中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

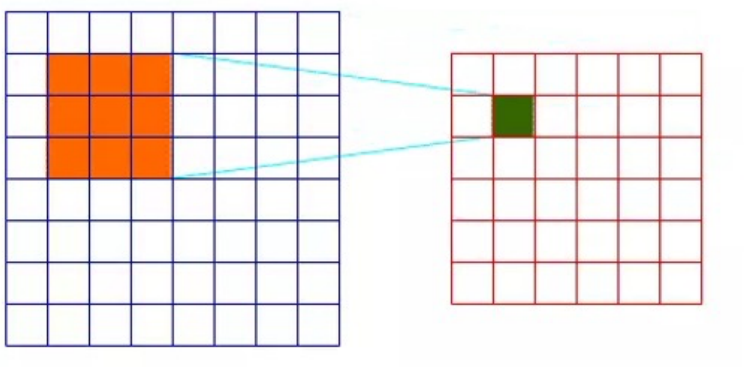
实验	CUDA卷积	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.6.11

1、实验目的

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×weight）与一个卷积核（3×3)。

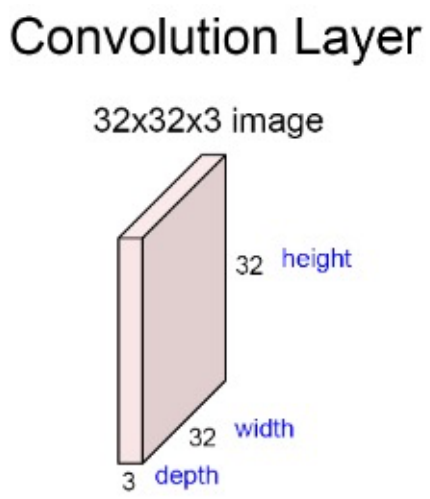
问题描述：用直接卷积的方式对输入二维图像进行卷积，通道数量（channel, depth）设置为3，卷积核个数为3，步幅（stride）分别设置为1/2/3，可能需要通过填充（padding）配合步幅（stride）完成卷积操作。注：实验的卷积操作不需要考虑bias (b)，bias设置为0。

输出：卷积结果图像（height-2×weight)及计算时间。

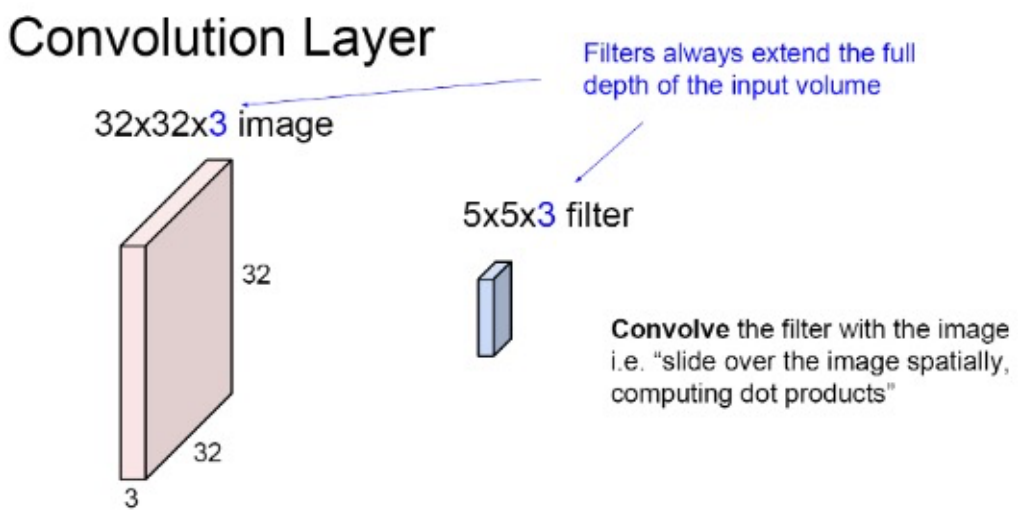
要求：使用CUDA实现并行图像卷积，分析不同图像大小、访存方式、任务/数据划分方式、线程块大小等因素对程序性能的影响。

