多核程序设计

作业一：计算二维数组中心熵

数据科学与计算机学院　17大数据与人工智能

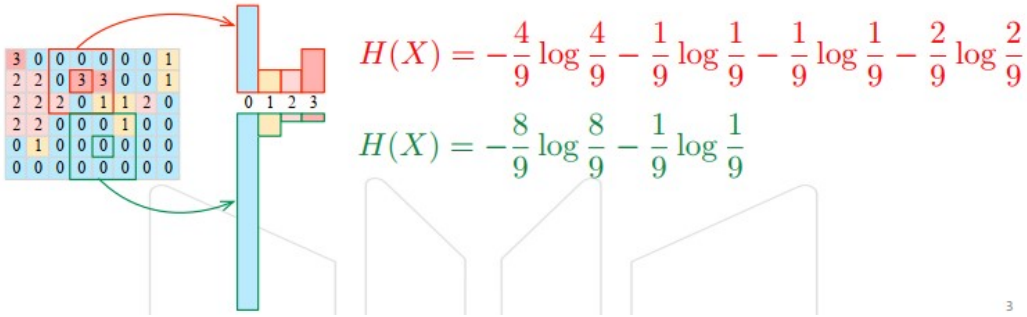
17341015　陈鸿峥

目录

一、 题目描述

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



回答以下问题：

1. 介绍程序整体逻辑，包含的函数，每个函数完成的内容。（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. 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）
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$\log 1 \sim \log 25$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。（15分）
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
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）
6. 可选做：使用OpenMP实现并与CUDA版本进行对比。（20分）

二、 问题简答

1. 见第4节。
2. 未加优化前的基线版本，输入矩阵放在全局内存中，其他计算所需数据则存放在寄存器中。由于输入矩阵在计算过程中需要被频繁访问，因此可将输入矩阵由全局内存读入至共

享内存，以提升存储访问性能。而对数表部分由于可以提前计算出来且在程序运行过程中不会修改，因此可以放在GPU的常量内存中。

3. 见第3节的推导。由于存在大量的内存访问冲突，同时计算开销小于存储访问开销，因此即使采用常量内存对数表的方式，也没有办法得到性能的大幅提升，见第5.2的实验结果。
4. 见第4.3和第5.2节。
5. 实验结果见第5.2节，可归纳知**内存访问**是影响CUDA程序性能的最大因素，数据到底放置在全局内存、共享内存还是常量内存对于程序整体性能有着巨大影响。但也并不是说数据存放在共享内存或是常量内存就一定快，数据访问模式往往与后面的算法计算方式密切相关。只有计算和存储相辅相成才有办法发挥硬件的最大效用。另外，**线程块**的大小同样对性能有着不可忽视的影响，要充分利用线程并做到负载均衡，同时要避免由于线程数与任务数除不尽导致的空闲线程，只有任务划分得好性能才会高。
6. 见第4.2和第5.2节。

三、 原理推导

设窗口大小为$M \times M$，本题中$M = 5$，由题意知中心熵为窗口内合法元素的熵之和。设窗口内合法的元素数目为N，每个元素值的数目为$N_i (i = 0, \ldots, 15)$，则有中心熵的计算公式

$$H(X) = \sum_{i=0}^{15} -\frac{N_i}{N} \log \frac{N_i}{N} , \tag{1}$$

且满足$N = \sum_{i=0}^{15} N_i$。

对式(1)进行变换有

$$H(X) = -\frac{1}{N} \sum_{i=0}^{15} N_i (\log N_i - \log N) \tag{2}$$

$$= -\frac{1}{N} \sum_{i=0}^{15} N_i \log N_i + \log N \tag{3}$$

进而可降低除法的次数，缩短计算时间。

同时可以看到这里的运算变成了整数的对数运算，由于$N_{\max} = M \times M = 25$，因此可以先计算出$[1, 25]$的所有对数值，然后通过查表法实现上述运算。

四、 程序逻辑

在本次实验中我实现了五个版本的程序，包括

- CPU单核程序：基础NumPy实现，参见`numpy`文件夹
- CPU并行程序：OpenMP实现，参见`openmp`文件夹

- GPU并行程序：CUDA三个优化版本实现
  - 基线版本：参见sources文件夹
  - 2D线程块版本：参见sources-2d文件夹
  - 共享内存版本：参见sources-shared_mem文件夹

