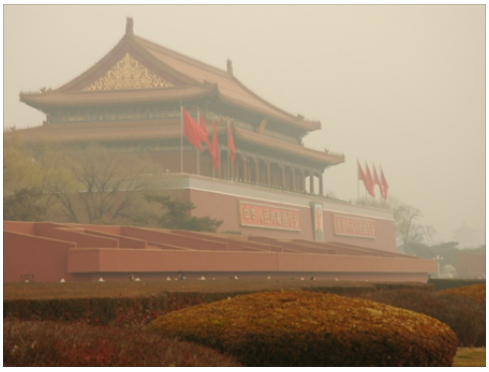
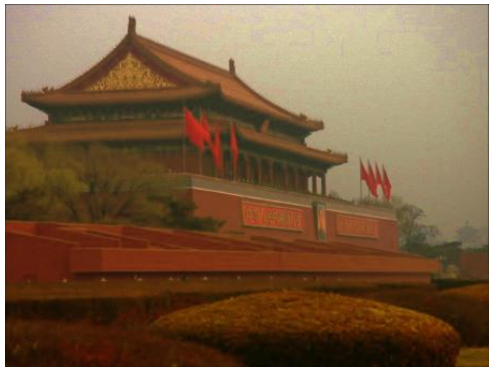
GPU并行计算实验： 图形去雾的并行计算加速

程序主要功能说明

- 使用GPU对图片进行并行处理
- 对含有雾气的图片实现去雾
- 功能示例

待去雾图片	去雾结果
	

暗通道先验去雾算法

暗通道先验去雾算法于2009年首次提出，该算法的提出论文获得了2009年CVPR最佳论文。该算法的核心时提出了暗通道先验理论，通过该理论可以从有雾图像中计算得到大气光强度和大气透射率，进而在利用大气散射模型，求得无雾图像。基于该方法求的的无雾图像，效果好并且真实，而且该方法的鲁棒性强，可以对多种环境下的有雾图像进行去雾。

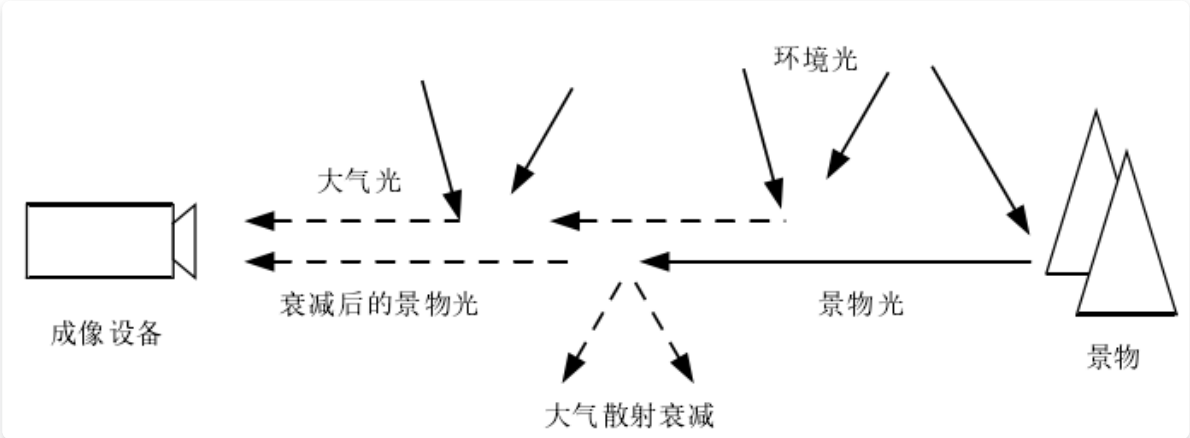
大气散射模型

成像设备对景物的成像过程是一个复杂的过程，主要包括两个内容：景物光（景物反射的光）的衰减和大气散射光。

景物光的衰减是只景物光在到达成像设备的过程中，会经过大气中的各种杂质的反射散射等，衰减了景物光的强度，使得成像设备接收到的景物光相比原始景物光，会变弱很多。大气总不是纯净的，其中包含了很多杂质，杂质体积也有大有小，尤其是不同天气情况下，大气中的杂质体积更是大小各异。在雾天情况下，大气中存在大量体积较大的颗，对景物光产生了较大的衰减作用，才使得到的图片清晰度降低。

大气散射光部分是指大气中杂质散射的环境光到达成像设备，和景物光一起成像的过程。大气对景物光的衰减作用，使得到达成像设备的景物光强度大大降低。但是最终成像后的团像的总亮度并没有降低，是由大气光的作用的导致。大气光中杂质对光具有反射和散射作用，其不仅可以把景物光散射到其他方向，使有用光强度减弱，也可以把大气中存在的各种光源光散射到成像设备方向，使得成像设备接收到的综合光强度增加。

