中山大学计算机学院2023 年春季超级计算原理与实践期末编程作业本科生实验报告

课程名称：超级计算机原理与操作

教学班级	专业（方向）	学号	姓名
2班	计算机科学与技术	21307174	刘俊杰

任务1：MPI+OpenMP实现卷积操作

任务1 实验内容

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CPU上利用MPI+OpenMP的并行方式实现卷积操作，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MPI+OpenMP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，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

报告要求：提供实现思路，适当解释核心代码（切忌大段复制源码），提供代码运行截图，使用图表统计输出结果，并对实验结果给出自己的理解与解释。

Answer:

1. 实验过程思路

(1)本实验使用的是MPI和openmp完成的,首先我们先获得进程的个数和当前进程的标号(本实验我们设置为4个进程,实验结果也是在4个进程下实现的)。

(2)接着我们创建对应尺寸的input三维数组和kernel三维数组(尺寸可直接在代码中修改或代码中有输入尺寸的代码，可将该段代码注释删去，然后手动输入尺寸)，然后初始化input和kernel(可随机生成也可手动输入)。(手动输入时需要只能在0进程输入，通过0进程将input和kernel通过信息传递传递给其他进程)

(3)接着进入循环，将stride步长从1取到3，为规定好选取的padding,我们在在这里选取能使得输出尺寸和输入尺寸一样的padding。

(4)创建并初始化output三维数组。

(5)开始使用滑窗法多进程并行计算卷积(每个进程负责计算一部分位置的output,这部分使用了#pragma omp parallel for 加速计算和 #pragma omp critical保证互斥)，0进程记录卷积计算的开始时间，当每个进程计算完毕

时，记录卷积结束时间(并行中记录时间不宜用clock(),并行中使用会导致记录时间比实际时间长)。最后每个进程获得负责计算的结果后，将结果通过MPI_Reduce归约到0进程中。

(6)每次循环stride得出结果后，0进程输出output和卷积所花费的时间。

(7)最后释放分配的资源。

2. 核心代码分析

(主要展示关于卷积的核心代码，其他例如input、kernel创建和初始化、输出output等在提交代码中展示)

(1)计算padding

为规定好选取的padding,我们在在这里选取能使得输出尺寸和输入尺寸一样的padding。

```
int calculate_padding(){//计算使得输入和输出高度尺寸相同的padding
    int p=0;
    while((HEIGHT-KERNEL_SIZE+2*p)/stride+1!=HEIGHT){
        p++;
    }
    return p;
}
```

(2)滑窗法计算卷积(实现细节在代码注释中)

首先根据该进程的标号和进程数，然后将输出output [d] [i] [j] (d取0,1,2)，即将输出的2d维度三个通道的三个位置当成一个计算单元，将所有这样output_height*output_weight个计算单元分给不同的进程计算。

前extra_n=output_sum%thread_count个进程计算average_n+1个单元，其他进程平均计算average_n=output_sum/thread_count个单元。

根据my_rank该线程的进程标号计算获得计算的输出位置的开头和结尾(不包括计算结果)

然后通过滑窗法计算(注意使用padding的填充)，注意修改output时，需要将这段代码改为互斥区(#pragma omp critical) 保证多线程对output_1d的互斥访问，防止造成冲突。