由于OpenMP和CUDA版本的实现均采用了助教提供的代码模板，因此其中关于文件读入和结果输出的部分在此不再赘述。

1. NumPy实现

之所以选择NumPy将本次实验进行实现，一方面是为了提供Baseline程序用于结果验证，另一方面则是尝试探索利用CPU计算性能可以达到多高。NumPy虽然是Python的库，但是其底层代码都是采用C进行编程，并且利用了Intel的MKL库，因此性能其实是很高的，作为CPU Baseline也相当合适。

在Python里实现起来也相当方便，对于数组内的每一元素遍历，通过NumPy数组的切片(slicing)访问，可以得到每个中心点对应的窗口，并对其计算直方图，最终利用向量计算即可求得中心熵。核心代码如下所示。

```
def calculate(mat):
    """
    Calculate the central entropy of the input matrix

    Parameters:
    -----------
    mat : A 2D numpy array

    Return:
    -----------
    res : The central entropy matrix of mat
    """
    height, width = mat.shape
    res = np.zeros(mat.shape)
    for i in range(height):
        for j in range(width):
            # extract window
            kernel = mat[max(0,i-KERNEL_RADIUS):min(height,i+KERNEL_RADIUS+1),
                         max(0,j-KERNEL_RADIUS):min(width,j+KERNEL_RADIUS+1)]
            size = kernel.shape[0] * kernel.shape[1]
            # count histogram
            unique_elements, counts_elements = np.unique(kernel, return_counts=True
                ↪ )
            res[i][j] = -np.sum(counts_elements * np.log(counts_elements)) / size +
                ↪  np.log(size)
```

```
24      return res
```

NumPy实现的亮点在于高效的**向量化操作**，而非普通的裸串行代码。完整代码请见`numpy/main.py`。

2. OpenMP实现

接下来我采用C++实现了CPU的并行版本，利用OpenMP进行加速，测试并行后的性能。

C++的代码逻辑没有Python那么紧凑，但是还是非常清晰的。外层两层循环遍历输入数组中的每一个元素，内层两层循环计算直方图。

注意到这里为了跟CUDA的实现保持一致，输入统一采用一维数组存取，需要先计算出下标`idx`再去读取数据。详细步骤如下：

1. 对于矩阵中的每个点，统计其周围5×5窗口内每个元素出现的次数。这里直接开设大小为16的桶`cnt`，每次遇到一个元素就往对应桶里加1。由于窗口超出原矩阵范围的点不纳入计算，因此需要另外采用变量`valid`记录合法元素数目。
2. 等遍历完该点对应窗口中的所有元素，按照熵的公式进行累积。
3. 采用公式1对累积和进行操作，得到最终的中心熵。

为利用多核的并行计算能力，直接在最外层循环添加`#pragma omp parallel for`。由于内层循环并没有发生共享内存的写冲突，因此无需采用原子操作等来对线程进行同步。核心代码逻辑如下，完整代码可见`openmp/src/core.cpp`。

```
1   /*!
2    * Core execution part of OpenMP
3    *   that calculates the central entropy of each point.
4    * \param width The width of the input matrix.
5    * \param height The height of the input matrix.
6    * \param input The input matrix.
7    * \param output The output matrix.
8    * \return void. Results will be put in output.
9    */
10  void kernel(int width, int height, float *input, float *output) {
11      #pragma omp parallel for num_threads(24)
12      for (int h = 0; h < height; ++h) {
13          for (int w = 0; w < width; ++w) {
14              int idx = h * width + w;
15              int cnt[16] = {0};
16              int valid = 0;
17              int x = idx % width;
18              int y = idx / width;
19              // each thread first counts the histogram of idx
20              for (int i = -2; i < 3; ++i)
```

```
                for (int j = -2; j < 3; ++j) {
                    if (y + i >= 0 && y + i < height &&
                        x + j >= 0 && x + j < width) {
                        int in = input[idx + i * width + j];
                        cnt[in]++;
                        valid++;
                    }
                }
            // calculate entropy
            float sum = 0;
            for (int i = 0; i < 16; ++i) {
                int ni = cnt[i];
                if (ni != 0) {
                    #ifdef LOOKUP
                    sum += ni * log_table[ni];
                    #else
                    sum += ni * log(ni);
                    #endif
                }
            }
            #ifdef LOOKUP
            output[idx] = -sum / valid + log_table[valid];
            #else
            output[idx] = -sum / valid + log(valid);
            #endif
        }
    }
}
```

