COPY FROM PAPER, PASTE TO DOCUMENT

Submitted By:

RAGHAV RAMESH

Department of Electrical & Computer Engineering

IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2015

B. Eng. Dissertation

COPY FROM PAPER, PASTE TO DOCUMENT

By

RAGHAV RAMESH

National University of Singapore

2014/15

Project ID	:	H0201100
Supervisor(s)	:	Dr. Ooi Wei Tsang
Deliverables	:	Report : 1 Volume

Abstract

Copy-Paste is a very common command used in almost all computing platforms. The aim of this project is to improve the speed of the existing AutoComPaste-Paper software by using parallel programming techniques. The initial prototype of AutoComPaste-Paper added an OCR functionality to the existing AutoComPaste software which focused on reducing the task of switching between the text editor and the web browser while performing research.

The focus will now be to reduce the overall time taken for the software to execute. This strives to reduce the limitations of the existing software and to bring it a step closer to production quality.

Acknowledgements

I am using this opportunity to express my gratitude to everyone who supported me throughout the course of this project. I am thankful for their aspiring guidance, invaluably constructive criticism and friendly advice during the project work. I am sincerely grateful to them for sharing their truthful and illuminating views on a number of issues related to the project.

I would like to express my heartfelt gratitude to my supervisor Dr. Ooi Wei-Tsang for his continued guidance and tireless support shown to me throughout the entire duration of this project. His experience and advice gave me the much-needed directions when I was at various standstills of this project.

Contents

List of Figures

Chapter 1

Introduction

There are several actions performed on a daily basis, and they are sometimes so subtle and yet useful that we become heavily reliant on them. Copy-Paste is one such action and is one of the most important fundamentals used for manipulating objects on a computer. It is widely used mainly to reduce the human effort of typing.

A related feature that aids in increasing efficiency while working with text documents is Autocomplete. It is a feature to save time in many computer programs that include some form of text input. It typically uses a predictive algorithm to guess what a user is typing, thereby allowing the user to interact more efficiently with a computer. Word processors, web browsers, source code editors, search engines and database query tools are examples of programs that commonly use some form of auto completion. Some of them often come with a predetermined set of words they will attempt to auto-complete.

While there are many online programs and source code editors that have implemented auto-completion, little progress has been made in extracting information from a real world hard copy source, such as a reference book. The current implementations either rely on scanning the document before running it through an Optical Character Recognition (OCR) engine, or taking a picture with a mobile phone and then running it through an OCR engine. An OCR engine converts text present in a digital image to editable text. It recognizes characters through optical mechanisms.

Many companies have a large collection of paper forms and documents that need to be scanned. Searching these documents by hand may take a long time, and it is only natural to seek to automate this process. One way would be to scan the documents and store them as images on a computer, and then perform optical character recognition on the scanned images to extract the textual information into separate text files.

AutoComPaste-Paper, an extension of the AutoComPaste application suite, is one such project that aims to exploit the lack of a feasible solution to such a problem. This solution is able to transfer large strings of text and precise lines of text, from a hard copy source directly into a text processor such as Microsoft Word, all the while keeping the intermediate processing invisible to the user. However, the prototype is still not production-ready as several aspects such as speed of processing the hard copy source, and accuracy of text recognition, needs to be improved.

The objective of this project is to improve the overall speed of processing the image captured and extracting the text using OCR in order to create a seamless environment for the user.

Chapter 2

Background

2.1 AutoComPaste

Copy-Paste across documents is a common task. Example scenarios include the writing of a project progress bulletin, filling out a grant report, literature review, trip planning, etc. In such scenarios, a common practice is to open relevant documents (e.g., previous reports, emails, finance spreadsheets, web pages, etc.) in the background and copy-paste relevant information from these source documents into a target document while editing (Zhao (2012)).

AutoComPaste (ACP) is an existing product with an objective of easing the process of copy-pasting from web browsers to text editors. The implementation also includes an extensive auto-completion mechanism that suggests text from the content in pages opened in a web browser. By minimizing the mundane task of switching between windows, AutoComPaste benefits users working on text processing work, like drafting research papers or project theses.

The first implementation of AutoComPaste consists of a web browser extension, a Microsoft Word Plugin and a background application with a database. The browser extension captures the text from opened browser windows and stores it in a database. The Microsoft Word Plugin then suggests auto-completion data while the user types on a document. The version of AutoComPaste boasts several features that include easy navigation using a keyboard or a mouse; smart context based suggestions that includes locations and URLs; an intelligent sorting mechanism that sorts based on recently viewed; frequency and customization for hotkeys to cater to different styles of user needs.

2.2 AutoComPaste-Paper

AutoComPaste has since been aiming to aid similar text processing for text from hard copies, which is a much harder problem. The latest extension of AutoComPaste suite, AutoComPaste-Paper allows for extracting text from a hard copy.

Copy-paste and auto-complete are not just useful while working with Internet research. Plenty of information is found in papers and books that have been documented and written years ago. Many of these papers and books cannot be found online and hence forces users and companies to type out information while researching or referring for projects and theses. Knowledge contained in paper-based documents is more valuable for todays digital world when it is available in the digital form.

In the above-mentioned cases, a product that captures books/ papers and processes the text in them could really be convenient for users across various industries including banking for check reading, post offices for mail sorting and most importantly researchers. The initial prototype of AutoComPaste-Paper makes use of the webcam of the users laptop to capture images of the paper from which text needs to be extracted. This captured image is then processed and sent to an OCR engine, which extracts text and stores in the existing database.

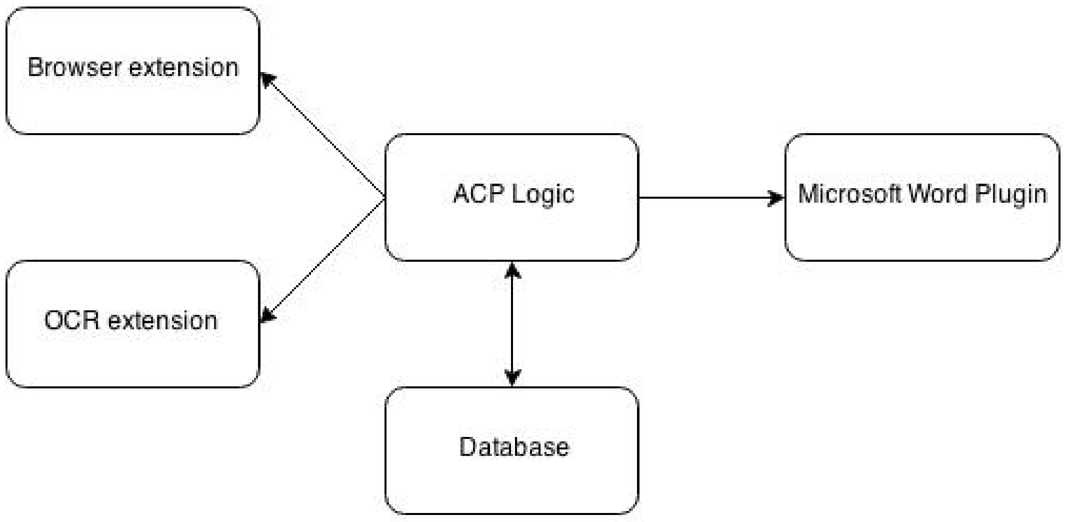


Figure 2.1: AutoComPaste Suite

2.3 Product Setup

The entire process was designed to be as simple and seamless to the user as possible. The procedure to operate the system is as follows:

Step 1: Place chosen Document under the External Camera.

Step 2: Type the beginning of the desired sentence and select the desired Auto-Complete choice.

To copy text from a different source document, replace the source under the camera and start typing on the text editor.

2.4 Working Procedure

Refer Figure 2.2 for the program sequence of ACP-Paper.

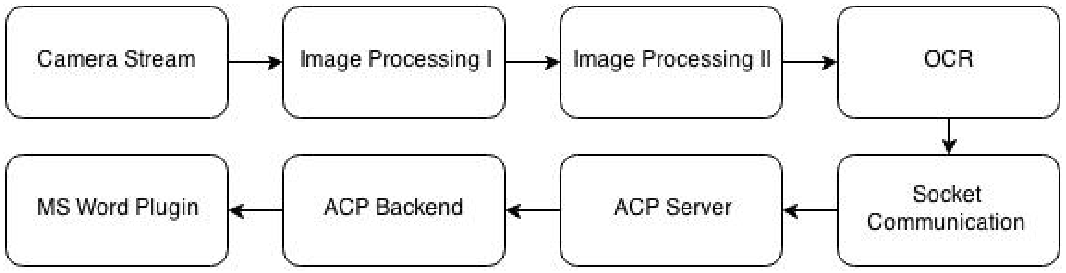


Figure 2.2: Working Procedure

2.5 Libraries Used

2.5.1 OpenCV

For performing Image Processing operations, AutoComPaste-Paper uses OpenCV (Open Source Computer Vision Library http://opencv.org), which is an open source BSD-licensed library (Cosenza). This library has been extensively used in this project for all image manipulations. This library was chosen, as it is open-source, has a well- documented API and is well supported by a dedicated community.

