并行程序设计与算法
作业 1：计算二维数组中心熵

20337025 崔璨明
计算机科学与技术（人工智能与大数据）

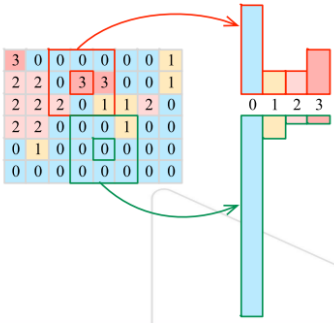
2023 年 6 月 16 日

目录

1 实验题目

计算二维数组中以每个元素为中心的熵（entropy）

- $H(X) = -\sum_i p_i.logp_i$
  - $p_i = p(X = x_i)$
- 信息论的基础概念
- 衡量随机变量分布的“混乱程度”
- 下图计算了红色及绿色的 3 × 3 窗口中的熵值



$$H(X) = -\frac{4}{9}\log\frac{4}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{2}{9}\log\frac{2}{9}$$

$$H(X) = -\frac{8}{9}\log\frac{8}{9} - \frac{1}{9}\log\frac{1}{9}$$

- 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 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）
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）

2 实验原理

中心熵（central entropy）是指以某个中心点为中心，计算窗口内数据的熵。在二维数组中，中心熵可以用于衡量窗口内数据的不确定性或信息量。

对于每个中心点，以其为中心的窗口可以看作是一个局部的子区域。中心熵可以用来描述该窗口内数据的混乱程度或信息的多少。通常情况下，中心熵越高，表示窗口内的数据越分散、不确定性越大；中心熵越低，表示窗口内的数据越集中、信息量越少。

计算中心熵的方法与计算熵的方法类似，都是基于概率的。对于以某个中心点为中心的窗口，我们可以统计窗口内每个元素的频率，并计算每个元素的概率。然后根据概率计算熵，得到中心熵。

中心熵的计算可以用来分析图像、信号处理、模式识别等领域中的局部特征，例如纹理分析、边缘检测等。通过计算不同中心点的中心熵，可以获得窗口内局部特征的变化情况，进而进行图像分割、目标检测等任务。

3 程序逻辑

在本次实验中，我实现了 3 种版本的程序，总共 8 个文件，包括：

- CPU 串行版本的计算二维矩阵中心熵程序及其优化，包括：
  - baseline.cpp
  - baseline_lu.cpp
- 使用 OpenMP 实现的 CPU 并行版本的计算二维矩阵中心熵程序及其优化，包括：
  - openmp.cpp
  - openmp_lu.cpp
- CUDA 版本的 GPU 并行程序以及三个优化版本，包括：
  - cuda_bl.cu: 基线版本
  - cuda_bl_lu.cu：log 运算查表加速的基线版本
  - cuda_2d.cu：2D 线程块优化版本
  - share_mem.cu：共享内存优化版本

接下来将对上述程序依次进行介绍。

3.1 CPU 串行版本

实现该版本的目的在于提供一个基线版本，以和 OpenMp 版本以及 CUDA 版本进行对比，使实验结果和性能提升更加直观。

串行版本实现起来较为容易，首先随机生成不同大小的二维数组，然后调用 calculateEntropy 函数计算每个数组的中心熵。窗口大小设置为 5*5，函数 calculateEntropy 通过统计窗口内各个元素值的出现次数，然后根据这些计数计算熵值。我使用了窗口的起始坐标和结束坐标来确定窗口的大小，通过循环遍历窗口内的元素进行计数和熵值的累加，最后返回计算得到的熵值。

