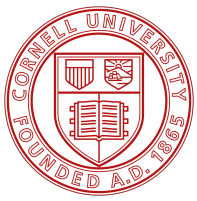
Cornell University



ECE 5720 - Introduction To Parallel Computing

Final Project Report - Game of Life on CUDA

Yu Zhang (yz2729) Bin Xu (bx83) Hongyi Wu (hw727)

May 16, 2022

Introduction

Game of life is a well-known cellular automaton invented by Cambridge mathematician John Conway. Cited from Wikipedia, it is a zero-players game, which means that its evolution is based on its initial state, requiring no further input. Players can only decide the initialization information of the game and then observe the game evolves in generations.

The scene of the game is set on an infinite two-dimensional board. Each cell of the board is in one of two possible states, live or dead which depends on the number of living cells around it. Every cell can only interact with its 8 connected neighbours. The rules of Game of life can be summarized by following items:

- Any live cell with fewer than two live neighbours dies, as if by under population
- Any live cell with two or three live neighbours lives on to the next generation
- Any live cell with more than three live neighbours dies, as if by overpopulation
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction

The rules can be observed in Figure 1.

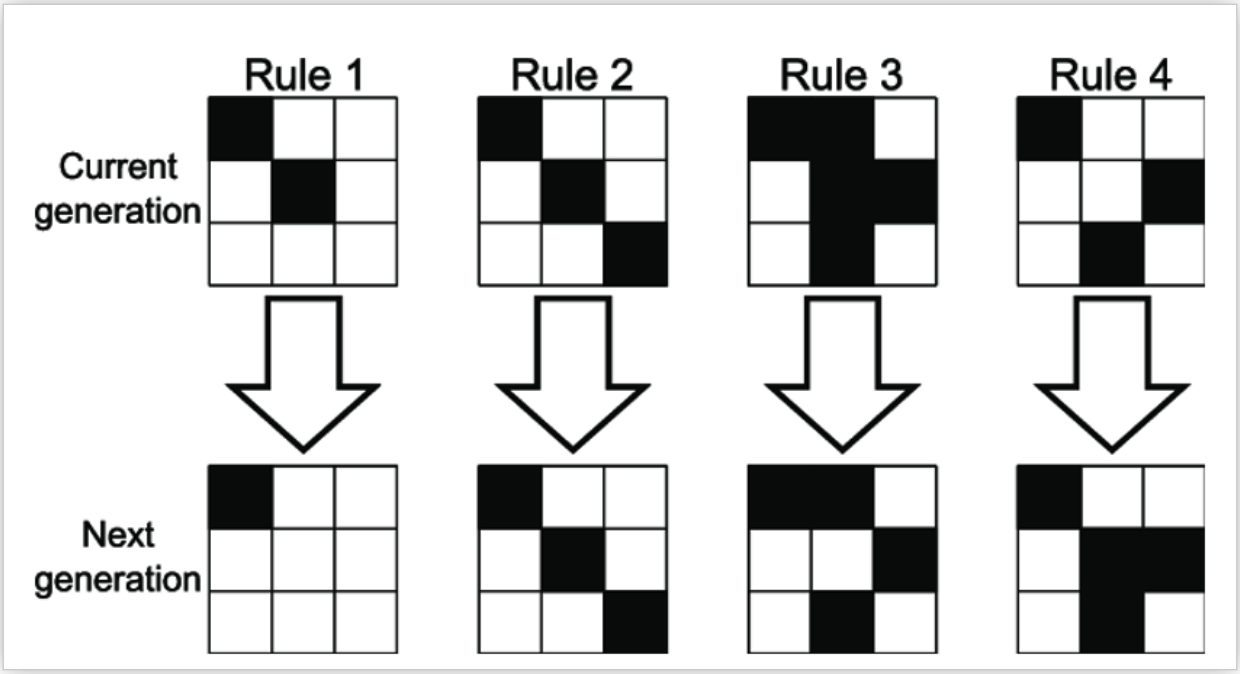


Figure 1: Rules of GoL

This project aims to compare performance of CPU and GPU in evaluation of Game of Life. The performance is tested by three different implementations: CPU sequential version, CPU parallel version and GPU parallel version. In the following sections, we will first introduce the motivation of our project, present our algorithm design and implementation, then show the computational results, and finally conclude.

