CUDA作业1-计算二维数组子矩阵的信息熵

17341146 王程钥

1 简介

本次作业的任务如下

> 给定一个二维数组，计算以其中每个元素为中心的熵的和。

$H(X) = -\sum_i p_i * log(p_i)$
本次实验中我分别使用CUDA和OpenMP进行实现，并使用了多种方法进行优化。最终处理一个2560*2560 的矩阵耗时可以达到 23.649s。

2 方法与优化

本次实验中我首先实现了一个比较简单的baseline，然后在算法层面进行优化，后又在存储器层面进行优化，得到最终版本的算法。这里展示优化方法，优化效果见第四章。

2.1 baseline

baseline的做法比较简单。枚举每个位置，计算每个位置对应的5*5矩阵所在存储位置（如果在边界，相应减小矩阵大小）。后在[0,15]范围内枚举每个数字，对于每个数字枚举矩阵中每个位置计算出现次数。最后代入信息熵公式完成计算。时间复杂度 O(15*5*5)，简要代码如下。

```
float ans = 0, prob;
int Cnt;
for(int k=0;k<16;k++) {
    Cnt = 0;
    for(int i=L;i<R;i++)
        for(int j=D;j<U;j++) {
            if(input[i * height + j]==k) Cnt++;
        }
    if(Cnt) {
        prob = 1.0f * Cnt / all_idxs;
        ans -= prob * logf(prob);
    }
}
output[idx] = ans;
```

2.2 算法优化

2.2.1 空间换时间

这里采用了空间换时间的思想，对于每个线程开一个大小为15的临时数组用于统计每个数字的出现次数。时间复杂度可优化到 $O(5 * 5 + 15)$ 。

2.3.2 对log值查表

这是作业文件中提到的一个方法，下面验证一下此法的正确性。
程序中一个矩阵的大小可能是33，34，35，44，45，55，也就是说一种数字出现的概率的分母只可能是以上这几个数字中的一个。概率的分子肯定也在0到以上若干数字中的某个之间。因此概率的分子和分母都在[0, 25]的范围内。根据以下公式，
$log(p) = log(\frac{x}{y}) = log(x) - log(y)$
需要使用的log值只和1~25这25个数字相关。因此预处理这25个log值，后进行查表即可。

经过两种算法优化后的简要代码如下。

```
char Cnt[16] = {0};
for(int i=L;i<R;i++)
    for(int j=D;j<U;j++) {
        Cnt[(int)input[i * height + j]]++;
    }
for(int i=0;i<16;i++)
    if(Cnt[i]) {
        prob = 1.0f * Cnt[i] / all_idxs;
        ans -= prob * logf(prob);
    }
output[idx] = ans;
```

2.3 存储器优化

sample代码中所有CUDA相关的数组全部采用动态全局内存进行存储，形参和局部变量则存储在寄存器中。这里我尝试了两种优化。一是尝试使用预先开好的静态全局内存，二是尝试使用共享内存进行存储。

2.3.1 静态全局内存

每次请求动态内存空间是一笔很大的开销，如果能够预知大致的内存需求量，直接开静态全局内存，就可以省去这部分时间开销了。
根据`data.bin`的数据规模，这里预开了两个大小为 700,0010 的数组 input_g，output_g 用于CPU向GPU的数据传输。

```
#define N 7000010
__device__ float input_g[N], output_g[N];
```

CPU向GPU传数据。

```
CHECK(cudaMemcpyToSymbol(input_g, sample, sizeof(float)*size));
```

GPU向CPU回传数据。

```
CHECK(cudaMemcpyFromSymbol(*result, output_g, sizeof(float)*size));
```

不过由于预开内存存在局限性，对程序输入缺乏灵活性，对于输入输出矩阵数据，这种方法最终没有被使用。但是对于预处理log函数的部分，因为空间大小开销确定，所以采用这种方法进行存储。

2.3.2 共享内存

如果使用全局内存，每次对矩阵上的一个点进行计算需要访问其周围5*5大小的矩阵的内存，总访问复杂度就是O(5*5*N*M)，而如果先将部分数据放入共享内存，则理论访存的复杂度可以优化到O(N*M)级别。
共享内存的具体使用方法如下。首先对于每个线程开设一个共享内存空间，对应的sid就是线程编号。

对于每个线程计算出每个点对应和行数[L, R]，对于每一行进行同步。对于每个位置（x, y），将它的值放入共享的sid位置中。
在调用的时候，先判断一下这个位置附近几个需要被调用的位置是否完全放入共享内存。如果放入则调用共享内存计算，否则使用全局内存计算。
如何判断？首先，如果位置在矩阵的边缘（即5*5的矩阵不完全存在），则使用全局内存计算。其次，如果线程编号<2或线程编号>=Threadnum-2，则其相邻位置肯定没有全部被放入共享内存中，也使用全局内存计算。相关代码如下。

```
for(int i=L;i<R;i++){
    Cur[sid] = input[i * height + y];
    __syncthreads();
    // load share memory
    if(y<2 || y>=height-2 || sid<2 || sid+2>=ThreadNum) {
        for(int j=D;j<U;j++) {
            Cnt[(int)input[i * height + j]]++;
        }
        // if incomplete in share memory
    }
    else {
        for(int j=-2;j<=2;j++) {
            Cnt[(int)Cur[sid + j]]++;
        }
        // if all in share memory
    }
}
```

