Final Project Report
Introduction to Parallel Computing

110065504 Jhih-Ching Yeh
Professor: Chun-Yi Lee
Institute of Information Systems and Applications
National Tsing Hua University

1 Introduction

1.1 *Motivation*

The theme of this final project is mainly inspired by HW3 Sobel and HW5 N-body Simulation. For the former, HW3 is mainly doing paralization on reading pixels, which can be computed separately from each other, so it is not necessary to synchronize directly after the calculation. At that time, I thought this acceleration method is very intuitive and highly improved. For the latter, HW5 is the program to calculate the time required for the game to simulate 200,000 times, and did the parallel technic out of the loop because the calculation result of this time will be used next time. This kind of game parallel method is very special, which means either parallelizing all for loops is the best option.

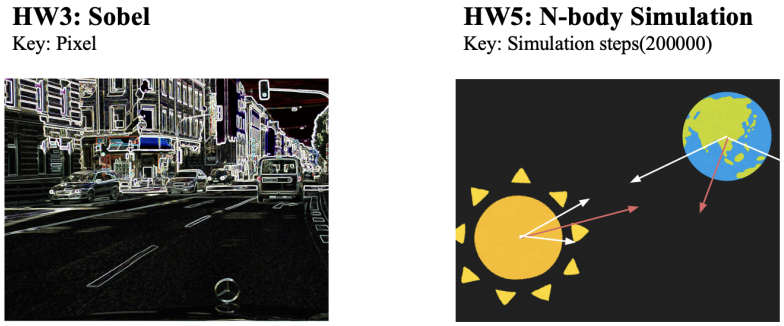

Figure 1: Motivation

All in all, the combination of the two, I came up with the program of Conway's Game

of Life previously designed with OpenGL. Since OpenGL is used for drawing graphics, it is not easy to parallelize, so we use the C programming language to present this game.

1.2 *Background*

The Game of Life, also known simply as Life, is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is a well-known example of a cellular automaton. The game is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players.

In detail, The Game of Life is played on an infinite two-dimensional rectangular grid of cells. Each cell can be either alive or dead. The status of each cell changes each turn of the game (also called a generation) depending on the statuses of that cell's 8 neighbors. The initial pattern is the first generation. The second generation evolves from applying the rules simultaneously to every cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. For each generation of the game, a cell's status in the next generation is determined by a set of rules. These simple rules are as follows:

- Any live cell with fewer than two live neighbors dies, as if by loneliness.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any live cell with two or three live neighbors lives, unchanged, to the next generation.
- Any dead cell with exactly three live neighbors comes to life.

Many different types of patterns occur in the Game of Life. Here, we show the frequently occurring examples of the stable type in the below figure, with live cells shown in black and dead cells in white.

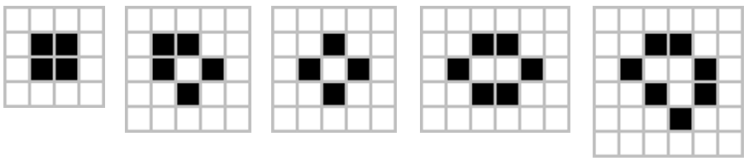

Figure 2: Stable Type

1.3 *Main Idea*

In summary, I implemented a parallel Conway's Game of Life with dynamic work allocation on both CUDA and OpenMP and compared their computing time.

2 Algorithm

The most straightforward algorithm for Life on today's real computers is to store the grid in memory with alive cells as bytes set to 1 and dead cells as bytes set to 0. Two copies of the grid would be required so that the next generation could be calculated from the previous generation without disrupting the values needed for the calculation. Therefore, I designed our model to give a grayscale (1byte) black and white image, after N life cycles, output .png and view the result. Then, compute the difference in operation between opemMP and CUDA after running 1000, 2000 to 5000 times respectively.

