Parallelism – 1

EECE695D: Efficient ML Systems

Spring 2025

Recap

- **Last two weeks.** Efficient Training
    - Idea. Re-use the experience of previous training runs

- **Today.** Parallelism
    - Accelerate training by using multiple devices in parallel
    - Key question. How do we coordinate the computations in many devices?

Motivation

- Modern models require too much **computation** to be trained

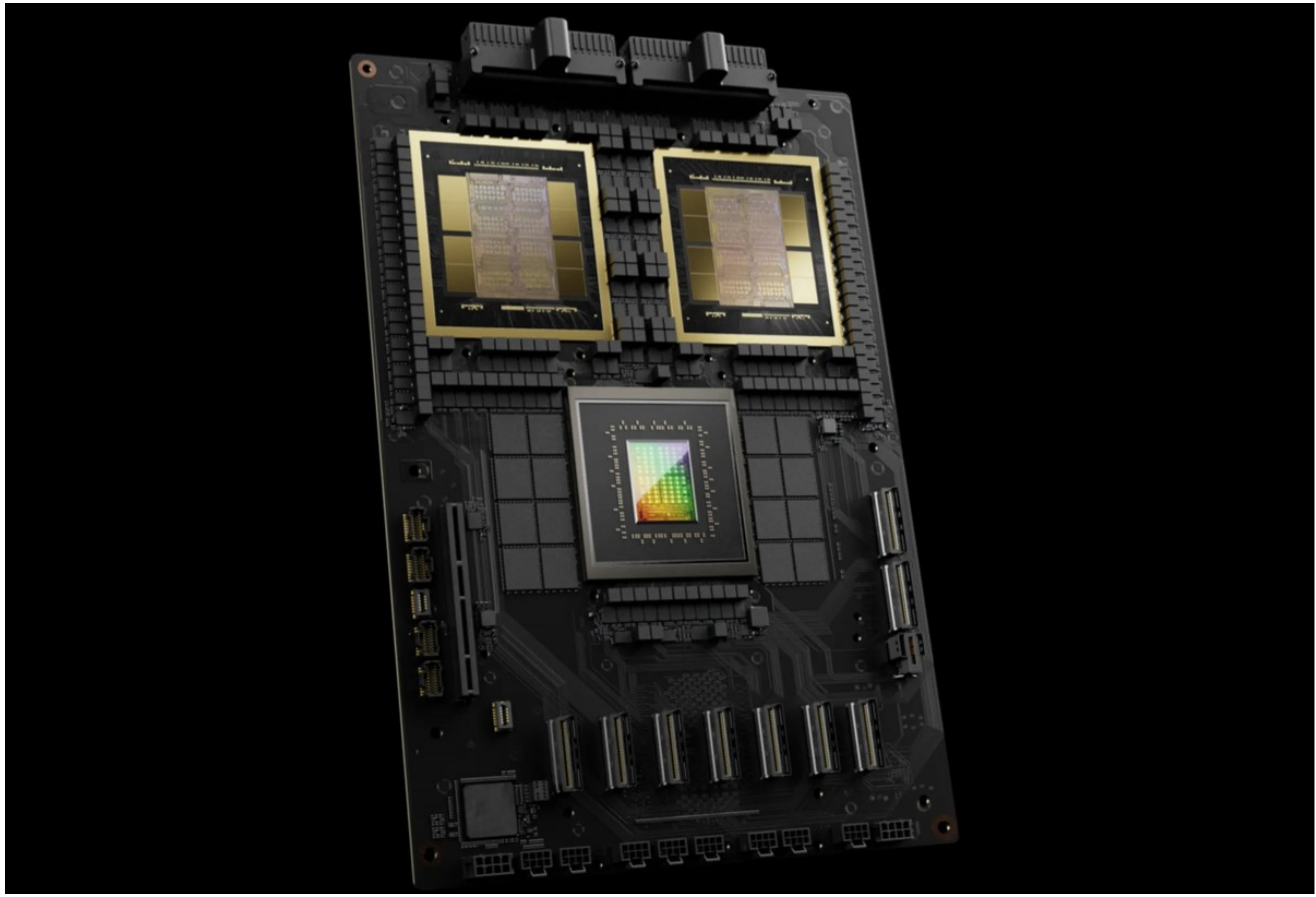
- Example.
  Estimated training cost of GPT-4

  $$\sim 2.0 \times 10^{25} \text{ FLOPs}$$

  NVIDIA B200 GPU handles, in FP16,

  $$2.25 \times 10^{15} \text{ FLOPS}$$

  That is, **282 years** of training!



Motivation

- Modern models require too much **parameters & RAM** to be trained

- Example.
  Fine-tuning a LLaMA-65B requires

  ~ 457GBs of RAM

  NVIDIA B100 GPU has 192GB

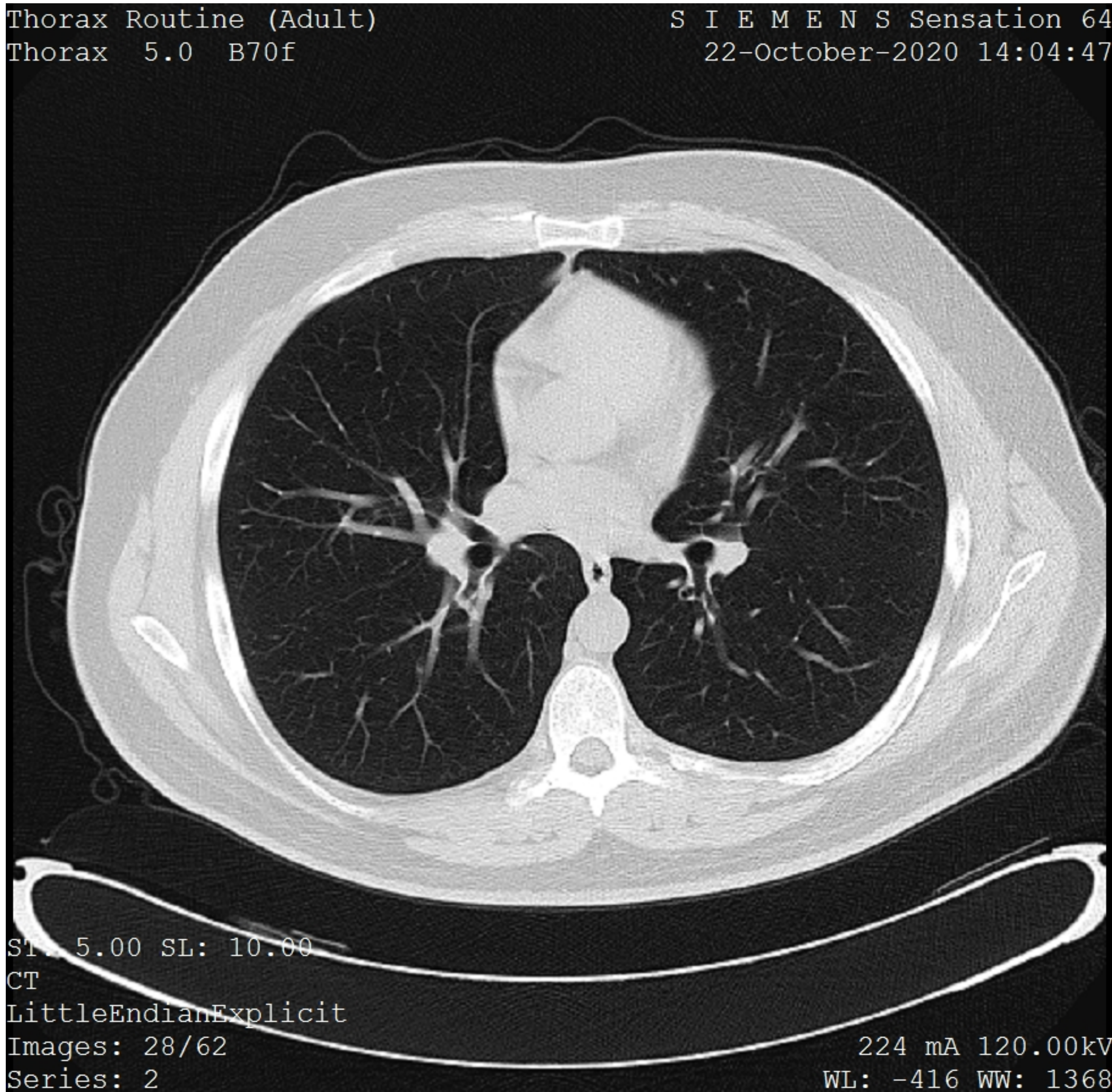
  That is, can only train **27B** model!

Motivation

- Modern models require too much **data** to be trained
  - Too large to be store in single node
- Example.
  DBRX was trained on 12T tokens

  ~ 60TB

  8-GPU servers of my group has only 13TB of storage
- Some data are private or classified
  - Medical or military

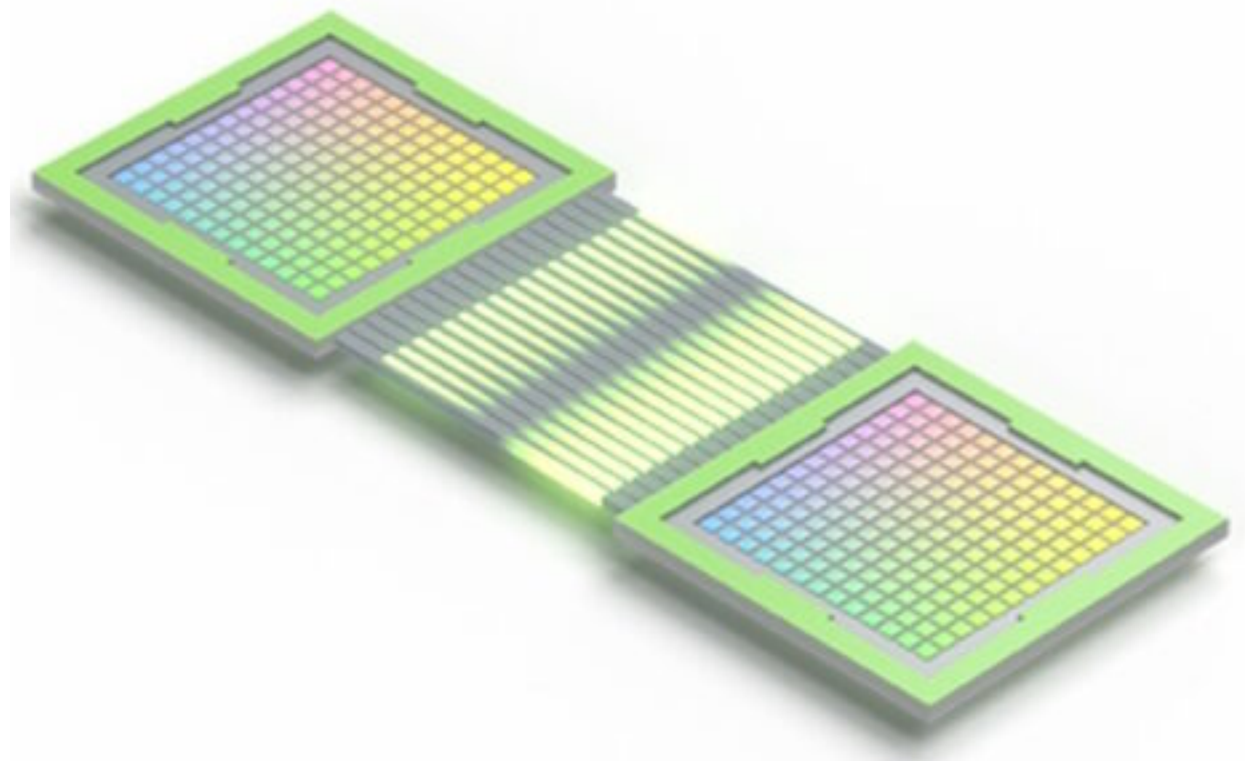


Motivation

- Modern models require too much **energy** to be trained
    - Not many are renewable or green
- Some renewable energy sources require a careful scheduling
- Inefficient to store or send to remote locations

Key challenge

- 100x resource ≠ 100x faster
- Communication between resources (NVLink, InifiniBand, …)
  - e.g., gradients, parameter updates, optimizer states
- Synchronization between resources
  - e.g., 7 fast GPUs and 1 slow GPU



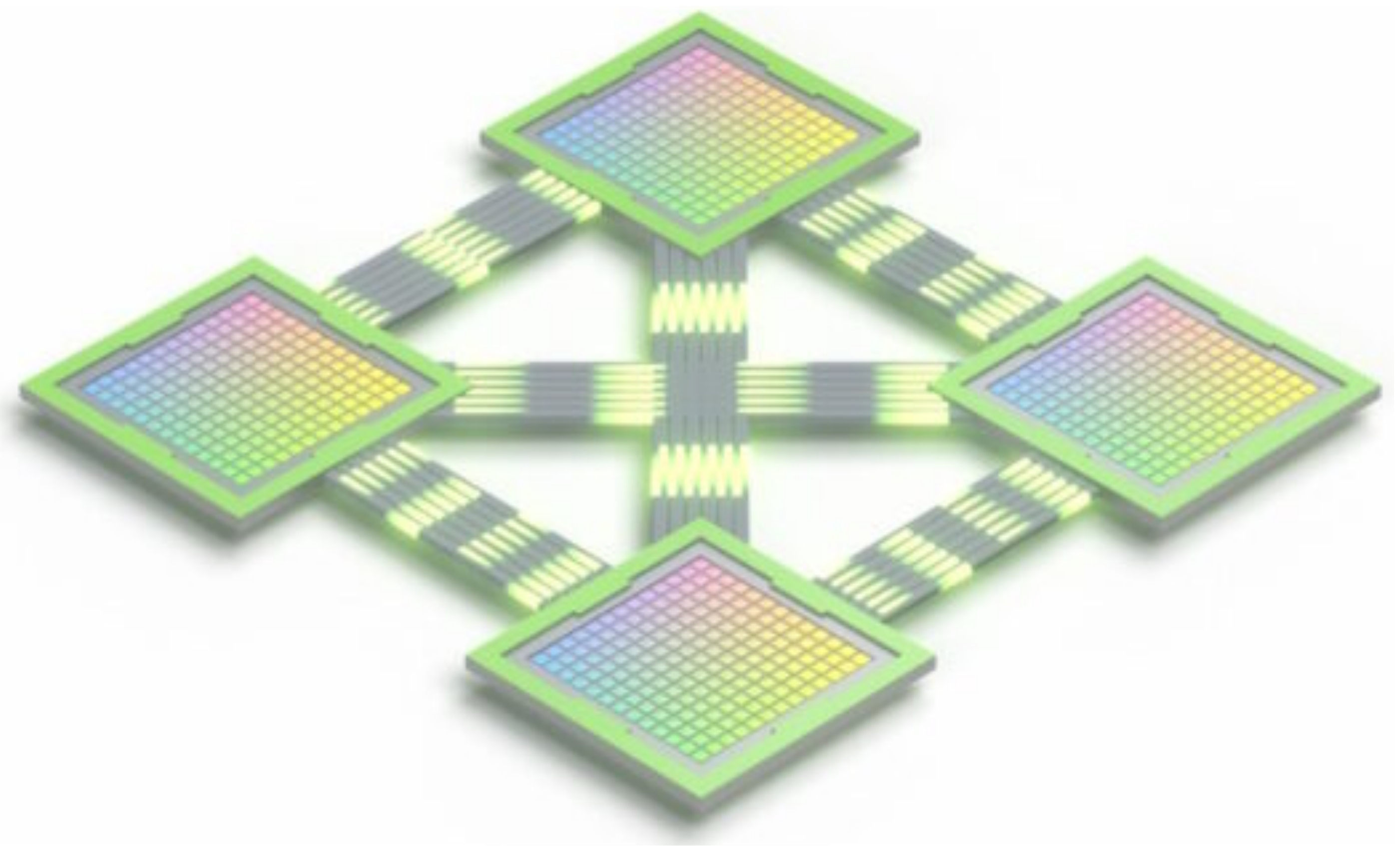
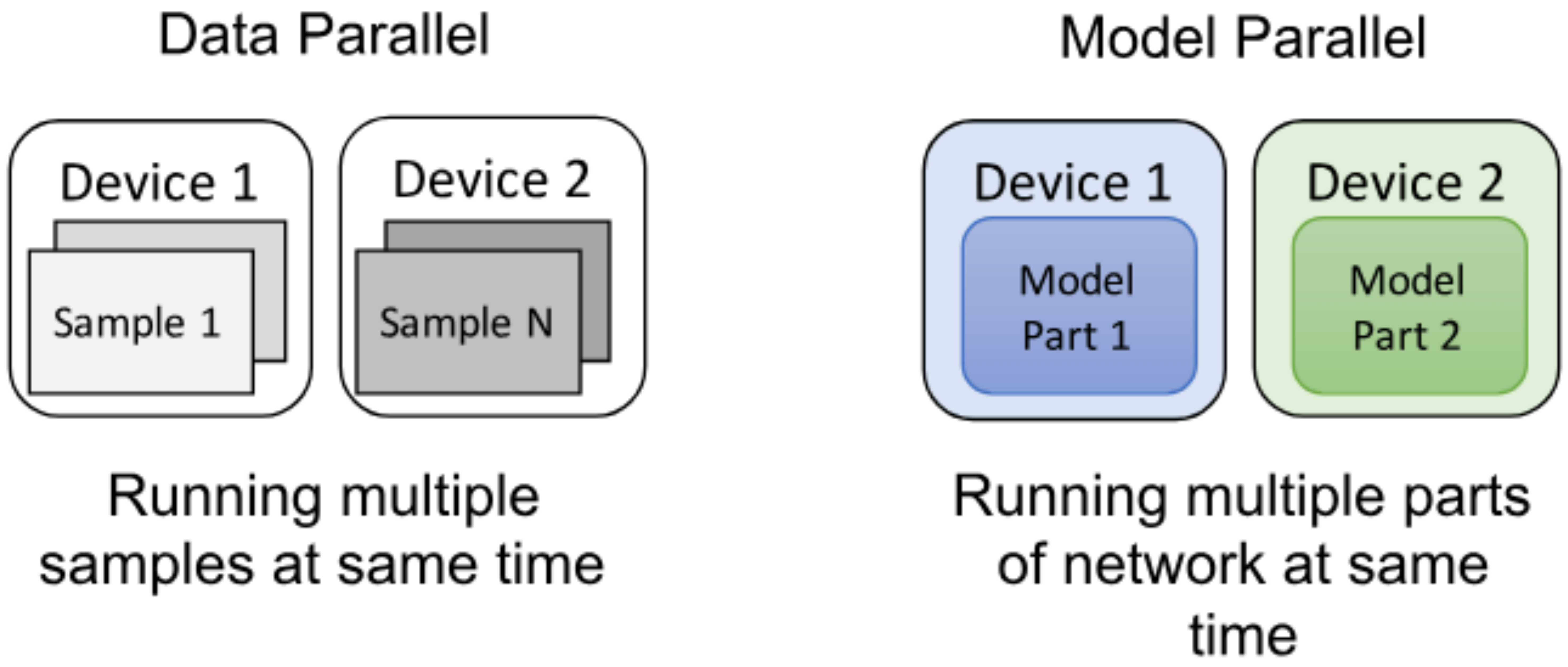


