

任务 1：MPI+OpenMP 实现卷积操作

一、卷积结果展示：

测试用例展示 1：输入为 $32 \times 32 \times 3$ 随机数，Kernel (Filter) 为 $3 \times 3 \times 3$ 随机数，步幅(stride) 为 1，填充(padding) 为 1。其输入输出如下图所示。

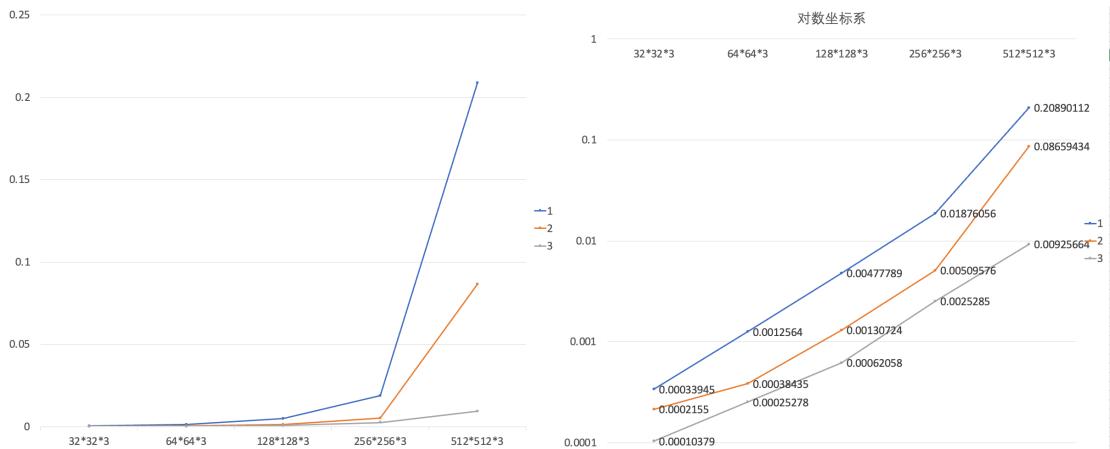
测试用例展示 2：输入大小 $32 \times 32 \times 3$ ，内容为全 1。Kernel (Filter) 大小 $3 \times 3 \times 3$ ，内容为全 1。步幅 (stride) 为 1，填充 (padding) 为 1。其输出如下图所示。

可以看到，各个位置的输出均正确

二、计算时间和分析

1、只使用 mpi

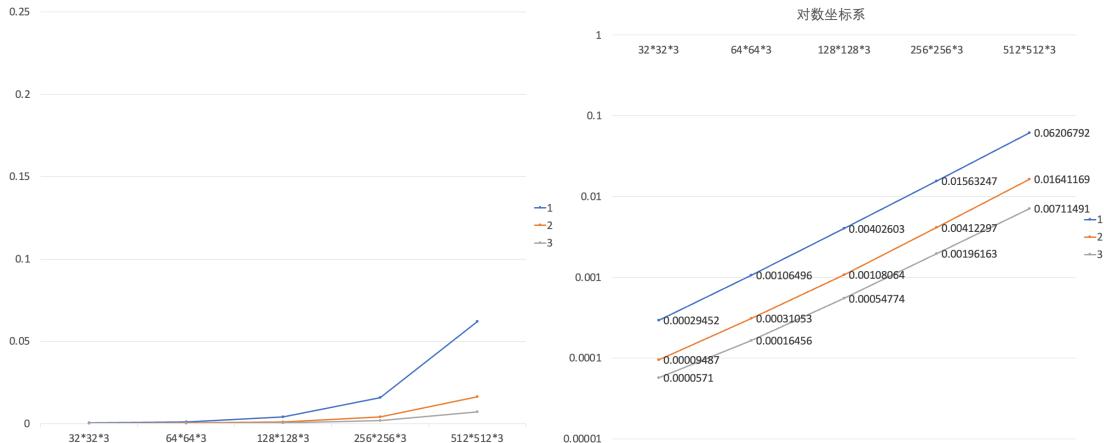
步幅 \ 输入维度	32*32*3	64*64*3	128*128*3	256*256*3	512*512*3
1	0.00033945	0.00125640	0.00477789	0.01876056	0.20890112
2	0.00021550	0.00038435	0.00130724	0.00509576	0.08659434
3	0.00010379	0.00025278	0.00062058	0.00252850	0.00925664



可以看到，当步幅为 3 时运行时间随输入维度增长最慢，这主要原因是由运算量决定的，例如，表中输入维度为 128*128*3，步幅为 1，填充为 1 时，输出维度为 128*128；输入维度为 256*256*3，步幅为 2，填充为 1 时，输出维度也为 128*128；输入维度为 512*512*3，步幅为 3，填充为 1 时，输出维度为 171*171。这三者的运行用时分别为 0.0047、0.0050、0.0092。可见，运行时间的主要影响因素为运算量，运算量相近的运行用时会相近。相同运算量下，输入维度大的运行时间更长，其原因可能是参数传递和 cache 缺失率等影响。

2、使用 mpi+openmp

步幅 \ 输入维度	32*32*3	64*64*3	128*128*3	256*256*3	512*512*3
1	0.00029452	0.00106496	0.00402603	0.01563247	0.06206792
2	0.00009487	0.00031053	0.00108064	0.00412297	0.01641169
3	0.00005710	0.00016456	0.00054774	0.00196163	0.00711491



可以看到，增加结合使用了 openmp 后性能得到提升，运行用时得到 3 至 5 倍的优化。同时从对数坐标系图像可以明显看出，随着输入维度的增大，使用 openmp 后，运行用时上升的趋势得到减缓，即运行用时的增加率得到下降。这与我们程序的设计是一致的，程序使用 mip 创建 3 个进程分别对应 3 个卷积核的运算，对于输入维度的增加，mip 并无法起到优化。对于卷积运算的优化我们使用 openmp 创建多个线程并行化运算，所以增加了 openmp 后，对于输入维度上升导致卷积运算变得更复杂，其能够起到优化作用。

3、openmp 设置对运行时间的影响

通过在#pragma omp parallel for 后增加 num_threads、private、shared、schedule、collapse 等设置的排列组合，以及用#pragma omp parallel 将循环括起来和改变#pragma omp parallel for 的数量和位置等方法，发现在当前环境下使用下方设置具有较好性能。

```

void convolution(int input[INPUT_SIZE][INPUT_SIZE][CHANNEL], int kernel[KERNEL_SIZE][KERNEL_SIZE][CHANNEL], int
output[OUTPUT_SIZE][OUTPUT_SIZE])
{
    int i, j, m, n, c, sum;

    #pragma omp parallel for num_threads(omp_get_num_threads()) private(i, j, m, n, c) shared(input, kernel, output)
    for (i = 0; i < OUTPUT_SIZE; i++)
    {
        #pragma omp parallel for num_threads(omp_get_num_threads()) private(i, j, m, n, c) shared(input, kernel, output)
        for (j = 0; j < OUTPUT_SIZE; j++)
        {
            for (c = 0; c < CHANNEL; c++)
            {
                sum = 0;
                for (m = 0; m < KERNEL_SIZE; m++)
                {
                    for (n = 0; n < KERNEL_SIZE; n++)
                    {
                        int x = i * STRIDE - PADDING + m;
                        int y = j * STRIDE - PADDING + n;
                        if (x >= 0 && x < INPUT_SIZE && y >= 0 && y < INPUT_SIZE)
                        {
                            sum += input[x][y][c] * kernel[m][n][c];
                        }
                    }
                }
                output[i][j] += sum;
            }
        }
    }
}

```

这里让程序自行决定了线程数，相比比手动设置线程数其效果更好，可能的原因是由于输入维度不断变化的关系。同时这里尝试了改变各个 for 循环的位置，比如将 `for (c = 0; c < CHANNEL; c++)` 提到第 1 个 for 循环的位置等等。最终发现无论各个 for 循环的位置如何，将变量 i 和 j 对应的 for 循环并行化会有较好的效果。这与代码逻辑也是符合的，不同 i 和 j 对应了每次不同位置的卷积核的运算操作，将卷积核每次在不同位置的运算操作进行并行化是较为合理的。如果对其他 for 循环进行并行化处理，那只是将一次卷积核运算操作中的某些步骤进行并行化，比如对每 $3 \times 3 \times 3$ 中每一列的计算并行化处理，这样处理由于数据较少所以将比较难起到优化效果，线程创建的开销也容易会大于优化。所以最终使用了上面所示的只对每次卷积核的运算操作进行并行化的方法，上面代码的实现方式也可以用 collapse(2) 代替达到类似效果。