2.5.2 Tesseract

For performing OCR operations, AutoComPaste-Paper uses Tesseract-OCR. Tesseract is written in C++ and was chosen because it is open-source and is actively developed and maintained by Google, at http://code.google.com/p/tesseract-ocr/. Google also claims that it is the best OCR engine available (Google (2015)).

2.5.3 OpenCL

OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers and handheld/embedded devices (Group). OpenCL (Open Computing Language) greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and medical software.

The reason for choosing this library over the rest is because of its cross-platform compatibility.

Chapter 3

Literature Review

There has been a lot of research on techniques to improve OCR speed. Pre-processing of an image is an important step before sending to OCR because the bulk of the time taken is during the recognition of characters. Hence, reducing the noise in an image reduces the time taken for recognizing characters. Techniques that allow for parallel computing are also discussed.

3.1 Pre-processing

3.1.1 Binarisation

Modern computers can recognize up to 4 billion colors. This means that every pixel of an image requires 32 bits to be represented, which implies 4 bytes of memory. OCR, however, is independent of colors, i.e., a blue letter is the same as a black letter. Binarisation is a method to reduce color images into black and white. Each pixel now requires only a single bit to store the color information. One algorithm to perform binarisation is the threshold algorithm, which calculates an arbitrary value T as the threshold. Pixels with color greater than the threshold are converted to white pixels and those lesser, black. This method is very fast and simple (Palkovic (2008)).

However, there is a key flaw in this method, which is the reliance on the threshold value. The threshold value is usually calculated by averaging out the darkest and the lightest pixel. Experimental results have shown that low values of the threshold produce letters with holes, because pixels that should have been black were chosen to be white. On the other hand, higher values for the threshold produced blurry characters.

Local binarisation is a technique that can be used to avoid this problem. This technique estimates the threshold based on analyzing pixels around the area (Palkovic (2008)).

3.1.2 Thinning

Thinning is an algorithm that removes unwanted information from the characters in order to reduce the complexity of the image. This algorithm converts a thick line into one with thickness of 1 pixel. This is a simple and fast algorithm with no flaws (Palkovic (2008)).

3.1.3 Patch based techniques

Another method to identify parts of the images that can be processed / recognized in parallel is by computing patches of the image. Researchers have developed several sophisticated methods to analyze and edit digital images. However, most of the powerful methods are computationally intensive. Images are divided into patches and these patches are analyzed. Patches contain contextual information and have advantages in terms of computation and generalization (Liviu Petrisor Dinu and Popescu).

3.1.4 Despeckle Images

Three methods can be applied to despeckle (smooth edges to preserve features) images such as Gaussian blur, bilateral filter and median blur. While Gaussian Blur has already been implemented in the existing version of AutoComPaste-Paper, the other techniques have not been implemented.

Bilateral filter is a non-linear, edge preserving and noise reducing smoothing filter for images. The intensity of each pixel is replaced by the weighted average of the intensity values of pixels nearby. This preserves sharp edges by systematically looping through each pixel and adjusting the weights of the adjacent pixels accordingly (OpenCV).

Another method that preserves edges while removing noise is Median Blur. A median blur runs through every element of the signal (image) and replaces them with the median of its neighboring pixels (located in a square neighborhood around the evaluated pixel) (OpenCV (2014)).

3.1.5 Linguistic Pre-processing

Using the linguistic and statistical knowledge in pre-processing, higher speed of processing can be achieved. This method replaces the rough classification, which occupies a large part of the recognition time, with character prediction. In Japanese, this method was found to be 20% faster than traditional Optical Character Recognition (Kigo (1993)).

3.2 Extracting paragraph breaks

The first step before running OCR using multiple cores in CPU or GPU is to identify the areas that can be run in parallel. Unfortunately, we cannot divide the page into equal parts because this may lead to some of the characters being damaged if they are stuck in the page division. Instead, if it is possible to isolate paragraphs of text in an image, these paragraphs can then be sent to OCR simultaneously.

An image can be broken up into multiple paragraphs by looking at the entropy of each 5-10 pixel horizontal slice/strip. If a strip is not busy, we can assume that there is no text here. We can now divide the page in this region (ODell (2011)).

3.3 Stroke Width Transform

While researching for an appropriate algorithm for recognizing text, Stroke Width Transform (SWT) was considered. The algorithm transforms the image data color values per pixel to the most likely stroke width. The resulting system is therefore, able to detect text from images irrespective of its scale, direction, font or language (Boris Epshtein (2008)).

When the traditional OCR engines are applied to images containing natural scenes, the accuracy rate drops drastically. The reason why this happens is because the engines are designed for scanned text and so, they heavily rely on segmentation that separates the background from the actual text that needs to be extracted. This is easy for images that contain scanned text such as papers or books, however, is harder for text present in natural scenes (Boris Epshtein (2008)). Other factors such as blur and noise also come into play which are much higher in natural scenes than scanned papers or screenshots.

Further research on SWT was not done because majority of the use cases of this project were research papers and books and do not involve natural scenes.

However, this algorithm can be used when text needs to be recognized from the images in a page, for instance, flow charts.

3.4 OCR-based Projects

3.4.1 OCRAD

GNU Ocrad is an OCR (Optical Character Recognition) program and library based on a feature extraction method. It reads images in pbm (bitmap), pgm (greyscale) or ppm (color) formats and produces text in byte (8-bit) or UTF-8 formats. The pbm, pgm and ppm formats are collectively known as pnm (wp- (2014)). A layout analyser is also used which is capable of separating the columns or blocks of text normally found on printed pages (Foundation (2014)).

It can be used as a stand-alone command-line program or as a backend to other programs. A secondary experiment of comparison between Tesseract and Ocrad was referred to in order to make a decision about whether Ocrad could be chosen as the backend OCR engine for this project.

The aim of this test was to find out which OCR engine fares the best when fed with images of different dpis (Kay (2007)).

Test Conditions	ocrad	Tesseract
200dpi, very clean, includes italics/bold	95%	100%
72dpi, black and white, clean	0%	97%
72dpi with minor linear distortion	0%	97%
72dpi, minor linear distortion, and skewed 2 degrees	0%	96%

Table 3.1: Tesseract vs ocrad Results

Its usage of a backend was to be explored further, however there were certain drawbacks that over weighed these advantages. The engine can only recognise bitmap, greyscale and ppm formats. This engine does not have a strong community support. There is no structured method for reporting issues, such as those available in Google Code, GitHub or BitBucket. All queries needs to be sent to an email bug-ocrad@gnu.org. This method is outdated and does not help other users who have the same or similar problem. It is not possible to track if a particular issue has been resolved or not.

3.4.2 Project Naptha

Project Naptha applies computer vision algorithms on all images in the viewport of a user's browser. Its available as a browser extension. The result is a streamline and intuitive experience, where you can highlight, copy and paste and even edit and translate the text that was previously contained within an image (Kwok (2015)).

The primary focus of Project Naptha is actually detecting the text, rather than accurately recognising the text that was detected. It runs the Stroke Width Transform algorithm, invented by Microsoft Research in 2008, which is capable of identifying regions of text (especially in images with noise) in a language-agnostic manner (Kwok (2015)).

Project Naptha is actually continually watching cursor movements and extrapolating half a second into the future so that the processing of the image can be started earlier in order to make the user experience seem smoother and real-time (Kwok (2015)).

3.4.3 DetectText

DetectText is an application that takes an image input and recognises the text contained using the Stroke Width Transform technique. In addition to optical character recognition, the application also uses OpenCV and Boost for its image processing. The output is the text detected from the input image (Perrault).

3.4.4 Capture2Text

Allows the user to quickly snapshot a small portion of the screen, OCR it and (by default) save the result to the clipboard. This application makes use of Tesseract for its backend to recognize the text captured in the screenshot (Capture2Text (2015)).

3.4.5 OpenOCR

OpenOCR provides Open Source OCR with the help of Tesseract and Docker. It makes it easy to host a developer's OCR REST API. Some of its features include a scalable message passing architecture via RabbitMQ. They achieve platform independence via Docker containers. They provide a layer above the Tessaract OCR engine with capabilities such as an image pre-processing chain such as Stroke Width Transform. You can also pass arguments to Tesseract such as white-listing the set of characters required to recognized by Tesseract (Leyden (2015)).

Chapter 4

Re-engineering AutoComPaste-Paper

4.1 Object Oriented Paradigm

The first task of this project was to convert the existing version of AutoComPaste-Paper from a procedural style to an Object-Oriented style. This was decided because Object Oriented design has great benefits over procedural style, like encapsulation and code reuse. This also allowed for decoupling the classes and also removed the need for passing around instances of classes.

