并行计算程序设计（双语）

作业报告

学生：王磊

学号：202231060435

课程作业二　共享内存与图像处理

（一）目的

熟悉基本的CUDA内存架构以及掌握如何在并行编程内存优化中使用共享内存；

（二）内容

任务描述：

1）实现2个矩阵相乘，矩阵A（Height=2047，Width=1023），矩阵B（Height=1023, Width=511）输入的矩阵A,B按照以下要求初始化，矩阵A的初始值全为本人学号的最后1位数字，矩阵B的初始值全为本人学号的倒数第2位数字。利用并行分块矩阵乘法（必须用共享内存）实现并比较与作业一所采用的并行矩阵乘法的耗时。

＊ 建议用二维线程结构来计算矩阵加法，每一个线程对应一个矩阵元素

2）编写image blur（图像模糊化，也称为均值滤波）CUDA并行计算程序

基于已提供的kernel.cu源代码，编写均值滤波CUDA并行计算程序，分别测试均值滤波模板尺寸为3x3,5x5两种情况下的计算效率。测试用图片为lena_noise.pgm。

＊ 建议用二维线程结构来处理本问题，每一个线程对应一个输出的图像像素

3）编写利用共享内存优化的CUDA的均值滤波并行计算程序

在任务2的基础上，编写利用共享内存优化的均值滤波CUDA并行计算程序，分别测试均值滤波模板尺寸为3x3,5x5两种情况下的计算效率。测试用图片为lena_noise.pgm。对比任务2和任务3的时间开销。

＊ 建议用二维线程结构来处理本问题，每一个线程对应一个输出的图像像素

4）　编写中值滤波的CUDA并行计算程序

基于已提供的kernel.cu源代码，编写中值滤波的CUDA并行计算程序，分别测试均值滤波模板尺寸为3x3,5x5两种情况下的计算效率。测试用图片为lena_noise.pgm

＊ 建议用二维线程结构来处理本问题，每一个线程对应一个输出的图像像素

5） 编写利用共享内存优化的中值滤波 CUDA 并行计算程序

在任务 4 的基础上，编写利用共享内存优化的中值滤波 CUDA 并行计算程序，分别测试均值滤波模板尺寸为 3x3,5x5 两种情况下的计算效率。测试用图片为 lena_noise.pgm

＊ 建议用二维线程结构来处理本问题，每一个线程对应一个输出的图像像素

步骤一 列出任务 1 的 CPU 代码和 GPU 代码及运行时间结果

CPU:

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;

// CPU 矩阵乘法
void matrixMultiplyCPU(const vector<vector<int>>& A, const vector<vector<int>>& B, vector<vector<int>>& C, int M, int N, int K) {
    for (int i = 0; i < M; ++i) {
        for (int j = 0; j < K; ++j) {
            C[i][j] = 0;
            for (int k = 0; k < N; ++k) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}

int main() {
    int M = 2047, N = 1023, K = 511;

    vector<vector<int>> A(M, vector<int>(N, 5));
    vector<vector<int>> B(N, vector<int>(K, 3));
    vector<vector<int>> C(M, vector<int>(K));
```

```
    clock_t start = clock();
    matrixMultiplyCPU(A, B, C, M, N, K);
    clock_t end = clock();

    double cpuTime = static_cast<double>(end - start) / CLOCKS_PER_SEC;
    cout << "CPU time: " << cpuTime << " seconds" << endl;

    return 0;
}
```

输出结果：

```
(base) root@853d05367535:~# ./matrixMulSharedCpu
CPU time: 10.4401 seconds
```

GPU:

```
#include <cuda_runtime.h>
#include <iostream>
using namespace std;

