GPU Programming Assignment

Cristian Di Pietrantonio, Maciej Cytowski

May 10, 2021

In this assignment you are asked to write programs that run on GPUs using CUDA and chosen directive-based GPU programming model (OpenACC or OpenMP).

1 Preliminaries

1.1 Using the GPU cluster

Pawsey provides the Topaz GPU cluster consisting of

- 22 compute nodes with 2 V100 GPUs each, distributed across the `gpuq` and `gpuq-dev` Slurm partitions; and
- 11 compute nodes with 4 P100 GPUs each, in the `nvlinkq` and `nvlinkq-dev` Slurm partitions.

You should be using the first set of compute nodes, because the ones in `nvlinkq` are configured for exclusive usage; that is, once you are granted access to the node, no one else can be. If you only use one GPU, the remaining three are left idle!

To facilitate course activities, a Slurm *reservation* has been created, called `courses0100`, that grants course attendees exclusive access to 2 `gpuq` nodes, for a total of 4 GPUs. While it is impossible for all the course attendees to use them at the same time, you can still submit jobs to `gpuq` and `gpuq-dev` without reservation, using other nodes as they become available (which should not take much time). If you encounter an issue, write to the mailing list.

The login node of Topaz is available at `topaz.pawsey.org.au`; username and project code are the same as the ones used in previous lectures. Use any SSH client you prefer.

1.2 Assignment structure

The assignment archive, `gpu-assignment.zip`, is structured as follow:

- `assignment.pdf`: this file, containing the description of the assignment;
- `ex1-gol-cuda/`: directory for the first exercise;
- `ex2-gol-gpu-directives/`: directory for the second exercise;

2 Exercise 1: the game of life using CUDA (0.5 points total)

This exercise will make you familiar with the process of adapting an existing serial CPU code to make it run on a GPU. This process is often called *porting*. In particular, the program implements The Game of Life, which was described and used in the OpenMP assignment.

For this exercise, you will work in the `ex1-gol-cuda` directory. It contains

- a `Makefile` file to assist you in compiling both the CPU code and the GPU one (that does not exist yet);
- a `src/` directory to collect source files for this exercise. Three files are already there that implement the CPU program;
- a `LICENSE` file, containing the license which this material is distributed under.

The first thing to try is compiling the CPU code. To do so, execute the `make` command. It will generate the executable `bin/game_of_life`, but also prints an error regarding the `src/game_of_life_cuda.cu` file. It happens because the make tries to build the source code for the GPU version, but you have not created it yet. So, to start with the assignment, create the missing file. It is easier if you start working with a copy of the serial version instead of an empty file: `cp src/game_of_life.c src/game_of_life_cuda.cu`.

2.1 Task 1.1 - GPU porting (0.35 points)

Adapt the serial version of the code to run the most computational intensive part on a GPU using CUDA C. You DON'T HAVE TO port code in `common.*`. Evaluation criteria include:

- correctness: most of the computation runs on GPU and produces the correct result. To check this, you should use the `cpu_game_of_life` function after you have added at the top of the `src/game_of_life_cuda.cu` file the following lines:
  ```
  #define INCLUDE_CPU_VERSION
  #include "game_of_life.c"
  ```

- code safety: the code reports if something went wrong so that the user knows the output is not valid.

2.2 Task 1.2 - performance evaluation (0.15 points)

The *speedup* measure compares the execution time of a parallel code against its serial implementation. It is defined as $S = \frac{T_s}{T_p}$, where T_p is the execution time of the parallel code and T_s is the execution time of the serial version. The greater S, the better. In this task you are asked to compute the speedup of your GPU code against the original one, for inputs of various size.

To do so, you must measure execution times first. The serial code has already timers in place; on the GPU code you can measure the total execution time (actual GPU computation plus any other CUDA API call and cpu code to support it) in a similar way. it is extremely interesting and useful to compute the percentage of the total execution time taken by just the overall CUDA kernel execution time; again, as the input size increases.

