DSCI 565: OPTIMIZATION ALGORITHMS

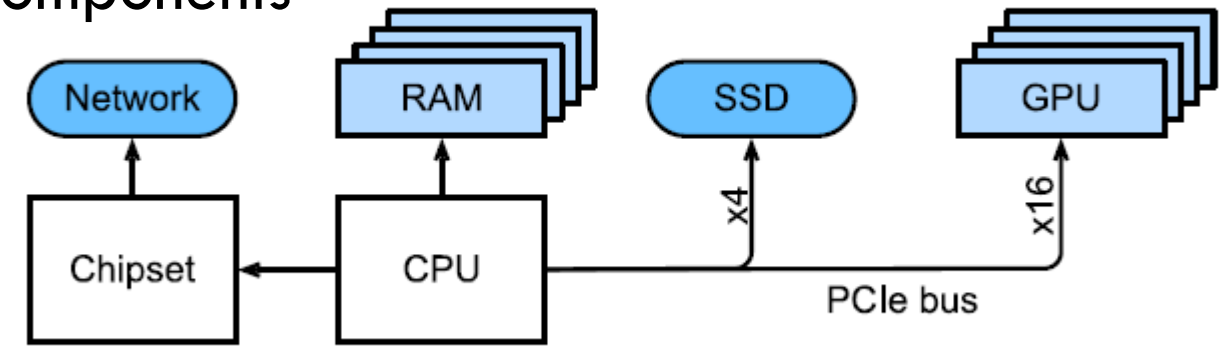
This content is protected and may not be shared, uploaded, or distributed.

Ke-Thia Yao

Lecture 18: 2024 November 4

Hardware

- The hardware components and the connectivity between them can have a large effect on the computational performance
- Section 13.4 of Zhang *et al*. has lots of interesting details
- Hardware include:
  - Components: Processor (CPU, GPU), memory (RAM), storage (SSD, hard drive), network
  - Connectivity between components



Connectivity

- Connectivity is typically measured by
  - Bandwidth, the amount of data transferred per second
  - Latency, the time delay before the transfer starts

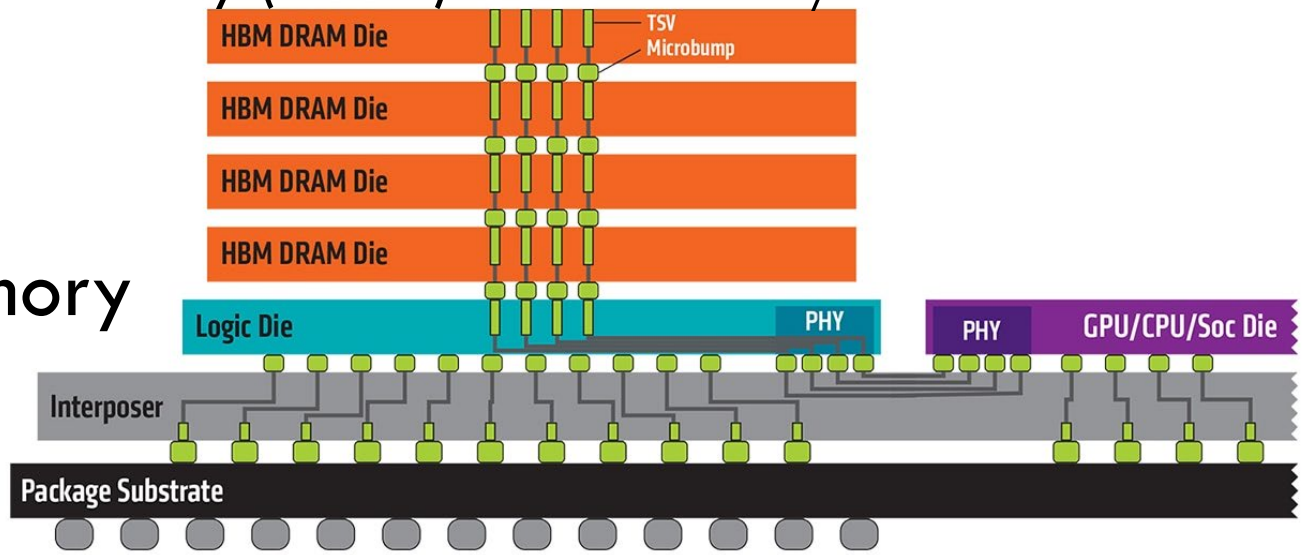
CPU-Memory Connectivity

- A typical DDR4 RAM offers 20-25 GB/s bandwidth per module, where each module has a 64-bit-wide bus
- A CPU have between 2 and 4 memory channels, i.e., peak memory bandwidth between 40GB/s to 100GB/s
- Read a 64-bit record takes just 0.2ns at 40GB/s
- But setting up the transfer (send address to RAM then wait for start of transfer) is 100ns (500 times!)

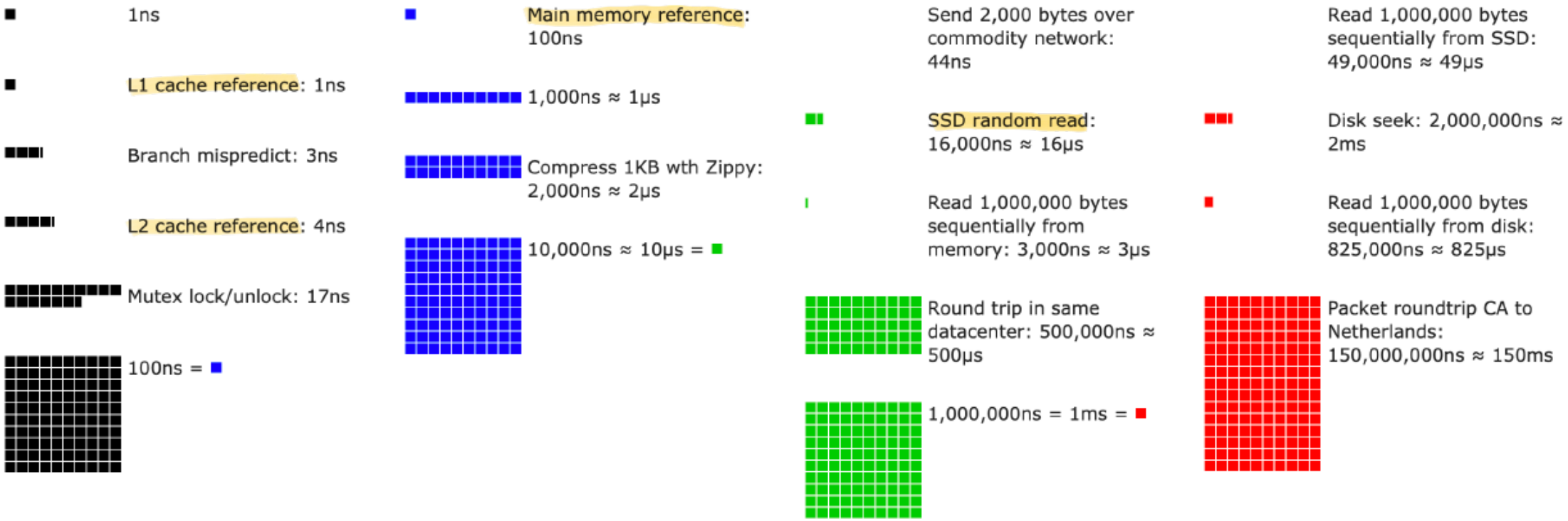
GPU-Memory Connectivity

- GPUs require much higher bandwidth
- Use wider bus, e.g., NVIDIA RTX 2080 has 352-bit-wide bus
- Use higher speed memory
  - NVIDIA RTX 2080 using GDDR6 has 500 GB/s bandwidth
  - NVIDIA V100 using High Bandwidth Memory (HBM2) has 900 GB/s
  - NVIDIA H100 using High Bandwidth Memory (HBM3) has 3300 GB/s

High Bandwidth Memory



Latency



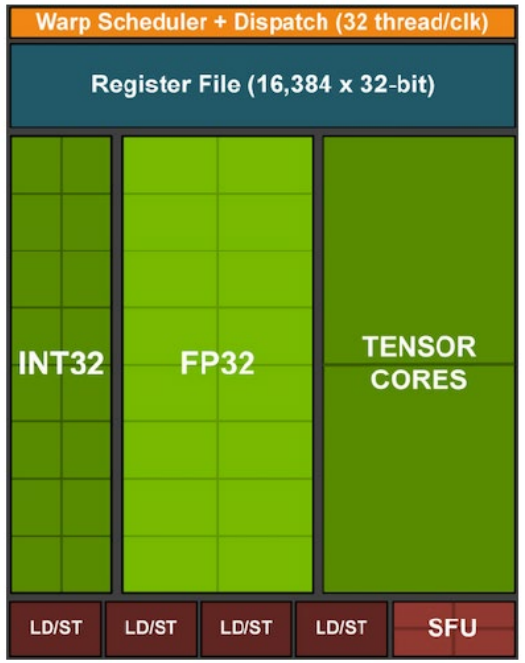
https://colin-scott.github.io/personal_website/research/interactive_latency.html

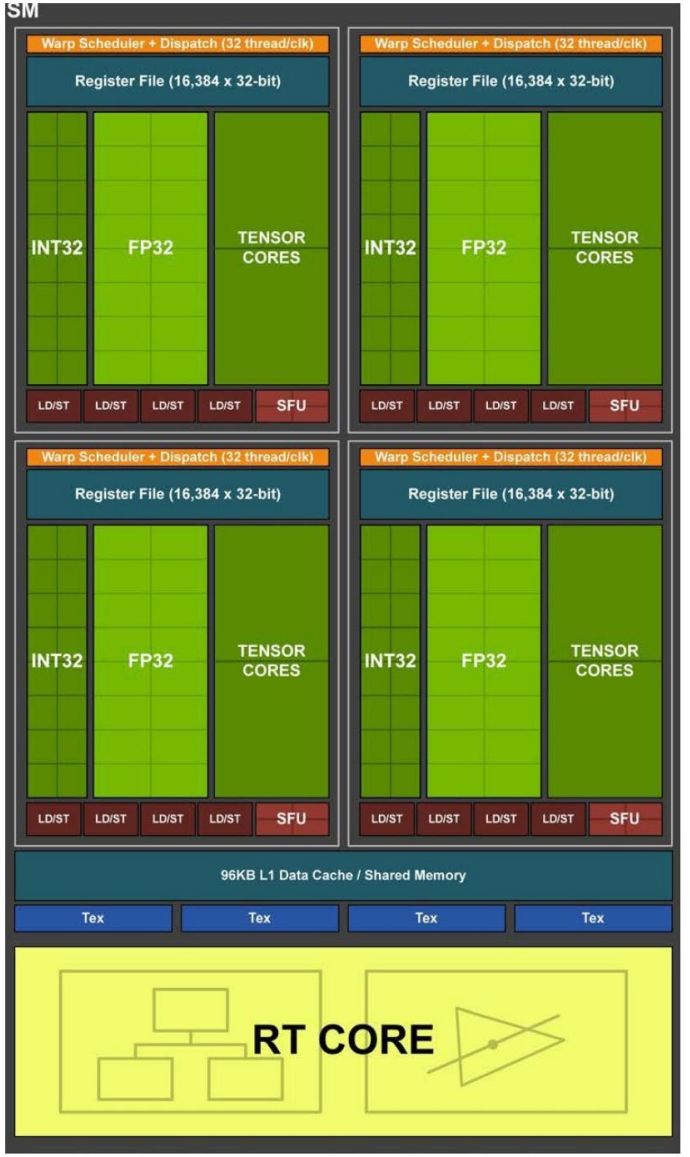
GPU

- Deep learning would not have been successful without the GPUs
- GPU design strategy
  - Many more core
  - Support matrix operations, not just vector operations (tensor cores)

