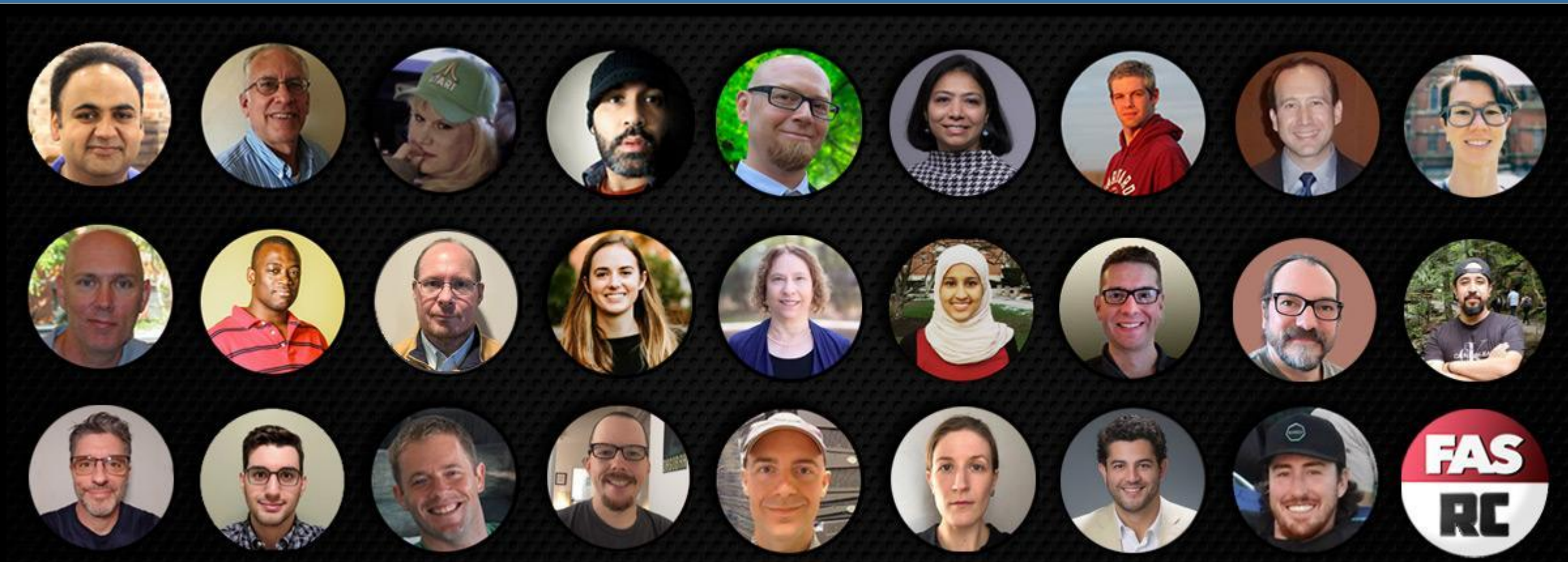


Distributed GPU Computing

Harvard - FAS Research Computing

Objectives

- Understand GPU communication paradigms
- Scale GPU workflows from single GPU to multi-GPU/multi-node
- Implement multi-GPU training with *PyTorch* DDP
- Evaluate parallelism strategies for your workloads
- Understand modern large-scale training

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

Training Materials

➔ Download slides

https://docs.rc.fas.harvard.edu/kb/training-materials/

➔ Login to Cannon

```
ssh <username>@login.rc.fas.harvard.edu
```

➔ Clone FASRC User Codes repository https://github.com/fasrc/User_Codes/tree/master

SSH: `git clone git@github.com:fasrc/User_Codes.git`

HTTPS: `git clone https://github.com/fasrc/User_Codes.git`

➔ Go to the training folder

```
cd User_codes/Training/Distributed_GPU_Computing/
```

Schedule

11:00am - 12:00pm - Introduction & Background
12:00pm - 12:30pm - Lunch
12:30pm - 1:00pm - MPI and CUDA
1:00pm - 1:30pm - Scaling on a single node
1:30pm - 1:40pm - *Break*
1:40pm - 2:10pm - NCCL (single- and multi-node)
2:10pm - 3:00pm - Distributed PyTorch

Workshop Reservation

Add the following to any jobs & be mindful of the partition:

Interactive:

```
salloc --reservation=workshop -p gpu
```

Batch:

```
#SBATCH --reservation=workshop
#SBATCH -p gpu
```

If you have problems, please let us know.

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

Required Memory Estimation

Just Loading Model Weights

- ➔ Model parameters in `bfloat16` (2 Bytes)
- ➔ Optimizer states in `float32` (4 Bytes each) — master weights + momentum + variance

$$MEM = (P \times 2) + (3 \times P) \times 4 = 14 \times P \text{ (Bytes)}$$

*P = Parameters

Note: This excludes gradients, activations, temporary buffers, communication buffers, sharding overhead, KV cache, fragmentation, and framework overhead. Real full training memory can be significantly larger.

Why Distributed GPU Computing?

The Scale Problem

- GPT-2 (2019): 1.5B parameters
- GPT-3 (2020): 175B parameters
- GPT-4 (2023): 1.7 T parameters
- GPT-5 (2026): ~5-50T parameters
- A single A100, or H200, GPU has 80 GB of memory

Why Distributed GPU Computing?

The Scale Problem

- GPT-2 (2019): 1.5B parameters 21 GB
- GPT-3 (2020): 175B parameters ~2.4 TB
- GPT-4 (2023): 1.7 T parameters ~24 TB
- GPT-5 (2026): ~5-50T parameters ~700TB
- A single A100, or H200, GPU has 80 GB of memory

Why Distributed GPU Computing?

The Scale Problem

- GPT-2 (2019): 1.5B parameters 21 GB
- GPT-3 (2020): 175B parameters ~2.4 TB **REPORTED / ESTIMATED VALUES**
- GPT-4 (2023): 1.7 T parameters ~24 TB
- GPT-5 (2026): ~5-50T parameters ~700TB
- A single A100, or H200, GPU has 80 GB of memory

Why Distributed GPU Computing?

The Scale Problem

- GPT-2 (2019): 1.5B parameters 21 GB
- GPT-3 (2020): 175B parameters ~2.4 TB **REPORTED / ESTIMATED VALUES**
- GPT-4 (2023): 1.7 T parameters ~24 TB
- GPT-5 (2026): ~5-50T parameters ~700TB
- A single A100, or H200, GPU has 80 GB of memory

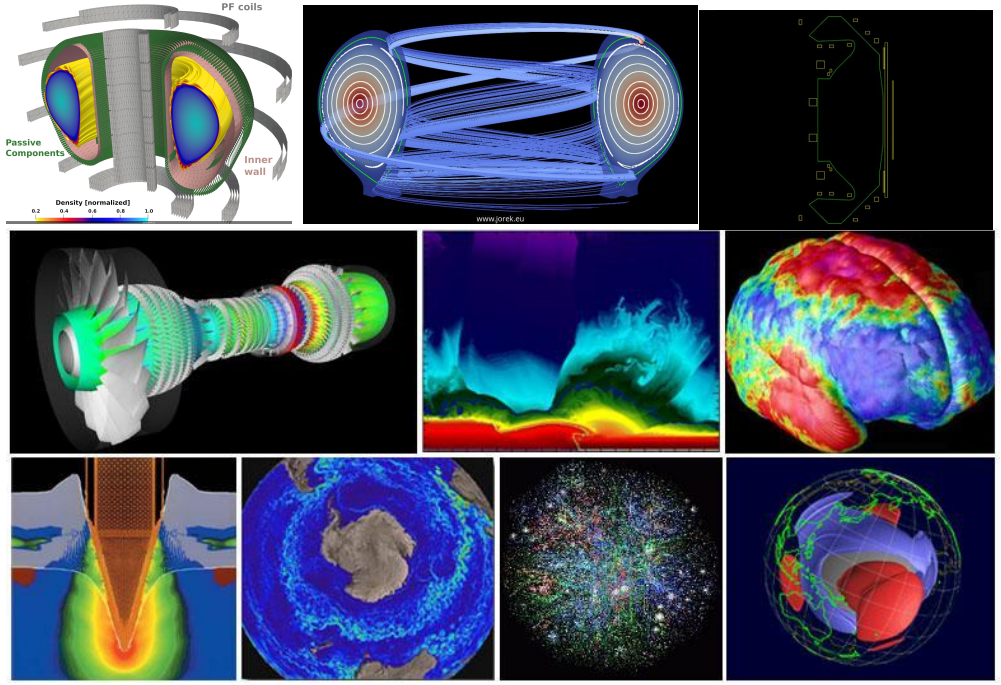
Why Go Distributed?

- **Memory:** Model doesn't fit on one GPU
- **Speed:** Training weeks → hours
- **Throughput:** Process more data per unit time
- **Science:** Simulations (MD, climate, CFD/MHD) at realistic scale

Why Distributed GPU Computing?

Use Cases Beyond LLMs

- Atmosphere, Earth, Environment,
- Weather forecasting and climate modeling
- **Physics** - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics
- Defense

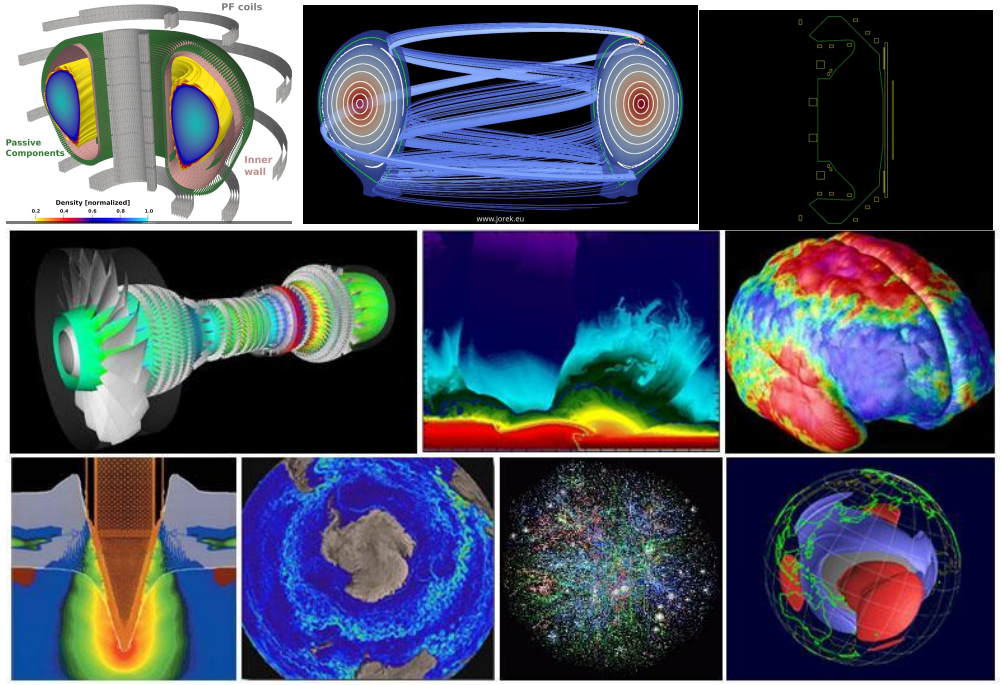


Credit: LLNL & JOREK.EU

Why Distributed GPU Computing?

Use Cases Beyond LLMs

- Atmosphere, Earth, Environment,
- Weather forecasting and climate modeling
- **Physics** - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics
- Defense



Credit: LLNL & JOREK.EU

The bottleneck shifts from compute to communication as you scale!

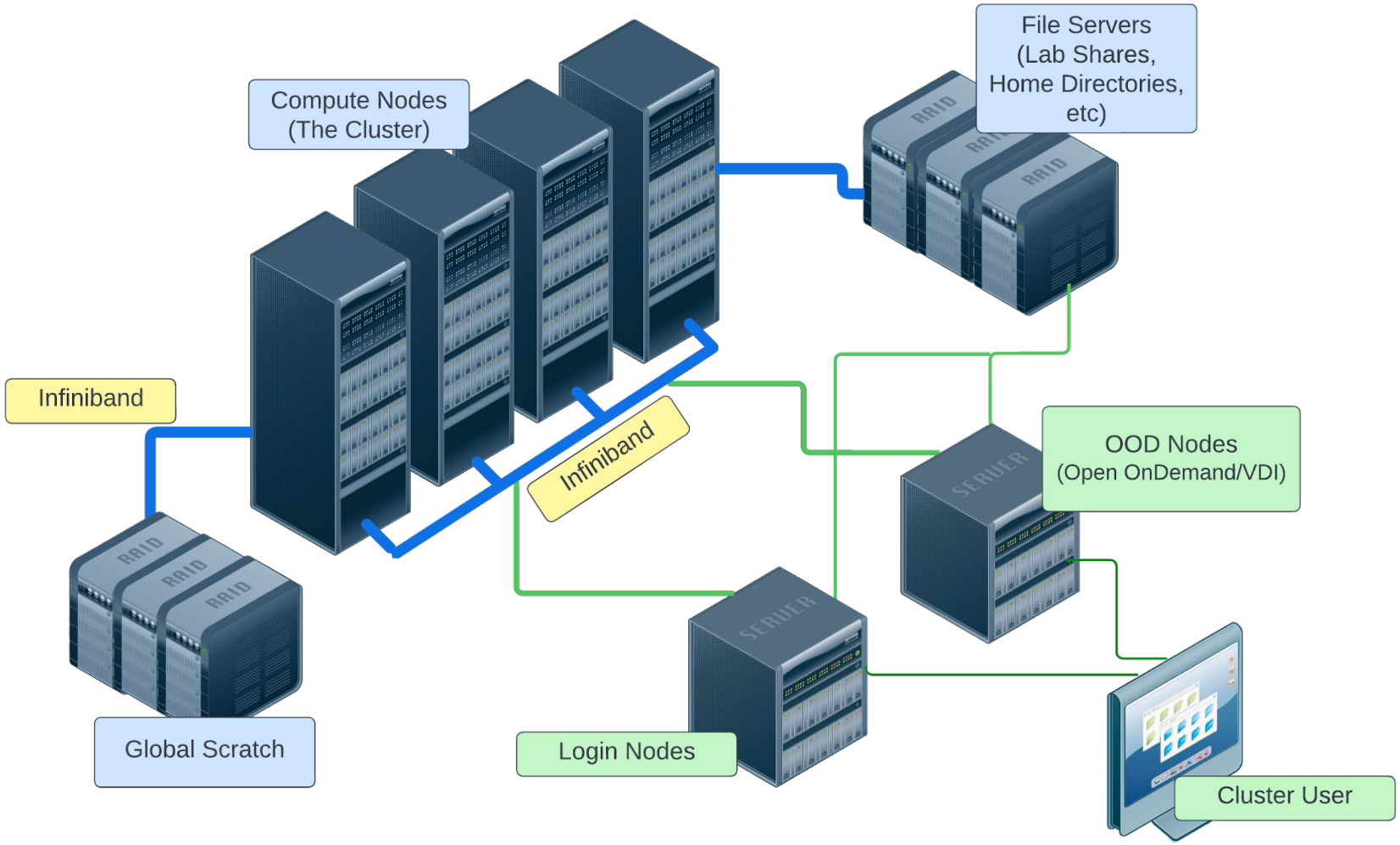
FASRC clusters

Massachusetts Green HPC Center (MGHPCC)

