用 CUDA 在 GPU 上实现遗传算法

2021 年 1 月 19 日

1 引言

1.1 遗传算法介绍

遗传算法本身并没有严格的定义，但是已经知道的是，它的思想大概是遵循达尔文主义，即适者生存. 大概的图景是这样：对于一个种群 $\mathcal{P}$ 里边的每个个体 p，用一个适应度函数 f 去计算每一个个体 p 的适应度 $f(p)$ 得到种群中每个个体的适应度 $F = \{(p, f(p)) : p \in \mathcal{P}\}$，然后根据 F，留下那些适应度高的，进行变异，并且通过互相杂交来产生下一代，然后再重复前面的过程.

1.2 并行计算

每个个体的杂交、变异和适应度的计算都是可以同时进行的，例如说，对于种群 $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$，在计算适应度的时候，我们可以先计算 $f(p_1)$，再计算 $f(p_4)$，再计算 $f(p_2)$，再计算 $f(p_3)$，亦可以先后分别计算 $f(p_2), f(p_3), f(p_4), f(p_1)$，就是因为这个计算次序对于计算结果是没有影响的，所以遗传算法的进化过程可以很容易地并行化，而这正是 GPU 所擅长的.

2 实现

我们相信，亲自动手去实现它，比空谈概念和公式更加容易学到知识. 所以我们会先一步一步地完成一个 Prototype，再去慢慢研究它.

2.1 适应度函数

遗传算法的最终输出应该是一些适应度比较高的个体，所以我们首先要明确适应函数. 不同的问题，当使用遗传算法的方法来进行建模时，会有不同的适应度函数. 在本篇文章中，我们以经典的 TSP 问题（旅行商问题）为例：一个旅行商要经过 m 个地点，已知这 m 个地点两两之间的距离并且它们都是两两互通的，求最短路径.

首先我们定义距离函数

$$d : \mathbb{N}_m \times \mathbb{N}_m \longrightarrow \mathbb{R}^+ \tag{2.1}$$

$$(i, j) \longmapsto D[i, j] \tag{2.2}$$

这个距离矩阵 D 应该是根据问题的具体情况来确定的，合法的距离矩阵 D 应使得 d 成为一合法的距离算子. 接下来为了模拟自然界生物的进化过程，我们用一个固定长度的行向量来模拟生物体内的染色体，对于规模为 m 的 TSP 问题，这个行向量，也就是一个染色体的维数是 m，例如

$$\boldsymbol{p}_1 = (p_{1,1}, p_{1,2}, \cdots, p_{1,m}) \tag{2.3}$$

称得上是一条染色体，在实际编码中，每一个这样的 $p_{i,j}$ 都是存储在一个 `uint32` 类型的变量中的，因此我们采用的编码方式是二进制编码，当然也有不同的编码方式．CUDA，或者说 GPU，比较擅长处理矩阵，所以，一个种群 $\mathcal{P}$，假设它有 n 个个体，每个个体用一个 m 维向量来表示（在实现中我们还说是 m 个基因位点），可以写成一个 n 行 m 列的矩阵

$$P = \begin{bmatrix} \boldsymbol{p}_1 \\ \boldsymbol{p}_2 \\ \vdots \\ \boldsymbol{p}_n \end{bmatrix} = \begin{bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,m} \end{bmatrix} \tag{2.4}$$

这样一来，「种群」这一概念，在计算机中就变得生动形象而具体了．然而，在每一行，我们只知道 $p_{i,j}$ 是整数，但是它可能有重复的，也可能有不在 $[1, m]$ 中的整数，这可怎么办呢？对 P 的每一行，我们对它排序，然后取无重复的秩 (order)，举例：

$$\boldsymbol{p}_3 = (2, 1, 6, 5, 5, 4, 6) \tag{2.5}$$

我们可以给它的每一个元素一个从小到大的排名，并且其中不让出现重复的排名，类似于对它们求「秩」，只不过对于这个「秩」的结不作处理：

$$\boldsymbol{r}_3 = \mathrm{order}(\boldsymbol{p}_3) = (2, 1, 6, 5, 4, 3, 7) \tag{2.6}$$

