多核实验1

个人信息

姓名	廖浩淳	学号	17341096

> 本次作业不同版本使用`git`分支区分，详情见附录

回答问题

1. 介绍程序整体逻辑，包含的函数，每个函数完成的内容。对于核函数，应该说明每个线程块及每个线程所分配的任务

   程序由`mian.cu`按顺序读入输入样例，调用`cudaCallback`来准备cuda内存，并在`kernal`中具体实现计算功能。具体如下：

   - `main.cu`程序的入口，按序读入样例，处理并写入结果
   - `cudaCallback`准备cuda内存，并调用`kernal`函数
   - `kernal`具体计算熵的函数
     - 以最终版本为例，每个线程块包含25个线程，共同负责一个点的结果；每个线程负责这个点的25个可能合法的位置的求值

2. 解释程序中涉及哪些类型的存储器(如，全局内存，共享内存，等)，并通过分析数据的访存模式及该存储器的特性说明为何使用该种存储器。

   - 全局内存
     - logRes是一个长度为26的float类型数组，我用它来存储`n*log(n)`的值常量，用于加快`kernal`中计算log（其实效果微乎其微，因此在最终版本没有使用==）

       logRes在`kernal`中是一个`只读`数组，所有block中的所有thread都不会对其进行写入。并且全局内存可以被任何`kernal`中的线程访问到相同的数据，因此很适合把logRes数组放在全局内存，减少`logf`函数的调用，加快计算速度
   - 共享内存
     - counts是一个长度为16的int数组，我用它来存储`n`在25个可能合法的位置中的个数

       每个block中有25个线程，读到input中相应位置的值作为下表去给counts相应位置加一，大家是对同一个数组内存区域进行写入操作，而共享内存对同一个block内的所有线程是共享可见的，因此很适合把counts放在共享内存
     - logCounts是一个长度为16的float数组，我用它来存储每个值的个数相应的log值

logCounts与counts部分功能基本一致，最后会执行二分求和，也是需要在block内共享的内存，因此使用共享内存进行存储

3. 程序中的对数运算实际只涉及对整数[1,25]的对数运算，为什么?如使用查表 对log1~log25进行查表，是否能加速运算过程?请通过实验收集运行时间，并 验证说明。

   是的，因为求熵的公示可以进行化简，以有25个合法元素的点为例，初始公式大致为

$$H(x) = \sum -\frac{count(n)}{25} \log \frac{count(n)}{25}$$

   其中`count(n)`表示为各值的个数

   公式可以化简为

$$H(x) = \log 25 - \frac{1}{25} \sum count(n) * \log count(n)$$

   可以看到，其中只涉及了对[1~25]的log值的计算，因此理论上可以通过对log1~log25进行查表进行加速。但是实验结果表现与理论还是有差距，加速效果并不明显，猜测是`logf`这个函数本身就不太耗时。（具体实验过程见后文）

4. 请给出一个基础版本(baseline)及至少一个优化版本。并分析说明每种优化对性能的影响。

   > 重要声明：我对每一个版本的优化改造都在以`git`分支的形式呈现，比如基础串行版本可以通过调用`git checkout feat-1-serial`获得，分支介绍详情见附录

   1. 基础串行版本`git checkout feat-1-serial`

```
__global__ void kernel(int width, int height, float *input, float
*output) {
    for(int i = 0; i < width; i++){
        for(int j = 0; j < height; j++){

            int sum = 0;
            int counts[NUMLEN] = {0};
            for(int m = i-2; m <= i+2; m++){
                for(int n = j-2; n <= j+2; n++){
                    if(0 <= m && m < height && 0 <= n && n < width){
                        int num = input[m*width+n];
                        counts[num]++;
                        sum++;
                    }
                }
            }
```

```
            double res = logf(sum);
            for(int m = 0; m < NUMLEN; m++){
                int count = counts[m];
                if(count != 0){
                    res -= count * logf(count) / sum;
                }
            }

            output[i*width+j] = res;
        }
    }
}
```

