CUDA Kernel Optimization and Benchmark Study for Cellular Automatons

Ayush Gala
agala2@ncsu.edu
North Carolina State University
Raleigh, NC, USA

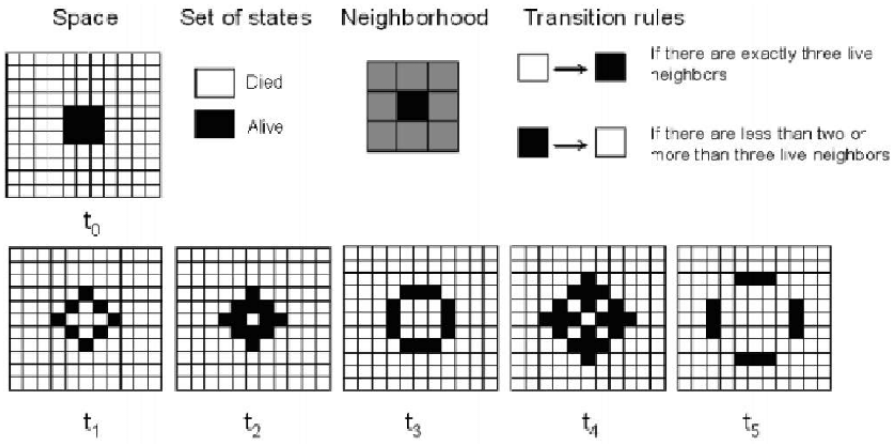

Figure 1: Element of cellular automata (top) with Conway's game of life as illustration (bottom) (Moreno, 2008)

Abstract

This project presents a parallelized simulation of Conway's Game of Life (GoL), a well-known cellular automaton and investigates the broader applicability of GPU acceleration in cellular automatons and simulations. [11] Simulating GoL involves significant computational overhead due to the per-generation update of each cell based on its neighbors, which becomes computationally prohibitive on large grids when executed serially. [10] To address this, I implemented and benchmarked three approaches: a naïve serial CPU implementation, a multithreaded CPU version using OpenMP, and a GPU-accelerated implementation using CUDA. The simulation operates on a 1000×1000 grid over 557 generations, utilizing the 0hd Demonoid pattern as a reproducible and complex benchmark. Optimization strategies include the use of toroidal boundary conditions with ghost cells to eliminate conditional branching at the edges, memory layout transformations for coalesced access, and the integration of pinned memory with CUDA streams to overlap computation with data transfer. Experimental results demonstrate that the CUDA implementation achieves a 111× speedup compared to the serial baseline, whereas the OpenMP version exhibits only moderate improvement due to memory bandwidth constraints. These findings underscore the critical role of memory hierarchy optimization, architectural awareness, and asynchronous execution in leveraging the full potential of modern GPUs. The accompanying benchmarking framework and codebase lay the groundwork for future research in parallel algorithm design and scalable cellular automata simulation.

CSC-548 Parallel Systems Spring'25,

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

Keywords

Cellular automata, Parallel Systems, GPU computing, CUDA, OpenMP, Memory Access Patterns, CUDA Streams

ACM Reference Format:
Ayush Gala. 2025. CUDA Kernel Optimization and Benchmark Study for Cellular Automatons. In *Proceedings of April 30–05, 2025 (CSC-548 Parallel Systems Spring'25).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts

• **Theory of computation** → *Automata over infinite objects*; • **Computing methodologies** → *Shared memory algorithms*; • **Computer systems organization** → **Single instruction, multiple data**.

Source Code Repository

The complete source code, along with build instructions, test scripts, and performance logs, is available on GitHub at: **https://github.com/Ayush-Gala/cuda-life-benchmark**

1 Introduction

The Game of Life (GoL) is a zero-player game that was invented by John Conway in 1970 [3]. The game consists of a grid of cells. Each cell can be alive or dead. This is usually denoted by coloring live and dead cells with different colors like black and white. All the cells are initialized to being dead. The player sets a subset of the cells to live. The game then evolves without any further player intervention (hence the name zero-player game) based on local rules. These local rules depend on only one factor: how many live/dead neighbor cells a given cell has. A neighborhood is defined by adjacent cells either vertically, horizontally, or diagonally. This defines 8 adjacent/neighbor cells for each cell. The original Conway's GoL was a 2D grid of cells that evolves according to the following rules:

1. Any live cell having two or three live neighbors survives into the next generation. Otherwise it dies.

2. Any dead cell with three live neighbors becomes live in the next generation. Otherwise it stays dead. The same rules are applied over and over again to progress the game from a generation to the next ad infinitum. Surprisingly with only these simple local rules various complex behaviors, shapes and dynamics emerge.

