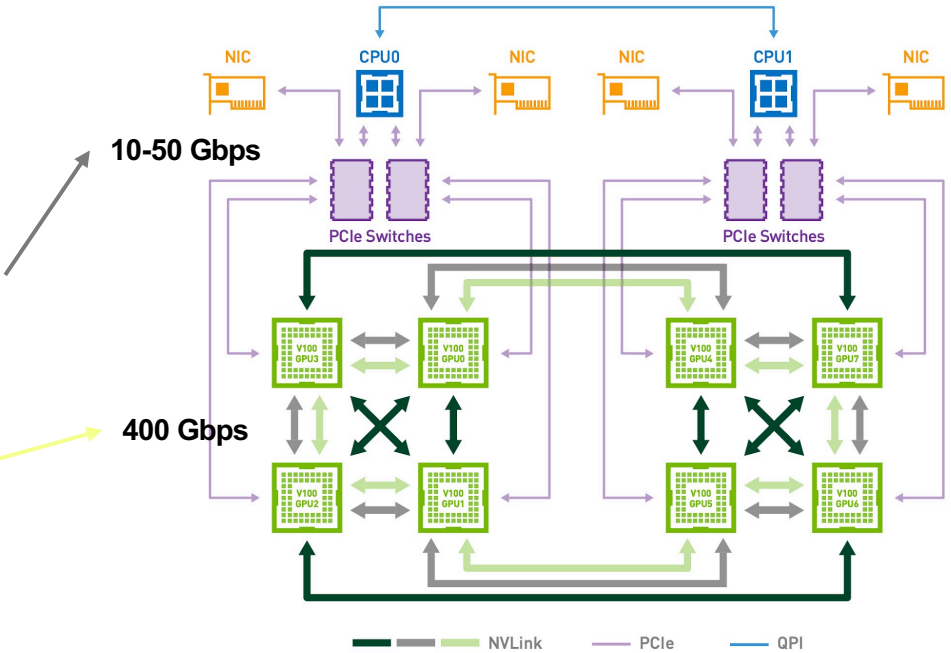
- How many processes to assign for DP
- Which axes to break the model: operator vs pipeline
- How to efficiently map the GPUs to the resulting execution graph
- …

For a general non-linear graph this leads to a combinatorically large search space

Alpa: An Automated Hybrid Method

Pipeline Parallelism*
(less communication)
*aka inter-op parallelism

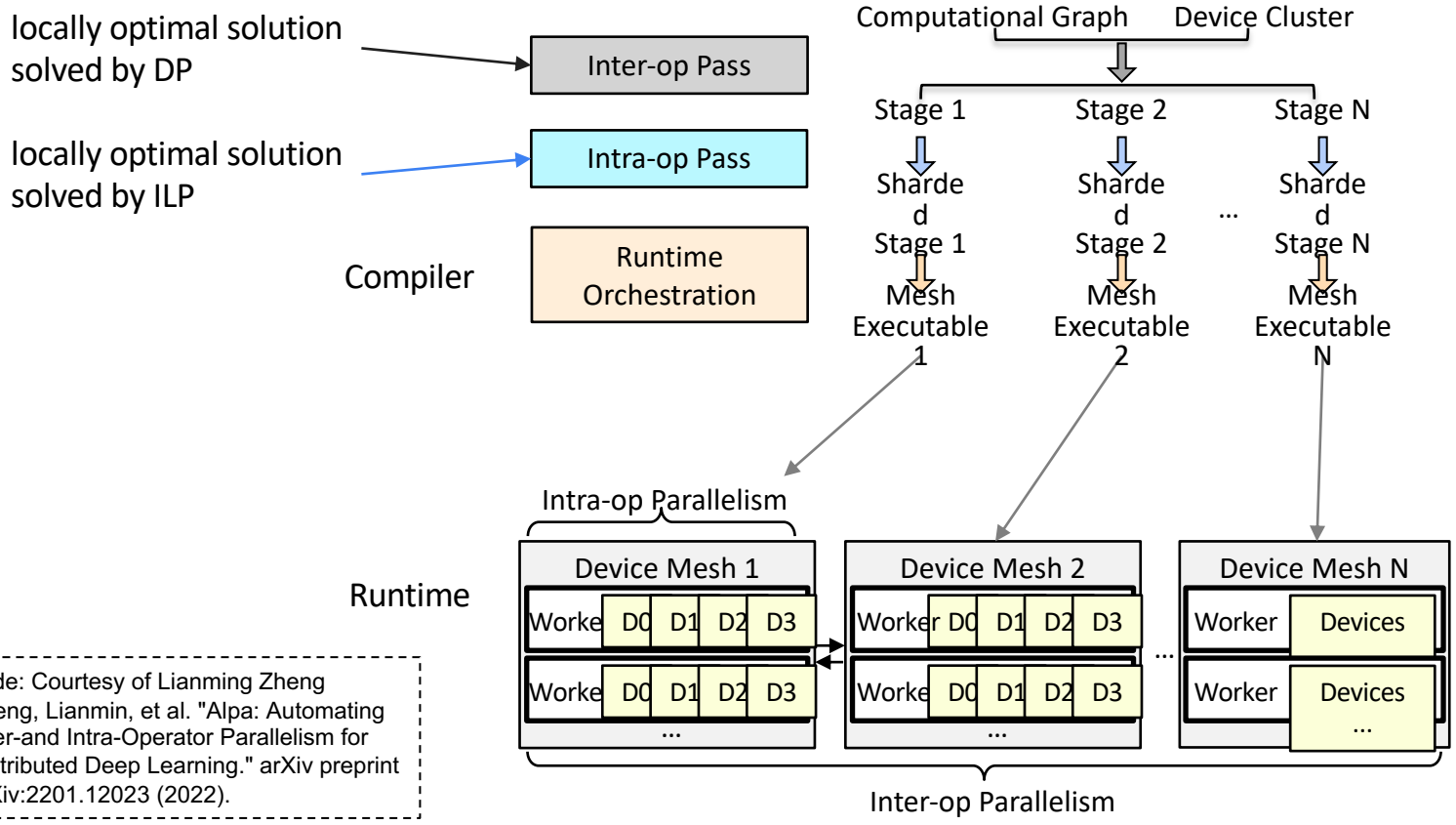
Model Parallelism*
(more communication)
* aka intra-op parallelism



