multicore hw1 scaffold

学校	学院	专业	年级	学号	姓名
中山大学	数据科学与计算机学院	计算机科学与技术 （超算方向）	17	17341163	吴坎

按照公式 $H(X) = -\sum_i p(X = x_i) \log p(X = x_i)$ 计算二维数组中以每个元素为中心的熵：

- 输入二维数组及其大小
  - 数组元素为 $[0, 15]$ 的整型
- 输出一个浮点型二维数组
  - 输出精度保留到小数点后五位
  - 每个元素值为输入数组中对应位置为中心大小为五的窗口中值的熵
  - 边界窗口只考虑数组内的值

实验中分别在 CUDA（单卡）、OPENMP、OPENMP+CUDA（四卡）三种计算环境上实现了上述实验要求，并实现了若干种优化版本进行对比。由于提供的最大的样例输入大小只有 2560×2560 这一数量级，并没有完全发挥多核计算环境的优势，我也分别构造了三组大小分别为 400×400、2560×2560、10240×10240 的输入对程序进行了 benchmark 测试。最终在实验结果的基础上总结得到了一些影响本次程序性能的因素。

如果要运行我的实验程序，可以终端直接在当前目录运行如下指令：

```
#chmod 777 ./RUNME.sh # 可解决有时出现文件权限错误的问题
./RUNME.sh | tee screen.log
```

该条语句会自动编译并运行实验结果，同时保存屏幕日志到 `screen.log` 文件中。我的运行结果可以直接看当前目录下的 `screen.log`

        - v8
        - v9
        - v10
        - v11
        - v12
        - v13
        - v14
    - 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
        - 读入对数表
        - 读入输入矩阵
        - 写出输出矩阵
    - 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 $\log 1$ ~ $\log 25$ 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
    - 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
    - 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
        - 分析
        - 总结
        - 其他可提升的地方
    - 可选做：使用 OpenMP 实现并与 CUDA 版本进行对比

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

我分别使用 CUDA 实现了 v0~v9 版本、OPENMP 实现了 v10~v13 版本、OPENMP+CUDA 实现了 v14 版本。不同版本使用 namespace 进行封装，除了 v14 版本复用了 v7、v12 版本之外，各命名空间互不干扰，没有冲突。这样做既方便了代码复用，也方便横向对比各个版本之间的性能差异。如下，如果要在它们之间进行切换，只需要修改 `sources/src/core.cu` 最后几行 `cudaCallback` 函数中实际调用的版本即可。

```
void cudaCallback(
    int width,
    int height,
    float *sample,
    float **result)
{
    v14::cudaCallback(width, height, sample, result);
}
```

对于 v0~v9 版本来说，每个命名空间里仅包含两个函数：`cudaCallback` 函数是外部调用的接口，`cudaCallbackKernel` 函数是其调用的核函数。

对于 v10~v14 版本来说，每个命名空间里仅包含一个函数 `cudaCallback` 作为外部调用的接口。虽然里面的内容实质上与 cuda 不怎么相关，但是还是保持了命名上的一致性。

v14 版本是最后得到的最优版本。

v0

v0 版本是 cuda 版本的 baseline。实现的逻辑非常简单粗暴：将各个线程按照二维方式组织和分块，线程一一映射到对应位置上，每个线程使用串行方法计算一个位置的熵值。每个线程块在逻辑上没有额外的任务。

