Simulating Conway's game of life with CUDA

Marco Wang **ShaoBo Zhang**

Introduction

In this report, we explore CUDA implementations that speed up Conway's Game of Life, a well-known cellular automaton simulation. The Game of Life is a simple yet fascinating simulation that exhibits complex emergent behaviors, making it a popular choice for exploring the dynamics of complex systems. The sheer number of computations required to simulate a moderately sized world with CPU can be a significant bottleneck in either speed or memory needed. In this context, we want to leverage the power of parallel computing capabilities of CUDA to potentially provide significant performance gains and enable faster / more extensive simulations. We present a detailed analysis of our implementation and provide insights into the key factors that impact performance. Lastly, our experiment revolve around comparing different levels of optimization with different CUDA-based implementation and the traditional CPU-based implementations

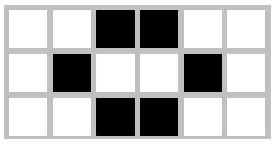
Background

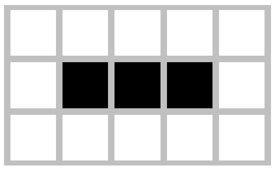
Conway's Game of Life (GoL) is a type of Cellular automata (CA), discrete mathematical models that exhibit complex behaviors through simple local interactions between cells arranged in a regular grid. Game of Life was first introduced by John Conway in 1970. The game consists of a grid of cells that can be either "alive" or "dead," and the state of each cell evolves over time according to a set of rules based on the states of its neighboring cells. Despite its simplicity, the Game of Life has been a topic of study in diverse fields, such as physics, mathematics, biology, and computer science. It has been used to model various phenomena, including population dynamics, disease spread, and self-organization in complex systems.

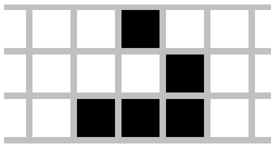
There are four rules in Conway's Game of Life. At each generation, we update all cell state according to these rules:

- A dead cell with exactly three live neighbors becomes a live cell (birth)
- A live cell with two or three live neighbors survives to the next generation (survival)
- A live cell with fewer than two live neighbors dies of loneliness (underpopulation)
- A live cell with more than three live neighbors dies of overcrowding (overpopulation)

In this context, we will introduce some basic patterns of Conway's Game of Life to familiarize you with the rules.


Stable:
Does not change over time


Blinker:
Oscillates between two states


Glider:
Glides towards bottom right

Cyclic Representation

Since computers do not have infinite memories to represent the infinite grids in Conway's Game of Life, we decided to wrap the grids around itself to create a cyclic world that still represents the simulation in a meaningful way. This proved to be extremely costly in memory access later on, but we decided to stick with this because it made the simulation feel more realistic.

There are three scenarios for the cyclic world representation: middle, edge and corner. Middle pieces (figure 1) are quite straight forward. We simple find the surrounding eight neighbors and update its cell state. Edge pieces (figure 2) are less straight forward, we need to find the wrap around piece either to the top or bottom. Corner pieces (figure 3) are the least convenient, requiring us to find the wrap around in all four corners.

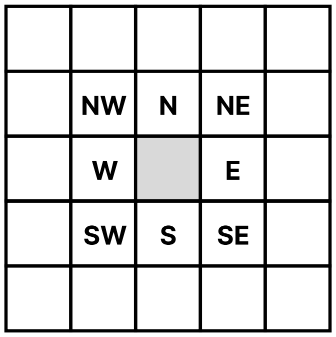


figure 1

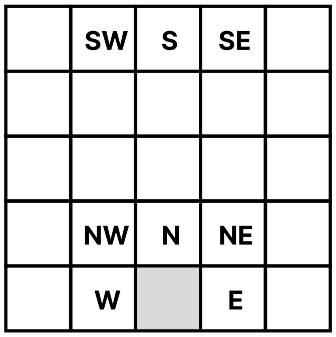


figure 2

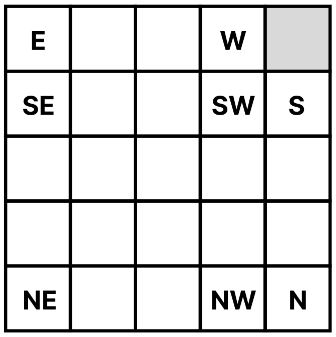


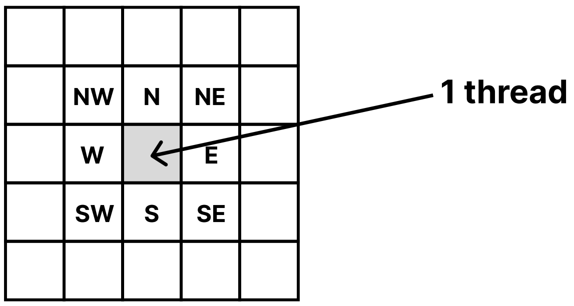
figure 3

CPU Benchmark

To test against our CUDA versions, we first created a brute-force CPU version of Conway's Game of Life simulation. We would flatten the array and loop through the entire grid. At each cell, we would check all of its neighbors and update its state to see if it's alive or dead. We then store the updated state of the cell in a new array and update the entire grid once the entire iteration is finished. We run this for a designated number of iterations and test it against other implementations

