Lab11 - CUDA图像卷积

实验要求

任务一

通过CUDA实现直接卷积（滑窗法），输入从256增加至4096或者输入从32增加至512.

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或者输入从32增加至512

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。

输出：卷积结果和时间。

任务三

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。

实验过程

1.任务一

实现直接卷积，先根据输入图像规模，分配好input和output以及Kernel的内存，为了验证卷积操作的正确性，使用python随机生成输入并使用`torch.nn.functional.conv2d`得到卷积输出，保存为二进制文件，在实现CUDA卷积时则直接读取二进制文件得到图像输入、Kernel输入以及期望卷积输出，python生成样本代码如下：

```
import torch

def save_tensor(tensor, filename):
    tensor.numpy().astype('float32').tofile(filename)

input_tensor = torch.randn(1, 3, 256, 256)   # (batch_size, channels, height,
width)
kernel_tensor = torch.randn(1, 3, 3, 3)    # (out_channels, in_channels,
kernel_height, kernel_width)
```

```
# Save input tensors to binary files
save_tensor(input_tensor, 'input_tensor.bin')
save_tensor(kernel_tensor, 'kernel_tensor.bin')

# Perform convolution
output_tensor = torch.nn.functional.conv2d(input_tensor, kernel_tensor,
stride=1, padding=0)

# Save tensors to binary files
save_tensor(output_tensor, 'output_tensor.bin')
```

注意到output_width = ((input_width + 2 * padding - 3) / stride + 1)，实验中设置input_width = 256、1024、4096，要求使用不同的stride，当使用stride = 1时，padding = 0；stride = 2或3时，padding = 1，能够保证矩阵元素都被计算到，根据stride和padding的不同，得到output_width 不同，在随机生成kernel和调用conv2d计算output需要分为3次。因此初始化数据过程比较冗长，此处不做展示，以下是实现直接卷积部分的代码：

```
__global__ void directConv2D(const float* input, const float* kernel, float*
output,
                             int height, int width,int out_height,int
out_width,int stride, int padding) {
    // 每个线程负责输出矩阵的一个元素
    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;

    if (out_x < out_width && out_y < out_height) {
        float result = 0.0f;
        for (int c = 0; c < 3; ++c) {   // 3 channel
            for (int i = 0; i < 3; ++i) {   // kernel size : 3x3
                for (int j = 0; j < 3; ++j) {
                    int in_x = out_x * stride + i - padding;
                    int in_y = out_y * stride + j - padding;
                    if (in_x >= 0 && in_x < width && in_y >= 0 && in_y < height)
{
                        result += input[(c * height + in_y) * width + in_x] *
kernel[(c * 3 + j) * 3 + i];
                    }
                }
            }
        }
        output[out_y * out_width + out_x] = result;
    }
}
```

编译运行指令：

```
nvcc matConv.cu -o con
./con
```

运行结果：

```
jovyan@jupyter-21307347:~/parallel$ nvcc matConv.cu -o con
jovyan@jupyter-21307347:~/parallel$ ./con
Time taken: 578.000000 us
 Top left corner of Conv output_1:
    0.254134   -6.028669    0.393331    2.634900    6.251182   -5.798330
   -4.890934   -6.214479    2.603835   -3.327109   -1.215405   -1.090536
    0.123056   -0.614581    3.910275    2.786036    6.484142   -3.664891
    3.421272   -1.191729    8.981674   -1.687784   -7.214384    2.527701
   -4.444255    1.097696    7.666762    3.974169   -3.229892   -5.425713
   -3.533833    2.642562   -5.355713   -8.733497    3.126179    0.403254
 Top left corner of Conv output_2:
    5.737075   -1.566768    2.443383    0.265675   -4.310385  -10.765094
   -0.634832   -1.041475   -4.968822   -2.450763    4.063087   -7.696237
   -1.891620   -1.941237  -11.025776   -1.532724   -7.259421   -6.527228
   -0.169915   -4.392006    1.865724   -7.637884    5.663870    5.713378
    6.361787    6.012307   -1.254977   -4.359553   -3.488323   -1.835774
    4.276291    2.387427   -2.317335    0.524140    1.680303   -3.488412
 Top left corner of Conv output_3:
    7.160560    8.428594    9.259590    3.373339    8.428622   -5.965709
   -4.877679    3.407251    5.161260    4.489237   -2.601814    3.167477
   -7.459792    0.612131   -7.839212   -6.564108   -3.252229   -0.628158
   -1.748612   -6.080236   -7.027942    4.498902    1.006841   -3.637434
   -0.451495   -5.017364   -2.682262    3.220837  -13.028701  -13.034504
   -9.816696   -8.221527    4.748318   -1.804021    1.468058    9.161618
Convolution result is correct
```

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，主要过程是将input转为矩阵形式，再使用kernel对矩阵进行乘积，分为两个使用核函数的过程。

使用CUDA将input转为矩阵形式，每个线程负责矩阵的一个元素，计算得到该元素对应input的位置索引，赋值

```
__global__ void imtocol(const float* input, float* mat,int col_size,
                        int height, int width, int stride, int padding,
                        int output_height, int output_width) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    int n = index;
    if(index >= col_size)
        return;

    // index to input_index
    int w_out = index % output_width;
    index /= output_width;
    int h_out = index % output_height;
    index /= output_height;
    int k_w = index % 3;
    index /= 3;
    int k_h = index % 3;
    index /= 3;
    int c = index;
```

```
    int h_in = h_out * stride + k_h - padding;
    int w_in = w_out * stride + k_w - padding;

    mat[n] = (w_in >= 0 && w_in < width && h_in >= 0 && h_in < height) ?
                input[(c * height + h_in) * width + w_in] : 0;
}
```