Nvidia Turing Architecture

- Basic Processing Block
  - 16 INT32 and 16 FP32 units
  - 2 tensor cores
- Streaming multiprocessor (SM)
  - 4 processing blocks
  - L1 Cache
  - 1 Ray Tracing (RT) core



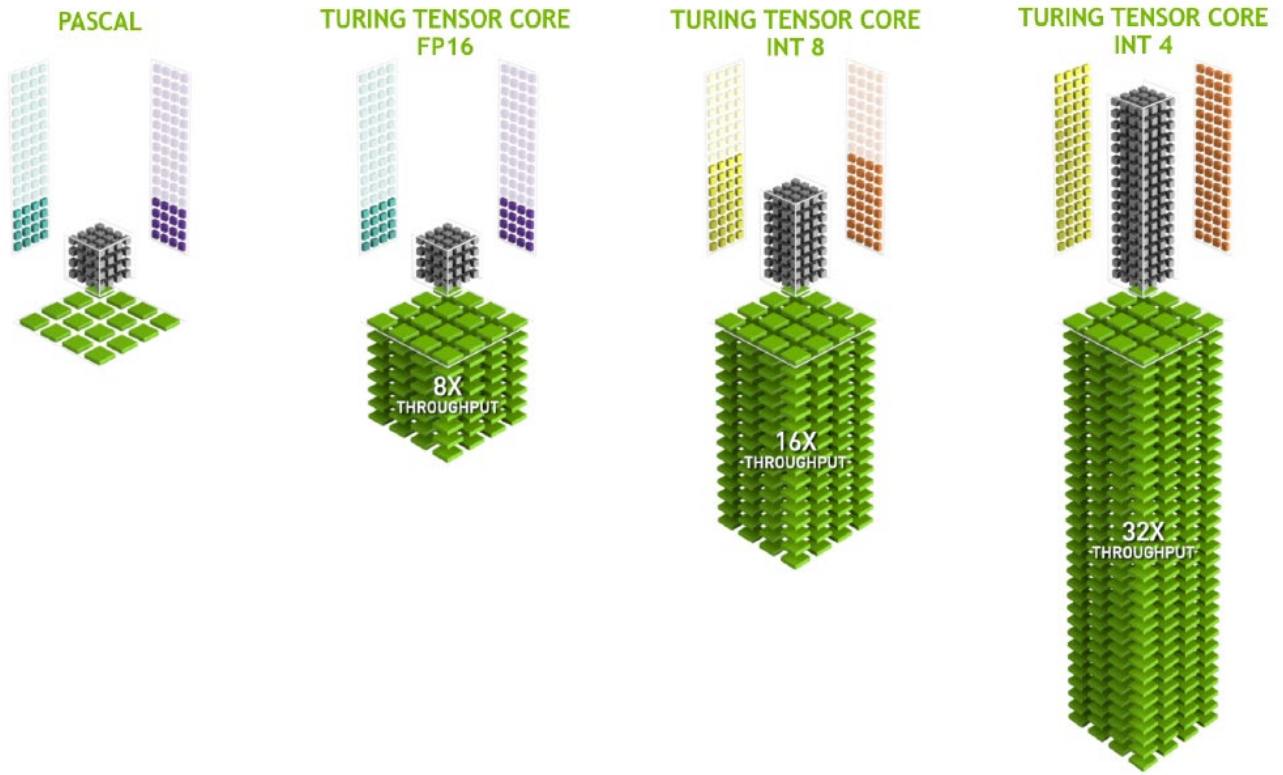


Nvidia Turing Architecture

- Graphics processing cluster
  - 12 streaming multiprocessors
- GPU
  - 6 processing clusters (72 SMs)
  - L2 cache
  - NVLink

Tensor Core

- Performs matrix multiplication and accumulate
- Mixed precession calculation: multiple in FP16 and accumulate in FP32
- Optimized for 4×4 and 16×16 matrix operations



NVIDIA H100

- Basic Processing Block
  - 16 INT32 and 32 FP32 units
  - 16 FP64 units
  - 1 4th generation tensor core
- Streaming multiprocessor
  - 4 processing blocks
  - L1 Cache

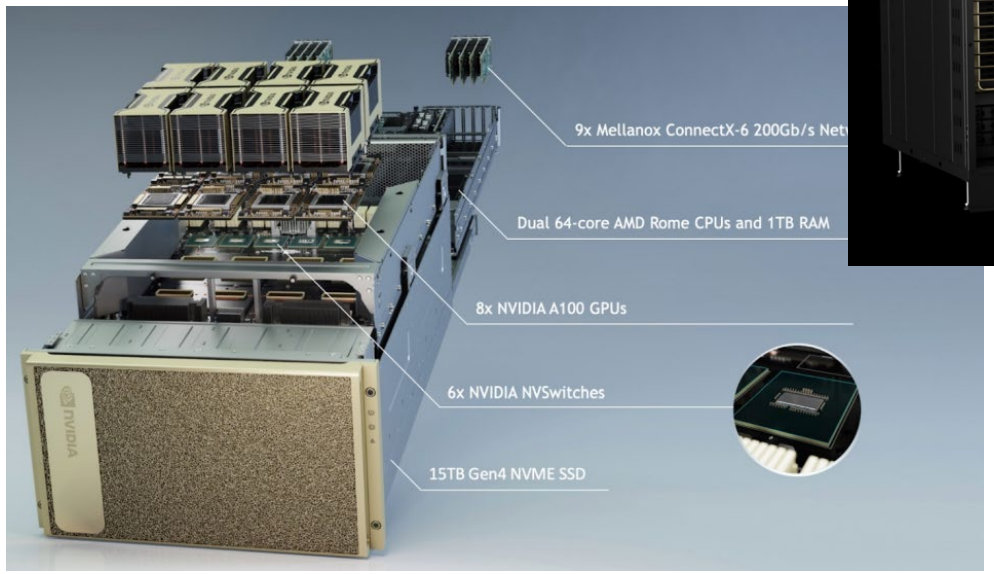
NVIDIA H100

- GH100 with 144 SMs

GPU Data Centers

4 DGX per rack

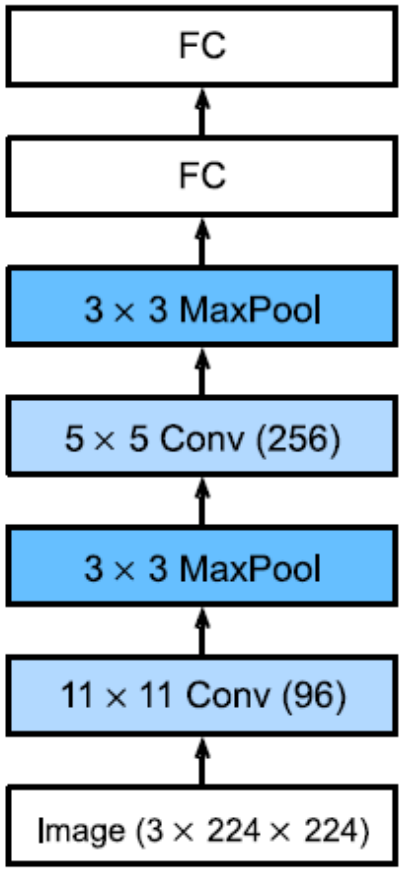
Nvidia DGX (8 GPUs)