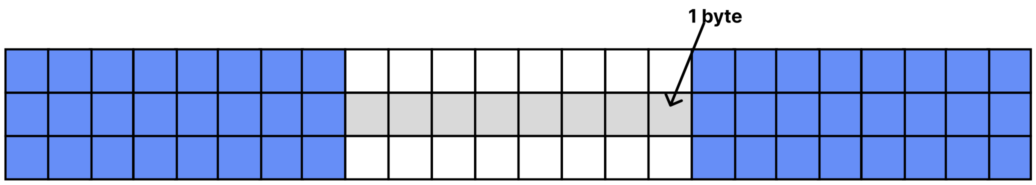
GPU Benchmark

We then began parallelizing the brute force CPU implementation by naively parallelizing the code. Instead of running a for loop that checks every cell sequentially, we simply let each thread represent a cell in the grid and update its state in parallel. We then return the new grid after the iteration is finished.

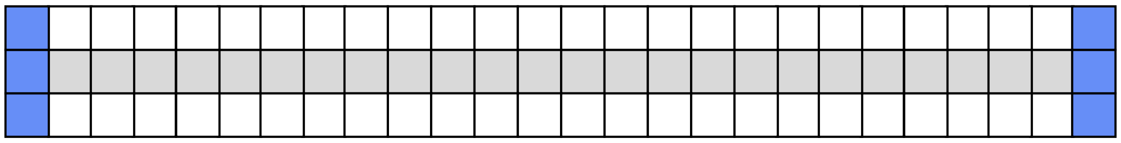


Bit Representation

The problem with the brute force GPU implementation is that we are accessing memory and using threads in a very inefficient way. We store the state in an unsigned 32-bit integer, making it extremely inefficient the store the state since the state is binary (0 or 1). We also are using one thread to access and change one cell at a time. For larger worlds, this would require the kernel to launch a lot more threads than there are SMs to process. Therefore, a more memory-dense representation will allow the threads to do more work in less memory accesses. We first decided to store the world state in bytes, with each cell stored in one bit.



According to this figure, we can see that a single byte (in grey) contains eight different cells, and we would need six bytes on the side and two from top/bottom to update the entire byte's cell state. However, as we scale the amount of work we assign to a single thread, we can read in bytes sequentially from left to right. We will only need to read in an additional three bytes on the right to evaluate a new byte. Thus, there is a lower overhead per evaluating each new byte. As we can see from the following figure, we'd still need only 6 bytes of data on the sides to update all the grey cells.



A problem we quickly realized from this method is the fact that if we assign too many bytes to a single thread, it will stall the GPU since each thread is doing more work, and the algorithm becomes less parallelized. To balance the tradeoff between the speedup from having less average overhead and needing to launch less blocks through thread coarsening, and the slowdown from doing more work per thread. We dynamically adjusting the number of bytes we assign to a single thread based on the size of our world, by testing out different setups with world sizes, and the results are shown in following table.

World Size	**Width**	**Height**	**Block Size**	**Bytes per thread**
16384	128	128	8	2
65536	256	256	16	2
262144	512	512	32	4
1048576	1024	1024	128	4
4194304	2048	2048	128	16
16777216	4096	4096	128	16
67108864	8192	8192	128	16
268435456	16384	16384	128	16

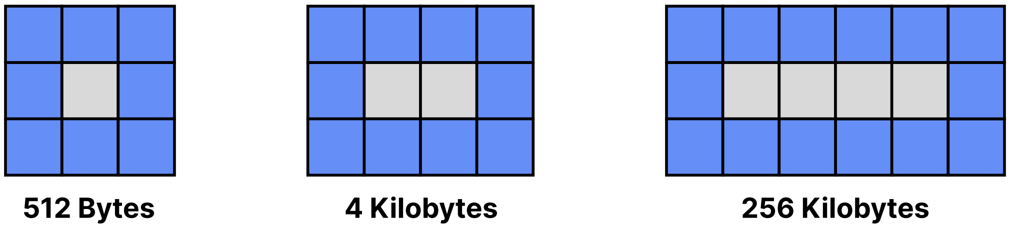
After implementing this version of GPU speedup, we realized that we can pack even more bits into a 32 bits word instead of just using a single byte. However, our initial attempt was unsuccessful so we moved on to the next optimization before coming back to this optimization again.

