Parallelizing LLM Training

Deepak Narayanan

CSE 599K

LLM training is a huge computational challenge!

Cloud compute cost to train frontier AI models over time

Cost (2023 USD, log scale)

Regression mean
95% CI of mean

1B
100M
10M
1M
100k
10k
1000
100
10

2016
2017
2018
2019
2020
2021
2022
2023
2024

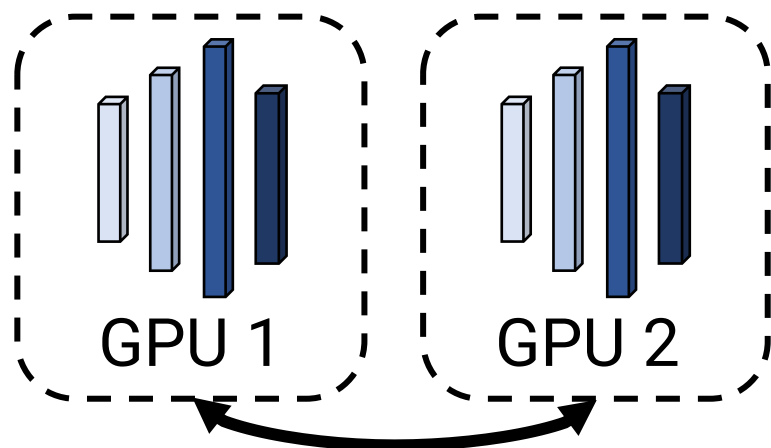
Publication date

Gemini 1.0 Ultra
GPT-4
PaLM (540B)
GPT-3 175B (davinci)
Inflection-2
GNMT
AlphaZero
DALL-E
2.6x/year

https://epochai.org/blog/how-much-does-it-cost-to-train-frontier-ai-models

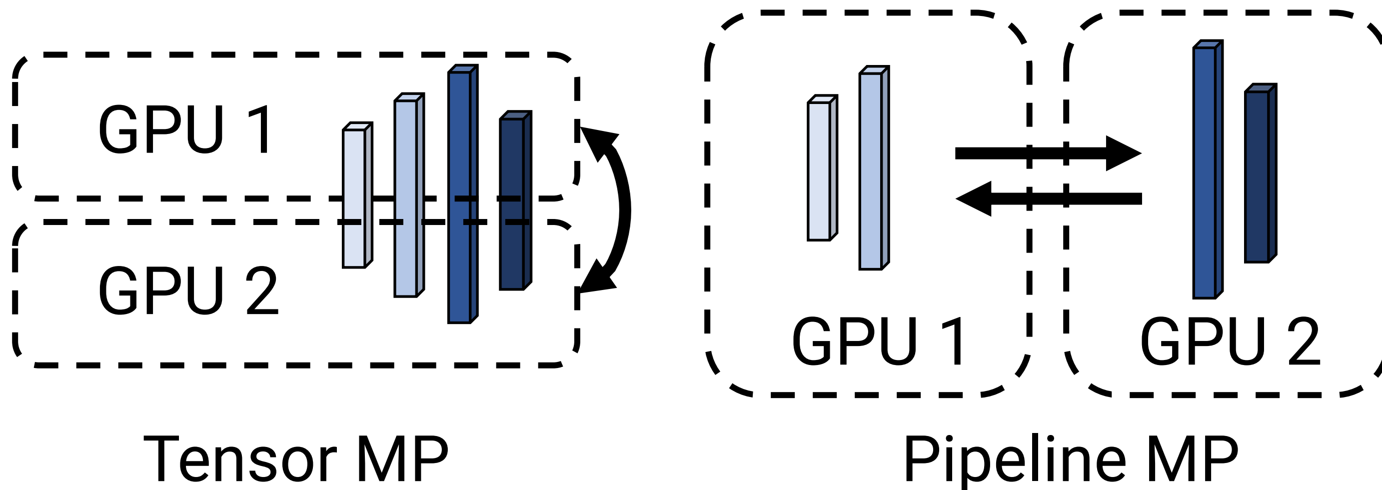
Parallel model training: an overview

Data parallelism (DP)



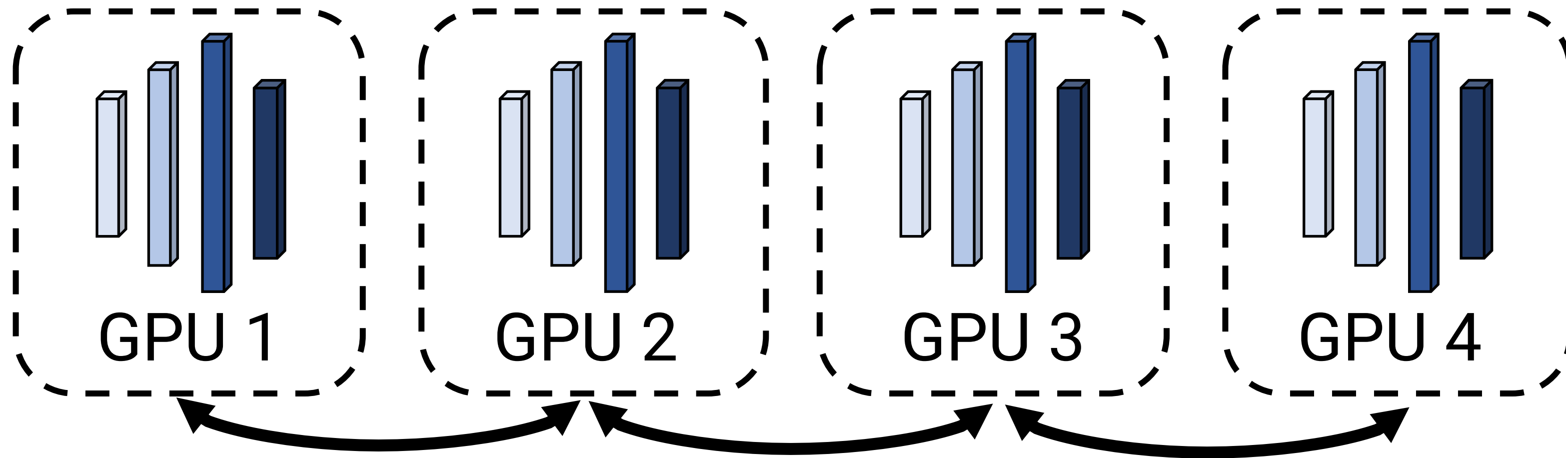
n copies of model parameters

Model parallelism (MP)



Single copy of model parameters

Data parallelism



- Naïvely, model copy on each GPU
- Reductions of weight gradients at the end of every iteration to coalesce updates across replicas
- Typically can be implemented by a wrapper class around model

Distributed optimizer to reduce memory

Number of bytes of state per parameter $= \underbrace{2 + 4}_{\text{bf16 params, fp32 grads}} + \underbrace{4 + 4 + 4}_{\text{fp32 copy of params, fp32 Adam states}}$