#define TILE_SIZE 16

// GPU 核函数：共享内存优化的矩阵乘法
__global__ void matrixMulShared(float *A, float *B, float *C, int M, int
N, int K) {
    __shared__ float tileA[TILE_SIZE][TILE_SIZE];
    __shared__ float tileB[TILE_SIZE][TILE_SIZE];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float value = 0.0f;

    for (int i = 0; i < (N + TILE_SIZE - 1) / TILE_SIZE; ++i) {
        if (row < M && i * TILE_SIZE + threadIdx.x < N) {
            tileA[threadIdx.y][threadIdx.x] = A[row * N + i * TILE_SIZE
+ threadIdx.x];
```

```
        } else {
            tileA[threadIdx.y][threadIdx.x] = 0.0f;
        }
        if (col < K && i * TILE_SIZE + threadIdx.y < N) {
            tileB[threadIdx.y][threadIdx.x] = B[(i * TILE_SIZE +
threadIdx.y) * K + col];
        } else {
            tileB[threadIdx.y][threadIdx.x] = 0.0f;
        }
        __syncthreads();

        for (int j = 0; j < TILE_SIZE; ++j) {
            value += tileA[threadIdx.y][j] * tileB[j][threadIdx.x];
        }
        __syncthreads();
    }

    if (row < M && col < K) {
        C[row * K + col] = value;
    }
}

int main() {
    int M = 2047, N = 1023, K = 511;

    float *h_A = new float[M * N];
    float *h_B = new float[N * K];
    float *h_C = new float[M * K];

    for (int i = 0; i < M * N; ++i) h_A[i] = 5.0f;
    for (int i = 0; i < N * K; ++i) h_B[i] = 3.0f;

    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, M * N * sizeof(float));
```

```
    cudaMalloc((void**)&d_B, N * K * sizeof(float));
    cudaMalloc((void**)&d_C, M * K * sizeof(float));

    cudaMemcpy(d_A, h_A, M*N*sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, N*K*sizeof(float), cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(TILE_SIZE, TILE_SIZE);
    dim3 numBlocks((K + TILE_SIZE - 1) / TILE_SIZE, (M + TILE_SIZE - 1)
/ TILE_SIZE);

    // 使用 CUDA Events 记录时间
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    // 启动核函数
    matrixMulShared<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, M, N,
K);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime = 0.0f;
    cudaEventElapsedTime(&gpuTime, start, stop);

    cudaMemcpy(h_C, d_C, M*K*sizeof(float), cudaMemcpyDeviceToHost);

    cout << "GPU time: " << gpuTime / 1000.0f << " seconds" << endl;

    // 清理资源
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
```

```
    delete[] h_A;
    delete[] h_B;
    delete[] h_C;

    return 0;
}
```

输出结果：

```
(base) root@853d05367535:~# nvcc -o matrixMulSharedGpu matrixMulSharedGpu.cu
(base) root@853d05367535:~# ./matrixMulSharedGpu
GPU time: 0.00206419 seconds_
```

与实验一对比：

CPU：10.4401s < 22.912

GPU：0.00206419s < 0.0236174s

加速比：10.4401/0.00206419=5057.722 > 970.132

结论

1. 共享内存优化在任务一的矩阵乘法中表现显著。
2. GPU 的性能提升尤为明显，尤其是在大规模矩阵计算任务中。

步骤二 列出任务 2 的 CPU 代码和 GPU 代码及运行时间和图像处理

“pgm_io.h”

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

// 读取 PGM 文件
bool readPGM(const string &filename, vector<unsigned char> &image, int
&width, int &height) {
    ifstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return false;
```

```
    }

    string magic;
    file >> magic; // 读取文件头
    if (magic != "P5") {
        cerr << "Invalid PGM file format" << endl;
        return false;
    }

    file >> width >> height; // 读取图像宽度和高度
    int maxVal;
    file >> maxVal; // 读取灰度值最大值
    file.ignore();  // 跳过换行符

    image.resize(width * height);
    file.read(reinterpret_cast<char *>(image.data()), width * height);
    file.close();

    return true;
}

// 保存 PGM 文件
bool writePGM(const string &filename, const vector<unsigned char> &image,
int width, int height) {
    ofstream file(filename, ios::binary);
    if (!file.is_open()) {
        cerr << "Error opening file: " << filename << endl;
        return false;
    }

    file << "P5\n" << width << " " << height << "\n255\n";
    file.write(reinterpret_cast<const char *>(image.data()), width *
height);
    file.close();
```

```
    return true;
}
```

