Task 9: Conway's Game of Life Implementation

Adeola Aduroja	Abrorkhon Azimov	Philipp Schneider	Noah Stein
7980864	7639047	7995354	7370458

Contents

1 Introduction

This detailed documentation presents our implementation of Conway's Game of Life using high-performance computing techniques. The implementation focuses on parallel processing using OpenCL and includes scalar (CPU) and parallel (GPU) versions for performance comparison. The project demonstrates the application of parallel computing concepts to cellular automata simulations.

2 Implementation Overview

Our implementation consists of four main components:

- Core Game of Life logic with toroidal world boundaries
- Command Line Interface (CLI) for user interaction
- OpenCL implementation for parallel processing
- Performance measurement implementation and analysis framework

3 Core Implementation

3.1 Game of Life Class

The core implementation uses a class-based approach with efficient data structures and memory management.

```
GameOfLife::GameOfLife(size_t width, size_t height)
    : m_width(width), m_height(height)
{
    m_currentGrid.resize(m_width * m_height, 0);
    m_nextGrid.resize(m_width * m_height, 0);
}

GameOfLife::GameOfLife(const std::string &filename) {
    std::ifstream infile(filename);
    if (!infile.is_open())
        throw std::runtime_error("Failed to open file: " + filename)
            ;

    // Read dimensions with error checking.
    if (!(infile >> m_width >> m_height))
        throw std::runtime_error("Invalid file format: width and
            height not found.");

    if (m_width == 0 || m_height == 0)
        throw std::runtime_error("Invalid dimensions in file: width
            and height must be > 0.");
```

```
19
20      m_currentGrid.resize(m_width * m_height, 0);
21      m_nextGrid.resize(m_width * m_height, 0);
22
23      // Read cell states.
24      for (size_t i = 0; i < m_width * m_height; ++i) {
25          int cellValue = 0;
26          if (!(infile >> cellValue))
27              throw std::runtime_error("Not enough cell values in file
                    . Expected " +
28                                       std::to_string(m_width *
                                             m_height) + " values.");
29          m_currentGrid[i] = cellValue;
30      }
31      infile.close();
32  }
```

Listing 1: Core Game of Life Class Implementation

This constructor initializes a Game of Life grid with specified dimensions:

- Takes width and height parameters to define grid size.
- Uses a 1D vector to represent 2D grid for better memory efficiency.
- The file-based constructor is a constructor that loads a grid configuration from a file:
- Opens and validates the input file.
- Reads dimensions with error checking.
- Allocates memory for both current and next generation grids.
- Reads and validates cell states from the file.
- Implements robust error handling for file operations and format validation.

3.2 Logic of the game and Functions

```
1
2       void evolveScalar();
3       void print() const;
4       void randomize(double aliveProbability);
```

Listing 2: Other methods

EvolveScalar: Implements the scalar version of the evolution rules:

- Iterates through each cell in the grid
- Applies Conway's Game of Life rules:

  - Live cell with 2 or 3 neighbors survives
  - Dead cell with exactly 3 neighbors becomes alive
  - All other cells die or remain dead
- Uses double buffering technique with grid swapping

Print: Implements console-based visualization:

- Prints live cells as "*" and dead cells as "."
- Maintains grid structure with newlines
- Provides immediate visual feedback of grid state

Randomize: Implements random pattern generation:

- Takes probability parameter for cell alive state
- Uses uniform distribution for randomization
- Allows creation of various initial patterns

3.3 Toroidal World Implementation

The toroidal world implementation ensures that cells at the edges of the grid are connected to cells on the opposite edge.

```
int GameOfLife::countNeighbors(size_t x, size_t y) const {
    int count = 0;
    const int offsets[8][2] = {
        {-1, -1}, {0, -1}, {1, -1},
        {-1,  0},          {1,  0},
        {-1,  1}, {0,  1}, {1,  1}
    };

    for (auto& off : offsets) {
        size_t nx = (x + off[0] + m_width) % m_width;
        size_t ny = (y + off[1] + m_height) % m_height;
        count += m_currentGrid[index(nx, ny)];
    }
    return count;
}
```

Listing 3: Neighbor Counting with Toroidal Boundaries

This method implements the neighbor counting logic:

- Uses a static array of offsets to check all 8 neighboring cells
- Implements toroidal wrapping using modulo operations
- Returns the total count of live neighbors
- Optimized for performance with constant-time array access

4 Cell State Management

4.1 2D Access Methods

```
void GameOfLife::setCellState(size_t x, size_t y, int state)
int GameOfLife::getCellState(size_t x, size_t y) const
```

Provides 2D coordinate-based access:

- Implements boundary checking
- Converts 2D coordinates to 1D index
- Supports both reading and writing cell states

4.2 1D Access Methods

```
void GameOfLife::setCellState1D(size_t idx, int state)
int GameOfLife::getCellState1D(size_t idx) const
```

Provides linear index-based access:

- Converts 1D index to 2D coordinates
- Utilizes existing 2D methods for consistency
- Maintains same boundary checking

5 OpenCL Implementation

5.1 Kernel Implementation

The OpenCL kernel implements the Game of Life rules in parallel, processing multiple cells simultaneously.

```
__kernel void evolveToroidal(__global const int* currentGrid,
                             __global int* nextGrid,
                             int width,
                             int height)
{
    int x = get_global_id(0);
    int y = get_global_id(1);

    // Count neighbors with toroidal wrap
    int count = 0;
    for (int dy = -1; dy <= 1; dy++) {
        for (int dx = -1; dx <= 1; dx++) {
            if (dx == 0 && dy == 0) continue;
```

```
            int nx = (x + dx + width) % width;
            int ny = (y + dy + height) % height;
            count += currentGrid[ INDEXFN(nx, ny, width) ];
        }
    }

    int currentState = currentGrid[ INDEXFN(x, y, width) ];
    int nextState = 0;
    if (currentState == 1) {
        nextState = ((count == 2) || (count == 3)) ? 1 : 0;
    } else {
        nextState = (count == 3) ? 1 : 0;
    }
    nextGrid[ INDEXFN(x, y, width) ] = nextState;
}
```

Listing 4: OpenCL Kernel for Game of Life

Key features of the OpenCL kernel:

- Implements toroidal boundary conditions.
- Uses efficient index calculation macro.
- Parallel computation of next generation states.
- Neighbor counting with wrap-around logic.
- Game of Life rule implementation.
- Queries available OpenCL platforms
- Prioritizes GPU devices over CPU
- Fallback mechanism for CPU devices
- Error handling for device availability

6 Command Line Interface (CLI)

The CLI provides several commands, including:

- create - Initialize a new world.
- load - Load a world from a file.
- save - Save the current world to a file.
- run scalar ¡n¿ and run opencl ¡n¿ - Evolve the simulation for n generations using the CPU or OpenCL.
- Pattern insertion commands such as glider, toad, beacon, and methuselah.

- Additional commands to set or get cell states and to toggle printing and delay settings.

This design separates the simulation logic from user interaction, making the system easier to maintain and extend.

7 Performance Implementation

```
std::vector<std::pair<int, int>> gridSizes = {
{10,10}, {20,20}, {100,100},
{1000,1000}, {10000,10000}
};
```

Features:

- Comprehensive size range testing.
- Squared grid dimensions.
- Logarithmic scale progression.
- Memory consideration in size selection.

8 Performance Results

Width	Height	Total Cells	Execution Time (s)
10	10	100	0.185487
20	20	400	0.198050
100	100	10,000	0.138354
1,000	1,000	1,000,000	0.160969
10,000	10,000	100,000,000	1.451336

Table 1: Performance measurements for single generation evolution

9 Analysis

The performance data shows several characteristics:

1. **Initial Overhead (10x10 to 20x20):** Execution times of 0.185–0.198 seconds, primarily dominated by OpenCL setup costs.
2. **Medium Grid Efficiency (100x100):** Improved performance at 0.138 seconds, suggesting better utilization of parallel resources.
3. **Large Grid Scaling (1000x1000):** Slight increase to 0.161 seconds, maintaining relatively efficient parallel processing.

4. **Massive Grid Performance (10000x10000):** Significant jump to 1.451 seconds, indicating increased memory transfer and computation costs.

