

Model-Parallel Deep Learning

Efficient DL, Episode VI '23

Yandex
Research



Training large DL models

~~Model-Parallel Deep Learning~~

Efficient DL, Episode III '22

Yandex
Research

LAMBDA 



Recap: large models

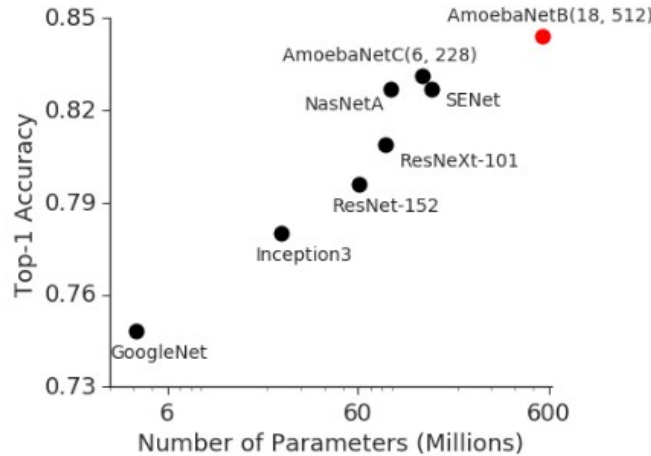
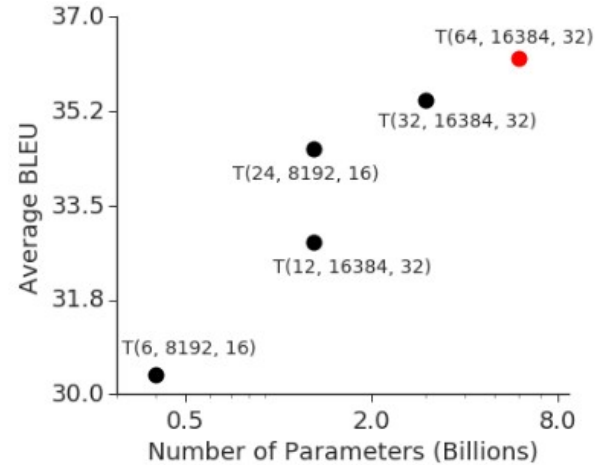


Image Classification
ImageNet

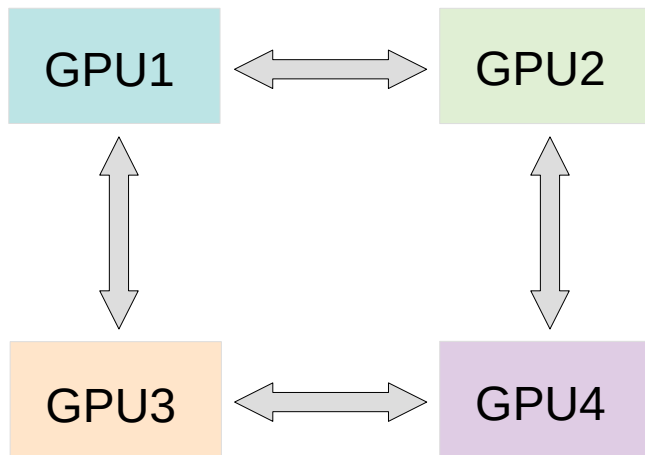


Machine Translation
average over WMT

Source: <https://arxiv.org/abs/1811.06965>

Recap: Ring allreduce

Bonus quest: you can only send data between **adjacent** gpus



Ring topology



Image: [graphcore](#) ipu server

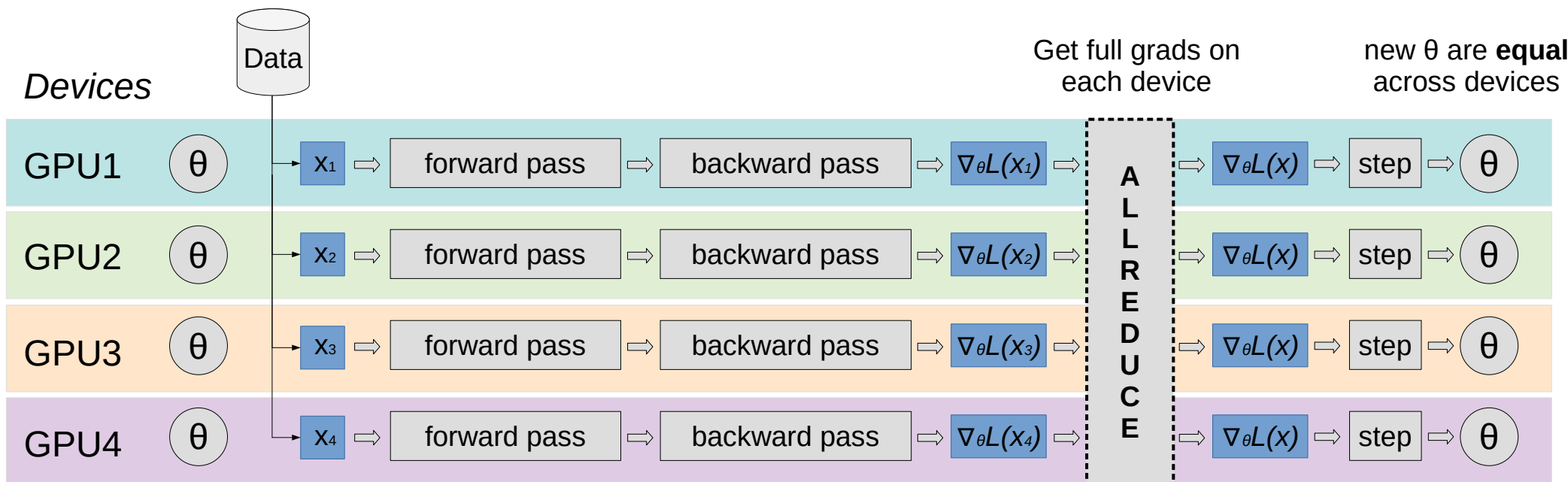
Answer & more: tinyurl.com/ring-allreduce-blog

Recap: All-Reduce SGD

arxiv.org/abs/1706.02677

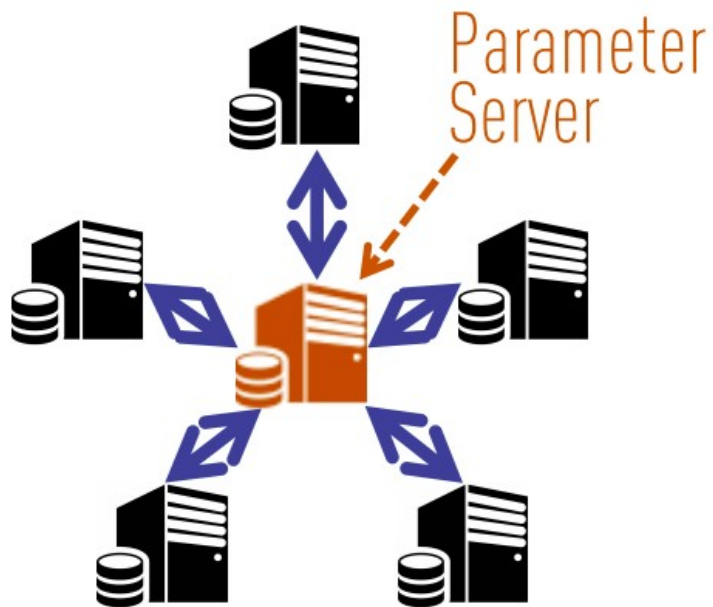
Idea: get rid of the host, each gpu runs its own computation

Q: why will weights be equal after such step?

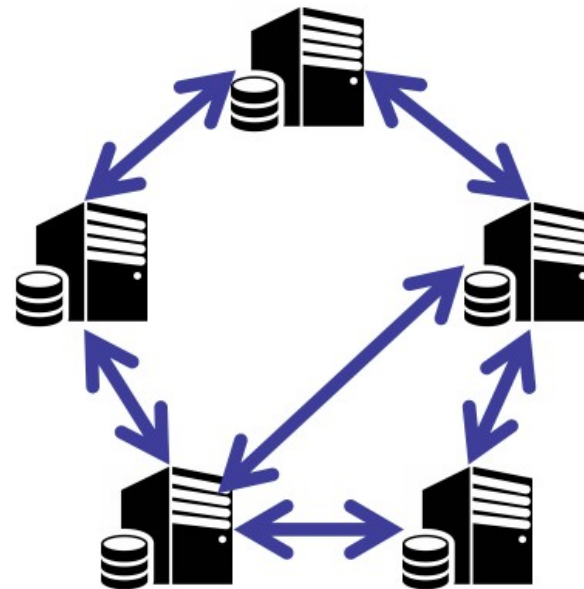


Recap: Decentralized SGD

Gossip (communication): <https://tinyurl.com/boyd-gossip-2006>
Gossip outperforms All-Reduce: <https://tinyurl.com/can-dsgd-outperform>



(a) Centralized Topology

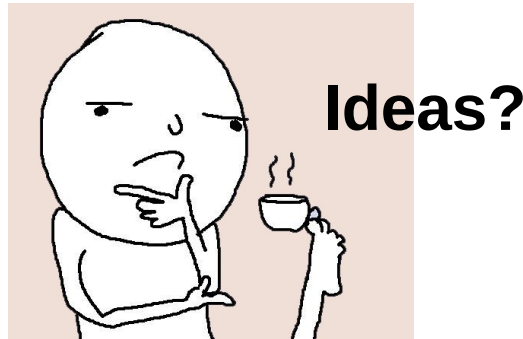


(b) Decentralized Topology

Q: What if a model is larger than GPU?

Q: What if a model is larger than GPU?
easy mode: cannot fit the right batch size
hard mode: cannot fit a single sample
expert mode: not even parameters!

Q: What if a model is larger than GPU?
easy mode: cannot fit the right batch size
hard mode: cannot fit a single sample
expert mode: not even parameters!



Q: What if a model is larger than GPU?
easy mode: cannot fit the right batch size
hard mode: cannot fit a single sample
expert mode: not even parameters!

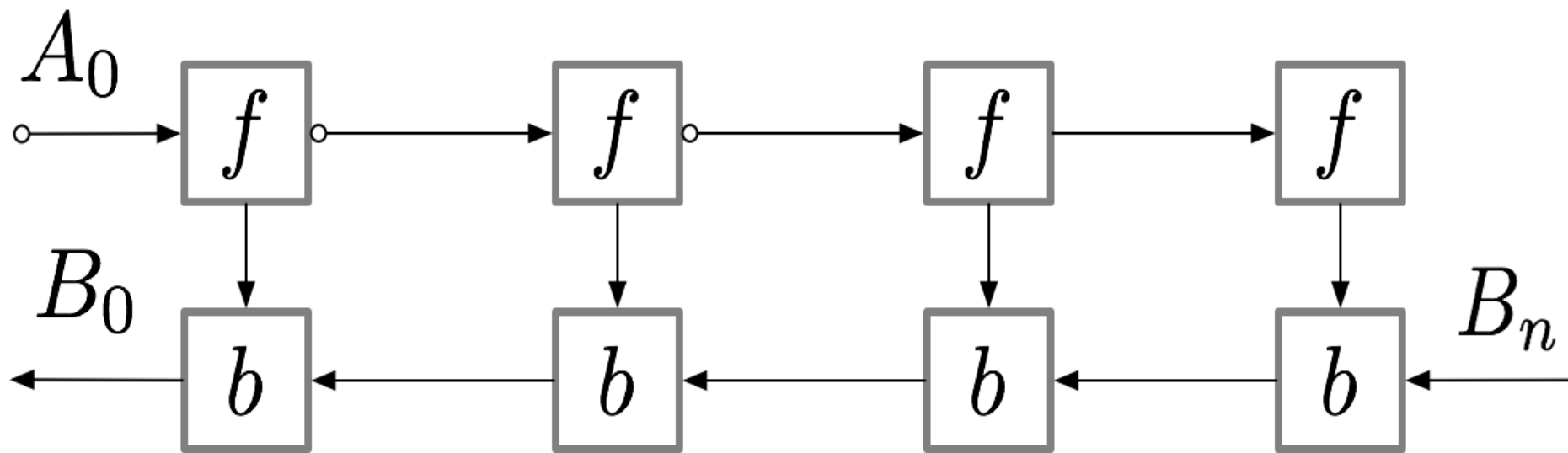
Solution: accumulate
grads from several
training batches

```
[ ] 1 optimizer.zero_grad()  
    2 for i in range(B):  
    3     loss = model(**next_batch())  
    4     (loss / B).backward()  
    5 optimizer.step()
```

Q: What if a model is larger than GPU?
easy mode: cannot fit the right batch size
hard mode: cannot fit one training sample
expert mode: not even parameters!

Gradient checkpointing

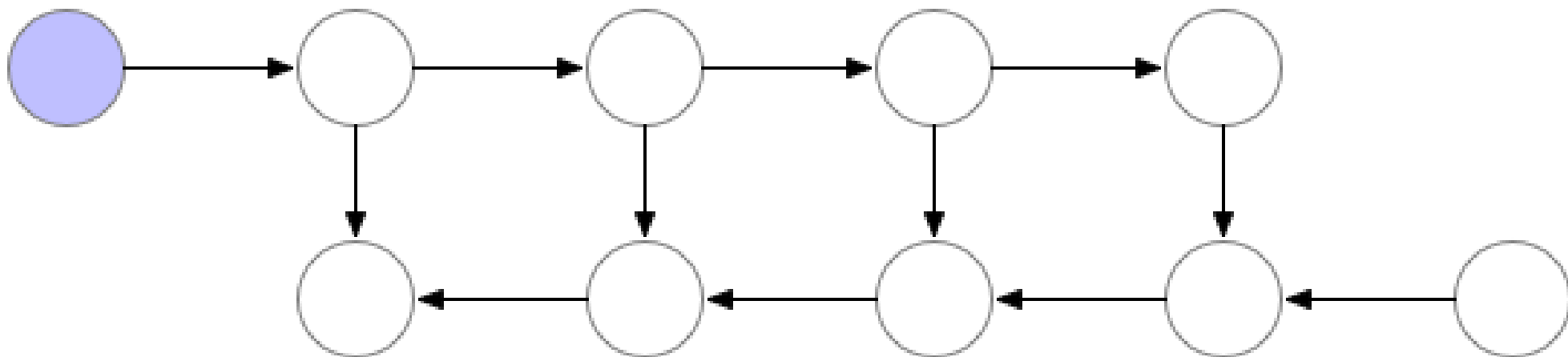
aka rematerialization



Paper (DL): arxiv.org/pdf/1604.06174.pdf
TF: github.com/cybertronai/gradient-checkpointing
Pytorch: pytorch.org/docs/stable/checkpoint.html

Gradient checkpointing

Normal backprop



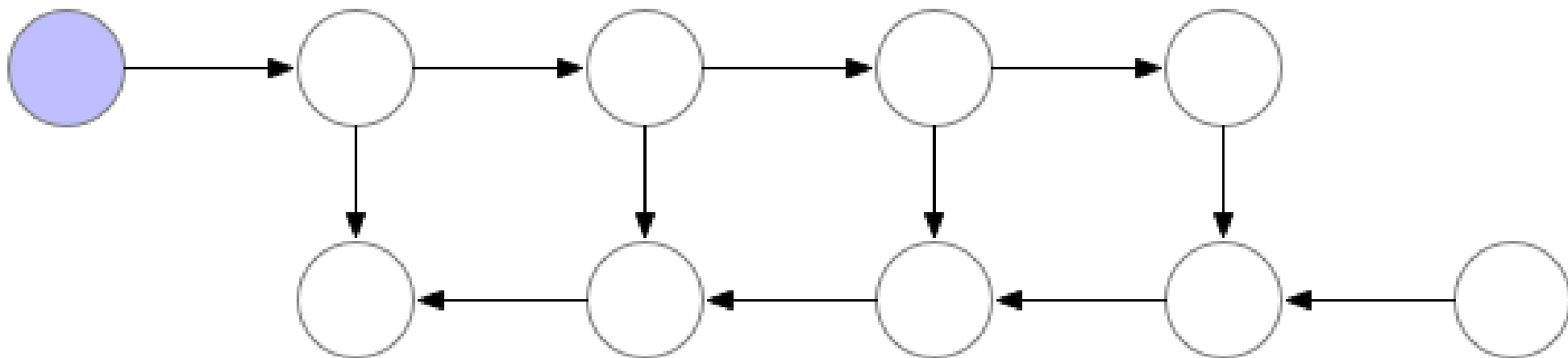
Paper (DL): arxiv.org/pdf/1604.06174.pdf

TF: github.com/cybertronai/gradient-checkpointing

Pytorch: pytorch.org/docs/stable/checkpoint.html

Gradient checkpointing

Full rematerialization



Paper (DL): arxiv.org/pdf/1604.06174.pdf

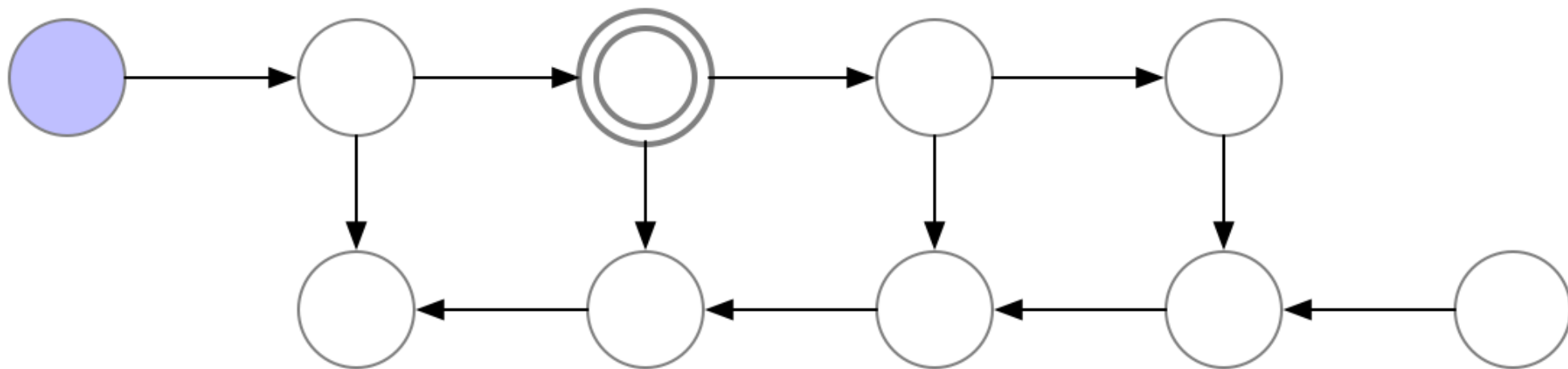
TF: github.com/cybertronai/gradient-checkpointing