Motivation

Our motivation can be summarized in the following three points:

- The implementation of Game of life should be a very typical example of the value of parallelization. It is set on an infinite grid, have the feature to be split into different parts, computed separately
- With such a computational process, we can improve the computational efficiency of the game and reduce the computation time. In addition, implementing Game of Life with multiple threads is more efficient because parallel computing can directly takes the advantage of the GPU
- The Game of Life can be easily visualized on their iterations so we can have more intuitive demos about the benefit of computation

Design & Implementation

In this section, we will describe in detail the design and implementation of our program, which is divided into four parts: CPU sequential design, CPU parallel design, GPU parallel design, and visualization design.

CPU Sequential

The design of the CPU sequential is very straightforward. Its algorithmic flow is shown in Figure 2. First we initialize the data at the beginning of the matrix, and then iterate through the matrix to calculate the next state of each cell in each iteration, and then update the data and go to the next iteration until the end of the iteration.

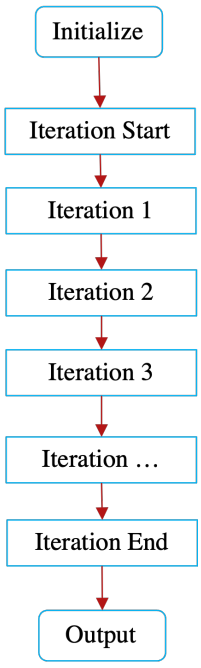


Figure 2: CPU Sequential

The implementation of CPU sequential is also not complicated. Inside each iteration, we compute the matrix of the next state based on the matrix of the previous state each time. And the next state is saved in the previous state for the next iteration. When updating the matrix, we iterate through the values of each cell in the previous state and the values around it to calculate its surviving state in the next state. The code to update the matrix is shown in code listing 1.

```
/************* World calculation ***************/
void updateMatrix(int **prev, int **succ, int m, int n)
{
/************* Parameters Define ***************/
  int alive, p, q, i, j;

/** Go through 8 neighbours, find num of lives **/
  for (p = 1; p < m - 1; p++){
    for (q = 1; q < n - 1; q++){
        alive = 0;
        for (i = -1; i <= 1; i++){
            for (j = -1; j <= 1; j++){
                alive += prev[p + i][q + j];
            }
        }
        //Deduct cell itself
        alive -= prev[p][q];

        //Rule 1: underpopulation
        if (alive < 2 && prev[p][q] == 1){
            succ[p][q] = 0;
        }
        //Rule 2: overpopulation
        else if (alive > 3 && prev[p][q] == 1){
            succ[p][q] = 0;
        }
        //Rule 3: reproduction
        else if (alive == 3 && prev[p][q] == 0){
            succ[p][q] = 1;
        }
        //Rule 4: No change
        else{
            succ[p][q] = prev[p][q];
        }
    }
  }
}
```

Listing 1: Update World - CPU Sequential

CPU Parallel

The flow of CPU parallel computing is actually very similar to the flow of CPU sequential, the biggest difference lies in the iterative part. As shown in figure 2, we use `OpenMP` to implement CPU parallel computing on the iterative part. Using `OpenMP` to open multiple threads to update the matrix can greatly increase the speed of computation.

