CONWAY'S "GAME OF LIFE"

Homework # 5

By

Marcelo Torres

CS 481/581 High Performance Computing

December 3, 2024

Problem Specification:

The objective of this assignment is to design and implement a simulation of Conway's "Game of Life," a cellular automaton devised by mathematician John Horton Conway. The "Game of Life" is played on a two-dimensional grid of cells, where each cell can be either alive or dead. The state of the cells evolves over discrete time steps according to a set of rules based on the states of neighboring cells. The rules are applied differently to living and dead cells. For living cells, if a cell has less than two living neighbors, it dies by underpopulation. If a cell has two or three live neighbors, it remains alive. Finally, if a live cell has more than three neighbors, it dies by overpopulation. Dead cells, on the other hand, remain dead unless they have exactly three live neighbors, in which case they become live cells by reproduction.

The goal of this project is to demonstrate the effect of different parallel approaches on Conway's "Game of Life" by writing a Cuda based program and comparing its performance to similar cpu based programs. The program must be able to calculate the next game board and write results to an output file. The program must also be able to check results against the contents of an output file to verify that changing the number of processes does not affect the accuracy of the output. To that end, the program designed must use a dynamically allocated game board and accept the board size, maximum iteration count, number of threads, output directory, and test file as input.

Program Design:

In this iteration of my design, I used the same structure as my MPI approach, and I utilized the same logic for checking the cells and checking for updates. However, this iteration of the project also contained some significant changes. Cuda requires a different approach to any problem and Conway's Game Of Life is no exception. First, I implemented the game using a standalone kernel that stored all its data in global memory and did not check for any updates. Next, I used atomic functions and warp synchronization primitives to implement an efficient method of detecting updates. Lastly, I rewrote the main kernel to work using the shared cell structure outlined in the paper Performance analysis and comparison of cellular automata GPU implementation. The authors of that work were kind enough to provide code which I used as reference for my work. In the shared cell model, each thread updates two cells instead of one. This should theoretically reduce the number of calls to memory and make the program faster, but the authors found that it was comparable to the baseline code in testing. I also found similar results in my testing, although the shared memory approach performed slightly better on my computer.

I believe that this program is well optimized, but further optimization could be found in the layout of the grid and blocks as well as the design of the update checks and the algorithm. I still believe that a lot of computation could be saved by marking sections as dead or ignored when they do not need to compute, but that approach continues to be too far outside the scope of this project.

Testing Plan

The game logic was tested using small board sizes of 5x5 and a single iteration. The results were printed and compared to the website https://playgameoflife.com/ which provides a free interactive implementation of Conway's "Game of Life." An example is shown below:

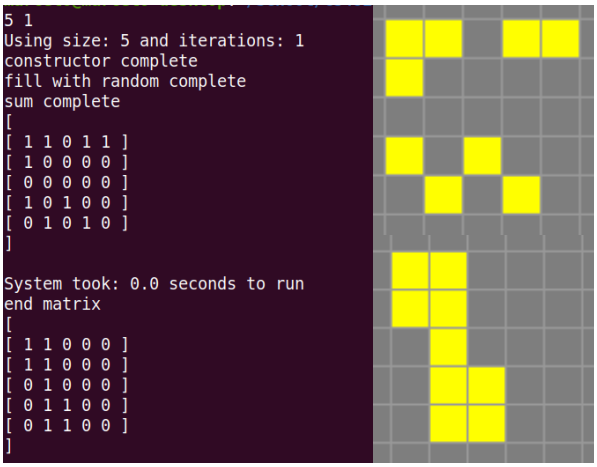

Figure 1: A validation of the game logic

The above figure depicts a printout of the game state before and after an iteration, as well as a graphic representation of the state taken from the website listed above. For this project, this testing was extrapolated to include the new code running on an Nvidia GeForce RTX 3070. The results of that testing were successful and can be viewed fully in `homework5Submission/test-results/ validate-results.txt` in the GitHub repository.

Validation for early stopping was tested using a 5x5 identity matrix. The results of the test can be found in `homework5Submission/test-results/validate-validate-early-stop.txt`. This file contains the results of the diffs with the single thread version as well as the iterations performed on the board.