三、实现思路解释

使用 mpi 创建 3 个进程，每个进程处理一个卷积核的运算。0 号进程作为主进程，其随机生成输入和卷积核，并使用 mpi 将输入和卷积核传送给其他进程。之后各个进程使用 openmp 并行化进行卷积运算，将运算结果存储在二维矩阵 output 中，最后各个进程通过 MPI_Gather 函数调用将各自的 output 矩阵合并到 result 三维矩阵中，最后将结果输出。详见下方主函数注释。

```

int main(int argc, char **argv)
{
    // 初始化 MPI 并获取进程号 (rank) 和进程的总数 (size)
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 3) // 检查进程的总数
    {
        printf("Error: Number of processes must be at least 4\n");
        MPI_Finalize();
        return 1;
    }

    int input[INPUT_SIZE][INPUT_SIZE][CHANNEL];           // 输入矩阵
    int kernel[KERNEL_SIZE][KERNEL_SIZE][CHANNEL];       // 卷积核
    int output[OUTPUT_SIZE][OUTPUT_SIZE] = {0};           // 单个卷积核的输出
    int result[KERNEL_AMOUNT][OUTPUT_SIZE][OUTPUT_SIZE]; // 输出矩阵

    if (rank == 0)
    {
        Generate_Data(INPUT_SIZE, INPUT_SIZE, CHANNEL, input); // 主进程随机生成输入数据
        // printf("-----input-----\n");
        // print_array(INPUT_SIZE, INPUT_SIZE, CHANNEL, input);

        for (int i = 1; i <= size - 1; i++)
        {
            Generate_Data(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernel); // 主进程随机生成卷积核
            // printf("-----kernel_%d-----\n", i + 1);
            // print_array(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernel);
            MPI_Send(&kernel[0][0][0], KERNEL_SIZE * KERNEL_SIZE * CHANNEL, MPI_INT, i, 0, MPI_COMM_WORLD); // 将卷积核发送给对应的进程
        }

        Generate_Data(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernel); // 主进程随机生成自己的卷积核
        // printf("-----kernel_1-----\n");
        // print_array(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernel);
    }

    MPI_Bcast(&input[0][0], INPUT_SIZE * INPUT_SIZE * CHANNEL, MPI_INT, 0, MPI_COMM_WORLD); // 其他进程接收输入数据

    if (rank != 0) // 其他进程从主进程接收对应卷积核
    {

```

```

MPI_Recv(&kernel[0][0][0], KERNEL_SIZE * KERNEL_SIZE * CHANNEL, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
}

MPI_Barrier(MPI_COMM_WORLD); // 进程同步
double start_time, end_time, total_time;
start_time = omp_get_wtime();

Convolution(input, kernel, output); // 卷积运算

end_time = omp_get_wtime();
total_time = end_time - start_time; // 计时

MPI_Gather(output, OUTPUT_SIZE * OUTPUT_SIZE, MPI_INT, result, OUTPUT_SIZE * OUTPUT_SIZE, MPI_INT, 0,
MPI_COMM_WORLD); // 传送运算结果

if (rank == 0) // 主进程输出相应信息
{
    // printf("-----output-----\n");
    // print_result(result);
    printf("输入维度:%d*%d*3\n", INPUT_SIZE, INPUT_SIZE);
    printf("步幅:%d\n", STRIDE);
    printf("填充:%d\n", PADDING);
    printf("运行时间:      %.8f\n", total_time);
}

MPI_Finalize();
return 0;
}

```

任务 2: im2col 方法实现卷积

一、卷积结果展示：

测试用例展示 1：输入为 $32 \times 32 \times 3$ 随机数，Kernel (Filter) 为 $3 \times 3 \times 3$ 随机数，其输入输出如下图所示。