3 代码实现

这里只展示经过优化后最终版本的代码，因此展示的代码和提交的代码略有不同。出于直观等因素，版本相关的判断语句会被删除，函数名的版本编号也会被删掉。同时，因为本次实验中我只需要编写core.cu和core.h两个文件，所以也就只展示这两个文件的相关代码。
首先是预处理log的部分，这里定义静态全局数组。

```
__device__ float logs[26];
```

`prepare_log`函数，预处理1-25的log值。该函数会在先在执行`cudaCallback`前先被`main.cu`调用。

```
void prepare_log() {
    float _logs[26];
    for(int i=1;i<=25;i++) _logs[i] = logf(i);
    CHECK(cudaMemcpyToSymbol(logs, _logs, sizeof(float)*26));
}
```

`cudaCallback`函数，从该函数调用CUDA核函数。该函数负责内存拷贝等功能。
这里采用动态内存的方法，在执行CUDA核函数前先在GPU上开好input和output数组，后将输入数据导入input数组执行。调用核函数后从output数组中去除结果放入result数组，并删除input和output数组的空间。
这里参考样例代码，将所有输入数据分为1024个线程，每个线程负责在内存中连续的$\frac{size}{1024}$个位置的信息熵计算。

```
void cudaCallback(int width, int height, int Flag, float *sample, float
**result) {
    int size = width * height;
```

```
    float *input_d, *output_d;

    CHECK(cudaMalloc((void **)&input_d, sizeof(float)*size));
    CHECK(cudaMalloc((void **)&output_d, sizeof(float)*size));
    CHECK(cudaMemcpy(input_d, sample, sizeof(float)*size,
cudaMemcpyHostToDevice));
    // Allocate device memory and copy data from host to device

    // Invoke the device function
    int ThreadNum = divup(size, 1024);
    kernel<<< ThreadNum, 1024 >>>(width, height, input_d, output_d,
ThreadNum);
    cudaDeviceSynchronize();

    // Copy back the results and de-allocate the device memory
    *result = (float *)malloc(sizeof(float)*size);
    CHECK(cudaMemcpy(*result, output_d, sizeof(float)*size,
cudaMemcpyDeviceToHost));
    CHECK(cudaFree(input_d));
    CHECK(cudaFree(output_d));
}

```

`kernel`函数，CUDA的核函数。作为一个CUDA线程的主调函数，执行多核并行相关逻辑。具体逻辑如下：

- 1. 将一维坐标idx转换为二维坐标（x, y）
- 2. 算出以点（x, y）为中心的5*5矩阵的范围 L, R, U, D。
- 3. 根据2.3.2的方法对共享内存进行存储
- 4. 枚举5*5矩阵内的每个点，获取每个点的数值。如果数据完整存在共享内存里则从共享内存获取，否则从全局内存获取。
- 5. 采用空间换时间的方法，记录每个数字的出现次数
- 6. 枚举[0,15]中每个数字，计算信息熵，得到结果。

```
__global__ void kernel(int width, int height, float *input, float *output,
int ThreadNum) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int x = idx / height, y = idx % height;
    int L = max(0, x-2), R = min(x+3, width);
    int D = max(0, y-2), U = min(y+3, height);
    // get x/y/L/R/U/D
    int all_idxs = (R-L) * (U-D);
    float ans = 0, prob;

    extern __shared__ float Cur[6500];
    int sid = threadIdx.x;

    char Cnt[16] = {0};
    for(int i=L;i<R;i++){
        Cur[sid] = input[i * height + y];
        __syncthreads();
        // load share memory
        if(y<2 || y>=height-2 || sid<2 || sid+2>=ThreadNum) {
            for(int j=D;j<U;j++) {
                Cnt[(int)input[i * height + j]]++;
            }

```

```
            // if incomplete in share memory
        }
        else {
            for(int j=-2;j<=2;j++) {
                Cnt[(int)Cur[sid + j]]++;
            }
            // if all in share memory
        }
    }
    float log_base = logs[all_idxs];
    for(int i=0;i<16;i++)
        if(Cnt[i]) {
            prob = 1.0f * Cnt[i] / all_idxs;
            ans -= prob * (logs[Cnt[i]] - log_base);
        }
    // calc entropy
    output[idx] = ans;
}
```

4 实验结果与分析

4.1 测试数据与评价标准

测试数据采用sample里提供的`data.bin`，其中共包含12组数据，最大的矩阵大小为2560*2560。采用如下两个评价标准，一是所有数据的总运行时间，二是所有数据中运行时间最久的一组的运行时间。因为运行时间并不稳定，这里的测试结果仅作参考。另外，因为测试时第一块数据的运行时间包括各种初始化、等待的时间，不太稳定且较长，鉴于第一块大小只有5*5，我决定在记录总时间时将其忽略。

4.2 算法优化

以下为测试结果。TimeMemo表示空间换时间优化，PreLog表示预处理log的优化。

方法	总时间	最大时间
baseline	83.166 ms	25.353 ms
TimeMemo	80.337 ms	24.399 ms
TimeMemo+PreLog	79.610 ms	24.298 ms

可以看到使用空间换时间（TimeMemo）的方法确实可以提升模型的运行速度。预处理log（PreLog）方法的实验效果一般，并没有显出提升。
PreLog方法效果一般的主要原因时这样的。在kernal函数中相比于计算信息熵的部分，枚举位置部分的时间开销更大。4.4节对算法的瓶颈进行了探究，探究结果表明，计算信息熵与否对运行时间基本没什么影响。优化一个小头的开销显然不会对算法产生很大的速度上的提升。

4.3 内存优化

这里设定4.2中的 TimeMemo+PreLog 算法为baseline，继续探索内存优化的作用。

方法	总时间	最大时间
baseline	79.610 ms	24.298 ms
StaticMemo	74.204 ms	23.649 ms
ShareMemo	78.127 ms	24.640 ms

如表格所示，共享内存对算法的优化有限，但是静态内存对算法的优化明显。这是因为对于本次实验来说，性能瓶颈既不在log计算上也不再全局内存访问上，而是在动态内存开设和内存拷贝上。4.4节会对本次实验的性能瓶颈进行探究。

4.4 什么是性能瓶颈

这里我对算法的性能瓶颈进行探究。取TimeMemo版本的代码作为baseline，我们进行了如下实验。

- 1. 只扫描矩阵位置，不计算交叉熵
- 2. 不扫描矩阵位置，不计算交叉熵
- 3. 使用静态内存，不进行Malloc（参考2.3.1），kernal函数也不做任何计算

方法	总时间	最大时间
TimeMemo	80.337 ms	24.399 ms
TimeMemo+NoCalc	79.214 ms	24.333 ms
TimeMemo+NoScan	76.201 ms	24.380 ms
TimeMemo+NoScan+NoMalloc	67.364 ms	21.609 ms

可以看到不计算交叉熵并不会节省多少时间，不扫描矩阵也不会节省多少时间。不Malloc确实可以剩下一定的时间。剩余的时间消耗主要来源于CPU和GPU间的数据传输。

5 OpenMP版本对比

这里使用OPEMMP实现了相同的功能。这里只实现了一个较为简单的版本，只使用了时间换空间这一个优化。实现思路比较简单，将输入数据分为1000个block，每个block执行一段坐标点的信息熵计算。相关代码如下。

```
void ompwork(int width, int height, float *input, float *output) {
    int k;
    int size = width * height;
    int TH_NUM = min(size, 1000);
    int Blo_Size = size / TH_NUM;
    #pragma omp parallel for private(k)
    for(k=0; k<TH_NUM; k++)
    {
        int _l = Blo_Size * k;
        int _r = _l + Blo_Size;
        if(k == TH_NUM-1) _r = size;
        for(int idx=_l;idx<_r;idx++){
            int x = idx / height, y = idx % height;
            int L = max(0, x-2), R = min(x+3, width);
            int D = max(0, y-2), U = min(y+3, height);
            int all_idxs = (R-L) * (U-D);
            float ans = 0, prob;
```

```
            char Cnt[16] = {0};
            for(int i=L;i<R;i++)
                for(int j=D;j<U;j++) {
                    Cnt[(int)input[i * height + j]]++;
                }
            for(int i=0;i<16;i++)
                if(Cnt[i]) {
                    prob = 1.0f * Cnt[i] / all_idxs;
                    ans -= prob * log(prob);
                }
            output[idx] = ans;
        }
    }
}
```

这里和只进行空间换时间优化的程序进行对比。可以发现使用OpenMP的程序运行时间慢了很多。这主要时因为GPU的核数多余CPU核，比如集群上这台机器的CPU只有32个核，并行计算能力显然弱于GPU。

方法	总时间	最大时间
TimeMemo+CUDA	80.337 ms	24.399 ms
TimeMemo+OMP	656.106 ms	2826.132 ms

6 总结

本次实验我分别使用CUDA核OpenMP实现了矩阵求信息熵的算法，根据算法特性和存储特性进行优化，并在GPU集群上进行测试。除此之外，我还对程序的计算瓶颈进行测试，并得到了一些结论。