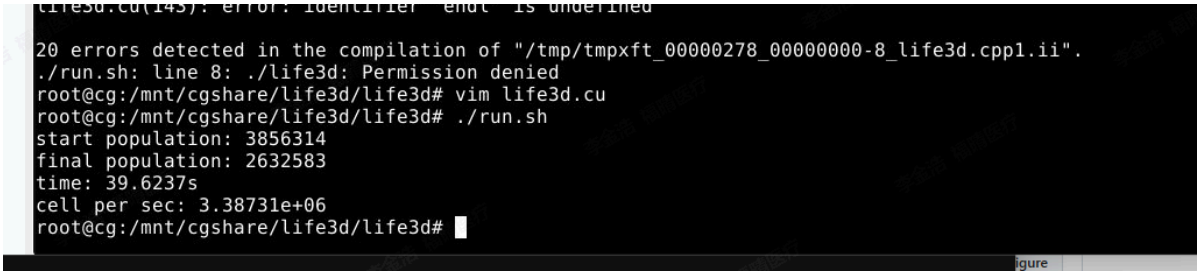
CUDA实验-SA248163-李金浩

1. 基础代码运行

按照实验要求步骤在实验平台上运行基础的cuda代码，基础代码中有一处错误，不然在实验平台上运行会出现命名空间报错，只需要添加using namespace std即可。运行的结果如下所示：



其中两个比较重要的性能指标为time和cell per sec
分别为

time	cell per sec
39.6237s	3.38731e+06

2. 并行化和优化的思路

网格和线程块划分

- 在程序中，程序将三维空间中的每个细胞分配给一个线程处理。整个三维空间被划分为多个线程块，每个线程块又包含多个线程。
- 线程块的大小被设置为*8*8*8，为了适应较大的N值，也就是三维空间的大小。每个线程快的大小可以根据实际的GPU资源和三维空间的大小进行调整，以达到最佳的并行度和负载均衡。

```
dim3 threadsPerBlock(8, 8, 8);
dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (N + threadsPerBlock.y - 1) / threadsPerBlock.y,
                   (N + threadsPerBlock.z - 1) / threadsPerBlock.z);
```

内存管理

- 关于设备分配。在GPU上分配了两个设备内存缓冲区d_universe和d_next_universe，用于存储当前状态和下一个状态的细胞数据。
- 在每次迭代开始时，都会将当前的状态从主机内存复制到设备内存，在每次迭代结束时，将下一个状态从设备内存复制会当前状态的设备内存。
- 双缓冲，通过两个设备内存缓冲区来实现双缓冲，避免了在每次迭代中从设备内存复制数据到主机内存再复制会设备内存的开销。

代码如下：

```
cudaMalloc(&d_universe, size);
cudaMalloc(&d_next_universe, size);
cudaMemcpy(d_universe, universe, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_universe, d_next_universe, size, cudaMemcpyDeviceToDevice);
cudaMemcpy(universe, d_universe, size, cudaMemcpyDeviceToHost);
```

核函数并行计算

- 核函数life3d_run_kernel负责计算每个细胞的下一个状态。每个线程独立计算一个细胞周围的活细胞数，并根据规则更新该细胞的状态。
- 每个线程通过blockIdx和threadIdx计算其在三维空间中的位置，确保各自独立处理不同的细胞。

CUDA核函数的代码如下：

```
__global__ void life3d_run_kernel(int N, char *universe, char *next_universe) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int z = blockIdx.z * blockDim.z + threadIdx.z;

    // 确保计算在宇宙范围内
    if (x >= N || y >= N || z >= N) return;

    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 = (x + dx + N) % N;
                int ny = (y + dy + N) % N;
                int nz = (z + dz + N) % N;
                alive += AT(nx, ny, nz);
            }
        }
    }

    char current_state = AT(x, y, z);
    char next_state = current_state;
    if (current_state && (alive < 5 || alive > 7)) {
        next_state = 0;
    } else if (!current_state && alive == 6) {
        next_state = 1;
    }

    AT_NEXT(x, y, z) = next_state;
}
```

