VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

AI-FRIENDLY EUROHPC SYSTEMS

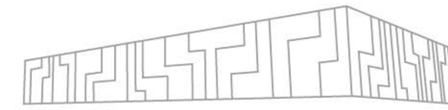
BEST PRACTICES AND USEFUL INSTRUCTIONS

Khyati Sethia

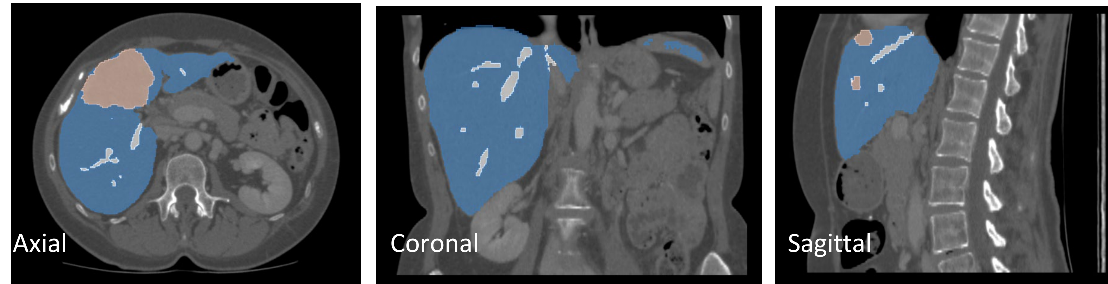
Researcher

IT4Innovations National Supercomputing Center,
Czech Republic

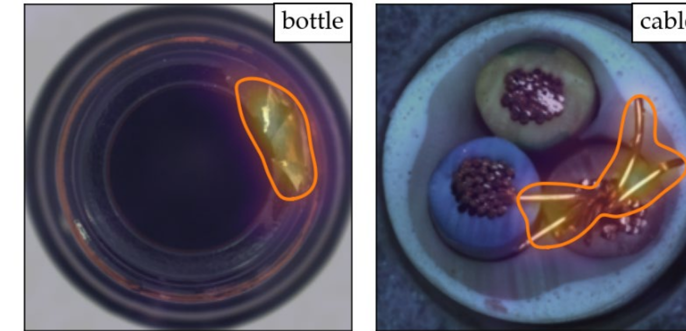
SOME OF THE AI PROJECTS ON KAROLINA INFRASTRUCTURE



Liver tumor and veins segmentation

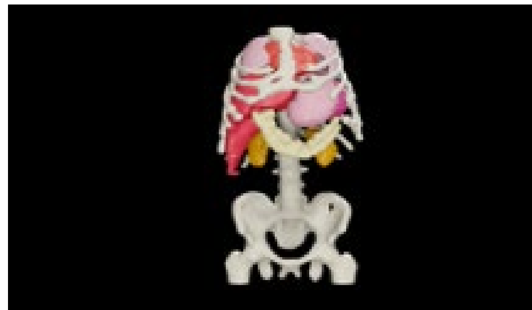


Anamoly detection on 3D models



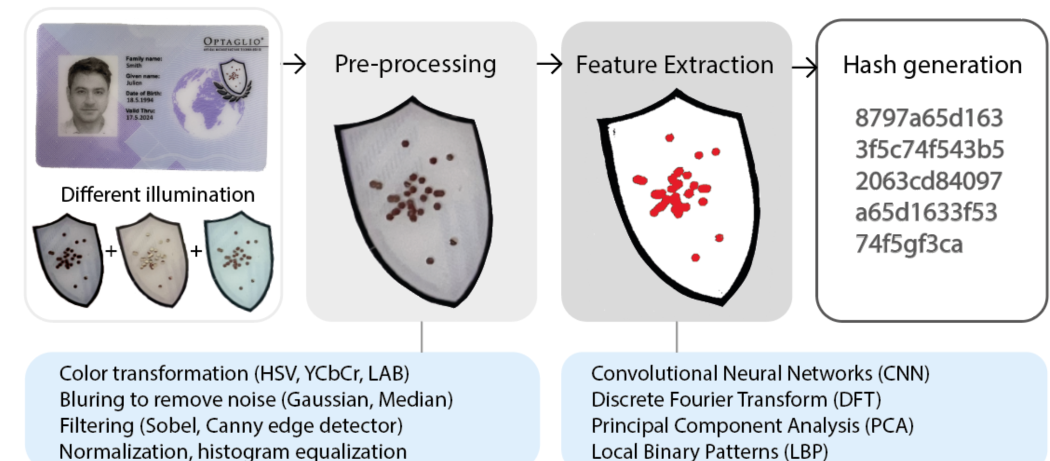
Source: https://doi.org/10.48550/arXiv.2106.08265

