中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　　　　　　　　　**批改人：**

实验	CUDA 卷积	专业（方向）	**计算机科学与技术**
学号	21307082	姓名	**赖耿桂**
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.6.21

1. 实验目的（200 字以内）

- **实验一：**

使用滑窗法实现图像卷积，并分析不同图像大小、访存方式、任务/数据划分方式、线程块大小等因素对程序性能的影响。

- **实验二：**

使用 im2col 方法实现图像卷积，并分析不同图像大小、访存方式、任务/数据划分方式、线程块大小等因素对程序性能的影响。

- **实验三：**

使用 cuDNN 实现图像卷积，记录其相应 Input 的卷积时间，与自己实现的卷积操作进行比较。如果性能不如 cuDNN，用文字描述可能的改进方法。

2. 实验过程和核心代码（600 字左右，图文并茂）

- **实验一：**

我使用 3 个相同大小的矩阵来模拟三通道图像的三个通道的像素，使用以下函数来随机生成一个指定大小的矩阵：

```
__host__ void init(float* mat, int row, int col) {
    // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_int_distribution<int> distr(0, 255);

    for (int i = 0; i < row; ++i) {
        for (int j = 0; j < col; ++j) {
            mat[i * col + j] = distr(eng);
        }
    }
}
```

然后为了保存卷积结果，我编写了以下函数来将卷积结果保存到一个文本文件中：

```
__host__ void write_result(float* r, float* g, float *b, int row, int col, string filename){
    ofstream f(filename);
    if(f.is_open()){
        f << "red:\n";
        for(int i = 0; i < row; ++i){
            for(int j = 0; j < col; ++j){
                if(r[i * col + j] > 255){
                    r[i * col + j] = 255;
                }
                if(r[i * col + j] < 0){
                    r[i * col + j] = 0;
                }
                f << r[i * col + j]<<" ";
            }
            f << "\n";
        }
        f << "\n" << "green:\n";
        for(int i = 0; i < row; ++i){
            for(int j = 0; j < col; ++j){
                if(g[i * col + j] > 255){
                    g[i * col + j] = 255;
                }
                if(g[i * col + j] < 0){
                    g[i * col + j] = 0;
                }
                f << g[i * col + j]<<" ";
            }
            f << "\n";
        }
        f << "\n" << "blue:\n";
        for(int i = 0; i < row; ++i){
            for(int j = 0; j < col; ++j){
                if(b[i * col + j] > 255){
                    b[i * col + j] = 255;
                }
                if(b[i * col + j] < 0){
                    b[i * col + j] = 0;
                }
                f << b[i * col + j]<<" ";
            }
            f << "\n";
        }
    }
}
```

接下来是卷积操作的代码：
访存方式为全局内存：

```
__global__ void convolution_global(float* mat, float* res, float* filter, int w_in, int h_in,
                                  int w_out, int h_out, int f_w, int f_h, int s){
    int y = blockIdx.y * blockDim.y + threadIdx.y, x = blockIdx.x * blockDim.x + threadIdx.x;
    if(x >= 0 && x < w_out && y >= 0 && y < h_out){
        float sum = 0;
        for(int i = 0; i < f_h; ++i){
            for(int j = 0; j < f_w; ++j){
                sum += filter[i * f_w + j] * mat[(y * s + i) * w_in + (x * s + j)];
            }
        }
        atomicAdd(&res[y * w_out + x], sum);
    }
}
```

访存方式为共享内存：

```
__global__ void convolution_shared(float* mat, float* res, float* filter, int w_in, int h_in,
                                  int w_out, int h_out, int f_w, int f_h, int s){
    extern __shared__ float smem[];
    int y = blockIdx.y * blockDim.y + threadIdx.y, x = blockIdx.x * blockDim.x + threadIdx.x;
    if(x >= 0 && x < w_out && y >= 0 && y < h_out){
        for(int i = 0; i < f_h; ++i){
            for(int j = 0; j < f_w; ++j){
                smem[i * (f_w + 1) + j] = mat[(y * s + i) * w_in + (x * s + j)];
            }
        }
        __synthreads();
        float sum = 0;
        for(int i = 0; i < f_h; ++i){
            for(int j = 0; j < f_w; ++j){
                sum += filter[i * f_w + j] * smem[i * (f_w + 1) + j];
            }
        }
        atomicAdd(&res[y * w_out + x], sum);
    }
}
```

说明：由于需要将三个卷积核的卷积结果加起来作为最终结果，因此在最后直接将该部分的卷积结果 sum 加到卷积结果的相应位置上，此处使用 atomicAdd 来避免冲突。

接下来在 main 函数中，定义一些变量并输入：

```
int w, h, w_pad, h_pad, w_out, h_out, f_w = 3, f_h = 3, s, threads, blockDim_x = 1, blockDim_y = 1;
    printf("请输入图像大小：\n");
    scanf("%d %d", &h, &w);
    printf("请输入步长：\n");
    scanf("%d", &s);
    printf("请输入输出图像大小：\n");
    scanf("%d %d", &h_out, &w_out);
    printf("请输入每个线程块内的线程数：\n");
    scanf("%d", &threads);
    printf("请输入线程块的维度：\n");
    while (scanf("%d %d", &blockDim_y, &blockDim_x) == 2 && blockDim_y * blockDim_x != threads) {
        printf("输入的维度不符合要求，两个维度的乘积要等于每个块内的线程数，请重新输入：\n");
    }
```

说明：w/h 为输入图像的宽高、w_pad/h_pad 为进行填充后的图像的宽高，w_out/h_out 为最终卷积后的输出图像的宽高，f_w/f_h 为卷积核的宽高，s 为步长，threads 为每个线程块内的线程数，blockDim_x/blockDim_y 为线程块的维度。

