Lattice Practices 2024 GPU Computing Tutorial

Cluster setup

```
module load CUDA/11.8.0
module load OpenMPI/4.1.4-GCC-11.3.0
module load CMake/3.24.3-GCCcore-11.3.0
module load Ninja/1.10.2-GCCcore-11.3.0
export LD_LIBRARY_PATH+=:/usr/lib64/
```

and if you try the C++ standard parallelism

```
module load NVHPC/23.7-CUDA-12.1.1
```

To run a job on a A100 GPU please prefix your command with

```
srun -N1 -p a100 --gpus=1 --reservation=short
```

Saxpy tutorial (Wednesday)

The saxpy directory contains 3 tasks showing 3 different ways how to calculate `saxpy` on a GPU.

- **task0**: Use cublas to
- **task1**: Use CUDA to parallelize saxpy
- **task2**: Use C++ stdpar to parallelize saxpy (*Note*: This needs the nvhpc module)

For all 3 examples there are `TODO` annotations in the source **and** Makefile. Please take a look at these files to find these and the tasks that are associated with them.

Jacobi tutorial (Wednesday)

This part starts with a more advanced example, a 2d Jacobi solver. The first part is to parallelize the CPU code with CUDA and explore multiple options for parallelization. This exploration should be guided

Task 0: Parallelize a CPU Jacobi solver with CUDA

The task here is to demonstrate the workflow steps when porting a code to a GPU. The repeating pattern here is to identify and expose the problem, measure its performance, use the tools to understand the performance and then try to improve on that.

There are some `TODO` markers in the code to get you started on creating an initial CUDA version. From there you should measure the code performance with NSight Systems and NSight Compute and just spent some time to play with the tools and the code. Some suggestions for different parallelization strategies are:

- Parallelize the `iy` loop using CUDA. Leave the `ix` loop unchanged.
- Experiment with using 1 thread, 1 thread block, multiple blocks with 1 thread and multiple threads and blocks. It might be useful to use a grid-strided loop
- Instead of the `iy` loop parallelize the `ix` loop.
- Use a 2d block and parallelize both the `ix` and `iy` loop.
- Does it matter how you map `ix` and `iy` to `threadIdx.x` and `threadIdx.y`

Note: Calculating the l2 norm will need some special care. If you allocate the l2 norm in managed memory you can use `atomicAdd`, see Atomic Functions in the CUDA programming guide. A more advanced way would be the use of a reduction algorithm as shown in the Multi-GPU version. For the comparison of the timing of the different versions of the kernel it is best to skip the calculation of the norm in this simple way, as it being located in managed memory accessed by the CPU and GPU in each iteration this causes a lot of page faults and memory migrations that can overshadow the effects we want to measure here.

Task 0b: Explore the output with Nsight Tools (Thursday)

A way to measure the code with Nsight Systems could look like:

```
nsys profile --trace=cuda,nvtx --stats=true -o jacobi.nsys-rep
./jacobi_gpu
```

You can also skip the `-o` option to get just the stats on the command line.

If you profile multiple versions please remember to change the output filename accordingly. You should also keep track of the exact code version that was used for a particular profile.

Profiling the kernel with Nsight Compute could look like

```
ncu ncu --set detailed -k  jacobi_kernel  -c 1 -s 1 -o jacobi.ncu-rep
./jacobi_gpu
```

The GUI allows you to compare different versions using a baseline.

Again, if you skip the `-o` option you can also see the most important output on the command line.

Task 1: Parallelize Jacobi Solver for Multiple GPUs using CUDA-aware MPI (Wednesday/Thursday)

The task is to use CUDA-aware MPI to parallelize a Jacobi solver for multiple GPUS. The starting point of this task is a skeleton `jacobi.cu`, in which the CUDA kernel is already defined and also some basic setup functions are present. There is also a single-GPU version with which the performance and numerical results are compared. Take some time to get familiar with the code. Some functions (like NVTX) will be explained in next sessions. They can be ignored for now (e.g. the `PUSH` and `POP` macros). Once you are familiar with the code, please work on the `TODOs` in `jacobi.cu`:

- Get the available GPU devices and use it and the local rank to set the active GPU for each process

- Compute the top and bottom neighbors. We are using reflecting/periodic boundaries on top and bottom, so rank0's Top neighbor is (size-1) and rank(size-1) bottom neighbor is rank 0
- Use `MPI_Sendrecv` to exchange data between the neighbors
    - use CUDA-aware MPI, so the send - and the receive buffers are located in GPU-memory
    - The first newly calculated row (`iy_start`) is sent to the top neighbor and the bottom boundary row (`iy_end`) is received from the bottom process.
    - The last calculated row (`iy_end-1`) is send to the bottom process and the top boundary (`0`) is received from the top
    - Don't forget to synchronize the computation on the GPU before starting the data transfer
    - use the self-defined MPI_REAL_TYPE. This allows an easy switch between single- and double precision

Note: To request more than 1 GPU use e.g.

```
srun -n2 --gres=gpu:2
```

Task 1b: Optimize Load Balancing

The work distribution of the first task is not ideal, because it can lead to the process with the last rank having to calculate significantly more than all the others. Therefore, the load distribution is to be optimized in this task.

- Compute the `chunk_size` that each rank gets either `(ny - 2) / size` or `(ny - 2) / size + 1` rows.
- Compute how many processes get `(ny - 2) / size` resp. `(ny - 2) / size + 1` rows
- Adapt the computation of (`iy_start_global`)

QUDA / MILC (Wednesday)

The instructions for this part are available in the slides `LatticePractices2024-QUDA`. Input lattices are located in