上图可以描述运行中某两个线程的工作情况。每个线程读入自身及周围邻域的元素值（图中实线表示的区域），写出到输出矩阵对应位置（图中内层虚线表示的区域）。

v1

v1 版本在 v0 版本的基础上预处理了整数 $[0, 25]$ 范围内的对数表（原理见下），并直接保存在寄存器上。

v2

v2 版本同样在 v0 版本的基础上预处理了对数表，但是对数表保存在 shared memory 中。

v3

v3 版本同样在 v0 版本的基础上预处理了对数表，但是对数表保存在 constant memory 中。

v4

v4 版本同样在 v0 版本的基础上预处理了对数表，但是对数表保存在 device memory 中。

v5

v5 版本在 v4 版本的基础上使用 texture memory 加速对数表的读取。

v6

v6 版本在 v1 版本上换用更小的 `signed char` 类型代替 `int` 型作为计数器。

v7

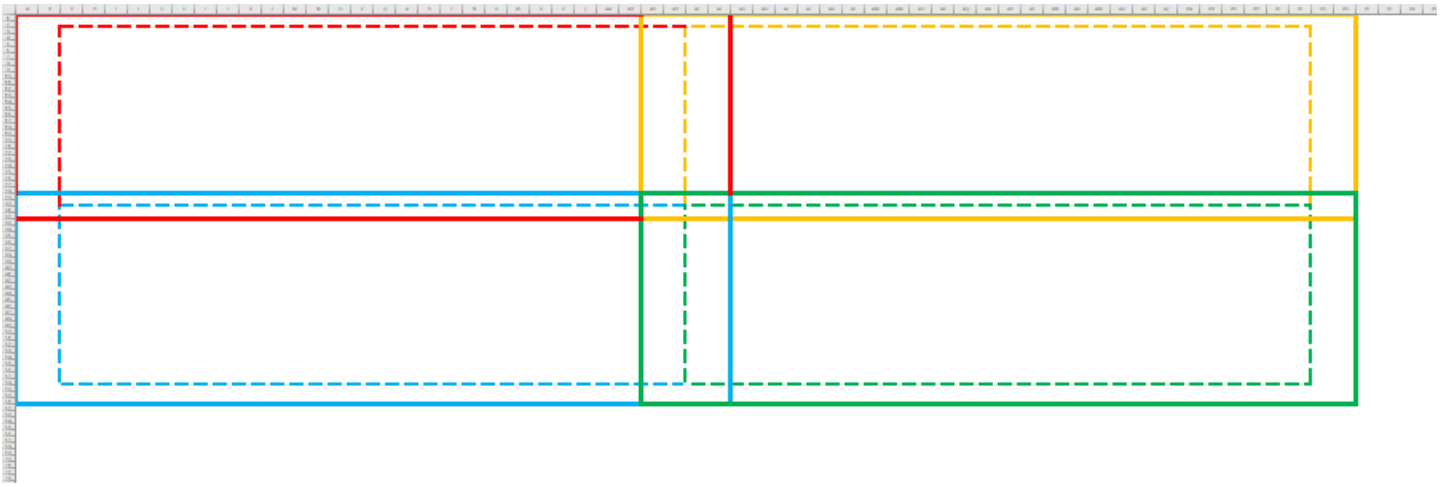
v7 版本在 v6 版本的基础上，使用更小的 `float` 类型来存储对数表，但是计算时使用 `double` 类型进行计算（即混合精度）。

v8

v8 版本在 v7 版本的基础上，使用 2D texture memory 加速输入矩阵的读取。

v9

v9 版本在 v7 版本的基础上，使用 shared memory 加速输入矩阵的读取。在这个版本里，单个线程块的大小为 32×32，每次计算 28×28 大小的矩阵（边界宽度为 2 的环的熵并非最后答案的熵）。



上图等比例地反映了四个空间上相邻的 block 所分到的任务。其中，不同元素的实线代表每个线程块实际读取的范围，虚线代表实际写出的范围。要注意的是上面实线虚线之间会有覆盖，但是不同矩形实际对应空间是能够明显区分的。

v10

v10 版本是 openmp 版本的 baseline，其原理和 v0 版本相同，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。

v11

v11 版本类似于 v1 版本，在 v10 版本的基础上预处理对数表到寄存器。

v12

v12 版本类似于 v7 版本，使用混合精度加速计算过程。

v13

v13 版本在 v12 版本的基础上预处理二维前缀和，降低统计各元素出现次数的时间复杂度。

具体来说，我针对 $X = x_i, i \in [0, 15]$ 分别线性预处理 16 个二维前缀和：$S_{a,b} = \sum_{p=0}^{a} \sum_{q=0}^{b} [X_{p,q} == x_i]$。

