中山大学计算机院本科生实验报告
（2024 学年春季学期）

课程名称：并行程序设计　　批改人：

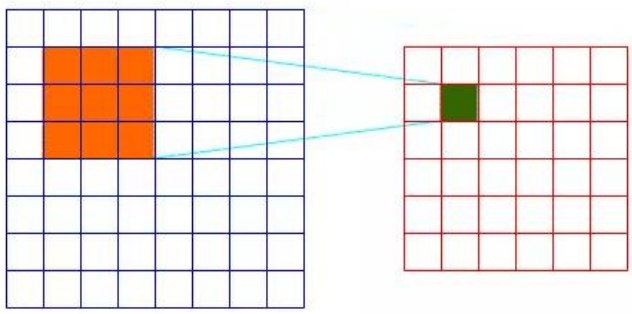
实验	11-CUDA 卷积	专业（方向）	计算机科学与技术
学号	21307174	姓名	刘俊杰
Email	liujj255@mail2.sysu.edu.cn	完成日期	2024/6/8

1. 实验目的

1.1 滑窗法实现 CUDA 并行卷积

使用 CUDA 实现二维图像的直接卷积（滑窗法）。在信号处理、图像处理和其他工程/科学领域，卷积是一种使用广泛的技术。在深度学习领域，卷积神经网络(CNN)这种模型架构就得名于这种技术。在本实验中，我们将在 GPU 上实现卷积操作，注意这里的卷积是指神经网络中的卷积操作，与信号处理领域中的卷积操作不同，它不需要对 Filter 进行翻转，不考虑 bias。

下图展示了滑窗法实现的 CUDA 卷积，其中蓝色网格表示输入图像，红色网格表示输出图像，橙色网格展示了一个3×3的卷积核，卷积核中每个元素为对应位置像素的权重，该卷积核的输出值为像素值的加权和，输出位置位于橙色网格中心，即红色网格中的绿色元素。滑窗法移动该卷积核的中心，从而产生红色网格中的所有元素。



输入：一张二维图像（$height \times width$）与一个卷积核（3×3）。

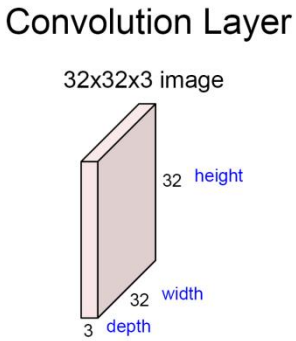
问题描述：用直接卷积的方式对输入二维图像进行卷积，通道数量（channel，depth）设置为 3，卷积核个数为 3，步幅（stride）分别设置为 1/2/3，可能需要通过填充（padding）配合步幅（stride）完成卷积操作。注：实验的卷积操作不需要考虑 bias（b），bias 设置为 0。

输出：卷积结果图像（$\boldsymbol{height-2\times width-2}$）及计算时间。

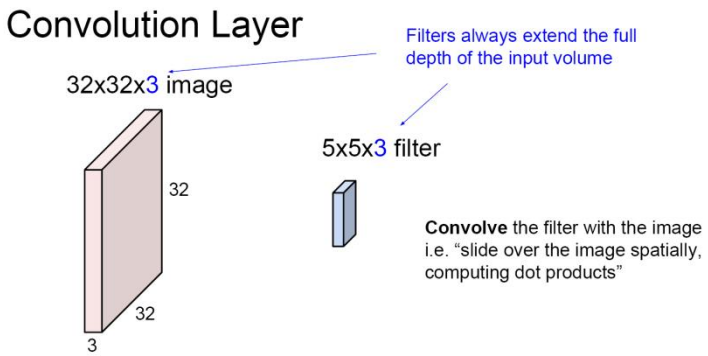
要求：使用 CUDA 实现并行图像卷积，分析不同图像大小、访存方式、任务/数据划分方式、线程块大小等因素对程序性能的影响。

参考：

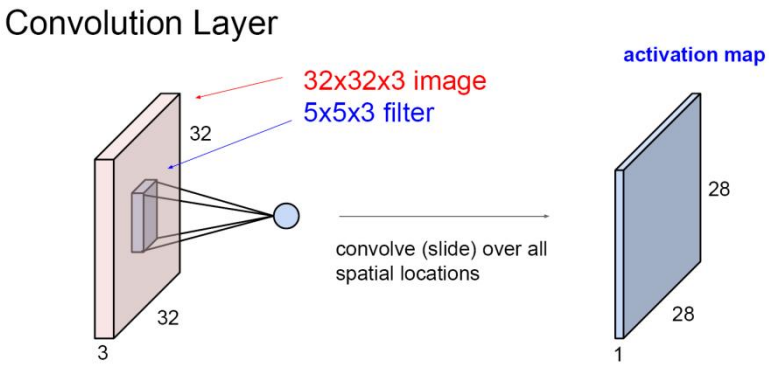
1）输入图像举例



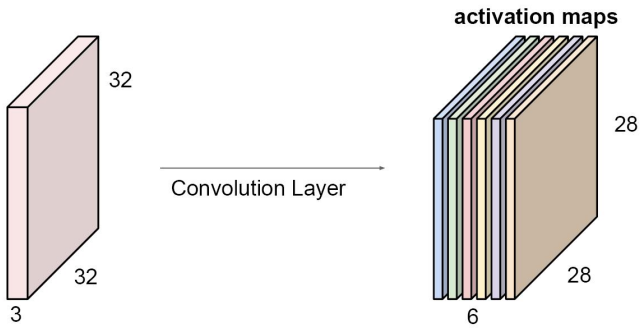
2）输入图像与卷积核（3 通道）



3）卷积计算过程



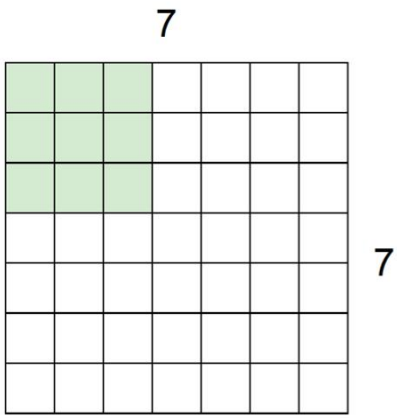
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

