Project3

12112504 郭城

程序设计：以程序运行顺序介绍

程序需要两个 txt 文件，一个为颜色数据文件，另一个为 filter 文件（含 kernel 大小以及 filter 矩阵）

注：图片转矩阵以及矩阵转图片操作均使用库更加成熟的 python 进行实现，因学习目标在于卷积以及优化，故在此省略）

注：更改 padding 以及 strides 需手动在程序中进行更改

1. 程序可以读取 txt 中的预先 RGB 矩阵化后的图片文件，文件第一行格式固定为 3 个数据，分别为高（**h**）宽（**w**）和层数（本程序默认为 3 层）。其余颜色数据只支持 float 及以下类型，txt 中不含标点符号，数据以空格和换行隔开。
2. 读取第一行数据后，按照给定的高宽使用 **arrMalloc** 方法给 3 个数组（分别代表 RGB）分配内存，

arrMalloc 分配内存：输入高和宽，**h，w**，创建指针 float ** arr，

给 arr 分配内存 arr = (float **)malloc(h * sizeof(float*));
然后给内部 h 大小的指针循环分别分配内存 arr[i] = (float *)malloc(w * sizeof(float));
以此达成内存连续，且使用时依旧可以作为二位数组使用。返回 arr

3. 使用两个 for 循环将数据 fscanf 输入数组（作为前期准备工作，未进行优化，且未写成方法，因为数据排布方式以像素为基点，所以在 main 中直接 for 循环将有利于直接生成 3 组数据）
4. 同样方式 **DataReading（for 循环读取单组数据）**读取 filter，分配内存并录入矩阵 filter
5. 设定 kernal 大小 kernal1，kernal2，填充大小 padding，步长大小 strides
6. 对三层颜色数据进行卷积层操作 **Convolutional**

Convolutional：输入变量，颜色矩阵，filter，高宽，kernel 大小，填充，步长

创建分配与颜色矩阵同样大小内存的矩阵 arr，以步长为间隔对高宽进行循环，循环内对每一个像素进行卷积：**clOneBlock**

clOneBlock: 输入需要卷积的目标像素坐标 **x，y**，其余输入与 convolutional 相同。

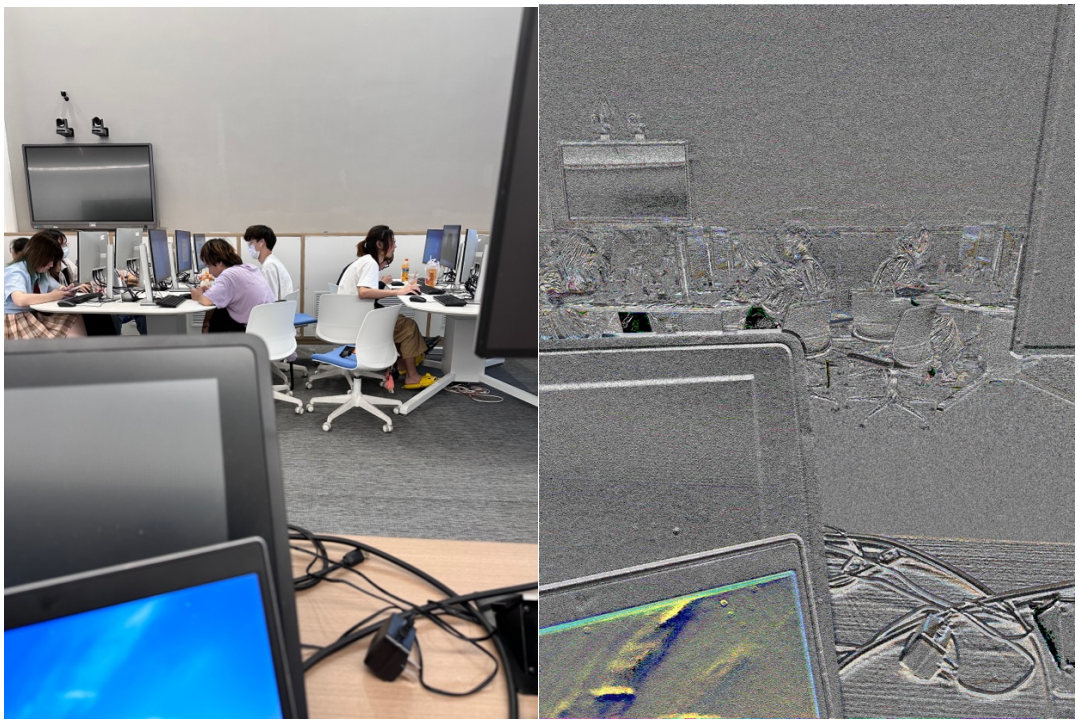
双重 for 循环求对原图核覆盖部份与 filter 相乘并求卷积和，超出边界部分用 padding 填充，并加入计算。（未优化版本）
回到 **Convolutional** 将得出的颜色值录入 arr，返回卷积结果 arr 矩阵。