关键代码和详细注释如下：

```cpp
void generateRandomArray(std::vector<std::vector<int>>& array, int size)
{
    srand(static_cast<unsigned int>(1234)); // 设置随机种子
    array.resize(size, std::vector<int>(size)); // 调整数组大小为指定大小
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++)
            array[i][j] = rand() % ELEMENT_RANGE; // 生成随机数并赋值给数组元素
}

// 计算熵
double calculateEntropy(const std::vector<std::vector<int>>& array, int x, int y)
{
    // 统计窗口内每个元素值的出现次数
    std::vector<int> counts(ELEMENT_RANGE, 0);
    // 计算窗口的起始坐标,结束坐标
    int startX = std::max(0, x - WINDOW_SIZE / 2);
    int startY = std::max(0, y - WINDOW_SIZE / 2);
    int endX = std::min(static_cast<int>(array.size()) - 1, x + WINDOW_SIZE / 2);
    int endY = std::min(static_cast<int>(array.size()) - 1, y + WINDOW_SIZE / 2);
    // 统计窗口内每个元素值的出现次数
    for (int i = startX; i <= endX; i++)
        for (int j = startY; j <= endY; j++)
            counts[array[i][j]]++;
    double entropy = 0.0; // 熵值
    // 窗口大小
    int windowSize = (endX - startX + 1) * (endY - startY + 1);
    // 计算熵值
    for (int i = 0; i < ELEMENT_RANGE; i++) {
        // 计算元素值在窗口中的概率
        double probability = double(counts[i]) / windowSize;
        if (counts[i] != 0) {
            // 使用对数的换底公式计算熵值
            entropy -= probability * log2(probability);
        }
    }
    return entropy; // 返回计算得到的熵值
```

```
}
```

对于对数运算范围有限的情况，使用查表可以加速运算过程，特别是当计算的对数值的范围较小的情况。改进上述函数，可以通过查表方式来计算对数值，而不是直接调用 log2f 函数。只需添加以下函数：

```
const int LOG_TABLE_SIZE = 25;
std::vector<double> logTable(LOG_TABLE_SIZE);
// 初始化对数表
void initializeLogTable(){
    for (int i = 1; i <= LOG_TABLE_SIZE; i++)
        logTable[i - 1] = log2(i);
}
// 查找对数值
double lookupLog(int n){
    if (n >= 1 && n <= LOG_TABLE_SIZE)
        return logTable[n - 1];
    else
        // 处理超出查表范围的情况
        return log2(n);
}
```

再在 calculateEntropy 函数中将 log2 函数修改成查表取值即可。

3.2 CPU 并行版本

只需要在 CPU 串行版本的代码上进行少量修改即可，直接使用 OpenMP 的并行计算指令 pragma omp parallel for 将计算中心熵的 for 循环并行化，使得每个迭代可以在多个线程上并行执行：

```
#pragma omp parallel for num_threads(24)
for (int i=0;i<sizes.size();i++) {
    // 生成随机二维数组
    int size=sizes[i];
    std::vector<std::vector<int> > array(size, std::vector<int>(size));
    generateRandomArray(array,size);
    clock_t start, finish;
    start = clock();
    // 计算熵
    std::vector<std::vector<double> > entropyArray(size, std::vector<
        double>(size));
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            entropyArray[i][j] = calculateEntropy(array, i, j);
        }
    }
```

```
    finish=clock();
    // 输出结果
    std::cout << "Array Size: " << size << " using time: "<<1000*double
        (finish - start) / CLOCKS_PER_SEC<<" ms"<<std::endl;
}
```

同样也可以使用查表的方式来加速对数运算，在此就不在赘述。

3.3 CUDA 实现

CUDA 允许我们将任务划分为多个并行执行的线程。每个线程在 GPU 上执行相同的指令，但处理不同的数据。这样可以同时处理多个数据，充分利用 GPU 的并行处理能力。

线程块和网格：CUDA 使用线程块（thread block）和网格（grid）的概念来组织并行执行的线程。线程块是一组线程的集合，这些线程可以协同工作并访问共享内存。网格是线程块的集合，用于管理大量的线程。

3.3.1 基线版本 cuda_bl.cu

在 CUDA 的实现中，需要定义 CUDA 的网格和块大小，并用 ___global___ 来声明核函数，计算以每个元素为中心的窗口中的熵，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。