- Organize inter- and intra-op parallelism as a two-level hierarchical space
- Design algorithms to derive optimal plans at each level

Slide: Courtesy of Lianming Zheng
Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." arXiv preprint arXiv:2201.12023 (2022).

Alpa: Architecture Overview

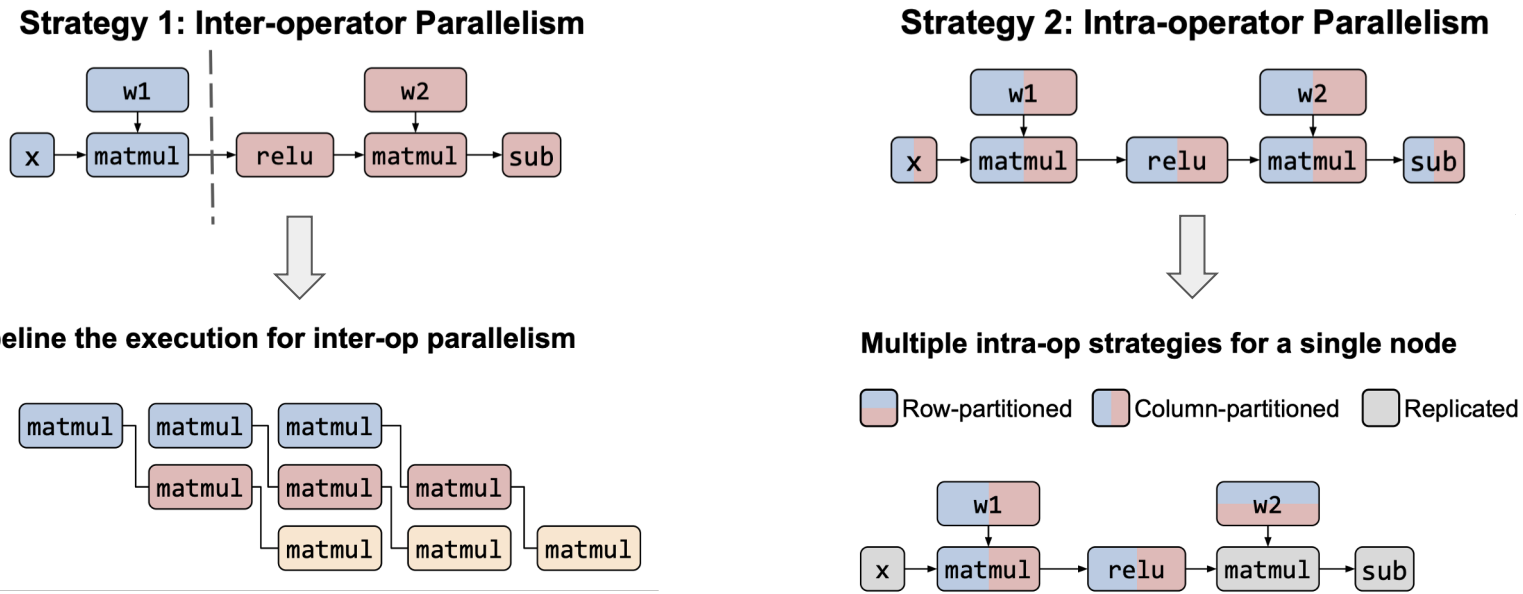


Slide: Courtesy of Lianming Zheng

Zheng, Lianmin, et al. "Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning." arXiv preprint arXiv:2201.12023 (2022).

Automatic Parallelization with Alpa

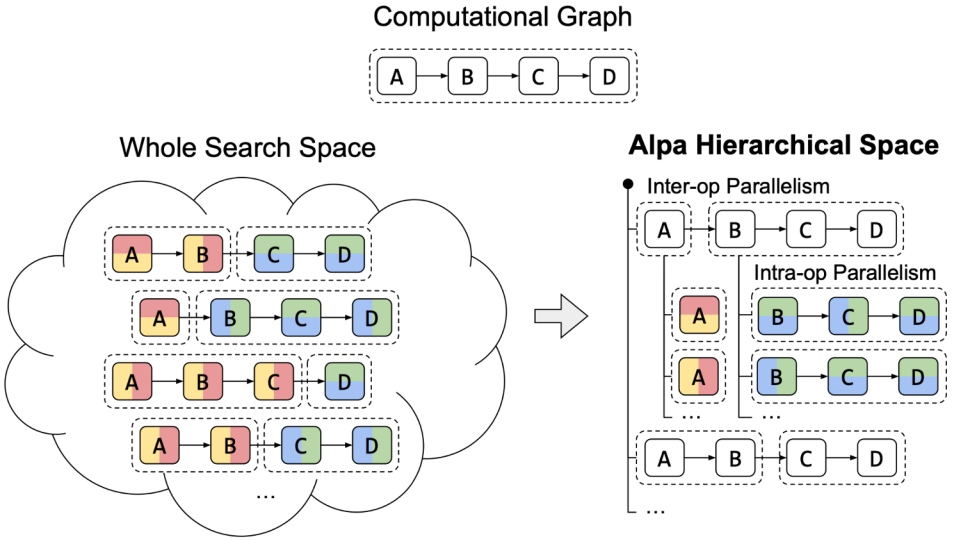
- Alpa is a system for automatically selecting and executing the optimal parallelism strategy for a given model and cluster
- Alpa separately considers *inter-operator* parallelism (i.e. pipeline parallelism) and *intra-operator* parallelism (i.e. data and tensor model parallelism)

