高性能计算程序设计基础 秋季2021

任务1：

> 通过*CUDA*实现通用矩阵乘法（*Lab1*）的并行版本，*CUDA Thread Block size*从*32*增加至*512*，矩阵规模从*512*增加至*8192*。
>
> 通用矩阵乘法（*GEMM*）通常定义为：
>
> $C = A * B$
>
> $C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$
>
> 输入：*M*, *N*, *K*三个整数（*512 ~8192*）
>
> 问题描述：随机生成*M***N*和*N***K*的两个矩阵*A*,*B*,对这两个矩阵做乘法得到矩阵*C*。
>
> 输出：*A*,*B*,*C*三个矩阵以及矩阵计算的时间

代码

- 一个用于输出矩阵的主机上的函数

```
__host__ void PRINT(double *A, int x, int y){
    for(int i=0;i<x;++i){
        for(int j=0;j<y;++j) cout<<A[i*y+j]<<"  ";
        cout<<endl;
    }
}
```

- kernel函数，用于计算矩阵相乘，使用GEMM：

  n是计算现在是第几个线程，然后根据这个值计算得到算的是第几行第几列的输出结果。

```
__global__ void GEMM(double *A, double *B, double *C,int M, int N, int K){
    int n = blockIdx.x*blockDim.x + threadIdx.x;
    int i = n/N;
    int j = n%N;
    C[i*N+j] = 0;
    for(int k=0; k<K; ++k)
        C[i*N+j] += A[i*K+k]*B[j+k*N];
}
```

- 下面都是主函数中的内容：
- 初始化，获取两个矩阵的规模大小（A是$M*K$，N是$K*N$），并开始计时：

```
    int M, K, N, thread_block_size;
    double *A,*B,*C;
    thread_block_size = atoi(argv[1]);
    M = atoi(argv[2]);
    K = atoi(argv[3]);
    N = atoi(argv[4]);
    cout<<"the size of matrix: M = "<<M<<", K = "<<K<<", N = "<<N<<endl;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);//在GPU中分配显存也算时间
```

- 分配内存空间，并给矩阵赋值随机变量：

  这里使用函数`cudaMallocManaged()`来分配空间，这个函数可以同意内存管理，就是申请一次就好，不用先在CPU中申请后在GPU上申请，再拷贝来拷贝去。但是，把Host数据往Device迁移的操作，可以提升性能.

  `cudaDeviceSynchronize();`之前也说过了，主要是由于核函数是异步启动，需要CPU等待GPU代码执行完成.

```
    cudaMallocManaged((void**)&A, M*K*sizeof(double));
    cudaMallocManaged((void**)&B, K*N*sizeof(double));
    cudaMallocManaged((void**)&C, M*N*sizeof(double));

    for(int i=0; i<M; ++i)
        for(int j=0; j<K; ++j) A[i*K+j] = rand()%100;
    for(int i=0; i<K; ++i)
        for(int j=0; j<N; ++j) B[i*N+j] = rand()%100;
```

- 调用kernel函数进行计算矩阵乘法

  `cudaDeviceSynchronize();`用于同步，主要是由于核函数是异步启动，需要CPU等待GPU代码执行完成。

```
    dim3 blockSize(thread_block_size);
    dim3 gridSize((M*N)/thread_block_size);

    GEMM<<<gridSize, blockSize>>>(A, B, C, M, N, K);

    cudaDeviceSynchronize();//设备同步
```

- 计算运行时间、输出计算结果、释放内存空间：

```
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elapsedTime=0;
    cudaEventElapsedTime(&elapsedTime, start, stop);

    cout<<"所用时间为: "<< elapsedTime<<" <ms>"<<endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
```

```
9
10      // cout<<"A = "<<endl;
11      // PRINT(A, M, K);
12      // cout<<"B = "<<endl;
13      // PRINT(B, K, N);
14      // cout<<"C = "<<endl;
15      // PRINT(C, M, N);
16
17      cudaFree(A);
18      cudaFree(B);
19      cudaFree(C);
```

编译以及运行结果

- 编译

```
1  $ nvcc -o 1 1.cu
```

- 运行，thread_block_size是每个block上分配的线程数，M是矩阵A、C的行数，K是矩阵A的列数、矩阵B的行数，N是矩阵B、C的列数。

```
1  $ ./1 <thread_block_size> <M> <K> <N>
```

- 验证计算结果：

```
the size of matrix: M = 5, K = 5, N = 5
所用时间为: 0.521216 <ms>
A =
83  86  77  15  93
35  86  92  49  21
62  27  90  59  63
26  40  26  72  36
11  68  67  29  82
B =
30  62  23  67  35
29  2  22  58  69
67  93  56  11  42
29  73  21  19  84
37  98  24  15  70
C =
14019  22688  10660  13076  19843
11906  16533  9382  9591  16609
12715  22749  9811  8776  17179
7102  12894  5310  6256  13330
10666  17202  8078  7199  16067
```

通过其他方式计算，验证这个结果是正确的。

- CUDA Thread Block size从32增加至512，矩阵规模从512增加至8192的运行时间：

```
jovyan@jupyter-lab6use:~$ nvcc -o 1 1.cu
jovyan@jupyter-lab6use:~$ ./1 32 512 512 512
the size of matrix: M = 512, K = 512, N = 512
所用时间为: 16.6339 <ms>
jovyan@jupyter-lab6use:~$ ./1 64 1024 1024 1024
the size of matrix: M = 1024, K = 1024, N = 1024
所用时间为: 66.3982 <ms>
jovyan@jupyter-lab6use:~$ ./1 128 2048 2048 2048
the size of matrix: M = 2048, K = 2048, N = 2048
所用时间为: 353.147 <ms>
jovyan@jupyter-lab6use:~$ ./1 256 4096 4096 4096
the size of matrix: M = 4096, K = 4096, N = 4096
所用时间为: 1741.74 <ms>
jovyan@jupyter-lab6use:~$ ./1 512 8192 8192 8192
the size of matrix: M = 8192, K = 8192, N = 8192
所用时间为: 13996.2 <ms>
```

任务2：

> 通过*NVDIA*的矩阵计算函数库*CUBLAS*计算矩阵相乘，矩阵规模从*512*增加至*8192*，并与任务*1*和任务*2*的矩阵乘法进行性能比较和分析，如果性能不如*CUBLAS*，思考并文字描述可能的改进方法（参考《计算机体系结构-量化研究方法》第四章）。

> *CUBLAS*参考资料《*CUBLAS_Library.pdf*》，*CUBLAS*矩阵乘法参考第70页内容。
>
> *CUBLAS*矩阵乘法例子，参考附件《*matrixMulCUBLAS*》

代码

- 头文件

```
#include <iostream>
#include <cstdlib>
#include <cuda_runtime.h>
#include "cublas_v2.h"
#include <device_launch_parameters.h>
```

- 主机上有一个用于输出矩阵的函数，和任务1中的完全一样，不再赘述。
- 主函数：
- 初始化、分配空间、赋值，并开始计时：

```
    int M, K, N;
    double *A,*B,*C;
    M = atoi(argv[1]);
    K = atoi(argv[2]);
    N = atoi(argv[3]);
    cout<<"the size of matrix: M = "<<M<<", K = "<<K<<", N = "<<N<<endl;

    cudaMallocManaged((void**)&A, M*K*sizeof(double));
    cudaMallocManaged((void**)&B, K*N*sizeof(double));
    cudaMallocManaged((void**)&C, M*N*sizeof(double));

    for(int i=0; i<M; ++i)
        for(int j=0; j<K; ++j) A[i*K+j] = rand()%100;
    for(int i=0; i<K; ++i)
        for(int j=0; j<N; ++j) B[i*N+j] = rand()%100;

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
```

- 调用NVDIA的矩阵计算函数库CUBLAS计算矩阵相乘：

  创建句柄，并验证是否被正确创建，然后调用计算double类型矩阵乘法的库函数`cublasDgemm()`，参数含义都有介绍这里不再赘述。

```
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    if (status != CUBLAS_STATUS_SUCCESS)
    {
          if (status == CUBLAS_STATUS_NOT_INITIALIZED) {
             cout << "CUBLAS 对象实例化出错" << endl;
          }
          getchar ();
          return EXIT_FAILURE;
    }

    double a = 1, b = 0;
    cublasDgemm (
            handle,     // blas 库对象
            CUBLAS_OP_T,     // 矩阵 A 属性参数
            CUBLAS_OP_T,     // 矩阵 B 属性参数
```

```
            M,     // A, C 的行数
            N,     // B, C 的列数
            K,     // A 的列数和 B 的行数
            &a,     // 运算式的 α 值
            A,     // A 在显存中的地址
            K,     // lda
            B,     // B 在显存中的地址
            N,     // ldb
            &b,     // 运算式的 β 值
            C,     // C 在显存中的地址(结果矩阵)
            M    // ldc
        );
```

- 最后，计算运行时间，输出计算结果并释放内存空间。

```
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elapsedTime=0;
    cudaEventElapsedTime(&elapsedTime, start, stop);

    cout<<"所用时间为: "<< elapsedTime<<" <ms>"<<endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    // cout<<"A = "<<endl;
    // PRINT(A, M, K);
    // cout<<"B = "<<endl;
    // PRINT(B, K, N);
    // cout<<"C = "<<endl;
    // PRINT(C, M, N);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
```

编译以及运行结果

- 编译，需要动态链接库

```
$ nvcc -o 2 2.cu -lcublas
```

- 运行，A=M*K，B=K*N，C=M*N

```
$ ./2 <M> <K> <N>
```

- 验证计算结果

```
the size of matrix: M = 5, K = 5, N = 5
所用时间为: 603.086 <ms>
A =
83  86  77  15  93
35  86  92  49  21
62  27  90  59  63
26  40  26  72  36
11  68  67  29  82
B =
30  62  23  67  35
29  2  22  58  69
67  93  56  11  42
29  73  21  19  84
37  98  24  15  70
C =
14019  11906  12715  7102  10666
22688  16533  22749  12894  17202
10660  9382  9811  5310  8078
13076  9591  8776  6256  7199
19843  16609  17179  13330_ 16067
```