参考：

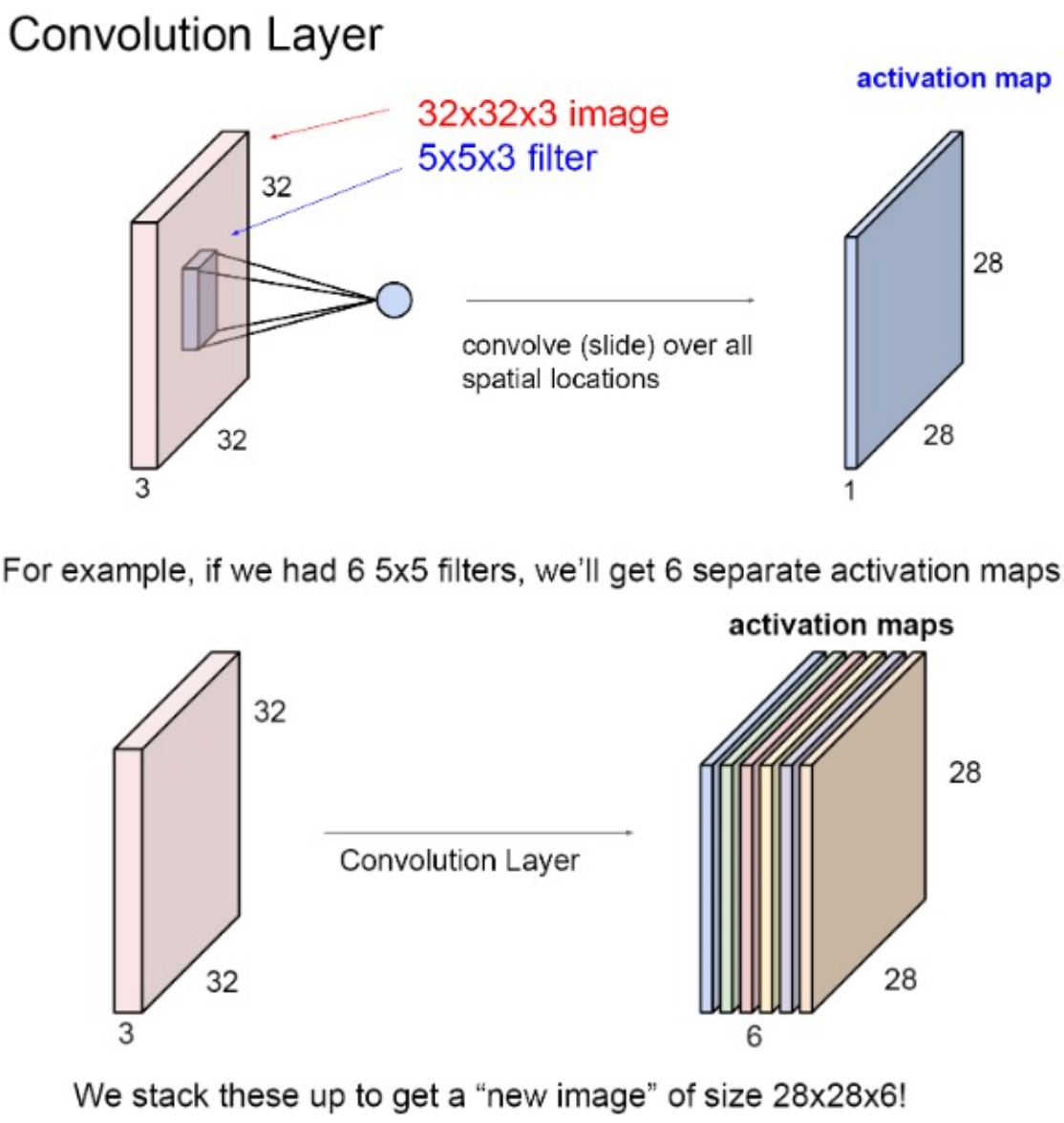
1）输入图像举例



2）输入图像与卷积核（3通道）

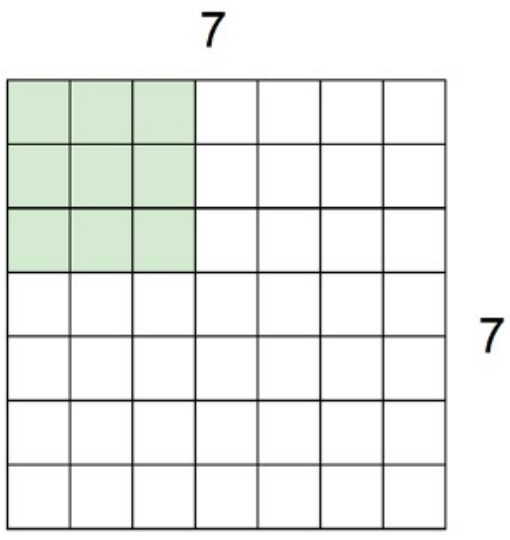


3）卷积计算过程



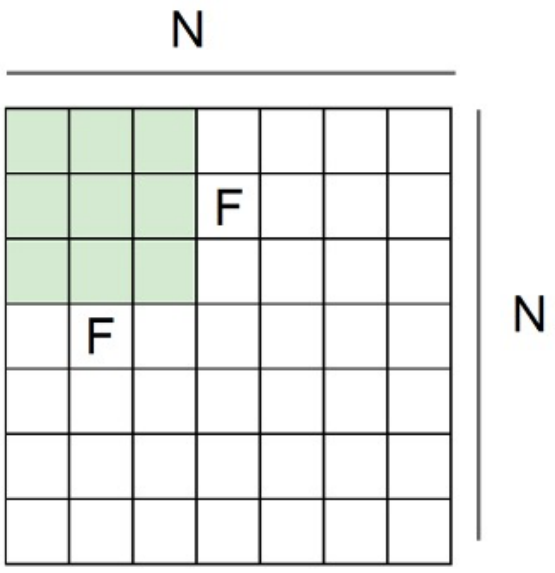
4）卷积操作的步幅（stride）与填充（padding）

A closer look at spatial dimensions:



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

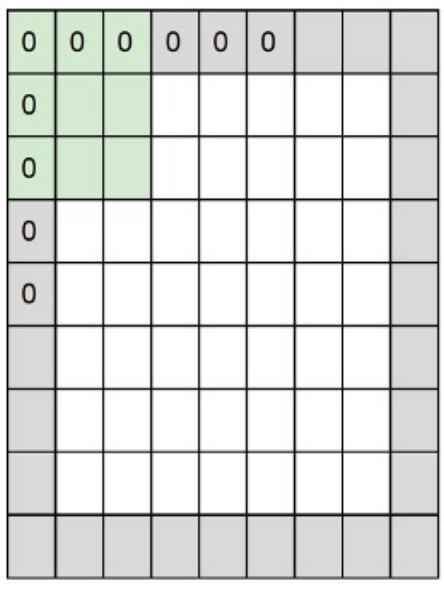
doesn't fit!
cannot apply 3x3 filter on
7x7 input with stride 3.



