

西安交通大学

系（所） 计算机系  
系（所）主任 胡成臣  
批准日期 2017.02.20

## 毕业设计(论文)任务书

电子与信息工程学 院 计算机科学与技术 系 计算机 33 班 学生 张志源

毕业设计(论文)课题 CPU-GPU 异构平台上平面光源检测方法的并行化设计与实现

毕业设计(论文)工作自 2017 年 2 月 20 日起至 2017 年 6 月 20 日止

毕业设计(论文)进行地点： 西安交通大学计算机系统结构与网络研究所

课题的背景、意义及培养目标

本课题以国家智能制造“十三五”规划为背景，选题具有一定理论和实际应用价值。重点设计和实现 CPU-GPU 异构平台上平面光源检测方法的并行化方案，包括并行化检测方法和技术的调研、整体方案设计、编码实现、实验评测等。通过本次毕业设计，可以培养学生灵活运用所学计算机科学与技术知识解决实际问题的能力。

设计(论文)的原始数据与资料

1. 前期平面光源检测软件材料。

2. 并行化技术相关材料。

3. 前期实验数据。

课题的主要任务

1. 了解平面光源检测方法，调研 CPU-GPU 异构平台并行化技术。

2. 设计 CPU-GPU 异构平台上平面光源检测方法并行化整体方案。

3. 基于以上设计，进行代码实现，完成实验评测。

课题的基本要求(工程设计类题应有技术经济分析要求)

1. 掌握主流平面光源检测方法，独立完成 CPU-GPU 异构平台并行化技术调研。
  2. 独立设计基于 CPU-GPU 异构平台的平面光源检测并行化整体方案。
  3. 独立完成设计方案的编码实现和实验评测。
- 
- 
- 
- 

完成任务后提交的书面材料要求(图纸规格、数量，论文字数，外文翻译字数等)

1. 毕业论文 1 篇，正文字数不少于 15000 字，书写规范符合《西安交通大学 2017 届本科毕业设计（论文）工作手册》中关于论文质量规范的要求。
  2. 翻译英文论文一篇，中文译文字数 2000~3000 字。
- 
- 
- 
- 

主要参考文献

1. 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, 2015.
  2. 覃晓伟. 基于机器视觉的手机屏幕坏点检测系统研究[D]. 电子科技大学, 2012.
  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.
- 
- 
- 

指导教师\_\_\_\_\_吴茜媛\_\_\_\_\_

接受设计(论文)任务日期\_\_\_\_\_2017.02.20\_\_\_\_\_

（注：由指导教师填写）

学生签名：\_\_\_\_\_张志源\_\_\_\_\_

## 西 安 交 通 大 学

# 毕业设计(论文)考核评议书

电子与信息工程学院 院 计算机科学与技术 系(专业) 计算机 33 班级

指导教师对学生 张志源 所完成的课题为 CPU-GPU 异构平台上平面光源检测方法的并行化设计与实现

的毕业设计(论文)进行的情况,完成的质量及评分的意见: 论文围绕国家智能制造“十三五”规划的智能检测方向开展工作,选题具有一定的理论意义和实用价值。

论文基于 CPU-GPU 异构平台,以平面光源检测中的手机缺陷检测为例,设计和实现了一套并行化检测方法,整体框架基于 CPU 的多线程进行并行化,局部函数和代码基于 GPU 进行并行化。经测试,该方法能提高检测的实时性。

张志源同学在本科毕业设计期间,认真严谨,工作积极,进度良好。论文阐述条理清楚,数据翔实,达到本科毕业论文要求,同意进行答辩。

指导教师建议成绩: 良好

指导教师 吴茜媛

2017 年 6 月 10 日

## 毕业设计(论文)评审意见书

评审意见: 论文基于手机屏幕缺陷自动化检测的项目,在调研图像检测方法的基础上,设计和实现了多线程处理机制。该机制使用 OpenCV 的 CUDA 模块在 GPU 上进行并行处理,以提高检测效率,满足工业生产过程中的要求。论文对所设计实现方法进行了相应的性能测试和对比分析。其工作具有一定的实用价值,同意答辩。建议成绩为良。

评阅教师建议成绩: 良好

评阅人 张克旺 职称 讲师

2017 年 6 月 10 日

# 毕业设计(论文)答辩结果

电子与信息工程学院

计算机科学与技术系(专业)

毕业设计(论文)答辩组对学生\_\_\_\_\_张志源\_\_\_\_\_所完成的课题为

CPU-GPU 异构平台上平面光源检测方法的并行化设计与实现

的毕业设计(论文)经过答辩,其意见为\_\_\_\_\_

并确定成绩为\_\_\_\_\_

毕业设计(论文)答辩组负责人\_\_\_\_\_

答辩组成员\_\_\_\_\_

\_\_\_\_\_年 月 日

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。本文以平面光源检测方法中典型的手机屏幕缺陷的自动化检测为例，主要研究了图像处理过程中的程序优化方法，采用在 CPU 和 GPU 异构平台上进行并行化优化，从而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: Parallel Design and Implementation of Planar Light Source Detect Method on CPU-GPU Heterogeneous Platform**

**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, by using multi-threaded parallel programming method to optimize the original program according to the actual needs. In order to solve the bottleneck of CPU computing power, GPU high concurrent computing is chosen to improve the performance of some image processing functions.

