# 多核作业 2

18340146 计算机科学与技术 宋渝杰

## 实验题目

计算二维数组中以每个元素为中心的熵 $H(x)=-\sum_i p_i \log p_i$，其中 $p_i=p(X=x_i)$

输入：二维数组及其大小（在这里我的程序输入方式为：先输入矩阵的高和宽，之后输入二维数组）

- 二维数组的元素均为 [0, 15] 的整形

输出：浮点型二维数组（保留 5 位小数）

- 每个元素中的值为以该元素为中心的大小为 5 的窗口中值的熵
- 当元素位于数组的边界窗口越界时，只考虑数组内的值

## 实验过程

### 基础版本（baseline）程序

下文先介绍一下 baseline 程序：

### 介绍程序整体逻辑，包含的函数，每个函数完成的内容

**整体逻辑：**cuda 一个线程计算输出矩阵（下面称为矩阵 B）一个位置的值，计算原理为计算输入矩阵（下面称为矩阵 A）的以该位置为中心的大小为 5 的窗口中值的熵。

**包含的函数：**程序关键函数为 cuda 核函数 `cal()`，辅助函数有打印矩阵函数 `print()`，而申请内存、输入矩阵、计时等都放在 `main()` 函数执行。

**每个函数完成的内容：**

**核函数 `cal()`：**一个线程所分配的任务为【计算矩阵 B 一个位置的值】，代码层面上的计算方式为：

1. 线程取得自己需要计算的位置的 x、y 坐标（通过 blockIdx * blockDim + threadIdx 得到）。
2. 判断坐标是否越界（因为矩阵大小不一定整除线程块大小），不越界则进行下一步的计算。
3. 计算自己负责的矩阵 A 的窗口中数值 [0, 15] 分别的个数（用一个数组 $num$ 记录个数，遍历窗口，之后对应位置++即可）。
4. 计算熵，按照下述公式计算即可：

$$H(x)=-\sum_i p_i \log p_i=-\sum_i \frac{num_i}{sum}*\log\frac{num_i}{sum}$$

   其中 $num_i$ 为窗口内数值 $i$ 的个数，$sum$ 为窗口内所有数值的个数。

5. 将计算结果赋值到矩阵 B 相应位置上。

```
struct Matrix{    // 矩阵结构体
```

```
    int x, y;     // 高、宽
    double** v;  // 二维矩阵
};

/*
################################################################
##  函数：cal
##  函数描述：核函数，计算输入矩阵a的熵，计算结果存在矩阵b里
##  参数描述：
##  Matrix *a：输入矩阵a
##  Matrix *b：结果矩阵b
################################################################
*/

__global__ void cal(Matrix *a, Matrix *b) {
    int x = blockIdx.y * blockDim.y + threadIdx.y, y = blockIdx.x * blockDim.x +
threadIdx.x; // 1
    if (x < a->x && y < a->y) { // 2
        int num[16] = {0}, sum = 0;
        double ans = 0;
        for (int i = max(x - 2, 0); i < min(x + 3, a->x); i++) {
            for (int j = max(y - 2, 0); j < min(y + 3, a->y); j++) {
                num[(int)(a->v[i][j])]++; // 3
                sum++;
            }
        }
        for (int i = 0; i < 16; i++) // 4
            if (num[i]) ans -= (double)num[i] / sum * log((double)num[i] / sum);
        b->v[x][y] = ans; // 5
    }
}
```

自然地，每个线程块负责的是矩阵 B 一个小区域值的计算。

**打印矩阵函数 `print()`**：接收矩阵指针参数，并按行列将矩阵值进行输出即可（记得保留五位小数）。

```
/*
################################################################
##  函数：print
##  函数描述：按二维矩阵形式输出矩阵
##  参数描述：
##  Matrix *m：Matrix类型的结构体矩阵
################################################################
*/

void print(Matrix *m) {
    for (int i = 0; i < m->x; i++) {
        for (int j = 0; j < m->y; j++)
            printf("%12.5f ", m->v[i][j]); // 宽度为12，小数点后5位
```

```
        printf("\n");
    }
}
```

