并行计算实验五报告
——常用图像处理算法的并行及优化

PB20111701 叶升宇 **PB20111689 蓝俊玮**

说明

本此实验由 PB20111689 蓝俊玮完成。

1. 实验题目

常用图像处理算法的并行及优化。

这里选择均值滤波器。(本质上与中值滤波器，锐化滤波器等是相同的，都是对图像进行卷积操作，只需要读取相应的领域信息，再做相应的处理。)

2. 实验环境

服务器：

- 操作系统：Ubuntu 18.04.6 LTS
- 处理器：Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz
- GPU: NVIDIA GeForce RTX 2080 Ti

3. 均值滤波

均值滤波(Mean Filter)是一种常见的图像处理方法，用于平滑图像并减少噪声。该方法通过将每个像素的值替换为其周围像素值的平均值来实现。

均值滤波的步骤如下：

- 对于每个像素，取其周围一个固定大小的邻域(通常为一个正方形或矩形)，例如 3×3 或 5×5 的邻域。
- 对于邻域中的所有像素，计算它们的像素值的平均值。
- 将该像素的像素值替换为平均值。

均值滤波可以有效地去除图像中的噪声，例如椒盐噪声和高斯噪声。同时，它也可以平滑图像并减少细节，因此在一些应用中可能会导致图像模糊。由于均值滤波的计算效率相对较低，特别是当邻域大小较大时。因此，在实际应用中，可能需要使用更高效的算法或 GPU 加速来加速均值滤波的计算。

所以下面将介绍如何通过 Cuda 编程来实现高效高性能的均值滤波器。

3.1. 单 block 多 threads

这个是最简单的实现形式，将 threads 定义成为二维的，然后在每一维对每个像素点进行并行计算。

```
__global__ void mean_filter1(unsigned char *img, unsigned char *filter, int width, int
height) {
    float temp_f;

    for(int y = threadIdx.y; y < height; y += blockDim.y) {
        if (y <= 0 || y >= height - 1)
            continue;
```

```
        for (int x = threadIdx.x; x < width; x += blockDim.x) {
            if (x <= 0 || x >= width - 1)
                continue;
            temp_f = 0.0;
            temp_f += img[(y - 1) * width + x - 1];
            temp_f += img[(y - 1) * width + x];
            temp_f += img[(y - 1) * width + x + 1];
            temp_f += img[y * width + x - 1];
            temp_f += img[y * width + x];
            temp_f += img[y * width + x + 1];
            temp_f += img[(y + 1) * width + x - 1];
            temp_f += img[(y + 1) * width + x];
            temp_f += img[(y + 1) * width + x + 1];
            temp_f /= 9;
            filter[y * width + x] = (unsigned char) temp_f;
        }
    }
}
```

3.2. 多 blocks 多 threads

同理，采用多 blocks 的时候也是一样的。只需要修改每个线程处理的起始位置和步长即可。实现也很简单。

```
__global__ void mean_filter2(unsigned char *img, unsigned char *filter, int width, int
height) {
    float temp_f;
    int id_y = blockIdx.y * blockDim.y + threadIdx.y;
    int id_x = blockIdx.x * blockDim.x + threadIdx.x;
    int stride_y = gridDim.y * blockDim.y;
    int stride_x = gridDim.x * blockDim.x;

    for(int y = id_y; y < height; y += stride_y) {
        if (y <= 0 || y >= height - 1)
            continue;
        for (int x = id_x; x < width; x += stride_x) {
            // 与上面的相同
        }
    }
}
```

3.3. 共享内存

说明

首先需要说明的是，共享内存的使用和图像的大小是有密切联系的。在设置 `blockSize` 和 `gridSize` 的时候，必须要参考图像的大小进行设置。在本次的实验中，我所处理的图像是加了椒盐噪声的 lena 灰度图像，其图像大小是 256×256 的，因此我们将 `blockSize` 和 `gridSize` 分别设置为 16×16 和 16×16 大小。