Computer scientists from all spheres of the community have been obsessed with Conway's GoL. It is no surprise that it has also been the subject of many papers involving parallelization. [11] [18] Cellular automaton algorithms including GoL have been thoroughly explored with seven different algorithms [19] concluding that GPU-based algorithms show a large performance increase in some problem sets and sizes. Memory access patterns of cellular automata on GPUs are explored with a special emphasis on GoL [1] recognizing the solutions that speed up these mathematical models are far-reaching and include the fields of computer vision, and image processing, highlighting that higher memory bandwidth resulted in enormous speedups. Recently, researchers have been proposing the use of GPU acceleration for exploring the massive search space to find specific patterns in the game.

In the context of Conway's Game of Life, a soup refers to a random initial configuration of cells, where each cell has a certain probability (often 50%) of being alive or dead [13]. This term is widely used by Life enthusiasts to study the general dynamics and emergent behaviors of the automaton. The nomenclature of soups often includes references to their symmetry (e.g., asymmetric or symmetric soups), density (the proportion of live cells), and longevity (how many generations they persist before stabilizing or dying out). Patterns that arise from soups are categorized based on their eventual outcomes: some soups quickly stabilize into still lifes or oscillators, while others produce spaceships or more complex objects as can be seen in in figure 2.

A key area of research is classifying soups by the types of objects they generate as they evolve. [12] For example, some soups transition into well-known engineered patterns like the 0hdDemonoid in figure 4, a type of self-replicating spaceship, while others remain in a chaotic state for thousands of generations before settling or dying out. The study of these transitions helps researchers understand the landscape of possible behaviors in the Game of Life, from orderly and predictable to highly chaotic and unpredictable. [6] [13] The diversity of outcomes from simple random soups underscores the system's rich potential for complexity and discovery.

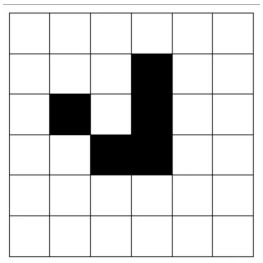

Figure 2: A class IV object - glider

The Game of Life is a classic demonstration of how simple, local rules can give rise to highly complex and unpredictable global behaviors. From random or structured initial states, the system can spontaneously generate stable patterns, oscillators, gliders, and even self-replicating machines. Studying the parallelization of cellular automata like the Game of Life has significant research implications. Because each cell's state depends only on its immediate neighbors, the computation can be efficiently distributed across multiple processors or even GPUs, enabling the simulation of extremely large grids and long time scales. This scalability is crucial for exploring rare or large-scale phenomena, such as the evolution of complex engineered patterns or the statistical properties of soups. Insights gained from parallelized simulations are valuable not only for mathematics and computer science but also for fields like physics, biology, and image processing, where similar local-interaction models are used to study real-world complex systems.

2 Literature Overview

The cellular automaton is an important tool in science that can be used to model a variety of natural phenomena. Cellular automata can be used to simulate brain tumor growth by generating a 3-dimensional map of the brain and advancing cellular growth over time [7]. In ecology, cellular automata can be used to model the interactions of species competing for environmental resources [20]. These are just two examples of the many applications of cellular automata in various fields of science, including biology [2], ecology [21], cognitive science, hydrodynamics [16], dermatology, chemistry [8], environmental science, agriculture, operational research, and many others. Cellular automata are so named because they perform functions automatically on a grid of individual units called cells. One of the most significant and important examples of the cellular automaton is John Conway's Game of Life, which first appeared in [3]. Conway wanted to design his automaton such that emergent behavior would occur, in which patterns that are created initially grow and evolve into other, usually unexpected, patterns. He also wanted to ensure that individual patterns within the automaton could dissipate, stabilize, or oscillate. Conway's automaton is capable of producing patterns that can move across the grid (gliders or spaceships), oscillate in place (flip-flops), stand motionless on the grid (still lifes), and generate other patterns (guns).

2.1 Evolution of Parallelization Approaches

The parallelization of Conway's Game of Life has evolved significantly over recent decades as computing architectures have advanced. Early efforts focused on multi-core CPU implementations [17], while more recent research has shifted toward GPU acceleration and algorithmic optimizations. This evolution reflects the broader trend in high-performance computing toward massively parallel architectures. Conway's Game of Life presents an ideal candidate for parallelization due to its uniform computation pattern and the independence of cell updates within each generation. As Ma et al. note in their performance analysis, the computational requirements grow quadratically with the size of the simulation grid, making parallel processing essential for large-scale simulations. [10]

2.2 CPU-based parallelization techniques

Initial parallelization efforts for the Game of Life focused on utilizing multi-core CPU architectures through frameworks like OpenMP and MPI. Panagiotis and colleagues implemented efficient parallel code using both OpenMP for shared memory systems and MPI for distributed computing environments. Their implementation demonstrated significant performance improvements over sequential code, particularly for large grid sizes. The work by Ma et al. in "A Performance Analysis of the Game of Life Based on Parallel Algorithm" [10] provides a comprehensive examination of OpenMP-based implementations. Their research compared several parallel methods and highlighted the importance of algorithm design in parallel problem-solving. They demonstrated that simple parallelization of the update loop yields substantial performance improvements, but careful consideration of memory access patterns and workload distribution is necessary to achieve optimal results. Additionally, Millán et al. explored distributed memory approaches using MPI, showing how the decomposition of the simulation space across multiple nodes affects performance and communication overhead [14]. Their research highlighted the importance of minimizing communication between processes while ensuring correct boundary handling for cell neighborhoods that span process boundaries.