`bf16` params
`fp32` grads

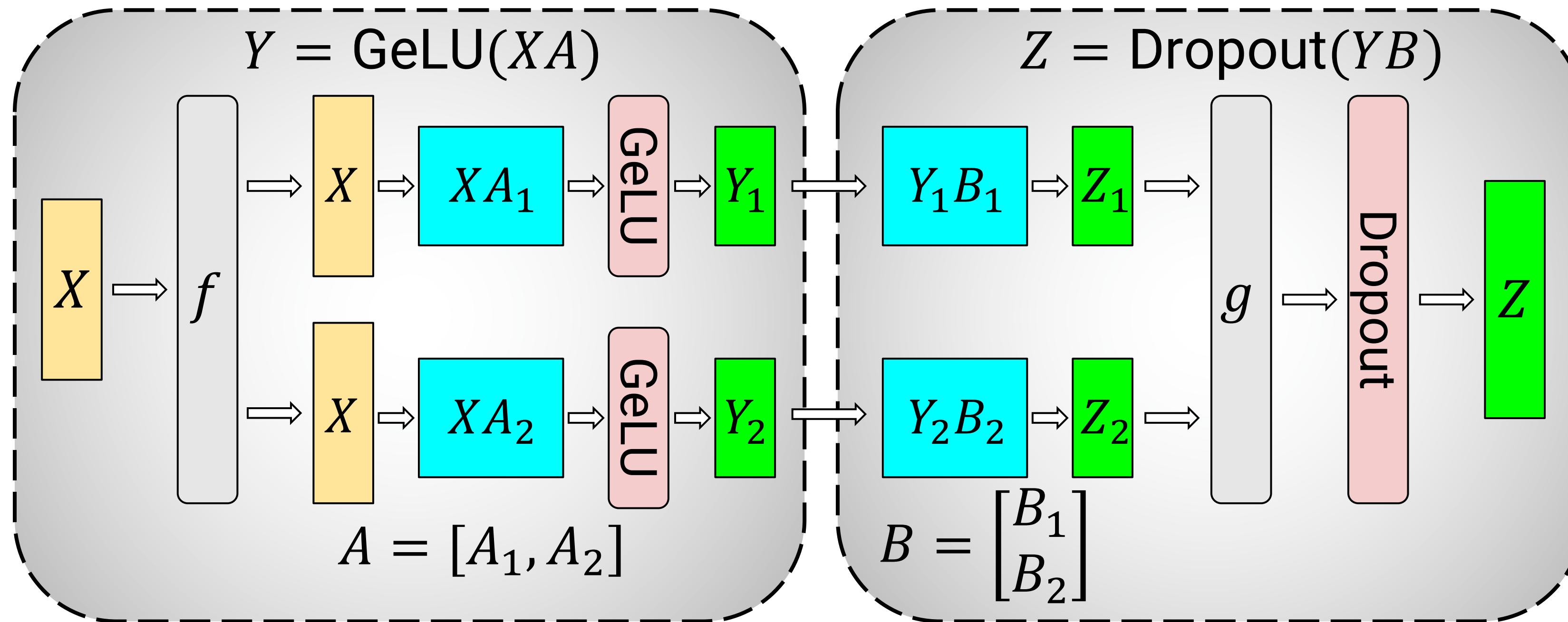
`fp32` copy of params
`fp32` Adam states

Number of bytes of state for Nemotron-4 340B model $= 18 \cdot 340\text{B} =$ **6120 GB**

Redundant optimizer state over DP replicas can be partitioned
`fp32` gradient all-reduces →
`fp32` gradient reduce-scatters + `bf16` param all-gathers

Tensor model parallelism

Each layer of model is partitioned over multiple GPUs



$g \rightarrow$ All-reduction $(Y_1B_1 + Y_2B_2)$ in forward pass

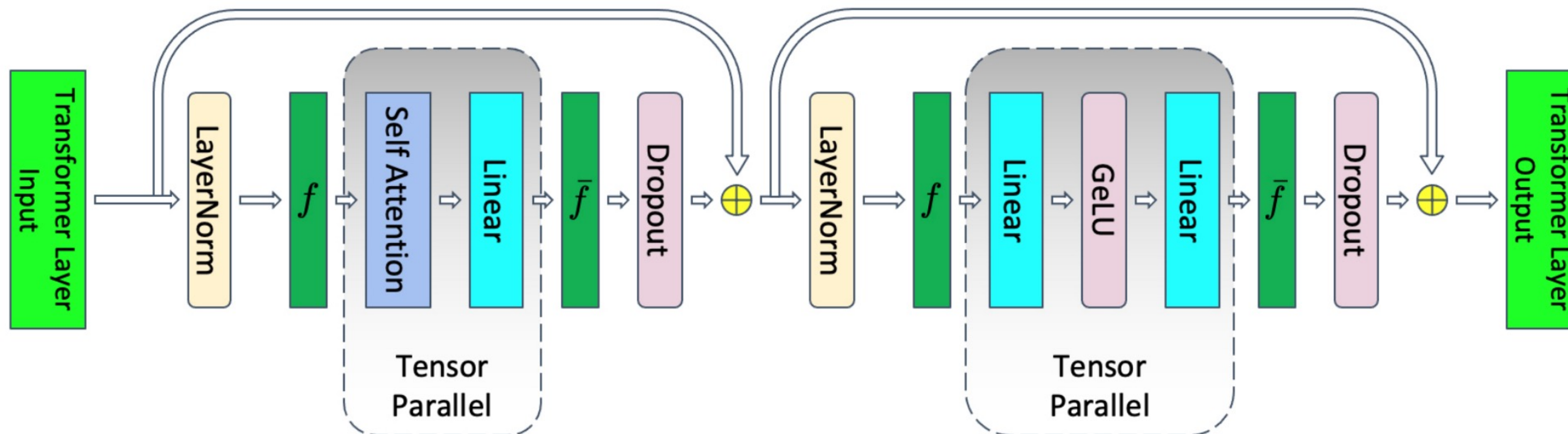
Slow across inter-server communication links

Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, arXiv:1909.08053, Shoeybi et al.

Tensor model parallelism implementation

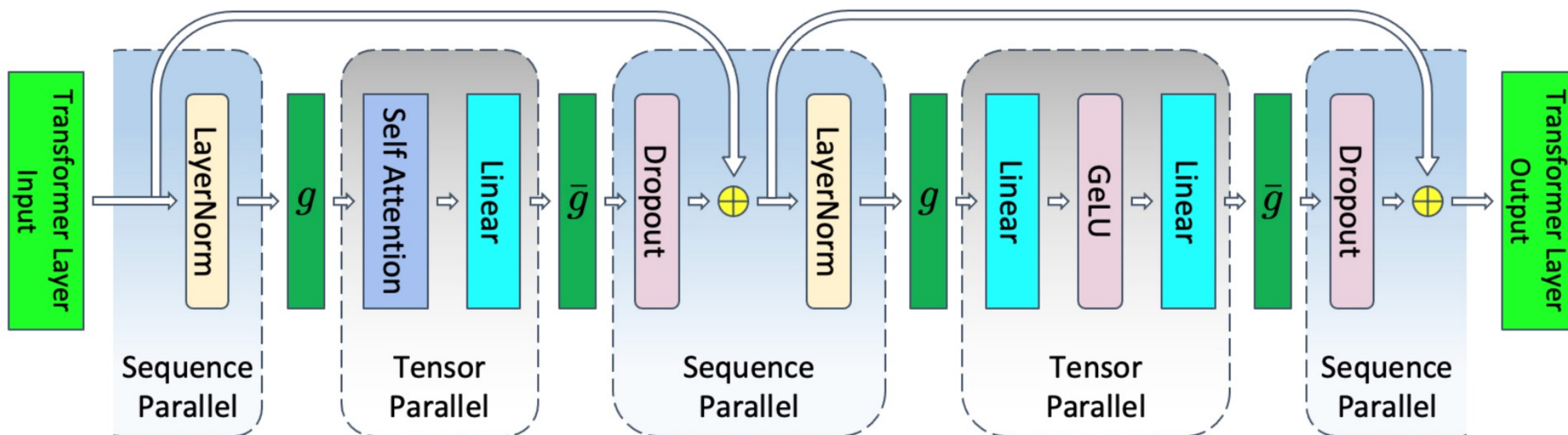
- Core building blocks are RowParallelLinear and ColumnParallelLinear
  - Simple underlying implementation: [Optional communication] → Linear operator → [Optional communication]
- Modules in models use these primitives under the hood as needed
  - E.g., Previous MLP would use ColumnParallelLinear layer followed by RowParallelLinear layer

Activation footprint with naïve tensor parallelism



Activation memory footprint per layer $= bsh(10 + \frac{24}{t} + \ ...)$

Reduce activation footprint further



$$\text{Activation memory footprint per layer} = bsh\left(\frac{10}{t} + \frac{24}{t} + \ \dots\right)$$

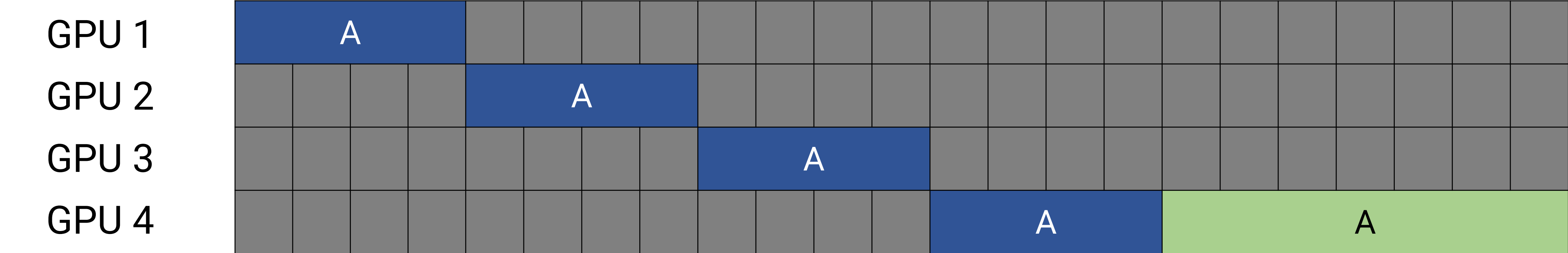
All-reduce → Reduce-scatter + all-gather

Reducing Activation Recomputation in Large Transformer Models, arXiv:2205.05198, Korthikanti et al.

