基于双边滤波算法的图像去噪
CUDA 优化

徐雅婷
18281221

2020 年 12 月 20 日

摘要

从数字图像处理的早期开始，一个常见的问题就是如何从图像中去除噪声，这是平滑滤波器的最常见用途之一。平滑滤波器是使图像平滑或模糊的数字滤波器，通常会消除图像中的噪点细节，但也会产生使图像的特征模糊的效果，这可能对许多图像处理或计算机视觉任务有害。已经存在一些更先进的技术来解决这个问题，其中之一是双边滤波器。双边滤波器扩展了典型平滑滤波器，具有边缘检测特性，以避免图像边缘模糊。这样可以使图像形成平滑的表面，而不会模糊其主要细节。

双边滤波在计算上是昂贵的，并且具有使优化复杂化的非线性分量。当在中央处理单元（CPU）上顺序运行时，这会花费很多时间，并且双边滤波算法的快速逼近范围很广，使得其算法优化极其困难。本次实验不关注于优化算法本身，而是通过在 GPU 上大规模并行运行常规形式双边滤波算法来加速计算。

在进行图像处理时，由于计算量大，常常无法到达实时的效果，因此需利用 GPU 处理，使用 CUDA 进行优化。尤其是图像滤波这种，(1) 并行度高，线程间耦合度低，每个像素的处理并不相互影响；(2) 像素传输量小，计算量大；特别适合 CUDA 进行计算。

1 问题描述

数字图像由像素组成，这些像素具有自己的数值，这些数值确定像素的视觉属性。这些值代表什么取决于图片中使用的颜色格式。通常，灰度图像中的像素仅包含一个确定其强度的 8-bit 值。该值的范围是 0 到 255，其中 0 通常是纯黑色，而 255 是纯白色。通过对图像像素强度值进行处理，能实现图像平滑，在下面的内容中我将层次递进逐步导出双边滤波。

1.1 均值滤波

均值滤波指的是用周围区域的平均值代替每个像素。设 I 为输入图像，设 O 为输出图像。然后，I（x，y）是水平位置 x 和垂直位置 y 的像素的强度值，O（x，y）是输出图像中对应位置的像素的强度值。可得：

$$O(x,y) = \frac{1}{n^2} \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} I(x+i, y+j)$$

这是像素（x, y）的边长为 n 的方阵区域内像素的平均值，其中 n^2 是矩阵中项的总数。用于计算平均值的该区域的大小对结果有明显的影响。用于计算的该区域称为 kernel，或 convolution matrix，或 mask，均值滤波中，kernel 的权重均为 1。增加 kernel 的大小意味着图像会变得更平滑、模糊。迭代地运行均值滤波会逼近高斯滤波的图像处理效果。

1.2 高斯滤波

高斯滤波类似于均值滤波，但是 convolution matrix 中的权重根据高斯函数加权，也称为正态分布：

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

在 2D 图像中，使用以下二维高斯函数计算 convolution matrix 的权重，该函数是两个一维高斯函数的乘积：

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

由均值滤波公式易得高斯滤波公式：

$$O(x,y) = \frac{1}{k} \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G(i,j) \times I(x+i, y+j)$$

其中：

$$k = \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G(i,j)$$

由于二维高斯函数具有旋转对称性及可分离性，对于给定的高斯核直径和 sigmaSpace，所有像素的高斯核都是一样的，一次在计算 2D 图像高斯滤波权重时，只需计算一遍一维高斯函数，这一部分将不会放在 GPU 上运行。这样可以节省大量的处理时间。使用高斯滤波的一个问题是它会平滑边缘，使图像模糊，因为它在滤波过程中只关注了位置信息，这可能导致图像信息被覆盖。双边滤波在高斯滤波器和其他类似的平滑滤波的这一方面进行了改进。

1.3 双边滤波

在高斯滤波器中，空间距离较大处的像素的权重小于距离较小处的像素的权重。在双边滤波中，此原理也适用，同时，在高斯滤波的基础上加入了像素值权重项，也就是说既要考虑距离因素，也要考虑像素值差异的影响，像素值越相近，权重越大。由高斯滤波公式可得双边滤波公式：

$$O(x,y) = \frac{1}{k(x,y)} \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G_s(i,j) \times G_r(I(x,y) - I(x+i, y+j)) \times I(x+i, y+j)$$

其中：

$$k(x,y) = \sum_{i=-n/2}^{n/2} \sum_{j=-n/2}^{n/2} G_s(i,j) \times G_r(I(x,y) - I(x+i, y+j))$$