计算完成后，调用MPI_Reduce()将结果归约到0进程。

```
void conv_parallel(int ***input,int ****kernel,int ***output,int comm_sz,int my_rank,int**output_2d,int n){//并行进行卷积运算

    int process_count=comm_sz;//获取进程数

    //将输出output[d][i][j](d取0,1,2)，即将输出的三个通道的同一位置，将所有这样output_height*output_weight个输出位置分给不同的进程计算
    int output_sum=output_height*output_weight;//总共要计算的位置个数
    int average_n=output_sum/process_count;//平均每个取进负责的计算输出位置的个数
    int extra_n=output_sum%process_count;//前extra_n个取进多负责一个位置
    int local_start,local_end;//每个取进负责计算的输出位置的开头和结尾(不包括计算结果)
```

```
    if(my_rank<extra_n){//前extra_n个取进多负责一个位置
        local_start=my_rank*average_n+my_rank;
        local_end=local_start+average_n+1;
    }
    else{//其他取进计算average_n个位置
        local_start=my_rank*average_n+extra_n;
        local_end=local_start+average_n;
    }

    //存储计算结果
    int*output_1d=(int*)malloc(sizeof(int)*(output_height*output_weight));
    for(int i=0;i<output_height*output_weight;i++)output_1d[i]=0;

        #pragma omp parallel for
        for(int d=0;d<output_depth;d++){//选取计算输出的哪一个通道
            for(int i=local_start;i<local_end;i++){//选取该进程负责计算的位置
                int row=i/output_weight;//计算位置的行下标
                int col=i%output_weight;//计算位置的列坐标
                for(int k1=0;k1<KERNEL_SIZE;k1++){
                    for(int k2=0;k2<KERNEL_SIZE;k2++){
                        int input_row=-padding+stride*row+k1;//对应输入位置行下标
                        int input_col=-padding+stride*col+k2;//对应输入位置列下标

if(input_row>=HEIGHT||input_col>=WEIGHT||input_row<0||input_col<0)output_1d[i]+=pa
dding_num*kernel[n][d][k1][k2];//对应输入位置位于padding填充的位置

                        #pragma omp critical
                        else  output_1d[i] += input[d][input_row]
[input_col]*kernel[n][d][k1][k2];

                    }
                }
            }
        }
        //将计算结果归约到0进程中
        MPI_Reduce(output_1d, output_2d[n],  output_height*output_weight, MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD );

}
```

(3)主函数中调用计算卷积

将stride从1循环到3，每次循环中每个进程通过循环调用函数计算不同通道的output。

当所有进程完成计算后，结果通过MPI_Reduce()归约到0进程(若有的进程MPI_Reduce未完成，其他就会阻塞，保证了各进程之间的同步运算)，0进程输出output和运行时间。

```
//用步长分别为1,2,3计算卷积
    for(stride=1;stride<=3;stride++){
    //计算适应的padding
```

```
    padding=calculate_padding();
    output_height=(HEIGHT-KERNEL_SIZE+2*padding)/stride+1;
    output_weight=(WEIGHT-KERNEL_SIZE+2*padding)/stride+1;
    output_depth=KERNEL_NUM;


    //计算output的尺寸 stride padding
    if(my_rank==0)printf("The stride is %d,the padding is %d\n",stride,padding);
    if(my_rank==0)printf("The size of output is %d*%d*%d\n",
output_depth,output_height,output_weight);

    //创建初始化output output_2d用来接收归约结果
    int**output_2d=construct_two(output_depth,output_height*output_weight);
    int ***output=construct_three(output_depth,output_height,output_weight);
    initialize_output(output);
    initialize_output_col2(output_2d);

    //记录卷积计算开始时间
    struct timeval start,end;
    if(my_rank==0)gettimeofday(&start, NULL );

    //并行计算卷积
    for(int
i=1;i<=3;i++)conv_parallel(input,kernel,output,comm_sz,my_rank,output_2d,i-1);

    //记录卷积结束时间
    if(my_rank==0){
    gettimeofday(&end, NULL );
    //计算输出卷积时间
    long timeuse =1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
start.tv_usec;
    printf("Costing time=%fs\n",timeuse /1000000.0);
    printf("------------------------\n");
    trans_output(output,output_2d);
    show_output(output);
    }

    }
```

3. 运行结果展示

运行结果都是使用了4个进程

(1)编译运行

```
----------------------
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpicc -g -o ex1.o ex1.c
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpiexec -n 4 ./ex1.o
```