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

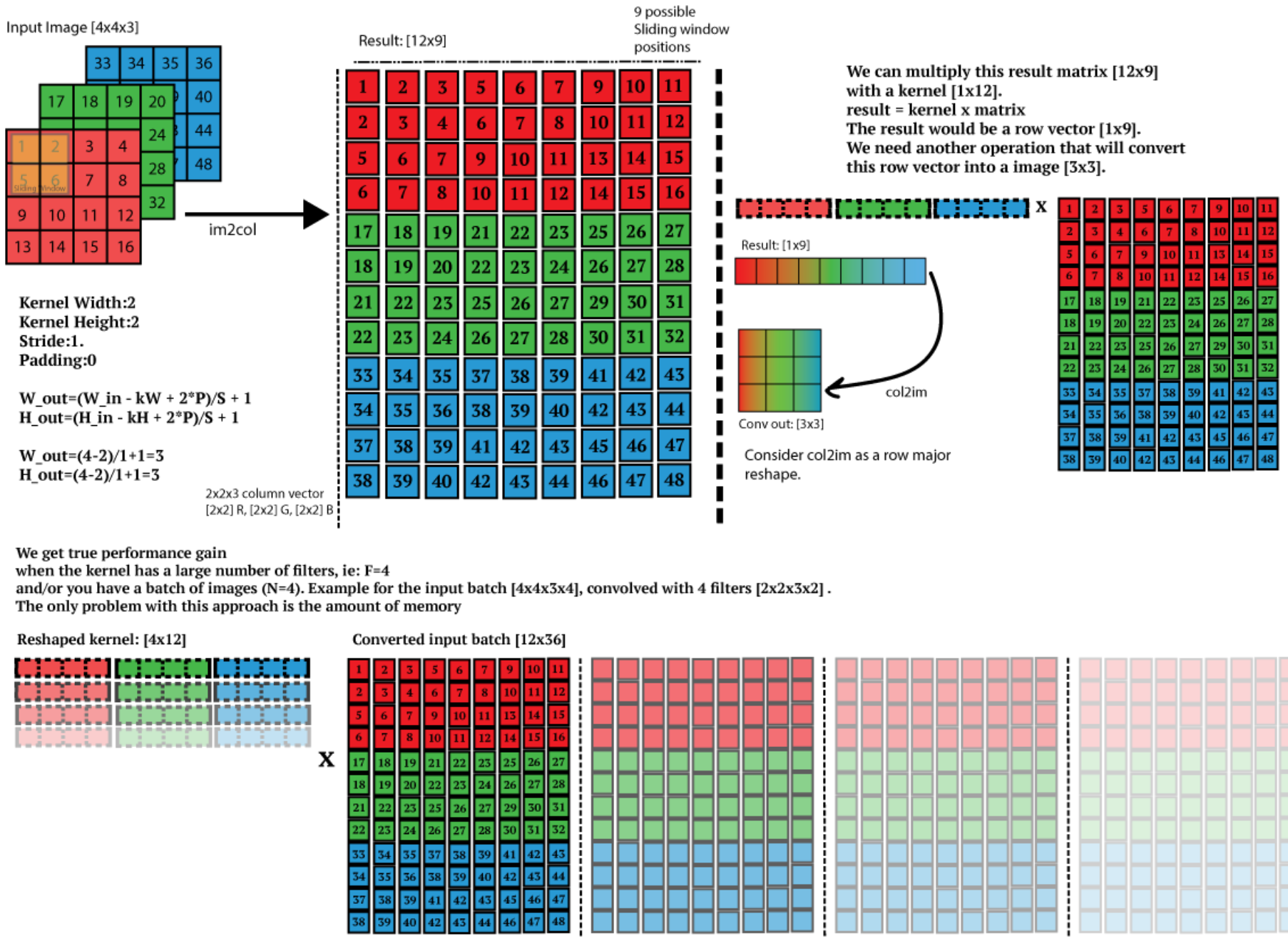


e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

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

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$w^T \cdot x$，其中w^T为1×9的权重矩阵，x为9×1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。

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

`cudnn`库配置方法

```
sudo cp cuda/include/cudnn.h    /usr/local/cuda/include
sudo cp cuda/lib64/libcudnn*    /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn.h
/usr/local/cuda/lib64/libcudnn*
export LD_LIBRARY_PATH=/cuda/lib64:$LD_LIBARARY_PATH
sudo cp /usr/local/cuda/lib64/libcudnn.so.7 /usr/local/lib/libcudnn.so.7
sudo ldconfig
```

执行命令

```
nvcc -o main1 main1.cu
nvcc -o main2 main2.cu
nvcc main3.cu -o main3 -I/opt/conda/include -L/opt/conda/lib -lcudnn
```

1.滑窗法实现CUDA并行卷积

计算出增加的 padding 和增加 padding 后的 input 大小

```
#define padding_height ((filter_height / 2) * 2)
#define padding_width ((filter_width / 2) * 2)
#define input_height (mat_height + padding_height)
#define input_width (mat_width + padding_width)
```

矩阵和 filter 的分配内存、初始化等操作，实验通过对比来检查结果的准确性，需要同时设计 CUDA 版本的卷积计算和 cpu 串行版的卷积计算。

将数据拷贝到 CUDA 中

```
    for (int i = 0; i < 3; i++) {
        cudaMemcpy(d_inputs[i], inputs[i], input_height * input_width *
sizeof(float), cudaMemcpyHostToDevice);
        cudaMemcpy(d_filters[i], filters[i], filter_height * filter_width *
sizeof(float), cudaMemcpyHostToDevice);
    }
```

