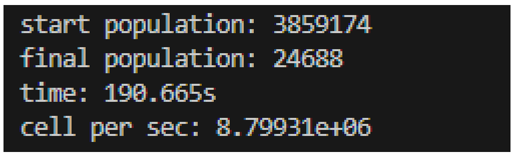
CUDA实验

汤璇 SA24011270

使用自己设备RTX4080完成实验

CPU测试：

纯CPU计算结果如图所示： （以下实验N = 256，T = 100）



```
start population: 3859174
final population: 24688
time: 190.665s
cell per sec: 8.79931e+06
```

未优化的GPU算法实现：

可以将世界划分为三维空间，每个空间都可以独立计算，具有很高的并行度，以下是单次迭代核心代码：

单次迭代函数：

```
__device__ int mod(int x, int n) {
    return (x + n) % n;
}

__device__ bool check(char *universe, int x, int y, int z, int N){
    int alive = 0;
    for (int dx = -1; dx <= 1; ++dx) {
        for (int dy = -1; dy <= 1; ++dy) {
            for (int dz = -1; dz <= 1; ++dz) {
                if (dx == 0 && dy == 0 && dz == 0) continue;
                int nx = mod(x + dx, N);
                int ny = mod(y + dy, N);
                int nz = mod(z + dz, N);
                alive += AT(nx, ny, nz);
            }
        }
    }
    if (AT(x, y, z) && (alive < 5 || alive > 7))
        return 0;
    else if (!AT(x, y, z) && alive == 6)
        return 1;
    else
        return AT(x, y, z);
}
__global__ void update(int N, char *universe, char* next){
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;
    if (x >= N || y >= N || z >= N) return;
    int idx = x * N * N + y * N + z;
    next[idx] = check(universe, x, y, z, N);
}
```

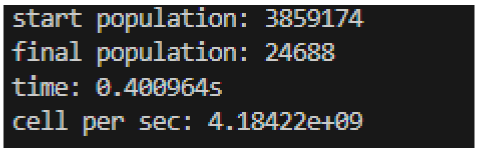
三维索引转一维索引，然后计算邻居存活数，然后迭代更新该位置的状态。

运行函数：

```
__host__ void life3d_run(int N, char *universe, int T)
{
    dim3 blockDim(blockSize, blockSize, blockSize);
    dim3 gridDim((N + blockSize - 1) / blockSize,
    (N + blockSize - 1) / blockSize,
    (N + blockSize - 1) / blockSize);
    size_t size = N * N * N * sizeof(char);
    char *universeInDevice, *next;
    cudaMalloc(&universeInDevice, size);
    cudaMalloc(&next, size);
    cudaMemcpy(universeInDevice, universe, size, cudaMemcpyHostToDevice);
    for(int t = 0; t < T; ++ t){
        update<<<gridDim, blockDim>>>(N, universeInDevice, next);
        cudaDeviceSynchronize();
        std::swap(next, universeInDevice);
        //cudaMemcpy(universe, universeInDevice, size, cudaMemcpyDeviceToHost);
        //std::cout << "Step :" << t + 1 << "\n";
    }
    cudaMemcpy(universe, universeInDevice, size, cudaMemcpyDeviceToHost);
    cudaFree(universeInDevice);
    cudaFree(next);
}
```

首先定义三维世界块的大小，以及分配迭代需要的内存。然后进行 T 次迭代，每次迭代调用update更新状态，最后将最终状态从 device 复制回 host。

实验结果：



```
start population: 3859174
final population: 24688
time: 0.400964s
cell per sec: 4.18422e+09
```

可以看到时间从190s优化到了0.4s，有了巨大的提升，但仍然存在大量对global memory的访问，下面考虑通过使用共享内存，以减少全局内存访问的延迟，从而提高性能。

共享内存优化的算法实现：

只需要讲global memory按块拷贝到shared memory即可，因为要计算邻居，这里shared memory的大小为*(blocksize + 2, blocksize + 2, blocksize + 2)*。