我们可以这样理解，$\boldsymbol{r}_i$ 是这样的 m 维向量，它使得 $\boldsymbol{p}_i[\boldsymbol{r}_i[j]] \geq \boldsymbol{p}_i[\boldsymbol{r}_i[j+1]]$, $\forall j \in [1, m]$．对 P 的每一行求秩，记

$$R = \mathrm{order}(R) \tag{2.7}$$

那么 R 的每一行就都能够拿来表示一条路径[1]．

设 f 是适应度函数，那么 $f(\boldsymbol{p}_i)$ 是这样子计算：

$$f(\boldsymbol{p}_i) = -\sum_{j=1}^{m} d(R[i, j], R[i, j+1]) \tag{2.8}$$

式中，$R[i, j]$ 表示矩阵 R 第 i 行第 j 列的元素．那么适应度函数也就是把 $\boldsymbol{p}_i$ 所描述的那条路径的距离算出来．并且，对任意整数 $1 \leq i \leq n$，我们约定：$R[i, m+1] = R[i, 0]$.

2.2 编程实现

我们要用到 CuPy，它是 CUDA 在 Python 上的编程接口，另外它还很大程度上兼容 NumPy 和 SciPy，方便对矩阵和高维数组进行处理．整个程序都是基于 CuPy 的.

```
1  import cupy as cp
2
3  # 生成初始种群，种群有 pop_size 个个体，每个个体有 problem_size 个基因位点
4  pop_size = 100
5  problem_size = 12
6  population = cp.random.randint(
7      low = 0, high = problem_size,
8      size = (pop_size, problem_size,),
9      dtype = cp.uint32
10 )
```

[1] 一条路径是 m 个地点序号构成的有序元组，且其中无重复、无遗漏.

```
11
12 distance_mat = cp.random.randint(
13     low = 1,
14     high = 1000,
15     size = (problem_size, problem_size,)
16 ).astype(cp.float32)
```

在上列代码中，我们先随机生成了一个规模为 pop_size 的种群，然后又随机生成了一个长宽皆为 problem_size 的矩阵来描述 TSP 问题中任意两个地点之间的距离.

接下来，我们要计算每个个体的得分，在此之前，需要先计算出路径:

```
routes = cp.argsort(population, axis = 1).astype(cp.uint32)
```

算出路径之后，算距离:

```
1  # 输入:
2  # in_routes: 每一行对应一个 route , 每一个 route 是 vertex 的列表
3  # in_distance_mat: 它的 [i, j] 元素 是 vertex i 到 vertex j 的距离
4  # problem_size: in_routes 的列维数
5  # pop_size: in_routes 的行维数
6  #
7  # 输出:
8  # out_distance: 是输出值, 对应 in_routes 每一行所对应的 route 的距离
9  distance_kernel = cp.RawKernel(
10     r'''
11     extern "C" __global__
12     void distance(
13         unsigned int *in_routes,
14
15         float *in_distance_mat,
16         float *out_distances,
17
18         int problem_size,
19         int pop_size
20     )
21     {
22         int row = blockIdx.y * blockDim.y + threadIdx.y;
23         int col = blockIdx.x * blockDim.x + threadIdx.x;
24
25         if (row >= pop_size || col >= problem_size)
26         {
27             return;
28         }
29
30         int from = in_routes[row * problem_size + col];
31         int to = in_routes[row * problem_size + 0];
32
33         if (col < problem_size - 1)
34         {
35             to = in_routes[row * problem_size + col + 1];
36         }
37
38         out_distances[row * problem_size + col] =
```

```
        in_distance_mat[from * problem_size + to];
    }
    ''',
    'distance'
)
```

