并行程序设计与算法实验 11

实验	CUDA实现卷积	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazp@mail2.sysu.edu.cn	完成日期	2024/06/05

1. 实验要求

> - 使用滑窗法、im2col方法实现CUDA并行图像卷积
> - 分析不同因素性能的影响
>   - 线程块大小、图像大小：如何提高占用率?
>   - 访存方式：何时使用何种存储?
>   - 数据/任务划分方式：按行，列，数据块划分，等
> - 使用cuDNN提供的卷积操作进行对比实验，分析自身实现的性能

2. 实验原理

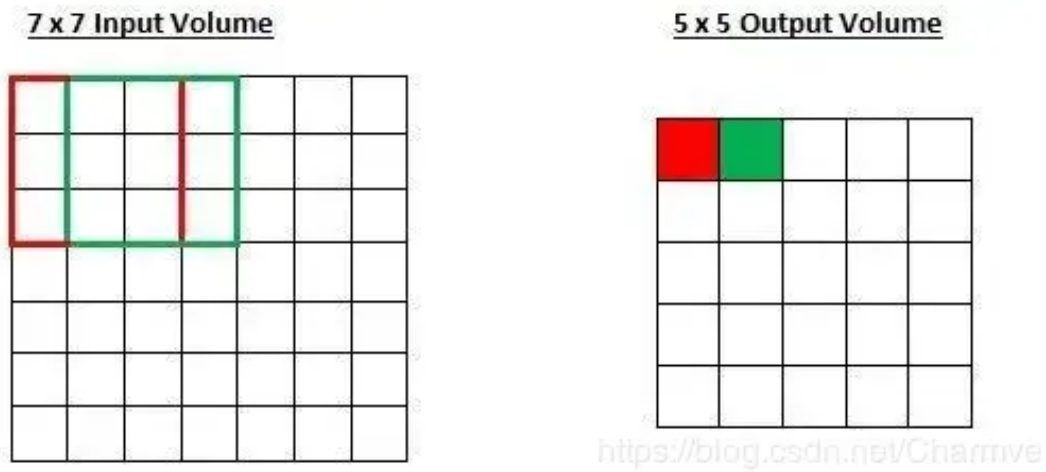
卷积操作实现的原理借鉴自知乎

卷积操作（Convolution Operation）是计算机视觉和信号处理领域中一个基本而重要的概念。在深度学习中特别是在卷积神经网络（Convolutional Neural Networks, CNNs）中，卷积操作广泛用于提取图像特征。以下是卷积操作的原理和流程的详细介绍：

卷积操作本质上是通过一个核（kernel）或滤波器（filter）在输入数据上滑动，并计算每个位置上的加权和，从而产生一个输出图像或特征图（feature map）。

2.1 核（Kernel）或滤波器（Filter）

- 核是一个小矩阵，通常尺寸为 (k \times k)，如 (3 \times 3)、(5 \times 5) 等。
- 核的每个元素称为权重（weights），它们在卷积过程中与输入图像的对应部分相乘。



2.2 卷积操作

1. 输入图像 (Input Image): 设输入图像为二维矩阵 I, 尺寸为 $H \times W$ 。
2. 核 (Kernel): 设核为二维矩阵 K, 尺寸为 $k \times k$ 。
3. 滑动窗口 (Sliding Window)：核在输入图像上滑动, 每次移动一个步长 (stride), 通常为1。
4. 计算卷积和 (Convolution Sum) :

- 对于输入图像中的每个位置 (i, j), 核覆盖的区域为 $I[i : i + k, j : j + k]$ 。
- 计算覆盖区域与核对应元素的逐元素乘积并求和:

$$O[i, j] = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I[i + m, j + n] \cdot K[m, n] \tag{1}$$

- 结果 $O[i, j]$ 就是输出特征图在位置 (i, j) 的值。

3. 实验代码

代码详见附件或github

3.1 划窗法（直接卷积）

3.1.1 CPU实现

该方法就是对上述算法的一个具体实现。

```
void CPU_Conv(float *input, float *output, float *kernel) {
    for (int row = 0; row < convOutSize; row ++) {
        for (int col = 0; col < convOutSize; col ++) {
            float sum = 0.f;
            for (int i = 0; i < K; i ++) {
                for (int j = 0; j < K; j ++) {
                    int curCol = col + j;
                    int curRow = row + i;
                    // 3通道
                    sum += kernel[i * K + j] * input[curRow * N + curCol] + kernel[i * K + j + K * K] * input[curRow * N + curCol + N * N] + kernel[i * K + j + 2 * K * K] * input[curRow * N + curCol + 2 * N * N];
                }
            }
            output[row * convOutSize + col] = sum;
        }
    }
}
```