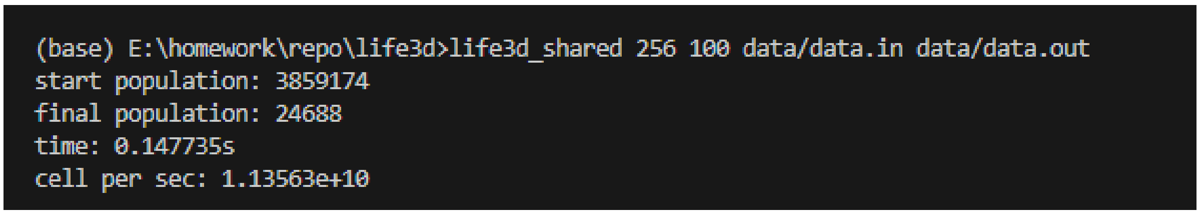
以下是共享内存的更新代码：

```
__global__ void update(int N, char *universe, char* next) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;

    if (x >= N || y >= N || z >= N) return;

    extern __shared__ char shared_mem[];

    int shared_size = blockDim.x + 2;
    int local_x = threadIdx.x + 1;
    int local_y = threadIdx.y + 1;
    int local_z = threadIdx.z + 1;

    int global_idx = x * N * N + y * N + z;
    int shared_idx = local_x * shared_size * shared_size + local_y * shared_size
+ local_z;
    shared_mem[shared_idx] = universe[global_idx];

    // 边界面
    if (threadIdx.x == 0) {
        shared_mem[(local_x - 1) * shared_size * shared_size + local_y *
shared_size + local_z] = universe[mod(x - 1, N) * N * N + y * N + z];
    }
    if (threadIdx.x == blockDim.x - 1 || x == N - 1) {
        shared_mem[(local_x + 1) * shared_size * shared_size + local_y *
shared_size + local_z] = universe[mod(x + 1, N) * N * N + y * N + z];
    }
    if (threadIdx.y == 0) {
        shared_mem[local_x * shared_size * shared_size + (local_y - 1) *
shared_size + local_z] = universe[x * N * N + mod(y - 1, N) * N + z];
    }
    if (threadIdx.y == blockDim.y - 1 || y == N - 1) {
        shared_mem[local_x * shared_size * shared_size + (local_y + 1) *
shared_size + local_z] = universe[x * N * N + mod(y + 1, N) * N + z];
    }
    if (threadIdx.z == 0) {
        shared_mem[local_x * shared_size * shared_size + local_y * shared_size +
(local_z - 1)] = universe[x * N * N + y * N + mod(z - 1, N)];
    }
    if (threadIdx.z == blockDim.z - 1 || z == N - 1) {
        shared_mem[local_x * shared_size * shared_size + local_y * shared_size +
(local_z + 1)] = universe[x * N * N + y * N + mod(z + 1, N)];
    }

    // 边界边
    if (threadIdx.x == 0 && threadIdx.y == 0) {
        shared_mem[(local_x - 1) * shared_size * shared_size + (local_y - 1) *
shared_size + local_z] = universe[mod(x - 1, N) * N * N + mod(y - 1, N) * N + z];
    }
    if (threadIdx.x == blockDim.x - 1 && threadIdx.y == blockDim.y - 1) {
        shared_mem[(local_x + 1) * shared_size * shared_size + (local_y + 1) *
shared_size + local_z] = universe[mod(x + 1, N) * N * N + mod(y + 1, N) * N + z];
    }
    if (threadIdx.x == 0 && threadIdx.z == 0) {
        shared_mem[(local_x - 1) * shared_size * shared_size + local_y *
shared_size + (local_z - 1)] = universe[mod(x - 1, N) * N * N + y * N + mod(z -
1, N)];
    }
    if (threadIdx.x == blockDim.x - 1 && threadIdx.z == blockDim.z - 1) {
        shared_mem[(local_x + 1) * shared_size * shared_size + local_y *
shared_size + (local_z + 1)] = universe[mod(x + 1, N) * N * N + y * N + mod(z +
1, N)];
    }
    if (threadIdx.y == 0 && threadIdx.z == 0) {
        shared_mem[local_x * shared_size * shared_size + (local_y - 1) *
shared_size + (local_z - 1)] = universe[x * N * N + mod(y - 1, N) * N + mod(z -
1, N)];
    }
    if (threadIdx.y == blockDim.y - 1 && threadIdx.z == blockDim.z - 1) {
        shared_mem[local_x * shared_size * shared_size + (local_y + 1) *
shared_size + (local_z + 1)] = universe[x * N * N + mod(y + 1, N) * N + mod(z +
1, N)];
    }
    if (threadIdx.x == 0 && threadIdx.y == blockDim.y - 1) {
        shared_mem[(local_x - 1) * shared_size * shared_size + (local_y + 1) *