CPU：

```
#include <iostream>
#include <vector>
#include <ctime>
#include <string>
#include "pgm_io.h" // 包含 PGM 文件读取和写入的辅助函数

using namespace std;

// CPU 均值滤波实现
void meanFilterCPU(const vector<unsigned char> &input, vector<unsigned
char> &output, int width, int height, int filterSize) {
    int radius = filterSize / 2;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            int sum = 0, count = 0;
            for (int dy = -radius; dy <= radius; ++dy) {
                for (int dx = -radius; dx <= radius; ++dx) {
                    int nx = x + dx, ny = y + dy;
                    if (nx >= 0 && nx < width && ny >= 0 && ny < height)
{
                        sum += input[ny * width + nx];
                        count++;
                    }
                }
            }
            output[y * width + x] = sum / count;
        }
    }
```

```
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string outputFilename = "image/lena_output_cpu.pgm";

    // 读取 PGM 图像
    vector<unsigned char> input;
    int width, height;
    if (!readPGM(inputFilename, input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> output(width * height); // 输出图像存储

    // 记录 CPU 运行时间
    clock_t start = clock();
    meanFilterCPU(input, output, width, height, 3); // 使用 3x3 均值滤
波
    clock_t end = clock();

    double cpuTime = static_cast<double>(end - start) / CLOCKS_PER_SEC;
    cout << "CPU time: " << cpuTime << " seconds" << endl;

    // 保存处理后的图像
    if (!writePGM(outputFilename, output, width, height)) {
        cerr << "Failed to write output image" << endl;
    } else {
        cout << "Output image saved as " << outputFilename << endl;
    }

    return 0;
}
```

输出结果：

3×3

```
(base) root@853d05367535:~# g++ -o mean_filter_Cpu mean_filter_Cpu.c
(base) root@853d05367535:~# ./mean_filter_Cpu
CPU time: 0.021355 seconds
Output image saved as image/lena_output_cpu.pgm
```

5×5

```
(base) root@853d05367535:~# ./mean_filter_Cpu
CPU time: 0.039541 seconds
Output image saved as image/lena_output_cpu_5*5.pgm
```

GPU：

```
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <string>
#include "pgm_io.h" // 辅助函数

using namespace std;

// GPU 核函数：均值滤波
__global__ void meanFilterGPU(const unsigned char *input, unsigned char
*output, int width, int height, int filterSize) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int radius = filterSize / 2;
    if (x < width && y < height) {
        int sum = 0, count = 0;
        for (int dy = -radius; dy <= radius; ++dy) {
            for (int dx = -radius; dx <= radius; ++dx) {
                int nx = x + dx, ny = y + dy;
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    sum += input[ny * width + nx];
                    count++;
                }
            }
        }
```

```
        output[y * width + x] = sum / count;
    }
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string outputFilename = "image/lena_output.pgm";

    // 读取 PGM 图像
    vector<unsigned char> h_input;
    int width, height;
    if (!readPGM(inputFilename, h_input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> h_output(width * height); // 主机上的输出图像
    size_t size = width * height * sizeof(unsigned char);

    unsigned char *d_input, *d_output;
    cudaMalloc((void **)&d_input, size);
    cudaMalloc((void **)&d_output, size);

    cudaMemcpy(d_input, h_input.data(), size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (height + threadsPerBlock.y - 1) / threadsPerBlock.y);

    int filterSize = 3; // 滤波模板大小

    // 记录时间
```

```
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    meanFilterGPU<<<numBlocks, threadsPerBlock>>>(d_input, d_output,
width, height, filterSize);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime = 0.0f;
    cudaEventElapsedTime(&gpuTime, start, stop);
    cout << "GPU time: " << gpuTime / 1000.0f << " seconds" << endl;

    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);

    // 保存结果到 PGM 文件
    if (!writePGM(outputFilename, h_output, width, height)) {
        cerr << "Failed to write output image" << endl;
    } else {
        cout << "Output image saved as " << outputFilename << endl;
    }

    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}
```