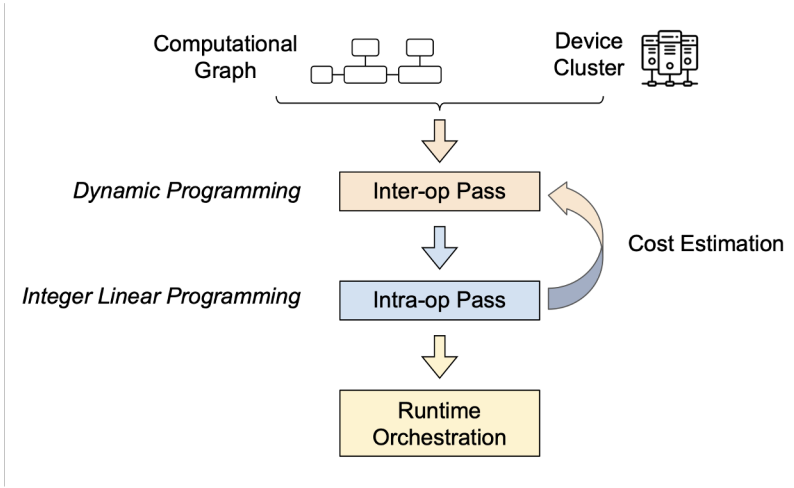


Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. Zheng et al. OSDI 2022 https://arxiv.org/pdf/2201.12023.pdf

Automatic Parallelization with Alpa

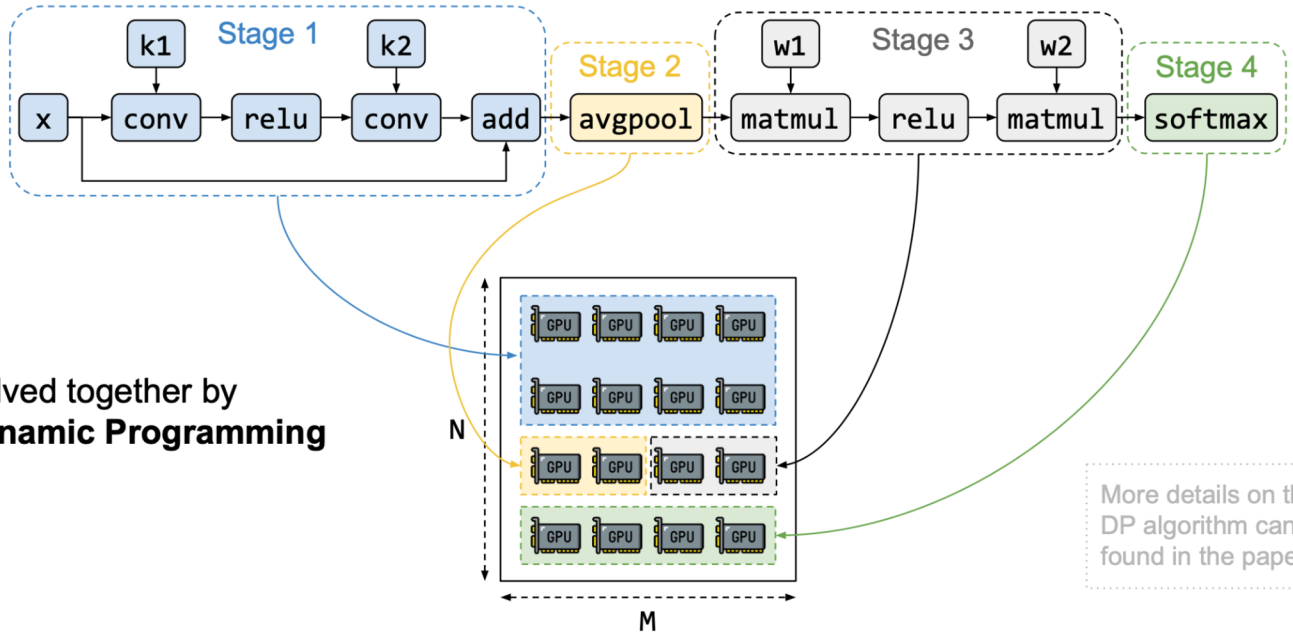
- Alpa will automatically search over the space of possible parallelism strategies for a given graph and choose the optimal combination of inter-op and intra-op parallelism