Object oriented paradigm (OOP) meant better software productivity. The previous check-in of AutoComPaste-Paper contained classes for each major concept that the project was addressing - Image Processing, OCR, Socket Connection to the ACP server, etc. This gives a false impression of separation of concerns. Several instances of this class were passed around and often a new developer can get lost in tracking the values of these object instances. In order to solve this problem, a clear object oriented architecture was required that allows better separation of concerns.

OOP is also extensible, as objects can be extended to include new attributes and behaviors. Objects can be reused within and across applications. Due to these three factors, modularity, extensibility, and reusability, object-oriented programming provides improved software development productivity over traditional procedure-based programming techniques (Foundation (2015)).

4.2 Architecture Design

Speaking of good software engineering principles, the logical next step after deciding to implement an Object-oriented paradigm, was to design the architecture of the project. A class diagram was drawn to do the same. The purpose of a class diagram is to model the static view of the application. They are the closest representation that can be directly mapped to object oriented language and hence was chosen at the time of architecture design (Tutorials Point). The class diagram in Figure 4.1 was drawn to describe the responsibilities of the system.

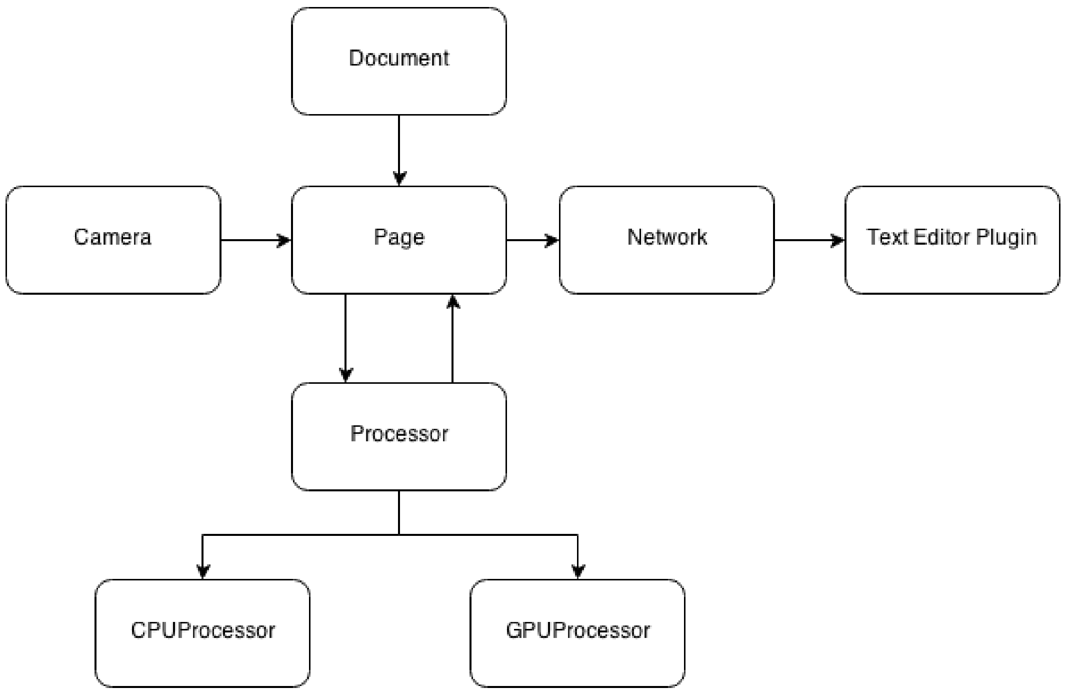


Figure 4.1: AutoComPaste-Paper Architecture

4.3 Documentation

The first step before venturing into implementation of the above architecture was to document the code. This was important as this documentation would lead the programmer into foreseeing the structure of the classes, and also acts as a reference during, and after the implementation.

The documentation for the classes in the above architecture was created using Doxygen, the de-facto standard tool for generating documentation for annotated C++ sources. Methods and attributes were defined and documented in a black-box fashion. The documentation can be found in the GitHub repository of ACP-Paper. Another aspect of documentation was maintaining existing code. The first code check-in of the existing product was stored in a repository in GitHub and further changes during the implementation of this project were committed to this repository. Different releases were maintained meticulously to allow for easy conversion to open source should it be decided to later on.

4.4 Achieving Platform Independence

Following the completion of the documentation, the next task was to implement the product. A decision was made to implement the product in OS X, while another fellow FYP student decided to implement in Windows. This was an attempt to make the product cross platform. The first version of AutoComPaste-Paper was written in a Windows platform with several Windows specific libraries like, Winsock and some Windows Desktop technologies like synchronization methods for threads.

Achieving cross-platform capability for the product implied that Windows specific libraries could no longer be used. POSIX Threads were instead chosen to replace the Windows synchronization methods.

The next step was to implement the pre-documented classes.

4.5 Camera

To obtain the video stream, the computer interfaces with an external hardware camera.

4.5.1 Capturing stream

The OpenCV functions cvCaptureFromCAM/VideoCapture are used to obtain the camera stream. Each time the function is called a single static image is returned.

When called at regular time intervals a video stream is formed. All subsequent manipulations are done on the individual images. Frames are stored on a stack, to allow image processing modules to track the changes to the frames.

4.5.2 Capture Resolution

The input stream from the camera can be changed for different standard resolutions. This will accommodate the different hardware settings that are found on different PC setups. A key optimization feature used in the page detection module is the ability to change the capture resolution of the input stream on the fly, as higher resolution streams require more computational time, while lower resolution can be processed faster.

This is due to the hardware limitations on most high-end webcams, such as the Microsoft LifeCam Studio

4.6 Document

The Document class is used to store a collection of 'Page's. This class can be used to check the validity of the page under the camera. All image processing operations, such as, de-skewing and flipping of pages can be decided after checking if the page of the document has been flipped or if the current page of the document is actually the same page de-skewed at a particular angle.

If the page has been flipped, the camera captures another set of images and the process is repeated. Whereas if the there is a skew in the position of the page due to some environment changes such as an accidental re-positioning or due to the presence of an object, the image is accordingly de-skewed.

In addition to Image Processing checks, the Document class contains specific attributes of the book / paper under research. These attributes can be used for obtaining an online version of the same book / paper.

4.7 Page

The Page class consists of all necessary attributes of the page of the document at different points of execution of the program. Each page has an image that will be processed or sent to OCR. This image (a cv::Mat object) will be passed to the Document and Processor classes to perform operations.

Some typical methods and attributes include the dimensions and the set of the frames of the specific page, to be used for averaging.

4.8 Processor

The Processor class contains the main models of the program. This class is used as a layer that connects the Controller (main) to the external libraries, namely OpenCV and Tesseract, used in the project.

A Processor object takes the image attribute of the Page and performs a variety of functions. The Processor Models API includes Image Processing methods, OCR methods and the logic for identifying words and white regions in the image.

Dilate, erode, rotate and resize image are some of the image processing methods that are available in the class. These methods are performed on the image for removing blemishes in the image before sending to the OCR engine. The OCR methods include extracting text given an image and white listing the characters to be recognised by the OCR engine.

In addition to the Image Processing and OCR functions, the class also contains the logic for intelligently identifying white regions of the image, methods to detect letters, contours and cutting the image into an array of sub images at these identified white regions.

Another class, AdvancedProcessor is also implemented. This class enables the usage of OpenCL, if available on the user's machine. The kernel source takes multiple images as an input and returns the text contained in the images as the output. The processAcrossCores() method implemented in the AdvancedProcessor is an alternative to the method extractTextFromImage() in the Processor class. As the name suggests, the image of the page is split into an optimum number of sub images intelligently by the Processor class, and sent to the AdvancedProcessor to be computed across the cores available in the user's machine.

4.9 Network

The Network class acts as the layer that connects the Controller with the database. Currently implemented as an interface, Network Contains abstract socket functions to connect the AutoComPaste server.

The interface exposes a CRUD API layer. Consider the stage of the program where the text of the image has been extracted by the OCR engine, this text can be sent to the database using an implementation of the Network interface. This text that is stored in the database will then be used to suggest in the text processor that the end-user is using.

Chapter 5

Understanding Libraries

One significant advantage of using Open-Source libraries is the fact that there is access to the source code. Therefore, it is imperative to not treat the external libraries as black boxes. This is to prevent redundancy in code and ensuring that the resources that are requested by the developer are not already occupied by the code present in the external library.

5.1 Tesseract

5.1.1 Pre-processing methods

Tesseract does various image processing operations internally (using the Leptonica library) before doing the actual OCR. It generally does a very good job of this, but there will inevitably be cases where it is not good enough, which can result in a significant reduction in accuracy (Tesseract (2015)). Its important to identify when to implement pre-processing methods and when not to, before sending the input image to the OCR engine.