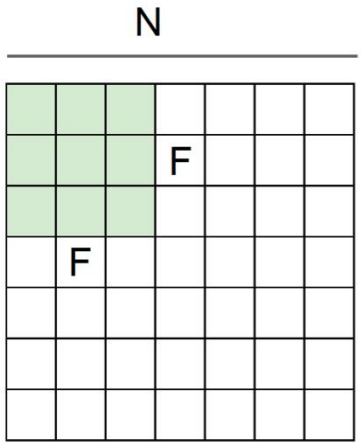
4）卷积操作的步幅（stride）与填充（padding）

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

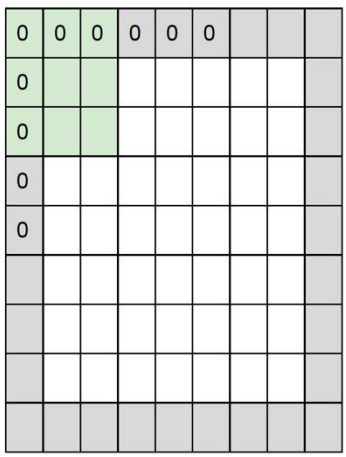
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33

In practice: Common to zero pad the border

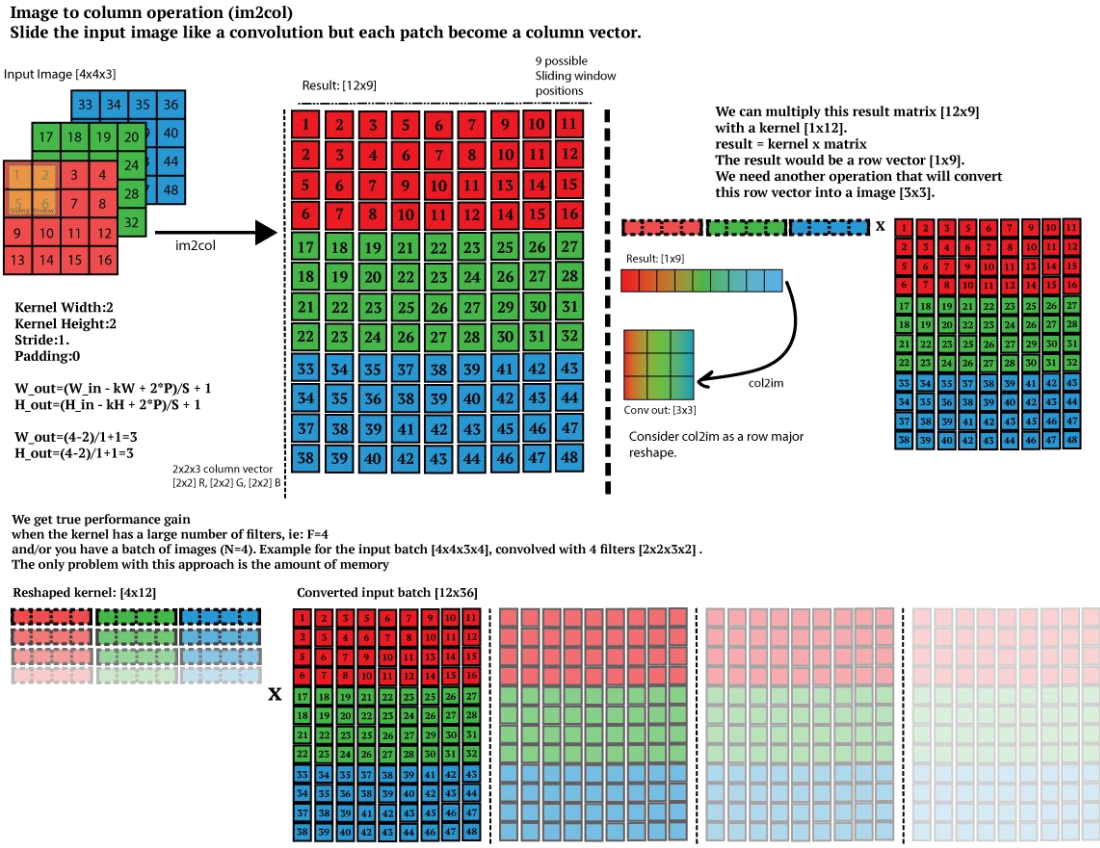


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

1.2 使用 im2col 方法实现 CUDA 并行卷积

滑窗法使用3×3的卷积核对3×3窗口内的图像像素求加权和，此过程可以写做矩阵乘法形式$\boldsymbol{w}^T \cdot \boldsymbol{x}$，其中$\boldsymbol{w}^T$为$1 \times 9$的权重矩阵，$\boldsymbol{x}$为$9 \times 1$的像素值矩阵。将图像中每个需要进行卷积的窗口平铺为9×1的矩阵（列向量）并进行拼接，可将卷积计算变为矩阵乘法，从而利用此前实现的并行矩阵乘法模块实现并行卷积。具体拼接方式见下图：



问题描述：用 im2col 方法对输入二维图像进行卷积。其他设置与任务 1（滑窗法并行卷积）相同。

1.3 使用 cuDNN 方法实现 CUDA 并行卷积

NVIDIA cuDNN 是用于深度神经网络的 GPU 加速库。它强调性能、易用性和低内存开销。

要求：使用 cuDNN 提供的卷积方法进行卷积操作，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

2.1 滑窗法实现 CUDA 并行卷积

2.1.1 全局内存

使用 CUDA 的并行计算能力，每个线程负责计算输出特征图中的一个像素值。通过 CUDA 提供的线程和线程块索引，可以实现高效的并行卷积运算。

实现代码：

```
// 2D卷积核函数
__global__ void conv2d_global(float* input, float* kernel, float* output) {
    // 计算输出的高度和宽度
    int outHeight = (inputHeight - kernelSize) / stride + 1;
    int outWidth = (inputWidth - kernelSize) / stride + 1;

    // 计算线程的全局索引
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 如果线程的索引在输出范围内，执行卷积操作
    if (row < outHeight && col < outWidth) {
        for (int kn = 0; kn < kernelNum; ++kn) {
            float sum = 0.0f; // 初始化卷积和为0
            // 对每个输入通道和卷积核进行迭代
            for (int kc = 0; kc < inputChannels; ++kc) {
                for (int i = 0; i < kernelSize; ++i) {
                    for (int j = 0; j < kernelSize; ++j) {
                        int r = row * stride + i; // 计算输入的行索引
                        int c = col * stride + j; // 计算输入的列索引
                        // 计算卷积和
                        sum += input[(kc * inputHeight + r) * inputWidth + c] *
                               kernel[((kn * inputChannels + kc) * kernelSize + i) * kernelSize + j];
                    }
                }
            }
            // 将计算结果存储到输出数组
            output[(kn * outHeight + row) * outWidth + col] = sum;
        }
    }
}
```