鉴于以上两个方面，大气中景物成像过程可以描述如下。



大气散射模型

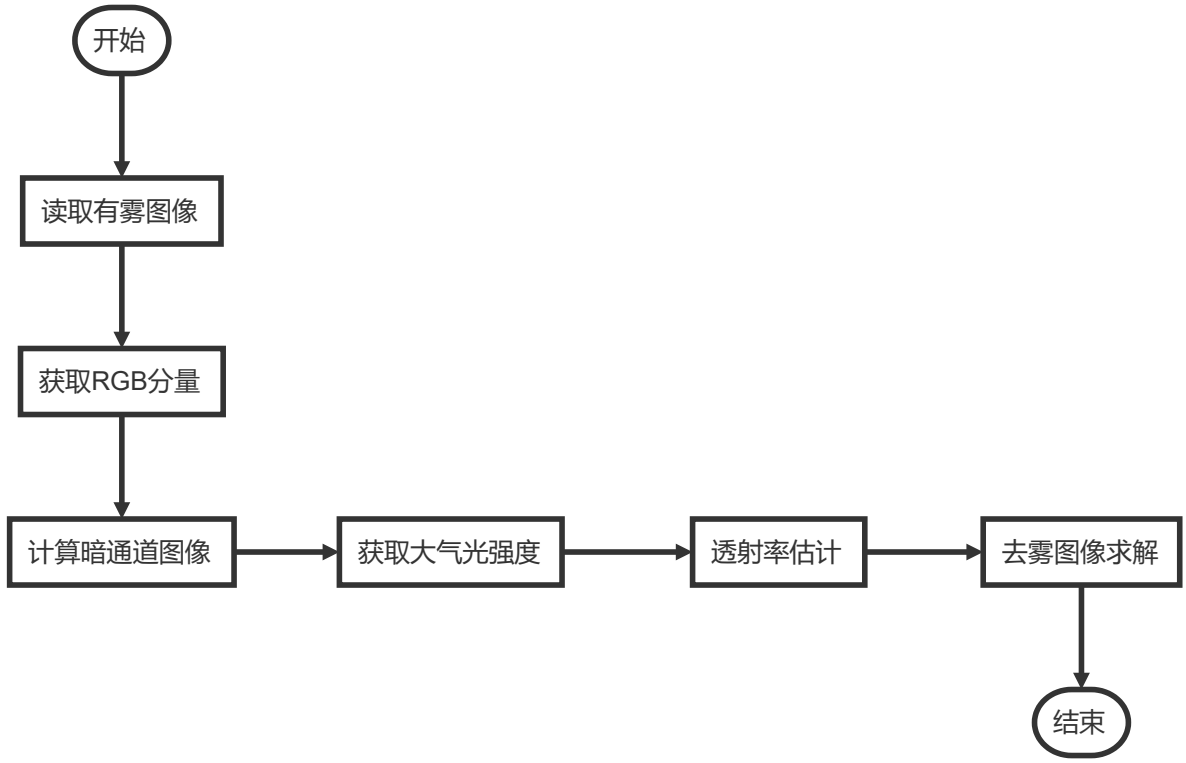
用数学建模方式对其进行建模，经过计算和推导，得到如下描述大气散射模型的公式。

$$I^c(x) = J^c(x)t(x) + A(1 - t(x)) \tag{1}$$

式中，$I^c(x)$为成像设备得到的图像，即有雾图像，$J^c(x)$为原始景物图像，即无雾图像，$t(x)$为大气环境透射率，A为大气光强度，即全局大气光，x为成像图象中的像素点，c为颜色通道（RGB颜色通道）。

暗通道先验去雾算法流程

暗通道先验去雾算法流程如下图所示：



暗通道先验理论

暗通道先验理论是一种统计理论，该理论是在对大量图像进行研究后所得。研究发现，在晴朗天气条件下获得的图像中，除去天空部分外，图像中其他部分都存在共同特点，他们的暗通道值都非常低，趋近于零，很多点的暗通道值甚至直接等于 0。暗通道图像是图像的另一种信息体现形式，是对原始图像中的每个像素点取各个颜色通道中的最小值，然后再进行一定窗口大小的最小值滤波，最终得到的一副图像。用公式表示如下。

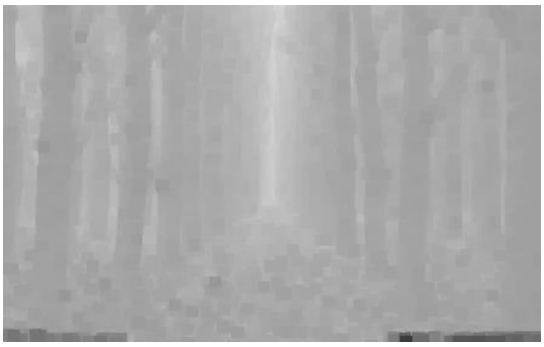
$$J^{dark}(x) = min_{y\in\Omega(x)}(min_{c\in r,g,b}(J^c(y))) \tag{2}$$

其中,$J^{dark}(x)$是处理后得到的暗通道图像，$J^c(y)$是原始图像，$\Omega(x)$是以x为中心的一块图像区域（即滤波窗口区域，x表示暗通道图像中的像素带年，y表示滤波窗口内原始图像中的像素点，c表示颜色通道.在无雾情况下，除去图像中的天空明亮区域，$J^{dark}(x)$的值总是很低并且趋近于0，即:

$$J^{dark}(x) = 0 \tag{3}$$

无雾图片	暗通道图像
	
	
	

暗通道图像是对原始图像每个像素点求各个通道的最小值，然后再进行最小值滤波。可以发现无雾图象中除去天空部分，暗通道图像的每个像素点的值都很小，且趋近于0，整个暗通道图像偏暗。

有雾图片	暗通道
	
	

而对于有雾图像，其暗通道图像计算方法相同,但是暗通道图像整体偏亮，亮度值与雾的浓度线管，因此可以根据暗通道先验进行去雾处理。

获取大气光强度

针对如何获取大气光强度已经有了一个比较成熟的方法，即选定有雾图像中亮度最大的像素点的像素值作为大气光强度，因为 Tan 等人认为雾浓度最大的地方往往是亮度最大的地方。但是暗通道先验去雾算法的作者何凯明博士认为，Tan 等人的方法存在一定问题，当有雾图像中存在特别明亮或者白色的物体，如白色的墙壁或者白色的轿车等，计算得到的大气光强度就会存在很大的误差。因此何凯明博士提出了自己计算大气光强度的方法。

其方法为：在暗通道图像中，首先获取灰度值大小排列在所有像素点中前 0.1%的像素点，再依据这些像素点，遍历其在原始图像中对应的各个点的亮度值，找出亮度最大的点，其亮度值作为大气光强度值。 这种方式得到的大气光强度值，不一定是有雾图像中亮度最大的点，因此可以避免一些明亮物体带来的误差。但是此方法需要大量的比较和排序计算，耗时长，并且效果上也存在很大的局限性。当图像中明亮区域较大，或者在计算暗通道图像时，最小值滤波窗口偏小，也容易使计算得到的大气光强度偏差过大，导致去雾效果不理想。另一方面，此方法计算得到的大气光强度值普遍偏大，从而使得去雾后的图像整体偏暗。

透射率估计

大气光强度是一个全局值，即对于同一幅图片来说，其值是不变的，由此对(1)式变形为：

$$\frac{I^c(x)}{A} = t(x)\frac{J^c(x)}{A} + 1 - t(x) \tag{3}$$

透射率虽然是一个局部值变量，但是在最小值滤波中，同一个滤波窗口内，透射率值可以看作为一个常数。设其值为$\hat{t}(x)$。由（4）式有：

$$min_{y\in\Omega(x)}(min_c(\frac{I^c(y)}{A})) = \hat{t}(x)min_{y\in\Omega(x)}(min_c(\frac{J^c(y)}{A})) + 1 - \hat{t}(x) \tag{4}$$