2.3 GPU Acceleration Techniques

The advent of general-purpose GPU computing has revolutionized Game of Life simulations by enabling massive parallelism. Rokicki's research demonstrates that GPU implementations consistently outperform CPU-based approaches by orders of magnitude for large-scale simulations. Tomas Rokicki, in "Life Algorithms," examined various algorithmic approaches for Game of Life simulation [18]. He observed that "normal life algorithms are generally highly parallel, so they are easily sped up as CPUs gain more cores and as highly parallel GPUs take over more and more the computation". This observation underscores the natural fit between cellular automata and GPU architecture. Memory access patterns are critical for GPU performance, as demonstrated by Bennun et al. in "Memory Access Patterns: The Missing Piece of the Multi-GPU Puzzle". They implemented a Game of Life simulation that utilized proper memory access patterns to optimize performance across multiple GPUs. Their framework demonstrated the importance of proper memory management and data partitioning strategies for multi-GPU scenarios. Millán et al. conducted comprehensive performance analysis across five different NVIDIA GPU architectures from Tesla to Maxwell, simulating systems with up to a billion cells. [14] Their research identified optimal implementation strategies across different GPU generations and highlighted the evolution of performance characteristics as GPU architecture advanced.

2.4 HashLife Algorithm and Parallelization Challenges

The HashLife algorithm [4], developed by Bill Gosper in the early 1980s, represents a fundamentally different approach to Life simulation. As described in Wikipedia, "Hashlife is a memoized algorithm for computing the long-term fate of a given starting configuration in Conway's Game of Life and related cellular automata, much more quickly than would be possible using alternative algorithms". Instead of simulating each generation cell by cell, HashLife exploits spatial and temporal redundancy in Life patterns using quadtree representation and memoization of results. This allows it to effectively "skip ahead" many generations when processing repetitive patterns. For suitable patterns, HashLife can compute trillions of generations in seconds.

However, HashLife presents unique challenges for parallelization. According to Rokicki, "So far, hashlife has been resistant to parallelization. A big challenge for the community is to write an effective, efficient, parallel hashlife, perhaps based on lock-free hashtables". [18] The algorithm's effectiveness stems from its memoization approach, which creates dependencies that complicate parallel implementation. Recent research by John Hutton explores Python implementations of Hashlife and notes that while the algorithm is challenging to parallelize in its standard form, certain components could potentially be executed on GPUs [5].

3 Problem Definition

The primary computational challenge of GoL lies in its memory-intensive nature rather than computational complexity. Each cell update requires reading nine values (the cell itself and its eight neighbors), performing simple calculations, and writing a single value back to memory. [15] This memory-bound characteristic makes it an excellent candidate for exploring GPU optimization techniques that can efficiently handle parallel operations across numerous cells simultaneously.

This project focuses on developing and benchmarking various CUDA implementations to understand the performance characteristics and identify the most effective optimization strategies. The specific problem addressed is the efficient simulation of a large cellular automaton grid (1000×1000) evolving through multiple generations (557) using the NVIDIA CUDA framework. I aim to quantify performance gains and identify memory transfer bottlenecks that dominate execution time in GPU implementations.

3.1 Data Structures

The implementation of Conway's Game of Life requires appropriate data structures to efficiently represent and manipulate the cellular grid.