3×3：

```
(base) root@853d05367535:~# ./mean_filter_Gpu
GPU time: 2.8256e-05 seconds
Output image saved as image/lena_output.pgm
```

5×5

```
(base) root@853d05367535:~# ./mean_filter_Gpu
GPU time: 3.1744e-05 seconds
Output image saved as image/lena_output_GPU_5.pgm
```

前后结果：

前结果：

lena_noise.pgm

后结果：

3×3：

CPU

lena_output_cpu.pgm

GPU

lena_output.pgm

5×5

CPU

lena_output_cpu_5_5.pgm

GPU

lena_output_GPU_5.pgm

步骤三 列出任务 3 的 GPU 代码及与任务 2 的 GPU 代码运行时间及图像处理结果

代码：

```
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <string>
#include "pgm_io.h" // 包含 PGM 文件读取和保存的辅助函数

using namespace std;

// GPU 核函数：共享内存优化的均值滤波
__global__ void meanFilterShared(const unsigned char *input, unsigned
char *output, int width, int height, int filterSize) {
    extern __shared__ unsigned char sharedMem[];

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int radius = filterSize / 2;
    int sharedWidth = blockDim.x + 2 * radius;
    int sharedHeight = blockDim.y + 2 * radius;

    int localX = threadIdx.x + radius;
    int localY = threadIdx.y + radius;

    // 加载共享内存
    for (int dy = -radius; dy <= radius; ++dy) {
        for (int dx = -radius; dx <= radius; ++dx) {
            int globalX = x + dx;
            int globalY = y + dy;
            int sharedIdxX = localX + dx;
            int sharedIdxY = localY + dy;
```

```
            if (globalX >= 0 && globalX < width && globalY >= 0 && globalY
< height) {
                sharedMem[sharedIdxY * sharedWidth + sharedIdxX] =
input[globalY * width + globalX];
            } else {
                sharedMem[sharedIdxY * sharedWidth + sharedIdxX] = 0; //
边界外初始化为0
            }
        }
    }

    __syncthreads();

    // 计算均值滤波
    if (x < width && y < height) {
        int sum = 0, count = 0;
        for (int dy = -radius; dy <= radius; ++dy) {
            for (int dx = -radius; dx <= radius; ++dx) {
                sum += sharedMem[(localY + dy) * sharedWidth + (localX
+ dx)];
                count++;
            }
        }
        output[y * width + x] = sum / count;
    }
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string                  outputFilenameShared3x3                 =
"image/lena_output_shared_3x3.pgm";
    string                  outputFilenameShared5x5                 =
"image/lena_output_shared_5x5.pgm";
```

```
    // 读取 PGM 图像
    vector<unsigned char> h_input;
    int width, height;
    if (!readPGM(inputFilename, h_input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> h_output(width * height); // 输出图像
    size_t size = width * height * sizeof(unsigned char);

    unsigned char *d_input, *d_output;
    cudaMalloc((void **)&d_input, size);
    cudaMalloc((void **)&d_output, size);

    cudaMemcpy(d_input, h_input.data(), size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (height    +    threadsPerBlock.y    -    1)    /
threadsPerBlock.y);

    // 测试 3x3 模板
    int filterSize = 3;
    size_t sharedMemSize = (threadsPerBlock.x + 2 * (filterSize / 2)) *
(threadsPerBlock.y + 2 * (filterSize / 2)) * sizeof(unsigned char);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cudaEventRecord(start, 0);
    meanFilterShared<<<numBlocks,                    threadsPerBlock,
```

```
sharedMemSize>>>(d_input, d_output, width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime3x3 = 0.0f;
    cudaEventElapsedTime(&gpuTime3x3, start, stop);
    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);

    if (!writePGM(outputFilenameShared3x3, h_output, width, height)) {
        cerr << "Failed to write 3x3 filtered image" << endl;
    } else {
        cout <<"3x3 filtered image saved as " <<outputFilenameShared3x3
<< endl;
    }

    // 测试 5x5 模板
    filterSize = 5;
    sharedMemSize = (threadsPerBlock.x + 2 * (filterSize / 2)) *
(threadsPerBlock.y + 2 * (filterSize / 2)) * sizeof(unsigned char);

    cudaEventRecord(start, 0);
    meanFilterShared<<<numBlocks,                    threadsPerBlock,
sharedMemSize>>>(d_input, d_output, width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime5x5 = 0.0f;
    cudaEventElapsedTime(&gpuTime5x5, start, stop);
    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);

    if (!writePGM(outputFilenameShared5x5, h_output, width, height)) {
        cerr << "Failed to write 5x5 filtered image" << endl;
```

```
    } else {
        cout << "5x5 filtered image saved as " << outputFilenameShared5x5
<< endl;
    }

    // 输出时间信息
    cout << "GPU time (3x3 filter): " << gpuTime3x3 / 1000.0f << " seconds"
<< endl;
    cout << "GPU time (5x5 filter): " << gpuTime5x5 / 1000.0f << " seconds"
<< endl;

    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}
```