而由于无雾图像的暗通道图像中每个像素点的值都很低，趋近于 0，于是又：

$$J^{dark}(x) = min_{y\in\Omega(x)}(min_c(J^c(y))) = 0 \tag{5}$$

而大气光强度A又是一个定制，于是有：

$$min_{y\in\Omega(x)}(min_c(\frac{J^c(y)}{A}))=0 \tag{6}$$

根据（4）式，于是有：

$$\hat{t}(x)=1-min_{y\in\Omega(x)}(min_c(\frac{I^c(y)}{A})) \tag{7}$$

然而在日常生活中，即便是晴空万里，大气也不是完全纯净的，其中总会包含很多各种各样的杂质，这些杂质又总会对景物成像产生一定的影响，因此获得的图像，不可能完全无雾。鉴于此，为了让去雾处理后的图像，更加真实，可以人为的增加一个去雾的程度系数$\omega\in[0,1]$，修正后的透射率粗估计公式如下：

$$\hat{t}(x)=1-\omega*min_{y\in\Omega(x)}(min_c(\frac{I^c(y)}{A})) \tag{8}$$

ω的值是一个经验值，不是固定不变的，可以根据个人判断或者目标要求，设置其大小，一般设置为 0.95 会使得去雾后的图像最为真实。

去雾图像求解

对（1）式变形后得到：

$$J^c(x)=\frac{I^c(x)-A}{t(x)}+A \tag{9}$$

$J^c(x)$就是最终的去雾图像。为了避免计算得到的透射率值过小，导致$J^c(x)$的值偏大，从而使得整个图像在某些颜色上会比较鲜艳，在实际实现过程中增加一个最小的限定值t_0，一般取值$t_0=0.1$，因此去雾图像求解公式转化为：

$$J^c(x)=\frac{I^c(x)-A}{max(t_0,t(x))}+A \tag{10}$$

GPU并行加速实现图像去雾

显然利用CPU串行程序可以实现，但是在暗通道求解，大气光强度求解等诸多过程中，像素之间的联系并不是很大，因此我们可以将程序改为并行程序。

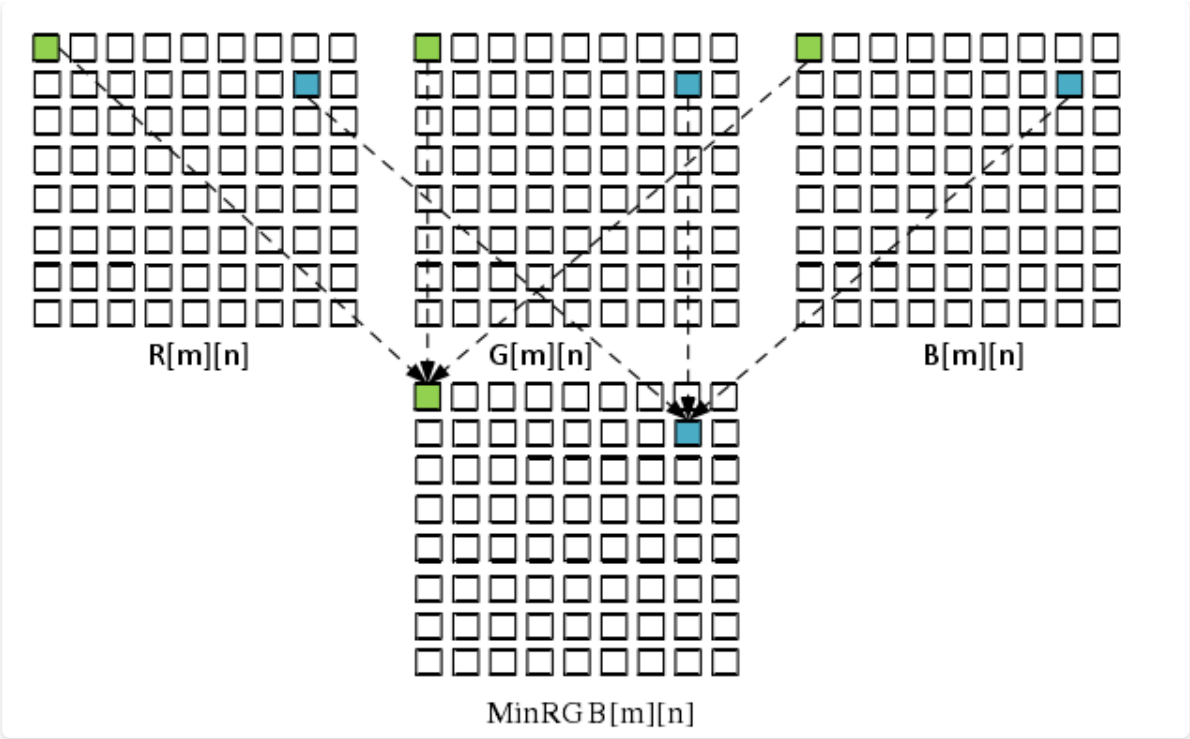
暗通道图像求解

暗通道求解过程主要分为求解RGB三通道各自的最小值和最小值滤波。

求解RGB最小值

在计算原始有雾图像中每个像素点的各个通道最小值时，在 CPU 端一般采用循环遍历的方式对每个点进行计算。这种方式对每个点的计算过程是顺序执行的，只有完成一个点的计算才会移动到下一个点继续进行计算。如计算一副$m\times n$大小的图片，则需要循环移动并计算$m\times n-1$次。

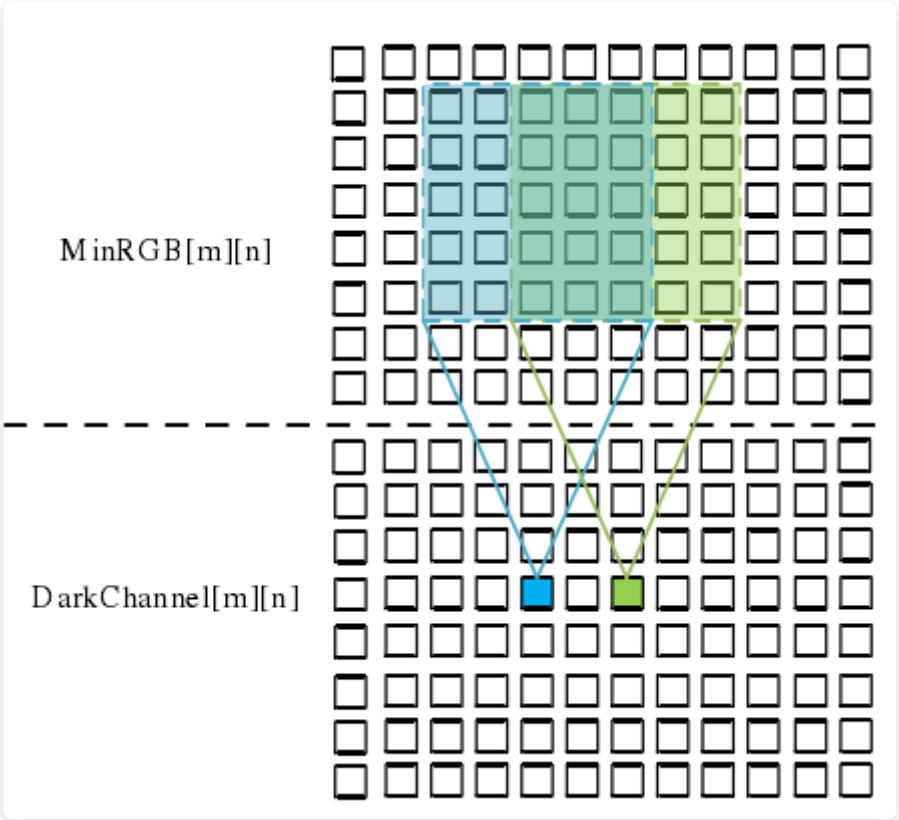
由于对每个点的计算只和当前点的 RGB 各通道值有关，和其他点的各通道值无关，因此可以分配每个处理单元负责不同像素点的计算，并且可以同步进行，以此来实现其计算的并行性。