这里的 G_s 是高斯空间距离权重（即高斯滤波权重）；而 G_r 是一维高斯函数，用于计算像素差异权重，以两个像素之间的色差作为输入，在灰度图像中，这仅仅是强度值之间的差异。RGB 图像之间的差异则需要更复杂地计算。本次实验中为简化运算，只考虑对灰度图像进行双边滤波。G_r 的结果将根据比较的像素而有所不同, 为每个允许的色差预先计算权重, 这对于许多应用来说是不切实际的，并且可能需要降低颜色分辨率以避免大的开销。G_r 在整个图像上都是动态的另一个结果是归一化因子 k 不是常数，必须根据公式为每个像素分别计算。这也意味着双边滤波是不可分离的，使得实现效率的并行化变得更加必要。

2 平台描述

2.1 CUDA 概述

CUDA 编程模型是一个异构模型，需要 CPU 和 GPU 协同工作。在 CUDA 中，host 和 device 是两个重要的概念，我们用 host 指代 CPU 及其内存，而用 device 指代 GPU 及其内存。CUDA 程序中既包含 host 程

序，又包含 device 程序，它们分别在 CPU 和 GPU 上运行。同时，host 与 device 之间可以进行通信，这样它们之间可以进行数据拷贝。典型的 CUDA 程序的执行流程如下：

(1) 分配 host 内存，并进行数据初始化

(2) 分配 device 内存，并从 host 将数据拷贝到 device 上

(3) 调用 CUDA 的核函数在 device 上完成指定的运算

(4) 将 device 上的运算结果拷贝到 host 上

(5) 释放 device 和 host 上分配的内存

上面流程中最重要的一个过程是调用 CUDA 的核函数来执行并行计算，kernel 是 CUDA 中一个重要的概念，kernel 是在 device 上线程中并行执行的函数，核函数用__global__符号声明，在调用时需要用 $<<<grid, block>>>$ 来指定 kernel 要执行的线程数量，在 CUDA 中，每一个线程都要执行核函数，并且每个线程会分配一个唯一的线程号 thread ID，这个 ID 值可以通过核函数的内置变量 threadIdx 来获得。

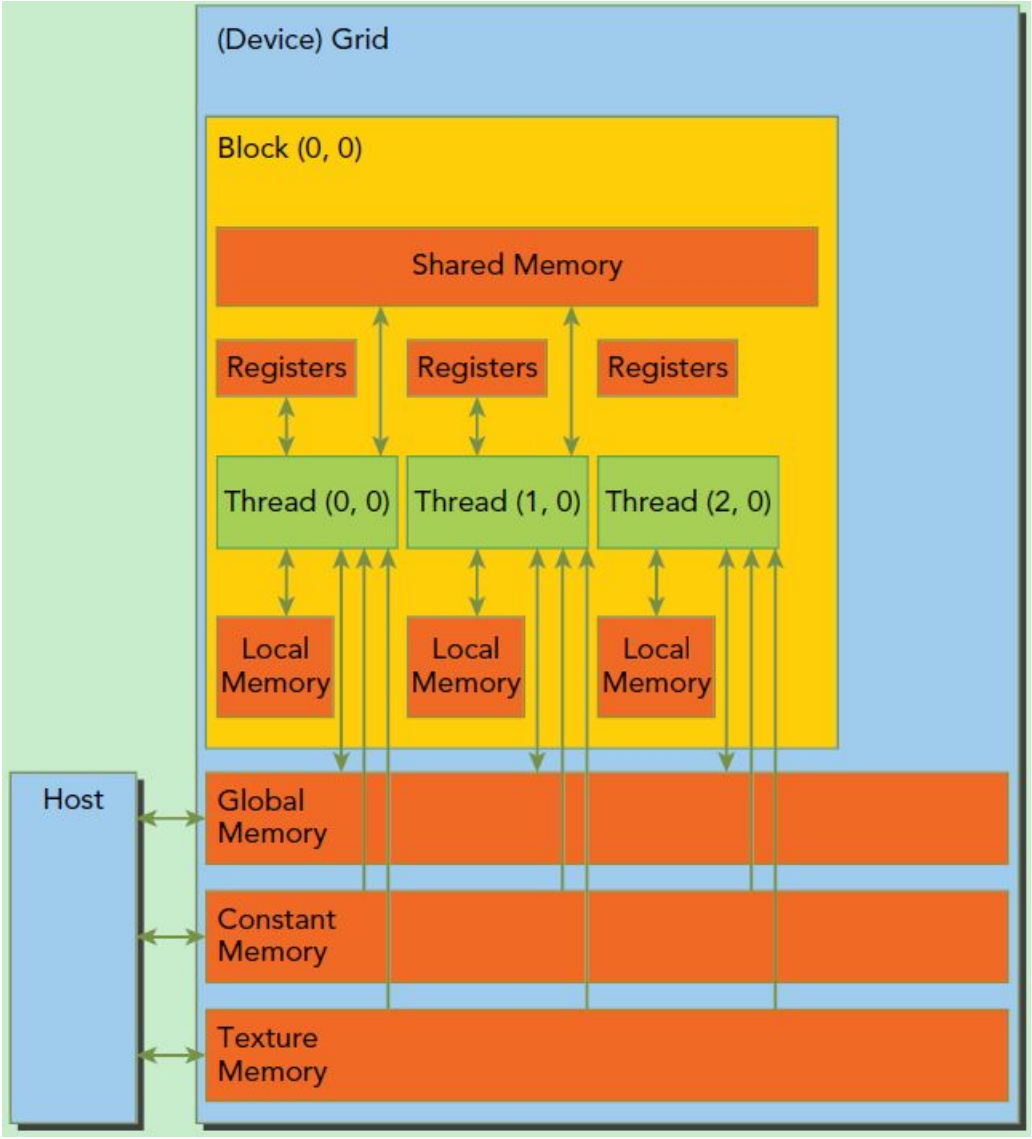
由于 GPU 实际上是异构模型，所以需要区分 host 和 device 上的代码，在 CUDA 中是通过函数类型限定词开区别 host 和 device 上的函数，主要的三个函数类型限定词如下：

