

# LUMI

Scaling AI training  
to multiple GPUs

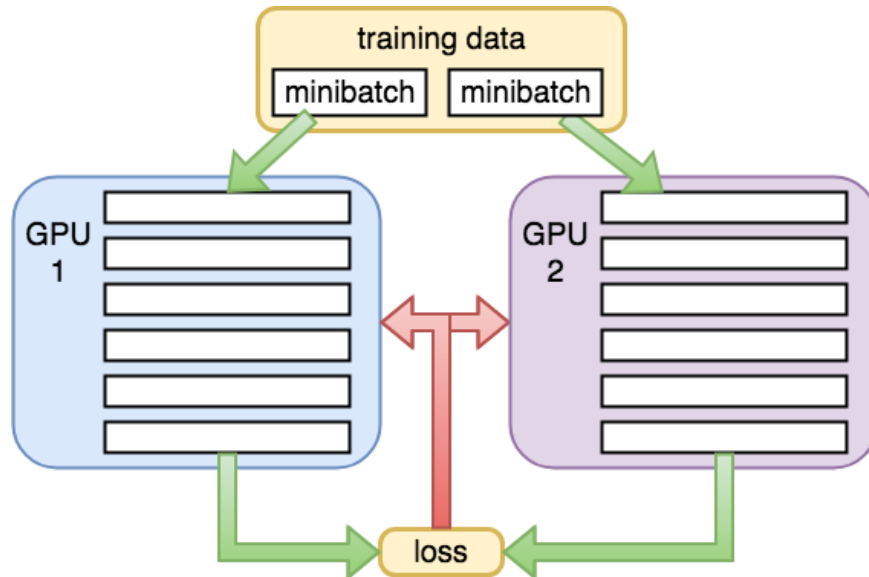


Mats Sjöberg, Marlon Tobaben – CSC – IT Center for Science, Finland

# Reasons to use multiple GPUs

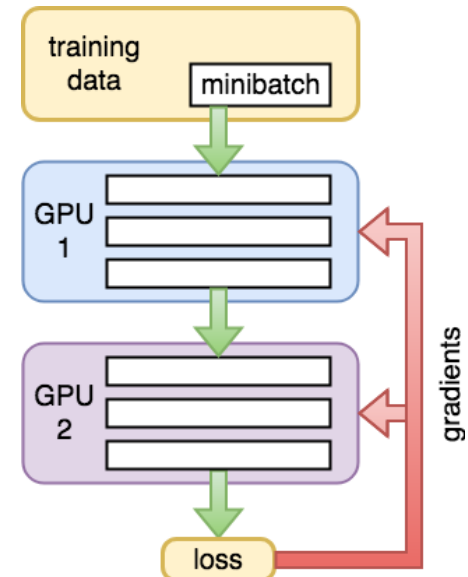
*Training takes a long time and  
I have a lot of data*

