Conway's

Game Of Life - 3D

Team Members:

1. Hoyath Ali (Hsing093)

Demo Video URL: GameOfLife_Hoyath_3d_CUDA

Table Of Contents

Introduction

Conway's Game of Life 3D in GPU Python Numba project introduces a three-dimensional adaptation of the classic Game of Life, leveraging GPU acceleration for enhanced performance. Using the Numba library for Just-In-Time compilation, the project optimizes computational efficiency, taking advantage of parallel processing on GPUs. The goal is to provide a faster and visually engaging implementation of cellular automata in a three-dimensional space.

Basic Rules of Game

The rules of the Game of Life 3D are an extension of the classic Game of Life rules into three dimensions. Here are the key points:

1. Birth and Survival: A cell in the 3D grid is "born" if it has exactly a certain number of live neighbors, similar to the 2D version. Additionally, a live cell survives to the next generation if it has a specific number of live neighbors.

2. Neighborhood Definition: In the 3D version, each cell has 26 neighbors, accounting for all adjacent cells in the three dimensions (including diagonals).

3. Overpopulation and Underpopulation: If a live cell has too few (underpopulation) or too many (overpopulation) live neighbors, it dies due to isolation or overcrowding, respectively.

4. **Birth Conditions**: A cell becomes alive in the next generation if it is currently dead and has a specific number of live neighbors, simulating reproduction.

5. Static and Oscillating Patterns: The Game of Life 3D, like its 2D counterpart, can exhibit static patterns where cells remain unchanged, or oscillating patterns where cells alternate between different states over generations, providing dynamic and evolving structures in the three-dimensional space.

Technical Details

Parallel Algorithm Design/Implementation:

The parallel algorithm design is realized through the `**update_conways_game_of_life**` kernel function, which is decorated with `@cuda.jit`. This decorator signals that the function is designed to be executed in parallel on the GPU.

```
def kernel_update_conways_game_of_life(current_state, next_state):

    i, j, k = cuda.grid(3)

    dim_x, dim_y, dim_z = current_state.shape

    if 0 <= i < dim_x and 0 <= j < dim_y and 0 <= k < dim_z:

        alive_neighbors = 0

        for di in range(-1, 2):

            for dj in range(-1, 2):

                for dk in range(-1, 2):

                    ni, nj, nk = i + di, j + dj, k + dk

                    if 0 <= ni < dim_x and 0 <= nj < dim_y and 0 <= nk < dim_z:

                        alive_neighbors += current_state[ni, nj, nk]

        alive_neighbors -= current_state[i, j, k]

        if current_state[i, j, k] == 1:

            next_state[i, j, k] = 1 if 1 <= alive_neighbors <= 4 else 0

        else:

            next_state[i, j, k] = 1 if alive_neighbors == 3 else 0
```

- **Walking through the kernel Code:**
    - Imagine we're looking at each cell in a 3D grid one at a time.
- **Checking the Neighborhood:**
    - For each cell, we're checking its surroundings, like its neighbors in a 3x3x3 cube around it.
- **Staying Within the Grid:**
    - We're careful not to go outside the boundaries of the grid. If our cell is near the edge, we don't want to peek beyond.
- **Counting Neighbors:**
    - For every cell, we count how many of its neighbors are alive (have a value of 1).
- **Deciding Fate:**
    - **If a cell is currently alive:** It stays alive in the next generation if it has between 1 and 4 living neighbors. Otherwise, it dies.
    - **If a cell is currently dead:** It comes to life in the next generation only if it has exactly 3 living neighbors. Otherwise, it stays dead.
- **Efficient Processing:**
    - We're doing all this efficiently, using the power of the GPU (graphics processing unit) with CUDA. It helps us process a lot of cells at once.

Thread/Thread Block Partitioning:

- **Block Size and Grid Size:** The CUDA kernel kernel_update_conways_game_of_life is designed to update a 3D grid of cells in Conway's Game of Life. A block size of 16x16x16 threads is chosen for parallel processing within a block.
- **Dynamic Grid Size Calculation:**
    - The grid size is dynamically calculated based on the dimensions of the 3D grid (DIM_X, DIM_Y, DIM_Z).
    - The number of blocks needed in each dimension is calculated to cover the entire grid using the chosen block size.
    - This ensures that the CUDA kernel efficiently processes the entire 3D grid, adapting to different grid dimensions.