最小值求解示意

最小值滤波

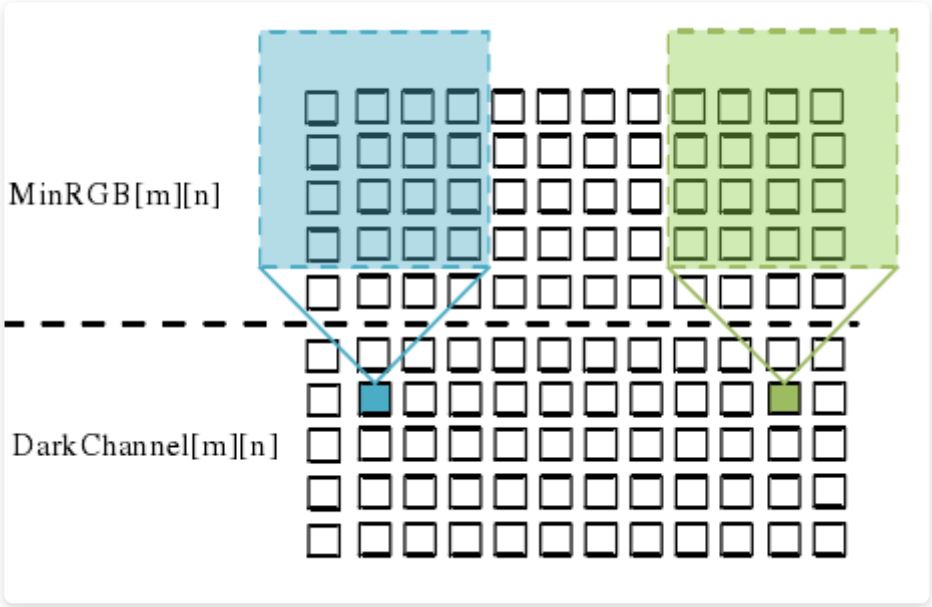
最小值滤波的原理和求解单个像素点各个通道最小值的原理相似，都是通过对多个值进行比较，找到最小值。他们的不同之处在于最小值滤波是对不同像素点灰度值的比较，而求解各通道最小值是对同一个点的不同颜色通道值进行比较。虽然最小值滤波是对多个像素点进行操作，但其依旧具有相互独立性。把需要进行最小值滤波的数据作为公共数据，而每个点的计算只取其所需要的数据，计算结果单独保存，不对公共数据做任何修改，从而不改变和影响其他点所需要的数据，进而可以实现每个点计算的独立性。每个点的最小值滤波的计算可以放在一个单独的处理单元上进行处理，从而实现整个最小值滤波计算的并行计算。



最小值滤波示意

在进行最小值滤波时，相邻点之间的计算数据会有交叉，但在实际计算时，可以把计算结果单独存放，不更改原始数据，这样可以实现不同点之间的独立计算，互不影响。虽然不同点的计算可以实现并行，但是每个点所在窗口中的最小值寻找只能采用 for 循环的形式，无法实现更深层次的并行。

在对窗口中的值进行遍历求解最小值时，会存在边界值不存在的情况，如下图所示，当滤波窗口超出图像范围时，可以选择对不存在的点进行填充，也可以直接忽略，然后按照现有窗口内的数据进行最小值的求解。本次GPU程序实现中，计算时采用的方式是在遍历每个点时，都进行判断，不存在的点则直接跳过，只计算真实存在的点。



