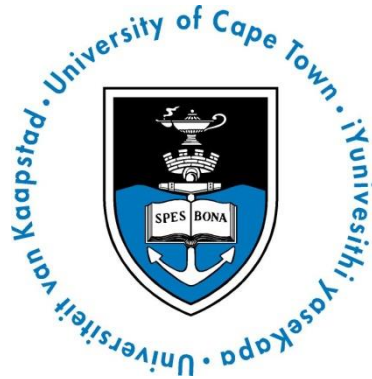


GPU Acceleration of the Fynbos Leaf-based Online Recognition Application



Prepared by:

Khagendra Naidoo
NDXKHA001

Department of Electrical Engineering
University of Cape Town

Prepared for:

Simon Winberg

Department of Electrical Engineering
University of Cape Town

September 2015

Submitted to the Department of Electrical Engineering at the University of Cape Town in partial fulfilment of the academic requirements for a Bachelor of Science degree in Electrical and Computer Engineering.

Key Words: FLORA, GPU acceleration, heterogeneous computing, image processing, fynbos, CUDA

Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this final year project report from the work(s) of other people, has been attributed and has been cited and referenced.
3. This final year project report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof

Name: Khagendra Naidoo

Signature: _____

Date: 14 October 2015

Terms of Reference

This thesis report serves as one of the final deliverables for the 4022S course. My supervisor is Dr Simon Winberg, this project involves the development of a set of code modules or a modified version of the FLORA program that makes use of GPU acceleration to improve system performance. This project relates to GPGPU (General Purpose computing on Graphics Processing Unit) computing, program optimisation, image processing and machine learning. The program or modules to be developed are required to be compatible with the existing FLORA front end system and should provide a system speedup when implemented.

Some expected deliverables are:

- A GPU accelerated implementation of the FLORA system
- Code modules that are easy to understand and modify
- Experimental data verifying the achieved performance of the new FLROA system
- Detailed analysis of system behaviour and performance

Acknowledgements

First I would like to thank my supervisor, Dr Simon Winberg, for all his advice and guidance throughout the course of this project. Secondly I would like to thank my parents for all their support throughout my academic career and last but not least I'd like to give special thanks to my best friend Ami, for always being there for me through thick and thin.

Abstract

The Fynbos Leaf-based Online Recognition Application is a web based application that can be used to identify different species of fynbos plants based on images of their leaves. The application uses several computer vision based algorithms and pattern matching algorithms in order to compare a submitted image against a database of known species. The application currently suffers from poor speed performance due to the large processing load that must be bared by the server's CPU. Therefore the aim of this project is to modify the existing FLORA application to run on a heterogeneous framework, allowing the processing load to be split effectively across the CPU and GPU to achieve significantly better performance.

The original FLORA implementation was analysed in order to determine exactly what processing functions are used by the system and how they work. Once a sufficient understanding of the system was gained the system was re-implemented in C++. This C++ implementation formed the basis on which a GPU accelerated implementation could be built.

Bottlenecks were then identified in the C++ implementation and GPU accelerated system implementations were developed with different GPU accelerated functions in order to remove those bottlenecks. The developed FLORA implementations were then analysed to check prediction accuracy and optimised further where possible before the final fully optimised implementations were produced.

The new FLORA implementations were then integrated with the existing FLORA front end and tested to ensure they worked correctly.

The results of this project showed that the C++ implementation performed the best with standard input, while the best CUDA accelerated implementation only performed well with high resolution input. It was also shown that initialisation overhead was a major bottleneck in the system but it was shown that this could be avoided in order to achieve much better system performance.

Table of Contents

Declaration	i
Terms of Reference	ii
Acknowledgements.....	iii
Abstract.....	iv
List of Figures.....	viii
1. Introduction.....	1
1.1 Project Background.....	1
1.2 Project Objectives	1
1.2.1 Main Objective	1
1.2.2 Secondary Objective.....	1
1.2.3 Problems to be investigated	1
1.3 Project Requirements.....	2
1.4 Project Motivation.....	2
1.5 Scope and Limitations.....	2
1.6 Plan of development.....	2
2. Literature Review	4
2.1 Fynbos Leaf-based Online Recognition Application (FLORA).....	4
2.1.1 FLORA Front End Overview.....	4
2.1.2 FLORA Back End Overview	5
2.2 FLORA Back End – Processing Breakdown.....	6
2.2.1 Conversion to Black & White.....	6
2.2.2 Region Detection	7
2.2.3 Rotate	8
2.2.4 Extract Features.....	9
2.2.5 KNN Prediction.....	10
2.3 General-purpose Computing on Graphics Processing Unit (GPGPU).....	11
2.3.1 The GPU Programming Model.....	11
2.4 Past FLORA Optimisation Attempts	12
2.5 Review of Current GPGPU Platforms	12
2.5.1 Open Computing Language (OpenCL)	12
2.5.2 Compute Unified Device Architecture (CUDA)	13
2.5.3 AMD Accelerated Parallel Processing (AMD APP).....	13
2.5.4 Open Source Computer Vision (OpenCV).....	13
2.6 Review of existing GPU Implementations of Processing Algorithms	13
2.6.1 Convert to Black & White.....	13
2.6.2 Region Detection	14
2.6.3 Rotate	15
2.6.4 Extract Features.....	15
2.6.5 KNN Prediction.....	16
3. Methodology	18
3.1 Outline	18
3.2 Phase 1: Research and Literature Review	18
3.3 Phase 2: Selection & Set Up of Development Tools	18
3.3.1 Host Computer	18
3.3.2 Chosen GPGPU platform and programing environment.....	18

3.4	Phase 3: Initial FLORA Analysis and Test Data Acquisition	19
3.5	Phase 4: Optimised CPU Implementation	20
3.6	Phase 5: GPU Design Iteration	21
3.7	Stage 6: Performance Analysis	22
3.7.1	Prediction Accuracy Calculation Method	22
3.8	Stage 7: Implementation Optimisation	22
3.8.1	Execution Time & Prediction Accuracy Optimisation	22
3.8.2	Image Resolution Optimisation	22
3.9	Final Integration with FLORA Front End	22
4.	FLORA C++ Design.....	23
4.1	Overview.....	23
4.2	A Note on OpenCV Image Processing Library Initialisation Overhead.....	24
4.3	Image Input	24
4.4	Greyscaling	24
4.5	Otsu Thresholding.....	25
4.6	Finding Image Contours	25
4.7	Image Blurring	26
4.8	Calculation of Image Orientation.....	27
4.9	Calculation of Rotation Matrix.....	27
4.10	Apply Rotation Matrix Transformation	28
4.11	Determining Region Convex Hull	28
4.12	Determining Region Bounding Box	29
4.13	Region Area & Perimeter Length.....	29
4.14	Convex Hull Area & Perimeter Length	30
4.15	Calculation of Feature Descriptors	30
4.16	Prediction Using KNN Classifier	31
4.16.1	Creation & Training of a KNN Classifier.....	31
4.16.2	Prediction Using KNN Classifier.....	32
4.16.3	Testing of the KNN Classifier	33
5.	FLORA CUDA Acceleration	34
5.1	Initial Setup	34
5.2	A Note on OpenCV CUDA Accelerated Image Processing Library Initialisation	35
5.3	Iteration 1: Image Greyscaling	35
5.4	Iteration 2: Apply Rotation Matrix.....	36
5.5	Iteration 3: Otsu Thresholding.....	37
5.6	Iteration 4: Find Image Contours	39
5.6.1	Prepare Input Image	40
5.6.2	Kernel 1: Solve Local CCL Problem	41
5.6.3	Kernel 2: Merge Set of Tiles	43
5.6.4	Kernel 3 Updating Tile Border Labels	44
5.6.5	Kernel 4: Update All Labels	45
5.6.6	Scan Final Label Values.....	45
5.7	Iteration 5: Average Filter (Image blurring)	45
5.8	Termination of Iteration	46
6.	Integration with FLORA Front End	46
7.	Performance Results.....	48
7.1	Overview.....	48
7.2	Analysis of Original FLORA Implementation.....	48
7.3	Analysis of Optimised C++ Implementation	48

7.3.1	Comparison with Original FLORA Implementation	50
7.4	Analysis of CUDA Accelerated Functions	51
7.5	Response to Variation of Image Resolution	52
8.	Discussion	53
8.1	Execution Time Performance.....	53
8.2	Prediction Accuracy	54
8.3	Other Advantages of New FLORA implementations	54
9.	Conclusions	55
10.	Recommendations	56
11.	List of References	56
12.	Appendix A: Code Performance Data	60
12.1	Execution Time Profiling Results	60
12.1.1	Original FLORA Octave Implementation	60
12.1.2	FLORA C++ Implementation	60
12.1.3	FLORA CUDA Accelerated Implementations.....	61
12.2	Code Execution Time Profiling Results – High Resolution	63
12.2.1	Original FLORA Octave Implementation with High Resolution Input.....	63
12.2.2	FLORA C++ Implementation with High Resolution Input	63
12.2.3	FLORA CUDA Accelerated Implementations with High Resolution Input.....	64

List of Figures

Figure 2-1 FLORA front end sequence of operations	4
Figure 2-2 View of FLORA frontend home page shown in web browser.....	5
Figure 2-3 Back end sequence of operations.....	5
Figure 2-4 Example of Greyscale Image with Grey level Histogram.....	7
Figure 2-5 Example of 4-connectivity (left) and 8-connectivity (right)	8
Figure 2-6 Parallelisation of Connected Component Labelling [34]	14
Figure 2-7 Example of parallel matrix multiplication.....	15
Figure 2-8 Example of Quickhull Algorithm.....	16
Figure 3-1 Sample images from chosen testing database	20
Figure 3-2 GPU Implementation Design Processes.....	21
Figure 4-1 Overview of FLORA C++ implementation Processing Steps	23
Figure 4-2 Example of Greyscaled Image.....	25
Figure 4-3 Example of Otsu Threshold.....	25
Figure 4-4 Example of Extracted Region Contours	26
Figure 4-5 Example of Blurred Image.....	26
Figure 4-6 Example of Image Rotation.....	28
Figure 4-7 Example of Image Convex Hull Output	29
Figure 4-8 Example of Bounding Box Output.....	29
Figure 5-1 Input Image Used for Profiling (800 X 533 Resolution)	35
Figure 5-2 Example of CUDA Accelerated Output	36
Figure 5-3 Example of CUDA Accelerated Otsu Threshold	39
Figure 5-4 Overview of FLORA GPU CCL Algorithm	40
Figure 5-5: Example of Input Image Split into Tiles	41
Figure 5-6 Example of Input Image with Borders Extended.....	41
Figure 5-7 Graphical Example of Kernel 1 Operation [34]	42
Figure 5-8 Example Application of Kernel 2 [34]	44
Figure 5-9 Example of Image blurred using CUDA Accelerated Blur Function	46
Figure 6-1 Example of FLORA Web App Running the New Backend System.....	47
Figure 7-1 Execution Time of Original FLORA Implementation.....	48
Figure 7-2 Execution Timing of C++ Implementation (Excluding Initialisation Overhead)	49
Figure 7-3 Effect of Image Resolution on Speedup & Prediction Accuracy	52

Code Listings

Listing 4-1: Dummy Operation Called to Initialise OpenCV Image Processing Library	24
Listing 4-2: C++ Greyscaling.....	24
Listing 4-3: C++ Otsu Thresholding.....	25
Listing 4-4: C++ Find Image Contours	25
Listing 4-5: C++ Image Blurring.....	26
Listing 4-6: C++ Calculation of Image Moments.....	27
Listing 4-7: C++ Calculation of Image Orientation Using Calculated Moments.....	27
Listing 4-8: C++ Calculation of Rotation Matrix.....	27
Listing 4-9: C++ Application of Rotation Matrix using a Matrix Transform.....	28
Listing 4-10: C++ Convex Hull.....	28
Listing 4-11: C++ Bounding Box.....	29
Listing 4-12: C++ Region Area & Perimeter	29
Listing 4-13: C++ Convex Hull Area & Perimeter.....	30

Listing 4-14: C++ Calculation of Feature Descriptors	30
Listing 4-15: C++ Pseudocode for KNN Classifier Creation	31
Listing 4-16: C++ Pseudocode for Prediction Using KNN Classifier	32
Listing 4-17: C++ Struct Used for Holding KNN Prediction Results	33
Listing 4-18: C++ Code Snippet Showing Calculation of Prediction Accuracy Percentages	33
Listing 4-19: C++ Pseudocode for Testing of a Classifier	33
Listing 5-1 Dummy Operation Called to Initialise OpenCV CUDA Library	35
Listing 5-2: CUDA Accelerated Application of Matrix Transform	36
Listing 5-3: Method for Uploading & Downloading Images to and from GPU Memory	36
Listing 5-4: Pseudocode for Otsu Thresholding on GPU	37
Listing 5-5: CUDA Accelerated Grey Level Histogram Calculation	37
Listing 5-6: Custom C++ implementation of Otsu's Method	38
Listing 5-7: CUDA Accelerated Thresholding	38
Listing 5-8: C++ Code Used to Resize Input Image for Use with GPU CCL Algorithm	40
Listing 5-9: Pseudocode for Kernel 1 - Local CCL Solver	41
Listing 5-10: Pseudocode for Kernel 2 - Merging Set of Tiles	43
Listing 5-11: Pseudocode for Kernel 3 - Update Border Labels	44
Listing 5-12: Pseudocode for Kernel 4 - Final Label Update	45
Listing 5-13: CUDA Accelerated Averaging Filter	46
Listing 6-1: PHP Script for Calling Original FLORA Backend	46
Listing 6-2: PHP Script for Calling New FLORA Backend	47

List of Tables

Table 2-1 Octave functions used in back end operations	6
Table 3-1 Additional Software Tools	19
Table 6-1 Function Equivalences between Octave and C++ Implementations	49
Table 6-2 Speedup Achieved by C++ Implementation compared to Octave Implementation	50
Table 6-3 Specific Function Speedup of CUDA Accelerated Functions (Excluding Overheads)	51
Table 6-4 Performance Optimisation for CUDA Implementations	51
Table 7-5 Effect of Image Resolution on Speedup & Prediction Accuracy	53
Table 12-1 Execution Time of Original FLORA Implementation for Low Resolution Input	60
Table 12-2 Execution Time of FLORA before GPU Design Iteration 1 and 2 for Low Resolution Input	60
Table 12-3 Execution Time of FLORA after GPU Design Iteration 2 for Low Resolution Input	61
Table 12-4 Execution Time of FLORA after GPU Design Iteration 3 for Low Resolution Input	61
Table 12-5 Execution Time of FLORA after GPU Design Iteration 4 for Low Resolution Input	62
Table 12-6 Execution Time of FLORA after GPU Design Iteration 5 for Low Resolution Input	62
Table 12-7 Execution Time of Original FLORA System	63
Table 12-8 Execution Time of FLORA before GPU Design Iteration 1 & 2 with High Resolution Input	63
Table 12-9 Execution Time of FLORA after GPU Design Iteration 2 with High Resolution Input	64
Table 12-10 Execution Time of FLORA after GPU Design Iteration 3 with High Resolution Input	64
Table 12-11 Execution Time of FLORA after GPU Design Iteration 4 with High Resolution Input	65
Table 12-12 Execution Time of FLORA after GPU Design Iteration 5 with High Resolution Input	65

List of Equations

Equation (2-1)	6
Equation (2-2)	7
Equation (2-3)	7
Equation (2-4)	7

Equation (2-5)7

Equation (2-6)8

Equation (2-7)8

Equation (2-8)8

Equation (2-9)9

Equation (2-10).....9

Equation (3-1) 22

Equation (4-1) 27

1. Introduction

1.1 Project Background

The Fynbos Leaf-based Online Recognition Application (FLORA) is a web based application that identifies fynbos plant species by analysing images of their leaves. The application has been developed by S. Katz and currently has a prediction accuracy of about 89.74%. There are however several limitations that exist with FLORA in its current state, one of the most serious being speed performance, especially with high resolution inputs.

FLORA makes use of several image processing techniques and algorithms that fall within the field of computer vision. Computer vision is the area of computer science that deals with the acquisition and processing of real world data, particularly visual data, in order to produce numerical data that a computer can understand. The goal behind computer vision algorithms is to give computers a way to 'understand' or 'recognise' real world data, such as images or sound, so that the decisions can be made and work can be done based on the perceived data.

The field of computer vision has evolved significantly over the past few decades as the capabilities of modern computers have improved. One of the most recent innovations in the field is the implementation of computer vision algorithms on graphical processing units or GPUs. This is made possible through the emergence of GPGPU platforms that allow acceleration of general purpose programming algorithms by mapping them to GPU compatible algorithms. GPU's can achieve much greater throughput compared to CPUs when running highly parallelised algorithms this improves program performance and also reduces the load on the host CPU.

This project will involve the optimisation of FLORA by modifying or re-implementing FLORA's current computer vision and image processing algorithms on a GPU in an attempt to improve the application's performance.

1.2 Project Objectives

1.2.1 *Main Objective*

The main objective of this project is to create a GPU accelerated implementation of FLORA and integrate it into the existing FLORA web based framework.

1.2.2 *Secondary Objective*

Provided the first objective is met the second objective of this project will be to achieve a speedup of at least 10 times compared to the original FLORA implementation.

1.2.3 *Problems to be investigated*

There are several problems that were to be solved through this project, they include:

- How will application performance be measured?
- What are the performance goals?
- What is the best GPGPU implementation to use?
- Is the GPU optimised implementation better than a CPU optimised implementation?

1.3 Project Requirements

There are certain requirements that must be met in order to ensure the new FLORA implementation not only meets the project objectives but will have real world use beyond completion of this project, these are:

- The new FLORA implementation must be built using free or open source tools to allow for future development
- The code produced for this new implementation must be well commented and easy to understand to allow for future development
- The new FLORA implementation must maintain at least 80% prediction accuracy to be considered successful

1.4 Project Motivation

The FLORA system in its current state suffers from poor speed performance and this is a major issue due to the fact that FLORA is a web based app that runs off a single server. Web based apps are intended to be accessible by many users at once and this means that even a processing time of a few seconds can quickly add up as more users try to make requests to the server at a time. Long waiting periods can effectively render the app unusable and therefore it must be ensured that the app server can perform well under as large a load as possible.

One can alleviate this problem by either improving the server hardware to reduce processing time or implementing multiple servers at a time to share the processing load, however this is an expensive solution and should only be considered as a last resort. The most cost effective solution is to try and optimise the application itself. By modifying the application to better utilise the existing hardware one can improve performance without any additional hardware cost.

Application acceleration through GPU implementation has shown that it is well suited to computer vision algorithms [1]–[5] producing speedups of several times, especially when working with high resolution images, compared to CPU implementations. The cost of high capability GPUs has also decreased significantly over the past few decades, allowing GPUs to achieve better cost to performance ratios when used for image processing or computer vision based tasks. Therefore due to FLORA's heavy use of computer vision and image processing based algorithms, GPU acceleration provides a cost effective solution that should achieve a good speed up.

1.5 Scope and Limitations

The scope of this project is limited to optimisation of the FLORA system, the project is also limited by time constraints. The project scope is therefore limited as follows:

- Only processing algorithms currently used by FLORA will be considered for optimisation
- The performance optimisation attempted in this project will be limited to execution time or 'latency' performance.
- Other performance factors such as hardware utilisation factors and power consumption will not be taken into consideration except for prediction accuracy.

This project was given 12 weeks in total to be carried out, therefore the complexity of the GPU implementation will be limited to allow completion within that time frame.

1.6 Plan of development

This project has been organised into 10 Chapters as follows:

Chapter 2 Gives a summary of all the relevant research done and literature reviewed in the areas of GPU computing, image processing and machine learning. A review of the original FLORA application is also given in this chapter as this forms the basis of the project work

Chapter 3 Breaks down the methodology used throughout this project to achieve the project objectives. It is shown how the project was split into phases, what goal was to be achieved in each phase and the methods used to achieve those goals

Chapter 4 This chapter details the design and implementation of the C++ FLORA implementation. Each of the processing steps is explained in detail.

Chapter 5 This chapter details the design and implementation of the CUDA accelerated functions developed during this project.

Chapter 6 In this Chapter it is shown how the new FLORA implementations were successfully integrated into the existing FLORA frontend.

Chapter 7 Provides all the performance results of the C++ and CUDA accelerated FLORA implementations

Chapter 8 This chapter discusses various aspects of the new FLORA implementations including results analysis and observations made

Chapter 9 Conclusions are drawn based on the results analysis and observations

Chapter 10 Recommendations are made for future work

2. Literature Review

This section provides a concise review of past technical papers, journal articles, books and other sources consulted during the course of this project. The literature is split into further sections based on the different aspects dealt with throughout the design, implementation and evaluation phases.

2.1 Fynbos Leaf-based Online Recognition Application (FLORA)

This project is based on FLORA, a leaf recognition web app created by Shaun Katz in 2011[6]. The web app allows one to upload an image of a fynbos leaf through a web browser friendly front end system and then have a back end server process the image in order to try and predict the species of the uploaded leaf. The app has been designed to be hosted on a LAMP (Linux Apache MySQL PHP) server, with the front end written purely in HTML and PHP. The back end functionality is written in Octave code and run using GNU Octave through a PHP interface.

2.1.1 FLORA Front End Overview

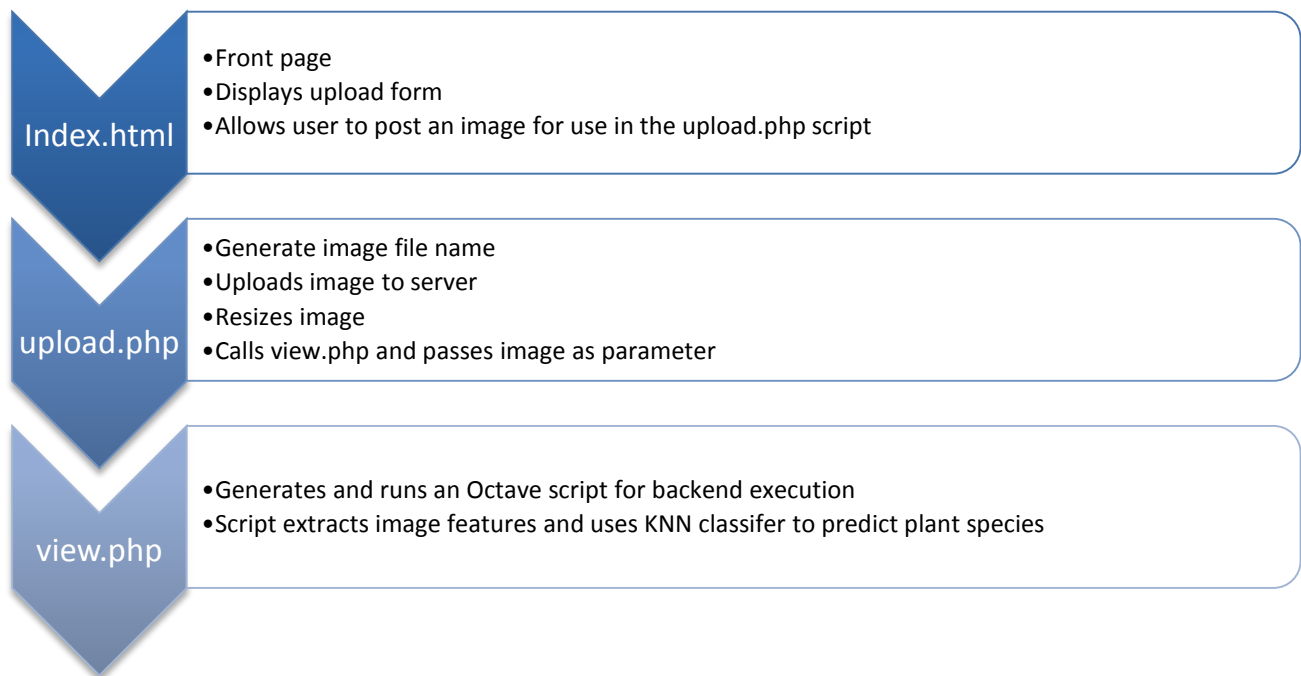


Figure 2-1 FLORA front end sequence of operations

Figure 2-1 shows a brief overview of how the browser based front end system is used to acquire an image and upload it to the server based back end. The front end system presents the user with the web interface shown in Figure 2-2, the user can then choose a file to upload and send it to the server for processing on the server backend system. More details can be found in [6]. With the exception of image resizing, the front end system only involves file input and output and calls the backend to process input. The only performance bottleneck in the front end system is the uploading time of the image, which is limited by image file size and connection speed. These factors are completely

determined by the client and therefore cannot be optimised on the server side.

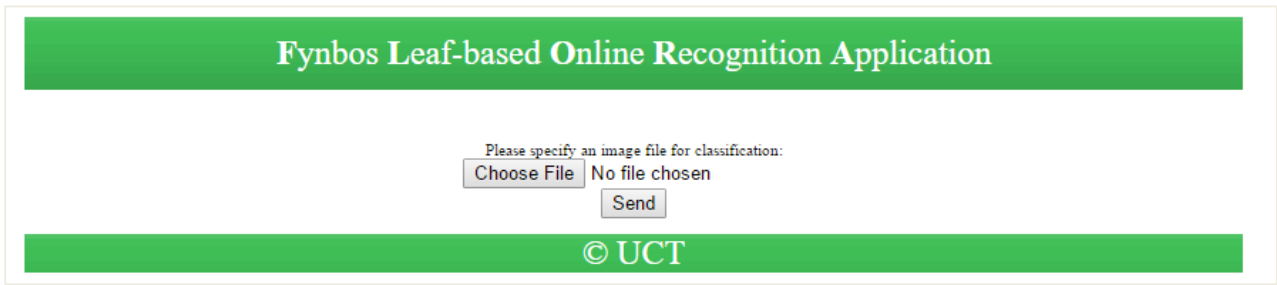


Figure 2-2 View of FLORA frontend home page shown in web browser

2.1.2 FLORA Back End Overview

Figure 2-3 shows a brief overview of the processing done by the back end system in order to make a prediction and Table 2-1 further breaks down the Octave functions used for each operation. For a more detailed breakdown of the Octave code refer to [6]. The processing stages are further expanded upon in section 2.2.

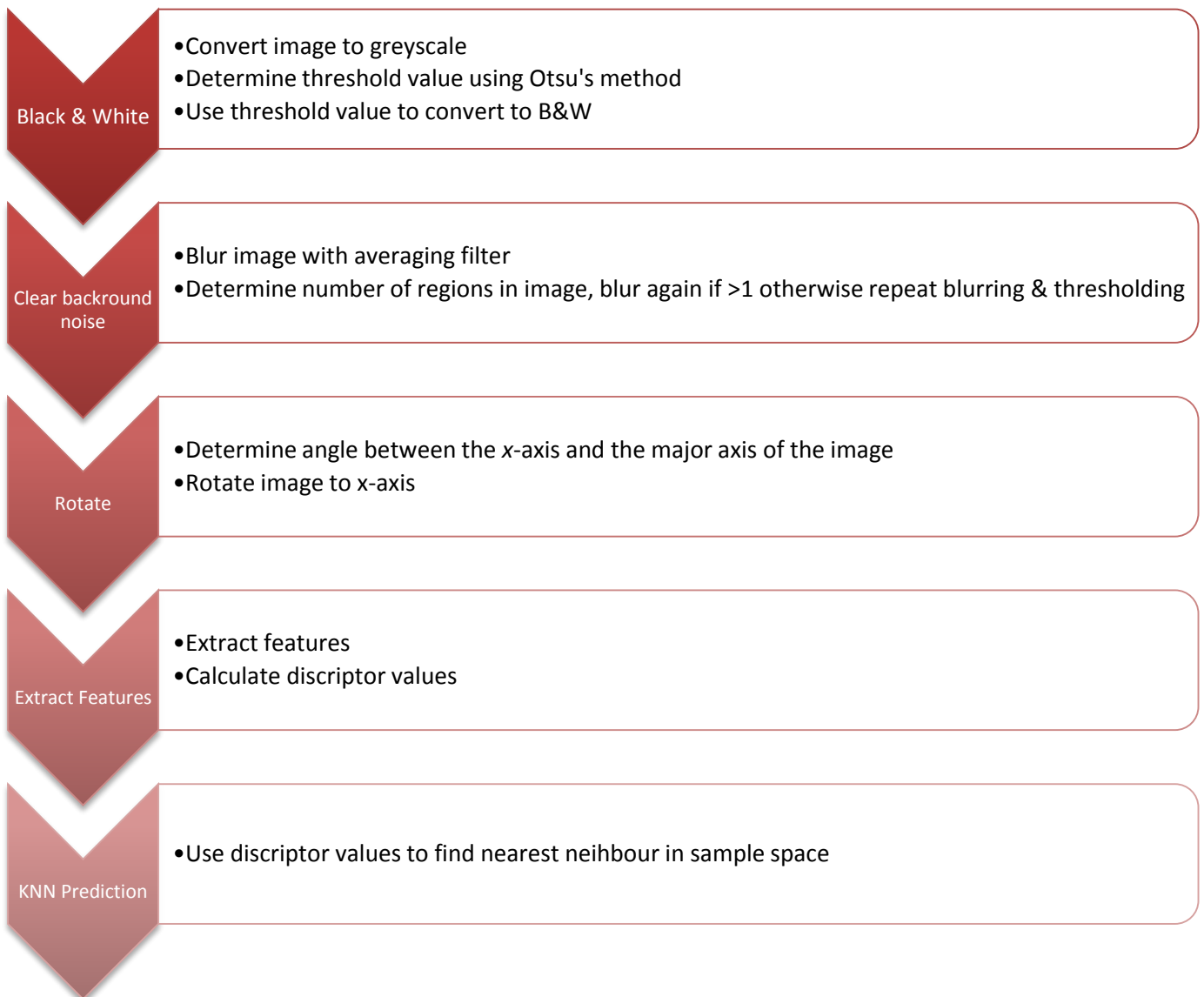


Figure 2-3 Back end sequence of operations

Table 2-1 Octave functions used in back end operations

Operation	Octave functions used
Convert to Black & White	<code>rgb2gray()</code> – Converts an image from colour to greyscale <code>graythresh()</code> – Determines threshold value using Otsu’s method <code>im2bw()</code> – Use threshold value to convert an image to black and white
Clear background noise	<code>filter2()</code> – Used to apply an averaging filter to the image <code>bwlabel()</code> – Determines number of regions in binary image using connected component labelling
Rotate	<code>regionprops('orientation')</code> – determines angle between the x-axis and the major axis of the ellipse that has the same second-moments as the region <code>imrotate()</code> – Rotates an image by a specified angle
Extract Features	<code>regionprops()</code> – Extracts certain features from region such as area, bounding box, perimeter etc. <code>imConvexHull()</code> – Produces a convex hull that fits the region in the image.
KNN Prediction	This operation used custom code by S. Katz [6]. It is a standard 1 st nearest neighbour algorithm.

2.2 FLORA Back End – Processing Breakdown

From section 2.1.2 it can be seen that there are several processing steps that are used in order to achieve a leaf image prediction. These processing steps make up the total of work done by the back end system and therefore they will be the targets for optimisation and GPU implementation in this project. The following sub sections detail the theoretical background for each processing step used.

2.2.1 Conversion to Black & White

The first step in the processing chain is to convert the colour input image to greyscale this is achieved by the `rgb2gray` function which removes the hue and saturation information from an image, but retains the luminance. It replaces the Red, Green and Blue values of each pixel with a single greyscale intensity value. The greyscale value is calculated as a weighted sum of the RGB values as follows:

$$I_{gray} = 0.2989I_{red} + 0.5870I_{green} + 0.1140I_{blue} \quad (2-1)$$

After greyscaling, the image is converted to a binary image (black and white) using Otsu’s thresholding [6]. Otsu’s method is a global binarization method proposed by N. Otsu [7], it is designed to work on images in which there is a clear distinction between the image foreground and background [8]. Pixels of an image are split into two classes C_0 (foreground) and C_1 (background). The classes are separated by a threshold T such that pixels of class C_0 have grey intensities in the range $[1, 2, \dots T]$ and pixels of class C_1 have grey intensities in the range $[T + 1 \dots L]$ where L is the maximum intensity value. In order to calculate the value of T a grey level histogram is used, this is a histogram that shows the distribution of

the grey level intensities of pixels in an image. Figure 2-4 shows an example of a greyscale image with its corresponding grey level histogram (image source: [9]).

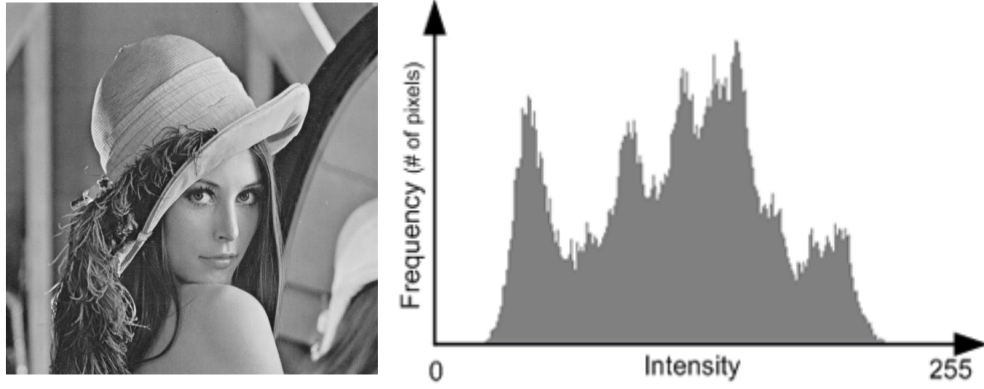


Figure 2-4 Example of Greyscale Image with Grey level Histogram

The histogram can be calculated easily by iterating through all the pixels of a greyscale image and counting the number of pixels found of different intensity levels. Once the histogram is obtained it is normalised and treated as a probability distribution [8]. An exhaustive search is then done to minimise the intra-class variance, defined as a weighted sum of the variances of the two classes:

$$\sigma_{\omega}^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t) \quad (2-2)$$

Where ω_i are the probabilities of the two classes, σ_i^2 the variances of the classes and t the threshold value. Otsu shows that minimising the intra-class variance is the same as maximising inter-class variance:

$$\sigma_b^2(t) = \sigma^2 - \sigma_{\omega}^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2 \quad (2-3)$$

Where μ_i are the class means. Otsu also provides a concise algorithm for finding the desired value of t called 'Otsu's method'[7]:

1. Compute the grey level histogram and probabilities of each grey intensity level
2. Initialise $\omega_i(t)$ and $\mu_i(t)$ with $t = 0$
3. Iterate through all possible threshold values t , updating ω_i , μ_i and computing values for $\sigma_b^2(t)$
4. Desired threshold corresponds to the maximum $\sigma_b^2(t)$ calculated

Once the Otsu value $t = T$ is found, the input image can be thresholded in order to produce the desired binary (black & white) image. Given a greyscale image $G(x, y)$ where sets of x and y correspond to individual pixels on the 2D plane, the thresholding operation can be defined as follows:

$$B(x, y) = \begin{cases} 1 & \text{if } G(x, y) \geq T \\ 0 & \text{otherwise} \end{cases} \quad (2-4)$$

Where $B(x, y)$ is the resulting binary image.

2.2.2 Region Detection

This step involves repeatedly blurring the input image using a square averaging filter and then determining the number of regions in the area using connected component labelling. The image is blurred and thresholded continuously until only one region can be found in the image.

The averaging filter works by treating the image as a 2D matrix of integer values $I(x, y)$ that represent the pixel intensities, the pixel intensity of each pixel in the image is then replaced with the average intensity of a square group of pixels of which the target pixel is the centre:

$$\text{for square size } S^2: A(x, y) = \frac{\sum_{n=0}^S \sum_{m=0}^S I(x - \frac{S}{2} + n, y - \frac{S}{2} + m)}{S^2} \quad (2-5)$$

Connected component labelling (CCL) is an algorithmic application of graph theory where sets of connected components are identified and labelled based on certain rules. In image processing, connected component labelling is used to identify regions of a binary image that are separate from one another. In the case of FLORA's implementation a region is defined as a group of non-zero pixels (white) that are surrounded by zero pixels (black).

In Octave, connected component labelling is achieved using the `bwlabel` function which is derived from the algorithm outlined by Haralick et al [10]. Octave also defines pixel groups using "8-connectivity", meaning any pixel touching one of the four edges is considered part of its group and any pixel touching one of the four vertices is also considered part of its group.

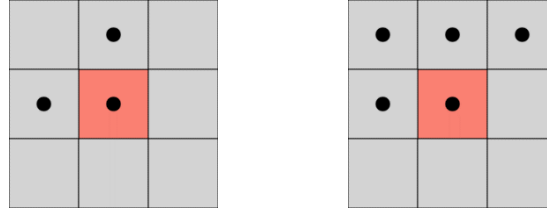


Figure 2-5 Example of 4-connectivity (left) and 8-connectivity (right)

Figure 2-5 shows an example of 4-connectivity and 8-connectivity. In each image the gray squares with black dots are considered neighbors of the red square with the black dot.

2.2.3 Rotate

This step involves rotating an image so that its major axis lies flat with the x-axis. The major axis in the case of FLORA refers to the major axis of the ellipse that has the same normalized second central moments as the region produced after the 'Clear background noise' stage. This is required for extracting features that are not rotation invariant, specifically Rectangularity (refer to section 2.2.4i). Octave does not specify the exact algorithm used to find the angle of rotation, but a commonly used method involves the calculation of the image moments as follows [11]:

First if we consider an input image as a discrete function $I(x, y)$ where sets of x and y refer to single pixel coordinates, the spatial image moment M_{pq} can be defined as:

$$M_{pq} = \sum_x \sum_y x^p y^q I(x, y) \quad (2-6)$$

The 'order' of the image moment is determined by the values of p and q :

$$\text{moment order} = p + q$$

Therefore the 0th order moment is M_{00} , the 1st order moments are M_{01} and M_{10} , the 2nd order moments are M_{02} , M_{20} and M_{11} etc. We also define the central image moments μ_{pq} as follows:

$$\mu_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x, y) \quad (2-7)$$

Where $\bar{x} = \frac{M_{10}}{M_{00}}$ and $\bar{y} = \frac{M_{01}}{M_{00}}$

Using the central moments a covariance matrix can be derived as follows:

$$\text{cov}[I(x, y)] = \begin{bmatrix} \mu'_{20} & \mu'_{11} \\ \mu'_{11} & \mu'_{02} \end{bmatrix} \quad (2-8)$$

Where $\mu'_{ij} = \frac{\mu_{ij}}{\mu_{00}}$.

The eigenvectors of this matrix form the major and minor axes of the image region, the angle of orientation θ can be calculated from the covariance matrix as follows:

$$\theta = \frac{1}{2} \arctan \left(\frac{2\mu'_{11}}{\mu'_{20} - \mu'_{02}} \right) \quad (2-9)$$

This angle θ can then be used to rotate the image as needed. Octave does not specify the exact algorithm used to achieve image rotation, however it does specify that nearest neighbor interpolation is used to ensure there is no loss of image quality. Usually image rotation can be achieved using a rotation matrix and a matrix transform operation. The general form of the rotation matrix R that rotates a matrix θ degrees is as follows:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad (2-10)$$

A point (x, y) can then be rotated to (x', y') by applying the following matrix transform:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = R(\theta) \begin{bmatrix} x \\ y \end{bmatrix}$$

2.2.4 Extract Features

Feature extraction is a process used in machine learning where special mathematical values are derived from a larger data set such that the values derived are in some way informative and can be used to describe the original data set. [12] In the case of FLORA the data sets used are images of leafs and the extracted features are intended to be characteristic of the specific plant species. Before feature extraction can take place it is necessary to determine what features can effectively be extracted from the image and it must be determined what set of features will produce the most accurate results. Identification of features and the following feature set optimisation was completed for FLORA by Katz in [6]. Katz made use of several morphological features for FLORAs feature descriptor set. The main requirement for the features chosen were that they had to be scale invariant as different images taken of the same leaf species will almost definitely result in images with different size leaf regions. The chosen features were as follows:

i. Rectangularity

Rectangularity is a measure of how similar the extracted region is to the smallest bounding rectangle that fits the region[13]. It is calculated as:

$$R = \frac{A_r}{L_p W_p}$$

Where L_p is the maximum length of the region on its major axis, W_p is the maximum width of the region perpendicular to its major axis and A_r is the region area. Calculation of the rectangularity in FLORA requires 2 operations, the first is calculation of region area and the second is calculating the bounding rectangle.

These operations are achieved using the Octave `regionprops` function. Octave does not specify the exact algorithms used to determine these values, but they can be calculated easily by doing a full traversal of the image produced after the rotation stage. Bounding rectangle can be calculated by determining the top-most, left-most, bottom-most and right-most pixels in the region. Leaf area can be calculated by simply counting the number of pixels in the region. It can also be calculated through application of Green's Theorem if the leaf perimeter is known [14].

ii. Solidity

Solidity or 'area ratio of convex hull' is the ratio of the region area to the area of the convex hull that fits around the region[13]. The convex hull is defined as the smallest convex set of points that contains the region[15]. Solidity is calculated as follows:

$$S = \frac{A_r}{A_{hull}}$$

Where A_r is the region area and A_{hull} is the area of the convex hull.

FLORA uses a convex hull algorithm written by Mark Hayworth [16]. There are however several other well-known convex hull algorithms that exist such as Gift Wrapping, Graham Scan [17], Quick Hull [18] and more. Once the convex hull is determined the area can be calculated, the original FLORA system achieves this using the Octave `bwarea` function. This function simply calculates a weighted sum of the pixels in the region. Weights are determined by the number of non-zero pixel neighbours each pixel has [19]. The area of a convex hull can also be calculated by using a line integral determined using Greens Theorem [14].

iii. **Convex Perimeter Ratio**

Convex perimeter ratio is similar to Solidity except it is the ratio of the region perimeter to the perimeter of the convex hull [20]. It is calculated as follows:

$$PC = \frac{P_r}{P_{hull}}$$

Where P_r is the perimeter of the region and P_{hull} is the perimeter of the convex hull. Perimeter of the leaf region and convex hull can be calculated using a border following algorithm such as the Square Tracing algorithm or the Moore-Neighbour Tracing algorithm. [21]

iv. **Form Factor**

Form factor is a ratio used to describe how much the region differs from a circle [13]. It is calculated as follows:

$$FF = \frac{4\pi A_r}{P_r^2}$$

Where A_r is the area of the leaf region and P_r is the perimeter of the region.

2.2.5 **KNN Prediction**

The final processing step that is required is for prediction is for the input image to be classified as a specific species using the features extracted. Most classification techniques work on the assumption that if an input sample has similar features to a data sample (or set of samples) in the known sample space then that input sample can be classified in the same class as the known data sample.

Several different classification techniques were investigated by Katz [6] and tested for applicability to FLORA, he concluded that the K Nearest Neighbour (KNN) algorithm provided the best accuracy. The algorithm used by Katz was described by Sinha in [22].

The KNN algorithm works by creating an N dimensional feature space in which each of the N dimensions represents a particular feature. In the case of FLORA, 4 different features are used and thus the feature space has 4 dimensions. In order to classify an input sample using the KNN algorithm in FLORA the following steps are followed:

1. First the classifier must be trained, this involves creating a database of points in 4 dimensional space, where each point represents a particular plant species. The 'coordinates' of each point are determined by the numerical values of extracted features and these features must be extracted from images of leafs of which the species is known. Each of the 4 dimensions represents one of the features described in section 2.2.4.
2. Next the input sample must be processed by feature extraction and its own point in 4 dimensional space is produced.

3. Lastly the Euclidean distance between the calculated input point and the other existing points in the database must be calculated. The point in the database that has the smallest Euclidean distance to the input point is then used to classify the input point, in other words it is predicted that the input leaf image is of the same species as the species represented by its nearest neighbour in the feature space. For a point p and q in N dimensional space Euclidean distance is calculated as follows:

$$d(p, q) = \sqrt{\sum_{i=1}^N (p_i - q_i)^2}$$

Where p_i and q_i represent the i^{th} dimensional component of points p and q respectively.

The classification algorithm described above is in fact a specific implementation of K Nearest Neighbour known as 1st Nearest Neighbour (K=1) or just Nearest Neighbour. Different variations of the KNN algorithm exist where one may use several nearest neighbours (K>1) with different weightings or even different distance metrics in order to improve accuracy [22]. Katz experimented with different values for K and concluded that the 1st Nearest Neighbour was optimal in terms of prediction accuracy [6].

2.3 General-purpose Computing on Graphics Processing Unit (GPGPU)

GPGPU refers to the use of graphics processing units (GPUs), which are typically used for processing of computer graphics, to perform computations that are normally reserved for processing on the CPU [23]. The advantages of using a GPU for general purpose computations are that it allows for much greater throughput, while maintaining similar levels of precision and power consumption[1].

Hardware manufacturers such as the Nvidia Corporation also claim that that GPU's are improving at a much higher rate compared to CPU's due to their specialised nature [1]. GPGPU computing does however have several disadvantages, these include a high level of programing difficulty and an inherent unsuitability to computations that are not highly parallelisable.

An analysis done by Vuduc et al [24] shows that in some cases a well optimised CPU implantation of a particular problem may be more effective (in terms of performance or development time) compared to a GPU implementation.

GPGPU implementations have generally been well suited to matrix operations and image processing [1], therefore since the FLORA back-end consists mainly of these types of computations it is likely that FLORA could greatly benefit from GPU implementation of some of its back-end computations.

2.3.1 The GPU Programming Model

Available consumer GPU's range from large high performance high throughput GPU's to small low power mobile GPUs. Although the different types of GPUs may differ considerably between manufacturers and between different models produced by a particular manufacturer, the majority of modern GPU's share a similar hardware architecture and programming model. Owens et al gives a good description of the evolution of GPU hardware architectures in [23], however with focus being on utilisation of the GPU rather than on the actual GPU hardware, only the GPU programming model will be reviewed further.

The computational units of a GPU use the single-program multiple-data (SPMD) programming model [23]. To achieve a high throughput and efficiency the GPU does many computations in parallel using the same program, in the base model it is assumed that computations occur independently of one another and there can be no communication between different instances of the program until completion, data can only be read from and written to a shared global memory. This method of processing is well suited to the single instruction, multiple data programming model (SIMD), but to give developers more flexibility, modern GPU's allow different execution paths for instances of the same program leading to

the SPMD model. There is however a performance penalty that is incurred if the program code is not structured to have coherent branching. [23]

Since GPU's are designed primarily for graphics processing only, this means that general-purpose computations must be mapped onto the GPU such that they can be processed in the same way as a graphical computation. Historically this made programming difficult because programmers have to use graphics API's in order to perform non-graphical computations. However with the advent of GPGPU platforms described in section 2.4 programmers are now able to write GPU code in a more natural and non-graphical oriented way. Owens et al[23] defines the structure of modern general purpose GPU programs in the following way:

1. The developer directly defines the computation domain as a structured array of threads
2. A single SPMD program is executed on each thread in parallel
3. A return value is computed for each thread using a combination of mathematical operations and read and write accesses to global memory.
4. The resulting values in global memory can then be used as an input for further computations.

As a result of the adoption of this programming model, developers are able to take advantage of the data parallelism of their programs by explicitly defining that parallelism in GPU code.

2.4 Past FLORA Optimisation Attempts

Attempts have been made in the past to improve the FLORA system. J. Esselar in [25] first attempted to improve the FLORA system performance by creating a hybrid C and Octave implementation, he also tried to improve prediction accuracy using an 'Extreme Learning Machine' classifier in place of the KNN classifier. Following Esselar, M. Ramone in [26] also attempted to improve the FLORA system by implementing and testing a number of different algorithms for greyscaling, binerisation and region detection used in the Convert to Black & White stage (Section 2.2.1) he also implemented the 'Elliptic Fourier Analysis' classifier to try and improve prediction accuracy compared to the original KNN classifier.

Both Esselar and Ramone failed in their attempts to improve FLORA. This serves to reinforce how effective the original FLORA implementation by Katz is. In terms of prediction accuracy it can be seen from Esselar and Ramone's results that the feature set and KNN classifier chosen by Katz is also very effective compared to other types of classifiers. Esselar does however point out that speed performance optimisation is hindered by FLORA's reliance on Octave to carry out its image processing due to Octave's limited optimisation capabilities.

2.5 Review of Current GPGPU Platforms

2.5.1 Open Computing Language (OpenCL)

OpenCL is a framework for writing programs that can execute across multiple types of processing units including Central Processing Units (CPUs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs) and Field Programmable Gate Arrays (FPGAs), these are collectively called 'compute devices'[27]. Platforms that make use of more than one type of processing unit are generally referred to as 'heterogeneous computing platforms'[27], GPGPU platforms can be considered as a specific type of heterogeneous computing platform in which only CPUs and GPUs are used.

OpenCL views a hardware platform as a number of compute devices that are controlled by a host processor (Typically a CPU). OpenCL provides a C-like programming language in which program

modules called 'kernels' can be written, these kernels can then be run on any number of compute devices in parallel. [28]

OpenCL also provides an API that can be used by other programs in order to launch kernels of their own to help improve program performance. The standard API has been developed for C and C++, but 3rd party API's also exist for other languages such as Python and Java. Many 3rd party developers, particularly hardware developers, include support for the OpenCL standard in their own acceleration platforms and software.

2.5.2 Compute Unified Device Architecture (CUDA)

CUDA is a proprietary parallel computing architecture and application program interface (API) created by NVidia. It allows programmers to access low level capabilities of CUDA enabled graphics processing units in order to carry out general purpose computations. [29] Programmers can have access to the CUDA platform through specially written libraries, compiler directives or through extensions of existing programming languages such as C++ and Fortran. CUDA also has support for the OpenCL standard [29]. CUDA is currently one of the most widely used and mature GPGPU platforms used today since it is also one of the oldest.

2.5.3 AMD Accelerated Parallel Processing (AMD APP)

The AMD APP technology is a GPGPU platform designed to allow GPU acceleration of general purpose computations using AMD GPU's. Like NVidia's CUDA it provides programmers access to low level GPU functionality and provides a high level API that provides support for industry standard acceleration platforms like OpenCL and C++ AMP [30]. AMD APP is one of the newer GPGPU platforms available today and as such the available code base is currently quite small.

2.5.4 Open Source Computer Vision (OpenCV)

OpenCV is a library of real-time computer vision functions, the library is cross-platform and open source. It is currently maintained by the Russian company Itseez [31]. Among other computer vision functions, OpenCV includes functions for 2D and 3D image processing, segmentation, recognition and machine learning. Later versions of OpenCV also have limited support for GPU acceleration with select functions having OpenCL and/or CUDA accelerated implementations. [32]

The most current OpenCV libraries (version 2 and later) are written in C++, but full interfaces exist for compatibility with Python and Java. Programs written using OpenCV can also be compiled into Matlab or Octave executables for use in Matlab or Octave. [31]

2.6 Review of existing GPU Implementations of Processing Algorithms

This section serves as a review of current GPU accelerated implementations of image processing and machine learning algorithms used by the FLORA back end. The processing algorithms used by FLORA are detailed in section 2.2. The GPU programming model has been reviewed in section 2.3.1, therefore this section will only focus on GPU based algorithms and pseudocode that could be applied as programs or 'kernels' in any of the GPGPU platforms (CUDA, OpenCL, AMD APP etc).