- **CUDA Streams:**
    - Two CUDA streams, stream_copy and stream_kernel, are created to manage asynchronous operations.
    - stream_copy is used for memory copy operations between the CPU and GPU.
    - stream_kernel is used for launching the CUDA kernel to update the generations in parallel.

```
stream_copy = cuda.stream()

stream_kernel = cuda.stream()

start_time = time.time()

current_generation_device=cuda.to_device(current_generation,
stream=stream_copy)

next_generation_device=cuda.to_device(next_generation, stream=stream_copy)

update_matrix_gpu(current_generation_device, next_generation_device)

next_generation_device.copy_to_host(next_generation, stream=stream_copy)

cuda.synchronize()

end_time = time.time()
```

Parallelization Stage:

The primary parallelization stage occurs within the `update_conways_game_of_life` kernel. Each thread independently calculates the next state of the cell it is responsible for, based on the rules of Conway's Game of Life. The nested loops iterate over the neighboring cells, counting the number of alive neighbors. This parallel approach allows for the simultaneous computation of cell states, improving the overall performance compared to a sequential implementation.

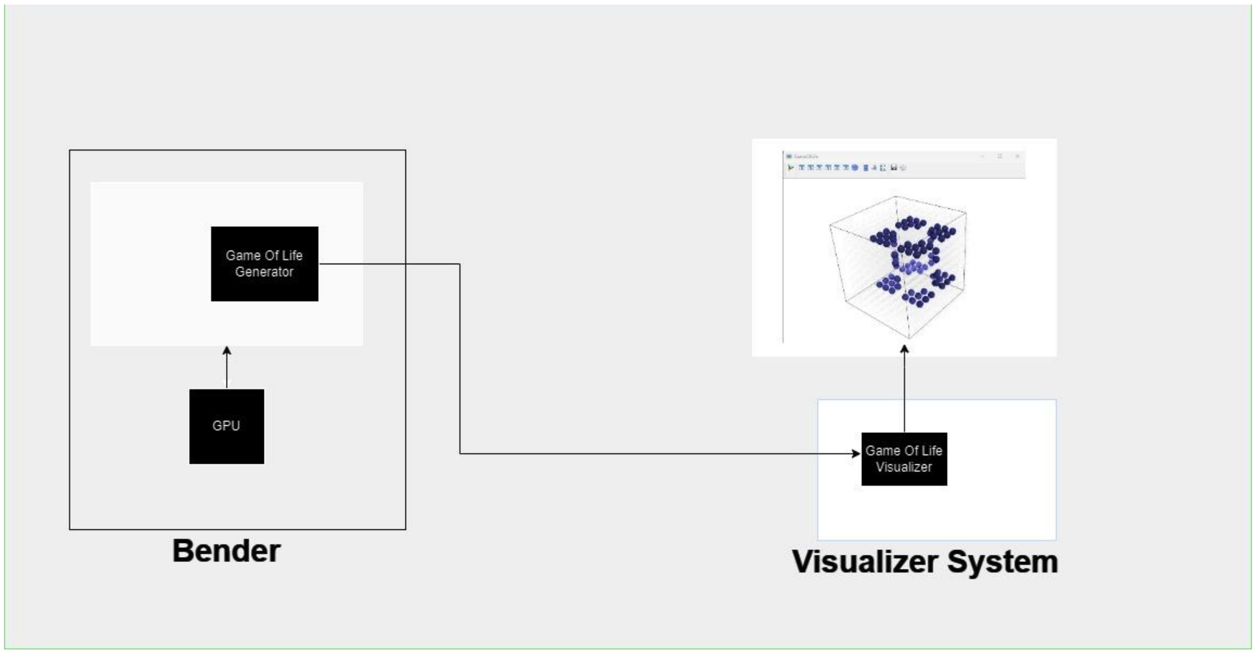
Libraries Used:

- Numba
- Numpy
- Flask
- Mayavi
- Other python libraries

Implementation details

The primary challenge in this implementation was visualizing the results, given limited GPU system access via the command line. To address this, I devised a solution to effectively visualize Conway's Game of Life, as traditional GUIs like Tkinter couldn't be launched on the remote system. The approach involved setting up two Flask applications—one on the bender system and another on my personal computer.

The bender system's Flask app acts as the server, handling the computation of cellular generations in the game and sending the output via Flask. On my personal system, the second Flask server receives these messages, which essentially consist of 3-dimensional arrays. It parses the data and facilitates an effective visualization of the results.



Data Transmission:

The initial Python application, running on a remote system, generates and computes new generations of the cellular automaton using the pulsar pattern and the Game of Life rules.