①计算输出的高度和宽度：

```
// 计算输出的高度和宽度
int outHeight = (inputHeight - kernelSize) / stride + 1;
int outWidth = (inputWidth - kernelSize) / stride + 1;
```

`outHeight` 和 `outWidth` 分别表示卷积操作的输出特征图的高度和宽度。

②计算线程的全局索引：

```
// 计算线程的全局索引
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

row 和 col 是当前线程在输出特征图中的全局索引。blockIdx.y, blockDim.y, threadIdx.y, blockIdx.x, blockDim.x, 和 threadIdx.x 是 CUDA 提供的内置变量，用于描述线程和线程块的索引和大小。

③执行卷积操作：

```
// 如果线程的索引在输出范围内，执行卷积操作
if (row < outHeight && col < outWidth) {
    for (int kn = 0; kn < kernelNum; ++kn) {
        float sum = 0.0f; // 初始化卷积和为0
        // 对每个输入通道和卷积核进行迭代
        for (int kc = 0; kc < inputChannels; ++kc) {
            for (int i = 0; i < kernelSize; ++i) {
                for (int j = 0; j < kernelSize; ++j) {
                    int r = row * stride + i; // 计算输入的行索引
                    int c = col * stride + j; // 计算输入的列索引
                    // 计算卷积和
                    sum += input[(kc * inputHeight + r) * inputWidth + c] *
                           kernel[((kn * inputChannels + kc) * kernelSize + i) * kernelSize + j];
                }
            }
        }
        // 将计算结果存储到输出数组
        output[(kn * outHeight + row) * outWidth + col] = sum;
    }
}
```

首先判断当前线程的索引是否在输出特征图的范围内。
如果在范围内，开始对每个卷积核进行卷积操作。
对每个输入通道（inputChannels）和卷积核（kernel）进行迭代计算。
计算每个输出元素的值 sum，并存储到输出数组（output）的相应位置。

2.1.2 共享内存

使用 CUDA 的共享内存来加速二维卷积操作。通过每个线程块加载输入数据到共享内存中，可以减少全局内存访问延迟，从而提高卷积运算的效率。同时，使用 __syncthreads() 来确保共享内存中的数据加载完成后再进行计算，避免数据竞争和错误的结果。

实现代码：

```
// 2D卷积核函数（共享内存版本）
__global__ void conv2d_shared(float* input, float* kernel, float* output) {
    // 计算输出的高度和宽度
    int outHeight = (inputHeight - kernelSize) / stride + 1;
    int outWidth = (inputWidth - kernelSize) / stride + 1;

    // 计算线程的全局索引
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 定义共享内存
    extern __shared__ float sharedMem[];

    // 共享内存中每个线程块加载的数据块大小
    int sharedMemWidth = blockDim.x * stride + kernelSize - 1;
    int sharedMemHeight = blockDim.y * stride + kernelSize - 1;
    int sharedMemSize = sharedMemWidth * sharedMemHeight;

    // 计算当前线程在共享内存中的索引
    int localRow = threadIdx.y * stride;
    int localCol = threadIdx.x * stride;

    // 每个线程块加载其共享内存区域中的所有数据
    for (int kc = 0; kc < inputChannels; ++kc) {
        for (int i = threadIdx.y; i < sharedMemHeight; i += blockDim.y) {
            for (int j = threadIdx.x; j < sharedMemWidth; j += blockDim.x) {
                int r = blockIdx.y * blockDim.y * stride + i;
                int c = blockIdx.x * blockDim.x * stride + j;
                if (r < inputHeight && c < inputWidth) {
                    sharedMem[(kc * sharedMemSize) + i * sharedMemWidth + j] =
                        input[(kc * inputHeight + r) * inputWidth + c];
                } else {
                    sharedMem[(kc * sharedMemSize) + i * sharedMemWidth + j] = 0.0f;
                }
            }
        }
    }

    // 等待所有线程完成共享内存加载
```

```
    __syncthreads();

    // 如果线程的索引在输出范围内，执行卷积操作
    if (row < outHeight && col < outWidth) {
        for (int kn = 0; kn < kernelNum; ++kn) {
            float sum = 0.0f; // 初始化卷积和为0
            // 对每个输入通道和卷积核进行迭代
            for (int kc = 0; kc < inputChannels; ++kc) {
                for (int i = 0; i < kernelSize; ++i) {
                    for (int j = 0; j < kernelSize; ++j) {
                        int r = localRow + i; // 计算共享内存的行索引
                        int c = localCol + j; // 计算共享内存的列索引
                        // 计算卷积和
                        sum += sharedMem[(kc * sharedMemSize) + r * sharedMemWidth + c] *
                               kernel[((kn * inputChannels + kc) * kernelSize + i) * kernelSize + j];
                    }
                }
            }
            // 将计算结果存储到输出数组
            output[(kn * outHeight + row) * outWidth + col] = sum;
        }
    }
}
```

①计算输出的高度和宽度：

```
// 计算输出的高度和宽度
int outHeight = (inputHeight - kernelSize) / stride + 1;
int outWidth = (inputWidth - kernelSize) / stride + 1;
```

outHeight 和 outWidth 表示卷积操作的输出特征图的高度和宽度。

②计算线程的全局索引：

```
// 计算线程的全局索引
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

row 和 col 是当前线程在输出特征图中的全局索引。

③定义共享内存：

```
// 定义共享内存
extern __shared__ float sharedMem[];
```

④计算共享内存中的数据块大小：

```
// 共享内存中每个线程块加载的数据块大小
int sharedMemWidth = blockDim.x * stride + kernelSize - 1;
int sharedMemHeight = blockDim.y * stride + kernelSize - 1;
int sharedMemSize = sharedMemWidth * sharedMemHeight;
```

sharedMemWidth 和 sharedMemHeight 分别表示共享内存中每个线程块加载的数据块的宽度和高度。

sharedMemSize 是共享内存的总大小。

⑤计算当前线程在共享内存中的索引：

```
// 计算当前线程在共享内存中的索引
int localRow = threadIdx.y * stride;
int localCol = threadIdx.x * stride;
```

localRow 和 localCol 是当前线程在共享内存中的局部索引。

⑥加载数据到共享内存：

```
// 每个线程块加载其共享内存区域中的所有数据
for (int kc = 0; kc < inputChannels; ++kc) {
    for (int i = threadIdx.y; i < sharedMemHeight; i += blockDim.y) {
        for (int j = threadIdx.x; j < sharedMemWidth; j += blockDim.x) {
            int r = blockIdx.y * blockDim.y * stride + i;
            int c = blockIdx.x * blockDim.x * stride + j;
            if (r < inputHeight && c < inputWidth) {
                sharedMem[(kc * sharedMemSize) + i * sharedMemWidth + j] =
                    input[(kc * inputHeight + r) * inputWidth + c];
            } else {
                sharedMem[(kc * sharedMemSize) + i * sharedMemWidth + j] = 0.0f;
            }
        }
    }
}
```

