实验名称 细胞自动机

一. 实验内容:

假设有一个无限大的二维平面网格，每一个网格和周围的8个网格相邻接，开始时有一些网格被占据（存活），其他的网格为空（死亡）。
按照如下规则进行模拟:
在每一次模拟时:
1.对每一个存活的网格：若其邻居有2或3个存活，则其保持存活，否则使其死亡。
2.对每一个死亡的网格：若其恰好有3个存活的邻居，则使其存活，否则保持死亡。

在实际的问题中，平面是有大小限制的，不对超出限制的网格进行模拟，位于边界的网格在模拟时，将超出边界的网格视作死亡。

输入:
n:输入方阵的阶数。
Matrix：一个Boolean矩阵，代表需要进行模拟的问题空间，true代表该位置的网格生存，false代表死亡。
numGen:进行模拟的次数

输出:
Matrix: 一个Boolean矩阵，表示模拟结束后的问题状态。

要求：测试(n, numGen)=(400，100)，(1000，200)时的实验性能

二. 实验环境:

操作系统:	ubuntu 16.04 LTS 64bit
编译器:	gcc 5.3.1 (for openmp)
	nvcc 7.5.17 (for cuda)
CPU:	Intel i7-4720HQ CPU @ 2.60GHz * 8
GPU:	NVIDIA GTX965M 2G GDDR5
memory:	8GB DDR3L

三. 实现步骤:

算法没什么好说的，实现步骤很明确。

1. OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>
#include <time.h>

#define NUM_THREAD 8

int** create_world(int length, int if_init)
{
    int i, j;
    int** world = (int**)malloc((length + 2) * sizeof(int*));
    if (world == NULL)
```

```
    {
        printf("\nmalloc error!\n");
        exit(0);
    }
    for (i = 0; i < length + 2; i++)
    {
        world[i] = (int*)malloc((length + 2) * sizeof(int));
        if (world[i] == NULL)
        {
            printf("\nmalloc error!\n");
            exit(0);
        }
    }
    for (int k = 0; k < length + 2; k++)
    {
        world[0][k] = world[length + 1][k]
            = world[k][0] = world[k][length + 1] = 0;
    }
    for (int k = 0; k < length + 2; k++)
    {
        world[k] = world[k] + 1;
    }
    world = &(world[1]);
    if (if_init == 1)
    {
        srand(time(0));
        for (i = 0; i < length; i++)
            for (j = 0; j < length; j++)
                world[i][j] = rand() % 2;
    }
    else if (if_init == 2)
    {
        char temp;
        freopen("debug.txt", "r", stdin);
        for (i = 0; i < length; i++)
        {
            for (j = 0; j < length; j++)
            {
                temp = getchar();
                world[i][j] = (temp == '*');
            }
            getchar();
        }
    }
    printf("ok");
    return world;
}

int get_alive_neighbors(int** world, int length, int row, int column)
{
    int rowLeft      = row - 1;
    int rowRight     = row + 1;
    int columnUp     = column - 1;
    int columnBottom = column + 1;

    return world[rowLeft][columnUp] +
```

```
        world[rowLeft][column] +
        world[rowLeft][columnBottom] +
        world[row][columnUp] +
        world[row][columnBottom] +
        world[rowRight][columnUp] +
        world[rowRight][column] +
        world[rowRight][columnBottom];
}

void display_world(int **world, int length)
{
    int i, j;
    putchar('\n');
    for (i = 0; i < length; i++)
    {
        for (j = 0; j < length; j++)
        {
            if (world[i][j] == 0)
                putchar('-');
            else
                putchar('*');
        }
        putchar('\n');
    }
    putchar('\n');
}

int main(int argc, char* arg[])
{
    int i;
    int j;
    double wtime;
    int iter = atoi(arg[1]);
    int length = atoi(arg[2]);
    int** world = create_world(length, 1);
    int** next_world = create_world(length, 0);
    int** temp_world;