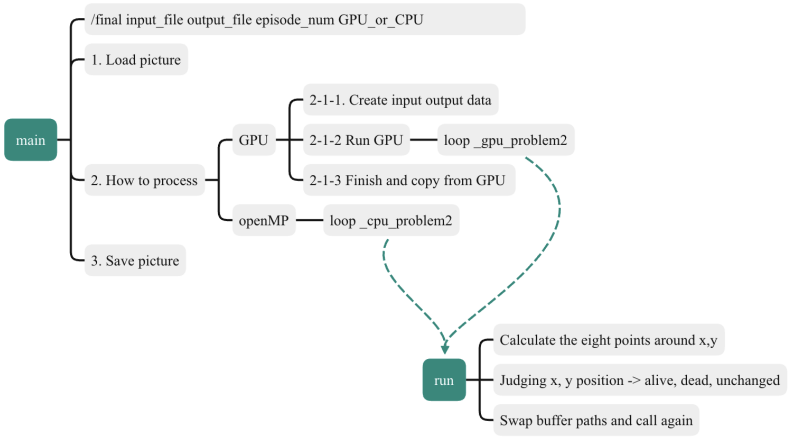

Figure 3: Program Structure

According to the above introduction, we would discuss the algorithm in two parts. The first part is the alive state used to store the cell state each time, and its size is 512*512. The other is the rules, which would adjust the central cell in reference to the state of the 8 cells around. The rules are arranged in the following table and code.

1) Part one: Alive State(512*512)

2) Part Two: Rule(3*3)

Table 1: Table 1 Rule

number	<2	2	3	>3
Alive	Die	Continue to live	Continue to live	Dead
Dead	Remain dead	Remain dead	Come to life	Remain dead

Algorithm 1 ALGORITHM NEXTGENERATION(*cell*)

```
num_neighbors ← cell.COUNT_NUM_NEIGHBORS()
if cell.ISALIVE() then
  if num_neighbors < 2 or num_neighbors > 3 then
    cell.setState(DEAD)
  end if
else
  if num_neighbors = 3 then
    cell.setState(ALIVE)
  end if
end if
```

2.1 *Sequential Code Optimization*

2.1.1 *Rolling Storing Space*

Because the result of this time will be used next time, the program must wait for all pixels to finish executing, which we define it as part one in the above description. To reduce the time, a rolling storing space will be established. The operation function will be called twice during processing, and the interleaving drawing is a loop-seeking.

2.1.2 *Data structure*

In order to improve the time of judge state in the part two of sequential code, I changed the bool variable type with only one digit to a char variable with eight digits. The reason is

that the state is influenced by the other eight surrounding cell states. Thus, the number of alive cells around is at most 8, which is 1000 in binary format. Then, the maximum digit of combining these four digits and their own state is five. Since C++ does not have five-digit variable types, so we choose char type, and the remaining three digits are filled with 0.

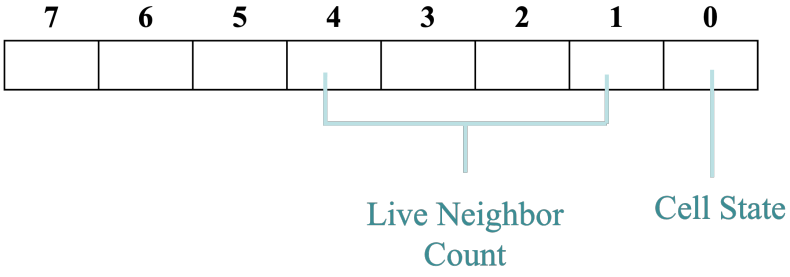

Figure 4: Pruning

2.1.3 *Bounding Box*

In the beginning, I set multiple bounding boxes in the part two. However, I discovered that the computational overhead was not worth the gains of providing finer-grained bounding boxes. The reasons is in the following. First, coalescing and splitting bounding boxes turned out to be a more complex problem than anticipated. Moreover, too many if judgments are not easy to parallelize. Finally, I only set simple bounding box and prun when all eight digits is zero, which means all of them are dead.

2.2 *Parallel Code Optimization*

It is simple to understand how the preceding approach could be parallelized to compute Life. An m by n grid requires performing m n calculations, although none of them are interdependent. So, theoretically, the calculation might be completed entirely in parallel using up to m n processors. Additionally, it can be used for 3 x 3 rule processing.

When I implemented a parallel algorithm on both 3*3 and 512*512 by OpenMP and CUDA respectively. The result in the following table is different from what I thought, the computing time of CUDA is more than OpenMP. After thinking, I consider that it is because the alive state needs to wait for all of the pixels to finish their computation. In other words, take an example. A group of people goes out to have a picnic, and the car will

not be driven until all members arrive. This theory is similar to the thread, that it needs to wait for all of the others.

Table 2: RunTime

Type	OpenMP	CUDA
RunTime	6.58	9.25

2.2.1 *OpenMP*

Since I wrote this program in C++, parallelizing the code listing above is simple. Just add an OpenMP directive telling the compiler to parallelize the loop. By using "#pragma omp parallel for" before the rule and an alive state for loop, we implemented it.

