Practical 2 - Running AI and HPC application with and without UVM

Useful shortcut

Storage

- `${SCRATCH}` : scratch area
- `${PROJECTDIR}` : common area where are stored pre-built SIF containers and spack environment

Useful alias

Start an interactive session on a full node:

```
alias interactive_node="srun --gpus=4 --time=00:15:00 --exclusive --reservatio
```

Access NVIDIA Nsight System without loading a full module (useful to avoid version conflicts):

```
alias my_nsys="/opt/nvidia/hpc_sdk/Linux_aarch64/24.11/profilers/Nsight_System
```

Useful commands to explore node capabilities

- `lscpu` : tells CPU model
- `numactl` : tells aboutn UMA topology
- `nvidia-smi -m topo` : tells about GPU-GPU and CPU-CPU connectivity
- `nvidia-smi -q -d POWER` : tells about GH200 Superchip power capabilities (min, max, current, capped)
- `nvidia-smi -g 0 -q -d POWER` : tells about power capabilities (min, max, current, capped) of a specific GH200 Superchip (GPU 0).
- `perf list` : tells CPU hardware performance counters
- `ml load papi/7.2.0.1 && papi_avail` : tells CPU hardware performance counters via the common PAPI interface

[01] `uvm_cuda`

Compile and run code samples that illustrate managed vs unified memory capabilities. All examples are very simple, very fast and single GPU.

Load the environment:

```
ml load cudatoolkit/24.11_12.6
```

To compile (where X is {1, 2, 3, 4, 5}):

```
nvcc -arch=sm_90 exX.cu -o exX.x
```

The examples cover:
* `ex1` : CPU reference code
* `ex2` : GPU code with explicit GPU memory copies
* `ex3` : GPU code with use of managed GPU
* `ex4` : GPU code which leveraged GPU unified memory and host dynamic allocations
* `ex5` : GPU code which leveraged GPU unified memory and mix both static and dynamic allocations

NOTE - GH200 is equipped with the same Hopper GPU architecture present in H100/H200 SXM or PCie form factors. The compute capability is **90**.

[02] `uvm_pytorch`

[02.a] Prepare PyTorch environment

Bootstrap a baremetal Python environment

Start a new python virtual environment:

```
ml load cray-python/3.11.7
python3 -m venv $SCRATCH/pytorch_baremetal
source $SCRATCH/pytorch_baremetal/bin/activate
```

Install PyTorch and using **pip**:

```
pip install torch==2.6.0 torchvision==0.21.0 torchaudio==2.6.0 --index-url htt
```

Prepare PyTorch containers

To avoid clogging `$HOME` with temporary files, export the following:

```
rm -rf $HOME/.apptainer/cache
mkdir -p ${SCRATCH}/.local_singularity/tmp
mkdir -p ${SCRATCH}/.local_singularity/cache
export APPTAINER_TMPDIR=${SCRATCH}/.local_singularity/tmp
export APPTAINER_CACHEDIR=${SCRATCH}/.local_singularity/cache
```

To fetch a container from NGC (NVIDIA GPU Container registry), run

```
singularity pull ${SCRATCH}/pytorch_25.08-py3.sif docker://nvcr.io/nvidia/pyto
```

NOTE - Not every PyTorch version is available via pip wheel for aarch64. You can check on the PyTorch website for a compatibility match. If there is no perfect match, my advise is to pick the closer lower bound to the system one (e.g. system has 12.5, better pick 12.4 or 12.1 instead of 12.6). Forward and backward compatibility boundaries across major / minor CUDA releases is documented on the CUDA SDK Documentation.

[02.b] Verify PyTorch is working correctly

Running baremetal

Grab an interactive node. Load the previously built python virtual environment:

```
ml load cray-python/3.11.7
source $SCRATCH/pytorch_baremetal/bin/activate
```

Start python and check if PyTorch detects the GPU. Here an example:

```
(pytorch_baremetal) fspiga.t5c@nid010435:~> python
Python 3.11.7 (main, Jun 17 2024, 15:39:30) [GCC 12.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
>>> torch.cuda.device_count()
4
>>> torch.cuda.current_device()
0
>>> torch.cuda.get_device_name(0)
'NVIDIA GH200 120GB'
```

Prepare PyTorch containers

Grab an interactive node. Start singularity in interactive mode:

```
export CONT=${PROJECTDIR}/pytorch_25.08-py3.sif
singularity run --nv "${CONT}"
```

Start python and check if PyTorch detects the GPU. Here an example:

```
Apptainer> python
Python 3.12.3 (main, Jun 18 2025, 17:59:45) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> torch.cuda.is_available()
True
>>> torch.cuda.device_count()
4
>>> torch.cuda.current_device()
0
>>> torch.cuda.get_device_name(0)
'NVIDIA GH200 120GB'
```

[02.c] Running simple PyTorch model (MNIST)

We will use MNIST from PyTorch public examples. For convenience, we have copied the script locally.