2.6.1 Convert to Black & White

i. Greyscaling

As described in section 2.2.1 the first stage of conversion to black and white is greyscaling. Greyscaling is achieved through a simple weighted sum operation shown in equation (2-1). The greyscaling of each pixel in an image can occur completely independently of one another as there is no dependency between pixels. Therefore the greyscaling operation is well suited for implementation as a GPU kernel as is. OpenCV also provides a GPU accelerated function for greyscaling an image [32]

ii. **Otsu Thresholding**

The second stage of the conversion to black and white is Otsu thresholding, Singh et al proposed a GPU implantation of Otsu thresholding in [8]. Their implementation uses a mixture of CPU and GPU code, calculating the grey level histogram on the GPU, then processing the histogram on the CPU and finally thresholding the image using the GPU.

Singh et al managed to achieve an average speed up of 1.6x over their fully CPU implementation. They also showed that the speedup can increase significantly with higher resolution images.

2.6.2 **Region Detection**

i. **Averaging Filter**

Region detection in the FLORA back end is a two stage process described in section 2.2.2, the first stage is the application of a square averaging filter or 'box' filter. In the context of GPU computing, multi-pixel to pixel transforms such as the box filter fall under the category of image 'convolution' where the pattern of pixels used (box) and those pixels' relative contribution (equal) to the final pixel in the output image is called the 'filter kernel' [33] R Fernando provides an example of a GPU box filter program in [33],

The GPU accelerated box filter works by loading the original image into GPU memory and then splitting the input image into multiple parts. The GPU then computes the averaging filter on each part in parallel and stores the results into a new image. Once the new image has been constructed it is returned to the host.

ii. **Connected Component Labelling**

After applying the box filter the second stage is to determine the number of components in the image using connected component labelling. GPU implementations of CCL have been proposed by O. Štava in [34] and V. Oliveira in [35]. In both cases the GPU implementations were based on the 'Union-Find CCL' algorithm [36], parallelisation is achieved by splitting the input image into multiple parts (Figure 2-5a) and implementing the Union-Find algorithm on each part in parallel to produce multiple 'local' solutions. The local solutions are then recursively merged in order to produce the final global solution (Figure 2-5b) [34].

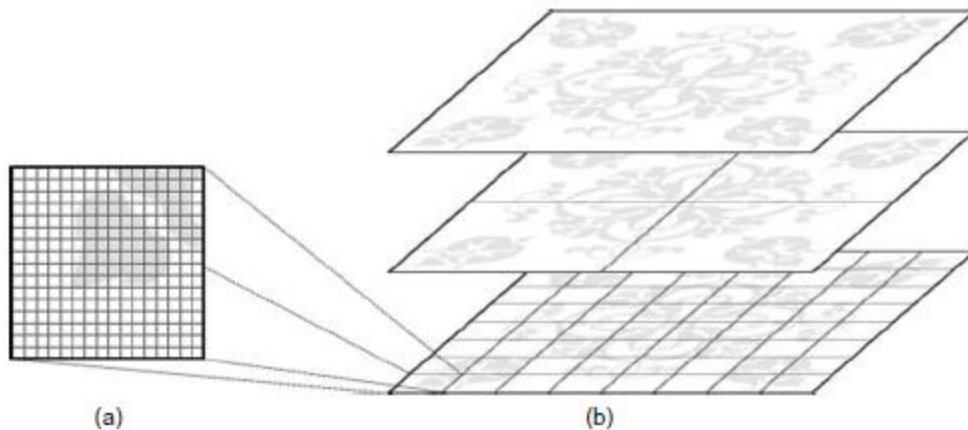


Figure 2-6 Parallelisation of Connected Component Labelling [34]

O. Štava was able to achieve an average speed up of about 10x and V.Oliveira's implementation was able to achieve an average speed up of about 5x compared to their respective CPU implementations.

2.6.3 Rotate

i. Image Moments

The first step in the rotation process requires the calculation of various image moments as outlined in section 2.2.3. J Fung proposes a simple technique for calculating image moments in [37]. Fung proposes given a moment calculation as follows:

$$M_{pq} = \sum_x \sum_y x^p y^q I(x, y)$$

This can be applied to the GPU by first building a 2D matrix of elements in a GPU buffer such that each element is equal to:

$$x^p y^q I(x, y)$$

Where x and y are the element coordinates in the matrix. This matrix can then be summed to a single value using a basic GPU summation function. A similar process can be followed to calculate the central moments μ_{pq} . A parallelised implementation of 2D image moments has also been proposed by K. Chen in [11].

ii. Rotational Matrix Transform

After the rotation angle has been found rotation can be accomplished using a rotation matrix and a matrix multiplication. R. Hochberg proposes a GPU implementation for matrix multiplication in [38], his approach is given a matrix A and B , the matrix $C = A \times B$ can be calculated element by element in by doing each row-column operation in parallel:

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 12 & 12 & 12 \\ 15 & 15 & 15 \\ 18 & 18 & 15 \end{pmatrix}$$

Figure 2-7 Example of parallel matrix multiplication

Figure 2-7 Shows how matrix multiplication can be done in parallel by extracting row-column pairs from the two source matrices and calculating each output element separately. Hochberg also gives modified implementations for handling non-square matrices and details an optimised implementation that makes better use of a GPU's shared memory by extracting multiple row-column pairs at a time, thus reducing memory accesses and improving performance.

Another similar implementation is proposed by Chrzyszczuk et al in [39], their implementation makes use of 'cuBLAS' a CUDA implementation of the Basic Linear Algebra Subprograms (BLAS) specification. BLAS is a standard specification of low level linear algebra functions that are widely used in industry today[40]. BLAS implementations are intended to take advantage of low level hardware capabilities to improve performance.

2.6.4 Extract Features

The feature extraction process is detailed in section 2.2.4. Feature extraction in FLORA requires two main processing steps: calculation of the up-right bounding rectangle and calculation of the extracted region's convex hull. These two calculations can then be used to calculate all the features described in section 2.2.4 using simple mathematical operations on the CPU.

i. Bounding Rectangle

As stated in section 2.2.4, the up-right bounding rectangle can be calculated easily from a simple traversal of the binary image produced after the Rotation stage. A GPU implementation of this stage can

be achieved simply by splitting the input image into several blocks and computing the region bounds on each block, then recursively merging the results to find the full region bounds.

ii. **Convex Hull**

A GPU implementation for finding the convex hull in a point set is proposed by Srungarapu et al in [41] and Tzeng et al in [5]. Their implementations are based on the ‘quickhull’ method proposed in [18]. The quickhull method works as follows:

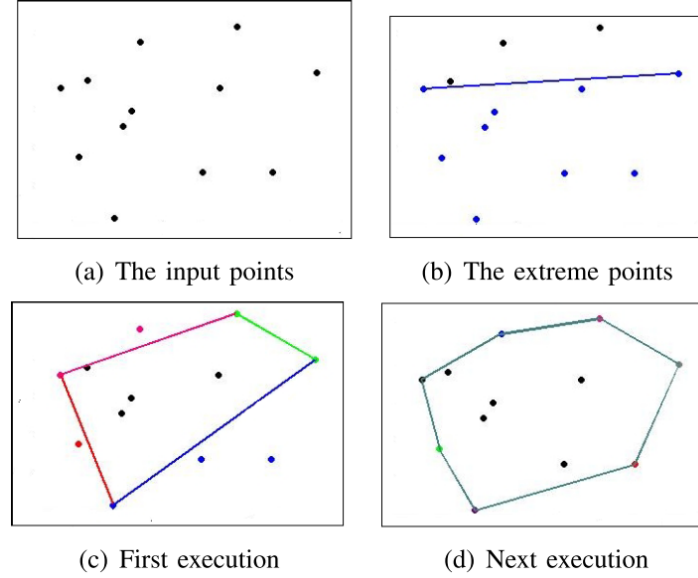


Figure 2-8 Example of Quickhull Algorithm

Starting with a set of points on a 2D plane (Figure 2-8a), the quickhull algorithm first finds the left-most and right-most points (Figure 2-8b). A line is ‘drawn’ between these two points splitting the point set into two separate point sets or segments. The point that is furthest away from this line perpendicularly is then found and used to construct a polygon (Figure 2-8c), then for each side of this polygon the point furthest away from each side perpendicularly (on the outside) is found if possible. If any points are found the polygon is modified to include the found points and the process is repeated with the new polygon. Once no more points can be found the resulting polygon is the convex hull (Figure 2-8d).

In order to implement this algorithm on the GPU it must be parallelised. Srungarapu achieved this by calculating the perpendicular distances of all outer points and extending the sides of the polygon all in parallel. Tzeng et al’s implantation differs in that they have implemented a ‘permute’ operation which permutes the points in the image and generate segments, points are then removed from these segments using a ‘compact’ operation until the final convex hull is produced. Tzeng’s implementation has been tailor made to suit the GPU architecture while Srungarapu’s implementation focuses purely on parallelisation in general. Srungarapu’s implementation provided a good speed up for images with 100k points or more with an average speedup of 13 to 14 times. For images below 100k points the execution time for both the CPU and GPU implementations were negligible. Tzeng et al managed to attain a speed up of one order of magnitude compared to their CPU implementation, however this was also only achieved with images that contained 100k or more points. Both Tzeng et al and Srugarapu et al used the ‘qhull’ implementation [42] for their CPU version.

2.6.5 **KNN Prediction**

The KNN prediction algorithm used by FLORA is detailed in section 2.2.5. KNN prediction in FLORA requires the training of a classifier, then prediction is done by extracting features from an input image and using the classifier to find the nearest neighbour[6]. Training of the classifier need only be carried

out once and the classifier can be updated if new plant species are to be added. GPU implementation of feature extraction is dealt with in section 2.5.4. This section will therefore only deal with GPU implementations of the actual KNN search done after feature extraction.

The KNN search in FLORA involves calculating Euclidean distances between points in sample space and ordering these distances so that the shortest distance (nearest neighbour) can be found. Several GPU implementations of KNN have been developed in the past [43]–[47].

Liang et al's implementation of KNN on the GPU [44] was achieved by implementing the distance calculation function and sorting function as two separate kernels. The classifier database of points in sample space (classes) is first split up into blocks and these blocks are operated on by the distance calculation kernel in parallel, this kernel calculates all the distances between the input point generated and the existing points allocated in a particular block. The results are stored in the blocks' shared memory. The sorting kernel then sorts the calculated distances so that the nearest neighbour/s can be found. Due to the limited size of GPU blocks however it is likely that multiple blocks will be used and each will produce a nearest neighbour value. Liang et al refers to these values as 'local-k nearest neighbours'. The sorting kernel spawns a thread to calculate the final k-nearest neighbour/s or 'global-k nearest neighbours' once all the local neighbours of each block are found. The final step in the procedure is to use the k-nearest neighbours to determine the class of the input through a voting mechanism, Liang et al implements this on the CPU since k is usually small and therefore this step will have negligible impact on performance.

Liang managed to achieve a speed up of about 5 times with a database of about 65k points [44], this speed-up was linearly proportional to the database size approaching 0 as the database size decreased. Garcia et al's implementation [46] followed a similar approach to Liang et al's but used a distance matrix to store the calculated distances and took advantage of cuBLAS optimised matrix operations to sort the values. They achieved a speed-up of about 55 times for a database of 65k points, the speedup was also proportional to the database size.

3. Methodology

3.1 Outline

This project development cycle has been split into multiple phases in order to ensure the project is well structured and all objectives are met satisfactorily. The first phase (Phase 0) involved defining the project in terms of the project hypothesis, objectives and scope. Once the project was defined research was done on various topics including a review of the existing FLORA system, computer vision algorithms and GPGPU programming. Based on this research design preparation took place, this included selection and setting up of development environment and tools. The design and implementation phases could then follow on from that point. The final stage of the project involved validation and performance analysis of the new FLORA implementation/s.

3.2 Phase 1: Research and Literature Review

The first phase of development involved doing research and compiling a literature review related to GPU computing platforms and computer vision algorithms. A review of the current FLORA implementation was also necessary in order to identify which computer vision algorithms must be reviewed.

The purpose of this phase is to get an in depth understanding of how the FLORA system works, and to review current work done in the field of GPU optimisation in order to see what kind of optimisations can be applied to FLORA and what kind of speed ups can be achieved. Current GPGPU platforms and programming tools were identified and investigated so that the most optimal solution could be chosen for phase 2.

3.3 Phase 2: Selection & Set Up of Development Tools

3.3.1 *Host Computer*

This phase involved selection of the development environment and tools based on the research done and what hardware/tools are available for use.

This project requires a host computer that can act as the backend server for the FLORA system, the computer used has the following specifications:

- CPU: 4.0GHz Quad core Intel Core i7-4790k
- GPU: 1.0GHz NVidia GTX 780ti
 - 2880 CUDA cores
 - CUDA Compute Capability 3.5
- RAM: 16GB at 2133MHz
- Storage: 250GB OCZ Vector SSD
- OS: Microsoft Windows 10 Professional

3.3.2 *Chosen GPGPU platform and programming environment*

After taking the research done into consideration it was decided that the CUDA platform would be the optimal choice for FLORA's GPU implementation. CUDA was chosen mainly because the host computer's GPU is designed for use with CUDA. CUDA is also compatible with the OpenCL specification and many of OpenCV's libraries have CUDA implementations that can be used easily. This allows for maximum flexibility when coding. Of all the GPGPU platforms CUDA also has the most mature code base and plenty of resources are available online and in books.

The chosen programming language is C++, C++ was chosen because it is backwards compatible with C which is the language used by CUDA and because OpenCV's libraries are written in C++. C++ has also been designed with flexibility and performance at its core [48] allowing for several code optimisations such as multithreading that are not available in Octave. Another advantage of C++ is that it is cross platform and can run natively on the host operating system without the need for an additional runtime environment like Octave. Table 3-1 shows the additional software tools used for development:

Table 3-1 Additional Software Tools

Tool name and version	Reason for Selection
GNU Octave 4.0.0	Runtime environment needed for original FLORA system
XAMPP Server 3.2.1	Server runtime environment, needed to run the FLORA web page interface. The AMPP server is cross platform and open source
Microsoft Visual Studio 2013	Free IDE for C++ development. OpenCV and CUDA use Microsoft's Visual C and C++ compilers that come with Visual Studio. Also includes tools for code profiling and debugging
CUDA Development Toolkit v7.5	Contains all the tools needed for writing programs with CUDA
OpenCV 3.0.0	Contains several highly optimised computer vision and image processing libraries written in C++. Also includes some CUDA accelerated libraries
NVidia Graphics Driver 355.98	Drivers for the GPU

3.4 Phase 3: Initial FLORA Analysis and Test Data Acquisition

Following the selection and setup of the development tools, the original FLORA system was then implemented on the new host computer in order to determine its performance characteristics. Both the prediction accuracy and the execution time of each of FLORA's processing steps (Explained in section 2.1.2) are determined on the new host machine. This ensures that a fair comparison can be made between the new FLORA implementation and the original.

Additionally, a database of test images is needed for testing the speed and accuracy of the FLORA implementations. It was decided that the image dataset 3 compiled by Katz [6] would be used. Figure 3-1 shows a sample of the images used in the testing database.



Figure 3-1 Sample images from chosen testing database

This database contains 130 images of Fynbos leaves taken from 13 different species (10 of each species). This data set was chosen because it was the dataset used to test the original FLORA implementation by Katz and because these images are of the same type that the FLORA system would expect when being used in the field. The images in this database are high resolution (around 3456 X 2304 pixels) however Katz explains in [6] that images with resolutions around 691 x 461 produce the best performance with minimal impact on prediction accuracy. Therefore all images in this database were resized to a smaller resolution before being used for testing.

3.5 Phase 4: Optimised CPU Implementation

Since the new FLORA back end system algorithms will be implemented in C and C++, it was deemed necessary to create a full C++ implementation of the FLORA system that can serve as the foundation for the subsequent GPU acceleration. It was also decided that this C++ implementation would be designed to be as highly optimised as possible. From the research done it was found that in some cases CPU optimisation of certain algorithms can outperform GPU implementations [24]. In order to achieve the best possible performance of the new FLORA implementation it is therefore necessary to compare both CPU optimised and GPU optimised versions of the FLORA processing algorithms and use only the best performing solutions.

The CPU optimised version of the new FLORA system was implemented using OpenCV libraries, these libraries were chosen because they are known to perform well and already make use of various CPU optimisations such as Streaming SIMD Extensions (SSE2) and Advanced Vector Extensions (AVX) [49]. The design of the C++ implementation of FLORA followed the same approach as shown in Figure 2-3. The design was kept as similar to the original Octave implementation as possible in order to maintain

the prediction accuracy. Additionally the C++ implementation was made using a modular approach, this not only makes the code easier to understand and debug but it made it easy to 'swap out' standard C++ algorithms for GPU accelerated ones.