Image Source: https://blogs.nvidia.com/blog/what-is-nvidia-nvlink/

Scope

- Data Parallelism
- Model Parallelism
    - Pipeline parallel
    - Tensor parallel
    - Expert parallel



- **Next class.** Sequence parallelism, Automation, Gradient Compression, ZeRO

Image Source: https://www.cerebras.net/blog/data-model-pipeline-parallel-training-neural-networks/

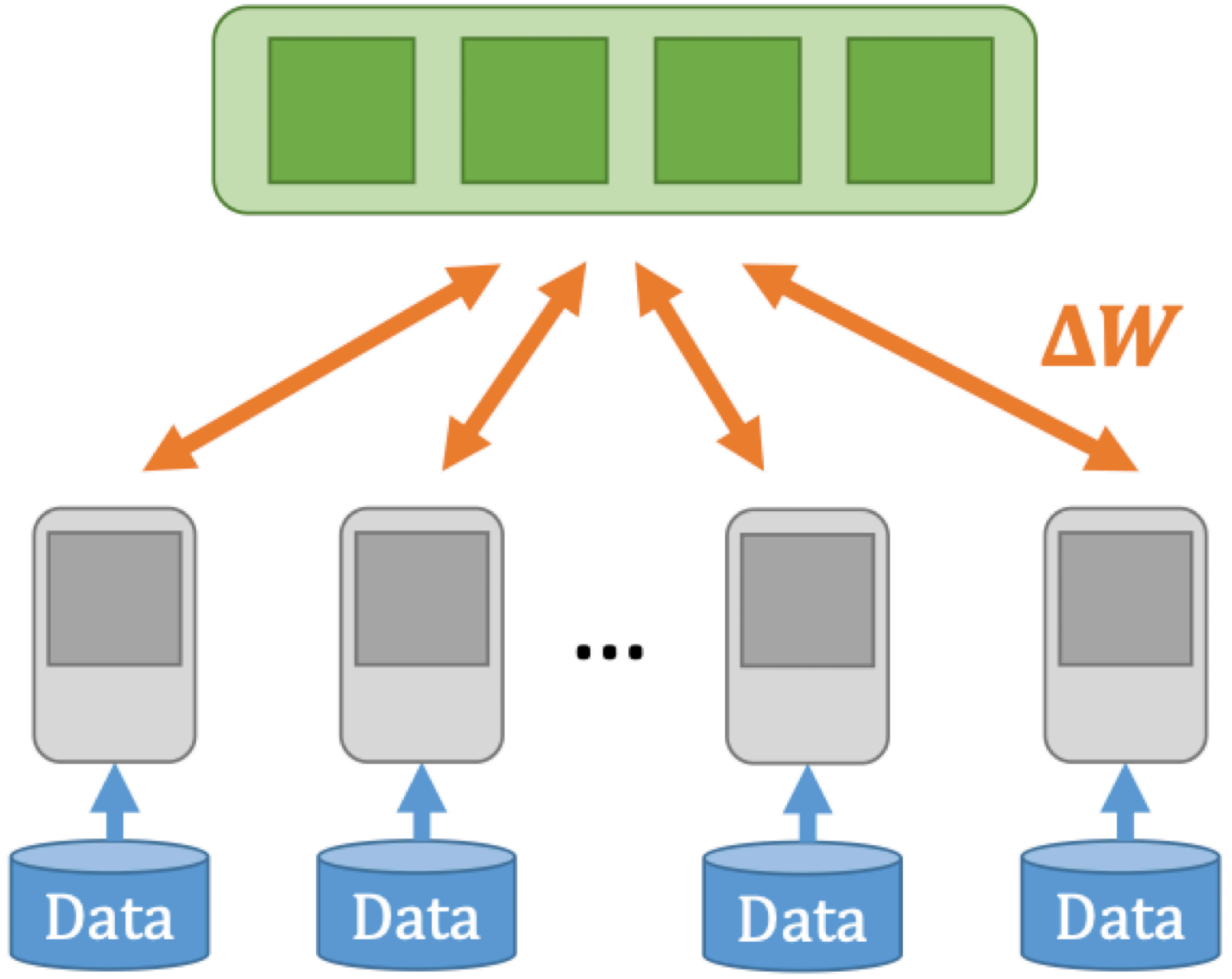
Data parallelism

Basic idea

- All workers share the **same model**, but have **different data**
  - In each step, i-th worker conducts:
    - Pull master weights w
    - Draw a data batch $B^{(i)}$
    - Compute the local gradient $\nabla w^{(i)}$
    - Push gradients to master
  - Master updates as:

$$w \leftarrow w - \eta\left(\sum \nabla w^{(i)}/K\right))$$

Master (Parameter server)
Coordinates the training



Workers (Computing)
GPU & Data storage

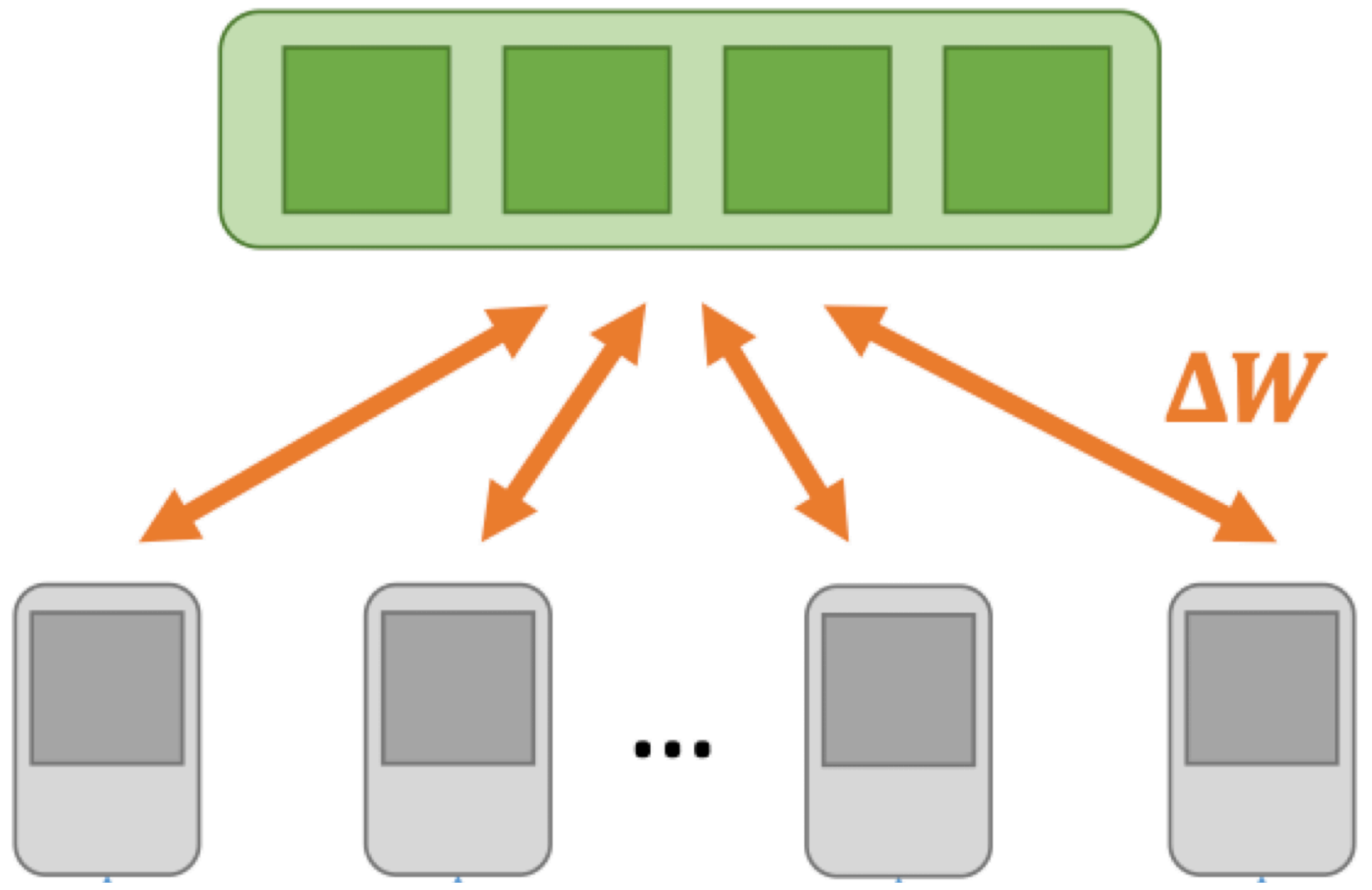
Lin et al., "Deep gradient compression: Reducing the communication bandwidth for distributed training" ICLR 2018

Basic idea

- **Data.** The whole dataset is usually **evenly split** among K workers
  - Possibly overlaps
    - Useful when some nodes are not reliable
    - Master can decide “indices” that each client will use
  - Can be dynamically fetched from a common data pool
    - Common when each node is a CPU, not a server

Basic idea

- **Communication.** Usually the key bottleneck
  - Worker requires:
    - Uplink: Gradient Size
    - Downlink: Model Size
  - Master requires:
    - Uplink: K * Model Size
    - Downlink: K * Gradient Size

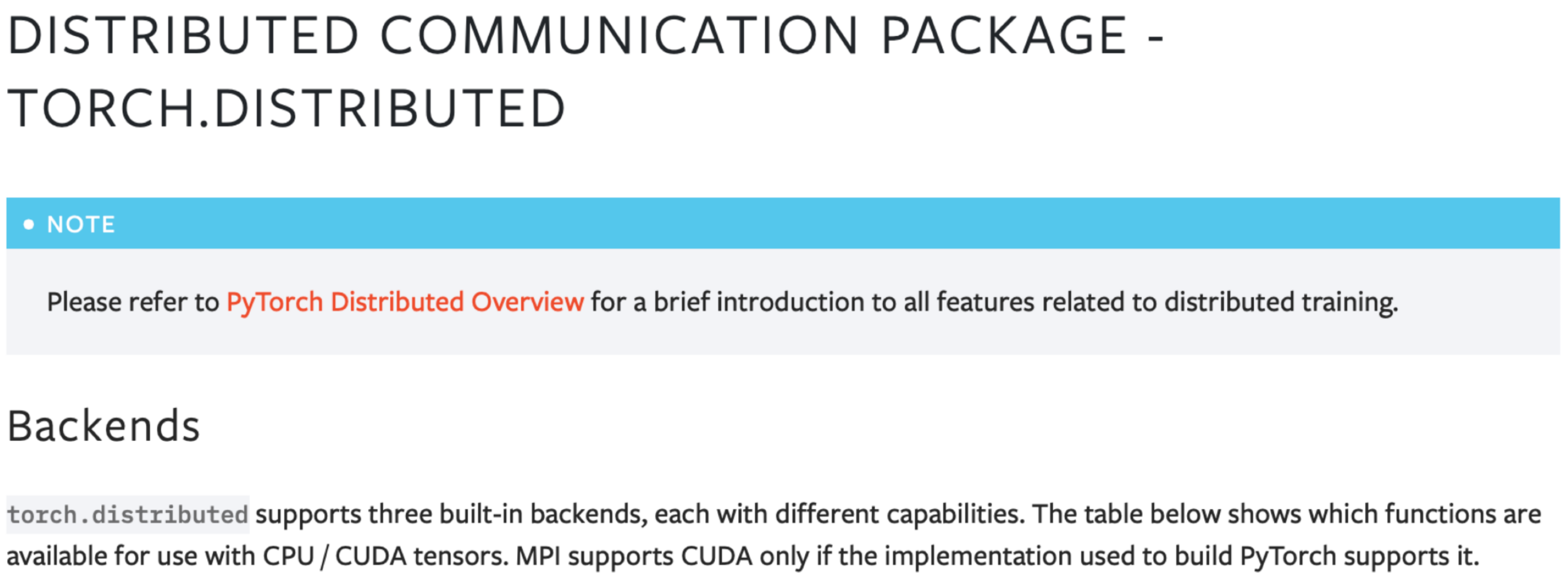


Basic idea

- Example. Training a ResNet-50 with V100s
  - Model parameters (or gradients) are ≈ 0.1GB
  - Suppose that we have 256 workers
  - If we use batch size 32:
    - Gradient computation. Takes ≈ 0.33 sec/step
    - Communication. Adds ≈ 0.16 sec/step
      - Assuming using 300GB/s bandwidth NVLink
  - => Communication adds 50% of the time!

Mitigating the comm. bottleneck

- **Idea.** Don't do **one-to-one** communication
    - Alternative communication strategies
        - Standardized as, e.g., Sockets / MPI



DISTRIBUTED COMMUNICATION PACKAGE - TORCH.DISTRIBUTED

• NOTE

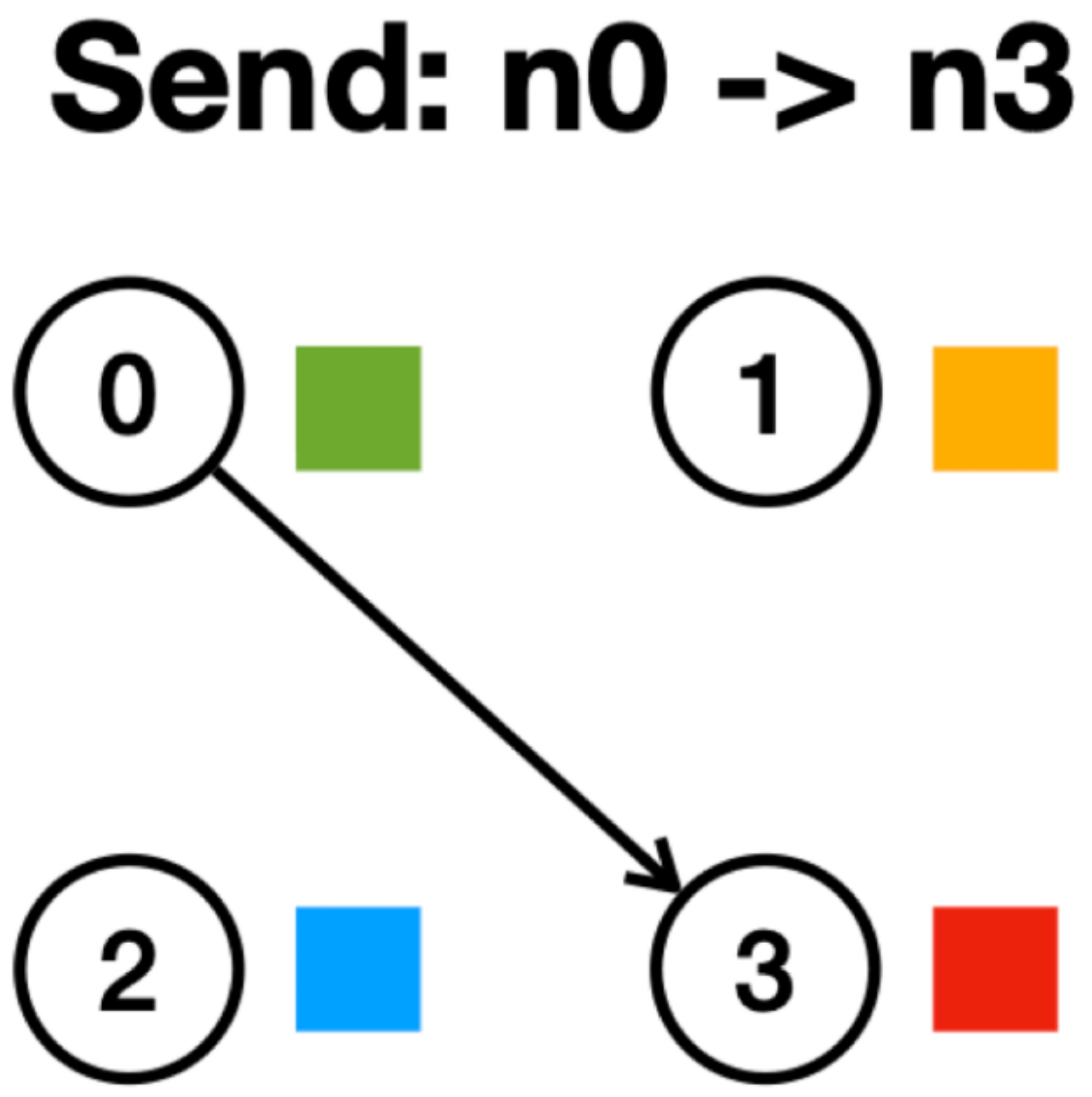
Please refer to PyTorch Distributed Overview for a brief introduction to all features related to distributed training.