Automatic Parallelization with Alpa

Inter-op Pass

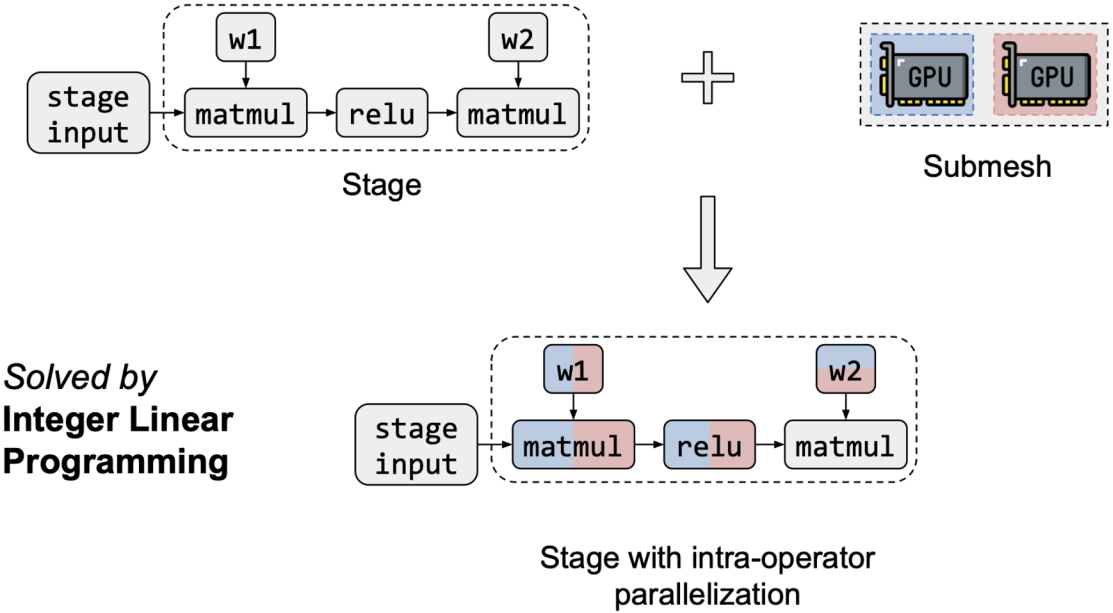


Solved together by
Dynamic Programming

More details on the DP algorithm can be found in the paper.

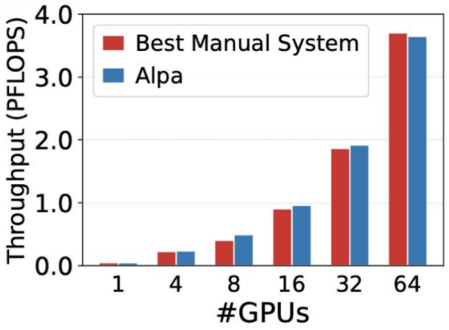
Automatic Parallelization with Alpa

Intra-op Pass



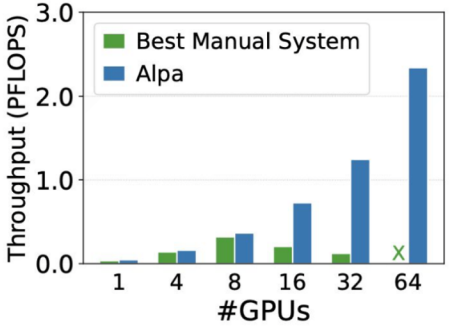
Automatic Parallelization with Alpa

GPT (up to 39B)



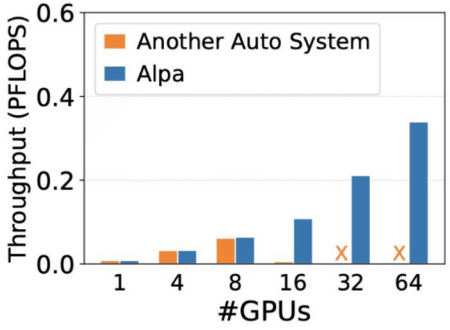
Match specialized manual systems.

GShard MoE (up to 70B)



Outperform the manual baseline by up to 8x.

Wide-ResNet (up to 13B)



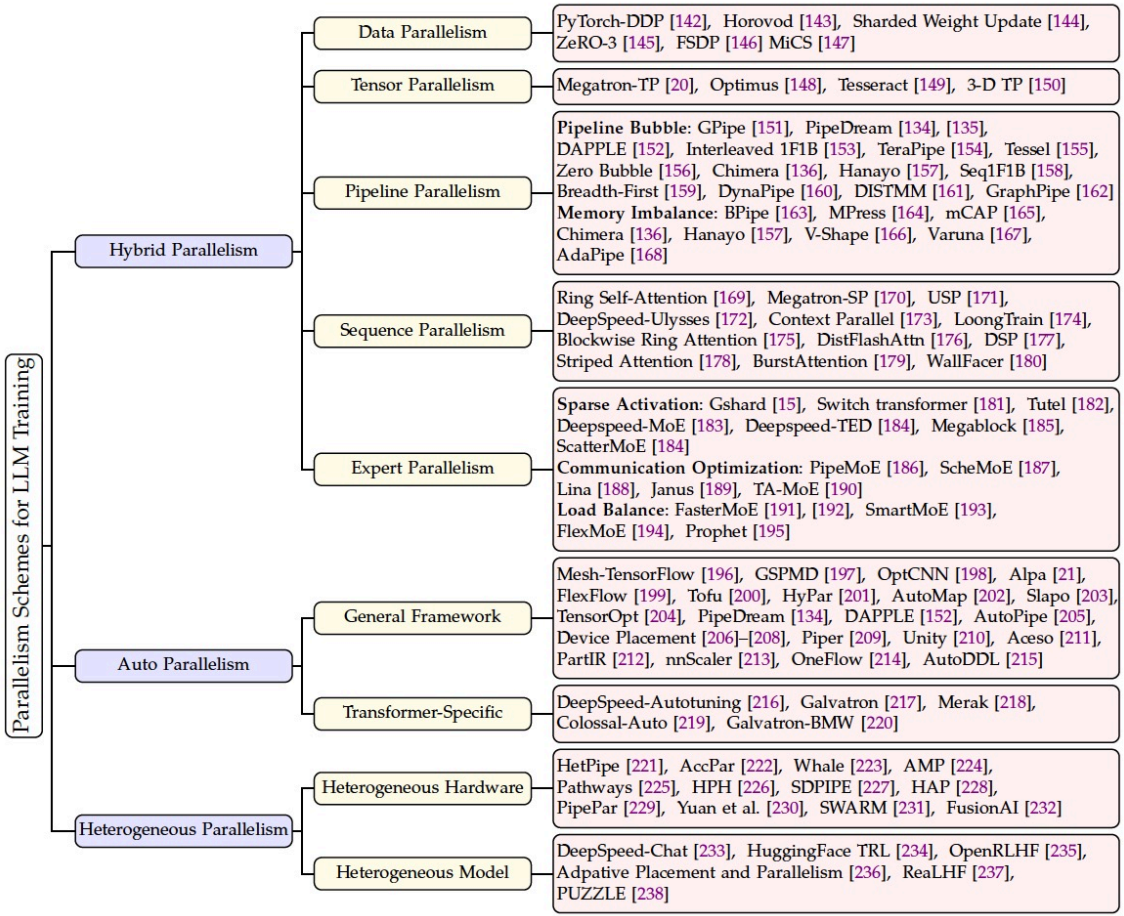
Generalize to models without manual plans.