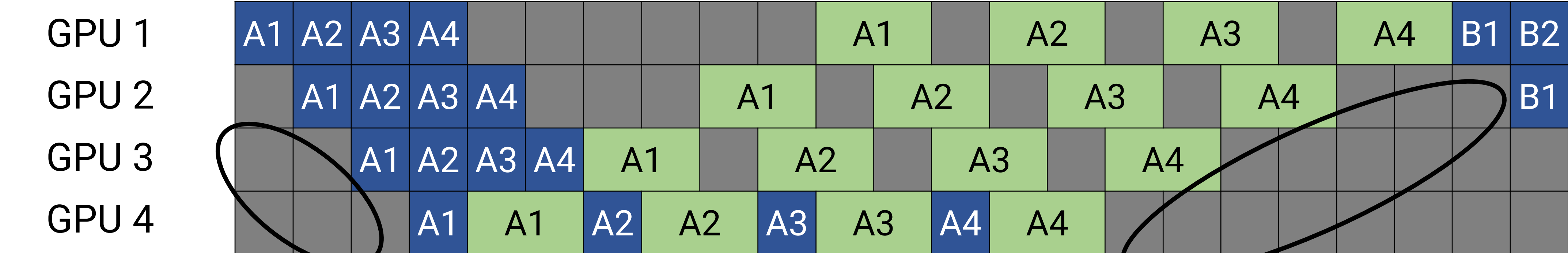
Pipeline model parallelism

- Layers / operators in model sharded over GPUs (i.e., each GPU is responsible for a subset of layers in the model)
- Each batch split into smaller microbatches and execution pipelined across these microbatches

Pipeline model parallelism



Split batch into microbatches and pipeline execution

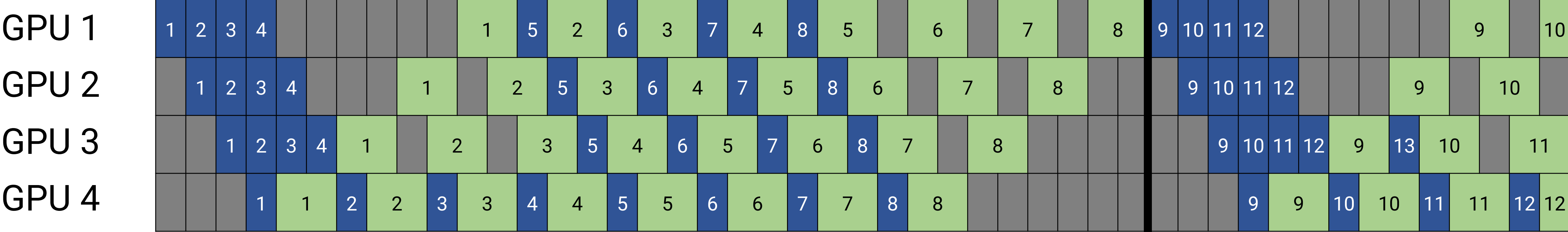


Forward Pass
Backward Pass

Pipeline model parallelism

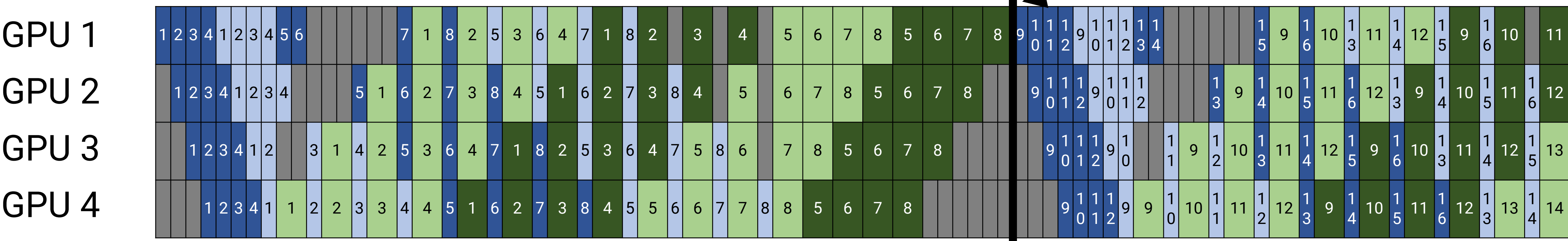
- Layers / operators in model sharded over GPUs (i.e., each GPU is responsible for a subset of layers in the model)
- Each batch split into smaller microbatches and execution pipelined across these microbatches
- Point-to-point communication between consecutive pipeline stages
- Pipeline bubble at the start and end of every batch (equal to $(p-1)$ microbatches' forward and backward passes)

Interleaved pipeline parallelism: fancier assignments of layers to GPUs!



Time →

Assign multiple stages to each device (interleaved schedule)



Time →

Smaller pipeline bubble but more communication

Pipeline model parallelism implementation

- Can abstract away most of the complex scheduling code that coordinates scheduling of forward and backward passes for different microbatches, along with associated communication; doesn’t need to be rewritten for a new model*

- Model should be written in a way that makes it easy to separate into different pipeline stages

Pipeline model parallelism implementation details

From megatron/core/models/gpt/gpt_model.py:

```
if self.pre_process:
    self.embedding = LanguageModelEmbedding(
        config=self.config, vocab_size=self.vocab_size, ...
    )


self.decoder = TransformerBlock(
    config=self.config, spec=transformer_layer_spec, ...)


if post_process:
    self.output_layer = tensor_parallel.ColumnParallelLinear(
        config.hidden_size, self.vocab_size, ...)
```

…how about combining them?

Different schemes have different tradeoffs

- Each of these parallelism dimensions have different limiting factors
  - E.g., pipeline parallelism only scales up to number of model layers
    ⇒ Need to combine parallelisms if scaling to 1000s of GPUs

- Naïvely combining parallelisms leads to poor throughput
  - E.g., tensor-model-parallel communication can dominate

- Using 3D parallelism efficiently (i.e., determining degrees of each parallelism) requires one to reason through these interactions

Tradeoffs between tensor and pipeline MP

Assume that total number of GPUs is n, tensor-model-parallel size is t, pipeline-model-parallel size is p ($t \cdot p = n$)

- Pipeline bubble size (fraction of ideal time spent idle) is then:

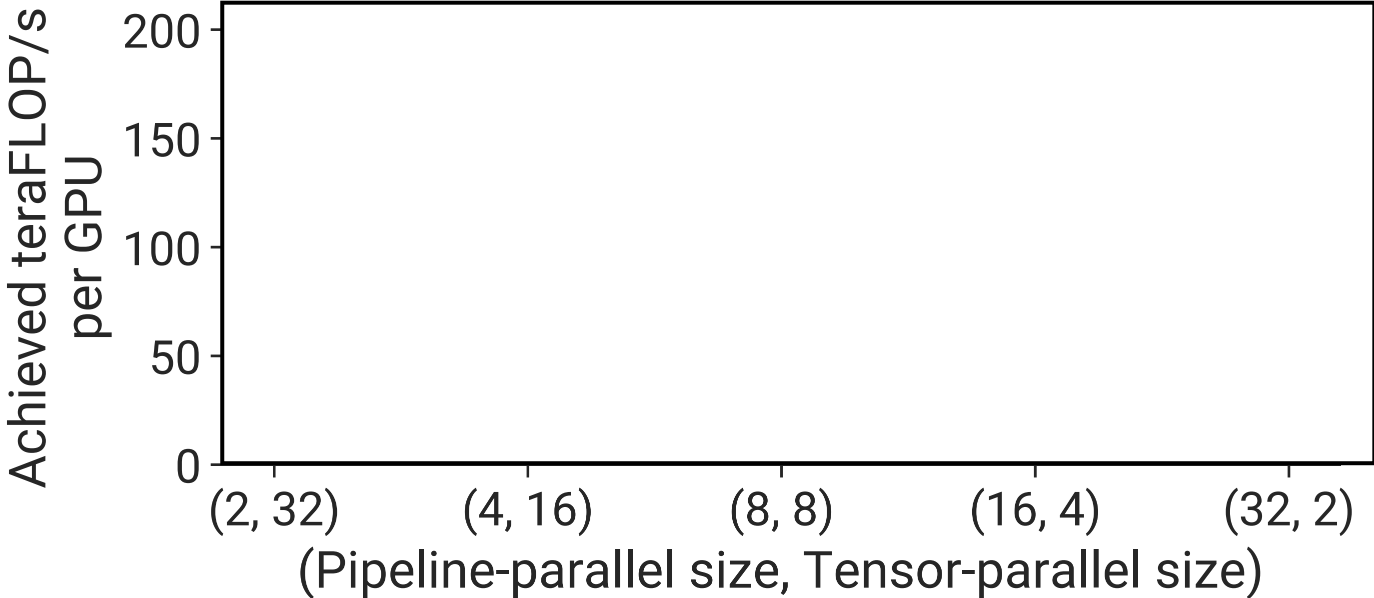
$$\frac{(p-1)\cdot\left(t_f+t_b\right)}{m\cdot\left(t_f+t_b\right)}=\frac{p-1}{m}=\frac{n/t-1}{m}$$