**主函数 `main()`**：主要负责的是初始化（内存申请、矩阵输入），调用核函数（同时计时），输出运算时间和结果矩阵等功能。

```
int readInFlie = true, printMatrix = false;

/*
################################################################
##  函数：main
##  函数描述：程序主函数，负责读入矩阵、申请内存、调用核函数、输出计算时间和矩阵等工作
##  参数描述：
##  int argc, char* argv[]：可变输入参数，实际上只接受第一个输入，即输入文件名
################################################################
*/

int main() {
    // 程序初始化
    if (readInFlie) freopen("in.txt", "r", stdin); // 从文件读入 or 从标准输入流读入
    Matrix *a, *b;
    cudaMallocManaged((void**)&a, sizeof(Matrix)); // cudaMallocManaged 这个函数能同时申请
cpu 和 gpu 内存，并自动同步对应值，后续不需要 host to device 操作
    cudaMallocManaged((void**)&b, sizeof(Matrix));
    scanf("%d%d", &a->x, &a->y);
    b->x = a->x; b->y = a->y;
    cudaMallocManaged((void**)&a->v, a->x * sizeof(double*)); // 二维矩阵申请内存
    cudaMallocManaged((void**)&b->v, b->x * sizeof(double*));
    for (int i = 0; i < a->x; i++) {
        cudaMallocManaged((void**)&a->v[i], a->y * sizeof(double));
        cudaMallocManaged((void**)&b->v[i], b->y * sizeof(double));
    }
    for (int i = 0; i < a->x; i++)
        for (int j = 0; j < a->y; j++)
            scanf("%lf", &a->v[i][j]);

    // 调用核函数并计时
    dim3 block(sizex, sizey), grid(a->x / sizex + 1, a->y / sizey + 1);
    timeval t1, t2;
    gettimeofday(&t1, NULL);
    cal <<< grid, block >>> (a, b);
    cudaDeviceSynchronize();
    gettimeofday(&t2, NULL);

    // 输出时间和矩阵
    printf("Time: %.4fs\n", (t2.tv_sec-t1.tv_sec+(t2.tv_usec-t1.tv_usec)/1.0e6));
    if (printMatrix) { // 是否输出矩阵
        printf("Matrix a: \n");
```

```
        print(a);
        printf("\nMatrix b: \n");
        print(b);
    }
    return 0;
}
```

## 存储器类型

**全局内存：**输入矩阵 A 和输出矩阵 B 均使用全局内存。

- 数据访存模式：矩阵 A 需要被所有线程读，矩阵 B 需要被所有线程写。
- 存储器的特性：全局内存空间最大，但访寸时间最长。由于矩阵 A、B 规模均过大，也使得它们只能放在全局内存，其他形式的内存会放不下。

**线程私有内存：**num 数组（用于统计窗口各数值个数）、sum 变量（用于计算窗口大小）、ans 变量（用于计算窗口的熵）使用私有内存。

- 数据访存模式：这些数据只和线程本身计算有关，不需要也不能和其他线程进行数据共享。
- 存储器的特性：线程私有内存访问速度比全局内存快，但仅限线程内部访问。由于访问速度更快，因此不将其放在全局内存。

**共享内存：**本算法目前暂时不需要线程之间共享信息，因此无需共享内存（不过下文打表会用到）。

**常量内存：**目前算法也不需要常量（不过下文打表会用到）。

## 对数查表加速

首先我们对 baseline 程序进行运算时间测试，线程块大小为 8*8，三个输入矩阵大小分别为 1024*1024，4096*4096，512*8192，测试结果如下：（Tesla V100）

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0083s | 0.0082s | 0.0080s | 0.0082s |
| 4096*4096 | 0.0862s | 0.0879s | 0.0855s | 0.0865s |
| 512*8192 | 0.0134s | 0.0128s | 0.0135s | 0.0132s |

由于我们可以修改熵的公式为：

$$H(x) = -\sum_i \frac{num_i}{sum} * \log\frac{num_i}{sum} = -\sum_i \frac{num_i}{sum} * (\log num_i - \log sum)$$

而 $num_i$ 和 $sum$ 的取值为 [0, 25] 和 [9, 25]，此时我们可以简单的进行“打表”以避免复杂的对数计算（由于 $num_i$ 为 0 时 $H(x)$ 为 0，因此可以令 $\log num_i = 0$）。