通过其他方式计算，验证这个结果是正确的。

但是注意计算时cublas变成了行优先存储，主维就变成了矩阵列数，而C并没有转置，仍为列优先存储，主维为矩阵行数，所以最后拷贝出来时矩阵为结果的转置，这里的输出结果C是实际结果的转置。

- 矩阵规模从512增加至8192：

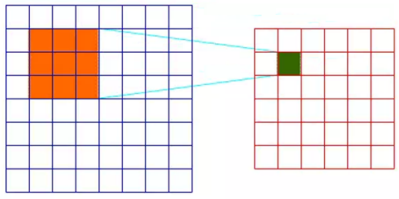
```
jovyan@jupyter-lab6use:~$ nvcc -o 2 2.cu -lcublas
jovyan@jupyter-lab6use:~$ ./2 512 512 512
the size of matrix: M = 512, K = 512, N = 512
所用时间为: 587.922 <ms>
jovyan@jupyter-lab6use:~$ ./2 1024 1024 1024
the size of matrix: M = 1024, K = 1024, N = 1024
所用时间为: 559.729 <ms>
jovyan@jupyter-lab6use:~$ ./2 2048 2048 2048
the size of matrix: M = 2048, K = 2048, N = 2048
所用时间为: 579.134 <ms>
jovyan@jupyter-lab6use:~$ ./2 4096 4096 4096
the size of matrix: M = 4096, K = 4096, N = 4096
所用时间为: 684.207 <ms>
jovyan@jupyter-lab6use:~$ ./2 8192 8192 8192
the size of matrix: M = 8192, K = 8192, N = 8192
所用时间为: 1426.14 <ms>
```

与任务1的运行时间进行对比：

N	GEMM（ms）	cublas（ms）
512	16.6339	587.92
1024	66.3982	559.729
2048	353.147	579.134
4096	1741.74	684.207
8192	13996.2	1426.14

与任务1相比，要虽然在规模为512、1024、2048的时候GEMM的运行时间要比使用cublas库的运行时间少，但是当矩阵规模为4096和8192的时候cublas库的运行效果要好多，尤其规模为8192的时候，运行时间要比GEMM的运行时间（13996.2ms）少非常多。想到之前说分配空间使用多步复制传送的性能要比统一内存管理的性能好一些，可以在这里进行改进，还有就是block和grid的分配也能进行优化。

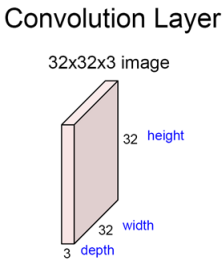
任务3：

> 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*GPU*上实现卷积操作，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*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*，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

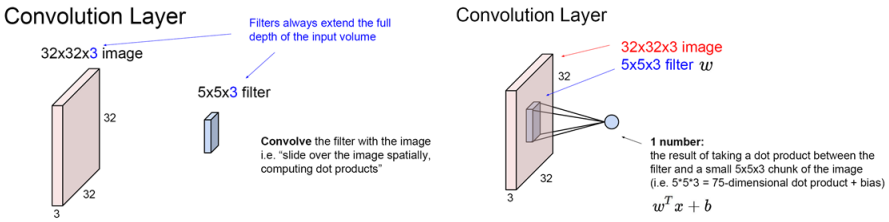
以下是部分*CNN*操作的解释*ppt*，具体参考附件中的人工智能课件。

*1）CNN Input Image*举例

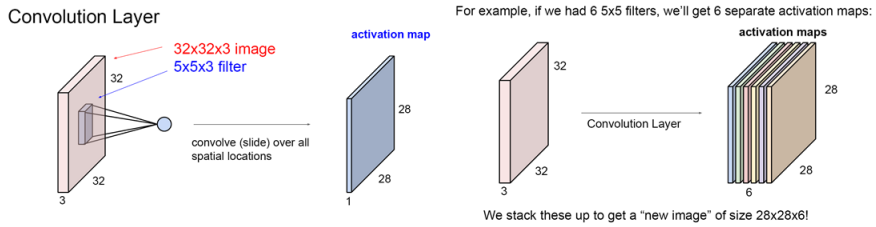
(Input)



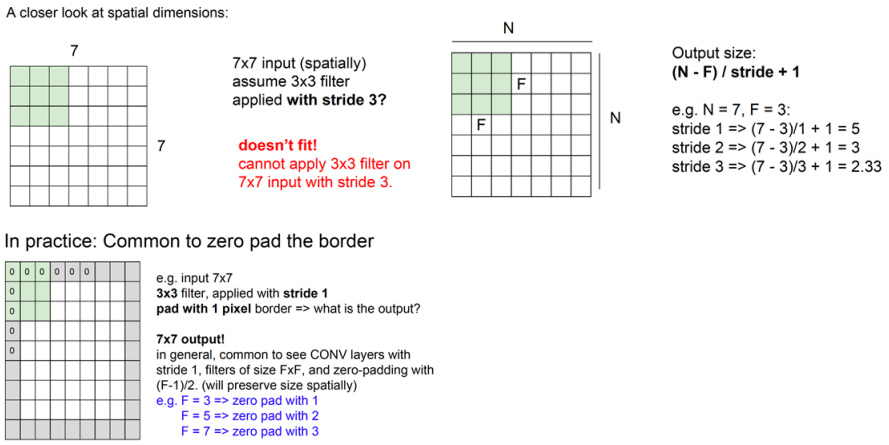
2）CNN Input Image 和 *Kernel(Filter)*



3)CNN 操作过程



*4）CNN*步幅*stride*和填充*padding*



代码

- 主机上用于输出的函数，输出宽高相同的多通矩阵：

```
__host__ void PRINT(double **matrix, int dim, int channel = 1){
    for(int i=0; i<channel; ++i){
        cout<<"when channel = "<< i << endl;
        for(int a=0; a<dim; ++a){
            for(int b=0; b<dim; ++b){
                cout<<matrix[i][a*dim + b]<<" ";
            }
            cout<<endl;
        }
        cout<<endl;
    }
}
```

- 主机上用于给三通道的矩阵分配空间的函数，一旦分配空间失败有异常就输出：

```
__host__ void CUDA_malloc(double **matrix, int N){
    cudaError_t temp;
    for(int i=0; i<3; ++i){
        temp = cudaMallocManaged(&matrix[i], N*N*sizeof(double));
        if(temp != 0) cout<<"ERROR! "<<temp<<endl;
    }
}
```

- 主机上用于给三通道矩阵赋值[a,b]范围的随机数。

```
__host__ void get_rand(double **matrix, int N, int a, int b){
    srand(clock());
    for(int i=0; i<3; ++i){
        for(int j=0; j<N; ++j){
            matrix[i][j] = rand()%(b-a+1) + a;
        }
    }
}
```

- 主机上用于给三通道矩阵填充的函数，根据本次作业的需要padding指的是需要扩充的列/行数，而不是圈数：

```
__host__ void pad(double **matrix, int dim, double **matrix2, int padding){
    //matrix是需要被扩充的矩阵，dim是它的维数，matrix2是扩充之后的矩阵，padding是需要扩充的列/行数
    CUDA_malloc(matrix2, dim+padding);
    for(int x=0; x<3; ++x){//3个channnel
        for(int i=0; i<dim+padding; ++i){
            for(int j=0; j<dim+padding; ++j){
                if(i == 0 || j == 0){
                    matrix2[x][i*(dim+padding)+j] = 0;
                }
                else{
                    matrix2[x][i*(dim+padding)+j] = matrix[x][(i-1)*dim + (j-1)];
                }
                if(padding == 2){
                    //如果只扩充1行/列，就只在第一行和第一列填充0，最后一行/列就不用填充了，
                    //由于本次的kernel是3*3的，扩充的行/列数只能是1/2。
                    if(i==dim+padding-1 || j==dim+padding-1){
                        matrix2[x][i*(dim+padding)+j] = 0;
                    }
                }
            }
        }
    }

}
```

- 最关键的kernel函数：

  每个通道的input和每个通道的kernel对应相乘，对应的卷积结果需要相加，最后就得到了结果。

  这里之前写的时候是输入的二维指针来表示input和kerel而不是像现在这样分开三个channel来写，但是这样写遇到了问题，代码总是报错，猜测在设备上用二维指针不太行，在主机上同样的代码用二维指针就能得到正确的结果，这里废了很多时间，最后改为一维指针传入就能够正确运算了。

```
__global__ void CNN(double *image1, double *image2, double *image3, int image_dim, \
    double *kernel1_, double *kernel2_, double *kernel3_, int kernel_dim, \
    int stride, double *map, int map_dim){
    //这里是三个通道的input矩阵，以及每个矩阵的维数
    //kernel的三个通道，以及kernel的维数，默认是3
    //移动的步数stride，以及输出结果的指针，和输出结果的维数

    int n = blockIdx.x*blockDim.x + threadIdx.x;
    int i = n/map_dim;
    int j = n%map_dim;//卷出来的是第i行j列的元素
    map[i*map_dim + j] = 0;

    for(int a=0; a<kernel_dim; ++a){
        for(int b=0; b<kernel_dim; ++b){
            map[i*map_dim + j] += image1[(i*stride+a)*image_dim+j*stride+b] *
kernel1_[a*3+b];
        }
    }

    for(int a=0; a<kernel_dim; ++a){
        for(int b=0; b<kernel_dim; ++b){
            map[i*map_dim + j] += image2[(i*stride+a)*image_dim+j*stride+b] *
kernel2_[a*3+b];
        }
    }

    for(int a=0; a<kernel_dim; ++a){
        for(int b=0; b<kernel_dim; ++b){
            map[i*map_dim + j] += image3[(i*stride+a)*image_dim+j*stride+b] *
kernel3_[a*3+b];
        }
    }

}
```

