

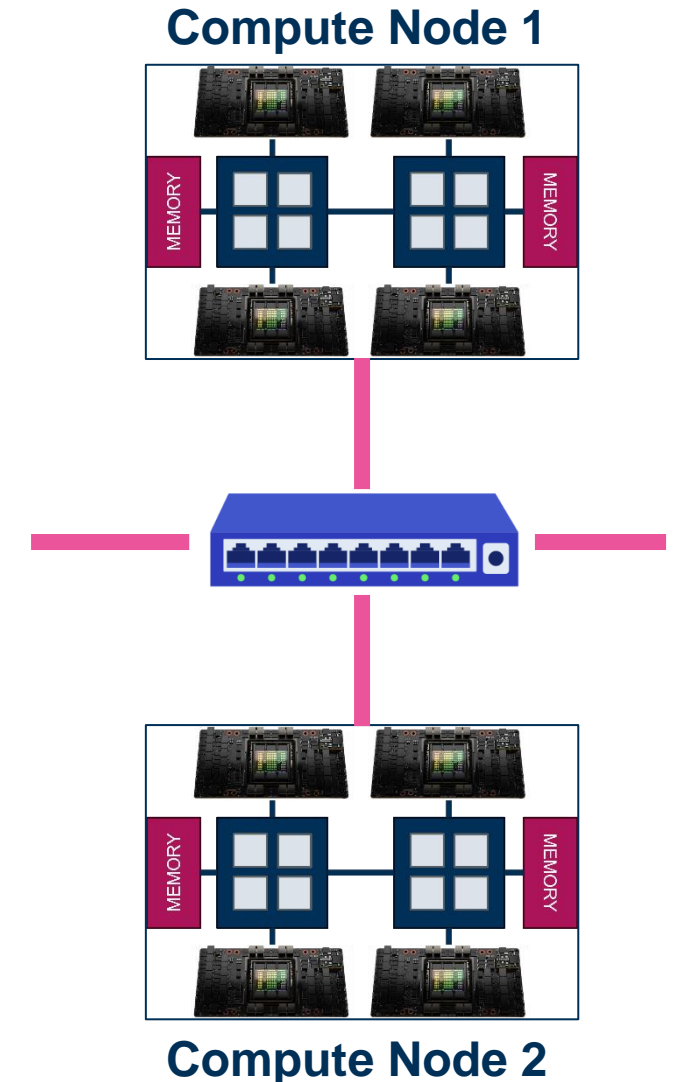


PPCES: Machine and Deep Learning

Distributed Deep Learning + Case Study – PyTorch



- **Why do we need distributed deep learning?**
 - One compute node or GPU is not enough
 - Limited host memory → dataset too large for single compute node
 - Limited GPU memory → model too large to fit into GPU memory
 - Speedup training (or inference)
 - Single GPU could run workload
 - But: Multiple GPUs can complete workload much faster
- **Challenges**
 - Communication necessary (data, parameters, weights)
 - Common backends: MPI, NCCL, Gloo (file-based)
 - Model consistency (centralized vs decentralized)
 - Learning rate scaling might be required



– Three most common strategies



Data Parallelism

- Split data in micro-batches
- Easy to implement
- Best strong scalability
- Does not improve memory consumption

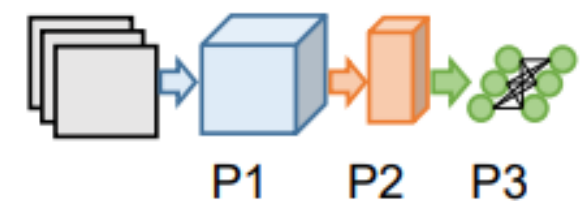
➔ **Faster training**



Model Parallelism

- Split layers in partitions
- Necessary for large layers
- Can improve load balancing
- Hard and time consuming to implement