In the multi-threaded parallel program, designed two types of threads according to the program's workflow, explains how to allocate threads, and implement parallel processing for loop?code in the support of OpenMP. 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
1.1 背景与意义.....	1
1.2 研究现状.....	1
1.3 论文主要内容.....	2
1.4 论文组织框架.....	2
2 手机屏幕缺陷检测流程概况和并行优化方案.....	4
2.1 手机屏幕缺陷检测的工作流程概况.....	4
2.1.1 读取算法参数.....	4
2.1.2 图像采集和输入.....	5
2.1.3 图像检测.....	7
2.2 手机屏幕缺陷检测方法的整体优化方案.....	7
2.2.1 基于 CPU 平台的并行优化.....	8
2.2.2 基于 GPU 平台的检测算法优化.....	10
2.3 本章小结.....	13
3 CPU 多线程并行和 GPU 加速的优化实现.....	14
3.1 CPU 多线程并行优化算法的实现.....	14
3.1.1 CPU 多线程并行优化算法的实现.....	14
3.1.2 Frame 线程的执行过程和 Inspector 线程分配.....	15
3.1.3 应用 OpenMP 的多线程并行.....	18
3.2 基于 OpenCV 和 CUDA 的检测算法优化.....	19
3.2.1 代码移植的程序框架设计.....	19
3.2.2 检测算法中的重要工具函数.....	21
3.2.3 关键函数代码移植示例.....	22
3.2.4 CUDA 设备初始化的处理.....	24
3.3 本章小结.....	25
4 并行化设计的设备环境和整体测试.....	26
4.1 OpenCV 和 CUDA 环境搭建.....	26
4.1.1 软硬件环境.....	26
4.1.2 Cmake 重新编译 OpenCV.....	26
4.1.3 Visual Studio 2013 下对项目的配置方法.....	26
4.2 CPU 多线程优化的性能测试.....	27

4.3 在 GPU 下的图像检测算法的性能.....	28
4.4 OpenCV 中的 CUDA 模块存在的问题.....	29
4.5 应用 GPU 计算的 CPU 多线程并行的性能.....	29
4.6 本章小结.....	30
5 总结与展望.....	31
5.1 论文工作总结.....	31
5.2 下一步研究内容.....	31
参考文献.....	32
附录 1：配置文件的算法参数说明.....	33
附录 2：配置 OpenCV 3.2.0 所需的库文件名.....	34
附录 3：外文原文与翻译.....	35
致 谢.....	41



# 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业，改变中国制造业“大而不强”的局面<sup>[1]</sup>。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<sup>[2]</sup>。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<sup>[3]</sup>。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）<sup>[4]</sup>。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<sup>[3]</sup>。

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<sup>[6]</sup>。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。

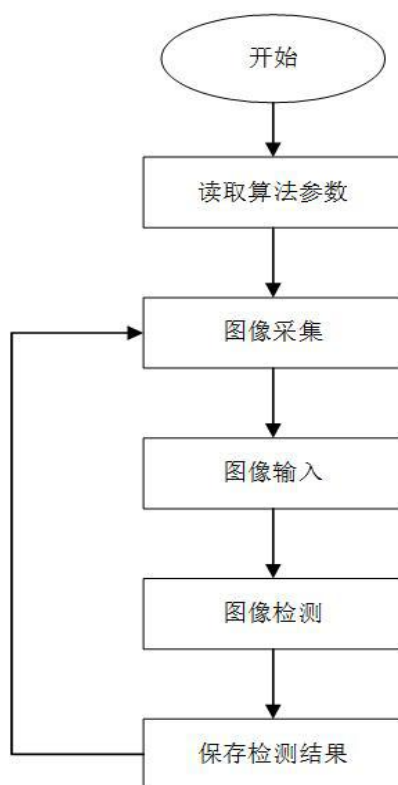


图 2-1 手机屏幕缺陷检测整体流程

#### 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，程序中的算法参数使用结构体保存。读取算法参数时则将文件读入并逐行识别字符。配置文件的书写格式如

下所示：

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

读取到“system”字符后，可判断出以下字符代表的是系统参数，根据设置的值给系统参数结构体的相应元素赋值。具体算法参数说明可参考附录 1。使用配置文件来调整程序中用到的参数值，主要目的是为了更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。

