Trenton Smiley

Homework 3 CMSE 401

March 23, 2025

Dr. Haut

Accelerating Conway's Game of Life: A Benchmarking Study of OpenMP and CUDA Implementations

Introduction

Conway's Game of Life is a cellular automaton consisting of a grid of cells that evolve over discrete time steps based on simple neighbor-based rules. While the rules are straightforward, simulating large grids over many iterations can be computationally intensive. This project explores the performance of a provided parallel CPU implementation using OpenMP and compares it to a GPU-accelerated version using CUDA. The goal was to understand the effectiveness of parallelization in both models and to determine if CUDA offers a performance advantage over OpenMP in this application.

Benchmarking the OpenMP Implementation

The starting point for this assigment was a C program that implemented Conway's Game of Life with OpenMP directives to parallelize the cell updates. Each iteration consisted of sweeping over the grid and applying the update rules in parallel using OpenMP's #pragma omp parallel for directive. To benchmark this version, I bypassed the input parsing by allowing the simulation to generate a random board when n <= 0. I ran the simulation for 10 iterations, as specified, and used a bash loop to execute each test five times for consistency. Execution time was measured using clock() from the C standard library, with the timer wrapped around the iteration loop. The average time for each board size was recorded across five runs. Grid sizes tested included n = 128, 256, 512, 1024, 2048.

CUDA Implementation and Optimization

To further accelerate the simulation, I rewrote the core update logic as a CUDA kernel. The kernel, updatePlate, was designed to parallelize updates across both dimensions of the grid using a two-dimensional block and grid structure. Device memory was allocated to store both buffers of the grid, and the initial state was copied to the GPU before the iteration loop began.

Each iteration involved launching the kernel with dim3 block and grid dimensions computed based on the board size. I used CUDA's cudaEvent_t timers to record the duration of the iteration loop with high precision. Synchronization was enforced after each kernel launch using cudaDeviceSynchronize(), ensuring accurate timing. After the simulation, the final board state was copied back to the host for optional printing or image export.

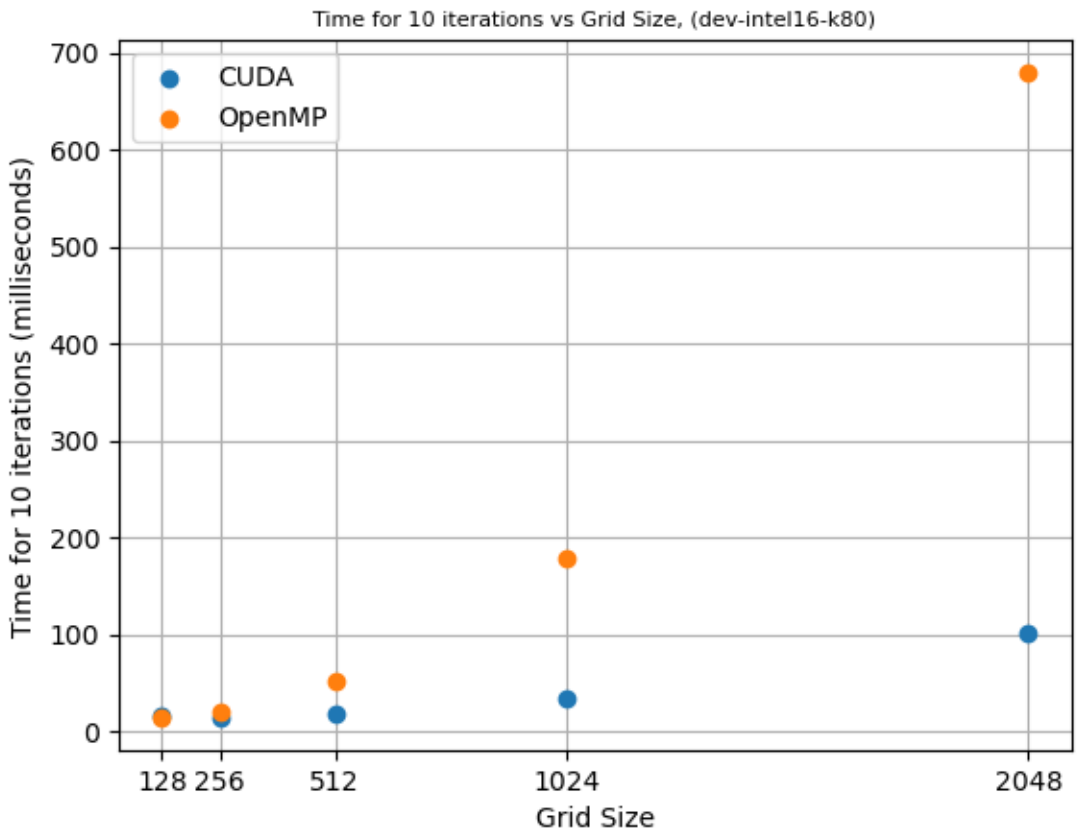
The CUDA code was compiled using nvcc, and tests were run on the dev-intel16-k80 node of Michigan State University's HPCC, which provides access to an NVIDIA Tesla K80 GPU. Like the OpenMP version, each test used a randomly initialized grid and ran for 10 iterations across the same board sizes.

Benchmark Results and Discussion

The graph below shows the execution time (in milliseconds) for both the OpenMP and CUDA implementations across increasing board sizes. Each data point represents the average time over five runs.

As expected, the execution time increased with board size for both implementations. The OpenMP version performed adequately for smaller grids but scaled poorly beyond n = 512, with execution time rising sharply due to the increased number of memory accesses and synchronization overheads across CPU cores. In contrast, the CUDA implementation demonstrated significantly better scalability. For large grids (e.g., n = 1024 and above), the GPU version outperformed the OpenMP version by an order of magnitude.

An interesting observation was that for very small board sizes (e.g., n = 128), the CUDA version did not show a large speedup over OpenMP. This is likely due to kernel launch overhead and memory transfer costs, which become negligible only when amortized over large numbers of parallel computations. In other words, CUDA shines when the problem size is large enough to saturate the GPU's parallel cores.



No additional optimizations were made to the OpenMP code beyond what was provided. However, performance could potentially be improved with dynamic scheduling, better memory locality, or thread affinity settings. On the CUDA side, optimizations such as using shared memory or minimizing divergent branching could offer further improvements.

Conclusion and Lessons Learned

This project demonstrated how GPU acceleration using CUDA can significantly improve the performance of a computational simulation like Conway's Game of Life. While the OpenMP version provided a good baseline for multi-core CPU parallelism, the CUDA implementation achieved far better scalability for large board sizes.

Through this experience, I learned:

- How to convert a CPU-based parallel program to CUDA, including memory management and kernel design.

- The importance of using proper synchronization and precise timing tools for benchmarking.
- GPU acceleration has overhead, and speedups depend on problem size and architecture.
- How to write reproducible benchmarks and compare performance across architectures.

Ultimately, while CUDA introduces additional complexity, it can be a powerful tool for high-performance computing when applied to the right problem domains.

Example Output for 5x5 grid: