多核程序设计与实践

作业二：计算二维数组的中心熵

中山大学计算机学院　计算机科学与技术

19335174　施天予

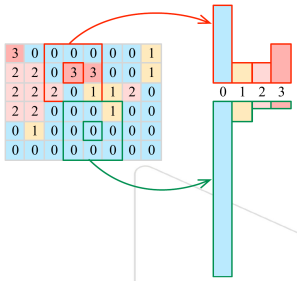
目录

一、题目要求

计算二维数组中以每个元素为中心的熵 (entropy):

- $H(X) = -\sum_i p_i \cdot logp_i$

  $p_i = p(X = x_i)$
- 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 窗口的熵值



$$H(X) = -\frac{5}{9}\log\frac{5}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{1}{9}\log\frac{1}{9} - \frac{2}{9}\log\frac{2}{9}$$

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

- 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)

二、程序逻辑

1. 文件组织

本次实验共有 5 个代码文件，各文件内容如下：

- utils.h：包括 CUDA 调用的检查宏 CHECK，随机初始化矩阵函数 Generate，检查计算结果函数 Evaluate
- kernel.cu：CUDA 调用的各种优化版本的核函数
- main_cuda.cu：CUDA 主函数
- main_omp.cpp：OpenMP 主函数及实现
- main.cu：老师提供的部分代码（我将代码移植入该文件，仅用于验证两个给定矩阵输出的正确性，性能测试由上面 4 个文件完成）

2. 通用函数

随机初始化 $m \times n$ 的二维矩阵，随机数设置为 0-15 的整型数。

```
void Generate(int **in, float **out, int m, int n) {
    *in = new int[m*n], *out = new float [m*n];
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j)
            (*in)[i*n+j] = rand() % 16;
}
```

Evaluate 函数用于检验计算结果的正确性，也可以看作是一个 CPU 串行版本。

```
void Evaluate(int *in, float *out, int m, int n) {
    for (int i = 0; i < m; ++i)
        for (int j = 0; j < n; ++j) {
            int cnt[16] = {0}, count = 0;
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = i + dx;
                if (x >= 0 && x < m) {
                    for (int dy = -2; dy <= 2; ++dy) {
                        const int y = j + dy;
                        if (y >= 0 && y < n) {
                            ++cnt[in[x*n+y]];
                            ++count;
                        }
                    }
                }
            }
            double ans = 0;
            for (int k = 0; k < 16; ++k)
                if (cnt[k])
```

```
                ans -= (double)cnt[k] * (1.0 / count) * log2f((float)cnt[k]/count
                    ↪ );
            if (fabs(out[i*n+j] - ans) > 1e-5) {
                printf("Computation Error In %d Row %d Col!\n", i, j);
                return;
            }
        }
    printf("Computation Correct!\n");
}
```

3. CUDA 实现

(i) 参数设置

从命令行读取矩阵大小和线程块大小，以及我们使用 CUDA 计算的核函数版本（核函数设计详见 Section 3，main.cu 使用给定的标准输入输出）。

```
    int *in, *d_in;
    float *out, *d_out, *d_log;
    // 随机初始化矩阵
    int m = strtol(argv[1], NULL, 10);
    int n = strtol(argv[2], NULL, 10);
    Generate(&in, &out, m, n);
    // 线程块大小
    int b1 = strtol(argv[3], NULL, 10);
    int b2 = strtol(argv[4], NULL, 10);
    // CUDA优化版本号0-7
    int v = strtol(argv[5], NULL, 10);
```

(ii) 显存分配与数据拷贝

为输出矩阵 in 和输出矩阵 out 分配显存 d_in 和 d_out，pre_log 是预处理的对数表，可用常量内存、纹理内存等不同方式存储，pre_cal 是用对数表的进一步优化计算。在数据拷贝时，使用预定义的 CHECK 宏检查错误。另外，根据命令行读取的线程块大小 b1 和 b2，我们可以计算出 blockSize 和 gridSize。

```
    // 显存分配
    CHECK(cudaMalloc((void**)&d_in, m * n * sizeof(int)));
    CHECK(cudaMalloc((void**)&d_out, m * n * sizeof(float)));
    CHECK(cudaMalloc((void**)&d_log, sizeof(pre_log)));
    // 数据拷贝
    CHECK(cudaMemcpy((void*)d_in, (void*)in, m * n * sizeof(int),
        ↪ cudaMemcpyHostToDevice));
    CHECK(cudaMemcpy((void*)d_log, (void*)pre_log, sizeof(pre_log),
        ↪ cudaMemcpyHostToDevice));
    CHECK(cudaMemcpyToSymbol(const_log, (const float*)pre_log, sizeof(pre_log)));
```

```
CHECK(cudaBindTexture(0, texture_log, d_log, sizeof(pre_log)));
// blockSize和gridSize
dim3 blockSize(b1, b2);
dim3 gridSize(ceil((float) n / blockSize.x),
              ceil((float) m / blockSize.y));
```

(iii) 核函数调用

根据命令行获取的版本号 v 调用对应的核函数。

```
switch (v) {
    case 0:
        printf("Baseline: \n");
        cudaEventRecord(t1, 0);
        v0_baseline <<<gridSize, blockSize>>> (d_in, d_out, m, n);
        cudaEventRecord(t2, 0);
        break;
    case 1:
        printf("Unsigned Char\n");
        cudaEventRecord(t1, 0);
        v1_char <<<gridSize, blockSize>>> (d_in, d_out, m, n);
        cudaEventRecord(t2, 0);
        break;
    case 2:
        printf("Log Table in Register: \n");
        cudaEventRecord(t1, 0);
        v2_registerTable <<<gridSize, blockSize>>> (d_in, d_out, m, n);
        cudaEventRecord(t2, 0);
        break;
    case 3:
        printf("Log Table in Global Memory: \n");
        cudaEventRecord(t1, 0);
        v3_globalTable <<<gridSize, blockSize>>> (d_in, d_out, m, n, d_log);
        cudaEventRecord(t2, 0);
        break;
    case 4:
        printf("Log Table in Texture Memory: \n");
        cudaEventRecord(t1, 0);
        v4_textureTable <<<gridSize, blockSize>>> (d_in, d_out, m, n);
        cudaEventRecord(t2, 0);
        break;
    case 5:
        printf("Log Table in Constant Memory: \n");
        cudaEventRecord(t1, 0);
        v5_constTable <<<gridSize, blockSize>>> (d_in, d_out, m, n);
        cudaEventRecord(t2, 0);
```

```
            break;
        case 6:
            printf("Log Table in Shared Memory: \n");
            cudaEventRecord(t1, 0);
            v6_sharedTable <<<gridSize, blockSize>> >(d_in, d_out, m, n);
            cudaEventRecord(t2, 0);
            break;
        case 7:
            printf("Optimal: \n");
            cudaEventRecord(t1, 0);
            v7_optimal <<<gridSize, blockSize, b1 * b2 * 16 * sizeof(unsigned char)
                ↪ >>> (d_in, d_out, m, n);
            cudaEventRecord(t2, 0);
            break;
        default:
            cudaEventRecord(t1, 0);
            cudaEventRecord(t2, 0);
            break;
    }
```

(iv) 释放显存

最后将结果返回 host 端并释放显存，用 Evaluate 函数检验正确性。

```
    // 拷贝回host并释放显存
    CHECK(cudaUnbindTexture(texture_log));
    CHECK(cudaMemcpy((void*)out, (void*)d_out, m * n * sizeof(float),
        ↪ cudaMemcpyDeviceToHost));
    CHECK(cudaFree(d_in));
    CHECK(cudaFree(d_out));
    CHECK(cudaFree(d_log));
    // 检验正确性
    Evaluate(in, out, m, n);
```

4. OpenMP 实现

OpenMP 版本使用 thread_num 个线程并行外层 for 循环，并使用预处理的对数表加速计算（便于对比 CUDA 最优的版本）。

```
    // 开始时间
    double start = omp_get_wtime();
    // OpenMP计算
    int cnt[16];
#   pragma omp parallel for num_threads(thread_num) private(cnt)
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j) {
                int count = 0;
```

```
            memset(cnt, 0, sizeof(cnt));
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = i + dx;
                if (x >= 0 && x < m) {
                    for (int dy = -2; dy <= 2; ++dy) {
                        const int y = j + dy;
                        if (y >= 0 && y < n) {
                            ++cnt[in[x*n+y]];
                            ++count;
                        }
                    }
                }
            }
            double ans = 0;
            for (int k = 0; k < 16; ++k)
                if (cnt[k])
                    ans -= (double)cnt[k] * (1.0 / count) * (pre_log[cnt[k]]-
                        ↪ pre_log[count]);
            out[i*n+j] = ans;
        }
    // 输出运行时间
    double end = omp_get_wtime();
    printf("Time cost (OpenMP): %.3f ms\n", (end - start) * 1000);
    // 检验正确性
    Evaluate(in, out, m, n);
```

