10-423/623: Generative AI Lecture 17 – Distributed Training

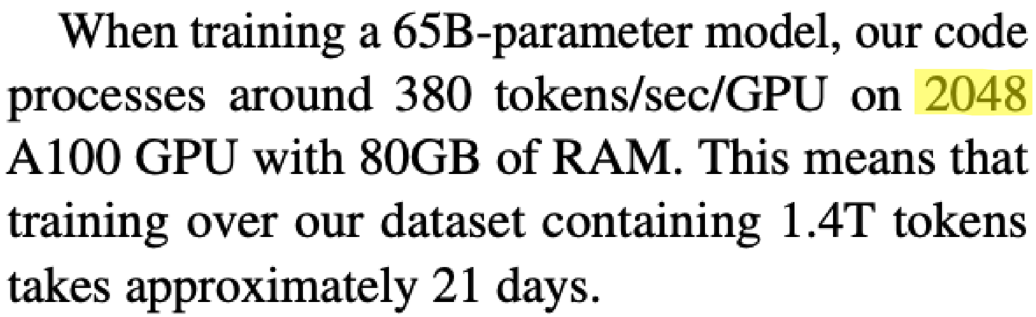
Henry Chai & Matt Gormley

10/30/24

Front Matter

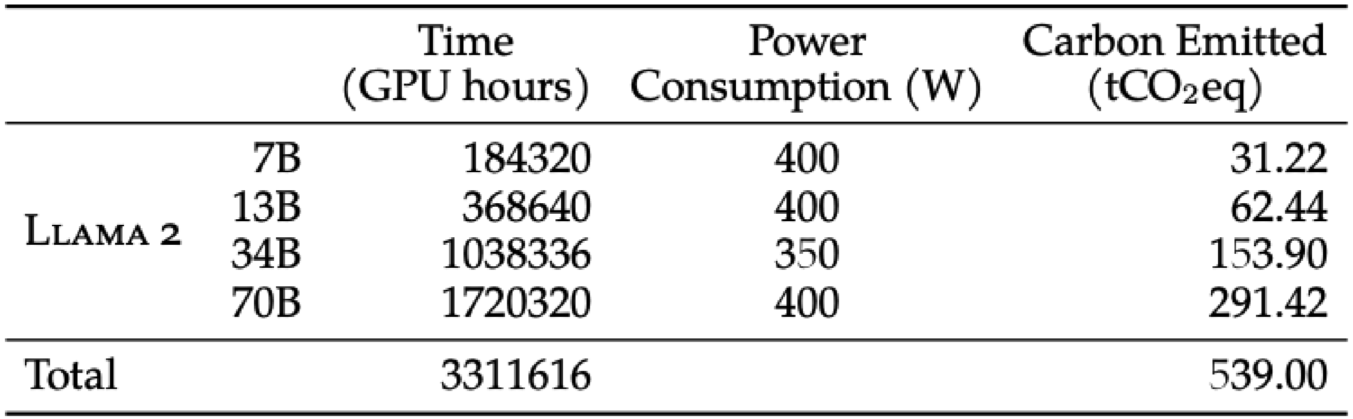
- Announcements:
  - HW4 released 10/25, due 11/5 at 11:59 PM
    - Please be mindful of your grace day usage!
  - HW623 to be released on 11/4, due 12/2 at 11:59 PM
    - Only students enrolled in 10-623 should complete HW623; **please do not submit HW623 if you are enrolled in 10-423**

Llama-1



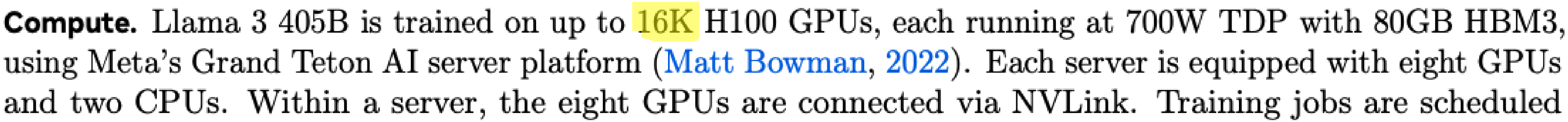
When training a 65B-parameter model, our code processes around 380 tokens/sec/GPU on 2048 A100 GPU with 80GB of RAM. This means that training over our dataset containing 1.4T tokens takes approximately 21 days.

Llama-2



		Time (GPU hours)	Power Consumption (W)	Carbon Emitted (tCO_2eq)
Llama 2	7B	184320	400	31.22
	13B	368640	400	62.44
	34B	1038336	350	153.90
	70B	1720320	400	291.42
Total		3311616		539.00

Llama-3



Compute. Llama 3 405B is trained on up to 16K H100 GPUs, each running at 700W TDP with 80GB HBM3, using Meta's Grand Teton AI server platform (Matt Bowman, 2022). Each server is equipped with eight GPUs and two CPUs. Within a server, the eight GPUs are connected via NVLink. Training jobs are scheduled

Recall: How much did it cost to train LLaMa?

Source: https://arxiv.org/pdf/1711.00937

- “the newly announced clusters both contain 24,576 Nvidia Tensor Core H100 GPUs. This is a significant increase over the original clusters, which contained 16,000 Nvidia A100 GPUs.”

Recall: How much did it cost to train LLaMa?

Source: https://www.datacenterdynamics.com/en/news/meta-reveals-details-of-two-new-24k-gpu-ai-clusters/

GPU Comparison

	NVIDIA H100	NVIDIA A100
Memory size	80 GB	80 GB
Peak memory bandwidth	1.6 TB / second	2.0 TB / second
Inter-GPU bandwidth	~ 0.6 TB / second	~ 0.9 TB / second
Price	~ $35K	~ $15K

- Key takeaway: inter-GPU communication is the primary bottleneck in distributed systems (and speeding it up is worth a lot of money!)

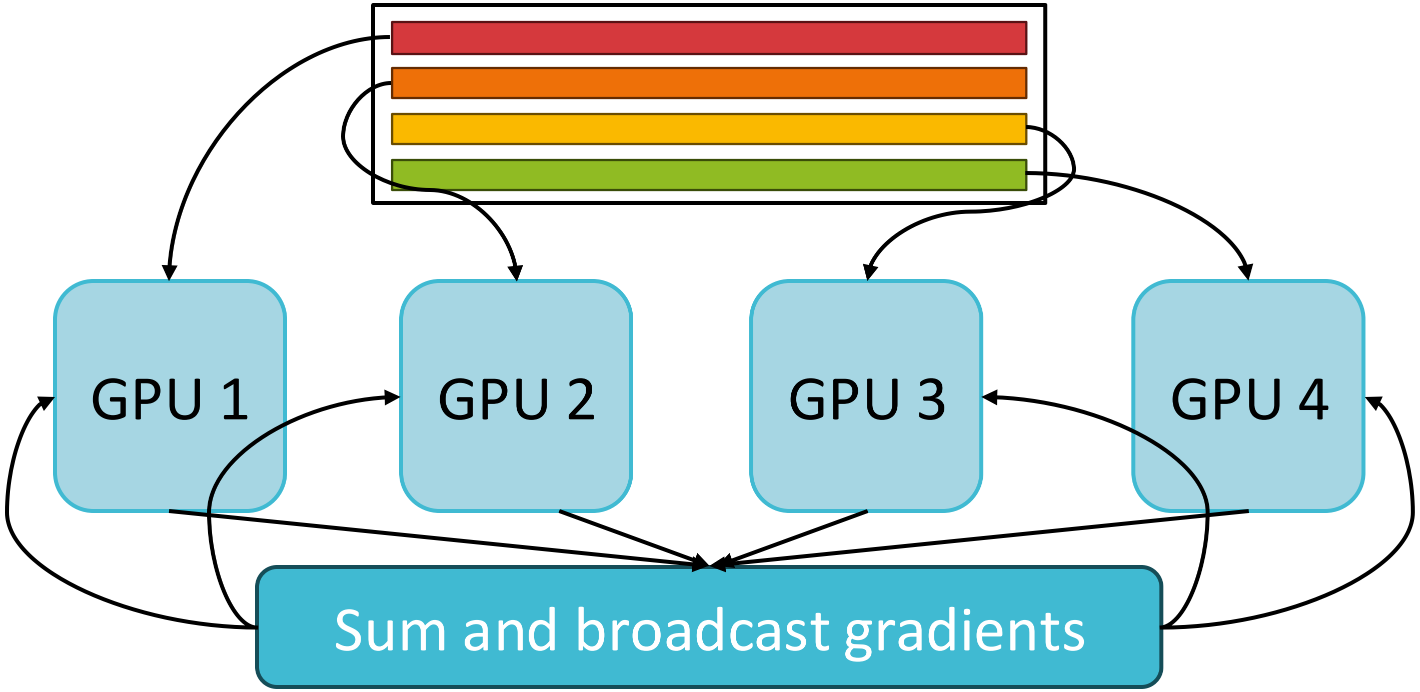
Source: https://www.nvidia.com/content/dam/en-zz/Solutions/gtcs22/data-center/h100/PB-11133-001_v01.pdf/
https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf

Parallelism in LLM Training

- Goal: divide the work of training an LLM across multiple GPUs such that inter-GPU communication is minimized
- Good news: Transformer-based LLM architectures are highly parallelizable!
- Easily exploitable parallelism in LLM training includes:
  - Data parallelism
  - Model or tensor parallelism
  - Pipeline parallelism
  - Optimizer-based parallelism
  - Token parallelism
  - Expert parallelism

Data Parallelism

- Approach: during training, split each minibatch of data points evenly across multiple GPUs and have each GPU compute the forward and backward pass for its data points.



