Assignment 2
CUDA Implementation of Game of Invasions

CS3210 – 2021/22 Semester 1

7 October 2021

Learning Outcomes

This assignment lets you explore the intricacies of building a parallel application using NVIDIA CUDA for a problem you are already familiar with.

1 Problem Scenario

In this assignment, you will re-implement Game of Invasions described in Assignment 1 in CUDA.

1.1 Simulation Rules

The simulation rules are exactly the same as in Assignment 1. Refer to Assignment 1 write-up for further details.

1.2 Inputs and Outputs

Your program should accept **eight** command-line arguments.

- **The first** is a path to an input file with the simulation parameters. The program should then run the simulation, and compute the death toll due to fighting.
- Output a single integer representing the death toll due to fighting to the path specified by **the second** command-line argument.
- The **third to eighth** arguments specify the grid and block sizes that the program will run with in the following order: `GRID_X`, `GRID_Y`, `GRID_Z`, `BLOCK_X`, `BLOCK_Y`, `BLOCK_Z`.

The formats and constraints of the input and output files are the same as in Assignment 1, with one exception: **please remove all prints to `stdout` and `stderr` in your submission**.

Sample Program Execution

```
$ ./goi_cuda sample_input.in output.out 1 2 3 4 5 6
```

Explanation of Command-line Arguments

`goi_cuda` will read simulation parameters from `sample_input.in` with grid dimensions `1x2x3` and block dimensions `4x5x6` and output the death toll due to fighting to `output.out`.

1.3 Starter Code

We provide some utility functions and example usage code to export world states for use in the GOI visualizer. The code structure is shown in Table 1.

Files/Folders	Description
`check_zip.sh`	Script to check that your archive follows the required structure.
`exporter.cu` `exporter.h`	These files contain the library we wrote to export world states to a format that the GOI visualizer can understand. As usual, feel free to use, ignore or delete these files as long as your program follows specifications. You will not receive credit for modifications to these files.
`export_example.cu`	This file shows example usage of the `exporter` module in a "CUDA" program.
`Makefile`	Contains one recipe `example` to build `export_example`.
`README`	Contains information about how to use the `exporter` module with CUDA. Feel free to delete after reading, like in a spy movie.
`sb/`	This folder contains code for a string builder library imported to implement the exporter module. The same rules apply as in `exporter.cu`.
`sample_inputs/` `sample_outputs/`	These folders contain sample input and output files for you to test with, as in assignment 1.

Table 1: Code Structure

1.3.1 GOI Visualizer

The same visualizer application from assignment 1 can be used for assignment 2, and can be found (in the same place) here. As usual, using or even downloading the visualizer is not necessary at all for completion of this assignment.
If you experience compatibility issues or have any feedback/suggestions, email Benedict (benedictkhoo.mw@u.nus.edu).

1.4 Your Task

Your task is to implement a parallel version of Game of Invasions using CUDA. Your parallel implementation should be bug-free, make reasonable effort to minimize memory leaks (i.e. do not forget to free memory you `malloc`) and should run faster than your OpenMP implementation for a large enough world size (otherwise there is no point using CUDA). You will also need to conduct some performance measurements and write a report.

Your parallel implementations should give the same result (output) as your OpenMP implementation (on the machines on the SoC compute cluster), and execute faster for a large enough world size.

1.5 Optimizing your Solution

While correctness is important in a parallel program, improving performance is the reason we parallelize. After implementing a working CUDA program, you should investigate various modifications of the code and how they affect different parallel performance metrics (e.g. speedup). These modifications include, but are not limited to:

- Different block and grid sizes. **Your implementation should work on varying grid and block sizes**.
- Different data/task distribution methods.

Distinguish any alternative implementations you include in your submission clearly from the final parallel implementations to be graded.

2 Admin Issues

2.1 Running your Programs

During development you might use your personal computer (if you have a CUDA-capable GPU) or any of the 14 machines (with one or two GPGPUs each) from the SoC Compute Cluster reserved for CS3210. Their hostnames are: `xgpc0-7` and `xgpd0-7`.

Your code should successfully compile and run on the SoC Compute Cluster nodes mentioned above. Run your correctness tests and performance measurements on these machines.

2.2 Bonus

You may obtain up to 2 bonus marks for the following:

- **up to 2 bonus marks for analyzing different data/task distribution methods**
- **up to 2 bonus marks for speedup contest**: for achieving the best speedup among all CUDA submissions. We will assign in total 4 bonus marks to the class, two marks for obtaining the best speedup on each type of GPU (listed above). If an implementation tops on multiple GPUs only two bonus marks will be allocated to the student(s), and we will consider the next best speedup. Partial marks can be obtained for the second and third best on each GPU.

You can obtain a maximum of 2 bonus marks as bonus for this assignment.