Backends

`torch.distributed` supports three built-in backends, each with different capabilities. The table below shows which functions are available for use with CPU / CUDA tensors. MPI supports CUDA only if the implementation used to build PyTorch supports it.

One-to-One

- Transfer data from one process to another
  - **Send.** Send a tensor to another
  - **Receive.** Receive a tensor from another



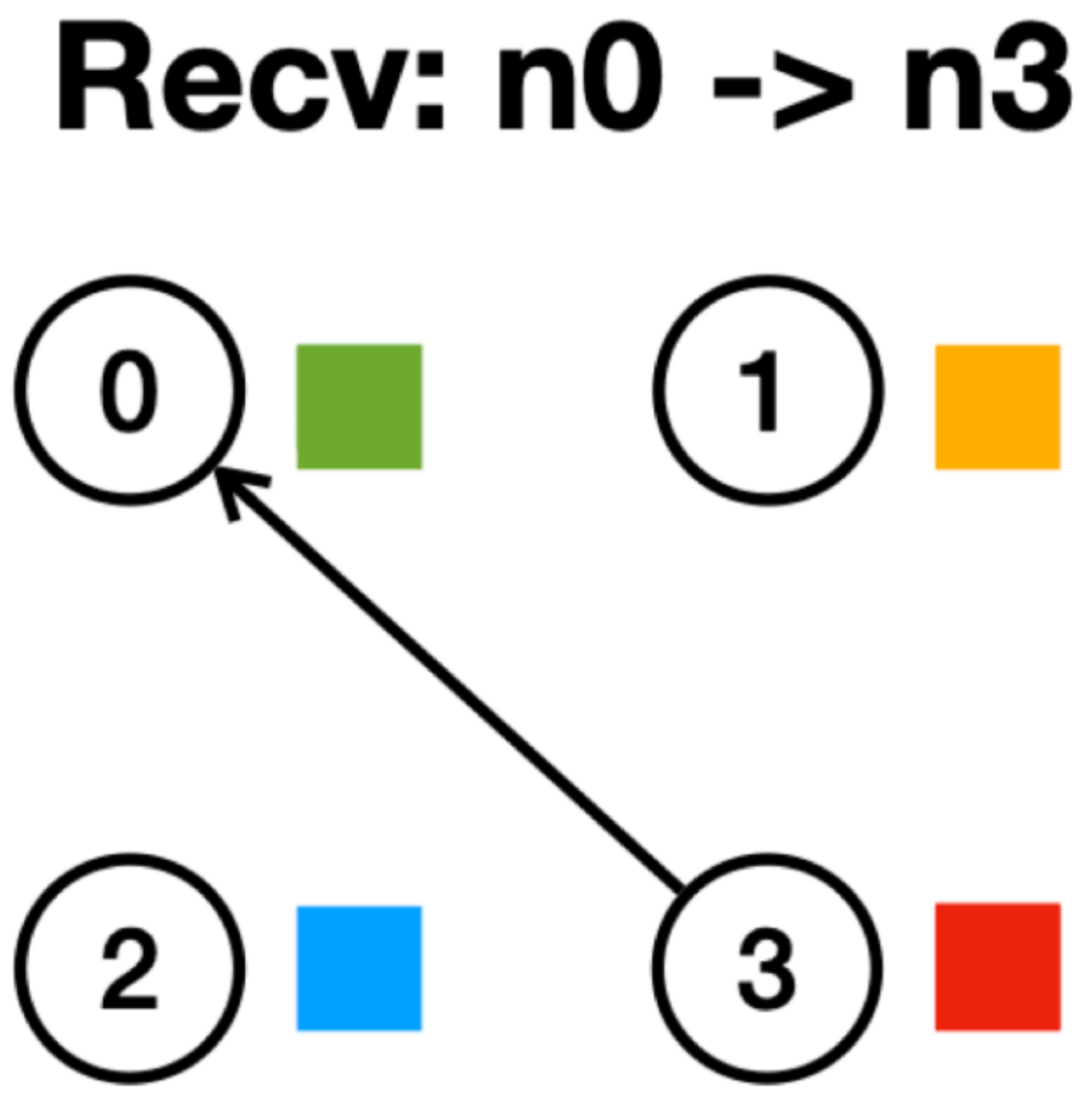
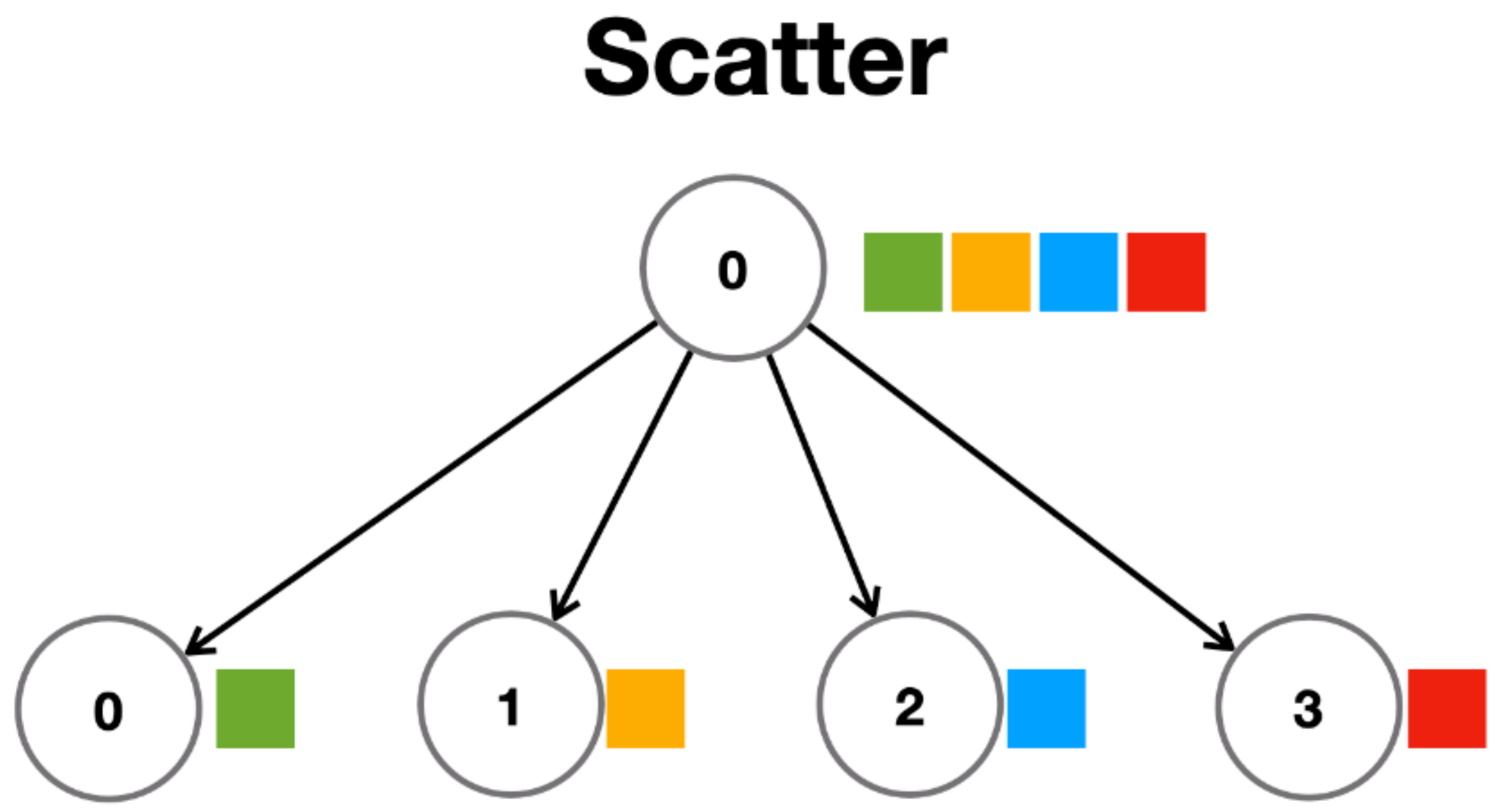


Image Source: efficientml.ai

One-to-Many

- Transfer data from one process to many other processes, or vice versa
    - **Scatter.** Send a tensor to many workers
    - **Gather.** Receive a tensor from many workers
        - Not many things we can do for these



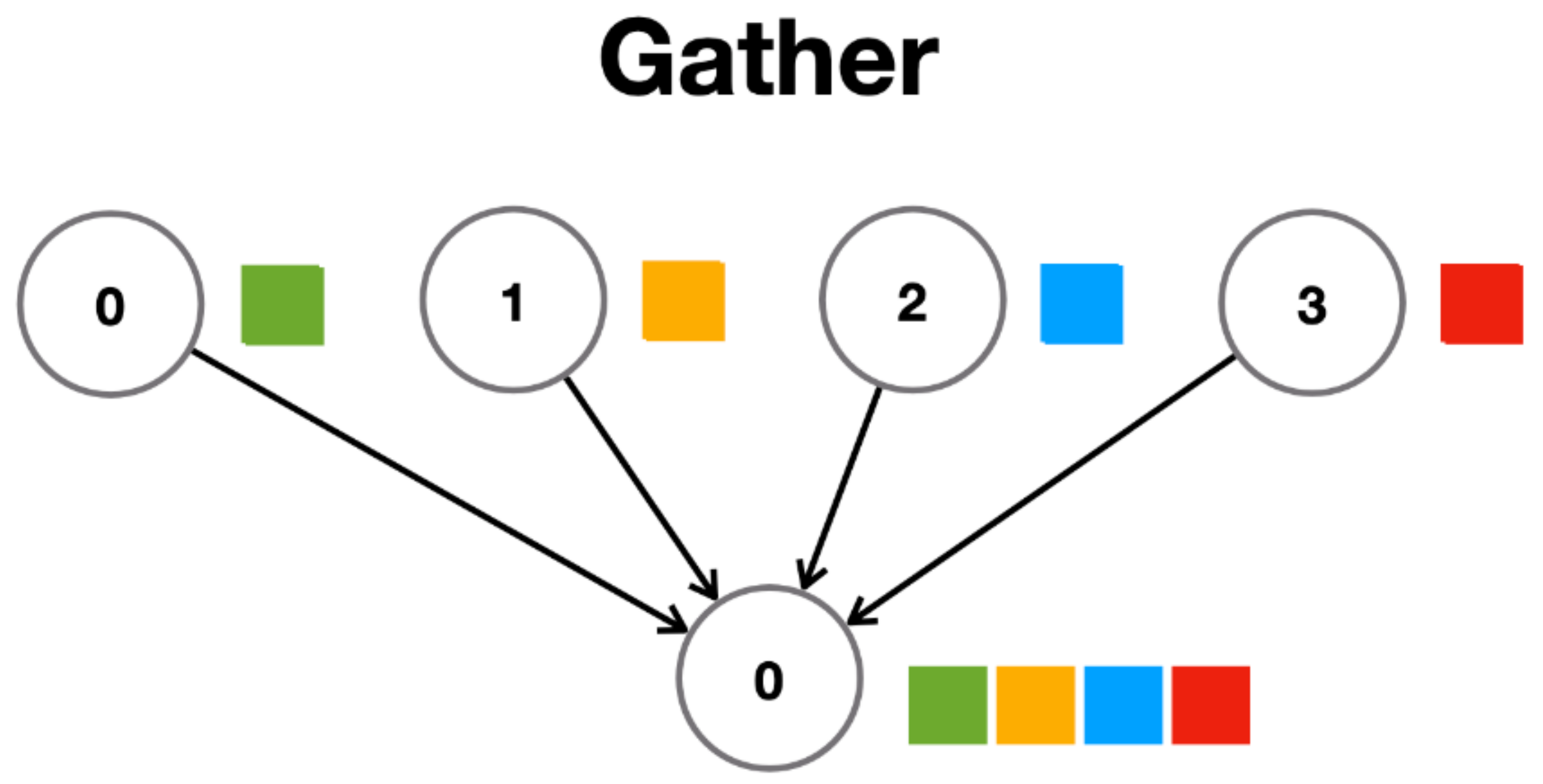
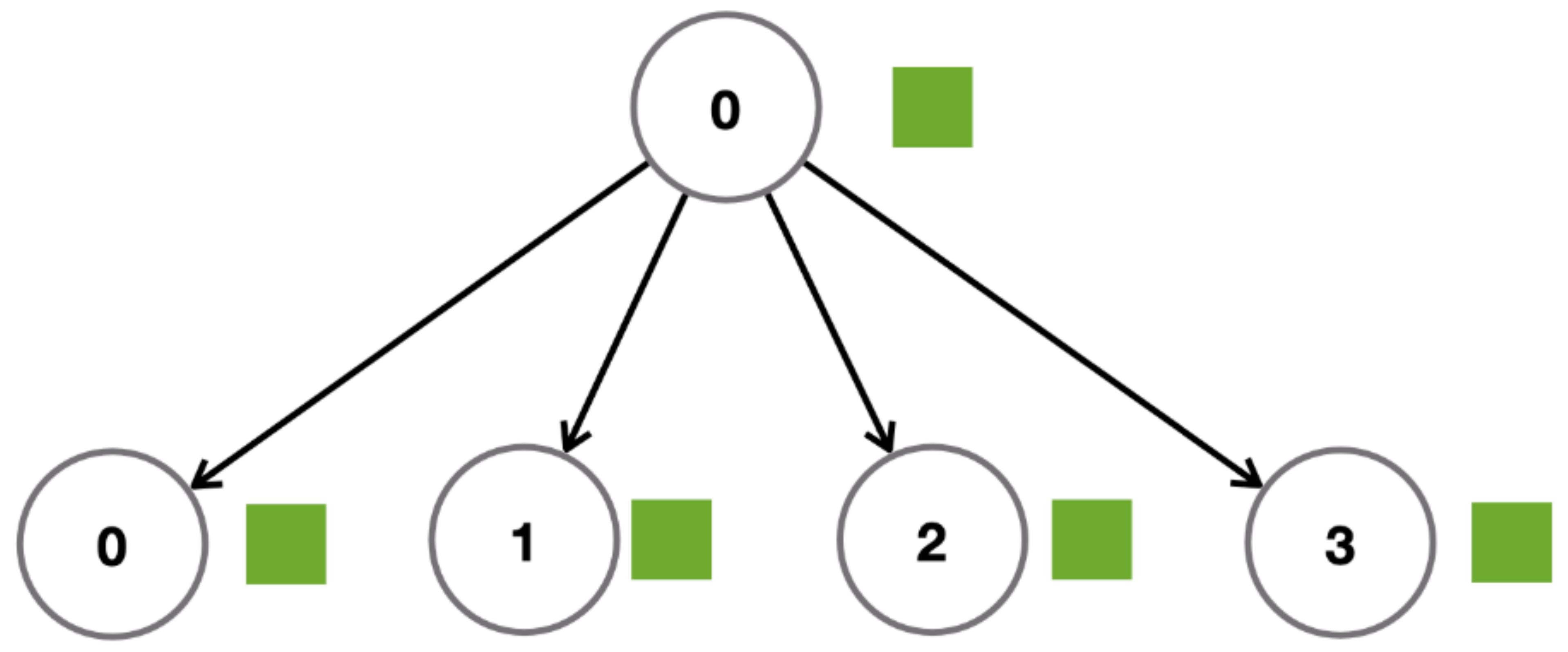


Image Source: efficientml.ai

One-to-Many

- Sometimes, we only care about a **single tensor** (our interest)
  - **Broadcast.** Send the same tensor to many workers
  - **Reduce.** Receive tensors, while averaging into a single tensor
    - Time $= O(1)$, Peak BW $= O(K)$, Total Comm $= O(K)$