其他实现和 CPU 版本的实现大致相同，在这个基线版本中，我使用了一维的 Grid 和一维的 Block，每个线程计算的是以输入数组中的每个元素为中心的窗口的熵。具体来说，CUDA 核函数 calculateEntropy 中的每个线程都有一个唯一的索引 index，通过以下计算得到：

```
int index = blockIdx.x * blockDim.x + threadIdx.x;
```

然后，每个线程使用该索引计算出对应的行和列：

```
int row = index / size;
int col = index % size;
```

接下来，每个线程确定窗口的起始行、起始列、结束行和结束列，并进行边界处理，以确保窗口不超出输入数组的范围。然后，线程计算窗口内元素的频率，即统计窗口内每个元素的出现次数。最后，线程根据频率计算窗口的熵值，并将结果存储在输出数组中。

关键代码如下：

```
// CUDA核函数，计算以每个元素为中心的窗口中的熵
__global__ void calculateEntropy(int* input, float* output, int size)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < size * size)
    {
        int row = index / size;
        int col = index % size;

        int windowSize = 5;
        int windowStartRow = row - windowSize / 2;
```

```
12          int windowStartCol = col - windowSize / 2;
13          int windowEndRow = windowStartRow + windowSize;
14          int windowEndCol = windowStartCol + windowSize;
15
16          // 边界处理
17          if (windowStartRow < 0)windowStartRow = 0;
18          if (windowStartCol < 0)windowStartCol = 0;
19          if (windowEndRow >= size)windowEndRow = size - 1;
20          if (windowEndCol >= size)windowEndCol = size - 1;
21
22          float entropy = 0.0f;
23          int windowElements = (windowEndRow - windowStartRow + 1) * (
                windowEndCol - windowStartCol + 1);
24
25          // 计算窗口内元素的频率
26          int frequency[16] = { 0 };
27          for (int i = windowStartRow; i <= windowEndRow; i++)
28          {
29              for (int j = windowStartCol; j <= windowEndCol; j++)
30              {
31                  int value = input[i * size + j];
32                  frequency[value]++;
33              }
34          }
35
36          // 计算熵
37          for (int k = 0; k < 16; k++)
38          {
39              float prob = static_cast<float>(frequency[k]) /
                    windowElements;
40              if (prob > 0.0f)
41                  entropy -= prob * log2f(prob);
42          }
43
44          output[index] = entropy;
45      }
46  }
```

每个线程块的任务：每个线程块负责处理一个窗口内的元素，窗口的大小由 windowSize 确定，固定为 5。每个线程块处理一个窗口，通过计算窗口的起始行、起始列、结束行和结束列来确定窗口的位置。线程块的数量由 gridSize 确定，计算方法为 (size * size + BLOCK_SIZE - 1) / BLOCK_SIZE，保证覆盖整个输入数组。

每个线程的任务：每个线程负责处理窗口内的一个元素，线程的索引由 index = blockIdx.x *

blockDim.x + threadIdx.x 计算得到；判断线程的索引是否在数组范围内，如果超出则不执行任务；根据线程的索引计算元素在输入数组中的位置，得到行和列的值；执行边界处理，确保窗口不会超出输入数组的范围；计算熵并将计算结果存储到输出数组中的对应位置。

3.3.2 查表加速对数运算

同样可以用查表的方式来进行加速，但实现方法有点不一样，需要直接在 main 函数中预计算对数表，然后作为参数传入 CUDA 核函数中，具体代码如下：

```
// 预计算对数表
float logTable[25];
for (int k = 0; k < 25; k++)
{
    logTable[k] = log2f(static_cast<float>(k + 1));
    //printf("%f ",logTable[k]);
}
// 调用CUDA核函数
calculateEntropy<<<gridSize, blockSize>>>(deviceInput, deviceOutput,
    size, logTable);
```