串行版本的`kernal`函数复杂度是惊人的`O(n^4)`(不严谨地认为)，根本都没法跑完全部的测试样例，采用了<<< 1, 1 >>>的线程方案，一个block一个线程，负责size个点的所有任务，遍历每一个点，再扫描25个位置后计算结果。

性能：这个方案使用的时间可以近似认为是无穷大，因为会超时导致程序挂掉，但是穿行版本可以用来校验程序的正确性，因此被我保留了下来。

2. 初始并行版本`git checkout feat-2-parallel`

```
__global__ void kernel(int width, int height, float *input, float
*output) {
    int i = blockIdx.x / width;
    int j = blockIdx.x % width;

    int sum = 0;
    int counts[NUMLEN] = {0};
    for(int m = i-2; m <= i+2; m++){
        for(int n = j-2; n <= j+2; n++){
            if(0 <= m && m < height && 0 <= n && n < width){
                int num = input[m*width+n];
                counts[num]++;
                sum++;

            }
        }
    }

    double res = logf(sum);
    for(int m = 0; m < NUMLEN; m++){
        int count = counts[m];
        if(count != 0){
            res -= count * logf(count) / sum;
        }
    }

    output[i*width+j] = res;
```

```
}
```

这个版本是比较低级的并行版本，没有使用共享内存，分配了size个blocks，每个block一个线程，这个线程负责相应位置的25个可能合法的元素的求值，他的复杂度是`O(n^2)`(不严谨地认为)。

性能：对于最后一个测试用例，他需要的时间是184ms

```
sample:
No. 11, width: 2000, height: 3000, size: 6000000
 6.00000  4.00000 14.00000 14.00000  9.00000
 7.00000 13.00000 10.00000 10.00000  8.00000
13.00000  1.00000  8.00000  2.00000 12.00000
 6.00000  4.00000  8.00000  1.00000  9.00000
12.00000 12.00000  8.00000  5.00000 12.00000
result:
 2.52519  2.51159  2.54612  2.36273  2.11865
 2.51159  2.28979  2.40069  2.27642  1.99135
 2.43521  2.45614  2.38710  2.20711  1.96869
 2.25026  2.27642  2.23327  2.04674  1.79176
 1.95646  2.08377  2.02623  1.82008  1.67699
cudaCallback: 184.423 ms
```

3. 并行优化版本 增加log常量数组 `git checkout feat-3-parallel`

```
// core.cu

float tmp[26] = {0.0};
for (int i = 1; i <= 25; i++){
    tmp[i] = i * logf(i);
}
cudaMemcpyToSymbol(logRes, tmp, 26 * sizeof(float));
```

在`core.cu`内部为全局内存`lorRes`进行赋值，存储的值是相应下表的`i * logf(i)`，使用全局内存的方式来代替调用`logf`函数

```
sample:
No. 11, width: 2000, height: 3000, size: 6000000
 6.00000  4.00000 14.00000 14.00000  9.00000
 7.00000 13.00000 10.00000 10.00000  8.00000
13.00000  1.00000  8.00000  2.00000 12.00000
 6.00000  4.00000  8.00000  1.00000  9.00000
12.00000 12.00000  8.00000  5.00000 12.00000
result:
 3.21888  3.21888  3.21888  2.99573  2.70805
 3.21888  3.21888  3.21888  2.99573  2.70805
 3.21888  3.21888  3.21888  2.99573  2.70805
 2.99573  2.99573  2.99573  2.77259  2.48491
 2.70805  2.70805  2.70805  2.48491  2.19722
cudaCallback: 182.872 ms
```

性能：比前一个并行版本运行时间少了2ms，几乎没有优化效果

4. 最终版本 `git checkout feat-4-atomic`

   采用<<< size, 25 >>>的方案调用`kernal`函数，每个位置一个block，每个block25个线程，每个线程都负责其中一个可能的合法位置的元素。

