Agenda:

- Limitations of Data Parallel and ZeRO
- model parallel
  - Tensor model parallel
  - split at layer boundaries
  - Pipeline model Parallel

* ISSUE WITH DP: MEMORY

· each GPU saves a replica of the entire model

· can't train large models that exceed GPU memory! (including BOTH model weights + saved activations)

size of hidden layers

	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

doesn't fit on A100 or V100

· NVIDIA V100: 16GB or 32 GB

· NVIDIA A100: 40GB or 80GB

★ ZeRO (on which FSDP is based): zero-redundancy optimizer

- where is redundancy coming from in model training?

- review: SGD / Adam

SGD:

for t = 1 .. T:

learning rate backward pass forward pass

$$\Delta w = \eta \times \frac{1}{B} \sum_{i=1}^{B} \nabla(loss(f_w(x_i, y_i)))$$

w -= Δw // apply update

ADAM:

for t = 1 .. T:

$$grad = \frac{1}{B} \sum_{i=1}^{B} \nabla(loss(f_w(x_i, y_i)))$$

Δw = ADAM (grad) →

w -= Δw // apply update

Adam might involve:

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

g_t : *Gradient at time t along* ω^j
ν_t : *Exponential Average of gradients along* ω_j
s_t : *Exponential Average of squares of gradients along* ω_j
β_1, β_2 : *Hyperparameters*

optimizer needs to store:

- params themselves (to make final update)
- momentum (ν_t)
- variance (s_t)

- * where does REDUNDANCY come from?

- in DP - each worker stores a copy of:

→ optimizer states (even though... model updates end up being same)

→ gradients (which are the same after all-reduce)

→ params

could be 2 for fp16, 4 for fp32

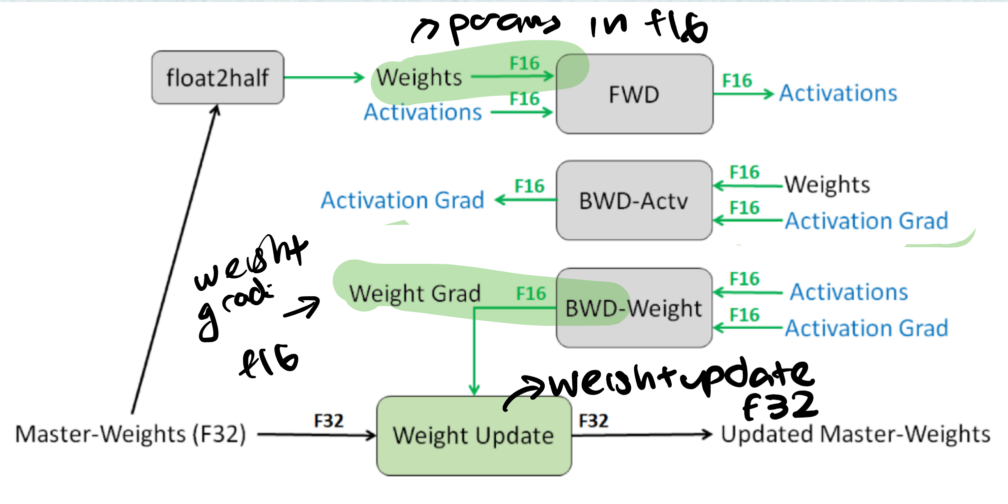
→ params = $P \cdot \text{sizeof}(\text{precision}_{\text{params}})$

→ gradients = $P \cdot \text{sizeof}(\text{precision}_{\text{gradient}})$

→ OS state = $K \cdot P \cdot \text{sizeof}(\text{precision}_{\text{optimizer}})$

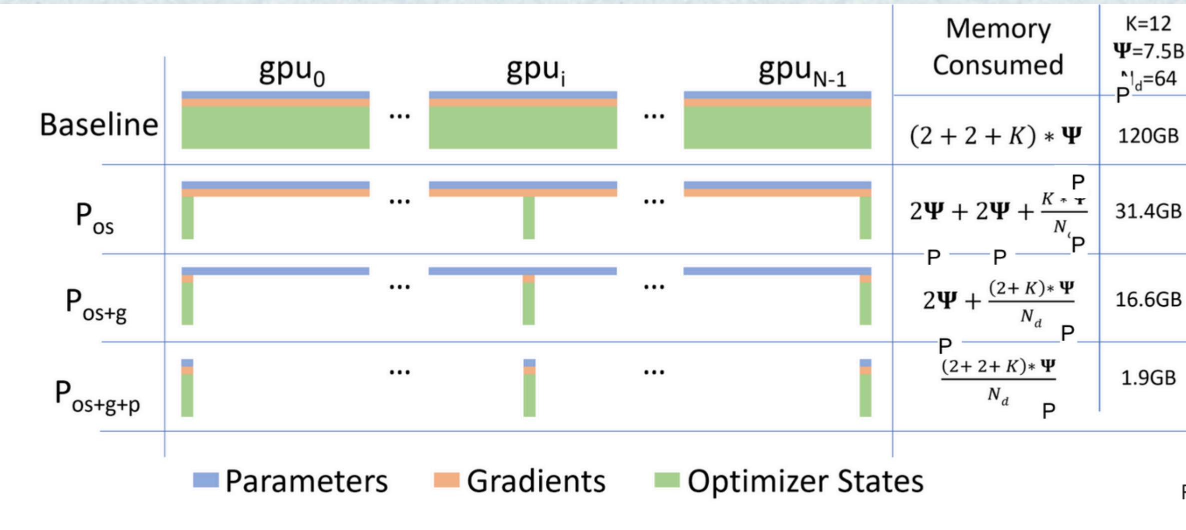
↳ # of variables optimizer class holds PER param (1 for SGD, 3 for Adam)