3.3.3 二维线程块优化

基线版本的程序使用的是一维的 Grid 和一维的 Block，现在将其改进为二维的 Grid 和 Block，使用二维的线程索引（如 blockIdx.x、blockIdx.y、threadIdx.x、threadIdx.y）计算出每个线程的行和列索引，以确定对应的窗口区域。行和列索引直接映射到二维数组的行和列。二维的 Grid 和 Block 在索引计算和数据访问上更直观和方便，可以更自然地处理二维问题：

```
// 定义CUDA的网格和块大小
    dim3 gridSize((size + BLOCK_SIZE - 1) / BLOCK_SIZE, (size +
        BLOCK_SIZE - 1) / BLOCK_SIZE);
    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
```

核函数代码如下，省略了相同的部分：

```
// CUDA核函数，计算以每个元素为中心的窗口中的熵
__global__ void calculateEntropy(int* input, float* output, int size)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size)
    {
        ...
    }
}
```

每个线程块的任务：负责处理一个窗口内的元素，线程块的索引由 blockIdx.x 和 blockIdx.y 确定。线程块的数量由 gridSize 确定，其中 gridSize 是一个 dim3 类型的变量，表示网格的大小。

每个线程的任务：每个线程负责处理窗口内的一个元素。线程的索引由 threadIdx.x 和 threadIdx.y 确定，计算元素在输入数组中的位置，得到行和列的值；判断行和列是否在数组范围内，如果超出则不执行任务；根据窗口大小和行列值计算窗口的起始行、起始列、结束行和结束列；计算熵并将计算结果存储到输出数组中的对应位置。

3.3.4 共享内存优化

以采用二维的 Grid 和 Block 的程序为例进行共享内存的优化，在 CUDA 核函数 calculateEntropy 中，声明一个共享内存数组 sharedInput，用于存储输入数据的子集。在每个线程块中，每个线程都会将输入数据的一部分复制到共享内存中。这样做的好处是，共享内存的访问速度比全局内存更快，可以提高数据访问的效率。

接下来，通过使用 ___syncthreads() 函数来确保所有线程都完成了数据拷贝到共享内存的操作，以保证数据的一致性和同步。

然后，在计算窗口内元素的频率时，线程块中的每个线程都会访问共享内存中的数据，而不是访问全局内存。这样可以避免大量的全局内存访问，减少数据访问的延迟。

最后，在计算熵时，频率数组 frequency 也存储在共享内存中，线程块中的每个线程可以直接访问共享内存中的频率数组。这样可以加快频率数组的访问速度，提高计算熵的效率。

通过使用共享内存，可以减少对全局内存的访问次数，并且可以充分利用线程块内的数据重用和数据局部性。这样，可以提高数据访问效率，加速程序的执行。

关键代码如下：

```
// CUDA核函数，计算以每个元素为中心的窗口中的熵
__global__ void calculateEntropy(int* input, float* output, int size)
{
    // 声明共享内存数组
    __shared__ int sharedInput[BLOCK_SIZE][BLOCK_SIZE];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    // 将输入数据拷贝到共享内存
    if (row < size && col < size)
    {
        sharedInput[threadIdx.y][threadIdx.x] = input[row * size + col
            ];
    }
    // 确保所有线程都完成数据拷贝
    __syncthreads();
    if (row < size && col < size)
    {
        int windowSize = 5;
        int windowStartRow = row - windowSize / 2;
        int windowStartCol = col - windowSize / 2;
        int windowEndRow = windowStartRow + windowSize;
```

```
21          int windowEndCol = windowStartCol + windowSize;
22          // 边界处理
23          if (windowStartRow < 0) windowStartRow = 0;
24          if (windowStartCol < 0) windowStartCol = 0;
25          if (windowEndRow >= size) windowEndRow = size - 1;
26          if (windowEndCol >= size) windowEndCol = size - 1;
27
28          float entropy = 0.0f;
29          int windowElements = (windowEndRow - windowStartRow + 1) * (
                windowEndCol - windowStartCol + 1);
30          // 计算窗口内元素的频率
31          int frequency[16] = { 0 };
32          for (int i = windowStartRow; i <= windowEndRow; i++)
33          {
34              for (int j = windowStartCol; j <= windowEndCol; j++)
35              {
36                  int value = sharedInput[i - row + threadIdx.y][j - col
                        + threadIdx.x];
37                  frequency[value]++;
38              }
39          }
40          // 计算熵
41          for (int k = 0; k < 16; k++)
42          {
43              float prob = static_cast<float>(frequency[k]) /
                    windowElements;
44              if (prob > 0.0f)
45                  entropy -= prob * log2f(prob);
46          }
47          output[row * size + col] = entropy;
48      }
49  }
```

