论文题目：**CPU-GPU异构平台上平面光源检测方法的并行化设计与实现**
学生姓名：张志源
指导教师：吴茜媛

摘 要

在高新技术不断发展的当下，实现完全的自动化是未来工业制造的发展方向。智能检测作为自动化的关键环节，对机器运作的实时性要求是一个硬性需求。本文基于手机屏幕的缺陷的自动化检测，主要研究了图像处理过程中的程序优化方法，提高运行速度以满足实时性的要求。

本文先介绍了手机屏幕缺陷检测系统在生产线上的工作流程，根据实际的需求采用了多线程并行的程序设计方法来进行优化。为了解决CPU计算能力的瓶颈，利用了GPU的高并发计算提高部分图像处理函数的性能。

多线程并行的方案中，根据程序的工作流程设计了两种线程类型，说明了如何分配线程，并利用了OpenMP对一些循环程序进行了并行化处理。对GPU计算的利用主要是在CUDA架构下，借助了OpenCV的CUDA模块来实现，针对检测算法中的特定函数进行移植；并在移植过程中设计了易扩展和维护的程序框架。

最后，本文对并行化后的程序进行了多方面的测试和分析，总结了毕设的工作成果，也提出了在GPU的利用方面存在的不足，指明了后续研究的方向。

关键词：多线程；图像处理；GPU；OpenCV；CUDA

Title:

name:ZhiYuan Zhang

Supervisor: XiYuan Wu

ABSTRACT

In the continuous development of high-tech, to achieve full automation is the future direction of industrial manufacturing. Intelligent detection as a key link in the automation, the real-time capability of the operation of the machine is a rigid demand. Based on the automatic detection of defects in mobile phone screen, this paper mainly studies the program optimization method in the image processing process, and improves the running speed to meet the real-time capability. This paper introduces the workflow of the mobile phone screen defect detection system in the production line, according to the actual needs of the use of multi-threaded parallel programming method to optimize. In order to solve the bottleneck of CPU computing power, the use of GPU high concurrent computing to improve the performance of some image processing functions.

Multi-threaded parallel program, according to the program's workflow design of the two types of threads, explains how to allocate threads, and the use of OpenMP to implement parallel processing for loop?code. The use of GPU computing is based on the CUDA architecture, with the help of OpenCV CUDA module to run the specific functions of detection algorithm.And also designed a program framework that is easy to expand and maintain during the work.

Finally, this paper has carried on the multidimensional test and the analysis to the parallel procedure, has summarized the work result, also put forward the insufficiency in the utilization of the GPU, pointed out the follow-up research direction.

KEY WORDS:Multi-Thread;Image Processing;GPU;OpenCV;CUDA

目 录

1 绪论

1.1 背景与意义

智能制造是当前中国产业变革的主攻方向，2015年首次提出的“中国制造2025”，是中国政府实施制造强国战略第一个十年的行动纲领，旨在发展高技术含量的制造行业，改变中国制造业“大而不强”的局面[1]。而智能检测是智能制造中的关键环节之一；能否满足智能检测中的实时性要求，直接影响了智能制造流水线中的生产效率。

在智能手机的生产线上，在手机液晶屏幕的质量检测这一环节，缺少直接有效的方法来实现自动化，因此只能安排专人来把关，人工来检测产品缺陷[2]。笔者所参与的项目，主要的工作是使用基于图像识别的方法，在生产线上实现自动化设备来检测手机屏幕缺陷，以此代替传统人工检测，减少不必要的人力资源开销，并提高生产效率。

为了满足实时性的要求，最根本的途径是提高检测算法的执行速度，这也是笔者的主要工作和研究重点；如果算法的时间开销过大，检测系统将难以和现场的生产设备对接。如何提高检测环节中复杂算法的执行效率，就是一个的亟待解决、且有广泛应用价值的问题。

1.2 研究现状

在有限的计算资源下，利用并行计算是提高算法执行效率的最直接的方法。在该项目下，对同一组手机屏幕的一次检测中需要拍摄多张图片，也就意味着要在短时间内对多张图片执行相同的检测算法，在这个地方可以利用多核CPU 的多线程并发，对每张图片分配一个线程执行算法。

如果深入到处理器的计算性能，在大规模并行计算领域，GPU 和CPU 相比，展现了更强大的浮点运算能力。GPU 的可编程性在未开始发展时，开发人员要借助复杂的计算机图形学API来对GPU进行编程，这对非专业人员造成极大的困难[3]。而近年来，GPU在计算性能不断提高的同时，它的可编程性也在不断提高，意味着GPU可

以在通用计算领域得到更广泛的应用；像这一类的GPU被称为通用GPU，即GPGPU（General Purpose GPU）[4]。目前应用较广泛的GPGPU 平台主要有CUDA （Compute Unified Device Architecture，统一计算设备架构）、OpenCL （Open Graphics Library，开放计算语言）。CUDA是显卡厂商NVIDIA推出的、基于自家公司生产的GPU 开发出来的，使用C 语言来设计需要的程序，对所进行的计算进行分配和管理[3]。

借助CUDA的架构来对算法进行移植是目前常用的解决性能问题的方法。问题在于，对现有图像处理算法进行移植和优化仍然需要比较强的专业和理论基础，如果只是针对特定算法倒是可行，但是要用上述方法对该项目的检测算法进行移植，涉及到复杂的处理流程，移植算法需要很长的学习和研发周期。考虑到易用性和友好程度，这里着重关注OpenCV（Open Source Computer Vision Library）的GPU模块；这个模块最早在NVIDIA 公司支持下进行开发，并于2011 年春正式发行[6]。目前为止也更新了大量由CUDA 代码编写的图形处理算法，这也就意味着开发人员可以使用这些通用的算法API来利用GPU 进行计算，免去了繁杂的算法设计和优化。由于OpenCV 的开源特性，专家和爱好者可以共同维护和开发OpenCV的GPU模块，并不断完善，在图形处理方面有着良好的发展前景。

本次毕设的主要工作就是在CPU-GPU异构平台上，使用CPU多线程并行优化程序以满足基本的性能需求，利用CUDA 架构对相关的算法函数进行移植，从计算性能的角度对程序进行优化，探索其性能提高的方法。

1.3 论文主要内容

毕设工作围绕手机屏幕缺陷检测项目中算法程序的优化展开，针对项目实际需求使用了CPU多线程并行，调研并研究了如何利用GPU高并发的计算来加速图像处理，设计程序框架并以代码实现，进行性能评估。论文主要内容如下：1）手机屏幕缺陷检测流程概况：介绍毕设工作所处的项目背景，阐述项目所要解决的问题，项目的主要工作流程；通过项目的实际需求，说明了程序优化的目的和方向。

2）关于利用GPU计算加速算法的调研：介绍NVIDIA公司推出的CUDA架构，以及OpenCV的CUDA模块在GPU计算中的应用，主要目的是为了利用GPU的高并行计

算来解决图像处理过程中的计算瓶颈；并对OpenCV中CUDA模块的滤波函数进行了初步的性能测试。

3）CPU多线程并行的优化方案和OpenMP的应用：介绍在该项目中如何利用多线程来对检测算法进行整体的优化，以及使用OpenMP 对局部代码进行并行优化，探讨其中发现的问题。

4）利用OpenCV的CUDA模块进行算法移植和测试：介绍笔者针对该项目的算法、借助OpenCV中CUDA模块的API对原有代码进行改写，调用GPU来进行一些图像处理的计算，并对移植后的算法进行性能的测试和分析。

5）CUDA函数在CPU多线程代码中的运行状态：在之前算法移植的工作基础上，研究移植后的代码在CPU多线程条件下的运行状态。

6）毕设工作总结：总结毕设工作，研究工作中存在的问题，介绍其它可行的解决方案和下一步的工作。

1.4 论文组织框架

第一章为绪论部分，主要概述论文所研究的问题和实际意义，包括研究背景调研和笔者采用的解决方案。

第二章主要介绍笔者的研究所处的项目背景和笔者的主要工作，解决问题的程序设计方案，包括多线程的设计，GPU通用计算的使用。

第三章对第二章提到的解决方案的实施进行详细的说明，包括程序总体框架的设计、程序中具体线程的分配，和OpenCV中CUDA模块的具体应用；

第四章介绍研究工作的展开和具体实现，包括开发环境的部署和配置，局部和整体的性能测试，以及探讨工作中出现的问题；

第五章则总结了本论文的研究工作，提出存在的问题和待研究的方向。

2　手机屏幕缺陷检测流程概况和并行优化方案

本章先介绍了缺陷检测的大致流程步骤，基于实际的需要，才会有并行化处理的性能优化方案。

2.1　手机屏幕缺陷检测的工作流程概况

手机屏幕缺陷检测系统的抽象工作流程如图 2-1 所示。程序启动到读取算法参数属于初始化的过程，系统从图像采集开始进入正常的流水工作。图像采集使用高精度的相机进行拍摄，使其能够捕捉到手机液晶屏表面沾染的微小污点；拍摄完毕后进行图像的检测，图像的输入格式为BMP （Bitmap，Windows标准图像文件格式），经过分离和平滑等预处理后，执行缺陷检测；把检测到的缺陷个数返回作为检测结果，同时保存结果图像。一般情况下，只要存在任何缺陷就可以判定产品不合格。