首先，这是一个CPU的简单实现。之所以实现CPU版本的卷积，主要是方便后续可以进行结果的验证。

CPU实现时，已经对3个通道进行展开了，并且涉及到4层for循环。时间复杂度高达$O(convSize^2K^2)$

考虑到卷积的高度并发性，我们可以用cuda的方式来去实现。

3.1.2 CUDA实现

```
__global__ void Conv2DKernel(float *input, float *output) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;

    if (col >= convOutSize || row >= convOutSize) {
        return;
    }
    int curCol = 0, curRow = 0;
    float sum = 0.f;
    for (int i = 0; i < K; i ++) {
        for (int j = 0; j < K; j ++) {
            curCol = col + j;
            curRow = row + i;
            // 3 通道
            sum += kernel_dev[i * K + j] * input[curRow * N + curCol] + kernel_dev[i * K + j + K * K] * input[curRow * N + curCol + N * N] + kernel_dev[i * K + j + 2 * K * K] * input[curRow * N + curCol + 2 * N * N];
        }
    }
    output[row * convOutSize + col] = sum;
}
```

这段代码实现的是一个二维卷积操作，以下是算法流程的详细介绍：

1. **线程索引计算：**
   - 计算当前线程负责的输出元素位置 `(col, row)`，通过以下公式：

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

   - `threadIdx.x` 和 `threadIdx.y` 是当前线程在线程块内的索引，`blockDim.x` 和 `blockDim.y` 是线程块的维度大小，`blockIdx.x` 和 `blockIdx.y` 是线程块的索引。
2. **边界检查：**
   - 检查计算位置是否超过输出矩阵的大小，如果超过则退出线程：