接下来初始化矩阵，并根据实际情况对其进行填充，此外将三个卷积核分别设置成元素全为 0.01、0.05、0.09 的 3*3 的三通道的矩阵也即每个矩阵内共有 27 个相同的元素：

```
init(r, h, w);
    init(g, h, w);
    init(b, h, w);
    int delta_w = (w_out - 1) * s - w + f_w, delta_h = (h_out - 1) * s - h + f_h;//根据输出维度、步长和卷积核大小，计算出需要额外填充的列数和行数
    bool flag = (delta_w != 0) && (delta_h != 0); //行和列是否都需要填充(由于宽高相等，因此行和列要么都要进行填充，要么都不需要进行填充)
    if(flag){
        // 计算上下左右外围需要填充的行数和列数，如果能被 2 整除，上下/左右各填充一半，如果不行则下方/右侧多填充一行/列
        int pad_half_w_1 = delta_w / 2, pad_half_w_2 = (delta_w % 2 == 0) ? delta_w / 2 : delta_w / 2 + 1;
        int pad_half_h_1 = delta_h / 2, pad_half_h_2 = (delta_h % 2 == 0) ? delta_h / 2 : delta_h / 2 + 1;
        w_pad = w + delta_w;
        h_pad = h + delta_h;
        cudaMallocHost(&r_pad, sizeof(float) * w_pad * h_pad);
        cudaMallocHost(&g_pad, sizeof(float) * w_pad * h_pad);
        cudaMallocHost(&b_pad, sizeof(float) * w_pad * h_pad);
        for(int i = 0; i < h_pad; ++i){
            for(int j = 0; j < w_pad; ++j){
                int index = i * w_pad + j;
                if(i >= pad_half_h_1 && i < h_pad - pad_half_h_2 &&
                   j >= pad_half_w_1 && j < w_pad - pad_half_w_2){
                    int original_index = (i - pad_half_h_1) * w + (j - pad_half_w_1);
                    r_pad[index] = r[original_index];
                    g_pad[index] = g[original_index];
                    b_pad[index] = b[original_index];
                }
                else{
                    r_pad[index] = 0;
                    g_pad[index] = 0;
                    b_pad[index] = 0;
                }
            }
        }

    }
    else{//不用进行零填充,直接将前面的r、g、b 复制给r_pad、g_pad、b_pad 即可
        w_pad = w;
        h_pad = h;
        cudaMallocHost(&r_pad, sizeof(float) * w_pad * h_pad);
        cudaMallocHost(&g_pad, sizeof(float) * w_pad * h_pad);
        cudaMallocHost(&b_pad, sizeof(float) * w_pad * h_pad);
        cudaMemcpy(r_pad, r, w_pad * h_pad * sizeof(float), cudaMemcpyHostToHost);
        cudaMemcpy(g_pad, g, w_pad * h_pad * sizeof(float), cudaMemcpyHostToHost);
        cudaMemcpy(b_pad, b, w_pad * h_pad * sizeof(float), cudaMemcpyHostToHost);
    }

    // 初始化3 个三通道卷积核
    cudaMallocHost(&f_1, sizeof(float) * f_h * f_w * 3);
    cudaMallocHost(&f_2, sizeof(float) * f_h * f_w * 3);
    cudaMallocHost(&f_3, sizeof(float) * f_h * f_w * 3);
    for(int i = 0; i < f_h * f_w * 3; ++i){
        f_1[i] = 0.01;
        f_2[i] = 0.05;
        f_3[i] = 0.09;
    }
```

接下来计算块内信息和描述网格的配置信息，调用核函数并进行计时：
访存方式为全局内存：

```
    dim3 blockDim(blockDim_x, blockDim_y, 1);
    dim3 gridDim((int)((w_out + blockDim.x - 1)/blockDim.x), (int)((h_out + blockDim.y - 1)/blockDim.y));
    cudaEvent_t start, end;
    cudaEventCreate(&start);
    cudaEventCreate(&end);
    cudaMallocHost(&r_res, sizeof(float) * h_out * w_out);
    cudaMallocHost(&g_res, sizeof(float) * h_out * w_out);
    cudaMallocHost(&b_res, sizeof(float) * h_out * w_out);
    cudaEventRecord(start, 0);
    convolution_global<<<gridDim, blockDim>>>(r_pad, r_res, f_1, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(g_pad, g_res, f_1+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(b_pad, b_res, f_1+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(r_pad, r_res, f_2, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(g_pad, g_res, f_2+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(b_pad, b_res, f_2+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(r_pad, r_res, f_3, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(g_pad, g_res, f_3+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_global<<<gridDim, blockDim>>>(b_pad, b_res, f_3+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    cudaEventRecord(end, 0);
    cudaEventSynchronize(end);

    float ms = 0;
    cudaEventElapsedTime(&ms, start, end);
```

访存方式为共享内存(仅改变调用核函数部分，选取像素矩阵的 3*3 网格的元素，注意防止 bank 冲突)：

```
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(r_pad, r_res, f_1, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(g_pad, g_res, f_1+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
```

```
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(b_pad, b_res, f_1+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(r_pad, r_res, f_2, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(g_pad, g_res, f_2+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(b_pad, b_res, f_2+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(r_pad, r_res, f_3, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(g_pad, g_res, f_3+9, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
        convolution_shared<<<gridDim, blockDim, sizeof(float) * f_h * (f_w + 1)>>>(b_pad, b_res, f_3+18, w_pad, h_pad, w_out, h_out, f_w, f_h, s);
```

接下来输出相关信息、保存卷积结果并释放内存：
访存方式为全局内存：

```
        printf("图像大小:%d*%d;线程块维度:(%d,%d);访存方式:全局内存;所用时间:%f ms\n",h,w,blockDim_y,blockDim_x,ms);
        string output_filename = to_string(h) + "*" + to_string(w) + " " + "(" + to_string(blockDim_y) + "," +  to_string(blockDim_x) + ") sliding_window global.txt";
        write_result(r_res, g_res, b_res, h_out, w_out, output_filename);
        cudaFreeHost(r);
        cudaFreeHost(g);
        cudaFreeHost(b);
        cudaFreeHost(r_res);
        cudaFreeHost(g_res);
        cudaFreeHost(b_res);
        cudaFreeHost(r_pad);
        cudaFreeHost(g_pad);
        cudaFreeHost(b_pad);
        cudaFreeHost(f_1);
        cudaFreeHost(f_2);
        cudaFreeHost(f_3);
        cudaEventDestroy(start);
        cudaEventDestroy(end);
```

访存方式为共享内存(仅输出相关信息和保存卷积结果时文件名不同)：

```
        printf("图像大小:%d*%d;线程块维度:(%d,%d);访存方式:共享内存;所用时间:%f ms\n",h,w,blockDim_y,blockDim_x,ms);
        string output_filename = to_string(h) + "*" + to_string(w) + " " + "(" + to_string(blockDim_x) + "," +  to_string(blockDim_y) + ") sliding_window shared.txt";
```