如上，红色面积为 $S_{a,b}$，红+黄部分面积总和为 $S_{a+5,b}$，红+蓝部分面积总和为 $S_{a,b+5}$，红黄蓝绿四部分面积总和为 $S_{a+5,b+5}$。这样要检索一个 5×5 的绿色窗口中特定元素的出现次数只需要四次访存即可：$S_{a+5,b+5} - S_{a+5,b} - S_{a,b+5} + S_{a,b}$。

当然，由于本问题中变量的值域有 16 个元素，实际上总的访存次数 $16 \times 4 = 64$ 是要超过 baseline 版本的 25 次的。但是，当问题的窗口变得更大（例如 9×9），或是变量的值域变小的时候，此算法因为有更低的时间复杂度而值得期待。

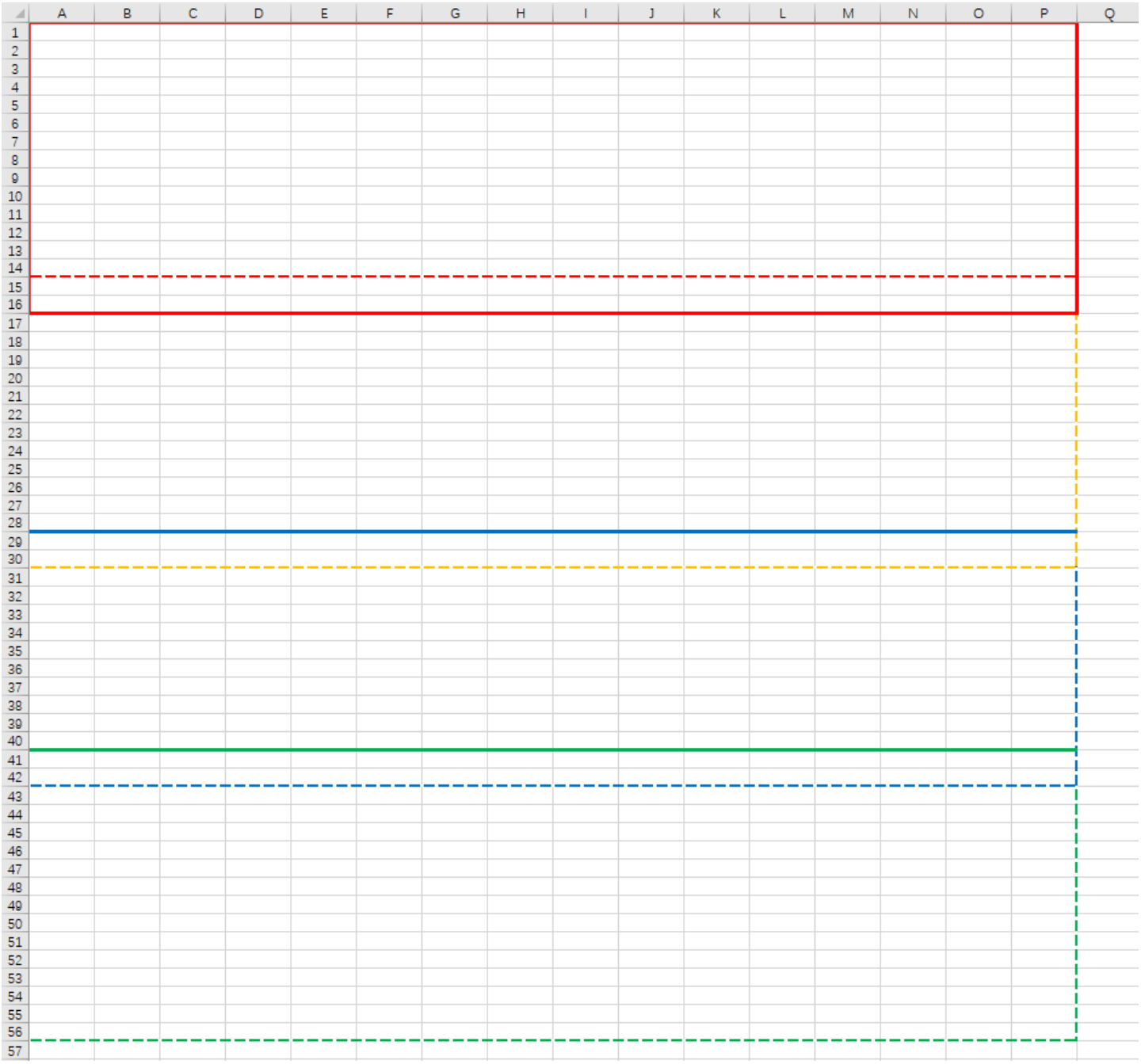
v14

v14 版本使用了 OPENMP+CUDA。由于提供的实验环境中有四张显卡，只使用其中的一张显卡未免过于浪费，于是考虑使用多张显卡对程序进行加速。

对于这个程序来说，要使用多张显卡对程序进行加速，我有大致如下两个思路：

1. 任务并行，每张显卡负责计算不同的 $X = x_i$ 对于答案的贡献，然后使用多卡上的 Reduce 算法将其汇总
2. 数据并行，每张显卡计算答案矩阵的一个部分

我选择使用第二个思路，从而可以直接复用之前单卡的版本，只需要特殊处理一下边界即可。我按照 height 将输入划分成四个子问题，随后多线程调用最佳的单卡版本（v7）进行计算（每个线程绑定不同的显卡）。



如上，四种颜色代表每张显卡实际划分到的任务区间，虚线代表实际划分到任务中有效的部分，注意到边界处有高为 2 的重叠部分。这样划分的好处是无需对输入数据重新打包，只需要相对于初始地址计算不同的偏移量。

根据官网上查到的数据，我们集群使用的 Tesla V100 有 80 个 SM，每个 SM 最多同时有 2048 个活跃线程。因此，使用多于 $80 \times 2048 = 163840$ 个线程的时候能够使显卡满载。于是在输入的数据数量小于 163840 的时候（当然，没有显卡的时候也会）直接调用表现更佳的 OPENMP 版本。经测试结果来看，这个界限的划分是比较合理的。