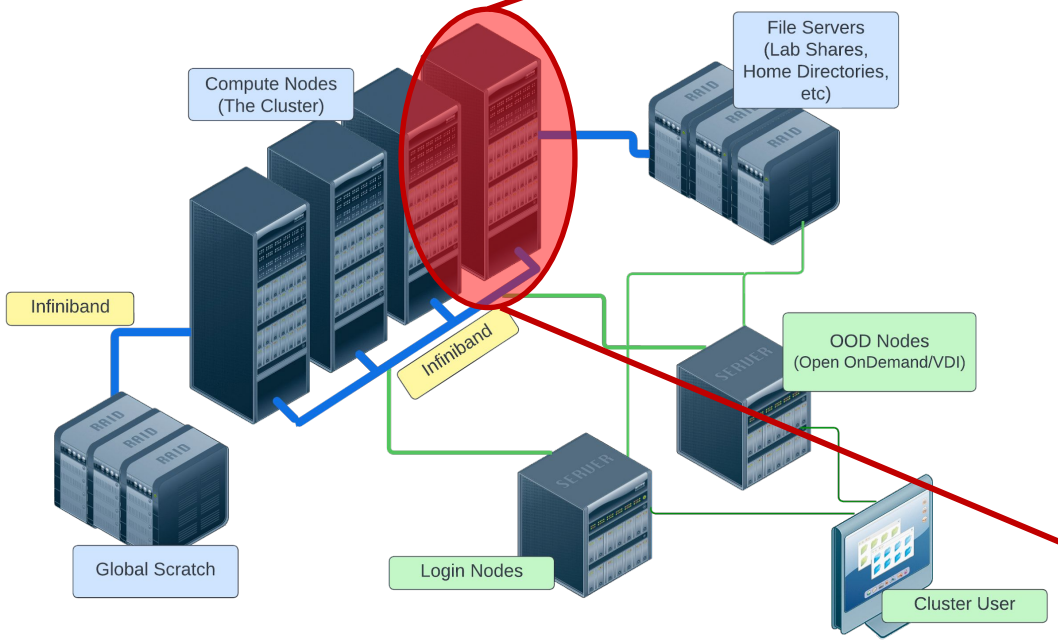
Cannon cluster

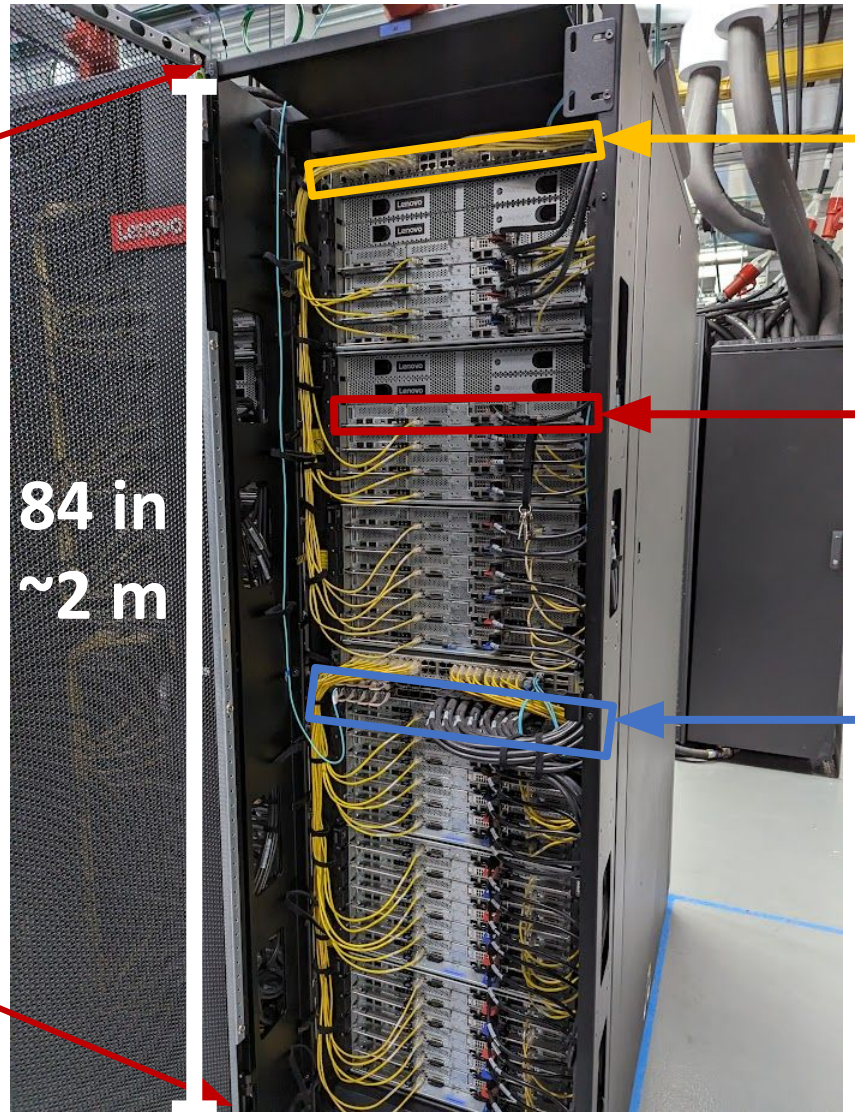
From https://www.servethehome.com/the-harvard-cannon-powered-by-lenovo-neptune/

Cluster architecture



Rack



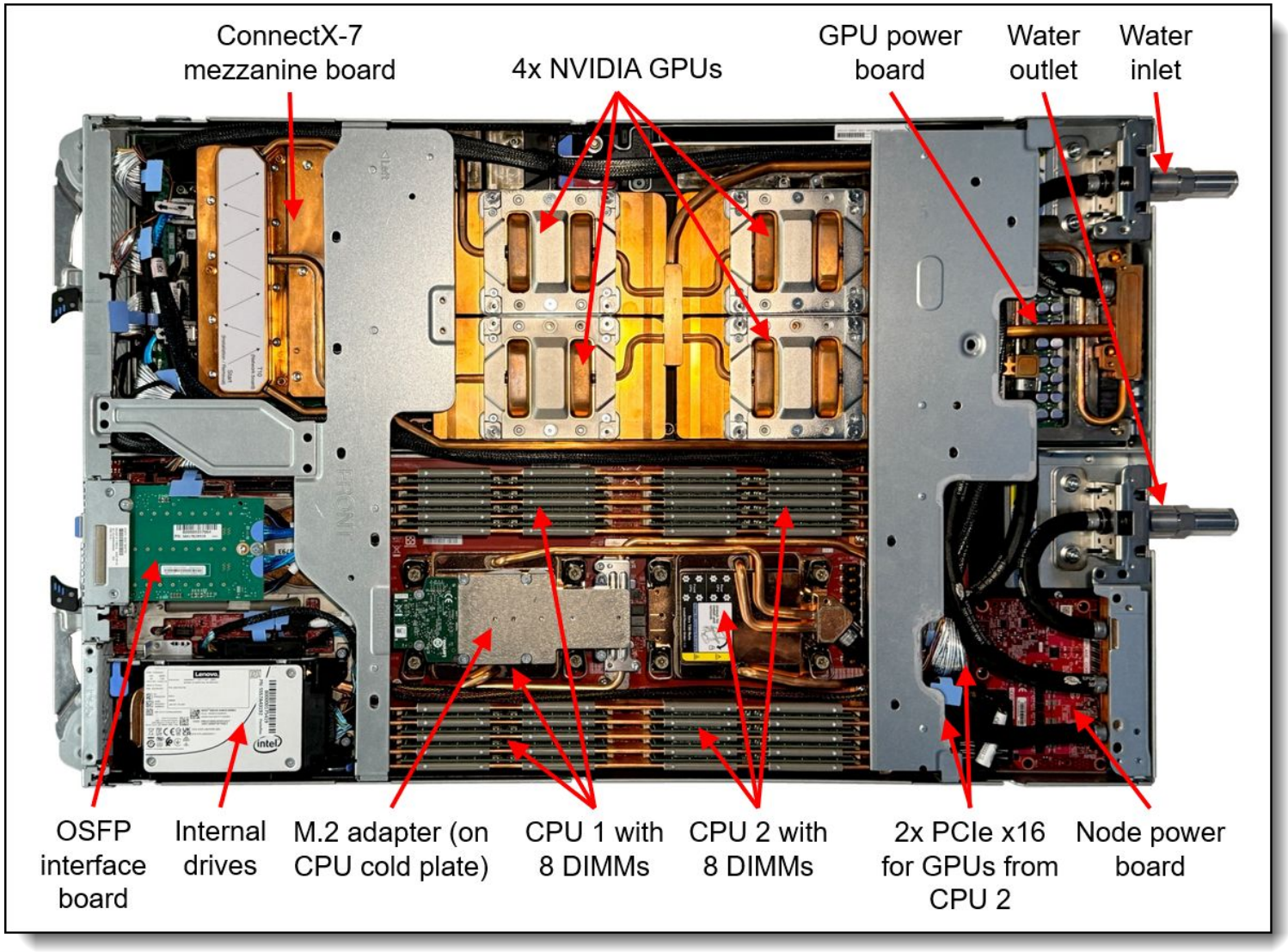


Ethernet switch

Compute nodes

Infiniband switch

GPU Compute Node



GPU Partitions

Documentation: https://docs.rc.fas.harvard.edu/kb/convenient-slurm-commands

```
[jharvard@boslogin08 ~]$ spart | grep -i gpu
Partition                State       Cores      GPUs       Average Mem/Node(GB)      Nodes      Time
Limit
gpu                      UP          2304       144        1007                      36
3-00:00:00
gpu_h200                 UP          2464       88         1007                      22
3-00:00:00
gpu_requeue              UP          24240      1336       1174                      316
3-00:00:00
gpu_test                 UP          768        96         503                       12         12:00:00
```

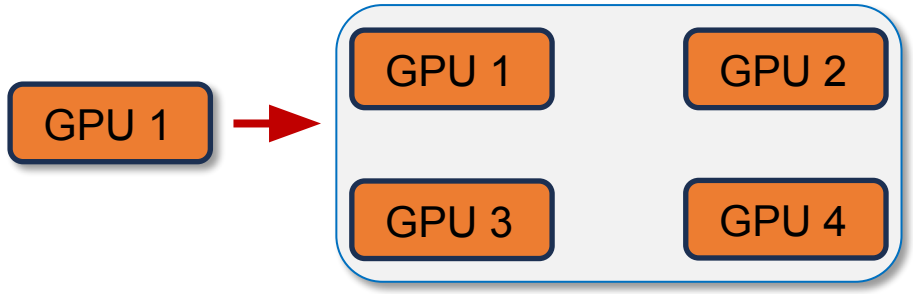
Scaling Levels

Single GPU

Scaling Levels

Single GPU

Multi-GPU (single node)

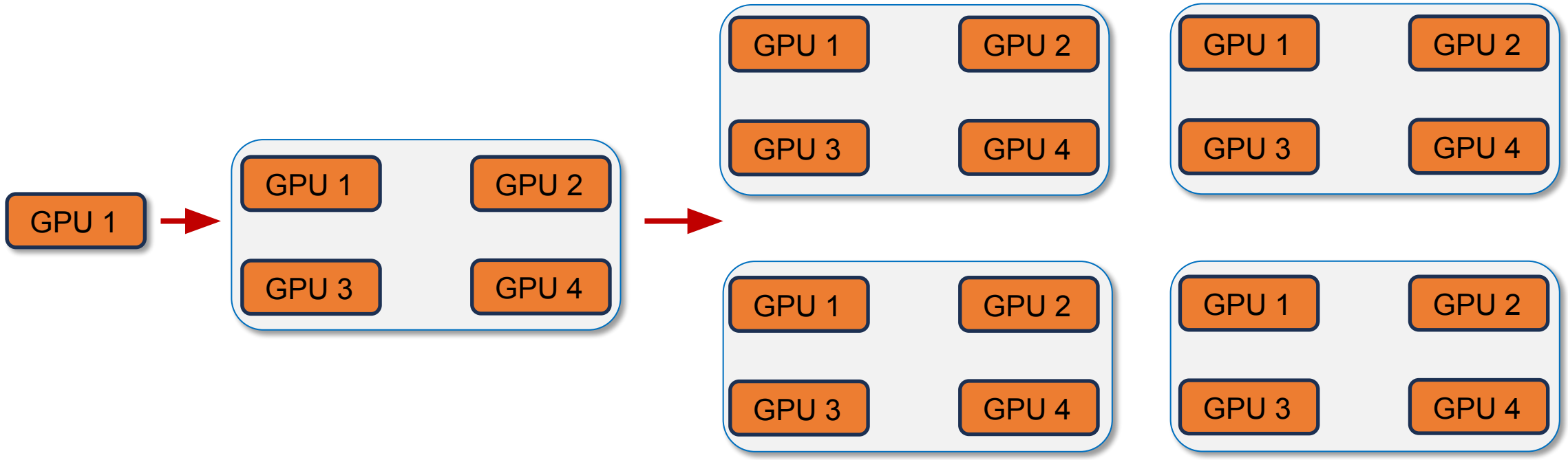


Scaling Levels

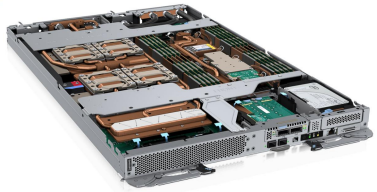
Single GPU

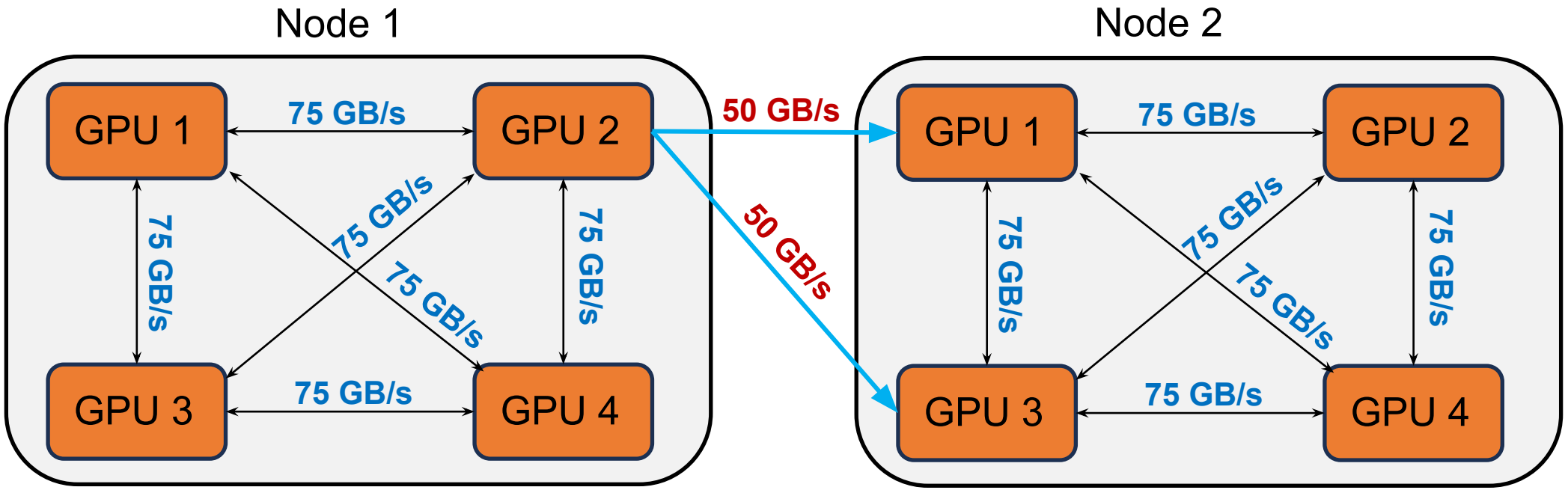
Multi-GPU (single node)

Multi-GPU (multi-node)



GPU - to - GPU Communication


H100 Node

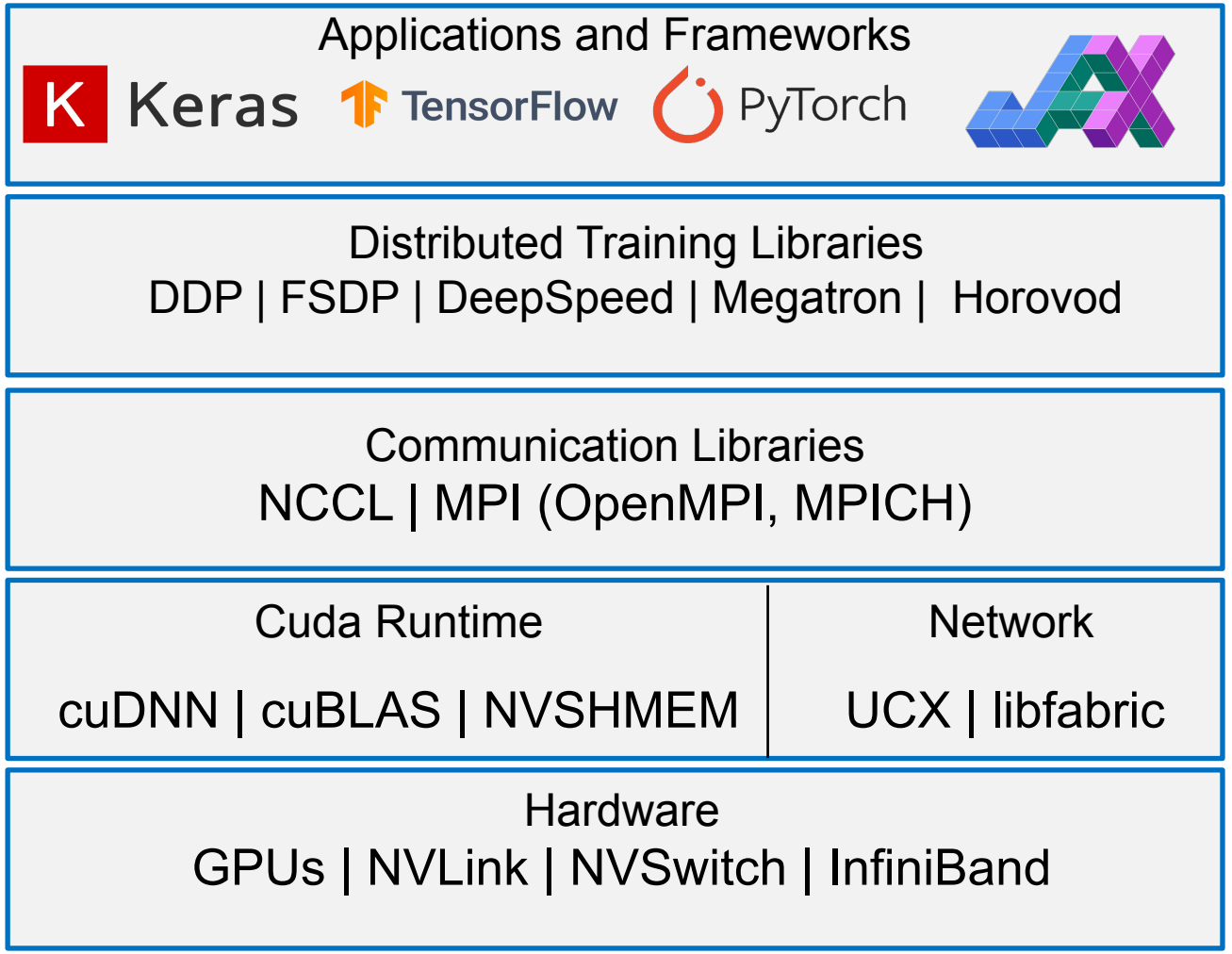


Inside the node (NVLink): Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed. 75 GB/s x 6 x 2 = 900 GB/s