Number of microbatches in batch ⟵ m

As t increases, pipeline bubble size decreases

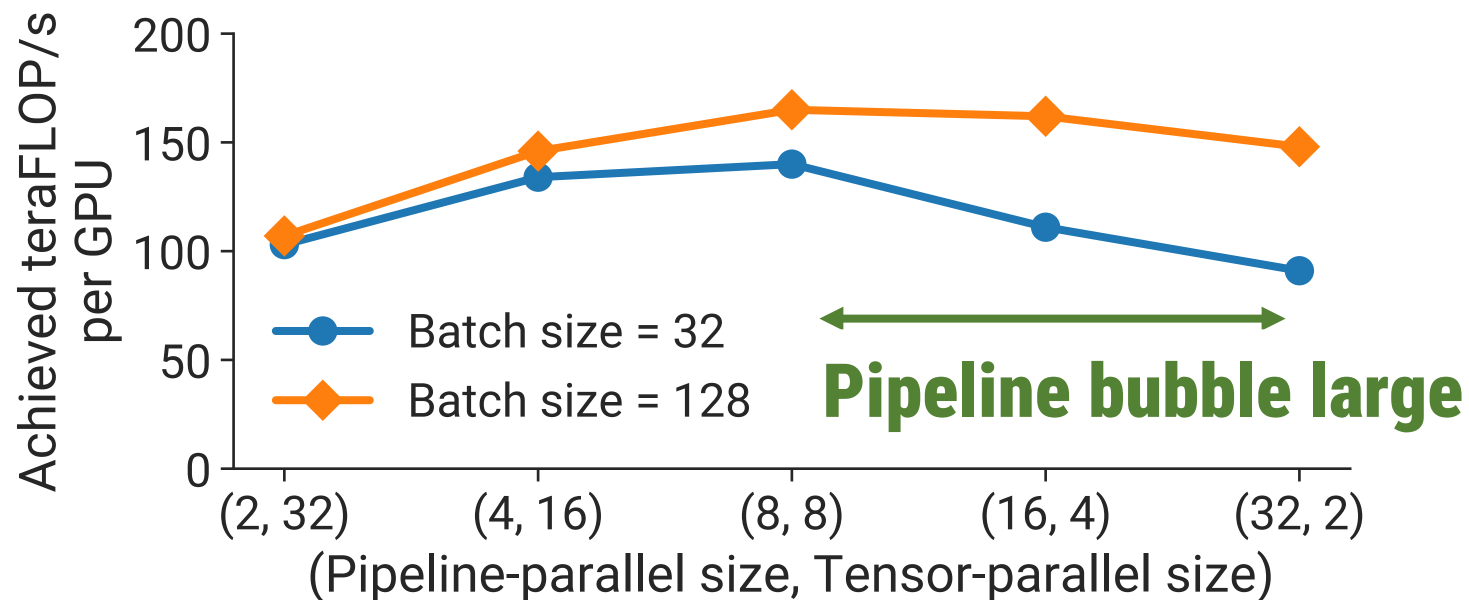
- However, as t increases beyond the number of GPUs in a server, all-reduce communication is now cross-server and more expensive

Tradeoffs between tensor and pipeline MP



Tradeoffs between tensor and pipeline MP

162B GPT model
64 80-GB A100 GPUs



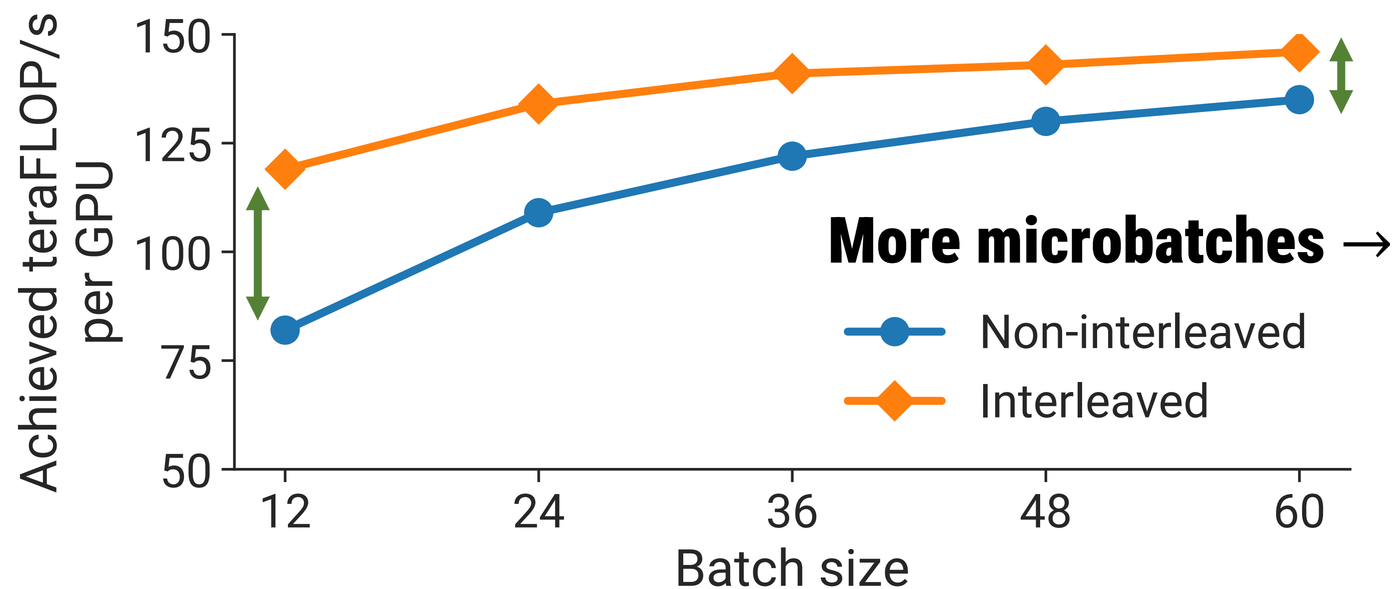
All-reductions for tensor MP across servers

How do we navigate this configuration space?

Degree of pipeline, tensor, and data parallelism

Pipelining schedule

New pipeline schedules affect throughput too!



Large throughput increases at small batch sizes, smaller at large batch sizes

How do we navigate this configuration space?

Degree of pipeline, tensor, and data parallelism

Pipelining schedule

Global batch size

Microbatch size

Each of these influence amount of communication, size of pipeline bubble, memory footprint, convergence rate

More details in our paper: https://arxiv.org/pdf/2104.04473.pdf

General rule of thumb: don’t over-parallelize!

- Use just data parallelism + distributed optimizer if possible
- If this OOMs, use tensor model parallelism + sequence parallelism
- If this OOMs, then add pipeline parallelism into the mix

Picking right configuration parameters necessary but not sufficient!

SOTA models no longer just have more parameters!