这里需要注意，由于C++没有Python中的切片功能，因此需要人工判断数组是否越界。同时，OpenMP的实现中也给出了两种不同计算log值的方式，一种采用预设好的对数表进行查表计算（开启`LOOKUP`的宏），另一种则采用`<math.h>`库中的`log`函数。两种方法的比较会在最后的实验部分（第5节）给出。

3. CUDA实现

然后我又着重在GPU上实现并优化了本次作业的内容。

(i) 基线版本

基线版本的CUDA程序与OpenMP的实现非常相似，不同之处在于CUDA的实现需要用`__global__`来声明核函数，同时在核函数内部只用考虑单一线程的计算情况，因此只需描述内部窗口的两层循环。

其他计算过程与OpenMP一致，即统计直方图，并计算中心熵。这里同样提供了使用查找

表和不使用查找表算对数的方法。由于$[\log 1, \log 25]$的值是可以预先算出来且在执行过程中保持不变的，因此CUDA里的查找表可放在常量内存中，通过__constant__声明。后文实验中会详细对这两种方法进行比较。

核函数如下所示，完整代码请见sources文件夹。

```
/*!
 * Core execution part of CUDA
 *   that calculates the central entropy of each point.
 * \param size The size of the input matrix.
 * \param width The width of the input matrix.
 * \param height The height of the input matrix.
 * \param input The input matrix.
 * \param output The output matrix.
 * \return void. Results will be put in output.
 */
__global__ void kernel(int size, int width, int height, float *input, float *
    ↪ output) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < size) {
        int cnt[16] = {0};
        int valid = 0;
        int x = idx % width;
        int y = idx / width;
        // each thread first counts the histogram of idx
        for (int i = -2; i < 3; ++i)
            for (int j = -2; j < 3; ++j) {
                if (y + i >= 0 && y + i < height &&
                    x + j >= 0 && x + j < width) {
                    int in = input[idx + i * width + j];
                    cnt[in]++;
                    valid++;
                }
            }
        // calculate entropy
        float sum = 0;
        for (int i = 0; i < 16; ++i) {
            int ni = cnt[i];
            if (ni != 0) {
                #ifdef LOOKUP
                sum += ni * log_table[ni];
                #else
                sum += ni * logf(ni);
                #endif
            }
        }
```

```
        #ifdef LOOKUP
        output[idx] = -sum / valid + log_table[valid];
        #else
        output[idx] = -sum / valid + logf(valid);
        #endif
    }
}
```

基础版本采用一维的Grid和一维的Block，因为本来输入也是线性的，故在计算坐标时稍微转换一下即可。每个Block内开1024个线程，共开$\lceil W \cdot H/1024 \rceil$个Grid，这里的$W$和$H$分别为输入矩阵的宽和高。 **每个线程都计算某一点的中心熵。**

```
    // Invoke the device function
    kernel<<< divup(size, 1024), 1024 >>>(size, width, height, input_d, output_d);
```

(ii) 二维线程块版本

第二个版本的CUDA程序采用了二维的Grid和Block，通过对原输入矩阵进行tiling操作，即可将原矩阵划分成多个小矩阵。每个Block计算一个小矩阵，而Block内的**每一个线程则对小矩阵内的对应元素计算中心熵**。

这里利用dim3创建二维的Grid和Block，并按照如下方式调用核函数。

```
    // Invoke the device function
    const dim3 grid(divup(width, blockW), divup(height, blockH));
    const dim3 threadBlock(blockW, blockH);
    kernel<<< grid, threadBlock >>>(size, width, height, input_d, output_d);
```

核心代码逻辑依然跟上述的基线版本相同，变化的只是下面的坐标映射部分。完整代码可见sources-2d文件夹。

```
    const int x = blockIdx.x * blockW + threadIdx.x;
    const int y = blockIdx.y * blockH + threadIdx.y;
    const int idx = y * width + x;
```

(iii) 共享内存版本

注意到程序中涉及到大量输入矩阵的读取，计算每个点的中心熵都需要将周围5×5的所有元素读入，这显然是一个很大的开销。前面两个版本的程序都是将输入矩阵放在*全局内存*中，那么可以考虑将部分元素提前读入*共享内存*，以实现加速的读取。