→ data parallelism



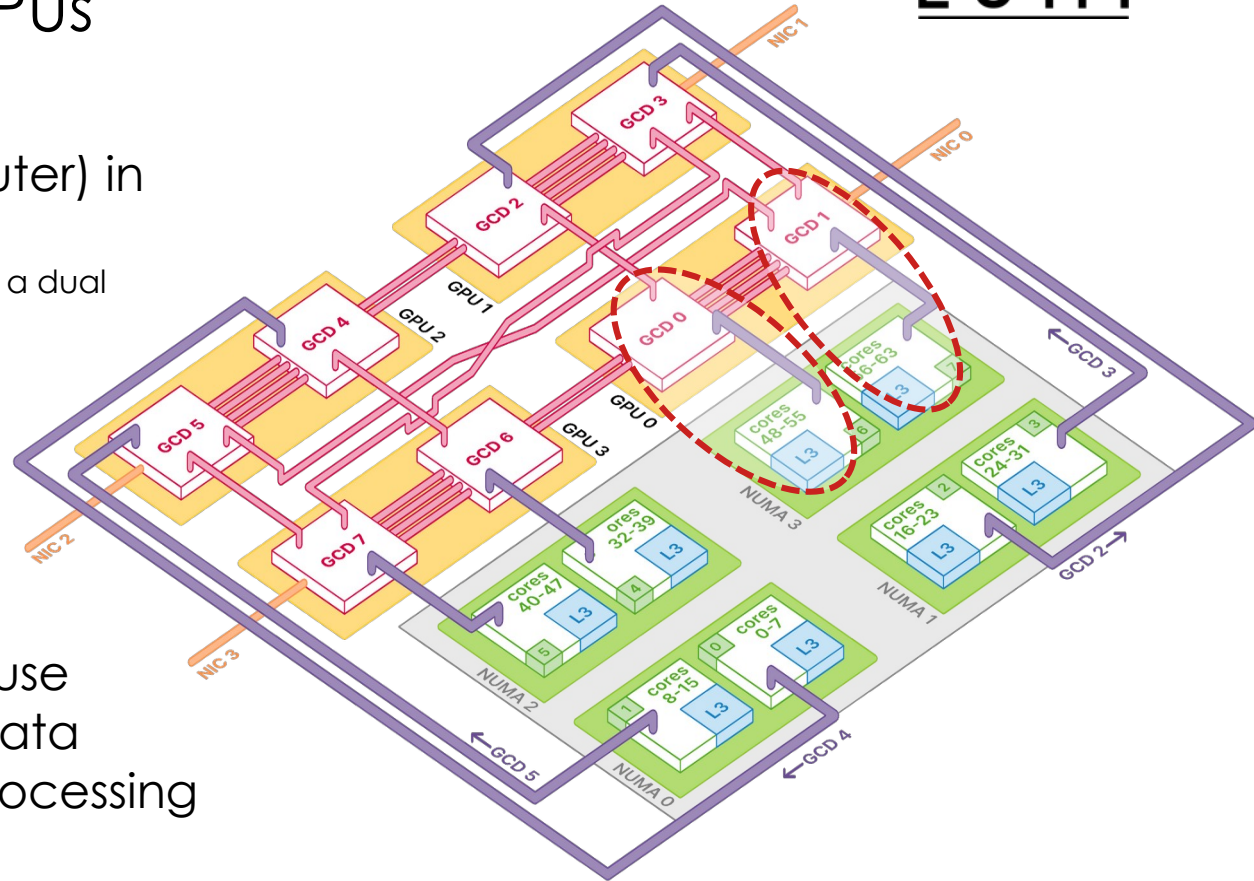
*My model is too big to fit  
into one GPU*

→ model parallelism



# Using multiple GPUs

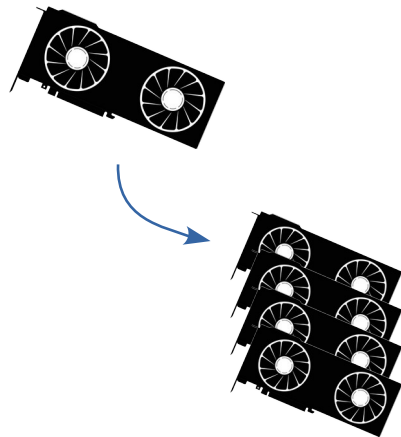
- Each node (computer) in LUMI has 8 GPUs  
(Actually 4 x MI250x, which is a dual chip card = 8 GCDs)
- For each GPU you use 1-7 CPU cores for data loading and pre-processing



# Using multiple GPUs

LUMI

- Not automatic: **your code needs to support multiple GPUs**
- Frameworks like Hugging Face, Lightning or Accelerate *may* auto-detect multiple GPUs (with the right options)
- For pure PyTorch code, there are many options depending on the scenario:
  - DistributedDataParallel (DDP)
  - Hybrid approaches for models too big for a single GPU:
    - Fully-sharded Data Parallel (FSDP)
    - DeepSpeed