Rows of racks in a data center

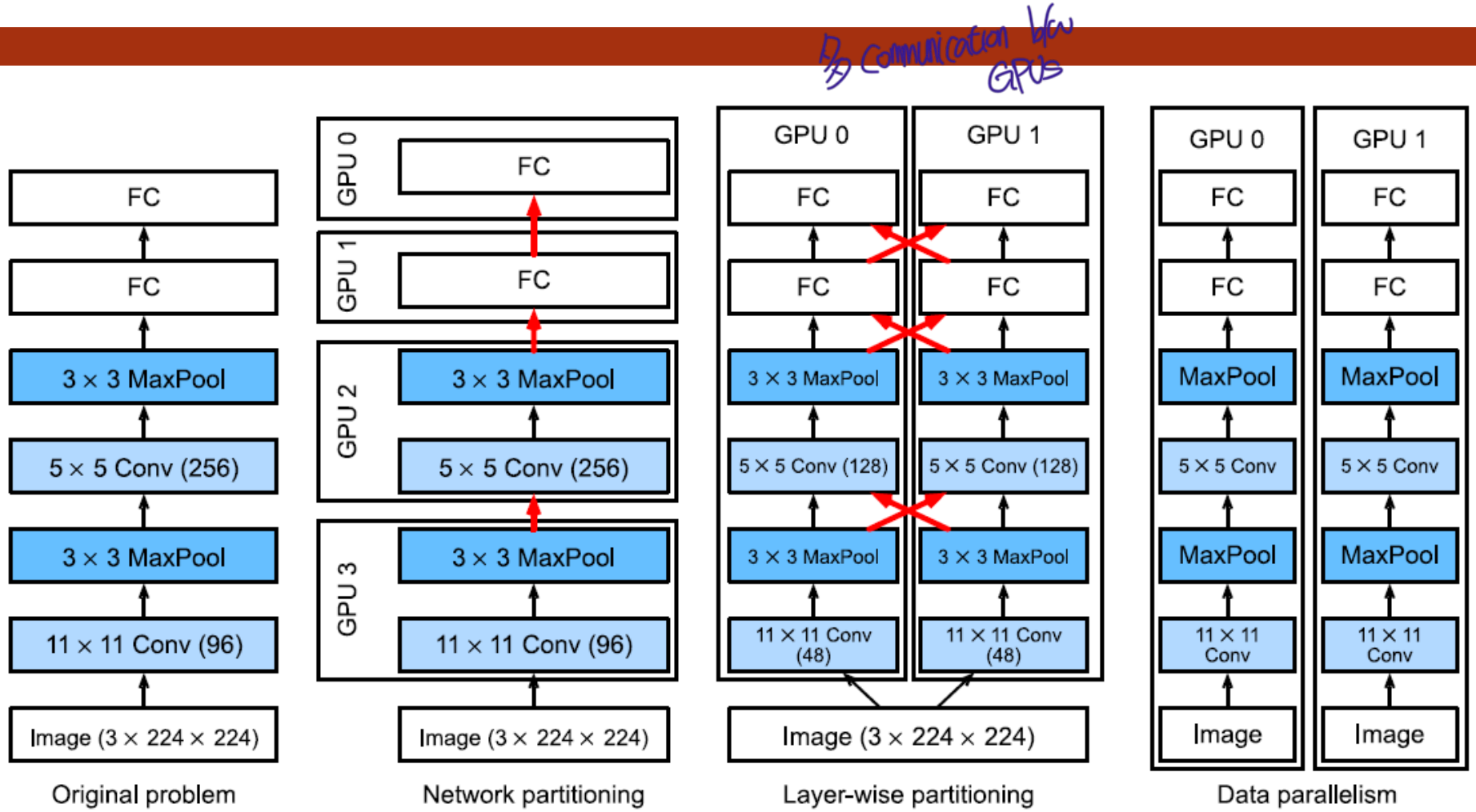
Training on Multiple GPUs

- How to split up the problem to train on multiple GPUs
- Pipeline parallelism / network partitioning
  - Partition the network by layers, i.e., place one or more layers in a GPU
- Tensor parallelism / layer-wise partitioning
  - Tensor in each layer is partition among multiple GPUs
- Data parallelism
  - Partition the minibatch. Each GPU holds a complete copy of the network



Original problem

Parallelization on multiple GPUs

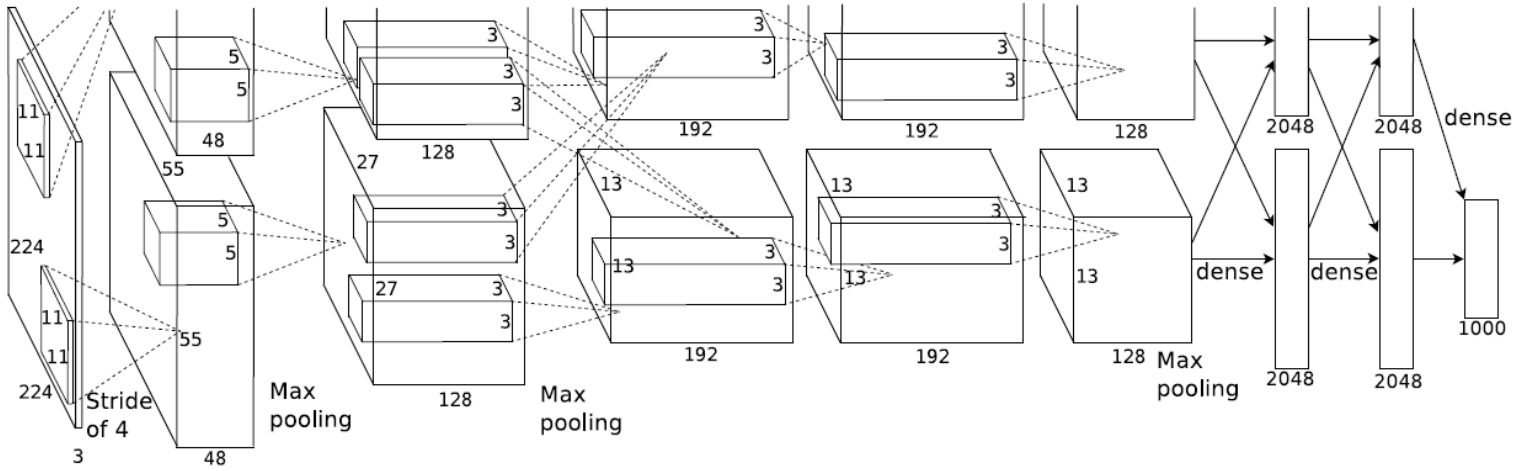


Pipeline Parallelism / Network Partitioning

- Advantages
  - Memory footprint per GPU is smaller (only a few layers of the network)
- Disadvantages
  - Balance workload across GPUs can be tricky. Some layers are more computationally intensive than other layers
  - Require large amount of data transfer between GPUs
  - Difficult to achieve linear scaling (compute time decreases linearly with number of GPUs)

Tensor Parallelism / Layer-wise Partitioning

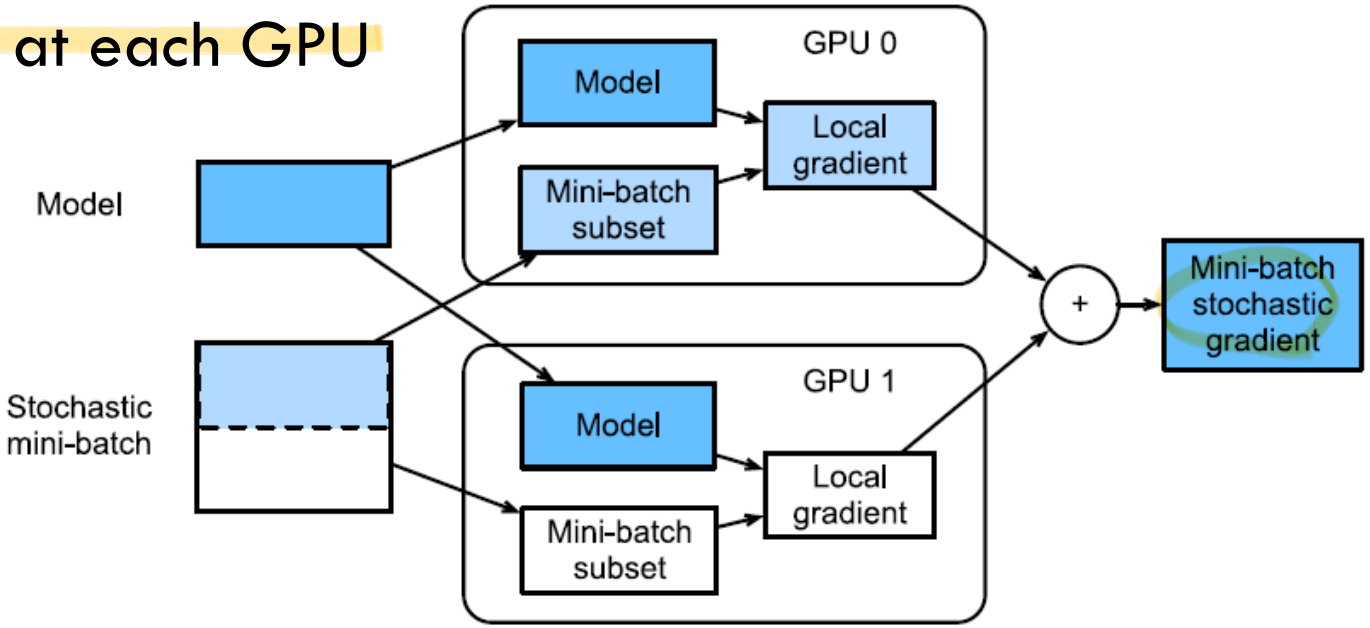
- Advantages
  - Memory footprint per GPU is smaller, e.g., with 4 GPUs and a layer with 64 channels each GPU gets 16 channels
- Disadvantages
  - Very large number of synchronization/barrier operations
  - Require large amount of data transfer between GPUs



AlexNet

Data Parallelism

- Advantages
  - Easy to implement
  - Near linear scaling
- Disadvantages
  - Need to store entire model at each GPU



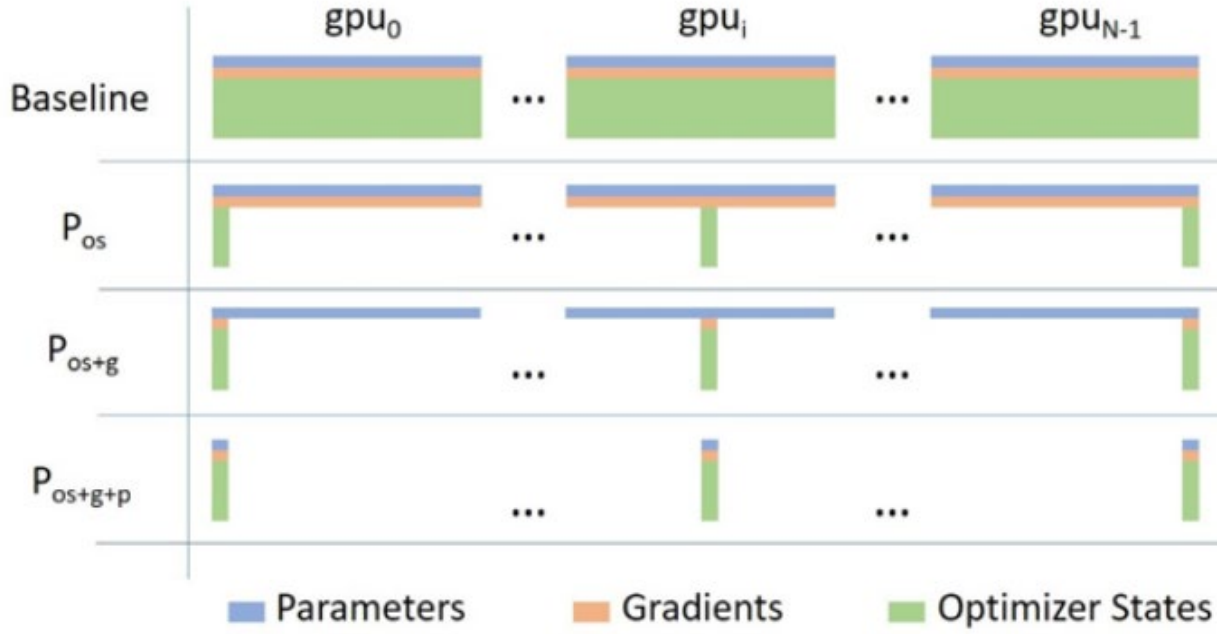
Data Parallelism Notebook

- chapter_computational-performance/multiple-gpus.ipynb
  - Toy network
  - Data synchronization: get_param(), allreduce()
  - Distributing data
  - Training
- chapter_computational-performance/multiple-gpus-concise.ipynb
  - `net = nn.DataParallel(net, device_ids=devices)`