我们打算让每一个 CUDA thread 去计算一个 $R[i,j]$ 到 $R[i,j+1]$ 的距离，一样地，还是规定 $R[i,m+1]=R[i,0]$．每个 thread 把这段距离计算好后，就会记在输出矩阵的 $[i,j]$ 位置．形象一点，比如说 R 是这样的

$$R=\begin{bmatrix}2&0&1\\1&2&0\end{bmatrix} \tag{2.9}$$

然后输出矩阵，记做 O，是

$$O=\begin{bmatrix}12&100&20\\20&12&20\end{bmatrix} \tag{2.10}$$

那么矩阵 O 的第一行的 12 就表示从 2 到 0 的距离是 12，100 表示从 0 到 1 的距离是 100，20 表示从 1 到 2 的距离是 20，第二行同理．然后再对这个 D 的每一行求和就可以了：

```
# 确定要调用的 GPU 资源的规模，参看 CUDA Programming Guide
# out_distances[i, j] 代表 routes[i] 的从 routes[i, j] 走到 routes[i, j+1] 的距离
# 如果 j+1 == routes.shape[1]，则 out_distances[i, j] = 距离[routes[i, j], routes[i, 0]]
BLOCK_SIZE = 16
dim_block = (BLOCK_SIZE, BLOCK_SIZE, )
dim_grid = (ceil(self.n_genes/dim_block[0]), ceil(self.genes.shape[0]/dim_block[1]), )
out_distances = cp.zeros(shape = self.genes.shape, dtype = cp.float32)
distance_kernel(
    dim_grid,
    dim_block,
    (routes, distance_mat, out_distances, problem_size, pop_size,)
)

# distances_per_sample[i] 是 routes[i] 走过的总距离，
# 也就是，从 routes[i, 0] 出发，再回到 routes[i, 0] 走过的总距离
distances_per_sample = cp.sum(out_distances, axis=1)

# 距离越长，得分越低
scores = cp.subtract(0, distances_per_sample)
```

这样就计算出来每个个体的得分了，这些得分保存在变量 `scores` 中．有了得分，进一步地，我们用概率来表示每个个体的生存几率：

```
# 升序排列，数字越高，对应的适应度越高
order = cp.argsort(cp.argsort(scores))

# probs[i] 是第 i 个个体被选入下一代的概率
probs = cp.divide(order, cp.sum(order))
```

其中 `probs` 这个变量将会被多次用到．

有了适应度和生存几率，就可以进行选择了：

```
# 那么就按照 probs 作为概率，选出下一代
next_gen_indexes = cp.unique(cp.random.choice(
    pop_size,
```

```
    size = pop_size,
    p = probs
))

population = population[next_gen_indexes, :]
```

经过选择后，适应度较高的容易留下来，适应度低的不容易留下来.

留下来的那些，逃过了自然选择，逃不过宇宙规律，因此还要随机地参与基因突变，这是为了增加基因库的多样性，避免算法过早收敛于局部:

```
# 随机确定选哪些个 population 进行 mutate
# 如果 mutate[i] == 1，则表示要对 population[i] 进行 mutate 操作
mutate = cp.random.randint(
    low = 0,
    high = 2,
    size = (pop_size, 1),
    dtype = cp.uint16
)

# 对那些被选中的个体，随机选择一个基因位点进行操作
col_ind = cp.random.randint(
    low = 0,
    high = problem_size,
    size = (pop_size, 1),
    dtype = cp.uint32
)

# 对每个基因位点，随机选择一个 bit 进行翻转
# 每个基因位点存储在一个 unsigned int 中，是 32 位的
# 所以生成 [0, 31] 也就是 [0, 32) 的随机整数来决定翻转哪一个 bit
bit_ind = cp.random.randint(
    low = 0,
    high = 32,
    size = (pop_size, 1),
    dtype = cp.uint32
)

# 设定 GPU 资源参数，开始调用
BLOCK_SIZE = 16
dim_block = (1, BLOCK_SIZE, )
dim_grid = (1, ceil(pop_size/dim_block[1]),)
bit_flop_kernel(
    dim_grid,
    dim_block,
    (population, pop_size, problem_size, col_ind, bit_ind, mutate, )
)
```