- **实验二：**

实验二的大部分代码和实验一的一致，主要区别来自于要将需要进行卷积的像素矩阵的元素提取出来并将它们组成一个列向量，卷积操作则改为使用卷积核和该线程块所需计算的列向量进行矩阵乘法即可。

im2col 核函数的代码如下：
访存方式为全局内存：

```
__global__ void im2col(float* input, float* output, int f_w, int f_h, int w_in, int h_in, int w_out, int h_out, int s){
    int y = blockIdx.y * blockDim.y + threadIdx.y, x = blockIdx.x * blockDim.x + threadIdx.x;
    if(x >= 0 && x < w_out && y >= 0 && y < h_out){
        int col_index = y * w_out + x;
        for(int k = 0; k < f_h; ++k){
            for(int p = 0; p < f_w; ++p){
                int row_index = k * f_w + p;
                int input_row = y * s + k, input_col = x * s + p;
                output[row_index * (w_out * h_out) + col_index] = input[input_row * w_in + input_col];
            }
         }
    }
}
```

访存方式为共享内存：

```
__global__ void convolution_shared(float* input, float* output, float* filter, int w_out, int h_out, int f_w, int f_h){
    extern __shared__ float smem[];
    int y = blockIdx.y * blockDim.y + threadIdx.y, x = blockIdx.x * blockDim.x + threadIdx.x;
    if(x >= 0 && x < w_out && y >= 0 && y < h_out){
        int index_1 = y * w_out + x;
        for(int i = 0; i <f_h; ++i){
            for(int j = 0; j < f_w; ++j){
                smem[i * f_w + j] = input[(i * f_w + j) * (w_out * h_out) + index_1];
            }
        }
        __syncthreads();
        float sum = 0;
        for(int i = 0; i < f_h; ++i){
            for(int j = 0; j < f_w; ++j){
                int index_2 = i * f_w + j;
                sum += smem[index_2] * filter[index_2];
            }
        }
        atomicAdd(&output[index_1], sum);
    }
}
```

此外还多定义了三个数组 r_col、g_col、b_col：

```
float *r_col, *g_col, *b_col;
cudaMallocHost(&r_col, sizeof(float) * h_out * w_out * f_w * f_h);
cudaMallocHost(&g_col, sizeof(float) * h_out * w_out * f_w * f_h);
cudaMallocHost(&b_col, sizeof(float) * h_out * w_out * f_w * f_h);
```

在调用核函数时，需要先调用 im2col 函数：

访存方式为全局内存：

```
    cudaEventRecord(start, 0);
    im2col<<<gridDim, blockDim>>>(r_pad, r_col, f_w, f_h, w_pad, h_pad, w_out, h_out, s);
    im2col<<<gridDim, blockDim>>>(g_pad, g_col, f_w, f_h, w_pad, h_pad, w_out, h_out, s);
    im2col<<<gridDim, blockDim>>>(b_pad, b_col, f_w, f_h, w_pad, h_pad, w_out, h_out, s);
    convolution_global<<<gridDim, blockDim>>>(r_col, r_res, f_1, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(g_col, g_res, f_1+9, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(b_col, b_res, f_1+18, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(r_col, r_res, f_2, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(g_col, g_res, f_2+9, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(b_col, b_res, f_2+18, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(r_col, r_res, f_3, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(g_col, g_res, f_3+9, w_out, h_out, f_w, f_h);
    convolution_global<<<gridDim, blockDim>>>(b_col, b_res, f_3+18, w_out, h_out, f_w, f_h);
    cudaEventRecord(end, 0);
    cudaEventSynchronize(end);
```

访存方式为共享内存(修改卷积核函数调用部分)：

```
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(r_col, r_res, f_1, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(g_col, g_res, f_1+9, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(b_col, b_res, f_1+18, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(r_col, r_res, f_2, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(g_col, g_res, f_2+9, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(b_col, b_res, f_2+18, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(r_col, r_res, f_3, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(g_col, g_res, f_3+9, w_out, h_out, f_w, f_h);
    convolution_shared<<<gridDim, blockDim, sizeof(float) * f_w * f_h>>>(b_col, b_res, f_3+18, w_out, h_out, f_w, f_h);
```

此外修改了相关信息输出和输出文件的文件名部分：
访存方式为全局内存：

```
printf("图像大小:%d*%d;线程块维度:(%d,%d);访存方式:全局内存;所用时间:%f ms\n",h,w,blockDim_y,blockDim_x,ms);
string output_filename = to_string(h) + "*" + to_string(w) + " " + "(" + to_string(blockDim_x) + ", " +  to_string(blockDim_y) + ") im2col global.txt";
```

访存方式为共享内存：

```
printf("图像大小:%d*%d;线程块维度:(%d,%d);访存方式:共享内存;所用时间:%f ms\n",h,w,blockDim_y,blockDim_x,ms);
string output_filename = to_string(h) + "*" + to_string(w) + " " + "(" + to_string(blockDim_x) + ", " +  to_string(blockDim_y) + ") im2col shared.txt";
```

释放内存时要多释放 r_col、g_col、b_col：

```
    cudaFreeHost(r_col);
    cudaFreeHost(g_col);
    cudaFreeHost(b_col);
```

- **实验三：**

首先需要配置 cuDNN 环境，参考链接：Ubuntu 安装 cuda+cudnn 保姆级教程记录 - 知乎 (zhihu.com)。

首先定义一个宏，用于检查 cuDNN API 调用的返回状态，并在遇到错误时输出错误信息并终止程序：

```
#define Check(expression)                               \
  {                                                          \
    cudnnStatus_t status = (expression);                     \
    if (status != CUDNN_STATUS_SUCCESS) {                    \
      cerr << "Error on line " << __LINE__ << ": "      \
                << cudnnGetErrorString(status) << endl; \
      exit(EXIT_FAILURE);                               \
    }                                                        \
  }
```

然后和前面两个实验一样，通过键盘输入原图像的宽高、步长、输出图像的宽高，不再赘述。

创建句柄，初始化 cuDNN：

```
cudnnHandle_t handle;
```

```
Check(cudnnCreate(&handle));
```

初始化输入矩阵和卷积核(此处卷积核与前面不同，全为 0.01)、为输出矩阵分配空间、计算填充的部分：

```
float* filter;
cudaMallocHost(&filter, f_w * f_h * 3 * 3 * sizeof(float));
for(int i = 0; i < f_w*f_h*3*3; ++i){
    filter[i] = 0.01;
}
int imageBytes = w * h * 3 * sizeof(float);
float *image;
cudaMallocHost(&image, imageBytes);
init(image, h * 3, w);

int delta = (w_out - 1) * stride - w + f_w;
int pad = (delta + 1) / 2;
float* res;
int outBytes = w_out * h_out * 3 * sizeof(float);
cudaMallocHost(&res, outBytes);
```