**而打表过程有多种方式：**

**全局内存：**在 main 函数声明全局变量 `loge`：

```
double* loge;
cudaMallocManaged((void**)&loge, 25 * sizeof(double));
loge[0] = 0.0; loge[1] = 0.0; loge[2] = 0.693147; loge[3] = 1.098612; loge[4] =
1.386294; loge[5] = 1.609437; loge[6] = 1.791759; loge[7] = 1.945910; loge[8] =
2.079441; loge[9] = 2.197224; loge[10] = 2.302585; loge[11] = 2.397895; loge[12] =
2.484906; loge[13] = 2.564949; loge[14] = 2.639057; loge[15] = 2.708050; loge[16] =
2.772588; loge[17] = 2.833213; loge[18] = 2.890371; loge[19] = 2.944438; loge[20] =
2.995732; loge[21] = 3.044522; loge[22] = 3.091042; loge[23] = 3.135494; loge[24] =
3.178053; loge[25] = 3.218875;
cudaDeviceSynchronize();
```

之后将 `loge` 也作为核函数的参数传参进行运算，运算过程改为：

```
for (int i = 0; i < 16; i++) // 4
    if (num[i]) ans -= (double)num[i] / sum * (loge[num[i]] - loge[sum]);
```

测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
|---|---|---|---|---|
| 1024*1024 | 0.0080s | 0.0080s | 0.0080s | 0.0080s |
| 4096*4096 | 0.0847s | 0.0841s | 0.0820s | 0.0836s |
| 512*8192 | 0.0132s | 0.0129s | 0.0126s | 0.0129s |

可以看出全局内存打表之后运算时间比 baseline 快一点点，但快的很不明显。

**线程私有内存：**在核函数声明私有变量 `loge`：

```
double loge[26] = {0.0, 0.0, 0.693147, 1.098612, 1.386294, 1.609437,
                   1.791759, 1.945910, 2.079441, 2.197224, 2.302585,
                   2.397895, 2.484906, 2.564949, 2.639057, 2.708050,
                   2.772588, 2.833213, 2.890371, 2.944438, 2.995732,
                   3.044522, 3.091042, 3.135494, 3.178053, 3.218875};
```

运算过程和全局内存一样，测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
|---|---|---|---|---|
| 1024*1024 | 0.0100s | 0.0094s | 0.0096s | 0.0097s |
| 4096*4096 | 0.0851s | 0.0825s | 0.0829s | 0.0835s |
| 512*8192 | 0.0140s | 0.0142s | 0.0134s | 0.0139s |

可以看出线程私有内存打表之后除了 4096*4096 比 baseline 快一点之外，其它均略慢于 baseline，个人认为是每个线程都花时间去初始化这样一个表（初始化了 26 个数），而原本对数运算最多只需要运算 16 次，因此运算速度变慢了。

**共享内存**：在线程私有内存的基础上加上 `__shared__` 将其声明为共享内存，但是共享内存不支持线程私有内存那样直接初始化，因此正确的声明方法为：

```
__shared__ double loge[26];
loge[0] = 0.0; loge[1] = 0.0; loge[2] = 0.693147; loge[3] = 1.098612; loge[4] =
1.386294; loge[5] = 1.609437; loge[6] = 1.791759; loge[7] = 1.945910; loge[8] =
2.079441; loge[9] = 2.197224; loge[10] = 2.302585; loge[11] = 2.397895; loge[12] =
2.484906; loge[13] = 2.564949; loge[14] = 2.639057; loge[15] = 2.708050; loge[16] =
2.772588; loge[17] = 2.833213; loge[18] = 2.890371; loge[19] = 2.944438; loge[20] =
2.995732; loge[21] = 3.044522; loge[22] = 3.091042; loge[23] = 3.135494; loge[24] =
3.178053; loge[25] = 3.218875;
```

运算过程和全局内存一样，测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
|---|---|---|---|---|
| 1024*1024 | 0.0081s | 0.0084s | 0.0082s | 0.0082s |
| 4096*4096 | 0.0682s | 0.0671s | 0.0677s | 0.0677s |
| 512*8192 | 0.0131s | 0.0137s | 0.0135s | 0.0134s |