* Example of MIXED-PRECISION TRAINING



Mixed precision training

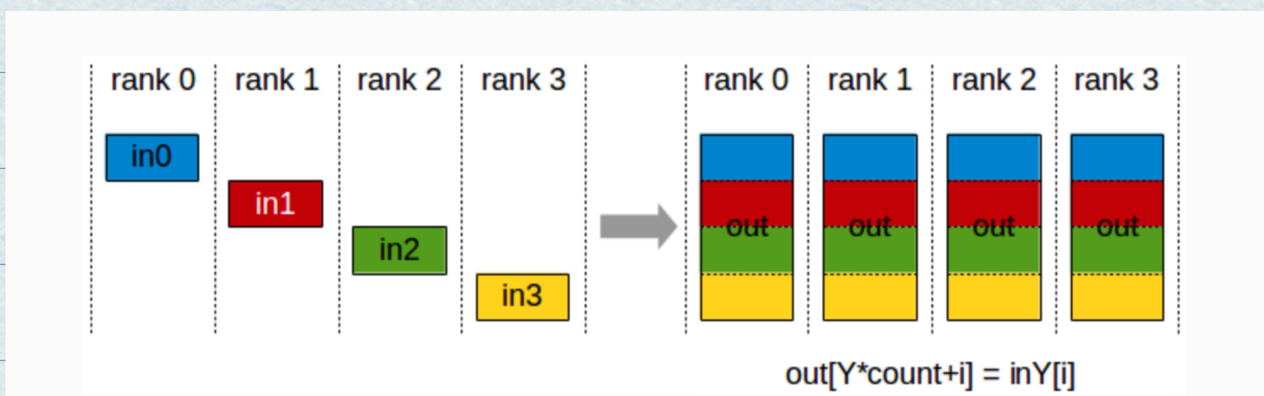
* Zero: shard all the things!



overhead per-GPU for MP training of model w/ 7.5 Billion params 64 workers, Adam optimizer

* ZeRO STAGE 1: OPTIMIZER-STATE PARTITIONING

- Group optimizer into Nd equal partitions
- i_{th} worker only updates state for i_{th} partition
- How does each worker actually get the updated param from optimizer?
  - use ALL-GATHER operation:
    - gathers $\left(\frac{\text{optstate}}{Nd}\right)$ data from each worker
    - distributes result to all workers



AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

- when local process is DONE w/ all-gather, can update local params and free

everything except its partition

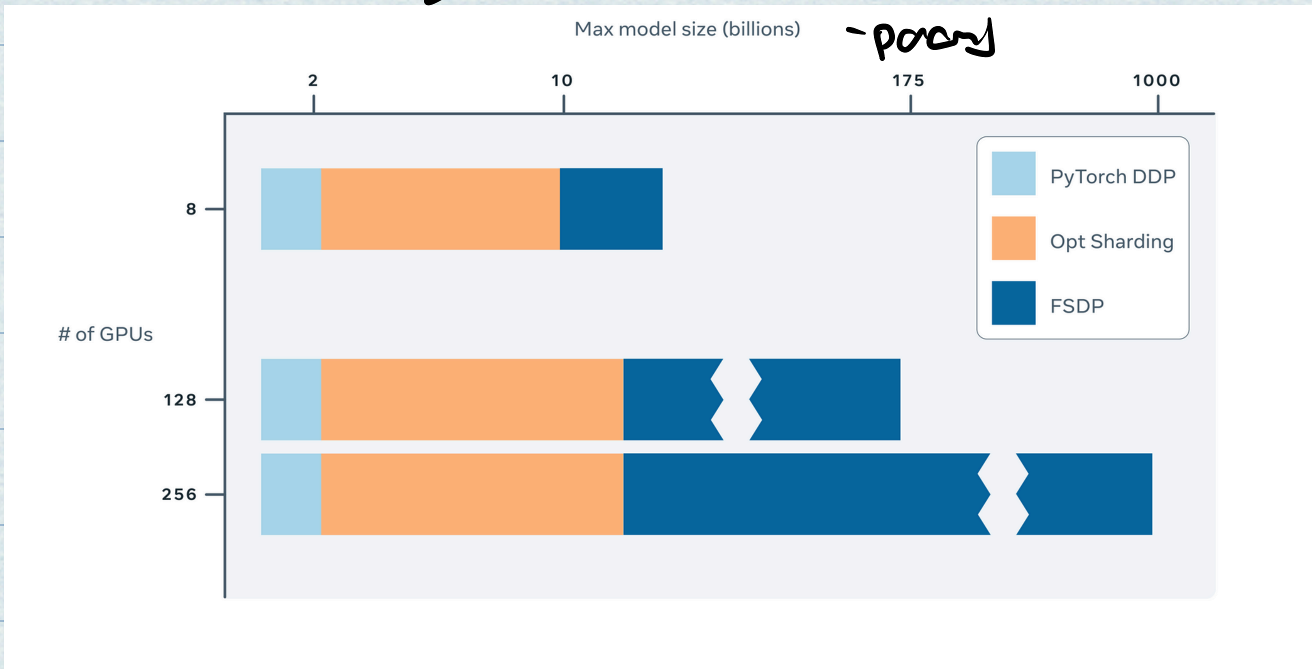
* ZeRO stage 2: Partition Gradients Also

- Because local optimizers are only responsible for updating a portion of params, don't need to keep all gradients on all workers - need to only store gradients for its partition

* ZeRO stage 3: Add partitioning of PARAMS

  - instead of storing all params on all layers - fetch params as needed for FW and BW pass
  - increases comm. to 1.5x compared to DP (see zero-paper), but reduces memory proportionally by # of workers

meta's experiences w/ FSDP (pytorch implementation of zero)



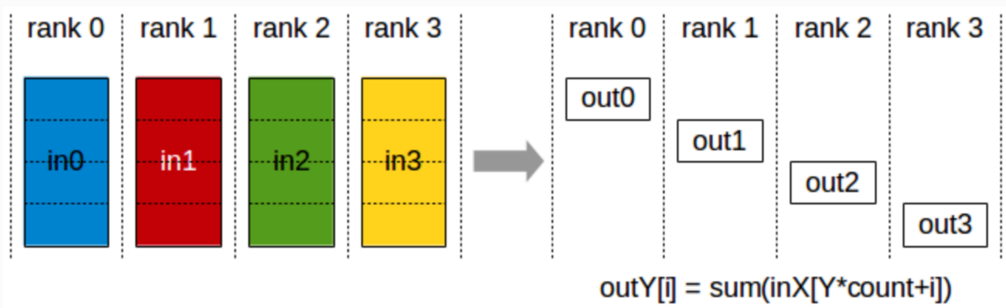
- poorly

* Intuition for gradient sharding
  - q: how do we do grad update if grad is distributed?
  - combine REDUCE-SCATTER w/ All-gather:

ReduceScatter

The ReduceScatter operation performs the same operation as Reduce, except that the result is scattered in equal-sized blocks between ranks, each rank getting a chunk of data based on its rank index.