声明输入/输出描述符、滤波器描述符、卷积描述符：

```
cudaMallocHost(&res, outBytes);
cudnnTensorDescriptor_t input_desc, output_desc;
cudnnFilterDescriptor_t filter_desc;
cudnnConvolutionDescriptor_t conv_desc;
```

定义输入描述符，设置输入矩阵的格式、类型、批处理大小、通道数、高度和宽度等：

```
Check(cudnnCreateTensorDescriptor(&input_desc));
Check(cudnnSetTensor4dDescriptor(input_desc, CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, 1, 3, h, w));
```

定义滤波器描述符，设置格式、类型、输入输出通道数、高度、宽度等：

```
Check(cudnnCreateFilterDescriptor(&filter_desc));
Check(cudnnSetFilter4dDescriptor(filter_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 3, 3, 3, 3))
;
```

定义卷积描述符，设置两种方向上填充 pad 大小、步长 stride、dilation_height、模式和类型：

```
Check(cudnnCreateConvolutionDescriptor(&conv_desc));
Check(cudnnSetConvolution2dDescriptor(conv_desc, pad, pad, stride, stride, 1, 1, CUDNN_CROSS_CO
RRELATION, CUDNN_DATA_FLOAT));
```

定义输出描述符，设置输入矩阵的格式、类型、批处理大小、通道数、高度和宽度等：

```
Check(cudnnCreateTensorDescriptor(&output_desc));
Check(cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NHWC, CUDNN_DATA_FLOAT, 1, 3, h_out, w_out));
```

使用上述定义好的描述符，定义卷积算法 conv_algorithm：

```
cudnnConvolutionFwdAlgo_t conv_algorithm;
int returnedAlgoCount;
cudnnConvolutionFwdAlgoPerf_t conv_algorithm_perf;
Check(cudnnGetConvolutionForwardAlgorithm_v7(handle, input_desc, filter_desc, conv_desc, output_desc, 1, &returnedAlgoCount, &conv_algorithm_perf));
conv_algorithm = conv_algorithm_perf.algo;
```

由于使用的 cuDNN 版本为 8.7，因此此处应使用 **cudnnGetConvolutionForwardAlgorithm_v7**，使用 **cudnnConvolutionFwdAlgoPerf_t** 结构体来获取最佳算法。

用 cudnnGetConvolutionForwardWorkspaceSize() 函数计算整个运算所需要的空间大小 workspaceBytes：

```
size_t workspaceBytes = 0;
Check(cudnnGetConvolutionForwardWorkspaceSize(handle, input_desc, filter_desc, conv_desc, output_desc, conv_algorithm, &workspaceBytes));
```

为 workspace 分配空间：

```
void* workspace;
cudaMallocHost(&workspace, workspaceBytes);
```

做卷积前的准备：

```
float alpha = 1, beta = 0;
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
```

调用函数 cudnnConvolutionForward() 计算卷积结果并计时：

```
cudaEventRecord(start, 0);
Check(cudnnConvolutionForward(handle, &alpha, input_desc, image, filter_desc, filter, conv_desc, conv_algorithm, workspace, workspaceBytes, &beta, output_desc, res));
cudaEventRecord(end, 0);
cudaEventSynchronize(end);
float ms = 0;
```

```
    cudaEventElapsedTime(&ms, start, end);
```

输出相关信息、将运算结果保存到一个文本文件中，并释放内存：

```
    printf("图像大小:%d*%d;所用时间:%f ms\n",h,w,ms);
    string output_name = to_string(h)+"*"+to_string(w)+"_cuDNN.txt";
    ofstream f(output_name);
    if(f.is_open()){
        for(int i = 0 ; i < 3; ++i){
            for(int j = 0; j < h_out; ++j){
                for(int k = 0; k < w_out; ++k){
                    if(res[j * w_out + k] > 255){
                        res[j * w_out + k] = 255;
                    }
                    if(res[j * w_out + k] < 0){
                        res[j * w_out + k] = 0;
                    }
                    f << res[j * w_out + k]<<" ";
                }
                f << "\n";
            }
            f << "\n";
        }
    }
    cudnnDestroyTensorDescriptor(input_desc);
    cudnnDestroyTensorDescriptor(output_desc);
    cudnnDestroyFilterDescriptor(filter_desc);
    cudnnDestroyConvolutionDescriptor(conv_desc);
    cudaFreeHost(workspace);
    cudaFreeHost(res);
    cudaFreeHost(image);
    cudaFreeHost(filter);
    cudnnDestroy(handle);
    cudaEventDestroy(start);
    cudaEventDestroy(end);
```