The iterations of the loop are evenly distributed across the default number of threads typically the number of processors available on the machine that are created at the point in the program where the pragma is placed. Except for the implicit barrier at the conclusion of the for loop, no synchronization is required.

It would be preferable to construct threads at a level before the compute next generation() function if several generations need to be computed so that the threads could work on multiple generations with minimal synchronization before being killed. I carried out my implementation in this manner.

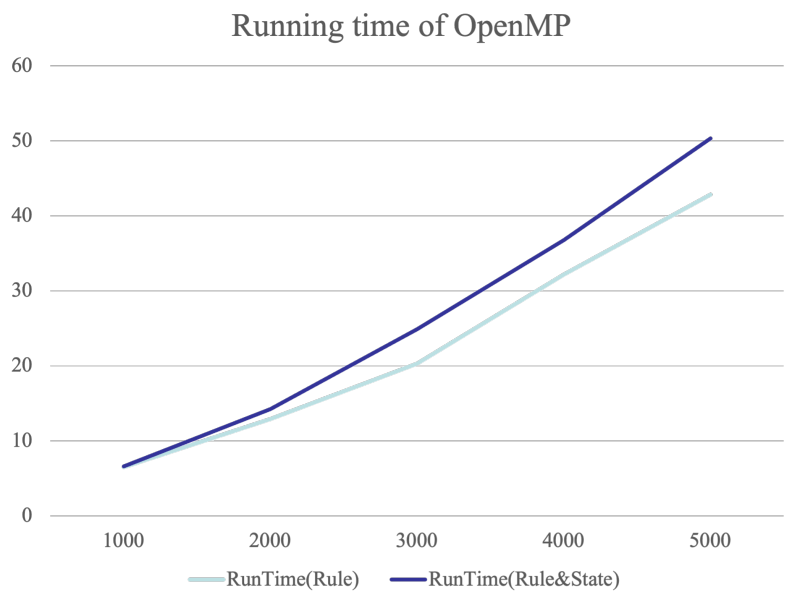

Figure 5: OpenMP Computing Time

The results above show that the computation time for rule and alive state parallel execution is slower, but the difference is not much. According to estimates, the outcome of this alive state must be employed again soon, making synchronization more time-consuming.

2.2.2 *CUDA*

The lack of a for loop in the CUDA code is arguably its weirdest feature. This is due to the fact that each data element will be assigned to one thread if the kernel is called with the appropriate number of thread blocks and threads per block. Although the computation is only expressed for a single element, it is carried out in massively parallel by starting the kernel with the right number of thread blocks and threads per block. On a CPU, this would be a very bad way to write the program. But because the CUDA architecture makes use of extremely lightweight threads that are implemented in hardware, writing the application in this manner on the GPU is both feasible and advantageous. Since the GPU consists of several multiprocessors, each of which may control 32 threads simultaneously, many of these threads can be run concurrently.

The grid of Life cells is separated into thread blocks in the code above that measure 8 by 8 by 1. Consequently, there are 64 threads in each thread block. The grid dimension

must be evenly divisible by 16 as written in the code for all grid cells to be computed. The thread determines which grid element it is in charge of in the first few lines of the kernel; the calculation is then completed in the remaining lines.

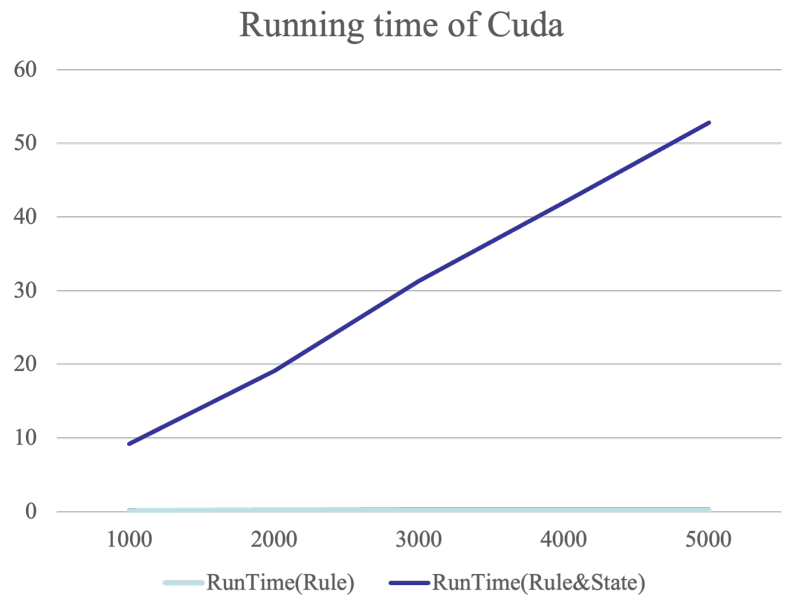

Figure 6: CUDA Computing Time

The aforementioned results demonstrate how considerably slower computation times for rule and living state parallel execution are—the difference is close to 158x. Estimates indicate that the outcome of this living state needs to be used once again soon, adding time to the synchronization process. Because of this, CUDA's implemented rule parallelization yields significantly superior results.