得益于对之前 CUDA 单卡版本和 OPENMP 版本的代码复用，多卡版本只用了非常少的代码（40 行以内）就完成了数据并行，还是非常划算的。

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

对于运行在 CPU 版本，内存结构在 C 语言的层面几乎是不可见的，通常来说数据是被存放在内存上，CPU 通过缓存读取；少部分经常访问的数据编译器会将其存放在寄存器中。因此下面主要针对 GPU 上的访存进行分析，主要有如下三个部分：读入对数表（如果有做预处理的话）、读入输入矩阵、写出输出矩阵。

读入对数表

最直观的想法就是把对数表直接保存在寄存器中。在 v1 版本中，我将对数表保存在寄存器中（值得一提的是，为保证结果的计算精度，保存的对数表是 `double` 类型）。通过检查编译生成的 `.ptx` 汇编文件，可以发现其值在编译时就已经计算完成。然而 v1 版本甚至比 v0 版本还要慢，猜测过大的寄存器占用减少了运行时活跃线程的数量，或是溢出的寄存器被保存到 global memory 中了。

一个很自然的想法是要减少每个线程占用寄存器的数量。注意到对数表是固定的，因此在 v2 版本中我对于一个 block 中的所有线程可以使用 shared memory 共享同一张表，最终的运行时间要小于 v0 版本。由于 shared memory 并不允许直接初始化，需要在运行时由对应位置的线程计算，超出部分的线程空转，然后在 block 内进行线程同步。

如果在更大的范围共享这张表会如何？注意到对数表是常量，天生适合使用 constant memory。对 constant memory 的单次读操作可以广播到同个半线程束的其他 15 个线程，这种方式产生的内存流量只是使用全局内存时的 $\frac{1}{16}$。同时硬件主动把 constant memory 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，这同样减少了额外的内存流量。然而对于 v2 版本来说，使用 constant memory 的 v3 版本提速的很少，不到 3%。