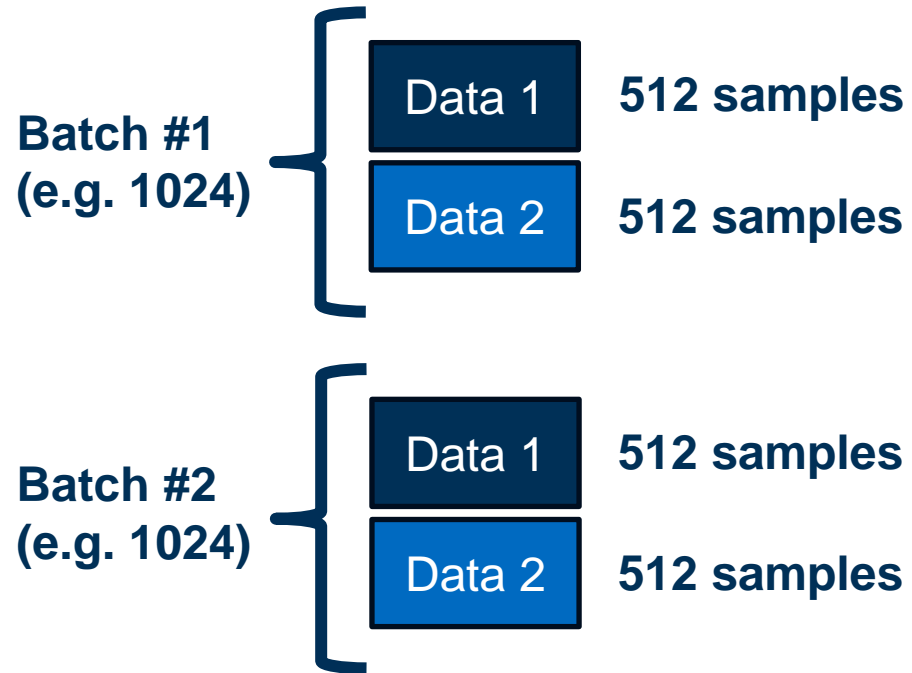
➔ **Bigger models**



Layer Pipelining

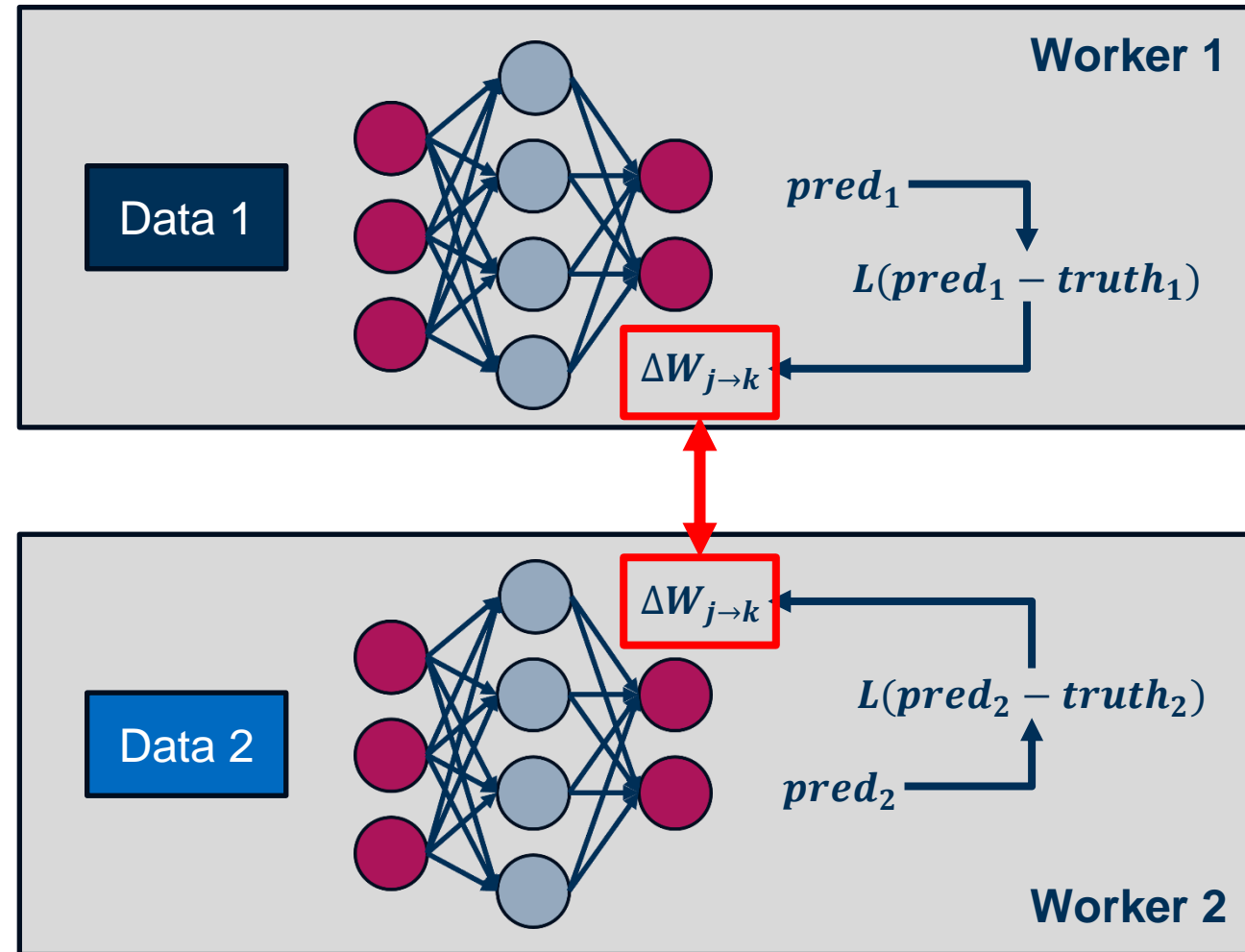
- Split DNN in partitions
- Improves memory consumption
- Hard to achieve good load balance
- Difficult to implement

