Contents

List of Figures

List of Tables

1 Overview

The entire code repository can be found at https://github.com/dgsaf/game-of-life-gpu. Conway's Game of Life has been accelerated using GPU programming with two models, CUDA and OpenACC, both using C. The code has been derived from the original code which was provided by Cristian Di Pietrantonio, and Maciej Cytowski. It consists of the following items of interest:

- report/: The directory containing this tex file and its resulting pdf file.
- ex1-gol-cuda/, ex2-gol-gpu-directives/openacc/: The CUDA and OpenACC directories have a similar structure, which consists of:
  - cpu.slurm: A slurm script for submitting CPU jobs on Topaz, for given `n, m, nsteps`. The CPU code timing output, recorded in [ms], is written to `output/timing-cpu.n-<n>.m-<m>.nsteps-<nsteps>.txt`.
  - gpu.slurm: A slurm script for submitting GPU CUDA jobs on Topaz, for given `n, m, nsteps`. The GPU code timing output, recorded in [ms], is written to output/timing-gpu-cuda.n-<n>.m-<m>.nsteps-<nsteps>.txt for the CUDA code, and output/timing-gpu-openacc.n-<n>.m-<m>.nsteps-<nsteps>.txt for the OpenACC code.
  - jobs.sh: A bash script which batches a set of jobs, for both the CPU and the GPU codes, on Topaz, for `nsteps = 100` and `n = m = 1, 2, 4, 8, ..., 16384`.
  - extract.sh: A bash script which, from the jobs batched in jobs.sh, `n = m = 1, 2, 4, 8, ..., 16384`, extracts the timing output `cpu_elapsed_time, cpu_elapsed_time, kernel_time`, calculates `speedup`, and writes this performance evaluation to output/performance.nsteps-<nsteps>.txt.
  - output/performance.txt: A txt file which, after the jobs have been submitted and the timing output extracted, contains for each job `n = m = 1, 2, 4, 8, ..., 16384` the performance characteristics `cpu_elapsed_time, cpu_elapsed_time, speedup, kernel_time`.

2 CPU Code

The original code for both the CUDA and OpenACC models has been modified slightly.

- Minor C formatting changes have been made, although only where the original code was modified - unmodified regions of the code remain unadjusted.
- Debugging macros have been utilised to annotate the code for clarity, and can be compiled away to yield performant code.
- The timing methods in common.c, common.h, have been standardised across the CUDA and OpenACC codes; having originally yielding different return types in each model. The function `float get_elapsed_time(struct timeval start)` now returns the time since `start` was initialised, in [ms], for both codes.
- In common.c, common.h, the ASCII visualisation has been modified in the following ways: it truncates the grid so that even large grids can be partially visualised in the terminal - allowing for easier verification of grid states across codes, it builds the visualisation output in a

buffer string which is the printed in one call to avoid interference from asynchronous terminal behaviour.

It should be noted that the debugging macros are also ported to the GPU codes, and so, are common to the CPU and GPU codes across both models. The debugging macros are shown in Listing 1, and can be called with a format string, and variable number of arguments similar to how `printf()` is called.

```
3  // debug flags
4  // - ‘debug_verbose != 0‘ will annotate, to stderr, the program as it is
5  //   executed
6  // - ‘debug_timing != 0‘ will annotate, to stderr, the timing variables as
7  //   they are calculated
8  // - ‘debug_visual != 0‘ will annotate, to stderr, the ascii visualisation of
9  //   grid variables as they are initialised and updated
10 const int debug_verbose = 1;
11 const int debug_timing = 1;
12 const int debug_visual = 1;
13
14 // verbose macro
15 #define verbose(format, ...)                                    \
16   if (debug_verbose) {                                          \
17     fprintf(stderr, "[verbose] "format"\n", ##__VA_ARGS__); \
18   }
19
20 // timing macro
21 #define timing(format, ...)                                     \
22   if (debug_timing) {                                           \
23     fprintf(stderr, "[timing] "format"\n", ##__VA_ARGS__);  \
24   }
25
26 // visual macro
27 #define visual(current_step, grid, n, m, format, ...)           \
28   if (debug_visual) {                                           \
29     fprintf(stderr, "[visual] "format"\n", ##__VA_ARGS__);  \
30     visualise_ascii(current_step, grid, n, m);                  \
31   }
```

Listing 1: Debugging macros from `ex1-gol-cuda/src/game_of_life.c`. Note that these macros are common to both CPU and GPU codes across both models.