Multi organ segmentation

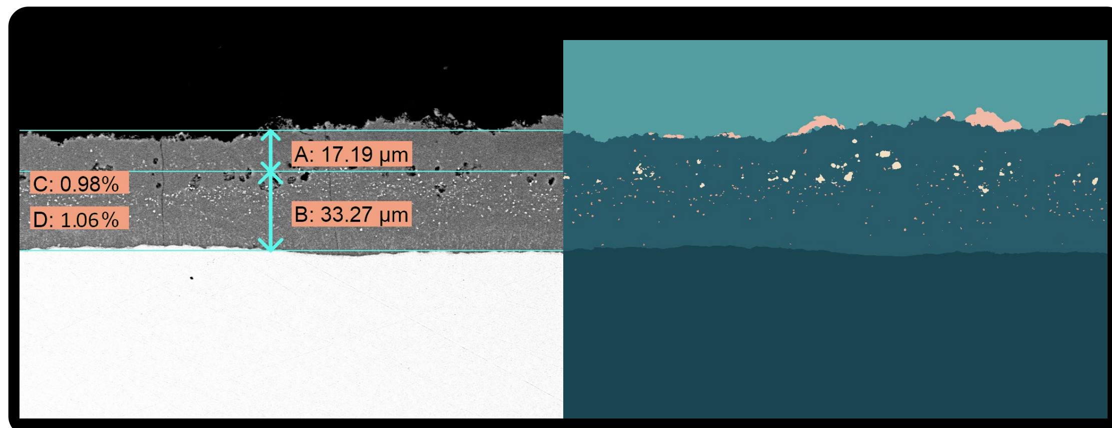


Object detection

Combination of holographic and digital safety protection

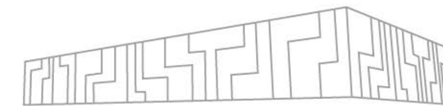


Analysing SEM images of slurry coatings



VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER

KAROLINA INFRASTRUCTURE



CPU 0
AMD EPYC 7763
64 cores
512 GiB DDR4

Infinity fabric

CPU 1
AMD EPYC 7763
64 cores
512 GiB DDR4

PCIe 4.0 x16
32 GB/s

NIC
IB 200 Gb/s

NIC
IB 200 Gb/s

NIC
IB 200 Gb/s

NIC
IB 200 Gb/s

PCIe switch

PCIe switch

PCIe switch

PCIe switch

GPU 0
A100
40 GiB
HBM2

GPU 1
A100
40 GiB
HBM2

GPU 2
A100
40 GiB
HBM2

GPU 3
A100
40 GiB
HBM2

GPU 4
A100
40 GiB
HBM2

GPU 5
A100
40 GiB
HBM2

GPU 6
A100
40 GiB
HBM2

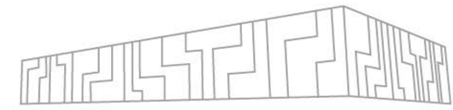
GPU 7
A100
40 GiB
HBM2

NVLink 3.0
300+300 GB/s

NVSwitch network

More Info: https://docs.it4i.cz/karolina/compute-nodes/

ACCESSING THE KAROLINA CLUSTER



Get Access:

- Obtain an e-INFRA cz or IT4I Account.
- Get project.
- Generate an SSH key for authentication to clusters.

Connect to Karolina cluster via SSH:

```
local $ ssh -i /path/to/id_rsa username@karolina.it4i.cz
```

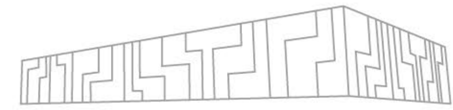
Recommendations:

- Configure SSH to simplify connections: Use the ~/.ssh/config file.

```
Host karolina
   HostName login2.karolina.it4i.cz
   IdentityFile /path/to/id_rsa
   User username
```

More Info: https://docs.it4i.cz/general/access/account-introduction/
https://docs.it4i.cz/general/accessing-the-clusters/shell-access-and-data-transfer/ssh-keys/

DATA TRANSFER ON KAROLINA CLUSTER



Serial Transfer Methods:

- SCP: secure copy files between hosts on a network.
- SSHFS: Mount Karolina directory locally via SSH
- SFTP: Interactive file transfer and directory browsing.

```
local $ scp -i /path/to/id_rsa -r my-local-dir username@karolina.it4i.cz:directory
```

Parallel Transfer Methods:

- rsync: Efficient syncing for large datasets.
- HyperQueue: Efficient for many small files.

Specialized Transfers:

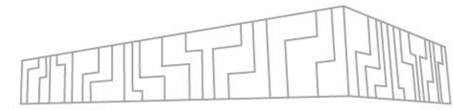
- hqtransfer: Optimized for single large file.

Midnight Commander: File manager for managing local & remote files within a terminal.

Windows Users: Use PuTTY and WinSCP for SCP and SFTP.

More Info: https://docs.it4i.cz/general/shell-and-data-access/

ALLOCATING CLUSTER RESOURCES



CPU

- Running Batch Jobs

```
#!/usr/bin/bash
#SBATCH --job-name MyJobName
#SBATCH --account PROJECT-ID
#SBATCH --partition qcpu
#SBATCH --time 12:00:00
#SBATCH --nodes 8
...
```

- Running Interactive Jobs

```
$ salloc -A PROJECT-ID -p qcpu -N 4 --ntasks-per-node 128 -t 2:00:00
```

GPU

- Running Batch Jobs

```
#!/usr/bin/bash
#SBATCH --job-name MyJobName
#SBATCH --account PROJECT-ID
#SBATCH --partition qgpu
#SBATCH --time 12:00:00
#SBATCH --gpus-per-node 8
#SBATCH --nodes 2
...
```

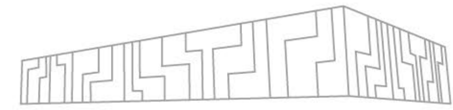
- Running Interactive Jobs

```
$ salloc -A PROJECT-ID -p qgpu --gpus-per-node=8 --nodes=2 -t 03:00:00
```

- Slurm workload manager is used to allocate and access Karolina cluster's resources.
- Using GPU Queues, Nodes per job limit is 16.
- Karolina Dashboard: https://extranet.it4i.cz/grafana/d/IYj4zYL7k/karolina-cluster?orgId=1&kiosk&refresh=1m
- Graphical representation of cluster usage, partitions, nodes, and jobs could be found at https://extranet.it4i.cz/rsweb/karolina

More info: https://docs.it4i.cz/general/karolina-slurm/

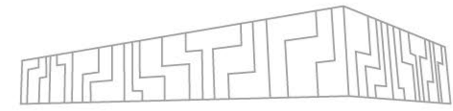
MODULES ON KAROLINA



- List Available Modules:
  - Use ml av or module avail.
    - Example: ml av tensorflow
- Search for Modules:
  - Use ml spider <module_name> or module spider.
- Load a Module:
  - Execute ml <module_name> or module load <module_name>.
- Unload a Module:
  - Use ml -<module_name> or module unload <module_name>.
- List Loaded Modules:
  - Use ml or module list.
- Module unload all modules:
  - ml purge.
- Module show:
  - module show CUDA.

More info: https://docs.it4i.cz/software/modules/lmod/
https://docs.it4i.cz/modules-karolina/

MULTIPLE WAYS TO RUN DL FRAMEWORKS



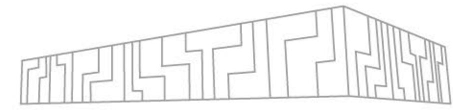
- **Conda Environments**
  - Conda is used when project requires non-Python dependencies, such as specific versions of libraries or tools that are not available via pip.
  - It includes its own package manager and can handle complex dependencies that involve compiled libraries, making it suitable for more complex environments.

- **Virtual Environments (venv)**
  - Python's built-in tool to create isolated environments.
  - Venv is used for smaller projects or projects that only need Python packages.

- **Singularity/Apptainer**
  - Apptainer is a container platform that allows you to create portable, reproducible containers on your laptop and run them across various environments like HPC clusters, cloud, or workstations, simplifying software deployment with a single file.

- **Direct Install**
  - Install PyTorch directly on the system without isolation.

