中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	CUDA 卷积计算	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.6.9

实验目的

- 通过CUDA实现直接卷积
- 使用im2col结合实现的通用矩阵乘法实现卷积操作
- 使用cuDNN提供的卷积方法进行卷积操作

实验过程和核心代码

在获取了input大小后, 我们先初始化input和kernel数据

```
int in_elements = channel * in_size * in_size;
int k_elements = channel * k_size * k_size;

// host分配内存
float* h_in = new float[in_elements];
float* h_k = new float[k_elements];

randomInit(h_in, in_elements);
// 初始化卷积核为1, 便于验证答案
oneInit(h_k, k_elements);

// cuda分配内存
float* d_in;
float* d_k;
cudaMalloc(&d_in, in_elements * sizeof(float));
cudaMalloc(&d_k, k_elements * sizeof(float));
cudaMemcpy(d_in, h_in, in_elements * sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_k, h_k, k_elements * sizeof(float), cudaMemcpyHostToDevice);
```

随后使用一个循环来分别运行stride为1,2,3的卷积计算. 在得到确定的stride数据后, 我们采用 SAME Padding 策略来计算需要的padding大小. SAME Padding 策略指的是使输出大小等于输入大小除以步长:

$$\text{Output} = \lceil \frac{\text{Input}}{\text{S}} \rceil$$

所以我们可以根据输入大小与步长来计算出输出大小, 根据输出大小与下面的公式来计算padding值:

$$\text{Output} = \frac{\text{Input} - K + 2P}{S} + 1$$

注意, padding 计算过程中有除2操作, 可能导致 padding 的计算结果代入上式后计算出的 output 大小不一致, 所以使用计算得到的padding根据上式重新计算 output 大小作为最终的 output 大小. 计算代码如下:

```
// 采用SAME Padding策略(输出大小约等于输入大小除以步长), 根据input_size和stride计算padding
int out_size = ceil(in_size / (float)stride);
int padding = ((out_size - 1) * stride + k_size - in_size) / 2;
// 计算出的结果可能无法被2整除, 使用对称padding, 所以还需要根据padding重新计算输出大小
out_size = (in_size - k_size + 2 * padding) / stride + 1;
```

得到所需的参数后, 为输出分配内存. 然后运行`runConv`函数. 该函数主要通过一个循环来依次调用不同的卷积方式, 并输出结果与运行时间.

SlideConvolution 滑窗法

该实现方法相关代码主要在文件`SlideConvolution.cu`中
滑窗法的实现思路很简单: 每个线程负责计算一个位置的结果值. 所以线程数量要匹配 output 大小. 在`slide`函数中, 根据线程块大小, 计算出 grid 大小, 并调用滑窗法核函数. 并添加计时逻辑.

滑窗法核函数的思路是通过一个循环来累加不同通道的计算结果, 在某个通道的计算中通过双层循环来遍历计算 kernel 与对应位置 input 的乘积. 核心在于根据线程负责计算的 output 位置与 kernel 位置来获取对应 input 的位置.

假设线程负责计算 output 的位置为(i, j), kernel 的位置为(K_i, K_j).
由卷积定义可知 output 位置代表 kernel 移动步数, 乘以步长就可以得到对应在 input 的滑动距离, 所以 kernel 滑动距离为$(i \times S, j \times S)$.
由于我们padding没有实际加到input中, 所以还需要减去padding值, 最后加上 kernel 的位置即可得到最终结果:

$$\begin{aligned} \mathrm{In}_i &= i \times S - P + K_i \\ \mathrm{In}_j &= j \times S - P + K_j \end{aligned}$$

在获取到 In_i 和 In_j 后, 由于我们使用一维数组来存储数据, 所以还需要根据通道值 C 来将 $(C, \mathrm{In}_i, \mathrm{In}_j)$ 转换为一维下标:

$$\mathrm{idx} = (C \times \mathrm{H}_{in} + \mathrm{In}_i) \times \mathrm{W}_{in} + \mathrm{In}_j$$

