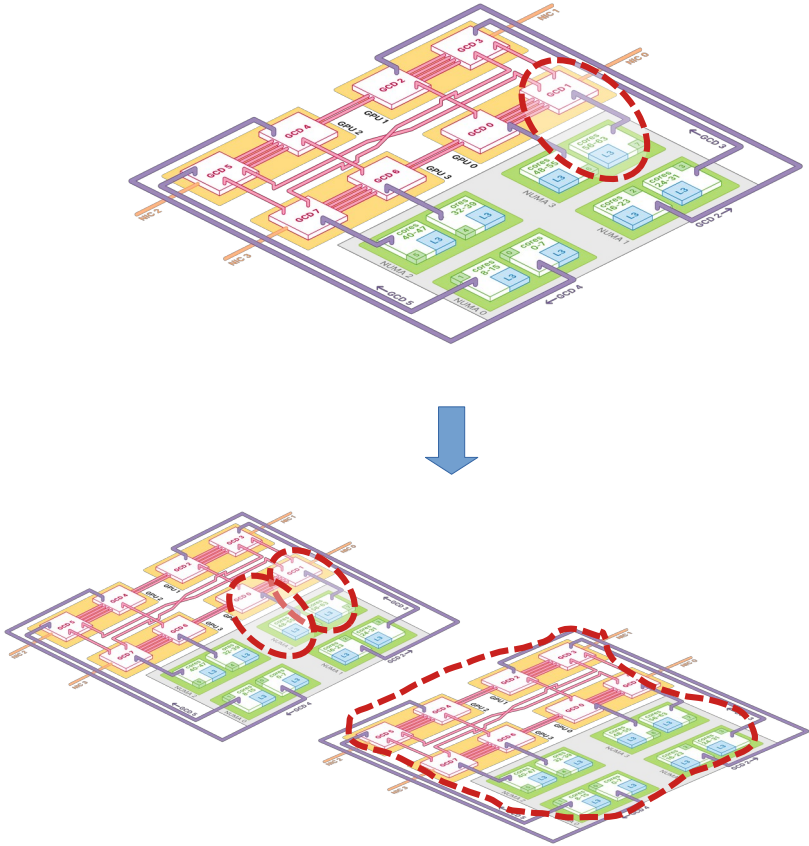
Scaling AI training to multiple GPUs

Mats Sjöberg, Lukas Prediger – CSC – IT Center for Science, Finland

Using multiple GPUs

- Not automatic: your code needs to support multiple GPUs
  - Frameworks like Hugging Face, PyTorch Lightning or accelerate may auto-detect multiple GPUs (with the right options)
  - For pure PyTorch code use DistributedDataParallel (DDP)
- Pick a distributed training strategy
  - If your model fits into the GPU memory = data parallel = PyTorch DDP
  - If your model > GPU memory (64 GB on LUMI) look into model and pipeline parallelism (see FSDP, DeepSpeed and others)



Multi-GPU resource allocation on LUMI

LUMI

- Use `--gpus-per-node` rather than `–gpus-per-task`
  - Due to bug in Slurm, fix coming...
- Rule-of-thumb, allocate **~1/8 of resources per GCD**:
  - 60 GB RAM and 7 CPU cores per GPU
  - Full node: 480 GB and 56 cores (leaving some "slack" for the system)

Multi-GPU resource allocation on LUMI

- All approaches use **one Python process per GCD**:
  - Can be done with Slurm (`--tasks-per-node=8`)
  - Tools like torchrun can handle launching the processes – use Slurm only to launch a single task (`--tasks-per-node=1`)
  - Each process should know which GPU to use, via `$ROCR_VISIBLE_DEVICES=$SLURM_LOCALID` or `$LOCAL_RANK`
    - In PyTorch:

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

Multi-GPU communication and tips

LUMI

- Setting up communication between the processes (e.g., setting `$MASTER_ADDR` or `--rdzv-endpoint`)
  - Also remember RCCL and libfabric for efficient communication
- 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")
```

Training on two GCDs on a single node

SLURM multi-GPU batch script (2 GPUs)

```
#!/bin/bash
#SBATCH --account=project_123456
#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 ≤ 1/8 of CPU cores and memory *per GPU*

torchrun will take care of launching one process per GPU

Training on all GCDs on a single node

SLURM multi-GPU batch script (all 8 GPUs)

```
#!/bin/bash
#SBATCH --account=project_123456
#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`

Training on all GCDs on a single node without torchrun

LUMI

SLURM multi-GPU batch script (all 8 GPUs, no torchrun)

```
#!/bin/bash
#SBATCH --account=project_123456
#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>

export MASTER_ADDR=$(scontrol show hostname ${SLURM_NODELIST} | head -n 1)
export MASTER_PORT=24500                                   Where to connect to?
export WORLD_SIZE=$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 code?