    wtime = omp_get_wtime();
    // debug
    // display_world(world, length);
    for (int k = 0; k < iter; k++)
    {
        #pragma omp parallel for private(i, j) num_threads(NUM_THREAD)
        for (i = 0; i < length; i++)
        {
            for (j = 0; j < length; j++)
            {
                int neighbors = get_alive_neighbors(world, length, i, j);
                // debug
                // printf("%d %d %d\n", i + 1, j + 1, neighbors);
                if(world[i][j] == 0)
                    next_world[i][j] = neighbors == 3 ? 1 : 0;
                else if(world[i][j] == 1)
                    next_world[i][j] = (neighbors == 2 || neighbors == 3) ? 1 : 0;
            }
```

```
        }
        // debug
        // display_world(next_world, length);
        temp_world = next_world;
        next_world = world;
        world = temp_world;
    }
    wtime = omp_get_wtime() - wtime;

    printf("time: %lf s\n", wtime);

    return 0;
}
```

2. CUDA

```
#include <stdio.h>
#include <ctime>


static void HandleError( cudaError_t err,
                         const char *file,
                         int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
                file, line );
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))

template< typename T >
void swap( T& a, T& b ) {
    T t = a;
    a = b;
    b = t;
}

struct DataBlock
{
int *outbitmap;
int *dev_in;
int *dev_out;
int *bitmap;
};

__global__ void update(int *in, int *out, int dim){
    int offset = threadIdx.x + blockIdx.x * blockDim.x;
    int x = offset % dim;
    int y = (int)(offset / dim);
    while (offset < dim * dim) {
        int sum = 0;
        for(int i=-1; i < 2; i++) {
            for(int j=-1; j < 2; j++) {
                int xtemp = (x + i + dim) % dim;
                int ytemp = (y + j + dim) % dim;
                int offsettemp = xtemp + ytemp * dim;
                sum = sum + in[offsettemp];
            }
        }
        sum = sum - in[offset];
        if (in[offset] == 1) {
```

```
            if (sum == 2 || sum ==3) {
                out[offset] = 1;
            }
            else {
                out[offset] = 0;
            }
        }
        else {
            if( sum == 3) {
                out[offset] = 1;
            }
            else {
                out[offset] = 0;
            }
        }
        offset = offset + blockDim.x * gridDim.x;
    }
}

int main(int argc, char *argv[]) {

    clock_t start;
    clock_t gpu_start;
    float gpu_comp_time = 0;
    float gpu_mem_to_time = 0, gpu_mem_back_time=0;
    int dim = atoi(argv[1]);
    int nStep = atoi(argv[2]);
//    int frequency = atoi(argv[3]);
    int size = dim * dim;
    int step;
    DataBlock data;
    data.bitmap=(int *)malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        data.bitmap[i] = 0;
    }
    data.bitmap[1]=1;
    data.bitmap[dim+2] = 1;
    data.bitmap[2 * dim + 0] = 1;
    data.bitmap[2 * dim + 1] = 1;
    data.bitmap[2 * dim + 2] = 1;
    data.outbitmap=(int *)malloc(size * sizeof(int));
    int bitmapSize=size * sizeof(int);

    start=clock();

    gpu_start = clock();
    HANDLE_ERROR(cudaMalloc( (void **)&(data.dev_in), bitmapSize));
    HANDLE_ERROR(cudaMalloc( (void **)&(data.dev_out), bitmapSize));
            HANDLE_ERROR(cudaMemcpy(data.dev_in, data.bitmap, bitmapSize,
cudaMemcpyHostToDevice));
    gpu_mem_to_time = ((float)(clock() - gpu_start)) / CLOCKS_PER_SEC;