实现代码如下

```
// 本线程负责计算output的位置(row, col)
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

if(row >= out_size || col >= out_size) {
    return;
}

float sum = 0.0f;
// 每个channel计算的值累加
for(int c = 0; c < channel; ++c) {
    // 计算 c 通道下 kernel 每个位置与对应input位置的值的结果
    for(int k_row = 0; k_row < k_size; ++k_row) {
        for(int k_col = 0; k_col < k_size; ++k_col) {
            // kernel 在(k_row, k_col) 对应 input 的位置(in_row, in_col)计算
            int in_row = row * stride - padding + k_row;
            int in_col = col * stride - padding + k_col;
            // 处于padding区域, 值为0, 无需进行后续计算
            if(in_row < 0 || in_row >= in_size || in_col < 0 || in_col >= in_size) {
                continue;
            }
            // 由于使用一维数组, 将 (c, in_row, in_col) 转换为一维下标
            int in_idx = (c * in_size + in_row) * in_size + in_col;
            int k_idx = (c * k_size + k_row) * k_size + k_col;
            sum += d_in[in_idx] * d_k[k_idx];
        }
    }
}
// 输出(row,col)转换为一维下标
int out_idx = row * out_size + col;
d_out[out_idx] = sum;
```

Im2col & GEMM

该实现方法相关代码主要在文件`Im2colConvolution.cu`中

im2col重点在于将 input 转换为 col 矩阵, 将 kernel 展开为一维后直接与 col 矩阵做矩阵乘法就能得到最终的结果.

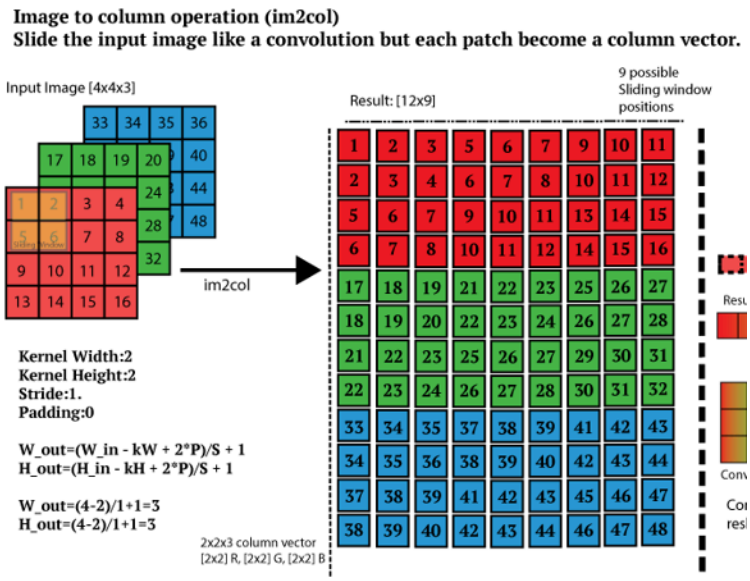
首先计算 col 矩阵的维度, 高度 H_{col} 与展开后的 kernel 长度相同, 宽度 W_{col} 为结果 output 矩阵的元素个数, 计算公式如下:

$$H_{col} = K \times K \times C_k$$
$$W_{col} = H_{out} \times W_{out}$$

为 col 矩阵在 cuda 上分配内存, 并根据设置的线程块与 col 矩阵维度计算 grid 的大小, 使每个线程负责计算 col 矩阵一个位置的值

```
float* d_col;
cudaMalloc(&d_col, col_x * col_y * sizeof(float));
dim3 gridDim((col_x + blockDim.x - 1) / blockDim.x,
             (col_y + blockDim.y - 1) / blockDim.y);
```

接着使用 `im2colKernel` 核函数完成 im2col 过程.
im2col过程的核心在于获取线程负责 col 矩阵(x, y)位置对应 input 位置的值, 并赋值到 col 矩阵的(x,y)位置.



观察im2col的示例图
我们容易知道(x,y)位置对应 input 的 channel 为:

$$\lfloor \frac{x}{H_k \times W_k} \rfloor$$

然后通过 x 的值计算当前位于展开一维 kernel 的哪个位置:

$$\text{index}_k = x\%(H_k \times W_k)$$

再转换为二维 kernel 的坐标(x_k, y_k):

$$x_k = \text{index}_k // W_k$$
$$y_k = \text{index}_k \% W_k$$

至此我们通过 x 的值计算出了 y=0 时, 第一个 kernel 的(x_k, y_k)位置, 此时 y=0, 则我们需要的 input 位置也为(x_k, y_k). 接下来考虑 y != 0 带来的影响.

y 代表 kernel 的移动步数, 而 kernel 的移动会影响 kernel 的(x_k, y_k)位置 对应原 input 的位置. 显然, 我们只需要计算出 kernel 在x,y方向的移动步数, 再乘以步长即可得到移动偏移$(\text{offset}_x, \text{offset}_y)$. 而 output 的 W_{out} 就代表 kernel 在 y 方向最多的移动步数, 所以:

$$\text{offset}_x = y // W_{out} \times S$$
$$\text{offset}_y = y \% W_{out} \times S$$

