19335030-陈至雪-作业2

作业要求

> 1. (coding) 利用CUDA计算二维数组中以每个元素为中心的熵
> – 给定一个整型矩阵，其中元素值的范围为[0, 15]
> – 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
> note：
> – 矩阵的读取、写入代码已提供。
> – 提供的测试数据中，input1.bin为 8 * 8 维的矩阵，input2.bin为 2048 * 2048 维的矩阵，可通过如下的脚本进行调用（默认调用input1.bin）：
> ./main input1.bin output1.bin
> ./main input2.bin output2.bin
> 最终应生成output1.bin，output2.bin 两个文件。
> 提供的矩阵仅用于测试代码正确性。
> – 最终提交的代码中要求对于长宽都大于0，总的大小不超过显存大小的矩阵都能正常运行。
> 2. (writing) 回答以下问题
> – 2.1. 介绍程序整体逻辑，包含的函数，每个函数完成的内容。（10分）
> • 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
> – 2.2 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。（15分）
> – 2.3. 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。（15分）
> – 2.4. 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。（40分）
> • 例如，使用共享内存及不使用共享内存
> • 优化失败的版本也可以进行比较
> – 2.5. 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。（20分）
> – 2.6. 可选做：使用OpenMP实现并与CUDA版本进行对比。（20分）
>
> submission:
> 作业提交内容：需提交一个.zip文件，需包括main.cu，output1.bin， output2.bin 以及实验报告pdf。
> zip文件命名格式：姓名 _ 学号 _ homework2; 如需提交不同版本，则命名格式：姓名 _ 学号 _ homework2 _ v2等。
> 作业提交方式: https://www.scholat.com/course/muticore2022
> 作业提交截止时间： 6月7日晚11时59分

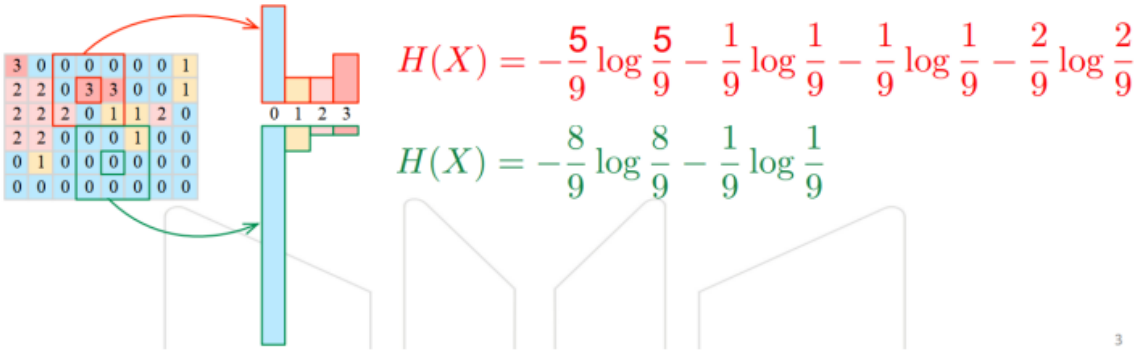
实验原理

熵是信息论中的基础概念，用于衡量随机变量分布的“混乱程度”。熵的公式如下所示：

$$H(X) = -\sum_i p_i . \log p_i$$

其中$p_i = p(X = x_i)$。

下面的图解释了熵的计算：



设 n 为领域内元素的总数，n_i为该领域内元素值为i出现的次数，则有
$H(X) = \log(n) - \sum_i \frac{n_i}{n}\log(n_i)$

CudaBaseline版：

> 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

按照二维的方式对线程进行组织和分块，将线程一一映射到对应位置上，每个线程适用串行的方法计算对应位置的熵值。除此之外，每个线程在逻辑上没有分配额外的任务。

如图所示，每个线程读入其负责的块的边界框内的元素，并对每个数出现的频数进行统计。由于每个元素为[0,15]的整型，因此为每个线程创建一个本地数组 num[16]，记录在此区间内出现的数字的频数。

Version1——使用表存储log1~log25的值

> – 2.3. 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?

由于边界框的大小是 5 * 5，这说明了每计算一次熵只有 25 个元素被考虑在内，因此p_i可能的取值范围是$\frac{0}{25}$~$\frac{25}{25}$，取对数后，对数计算就只局限在 log1 ~ log25，将这几个常数存储在寄存器中，而不必每次都计算一遍，可以加快计算速率。

Version2——使用全局内存存储表

使用：

- 声明一个全局变量要在全局作用域内用`__device__`关键字修饰；
- 初始化用`CHECK(cudaMemcpyToSymbol(logcons, (const double*)logcons_h,sizeof(logcons_h)));`
    - 其中`logcons_h`为在main函数初始化好的常类型变量。