- __global__：在 device 上执行，从 host 中调用（一些特定的 GPU 也可以从 device 上调用），返回类型必须是 void，不支持可变参数参数，不能成为类成员函数。注意用__global__定义的 kernel 是异步的，这意味着 host 不会等待 kernel 执行完就执行下一步
- __device__：在 device 上执行，单仅可以从 device 中调用
- __host__：在 host 上执行，仅可以从 host 上调用，一般省略不写

2.2 kernel 的线程层次结构

要深刻理解 kernel，必须要对 kernel 的线程层次结构有一个清晰的认识。首先 GPU 上很多并行化的轻量级线程。kernel 在 device 上执行时实际上是启动很多线程，一个 kernel 所启动的所有线程称为一个网格（grid），同一个网格上的线程共享相同的全局内存空间，grid 是线程结构的第一层次，而网格又可以分为很多线程块（block），一个线程块里面包含很多线程，这是第二个层次。

2.3 CUDA 的内存模型



如图所示。可以看到，每个线程有自己的私有本地内存（Local Memory），而每个线程块有包含共享内存（Shared Memory），可以被线程块中所有线程共享，其生命周期与线程块一致。此外，所有的线程都可以访问全局内存（Global Memory）。还可以访问一些只读内存块：常量内存（Constant Memory）和纹理内存（Texture Memory）。

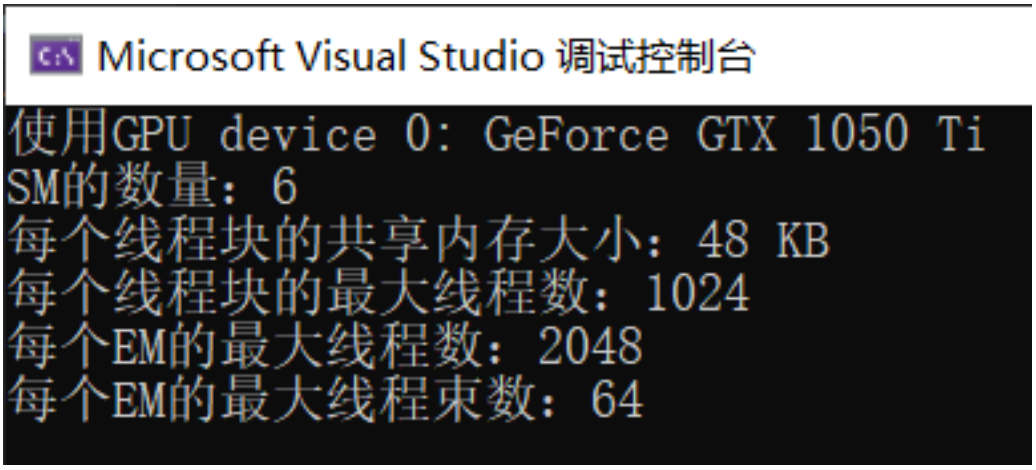
2.4 GPU 硬件的简单描述

上面说到了 kernel 的线程组织层次，那么一个 kernel 实际上会启动很多线程，这些线程是逻辑上并行的，但是在物理层却并不一定。这其实和 CPU 的多线程有类似之处，多线程如果没有多核支持，在物理层也是无法实现并行的。但是好在 GPU 存在很多 CUDA 核心，充分利用 CUDA 核心可以充分发挥 GPU 的并行计算能力。

