

LUMI

Scaling AI training
to multiple GPUs

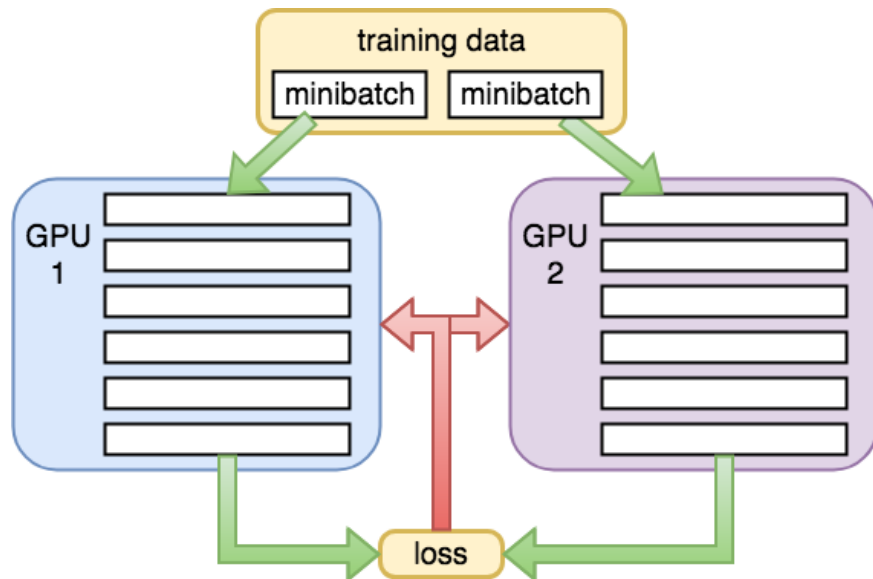


Mats Sjöberg, Oskar Taubert – CSC – IT Center for Science, Finland

Reasons to use multiple GPUs

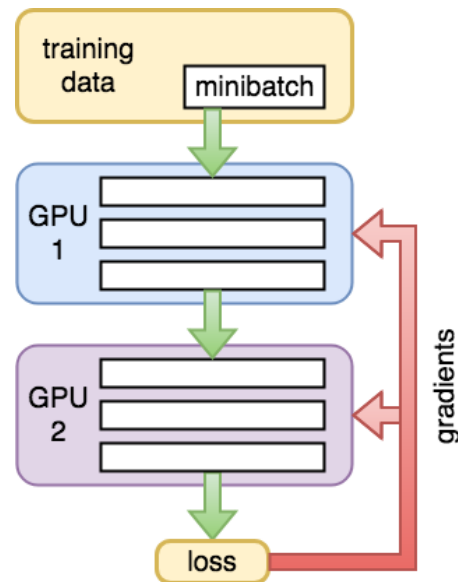
*Training takes a long time and
I have a lot of data*