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

3. **卷积操作：**
   - 初始化累加和变量 `sum`：

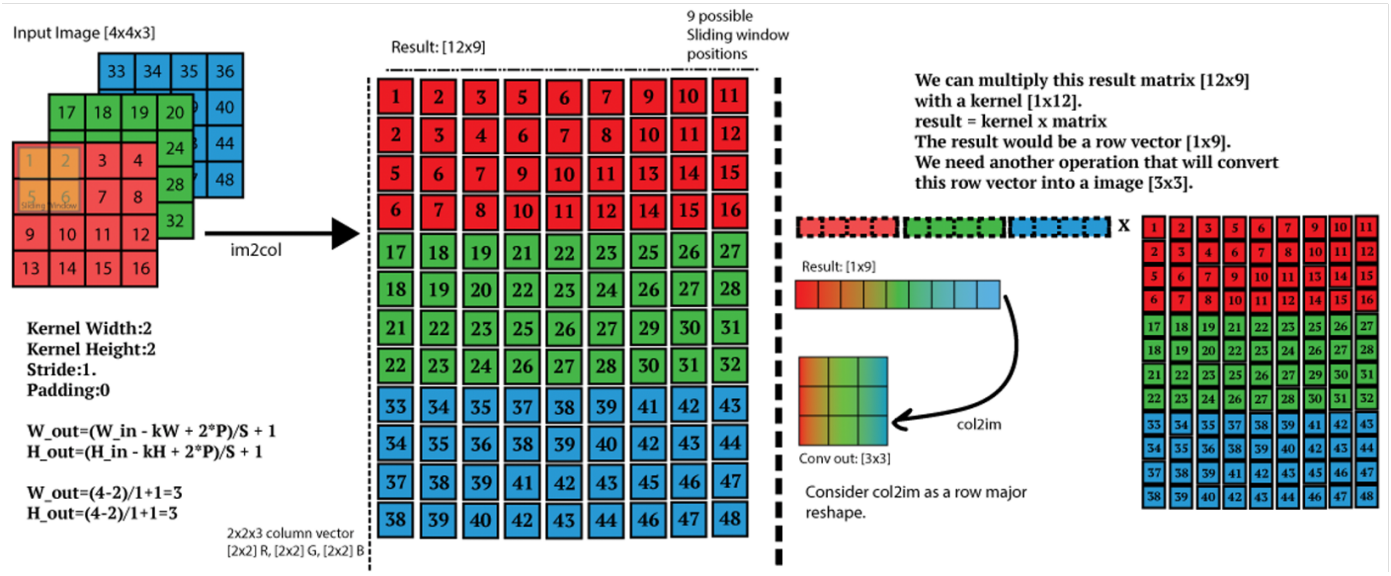
     ```
     float sum = 0.f;
     ```

- 遍历卷积核的每个元素进行卷积操作。假设卷积核大小为 `K x K`，输入图像大小为 `N x N`，并且有3个通道：

```
for (int i = 0; i < K; i ++) {
    for (int j = 0; j < K; j ++) {
        curCol = col + j;
        curRow = row + i;
        // 3 通道
        sum += kernel_dev[i * K + j] * input[curRow * N + curCol]
               + kernel_dev[i * K + j + K * K] * input[curRow * N + curCol + N * N]
               + kernel_dev[i * K + j + 2 * K * K] * input[curRow * N + curCol + 2 * N * N];
    }
}
```

- 上述循环遍历卷积核的所有元素，并且对每个元素与对应位置的输入图像像素进行乘法累加。这里假设 `kernel_dev` 是一个一维数组，包含了卷积核的所有元素。对于三通道的情况，卷积核在内存中连续存储，第一个通道的元素在 `kernel_dev` 的前 `K*K` 个位置，第二个通道在 `K*K` 到 `2*K*K` 位置，第三个通道在 `2*K*K` 到 `3*K*K` 位置。

3.2 im2col方法



这一幅图大致介绍了im2col方法。简单来说，就是根据卷积核，逐个确认每个卷积核与图像相交的元素，并构成列向量。得到中间的result矩阵。举个例子，result的第一列，`1 2 5 6 17 18 21 22 33 34 37 38`分别就是红绿蓝三个通道的第一个窗口的元素。

得到，result矩阵之后，我们就可以用矩阵乘法来实现了：

$$w^T \cdot x \tag{2}$$

代码实现：

```
__global__ void im2col(float *input, float *col_vec, float *res_dev) {
    int col = threadIdx.x + blockDim.x * blockIdx.x;
    int row = threadIdx.y + blockDim.y * blockIdx.y;
```

```
4
5       if (col >= convOutSize || row >= convOutSize) {
6           return;
7       }
8       int curCol = 0, curRow = 0;
9
10      const int col_vec_row = convOutSize * convOutSize;
11      // const int col_vec_col = K * K * 3;
12
13      for (int i = 0; i < K; i ++) {
14          for (int j = 0; j < K; j ++) {
15              curRow = row + i;
16              curCol = col + j;
17              // 1. input
18              // 2. col_vec: i * K + j 行
19              //             row * convOutSize + col 列
20              col_vec[(i * K + j) * col_vec_row + row * convOutSize + col] =
    input[curRow * N + curCol];
21              col_vec[(i * K + j) * col_vec_row + row * convOutSize + col + col_vec_row
    * K * K] = input[curRow * N + curCol + N * N];
22              col_vec[(i * K + j) * col_vec_row + row * convOutSize + col + 2 *
    col_vec_row * K * K] = input[curRow * N + curCol + N * N * 2];
23          }
24      }
25      __syncthreads();
26      matrix_mul(col_vec, res_dev);
27  }
```

这里有几个点需要明确：

1. result矩阵列向量的长度（行数）取决于kernel size，这是直观的，因为一个列向量对应于一个窗口
2. result矩阵行的大小为`convSize * convSize`，因为一共会有`convSize * convSize`个窗口
3. index的计算：

```
1   col_vec[(i * K + j) * col_vec_row + row * convOutSize + col]
2     = input[curRow * N + curCol];
```

- 对于第0个通道来说，窗口的中的第几个元素确定了它在第几行；它在卷积结果中的第几个元素，确定了它在哪一列。从而可以得到`col_vec[(i * K + j) * col_vec_row + row * convOutSize + col]`
    - 相应的，将`input[curRow * N + curCol]`填入即可。
- 而对于第1，2通道而言，只需要`col_vec`的index对应按需加上`col_vec_row * K * K`，input按需加上`N * N`即可。

3.3 cuDNN

课程并未涉及cuDNN，这一部分均为网上自学，若有分析不对，还请海涵。

1. **cuDNN 句柄初始化：**

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

这里创建了一个cuDNN句柄 `cudnn`，它将用于后续的cuDNN API调用。

2. **输入张量描述符初始化：**

```
cudnnTensorDescriptor_t in_desc;
cudnnCreateTensorDescriptor(&in_desc);
cudnnSetTensor4dDescriptor(in_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, 1,
channel, N, N);
```