Training Compute-Optimal Large Language Models

Jordan Hoffmann*, Sebastian Borgeaud*, Arthur Mensch*, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals and Laurent Sifre*

*Equal contributions

We investigate the optimal model size and number of tokens for training a transformer language model under a given compute budget. We find that current large language models are significantly undertrained, a consequence of the recent focus on scaling language models whilst keeping the amount of training data constant. By training over 400 language models ranging from 70 million to over 16 billion parameters on 5 to 500 billion tokens, we find that for compute-optimal training, the model size and the number of training tokens should be scaled equally: for every doubling of model size the number of training tokens should also be doubled. We test this hypothesis by training a predicted compute-optimal model, *Chinchilla*, that uses the same compute budget as *Gopher* but with 70B parameters and 4× more more data. *Chinchilla* uniformly and significantly outperforms *Gopher* (280B), GPT-3 (175B), Jurassic-1 (178B), and Megatron-Turing NLG (530B) on a large range of downstream evaluation tasks. This also means that *Chinchilla* uses substantially less compute for fine-tuning and inference, greatly facilitating downstream usage. As a highlight, *Chinchilla* reaches a state-of-the-art average accuracy of 67.5% on the MMLU benchmark, greater than a 7% improvement over *Gopher*.

Compute-optimal models: systems implications

- Models staying roughly the same size but the number of training tokens is drastically increasing

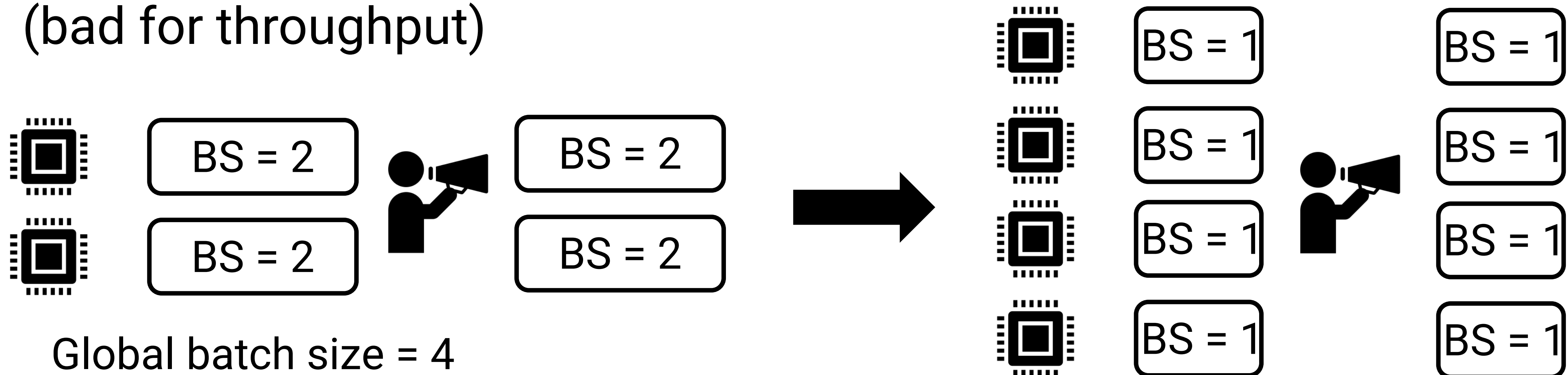
Need to scale out: use more GPUs to get results faster (as opposed to using more GPUs to just fit model parameters in GPU memory)

- ⇒ We need to train models efficiently in regimes where the batch size per GPU is much smaller (e.g., batch size / # GPUs = ¼)

What happens when batch size per GPU is small?

- Number of microbatches in a pipeline is small → pipeline bubble ($=(p-1)/m$) is large (bad for throughput)
  - Use interleaved schedule which makes pipeline bubble smaller but also increases communication?

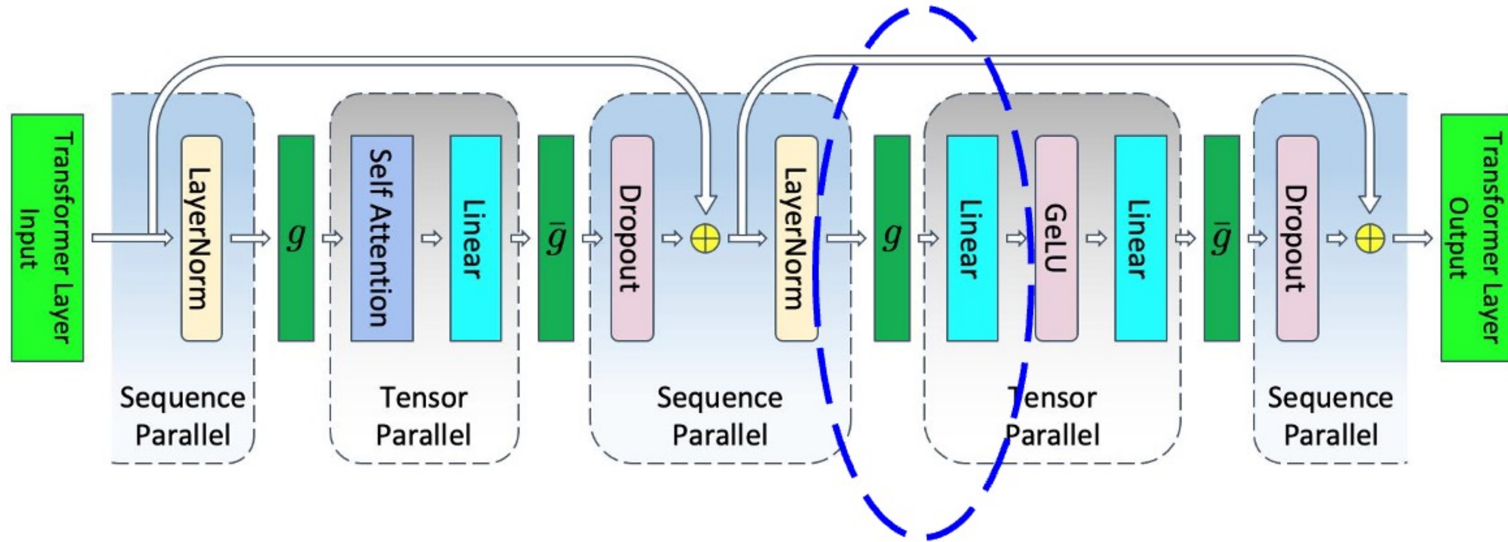
- Need to perform data-parallelism communication more frequently (bad for throughput)

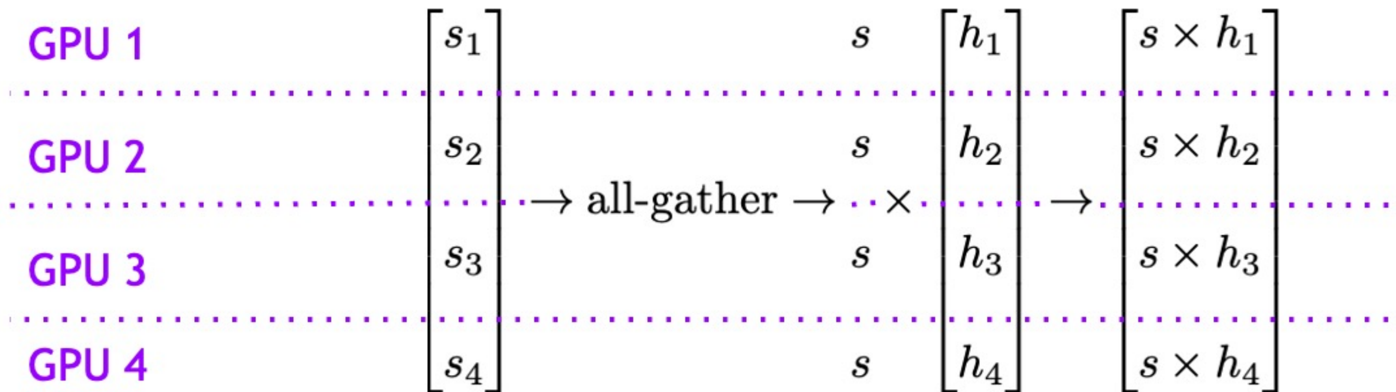


How can we mitigate communication overhead?