- 下面的代码都是主函数上的内容：
- 初始化：

```
    int N, thread_block_size;
    N = atoi(argv[1]);//the highth or width of three channels picture
    thread_block_size = atoi(argv[2]);//每个block用到的线程数

    double *image[3];
    double *kernel_1[3];
    double *kernel_2[3];
    double *kernel_3[3];
    double *map1[3]; //stride = 1时，3种kernel的卷积结果
    double *map2[3]; //stride = 2时，3种kernel的卷积结果
    double *map3[3]; //stride = 3时，3种kernel的卷积结果
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);//开始计时

```

```
    //分配空间给image和kernel
    CUDA_malloc(image, N);
    CUDA_malloc(kernel_1, 3);
    CUDA_malloc(kernel_2, 3);
    CUDA_malloc(kernel_3, 3);

    //获取某个范围[a,b]内的随机整数
    get_rand(image, N*N, 0, 50);
    get_rand(kernel_1, 3*3, -1, 1);
    get_rand(kernel_2, 3*3, -1, 1);
    get_rand(kernel_3, 3*3, -1, 1);
```

- stride = 1时进行卷积

```
    //stride=1
    int output_size[3];
    output_size[0] = (N-3)/1 + 1;
    CUDA_malloc(map1, output_size[0]);//先计算得到output size
    dim3 blockSize(thread_block_size);
    dim3 gridSize((output_size[0]*output_size[0])/thread_block_size);
    //然后根据这个值设置blockSize、gridSize

    //调用kernel函数计算三个kernel_的卷积结果,stride = 1
    CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_1[0], kernel_1[1], kernel_1[2], 3, 1, map1[0], output_size[0]);
    CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_2[0], kernel_2[1], kernel_2[2], 3, 1, map1[1], output_size[0]);
    CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_3[0], kernel_3[1], kernel_3[2], 3, 1, map1[2], output_size[0]);
```

- stride = 2时进行卷积

```
    //判断stride=2的时候是否需要填充
    int padding = (N-3)%2 ;
    if (padding){//需要填充
        double *image_new[3];
        pad( image, N, image_new, padding);//扩展成一个新矩阵
        output_size[1] = (N+padding-3)/2 + 1;
        CUDA_malloc(map2, output_size[1]);//计算得到卷积结果的大小
        blockSize = thread_block_size;
        gridSize = (output_size[1]*output_size[1])/thread_block_size;//根据卷积结果的大小设置这两个值

        //调用kernel函数计算三个kernel_的卷积结果,stride = 2
        CNN<<<gridSize, blockSize>>>(image_new[0], image_new[1], image_new[2], N+padding, kernel_1[0], kernel_1[1], kernel_1[2], 3, 2, map2[0], output_size[1]);
        CNN<<<gridSize, blockSize>>>(image_new[0], image_new[1], image_new[2], N+padding, kernel_2[0], kernel_2[1], kernel_2[2], 3, 2, map2[1], output_size[1]);
        CNN<<<gridSize, blockSize>>>(image_new[0], image_new[1], image_new[2], N+padding, kernel_3[0], kernel_3[1], kernel_3[2], 3, 2, map2[2], output_size[1]);
        // cout<<"After expanding the original image, the new image, stride = "<<2<<", padding ="<<padding<<endl;
        // PRINT(image_new,N+padding,3);
        for(int i=0;i<3;++i){
            cudaFree(image_new[i]);
        }
    }
    else{//不需要填充
        output_size[1] = (N-3)/2 + 1;
        CUDA_malloc(map2, output_size[1]);
        blockSize = thread_block_size;
        gridSize = (output_size[1]*output_size[1])/thread_block_size;
```

```
26
27          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_1[0],
    kernel_1[1], kernel_1[2], 3, 2, map2[0], output_size[1]);
28          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_2[0],
    kernel_2[1], kernel_2[2], 3, 2, map2[1], output_size[1]);
29          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_3[0],
    kernel_3[1], kernel_3[2], 3, 2, map2[2], output_size[1]);
30      }
```

- stride = 3时进行卷积，原理和上面stride=2的一样：

```
1       //判断stride=3的时候是否需要填充
2       padding = (3 - (N-3)%3)%3;//如果除余的结果是1，就需要填充"一圈",增加两个size；如果是2，说明
    需要填充一维（）这里随机加在最上面和最左边
3       if (padding){
4           double *image_new2[3];
5           pad( image, N, image_new2, padding);
6           output_size[2] = (N+padding-3)/3 + 1;
7           CUDA_malloc(map3, output_size[2]);
8           blockSize = thread_block_size;
9           gridSize = (output_size[2]*output_size[2])/thread_block_size;
10          CNN<<<gridSize, blockSize>>>(image_new2[0], image_new2[1], image_new2[2],
    N+padding, kernel_1[0], kernel_1[1], kernel_1[2], 3, 3, map3[0], output_size[2]);
11          CNN<<<gridSize, blockSize>>>(image_new2[0], image_new2[1], image_new2[2],
    N+padding, kernel_2[0], kernel_2[1], kernel_2[2], 3, 3, map3[1], output_size[2]);
12          CNN<<<gridSize, blockSize>>>(image_new2[0], image_new2[1], image_new2[2],
    N+padding, kernel_3[0], kernel_3[1], kernel_3[2], 3, 3, map3[2], output_size[2]);
13          // cout<<"After expanding the original image, the new image, stride = "<<3<<"
    , padding ="<<padding<<endl;
14          // PRINT(image_new2,N+padding,3);
15          for(int i=0;i<3;++i){
16              cudaFree(image_new2[i]);
17          }
18      }
19      else{
20          output_size[2] = (N-3)/3 + 1;
21          CUDA_malloc(map3, output_size[2]);
22          blockSize = thread_block_size;
23          gridSize = (output_size[2]*output_size[2])/thread_block_size;
24          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_1[0],
    kernel_1[1], kernel_1[2], 3, 3, map3[0], output_size[2]);
25          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_2[0],
    kernel_2[1], kernel_2[2], 3, 3, map3[1], output_size[2]);
26          CNN<<<gridSize, blockSize>>>(image[0], image[1], image[2], N, kernel_3[0],
    kernel_3[1], kernel_3[2], 3, 3, map3[2], output_size[2]);
27      }
```

- 最后进行同步、输出运行时间、结果以及释放空间：

```
1       cudaDeviceSynchronize();
2
3       cudaEventRecord(stop, 0);
4       cudaEventSynchronize(stop);
5       float elapsedTime = 0;
6       cudaEventElapsedTime(&elapsedTime, start, stop);
7       cout<<"所用时间为: "<< elapsedTime<<" <ms>"<<endl;
8       cudaEventDestroy(start);
9       cudaEventDestroy(stop);
10
11
12      // cout<<"image: "<<endl;
13      // PRINT(image, N, 3);
```

```
    // cout<<"kernel:"<<endl;
    // cout<<"kernel_1: "<<endl;
    // PRINT(kernel_1, 3, 3);
    // cout<<"kernel_2: "<<endl;
    // PRINT(kernel_2, 3, 3);
    // cout<<"kernel_3: "<<endl;
    // PRINT(kernel_3, 3, 3);
    // cout<<"answer:"<<endl;
    // cout<<"when stride = 1, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map1, output_size[0], 3);
    // cout<<"when stride = 2, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map2, output_size[1], 3);
    // cout<<"when stride = 3, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map3, output_size[2], 3);


    for(int i=0; i<3; ++i){
        cudaFree(image[i]);
        cudaFree(kernel_1[i]);
        cudaFree(kernel_2[i]);
        cudaFree(kernel_3[i]);
        cudaFree(map1[i]);
        cudaFree(map2[i]);
        cudaFree(map3[i]);
    }
```

编译以及运行结果

- 编译指令

```
$ nvcc -o 3 3.cu
```

- 运行指令，<N>是input的矩阵每个channel的规模大小，<thread_block_size>是每个block用到的线程数。

  注意：编译的时候$thread_block_size < N$，否则计算结果不正确。

```
$ ./3 <N> <thread_block_size>
```

- 验证填充的结果：

  比如input的大小为7时，在stride = 3时需要填充一圈0，填充结果：

```
jovyan@jupyter-lab6use: $ nvcc -o 3 3.cu
jovyan@jupyter-lab6use:~$ ./3 7
After expanding the original image, the new image, stride = 3 , padding =2
when channel = 0
0 0 0 0 0 0 0 0 0
0 0 30 16 9 41 9 38 0
0 4 5 48 5 18 24 9 0
0 45 10 41 3 14 34 22 0
0 32 6 23 19 32 35 40 0
0 26 27 29 27 31 46 36 0
0 22 29 48 0 8 45 5 0
0 27 19 14 46 29 29 49 0
0 0 0 0 0 0 0 0 0

when channel = 1
0 0 0 0 0 0 0 0 0
0 17 37 45 23 43 18 42 0
0 49 2 32 25 3 10 26 0
0 35 30 11 31 33 33 31 0
0 42 2 10 43 21 49 38 0
0 24 1 10 41 38 5 13 0
0 5 48 30 28 24 11 2 0
0 27 46 28 36 0 13 41 0
0 0 0 0 0 0 0 0 0

when channel = 2
0 0 0 0 0 0 0 0 0
0 33 21 21 24 23 5 16 0
0 18 3 28 42 4 39 6 0
0 17 18 44 22 15 23 50 0
0 39 8 27 40 4 29 0 0
0 4 43 41 11 38 37 36 0
0 10 16 26 2 20 29 44 0
0 49 17 24 15 35 17 37 0
0 0 0 0 0 0 0 0 0
```