Fix the number of steps to 100. Then, run both the CPU and GPU implementations of The Game of Life for (at least) the following inputs:

1. $n = m = 10$
2. $n = m = 100$
3. $n = m = 1000$
4. $n = m = 10000$

For each of the above inputs, report:

1. T_s, the execution time of the CPU code,
2. T_p, the execution time of the GPU implementation,
3. T_s/T_p, the speedup,
4. t_k, the percentage of T_p spent executing only kernel code.

You should provide the information in a tabular format, in a text file named `performance.txt`, where each row presents the requested information for a given input. You are encouraged (although not mandatory) to also plot a curve for each measure in function of the input size.

2.3 What to submit

Submit the whole `ex1-gol-cuda` directory containing all the modified files, that is, `src/game_of_life_cuda.cu` and the `performance.txt` text file providing measurements of performance indicators.

3 Exercise 2: the game of life using directive based GPU programming models (0.5 points total)

This exercise will make you familiar with the process of adapting an existing serial CPU code to make it run on a GPU with directive based GPU programming models OpenACC and OpenMP. This process is often called *porting*. In particular, the program implements The Game of Life, which was described and used in the OpenMP CPU assignment.

For this exercise, you will work in the `ex2-gol-gpu-directives/` directory. It contains two directories `openacc/` and `openmp/` each with the following structure:

- a `Makefile` file to assist you in compiling both the CPU code and the GPU one (that does not exist yet) both for C and Fortran;
- a `src/` directory to collect source files for this exercise.;
- a `LICENSE` file, containing the license which this material is distributed under.

For this exercise you will need to choose between the following options:

- C + OpenACC implementation
- C + OpenMP implementation
- Fortran + OpenACC implementation
- Fortran + OpenMP implementation

Please focus on implementation and optimisation only for single chosen option from the above list.

To compile the project type `make` command. It will require an appropriate compiler module, i.e.:

- use `pgi/19.7` module for OpenACC solutions,
- use `gcc/10.2.0` module for OpenMP solutions.

The `make` command will compile and build CPU and GPU versions, although the GPU versions initially do not contain any directives.

To start the assignment edit appropriate GPU source file, e.g. if you have chosen to work on Fortran + OpenMP implementation edit

$$openmp/src/02\_gol\_gpu\_openmp\_fort.f90$$

file, compile, execute and compare performance.

3.1 Task 2.1 - GPU porting (0.35 points)

Adapt the serial version of the code to run the most computational intensive part on a GPU using OpenACC or OpenMP. You DON'T HAVE TO port code in `common.*`. Evaluation criteria include correctness and portability. The code changes should be minimal and the GPU code should also execute correctly on CPUs when compiled without directives support.

For data transfers, please use `acc enter data`, `acc exit data` and `omp target enter data`, `omp target exit data`. They have the similar clauses as regular `data` directives, but allow you to implement data movement for the remainder of the program, or until a mathcing `exit data` directive deallocates the data. The regular data directives work only for a structured block of the program. With enter, exit data directives you can schedule data transfers from various functions (not only the one containing the GPU kernel).

For parallel loops, please make sure that all arrays used in GPU kernels have appropriate data types, some of the arrays need to be marked as `private` for performance and correctness.

3.2 Task 2.2 - performance evaluation (0.15 points)

The *speedup* measure compares the execution time of a parallel code against its serial implementation. It is defined as $S = \frac{T_s}{T_p}$, where T_p is the execution time of the parallel code and T_s is the execution time of the serial version. The greater S, the better. In this task you are asked to compute the speedup of your GPU code against the original one, for inputs of various size.

To do so, you must measure execution times first. The serial code has already timers in place.

Fix the number of steps to 100. Then, run both the CPU and GPU implementations of The Game of Life for (at least) the following inputs:

1. $n = m = 10$
2. $n = m = 100$
3. $n = m = 1000$
4. $n = m = 10000$

For each of the above inputs, report:

1. T_s, the execution time of the CPU code,
2. T_p, the execution time of the GPU implementation,
3. T_s/T_p, the speedup.

You should provide the information in a tabular format, in a text file named `performance.txt`, where each row presents the requested information for a given input. You are encouraged (although not mandatory) to also plot a curve for each measure in function of the input size.

3.3 What to submit

Submit the whole `ex2-gol-gpu-directives` directory containing all the modified files and the `performance.txt` text file providing measurements of performance indicators.