Broadcast



Reduce

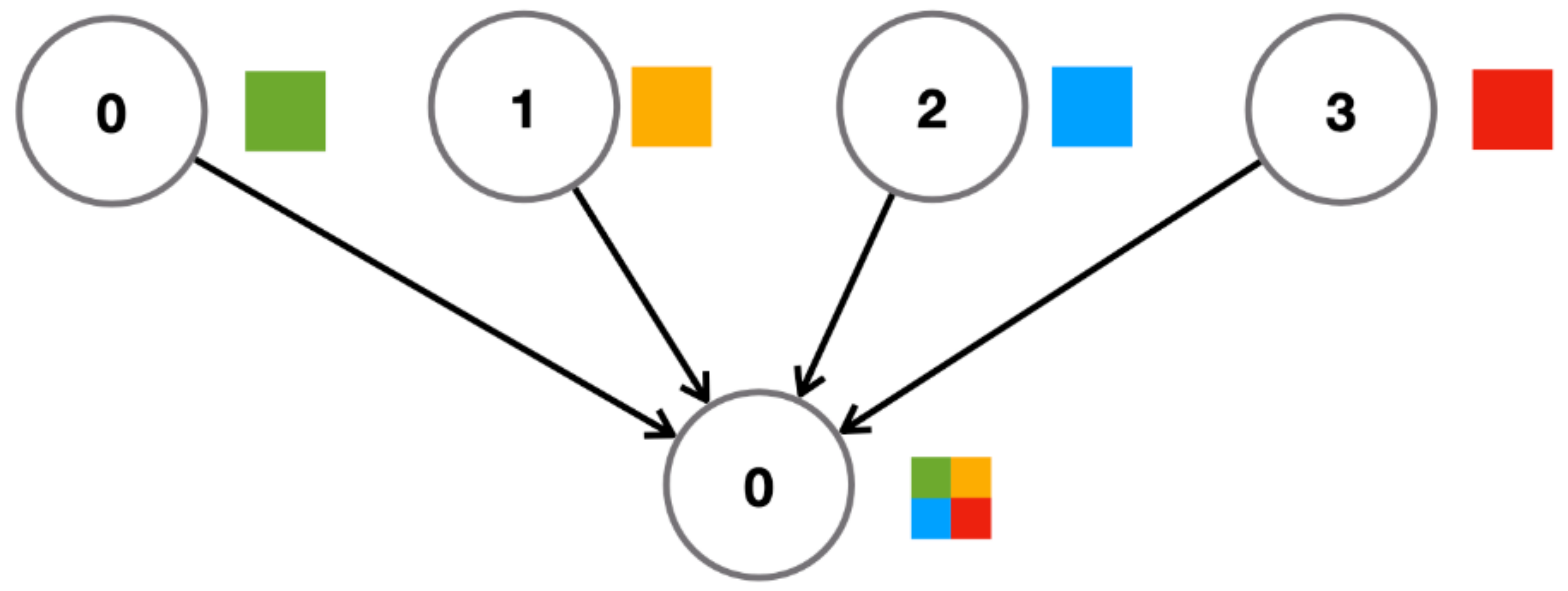


Image Source: efficientml.ai

One-to-Many

- **Idea.** Use inter-worker communication to avoid bottleneck at the master
- If we use a binary tree structure, each worker requires only
  - Up: Grad size + 2 * Model size
  - Down: 2 * Grad size + Model size
- Time $= O(\log K)$
  Peak BW $= O(1)$
  Total Comm $= O(K)$

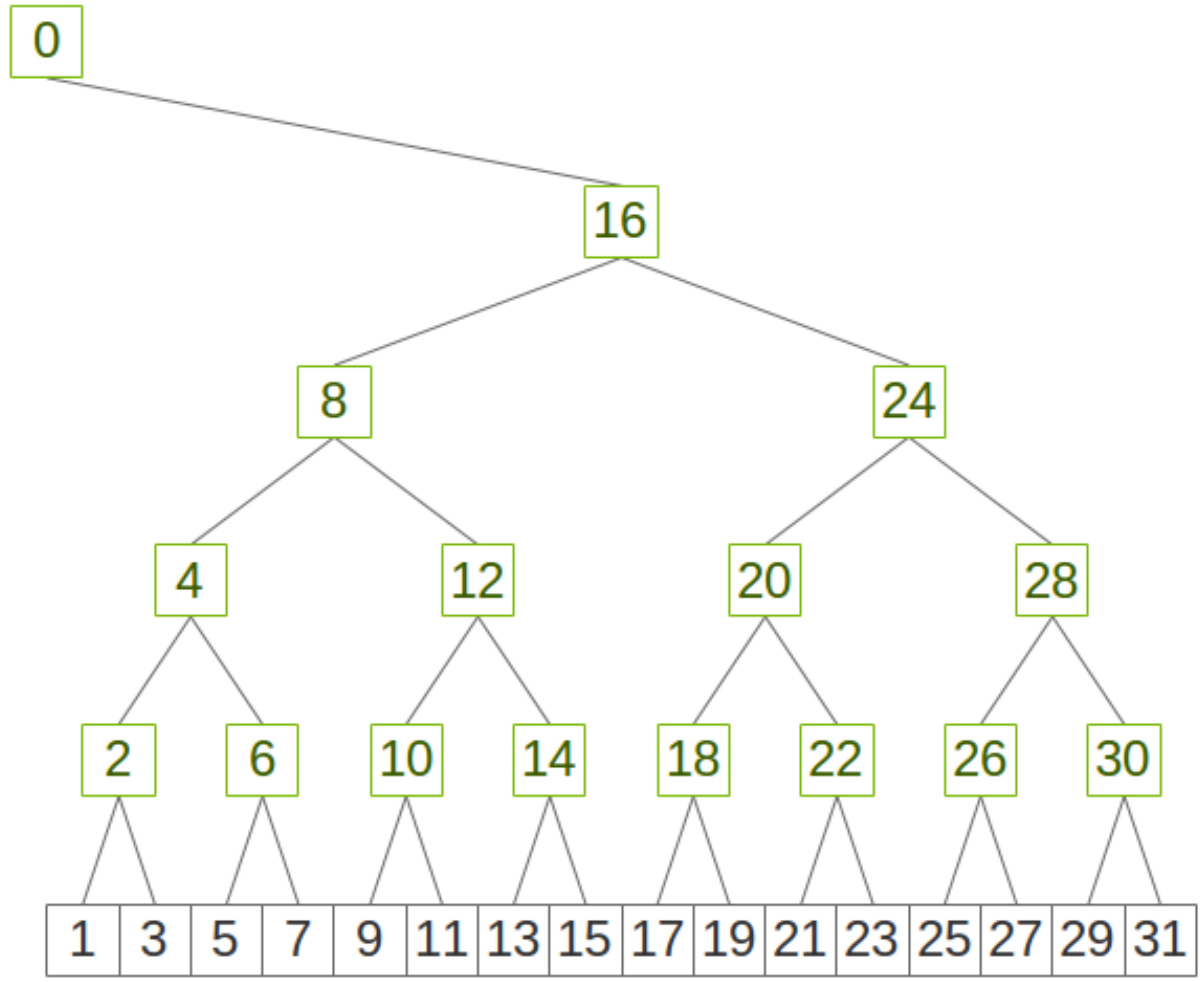
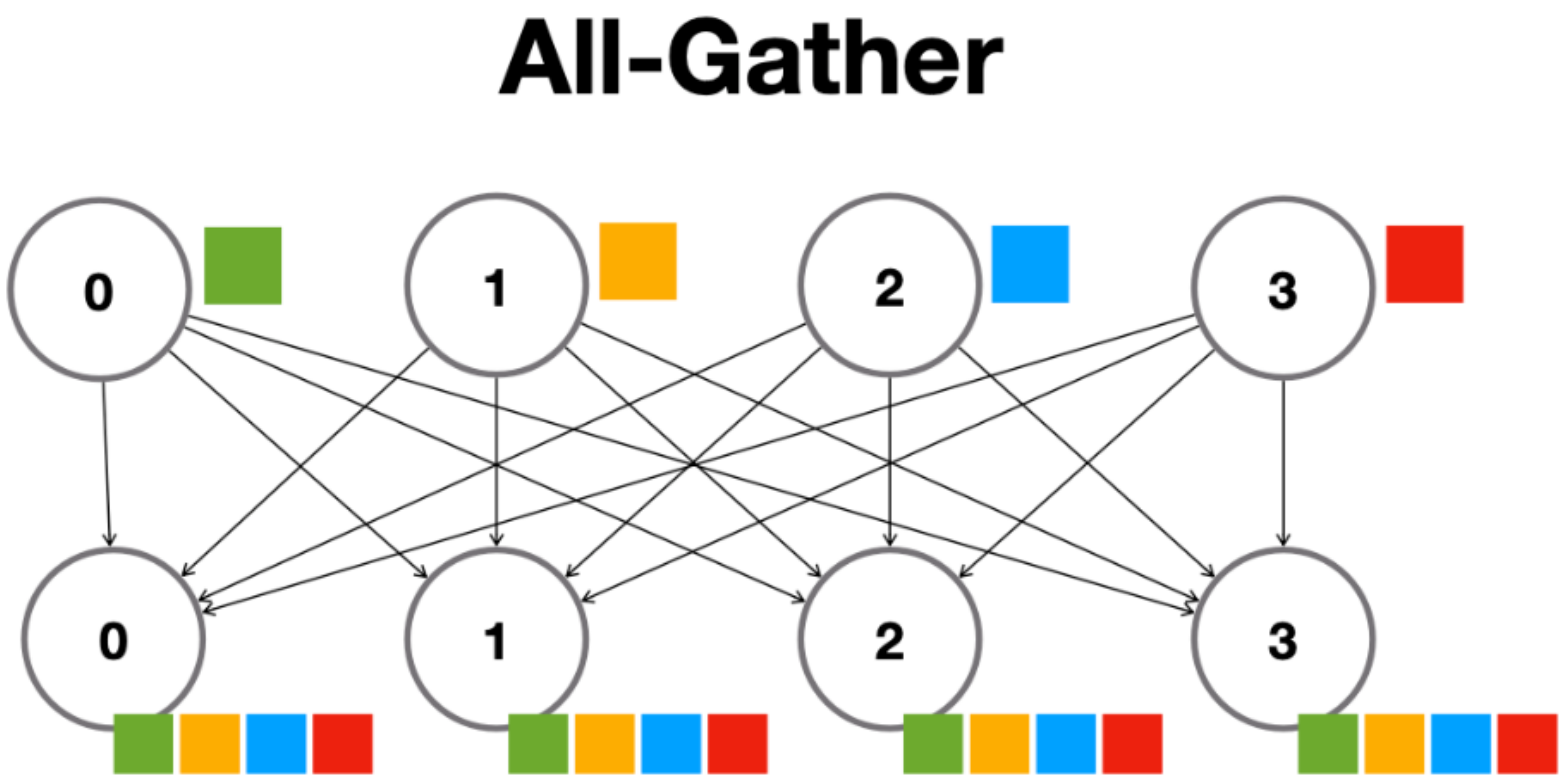


Image Source: efficientml.ai

Many-to-Many

- Transfer the data **without master**
  - **All-Gather.** Conduct gather on all workers
  - **All-Reduce.** Conduct reduce on all workers



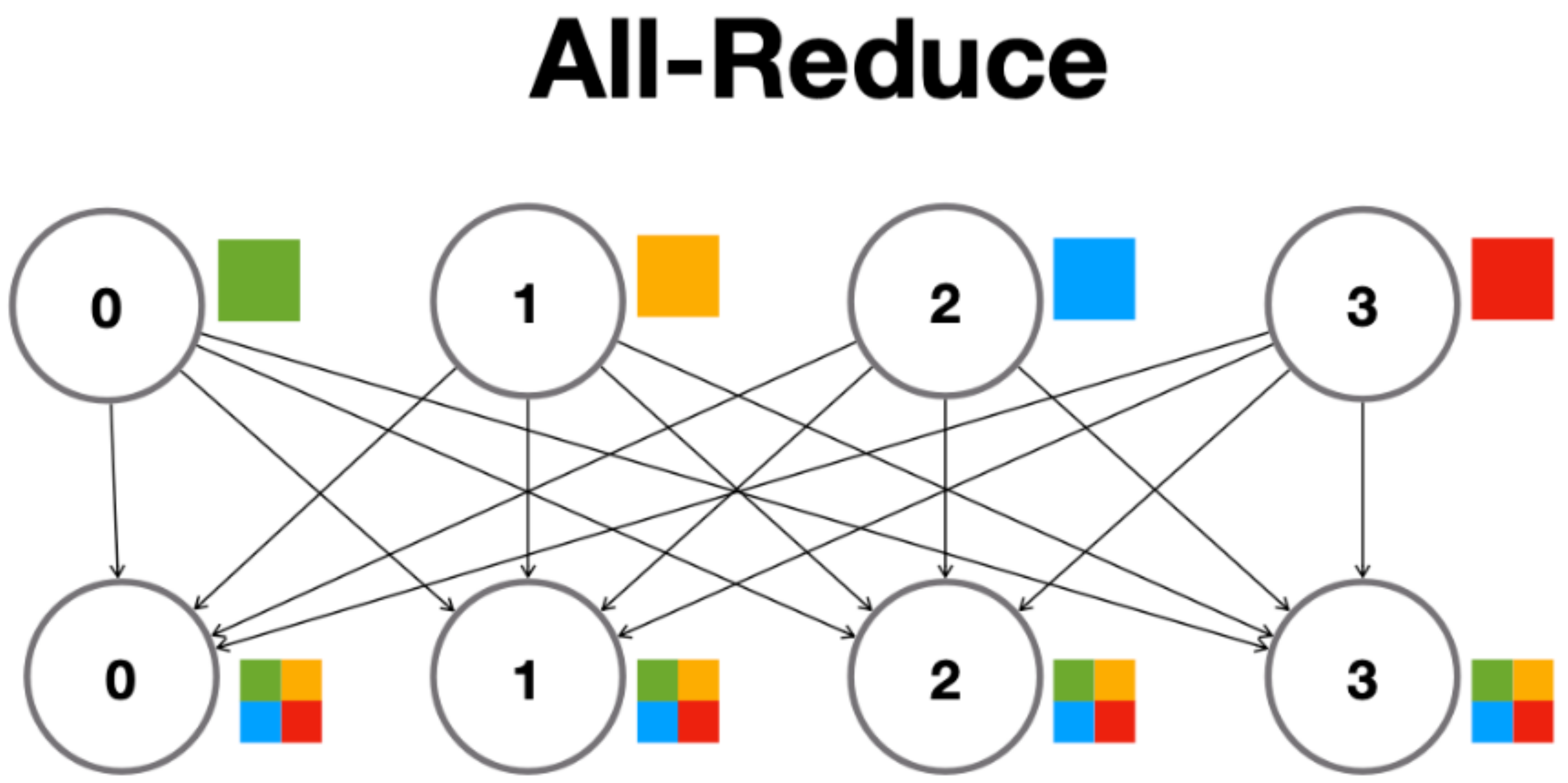


Image Source: efficientml.ai

Many-to-Many

- **Naïve.** Sequentially conduct reduce operations
  - Time $= O(K)$, Peak BW $= O(K)$, Total Comm $= O(K^2)$

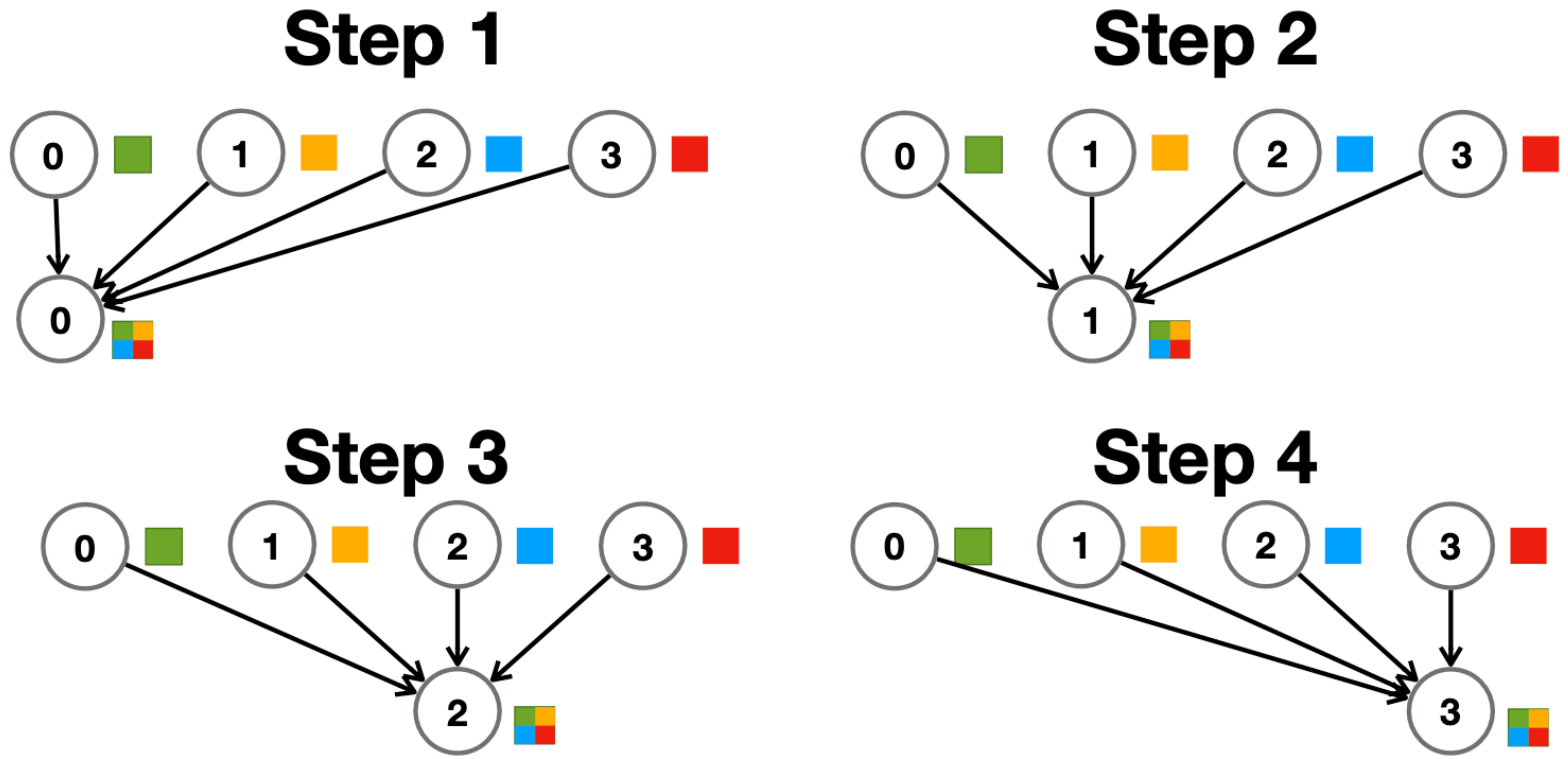
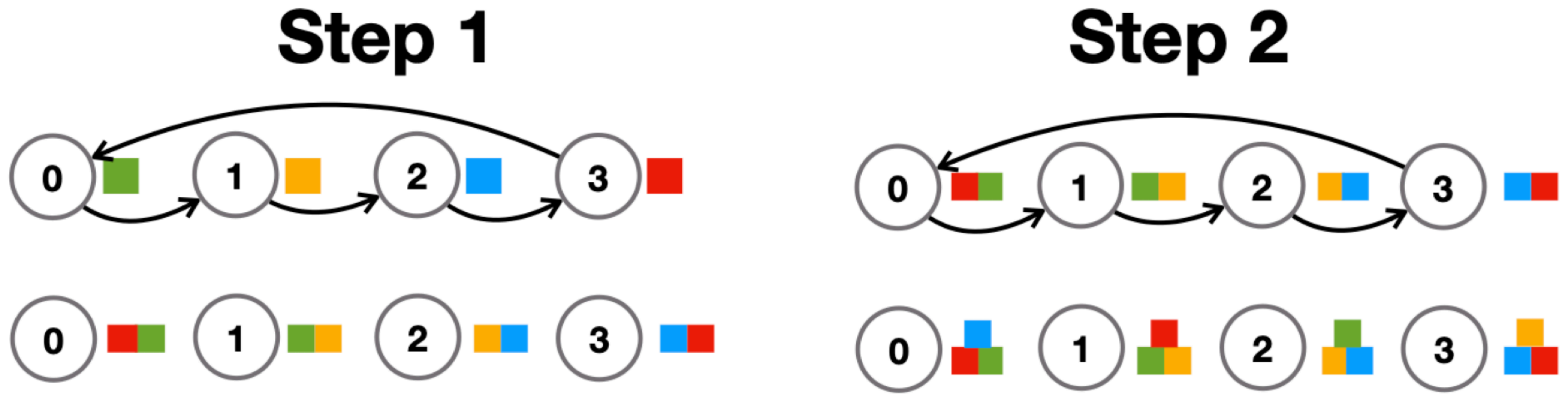