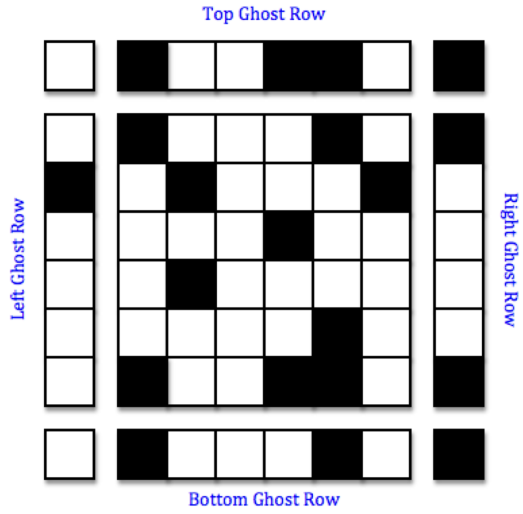

Figure 3: Toroidal Grid Implementation

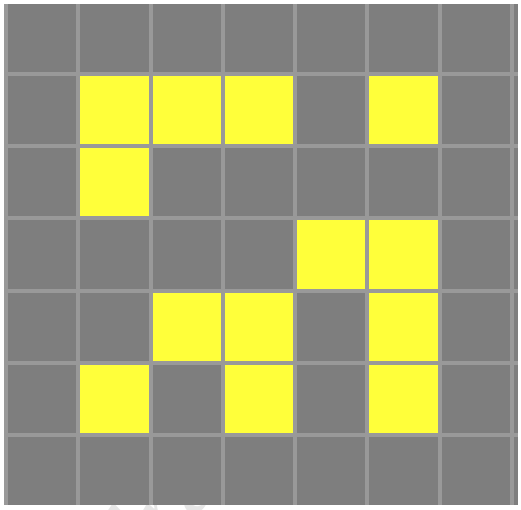

Figure 4: 0hd Demonoid Starting Soup

3.1.1 Grid Representation: The core data structure is a 2-dimensional grid of cells. For the CPU implementation, this was implemented as a direct 2D array. However, CUDA kernels do not efficiently handle multidimensional arrays, necessitating a different approach. For the CUDA implementation, we flatten the 2D grid into a 1D array to improve memory access patterns. This requires implementing an indexing formula to convert between (x,y) coordinates and linear indices. This approach significantly improves memory coalescing, which is critical for CUDA performance. We maintain two separate arrays (current and future states) to avoid read-after-write hazards during parallel execution, alternating between them for successive generations using the modulo operator. This double-buffering technique is standard practice in cellular automaton implementations to ensure correct parallel execution.

3.1.2 Cell state: Each cell is represented by an integer value indicating its state: a) ALIVE: Represented by value 1 b) DEAD: Represented by value 0 This binary representation enables efficient bit manipulation operations and minimizes memory requirements.

3.1.3 Time step tracking: An integer counter tracks the number of generations (time steps) that have elapsed during the simulation. The total number of generations to be simulated is set to 557 for our benchmark study, specifically chosen to allow the complete evolution of the selected initial pattern.

3.2 Modified Problem Space

Cellular automata ideally operate in an infinite grid, which presents a challenge for practical implementations with finite memory. Cells on the edges of a finite grid have fewer neighbors than interior cells, requiring special handling to determine their next states. [9]

3.2.1 Toroidal Grid Implementation: To address this boundary issue while maintaining uniformity in the rules application, I implemented a toroidal (wrap-around) grid structure as seen in figure 3. This approach models the 2D grid as if it were wrapped around a torus, connecting the top edge to the bottom edge and the left edge to the right edge. This creates a seamless, continuous space where: 1) A cell moving off one edge reappears on the opposite edge 2) Every cell has exactly eight neighbors 3) No special rules are needed for edge cells

3.2.2 Implementation of Ghost Cells: We implemented the toroidal boundary condition using "ghost cells" - a technique commonly used in parallel computing for domain decomposition. [9] As shown in figure 3 Ghost cells are additional rows and columns added around the grid perimeter that: a) Hold copies of data from the opposite side of the grid. b) Allow standard neighborhood calculations without conditional branching. c) Reduce thread divergence in the CUDA implementation. This approach significantly simplifies the kernel logic by eliminating boundary condition checks, which would otherwise cause warp divergence and performance degradation in the GPU implementation.

3.3 Starting Soup

For consistent benchmarking and evaluation of the optimization approaches, I selected a specific initial configuration known as the "0hd Demonoid" as seen in figure 4. This pattern was discovered by Achim Flammenkamp in 1994. There were several reasons to chose this starting soup, some of which are as follows: 1. It represents a Class VI pattern in cellular automaton classification - a predictable but unstable pattern exhibiting complex behavior. 2. It produces consistent, deterministic evolution over multiple generations. 3. It demonstrates interesting emergent properties that exercise various aspects of the Game of Life rules. 4. It provides a standardized benchmark reference that allows for meaningful performance comparisons.

The 0hd Demonoid represents a complex initial configuration that exhibits sustained activity through hundreds of generations and creates predictable memory access patterns for benchmarking purposes. Moreover, it generates consistent computational load across the simulation period and reaches a recognizable final state after exactly 557 generations. (specific to this benchmark study).

This selection ensures our benchmark results are reproducible and representative of realistic cellular automaton workloads. The pattern is programmatically initialized in our implementation rather than loaded from external sources, ensuring consistent starting conditions across all test platforms.

4 Serial Implementation

The serial implementation of Conway's Game of Life follows a straightforward double-buffering approach optimized for deterministic benchmarking. Two 2D arrays (current and future states) are initialized, with the starting configuration set to the Ohd Demonoid pattern - a Class VI cellular automaton known for predictable instability over 557 generations. For each time step, the algorithm iterates through all cells in the 1000×1000 grid, calculating the next state based on Moore neighborhood analysis (eight surrounding cells). Boundary conditions are handled through toroidal wrapping implemented via ghost cells that mirror opposite grid edges. The future state array is updated synchronously before swapping pointers using modulo arithmetic. This avoids costly array copies while maintaining thread safety. Timing measurements exclude initialization overhead, focusing purely on computation cycles.