Test Cases

All Tests were run using a random, uniform distribution with a constant seed, which yields 50.0036% population, to populate the board. Tests were computed in sets of 3s on a 5000x5000 board and a 10000x10000 board with 5000 iterations for each. For the MPI code, 20 threads were used for all runs. For Cuda, block sizes of 128 threads with dimensions 4x32 were used. These block sizes can only use half of the available blocks from each SM, but they align with the warp size of 32 hopefully allowing for more efficient memory access. The tests were compiled using nvcc or mpicxx where applicable and CMake. The code was run using one NVIDIA A100 SXM4 graphics card with 40GB of ram provided by the HPC. The results for each test were stored in an output file and compared to an output file generated using the already tested MPI implementation. The test results for the Cuda based code can be found in `homework5Submission/test-results/test-hw5-10000.o218540`. Tet results for the MPI code can be found in: `homework5Submission/test-results/test-hw4-20only.o207375` and `homework5Submission/test-results/test-hw4-20only.i207375`. All files mentioned are in the github repo.

There were no early stops during the entire run. Additionally, there were no mismatches between the code's output and the test file. Results were identical for every run. The results for each run are shown below:

GPU	Run 1	Run 2	Run 3	Average Time (s)
5000x5000	5.441	5.459	5.490	5.463
10000x10000	21.700	21.114	21.790	21.535
MPI (CPU)	Run 1	Run 2	Run 3	Average Time (s)
5000x5000	17.620	17.460	17.610	17.563
10000x10000	119.306	95.208	94.355	102.956

Table 1: Runtimes for each test

Analysis

The Cuda version of the program significantly outperformed the MPI version of the program. It completed the tasks 3-5 times as fast as the MPI version did. I suspect that increased board sizes would continue to widen this gap. I am unaware of the method to calculate efficiency for a GPU, but I will say that is likely less efficient per unit of processing power used. The following graph depicts some pertinent speedup metrics. On the left side the framework is listed along with the board size. Then the Speedup Compared to Column displays which measurement speedup was calculated from. MPI is compared only to the 1 thread version since it is a good comparison. Note that the MPI times listed in this table were all taken with 10 threads as compared to 1 so we see 20 times speedup. The N/A listing indicates that it is not useful to compare the speed of the 1000x10000 size board to that of the 5000x5000 sized board. Speedup for Cuda is calculated compared to the 1 thread runs with MPI but also compared to the 20 thread runs with MPI. Cuda ran multiple times faster than MPI even when compared to a 20-thread implementation of the code.

Board Size	Framework	Speedup Compared To	Average Time (s)	Speedup
5000x5000	MPI	1 Thread MPI	17.563	20.047
5000x5000	Cuda	1 Thread MPI	5.463	64.447
5000x5000	Cuda	20 Thread MPI	5.463	3.215
10000x10000	MPI	1 Thread MPI	102.956	N/A
10000x10000	Cuda	1 Thread MPI	21.535	16.350
10000x10000	Cuda	20 Thread MPI	21.535	4.781

Table 2: Average efficiency of each set of runs

The speedup and efficiency for the CUDA-based program were both better than that of the MPI program. An analysis of efficiency has been omitted from this report since CUDA's calculated efficiency when using all 221184 Cuda cores is 0.03% even when compared to the 1 thread MPI program. However, the speedup beyond the 20 thread MPI program clearly demonstrates the strength of this computing approach.

Conclusions

If I were to try to improve this system further, I would explore ways to track cells that do not need to be changed. I would also investigate different methods for storing and processing the board. Millán Et. Al. reference a paper that analyses the performance of MPI programs on Conway's game of life and introduces a method of storing the entire board as one contiguous 2D array where each iteration stores the new data and uses some previous calculations to continue moving forward through the board. I believe that the present codes speed is a demonstration of Cuda's computing power. I made very few changes to the structure of the code compared to the MPI version, yet I gained a 3-4 times speedup. Overall, I am satisfied with the work put forward in this project, and I am satisfied with what I have learned.

Sources

GitHub repo: https://github.com/Player1ce/CS481-HPC

Homework1-Report.docx by Marcelo Torres (URL not available)

Homework3-Report.docx by Marcelo Torres (URL not available)

Homework4-Report.docx by Marcelo Torres (URL not available)

https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

https://playgameoflife.com

Millán et. Al.: https://link.springer.com/article/10.1007/s10586-017-0850-3

Millán Et. Al.: https://www.sciencedirect.com/science/article/pii/S0045790615003225