3. 实验结果（500 字左右，图文并茂）

● **实验一：**

■ 编译：

```
nvcc -g -o filename filename.cu
```

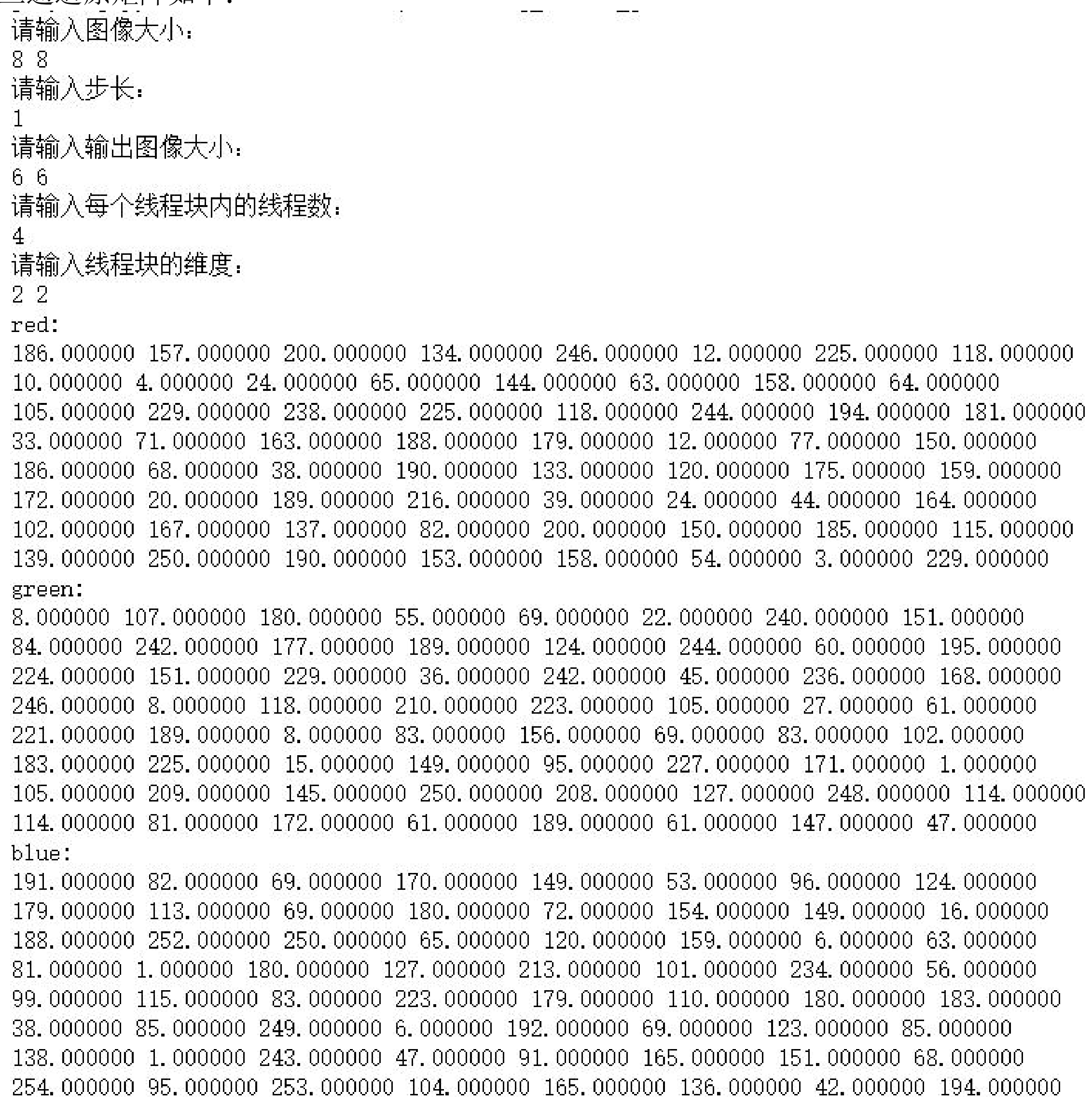
■ 运行：

```
./filename
```

(filename: Sliding_Window_global.cu/Sliding_Window_shared.cu，表示使用全局/共享内存的访存方式进行卷积)

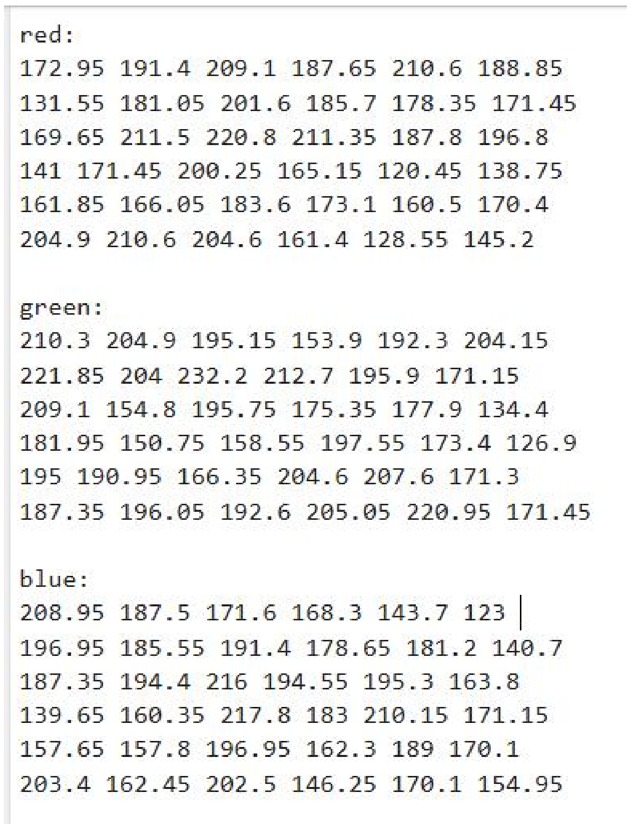
■ 运行结果示例，用于展示正确性：

三通道原矩阵如下：



```
请输入图像大小：
8 8
请输入步长：
1
请输入输出图像大小：
6 6
请输入每个线程块内的线程数：
4
请输入线程块的维度：
2 2
red:
186.000000 157.000000 200.000000 134.000000 246.000000 12.000000 225.000000 118.000000
10.000000 4.000000 24.000000 65.000000 144.000000 63.000000 158.000000 64.000000
105.000000 229.000000 238.000000 225.000000 118.000000 244.000000 194.000000 181.000000
33.000000 71.000000 163.000000 188.000000 179.000000 12.000000 77.000000 150.000000
186.000000 68.000000 38.000000 190.000000 133.000000 120.000000 175.000000 159.000000
172.000000 20.000000 189.000000 216.000000 39.000000 24.000000 44.000000 164.000000
102.000000 167.000000 137.000000 82.000000 200.000000 150.000000 185.000000 115.000000
139.000000 250.000000 190.000000 153.000000 158.000000 54.000000 3.000000 229.000000
green:
8.000000 107.000000 180.000000 55.000000 69.000000 22.000000 240.000000 151.000000
84.000000 242.000000 177.000000 189.000000 124.000000 244.000000 60.000000 195.000000
224.000000 151.000000 229.000000 36.000000 242.000000 45.000000 236.000000 168.000000
246.000000 8.000000 118.000000 210.000000 223.000000 105.000000 27.000000 61.000000
221.000000 189.000000 8.000000 83.000000 156.000000 69.000000 83.000000 102.000000
183.000000 225.000000 15.000000 149.000000 95.000000 227.000000 171.000000 1.000000
105.000000 209.000000 145.000000 250.000000 208.000000 127.000000 248.000000 114.000000
114.000000 81.000000 172.000000 61.000000 189.000000 61.000000 147.000000 47.000000
blue:
191.000000 82.000000 69.000000 170.000000 149.000000 53.000000 96.000000 124.000000
179.000000 113.000000 69.000000 180.000000 72.000000 154.000000 149.000000 16.000000
188.000000 252.000000 250.000000 65.000000 120.000000 159.000000 6.000000 63.000000
81.000000 1.000000 180.000000 127.000000 213.000000 101.000000 234.000000 56.000000
99.000000 115.000000 83.000000 223.000000 179.000000 110.000000 180.000000 183.000000
38.000000 85.000000 249.000000 6.000000 192.000000 69.000000 123.000000 85.000000
138.000000 1.000000 243.000000 47.000000 91.000000 165.000000 151.000000 68.000000
254.000000 95.000000 253.000000 104.000000 165.000000 136.000000 42.000000 194.000000
```