Zero Redundancy Optimization (ZeRO)

data parallelism + pipeline parallelism

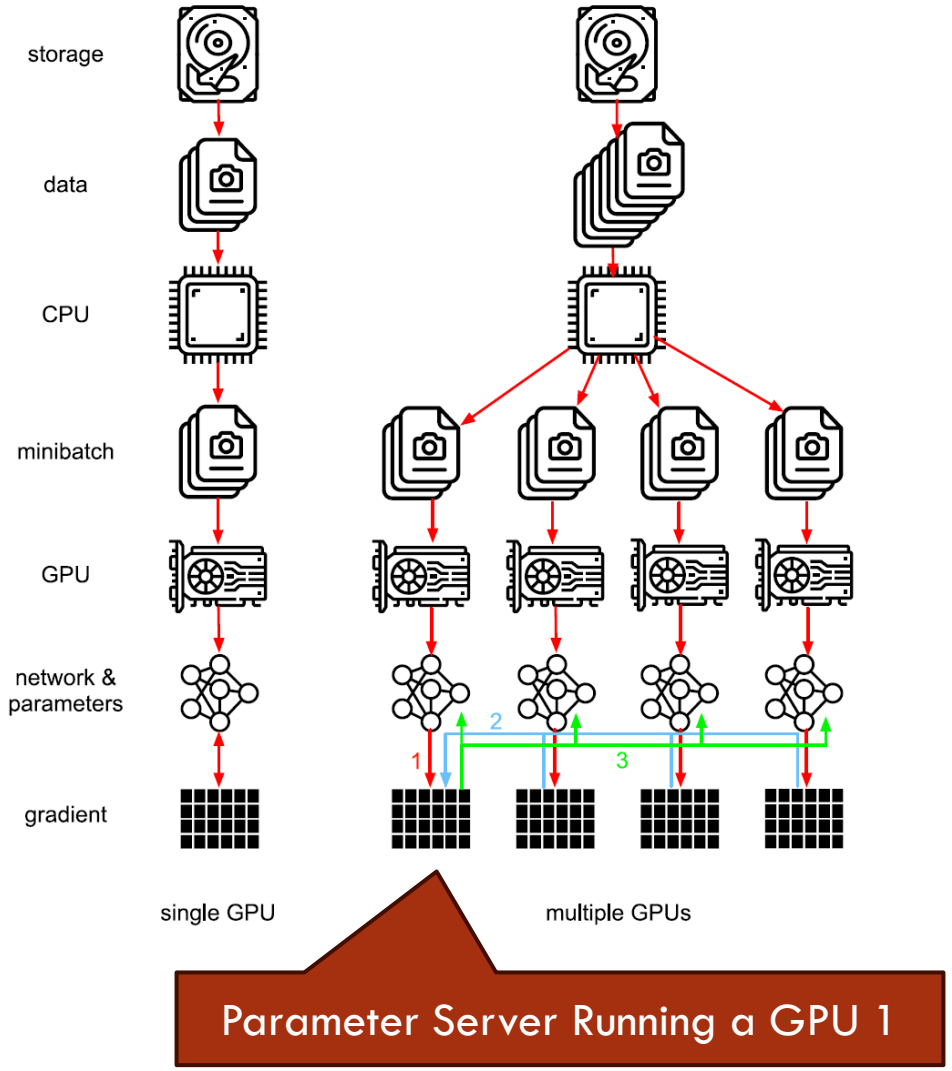
- Uses data parallelism and pipeline parallelism
- Avoid memory overhead by broadcasting data as needed
- Can train a trillion-parameter model
- ZeRO-2 can train BERT in 44 minutes using 1024 NVIDIA V100
- See animation



https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

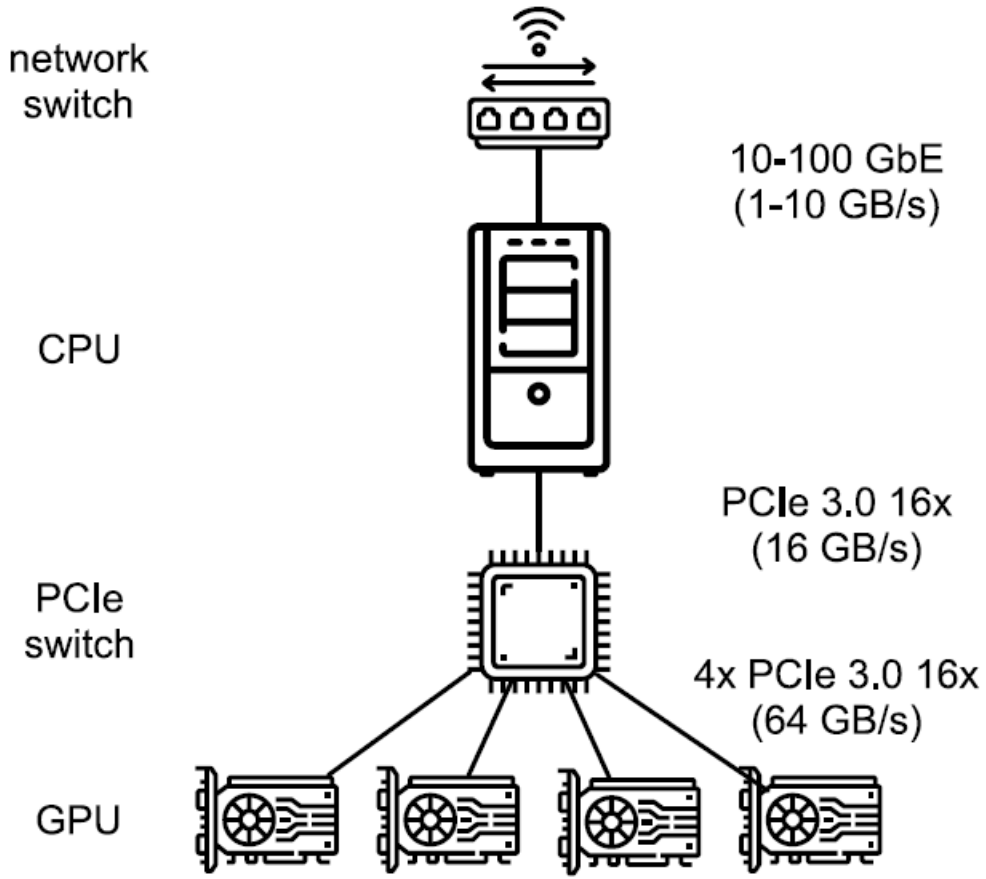
Parameter Server

- A parameter server stores, collects and distributes parameters/values needed by the deep learning model
- For example, for the data parallel approach a parameter server would
  - Aggregate gradients from all GPUs
  - Update the weight parameters with the gradients
  - Broadcast the updated parameters to all GPUs



Location of the Parameter Server

- Bandwidth is an import consideration in determining which device should host the parameter server
  - 1-10 GB/s for off-machine server connected through network switch
  - 16 GB/s for on-machine CPU server connected through PCIe bus
  - 64 GB/s (4 * 16 GB/s) for on-machine GPU server connected through 4x PCIe switch

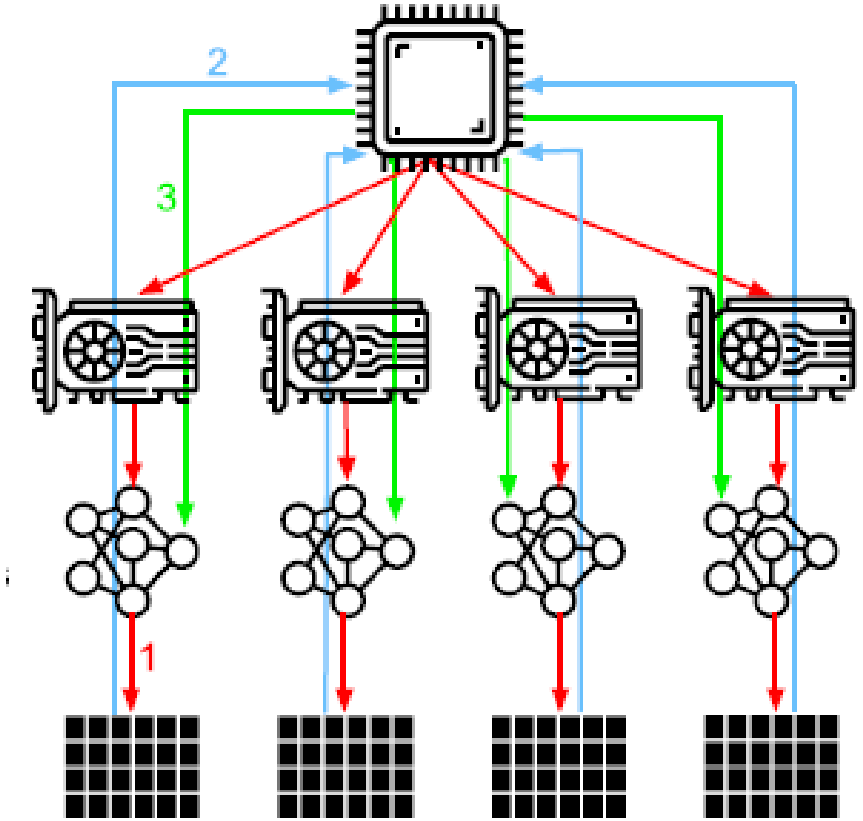


Parameter Server Example

Suppose the size of the gradients and the parameters are both 160 MB

CPU Server

- 40 ms $(4 \times 160\text{MB}/16\text{GB/s})$ to send gradients to CPU Server
- 40 ms to send updates parameters to all 4 GPUs
- 80 ms total