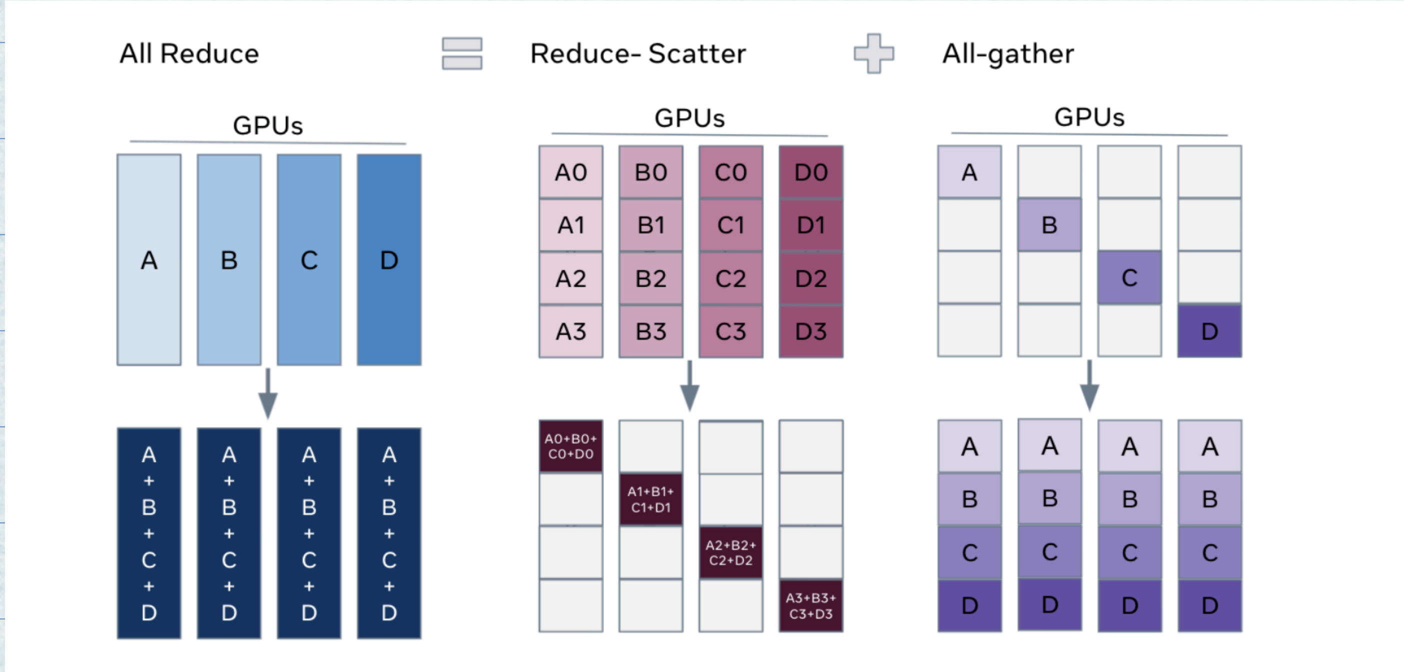
The ReduceScatter operation is impacted by a different rank to device mapping since the ranks determine the data layout.



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.

- shard gradient updates per param into 4 (0,1,2,3,4)

* TODO: understand sharding a bit better



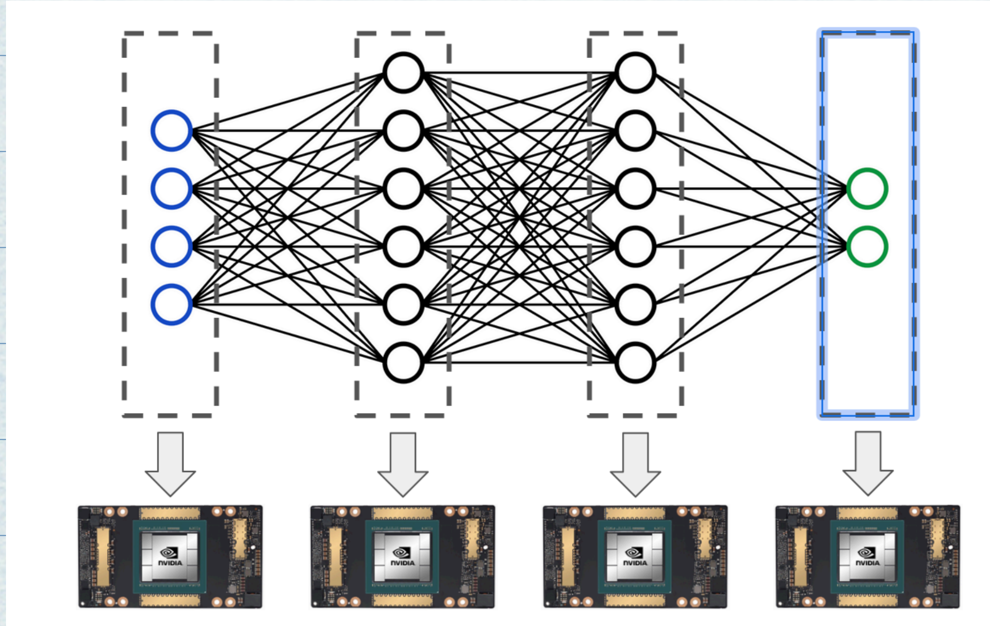
All-reduce as a combination of reduce-scatter and all-gather. The standard all-reduce operation to aggregate gradients can be decomposed into two separate phases: reduce-scatter and all-gather. During the reduce-scatter phase, the gradients are summed in equal blocks among ranks on each GPU based on their rank index. During the all-gather phase, the sharded portion of aggregated gradients available on each GPU are made available to all GPUs (see here for details on those operators).