Image Source: efficientml.ai

Many-to-Many

- **Ring-AllReduce.** Utilize inter-worker communication
    - Time $= O(K)$, Peak BW $= O(1)$, Total Comm $= O(K^2)$



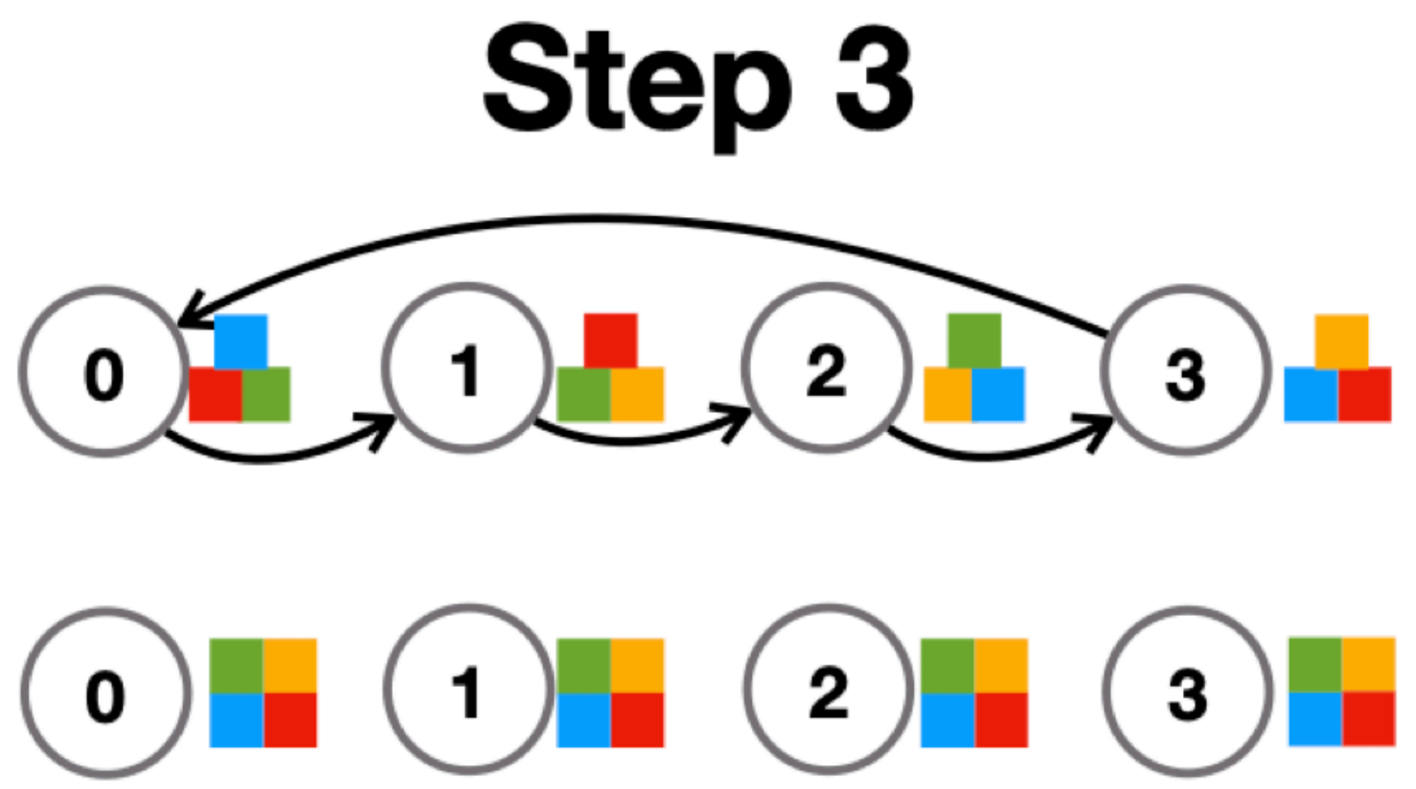
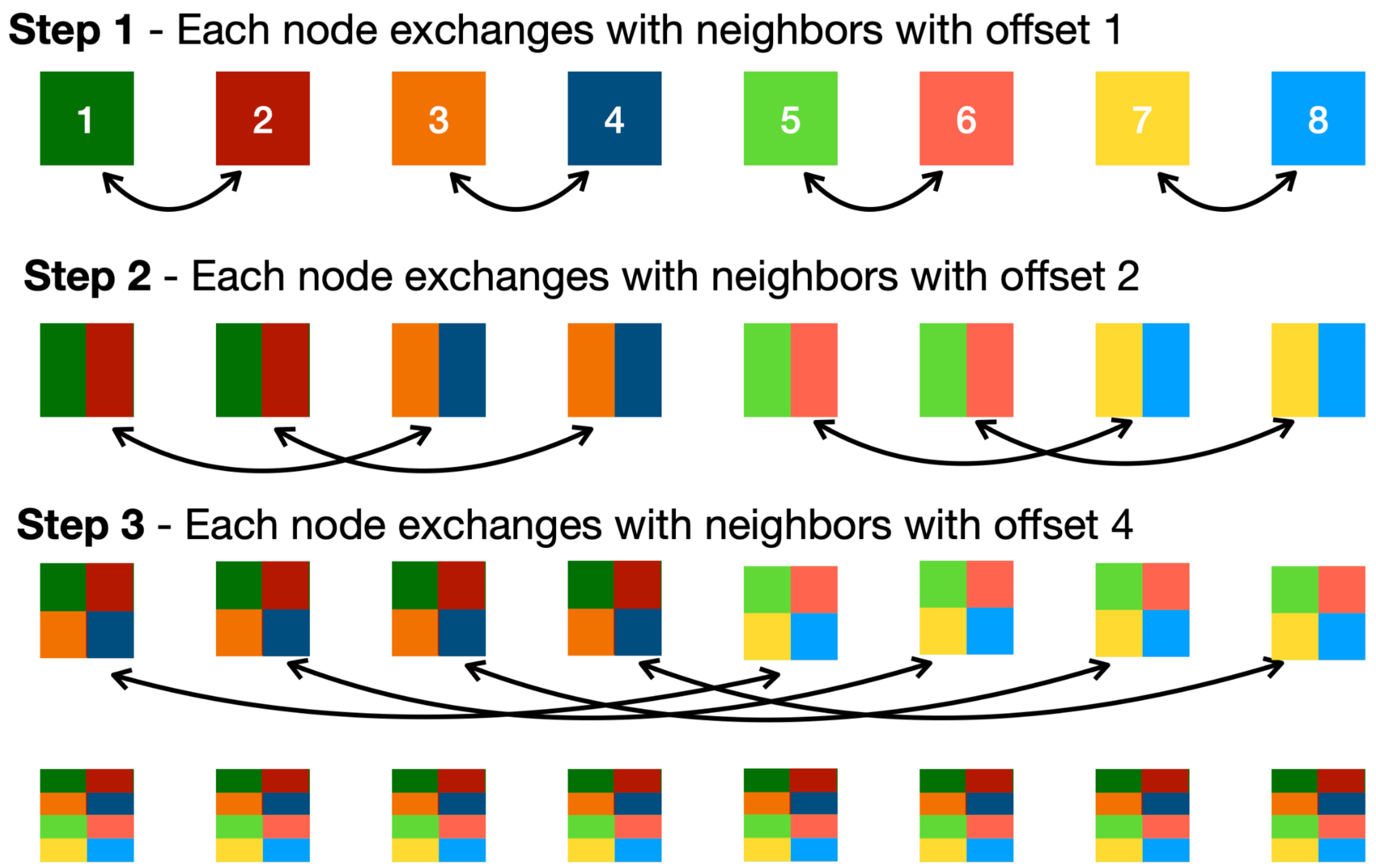


Image Source: efficientml.ai

Many-to-Many

- **Recursive Halving.** If inter-worker communication is dense,
  - Time $= O(\log K)$, Peak BW $= O(1)$, Total Comm $= O(K \log K)$

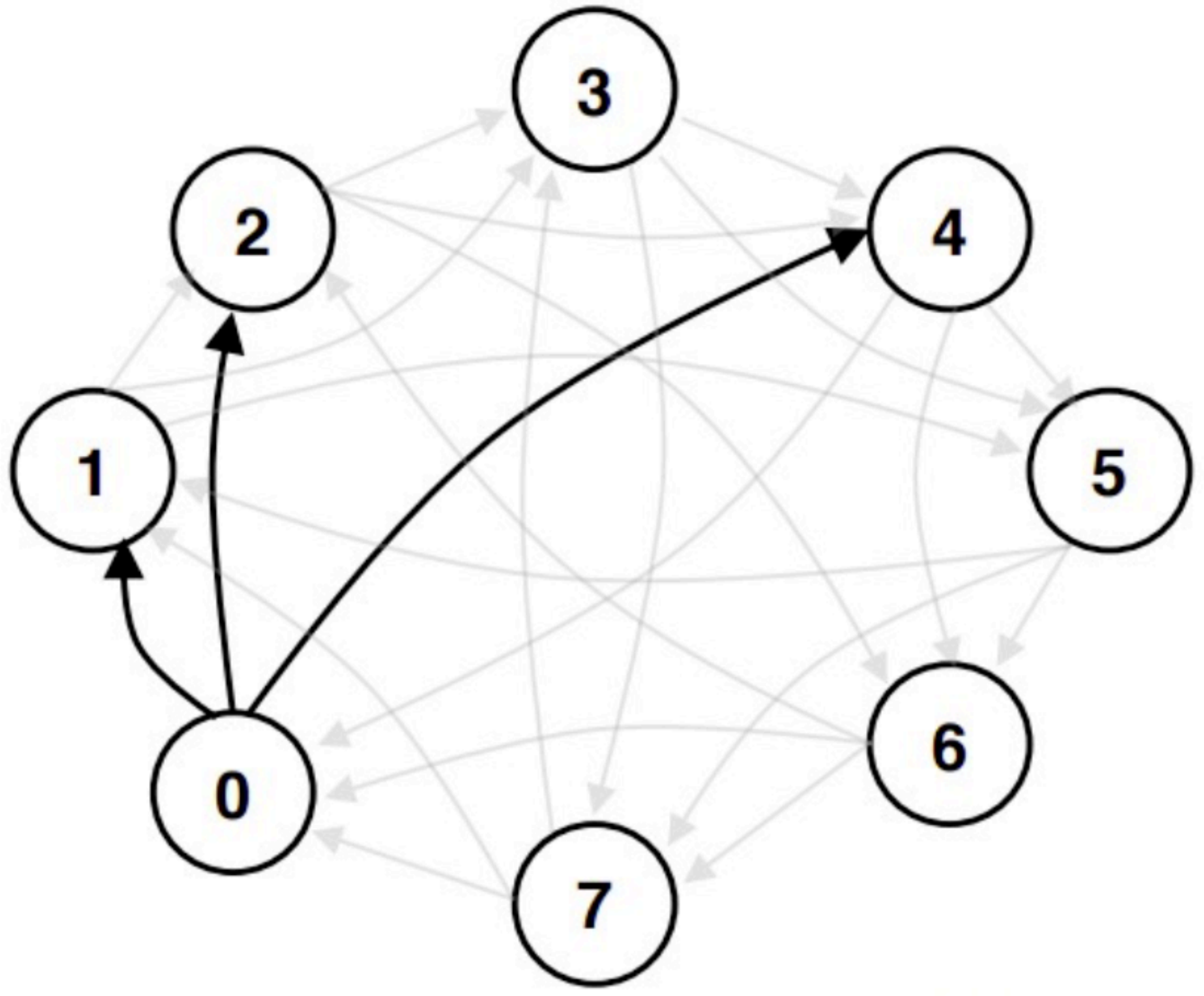


Step 1 - Each node exchanges with neighbors with offset 1
1
2
3
4
5
6
7
8
Step 2 - Each node exchanges with neighbors with offset 2
Step 3 - Each node exchanges with neighbors with offset 4

Image Source: efficientml.ai

Advanced Topics

- **Synchronization.** In practice, a full synchronization of GPUs is unnecessary
    - Can reduce the communication burden even further
    - Hogwild! (2011). Theoretically, one can still converge with updates based on gradients of slightly out-of-sync parameters
        - Stochastic gradient push (Assran et al., 2019)
        - Grouped all-reduce with intermittent group swapping (Li et al., 2021)

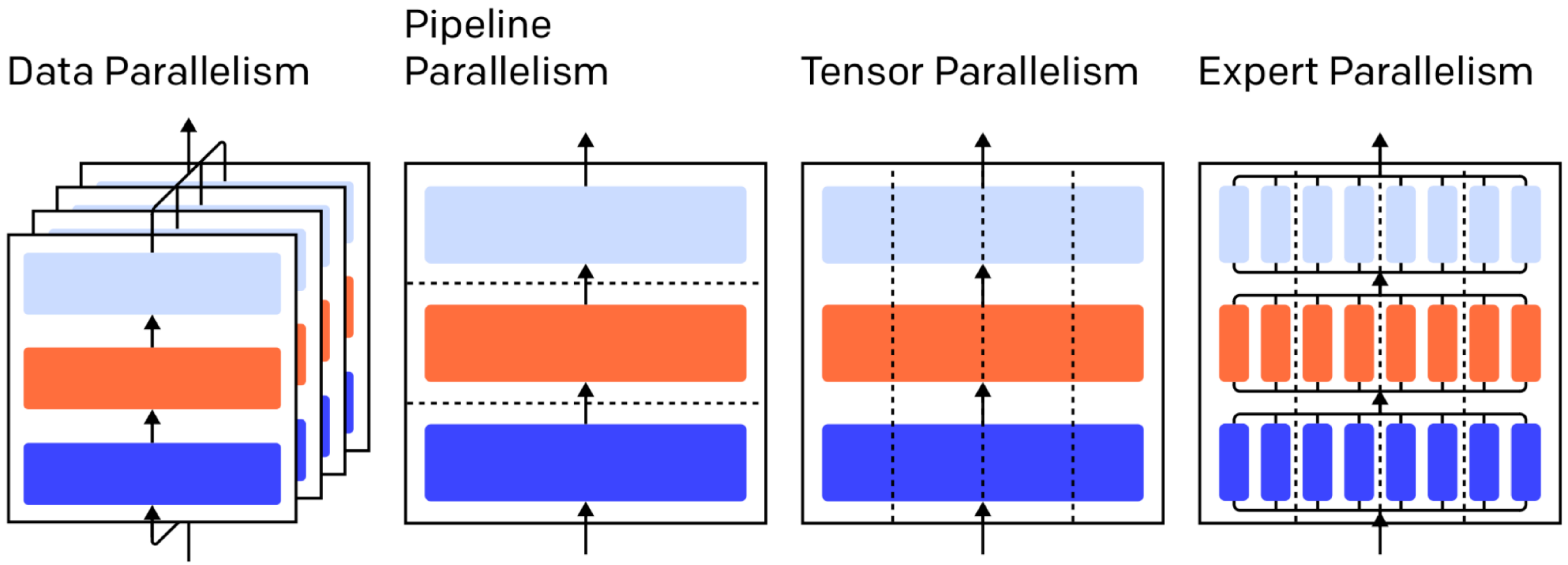


Niu et al., "HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," NeurIPS 2011
Assran et al., "Stochastic Gradient Push for Distributed Deep Learning" ICML 2019
Li et al., "Breaking (Global) Barriers in Parallel Stochastic Optimization with Wait-Avoiding Group Averaging," IEEE TDPS 2021

