Game of Life:
Performance Optimization with CUDA Parallelization

Lorenzo Cappetti
Department of Information Engineering
University of Florence
lorenzo.cappetti@unifi.it

Abstract

This work presents the development and analysis of a project on Conway's Game of Life*, with particular attention to **performance optimization** through **GPU parallelization**. Three implementations were developed: a sequential CPU version in Python (**NumPy/SciPy**), a parallel **CUDA** version, and an interactive version for **real-time visualization**.*

*The CUDA implementation leverages the GPU's **massive parallelism** and optimization techniques such as **tiling** with **shared memory**, **coalesced global-memory accesses**, use of the **read-only cache**, and an optimal choice of **block size**. Experimental results show that the GPU version achieves a **speedup** of up to $83\times$ over the sequential version, with a peak **throughput** of about 38 **Gcells/s** on 4096×4096 grids.*

*The performance analysis highlights that the workload is predominantly **memory-bound**, limiting **parallel efficiency** despite the large performance gain. The project demonstrates how the Game of Life provides an effective case study for the analysis and optimization of **stencil algorithms** on **GPU architectures**.*

1. Introduction

1.1. Background and Motivation

The Game of Life, conceived by the mathematician John Horton Conway in 1970, is one of the most widely studied cellular automata in computer science. Despite the simplicity of its rules, the system exhibits complex **emergent behavior** and is **Turing-complete** [1]. Each cell in a two-dimensional grid can be in one of two states (alive or dead) and evolves according to the number of live neighbors through four deterministic rules.

From a computational perspective, the Game of Life is a paradigmatic example of **stencil computation**, a fundamental class of algorithms in:

- **Computational fluid dynamics** (CFD);
- **Image processing** (convolution);
- Solving **partial differential equations**.

Stencil computations are characterized by a highly regular and local data-access pattern, making them well-suited for GPU acceleration thanks to:

1. High **memory bandwidth**;
2. **Massive parallelism** (thousands of concurrent threads);
3. Memory hierarchies optimized for **data reuse**;
4. **Coalesced global-memory accesses**.

1.2. Goals of This Work

This study aims to:

- Implement two optimized versions of the Game of Life using different computational paradigms;
- Quantitatively analyze performance through systematic **benchmarks**;
- Identify the limiting factors (**memory** vs. **compute**);
- Evaluate the impact of architecture-specific optimizations.

1.3. Contributions

The main contributions of this work are:

1. A highly optimized **CUDA** implementation;
2. An analysis of the impact of **block size** (from 1×1 up to 32×32);
3. A study of **strong/weak scalability** as the grid size varies;
4. A complete benchmarking infrastructure with **statistical averaging**.

2. Game of Life Rules

2.1. Formal Definition

Let $C_t(i,j) \in \{0,1\}$ be the state of the cell at position (i,j) at time t, where 0 denotes dead and 1 alive. The evolution is governed by the function:

$$C_{t+1}(i,j) = f(C_t(i,j), N_t(i,j)) \tag{1}$$

where $N_t(i,j)$ is the number of live neighbors in the Moore neighborhood (**8-connectivity**):

$$N_t(i,j) = \sum_{\substack{dx,dy \in \{-1,0,1\} \\ (dx,dy) \neq (0,0)}} C_t(i+dx, j+dy) \tag{2}$$

2.2. Transition Rules

The function f implements the following rules:

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

In words:

1. **Birth**: a dead cell with exactly 3 live neighbors becomes alive
2. **Survival**: a live cell with 2 or 3 live neighbors remains alive
3. **Death by underpopulation**: a live cell with <2 neighbors dies
4. **Death by overcrowding**: a live cell with >3 neighbors dies

2.3. Boundary Conditions

In this work we adopt **toroidal** (wrap-around) boundary conditions:

$$C_t(i \bmod H, j \bmod W) \tag{4}$$

where H and W are the grid height and width. This removes **edge effects** and preserves the periodicity of **oscillating structures**.