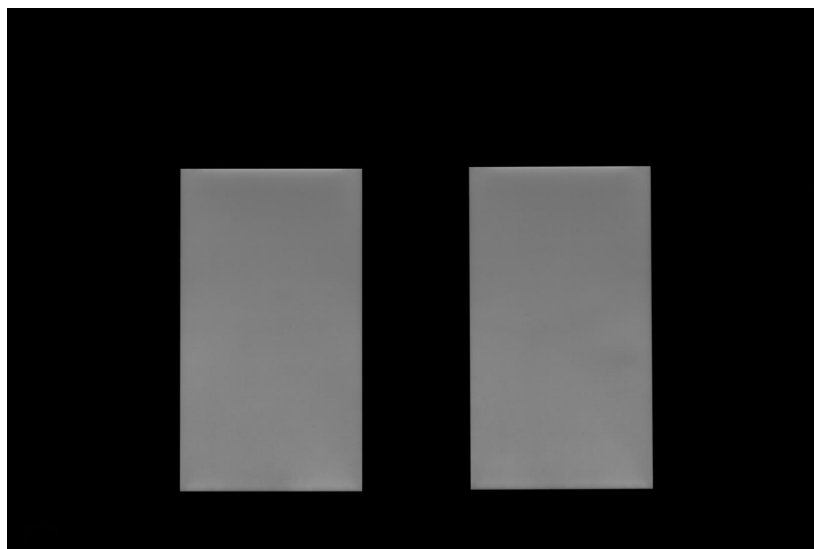


图 2-2 自发光条件下拍摄的图片

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。

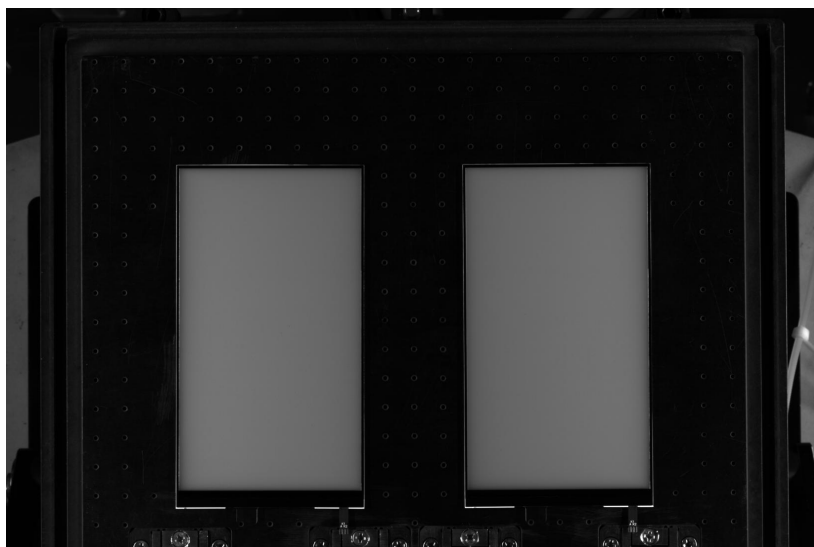


图 2-3 外光源条件下拍摄的图片

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 有效区域掩模版（左）

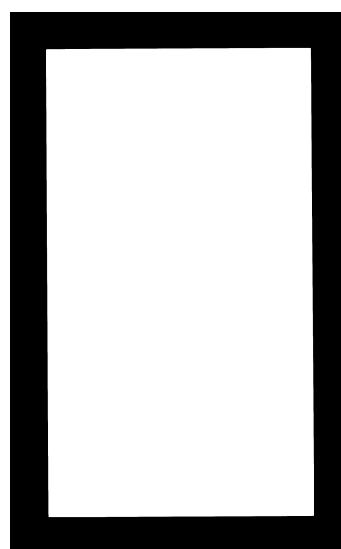


图 2-5 分离后的图像（左）

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<sup>[2]</sup>。

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。

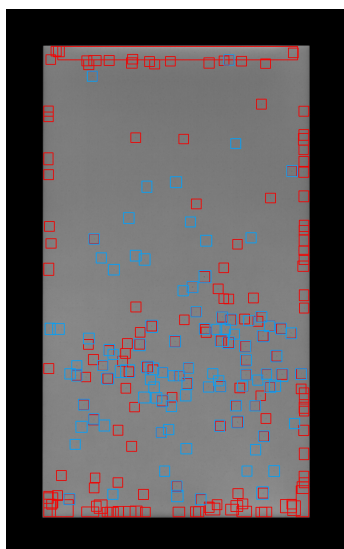


图 2-6 图像检测结果（左）

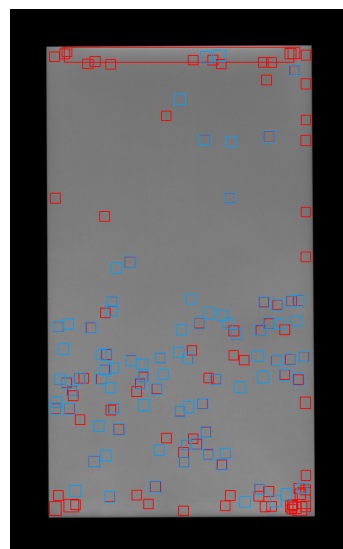


图 2-7 图像检测结果（右）

## 2.2 手机屏幕缺陷检测方法的整体优化方案

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，如图 2-8。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。

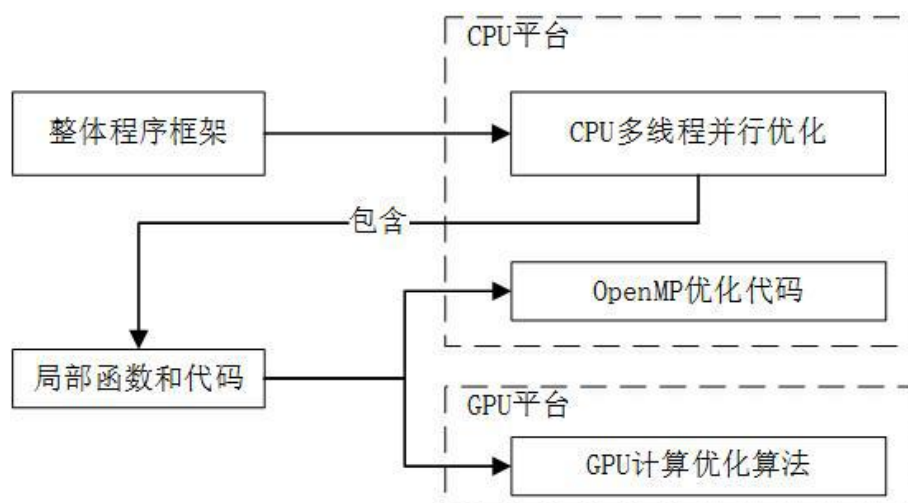


图 2-8 整体优化方案

### 1) CPU 多线程并行

该方案也是目前正在适用中的优化方案，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。

## 2.2.1 基于 CPU 平台的并行优化

### 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，线程类型和分配如图 2-9。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。

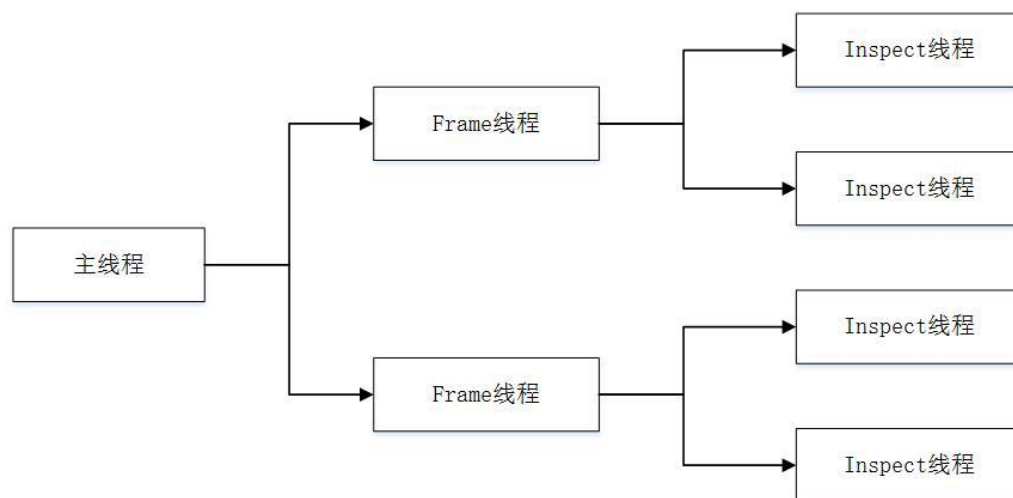


图 2-9 线程的类型和分配

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 线程的关系以及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。