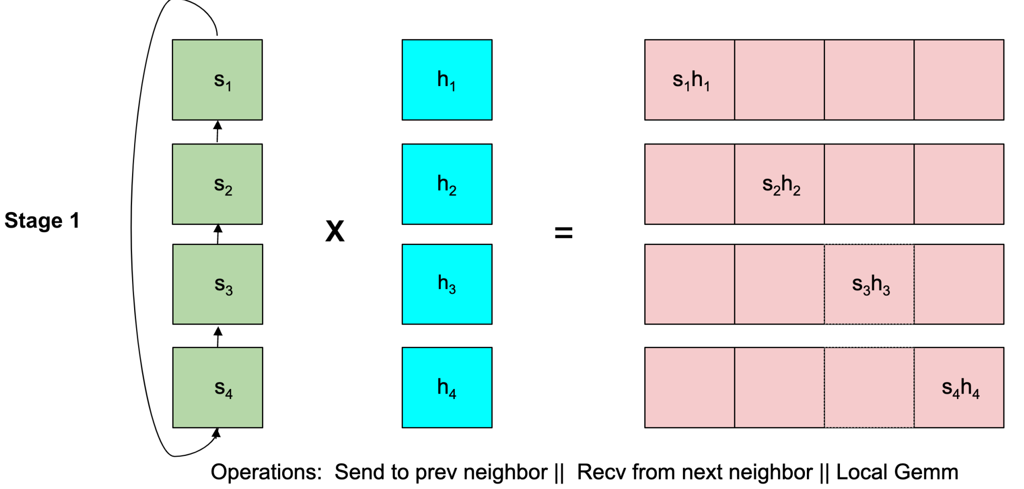
Is it possible to hide communication behind compute?

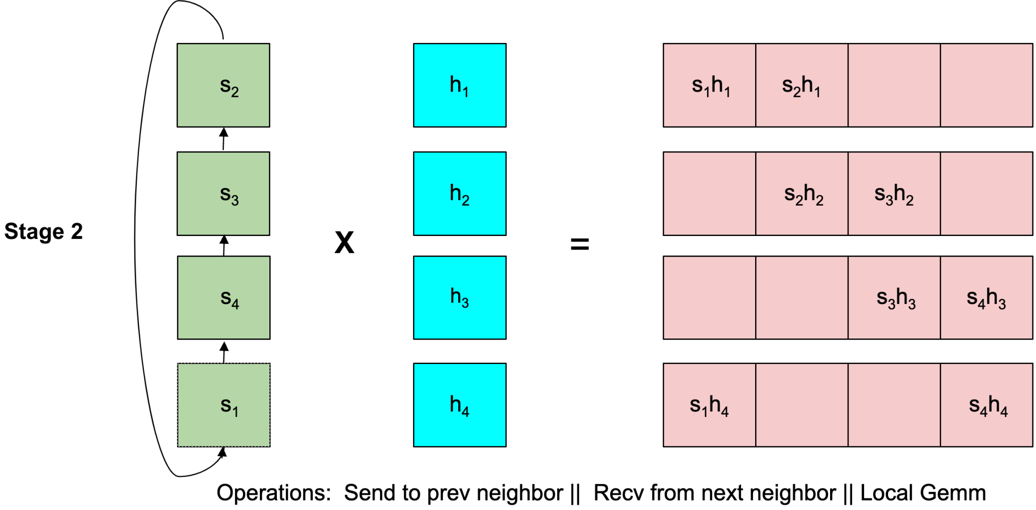
Communication overlap: tensor parallelism