GPU 硬件的一个核心组件是 SM，SM 是英文名是 Streaming Multiprocessor，翻译过来就是流式多处理器。SM 的核心组件包括 CUDA 核心，共享内存，寄存器等，SM 可以并发地执行数百个线程，并发能力就取决于 SM 所拥有的资源数。当一个 kernel 被执行时，它的 gird 中的线程块被分配到 SM 上，一个线程块只能在一个 SM 上被调度。SM 一般可以调度多个线程块，这要看 SM 本身的能力。那么有可能一个 kernel 的各个线程块被分配多个 SM，所以 grid 只是逻辑层，而 SM 才是执行的物理层。SM 采用的是 SIMT (Single-Instruction,

Multiple-Thread，单指令多线程) 架构，基本的执行单元是线程束（wraps)，线程束包含 32 个线程，这些线程同时执行相同的指令，但是每个线程都包含自己的指令地址计数器和寄存器状态，也有自己独立的执行路径。所以尽管线程束中的线程同时从同一程序地址执行，但是可能具有不同的行为，比如遇到了分支结构，一些线程可能进入这个分支，但是另外一些有可能不执行，它们只能死等，因为 GPU 规定线程束中所有线程在同一周期执行相同的指令，线程束分化会导致性能下降。当线程块被划分到某个 SM 上时，它将进一步划分为多个线程束，因为这才是 SM 的基本执行单元，但是一个 SM 同时并发的线程束数是有限的。这是因为资源限制，SM 要为每个线程块分配共享内存，而也要为每个线程束中的线程分配独立的寄存器。所以 SM 的配置会影响其所支持的线程块和线程束并发数量。总之，就是网格和线程块只是逻辑划分，一个 kernel 的所有线程其实在物理层是不一定同时并发的。所以 kernel 的 grid 和 block 的配置不同，性能会出现差异，这点是要特别注意的。还有，由于 SM 的基本执行单元是包含 32 个线程的线程束，所以 block 大小一般要设置为 32 的倍数。