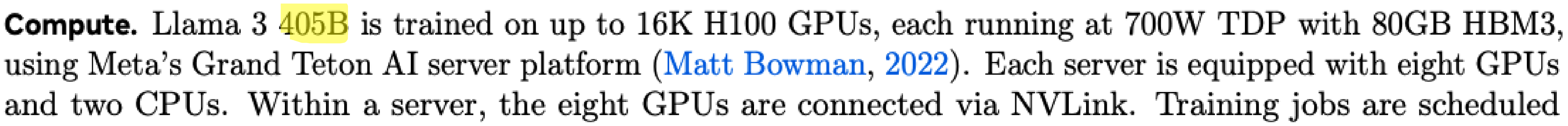
- The simplest, most effective form of parallelism *if attainable*
    - Only one inter-GPU communication per iteration

Suppose parameters are stored in FP16 => each parameter takes 16 bits = 2 bytes

405 parameters = 810 bytes = 810GB

only 80GB of RAM

Llama-3



Compute. Llama 3 405B is trained on up to 16K H100 GPUs, each running at 700W TDP with 80GB HBM3, using Meta's Grand Teton AI server platform (Matt Bowman, 2022). Each server is equipped with eight GPUs and two CPUs. Within a server, the eight GPUs are connected via NVLink. Training jobs are scheduled

How big is LLaMa-3?

Source: https://arxiv.org/pdf/1711.00937

Model Parallelism

- Approach: for a batch of data points, partition the forward and backward computations *within a layer* across multiple GPUs
  - Also called tensor parallelism
- Transformer based architectures have two primary modules that can be parallelized
  1. MLP blocks
  2. Attention blocks

Model Parallelism: MLP Blocks

- Given a batch of training data points $X \in \mathbb{R}^{N \times D}$, the operations to parallelize in MLP blocks are
  - $f(X) = \theta(Y = XW)$ in the forward pass and
  - $\nabla_X \ell\big(f(X)\big) = GW^T$ in the backward pass

for weight matrix $W \in \mathbb{R}^{D \times M}$, downstream gradient $G = \frac{\partial \ell}{\partial Y}$, and activation function θ

- Two ways of partitioning W across multiple GPUs
  - Column-parallel: $W = [c_1, c_2, \dots, c_M]$
  - Row-parallel: $W = \begin{bmatrix} r_1^T \\ r_2^T \\ \vdots \\ r_D^T \end{bmatrix}$

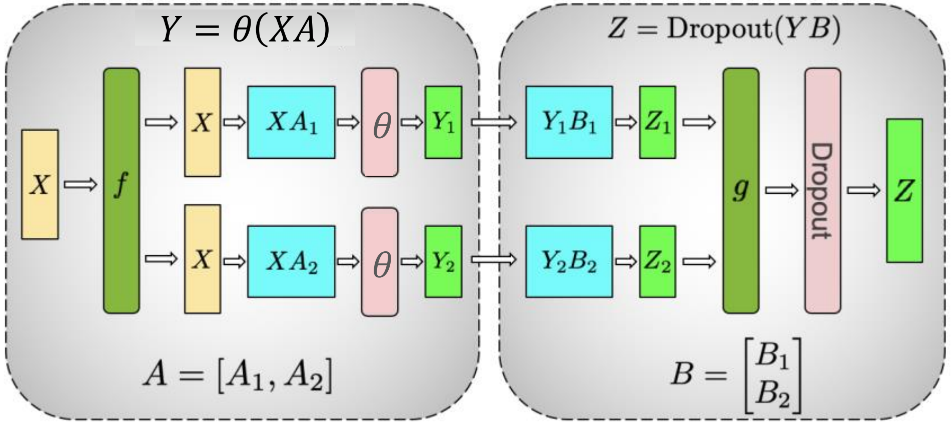
Model Parallelism: MLP Blocks

Two key observations:

1. The column-parallel and row-parallel forward and backward computations are identical, just reversed
2. Consecutive MLPs can alternate between column-parallel and row-parallel implementations to minimize the sums across GPUs!

- $f(X) = \underline{\theta(Y = XW)}$ in the forward pass and
- $\nabla_X \ell(f(X)) = \underline{GW^T}$ in the backward pass
- Column-parallel: $W = [c_1, c_2, \dots, c_M]$
  - Forward pass: $\Theta(XW) = [\Theta(Xc_1), \Theta(Xc_2), \dots, \Theta(Xc_M)]$
  - Backward pass: $[g_1, \dots, g_M]\begin{bmatrix} c_1^T \\ \vdots \\ c_M^T \end{bmatrix} = \sum_{m=1}^{M} g_m c_m^T$
- Row-parallel: $W = \begin{bmatrix} r_1^T \\ r_2^T \\ \vdots \\ r_D^T \end{bmatrix}$
  - Forward pass: $\Theta\left([x_1, \dots, x_D]\begin{bmatrix} r_1^T \\ \vdots \\ r_D^T \end{bmatrix}\right) = \Theta\left(\sum_{d=1}^{D} x_d r_d^T\right)$
  - Backward pass: $GW^T = [Gr_1, Gr_2, \dots, Gr_D]$

Model Parallelism: MLP Blocks



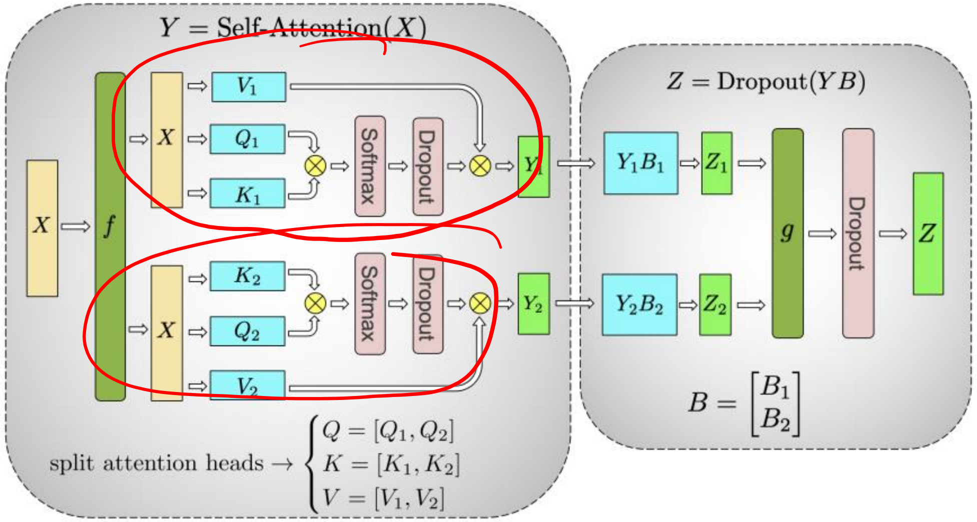
(a) MLP

Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

Source: https://arxiv.org/pdf/1909.08053

Model Parallelism: Attention Blocks

- Multi-headed attention blocks trivially parallelize across attention heads
- Assuming attention heads are concatenated horizontally, the output can be passed to a row-parallel MLP directly



(b) Self-Attention

Figure 3. Blocks of Transformer with Model Parallelism. f and g are conjugate. f is an identity operator in the forward pass and all reduce in the backward pass while g is an all reduce in the forward pass and identity in the backward pass.

Source: https://arxiv.org/pdf/1909.08053

Model Parallelism: Attention Blocks

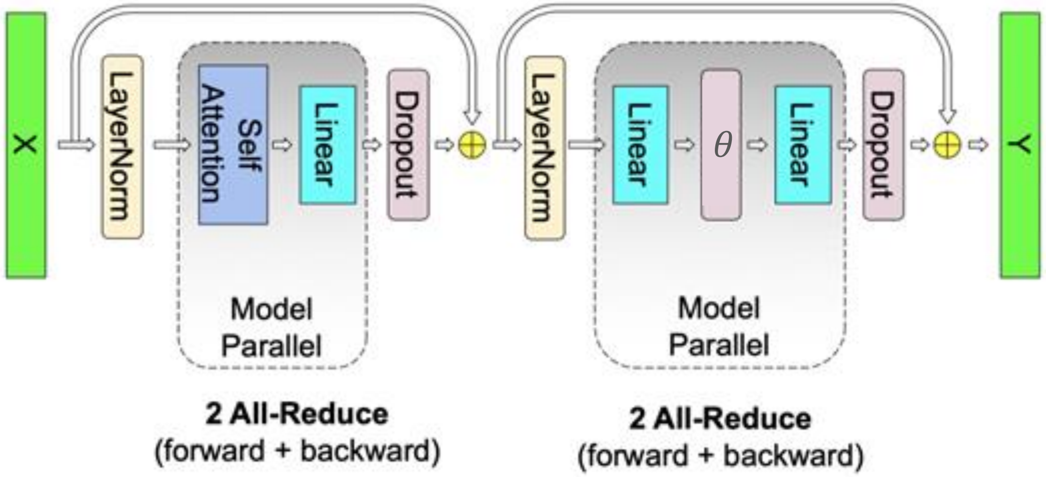
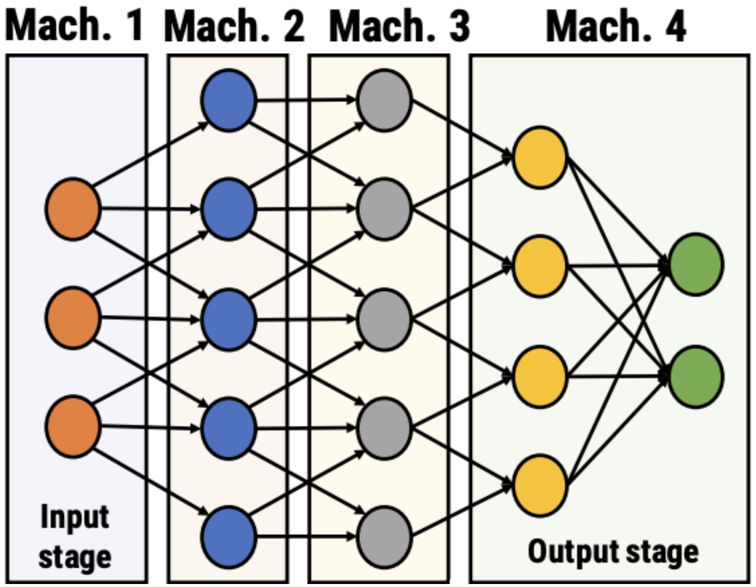