## Multi-GPU resource allocation on LUMI

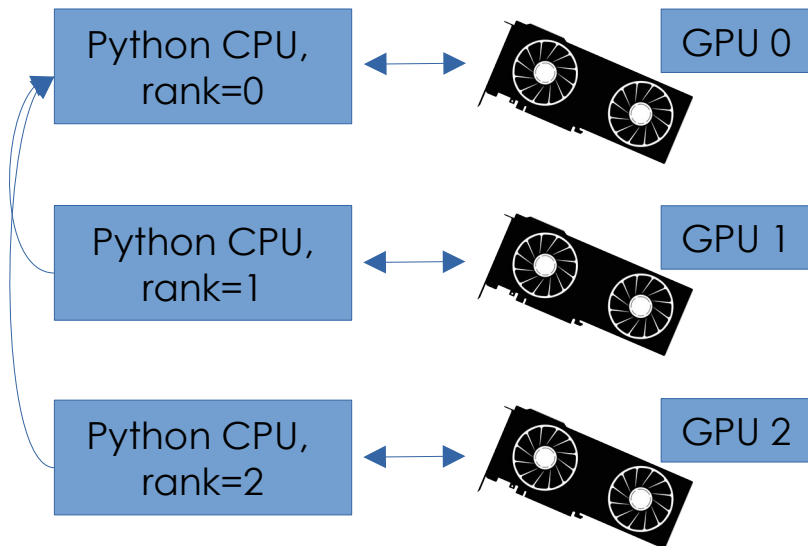
- Use `--gpus-per-node=N` where  $N=1, \dots, 8$ 
  - `--gpus-per-task` option not currently recommended due to bug in Slurm
- Max 8 GPUs in one node, for more GPUs, add more nodes:  
`--nodes=M`
  - More on multi-*node* jobs in the next lecture

# Multi-GPU resource allocation on LUMI



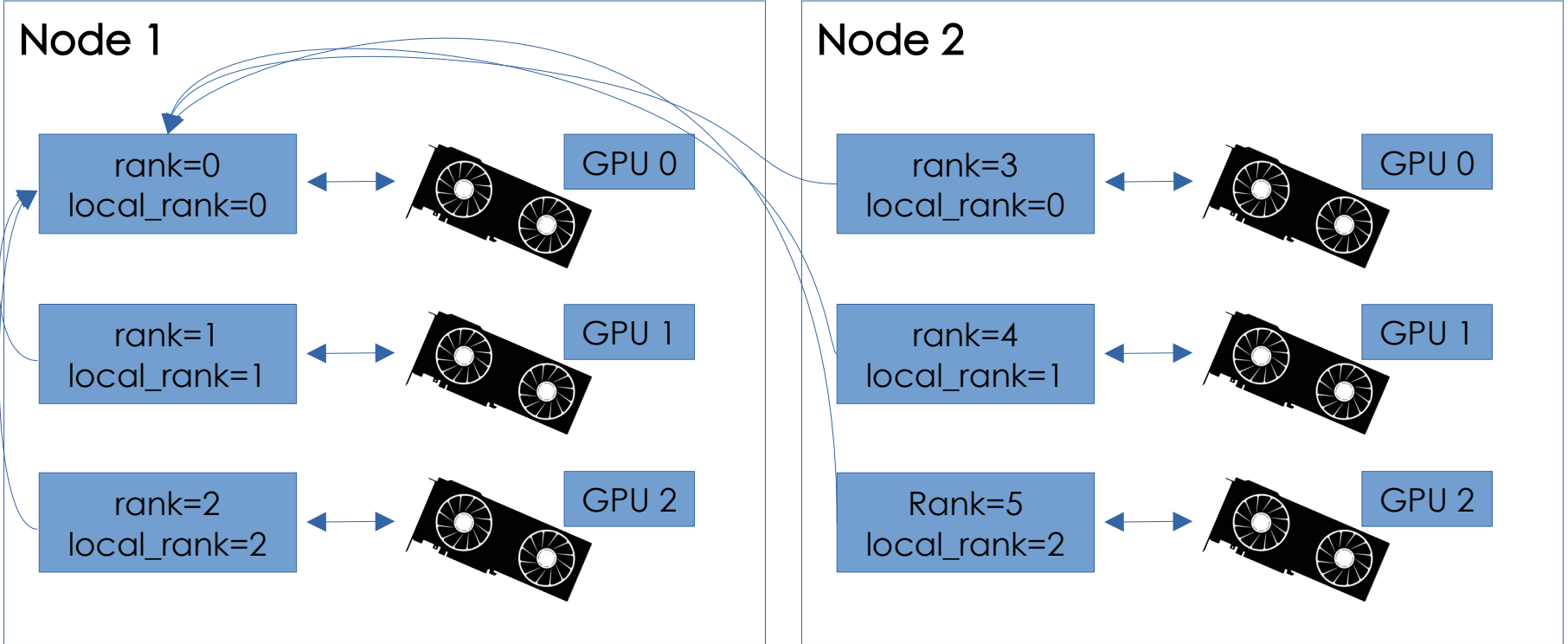
- Allocate a maximum of **1/8 of resources per GPU**:
  - 60 GB **CPU memory (RAM)** and 7 CPU cores per GPU
  - Full node: 480 GB and 56 cores  
(leaving some "slack" for the system)
  - Note: you **always get the full GPU memory (VRAM)** of 64 GB per GCD (no need to allocate that with Slurm options)
  - Resources billed in GPUh according to 1/8 slice