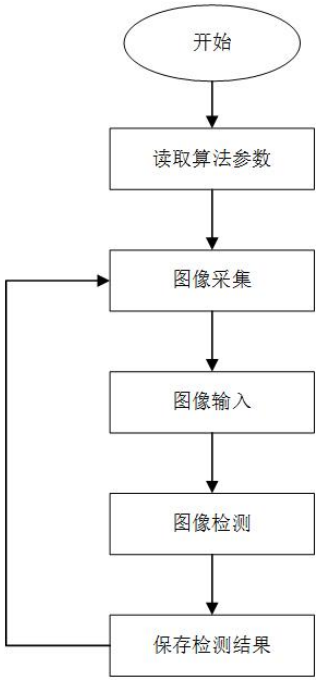

图 2-1　手机屏幕缺陷检测整体流程

2.1.1　读取算法参数

程序使用配置文件来保存程序中会用到的各种算法参数、变量，包括输入图像的

分辨率、有效区域范围等等。配置文件以ini的文件格式保存，读取算法参数时则将文件读入并逐行识别字符。文件的书写格式如下所示：

```
[system]
captured_img_width          = 6600
captured_img_height         = 4400
left_BGB_x0                 =1398
left_BGB_y0                 =1291
```

算法参数说明可参考附录2。使用配置文件来调整程序中用到的参数值，主要目的是为了方便算法程序与各种平台进行对接。例如，前台界面程序使用C#语言编写，通过调用算法的DLL（动态链接库）来实现图像的检测；前台界面需要对相关参数进行调整时，可直接修改配置文件相应位置的数值而不用去关心后台的代码。

2.1.2　图像采集和输入

在智能手机的自动化生产线上，手机液晶屏所在的工位一次装载两个手机屏幕（不包含其它手机零部件）。相机照明分自发光（液晶屏接通电源发光）和外光源（开启工位上的条形光源）两种,如图 2-2和图 2-3。

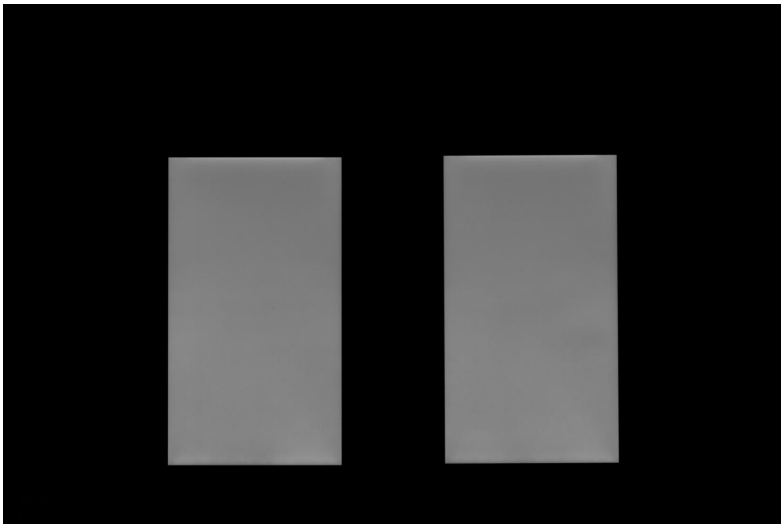

图 2-2　自发光条件下拍摄的图片

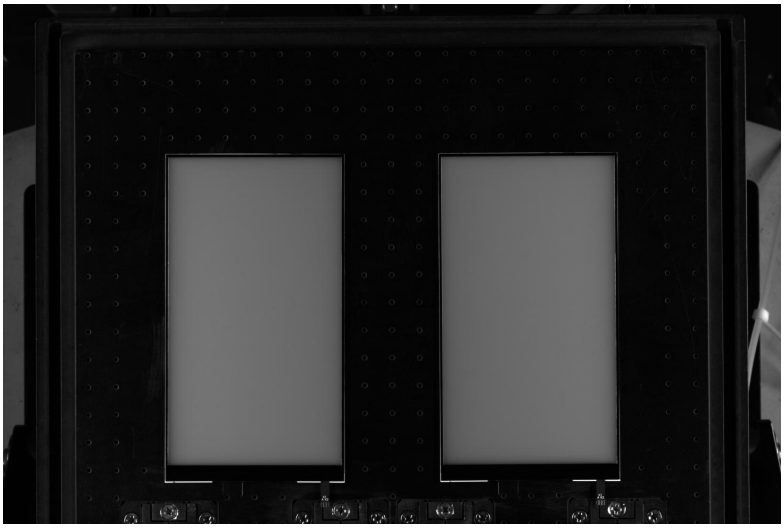

图 2-3　外光源条件下拍摄的图片

我们所看到的输入图像的有效区域其实就是屏幕所在的矩形区域。而自发光图片比起外光源图片具有更好的辨识度，因此在图像分离步骤中使用自发光图片来提取有效区域掩模版。

工位相机在两种不同的照明条件下各拍摄一次，然后使用真空泵覆一层膜再拍摄一次。两次拍摄的图像作为一组输入，输入的图像经过分离（结果如图 2-4）和有效区域提取（即手机屏幕所在的矩形区域，如图 2-5），分离后的图片暂称之为“单边图像”，分离后的左右两个样品分别进行一次检测。

图 2-4　分离后的图像（左）

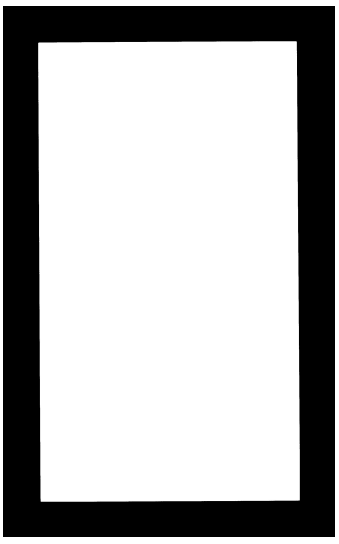

图 2-5　有效区域掩模版（左）

也就是说，现场检测一个工位的产品，至少要调用四次检测算法。这里使用真空

泵覆膜是厂商的要求，还不考虑斜视的情况；斜视的情况即是把工位上的手机屏幕倾斜一定角度，在上述的照明条件下拍摄一组图像输入检测，因为在传统人工检测过程中，手机屏幕的部分缺陷要倾斜一定角度才能用肉眼发现。由此来看，如果出于提高检测准确度的考虑，后续可能要处理的不止是覆膜和不覆膜这两组图片，调用的检测算法也不止四次；这里为了提高处理速度，使用多线程来处理是必须的。

2.1.3　图像检测

检测的缺陷对象有位于屏幕表面的污点和划痕（生产车间是无尘环境，所以这些类型的缺陷会比较少哦），还有因为复杂制作工艺产生的异物缺陷和短路缺陷等，检测过程主要采用了基于边缘检测的方法（借助Canny边缘检测算法实现）和基于滤波的方法；屏幕表面缺陷点的图像深度（即颜色深浅）和周围的像素差异较大的部分，即可认为是缺陷，在图像放大到一定程度的情况下，通过人眼也能辨识出。算法主要是通过识别出这种深度的差异来作判断[2]。

检测结果使用矩形框保存。对矩形框进行描边之后生成可视化的结果图像如图2-6和图 2-7所示。在实际情况中，需要检测系统返回的也就是“产品合格”和“产品不合格”两种结果。所以除了产生结果图像之外，还要返回一个检测出的缺陷数量，以方便工位上的机器作出反馈。这里所选取的样本表面较脏，因为不是在无尘环境下，所以检测出的缺陷数量比较多。

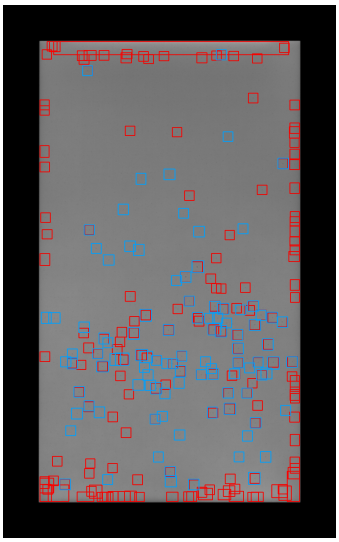

图 2-6　图像检测结果（左）

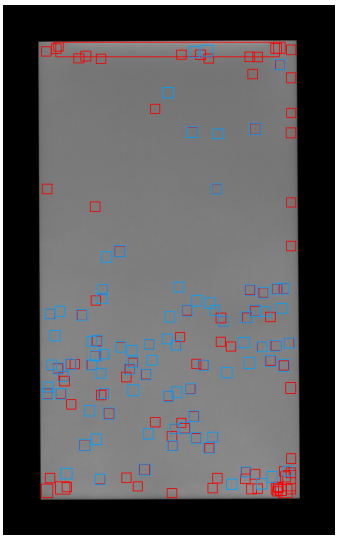

图 2-7　图像检测结果（右）

2.2 手机屏幕缺陷检测方法的总体优化方案

本节的内容基于缺陷检测的工作流程，根据不同的需要确定了优化方案。从程序整体框架出发使用CPU多线程并行；针对局部代码的优化，应用OpenMP对循环进行并行化处理，以及使用GPU 通用计算优化检测算法。

