中山大学数据科学与计算机院本科生实验报告

（2020学年秋季学期）

课程名称：**高性能计算程序设计**　　任课教师：**黄聃**　　批改人：

年级+班级	**18级计7**	**专业（方向）**	**计科**
学号	18340166	**姓名**	王若琪
Email	wangrq29@mail2.sysu.edu.cn	**完成日期**	2021.1.7

1. 实验目的

- **任务 1:**

  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。

  任务一通过 CUDA 实现直接卷积（滑窗法），输入从 256 增加至 4096.
  输入：Input 和 Kernel(3x3)

  问题描述：用直接卷积的方式对 Input 进行卷积，这里只需要实现 2D , height*width，通道 channel(depth)设置为 3，Kernel (Filter)大小设置为 3*3*3，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 CNN 操作。注：实验的卷积操作不需要考虑 bias(b)，bias 设置为 0 ．

  输出：输出卷积结果以及计算时间

- **任务 2:**

  任务二使用 im2col 方法结合上次实验实现的 GEMM 实现卷积操作。输入从 256 增加至 4096，具体实现的过程可以参考下面的图片和参考资料 ．

  输入：Input 和 Kernel (Filter)
  问题描述：用 im2col 的方式对 Input 进行卷积，这里只需要实现 2D, height*width，通道 channel(depth)设置为 3，Kernel (Filter)大小设置为 3*3*3，个数为 3。 注：实验的卷积操作不需要考虑 bias(b)，bias 设置为 0，步幅(stride)分别设置为 1，2，3。

- **任务 3:**

  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。

  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 CUDA 实现直接卷积

- 首先要保证将有数值的部分用滑窗法都划过，所以应该增加 padding 部分，并保证计算时不超出内存范围，所以计算出增加的 padding 和增加 padding 后的 input 大小如下：

```
// 计算 padding
#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)
```

- 接下来进行矩阵和 filter 的分配内存、初始化等操作。因为实验需要对比，并且需要检查结果的准确性，所以既要设计 CUDA 版本的卷积计算，也要设计 cpu 串行版的卷积计算用来对比和检验。因此，要给很多输入、中间值、输出进行动态分配内存和初始化，下面简单介绍各个变量的意义：

  float *input1 ， float *input2 ， float *input3 ：指向经过 padding 的、作为输入的矩阵，一共有三层。

  float *output11 ， float *output12 ， float *output13 : 指向经过cpu版本和各个卷积核卷积后输出的矩阵，一共有三层。

  float *result1 ， float *result2 ： 分别指向经过cpu运算、经过CUDA运算后，并且将三层结果相加后的最终结果。

  float *filter1 ， float *filter2 ， float *filter3 : 分别指向三个不同的卷积核。

  float *cuda_input1 ， float *cuda_input2，float *cuda_input3 ： 分别指向 CUDA 内存的三层输入矩阵。

  float *cuda_output1，float *cuda_output2，float *cuda_output3：指向经过 CUDA 计算和各个卷积核卷积后输出的矩阵，一共有三层。

  float *cuda_filter1，float *cuda_filter2，float *cuda_filter3：分别指向 CADA 内存中三个不同的卷积核。

  float *cuda_result ： 指向 CUDA 内存中，经过 CUDA 卷积运算后，并且将三层结果相加后的最终结果。

- 将数据（矩阵和 filter）拷贝到 CUDA 中：

```
    // 将矩阵和filter拷贝到gpu中
    cudaMemcpy(cuda_input1, input1, input_height * input_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_filter1, filter1, filter_height * filter_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_input2, input2, input_height * input_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_filter2, filter2, filter_height * filter_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_input3, input3, input_height * input_width * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_filter3, filter3, filter_height * filter_width * sizeof(float), cudaMemcpyHostToDevice);
```

- CUDA 卷积计算 stread 和 grid 设置：

```
   //设置 stread 和 grid
    dim3 threads(block_size_x, block_size_y);
    dim3 grid(int(ceilf(mat_width / (float)threads.x)), int(ceilf(mat_height / (float)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，就可以在其中进行对于 filter 的循环，并且结果的坐标需要除以 stride 求得。核心代码展示如下：

```
__global__ void cuda_convolution(float *output, float *input, float *filter)
{
    // 计算矩阵的坐标
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    // 如果符合步长要求就计算结果
    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;
    }
}
```

- 由于最后需要把三层计算的结果相加，所以还需要设计一个 CUDA 版本的加法函数，同样也是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。核心代码如下：