Copy the script:

```
cp -R ${PROJECTDIR}/mnist ${SCRATCH}/
```

NOTE - The directory contains the python script that define the Neurtal Network and orchestrate the trauining process. During first execution the full dataset will be automatically downloaded.

Running baremetal

Load the previously built python virtual environment:

```
ml load cray-python/3.11.7
source $SCRATCH/pytorch_baremetal/bin/activate
```

Run:

```
cd ${SCRATCH}/mnist
CUDA_VISIBLE_DEVICES=0 python main.py
```

Running PyTorch containers

Grab an interactive node. Run baremetal using only 1 GPU:

```
cd ${SCRATCH}/mnist
export CONT=${PROJECTDIR}/pytorch_25.08-py3.sif
singularity run --nv -B ${PWD}:/host_pwd --pwd /host_pwd "${CONT}" python main
```

[02.d] Add transparent UVM capabilities

TASK: Prepare the environment, modify the sourcer and run.

NOTE - Additional packages must be installed baremetal to access UVM capabilities via RAPIDS `rmm` module. The container has already all necessary files.

Add extra packages required for RAPIDS `rnn` module to be used:

```
pip install \
  --extra-index-url=https://pypi.nvidia.com \
  cudf-cu12==25.8.* \
  dask-cudf-cu12==25.8.* \
  cuml-cu12==25.8.* \
  cugraph-cu12==25.8.*
```

NOTE - This process can take up to 15 minutes.

Add the following in the approproate location:

```
import rmm
from rmm.allocators.torch import rmm_torch_allocator
rmm.reinitialize(pool_allocator=True, managed_memory=True)
torch.cuda.memory.change_current_allocator(rmm_torch_allocator)
```

[02.a] Add NVTX tags

TASK: Modify the source to add NVTX, tags isolate compute phases ("train", "test", "epoch") and visualize with nsys.

A new import is needed:

```
from torch.cuda import nvtx
```

To start a NVTX region, do the following:

```
nvtx.range_push("MY LABEL")
```

To close a NVTX region, do the following:

```
nvtx.range_pop()
```

NOTE - NVTX regions can be nested, important to maintain consistency in the use of push/pop pairs.

To run Nsight System profiler and capture the result, run:

```
my_nsys profile --trace=cuda,cudnn,nvtx python main_uvm_nvtx.py
```

NOTE - Copy your `.nsys-rep` file on your local machine and visualize it. Download and install Nsight System GUI from the official website.

[03] `hpc_gromacs`

TASK-1: Run successfully GROMACS with one or more GPUs, interactively or via batch script

NOTE - Tune GROMACS for optimal execution on GH20 is not in the scope of this tutorial.

[03.a] Water (48k atoms)

Requires GROMACS 2025.1 (container:= `${PROJECTDIR}/gromacs-2025.1.sif`).

Input is provided in `practical2/hpc_gromacs`

Execution is made in two steps:
* `gmx grompp` is the GROMACS pre-processor. It takes several input files (including a parameter file, coordinate file, and topology file) and combines them to produce a .tpr binary file, which contains all the initial information needed to start an MD simulation.
* `gmx mdrun` that starts the actual simulation. All parameters requird are reported in the `job.sh` example script.

By default this input runs 4 MPI and `${OMP_NUM_THREADS}` threads. This can be changes by carefully modifying `-ntmpi` and `-ntomp` command line parameters.

Performance is measured in `ns/day` and can be extracted with the following command:

```
grep -Ir "Performance:" md.log
```

[03.b] STMV (1M atoms)

Requires GROMACS 2023.2 (container `${PROJECTDIR}/gromacs-2023.2.sif`).

Copy input from `${PROJECTDIR}/gromacs_example/stmv` into `${SCRATCH}` .

No need of prepropcessing, simulation is ready to go.