Pytorch: pytorch.org/docs/stable/checkpoint.html

Gradient checkpointing

Single checkpoint

checkpoint



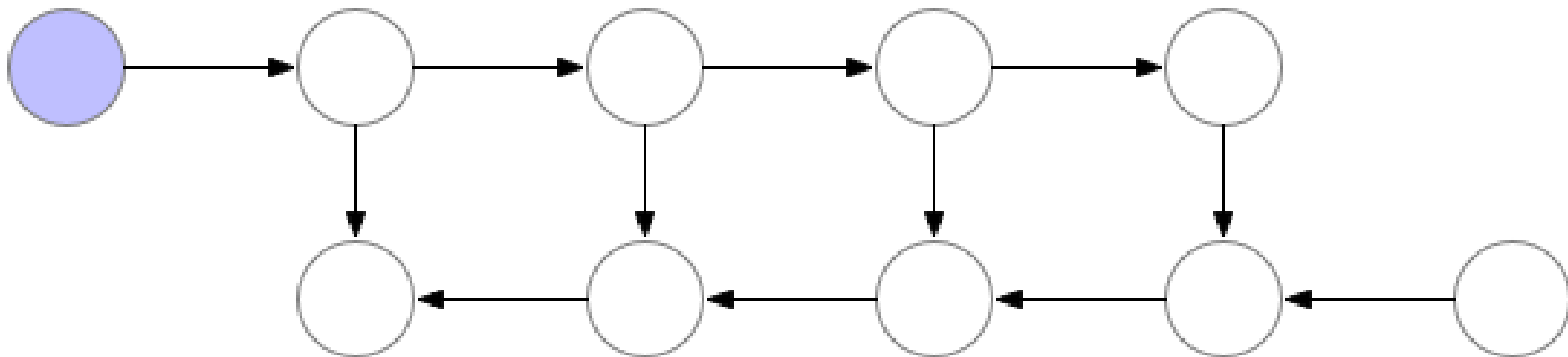
Paper (DL): arxiv.org/pdf/1604.06174.pdf

TF: github.com/cybertronai/gradient-checkpointing

Pytorch: pytorch.org/docs/stable/checkpoint.html

Gradient checkpointing

Single checkpoint



Paper (DL): arxiv.org/pdf/1604.06174.pdf

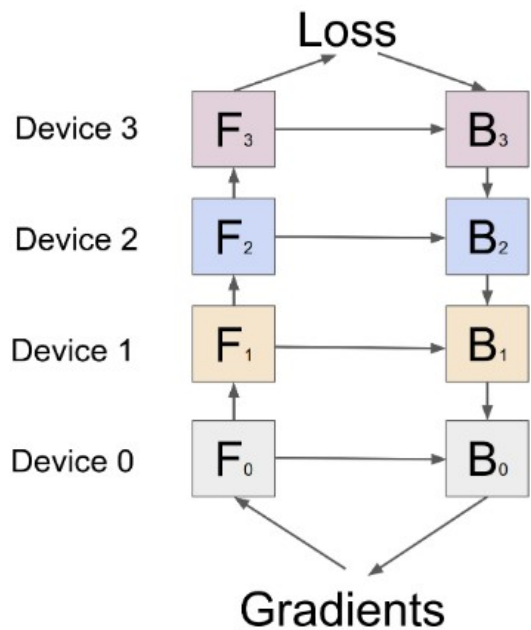
TF: github.com/cybertronai/gradient-checkpointing

Pytorch: pytorch.org/docs/stable/checkpoint.html

Q: What if a model is larger than GPU?
easy mode: cannot fit batch size 1
expert mode: not even parameters!

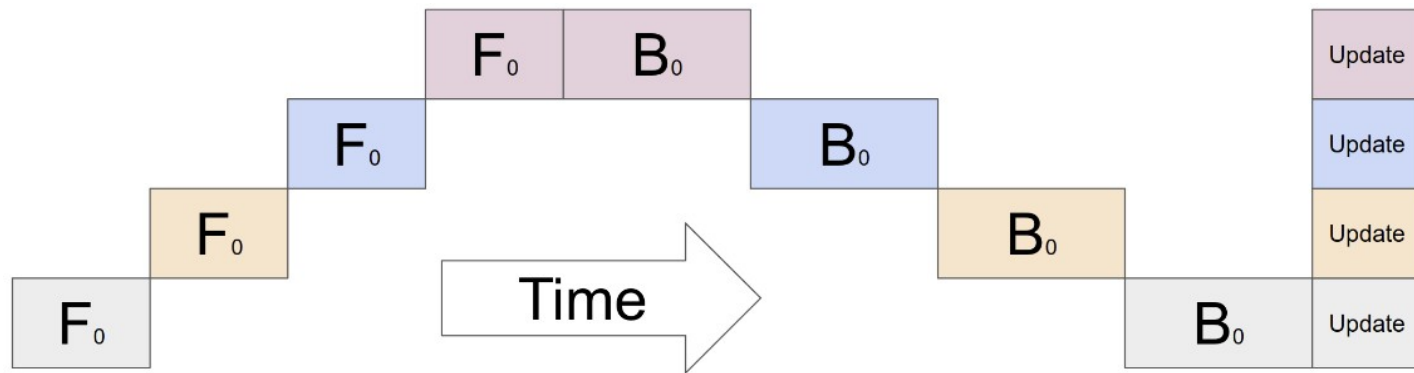
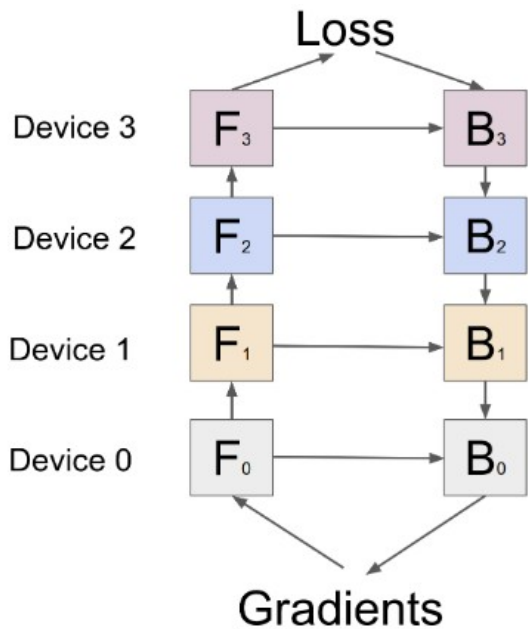
Model-parallel training

Q: What if a model is larger than GPU?



Model-parallel training

Q: What if a model is larger than GPU?



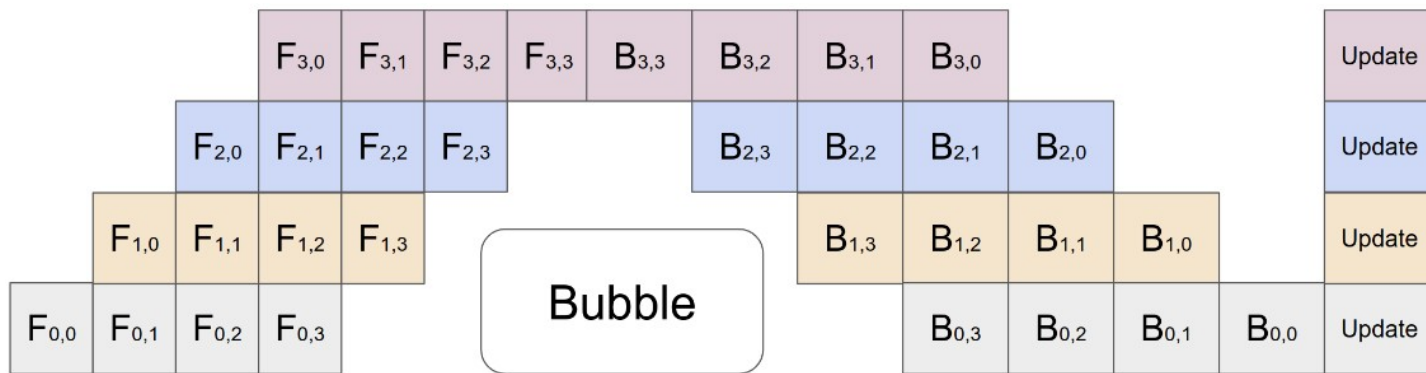
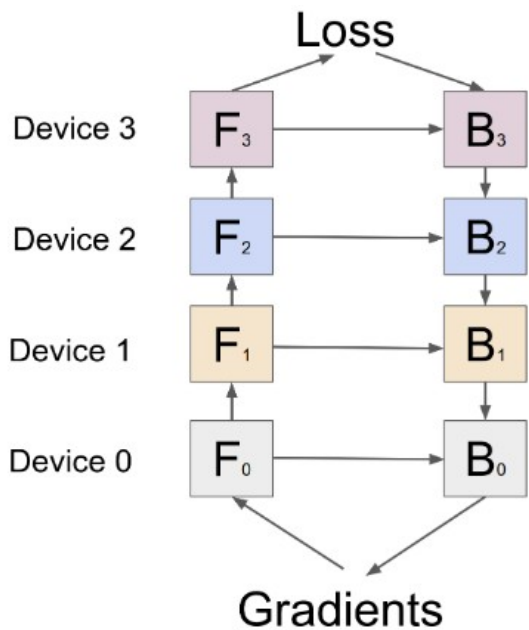
model size: $O(N)$
throughput: $O(1)$

Q: Can we go faster?

Pipelining

GPipe: arxiv.org/abs/1811.06965 – good starting point, *not* the 1st paper

Idea: split data into micro-batches and form a pipeline (right)

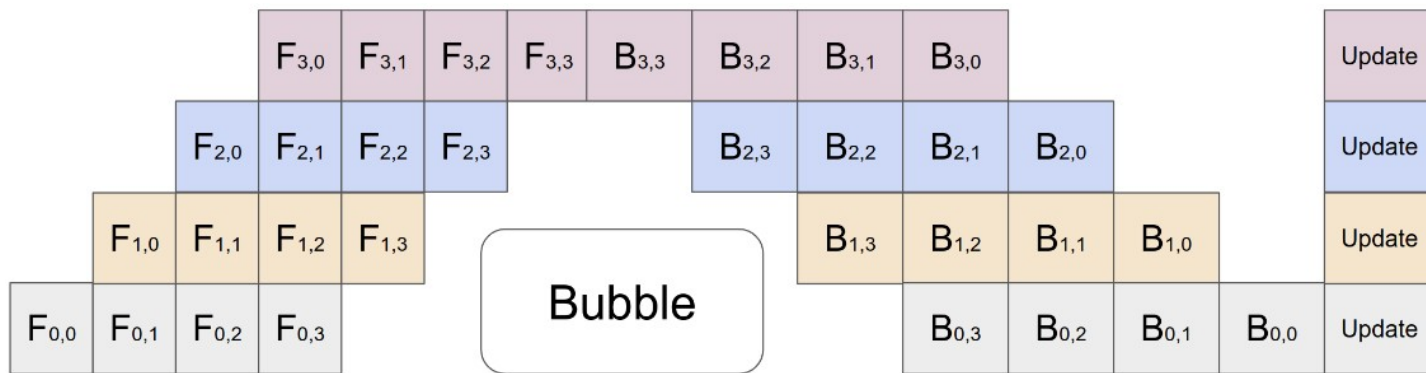
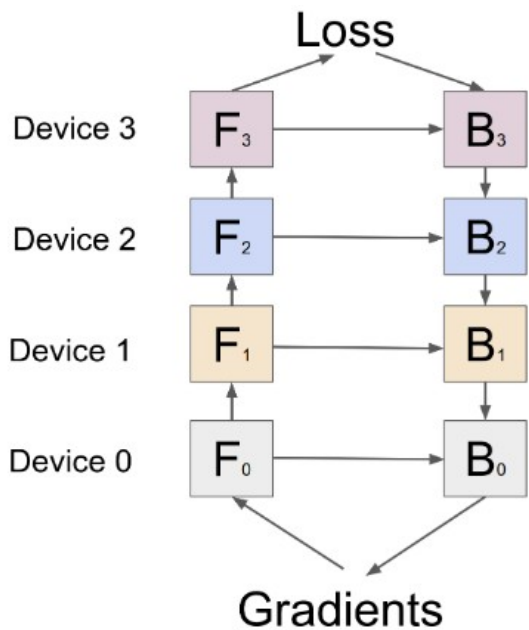


model size: $O(n)$
throughput: $O(n)$ – with caveats

Pipelining

GPipe: arxiv.org/abs/1811.06965 – good starting point, *not* the 1st paper

Idea: split data into micro-batches and form a pipeline (right)



model size: $O(n)$
throughput: $O(n)$ – with caveats

Q: Even faster?

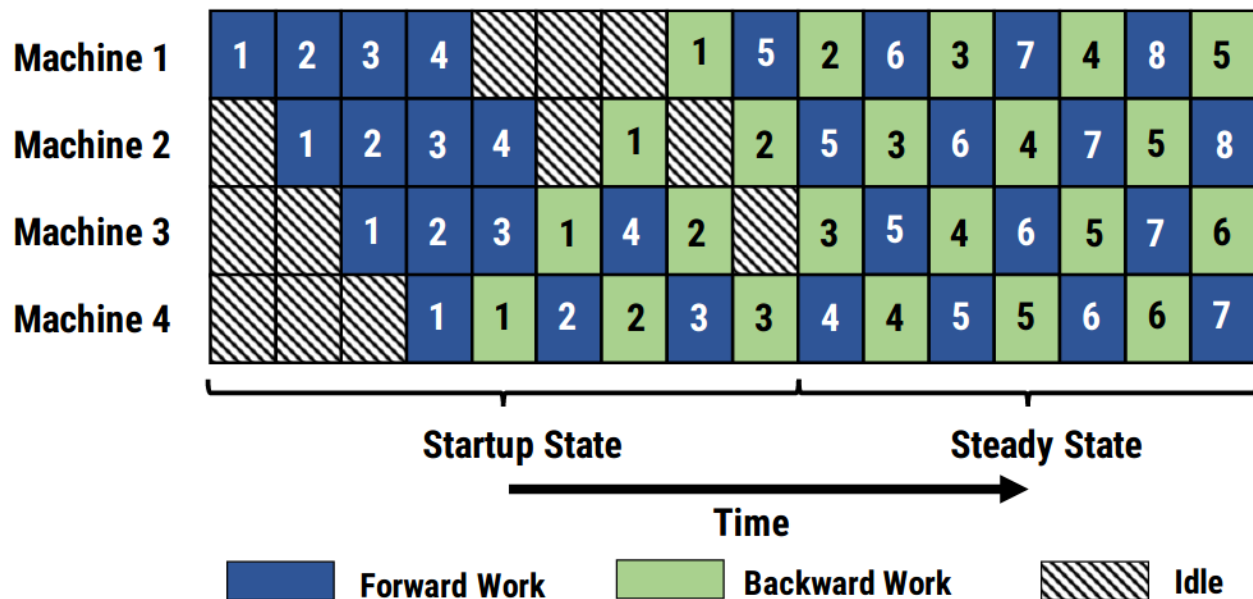
Pipeline-parallel training

PipeDream: arxiv.org/abs/1806.03377

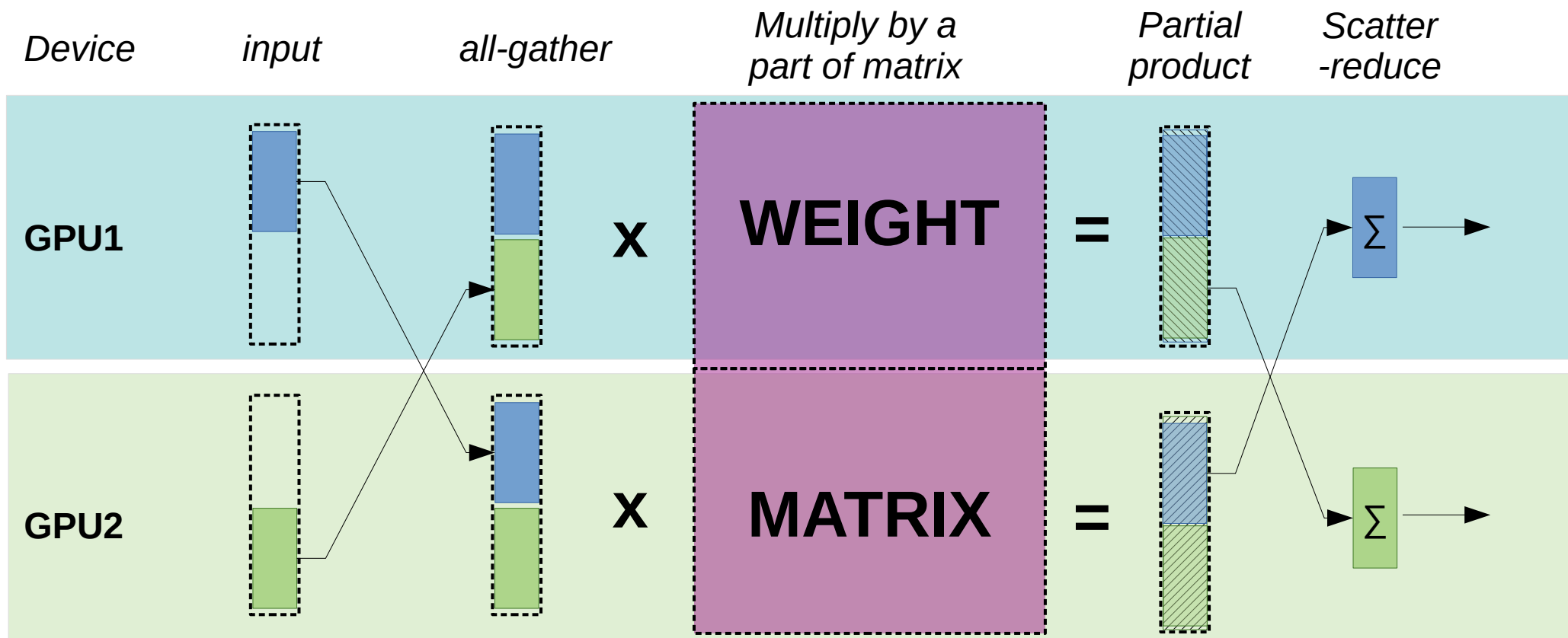
Idea: apply gradients with every microbatch for maximum throughput