Otsu's Threshold

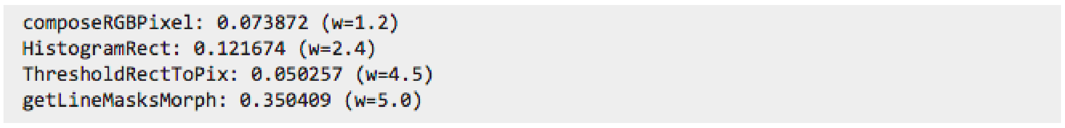
This is a simple thresholding method applied to binarise the input image. This thresholding method is clustering based. The algorithm assumes that the image contains two classes of pixels following bi-modal histogram (foreground pixels and background pixels), it then calculates the optimum threshold separating the two classes so that their combined spread (intra-class variance) is minimal (Wikipedia (2015a)).

Since Tesseract binarises the image, the developer should not threshold the image before sending to the OCR engine. However, in this project, there is a need for identifying the white regions in an image because the image needs to be divided in non-important regions only. This means that the thresholding method applied by Tesseract needs to be by-passed. The image need not be binarised if the image extraction is not distributed across cores. It is also possible to obtain the binarised image after Tesseract applies the Otsu's threshold in order to compare the binarising methods applied by the developer and Tesseract.

The documentation of Tesseract advises the developer to apply an adaptive threshold for images that are noisy or do not have a good contrast between the color of the text and that of the background.

5.1.2 OpenCL API

Tesseract makes use of OpenCL to offload some of its compute-intensive portions to available CPU cores and GPUs. Tesseract detects patterns/letters in an image. This is roughly speaking image processing, where each pixel can be computed in parallel, therefore a very good candidate for GPU (OpenCL/CUDA/etc..) (Zeros (2015)).



```
composeRGBPixel: 0.073872 (w=1.2)
HistogramRect: 0.121674 (w=2.4)
ThresholdRectToPix: 0.050257 (w=4.5)
getLineMasksMorph: 0.350409 (w=5.0)
```

Figure 5.1: Pre-processing done using OpenCL

From the profiling code it can be understood that Tesseract is doing some pixel conversion in GPU (very efficient), then a histogram and finally thresholding the image (Zeros (2015)).

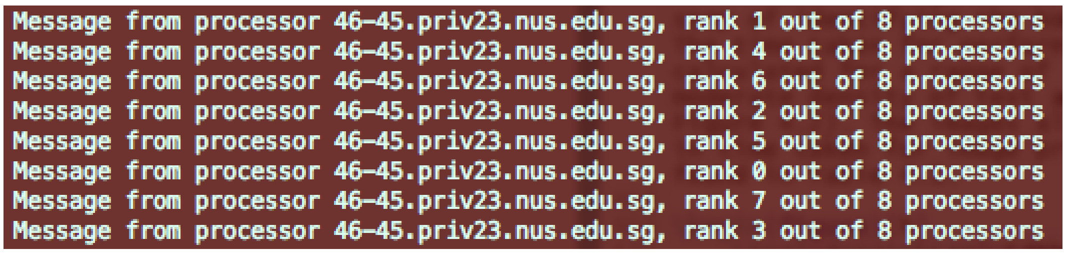
The profiling code is just measuring the time taken by each OpenCL device, and selecting the best one. After it selects the best device, it is the task of the device to do the compute load. In the case of GPU it will use all the shading cores (compute cores). In case of CPU it will simply run a thread pool. But OpenCL does not perform any GPU-CPU work balance (Zeros (2015)).

5.2 MPI

In a typical parallel program, in order to complete a task, various messages are passed among processes - the message passing model. For instance, the master process, sends data to the slave processes and waits for the message from the slave processes after their completion. The message passing model can describe almost any parallel program (Kendall (2015)).

Due to the common prevalence of this message passing model, a standard was created in 1992, called the Message Passing Interface (MPI). Similar to other standards, this standard would enable programmers to write portable message passing parallel programs (Kendall (2015)). In the task at hand, the main method is the master and the slave processes are the threads that extract text from an image using the OCR engine. Using this analogy, a decision was made to choose to use MPI as an attempt to complete the task of speeding up

AutoComPaste-Paper.



```
Message from processor 46-45.priv23.nus.edu.sg, rank 1 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 4 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 6 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 2 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 5 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 0 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 7 out of 8 processors
Message from processor 46-45.priv23.nus.edu.sg, rank 3 out of 8 processors
```

Figure 5.2: MPI workers in action

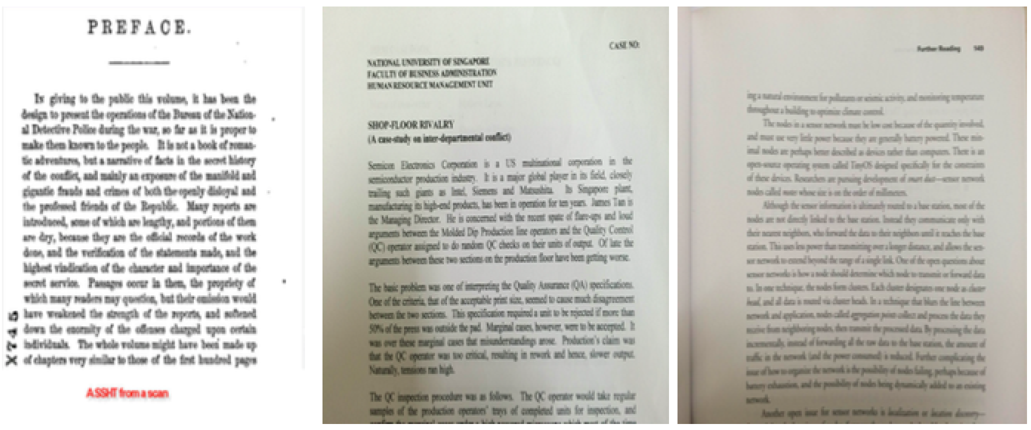
A couple of implementations mpich2 and open-mpi were tried for the said purpose. However, MPI had the ability to pass messages within cores of the CPU, it cannot be used to allocate memory objects such as images to the cores for input and output. Hence, this approach was rejected and OpenCL was resumed for the task of speeding up ACP-Paper.

Chapter 6

Implementation

6.1 Identifying the Bottleneck

Prior to venturing into optimizing the speed of the product, an essential step of identifying the bottleneck in the current version was performed. The time taken for performing image processing and OCR methods were calculated for a range of images, i.e., an image with scarce text, an image with moderate density of text and an image with high density of text (refer Figure 4.2). The images also varied in dimensions to observe the variations in time taken for image processing. The results of the test conducted are found in Table 6.1.



PREFACE.

In giving to the public this volume, it has been the design to present the operations of the Bureau of the National Detective Police during the war, so far as it is proper to make them known to the people. It is not a book of romantic adventures, but a narrative of facts in the secret history of the conflict, and mainly an exposure of the manifold and gigantic frauds and crimes of both the openly disloyal and the professed friends of the Republic. Many reports are introduced, some of which are lengthy, and portions of them are dry, because they are the official records of the work done, and the verification of the statements made, and the highest vindication of the character and importance of the secret service. Passages occur in them, the propriety of which many readers may question, but their omission would have weakened the strength of the reports, and softened down the enormity of the offenses charged upon certain individuals. The whole volume might have been made up of chapters very similar to those of the first hundred pages

X745

ASSHT from a scan

CASE NO:

NATIONAL UNIVERSITY OF SINGAPORE
FACULTY OF BUSINESS ADMINISTRATION
HUMAN RESOURCE MANAGEMENT UNIT

SHOP-FLOOR RIVALRY
(A case-study on inter-departmental conflict)

Semicon Electronics Corporation is a US multinational corporation in the semiconductor production industry. It is a major global player in its field, closely trailing such giants as Intel, Siemens and Matsushita. Its Singapore plant, manufacturing its high-end products, has been in operation for ten years. James Tan is the Managing Director. He is concerned with the recent spate of flare-ups and loud arguments between the Molded Dip Production line operators and the Quality Control (QC) operator assigned to do random QC checks on their units of output. Of late the arguments between these two sections on the production floor have been getting worse.

The basic problem was one of interpreting the Quality Assurance (QA) specifications. One of the criteria, that of the acceptable print size, seemed to cause much disagreement between the two sections. This specification required a unit to be rejected if more than 50% of the press was outside the pad. Marginal cases, however, were to be accepted. It was over these marginal cases that misunderstandings arose. Production's claim was that the QC operator was too critical, resulting in rework and hence, slower output. Naturally, tensions ran high.

The QC inspection procedure was as follows. The QC operator would take regular samples of the production operators' trays of completed units for inspection, and

Further Reading 149

ing a natural environment for pollutants or seismic activity, and monitoring temperature throughout a building to optimize climate control.

The nodes in a sensor network must be low cost because of the quantity involved, and must use very little power because they are generally battery powered. These minimal nodes are perhaps better described as devices rather than computers. There is an open-source operating system called TinyOS designed specifically for the constraints of these devices. Researchers are pursuing development of smart dust—sensor network nodes called motes whose size is on the order of millimeters.

