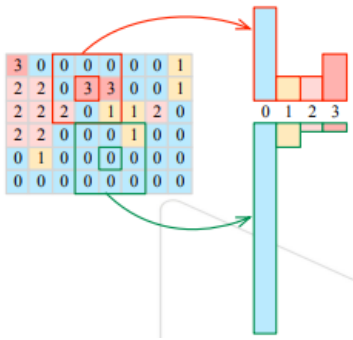
多核设计HW2

葉珺明 19335253

利用CUDA计算二维数组中以每个元素为中心的熵

- 计算二维数组中以每个元素为中心的熵（entropy）
  - 输入：二维数组及其大小
    - 假设元素为[0,15]的整型
  - 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



$$H(X) = -\frac{4}{9}\log\frac{4}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{2}{9}\log\frac{2}{9}$$

$$H(X) = -\frac{8}{9}\log\frac{8}{9} - \frac{1}{9}\log\frac{1}{9}$$

3

1 程序整体逻辑

程序设计分为两部分，核函数和主函数

1.1 核函数部分

- 核函数：按实现的不同分为全局内存、全局内存+log表查询和共享内存的方式三种
  - 线程块和每个线程的分配任务

```
dim3 block(BDIM, BDIM, 1);
dim3 grid(divup(width, BDIM), divup(height, BDIM), 1);
```

每个线程块的维度为$(8，8，1)$：对于8×8的矩阵，只需要1个$Block$；对于2048×2048的矩阵，需要256×256个$Block$

每个线程计算一个中心位置对应窗口的熵

  - 全局内存的实现：

将需要进行的数据存入全局内存中，每个线程都索引全局内存中得数据进行熵的计算；

对于边界窗口越界的情况，需要判断$[row \pm r, col \pm r]$是否在$[0, height)$和$[0, width)$之间；

开辟长度为16的数组，因为矩阵元素是$[0, 15]$的整型，只需要记录窗口内各元素的个数；

```
/*****************************/
//    "global version":
__global__ void global_cal_entropy(int *in, float *out, int width, int height){
    int tid_x = blockIdx.x * blockDim.x + threadIdx.x;
    int tid_y = blockIdx.y * blockDim.y + threadIdx.y;
    int all = 0;
    int cnt[16] = {0};

    for(int i=-r; i<=r ; i++){
        for(int j=-r; j<=r; j++){
            if(tid_y+j>=0 && tid_y+j<height && tid_x+i>=0 && tid_x+i<width){
                all++;
                cnt[in[(tid_y+j)*width+tid_x+i]] += 1;
            }
        }
    }
    float t = 0.0f;
    for(int k=0;k<16;k++){
        if(cnt[k]==0)
            continue;
        t += -(float)cnt[k]/(float)all*(float)log2((float)cnt[k]/all);
    }
    out[tid_y*width+tid_x] = t;
}
```

- 全局内存+查表方式：

  与全局内存方式相似，增加log表的传递，将log的数值表存入全局内存中：

```
for(int k=0;k<16;k++){
    if(cnt[k]==0)
    continue;
    t += -(float)cnt[k]/(float)all*(log_d[cnt[k]]-log_d[all]);
}
```

- 共享内存方式（优化版本）：

  开辟共享内存的空间存入一个$Block$中所有线程涉及的数据，访问共享内存的速度比访问全局内存的速度要快很多，能够大大减少数据访问的耗时，提高程序速度。

  共享内存的大小是由线程块和窗口的大小决定的，对于边界窗口越界的部分，用-1元素填充；在第一个线程实现所有-1元素的填充和其本身的数据写入共享内存；

  需要调用__syncthreads();实现线程间的同步；

  窗口内熵的计算与全局内存方式相似，换成共享内存的访问。

```
/*****************************/
//    "share mem version":
__global__ void share_cal_entropy(int *in, float *out, int width, int height){
    __shared__ int smem[BDIM+2*r][BDIM+2*r];
    int tid_x = blockIdx.x * blockDim.x + threadIdx.x;
    int tid_y = blockIdx.y * blockDim.y + threadIdx.y;

    int sid_x = threadIdx.x + r;
```

```
    int sid_y = threadIdx.y + r;

    smem[sid_y][sid_x] = in[tid_y*width+tid_x];
    if(tid_x==0&&tid_y==0){
        for(int i=0; i<BDIM+r*2; i++){
            for(int j=0; j<BDIM+r*2; j++){
                if(i<2||i>BDIM+r-1){
                    smem[i][j] = -1;
                }
                if(j<2||j>BDIM+r-1){
                    smem[i][j] = -1;
                }
            }
        }
    }
    __syncthreads();

    int all = 0;
    int cnt[16] = {0};
    for (int i=-r; i<=r; i++){
        for (int j=-r; j<=r; j++){
            if (smem[sid_y+j][sid_x+i] != -1){
                all++;
                cnt[smem[sid_y+j][sid_x+i]] += 1;
            }
        }
    }

    float t = 0.0f;
    for(int k=0;k<16;k++){
        if(cnt[k]==0)
            continue;
        t += -(float)cnt[k]/(float)all*(float)log2((float)cnt[k]/all);
    }
    out[tid_y*width+tid_x] = t;

}
```

1.2 主函数部分

- 主函数：分别为对三个核函数各个调用
  - 主机端数据写入设备端：

```
int *in;
float *out;
CHECK(cudaMalloc((void **)&in, sizeof(float)*size));
CHECK(cudaMemcpy(in, input, sizeof(float)*size, cudaMemcpyHostToDevice));
CHECK(cudaMalloc((void **)&out, sizeof(float)*size));
CHECK(cudaMemcpy(out, result, sizeof(float)*size,
cudaMemcpyHostToDevice));
```