每个线程块负责加载其共享内存区域中的所有数据块。

blockIdx.y * blockDim.y * stride + i 和 blockIdx.x * blockDim.x * stride + j 计算输入矩阵中的索引，并检查是否超出输入矩阵的边界。

⑦等待所有线程完成共享内存加载：

```
// 等待所有线程完成共享内存加载
__syncthreads();
```

使用 __syncthreads() 同步同一个线程块内的所有线程，确保共享内存中的数据加载完成。

⑧执行卷积操作：

```
// 如果线程的索引在输出范围内，执行卷积操作
if (row < outHeight && col < outWidth) {
    for (int kn = 0; kn < kernelNum; ++kn) {
        float sum = 0.0f; // 初始化卷积和为0
        // 对每个输入通道和卷积核进行迭代
        for (int kc = 0; kc < inputChannels; ++kc) {
            for (int i = 0; i < kernelSize; ++i) {
                for (int j = 0; j < kernelSize; ++j) {
                    int r = localRow + i; // 计算共享内存的行索引
                    int c = localCol + j; // 计算共享内存的列索引
                    // 计算卷积和
                    sum += sharedMem[(kc * sharedMemSize) + r * sharedMemWidth + c] *
                           kernel[((kn * inputChannels + kc) * kernelSize + i) * kernelSize + j];
                }
            }
        }
        // 将计算结果存储到输出数组
        output[(kn * outHeight + row) * outWidth + col] = sum;
    }
}
```

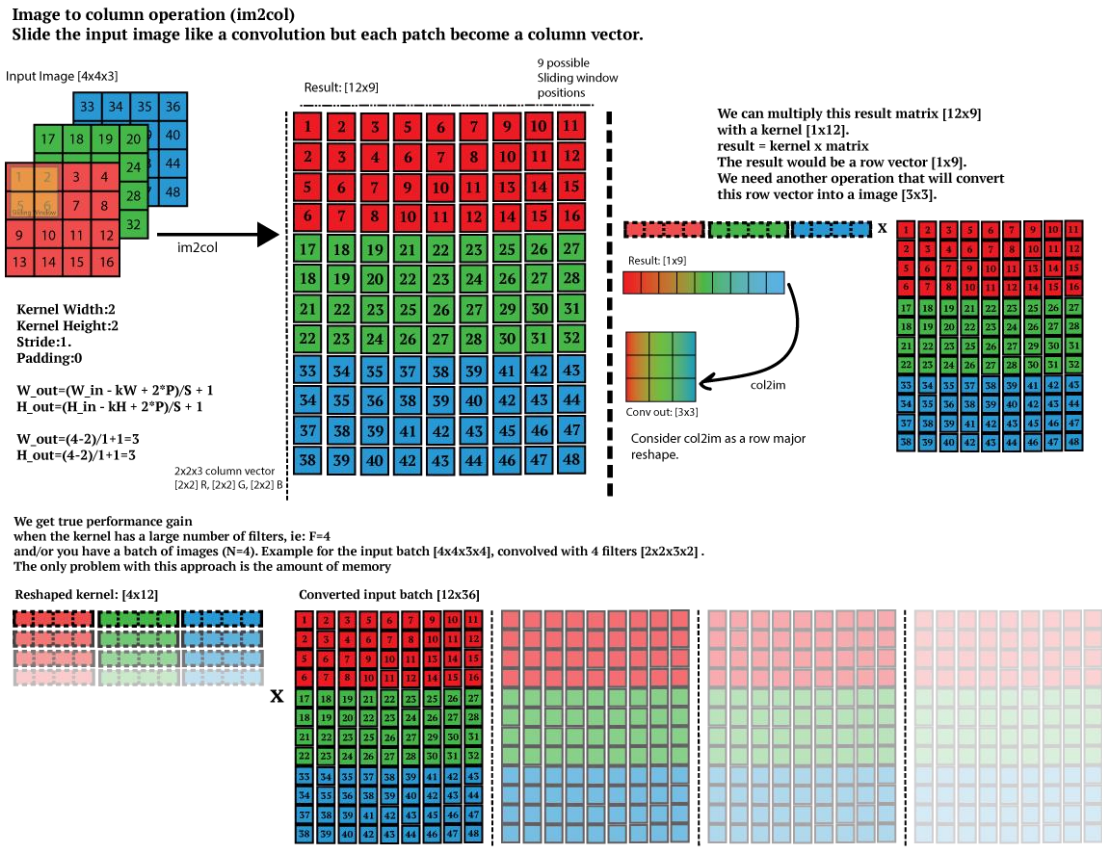
首先检查当前线程的索引是否在输出范围内。

对每个卷积核进行卷积操作，使用共享内存中的数据加速计算。

将计算结果存储到输出数组中。

2.2 使用 im2col 方法实现 CUDA 并行卷积

2.2.1 实现 im2col



im2col 将图像块展开为列向量，使得数据在内存中是连续的。这种数据布局有助于缓存的利用，提高内存访问效率。相比之下，直接在图像上滑动滤波器会导致不规则的内存访问，影响性能。但由于 C 语言是按行存储,故实际上 im2col 的结果的存储是按上图中的转置存储的。

实现代码:

根据 im2col 后的结果实现卷积:

```
float* im_to_col(float*input,int outputHeight,int outputWidth){
    size_t row = outputHeight * outputWidth ;
    size_t col = kernelSize * kernelSize * inputChannels;
    float *im2col = (float*)malloc(row * col * sizeof(float));
    for(int i = 0; i < outputHeight ; i++ ){
        for(int j = 0 ; j < outputWidth ; j++){
            int n = 0 ;
            for(int c = 0 ;c < inputChannels ; c++){
                    for(int k1 = 0; k1 < kernelSize ;k1++){
                        for(int k2 = 0 ;k2 < kernelSize ;k2++){
                            int startRow = i * stride;
                            int startCol = j * stride;
                            int rindex = startRow + k1;
                            int cindex = startCol +k2;
                            im2col[ (i * outputWidth +j) * col  + n] =
                                input[c * inputHeight * inputWidth + rindex * inputWidth + cindex];
                                n++;
                        }
                    }
                }
            }

            }

    return im2col;
}
```