Outside the node (InfiniBand Network NDR): Each GPU communicates to other GPUs on another node at 50 GB/s, totalling 400 GB/s

https://arxiv.org/pdf/2408.14090

The Distributed GPU Software Stack



Communication Primitives

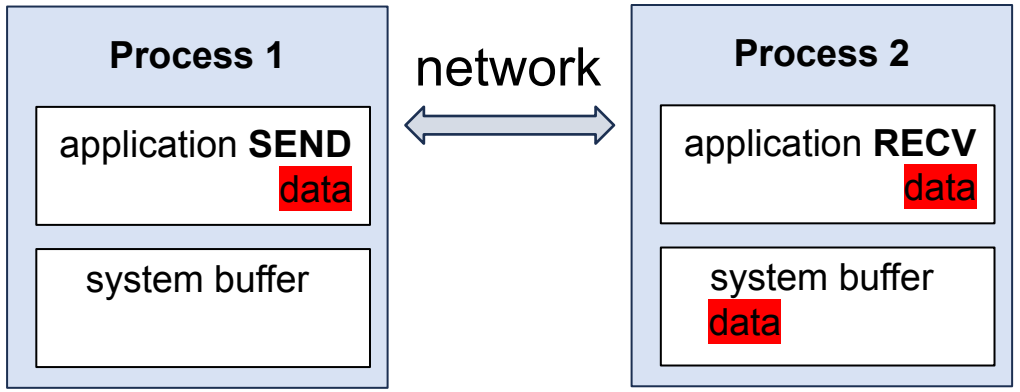
Point-to-Point

`Send` / `Recv` - direct message between two specific ranks

Communication Primitives

Point-to-Point

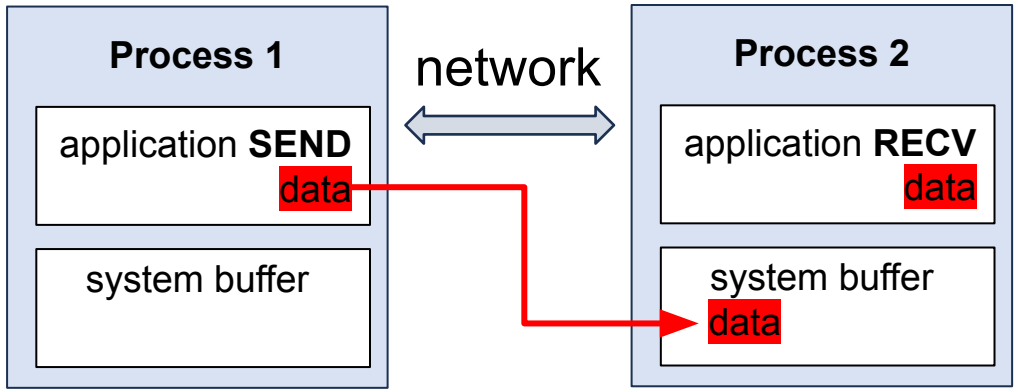
`Send` / `Recv` - direct message between two specific ranks



Communication Primitives

Point-to-Point

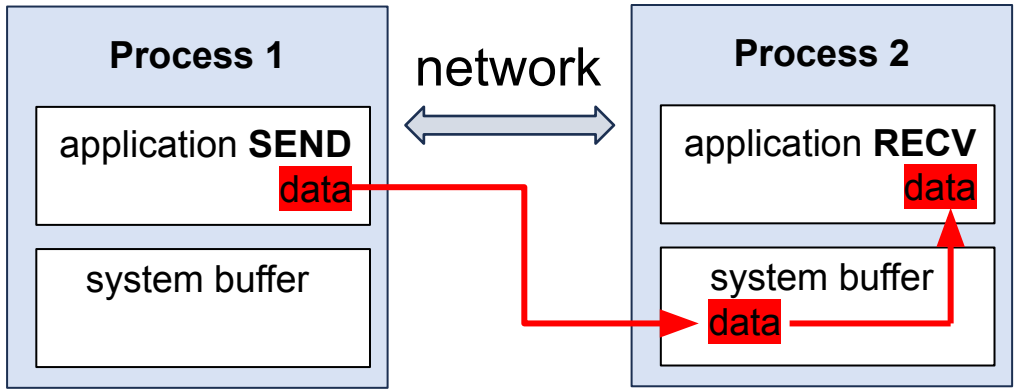
`Send` / **`Recv`** - direct message between two specific ranks



Communication Primitives

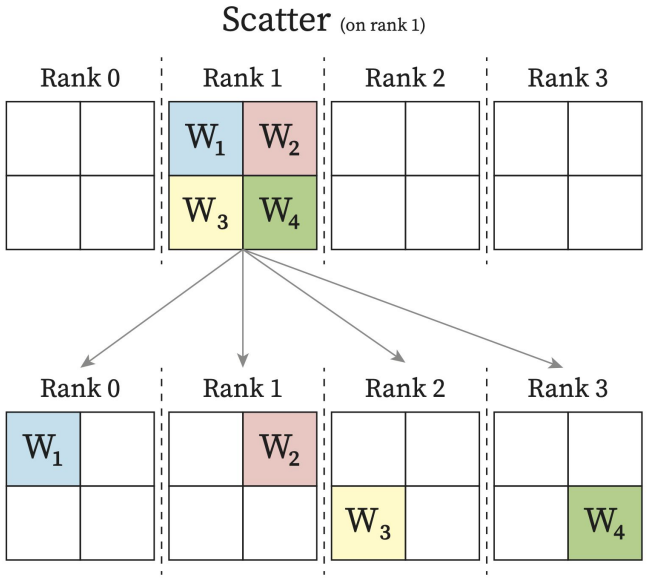
Point-to-Point

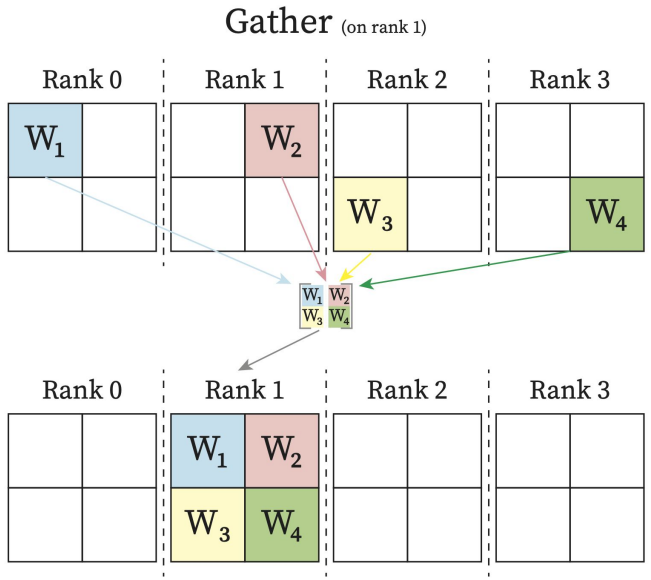
`Send` / **`Recv`** - direct message between two specific ranks

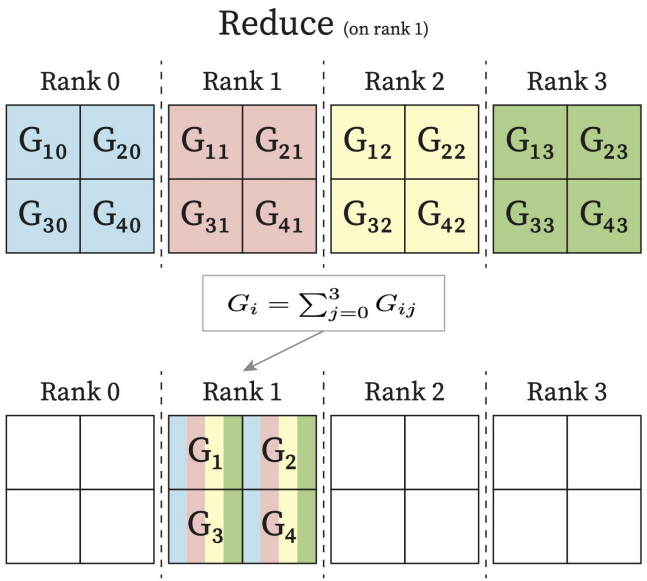


Communication Primitives (1)

Collective Operations (all ranks participate)







Scatter: From one rank, data will be distributed across all rank

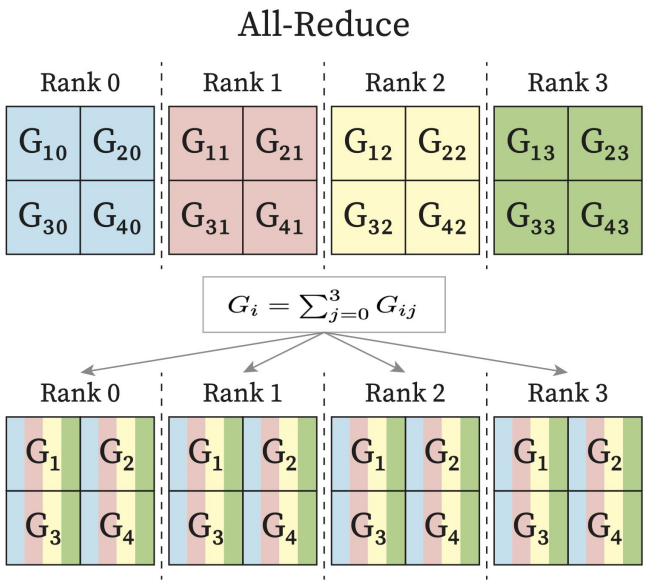
Gather: One rank will receive the aggregation of data from all ranks

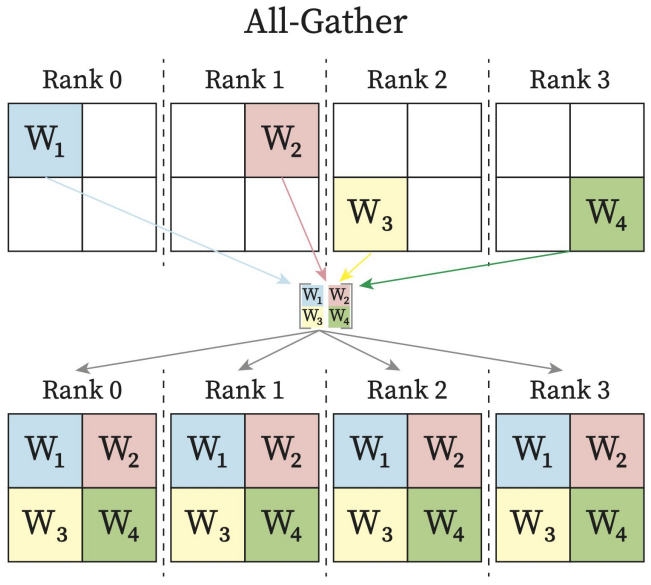
Reduce: One rank receives the reduction of input values across ranks

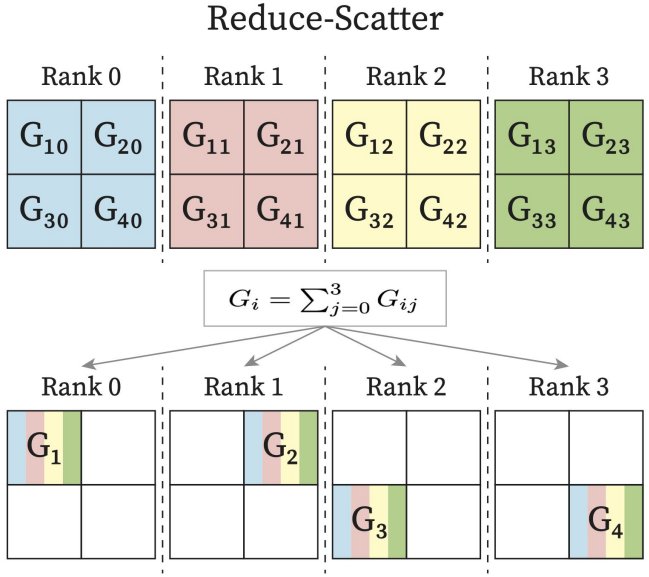
Figures from: https://handbook.eng.kempnerinstitute.harvard.edu/intro.html

Communication Primitives (2)

Collective Operations (all ranks participate)







AllReduce: Each rank receives the reduction of input values across ranks (SUM, MAX, etc.)

AllGather: Each rank receives the aggregation of data from all ranks in the order of the ranks

ReduceScatter: Input values are reduced across ranks, with each rank receiving a subpart of the result

Figures from: https://handbook.eng.kempnerinstitute.harvard.edu/intro.html

AllReduce: The Heart of Data Parallelism

What it does: Every process contributes data every process receives the global reduction (sum, max, min etc.)

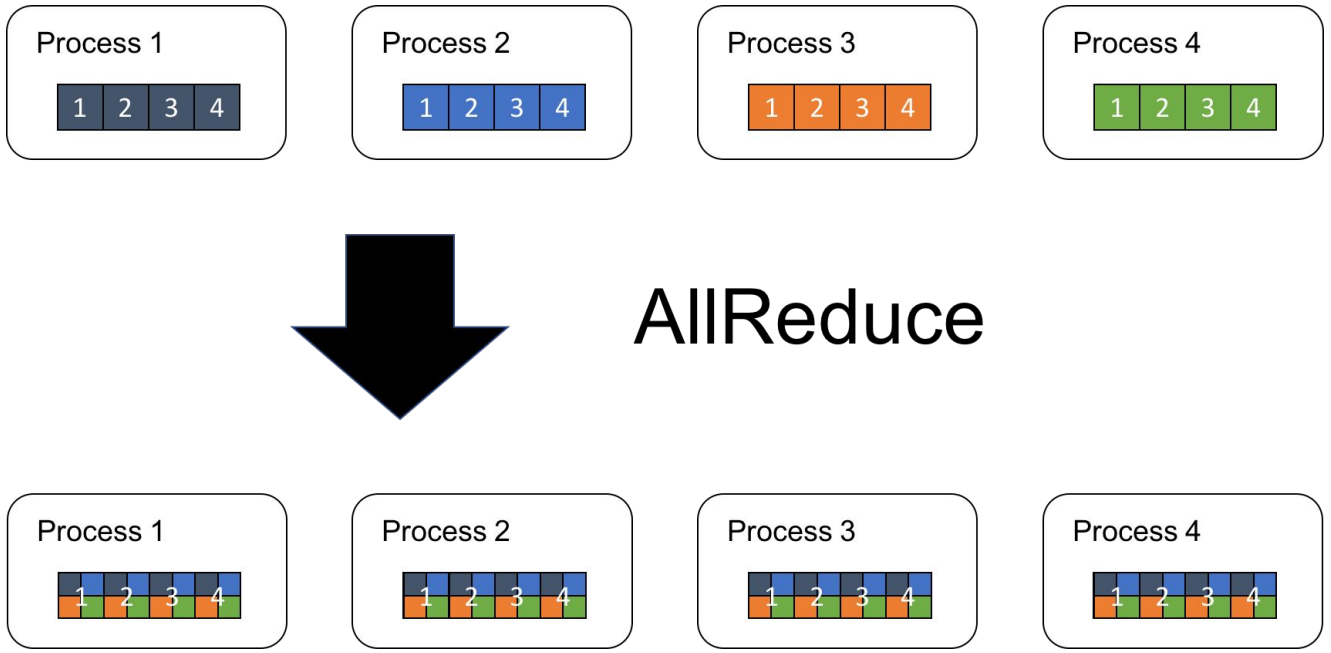
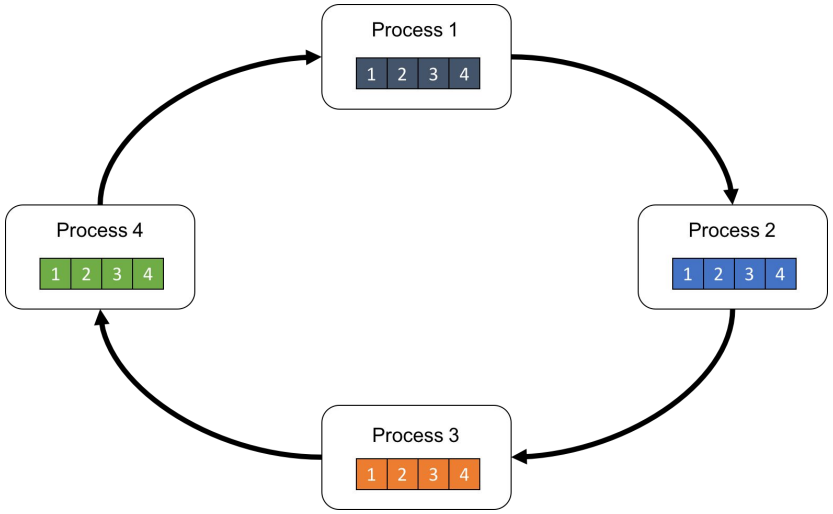
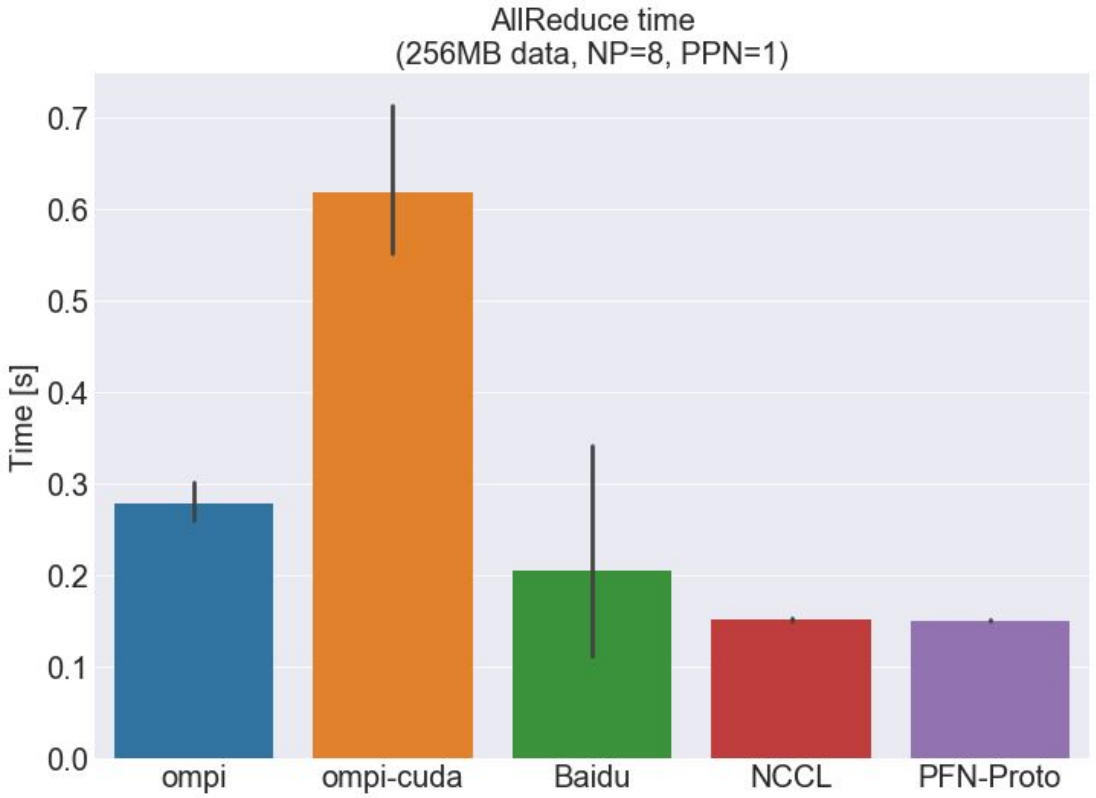


Figure from:
https://tech.preferred.jp/

In distributed deep learning, the SUM operation is used to compute the mean of gradients.