Although the sensor information is ultimately routed to a base station, most of the nodes are not directly linked to the base station. Instead they communicate only with their nearest neighbors, who forward the data to their neighbors until it reaches the base station. This uses less power than transmitting over a longer distance, and allows the sensor network to extend beyond the range of a single link. One of the open questions about sensor networks is how a node should determine which node to transmit or forward data to. In one technique, the nodes form clusters. Each cluster designates one node as cluster head, and all data is routed via cluster heads. In a technique that blurs the line between network and application, nodes called aggregation points collect and process the data they receive from neighboring nodes, then transmit the processed data. By processing the data incrementally, instead of forwarding all the raw data to the base station, the amount of traffic in the network (and the power consumed) is reduced. Further complicating the issue of how to organize the network is the possibility of nodes failing, perhaps because of battery exhaustion, and the possibility of nodes being dynamically added to an existing network.

Another open issue for sensor networks is localization or location discovery—

Figure 6.1: Images chosen for time experiment

Image	Dimensions	Image Processing time (in ms)	OCR time (in ms)
1	690 x 667	12	22412
2	3264 x 2448	245	29028
3	2048 x 1536	88	49498

Table 6.1: Time Taken for Image Processing And OCR

Clearly, the bottleneck in the project is text recognition. After this was found, the next was to research ways to improve the speed. Since a single parse of the image by the OCR engine takes about 10 - 40 seconds depending on the dimensions and density of the image, the logical approach would be to parse multiple sub images simultaneously.

OpenCL can be used to execute the method that extracts the text from the image in parallel with the help of the available cores of the CPU or GPU in the user's machine. This approach can be broadly divided into two major sub parts - splitting the parent image into sub-images in such a way so that the text of image is not being cut out or damaged; and building the kernel source code to process these images in parallel.

6.2 Evidence of Speedup using Tesseract with OpenCL

Before venturing into writing the CL kernel source, it was important to research if there was actual speed by using OpenCL in conjunction with Tesseract.

The contributions from AMD engineering team saw ample opportunities for offloading the compute intensive portions while maintaining the accuracy of the program. They do so by offloading modules on to the GPUs and execute using OpenCL which is OS and platform independent. With the help of the time information given by Tesseract, it is possible to derive certain conclusions regarding the speedup gained by using Tesseract along with OpenCL (Slovak (2014)).

Experiments were conducted on a Linux and a Windows machine to test if using OpenCL extracted text from an image was faster than without using it. On a linux machine it was observed that an image took 12.18 seconds while using OpenCL and 21.85 seconds without it. Hence, there is a 44% speedup. This is the case even when there was no GPU involved (Slovak (2014)).

However, the same image when processed on a Windows machine, there was a slight performance degradation using OpenCL than without. However, this was due the difference in hardware used in both the systems and it is not fair to compare the two systems due to the difference in compilers as well (Slovak (2014)).

6.3 Approaches to identify white regions

After obtaining necessary evidence that OpenCL could indeed be used to improve the speed of text recognition, the next approach was to tackle the first of the two sub parts i.e., identifying the white regions in an image, in order to split the image at those regions.

6.3.1 Blobs of text

The first approach to split the page in regions where the text is not present was to identify the paragraph beginning and endings. OpenCV could be used to identify these blocks of text and these blocks can then be sent to the OCR engine individually for text extraction.

The following were the steps followed to achieve the desired result:

1. Load the image containing the text.

2. Extract pixels representing individual characters of text.

   (a) This could be done using two approaches, using Stroke Width Transform. For SWT, the algorithm needs to know if the text is light-on-dark background or dark-on-light background. If this is not known, then two passes of this algorithm needs to be made.

   (b) The next step was to flood fill the image repetitively until all pixels were covered. For every seed pixel, the pixel that got included in the flood-fill is marked and flood-fill is not run for these pixels again.

3. The components are grouped to make text words, then to text lines and finally into text blocks.

4. This text block is then passed to Tesseract and the same steps are repeated for the remaining text blocks (Wilke (2014)).

Limitations

Since the blocks of text were flood-filled to become a blob, it was necessary to undo this change before sending to Tesseract so that the OCR engine will be able to recognize the characters in the image. This proved to be a tedious task and hence this method was overlooked.

6.3.2 Contours

A different approach was to identify the regions of text instead of the white regions. Drawing bounding contours at the paragraph level or the line level could do this. By doing so, the page can be split in areas that are not bounded by the text regions. An OpenCV method findContours() of the Structural

Analysis and Shape Descriptors API was used to achieve the desired result. The method takes in the source and destination variables of the image and mode of contour. However, a side effect of adopting this method led to the background of the image to turn black. Hence, this method was overlooked.

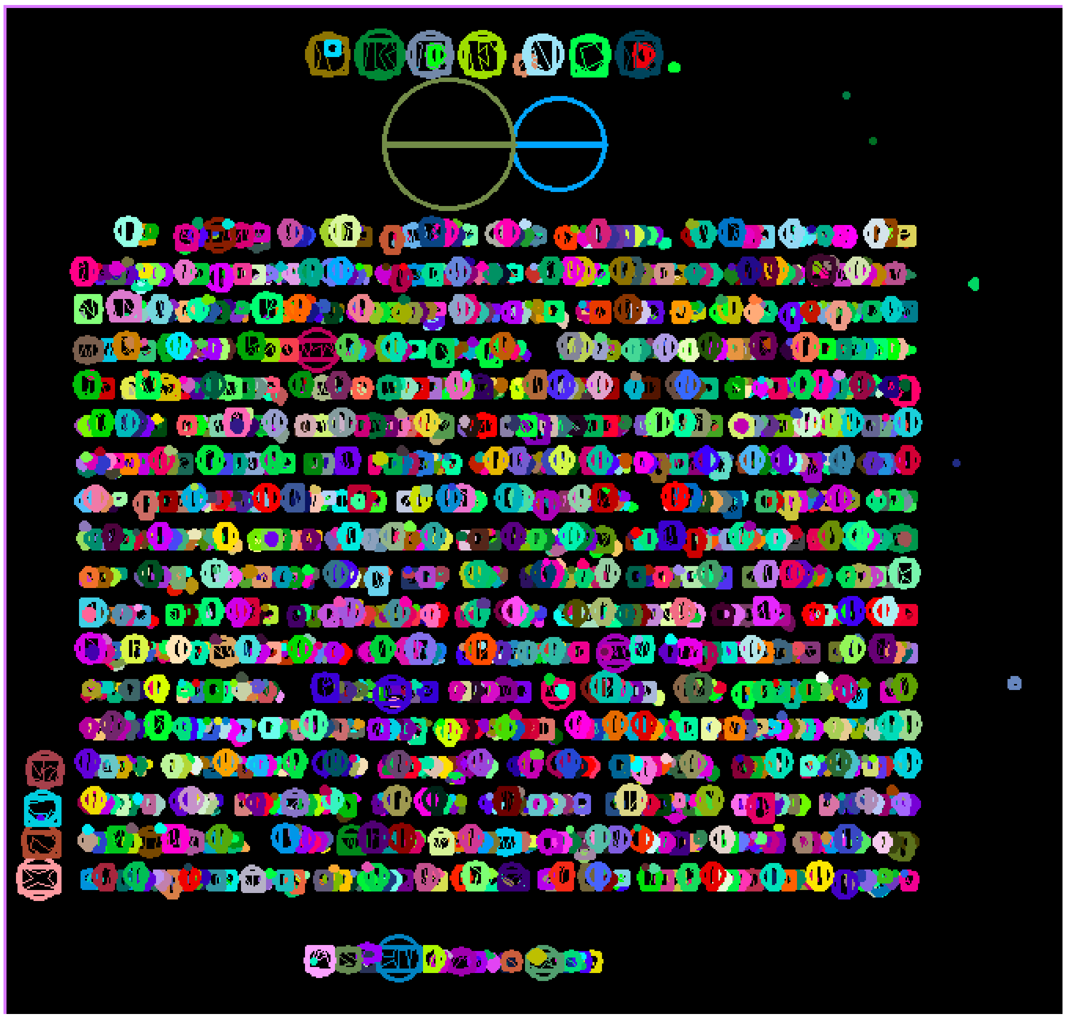


Figure 6.2: Contours drawn around letters

6.3.3 Erosion of paragraphs

This method was similar to the previous approach of flood-filling the blobs of text. The difference between the previous approach and this one is that, instead of flood-filling the paragraphs, they were eroded to become a big black blob. Then, bounding boxes or contours of these black blobs were found.

This approach eased the process of identifying white regions in the image because all that areas covered by text are now replaced with black blobs and hence it was easy to find the paragraph breaks.

