HPP Project Report

We use `uint8_t` to minimize memory usage and improve cache efficiency, since each cell only stores a binary state. This reduces memory bandwidth pressure and improves scalability.

We use a flat 1D array instead of a pointer-based 2D structure to ensure contiguous memory layout, better cache locality, reduced pointer indirection, and improved parallel decomposition. This choice aligns with best practices for high-performance stencil computations.

Correctness Verification

To verify correctness of the serial implementation, we tested three canonical Game of Life patterns:

- Single cell (dies after one generation)
- 2×2 block (stable still life)
- Blinker (period-2 oscillator)

In all cases, the observed evolution matched the expected theoretical behavior.

Serial Optimizations and Baseline Measurements (Day 2)

After validating correctness with the serial implementation, I improved the code to make it suitable as a performance baseline for later Pthreads parallelization. The main goal was to keep the implementation simple and correct while removing avoidable overhead.

Implemented Serial Optimizations

1) Double buffering with pointer swapping (no copying)
The simulation uses two grids (`grid` for current state and `next` for the next state). Instead of copying arrays after each timestep, I swap the pointers:

```
uint8_t *tmp = grid;
grid = next;
next = tmp;
```

This avoids an `O(W·H)` memory copy every timestep and keeps the cost per timestep focused on the actual Game of Life update.

2) Cache-friendly memory layout (flat 1D array)
The grid is stored as a single contiguous `uint8_t` array (row-major). This improves spatial

locality compared to a 2D array-of-pointers, making memory accesses more cache-friendly during neighbor reads and writes.

3) High optimization compilation (`-O3 -march=native`)
All performance runs were compiled with aggressive compiler optimizations (explained below), to ensure the baseline reflects what the CPU can realistically deliver without changing the algorithm.

Timing Methodology

To establish a reliable serial baseline, I ran **100 timesteps** for each grid size and repeated each experiment **5 times**. The program reports:

- **Elapsed time (s)** for the full simulation
- **Updates/sec**, defined as:

$$\text{Updates/sec} = \frac{W \cdot H \cdot \text{steps}}{\text{elapsed time}}$$

This metric represents how many cell updates per second the serial code achieves.

The following table summarizes the runs (best and mean values across 5 repetitions):

Grid size	Steps	Best time (s)	Mean time (s)	Best updates/s	Mean updates/s
500×500	100	0.033024	0.033646	7.570×10^8	7.436×10^8
1000×1000	100	0.122238	0.128496	8.181×10^8	7.790×10^8
2000×2000	100	0.472692	0.502454	8.462×10^8	7.976×10^8

Observation: performance stays around **~7.4×10^8 to ~8.5×10^8 updates/sec**, which suggests the computation is heavily influenced by **memory access patterns** (each cell update requires reading several nearby cells) rather than expensive arithmetic.

I used `-O3` because it typically provides the best speed for CPU-bound loops without changing language rules or numerical semantics.

- `-O0`: no optimization; useful for debugging but not meaningful for performance.
- `-O1` / `-O2`: enables many optimizations, often good, but may miss more aggressive ones that benefit tight loops.
- **`-O3`**: includes `-O2` plus more aggressive transformations such as:
    - stronger **inlining**

    - more aggressive **loop optimizations** (unrolling, code motion)
    - improved opportunities for **auto-vectorization** (when safe)

- `-Os`: optimizes for size, not speed → can reduce performance.
- `-Ofast`: enables optimizations that may break strict standards/IEEE behavior (more "unsafe" assumptions). Since this project is about correctness + reproducible benchmarking, I avoided `-Ofast`.

Additionally, `-march=native` allows the compiler to generate instructions optimized for the specific CPU where the experiments are run (e.g., better vector instructions), giving a more realistic baseline for that machine.

Additional optimizations:

`fill_zero()` optimized to a single linear traversal
The grid-zeroing function was rewritten from nested `(i,j)` loops into a single loop over `total = W*H`. Since the grid is stored contiguously, linear traversal reduces index arithmetic (no `i*W + j`) and keeps memory access sequential and cache-friendly.

Serial Loop Optimization

The initial serial implementation used a generic `count_neighbors()` function to compute the number of live neighbors for each cell. This function iterated over the 3×3 neighborhood using nested loops and performed bounds checking for each potential neighbor. Although correct, this approach introduced significant overhead inside the innermost loop of the simulation.

The optimized implementation improves performance in three main ways:

1. Elimination of Bounds Checks

Instead of updating all cells and checking whether each neighbor is within bounds, the optimized version updates only interior cells (`i = 1..H-2`, `j = 1..W-2`). Border cells are explicitly set to zero at each timestep to enforce the fixed-dead boundary condition.

By restricting computation to interior cells, all neighbor accesses are guaranteed to be valid, eliminating conditional boundary checks inside the hot loop. This removes branch instructions that would otherwise execute millions of times per timestep.

2. Inlining of Neighbor Computation

The original version computed neighbors via a separate function call:

```
n = count_neighbors(grid, i, j, W, H);
```

This introduced:

- Function call overhead
- Two nested loops (`di`, `dj`)
- A branch to skip the center cell
- Bounds checks for every neighbor

In the optimized version, the eight neighbor values are summed directly inside the inner loop using fixed offsets. This removes loop overhead, function calls, and extra branches, producing a tighter and more predictable instruction sequence.

3. Reduction of Repeated Index Multiplications

In the naive implementation, index expressions such as `i * W + j` and `(i ± 1) * W` were recomputed multiple times per cell update.