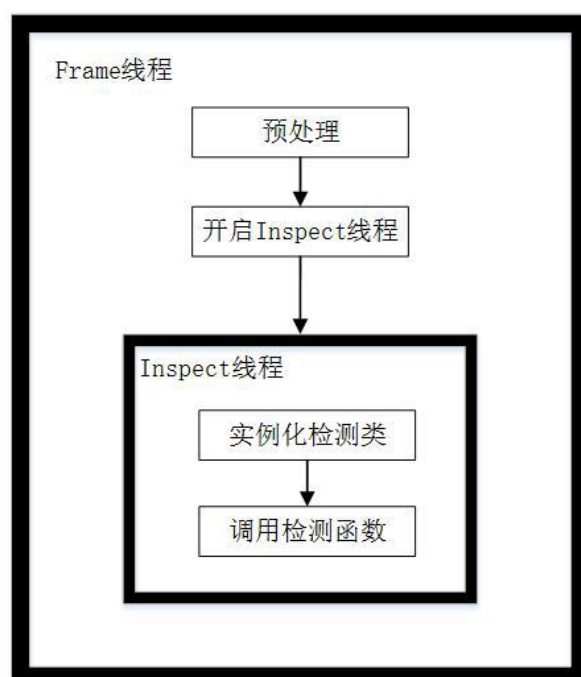


图 2-10 线程 Frame 和 Inspect 的关系

## 2) 基于 OpenMP 的局部代码并行优化

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 平台的检测算法优化

#### 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<sup>[8]</sup>。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<sup>[5]</sup>。如下使用了 CUDA 代码实现了一个 CUDA 函数<sup>[9]</sup>：

```
__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介于 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 的运算速度有多快,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。

表 2-1 CPU 和 GPU 高斯滤波的处理速度对比

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

接下来使用不同大小的 kernel 来测试，如表 2-2。

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	分辨率	上传/ms	下载/ms	kernel 大小	GPU 运算/ms	CPU 运算/ms
0	512*512	0	0	3	0	16
1	6600*4400	31	43	3	129	534
1	6600*4400	31	15	5	125	843
1	6600*4400	31	36	7	125	1687
1	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。说明并行计算的规模越大，GPU 的计算效率就越高。

通过测试，可知 OpenCV 的 CUDA 函数确实能利用 GPU 来对图像处理算法进行并行优化，有效提升性能。

## 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

本章根据第二章提出的方案来具体实施手机屏幕缺陷检测方法的并行化，从 C++ 多线程并发、线程类型和分类、OpenMP 的并行化应用和 OpenCV 的 CUDA 模块应用几个方面展开。

#### 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 CPU 多线程并行优化算法的实现

根据实际需求，要在主线程中开启两个 `Frame` 线程。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，执行过程如图 3-1 所示。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。

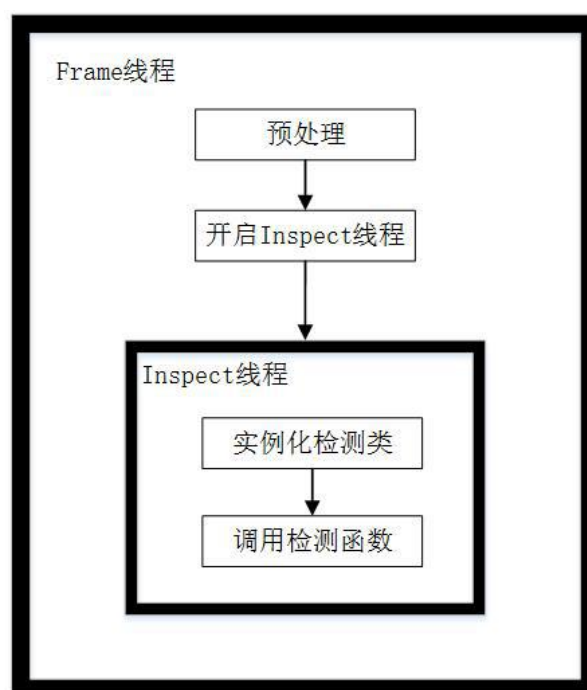


图 3-1 主线程函数执行过程

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。

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。



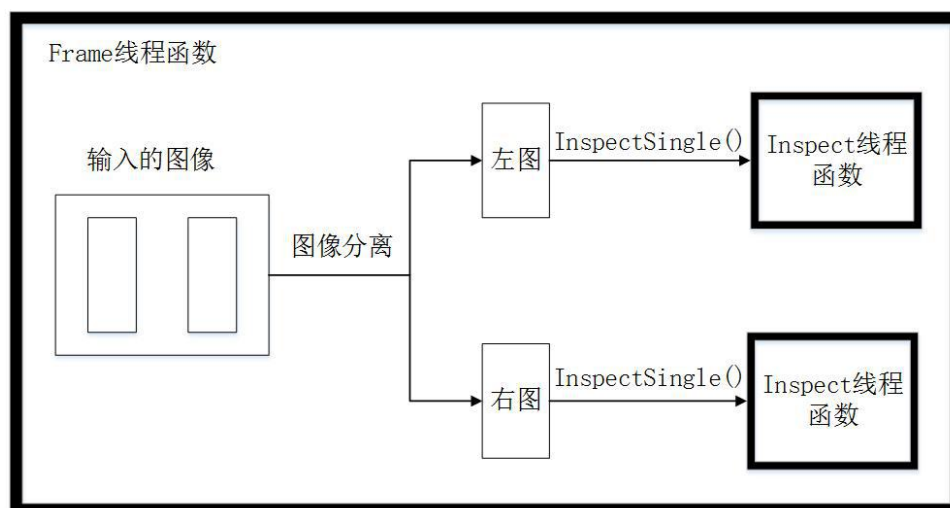


图 3-2 Frame 线程函数执行过程

线程内的 InspectSingleBGB()函数的执行过程如图 3-3:

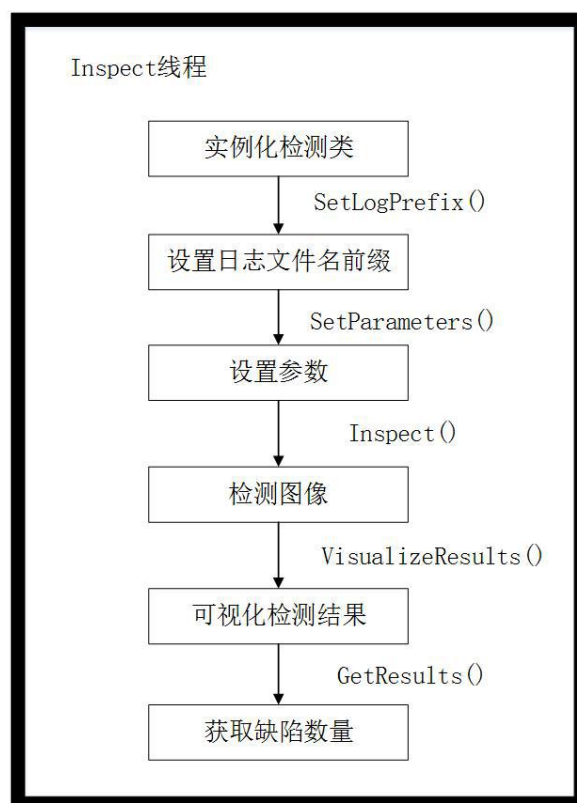


图 3-3 Inspect 线程函数执行过程

实例化一个检测类 Inspector 之后，使用以下方法来调用图 3-3 所表示流程的相关函数：

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

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<sup>[10]</sup>：

