PARALLEL IMPLEMENTATION OF CONWAY'S GAME OF LIFE

Homework # 5

By

Nolan McKivergan

CS 581 High Performance Computing
November 29, 2024

GitHub Link: https://github.com/npmckivergan/Fall2024_CS581_HW5

Problem Specification:

Conway's Game of Life is a cellular automaton invented by mathematician John Conway in 1970. It is a zero-player game, meaning the progression is determined purely by the initial input. The game is inspired by the motion of biological colonies over many generations and is used in a wide variety of fields ranging from biology, to computer science, to cryptography.

The game is played on a square two-dimensional grid, where each cell can be in one of two states: dead or alive. For each generation, every cell in the grid is updated simultaneously according to a set of rules regarding the status of the neighbors. Each cell has eight neighbors since diagonals are included.

The rules determining whether a cell becomes alive or dead are as follows:

1. If the cell is alive:
    a. If the cell has less than two living neighbors, the cell dies from loneliness.
    b. If the cell has more than three living neighbors, the cell dies from overpopulation.
    c. If the cell has exactly two or three living neighbors, it remains alive.
2. If the cell is dead:
    a. If the cell has exactly three living neighbors, it becomes alive from reproduction.
    b. Otherwise, the cell remains dead.

This game was to be implemented using C/C++ and CUDA on an Nvidia GPU. The grid size, maximum number of generations, number of processes, and output file location passed in as command-line arguments. The cells were to be initialized to alive or dead states randomly at the start of the program. The program runs without printing anything and terminates when there are no changes between two subsequent boards or the maximum number of generations has been reached. To parallelize this with the GPU, CUDA is utilized to break the board into parallel regions that can be computed simultaneously. Several versions must be implemented: one basic version using global memory, one optimized version using shared memory, and one version using an optimization technique from the paper specified in Blackboard.

Program Design:

The base of these parallel programs is derived from my code in Homework 1, which serves as the serial baseline for both correctness validation and performance benchmarking. CUDA is then integrated into this existing code to parallelize the workload and take advantage of GPU resources.

The first program, `gpu_global.cu`, utilizes global memory to store matrices. In this version, the matrices are divided into 16x16 cell blocks that the GPU processes concurrently. Each matrix element corresponds to one thread, with each block containing 256 threads, corresponding to a 16x16 block of cells. While this approach efficiently leverages multithreading, it still suffers from a significant bottleneck: each block accesses global memory to update the matrices. Global memory access is relatively slow compared to other types of memory, leading to potential delays and limiting overall performance.

In the second program, the core logic remains similar, but with a key optimization. Instead of allocating matrices in global memory, smaller submatrices are allocated within each block in shared memory at the start of the `updateKernel` function. This submatrix is local to the block, meaning that only threads within the same block can access it. By moving the submatrices to shared memory, the time it takes to load and access matrix values is greatly reduced, as shared memory is much faster than global memory. Each block now operates more efficiently, as it accesses its own local copy of the matrix rather than repeatedly accessing the global matrix.

The third program, `gpu_paper.cu`, introduces several additional optimizations based on the strategies outlined in the paper. While the core logic remains consistent, the following key improvements are implemented:

1. **Global Change Detection for Early Exit**: Each block maintains a local change flag, which is used to detect if any changes occurred during the computation. If a block detects a change, it updates the global change flag using atomic operations. This allows the simulation to terminate early if no changes are detected, indicating that the grid has stabilized. This optimization reduces unnecessary kernel launches when the system has converged before reaching the maximum number of generations.
2. **Pointer Swapping Instead of Matrix Copying**: To further reduce memory overhead, the program swaps device pointers between the current and next matrices instead of copying the data from one matrix to the other. This eliminates the need to perform potentially expensive memory copies after every kernel invocation, significantly improving both runtime and memory bandwidth efficiency.
3. **Explicit Handling of Corner Regions (Ghost Cells)**: The third program also introduces explicit handling for boundary and corner regions of the matrices. These regions (often referred to as "ghost cells") are handled to ensure that out-of-bounds memory accesses do not occur, even in the corners of the grid. This is particularly important in non-square or edge-aligned grids, where the handling of matrix boundaries is more complex. By explicitly managing these boundary conditions, the program avoids memory errors and ensures the correctness of the simulation across the entire grid.

Testing plan:

This program had to be tested for both correctness and performance. These test cases were performed separately and are detailed below.

To test for correctness, all GPU programs were executed with 5000x5000 and 10000x10000 cell grids with 5000 maximum generations, with the outputs written to the scratch directory. Then the serial program was executed with the same parameters. Then the "diff" command was used to compare these files. If no difference is found, then that serves as proof the GPU programs work properly.

To test for performance, the same test cases of 5000x5000 and 10000x10000 cell grids with 5000 maximum generations were executed three times for all GPU programs, and every CPU version implemented thus far. This includes the serial program, the OpemMP program, the the MPI program. For the multithreaded programs, 20 threads were used and 20 cores allocated to the job so each thread can have its own core.