CONDA ENVIRONMENTS



Pytorch conda environment

- Requirements yml file

```
name: conda_env_pytorch_gpu
channels:
  - pytorch
  - nvidia
  - defaults
  - conda-forge
dependencies:
  - python=3.10
  - pip
  - numpy>=1.24,<2.0
  - pytorch==2.4.1
  - torchvision==0.19.1
  - torchaudio==2.4.1
  - pytorch-cuda=12.4
  - tensorboard==2.17.0
  - pip:
    - monai==1.3.2
    - nibabel==5.2.1
    - matplotlib==3.9.2
    - opencv-python==4.10.0.84
    - tqdm==4.66.5
```

- Create a conda environment (conda_env_pytorch_gpu_create.sh)

```
#!/bin/bash

ml purge
conda env create --prefix
/path_to_env/conda_env_pytorch_gpu -f
/path_to_env/conda_env_pytorch_gpu_requirements.yml
conda activate /path_to_env/conda_env_pytorch_gpu
```

- Running the script

```
./conda_env_pytorch_gpu_create.sh
```

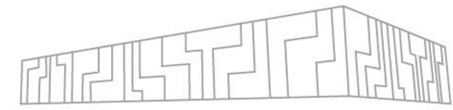
- Running the environment (conda_env_pytorch_gpu_run.sh)

```
#!/usr/bin/bash
#SBATCH --job-name conda_env_pytorch_gpu
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 1
#SBATCH --time 0:02:00
#SBATCH --gpus-per-node 1
#SBATCH --ntasks-per-node 1

conda deactivate
ml purge
ml Anaconda3/2024.02-1
. "/apps/all/Anaconda3/2024.02-1/etc/profile.d/conda.sh“

# Run environment
conda activate /path_to_env/conda_env_pytorch_gpu/bin/activate

# Run python script
python -c "import torch; print(f'PyTorch version:
{torch.__version__}'); print(f'CUDA available:
{torch.cuda.is_available()}'); print(f'CUDA version:
{torch.version.cuda}')"
```

- Running the script

```
sbatch conda_env_pytorch_gpu_run.sh
```

CONDA ENVIRONMENTS



Tensorflow conda environment

- Requirements yml file

```
name: conda_env_tensorflow_gpu
channels:
  - defaults
  - nvidia
dependencies:
  - python=3.8
  - pip==23.3.1
  - cudatoolkit=11.8.0
  - cudnn==8.9.2.26
  - tensorflow[and-cuda]==2.13.1
  - pillow==10.0.1
  - matplotlib==3.7.2
  - scikit-learn==1.3.0
  - pip:
    - colormath==3.0.0
    - opencv-python==4.9.0.80
    - tifffile==2023.7.10
```

- Create a conda environment (conda_env_tensorflow_gpu_create.sh)

```
#!/bin/bash

ml purge
conda env create --prefix /path_to_env/
conda_env_tensorflow_gpu -f /path_to_env/
conda_env_tensorflow_gpu_requirements.yml
```

- Running the script

```
./conda_env_tensorflow_gpu_create.sh
```

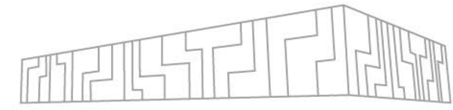
- Running the environment (conda_env_tensorflow_gpu_run.sh)

```
#!/usr/bin/bash
#SBATCH --job-name conda_env_tensorflow_gpu
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 1
#SBATCH --time 0:02:00
#SBATCH --gpus-per-node 1
#SBATCH --ntasks-per-node 1

conda deactivate
ml purge
ml Anaconda3/2024.02-1
. "/apps/all/Anaconda3/2024.02-1/etc/profile.d/conda.sh“
ml CUDA/11.7.0

export LD_LIBRARY_PATH+=":/apps/all/CUDAcore/11.6.0/lib64"
export LD_LIBRARY_PATH+=":/path_to_tensorrt/tensorrt"

# Run environment
conda activate /path_to_conda_env/bin/activate

# Run python script
python -c "import tensorflow as tf; print('TensorFlow version:',
tf.__version__); print('GPUs:', tf.config.list_physical_devices('GPU'))"
```

- Running the script

```
sbatch conda_env_tensorflow_gpu_run.sh
```

VIRTUAL ENVIRONMENTS (VENV)



Pytorch virtual environment

- Requirements file

```
numpy>=1.24,<2.0
torch==2.4.1
torchvision==0.19.1
torchaudio==2.4.1

# Specify the extra index URL for PyTorch CUDA builds
--index-url https://download.pytorch.org/whl/cu121

# Reset back to the default PyPI index for other packages
--index-url https://pypi.org/simple

monai==1.3.2
nibabel==5.2.1
tqdm==4.66.5
opencv-python==4.10.0.84
matplotlib==3.9.2
tensorboard==2.17.1
```

- Create a virtual environment (venv_pytorch_gpu_create.sh)

```
#!/bin/bash

ml purge
ml Python/3.10.8-GCCcore-12.2.0

python3.10 -m venv /path_to_env/venv_pytorch_gpu
source /path_to_env/venv_pytorch_gpu/bin/activate

python -m pip install --upgrade pip
pip cache purge
pip install -r venv_pytorch_gpu_requirements.txt

which python
echo $PYTHONPATH
```

- Running the script

```
./venv_pytorch_gpu_create.sh
```

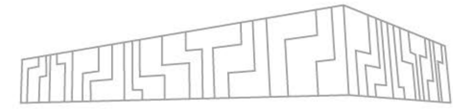
- Running the environment (venv_pytorch_gpu_run.sh)

```
#!/usr/bin/bash
#SBATCH --job-name venv_pytorch_gpu
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 1
#SBATCH --time 0:02:00
#SBATCH --gpus-per-node 1
#SBATCH --ntasks-per-node 1

# Load modules
ml --force purge
ml Python/3.10.8-GCCcore-12.2.0

# Run environment
source /path_to_env/venv_pytorch_gpu/bin/activate

# Run python script
python -c "import torch; print(f'PyTorch version:
{torch.__version__}'); print(f'CUDA available:
{torch.cuda.is_available()}'); print(f'CUDA version:
{torch.version.cuda}')"
```

- Running the script

```
sbatch venv_pytorch_gpu_run.sh
```

VIRTUAL ENVIRONMENTS (VENV)