Ring-AllReduce Algorithm (bandwidth optimal)





Why it's efficient:

- Data sent per GPU = **$2 \times (N-1)/N \times \text{array\_size} \approx 2 \times \text{array\_size}$**
- **Independent of N** - scales perfectly!
- NCCL achieves ~90%+ of theoretical NVLink bandwidth with ring AllReduce

Figures from:
https://tech.preferred.jp/

Exercise 0

Given:
GPU0 = [1], GPU1 = [2], GPU2 = [3], GPU3 = [4]

What happens for:

- Reduce (SUM)
- AllReduce (SUM)
- AllGather

Exercise 0 - Answers

Given:
GPU0 = [1], GPU1 = [2], GPU2 = [3], GPU3 = [4]

What happens for:

- Reduce (SUM): GPU0 = [10] , GPU1 = [2], GPU2 = [3], GPU3 = [4]
- AllReduce (SUM): GPU0 = [10] , GPU1 = [10], GPU2 = [10], GPU3 = [10]
- AllGather: GPU0 = [1, 2, 3, 4], GPU1 = [1, 2, 3, 4], GPU2 = [1, 2, 3, 4], GPU3 = [1, 2, 3, 4]

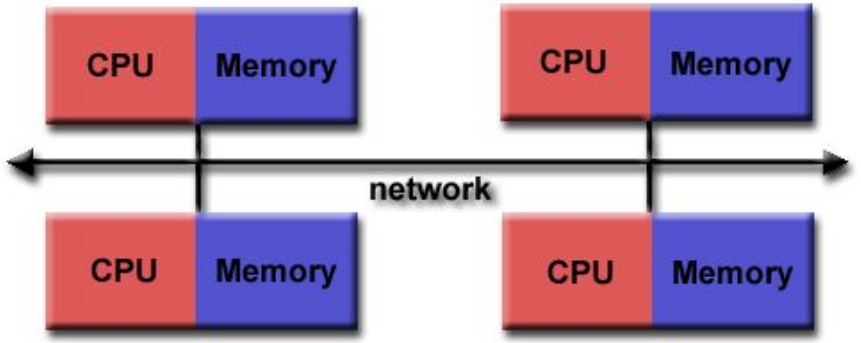
Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

What is MPI?

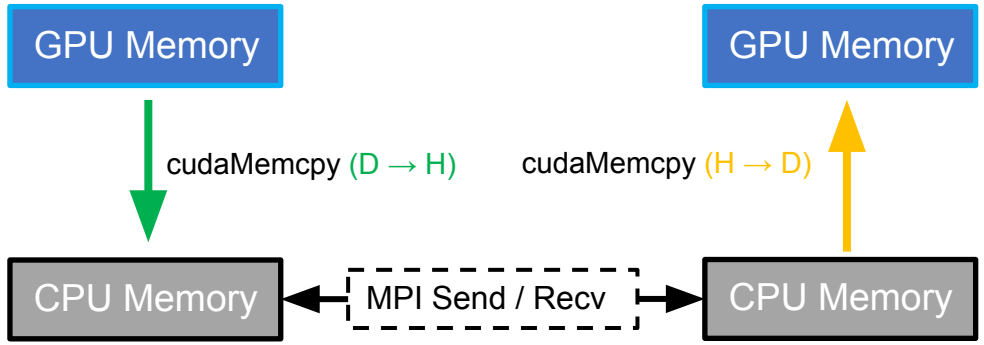
Message Passing Interface

- The de facto standard for distributed HPC since 1994
- Process-based parallelism: each process has its own memory space
- Processes communicate by explicitly sending and receiving messages
- Portable: runs on laptops → supercomputers



CUDA-Aware MPI

Traditional MPI workflow

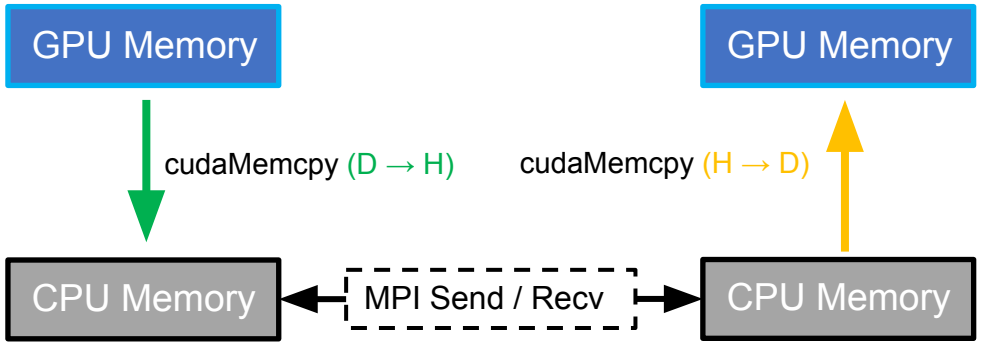


MPI communication happens through **CPU (host) memory**, so GPU data must first be copied:

1. GPU → CPU (**D** → **H**)
2. CPU → CPU via MPI
3. CPU → GPU (**H** → **D**)

CUDA-Aware MPI

Traditional MPI workflow

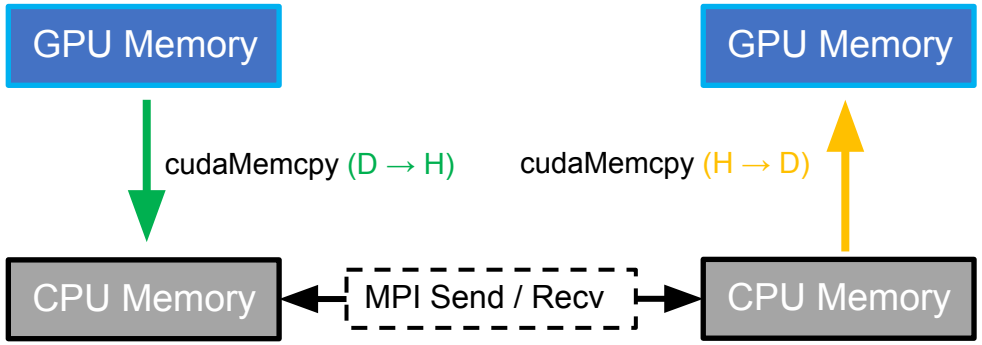


MPI communication happens through **CPU (host) memory**, so GPU data must first be copied:

1. GPU → CPU (**D** → **H**)
2. CPU → CPU via MPI
3. CPU → GPU (**H** → **D**)

CUDA-aware MPI (without GPUDirect)

- ✔ GPU pointers accepted
- ✔ CPU staging hidden



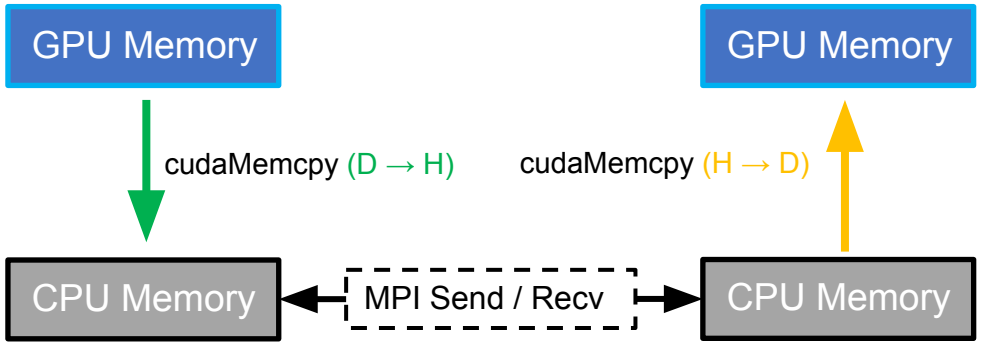
Hidden GPU → CPU → Network → CPU → GPU path

GPU-aware MPI "understands" GPU memory, but internally still does:

1. GPU → CPU (**D** → **H**)
2. CPU → CPU via MPI
3. CPU → GPU (**H** → **D**)

CUDA-Aware MPI

Traditional MPI workflow

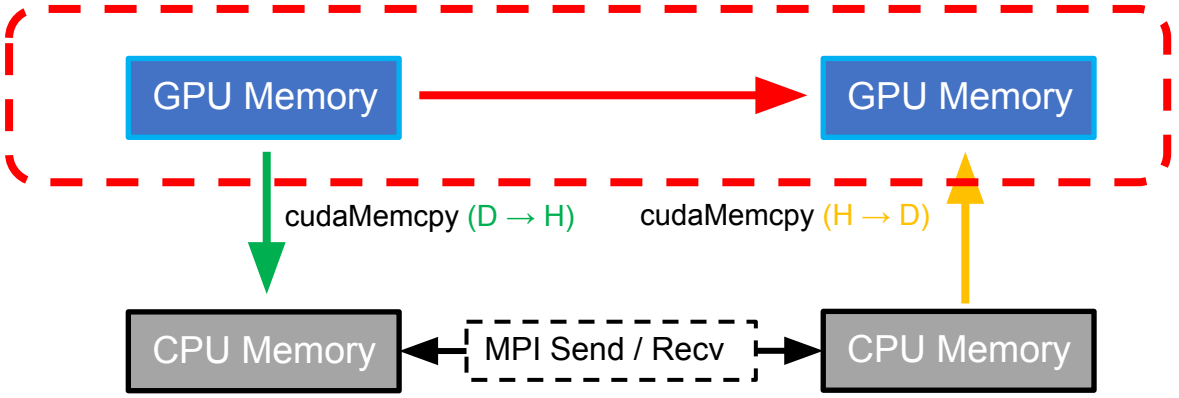


MPI communication happens through **CPU (host) memory**, so GPU data must first be copied:

1. GPU → CPU (**D** → **H**)
2. CPU → CPU via MPI
3. CPU → GPU (**H** → **D**)

CUDA-aware MPI (without GPUDirect)

✔ GPU pointers accepted
✔ CPU staging hidden

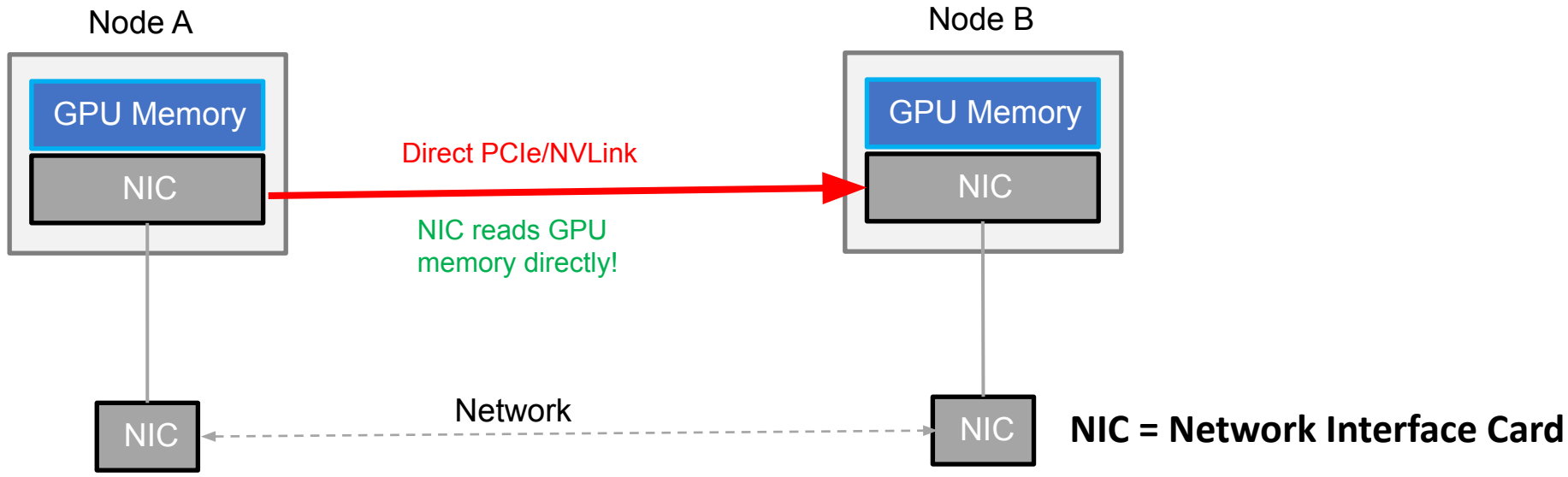


Hidden GPU → CPU → Network → CPU → GPU path

GPU-aware MPI "understands" GPU memory, but internally still does:

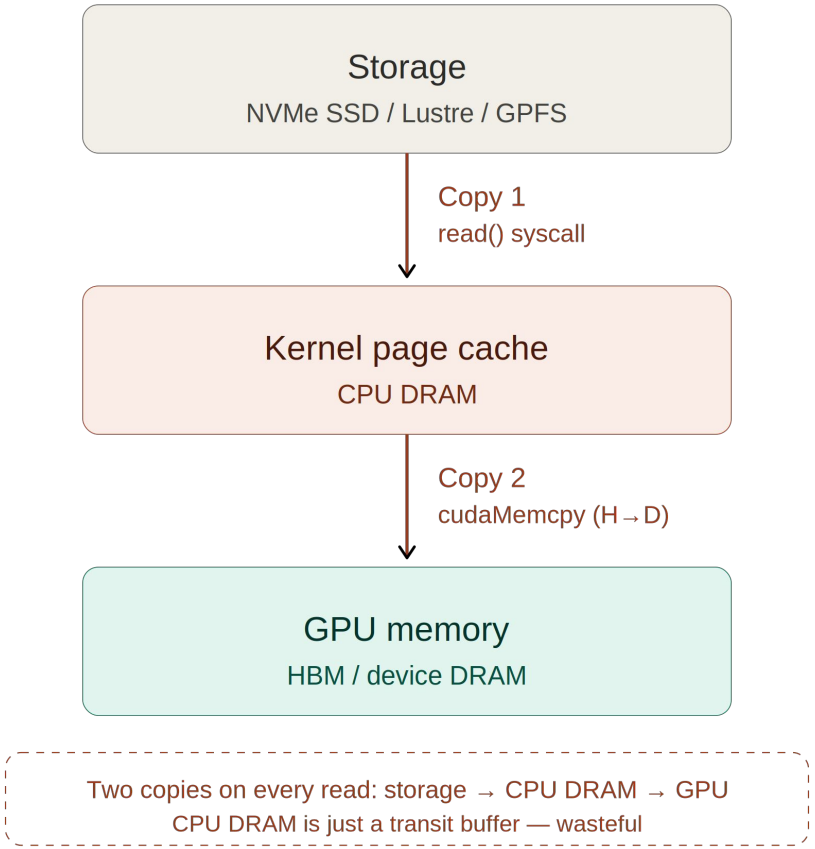
1. GPU → CPU (**D** → **H**)
2. CPU → CPU via MPI
3. CPU → GPU (**H** → **D**)