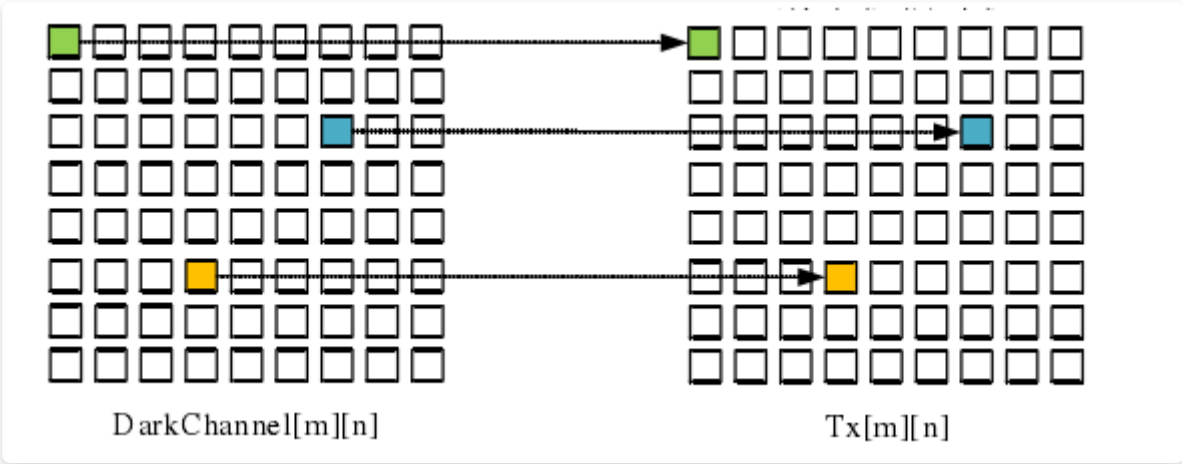
最小值滤波边界示意

透射率估计

由（8）式知道：

$$\hat{t}(x) = 1 - \omega * \frac{DarkChannel(x)}{A} \tag{11}$$

可以看出，透射率的估计值和暗通道图像值存在一一对应的关系，并且每个点的计算也是相互独立的，可以由不同的处理单元处理，因此可以实现并行。



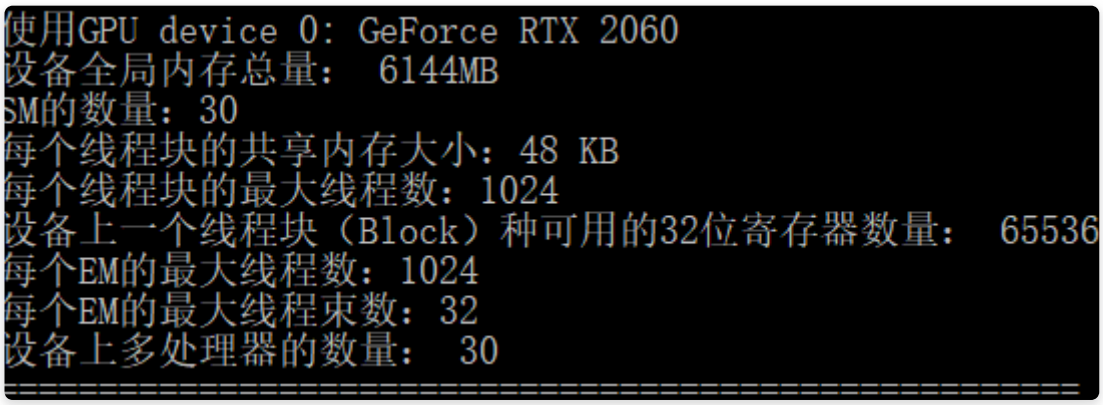
透射率与暗通道值映射关系示意

去雾图像求解

显然，同透射率估计一样，最后的恢复过程也是存在相对应的一一对应关系，故而也可以实现并行处理，这里不再赘述。

GPU实现并行化的图像去雾

GPU配置



GPU配置

CPU和CPU运行比较

- 运行效果上，由于只是串行改为并行，因此求解的效果图一致。
- 运行时间比较：

待去雾图片	去雾结果图片
	

对上面的待去雾图片处理，分别使用CPU串行处理和GPU进行处理。运行时间如下。

主要处理功能	CPU运行时间/ms	GPU运行时间 /ms
minChannel（求解RGB通道最小值)	88	0.177
darkChannel（最小值滤波）	1823	2.767
transmission（透射率估计）	266	0.321
recover（去雾图像恢复）	272	0.339

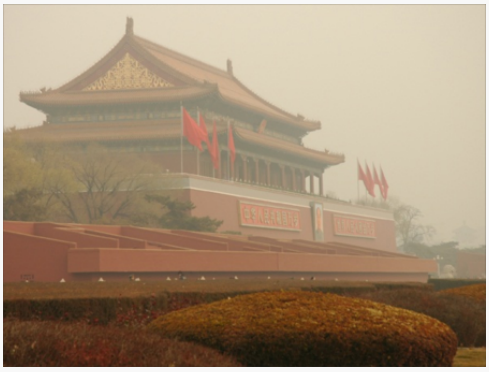
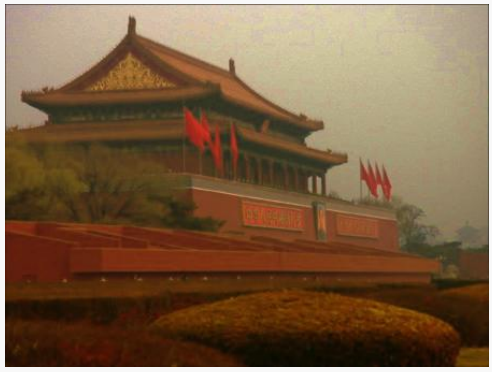
```
minChannel: 88ms
darkChannel: 1823ms
transmission: 266ms
recover: 272ms
```

CPU运行时间

```
minChannel time: 0.177024 ms
darkChannel time: 2.76707 ms
transmission time: 0.321056 ms
recover time: 0.338624 ms
```

GPU运行时间

- 程序处理图片效果展示

去雾图片	去雾结果图片
	
	
	

实验完整代码及功能简要说明

代码包含了CPU和GPU两种实现版本，为此也提供了相应的main函数入口，请根据需求，自行注释暂不运行的main函数。**例如运行CPU版本的程序时，需注释GPU版本的main函数。**

- 自定义的工具类(**util.h,util.cpp**)
    - **util.h**

```
//编译指令
#pragma once
//导入opencv库和iostream处理输入输出
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
//使用命名空间 std和cv
using namespace std;
using namespace cv;
//自定义结构体node和重载操作符(<)，用于后续亮度筛选
struct node {
    int x;
    int y;
    uchar val;
    node() {};
    node(int _x, int _y, int _val) : x(_x), y(_y), val(_val) {};
    bool operator < (const node & b) {
        return val > b.val;
    }
};
//二分法在node结构体数组vector中查找阈值(threshold)的索引
int find(vector<node>& t, uchar threshold);
//求解大气光亮度
Vec3b atmosphic(Mat pic, Mat img);
```

    - **util.cpp**

```
//导入opencv库iostream处理输入输出
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
//自定义util
#include "util.h"
//使用命名空间 std和cv
using namespace std;
using namespace cv;
//二分法在node结构体数组vector中查找阈值(threshold)的索引 实现源码
int find(vector<node>& t, uchar threshold)
{
    int l = 0, r = static_cast<int>(t.size()) - 1;
    while (l < r) {
        int mid = (l + r + 1) >> 1;
        if (t[mid].val >= threshold) l = mid;
        else r = mid - 1;
    }
    return r;
}
//求解大气光亮度 实现源码
Vec3b atmosphic(Mat pic, Mat img)
{
    vector<node> t;
```

```
    for (int i = 0; i < img.rows; i++)
    {
        for (int j = 0; j < img.cols; j++) {
            node p = node(i, j, img.at<uchar>(i, j));
            t.push_back(p);
        }
    }
    sort(t.begin(), t.end());
    double p = 0.001;
    int n = static_cast<int>((t.size()) * p);
    uchar threshold = t[n].val;
    Vec3b A = Vec3b(0, 0, 0);
    int index = find(t, threshold);
    vector<Mat> channels;
    split(pic, channels);
    for (size_t i = 0; i < pic.channels(); i++) {
        Mat pic_t = channels.at(i);
        for (int j = 0; j <= index; j++) {
            A[i] = max(A[i], pic_t.at<uchar>(t[j].x, t[j].y));
        }
    }
    return A;
}
```