Test Cases:

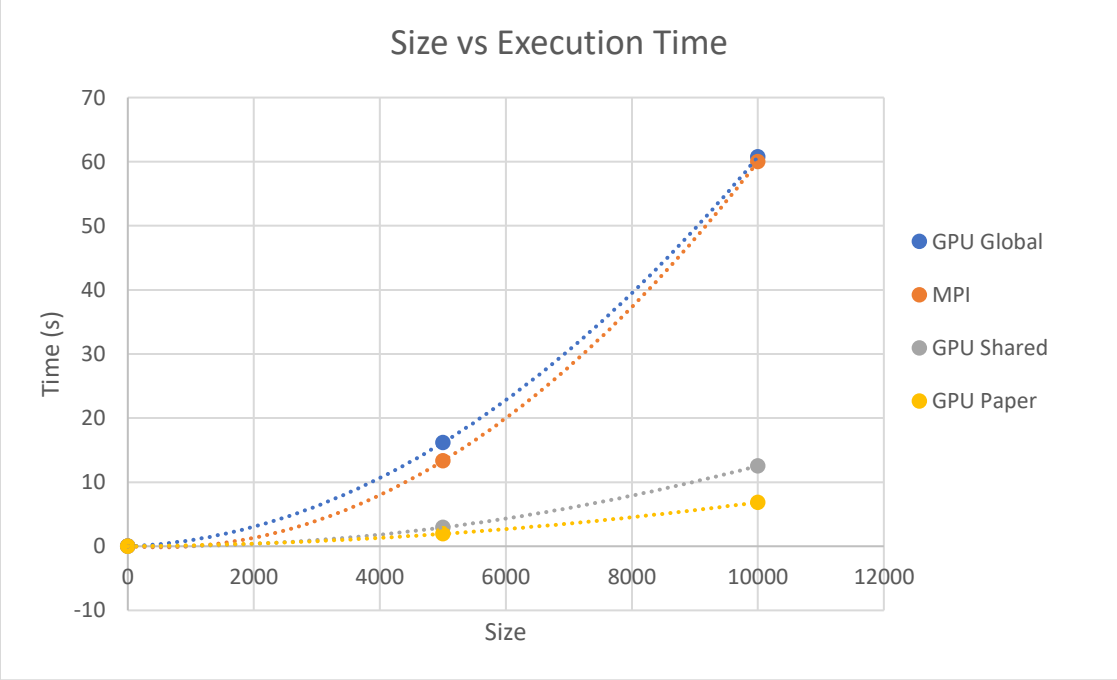
To test for correctness, I used the "correctness.sh" script to run the test cases using the GPU and serial programs, produce the output files, and check for differences between all of them. In the "correctness" job output file, nothing was printed out by the "diff" command, indicating no difference was found between any of the boards. This shows that the final boards are identical for all GPU programs and the serial program, indicating all the GPU programs work correctly.

To test performance, I used the "cpu_performance.sh" and "gpu_performance.sh" to run the performance test cases described previously. The resulting runtimes are shown in the tables below.

Test Case	Serial	OpenMP	MPI	GPU Global	GPU Shared	GPU Paper
5000	381.926	205.41	13.322	16.163	2.938	1.936
10000	1720.003	925.064	59.995	60.723	12.532	6.856

Analysis and Conclusion:

The following plots show the measured runtimes for each implementation of the Game of Life, for both the 5000x5000 and 10000x10000 test cases. Two plots were used to make it easier to distinguish between the performances of the MPI program and the GOPU programs, as both were so much mor efficient than the serial one it is impossible to see the difference with all in the same graph.



These graphs certainly show that the multithreaded programs are tremendously faster than the serial one. However, the optimized GPU programs are significantly faster than the fastest MPI multithreaded CPU program. Most notably, the GPU program with optimizations in both shared memory and techniques from the research paper executed much faster than the MPI program by about 8x, and even the global memory GPU programs in the largest 1000x10000 test case. Performance was less improved compared to the shared memory version, but still somewhat faster. Overall, it is apparent that a well-crafted GPU implementation will finish light-years ahead of a serial program for large test cases, and still absolutely smoke even a carefully optimized multithreaded CPU version.

Questions:

1. Did you achieve the goals or objectives of this assignment?

   Yes. I created an efficient and accurate implementation of Conway's Game of Life utilizing Nvidia GPUs with Cuda for parallelization, which was the specific objective of this assignment. I built my solution with serial code from a previous assignment and implemented multithreading. I then optimized the program with shared memory access and a variety of techniques entailed in the research paper.

2. How did you achieve these goals (be specific)?

   I studied online resources to develop a solid understanding of the game. I read the assignment document to understand how I was to handle all the small nuances of the game for this purpose. I became familiar with the CUDA libraries and how to properly divide the grid into blocks and synchronize the blocks. I carefully optimized my code to use shared memory access similar to the provided examples. I thoroughly read the assigned research paper and implemented the techniques discussed there to further optimize the performance.

3. What can you infer from the performance results you have obtained? Does the performance results match your expectation? Explain.

   As I said in the analysis section, based the results we can infer that MPI is much more effective at parallelizing tasks on a distributed-memory system than OpenMP. Furthermore, by using non-blocking end-to-end calls to exchange information among the processes rather than the built-in MPI_Sendrecv functions, even better performance is possible. Using MPI, tasks on a distributed system such as the ASC Supercomputer Cluster can be parallelized with an almost perfect degree of efficiency provided there are as many or more physical cores allocated to the job as processes in the program.

4. Looking back on the project what would you do differently?

   If I were to do this project again, I would start sooner. Implementing GPU multiprocessing was difficult, and it took a lot of time and effort to get it working properly and test it for correctness.

5. What changes would you make in the design phase of your project?

   During the design phase, I would take greater care to properly understand how the processes communicate and synchronize in the code before making any modifications. It took me several tries to implement the code, and I spent a lot of time just trying to change random factors to get it to work. A more design-oriented approach would have been easier in the long run.

6. What additions would you suggest to improve the working of your program?

   Honestly, I don't know of any way to improve my program from a design perspective. I properly implemented the Game of Life on an Nvidia GPU with CUDA, and leveraged shared memory access and advanced optimization techniques. I don't believe there is much room for improvement with the provided hardware constraints and scope of this project.

References:

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

https://playgameoflife.com/

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

https://github.com/kamalchowdhury326/HW5_helper