Also neat:

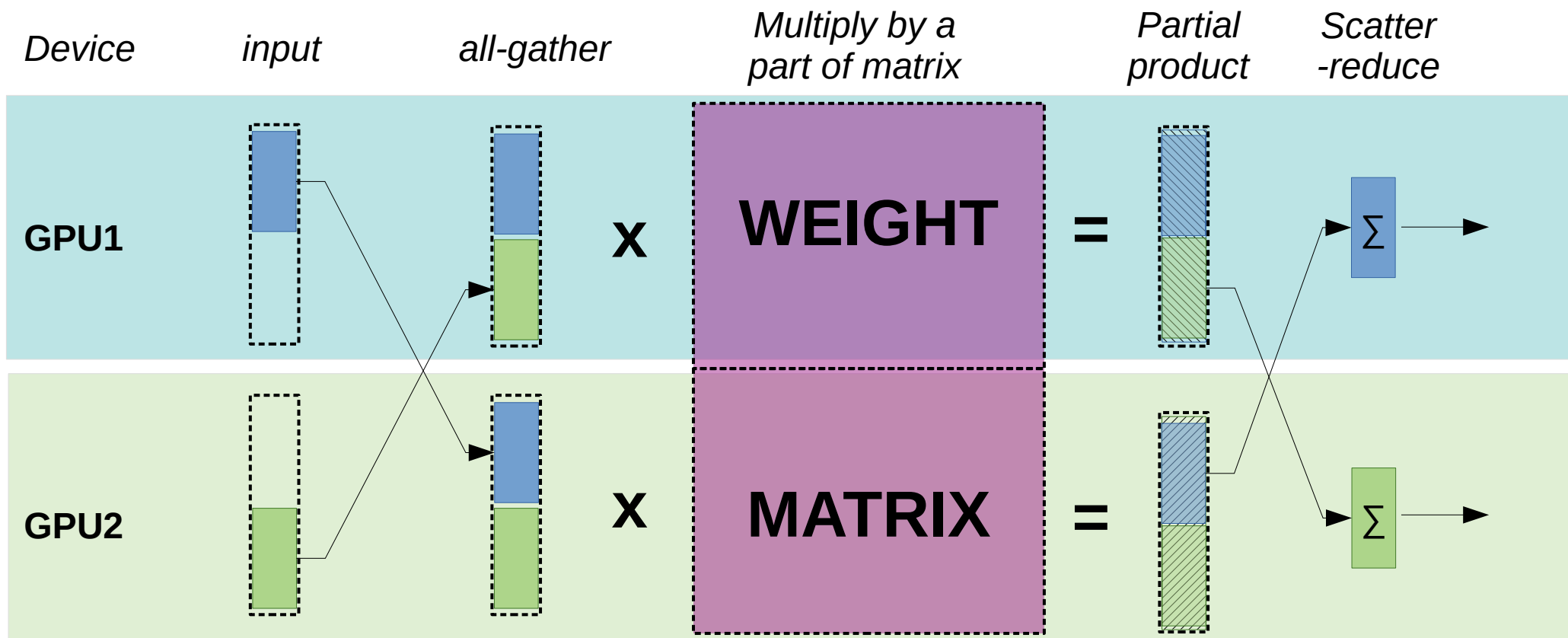
- Automatically partition layers to GPUs via dynamic programming
- Store k past weight versions to reduce gradient staleness
- Aims at high latency



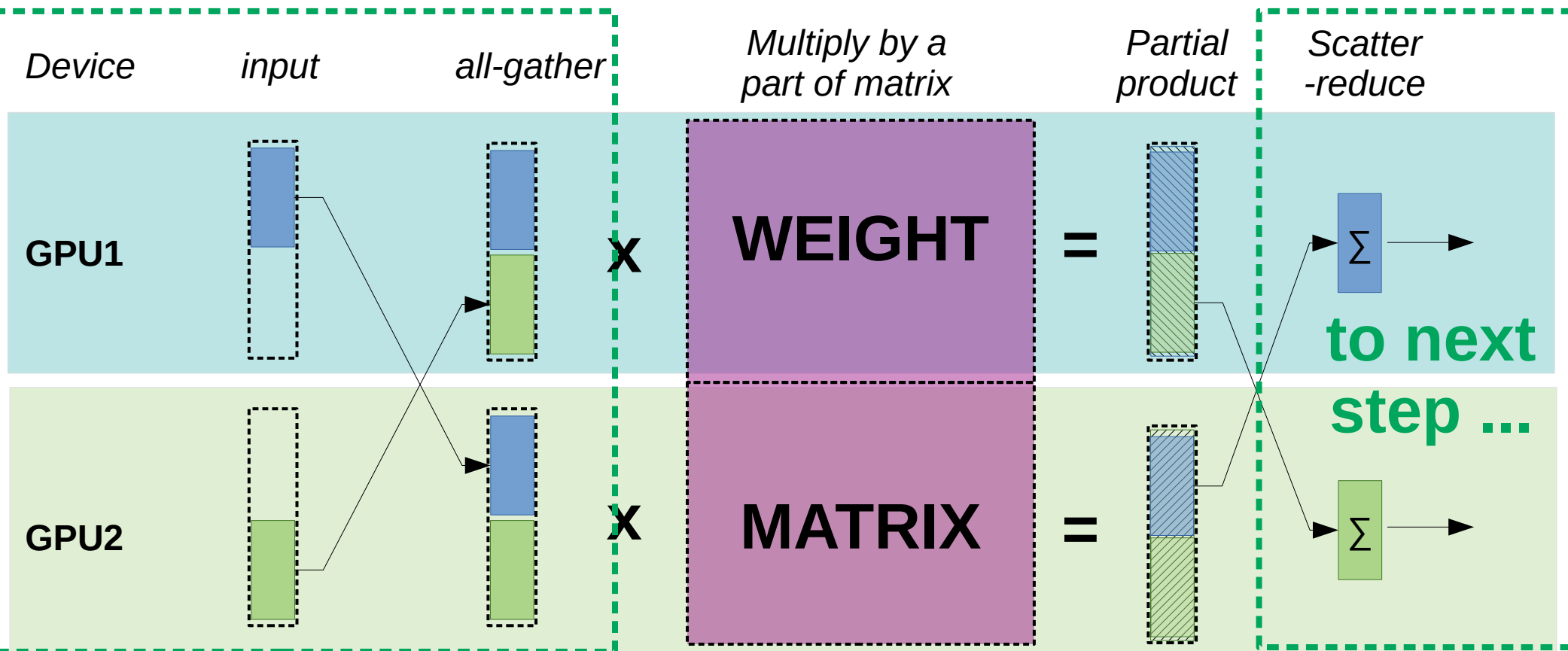
Tensor-parallel training



Q: find AllReduce op here



Q: find AllReduce op here



</Model-parallel>

- + model larger than GPU
- + faster for small
- * typical size: 2-8 gpus
- model partitioning is tricky
 - tensor parallelism is easier, but requires ultra low latency
- latency is critical, go buy nvlink
 - except for PipeDream*
- *often combined with gradient checkpointing*

Tutorials:

- Simple pipelining in PyTorch – tinyurl.com/pytorch-pipelining
- Distributed model-parallel with torch RPC - <https://tinyurl.com/torch-rpc>
- Automatic tensor parallelism [pip install tensor_parallel](#)

</Model-parallel>

- + model larger than GPU
- + faster for small
- * typical size: 2-8 gpus
- model partitioning is tricky
 - tensor parallelism is easier, but requires ultra low latency
- latency is critical, go buy nvlink
 - except for PipeDream*
- *often combined with gradient checkpointing*

Tutorials:

- Simple pipelining in PyTorch – tinyurl.com/pytorch-pipelining
- Distributed model-parallel with torch RPC - <https://tinyurl.com/torch-rpc>
- Automatic tensor parallelism [pip install tensor_parallel](#)

Q: what if you have 1024 GPUs, but the model fits on 8?

</Model-parallel>

- + model larger than GPU
- + faster for small
- * typical size: 2-8 gpus
- model partitioning is tricky
 - tensor parallelism is easier, but requires ultra low latency
- latency is critical, go buy nvlink
 - except for PipeDream*
- *often combined with gradient checkpointing*

Tutorials:

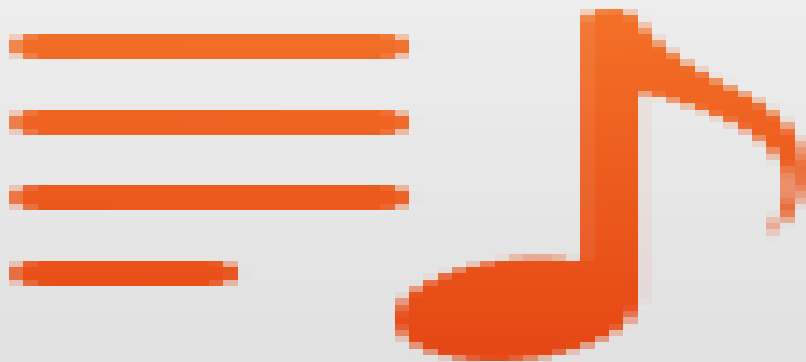
- Simple pipelining in PyTorch – tinyurl.com/pytorch-pipelining
- Distributed model-parallel with torch RPC - <https://tinyurl.com/torch-rpc>
- Automatic tensor parallelism [pip install tensor_parallel](#)

Large-scale training: combine model- and data-parallel

How 'bout a short break?
[after the break: ~~movies!~~ cool video]

Case study: DeepSpeed

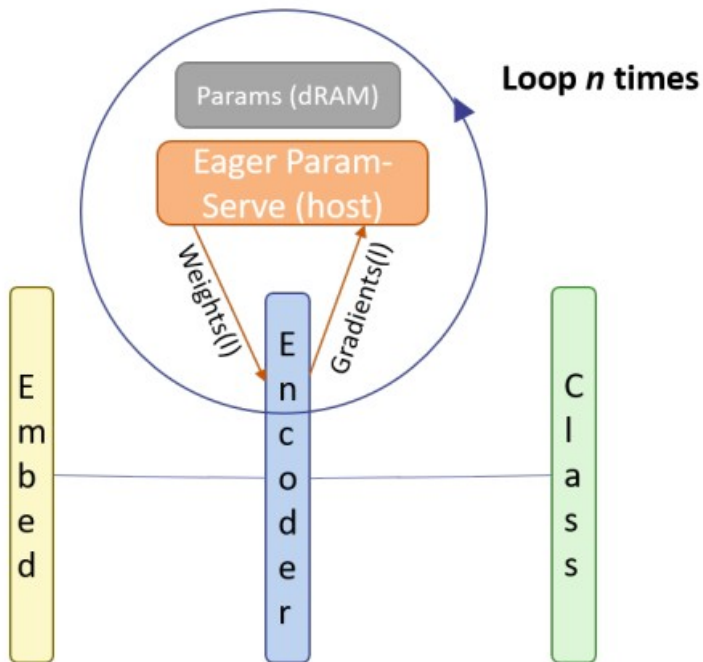
Source: [microsoft](#)



Memory offloading

L2L: <https://arxiv.org/abs/2002.05645>

EPS with L2L execution



- Initialize all layers on CPU
- Move k layers at a time to GPU
- Remove layers after computation
- Fetch $k+1$ -st layer while k -th runs
- Still 20-50% overhead

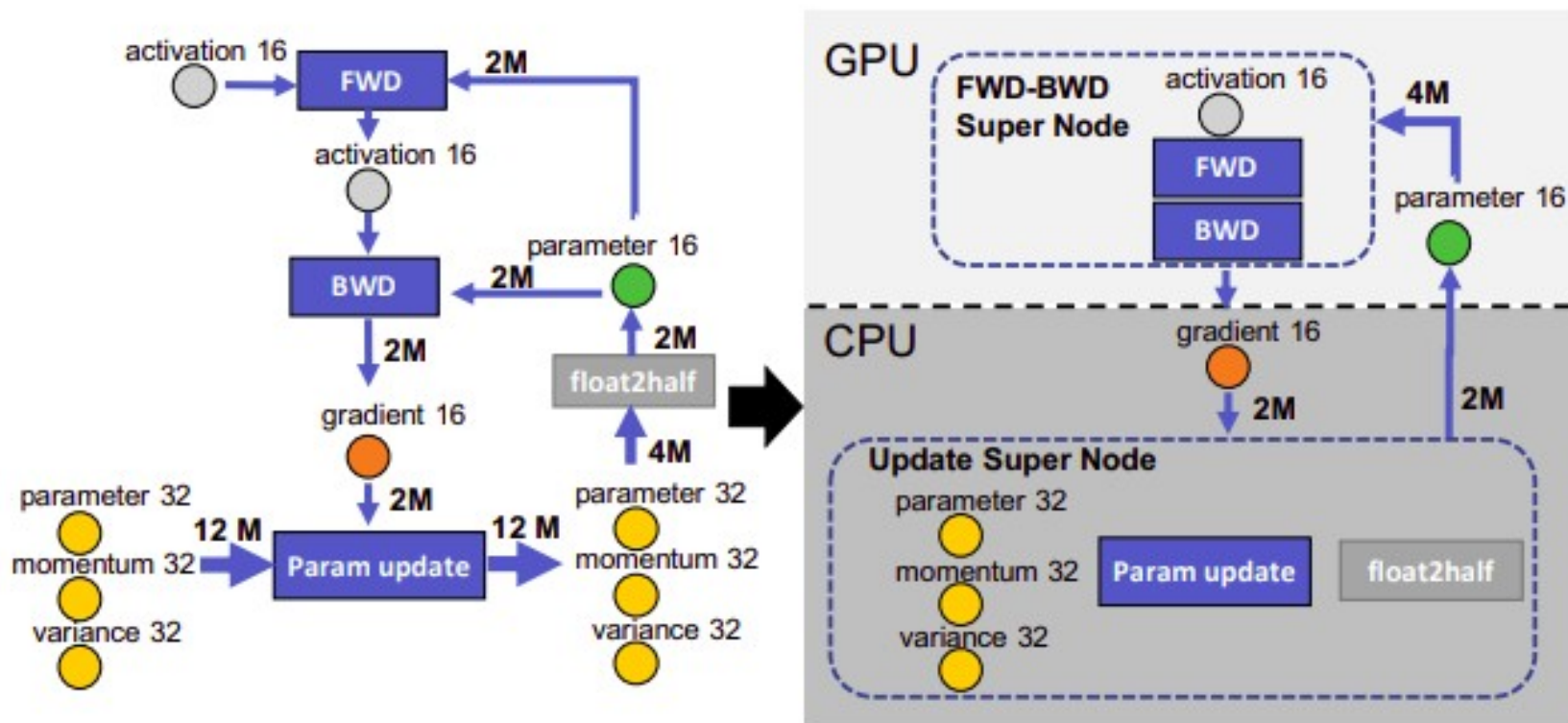
Memory offloading

L2L: <https://arxiv.org/abs/2002.05645>

METHOD	UBATCH SIZE	DEVICE BATCH SIZE	#LAYER	#PARAMETERS	MEMORY (GB)
BASILINE	2	2	24	300 MILLION	9.23
BASILINE	2	2	48	600 MILLION	OOM
L2L-STASH ON GPU	64	64	24	300 MILLION	5.22
L2L-STASH ON GPU	64	64	48	600 MILLION	6.76
L2L-STASH ON GPU	64	64	96	1.2 BILLION	9.83
L2L-STASH ON CPU	64	64	24	300 MILLION	3.69
L2L-STASH ON CPU	64	64	96	1.2 BILLION	3.69
L2L-STASH ON CPU	64	64	384	4.8 BILLION	3.69

Memory offloading

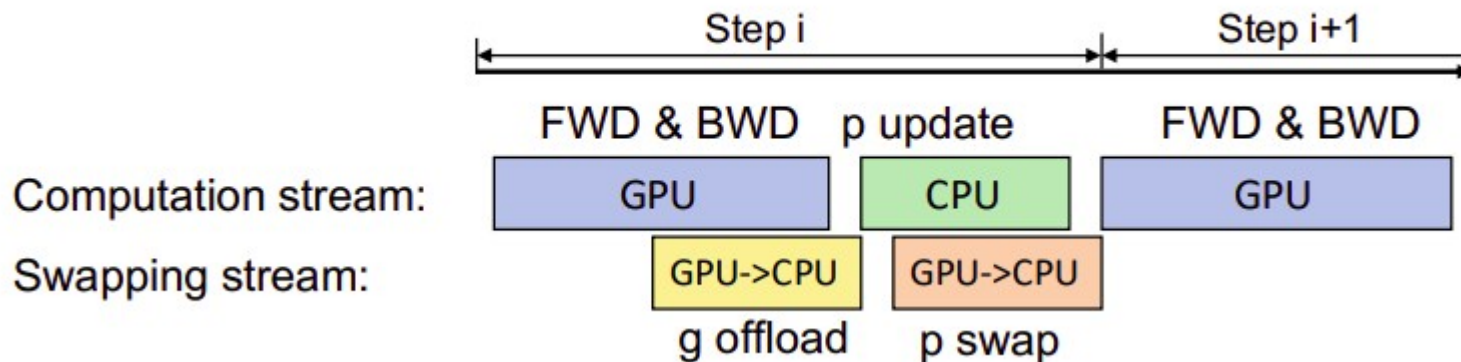
ZeRO-offload: <https://arxiv.org/abs/2101.06840>



Memory offloading

ZeRO-offload: <https://arxiv.org/abs/2101.06840>

- Offload **in parallel** with computation
- Use gradient checkpointing
- Delayed parameter update



Memory offloading

ZeRO-offload: <https://arxiv.org/abs/2101.06840>

- Offload in parallel with computation
- Use gradient checkpointing
- **Delayed parameter update**