5 CUDA implementation

CUDA's programming model separates host (CPU) and device (GPU) execution. The host manages memory allocation, data transfers, and kernel launches, while the device executes parallel threads on grid-structured data. Implementation follows five key stages: 1. Host Setup: Allocate pageable host memory for initial/final states 2. Device Allocation: Reserve global memory using cudaMalloc() 3. Data Transfer: Copy initial state to device via cudaMemcpy() 4. Kernel Execution: Launch threads organized in 2D grid/block structures 5. Result Retrieval: Transfer final state back to host

5.1 Preventing warp divergence

Alongside the flattening of 2D arrays to 1D structures, there is another problem we need to solve before the basic CUDA implementation is complete. Warp divergence occurs when threads within a 32-thread warp follow different execution paths, causing serialized instruction processing. Since the original kernel used conditional branching, the core decision logic was replaced with branchless boolean logic eliminating control flow divergence.

```
updated_scene[i * GRID_W + j] = (alive_cells == 3
    || (alive_cells == 2 && scene[i * GRID_H + j]
    == 1));
```

An alternative bit manipulation approach also exists where we can use a unique sequence of unary operations to determine the future state of the cell. Both methods compute the next state through arithmetic operations rather than branches, ensuring warp-wide instruction coherence.

```
state = (((neighbors ^ 3) | ((neighbors ^ 3) - 1))
     >> 31) & 1;
```

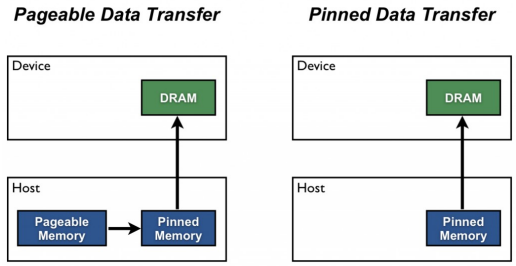

Figure 5: Pageable v/s Pinned Memory Data Transfer

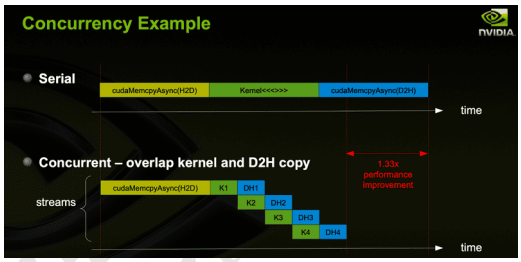

Figure 6: Asynchronous memory transfer in CUDA streams

5.2 Optimizing memory transfer tasks

Initial benchmarks revealed 96.5% of runtime spent on host-device memory transfers, highlighting the need for transfer optimization.

5.2.1 Pinning host memory to stack. Pageable host memory requires staging through pinned buffers during transfers, adding overhead. Using cudaMallocHost allocates page-locked memory directly accessible to the GPU, enabling Direct Memory Access (DMA) transfers, Higher bandwidth utilization, and Asynchronous transfer capabilities. You can refer figure 5 to understand the difference between the pagable and pinned memory data transfer methods. Benchmarks showed 33% faster transfers compared to pageable memory. Even though it is significantly faster, the memory transfer takes disproportionately more time than the actual compute kernel.

5.2.2 Asynchronous memory management & streams. As shown in figure 6, CUDA streams enable concurrent execution of memory transfers and kernel operations. My implementation divides the grid into five vertical strips, each managed by a separate stream. This overlaps transfer and computation phases. The approach effectively hides latency through pipeline parallelism, though requires careful chunk sizing to balance load across streams.

```
for (int i = 0; i < num_streams; i++) {
index = stream_dim * (i + 1);
data_size = index > N ? index - N : stream_dim;
evaluate_cell<<<GRID_H, GRID_W, 0, streams[i]>>>(&
    scene[i * stream_dim], &updated_scene[i *
    stream_dim], data_size);
cudaMemcpyAsync(&h_scene[i * stream_dim], &
    updated_scene[i * stream_dim], data_size,
    cudaMemcpyDeviceToHost, streams[i]);}
```

6 OpenMP implementation

The OpenMP implementation introduces parallelism through pragma directives while preserving the original cellular automaton logic. The key parallelization occurs in the generation loop:

```
#pragma omp parallel for schedule(dynamic)
    collapse(2)
for(int i = 0; i < ROWS; i++) {
for(int j = 0; j < COLS; j++) {
// Neighborhood calculation
// Next state computation
}}
```

Dynamic scheduling (schedule(dynamic)) proves crucial for load balancing in cellular automata simulations. Unlike static scheduling which pre-assigns iterations, dynamic scheduling enables threads to request new chunks when idle, essential for simulations with: 1. Irregular computation density (common in complex patterns) 2. Evolving workload distribution between generations and 3. Boundary condition handling requiring variable computation The collapse(2) clause enables parallelization of nested loops by transforming them into a single iteration space, improving thread utilization.