Lookup Table

To further improve our bit representation algorithm, we studied one of the most famous cpu algorithm for Conway's Game of Life speedup - Hashlife. Hashlife utilizes the fact that it can compute all possible outcome and store it in a hash table so at every cell update, the algorithm only has to perform a constant time lookup since it already knows the answer and does not need to perform any calculations.

Inspired by this, we also implemented our own version using a lookup table. The lookup table is stored in an byte array. The lookup table stores all possible states of a 6x3 area, which contains the needed cell states to update the 4 bits in the middle of the area, as seen in grey in the digram below. The lookup table is stored in a byte array, indexed by 18-bit integers that represent every possible 6x3 configuration. The value of each element represents the updated state of the middle 4 bits. This allows each thread to bypass the bitwise operations required in the previous implementation, and rely on the constant lookup times of the lookup table instead, greatly reducing the work done in each thread.

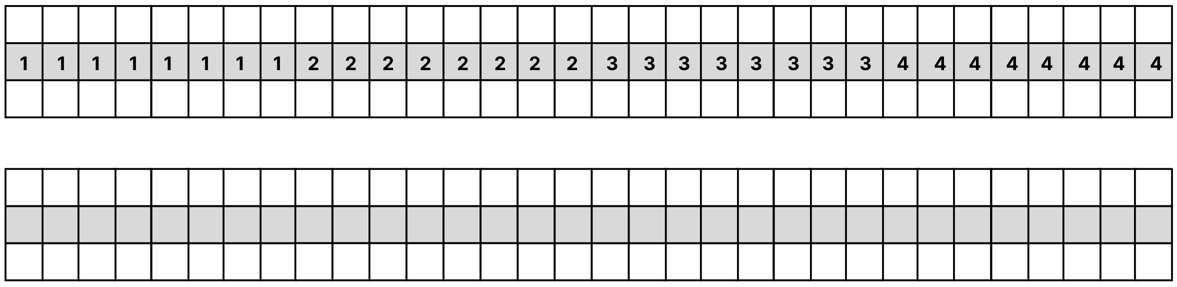
An obvious problem that emerges is the fact that the bigger chunk we look up at once, the bigger the lookup table becomes. In fact, it grows exponentially to the total number of cells. According to the following figures, we can see that to update 1 cell, we would need the state of 9 cells (the cell we are updating and its neighbors). To get all possibilities of all 9 cells we would need 2^9 bytes worth of memory since we are storing the values of the lookup table in a byte. To follow this formula of 2^(number of cells), we would need 4 kilobytes and 256 kilobytes for us to update 2 and 4 cells at once. Making this even bigger (8 cells) would imply that we would need 1 gigabytes worth of memory to store the lookup table. This would very costly to load and store into the GPU's limited memory, making it impractical for our solution



We ultimately decided with accessing four bits at once since it is not too memory intensive for the GPU and more efficient than smaller size lookups.

Improving Bit Representation

After successfully implementing the lookup table implementation. We decided to loop back to our bit representation to try to even further optimize the memory access by storing data into 32 bit integer array. This lets us to access the global memory even less often, which further improves the performance. From the figure below, you can see that instead of accessing the memory four times, we can evaluate and access 32 cell states in one memory read.



Comparing to storing data in a single byte, storing data in a 32 bit integer array requires us to do more bit manipulation and requires us to read in more cell states on the edges. However, the speedup it provides with less global memory reads and allowing each thread to doing more work outweighs the effects of the overhead. At bigger world sizes, this algorithm performs best reading 32 bytes per thread.

Evaluation Methodology

We evaluate the performance of our implementations by wrapping a cutil timer around the parts we try to evaluate. Specifically we use cutCreateTimer, cutStartTimer, and cutStopTimer.

For our CPU benchmark evaluation, we wrapped the timer around the implementation and measure its performance over 100 iterations/generations in the simulation for Conway's Game of Life.

On the other hand, we measure our GPU performance similar to how we ran them in the homework. We would launch the kernel once to start it up in order to cut out the startup overhead and then wrap the timer around the kernel call (also running 100 iterations), timing everything from when the kernel gets called to when the kernel returns. We average the time over ten trials and run every test back to back to make sure there is as little variation as possible since the Wilkinson lab machines do not perform very consistently through out the day/week.

For our implementations with lookup tables (lookup table GPU and 32-bit GPU), we also pre-computed the lookup table and passed it into the kernel. This means the computation time of the lookup table is not a part of recorded time.

For the world sizes, we start from 256 by 256 and scale all the way to 16384 by 16384 worlds to time. We also compare the end result of the naive CPU vs the GPU implementations we test it against to make sure the speedup also guarantees correctness.