4 实验结果

4.1 实验环境

在一个服务器上进行实验，该服务器一些配置如下：

- CPU 为 Intel(R) Xeon(R) CPU E5-2630
- CPU 核心数：20 个物理核，40 个逻辑核
- GPU：NVIDIA TITAN Xp

- 编译器：gcc 5.5.0，nvcc 10.1
- 系统版本：Ubuntu 16.04

4.2 实验结果

首先进行正确性测试，验证程序运行的正确性，读入助教给的 test1.in 文件和 test2.in 文件，并将自己输出的熵矩阵与 test1.out 和 test2.out 中的数据进行对比，可知结果正确。然后按照实验要求，对每个程序，都生成了输入和输出文件，详见 output 文件夹，若要验证程序的正确性，则可以按照指示修改路径，运行程序。

```
Array Size: 5 using time: 0.034 ms
2.94770 3.08496 3.37356 3.08496 3.16993
3.02206 3.32782 3.58418 3.37500 3.41830
3.32323 3.44644 3.54347 3.30869 3.37356
3.08496 3.20282 3.34644 3.20282 3.25163
2.94770 2.85539 3.05656 2.85539 2.72548
```

所有实验均为运行多次取平均的结果，每次实验时每个程序都采用相同的随机数种子以保证测试的矩阵相同，后缀 lu 表示查表加速，bl 表示基线版本。

4.2.1 CPU 版本和 GPU 版本

首先是 CPU 版本和 GPU 版本的比较，这里采用对数坐标轴，这意味着纵坐标上的值是以对数形式显示的，从而可以更好地展示不同算法之间的时间差异。如图 1 所示。

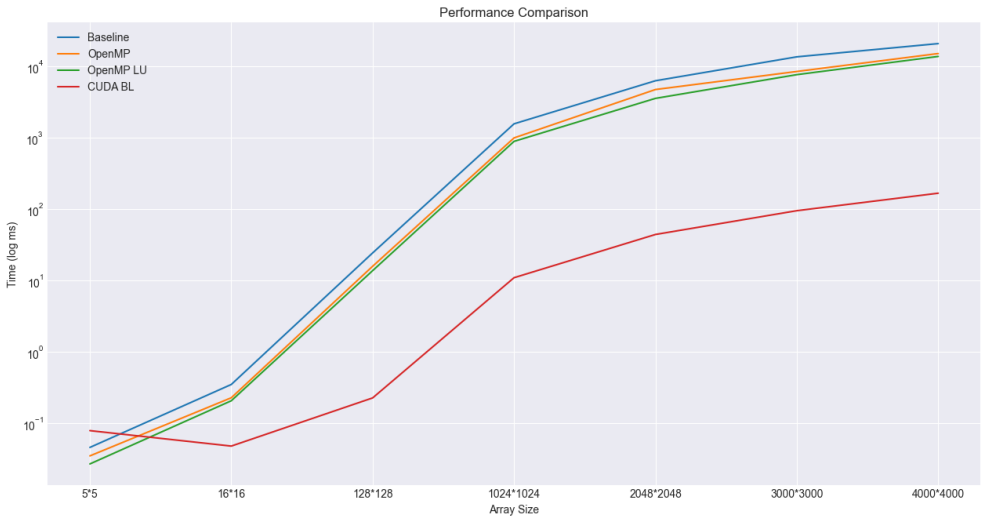


图 1: CPU 版本和 GPU 版本的比较

- 可以看到对于规模较小的样例，OpenMP 的性能要优于 CUDA。这主要是因为 GPU 程序需要进行数据传输，而 CPU 程序可以直接访问数据进行计算。对于规模较大的样例，CUDA 的实现始终比 OpenMP 快一两个数量级。这是因为 GPU 上有大量的核心可以并行计算，而 CPU 只有几十个物理核心（在本实验中使用了 40 个线程）。