LUMI

- 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)

LUMI

- Recommended for pure ("low-level") PyTorch
- Data parallel
  - Model is duplicated on many GPUs
  - Data is distributed, and gradient updates aggregated
- PyTorch DDP supports both single- and multi-node runs
- Launch with `torchrun`
- Uses a dedicated Python process for each GPU
- (Not to be confused with PyTorch DataParallel (DP) which uses multi-threading – not recommended to use)

PyTorch DistributedDataParallel (DDP)

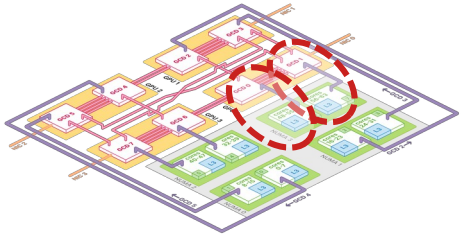
PyTorch DDP code changes

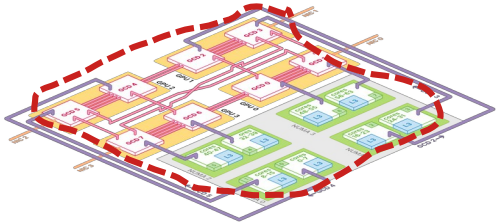
```
torch.distributed.init_process_group(backend='nccl')
…
model = torch.nn.parallel.DistributedDataParallel(model, …)
…
train_dataset = …
train_sampler = DistributedSampler(train_dataset)
train_loader = DataLoader(dataset=train_dataset,
                          shuffle=False,
                          sampler=train_sampler)
```

Check that you are actually using all GPUs!

Check GPU utilization

```
$ srun --overlap --pty --jobid=987654 bash
@compute_node$ rocm-smi
```

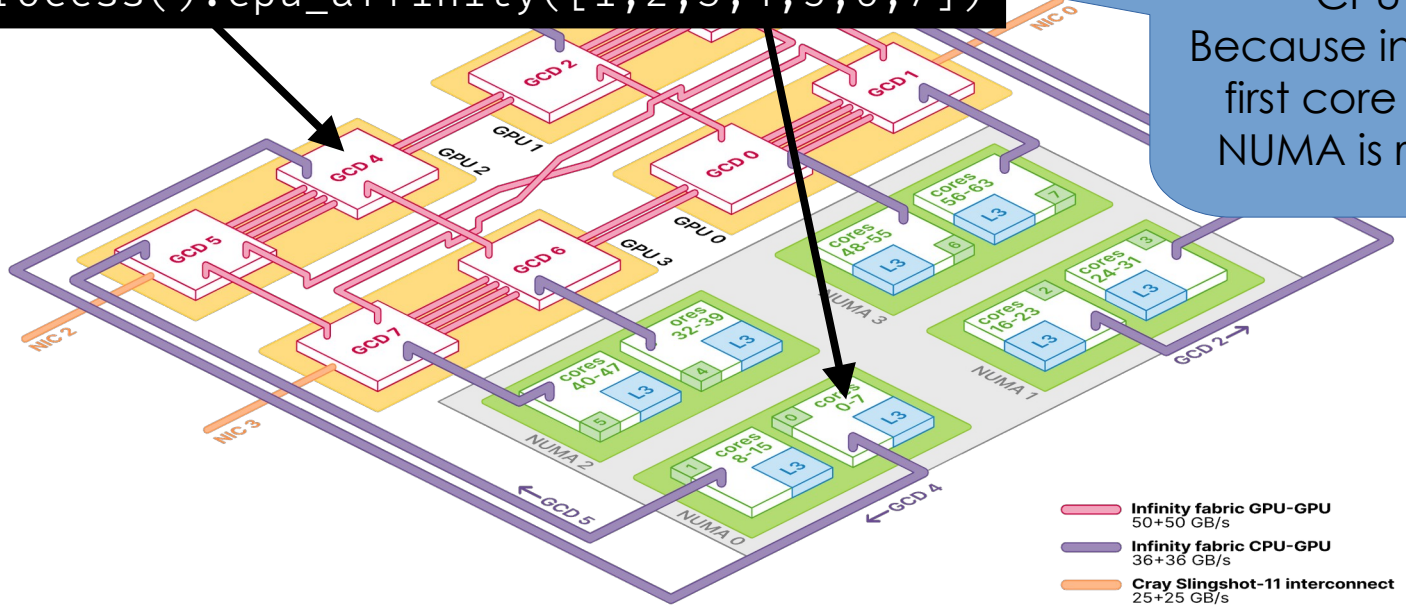




GPU and CPU Bindings

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



https://docs.lumi-supercomputer.eu/runjobs/scheduled-jobs/distribution-binding/#gpu-binding