3.6 Phase 5: GPU Design Iteration

This phase of the development cycle involved identifying bottlenecks in the C++ implementation, then designing and implementing GPU accelerated algorithms in order to remove those bottlenecks. Figure 3-2 below shows how the design processes was carried out.

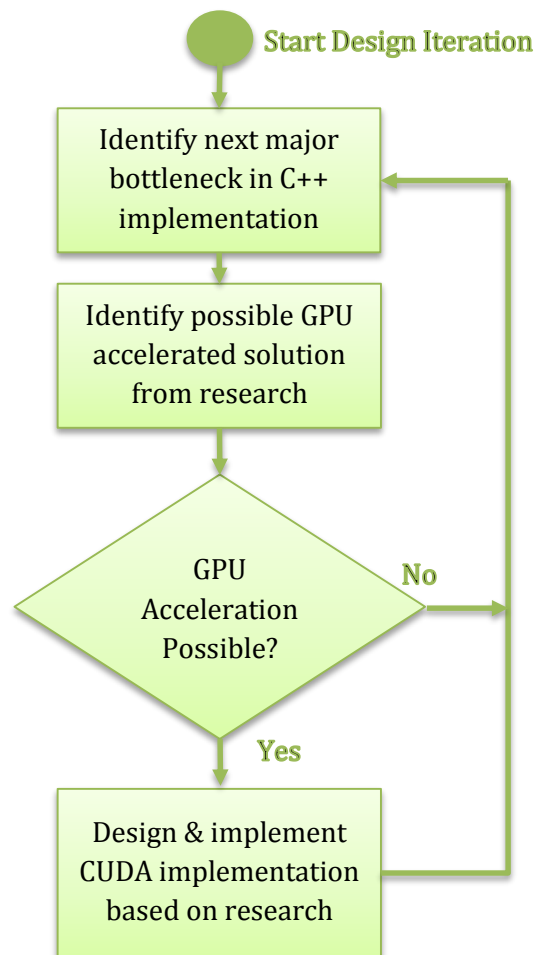


Figure 3-2 GPU Implementation Design Processes

The design approach used for the GPU implementation illustrated in Figure 3-2 is ideal because it ensures that the largest bottlenecks in the FLORA system are dealt with first. This is important because it is unfeasible and impractical to simply implement every processing algorithm used by FLORA in CUDA due to time constraints. This approach also helps to avoid trying to implement algorithms on the GPU that already run effectively without GPU optimisation, implementing these kinds of algorithms on the GPU can actually cause a reduction in performance due to the overheads involved with GPU implementations.

Due to the nature of computer programming a bottleneck will always exist in any program, therefore as can be seen in Figure 3-2 this phase did not have a defined end. This phase was iterated as many times as possible in order to improve the performance as much as possible until no further gains in performance were observed or until iteration had to be ceased due to time constraints.

3.7 Stage 6: Performance Analysis

The final stage of the development process involved performance benchmarking of the standard C++ and GPU accelerated implementations of the FLORA backend system. The execution time of each of FLORA's processing algorithms were determined for both implementations and compared with the original Octave implementation. The execution times for the C++ and CUDA accelerated implementations were determined using the Microsoft Visual Studio 2013 profiler. The prediction accuracy was also checked and compared for each implementation.

3.7.1 Prediction Accuracy Calculation Method

In order to properly test the accuracy of each FLORA implementation and variation, the database of input images was split into two separate image databases: A 'Training' database and a 'Testing' database. The KNN classifier was trained using the Training database and used to predict the species of all the leaf images in the Testing database. The accuracy could be calculated using the following formula:

$$\text{Prediction Accuracy} = \frac{\text{Total Number of Images in Testing database}}{\text{Total number of correct predictions}} \times 100 \quad (3-1)$$

The prediction accuracy was also confirmed by swapping the Testing and Training databases and recalculating the prediction accuracy.

3.8 Stage 7: Implementation Optimisation

3.8.1 Execution Time & Prediction Accuracy Optimisation

Since multiple different CUDA accelerated functions were to be developed it was therefore necessary to test different combinations of standard C++ and GPU accelerated functions in order to determine which set of functions perform the best in terms of execution time and prediction accuracy.

3.8.2 Image Resolution Optimisation

Once the optimal set of algorithms was determined for the final FLORA back end implementation, an additional test was carried out by varying the image resolution of the input image. This test was used to determine how image resolution affects the execution time and prediction accuracy of the new FLORA implementation. It also served to determine whether a different set of C++ and/or CUDA accelerated functions was optimal for images of higher resolution. An optimal image resolution was then chosen based on the results of this experiment.

3.9 Final Integration with FLORA Front End

Once experimentation and optimisation was completed the final FLORA implementation was then ready for deployment. The final step was to integrate the new FLORA back end implementation with the original FLORA front end and to test the web app. Once the web app was proved to be working with the new FLORA back end, the design and implementation phases were concluded and conclusions could then be drawn.

4. FLORA C++ Design

4.1 Overview

The C++ implementation of the FLORA back end followed a similar design as shown in Figure 2-2. In order to reduce the development time, ensure good performance and acceptable accuracy of the C++ implementation all processing algorithms were implemented using the OpenCV libraries. The code was also kept as modular as possible to ensure the code is easy to understand and modify. Figure 4-1 shows an overview of the processing steps used in the C++ implementation.

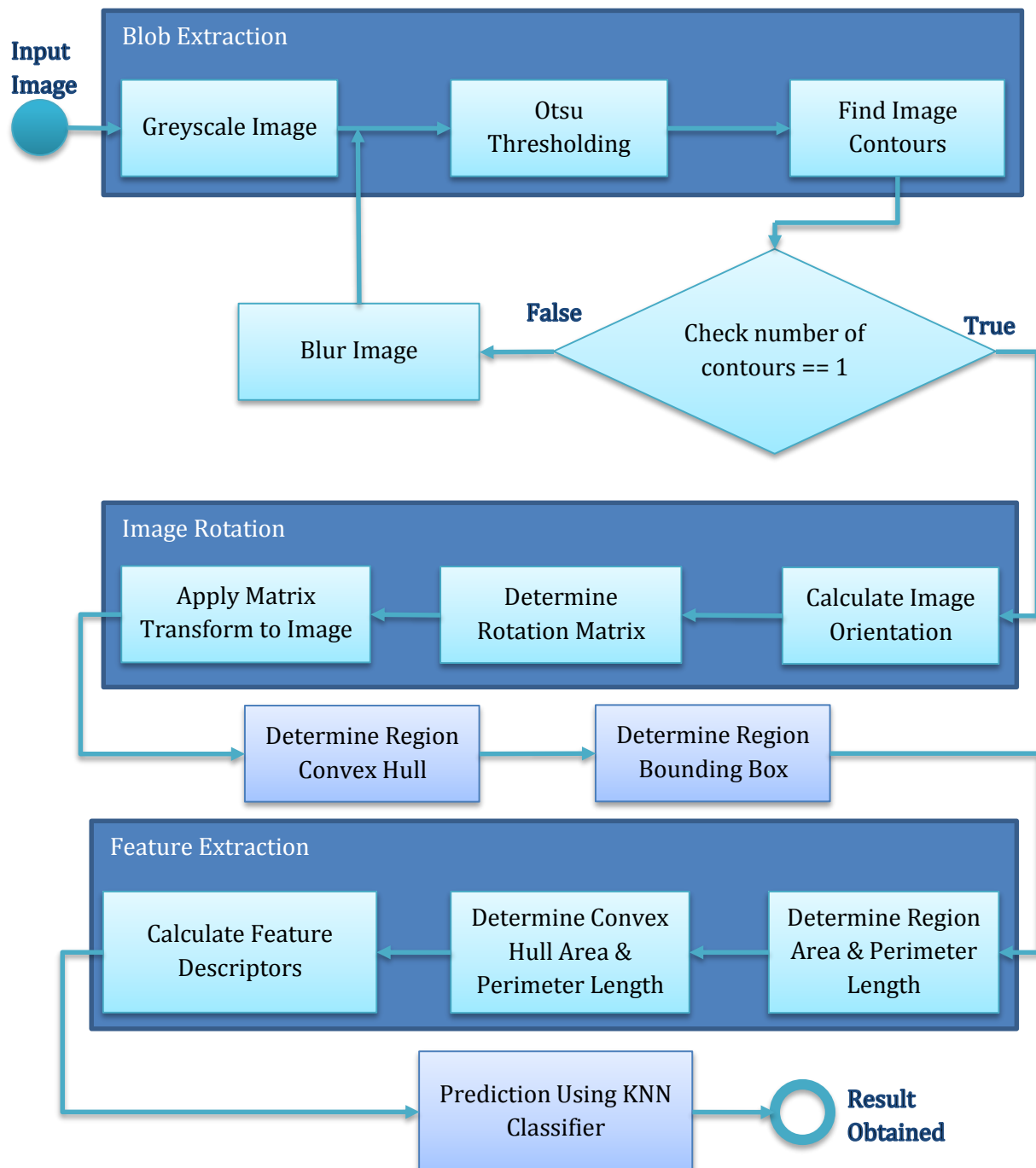


Figure 4-1 Overview of FLORA C++ implementation Processing Steps

As can be seen from Figure 4-1 certain groups of processing steps have been grouped together under specific function names. These correspond to how the function modules are defined in the C++ implementation. The 'Determine Region Convex Hull', 'Determine Region Bounding Box' and 'Prediction Using KNN Classifier' processes are also implemented as separate modules.

4.2 A Note on OpenCV Image Processing Library Initialisation Overhead

It was discovered during implementation of the C++ implementation that when using OpenCV image processing and computer vision libraries, there is an initialisation operation that occurs upon the first function call. This initialisation took an average of 250ms to complete and always occurred regardless of which OpenCV image processing function was called first.

In order to ensure this initialisation operation did not affect the actual function execution timings of the main program, a small 'dummy' function was created that is called at the beginning of the program. This function is shown in Listing 4-1.

Listing 4-1: Dummy Operation Called to Initialise OpenCV Image Processing Library

```
void Dummy(){  
  
    Mat temp = Mat::zeros(8, 8, CV_8UC1);  
    cv::blur(temp, temp, Size(2, 2));  
}
```

The dummy function applies a simple averaging filter to a small 8 X 8 array of arbitrary values. This function would normally have negligible execution time and therefore any significant delay that occurs when this function is called can be counted as the OpenCV library initialisation delay.

4.3 Image Input

Like the original FLORA system created by Katz [6] the C++ implementation expects input images to contain only a single fynbos leave on a white background. This ensures that only the leave region is extracted during processing.

4.4 Greyscaling

Greyscaling is the first step of the conversion to black and white process as explained in section 2.2.1. Using OpenCV the input image is greyscaled during the image loading process using the `imread` function as follows:

Listing 4-2: C++ Greyscaling

```
Mat image_data = imread(file_path,IMREAD_GREYSCALE);
```

The `imread` function loads the image pointed to by `file_path` into an OpenCV matrix object called `image_data`. The `IMREAD_GREYSCALE` flag causes the function to greyscale each pixel during the loading process according to Equation 2-1. The pixel intensity values are stored as 8 bit unsigned characters. The results of greyscaling are shown in Figure 4-2.

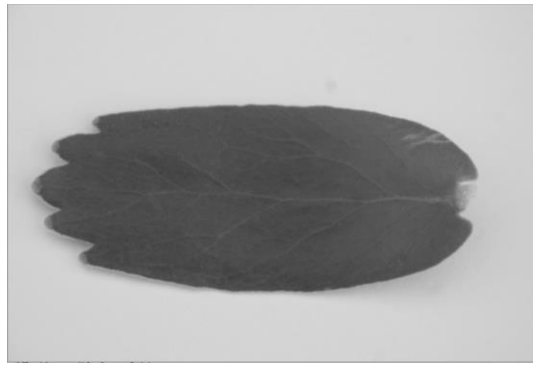


Figure 4-2 Example of Greyscaled Image

4.5 Otsu Thresholding

Otsu thresholding is achieved using the OpenCV `threshold` function as follows:

Listing 4-3: C++ Otsu Thresholding

```
threshold(src_image, dst_image, threshold_value, max_threshold, THRESH_OTSU);
```

This function applies the thresholding algorithm explained in section 2.2.1 to `src_image` and stores the result in `dst_image`. The threshold value is normally set to the value of `threshold_value`, but by providing the `THRESH_OTSU` flag the function calculates the threshold value by Otsu's method and uses this value instead. The value of `max_threshold` must be set to the maximum possible threshold value which is 255 since this implementation uses 8 bit unsigned pixel values.

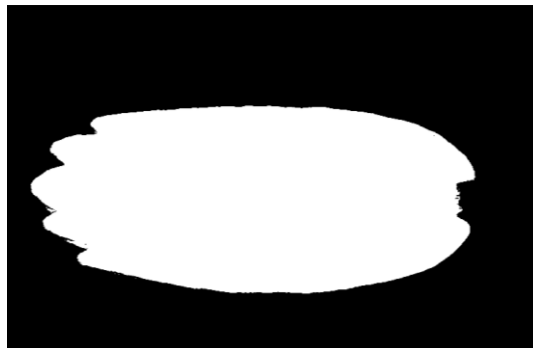


Figure 4-3 Example of Otsu Threshold

4.6 Finding Image Contours

This function serves the same purpose as the 'Region Detection' stage explained in section 2.2.2. However in addition to detecting regions in the input image, the regions' outline or 'contours' are also extracted for use in the Extract Features step later on. This processing step is implemented using the OpenCV `findContours` function:

Listing 4-4: C++ Find Image Contours

```
vector<vector<Point>> contours; //Leaf outline as a vector of 2D Points
findContours(src_image, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_NONE);
```

The `findContours` function detects all the regions in the binary (black and white) image `src_image`. Regions are defined as any 8-connected group of pixels that have a pixel intensity greater than 0 (not

black). OpenCV uses the algorithm described by S. Suzuki et al in [50] for detecting regions and contours. The function then extracts the region contours and stores them as vectors of points in contours. Figure 4-4 shows an example of the input image with its contours extracted. OpenCV can extract region contours in several different ways, for this implementation only the external contours of the region are needed as these correspond to the leaf outline, to achieve this the CV_RETR_EXTERNAL flag is specified. OpenCV can also approximate the region contours to reduce processing time, however this would negatively impact prediction accuracy. Therefore no contour approximation was used by specifying the CV_CHAIN_APPROX_NONE flag.

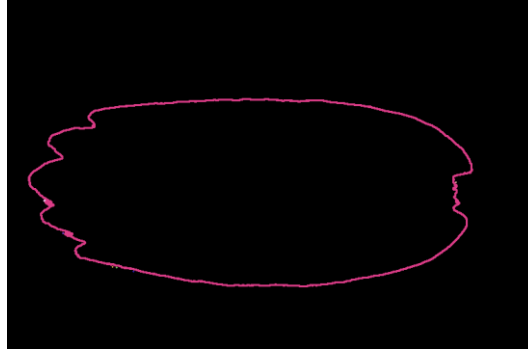


Figure 4-4 Example of Extracted Region Contours

4.7 Image Blurring

Image blurring is necessary if more than one contour is found in the image, blurring implemented using OpenCV's blur function:

Listing 4-5: C++ Image Blurring

```
if (contours.size() > 1)
{
    blur(src_image, dst_image, window_size);
    window_size.height = window_size.height + 2;
    window_size.width = window_size.width + 2;
}
```

The blur function applies an average filter to src_image and stores the result in dst_image. The averaging window size is determined by the window_size parameter. This function was called repeatedly with larger window sizes until only one region is found in the image, this region corresponds to the leaf profile.

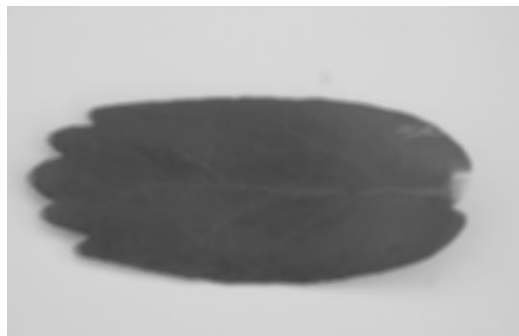


Figure 4-5 Example of Blurred Image

4.8 Calculation of Image Orientation

Calculation of the image orientation was implemented through calculation of image moments as explained in section 2.2.3. To achieve this in the FLORA C++ implementation the OpenCV moments function:

Listing 4-6: C++ Calculation of Image Moments

```
Moments image_moments = moments(src_image, is_binary);

//Spatial moments
double m00 = image_moments.m00;
double m01 = image_moments.m01;
...
//Central moments
double mu00 = image_moments.mu00;
double mu01 = image_moments.mu01;
...
```

The moments function calculates all spatial and central image moments up to the third order from `src_image` and stores them in the Moments object `image_moments`. Image moments are calculated in OpenCV using equations (2-6) and (2-7) in section 2.2.3 [49]. By specifying the `is_binary` flag to be true the function will treat all non-zero pixels as 1s (rather than the actual value for white pixels which is 255). This ensures that every pixel of the leaf image has equal ‘weight’ and the image orientation calculated will be correct. Additionally this means that the 0th moment (`m00`) will correspond to the region area which is used later on in the Extract Features step.