3 Result

The picture on the left in the figure below is the input cell state, and the size is 512*512. The other is the output of a stable cell state after iterations.

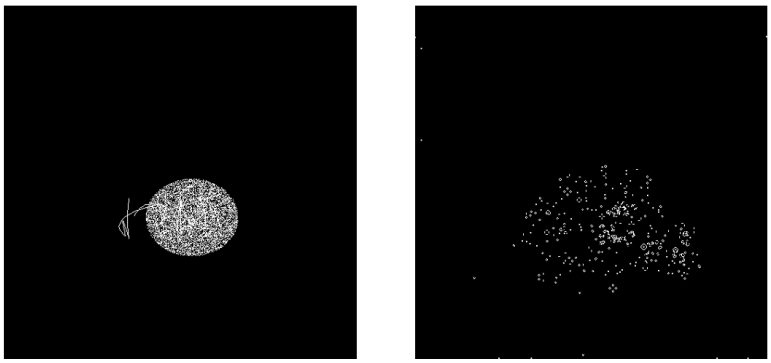

Figure 7: Result of Picture

As can be seen from the following pictures, the parallel time of using CUDA only parallel rule is the fastest, but if using CUDA parallel rule and alive state is the slowest, the difference between the two is huge. Therefore, it is very important to choose a suitable parallel method.

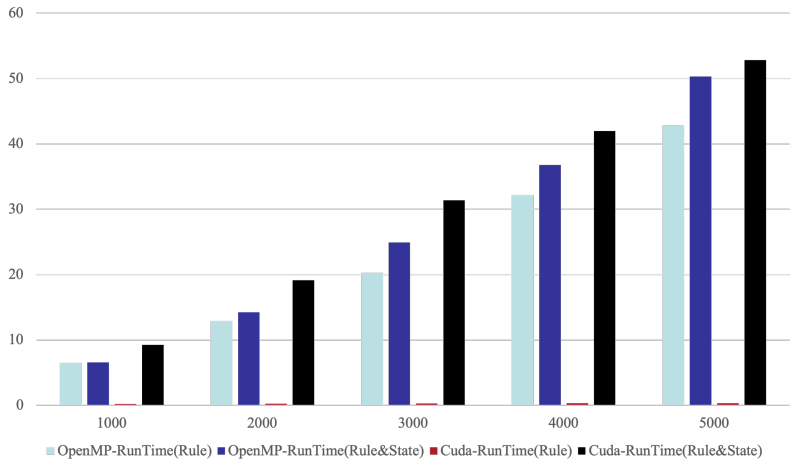

Figure 8: Result

4 Conclusion

In summary, not doing all of parallel function is a good choice since sometimes synchronization takes more time. In this project, it is faster to only parallelize rules by CUDA. Compared with the OpenMP method, it is 128 times faster. The resultant speedups are

rather impressive.

Table 3: Parallelize Way RunTime

Parallelize Way	Rule and Alive Neighbor	Rule
Fast Type	OpenMP	CUDA
RunTime	1.04x	128.81x

5 Discussion

In the class last week, the professor and the TA recommended that adding the stream parallelism function would reduce the computing time. Here I would explain this problem in detail.

When I add the stream to parallelize, the result shows that the computing time is much slower, almost equal to the result when I did all the parallels in section 2.2, about ten seconds. I think the reason is still the same as in section 2.2 because the alive state needs to wait for everyone (512*512) to finish before executing the next time. In summary, that the results of this time will be used next time is the key point.

Because this question was not explained clearly in class, I would like to explain all of the questions in detail here.

1. I put the data on the GPU and did not move the data back to the CPU. I only read it after trying N times.

2. About the above-mentioned reason, "the results of this time will be used next time", it is like saying that a group of people go travel in a tourist bus, and they will drive only when all of them arrive, but after arriving at the destination, they can go and play on their own.

3. Therefore, after removing the parallelism of the alive state, the following two times can be solved.

   1) Thread synchronization

   2) The number of for judgments