    // dim3 dimgrid(dim / 16, dim / 16);
    // dim3 dimblock(16, 16);
    int grid_dim;
    int block_dim;
    if (dim < 1024) {
        grid_dim = dim;
        block_dim = dim;
    }
    else {
        grid_dim = 1024;
        block_dim = 1024;
    }

    gpu_start = clock();
```

```
    for(step = 0; step < nStep; step++ ){

       update<<<grid_dim, block_dim>>>(data.dev_in, data.dev_out,dim);

       swap(data.dev_in,data.dev_out);
    }

    cudaDeviceSynchronize();
    gpu_comp_time = ((float)(clock() - gpu_start)) / CLOCKS_PER_SEC;
    gpu_start = clock();
            HANDLE_ERROR(cudaMemcpy(data.outbitmap, data.dev_out, bitmapSize,
cudaMemcpyDeviceToHost));
    gpu_mem_back_time = ((float)(clock() - gpu_start)) / CLOCKS_PER_SEC;
    HANDLE_ERROR(cudaFree(data.dev_out));
    HANDLE_ERROR(cudaFree(data.dev_in));

    printf("%f %f %f ", gpu_comp_time, gpu_mem_to_time, gpu_mem_back_time);
    printf("%f\n", ((float)(clock() - start)) / CLOCKS_PER_SEC);
}
```

四. 实验结果报表

因为问题规模不够大，所以又加了（**5000， 200**）这一个比较大的规模

1. OpenMP

本次实验代码总行数	138 lines
累计耗费时间	？？？

运行时间(sec)

规模 \ 核数	1	2	4	8
(400, 100)	0.1404	0.1389	0.1121	0.07183
(200, 1000)	1.6834	1.6980	1.3507	0.8527
(5000,200)	83.0628	42.5347	26.5055	19.9607

加速比

规模 \ 核数	1	2	4	8
(400, 100)	1	1.0108	1.2525	1.9546
(200, 1000)	1	0.9914	1.2463	1.9742
(5000,200)	1	1.9528	3.1338	4.1613

3.CUDA

本次实验代码总行数	130 lines
累计耗费时间	？？？

运行时间(sec)

规模 \ 线程数	blocks_num = 1024 threads_num = 1024 (把二维数组看作一维数组)
(400, 100)	0.0701
(200, 1000)	0.1372
(5000,200)	1.2808

加速比

规模 \ 线程数	blocks_num = 1024 threads_num = 1024 (把二维数组看作一维数组)
(400, 100)	0.1404
(200, 1000)	1.6834
(5000,200)	83.0628

五. 对本次实验的收获和总结

在这个问题中，可以根据原二维平面，并行地计算迭代一次之后的新的二维平面。因为对于原平面中某一个网格，它在新平面中的存活情况只取决于它在原平面中的邻居，所以这个问题很适合并行化。
另外，从实验结果中可以看出来，当问题规模比较小的时候，串行部分的开销的占比会比较大，所以多线程的结果反而不如串行的结果好。

六. 其他问题（请在符合自己情况的地方填入*）

1.你使用的编程模型是 (多选)

OpenMP	MPI	CUDA	MapReduce
*		*	

注：以下只需填实验中选择的实现方式对应的行，每行单选。

2.在进行并行编程时，我愿意经常使用此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合

OpenMP	*				
MPI					
CUDA		*			
MapReduce					

3.我认为此编程模型有许多非必要而复杂的部分(单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP				*	
MPI					
CUDA				*	
MapReduce					

4.我认为在此编程模型下编程是容易的(单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP	*				
MPI					
CUDA				*	
MapReduce					

5.我认为我需要有技术人员的支持才能使用此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP				*	
MPI					
CUDA				*	
MapReduce					

6.我认为在此编程模型中，有许多功能都很好地整合在一起 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP	*				
MPI					
CUDA		*			
MapReduce					

7.我认为在此编程模型中有许多不一致的地方 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP				*	
MPI					
CUDA				*	
MapReduce					

8.我认为大多数人都能很快地掌握此编程模型 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP		*			
MPI					
CUDA				*	
MapReduce					

9.我认为此编程模型不灵活 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP		*			
MPI					
CUDA				*	
MapReduce					

10.对于在此编程模型下编程，我对问题能够被解决感到很自信 (单选)

	非常符合	比较符合	不确定	比较不符合	非常不符合
OpenMP			*		
MPI					
CUDA			*		
MapReduce					

11.在使用此编程模型前，我需要了解大量的知识(单选)