1）CPU多线程并行

该方案是现在正适用中的优化方案，针对实际中需要同时对多张图像进行处理的需要（对一组产品需要同时处理覆膜和不覆膜两组图像，每组组片要同时处理左、右两张分离出来的单边图像），利用多核CPU 的线程并发来同时处理多张图像。该方案并不是从计算性能的角度来进行加速，通过多核CPU 来同时处理多个事务；同时输入的多张图片执行相同的检测流程，由原来的串行处理变成并行处理。

2）应用OpenMP对代码进行并行化处理

同样是利用CPU的多线程并行来优化代码，不同的是OpenMP在这里的应用是针对特定函数中的循环语句，使用OpenMP来对这些串行执行的循环进行并发处理，通过提高这些局部代码的效率来提高程序整体的运行效率。

3）CUDA架构下的GPU通用计算优化检测算法

这个方法从计算性能的角度出发，利用GPU的通用计算来对检测算法中特定的函数进行加速优化。该项目涉及的计算主要是图像处理领域的算法，因此，初步的方案是使用OpenCV的GPU模块（OpenCV 3.0以上版本称为CUDA模块），借助该模块丰富的图形处理算法来对源代码进行移植，并对各模块逐个进行性能测试。该方案实际在项目中是备选方案，因为GPU模块并没有完全实现所有的OpenCV 原有的图像处理算法，在移植过程中一旦出现问题，并不能保证能在短时间内解决；此外，使用CUDA架构也有着额外的时间开销，比如内存和显存之间的数据交换等，要详细进行性能的比较和评估。在检测流程中的关键函数移植完毕后，就可以把这些函数应用到CPU多线程条件下，进一步提高性能。

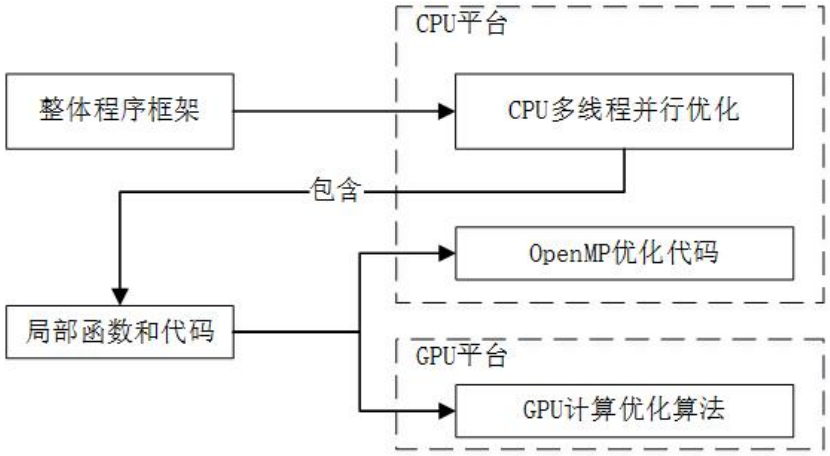

图 2-8　整体优化方案

2.2.1　CPU多线程并行优化方案

1）基于CPU多线程对图像处理的整体优化

创建的线程类型根据调用的函数不同，为了方便说明暂命名为frame线程和Inspect线程。主线程的操作包括实例化框架类、读取图像和配置文件，开启frame线程调用框架类的成员函数。

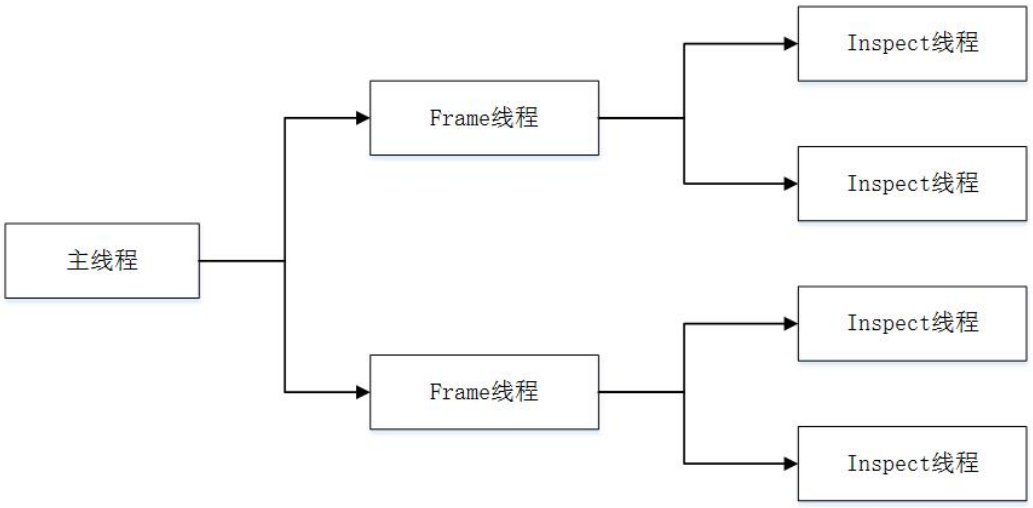

图 2-9　线程的类型和分配

如果不考虑手机屏幕覆膜和不覆膜的区别，正常输入的是自发光和外光源各一张图像，开启一个frame线程。在一个frame线程里，要检测左右两个被分离出来的单边图像，因此再分别开启两个Inspect线程来对图像进行检测。因为要考虑手机屏幕覆膜情况，所以要再开启一个frame线程处理覆膜图像，过程同上述。所以一共要开启四

个Inspect线程，执行四次检测算法。

frame线程和Inspect线程的关系和线程所处理的事务如图 2-10 所示。frame线程包含了一部分的图像预处理，包括图像分离和提取有效区域掩模版；而Inspect线程开启后，在执行的线程函数内部实例化一个检测类，再调用检测类的相关函数进行进一步的预处理、缺陷检测和结果保存。

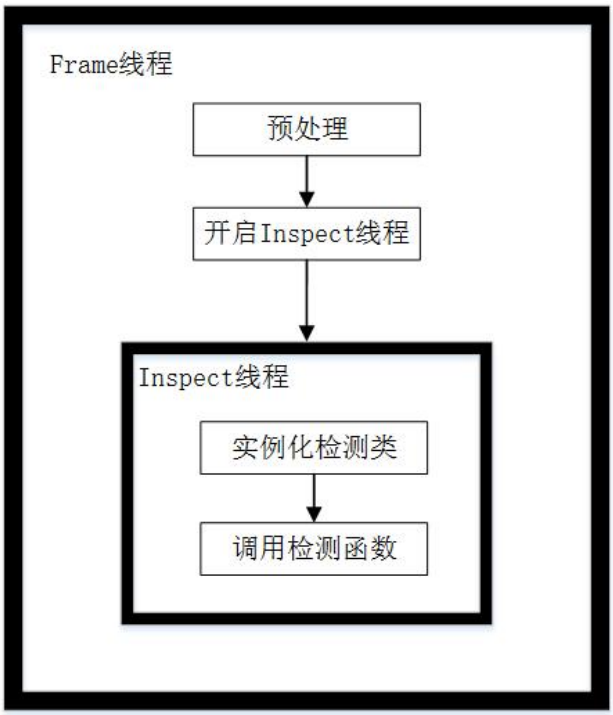

图 2-10　线程Frame和Inspect的关系

2）应用OpenMP对局部代码的并行优化

在Visual Studio中支持使用OpenMP的预编译指令来对一些循环代码来进行并行计算。程序中的局部代码使用for循环来对图像中的像素进行处理。包括其它计算量较大的循环语句，都有必要进行优化。

2.2.2　基于GPU通用计算的检测算法优化

1）GPU通用计算和CUDA应用方案的选择

随着游戏市场的壮大，仿真计算的需求等因素，推动了GPU的性能不断提升。而GPU内部独特的架构使其在多线程并发中拥有强大的计算能力，并可以作为通用计算设备应用于医学、天文和金融等邻域。计算机中的图像是以矩阵的形

式存储，而GPU的高并发计算能力使其在图像处理方面拥有比CPU更快的效率[8]。而NVIDIA推出的基于C语言编程的CUDA架构，进一步降低了GPU的编程门槛[5]。如下使用了CUDA代码实现了一个CUDA函数[9]:

```
__global__ void VectorAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
int main()
{
VectorAdd<<<1, N>>>(A, B, C);
}
```

VectorAdd()函数实现了两个一维向量的相加，并且以N个线程来并发执行。而在本次手机屏幕缺陷检测的项目中，涉及到的检测算法处理流程复杂，如果全部使用上述CUDA代码进行移植，需要熟悉很多图像处理算法的专业知识，而且无法在短时间完成所有的移植工作；CUDA编程涉及到具体的线程分配和专业的算法优化，同时也需要长期的测试和改良。考虑到算法中涉及到的对性能影响较大的主要是一些常规的图像处理，可以借助OpenCV中的CUDA模块来完成算法的移植。