运行结果：

```
3x3 filtered image saved as image/lena_output_shared_3x3.pgm
5x5 filtered image saved as image/lena_output_shared_5x5.pgm
GPU time (3x3 filter): 2.9792e-05 seconds
GPU time (5x5 filter): 2.4704e-05 seconds
```

任务二与任务三对比：

2.8256e-05s < 2.9792e-05s

3.1744e-05s > 2.4704e-05s

3×3 模板情况下，任务二时间开销小于任务三

5×5 模板情况下，任务二时间开销大于任务三

结论：

1. 对于小模板（3x3），共享内存优化带来的额外开销（加载边界数据、同步）可能超过其减少的全局内存访问开销，因此任务二比任务三稍快。
2. 对于大模板（5x5），共享内存显著减少了全局内存访问次数，因此任务三表现出更高的效率。

图像处理结果：


lena_output_shared_3x3.pgm　　lena_output_shared_5x5.pgm

步骤四 列出任务 4 的 CPU 代码和 GPU 代码及运行时间和图像处理

CPU

```
#include <iostream>
#include <vector>
#include <ctime>
#include <string>
#include <algorithm>
#include "pgm_io.h" // 包含 PGM 文件读取和写入的辅助函数

using namespace std;

// CPU 中值滤波实现
void medianFilterCPU(const vector<unsigned char> &input,
vector<unsigned char> &output, int width, int height, int filterSize) {
    int radius = filterSize / 2;
    int filterArea = filterSize * filterSize;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            vector<unsigned char> window;
            for (int dy = -radius; dy <= radius; ++dy) {
                for (int dx = -radius; dx <= radius; ++dx) {
                    int nx = x + dx;
                    int ny = y + dy;
                    if (nx >= 0 && nx < width && ny >= 0 && ny < height)
{
                        window.push_back(input[ny * width + nx]);
                    }
```

```
                }
            }

            // 对窗口像素值进行排序
            sort(window.begin(), window.end());

            // 选择中值
            output[y * width + x] = window[window.size() / 2];
        }
    }
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string outputFilename3x3 = "image/lena_output_median_cpu_3x3.pgm";
    string outputFilename5x5 = "image/lena_output_median_cpu_5x5.pgm";

    // 读取 PGM 图像
    vector<unsigned char> h_input;
    int width, height;
    if (!readPGM(inputFilename, h_input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> h_output(width * height); // 输出图像存储
    size_t size = width * height * sizeof(unsigned char);

    // 测试 3x3 滤波器
    clock_t start = clock();
    int filterSize = 3;
    medianFilterCPU(h_input, h_output, width, height, filterSize);
    clock_t end = clock();
    double cpuTime3x3 = static_cast<double>(end - start) /
```

```
CLOCKS_PER_SEC;
    if (!writePGM(outputFilename3x3, h_output, width, height)) {
        cerr << "Failed to write 3x3 filtered image" << endl;
    } else {
        cout << "3x3 filtered image saved as " << outputFilename3x3 <<
endl;
    }

    // 测试 5x5 滤波器
    start = clock();
    filterSize = 5;
    medianFilterCPU(h_input, h_output, width, height, filterSize);
    end = clock();
    double cpuTime5x5 = static_cast<double>(end - start) /
CLOCKS_PER_SEC;
    if (!writePGM(outputFilename5x5, h_output, width, height)) {
        cerr << "Failed to write 5x5 filtered image" << endl;
    } else {
        cout << "5x5 filtered image saved as " << outputFilename5x5 <<
endl;
    }

    // 输出 CPU 运行时间
    cout << "CPU time (3x3 filter): " << cpuTime3x3 << " seconds" << endl;
    cout << "CPU time (5x5 filter): " << cpuTime5x5 << " seconds" << endl;

    return 0;
}
```