2.2.2 im2col 卷积

```cpp
// 2D卷积核函数
__global__ void conv2d_global(float* im2col, float* kernel, float* output) {
    // 计算输出的高度和宽度
    int outHeight = (inputHeight - kernelSize) / stride + 1;
    int outWidth = (inputWidth - kernelSize) / stride + 1;

    // 计算线程的全局索引
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // 如果线程的索引在输出范围内，执行卷积操作
    if (row < outHeight && col < outWidth) {
        //printf("%d %d\n",row,col);
        for (int kn = 0; kn < kernelNum; ++kn) {
            float sum = 0.0f; // 初始化卷积和为0
            // 对每个输入通道和卷积核进行迭代
            int n = 0;
            for (int kc = 0; kc < inputChannels; ++kc) {

                for (int i = 0; i < kernelSize; ++i) {
                    for (int j = 0; j < kernelSize; ++j) {
                        int length = inputChannels * kernelSize * kernelSize;

                        // 计算卷积和
                        sum += im2col[(row * outWidth + col) * length + n] *
                               kernel[((kn * inputChannels + kc) * kernelSize + i) * kernelSize + j];
                        n++;

                    }
                }

            }
            // 将计算结果存储到输出数组
            output[(kn * outHeight + row) * outWidth + col] = sum;
        }
    }
}
```

2.3 使用 cuDNN 方法实现 CUDA 并行卷积

实现代码：

```cpp
#include <iostream>
#include <cudnn.h>
#include <cuda_runtime.h>
#include <chrono>

// CUDA 错误检查
#define CHECK_CUDA(call) \
    do { \
        cudaError_t err = call; \
        if (err != cudaSuccess) { \
            std::cerr << "CUDA error: " << cudaGetErrorString(err) << std::endl; \
            std::exit(EXIT_FAILURE); \
        } \
    } while (0)
// cuDNN 错误检查
#define CHECK_CUDNN(call) \
    do { \
        cudnnStatus_t err = call; \
        if (err != CUDNN_STATUS_SUCCESS) { \
            std::cerr << "cuDNN error: " << cudnnGetErrorString(err) << std::endl; \
            std::exit(EXIT_FAILURE); \
        } \
    } while (0)
```

```cpp
int main() {
    // 初始化 cuDNN
    cudnnHandle_t cudnn;
    CHECK_CUDNN(cudnnCreate(&cudnn));
    // 输入特征图参数
    const int batch_size = 1;
    const int channels = 3;
    const int height = 5;
    const int width = 5;
    // 卷积核参数
    const int kernel_size = 3;
    const int kernel_channels = 3;
    const int num_kernels = 2;
    // 初始化输入数据和卷积核为 1
    float input[batch_size * channels * height * width];
    float kernel[num_kernels * kernel_channels * kernel_size * kernel_size];
    std::fill_n(input, batch_size * channels * height * width, 1.0f);
    std::fill_n(kernel, num_kernels * kernel_channels * kernel_size * kernel_size, 1.0f);
    // 分配 GPU 内存
    float *d_input, *d_kernel, *d_output;
    CHECK_CUDA(cudaMalloc(&d_input, sizeof(input)));
    CHECK_CUDA(cudaMalloc(&d_kernel, sizeof(kernel)));
    // 输出特征图尺寸
    int output_height = height - kernel_size + 1;
    int output_width = width - kernel_size + 1;
    CHECK_CUDA(cudaMalloc(&d_output, batch_size * num_kernels * output_height * output_width * sizeof(float)));
    // 将数据从主机复制到设备
    CHECK_CUDA(cudaMemcpy(d_input, input, sizeof(input), cudaMemcpyHostToDevice));
    CHECK_CUDA(cudaMemcpy(d_kernel, kernel, sizeof(kernel), cudaMemcpyHostToDevice));
    // 创建输入特征图描述符
    cudnnTensorDescriptor_t input_desc;
    CHECK_CUDNN(cudnnCreateTensorDescriptor(&input_desc));
    CHECK_CUDNN(cudnnSetTensor4dDescriptor(input_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, batch_size, channels, height, width));
    // 创建卷积核描述符
    cudnnFilterDescriptor_t kernel_desc;
    CHECK_CUDNN(cudnnCreateFilterDescriptor(&kernel_desc));
    CHECK_CUDNN(cudnnSetFilter4dDescriptor(kernel_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, num_kernels, kernel_channels, kernel_size,
kernel_size));
    // 创建卷积描述符
    cudnnConvolutionDescriptor_t conv_desc;
    CHECK_CUDNN(cudnnCreateConvolutionDescriptor(&conv_desc));
    CHECK_CUDNN(cudnnSetConvolution2dDescriptor(conv_desc, 0, 0, 1, 1, 1, 1, CUDNN_CONVOLUTION, CUDNN_DATA_FLOAT));
    // 创建输出特征图描述符
    cudnnTensorDescriptor_t output_desc;
    CHECK_CUDNN(cudnnCreateTensorDescriptor(&output_desc));
    CHECK_CUDNN(cudnnSetTensor4dDescriptor(output_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, batch_size, num_kernels, output_height,
output_width));
    // 卷积算法选择
    cudnnConvolutionFwdAlgo_t conv_algo;
    CHECK_CUDNN(cudnnGetConvolutionForwardAlgorithm(cudnn, input_desc, kernel_desc, conv_desc, output_desc, CUDNN_CONVOLUTION_FWD_PREFER_FASTEST,
0, &conv_algo));
    // 分配工作空间
    size_t workspace_size;
    CHECK_CUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn, input_desc, kernel_desc, conv_desc, output_desc, conv_algo, &workspace_size));
    void *d_workspace;
    CHECK_CUDA(cudaMalloc(&d_workspace, workspace_size));
    // 记录开始时间
    auto start = std::chrono::high_resolution_clock::now();
    // 执行卷积
    const float alpha = 1.0f, beta = 0.0f;
    CHECK_CUDNN(cudnnConvolutionForward(cudnn, &alpha, input_desc, d_input, kernel_desc, d_kernel, conv_desc, conv_algo, d_workspace, workspace_size
&beta, output_desc, d_output));
    // 记录结束时间
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<float, std::milli> duration_ms = end - start;
    // 将结果从设备复制到主机
    float output[batch_size * num_kernels * output_height * output_width];
    CHECK_CUDA(cudaMemcpy(output, d_output, sizeof(output), cudaMemcpyDeviceToHost));
    // 打印输出结果
    std::cout << "Output:\n";
    for (int i = 0; i < batch_size * num_kernels * output_height * output_width; ++i) {
        std::cout << output[i] << " ";
        if ((i + 1) % output_width == 0) std::cout << "\n";
        if ((i + 1) % (output_height * output_width) == 0) std::cout << "\n";
    }
    // 打印卷积时间
    std::cout << "Convolution time: " << duration_ms.count() << " ms\n";
    // 清理资源
    CHECK_CUDA(cudaFree(d_input));
    CHECK_CUDA(cudaFree(d_kernel));
    CHECK_CUDA(cudaFree(d_output));
    CHECK_CUDA(cudaFree(d_workspace));
    CHECK_CUDNN(cudnnDestroyTensorDescriptor(input_desc));
    CHECK_CUDNN(cudnnDestroyFilterDescriptor(kernel_desc));
    CHECK_CUDNN(cudnnDestroyConvolutionDescriptor(conv_desc));
    CHECK_CUDNN(cudnnDestroyTensorDescriptor(output_desc));
    CHECK_CUDNN(cudnnDestroy(cudnn));
```

```
    return 0;
}
```