测试用例展示 2：输入大小 $32 \times 32 \times 3$ ，内容为全 1。Kernel (Filter) 大小 $3 \times 3 \times 3$ ，内容为全 1。其输出如下图所示。

Layer 3a
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

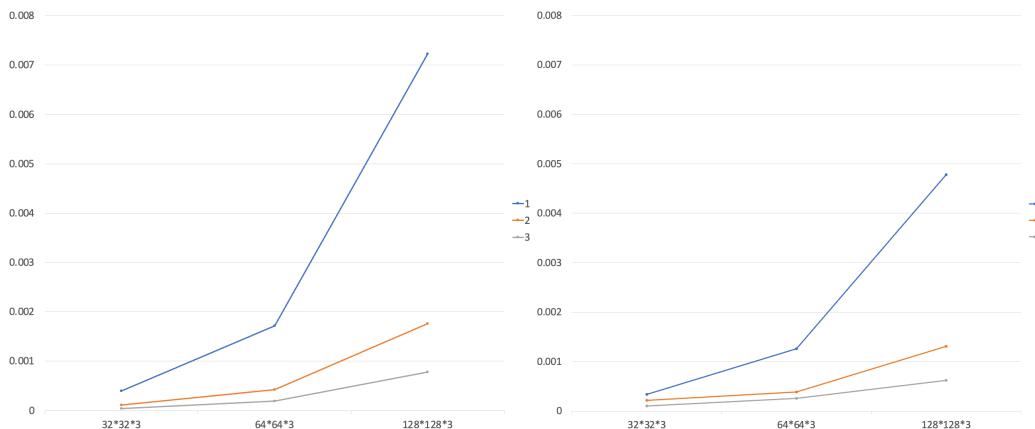
可以看到，各个位置的输出均正确

二、计算时间和分析

由于超算习堂分配内存有限，这里最大输入只能到 $128 \times 128 \times 3$

1、没有使用 openmp (右图为任务一中只使用 mpi 情况)

步幅\输入维度	32*32*3	64*64*3	128*128*3
1	0.00039615	0.00171657	0.00722349
2	0.00011104	0.00042295	0.00176149
3	0.00004466	0.00019473	0.00077705

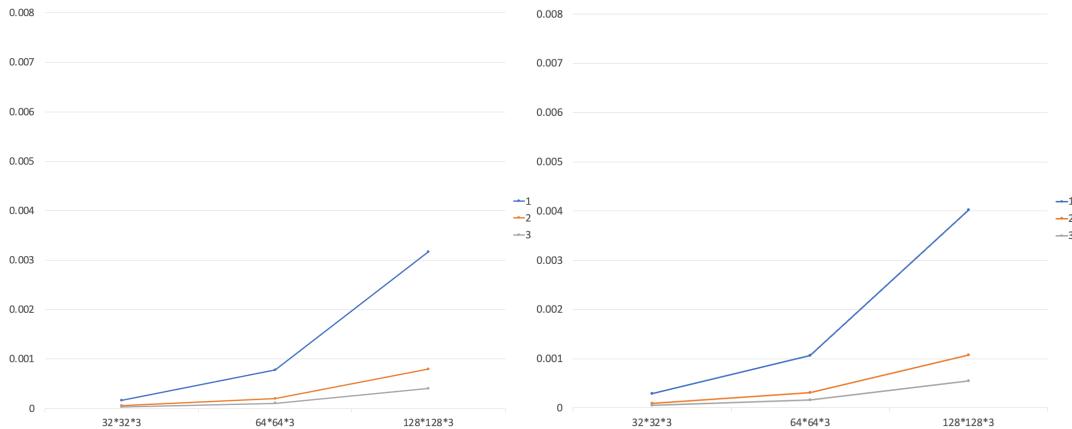


可以看到左图 `im2col` 方法相比右图滑窗法+`mpi` 并行化耗时更久，但是由于实验中处理的是三个卷积核，滑窗法由于使用了 `mpi` 并行化，可以将其用时粗略视为是一个卷积核一次卷积运算的时间，而 `im2col` 方法是一次性计算了三个卷积核的卷积运算结果，并且 `im2col` 方法耗时远低于滑窗法+`mpi` 耗时的 3 倍，