Tensorflow virtual environment

- Requirements file

```
# Core TensorFlow with GPU support
tensorflow[and-cuda]==2.13.1

# For TensorRT optimizations
tensorrt

# Common ML and scientific computing packages
numpy>=1.24,<2.0
pandas==2.2.3
scipy==1.14.1
matplotlib==3.9.2
h5py==3.11.0
pillow==10.4.0
opencv-python==4.10.0.84
tqdm==4.66.5
tensorboard==2.13.0

# Medical imaging (optional)
nibabel==5.2.1

# For deep learning model serialization (optional)
protobuf<4.0.0
```

- Create a virtual environment (venv_tensorflow_gpu_create.sh)

```
#!/bin/bash

ml purge
ml Python/3.10.8-GCCcore-12.2.0
ml CUDAcore
ml CUDA/11.7.0

python3.10 -m venv /path_to_env/venv_tensorflow_gpu
source /path_to_env /venv_tensorflow_gpu/bin/activate

python -m pip install --upgrade pip
pip cache purge
pip install -r venv_tensorflow_gpu_requirements.txt
```

- Running the script

```
./venv_tensorflow_gpu_create.sh
```

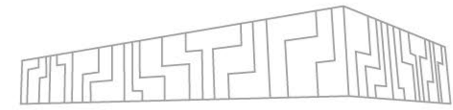
- Running the environment (venv_tensorflow_gpu_run.sh)

```
#!/usr/bin/bash
#SBATCH --job-name venv_tensorflow_gpu
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 1
#SBATCH --time 0:02:00
#SBATCH --gpus-per-node 1
#SBATCH --ntasks-per-node 1

# Load modules
ml --force purge
ml Python/3.10.8-GCCcore-12.2.0

export LD_LIBRARY_PATH+=":/path_to_CUDA"
export LD_LIBRARY_PATH+=":/path_to_tensorrt"

# Run environment
source /path_to_env/venv_tensorflow_gpu/bin/activate

# Run python script
python -c "import tensorflow as tf; print('TensorFlow version:',
tf.__version__); print('GPUs:', tf.config.list_physical_devices('GPU'))"
```

- Running the script

```
sbatch venv_tensorflow_gpu_run.sh
```

CONTAINERS



- Load apptainer

```
ml apptainer
```

- Pull Container from Library/Hub

```
apptainer pull <container_name.sif> <library://user/collection/container:tag>
```

- Run a Container

```
apptainer run <container_name.sif>
```

- Execute a Command Inside a Container

```
singularity exec <container_name.sif> <command>
```

- Shell into a Container

```
singularity shell <container_name.sif>
```

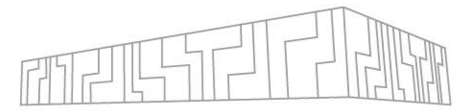
- Mount a Host Directory or Bind Path

```
apptainer exec --bind /path/on/host:/path/in/container <container_name.sif> <command>
```

```
apptainer exec -B /path/on/cluster:/workspace -B /apps/all/CUDAcore/11.6.0/:/usr/local/cuda --nv
container.simg /bin/bash
```

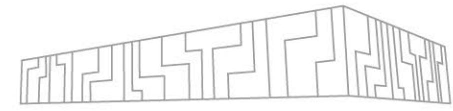
```
singularity shell -B /apps/all/CUDAcore/11.6.0/:/usr/local/cuda --nv monai.sif
```

More info: https://docs.it4i.cz/software/tools/apptainer/



EXAMPLE PROJECT

ABOUT THE PROJECT



This project is focused on 3D medical image segmentation. Specifically, targeting liver tumor segmentation from CT scans. It is optimizing the training process by using distributed data parallelism and training on multiple GPUs.

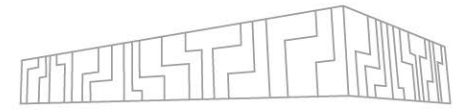
1
- Data Preprocessing and Loading
  - Loading medical image files (in NIfTI format, .nii or .nii.gz).
  - Data augmentation and preprocessing, such as intensity scaling, cropping, flipping, rotation, and more.

2
- Model Training
  - Distributed Training
  - Distributed data parallel (DDP)
  - TensorBoard Logging
  - Checkpointing and Early Stopping
  - Validation on a separate validation set
  - Sliding window inference
  - DiceCELoss Loss function

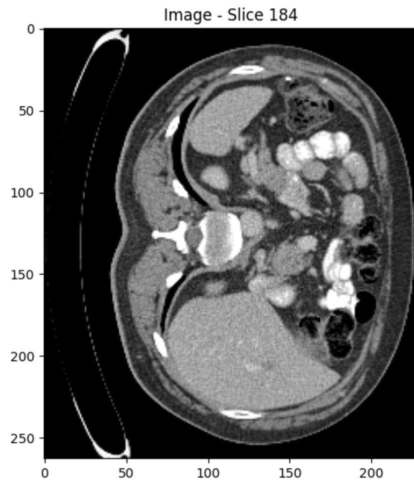
3
- Inference
  - Segment liver tumors from new, unseen medical images