针对一维矩阵kernel与矩阵乘积的核函数如下

```
template<int BLOCK_DIM>
__global__ void mat_multi(float *A, float *B, float *C, int m, int n, int k)
{
    int thread_x = threadIdx.x;
    int index = blockIdx.x * BLOCK_DIM + thread_x;
    // 将数组A放入共享内存中,因此注意 BLOCK_DIM 需大于等于 3*3*3
    __shared__ float s_A[27];
    if(thread_x < 27){
        s_A[thread_x] = A[thread_x];
    }
    __syncthreads();
    if (index >= k)
        return;
    float sum = 0;
    for(int i = 0 ;i < 27;i++){
        sum += s_A[i] * B[i*k + index];
    }
    C[index] = sum;
}
```

编译运行指令：

```
nvcc matIm2col.cu -o col
./col
```

运行结果：

```
jovyan@jupyter-21307347:~/parallel$ nvcc matIm2col.cu -o col
jovyan@jupyter-21307347:~/parallel$ ./col
Time taken: 648.000000 us
 Top left corner of Conv output_1:
    0.254134   -6.028669    0.393332    2.634900    6.251181   -5.798331
    5.067637   -2.736500   -4.890934   -6.214478    2.603835   -3.327110
    4.507093   -2.195582   -2.020916    9.308722    0.123056   -0.614581
    4.758551   -4.381684    2.660596    1.447884    3.693290    5.455160
    0.201106   -7.430236   -4.657875    6.813475    5.116107    0.387727
    3.567707    7.765707   -0.375183   -1.629815   -2.847869    8.868262
 Top left corner of Conv output_2:
    5.737075   -1.566768    2.443383    0.265675   -4.310385  -10.765094
    1.802785   -0.832445    3.162880   -8.818798    6.603642  -10.179982
   -0.634832   -1.041475   -4.968822   -2.450763    4.063087   -7.696238
   -1.619191    5.388216   11.256272   -5.342083   -4.971626   -2.161031
   -1.891620   -1.941238  -11.025777   -1.532724   -7.259421   -6.527227
    4.012597   10.429382   -2.527553    3.206651    3.515674    3.649584
 Top left corner of Conv output_3:
    7.160561    8.428594    9.259590    3.373339    8.428622   -5.965709
   -7.030927   -3.790965   -0.786872    1.917156   -8.293985   -1.742874
    3.289908   -3.512911    2.965688   -5.064642   -4.638582   -6.280414
    5.161260    4.489238   -2.601814    3.167478   -4.276539    6.539665
   -1.040251   -7.665965   -1.864578    1.174596   -3.656366   -3.102759
   -4.793063    8.891558   -2.127961   -3.386523    2.647028   10.663897
Convolution result is correct
```

经实验微调，使用直接卷积时，在`BLOCK_DIM`为64时能够取得相对好性能；使用im2col时，在`BLOCK_DIM_1`为64，`BLOCK_DIM_2`为108时能够取得相对好性能。因此在该条件下对比两种方法的耗时性能，时间单位 us。

方式\规模	256	1024	4096
直接卷积	578	3584	53148
im2col	648	4843	72192

3.任务三

使用cuDNN的卷积操作进行同样操作（对同一input使用三个kernel设置3个stride得到3个output）

编译运行指令：

```
export LD_LIBRARY_PATH=/opt/conda/lib:$LD_LIBRARY_PATH
nvcc matcuDNN.cu -I/opt/conda/include -L/opt/conda/lib -lcudnn -o cud
./cud
```

运行结果：

```
jovyan@jupyter-21307347:~/parallel$ nvcc matcuDNN.cu -I/opt/conda/include -L/opt/conda/lib -lcudnn -o cud
jovyan@jupyter-21307347:~/parallel$ ./cud
Time taken: 425.000000 us
 Top left corner of Conv output_1:
   -0.300815   11.152697   -6.124198   -0.998235    1.070946   -6.506938
   -3.411454    2.446485    5.166219    8.431587   -0.912940   -3.551409
    4.719252   -0.827431   -3.091829   -0.351792   -5.751691    7.388699
   -5.142766    5.160377   -0.854922    4.152200    2.150512   -7.319922
   -3.835061    1.235494    2.273860   -3.695239   -4.082405   -1.157541
   -0.993468   -4.507921    6.649927    0.117435   -3.412103    6.653762
 Top left corner of Conv output_2:
    2.557164    2.146547   -0.357057    0.825796    8.703426    2.991445
   -2.392913   -3.533831   -0.973276   -7.211196    1.933025    5.710457
    2.416268    8.152512   -5.326787    4.637668    9.999058    0.360640
   -5.717866    2.192028    2.416409    2.581405    3.147251   -1.125310
    4.487141   -2.531218   -6.169483   -0.992963   -3.592781   -6.510606
   -4.664241   -0.681915   -2.080979    4.025827    2.363026   -2.427620
 Top left corner of Conv output_3:
   -1.658095    0.097229   -6.824610   -3.564406   -5.537218    2.451902
   -0.217635   -1.875881   -1.330627    0.390608    3.153630   11.032455
   -5.125171   -0.435975    1.166107    3.865336   -5.095435   -0.435825
   -1.337498   -6.902801   -4.887091    1.279805    1.067882    0.573803
   -7.878808   -4.026719    5.494079   -3.742556   -9.089760    8.729692
   -0.786048   -3.140962   -0.454498    5.017482   -5.294424   -1.420781
```

经多次运行可以看出，使用cuDNN库实现的卷积操作要明显耗时少于自己实现的两种方法，要提升卷积速度还应该注意内存管理，减少内存分配和释放的开销，减少冗余计算，思考如何更高效地使用共享内存以及让读取数据在内存中对齐，利用缓存优化性能