   使用一个长度为16的共享内存，用户计算每个数字在合法区域内的个数，这里面使用到了原子操作`atomicAdd(&counts[num], 1);`，它可以让多个线程访问共享内存的时候进行互斥访问，保证结果的正确性。

   `__syncthreads();`同步语句用于block内进程的同步，保障逻辑的正确性。

   求和采用了二分求和的方法，可以将整个函数的复杂度控制在O(log(n))（不严谨地认为）

```
__global__ void kernel(int width, int height, float *input, float
*output) {
    int i = blockIdx.x / width;
    int j = blockIdx.x % width;
    int tx = threadIdx.x;

    // 初始化共享变量 计算元素个数
    __shared__ int counts[NUMLEN];
    if(tx < NUMLEN){
        counts[tx] = 0;
    }
    __syncthreads();

    // 利用原子操作计算元素个数
    int m = i - 2 + tx / width;
    int n = j - 2 + tx % width;
    if(0 <= m && m < height && 0 <= n && n < width){
        int num = input[m*width+n];
        atomicAdd(&counts[num], 1);
    }
```

```
    __syncthreads();

    // 计算合法区域大小
    int up = i >= 2 ? 2 : i;
    int down = height-1-i >= 2 ? 2 : height-1-i;
    int left = j >= 2 ? 2 : j;
    int right = width-1-j >= 2 ? 2 : width-1-j;
    int sum = (up + 1 + down) * (left + 1 + right);

    // 计算相应log值
    __shared__ float logCounts[NUMLEN];
    if(tx < NUMLEN){
        int count = counts[tx];
        if(count != 0){
            logCounts[tx] = count * logf(count) / sum;
        }else{
            logCounts[tx] = 0;
        }
    }
    __syncthreads();

    // 求和
    int t = tx;
    int k = NUMLEN / 2;
    while (k != 0) {
        if(t < k){
            logCounts[t] += logCounts[t + k];
        }
        __syncthreads();
        k /= 2;
    }

    // 写入结果
    if (tx == 0){
        output[i*width+j] = logf(sum) - logCounts[0];
    }
}
```

性能：最后一个测试用例，宽2000，长3000，总大小为6000000的测试用例，总共花费时间45ms，是个不错的效果，实验结束。

```
sample:
No. 11, width: 2000, height: 3000, size: 6000000
 6.00000  4.00000 14.00000 14.00000  9.00000
 7.00000 13.00000 10.00000 10.00000  8.00000
13.00000  1.00000  8.00000  2.00000 12.00000
 6.00000  4.00000  8.00000  1.00000  9.00000
12.00000 12.00000  8.00000  5.00000 12.00000
result:
 3.16342  3.16342  3.16342  2.92642  2.61563
 3.16342  3.16342  3.16342  2.92642  2.61563
 3.16342  3.16342  3.16342  2.92642  2.61563
 2.92642  2.92642  2.92642  2.68595  2.36938
 2.52321  2.61563  2.70805  2.48491  2.19722
cudaCallback: 45.463 ms
```

5. 对实验结果进行分析，从中归纳总结影响CUDA程序性能的因素

    1. 线程数和每个线程的任务

        这个是最重要的影响因素，一般来说，线程数越多，每个线程的工作就越少，整体的性能就越好。

        实验从前往后的几个版本，串行版本只有一个线程，负责所有点的所有25个元素的计算求和，肯定是效率很低下的；第一个并行版本每个位置有一个线程，负责25个元素的计算求和，比起串行版本有了不错的效率提升；最终版本每个位置有25个线程，每个线程只需要负责当前点的其中一个可疑元素的计算，工作量大大降低，效率提高很多。