在进行 CUDA 编程前，先检查一下自己的 GPU 的硬件配置，这样才可以有的放矢：



3 程序代码

kernel.cu

```
#include <iostream>
#include <algorithm>
#include <ctime>
#include <opencv2/opencv.hpp>
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <cuda.h>
#include <device_functions.h>
#define M_PI           3.14159265358979323846


using namespace std;
using namespace cv;
//一维高斯kernel数组。使用常量存储器，位于显存，拥有缓存加速。空间较小（64K），
    保存频繁访问的只读数据，
```

```
__constant__ float cGaussian[64];
//声明纹理参照系，以全局变量形式出现
texture<unsigned char, 2, cudaReadModeElementType> inTexture;


//计算一维高斯距离权重，二维高斯权重可由一维高斯权重做积得到
void updateGaussian(int r, double sd)
{
        float fGaussian[64];
        for (int i = 0; i < 2 * r + 1; i++)
        {
                float x = i - r;
                fGaussian[i] = 1 / (sqrt(2 * M_PI) * sd) * expf(-(x * x) / (2 *
                    sd * sd));
        }
        //存入常量存储器中
        cudaMemcpyToSymbol(cGaussian, fGaussian, sizeof(float) * (2 * r + 1));
}

// 一维高斯函数，计算像素差异权重
__device__ inline double gaussian(float x, double sigma)
{
        return 1 / (sqrt(2 * M_PI) * sigma) * __expf(-(powf(x, 2)) / (2 * powf(
            sigma, 2)));
}

__global__ void gpuCalculation(unsigned char* input, unsigned char* output,
        int width, int height,
        int r,double sigmaColor)
{
        int txIndex = blockIdx.x * blockDim.x + threadIdx.x;
        int tyIndex = blockIdx.y * blockDim.y + threadIdx.y;

        if ((txIndex < width) && (tyIndex < height))
        {
                double iFiltered = 0;
                double k = 0;
                //纹理拾取，得到要计算的中心像素点
                unsigned char centrePx = tex2D(inTexture, txIndex, tyIndex);
                //进行卷积运算
                for (int dy = -r; dy <= r; dy++) {
                        for (int dx = -r; dx <= r; dx++) {
                                //得到kernel区域内另一像素点
                                unsigned char currPx = tex2D(inTexture, txIndex
```

```
                                    + dx, tyIndex + dy);
                                // Weight = 1D Gaussian(x_axis) * 1D Gaussian(
                                    y_axis) * Gaussian(Color difference)
                                double w = (cGaussian[dy + r] * cGaussian[dx + r
                                    ]) * gaussian(centrePx - currPx, sigmaColor);
                                iFiltered += w * currPx;
                                k += w;
                            }
                        }
                        output[tyIndex * width + txIndex] = iFiltered / k;
                }
}

void MyBilateralFilter(const Mat& input, Mat& output, int r, double sigmaColor,
    double sigmaSpace)
{
    //GPU计时事件
        cudaEvent_t start, stop;
        float time;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);

        //计算图片大小
        int gray_size = input.step * input.rows;

        //在device上开辟2维数据空间保存输入输出数据
        unsigned char* d_input = NULL;
        unsigned char* d_output;

        updateGaussian(r, sigmaSpace);

        //分配device内存
        cudaMalloc<unsigned char>(&d_output, gray_size);

        //使用纹理内存存储图像数据，提升性能
        size_t pitch;
        //描述获取纹理时返回值的格式
        cudaChannelFormatDesc desc = cudaCreateChannelDesc<unsigned char>();
        //分配device内存，使用cudaMallocPitch()建立内存空间,以保证段对齐
        cudaMallocPitch(&d_input, &pitch, sizeof(unsigned char) * input.step,
            input.rows);
        //从host将数据拷贝到device上
        cudaMemcpy2D(d_input, pitch, input.ptr(), sizeof(unsigned char) * input.
            step, sizeof(unsigned char) * input.step, input.rows,
```

```
            cudaMemcpyHostToDevice);
        //将texture reference绑定到一个CUDA数组d_input
        cudaBindTexture2D(0, inTexture, d_input, desc, input.step, input.rows,
            pitch);

        int imgHeight = input.rows;
        int imgWidth = input.cols;
        dim3 block(16, 16);
        dim3 grid((imgWidth + block.x - 1) / block.x, (imgHeight + block.y - 1)
            / block.y);

        //在GPU上进行计算
        cudaEventRecord(start, 0);
        gpuCalculation <<< grid, block >>> (d_input, d_output, input.cols, input
            .rows, r, sigmaColor);
        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);

        //将device上的运算结果拷贝到host上
        cudaMemcpy(output.ptr(), d_output, gray_size, cudaMemcpyDeviceToHost);

        //释放device上分配的内存
        cudaFree(d_input);
        cudaFree(d_output);

        // Calculate and print kernel run time
        cudaEventElapsedTime(&time, start, stop);
        printf("Time for the GPU: %f ms\n", time);
}
```

main.cpp

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace std;
using namespace cv;

void MyBilateralFilter(const Mat& input, Mat& output, int r, double sI, double
    sS);

int main() {
        //高斯核直径
```

```
    int d = 9;
    double sigmaColor = 75.0, sigmaSpace = 75.0;
    //将原始图像转化为灰度图像再打开
    Mat srcImg = imread("1.jpg", IMREAD_GRAYSCALE);
    //分配host内存
    Mat dstImg(srcImg.rows, srcImg.cols, CV_8UC1);
    Mat dstImgCV;

    //在GPU上运行并计时
    MyBilateralFilter(srcImg, dstImg, d/2, sigmaColor, sigmaSpace);