3. Sequential Implementation

3.1. Algorithmic Strategy

The baseline Python implementation was designed to maximize **single-core performance**, mitigating interpreter overhead through a **fully vectorized** approach based on scientific libraries (*NumPy* and *SciPy*). The strategy is articulated around the following key points:

- **End-to-end Vectorization**: The design completely avoids iterative loops over individual cells, delegating computation to optimized **C routines** that operate on whole matrices.
- **Mathematical Reformulation**: The neighbor-counting problem was recast as a **2D convolution** with **toroidal boundary conditions** (wrap). By applying a 3×3 kernel, the neighborhood map is obtained in a single matrix pass, drastically reducing overhead compared to direct per-cell memory access.
- **Mask-Based Logic**: The state update does not rely on conditional branches (`if/else`), but exploits **bitwise logical operators**. The birth and survival rules generate boolean masks that are combined to produce the next generation simultaneously.

3.2. Optimizations and Performance Analysis

To ensure rigorous benchmarking and minimize bottlenecks, specific technical optimizations were adopted:

- **Algorithmic Acceleration (SciPy)**: Using `scipy.ndimage.convolve` provides a significant **speedup** (2–3×) over methods based on memory shifts (`np.roll`), thanks to an underlying **C implementation** that maximizes **cache locality**.
- **Timing Determinism**: The Garbage Collector (GC) is programmatically disabled during measurement phases to eliminate variance introduced by automatic memory management and to ensure **reproducible results**.
- **Instruction-Level Parallelism (SIMD)**: Despite single-thread execution, using compact data types (`uint8`) allows the CPU to exploit **vector instructions** (SIMD), processing multiple cells per clock cycle.

4. CUDA Implementation

4.1. Architecture and Memory Management

The parallel GPU implementation exploits the **massively parallel** nature of the CUDA architecture to compute the evolution of the cellular automaton. The game grid is spatially decomposed into two-dimensional blocks, where each thread is responsible for updating a single cell.

To mitigate **global-memory latency**, the kernel adopts a **spatial tiling** strategy assisted by **shared memory**. Each CUDA block loads not only the portion of the grid it owns (the **tile**), but also the adjacent perimeter cells (**halo** or **ghost cells**) required to compute neighbors along the block boundaries. Mathematically, for a block of size $B \times B$, a

shared-memory tile of size $(B+2)\times(B+2+1)$ is allocated. This approach drastically reduces **off-chip** global-memory accesses, since data are read once from **VRAM** and then reused on-chip through **L1 cache/shared memory**.

Data loading is performed cooperatively:

1. Each thread loads the cell corresponding to its position in the central tile.
2. Threads located at the block boundaries (local indices 0 or $DIM-1$) cooperatively load the halo rows and columns from neighboring blocks, correctly handling **toroidal wrap-around** at the global-grid borders.
3. An intra-block **synchronization barrier** ensures that the entire tile is populated before the computation phase starts.

Once data are in on-chip memory, neighbor counting and the application of Conway's rules are performed by accessing **shared memory only**, maximizing arithmetic **throughput**.

4.2. Implemented Optimization Techniques

The implementation integrates several advanced techniques to maximize hardware utilization:

1. **Exploiting the Read-Only Data Cache**: Global-memory accesses for loading the current grid are performed via dedicated intrinsic instructions (`__ldg`). This signals the hardware to use the separate **texture/read-only cache** rather than the standard L1 cache. Since generation-t data are immutable while computing generation $t+1$, this reduces pressure on **L1 cache coherence** and provides higher effective bandwidth for spatially localized access patterns.
2. **Avoiding Bank Conflicts (Shared-Memory Padding)**: GPU shared memory is organized into **memory banks** that can be accessed in parallel. However, if multiple threads within the same warp access different addresses that map to the same bank, the access is serialized (**bank conflict**). To avoid this effect during column-wise accesses (common in neighbor computations), the shared-memory array is sized with an extra **padding column**. This modified **stride** makes vertically adjacent cells fall into different banks, preserving full parallelism for memory accesses.
3. **Coalesced Global-Memory Accesses**: Thread and block indexing are designed to preserve spatial locality. Consecutive thread IDs (within the same **warp**) access contiguous and aligned memory addresses. This enables the GPU memory controller to **coalesce** individual loads into a minimal number of high-bandwidth transactions (e.g., 128 bytes), efficiently saturating the memory bus.
4. **Double Buffering (Ping-Pong)**: To avoid **race conditions** without expensive locks, the simulation uses two buffers permanently allocated in global memory. At each time step, pointers to the **current** and **next** buffers are swapped (the **ping-pong** technique). This removes the need for explicit memory copies between generations: the output of generation t becomes the **zero-copy** input for generation $t+1$.