3 GPU Code

For both models, on Topaz, the script `jobs.sh` was used to submit, for each of `n = m = 1, 2, 4, 8, ..., 16384`, pairs of jobs, running the CPU and GPU codes for these values of `n, m`, and with `nsteps = 100`. The CPU and GPU codes were compiled with `-O2`, and with all debugging macros (including timing and visualisation macros) turned off. After every job had been completed, the script `extract.sh` was used to extract the timing data for the entire job set, and calculate the GPU speedup relative to the CPU, to the file `output/performance.nsteps-100.txt`.

3.1 CUDA Code

The performance characteristics of the CUDA code (that is, the extracted timing data) are shown in Table 1 and the timing data is plotted in Figure 1.

n = m	cpu_elapsed_time	gpu_elapsed_time	speedup	kernel_time
1	0.03	240.13	0.0001	1.99
2	0.03	168.12	0.0002	1.14
4	0.05	143.84	0.0004	0.51
8	0.13	149.96	0.0009	0.52
16	0.52	136.58	0.0038	0.56
32	1.17	146.68	0.0080	0.58
64	7.97	132.94	0.0599	0.56
128	16.80	135.33	0.1241	0.60
256	54.00	145.73	0.3706	0.59
512	264.99	131.83	2.0100	0.93
1,024	772.04	135.65	5.6913	1.93
2,048	3,320.08	169.46	19.5919	6.90
4,096	10,886.09	187.74	57.9852	23.89
8,192	79,329.53	377.86	209.9420	94.91
16,384	203,088.16	1,123.27	180.8010	381.04

Table 1: Performance characteristics of CUDA code, compiled with `-O2`, and with debugging statements turned off. All times are presented in units of ms.

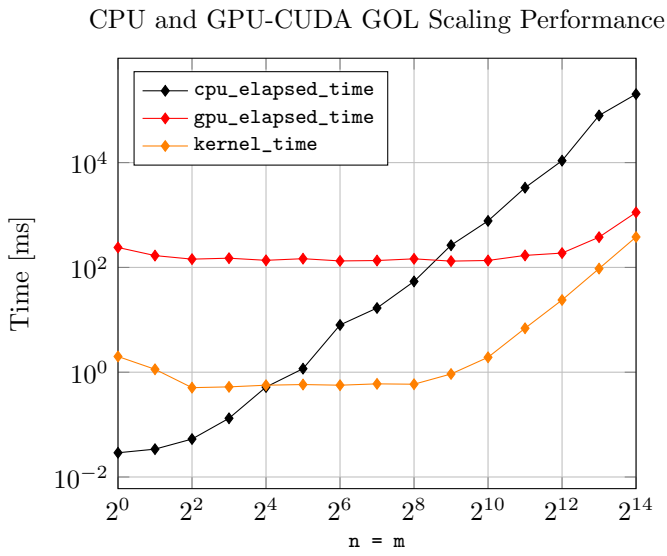

Figure 1: The scaling performance of the CPU and GPU-CUDA GOL codes are shown for `n = m = 1, 2, 4, 8, ..., 16384`*, with* `nsteps = 100`*. Note that the x-axis is presented in* $\log_2$ *scale, and the y-axis is presented in* $\log_{10}$ *scale.*