CUDA 卷积计算 stread 和 grid 设置

```
    dim3 threads(block_size_x, block_size_y);
    dim3 grid((mat_width + threads.x - 1) / threads.x, (mat_height +
threads.y - 1) / threads.y);
```

设计 CUDA 卷积计算的 __global__ 函数 `cuda_convolution(float *output, float *input, float*filter)`，先计算出矩阵的坐标，当算出的 x，y 能够整除 stride 时，说明此时符合计算要求，就可在其中进行对于 filter 的循环，并且结果的坐标需要除以 stride 求得

```
__global__ void cuda_convolution(float *output, float *input, float *filter)
{
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    if (y % stride == 0 && x % stride == 0) {
        for (int i = 0; i < filter_height; i++) {
            for (int j = 0; j < filter_width; j++) {
                sum += input[(y + i) * input_width + x + j] * filter[i *
filter_width + j];
            }
        }
        output[y / stride * mat_width + x / stride] = sum;
    }
}
```

CUDA 版本的加法函数，把三层计算的结果相加，先计算出矩阵的坐标，当算出的 x，y 能够整除 stride 时，说明此时符合计算要求，就可以进行相加运算

```
__global__ void cuda_add(float *arr1, float *arr2, float *arr3, float
*result) {
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    if (y % stride == 0 && x % stride == 0) {
        result[y / stride * mat_width + x / stride] = arr1[y / stride *
mat_width + x / stride] + arr2[y / stride * mat_width + x / stride] + arr3[y
/ stride * mat_width + x / stride];
    }
}
```

分三次调用卷积计算函数，分三层进行计算，并将得到的三个结果相加

```
    for (int i = 0; i < 3; i++) {
cuda_convolution<<<grid, threads>>>(d_outputs[i], d_inputs[i], d_filters[i]);
    }
cuda_add<<<grid, threads>>>(d_outputs[0], d_outputs[1], d_outputs[2],
d_result);
```

将输出结果拷贝到内存空间，并且输出结果到文件中

```
    cudaMemcpy(result_cuda, d_result, mat_height * mat_width * sizeof(float),
cudaMemcpyDeviceToHost);
```

2.使用im2col方法实现CUDA并行卷积

函数`im2col_get_data()`：根据值的行数、列数、通道数的信息，在数组中确定该位置的值，并且如果判断该位置是 padding 的部分，就自动返回0。

`im`存储多通道二维图像的数据的格式为：各通道所有行并成一行，再多通道依次并成一行，因此`width*height*channel`首先移位
到所在通道的起点位置，加上 `width*row` 移位到所在指定通道所在行，再加上`col`移位到所在列。

```
float im2col_get_data(float *im, int row, int col, int channel) {
    row -= PAD;
    col -= PAD;
    if (row < 0 || col < 0 || row >= HEIGHT || col >= WIDTH) return 0;
    return im[col + WIDTH * (row + HEIGHT * channel)];
}
```

函数`im2col()`：先计算卷积后的尺寸，再用循环获取每个位置对应的值，先计算卷积核上的坐标(w_offset,h_offset)，然后进行两层内循环，获取输入图像对应位置的值，其中 col_index 为经过重排后图像的值的索引。

```
void im2col(float *data_im, float *data_col) {
    int height_col = (HEIGHT + 2 * PAD - FILTER_SIZE) / STRIDE + 1;
    int width_col = (WIDTH + 2 * PAD - FILTER_SIZE) / STRIDE + 1;
    int channels_col = CHANNELS * FILTER_SIZE * FILTER_SIZE;
    for (int c = 0; c < channels_col; ++c) {
        int w_offset = c % FILTER_SIZE;
        int h_offset = (c / FILTER_SIZE) % FILTER_SIZE;
        int c_im = c / FILTER_SIZE / FILTER_SIZE;
        for (int h = 0; h < height_col; ++h) {
            for (int w = 0; w < width_col; ++w) {
                int im_row = h_offset + h * STRIDE;
                int im_col = w_offset + w * STRIDE;
                int col_index = (c * height_col + h) * width_col + w;
                data_col[col_index] = im2col_get_data(data_im, im_row,
im_col, c_im);
            }
        }
    }
}
```