三、核函数设计

1. V0：Baseline 版本

Baseline 版本的实现比较常规，对于核函数的任务分配，我们让每个线程计算二维矩阵中一个位置所对应的信息熵，使有效范围内的每个线程与二维矩阵元素一一对应。

我们用 idx 和 idy 对应二位线程块 x 方向和 y 方向的索引。cnt 代表 0-15 不同数在该元素邻域的计数器，count 代表该元素领域的元素个数（≤ 25）。通过对 cnt 和 count 的计算，我们可以得出用一个线程计算一个元素的信息熵，并将其值赋予 out 矩阵。另外，这里需要注意要用 if (cnt[k]) 判别当 0-15 元素的个数为 0 时不用计算，如果不考虑的话浮点数 float 对于 0 的计算可能会出现 nan 的情况。

```
__global__ void v0_baseline(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
```

```
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * log2f((float)cnt[k]/count);
        out[idy*n+idx] = ans;
    }
}
```

2. V1：用 unsigned char 代替 int 做计数器

由于二维信息熵的每个 domain 大小仅为 25 ，所以计数器数组 cnt 以及计数器变量 count 用 int 类型变量会浪费寄存器资源。unsigned char 可以表示 0-255 的范围，相对于 25 来说已经够用，并且能讲将内存空间降低到 $\frac{1}{4}$，大大节约寄存器占用。

```
__global__ void v1_char(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned char cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * log2f((float)cnt[k]/count);
        out[idy*n+idx] = ans;
    }
}
```

3. V2：预处理对数表至寄存器

由于单个元素的最大出现次数为 25，所以我们可以将对数 log2(0)–log2(25) 预先计算。为方便计算，所以我们可以将 log2(0) 设置为 0。我们先将对数表存入寄存器中，然而对于每个线程，对数表需要占用 $26 \times 4 = 104$ 个字节的寄存器空间。因此寄存器会发生资源紧张，将对数表放入延迟较高的本地内存，导致程序性能降低。

```
__global__ void v2_registerTable(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned char cnt[16] = {0}, count = 0;
    const float pre_log[26] = { 0.0, log2f(1.0), log2f(2.0), log2f(3.0), log2f(4.0)
        ↪ , log2f(5.0),
                                log2f(6.0), log2f(7.0), log2f(8.0), log2f(9.0), log2f
                                    ↪ (10.0),
                                log2f(11.0), log2f(12.0), log2f(13.0), log2f(14.0),
                                    ↪ log2f(15.0),
                                log2f(16.0), log2f(17.0), log2f(18.0), log2f(19.0),
                                    ↪ log2f(20.0),
                                log2f(21.0), log2f(22.0), log2f(23.0), log2f(24.0),
                                    ↪ log2f(25.0)};
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * (pre_log[cnt[k]]-pre_log[
                    ↪ count]);
        out[idy*n+idx] = ans;
    }
}
```

4. V3：预处理对数表至全局内存

将对数表在 GPU 上分配相应内存空间，从 host 端拷贝到 device 端，就能使用全局内存访问对数表。全局内存的使用比较常规，因此我们先用全局内存进行尝试。

```
__global__ void v3_globalTable(int *in, float *out, int m, int n, float *
    ↪ global_log) {
```

```
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned char cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * (global_log[cnt[k]]-
                    ↪ global_log[count]);
        out[idy*n+idx] = ans;
    }
}
```

5. V4：预处理对数表至纹理内存

纹理内存的缓存具有高维空间局部性，然而对数表不大所以可能效果不明显。使用纹理内存存储对数表，需要先分配内存空间，绑定纹理，在调用核函数后解除绑定，并释放内存。我们使用一维纹理内存存储对数表。

```
texture<float, 1> texture_log;
__global__ void v4_textureTable(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned char cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * (tex1Dfetch(texture_log, cnt[
```

```
                ↪ k])-tex1Dfetch(texture_log, count));
        out[idy*n+idx] = ans;
    }
}
```