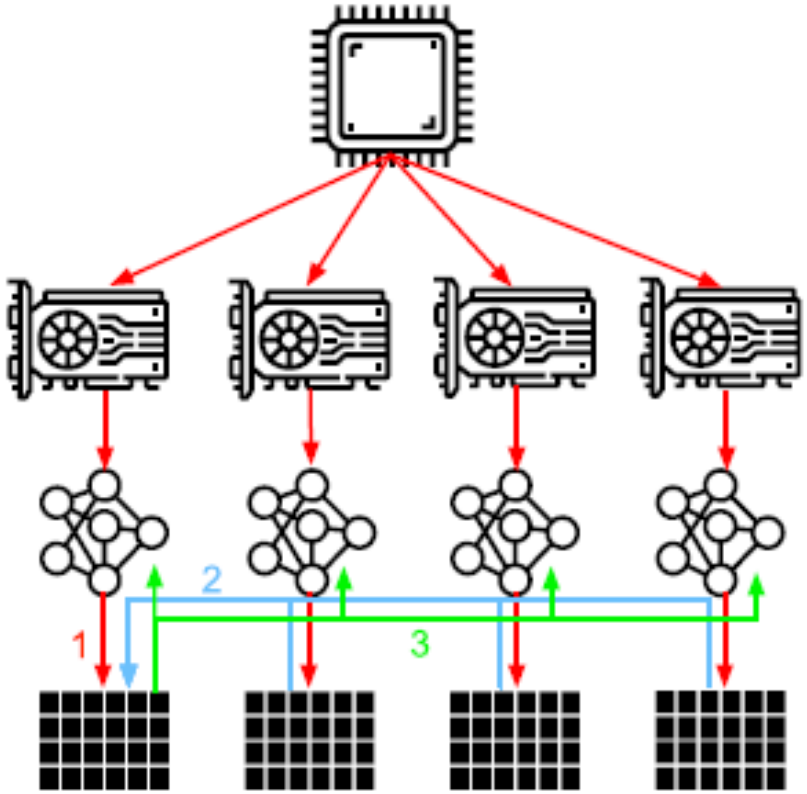
Parameter server in CPU

Parameter Server Example

Suppose the size of the gradients and the parameters are both 160 MB

GPU Server

- 30 ms $(3 \times 160\text{MB}/\ 16\text{GB/s} = 3 \times 10\text{ms})$ to send gradients to GPU Server
- 30 ms to send updated parameters to other 3 GPUs
- 60 ms total



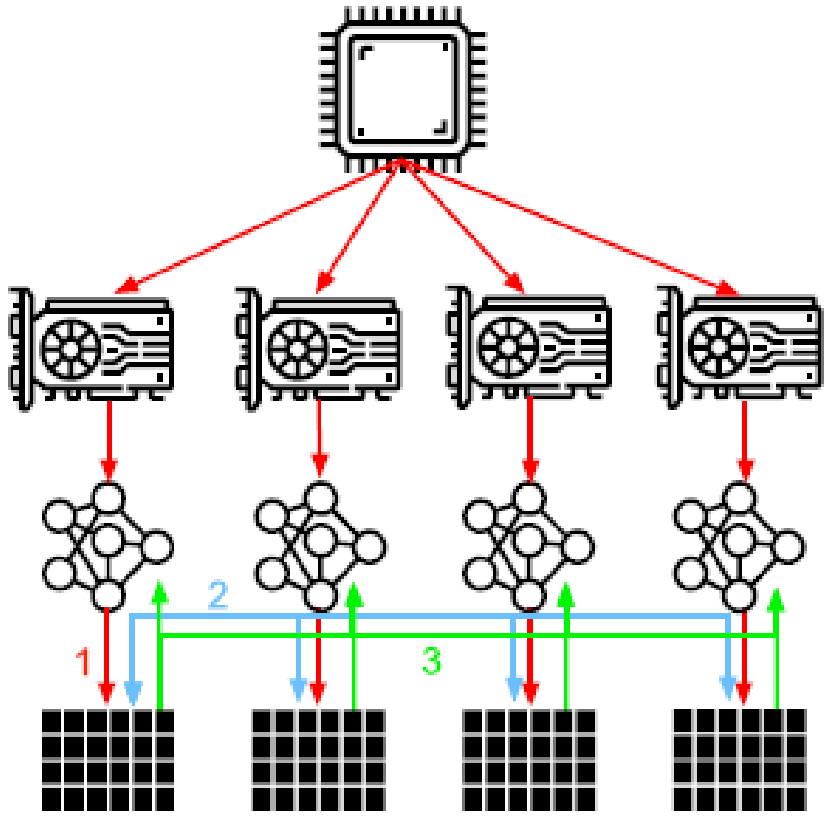
Parameter server in GPU 0

Parameter Server Example

ZeRO used

Distributed GPU

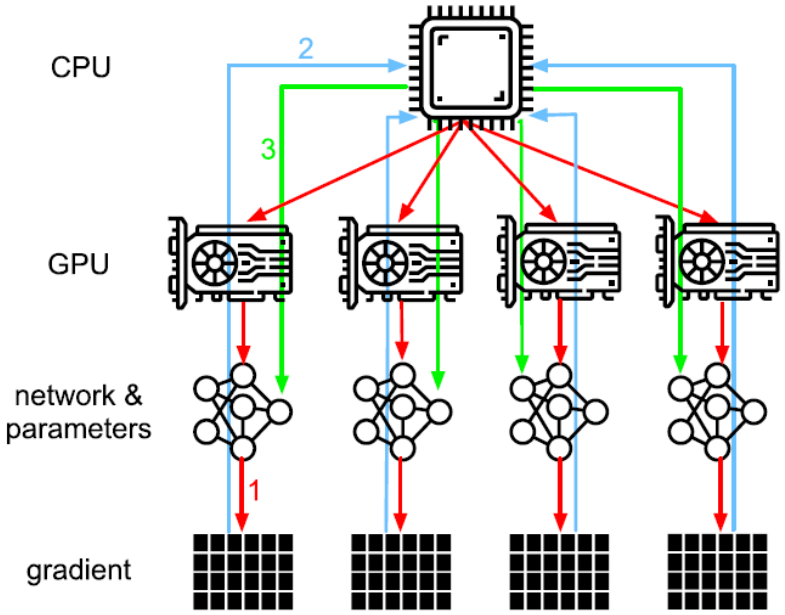
- Suppose each of four GPUs holds ¼ (40MB) of the data
- 7.5 ms $(3 \times 40\text{MB} / 16\ \text{GB/s} = 3 \times 2.5\text{ms})$ send gradients
  - Each GPU sends its portion of the gradient to other GPUs
  - This can be done simultaneously, since GPUs are connected by a switch
- 7.5 ms to send updated parameters
- 15 ms total