2）在OpenCV中调用CUDA的方法

OpenCV中的CUDA模块（或者说GPU模块）的使用一般分为这样几个步骤：

a）支持CUDA的设备初始化；

b）上传待处理数据到GPU（从Mat容器到GpuMat容器）；

c）调用OpenCV支持的GPU的处理函数；

d）下载处理结果到CPU（从GpuMat容器到Mat容器）[4]。

Mat类是OpenCV中用于存储矩阵和图像的容器，而GpuMat 类则是对应Mat 而设计出来，在显存上代替Mat的职能。上传是用了GpuMat 自带的upload() 方法，自动分配显存空间并将CPU上的Mat对象上传至显存上的GpuMat对象。而GpuMat

的download()方法可以把显存上的GpuMat对象下载至CPU，一般的用法是下载最终的处理结果。因为上传和下载都是需要一定时间的，频繁上传和下载势必会影响效率；可以定义需要的GpuMat类型的成员变量来直接引用，避免重复的对象实例化和显存分配。

这里使用一个代码片段，以图像的高斯滤波处理为例来来简要说明笔者如何具体应用OpenCV中的CUDA模块：

```
void gaussianBlur_GPU(Mat &src, Mat &dst, int size)
{
        if (getCudaEnabledDeviceCount() < 0)
        {
                cout << "No_Device_Enabled_For_Cuda!\n";
                return;
        }
        GpuMat gsrc, gdst;
        //registerPageLocked(src);   //锁页内存
        gsrc.upload(src);
        Ptr<cuda::Filter> p = cuda::createGaussianFilter
       (gsrc.type(), gsrc.type(), Size(size, size), 3);
        p->apply(gsrc, gdst);
        gdst.download(dst);
        //unregisterPageLocked(src);//解除锁页
        //imshow("dst_Gpu", dst);     //显示滤波结果
        //waitKey(0);
}
```

首先判断当前的环境和设备是否可用，使用getCudaEnabledDeviceCount()可用来返回可用的GPU设备数量，这个返回值一般是1；如果是多个GPU设备级联运算的情况下，这个返回值才会大于1。当返回值为0的情况下，表示当前的CUDA环境配置不

正确或者GPU设备不支持CUDA，因此OpenCV 的CUDA模块就不可用了。

然后是调用GpuMat的upload()函数上传源图像到显存，这里在上传过程中使用registerPageLocked() 函数来锁页内存可提高对GPU的访存速度。但是在该段代码的实际运行过程中，使用锁页与否并没有明显差异。

如果没有手动进行CUDA设备的初始化，那么OpenCV会在第一次调用CUDA函数的位置自动初始化；在该段代码中则会在调用upload()函数时自动初始化。初始化的开销在不同机器上都有所不同，通常会有上千毫秒，在笔者的笔记本上一般会界于2200到2500毫秒。所以建议在程序的主体部分未开始前，使用cuda::SetDevice(0)进行手动初始化CUDA；否则，在每一次运行程序时，都会导致第一次调用CUDA的函数会因为初始化而严重延迟。

11到13行的代码使用模板指针创建了一个高斯滤波器并调用。cuda模块的滤波操作，包括本次毕设工作还会用到的形态学操作，边缘检测等方法都有对应的类模板；跟普通的opencv函数调用不一样的地方是，cuda模块使用了工厂方法来生产这些需要的类，再通过这些类的方法（如13行的apply()）来完成操作。与直接编写CUDA代码相比，熟悉这些API之后的上手和编程效率会更快。

3）OpenCV的CUDA函数性能测试在正式实施OpenCVCUDA模块的应用之前，先对解决方案的可行性进行评估。使用OpenCV的CUDA函数对算法进行移植主要针对滤波、形态学操作等时间开销占比大的处理流程。所以这里使用OpenCV原本的高斯滤波函数和GPU模块的高斯滤波函数，来分别测试CPU和GPU的计算能力。高斯滤波又称为高斯模糊，主要操作是把图像的每个像素点与邻域进行加权平均，处理的效果如图 2-11和图 2-12所示。

图 2-11　原始图像

图 2-12　高斯滤波的结果

该测试的GPU和CPU硬件条件：NVIDIA GT750M 2G显存DDR3独立显卡；Intel Core i5-4200M 4G内存。表 2-1中的bmp格式的图片即用相机采集的手机屏幕样片；表中的上传和下载的时间开销分别指的是：在调用GPU的过程中，图片上传至显存的开销，和从处理结果从显存下载至内存的开销；此外，表中GPU运算时间开销不包含上传和下载的部分。 可以看出，即使加上上传和下载的时间，GPU的时间开销也是

表 2-1　CPU和GPU高斯滤波的处理速度对比

图片编号	图片大小	分辨率	上传/ms	下载/ms	Kernel大小	GPU 运算/ms	CPU运算/ms
0	89.6KB	512*512	0	0	3	0	16
1.bmp	27.6MB	6600*4400	31	43	3	129	534
2.bmp	27.6MB	6600*4400	78	16	3	114	531
3.bmp	27.6MB	6600*4400	31	36	3	125	531
4.bmp	27.6MB	6600*4400	31	31	3	110	532
5.bmp	27.6MB	6600*4400	31	32	3	109	531
6.bmp	27.6MB	6600*4400	31	16	3	109	516

比CPU要小的。而使用GPU时，会固定存在一个显存和内存之间数据交换的时间开销，也就是说，无论GPU的运算速度有多块，也不会影响到数据交换的速度。这导致数据交换成了一个性能瓶颈，这就是GPU的局限；从改善硬件条件的方向考虑，性能提升的空间也是很有限的；唯一能改善的方法就是尽量避免不必要的显存和内存的数据交换。

接下来使用不同大小的kernel来测试。

表 2-2　不同kernel大小时CPU和GPU的高斯滤波处理速度

图片编号	图片大小	分辨率	上传/ms	下载/ms	Kernel大小	GPU 运算/ms	CPU运算/ms
1	89.6KB	512*512	0	0	3	0	16
1	27.6MB	6600*4400	31	43	3	129	534
1	27.6MB	6600*4400	31	15	5	125	843
1	27.6MB	6600*4400	31	36	7	125	1687
1	27.6MB	6600*4400	31	31	9	141	2131

高斯滤波对每个像素点在其邻域内进行加权平均，而kernel的大小即是这个领域范围。设kernel大小为3，表示领域为3x3大小的矩阵，则每个像素与其周围的紧邻的八个像素点进行加权平均。kernel越大，意味着计算量越大。由表 2-2可知，当kernel增大时，GPU的运算时间变化不明显，而CPU的运算时间则是随着运算量的增大而明显增大。

4）GPU加速优化的程序设计方案

函数移植的方案是使用一个子类InpectorGPU来实现在GPU上运行的相关函数。子类继承原有的检测类BGBInpector，在这个基础上来重写父类的方法；重写的方法会覆盖父类的方法，同时其余继承自父类的方法没有变化。这样在移植过程中可以循序渐进，也方便测试性能变化。

2.3　本章小结

本章从实际的项目出发，先介绍了手机屏幕缺陷检测的整体工作流程，根据实际需要设计了CPU多线程的优化方案，并选择了借助OpenCV的CUDA模块来进行检测算法的移植，期望使用GPU计算来优化图像处理的性能。最后以高斯滤波为例，测试了GPU的计算性能并与CPU进行对比，证实了方案的可行性。

3 CPU多线程并行和GPU加速的优化实现

本章具体实施来手机屏幕缺陷检测方法的并行优化方法，从C++多线程并发、类型类型和分类、OpenMP的并行化应用和GPU计算几个方面展开。

3.1 CPU多线程并行优化算法的实现

在C++中使用多线程并行，可用thread类来实现；首先要包含thread头文件[7]。在C++中支持多线程的还有可能需要mutex来进行线程同步。但是在这里，同类型的线程之间的关系相对独立，不需要通信。

3.1.1 主线程的执行过程和Frame线程分配

接下来我们要在主线程中开启两个Frame线。先实例化一个框架类frame，用thread类创建两个线程，一个用于处理覆膜组的图像，另一个用于处理不覆膜组的图像；在线程中调用frame类中相关的处理函数。

```
void test()
{
CBGBFrame frame;
/*读取算法参数和图像*/
thread frame_filmed(bind(&CBGBFrame::SetFilmedImages,
    &frame, ref(filmed_self), ref(filmed_ext)));
thread frame_unfilmed(bind(&CBGBFrame::SetUnfilmedImages,
    &frame, ref(unfilmed_self), ref(unfilmed_ext)));
frame_filmed.join();
frame_unfilmed.join();
}
```

在DLL工程里编写对外接口也是同样的写法，创建线程然后加入线程池。第一个Frame线程frame_filmed调用的是类frame的SetFilmedImages() 函数，第二个线程frame_unfilmed 调用的则是SetUnfilmedImages()函数;两个函数的功能是一样的，区别在于一个处理覆膜组的图片，另一个处理不覆膜组的图片。