其他优化

- 通过合理的线程块和网格划分，每个线程访问的数据在三维空间中是局部的，这可以提高内存访问的效率。
- 使用双缓冲，减少全局内存的访问次数，仅在迭代结束时将计算结果从d_next_universe复制到d_universe
- 在每次核函数调用后和每次设备同步后，程序检查CUDA错误，来确保计算的正确性和稳定性。

代码如下：

```
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    std::cerr << "CUDA kernel error: " << cudaGetErrorString(err) << std::endl;
    exit(1);
}

err = cudaDeviceSynchronize();
if (err != cudaSuccess) {
    std::cerr << "CUDA sync error: " << cudaGetErrorString(err) << std::endl;
    exit(1);
}
```

3. 运行结果

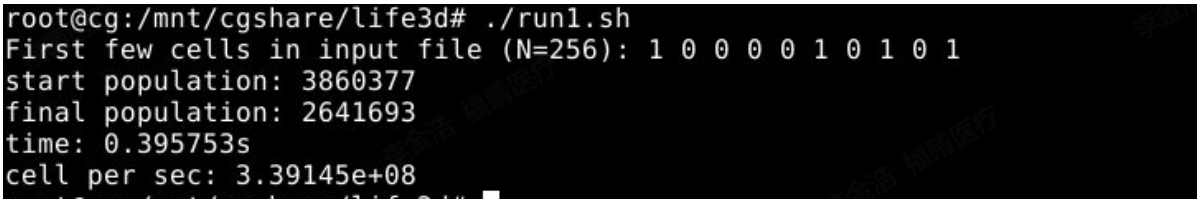
优化后的运行结果

- 运行脚本内容

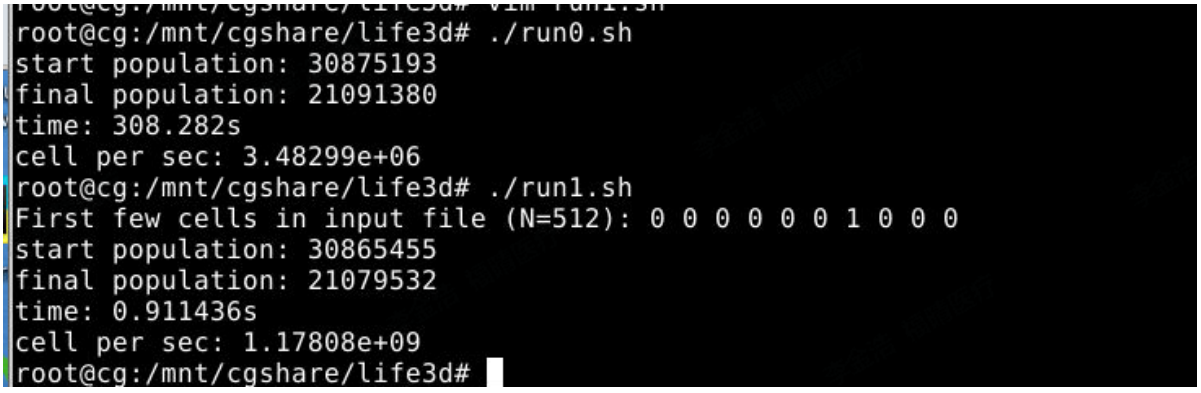
```
#!/usr/bin/bash
N=256
T=8

python3 RandGen.py $N
nvcc life3d.cu -o life3d
./life3d $N $T data/data.in data/data.out
```

- 运行结果1



- 运行结果2



对比分析

类别	start population	final population	time	cell per sec
优化前	3856314	2632583	39.6237s	3.38731e+06
优化后	3860377	2641693	0.395753s	3.39145e+08

类别	start population	final population	time	cell per sec
优化前	30875193	21091380	3082.282s	3.45299e+06
优化后	30865455	21079532	0.911436s	1.17808e+09

当输入的N为128时，处理的速度提高了将近100倍。当我把N调整为512时，性能提高了将近350倍，可以看出，优化的效果还是很不错的