应用CUDA 版`gemm`来对进行重新排列过的矩阵和`filter`进行通用矩阵乘法，得到卷积结果

先为各个矩阵分配内存空间：`im`指向原矩阵，`col`指向经过`im2col`重排的矩阵，按照计算好的内存大小进行分配

```
    float *im = (float *)malloc(HEIGHT * WIDTH * CHANNELS * sizeof(float));
    float *col = (float *)malloc(channels_col * height_col * width_col *
sizeof(float));
    float *filter = (float *)malloc(CHANNELS * FILTER_SIZE * FILTER_SIZE *
sizeof(float));
    float *c = (float *)malloc(CHANNELS * width_col * height_col *
sizeof(float));
```

进行重排

```
im2col(im, col);
```

在cuda中给矩阵分配空间

```
    float *cuda_a, *cuda_b, *cuda_c;
    cudaMalloc((void **)&cuda_a, sizeof(float) * CHANNELS * FILTER_SIZE *
FILTER_SIZE);
    cudaMalloc((void **)&cuda_b, sizeof(float) * channels_col * height_col *
width_col);
    cudaMalloc((void **)&cuda_c, sizeof(float) * CHANNELS * width_col *
height_col);
```

将`filter`和`col`复制到显存中

```
    cudaMemcpy(cuda_a, filter, sizeof(float) * CHANNELS * FILTER_SIZE *
FILTER_SIZE, cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_b, col, sizeof(float) * channels_col * height_col *
width_col, cudaMemcpyHostToDevice);
```

执行`gemm`并且将结果拷贝回内存

```
    int BLOCK_SIZE = HEIGHT;
    int blocks_num = (CHANNELS * width_col * height_col + BLOCK_SIZE - 1) /
BLOCK_SIZE;

    matMultCUDA<<<blocks_num, BLOCK_SIZE>>>(cuda_a, cuda_b, cuda_c, CHANNELS,
FILTER_SIZE * FILTER_SIZE, width_col * height_col, BLOCK_SIZE);

    cudaMemcpy(c, cuda_c, sizeof(float) * CHANNELS * width_col * height_col,
cudaMemcpyDeviceToHost);
```

3.使用cuDNN方法实现CUDA并行卷积

计算并定义`OUT_HEIGHT`以及`OUT_WIDTH`

```
#define OUT_HEIGHT (HEIGHT + 2 * PAD - FILTER) / STRIDE + 1
#define OUT_WIDTH (WIDTH + 2 * PAD - FILTER) / STRIDE + 1
```

定义`input_descriptor`，设置输入矩阵的格式、类型、批处理大小、高度和宽度等信息

```
cudnnTensorDescriptor_t input_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor, CUDNN_TENSOR_NHWC,
CUDNN_DATA_FLOAT, 1, CHANNELS, HEIGHT, WIDTH));
```

定义`output_descriptor`，设置输出矩阵的格式、类型、批处理大小、通道数、高度、宽度等信息

```
cudnnTensorDescriptor_t output_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&output_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor, CUDNN_TENSOR_NHWC,
CUDNN_DATA_FLOAT, 1, 1, OUT_HEIGHT, OUT_WIDTH));
```

定义`kernel_descriptor`，设置格式、类型、输入输出通道数、高度、宽度等信息

```
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor, CUDNN_DATA_FLOAT,
CUDNN_TENSOR_NCHW, 1, CHANNELS, FILTER, FILTER));
```

定义`convolution_descriptor`，设置两种方向上填充`pad`大小、步长`stride`
`dilation_height`、模式和类型

```
cudnnConvolutionDescriptor_t convolution_descriptor;
checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution_descriptor));
checkCUDNN(cudnnSetConvolution2dDescriptor(convolution_descriptor, PAD, PAD,
STRIDE, STRIDE, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
```

定义`convolution_algorithm`

```
cudnnConvolutionFwdAlgo_t convolution_algorithm;
checkCUDNN(cudnnGetConvolutionForwardAlgorithm(cudnn, input_descriptor,
kernel_descriptor, convolution_descriptor, output_descriptor,
CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &convolution_algorithm));
```