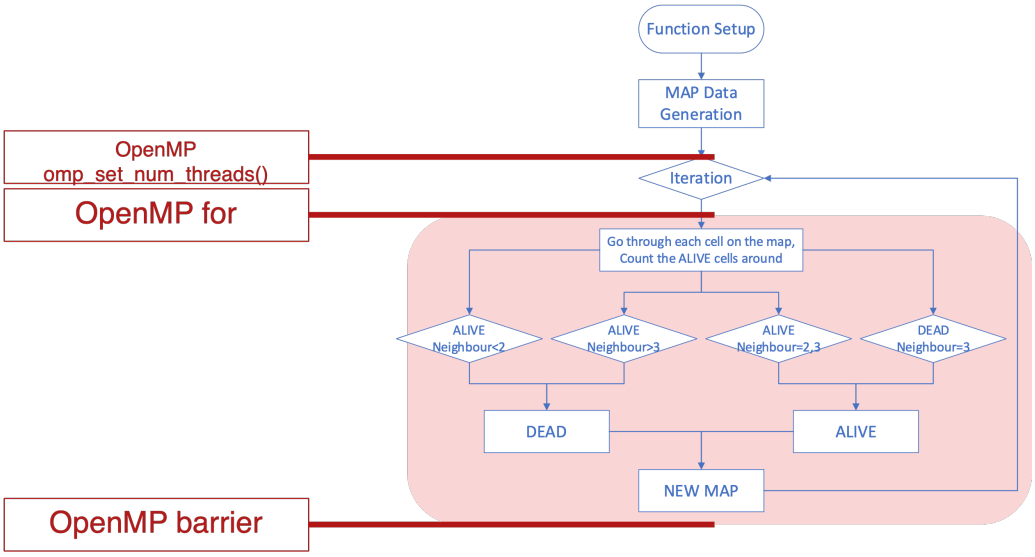


Figure 3: CPU Parallel

When updating the matrix, we first set the number of threads with `omp parallel num_threads(8)`, and then use `pragma omp for schedule(static)` to declare the code for parallel operations. Finally, after updating the matrix, we use `pragma omp barrier` to synchronize the progress of all threads. The code for updating matrix in CPU parallel mode is shown in code listing 2.

```
/************* World calculation ***************/
void updateMatrix(int **prev, int **succ, int m, int n)
{
/************* Start OpenMP ***************/
    #pragma omp parallel num_threads(8)  // Set treads to 8
    #pragma omp for schedule(static)  // OpenMP parallel for

/** Go through 8 neighbours, find num of lives **/
    for (int p = 1; p < m - 1; p++){
        for (int q = 1; q < n - 1; q++){
            int alive = 0;
            for (int i = -1; i <= 1; i++){
                for (int j = -1; j <= 1; j++){
                    alive += prev[p + i][q + j];
                }
            }
        //Deduct cell itself
        alive -= prev[p][q];
```

```
        //Rule 1: underpopulation
        if (alive < 2 && prev[p][q] == 1){
            succ[p][q] = 0;
        }
        //Rule 2: overpopulation
        else if (alive > 3 && prev[p][q] == 1){
            succ[p][q] = 0;
        }
        //Rule 3: reproduction
        else if (alive == 3 && prev[p][q] == 0){
            succ[p][q] = 1;
        }
        //Rule 4: No change
        else{
            succ[p][q] = prev[p][q];
        }
    }
  }
  #pragma omp barrier  // Synchronize
}
```

