# Game of Life: Performance Optimization with CUDA Parallelization

**Lorenzo Cappetti**

January 2026


UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI FIRENZE

# Table of Contents

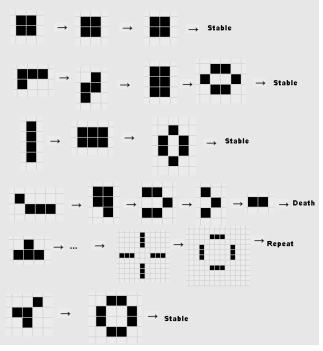
# Conway's Game of Life (GoL)

## 1 Introduction

- 2D cellular automaton on a grid of cells.
- Each cell has two states: **alive (1)** / **dead (0)**.
- **Deterministic** evolution based on the Moore neighborhood: **8 neighbors**.

### Why parallelization makes sense

- Each cell update depends only on its **local neighbors** $\rightarrow$ the same computation is repeated across the whole grid.
- Each step is a **2D stencil**-like operation (8-neighbors): many cells can be updated **in parallel**.
- The problem is typically **memory-bound**: it benefits from high throughput and bandwidth (GPUs).




Example pattern

# Update rule ($t \to t+1$)



## State and neighborhood

- Cell state: $C_t(i,j) \in \{0,1\}$
- Moore neighborhood: 8 cells around $(i,j)$

## Counting live neighbors

$$N_t(i,j) = \sum_{\substack{dx,dy \in \{-1,0,1\} \\ (dx,dy) \neq (0,0)}} C_t(i+dx,\ j+dy)$$

## Synchronous update

$$C_{t+1}(i,j) = \begin{cases} 1 & \text{if } N_t(i,j) = 3 \quad \text{(birth)} \\ C_t(i,j) & \text{if } N_t(i,j) = 2 \quad \text{(survives)} \\ 0 & \text{otherwise} \quad \text{(death)} \end{cases}$$

- **Boundary conditions**: toroidal grid (wrap-around) $\to$ edges "wrap" to the opposite side.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Table of Contents

# Project approach: 3 implementations



- **CPU baseline (Python)**:
  - end-to-end vectorization (NumPy/SciPy).
  - neighbor counting reformulated as matrix operations.
  - measurement tweaks: GC disabled for stable timings.
- **CUDA (GPU)**:
  - 2D grid of blocks/threads.
  - 1 thread $\rightarrow$ 1 cell.
- **Interactive version (Python + Pygame)**:
  - target 60 FPS on small grids ($\leq 200 \times 150$).
  - useful for debugging / educational demos.
  - not suitable for large benchmarks (overhead + rendering).

# CUDA: parallelization and key optimizations

## 2 Approach

- **Mapping:** 1 thread $\rightarrow$ 1 cell $(i, j$ from block/thread indices).
- **Double buffering (ping–pong):**
  - all threads read $C_t$ (read-only) and write $C_{t+1}$.
  - avoids race conditions, synchronous update per generation.
- **Main issue:** each cell requires many reads $(3 \times 3) \Rightarrow$ a **memory-bound** kernel.

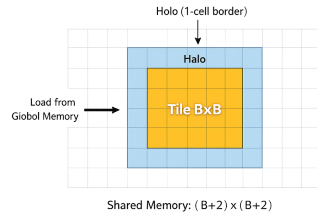
### Optimizations

- **Shared-memory tiling + halo**.
- **Coalesced accesses** and **read-only cache** (`ldg`).
- **Padding** to reduce *bank conflicts* + `__syncthreads()`.

# Main optimization

## 2 Approach

- **Problem:** without tiling, each thread reads from global memory almost the same neighbors as adjacent threads.
- **Idea:** the block loads *only once* a $(B+2) \times (B+2)$ tile (including the halo).
- **Phase 1 (cooperative load):**
  - each thread loads one (or a few) cells into the tile.
  - including the halo avoids global-memory accesses when computing the block borders.
- **Phase 2 (compute):** count neighbors reading **only** from shared memory.
- **Synchronization:** `__syncthreads()` between load and compute (intra-block only).



UNIVERSITÀ DEGLI STUDI FIRENZE

# Table of Contents

3 Results

# Benchmark setup

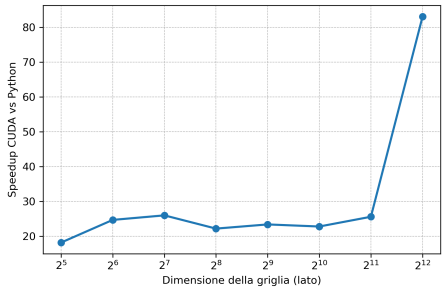
## 3 Results

- **Tested grids**:
  - Small: $32^2,\ 64^2,\ 128^2$
  - Medium: $256^2,\ 512^2$
  - Large: $1024^2,\ 2048^2$
  - Extra-large: $4096^2$
- **CUDA block size**: $1\times1,\ 4\times4,\ 8\times8,\ 16\times16,\ 32\times32$
- **Procedure**:
  1. Warm-up: 2 runs;
  2. Measurement: 10 independent runs;
  3. Metrics: mean, std, median, min/max, coefficient of variation;
  4. Timing: CUDA events (GPU), `perf_counter` (CPU);