7. 将得出结果输出为 txt 文件
8. 最终使用 free 释放多个 array 的内存。

测试：使用 3*3 边缘检测 filter

-1 0 1
-2 0 2
-1 0 1

对图像（4K）进行处理，填充为 0，步长为 1
查看结果：



结果正确

优化 :主要对 **clOneBlock** 函数进行优化 **clOneBlock_opt**, 由于 filter 存在不同大小 1*1,

3*3，5*5，以及更多，因此只要不为 1*1，就可以对乘加的过程进行 SIMD 优化，使 4 次乘加同时进行，因 filter 可能不为 4 的倍数，因此需要对溢出的 1-3 个数据进行单独处理

clOneBlock_opt :优化函数创建 float32x4_t 向量 **vec** 默认赋值 0.0f, 以及 float32x4_t 向量 **pad**，默认赋值 **padding**。
以 4 为间隔进行递归，递归内每次循环创建图像以及 filter 对应相乘位置的 float32x4_t 数据 **image_v** 以及 **filter_v**,并给 vec 赋值 vmlaq_f32(vec, image_v , filter_v);即二者相乘的结果。
若所使用像素格超出边界，则使用 pad 填充与 filter_v 相乘 vmlaq_f32(vec, pad, filter_v);
离开循环后，将值赋给四维 float 数组并加和，得出该格卷积初步结果。
若 filter 不能被 4 整除，则一定余数为 1，2，3 中的一个，分别进行补充操作。
最终返回结果

方法中 **Convolutional_opt** 仅为使用 **clOneBlock_opt** 的优化方法，未进行更多优化

结果：（编程时有考虑到使用 openmp 对程序进行进一步加速，程序中包括 **Convolutional，**都含有 for 循环，但是由于使用设备为 M2 芯片 macbook，多次尝试对 openmp 库调用但是没找到合适调用方式因此搁置）

在 main 中分别对 **clOneBlock** 以及 **clOneBlock_opt** 优化前以及优化后对同一图片 3 通道矩阵进行处理并记录程序运行时间，以下数据均为多次运行后观测无较大波动结果使用图片大小为 4032*2268*3，filter 为上述 3*3 边缘检测 filter，kernal 相应为 3*3，填充为 0，步长为 1：

1*1kernal filter

```
normal: 0.181356 ms
optmized: 0.675670 ms
```

3*3kernal filter

```
normal: 0.801554 ms
optmized: 0.994385 ms
```

5*5kernal filter

```
normal: 2.378318 ms
optmized: 1.728465 ms
```

从数据看出，在 1*1 时 “优化后” 方法远远慢于未优化，随着 kernal 提升，“优化后” 方法效率将高于未优化，分析其原因为，因使用 SIMD 的优化算法中在循环意外增加了大量 if，else 判断（包含不能整除 4 后的补值操作等）以及赋值操作，在循环量较低时，即 kernal 很小时，“优化后” 算法因增加的大量判断将慢于未优化算法，但是在数据量增大后，for 循环内优化后 4 条路同时进行的优势将体现。

因此该优化适用于 kernal 较大时，进一步优化想法为：小 kernal 时不使用优化。

使用-o3 进行编译时未对结果产生影响。

注：整体程序运行时间远大于上述数据，判断为读取以及写出数据所使用时间较长。

程序特点：除对卷积的优化以外，在使用数组时对内存的处理也为本程序特点，以上述方法进行内存分配可以得到连续内存，且使用时与常规数组相同。释放内存时使用 **freeArr** 方法

freeArr：

```
for (int i = 0; i < h; i++) {
        free(arr[i]);
    }
    free(arr);
```

释放内部内存之后再释放整体内存，以完整释放内存