Listing 2: Update World - CPU Parallel

GPU Parallel

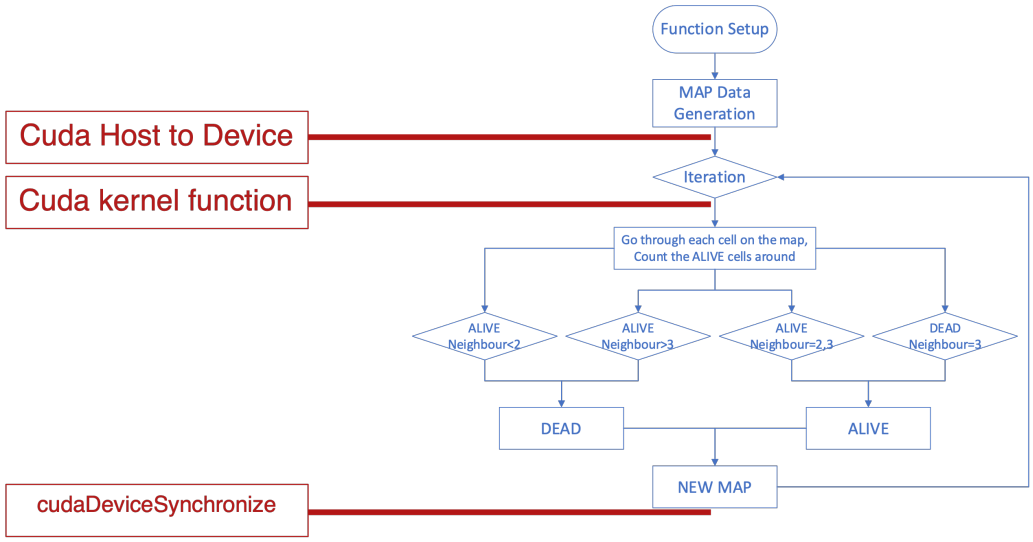


Figure 4: GPU Parallel

The algorithm of GPU parallel computing can be found in figure 4. The main structure is the same as the OpenMP. Before each iteration, the program generates a map with the desired size. Then copy the map from host to device for GPU calculation. Before the final solution, the data will not be transmitted between CPU and GPU. After each iteration, the CUDA kernel function will be executed. At the end of each iteration, the program synchronizes the data.

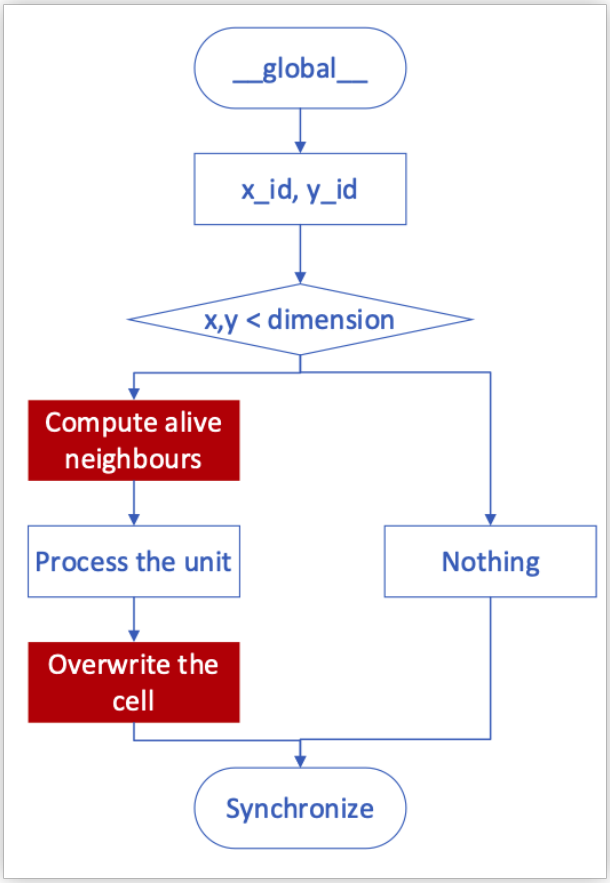


Figure 5: Cuda Kernel Function - Global Memory

Two versions of CUDA kernel functions are implemented with global memory and shared memory. Global memory is faster than shared memory, according to the theory. For the global version, shown in figure 5, after entering the kernel function, the system allocates each map cell a thread with IDx and IDy. A grid size 256*256 and block size 32*32 is used in this program because of the iteration of world size. For those threads that exceed the world size, we pass the execution. The others compute the alive neighbors with global memory, process the unit, and update the information.