首先我们知道，在进行卷积操作的时候，由于我们设置的滤波器核大小为 3×3，而滤波器滑动的步长是 1，因此在滑动的过程中存在重复访存的现象(即访问同一地址的像素值)。因此我们可以使用共享内存来存储像素值，以减少访问全局内存的次数，加快图像处理的速度。

为了存储这些像素值，我们定义了共享内存 `__shared__ unsigned char img_shared[18][18];` 将共享内存的大小定义为 18×18 ,这是为了填补边界信息 ,使得可以完成统一的存储过程(即减少边界判断的情况并且可以处理图像的边缘)。

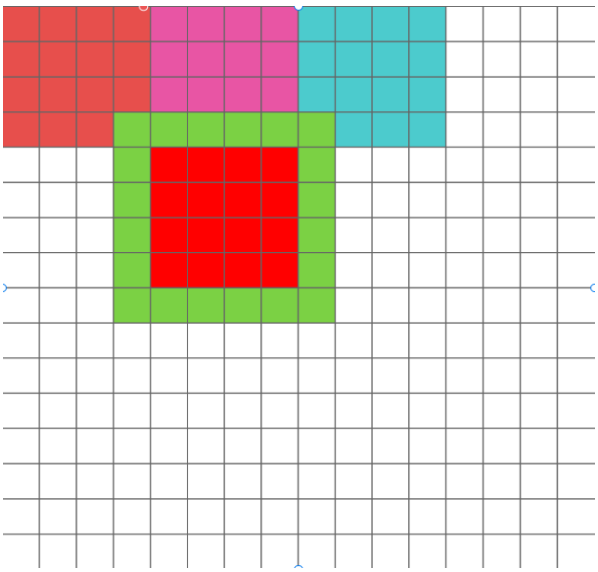
由于我们扩展了边界，所以在存储正常像素的时候，需要将位置加 1 ：`img_shared[tid_y + 1][tid_x + 1] = img[y * width + x];`。然后在处理图像边缘的时候，需要通过线程号码来进行判断(实际上等同于边界检查)。

说明

在这里，我说的图像边界实际上不是原来 256×256 图像的边界，实际上指的是每个线程块内的图像边界。因为不同线程块是不能共享内存的，因此当处理到当前线程块内的边界时，但实际上它在原图像中并不是边界，但是我们需要获取其领域像素值。此时其领域像素值位于其它线程块内，我们不能再通过统一的获取方式 (`img_shared[tid_y + 1][tid_x + 1] = img[y * width + x];`) 来获取其信息了，因此需要通过检查线程号来进行存储

举个例子：

设当前线程的信息为：`tid_x = 0, tid_y = 0, bid_x = 1, bid_y = 1`。表明当前线程处理的实际上是原图像 (16, 16) 的位置。此时该像素点的坐标并不是原图像的边界。但是它在当前线程块内是处于边界的 (因为 `tid_x = 0, tid_y = 0`)。我们要获取其领域的信息，如其上面的像素值 `tid_x = 0, tid_y = 15, bid_x = 1, bid_y = 0` 属于另一个线程块内。所以我们需要通过特定的判断，来补充该图像缺失的边缘信息。用图像来直观展示：



可以看出，在上面的例子，`tid_x = 0, tid_y = 0, bid_x = 1, bid_y = 1` (这里假设 blockSize 为 4×4) ，即我们可以看到 ，虽然一个 block 的大小是 4×4 的，但是我们需要存储的像素值不止 4×4 ，而是要将其周围的边界信息都加上 ，这是这圈边界信息是在 block 之外的。因此我们需要对这些边界像素进行存储。因此可以解释下面的做法：

```
__global__ void mean_filter3(unsigned char *img, unsigned char *filter, int width, int
height) {
    __shared__ unsigned char img_shared[18][18];
    float temp_f;

    int tid_x = threadIdx.x;
    int tid_y = threadIdx.y;
```