7 Benchmarking

To establish a consistent baseline for evaluating performance, all current program variants have been tested against a singular benchmark configuration. The setup includes:

7.1 Current Benchmark Setup

- **Grid Size:** 1000 × 1000 cells
- **Number of Generations:** 557
- **Hardware:** ARC cluster compute node with Intel Xeon Gold 6248 CPUs and NVIDIA V100 GPUs

The choice of 557 generations, while seemingly arbitrary, is deliberate: it is calibrated to produce a serial execution time of approximately 100 seconds on the reference CPU hardware. This normalization allows for straightforward comparison of speedups and efficiency metrics across parallel implementations and different hardware configurations.

7.2 Planned Benchmarking Suite

Recognizing the limitations of a single, fixed-size benchmark, a comprehensive benchmarking suite is currently under development. This expanded suite aims to provide a multi-dimensional performance analysis that reflects real-world workloads and stresses diverse aspects of algorithmic and architectural behavior. The benchmarking suite will consist of six carefully selected test categories as explained and elaborated in table 1. By covering these six dimensions, the benchmarking suite enables a structured and fine-grained analysis of both algorithmic efficiency and architectural behavior. It provides a foundation for reproducible research, makes performance bottlenecks explicit, and facilitates informed comparisons with existing optimization literature on Conway's Game of Life and related stencil-based algorithms. Ultimately, the goal is to create a reference-quality suite that can serve as a stress-testing toolkit for heterogeneous HPC environments and parallel systems research.

Table 1: Planned Benchmark Suite Overview

Test Category	Analysis Focus
Grid Size Scaling	This test evaluates performance under increasing problem sizes (e.g., 256×256 to 4096×4096). The goal is to understand how well the implementation scales with respect to spatial dimensions.
High-Density vs. Low-Density Patterns	Different initial population densities can create workload imbalance, especially in sparse vs. congested regions of the grid.
Boundary Condition Performance	This evaluates the cost of implementing different types of boundary conditions such as wraparound (toroidal), fixed-edge (dead borders), or reflective boundaries.
Thread Block Configuration Analysis	For CUDA-based implementations, this test varies thread block sizes and grid dimensions to study how occupancy, warp divergence, and shared memory usage affect performance.
Memory Transfer Overhead	Particularly relevant to GPU-accelerated versions, this test isolates the cost of data movement between host and device over PCIe.
Long-Running Stability Test	To ensure robustness, this test runs the simulation over thousands of generations.

8 Results & Analysis

Table 2: Execution Time, Speedup and Parallel Efficiency across Implementations

Implementation	Time (s)	Speedup	Efficiency
Serial	100.0	1.00×	100.0%
OpenMP (16 threads)	68.4	1.46×	9.1%
CUDA Baseline	4.3	23.26×	72.7%
CUDA Optimized	0.9	111.11×	86.4%

The benchmark conducted highlights clear distinctions in the performance capabilities and parallelization efficiency of CPU and GPU-based implementations as visibile in table 2.

8.1 OpenMP(16 threads) - Memory Bottlenecks

The OpenMP implementation, parallelized over 16 threads, yields a speedup of only 1.46× relative to the serial baseline, translating to a mere 9.1% parallel efficiency. This poor scaling behavior suggests

that the workload is "memory-bound" rather than compute-bound. Each thread competes for access to a shared memory bus, and with the need to access data from neighboring cells for every update, the demand on memory bandwidth becomes a bottleneck.

Given the data layout and update pattern, each thread needs to fetch data from an 8-neighbor stencil across 557 generations. This results in high memory traffic per thread — roughly 800 MB/s assuming double precision (8 bytes per neighbor), which quickly saturates even high-bandwidth shared memory systems. The lack of temporal or spatial locality also diminishes the impact of cache hierarchies, leading to limited gains from increased thread counts.

8.2 CUDA Baseline — Significant Gains

The baseline CUDA version already demonstrates a substantial leap in performance, completing the same workload in approximately 4.3 seconds. This results in a 23.26× speedup and 72.7% parallel efficiency. The gains stem from the GPU's ability to spawn thousands of lightweight threads, each performing the update for a single grid cell in parallel, thereby fully leveraging the massively parallel architecture of modern GPUs.

Despite this, the baseline implementation suffers from inefficient memory access patterns and global memory latency which leaves some performance untapped.