It can be seen that the CPU code is more performant that the GPU-OpenACC code for small grid sizes up to `n = m = 512`. We note also that is scales exponentially with the grid size, for all grid sizes. It can be seen that OpenACC code is uniformly performant for grid sizes up to `n = m = 4096`, at which point it begins to scale exponentially with the grid size, but at a slower rate than the CPU code. It can also be seen that `kernel_time` is significantly smaller than `gpu_elapsed_time`, and is uniform for grid sizes up to `n = m = 512`, at which point it begins to scale exponentially with the grid size, until it converges with `gpu_elapsed_time`.
This behaviour reflects the nature of GPU programming, in that `gpu_elapsed_time` can be separated into the time spent setting up the computation on the GPU, `setup_time`, and the time it actually takes the GPU to perform the computation, `kernel_time`; that is, `gpu_elapsed_time = setup_time + kernel_time`. Typically, `setup_time` scales constantly with grid size, since it's limiting factor is usually the I/O between the CPU and the GPU, memory allocation, et cetera.
For small grid sizes, `kernel_time` is significantly smaller than `setup_time`, and so `gpu_elapsed_time` scales constantly with grid size. For larger grid sizes, `kernel_time` increases as the GPU computational overhead increases, and as the GPU blocks begin to be allocated larger sections of the grid to work on. Hence, `gpu_elapsed_time` will tend to be pre-dominated by `kernel_time` past a certain grid size, and begin scaling exponentially with the grid size.

3.2 OpenACC Code

The performance characteristics of the OpenACC code (that is, the extracted timing data) are shown in Table 2 and the timing data is plotted in Figure 2.

n = m	cpu_elapsed_time	gpu_elapsed_time	speedup	kernel_time
1	0.01	157.22	0.0001	0.95
2	0.01	180.11	0.0000	0.98
4	0.05	178.12	0.0003	1.01
8	0.08	182.70	0.0004	1.03
16	0.60	163.22	0.0037	0.98
32	1.34	176.89	0.0076	0.97
64	7.25	160.52	0.0451	0.96
128	21.63	182.07	0.1188	1.25
256	60.67	163.94	0.3701	1.07
512	269.47	163.32	1.6499	1.24
1,024	895.33	153.91	5.8174	2.50
2,048	4,185.30	189.78	22.0535	6.47
4,096	13,039.05	201.45	64.7247	22.39
8,192	92,779.27	342.34	271.0120	87.18
16,384	243,636.23	974.39	250.0390	393.43

Table 2: Performance characteristics of OpenACC code, compiled with `-O2`, and with debugging statements turned off. All times are presented in units of ms.

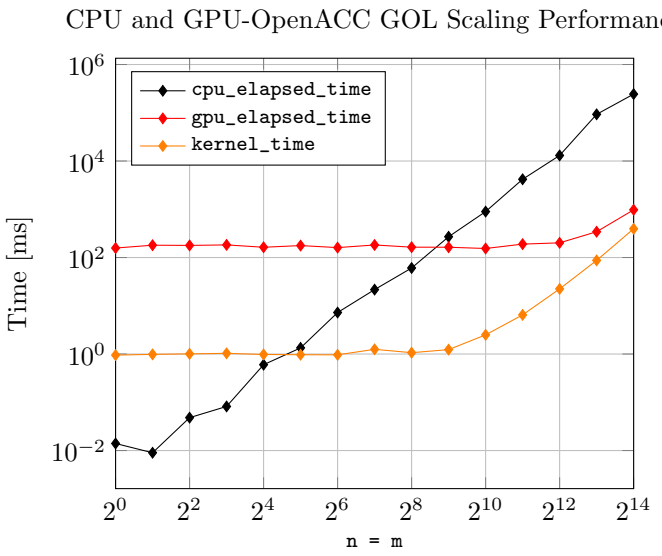

Figure 2: The scaling performance of the CPU and GPU-OpenACC GOL codes are shown for `n = m = 1, 2, 4, 8, ..., 16384`*, with* `nsteps = 100`*. Note that the x-axis is presented in* $\log_2$ *scale, and the y-axis is presented in* $\log_{10}$ *scale.*

It can be seen that the CPU code is more performant that the GPU-OpenACC code for small grid sizes up to `n = m = 512`. We note also that is scales exponentially with the grid size, for all grid sizes. It can be seen that OpenACC code is uniformly performant for grid sizes up to `n = m = 4096`, at which point it begins to scale exponentially with the grid size, but at a slower rate than the CPU code. It can also be seen that `kernel_time` is significantly smaller than `gpu_elapsed_time`, and is uniform for grid sizes up to `n = m = 512`, at which point it begins to scale exponentially with the grid size, until it converges with `gpu_elapsed_time`.
This behaviour reflects the nature of GPU programming, and was discussed in detail in subsection 3.1 with regard to the CUDA code, but is similar for the OpenACC code also.