NVIDIA GPU IMPLEMENTATION OF CONWAY'S GAME OF LIFE,

Homework 5

By

Gowtham Prasad

CS 581 High Performance Computing
December 5. 2024

Problem Specification:

The objective of this project is to design and implement a GPU version of the "Game of Life" program in CUDA C++.

The goal of the CUDA C++ code is to create and run a simulation of Conway's Game of Life, a cellular automaton in which a one-dimensional array of cells to represent a 2d array and evolves according to predefined rules. It was American mathematician John Von Neumann who first proposed the concept of a universal constructor, or a software that could replicate itself automatically by analyzing data, back in the 1940s, when life first originated. Von Neumann created a discrete game with a maximum of 29 distinct states for each cell in a two-dimensional grid of square cells to explore this problem. In this game, a cell's state in each generation is determined by the state of adjacent cells. This universal constructor fascinated British mathematician John Conway, who used it as motivation to study a new computable discrete world. Alan Turing's idea of a “universal computer,” or a device that can mimic any computational process by adhering to a brief set of instructions, served as the foundation for this study. Conway thus began to search for a computable “simple universe,” a condensed set of states and laws. After experimenting with numerous configurations on a two-dimensional grid to establish an optimal balance between extinction and unlimited cellular proliferation, he named this set of rules the Game of Life.

Each cell is categorized as either “alive” or “dead” based on the influences of its eight neighbors. The simulation must precisely update the grid over a number of generations and terminate when either a stable state is reached, or the maximum number of generations is reached. Only if a surviving cell in this generation has two or three neighbors will it survive until the next generation; otherwise, it dies from loneliness or overcrowding. Even a dead cell can spontaneously regenerate into a living one in the next generation if it has exactly three neighbors in the present generation. The seed of the system is the original pattern and is an N x N matrix. The criteria are applied simultaneously to all of the seed's cells, whether they are living or dead, to generate the first generation. Births and deaths take place at the same time, and this discrete moment is frequently referred to as a tick. Every generation is solely dependent on the one before it. The guidelines are consistently followed in order to produce new generations.

Program Design:

The CUDA implementation of Conway's Game of Life is designed to leverage the parallel processing capabilities of NVIDIA GPUs. The program represents the simulation grid as a one-dimensional array for efficient memory management and employs CUDA kernels to update the grid based on the rules of the Game of Life. Two kernel implementations are utilized: a baseline version using global memory and an optimized version leveraging shared memory. The shared memory optimization minimizes memory access latency by loading grid sections into faster shared memory during each iteration. This allows each thread to access neighbor states

more quickly, improving overall performance. The program also includes mechanisms to detect convergence by monitoring changes in the grid across generations, using atomic operations to track stability efficiently.

Testing Plan:

The CUDA implementation was rigorously tested on an HPC cluster equipped with NVIDIA A100 GPUs to evaluate its performance and correctness. The tests involved varying matrix sizes and generations to assess scalability and efficiency. For consistency, a fixed seed initialized the grid to ensure reproducibility across runs. Each test case simulated 5000 generations with matrix sizes ranging from 5000x5000 to 10,000x10,000. The baseline and shared memory-optimized kernels were compared against a CPU implementation, measuring execution time to determine the relative speedup. The results were validated by ensuring the output grid matched the expected evolution rules. Tests were performed under constrained conditions, including a 1-hour time limit, 8GB of memory, and exclusive GPU usage, ensuring a fair evaluation of the implementation's performance. The analysis focused on execution time, scalability, and the effectiveness of shared memory optimization.

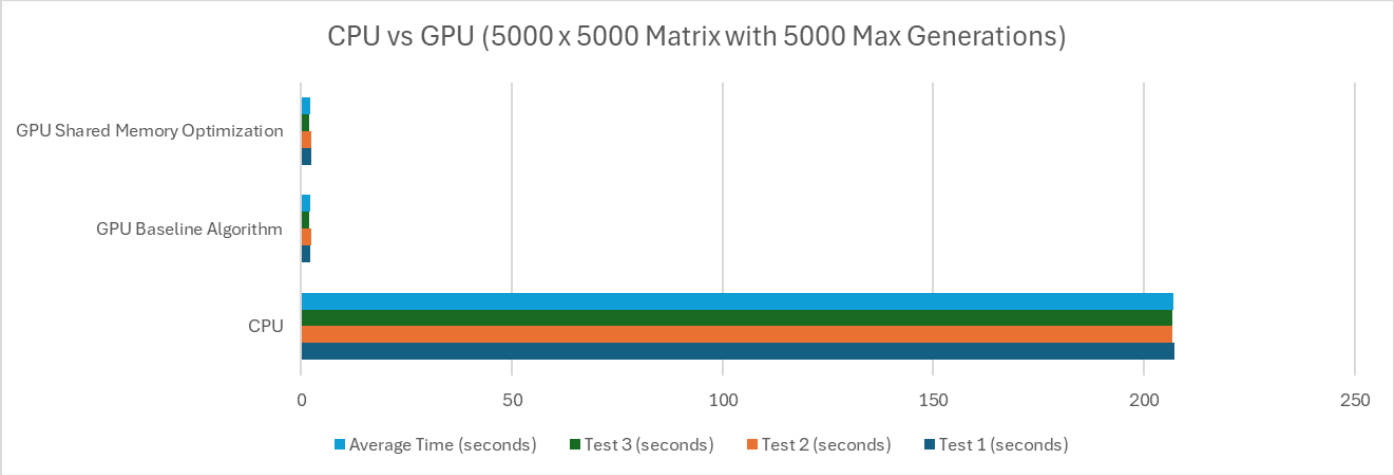
Test Cases:

Tests 1 through 3 for all iterations of the Game of Life experienced the following conditions in the HPC's ASC cluster:

- Sent to the classgpu queue
- A time limit of 1:00:00
- A memory limit of 8gb
- 1 Core
- 1 NVIDIA A100 GPU

Type	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
CPU	207.133746	206.685801	206.810894	206.8768137
GPU Baseline Algorithm	2.239413	2.271665	1.841294	2.117457333
GPU Shared Memory Optimization	2.271594	2.278922	1.917323	2.155946333

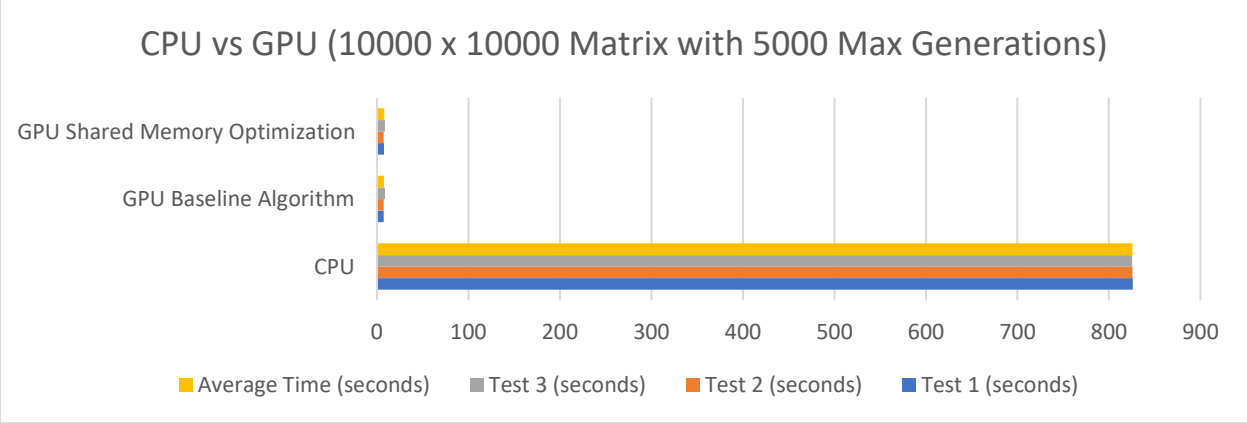
Table 1. CPU vs GPU (5000 x 5000 Matrix with 5000 Max Generations) Table



Graph 1. CPU vs GPU (5000 x 5000 Matrix with 5000 Max Generations) Plot

Type	Test 1 (seconds)	Test 2 (seconds)	Test 3 (seconds)	Average Time (seconds)
CPU	826.09669	825.867047	825.425572	825.7964363
GPU Baseline Algorithm	7.450346	7.45398	8.732892	7.879072667
GPU Shared Memory Optimization	7.547895	7.508908	8.761515	7.939439333

Table 2. CPU vs GPU (10000 x 10000 Matrix with 5000 Max Generations) Table



Graph 2. CPU vs GPU (10000 x 10000 Matrix with 5000 Max Generations) Plot

Analysis and Conclusions:

The comparison between MPI, OpenMP, and CUDA implementations of Conway's Game of Life highlights distinct performance characteristics and suitability for different scenarios. In the MPI implementations evaluated in Homework 4, the nonblocking version

outperformed the blocking version, especially at higher processor counts. The nonblocking approach reduced communication overhead by overlapping computation with communication, resulting in better scalability. OpenMP, on the other hand, performed well up to 10 threads, but its scalability was limited by thread management and synchronization overheads, which caused diminishing returns as the thread count increased.

The CUDA implementations in Homework 5 demonstrated a significant leap in performance compared to the CPU, OpenMP, and MPI versions. The GPU-based CUDA algorithms executed the simulation in approximately 2 seconds for a 5000x5000 matrix over 5000 generations, in stark contrast to the ~200 seconds required by the CPU version and the ~121 seconds for the fastest MPI implementation. Between the two CUDA algorithms, shared memory optimization provided a marginal improvement over the baseline algorithm, especially for larger grid sizes, by minimizing memory access latency.

In terms of scalability, CUDA emerged as the most efficient solution, leveraging the massive parallelism of NVIDIA GPUs and maintaining high performance even for large matrix sizes. The nonblocking MPI implementation also showed good scalability, maintaining higher efficiency than its blocking counterpart, making it a strong alternative in distributed memory environments where GPU resources are unavailable. OpenMP, while effective for smaller-scale simulations, struggled with efficiency at higher thread counts due to synchronization and thread management costs.

In conclusion, CUDA shared memory optimization offers the fastest execution times and best scalability, making it the optimal choice for high-performance simulations on GPU hardware. For CPU-based parallelism, MPI nonblocking is more scalable and efficient than OpenMP, especially in distributed memory systems. Overall, CUDA is ideal for extremely large-scale simulations requiring high throughput, while MPI and OpenMP remain valuable for scenarios constrained to CPU-based parallelism.

References:

1. Code sample life.c, Md Kamal Chowdhury, September 24, 2024
2. Code sample hw4_kamal.c, Md Kamal Chowdhury, November 1, 2024
3. “Play John Conway’s Game of Life.” Playgameoflife.com, playgameoflife.com/. Accessed September 10, 2024

Github Link: https://github.com/gdprasad1201/CS-581-HPC-Fall-2024/tree/main/HW%205