Model parallelism

Basic idea

- All workers share the **same data**, but have **different model parts**
  - **Pipeline.** Sequential processing
  - **Tensor.** Parallel processing
  - **Expert.** Conditional processing



Pipeline parallelism

- Each worker has different layers
  - Thus, less burden for
    - **Memory.** Keeping the parameters and activations on RAM
    - **Computation.** Computing forward & backward

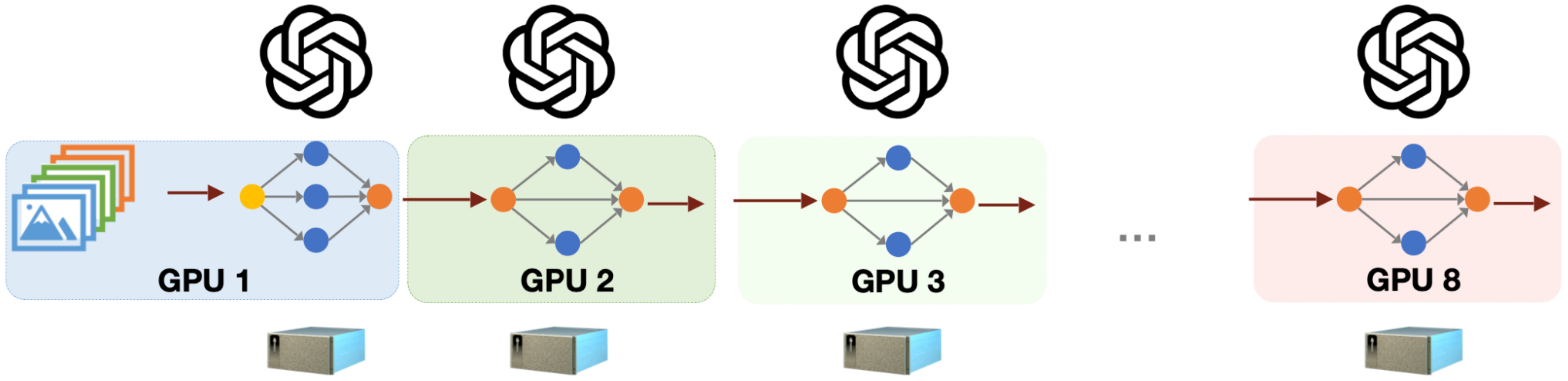
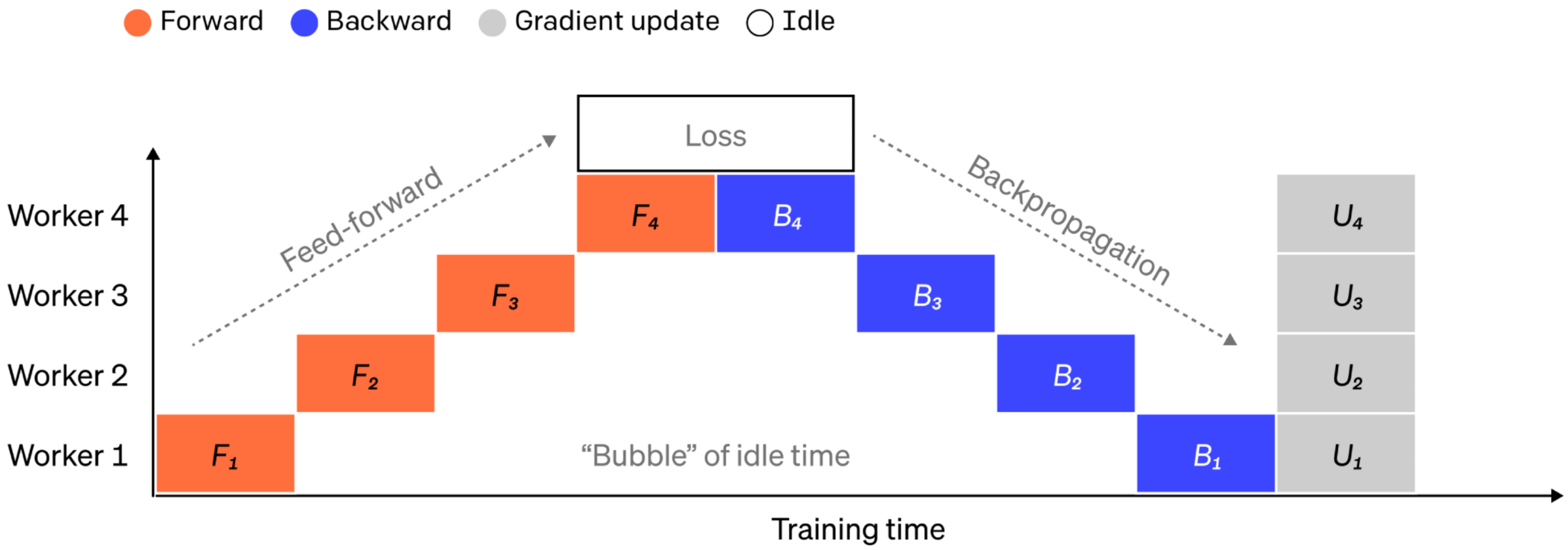


Image Source: efficientml.ai

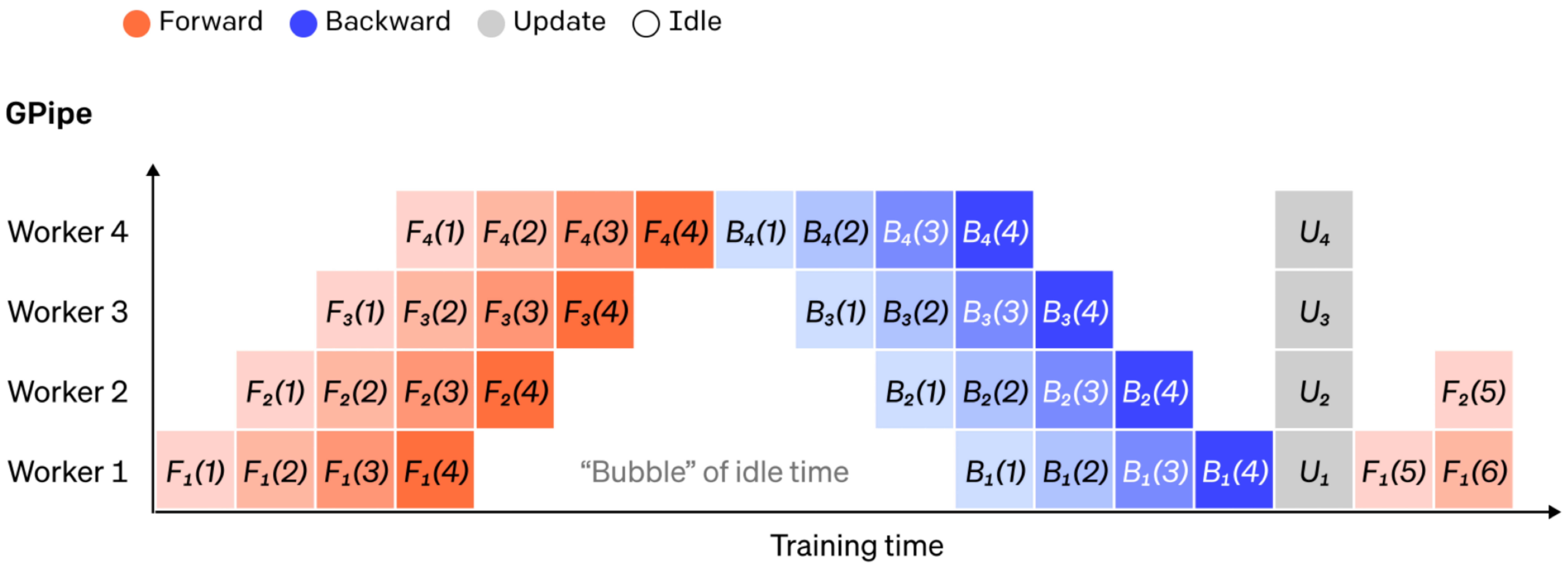
Pipeline parallelism

- **Naïve.** Simply activate all workers in series
  - Low GPU utilization ratio
  - No speedup (slower!)



GPipe (2019)

- Split a single batch into multiple **micro-batches**
    - Process micro-batches without gradient updates in between



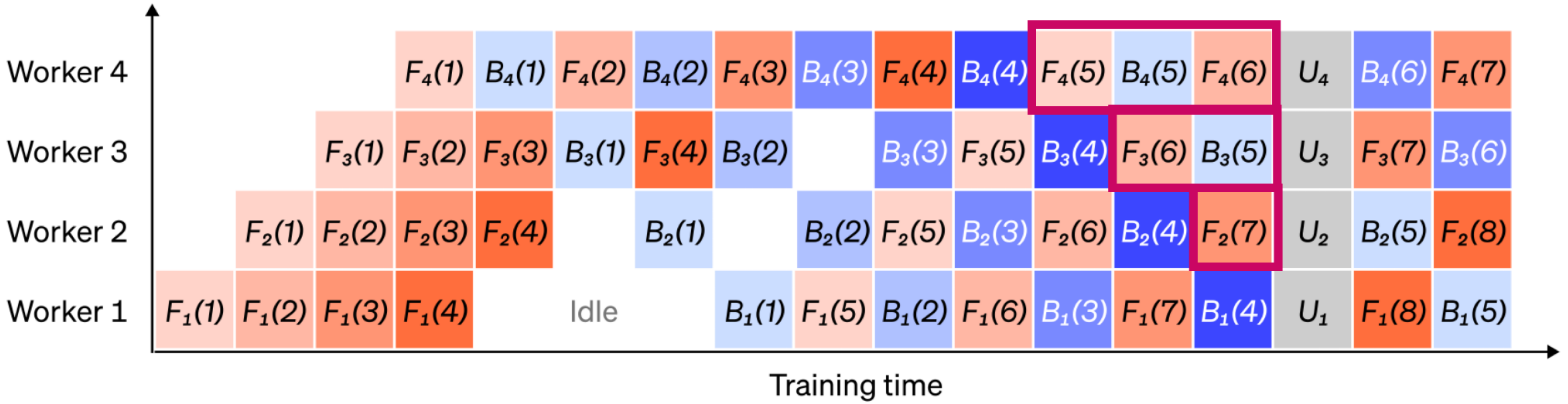
● Forward
● Backward
● Update
○ Idle
GPipe
Worker 4
Worker 3
Worker 2
Worker 1
$F_4(1)$ $F_4(2)$ $F_4(3)$ $F_4(4)$ $B_4(1)$ $B_4(2)$ $B_4(3)$ $B_4(4)$ U_4
$F_3(1)$ $F_3(2)$ $F_3(3)$ $F_3(4)$ $B_3(1)$ $B_3(2)$ $B_3(3)$ $B_3(4)$ U_3
$F_2(1)$ $F_2(2)$ $F_2(3)$ $F_2(4)$ $B_2(1)$ $B_2(2)$ $B_2(3)$ $B_2(4)$ U_2 $F_2(5)$
$F_1(1)$ $F_1(2)$ $F_1(3)$ $F_1(4)$ $B_1(1)$ $B_1(2)$ $B_1(3)$ $B_1(4)$ U_1 $F_1(5)$ $F_1(6)$
"Bubble" of idle time
Training time

Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism" NeurIPS 2019

PipeDream (2019)

- Interleave some out-of-sync ("stale") operations from succeeding batch
    - called **inter-batch pipelining**
    - PipeDream automatized such interleaving

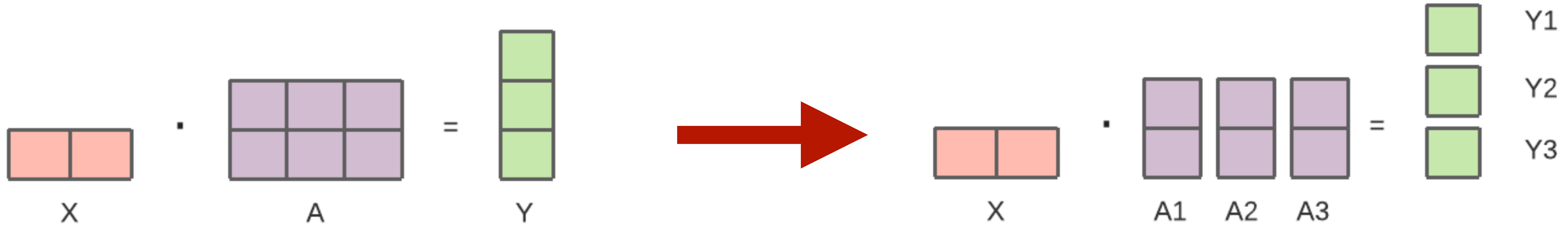
PipeDream


Worker 4
Worker 3
Worker 2
Worker 1
$F_4(1)$ $B_4(1)$ $F_4(2)$ $B_4(2)$ $F_4(3)$ $B_4(3)$ $F_4(4)$ $B_4(4)$ $F_4(5)$ $B_4(5)$ $F_4(6)$ U_4 $B_4(6)$ $F_4(7)$
$F_3(1)$ $F_3(2)$ $F_3(3)$ $B_3(1)$ $F_3(4)$ $B_3(2)$ $B_3(3)$ $F_3(5)$ $B_3(4)$ $F_3(6)$ $B_3(5)$ U_3 $F_3(7)$ $B_3(6)$
$F_2(1)$ $F_2(2)$ $F_2(3)$ $F_2(4)$ $B_2(1)$ $B_2(2)$ $F_2(5)$ $B_2(3)$ $F_2(6)$ $B_2(4)$ $F_2(7)$ U_2 $B_2(5)$ $F_2(8)$
$F_1(1)$ $F_1(2)$ $F_1(3)$ $F_1(4)$ Idle $B_1(1)$ $F_1(5)$ $B_1(2)$ $F_1(6)$ $B_1(3)$ $F_1(7)$ $B_1(4)$ U_1 $F_1(8)$ $B_1(5)$
Training time

Narayanan et al., "PipeDream: Generalized pipeline parallelism for DNN training" SOSP 2019

Tensor parallelism

- Make the operations parallel by **partitioning each tensor**
    - Less bubble
- **Key challenge.** The output becomes sharded as well



Tensor parallelism

- **Idea.** Splitting direction matters!
- Suppose we have a matmul $Y = \sigma(XA)$
- Splitting by row. We conduct

$$X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}, \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$$

  - Thus, we have

$$Y = \sigma(X_1 A_1 + X_1 A_2)$$

  - The output requires all-reduce before activation

Tensor parallelism

- Splitting by column. We conduct

$$X = X, \quad A = \begin{bmatrix} A_1 & A_2 \end{bmatrix}$$

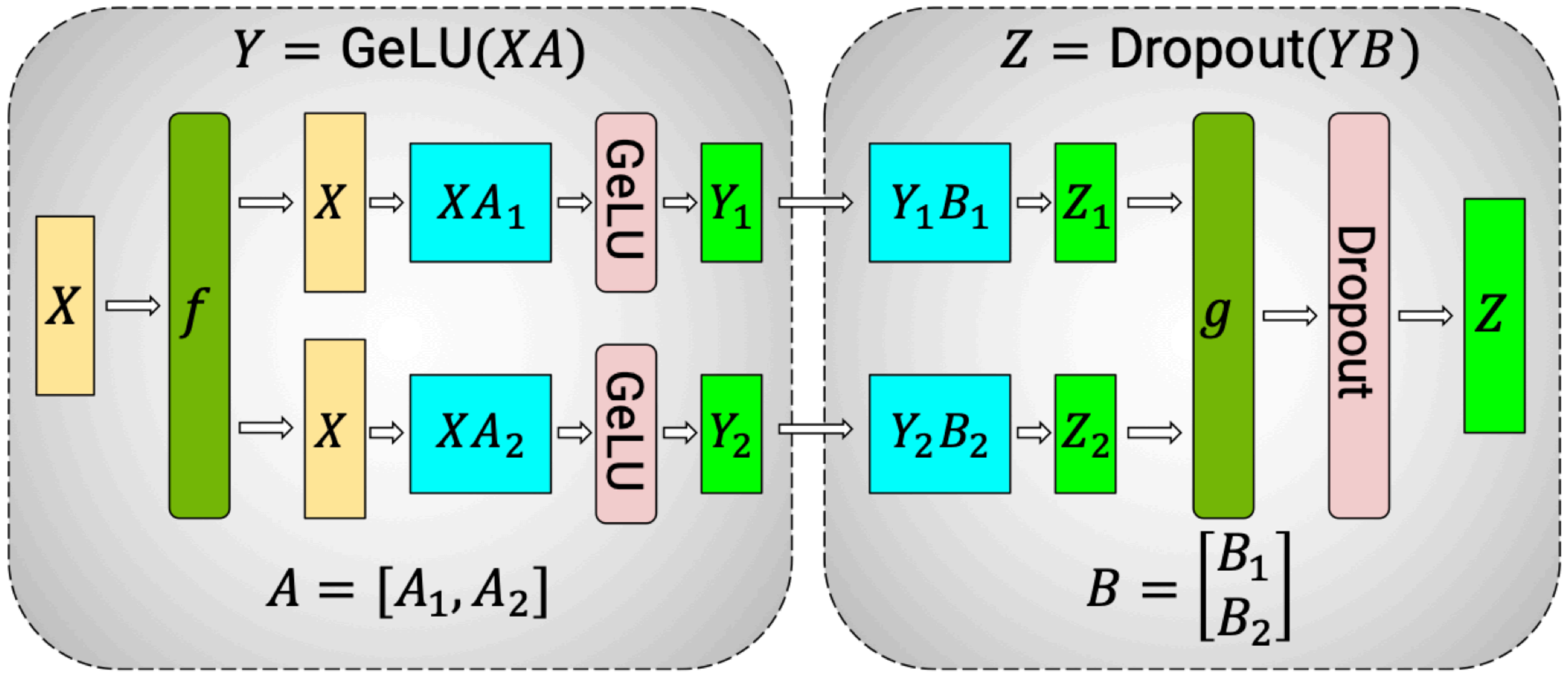
  - Then, we have

$$Y = \begin{bmatrix} \sigma(XA_1) & \sigma(XA_2) \end{bmatrix}$$

  - The output does not require all-reduce
    - But Y are sharded, forcing row-splitting in the next layer

