

CPU-GPU异构平台上平面光源检测方法的 并行化设计与实现

张志源

（西安交通大学 电信学院 计算机33，陕西 西安）

2017年5月

目录

1	绪论	4
1.1	背景与意义	4
1.2	研究现状	4
1.3	论文主要内容	5
1.4	论文组织框架	5
2	项目需求和整体解决方案	6
2.1	项目概况	6
2.1.1	读取算法参数	7
2.1.2	图像采集和输入	7
2.1.3	图像检测	7
2.2	多线程加速设计方案	10
2.2.1	图像处理过程中的线程分配	10
2.2.2	OpenMP的应用	10
2.3	GPU通用计算的应用	10
2.4	CUDA简介	10
2.5	在OpenCV中调用GPU的方法	10
2.6	本章小结	11
3	GPU图像检测算法的实现	11
3.1	代码移植的程序框架设计	11
3.2	本章小结	12
4	并行化设计的整体实现与测试	12
4.1	OpenCV和CUDA环境搭建	12
4.1.1	软硬件环境	12
4.1.2	Cmake重新编译OpenCV	13
4.1.3	Visual Studio 2013下对项目的配置方法	13
4.2	多线程优化的性能测试	15

4.3	在GPU下的图像检测算法性能测试	15
4.3.1	OpenCV中的CUDA模块存在的问题	16
4.4	多线程条件下GPU函数的执行情况	16
4.5	本章小结	16
5	总结与展望	16
5.1	论文工作总结	16
5.2	下一步研究内容	16
5.3	本章小结	16

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^[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。

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^[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 Li-

brary，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^[4]。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，利用CUDA架构对原有的算法进行移植，利用GPU进行加速，并尝试与CPU多核多线程并行结合，探索其性能提高的方法。

1.3 论文主要内容

- 1.手机屏幕缺陷检测系统概况；
- 2.关于利用GPU计算加速算法的调研；
- 3.CPU多线程的优化方案和OpenMP的应用；
- 4.利用OpenCV的CUDA模块进行算法移植和测试；
- 5.CUDA函数在CPU多线程代码中的运行状态；

1.4 论文组织框架

第一章和第二章主要概述论文所研究的问题，包括其研究背景调研，所处的项目背景和实际意义。并介绍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。

第三章主要介绍如何借助OpenCV的CUDA模块来对算法进行移植，利用GPU设备的计算性能对算法进行优化。

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 项目需求和整体解决方案

2.1 项目概况

手机屏幕缺陷检测系统的抽象工作流程如图 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流水工作。

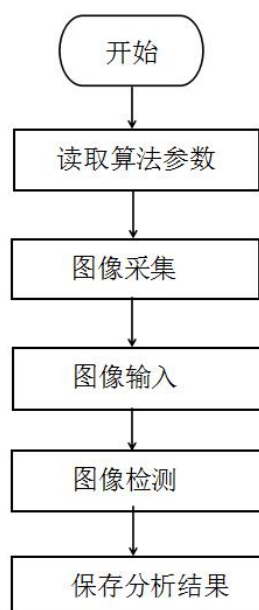


图 1: 手机屏幕缺陷检测整体流程

2.1.1 读取算法参数

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。

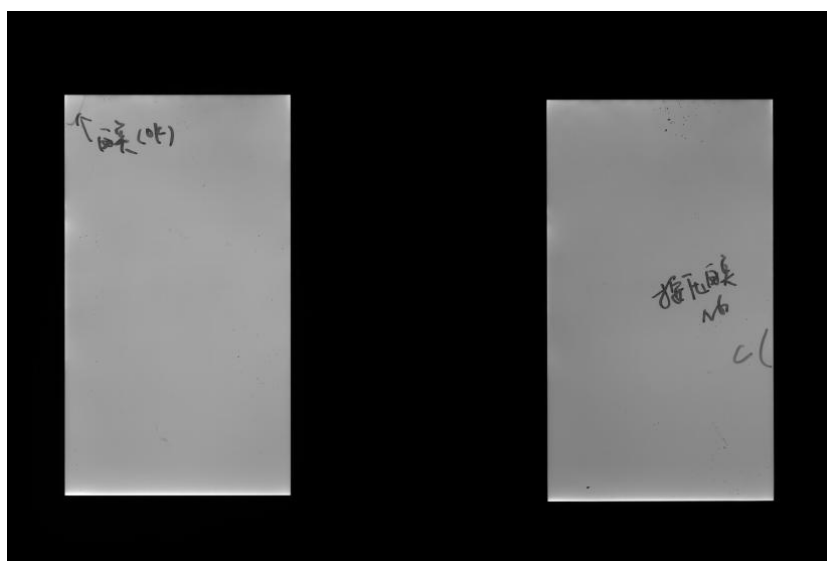


图 2: 自发光条件下拍摄的图片

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，左右两个样品分别进行一次检测。也就是说，现场检测一个工位的产品，要调用四次检测算法。为了提高处理速度，使用多线程来处理是必须的。