```
__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];
}
```

- 接下来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，此过程核心代码如下：

```
    cuda_convolution<<<grid, threads>>>(cuda_output1, cuda_input1, cuda_filter1);
    cuda_convolution<<<grid, threads>>>(cuda_output2, cuda_input2, cuda_filter2);
    cuda_convolution<<<grid, threads>>>(cuda_output3, cuda_input3, cuda_filter3);
    cuda_add<<<grid, threads>>>(cuda_output1, cuda_output2, cuda_output3, cuda_result);
```

- 将输出结果拷贝到内存空间，并且输出结果到文件中。拷贝部分的代码展示如下：

```
    cudaMemcpy(result2, cuda_result, mat_height * mat_width *
sizeof(float), cudaMemcpyDeviceToHost);
```

- 最后需要比较通过 cpu 计算和通过 CUDA 计算的结果，如果误差在 1e-6f 的范围之内，就认为运算结果是正确的。此部分核心代码如下：

```
// 判断结果是否在误差范围内
int check(float *arr1, float *arr2, int n)
{
    int errors = 0;
    for (int y = 0; y < mat_height; y += stride)
    {
        for(int x = 0; x < mat_width; x += stride){
            int i = y / stride * mat_width + x / stride;
            if (isnan(arr1[i]) || isnan(arr2[i]))
                errors++;
            float diff = (arr1[i] - arr2[i]) / arr1[i];
            if (diff > 1e-6f)
                errors++;
        }
    }
    return errors;
}
```

2.2 使用 im2col 方法和 GEMM 实现卷积

首先完成函数 im2col_get_data()，此函数的功能是根据值的行数、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，就自动返回零。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。该部分核心代码如下：

```
float im2col_get_data(float *im, int row, int col, int channel){
    row -= pad;
    col -= pad;    // padding补0
    if (row < 0 || col < 0 ||
        row >= height || col >= width) return 0;
    // im[col + width*(row +
height*channel)]=im[col+width*row+width*height*channel]
    return im[col + width*(row + height*channel)];
}
```

接下来完成函数 im2col 的编写，首先计算卷积后的尺寸：

```
    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;
```

之后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，等于 c * height_col * width_col + h * width_col +w。此部分关键代码如下：

```
    for (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 (h = 0; h < height_col; ++h) {
            for (w = 0; w < width_col; ++w) {
                // 获取原图中对应的坐标
                int im_row = h_offset + h * stride;
                int im_col = w_offset + w * stride;
                // col_index为重排后图像中的索引
                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);
            }
        }
    }
```

接下来，应用上次作业完成的 CUDA 版 gemm 用来对进行重新排列过的矩阵和 filter 进行通用矩阵乘法，就可以得到卷积结果。整个过程关键代码如下展示：

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：

```
    // 动态分配内存
    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));
```

再进行重排：

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

在 cuda 中给矩阵分配空间：

```
    //cudaMalloc 分配空间
    cudaMalloc((void **)&cuda_a, sizeof(float) * channels * (filter_size *
filter_size) );
    cudaMalloc((void **)&cuda_b, sizeof(float) * channels_col * (width_col *
height_col));
    cudaMalloc((void **)&cuda_c, sizeof(float) * channels * (width_col *
height_col));
```

将 filter 和 col 复制到显存中：

```
    //cudaMemcpy 将矩阵复制到显存中
    cudaMemcpy(cuda_a, filter, sizeof(float) * channels * (filter_size *
filter_size), cudaMemcpyHostToDevice);
    cudaMemcpy(cuda_b, col, sizeof(float) * channels_col * (width_col *
height_col), cudaMemcpyHostToDevice);
```

执行 gemm 并且将结果拷贝回内存：

```
    int BLOCK_SIZE = height;
    const int blocks_num = (channels * (width_col * height_col) + BLOCK_SIZE - 1) / BLOCK_SIZE;

    // 在CUDA 中执行 gemm 函数
    matMultCUDA<<<blocks_num, BLOCK_SIZE, 0>>>(cuda_a, cuda_b, cuda_c, channels, (filter_size * filter_size), (width_col * height_col), BLOCK_SIZE);