- 与串行的 CPU 实现相比，使用 OpenMP 实现的 CPU 并行版本要快一点。
- 在 OpenMP 实现的 CPU 并行版本中利用对数查表加速，可以看到性能有所提升。

4.2.2 GPU 版本和优化

接着是 CUDA 实现的 GPU 基线版本和 3 个优化之间的比较（查表加速、二维块、共享内存），如图 2 所示。

可以看到不同优化版本之间还是有相对的性能差异：

- 在性能上，采用共享内存的版本表现最好，这是因为共享内存位于芯片上，与计算单元距离较近，可以通过高速缓存进行快速访问。这样，当多个线程需要访问相同的数据时，它们可以从共享内存中快速获取，而无需每次都从全局内存中读取。
- 可以看到查表加速版本比 GPU 基线版本的性能略好，但差异不明显。
- 改进为二维的 Grid 和 Block 后，程序的性能有所提升，起到了一定的优化作用。
- 从曲线变化的趋势来看，若矩阵规模再大的话，共享内存的优化带来的性能提升会更加明显。

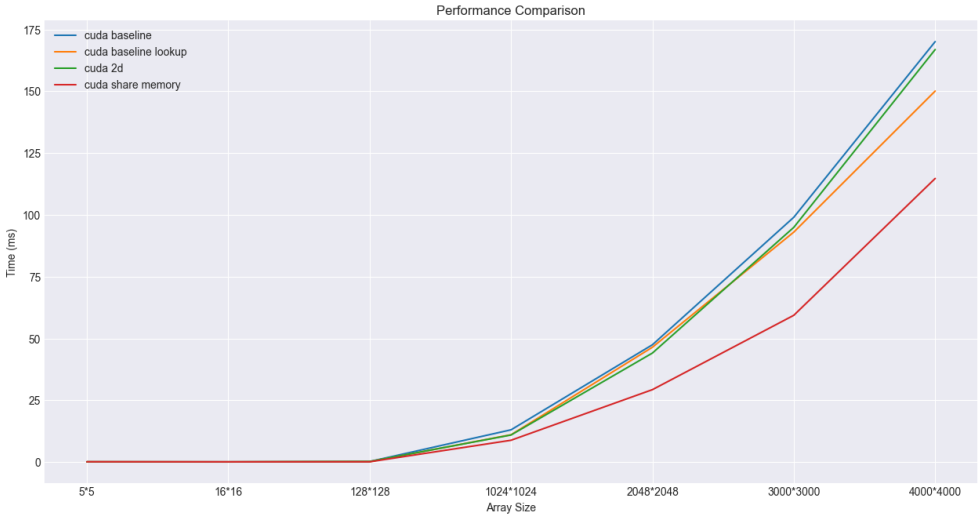


图 2: GPU 版本和优化的比较

4.2.3 不同大小的 Block 的影响

在上述的实验中，我使用的 BLOCK_SIZE 为 16，为了探究不同大小的 block 对程序性能产生的影响，我以共享内存版本的程序为例，设置不同的 BLOCK_SIZE，比较其运行时间，得到结果如图 3 所示。

可以看到线程块的大小不能开太大或者太小，需要通过多次实验选取一个适中的值，只有这样才能充分发挥 GPU 的性能。比如在本实验的采用共享内存的 CUDA 程序中中就是 8×8 的块大小表现最好，在大多数样例下都比其他的块大小性能要优。