After the image moments are calculated the image orientation is calculated using equation (2-9):

Listing 4-7: C++ Calculation of Image Orientation Using Calculated Moments

```
double theta = 0.5*atan2(2*(mu11/m00), (mu20/m00) - (mu02/m00));
```

4.9 Calculation of Rotation Matrix

Once the rotation angle is determined the rotation matrix is then determined:

Listing 4-8: C++ Calculation of Rotation Matrix

```
Point2f centre_pt = (src_image.cols, src_image.rows);
Mat rotation_matrix = getRotationMatrix2D(centre_pt, -theta, isf);
```

The rotation matrix is created using OpenCV’s `getRotationMatrix2D` function, this function calculates a rotation matrix as using a slightly different formula as the one proposed in equation (2-10)[32]:

$$R(\theta) = \begin{bmatrix} \alpha & \beta & (1 - \alpha) \times \text{centre_pt.x} - \beta \times \text{centre_pt.y} \\ -\beta & \alpha & \beta \times \text{centre_pt.x} + (1 - \alpha) \times \text{centre_pt.y} \end{bmatrix} \quad (4-1)$$

Where $\alpha = \cos(\theta)$ and $\beta = \sin(\theta)$

The function produces a rotation matrix that can be used to rotate any set of 2D points θ degrees about the point `centre_pt`. For the FLORA implementation the `centre_pt` is defined to be the centre of the input image since it is expected that the leaf region will be in the centre of the image. OpenCV also allows one to specify an isotropic scale factor `isf`, for the FLORA implementation this was set to 1.0.

4.10 Apply Rotation Matrix Transformation

The final step in the image rotation stage is to apply the calculated rotation matrix to the input image. To achieve this the input image is treated as a 2D matrix of values and the rotation matrix is applied using the OpenCV `warpAffine` function:

Listing 4-9:C++ Application of Rotation Matrix using a Matrix Transform

```
int max_len = max(src_image.cols, src_image.rows);
Size new_size(max_len,max_len);
cv::warpAffine(src_image, dst_image, rotation_matrix, new_size);
```

The `warpAffine` function transforms the image `src_image` using `rotation_matrix` and stores the result in `dst_image`. The `new_size` parameter determines what the size of the destination image should be, for the FLORA implementation this is set as a 2D square with each side the length of either the width or height of the input image, depending on which is bigger. This ensures that the rotated region does not get cut off during rotation. Figure 4-6 shows an example of the output of the rotation function, the image on the left is inputted to the system and the image on the right is the result after rotation.

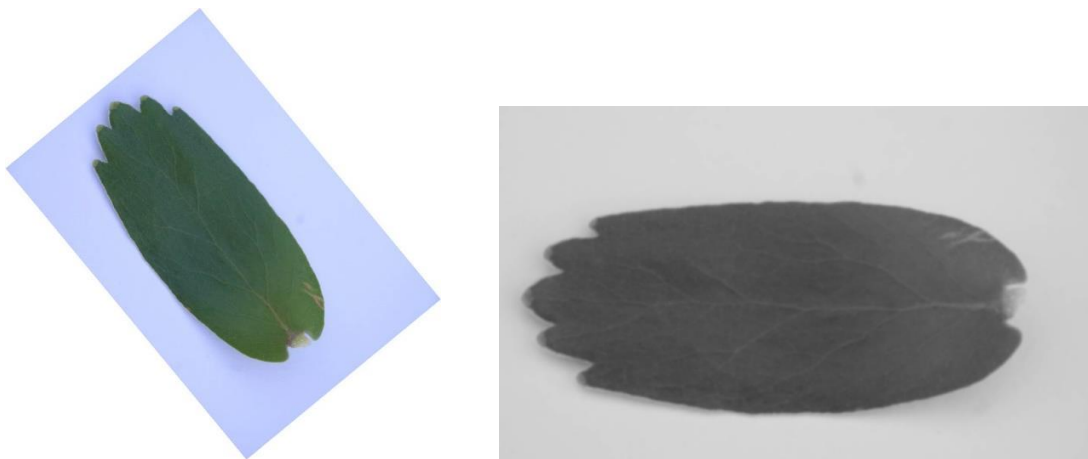


Figure 4-6 Example of Image Rotation

4.11 Determining Region Convex Hull

The convex hull of the image region is calculated using the OpenCV `convexHull` function:

Listing 4-10: C++ Convex Hull

```
vector<vector<Point>> convex_hull;
convexHull(input_points, convex_hull);
```

The function takes a set of points that exist in the matrix or vector `input_points`, determines the convex hull and stores it in the point vector `hull`. For the FLORA implementation the set of input points is simply the image produced following the rotation step.

OpenCV calculates the convex hull using an algorithm proposed by J. Sklansky in [51]. Figure 4-7 shows an example of a convex hull generated for the input image.

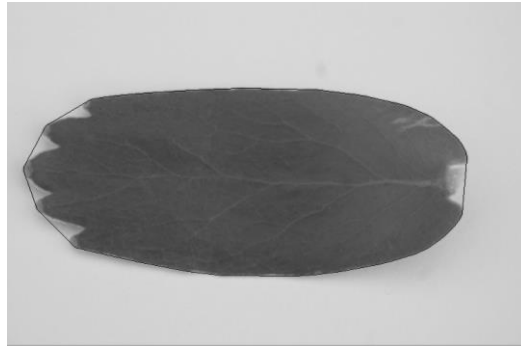


Figure 4-7 Example of Image Convex Hull Output

4.12 Determining Region Bounding Box

The up-right bounding box of the image region is calculated using the OpenCV `boundingRect` function:

Listing 4-11: C++ Bounding Box

```
Rect boundRect = boundingRect(src_image);
```

This function simply determines the vertex coordinates of the smallest up-right bounding rectangle that contains the region in `src_image`. Figure 4-8 shows an example of the bounding box generated around the input image.

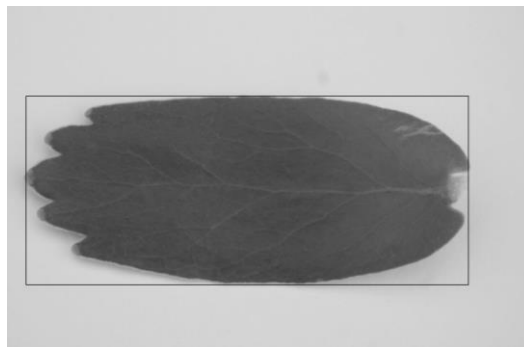


Figure 4-8 Example of Bounding Box Output

4.13 Region Area & Perimeter Length

As stated in in section 4.7 the image area corresponds to the 0th spatial image moment calculated during the Calculation of Image Orientation Step. The length of the perimeter is simply calculated by adding up the number of pixels in the region contour extracted during the Finding Image Contours step, OpenCV provides the `arcLength` function for this purpose:

Listing 4-12: C++ Region Area & Perimeter


```
double regionArea = m00;
double regionPerimeter = arcLength(contours, is_closed_contour);
```

The `arcLength` function takes an additional `is_closed_contour` parameter that is used to tell the function that the contour being traced is a closed contour. This is set to `true` for the FLORA implementation.

4.14 Convex Hull Area & Perimeter Length

Since the convex hull was already calculated in the Determine Region Convex Hull step, the convex hull area can easily be calculated by applying Green's Theorem [14]:

Given the convex hull $f(x, y)$, it is treated such that:

$$f(x, y) = \begin{cases} 1, & \text{if } (x, y) \text{ falls on the convex hull} \\ 0, & \text{otherwise} \end{cases}$$

The area A of the convex hull is calculated as:

$$A = \oint F \cdot ds$$

Where F is a vector field such that:

$$\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y} = f(x, y) = 1$$

One such solution for F is $F(x, y) = (-\frac{y}{2}, \frac{x}{2})$, This produces the solution:

$$A = \frac{1}{2} \oint x \cdot dy - y \cdot dx$$

OpenCV provides the `contourArea` function that implements the above computation and therefore this function was chosen for use in the FLORA implementation. The convex hull perimeter is calculated in the same way as the region area in section 4.12:

Listing 4-13: C++ Convex Hull Area & Perimeter

```
double hullArea = contourArea(convex_hull);
double hullPerimeter = arcLength(convex_hull, is_closed_contour);
```

4.15 Calculation of Feature Descriptors

The feature descriptors used by FLORA are explained in section 2.2.4, they are calculated in the C++ implementation as follows:

Listing 4-14: C++ Calculation of Feature Descriptors

```
//Rectangularity
double recty = regionArea / boundRect.area();
//Solidity
double solidy = RegionArea / hullArea;
//Convex Perimeter Ratio
double cvp = regionPerimeter / hullPerimeter;
//Formfactor
double ff = 4 * PI*M00 / pow(regionPerimeter,2);
```

These 4 feature values are then passed to the KNN classifier can either be used for training the classifier or to make a prediction.

4.16 Prediction Using KNN Classifier

As explained in section 2.2.5 the KNN Classifier must be created and trained initially before it can be used to make predictions. The C++ FLORA implementation includes a KNN module that allows one to easily create, train and make predictions using OpenCV's implementation of the KNN classifier. Additionally a testing mode has also been included that allows the prediction accuracy to be tested easily.

For ease of use the KNN Classifier has been designed in such a way that it can be easily integrated with the FLORA front end but at the same time can also work as a standalone application that can be run from a command line. When run from command line the KNN program can be used to create KNN classifiers using any specified sample databases, it can make predictions using a created classifier or it can test a classifier's prediction accuracy by testing it against a test database.

4.16.1 Creation & Training of a KNN Classifier

In order to create a KNN classifier the OpenCV `KNearest` class is used. An empty `KNearest` object is first created and then trained using a folder of training images, the classifier is saved to a file for quick and easy loading. The code in listing 4-15 shows the pseudocode for the training function:

Listing 4-15: C++ Pseudocode for KNN Classifier Creation

```
void train(string save_file_name, bool append_data, string folder_path){

    Mat data;                //OpenCV matrix that represents the final classifier database, each
                             //row is a set of feature descriptors that belong to a plant species

    vector<int> responses;    //An integer vector of responses (species numbers) that correspond to
                             // each row in the data matrix

    Mat row;                 //Represents a row in data matrix
    KNearest knn = KNearest::create(); //Create empty KNN classifier object

    for(all valid images in folder_path){

        //Extract next image's features
        vector<float> extracted = extractFeatures(next_image);
        //populate row with extracted features
        row.at<float>(0) = extracted[0];
        row.at<float>(1) = extracted[1];
        row.at<float>(2) = extracted[2];
        row.at<float>(3) = extracted[3];
        //add row to database matrix
        data.push_back(row);
        //add species number to responses
        responses.push_back(current_image_species_number);
    }
    if (!append_data){//if there's no more folders to process we can train the classifier

        //First we create OpenCV Training Data using the extracted data rows and responses
        TrainData tdata = TrainData::create(data, cvresponses);
        //Set default k to 1 for 1st nearest neighbour
        knn->setDefaultK(1);
        //Train the classifier using the Training Data
        knn->train(tdata);
        //Save classifier to file and training is complete!
        knn->save(save_file_name);
    }
}
```

```
}  
}
```

The train function is called multiple times for each folder of training images specified by the user, the classifier database is only saved to file once all images have been processed. The classifier database is stored in a YAML file with the default name 'knn_classifier.yaml'. YAML is a human readable data serialisation format [52], it was chosen to allow easy editing and updating of classifier databases. The database can also be saved to XML format if desired by modifying the file extension to '.xml'.

In order for the function to recognise training images they must be named using the following naming scheme:

'species number'-'incremental number'.jpg

Where the 'species number' is the number that corresponds to a particular plant species defined in the class_info.php file on the FLORA server and the 'incremental number' is a counting number that starts from 1 and increases for each image that belongs to the same species. The function has been written to handle gaps in the image naming within any specified training folder.

To initiate training one must simply run the compiled FLORA program from a command line specifying the '-train' parameter, additionally the '-folder-paths' parameter can be specified with at least one folder path to train a classifier using user specified image folders. If the '-folder_paths' parameter is left out, the default 'Train' folder will be used.

4.16.2 Prediction Using KNN Classifier

To carry out a prediction using a previously created KNN classifier the classifier must first be loaded, then the input image must have its feature's extracted and finally these features must be passed to the classifier to perform a KNN search which will return the desired prediction. The pseudocode in listing 4-16 shows how this is achieved in the C++ implementation.

Listing 4-16: C++ Pseudocode for Prediction Using KNN Classifier

```
knn_result findNearest(string image_path){  
  
    KNearest knn = load_classifier("knn_classifier.yaml");//Load pre-trained model  
    Mat input_features; //Represents a set of extracted features  
    vector<float> neighbors; //vector of found nearest neighbors ordered by distance  
    vector<float> distances; //vector of distance values corresponding to the found  
                           //nearest neighbours  
  
    //Extract sample features  
    vector<float> extracted = extractFeatures(image_path);  
    //populate row with extracted features  
    input_features.at<float>(0) = extracted[0];  
    input_features.at<float>(1) = extracted[1];  
    input_features.at<float>(2) = extracted[2];  
    input_features.at<float>(3) = extracted[3];  
    // find NUM_NEAREST (default 20) nearest neighbors and their distances from the  
    // sample in the feature space  
    knn->findNearest(input_features, NUM_NEAREST, neighbors, distances);  
    knn_result result; //create an object to hold results of knn search  
    result.prediction = neighbours[0]; //The predicted species is the nearest neighbor  
    result.neighbours = neighbours; //Array of 20 nearest neighbors  
    result.distances = distances; //Distance of each neighbor from sample  
    return result;  
}
```

```
}
```

The FLORA front end requires several nearest neighbours to be found so that a list of other possible matches can be shown to the user in order of matching percentage. For this reason when doing a KNN search the function will return the 20 nearest neighbours along with their distances from the sample in the feature space. This number can be increased by modifying the NUM_NEAREST constant in the knn.cpp file. Since multiple results are returned, they are stored in a special 'knn_result' object shown in Listing 4-17.

Listing 4-17: C++ Struct Used for Holding KNN Prediction Results

```
struct knn_result{
    int prediction;           //The predicted species number (same as neighbors[0])
    vector<float> neighbors;  //Vector of nearest neighbors ordered by distance
    vector<float> distances;  //Vector of distances corresponding to nearest neighbor
};
```

The matching accuracy percentages are then calculated by normalising the returned distances and subtracting them from 1 as shown in Listing 4-18:

Listing 4-18: C++ Code Snippet Showing Calculation of Prediction Accuracy Percentages

```
//first we find the farthest neighbor and its distance
float max =0;
for (int i = 0; i < result.distances.size() ;i++){
    if (result.distances[i] > max) max = result.distances[i];
}
...
//Matching percentage can then be calculated for each neighbour as follows
float percentage_match = (1.0 - result.distances[i] / max) * 100;
...
```

This function also includes additional code to ensure that duplicate species are not returned, in other words if any more than 1 of the 20 nearest neighbours belong to the same species, only the nearest neighbour is returned and the rest of the neighbours with the same species are disregarded. This is a similar approach to the one used by Katz in [6].

This function is designed to be called directly from the FLORA front end as well as from the command line. To call from the command line simply run the program specifying the '-f' flag followed by an image path, the program will then print out its prediction to the command line. If called from the FLORA front end, the function will output the results in the format the FLORA front end expects.

4.16.3 Testing of the KNN Classifier

The C++ KNN implementation also includes a testing function for added convenience. This function allows the user to specify a folder of test images to test a particular KNN classifier against. This function can be used to test the KNN classifier using the method described in section 3.7.1. Listing 4-19 shows the C++ pseudocode for testing of a classifier.

Listing 4-19: C++ Pseudocode for Testing of a Classifier

```
float test_knn(string knn_file, string test_path){
    KNearest knn = load_classifier(knn_file); //Load pre-trained model

    Mat test_data; //Matrix to hold test samples
    vector<float> expected; //vector to hold expected responses
    vector<float> results; //vector holding actual responses
```

```

Mat row; // single row of sample data;
int correct = 0;
int total = 0;

for(all valid images in test_path){

    std::vector<float> extracted = extractFeatures(next_image);
    //populate a row with extracted features
    row.at<float>(0) = extracted[0];
    row.at<float>(1) = extracted[1];
    row.at<float>(2) = extracted[2];
    row.at<float>(3) = extracted[3];
    //add row to test data matrix
    test_data.push_back(row);
    //add species number to expected result vector
    expected.push_back(i);
    //increment total counter
    total++;

}

//find 1st nearest neighbor for each sample row in test data matrix and store the
//predictions in results
knn->findNearest(test_data,1, results);

//Check the accuracy of the classifier by comparing the predictions to expected
//results
for (int i = 0; i < results.size(); i++){
    if (results[i] == expected[i]){
        correct++;
    }
}
//Return the overall percentage accuracy
return (float)correct / (float)total;
}

```

This function requires that all images in the test folder be named according to the naming scheme defined in section 4.16.1. This function is used in the Performance Analysis & Results section to evaluate overall prediction accuracy.