4.3. Visual (Interactive) Implementation

The interactive version is developed in Python using the **Pygame** library for graphical rendering and **NumPy** for the update logic. The main features include:

- **Rendering**: Smooth **real-time** visualization (target 60 FPS) with dynamic updates of the game surface.
- **User Input**: Full interaction via mouse to draw/erase cells and keyboard shortcuts to control the simulation (pause, reset, speed).
- **Advanced Features**: Quick insertion of complex patterns (e.g., *glider*, *glider gun*) and **zoom/pan** tools to explore the grid.
- **Optimization**: Use of vectorized NumPy operations (`np.roll`) for neighbor computation, avoiding slow native Python loops.

This implementation is designed to maximize **responsiveness** and **usability** for educational and debugging purposes. Although NumPy provides good performance for small grids ($\leq 200 \times 150$), the main bottleneck remains CPU–GPU data transfer for rendering and **Python interpreter overhead**, making this version unsuitable for large-scale benchmarking compared to the CUDA counterpart.

5. Experimental Setup

5.1. Hardware Platform

Component	Specification
GPU	NVIDIA GeForce RTX 4060 Ti
CUDA Cores	4352
Bandwidth	288 GB/s
VRAM	16 GB GDDR6
CPU	AMD Ryzen 5 7600X
Cores/Threads	6/12
Frequency	Up to 5.4 GHz
RAM	32 GB DDR5-4800
Software	CUDA 12.0, Python 3.11
	NumPy 1.24, SciPy 1.10

Table 1. Hardware and software configuration

5.2. Benchmark Protocol

Tested Grid Sizes

- Small: 32×32, 64×64, 128×128;
- Medium: 256×256, 512×512;
- Large: 1024×1024, 2048×2048;
- Extra-large: 4096×4096.

Tested CUDA Block Sizes 1×1, 4×4, 8×8, 16×16, 32×32

Measurement Procedure

1. **Warm-up**: 2 runs to initialize caches/drivers;
2. **Measurement**: 10 independent runs for **statistical averaging**;
3. **Metrics**: mean, standard deviation, median, min/max, coefficient of variation;
4. **Timing**: **CUDA events** for the GPU, `perf_counter` for the CPU.

6. Experimental Results

6.1. Speedup Analysis

Table 2. CUDA vs Python Speedup (100 generations, BS=16)

Grid	Python (ms)	CUDA (ms)	Speedup
32×32	5.41	0.30	18.2×
64×64	5.84	0.24	24.3×
128×128	6.51	0.25	26.0×
256×256	9.50	0.43	22.2×
512×512	22.52	0.96	23.4×
1024×1024	68.25	3.00	22.8×
2048×2048	278.10	10.86	25.6×
4096×4096	3576.37	43.07	**83.0**×

Key observations:

- The **speedup** increases with grid size;
- For small grids, **kernel-launch overhead** limits the benefits;
- Maximum speedup: **83**× on 4096×4096;
- The minimum speedup is at least $18\times$ across all sizes.

