

# Data-Parallel Deep Learning

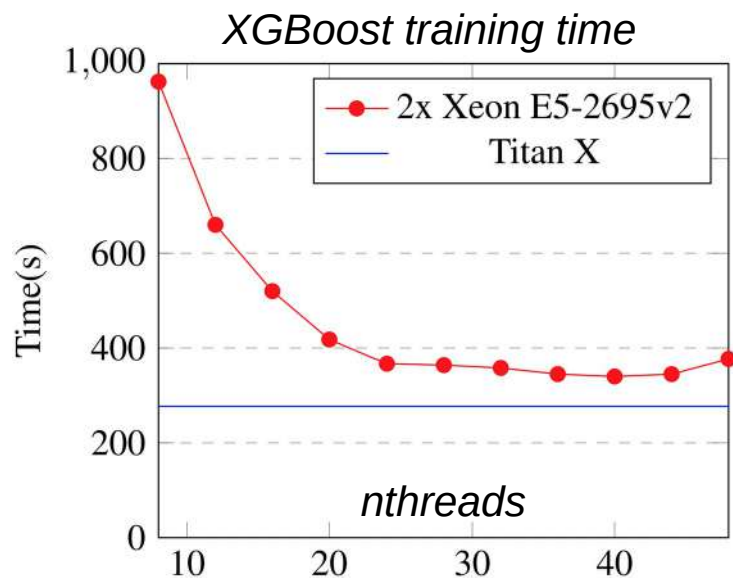
## Efficient DL, Episode 3, 2026

Yandex  
Research

LAMBDA

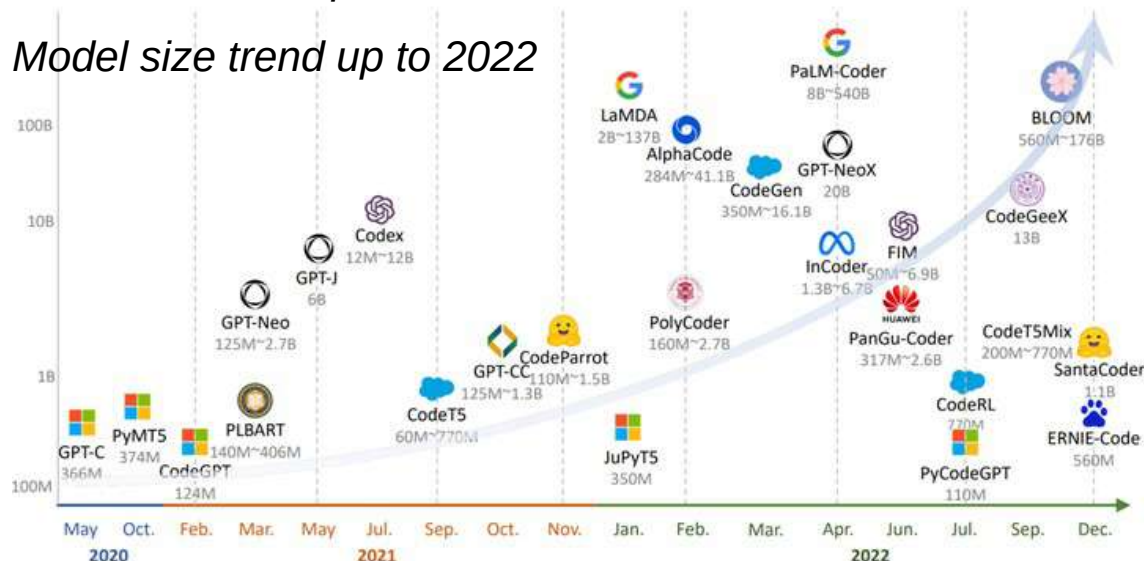


# Зачем это всё?



*parameters vs time*

*Model size trend up to 2022*

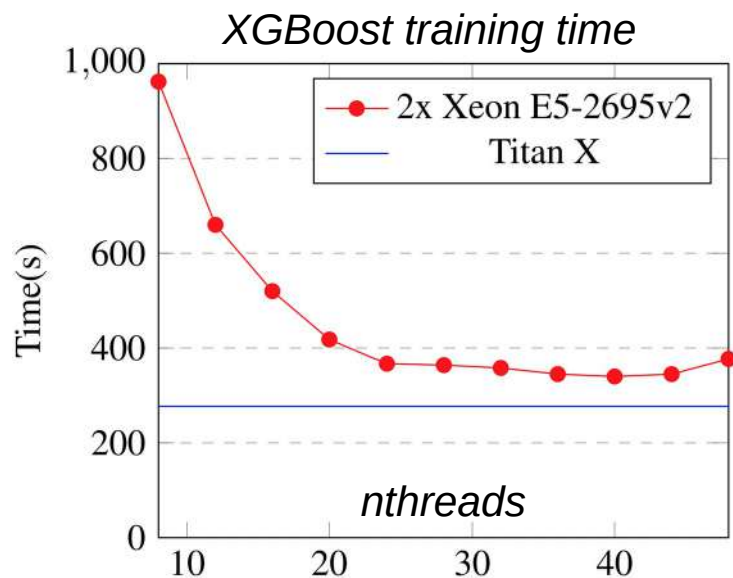


**BERT-Large Training Times on GPUs**

Time	System	Number of Nodes	Number of V100 GPUs
47 min	DGX SuperPOD	92 x DGX-2H	1,472
67 min	DGX SuperPOD	64 x DGX-2H	1,024
236 min	DGX SuperPOD	16 x DGX-2H	256

*(single GPU – over 2 weeks)*

# Зачем это всё?



**BERT-Large Training Times on GPUs**

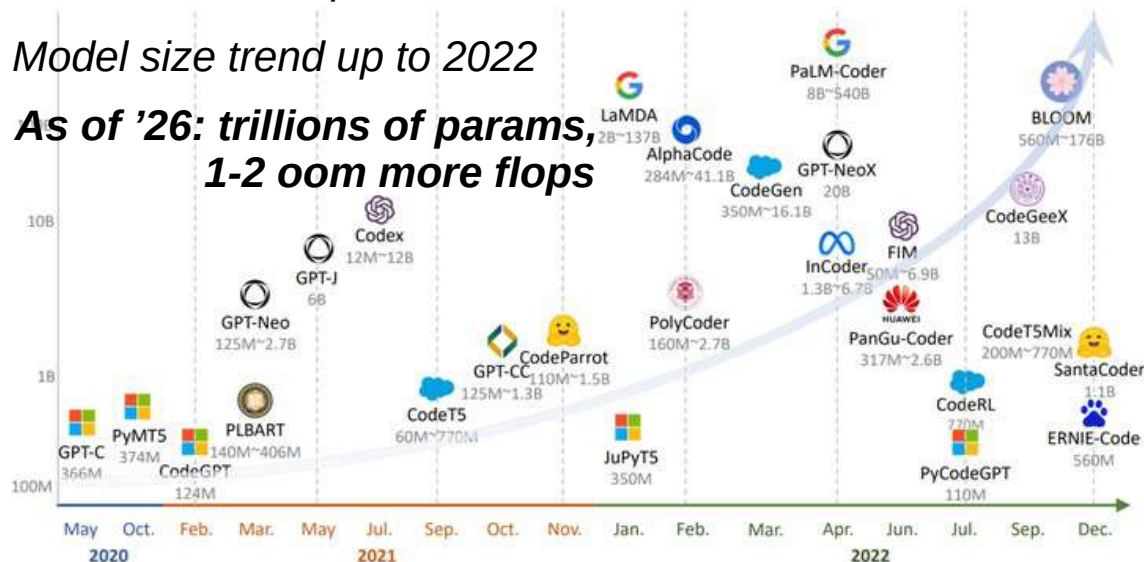
Time	System	Number of Nodes	Number of V100 GPUs
47 min	DGX SuperPOD	92 x DGX-2H	1,472
67 min	DGX SuperPOD	64 x DGX-2H	1,024
236 min	DGX SuperPOD	16 x DGX-2H	256

(single GPU – over 2 weeks)

*parameters vs time*

*Model size trend up to 2022*

**As of '26: trillions of params,  
1-2 oom more flops**



# Зачем мы тут?

Заставить много железяк вместе учить одну модель



# Зачем мы тут?

Заставить много железяк вместе учить одну модель

понять общие подходы

закодировать своими руками

на python / pytorch

# TL;DR our plan

*next few lectures*

- Data-parallel deep learning

*Train BERT-base on wikipedia in 20 minutes or less*

# TL;DR our plan

*next few lectures*

- Data-parallel deep learning

*Train BERT-base on wikipedia in 20 minutes or less*

- Model-parallel deep learning

*Fine-tune and deplov models with 100B+ parameters*

# TL;DR our plan

*next few lectures*

- Data-parallel deep learning

*Train BERT-base on wikipedia in 20 minutes or less*

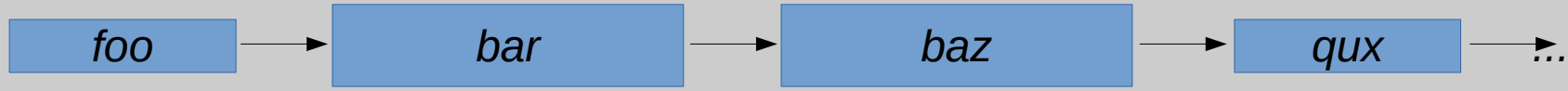
- Model-parallel deep learning

*Fine-tune and deploy models with 100B+ parameters  
like OPT, Llama, Qwen, DeepSeek R1, ...*

- Advanced techniques

*Sharding (FSDP), mixed / hybrid parallelism, practice*

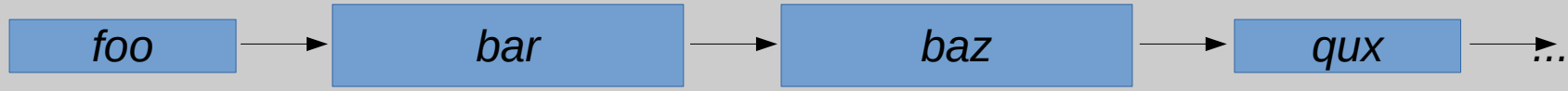
# Rules: Process



## Process:

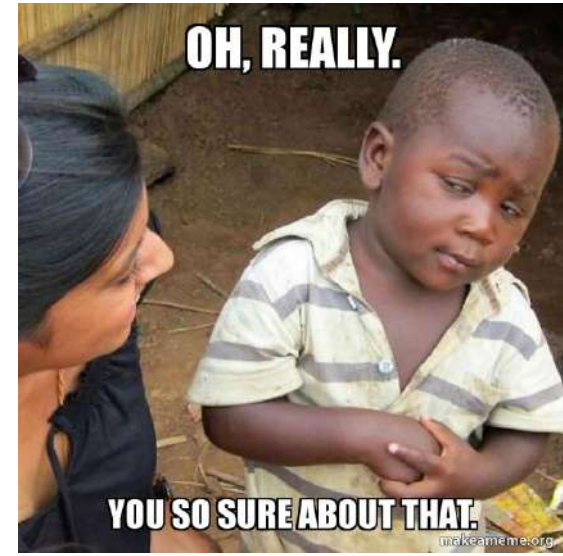
- Runs some code
- Has some memory
- No one else can access your memory

# Rules: Process

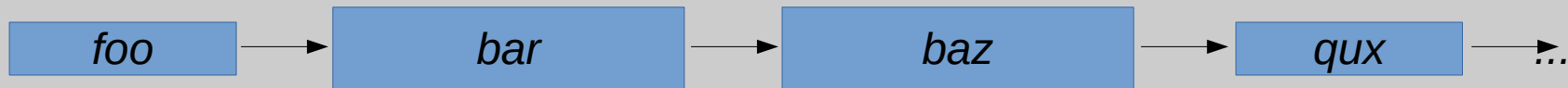


## Process:

- Runs some code
- Has some memory
- No one else can access your memory



# Rules: Process

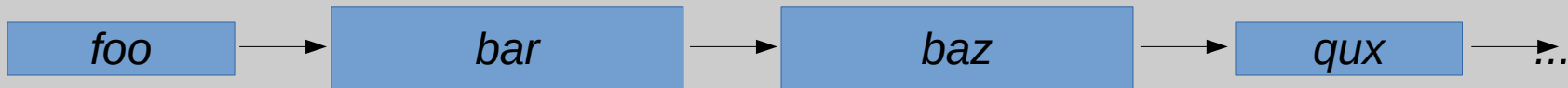


## Process:

- Runs some code
- Has some memory
- No one else can access your memory\*

\* – not if you use shared memory

# Rules: Process



## Process:

- Runs some code
- Has some memory
- No one else can access your memory<sup>\*†</sup>

<sup>\*</sup> – not if you use shared memory

<sup>†</sup> – superuser can still do that (os-dependent)

# Rules: Process



## Process:

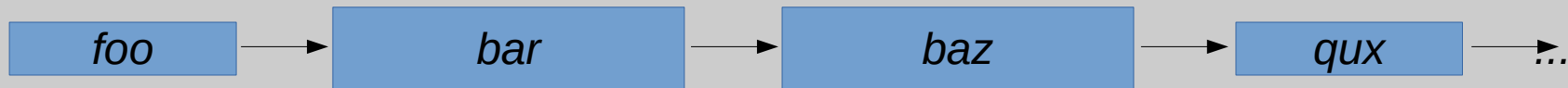
- Runs some code
- Has some memory
- No one else can access your memory<sup>\*†‡</sup>

\* – not if you use shared memory

† – superuser can still do that (os-dependent)

‡ – attacker can do that through spectre/meltdown/etc

# Rules: Process

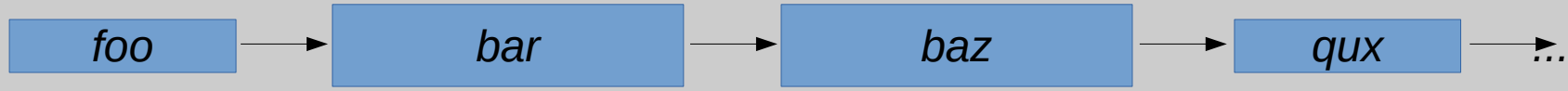


## Process:

- Runs some code
- Has some memory
- No one else **should** access your memory<sup>\*†‡</sup>

<sup>\*†‡</sup> – not relevant for this course

# Rules: Process



## Process:

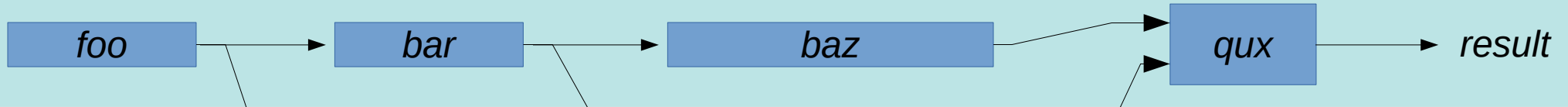
- Runs some code
- Has some memory
- No one else **should** access your memory<sup>\*†‡</sup>

<sup>\*†‡</sup> – not relevant for this course

**Q:** How do we make processes work together?

# Rules: Channel / Pipe

Process A:



Process B:



## Channel (pipe):

- Communication in  $O(\text{message size})$
- Asynchronous read/write

# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model



*Devices*

.cuda()

GPU1



# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model

$\theta$

batch

x

...