代码中的 `bit_flop_kernel` 是 CUDA 核函数, 我们实际上为每一个个体分配一个 thread, 对于下标为 i 的 thread, 这个 thread 读取 `mutate[i]` 来决定是否继续, 如果继续, 则读取 `col_ind` 和 `bit_ind` 来决定具体怎么变异:

```
对于 i = 1, 2, ..., n_rows,
```

```
# 将 data[i, col_ind[i]] 的 bit_ind[i] 位翻转
# n_rows 是 data 的行维数，n_cols 是 data 的列维数
# mutate[i] 取 0 表示 data[i] 不参与位翻转，取 1 表示参与
bit_flop_kernel = cp.RawKernel(
    r'''
    extern "C" __global__
    void bit_flop(
        unsigned int *data,
        int n_rows,
        int n_cols,

        unsigned int *col_ind,
        unsigned int *bit_ind,

        unsigned short *mutate
    )
    {
        int row = blockIdx.y * blockDim.y + threadIdx.y;
        int col = blockIdx.x * blockDim.x + threadIdx.x;

        if (row >= n_rows || col >= 1)
        {
            return;
        }

        if (mutate[row] == 0)
        {
            return;
        }

        unsigned int mask = 1;
        mask = mask << bit_ind[row];
        data[row * n_cols + col_ind[row]] ^= mask;

    }
    ''',
    'bit_flop'
)
```

变异之后就是杂交，是为了把变异产生的新基因进行传播：

```
同时让每一个 population 开始 cross
# population 是要参与 cross 的，每一行对应一个个体
# match 是一个 pop_size 行的向量，
# 并且 population[i] 将与 population[match[i]] 进行 cross
# 让 population[i] 的左半部分与 population[match[i]] 的右半部分拼接，得一个后代
# 再让 population[i] 的右半部分与 population[match[i]] 的左半部分拼接，得一个后代
# born 是 2 * pop_size 行的，列维数与 population 的相同也是 n_cols
cross_kernel = cp.RawKernel(
    r'''
    extern "C" __global__
    void cross(
```

```
12          unsigned int *population,
13          unsigned int *match,
14          unsigned int *born,
15
16          int n_rows,
17          int n_cols
18      )
19      {
20          int row = blockIdx.y * blockDim.y + threadIdx.y;
21          if (row >= n_rows || n_cols < 2)
22          {
23              return;
24          }
25
26          int mid_point = 0;
27          if (n_cols % 2 == 0)
28          {
29              mid_point = (n_cols / 2) - 1;
30          }
31          else
32          {
33              mid_point = (n_cols-1) / 2;
34          }
35
36          if (threadIdx.x == 0)
37          {
38              int i = 0;
39              while (i <= mid_point)
40              {
41                  born[2 * row * n_cols + i] = population[row * n_cols + i];
42                  i = i + 1;
43              }
44
45              while (i <= (n_cols-1))
46              {
47                  born[2 * row * n_cols + i] = population[match[row] * n_cols + i];
48                  i = i + 1;
49              }
50          }
51          else if (threadIdx.x == 1)
52          {
53              int i = 0;
54              while (i <= mid_point)
55              {
56                  born[(2*row + 1) * n_cols + i] = population[match[row] * n_cols + i];
57                  i = i + 1;
58              }
59
60              while (i <= (n_cols-1))
61              {
62                  born[(2*row + 1) * n_cols + i] = population[row * n_cols + i];
```

```
                i = i + 1;
            }
        }
        else
        {
            return;
        }
    }
    ''',
    'cross'
)
```

每个个体产生两个子带，所以，每个个体将被分配到 2 个 CUDA thread 进行处理，具体地，对于下标为`i`的个体，假设它将和下标为`j` 的个体进行杂交 [2]，那么个体`i` 的左半部分与个体`j` 的右半部分产生一个个体，`i` 的右半部分与`j` 的左半部分再产生一个个体，这样就产生了两个子代．具体可以参考上列代码．