卷积结果如下：

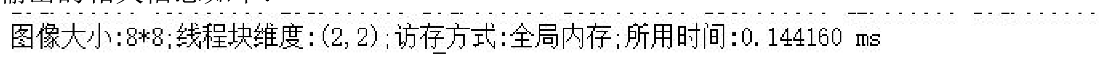


```
red:
172.95 191.4 209.1 187.65 210.6 188.85
131.55 181.05 201.6 185.7 178.35 171.45
169.65 211.5 220.8 211.35 187.8 196.8
141 171.45 200.25 165.15 120.45 138.75
161.85 166.05 183.6 173.1 160.5 170.4
204.9 210.6 204.6 161.4 128.55 145.2

green:
210.3 204.9 195.15 153.9 192.3 204.15
221.85 204 232.2 212.7 195.9 171.15
209.1 154.8 195.75 175.35 177.9 134.4
181.95 150.75 158.55 197.55 173.4 126.9
195 190.95 166.35 204.6 207.6 171.3
187.35 196.05 192.6 205.05 220.95 171.45

blue:
208.95 187.5 171.6 168.3 143.7 123
196.95 185.55 191.4 178.65 181.2 140.7
187.35 194.4 216 194.55 195.3 163.8
139.65 160.35 217.8 183 210.15 171.15
157.65 157.8 196.95 162.3 189 170.1
203.4 162.45 202.5 146.25 170.1 154.95
```

输出的相关信息如下：



```
图像大小:8*8;线程块维度:(2,2);访存方式:全局内存;所用时间:0.144160 ms
```

共享内存同理。

■ 实验结果：

全局内存：

stride 为 1：

线程块大小	图像大小		
	1024	2048	4096
64	69.129150ms 42.127998ms 551.961731ms	323.232422ms 180.832520ms 2272.778320ms	1172.156738ms 713.587830ms 9218.918945ms
256	35.602818ms 39.311039ms 874.031189ms	148.748642ms 182.223129ms 3571.016602ms	586.418579ms 698.351013ms 14435.688477ms
1024	27.381952ms 42.108383ms 1205.381348ms	107.167068ms 178.587769ms 4788.390625ms	433.462433ms 680.412903ms 19282.039062ms

stride 为 2：

线程块大小	图像大小		
	1024	2048	4096
64	93.710876ms	367.012299ms	1500.975098ms

	72.167137ms 861.114990ms	310.521790ms 3502.279541ms	1262.296997ms 13907.037109ms
256	65.626434ms 71.750206ms 1176.142822ms	290.051575ms 304.314575ms 4726.913574ms	1160.820557ms 1232.099731ms 18971.720703ms
1024	61.181343ms 70.706398ms 1298.643677ms	241.674118ms 305.346039ms 5114.544922ms	974.830139ms 1222.972778ms 21585.412109ms

stride 为 3：

线程块大小	图像大小		
	1024	2048	4096
64	151.400574ms 114.687134ms 960.939453ms	588.239990ms 440.390747ms 3809.122803ms	2376.727295ms 1762.500244ms 15368.015625ms
256	115.762077ms 108.648766ms 1207.178833ms	520.577637ms 429.909302ms 4971.554688ms	2074.884033ms 1740.344604ms 19551.083984ms
1024	112.720322ms 104.944290ms 1312.314575ms	473.851562ms 433.507416ms 5553.537598ms	1882.430908ms 1731.501465ms 33176.207031 ms

共享内存：

stride 为 1：

线程块大小	图像大小		
	1024	2048	4096
64	68.620926ms	278.621338ms	1101.816040ms
256	35.161022ms	142.007614ms	585.647583ms
1024	26.071615ms	105.564606ms	431.394135ms

stride 为 2：

线程块大小	图像大小		
	1024	2048	4096
64	91.904259ms	331.314667ms	1462.527588ms
256	72.212769ms	290.637329ms	1154.681519ms
1024	61.393726ms	241.925735ms	1005.998413ms

stride 为 3：

线程块大小	图像大小		
	1024	2048	4096
64	140.342270ms	575.384399ms	2319.340820ms
256	128.460251ms	502.424133ms	2057.175293ms
1024	120.843391ms	473.647583ms	1910.906006ms

注：上述数据的输出结果大小均为(height-2)*(width-2)；共享内存仅考虑(sqrt(N), sqrt(N))的线程块；全局内存的每个条目中从上到下分别为按块划分、按行划分和按列划分。

①从上述结果可以看到，随着图像的增大，卷积的任务量自然增大，时间自然增加；

②使用共享内存的性能高于全局内存的性能，这是因为共享内 存只被线程块内的线程所共享，读写周期要小于全局内存的读写周期，当没有 发生 bank 冲突时，访问共享内存和访问寄存器一样快；

③可以看到不同数据/任务划分方式的性能为：按行划分>按块划分>按列划分，由于此次任务并不像矩阵乘法一样需要一次性访问整行/整列的元素，按行划分可以更好地利用空间局部性，而对空间局部性利用最差的是按列划分，且可以看到随着线程块增大，按列划分的效率是越来越低的，这是因为需要访问更多列，对空间局部性的利用自然更差；

④而线程块大小的影响很明显，除了按列划分，其余两种划分方式都是线程块越大、并行度越大，性能越高。

(由于输出大小固定，所以步长增大时，填充的部分更多，填充后的矩阵更大，计算量也就更大，且对空间局部性的利用也会在一定程度上下降，导致性能也会降低。)