*Devices*

.cuda()

.cuda()

GPU1

$\theta$

x

⇒

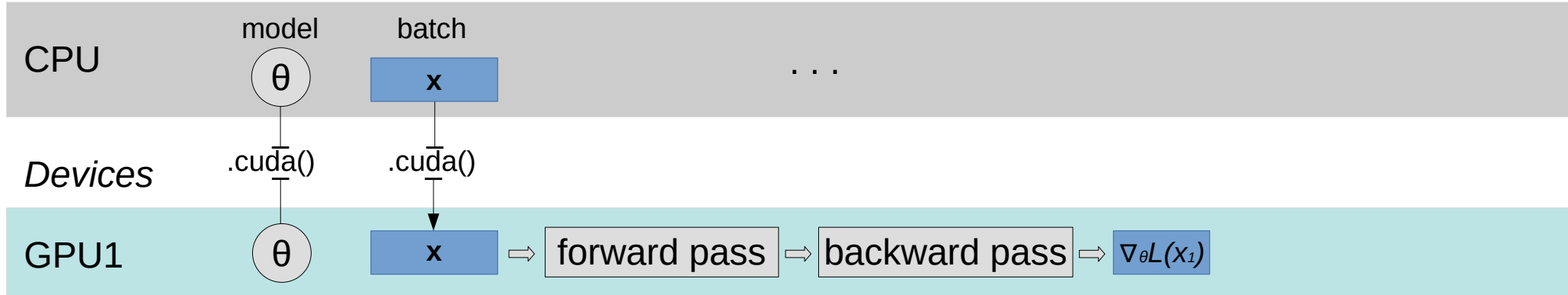
forward pass

⇒

backward pass

⇒

$\nabla_{\theta} L(x_1)$



# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model

$\theta$

batch

$x$

...

*prepare next batch*

*Devices*

.cuda()

.cuda()

new model

GPU1

$\theta$

$x$

$\Rightarrow$

forward pass

$\Rightarrow$

backward pass

$\Rightarrow$

$\nabla_{\theta} L(x_1)$

$\Rightarrow$

step

$\Rightarrow$

$\theta$

# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model

$\theta$

*Devices*

replicate

GPU1

$\theta$

GPU2

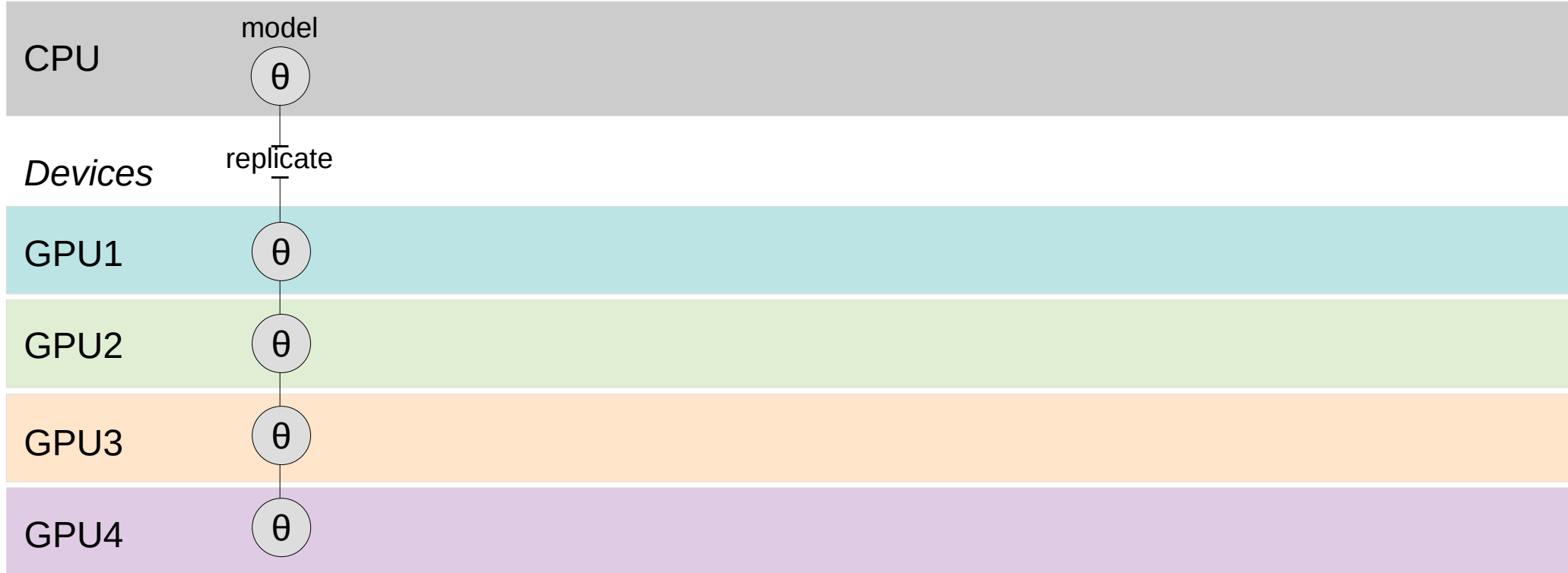
$\theta$

GPU3

$\theta$

GPU4

$\theta$



# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model

$\theta$

batch

$x_1$   $x_2$   $x_3$   $x_4$

*Devices*

replicate

scatter

GPU1

$\theta$

$x_1$

GPU2

$\theta$

$x_2$

GPU3

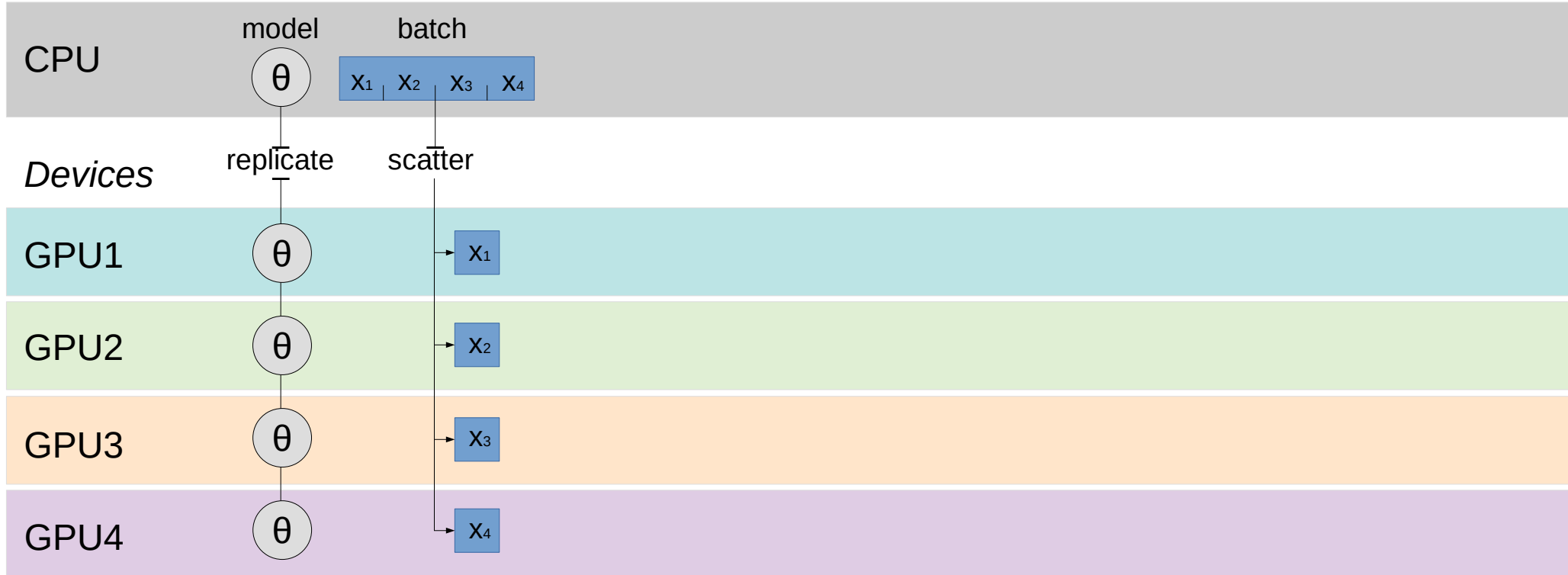
$\theta$

$x_3$

GPU4

$\theta$

$x_4$



# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

CPU

model

$\theta$

batch

$x_1$   $x_2$   $x_3$   $x_4$

...

*Devices*

replicate

scatter

GPU1

$\theta$

$x_1 \Rightarrow$  forward pass  $\Rightarrow$  backward pass  $\Rightarrow \nabla_{\theta} L(x_1)$

GPU2

$\theta$

$x_2 \Rightarrow$  forward pass  $\Rightarrow$  backward pass  $\Rightarrow \nabla_{\theta} L(x_2)$

GPU3

$\theta$

$x_3 \Rightarrow$  forward pass  $\Rightarrow$  backward pass  $\Rightarrow \nabla_{\theta} L(x_3)$

GPU4

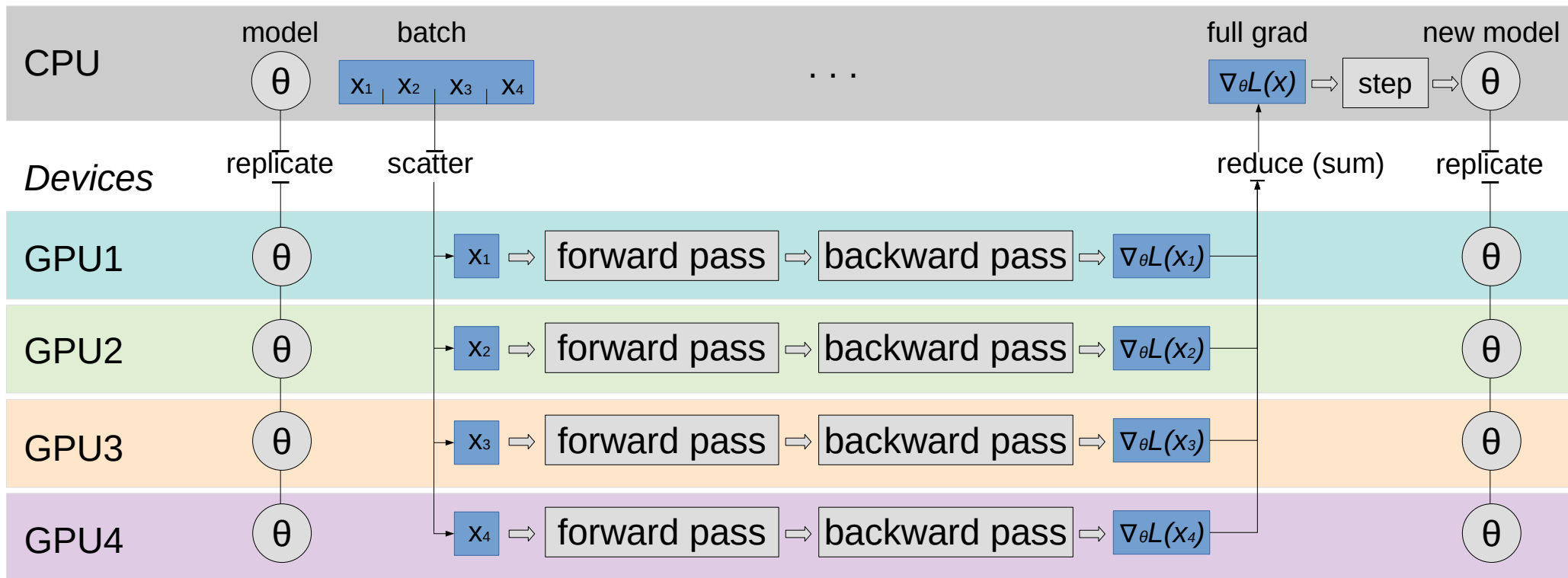
$\theta$

$x_4 \Rightarrow$  forward pass  $\Rightarrow$  backward pass  $\Rightarrow \nabla_{\theta} L(x_4)$

# Data-parallel training (naive)

[cs.cmu.edu/~muli/file/parameter\\_server\\_osdi14.pdf](https://cs.cmu.edu/~muli/file/parameter_server_osdi14.pdf)