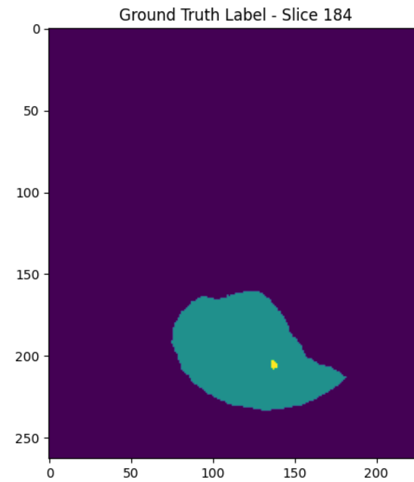
3
Post-Processing and Analysis

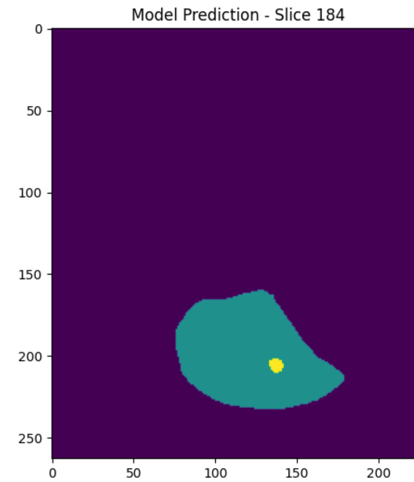
ABOUT THE PROJECT

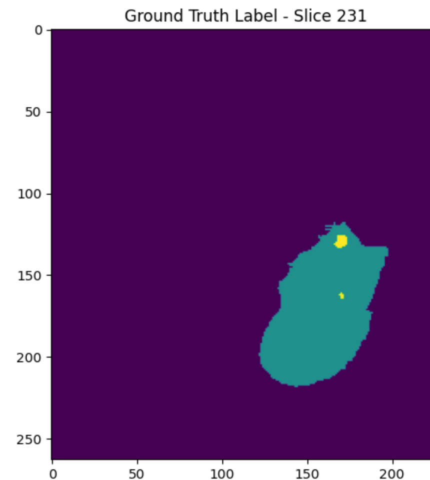


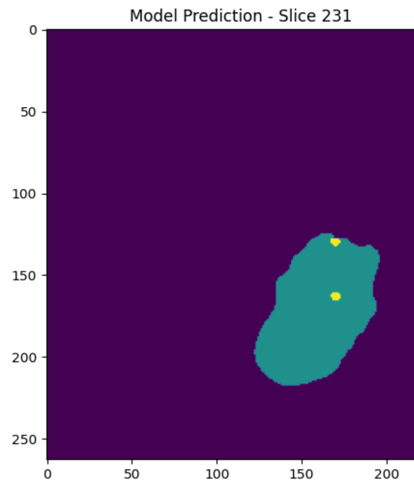
Inference on new unseen dataset.

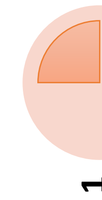






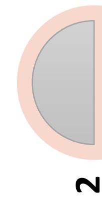






Time on single GPU

- **5 hours 40 minutes to reach validation dice score to 85%**

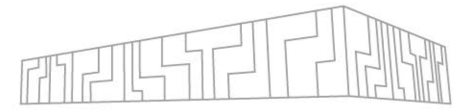


Time on 16 GPUs

- **50 minutes to reach validation dice score to 84%**

Code: https://code.it4i.cz/set0013/liver_tumor_segmentation.git

PARALLEL PROCESSING ON MULTIPLE GPUS

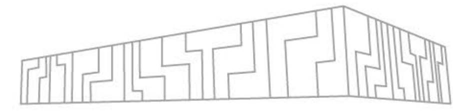


- **Set and run SLURM bash script (job_script.sh)**

```
#!/usr/bin/bash
#SBATCH --job-name Liver_Tumor_Seg_Distributed_Training
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 2
#SBATCH --time 02:00:00
#SBATCH --gpus-per-node 8
#SBATCH --ntasks-per-node 8

export MASTER_ADDR=$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head -n 1)
export MASTER_PORT=12340

# Activate your conda environment
conda deactivate
ml purge
ml Anaconda3/2024.02-1
. "/apps/all/Anaconda3/2024.02-1/etc/profile.d/conda.sh"
conda activate /path_to_conda_env

which python

# Run the Python script with srun
srun python /path_to_script/train.py
```

```
sbatch job_script.sh
```

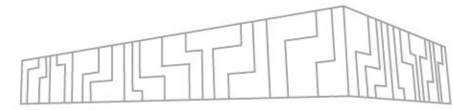
PARALLEL PROCESSING ON MULTIPLE GPUS



- Environment Variables
  - The script uses environment variables such as RANK, WORLD_SIZE, and SLURM_LOCALID to assign each GPU a unique rank and determine the world size (number of processes across nodes). This ensures each GPU is assigned to a specific process for parallel training.
- Distributed Data Parallel Initialization
  - The script initializes the process group with torch.distributed.init_process_group(backend="nccl"), which is the required backend for multi-GPU training on NVIDIA hardware.

```
def setup_distributed():
    rank = int(os.environ['SLURM_PROCID'])
    local_rank =  int(os.environ['SLURM_LOCALID'])
    world_size = int(os.environ['SLURM_NTASKS'])
    ngpus = int(os.environ['SLURM_GPUS_ON_NODE'])

    torch.cuda.set_device(local_rank)
    dist.init_process_group(backend="nccl", rank=rank, world_size=world_size)
    return rank, local_rank, world_size

rank, local_rank, world_size = setup_distributed()
```

PARALLEL PROCESSING ON MULTIPLE GPUS



- Using DistributedSampler for DataLoader
  - The script uses a DistributedSampler to split the dataset among GPUs. Each process only receives a subset of the data, ensuring data parallelism across GPUs.

```
train_sampler = DistributedSampler(train_ds, num_replicas=world_size, rank=rank)

train_loader = DataLoader(train_ds, batch_size=2, sampler=train_sampler, pin_memory=True,
num_workers=4)
```

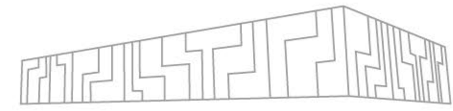
- Moving Model to Device
  - The model needs to be explicitly moved to the appropriate GPU (device) before training begins.

```
model = (model).to(device)
```

- Using DistributedDataParallel (DDP) for the Model
  - The script wraps the model with torch.nn.parallel.DistributedDataParallel (DDP), ensuring that the gradients are synchronized between the GPUs.

```
model = DDP(model, device_ids=[local_rank], output_device=local_rank)
```

PARALLEL PROCESSING ON MULTIPLE GPUS



- Synchronization via dist.barrier
  - Before saving the model, the script uses dist.barrier() to ensure all processes finish before proceeding.

```
dist.barrier()
```

- Adjust the Learning Rate
  - If you increase the number of GPUs, the effective batch size will increase. To counterbalance this, you may need to adjust the learning rate by scaling it with the number of GPUs.

```
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4 * world_size)
```

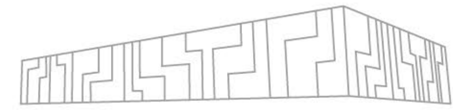
- Model Saving
  - Only the process with rank == 0 saves the model, preventing multiple processes from trying to save simultaneously.

```
if rank == 0:
    torch.save(model.state_dict(), "weights/model_name.pth")
```

- TensorBoard Logging
  - Only the main process (rank 0) logs metrics to TensorBoard.

```
if rank == 0:
    writer = SummaryWriter(log_dir=log_dir)
```

TENSORFLOW WITH TENSORBOARD



TensorBoard provides the visualization and tooling needed for machine learning experimentation:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph (ops and layers)
- Viewing histograms of weights, biases, or other tensors as they change over time
- Projecting embeddings to a lower dimensional space
- Displaying images, text, and audio data
- Profiling TensorFlow programs

```
import os
import datetime
from tensorflow.keras.callbacks import TensorBoard

root_dir = os.getcwd()
log_dir = os.path.join(root_dir, 'logs', 'fit', datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))

tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

callbacks = [tensorboard_callback]
# assuming model, train_dataset, val_dataset, num_of_epochs are already defined
history = model.fit(train_dataset, validation_data=val_dataset, epochs=num_of_epochs, callbacks=callbacks)
```