用`cudnnGetConvolutionForwardWorkspaceSize()`函数计算整个运算所需要的空间大
小`workspace_bytes`

```
size_t workspace_bytes = 0;
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn, input_descriptor,
kernel_descriptor, convolution_descriptor, output_descriptor,
convolution_algorithm, &workspace_bytes));
```

给`d_workspace`、`d_input`、`d_output`在 cuda 上分配空间

```
    void *d_workspace{nullptr};
    cudaMalloc(&d_workspace, workspace_bytes);

    float *d_input{nullptr}, *d_output{nullptr}, *d_kernel{nullptr};
    size_t image_bytes = HEIGHT * WIDTH * CHANNELS * sizeof(float);

    cudaMalloc(&d_input, image_bytes);
    cudaMemcpy(d_input, image, image_bytes, cudaMemcpyHostToDevice);

    cudaMalloc(&d_output, image_bytes);
    cudaMemset(d_output, 0, image_bytes);
```

调用函数`cudnnConvolutionForward()`计算卷积结果

```
checkCUDNN(cudnnConvolutionForward(cudnn, &alpha, input_descriptor, d_input,
kernel_descriptor, d_kernel, convolution_descriptor, convolution_algorithm,
d_workspace, workspace_bytes, &beta, output_descriptor, d_output));
```

3、实验结果

1.滑窗法实现CUDA并行卷积（运行时间，单位：秒）

规模/stride	1	2	3
256	0.010350	0.003936	0.001459
512	0.031877	0.008878	0.005321
1024	0.117281	0.035394	0.018179
2048	0.472760	0.140865	0.070703
4096	1.898768	0.667811	0.281054

实验结果分析：

1.随着输入规模的增加，运行时间呈指数级增加，尤其是在stride较小（例如stride为1）时，这种增长更加明显，因为较大的输入规模需要处理更多的数据，导致计算量显著增加。

2.增大步长能够显著减少运行时间，对于相同的输入规模，较大的步长减少了所需的卷积运算次数，从而减少了总的计算量，stride越大，运行时间越短。

2.使用im2col方法实现CUDA并行卷积（运行时间，单位：秒）

规模/stride	1	2	3
256	0.341578	0.292111	0.289179
512	0.552698	0.321170	0.307517
1024	1.013299	0.567285	0.343559
2048	1.383452	0.567002	0.408433
4096	5.146828	1.431801	0.802262

实验结果分析：

当规模固定时，运算时间会随 stride 增大而明显减少，当`stride`固定时，运算时间会随着矩阵规模的增大而明显边长，但是与之前的 CUDA 实现直接卷积相比较，性能明显下降了，因为`im2rol`耗费了大量时间

3.使用cuDNN方法实现CUDA并行卷积（运行时间，单位：秒）

规模/stride	1	2	3
256	0.000017	0.000011	0.000007
512	0.000066	0.000018	0.000013
1024	0.000202	0.000059	0.000031
2048	0.001617	0.000221	0.000103
4096	0.004806	0.001693	0.000393

实验结果分析：

1.输入规模一定时，当步长增大，运行时间缩短。

2.步长一定时，运行时间会随矩阵规模增大而增长。

4、实验感想

1. 通过实现不同的卷积方法，更深入地理解了卷积在图像处理和深度学习中的重要性，了解了卷积操作的原理以及不同实现方法对性能的影响，对于进一步优化和理解深度学习模型的运行机制非常有帮助。
2. 通过使用CUDA进行并行计算，学会了如何利用GPU的并行性来加速计算过程，通过并行计算可以显著提高卷积操作的速度，尤其是对于大规模的图像和复杂的卷积核。
3. 通过比较滑窗法、im2col方法和cuDNN方法的性能，可以清楚地发现不同实现方法之间的差异，cuDNN通常会提供更高的性能，但也需要考虑到其可能带来的额外内存开销和依赖性。
4. 除此之外，还存在一些潜在的优化空间。例如，可以考虑优化内存访问模式、调整线程块大小、改进数据划分方式等来提高性能。