  未填充时的input：

```
image:
when channel = 0
0 30 16 9 41 9 38
4 5 48 5 18 24 9
45 10 41 3 14 34 22
32 6 23 19 32 35 40
26 27 29 27 31 46 36
22 29 48 0 8 45 5
27 19 14 46 29 29 49

when channel = 1
17 37 45 23 43 18 42
49 2 32 25 3 10 26
35 30 11 31 33 33 31
42 2 10 43 21 49 38
24 1 10 41 38 5 13
5 48 30 28 24 11 2
27 46 28 36 0 13 41

when channel = 2
33 21 21 24 23 5 16
18 3 28 42 4 39 6
17 18 44 22 15 23 50
39 8 27 40 4 29 0
4 43 41 11 38 37 36
10 16 26 2 20 29 44
49 17 24 15 35 17 37
```

- 3个kernel：

```
kernel:
kernel_1:
when channel = 0
-1 -1 0
-1 1 -1
1 0 1

when channel = 1
-1 1 -1
-1 -1 -1
0 1 -1

when channel = 2
1 0 0
1 -1 1
0 1 1

kernel_2:
when channel = 0
-1 -1 0
-1 1 -1
1 0 1

when channel = 1
-1 1 -1
-1 -1 -1
0 1 -1

when channel = 2
1 0 0
1 -1 1
0 1 1

 kernel_3:
 when channel = 0
 -1 -1 0
 -1 1 -1
 1 0 1

 when channel = 1
 -1 1 -1
 -1 -1 -1
 0 1 -1

 when channel = 2
 1 0 0
 1 -1 1
 0 1 1
```

- stride=1、2、3的计算结果，（input和kernel是上面的值）：

```
answer:
when stride = 1, keren11/ kernel2/ kernel, the answer :
when channel = 0
58 15 -110 79 -54
-97 -34 -4 -44 -2
31 -44 -20 85 -102
20 -61 25 -91 -18
-86 -44 13 -30 49

when channel = 1
58 15 -110 79 -54
-97 -34 -4 -44 -2
31 -44 -20 85 -102
20 -61 25 -91 -18
-86 -44 13 -30 49

when channel = 2
58 15 -110 79 -54
-97 -34 -4 -44 -2
31 -44 -20 85 -102
20 -61 25 -91 -18
-86 -44 13 -30 49
```

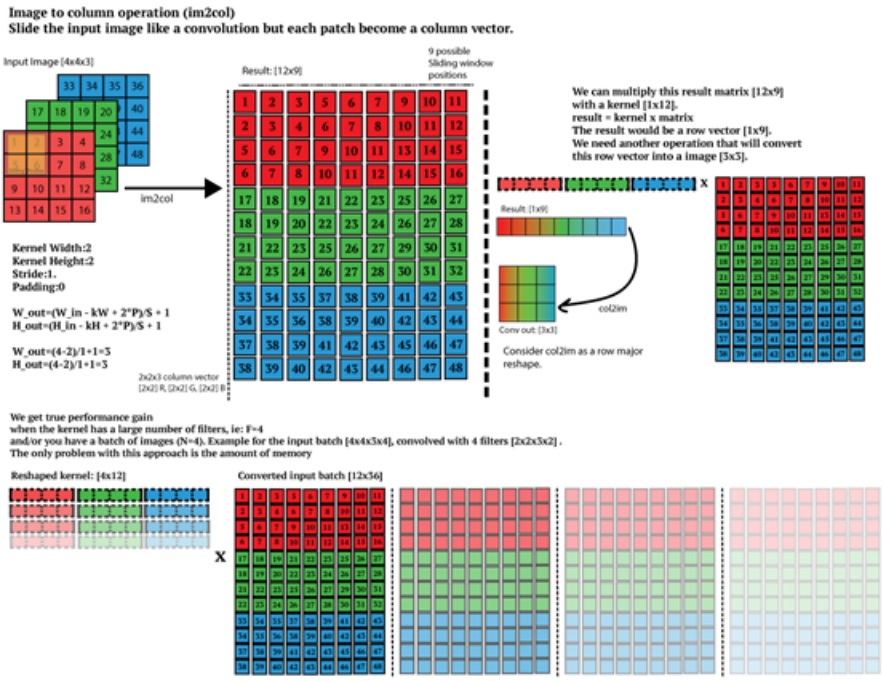
```
when stride = 2, keren11/ kernel2/ kernel, the answer :
when channel = 0
58 -110 -54
31 -20 -102
-86 13 49

when channel = 1
58 -110 -54
31 -20 -102
-86 13 49

when channel = 2
58 -110 -54
31 -20 -102
-86 13 49

when stride = 3, keren11/ kernel2/ kernel, the answer :
when channel = 0
-23 -5 14
8 -20 7
-162 -65 -84

when channel = 1
-23 -5 14
8 -20 7
-162 -65 -84

when channel = 2
-23 -5 14
8 -20 7
-162 -65 -84
```

- 经过验算，上面的计算结果正确。
- 输入从128增加至2048（因为当N=4096会发生段错误，是因为N=4096要3个多G的显存放不下，计算了三种stride），thread_block_size从64增加到1024：

```
jovyan@jupyter-lab6use:~$ nvcc -o 3 3.cu
jovyan@jupyter-lab6use:~$ ./3 128 64
所用时间为: 3.99974 <ms>
jovyan@jupyter-lab6use:~$ ./3 256 128
所用时间为: 10.325 <ms>
jovyan@jupyter-lab6use:~$ ./3 512 256
所用时间为: 37.3187 <ms>
jovyan@jupyter-lab6use:~$ ./3 1024 512
所用时间为: 163.422 <ms>
jovyan@jupyter-lab6use:~$ ./3 2048 1024
所用时间为: 590.831 <ms>
```

任务4：

> 使用*im2col*方法结合任务*1*实现的*GEMM*（通用矩阵乘法）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*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。



原理见：[im2col方法实现卷积算法 - 知乎 (zhihu.com)](#)，这篇文章介绍得非常详细。

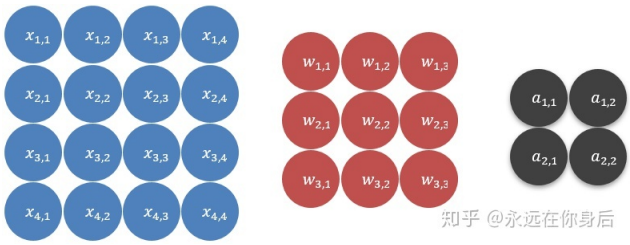
代码

- 主机上的一些函数：
  - 用于输出宽高相同的多通矩阵的函数`__host__ void PRINT(double **matrix, int dim, int channel = 1);`
  - 用于给三通道的矩阵分配空间的函数`__host__ void CUDA_malloc(double **matrix, int N);`
  - 用于给三通道矩阵赋值[a,b]范围的随机数的函数`__host__ void get_rand(double **matrix, int N, int a, int b);`
  - 以及用于给三通道矩阵填充0的函数`__host__ void pad(double **matrix, int dim, double **matrix2, int padding);`

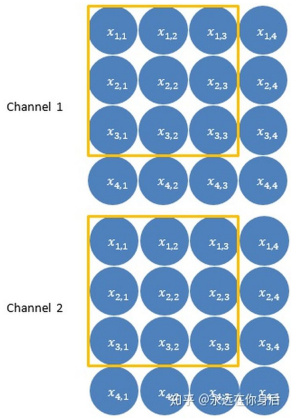
  这些函数和任务3中的同名函数完全相同，这里不再赘述，下面的一些主机函数时任务3中没有的。
- 两个主机上用于输出的函数，一个用于输出宽高相同的多通矩阵，一个用于输出不同宽高的矩阵：

```
__host__ void PRINT_single(double *matrix, int M, int N){
        for(int a=0; a<M; ++a){
            for(int b=0; b<N; ++b){
                cout<<matrix[a*N + b]<<" ";
            }
            cout<<endl;
        }
        cout<<endl;
}
```

- 转换，将按照im2col的原理将分割成的矩阵多通道依次排列：



将输入根据stride，划分成outputsize个子集，每个子集里有kernel_size*kernel_size个数据，具体的划分过程是，几个通道（下面的例子是2通道）各自划一个子集，与卷积核对应的通道大小相同：



将两个通道的子集上下拼接变成一个6行3列的矩阵，它们之间元素的顺序用下标表示：

$$x^{(1)} = \begin{bmatrix} x_{1,1,1} & x_{1,1,2} & x_{1,1,3} \\ x_{1,2,1} & x_{1,2,2} & x_{1,2,3} \\ x_{1,3,1} & x_{1,3,2} & x_{1,3,3} \\ x_{2,1,1} & x_{2,1,2} & x_{2,1,3} \\ x_{2,2,1} & x_{2,2,2} & x_{2,1,3} \\ x_{2,3,1} & x_{2,3,2} & x_{2,3,3} \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \\ x_7 & x_8 & x_9 \\ x_{10} & x_{11} & x_{12} \\ x_{13} & x_{14} & x_{15} \\ x_{16} & x_{17} & x_{18} \end{bmatrix}$$

然后同样的将其展开成一个横向量：$x^{(1)} = [x_1 \quad \cdots \quad x_{18}]$，剩下的就是重复上述两步，得到4个横向量，得到一个矩阵：

$$A = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{bmatrix}$$

按照上面的要求，将3通道的矩阵进行转换，最后成为上面的形式。

```
__host__ void convertx(double **image, int dim, int stride, double *matrix){
    //image是原来的3通道的矩阵，dim是每个通道的矩阵维数，stride是移动的步数，每个要被划分的矩阵的起始位置需要借助stride来确定，matrix是最后输出的转换结果。
    int col = 0;//表示此时是转换之后的第几行
    for(int i=0; i<dim-2; i=i+stride){
        for(int j=0; j<dim-2; j=j+stride){//i j表示这次划分从image的i行j列开始
            for(int x=0; x<3; ++x){//3是通道数
                for(int a=0; a<3; ++a){//3是kernel的维度
                    for(int b=0; b<3; ++b){//a b x控制matrix的列数
                        matrix[col*27 + (x*3+a)*3+b] = image[x][(i+a)*dim+(j+b)];
                    }
                }
            }
            ++col;
        }
    }
}
```