其中，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。

2.1.3 图像检测

检测对象主要位屏幕表面的污点和划痕，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）；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 6所示。因为所使用样本表

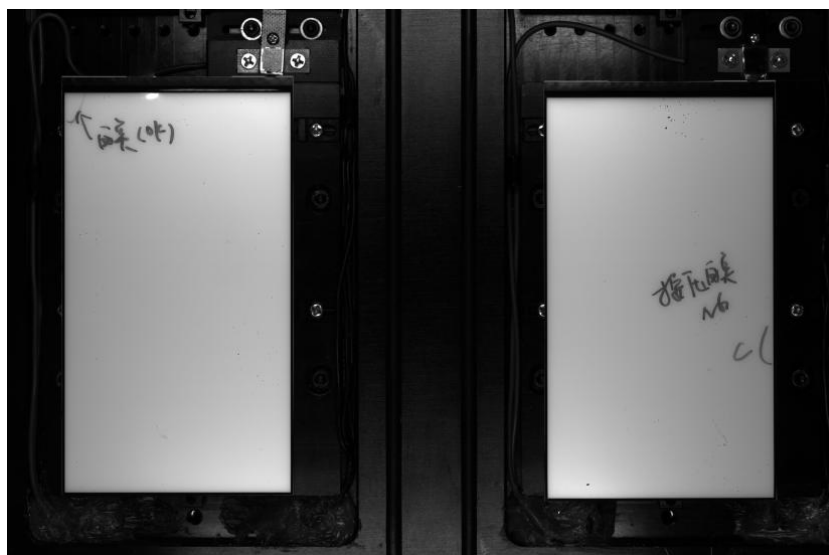


图 3: 外光源条件下拍摄的图片



图 4: 分离后的单张图像



图 5: 有效区域的掩模版

面较脏，所以检测出的缺陷数量比较多，也更醒目。

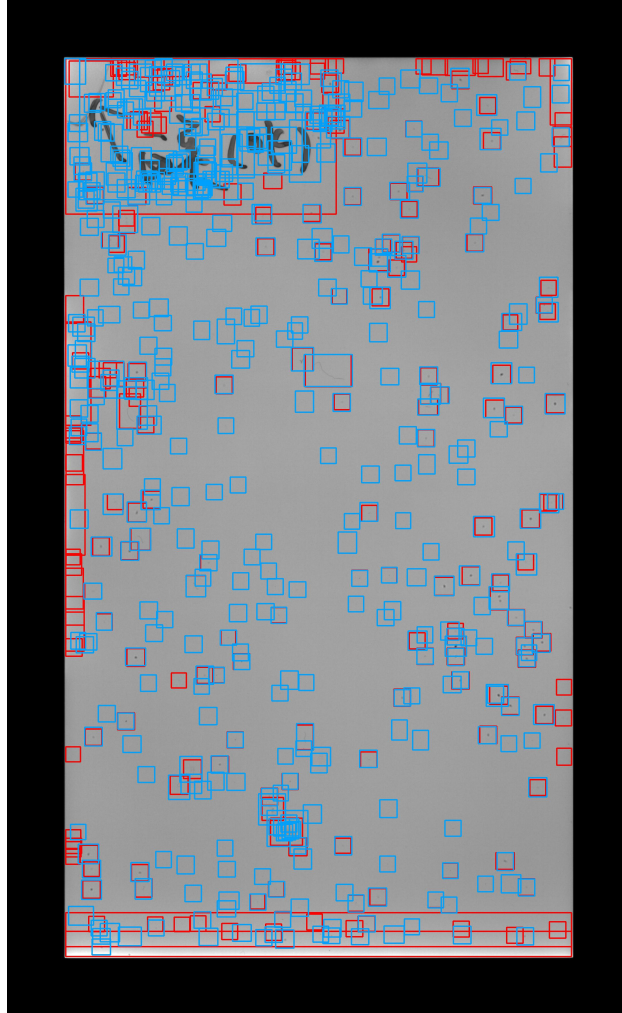


图 6: 图像检测结果

2.2 多线程加速设计方案

2.2.1 图像处理过程中的线程分配

输入的图片一共四张，分覆膜和不覆膜各一组，分别开启一个线程来处理。

同一组样片在两种不同光照条件下拍摄的两张图像输入后，先把大图分离成左右两个小图，产生四个小图；因为这里暂时不需要检测外光源情况下的图，所以开启两个线程针对自发光情况来调用检测算法，而不是四个线程。