Weak scaling results where the model size grow with #GPUs.
Evaluated on 8 AWS EC2 p3.16xlarge nodes with 8 16GB V100s each (64 GPUs in total).

Conclusion

- We need to parallelize data and models to train and serve Foundation Models at scale
- The most widely used forms of parallelism are Data parallelism, Tensor parallelism, and Pipeline parallelism ;
- Emerging support of Sequence (aka Context) parallelism
- Each of these parallelization strategies has trade-offs depending on the amount of memory usage and communication overhead
- Techniques such as Alpa (still researchie) can help automate the decision of which parallelization strategy or strategies to use
- Refer to: Duan et al, "Efficient Training of Large Language Models on Distributed Infrastructures: A Survey", https://arxiv.org/pdf/2407.20018, for a recent survey on Distributed Training for LLMs:

Studies on Parallelism schemes for Distributed LLM Training



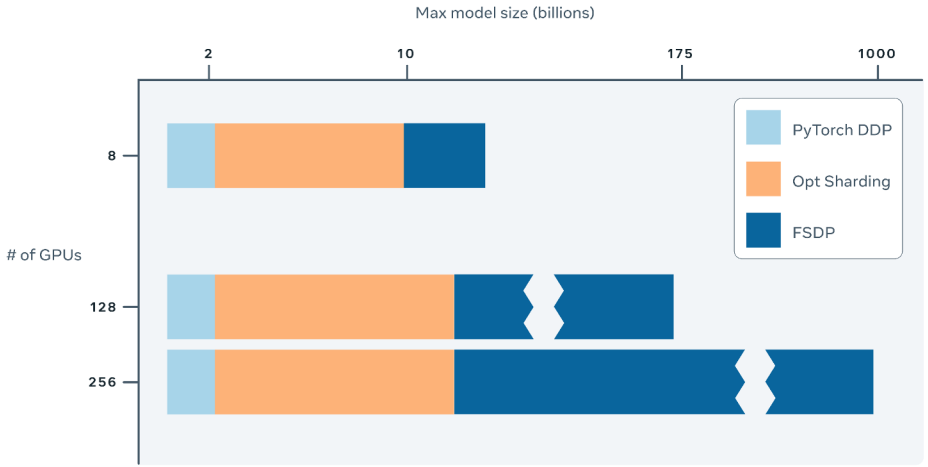
Source: Jiangfei Duan et al, “Efficient Training of LLMs on Distributed Infrastructures: A Survey”, July 2024

Backup Slides on

Fully Shard Data Parallel (FSDP) framework and its terminologies

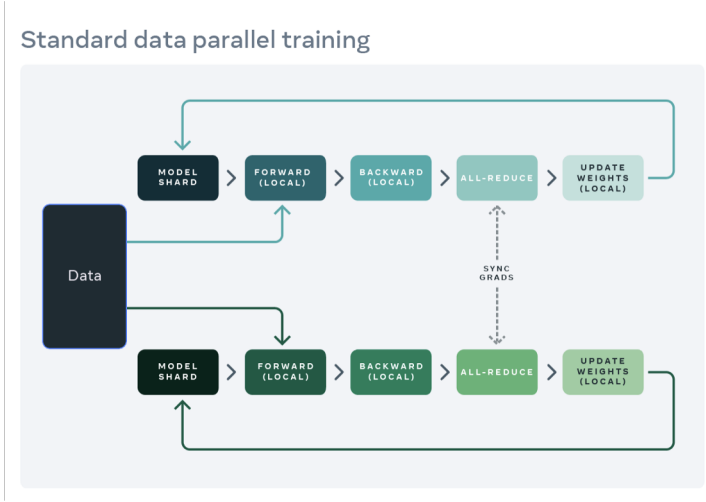
Pytorch FSDP

- **F**ully **S**harded **D**ata **P**arallel.
- Initially by Fairscale.
- Supported in PyTorch June 2023
- Meta's version
  - Partially inspired by Deepspeed
  - Very similar to ZeRO3



DDP

- Distributed Data Parallel (DDP)
  - Every worker processes a batch of data
  - Feed forward locally
  - Backward locally
  - **Ring all reduce** (gradient) across worker nodes
  - Update optimizer states and weights locally