为了验证 constant memory 的加速效果，我在 v4 版本中把对数表换成了 device memory（或者说就是 global memory），和 v3 版本相比确实没有明显变化。这说明在这里 constant memory 相对于 device memory 没有明显优势。为什么？我猜测，对于同一个半线程束的所有线程，其对应的元素及邻域不同，那么每种元素出现的次数是大概率不相同的。因此，线程束上的线程在同一时刻访问的对数表位置大概率不同。

经过上述分析，可以知道 constant memory 其实并不是很适合这个场景。于是我又想到了 texture memory。texture memory 不需要满足 global memory 的合并访问条件也可以优化邻域上的数据读取。我直接将 v4 版本中的 device memory 绑定 texture 进行访问。相对于 baseline，在 2560×2560 规模的输入上平均提高了大约 16% 的性能。我猜测原因是，由于这个表大小仅为 26，访问 texture memory 时很容易就把表的其他值传给了附近的线程，从而减少了内存流量，也没有带来过大的寄存器压力（因为不是所有线程都获得了整个表）。CUDA 还有一种 surface memory，相当于带写操作的 texture memory，这里不再尝试了。

值得注意的是，CUDA 中 texture memory 访问的带宽仅有 4 字节，意味着在这里我仅能使用单字节的 `float` 类型存储对数表。当然，也可以使用 `double` 类型存储，然后在核函数中读取两次然后使用强制类型转换，不过这么做并不优雅。但是如果直接使用 `float` 类型进行计算，由于 `float` 类型仅有七位有效数字，在进行乘法运算的时候非常容易丢失精度（经过测试数据验证，会在小数点后四

位开始产生比较大的误差）。因此，我选择使用**混合精度**的方法解决这个问题，即对数表使用 `float` 类型进行存储，但是在计算的过程中使用`double`进行计算。经过检验，使用混合精度的方案得出的结果在小数点后五位几乎没有差别，而运行时间却与完全使用 `float` 类型计算相差无几。

出于控制变量的原则，texture memory 带来的优化效果是否是因为使用了混合精度呢?

首先我要确认的是，减少寄存器占用确实能够提速。我在 v1 版本的基础上，将使用的计数器由 `int` 换成了 `signed char`，得到了 v6。由于邻域大小最大不过 25， `signed char` 有效范围是-128~127，因此这么做不会带来任何结果上的错误。效果是非常明显的，相对于 v1 版本带来了几乎一倍的速度提升!

随后我在 v6 版本基础上使用混合精度得到 v7，仍然能够得到 15% 左右的性能提升。

但是，经过验证，在 v7 版本上继续应用 shared memory、constant memory、texture memory 时却带来了负优化，这说明此时寄存器压力不再是程序的瓶颈，之前的问题得到了解答。至此对于对数表访存的优化告一段落。

读入输入矩阵

对于输入矩阵来说，每个位置上的元素被访问了 25 次之多，但是输入矩阵的大小通常大于 64K，不适合使用 constant memory。于是考虑使用 texture memory 和 shared memory 对其进行优化。

首先，我在 v7 版本的基础上使用 texture memory 得到 v8。和 v5 版本中的对数表相比，输入矩阵是二维存储的，因此这里也略有不同，绑定的是 2D texture memory。

然后，我在 v7 版本上使用 shared memory 对输入进行优化得到 v9。然而从结果上来看，v9 版本却是负优化（原因分析见后），因此我没有继续考虑 shared memory 和 texture memory 混合使用的方案了。

至此对输入矩阵的访存优化结束。从结果上来看，v8 版本是单卡上表现最佳的版本，不过与 v7 版本几乎相同。然而后续在多卡上进行测试时，v8 版本却明显比 v7 慢一些（原因不明，也许和纹理内存的调度机制有关），因此最后多卡版本中使用的是 v7 版本。