可以看出采用共享内存后，4096*4096 比 baseline 快了不少，而其它和 baseline 基本相同，因此共享内存方式更适合大规模数据。

**常量内存**：常量内存的声明比较麻烦，首先需要在函数外部声明 `__constant__` 变量，之后在 main 函数声明 cpu 变量并赋值，最后调用 `cudaMemcpyToSymbol` 函数将 cpu 变量的值拷贝进常量内存中。

```
// main函数外部定义
__constant__ const double loge[26] = {0};

// main函数内部定义
double logeHost[26] = {0.0, 0.0, 0.693147, 1.098612, 1.386294, 1.609437,
                       1.791759, 1.945910, 2.079441, 2.197224, 2.302585,
                       2.397895, 2.484906, 2.564949, 2.639057, 2.708050,
                       2.772588, 2.833213, 2.890371, 2.944438, 2.995732,
                       3.044522, 3.091042, 3.135494, 3.178053, 3.218875};
cudaMemcpyToSymbol(loge, logeHost, sizeof(double) * 26);
```

运算过程和全局内存一样，测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0078s | 0.0082s | 0.0081s | 0.0080s |
| 4096*4096 | 0.0853s | 0.0878s | 0.0890s | 0.0874s |
| 512*8192 | 0.0138s | 0.0141s | 0.0139s | 0.0139s |

可以看出采用常量内存后，运行速度不仅劣于全局变量，而且还不如 baseline 程序。查询得知常量内存的原理为：当处理常量内存时，硬件将主动把这个常量数据缓存在 gpu 上。在第一次从常量内存的某个地址上读取后，当其他半线程束请求同一个地址时，那么将命中缓存，这能够减少了额外的内存流量。虽然当所有线程都读取相同地址时，这个功能可以极大提升性能，但当所有线程分别读取不同的地址时，它实际上会降低性能。因为这些不同的读取操作会被串行化，从而需要更多的时间来发出请求。但如果从全局内存中读取，那么这些请求会同时发出。观察我的代码可以发现，对于同一时间 `loge[num[i]]` 的访存，每个线程的 `num[i]` 大概率是不同的，因此本次实验并不适合采用常量内存。

**对数查表总结：**由上面实验可以总结出，对于较小数据范围，加入打表且采用各种打表方法的差别其实不大，但当数据范围变得比较大时，**加入打表且采用共享内存**的方式会使得程序计算速度变快。

## 优化版本及优化过程

**打表优化：**

由上文可以看出，加入打表且采用共享内存的方式使得矩阵规模为 4096*4096 的计算时间由 0.0865s 优化为 0.0677s，因此第一步加入采用共享内存方式的打表。

**一维矩阵优化：**

由上一次实验可知，将矩阵由二维优化为一维之后，会使得运算速度变快（主要是二维矩阵访存需要地址重定位，速度会更慢），因此修改矩阵结构体和申请内存方式如下：

```
struct Matrix{
    int x, y;
    double* v; // 一维矩阵
};

// main函数
Matrix *a, *b;
cudaMallocManaged((void**)&a, sizeof(Matrix));
cudaMallocManaged((void**)&b, sizeof(Matrix));
cudaMallocManaged((void**)&a->v, a->x * a->y * sizeof(double));
cudaMallocManaged((void**)&b->v, b->x * b->y * sizeof(double));

// 访存代码由 a->v[x][y] 修改为 a->v[x * a->y + y]
```

测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0062s | 0.0064s | 0.0061s | 0.0062s |
| 4096*4096 | 0.0585s | 0.0586s | 0.0588s | 0.0586s |
| 512*8192 | 0.0121s | 0.0121s | 0.0120s | 0.0121s |

4096*4096 结果由 0.0677s 再次优化到 0.0586s。

**矩阵结构体优化：**

由上文结构体结构可知，每一个 Matrix 是由两个 int 和 x * y 个 double 组成，由于 int 的插入会导致 double 部分不能均匀地占用 x * y / cache_line 行，因此考虑将 int 从矩阵结构体中剥离，并通过别的方式进行传参。