同理, 我们没有对 input 进行实际的 padding, 所以最后还需要减去 padding带来的偏移, 最终对应 input 的位置 $(\text{in}_c, \text{in}_x, \text{in}_y)$计算如下:

$$\text{in}_c = x // (H_k \times W_k)$$
$$\text{in}_x = x_k + \text{offset}_x - P$$
$$\text{in}_y = y_k + \text{offset}_y - P$$

将$(\text{in}_c, \text{in}_x, \text{in}_y)$转换为一维坐标然后将值赋值给 col 矩阵 (x, y) 位置即可.
代码实现如下:

```
int x = blockIdx.y * blockDim.y + threadIdx.y;
int y = blockIdx.x * blockDim.x + threadIdx.x;

if(x >= col_x || y >= col_y) {
    return;
}

int k_col = k_size * k_size;
// col (x,y) 位置对应 im (in_c, in_x, in_y) 位置的值
int in_c = x / k_col;
// x % k_col / k_size: 第一个 kernel 的 x 位置; y /out_size * stride: kernel 向下移动偏移, padding: padding移
int in_x = x % k_col / k_size + y / out_size * stride -padding;
// x % k_col % k_size: 第一个 kernel 的 y 位置; y %out_size * stride: kernel 向右移动偏移: padding: padding移
int in_y = x % k_col % k_size + y % out_size * stride -padding;

// 处于padding部分, 赋值为0
if(in_x < 0 || in_x >= in_size || in_y < 0 || in_y >=in_size) {
    d_col[x * col_y + y] = 0.0f;
} else {
    // (in_c, in_x, in_y) 转换为一维下标
    d_col[x * col_y + y] = d_in[(in_c * in_size + in_x) * in_size + in_y];
}
```

im2col 完成后在进行通用矩阵乘法的核函数, 注意计算之前要重新计算 grid 的维度, 因为通用矩阵乘法的结果维度为(1, $H_{out} \times W_{out}$), 每个线程负责一个结果位置计算:

```
// 任务变化, GEMM结果维度为(1, out_size * out_size), 重新计算gridDim
gridDim.x = (1 + blockDim.x - 1) / blockDim.x;
gridDim.y = (out_size * out_size + blockDim.y - 1) / blockDim.y;

GEMM<<<gridDim, blockDim>>>(d_k, d_col, d_out, 1, channel * k_size * k_size, col_y);
```

通用矩阵乘法采用上次实验中的共享内存版本, 但是也结合了循环展开, 即在计算子块乘法的结果时也采用循环展开来进一步优化:

```
// ...

float4 sum = make_float4(0.0f, 0.0f, 0.0f, 0.0f);

// ...

// 循环处理分块
for(int tile = 0; tile < tile_count; ++tile) {

    // ...

    __syncthreads();
    // 循环展开 计算子块乘法结果
    int floor4 = (blockDim.x & (~3));
    for(int i = 0; i < floor4; i += 4) {
        sum.x += sharedTileA[ty][i] * sharedTileB[i][tx];
        sum.y += sharedTileA[ty][i + 1] * sharedTileB[i + 1][tx];
        sum.z += sharedTileA[ty][i + 2] * sharedTileB[i + 2][tx];
        sum.w += sharedTileA[ty][i + 3] * sharedTileB[i + 3][tx];
    }

    // 不足4列的部分
    for(int i = floor4; i < blockDim.x; ++i) {
        sum.x += sharedTileA[ty][i] * sharedTileB[i][tx];
    }
    __syncthreads();
}
```