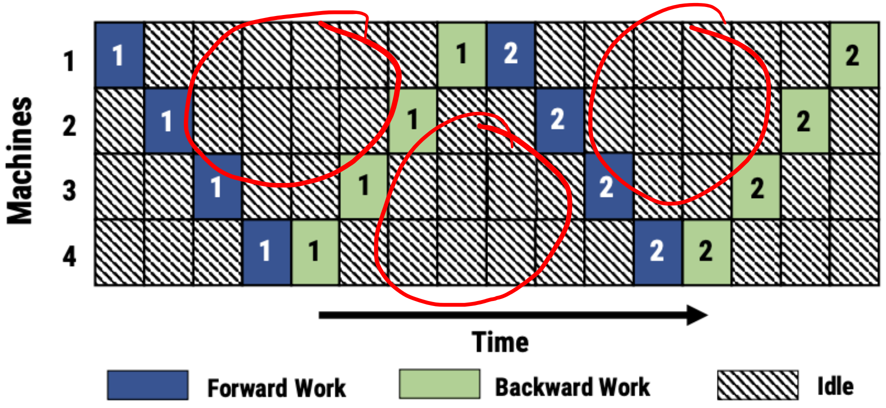
Figure 4. Communication operations in a transformer layer. There are 4 total communication operations in the forward and backward pass of a single model parallel transformer layer.

Source: https://arxiv.org/pdf/1909.08053

Pipeline Parallelism

- The previous examples of tensor parallelism also implicitly leverage *pipeline parallelism*, where different layers or modules are put on different GPUs
- Issue: the computation across layers is inherently **sequential**, not parallel!



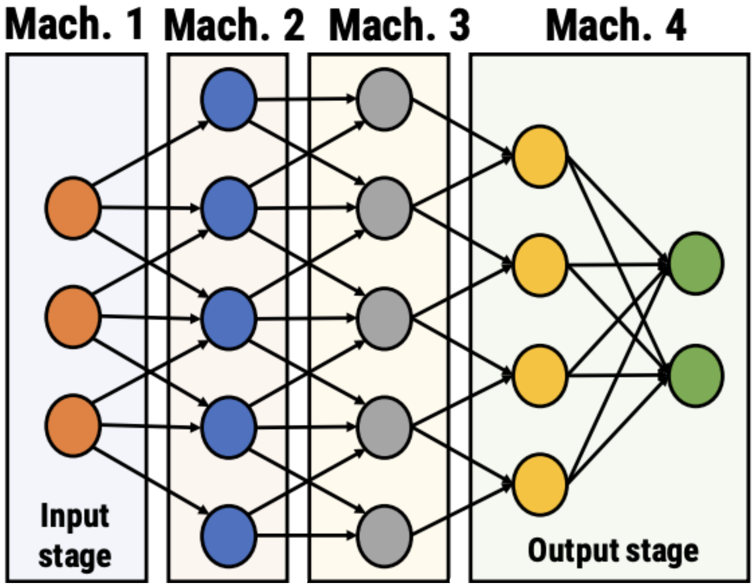


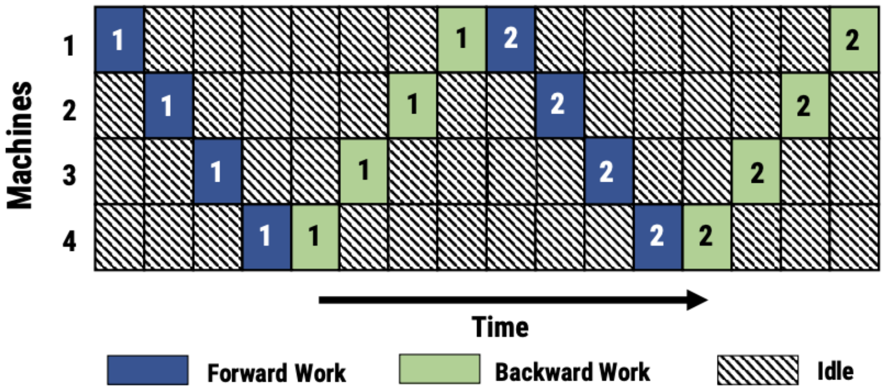
- Naïve implementation has a ton of idle GPU time!

Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism

- The previous examples of tensor parallelism also implicitly leverage *pipeline parallelism*, where different layers or modules are put on different GPUs
- Issue: the computation across layers is inherently **sequential**, not parallel!



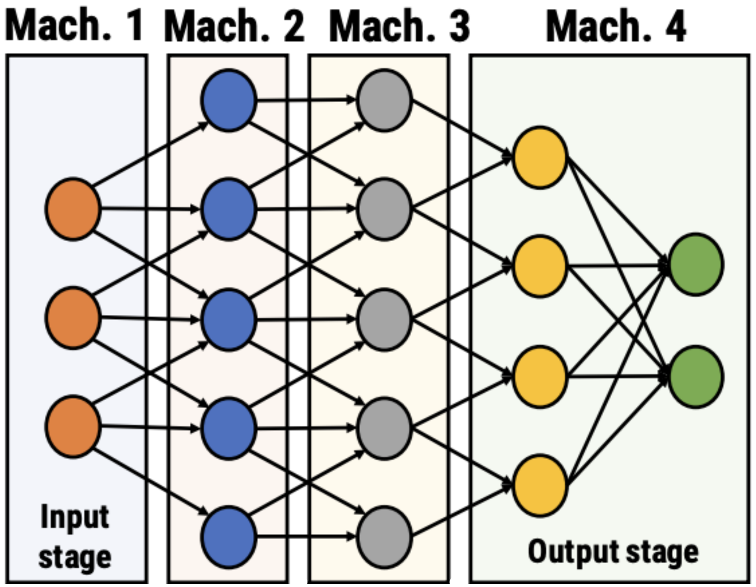


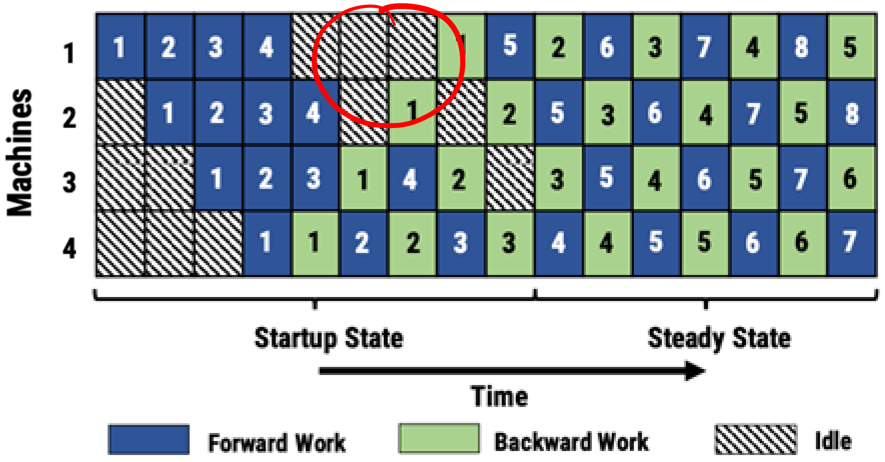
- Idea: work on multiple *microbatches* concurrently!

Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism

- The previous examples of tensor parallelism also implicitly leverage *pipeline parallelism*, where different layers or modules are put on different GPUs
- Issue: the computation across layers is inherently **sequential**, not parallel!



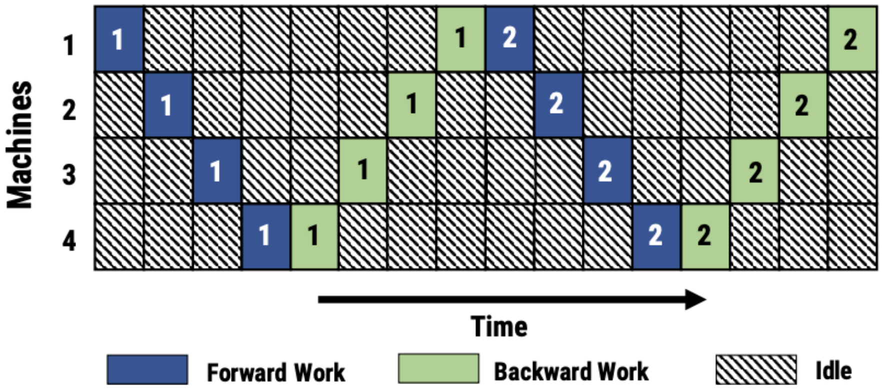


1	1	2	3	4	▨	▨	▨	1	5	2	6	3	7	4	8	5
2	▨	1	2	3	4	▨	1	▨	2	5	3	6	4	7	5	8
3	▨	▨	1	2	3	1	4	2	▨	3	5	4	6	5	7	6
4	▨	▨	▨	1	1	2	2	3	3	4	4	5	5	6	6	7

- Idea: work on multiple *microbatches* concurrently!