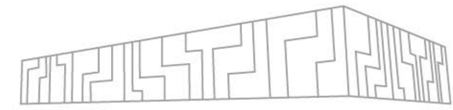
PORT FORWARDING:

- SSH port forwarding is used to access TensorBoard running on Karolina and to monitor locally.

```
ssh -i path_to_id_rsa -L 6006:127.0.0.1:6006 userid@login1.karolina.it4i.cz
conda activate path_to_virtual_env
conda install tensorboard
tensorboard --logdir=paath_to_logs --port=6006
```

View in local browser: http://localhost:6006

CHECKPOINTING



- Periodically save the model's state during training using checkpoints.
- Checkpoints are essential for large datasets and long training jobs (e.g., 48+ hours on Karolina) to ensure training can resume after interruptions.
- Checkpoints typically save the following: Model weights, Optimizer state, Training epoch/batch
- Benefits of checkpointing:
  - Training continuation: Resume training from the last saved point if the job is interrupted.
  - Fine-tuning: Use a pre-trained checkpoint to fine-tune the model on a new dataset or with different hyperparameters.
  - Experimentation: Test different hyperparameters (e.g., learning rate, batch size) without starting from scratch.

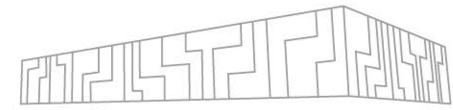
TensorFlow Example:

```
# Save model checkpoint
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(filepath=filepath, save_weights_only=True)

# Load saved checkpoint and resume training
model.load_weights(filepath)
```

PyTorch Example:

```
# Save checkpoint
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    }, PATH)

# Load checkpoint and resume training
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']
```

JOB DEPENDENCY



- We're setting up two jobs on the Karolina cluster, where the second job uses a checkpoint from the first. The second job starts only after the first completes successfully, allowing for efficient resource use by resuming long-running computations instead of starting over.

job1.slurm

```
#!/bin/bash
#SBATCH --job-name=program1
#SBATCH --output=program1.out
#SBATCH --error=program1.err
#SBATCH -A OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 2
#SBATCH --time 1:00:00
#SBATCH --gpus-per-node 8
#SBATCH --ntasks-per-node 8

ml python/3.8
python program1.py --checkpoint
program1_checkpoint.ckpt # Specify checkpoint output
```

job2.slurm

```
#!/bin/bash
#SBATCH --job-name=program2
#SBATCH --output=program2.out
#SBATCH --error=program2.err
#SBATCH -A OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 2
#SBATCH --time 1:00:00
#SBATCH --gpus-per-node 8
#SBATCH --ntasks-per-node 8

ml python/3.8
python program2.py --checkpoint
program1_checkpoint.ckpt # Load checkpoint from the first
job
```

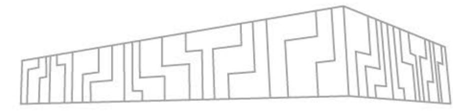
submit_jobs.sh

```
#!/bin/bash

# Submit the first job
jobid1=$(sbatch job1.slurm)
jobid1=${jobid1##* }  # Extract the job ID

# Submit the second job with a dependency on the first job
jobid2=$(sbatch --dependency=afterok:$jobid1 job2.slurm)
jobid2=${jobid2##* }

echo "Submitted jobs with dependencies:"
echo "Job 1 ID: $jobid1"
echo "Job 2 ID: $jobid2"
```

run submit_jobs.sh

```
$ chmod +x submit_jobs.sh
$ ./submit_jobs.sh

$ squeue --me
```

DISTRIBUTED HYPERPARAMETER TUNING



- Ray Tune is commonly used automated hyperparameter tuning library

Define the Neural Network Model

```
class Net(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

Setup Ray Tune for Distributed Tuning

```
search_space = {
    "hidden_size": tune.randint(64, 512),
    "learning_rate": tune.loguniform(1e-5, 1e-1),
}

scheduler = ASHAScheduler(
    metric="loss",
    mode="min",
    max_t=10,  # Max number of epochs
    grace_period=1,
    reduction_factor=2
)
```

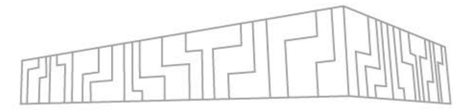
Train Function with GPU Support

```
def train_model(config, checkpoint_dir=None):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = Net(784, config["hidden_size"], 10).to(device)
    optimizer = optim.Adam(model.parameters(),
lr=config["learning_rate"])

    for epoch in range(10):
        inputs = torch.randn(64, 784).to(device)  # Fake data
        labels = torch.randint(0, 10, (64,)).to(device)
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        tune.report(loss=loss.item())
```

Ray Tune Distributed Setup

```
tune.run(
    DistributedTrainableCreator(train_model,
num_workers=2, use_gpu=True),
    resources_per_trial={"cpu": 2, "gpu": 1},
    config=search_space,
    scheduler=scheduler,
    num_samples=10,  # Number of trials
)
```

DISTRIBUTED HYPERPARAMETER TUNING



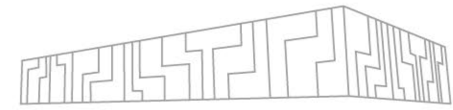
SLURM Job Script: ray_tune_gpu.sh

```
#!/usr/bin/bash
#SBATCH --job-name distributed_hyperparameter_tuning
#SBATCH --account OPEN-28-64
#SBATCH --partition qgpu
#SBATCH --nodes 2
#SBATCH --time 1:00:00
#SBATCH --gpus-per-node 8
#SBATCH --ntasks-per-node 8
#SBATCH --output=out_%A_%a.log
#SBATCH --error=err_%A_%a.log

export MASTER_ADDR=$(scontrol show hostnames
"$SLURM_JOB_NODELIST" | head -n 1)
export MASTER_PORT=12340