# One Python CPU control process per GPU



- We start one Python process (CPU) per GPU
- Each process needs to know which GPU it should talk to, given by the process **rank**
- The first process (rank=0) is the main process, and the others connect back to it

# Multiple nodes: rank and local\_rank



## Example: one full node, 8 GPUs

In the Python code:

```
gpu_id = int(os.environ["LOCAL_RANK"])  
device = torch.device("cuda", gpu_id)
```

```
#!/bin/bash  
#SBATCH --account=project_NNNNNNNN  
#SBATCH --partition=standard-g  
#SBATCH --gpus-per-node=8  
#SBATCH --ntasks-per-node=8  
#SBATCH --cpus-per-task=7  
#SBATCH --mem=480G  
#SBATCH --time=1:00:00  
## < module loading part as before - removed for readability>
```

*The line with **srun** will be launched multiple times according to the number of tasks*

```
export MASTER_ADDR=$(hostname)
```

```
export MASTER_PORT=25900
```

*Where to connect to?*

```
export WORLD_SIZE=$SLURM_NPROCS
```

*How many processes are there?*

*Which process am I?*

```
srun singularity run $CONTAINER \  
  bash -c "RANK=\$SLURM_PROCID LOCAL_RANK=\$SLURM_LOCALID python ..."
```

## Example: 2 nodes, $2 \times 8 = 16$ GPUs in total

```
#!/bin/bash
#SBATCH --account=project_NNNNNNNN
#SBATCH --partition=standard-g
#SBATCH --nodes=2
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=8
#SBATCH --cpus-per-task=7
#SBATCH --mem=480G
#SBATCH --time=1:00:00
## < module loading part as before - removed for readability>

export MASTER_ADDR=$(hostname)
export MASTER_PORT=25900
export WORLD_SIZE=$SLURM_NPROCS

srun singularity run $CONTAINER \
    bash -c "RANK=\$SLURM_PROCID LOCAL_RANK=\$SLURM_LOCALID python ..."
```

## Example: one full node, 8 GPUs, *with torchrun*

```
#!/bin/bash
#SBATCH --account=project_NNNNNNNN
#SBATCH --partition=standard-g
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=56
#SBATCH --mem=480G
#SBATCH --time=1:00:00
## < module loading part as before – removed for readability>

srun singularity run $CONTAINER \
    torchrun --standalone \
        --nnodes=1 \
        --nproc-per-node=${SLURM_GPUS_PER_NODE} \
        my_python_script.py
```

*Torchrun will take care of launching multiple processes, Slurm just needs to start one torchrun*

Example: 2 nodes,  $2 \times 8 = 16$  GPUs in total, with torchrun

**L U M I**

```
#!/bin/bash
#SBATCH ... < skipping some common Slurm options >
#SBATCH --nodes=2
#SBATCH --gpus-per-node=8
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=56
#SBATCH --mem=480G
## < module loading part as before – removed for readability>
```

*srun is needed again, as we want  
Slurm to start ONE torchrun PER node*

```
export RDZV_HOST=$(hostname)
export RDZV_PORT=25900
```

```
srun singularity run $CONTAINER \  
  torchrun --rdzv_id=${SLURM_JOB_ID} --rdzv_backend=c10d \  
          --rdzv_endpoint="$RDZV_HOST:$RDZV_PORT" \  
          --nnodes=${SLURM_JOB_NUM_NODES} \  
          --nproc-per-node=${SLURM_GPUS_PER_NODE} \  
          \
```

*Torchrun has it's own rendezvous  
mechanism for connecting to the  
main node, essentially we again need  
to tell it the hostname and port*

# Do we need to change the Python code?

- For plain PyTorch: **yes, use DistributedDataParallel (DDP)**
- For higher level frameworks, **mostly no**:
  - `transformers.Trainer` is automatically set up for distributed training when `WORLD_SIZE` & `RANK` environment variables are set
  - Similar for other high-level frameworks like PyTorch Lightning or Accelerate
- BUT: Pay attention to **global batch size vs per device batch size!**
  - Example: global batch size = 32 for one GPU, split over 8 GPUs, per-device batch size is 4
- Cosmetic: You might want to print some things only on rank 0