GPUDirect RDMA (Remote Direct Memory Access)



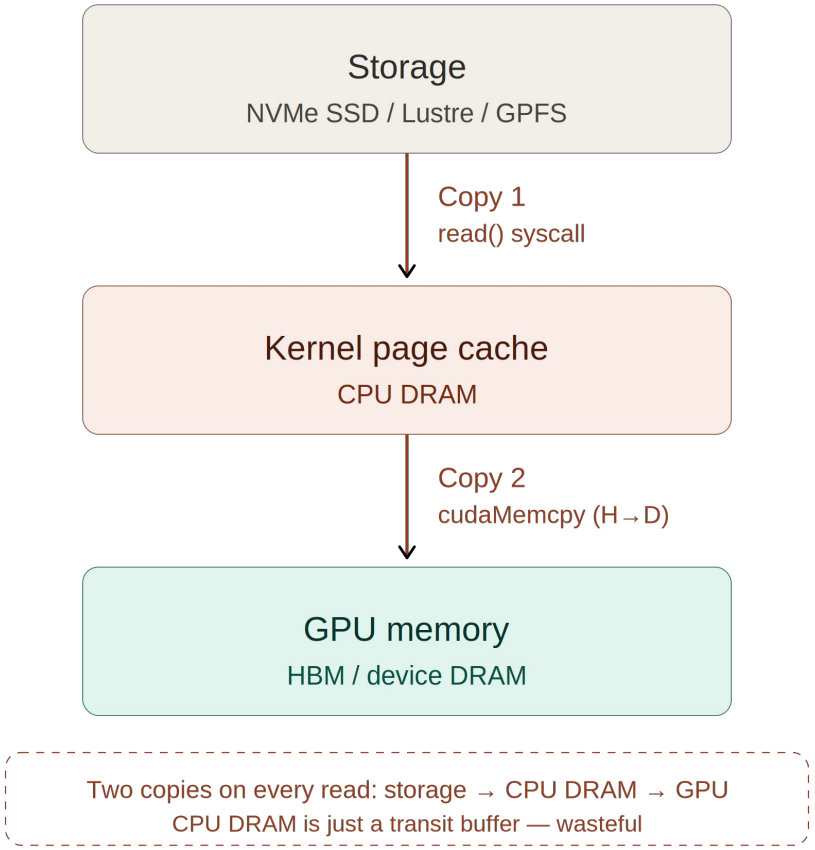
- GPUDirect RDMA enables the NIC to access GPU memory directly
- Data is transferred over the network without any CPU involvement
- Significantly reduces data transfer latency for GPU-GPU communication
- Ideal for high-performance applications that require minimal overhead and maximum throughput

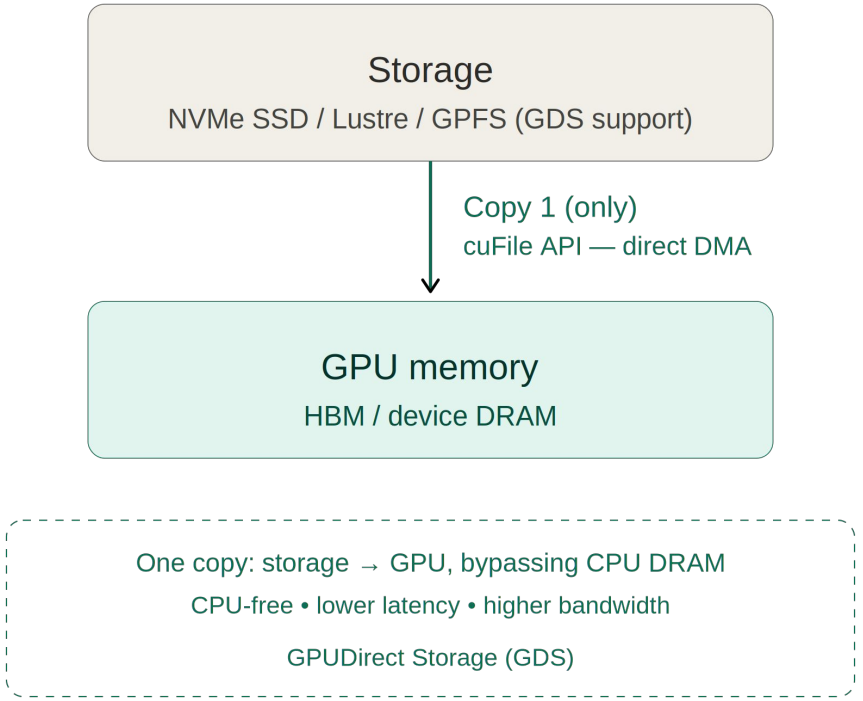
GPUDirect Storage (GDS)



Traditional storage → GPU data path: data is first read into CPU DRAM (page cache) and then copied to GPU memory, resulting in extra copies and CPU overhead.

GPUDirect Storage (GDS)



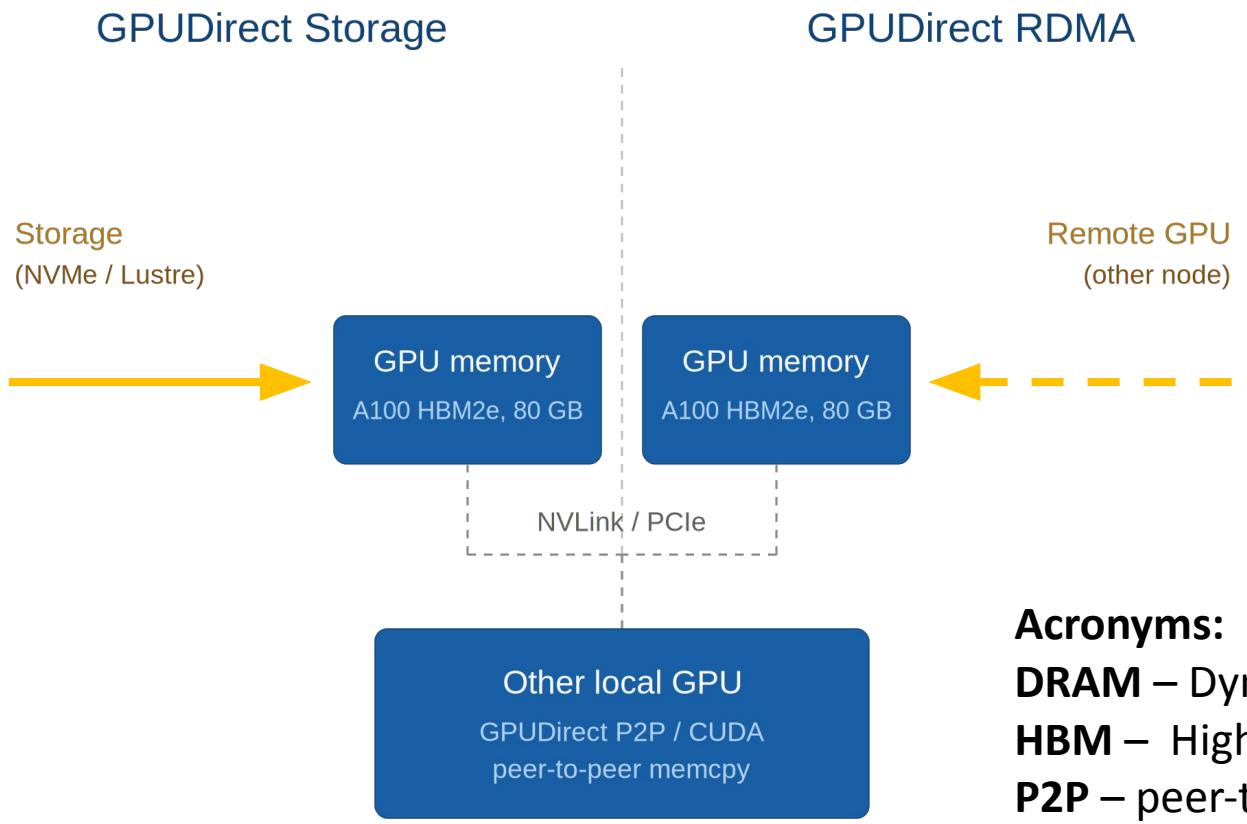


Traditional storage → GPU data path: data is first read into CPU DRAM (page cache) and then copied to GPU memory, resulting in extra copies and CPU overhead.

https://developer.nvidia.com/blog/gpudirect-storage

GPUDirect Storage (GDS) path: data is transferred directly from storage to GPU memory via DMA, bypassing CPU DRAM, eliminating extra copies, and reducing latency and overhead.

The Full GPUDirect Picture



The GPUDirect family: all paths bypass CPU DRAM
Storage, network, and peer GPUs write directly into HBM

Acronyms:
DRAM – Dynamic Random Access Memory
HBM – High Bandwidth Memory
P2P – peer-to-peer

Three Flavors of GPUDirect

Technology	What it does	Requires
GPUDirect P2P	GPU ↔ GPU (same node), no CPU involvement	NVLink or PCIe P2P
GPUDirect RDMA	GPU ↔ GPU (across nodes), no CPU involvement	InfiniBand / RoCE NIC + nvidia-peermem
GPUDirect Storage	Storage ↔ GPU, bypasses CPU DRAM	cuFile + GDS-capable file system

https://developer.nvidia.com/gpudirect

Hands-On Practice: Exercise Setup

Create space for the workshop materials, e.g.,

```
> mkdir /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop
```

Clone the `User_Codes` GitHub repository

```
> cd /n/netscratch/<LAB_NAME>/Lab/<USER_NAME>/Workshop/
> git clone https://github.com/fasrc/User_Codes.git
```

Go to the exercise directory

```
> cd User_Codes/Training/Distributed_GPU_Computing/
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing

Hands-On Practice: MPI + CUDA (single node)

Exercise 1: Hybrid Parallelism - MPI and CUDA on a single node

Problem: Compute π via numerical integration (midpoint rule)

- Domain is divided into bins (integration points) across MPI ranks
- Each MPI process handles a subset of bins (data decomposition)
- Each rank launches a CUDA kernel to compute partial sums on the GPU
- GPU threads evaluate: $\pi \approx \sum \frac{4}{1 + x^2} \Delta x$

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise1

Hands-On Practice: MPI + CUDA (single node)

Exercise 1: Hybrid Parallelism - MPI and CUDA on a single node

Parallel Workflow

- MPI layer (coarse-grain parallelism):
  - Distributes work across processes (ranks)
  - Computes partial π per rank
  - Uses `MPI_Allreduce` to combine results

- CUDA layer (fine-grain parallelism):
  - Each GPU executes many threads in parallel
  - Threads process bins in a strided / interleaved fashion
  - Partial sums reduced on host

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise1

Hands-On Practice: MPI + CUDA (single node)

Exercise 1: Hybrid Parallelism - MPI and CUDA on a single node

(1) Load required software modules

```
> module load gcc/12.2.0-fasrc01
> module load openmpi/5.0.5-fasrc02
```

(2) Compile the code

```
> cd Exercise1/
> nvcc -O3 -ccbin mpicxx -o  mpi_cuda_pi.x  mpi_cuda_pi.cu -lcudart -lcuda
```

or use the provided `Makefile`

```
> make
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise1

Hands-On Practice: MPI + CUDA (single node)

Exercise 1: Hybrid Parallelism - MPI and CUDA on a single node

(3) Prepare a batch-job submission script

```
#SBATCH -N 1
#SBATCH --ntasks-per-node=4
#SBATCH --gres=gpu:4
...
module load gcc/12.2.0-fasrc01
module load openmpi/5.0.5-fasrc02

srun -n $SLURM_NTASKS --mpi=pmix ./mpi_cuda_pi.x 10000
```

(4) Submit the job to the queue

```
> sbatch run_4.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise1

Hands-On Practice: MPI + CUDA (single node)

Exercise 1: Hybrid Parallelism - MPI and CUDA on a single node

(5) Explore the output

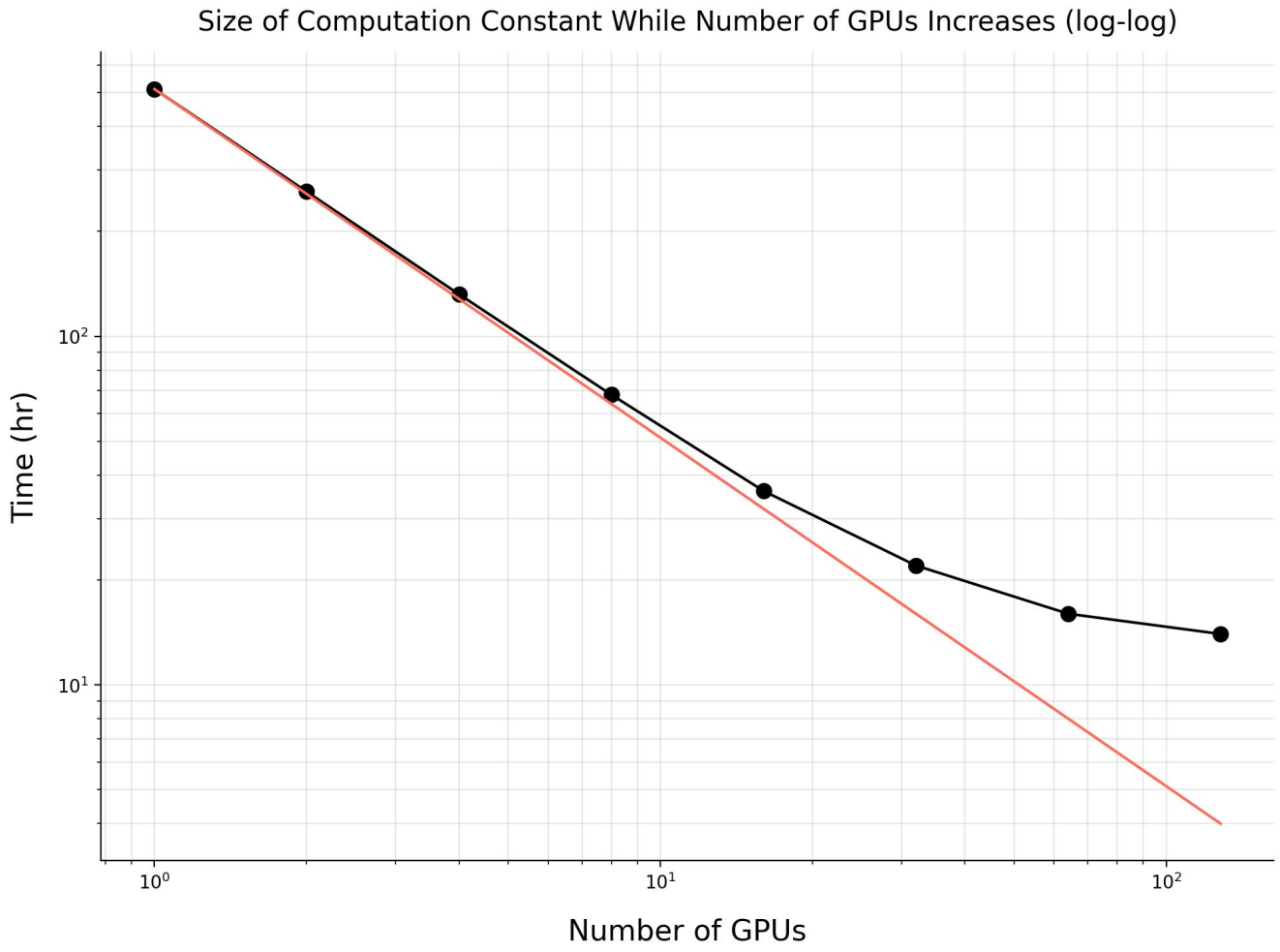
```
cat output.out
rank 0/4 | local_rank 0/4 | GPU 0 | bins = 2500 | blocks = 4 | threads/block = 768 | total threads = 3072 | partial pi = 0.979914653245634
rank 1/4 | local_rank 1/4 | GPU 1 | bins = 2500 | blocks = 4 | threads/block = 768 | total threads = 3072 | partial pi = 0.874675783824257
rank 2/4 | local_rank 2/4 | GPU 2 | bins = 2500 | blocks = 4 | threads/block = 768 | total threads = 3072 | partial pi = 0.719413999127246
rank 3/4 | local_rank 3/4 | GPU 3 | bins = 2500 | blocks = 4 | threads/block = 768 | total threads = 3072 | partial pi = 0.567588218225989

========== FINAL RESULTS ==========
Exact PI       = 3.14159265
Computed PI    = 3.14159265
Absolute error = 8.33333846e-10
Relative error = 2.65258402e-08 %
Total bins     = 10000
Wall time      = 0.000150 s
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise1

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

Strong Scaling



- Compute bound
- Keep problem size the same, increase number of GPUs
- At some point, communication overhead surpasses computation time