    2. 全局内存

        全局内存并不能显著提高CUDA程序的性能，猜测原因是`logf`这个函数的执行效率并不算太低。

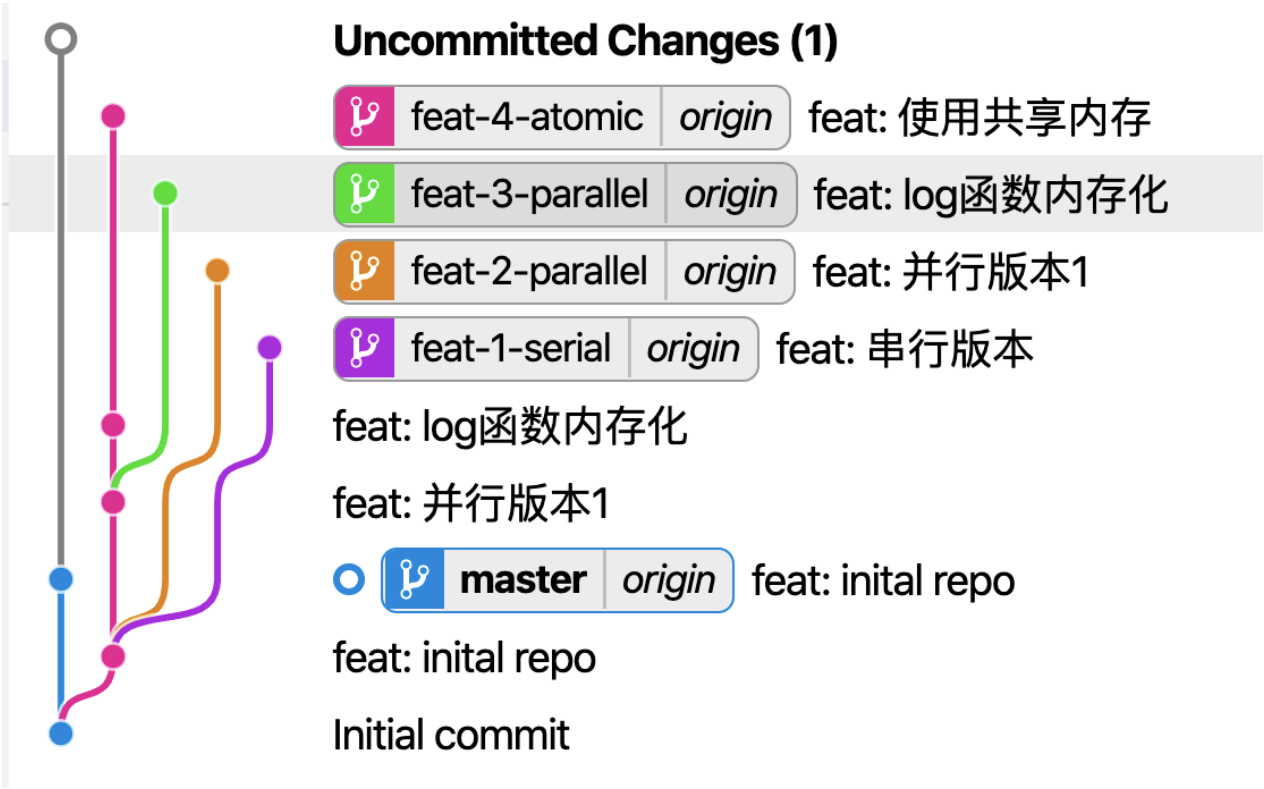
附录

git 分支简介

分支介绍

GitHub地址：https://github.com/liaohch3/sysu_multicore

```
|- master              // 最初的代码，无业务逻辑
  |- feat-1-serial     // 穿行版本分支
  |- feat-2-parral     // 并行版本分支，未使用共享内存
  |- feat-3-parral     // 并行版本分支，使用全局内存存储log值
    |- feat-4-atomic   // 并行版本分支，最终版本，使用共享内存
```


Uncommitted Changes (1)
feat-4-atomic origin feat: 使用共享内存
feat-3-parallel origin feat: log函数内存化
feat-2-parallel origin feat: 并行版本1
feat-1-serial origin feat: 串行版本
feat: log函数内存化
feat: 并行版本1
master origin feat: inital repo
feat: inital repo
Initial commit