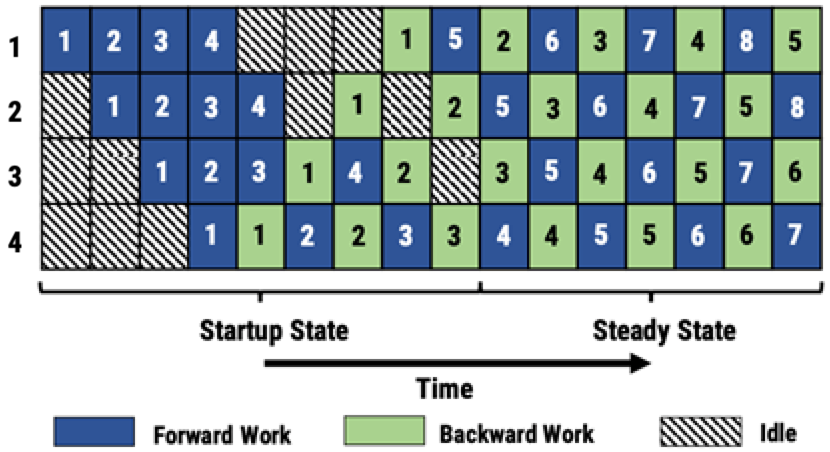
Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism



Machines																
1	🟦 1	▨	▨	▨	▨	▨	▨	🟩 1	🟦 2	▨	▨	▨	▨	▨	▨	🟩 2
2	▨	🟦 1	▨	▨	▨	▨	🟩 1	▨	▨	🟦 2	▨	▨	▨	▨	🟩 2	▨
3	▨	▨	🟦 1	▨	▨	🟩 1	▨	▨	▨	▨	🟦 2	▨	▨	🟩 2	▨	▨
4	▨	▨	▨	🟦 1	🟩 1	▨	▨	▨	▨	▨	▨	🟦 2	🟩 2	▨	▨	▨

🟦 Forward Work 🟩 Backward Work ▨ Idle



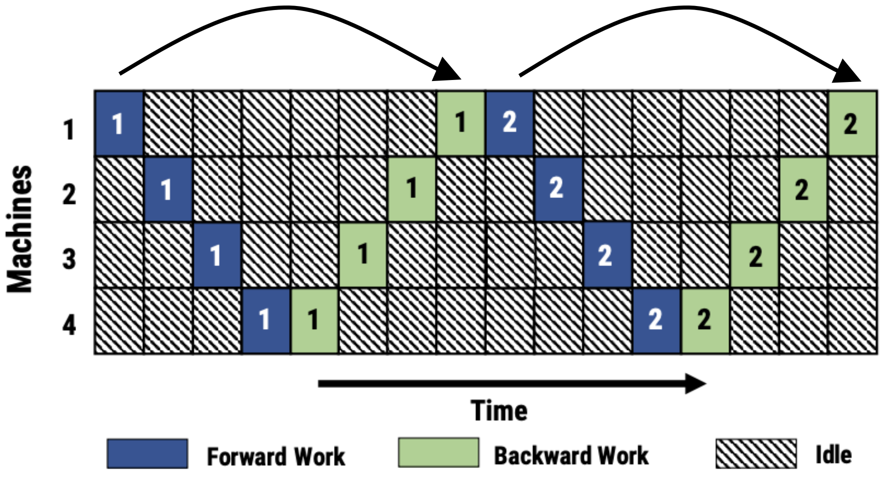
1	🟦 1	🟦 2	🟦 3	🟦 4	▨	▨	▨	🟩 1	🟦 5	🟩 2	🟦 6	🟩 3	🟦 7	🟩 4	🟦 8	🟩 5
2	▨	🟦 1	🟦 2	🟦 3	🟦 4	▨	🟩 1	▨	🟩 2	🟦 5	🟩 3	🟦 6	🟩 4	🟦 7	🟩 5	🟦 8
3	▨	▨	🟦 1	🟦 2	🟦 3	🟩 1	🟦 4	🟩 2	▨	🟩 3	🟦 5	🟩 4	🟦 6	🟩 5	🟦 7	🟩 6
4	▨	▨	▨	🟦 1	🟩 1	🟦 2	🟩 2	🟦 3	🟩 3	🟦 4	🟩 4	🟦 5	🟩 5	🟦 6	🟩 6	🟦 7

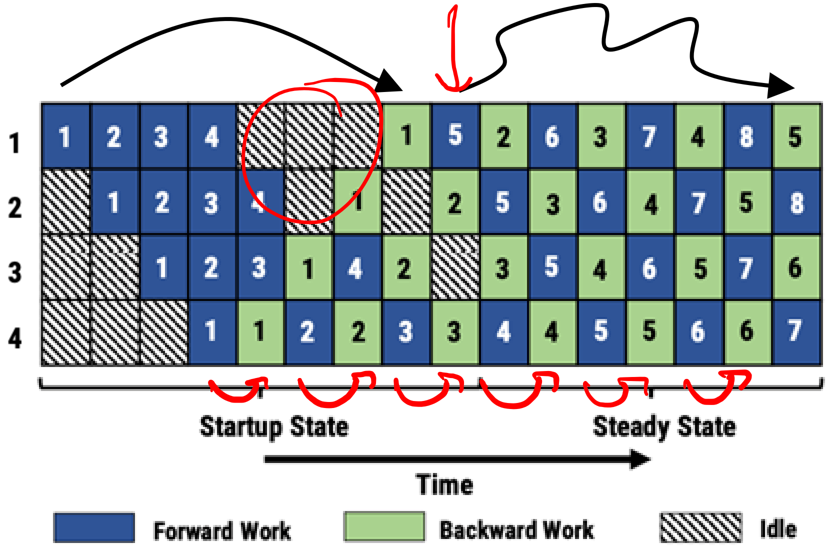
🟦 Forward Work 🟩 Backward Work ▨ Idle

- The schedule above is the one-forward-one-backward (1F1B) mechanism: in the steady state, each GPU alternates between one forward pass and one backward pass

Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism

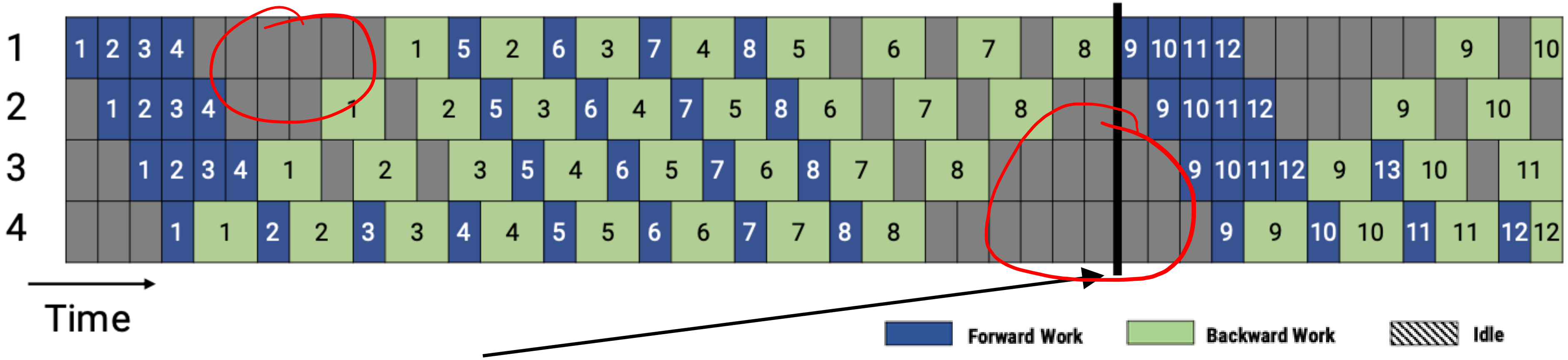




- Issue: if weights are updated in every backward pass, the weights used to compute the forward pass for a microbatch can be different from the weights used to compute the backward pass
  - The divergence is worse for earlier GPUs in the pipeline
  - Can lead to poor model convergence/optimization
- Solution: weight stashing – after every forward pass, store the weights and reload them for the corresponding backward pass

Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism



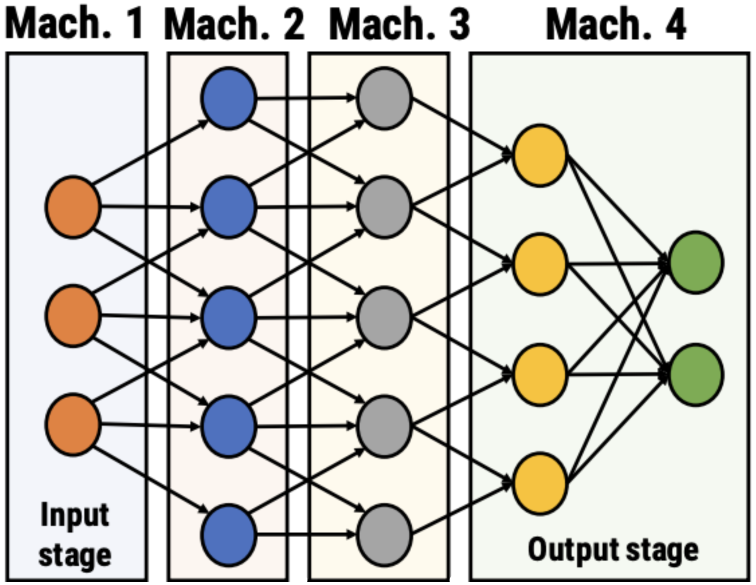
- Indicates a pipeline flush between minibatches; the sync is required to compute the parameters for the next minibatch's forward pass

- The 1F1B is still highly efficient even when forward and backward passes take different amounts of compute (as is typically the case)