    //cudaMemcpy 将结果从显存中复制回内存
    cudaMemcpy(c, cuda_c, sizeof(float) * channels * (width_col * height_col), cudaMemcpyDeviceToHost);
```

2.3 使用 cudnn 进行卷积操作

先计算并定义 output_height 以及 output_width：

```
#define output_height (input_height + 2*pad - filter_size) / stride + 1
#define output_width (input_width + 2*pad - filter_size) / stride + 1
```

先定义 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，这里设置 channels 为 3，高和宽分别为 input_height 和 input_width：

```
cudnnTensorDescriptor_t input_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&input_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor,
                                      /*format=*/CUDNN_TENSOR_NHWC,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*batch_size=*/1,
                                      /*channels=*/3,
                                      /*image_height=*/input_height,
                                      /*image_width=*/input_width));
```

再定义 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，这里设置通道数为 1， 高度宽度分别为 output_height、output_width：

```
//output descriptor
cudnnTensorDescriptor_t output_descriptor;
checkCUDNN(cudnnCreateTensorDescriptor(&output_descriptor));
checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor,
                                      /*format=*/CUDNN_TENSOR_NHWC,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*batch_size=*/1,
                                      /*channels=*/1,
                                      /*image_height=*/output_height,
                                      /*image_width=*/output_width));
```

再定义 kernel_descriptor，设置格式、类型、输入输出通道数、高度、宽度等信息，这里设置通道数为输出 1，输入 3， 高度宽度都为 3：

```
cudnnFilterDescriptor_t kernel_descriptor;
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                      /*dataType=*/CUDNN_DATA_FLOAT,
                                      /*format=*/CUDNN_TENSOR_NCHW,
                                      /*out_channels=*/1,
                                      /*in_channels=*/3,
                                      /*kernel_height=*/3,
                                      /*kernel_width=*/3));
```

再定义 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_height=*/pad,
                                           /*pad_width=*/pad,
                                           /*vertical_stride=*/stride,
                                           /*horizontal_stride=*/stride,
                                           /*dilation_height=*/1,
                                           /*dilation_width=*/1,

/*mode=*/CUDNN_CROSS_CORRELATION,

/*computeType=*/CUDNN_DATA_FLOAT));
```

接下来用以上定义并设置好的几个 descriptor，来定义 convolution_algorithm：

```
cudnnConvolutionFwdAlgo_t convolution_algorithm;
checkCUDNN(
    cudnnGetConvolutionForwardAlgorithm(cudnn,
                                        input_descriptor,
                                        kernel_descriptor,
                                        convolution_descriptor,
                                        output_descriptor,

 CUDNN_CONVOLUTION_FWD_PREFER_FASTEST,
                                        /*memoryLimitInBytes=*/0,
                                        &convolution_algorithm));
```

接下来用 cudnnGetConvolutionForwardWorkspaceSize() 函数计算整个运算所需要的空间大小 workspace_bytes：

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

之后给 d_workspace、d_input、d_output 在 cuda 上分配空间：

```
    void *d_workspace{nullptr};
    cudaMalloc(&d_workspace, workspace_bytes);
    int image_bytes = 1 * 3 * input_height * input_width * sizeof(float);
    float *d_input{nullptr};
    cudaMalloc(&d_input, image_bytes);
    cudaMemcpy(d_input, image, image_bytes, cudaMemcpyHostToDevice);
    float *d_output{nullptr};
    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));
```

最终将结果拷贝回内存、输出用时、输出结果到文件 result.txt 、释放空间即可。

编译连接方法：

找到动态链接库的位置，将其添加到 LD_LIBRARY_PATH 的路径里：

```
export LD_LIBRARY_PATH=/opt/conda/lib/:$LD_LIBRARY_PATH
```

编译：

```
nvcc 3.cu -o 3 -I/opt/conda/include -L/opt/conda/lib -lcudnn
```

---

3. 实验结果

3.1 CUDA 实现直接卷积结果

用 cuda 实现直接卷积，在不同规模的输入和不同大小的 stride 上分别做测试，运行时间如下表（单位：秒）：

规模\stride	1（cpu，cuda）	2（cpu，cuda）	3（cpu，cuda）
256	0.013347，0.002267	0.006197，0.002269	0.001408，0.002268
512	0.037134，0.000031	0.012639，0.002397	0.006119，0.002278
1024	0.148999，0.002340	0.043700，0.000063	0.020705，0.002304
2048	0.574062，0.000484	0.173779，0.000169	0.081471，0.000151
4096	2.378447，0.001162	0.807448，0.002033	0.324590，0.002827