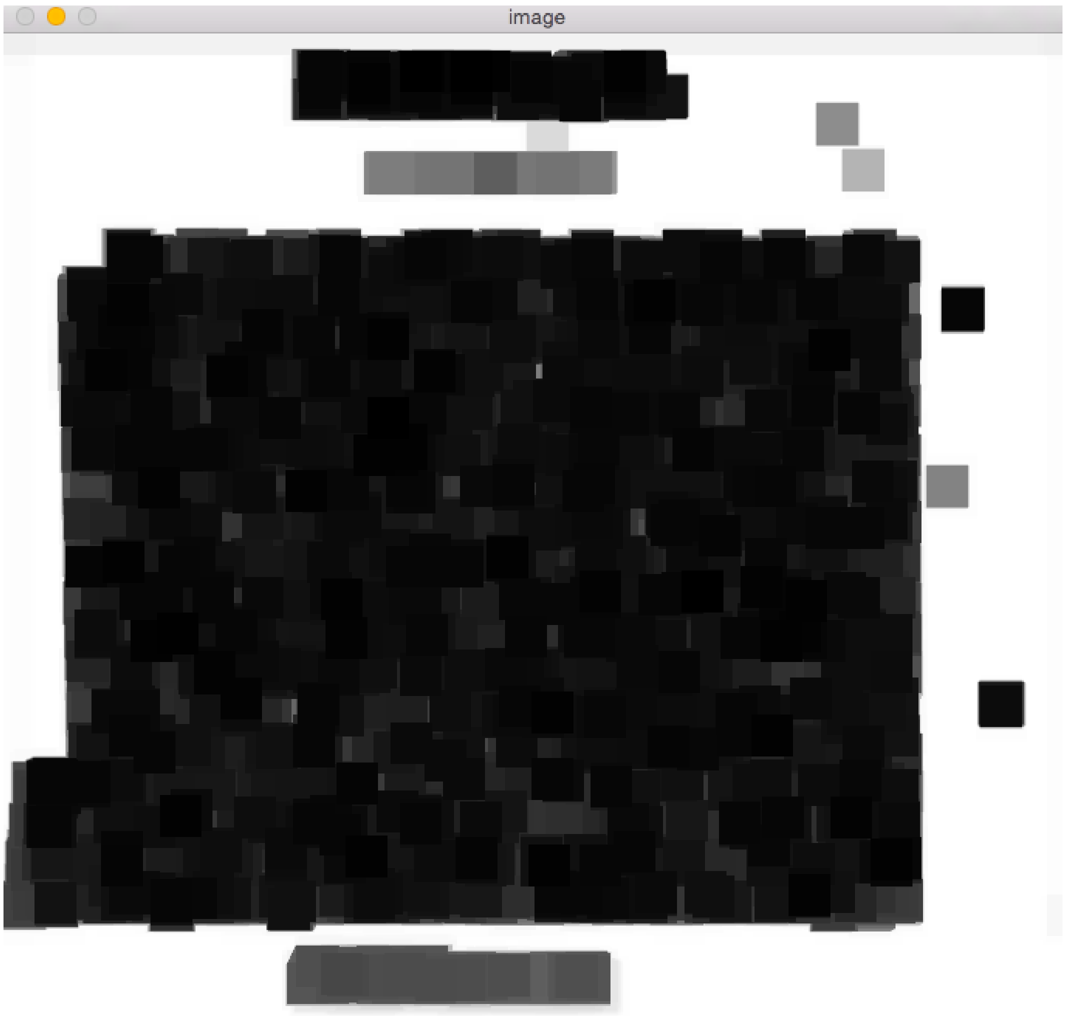


Figure 6.3: Erode Image

However, the step that involved undoing the process of converting text intro blob still remained the same from the previous method and hence this warranted for a change in the way splitting of image had to be approached.

6.3.4 Detect Letters

This method was a similar approach to finding contours of the text regions. The difference was that this method only identified the regions at a line level. The method takes in the source of the image on which the letters need to be detected and draws an outline around the lines. However, the problem with this method was the accuracy which was very low. There were several regions such as the middle of a paragraph that were detected as a white region because the method failed to bound the letters in those lines of the paragraph. This method was also discarded.

6.3.5 Identifying white lines

Another way of looking at splitting an image can be at the line level instead of a paragraph level. If it was possible to identify the white regions between lines, that would enable the parent image to be split into further number of images. This approach is advantageous when the parent image does not have a lot of paragraph breaks.

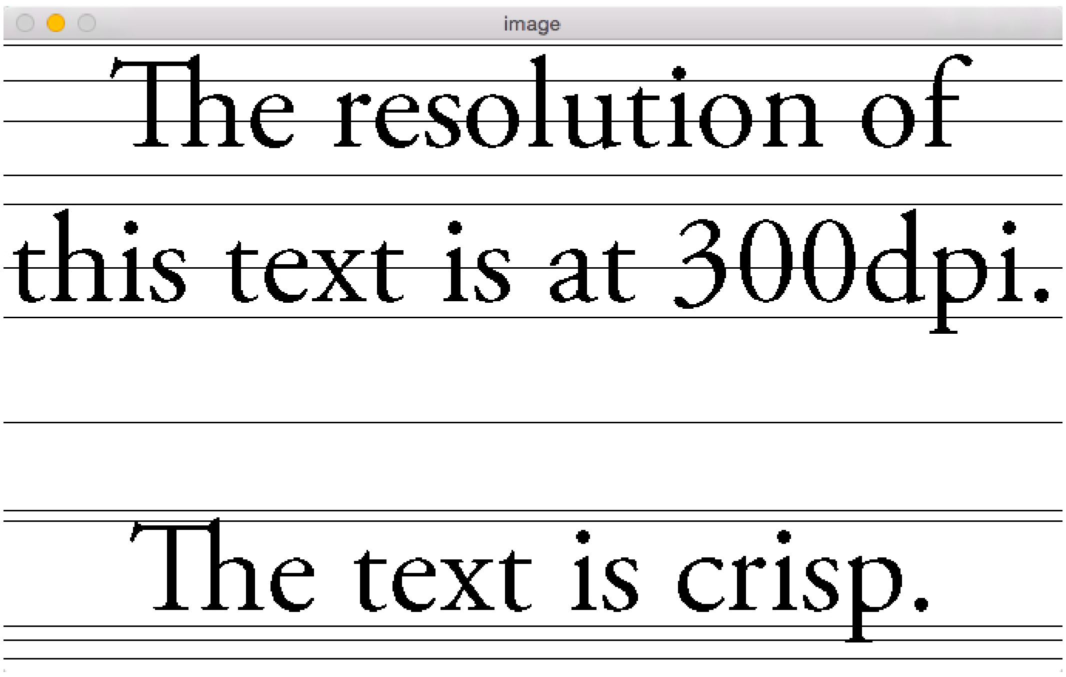
However, one important factor to keep in mind is the DPI of the image under consideration. The image should at least have 300 DPI in order to be able to correctly identify the white regions between the lines (Wilke (2014)).

There is also a limitation while using this approach over splitting in paragraph breaks. Tesseract OCR does not just scan the image, it also interprets the results. So if the line ends with a partial word ("extrac-") it is capable of finding the end of the word ("-ting") at the beginning of the next line. If the image is cut and attempt to process the lines one by one, the engine will hence miss all of those. Although, in many documents splitting words between lines are not done and so, this limitation can be safely overlooked (Wilke (2014)).

The next question that needs to be addressed is, 'What is a white line?' Assuming that the documents are black ink on white paper, when a document is scanned, the white paper appears as light gray in the image. The idea is pretty simple: to compute the average of X number of pixels and see how light or dark they are, if light enough, then those pixels can be considered white (binarisation) (Wilke (2014)).

One solution to identify the white regions in a text image is to calculate the sum of the values of each row and compare it against the next row find the difference between the two rows. If the difference is zero, there is a high probability that the two lines are white (Cyriel (2013)).

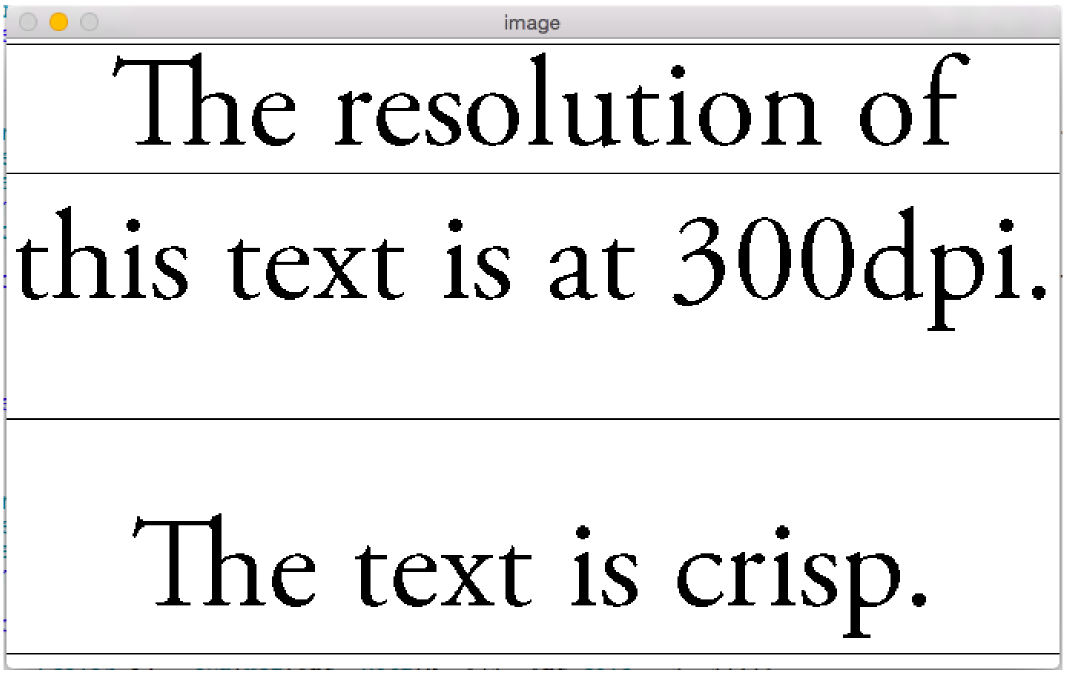
Even though this solution is straight forward, there is a chance that two very similar non-white lines maybe returned by this method. Since the lines contain similar number of black and white pixels, the sum calculated will be equal and hence will falsely count as a white region.