DDP

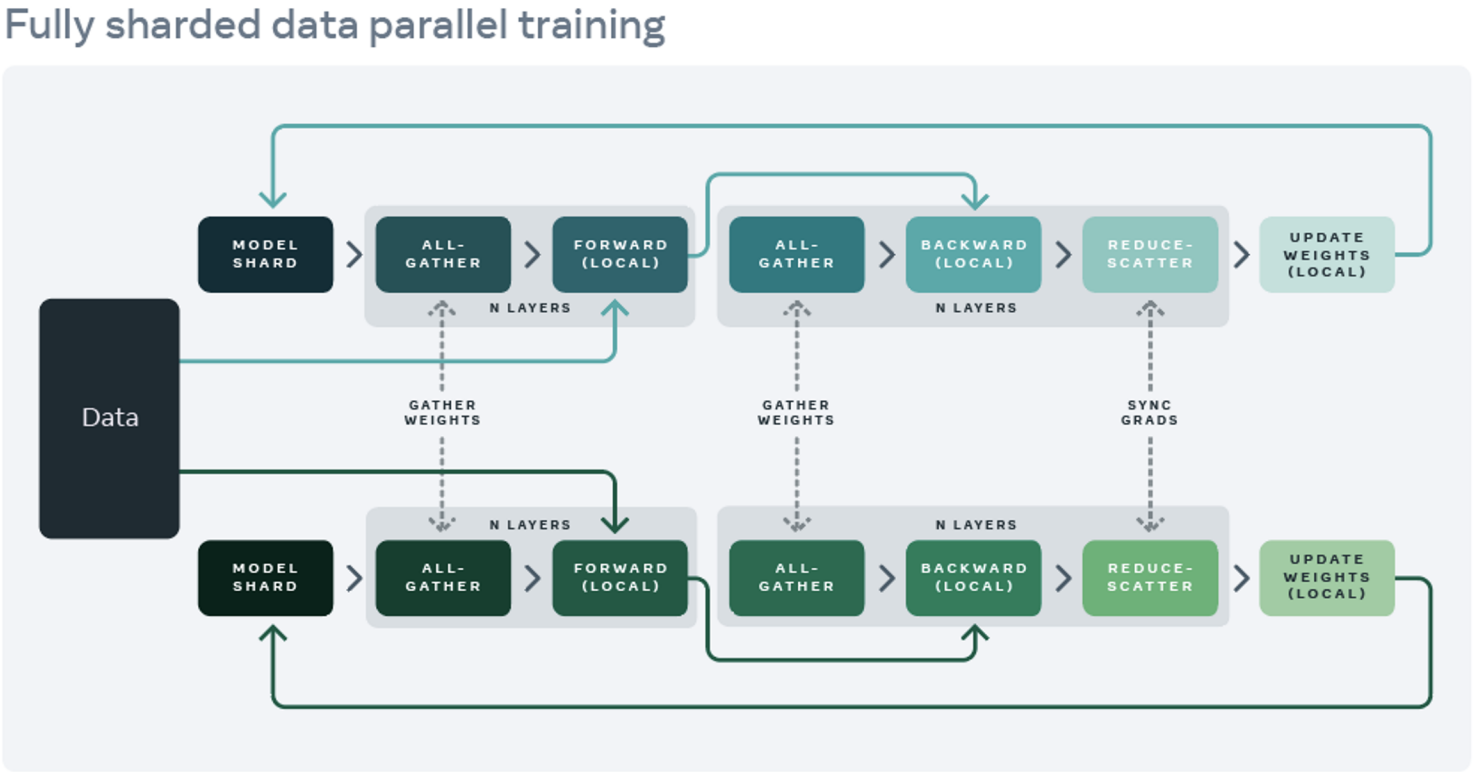
- DDP vs. DP
  - Distributed Data Parallel is multi-process parallelism; Processes can live on different machines
  - Data Parallel is a single-process multi-thread parallelism; works on a single machine (multiple GPUs)
    - Easier to use
    - Python GIL contention
    - All reduce to synchronize gradients. (send gradients to one machine)
      - Ring all reduce is more scalable.
    - DP accumulates gradient to the same .grad field;
  - Distributed Data Parallel is more scalable.
    - Use if you care about training speed.

FSDP (2021)

- All Reduce = Reduce-Scatter + All-gather

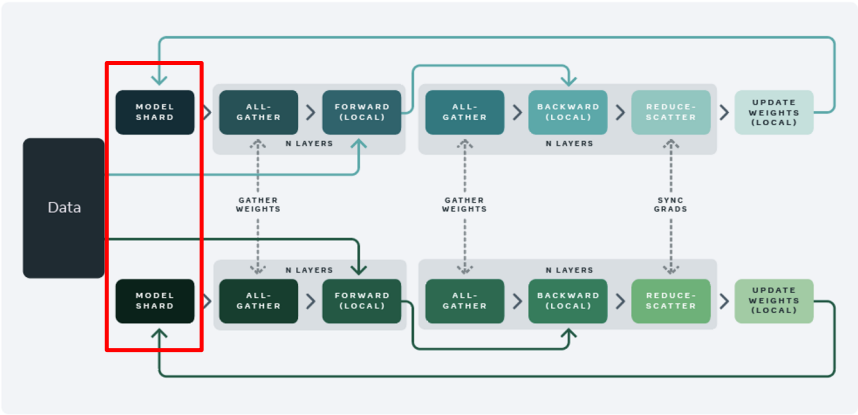


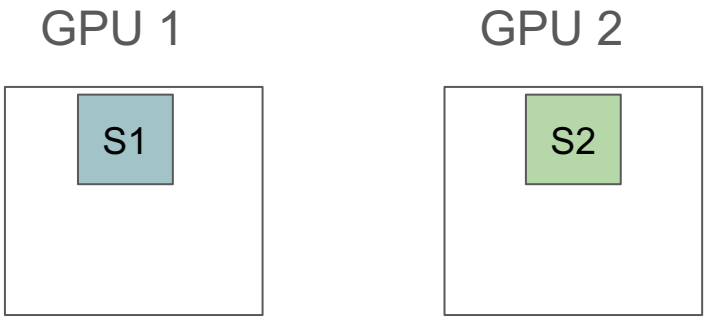
FSDP (2021)