3. 实验结果

3.1 滑窗法实现 CUDA 并行卷积

3.1.1 验证实验正确性

(输入和卷积核中的数据全为 1)

输入输出、卷积层、网格、线程块参数:

```
Input Channels: 3
Input Dimensions: 5x5
Kernel Size: 3
Kernel Dimensions: 3x3
Kernel Number: 3
Output Channels: 3
Output Dimensions: 3x3
Stride: 1
Thread Block Size: 16x16
Grid Size: 1x1
```

输入数据:

```
The input:

The 1th channel of input:
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
The 2th channel of input:
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
The 3th channel of input:
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
```

卷积核 Kernel:

```
The 1th kernel:
The 1th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 2th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 3th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000

The 2th kernel:
The 1th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 2th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 3th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000

The 3th kernel:
The 1th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 2th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
The 3th channel of the kernel:
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
1.000000 1.000000 1.000000
```

全局内存卷积结果:

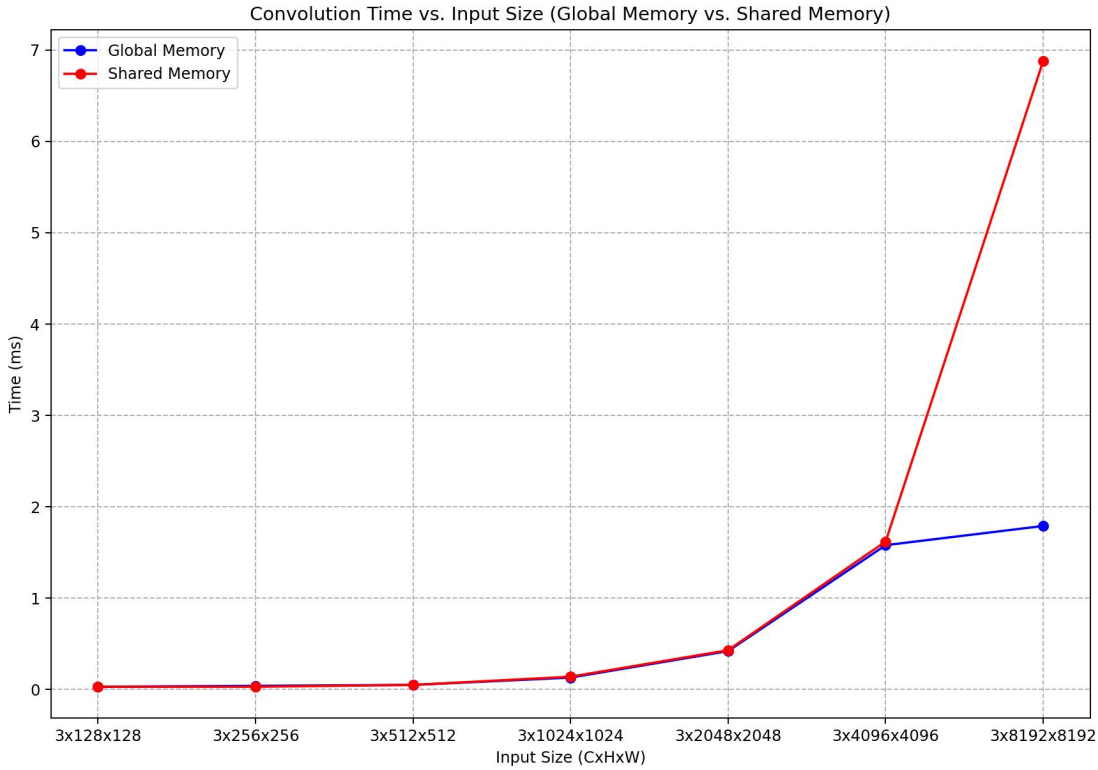
```
The output:

Channel 1 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 2 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 3 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
```

共享内存卷积结果

```
The output:

Channel 1 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 2 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 3 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
```