写出输出矩阵

由于输出矩阵中每个位置上仅有一次写操作，因此没有什么花里胡哨的操作了，直接写回 device memory 即可。

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 $\log 1 \sim \log 25$ 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

由于求解的窗口大小仅为 25，可以对求解公式做下述变形：

$$
\begin{aligned}
H(X) &= -\sum_i p(X=x_i)\log p(X=x_i) \\
&= -\sum_i \frac{n_i}{\sum_i n_i}\log\frac{n_i}{\sum_i n_i} \\
&= -\frac{1}{\sum_i n_i}\sum_i n_i \log\frac{n_i}{\sum_i n_i} \\
&= \frac{1}{\sum_i n_i}\left[\left(\sum_i n_i \log\sum_i n_i\right) - \left(\sum_i n_i \log n_i\right)\right] \\
&= \log\sum_i n_i - \frac{1}{\sum_i n_i}\sum_i n_i \log n_i
\end{aligned}
$$

在这个公式中，n_i 代表当前求值位置的邻域中满足 $X=i$ 的元素的数量，$\sum_i n_i$ 则表示邻域的大小，在这个问题里均不会超过 25。因此变形之后对数运算只涉及对整数 $[1,25]$ 的对数运算。此外，在对数表中赋值 $\log 0 \to 0$ 可以减少条件判断的次数。

实验中，我的 v1~v9 版本相对于 v0 版本、v11~v13 版本相对于 v10 版本均预处理了对数表。由于预处理结果存储位置不同，存储精度也不同，他们的表现也各有差异，详见下一问。

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

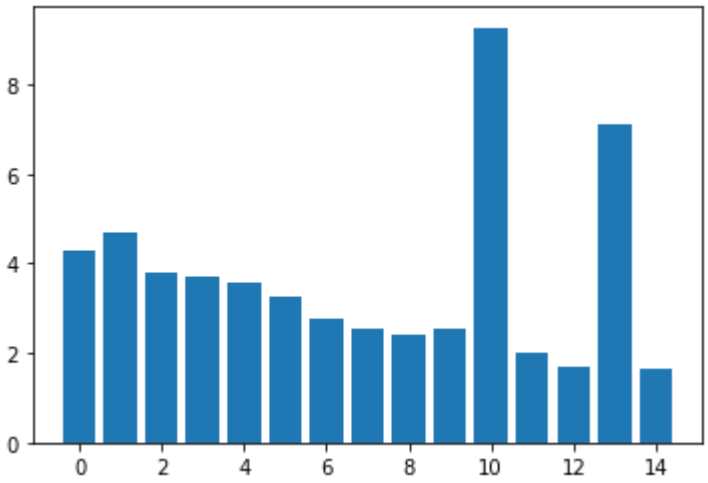
我一共实现了 v0~v14 共 15 个版本。由于 TA 提供的原始数据中只有 5×5、2560×2560 两个数量级别的测试数据，前者过小而使结果容易受系统噪声影响，后者也不是明显的大数据集，我也设计了 400×400、2560×2560、10240×10240 三组数据作为 benchmark。

我将 benchmark 的过程封装成 class，只需要在 `sources/src/core.cu` 最后几行中的相关内容取消注释即可运行。

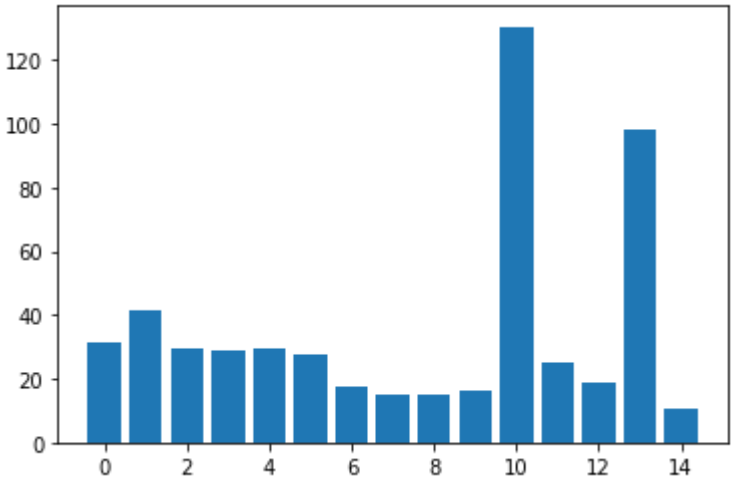
```
static WarmUP warm_up(1, 1);
static Benchmark
    benchmark400(400, 400),
    benchmark2560(2560, 2560),
    benchmark10240(10240, 10240);
```