由上表可知，当 stride 固定时，随着矩阵规模的增大，cpu 串行版本的卷积运算时间会明显变长，但是 CUDA 卷积运算的时间变化不明显。当规模固定，stride 增大时，cpu 串行执行的卷积运算时间会明显变短，但是 CUDA 版本的卷积运算不会有明显的时间变化。

3.2 im2rol卷积计算结果

用 im2rol + gemm 实现卷积，在不同规模的输入和不同大小的 stride 上分别做测试，运行时间如下表（单位：秒）：

规模\stride	1	2	3
256	0.231432	0.187113	0.124303
512	0.389672	0.211190	0.193387
1024	1.286314	0.371455	0.259507
2048	1.351249	0.412334	0.287018
4096	4.932192	1.348767	0.717027

由上表可知，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。但是整体与之前的 CUDA 实现直接卷积相比较，性能明显下降了，这是因为 im2rol 耗费了大量时间。

3.3 用 cudnn 计算结果与对比

用 cudnn 进行卷积，在不同规模的输入和不同大小的 stride 上分别做测试，运行时间如下表（单位：秒）：

规模\stride	1	2	3
256	0.000011	0.000008	0.000008
512	0.000027	0.000013	0.000009
1024	0.000089	0.000028	0.000019
2048	0.000360	0.000092	0.000049
4096	0.002233	0.000361	0.000162

由上表可以看出，同样的输入规模时，当步长增大，运行时间会变短。同样的步长时，运行时间会随矩阵规模增大而增大。

与自己实现的方法作比较，发现自己设计的两种方法性能都不如 cuDNN ，可能的改进方法描述见如下的 3.4 部分。

3.4 可能的改进方法

自己实现的卷积方法性能不如 cudnn，可能的改进方法如下：

- 最大化内存吞吐量：
  - 尽可能减少 host 和 device 之间的内存传输，如可以将多次传输合并为一次等；
  - 尽量使用page-locked内存；
  - 尽可能考虑局部性原理，考虑对齐问题，降低访存所消耗的时间。
- 优化指令流：
  - 尽量减少条件分支；
  - 用更高效的单精度浮点乘除法；
  - 避免多余的同步；
  - 使用 CUDA 算数指令中的快速指令。
- 资源均衡：
  - 可通过调整每线程处理的数据量，调整 block 大小等，使资源均等分配。

---

4. 实验感想

- 由于上一次作业已经熟悉了 CUDA 编程的基本操作，所以本次实验在 CUDA 的用法上没有遇到很多困难，实验过程中的主要问题集中在卷积计算输入、结果的划分操作上。
- 在做任务一时，由于要求用滑窗法将输入的每一个值都滑过，所以如果将矩阵的每个边都加上相同的 padding，将不能满足实验要求，因为总是偶数，当 stride 为 2 时会有不能滑完的现象。对于这个问题，我采用了一种简单高效的方法，就是将 padding 集中在矩阵的左边和下边，就能够保证滑窗划过所有有数值的地方，没有遗漏。
- 任务二使用 im2col 的方法，先将矩阵进行重新排列，然后用 gemm 的方法进行卷积，对于 im2col 的实现，一开始不太理解，后来参考了博客 https://blog.csdn.net/ChuiGeDaQiQiu/articl

e/details/81265471，用其中的方法进行了实现，但遗憾的是因为 im2col 的函数执行较慢，并没有达到加速的效果，这个问题还需要我之后仔细针对性研究并进行改进。

- 任务三的编程部分因为有很多参考资料所以不算很难，但最主要的难点是在编译运行的方法上，开始自己编译的时候会遇到很多报错，后来经过老师讲解介绍，才明白需要先找到动态链接库的位置，将其添加到 LD_LIBRARY_PATH 的路径里，然后编译的指令加上 -I/opt/conda/include -L/opt/conda/lib -lcudnn，就成功了。
- 总而言之，这次实验接触到了有关卷积实现的新知识和方法，之前都是直接调用深度学习框架中的 API 进行卷积操作，这次从底层实现卷积让我对 cuda 、卷积、cudnn 都有了更深的认识。