```
#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一一对应的关系；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<sup>[7]</sup>。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完成 GPU 函数的扩展，如图 3-4。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。

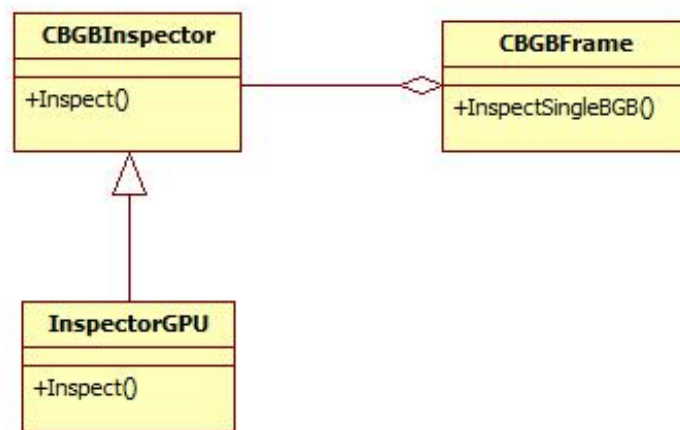


图 3-4 框架类、检测类及其子类

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 `InspectorGPU` 的实例化进行控制，如图 3-5。

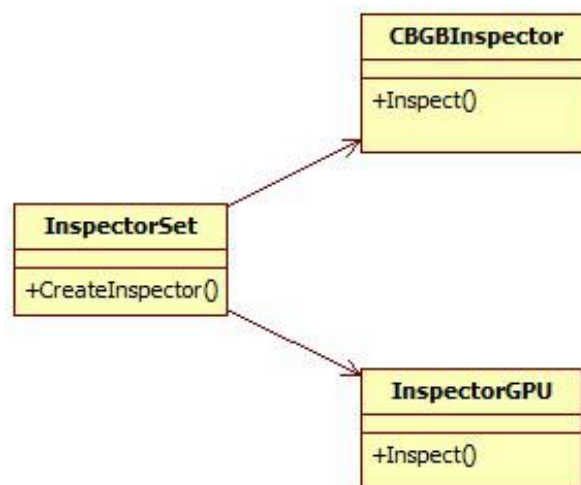


图 3-5 使用工厂类 `InspectorSet` 来生产检测类

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，所以以下代码第九行是用父类来实例化子类。

```
#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;
}
```

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。

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。

```
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);
    delete inspector;//不需要时主动释放
}
```

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。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 %

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：

```
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，使之容易阅读和理解。

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：

```
Ptr<cuda::Filter> GaussianFilter;
cuda::GpuMat g_tempIm, g_tempMask;
```

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：

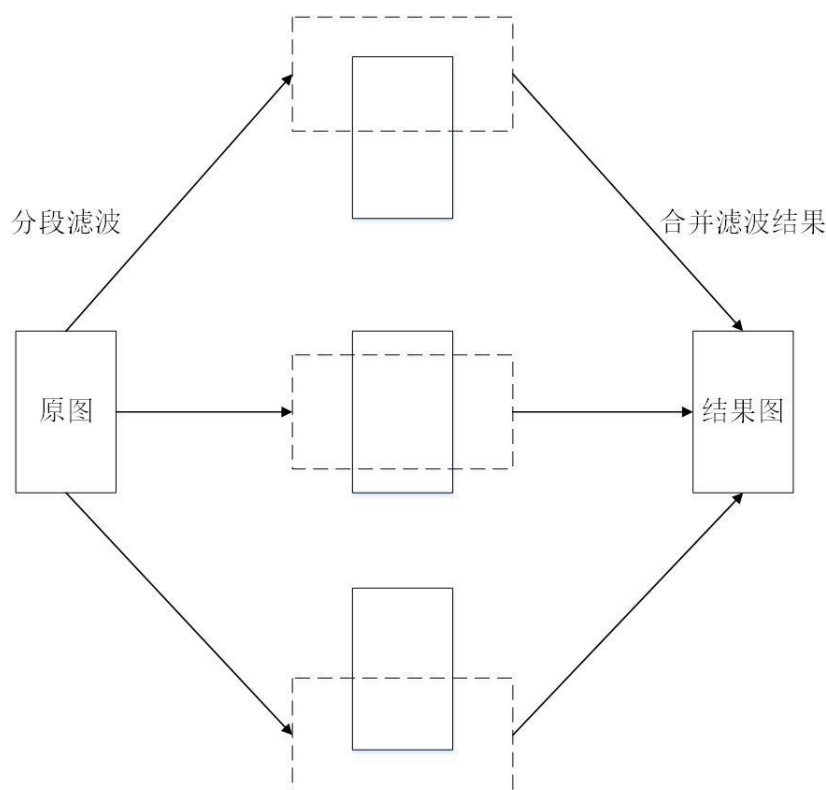


图 3-7 分段滤波循环的流程

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，熟悉和了解相关的函数接口<sup>[11]</sup>。

### 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。

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)
    {
        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();
}
```