The resulting data, representing the evolving state of the cellular automaton, is sent as messages to another Python application.

Reception and Visualization:

The second Python application, running on your personal computer, acts as a receiver for these messages.

Upon receiving the messages, it interprets the data, which is essentially a 3-dimensional array reflecting the state of the Game of Life.

Mayavi Visualization:

Using the Mayavi library, the Python application plots and visualizes the received 3D array data in a graphical format.

Mayavi provides a powerful framework for 3D scientific data visualization, allowing the representation of complex structures and patterns within the cellular automaton.

Dynamic Visualization Updates:

As new generations are computed and transmitted by the initial Python application, the second application dynamically updates the Mayavi visualization to reflect the evolving patterns in Conway's Game of Life.

Running the code

1. **Clone the Repository on to a system with GPU**

   https://github.com/hoyathali/GameOfLife3D

2. **Pip install all the required libraries (requirements.txt)**
   a. **pip install numba mayavi numpy flask flask-socketio requests**
3. **Run python main.py which will launch a mayavi visualization window**: This will launch a Mayavi window, providing an interactive visualization of the generations.
4. **Run python servermain.py on a system with GPU as this has the kernel code** (If we running these two files in two different systems then please update servermain.py files line 77 with the public URL to your system where main.py is running

   ```
   url = 'http://localhost:5000/receive'
   ```

5. **That's it we have our 3D Game of Life Ready.**

As the bender system computes new generations, observe the interactive Mayavi window on your personal computer. It will dynamically update to reflect the evolving patterns of Conway's Game of Life. We can explore various different initial patterns, by default this starts with a pulser pattern.

Evaluation

Comparison Report: GPU vs. CPU Execution

Objective:

To compare the execution times of a specific task with and without GPU acceleration.

We have performed many tests with different sizes of worlds, starting from (10*10*10) to (100 *100*100)

Run 1: Without GPU (30 * 30* 30 - Average Score for a generation)

Execution Time: 0.000773030098 seconds

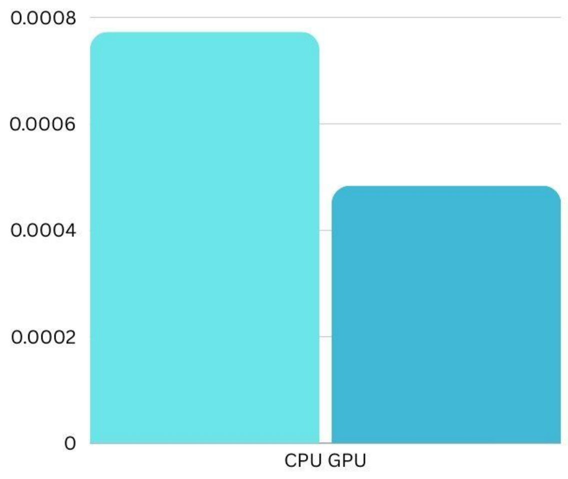
Run 2: With GPU (30 *30*30 - Average score for a generation)

Execution Time: 0.0004838885683 seconds

Comparison:

The execution time with GPU acceleration (Run 2) is approximately **37.5%** smaller than without GPU acceleration (Run 1).

This reduction in execution time suggests that utilizing GPU resources has led to a performance improvement for the given task.



Status

Feature Completeness:

The project has reached a feature-complete state with successful implementation of the core aspects, such as GPU-accelerated Conway's Game of Life in 3D with Numba and Python. The visualization using Mayavi on a personal computer enhances the overall experience.

Functionality Status:

The primary functionalities, including GPU acceleration using streams for overlapping computations with data transfer, data transmission between systems, and interactive Mayavi visualization, are operational.

However, there might be some aspects still under development or refinement, such as specific visual representations or optimizations.

Limitations:

The visualization process encounters challenges with larger matrices, impacting the efficiency of the display in Mayavi. However, it's important to note that on the computation side, leveraging GPU acceleration with Numba introduces no inherent limitations. Future refinements may focus on optimizing the visualization aspect for larger datasets..

Technical Challenges:

One major technical challenge involved overcoming the limitations of GPU access via the command line and devising a solution for effective visualization. The use of Flask applications and Mayavi on a personal computer addressed this challenge.

Task Breakdown

Task	Breakdown
GPU: Computation of generations	Hoyath
Flask communication between two systems	Hoyath
Visualization of the Generations	Hoyath

Thank you