而传参也有两种传参方式：全局内存和常量内存。

全局内存：在 main 函数声明全局变量 X 和 Y，由标准输入流或文件输入进行赋值，之后将其作为参数传参给核函数即可。

```
int X, Y;
cudaMallocManaged((void**)&X, sizeof(int));
cudaMallocManaged((void**)&Y, sizeof(int));
cal <<< grid, block >>> (a, b, X, Y);
```

测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0063s | 0.0062s | 0.0063s | 0.0063s |
| 4096*4096 | 0.0575s | 0.0584s | 0.0579s | 0.0579s |
| 512*8192 | 0.0119s | 0.0118s | 0.0119s | 0.0119s |

常量内存：在函数外部声明 `__constant__` 变量，之后在 main 函数声明 cpu 变量并由标准输入流或文件输入进行赋值，最后调用 `cudaMemcpyToSymbol` 函数将 cpu 变量的值拷贝进常量内存中。

```
// main函数外部定义
__constant__ int X, Y;

// main函数内部定义
int x, y;
cudaMemcpyToSymbol(X, &x, sizeof(int));
cudaMemcpyToSymbol(Y, &y, sizeof(int));
```

测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0064s | 0.0061s | 0.0062s | 0.0062s |
| 4096*4096 | 0.0571s | 0.0573s | 0.0574s | 0.0573s |
| 512*8192 | 0.0119s | 0.0121s | 0.0121s | 0.0120s |

4096*4096 结果由 0.0586s 再次优化到 0.0573s，实际上优化的并不是那么明显。

**数据类型优化：**

在上文的代码中，我们使用了大量 int 和 double 数据类型，但是实际上我们可以采用 short int 和 float 对其进行代替，原因如下：

- short int：在原代码中，int 用来表示矩阵的高和宽、循环变量、窗口每个数值计数、窗口总数计数等，但是可以发现它们的取值范围都在 short int 的取值范围内，因此可以用 short int 代替 int。
- float：在原代码中，double 用来表示矩阵 A、B 值的表示、熵的计算过程、log1～log25 的打表等，但是可以发现它们的进度都满足 6 位之内：
    - 熵的最大值（即矩阵元素最混乱：25 个数中 9 个数两两相同，7 个数唯一）约等于 2.72，熵的最小值（即矩阵元素最整齐：25 个数完全一样）等于 0，因此熵的范围在 [0, 2.72] 内，整数 1 位，小数保留 5 位，满足 float 6 位精度；
    - log1～log25 的打表值的范围在 [0, 3.22] 内，和上文同理。

因此可以用位数更低的 short int 和 float 代替 int 和 double，以减少计算时间。

值得特别提出的是，输入矩阵 A 由于题目规定是 [0, 15] 的整型，因此也可以将矩阵 A 改成 short int 类型（即和矩阵 B 类型不一样），继续优化时间。

因此本次修改方式为：将矩阵 A 修改为 short int 类型，矩阵 B 修改为 float 类型，其余的辅助变量（矩阵的高和宽、循环变量、窗口每个数值计数、窗口总数计数、熵的计算过程、log1～log25 的打表等）按照上文方式分别修改为 short int 和 float。

测试结果如下：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0027s | 0.0025s | 0.0029s | 0.0027s |
| 4096*4096 | 0.0310s | 0.0311s | 0.0305s | 0.0309s |
| 512*8192 | 0.0045s | 0.0047s | 0.0050s | 0.0047s |

4096*4096 结果由 0.0573s 再次优化到 0.0309s，得到了明显的优化。

**线程块大小优化：**

线程块（Block）的大小也会影响程序的运行速率，一个基本的原理为：每个块中线程数量应控制为线程束大小（32）的倍数（但是不能大于 1024）。但具体大小的设置更多还是要根据实验得到，因此我们不断调整线程块大小，试图寻找本次实验最优的线程块大小，具体过程和结果如下：

| 矩阵大小 | 线程块大小 | 运行时间1 | 运行时间2 | 平均时间 |
| --- | --- | --- | --- | --- |
| 4096*4096 | 8*8 | 0.0312s | 0.0311s | 0.0311s |
| | 16*16 | 0.0258s | 0.0260s | 0.0259s |
| | 32*32 | 0.0225s | 0.0225s | 0.0225s |
| | 64*16 | 0.0226s | 0.0224s | 0.0225s |
| | 128*8 | 0.0223s | 0.0226s | 0.0224s |
| | 256*4 | 0.0212s | 0.0216s | 0.0214s |
| | 512*2 | 0.0218s | 0.0219s | 0.0218s |