此外，实验中发现对于一张显卡上首次运行的核函数会有 30ms 左右的时间用于冷启动。为排除这部分对于 benchmark 的影响，我也封装了一个 `WarmUP` class，提前使用大小为 1×1 的数据来对每张显卡进行“热身”操作。

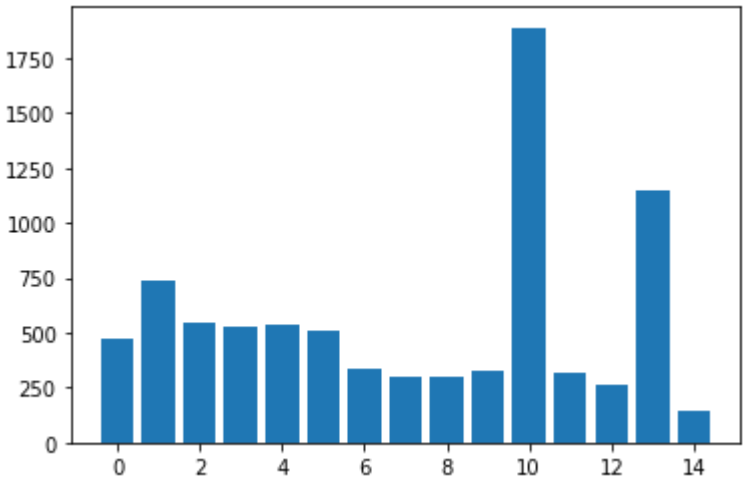
以下是对 400×400 结果的可视化（时间单位为 ms，下同）：



以下是对 2560×2560 结果的可视化：



以下是对 10240×10240 结果的可视化：



以下是详细的数据：

版本	400×400	2560×2560	10240×10240
v0	4.292960	31.354719	473.793365
v1	4.708352	41.344097	733.725342

版本	400×400	2560×2560	10240×10240
v2	3.796704	29.511265	549.282898
v3	3.714496	28.757376	530.392090
v4	3.548896	29.387968	532.152283
v5	3.276384	27.691393	508.423401
v6	2.763712	17.299105	331.066833
v7	2.552288	15.259968	297.580750
v8	2.426016	15.150208	296.813080
v9	2.561056	16.476641	323.676361
v10	9.266080	130.554169	1887.637573
v11	1.995008	24.977760	316.286682
v12	1.709536	18.499392	258.574951
v13	7.093696	98.223297	1148.376221
v14	1.636960	10.473536	139.679169

可以比较直观地看出各个优化手段的效果。由上面的数据可以看出，单卡版本中最快的的是 v7、v8；OPENMP 版本中最快的是 v12；全局最优秀的是多卡版本 v14，并且在大小 10240×10240 数据上相对于 v7 单卡版本有 $297.580750/139.679169 \approx 213\%$ 的加速比。详细分析见下一问。

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

分析

总的来说，v1~v5 版本虽然出发点是在围绕访存进行优化，但是实际上程序运行的瓶颈与提升效果都在于寄存器压力上（而且提升的并不多），后续 v6~v7 版本的优化效果也应证了这一点。

虽然单卡中最快的是 v8 版本，但是其相对于 v7 版本几乎没有提升，因为对输入数组的访问都是对连续地址的访问，texture memory 在这种情况下并不能额外提供多少加速（因为本身已经很快了）。