- **实验二：**

■ 编译：

```
nvcc -g -o filename filename.cu
```

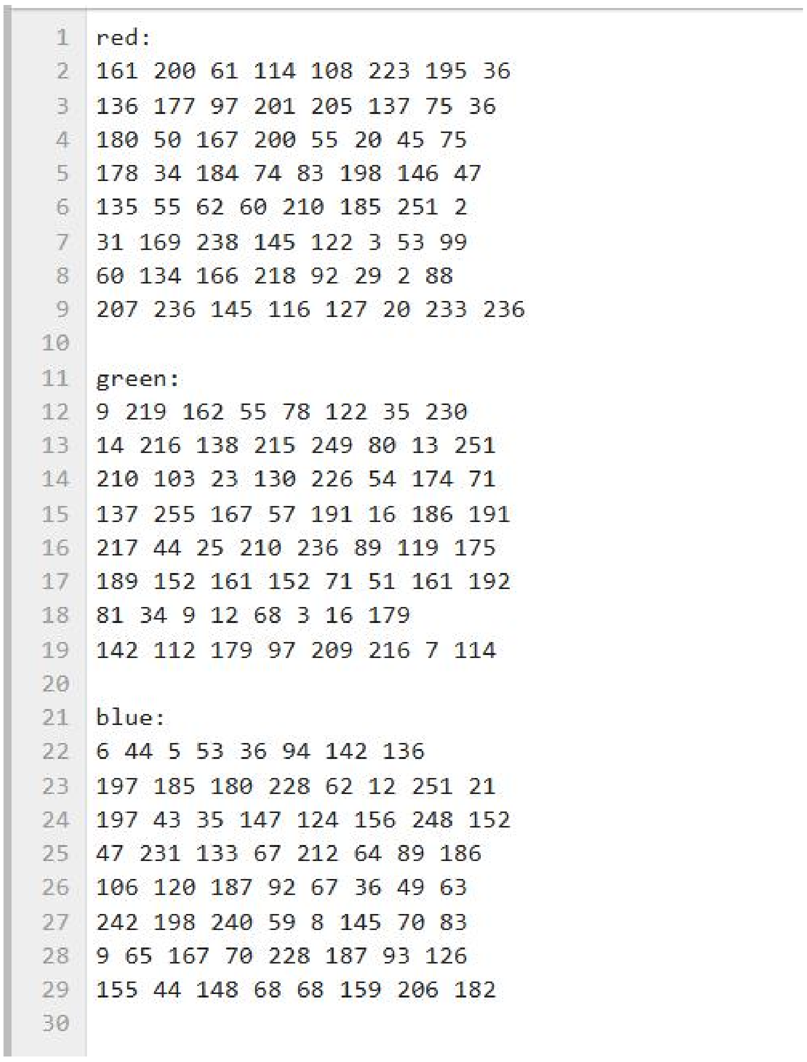
■ 运行：

```
./filename
```

(filename: Image2Col_global.cu/Image2Col_shared.cu，表示使用全局/共享内存的访存方式进行卷积)

■ 运行结果示例，用于展示正确性：

三通道原矩阵：



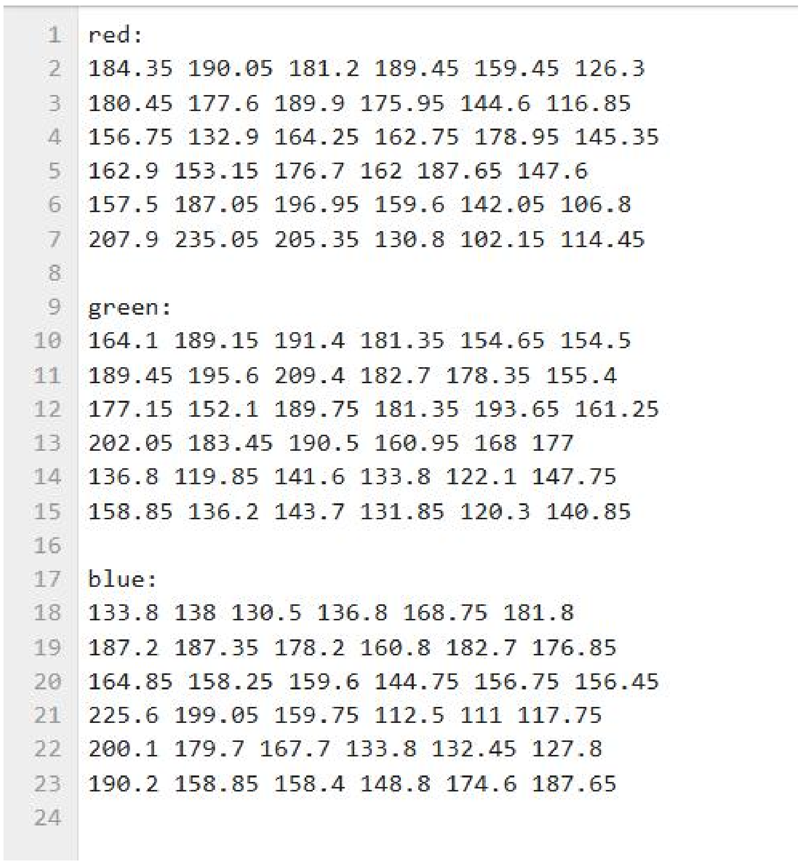
```
red:
161 200 61 114 108 223 195 36
136 177 97 201 205 137 75 36
180 50 167 200 55 20 45 75
178 34 184 74 83 198 146 47
135 55 62 60 210 185 251 2
31 169 238 145 122 3 53 99
60 134 166 218 92 29 2 88
207 236 145 116 127 20 233 236

green:
9 219 162 55 78 122 35 230
14 216 138 215 249 80 13 251
210 103 23 130 226 54 174 71
137 255 167 57 191 16 186 191
217 44 25 210 236 89 119 175
189 152 161 152 71 51 161 192
81 34 9 12 68 3 16 179
142 112 179 97 209 216 7 114

blue:
6 44 5 53 36 94 142 136
197 185 180 228 62 12 251 21
197 43 35 147 124 156 248 152
47 231 133 67 212 64 89 186
106 120 187 92 67 36 49 63
242 198 240 59 8 145 70 83
9 65 167 70 228 187 93 126
155 44 148 68 68 159 206 182

```

运算结果：

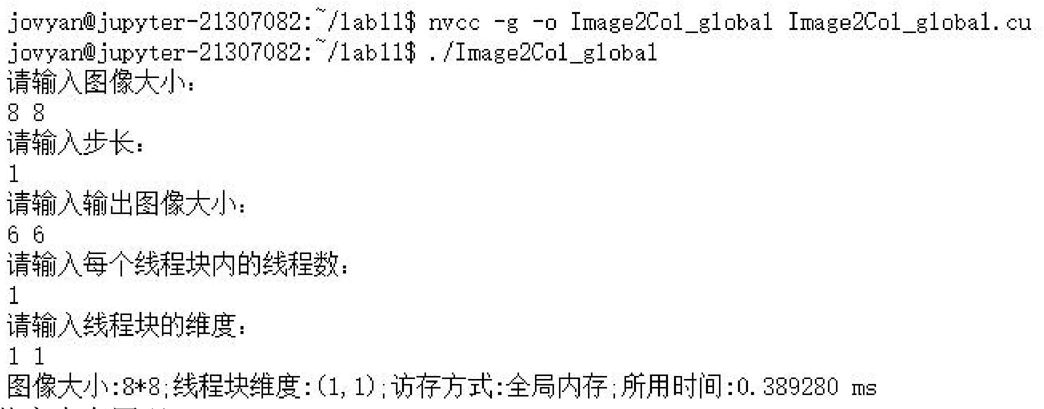


```
red:
184.35 190.05 181.2 189.45 159.45 126.3
180.45 177.6 189.9 175.95 144.6 116.85
156.75 132.9 164.25 162.75 178.95 145.35
162.9 153.15 176.7 162 187.65 147.6
157.5 187.05 196.95 159.6 142.05 106.8
207.9 235.05 205.35 130.8 102.15 114.45

green:
164.1 189.15 191.4 181.35 154.65 154.5
189.45 195.6 209.4 182.7 178.35 155.4
177.15 152.1 189.75 181.35 193.65 161.25
202.05 183.45 190.5 160.95 168 177
136.8 119.85 141.6 133.8 122.1 147.75
158.85 136.2 143.7 131.85 120.3 140.85

blue:
133.8 138 130.5 136.8 168.75 181.8
187.2 187.35 178.2 160.8 182.7 176.85
164.85 158.25 159.6 144.75 156.75 156.45
225.6 199.05 159.75 112.5 111 117.75
200.1 179.7 167.7 133.8 132.45 127.8
190.2 158.85 158.4 148.8 174.6 187.65
```