→ data parallelism



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

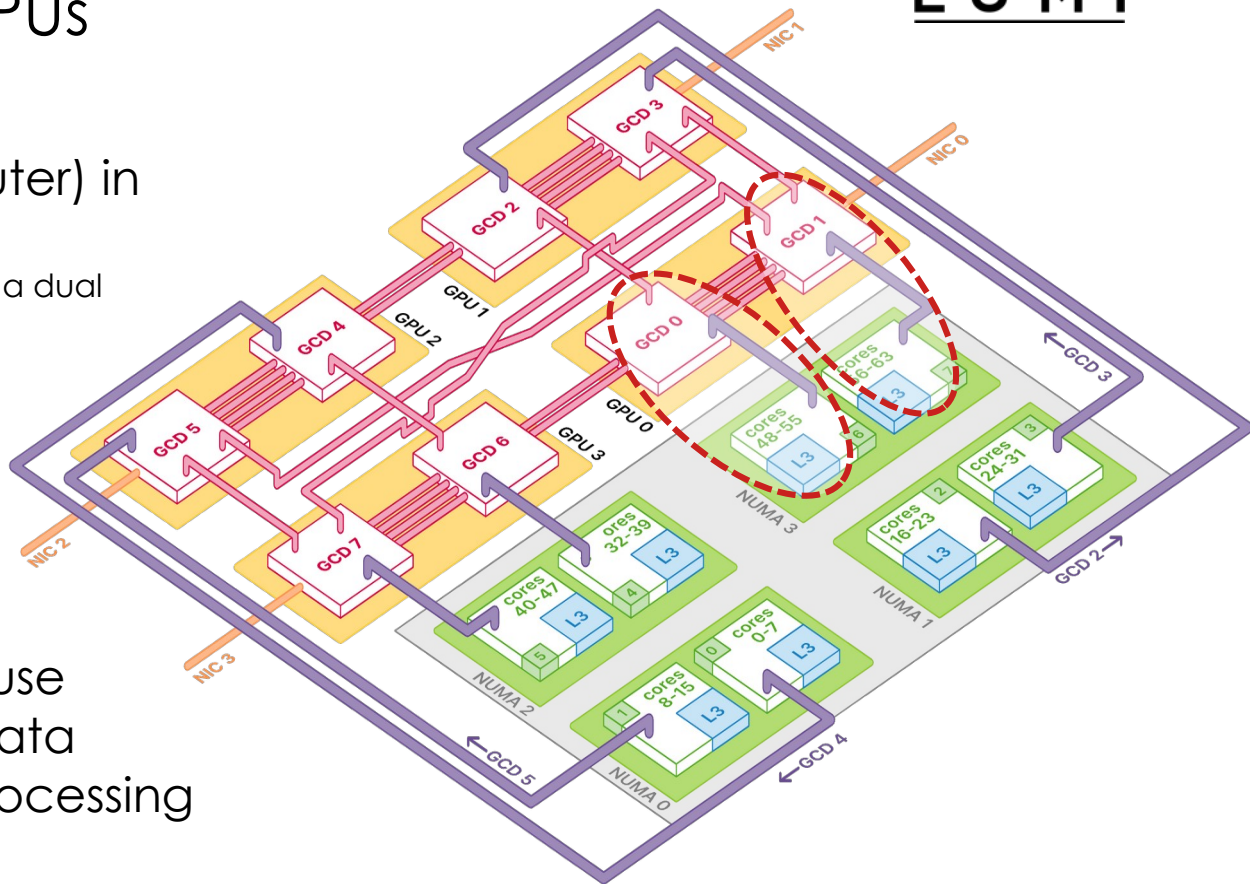
→ model parallelism



Using multiple GPUs

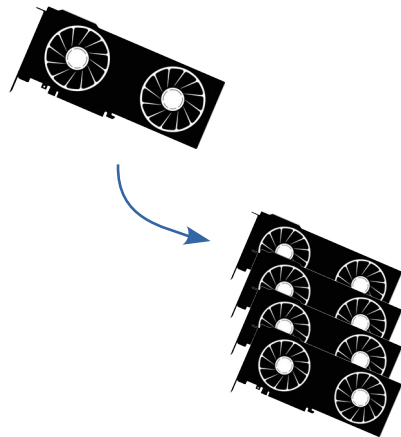
LUMI

- 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

- 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

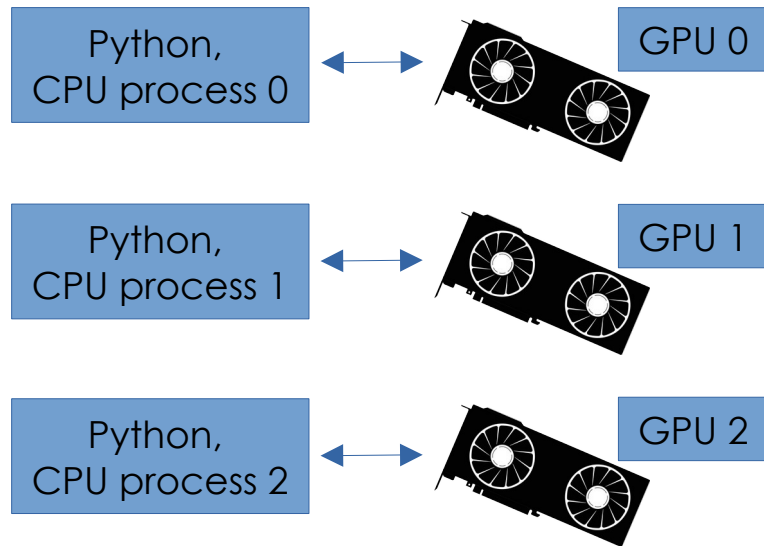
- Use **--gpus-per-node=N** where $N=1, \dots, 8$
(`--gpus-per-task` option not currently recommended due to bug in Slurm)
- Allocate a maximum of **1/8 of resources per GPU**:
 - 60 GB **CPU memory (RAM)** and 7 CPU cores per GPU
 - Full node: 480 GB and 56 cores
(leaving some "slack" for the system)
 - 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

LUMI

- Torchrn can handle launching the processes. Launch single torchrn:
`--tasks-per-node=1`
- Without torchrn, use Slurm tasks:
`--tasks-per-node=8`
- Each process should know which GPU to use, e.g.