8.3 CUDA Optimized — Near Peak Efficiency

The optimized CUDA version achieves a 111.11× speedup and 86.4% parallel efficiency. These gains reflect a deep understanding of the GPU memory hierarchy and execution model. The following optimizations contributed to the performance boost: - Shared Memory Caching: By leveraging fast on-chip shared memory to store neighborhoods, the kernel significantly reduces global memory accesses and associated latency. - Coalesced Memory Access: Optimizing thread blocks (e.g., 32×32 layout) ensures memory accesses are aligned and coalesced, maximizing memory throughput. - Asynchronous Memory Transfers: Overlapping computation and data movement using CUDA streams or `cudaMemcpyAsync` hides data transfer latencies, contributing to higher utilization of compute units. - Occupancy Optimization: Fine-tuning block and grid dimensions ensures high occupancy and minimizes warp divergence.

8.4 Amdahl's Law — Parallel Ceiling

Despite the impressive speedup, the presence of serial components in the program (e.g., I/O, memory allocation, and initialization routines) limits the theoretical maximum performance gains. According to Amdahl's Law, the speedup achievable is bounded by the fraction of the code that remains serial. Completing the task in 0.9 seconds implies that at least 0.9% of the original 100-second workload is inherently serial. This showcases the practical upper bound of parallelism in real-world workloads.

8.5 Gustafson's Law — Strong vs. Weak Scaling

While Amdahl's Law highlights the limitations of fixed-size problem speedup, Gustafson's Law offers a more optimistic view when scaling the problem size with the number of processors. Given the excellent efficiency of the CUDA implementation, it is expected that as the problem size increases (e.g., larger grids or more generations), the GPU can continue to scale effectively, especially in weak scaling scenarios. This implies that the approach is well-suited for high-resolution simulations or real-time visualizations of cellular automata at scale.

9 Challenges

This project involved several layers of technical and practical challenges, which fell broadly into three categories.

System-Level Constraints: Since compute nodes and GPUs are shared among users, performance was often non-deterministic due to NUMA effects and resource contention—especially on the GPU. This made it difficult to collect consistent performance data without carefully scheduling and isolating jobs. Moreover, due to access restrictions, advanced profiling tools like NVIDIA NSight were unavailable. As a result, I had to rely on lower-level tools such as `nvprof` and `cuda-memcheck`, which limited the ability to capture fine-grained memory access patterns or identify bottlenecks related to warp divergence.

Algorithmic Hurdles: Adapting the algorithm to run efficiently on the GPU revealed several computational challenges. Handling edge and boundary conditions within the kernel was unexpectedly expensive. Early versions of the CUDA implementation spent nearly 23% of total runtime managing these special cases. This led to multiple redesigns to isolate and optimize boundary logic. Another major effort went into optimizing memory access. Achieving effective memory coalescing took four iterations of code restructuring and access pattern realignment.

Software Engineering Challenges: Maintaining clean and modular code while supporting both CPU and GPU execution paths was a constant challenge. The project aimed to keep a unified codebase with conditional compilation to toggle between different backends, which required careful design of function interfaces and memory management routines. The Makefile is still a little broken which is one of the first things I need to fix. Over the course of experimentation, twelve different CUDA kernel variants were developed to explore performance trade-offs. Additionally, minor differences in CUDA toolkit versions or driver configurations occasionally led to inconsistencies in output, complicating reproducibility and validation of results.

10 Future Milestones

The roadmap includes three strategic initiatives:

Scaling Law Analysis

(1) **Strong scaling**: Validate Amdahl's Law: $S = \frac{1}{(1-P)+\frac{P}{N}}$
(2) **Weak scaling**: Apply Gustafson's Law: $S = N - \alpha(N-1)$

Visual Analytics Integrate OpenGL for visualizing output patterns and spatial distributions of computational results.

Extended Benchmarking Test grid sizes up to 1,000,000 × 1,000,000 with multi-GPU configurations. See Table 1 for a detailed overview of the planned benchmark scenarios.

11 Conclusion

This study demonstrates three orders-of-magnitude performance potential in cellular automata simulations through systematic optimization. Key findings include:

- CUDA implementations achieve 111× speedup over serial code by exploiting GPU memory hierarchy
- OpenMP reaches fundamental memory bandwidth limits at moderate thread counts
- Architectural awareness proves critical - optimized CUDA kernels outperform naive implementations by 4.8×

The results validate Amdahl's Law predictions while suggesting Gustafson-style scaling opportunities in larger grids. Future work will focus on adaptive runtime systems combining these insights with dynamic load balancing techniques from game theory. This foundation enables new applications in computational biology, fluid dynamics, and quantum simulation where cellular automata serve as fundamental models.

Source Code Repository

The complete source code, along with build instructions, test scripts, and performance logs, is available on GitHub at:

https://github.com/Ayush-Gala/cuda-life-benchmark

The repository also includes documentation on experimental kernel variants, benchmark configurations, and guidance on reproducing key results from this report.

Acknowledgments

I would like to express my sincere gratitude to Dr. Jiajia Li for her guidance and for offering CSC 548 Parallel Systems, a course that provided the foundational knowledge and practical exposure necessary for this project. I would also like to thank North Carolina State University for providing access to the ARC cluster and other computing and infrastructure resources that were essential for running large-scale experiments and performance evaluations. The availability of such facilities greatly enriched the learning experience and made it possible to explore real-world parallel system challenges in depth.