Part 2: Model Parallelism

- in DP we replicated model, but sharded data
- in MP: replicate data but partition model

- Two general approaches:
  1) slice model "vertically": place dif. subsets of dif. layers on each worker
  2) slice model horizontally: shard model weights across dif. GPUs

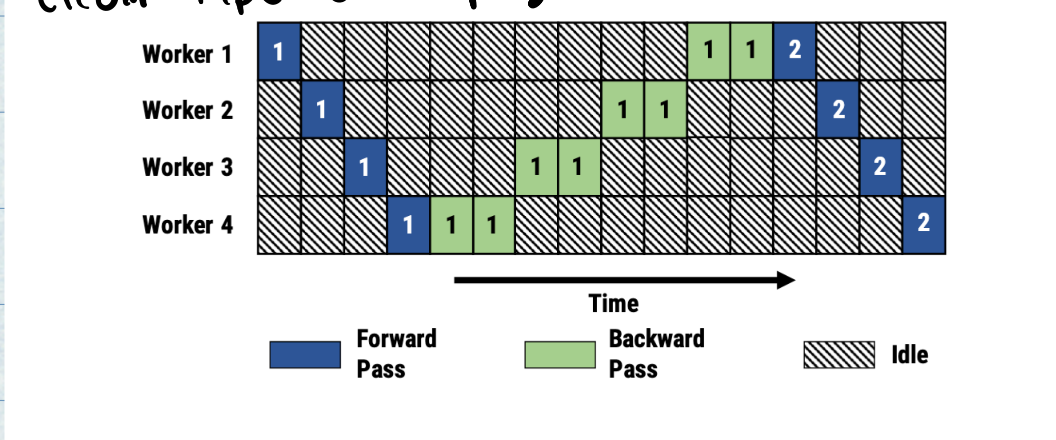
- * vertical slicing MP:



leads to:

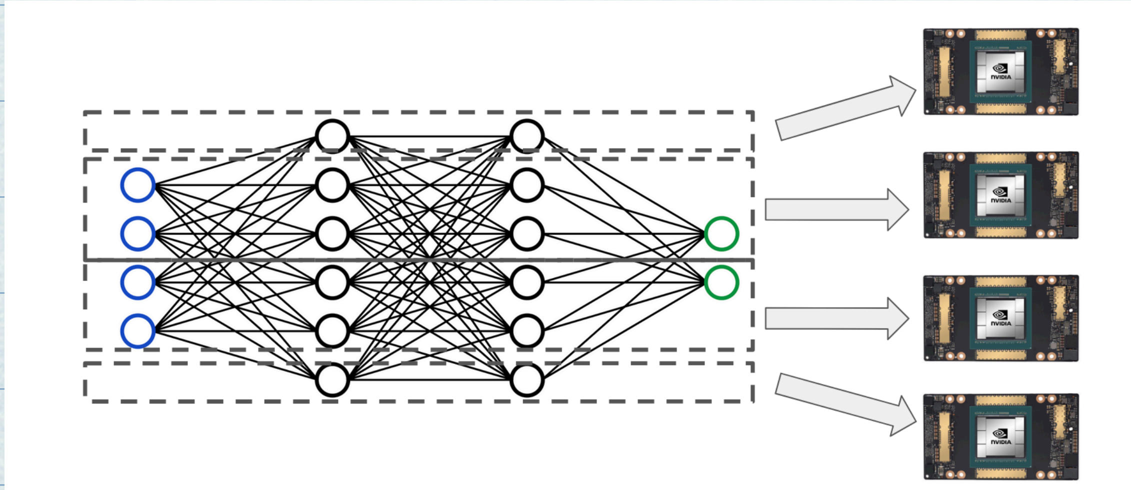
- under-utilization of compute resources (only 1 worker being used at a time)
- low overall throughput due to low utilization

(from Pipedream paper)