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

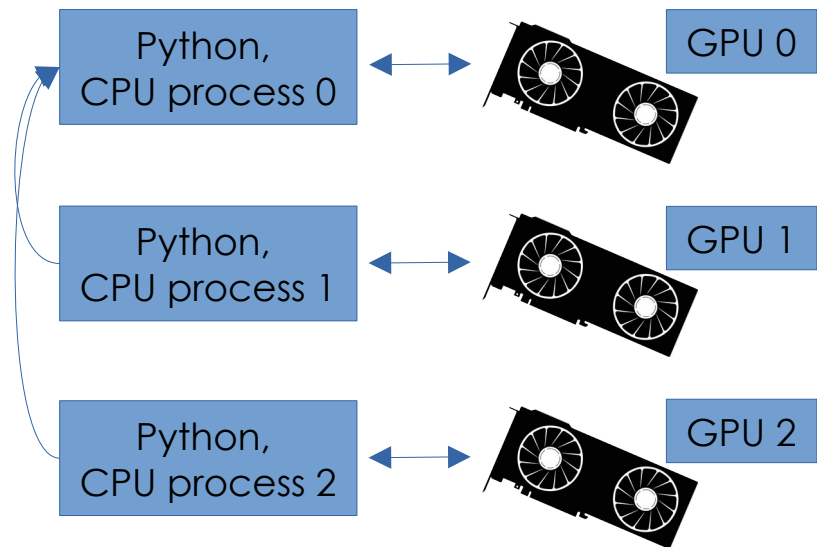


Set up communication between processes

- First process is the "master" and the others connect back to it
- Torchrund can handle this automatically:
`torchrund --standalone ...`
- Without torchrund you need to set up environment variables:

```
MASTER_ADDR=$(hostname)  
MASTER_PORT=25900  
WORLD_SIZE=$SLURM_NPROCS  
RANK=$SLURM_PROCID
```

- Note: rank needs to be set differently for each process, see exercise for example



Example: 2 GPUs with torchrun

```
#!/bin/bash
#SBATCH --account=project_465001958
#SBATCH --partition=small-g
#SBATCH --gpus-per-node=2
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=14
#SBATCH --mem=120G
#SBATCH --time=1:00:00
## < module loading part as before - removed for readability>

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

Remember rule-of-thumb:

- 1 GPU = 1/8 of node
- Use also $\leq 1/8$ of CPU cores and memory *per GPU*

torchrun will take care of launching one process per GPU

Example: 8 GPUs with torchrun

```
#!/bin/bash
#SBATCH --account=project_465001958
#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 exec $CONTAINER \
    torchrun --standalone \
        --nnodes=1 \
        --nproc-per-node=${SLURM_GPUS_PER_NODE} \
        my_python_script.py
```

Full node = we can also use
standard-g

Example: 8 GPUs *without* torchrun

LUMI

```
#!/bin/bash
#SBATCH --account=project_465001958
#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>
```

We use Slurm tasks to launch 8 Python processes

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

```
export MASTER_PORT=24500
```

Where to connect to?

```
export WORLD_SIZE=$SLURM_NPROCS
```

How many processes are there?

Which process am I?

```
srun bash -c "RANK=\$SLURM_PROCID LOCAL_RANK=\$SLURM_LOCALID singularity exec ..."
```

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:

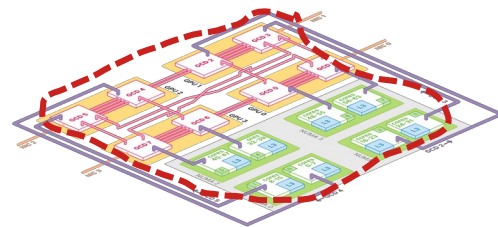
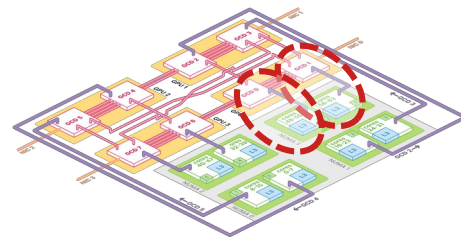
```
train_dataset = ...  
train_sampler = DistributedSampler(train_dataset)  
train_loader = DataLoader(dataset=train_dataset,  
                           shuffle=False,  
                           sampler=train_sampler)
```

Check that you are actually using all GPUs!

LUMI

```
Check GPU utilization
$ srun --interactive --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



Multi-GPU tips & tricks

- Use RCCL and libfabric for efficient communication on LUMI
 - AWS OFI RCCL plugin for containers:
<https://github.com/ROCm/aws-ofi-rccl>
- Add fault tolerance when possible, especially for huge jobs
 - Checkpointing!
- (Optionally) bind the processes to optimal CPU cores
 - Improves CPU-GPU I/O, might speed up cases with high I/O
- Issues with multi-worker data loaders segfaulting:

```
if __name__ == '__main__':  
    multiprocessing.set_start_method("spawn")
```


GPU and CPU Bindings

LUMI

Example: GCD 4:
`psutil.Process().cpu_affinity([1,2,3,4,5,6,7])`

Why do we skip
CPU 0?
Because in LUMI the
first core in each
NUMA is reserved