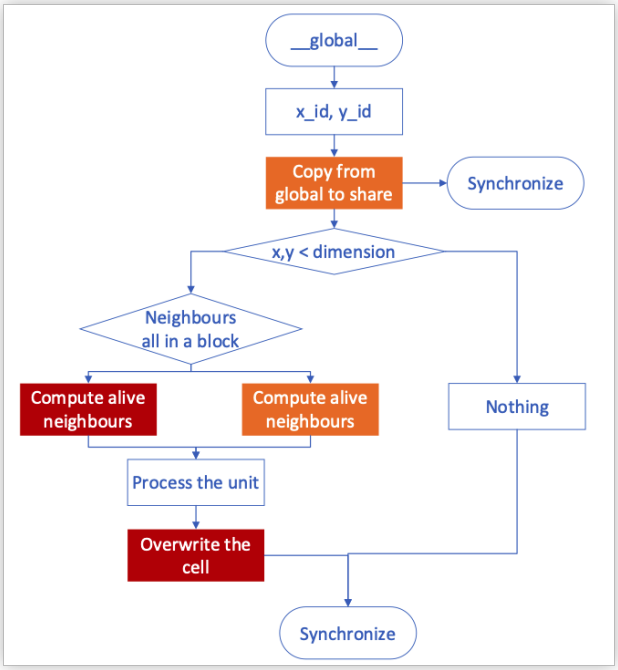


Figure 6: Cuda Kernel Function - Shared Memory

The shared memory version is similar and shown in figure 6. The difference could be another judgment of the cell's neighbors within the block's dimension. If the neighbors are all inside one block, we can use the shared memory directly. In contrast, we should copy the neighbors from global memory. Although the initiative is to accelerate the computation, there are some missing points in this design. The device needs to synchronize at the end of each iteration, which means that the fast memory access should wait for the slow global access. If the world is defined as the size of a block 32*32, the size could be too small, and the primary consumption could be copying data or other executions.

Visualization

As shown in the figure 7, Our visualization solution is that we store the intermediate results of each frame into an txt file. Then, we implemented a simple JavaScript application that can read the data by frame in the txt file according to the time interval we set. The time interval is calculated based on the total time we run and the number of iterations. In the actual demonstration, we artificially scaled up the runtime by a factor of 10,000 in order to amplify their differences.

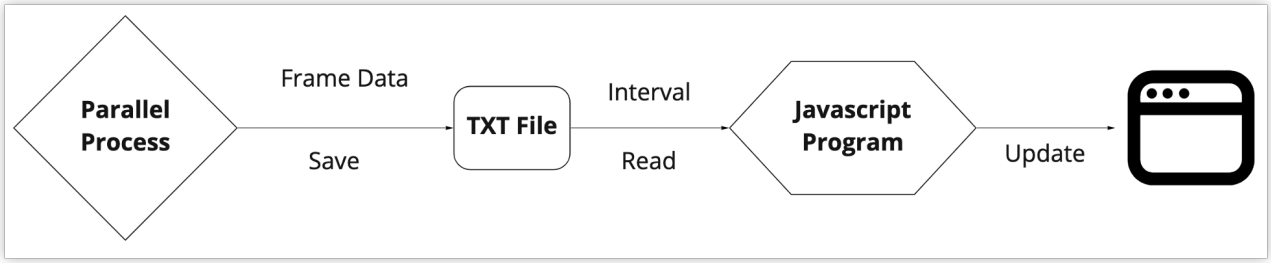


Figure 7: Visualization Design

The code to update the front-end page is shown in code listing 3, which reads the data inside the txt file at the time interval we set to dynamically update the front-end interface.

```
runIteration() {
    let newBoard = this.makeEmptyBoard();
    if(this.state.counter !== this.props.matrix_data.length){
        for (let x = 0; x < this.rows; x++) {
            for (let y = 0;  y < this.cols; y++) {
                newBoard[x][y] = this.props.matrix_data[x + this.state.
   counter][y] === 1 ? true : false;
            }
        }
        this.board = newBoard;
        this.setState({ cells: this.makeCells(), counter: this.state.counter
    + 50, time: this.state.time + this.props.interval /10000 });

        this.timeoutHandler = window.setTimeout(() => {
            this.runIteration();
        }, this.props.interval);
    } else{
        this.stopGame()
        this.setState({ counter: 0 });
    }
}
```