  - *Grid*和*Block*的开辟：

```
dim3 block(BDIM, BDIM, 1);
dim3 grid(divup(width, BDIM), divup(height, BDIM), 1);
```

- 全局内存方式的调用：

```
auto sta = getTime();
global_cal_entropy << < grid, block >> > (in, out, width, height);
auto end = getTime();
auto time = end - sta;
printf("GLOBAL USED TIME: %ld \n", time);

CHECK(cudaMemcpy(result, out, sizeof(float)*size,
cudaMemcpyDeviceToHost));
```

- 全局内存+查表方式的调用：

  需要在主机端完成*log*表的计算，并将该表存入设备端

```
 int len = 26;
 float *log_device;
 float *log_host = (float*)malloc(sizeof(float)*(len));
 log_host[0] = 0;

 for(int i=1; i<len; i++){
     log_host[i] = log2(i);
 }
 CHECK(cudaMalloc((void**)&log_device, sizeof(float)*len));
 CHECK(cudaMemcpy(log_device, log_host, sizeof(float)*len,
 cudaMemcpyHostToDevice));

 auto log_sta = getTime();
 log_cal_entropy << < grid, block >> > (in, out, width, height,
 log_device);
 auto log_end = getTime();
 auto log_time = log_end - log_sta;
 printf("GLOBAL+Log_table USED TIME: %ld \n", log_time);

 CHECK(cudaMemcpy(result, out, sizeof(float)*size,
 cudaMemcpyDeviceToHost));

```

- 共享内存方式的调用：

```
auto share_sta = getTime();
share_cal_entropy << < grid, block >> > (in, out, width, height);
auto share_end = getTime();
auto share_time = share_end - share_sta;
printf("SHARE USED TIME: %ld \n", share_time);

CHECK(cudaMemcpy(result, out, sizeof(float)*size,
cudaMemcpyDeviceToHost));
```

2 涉及的存储器

2.1 全局内存

- 在全局内存的方式中，整个输入和输出矩阵是存储在全局内存中，每个线程再对全局内存进行访问；
- 在全局内存+log表方式中，整个输入和输出矩阵和log表都是存储在全局内存中的
- 在共享内存的方式中，整个输出矩阵是存储在全局内存中

全局内存的特点的是容量大，适合大批量的数据存储，但访问速度慢。

2.2 共享内存

- 只有共享内存的方式涉及到共享内存，每个线程块需要用到的数据都存储在共享内存中

共享内存的特点是其容量小，访问速度快，能够大大减少数据访问耗时，对于一个$Block$中的共同数据部分，可以将数据存储在共享内存中，提高程序速度。

2.3 常量内存

整个程序设计没有涉及到常量内存，在设计的过程中，一般将常量存放在全局内存中了。

3 只涉及对整数$[1,25]$的对数运算

$$H(X)=-\sum p_i*\log(p_i)$$
$$p_i=p(X=x_i)$$

- 有公式：

$$\log\frac{M}{N}=\log M-\log N$$

窗口大小为5×5，那么对于有效元素最多为25个，，那么M，N的取值为$[1,25]$。

- 通过查表能够加速运算过程：

实验结果比较：

```
(base) jiayulong@lm04:~/lijinmin/yep/multiCore/homework2/code$ ./main input1.bin output1.bin
GLOBAL USED TIME: 28792
GLOBAL+Log_table USED TIME: 14090
SHARE USED TIME: 15276
(base) jiayulong@lm04:~/lijinmin/yep/multiCore/homework2/code$ ./main input2.bin output2.bin
GLOBAL USED TIME: 37430
GLOBAL+Log_table USED TIME: 14324
SHARE USED TIME: 16999
```

比较可以发现，不论是8×8的矩阵还是2048×2048的矩阵，全局+查表的方式都比全局的方式要快，提升了50%左右，因为通过查表可以减少CUDA计算log的时间，线程会重复构建对数运算，通过查表能够减少了这一重复繁琐的过程。

4 基础版本与优化版本

- 基础版本

  基础版本为全局方式，通过上图的比较可以观察发现全局方式花费的时间是最长的

- 优化版本
  - 全局方式与查表相结合

    全局方式与查表结合能够在全局方式的基础上减少CUDA对对数的构建运算，比起全局方式要快，程序运行速度提升了50%左右。

  - 共享内存的方式

    共享内存的方式将线程块的所需数据先存储到共享内存中，比起访问全局内存的方式，访问共享内存的方式速度要快很多，相比全局内存的方式提升了40%左右，但是比全局内存与查表相结合的方式要慢。

5 影响性能的因素

- $Grid$和$Block$的分配，对于不同的矩阵大小，需要进行的线程块分配是不一样的，对于需要较大线程块时，应尽量实现数据并行，创建更多并发的活跃线程束，找到合适的数据分配，使得线程负载的计算量不会过大或过小，即需要有效提高占用率。
- GPU适合数据密集型的计算，应当减少控制复杂的逻辑操作，如for循环或if判断操作会降低内核的执行效率。
- 对共享内存的使用，从全局内存方式和共享内存的方式实验结果比较可以看出，共享内存的访问速度是比全局内存的访问速度要高很多的，对于不同的变量，需要将其以合适的方式存储。