创建并设置了一个4D张量描述符 `in_desc`，用于描述输入张量。这里 `CUDNN_TENSOR_NCHW` 表示数据格式为批大小、通道数、高度和宽度。

3. **卷积核描述符初始化：**

```
cudnnFilterDescriptor_t filt_desc;
cudnnCreateFilterDescriptor(&filt_desc);
cudnnSetFilter4dDescriptor(filt_desc, CUDNN_DATA_FLOAT, CUDNN_TENSOR_NCHW, 1,
channel, K, K);
```

创建并设置了一个卷积核描述符 `filt_desc`，用于描述卷积核。这里的卷积核大小为 `channel x K x K`。

4. **卷积操作描述符初始化：**

```
cudnnConvolutionDescriptor_t conv_desc;
cudnnCreateConvolutionDescriptor(&conv_desc);
cudnnSetConvolution2dDescriptor(conv_desc, 1, 1, 1, 1, 1, 1, CUDNN_CONVOLUTION,
CUDNN_DATA_FLOAT);
```

创建并设置了一个卷积操作描述符 `conv_desc`，包括卷积的填充（padding）、步幅（stride）等参数。

5. **计算输出张量的维度：**

```
int out_n, out_c, out_h, out_w;
cudnnGetConvolution2dForwardOutputDim(conv_desc, in_desc, filt_desc, &out_n,
&out_c, &out_h, &out_w);
```

通过输入张量、卷积核和卷积操作描述符计算输出张量的维度。

6. **分配输出张量的内存：**

```
float* out_data;
cudaMalloc(&out_data, out_n * out_c * out_h * out_w * sizeof(float));
```

7. **输出张量描述符初始化：**

```
cudnnTensorDescriptor_t out_desc;
cudnnCreateTensorDescriptor(&out_desc);
cudnnSetTensor4dDescriptor(out_desc, CUDNN_TENSOR_NCHW, CUDNN_DATA_FLOAT, out_n,
out_c, out_h, out_w);
```

8. **获取卷积前向操作的工作空间大小并分配内存：**

```
size_t ws_size;
cudnnGetConvolutionForwardWorkspaceSize(cudnn, in_desc, filt_desc, conv_desc,
out_desc, CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM, &ws_size);

float* ws_data;
cudaMalloc(&ws_data, ws_size);
```

9. **执行卷积前向操作：**

```
float alpha = 1.0f;
float beta = 0.0f;

double iStart_cudnn = cpuSecond();
cudnnConvolutionForward(
    cudnn,
    &alpha,
    in_desc, matrix_dev,
    filt_desc, kernel_host,
    conv_desc,
    CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM,
    ws_data, ws_size,
    &beta,
    out_desc, out_data
);
double iElaps_cudnn = cpuSecond() - iStart_cudnn;
printf("cuDNN \t\t\tExecution Time elapsed %f sec\n", iElaps_cudnn);
```

`cudnnConvolutionForward`函数执行卷积操作，并记录操作时间。这里使用的算法是`CUDNN_CONVOLUTION_FWD_ALGO_IMPLICIT_GEMM`。

10. **将输出数据从设备内存拷贝到主机内存：**

```
float* host_out_data = new float[out_n * out_c * out_h * out_w];
cudaMemcpy(host_out_data, out_data, out_n * out_c * out_h * out_w * sizeof(float),
cudaMemcpyDeviceToHost);
```

11. **释放资源：**

```
cudaFree(ws_data);
cudaFree(out_data);
cudnnDestroyTensorDescriptor(out_desc);
cudnnDestroyConvolutionDescriptor(conv_desc);
cudnnDestroyFilterDescriptor(filt_desc);
cudnnDestroyTensorDescriptor(in_desc);
cudnnDestroy(cudnn);

delete[] host_out_data;
```

4. 实验结果

这里我听从老师的建议，有一个小优化。考虑到filter，也就是卷积是不变的。我们可以将filter设置为`__constant__`，我们都知道，常量内存是快于全局内存的。

> 常量内存的访问是通过硬件缓存优化的。每个 CUDA 设备都有一个专门的常量内存缓存，通常是 8 KB。所有多处理器（SMs）共享这一缓存。当一个线程块访问常量内存时，如果数据已经在缓存中，则访问速度非常快。如果不在缓存中，数据会从全局内存加载到缓存中，然后提供给线程。