# Activate your conda environment
conda deactivate
ml purge
ml Anaconda3/2024.02-1
. "/apps/all/Anaconda3/2024.02-1/etc/profile.d/conda.sh"
conda activate /path_to_env/pytorch_env_gpu2

which python

# Run the Python script with srun
srun python
/path_to_script/distributed_hyperparameter_tuning_gpu.py
```

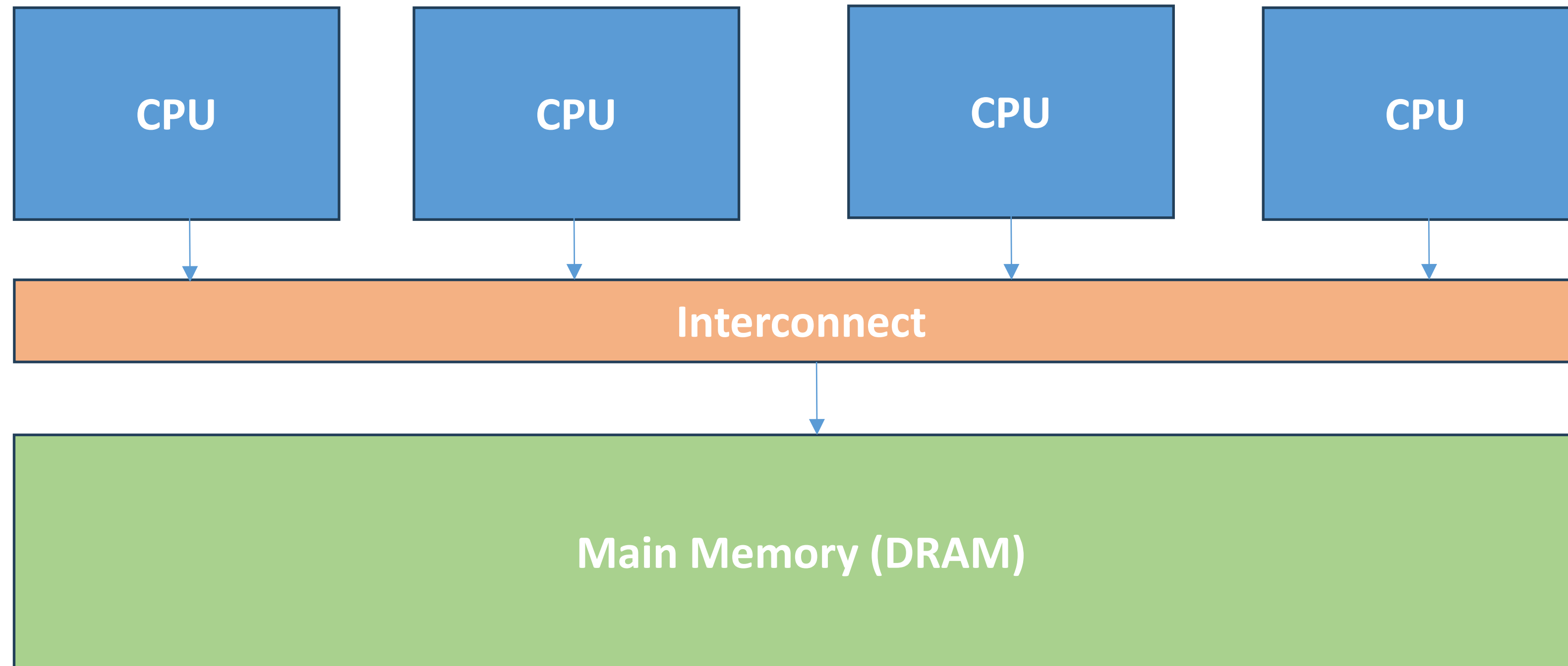
run ray_tune_gpu.sh

```
$ sbatch ray_tune_gpu.sh
```

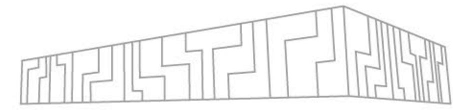
OPENMP



- OpenMP is primarily designed for multi-threading on shared memory architectures.
- OpenMP is not inherently suited for multi-GPU training, which often requires distributed memory management
- Can be used for Parallel Inference.



OPENMP



- Conceptual Steps to Use OpenMP for Parallel Inference:

C Code: Implements parallel inference using OpenMP.

```
#include <omp.h>

void parallel_inference(float* inputs, float* outputs, int num_samples) {
    #pragma omp parallel for
    for (int i = 0; i < num_samples; i++) {
        outputs[i] = inputs[i] * 2.0f;  // Example: simple doubling operation
    }
}
```

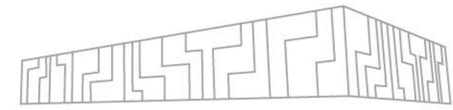
Compiling: Compiles the C code into a shared library that Python can use.

```
gcc -fopenmp -shared -o inference.so -fPIC inference.c
```

Python Code: Loads the shared library, defines argument types, and calls the parallel inference function.

```
import numpy as np
import ctypes

# Load the shared library
inference_lib = ctypes.CDLL('./inference.so')

# Define the function signature
inference_lib.parallel_inference.argtypes = (ctypes.POINTER(ctypes.c_float),
ctypes.POINTER(ctypes.c_float), ctypes.c_int)

def main():
    num_samples = 1000
    inputs = np.arange(num_samples, dtype=np.float32)
    outputs = np.zeros(num_samples, dtype=np.float32)

    # Call the OpenMP function
    inference_lib.parallel_inference(inputs.ctypes.data_as(ctypes.POINTER(ctypes.c_floa
t)),
                                     outputs.ctypes.data_as(ctypes.POINTER(ctypes.c_float)),
                                     num_samples)

    # Print a few results
    for i in range(10):
        print(f"Input: {inputs[i]}, Output: {outputs[i]}")

if __name__ == "__main__":
    main()
```

PARALLEL INFERENCE



- Parallelize Inference: Implement concurrent.futures for parallel inference.
- Each case runs independently, utilizing multiple CPU cores.