Source: https://arxiv.org/pdf/2104.04473

Pipeline Parallelism

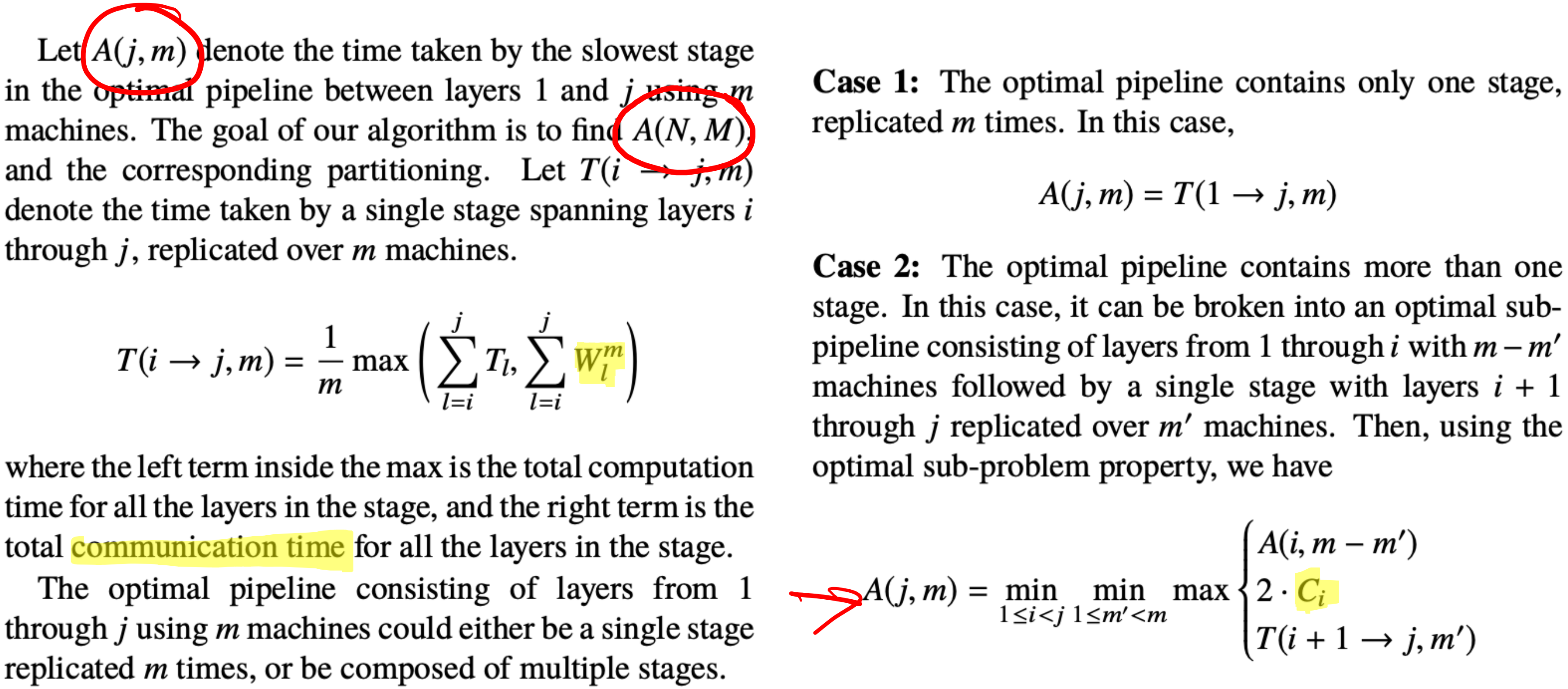
- Another interesting question that arises with pipeline parallelism: how should we partition the layers across GPUs?
- Solution: profile the code and apply dynamic programming!



Source: https://arxiv.org/pdf/1806.03377

Pipeline Parallelism

- Another interesting question that arises with pipeline parallelism: how should we partition the layers across GPUs?
- Solution: profile the code and apply dynamic programming!



Let $A(j, m)$ denote the time taken by the slowest stage in the optimal pipeline between layers 1 and j using m machines. The goal of our algorithm is to find $A(N, M)$, and the corresponding partitioning. Let $T(i \to j, m)$ denote the time taken by a single stage spanning layers i through j, replicated over m machines.

$$T(i \to j, m) = \frac{1}{m} \max \left(\sum_{l=i}^{j} T_l, \sum_{l=i}^{j} W_l^m \right)$$

where the left term inside the max is the total computation time for all the layers in the stage, and the right term is the total communication time for all the layers in the stage.

The optimal pipeline consisting of layers from 1 through j using m machines could either be a single stage replicated m times, or be composed of multiple stages.

Case 1: The optimal pipeline contains only one stage, replicated m times. In this case,

$$A(j, m) = T(1 \to j, m)$$

Case 2: The optimal pipeline contains more than one stage. In this case, it can be broken into an optimal sub-pipeline consisting of layers from 1 through i with $m - m'$ machines followed by a single stage with layers $i + 1$ through j replicated over m' machines. Then, using the optimal sub-problem property, we have

$$A(j, m) = \min_{1 \le i < j} \min_{1 \le m' < m} \max \begin{cases} A(i, m - m') \\ 2 \cdot C_i \\ T(i + 1 \to j, m') \end{cases}$$

Source: https://arxiv.org/pdf/1806.03377

The story so far:

Ultimately, data parallelism is still king and tensor / pipeline parallelism are used to support pushing more data through the forward and backward passes

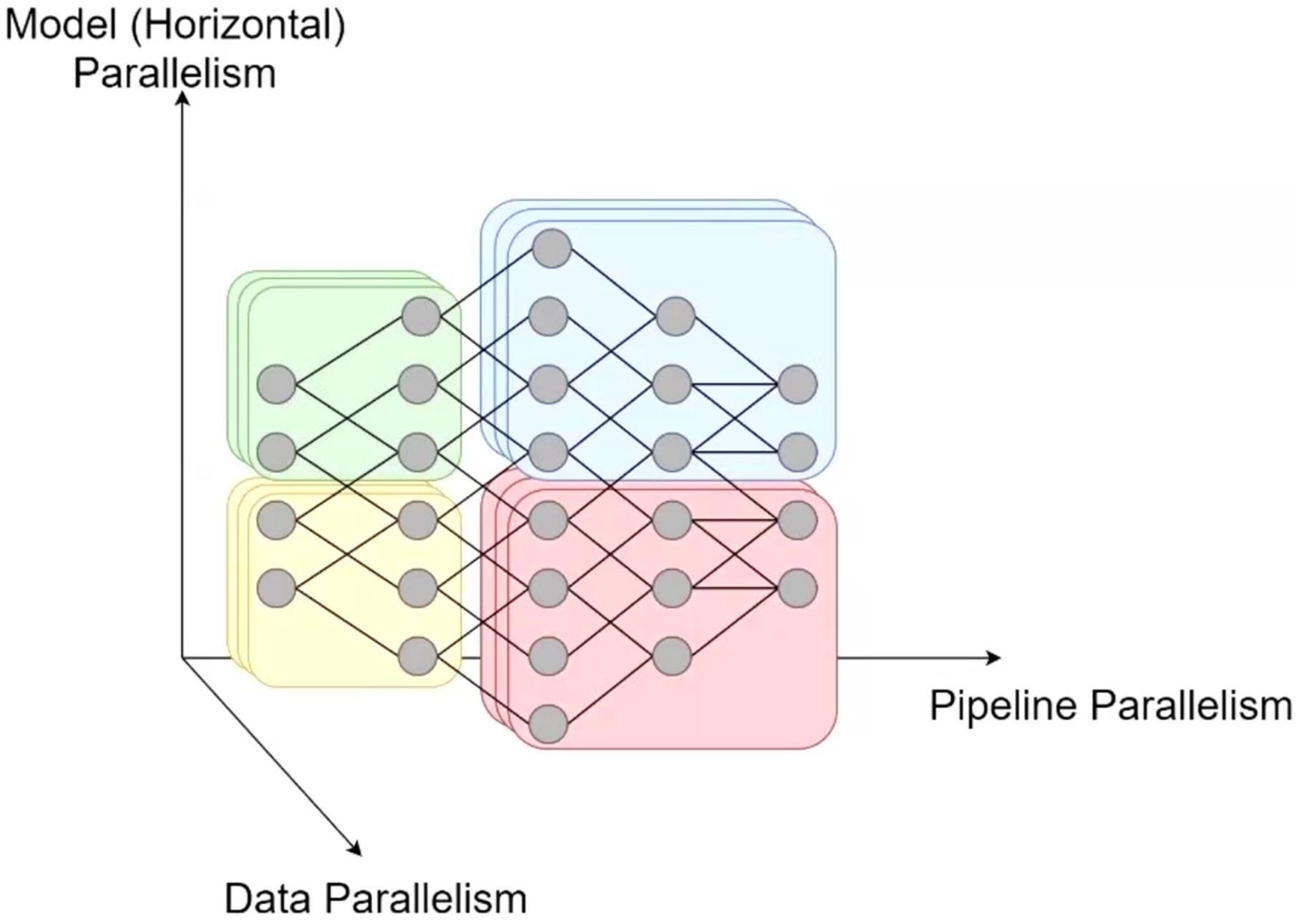


Figure courtesy of Yuanzhi Li

Optimizer Parallelism

- If we only need to compute/store the parameters and gradients, then data + model + pipeline parallelism (typically) suffices
  - E.g., for mini-batch SGD, the update only requires the gradients and the current weights:

$$W^{(t+1)} \leftarrow W^{(t)} - \gamma \nabla \underline{\ell^{(B)}\left(W^{(t)}\right)}$$

- However, many advanced optimization algorithms require storing additional intermediate or *state variables* to perform the parameter update

Adam (Adaptive Moment Estimation)

- High-level intuition: Adam combines SGD with momentum (memory of previous gradient steps) and RMSProp (scaled step sizes based on previous gradients)

$$W^{(t+1)} \leftarrow W^{(t)} - \frac{\boxed{\gamma}}{\sqrt{\underline{S_t}/(1-\beta_2^t)}} \underset{\uparrow}{\odot} \left(\frac{M_t}{1-\beta_1^t}\right)$$

where

- $M_t = \beta_1 M_{t-1} + (1-\beta_1)\underbrace{\nabla \ell^{(B)}\left(W^{(t)}\right)}$
- $S_t = \beta_2 S_{t-1} + (1-\beta_2)\left(\nabla \ell^{(B)}\left(W^{(t)}\right) \odot \nabla \ell^{(B)}\left(W^{(t)}\right)\right)$
- β_1 and β_2 are *decay* parameters that dictate how much M_t and S_t are defined by previous time steps
- M_{-1} and S_{-1} are initialized to matrices of all zeros