The resolution of
this text is at 300dpi.
The text is crisp.

Figure 6.4: Similar adjacent non-white lines

To avoid this, an extra check needs to be done to ensure that only white lines are returned by this method, since the important issue that is being tackled is to cut the image only in the non-important areas of the image. The logic of this method lies in the fact that non-white lines have a lower sum due the existence of black pixels, in comparison to white lines. A histogram can be plotted to identify the whitest lines in the image and returning only those regions as plausible regions to cut.



image

The resolution of

this text is at 300dpi.

The text is crisp.

Figure 6.5: Identifying only white lines

This is the method that is currently being used to identify the white regions in the page. One caveat to make note is that an extra functionality needs to be added to optimize the number of images that needs to be generated. This is because, too many images can cause the overhead of parallel execution to increase the overall time-taken to process the sub-image.

6.4 Approaches to splitting image

The next step after identifying the non-important (white) regions in the image, was to split the image in those areas.

6.4.1 Identifying the splitting locations

A method line() from the drawing functions API of OpenCV was used to draw a horizontal line on the image on the regions that were identified as white, by the findWhiteRegions method. The method takes in the starting and ending point of the line segment and the width and color of the line that needs to be drawn. This was an effective method to identify whether the regions that were identified were actually white or if the method was returning non-white but similar lines also.

The usage of this method was crucial in understanding the need for finding specific splitting locations in the image. This is because, the findWhiteRegions method returned a huge array of y axis values. When a black line of width 1 px was drawn, almost the entire image was colored black. This meant that only the middle-most white line between two occurrences of non-white lines had to be considered for drawing a black line. Rather, this line is the splitting location.

6.4.2 Splitting into smaller images

After finding the specific splitting locations, the image had to be divided into an array of sub-images. The Rect() method of OpenCV was used to cut a small section from a parent image. The method takes in the parent image, the starting coordinates to cut from and the width and height of the sub-image. This method was looped until all the parts are obtained.

6.5 OpenCL

At this stage, an image of a page under the camera was divided intelligently into sub-images. The next was to understand the working of OpenCL and write the kernel source to gain the necessary speed-up.

OpenCL is used to accelerate in parallel, compute intensive problems by leveraging CPUs and GPUs. Before choosing OpenCL there was a checklist that had to be answered on whether the library was the correct choice and if it was necessary.

- Is there a parallel workload?
- Or can a parallel workload be earned?
- The location of the data
- The destination of the data

If the answer to either of the first two points were yes, and if the outcome of using a parallel computation would be significant, then using OpenCL was the correct choice. The subsequent bullet points helped in understanding the use case better.

In the case of this project, the parallel workload was evident - which was to compute(extracting text from image) multiple parts of the page at the same time. The location of the data comes from the CPU and the destination can either be the GPU (if available) or the CPU.

Its a given that the OpenCL device will win for a situation like this. The win provided by the OpenCL computation covers for the transfer costs to and from the CPU memory to the OpenCL device memory.

Time taken for transferring to OpenCL device is none if it is the CPU because the memory space is shared. Maybe nothing if we are using the integrated GPU (my machine consists of Intel HD 5000 Graphics which is integrated). 'Maybe', because if its using buffers and not involving images, the transfer costs are nil. If our data comprises of an image (which is the case), then the device needs to maintain a copy of the image and hence the transfer costs come into play. For a discrete GPU such as an AMD Radeon or Nvidia, the transfer costs exist.

If the transfer cost is high, it is better to use the CPU or the integrated GPU as the OpenCL device, to reduce the transfer costs. It is important to note the display device to the user. If the display device is the GPU, then there is no need to transfer the computed image by the device back to the host, thereby saving that one time transfer cost and leaves the host to do other computation.

6.5.1 Programming Model

The programming model of OpenCL comprises of a C-like programming language and a Runtime API.

Its C-like as it contains the C language plus useful additional data types and built-in functions. Also, the kernel operates on a domain such as a 2D domain.

The Runtime API of OpenCL is broadly divided in to 3 stages - discovery, setup and execution. Discovery includes the parts such as identifying the OpenCL devices that are available and identifying from these available devices, the best way to break up the work.

The setup covers the portions such as compiling the kernel source written and setting aside memory to write the result.

Finally, the execution covers the set of commands that do the work. These include the run commands that computes the algorithm and stores the result in the allocated memory.

6.5.2 Execution Model

The Execution Model of OpenCL consists of three primary components - kernel, program and application.

Kernel, contains the basic unit of executable code which is essentially C-like program. The code can cater to either data-parallel or task-parallel execution.

A program, is a collection of kernels and other functions. This program is analogous to a dynamic library.

Applications queue kernel execution instances either in-order or out of order.

6.5.3 Scope for improvement

Just like the use of any other library, the usage of OpenCL can be optimized. A slow start-up can be created by the use of the methods - clBuildProgram, clCompileProgram, clLinkProgram. This happens when the Kernel is loading and when the C program is compiled at runtime. However, if its stored as an executable binary in the cache this significantly reduces the time taken for an OpenCL program during execution. Saving binary to cache after building results in the program executing in 0.1ms

6.6 Whitelist characters recognised by Tesseract

Slightly deviating from the main focus of this project, the following method was added to slight improve the accuracy of the text recognition.

Tesseract matches the words extracted from an image against a dictionary to improve the accuracy of words. Words such as 'world' that may be wrongly recognised as 'wor1d'. These will be corrected after parsing through the English dictionary. However, since the use case of this project includes research papers and books, there will be several occurrences of formulae, Nouns and equations. Due to this, the dictionary parsing has to be switched off.

In order to maintain the accuracy after switching off the dictionary parsing, all the allowed characters such as the English alphabets, numbers and common symbols have to be white-listed. The rest of the characters are replaced with a blank character. Even though, this may not increase the accuracy of the recognised text by a lot, its the first step towards improving the quality of the output from the OCR engine.

6.7 POSIX Threads

As mentioned in a previous section of this report, one of the goals were to make ACP-Paper cross-platform compatible. The previous code check-in was making use of Windows Synchronisation libraries for implementing threads. POSIX threads was a chosen replacement due to its cross-platform compatibility.

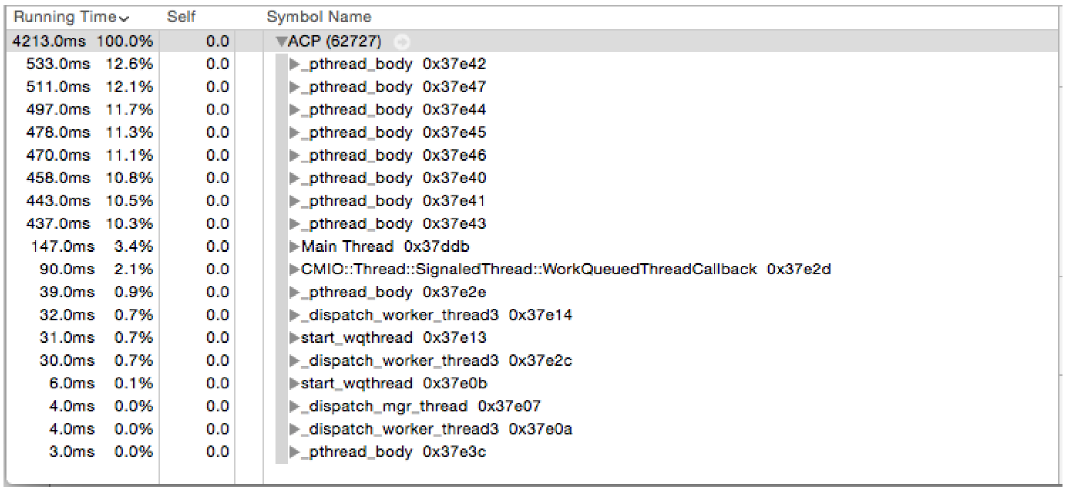
Threads are a set of instructions that can be scheduled to run independently by the operating system. In a shared memory architecture, threads can be used to implement parallelism. In the project's use case, each thread processes an image and returns to the main with the extracted text (Blaise Barney (2014)).