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。

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

本章介绍了开发环境的搭建，包括 OpenCV 和 CUDA 计算平台，并针对第三章设计实现的结果进行了性能测试和分析。

### 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 2 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 步时把附录 2 所示的库文件名复制一次, 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;  
 Opendv3 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<sup>[8]</sup>。

## 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 4-1 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 4-1 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主要为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)	总时间/ms	检测算法执行时间/ms	平均检测时间/ms
1	1	4861	2063	4861
1	2	5588	2450	2794
2	2	5848	2420	2922
2	4	7406	3550	1851

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；同时，从第四列的数据可看出，随着 Inspect 线程增加，每个线程下的检测算法的效率都收到了 CPU 调度的影响。

和串行条件下相比，比较每个图的平均检测时间可以看出，并行的总体效率是更高的；在正常运作（开启 2 个 Frame 线程和 4 个 Inspect 线程）的情况下，平均的检测效率提高了 61.9%。

### 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 Inspect 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，并不会使在单线程下产生的差距进一步拉开。

## 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，且每次的滤波操作需要的核大小不同，即每次的 `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	16573
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。

例如在多线程下，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。由表中可得应用 GPU 计算使得总体的性能相较 CPU 提高了约 9%，显然这不能达到理想中性能瓶颈的突破。

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

因为本次是借助 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，与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](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[CP/OL].<http://www.cplusplus.com/reference>.
- [8] Shane Cook.CUDA Programming\_A Developer's Guide to Parallel Computing with GPUs[F/OL].<http://www.nvidia.com>.
- [9] CUDA\_C\_Programming\_Guide[F/OL].<http://www.nvidia.com>.
- [10] Microsoft Developer Network.OpenMP Directives[CP/OL].  
[https://msdn.microsoft.com/en-us/library/0ca2w8dk\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/0ca2w8dk(v=vs.80).aspx).
- [11] OpenCV3.0参考文档[CP/OL].<http://opencv.org/documentation.html>.



## 附录 1：配置文件的算法参数说明

表 附录 1-1 部分算法参数说明

参数名	类型	含义
captured_img_width	int	相机直接拍摄的图像宽度
captured_img_height	int	相机直接拍摄的图像高度
left_BGB_x0	int	左边背光板的默认位置和范围
left_BGB_y0		
left_BGB_width		
left_BGB_height		
right_BGB_x0		
Right_BGB_y0	int	右边背光板的默认位置和范围
Right_BGB_width		
Right_BGB_height		
BGB\_position\_tolerance	Int	背光板位置的误差
BGB_image_margin_width	int	找到背光板区域后，向外扩展的宽度
Num_img_denoise	int	用来做多图去噪的图像个数
BGB_segment_thr	int	用于分割背光板的阈值
Resize_ratio	double	用于对图像行进缩放的比率

## 附录 2：配置 OpenCV 3.2.0 所需的库文件名

```
opencv_calib3d320d.lib
opencv_core320d.lib
opencv_cudaarithm320d.lib
opencv_cudabgsegm320d.lib
opencv_cudacodec320d.lib
opencv_cudafeatures2d320d.lib
opencv_cudafilters320d.lib
opencv_cudaimgproc320d.lib
opencv_cudalegacy320d.lib
opencv_cudaobjdetect320d.lib
opencv_cudaoptflow320d.lib
opencv_cudastereo320d.lib
opencv_cudawarping320d.lib
opencv_cudev320d.lib
opencv_features2d320d.lib
opencv_flann320d.lib
opencv_highgui320d.lib
opencv_imgcodecs320d.lib
opencv_imgproc320d.lib
opencv_ml320d.lib
opencv_objdetect320d.lib
opencv_photo320d.lib
opencv_shape320d.lib
opencv_stitching320d.lib
opencv_superres320d.lib
opencv_video320d.lib
opencv_videoio320d.lib
opencv_videostab320d.lib
```

## 附录 3：外文原文与翻译

### 外文原文：

CUDA is an extension to the C language that allows GPU code to be written in regular C. The code is either targeted for the host processor (the CPU) or targeted at the device processor (the GPU). The host processor spawns multithread tasks (or kernels as they are known in CUDA) onto the GPU device. The GPU has its own internal scheduler that will then allocate the kernels to whatever GPU hardware is present. We'll cover scheduling in detail later. Provided there is enough parallelism in the task, as the number of SMs in the GPU grows, so should the speed of the program.

However, herein hides a big problem. You have to ask what percentage of the code can be run in parallel. The maximum speedup possible is limited by the amount of serial code. If you have an infinite amount of processing power and could do the parallel tasks in zero time, you would still be left with the time from the serial code part. Therefore, we have to consider at the outset if we can indeed parallelize a significant amount of the workload.

NVIDIA is committed to providing support to CUDA. Considerable information, examples, and tools to help with development are available from its website at <http://www.nvidia.com> under CudaZone. CUDA, unlike its predecessors, has now actually started to gain momentum and for the first time it looks like there will be a programming language that will emerge as the one of choice for GPU programming. Given that the number of CUDA-enabled GPUs now number in the millions, there is a huge market out there waiting for CUDA-enabled applications.

There are currently many CUDA-enabled applications and the list grows monthly. NVIDIA showcases many of these on its community website at [http://www.nvidia.com/object/cuda\\_apps\\_flash\\_new.html](http://www.nvidia.com/object/cuda_apps_flash_new.html). In areas where programs have to do a lot of computational work, for example, making a DVD from your home movies (video transcoding), we see most mainstream video packages now supporting CUDA. The average speedup is 5 to 10 times in this domain.

### ALTERNATIVES TO CUDA:

#### 1) OpenCL

So what of the other GPU manufacturers, ATI (now AMD) being the prime example. AMD's product range is as impressive as the NVIDIA range in terms of raw computer power. However, AMD brought its stream computing technology to the marketplace a long time after NVIDIA brought out CUDA. As a consequence, NVIDIA has far more applications available for CUDA than AMD/ATI does for its competing stream technology.