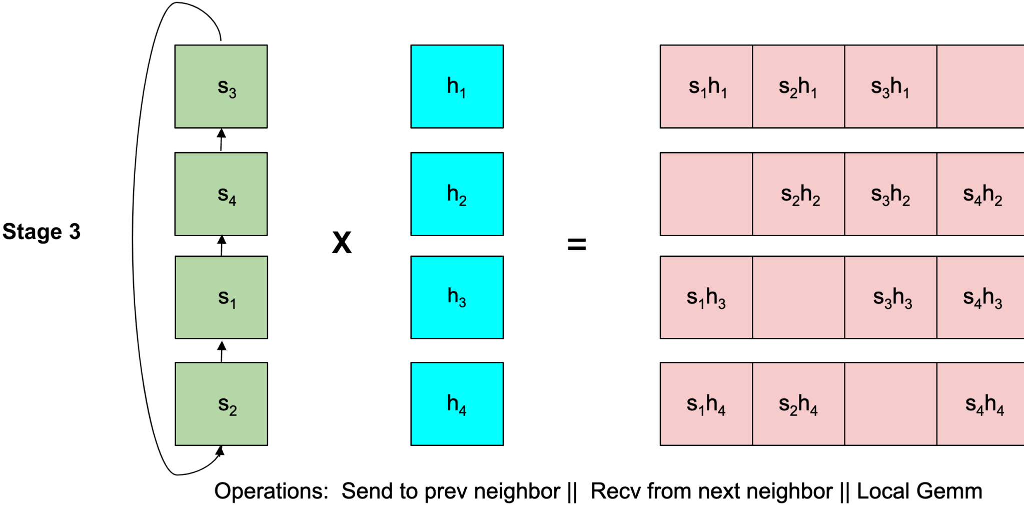


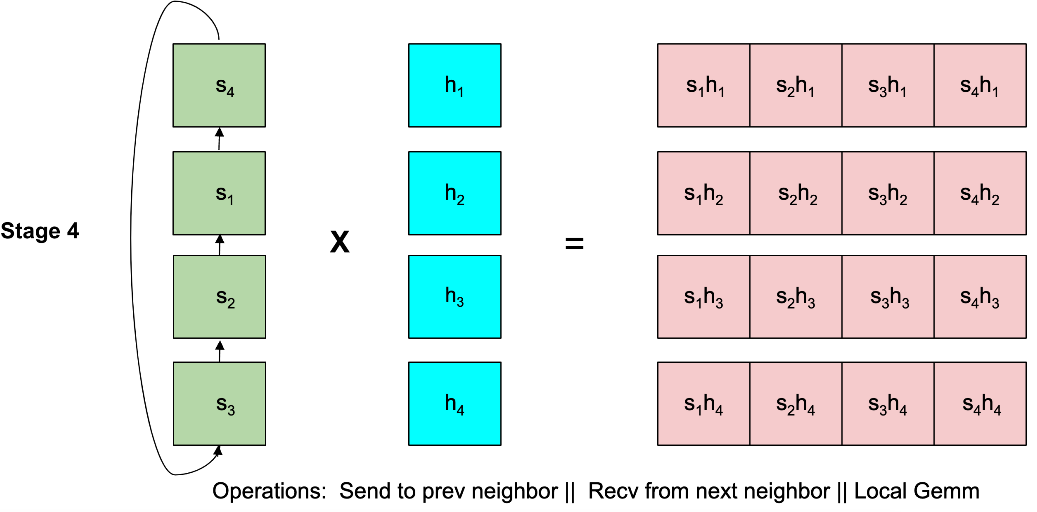


Communication overlap: tensor parallelism

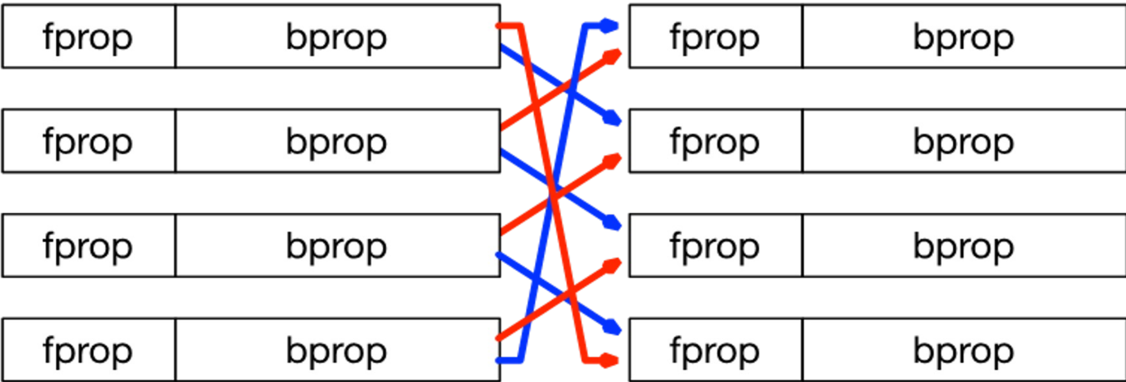


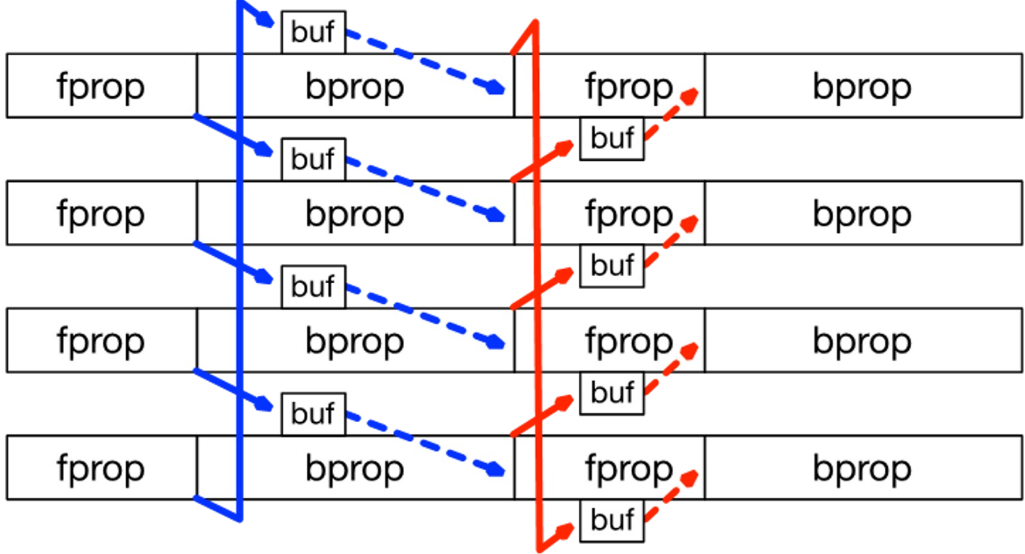




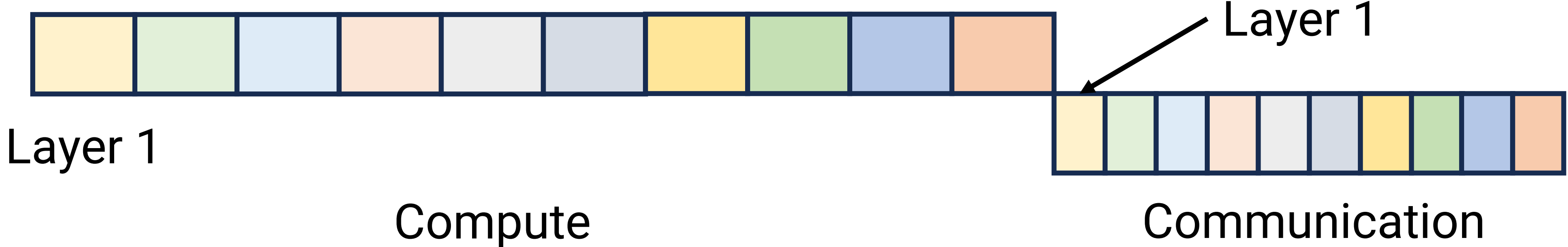


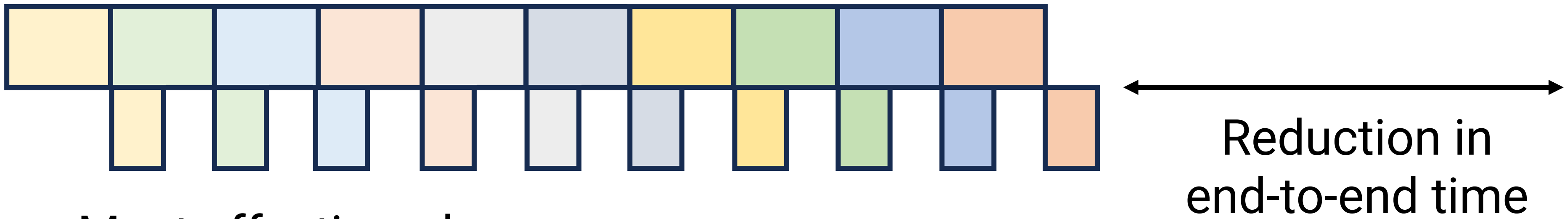
Communication overlap: pipeline parallelism





Communication overlap: data parallelism

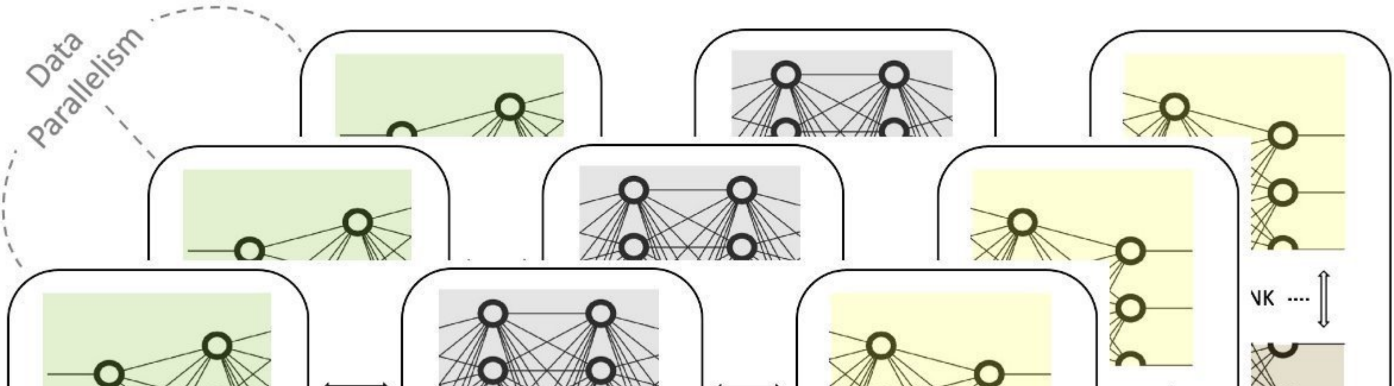




Most effective when
communication time < compute time

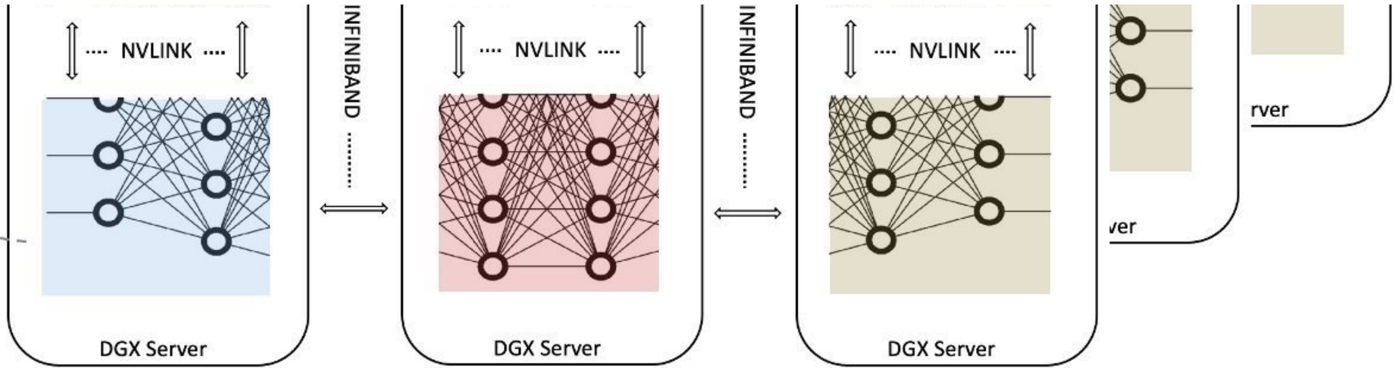
Putting it together for Nemotron-4 340B model