6. V5：预处理对数表至常量内存

首先声明常量内存存储对数表。常量缓存的访问延迟较低，并且由于我们的对数表不是很大，所以我们读取相应的常量缓存来得到对数值来说非常合适。

```
__constant__ float const_log[26];
__global__ void v5_constTable(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    unsigned char cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * (const_log[cnt[k]]-const_log[
                    ↪ count]);
        out[idy*n+idx] = ans;
    }
}
```

7. V6：预处理对数表至共享内存

将对数表存至共享内存数组中，需要进行块内同步保证计算正确性。使用共享内存时，我们需要进行块内的线程同步，而由于不同的 warp 是异步执行的，所以进行块内同步会造成一定开销，从而影响性能。

```
__global__ void v6_sharedTable(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int tid = threadIdx.y * blockDim.x + threadIdx.x;
    __shared__ float shared_log[26];
    if (tid != 0 && tid < 26)
        shared_log[tid] = log2f((float)tid);
```

```
    __syncthreads();
    unsigned char cnt[16] = {0}, count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[in[y*n+x]];
                    ++count;
                }
            }
        float ans = 0;
        for (int k = 0; k < 16; ++k)
            if (cnt[k])
                ans -= (float)cnt[k] * (1.0 / count) * (shared_log[cnt[k]]-
                    ↪ shared_log[count]);
        out[idy*n+idx] = ans;
    }
}
```

8. V7：多重优化

虽然在前几种存储对数表的方式中性能都相差无几，但相对来说常量内存性能最高，所以我们将对数表和 k 的乘积存入常量内存。

为避免寄存器紧张，我们也可以将计数器 cnt 用一个共享内存数组代替，这样可以避免寄存器资源不足导致本地内存的使用以及占用率过低的问题。另外，在求和时使用 if 控制语句会降低程序的性能，所以我们也可以用条件运算符 ? : 来进一步提升性能。

```
__global__ void v7_optimal(int *in, float *out, int m, int n) {
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const int idy = blockIdx.y * blockDim.y + threadIdx.y;
    int tid = threadIdx.y * blockDim.x + threadIdx.x;
    extern __shared__ unsigned char cnt[][16];
    memset(cnt + tid, 0, 16 * sizeof(unsigned char));
    __syncthreads();
    unsigned char count = 0;
    if (idy < m && idx < n) {
        for (int dy = -2; dy <= 2; ++dy)
            for (int dx = -2; dx <= 2; ++dx) {
                const int x = idx + dx, y = idy + dy;
                if (y >= 0 && y < m && x >= 0 && x < n) {
                    ++cnt[tid][in[y*n+x]];
                    ++count;
                }
            }
```

```
        float ans = 0;
        for (int k = 0; k < 16; ++k)
        ans -= cnt[tid][k] ? cnt[tid][k] * (1.0 / count) * (const_log[cnt[tid][k]]-
            ↪ const_log[count]) : 0;
        out[idy*n+idx] = ans;
    }
}
```

四、实验过程

1. 实验环境

CPU	Intel(R) Core(TM) i9-10900K CPU @ 3.70GHz
GPU	NVIDIA GeForce GTX TITAN
CUDA	Cuda compilation tools, release 9.1, V9.1.85

表 1: 软硬件环境

2. 编译运行

CUDA 计算，例如矩阵大小 10000×10000，块大小 32×32，优化版本号 7：

```
$ nvcc -w main_cuda.cu -o main_cuda
$ ./main_cuda 10000 10000 32 32 7
```

OpenMP 计算，例如矩阵大小 10000×10000，线程数 32：

```
$ g++ -fopenmp main_omp.cpp -o main_omp
$ ./main_omp 10000 10000 32
```

3. 实验结果

设置不同参数，每组参数测试 5 次取均值，结果如下（单位 ms）：

blockSize	V0	V1	V2	V3	V4	V5	V6	V7
4×4	14.461	9.621	10.499	11.525	9.653	9.594	9.743	8.733
8×8	15.814	9.278	10.401	9.619	9.639	9.610	9.821	4.900
16×16	16.992	9.850	10.397	9.397	9.564	9.358	9.515	4.897
32×32	22.229	10.871	10.935	10.607	10.722	10.353	10.390	4.936