- 一个主机上的函数将卷积计算结果进行变形恢复：

  卷积结果：

$$A = XW = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{bmatrix} [W_1 \quad W_2] = \begin{bmatrix} a_1 & a_5 \\ a_2 & a_6 \\ a_3 & a_7 \\ a_4 & a_8 \end{bmatrix}$$

  从上式可以得知，各个每个卷积核是独立与输入进行卷积计算，相互之间并不影响；然后，将计算结果变形，就得到了输出：

$$A=\begin{bmatrix} a_1 & a_5 \\ a_2 & a_6 \\ a_3 & a_7 \\ a_4 & a_8 \end{bmatrix} \rightarrow \begin{bmatrix} \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \\ \begin{bmatrix} a_5 & a_6 \\ a_7 & a_8 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a_{1,1,1} & a_{1,1,2} \\ a_{1,2,1} & a_{1,2,2} \end{bmatrix} \\ \begin{bmatrix} a_{2,1,1} & a_{2,1,2} \\ a_{2,2,1} & a_{2,2,2} \end{bmatrix} \end{bmatrix}$$

```
__host__ void recover_answer(double *answer, double **answer_output, int channel, int a, int b){
    for(int i=0; i<channel; ++i){
        for(int j=0; j<a; ++j){
            for(int k=0;k<b;++k){
                answer_output[i][j*b+k] = answer[(j*b+k)*channel+i];
            }
        }
    }
}
```

- 用于计算矩阵乘法的kernel函数：

```
__global__ void GEMM(double *A, double *B, double *C,int M, int N, int K){
    int n = blockIdx.x*blockDim.x + threadIdx.x;
    int i = n/N;
    int j = n%N;
    C[i*N+j] = 0;
    for(int k=0; k<K; ++k)
        C[i*N+j] += A[i*K+k]*B[j+k*N];
}
```

- 从下面开始都是主函数中的内容。
- 进行一些初始化：

```
    int N,thread_block_size;
    N = atoi(argv[1]);//the highth or width of three channels picture
    thread_block_size = atoi(argv[2]);//每个block用到的线程数

    double *image[3];//输入的3 channel"图像"
    double *kernel_1[3];
    double *kernel_2[3];
    double *kernel_3[3];//3个kernel，每个kernel 3个channel，每个channel规模为3*3
    double *map1[3]; //stride = 1时，3种kernel的卷积结果
    double *map2[3]; //stride = 2时，3种kernel的卷积结果
    double *map3[3]; //stride = 3时，3种kernel的卷积结果
    //[3]：是3通道

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);//开始计时

    //分配空间给image和kernel
    CUDA_malloc(image, N);
    CUDA_malloc(kernel_1, 3);
    CUDA_malloc(kernel_2, 3);
    CUDA_malloc(kernel_3, 3);

    //获取某个范围[a,b]内的随机整数
    get_rand(image, N*N, 0, 50);
    get_rand(kernel_1, 3*3, -1, 1);
    get_rand(kernel_2, 3*3, -1, 1);
    get_rand(kernel_3, 3*3, -1, 1);
```

- 将3个kernel进行变形：

原理：

对于第一个卷积核，处理的方式也和之前一样，展开成一个列向量，不过这里命名为 W_1 ：

$$W_1 = \begin{bmatrix} w_1 \\ \vdots \\ w_{18} \end{bmatrix}$$

然后对于第二个卷积核也是采取同样的操作展开成一个列向量，为 W_2 ，所以此时所有的卷积核可以用一个矩阵表示：

$$W = [W_1 \quad W_2]$$

然后是结算卷积结果：

$$A = XW = \begin{bmatrix} x^{(1)} \\ x^{(2)} \\ x^{(3)} \\ x^{(4)} \end{bmatrix} [W_1 \quad W_2] = \begin{bmatrix} a_1 & a_5 \\ a_2 & a_6 \\ a_3 & a_7 \\ a_4 & a_8 \end{bmatrix}$$

三个卷积核也是同理，这里先利用之前转换成像X那样行展开的形式（利用之前写好的函数convertx），然后再进行转置，就能得到上面的W的形式了。

```
    //kernel convert
    double *kernel_;
    cudaMallocManaged(&kernel_, 27*3*sizeof(double));//3个kernel
    double *kernel_temp[3];
    cudaMallocManaged(&kernel_temp[0], 27*sizeof(double));
    cudaMallocManaged(&kernel_temp[1], 27*sizeof(double));
    cudaMallocManaged(&kernel_temp[2], 27*sizeof(double));
    convertx(kernel_1, 3, 1, kernel_temp[0]);
    convertx(kernel_2, 3, 1, kernel_temp[1]);
    convertx(kernel_3, 3, 1, kernel_temp[2]);
    //先将每个kernel重新整合成一行，再将三个kernel重新组合排列按照要求

    for(int j=0; j<3; ++j){
        for(int i=0; i<27; ++i){
            kernel_[i*3+j] = kernel_temp[j][i];
        }
    }//转置
    for(int i=0;i<3;++i){
        cudaFree(kernel_temp[i]);
    }
```

- 计算stride = 1时的卷积结果：

```
    int output_size[3];//用来表示三种stride的卷积结果的规模大小。

    //stride=1,进行卷积
    //计算outputsize，并给output分配空间，并根据outputsize设置blockSize，gridSize
    output_size[0] = (N-3)/1 + 1;
    CUDA_malloc(map1, output_size[0]);
    dim3 blockSize(thread_block_size);
    dim3 gridSize((output_size[0]*output_size[0]*3)/thread_block_size);

    //将input根据im2col进行转化，转化结果就是上面介绍的A
    double *matrix ;
    cudaMallocManaged(&matrix, output_size[0]*output_size[0]*27*sizeof(double));
    convertx(image, N, 1, matrix);
    // PRINT_single(matrix, output_size[0]*output_size[0],27);

    //给卷积结果分配空间，并调用GEMM进行计算，然后进行同步
    double *answer ;
    cudaMallocManaged(&answer, output_size[0]*output_size[0]*3*sizeof(double));
    GEMM<<<gridSize, blockSize>>>(matrix, kernel_, answer,
output_size[0]*output_size[0], 3, 27);
```

```
    cudaDeviceSynchronize();
    // cout<<"answer1, when stride = 1 :"<<endl;
    // PRINT_single(answer, output_size[0]*output_size[0], 3);

    recover_answer(answer, map1, 3, output_size[0], output_size[0]);//将结果转化恢复
```

- 计算stride = 2时的卷积结果：

```
    //判断stride=2的时候是否需要填充
    int padding = (N-3)%2 ;
    if (padding){//需要填充
        double *image_new[3];
        pad( image, N, image_new, padding);//扩展成一个“新矩阵”，用0填充
        output_size[1] = (N+padding-3)/2 + 1;
        CUDA_malloc(map2, output_size[1]);//计算得到卷积结果的大小

        //根据outputsize设置blockSize，gridSize
        blockSize = thread_block_size;
        gridSize = (output_size[1]*output_size[1]*3)/thread_block_size;

        //根据im2col转化扩充后的矩阵
        double *matrix2 ;//= NULL;
        cudaMallocManaged(&matrix2, output_size[1]*output_size[1]*27*sizeof(double));
        convertx(image_new, N+padding, 2, matrix2);

        //计算,和steide=1的原理相同。
        double *answer2 ;//= NULL;
        cudaMallocManaged(&answer2, \
                          output_size[1]*output_size[1]*3*sizeof(double));
        GEMM<<<gridSize, blockSize>>>(matrix, kernel_, answer2, \
                                      output_size[1]*output_size[1], 3, 27);
        cudaDeviceSynchronize();
        // cout<<"answer2, when stride = 2 :"<<endl;
        // PRINT_single(answer2, output_size[1]*output_size[1], 3);

        recover_answer(answer2, map2, 3, output_size[1], output_size[1]);

        for(int i=0;i<3;++i){
            cudaFree(image_new[i]);
        }
    }
    else{//不用扩充，过程和stride = 1时相同，不再赘述
        output_size[1] = (N-3)/2 + 1;
        CUDA_malloc(map2, output_size[1]);

        blockSize = thread_block_size;
        gridSize = (output_size[1]*output_size[1]*3)/thread_block_size;

        double *matrix2 ;//= NULL;
        cudaMallocManaged(&matrix2, \
                          output_size[1]*output_size[1]*27*sizeof(double));
        convertx(image, N, 2, matrix2);

        double *answer2 ;//= NULL;
        cudaMallocManaged(&answer2, \
                          output_size[1]*output_size[1]*3*sizeof(double));
        GEMM<<<gridSize, blockSize>>>(matrix2, kernel_, answer2, \
                                      output_size[1]*output_size[1], 3, 27);
        cudaDeviceSynchronize();
        // cout<<"answer2, when stride = 2 :"<<endl;
        // PRINT_single(answer2, output_size[1]*output_size[1], 3);

```

```
        recover_answer(answer2, map2, 3, output_size[1], output_size[1]);

    }
```