2.3 运行展示

我们已经将完整的代码放在了仓库 `hsiaofongw/genetic-algorithm-cuda`，限于篇幅这里就不全部列出来了．

```
[3]: import cupy as cp
     from genetic import Population

     # 生成初始种群，种群有 pop_size 个个体，每个个体有 problem_size 个基因位点
     pop_size = 100
     problem_size = 12
     population = Population.generate_initial_population(
         pop_size,
         problem_size
     )

     distance_mat = cp.random.randint(
         low = 1,
         high = 1000,
         size = (problem_size, problem_size,)
     ).astype(cp.float32)
```

```
[4]: max_evolves_n = 100
     for i in range(max_evolves_n):
         population.evolve(distance_mat)
         population.update_chance(distance_mat)
         print("At generation: %s\n Score: %s\n" % (str(i), population.get_maximum_score(),))

At generation: 0
 Score: -2721.0

At generation: 1
 Score: -2502.0

At generation: 2
 Score: -2303.0

At generation: 3
 Score: -1914.0

At generation: 6
 Score: -1794.0

At generation: 7
 Score: -1579.0

At generation: 8
 Score: -1579.0

At generation: 9
 Score: -1274.0

At generation: 10
 Score: -1579.0

At generation: 11
 Score: -1579.0

At generation: 12
 Score: -1579.0

At generation: 13
 Score: -1565.0

At generation: 14
 Score: -1388.0

At generation: 15
 Score: -1388.0

At generation: 16
 Score: -1388.0
```

图 2-1: 设定参数

图 2-2: 快速收敛

图 2-1设定了参数，初始种群规模为 100，TSP 问题规模为 12，距离矩阵是随机生成的，每个元素是 1 到 999 范围内的整数；图 2-2现实了模型在经过仅仅 9 代就收敛到了一个较好的值 (Score $= -1274.0$).

[2]每个个体的这个 `j` 是多少也是根据 `probs` 变量随机确定的，`j` 取到 `t` 的几率是 `probs[t]`.

3 补充说明

3.1 遗传算法的参数

遗传算法虽然容易理解也容易实现，可是有许多不确定的地方，包括：

- 种群规模
- 变异方式
- 选择方式
- 编码方式
- 交叉方式

如果种群规模太小，程序容易过早收敛，如果规模太大，一是硬件条件可能不满足（超出显存/内存限制）；二是收敛速度也会变慢．至于变异方式也有很多灵活的实现方式：每一轮选谁变异？选多少个体进行变异？每个个体选多少个 bit 进行位反转？这些都有待尝试；不过可以确定的是，如果变异率太高，将会导致算法不稳定，因为过高的变异率使得优秀基因难以长久留存；然而，如果变异率太低，程序又容易收敛到局部最优解（过早收敛）．编码方式是采用二进制编码还是格雷码 (Gray Code)，也有不一样的效果．

3.2 本地测试

如果手头没有支持 CUDA 的设备，我们也不是不可以写 CUDA 代码，因为我们可以远程租用 GPU，但是由于这样做成本比较高，所以，作者个人推荐的开发方法是：先在本地认真仔细测试好代码，再放到远端 GPU 设备上去跑．

下面是一部分本地测试代码，用常见的 C++ 编译器就可以编译它并且在本地没有 CUDA 支持的设备上运行：

```
#include <iostream>
#include <random>
#include <math.h>

typedef struct
{
    int x;
    int y;
    int z;

} dim3;

void print_matrix(unsigned int *mat, int n_rows, int n_cols)
{
    for (int i = 0; i < n_rows; i++) {
        for (int j = 0; j < n_cols; j++) {
            std::cout << mat[i * n_cols + j] << ", ";
        }
        std::cout << std::endl;
    }
}