依然采用二维线程块划分的思路，如果对于每个块，将其对应的所有元素迁移到*共享内存*，那似乎就可以达成上述目的。但事实上在计算该块每个元素的中心熵时，往往还需要周围一些块的对应的元素，因此如果仅仅将块内的元素读入到*共享内存*则会发生访问错误的问题，因为边界元素未被读入。故正确的方式应该是将块内及块周围的元素都一并读入，然后再做计算。

这里我采用了padding的方法，比如对于一个16×16的Block，那么开$(16+2\times 2)^2 = 20^2 = 400$个线程，对这个Block周围一圈宽度为2的部分也一并读入共享内存。对于这些线程来说，每个线程只要读取它对应的那个元素到共享内存即可，并在读取完成后利用__syncthreads同步线程。需要考虑两个边界情况：

- 如果线程对应的元素超出原始矩阵的范围，那么直接置0（不做读入）。
- 如果线程对应的元素在原始矩阵的范围内，但超出了该块对应的范围，那么依然需要算出原始矩阵元素的坐标，并读入共享内存。

代码逻辑如下：

```
// true index (x,y)
const int x = blockIdx.x * blockW + threadIdx.x - RADIUS;
const int y = blockIdx.y * blockH + threadIdx.y - RADIUS;
const int idx = y * width + x;
// thread index (tx,ty) (with padding)
const int tx = threadIdx.x;
const int ty = threadIdx.y;
// copy data from global memory to shared memory (with padding)
__shared__ float smem[padH][padW];
if (x >= 0 && x < width && y >= 0 && y < height) {
    smem[ty][tx] = input[idx];
}
__syncthreads();
```

下面的中心熵计算逻辑基本与前面的方法一样，只是读取数据直接从共享内存中读取即可。另外需要判断当前线程是否是在合法的块内（而不是padding的部分），如果是padding的读入线程，那么其在计算环节将会被闲置（主要是任务不好划分，如果将其也加入到计算中，很容易引起负载不均的问题）。由于padding读入的线程数和实际计算的线程数不同，线程对应的ID映射也需要非常小心。如下，Grid的划分数目对应原始块的大小，但Block里的线程数目则是要通过padding后的块大小决定。

```
const dim3 grid(divup(width, blockW), divup(height, blockH));
const dim3 threadBlock(padW, padH);
kernel<<< grid, threadBlock >>>(size, width, height, input_d, output_d);
```

这里只放出与前面的版本相比有改动的部分，完整代码请见sources-shared_mem文件夹。

```
// only those threads in the window need to be calculated
if (x >= 0 && x < width && y >= 0 && y < height &&
    tx >= RADIUS && tx < padW - RADIUS &&
    ty >= RADIUS && ty < padH - RADIUS) {
    // each thread first counts the histogram of idx
    int cnt[16] = {0}; // histogram
```

```
        int valid = 0;
        for (int i = -2; i < 3; ++i)
            for (int j = -2; j < 3; ++j) {
                if (y + i >= 0 && y + i < height &&
                    x + j >= 0 && x + j < width) {
                    int in = smem[ty + i][tx + j];
                    cnt[in]++;
                    valid++;
                }
            }
```

五、 实验设置与结果

1. 环境设置

为了避免占用学院集群环境的大量资源，我采用了另外一台服务器进行测试。该服务器上装备了2个Intel Xeon Gold 5118处理器（24物理核/48逻辑核），另有一块Titan V GPU。操作系统为Ubuntu 18.04 LTS，编译器采用gcc 7.4.0和nvcc 10.2。

下列所有实验均为运行多次取平均的结果。对于OpenMP，统一使用24线程进行计算。为避免**冷启动**带来的性能损耗，我将第一组样例剔除不计入运行时间的统计。 LU代表Lookup Table，即log查找表的实现。

同时，我也在学院集群上进行了简单测试，运行时间均与下述在服务器上的测试差异不大。

2. 实验结果

首先我比较了CPU两个版本（NumPy、OpenMP）和GPU版本的性能，结果如图1所示。可以看到，即使运行时间采用了对数坐标，CPU版本与GPU版本依然有明显的差距。