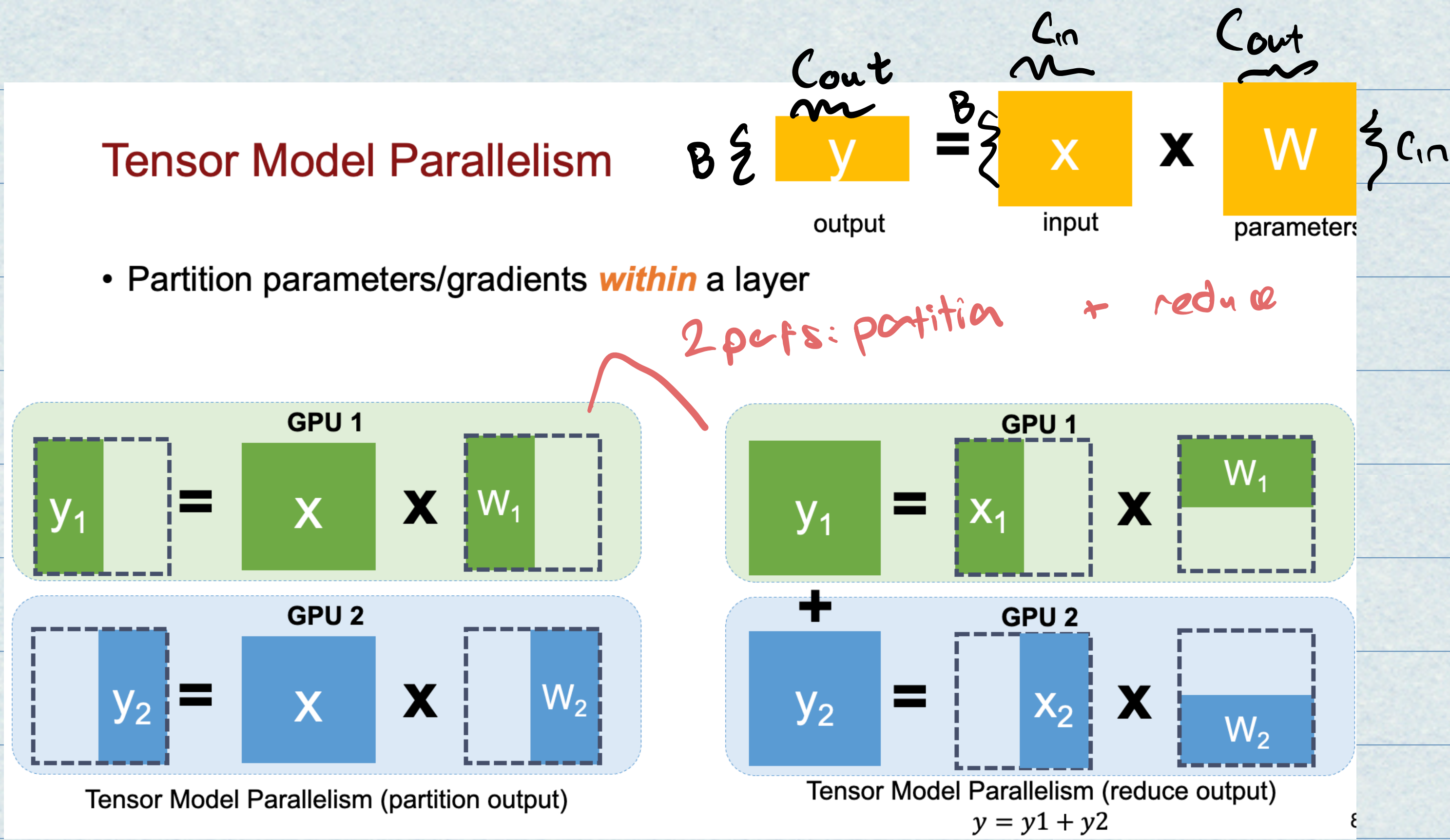


P1 FW → P2 FW → P3 FW → P4 FW → loss → P4 BW → P3 BW → P2 BW → P1 BW

• Tensor-model parallelism: slice layers horizontally



- how does this work?



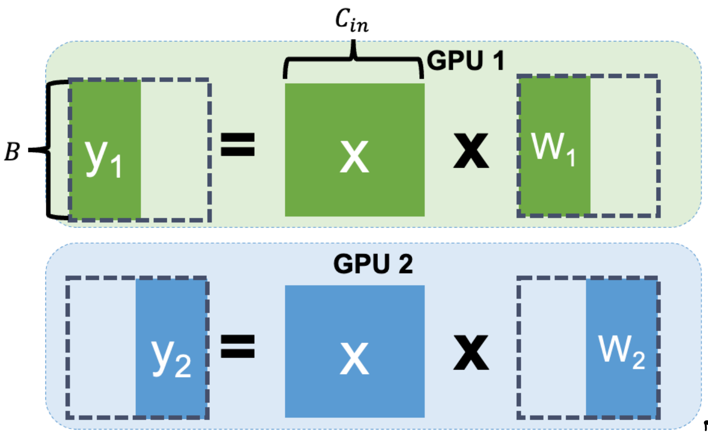
*what is the comm. overhead of tensor model parallelism?

- lets start by looking at data:

	FW processing	BW processing	Gradient sync
DATA PARALLEL	O	O	$O(C_{out} \times C_{in})$
Tensor model parallelism	partition: $O(B \times C_{in})$ reduce: $O(B \times C_{out})$	partition: $O(B \times C_{out})$ reduce: $O(B \times C_{out})$	O

↳ TMP require sync after each FW/BW pass to collect output. But no syncing of gradient

SHARD W BY COLUMN (duplicate x):



Tensor Model Parallelism (partition output)

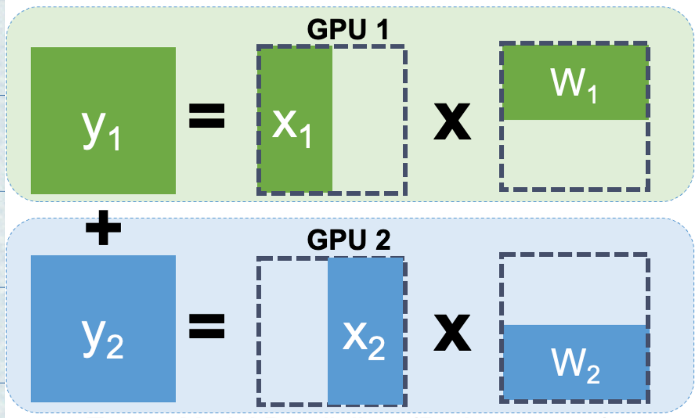
option 1: replicate X and split W by column