至于 v9 版本，虽然访存流量已经几乎减少到原来的二十五分之一，但是在各种规模上仍然没有提升！这是为什么呢？实际上，根据我在nvidia 官网上查到的数据，实验使用的显卡 tesla-v100 带宽高达 900GB /S，而 2560×2560 大小的矩阵只有 26.2144MB，理论上只需要不到 0.03ms 就可以完全读入（实际上会有差距，因为不是单次读入），因此输入数据完全不足以跑满显卡的带宽。而能够

跑满显卡带宽的数据大小已经远远的超过单张显卡 32GB 的显存了。而 v9 版本相对于之前的版本，代码逻辑更为复杂，所以可以说是得不偿失了。

因此，本次作业中影响程序性能的主要瓶颈并不是访存，而是**寄存器压力**，后者是通过影响**活跃线程数量**来影响最终性能表现的。这与我们通常编程时的一些常识与习惯（把尽量多的东西保存在寄存器里）有些冲突。

一方面，CPU 能够同时调度的线程数（核数）远少于 GPU，但是单个线程寄存器更少、流水线更短；另一方面，GPU 相对于 CPU $\frac{T_{计算}}{T_{访存}}$ 比更小，因此由于寄存器溢出导致活跃线程数量减少和对 global memory 的额外存取会很严重地影响程序表现。因此 GPU 上代码中优化策略的选择和效果与 CPU 版本会存在很多区别。

总结

结合本次实验和一些已有的经验，我认为写出一个比较高效的 CUDA 程序需要考虑以下这些因素：

- 选择好的并行算法，发掘更多的数据并行性
  - 避免过于复杂的核函数逻辑
- 保持 SM 尽可能忙碌，尽量利用所有的 SM 参与计算
  - 增加数据量
  - 减小线程块大小
  - 减少寄存器压力，增加活跃线程数量
- 优化存储器的使用
  - 全局存储器合并访问
  - 使用更快的 shared memory 或 constant memory、texture memory
- 增加计算的效率
  - 避免一些其他硬件上的限制
    - 减少同一个线程束上的条件分支
    - 减少 bank conflic
  - 使用更加高效的计算方法
    - 使用混合精度或更低精度的类型
    - 使用 `__fdividef(x, y)` 代替 `x / y` 等
    - 编译时开启 `-use_fast_math` 选项
    - 使用一些已有的高效实现，如 `<cuBlas_v2.h>` 等
  - 开启 `-O3` 选项，进行循环展开、流水线并发在内的很多编译器优化

其他可提升的地方

由于集群资源比较有限，我还有一些想法但是最后没有完成：

1. 多卡版本中使用 4 个线程分别调用单卡版本完成子问题的计算，然后将结果拷贝回最初的答案矩阵。实际上我们使用的集群有 32 个核，这里拷贝的过程可以让每个线程再次 fork 成 8 个线程，充分利用集群的算力。

2. 同上，既然要在多卡版本中同时利用 CPU 的算力的话，其实 CPU 也可以负载一部分的有效计算。不过这么做很容易带来木桶效应，因此就没有做了。
3. 多卡版本中数据划分是按照 height 来的。这样做虽然可以免去输入数据重新打包的开销，但是对于一些特殊的数据，比如大小为 10×10^7 的输入，按 height 划分显然不如按 width 划分来的优秀。
4. 一个有趣的问题是，使用的 V100 显卡是 NVIDA 生产的，而 nvcc 也是 NVIDA 开发的；然而使用的 CPU 是 Intel 生产的，对应的使用的 CPU 代码编译器 gcc 却不是 Intel 的，显然并不能充分挖掘其计算潜力。为此我使用了自己机器上的 icc 编译器重新编译了这个项目，成功让几个使用 OPENMP 的版本提速了 50%。但是对应的 CUDA 版本运行时会报错，提示纹理引用错误。可能是我在链接的时候哪里出了错吧…相同的代码在 gcc 下就可以正常运行，可以说是比较奇怪了。

可选做：使用 OpenMP 实现并与 CUDA 版本进行对比

详见之前的 v10~v14 版本，已经说的比较完整了，此处不再赘述。