# Results: CUDA speedup vs Python (NumPy)



- Main trend:
  - small grids $\rightarrow$ speedup limited by **kernel launch overhead**.
  - large grids $\rightarrow$ speedup increases up to the maximum.
- Key numbers:
  - minimum observed: $\geq 18\times$ ($32\times32$).
  - maximum: **83**$\times$ on $4096\times4096$.
- Interpretation: more work per kernel $\rightarrow$ better GPU saturation

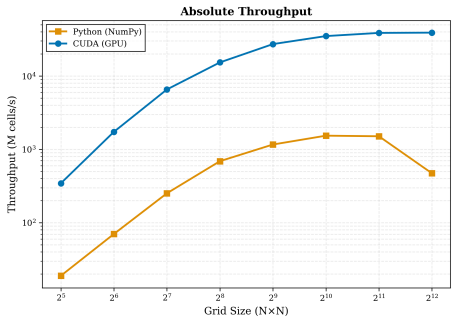



CUDA vs Python speedup as a function of $N \times N$ (log2 $x$-axis; 100 generations, BS=16 $\times$ 16).

# Results: Absolute throughput (Gcells/s)

3 Results

- Metric: cells updated per second (baseline-independent).
- Key results:
  - CUDA: peak **38.96 Gcells/s** on $4096 \times 4096$.
  - Python (NumPy): maximum **1.54 Gcells/s**.
- Trend:
  - GPU increases and then reaches a **plateau** $\rightarrow$ hardware saturation (bandwidth).
  - Python improves at first but **degrades** at the largest sizes (memory/L3 pressure).

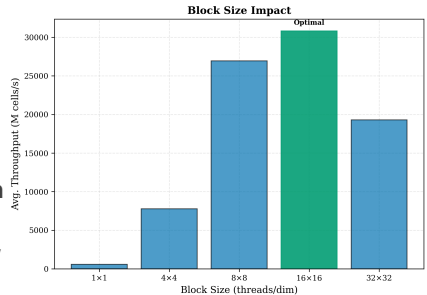



Absolute throughput: Python (NumPy) vs CUDA as $N \times N$ increases (Gcells/s).

# Optimal block size

## 3 Results

- Varying $B \times B$ with a **fixed grid size**.
- Throughput peaks at $16 \times 16$ $\rightarrow$ **best trade-off** between parallelism and resource usage.
- Explanation:
  - good number of **resident blocks per SM** $\Rightarrow$ **high occupancy** and memory-latency hiding.
  - layout favorable for **coalesced** accesses and low **warp divergence**.
  - $32 \times 32$ (1024 threads) $\rightarrow$ more register/resource pressure, **fewer concurrent blocks** $\Rightarrow$ throughput drop.




Average throughput vs block size: increases up to $16 \times 16$, then drops at $32 \times 32$ (occupancy/resources).

UNIVERSITÀ DEGLI STUDI FIRENZE

# Table of Contents

4 Conclusions

# Conclusions



- **Main results**
  - Speedup up to **83**$\times$ on large grids ($4096 \times 4096$).
  - Peak throughput **38.96 Gcells/s** (plateau due to HW saturation).
  - Best configuration: $16 \times 16$ **block size** (256 threads).
- **Why it works (and what limits performance)**
  - The kernel is **memory-bound**: intensity $\approx$ **1.1 ops/byte**.
  - Parallel efficiency **3–8%**: the bottleneck is the memory hierarchy (not compute).
  - Key optimizations (`ldg`, shared-memory padding, coalescing) provide a cumulative gain of $>$**60**$\times$.
- **Take-away**
  - For stencils/local rules, **good memory handling** (reuse, coalescing, occupancy) matters more than “more FLOPs”.

UNIVERSITÀ DEGLI STUDI FIRENZE

# Lessons learned

## 4 Conclusions

- **Locality** $>$ **compute**: shared memory + data reuse beat “arithmetic” optimizations.
- Performance depends on **access patterns**: coalescing, bank conflicts, resource pressure.
- **Block size is a trade-off**: $16 \times 16$ balances occupancy and resources, $32 \times 32$ degrades due to fewer resident blocks.
- **Experimental validation** is essential: warm-up, multiple runs, statistics (mean/std/median).

UNIVERSITÀ DEGLI STUDI FIRENZE

Game of Life: Performance Optimization with CUDA Parallelization

*Thank you for listening!*

UNIVERSITÀ DEGLI STUDI FIRENZE

# References

## 4 Conclusions


M. Gardner,
"Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game "Life","
*Scientific American*, 223(4), 1970.

J. H. Conway,
*The Game of Life*.

NVIDIA,
*CUDA C++ Programming Guide*,
NVIDIA Developer Documentation.

NVIDIA,
*CUDA C++ Best Practices Guide*,
NVIDIA Developer Documentation.

C. Cappetti,
*Game-of-Life (CUDA): code and benchmarks*,
GitHub repository: `https://github.com/Cappetti99/Game-of-Life`.