The test function can be called from the command line by specifying the ‘-test’ flag. Additionally the ‘-folder_path’ parameter can be specified along with a path to a test image folder in order to test the KNN classifier with a user specified test folder. If the ‘folder_path’ parameter is not specified then the default ‘Test’ folder is used.

5. FLORA CUDA Acceleration

This section details the GPU implementation designs created during the GPU design iteration phase of the development cycle. GPU acceleration of FLORA was achieved by re-implementing specific functions from the C++ implementation that were identified to be bottlenecks in the system. OpenCV’s CUDA accelerated libraries were used where possible to save time otherwise custom CUDA kernels were developed. After each GPU design iteration was completed, the system performance was evaluated. The performance results for the CUDA implementations are shown in Section 12.1.3.

5.1 Initial Setup

Each design iteration starts with execution time profiling of the full FLORA backend, this is achieved using the Microsoft Visual Studio 2013 performance profiler. To ensure all the relevant features of the

program are profiled, the program is run each time using the command line prediction mode by specifying the '-f' parameter with a chosen input image. The input image chosen is shown in figure 5-1, the image was scaled down to the same resolution used by the original FLORA system. The KNN classifier used was trained from Image Dataset 3 compiled by Katz in [6].



Figure 5-1 Input Image Used for Profiling (800 X 533 Resolution)

Before design iteration was initiated it was first ensured that the prediction accuracy of the C++ implementation was acceptable. Testing of the C++ implementation produced a prediction accuracy of 89.23%, this is virtually the same as the original FLORA implementation which achieved a prediction accuracy of 89.74% [6]. More information on accuracy results can be found in the results section.

5.2 A Note on OpenCV CUDA Accelerated Image Processing Library Initialisation

It was discovered that like the standard OpenCV image processing libraries, the CUDA accelerated libraries also require initialisation. This initialisation always occurs when the first CUDA accelerated function is called, regardless of which function was called. Initialisation of the CUDA libraries always took a constant time of about 1 second to complete. In order to prevent the CUDA library initialisation from affecting the actual function performance of the program a 'dummy' function was called at the beginning of the program that causes the CUDA library initialisation to take place before any actual program functions are called. Listing 5-1 shows the dummy function.

Listing 5-1 Dummy Operation Called to Initialise OpenCV CUDA Library

```
void Dummy(){
    cuda::GpuMat dummy;
    dummy.upload(Mat(8, 8, CV_8UC1));
    cuda::Filter boxFilter = cv::cuda::createBoxFilter(CV_8UC1, CV_8UC1, Size(2, 2));
    boxFilter->apply(dummy, dummy);
}
```

The code in Listing 5-1 applies a basic averaging filter to an array of 8 X 8 arbitrary values. This function would normally have negligible execution time therefore any delay that was experienced when calling this function at the beginning of the program could be directly attributed to the OpenCV CUDA library initialisation. Note that this CUDA initialisation delay occurs on top of the initialisation delay described in section 4.2, therefore if both standard C++ and CUDA accelerated OpenCV libraries are used, both dummy functions must be called at the beginning of the program to ensure accurate results.

5.3 Iteration 1: Image Greyscaling

Table 12-2 shows the execution time profile of the original C++ implementation before any GPU acceleration took place. As can be seen from the execution times, the first major bottleneck in the program is the loading of the input image in greyscale. This function can be split into two parts:

1. Load original colour image into host memory
2. Convert the colour pixel values to greyscale values

The first step cannot be implemented on the GPU as there is no mechanism available for loading images directly from storage to GPU memory. Greyscaling on the other hand can be implemented on the GPU as explained in section 2.6.1i.

It was found however that loading an image in colour is over 2 times slower than loading an image in greyscale, for this reason a GPU implementation of greyscaling was rendered redundant and not explored further.

5.4 Iteration 2: Apply Rotation Matrix

Since no GPU optimisation was possible in iteration 1, iteration 2 dealt with the next major bottleneck: Application of the rotation matrix. OpenCV provides a CUDA accelerated version of the `warpAffine` function used in the original C++ implementation shown in section 4.9:

Listing 5-2: CUDA Accelerated Application of Matrix Transform

```
cuda::warpAffine(src_image, dst_image, rotation_matrix, new_size);
```

This function works in much the same way as the original `warpAffine` function but the `src_image` and `dst_image` must be located in GPU memory, therefore the OpenCV `upload` and `download` functions are used to transfer the input image to the GPU and back to the host after the transformation. Listing 5-3 shows how this is achieved in code. Figure 5-2 shows an example of the input image (on the left) being rotated so that its major axis lies flat with the x axis (right). After implementing the CUDA accelerated rotation function the execution time and accuracy of the system were tested, the results can be found in Table 12-3.

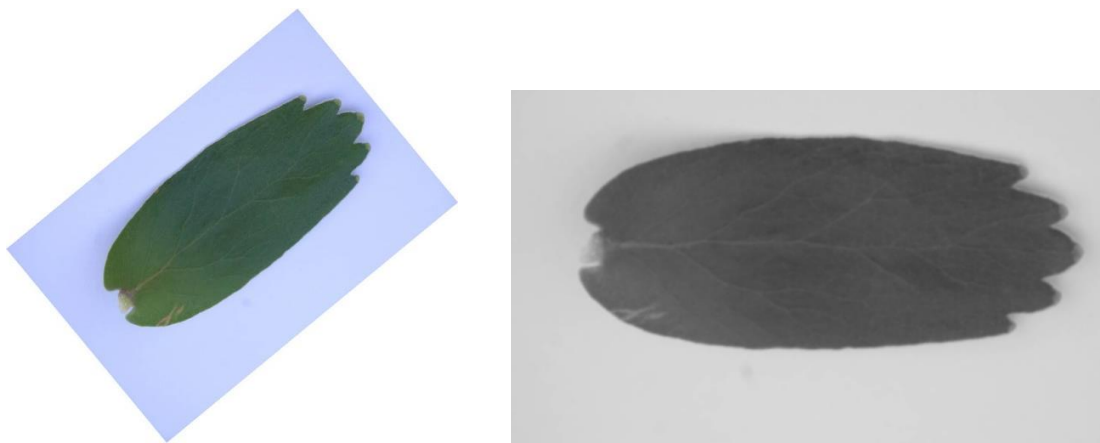


Figure 5-2 Example of CUDA Accelerated Output

Listing 5-3: Method for Uploading & Downloading Images to and from GPU Memory

```
Mat host_image;  
cuda::GpuMat gpu_image;  
gpu_image.upload(host_image); //upload from host to GPU  
  
... //GPU processing
```



```
gpu_image.download(host_image); //download from GPU to host
```

5.5 Iteration 3: Otsu Thresholding

As can be seen in Table 11-3, after iteration 2 the next major bottleneck in the system was Otsu Thresholding. To remove this bottleneck a GPU accelerated implementation of Otsu thresholding was implemented based on the implementation proposed by Singh et al in [8]. The pseudocode for the GPU code is shown in Listing 5-4.

Listing 5-4: Pseudocode for Otsu Thresholding on GPU

```
main()
{
    Define block and grid
    Kernal_Histogram(numOfPixels, image, histogram)
    Calculate threshold from histogram on CPU
    Kernal_Threshold(numOfPixels, image, threshold)
}

Kernal_Histogram(numOfPixels, image, histogram){
    Declare shared memory subHist[numOfThreads][256];
    for each data pixels in window
        subHist [threadId][currentPixelValue]++;
    end for
    Apply scan to merge subHist into histogram
}

Kernal_Threshold(numOfPixels, image, threshold){
    for each data pixels in window
        if(currentPixelValue<threshold)
            currentPixelValue = 0; //Black
        else
            currentPixelValue = 255; //White
    end for
}
```

In order to implement this function two CUDA kernels are required, one kernel to calculate the grey level histogram and one to apply the thresholding function. The calculation of Otsu's thresholding value is implemented on the CPU because the histogram is always a fixed size (256 elements in the FLORA implementation) which is usually small and processing of this histogram would have negligible performance impact.

To calculate the grey level histogram on the GPU, OpenCV's CUDA implementation of the `calcHist` function was used:

Listing 5-5: CUDA Accelerated Grey Level Histogram Calculation

```
cuda::calcHist(src_image, hist_gpu); //Calculate grey level histogram
```

This function takes a binary image `src_image` that has been loaded into GPU memory and calculates the grey level histogram as a 256 element array of integers `hist_gpu` also stored in GPU memory. The Otsu

value can then be calculated using Otsu's method [7] using equation (2-3). Listing 5-6 shows the C++ implementation of Otsu's method used in the FLORA implementation.

Listing 5-6: Custom C++ implementation of Otsu's Method

```
float findOtsuValue(){
    //calculate the probability of each histogram value and store them in a probability
    array
    for (int k = 0; k <= 255; k++){
        probability[k] = (float)hist_gpu.at<int>(k) / (float)src_image.area();
        mu += hist.at<int>(k)*k; //total sum of grey levels,
    }
    //Initial probability p1 which is equal to just the probability of the 1st grey value
    p1 = probability[0];
    //initial omega which is the sum of probabilities before 1, which is equal to p1
    omega1 = p1;
    //initial mean before the 1 has to be 0. mu(1)=0
    mu1 = 0;
    //initial mean after the 0. Initially set to 0. This gets reset in the algorithm
    mu2 = 0;
    //mean grey level (mu) calculation
    mu /= src_image.area();
    float value = 0;
    omega1prev = omega1; //set previous omega1, omega1(t), to equal the current omega1
    betweenvariance = 0; //between group variance
    maxbetweenvariance = 0; //max between group variance
    //Calculate ostu value

    for (int t = 1; t<256; t++){
        //omega1next = omega1(t+1)
        omega1next = omega1prev + probability[t + 1];
        //mu1next = mu1(t+1)
        mu1next = (omega1prev*mu1 + (t + 1)*(probability[t + 1])) / omega1next;
        //mu2next = mu2(t+1)
        mu2next = (mu - omega1next*mu1next) / (1 - omega1next);
        //calculate between group variance
        betweenvariance = omega1prev*(1 - omega1prev)*((mu1 - mu2)*(mu1 - mu2));
        // if current variance > max variance, change the max between group variance,
        and change the optimized threshold to t
        if (betweenvariance>maxbetweenvariance){
            maxbetweenvariance = betweenvariance;
            value = t; //set new optimized threshold
        }
        //set omega1(t) to omega1(t+1) for next iteration
        omega1prev = omega1next;
        //Set mu1(t) equal to mu1(t+1) for next iteration
        mu1 = mu1next;
        //set mu2(t) equal to mu2(t+1) for next iteration
        mu2 = mu2next;
        //this eliminates divide by 0 errors
        if (q1next == 0){
            mu1 = 0;
        }
    }
    return value;
}
```

Lastly the image is thresholded using OpenCV's CUDA implementation of the threshold function:

Listing 5-7: CUDA Acelerated Thresholding

```
cuda::threshold(src_image, dst_image, threshold_value, max_threshold, THRESH_BINARY);
```

This function applies a simple GPU optimized binary threshold to `src_image` and stores the result in `dst_image`. Both the image source and destination must exist in GPU memory. The `threshold_value` is taken from the result of the Otsu's method calculation. Figure 5-3 shows an example of the output of the CUDA accelerated Otsu Threshold function. After successfully implementing the Otsu Threshold function the performance of the system was tested again, the results are shown in Table 12-4.

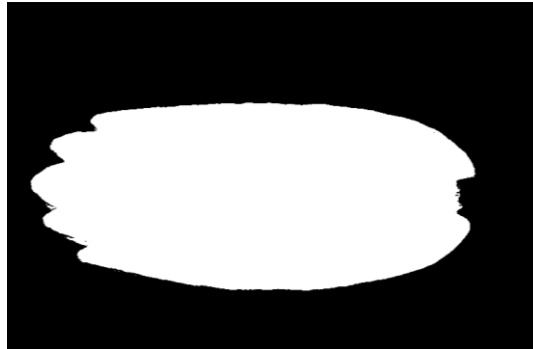


Figure 5-3 Example of CUDA Accelerated Otsu Threshold

5.6 Iteration 4: Find Image Contours

Table 12-4 shows the execution time profile of the FLORA system after iteration 3. As can be seen from the table the next major bottleneck is the Finding of Image Contours. In the original C++ implementation this was achieved by the OpenCV `findContours` function which performed two operations:

1. Identify connected regions in an input image
2. Determine the region contours of identified regions

Only the first step was implemented on the GPU, the second step was not implemented due to time constraints. Determining region contours requires the use of a contour tracing algorithm [21] however no suitable GPU accelerated implementation could be found for this function and due to time constraints one could not be developed from scratch.

To implement region identification on the GPU a modified version of the Connected Component Labelling algorithm proposed by O. Štava in [34] was used. An overview of this algorithm is given in section 2.6.2ii. The CCL algorithm is implemented directly in CUDA C code with C++ wrapper classes adapted from sample code provided by O. Štava for his implementation.

Figure 5-2 shows an overview of the GPU accelerated CCL algorithm implemented in FLORA.

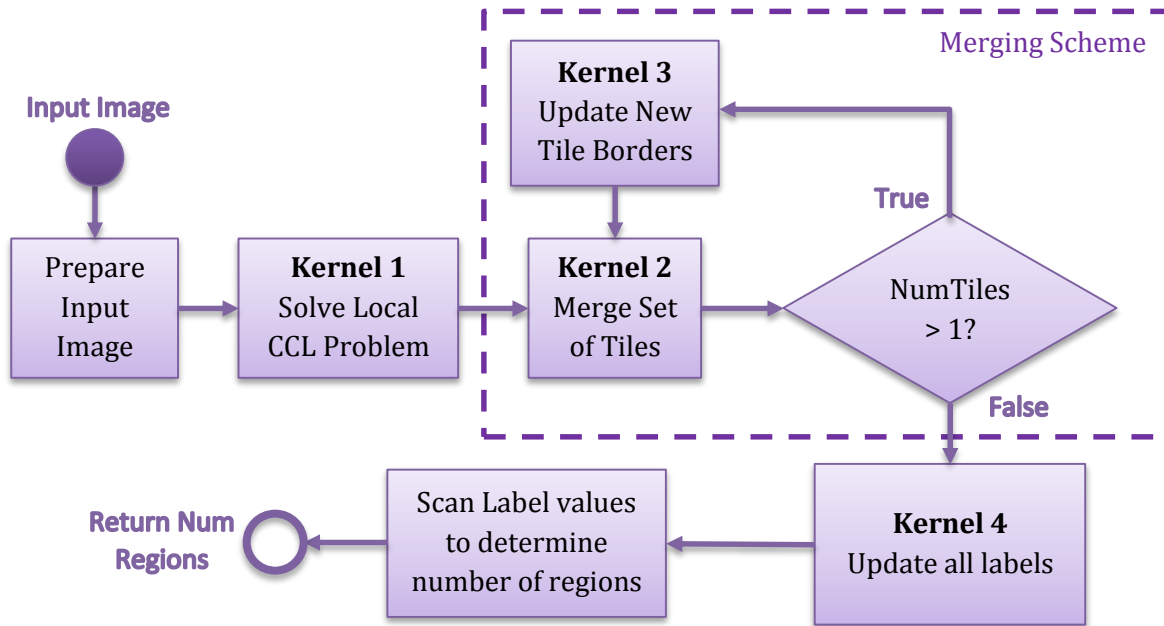


Figure 5-4 Overview of FLORA GPU CCL Algorithm

5.6.1 Prepare Input Image

Before the input binary image can be processed on the GPU it must first be prepared for processing. O. Štava's algorithm requires that the image size be a factor of 2, therefore the image borders must be extended if they are not the correct size. Listing 5-8 shows the C++ code used to resize the input image.

Listing 5-8: C++ Code Used to Resize Input Image for Use with GPU CCL Algorithm

```

double checkX = log2(src_image.cols);
double checkY = log2(src_image.rows);
double rX = floor(checkX);
double rY = floor(checkY);
int plus1 = 1;
//check if width or height is not a factor of 2
if (checkX - rX != 0 || checkY - rY != 0){
    double max = fmax(rX, rY);
    double max2 = fmax(checkX, checkY);
    //if the largest dimension is a factor of 2 then we don't need to increase the
    image size in both dimensions
    if (max2 - max == 0) plus1--;
    //The new image size is the smallest 2D square that can fit the image with
    dimensions a factor of 2
    int newSize = (int)pow(2, max + plus1);
    //Create a new empty matrix of the required size
    Mat resized = Mat::zeros(newSize, newSize, src_cpu.type());
    //Determine border widths
    int top = (newSize - src_cpu.rows) / 2;
    int right = (newSize - src_cpu.cols) / 2;
    int bottom = (int)ceil((newSize - src_cpu.rows) / 2.0);
    int left = (int)ceil((newSize - src_cpu.cols) / 2.0);
    //We replicate the edge pixels of the original image to fill up the added
    image area, this should preserve the grey level histogram.
    copyMakeBorder(src_cpu, resized, top, bottom, left, right, BORDER_REPLICATE);
    //Replace the original input image with the new border extended version
    src_cpu = resized;
}

```



Figure 5-6 Example of Input Image with Borders Extended

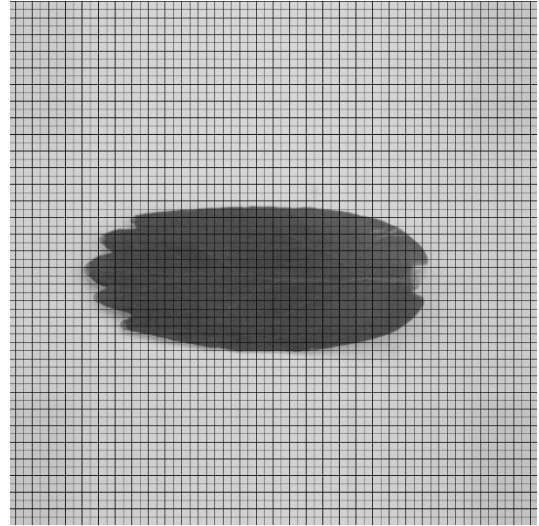


Figure 5-5: Example of Input Image Split into Tiles

Figure 5-6 shows how an input image is resized if its original size is not a factor of 2. The code used in Listing 5-5 makes use of the OpenCV `copyMakeBorder` function in `BORDER_REPLICATE` mode. This resizes the input image by replicating the border pixels in all directions, this ensures that the leaf features are not altered and the grey level histogram is not altered in a way that affects the Otsu thresholding.