Forward Processing	Backward Propagation	Gradients Sync
$O(B * C_{in})$	$O(B * C_{in})$	0

Communication Cost of Tensor Model Parallelism

$y = (y_1, y_2)$

SHARD W BY ROW (split X by column to work):



Tensor Model Parallelism (Reduce output)

$y = y1 + y2$

option 2: split X by col and W by row

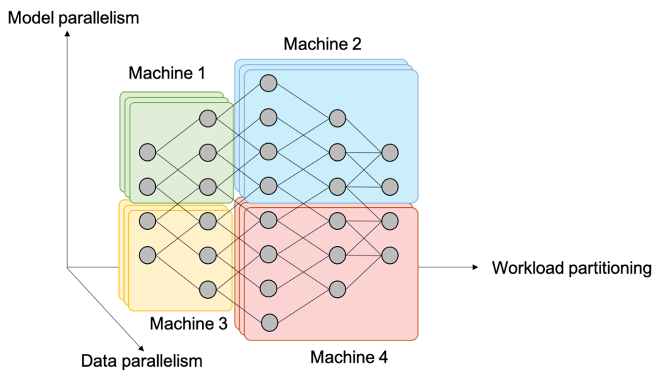
Forward Processing	Backward Propagation	Gradients Sync
$O(B * C_{out})$	$O(B * C_{out})$	0

Communication Cost of Tensor Model Parallelism

11

→ if there's a non-linear op after, need to gather outputs

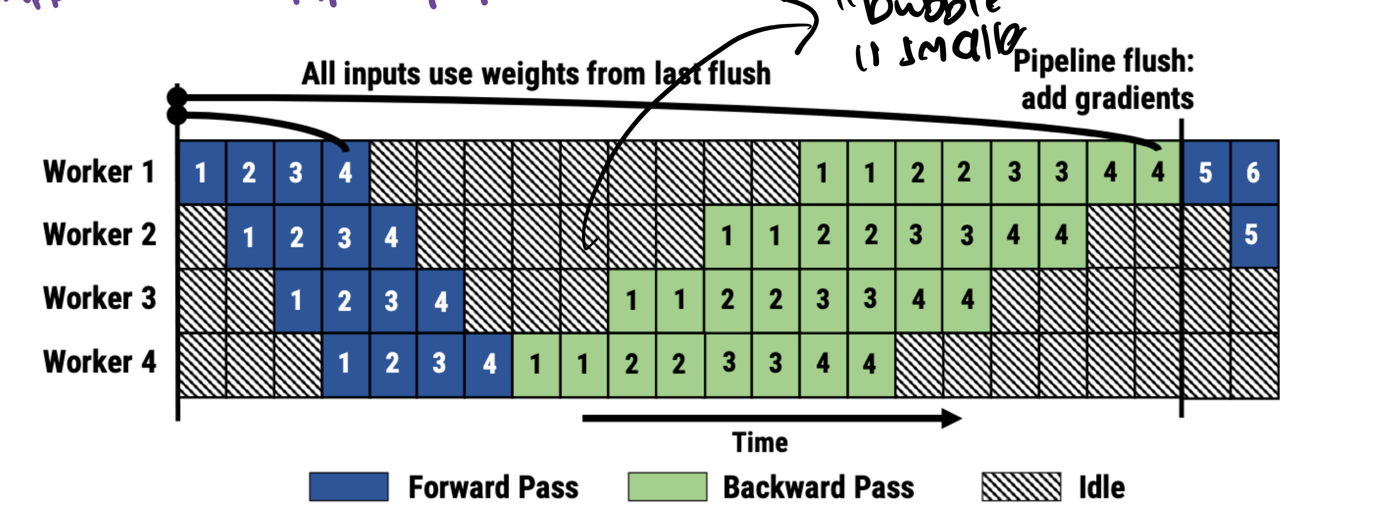
- could even combine model and data parallel:



- what are pros and cons of TMP?
  - Pro: reduces mem. per GPU
  - Pro: GPU util. high compared to vertical slicing
  - Con: FREQUENT All-reduces (partition AND reduce-output step) - need high BW network connection for this to work
  - Con: not too easy to implement

- Let's revisit VERTICAL SLICING from earlier:
  - idea: divide each batch into multiple microbatches
  - pipeline FW and BW computation across minibatches