*Host*

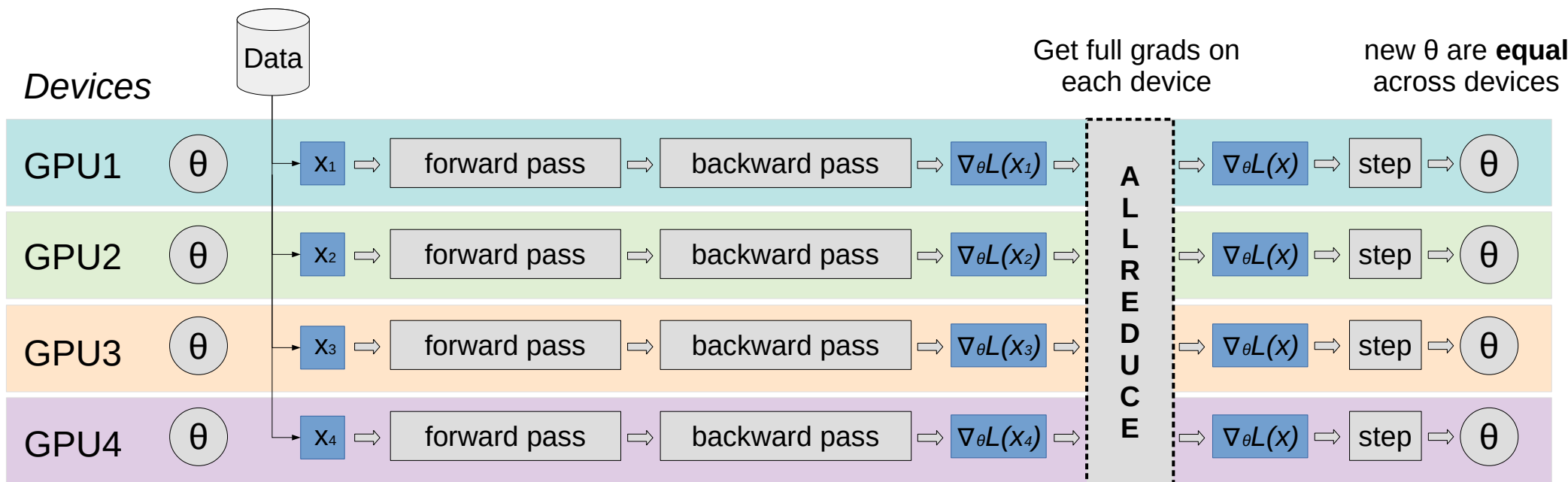


# All-Reduce data parallel

[arxiv.org/abs/1706.02677](https://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?

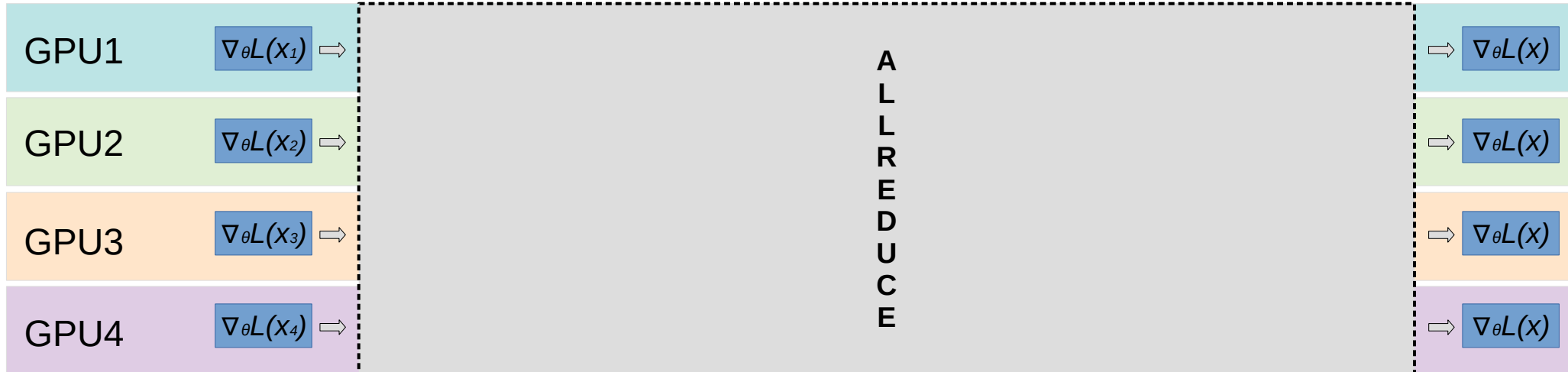


# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

*Devices*



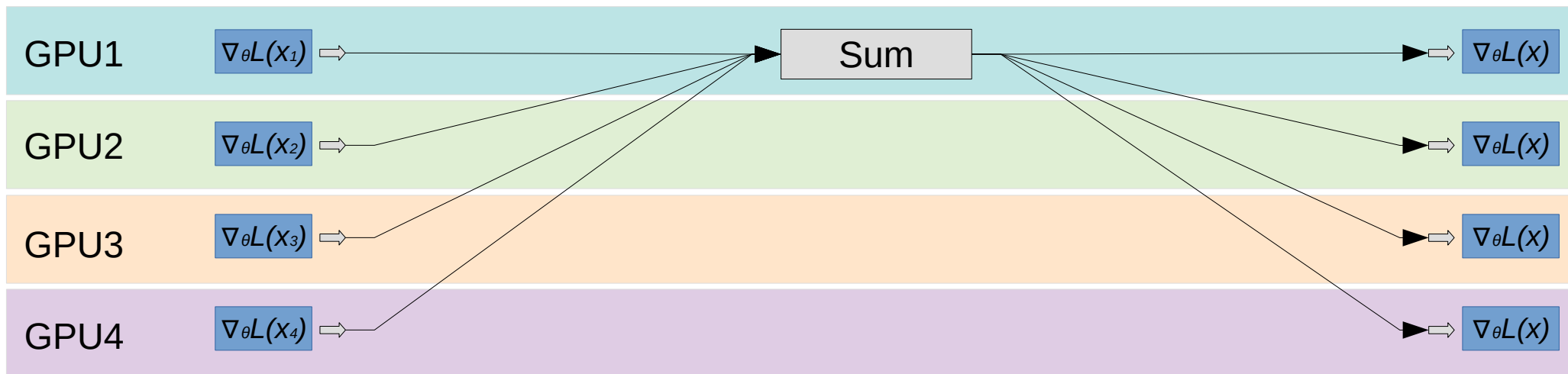
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

## Naive implementation

*Devices*



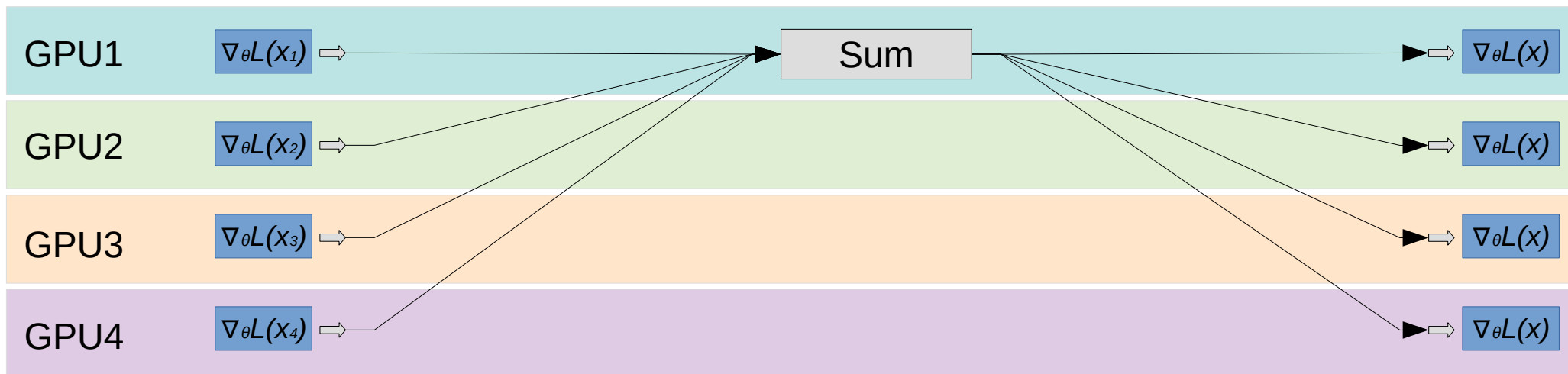
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

**Q:** Can we do better?

*Devices*



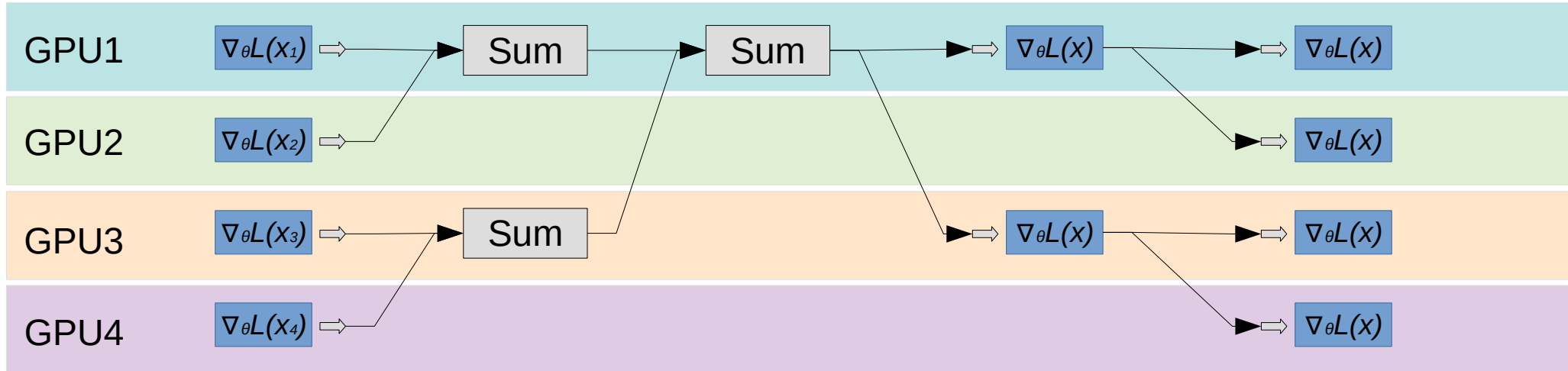
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

## Tree-allreduce

*Devices*



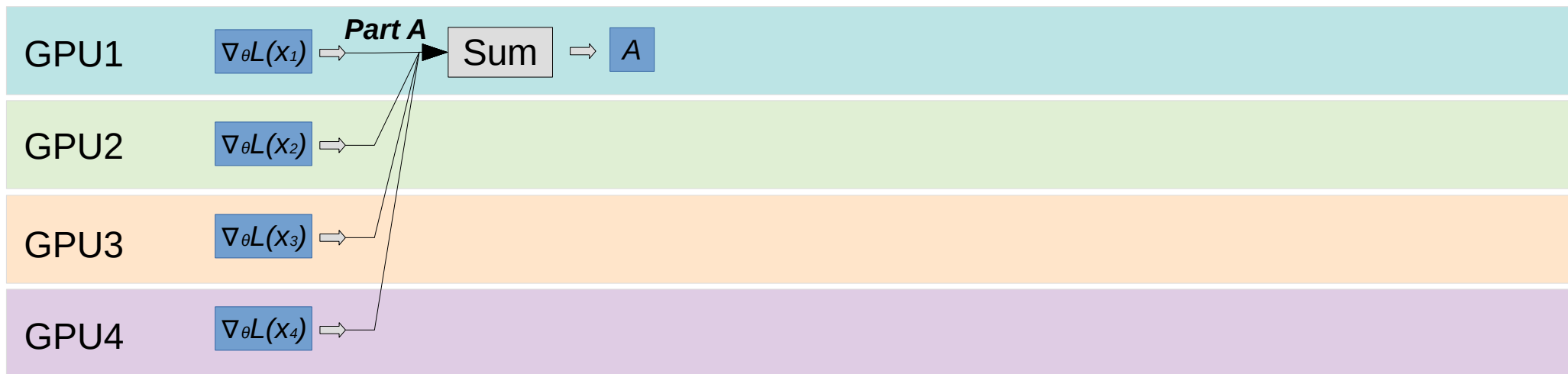
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

**Butterfly-allreduce – split data into chunks (ABCD)**

*Devices*



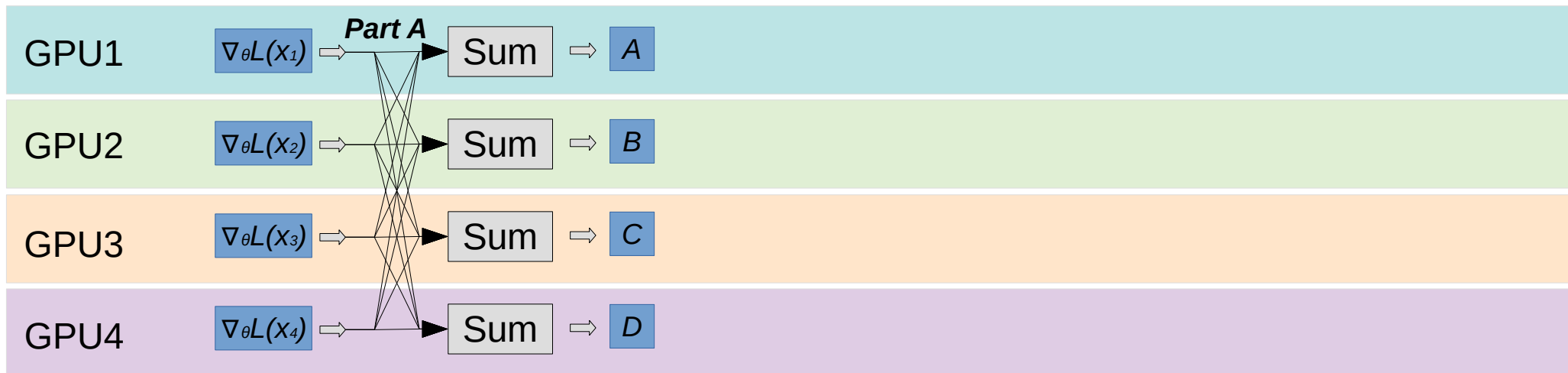
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