Strong Scaling

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

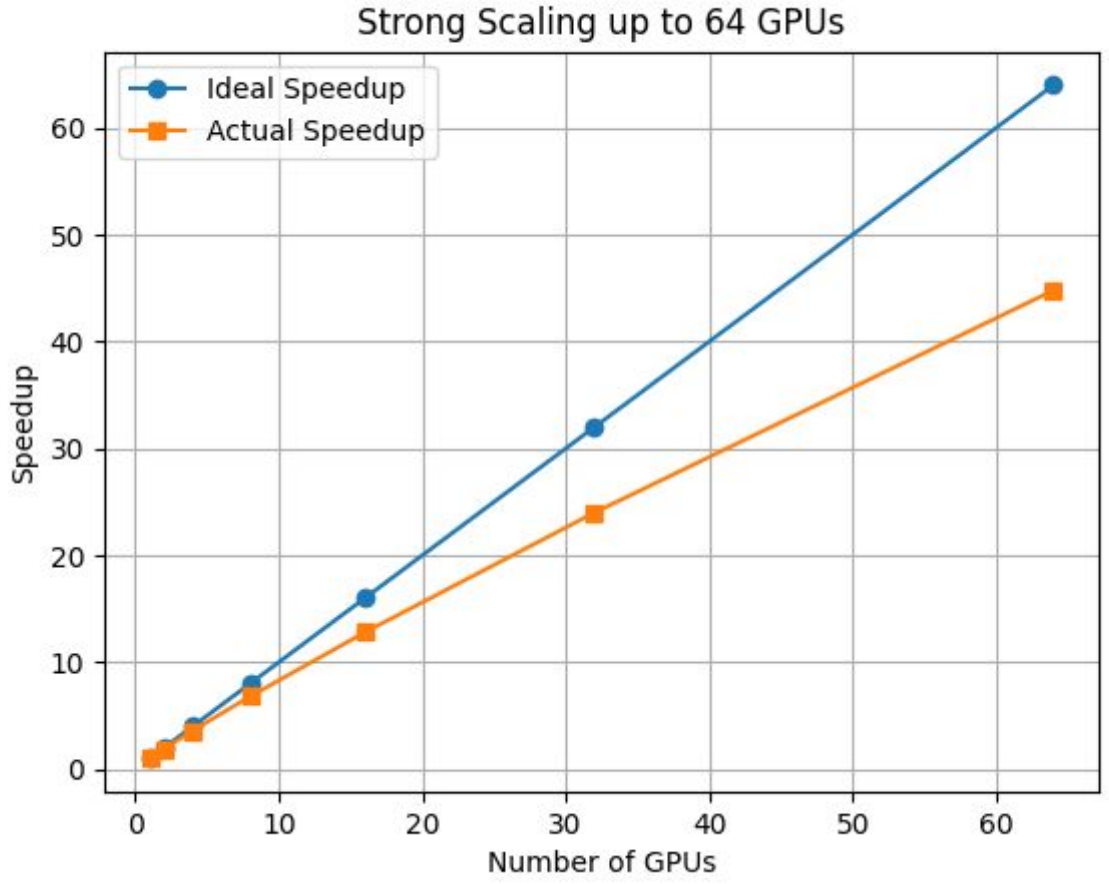
Time to complete the job on **one** process

Time to complete the job on **n** process

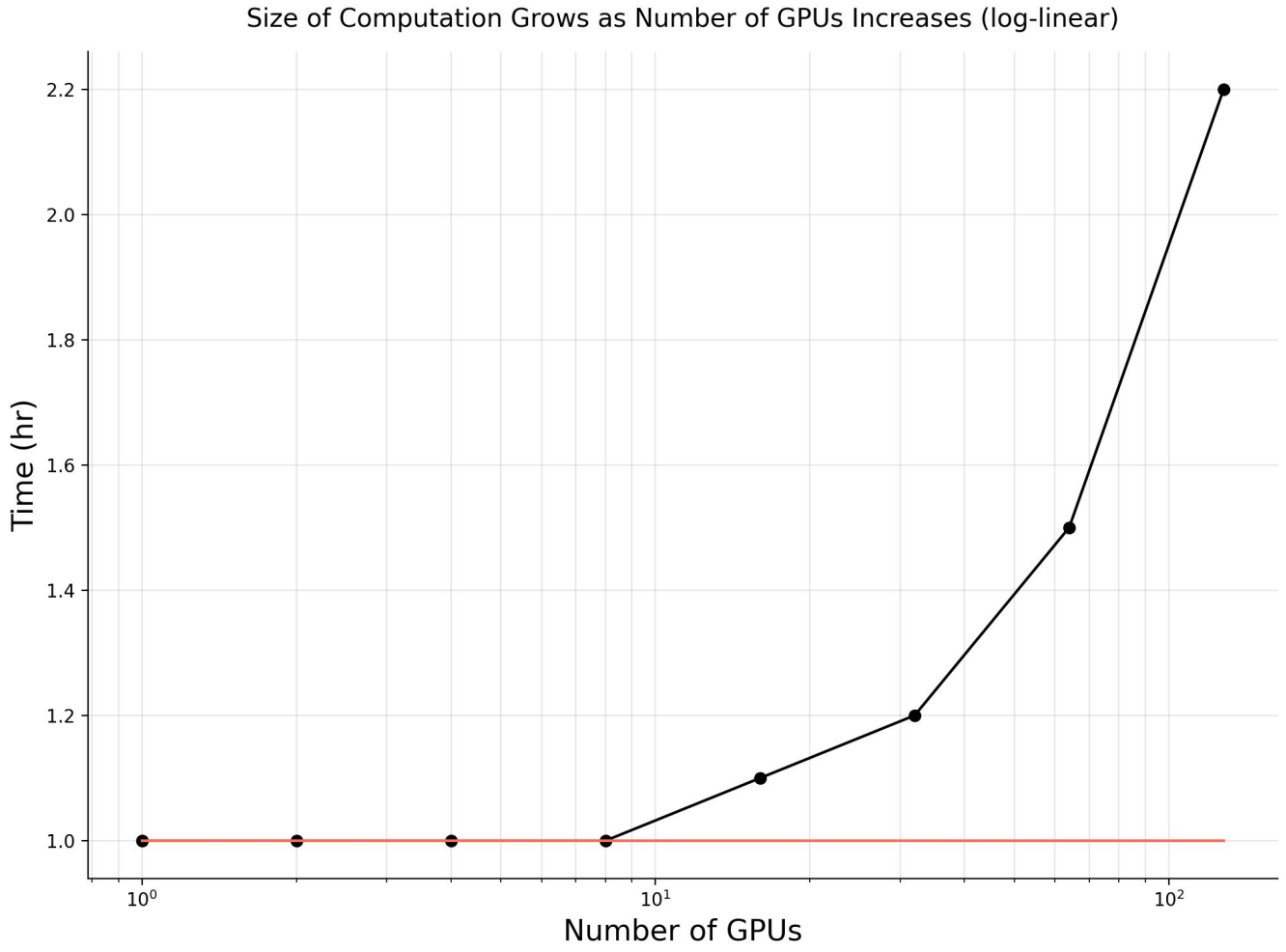
Efficiency:

$$E(n) = \frac{S(n)}{n}$$

Tells you how efficiently you parallelize your code



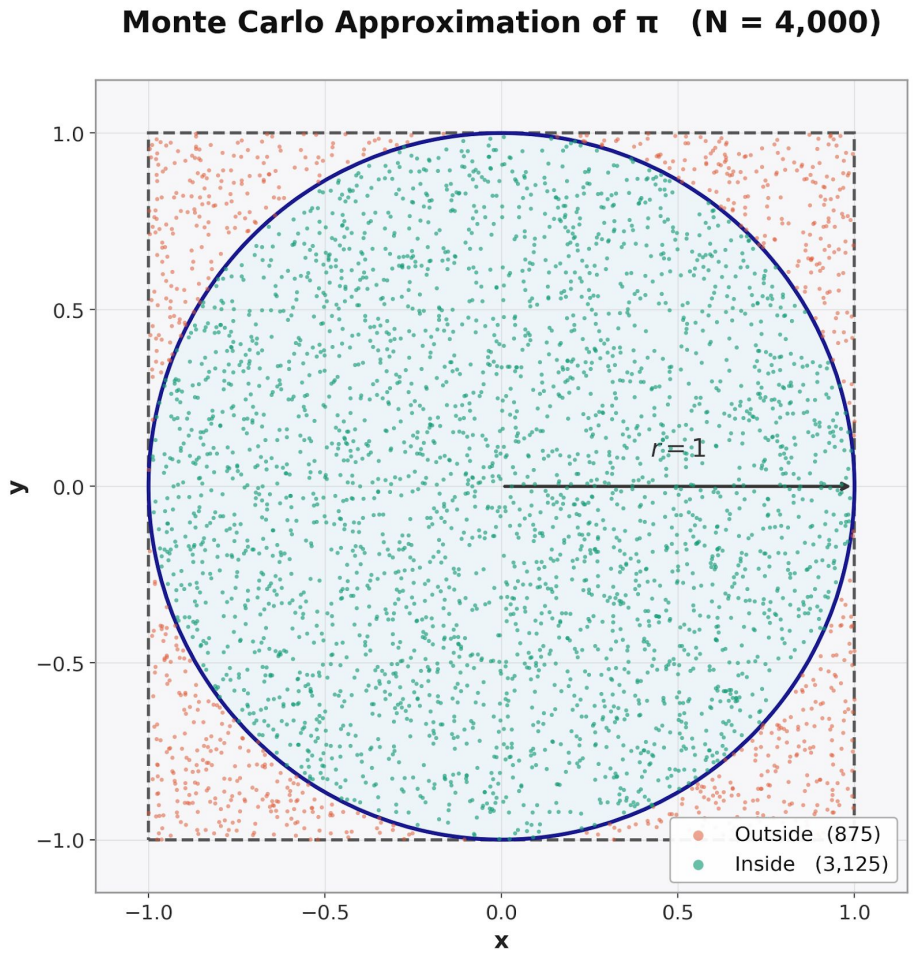
Weak Scaling



- Memory bound
- Increase problem size proportionally with resources
- At some point, communication overhead surpasses computation time

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π



Monte Carlo Estimation of π

Key idea

Randomly sample points in the unit square. Count how many fall inside the circle.

Geometric argument

$$\frac{A_{\text{circle}}}{A_{\text{square}}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

$$\Rightarrow \quad \hat{\pi} = 4 \times \frac{N_{\text{in}}}{N_{\text{total}}}$$

Results

N_{total} = 4,000

N_{in} = **3,125**

$\hat{\pi}$ = **3.12500**

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π

- Estimate π with Monte Carlo sampling instead of numerical integration
- Launch up to 4 MPI ranks on one node, mapping each rank to a local GPU
- Split samples across ranks; each GPU uses many CUDA threads to test random (x,y) points in parallel
- Reduce thread hit counts to one value per rank, then use `MPI_Reduce` to combine results on rank 0
- Measure runtime with MPI synchronization and report the maximum rank time as the wall time

Takeaway: Single-node multi-GPU scaling with MPI across ranks and CUDA within each GPU.

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise2

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π

(1) Load required software modules

```
> module load gcc/12.2.0-fasrc01
> module load openmpi/5.0.5-fasrc02
```

(2) Compile the code

```
> cd Exercise2/
> nvcc -O3 -ccbin mpicxx -o mpi_cuda_pi_mc.x mpi_cuda_pi_mc.cu -lcudart -lcuda
```

or use the provided `Makefile`

```
> make
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise2

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π

(3) Prepare a batch-job submissions script

```
#SBATCH -N 1
#SBATCH --ntasks-per-node=[1, 2, or 4]
#SBATCH --gres=gpu:[1, 2, or 4]
...
module load gcc/12.2.0-fasrc01
module load openmpi/5.0.5-fasrc02

srun -n $SLURM_NTASKS --mpi=pmix ./mpi_cuda_pi_mc.x 10000000000000 123
```

(4) Run the code with 1, 2, and 4 MPI tasks and 1 GPU/task

```
> sbatch run_1.sbatch
> sbatch run_2.sbatch
> sbatch run_4.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise2

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π

(3) Prepare a batch-job submissions script

```
#SBATCH -N 1
#SBATCH --ntasks-per-node=[1, 2, or 4]
#SBATCH --gres=gpu:[1, 2, or 4]
...
module load gcc/12.2.0-fasrc01
module load openmpi/5.0.5-fasrc02

srun -n $SLURM_NTASKS --mpi=pmix ./mpi_cuda_pi_mc.x 10000000000000 123
```

10 Trillion hits!!!

(4) Run the code with 1, 2, and 4 MPI tasks and 1 GPU/task

```
> sbatch run_1.sbatch
> sbatch run_2.sbatch
> sbatch run_4.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise2

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π

(4) Explore output and record wall time for each run

```
> cat output_4.out
...
rank 0/4 | local_rank 0/4 | GPU 0 | samples = 2500000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 1963494366269 | time = 11.409889 s
rank 1/4 | local_rank 1/4 | GPU 1 | samples = 2500000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 1963495905770 | time = 11.409886 s
rank 2/4 | local_rank 2/4 | GPU 2 | samples = 2500000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 1963494807727 | time = 11.409889 s
rank 3/4 | local_rank 3/4 | GPU 3 | samples = 2500000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 1963495651902 | time = 11.409895 s

========== FINAL RESULTS ==========
Exact PI       = 3.14159265
Estimated PI   = 3.14159229
Absolute error = 3.60922593e-07
Relative error = 1.14885230e-05 %
Total hits     = 7853980731668
Total samples  = 10000000000000
Wall time      = 11.409895 s
```

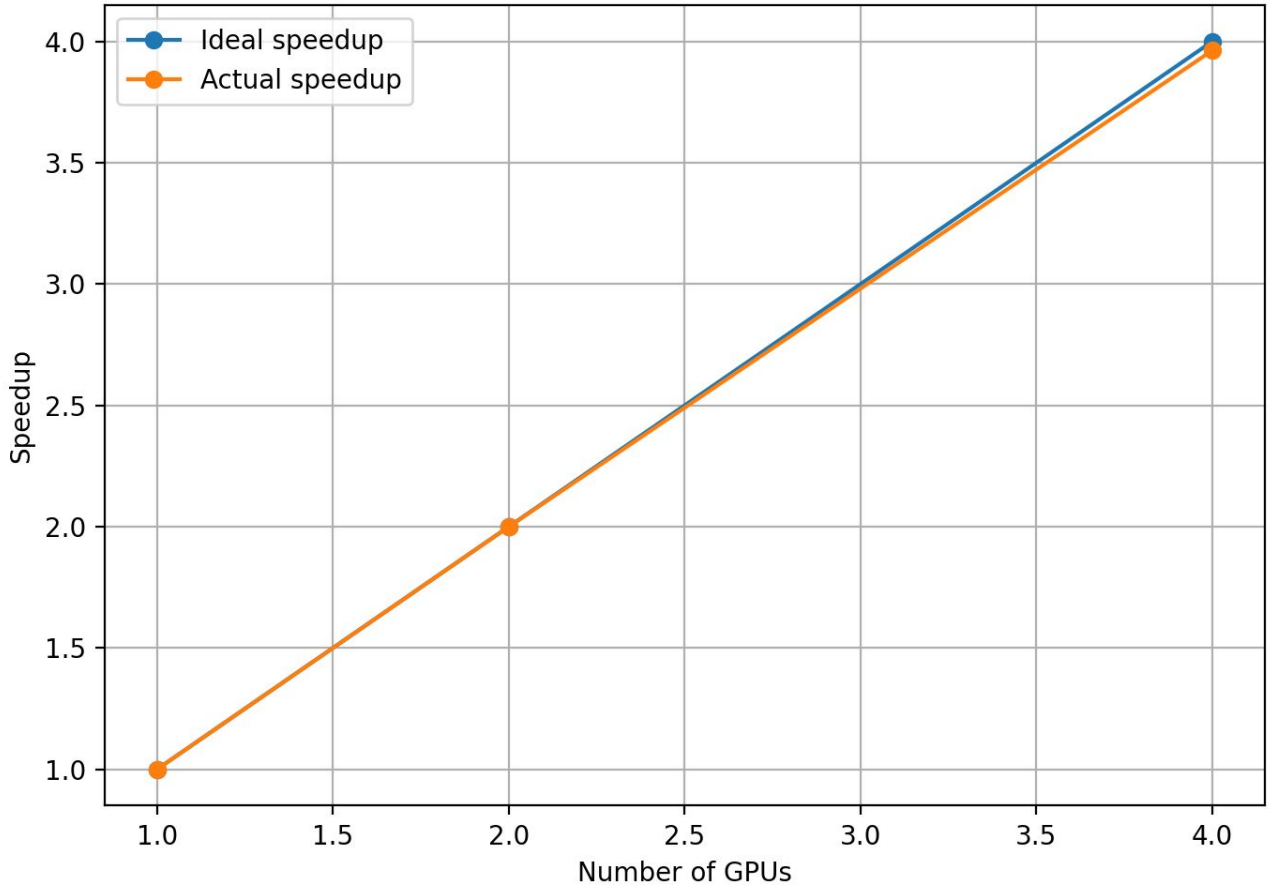
Use your favorite text editor (vi, emacs, nano, etc) to create a file `speedup.txt` the following content:

```
# N GPUs   Wall time
    1      45.212765
    2      22.608870
    4      11.409895
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise2

Hands-On Practice: Scaling (single node)

Exercise 2: Scaling with MPI and CUDA - Monte Carlo approximation of π



$$S(n) = \frac{T(1)}{T(n)} \rightarrow \text{speedup}$$

$$E(n) = \frac{S(n)}{n} \rightarrow \text{parallel efficiency}$$

GPUs	Wall time (s)	Speedup	Parallel efficiency
1	45.212765	1.0000	1.0000
2	22.608870	1.9996	0.9998
4	11.409895	3.9626	0.9906

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

Inter-GPU Communication - NCCL

NVIDIA Collective Communication Library (NCCL, pronounced "NICKEL") is used as backend in distributed strategies for NVIDIA GPUs

NCCL offers various collective communication primitives

- `ncclAllReduce`
- `ncclReduce`
- `ncclBroadcast`
- `ncclAllGather`
- `ncclGather`
- `ncclReduceScatter`

Why NCCL Instead of MPI?