2.3 FAQ

Frequently asked questions (FAQ) received from students for this assignment will be answered here. The most recent questions will be added at the beginning of the file, preceded by the date label. **Check this file before asking your questions.**
If there are any questions regarding the assignment, please post on the LumiNUS forum or email Benedict (benedictkhoo.mw@u.nus.edu) or Brian (e0310531@u.nus.edu).

Useful resources for Assignment 2:

- CUDA Programming Guide
- CUDA `nvprof` Guide

2.4 Submission Instructions

You are advised to work in groups of two students for this assignment (but you are allowed to work independently as well). You may discuss the assignment with others but in the case of plagiarism, both parties will be severely penalized. Cite your references or at least mention them in your report (what you referenced, where it came from, how much you referenced, etc.).
Assignment submission is due on **Mon, 25 October 2021, 11am**. The grades are divided as follows:

- 5 marks – CUDA implementation and a `Makefile` that compiles your implementation when calling `make build`
- 1 marks – the test cases that can be used to reproduce the results from your report
- 4 marks – a report that includes a performance comparison between CUDA and OpenMP implementations, and a description of modifications made.

Your CUDA implementation should:

- Give correct results (output), as specified in Assignment 1. You may use our sample OpenMP implementation found here to check the correctness.
- NOT output anything to `stdout` or `stderr`. If it does, your program may be flagged by the grading script.
- Make reasonable effort to minimize memory leaks (i.e. have a corresponding `free` for each `malloc`)
- Run faster than your OpenMP implementation from assignment 1. Specifically, to obtain full marks for the performance part of the implementation, your CUDA implementation should have a speedup of at least 10x compared to your OpenMP implementation on the SoC Compute Cluster for a world size of 3000×3000 and $10,000$ steps. (example input `sample7.in`, but another test case will be used for grading).

Your report should include:

- A brief description of your program's design and implementation assumptions, if any.
- A brief explanation of the parallel strategy you used in your CUDA implementation, e.g. synchronisation, work distribution, memory usage and layout, etc.
- Any special consideration or implementation detail that you consider non-trivial.
- Details on how to reproduce your results, e.g. inputs, execution time measurement, etc.
- Present and explain graphs showing the execution time and speedup (y-axis) variation with world size, and grid size (x-axis) (fixed input size). Show measurements with graphs showing how the block size/grid size (task granularity) impact on the execution time and speedup.
- Compare your CUDA implementation performance with your OpenMP implementation performance. Use a world size of 3000×3000 and $10,000$ steps.
- A description of the modifications made to your code (from your baseline correct CUDA implementation) and an analysis of their impact on performance.

Tips:

- There could be many variables that contribute to performance, and studying every combination could be highly impractical and time-consuming. A report that investigates two or three variables sensibly, with explanations as to why these variables might affect performance (and are worth investigating) is better than a report that blindly tries every combination of variables. You will be graded more on the quality of your investigations, not so much on the quantity of things tried or even whether your hypothesis turned out to be correct.
- Performance analysis may take longer than expected and/or run into unexpected obstacles (like your test script failing halfway). Start early and test selectively.
- The SoC Compute Cluster machines are shared with the entire class. Please be considerate and do not hog the machines or leave bad programs running indefinitely. Again, start early or you may be contending with everyone else.

There is no minimum or maximum page length for the report. Be **comprehensive**, yet **concise**.

Submit one zip archive named with your student number(s) (A0123456Z.zip - if you worked by yourself, or A0123456Z_A0173456T.zip - if you worked with another student) containing the following files and folders. Only one archive for both students must be submitted if you worked with another student. **Do not add any additional folder structure.**

1. Your C/C++ code for `goi_cuda.cu` and any source or header files needed to build them.
2. `Makefile` with a recipe named `build` that builds your implementation exactly as you intend it to be graded for correctness/performance. Also remember to remove unnecessary print/export statements if you think they will affect correctness/performance. The executable name produced should be `goi_cuda`. Be sure to include **everything** in your submission needed such that when `make build` is run on a SoC Compute Cluster machine, `goi_cuda` is built without issue.
3. Report in PDF format (A0123456Z_A0173456T_report.pdf or A0123456Z_report.pdf).
4. A folder, named `testcases`, containing any additional test cases (input and output) that you might have used.
5. An **optional** folder, named `scripts`, containing any additional scripts you used to measure the execution time and extract data for your report.

Once you have the zip file, you will be able to check it by doing:

```
$ chmod +x ./check_zip.sh
$ ./check_zip.sh A0123456Z_A0173456T.zip (replace with your zip file name)
```

During execution, the script prints if the checks have been successfully conducted, and which checks failed. Successfully passing the checks ensures that we can grade your assignment. You will receive 0.5% simply for having a valid submission file!
Note that for submissions made as a group, only the most recent submission (from any of the students) will be graded, and both students receive that grade. A penalty of 10% per day (out of your grade) will be applied for late submissions.