```
from concurrent.futures import ProcessPoolExecutor

# Run inference on all cases in parallel
with ProcessPoolExecutor() as executor:
    executor.map(run_inference, case_data_list)
```

- Time taken:

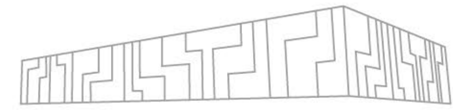
1

- **Time without parallelization on one process (inference and saving all slices of all patients)**
  - **~30 minutes**

2

- **Time with parallelization on 4 processes (inference and saving all slices of all patients)**
  - **~10 minutes**

MPI



Distribute training workload across multiple processors.

Install mpi4py:

- conda install -c conda-forge mpi4py mpi
- ml av mpi4py

MPI Initialization

```
from mpi4py import MPI

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()  # Unique ID for each process
size = comm.Get_size()  # Total number of processes
```

Data Distribution

```
# Split data among processes
def split_data(data, num_splits):
    return np.array_split(data, num_splits)

local_data = split_data(dataset, size)[rank]  # Each process
gets a part of the dataset
```

Training Loop

```
for epoch in range(epochs):
    model.train()
    # Assume local_data is available and batch_size is defined
    for batch in local_data:
        optimizer.zero_grad()
        output = model(batch)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
```

Weight Gathering and Averaging

```
local_weights = model.state_dict()
gathered_weights = comm.gather(local_weights, root=0)

if rank == 0:
    # Average weights
    averaged_weights = {key:
torch.mean(torch.stack([weights[key] for weights in
gathered_weights]), dim=0) for key in local_weights.keys()}
averaged_weights = comm.bcast(averaged_weights,
root=0)  # Broadcast averaged weights to all processes
```

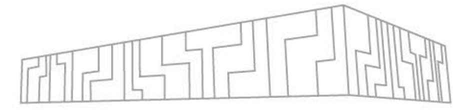
Testing the Model

```
if rank == 0:
    # Perform evaluation after training
    test_accuracy = evaluate_model(model,
test_data)
    print(f"Test Accuracy:
{test_accuracy:.2f}%")
```

Running the MPI Program

```
salloc -A OPEN-28-64 -p qgpu --ntasks-per-
node=4 --nodes=1 -t 00:30:00

conda activate /path_to_conda_env

mpiexec -n 4 python mpi_python.py
```

MODEL PARALLELISM



Performing model parallelism on the Karolina cluster involves distributing different parts of a machine learning model across multiple GPUs or nodes. This can help you handle larger models than can fit in the memory of a single GPU. Different strategies:

- Layer-wise Model Parallelism: Different layers of the model are placed on different devices.
- Shard-wise Model Parallelism: Different shards or components of the model are distributed across devices.

PyTorch Layer-wise Model Parallelism (Part 1)

```
import torch
import torch.nn as nn
import torch.distributed as dist

# Initialize the process group
dist.init_process_group(backend='nccl')

# Define a simple model
class ModelPart1(nn.Module):
    def __init__(self):
        super(ModelPart1, self).__init__()
        self.layer1 = nn.Linear(10, 10)

    def forward(self, x):
        return self.layer1(x)

class ModelPart2(nn.Module):
    def __init__(self):
        super(ModelPart2, self).__init__()
        self.layer2 = nn.Linear(10, 1)

    def forward(self, x):
        return self.layer2(x)
```

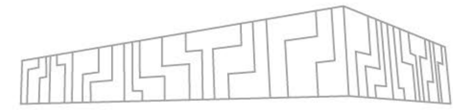
PyTorch Layer-wise Model Parallelism (Part 2)

```
# Instantiate the models
model1 = ModelPart1().cuda(0)  # First part on GPU 0
model2 = ModelPart2().cuda(1)  # Second part on GPU 1

# Example input
input_tensor = torch.randn(1, 10).cuda(0)

# Forward pass through the first part
output1 = model1(input_tensor)

# Send output to the second part on GPU 1
output1 = output1.cuda(1)

# Forward pass through the second part
output2 = model2(output1)

# Final output
print(output2)
```

Running the Code

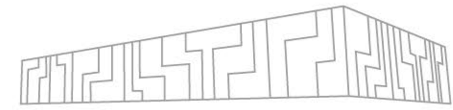
```
srun --ntasks=2 --gpus=2 python your_script.py
```

DEALING WITH LARGE DATASETS



- Chunked Data Loading
  - Load the dataset in smaller chunks into CPU memory to optimize resource usage.
- Data Transfer to GPU
  - Move data to GPU in batches, ensuring efficient processing without memory overflow.
- Memory Management
  - Free up memory after processing each batch; use `torch.cuda.empty_cache()` to clear GPU memory.
- Data Generators
  - Utilize built-in generators or data loaders for on-the-fly loading of data, enhancing flexibility.
- DataLoader Usage
  - The `DataLoader` iterates through the dataset, loading specified batch sizes for training.
- Shared Storage Approach
  - Use shared storage solutions (e.g., databases or large files) to allow GPUs to access only the necessary data chunks, minimizing CPU memory usage.
- Distributed Sampler
  - Employ PyTorch's `DistributedSampler` with `DataLoader` to ensure each GPU receives a unique subset of the dataset, avoiding CPU memory duplication.
- Memory-Mapped Files
  - Utilize libraries like NumPy to memory-map large arrays, loading only the required parts into memory.
- Data Sharding
  - Split the dataset into smaller shards, allowing each process or GPU to load only its assigned shard.
- On-Demand Loading
  - Implement logic to load only the necessary parts of the dataset for the current training step or epoch.

INFRASTRUCTURE MONITORING



- To monitor CPU usage interactively
  - top
  - htop
- Monitoring GPU Usage
  - nvidia-smi
  - watch -n 1 nvidia-smi
- Checking Available Partitions
  - sinfo
- Check the status of your jobs:
  - squeue --me
- View the history of completed jobs
  - sacct
- Cancel a job:
  - scancel <job_id>
- Viewing Job Details:
  - scontrol show job <job_id>
- View Resource Usage by User
  - sreport user top usage
- Show Detailed Information About a Node
  - scontrol show node <node_name>
- View Cluster Utilization by User (for a specific date range)
  - sreport cluster UtilizationByUser start=2024-09-01 end=2024-09-30
- View Job Resource Usage Statistics
  - sstat --format=JobID,MaxRSS,Elapsed,AveCPU
- Get Resource Usage Details for a Specific Job
  - sstat -j <job_id> -o "JobID,MaxRSS,MaxVMSize"
- Get Efficiency Summary for a Job
  - seff <job_id>
- View Accounting Data for a Specific User
  - sacct -u <username>
- View Historical Job Details for a Date Range
  - sacct --starttime=2024-09-01 --endtime=2024-09-30
- Display Job Summary with Key Information
  - sacct -o JobID,User,Partition,AllocCPUs,Elapsed,State

Khyati Sethia
khyati.sethia@vsb.cz

IT4Innovations National Supercomputing Center
VSB – Technical University of Ostrava
Studentská 6231/1B
708 00 Ostrava-Poruba, Czech Republic
www.it4i.cz

VSB TECHNICAL UNIVERSITY OF OSTRAVA | IT4INNOVATIONS NATIONAL SUPERCOMPUTING CENTER