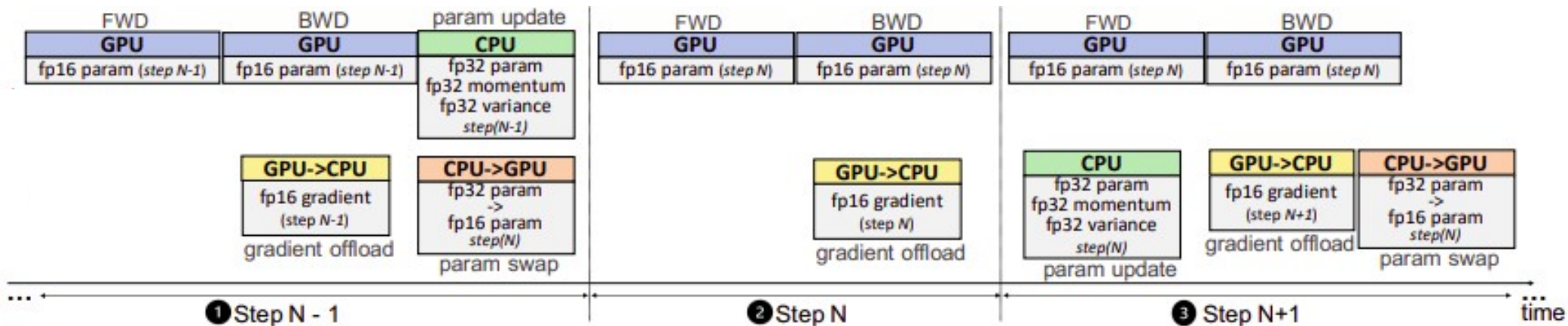
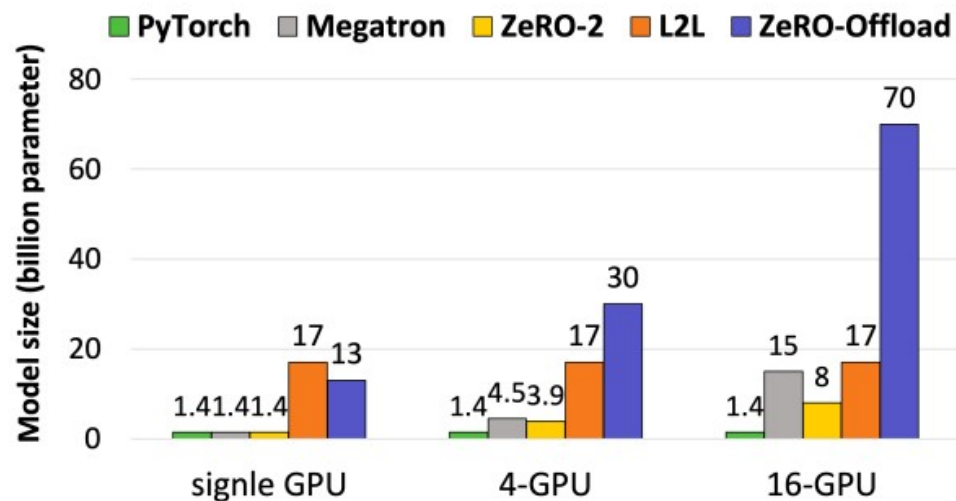
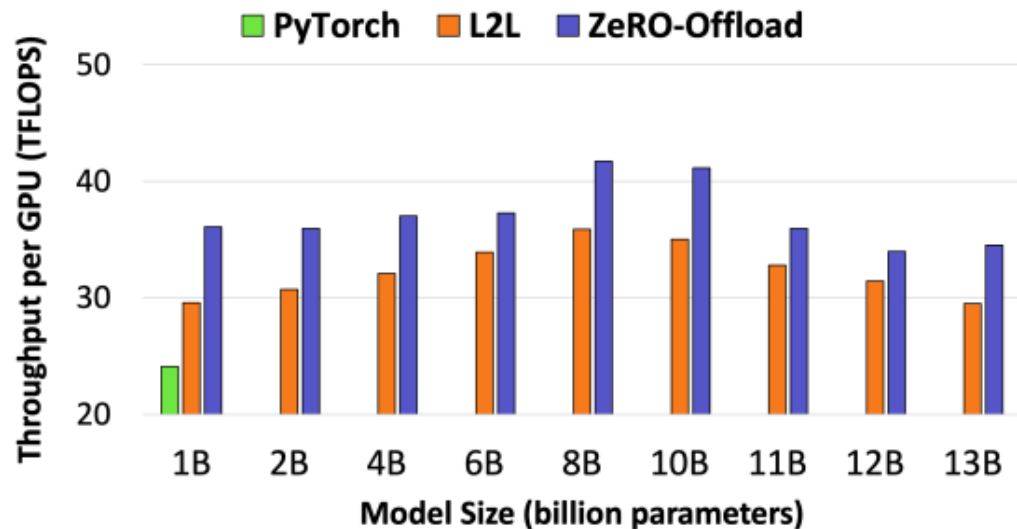


Figure 6: Delayed parameter update during the training process.

Memory offloading

ZeRO-offload: <https://arxiv.org/abs/2101.06840>

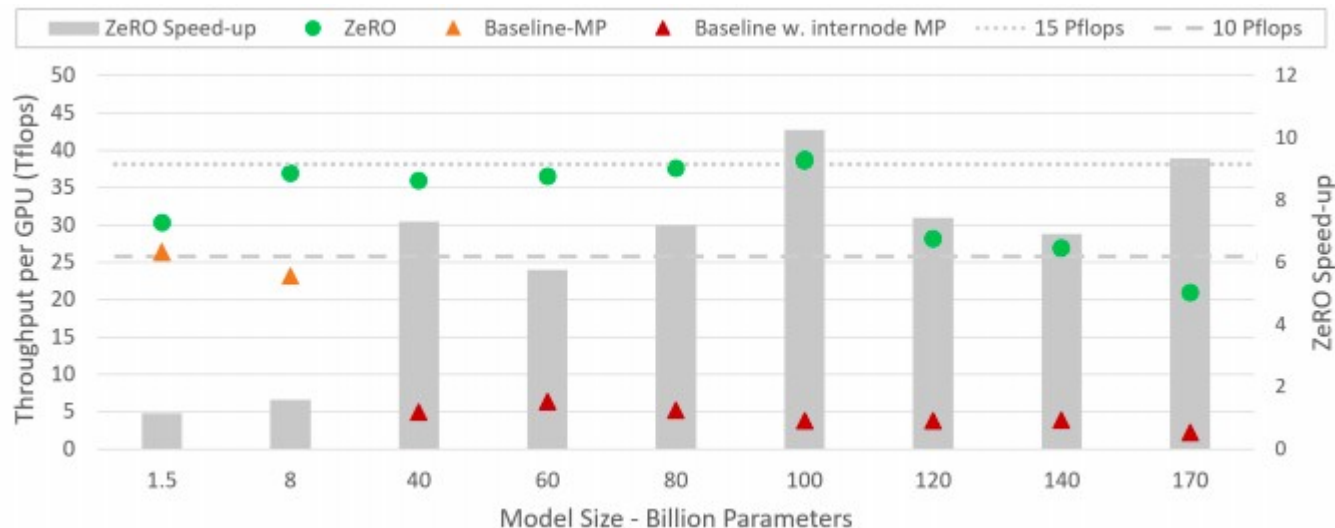
- Offload in parallel with computation
- Use gradient checkpointing
- Delayed parameter update



DeepSpeed / ZeRO

ZeRO: <https://arxiv.org/pdf/1910.02054v3.pdf>

- Combines sharded DPP and offload
- ... and some tensor parallelism
- ... and a ton of hacks



</ZeRO>

Multi-GPU strategies:

- * Pipeline model-parallel – allocate layers on different GPUs
- * Sharded data-parallel – split optimizer state and/or parameters

Single GPU strategies:

- * Small model – gradient checkpointing & virtual batch
- * Large model – optimizer state sharding (keep parameters on GPU)

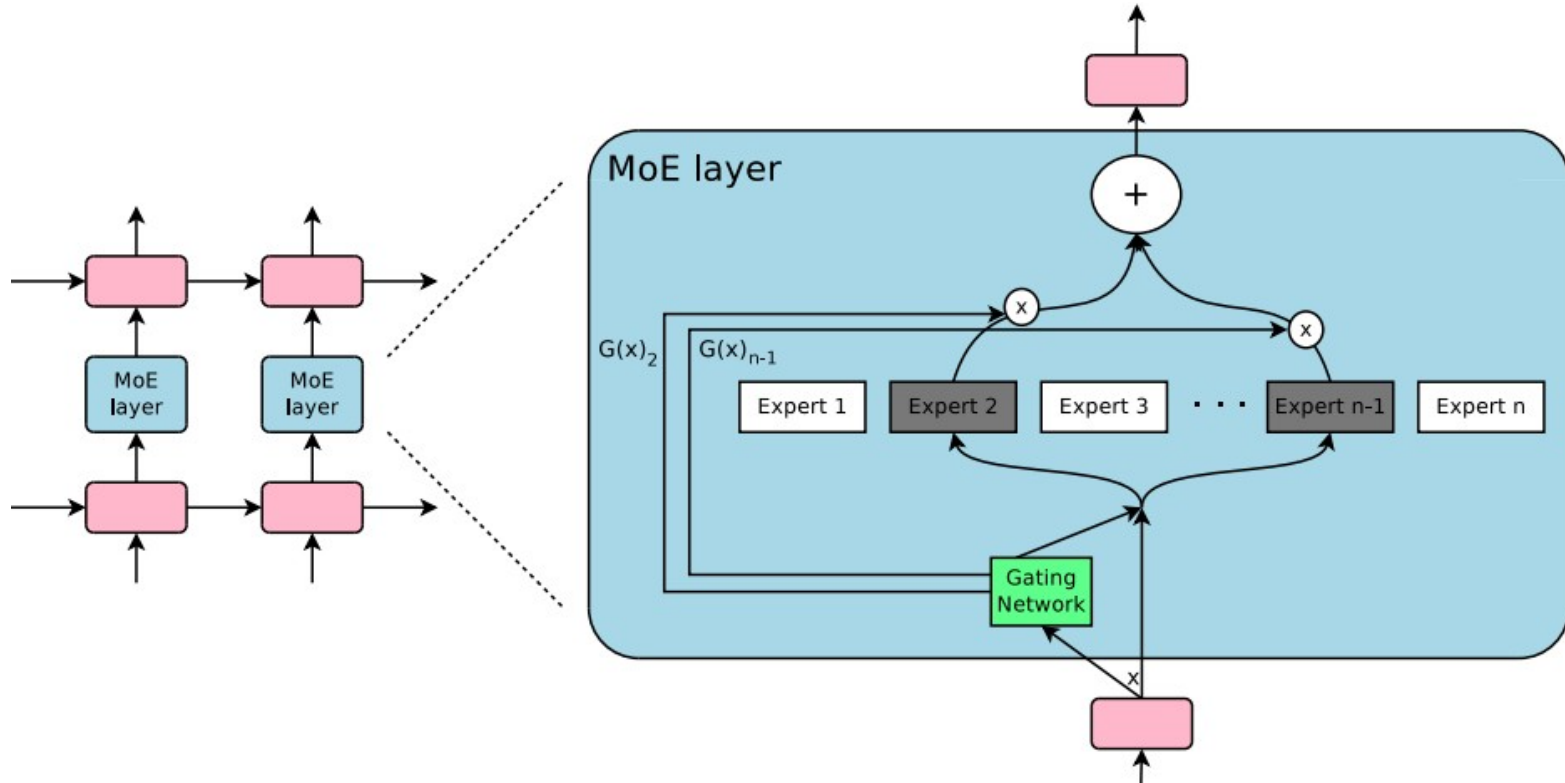
Implementations:

- **DeepSpeed** – sharded DP, offload, tensor parallelism, active development
 - Offload – <https://www.deepspeed.ai/news/2021/03/07/zero3-offload.html>
- **Fairscale** – most of DeepSpeed features with friendlier API
 - One great implementation – <https://github.com/NVIDIA/Megatron-LM>

If we have time...
(if not, skip to autoparallel)

Expert Parallelism

Sparsely gated MoE: <https://arxiv.org/pdf/1701.06538.pdf>

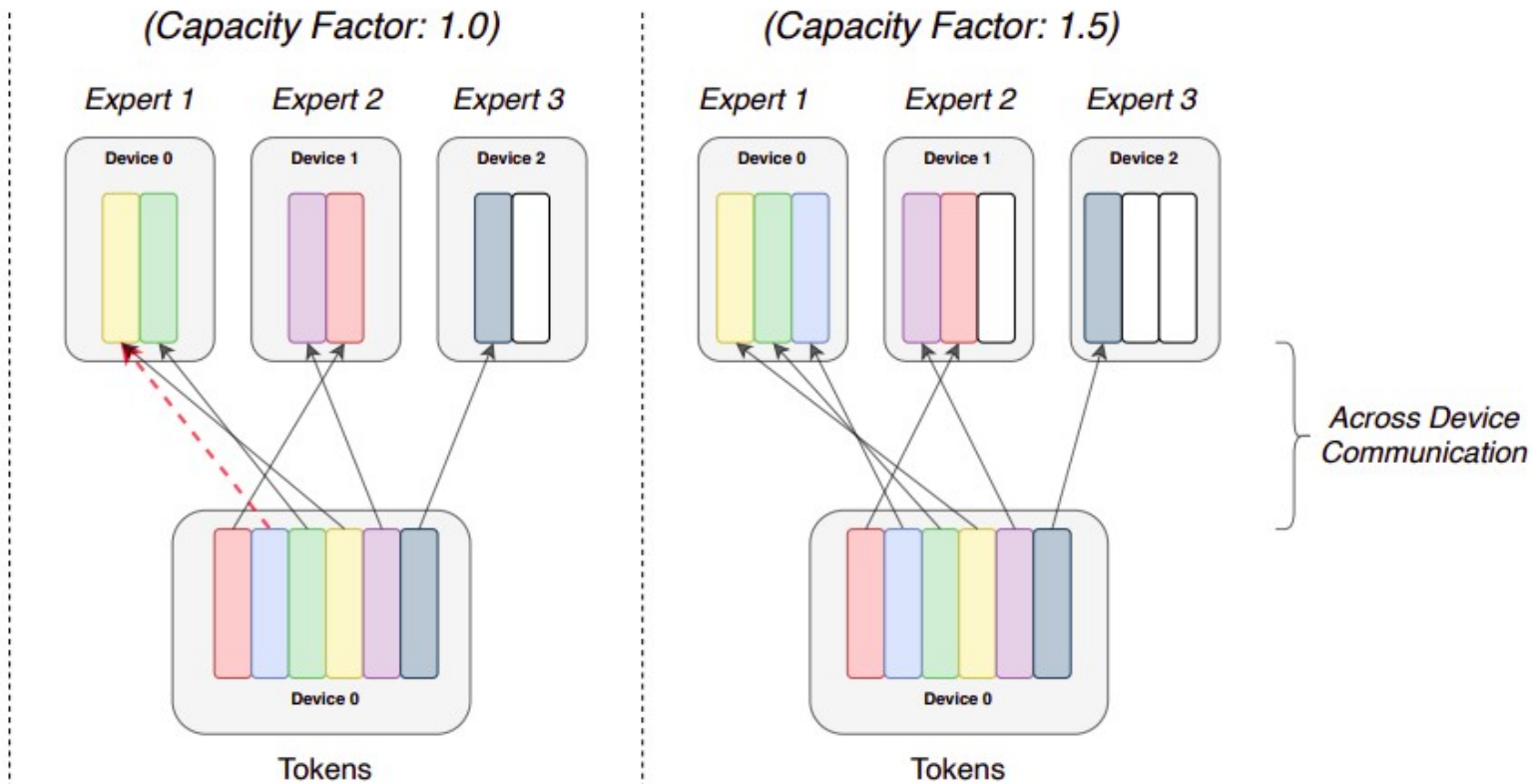


MoE Variant: Switch Transformer

Switch: <https://arxiv.org/pdf/2101.03961.pdf>

Terminology

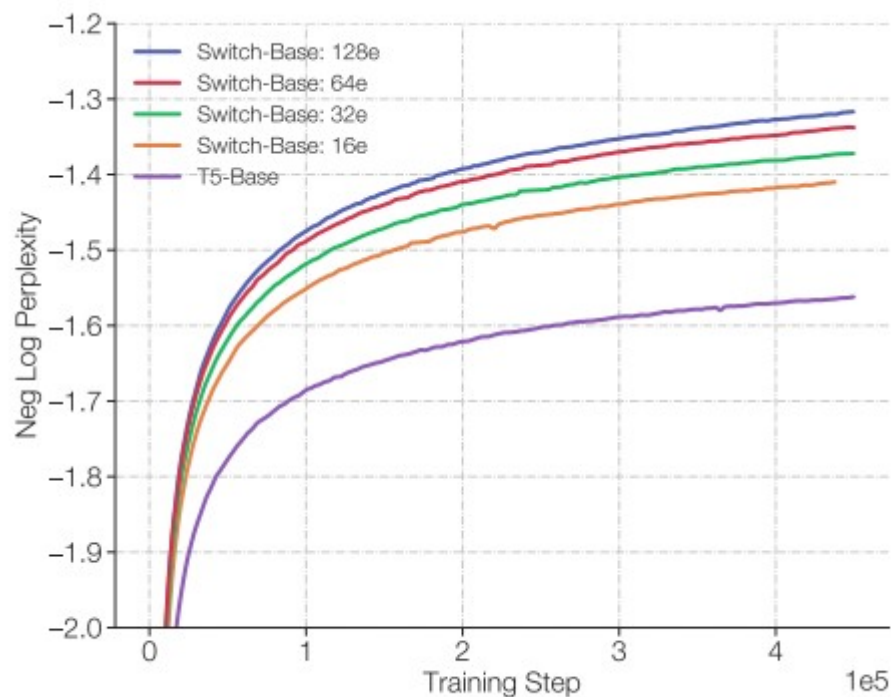
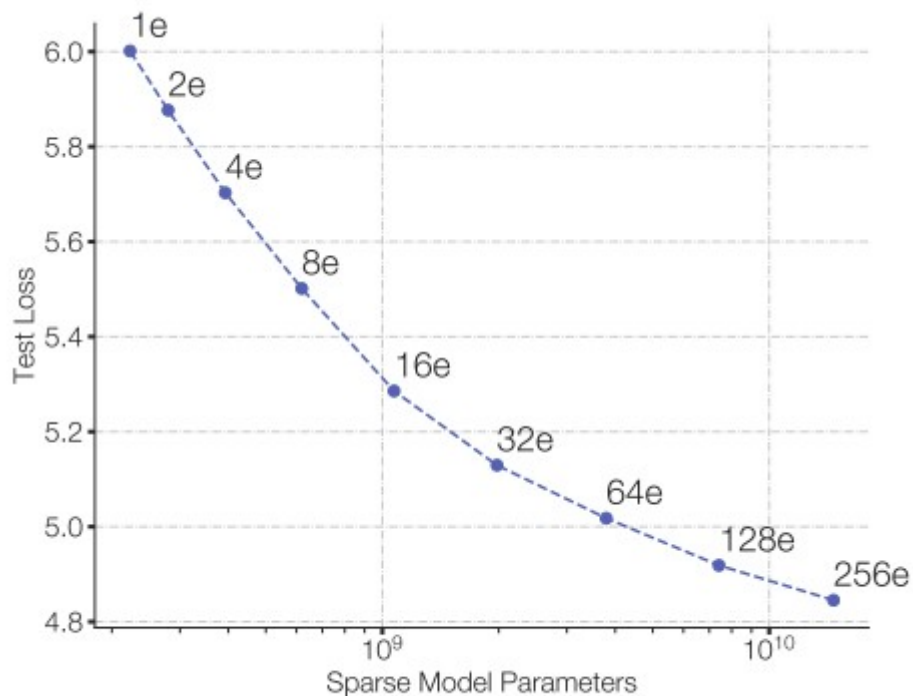
- **Experts:** Split across devices, each having their own unique parameters. Perform standard feed-forward computation.
- **Expert Capacity:** Batch size of each expert. Calculated as $(\text{tokens_per_batch} / \text{num_experts}) * \text{capacity_factor}$
- **Capacity Factor:** Used when calculating expert capacity. Expert capacity allows more buffer to help mitigate token overflow during routing.