输出结果：

```
3x3 filtered image saved as image/lena_output_median_cpu_3x3.pgm
5x5 filtered image saved as image/lena_output_median_cpu_5x5.pgm
CPU time (3x3 filter): 0.437111 seconds
CPU time (5x5 filter): 0.978948 seconds
```

GPU：

```
#include <cuda_runtime.h>
```

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include "pgm_io.h" // 包含 PGM 文件读取和写入的辅助函数

using namespace std;

// GPU 核函数：中值滤波
__global__ void medianFilterGPU(const unsigned char *input, unsigned
char *output, int width, int height, int filterSize) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int radius = filterSize / 2;
    int filterArea = filterSize * filterSize;

    if (x < width && y < height) {
        // 创建一个数组存储模板区域的像素值
        unsigned char window[25]; // 最大支持 5x5 滤波器

        int count = 0;
        for (int dy = -radius; dy <= radius; ++dy) {
            for (int dx = -radius; dx <= radius; ++dx) {
                int nx = x + dx;
                int ny = y + dy;
                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    window[count++] = input[ny * width + nx];
                }
            }
        }

        // 对窗口像素值进行排序，选择中值
        for (int i = 0; i < count - 1; ++i) {
```

```
            for (int j = i + 1; j < count; ++j) {
                if (window[i] > window[j]) {
                    unsigned char temp = window[i];
                    window[i] = window[j];
                    window[j] = temp;
                }
            }
        }

        // 将中值赋给输出像素
        output[y * width + x] = window[count / 2];
    }
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string outputFilename3x3 = "image/lena_output_median_3x3.pgm";
    string outputFilename5x5 = "image/lena_output_median_5x5.pgm";

    // 读取 PGM 图像
    vector<unsigned char> h_input;
    int width, height;
    if (!readPGM(inputFilename, h_input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> h_output(width * height); // 输出图像存储
    size_t size = width * height * sizeof(unsigned char);

    unsigned char *d_input, *d_output;
    cudaMalloc((void **)&d_input, size);
    cudaMalloc((void **)&d_output, size);
```

```
    cudaMemcpy(d_input, h_input.data(), size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (height + threadsPerBlock.y - 1) /
threadsPerBlock.y);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // 测试 3x3 模板
    int filterSize = 3;
    cudaEventRecord(start, 0);
    medianFilterGPU<<<numBlocks, threadsPerBlock>>>(d_input, d_output,
width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime3x3 = 0.0f;
    cudaEventElapsedTime(&gpuTime3x3, start, stop);

    cudaMemcpy(h_output.data(),             d_output,             size,
cudaMemcpyDeviceToHost);
    if (!writePGM(outputFilename3x3, h_output, width, height)) {
        cerr << "Failed to write 3x3 filtered image" << endl;
    } else {
        cout << "3x3 filtered image saved as " << outputFilename3x3 <<
endl;
    }