对于小的样例，OpenMP的性能会明显好于CUDA的性能，这很大一部分原因是GPU程序的运行需要进行数据的搬移，而CPU的程序可以直接读取数据进行计算。但对于大的样例来说（128×128及以上），CUDA的实现总能比OpenMP快很多，差不多可以达到一个数量级的稳定提升。而对比起CPU单核NumPy的实现，CUDA则最多可快上4个数量级。这是由于GPU上有大量的核（成千上万个线程）同时进行计算，而CPU最多也就只有几十个物理核（本实验最多只开了24个线程），因此对于大的任务，也能够有更多的计算资源同时在消耗，进而算得更快。

另外比较奇妙的事情是在CPU的OpenMP实现中采用log查找表可以带来明显的性能提升。这可以从图1的橙色线和绿色线的比较中看出，绿色线（采用查找表）始终在橙色线（不采用查找表）下面，即使用查找表总能带来性能提升。

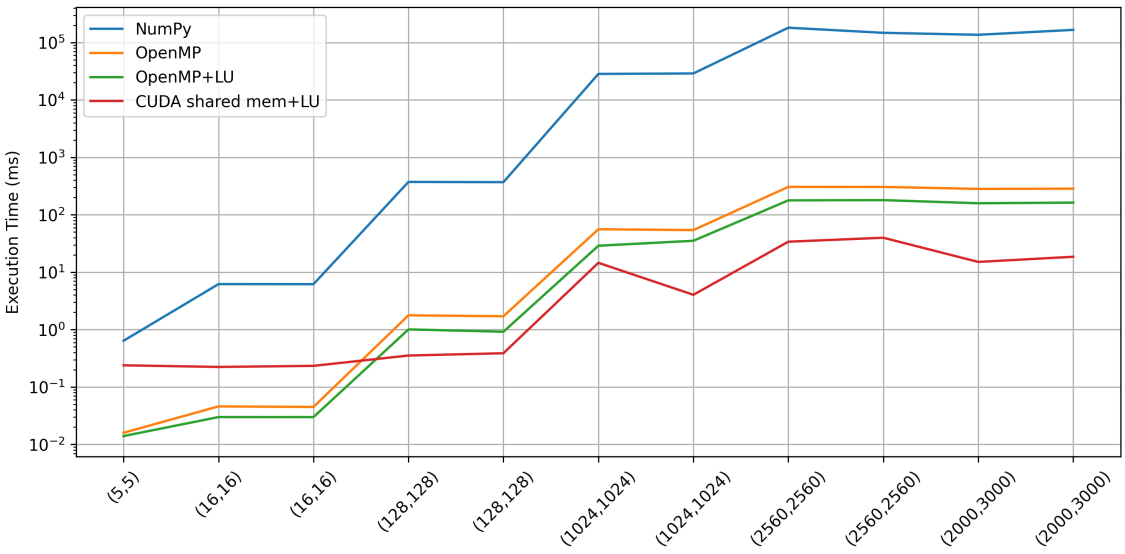


图 1: CPU与GPU比较

下面则着重比较CUDA不同优化版本之间的性能，可见图2的结果。这里采用了普通的坐标，可以看到虽然不同优化版本的曲线相距得比较近，但是相对的性能差异还是比较明显。有以下几点观察：

- 在GPU上采用常量内存存储log查找表并不能带来性能提升，甚至可能出现性能下降的情况。猜测可能的原因是GPU计算对数的速度依然快于常量内存的访问时间，因此查表还不如直接算。另一方面则是因为内存的访问冲突，由于对数表是25个元素的数组，但利用公式1计算时可能涉及到成千上万次的查表操作，这样根据抽屉原理，这个数组中的每个元素都会被大量访问，而访问带宽及bank数目是有限的，进而导致性能损耗。
- 从1D的线程块到2D的线程块，再到使用共享内存，这几个优化的提升是明显的。最好的共享内存版本实现在最大的数据集上达到了17.749ms的速度，相比起基线版本达到了1.6x的提升。之所以没有更明显的加速比，推测是因为数据集还是太小，还未充满GPU的计算能力，任务就已经结束了，所以性能的波动都比较大。