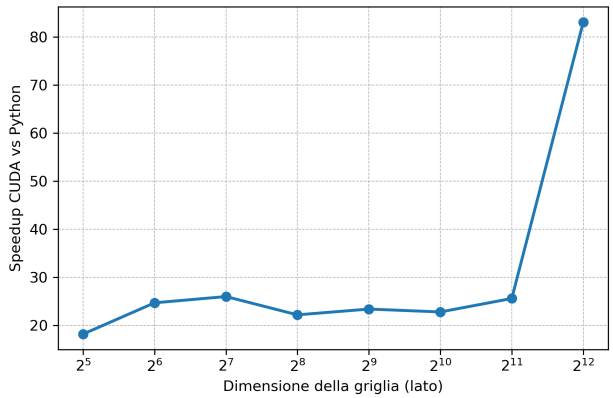

Figure 1. CUDA vs Python speedup as a function of grid size

Figure 1 shows the speedup achieved by the CUDA implementation relative to the Python version as the grid size changes. The x-axis is on a logarithmic scale (base 2) to reflect the exponential growth of the problem size. For small grids, speedup is constrained by **kernel-launch overhead**, whereas for large grids a substantial increase is observed, with a maximum of about $83\times$ for the 4096×4096 grid.

6.2. Absolute Throughput

The **CUDA** throughput reaches a peak of **38.96 Gcells/s** on 4096×4096 grids, while Python saturates at 1.54 Gcells/s. Python efficiency degrades on very large grids due to increased **L3-cache pressure**.

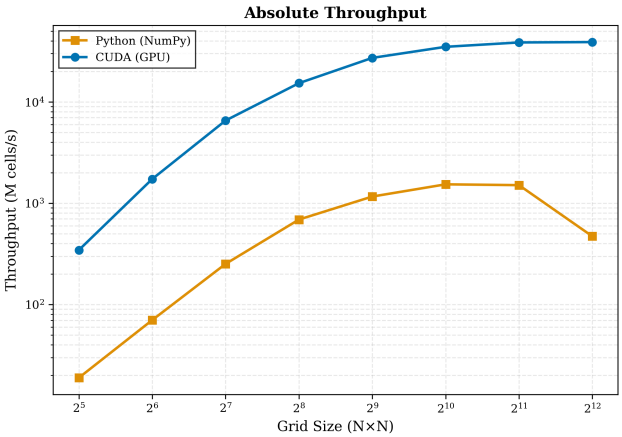

Figure 2. Comparison of absolute throughput between *Python (NumPy)* and *CUDA (GPU)* as the grid size $N \times N$ increases.

Figure 2 shows absolute throughput as a function of grid size, comparing Python (NumPy) and CUDA on the GPU. As N increases, the GPU scales much better, with a rapid throughput increase up to a **plateau** for large grids, indicating **hardware saturation**. Python improves initially, but soon reaches a maximum and then worsens for the largest

sizes, likely due to **memory limits** and computational overhead. Overall, the GPU delivers performance that is one to two orders of magnitude higher, especially for large-scale problems.

6.3. Impact of CUDA Block Size on Performance

CUDA block size plays a crucial role in determining the overall performance of a parallel kernel. To analyze its impact, several block configurations were considered while keeping the global grid size fixed.

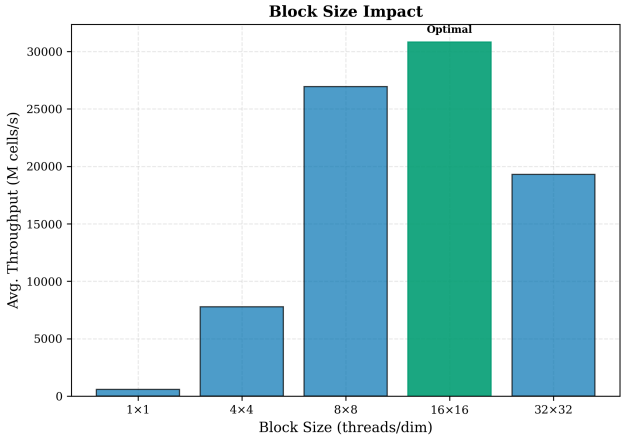

Figure 3. Impact of CUDA block size on average throughput.