- 计算stride = 3时的卷积结果，原理和stride = 2的相同，具体过程不再赘述：

```
    //判断stride=3的时候是否需要填充
    padding = (3 - (N-3)%3)%3;
    //如果除余的结果是1，就需要填充"一圈",增加两个size;
    //如果是2，说明需要填充一维（）这里随机加在最上面和最左边
    if (padding){
        double *image_new2[3];
        CUDA_malloc(image_new2, N+padding);
        pad( image, N, image_new2, padding);//bug1
        output_size[2] = (N+padding-3)/3 + 1;
        CUDA_malloc(map3, output_size[2]);

        blockSize = thread_block_size;
        gridSize = (output_size[2]*output_size[2]*3)/thread_block_size;

        double *matrix3;
        cudaMallocManaged(&matrix3, output_size[2]*output_size[2]*27*sizeof(double));
        convertx(image_new2, N+padding, 3, matrix3);//bug2

        double *answer3;
        cudaMallocManaged(&answer3, output_size[2]*output_size[2]*3*sizeof(double));
        GEMM<<<gridSize, blockSize>>>(matrix3, kernel_, answer3,
output_size[2]*output_size[2], 3, 27);
        cudaDeviceSynchronize();
        // cout<<"answer3, when stride = 3 :"<<endl;
        // PRINT_single(answer3, output_size[2]*output_size[2], 3);

        recover_answer(answer3, map3, 3, output_size[2], output_size[2]);//bug3

        for(int i=0;i<3;++i){
            cudaFree(image_new2[i]);
        }
    }
    else{
        output_size[2] = (N-3)/3 + 1;
        CUDA_malloc(map3, output_size[2]);

        blockSize = thread_block_size;
        gridSize = (output_size[2]*output_size[2]*3)/thread_block_size;

        double *matrix3;
        cudaMallocManaged(&matrix3, output_size[2]*output_size[2]*27*sizeof(double));
        convertx(image, N, 3, matrix3);

        double *answer3;
        cudaMallocManaged(&answer3, output_size[2]*output_size[2]*3*sizeof(double));
        GEMM<<<gridSize, blockSize>>>(matrix3, kernel_, answer3,
output_size[2]*output_size[2], 3, 27);
        cudaDeviceSynchronize();
        // cout<<"answer3, when stride = 3 :"<<endl;
        // PRINT_single(answer3, output_size[2]*output_size[2], 3);

        recover_answer(answer3, map3, 3, output_size[2], output_size[2]);

    }

```

- 最后输出运算时间，运算结果，并清理内存：

```
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elapsedTime = 0;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    cout<<"所用时间为: "<< elapsedTime<<" <ms>"<<endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);


    // cout<<"image: "<<endl;
    // PRINT(image, N, 3);
    // cout<<"kernel:"<<endl;
    // cout<<"kernel_1: "<<endl;
    // PRINT(kernel_1, 3, 3);
    // cout<<"kernel_2: "<<endl;
    // PRINT(kernel_2, 3, 3);
    // cout<<"kernel_3: "<<endl;
    // PRINT(kernel_3, 3, 3);
    // cout<<"answer:"<<endl;
    // cout<<"when stride = 1, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map1, output_size[0], 3);
    // cout<<"when stride = 2, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map2, output_size[1], 3);
    // cout<<"when stride = 3, kerenl1/ kernel2/ kernel, the answer :"<<endl;
    // PRINT(map3, output_size[2], 3);


    for(int i=0; i<3; ++i){
        cudaFree(image[i]);
        cudaFree(kernel_1[i]);
        cudaFree(kernel_2[i]);
        cudaFree(kernel_3[i]);
        cudaFree(map1[i]);
        cudaFree(map2[i]);
        cudaFree(map3[i]);
    }

```

编译以及运行结果

- 编译

```
$ nvcc -o 4 4.cu
```

- 运行,<N>表示计算的矩阵规模，<thread_block_size>是每个block用到的线程数。

  注意：编译的时候$thread_block_size < N$，否则计算结果不正确。

```
$ ./4 <N> <thread_block_size>
```

- 验证计算结果的正确性，以及一些中间变量的正确性：
  - 将input转化后的矩阵A：

```
48 45 47 29 49 30 19 2 21 30 33 18 3 31 2 46 3 12 36 39 22 46 44 46 2 42 33
45 47 35 49 30 7 2 21 27 33 18 34 31 2 24 3 12 26 39 22 40 44 46 13 42 33 4
47 35 44 30 7 6 21 27 28 18 34 33 2 24 40 12 26 35 22 40 13 46 13 1 33 4 40
29 49 30 19 2 21 40 49 9 3 31 2 46 3 12 43 19 11 46 44 46 2 42 33 47 25 17
49 30 7 2 21 27 49 9 40 31 2 24 3 12 26 19 11 14 44 46 13 42 33 4 25 17 9
30 7 6 21 27 28 9 40 44 2 24 40 12 26 35 11 14 20 46 13 1 33 4 40 17 9 25
19 2 21 40 49 9 17 30 3 46 3 12 43 19 11 39 28 43 2 42 33 47 25 17 26 1 44
2 21 27 49 9 40 30 3 50 3 12 26 19 11 14 28 43 48 42 33 4 25 17 9 1 44 37
21 27 28 9 40 44 3 50 24 12 26 35 11 14 20 43 48 17 33 4 40 17 9 25 44 37 15
```

转化之前的input：

```
image:
when channel = 0
48 45 47 35 44
29 49 30 7 6
19 2 21 27 28
40 49 9 40 44
17 30 3 50 24

when channel = 1
30 33 18 34 33
3 31 2 24 40
46 3 12 26 35
43 19 11 14 20
39 28 43 48 17

when channel = 2
36 39 22 40 13
46 44 46 13 1
2 42 33 4 40
47 25 17 9 25
26 1 44 37 15
```

- 将kernel转化后的$W = [W_1 \ W_2 \ W_3]$：

```
jovyan@jupyter-lab6use:~$ ./4test 5 1
1 1 1
-1 -1 -1
0 0 0
0 0 0
-1 -1 -1
1 1 1
1 1 1
1 1 1
1 1 1
-1 -1 -1
0 0 0
-1 -1 -1
1 1 1
0 0 0
1 1 1
1 1 1
0 0 0
-1 -1 -1
0 0 0
0 0 0
-1 -1 -1
1 1 1
-1 -1 -1
-1 -1 -1
0 0 0
1 1 1
1 1 1
```

未转化前的三个kernel：

```
kernel_1:
when channel = 0
1 -1 0
0 -1 1
1 1 1

when channel = 1
-1 0 -1
1 0 1
1 0 -1

when channel = 2
0 0 -1
1 -1 -1
0 1 1

kernel_2:
when channel = 0
1 -1 0
0 -1 1
1 1 1

when channel = 1
-1 0 -1
1 0 1
1 0 -1

when channel = 2
0 0 -1
1 -1 -1
0 1 1

kernel_3:
when channel = 0
1 -1 0
0 -1 1
1 1 1

when channel = 1
-1 0 -1
1 0 1
1 0 -1

when channel = 2
0 0 -1
1 -1 -1
0 1 1
```

- 转换后的计算结果;

```
answer:
when stride = 1, keren11/ kernel2/ kernel, the answer :
when channel = 0
-82 -171 -87
-126 -93 -83
36 -162 -60

when channel = 1
100 33 0
-36 81 109
-3 58 13

when channel = 2
33 34 -63
-13 -51 -10
47 -8 -9

when stride = 2, keren11/ kernel2/ kernel, the answer :
when channel = 0
-82 -87
36 -60

when channel = 1
100 0
-3 13

when channel = 2
33 -63
47 -9

when stride = 3, keren11/ kernel2/ kernel, the answer :
when channel = 0
-84 -197
53 -60

when channel = 1
63 -35
0 13
```

未转化前的计算结果：

```
answer1, when stride = 1 :
-82 100 33
-171 33 34
-87 0 -63
-126 -36 -13
-93 81 -51
-83 109 -10
36 -3 47
-162 58 -8
-60 0 0

answer2, when stride = 2 :
-82 100 33
-87 0 -63
36 -3 47
-60 13 -9

answer3, when stride = 3 :
-84 63 6
-197 -35 -48
53 0 13
-60 13 -9
```

经过验证，这些结果都是正确的。

- 同上一个任务，输入从128增加至2048（4096太大显存放不下），thread_block_size一直设置为1024：

```
jovyan@jupyter-lab6use:~$ nvcc -o 4 4.cu
jovyan@jupyter-lab6use:~$ ./4 128 1024
所用时间为: 13.2588 <ms>
jovyan@jupyter-lab6use:~$ ./4 256 1024
所用时间为: 38.1041 <ms>
jovyan@jupyter-lab6use:~$ ./4 512 1024
所用时间为: 127.023 <ms>
jovyan@jupyter-lab6use:~$ ./4 1024 1024
所用时间为: 544.831 <ms>
jovyan@jupyter-lab6use:~$ ./4 2048 1024
所用时间为: 2102.87 <ms>
```

任务5：

> *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。
>
> ***CNN***参考资料，见实验发布网站
>
> 斯坦福人工智能课件*Convolutional Neural Networks，by Fei-Fei Li & Andrej Karpathy & Justin Johnson*
>
> 其他参考资料 （搜索以下关键词）
>
> *[1]*如何理解卷积神经网络（*CNN*）中的卷积和池化
>
> *[2] Convolutional Neural Networks (CNNs / ConvNets)* https://cs231n.github.io/convolutional-networks/

> *[3]im2col的原理和实现*
>
> *[4]cuDNN安装教程*
>
> *[5] convolutional-neural-networks*
>
> *https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks*

代码

- 需要的一些库函数的头文件以及命名空间：

```
#include<iostream>
#include<cstdlib>
#include<cudnn.h>
using namespace std;
```

- 设置一个宏，用来检查cudnn调用后返回的状态对象是否存在任何错误条件，并在出现问题时中止程序的执行并输出出现问题的代码行数。然后，利用这个宏我们可以简单地包装我们用该宏调用的任何库函数：cudnnCreatecudnnStatus_t

```
#define checkCUDNN(expression)                                 \
  {                                                            \
    cudnnStatus_t status = (expression);                       \
    if (status != CUDNN_STATUS_SUCCESS) {                      \
      std::cerr << "Error on line " << __LINE__ << ": "       \
                << cudnnGetErrorString(status) << std::endl; \
      std::exit(EXIT_FAILURE);                                 \
    }                                                          \
  }
```

- 一个主机调用的函数，用来输出结果：

```
__host__ void PRINT(float *matrix, int channel, int h, int w){
    int x = 0;
    for(int i=0;i<h;++i){
        for(int j=0;j<w;++j){
            for(int k=0;k<channel;++k){
                cout<<matrix[x++]<<" ";
            }
            cout<<endl;
        }
    }
    cout<<endl;
}
```