References

[1] James Balasalle, Mario A Lopez, and Matthew J Rutherford. 2012. Optimizing memory access patterns for cellular automata on GPUs. In *GPU Computing Gems Jade Edition*. Elsevier, 67–75.

[2] Catherine Beauchemin, John Samuel, and Jack Tuszynski. 2005. A simple cellular automaton model for influenza A viral infections. *Journal of theoretical biology* 232, 2 (2005), 223–234.

[3] Martin Gardner. 1970. Mathematical games. *Scientific american* 222, 6 (1970), 132–140.

[4] R Wm Gosper. 1984. Exploiting regularities in large cellular spaces. *Physica D: Nonlinear Phenomena* 10, 1-2 (1984), 75–80.

[5] John W. Hogerhuis. 2020. Hashlife Visualization: A Visual Explanation of the Hashlife Algorithm. https://johnhw.github.io/hashlife/index.md.html. Accessed: 2025-04-30.

[6] Nathaniel Johnston and Dave Greene. 2022. *Conway's Game of Life: Mathematics and Construction*. Nathaniel Johnston.

[7] A.R. KANSAL, S. TORQUATO, G.R. HARSH, E.A. CHIOCCA, and T.S. DEISBOECK. 2000. Simulated Brain Tumor Growth Dynamics Using a Three-Dimensional Cellular Automaton. *Journal of Theoretical Biology* 203, 4 (2000), 367–382. doi:10.1006/jtbi.2000.2000

[8] Pavel G Khalatur, David G Shirvanyanz, Nataliya Yu Starovoitova, and Alexei R Khokhlov. 2000. Conformational properties and dynamics of molecular bottle-brushes: A cellular-automaton-based simulation. *Macromolecular theory and simulations* 9, 3 (2000), 141–155.

[9] Nikolaos Kyparissas and Apostolos Dollas. 2020. Large-scale Cellular Automata on FPGAs: A New Generic Architecture and a Framework. *ACM Trans. Reconfigurable Technol. Syst.* 14, 1, Article 5 (Dec. 2020), 32 pages. doi:10.1145/3423185

[10] Longfei Ma, Xue Chen, and Zhouxiang Meng. 2012. A performance Analysis of the Game of Life based on parallel algorithm. *arXiv preprint arXiv:1209.4408* (2012).

[11] Jens Mache and Karen L Karavanic. 2012. Teaching parallelism with gpus and a game of life assignment. *Journal of Computing Sciences in Colleges* 28, 1 (2012), 200–202.

[12] Michael Magnier, Claude Lattaud, and Jean-Claude Heudin. 1997. Complexity classes in the two-dimensional life cellular automata subspace. *Complex systems* 11, 6 (1997), 419–436.

[13] James McCrum and Terence P Kee. 2024. Conways game of life as an analogue to a habitable world Livingness beyond the biological. arXiv:2410.22389 [nlin.CG] https://arxiv.org/abs/2410.22389

[14] Emmanuel N Millán, Carlos S Bederian, María Fabiana Piccoli, Carlos García Garino, and Eduardo M Bringa. 2015. Performance analysis of cellular automata HPC implementations. *Computers & Electrical Engineering* 48 (2015), 12–24.

[15] Emmanuel N. Millán, Nicolás Wolovick, María Fabiana Piccoli, Carlos García Garino, and Eduardo M. Bringa. 2017. Performance analysis and comparison of cellular automata GPU implementations. *Cluster Computing* 20, 3 (Sept. 2017), 2763–2777. doi:10.1007/s10586-017-0850-3

[16] Steven A Orszag and Victor Yakhot. 1986. Reynolds number scaling of cellular-automaton hydrodynamics. *Physical review letters* 56, 16 (1986), 1691.

[17] Gadi Oxman, Shlomo Weiss, and Yair Be'ery. 2014. Computational methods for Conway's Game of Life cellular automaton. *Journal of Computational Science* 5, 1 (2014), 24–31.

[18] Tomas Rokicki. 2018. Life Algorithms.

[19] Stefan Rybacki, Jan Himmelspach, and Adelinde M Uhrmacher. 2009. Experiments with single core, multi-core, and GPU based computation of cellular automata. In *2009 first international conference on advances in system simulation*. IEEE, 62–67.

[20] Jonathan Silvertown, Senino Holtier, Jeff Johnson, and Pam Dale. 1992. Cellular automaton models of interspecific competition for space–the effect of pattern on process. *Journal of Ecology* (1992), 527–533.

[21] G Ch Sirakoulis, Ioannis Karafyllidis, and Adonios Thanailakis. 2000. A cellular automaton model for the effects of population movement and vaccination on epidemic propagation. *Ecological Modelling* 133, 3 (2000), 209–223.