```

```
23 void cross(
24
25     dim3 blockIdx,
26     dim3 blockDim,
27     dim3 threadIdx,
28
29     unsigned int *population,
30     unsigned int *match,
31     unsigned int *born,
32
33     int n_rows,
34     int n_cols
35 )
36 {
37     int row = blockIdx.y * blockDim.y + threadIdx.y;
38     if (row >= n_rows || n_cols < 2)
39     {
40         return;
41     }
42
43     int mid_point = 0;
44     if (n_cols % 2 == 0)
45     {
46         mid_point = (n_cols / 2) - 1;
47     }
48     else
49     {
50         mid_point = (n_cols-1) / 2;
51     }
52
53     if (threadIdx.x == 0)
54     {
55         int i = 0;
56         while (i <= mid_point)
57         {
58             born[2 * row * n_cols + i] = population[row * n_cols + i];
59             i = i + 1;
60         }
61
62         while (i <= (n_cols-1))
63         {
64             born[2 * row * n_cols + i] = population[match[row] * n_cols + i];
65             i = i + 1;
66         }
67     }
68     else if (threadIdx.x == 1)
69     {
70         int i = 0;
71         while (i <= mid_point)
72         {
73             born[(2*row + 1) * n_cols + i] = population[match[row] * n_cols + i];
```

```
            i = i + 1;
        }

        while (i <= (n_cols-1))
        {
            born[(2*row + 1) * n_cols + i] = population[row * n_cols + i];
            i = i + 1;
        }
    }
    else
    {
        return;
    }
}

int main()
{
    int n_rows = 6;
    int n_cols = 8;
    unsigned int *data = (unsigned int *) malloc(n_rows * n_cols * sizeof(unsigned int));
    unsigned int *match = (unsigned int *) malloc(n_rows * sizeof(unsigned int));
    unsigned int *born = (unsigned int *) malloc(2 * n_rows * n_cols * sizeof(unsigned int));

    std::default_random_engine generator;
    std::uniform_int_distribution<unsigned int> distribution(0, n_rows-1);

    for (int i = 0; i < n_rows; i++)
    {
        match[i] = distribution(generator);

        for (int j = 0; j < n_cols; j++)
        {
            data[i * n_cols + j] = distribution(generator);
        }
    }

    for (int i = 0; i < n_rows; i++)
    {
        for (int j = 0; j < n_cols; j++)
        {
            born[2 * i * n_cols + j] = 0;
            born[(2 * i + 1) * n_cols + j] = 0;
        }
    }

    const int BLOCK_SIZE = 16;

    dim3 blockDim;
    blockDim.x = 2;
    blockDim.y = BLOCK_SIZE;
    blockDim.z = 1;
```

```
125
126     dim3 gridDim;
127     gridDim.x = 1;
128     gridDim.y = (unsigned int) ceil((((double) n_rows)+0.0000001)/blockDim.y);
129     gridDim.z = 1;
130
131     dim3 threadIdx;
132     dim3 blockIdx;
133
134     for (blockIdx.x = 0; blockIdx.x < gridDim.x; blockIdx.x += 1)
135     {
136         for (blockIdx.y = 0; blockIdx.y < gridDim.y; blockIdx.y += 1)
137         {
138             for (threadIdx.x = 0; threadIdx.x < blockDim.x; threadIdx.x += 1)
139             {
140                 for (threadIdx.y = 0; threadIdx.y < blockDim.y; threadIdx.y += 1)
141                 {
142                     cross(
143                         blockIdx,
144                         blockDim,
145                         threadIdx,
146                         data,
147                         match,
148                         born,
149                         n_rows,
150                         n_cols
151                     );
152                 }
153             }
154         }
155     }
156
157     print_matrix(data, n_rows, n_cols);
158     std::cout << std::endl;
159     print_matrix(match, n_rows, 1);
160     std::cout << std::endl;
161     print_matrix(born, 2*n_rows, n_cols);
162     std::cout << std::endl;
163
164     free(data);
165     free(match);
166     free(born);
167 }
```

在上列代码中，我们主要是测试了核函数 `cross` 能否正常工作，为此，我们模拟了 CUDA 运行时的一部分环境，譬如说模拟了 `blockDim`, `gridDim`, `blockIdx`, `threadIdx` 等变量.

4 总结

我们首先介绍了遗传算法，然后指出遗传算法可以被并行化，然后列出了几个关键部分的典型实现，通过部分代码，介绍了我们是如何用 CUDA 来实现遗传算法的，也讲解了我们是如何将计算任务分配到每一个 thread 上面去的，最后我们给出了两张运行图示，最后引出了遗传算法的参数的问题，并且介绍了一种在本地测试 CUDA 代码的方式.