    //使用OpenCV bilateral filter在cpu上运行并计时
    clock_t start_s = clock();
    bilateralFilter(srcImg, dstImgCV, d, sigmaColor, sigmaSpace);
    clock_t stop_s = clock();
    cout << "Time for the CPU: " << (stop_s - start_s) / double(
        CLOCKS_PER_SEC) * 1000 << " ms" << endl;
    //展示图片
    imshow("原图", srcImg);
    imshow("GPU加速双边滤波", dstImg);
    imshow("CPU双边滤波", dstImgCV);
    cv::waitKey();
}
```

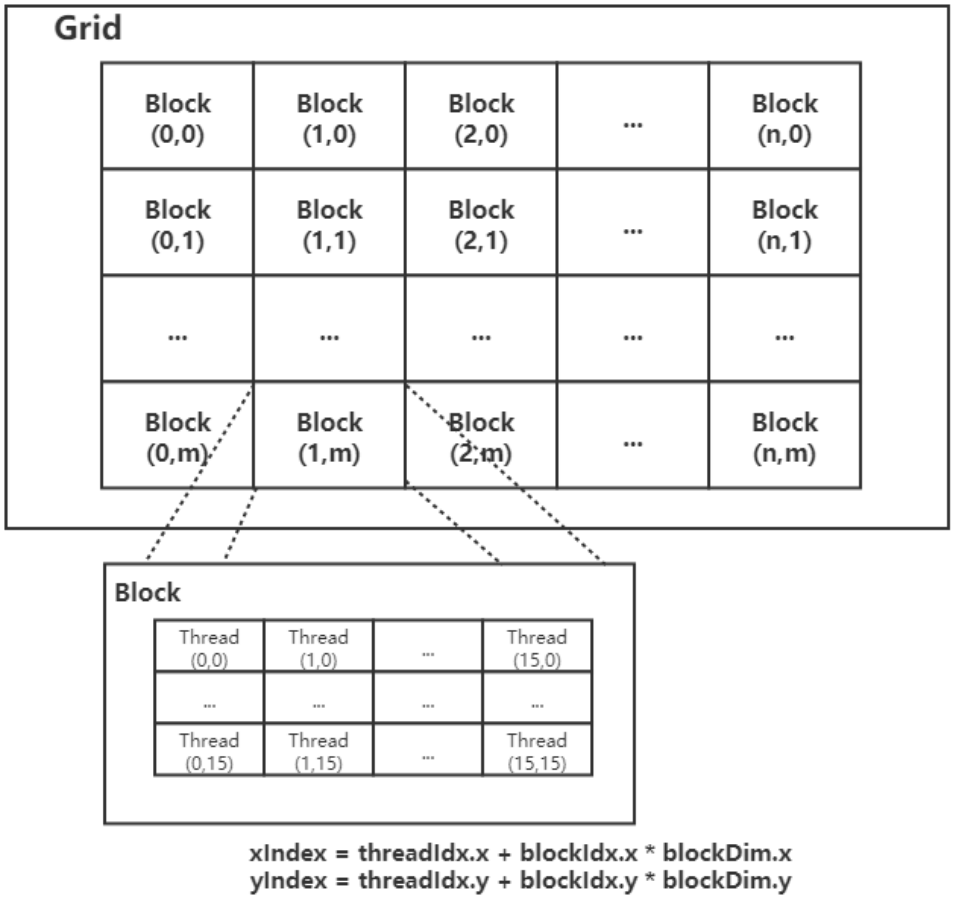
4 关键并行代码分析

4.1 kernel 的线程层次结构

这是一个 gird 和 block 均为 2-dim 的线程组织。grid 和 block 都是定义为 dim3 类型的变量。

```
    int imgHeight = input.rows;
    int imgWidth = input.cols;
    dim3 block(16, 16);
    dim3 grid((imgWidth + block.x - 1) / block.x, (imgHeight + block.y - 1)
        / block.y);
```

这里图片的像素数组大小为 imgHeight × imgWidth，则需要的 thread 数为 imgHeight × imgWidth，block 大小为（16 ，16），（则需要 $\frac{imgsize}{blocksize}$ 个 block），故 grid 大小为 $\frac{imgWidth+block.x-1}{block.x}, \frac{imgHeight+block.y-1}{block.y}$（此处处理了不能整除的情况，确保 grid 里的 thread 能 cover 给定图片所有的像素点，有一些线程是全程不工作的），kernel 的线程层级结构如下图所示：



4.2 使用 Constant Memory

Constant Memory 是 64 KB 的 device 内存，可以保存只读数据。使用 cudaMemcpyToSymbol 函数从 host 将其写入。Constant Memory 的优点是，如果一个 warp 中的所有线程都尝试访问相同的地址，则仅生成一个请求，然后将数据广播到其他线程。Constant Memory 的数据还将缓存起来，因此对相同地址的连续读操作不会产生额外的内存通信量。因为一维高斯距离权重在处理每个像素点时都会被多次计算，故可将其预计算后存入 Constant Memory：

```
__constant__ float cGaussian[64];
```

4.3 高斯函数计算

由于给定高斯核半径 r 和 sigmaSpace，在调用核函数之前即可在 host 上预先计算出一维高斯权重值，并存入常量存储器 cGaussian 中，以供频繁访问，实现加速。（该计算只进行一次，后续过程中需重复访问计算结果，故不在核函数中进行计算。）