Fully sharded data parallel training
Data
MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
N LAYERS
GATHER WEIGHTS
ALL-GATHER
BACKWARD (LOCAL)
N LAYERS
GATHER WEIGHTS
REDUCE-SCATTER
SYNC GRADS
UPDATE WEIGHTS (LOCAL)
MODEL SHARD
ALL-GATHER
FORWARD (LOCAL)
N LAYERS
ALL-GATHER
BACKWARD (LOCAL)
N LAYERS
REDUCE-SCATTER
UPDATE WEIGHTS (LOCAL)

FSDP (2021)

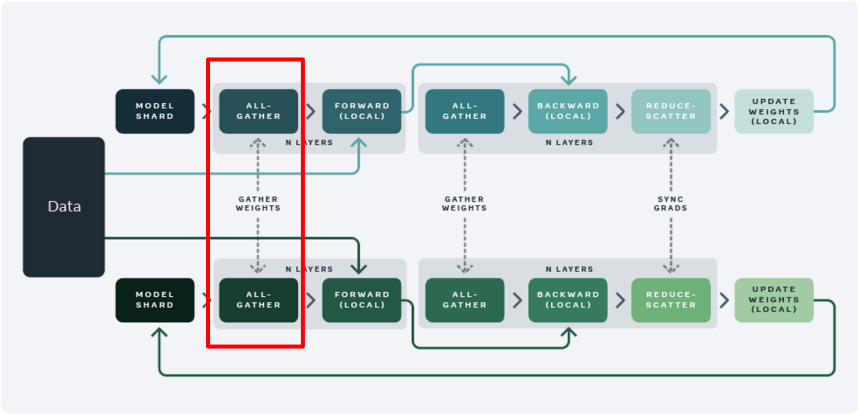
Fully sharded data parallel training

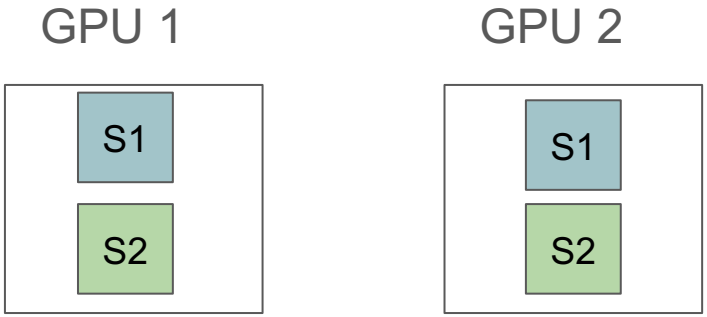




FSDP (2021)

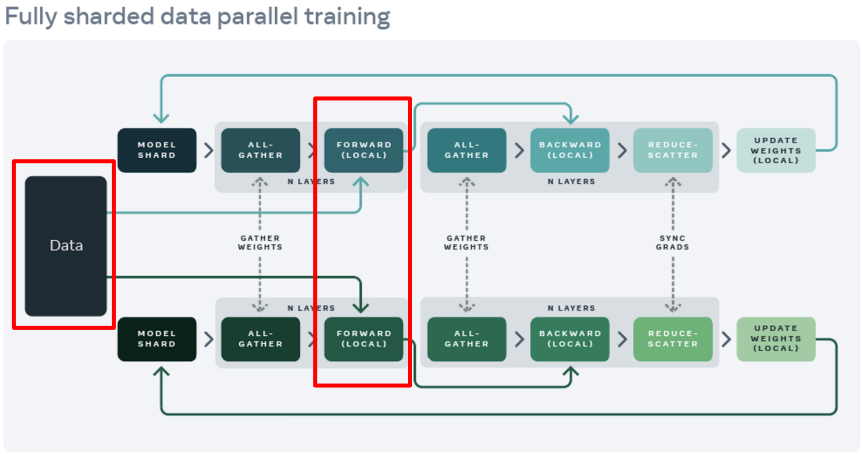
Fully sharded data parallel training

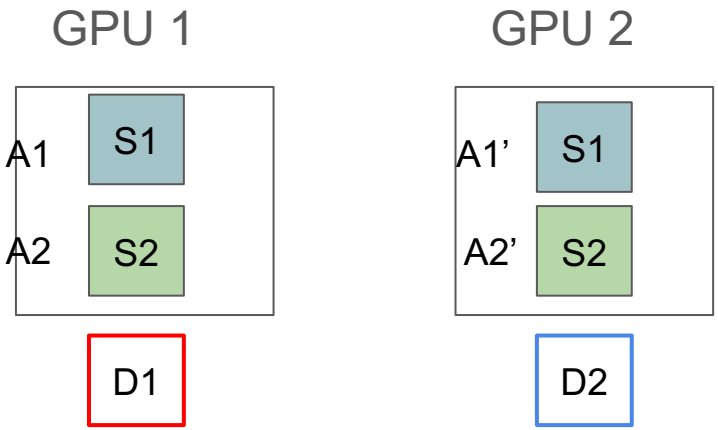




All gather model weights

FSDP (2021)