Megatron-LM (2019)

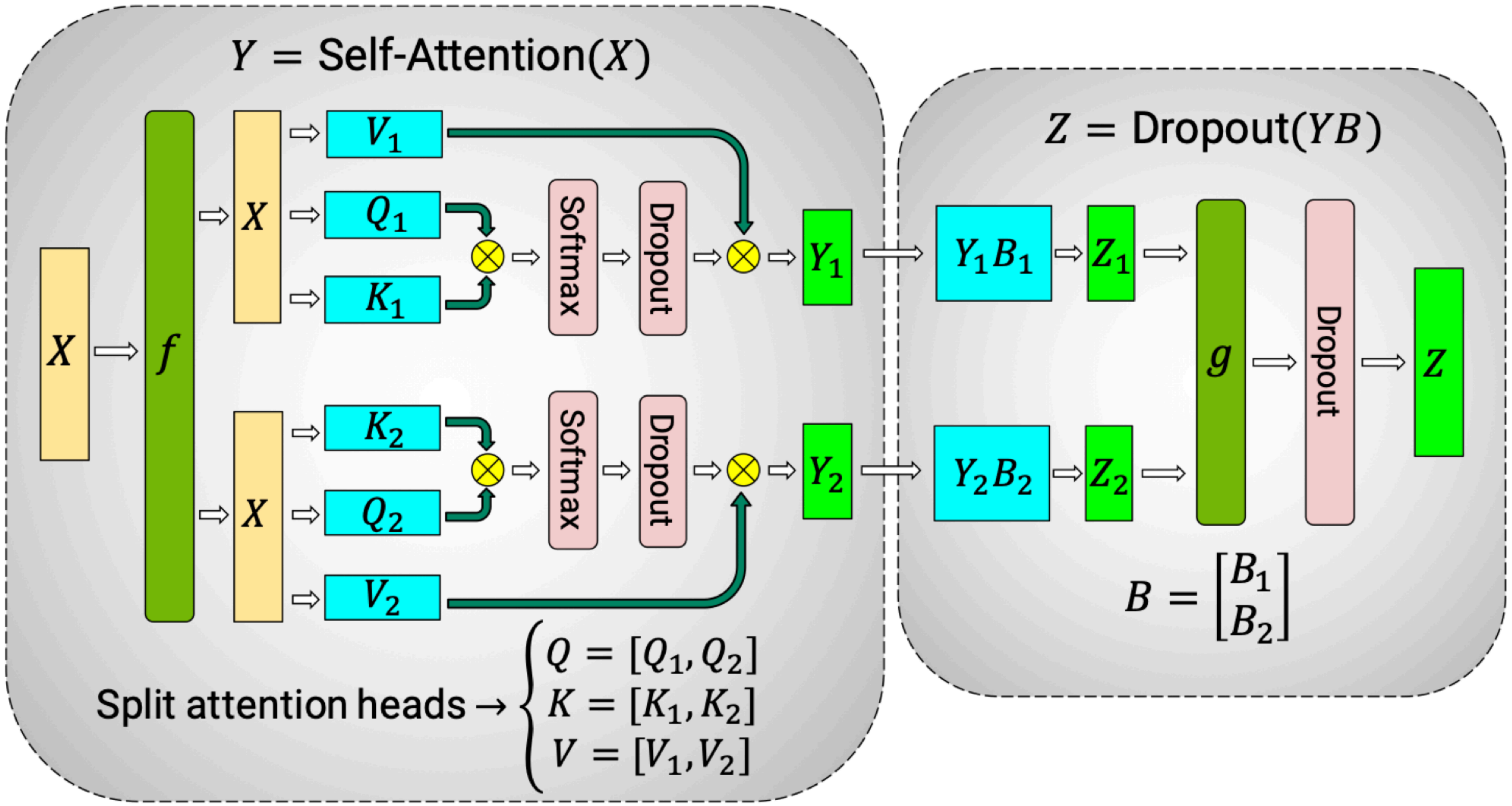
- A recipe customized for transformer-based LLMs
- For FFNs, conduct column-split first and then row-split
  - f : identity in forward-pass, all-reduce in backward pass
  - g : all-reduce in forward pass, identity in backward pass



(a) MLP.

Megatron-LM (2019)

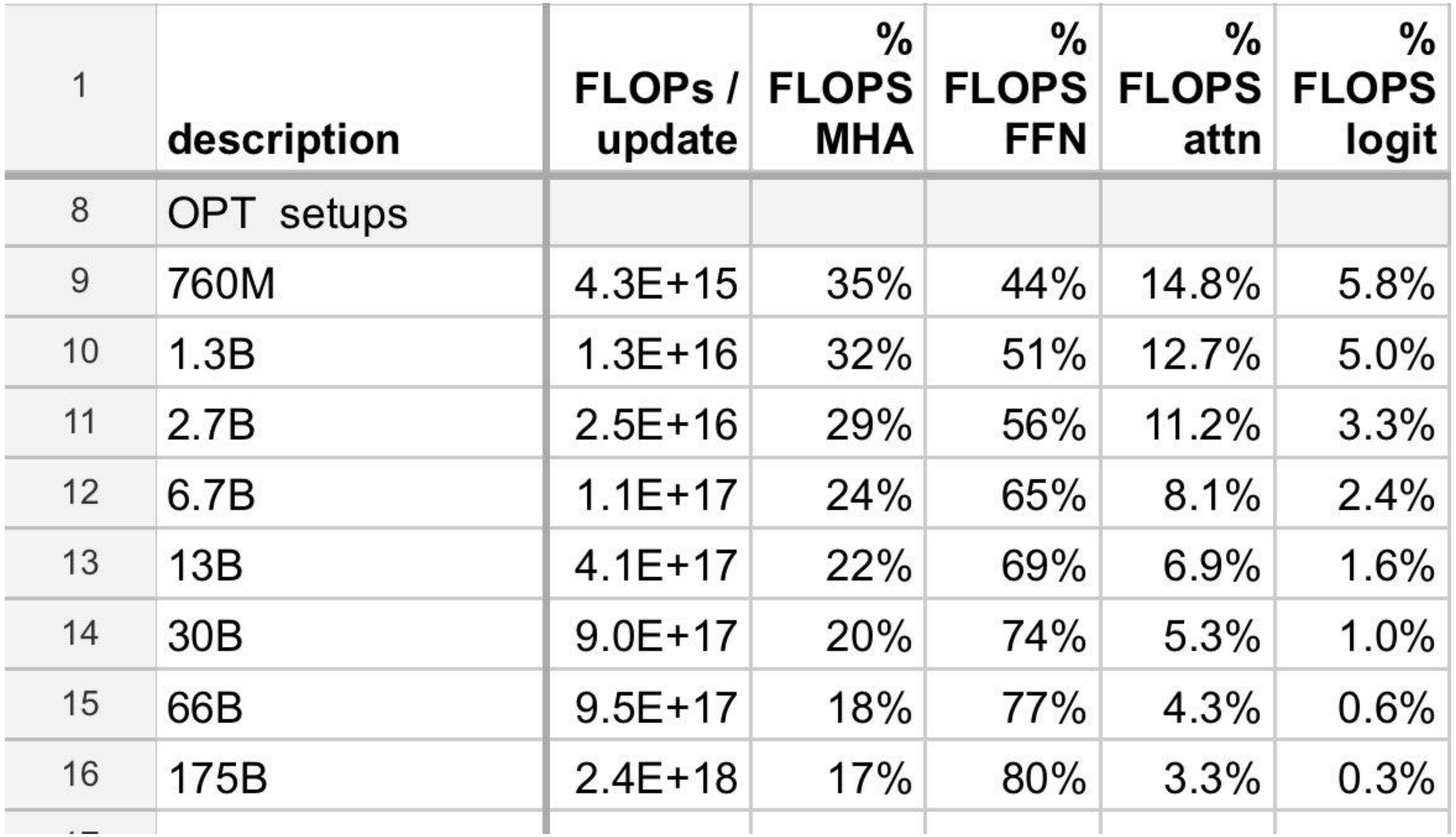
- For attentions, similarly split Q/K/V heads by columns
  - Output linear layer is split by rows



(b) Self-Attention.

Expert parallelism

- In very large LMs, the **FFNs** tend to take most parameters and computations



1	description	FLOPs / update	% FLOPS MHA	% FLOPS FFN	% FLOPS attn	% FLOPS logit
8	OPT setups					
9	760M	4.3E+15	35%	44%	14.8%	5.8%
10	1.3B	1.3E+16	32%	51%	12.7%	5.0%
11	2.7B	2.5E+16	29%	56%	11.2%	3.3%
12	6.7B	1.1E+17	24%	65%	8.1%	2.4%
13	13B	4.1E+17	22%	69%	6.9%	1.6%
14	30B	9.0E+17	20%	74%	5.3%	1.0%
15	66B	9.5E+17	18%	77%	4.3%	0.6%
16	175B	2.4E+18	17%	80%	3.3%	0.3%

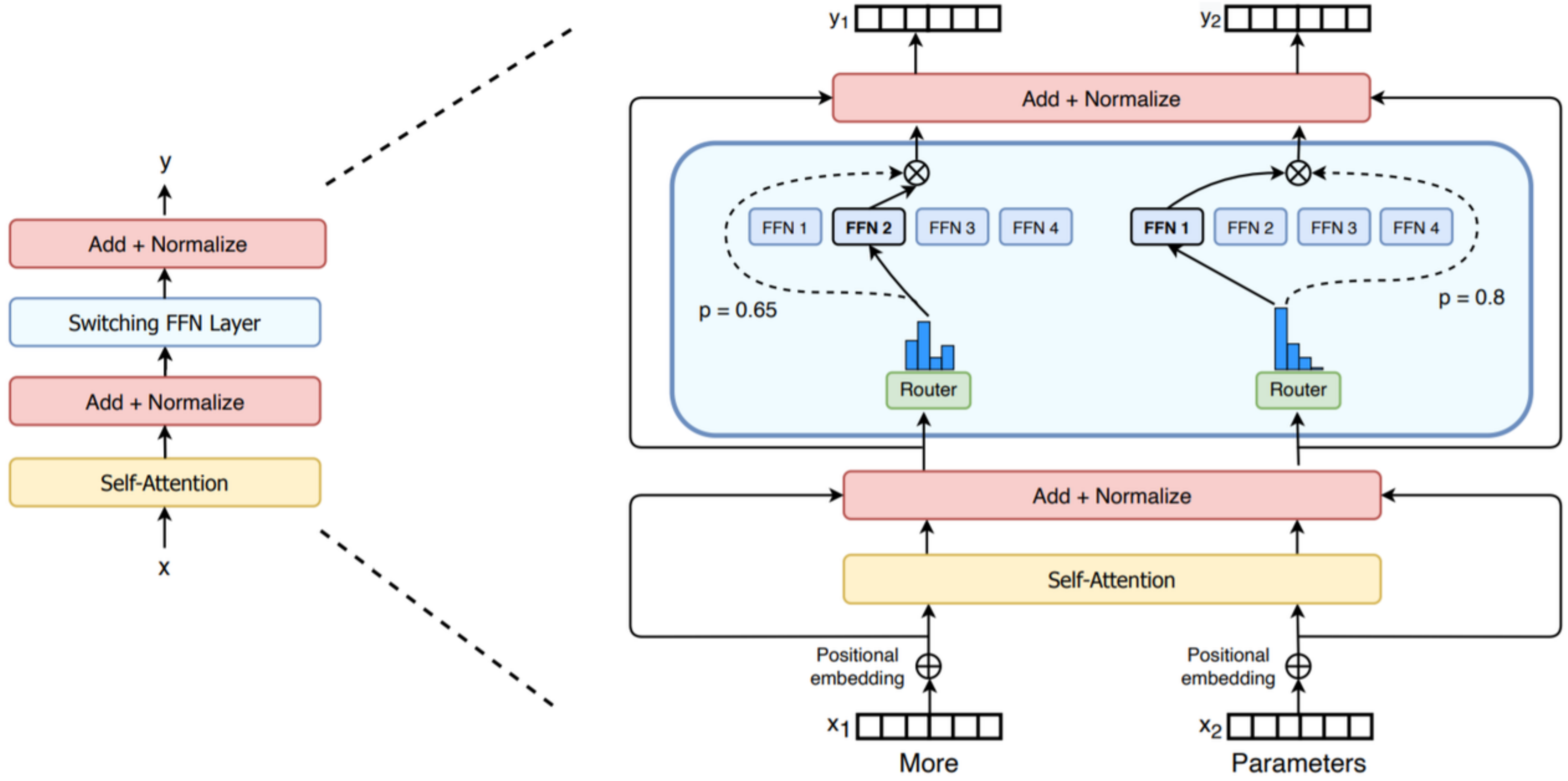
Expert parallelism

- **Idea.** Distribute FFNs only over the GPUs
    - Send a fraction of data in a batch to each GPU

- **Even better.** Specialize FFNs for different tokens (experts)
    - Do “routing” of tokens to each FFN

Mixture-of-Experts

- Existed from LSTM era, back in 2017
    - **Transformers.** GShards (2021), Switch Transformers (2022)