Adam (Adaptive Moment Estimation)

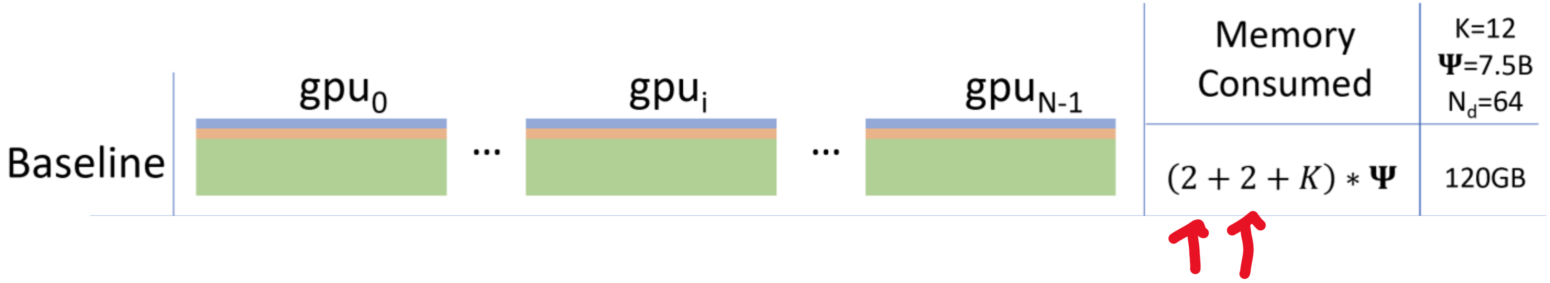
- High-level intuition: Adam combines SGD with momentum (memory of previous gradient steps) and RMSProp (scaled step sizes based on previous gradients)

$$W^{(t+1)} \leftarrow W^{(t)} - \frac{\gamma}{\sqrt{S_t/(1-\beta_2^t)}} \odot \left(\frac{M_t}{1-\beta_1^t}\right)$$

where

- $M_t = \beta_1 M_{t-1} + (1-\beta_1)\nabla\ell^{(B)}\left(W^{(t)}\right)$
- $S_t = \beta_2 S_{t-1} + (1-\beta_2)\left(\nabla\ell^{(B)}\left(W^{(t)}\right) \odot \nabla\ell^{(B)}\left(W^{(t)}\right)\right)$
- Mixed-precision training: M_t and S_t are the same dimensionality as $W^{(t)}$ and are typically stored in FP32 (instead of FP16) because of the squared term in S_t

Optimizer Parallelism



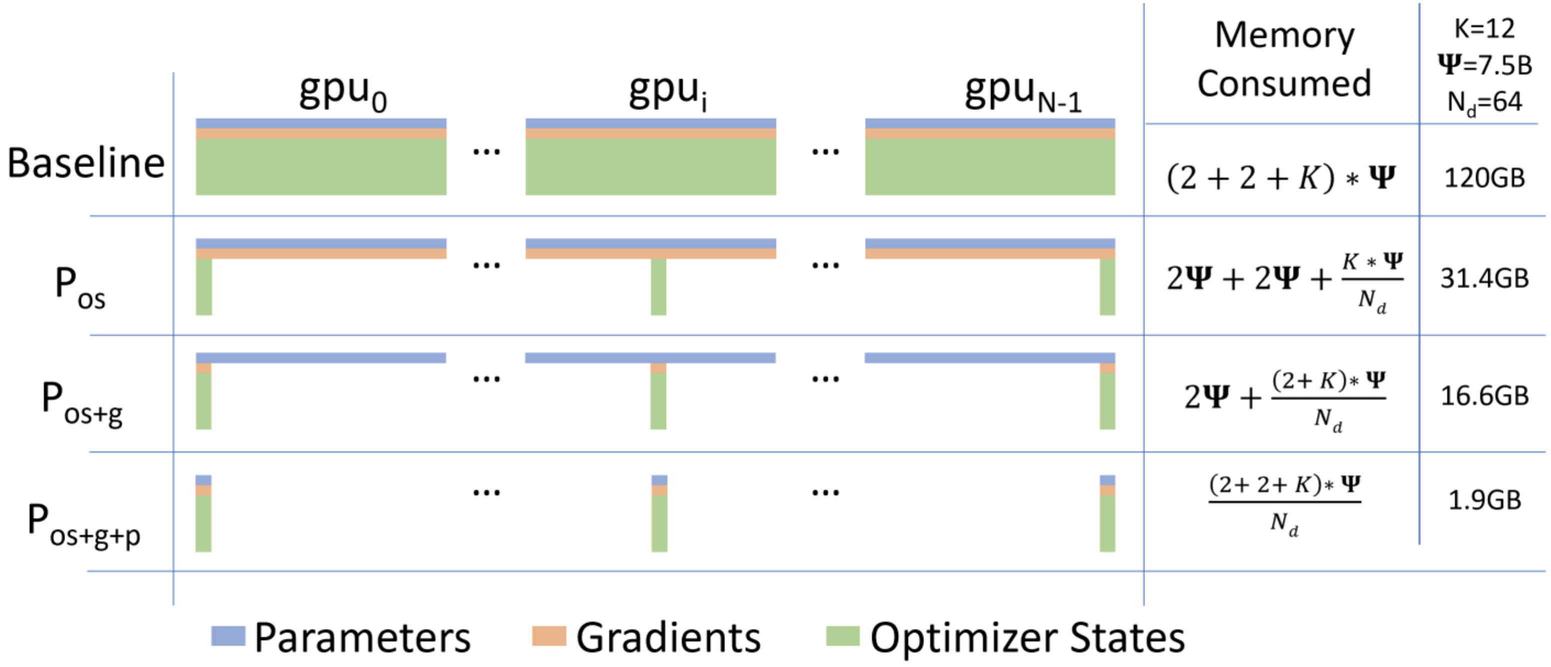
- For Adam, $K = 12 = 3 * 4$ because each parameter in $W^{(t)}$, M_t, and S_t is stored in FP32 = 4 bytes per parameter

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

Source: https://arxiv.org/pdf/1910.02054

Optimizer Parallelism



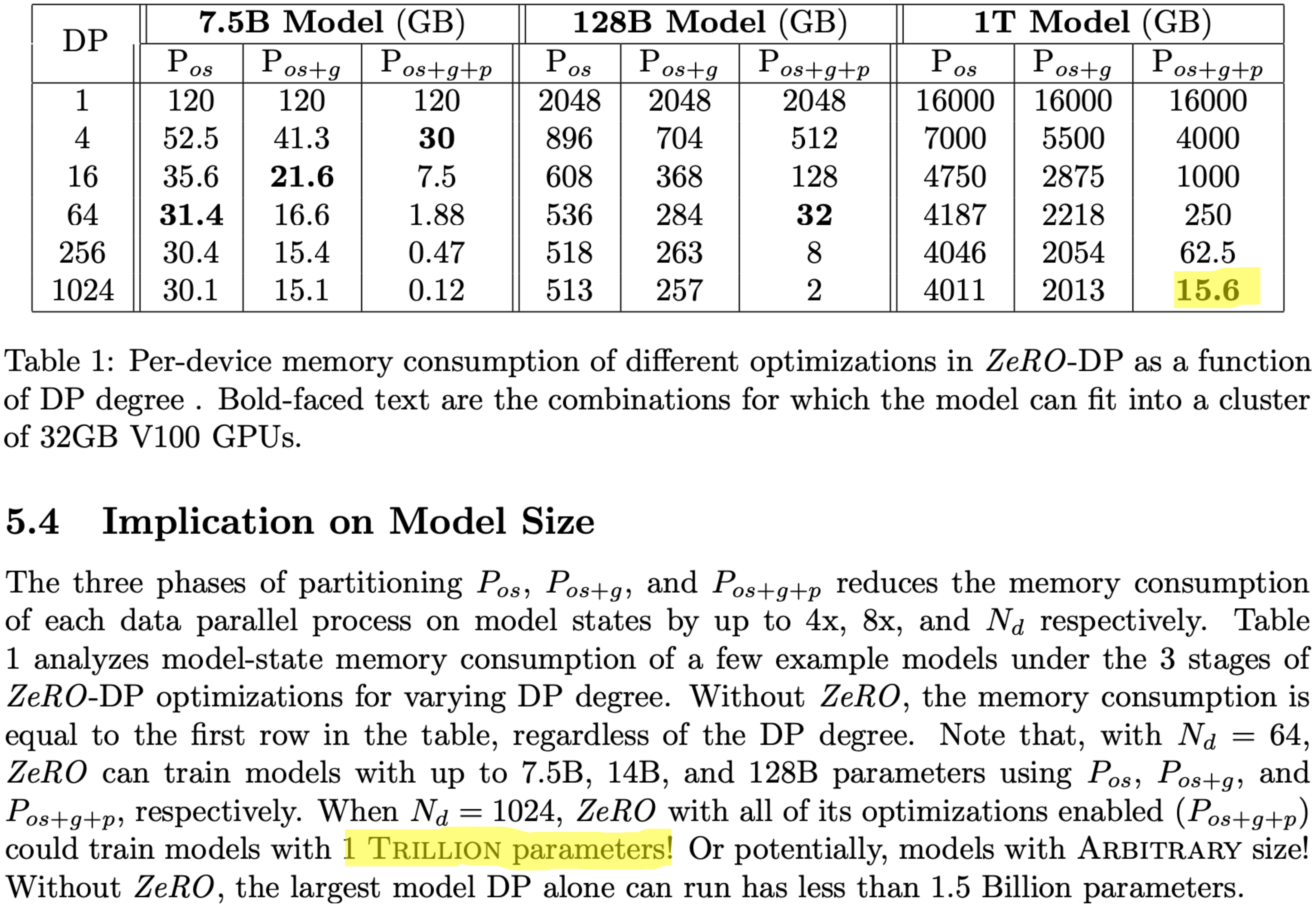
	Memory Consumed	K=12 $\boldsymbol{\Psi}$=7.5B N_d=64
Baseline	$(2+2+K)*\boldsymbol{\Psi}$	120GB
P_{os}	$2\boldsymbol{\Psi}+2\boldsymbol{\Psi}+\frac{K*\boldsymbol{\Psi}}{N_d}$	31.4GB
P_{os+g}	$2\boldsymbol{\Psi}+\frac{(2+K)*\boldsymbol{\Psi}}{N_d}$	16.6GB
P_{os+g+p}	$\frac{(2+2+K)*\boldsymbol{\Psi}}{N_d}$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

Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

(ZeRO-DP = Zero redundancy optimizer powered data parallelism)

Source: https://arxiv.org/pdf/1910.02054

Optimizer Parallelism



DP	**7.5B Model** (GB)			**128B Model** (GB)			**1T Model** (GB)		
	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}
1	120	120	120	2048	2048	2048	16000	16000	16000
4	52.5	41.3	**30**	896	704	512	7000	5500	4000
16	35.6	**21.6**	7.5	608	368	128	4750	2875	1000
64	**31.4**	16.6	1.88	536	284	**32**	4187	2218	250
256	30.4	15.4	0.47	518	263	8	4046	2054	62.5
1024	30.1	15.1	0.12	513	257	2	4011	2013	**15.6**

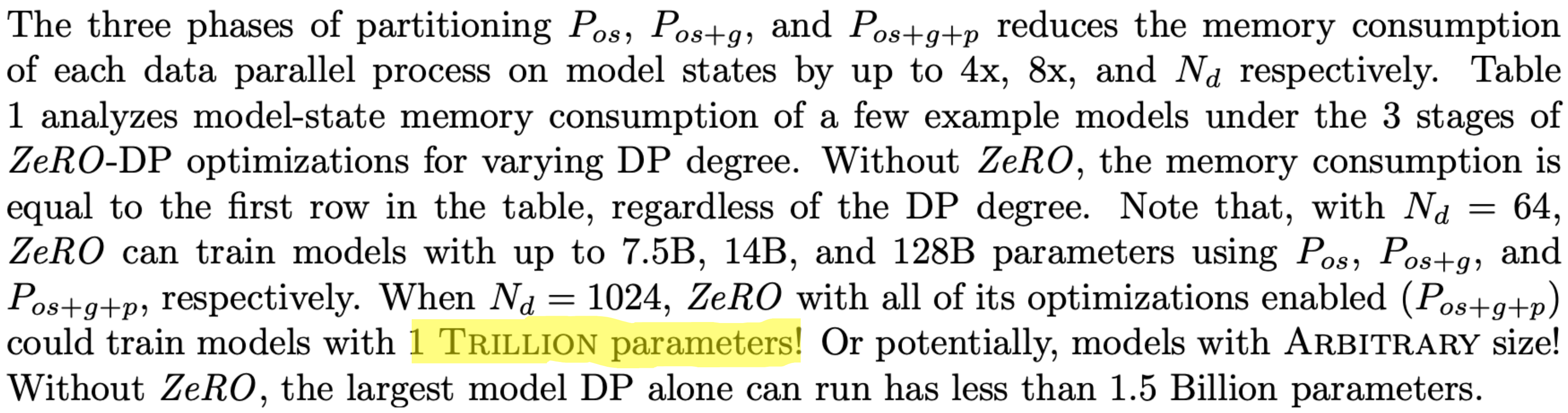
Table 1: Per-device memory consumption of different optimizations in *ZeRO*-DP as a function of DP degree . Bold-faced text are the combinations for which the model can fit into a cluster of 32GB V100 GPUs.

5.4 Implication on Model Size

The three phases of partitioning P_{os}, P_{os+g}, and P_{os+g+p} reduces the memory consumption of each data parallel process on model states by up to 4x, 8x, and N_d respectively. Table 1 analyzes model-state memory consumption of a few example models under the 3 stages of *ZeRO*-DP optimizations for varying DP degree. Without *ZeRO*, the memory consumption is equal to the first row in the table, regardless of the DP degree. Note that, with $N_d = 64$, *ZeRO* can train models with up to 7.5B, 14B, and 128B parameters using P_{os}, P_{os+g}, and P_{os+g+p}, respectively. When $N_d = 1024$, *ZeRO* with all of its optimizations enabled (P_{os+g+p}) could train models with 1 TRILLION parameters! Or potentially, models with ARBITRARY size! Without *ZeRO*, the largest model DP alone can run has less than 1.5 Billion parameters.

Source: https://arxiv.org/pdf/1910.02054

Optimizer Parallelism


The three phases of partitioning P_{os}, P_{os+g}, and P_{os+g+p} reduces the memory consumption of each data parallel process on model states by up to 4x, 8x, and N_d respectively. Table 1 analyzes model-state memory consumption of a few example models under the 3 stages of *ZeRO*-DP optimizations for varying DP degree. Without *ZeRO*, the memory consumption is equal to the first row in the table, regardless of the DP degree. Note that, with $N_d = 64$, *ZeRO* can train models with up to 7.5B, 14B, and 128B parameters using P_{os}, P_{os+g}, and P_{os+g+p}, respectively. When $N_d = 1024$, *ZeRO* with all of its optimizations enabled (P_{os+g+p}) could train models with 1 Trillion parameters! Or potentially, models with Arbitrary size! Without *ZeRO*, the largest model DP alone can run has less than 1.5 Billion parameters.

Source: https://arxiv.org/pdf/1910.02054

Recall: How large are LLMs?

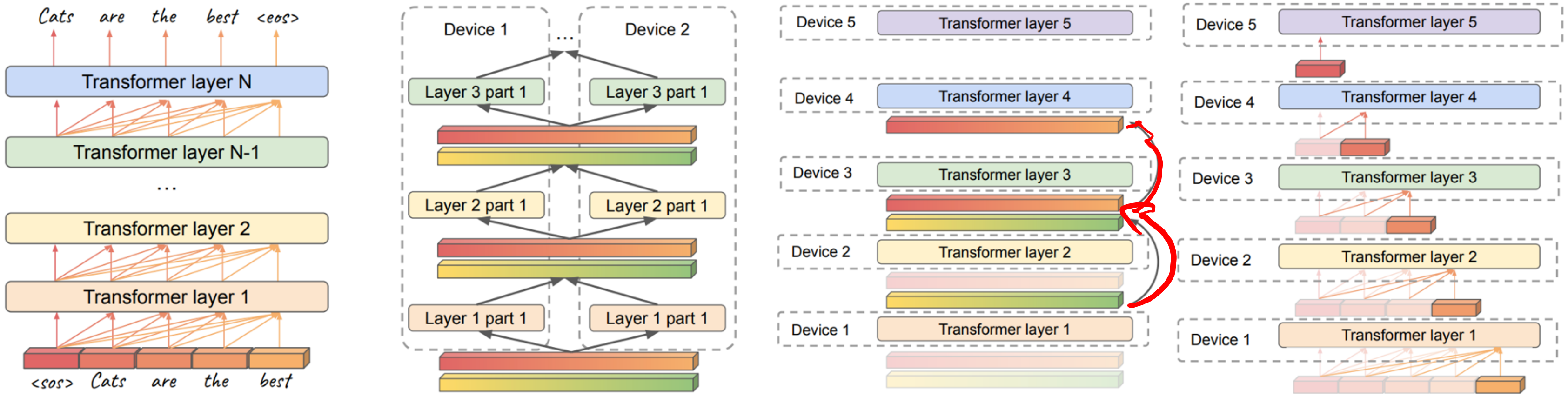
Model	Creators	Year of release	Training Data (# tokens)	Model Size (# parameters)
GPT-2	OpenAI	2019	~10 billion (40Gb)	1.5 billion
GPT-3 (cf. ChatGPT)	OpenAI	2020	300 billion	175 billion
PaLM	Google	2022	780 billion	540 billion
Chinchilla	DeepMind	2022	1.4 trillion	70 billion
LaMDA (cf. Bard)	Google	2022	1.56 trillion	137 billion
LLaMA	Meta	2023	1.4 trillion	65 billion
LLaMA-2	Meta	2023	2 trillion	70 billion
GPT-4	OpenAI	2023	?	**? (1.76 trillion)**
Gemini (Ultra)	Google	2023	?	**? (1.5 trillion)**
LLaMA-3	Meta	2024	15 trillion	405 billion

Parallelism in LLM Training

- Goal: divide the work of training an LLM across multiple GPUs such that inter-GPU communication is minimized
- Good news: Transformer-based LLM architectures are highly parallelizable!
- Easily exploitable parallelism in LLM training includes:
  - Data parallelism
  - Model or tensor parallelism
  - Pipeline parallelism
  - Optimizer-based parallelism
  - **Token parallelism**
  - **Expert parallelism**

Token Parallelism

- Insight: in a transformer LM with causal attention, the computation for each *token* in some transformer block only depends on the previous tokens at that layer
- Idea: instead of waiting for the entire previous layer to finish, start working on token t in layer l as soon as tokens 1 through $t - 1$ in layer $l - 1$ are done



(a) Transformer-based LM

(b) Operation partitioning (Megatron-LM)

(c) Microbatch-based pipeline parallelism (GPipe)

(d) Token-based pipeline parallelism (TeraPipe)