MoE Variant: Switch Transformer

Switch: <https://arxiv.org/pdf/2101.03961.pdf>

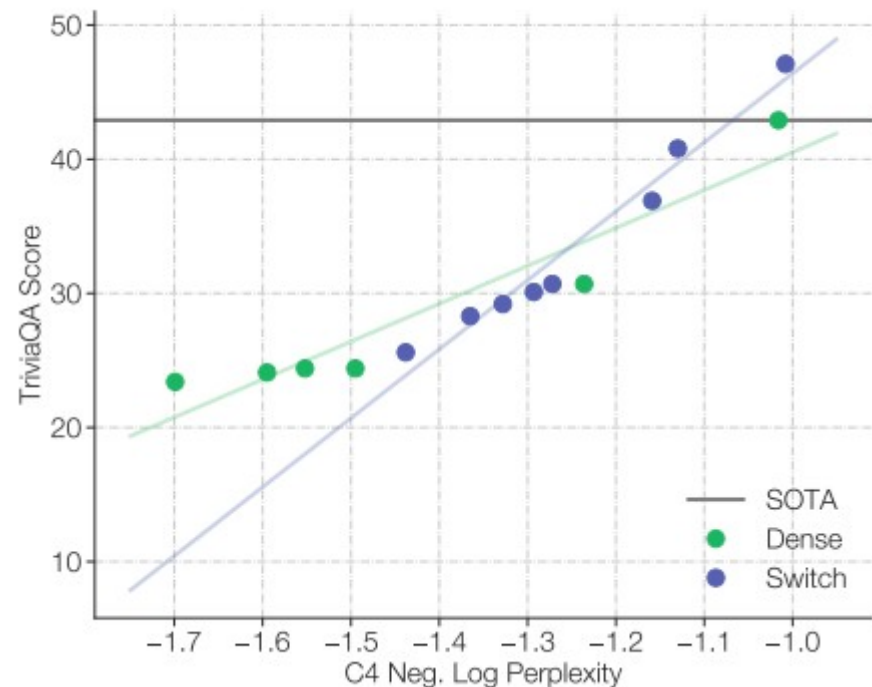
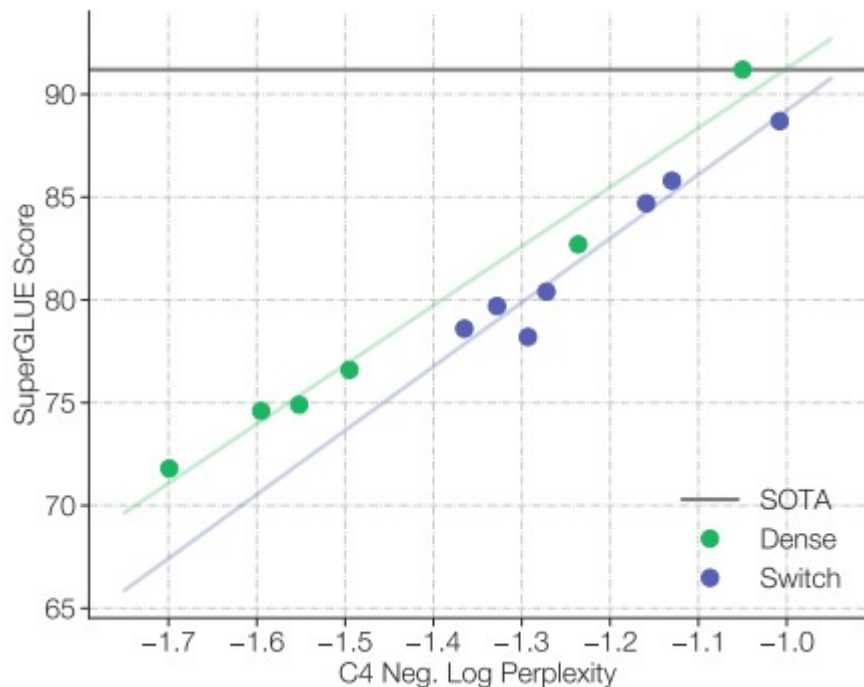
MLM pre-training objective [BERT-like]



MoE Variant: Switch Transformer

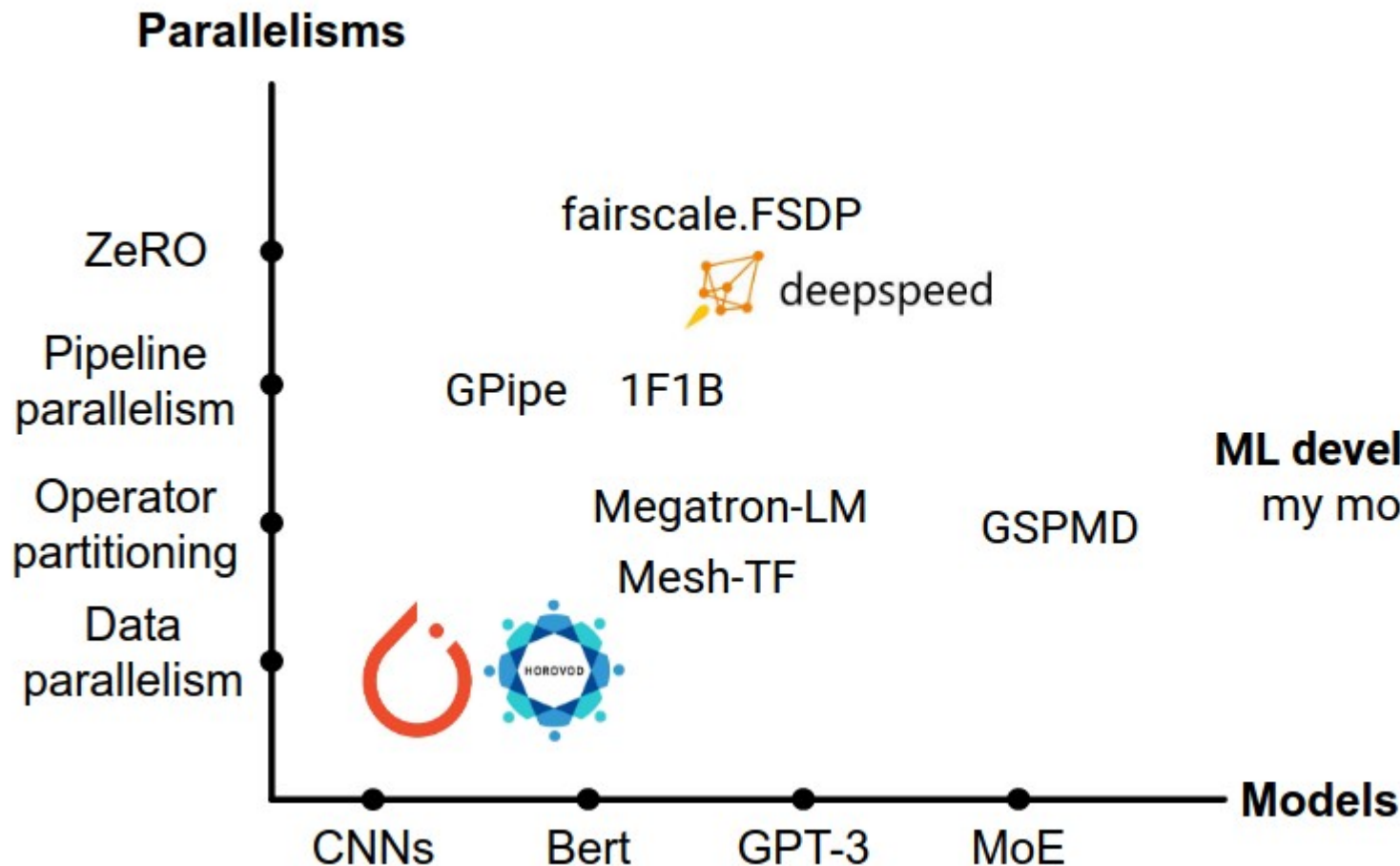
Switch: <https://arxiv.org/pdf/2101.03961.pdf>

Pre-training vs downstream quality



Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>



ML developer: which one is for my model and my cluster?

Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

Classic view

Data parallelism

Model parallelism

New view (this tutorial)

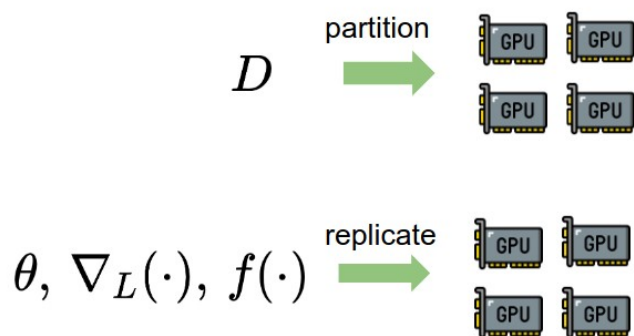
Inter-op parallelism

Intra-op parallelism

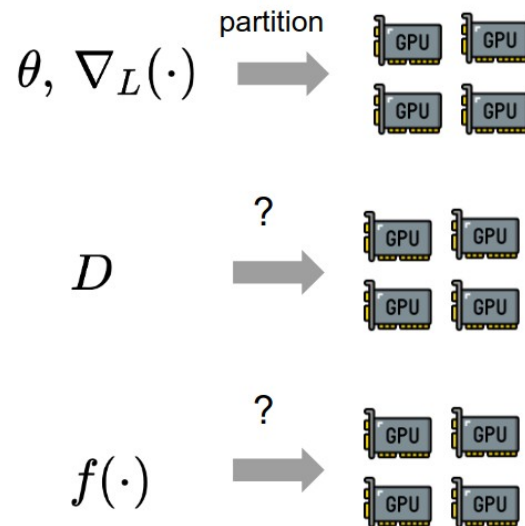
Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

Data parallelism



Model parallelism



$$\theta^{(t+1)} = f(\theta^{(t)}, \nabla_L(\theta^{(t)}, D^{(t)}))$$




Diagram illustrating the components of the equation:

- $\theta^{(t+1)}$: parameter
- f : weight update (sgd, adam, etc.)
- ∇_L : model (CNN, GPT, etc.)
- $D^{(t)}$: data



Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

Data and model parallelism

- Two pillars: **data** and **model**.
-  “Data parallelism” is general and precise.
-  “Model parallelism” is vague.
-  The view creates ambiguity for methods that neither partitions data nor the model computation.

New: Inter-op and Intra-op parallelism.

- Two pillars: **computational graph** and **device cluster**
-  This view is based on their computing characteristics.
-  This view facilitates the development of new parallelism methods.

Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

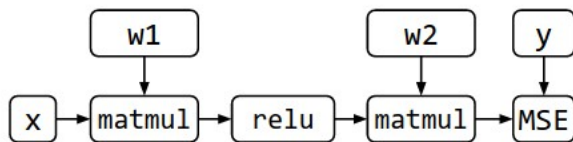
$$\theta^{(t+1)} = f(\theta^{(t)}, \nabla_L(\theta^{(t)}, D^{(t)}))$$

$$L = \text{MSE}(w_2 \cdot \text{ReLU}(w_1 x), y) \quad \theta = \{w_1, w_2\}, D = \{(x, y)\}$$

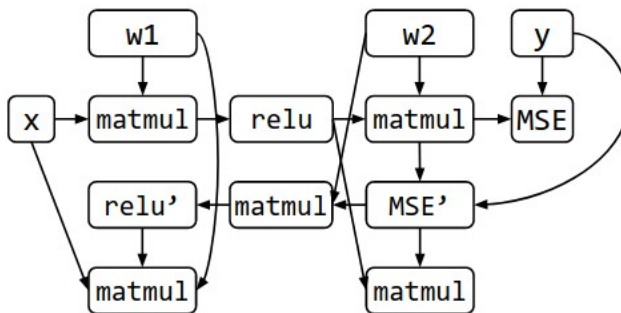
$$f(\theta, \nabla_L) = \theta - \nabla_L$$

☐ Operator / its output tensor \longrightarrow Data flowing direction

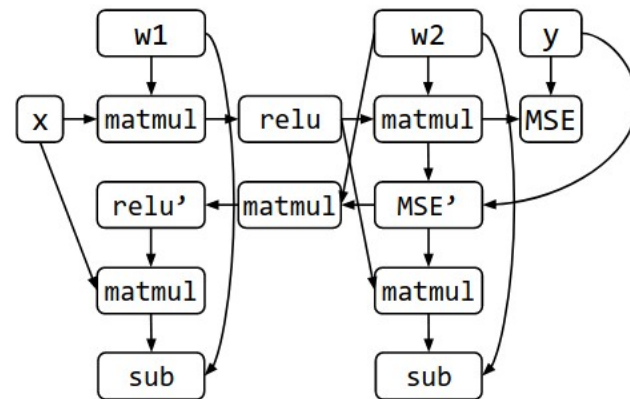
Forward



+Backward



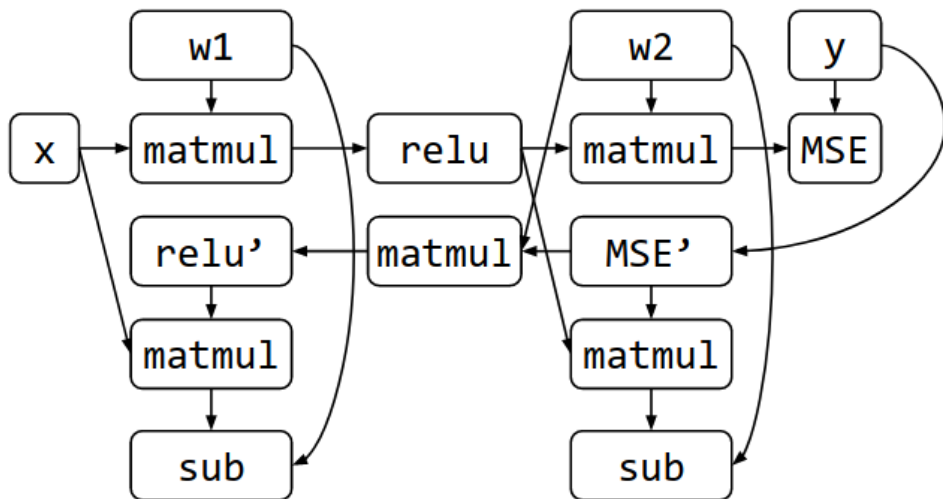
+Weight update



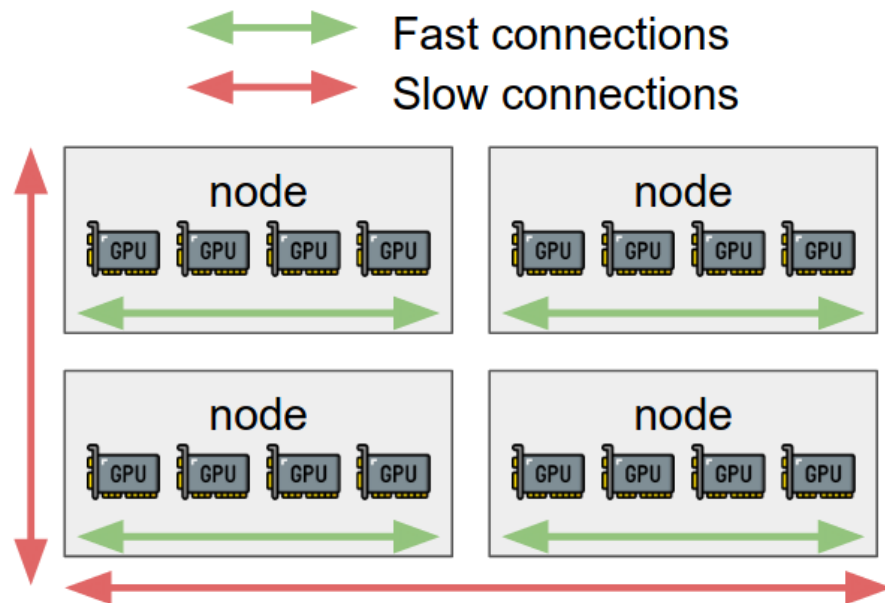
Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