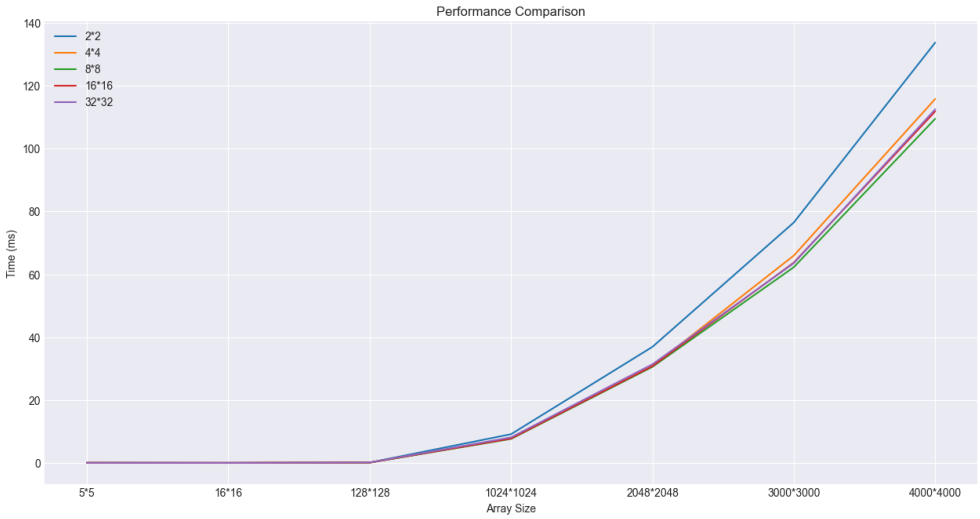


图 3: 不同大小的 Block 的比较

5 问题回答

1. 见第 3 部分

2. 程序涉及的存储器类型有全局内存、共享内存和常量内存。

   使用全局内存存储输入和输出数组，因为全局内存具有较大的容量，适用于存储整个数组。对于频繁访问的输入数组，可以将其复制到共享内存中，以提高存储访问性能。对数表部分使用常量内存，因为对数表在程序运行过程中不会被修改，可以在 GPU 的常量内存中预先计算并存储，以提高访问效率。

3. 设窗口大小为 M×M，M = 5，设窗口内元素数目为 N，每个元素值的数目为 Ni(i = 0, . . . , 15)，则中心熵的计算公式：

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

   进行变换可得

   $$H(X) = \sum_{i=0}^{15} -\frac{N_i}{N} (logN_i - logN)$$

   因为 $N_{max} = M \times M = 25$，所以程序中的对数运算实际只涉及对整数 [1,25] 的对数运算。由 4.2 中的实验结果可知，如使用查表对 log1 log25 进行查表，可以加速运算过程。

4. 见 3.3 和 4.2。

5. 实验结果及每个实验的分析见 4.2。总而言之，影响 CUDA 程序性能的因素有：

   - 内存访问
   - 数据存储的位置
   - 线程块的大小

内存访问是影响 CUDA 程序性能的最重要因素之一。数据存放的位置，无论是全局内存、共享内存还是常量内存，对整体性能都有巨大影响。但不是说数据存放在共享内存就一定更快，数据访问模式往往与后续的算法计算方式密切相关。只有在计算和存储之间相互配合的情况下，才能发挥硬件的最大效能。

此外，线程块的大小也对性能有显著影响，需要充分利用线程并实现负载均衡。同时，应避免由于线程数与任务数无法完全整除而导致的空闲线程。只有合理划分任务，才能实现高性能。

因此，在优化 CUDA 程序时，需要综合考虑内存访问方式、数据存放位置以及线程块的大小。只有在充分利用硬件资源的同时，合理地设计数据访问模式和任务划分，才能实现高性能的计算。。

6. OpenMP 实现的程序见 3.2，与 CUDA 版本进行对比见 4.2.1。

6 实验总结

6.1 遇到的困难

遇到的困难主要有对 CUDA 编程不熟悉、没有足够的资源进行实验等。但通过收集资料、借助服务器等方法都得到了解决，而在解决问题的过程中也锻炼了我的能力。

6.2 实验感想

通过完成这个实验，我深入理解了 CUDA 编程中的内存访问模式和存储器选择的重要性，也及时复习了课上所学到的理论知识。优化程序性能需要综合考虑多个因素，并通过实验数据进行评估和分析。此外，与其他并行编程模型的比较也可以帮助我们更好地理解 CUDA 的优势和局限性。