多次测试得出，当线程块大小取 256*4（即 `dim3 block(256, 4)`）时，运行时间最快，此时 4096*4096 结果由 0.0309s 再次优化到 0.0214s。

## 最终的优化版本

经过上述一系列优化后，得到了本次实验最终的优化版本（cuda_final.cu），对于 4096*4096 规模的输入数据运行时间由 0.0865s 优化为 0.0214s，并在此重新介绍：

**程序整体逻辑，包含的函数，每个函数完成的内容：**和 baseline 基本一样，优化实际上并没有改变程序逻辑和函数功能的本质。

**存储器类型：**

- 输入矩阵 A 和输出矩阵 B 使用**全局内存**：全局内存最大，矩阵规模很大，只能放在全局内存里；
- num 数组（用于统计窗口各数值个数）、sum 变量（用于计算窗口大小）、ans 变量（用于计算窗口的熵）使用**私有内存**：这些变量为每个线程计算过程需要的变量，线程之间相互独立，不允许共享；
- 对数查表加速使用**共享内存**：共享内存比全局内存更快，打表数据可以共享，且数据量不大可以放在共享内存里；
- X，Y 变量（矩阵的高和宽）使用**常量内存**：矩阵的高和宽可以作为常量看待，且实验过程中发现常量内存方式运行速度最快。

## 总结影响 cuda 程序性能的因素

- 基于 cuda 架构的因素：
  - 线程块大小：线程块大小需控制为线程束大小（32）的倍数，且不能大于 1024。但实际设定的大小需要根据实验测试进行调参；
  - 存储器类型：数据存放的位置也会影响 cuda 程序的速度。数据量较小且值固定的可以放在常量内存（最好所有线程能在同一代码位置访问同一常量）；值不固定或不能确保同时访问的可以放在共享内存；不能共享的优先放在私有内存，而不考虑在全局内存开辟内存空间；数据量极大的只能放在全局内存。
- 程序本身的因素：
  - 打表优化：一些需要重复计算的相同数据可以提前计算并放在内存中，以空间换取重复的计算时间；

- 一维矩阵优化：二维矩阵存在地址重定位，需要花费更多的计算时间；
- 数据结构优化：结构体存在多种数据类型的话可能会导致内存不对齐，可以尝试用多种数据类型的数组代替结构体数组；
- 数据类型优化：在确保数据范围和数据精度的情况下，可以用位更少的数据类型代替位更多的数据类型。

## OpenMP 程序

OpenMP 的程序可以很容易地由 cuda 程序转换而来，具体步骤为：

1. cuda 的核函数转变为 OpenMP 的一个函数；
2. cuda 的核函数的线程号定位修改为 OpenMP 函数的 x，y 循环变量；
3. OpenMP 函数的循环过程加入 #progma omp parallel for num_threads(12) 进行并行化

OpenMP 关键部分的代码如下：

```
/*
################################################################
##  函数：cal
##  函数描述：openmp并行程序，计算输入矩阵a的熵，计算结果存在矩阵b里
##  参数描述：
##  short *a：输入矩阵a
##  float *b：结果矩阵b
################################################################
*/

void cal(short* a, float* b) {
    #pragma omp parallel for num_threads(12) // openmp 并行
    for (short x = 0; x < X; x++) { // 两重循环代替 cuda 核函数定位
        for (short y = 0; y < Y; y++) {
            short num[16] = {0}, sum = 0; // 其余部分和 cuda 几乎一样
            float ans = 0, loge[26];
            loge[0] = loge[1] = 0.0; loge[2] = 0.693147; loge[3] = 1.098612; loge[4] =
1.386294; loge[5] = 1.609437;
            loge[6] = 1.791759; loge[7] = 1.945910; loge[8] = 2.079441; loge[9] =
2.197224; loge[10] = 2.302585;
            loge[11] = 2.397895; loge[12] = 2.484906; loge[13] = 2.564949; loge[14] =
2.639057; loge[15] = 2.708050;
            loge[16] = 2.772588; loge[17] = 2.833213; loge[18] = 2.890371; loge[19] =
2.944438; loge[20] = 2.995732;
            loge[21] = 3.044522; loge[22] = 3.091042; loge[23] = 3.135494; loge[24] =
3.178053; loge[25] = 3.218875;
            for (short i = max(x - 2, 0); i < min(x + 3, X); i++) {
                for (short j = max(y - 2, 0); j < min(y + 3, Y); j++) {
                    num[a[i * Y + j]]++;
                    sum++;
                }
            }
```

```
            for (short i = 0; i < 16; i++)
                if (num[i]) ans -= (float)num[i] / sum * (loge[num[i]] - loge[sum]);
            b[x * Y + y] = ans;
        }
    }
}
```