Example: Data Parallelism with 2 GPU Workers



– **Note:**

- Batch size limits parallelism
- Scaling batch size requires scaling of learning rate (linearly)



– Native support in frameworks

– TensorFlow: `tf.distribute.Strategy` ([Documentation](#))

– e.g., `MirroredStrategy`, `MultiWorkerMirroredStrategy`

– PyTorch: `torch.distributed` ([Documentation](#))

– Especially: `torch.nn.parallel.DistributedDataParallel` (DDP)

– Additional software such as Horovod

– Developed by Uber Engineering ([Documentation](#))

– Aiming at large scale workloads (MPI backend)

– With only little code modification supports

– TensorFlow

– PyTorch

– MXNet



- **Step 1:** Spawn multiple processes
 - Represent workers (usually 1 per GPU)
 - Can be spread across multiple compute nodes
 - Examples: MPI, `torchrun`, ...
- **Step 2:** Initialize and destroy process group
 - Processes need to communicate and exchange data
 - Requires: $\#_{procs}$, ID_{proc}
 - Built-in `torch.distributed` package
 - Available backends: NCCL, MPI, Gloo, ...
 - NCCL: Need to set `MASTER_ADDR` and `MASTER_PORT`

```
# execute with MPI/srun
srun ... python train_model.py <args>

# execute with MPI/srun
srun ... zsh -c '\
    source set_vars.sh && \
    aptainer exec -e --nv ${PYTORCH_IMAGE} \
    bash -c "python train_model.py <args>"'
```

```
# initialize process group
torch.distributed.init_process_group(
    backend='nccl', init_method="env://",
    world_size=args.world_size,
    rank=args.world_rank
)

# wait for completion
torch.distributed.barrier()
```

- **Step 3:** Wrap your model
 - Use `DistributedDataParallel` (DDP)
 - Model parameters will be synchronized at creation
 - Specify the desired device
- **Step 4:** Distribute data across workers
 - Each worker will get a split of the batches
 - Idea: Extend loader with `DistributedSampler`
 - Number of Replicas
 - Current Proc ID
 - Shuffle or not?

```
from torch.nn.parallel import
DistributedDataParallel

# wrap your model and specify device
model = DistributedDataParallel(
    model,
    device_ids=[args.local_rank]
)
```

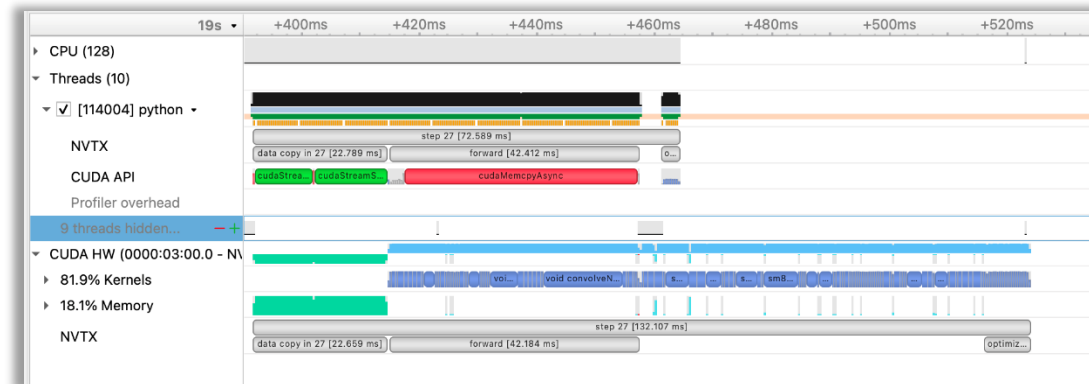
```
from torch.utils.data import
DistributedSampler, DataLoader

# create sampler
sampler = DistributedSampler(
    dataset, shuffle=True,
    num_replicas=args.world_size,
    rank=args.world_rank
)

# create data loader that is using the sampler
loader = DataLoader(dataset, sampler=sampler)
```