Approach in GPipe paper



- Q: how could we make PP more efficient?
  - Play w/ m (microbatch size).
    - Increase batch size or decrease

microbatch size

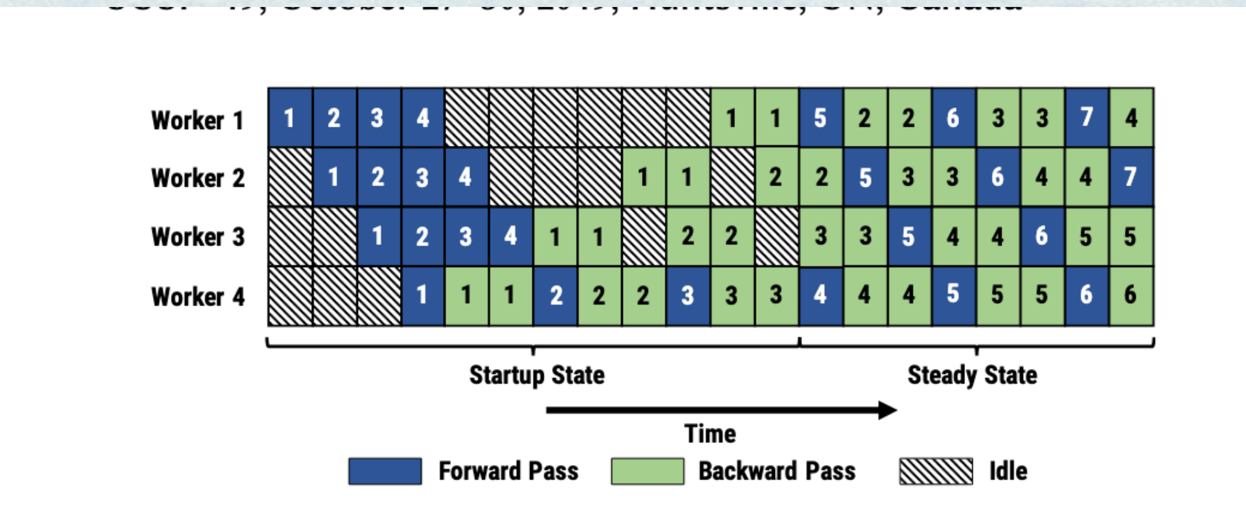
- caveat: larger minibatch sizes leads to accuracy loss

- smaller microbatch sizes reduces utilization

- Play w/ p (number of stages):
  - Decrease pipeline size
  - caveat: memory limits

- Another Issue: ACTIVE memory requirement
  - for EACH microbatch - need to keep output of intermediate layers - for later BW pass

- Idea: solve this w/ "1F1B" schedule:



- this way, each worker can delete

state for each microbatch after

BW pass

... see paper for more variations!

SUMMARY: when are dif. parallelism approaches effective?

- Data: if model weights and activations fit into GPU memory
- Tensor MP: model weights / activation DO NOT fit in GPU mem, but we have fast networking (in a single box)
- pipeline MP: effective if model weights / activations DON'T fit into GPU mem, but we have slow networking
    - either multiple machines
    - OR 1 box w/ slow interconnect.

* next time: intro to transformers!
  - we will revisit tensor model parallelism

References:

Model size table: CMU 15-442 Lecture on ML Parallelization Part 1 led by Zhihao Jia and Tianqi Chen
Adam optimizer method screenshot: Kingma and Ba, "Adam: A Method for Stochastic Optimization", 2014, https://arxiv.org/abs/1412.6980.
Mixed precision training: Mixed Precision Training. Narang, et al. https://arxiv.org/pdf/1710.03740.pdf
General Information about ZeRO: ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. Rajbhandari et al. https://arxiv.org/pdf/1910.02054.pdf
ZeRO diagram: From ZeRO paper (above), taken from Azalia's slides
All-gather image: NCCL documentation
Meta FSDP Article: https://engineering.fb.com/2021/07/15/open-source/fsdp/
Reduce-scatter image: NCCL documentation
All-reduce = reduce-scatter + All-gather: Meta Article
Model-Parallelism Diagram: Azalia's slides
Tensor-Parallelism Diagram: Azalia's slides
Breakdown of tensor-model parallelism of partition and reduce: CMU 15-442 Lecture on ML Parallelism Part 2
Combined Data and Model Parallel: CMU 15-442 Lecture on ML Parallelism Part 2
Vertical Model Parallel Utilization Diagram: Pipedream paper
Gpipe Utilization diagram: Pipedream paper
Pipedream 1F1B diagram: Pipedream paper
Megatron LM: https://arxiv.org/pdf/2104.04473

General Lecture Flow:
CMU 15-442 Lecture on ML Parallelization Part 1
CS 229s 2023, Lecture on Parallelism Fundamentals, Given By Azalia Mirhoseini