```
__constant__ float cGaussian[64];
//计算一维高斯距离权重，二维高斯权重可由一维高斯权重做积得到
void updateGaussian(int r, double sd)
{
```

```
        float fGaussian[64];
        for (int i = 0; i < 2 * r + 1; i++)
        {
                float x = i - r;
                fGaussian[i] = 1 / (sqrt(2 * M_PI) * sd) * expf(-(x * x) / (2 *
                    sd * sd));
        }
        //存入常量存储器中
        cudaMemcpyToSymbol(cGaussian, fGaussian, sizeof(float) * (2 * r + 1));
}
```

上述代码块实现公式

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

在 device 上调用一维高斯函数，用于计算像素差异权重，以两个像素之间的色差作为输入：

```
__device__ inline double gaussian(float x, double sigma)
{
        return 1 / (sqrt(2 * M_PI) * sigma) * __expf(-(powf(x, 2)) / (2 * powf(
            sigma, 2)));
}
```

4.4 使用 Texture Memory

Texture Memory 是一种缓存的 device 内存，通常用于在传统 GPU 任务中保存的纹理和结构。当线程访问彼此相邻的数据时，使用 Texture Memory 将拥有更快的访问速度。CUDA 中的 Texture 用法是通过 texture reference 来处理的，texture reference 是在编译时创建的，需要全局声明。本次实验中 Texture Memory 而不是 Global Memory 访问输入图像的像素。

此外，访问显存时，读取和存储必须对齐。如果没有正确的对齐，读写将被编译器拆分为多次操作，降低访存性能。一维数据使用 cudaMalloc() 开辟 gpu 全局内存空间，多维数据使用 cudaMallocPitch() 建立内存空间，以保证段对齐。cudaMallocPitch 函数分配的内存中，数组的每一行的第一个元素的开始地址都保证是对齐的。因为每行有多少个数据是不确定的，widthofx*sizeof(元素) 不一定是 256 的倍数。故此，为保证数组的每一行的第一个元素的开始地址对齐，cudaMallocPitch 在分配内存时，每行会多分配一些字节，以保证 widthofx*sizeof(元素)+ 多分配的字节是 256 的倍数 (对齐)。

```
//声明纹理参照系，以全局变量形式出现
texture<unsigned char, 2, cudaReadModeElementType> inTexture;
```

```
        //使用纹理内存存储图像数据，提升性能
        size_t pitch;
        //描述获取纹理时返回值的格式
        cudaChannelFormatDesc desc = cudaCreateChannelDesc<unsigned char>();
        //分配device内存，，使用cudaMallocPitch()建立内存空间,以保证段对齐
        cudaMallocPitch(&d_input, &pitch, sizeof(unsigned char) * input.step,
            input.rows);
```

```
        //从host将数据拷贝到device上
        cudaMemcpy2D(d_input, pitch, input.ptr(), sizeof(unsigned char) * input.
            step, sizeof(unsigned char) * input.step, input.rows,
            cudaMemcpyHostToDevice);
        //将texture reference绑定到一个CUDA数组d_input
        cudaBindTexture2D(0, inTexture, d_input, desc, input.step, input.rows,
            pitch);
```

4.5 核函数分析

```
__global__ void gpuCalculation(unsigned char* input, unsigned char* output,
        int width, int height,
        int r,double sigmaColor)
{
        int txIndex = blockIdx.x * blockDim.x + threadIdx.x;
        int tyIndex = blockIdx.y * blockDim.y + threadIdx.y;