2.2.2 OpenMP的应用

程序中大量使用for循环来对图像中的单个像素逐个处理。

note: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。

note: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。部分程序代码中使用size_t类型的变量来代替int类型，是出于跨平台的考虑

待完善

2.3 GPU通用计算的应用

2.4 CUDA简介

blank

2.5 在OpenCV中调用GPU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：

- 1.支持CUDA的设备初始化；
- 2.上传待处理数据到GPU（从Mat容器到GpuMat容器）；
- 3.调用OpenCV支持的GPU的处理函数；
- 4.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。

函数移植的方案是使用一个子类InspectorGPU来实现在GPU上运行的相关函数。子类继承原有的检测类BGBInspector，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。

2.6 本章小结

blank

3 GPU图像检测算法的实现

3.1 代码移植的程序框架设计

原代码包含了两个类：框架类（BGBFrame）和检测类（BGBInspector）。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（将左右两个样片分割成小图）。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，而其它没有变更的函数则沿用父类的方法。

在这之后，可以在框架类内部选择实例化检测类或者其子类来决定是否调用GPU计算，能比较直接的进行CPU和GPU的性能对比实验；同样的，在后续开发过程中，是否调用GPU的逻辑控制也变得简单了。

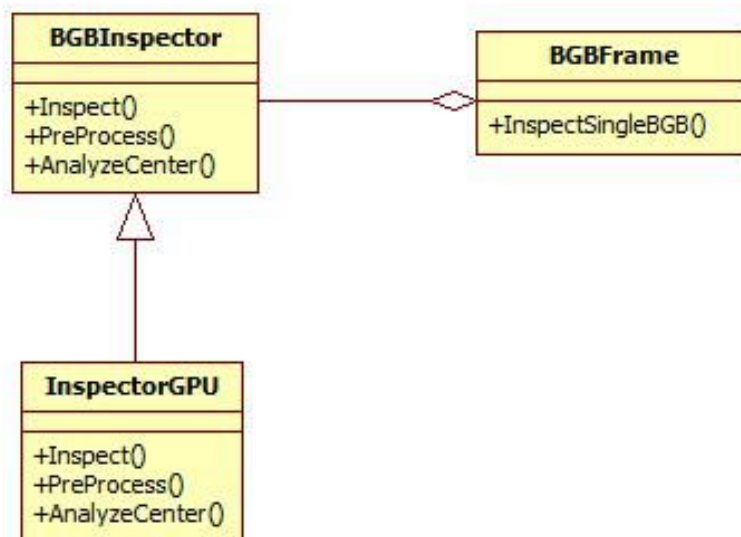


图 7: 框架类、检测类及其子类

3.2 本章小结

blank

4 并行化设计的整体实现与测试

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：

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

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。

4.2 多线程优化的性能测试

输入的图片分覆膜和不覆膜两组，每组输入自发光和外光源各一张图像，图片分离后左右两个样片各开一个线程进行检测；也就是说，同一时间的最大线程数为4。以下测试从单线程处理开始测试，每增加一个线程意味着增加一次图片检测过程。

表 1: 多线程下检测算法的性能测试			
图片大小	分辨率	线程数	时间开销
5	3	2	1
1	8	2	1

4.3 在GPU下的图像检测算法性能测试

插入代码片段

```
int main(int argc , char ** argv)
{
    printf("Hello , world!\n");
    return 0;
}
```

4.3.1 OpenCV中的CUDA模块存在的问题

note: CUDA模块的GPU函数在特定情况下存在性能低下的问题

4.4 多线程条件下GPU函数的执行情况

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，因此很有必要进行测试。

4.5 本章小结

5 总结与展望

5.1 论文工作总结

5.2 下一步研究内容

5.3 本章小结

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] OpenCV官网CUDA主页面[EB/OL].<http://opencv.org/platforms/cuda.html>.
- [6] 标准C++库参考文档[EB/OL].<http://www.cplusplus.com/reference>.
- [7] Shane Cook.CUDA Programming_A Developer's Guide to Parallel Computing with GPUs[EB/OL].<http://www.nvidia.com>
- [8] CUDA_C_Programming_Guide[EB/OL].<http://www.nvidia.com>