运算时间：



```
jovyan@jupyter-21307082:~/lab11$ nvcc -g -o Image2Col_global Image2Col_global.cu
jovyan@jupyter-21307082:~/lab11$ ./Image2Col_global
请输入图像大小：
8 8
请输入步长：
1
请输入输出图像大小：
6 6
请输入每个线程块内的线程数：
1
请输入线程块的维度：
1 1
图像大小:8*8;线程块维度:(1,1);访存方式:全局内存;所用时间:0.389280 ms
```

共享内存同理。

■ 实验结果：

全局内存：

stride 为 1：

线程块大小	图像大小		
	1024	2048	4096
64	260.578491ms 143.209824ms 1971.147583ms	1059.809326ms 604.930481ms 7938.947754ms	4107.438477ms 2413.893799ms 31918.988281ms
256	213.073792ms 142.466171ms 2238.211670ms	852.595642ms 564.342468ms 9035.308594ms	3560.818359ms 2349.966064ms 36904.902344ms

1024	169.449051ms 149.608032ms 2518.037354ms	692.001770ms 589.878540ms 10024.487305ms	2861.688232ms 2489.305664ms 39148.308594ms

stride 为 2：

线程块大小	图像大小		
	1024	2048	4096
64	274.409515ms 149.393631ms 2051.020508ms	1099.020874ms 620.615540ms 8430.100586ms	4341.060059ms 2498.575684ms 34239.074219ms
256	210.224731ms 146.601089ms 2275.598389ms	875.212463ms 580.600342ms 9041.789062ms	3498.249268ms 2457.582031ms 35447.890625ms
1024	176.909210ms 149.485123ms 3019.391113ms	717.538208ms 604.311157ms 10760.190430ms	2882.184326ms 2441.761719ms 37739.894531ms

stride 为 3：

线程块大小	图像大小		
	1024	2048	4096
64	291.154114ms 165.820679ms 2035.877075ms	1154.479858ms 664.635071ms 9877.622070ms	5203.737305ms 2990.192139ms 38180.113281ms
256	225.947968ms 156.016678ms 2286.933594ms	1136.022217ms 730.809265ms 12797.850586ms	4348.174316ms 2841.777100ms 42167.484375ms
1024	197.061371ms 155.264923ms 2367.098389ms	927.056458ms 715.068665ms 10601.425781ms	3424.196045ms 2876.175537ms 45300.976562ms

共享内存：

stride 为 1：

线程块大小	图像大小		
	1024	2048	4096
64	254.153625ms	1046.244019ms	4139.612793ms
256	207.952225ms	820.179932ms	3282.431885ms
1024	173.026428ms	709.783875ms	2828.401367ms

stride 为 2：

线程块大小	图像大小		
	1024	2048	4096
64	282.191772ms	1121.588501ms	4372.642578ms
256	220.954941ms	848.408142ms	3362.162354ms
1024	179.049728ms	720.716553ms	2826.308594ms

stride 为 3：

线程块大小	图像大小		
	1024	2048	4096
64	290.644623ms	1110.724121ms	4513.440918ms
256	231.068512ms	909.401855ms	3657.395996ms
1024	181.792130ms	679.374023ms	3063.044434ms

注：上述数据的输出结果大小均为(height-2)*(width-2)；共享内存仅考虑(sqrt(N), sqrt(N))的线程块；全局内存的每个条目中从上到下分别为按块划分、按行划分和按列划分。

可以看到图像大小、访存方式、、任务/数据划分方式以及线程块大小对程序性能的影响和滑窗法相同，不过由于 im2col 方法还有一步将要参与计算的数据转为列向量(该步也进行了计时)，因此所用时间多于滑窗法。

- **实验三：**

■ 编译前需要配置环境变量

```
export PATH=//usr/local/cuda-10.2/bin:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-10.2/lib64/
```

■ 编译：

```
nvcc -g -o cuDNN_Conv cuDNN_Conv.cu -I ~/cudnn-linux-x86_64-8.7.0.84_cuda10-archive/include -L ~/cudnn-linux-x86_64-8.7.0.84_cuda10-archive/lib -lcudnn
```

■ 运行：

```
./cuDNN_Conv
```

■ 实验结果：

Stride	图像大小		
	1024	2048	4096
1	4118.203125ms	4246.408203ms	5003.994629ms
2	4073.874512ms	4239.855957ms	4978.881836ms
3	4062.940918ms	4237.980469ms	4971.054199ms

可以看到，使用 cuDNN 的性能不如自己实现的卷积操作。可能的改进方法如下：

①由于是在主机上为矩阵分配空间，因此在进行卷积运算时会有主机传到 GPU 上的内存传输，可以考虑优化内存传输；

②尽可能更好地利用空间局部性和时间局部性；

③使用更高效的单精度浮点乘法；

④通过调整每个线程处理的数据量以及 block 的大小和维度，使得数据能够均衡负载。

4. 实验感想（300 字左右）

在此次实验中，通过前面两个实验，我对滑窗法实现卷积和使用 im2col 方法进行卷积的理解更加深入，对 CUDA 编程也更加熟悉。此外，我还初步学习了解了如何使用 cuDNN 进行卷积以及使用 cuDNN 进行卷积的性能与自己手动实现的卷积的性能上的差异。

在本次实验中，由于是初次使用 cuDNN，因此在进行安装和环境配置以及编译时遇到了一点小困难，不过在网上查看相关参考资料后解决了问题。