        if ((txIndex < width) && (tyIndex < height))
        {
                double iFiltered = 0;
                double k = 0;
                //纹理拾取，得到要计算的中心像素点
                unsigned char centrePx = tex2D(inTexture, txIndex, tyIndex);
                //进行卷积运算
                for (int dy = -r; dy <= r; dy++) {
                        for (int dx = -r; dx <= r; dx++) {
                                //得到kernel区域内另一像素点
                                unsigned char currPx = tex2D(inTexture, txIndex
                                    + dx, tyIndex + dy);
                                // Weight = 1D Gaussian(x_axis) * 1D Gaussian(
                                    y_axis) * Gaussian(Color difference)
                                double w = (cGaussian[dy + r] * cGaussian[dx + r
                                    ]) * gaussian(centrePx - currPx, sigmaColor);
                                iFiltered += w * currPx;
                                k += w;
                        }
                }
                output[tyIndex * width + txIndex] = iFiltered / k;
        }
}
```

首先获取线程的全局索引；然后每个线程处理其对应的像素点，并将结果存入 output。

处理过程使用双边滤波公式：

$$O(x,y)=\frac{1}{k(x,y)}\sum_{i=-n/2}^{n/2}\sum_{j=-n/2}^{n/2}G_s(i,j)\times G_r(I(x,y)-I(x+i,y+j))\times I(x+i,y+j)$$

$$k(x,y)=\sum_{i=-n/2}^{n/2}\sum_{j=-n/2}^{n/2}G_s(i,j)\times G_r(I(x,y)-I(x+i,y+j))$$

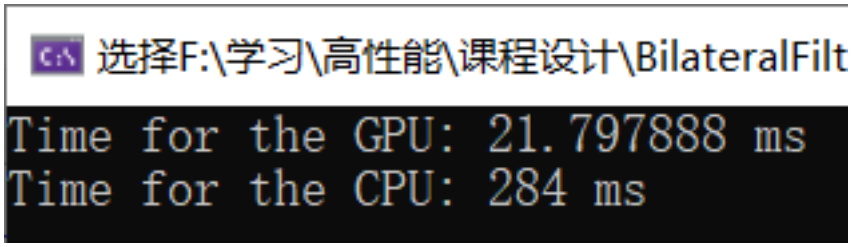
其中 $G_s(i,j)$ 由两个预计算的一维高斯函数相乘得到。本次实验中为简化运算，只考虑对灰度图像进行双边滤波，故 G_r 的输入是像素强度值之间的差异。可见，双边滤波具有使优化复杂化的非线性分量，使用多线程并行计算会极大提高运算速度。

5 运行结果截屏

5.1 效果展示

下图是当高斯核直径 d=9，sigmaColor = 15.0, sigmaSpace = 15.0 时在大小为 258*172 的测试图片上的运行结果：

最上面的图片是 Input Image, 左下角图片是在 GPU 上运行双边滤波的结果，右下角是在 CPU 上运行双边滤波的结果



5.2 当直径 d、sigmaSpace 不变，增大 sigmaColor 时

从左到右 sigmaColor 分别为 15.0，35.0，75.0

5.3 当直径 d、sigmaColor 不变，增大 sigmaSpace 时

从左到右 sigmaColor 分别为 15.0，35.0，75.0

5.4 当 sigmaSpace、sigmaColor 不变，增大直径 d 时

从左到右 d 分别为 9，12，24

6 运行结果分析

6.1 运行环境

- Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz

- 16.00 GB of RAM
- Windows 10 Pro 64-bit
- NVIDIA GeForce GTX 1050 Ti

6.2 测试结果

根据图片效果，双边滤波起到了“磨皮”的去噪效果，较为完整地保留了图片的重要细节。由上控制台输出可知，当高斯核直径 d=9，sigmaColor = 15.0, sigmaSpace = 15.0 时在大小为 258*172 的测试图片上，该算法在 GPU 上 CUDA 加速运行比在 CPU 上运行速度快了十多倍！

改变 kernel 直径 d 的大小，观察到运行时间如下：

直径 d	GPU	CPU
2	2.5ms	271ms
4	6.8ms	274ms
8	21.7ms	284ms
12	110.8ms	309ms

随着 d 增大，CUDA 加速效果逐渐变差，与 GPU 性能限制有关。

6.3 分析

(1) 首先，两个 sigma 值为 kernel 的方差，方差越大，说明权重差别越小，因此表示不强调这一因素的影响，反之，则表示更强调这一因素导致的权重的不均衡。因此，两个方面的某个的 sigma 相对变小表示这一方面相对较重要，得到强调。如 sigmaSpace 变小，表示更多采用近邻的值作平滑，说明图像的空间信息更重要，即相近相似。如 sigmaColor 变小，表示和自己同一类的条件变得苛刻，从而强调值域的相似性。

其次，sigmaSpace 表示的是空域的平滑，因此对于没有边缘的，变化慢的部分更适合；sigmaColor 表示值域的差别，因此强调这一差别，即减小 sigmaColor 可以突出边缘。sigmaSpace 变大，图像每个区域的权重基本都源于值域滤波的权重，因此对于空间邻域信息不是很敏感；sigmaColor 变大，则不太考虑值域，权重多来自于空间距离，因此近似于普通的高斯滤波，图像的保边性能下降。因此如果想更多的去除平滑区域的噪声，应该提高 sigmaSpace ，如果想保持边缘，则应该减小 sigmaColor 。

sigmaColor 越大边缘越模糊，sigmaSpace、直径 d 越大平坦区域越模糊

(2) 在 GPU 算力允许的情况下，CUDA 极大地加速了双边滤波的计算