- CPU版本的图像去雾代码(**DehazeCPU.cpp**)

```
//导入opencv库iostream处理输入输出
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
//导入时间库求解时间
#include <time.h>
//导入自定义库
#include "util.h"

#define _CRT_SECURE_NO_WARNINGS
//使用命名空间 std和cv
using namespace std;
using namespace cv;
//实现的几个功能函数
Mat minChannel(Mat img);
Mat darkChannel(Mat img);
Mat transmission(Mat img, Vec3b a, float w, float t0);
Mat recover(Mat img, Mat guide, Vec3b A);
Mat deHaze(Mat img);

int main() {
    Mat hazed = imread("extra1.jpg");

    clock_t start_t = clock();
    Mat dehazed = deHaze(hazed);
    clock_t end_t = clock();

    double total_t;
    total_t = (double)(end_t - start_t) / CLOCKS_PER_SEC * 1000;
    cout << "CPU 运行占用的总时间为: " << total_t << " ms" << endl;
```

```
    imshow("hazed image", hazed);
    imshow("Dehazed image", dehazed);
    waitKey(0);
    return 0;
}

Mat minChannel(Mat img)
{
    CV_Assert(!img.empty());
    //开始计时
    double start_t = clock();
    Mat _minChannel = Mat::zeros(img.size(), CV_8UC1);

    for (int i = 0; i < img.rows; ++i)
    {
        for (int j = 0; j < img.cols; ++j)
        {
            _minChannel.at<uchar>(i, j) = img.at<Vec3b>(i, j)[0];
            _minChannel.at<uchar>(i, j) = min(_minChannel.at<uchar>(i,
j), img.at<Vec3b>(i, j)[1]);
            _minChannel.at<uchar>(i, j) = min(_minChannel.at<uchar>(i,
j), img.at<Vec3b>(i, j)[2]);
        }
    }
    //停止计时
    double end_t = clock();
    cout << "minChannel: " << (end_t - start_t) / CLOCKS_PER_SEC * 1000
<< "ms" << endl;
    return _minChannel;
}

Mat darkChannel(Mat img)
{
    CV_Assert(!img.empty());
    //开始计时
    double start_t = clock();
    int r = 7;
    Mat _kernel = getStructuringElement(MORPH_RECT, Size(2 * r + 1, 2 * r
+ 1));
    int m = (_kernel.rows + 1) / 2;
    int n = (_kernel.cols + 1) / 2;
    Mat _darkChannel(img.size(), img.type());
    for (int i = 0; i < img.rows; ++i)
    {
        for (int j = 0; j < img.cols; ++j)
        {
            int xl = max(0, i - m + 1);
            int xr = min(i + m - 1, img.rows);
            int yl = max(0, j - n + 1);
            int yr = min(j + n - 1, img.cols);
            uchar t_inf = 0xff;
            for (int k = xl; k < xr; ++k)
            {
                for (int l = yl; l < yr; ++l)
                {
                    uchar s = img.at<uchar>(k, l);
                    t_inf = t_inf > s ? s : t_inf;
```

```
                }
            }
            _darkChannel.at<uchar>(i, j) = t_inf;
        }
    }
    //停止计时
    double end_t = clock();
    cout << "darkChannel: " << (end_t - start_t) / CLOCKS_PER_SEC * 1000
<< "ms" << endl;
    return _darkChannel;
}

Mat transmission(Mat img, Vec3b a, float w, float t0)
{
    CV_Assert(!img.empty());
    //开始计时
    double start_t = clock();
    vector<Mat> channels;
    split(img, channels);
    Mat t_hat(channels[0].size(), CV_32F);
    for (int i = 0; i < t_hat.rows; i++)
    {
        for (int j = 0; j < t_hat.cols; j++)
        {
            t_hat.at<float>(i, j) = (float)channels[0].at<uchar>(i, j) /
a[0];
            t_hat.at<float>(i, j) = min(t_hat.at<float>(i, j),
(float)channels[1].at<uchar>(i, j) / a[1]);
            t_hat.at<float>(i, j) = min(t_hat.at<float>(i, j),
(float)channels[2].at<uchar>(i, j) / a[2]);
        }
    }
    for (int i = 0; i < t_hat.rows; i++)
    {
        for (int j = 0; j < t_hat.cols; j++)
        {
            t_hat.at<float>(i, j) = max(t0, 1 - w * t_hat.at<float>(i,
j));
        }
    }
    //停止计时
    double end_t = clock();
    cout << "transmission: " << (end_t - start_t) / CLOCKS_PER_SEC * 1000
<< "ms" << endl;
    return t_hat;
}

Mat recover(Mat img, Mat guide, Vec3b A)
{
    CV_Assert(!img.empty());
    //开始计时
    double start_t = clock();
    vector<Mat> channels;
    split(img, channels);
    for (int c = 0; c < channels.size(); c++)
    {
        Mat pic_t = channels.at(c);
        for (int i = 0; i < channels[c].rows; i++)
```

```
        {
            for (int j = 0; j < channels[c].cols; j++)
            {
                pic_t.at<uchar>(i, j) = uchar(min(0xff,
int(uchar((pic_t.at<uchar>(i, j) - A[c]) / guide.at<float>(i, j) +
A[c])));
            }
        }
    }
    Mat _recover;
    merge(channels, _recover);
    //停止计时
    double end_t = clock();
    cout << "recover: " << (end_t - start_t) / CLOCKS_PER_SEC * 1000 <<
"ms" << endl;
    return _recover;
}

Mat deHaze(Mat img) {
    CV_Assert(img.channels() == 3);

    Mat _minChannel = minChannel(img);
    Mat _darkChannel = darkChannel(_minChannel);
    Vec3b _A = atmosphic(img, _darkChannel);
    float w = 0.85f, t0 = 0.1f;
    Mat _tx = transmission(img, _A, w, t0);
    Mat _recover = recover(img, _tx, _A);
    return _recover;
}
```