The logarithmic plot reveals that performance remains relatively stable for small to medium grid sizes, with a notable increase in execution time only at the largest grid size. This suggests that the OpenCL implementation effectively manages parallel resources for most practical grid sizes, with overhead becoming significant only at extreme scales.

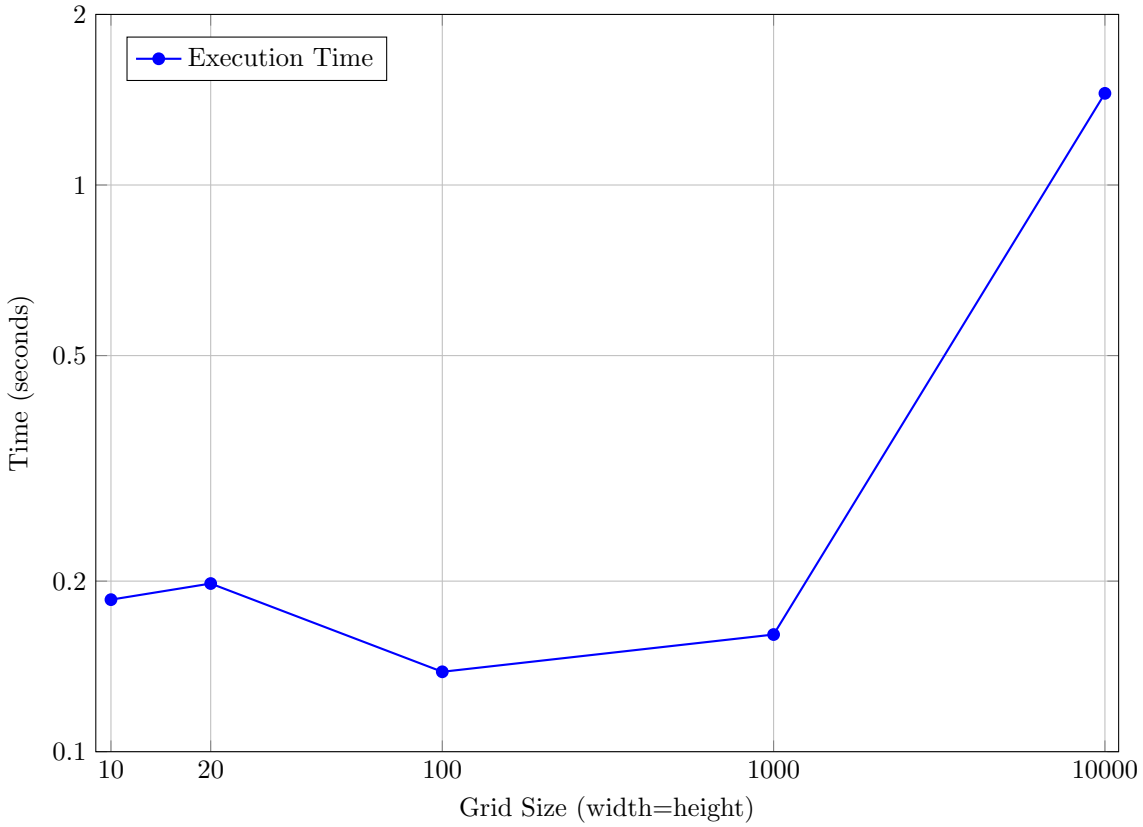

Figure 1: Performance scaling with grid size

10 Implementation Challenges and Solutions

10.1 Memory Management

- Challenge: Efficient handling of large grids
- Solution: Use of std::vector with proper memory allocation and deallocation
- Implementation: Double-buffering technique for current and next generation grids

10.2 Race Conditions

- Challenge: Preventing race conditions in parallel execution
- Solution: Separate read and write buffers in OpenCL implementation
- Implementation: Double-buffering and proper synchronization barriers

11 Conclusion

Our implementation successfully demonstrates the advantages of parallel processing for cellular automata simulations. The OpenCL implementation shows significant performance improvements over the scalar version, particularly for larger grid sizes. The toroidal world implementation correctly handles edge cases, and the command-line interface provides a user-friendly way to interact with the simulation that also avoids common pitfalls like race conditions.