**Butterfly-allreduce – split data into chunks (ABCD)**

*Devices*



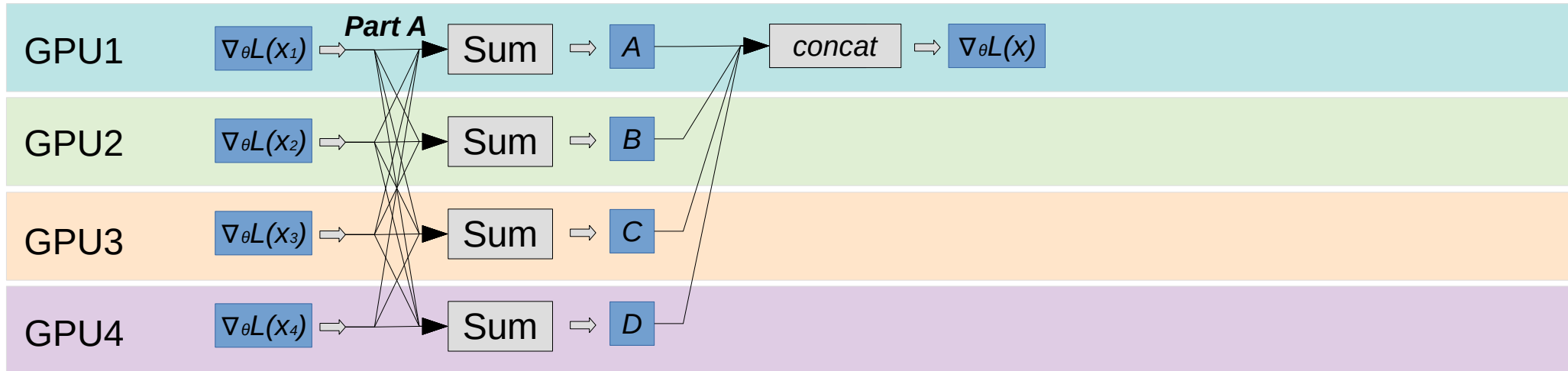
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

**Butterfly-allreduce – split data into chunks (ABCD)**

*Devices*



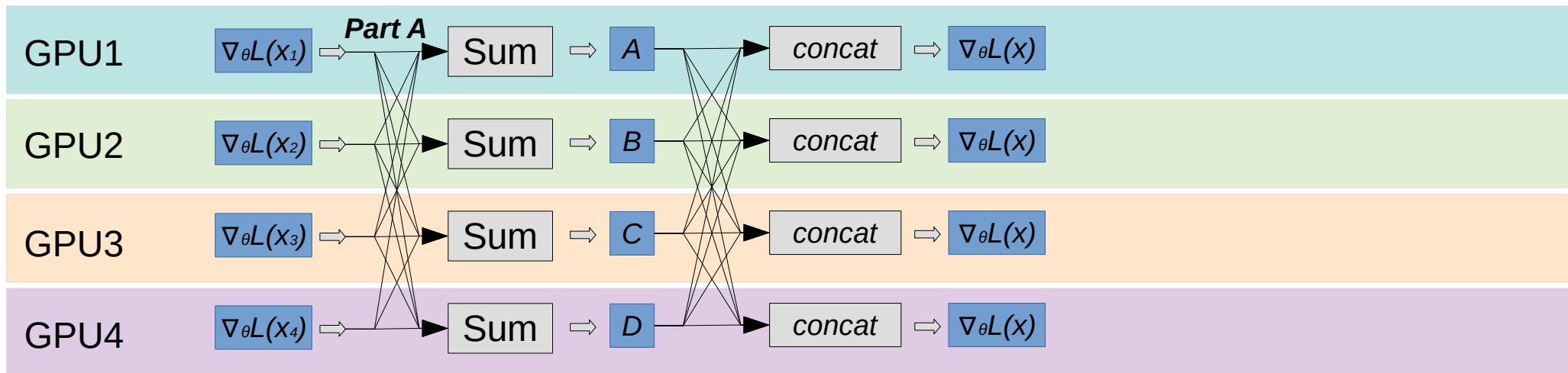
# Faster allreduce

**Input:** each device has its own vector

**Output:** each device gets a sum of all vectors

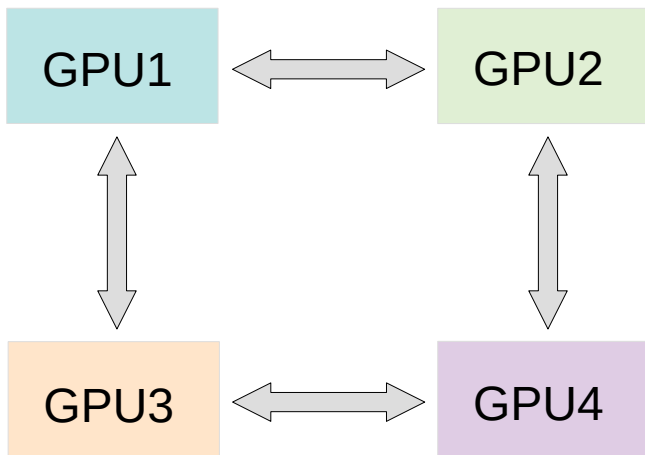
**Ring-allreduce – split data into chunks (ABCD)**

*Devices*



# 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](https://tinyurl.com/ring-allreduce-blog)

# Ring allreduce

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

*[Time to use the whiteboard]*

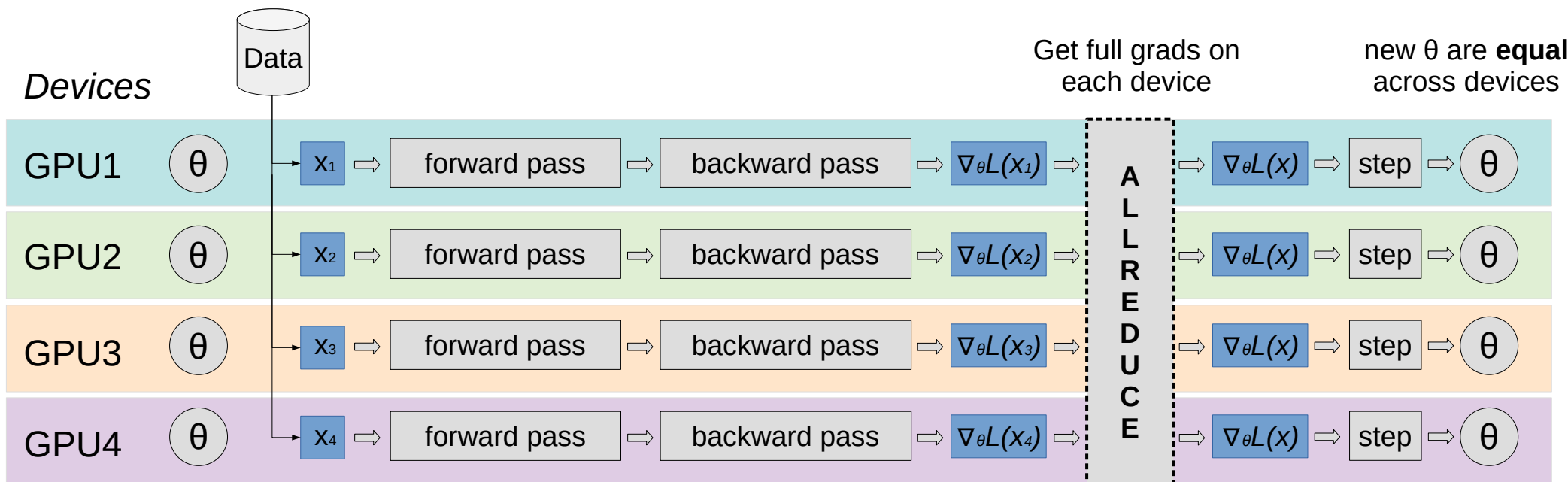
**Answer & more:** [tinyurl.com/ring-allreduce-blog](https://tinyurl.com/ring-allreduce-blog)

# All-Reduce data parallel

[arxiv.org/abs/1706.02677](https://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?



# </Data-parallel>

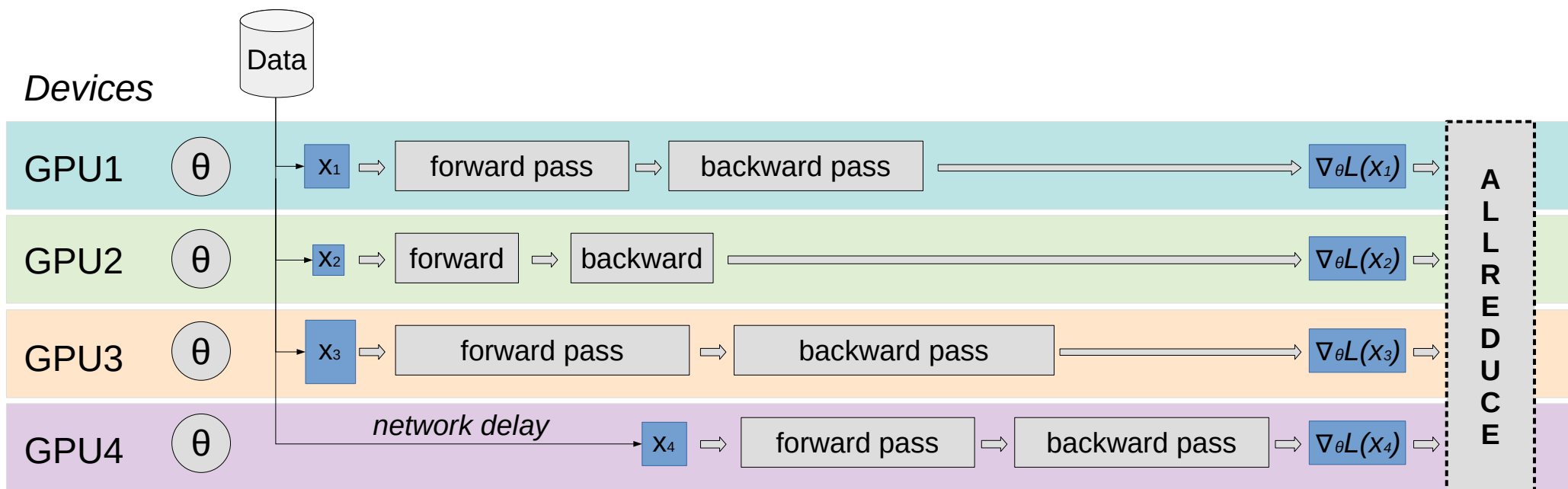
- + easy to implement
  - + can scale to 100s of gpus
  - + can be fault-tolerant
  - model must fit in 1 gpu
  - large batches aren't always good for generalization
- 
- 2-4 GPUs & no time – naive data parallel [tinyurl.com/torch-data-parallel](https://tinyurl.com/torch-data-parallel)
  - 4+ GPUs or multiple hosts – distributed (allreduce) [github.com/horovod/horovod](https://github.com/horovod/horovod)
    - Intro to pytorch distributed: [tinyurl.com/distributed-dp](https://tinyurl.com/distributed-dp) or in **15 minutes!**
  - Somewhat faulty GPU/network: synchronous data parallel + drop stragglers
  - Very faulty or uneven resources: asynchronous data parallel (more later)
  - Efficient training with large batches: LAMB <https://arxiv.org/abs/1904.00962>
  - Dynamically adding or removing resources: <https://tinyurl.com/torch-elastic>

# Decentralized training vs real-world tasks

[arxiv.org/abs/1706.02677](https://arxiv.org/abs/1706.02677)

Each gpu has different processing time & delays

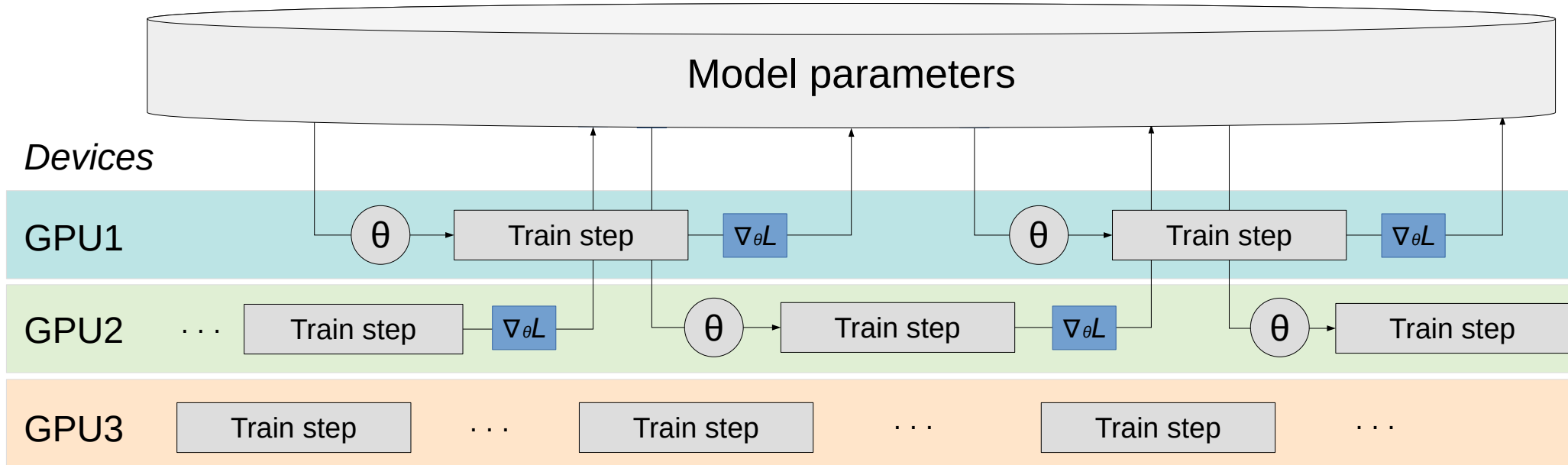
**Q:** can we improve device utilization?