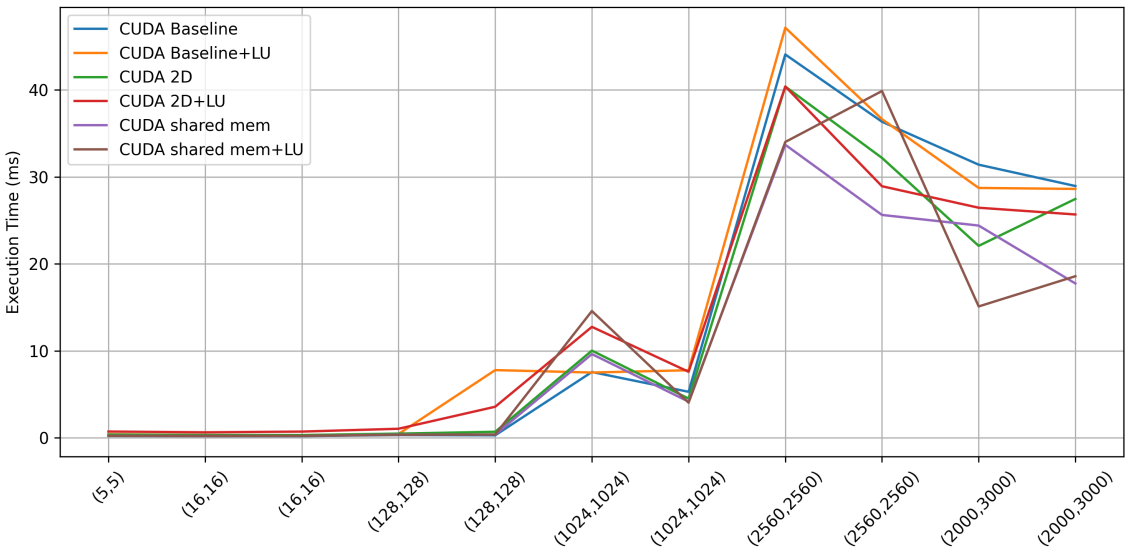


图 2: CUDA不同优化版本比较

最后我也测了不同Block大小对整体性能的影响，实验结果如图3所示。由于实验的GPU最多只能开1024个线程，故这里最大的块大小为28×28，加上padding部分即对应$(28+4)^2 = 1024$个线程。从图中可以看到线程块的大小不能开太大或者太小，只有适中的时候对GPU的性能才最好。比如在本实验中就是16×16的块大小，在大多数样例下都比其他的块大小性能要优。

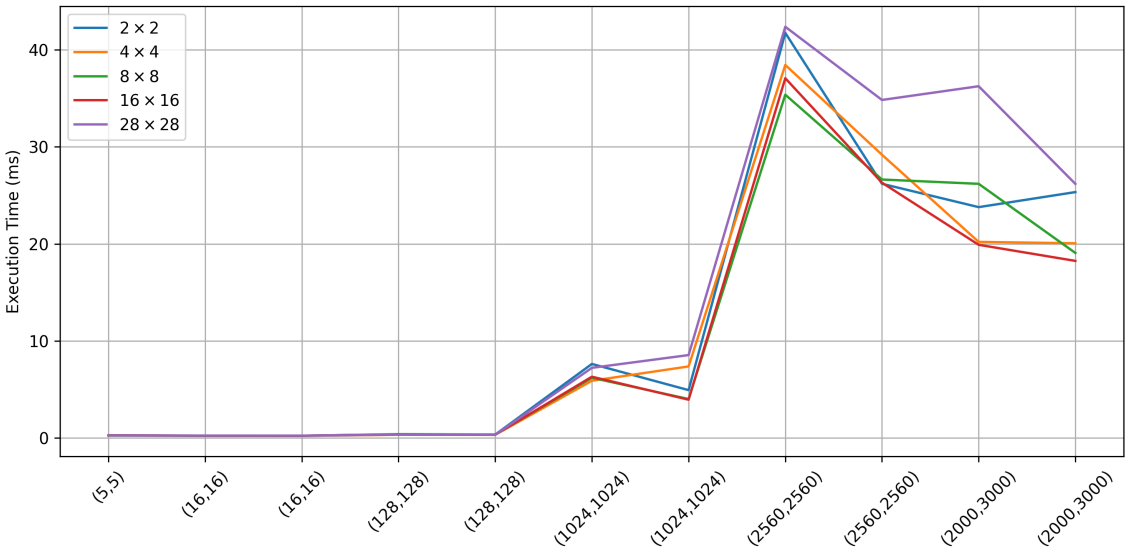


图 3: 不同块大小比较