README

December 6, 2023

1 Exercise 6 - Distributed Computing on GPUs

1.1 Introduction

In this exercise, you will implement functions to find the maximum integer in an array distributed across multiple GPUs (devices).

The exercise consists of three subtasks:

1. Implement the functions `reduce_kernel` and `reduce` in `src/reduce-single.cu`. Your task here is to find the maximum integer in an array allocated on a single device and return the result with `reduce`.
2. Extend your implementation to work on multiple devices on a single node in `src/reduce-multi.cu`. The functions to implement are `reduce_kernel` (you can re-use your solution from subtask (1) and `reduce`. The array is now distributed across multiple devices and each device should compute a single result. You can perform the final reduction step on the host by transferring the data with the device-to-host variant of `cudaMemcpy`. In your slurm command, you should allocate a single MPI task (`--ntasks 1`) but multiple devices (`--gpus-per-node n`).
3. Extend your implementation to work on multiple devices on multiple nodes using MPI in `src/reduce-mpi.cu`. The instructions are otherwise the same as in subtask (2), however, in this exercise you should allocate one process per device (for example `--ntasks-per-node 4 --nodes 2 --gpus-per-node 4`) and ensure your implementation works on multiple nodes.

   Note: The performance of your implementation is not graded and to get full points, you only need to ensure your implementations give the correct results. However, you should use the hardware relatively efficiently: an implementation that uses a single CUDA thread in task 1, or a single device in tasks 2 or 3, will receive 0 points.

1.2 Returnables

The solutions should be returned in a single zip file named `<your student number>.zip`, for example `12345.zip`. The archive should contain at least `src/reduce-mpi.cu`, `src/reduce-multi.cu`, and `src/reduce-single.cu`. You create the archive with the command `zip <your student number>.zip -r src/` (note the `-r` flag: the archive must contain the `src` directory).

1.3 Getting started

Run the following commands to get started:

```
module load gcc cuda cmake openmpi
git clone https://version.aalto.fi/gitlab/manterm1/pps-example-codes.git # HTTPS
# git clone git@version.aalto.fi:manterm1/pps-example-codes.git # SSH
cd pps-example-codes/GPU/sheet6
mkdir build && cd build
cmake .. && make -j
cd .. && make yourrundir && cd yourrundir
sbatch ../job_scripts/run-X.sh
```

Three binaries should appear corresponding to tasks 1, 2, and 3: `reduce-single`, `reduce-multi`, `reduce-mpi`.

The programs can then be run with corresponding job scripts in `sheet6/job-scripts`. Note! As they do not do anything sensible yet, then the job will terminate very quickly after the first failure is detected.

1.4 Hints

- See `src/main.cu` for the CPU model solution used for grading. Your implementation should give the same output.
- Only the interface function `int reduce(const int* arr, const size_t count)` declared in `src/reduce.cuh` is used for grading. You can add additional helper functions if needed.
- Remember to use `cudaSetDevice` to select the correct device in task 2 and 3.
- If your implementation is very slow, recall how GPUs differ from CPUs (lecture slides). Do not try to create a `for` loop inside the kernel that loops over all `n` elements. In addition, you should strive to avoid branch divergence and ensure that the workload is distributed evenly across the stream processors (CUDA cores).
- For an introduction to parallel reductions and hints on more advanced optimization techniques, see NVIDIA's slides on parallel reductions.