> 全局内存是GPU中最大，延迟最高且最常使用的内存。
>
> 它的声明可以在任何SM设备上被访问到，并且贯穿应用程序的整个生命周期
>
> 全局内存变量可以被静态声明或动态声明。
>
> 在设备代码中，静态地声明一个变量：`__device__`
>
> 动态分配则使用`cudaMalloc`
>
> 全局内存分配空间存在于应用程序的整个生命周期中，并且可以访问所有核函数中的所有线程。由于线程的执行不能跨线程块同步，**不同块内的线程并发修改全局内存的同一位置会出现问题**
>
> 全局内存常驻于设备内存中，可通过32/64/128字节的内存事务进行访问，**这些内存事务需要自然对齐**，即首地址必须是32/64/128字节的倍数
>
> 一般情况下，用来满足内存请求的事务越多，未使用的字节被传输回的可能性越高，这就造成数据吞吐量降低

Version3——使用共享内存存储表

对数表中的数据每个线程在计算时都会用到，因此将对数表存储在共享内存中，可以加快线程访问对数数据的速率。使用共享内存要用`__shared__`关键字对变量进行声明，并且它不能直接进行初始化，所以由线程并行地进行初始化会比较快，主要要使用`__syncthreads()`对线程进行同步。

> **共享内存（Shared memory）**是位于每个**流处理器组（SM）**中的高速内存空间，主要作用是存放一个**线程块（Block）**中所有线程都会频繁访问的数据。**流处理器（SP）**访问它的速度仅比**寄存器（Register）**慢，它的速度远比全局显存快。

Version4——OpenMP版本，使用寄存器存储对数表

使用OpenMP进行并行化计算，一次循环迭代计算输出矩阵一个位置的元素。由于循环之间没有依赖关系，可以直接用`#pragma omp parallel for`并行化。并使用寄存器将对数表进行存储。

查看GPU的硬件配置

```
Device Name : Tesla T4.
totalGlobalMem : 15843721216.
sharedMemPerBlock : 49152.
regsPerBlock : 65536.
warpSize : 32.
memPitch : 2147483647.
maxThreadsPerBlock : 1024.
maxThreadsDim[0 - 2] : 1024 1024 64.
maxGridSize[0 - 2] : 2147483647 65535 65535.
totalConstMem : 65536.
major.minor : 7.5.
clockRate : 1590000.
textureAlignment : 512.
deviceOverlap : 1.
multiProcessorCount : 40.
```

参数说明：

> **name**
> *用于标识设备的ASCII字符串;*
> **totalGlobalMem**
> *设备上可用的全局存储器的总量,以字节为单位;*
> **sharedMemPerBlock**
> *线程块可以使用的共享存储器的最大值,以字节为单位;多处理器上的所有线程块可以同时共享这些存储器;*
> **regsPerBlock**
> *线程块可以使用的32位寄存器的最大值;多处理器上的所有线程块可以同时共享这些寄存器;*
> **warpSize**
> *按线程计算的warp块大小;*
> **memPitch**
> *允许通过cudaMallocPitch()为包含存储器区域的存储器复制函数分配的最大间距(pitch),以字节为单位;*
> **maxThreadsPerBlock**
> *每个块中的最大线程数*
> **maxThreadsDim[3]**
> *块各个维度的最大值:*
> **maxGridSize[3]**
> *网格各个维度的最大值;*
> **totalConstMem**
> *设备上可用的不变存储器总量,以字节为单位;*
> **major,minor**
> *定义设备计算能力的主要修订号和次要修订号;*
> **clockRate**
> *以千赫为单位的时钟频率;*
> **textureAlignment**
> *对齐要求;与textureAlignment字节对齐的纹理基址无需对纹理取样应用偏移;*
> **deviceOverlap**
> *如果设备可在主机和设备之间并发复制存储器,同时又能执行内核,则此值为 1;否则此值为 0;*
> **multiProcessorCount**
> *设备上多处理器的数量。*

扩展矩阵规模

老师给定的数据集中只有 `8*8` 和 `4096 * 4096` 两种规模的矩阵，但在这两个规模的矩阵上，并不能看出不同优化策略下的改善效果，因此我另外使用了矩阵规模为 `10240 * 10240` 和 `16384 * 16384` 的矩阵验证不同版本的运行效率。

程序实现

- CudaBaseline版见文件 `cudaBaseline.cu`
- Version1见文件 `cuda_v1.cu`
- Version2见文件 `cuda_v2.cu`
- Version3见文件 `cuda_v3.cu`
- OpenMP版见文件 `omp.cpp`

- 查看GPU的硬件配置见文件`GetDeviceProperties.cu`
- 矩阵生成代码见文件`matrix.c`
- 数据可视化代码见文件`res.ipynb`

- cuda版编译：`nvcc -o cudabaseline.o cudaBaseline.cu`
- cuda版运行：`./cudabaseline.out input1.bin cudaBaseline_1.bin`
- OpenMP版编译：`g++ -fopenmp omp.cpp -o omp`
- OpenMP版运行：`./omp`