- GPU版本的图像去雾代码(**DehazeGPU.cpp**)

```
//导入opencv库，iostream处理输入输出，cuda相关库
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <stdio.h>
//导入自定义库
#include "util.h"
//使用命名空间 std和cv
using namespace std;
using namespace cv;

#pragma once
#ifdef __INTELLISENSE__
    void __syncthreads();
#endif

//实现的几个功能函数
__host__ void displayInfo();
__global__ void GPU_minChannel_Kernel(uchar*, uchar*, int);
__global__ void GPU_darkChannel_Kernel(uchar*, uchar*, int, int);
__global__ void GPU_transmission_Kernel(uchar*, float*, uchar*, int,
float, float);
```

```
__global__ void GPU_recover_Kernel(uchar*, float*, uchar*, uchar*, int);
__host__ Mat GPU_minChannel(Mat);
__host__ Mat GPU_darkChannel(Mat);
__host__ Mat GPU_transmission(Mat, Vec3b, float, float);
__host__ Mat GPU_recover(Mat, Mat, Vec3b);
__host__ void errCatch(cudaError_t);

int main() {
    displayInfo();
    Mat tohaze = imread("extra1.jpg");
    Mat minChannel = GPU_minChannel(tohaze);
    Mat darkChannel = GPU_darkChannel(minChannel);
    Vec3b A = atmosphic(tohaze, darkChannel);
    float w = 0.85f, t0 = 0.1f;
    Mat tx = GPU_transmission(tohaze, A, w, t0);
    Mat recover = GPU_recover(tohaze, tx, A);

    imshow("hazed image", tohaze);
    imshow("Dehazed image", recover);
    waitKey(0);
    return 0;
}

__global__
void GPU_minChannel_Kernel(uchar* _tohaze_, uchar* _minChannel_, int col)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int _row = tx / col;
    int _col = tx % col;
    uchar t_inf = 0xff;
    for (int t = 0; t < 3; t++) {
        t_inf = t_inf > _tohaze_[3 * tx + t] ? _tohaze_[3 * tx + t] :
t_inf;
    }
    _minChannel_[_row * col + _col] = t_inf;
}

__global__
void GPU_darkChannel_Kernel(uchar* _minChannel_, uchar* _darkChannel_,
int row, int col)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int _row = tx / col;
    int _col = tx % col;

    int m = 7, n = 7;

    int xl, xr, yl, yr;
    xl = _row - m + 1 > 0 ? _row - m + 1 : 0;
    xr = _row + m - 1 < row ? _row + m - 1 : row;
    yl = _col - n + 1 > 0 ? _col - n + 1 : 0;
    yr = _col + n - 1 < col ? _col + n - 1 : col;

    uchar t_inf = 0xff;
    for (int k = xl; k < xr; ++k)
    {
        for (int l = yl; l < yr; ++l)
        {
```

```
            uchar s = _minChannel_[k * col + l];
            t_inf = t_inf > s ? s : t_inf;
        }
    }
    _darkChannel_[_row * col + _col] = t_inf;
}

__global__
void GPU_transmission_Kernel(uchar* _tohaze_, float* _tx_, uchar* _a_,
int col, float w, float t0)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int _row = tx / col;
    int _col = tx % col;

    _tx_[_row * col + _col] = (float)_tohaze_[3 * tx + 0] / _a_[0];
    _tx_[_row * col + _col] = _tx_[_row * col + _col] > (float)_tohaze_[3
* tx + 1] / _a_[1] ? (float)_tohaze_[3 * tx + 1] / _a_[1] : _tx_[_row *
col + _col];
    _tx_[_row * col + _col] = _tx_[_row * col + _col] > (float)_tohaze_[3
* tx + 2] / _a_[2] ? (float)_tohaze_[3 * tx + 2] / _a_[2] : _tx_[_row *
col + _col];

    _tx_[_row * col + _col] = 1.0 - w * _tx_[_row * col + _col] > t0 ?
1.0 - w * _tx_[_row * col + _col] : t0;
}

__global__
void GPU_recover_Kernel(uchar* _tohaze_, float* _tx_, uchar* _a_, uchar*
_res_, int col)
{
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int _row = tx / col;
    int _col = tx % col;

    for (int i = 0; i < 3; i++)
    {
        int t = _tohaze_[3 * (_row * col + _col) + i] - _a_[i];
        int s = int(t / _tx_[_row * col + _col] + _a_[i]);
        //printf("%d %d %d %d %d\n", int(_tohaze_[3 * (_row * col + _col)
+ i]),int(_a_[i]),int(t), int(t / _tx_[_row * col + _col]), s);
        s = s > 0xff ? 0xff : s;
        _res_[3 * (_row * col + _col) + i] = uchar(s);
        //printf(" %d \n", int(_res_[3 * (_row * col + _col) + i]));
    }
}

__host__
void displayInfo() {
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);
    //cout << deviceCount << endl;
    for (int i = 0; i < deviceCount; i++)
    {
        cudaDeviceProp devProp;
        cudaGetDeviceProperties(&devProp, i);
        cout << "使用GPU device " << i << ": " << devProp.name << endl;
```

```
        cout << "设备全局内存总量:  " << devProp.totalGlobalMem / 1024 /
1024 << "MB" << endl;
        cout << "SM的数量:" << devProp.multiProcessorCount << endl;
        cout << "每个线程块的共享内存大小:" << devProp.sharedMemPerBlock /
1024.0 << " KB" << endl;
        cout << "每个线程块的最大线程数:" << devProp.maxThreadsPerBlock <<
endl;
        cout << "设备上一个线程块(Block)种可用的32位寄存器数量:  " <<
devProp.regsPerBlock << endl;
        cout << "每个EM的最大线程数:" <<
devProp.maxThreadsPerMultiProcessor << endl;
        cout << "每个EM的最大线程束数:" <<
devProp.maxThreadsPerMultiProcessor / 32 << endl;
        cout << "设备上多处理器的数量:  " << devProp.multiProcessorCount <<
endl;
        cout << "================================================="
<< endl;
    }
}

__host__
Mat GPU_minChannel(Mat img)
{
    CV_Assert(!img.empty());

    int col = img.cols, row = img.rows;
    uchar *_tohaze, *_minChannel;
    uchar *_cuda_tohaze, *_cuda_minChannel;

    _tohaze = (uchar*)malloc(sizeof(uchar) * col * row * 3);
    _minChannel = (uchar*)malloc(sizeof(uchar) * col * row);
    _tohaze = img.data;

    errCatch(cudaMalloc((void**)&_cuda_tohaze, 3 * col * row *
sizeof(uchar)));
    errCatch(cudaMalloc((void**)&_cuda_minChannel, col * row *
sizeof(uchar)));
    errCatch(cudaMemcpy(_cuda_tohaze, _tohaze, 3 * col * row *
sizeof(uchar), cudaMemcpyHostToDevice));

    dim3 dimGrid(row);
    dim3 dimBlock(col);

    //开始计时
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    GPU_minChannel_Kernel << < dimGrid, dimBlock >> > (_cuda_tohaze,
_cuda_minChannel, col);
    cudaThreadSynchronize();
    //停止计时