Feature	MPI (CUDA-aware)	NCCL
Designed for	**CPUs (GPU support added)**	**GPUs natively**
NVLink aware	**❌ No**	**✅ Yes**
Topology optimization	**Limited**	**Full (NVLink, PCIe, IB)**
CUDA streams (async)	⚠️ Partial	✅ Native
AllReduce bandwidth	**~60-75% of peak**	**~90-95% of peak**
Small message latency	Better	Comparable
DL framework integration	**Manual**	**Automatic (e.g., PyTorch uses NCCL)**
Setup complexity	*Lower*	*Slightly higher*
Topology Sensitivity	*Lower*	*Very high*

Hands-On Practice: NCCL (single node)

Exercise 3: NCCL + CUDA Monte Carlo approximation of π

Problem: Estimate π using Monte Carlo sampling, with each GPU generating random (x,y) points in parallel

- Assign one GPU per rank/process, each computing a local hit count independently
- Use CUDA kernels to perform massive parallel sampling and local reductions on each GPU
- Use NCCL `AllReduce` to efficiently combine hit counts across all GPUs (GPU-to-GPU, no CPU staging)
- Compute final π on one GPU/host from the globally reduced hit count

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise3

Hands-On Practice: NCCL (single node)

Exercise 3: NCCL + CUDA Monte Carlo approximation of π

(1) Load required software modules

```
> module load nvhpc/24.11-fasrc01 # NVIDIA HPC SDK (provides NCCL and CUDA)
```

(2) Compile the code

```
> cd Exercise3/
> nvcc -o nccl_pi_mc.x nccl_pi_mc.cu -lnccl
```

or use the provided `Makefile`

```
> make
```

(3) Submit the job to the queue

```
> sbatch run.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise3

Hands-On Practice: NCCL (single node)

Exercise 3: NCCL + CUDA Monte Carlo approximation of π

(4) Explore the output

```
> cat output.out
NCCL version 2.18.5+cuda12.2
GPU 0/4 | samples=2500000000 | local hits=1963486240
GPU 1/4 | samples=2500000000 | local hits=1963515235
GPU 2/4 | samples=2500000000 | local hits=1963455274
GPU 3/4 | samples=2500000000 | local hits=1963486136
------------------------------------------------
Exact PI       = 3.14159265
Estimated PI   = 3.14157715
Absolute error = 0.00001550
Relative error = 0.00049337 %
Total hits     = 7853942885
Total samples  = 10000000000
Total time     = 28.578 ms
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise3

Hands-On Practice: NCCL (multi-node)

Exercise 4: Scaling with NCCL and CUDA - Monte Carlo approximation of π

Problem: Estimate π using Monte Carlo sampling, with each GPU computing local hit counts in parallel

- **Use MPI only for bootstrapping:** rank assignment, exchanging `ncclUniqueId`, and initializing communicators
- Initialize NCCL communicators via MPI, then rely on NCCL for all GPU communication
- Perform GPU-local computation with CUDA kernels (random sampling + hit counting)
- Use NCCL `AllReduce` for GPU-to-GPU reduction, avoiding CPU staging and maximizing bandwidth

MPI is used only for setup/orchestration, while NCCL handles all high-performance GPU communication and CUDA performs the compute.

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise4

Hands-On Practice: NCCL (multi-node)

Exercise 4: Scaling with NCCL and CUDA - Monte Carlo approximation of π

(1) Load required software modules

```
> module load nvhpc/24.11-fasrc01  gcc/12.2.0-fasrc01 openmpi/4.1.5-fasrc03
```

(2) Compile the code

```
> cd Exercise4/
> nvcc -O3 -std=c++17 -o nccl_mpi_pi_mc.x nccl_mpi_pi_mc.cu -lnccl -lmpi
```

or use the provided `Makefile`

```
> make
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise4

Hands-On Practice: NCCL (multi-node)

Exercise 4: Scaling with NCCL and CUDA - Monte Carlo approximation of π

(3) Prepare a batch-job submission script, e.g., `run_8.sbatch`; 8 GPUs on 2 nodes with 4 GPUs/node

```
#SBATCH -N 2
#SBATCH --ntasks-per-node=4
#SBATCH --gpus-per-node=4
...
module load nvhpc/24.11-fasrc01                          # NCCL and CUDA
module load gcc/12.2.0-fasrc01 openmpi/4.1.5-fasrc03 # MPI

srun -n 8 --mpi=pmix ./nccl_mpi_pi_mc.x 10000000000000
```

(4) Submit the job to the queue (1, 2, 4 and 8 GPUs), e.g.,

```
> sbatch run_8.sbatch
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise4

Hands-On Practice: NCCL (multi-node)

Exercise 4: Scaling with NCCL and CUDA - Monte Carlo approximation of π

(5) Explore output and record wall time for each run (1, 2, 4, and 8 GPUs), e.g.,

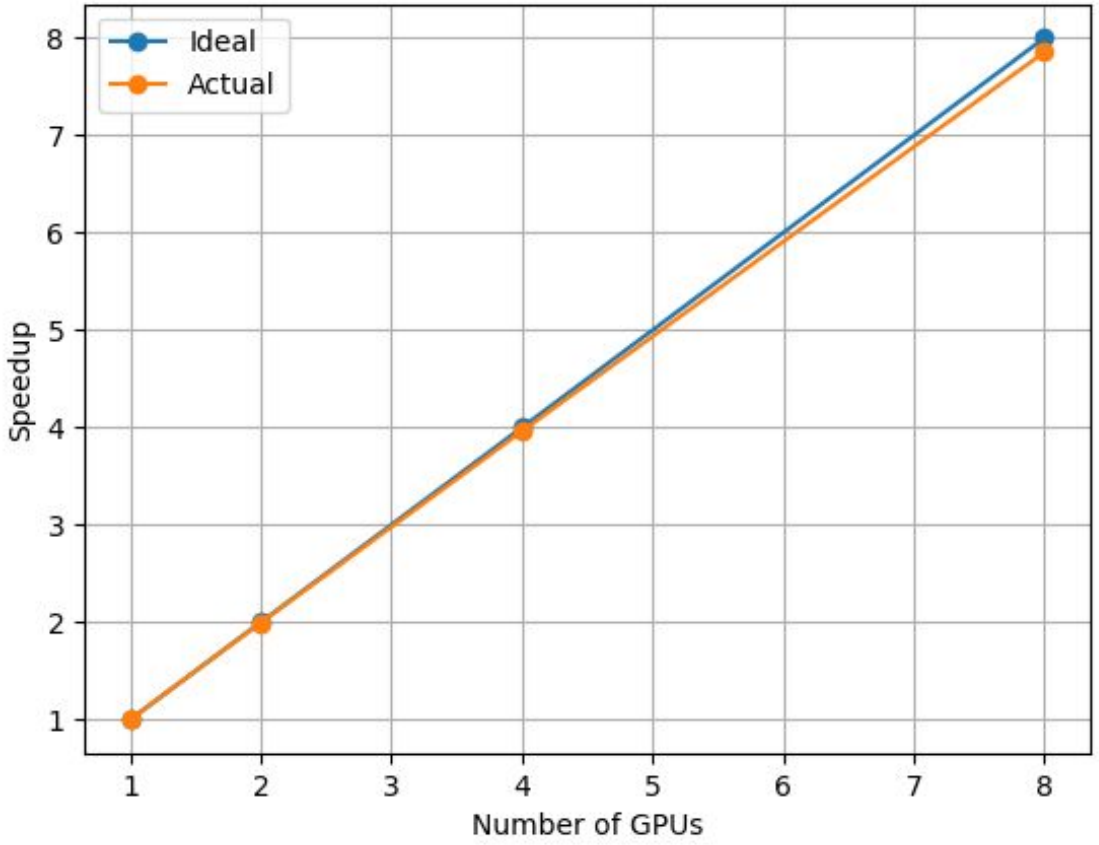
```
> cat output_8.out
rank 0/8 | local_rank 0/4 | GPU 0 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981747540162 | time = 5.748093 s
rank 1/8 | local_rank 1/4 | GPU 1 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981747959763 | time = 5.748150 s
rank 2/8 | local_rank 2/4 | GPU 2 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981748218555 | time = 5.748102 s
rank 3/8 | local_rank 3/4 | GPU 3 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981747733647 | time = 5.748149 s
rank 4/8 | local_rank 0/4 | GPU 0 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981748012275 | time = 5.748090 s
rank 5/8 | local_rank 1/4 | GPU 1 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981747129961 | time = 5.748145 s
rank 6/8 | local_rank 2/4 | GPU 2 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981748096166 | time = 5.748099 s
rank 7/8 | local_rank 3/4 | GPU 3 | samples = 1250000000000 | blocks = 3456 | threads/block = 1024 | total threads = 3538944 | hits = 981747197711 | time = 5.748145 s

========== FINAL RESULTS ==========
Exact PI       = 3.14159265
Estimated PI   = 3.14159276
Absolute error = 1.01706207e-07
Relative error = 3.23740912e-06 %
Total hits     = 7853981888240
Total samples  = 10000000000000
Wall time      = 5.748150 s
```

(6) Calculate speedup and parallel efficiency

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise4

Hands-On Practice: NCCL (multi-node)

Exercise 4: Scaling with NCCL and CUDA - Monte Carlo approximation of π



$$S(n) = \frac{T(1)}{T(n)} \rightarrow \text{speedup}$$

$$E(n) = \frac{S(n)}{n} \rightarrow \text{parallel efficiency}$$

GPUs	Wall time (s)	Speedup	Parallel efficiency
1	45.151563	1.0000	1.0000
2	22.624692	1.9960	0.9980
4	11.396042	3.9637	0.9909
8	5.748150	7.8573	0.9822

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise4

NCCL Bootstrapping Options

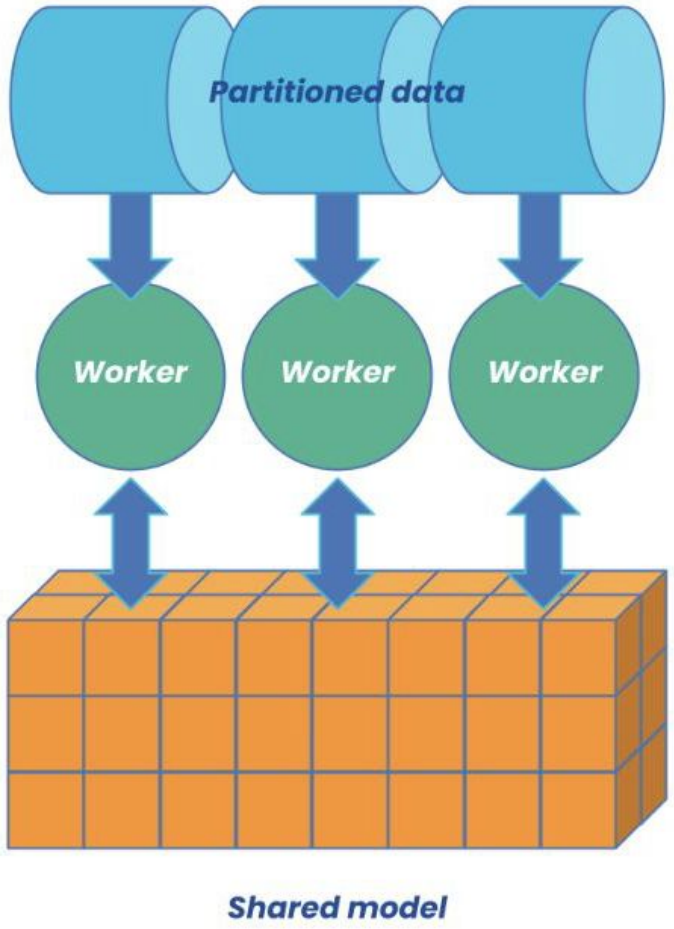
- MPI (most common): broadcast ncclUniqueId via MPI
- Shared file: write/read ID from shared filesystem (NFS, Lustre)
- PMIx / scheduler: exchange ID via Slurm / PMI interface
- Sockets (TCP): custom exchange of ID between ranks
- Frameworks (PyTorch): handled automatically via env variables

https://github.com/fasrc/User_Codes/tree/master/GPU_Computing/Dist_GPU_Computing/NCCL_and_CUDA/MultiNode

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

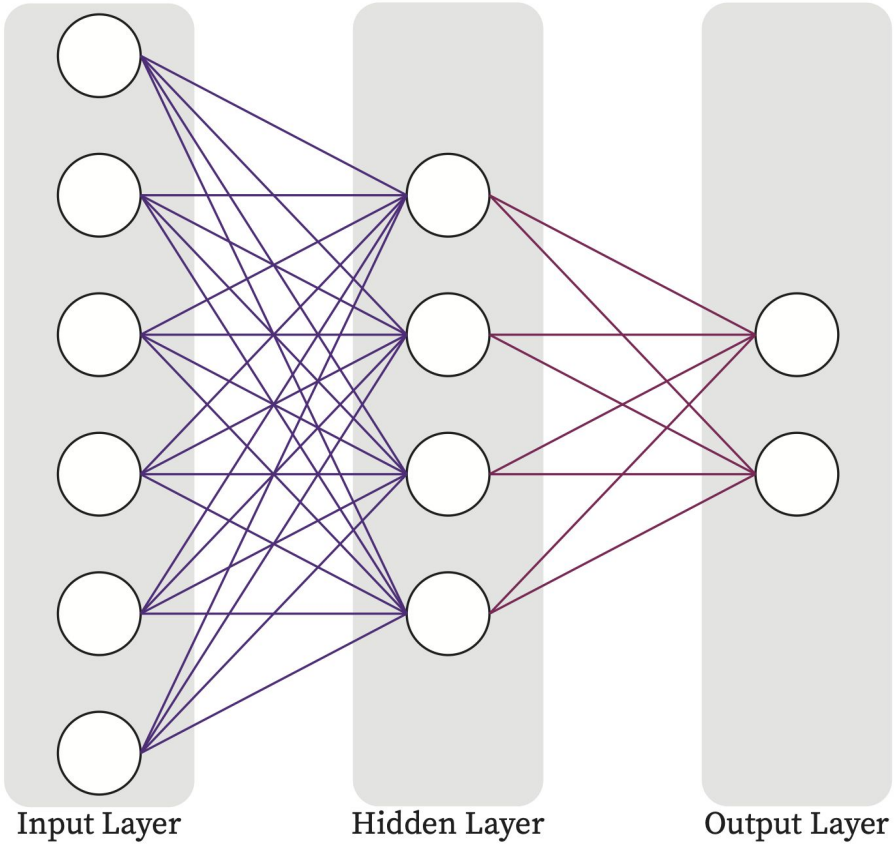
Distributed Data Parallel (DDP) Processing



- Most common approach to distributed training in machine learning
- Each GPU trains a copy of the model
- Dataset is split into different batches of data on each GPU

Figure from: https://www.anyscale.com/blog/what-is-distributed-training

Multi-Layer Perceptron (MLP)