(2)输入和卷积核

为方便我们观察计算结果是否正确，我们这里的input和kernel的所有位置都是1，padding填充的为0

input:

```
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpicc -g -o ex1.o ex1.c
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpiexec -n 4 ./ex1.o
The size of input is 3*5*5
The size of kernel is 3*3*3*3
The input:
The 1 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
The 2 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
The 3 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
----------------------
```

kernel:

```
The kernel:
The Num = 1
The Depth = 1
1 1 1
1 1 1
1 1 1
The Depth = 2
1 1 1
1 1 1
1 1 1
The Depth = 3
1 1 1
1 1 1
1 1 1
------------
The Num = 2
The Depth = 1
1 1 1
1 1 1
1 1 1
The Depth = 2
1 1 1
1 1 1
1 1 1
The Depth = 3
1 1 1
1 1 1
1 1 1
------------
The Num = 3
The Depth = 1
1 1 1
1 1 1
1 1 1
The Depth = 2
1 1 1
1 1 1
1 1 1
The Depth = 3
1 1 1
1 1 1
1 1 1
------------
```

(3)stride=1时的output和所用时间:

```
------------
The stride is 1,the padding is 1
The size of output is 3*5*5
Costing time=0.000405s
-----------------------
The output:
The 1 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
The 2 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
The 3 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
----------------------
```

(4)stride=2时的output和所用时间:

```
----------------------
The stride is 2,the padding is 3
The size of output is 3*5*5
Costing time=0.000160s
------------------------
The output:
The 1 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
The 2 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
The 3 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
----------------------
```

(5)stride=3时的output和所用时间:

```
----------------------
The stride is 3,the padding is 5
The size of output is 3*5*5
Costing time=0.000014s
------------------------
The output:
The 1 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
The 2 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
The 3 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
----------------------
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$
```

通过计算可以得出代码的运行结果的输出是正确的

串行计算

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.001407s	0.000747s	0.000556s
64	0.005950s	0.002744s	0.002059s
128	0.026009s	0.012124s	0.011140s
256	0.115134s	0.060120s	0.042222s
512	0.392373s	0.235712s	0.181902s

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
1024	1.599387s	0.891970s	0.658294s
2048	6.550370s	2.043790s	1.620010s

滑窗法并行计算不同尺寸input下运行时间(4个进程)

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.000462s	0.000380s	0.000369s
64	0.014010s	0.002061s	0.001964s
128	0.007777s	0.009194s	0.008088s
256	0.032521s	0.046304s	0.036393s
512	0.0150963s	0.110651s	0.099870s
1024	0.459417s	0.34574s	0.333663s
2048	1.774545s	1.8033864s	1.442759s

4. 实验结果分析

可以看到串行计算和并行计算随着输入尺寸的加大，运行的时间的也越久。

在输入尺寸较小时，串行计算时略快于并行计算的，这是因为并行计算中进程之间的通信和多个线程和进程的创建都消耗了时间。但随着输入尺寸的增大，并行计算的效果体现出来了，随着输入尺寸的增大并行计算越来越快于串行计算，这是因为并行计算将大量的任务分给多个进程同时进行，在工作量大时，大大提升了运算效率。

任务2：im2col方法实现卷积

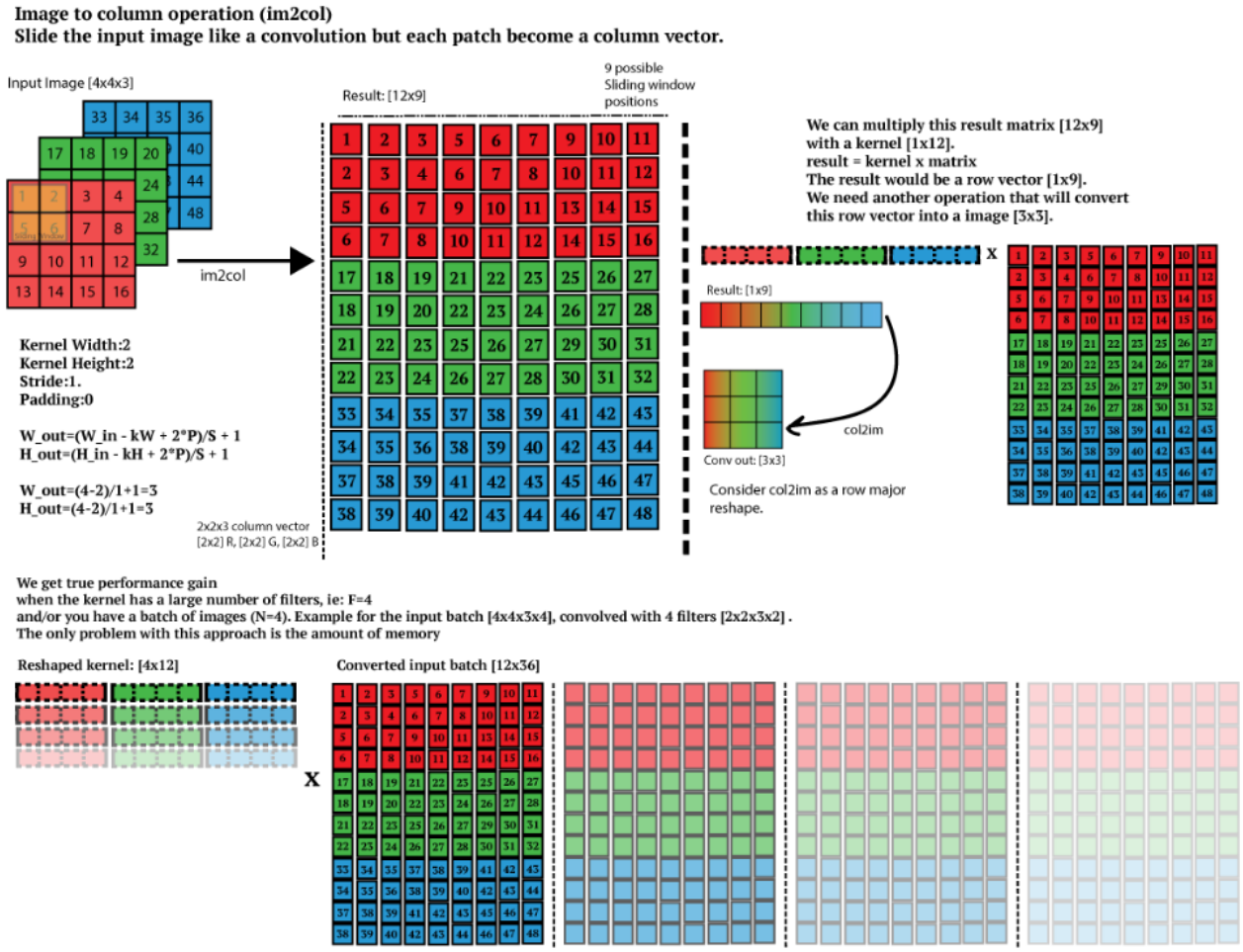
任务2 实验内容

任务二使用im2col方法结合任务一实现的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，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，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。



Answer:

1.实验过程思路

该实验与滑窗法卷积的最大不同在于使用了imcol2，要将输入、卷积核和输出都放在二位数组中，使用类似于矩阵乘法的方法得出答案output，再将output转为三维数组，目的是使用好空间局部性、将一整块内存调入cache中且这些数据都是时间上邻近访问的、减少cache未命中的次数，从而加块运行时间

c语言中数组的元素存放是按行来进行的，为了使用好空间局部性，我们将input转为二位数组input_col2,kernel也用二维数组存储，将计算结果放入output_col2二维数组中，再将output_col2转化为三维数组output。kernel是3*27的数组，input_col2是[output_height *output_weight, 27]的数组，在物理结构上我们这样尺寸能利用好局部性，在计算时逻辑上像矩阵计算。

(1)本实验使用的是MPI和openmp完成的,首先我们先获得进程的个数和当前进程的标号(本实验我们设置为4个进程,实验结果也是在4个进程下实现的)。

(2)接着我们创建对应尺寸的input三维数组和kernel二维数组，初始化 input和kernel，再将input转为input_col2的二维数组。

(3)接着进入循环，将stride步长从1取到3，为规定好选取的padding,我们在在这里选取能使得输出尺寸和输入尺寸一样的padding。

(4)创建并初始化output三维数组，创建并初始化output_col2二维数组用来存储归约的结果。

(5)开始使用滑窗法多进程并行计算卷积(每个进程负责计算一部分位置的output)，0号进程记录卷积计算的开始时间，当每个线程计算完毕时，记录卷积结束时间(并行中记录时间不宜用clock(),并行中使用会导致记录时间比实际时间长)。最后每个进程获得负责计算的结果后，将结果通过MPI_Reduce归约到0进程中。

(6)每次循环stride得出结果后，0进程输出output和卷积所花费的时间。

(7)最后释放分配的资源。

2. 核心代码分析

(1)计算padding

为规定好选取的padding,我们在在这里选取能使得输出尺寸和输入尺寸一样的padding。

```
int calculate_padding(){//计算使得输入和输出高度尺寸相同的padding
    int p=0;
    while((HEIGHT-KERNEL_SIZE+2*p)/stride+1!=HEIGHT){
        p++;
    }
    return p;
}
```

(2)将三维数组input转为二位数组input_col2

```
void transform_input_col2(int***input,int**input_col){//将三维input转为二维
input_col2
    for(int i=0;i<output_height;i++){
          for(int j=0;j<output_weight;j++){
            int n=0;
            for(int d=0;d<DEPTH;d++){
            for(int k1=0;k1<KERNEL_SIZE;k1++){
                for(int k2=0;k2<KERNEL_SIZE;k2++,n++){
                    int row=-padding+stride*i+k1;
                    int col=-padding+stride*j+k2;

if(row<0||col<0||col>=WEIGHT||row>=HEIGHT)input_col[i*output_weight+j]
[n]=padding_num;
                    else input_col[i*output_weight+j][n]=input[d][row][col];
                }
            }
            }
          }
        }
}
```

(3)将二维output_col2转为三维output

```
void transform_output(int**output_col2,int***output){//将二维output_col2转为三维
output
    for(int i=0;i<output_depth;i++){
        for(int j=0;j<output_height*output_weight;j++){
            output[i][j/output_weight][j%output_weight]=output_col2[i][j];
        }
    }
}
```

(4)滑窗法计算卷积(实现细节在代码注释中)

首先根据该进程的标号和进程数，然后将输出output [d] [i] [j] (d取0,1,2)，即将输出的2d维度三个通道的三个位置当成一个计算单元，将所有这样output_height*output_weight个计算单元分给不同的进程计算。

前extra_n=output_sum%thread_count个进程计算average_n+1个单元，其他进程平均计算average_n=output_sum/thread_count个单元。

根据my_rank该线程的进程标号计算获得计算的输出位置的开头和结尾(不包括计算结果)

然后通过imcol2计算(注意使用padding的填充)，注意修改output时，需要将这段代码改为互斥区(#pragma omp critical) 保证多线程对output_1d的互斥访问，防止造成冲突。

计算完成后，调用MPI_Reduce()将结果归约到0进程的output_col2二维数组。

```
void conv_parallel_col2(int**input_col2,int**kernel,int**output_col2,int
comm_sz,int my_rank,int d){//imcol2并行卷积计算

    int process_count=comm_sz;//获取进程数

     //将输出output[d][i][j](d取0,1,2)，即将输出的三个通道的同一位置，将所有这样
output_height*output_weight个输出位置分给不同的进程计算
    int output_sum=output_height*output_weight;//总共要计算的位置个数
    int average_n=output_sum/process_count;//平均每个获取进程数负责的计算输出位置的个
数
    int extra_n=output_sum%process_count;//前extra_n个获取进程数多负责一个位置
    int local_start,local_end;//每个获取进程数负责计算的输出位置的开头和结尾(不包括计算
结果)

    if(my_rank<extra_n){//前extra_n个获取进程数多负责一个位置
        local_start=my_rank*average_n+my_rank;
        local_end=local_start+average_n+1;
    }
    else{//其他获取进程数计算average_n个位置
        local_start=my_rank*average_n+extra_n;
        local_end=local_start+average_n;
    }


    int*output_1d=(int*)malloc(sizeof(int)*(output_height*output_weight));
    for(int i=0;i<output_height*output_weight;i++)output_1d[i]=0;
```

```
    #pragma omp parallel for
    for(int i=local_start;i<local_end;i++){//选取计算输出的哪一个通道
        //imcol2计算卷积
        for(int j=0;j<KERNEL_SIZE*KERNEL_SIZE*KERNEL_SIZE;j++){
            #pragma omp critical
            output_1d[i]+=input_col2[i][j]*kernel[d][j];
        }
    }
    MPI_Reduce(output_1d, output_col2[d],  output_height*output_weight, MPI_INT,
MPI_SUM, 0, MPI_COMM_WORLD );//将结果归约到0进程


}
```

(5)主函数中调用计算卷积

将stride从1循环到3，首先根据padding将input转化到对应的input_col2，每次循环中每个进程通过循环调用函数计算不同通道的output_col2，再将output_col2转为output。

当所有进程完成计算后，结果通过MPI_Reduce()归约到0进程的output_col2(若有的进程MPI_Reduce未完成，其他就会阻塞，保证了各进程之间的同步运算)，0进程输出output和运行时间。

```
    //用步长分别为1,2,3计算卷积
    for(stride=1;stride<=3;stride++){
    //计算适应的padding
    padding=calculate_padding();
    if(my_rank==0)printf("The stride is %d,the padding is %d\n",stride,padding);

    //计算output的尺寸
    output_height=(HEIGHT-KERNEL_SIZE+2*padding)/stride+1;
    output_weight=(WEIGHT-KERNEL_SIZE+2*padding)/stride+1;
    output_depth=KERNEL_NUM;

    //将三维input转为二维input_col
    int
**input_col2=construct_two(output_height*output_weight,KERNEL_SIZE*KERNEL_SIZE*KER
NEL_SIZE);
    transform_input_col2(input,input_col2);

    //创建二维output_col2和三维output
    int **output_col2=construct_two(output_depth,output_height*output_weight);
    int ***output=construct_three(output_depth,output_height,output_weight);
    initialize_output_col2(output_col2);

    //输出output尺寸
    if(my_rank==0)printf("The size of output is %d*%d*%d\n",
output_depth,output_height,output_weight);

    //记录卷积计算开始时间
    struct timeval start,end;
```

```
    if(my_rank==0)gettimeofday(&start, NULL );

    //并行计算卷积


    for(int
i=0;i<3;i++)conv_parallel_col2(input_col2,kernel,output_col2,comm_sz,my_rank,i);



    //记录卷积结束时间
    gettimeofday(&end, NULL );
    if(my_rank==0){
      transform_output(output_col2,output);
   long timeuse =1000000 * ( end.tv_sec - start.tv_sec ) + end.tv_usec -
start.tv_usec;
    //输出计算结果
    show_output(output);

    //计算输出卷积时间
   printf("Costing time=%fs\n",timeuse /1000000.0);
   printf("-------------------------------------------------------------------
\n");
    }


    }
```

3. 运行结果展示

(1)编译运行

```
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpicc -g -o ex2.o ex2.c
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$ mpiexec -n 4 ./ex2.o
```

(2)输入和卷积核

为方便我们观察计算结果是否正确，我们这里的input和kernel的所有位置都是1，padding填充的为0

input:

```
The size of input is 3*5*5
The size of kernel is 3*3*3*3
The input:
The 1 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
The 2 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
The 3 depth:
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
1 1 1 1 1
--------------------
```

kernel:

```
The kernel:
NUM:1  :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
NUM:2  :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
NUM:3  :
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
----------------------
```

(3)stride=1时的output和所用时间:

```
----------------------
The stride is 1,the padding is 1
The size of output is 3*5*5
The output:
The 1 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
The 2 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
The 3 depth:
12 18 18 18 12
18 27 27 27 18
18 27 27 27 18
18 27 27 27 18
12 18 18 18 12
----------------------
Costing time=0.000117s
```

(4)stride=2时的output和所用时间:

```
--------------------------------------------------------------------------
The stride is 2,the padding is 3
The size of output is 3*5*5
The output:
The 1 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
The 2 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
The 3 depth:
0 0 0 0 0
0 12 18 12 0
0 18 27 18 0
0 12 18 12 0
0 0 0 0 0
-----------------------
Costing time=0.000009s
```

(5)stride=3时的output和所用时间:

```
--------------------------------------------------------------------------
The stride is 3,the padding is 5
The size of output is 3*5*5
The output:
The 1 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
The 2 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
The 3 depth:
0 0 0 0 0
0 3 9 3 0
0 9 27 9 0
0 3 9 3 0
0 0 0 0 0
----------------------
Costing time=0.000442s
--------------------------------------------------------------------------
ljj@ljj-virtual-machine:~/Desktop/parallel_programming$
```

通过计算可以得出代码的运行结果的输出是正确的

并行imcol2计算不同尺寸input下运行时间(4个进程)

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.001947s	0.000642s	0.000584s
64	0.001623s	0.000918s	0.000845s
128	0.004335s	0.003720s	0.005614s
256	0.019460s	0.016976s	0.024739s
512	0.067742s	0.101204s	0.088155s
1024	0.288917s	0.436536s	0.275087s
2048	1.172413s	0.838276s	0.625258s

4. 实验结果分析

我们对串行计算、滑窗法和imcol2法三种方法计算进行分析

串行计算

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.001407s	0.000747s	0.000556s
64	0.005950s	0.002744s	0.002059s
128	0.026009s	0.012124s	0.011140s
256	0.115134s	0.060120s	0.042222s
512	0.392373s	0.235712s	0.181902s
1024	1.599387s	0.891970s	0.658294s
2048	6.550370s	2.043790s	1.620010s