The optimized version precomputes row offsets:

```
imW = (i-1) * W
iW  = i * W
ipW = (i+1) * W
```

These values are computed once per row and reused across all columns, reducing arithmetic operations inside the innermost loop.

📊 Complexity Analysis (Why Both Are O(W·H), but One Is Faster)

Both the naive and optimized implementations have the same asymptotic complexity:

$$T(W, H) = O(W \cdot H)$$

Each timestep processes all cells once, so the algorithm is linear in the number of grid elements.

However, Big-O notation hides constant factors.

Naive version cost per cell:

- Function call
- Two nested loops (3×3 neighborhood)
- 8 bounds checks
- Multiple index multiplications
- Several branch instructions

So although complexity is: $O(W \cdot H)$

the constant factor is large.

Optimized version cost per cell:

- 8 direct memory loads
- 1 conditional rule evaluation
- No bounds checks
- Reduced arithmetic

So it is still: $O(W \cdot H)$

but with a much smaller constant factor.

Since the grid size can reach millions of cells, reducing the constant factor significantly improves runtime.

In performance-critical loops, constant-factor reduction is often more important than asymptotic complexity.

1. Measured Performance Summary

500×500 (100 steps)

| Version | Best Time (s) | Mean Time (s) | Best Updates/sec |

| --- | --- | --- | --- |

| Original | 0.033024 | 0.033646 | 7.57e8 |

| Optimized | 0.005861 | 0.006787 | 4.27e9 |

Speedup (best case):

[

S = \frac{0.033024}{0.005861} \approx 5.63\times

]

1000×1000

| Version | Best Time (s) | Mean Time (s) | Best Updates/sec |

| --- | --- | --- | --- |

| Original | 0.122238 | 0.128496 | 8.18e8 |

| Optimized | 0.026820 | 0.027823 | 3.73e9 |

Speedup:

[

S = \frac{0.122238}{0.026820} \approx 4.56\times

]

2000×2000

Version	Best Time (s)	Mean Time (s)	Best Updates/sec
Original	0.472692	0.502454	8.46e8
Optimized	0.088200	0.099076	4.54e9

Speedup:

[
S = \frac{0.472692}{0.088200} \approx 5.36\times
]

Key Observation

Your update rate improved from:

~ **8×10^8 updates/sec**

to

~ **4×10^9 updates/sec**

That’s roughly a **5× improvement**.

That is a **huge gain**, especially since asymptotic complexity did not change.

Performance Results and Analysis

After applying the serial optimizations (removal of bounds checks, inlining neighbor computation, and reduction of repeated index multiplications), performance improved significantly across all grid sizes.

For a 2000×2000 grid with 100 timesteps:

* Original version: 0.4727 s (best)

* Optimized version: 0.0882 s (best)

This corresponds to a speedup of approximately:

[

S \approx 5.36\times

]

Similar improvements were observed for smaller grid sizes, with speedups ranging between 4.5× and 5.6×.

Although both implementations have the same asymptotic time complexity:

[

T(W,H) = O(W \cdot H)

]

the optimized version dramatically reduces the constant factor.

The original implementation incurred significant overhead per cell update due to:

* Function call overhead (`count_neighbors`)
* Nested loops over the 3×3 neighborhood
* Bounds checks for every neighbor
* Repeated index multiplications (`i*W + j`)

The optimized implementation eliminates these costs by:

* Updating only interior cells (removing bounds checks entirely)
* Inlining neighbor summation using fixed offsets
* Precomputing row offsets once per row
* Using a flat contiguous memory layout

The measured improvement of approximately 5× confirms that constant-factor optimization in performance-critical loops can have a dramatic effect, even when asymptotic complexity remains unchanged.

Grid Size	Version	Best Time (s)	Mean Time (s)	Best Updates/sec
500×500	Original	0.033024	0.033646	7.57e8
	Optimized	0.005861	0.006787	4.27e9
1000×1000	Original	0.122238	0.128496	8.18e8
	Optimized	0.026820	0.027823	3.73e9
2000×2000	Original	0.472692	0.502454	8.46e8
	Optimized	0.088200	0.099076	4.54e9

The optimized implementation achieves a consistent speedup between 4.5× and 5.6× across all tested grid sizes. The improvement confirms that eliminating bounds checks, reducing arithmetic overhead, and inlining neighbor computation significantly reduce the constant factor of the O(W·H) algorithm.

Transition to Parallelization

After optimizing the serial implementation, the next step is to exploit thread-level parallelism using Pthreads. The optimized version significantly reduced constant-factor overhead (bounds checks, function calls, and repeated index computations), achieving around

4×10^9

4×109 updates per second on large grids. With most computational overhead removed, performance is now largely limited by memory access. Therefore, parallelization aims to distribute the grid updates across multiple CPU cores, while synchronization is required only at the end of each timestep for pointer swapping. Using this optimized serial baseline ensures that measured speedups reflect true parallel gains rather than improvements from serial inefficiencies.

Synchronization Strategy

Mutexes are not required in the parallel implementation because shared writes are avoided by design. During each timestep, all threads read from the current grid (`grid`) and write results to a separate buffer (`next`). The output grid is partitioned by rows so that each thread updates a distinct, non-overlapping region, eliminating data races.

Synchronization is needed only at the end of each timestep to ensure all threads finish computing before the grid pointers are swapped. This is handled using `pthread_barrier_t`, which guarantees correct ordering without the overhead of lock-based synchronization.