As shown in Figure 3, throughput increases rapidly when moving from very small blocks to intermediate block sizes. In particular, a significant improvement is observed when going from the 1×1 and 4×4 configurations to 8×8 blocks, reaching peak performance with the 16×16 configuration. For larger blocks (32×32), throughput decreases, indicating a loss of efficiency due to a less balanced use of hardware resources.

Quantitative results are reported in Table 3, which shows the average throughput and the relative **speedup** with respect to the 1×1 reference configuration.

Block Size	Threads/ Block	Throughput (Mcells/s)	Speedup vs 1×1
1×1	1	620	$1.0\times$
4×4	16	7800	$12.6\times$
8×8	64	27000	$43.5\times$
16×16	**256**	**31000**	**50.0**$\times$
32×32	1024	19500	$31.5\times$

Table 3. Effect of CUDA block size on average throughput.

6.3.1 Occupancy Analysis

The observed behavior can be explained by analyzing GPU **occupancy**, defined as:

$$\text{Occupancy} = \frac{\text{Active warps}}{\text{Maximum warps per SM}} \tag{5}$$

For the 16×16 block, each block contains 256 threads, i.e., 8 warps. This configuration allows a good number of resident blocks per **Streaming Multiprocessor** (SM), ensuring high occupancy and an effective ability to **hide memory-access latency**. Moreover, the thread layout promotes **coalesced accesses** and low **warp divergence**.

In contrast, the 32×32 configuration uses 1024 threads per block (32 warps), reducing the number of concurrent blocks per SM and increasing pressure on registers and other hardware resources. This leads to lower effective occupancy and, consequently, a drop in overall throughput, as highlighted in both Figure 3 and Table 3.

In conclusion, the 16×16 configuration represents the best trade-off between **parallelism**, **resource utilization**, and **latency hiding**, resulting in the most efficient solution among those analyzed.

7. Conclusions

7.1. Main Results

This work demonstrated that:

1. The optimized **CUDA** implementation achieves up to $83\times$ **speedup** on large grids (4096×4096);
2. The optimal **block size** is 16×16 (256 threads), balancing **occupancy** and **resource usage**;
3. The workload is strongly **memory-bound** (intensity $\approx$ **1.1 ops/byte**), with **parallel efficiency** of 3–8%;
4. The key optimizations (`__ldg`, **shared-memory padding**, **coalescing**) provide a cumulative improvement of $> 60\times$.

7.2. Future Work

Possible extensions include:

- **Bit-packing**: storing 32 cells per word to reduce the memory footprint by $32\times$;
- **Multi-GPU**: domain decomposition for grids $> 8192 \times 8192$;
- **Warp intrinsics**: using `__ballot_sync` for more efficient neighbor counting;
- **Pattern detection**: automatic recognition of stable/periodic structures;
- **3D Game of Life**: extension to volumetric cellular automata.

7.3. Final Remarks

The Game of Life is an excellent case study for learning GPU optimization techniques. Despite its apparent simplicity, the analysis reveals the complexity of optimizing **memory-bound workloads**, where raw compute throughput is less important than efficiently managing the **memory hierarchy**.

The obtained results align with theoretical expectations and highlight the importance of:

- Quantitative analysis of **access patterns**;
- Optimizing **data locality**;
- Balancing **parallelism** and **resource usage**;
- Experimentally validating design choices.

This work provides a complete and reproducible benchmarking infrastructure for cellular automata, which can serve as a foundation for further research on **stencil algorithms** and **massively parallel** computations.

References

[1] Gardner, Martin. *Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'*. Scientific American, 223(4), 120-123, 1970.

[2] NVIDIA Corporation. *CUDA C Programming Guide*. Version 13.0, 2024.

[3] NVIDIA Corporation. *CUDA Samples: Game of Life*. GitHub repository, 2024.

[4] GitHub repository, `https://github.com/Cappetti99/Game-of-Life` *Game-of-Life*. GitHub repository, 2026.