可见 im2col 方法的运算速度会更快，但这里要求对应数据矩阵已使用 im2col 方法处理，其本质上应用空间换时间的思想。

1、使用 openmp (右图为任务一中使用 mpi+openmp 情况)

步幅\输入维度	32*32*3	64*64*3	128*128*3
1	0.00016969	0.00077924	0.00317409
2	0.00005480	0.00020015	0.00080163
3	0.00003238	0.00010745	0.00040461



可以看到，使用 openmp 并行化处理后，im2col 方法的运行用时得到很大的优化。和右图滑窗法+mpi+openmp 相比，im2col 方法+openmp 效果会更好一些，im2col 方法一次计算三个卷积核结果的用时已低于滑窗法使用 mpi 并行化后计算一个卷积核一次卷积计算的近似时间，可见 im2col 方法的计算性能会更好。不过实验中没有将 im2col 方法处理输入数据的时间计算上，并且由于数据量较小省略了线程创建的开销时间，如果将 im2col 方法处理输入数据的时间算上，im2col 方法的用时会比滑窗法+mpi+openmp 更长，当然也可以用 openmp 对 im2col 方法处理输入数据的过程并行化。总体来说 im2col 方法在卷积核数量多的情况下会更有优势，卷积核数量足够多时可以将 im2col 方法处理输入数据的时间抹平。

三、实现思路解释

实验中，首先随机生成输入矩阵和卷积核，然后将它们用 im2col 方法处理。之后将处理得到的 im2col 矩阵用于卷积运算，得到输出数据对应的 im2col 矩阵，将 im2col 输出矩阵进行转换，转换为卷积运算中正常格式的输出矩阵，最后即可将运算结果进行输出。

```
int main()
{
    int seed = time(0);
    srand(seed); // 使用当前时间作为随机数种子

    int input[INPUT_SIZE][INPUT_SIZE][CHANNEL]; // 输入矩阵
    int kernels[KERNEL_AMOUNT][KERNEL_SIZE][KERNEL_SIZE][CHANNEL]; // 卷积核
    int output[KERNEL_AMOUNT][OUTPUT_SIZE][OUTPUT_SIZE]; // 输出矩阵

    Generate_Data(INPUT_SIZE, INPUT_SIZE, CHANNEL, input); // 随机生成输入数据
    for (int i = 0; i < KERNEL_AMOUNT; i++)
    {
        Generate_Data(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernels[i]); // 随机生成卷积核
    }
```

```

// printf("-----input-----\n");
// print_array(INPUT_SIZE, INPUT_SIZE, CHANNEL, input);
// for (int i = 0; i < KERNEL_AMOUNT; i++)
//{
//     printf("-----kernel_%d-----\n", i + 1);
//     print_array(KERNEL_SIZE, KERNEL_SIZE, CHANNEL, kernels[0]);
//}

int input_im2col[INPUT_SIZE * INPUT_SIZE][KERNEL_SIZE * KERNEL_SIZE * CHANNEL];           // im2col 方法处理后
的输入矩阵
int kernels_im2col[KERNEL_AMOUNT][KERNEL_SIZE * KERNEL_SIZE * CHANNEL];                  // im2col 方法处
理后的三个卷积核
int output_im2col[KERNEL_AMOUNT][INPUT_SIZE - KERNEL_SIZE + 1][INPUT_SIZE - KERNEL_SIZE + 1]; // im2col 方法得到
的输出矩阵

im2col_input(input, input_im2col);      // 用 im2col 方法处理输入矩阵
im2col_kernel(kernels, kernels_im2col); // 用 im2col 方法处理卷积核

double start_time, end_time, total_time;
start_time = omp_get_wtime();
convolve(input_im2col, kernels_im2col, output_im2col); // im2col 方法进行卷积运算
end_time = omp_get_wtime();
total_time = end_time - start_time;

anti_im2col_output(output_im2col, output); // 将 im2col 方法得到的输出矩阵还原为正常卷积运算格式

// print_output(output);
printf("输入维度:%d*%d*3\n", INPUT_SIZE, INPUT_SIZE);
printf("运行时间:      %.8f\n", total_time);

return 0;
}

```