滑窗法

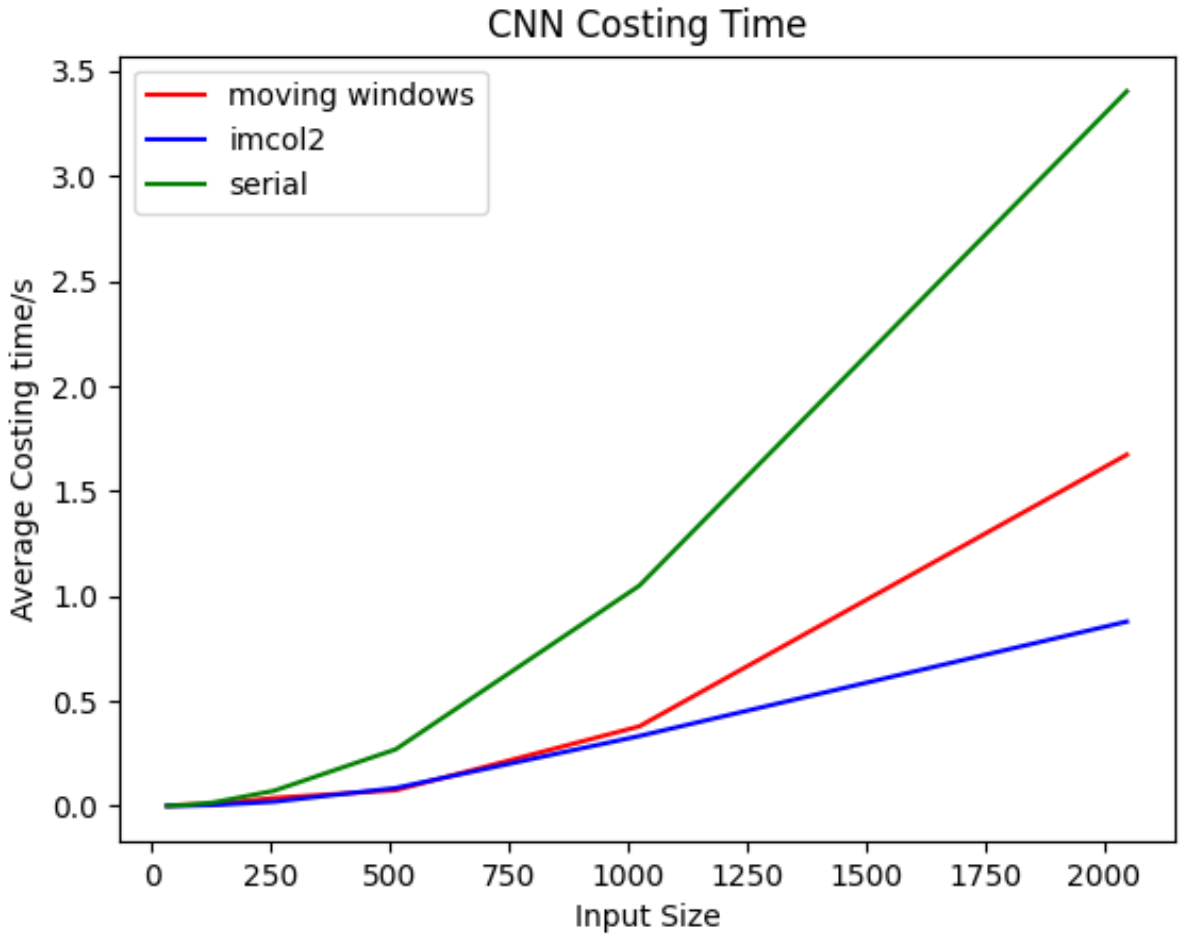
input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.000462s	0.000380s	0.000369s
64	0.014010s	0.002061s	0.001964s
128	0.007777s	0.009194s	0.008088s
256	0.032521s	0.046304s	0.036393s
512	0.0150963s	0.110651s	0.099870s
1024	0.459417s	0.34574s	0.333663s
2048	1.774545s	1.8033864s	1.442759s

imcol2

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
32	0.001947s	0.000642s	0.000584s
64	0.001623s	0.000918s	0.000845s
128	0.004335s	0.003720s	0.005614s
256	0.019460s	0.016976s	0.024739s
512	0.067742s	0.101204s	0.088155s
1024	0.288917s	0.436536s	0.275087s

input尺寸	stride=1 计算时间	stride=2 计算时间	stride=3 计算时间
2048	1.172413s	0.838276s	0.625258s

根据表格数据绘制图像：



分析

(1)可以看到串行计算和并行计算随着输入尺寸的加大，运行的时间的也越久。

(2)在输入尺寸较小时，串行计算时略快于并行计算的，这是因为并行计算中进程之间的通信和多个线程和进程的创建都消耗了时间。但随着输入尺寸的增大，并行计算的效果体现出来了，随着输入尺寸的增大并行计算越来越快于串行计算，这是因为并行计算将大量的任务分给多个进程同时进行，在工作量大时，大大提升了运算效率。

(3)并行计算中，从图中可以看出imcol2算法明显优越于滑窗法，这是因为imcol2算法充分利用了数据访问的局部性，从内存中读入一块数据到cache中时，这一块数据在imcol2算法下都是时间是邻近访问的，这就大大减少了cache的缺页率和对内存的访问时间，从而加速了运算时间。