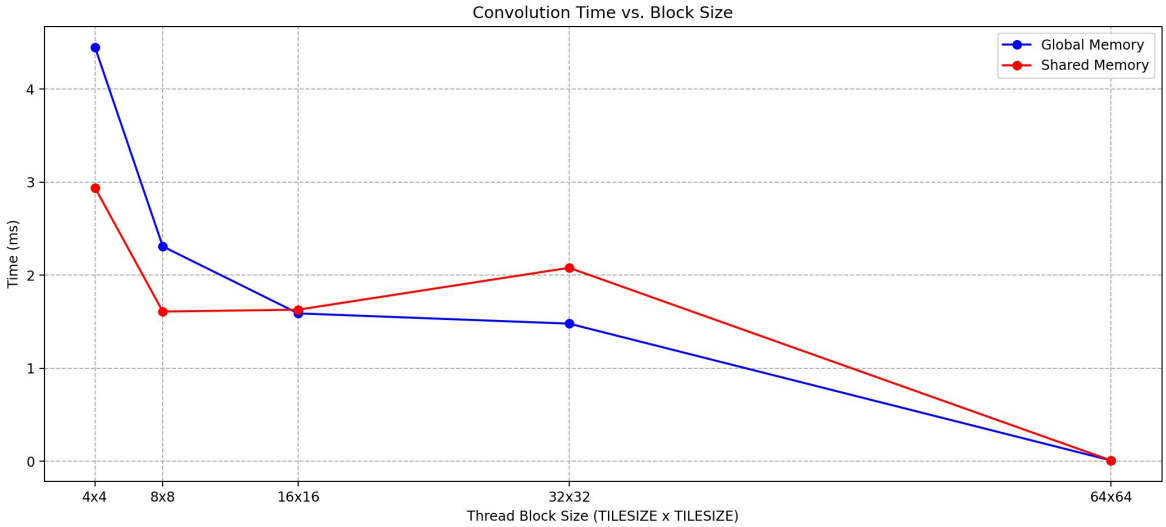
3.1.2 不同矩阵维度全局内存和共享内存的对比

输入尺寸 (Input Dimensions)	输出尺寸 (Output Dimensions)	全局内存时间 (ms)	共享内存时间 (ms)
3x128x128	3x126x126	0.03	0.03
3x256x256	3x254x254	0.04	0.03
3x512x512	3x510x510	0.05	0.05
3x1024x1024	3x1022x1022	0.13	0.14
3x2048x2048	3x2046x2046	0.42	0.43
3x4096x4096	3x4094x4094	1.58	1.62
3x8192x8192	3x8190x8190	1.79	6.88



3.1.3 不同线程块大小对程序性能的影响

线程块大小 (TILESIZE x TILESIZE)	全局内存时间 (ms)	共享内存时间 (ms)
4x4	4.45	2.94
8x8	2.31	1.61
16x16	1.59	1.63
32x32	1.48	2.08
64x64	0.01	0.01



可以看到随着线程块的增大,两种方式的运行时间都减少了,程序性能都得到了提升

3.2 使用 im2col 方法实现 CUDA 并行卷积

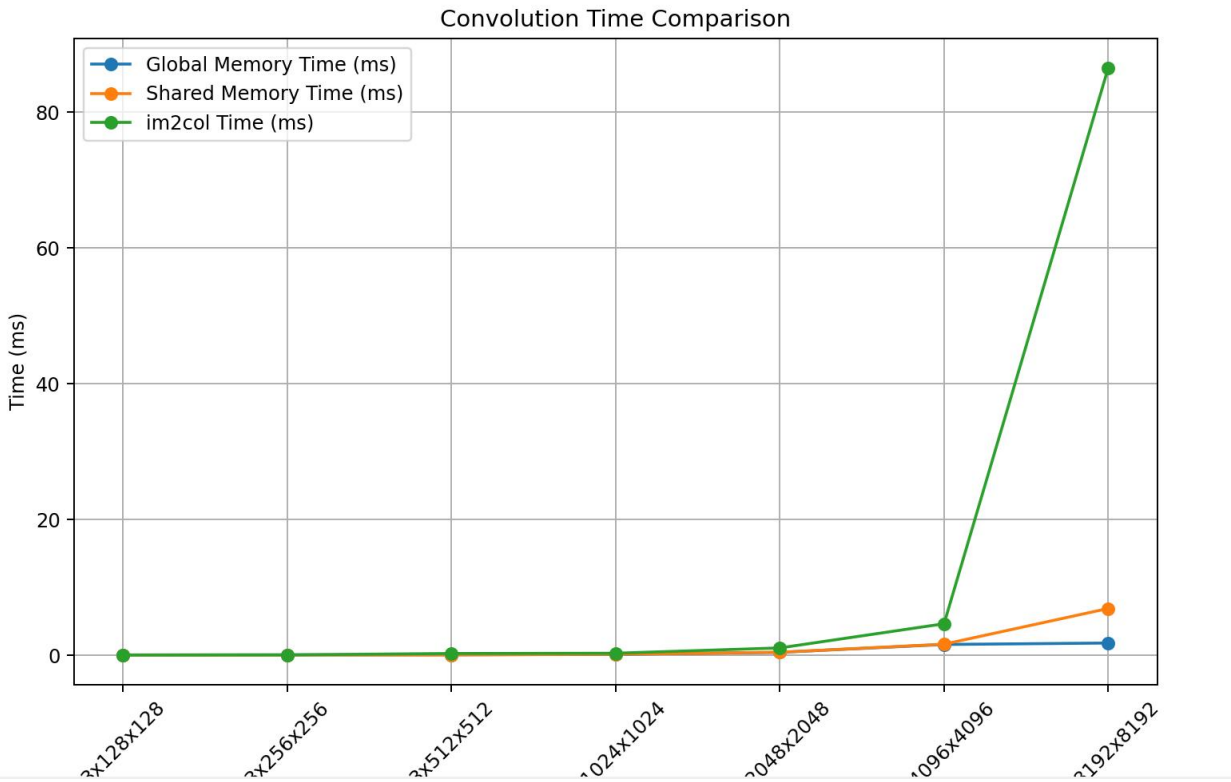
3.2.1 验证 im2col 卷积正确性

(输入和卷积核中的数据全为 1)

```
jovyan@jupyter-21307174:~/1jj$ nvcc -o im2col im2col.cu
im2col.cu(177): warning: variable "x" was declared but never referenced

jovyan@jupyter-21307174:~/1jj$ ./im2col
Convolution Layer Parameters:
Input Channels: 3
Input Dimensions: 5x5
Kernel Size: 3
Kernel Dimensions: 3x3
Kernel Number: 3
Output Channels: 3
Output Dimensions: 3x3
Stride: 1
Thread Block Size: 16x16
Grid Size: 1x1

The output:

Channel 1 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 2 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000

Channel 3 of the output:
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000
27.000000 27.000000 27.000000



Time for  convolution: 0.03 ms
jovyan@jupyter-21307174:~/1jj$
```