# Recap: Parameter Server

HOGWILD! [arxiv.org/abs/1106.5730](https://arxiv.org/abs/1106.5730)

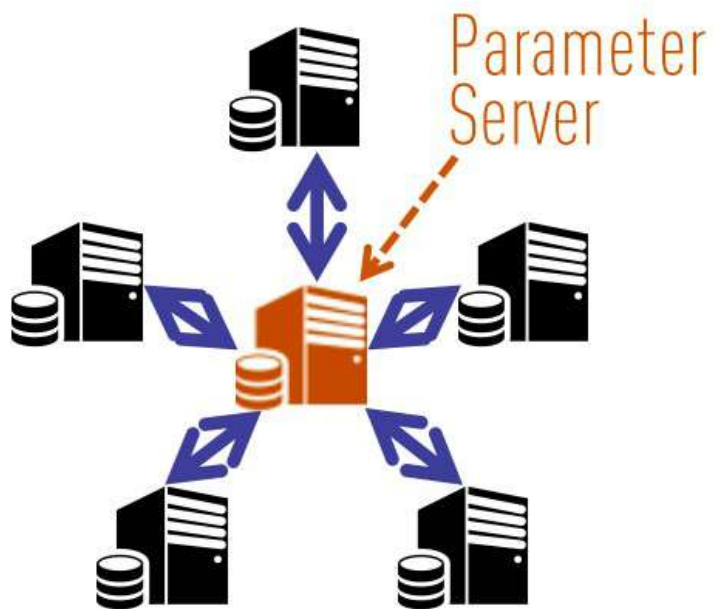
**Idea:** remove synchronization step altogether, use parameter server



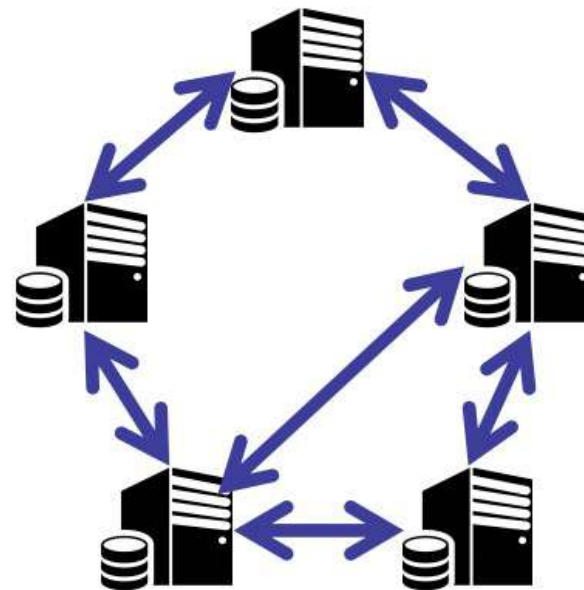
**Problem:** parameter servers need to ingest tons of data over training

# Decentralized Training with Gossip

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



**(a) Centralized Topology**



**(b) Decentralized Topology**

# Decentralized Training with Gossip

Source: <https://tinyurl.com/can-dsgd-outperform>

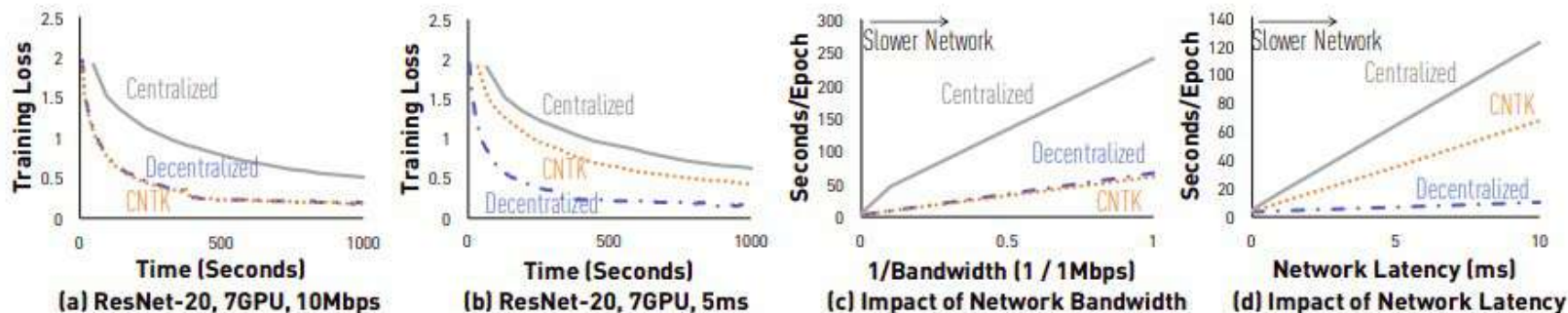


Figure 2: Comparison between D-PSGD and two centralized implementations (7 and 10 GPUs).

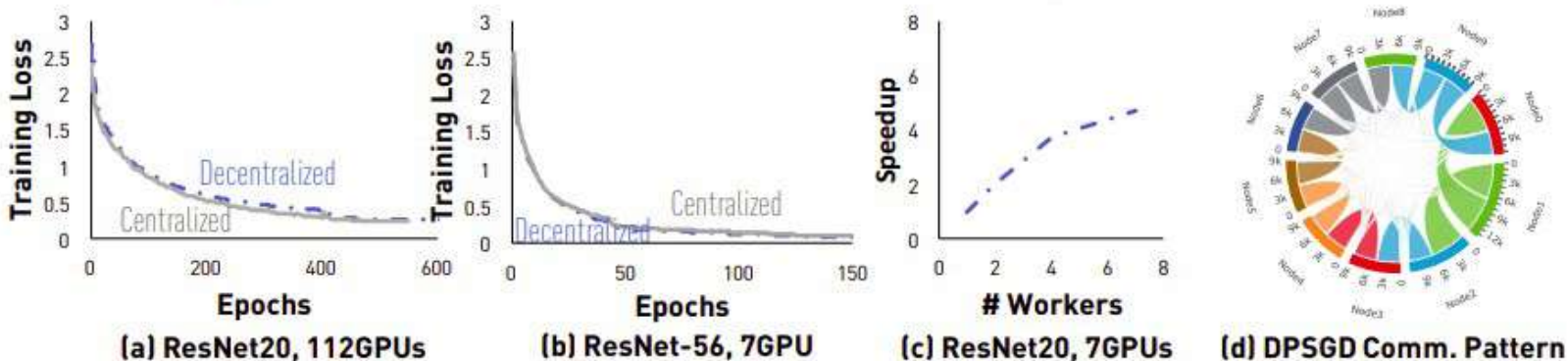
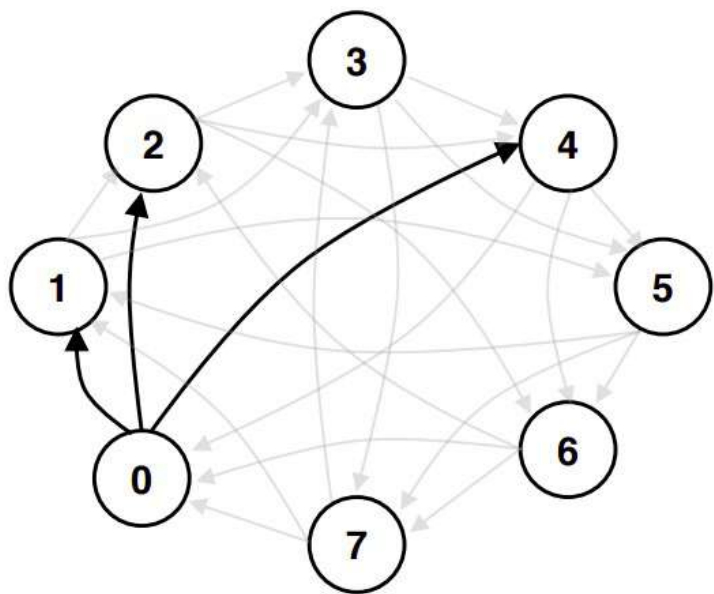


Figure 3: (a) Convergence Rate; (b) D-PSGD Speedup; (c) D-PSGD Communication Patterns.

# Stochastic Gradient Push

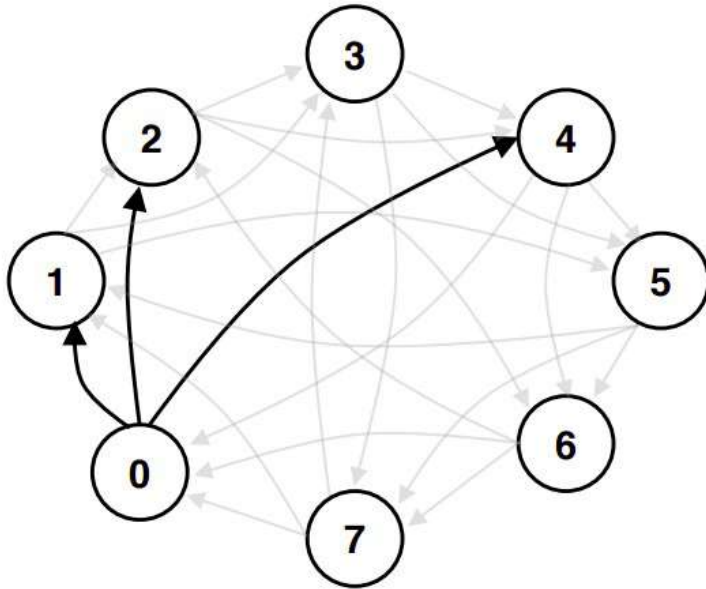
Source: <https://arxiv.org/abs/1811.10792>



(a) Directed Exponential Graph highlighting node 0's out-neighbours

# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>



(a) Directed Exponential Graph highlighting node 0's out-neighbours

---

**Algorithm 1** Stochastic Gradient Push (SGP)

---

**Require:** Initialize  $\gamma > 0$ ,  $\mathbf{x}_i^{(0)} = \mathbf{z}_i^{(0)} \in \mathbb{R}^d$  and  $w_i^{(0)} = 1$  for all nodes  $i \in \{1, 2, \dots, n\}$

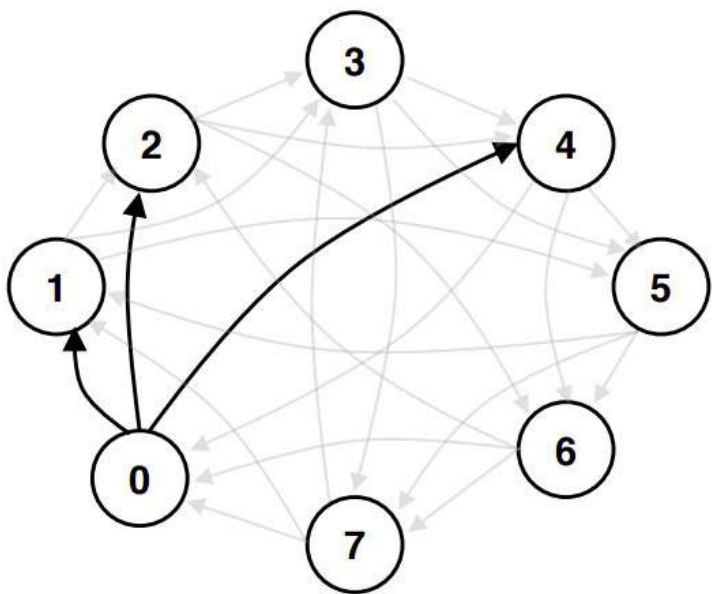
- 1: **for**  $k = 0, 1, 2, \dots, K$ , at node  $i$ , **do**
- 2:   Sample new mini-batch  $\xi_i^{(k)} \sim \mathcal{D}_i$  from local distribution
- 3:   Compute mini-batch gradient at  $\mathbf{z}_i^{(k)}$ :  $\nabla F_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$

<to be continued>

---

# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>



(a) Directed Exponential Graph highlighting node 0's out-neighbours

---

## Algorithm 1 Stochastic Gradient Push (SGP)

---

**Require:** Initialize  $\gamma > 0$ ,  $\mathbf{x}_i^{(0)} = \mathbf{z}_i^{(0)} \in \mathbb{R}^d$  and  $w_i^{(0)} = 1$  for all nodes  $i \in \{1, 2, \dots, n\}$

- 1: **for**  $k = 0, 1, 2, \dots, K$ , at node  $i$ , **do**
- 2:   Sample new mini-batch  $\xi_i^{(k)} \sim \mathcal{D}_i$  from local distribution
- 3:   Compute mini-batch gradient at  $\mathbf{z}_i^{(k)}$ :  $\nabla F_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$
- 4:    $\mathbf{x}_i^{(k+\frac{1}{2})} = \mathbf{x}_i^{(k)} - \gamma \nabla F_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$

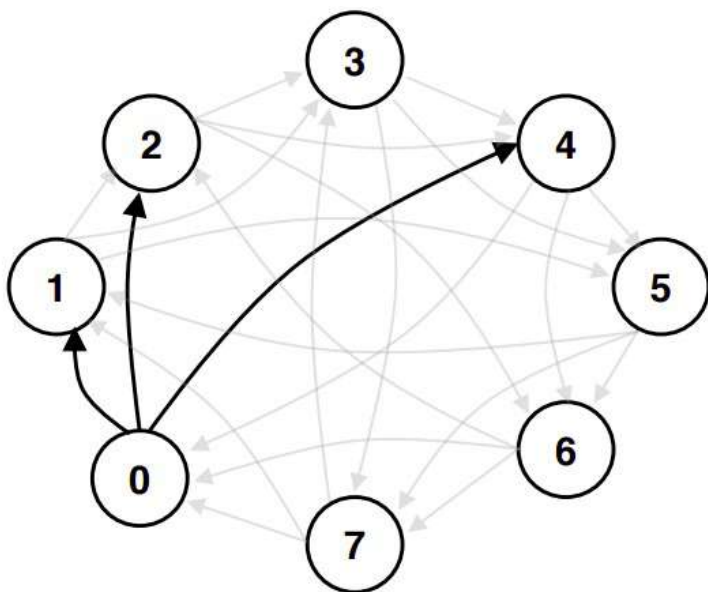
**normal GD step**

<to be continued>

---

# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>



(a) Directed Exponential Graph highlighting node 0's out-neighbours

---

## Algorithm 1 Stochastic Gradient Push (SGP)

---

**Require:** Initialize  $\gamma > 0$ ,  $\mathbf{x}_i^{(0)} = \mathbf{z}_i^{(0)} \in \mathbb{R}^d$  and  $w_i^{(0)} = 1$  for all nodes  $i \in \{1, 2, \dots, n\}$