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elaspedTime;
    cudaEventElapsedTime(&elaspedTime, start, stop);
    cout << "minChannel time: " << elaspedTime << " ms" << endl;
    cudaEventDestroy(start);
```

```
    cudaEventDestroy(stop);

    errCatch(cudaMemcpy(_minChannel, _cuda_minChannel, col * row *
sizeof(uchar), cudaMemcpyDeviceToHost));
    Mat minChannel(row, col, CV_8UC1, _minChannel);
    errCatch(cudaFree(_cuda_tohaze));
    errCatch(cudaFree(_cuda_minChannel));
    return minChannel;
}

__host__
Mat GPU_darkChannel(Mat img)
{
    CV_Assert(!img.empty());

    int col = img.cols, row = img.rows;
    uchar *_minChannel, *_darkChannel;
    uchar *_cuda_minChannel, *_cuda_darkChannel;

    _minChannel = (uchar*)malloc(sizeof(uchar) * col * row);
    _darkChannel = (uchar*)malloc(sizeof(uchar) * col * row);
    _minChannel = img.data;

    errCatch(cudaMalloc((void**)&_cuda_minChannel, col * row *
sizeof(uchar)));
    errCatch(cudaMalloc((void**)&_cuda_darkChannel, col * row *
sizeof(uchar)));
    errCatch(cudaMemcpy(_cuda_minChannel, _minChannel, col * row *
sizeof(uchar), cudaMemcpyHostToDevice));

    dim3 dimGrid(row);
    dim3 dimBlock(col);

    //开始计时
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    GPU_darkChannel_Kernel << < dimGrid, dimBlock >> > (_cuda_minChannel,
_cuda_darkChannel, row, col);
    cudaThreadSynchronize();
    //停止计时
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elaspedTime;
    cudaEventElapsedTime(&elaspedTime, start, stop);
    cout << "darkChannel time: " << elaspedTime << " ms" << endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    errCatch(cudaMemcpy(_darkChannel, _cuda_darkChannel, col * row *
sizeof(uchar), cudaMemcpyDeviceToHost));
    Mat darkChannel(row, col, CV_8UC1, _darkChannel);
    errCatch(cudaFree(_cuda_minChannel));
    errCatch(cudaFree(_cuda_darkChannel));
    return darkChannel;
}
```

```
__host__
Mat GPU_transmission(Mat img, Vec3b a, float w, float t0)
{
    CV_Assert(!img.empty());

    int col = img.cols, row = img.rows;
    uchar *_tohaze, *_cuda_tohaze;
    float *_tx, *_cuda_tx;
    uchar *_a, *_cuda_a;

    _tohaze = (uchar*)malloc(sizeof(uchar) * col * row * 3);
    _tx = (float*)malloc(sizeof(float) * col * row);
    _a = (uchar *)malloc(sizeof(uchar) * 3);
    _tohaze = img.data;
    for (int i = 0; i < 3; i++) _a[i] = a[i];

    errCatch(cudaMalloc((void**)&_cuda_tohaze, 3 * col * row *
sizeof(uchar)));
    errCatch(cudaMalloc((void**)&_cuda_tx, col * row * sizeof(float)));
    errCatch(cudaMalloc((void**)&_cuda_a, 3 * sizeof(uchar)));
    errCatch(cudaMemcpy(_cuda_tohaze, _tohaze, 3 * col * row *
sizeof(uchar), cudaMemcpyHostToDevice));
    errCatch(cudaMemcpy(_cuda_a, _a, 3 * sizeof(uchar),
cudaMemcpyHostToDevice));

    dim3 dimGrid(row);
    dim3 dimBlock(col);

    //开始计时
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    GPU_transmission_Kernel << < dimGrid, dimBlock >> > (_cuda_tohaze,
_cuda_tx, _cuda_a, col, w, t0);
    cudaThreadSynchronize();
    //停止计时
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elaspedTime;
    cudaEventElapsedTime(&elaspedTime, start, stop);
    cout << "transmission time: " << elaspedTime << " ms" << endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    errCatch(cudaMemcpy(_tx, _cuda_tx, col * row * sizeof(float),
cudaMemcpyDeviceToHost));
    Mat tx(row, col, CV_32F, _tx);
    errCatch(cudaFree(_cuda_tohaze));
    errCatch(cudaFree(_cuda_tx));
    errCatch(cudaFree(_cuda_a));
    return tx;
}

__host__
Mat GPU_recover(Mat img, Mat t, Vec3b a)
{
    CV_Assert(!img.empty());
```

```
    int col = img.cols, row = img.rows;
    uchar *_tohaze, *_cuda_tohaze;
    float *_tx, *_cuda_tx;
    uchar *_a, *_cuda_a;
    uchar *_res, *_cuda_res;

    _tohaze = (uchar*)malloc(sizeof(uchar) * col * row * 3);
    _tx = (float*)malloc(sizeof(float) * col * row);
    _a = (uchar *)malloc(sizeof(uchar) * 3);
    _res = (uchar*)malloc(sizeof(uchar) * col * row * 3);
    _tohaze = img.data;
    for (int i = 0; i < row; i++)for (int j = 0; j < col; j++) _tx[i *
col + j] = t.at<float>(i, j);
    for (int i = 0; i < 3; i++) _a[i] = a[i];

    errCatch(cudaMalloc((void**)&_cuda_tohaze, 3 * col * row *
sizeof(uchar)));
    errCatch(cudaMalloc((void**)&_cuda_tx, col * row * sizeof(float)));
    errCatch(cudaMalloc((void**)&_cuda_a, 3 * sizeof(uchar)));
    errCatch(cudaMalloc((void**)&_cuda_res, 3 * col * row *
sizeof(uchar)));
    errCatch(cudaMemcpy(_cuda_tohaze, _tohaze, 3 * col * row *
sizeof(uchar), cudaMemcpyHostToDevice));
    errCatch(cudaMemcpy(_cuda_tx, _tx, col * row * sizeof(float),
cudaMemcpyHostToDevice));
    errCatch(cudaMemcpy(_cuda_a, _a, 3 * sizeof(uchar),
cudaMemcpyHostToDevice));

    dim3 dimGrid(row);
    dim3 dimBlock(col);

    //开始计时
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);
    GPU_recover_Kernel << < dimGrid, dimBlock >> > (_cuda_tohaze,
_cuda_tx, _cuda_a, _cuda_res, col);
    cudaThreadSynchronize();
    //停止计时
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
    float elaspedTime;
    cudaEventElapsedTime(&elaspedTime, start, stop);
    cout << "recover time: " << elaspedTime << " ms" << endl;
    cudaEventDestroy(start);
    cudaEventDestroy(stop);

    errCatch(cudaMemcpy(_res, _cuda_res, 3 * col * row * sizeof(uchar),
cudaMemcpyDeviceToHost));
    Mat res(row, col, CV_8UC3, _res);
    errCatch(cudaFree(_cuda_tohaze));
    errCatch(cudaFree(_cuda_tx));
    errCatch(cudaFree(_cuda_a));
    errCatch(cudaFree(_cuda_res));
    return res;
}
```

```
__host__
void errCatch(cudaError_t err) {
    if (err != cudaSuccess) {
        cout << cudaGetErrorString(err) << endl;
        exit(EXIT_FAILURE);
    }
}
```