因此，除了矩阵的规模，方法不同以外，我们还可以引入`__constant__`加入对比。

4.1 `__device__` kernel

1. N = 128:

```
CPU                     Execution Time elapsed 0.001011 sec
CUDA                    Execution Time elapsed 0.000022 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000017 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000012 sec
Check result success!
```

我引入CPU的计算，来检查实验结果的正确性。可以看到，我的三个实验均是正确的。

2. N = 256:

```
CPU                     Execution Time elapsed 0.004188 sec
CUDA                    Execution Time elapsed 0.000039 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000029 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000020 sec
Check result success!
```

3. N = 512:

```
CPU                     Execution Time elapsed 0.017433 sec
CUDA                    Execution Time elapsed 0.000067 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000047 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000040 sec
Check result success!
```

4. N = 1024:

```
CPU                     Execution Time elapsed 0.072873 sec
CUDA                    Execution Time elapsed 0.000136 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000110 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000095 sec
Check result success!
```

5. N = 2048:

```
CPU                     Execution Time elapsed 0.307706 sec
CUDA                    Execution Time elapsed 0.000478 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000284 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000138 sec
Check result success!
```

表格：

\N	128	256	512	1024	2048
CPU	0.001011	0.004188	0.017433	0.072873	0.307706
direct	0.000022	0.000039	0.000067	0.000136	0.000478
im2col	0.000017	0.000029	0.000047	0.000110	0.000284
cuDNN	0.000012	0.000020	0.000040	0.000095	0.000138

可视化：

可以看到，随着矩阵规模的增大，不同实现方法的运行效率差距越来越大。cuDNN优于im2col，优于划窗法。

4.2 `__constant__` kernel

1. N = 128:

```
CPU                     Execution Time elapsed 0.000913 sec
CUDA                    Execution Time elapsed 0.000028 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.000017 sec
Check result success!
cuDNN                   Execution Time elapsed 0.556579 sec
Check result success!
```

2. N = 256:

```
CPU                     Execution Time elapsed 0.004185 sec
CUDA                    Execution Time elapsed 0.000042 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.040168 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000004 sec
Check result success!
```

3. N = 512:

```
CPU                     Execution Time elapsed 0.016942 sec
CUDA                    Execution Time elapsed 0.000050 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.041226 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000004 sec
Check result success!
```

4. N = 1024:

```
CPU                     Execution Time elapsed 0.075331 sec
CUDA                    Execution Time elapsed 0.000120 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.039806 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000004 sec
Check result success!
```

5. N = 2048:

```
CPU                     Execution Time elapsed 0.305270 sec
CUDA                    Execution Time elapsed 0.000366 sec
Check result success!
CUDA_im2col:            Execution Time elapsed 0.038586 sec
Check result success!
cuDNN                   Execution Time elapsed 0.000004 sec
Check result success!
```

表格：

\N	128	256	512	1024	2048
CPU	0.000913	0.004185	0.016942	0.075331	0.305270
direct	0.000028	0.000042	0.000050	0.000120	0.000366
im2col	0.000017	0.000031	0.000035	0.000089	0.000251
cuDNN	0.000012	0.000022	0.000026	0.000062	0.000113

可视化：

可以看到，图像的大体趋势是和`__device__`（即全局内存）一致。但是，感谢于高速的常量缓存，在我们实验中，`__constant__`的实现效率是要高于`__device__`（全局内存）大约10%的。

5. 实验感想

在这次实验中，我深入探讨了滑窗法、im2col方法和cuDNN库在CUDA环境下实现图像卷积操作的性能表现。通过实验分析了线程块大小、图像大小、访存方式和数据划分方式等因素对卷积性能的影响，发现了合理配置这些参数对提升计算效率的显著作用。特别是，在卷积核不变的情况下，将卷积核数据存储在常量内存中，可以显著提升卷积运算的速度，尤其是在处理大规模图像时，这种优化带来的性能提升尤为明显。

通过本次实验，我不仅掌握了CUDA编程的实际操作技能，还对并行计算的优化策略有了更深入的理解。实验结果表明，不同方法在处理不同规模数据时的性能表现差异显著，合理选择和优化算法对于提高计算效率至关重要。这些经验和技能对我未来从事高性能计算和并行编程的研究和应用将有重要的指导意义。