- 从下面开始都是主函数中的内容。
- 初始化，获取命令行中的参数，分别是用于卷积的矩阵的高和宽，以及边缘填充的圈数；

  设置句柄准备使用CUDNN的环境；

  并开始计时：

```
    int N,stride,padding;
    N = atoi(argv[1]);
    stride = atoi(argv[2]);
    padding = atoi(argv[3]);

    //handle
    cudnnHandle_t cudnn;
    cudnnCreate(&cudnn);

    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
```

- 下面对输入的数据进行描述，并为他分配空间，赋值随机数。

  input_descriptor保存了输入的信息，其中CUDNN_TENSOR_NHWC是数据数据的结构，这表示，input为一个四维的张量，四个维度分别为：N: Number这里作为输入的图片个数，这里设置为1；H: Height，即高；W: Width，即宽；C: Channels，即通道数，像前几个一样设置为3，依次类推。CUDNN_DATA_FLOAT为计算的数据类型。

```
    int batch = 1;
    int channels = 3;
    //input
    cudnnTensorDescriptor_t input_descriptor;
    checkCUDNN(cudnnCreateTensorDescriptor(&input_descriptor));
    checkCUDNN(cudnnSetTensor4dDescriptor(input_descriptor,
        /*format*/CUDNN_TENSOR_NHWC,// NHWC/NCWH
        /*dataType=*/CUDNN_DATA_FLOAT,
        /*batchsize*/batch,
        /*channels=*/channels,
        /*height=*/N,
        /*width=*/N));

    float *input;
    cudaMallocManaged( &input, batch*channels*N*N*sizeof(float) ) ;
    for(int channel=0; channel<channels;++channel){
        for(int h=0;h<N;++h){
            for(int w=0;w<N;++w) {
                input[channel*N*N+h*N+w] = rand()%50;
            }
        }
    }
```

- 下面对卷积核kernel进行描述，设置卷积中需要的一些参数，并为他分配空间，赋值随机数：

```
    //kernel
    cudnnFilterDescriptor_t kernel_descriptor;
    checkCUDNN(cudnnCreateFilterDescriptor(&kernel_descriptor));
    checkCUDNN(cudnnSetFilter4dDescriptor(kernel_descriptor,
                                CUDNN_DATA_FLOAT,
                                CUDNN_TENSOR_NCHW,
                                 /*相当于有几个卷积核out_channels=*/3,
                                 /*每个卷积核有多少个channel，in_channels=*/3,
                                 /*kernel_height=*/3,
                                 /*kernel_width=*/3));


    //描述卷积内核
    cudnnConvolutionDescriptor_t convolution_descriptor;
    checkCUDNN(cudnnCreateConvolutionDescriptor(&convolution_descriptor));
```

```
    checkCUDNN(cudnnSetConvolution2dDescriptor(convolution_descriptor,
                                        /*填充的圈数pad_height=*/padding,
                                               /*pad_width=*/padding,
                           /*kernel每次滑动的步数vertical_stride=*/stride,
                                               /*horizontal_stride=*/stride,
                                               /*dilation_height=*/1,
                                               /*dilation_width=*/1,
                                               /*mode=*/CUDNN_CROSS_CORRELATION,
                                               /*computeType=*/CUDNN_DATA_FLOAT));


    float *kernel_;// NCHW注意这里的排列方式和输入的NHCW不一样。
    cudaMallocManaged( &kernel_, 3*3*3*3*sizeof(float) );//3个kernel每个kernel有3个channel，每个channel有3*3个变量。
    int a = -1, b =1;//kernel的变量范围是[-1,1]的整数
    for (int kernel = 0; kernel < 3; ++kernel) {
       for (int channel = 0; channel < 3; ++channel) {
            for (int row = 0; row < 3; ++row) {
                for (int column = 0; column < 3; ++column) {
                    kernel_[((kernel*3+channel)*3+row)*3+column] = rand()%(b-a+1) + a;
                }
            }
        }
    }
```

- 计算卷积结果（输出）的大小，对输出进行描述，并为他分配空间：

```
    //计算输出的大小
    int out_n, out_c, out_h, out_w;
    checkCUDNN(cudnnGetConvolution2dForwardOutputDim(
        convolution_descriptor, input_descriptor, kernel_descriptor,
        &out_n, &out_c, &out_h, &out_w));
    cout<<"the output: "<<out_n<<" "<<out_c<<" "<<out_h<<" "<<out_w<<endl;

    //output
    cudnnTensorDescriptor_t output_descriptor;
    checkCUDNN(cudnnCreateTensorDescriptor(&output_descriptor));
    checkCUDNN(cudnnSetTensor4dDescriptor(output_descriptor,
        /*format*/CUDNN_TENSOR_NHWC,// NHWC
        /*dataType=*/CUDNN_DATA_FLOAT,
        /*batchsize*/out_n,
        /*channels=*/out_c,
        /*height=*/out_h,
        /*width=*/out_w));


        float *output;
        cudaMallocManaged( &output, out_n*out_c*out_h*out_w*sizeof(float) );
```

- 描述卷积算法：

```
    // 卷积算法的描述
    cudnnConvolutionFwdAlgo_t convolution_algorithm;
    checkCUDNN(
        cudnnGetConvolutionForwardAlgorithm(cudnn,
        input_descriptor,
        kernel_descriptor,
        convolution_descriptor,
        output_descriptor,
        CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, //
CUDNN_CONVOLUTION_FWD_SPECIFY_WORKSPACE_LIMIT（在内存受限的情况下，memoryLimitInBytes 设置非 0 值）
        /*memoryLimitInBytes=*/0,
        &convolution_algorithm));
```

- 计算卷积需要的内存空间，并为他分配这么多的内存空间：

```
// 计算 cuDNN 它的操作需要多少内存
size_t workspace_bytes{ 0 };
checkCUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn,
    input_descriptor,
    kernel_descriptor,
    convolution_descriptor,
    output_descriptor,
    convolution_algorithm,
    &workspace_bytes));

    // 分配内存， 从 cudnnGetConvolutionForwardWorkspaceSize 计算而得
    void* d_workspace{ nullptr };
    cudaMallocManaged(&d_workspace, workspace_bytes);
```

- 进行卷积操作（这里是前向卷积）,里面的变量都是之前设置的：

```
    const float alpha = 1.0f, beta = 0.0f;
    // 进行卷积操作
    checkCUDNN(cudnnConvolutionForward(cudnn,
        &alpha,
        input_descriptor,
        input,
        kernel_descriptor,
        kernel_,
        convolution_descriptor,
        convolution_algorithm,
        d_workspace, // 注意，如果我们选择不需要额外内存的卷积算法，d_workspace可以为nullptr。
        workspace_bytes,
        &beta,
        output_descriptor,
        output));
```

- 最后进行输出和清理内存：

```
    cout<<"kernel:"<<endl;
    PRINT(kernel_,3*3,3,3);
    cout<<"input:"<<endl;
    PRINT(input,3,N,N);
    cout<<"output:"<<endl;
    PRINT(output,out_c,out_h,out_w);

    cudaFree(d_workspace);
    cudaFree(input);
    cudaFree(output);
    cudaFree(kernel_);
```

```
    cudnnDestroyTensorDescriptor(input_descriptor);
    cudnnDestroyTensorDescriptor(output_descriptor);
    cudnnDestroyConvolutionDescriptor(convolution_descriptor);
    cudnnDestroyFilterDescriptor(kernel_descriptor);

    cudnnDestroy(cudnn);
```

编译以及运行结果

- 编译指令：

```
$ nvcc -o 5 5.cu -lcudnn -I /opt/conda/include/
```

使用的是学校的集群：

要用到cudnn，加上相关的编译参数`-lcudnn`。它动态链接库的so文件在`/opt/conda/lib/`中，由于编译的时候总是报错说`/usr/lib/`在找不到该动态库，参考linux下指定运行程序所需要的.so文件路径的四种方式 ，使用指令`cp  /opt/conda/lib/libcudnn.so /usr/lib`将.so文件放到/usr/lib目录下，就能成功使用`-lcudnn`，进行动态链接库了。

并且由于刚开始说找不到库函数`#include<cudnn.h>`，需要将库函数的路径在编译命令的最后进行链接：`-I /opt/conda/include/`。

- 运行：

```
$ ./5 <N> <stride> <padding>
```

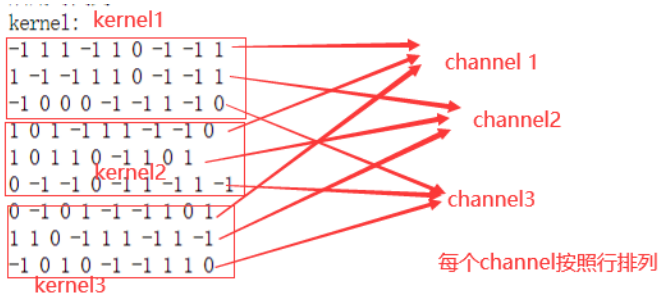
- 运行结果：

验证较大矩阵的运算结果的规模，N=128，stride=1，padding=0。运算结果的数量为1，channel为3（3个kernel的计算结果），N=126，这个输出规模符合预期：

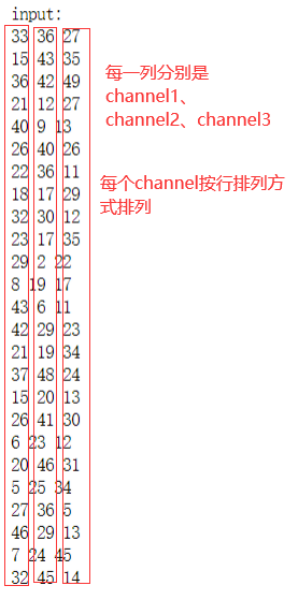
```
jovyan@jupyter-lab6use:~$ nvcc -o 5 5.cu -lcudnn -I /opt/conda/include/
jovyan@jupyter-lab6use:~$ ./5 128 1 0
the output: 1 3 126 126
所用时间为: 2.08794 <ms>
```