Listing 3: Update World - Front-end

Result

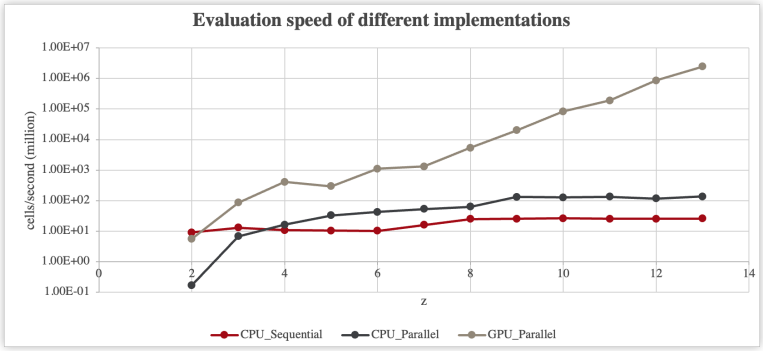


Figure 8: Result - Speed Comparison among different implementations

From the figure 8 we can see two findings:

- Generally, CUDA is faster than CPU parallel and CPU sequential and CPU parallel is faster than sequential one.
- As we increase the world size, we find the speed up of the GPU parallel gets faster and faster, it is because we increase the computing threads during the increasing and it has not reached the maximum threads.

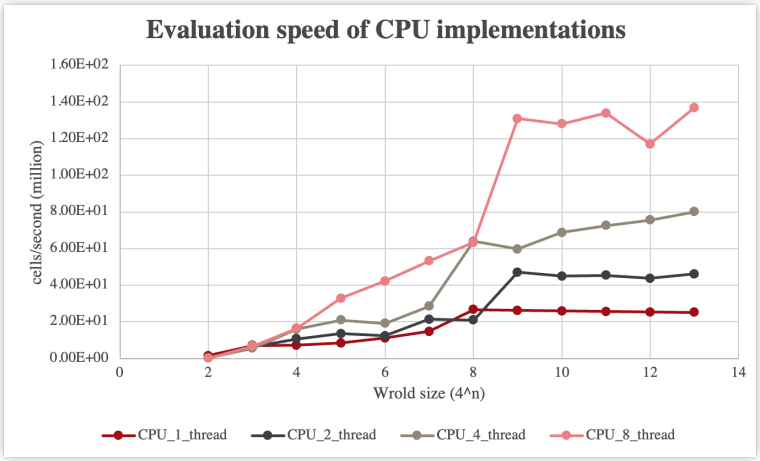


Figure 9: Result - CPU Speed Comparison

From the figure 9 we can see the as we increase the world size

- the speed up of the three methods get increased when the world size is smaller than 4^8, and they are not stable. However, after the world size get larger, the results seemed to be stable and proportional.
- We can see the proportion corresponds how many CPU kernels we used for the computing which verify the correction of the implementation.

From the figure 8 we can see

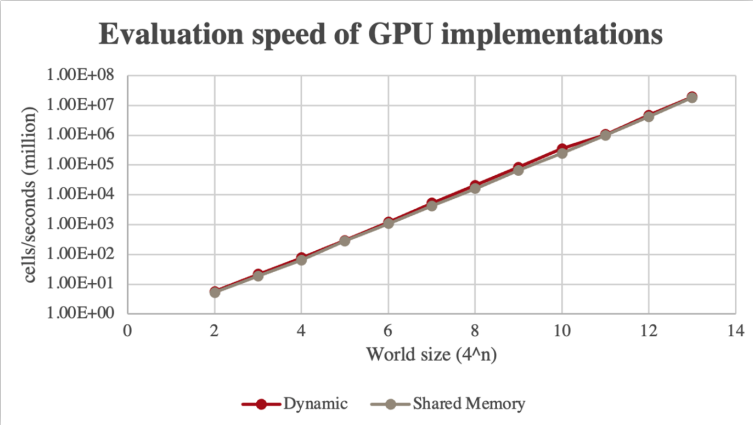


Figure 10: Result - GPU speed Comparison

The speed of the Dynamic method and the shared memory method is similar and the dynamic solution is slightly faster than the shared memory method. It could be because of data swap of the world boundary.