表 2: 矩阵大小 2000×2000

blockSize	V0	V1	V2	V3	V4	V5	V6	V7
4×4	90.309	60.553	66.110	71.175	50.127	50.184	51.240	48.952
8×8	81.345	57.098	58.963	51.817	49.400	49.290	60.451	30.461
16×16	101.988	61.457	53.505	48.963	48.983	48.247	69.274	27.275
32×32	142.369	66.646	68.107	60.380	56.605	53.521	53.976	30.608

表 3: 矩阵大小 5000×5000

blockSize	V0	V1	V2	V3	V4	V5	V6	V7
4×4	324.440	234.648	255.654	236.241	201.907	200.161	204.958	195.839
8×8	297.464	189.399	213.502	198.469	198.170	196.518	200.453	101.301
16×16	375.092	201.596	211.646	194.443	194.239	191.449	195.585	101.222
32×32	567.407	218.252	227.549	225.049	228.475	210.804	208.967	101.636

表 4: 矩阵大小 10000×10000

五、性能分析

1. 存储分析

在 v2 至 v6 中，我们将预处理的对数表分别放在寄存器、全局内存、纹理内存、常量内存和共享内存中，但是不同的存储内存在本次实验中相差并不大，使用常量内存相对更优。各种访存模式及存储器特性和使用原因已在核函数部分说明。

在最后的多重优化版本 v7 中，我们减少了寄存器的使用，将 cnt 用一个共享内存数组存储，防止寄存器不足后将变量放在读取延迟高的本地内存中，避免了过多的寄存器变量的使用。

2. 线程块大小

可以发现，大多数情况下使用 16×16 的线程块性能最高。当线程块大小过大时，因为 SM 中的寄存器、共享内存、缓存等资源是有限的，所以在使用过大的线程块可能会导致块内负载过大从而性能降低。

3. 预处理对数表优化版本

观察发现，所有优化版本性能从高到低大致为：v7(多重优化)，v5(预处理对数表至常量内存)，v4(预处理对数表至纹理内存)，v3(预处理对数表至全局内存)，v6(预处理对数表至共享内存)，v1(使用 unsigned char 代替 int)，v2(预处理对数表至寄存器)，v0 (Baseline 版本)。

将对数表预先存储后，实验证明能够加速运算。使用 unsigned char 替换 int 时性能得到了极大的提升，并且在使用共享内存减少寄存器的使用后，性能得到了非常明显的提升，所以可以再一次验证该程序需要缓解寄存器压力，并通过寄存器的使用情况影响活跃线程数量影响最

终的性能。

4. 影响 CUDA 程序性能的因素

根据上述分析，我们可以总结一些原因：

- 使用合理的存储器进行存储，包括纹理内存、共享内存、全局内存、常量内存等的使用。
- 使用适合的并行算法，可以提升程序性能。
- 增加 SM 的占用率，提高活跃线程的数量，包括控制线程块的大小、减少寄存器压力等。
- 优化计算语句，比如使用混合精度、减少分支控制语句等。

六、CUDA vs OpenMP

设置不同的线程数和矩阵大小，OpenMP 计算结果如下（单位 ms）：

$N \times N$ \ Thread Num	8	16	32	64
2000×2000	176.936	130.604	121.481	126.425
5000×5000	885.328	729.081	702.403	704.847
10000×10000	3282.223	2803.797	2768.166	2755.744

表 5: OpenMP 运行时间

我们将 CUDA 性能最高的配置和 OpenMP 性能最高的进行对比：

并行框架 \ $N \times N$	2000×2000	5000×5000	10000×10000
CUDA	4.897	27.275	101.222
OpenMP	121.481	702.403	2755.744

表 6: CUDA vs OpenMP

可以发现，使用 GTX TITAN 的 CUDA 计算比使用 OpenMP 性能提高了 25 倍左右。思考其中的原因，CPU 由专为顺序串行处理而优化的几个核心组成，而 GPU 则由数以千计的更小、更高效的核心组成，有着高吞吐量、高计算能力、高并发线程数等特点。GPU 可以启动大量线程高效地处理并行计算，也更加适合二维信息熵这种适合并行的任务，故 CUDA 程序能够取得更好的性能。