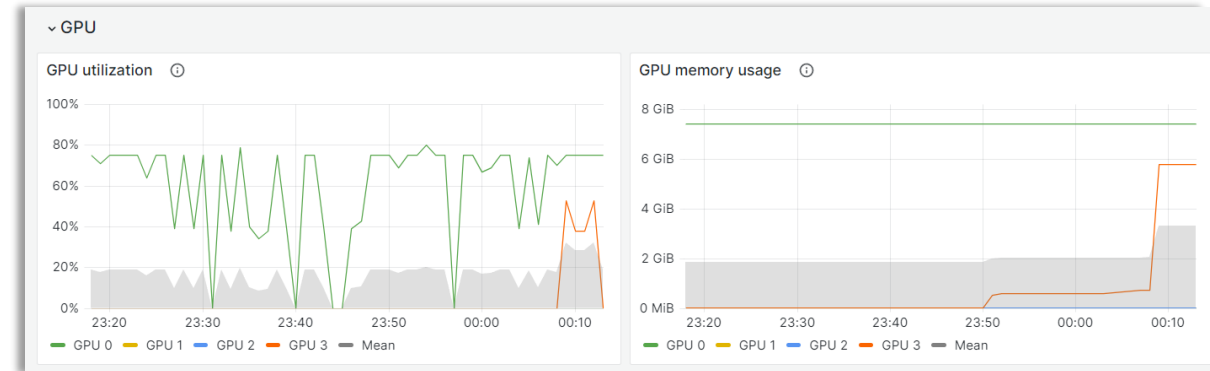
Identifying Resource Utilization and Efficiency

- **Question:** How do I know how much resources are used?
 - Compute vs. Memory
 - Does it make sense to scale to multiple GPU or even multiple nodes?
- **Various tools available**
 - Regular profiling / tracing tools
 - Score-P, CUBE, Vampire (Link)
 - Create detailed profiles and traces of your application
 - **TensorBoard (Link) / NVIDIA Nsight Systems (Link)**
 - Specifically designed for DL frameworks
 - Identify issues and receive recommendations
 - **Easier: monitoring (e.g., with `nvidia-smi`)**



- **Easy Option 1:** Review our continuous monitoring (Grafana)

- We collect several metrics every minute
- <https://perfmon.hpc.itc.rwth-aachen.de/>
- View monitoring data for your current / previous jobs
 - CPU / Memory / GPU / Power / Network



- **Easy Option 2:** NVIDIA System Management Interface (`nvidia-smi`)

- Provides monitoring and management capabilities
- Information about
 - Configuration (e.g., compute mode)
 - Clock speed, power draw & temperature
 - Utilization (GPU, memory, encoder / decoder)
- Output can be configured and written to csv or stdout
 - e.g., filter what you want to see and in what interval

Basic Resource Utilization Checks with `nvidia-smi`

```
# repeat query every 2 second and pipe result to separate file
nvidia-smi -query-gpu=
timestamp,index,compute_mode,pstate,utilization.gpu,utilization.memory,memory.us
ed,temperature.gpu,power.draw --format=csv --loop=2 &> data.txt &
```

```
# remember ID of process that has just been started in background
export proc_id_monitor=$!
```

```
# execute your program
<exec statement> <params>
```

```
# stop monitoring again
kill -2 ${proc_id_monitor}
sleep 5
```

```
timestamp, index, compute_mode, pstate, utilization.gpu [%],
utilization.memory [%], ...
2021/11/12 14:24:52.010, 1, Default, P0, 0 %, 0 %, 0 MiB, 40, 56.54 W
2021/11/12 14:24:54.012, 0, Default, P0, 81 %, 20 %, 2758 MiB, 44, 145.93 W
2021/11/12 14:24:54.013, 1, Default, P0, 0 %, 0 %, 0 MiB, 40, 56.01 W
2021/11/12 14:24:56.015, 0, Default, P0, 82 %, 22 %, 2758 MiB, 45, 147.40 W
2021/11/12 14:24:56.016, 1, Default, P0, 0 %, 0 %, 0 MiB, 38, 42.31 W
2021/11/12 14:24:58.017, 0, Default, P0, 82 %, 22 %, 2758 MiB, 44, 156.98 W
2021/11/12 14:24:58.019, 1, Default, P0, 0 %, 0 %, 0 MiB, 38, 41.82 W
2021/11/12 14:25:00.020, 0, Default, P0, 86 %, 22 %, 2758 MiB, 44, 139.66 W
2021/11/12 14:25:00.022, 1, Default, P0, 0 %, 0 %, 0 MiB, 38, 41.82 W
2021/11/12 14:25:02.023, 0, Default, P0, 83 %, 21 %, 2758 MiB, 45, 151.70 W
```

Basic Resource Utilization Checks with `nvidia-smi`

