用 GPU 加速运行生命游戏

2021 年 1 月 23 日

1 引言

在一个长宽给定的二维数组 (array) 中，将数组的每个元素视作一个「细胞」，若一个元素的值为 1，代表这个细胞「存活」，否则代表这个细胞不存活，在一次迭代中，对该数组的每一个元素，考虑与它最相邻的 8 个元素，将这 8 个元素的值求和，代表这个细胞它存活的邻居的数量，如果存活的邻居数量少于 2，那么这个细胞死去（通过对它置 0）；如果存活的邻居数量刚好是 2 或者 3，那么这个细胞继续存活；如果存活的邻居的数量超过 3，那这个细胞因为「养分不足」而死去；如果一个已经死去的细胞周围刚刚好有 3 个存活着的细胞，那么它重生（对它置 1）. 这样的计算规则叫做「生命游戏」(Conway's Game of Life)，他是相同维数的数组到数组的对应法则，并且每次计算只考虑每一个元素的「邻居」，这样的对应法则统称为元胞自动机 (cellular automaton). 在每一轮迭代更新中，所有元素都是被同步更新的，所以元胞自动机天然适合在 GPU 上实现.

2 实现

首先我们确定故事的主角，也就是那个二维数组的初始状态，这个可以随意设置:

```
import cupy as cp

initial_state = cp.array([
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
    [0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0],
    [0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,1,0,0,0,0],
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0],
    [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,1,0],
    [0,1,1,0,1,0,1,0,0,1,1,0,0,1,0,1,0,1,1,0],
    [0,0,0,0,1,0,1,0,1,0,0,1,0,1,0,1,0,0,0,0],
    [0,0,0,0,1,0,1,0,1,0,0,1,0,1,0,1,0,0,0,0],
    [0,1,1,0,1,0,1,0,0,1,1,0,0,1,0,1,0,1,1,0],
    [0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,1,0],
    [0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0],
    [0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,1,0,0,0,0],
    [0,0,0,0,0,1,1,0,0,0,0,0,0,1,1,0,0,0,0,0],
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
])
```

那么初始状态确定了，因为生命游戏是不改变数组的长宽的，所以数组的长宽也就确定了，顺便确定好要演化多少代:

```
n_genes = 10
```

```
n_rows = initial_state.shape[0]
n_cols = initial_state.shape[1]
```

我们计划为每一个细胞分配一个 thread 去更新，所以要提前算好资源的分配量：

```
from math import ceil

BLOCK_SIZE = 16
block_dim = (BLOCK_SIZE, BLOCK_SIZE,)
grid_dim = (ceil(n_cols/block_dim[0]), ceil(n_rows/block_dim[1]),)
```

然后，我们生成一个 `n_genes` 层，`n_rows` 行，`n_cols` 列的高维数组，每一代存储在每一层里面，并且将第一层置为初始状态：

```
data = cp.zeros(shape=(n_genes+1, n_rows, n_cols,), dtype=cp.uint32)
data[0, :, :] = initial_state
```