# PyTorch DistributedDataParallel (DDP)

1) Initialize PyTorch distributed:

```
torch.distributed.init_process_group(backend='nccl')
```

2) Wrap your model:

```
model = torch.nn.parallel.DistributedDataParallel(model, ...)
```

3) Use the distributed sampler:

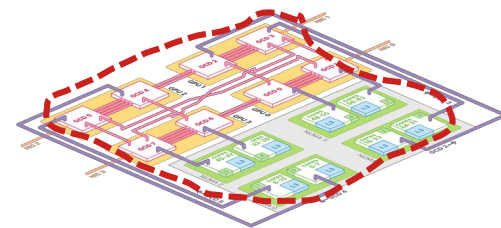
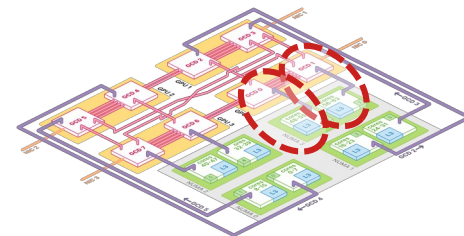
```
train_dataset = ...  
train_sampler = DistributedSampler(train_dataset)  
train_loader = DataLoader(dataset=train_dataset,  
                           shuffle=False,  
                           sampler=train_sampler)
```

# Check that you are actually using all GPUs!



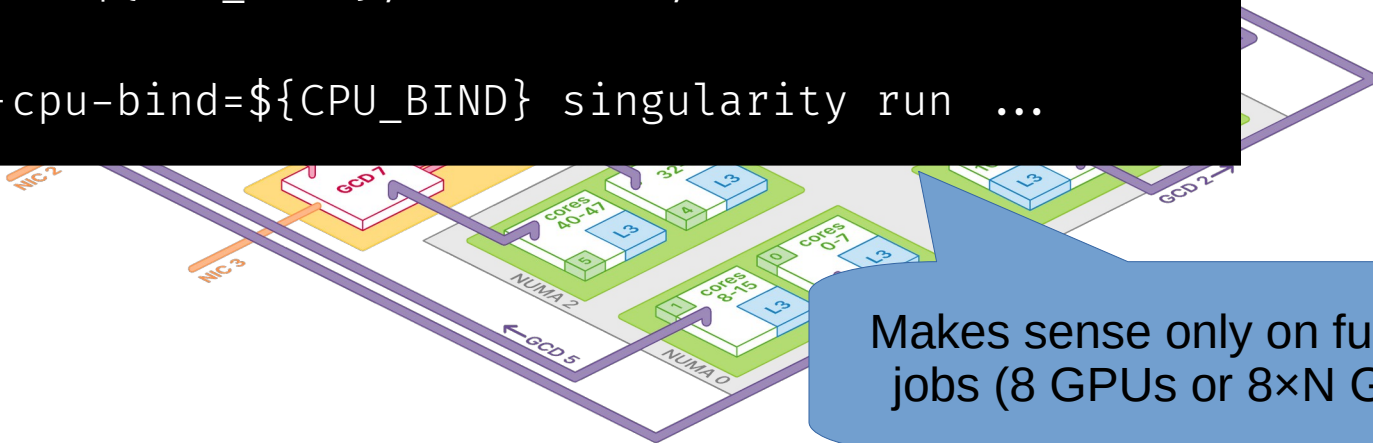
```
Check GPU utilization
$ srun --overlap --pty --jobid=123456 watch rocm-smi
```

- Utilization should be  $> 0\%$  for all requested GPUs
- Note: showing high utilization is a **necessary, but not sufficient** condition for it actually doing something useful!
  - Refer back to lecture 4 yesterday *Understanding GPU activity & checking jobs*
  - Check GPU power and use profiling



# GPU and CPU Bindings

```
CPU_BIND="mask_cpu:fe000000000000,fe00000000000000"  
CPU_BIND="${CPU_BIND},fe0000,fe000000"  
CPU_BIND="${CPU_BIND},fe,fe00"  
CPU_BIND="${CPU_BIND},fe00000000,fe000000000000"  
  
srun --cpu-bind=${CPU_BIND} singularity run ...
```



Makes sense only on full-node jobs (8 GPUs or 8×N GPUs)