Parameter server distributed over all GPUs

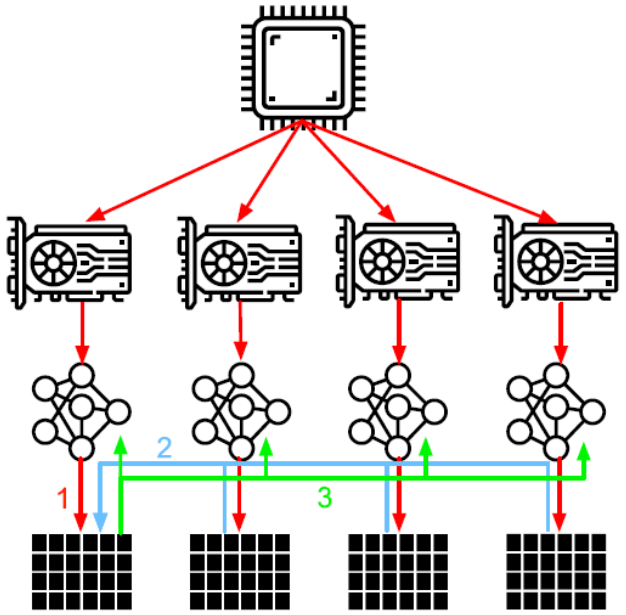
Parameter Server Example

CPU: 80ms



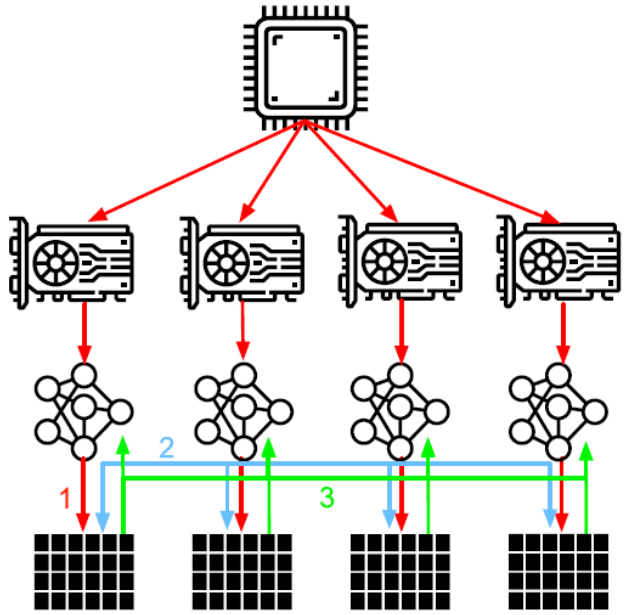
Parameter server in CPU

GPU: 60ms



Parameter server in GPU 0

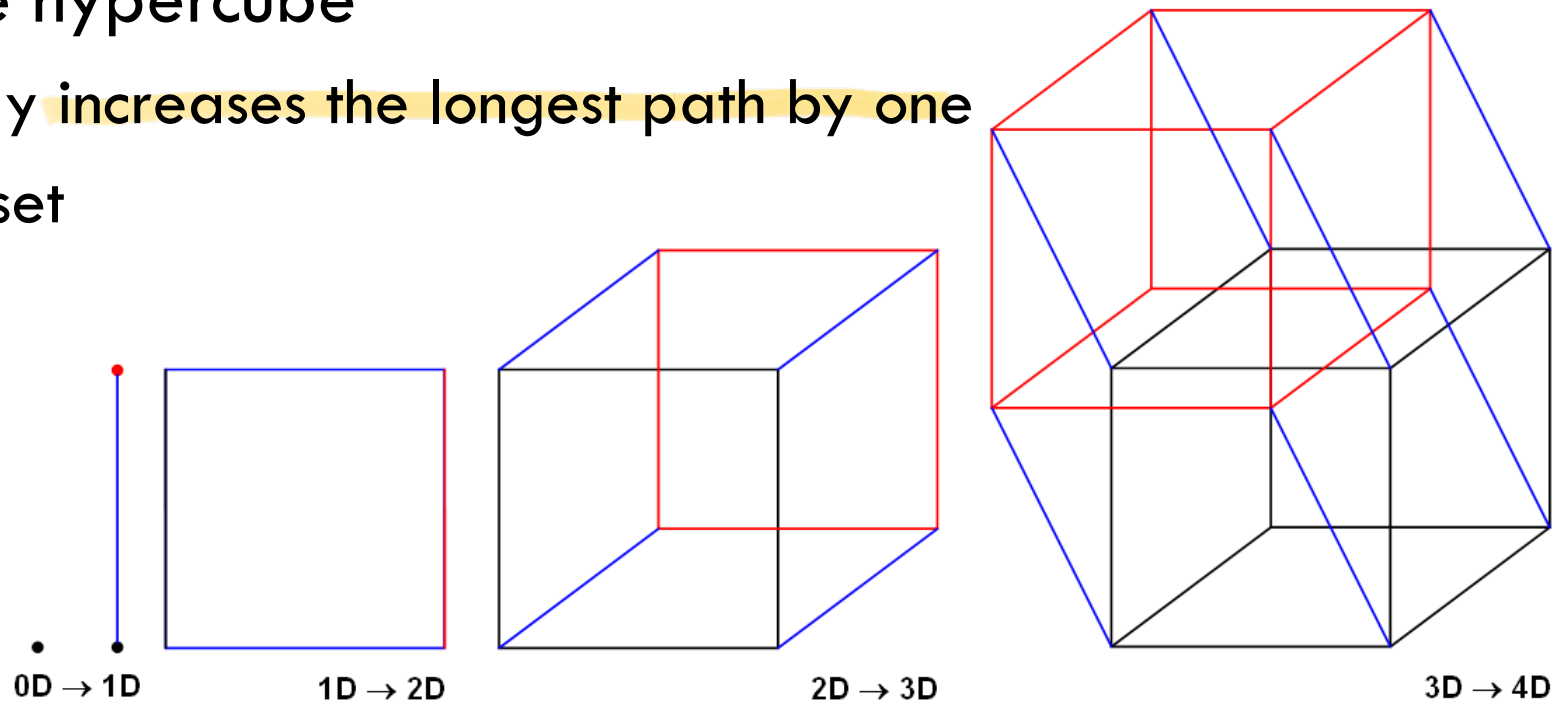
Distributed GPU: 15ms



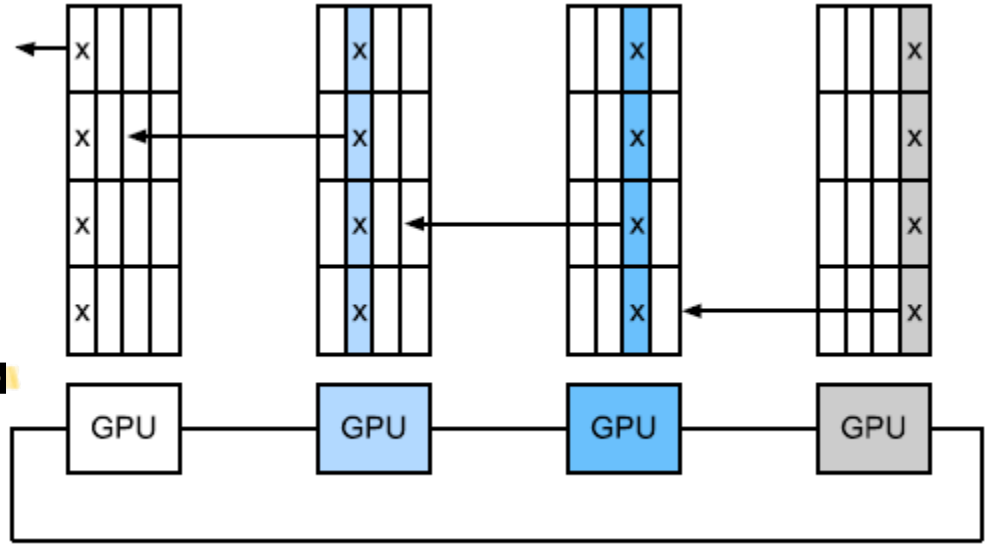
Parameter server distributed over all GPUs

Hypercube Topology

- Switches are more difficult to build
  - Any pair of nodes can connect directly
- Hypercube topology only have connections along the edges of the hypercube
  - Doubling the nodes only increases the longest path by one
  - Ring topology is a subset

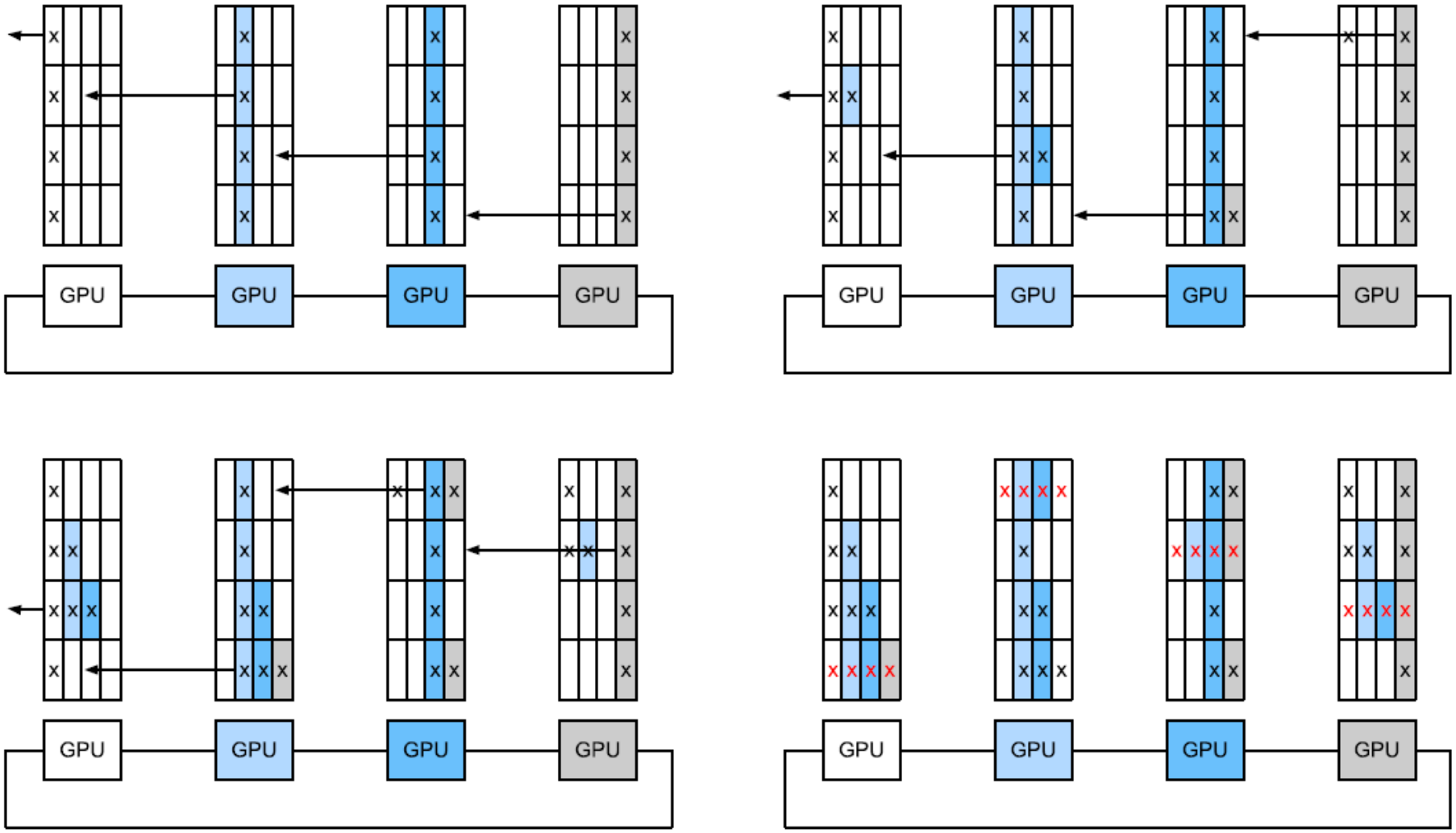


Ring Synchronization



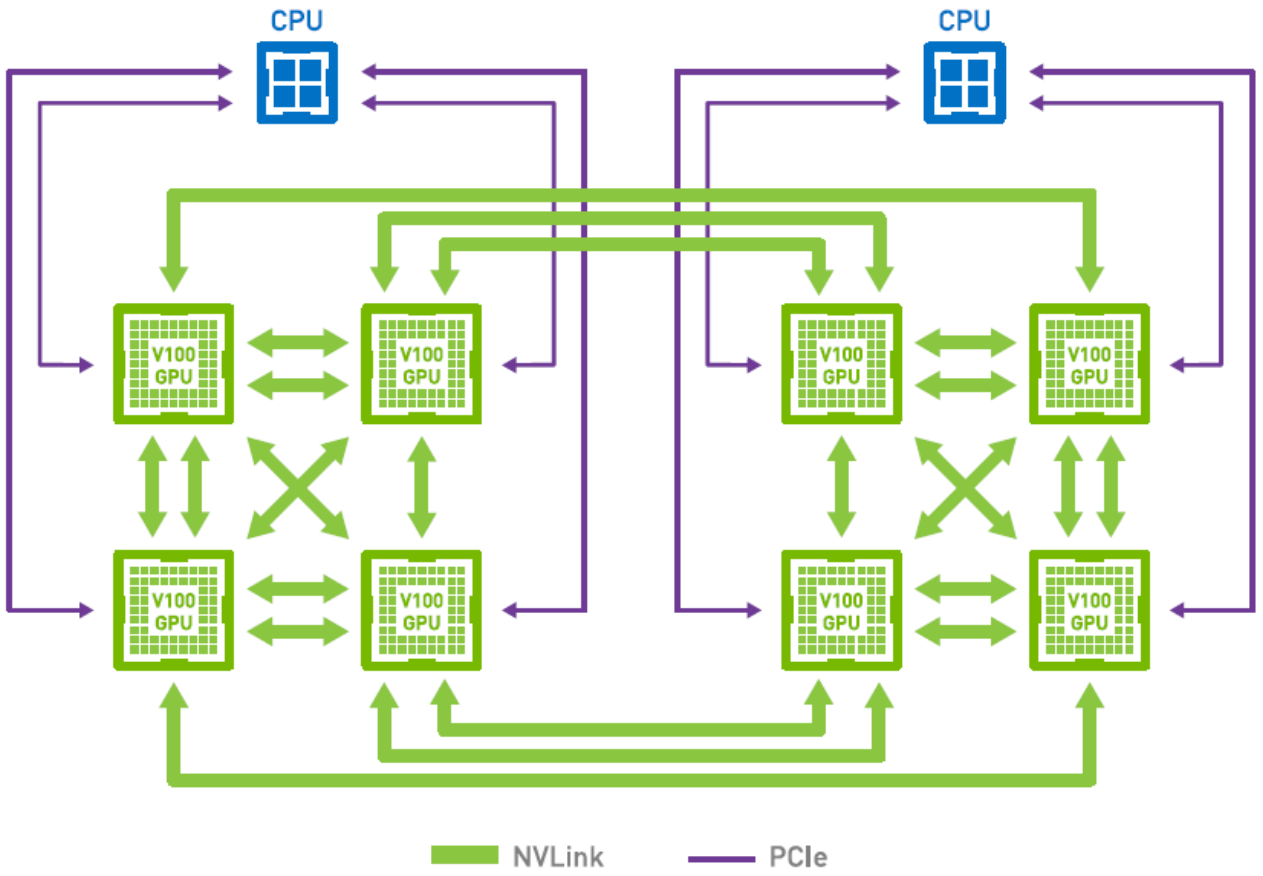
- Each node on a n-ring holds $\frac{1}{n}$ of the data
  - Node i holds part i of the data
- At each step $t = 0, \ldots, n-1$, each node
  - Sends part $i + t \bmod n$ data to its left neighbor
  - Receives part $i + t + 1 \bmod n$ data from its right neighbor
- After n steps, all nodes have all the data
- If T is the total time to send all the data
  - Each step takes time T/n
  - All n steps take time T