站在一个 thread 的视角上，按照生命游戏的规则去编写每个 thread 的工作剧本：

```
evolve = cp.RawKernel(
    r'''
    #include <cooperative_groups.h>

    using namespace cooperative_groups;

    extern "C" __global__
    void evolve(unsigned int *data, int n_rows, int n_cols, int max_generation)
    {
        if (sizeof(unsigned int) != 4)
        {
            return;
        }

        int row = (unsigned int) (blockIdx.y * blockDim.y + threadIdx.y);
        int col = (unsigned int) (blockIdx.x * blockDim.x + threadIdx.x);

        if ((row >= n_rows) || (col >= n_cols))
        {
            return;
        }

        for (int gen = 0; gen < max_generation; ++gen)
        {
            int l_row = row;
            int l_col = (col - 1) % n_cols;

            int r_row = row;
            int r_col = (col + 1) % n_cols;

            int u_row = (row - 1) % n_rows;
            int u_col = col;

            int d_row = (row + 1) % n_rows;
            int d_col = col;
```

```

        int ul_row = (row - 1) % n_rows;
        int ul_col = (col - 1) % n_cols;

        int ur_row = (row - 1) % n_rows;
        int ur_col = (col + 1) % n_cols;

        int dr_row = (row + 1) % n_rows;
        int dr_col = (col + 1) % n_cols;

        int dl_row = (row + 1) % n_rows;
        int dl_col = (col - 1) % n_cols;

        unsigned n_neighbors = 0;

        n_neighbors += data[gen * n_rows * n_cols + u_row * n_cols + u_col];
        n_neighbors += data[gen * n_rows * n_cols + d_row * n_cols + d_col];
        n_neighbors += data[gen * n_rows * n_cols + l_row * n_cols + l_col];
        n_neighbors += data[gen * n_rows * n_cols + r_row * n_cols + r_col];

        n_neighbors += data[gen * n_rows * n_cols + ur_row * n_cols + ur_col];
        n_neighbors += data[gen * n_rows * n_cols + ul_row * n_cols + ul_col];
        n_neighbors += data[gen * n_rows * n_cols + dr_row * n_cols + dr_col];
        n_neighbors += data[gen * n_rows * n_cols + dl_row * n_cols + dl_col];

        unsigned int this_index = gen * n_rows * n_cols + row * n_cols + col;
        unsigned int next_index = this_index + n_rows * n_cols;

        if (n_neighbors < 2 && data[this_index] == 1)
        {
            data[next_index] = 0;
        }
        else if ((n_neighbors == 2 || n_neighbors == 3) && data[this_index] == 1)
        {
            data[next_index] = 1;
        }
        else if (n_neighbors > 3 && data[this_index] == 1)
        {
            data[next_index] = 0;
        }
        else if (n_neighbors == 3 && data[this_index] == 0)
        {
            data[next_index] = 1;
        }
        else
        {
            data[next_index] = data[this_index];
        }

        grid_group grid = this_grid();
        grid.sync();
```

```
        }
    }
    ''',
    'evolve',
    backend='nvcc',
    enable_cooperative_groups=True
)
```

用到 `grid.sync()` 的原因是每个 thread 在更新每个细胞时，都要考虑这个细胞周围细胞的状态，所以必须得等所有细胞都完工才能进入下一代. 一切都准备好后就可以开始执行了:

```
evolve(grid_dim, block_dim, (data, n_rows, n_cols, n_genes))
```

通过将数组打印出来我们可以看到演化过程:

```
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 3)

for i in range(2):
    for j in range(3):
        k = i * 3 + j
        axs[i, j].matshow(data[k, :, :].get())
        axs[i, j].axis('off')
        axs[i, j].set_title(f"Gene: {k}")

plt.savefig('game-of-life.pdf', bbox_inches='tight')
```

输出见图 2-1

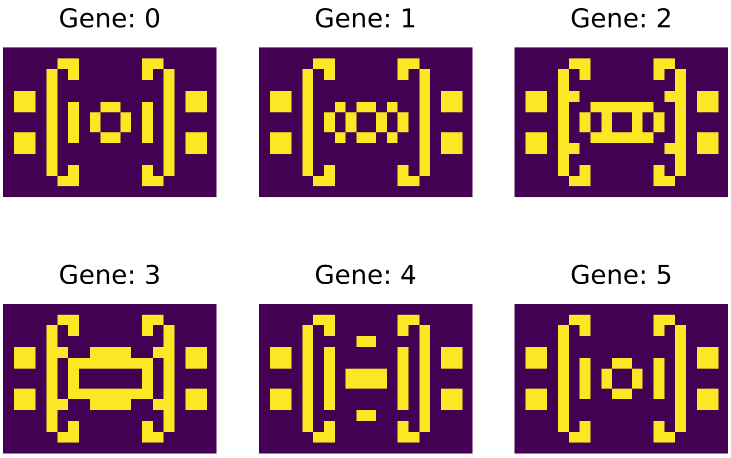

图 2-1: 元胞自动机演化过程

会看到第 6 代也就是 `data[5, :, :]` 和第 1 代 `data[0, :, :]` 是相同的，这说明如果选择某些特殊的初始状态，元胞自动机将会展现出周期性.