    // 测试 5x5 模板
    filterSize = 5;
    cudaEventRecord(start, 0);
```

```
    medianFilterGPU<<<numBlocks, threadsPerBlock>>>(d_input, d_output,
width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime5x5 = 0.0f;
    cudaEventElapsedTime(&gpuTime5x5, start, stop);

    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);
    if (!writePGM(outputFilename5x5, h_output, width, height)) {
        cerr << "Failed to write 5x5 filtered image" << endl;
    } else {
        cout << "5x5 filtered image saved as " << outputFilename5x5 <<
endl;
    }

    // 输出运行时间
    cout <<"GPU time (3x3 filter): " << gpuTime3x3 / 1000.0f <<" seconds"
<< endl;
    cout <<"GPU time (5x5 filter): " << gpuTime5x5 / 1000.0f <<" seconds"
<< endl;

    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}
```

输出结果：

```
(base) root@853d05367535:~# ./median_filter_Gpu
3x3 filtered image saved as image/lena_output_median_3x3.pgm
5x5 filtered image saved as image/lena_output_median_5x5.pgm
GPU time (3x3 filter): 6.7424e-05 seconds
GPU time (5x5 filter): 0.000356224 seconds
```

前后结果

前结果：

lena_noise.pgm

后结果：

CPU

lena_output_median_cpu_3x3.pgm

lena_output_median_cpu_5x5.pgm

GPU

lena_output_median_3x3.pgm

lena_output_median_5x5.pgm

步骤五 列出任务 5 的 GPU 代码及与任务 4 的 GPU 代码及运行结果

代码：

```
#include <cuda_runtime.h>
#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // for sorting
#include "pgm_io.h"  // 包含 PGM 文件读取和保存的辅助函数

using namespace std;

// GPU 核函数：共享内存优化的中值滤波
__global__ void medianFilterShared(const unsigned char *input, unsigned
char *output, int width, int height, int filterSize) {
    extern __shared__ unsigned char sharedMem[];