Compute graph



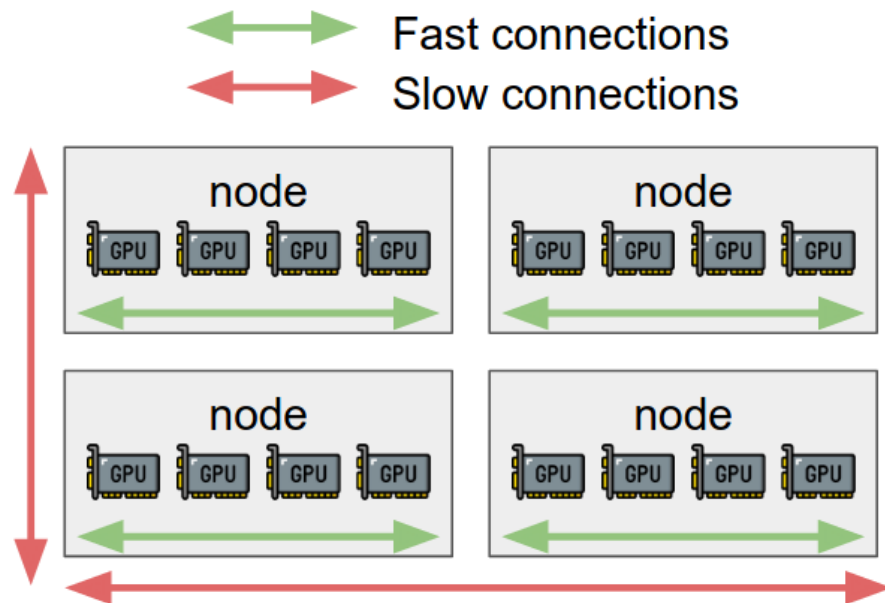
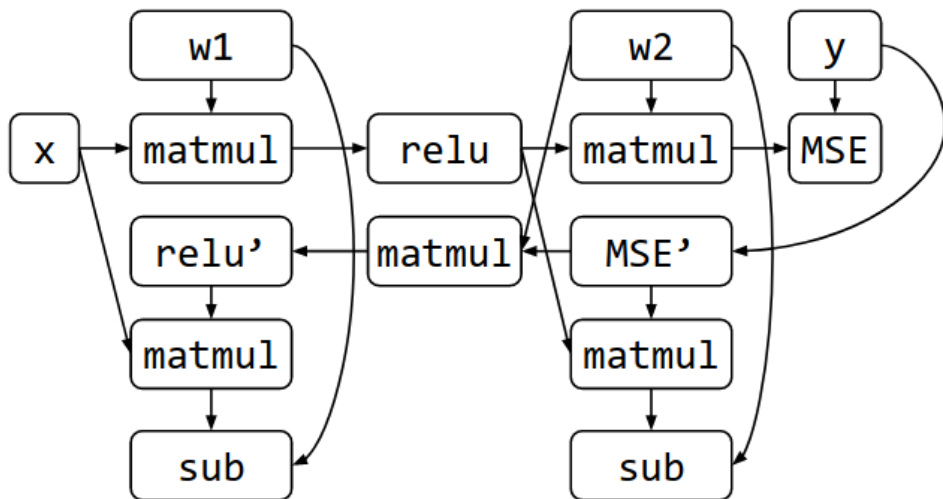
Device cluster



Automated parallelism

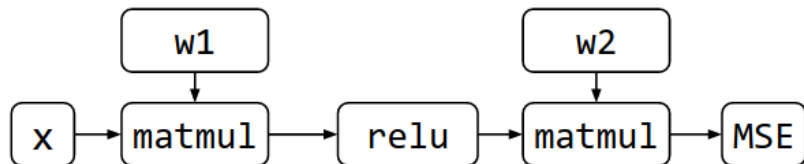
source: <https://sites.google.com/view/icml-2022-big-model>

Q: How to partition the graph on the device cluster?



Automated parallelism

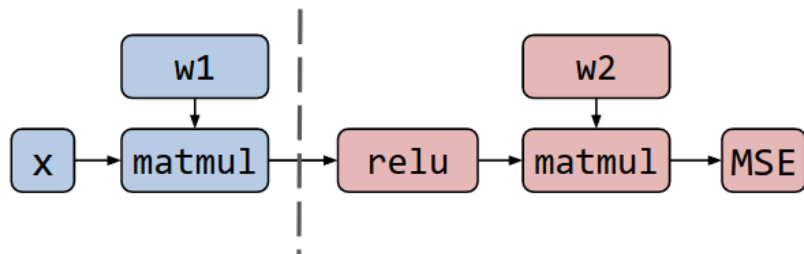
source: <https://sites.google.com/view/icml-2022-big-model>



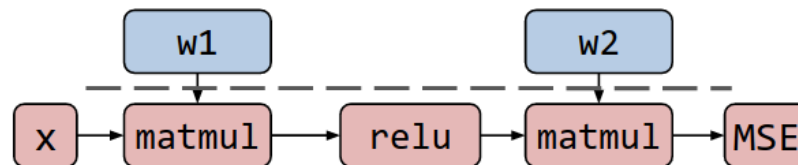
Device 1

Device 2

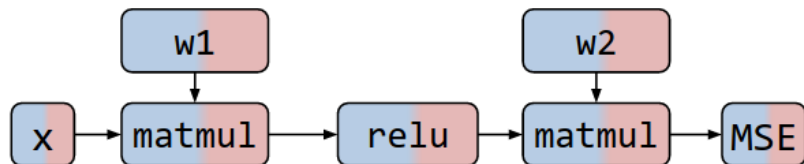
Strategy 1



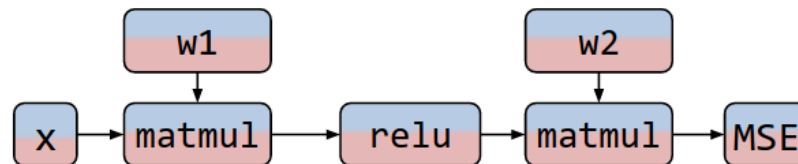
Strategy 2



Strategy 3

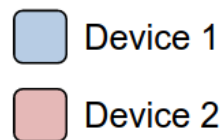
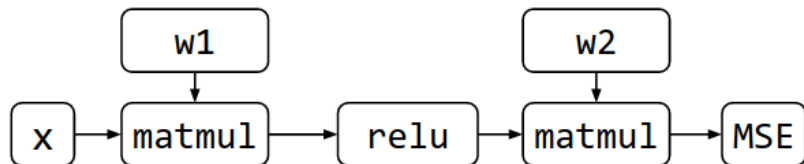


Strategy 4



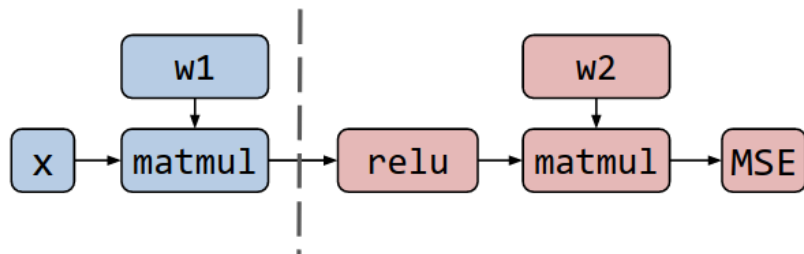
Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>

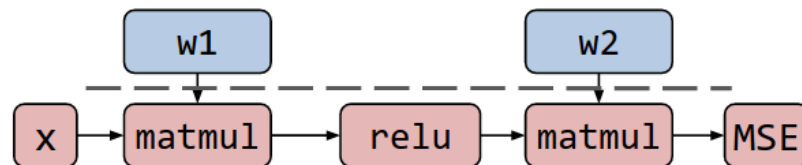


Q: have you seen S1/2/3/4 before?

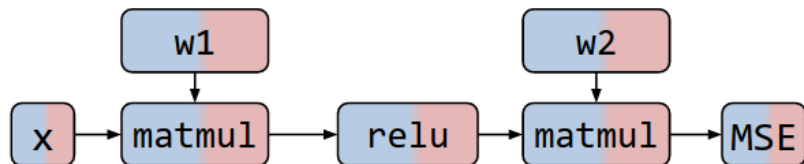
Strategy 1



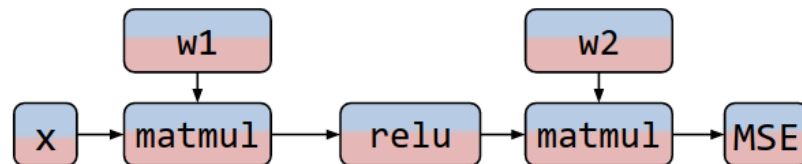
Strategy 2



Strategy 3

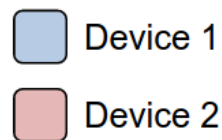
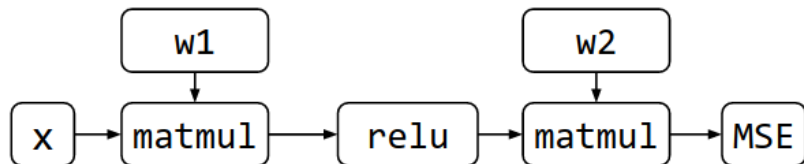


Strategy 4

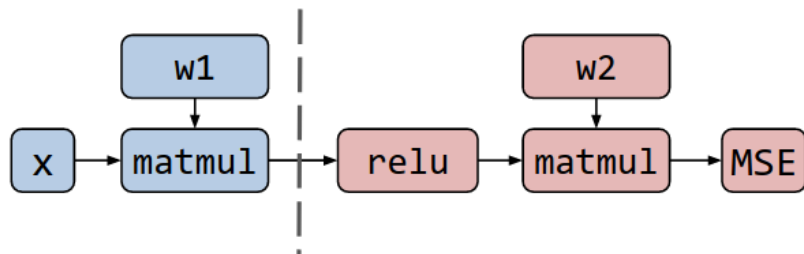


Automated parallelism

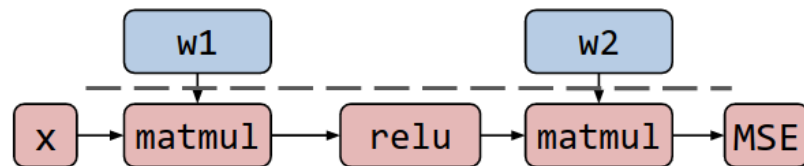
source: <https://sites.google.com/view/icml-2022-big-model>



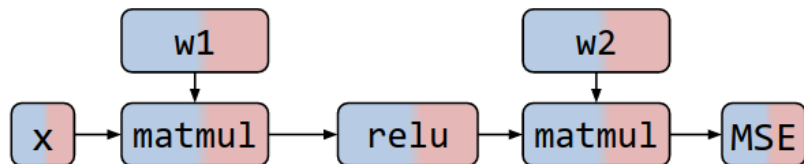
Pipeline MP



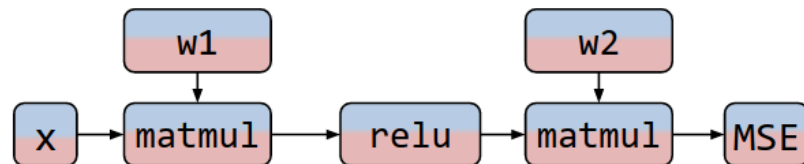
DP with offloading or PS



Tensor-parallel v1

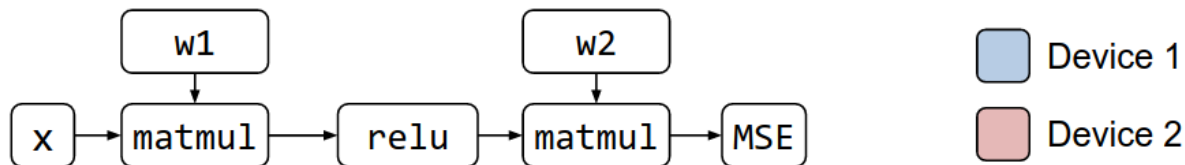


Tensor-parallel v2

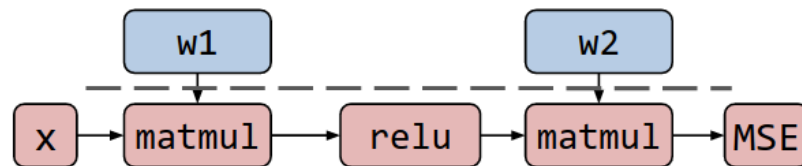
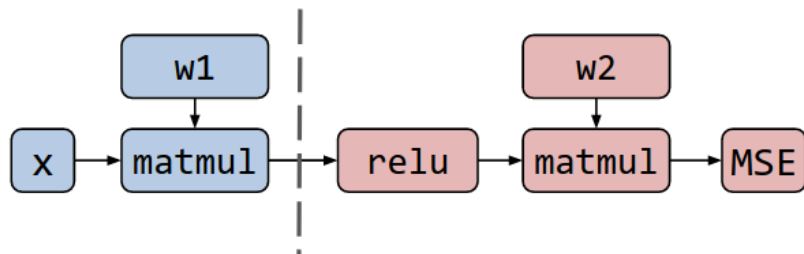


Automated parallelism

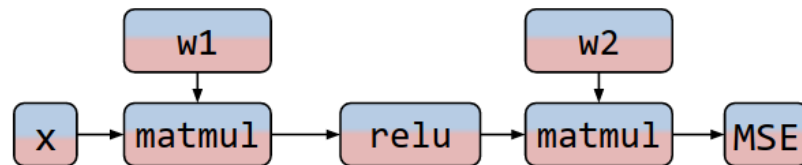
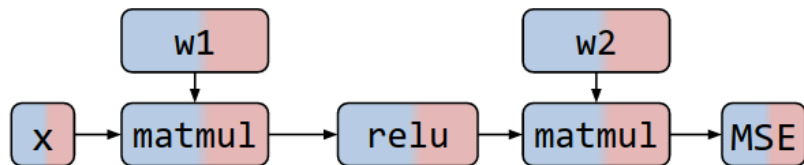
source: <https://sites.google.com/view/icml-2022-big-model>



~~Pipeline MP~~ **Inter-op parallelism** ~~DP with offloading or PS~~

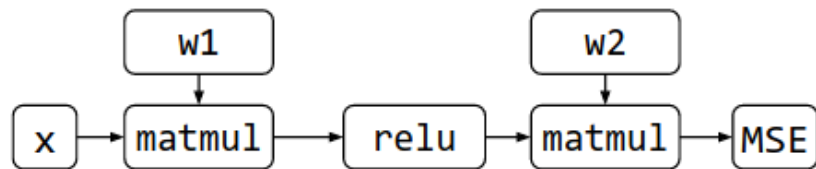


~~Tensor parallel v1~~ **Intra-op parallelism** ~~Tensor parallel v2~~



Automated parallelism

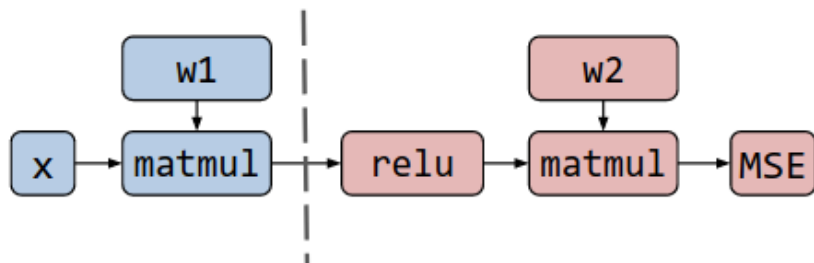
source: <https://sites.google.com/view/icml-2022-big-model>



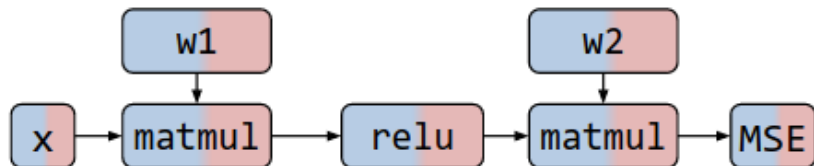
Device 1

Device 2

Inter-op parallelism



Intra-op parallelism

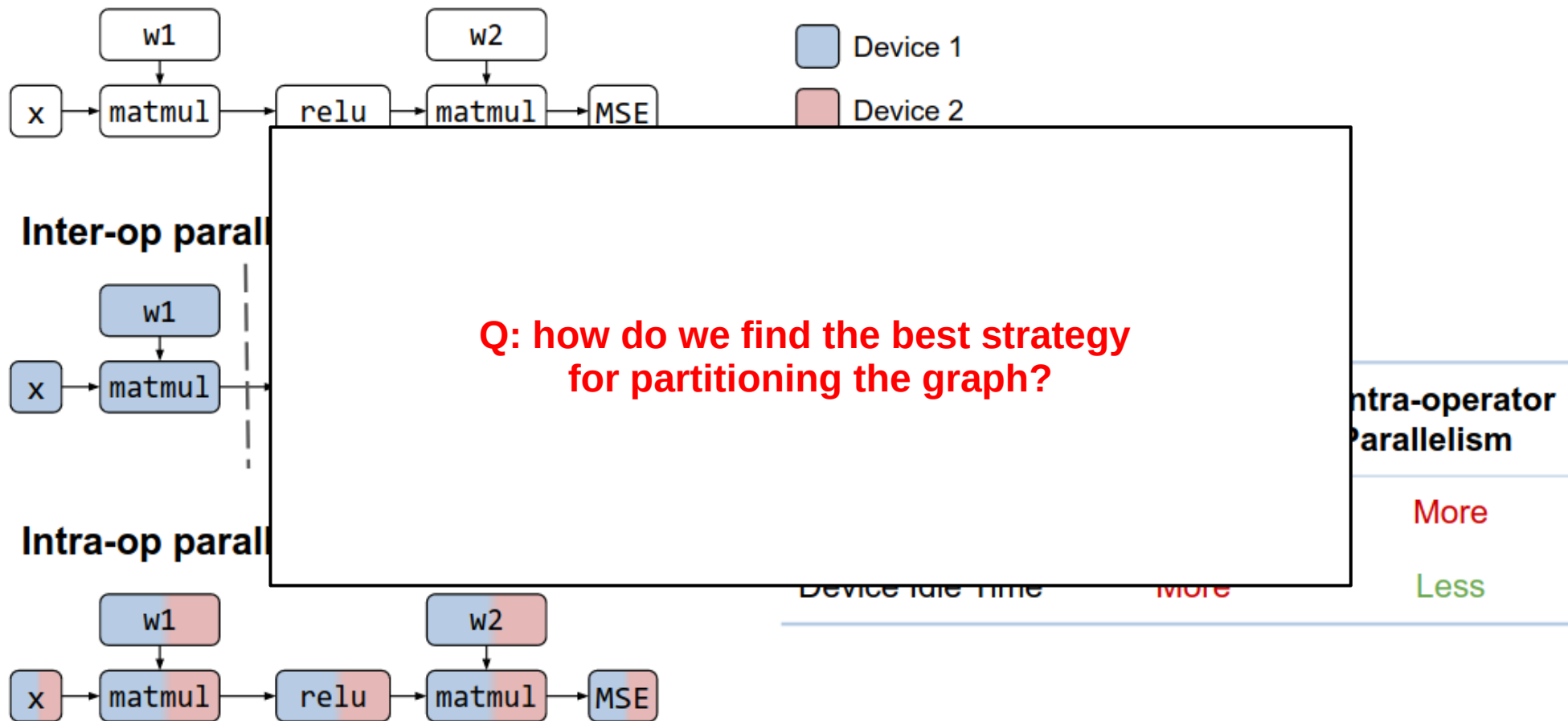


Trade-off

	Inter-operator Parallelism	Intra-operator Parallelism
Communication	Less	More
Device Idle Time	More	Less