OpenCL and Direct compute is not something we'll cover in this book, but they deserve a

mentioned in terms of alternatives to CUDA. CUDA is currently only officially executable on NVIDIA hardware. While NVIDIA has a sizeable chunk of the GPU market, its competitors also hold a sizeable chunk. As

developers, we want to develop products for as large a market as possible, especially if we're talking about the consumer market. As such, people should be aware there are alternatives to CUDA, which support both NVIDIA's and others' hardware.

OpenCL is an open and royalty-free standard supported by NVIDIA, AMD, and others. The OpenCL trademark is owned by Apple. It sets out an open standard that allows the use of compute devices. A compute device can be a GPU, CPU, or other specialist device for which an OpenCL driver exists. As of 2012, OpenCL supports all major brands of GPU devices, including CPUs with at least SSE3 support.

Anyone who is familiar with CUDA can pick up OpenCL relatively easily, as the fundamental concepts are quite similar. However, OpenCL is somewhat more complex to use than CUDA, in that much of the work the CUDA runtime API does for the programmer needs to be explicitly performed in OpenCL. You can read more about OpenCL at <http://www.khronos.org/opencl/>. There are also now a number of books written on OpenCL. I'd personally recommend learning CUDA prior to OpenCL as CUDA is somewhat of a higher-level language extension than OpenCL.

## 2) DirectCompute

DirectCompute is Microsoft's alternative to CUDA and OpenCL. It is a proprietary product linked to the Windows operating system, and in particular, the DirectX 11 API. The DirectX API was a huge leap forward for any of those who remember programming video cards before it. It meant the developers had to learn only one library API to program all graphics cards, rather than write or license drivers for each major video card manufacturer.

DirectX 11 is the latest standard and supported under Windows 7. With Microsoft's name behind the standard, you might expect to see some quite rapid adoption among the developer community. This is especially the case with developers already familiar with DirectX APIs. If you are familiar with CUDA and DirectCompute, then it is quite an easy task to port a CUDA application over to Direct-Compute. According to Microsoft, this is something you can typically do in an afternoon's work if you are familiar with both systems. However, being Windows centric, we'll exclude DirectCompute from many high-end systems where the various flavors of UNIX dominate.

Microsoft are also set to launch C++ AMP, an additional set of standard template libraries (STLs), which may appeal more to programmers already familiar with C++-style STLs.

## CPU ALTERNATIVES:

The main parallel processing languages extensions are MPI, OpenMP, and pthreads if you are developing for Linux. For Windows there is the Windows threading model and OpenMP. MPI and pthreads are supported as various ports from the Unix world.

1) MPI (Message Passing Interface) is perhaps the most widely known messaging interface. It is process-based and generally found in large computing labs. It requires an administrator to configure the installation correctly and is best suited to controlled environments. Parallelism is expressed by spawning hundreds of processes over a cluster of nodes and

explicitly exchanging messages, typically over high-speed network-based communication links (Ethernet or InfiniBand). MPI is widely used and taught. It's a good solution within a controlled cluster environment.

2)OpenMP (Open Multi-Processing) is a system designed for parallelism within a node or computer system. It works entirely differently, in that the programmer specifies various parallel directives through compiler pragmas. The compiler then attempts to automatically split the problem into N parts, according to the number of available processor cores. OpenMP support is built into many compilers, including the NVCC compiler used for CUDA. OpenMP tends to hit problems with scaling due to the underlying CPU architecture. Often the memory bandwidth in the CPU is just not large enough for all the cores continuously streaming data to or from memory.

3)Pthreads is a library that is used significantly for multithread applications on Linux. As with OpenMP, pthreads uses threads and not processes as it is designed for parallelism within a single node. However, unlike OpenMP, the programmer is responsible for thread management and synchronization. This provides more flexibility and consequently better performance for well-written programs.

4)ZeroMQ (0MQ) is also something that deserves a mention. This is a simple library that you link to, and we will use it later in the book for developing a multinode, multi-GPU example. ZeroMQ supports thread-, process-, and network-based communications models with a single crossplatform API. It is also available on both Linux and Windows platforms. It's designed for distributed computing, so the connections are dynamic and nodes fail gracefully.

5)Hadoop is also something that you may consider. Hadoop is an open-source version of Google's MapReduce framework. It's aimed primarily at the Linux platform. The concept is that you take a huge dataset and break (or map) it into a number of chunks. However, instead of sending the data to the node, the dataset is already split over hundreds or thousands of nodes using a parallel file system. Thus, the program, the reduce step, is instead sent to the node that contains the data. The output is written to the local node and remains there. Subsequent MapReduce programs take the previous output and again transform it in some way. As data is in fact mirrored to multiple nodes, this allows for a highly fault-tolerant as well as high-throughput system.

## 外文译文：

CUDA 是 C 语言的扩展，允许 GPU 代码以常规 C 语言编写。代码是针对中央处理器（CPU）或针对图形处理器（GPU）的。该主机处理器在 GPU 设备上产生多线程任务（或 CUDA 中已知的内核）。GPU 具有自己的内部调度器，然后将内核分配给当下的任何 GPU 硬件。稍后将详细介绍调度。如果在任务中有足够的并行性，程序的速度也应该如 GPU 中的 SM（流处理器）的数量增长。

但是，这里隐藏着一个大问题。你必须了解程序中可以并发运行的代码的百分比。最大的加速可能受到串行代码数量的限制。如果你有无限计算处理能力，可以在瞬间完成并行任务，您仍然要留下一些时间来运行串行部分的代码。因此，如果我们确实可以并行化，我们必须首先考虑大量的计算量。

NVIDIA 致力于为 CUDA 提供支持。值得参考的信息、例子和帮助开发的工具可从网站 <http://www.nvidia.com> 的 CudaZone 模块中下载。CUDA 与其前身不同，现在实际上已经开始获得势头，这是第一次看起来会有一种编程语言将成为 GPU 编程的首选。鉴于支持 CUDA 的 GPU 的数量现在达到数百万，所以有一个巨大的市场在那里等待支持 CUDA 的应用程序。