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

```
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int radius = filterSize / 2;
    int sharedWidth = blockDim.x + 2 * radius;
    int sharedHeight = blockDim.y + 2 * radius;

    int localX = threadIdx.x + radius;
    int localY = threadIdx.y + radius;

    // 加载共享内存
    for (int dy = -radius; dy <= radius; ++dy) {
        for (int dx = -radius; dx <= radius; ++dx) {
            int globalX = x + dx;
            int globalY = y + dy;
            int sharedIdxX = localX + dx;
            int sharedIdxY = localY + dy;

            if (globalX >= 0 && globalX < width && globalY >= 0 && globalY
< height) {
                sharedMem[sharedIdxY * sharedWidth + sharedIdxX] =
input[globalY * width + globalX];
            } else {
                sharedMem[sharedIdxY * sharedWidth + sharedIdxX] = 0; //
边界外初始化为0
            }
        }
    }

    __syncthreads();

    // 计算中值滤波
    if (x < width && y < height) {
        unsigned char window[25]; // 最大支持 5x5 滤波器
        int count = 0;
```

```
        for (int dy = -radius; dy <= radius; ++dy) {
            for (int dx = -radius; dx <= radius; ++dx) {
                window[count++] = sharedMem[(localY + dy) * sharedWidth
+ (localX + dx)];
            }
        }

        // 对窗口像素值进行排序
        for (int i = 0; i < count - 1; ++i) {
            for (int j = i + 1; j < count; ++j) {
                if (window[i] > window[j]) {
                    unsigned char temp = window[i];
                    window[i] = window[j];
                    window[j] = temp;
                }
            }
        }

        // 将中值赋给输出像素
        output[y * width + x] = window[count / 2];
    }
}

int main() {
    string inputFilename = "image/lena_noise.pgm";
    string                      outputFilename3x3                      =
"image/lena_output_median_shared_3x3.pgm";
    string                      outputFilename5x5                      =
"image/lena_output_median_shared_5x5.pgm";

    // 读取 PGM 图像
    vector<unsigned char> h_input;
    int width, height;
```

```
    if (!readPGM(inputFilename, h_input, width, height)) {
        cerr << "Failed to read input image" << endl;
        return -1;
    }

    vector<unsigned char> h_output(width * height); // 输出图像存储
    size_t size = width * height * sizeof(unsigned char);

    unsigned char *d_input, *d_output;
    cudaMalloc((void **)&d_input, size);
    cudaMalloc((void **)&d_output, size);

    cudaMemcpy(d_input, h_input.data(), size, cudaMemcpyHostToDevice);

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((width + threadsPerBlock.x - 1) / threadsPerBlock.x,
                   (height + threadsPerBlock.y - 1) /
threadsPerBlock.y);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    // 测试 3x3 模板
    int filterSize = 3;
    size_t sharedMemSize = (threadsPerBlock.x + 2 * (filterSize / 2)) *
(threadsPerBlock.y + 2 * (filterSize / 2)) * sizeof(unsigned char);

    cudaEventRecord(start, 0);
    medianFilterShared<<<numBlocks, threadsPerBlock,
sharedMemSize>>>(d_input, d_output, width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
```

```
    float gpuTime3x3 = 0.0f;
    cudaEventElapsedTime(&gpuTime3x3, start, stop);

    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);
    if (!writePGM(outputFilename3x3, h_output, width, height)) {
        cerr << "Failed to write 3x3 filtered image" << endl;
    } else {
        cout << "3x3 filtered image saved as " << outputFilename3x3 <<
endl;
    }

    // 测试 5x5 模板
    filterSize = 5;
    sharedMemSize = (threadsPerBlock.x + 2 * (filterSize / 2)) *
(threadsPerBlock.y + 2 * (filterSize / 2)) * sizeof(unsigned char);

    cudaEventRecord(start, 0);
    medianFilterShared<<<numBlocks,                  threadsPerBlock,
sharedMemSize>>>(d_input, d_output, width, height, filterSize);
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    float gpuTime5x5 = 0.0f;
    cudaEventElapsedTime(&gpuTime5x5, start, stop);

    cudaMemcpy(h_output.data(),            d_output,            size,
cudaMemcpyDeviceToHost);
    if (!writePGM(outputFilename5x5, h_output, width, height)) {
        cerr << "Failed to write 5x5 filtered image" << endl;
    } else {
        cout << "5x5 filtered image saved as " << outputFilename5x5 <<
endl;
    }
```

```
    // 输出运行时间
    cout << "GPU time (3x3 filter): " << gpuTime3x3 / 1000.0f << " seconds"
<< endl;
    cout << "GPU time (5x5 filter): " << gpuTime5x5 / 1000.0f << " seconds"
<< endl;

    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}
```

输出结果：

```
(base) root@853d05367535:~# ./median_filter_shared
3x3 filtered image saved as image/lena_output_median_shared_3x3.pgm
5x5 filtered image saved as image/lena_output_median_shared_5x5.pgm
GPU time (3x3 filter): 6.8128e-05 seconds
GPU time (5x5 filter): 0.000351936 seconds
```


lena_output_me dian_shared_3x3.

lena_output_me dian_shared_5x5.

与任务四对比：

6.8128e-05s > 6.7424e-05s

0.000351936s < 0.000356224s

结论：

1. 小模板（3x3）：
   (1) 共享内存优化的开销超过了其带来的性能收益。
   (2) 在这种情况下，直接访问全局内存可能更高效。
2. 大模板（5x5）：
   (1) 共享内存减少了全局内存访问的次数，显著降低了访问延迟。
   (2) 对于更大的模板（如 7x7 或 9x9），共享内存的性能提升将更加明显