Ring Synchronization

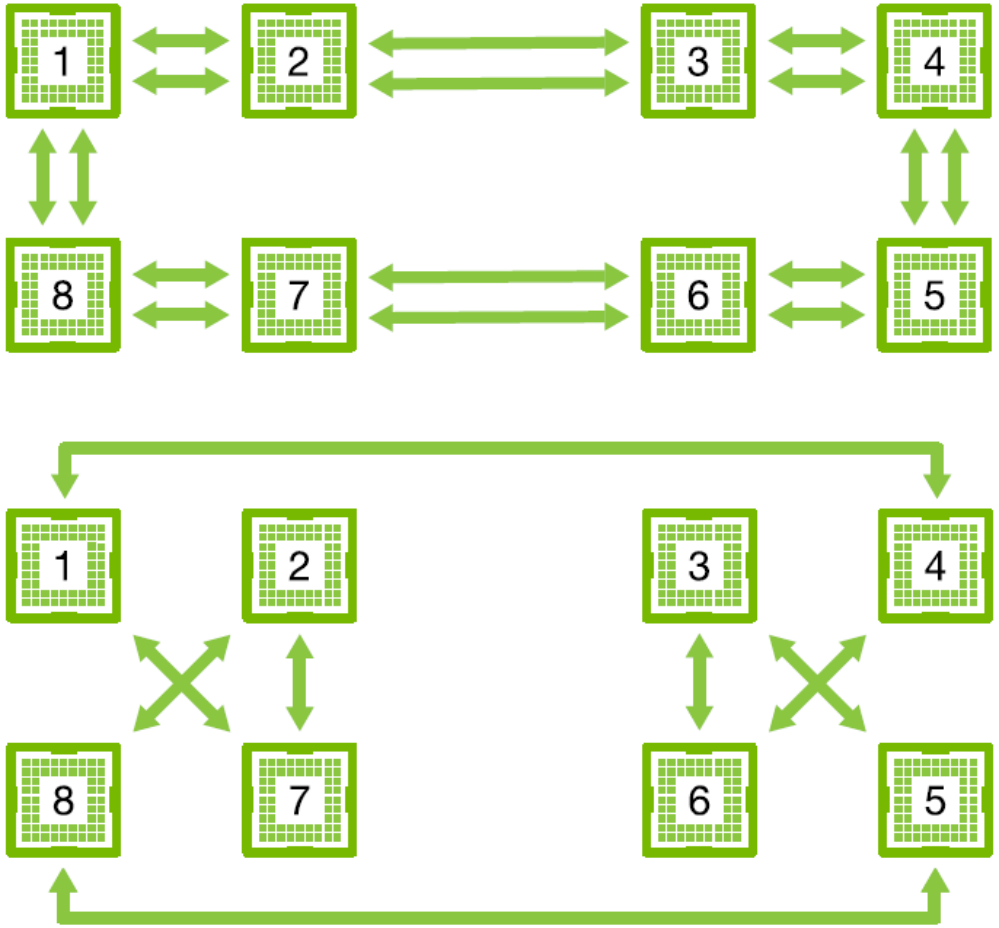


Custom Topology

NVLink on 8 V100 GPus



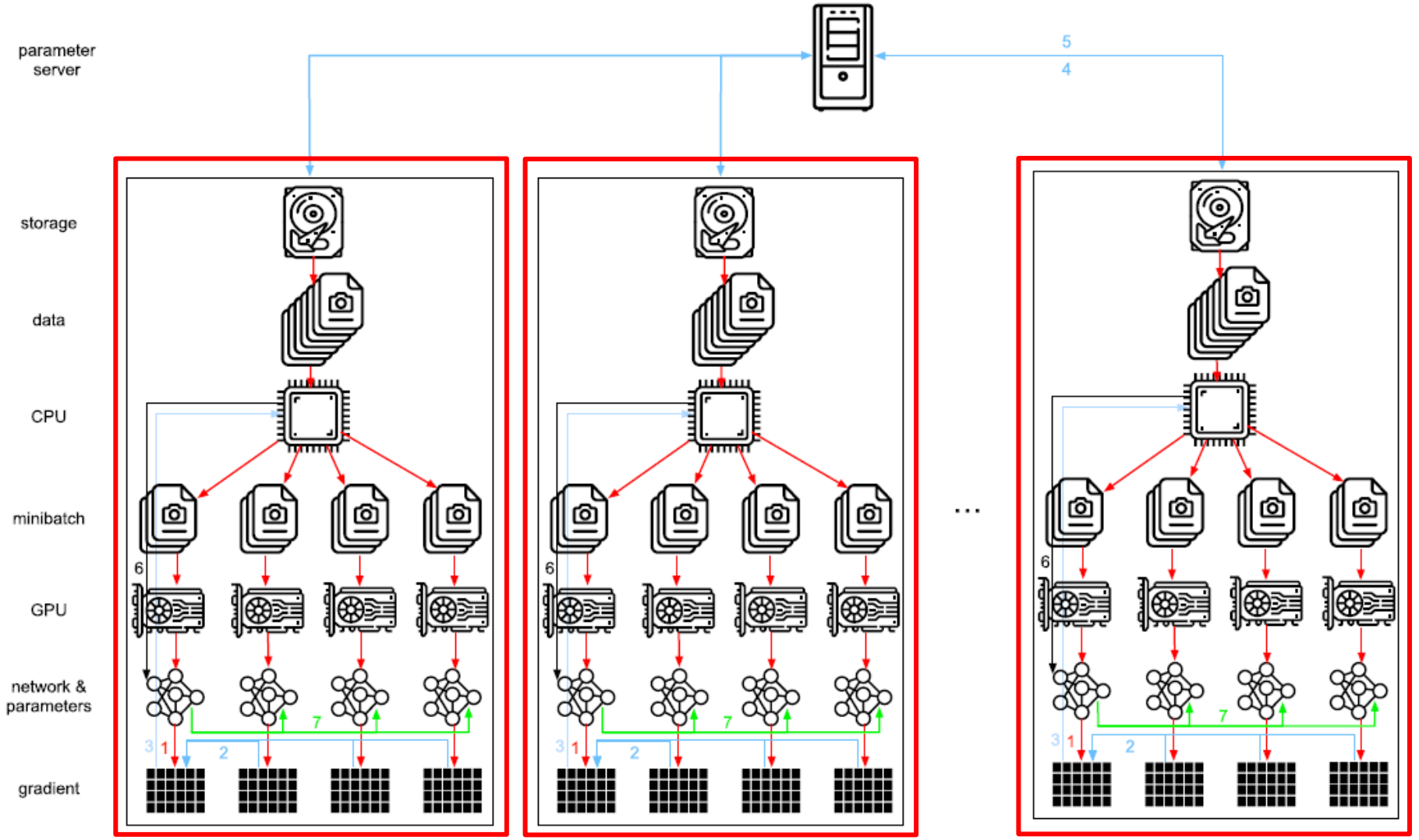
Decomposes into 2 rings



Multi-Machine Training

Train on more than one machine

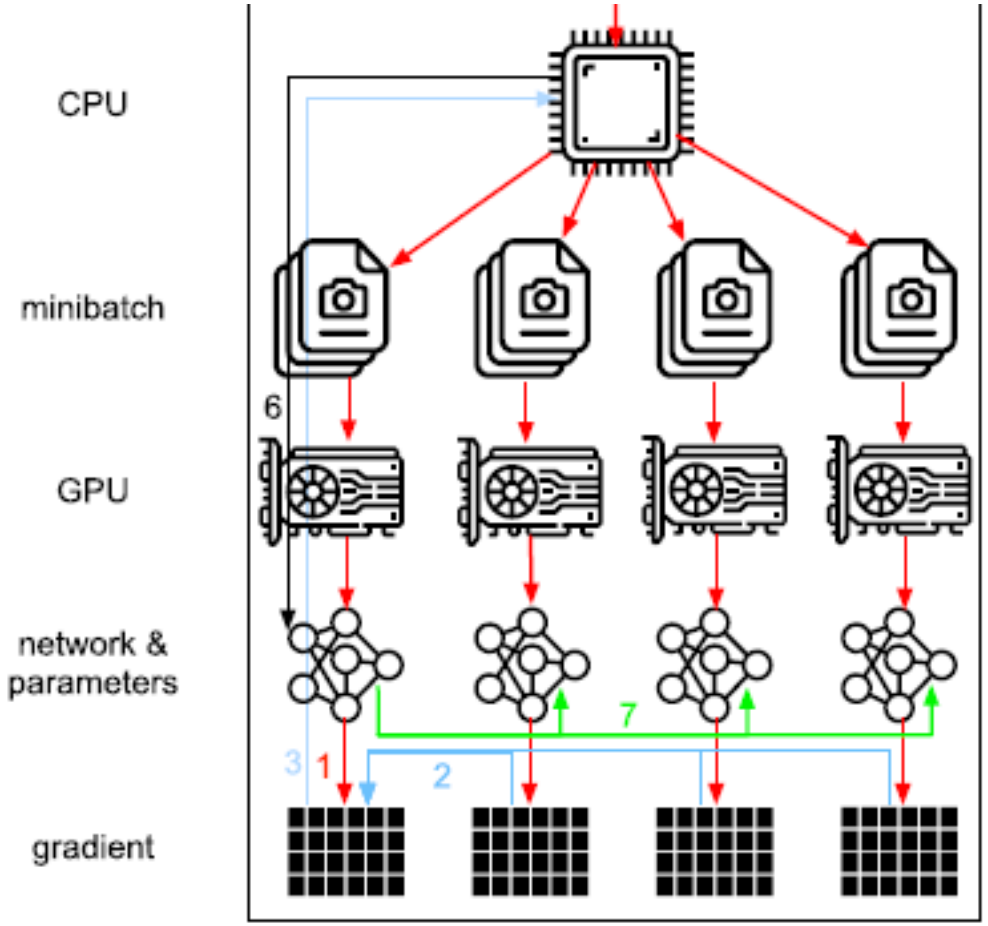
Need to synchronize tasks, since machine runs at slightly different speeds