Automated parallelism

source: <https://sites.google.com/view/icml-2022-big-model>



RL-based partitioning

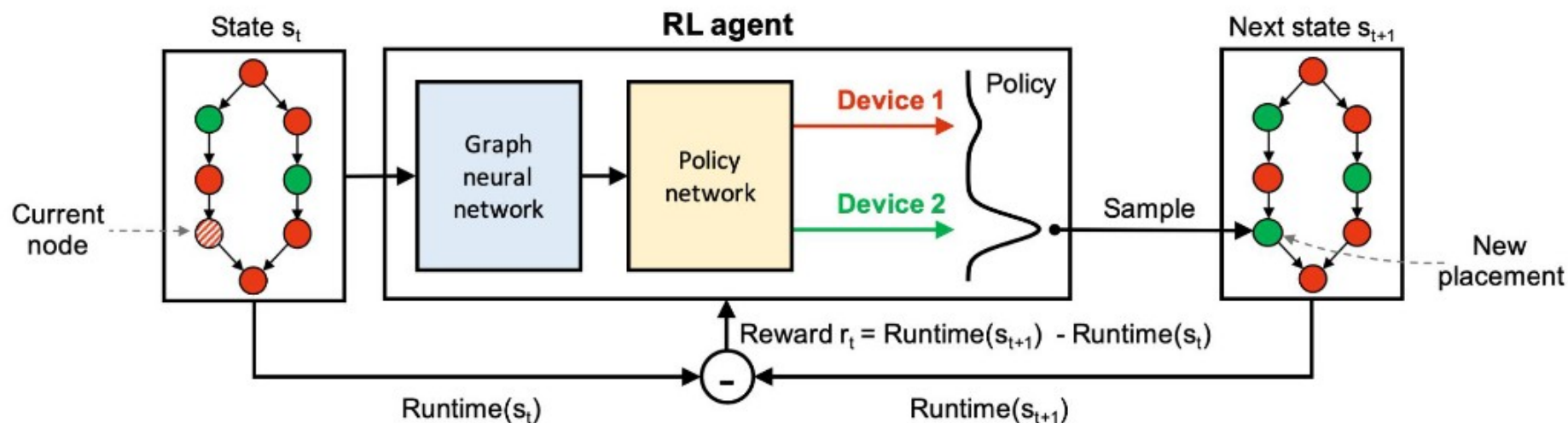
<https://people.csail.mit.edu/hongzi/content/publications/placeto-neurips19.pdf>

State: Device assignment plan for a computational graph.

Action: Modify the device assignment of a node.

Reward: Latency difference between the new and old placements.

Trained with **policy gradient** algorithm.



Optimization-based partitioning

<https://arxiv.org/abs/2006.16423>

Integer Linear Programming:

Variable: Decision variable vector for each operator, representing device assignment.

Minimize: Maximum finishing time of all operators.

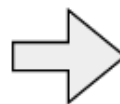
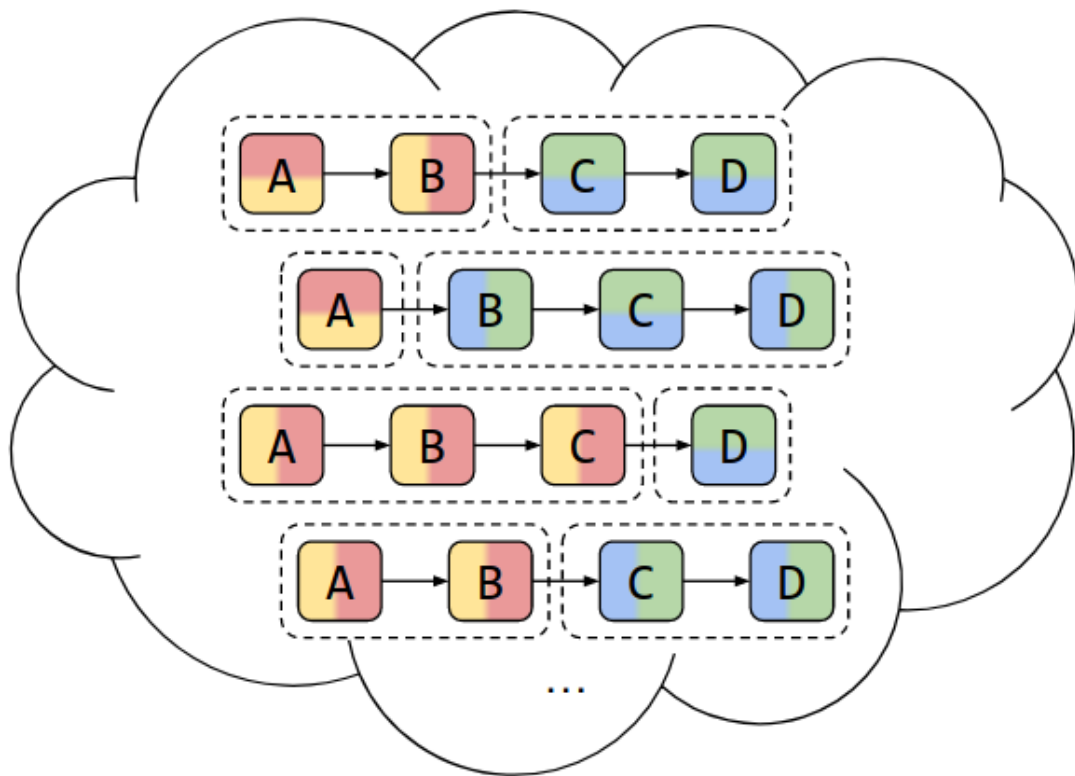
Constraint: Execution dependency & memory capacity of each device.

$$\begin{aligned} \min \quad & \text{TotalLatency} \\ \text{s.t.} \quad & \sum_{i=0}^k x_{vi} = 1 \\ & \text{subgraph } \{v \in V : x_{vi} = 1\} \text{ is contiguous} \\ & M \geq \sum_v m_v \cdot x_{vi} \\ & \text{CommIn}_{ui} \geq x_{vi} - x_{ui} \\ & \text{CommOut}_{ui} \geq x_{ui} - x_{vi} \\ & \text{TotalLatency} \geq \text{Latency}_v \\ & \text{SubgraphStart}_i \geq \text{Latency}_v \cdot \text{CommIn}_{vi} \\ & \text{SubgraphFinish}_i = \text{SubgraphStart}_i + \sum_v \text{CommIn}_{vi} \cdot c_v \\ & \quad + \sum_v x_{vi} \cdot p_v^{\text{acc}} + \sum_v \text{CommOut}_{vi} \cdot c_v \\ & \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} \\ & \text{Latency}_v \geq x_{v0} \cdot p_v^{\text{cpu}} + \text{Latency}_u \\ & \text{Latency}_v \geq x_{vi} \cdot \text{SubgraphFinish}_i \\ & x_{vi} \in \{0, 1\} \end{aligned}$$

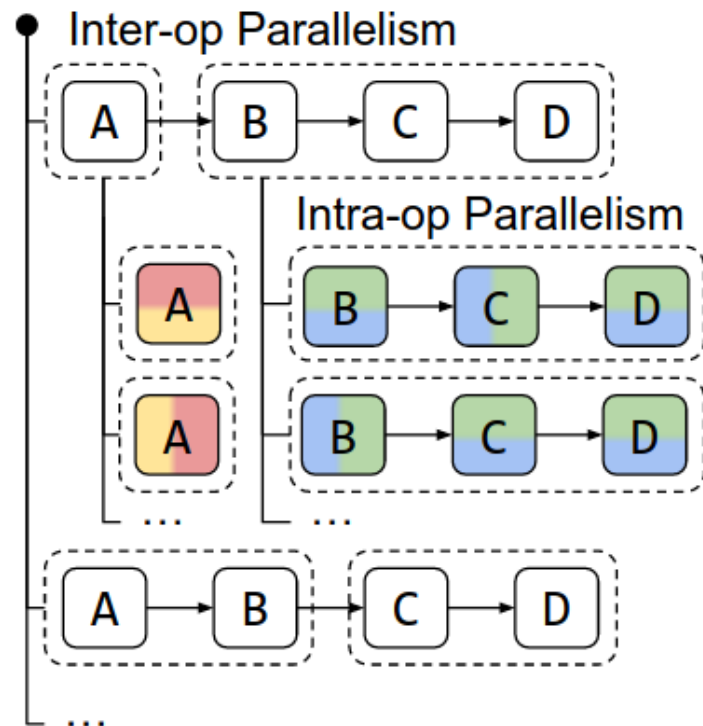
Alpa: optimization-based + reduced search space

<https://arxiv.org/abs/2201.12023>

Whole Search Space

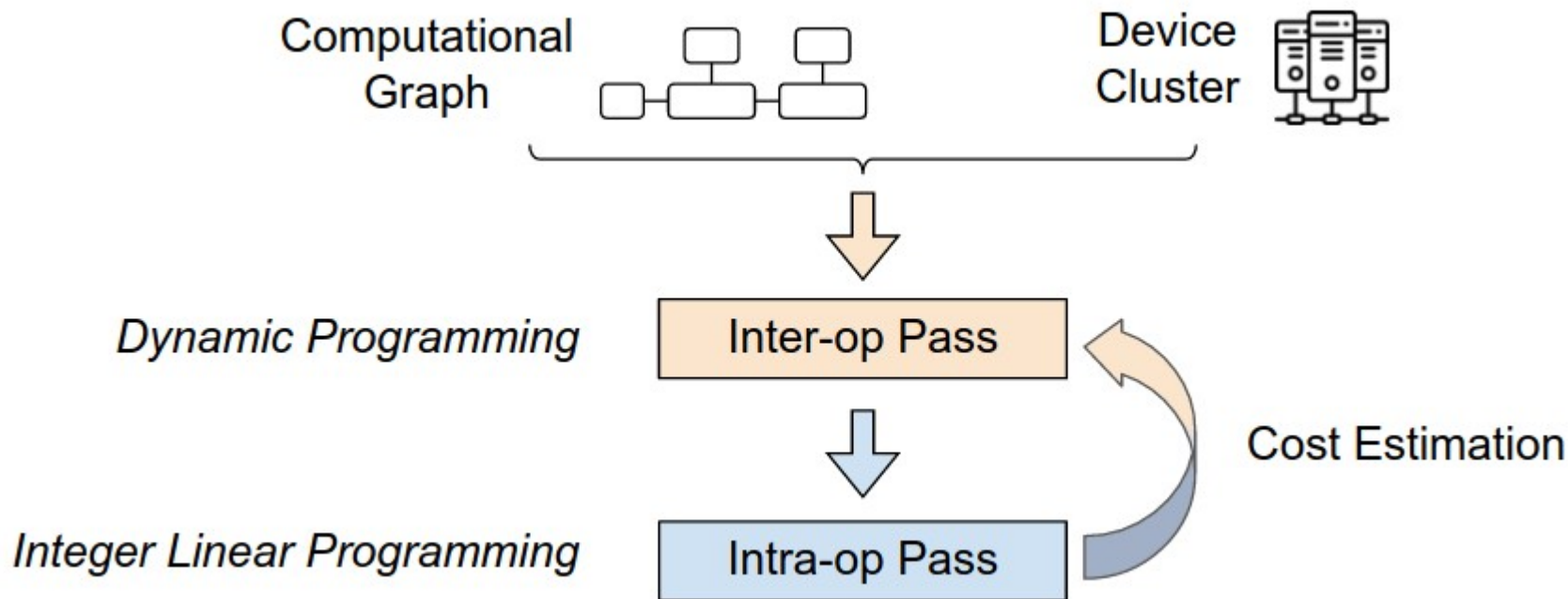


Alpa Hierarchical Space



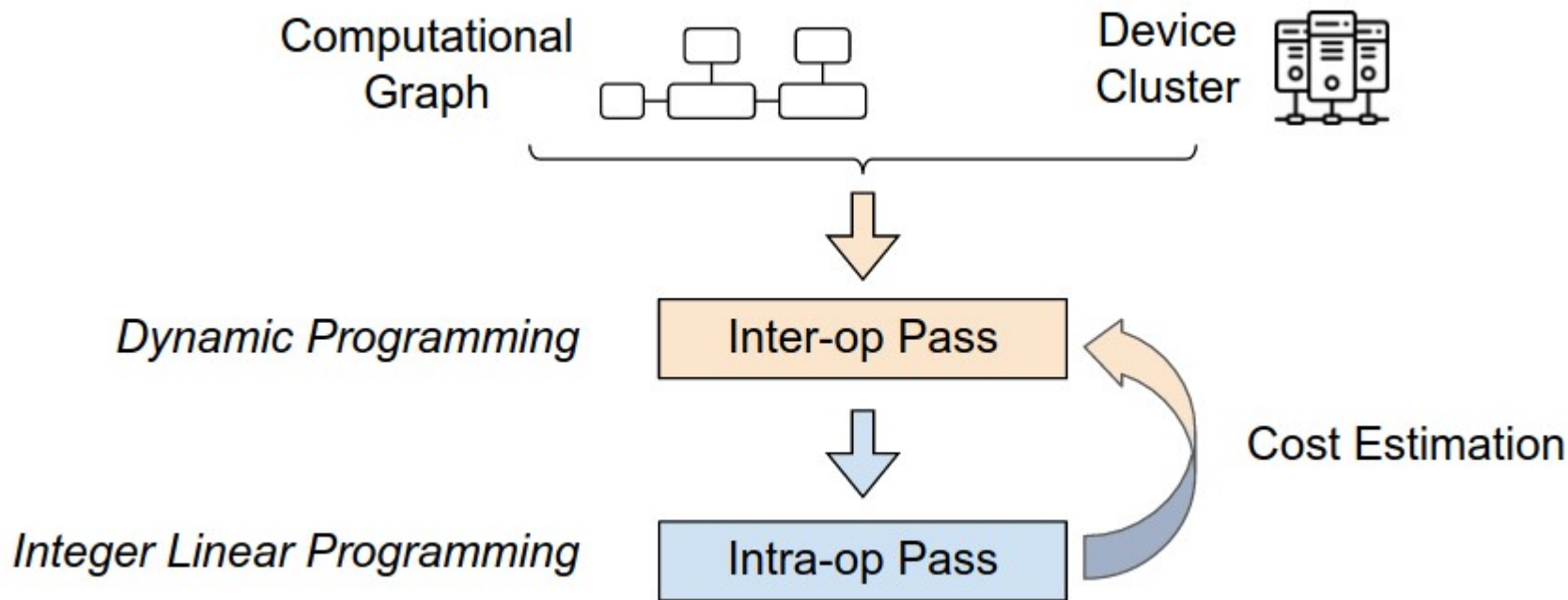
Alpa: optimization-based + reduced search space

<https://arxiv.org/abs/2201.12023>



Alpa: optimization-based + reduced search space

<https://arxiv.org/abs/2201.12023>



More details of each pass:

<https://sites.google.com/view/icml-2022-big-model>

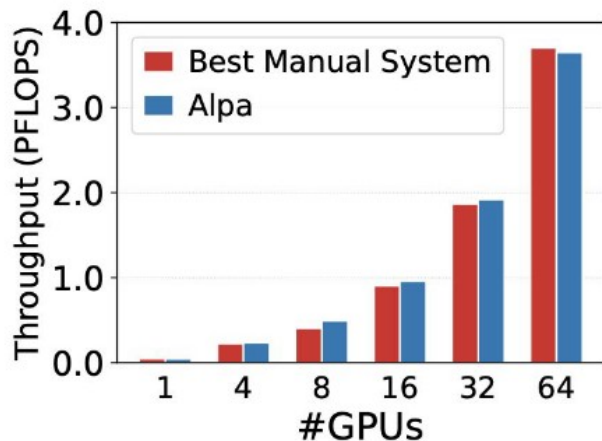
Alpa: optimization-based + reduced search space

<https://arxiv.org/abs/2201.12023>

Not the first algorithm for auto-parallelism...
but the first one that is usable* (* - most of the time)

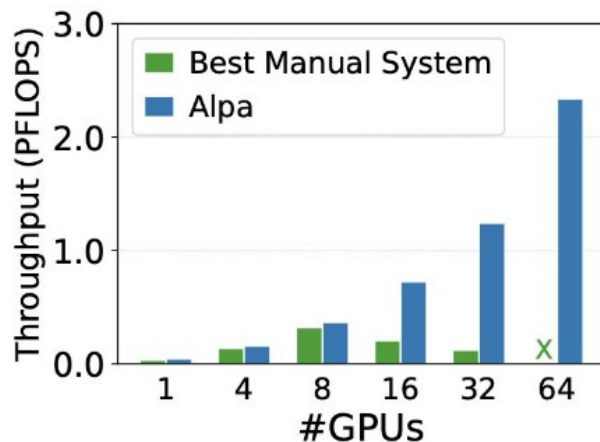
(benchmarks on AWS V100)

GPT (up to 39B)



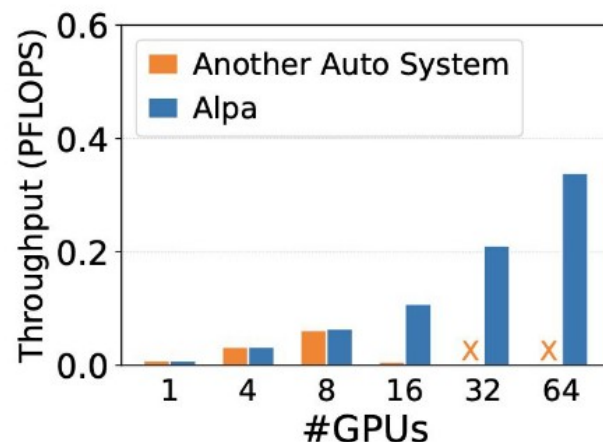
Match specialized manual systems.