cuDNN

该实现方法相关代码主要在文件 `cuDNN.cu` 中

通过阅读文档来实现 cuDNN. 首先创建 cuDNN Handle, 然后创建并设置张量描述符、卷积核描述符和卷积描述符. 最后设置卷积算法, 并调用cuDNN的卷积计算函数.

```
cudnnHandle_t cudnn;
checkCUDNN(cudnnCreate(&cudnn));

// 创建描述符
cudnnTensorDescriptor_t in_desc, out_desc;
cudnnFilterDescriptor_t kernel_desc;
cudnnConvolutionDescriptor_t conv_desc;
checkCUDNN(cudnnCreateTensorDescriptor(&in_desc));
checkCUDNN(cudnnCreateTensorDescriptor(&out_desc));
checkCUDNN(cudnnCreateFilterDescriptor(&kernel_desc));
checkCUDNN(cudnnCreateConvolutionDescriptor(&conv_desc));

// 设置描述符
checkCUDNN(cudnnSetTensor4dDescriptor(in_desc, CUDNN_TENSOR_NCHW,CUDNN_DATA_FLOAT, 1, channel, in_size, in_size));
checkCUDNN(cudnnSetTensor4dDescriptor(out_desc, CUDNN_TENSOR_NCHW,CUDNN_DATA_FLOAT, 1, channel, out_size, out_size));
checkCUDNN(cudnnSetFilter4dDescriptor(kernel_desc, CUDNN_DATA_FLOAT,CUDNN_TENSOR_NCHW, 1, channel, k_size, k_size));
checkCUDNN(cudnnSetConvolution2dDescriptor(conv_desc, padding, padding, stride,stride, 1, 1, CUDNN_CROSS_CORRELATION, CUDNN_DATA_FLOAT));
```

```
float alpha = 1.0f, beta = 0.0f;
cudnnConvolutionFwdAlgo_t algo = CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM;
cudaEvent_t start, end;
cudaEventCreate(&start);
cudaEventCreate(&end);
cudaEventRecord(start);

cudnnConvolutionForward(
    cudnn,
    &alpha,
    in_desc, d_in,
    kernel_desc, d_k,
    conv_desc,
    algo,
    nullptr, 0,
    &beta,
    out_desc, d_out
);

cudaEventRecord(end);
cudaEventSynchronize(end);
cudaEventElapsedTime(&time, start, end);
```

实验结果

完整输出 output 结果见`result_full`文件夹
输出部分 output 结果见`result`文件夹
文件最后的数字代表输入规模

block size 为 32 的运行结果如下:
时间单位：ms

Input Size	Stride	Slide	im2col+GEMM	cuDNN
32	1	0.163232	**0.056064**	0.137504
32	2	0.064416	0.056544	**0.011904**
32	3	0.045632	0.059680	**0.017920**
64	1	0.158176	0.115360	**0.082624**
64	2	0.067360	0.059136	**0.008832**
64	3	0.048992	0.051968	**0.008384**
128	1	0.169792	0.234656	**0.093088**

Input Size	Stride	Slide	im2col+GEMM	cuDNN
128	2	0.065344	0.108864	**0.008160**
128	3	0.050496	0.067264	**0.009440**
256	1	0.331520	0.811232	**0.080672**
256	2	0.084192	0.234048	**0.018912**
256	3	0.049664	0.126016	**0.008832**
512	1	0.611680	2.968800	**0.087904**
512	2	0.198528	0.787904	**0.009888**
512	3	0.115616	0.374112	**0.008736**

可以看到, 只有在 input size 为 32, stride 为 1 时 im2col 方法的运行时间最短, 其他情况下都是 cuDNN 的运行时间最短.

另外, 在 input size 较小时, stride 较大时, im2col 方法的性能略微比 slide 方法更快. 但是 随着 input size 增加, im2col + GEMM 性能对比其他方法差距很大, 尤其在 input size 为 512 并且 stride 为 1 时. 这是由于 **im2col + GEMM 方法中, 转换的 col 矩阵大小与 output 大小有关, 即受到 input size 和 stride 的影响**. 而 im2col 的过程需要从全局内存中取矩阵元素个数次数据, 导致运行运行时间显著增加.

在 cuDNN 的实现中, 我选用的是 `CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM` 算法, Implicit GEMM 算法与 im2col + GEMM 相似, **但是没有进行显示的 im2col 转换, 矩阵的转换发生在计算当中**. 在 GEMM 的过程中进行输入输出的坐标映射进行卷积计算. 另外在 GEMM 的实现中也可能使用了更高效的实现方式, 从而使卷积计算整体的运行速度非常快.

可能的改进方案: 对于 im2col + GEMM, 可以考虑实现上述的 Implicit GEMM 算法, 不直接将输入转换为 col 矩阵, 而是在 GEMM 的计算过程通过坐标映射来进行计算. 另外可以考虑实现更高效的 GEMM 算法, 例如流水并行化(Double Buffering). 在实现过程中充分利用共享内存可能有更好的效果.

实验感想

在本次实验中, 我使用 cuda 编程实现了滑动窗口卷积计算与 im2col + GEMM 卷积计算. 同时应用了 cuDNN 提供的卷积计算操作并与自己实现的方式性能进行了对比.
在这次实验中涉及非常多的坐标映射, 在进行坐标映射的计算中要充分理解算法的原理并进行清晰的推导, 不然非常容易出错.
另外, 在实现某些算法的过程中, 如果想要提高性能, 可以对于现有的高效算法进行分析理解, 提取出其高效的原因并应用到自己的实现中.