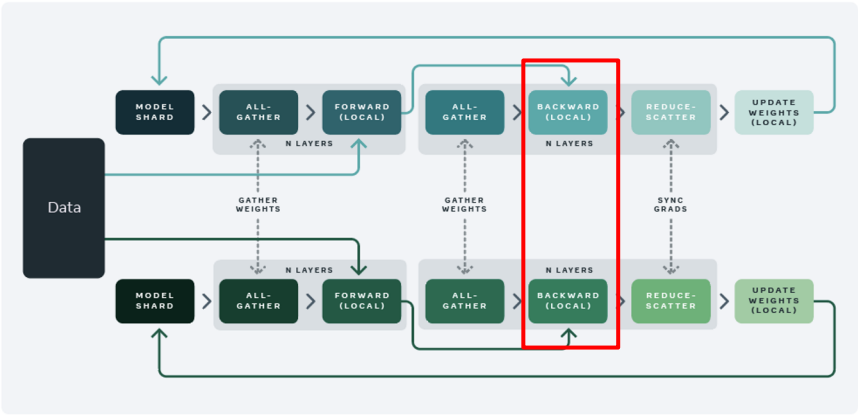


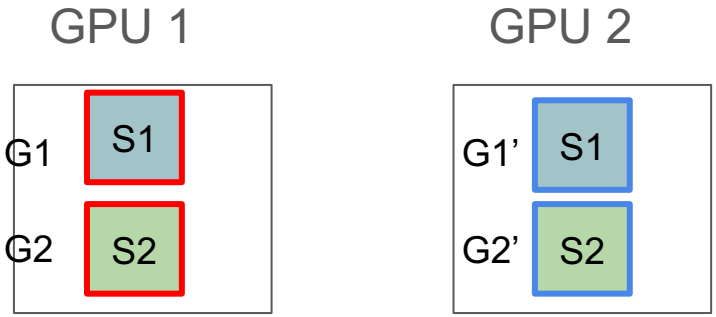
All gather model weights

Compute activations in all forward passes.

FSDP (2021)

Fully sharded data parallel training

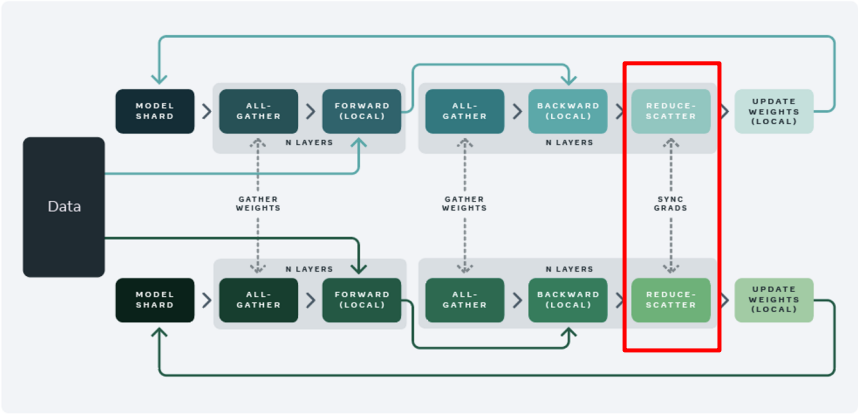


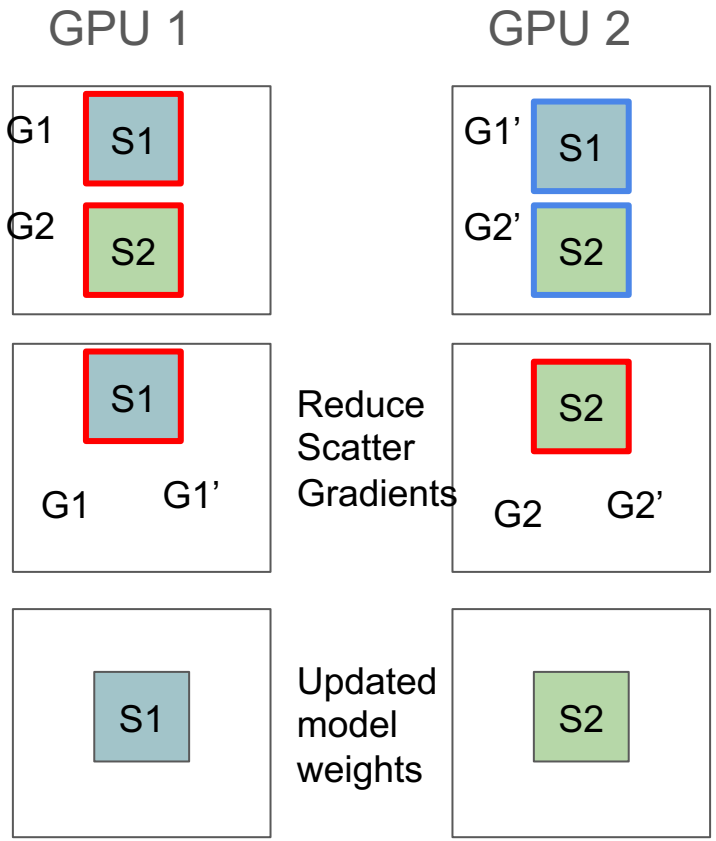


Compute gradients locally

FSDP (2021)

Fully sharded data parallel training





FSDP (2021)

- Reasons to use FSDP
  - Train LLMs
  - Trade memory for time.
    - More communications between GPUs
  - It’s simple
    - With PyTorch Lightning, Huggingface

```
model = MyModel()
trainer = Trainer(gpus=4, plugins='fsdp', precision=16)
trainer.fit(model)

trainer.test()
trainer.predict()
```

FSDP (2021)

- Reasons not to use FSDP
  - You are not training LLMs
  - Only supported in PyTorch in mid 2023
    - Dependencies
  - Incur more communication cost between GPUs.
  - Partial fine-tuning not trivial
  - lora