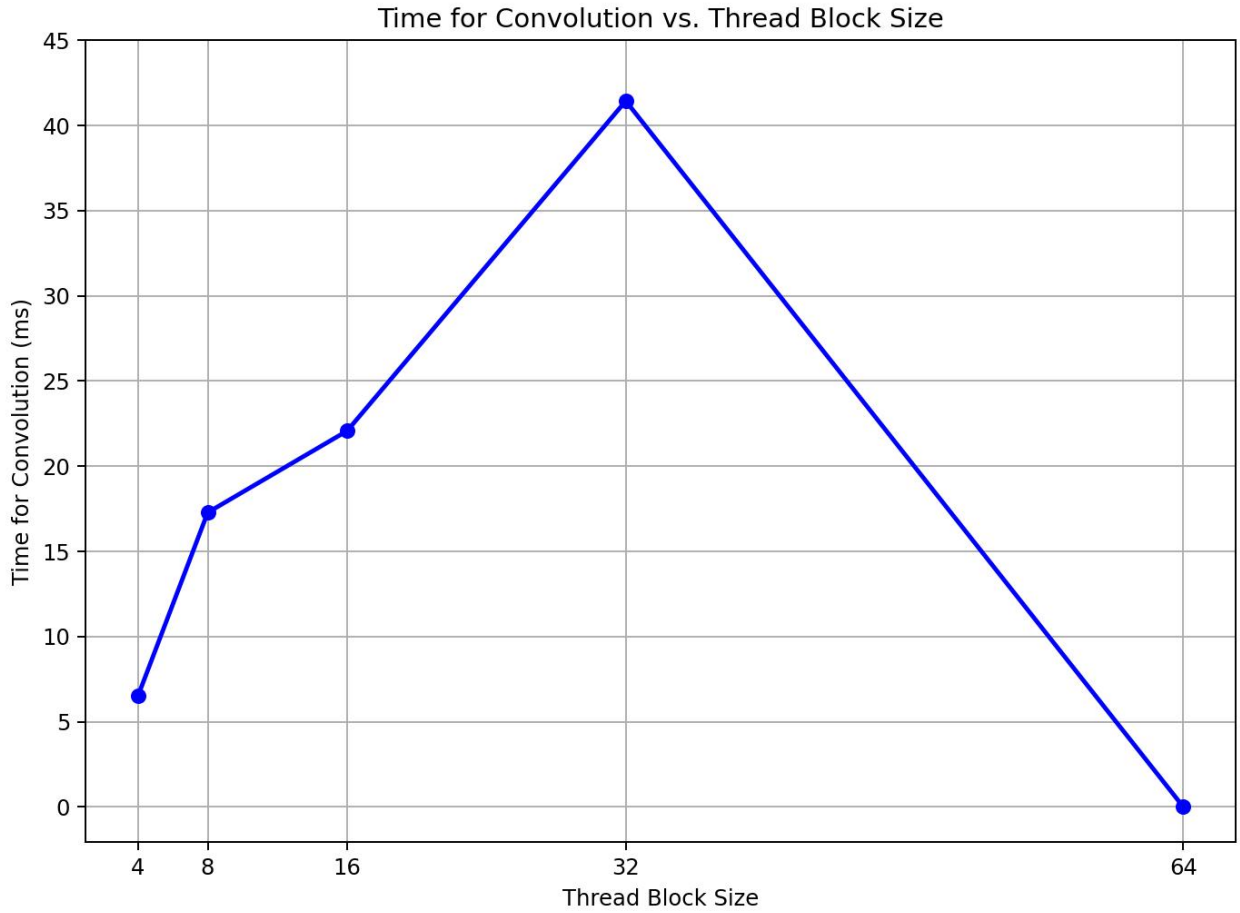
3.2.2 im2col 卷积时间随矩阵规模变化(与滑窗法对比)

输入尺寸 (Input Dimensions)	输出尺寸 (Output Dimensions)	全局内存时间 (ms)	共享内存时间 (ms)	im2col 时间 (ms)
3x128x128	3x126x126	0.03	0.03	0.03
3x256x256	3x254x254	0.04	0.03	0.05
3x512x512	3x510x510	0.05	0.05	0.25
3x1024x1024	3x1022x1022	0.13	0.14	0.29
3x2048x2048	3x2046x2046	0.42	0.43	1.07
3x4096x4096	3x4094x4094	1.58	1.62	4.63
3x8192x8192	3x8190x8190	1.79	6.88	86.53



3.2.3 比较不同线程块大小对 im2col 卷积的性能影响

Thread Block Size	Grid Size	Time for Convolution (ms)
4x4	1024x1024	6.52
8x8	512x512	17.28
16x16	256x256	22.07
32x32	128x128	41.44
64x64	64x64	0.01



3.3 使用 cuDNN 方法实现 CUDA 并行卷积

3.3.1 矩阵维度对程序性能的影响

输入尺寸 (H x W)	卷积时间 (ms)
128 x 128	0.039844
256 x 256	0.039543
512 x 512	0.041879

3.3.2 不同方法的比较

Input Dimensions	Global Memory Time (ms)	Shared Memory Time (ms)	im2col Time (ms)	cuDNN Time (ms)
3x128x128	0.03	0.03	0.03	0.039844
3x256x256	0.04	0.03	0.25	0.039543
3x512x512	0.05	0.05	1.07	0.041879

4. 实验感想

通过本次并行程序设计课程中的 CUDA 卷积实验，我深刻体会到了 GPU 加速在图像处理领域的强大能力。实验不仅加深了我对卷积神经网络中卷积操作的理解，也锻炼了我使用 CUDA 进行并行编程的技能。

在实现滑窗法时，我学习到了如何在 CUDA 中管理内存，并行处理图像数据。我了解了线程块和网格的概念，以及如何通过合理地划分任务来提高计算效率。特别是在处理边界条件和不同步幅的实现过程中，我认识到了细致编程的重要性。

im2col 方法将卷积转化为矩阵乘法，大大简化了卷积的实现。这种方法让我体会到了将复杂操作转化为更简单形式的力量。通过这个方法，我学会了如何将图像数据重构为适合矩阵乘法的格式，并在实践中体会到了这种转换在性能优化上的优势。

在实现 im2col 的过程中，我深刻认识到了算法转换在解决复杂问题时的重要性。虽然这种方法需要更多的内存来存储中间结果，但它的并行计算效率给我留下了深刻的印象。

本次实验是一个宝贵的学习经历，它不仅提升了我的编程能力，也加深了我对并行计算和深度学习中卷积操作的理解。我学会了如何分析和改进程序性能，这对于任何希望在高性能计算领域发展的程序员来说都是必备的技能。在未来的学习和研究中，我期待将这些知识应用到更多的实际问题中，进一步提升自己的技术水平。