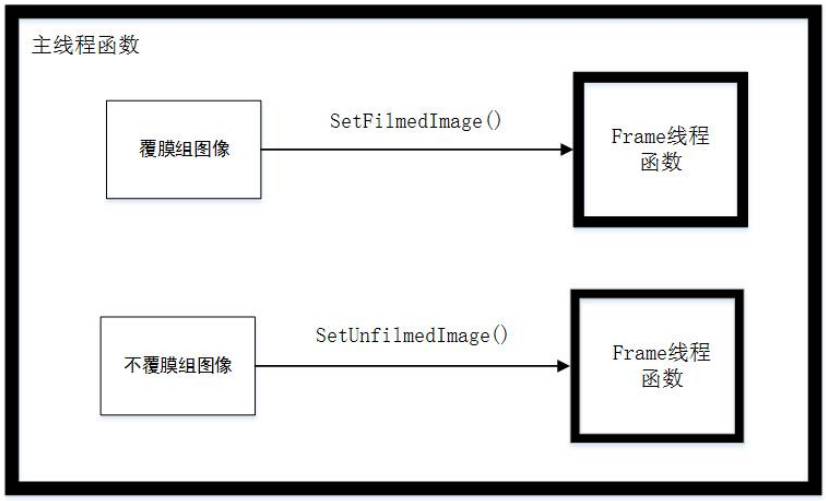

图 3-1　主线程函数执行过程

函数SetFilmedImages()中的filmed_self和filmed_ext参数，都是Mat类型的变量，在“读取图像”步骤中用来存储覆膜组的外光源条件下的图像和自发光条件下的图像。同样的，在SetUnfilmedImagesDebug()函数中的unfilmed_self和unfilmed_ext 参数分别存储不覆膜组的图像。线程启动后，进入相应的函数内部，再启动两个Inspect线程处理分别处理分离后的单边图像。

3.1.2　Frame线程的执行过程和Inspector线程分配

在进入Frame的线程函数之后，这里以SetFilmedImages()为例，把输入的覆膜组图像进行分离把左、右两个待检测手机屏幕的图像分别存放到一个Mat类型的变量中，对应以下代码中的img_filmed_self_left参数和img_filmed_self_right参数。这两个图像原本是自发光下拍摄的图像，也就是主线程中的img_filmed_self参数，严格来讲还需要再开启两个Inspect线程检测外光源情况下的图像。因为这里原定的检测方案要在两种发光情况下都进行检测完毕后，再调用一个比对算法对两种发光情况进行分析；但是这个研发工作还未完成，这里就只采用自发光情况下的图像进行检测。图像分离完毕

后，再提取左、右两张图像的有效区域掩模版（也就是手机屏幕在图中所在的矩形区域），掩模版对应的参数为mask_filmed_left和mask_filmed_right。至于最后两个参数，第一个表示该组图像的覆膜情况，”1“表示覆膜，”0“表示不覆膜；第二个参数为”1“表示处理的单边图像在左，为”0“表示在右。

```
int SetFilmedImages()
{
/*分离图像*/
/*提取有效区域掩模版*/
string str1 = dir + img_file_name + "_filmed_self_left";
string str2 = dir + img_file_name + "_filmed_self_right";
thread inspect_left(bind(&CBGBFrame::InspectSingleBGB, this,
    ref(img_filmed_self_left),ref(img_ext_empty),
    ref(mask_filmed_left),str1, ref(rect_left_result),1,1));
thread inspect_right(bind(&CBGBFrame::InspectSingleBGB, this,
    ref(img_filmed_self_right),ref(img_ext_empty),
    ref(mask_filmed_right),str2,ref(rect_right_result),1,0));
inspect_left.join();
inspect_right.join();
/*比对算法*/
}
```

此外，还有两个string类型的字符串变量”str1“和”str2“，这两个字符串根据当前的时间（包含年月日时分秒）和当前处理图像的覆膜情况、发光情况和放置位置来生成，并且包含了文件路径”result_dir“。str1和str2的作用是在检测缺陷结束、保存结果图像时作为图像的文件名前缀。例如结果图像文件名为”20170602_150615_filmed_self_left.jpg“表示当前图像属于自发光覆膜组，手机屏幕位于左边，处理时间是2017年6月2日15时6 分15秒。这个字符串表示时间的部分”20170602_150615“也可以用来生成日志文件名前缀。

这里的”分离图像“和”提取有效区域掩模版“过程，就是在2.3节提到的Frame线程负责的一部分预处理工作。在预先工作完成后，再针对左图和有图分别开启两个线程inspect_left和inspect_right执行检测，程序的处理流程如图 3-2所示。

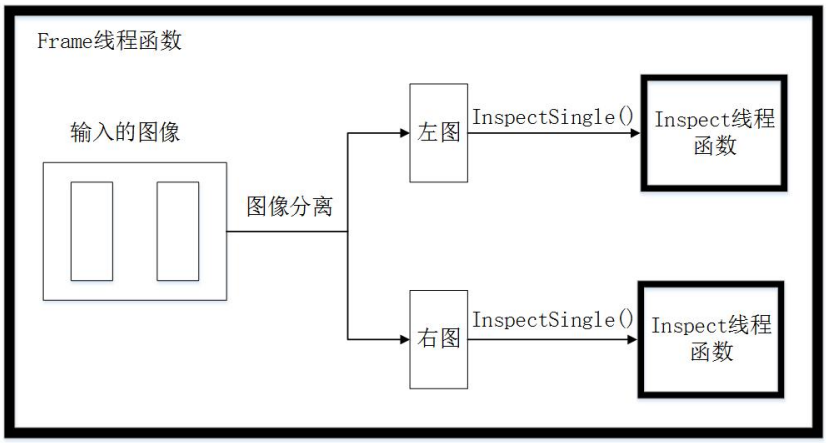

图 3-2　Frame线程函数执行过程

其中，线程函数调用的是CBGBFrame类的InspectSingleBGB()函数。这个函数包含了核心的缺陷检测流程，是整个程序运行耗时占比最大的部分，也是性能瓶颈所在。函数的执行过程如下：

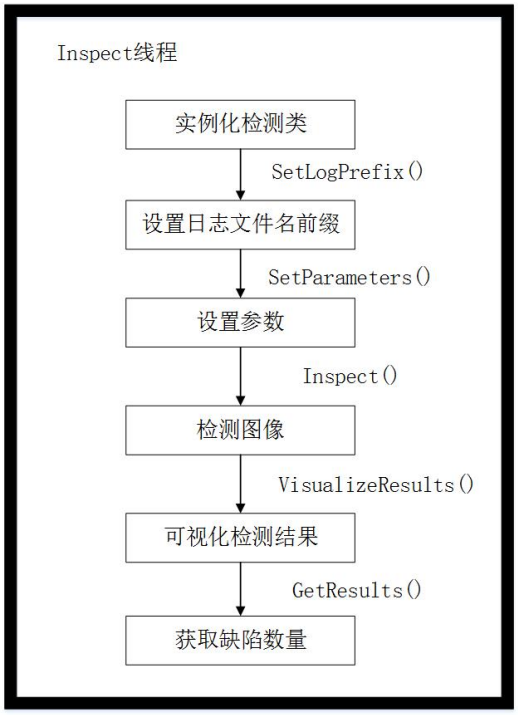

图 3-3　Inspect线程函数执行过程

实例化一个检测类Inspector之后，使用：

```
inspector->SetlogPrifix();
```

来调用图 3-3所表示流程的相关函数。检测过程中日志文件的作用是记录每一步的操作，以便在调试出错时能通过日志来快速定位发生错误的代码位置。设置参数即是把frame类已读取的参数进一步赋给Inspect类的。可视化检测结果即生成如2.1节所示的结果图像，将缺陷所在的矩形框绘制出来。

3.1.3 应用OpenMP的多线程并行

OpenMP预编译指令处理循环语句简化了并行化的编程，在Visual Studio中C++的应用方法如下代码所示[10]：

```
#pragma omp parallel for
for (int i = 0; i < contours.size(); i++)
{
    drawContours(bw_dst,contours,i,Scalar(255),CV_FILLED,8);
}
```

drawContours()是一个实现画轮廓的函数。其中参数bw_dst是保存检测结果图像的Mat变量，参数contours 是保存缺陷的Mat向量集合。通过一个for循环将contours上的轮廓画在bw_dst。在for循环之前加上OpenMP的预编译指令“#pragma omp parallel for”就能实现这个画图操作的并行执行。

OpenMP仍然存在着诸多限制。OpenMP从多指令并发来实现循环语句的并行执行，因此每个循环操作之间要保持低耦合才会适用；如上代码所示，每个循环的操作都不依赖于其它循环的运行结果，因此可以顺利执行并发。循环之间的耦合和依赖程度越高，自动并发的过程就越容易出错。