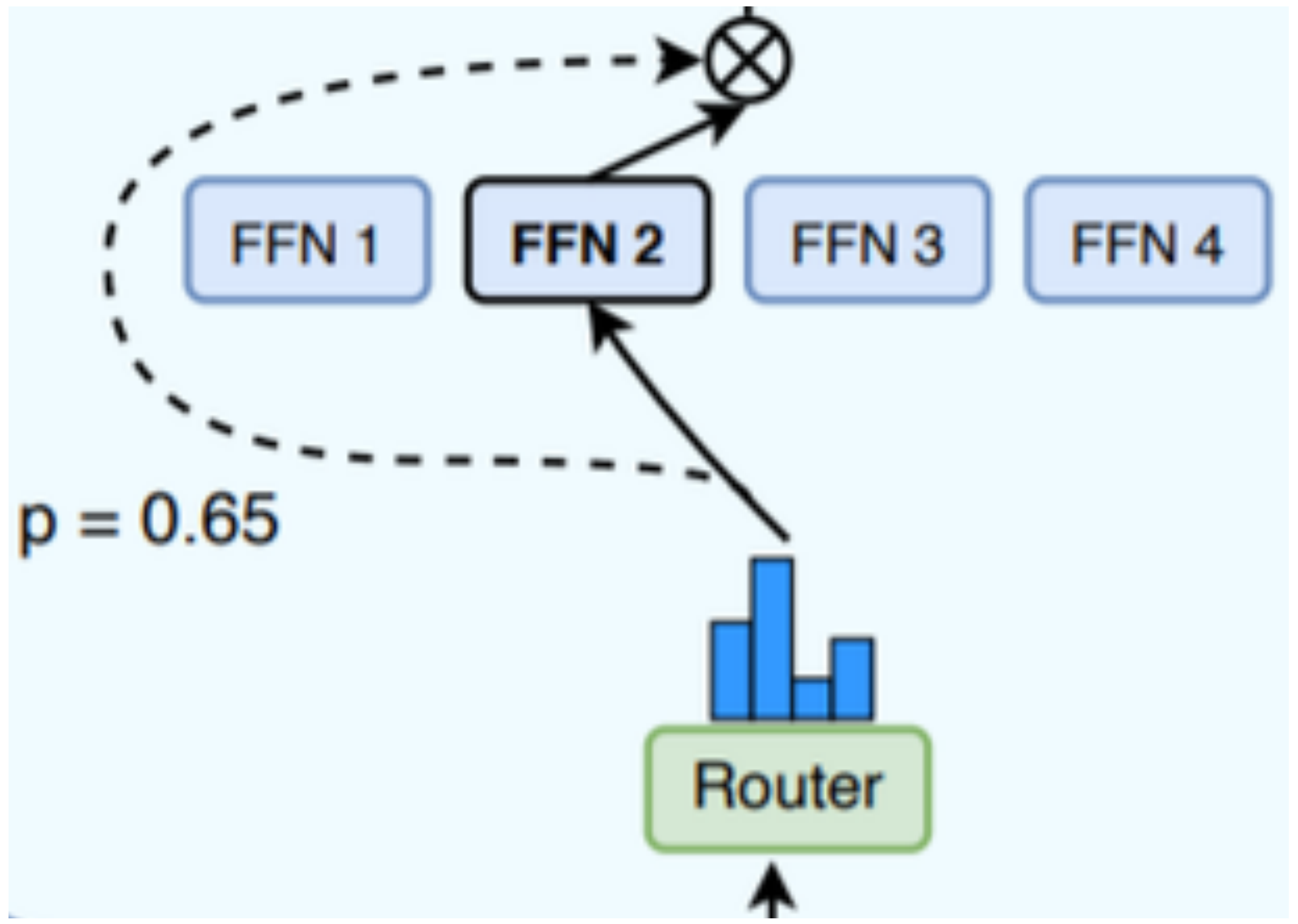


Mixture-of-Experts

- An output of an MoE module is

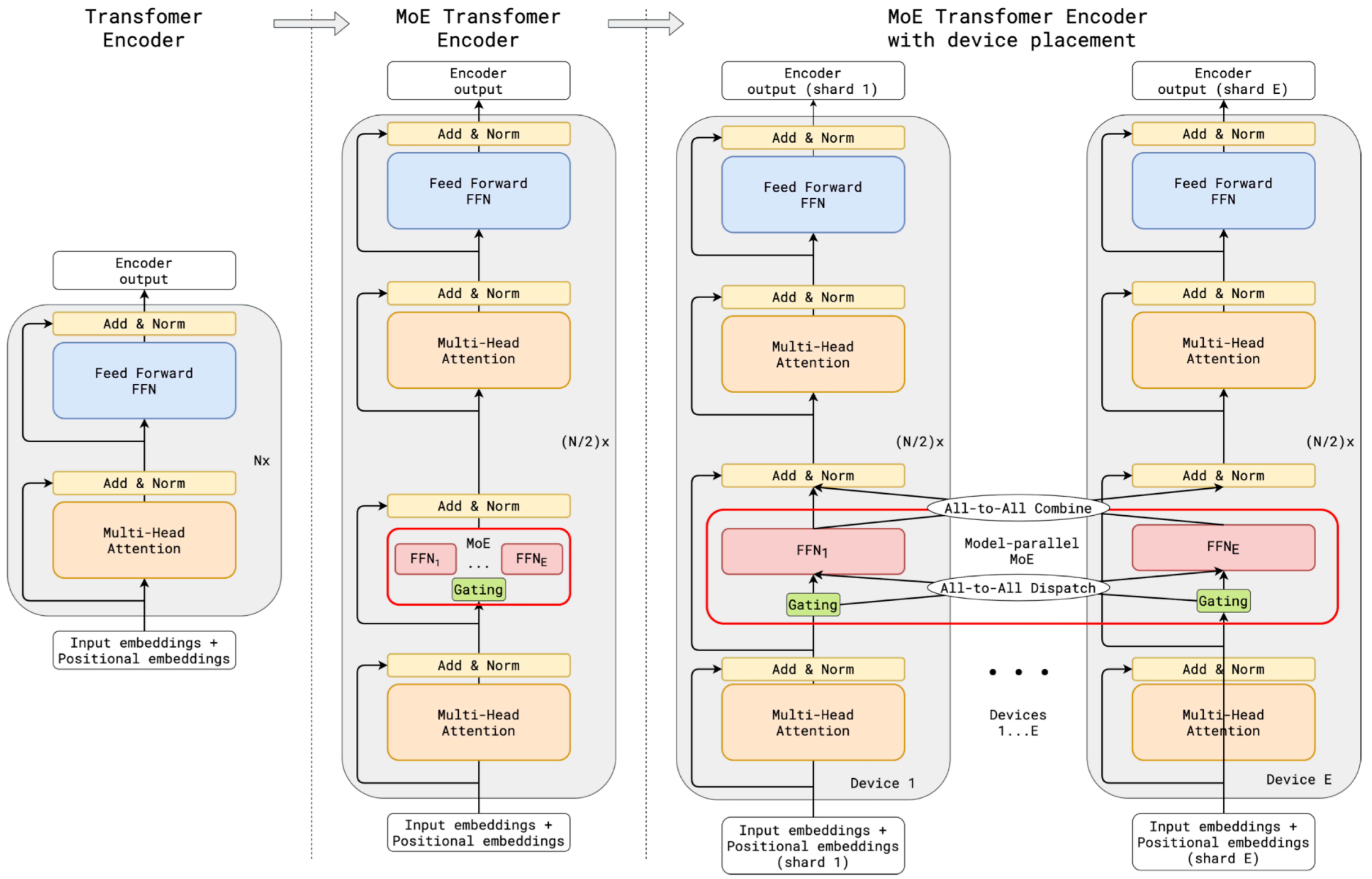
$$y = \sum_{i=1}^{n} G_i(x)E_i(x)$$

  - $E_i(\cdot)$: Output of expert i
  - $G_i(\cdot)$: Gating function



$$G(x) = \text{SoftMax}(\text{KeepTopK}(H(x)))$$ (or change the order of SM & TK)

- H can be a linear model $\quad H(x) = Wx + (\text{noise})$
  - Noise for load balancing



Transfomer Encoder
MoE Transfomer Encoder
MoE Transfomer Encoder with device placement
Encoder output
Add & Norm
Feed Forward FFN
Multi-Head Attention
Nx
Input embeddings + Positional embeddings
(N/2)x
MoE
FFN1
FFNE
Gating
Encoder output (shard 1)
Encoder output (shard E)
All-to-All Combine
Model-parallel MoE
All-to-All Dispatch
Devices 1...E
Device 1
Device E
Input embeddings + Positional embeddings (shard 1)
Input embeddings + Positional embeddings (shard E)

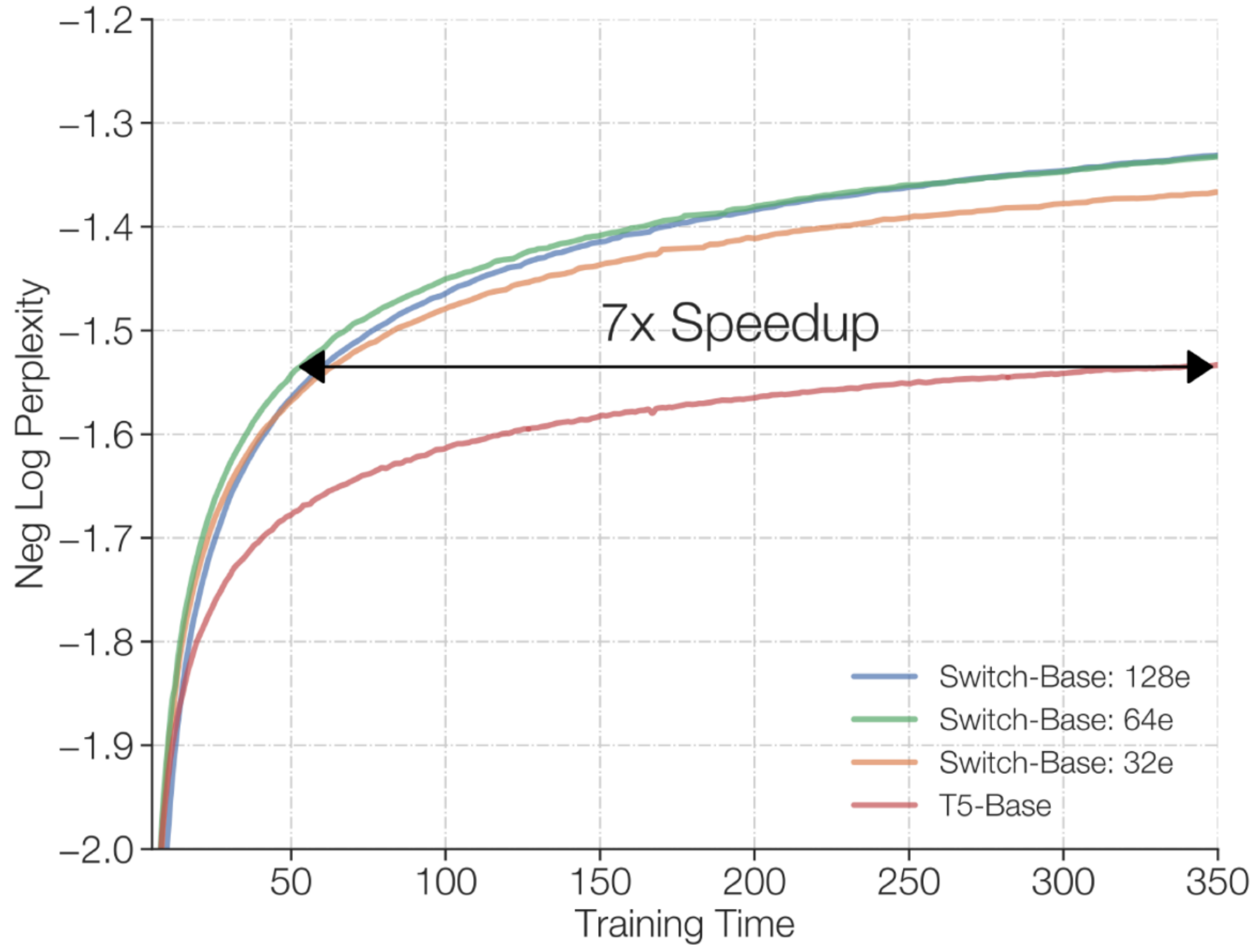
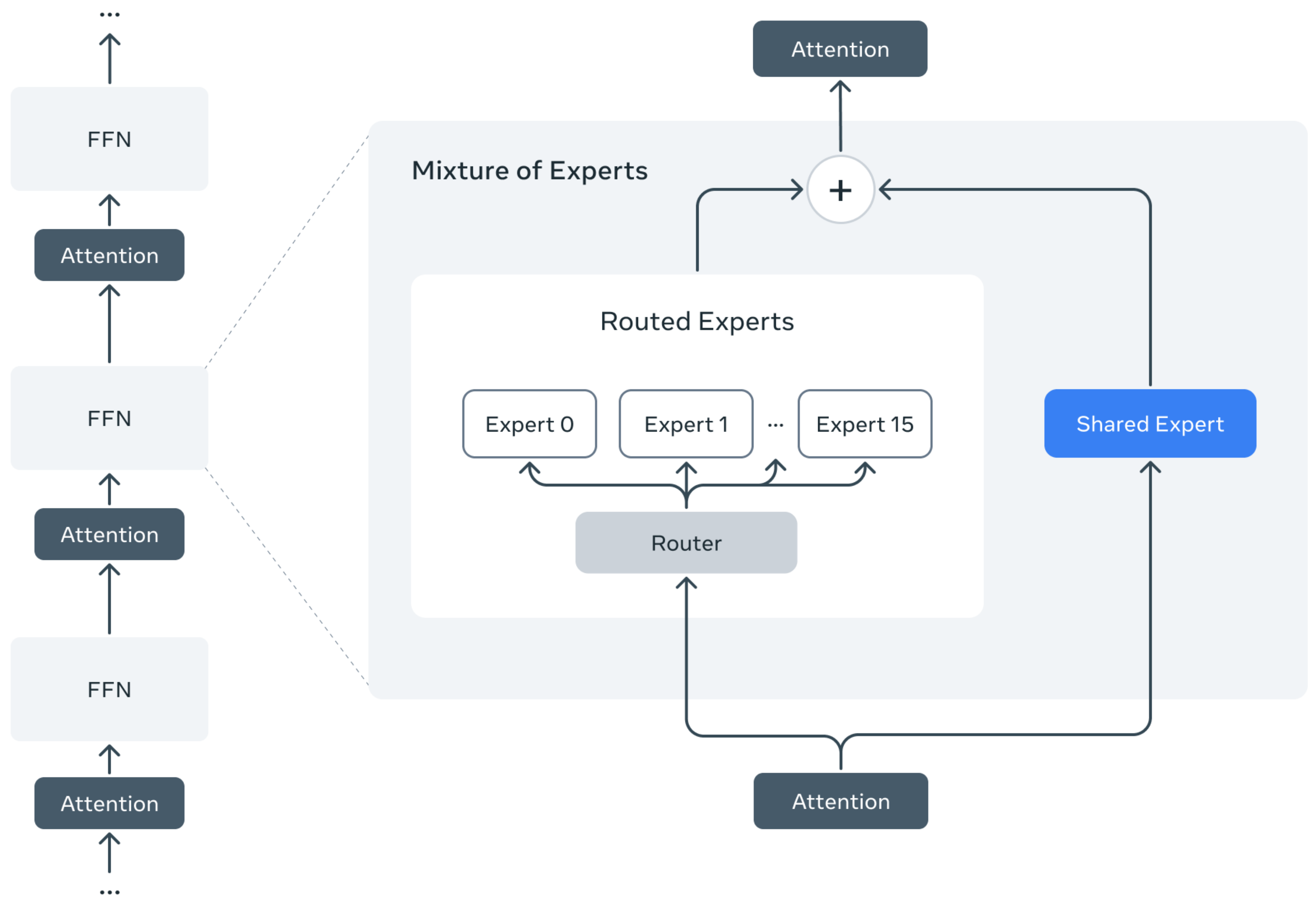


Figure 5: Speed advantage of Switch Transformer. All models trained on 32 TPUv3 cores with equal FLOPs per example. For a fixed amount of computation and training time, Switch Transformers significantly outperform the dense Transformer baseline. Our 64 expert Switch-Base model achieves the same quality in *one-seventh* the time of the T5-Base and continues to improve.

Advantages

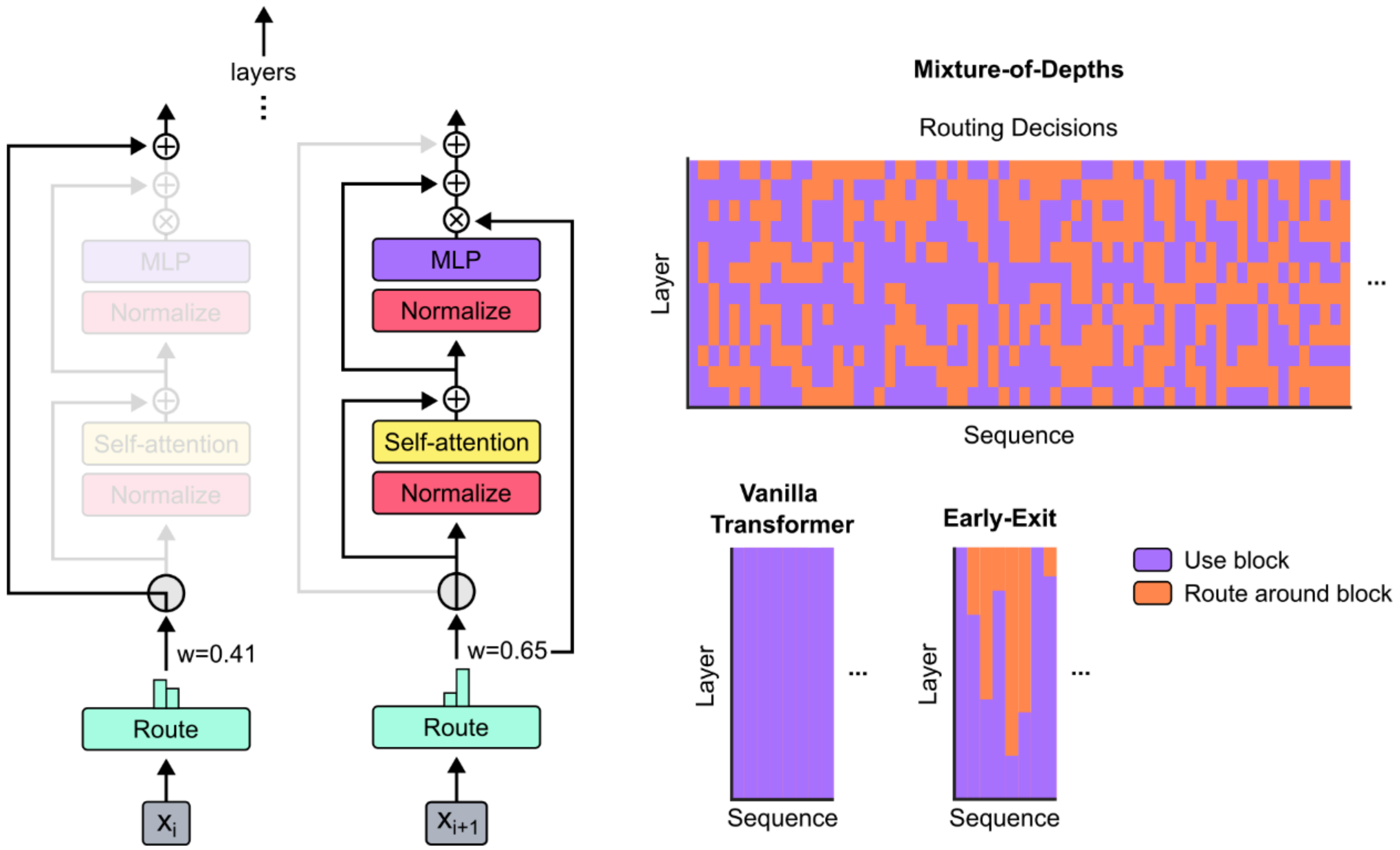
- **Training.** Can train overparameterized models with low cost
- **Inference.** Small number of active parameters

- Example: LLaMA-4.
    - Uses 14B active parameters
        - 128 routed experts
        - 1 shared expert



Further Readings

- Mixture-of-Depths
  - https://arxiv.org/abs/2404.02258



That’s it for today 🙌