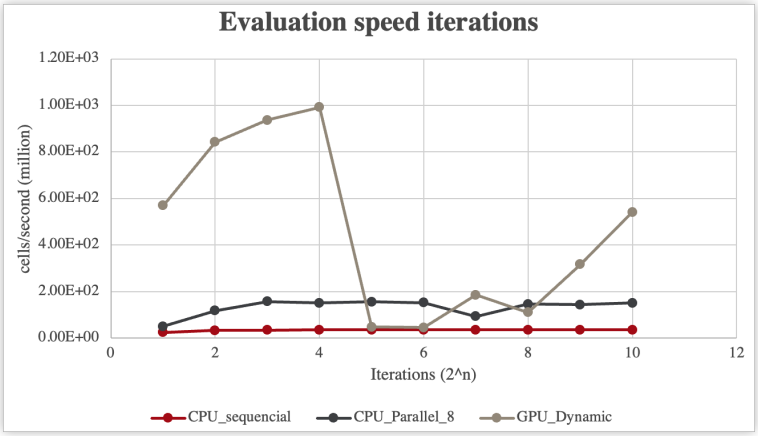


Figure 11: Result - Speed Comparison

The Figure 11 shows as we increase the iterations of the algorithm, we find the GPU dynamic is not so stable as the speed changed in different iterations.

Figure 12 shows the visualization of results in the web page we designed. There are three checkerboards on our web page, representing the CPU sequential, CPU parallel and GPU parallel respectively. Once the run button clicked, it means that this type of computation starts to run. We can see that GPU parallel is the first to finish, followed by CPU parallel, and the slowest is CPU sequential. It takes about twelve seconds.

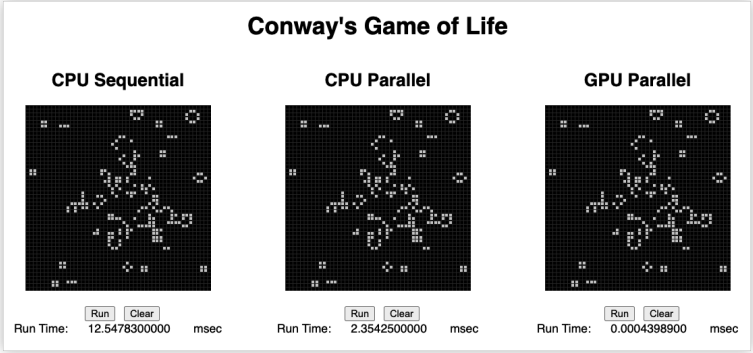


Figure 12: Result - Visualization

Challenges

The challenges we encountered in this project were in two main areas.

The first aspect is how to visualize the results.Initially we want to perform dynamic visualization about the while computing, however it seem quite challenging, so we make visualization with computed results and tested time.

The second part is about how to allocate GPU resources. After repeated experiments, we have succeeded in getting the best results by using grid, block and threads to solve the implementation and computation of a large-scale game map.

Conclusion

This project presented three different evaluation algorithms for Conway's Game of Life: CPU sequential, CPU parallel and GPU parallel. All three algorithms are carefully implemented and the visualization of their results are provided. The best algorithm among of them is the GPU parallel algorithm. It is thousands of times faster than other algorithms. Our findings can be summarized in five points as follows:

- Under most conditions, the more threads using, the faster the computation it performs
- For CPU parallel, the relation between execution speed and number of threads is linear
- Dynamic method performs slightly better than the shared memory method for GPU
- For GPU parallel, the larger block size, the faster computation
- When the world size is very small, a single master thread performs far better than parallel computing

For contribution, all team members were involved in the entire process. Hongyi Wu was mainly responsible for the CPU part, Bin Xu was mainly responsible for the GPU part, and Zhang Yu was mainly responsible for the visualization part.

Reference

[1] Conway J. The game of life[J]. Scientific American, 1970, 223(4): 4.

[2] Fišer, M. (2013, March). Conway's Game of Life on GPU using CUDA: Introduction – Marek-Fiser.com. http://www.marekfiser.com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA

[3] React-GameofLife source code https://github.com/charlee/react-gameoflife