在循环内部修改循环计数也是OpenMP不允许的操作。因为OpenMP的每个循环计数和线程之间是一对一的关系；如果以上代码若在循环内存在“i–”操作，那么就会存在同一个循环计数对应多个线程的情况，就变成了一对多的关系。

并且OpenMP的循环计数不支持unsigned（无符号整型）类型的变量；因此不能使用size_t类型来作为循环计数。

```
//#pragma omp parallel for
for (size_t i = 0; i < hist_sz; i++)
{
    /*循环操作*/
}
```

size_t是一个和特定系统相关的无符号整型数，比如在32位系统和64位系统中，size_t占有的字节长度会各有不同[7]。而部分程序代码中使用size_t类型的变量来代替int 类型（如上代码所示），是出于跨平台的考虑,也就不能使用OpenMP进行并行处理。

3.2　基于OpenCV和CUDA的检测算法优化

3.2.1　代码移植的程序框架设计

原代码包含了两个类：框架类（CBGBFrame）和检测类（CBGBInspector）。框架类包含了一部分图片预处理流程，主要有：加权平均降噪、图片分离和有效区域提取。检测类包含了预处理、后处理（记录缺陷形成可视化的检测结果图）和核心的缺陷检测方法Inspect()。而框架类在图像检测部分，通过实例化一个检测类来进行检测。

为了方便代码的移植和测试，这里使用一个检测类的子类来一些子函数的GPU实现。完成一个GPU函数的编写后，可直接在子类中调用，取代原有的函数实现，而其它没有变更的函数则继承父类的方法。

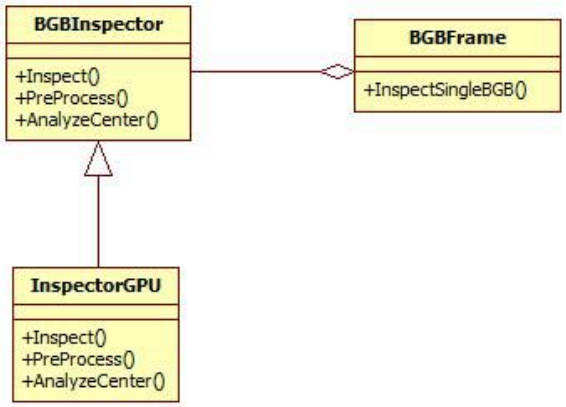

图 3-4　框架类、检测类及其子类

为了能方便控制程序启用和不启用GPU计算，使用了工厂模式的设计方法来对检测类CBGBInspector和InspectorBPU的实例化进行控制。

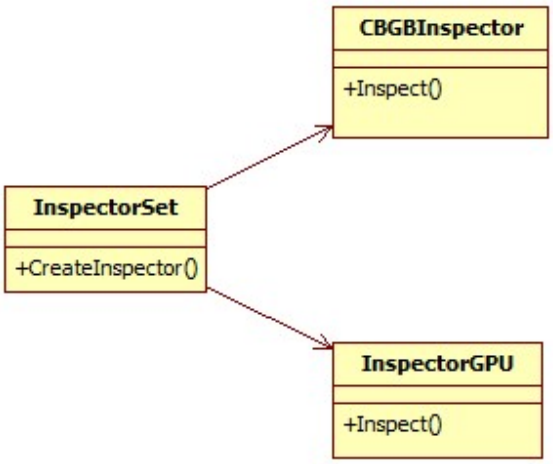

图 3-5　使用工厂类InspectorSet来生产检测类

工厂类InspectorSet的作用是生产检测类，在代码中定义如下的函数来实现这个功能。函数的返回类型是CBGBInspector的指针对象，而返回的类型是CBGBInspector和InspectorGPU两种。如图 3-4可知两者是继承关系，所以一下代码第九行是用父类来实例化子类，在C++中只能以“new”关键字来实现，而“new”关键字在对象实例化中作用的只能是指针对象，因此必须使用父类的对象指针作为返回值。其中CPU_MODE 和GPU_MODE是在头文件中进行的宏定义，在函数调用时作为条件判断依据。

```
#define CPU_MODE 0
#define GPU_MODE 1
CBGBInspector* InspectorSet::CreateInspector(int mode)
{
        CBGBInspector *inspector;
        if (mode == CPU_MODE)
                inspector = new CBGBInspector();
        else if (mode == GPU_MODE)
                inspector = new InspectorGPU();
        else
                inspector = NULL;
        return inspector;
}
```

以下代码是在Inspect线程中利用上述定义的工厂类的CreateInspector()方法来生产一个启用GPU的检测类。需要调用检测类的函数时，先定义一个检测类父类的指针，实例化一个工厂类InspectorSet的对象，再调用CreateInspector()返回一个需要的检测类。

```
int CBGBFrame::InspectSingleBGB()
{
//InspectorGPU inspector;//原来的对象实例化方式
//inspector.inspect();
InspectorSet set;
CBGBInspector *inspector = set.CreateInspector(GPU_MODE);
/*后续操作*/
delete inspector;
}
```

原来是用变量定义的方式来对检测类进行实例化，如上述的注释段代码。这种方式实例化使用的内存空间是栈区，由系统自动分配和释放；而使用“new”来进行实例化使用的是堆区，还要在最后用“delete”主动释放，避免在测试中可能会出现的内存泄漏。

此外，为了能让检测器对象成功调用子类InspectorGPU的Inspect()方法，必须把父类CBGBInspector的同名方法定义成虚函数。

```
public:
virtual int Inspect(Mat& im_original, Mat& mask_original);
}
```

而Inspect()内部会用到的其它工具函数不需要显示调用，因此不定义成虚函数也能正常调用。

3.2.2 检测算法中的重要工具函数

在上一节提到的检测函数Inspect()执行的效率影响到了整个程序；由Visual Studio的性能评估工具可得到该函数在单线程下的运行耗时占比达到了27.37%。图3-6是性能分析报告的一部分结果。

Function Name	Total CPU (%)
CBGBFrame::InspectSingleBGB	34.38 %
CBGBInspector::VisualizeResults	6.99 %
CBGBInspector::SetLogPrefix	0.02 %
CBGBInspector::Inspect	27.37 %
CBGBInspector::AnalyzeCenter	21.13 %
CBGBInspector::PreProcess	6.18 %
cv::Mat::copyTo	0.05 %
std::operator<<<std::char_trait...	0.02 %

图 3-6 Inspect()函数的耗时占比

Inspect()内的主要处理函数有预处理PreProcess() 和缺陷检测AnalyzeCenter()。两个过程中涉及到的主要工具函数有：

```
public:
int FitRectLines()                                    //拟合矩形Inspect
void PiecewiseSmooth()                                //分段平滑滤波
void Get4MarginRect()                                 //获取四个边缘图像
void ProcessCentralRegionBasedOnCanny()   //边缘检测的方法检测缺陷
bool ProcessCentralRegionBasedOnFilter() //平滑滤波的方法检测缺陷
void MeanFilter()                                     //均值滤波函数
}
```

笔者主要针对这些函数，使用OpenCV的CUDA模块来进行代码移植，达到用GPU来优化加速的目的。

3.2.3 关键函数代码移植示例

这里用分段平滑滤波函数的GPU实现PiecewiseSmooth_GPU()来说明算法移植的关键步骤。以下的代码说明简化了源代码不易于阅读的代码。

首先把需要的Mat类型的变量上传至显存（这里用构造函数来进行隐式的上传），保存在GpuMat类型的变量中：

```
cuda::GpuMat g_dst;    //用于保存处理结果
cuda::GpuMat g_imCrop(imCrop); //待处理图像
cuda::GpuMat g_maskCrop(maskCrop); //掩模版
}
```

定义cuda高斯滤波器和需要的中间变量：

```
1 Ptr<cuda::Filter> GaussianFilter;
2 cuda::GpuMat g_tempIm, g_tempMask;
3 }
```

分上、中、下三段对图片和掩模版进行高斯滤波，每一次循环结束完成后把该段

的滤波结果加到中间变量g_tempIm和g_tempMask。

```
for (size_t i = 0; i < 3; i++)
{
...
GaussianFilter = cuda::createGaussianFilter();
GaussianFilter->apply(g_imCrop.rowRange(), g_tempIm);
GaussianFilter = cuda::createGaussianFilter();
GaussianFilter->apply(g_maskCrop.rowRange(), g_tempMask);
cuda::add(g_fIm.rowRange(),g_tempIm.rowRange(),
                                g_fIm.rowRange());
cuda::add(g_fMask.rowRange(),g_tempMask.rowRange(),
                                g_fMask.rowRange());
}
```