Almost all hardware vendors develop their own proprietary versions of threads. This has made it difficult for developers to develop portable threaded programs, as each version varies significantly from the other. With the goal of achieving full advantages of threads, and to enable better portability, a standard programming interface was to be developed in 1995. Nowadays, each hardware vendor supports Pthreads along with their proprietary API.

POSIX Threads or Pthreads was developed as a standard for UNIX systems. However, the portable, open-source pthreads-w32 serves as a wrapper for Windows machines (Wikipedia (2015b)). Hence, Pthreads aligns with the cross-platform goal of the AutoComPaste-Paper project. Therefore, Pthreads has been chosen to replace the Windows threads used in the previous code check-in.

A pool of threads is created along with a structure that serves as the data-structure for the thread data. The structure consists of a Processor object, the image to be processed and the id that represents the thread. A method for extracting the text from the image is created which simply sends the image to the OCR engine and receives the output.

Once all the threads finish computing the respective extractText() methods, the control returns to the main along with the extracted text.



Running Time		Self	Symbol Name
4213.0ms	100.0%	0.0	▼ACP (62727)
533.0ms	12.6%	0.0	▶_pthread_body 0x37e42
511.0ms	12.1%	0.0	▶_pthread_body 0x37e47
497.0ms	11.7%	0.0	▶_pthread_body 0x37e44
478.0ms	11.3%	0.0	▶_pthread_body 0x37e45
470.0ms	11.1%	0.0	▶_pthread_body 0x37e46
458.0ms	10.8%	0.0	▶_pthread_body 0x37e40
443.0ms	10.5%	0.0	▶_pthread_body 0x37e41
437.0ms	10.3%	0.0	▶_pthread_body 0x37e43
147.0ms	3.4%	0.0	▶Main Thread 0x37ddb
90.0ms	2.1%	0.0	▶CMIO::Thread::SignaledThread::WorkQueuedThreadCallback 0x37e2d
39.0ms	0.9%	0.0	▶_pthread_body 0x37e2e
32.0ms	0.7%	0.0	▶_dispatch_worker_thread3 0x37e14
31.0ms	0.7%	0.0	▶start_wqthread 0x37e13
30.0ms	0.7%	0.0	▶_dispatch_worker_thread3 0x37e2c
6.0ms	0.1%	0.0	▶start_wqthread 0x37e0b
4.0ms	0.0%	0.0	▶_dispatch_mgr_thread 0x37e07
4.0ms	0.0%	0.0	▶_dispatch_worker_thread3 0x37e0a
3.0ms	0.0%	0.0	▶_pthread_body 0x37e3c

Figure 6.6: Lesser time taken per thread

Chapter 7

Project Summary

The aim of this project was to focus on improving the speed of AutoComPaste-Paper as a system.

7.1 My Contributions

7.1.1 Re-engineered AutoComPaste-Paper

The state of the ACP-Paper code base before the beginning of this project was in a procedural style. I have created a complete revamp of the existing code check-in by implementing an Object-oriented version. A clear architecture was designed with the Supervisor and the implementation was documented. The advantages and reasons of implementing an object-oriented design has been clearly stated in a previous section of this report.

7.1.2 Speedup

There were two approaches taken towards obtaining an overall speedup in the existing project. The first approach was to use OpenCL to execute the extractTextFromImage() method in parallel to obtain speedup. This method was at a higher level. The second approach was to test the inherent ability of

Tesseract's OpenCL functionality.

For the first approach, the white regions in the image were successfully identified and the page was divided in non-important regions. A major obstacle of incorrectly identifying similar non-white lines was handled by writing an addtional check. Before proceeding with the next step of writing the kernel source, it was decided to check if Tesseract was already saturating the cores of the CPU or GPU when the inherent OpenCL functionality was used.

A thorough research on how Tesseract is using OpenCL for offloading some of the compute intensive modules was done. A Tesseract version with OpenCL compatibility was compiled and tested on my machine. There were several limitations in the Tesseract source. There were multiple areas where malloc errors occurred. Tesseract was not handling the limitation in Apple devices where the work group size could not be 2-D, i.e., the second dimension had to be 1.

Hence, these two different approaches can now be compared and tested against each other to find the approach that results in a better speedup.

7.2 Uses of ACP-Paper

The RBR section of the Central Library only allows for loaning the books for only two hours. ACP-Paper allows for efficient scanning of papers and books for later usage. Most laptops these days come with a webcam, multiple cores and a GPU, which makes ACP-Paper run without the need for any extra requirements.

7.3 Limitations

Even though most laptops have a webcam facility, in order to capture an image of the research material, an additional facility is required to face the camera downwards.

The project relies on the availability of OpenCL compatible devices in order to obtain the necessary speedup. Without a compatible GPU or free cores of CPU, the time taken to execute the program directly depends on the complexity of the image to be processed.

The current implementation of the system is unable to detect rotation beyond 70 degrees. However, this limitation can be overlooked as such an incident would be obvious to the user and the necessary rotation will be performed.

7.4 Challenges

There were several roadblocks during the course of this project. A majority of those can be attributed to the lack of proper documentation of the libraries that were used.

For instance, there does not exist a clear documentation for compiling the source and linking the necessary libraries for OpenCV and Tesseract to work together while using Xcode. In order to use the OpenCL functionality that is available in Tesseract, a specific parameter needs to be enabled along with setting the required headers and libraries of the OpenCL framework. A clear documentation does not exist for integrated GPUs such as Intel HD 5000.

There is little to no evidence of previous work on improving the speed of Tesseract OCR engine using OpenCL other than using the functionality that comes along with the Tesseract source. Due to these limitations, the time taken to implement a speedup took longer than expected.

7.5 Future suggestions

The time taken for execution of the current implementation of AutoComPaste-Paper relies on the hardware configuration of the user's machine. However, if a server with a good configuration (more than 4 CPU cores, GPU) can be obtained, the process of extracting the text by OCR engine can be offloaded to the server. This will require an API to access the necessary functionality. The advantages of this implementation is that most of the intensive computation is now taken care of by the server. This reduces the workload on the user's machine and also removes a major roadblock in terms of power consumption in case the program needs to implemented for a mobile phone or a tablet.

While using a server can be advantageous for complex images, a simple check can be made for less complex images and the computation can be done by the user's machine itself, thereby saving the request and response time to and from the server.

The limitation that comes along with this implementation is that the user will no longer be able to work with the OCR extension offline, since a continuous connection to the server is required in order to obtain the text from the research material in use.

Bibliography

(2014). Gnu ocrad manual.

Blaise Barney, L. L. N. L. (2014). Posix threads programming. *Lawrence Livermore National Laboratory Tutorial.*

Boris Epshtein, Eyal Ofek, Y. W. (2008). Detecting text in natural scenes with stroke width transform. Paper, Microsoft Corporation.

Capture2Text (2015). Capture2text.

Cosenza, M. Opencv.

Cyriel (2013). Computer vision: How to split horizontally an image by the line with least entropy? *Stack Overflow.*

Foundation, F. S. (2014). Gnu ocrad.

Foundation, T. S. (2015). Advantages and disadvantages of object-oriented programming (oop).

Google (2015). Tesseract ocr.

Group, K. Opencl - the open standard for parallel programming of heterogeneous systems.

Kay, A. (01/07/2007). Tesseract: an open-source optical character recognition engine.

Kendall, W. (2015). Mpi tutorial introduction.

Kigo, K. (1993). Improving speed of japanese ocr through linguistic preprocessing. *Document Analysis and Recognition*, pages 214 – 217.

Kwok, K. (2015). Project naptha.

Leyden, T. (2015). Open source ocr-as-a-service using tesseract and docker.

Liviu Petrisor Dinu, R.-T. I. and Popescu, M. Local patch dissimilarity for images. Master's thesis, University of Bucharest, No. 14, Academei Street, Bucharest, Romania.

ODell, N. (2011). Extracting paragraphs from ocr text.

OpenCV. Bilateral filter.

OpenCV (2014). Smoothing images.

Palkovic, A. (2008). Improving optical character recognition. Master's thesis, Villanova University, United States.

Perrault, A. Detect text.

Slovak, P. (2014). Tesseract meets the opencl - first test. Technical report, Slovak Open Source.

Tesseract, G. (2015). Improving the quality of output.

Wikipedia (2015a). Otsu's method.

Wikipedia (2015b). Posix threads.

Wilke, A. (2014). Parallelise ocr by sending blocks/ paragraphs of text. *Stack Overflow*.

Zeros, D. (2015). How tesseract uses opencl?

Zhao, S. (2012). Autocompaste: Auto-completing text as an alternative to copy-paste.