具体的代码可以参考 openmp.cpp 文件，测试结果如下（ OpenMP 12 线程）：

| 矩阵大小 | 运行时间1 | 运行时间2 | 运行时间3 | 平均时间 |
| --- | --- | --- | --- | --- |
| 1024*1024 | 0.0330s | 0.0319s | 0.0322s | 0.0324s |
| 4096*4096 | 0.4898s | 0.4984s | 0.4933s | 0.4938s |
| 512*8192 | 0.1232s | 0.1229s | 0.1239s | 0.1233s |

显然 OpenMP 程序要远远慢于 cuda 程序，因为 OpenMP 程序基于 cpu 线程进行运算，而 cpu 线程数量有限且远远小于 gpu 线程数，因此速度也要慢上不少。

## 参考资料：

CUDA中共享内存、常量内存和纹理内存的概念和应用（小白入门）

## 附录：

**程序正确性：**可以将程序切换为手动输入矩阵、输出结果矩阵，测试过程如下：

```
jovyan@jupyter-songyj9-40mail2-2esysu-2eedu-2ecn:~/multi/lab2$ ./cuda_final
4 4
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
Time: 0.0002s
Matrix a:
     1.00000     2.00000     3.00000     4.00000
     2.00000     3.00000     4.00000     5.00000
     3.00000     4.00000     5.00000     6.00000
     4.00000     5.00000     6.00000     7.00000

Matrix b:
     1.52295     1.70455     1.70455     1.52295
     1.70455     1.84075     1.84075     1.70455
     1.70455     1.84075     1.84075     1.70455
     1.52295     1.70455     1.70455     1.52295
```

对输出矩阵 B 第一行第二列元素分析：窗口大小为 12，窗口包含 1 个 1，2 个 2， 3 个 3， 3 个 4，2 个 5， 1 个 6，熵的计算过程为：

$$H(x) = -\frac{1}{12} * \log\frac{1}{12} * 2 - \frac{2}{12} * \log\frac{2}{12} * 2 - \frac{3}{12} * \log\frac{3}{12} * 2 \approx 1.70455$$

其余位置的元素同理，可以看出程序对熵的计算是无误的。

**大型输入矩阵的生成：**可以简单地通过代码生成一个大型输入矩阵文件，下文以 python 为例：

```
import random

x, y = 4096, 4096
f = open('in.txt', 'w')
f.write(str(x) + ' ' + str(y) + '\n')
for i in range(x):
    for j in range(y):
        f.write(str(random.randint(0, 15)) + " ")
    f.write('\n')
```

**批处理文件：**由于实验过程中包含大量的测试工作，因此可以采用批处理命令一次性进行多次测试。

Linux 下执行 cuda 程序的批处理文件（test.sh）执行命令为 `sh test.sh cuda_final`，文件代码如下：

```
nvcc -w $1.cu -o $1
./$1 in1.txt
./$1 in1.txt
./$1 in1.txt
./$1 in2.txt
./$1 in2.txt
./$1 in2.txt
./$1 in3.txt
./$1 in3.txt
./$1 in3.txt
```

Windows 下执行 OpenMP 程序的批处理文件（test.bat）执行命令为 `test.bat`（或直接双击打开 test.bat 文件），文件代码如下：

```
g++ openmp.cpp -o openmp -fopenmp
openmp in1.txt
openmp in1.txt
openmp in1.txt
openmp in2.txt
openmp in2.txt
openmp in2.txt
openmp in3.txt
openmp in3.txt
openmp in3.txt
pause
```