该循环总体的过程如图 3-7：

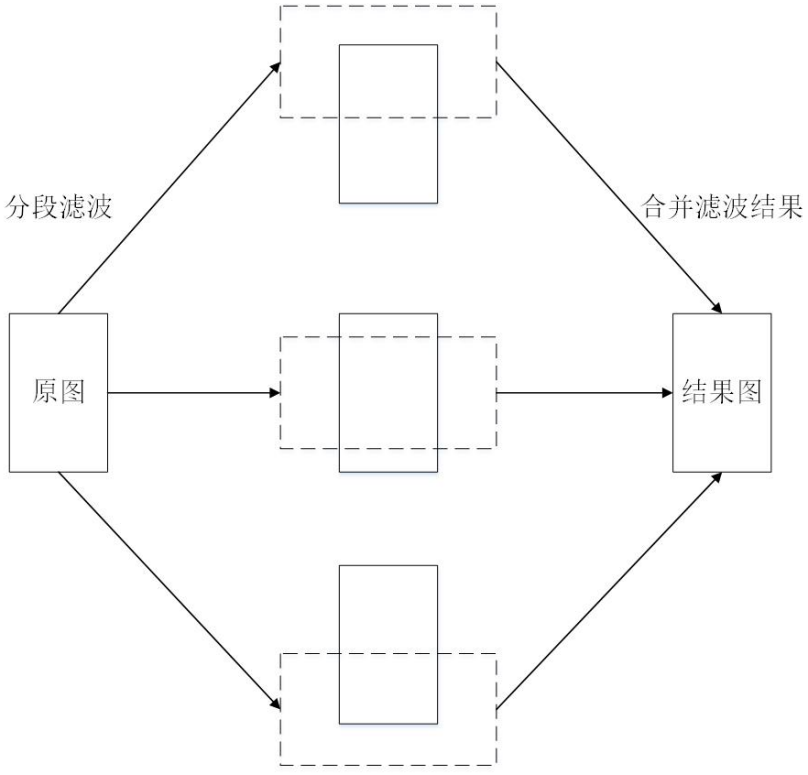

图 3-7　分段滤波循环的流程

图片的分段不是均分的，其中存在重叠的部分，新一轮的滤波结果在相加的过程中会直接覆盖上一次滤波结果和本次的重叠部分。

把滤波后的图片与滤波后的掩模版相除得到最终的滤波结果，并从显存下载至CPU。这里用download()方法显式进行了下载。

```
cuda::add(g_fMask, 1e-6, g_fMask);
cuda::divide(g_fIm, g_fMask, g_dst);
cuda::GpuMat convertedIm;
g_dst.convertTo(convertedIm, CV_8U);
convertedIm.download(dst);
```

其中在进行除操作之前，先给被除的”g_mask“加上一个极小数”1e-6“。在下载之前还有一个类型转换的操作convertTo()；一般OpenCV的Mat类型转换可以直接写：

```
dst.convertTo(convertedIm, CV_8U);
```

这里使用了一个GpuMat类型的中间变量convertedIm来完成转换的操作，因为cuda版本的convertTo输出参数不能为原对象。

对代码运行速度影响比较大的，除了滤波函数以外，涉及的图像处理方法还有各种形态学处理（开运算、闭运算和腐蚀等）、Canny边缘提取算法。

```
Ptr<cuda::Filter> morph = cuda::createMorphologyFilter();
morph->apply();
Ptr<cuda::CannyEdgeDetector> canny = cuda::createCannyEdgeDetector();
canny->detect();
```

形态学操作在CUDA模块中仍然以滤波器Filter为基类实现，可在构造模板指针的过程中指定运算类型（开运算、闭运算和腐蚀等）。而Canny边缘提取算法在CUDA模块中是一个独立的类CannyEdgeDetector，使用detect接口来应用算法。移植过程中最重要的是仔细查阅OpenCV提供的参考技术文档，熟悉和了解相关的应用函数接

口[11]。

3.2.4 CUDA设备初始化的处理

在2.2.2节中提到了CUDA设备的初始化问题。初始化相对于整个程序来说是一个很耗时的过程，而这个初始化其实只要在程序开始运行时进行一次就可以。因此对于一些过程简单而且不需要长期工作的图像处理程序来说，使用CUDA会使初始化的时间开销成为性能瓶颈。而在手机屏幕缺陷检测的项目中，程序要在生产线上长期运行，因此CUDA的初始化可以放在程序启动中完成，不会对正常的生产过程造成干扰。

这里使用一个类”CudaApi“来对其它平台提供CUDA的相关函数操作。

```
class CudaApi
{
public:
int getDeviceCount(); //获取支持cuda的GPU设备数量
bool initCuda();      //初始化cuda设备
};
```

initCuda()函数用来手动初始化CUDA，初始化成功则返回true。

```
bool CudaApi::initCuda()
{
        if (getDeviceCount() > 0)
        {
                cuda::setDevice(0);
                return true;
        }
        else return false;
}
```

getDeviceCount()函数用于获取可用的CUDA设备，成功时大部分情况下的返回值是1，即当前机器的GPU数量。初始化之前要执行这个函数以确定当前的环境可以正常启用CUDA。

```
int CudaApi::getDeviceCount()
{
        return cuda::getCudaEnabledDeviceCount();
}
```

因为该项目的工程师在前台使用C#设计界面，这里在DLL工程中提供对C#的接口：

```
extern "C" __declspec(dllexport) bool CudaInitialize();
bool CudaInitialize()
{
    CudaApi api;
    return api.initCuda();
}
```

这样前台的工程师就可以在程序启动时调用这个接口来手动初始化CUDA，避免CUDA的初始化开销影响到图像检测的效率。

3.3 本章小结

本章主要针对第二章的设计方案，介绍了CPU多线程优化整体程序在各个环节的具体实现，以及笔者如何设计程序框架来对原程序进行GPU的移植，在移植过程中剖析了检测算法的处理流程，以此来展现代码移植的细节处理。笔者把消除CUDA设备初始化对性能的影响，介绍了CUDA函数对外接口的设计和实现

4 并行化设计的设备环境和整体测试

4.1 OpenCV和CUDA环境搭建

4.1.1 软硬件环境

1）硬件：英伟达(NVIDIA) GT750M 2G DDR3独立显卡

2）库：OpenCV(Open Source Computer Vision Library)3.2.0

3）编程工具：Visual Studio 2013

4）Opencv编译工具：Cmake 3.6.3

5）运算平台：CUDA(Compute Unified Device Architecture) 8.0

6）操作系统：windows 10

在英伟达(NVIDIA)的官网(https://developer.nvidia.com/cuda-downloads) 上下载最新的CUDA 8.0并安装时，注意要选择适用于当前操作系统的版本，此过程可能会更新显卡驱动，安装后需重启。

4.1.2 Cmake重新编译OpenCV

可以在Cmake的图形界面下来实现OpenCV完整工程的编译，在CUDA环境搭建好的情况下，可编译出OpenCV用于GPU 图形计算的CUDA模块。1）源代码路径选择Opencv目录下的source文件夹，并选择Opencv 工程的生成路径；

2）编译工具选择Visual Studio 2013 win64；

3）点击Configure，再勾选WITH_CUDA；

4）再次Configure，点击Generate生成OpenCV的工程；

5）进入第一步选定的OpenCV工程的生成路径，打开OpenCV.sln，也就是打开Cmake生成的OpenCV工程，在Visual Studio下对工程进行编译；在debug模式和release 模式下分别编译一次，大概需要2-3 个小时的时间；

6）在工程的CMake Targets下，选择INSTALL模块右键点击Build Only Install生成OpenCV库；

7）进入OpenCV的工程生成目录，将install整个文件夹拷贝至opencv 安装目录下，将现有的build文件夹改为build.old，将install文件夹改为build，这样就可以把原来的库替代为编译好的带有CUDA模块的OpenCV 库。

4.1.3　Visual Studio 2013下对项目的配置方法

为了正常使用OpenCV及编译好的CUDA模块，需要在工程的配置管理器（Property Manager）进行配置，配置步骤如下（不分先后）：1）在VC++ Directories选项下，在包含目录(Include Directories) 填入：

F:/Program Files/opencv/build/include

F:/Program Files/opencv/build/include/opencv

F:/Program Files/opencv/build/include/opencv2

2）在库目录(Library Directories)填入：

F:/Program Files/opencv/build/x64/vc12/lib

3）在链接器（Linker）选项下选择输入（input），在附加依赖项（Additional Dependencies）填入如附录1 所示的库文件名。

4）以上是在debug下编译的配置方法，如果要在release下编译，则在release配置文件中重复以上1、2步；执行第3步时把以上所有库文件名复制一次，再去掉后缀“d”之后即release 下编译的库文件名。

5）编程时需要包含的头文件

OpenCV主要的库文件：

opencv2/opencv.hpp、opencv2/imgproc/imgproc.hpp；

Opencv3的CUDA模块根据需求可分别添加相应头文件，如：

cudaarithm.hpp：提供GpuMat对象的逻辑运算操作；

cudafilters.hpp：提供核心的滤波功能。

此外，为了避免测试过程中操作系统不必要的干预，应关闭微软图形驱动程序的超时检测与恢复Timeout Detection and Recovery (TDR)，防止GPU函数运行时被系统

终止并强制重启显卡驱动。具体操作是通过英伟达的Nsight Monitor工具，找到TDR选项，设置为false[8]。

4.2 CPU多线程优化的性能测试

本次测试中，Frame线程和Inspect线程数量最大同时为2个。首先把Frame线程数固定为1处理不覆膜组的图片,同表 中Frame 线程数为1的情况；在Inspect线程各为1 的情况，等价于在串行条件下处理一张经过左右分离的单边图像，各调用了一次框架类的预处理和检测类的检测算法。在Inspect 线程为2的情况，即是处理经分离后的两个单边图像，一共调用了一次框架类的预处理和两次检测算法。