运行较小的矩阵规模进行验证卷积结果的正确性，举证规模为5，stride为1，不进行填充`./5 5 1 0`：

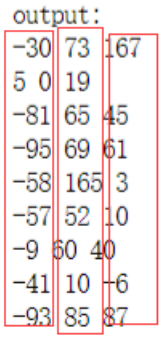
  - 输出kernel，按照NCHW排列:



  - 输出input，按照NHCW排列：



- 输出output，按照NHCW排列：



即：

kernel:

$$kernel1:channel1=\begin{bmatrix}-1&1&1\\-1&1&0\\-1&-1&1\end{bmatrix},channel2=\begin{bmatrix}1&-1&-1\\1&1&0\\-1&-1&1\end{bmatrix}.channel3=\begin{bmatrix}-1&0&0\\0&-1&-1\\1&-1&0\end{bmatrix}$$

$$kernel2:channel1=\begin{bmatrix}1&0&1\\-1&1&1\\-1&-1&0\end{bmatrix},channel2=\begin{bmatrix}1&0&1\\1&0-&1\\1&0&1\end{bmatrix},channel3=\begin{bmatrix}0&-1&-1\\0&-1&1\\-1&1&-1\end{bmatrix}$$

$$kernel3:channel1=\begin{bmatrix}0&-1&0\\1&-1&-1\\1&0&1\end{bmatrix},channel2=\begin{bmatrix}1&1&0\\-1&1&1\\-1&1&-1\end{bmatrix}channel3=\begin{bmatrix}-1&0&1\\0&-1&-1\\1&1&0\end{bmatrix}$$

input:

$$channel1=\begin{bmatrix}33&15&36&21&40\\26&22&18&32&23\\29&8&43&42&21\\37&15&26&6&20\\5&27&46&7&32\end{bmatrix},\quad channel2=\begin{bmatrix}36&43&42&12&9\\40&36&17&30&17\\2&19&6&29&19\\48&20&41&23&46\\25&36&29&24&45\end{bmatrix},$$

$$channel3=\begin{bmatrix}27&35&49&27&13\\26&11&29&12&35\\22&17&11&23&34\\24&13&30&12&31\\34&5&13&45&14\end{bmatrix}$$

output:

$$kernel1\text{的卷积结果：}\begin{bmatrix}-30 & 5 & -81\\ -95 & -58 & -57\\ -9 & -41 & -93\end{bmatrix}$$

$$kernel2\text{的卷积结果：}\begin{bmatrix}73 & 0 & 65\\ 69 & 165 & 52\\ 60 & 10 & 85\end{bmatrix}$$

$$kernel3\text{的卷积结果：}\begin{bmatrix}167 & 19 & 45\\ 61 & 3 & 10\\ 40 & -6 & 87\end{bmatrix}$$

经过验证这个计算结果正确。

- 由于之前任务5的CNN是在一个函数计算了三种stride，而cudnn中每次只计算一种stride，

  为了方便比较运行时间，下面是任务3中只计算stride = 1，且规模从128到2048的运行时间：

```
jovyan@jupyter-lab6use:~$ nvcc -o 3 3.cu
jovyan@jupyter-lab6use:~$ ./3 128 1024
所用时间为: 2.84979 <ms>
jovyan@jupyter-lab6use:~$ ./3 256 1024
所用时间为: 9.68499 <ms>
jovyan@jupyter-lab6use:~$ ./3 512 1024
所用时间为: 20.6397 <ms>
jovyan@jupyter-lab6use:~$ ./3 1024 1024
所用时间为: 90.7223 <ms>
jovyan@jupyter-lab6use:~$ ./3 2048 1024
所用时间为: 346.292 <ms>
```

  同样为了方便比较运行时间，下面是任务4中只计算stride = 1，且规模从128到2048的运行时间：

```
jovyan@jupyter-lab6use:~$ nvcc -o 4 4.cu
jovyan@jupyter-lab6use:~$ ./4 128 1024
所用时间为: 7.76192 <ms>
jovyan@jupyter-lab6use:~$ ./4 256 1024
所用时间为: 24.1306 <ms>
jovyan@jupyter-lab6use:~$ ./4 512 1024
所用时间为: 98.0664 <ms>
jovyan@jupyter-lab6use:~$ ./4 1024 1024
所用时间为: 350.899 <ms>
jovyan@jupyter-lab6use:~$ ./4 2048 1024
所用时间为: 1404.64 <ms>
```

  本次任务规模从128到2048的运行时间，为了方便将stride设置为1，padding设置为0，只改变矩阵规模的大小：

```
jovyan@jupyter-lab6use:~$ nvcc -o 5 5.cu -lcudnn -I /opt/conda/include/
jovyan@jupyter-lab6use:~$ ./5 128 1 0
所用时间为: 2.49754 <ms>
jovyan@jupyter-lab6use:~$ ./5 256 1 0
所用时间为: 5.21933 <ms>
jovyan@jupyter-lab6use:~$ ./5 512 1 0
所用时间为: 18.6112 <ms>
jovyan@jupyter-lab6use:~$ ./5 1024 1 0
所用时间为: 73.984 <ms>
jovyan@jupyter-lab6use:~$ ./5 2048 1 0
所用时间为: 296.592 <ms>
```

  对比：

N	CNN<ms>	im2col<ms>	cudnn<ms>
128	2.84976	7.76192	2.49754
256	9.68499	24.1306	5.21933
512	20.6397	98.0664	18.6112
1024	90.7223	350.899	73.984
2048	346.292	1404.64	296.592

很明显cudnn的性能最好，其次是直接计算CNN，最后是使用im2col的方法。优化还是从线程的分配以及内存的分配还有算法实现的优化这几方面入手。

- 还有需要注意的是上面对cudnn的计时忽略了创建句柄的时间，因为句柄是一个全局资源，可能存在加锁解锁操作，那就很有可能瓶颈在这个申请句柄全局资源上，这里对比一下不忽略申请句柄资源的时间，耗时非常多：

```
jovyan@jupyter-lab6use:~$ nvcc -o 5 5.cu -lcudnn -I /opt/conda/include/
jovyan@jupyter-lab6use:~$ ./5 128 1 0
所用时间为: 1522.88 <ms>
jovyan@jupyter-lab6use:~$ ./5 256 1 0
所用时间为: 1605.5 <ms>
jovyan@jupyter-lab6use:~$ ./5 512 1 0
所用时间为: 1446.19 <ms>
jovyan@jupyter-lab6use:~$ ./5 1024 1 0
所用时间为: 2073.03 <ms>
jovyan@jupyter-lab6use:~$ ./5 2048 1 0
所用时间为: 2422.22 <ms>
```

博客参考

- CUDA编程入门极简教程
- CUDA编程(三): GPU架构了解一下!
- cuda中threadIdx、blockIdx、blockDim和gridDim的使用
- cublas 矩阵乘法
- CUDA中的计时函数
- 一文搞定3D卷积
- CNN基础知识——卷积（Convolution）、填充（Padding）、步长(Stride)
- 卷积神经网络CNN概述下（RGB图像卷积与单层神经网络）
- im2col方法实现卷积算法 写im2col的时候就只看了这个，很清楚
- CUDA编程 cudaDeviceSynchronize()返回值700的意思_cudadevicesynchronize函数
- 【CUDA】常见错误类型cudaError_t人工智能算法与工程实践-*CSDN*博客cuda error
- 【 C语言指针详解 】(七）野指针
- 【CUDA教程】四、异常处理与编程技巧
- 【CUDA教程】二、主存与显存
- Convolutions with cuDNN – Peter Goldsborough
- 使用CuDNN进行卷积运算___cudnn 卷积
- 使用 CuDNN 进行卷积运算【读书笔记】
- 还有老师给的一些文档

总结与思考

了解并熟悉了cuda编程、cublas库、CNN、im2col方法以及cudnn等内容。并且学习了卷积方面相关的内容，收获良多。

- 遇到的问题：

  本次实验中还是遇到了很多问题：

  比如任务2中的向kernel函数传递参数，刚开始传递的是二维指针参数，但是运行结果出错，查找问题的时候发现计算的时候好像找不到输入的二维指针指向的对应位置的数据，但是把它写成主机函数又能找到了，也能得到正确的计算结果，最后尝试将二维指针改为一维指针，传入多个一维指针，运算结果正确。但是我还是不知道这个错误的根本原因是什么，可能还是要在之后的学习中去得到答案了。

  还有任务3，出现的问题是当矩阵规模到512的时候就出现了段错误`Segmentation fault (core dumped)`，这并不合理，到4096是显存不够，那512应该还是代码的问题，于是查找问题。这里的bug真的找了好久，最后发现是在stride=3时分配内存出了错，发现可以输出错误返回值，输出后发现分配内存的时候出现`cudaErrorIllegalAddress = 700`。然后查找原因，很多博客说这是传入的指针问题，又查看了一些野指针的问题，但是最后仍然没有改对。最后是随便尝试删掉了我在stride=2结束以后释放matrix和answer2的内存空

间的代码，结果能够正确运行了，我猜测这是释放内存空间导致的指针问题。最后我的经验教训就是对于指针的使用以及释放内存空间一定要小心使用。

任务5就是对cudnn的调用过程以及对各个参数的理解，如果参数写不对那运算结果一定是有问题的。还有就是在任务5没有搞清楚参数含义的时候出现报错`CUDNN_STATUS_BAD_PARAM`，这是由于输入的参数无法正确匹配。

以及一些需要动态链接库，添加相关的编译参数的问题，可能因为没有添加这些编译参数/头文件导致出错。

- 不足：

  最大的不足是任务3、4的CNN的实现上，因为实际上不同的stride的卷积可以写成一个部分，然后通过参数来控制本次卷积stride，但是我写这两个任务的时候没有意识到，直到做到任务5。所以我是将每种stride的实现都写出来了，实际上这块内容可以进行优化。