目前有许多支持 CUDA 功能的应用程序，并且这个数量日益增长。NVIDIA 在社区网站 [http://www.nvidia.com/object/cuda\\_apps\\_flash\\_new.html](http://www.nvidia.com/object/cuda_apps_flash_new.html) 上展示了许多这些应用程序。在程序必须做很多计算工作的领域，例如从你的家庭电影制作 DVD（视频转码），我们现在看到的许多主流视频包都支持 CUDA。CUDA 在这些领域的平均加速时间是 5 到 10 倍。

### CUDA 的替代品：

#### 1) OpenCL

那么其他的 GPU 制造商呢，ATI（现在的 AMD）就是最好的例子。AMD 的产品范围与 NVIDIA 的原始计算机功率一样令人印象深刻。不过 AMD 推出了它的流计算技术到市场上，是在 NVIDIA 推出 CUDA 很长一段时间的之后。结果，相较于 AMD/ATI 流计算技术，NVIDIA 有更多的可用于 CUDA 的应用程序。

OpenCL 和 Direct 计算不是我们在本书中所讨论的，但是值得一提在 CUDA 的替代品方面。CUDA 目前仅在 NVIDIA 硬件上正式可执行。虽然 NVIDIA 在 GPU 市场中占有相当大的一部分，但其竞争对手也拥有相当大的一部分。作为开发商，我们希望开发尽可能大的市场的产品，特别是当我们在谈论消费市场的话题。因此，人们应该意识到有 CUDA 的替代品会同时支持 NVIDIA 和其他硬件。

熟悉 CUDA 的人可以比较容易地掌握 OpenCL，它们在基础概念上非常相似。然而，OpenCL 在使用上比 CUDA 要复杂得多，程序员需要在 OpenCL 中显式执行 CUAruntime API 所做的工作。你可以在 <http://www.khronos.org/opencvl/> 上阅读更多关

于 OpenCL 的信息。现在还有大量有关 OpenCL 的书籍。在 CUDA 之前, 我个人推荐在 OpenCL 之前学习 CUDA, 因为 CUDA 对于 OpenCL 来说是更高级别的语言扩展。

## 2) DirectCompute

DirectCompute 是微软替代 CUDA 和 OpenCL 的平台。它是一个与 Windows 操作系统关联的专有的产品, 特别是 DirectX 11 API。对于任何之前编写过视频卡的人来说, DirectX API 是一个巨大的飞跃。这意味着开发人员必须只学习一个 API 库来编写所有显卡, 而不是给每个主要视频卡制造商的驱动程序编写程序或发布许可。

DirectX 11 是最新的标准, 并支持 Windows 7。在这个标准里有 Microsoft 的名头, 你可以在开发人员社区中看到一些相当迅速的意见采用, 特别是这个开发者已经熟悉 DirectX API 的情况。如果你熟悉 CUDA 和 DirectCompute, 那么将 CUDA 应用程序移植到 DirectCompute 是很简单的。根据微软的说法, 如果你这样做, 你通常可以在一个下午通过一些功课来熟悉两个系统。然而, 以 Windows 为中心, 我们将从 UNIX 主导的许多高端系统排除 DirectCompute。微软还将推出一套额外的标准模板库 (STL) C++AMP, 这可能吸引更多熟悉 C++ 风格 STL 的程序员。

## CPU (解决方案) 的选择:

目前主要的并行处理语言扩展是 MPI 和 OpenMP, 如果你是 Linux 开发者则会用到 pthreads。对于 Windows, 有 Windows 线程模型和 OpenMP。MPI 和 pthreads 在 Unix 可为各种端口提供支持。

1) MPI (消息传递接口) 可能是最广为人知的消息接口。它是基于过程的, 并且在大型计算实验室中普遍应用。它需要一个管理员正确配置安装, 最适合受控环境。并行度是通过在一组节点上产生数百个进程并明确的进行消息交换, 通常通过高速的基于网络的通信链路 (以太网或无限带宽技术)。MPI 被广泛使用和传授。这是一个集群受控环境中的很好的解决方案。

2) OpenMP (开放多线程) 是一种设计用于在一个节点内并行的系统级系统。它的工作方式完全不同, 程序员指定了各种各样的并行指令通过编译器编译指示。然后, 编译器会自动尝试根据可用的处理器核心数量将问题分解成 N 个部分。许多编译器都支持 OpenMP, 包括用于 CUDA 的 NVCC 编译器。OpenMP 的由于底层 CPU 体系结构, 容易造成缩放问题。因为 CPU 中的带宽不足以让所有的内核不断地将数据流传输。

3) Pthreads 是一个在 Linux 上用于多线程应用程序的库。和 OpenMP 一样, pthreads 使用线程而不是进程, 因为它被设计用于并行化单个节点。然而与 OpenMP 不同, 程序员负责线程管理和同步。这提供了更多的灵活性, 因此能以更好的性能运行精心编写的程序。

4) ZeroMQ (0MQ) 也值得一提。这是一个你可以链接到的简单库, 我们将在本书的后面使用它来开发一个多节点, 多 GPU 的例子。ZeroMQ 支持使用基于线程的单个交叉平台, 过程基于网络的通信模型 API。它也可在 Linux 和 Windows 平台上使用。

它是为分布式计算设计的，因此连接是动态的，会出现节点失败。

5) Hadoop 也是你可以考虑的。Hadoop 是 Google 的开源版本 MapReduce 框架。它主要针对 Linux 平台。这个框架的概念是把采集到的一个巨大的数据集，将其分成（或映射）成多个块。数据集使用并行划分成数百个或数千个节点文件系统，而不是单纯的发送数据到节点。取而代之的是将包含数据的程序发送到节点。输出将写入本地节点并保留在该节点。随后的 MapReduce 程序以前的输出，并以某种方式再次转换。由于数据实际上是多个节点的镜像，这允许高度容错以及高吞吐量系统。



## 致 谢

感谢导师吴茜媛老师在毕设工作中的督促、关心和多方面的建议和指导，很荣幸能与背光板项目组的各位老师、前辈一起工作，同时也感谢所有人在这次工作中提供的帮助和教诲。