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

    if (y > 0 && y < height && x > 0 && x < width) {
        img_shared[tid_y + 1][tid_x + 1] = img[y * width + x];
    }

    if (tid_x == 0 && x > 0) {
        img_shared[tid_y + 1][0] = img[y * width + x - 1];
    }

    if (tid_x == 15 && x < width - 1) {
        img_shared[tid_y + 1][17] = img[y * width + x + 1];
    }

    if (tid_y == 0 && y > 0) {
        img_shared[0][tid_x + 1] = img[(y - 1) * width + x];
    }

    if (tid_y == 15 && y < height - 1) {
        img_shared[17][tid_x + 1] = img[(y + 1) * width + x];
    }

    if (tid_x == 0 && tid_y == 0 && x > 0 && y > 0) {
        img_shared[0][0] = img[(y - 1) * width + x - 1];
    }

    if (tid_x == 15 && tid_y == 0 && x < width - 1 && y > 0) {
        img_shared[0][17] = img[(y - 1) * width + x + 1];
    }

    if (tid_x == 0 && tid_y == 15 && x > 0 && y < height - 1) {
        img_shared[17][0] = img[(y + 1) * width + x - 1];
    }

    if (tid_x == 15 && tid_y == 15 && x < width - 1 && y < height - 1) {
        img_shared[17][17] = img[(y + 1) * width + x + 1];
    }

    __syncthreads();

    if (y > 0 && y < height - 1 && x > 0 && x < width - 1) {
        temp_f = 0.0;

        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                temp_f += img_shared[tid_y + i + 1][tid_x + j + 1];
            }
        }

        temp_f /= 9;
        filter[y * width + x] = (unsigned char) temp_f;
    }
}
```

3.4. 纹理存储

由于老师课件上的纹理存储代码很多已经被弃用，因此我接下来将采用纹理对象进行实现。

首先创建一个一维的 `cudaArray *srcArray` 来存储图像的像素值。这里将二维的图像像素值转化成为了一维数组。

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc(8, 0, 0, 0, cudaChannelFormatKindUnsigned);
cudaMallocArray(&srcArray, &channelDesc, width, height);
cudaMemcpy2DToArray(srcArray, 0, 0, image.data, width * sizeof(unsigned char), width * sizeof(unsigned char), height, cudaMemcpyHostToDevice);
```

说明

由于通过 `cv::imread()` 读取的像素值的类型是灰度值的 `unsigned char`，所以创建通道 channel 只需要一维，且类型为 `unsigned` 类型，由于 `unsigned char` 是 8 字节的，因此我们在第一维的信息为 8 (这里需要和官网以及网上大多数示例中的 `cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindFloat);` 区别开)

接着就是创建纹理对象。首先创建源信息 `struct cudaResourceDesc resDesc`，并且将图像的像素值绑定上去 `resDesc.res.array.array = srcArray;`。然后创建纹理目标，这里将寻址模式设置为 `cudaAddressModeBorder`，即超出图像边界的值都按 0 进行处理。同时将滤波模式 `filterMode` 设置为点滤波 `cudaFilterModePoint`。

说明

需要注意的是，滤波模式不能设置为线性滤波 `cudaFilterModeLinear`，这是因为我们采用的类型 `unsigned char`，这是整型类型，如果采用线性滤波的话，会产生浮点值，会与该类型冲突。

接着就是对坐标进行归一化 (不归一化也是可以的)。

```
struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeArray;
resDesc.res.array.array = srcArray;

struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.addressMode[0] = cudaAddressModeBorder;
texDesc.addressMode[1] = cudaAddressModeBorder;
texDesc.filterMode = cudaFilterModePoint;
texDesc.readMode = cudaReadModeElementType;
texDesc.normalizedCoords = 1;

cudaTextureObject_t texObj = 0;
cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);
```

使用纹理存储的核函数如下：

```
__global__ void mean_filter4(unsigned char *filter, cudaTextureObject_t texObj, int width, int height) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    float temp_f;
```

```
    if (y > 0 && y < height - 1 && x > 0 && x < width - 1) {
        temp_f = 0.0;

        for (int i = -1; i <= 1; i++) {
            for (int j = -1; j <= 1; j++) {
                temp_f += tex2D<unsigned char>(texObj, (x + j + 0.5) / width, (y + i +
0.5) / height);
            }
        }
        temp_f /= 9;
        filter[y * width + x] = (unsigned char) temp_f;
    }
}
```