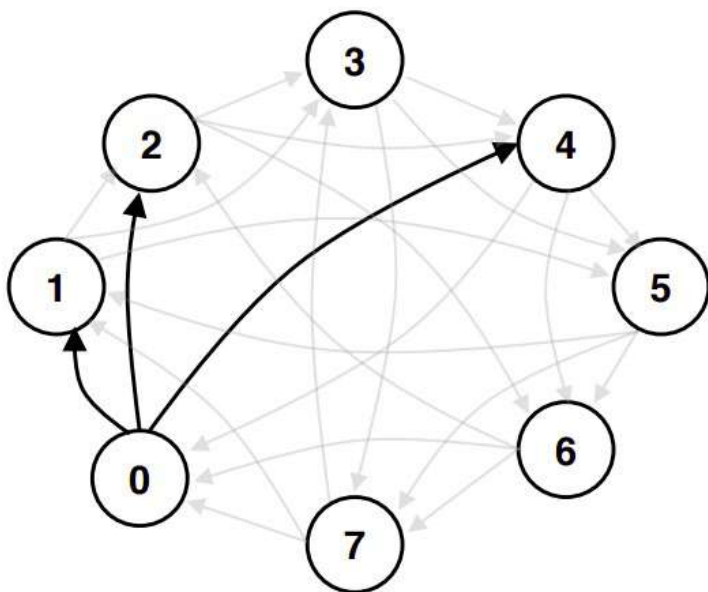
- 1: **for**  $k = 0, 1, 2, \dots, K$ , at node  $i$ , **do**
- 2:   Sample new mini-batch  $\xi_i^{(k)} \sim \mathcal{D}_i$  from local distribution
- 3:   Compute mini-batch gradient at  $\mathbf{z}_i^{(k)}$ :  $\nabla \mathbf{F}_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$
- 4:    $\mathbf{x}_i^{(k+\frac{1}{2})} = \mathbf{x}_i^{(k)} - \gamma \nabla \mathbf{F}_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$
- 5:   Send  $(p_{j,i}^{(k)} \mathbf{x}_i^{(k+\frac{1}{2})}, p_{j,i}^{(k)} w_i^{(k)})$  to out-neighbors;  
      receive  $(p_{i,j}^{(k)} \mathbf{x}_j^{(k+\frac{1}{2})}, p_{i,j}^{(k)} w_j^{(k)})$  from in-neighbors

<to be continued>

---

# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>



(a) Directed Exponential Graph highlighting node 0's out-neighbours

---

## Algorithm 1 Stochastic Gradient Push (SGP)

---

**Require:** Initialize  $\gamma > 0$ ,  $\mathbf{x}_i^{(0)} = \mathbf{z}_i^{(0)} \in \mathbb{R}^d$  and  $w_i^{(0)} = 1$  for all nodes  $i \in \{1, 2, \dots, n\}$

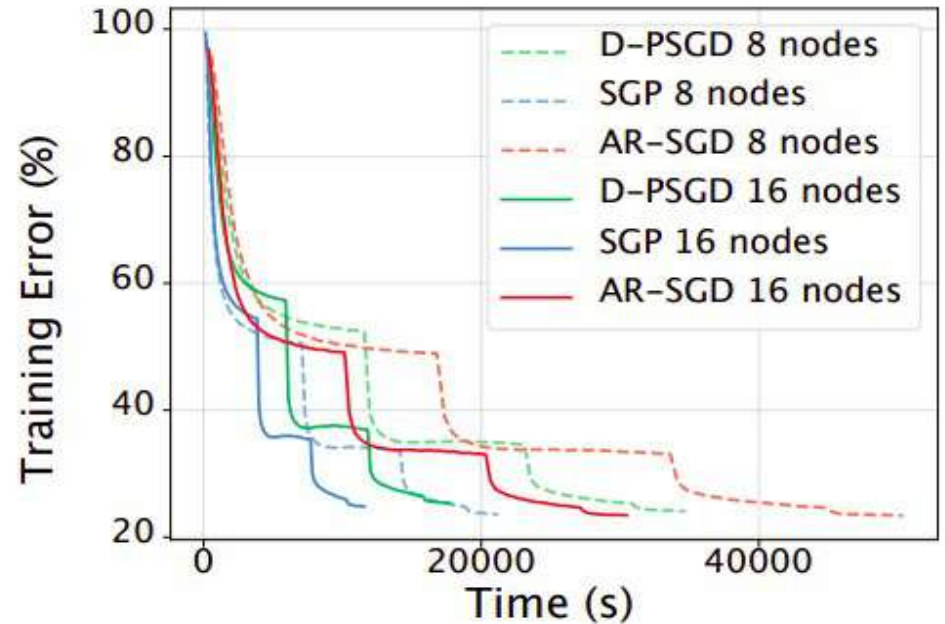
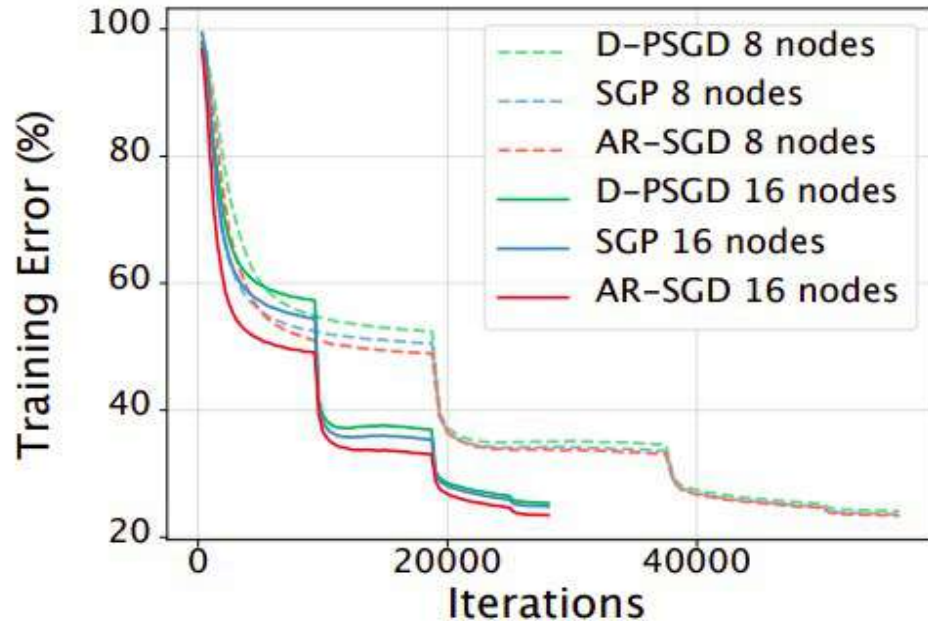
- 1: **for**  $k = 0, 1, 2, \dots, K$ , at node  $i$ , **do**
- 2:   Sample new mini-batch  $\xi_i^{(k)} \sim \mathcal{D}_i$  from local distribution
- 3:   Compute mini-batch gradient at  $\mathbf{z}_i^{(k)}$ :  $\nabla \mathbf{F}_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$
- 4:    $\mathbf{x}_i^{(k+\frac{1}{2})} = \mathbf{x}_i^{(k)} - \gamma \nabla \mathbf{F}_i(\mathbf{z}_i^{(k)}; \xi_i^{(k)})$
- 5:   Send  $(p_{j,i}^{(k)} \mathbf{x}_i^{(k+\frac{1}{2})}, p_{j,i}^{(k)} w_i^{(k)})$  to out-neighbors;  
       receive  $(p_{i,j}^{(k)} \mathbf{x}_j^{(k+\frac{1}{2})}, p_{i,j}^{(k)} w_j^{(k)})$  from in-neighbors
- 6:    $\mathbf{x}_i^{(k+1)} = \sum_j p_{i,j}^{(k)} \mathbf{x}_j^{(k+\frac{1}{2})}$
- 7:    $w_i^{(k+1)} = \sum_j p_{i,j}^{(k)} w_j^{(k)}$
- 8:    $\mathbf{z}_i^{(k+1)} = \mathbf{x}_i^{(k+1)} / w_i^{(k+1)}$
- 9: **end for**

**weighted  
average**

# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>

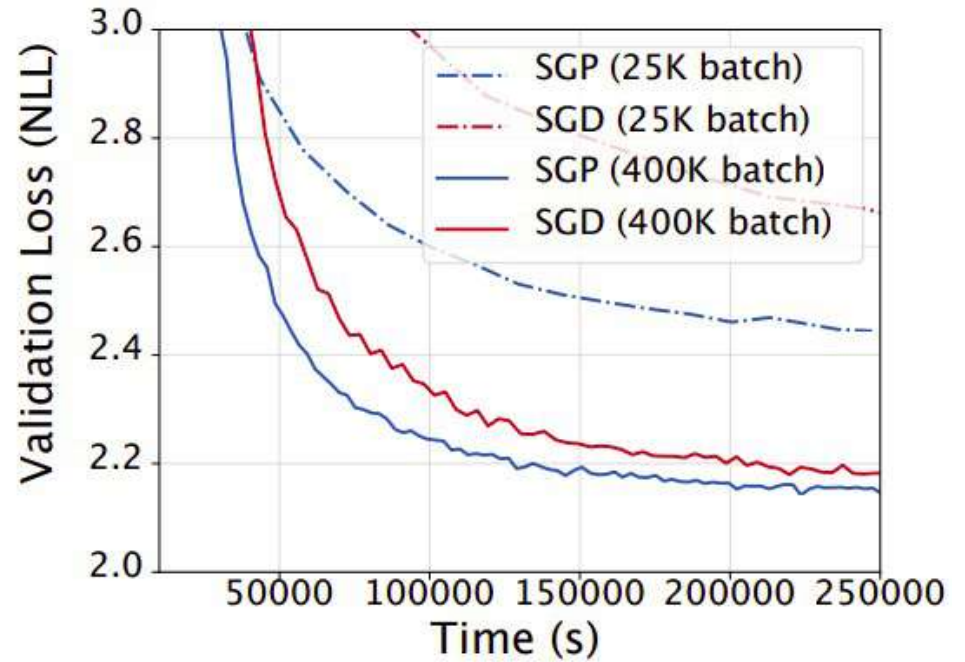
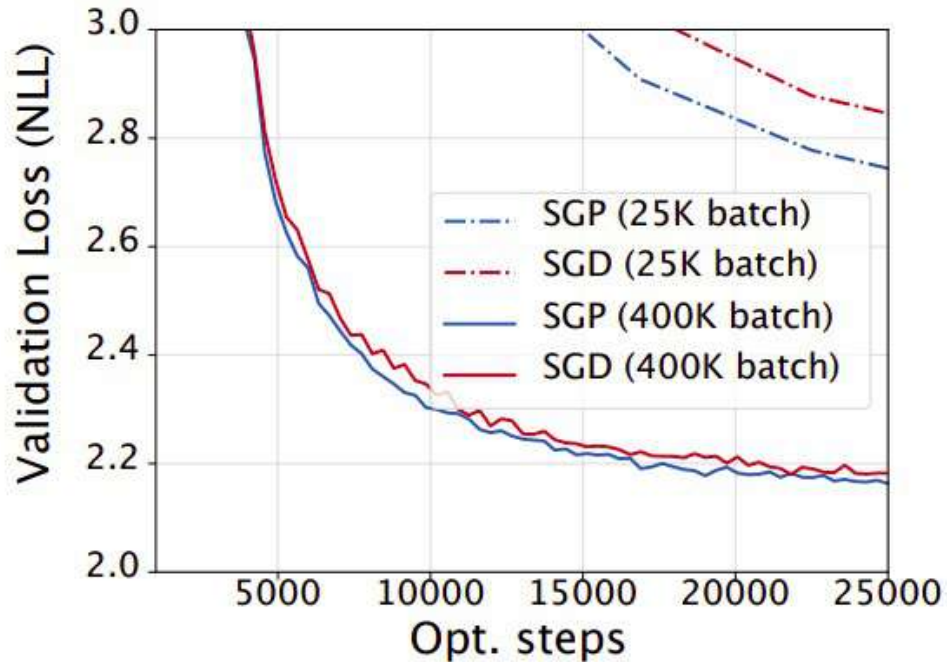
SGP vs ImageNet (ResNet50 + SGD w/ momentum)



# Stochastic Gradient Push

Source: <https://arxiv.org/abs/1811.10792>

## SGP vs WMT English-German (Transformer, Adam)



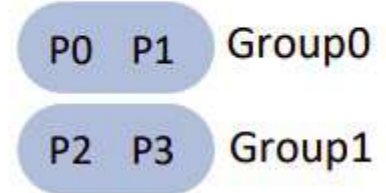
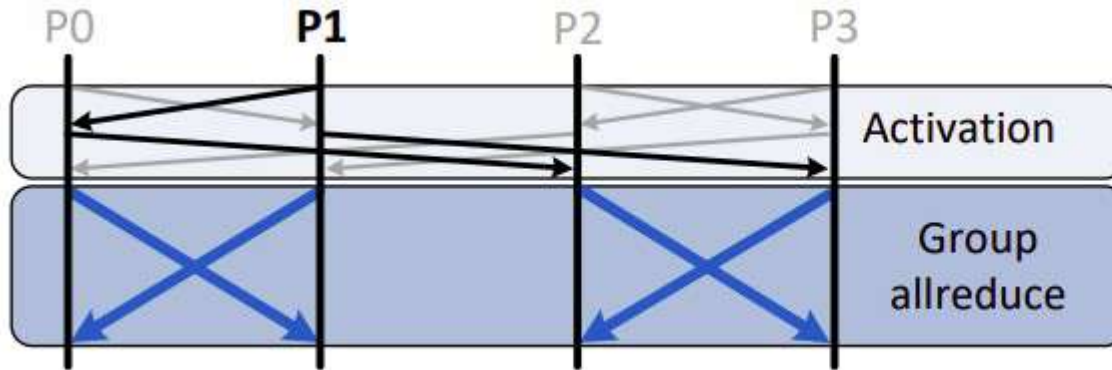
# Gossip vs All-Reduce

Your thoughts?