Source: https://arxiv.org/pdf/2102.07988

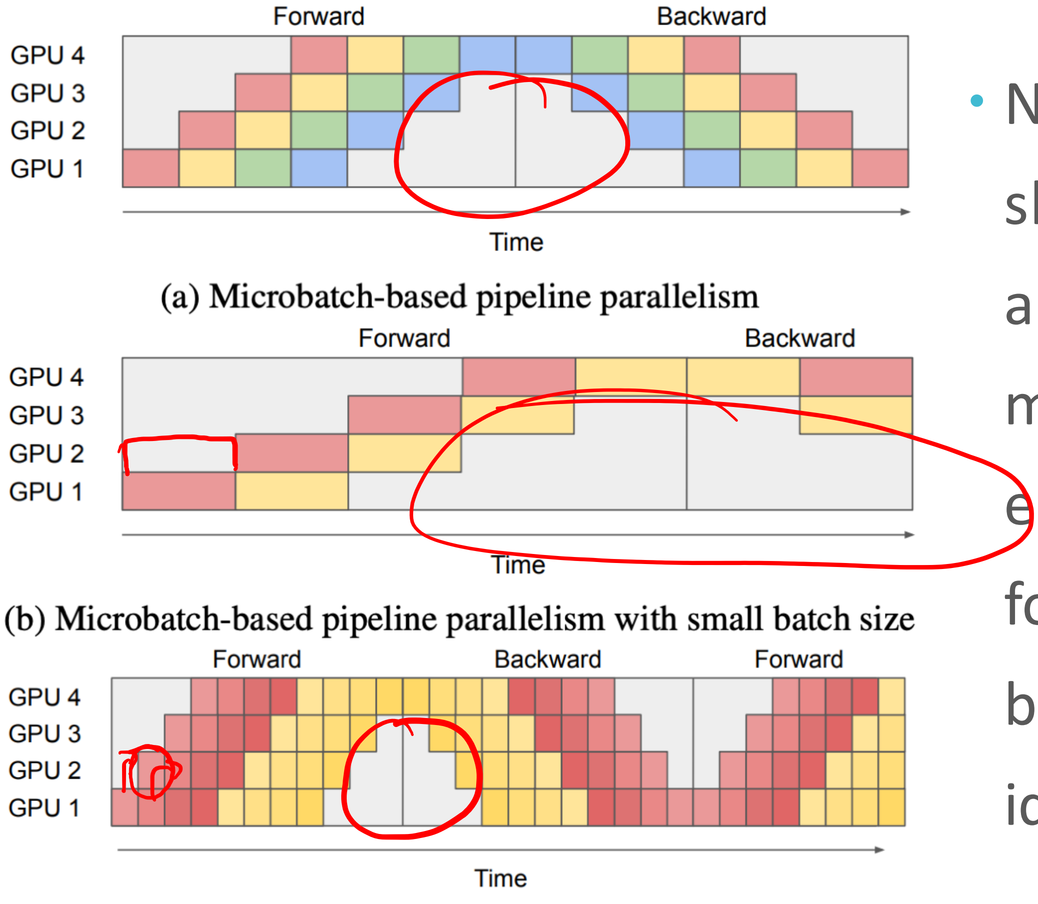
Token Parallelism

- Insight: in a transformer LM, the computation for each *token* in some transformer block only depends on the previous tokens at that layer

- Idea: instead of waiting for the entire previous layer to finish, start working on token t in layer l as soon as tokens 1 through $t - 1$ in layer $l - 1$ are done

- Intuition: increasing pipeline granularity reduces idle time in the pipeline!
  - Efficiency gains scale with sequence lengths and models have been moving towards longer contexts

Token Parallelism



(a) Microbatch-based pipeline parallelism

(b) Microbatch-based pipeline parallelism with small batch size

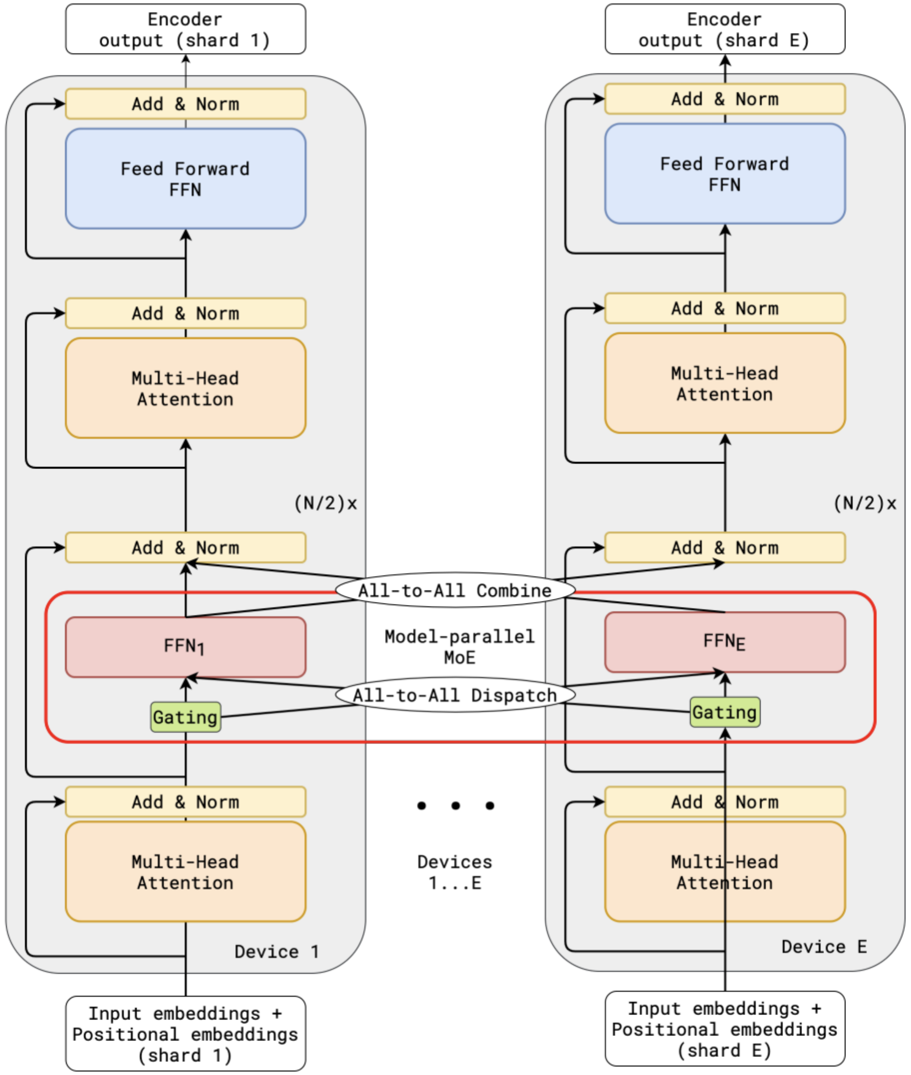
(c) TeraPipe

- Note: this figures shows an all-forward-all-backward mechanism, which eliminates the need for weight stashing but introduces more idle time

- Intuition: increasing pipeline granularity reduces idle time in the pipeline!
    - Efficiency gains scale with sequence lengths and models have been moving towards longer contexts

Expert Parallelism

- Approach: just put each expert on a different GPU!

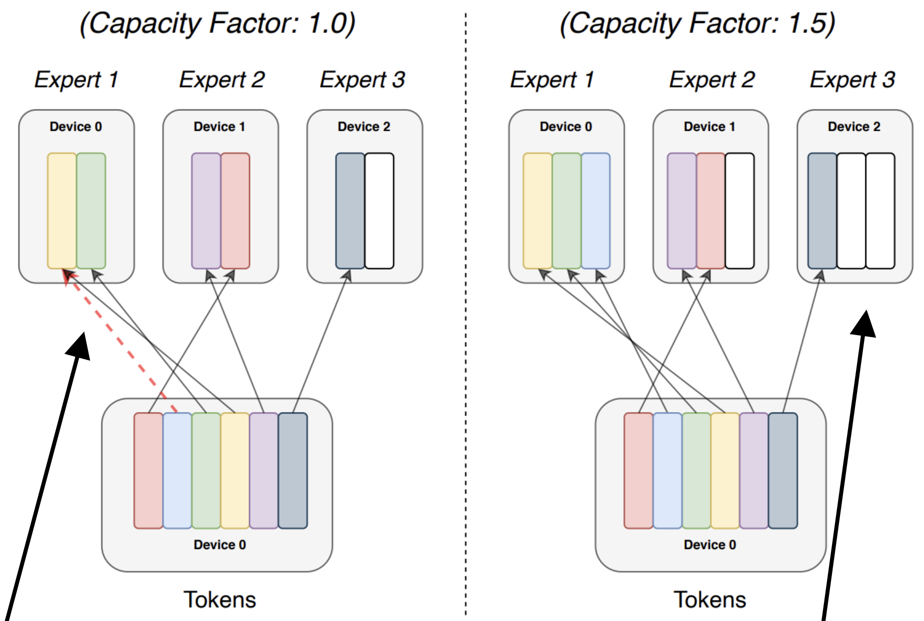


- Key consideration: to enforce a balanced load across GPUs, each expert is assigned a *capacity*
  - If an expert is assigned more tokens than its capacity, those tokens are just passed directly to the next layer via residual connections

Source: https://arxiv.org/pdf/2006.16668

Expert Parallelism

- Approach: just put each expert on a different GPU!



- If the capacity factor is too small, overallocation is likely
- If the capacity factor is too large, some GPUs will likely be underutilized

- Key consideration: to enforce a balanced load across GPUs, each expert is assigned a *capacity*
  - If an expert is assigned more tokens than its capacity, those tokens are just passed directly to the next layer via residual connections

Source: https://arxiv.org/pdf/2101.03961