Data parallelism = 64 or 96



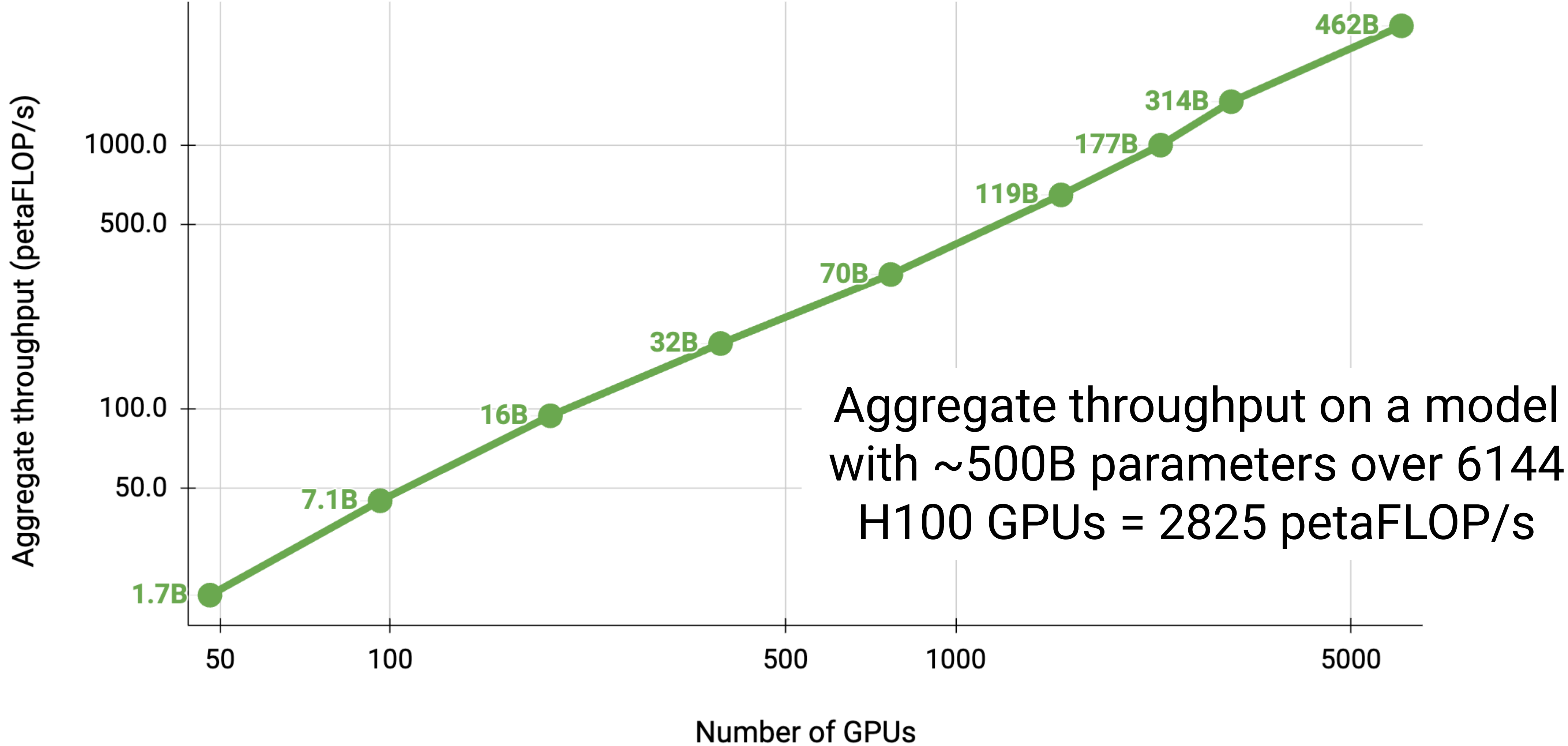
Throughput of ~405 TFLOP/s/GPU with 6144 GPUs (41% of peak)

Tensor parallelism = 8



Pipeline parallelism = 12

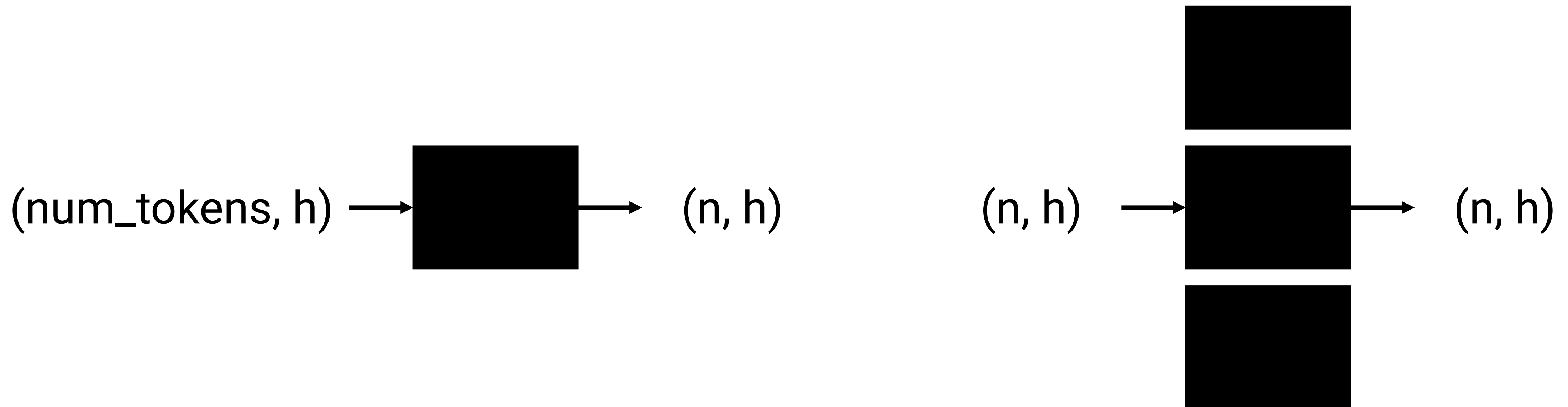
End result: efficient scaling to 1000s of GPUs!



Other forms of parallelism

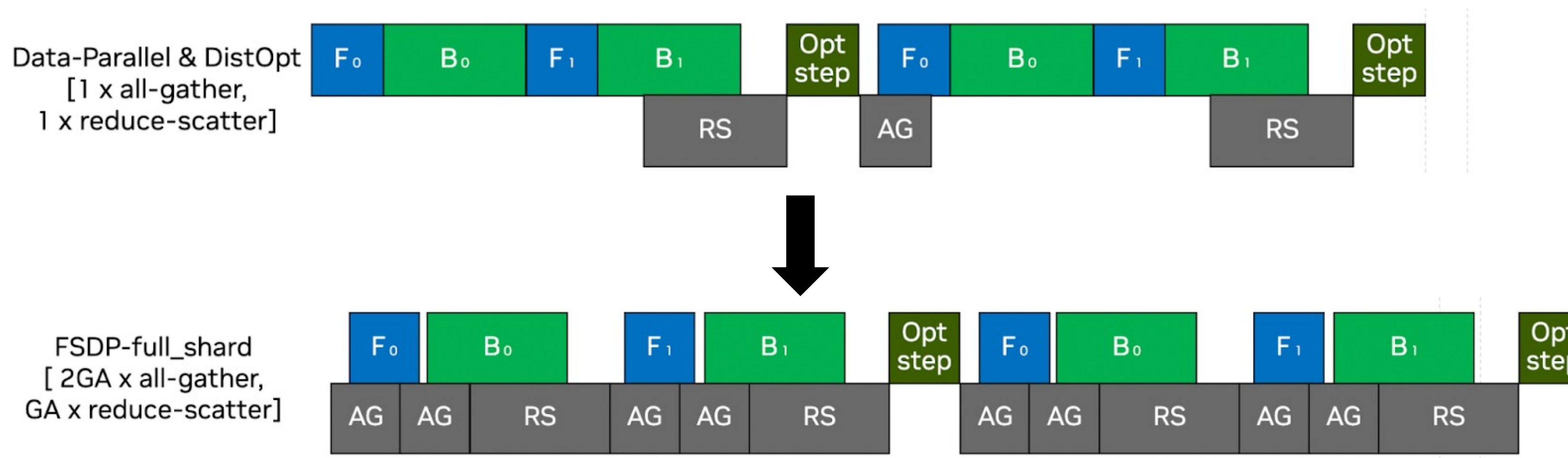
- **Long-context training:** Context parallelism. Intermediate activations of all layers sharded across multiple GPUs along sequence dimension.

- **Mixture-of-expert models:** Expert parallelism
  - Expert weights sharded across multiple GPUs.



Other forms of parallelism

- **Training of large models with no model code changes*:** FSDP
  - Just-in-time gathering of model parameters when needed for forward and backward computations
  - **Pros:** should just work out of the box with a DDP-like wrapper*
  - **Cons:** involves a lot of communication and can lead to poor throughput, especially if you are not careful



Open questions

- Can we automate partitioning instead of applying human intuition for every different type of model architecture?

- What about inference?
  - Autoregressive inference involves potentially long bandwidth-bound token generation phase (GEMVs instead of GEMMs)
  - Requests of different types and sizes can come in at any time without prior notification, making it hard to schedule computation efficiently

- What about self-play models like o1?

Megatron-LM: efficient scaling to 1000s of GPUs

We can train transformer models with 100s of billions of parameters at scale with high efficiency: **2825 petaFLOP/s or 47% of peak**

- Carefully composing pipeline, tensor and data parallelism
- Overlapping communication with computation as much as possible

Implementation available at https://github.com/nvidia/megatron-lm

https://deepakn94.github.io/

dnarayanan@nvidia.com