# Gossip + All-Reduce

Source: [arxiv.org/abs/2005.00124](https://arxiv.org/abs/2005.00124)

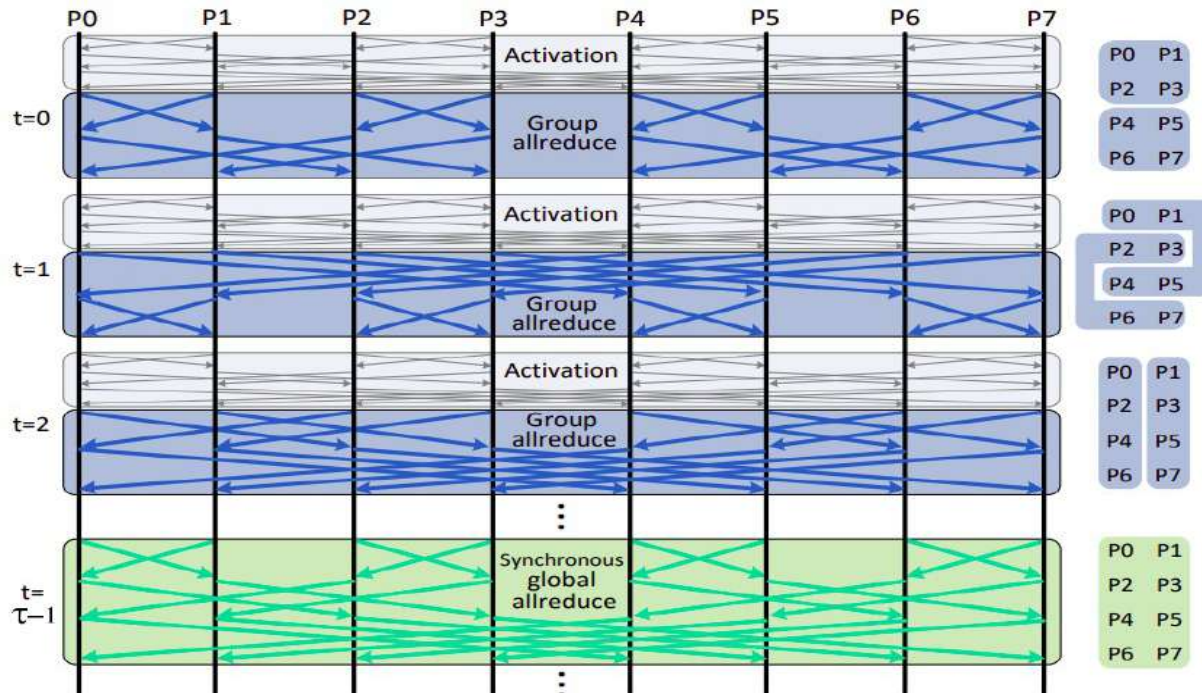
**Core idea:** run all-reduce in independent groups  
You only have to synchronize for your small group  
Swap groupmates between iterations



# Gossip + All-Reduce

Source: [arxiv.org/abs/2005.00124](https://arxiv.org/abs/2005.00124)

**Core idea:** run all-reduce in independent groups  
You only have to synchronize for your small group  
**Swap** groupmates between iterations

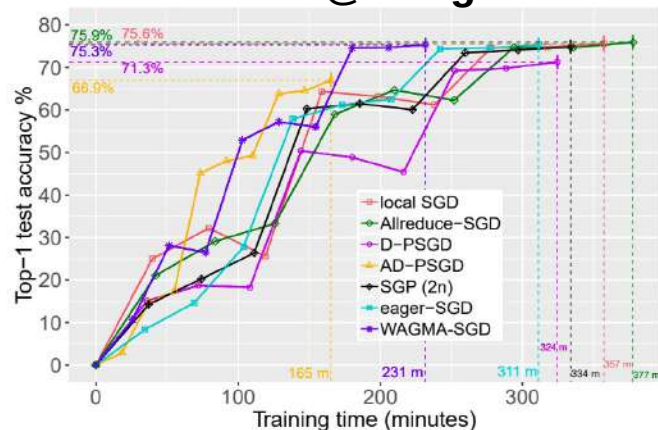


# Gossip + All-Reduce

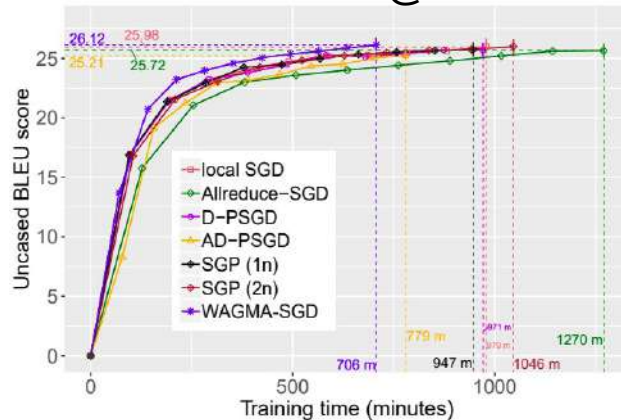
Source: [arxiv.org/abs/2005.00124](https://arxiv.org/abs/2005.00124)

**Experiment setup:** up to 1024 GPU,  
Natural (or emulated) network latency

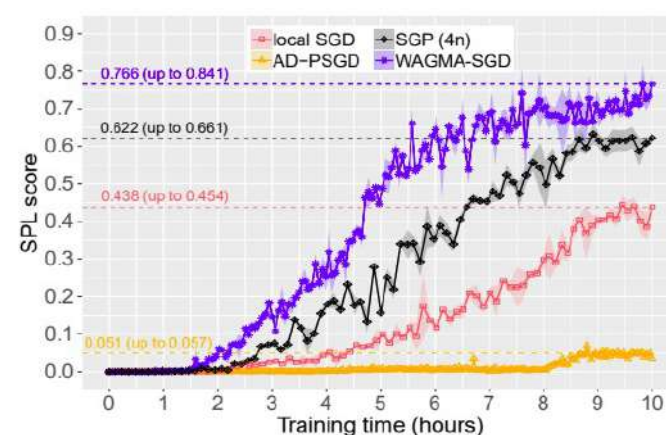
**Image Classification**  
**ResNet50 @ ImageNet**



**Machine Translation**  
**Transformer @ WMT17**



**Reinforcement Learning**  
**DDPO on Habitat**



Q: what if sending tensors during  
AllReduce takes too long?

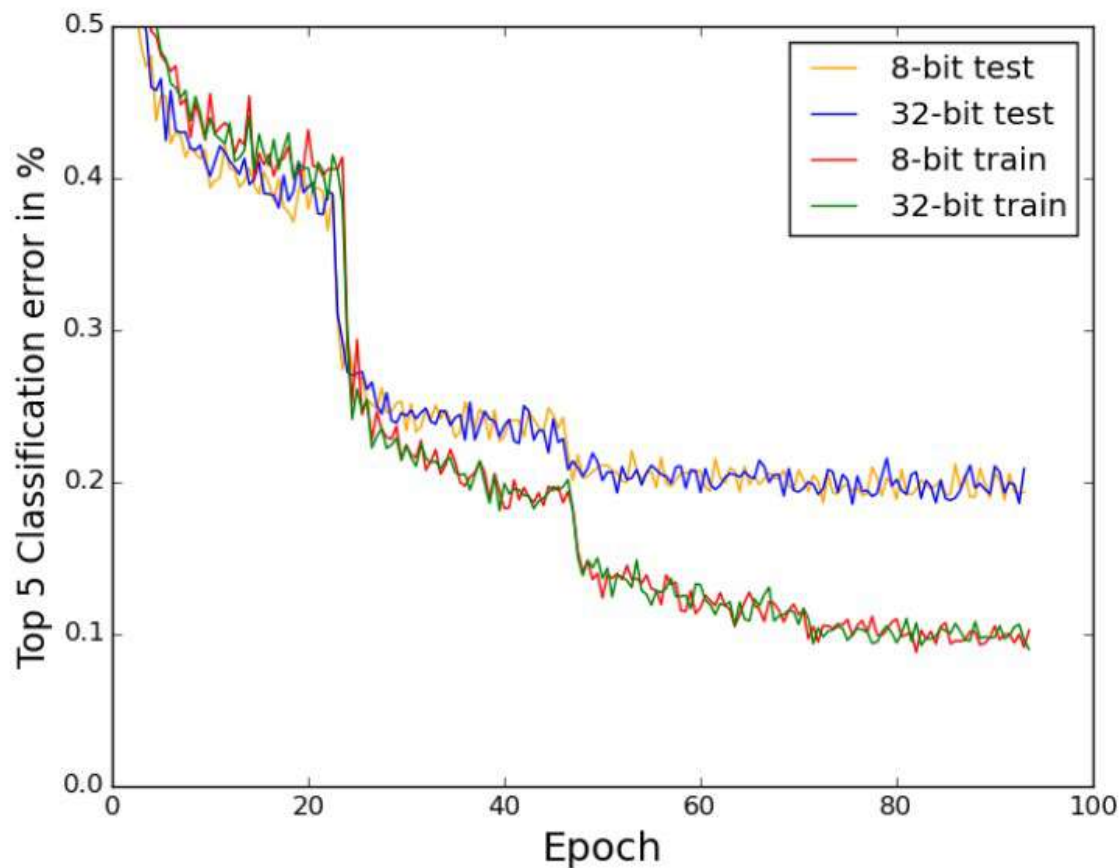
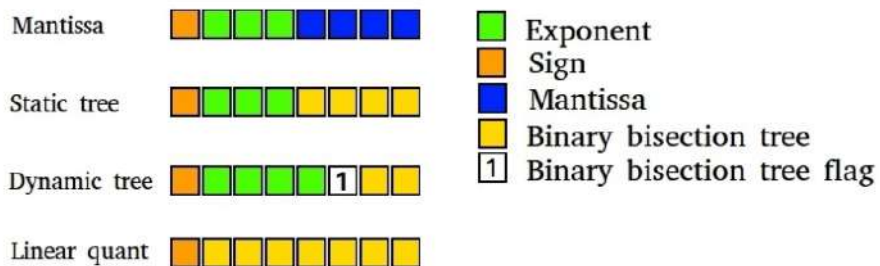
# Quantized communication

<https://arxiv.org/abs/1511.04561>

TL;DR

- send data in 8-bit
- all computations in 32-bit
- choose best data format

PROFIT: same quality as float16



Can we compress further?  
*without losing quality*

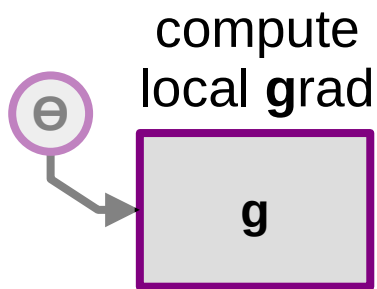
# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

TL;DR - use extreme compression, e.g. 1-bit or top-5% gradients

- if you lose something in compression, **reuse it on the next step**



time flows left to right →

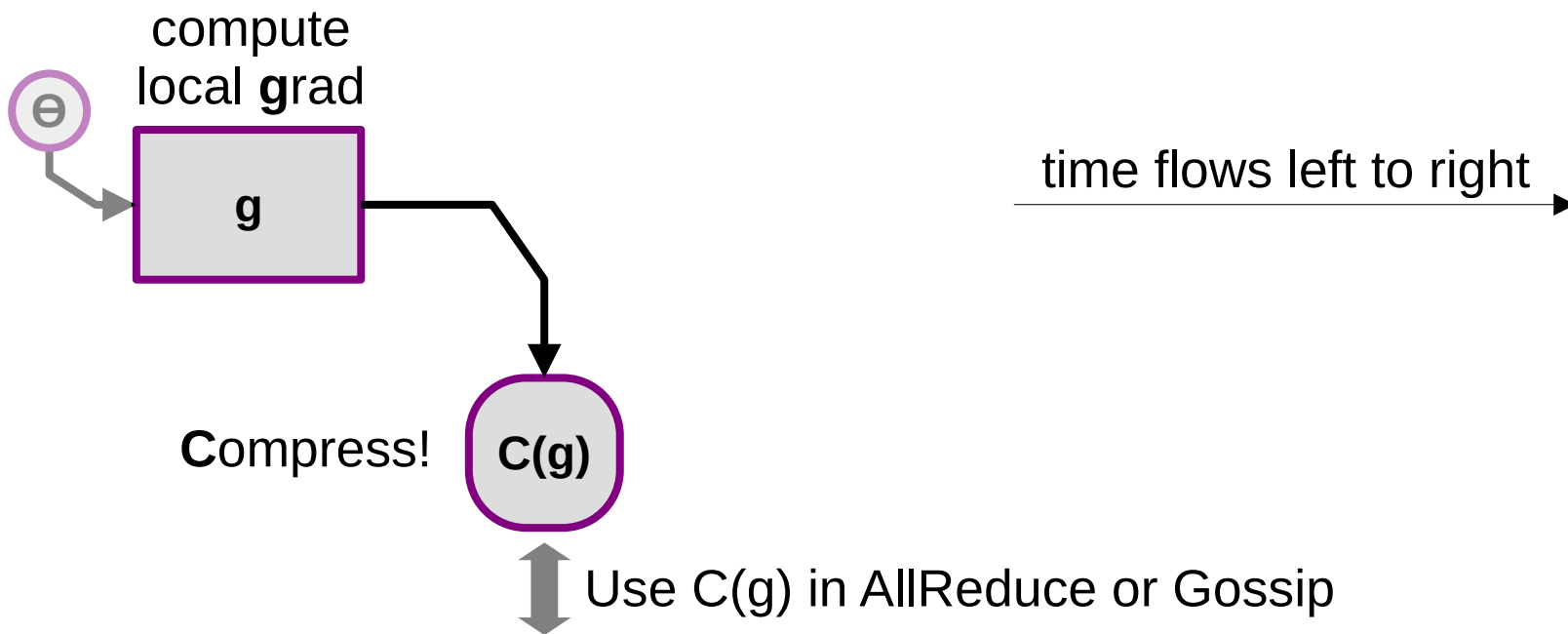
# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