GShard MoE (up to 70B)



Outperform the manual baseline by up to 8x.

Wide-ResNet (up to 13B)



Generalize to models without manual plans.

Alpa: optimization-based + reduced search space

<https://arxiv.org/abs/2201.12023>

Not the first algorithm for auto-parallelism...
but the first one that is usable* (* - most of the time)

```
# Define the training step. The body of this function is the same as the  
# ``train_step`` above. The only difference is to decorate it with  
# ``alpa.parallelize``.
```

@alpa.parallelize **auto best strategy**

```
def alpa_train_step(state, batch):  
    def loss_func(params):  
        out = state.apply_fn(params, batch["x"])  
        loss = jnp.mean((out - batch["y"])**2)  
        return loss works in jax  
  
    grads = jax.grad(loss_func)(state.params)  
    new_state = state.apply_gradients(grads=grads)  
    return new_state
```

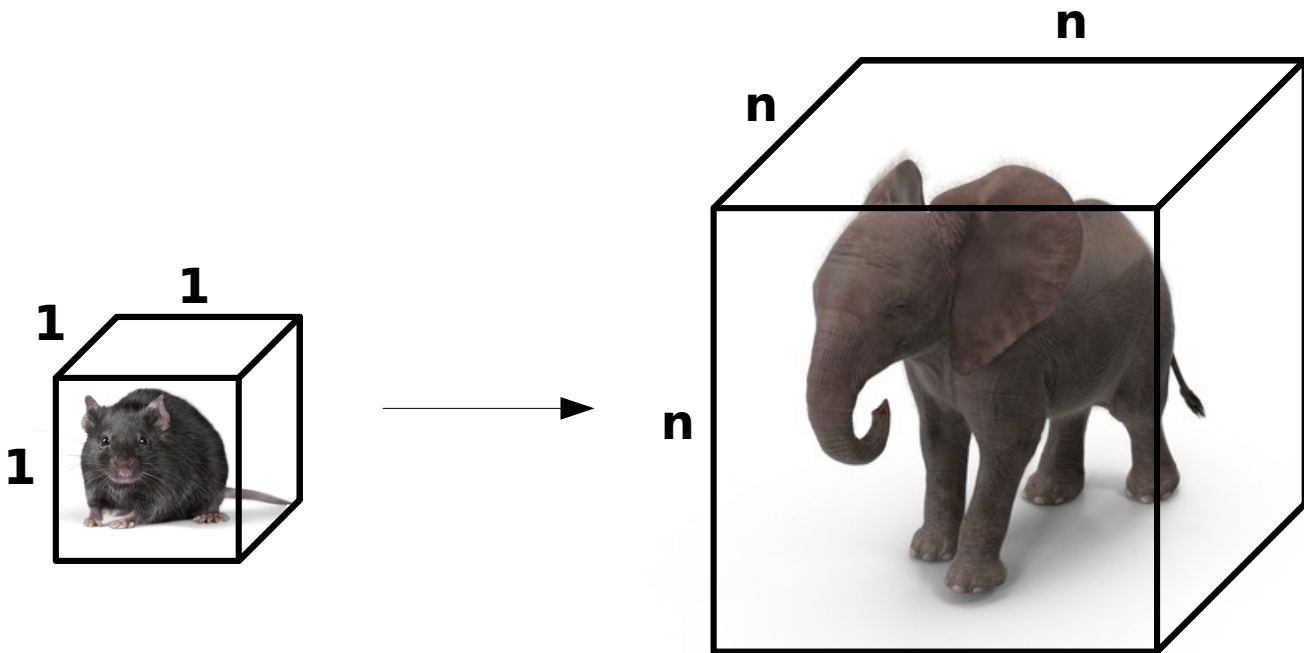
```
# Test correctness
```

```
actual_state = alpa_train_step(state, batch)  
assert_allclose(expected_state.params, actual_state.params, atol=5e-3)
```

If we have time...
(if not, finish here)

The square-cube law of deep learning

Explainer: <https://www.youtube.com/watch?v=f7KSfjv4Oq0>

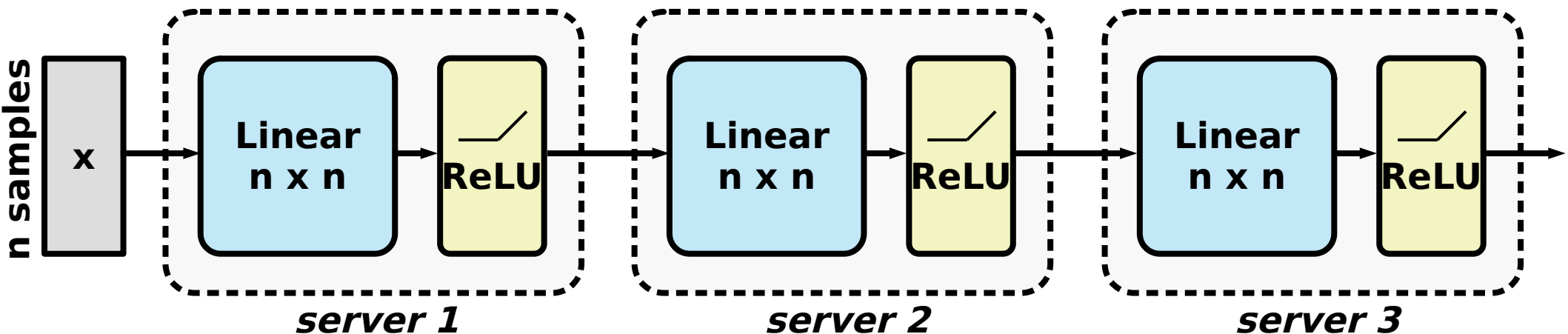


Surface area $O(n^2)$

Volume $O(n^3)$

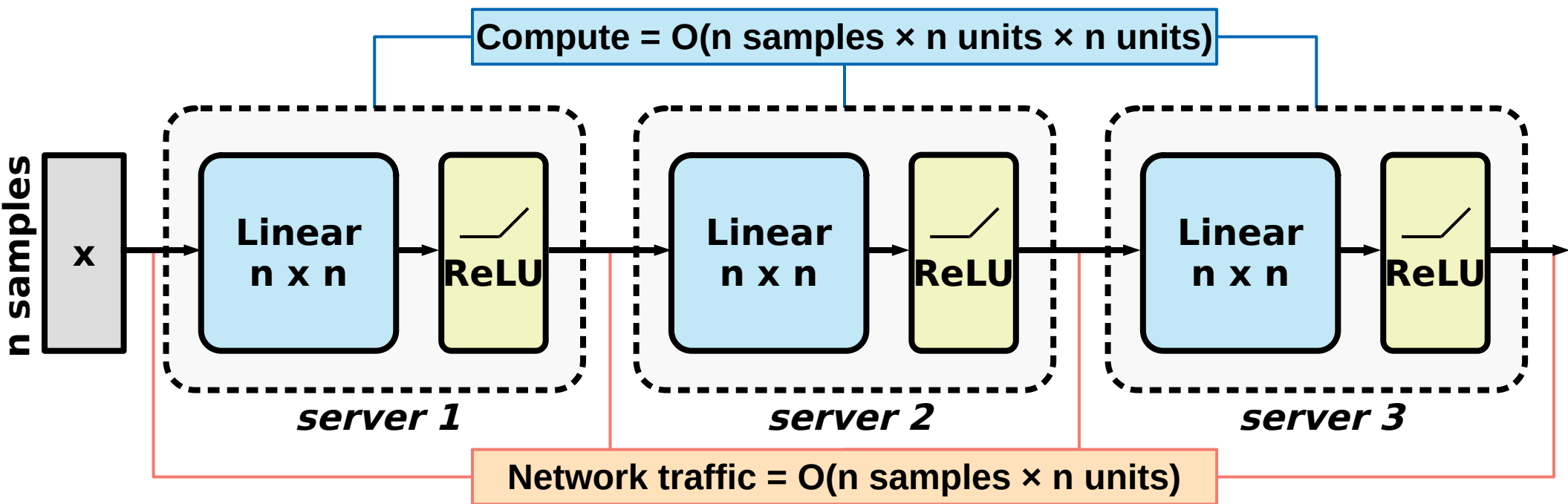
The square-cube law of deep learning

Explainer: <https://www.youtube.com/watch?v=f7KSfjv4Oq0>



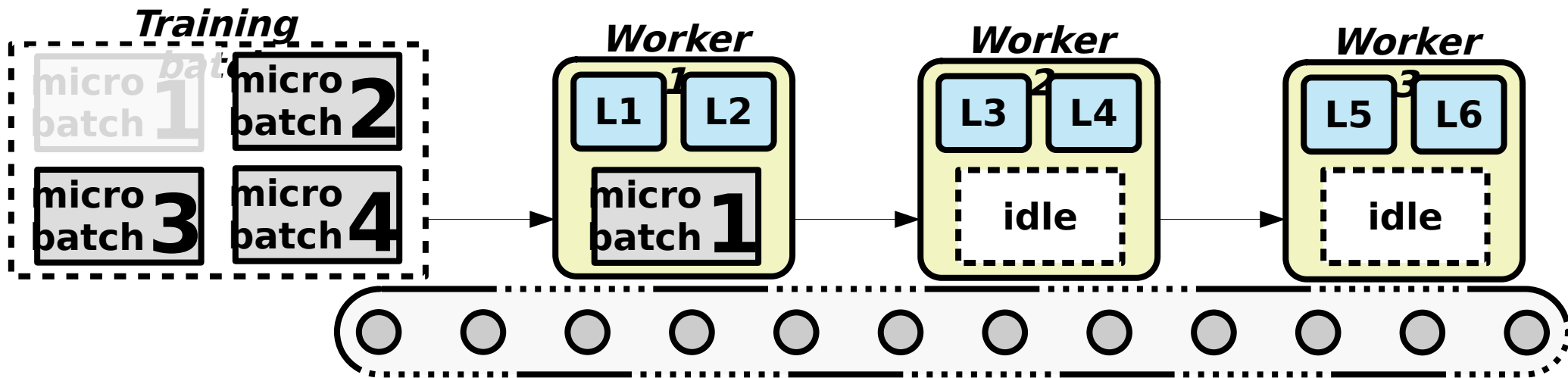
The square-cube law of deep learning

Explainer: <https://www.youtube.com/watch?v=f7KSfjv4Oq0>



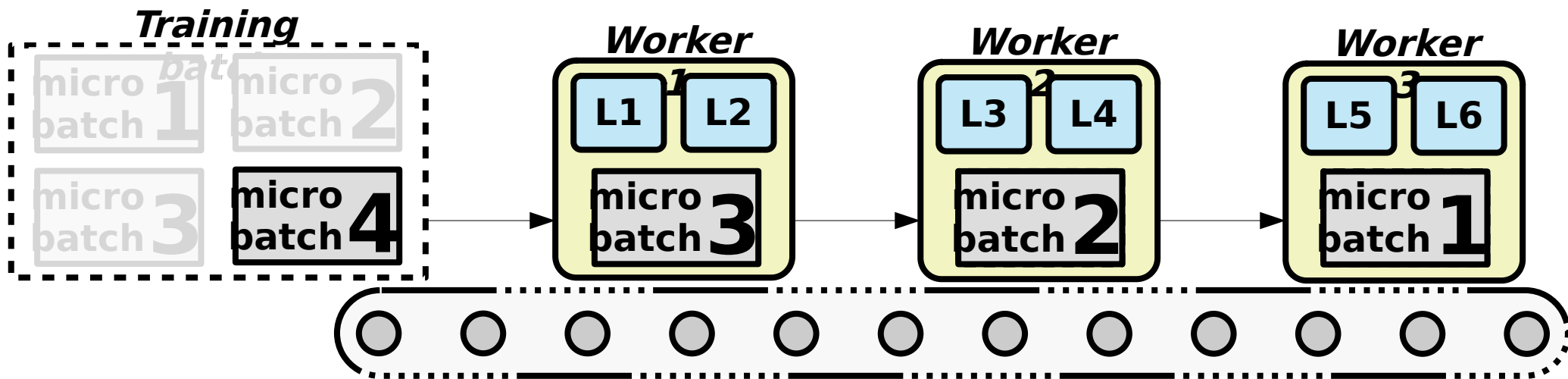
The square-cube law of deep learning

Explainer: <https://www.youtube.com/watch?v=f7KSfjv4Oq0>



The square-cube law of deep learning

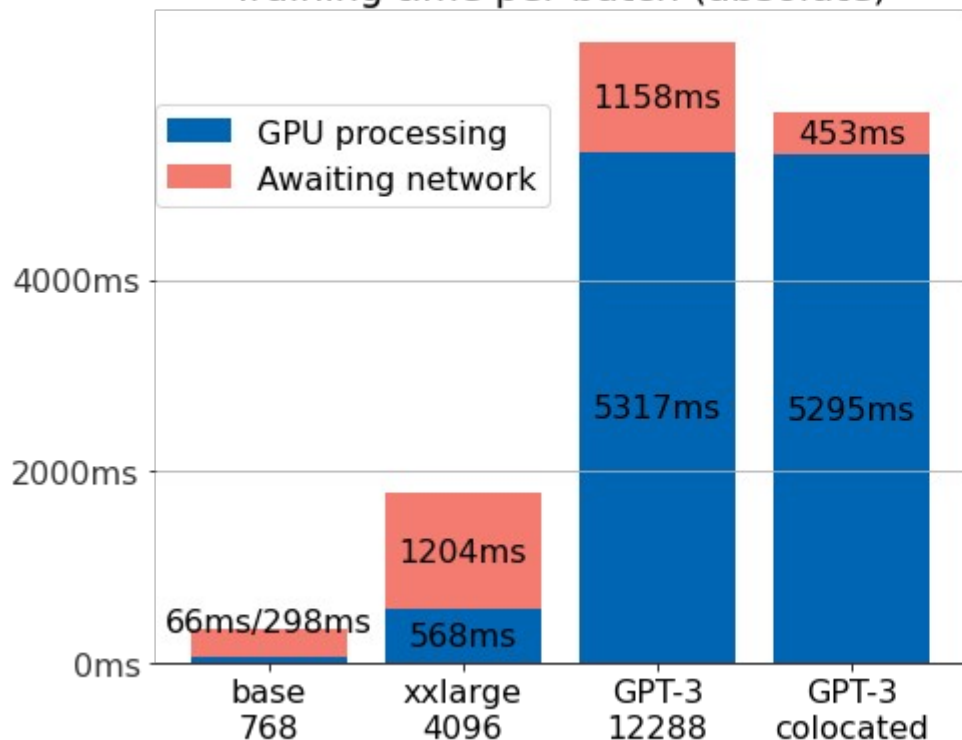
Explainer: <https://www.youtube.com/watch?v=f7KSfv4Oq0>



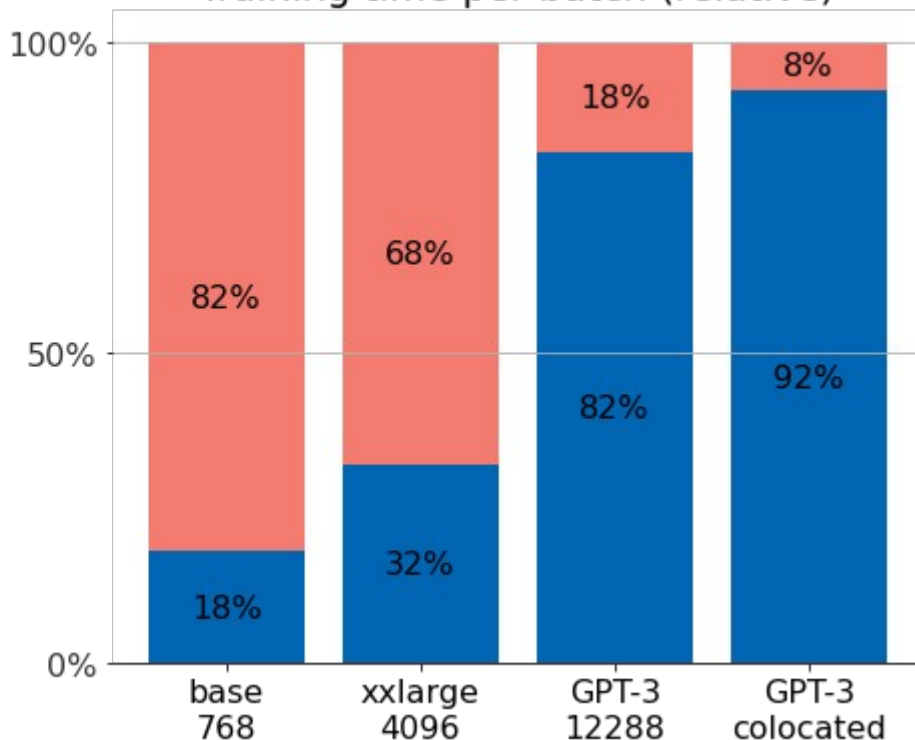
The square-cube law of deep learning

Explainer: <https://www.youtube.com/watch?v=f7KSfv4Oq0>

Training time per batch (absolute)



Training time per batch (relative)



Petals

<https://petals.ml>

TL;DR: you can run 100B+ models over the internet, BitTorrent style

```
from petals import DistributedBloomForCausalLM

model = DistributedBloomForCausalLM.from_pretrained("bigscience/bloom-petals", tuning_mode="ptune",
# Embeddings & prompts are on your device, BLOOM blocks are distributed across the Internet

inputs = tokenizer("A cat sat", return_tensors="pt")["input_ids"]
outputs = model.generate(inputs, max_new_tokens=5)
print(tokenizer.decode(outputs[0])) # A cat sat on a mat...

# Fine-tuning (updates only prompts or adapters hosted locally)
optimizer = torch.optim.AdamW(model.parameters())
for input_ids, labels in data_loader:
    outputs = model.forward(input_ids)
    loss = cross_entropy(outputs.logits, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```



That's all Folks!