Experimental Result

As discussed, we compared the performance of the five different implementation we have tried out. Starting from the brute force CPU implementation to GPU, then speeding it up with bit representation, lookup table, and finally the upgraded bit representation. We run it on three different world sizes with each of them iterating 100 generations. All time is measured in millisecond.

World Size	CPU	Naive GPU	Bit Rep GPU	Lookup Table GPU	32-Bit GPU
16777216	14979 ms	764 ms	71 ms	16 ms	11 ms
67108864	59759 ms	2875 ms	250 ms	64 ms	43 ms
268435456	256852 ms	11381 ms	862 ms	233 ms	178 ms

Other Attempts

Aside from the five implementations we have discussed, we also tried to enhance the performance of our program with several other optimizations. Before we looked into implementing the lookup table version of Conway's Game of Life, we looked into running Hashlife on GPU for speedup. However, we quickly realized that hashlife has been resistant to parallelization due to the nature of the algorithm. This made us pivot into the lookup table implementation, inspired by the normal life algorithm.

We also tried to improve our memory access by putting parts of the grid into the shared memory. We initially tried to put most parts of the grid in and then slowly reduced it to intersection points between bytes. However, in practice it was ~15% slower than previous implementations, so we further reduced the cell states we store in the shared memory by only including cells on the east and west sides of the byte, since those are the only cells that do not require if-else statements during evaluation. However, this was still ~10% slower compared to the pure global memory implementation. We suspect that this is due to that each thread is required to make two global memory reads to populate the shared memory, and these initial reads are also all strided, and each cell in shared memory is ultimately only accessed 6 times.

Lastly, we also tested the limit of the lookup table by attempting to check 8 cells at once. However, the size of the lookup table grew exponentially and became apparent that we cannot store more into the lookup table due to memory constraint.

Related Works

There are many related works that involves speeding up Conway's Game of Life. Earlier works are more driven through algorithms ran on CPU utilizing a faster compute time and memoization. We now see a lot more work done in the space of parallel algorithms with Conway's Game of Life. Specifically, these parallel algorithms focus on efficiently storing data, accessing data, simulating/updating data, and manipulating data. We have touched on three of the four aspects mentioned, but there are research that goes beyond the scope of our implementation.

For even better 32 bit optimization, some have done work in Bitwise Parallel Bulk Computation that further improves computational speed. Some have also taken inspiration from part of Hashlife where it simulates beyond the scope of one iteration to reduce overhead caused by data transferring. Lastly, there are also implementations with warp shuffling to better transfer data of the current cell state.

Conclusion

We are not entirely satisfied with our implementation since some of our ideas did not work in the end. However, it was a rewarding learning experience as we gained better understanding of writing custom CUDA kernel code. Moreover, through our experimentation and testing, we found differences and tradeoffs in the CUDA memory hierarchy, which were surprising and different from our assumptions at times. Lastly, we also gained more insights in how we can approach problem solving in the space of speeding up algorithms with parallelization.

In retrospect, we could have spent more time thinking about the details of implementing some of our ideas before we executed since we did go back and forth on both the 32-bit and the shared memory access implementation. If we have thought about the details more, we believe we could have saved more time implementing and testing these two ideas.

All in all, this project was great learning experience for us to better understand the inner workings of CUDA and learn more about both sequential and parallel algorithms that speedup Conway's Game of Life.

References

Fujita, Toru, Daigo Nishikori, Koji Nakano, and Yasuaki Ito, “Efficient GPU Implementations for the Conway’s Game of Life”, 2015 Third International Symposium on Computing and Networking.

Rokicki, Tomas G., “Life Algorithms”, gathering4gardner, June 2018.

Wilhelmsen, B. (2021, June 5). My experience learning cuda to accelerate Conway's Game of Life. Medium. Retrieved March 12, 2023, from https://brendanrayw.medium.com/my-experience-learning-cuda-to-accelerate-conways-game-of-life-5d52eabc2dfb.

Niemiec, Mark B., “Life Algorithms.” Byte Magazine, January 1979.

M. Bailey and S. Cunningham, A hands-on environment for teaching GPU program- ming, Proc. of SIGCSE Technical Symposium on Computer Science Education (ACM, 2007), pp. 254–258.

Rokicki, Tomas G., “An Algorithm for Compressing Space and Time”, Dr. Dobbs, April 2006.

Rhythm Goyal, Charan Lalchand Soneji, Bhanu Venkata Kiran Velpula, and Sairabanu J, “Performance Enhancement of Conway’s Game of Life Using Parallel and Distributed Computing Platform”, School of Computer Science and Engineering Vellore Institute of Technology, Vellore, 2020.