To run on a single GPU (which will be shared among processes):

```
export GMX_ENABLE_DIRECT_GPU_COMM=1
export CONT=${SCRATCH}/gromacs-2023.2.sif
singularity run --nv -B ${PWD}:/host_pwd --pwd /host_pwd "${CONT}" gmx mdrun -
```

To run on all 4 GPUs:

```
export GMX_ENABLE_DIRECT_GPU_COMM=1
export CONT=${SCRATCH}/gromacs-2023.2.sif
singularity run --nv -B ${PWD}:/host_pwd --pwd /host_pwd "${CONT}" gmx mdrun -
```

Performance is measured in `ns/day` and can be extracted with the following command:

```
grep -Ir "Performance:" md.log
```

[04] Build `nvbandwidth` from source

To build `nvbandwidth` tool from source, Boost is required. CUDA SDK is already installed as module on the system.

Here a series of steps to concretize a spack environment in `$SCRATCH`:

```
cd ${SCRATCH}

git clone --depth=2 --branch=releases/v0.23 https://github.com/spack/spack.git
. spack/share/spack/setup-env.sh

git clone --depth 1 --branch=releases/v1.3 https://github.com/isambard-sc/buil

export SPACK_DISABLE_LOCAL_CONFIG=true

spack env create -d ${SCRATCH}/myenv
spack env activate ${SCRATCH}/myenv

spack config add -f buildit/config/aip2/v0.23/linux/compilers.yaml
spack config add -f buildit/config/aip2/v0.23/packages.yaml
spack config add config:build_jobs:8
spack config add view:true
spack config add concretizer:unify:true
spack config add concretizer:reuse:false

spack repo add ./buildit/repo/v0.23/isamrepo

spack compiler list

spack add boost@1.86.0%gcc@13.3.0 +program_options

spack concretize

spack install
```

Once concretization is completed, unload / load the environment before build `nvbandwidth`:

```
spack env deactivate && spack env activate ${SCRATCH}/myen

export LD_LIBRARY_PATH=${SCRATCH}/myenv/.spack-env/view/lib:$LD_LIBRARY_PATH
```

Follow the instruction on https://github.com/NVIDIA/nvbandwidth to download and build the tool.

Interesting tests worth running

`nvbandwidth` support multiple single and multi-GPU tests, we will use these 4:

```
0, host_to_device_memcpy_ce:
    Host to device CE memcpy using cuMemcpyAsync

1, device_to_host_memcpy_ce:
    Device to host CE memcpy using cuMemcpyAsync

16, host_to_device_memcpy_sm:
    Host to device SM memcpy using a copy kernel

17, device_to_host_memcpy_sm:
    Device to host SM memcpy using a copy kernel
```

Example running on different GPUs , affinity sored out automatically:

```
export CUDA_VISIBLE_DEVICES=0
./nvbandwidth -t host_to_device_memcpy_ce

export CUDA_VISIBLE_DEVICES=1
./nvbandwidth -t host_to_device_memcpy_ce
```

Example running on different GPUs , auto-affinity disabled:

```
export CUDA_VISIBLE_DEVICES=0
./nvbandwidth -d -t host_to_device_memcpy_ce

export CUDA_VISIBLE_DEVICES=1
./nvbandwidth -d -t host_to_device_memcpy_ce
```

Example running on different GPUs , auto-affinity disabled, forcing binding for memory and CPU:

```
export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 0 --membind 0  ./nvbandwidth -d -t host_to_device_memcpy

export CUDA_VISIBLE_DEVICES=1
numactl --cpunodebind 0 --membind 0 ./nvbandwidth -d -t host_to_device_memcpy_

export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 1 --membind 1 ./nvbandwidth -d -t host_to_device_memcpy_

export CUDA_VISIBLE_DEVICES=1
numactl --cpunodebind 1 --membind 1 ./nvbandwidth -d -t host_to_device_memcpy_
```

Example running with proper affinity used, 2 different approaches:

```
export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 0 --membind 0  ./nvbandwidth -d -t host_to_device_memcpy

export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 0 --membind 0  ./nvbandwidth -d -t device_to_host_memcpy

export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 0 --membind 0  ./nvbandwidth -d -t host_to_device_memcpy

export CUDA_VISIBLE_DEVICES=0
numactl --cpunodebind 0 --membind 0  ./nvbandwidth -d -t device_to_host_memcpy
```