shared_size + local_z] = universe[mod(x - 1, N) * N * N + mod(y + 1, N) * N + z];
    }
    if (threadIdx.x == blockDim.x - 1 && threadIdx.y == 0) {
        shared_mem[(local_x + 1) * shared_size * shared_size + (local_y - 1) *
shared_size + local_z] = universe[mod(x + 1, N) * N * N + mod(y - 1, N) * N + z];
    }
    if (threadIdx.x == 0 && threadIdx.z == blockDim.z - 1) {
        shared_mem[(local_x - 1) * shared_size * shared_size + local_y *
shared_size + (local_z + 1)] = universe[mod(x - 1, N) * N * N + y * N + mod(z +
1, N)];
    }
    if (threadIdx.x == blockDim.x - 1 && threadIdx.z == 0) {
        shared_mem[(local_x + 1) * shared_size * shared_size + local_y *
shared_size + (local_z - 1)] = universe[mod(x + 1, N) * N * N + y * N + mod(z -
1, N)];
    }
    if (threadIdx.y == 0 && threadIdx.z == blockDim.z - 1) {
        shared_mem[local_x * shared_size * shared_size + (local_y - 1) *
shared_size + (local_z + 1)] = universe[x * N * N + mod(y - 1, N) * N + mod(z +
1, N)];
    }
    if (threadIdx.y == blockDim.y - 1 && threadIdx.z == 0) {
        shared_mem[local_x * shared_size * shared_size + (local_y + 1) *
shared_size + (local_z - 1)] = universe[x * N * N + mod(y + 1, N) * N + mod(z -
1, N)];
    }
    // 边界点（只在一个线程处理，省去很多分支判断）
    if (threadIdx.x == 0 && threadIdx.y == 0 && threadIdx.z == 0) {
        for (int corner = 0; corner < 8; ++corner) {
            int dx = (corner & 1) ? blockDim.x : -1;
            int dy = (corner & 2) ? blockDim.y : -1;
            int dz = (corner & 4) ? blockDim.z : -1;

            int neighbor_x = mod(x + dx, N);
            int neighbor_y = mod(y + dy, N);
            int neighbor_z = mod(z + dz, N);

            int global_neighbor_idx = neighbor_x * N * N + neighbor_y * N +
neighbor_z;
```

```
            int local_neighbor_x = (dx == -1) ? 0 : (blockDim.x + 1);
            int local_neighbor_y = (dy == -1) ? 0 : (blockDim.y + 1);
            int local_neighbor_z = (dz == -1) ? 0 : (blockDim.z + 1);

            int shared_neighbor_idx = local_neighbor_x * shared_size *
shared_size +
                                      local_neighbor_y * shared_size +
                                      local_neighbor_z;

            shared_mem[shared_neighbor_idx] = universe[global_neighbor_idx];
        }
    }

    __syncthreads();

    int alive = 0;
    for (int dx = -1; dx <= 1; ++dx) {
        for (int dy = -1; dy <= 1; ++dy) {
            for (int dz = -1; dz <= 1; ++dz) {
                if (dx == 0 && dy == 0 && dz == 0) continue;
                alive += shared_mem[(local_x + dx) * shared_size * shared_size +
(local_y + dy) * shared_size + (local_z + dz)];
            }
        }
    }

    if (shared_mem[shared_idx] && (alive < 5 || alive > 7)) {
        next[global_idx] = 0;
    } else if (!shared_mem[shared_idx] && alive == 6) {
        next[global_idx] = 1;
    } else {
        next[global_idx] = shared_mem[shared_idx];
    }
}
```

可以看到，共享内存的拷贝需要考虑每个块的边界面、边界边和边界点，这里边界点处理只用了一个线程来处理，能节省许多分支判断

实验结果：



```
(base) E:\homework\repo\life3d>life3d_shared 256 100 data/data.in data/data.out
start population: 3859174
final population: 24688
time: 0.147735s
cell per sec: 1.13563e+10
```

实验结果得出，利用共享内存优化后的代码比原始CUDA代码提高了一倍的运行速度。