Training with Centralize Parameter Server

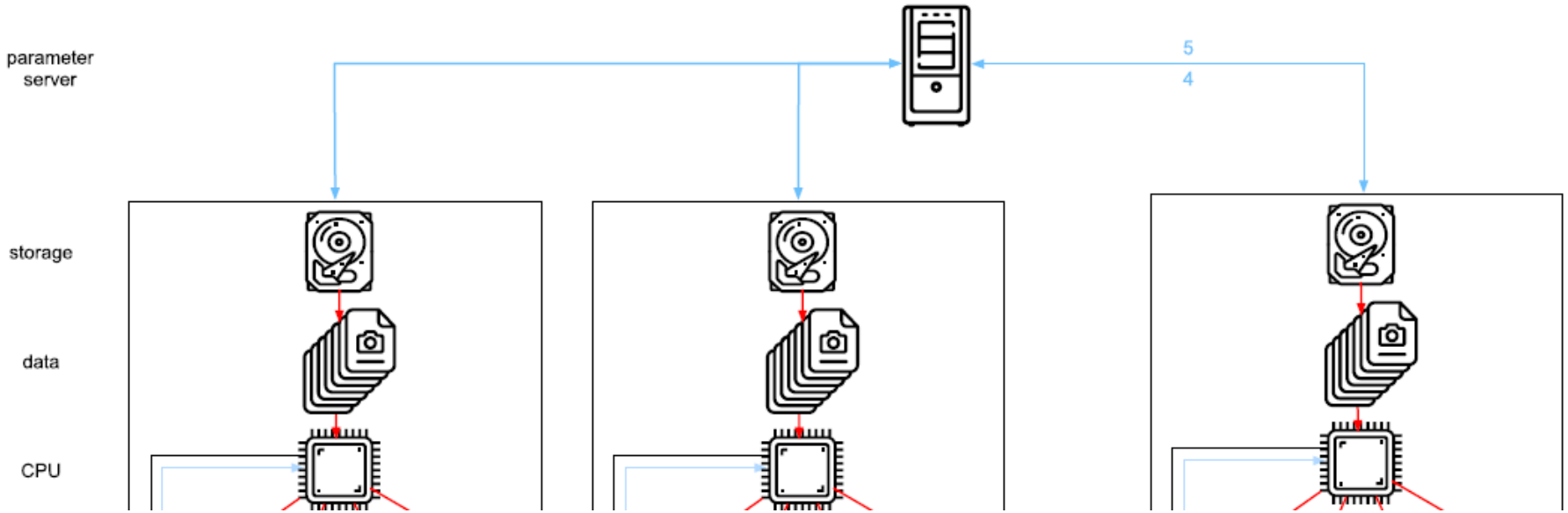
Centralize Parameter Server Example

1. Data split across machines and GPUs. Each GPU computes gradient
2. Gradients from local GPUs are aggregated on one GPU
3. Send gradients to CPU



Training with Centralize Parameter Server

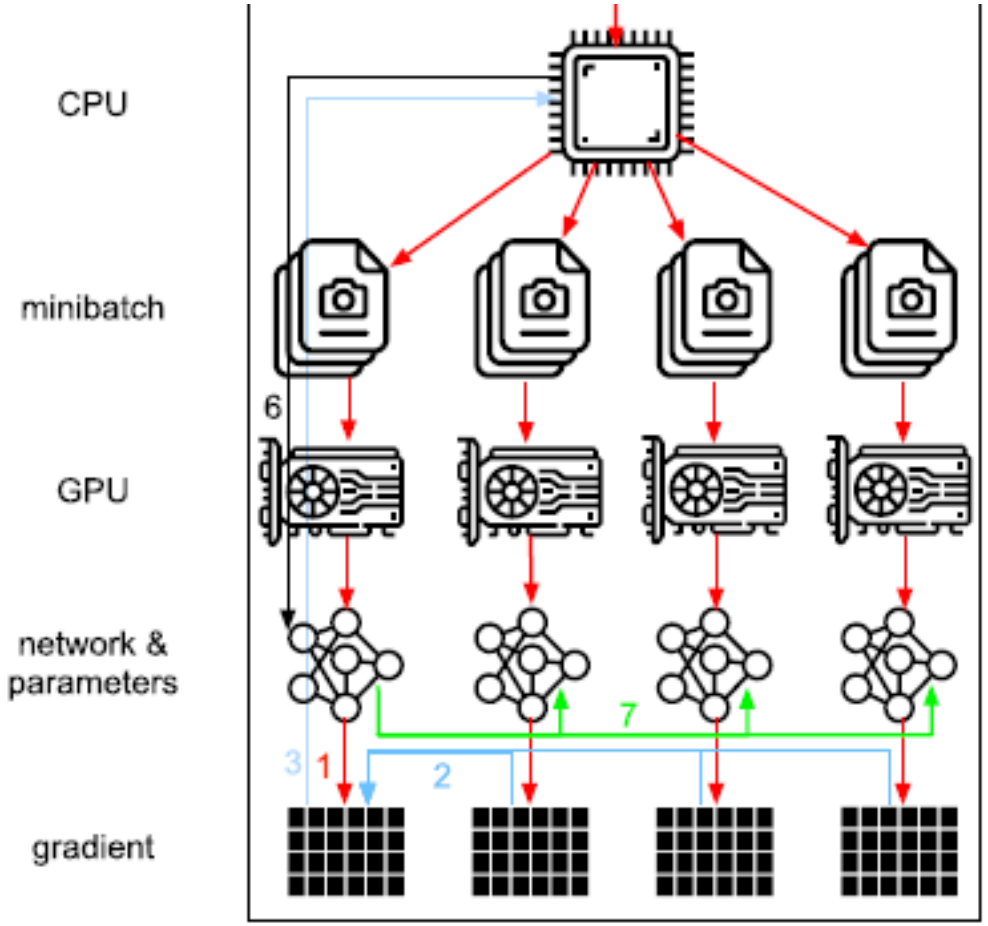
4. The CPUs send the gradients to a central parameter server
5. Update parameters with the aggregated gradients . Broadcast updated parameters to all CPUs



Training with Centralize Parameter Server

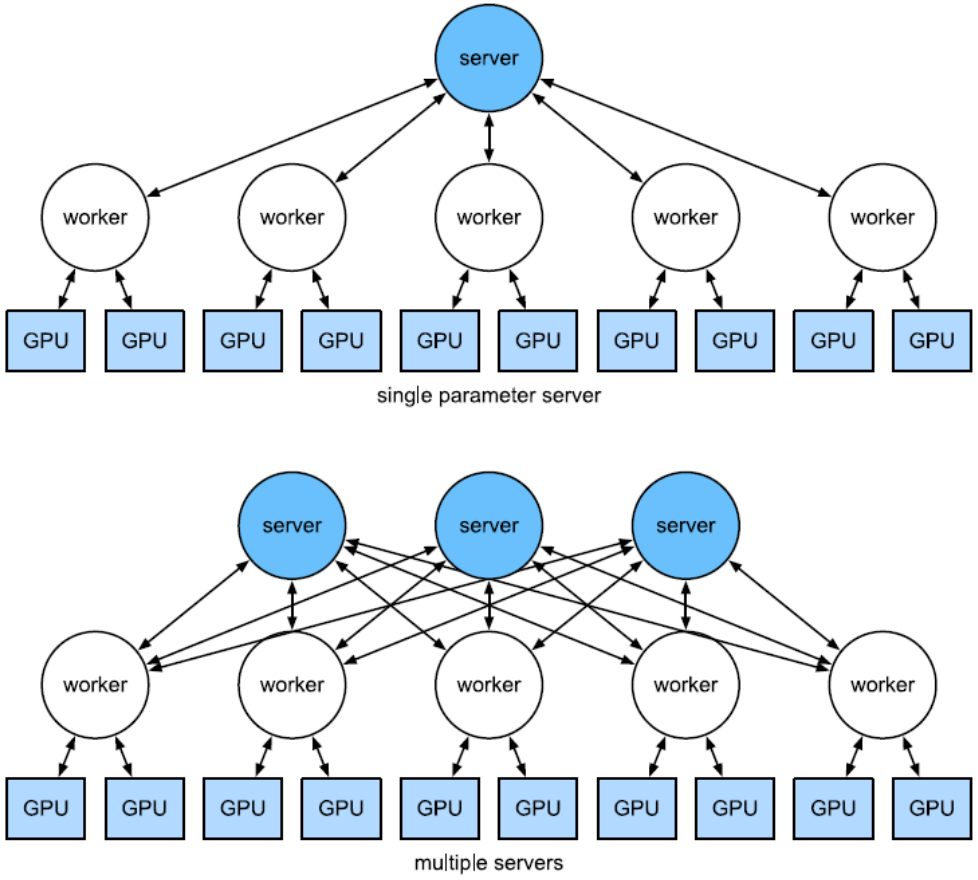
Centralize Parameter Server Example

6. Parameters are sent to one GPU
7. Updated parameters are spread across all GPUs



Distributed Parameter Server

- In general, having centralized server in distributed computing is a bad idea
- With m workers it takes $\mathcal{O}(m)$ to send gradients, due to server bandwidth limitations
- With n servers it takes $\mathcal{O}\left(\frac{m}{n}\right)$



Parameter Server Operations

- Implements specialized key-value stores
  - `push(key, value)` sends a particular gradient (value) from worker to a common storage. There the value is aggregated, e.g., summing.
  - `pull(key, value)` retrieves values from common storage
- Assumes the aggregation operation is commutative (e.g., sum). The order of the push does not matter
- Synchronization operations are hidden behind the push/pull operations