TL;DR - use extreme compression, e.g. 1-bit or top-5% gradients

- if you lose something in compression, **reuse it on the next step**



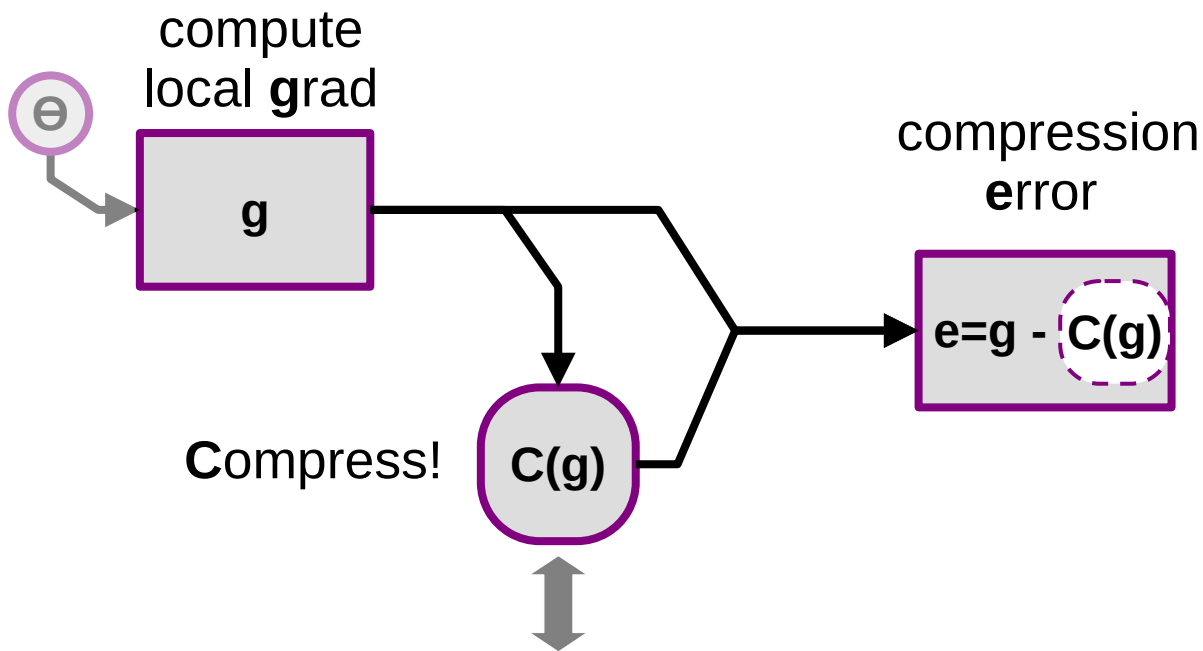
# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

TL;DR - use extreme compression, e.g. 1-bit or top-5% gradients

- if you lose something in compression, **reuse it on the next step**



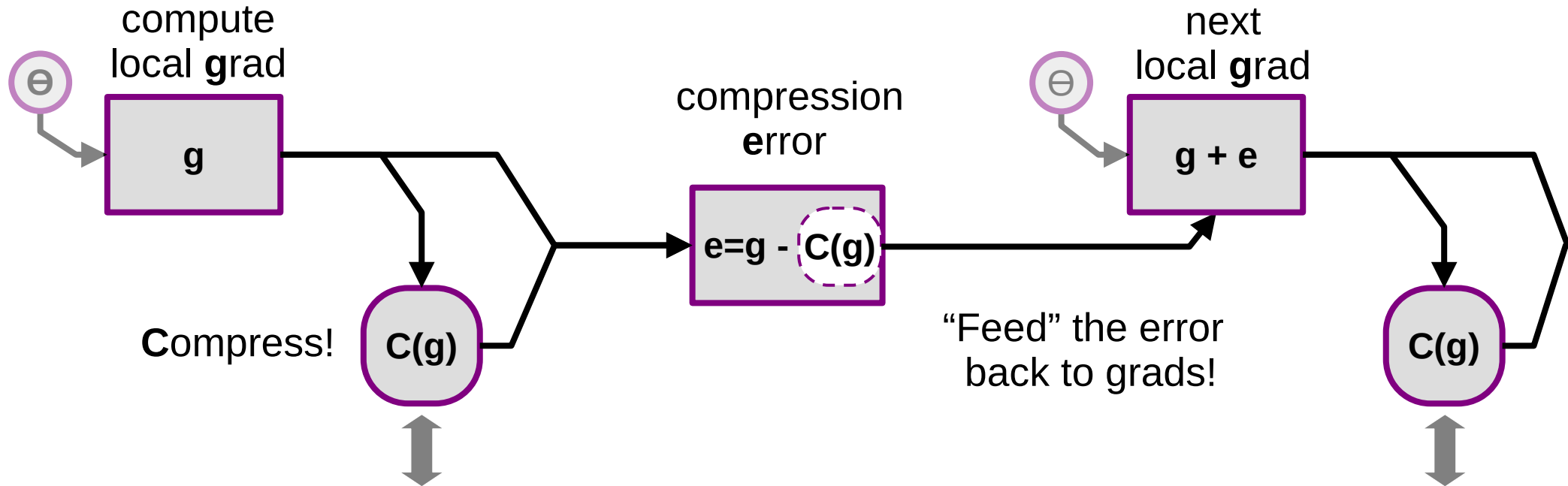
# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

TL;DR - use extreme compression, e.g. 1-bit or top-5% gradients

- if you lose something in compression, **reuse it on the next step**



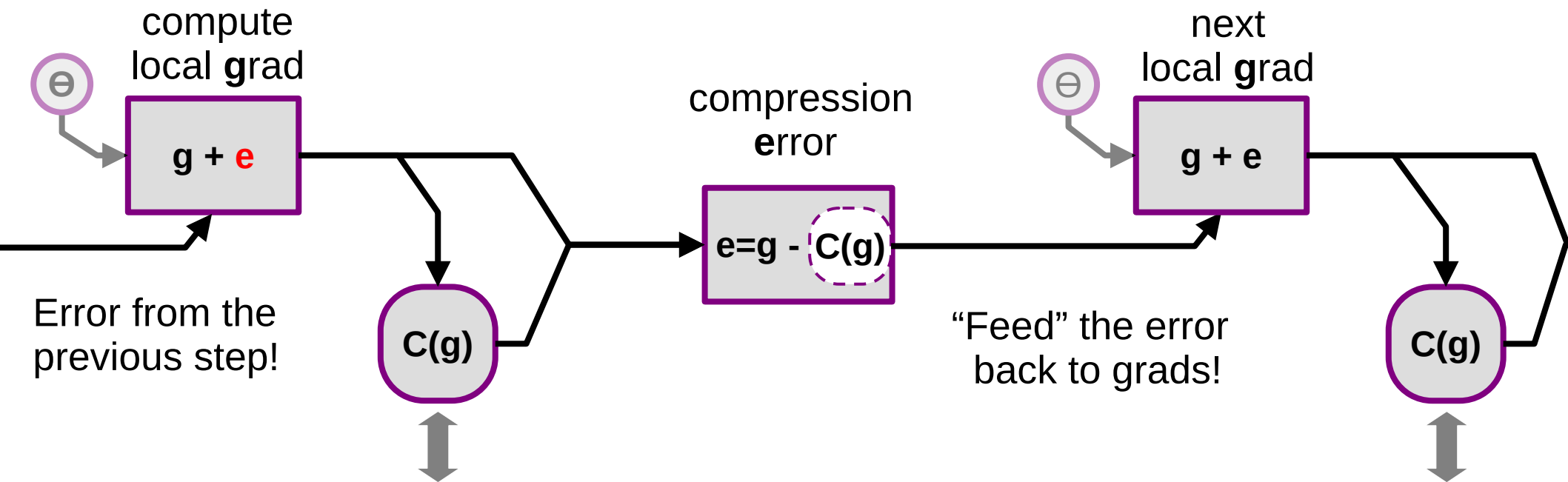
# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

TL;DR - use extreme compression, e.g. 1-bit or top-5% gradients

- if you lose something in compression, **reuse it on the next step**



# Error Feedback + PowerSGD

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

```
1: hyperparameters: learning rate  $\gamma$ , momentum parameter  $\lambda$ 
2: initialize model parameters  $\mathbf{x} \in \mathbb{R}^d$ , momentum  $\mathbf{m} \leftarrow \mathbf{0} \in \mathbb{R}^d$ , replicated across workers
3: at each worker  $w = 1, \dots, W$  do
4:   initialize memory  $\mathbf{e}_w \leftarrow \mathbf{0} \in \mathbb{R}^d$ 
5:   for each iterate  $t = 0, \dots$  do
6:     Compute a stochastic gradient  $\mathbf{g}_w \in \mathbb{R}^d$ .
7:      $\Delta_w \leftarrow \mathbf{g}_w + \mathbf{e}_w$  ▷ Incorporate error-feedback into update
8:      $\mathcal{C}(\Delta_w) \leftarrow \text{COMPRESS}(\Delta_w)$ 
9:      $\mathbf{e}_w \leftarrow \Delta_w - \text{DECOMPRESS}(\mathcal{C}(\Delta_w))$  ▷ Memorize local errors
10:     $\mathcal{C}(\Delta) \leftarrow \text{AGGREGATE}(\mathcal{C}(\Delta_1), \dots, \mathcal{C}(\Delta_W))$  ▷ Exchange gradients
11:     $\Delta' \leftarrow \text{DECOMPRESS}(\mathcal{C}(\Delta))$  ▷ Reconstruct an update  $\in \mathbb{R}^d$ 
12:     $\mathbf{m} \leftarrow \lambda \mathbf{m} + \Delta'$ 
13:     $\mathbf{x} \leftarrow \mathbf{x} - \gamma (\Delta' + \mathbf{m})$ 
14:  end for
15: end at
```

# PowerSGD: low-rank approx grads + Error Feedback

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/1905.13727> - PowerSGD

- 1: The update vector  $\Delta_w$  is treated as a list of tensors corresponding to individual model parameters. Vector-shaped parameters (biases) are aggregated uncompressed. Other parameters are reshaped into matrices. The functions below operate on such matrices independently. For each matrix  $M \in \mathbb{R}^{n \times m}$ , a corresponding  $Q \in \mathbb{R}^{m \times r}$  is initialized from an i.i.d. standard normal distribution.
- 2: **function** COMPRESS+AGGREGATE(update matrix  $M \in \mathbb{R}^{n \times m}$ , previous  $Q \in \mathbb{R}^{m \times r}$ )
- 3:      $P \leftarrow MQ$
- 4:      $P \leftarrow \text{ALL REDUCE MEAN}(P)$  ▷ Now,  $P = \frac{1}{W}(M_1 + \dots + M_W)Q$
- 5:      $\hat{P} \leftarrow \text{ORTHOGONALIZE}(P)$  ▷ Orthonormal columns
- 6:      $Q \leftarrow M^\top \hat{P}$
- 7:      $Q \leftarrow \text{ALL REDUCE MEAN}(Q)$  ▷ Now,  $Q = \frac{1}{W}(M_1 + \dots + M_W)^\top \hat{P}$
- 8:     **return** the compressed representation  $(\hat{P}, Q)$ .
- 9: **end function**
- 10: **function** DECOMPRESS( $\hat{P} \in \mathbb{R}^{n \times r}$ ,  $Q \in \mathbb{R}^{m \times r}$ )
- 11:     **return**  $\hat{P}Q^\top$
- 12: **end function**

# Read More: gradient compression

<https://arxiv.org/abs/1901.09847> - EF theory

<https://arxiv.org/abs/2106.05203> - better EF'21

<https://arxiv.org/abs/1905.13727> - PowerSGD

<https://arxiv.org/abs/2110.03294> - more EF'21

```
1 import torch.distributed.algorithms.ddp_comm_hooks.powerSGD_hook as powerSGD
2
3 ddp_model = nn.parallel.DistributedDataParallel(
4     module=model,
5     device_ids=[rank],
6 )
7
8 state = PowerSGD.PowerSGDState(
9     process_group=process_group,
10    matrix_approximation_rank=1,
11    start_powerSGD_iter=1_000,
12 )
13 ddp_model.register_comm_hook(state, PowerSGD.powerSGD_hook)
```



*"That's all Folks!"*

l s b e r g®

# Summary: operation parallelism

***Data-parallel:***

***???***

---

***Model-parallel:***

***???***

# Summary: operation parallelism

**Data-parallel:** *one process applies all model on **partial data**  
best for smaller model, more computations*

---

**Model-parallel:** *one process applies **partial model** on all data  
best for larger model, fewer computations*

*Which one is better..*

*for ResNet50?*

*for Llama 70B?*

*In general?*

# Summary: operation parallelism

**Data-parallel:** *one process applies all model on **partial data**  
best for smaller model, more computations*

---

**Model-parallel:** *one process applies **partial model** on all data  
best for larger model, fewer computations*

~~Which one is better..  
for ResNet50?  
In Llama 70B?~~

*It depends...*  
- on model size  
- on compute