结果分析

CudaBaseline

计算时间为计算十次取平均时间。

矩阵规模	计算时间（s）
8 * 8	0.000193
2048 * 2048	0.047312
4096 * 4096	0.157847
10240 * 10240	0.576712
16384 * 16384	1.279435

cuda_v1：使用表存储log1~log25的值

矩阵规模	计算时间（s）
8 * 8	0.000160
2048 * 2048	0.058350
4096 * 4096	0.216828
10240 * 10240	1.051273
16384 * 16384	2.657742

cuda_v2：使用全局内存存储表

矩阵规模	计算时间
8 * 8	0.000164
2048 * 2048	0.039906
4096 * 4096	0.14313
10240 * 10240	0.508098
16384 * 16384	1.059883

cuda_v3：使用共享内存存储表

矩阵规模	计算时间（s）
8 * 8	0.000161
2048 * 2048	0.035979
4096 * 4096	0.142145
10240 * 10240	0.458670
16384 * 16384	0.975408

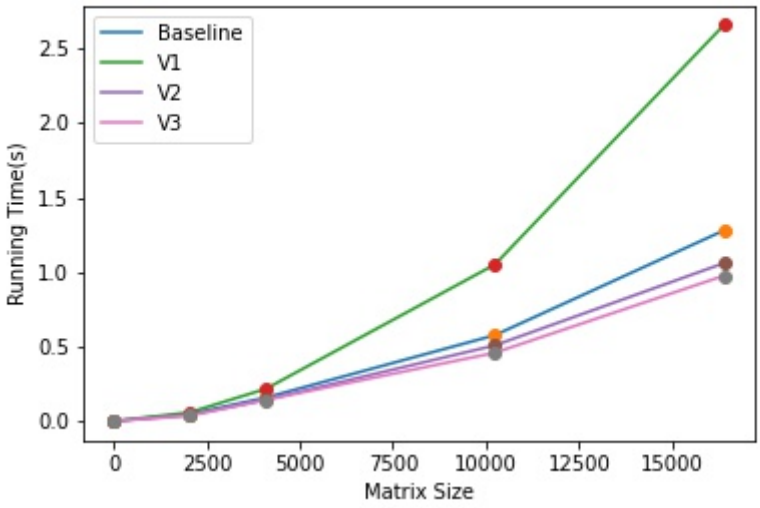
> 如使用查表对log1~log25进行查表，是否能加速运算过程？请通过实验收集运行时间，并验证说明。（15分）

OpenMP：使用寄存器存储对数表

矩阵规模	计算时间（s）
8 * 8	0.630909
2048 * 2048	23.623844
4096 * 4096	81.882156
10240 * 10240	504.479889
16384 * 16384	1312.428223

数据可视化

CUDA各版本间的对比：



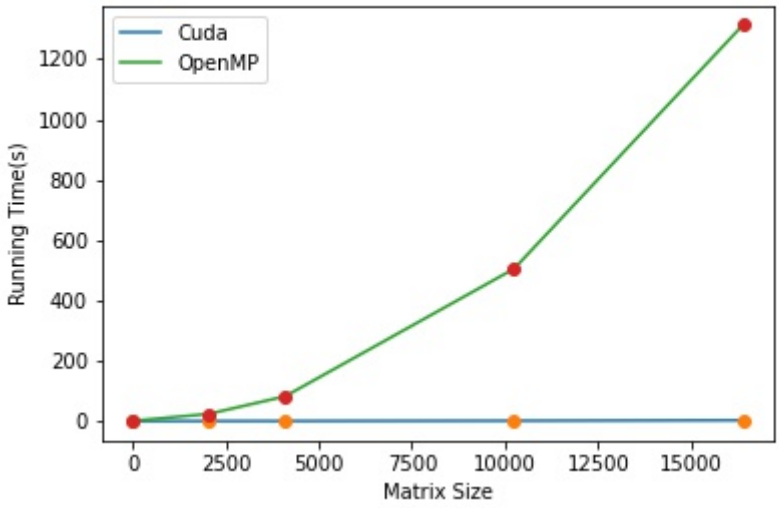
可以看到，对前两个矩阵规模的运行，各cuda版本的表现差距不大。随着矩阵规模增大，只有V1版本（用寄存器存储对数表）的运行时间比CudaBaseline版本的运行时间长，可能导致的原因是存储对数表占用了线程运行时的寄存器资源，使活跃线程数减少，或是溢出的寄存器数据被保存到了全局内存中，使数据访问效率降低。

> 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。

V2版本（使用全局内存存储对数表）和 v3版本（使用共享内存存储对数表）的运行时间比Baseline版本短，但是 V2 版本比 V3 版本的运行时间长。说明使用查表对log1~log25进行查表，能加速运算过程。而且使用共享内存对对数进行存储更适合该情况，原因是，对于一个 block 中的所有线程可以使用 shared memory 共享同一张表，且对共享内存的访问比全局内存快。

> – 2.6. 可选做：使用OpenMP实现并与CUDA版本进行对比

CUDA VS OpenMP：



此处的CUDA版本取了使用寄存器存储对数表的 V2 版本，可以看到当矩阵规模增大时，OpenMP版本的运行时间呈指数级增长，而CUDA版本的运行时间则变化相对较小，说明对于数据量大的运算，CUDA显现出其优势。