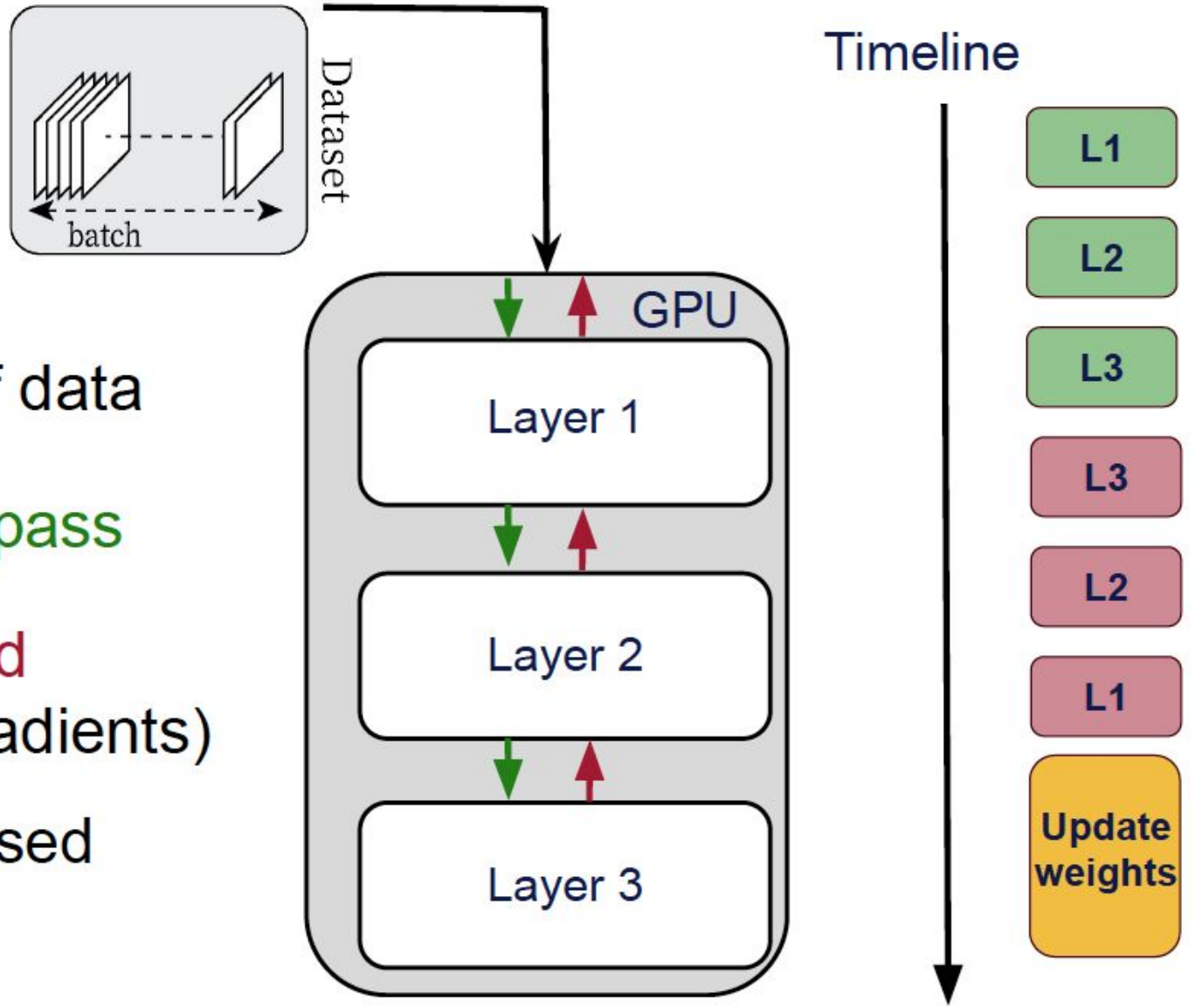
$$x_i = [x_1\ x_2\ x_3\ x_4\ x_5\ x_6] \qquad W_1 = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_5 & w_6 & w_7 & w_8 \\ w_9 & w_{10} & w_{11} & w_{12} \\ w_{13} & w_{14} & w_{15} & w_{16} \\ w_{17} & w_{18} & w_{19} & w_{20} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \qquad W_2 = \begin{bmatrix} z_1 & z_2 \\ z_3 & z_4 \\ z_5 & z_6 \\ z_7 & z_8 \end{bmatrix}$$

$$y_i = [y_1\ y_2] \qquad b_1 = [b_1\ b_2\ b_3\ b_4] \qquad b_2 = [c_1\ c_2]$$

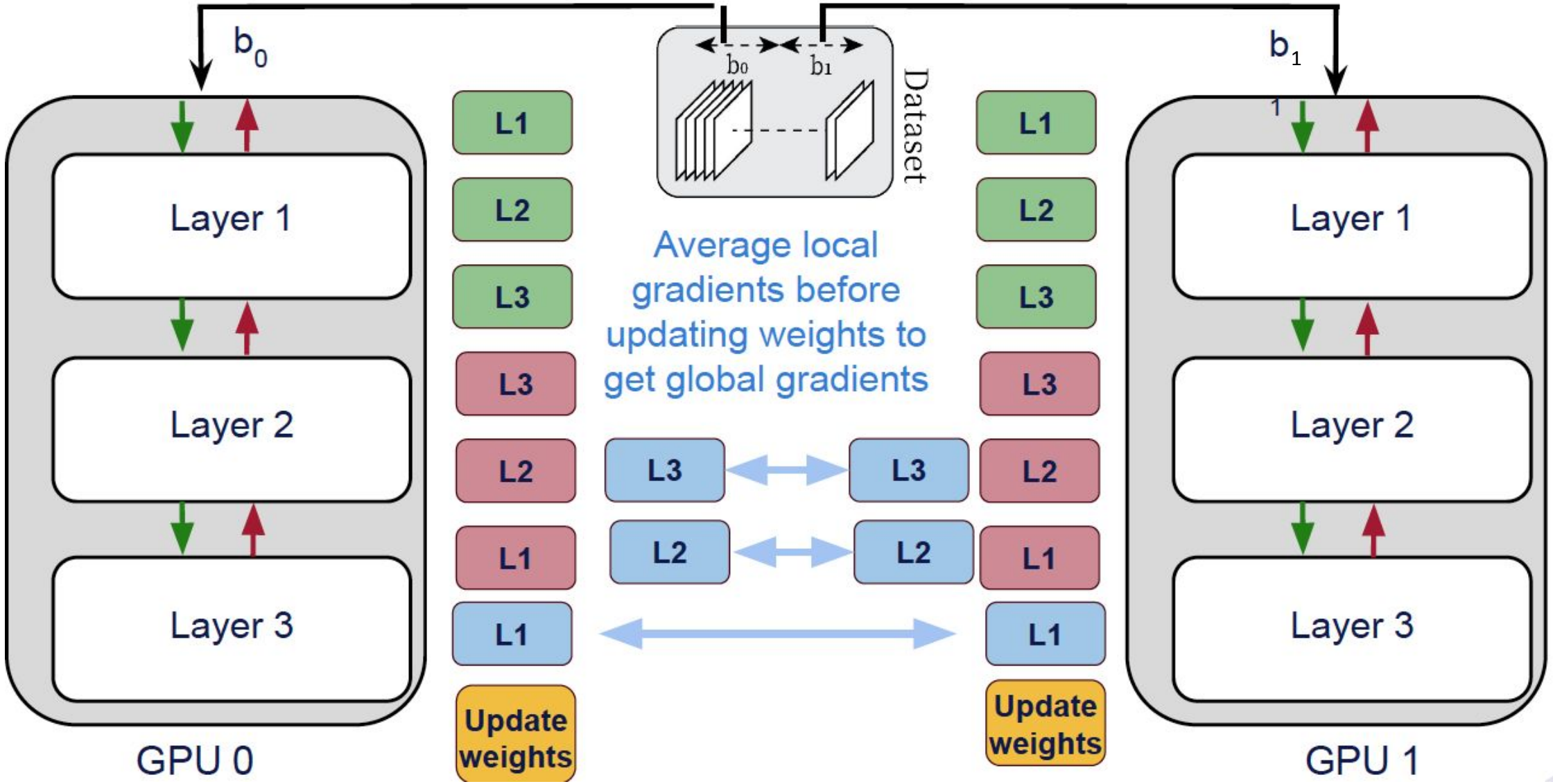
https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/distributed_gpu_computing.html

Single GPU MLP Training

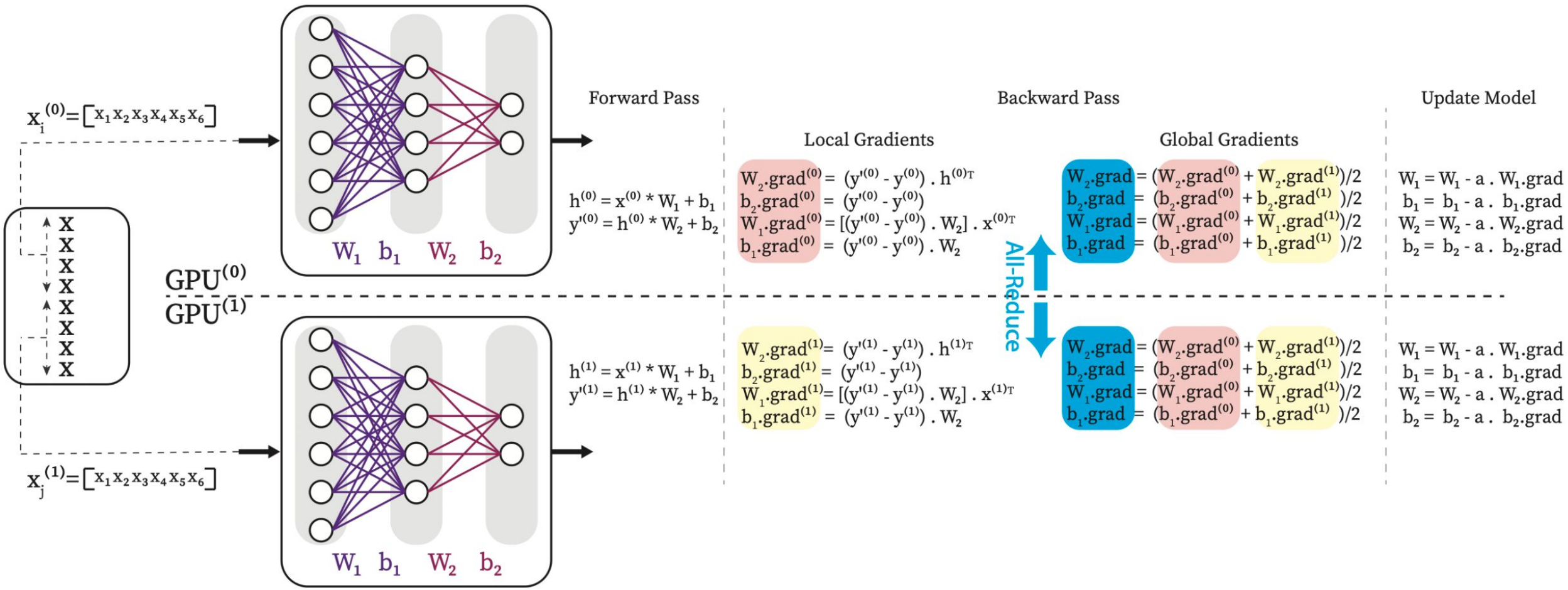
1) Model gets batch of data
2) Computes forward pass
3) Computes backward pass (computing gradients)
4) Updates weights based on gradients



https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/distributed_gpu_computing.html



https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/distributed_gpu_computing.html



https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/distributed_gpu_computing.html

PyTorch Conda Environment

Workshop PyTorch Conda Environment

```
# Load a Python module and activate the conda environment
module load python
conda activate /n/netscratch/rc_admin/Everyone/dist-gpu-training/pt2.9.1_cuda12.9
```

Local PyTorch Conda Environment with GPU support

```
# Start an interactive session
salloc -p gpu -t 0-06:00 --mem=8000 --gres=gpu:1

# Load a Python module
module load python

# Create a conda environment
mamba create -n pt2.9.1_cuda12.9 pip wheel

# Activate the conda environment
conda activate pt2.9.1_cuda12.9

# Install PyTorch with GPU support
pip3 install torch torchvision --index-url  https://download.pytorch.org/whl/cu129
```

https://docs.rc.fas.harvard.edu/kb/pytorch/

Hands-On Practice: PyTorch DDP

Exercise 5: PyTorch Multi-GPU MLP with DDP

- **The idea:** each GPU holds a full copy of the model; gradients are averaged across GPUs every step, so all ranks stay in sync
- **The example:** tiny MLP (6 → 4 → 2) on 1024 synthetic samples, one process per GPU, across one or more nodes
- **Core DDP calls:** `init_process_group`, wrap model with `DDP(...)`, use `DistributedSampler` to partition data, `destroy_process_group` at the end
- **Launch:** SLURM runs one task per GPU; rank comes from `SLURM_PROCID`, rendezvous via `MASTER_ADDR` / `MASTER_PORT`
- **Heads-up:** this demonstrates the mechanics of DDP, not speedup - the model is too small for communication cost to pay off

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise5

Hands-On Practice: PyTorch DDP

Exercise 5: PyTorch Multi-GPU MLP with DDP

Run the code

```
> cd Exercise5/
> sbatch run.sbatch mlp_ddp.py
```

Explore the output

```
> cat mlp_multi_gpu_13692.out
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise5

DDP template (PyTorch)

Key points

`rank` = global ID, `local_rank` = GPU index on this node - not the same on multi-node

Call `torch.cuda.set_device(local_rank)` *before* `init_process_group`

Use `DistributedSampler` so each rank trains on a different slice of the data

```python
import os, torch
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup():
    rank       = int(os.environ["RANK"])
    local_rank = int(os.environ["LOCAL_RANK"])
    world_size = int(os.environ["WORLD_SIZE"])

    torch.cuda.set_device(local_rank)
    dist.init_process_group(
        backend="nccl",
        rank=rank, world_size=world_size,
        device_id=torch.device(f"cuda:{local_rank}"),
    )
    return rank, local_rank, world_size

def cleanup():
    dist.destroy_process_group()

# --- usage ---
rank, local_rank, world_size = setup()
model = DDP(MyModel().to(local_rank), device_ids=[local_rank])
# ... training loop ...
cleanup()
```

Launch PyTorch DDP

```
...
#SBATCH --nodes=2                    # total nodes
#SBATCH --ntasks-per-node=4          # 1 task per GPU
#SBATCH --gpus-per-node=4            # GPUs per node
#SBATCH --cpus-per-task=8            # CPUs per rank (for DataLoader)
...
# Rendezvous
MASTER_HOST=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n1)
export MASTER_ADDR=$(getent ahostsv4 "$MASTER_HOST" | awk '{print $1; exit}')
export MASTER_PORT=$((29500 + SLURM_JOB_ID % 20000))
export WORLD_SIZE=$SLURM_NTASKS

srun --cpu-bind=cores python -u train.py
```

Key points

- World size = `nodes x ntasks-per-node` (here: 2 x 4 = 8 GPUs)
- One SLURM task per GPU; `SLURM_PROCID` → `RANK`, `SLURM_LOCALID` → `LOCAL_RANK` (set by `srun`)
- Resolve `MASTER_ADDR` to IPv4 to avoid IPv6 warnings; unique `MASTER_PORT` per job avoids collisions

Hands-On Practice: PyTorch DDP

Exercise 6: PyTorch Multi-GPU MNIST CNN with DDP

- **The idea:** same DDP pattern as before, but on a real dataset and a real CNN - each GPU trains on its own slice of MNIST, gradients are averaged every step
- **The example:** small CNN (2 conv layers + 2 FC) on 60k MNIST images, 10 epochs, scales across multiple GPUs and nodes
- **What's new vs. Exercise 5:** real training loop with validation, train/val accuracy reduced across ranks, `DistributedSampler` with `set_epoch()` for proper shuffling, learning rate scaled linearly with world size
- **Launch (FASRC):** one SLURM task per GPU; `MASTER_ADDR` resolved to IPv4 to avoid IPv6 warnings; `MASTER_PORT` derived from `SLURM_JOB_ID` to prevent collisions
- **Result:** ~99% validation accuracy in 10 epochs on 8 GPUs × 2 nodes DDP actually pays off here, unlike the toy MLP

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise6

Hands-On Practice: PyTorch DDP

Exercise 6: PyTorch Multi-GPU MNIST CNN with DDP

Run the code

```
> cd Exercise6/
> sbatch run.sbatch train_mnist_ddp.py
```

Explore the output, e.g.,

```
> cat  mnist_ddp_13694.out
```

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise6

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A

Modern Distributed Training for Large Models

Data Parallelism (DDP) — replicate the model on each GPU, split the data, average gradients every step. Simple and what we've been doing. Breaks down when the model itself doesn't fit on one GPU.

Tensor Parallelism (TP) — split individual layers across GPUs (e.g. shard a large matmul column-wise across 4 devices). Needed when a single layer is too big for one GPU. High communication cost; usually confined to within a node.

Pipeline Parallelism (PP) — split the model by layer across GPUs: GPU 0 runs layers 1–10, GPU 1 runs 11–20, etc. Activations flow forward; gradients flow backward. Requires microbatching to keep all stages busy ("bubble" otherwise).

Fully Sharded Data Parallelism (FSDP / ZeRO) — like DDP, but shard parameters, gradients, and optimizer states across ranks and gather them on demand. Massive memory savings, moderate extra communication. The default choice for models that are "big but not enormous."

3D Parallelism — combine DP + TP + PP. Used for frontier-scale training (hundreds of billions of parameters, thousands of GPUs). TP within a node, PP across nodes, DP on top.

https://github.com/fasrc/User_Codes/tree/master/Training/Distributed_GPU_Computing/Exercise6

Modern Distributed Training for Large Models

Rule of thumb

Model fits on 1 GPU → DDP

Model doesn't fit, but activations do → FSDP

Single layer doesn't fit → add TP

Model so deep it doesn't fit even with TP → add PP

Outline

- Introduction & Background
- MPI + CUDA
- Scaling on a Single Node (4 GPUs)
- NCCL: Single & Multi-Node
- PyTorch DDP
- Modern Approaches
- Tips and Q&A ←

Tips & Best Practices

- Start single-GPU, then scale. Get the loop correct on one GPU before adding DDP.
- Know your algorithm's limits. More GPUs ≠ better answer. Monte Carlo π converges as $1/\sqrt{N}$; going from 1M to 16M samples only halves the error, no matter how many GPUs you throw at it.
- Know your ranks. `rank` = global ID, `local_rank` = GPU on this node. Use `local_rank` for `set_device`, `rank` for data.
- Validate small first. Debug on 2 ranks before launching 64. Most bugs surface at 2 and cost 32 x less to fix.
- DDP isn't always faster. For small models, communication dominates. Benchmark against single-GPU before assuming speedup.

FASRC documentation

- FASRC docs: FASRC DOCS
- Training
  - Calendar: Training Calendar | FAS Research Computing
  - Material: Training Materials – FASRC DOCS
- Getting help
  - Office hours: Virtual Office Hours | FAS Research Computing
  - Ticket:
    - HUIT Service Portal -> Submit Ticket: Submit Ticket - IT Help
    - Email: rchelp@rc.fas.harvard.edu

Upcoming Trainings

Training calendar: [Training Calendar | FAS Research Computing]

FASRC: AI-ready Data

Training focused on what is AI-ready data, how to obtain it, and some of the tools available on FASRC clusters to achieve that.

Audience: FASRC Users (all levels)

Details:
https://www.rc.fas.harvard.edu/events/ai-ready-data/

Training session evaluation

Please, fill out our training session evaluation. Your feedback is essential for us to improve our trainings!!

https://tinyurl.com/5n777e7f

Questions, Comments, Concerns?