然后再多开启一个Frame线程同时处理覆膜组和不覆膜组，同表 中Frame线程数为2的情况。Inspect线程为2的情况即是处理两组图片经过分离后的单边图像，各调用两次框架类的预处理和检测类的检测算法。Inspect线程数为4的情况即是对两组输入图像包括所有单边图像进行完整的处理，调用了两次框架类的预处理和四次检测类的检测算法（即对输入的两组图像进行左右分离后，再分别对左右两张图像进行处理）。

测试过程中所选用的图片即手机屏幕在工位上的样本图片，输入格式为bmp，大小固定为27.6MB，分辨率为6600x4400像素。测试的对象包括总时间开销和单个Inspect线程中检测函数inspect()的时间开销。

表 4-1　CPU多线程下检测算法的性能

线程数(Frame)	线程数(Inspect)	cpu平均时间/ms	inspect()函数平均时间/ms
1	1	4861	2063
1	2	5588	2450
2	2	5845	2420
2	4	7406	3550

从第二和第三行可知，增加一个Frame线程使开销只增加了257毫秒。因为Frame线程包含的处理过程在整个程序中的耗时比并不大；而Inspect线程包含

的检测函数的开销平均高达3500毫秒，是程序中耗时比最大的部分。

理想中采用多线程并行跟单线程串行对比，效率应该是成倍增长；也就是说，串行条件下每增加一个处理流程，总时间开销应增加一倍，而并行条件下增加一条线程，总时间开销则没什么变化。实际的情况从表 4-1的第一和第二行可以看出，增加一个Inspect线程使总时间开销增加了727毫秒；再看第三和第四行，增加两个Inspect线程使开销增加1561毫秒，也就是说每增加一个Inspect线程，则开销接近于线性增长。和串行条件下相比，并行的总体效率在这里比起串行还是要快非常多的。

4.3　在GPU下的图像检测算法的性能

对原有算法（算法主要处理过程在inspect()函数内）使用OpenCV的CUDA函数进行整体的移植完毕后，在Inspect线程下执行一次inspect()函数进行检测的性能测试结果如表 4-2。

表 4-2　在GPU下的图像检测算法的性能

线程数(Frame)	线程数(Inspect)	inspect()函数平均时间/ms	GPU下inspect()函数平均时间/ms
1	1	2063	1656
1	2	2450	2152
2	2	2420	2047
2	4	3550	3047

inspect()函数在GPU计算的辅助下，单线程条件下比原来的程序快了400毫秒，在四个线程全开的情况下快了约500毫秒；多线程下因为并发的关系，并不会单线程时的差距进一步拉开。

4.4　OpenCV中的CUDA模块存在的问题

CUDA模块的CUDA函数在特定情况下存在性能低下的问题。比如在检测算法中，会用到一个工具函数MeanFilter(const Mat& src, Mat& dst, int kernel_size)；这个函数包含了一个线性滤波操作filter2D()，对应的CUDA滤波器类生产方法

为createLinearFilter()。在算法中，MeanFilter()要连续调用两次，且每次的kernel_size参数为不同值。表 4-3测试了MeanFilter()函数和GPU下的MeanFilter_GPU()函数在不同kernel_size下的性能。

表 4-3　工具函数MeanFilter()的性能

kernel_size	MeanFilter() 平均时间/ms	MeanFilter_GPU() 平均时间/ms
145	219	16572
7	328	63

当kernel_size为7时，包含CUDA函数的MeanFilter_GPU()处理速度快了5倍之多。但是，当kernel_size为145时，CUDA函数的处理开销达到了16秒之多；在没有关闭TDR的情况下，程序跑到这里时,，由于运算超时，系统会重启图形驱动程序，从而导致程序崩溃。这是线性滤波函数本身的设计问题，没有考虑到这种极端情况下的优化。由于存在这种问题，在写代码的过程中就要做好性能的评估，有所取舍。

可以用如下的代码解决上述问题：

```
MeanFilter(im_filter , im_Big , wBig.height);
MeanFilter_GPU(im_filter , im_Small , wSmall.height);
```

其中wBig.height的值对应表 4-3中数值大小为145的kernel_size参数，使用原函数处理；而wSmall.height 对应数值7，使用GPU函数处理。从表 4-3可知这样两个函数的总时间开销是282毫秒，比起两个MeanFilter()处理快了48.4%。

4.5　应用GPU计算的CPU多线程并行的性能

在CPU多线程下调用GPU函数，程序同时在不同的设备上运行。执行普通代码是在CPU上；执行GPU函数则是在GPU上进行计算，计算结果再从GPU反馈到CPU。因为在CPU-GPU异构平台上运行，程序的稳定性和性能都是未知的，比较容易出现问题；比如在多线程下，有可能会在显存分配时出现冲突。表 4-4仿照CPU多线程并发的线程对比设置，在应用GPU的模式下进行了测试，并以表 4-1 的“CPU平均时

间”一项为基准计算加速比。 应用GPU计算使得总体的性能提高了约9%。因为本

表 4-4　多线程下应用GPU的检测算法性能测试

线程数(Frame)	线程数(Inspect)	平均时间/ms	加速比/%
1	1	4438	8.7
1	2	4902	12.3
2	2	5320	9.0
2	4	6727	9.1

次是借助OpenCV的CUDA模块来完成，主要针对一些通用的算法进行优化，而且由于CUDA模块并未十分成熟，在移植过程中会遇到部分CUDA函数出现严重延迟，因此不能全部使用GPU来处理。如果将更多算法借助CUDA进行移植，在CPU-GPU异构平台上，程序性能提升的空间依然很大。

4.6　本章小结

这一章介绍了本次毕设工作的平台环境的部署过程，以及测试过程中所使用的硬件条件。测试了程序在CPU多线程框架下的运行开销，于单线程的情况进行了简要的对比，可知使用多线程使程序性能得到显著提升。针对inspect() 函数的部分工具函数进行GPU加速优化后的进一步测试结果表明，GPU计算使得图像检测的整体性能有了略微的提升；而且使用GPU来进行加速仍然有着很大的性能改良潜力。

5 总结与展望

5.1 论文工作总结

本文从实际的需求出发，强调了工业自动化制造过程中实时性的重要性。笔者基于手机屏幕缺陷自动化检测的项目，调研和学习了如何加快图像检测速度的方法。

本文通过对手机缺陷检测工作流程的介绍，根据实际生产的需要分析了程序优化的可行方案，并设计了具体的多线程处理流程。探索了借助GPU计算优化图像处理算法的方案，确定了使用OpenCV的CUDA模块来移植函数。

接下来介绍了笔者对已有方案的具体实现，包括了在图像检测程序中具体实施的线程分配，GPU移植算法的程序框架和类的设计，以及具体的编程实现方法。

在测试过程中以线程类型和数量作为变量进行了详细的性能测试和对比分析，展示了工作的成果；同时指出了OpenCV的CUDA模块的不足，从测试中也得出了本次工作并没有利用GPU计算来达到显著的性能提升，但是对于GPU计算的应用在今后工作中仍然是重点。

5.2 下一步研究内容

OpenCV的CUDA模块仍然不成熟，使用CUDA代码来开发更优的代码是以后优化图像处理速度的趋势，因为这部分的性能瓶颈只能通过更优的GPU计算平台来突破；而GPU平台上通用算法的开发仍然需要长期的发展来完善。

本次工作只接触了CUDA架构，后续还可以结合OpenCL，OpenACC等平台来实现图像处理的GPU加速，并比较各平台的优劣。

参考文献

[1] 国务院印发.中国制造2025[EB/OL].http://news.xinhuanet.com/politics/2015-05/19/c_1115331338.html.

[2] 易松松. 基于机器视觉的手机面板缺陷检测方法研究[D].哈尔滨工业大学, 2016.

[3] 吕向阳. 基于CPU+GPU的图像处理异构并行计算研究[D].南昌大学, 2014.

[4] 王锋,杜云飞,陈娟. GPGPU性能模型研究[J]. 计算机工程与科学,2013,35(12):1-7.

[5] 刘鑫,姜超,冯存永.CUDA和OpenCV图像并行处理方法研究[J]. 测绘科学,2012,37(4):123-125.

[6] OpenCV官网CUDA主页面[EB/OL].http://opencv.org/platforms/cuda.html.

[7] 标准C++库参考文档[EB/OL].http://www.cplusplus.com/reference.

[8] Shane Cook.CUDA Programming_A Developer's Guide to Parallel Computing with GPUs[EB/OL].http://www.nvidia.com

[9] CUDA_C_Programming_Guide[EB/OL].http://www.nvidia.com

[10] Microsoft Developer Network.OpenMP Directives[EB/OL].https://msdn.microsoft.com/en-us/library/0ca2w8dk(v=vs.80).aspx

[11] OpenCV3.0参考文档[EB/OL].http://opencv.org/documentation.html