说明

我们通过 `tex2D<unsigned char>` 来获取纹理内的信息，记得要有 `<unsigned char>`。然后就是取坐标的时候要加上 0.5，加上 0.5 是为了使纹理坐标指向像素的中心，如果不考虑像素的中心，会导致采样到的像素值不准确。

同时用于之前我们设置了归一化坐标，因此我们需要在取值时对坐标进行处理：`(x + j + 0.5) / width, (y + i + 0.5) / height`

3.5. 性能测试

实现方法	时间(ms)	加速比
串行程序	1.296000	-
单block多threads	0.095072	13.63
多blocks多threads	0.010848	119.47
共享内存	0.010688	121.26
纹理存储	0.033472	38.72

可以看出，并行的优化效果还是不错的。多 blocks 多 threads 和共享内存的执行效果差不多。感觉只是因为图片的规模太小，只有 256×256，所以共享内存的优势并没有体现出来。而纹理存储的执行效果就较差。我觉得原因是因为读取纹理存储的速度要慢于共享内存，纹理存储的访问方式更加复杂，通常需要进行插值和边缘处理等操作，以保证采样的精度和正确性，这些操作会增加纹理存储的访问延迟。同时在绑定纹理对象的时候，我们也能看到纹理存储通常需要进行数据类型转换和格式转换等操作，这些操作也会增加访问延迟和计算开销。同时这些增加的处理时耗会大于纹理存储提供的便捷性，因此表现出比多 blocks 多 threads 程序更慢的效果。原因可能还是在于图像的大小不够大。

3.6. 结果展示

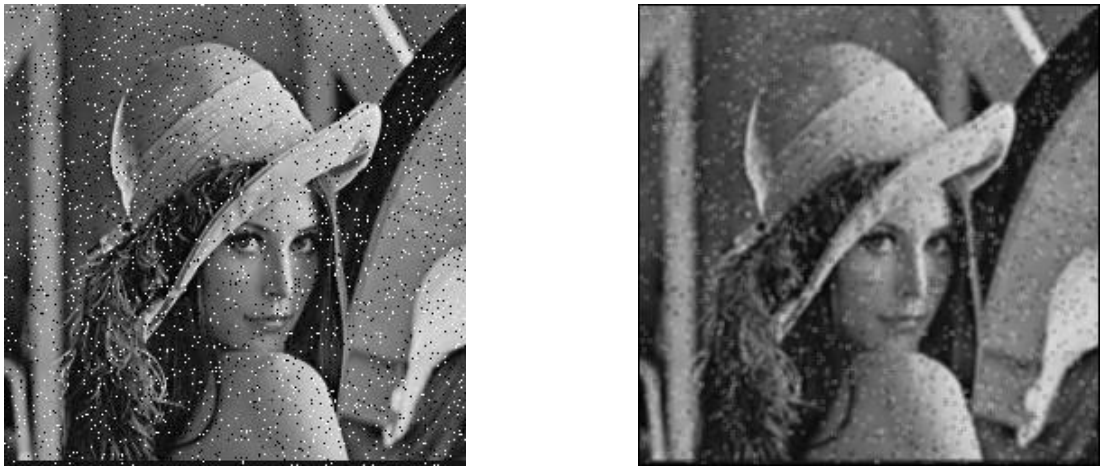


图 1: 椒盐噪声图像与均值滤波处理后的图像

4. 实验总结

本次实验通过对均值滤波器的一步步优化，体验到了 Cuda 编程的魅力：在于增强线程与数据之间的联系，使得它们以一种合理的方式进行操作，从而达到优化算法的目的。同时通过共享内存和纹理存储的使用，充分感受到了这些硬件在 gpu 中扮演的重要角色，并且了解到这些硬件的布局以及使用，将我们的并行思维不再局限于并行地执行循环语句。