After image resizing the image must be divided into ‘tiles’, the CCL problem is solved on each tile in parallel using kernel 1 and then merged using the merging scheme to produce the final solution. Tile size was set to 256 elements per tile. Figure 5-3 shows how the input image is split into tiles for processing.

5.6.2 Kernel 1: Solve Local CCL Problem

Kernel 1 computes a modified version of the Union-Find algorithm [36] for each tile in the input image. Each tile corresponds to an instance of the kernel and each element of each tile corresponds to a single CUDA thread. The index of each CUDA thread will therefore map directly to a particular pixel in the image. Listing 5-9 shows the pseudocode for kernel 1 (Adapted from [34]).

Listing 5-9: Pseudocode for Kernel 1 - Local CCL Solver

INPUT: `dImgData` //2D array of input data

OUTPUT: `dLabelsData` //2D array of labels (equivalence array)

```
shared sImg[]; //shared memory used to store the image data
shared sLabels[]; //shared memory that is used to compute the local solution
localIndex = localAddress(thread_index); //local address of the element in the shared
memory
sImg[localIndex] = loadImgDataToSharedMem(dImgData, thread_index.x, thread_index.y);
sImg[borders] = setBordersAroundSubBlockToZero();
shared sChanged[1];
syncThreads();
label = localIndex;
while(true) {
    //Pass 1 of the CCL algorithm
    sLabels[localIndex] = label;
    if(thread_index == (0,0)) sChanged[0] = 0;
    syncThreads();
    newLabel = label;
    //find the minimal label from the neighboring elements
```

```

for(allNeighbors)
    if(sImgs[localIndex] == sImgs[neighIndex]) newLabel = min(newLabel,
        sLabels[neighIndex]);

syncThreads();
//If the new label is smaller than the old one merge the equivalence trees
if(newLabel < label) {
    atomicMin(sLabels+label, newLabel);
    sChanged[0] = 1
}
syncThreads();
if(sChanged[0] == 1) break; //the local solution has been found, exit the loop
//Pass 2 of the CCL algorithm
label = findRoot(sLabels, label);
syncThreads();
}
//store the result to the device memory
globalIndex = globalAddress(block_index, thread_index)
dLabelsData[globalIndex] = transferAddressToGlobalSpace(label);

int findRoot(equivalence_array, element_address){
    while(equivalence_array[element_address] != element_address)
        element_address = equivalence_array[element_address];
    return element_address;
}

```

Kernel 1 applies the CCL algorithm each time two non-zero neighbours are found that do not share the same label. Once all connected components in the local solution are labelled correctly, the kernel ends. Kernel 1 also adds a border of empty pixels around each tile for use in the merging scheme. Figure 5-7 (taken from [34]) shows a graphical representation of how kernel 1 produces each local solution.

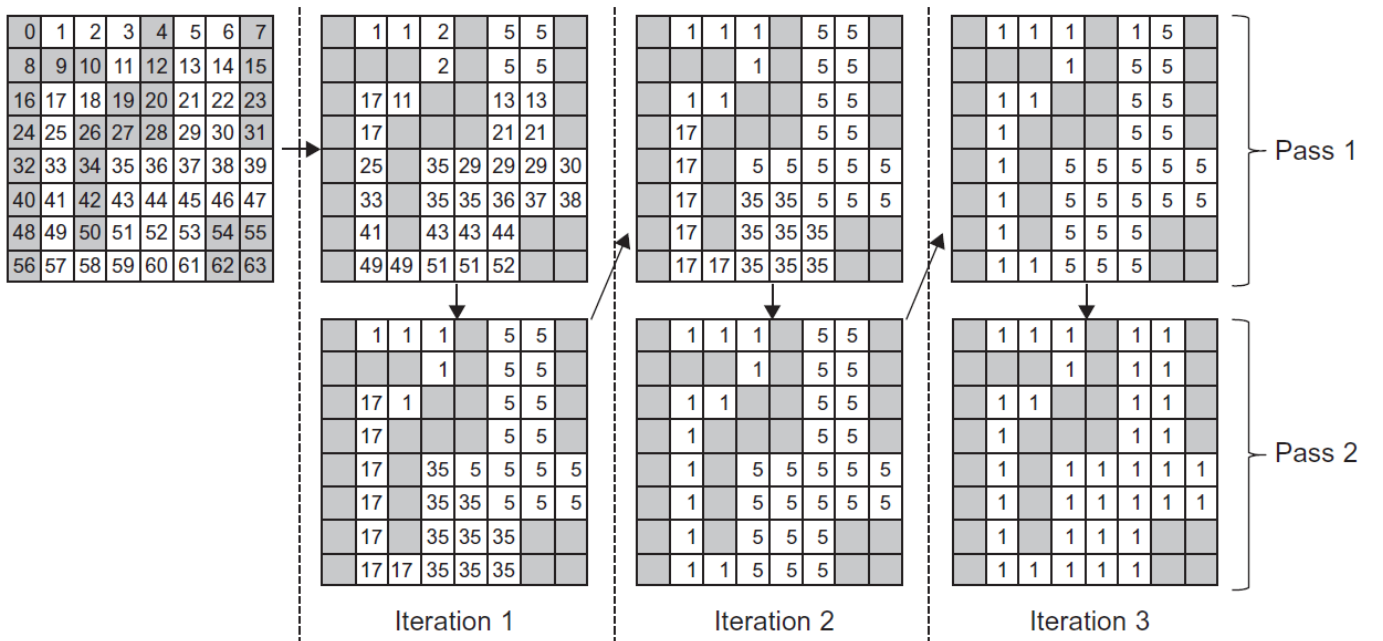


Figure 5-7 Graphical Example of Kernel 1 Operation (Adapted from [34])

To simplify the process of finding the root of a particular element in an equivalence tree (in the 2D label array) all labels are initialised to their array index. After applying the first pass of the CCL algorithm element labels are updated with new labels from their connected labels effectively creating equivalence trees. These equivalence trees are flattened in pass 2 by treating the new element labels as array indexes, if an element's label is equal to its array index then this means it is the root of an equivalence tree, if it

is not then this means that the element label is pointing to another element further down in the equivalence tree. The equivalence tree must then be flattened in order to find the root label, the original element label is then updated with the root label.

5.6.3 Kernel 2: Merge Set of Tiles

Once all the local solutions of the CCL algorithm have been calculated by kernel 1, the merging scheme is initiated. Kernel 2 merges sets of processed tiles into new 'larger' tiles, this kernel is called multiple times until only one single tile exists that includes the entire image area. This approach allows for the merging scheme to be parallelised which is better suited to GPU implementation compared to merging all tiles in a single operation. Listing 5-10 details the pseudocode for kernel 2.

Listing 5-10: Pseudocode for Kernel 2 - Merging Set of Tiles

```

INPUT:  dImgData      //2D array of segmented input data
        dLabelsData   //2D array of labels
        dSubBlockDim  //dimensions of the merged tiles
OUTPUT: dLabelsData   //2D array of labels

subBlockId = thread_index + block_index * block_dim; //ID (x,y) of the merged tile
repetitions = subBlock_dim / block_dim/z; //how many times are the thread reused for
                                           the given subblock?

shared sChanged[1]; //shared memory used to check whether the solution is final or not
while(true) {
    if(thread_index == (0,0,0)) sChanged[0] = 0;
    syncThreads();
    //process the bottom horizontal border
    for(i in 0:repetitions) {
        x = subBlock_id.x * subBlock_dim + thread_index.z + i * blockdim.z;
        y = (subBlock_id.y+1) * subBlock_dim - 1;
        if(!leftBorder) Union(dLabelsData, dImgData, globalAddr(x,y),
                               globalAddr(x-1,y+1), sChanged);
        Union(dLabelsData, dImgData, globalAddr(x,y), globalAddr(x, y+1),
               sChanged);
        if(!rightBorder) Union(dLabelsData, dImgData, globalAddr(x,y),
                               globalAddr(x+1,y+1), sChanged);
    }
    //process the right vertical border
    for(i in 0:repetitions) {
        x = (subBlock_id.x+1) * subBlock_dim - 1;
        y = subBlock_id.y * subBlock_dim + thread_index.z + i * block_dim.z;
        if(!topBorder) Union(dLabelsData, dImgData, globalAddr(x,y),
                               globalAddr(x+1, y-1), sChanged);
        Union(dLabelsData, dImgData, globalAddr(x,y), globalAddr(x+1, y),
               sChanged);
        if(!bottomBorder) Union(dLabelsData, dImgData, globalAddr(x,y),
                               globalAddr(x+1, y+1), sChanged);
    }
    syncThreads();
    if(sChanged[0] == 0) break; //no changes means the tiles are merged
    syncThreads();
}

void Union(equivalenceArray, elementAddress0, elementAddress1){
    root0 = FindRoot(equivalenceArray, elementAddress0);
    root1 = FindRoot(equivalenceArray, elementAddress1);

```

```

//connect an equivalence tree with a higher label to the tree with a lower label
if(root0 < root1) equivalenceArray[root1] = root0;
if(root1 < root0) equivalenceArray[root0] = root1;
}

```

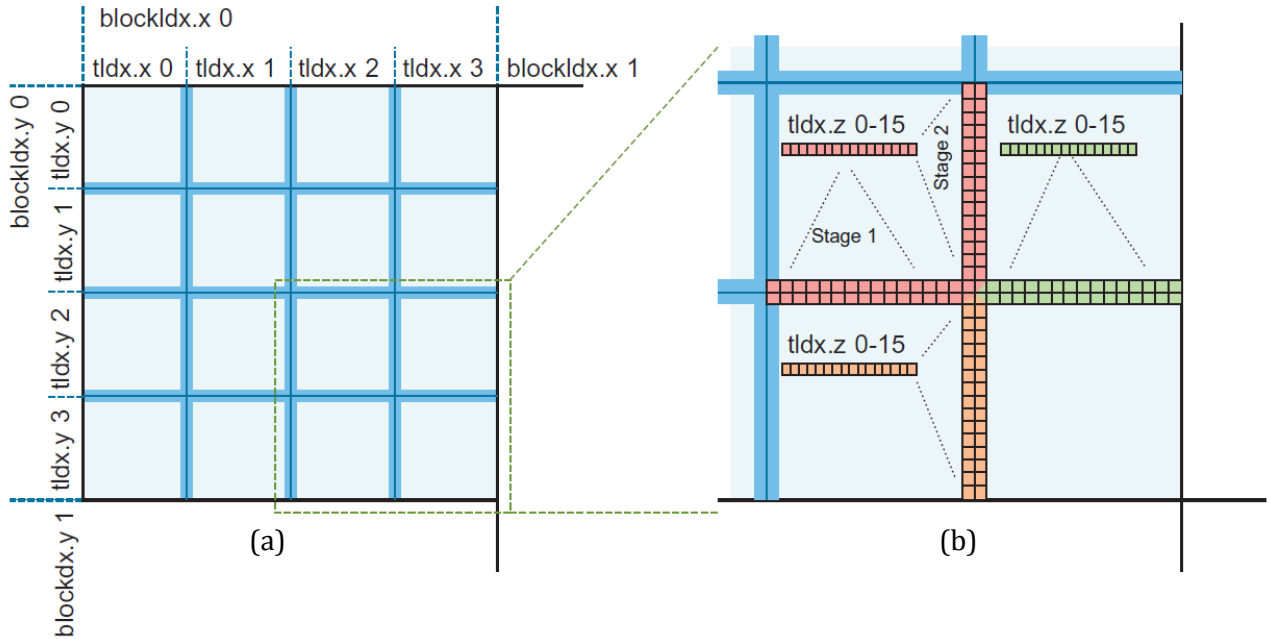


Figure 5-8 Example Application of Kernel 2 [34]

Kernel 2 merges sets of tiles in parallel to produce larger tiles, for the FLORA implementation a set size of 4 X 4 tiles was used, O. Štava found that this set size produced the best performance [34]. Neighbouring tiles are merged by checking the connectivity of pixels at the border of adjacent tiles and updating the equivalence trees of each tile. A set of tiles is processed using a 3D block of CUDA threads where the x and y coordinates of each thread correspond to a particular tile (Figure 5-8a) and the z dimension contains a number of threads used to process a particular tile (Figure 5-8b). In order to ensure the number of threads created do not exceed the maximum number of threads supported by the GPU each thread will process a number of tile boundary elements sequentially. The kernel first processes the bottom boundary of each tile followed by the right boundary, boundaries are processed using the Union function shown in Listing 5-10.

5.6.4 Kernel 3 Updating Tile Border Labels

A problem that arises from merging equivalence trees in kernel 2 is that the depth of equivalence trees on the borders of merged tiles can increase significantly if merging is done many times consecutively. For this reason O. Štava proposed the use of kernel 3 to flatten the equivalence trees of the border elements after application of kernel 2. Although kernel 3 is not necessary to achieve the desired results, its use provides a performance gain between 15-20% [34]. The pseudocode for kernel 3 is shown in Listing 5-11.

Listing 5-11: Pseudocode for Kernel 3 - Update Border Labels

```

INPUT:  dLabelsIn    //2D array of labels

OUTPUT: dLabelsOut    //2D array of labels

blocksPerTile = tileSize / blockWidth; //Calculate the number of thread blocks
                                         per tile

```



```

//multiple thread blocks can be used to update border of a single tile
tileX = block_index.x / blocksPerTile;
tileOffset = block_dim.x*(block_index.x & (blocksPerTile-1));
tileY = threadIdx.y + (block_index.y*block_dim.y);
maxTileY = grid_dim.y*block_dim.y-1;

//a single thread is used to update both the horizontal and the vertical boundary on
both sides of two merged tiles

//first process horizontal borders
if(tileY < maxTileY) {
    y = (tileY+1)*tile_dim-1;
    x = tileX*tile_dim+thread_index.x+tileOffset;
    flattenEquivalenceTree(x, y, dLabelsOut, dLabelsIn);
    flattenEquivalenceTreesl(x, y+1, dLabelsOut, dLabelsIn);
}
//process vertical borders
if(tileX < gridDim.x-1) {
    uint x = (tileX+1)*tileDim-1;
    uint y = tileY*tileDim+threadIdx.x+tileOffset;
    flattenEquivalenceTrees(x, y, dLabelsOut, dLabelsIn);
    flattenEquivalenceTrees(x+1, y, dLabelsOut, dLabelsIn);
}

```

5.6.5 Kernel 4: Update All Labels

Kernel 4 is called once all tiles have been merged, it updates all the labels in the final label array. The pseudocode for kernel 4 is shown in Listing 5-12.

Listing 5-12: Pseudocode for Kernel 4 - Final Label Update

```

INPUT:  dLabelsIn    //2D array of labels

OUTPUT: dLabelsOut   //2D array of labels

//flatten the equivalence trees on all elements
uint x = (block_index.x*block_dim.x)+thread_index.x;
uint y = (block_index.y*block_dim.y)+thread_index.y;
flattenEquivalenceTrees(x, y, dLabelsOut, dLabelsIn);

```

Once kernel 4 completes the final label array is finished, this array contains all connected components in the original input image with each connected region having a unique label.

5.6.6 Scan Final Label Values

The final step in finding the number of connected regions in the image is to check the label array to see if it contains more than one unique label. O. Štava provided a CUDA implementation for scanning the label array in the sample code provided with [34], his implementation was adapted directly from the sample code. Due to time constraints a custom implementation for scanning could not be developed. Once the number of regions in the image is determined it is returned to the host for use.

5.7 Iteration 5: Average Filter (Image blurring)

Due to the large slow down resulting from iteration 4, the CUDA CCL algorithm was disregarded for iteration 5 and instead the FLORA implementation following iteration 3 was used. The next major bottleneck identified after the Find Image Contours step in Table 12-5 is then the Calculate Image

Moments step, however due to time constraints, a GPU accelerated function was not developed for this processing step.

The last bottleneck identified after Image Moment Calculation was Image blurring or the averaging filter. OpenCV provides a CUDA implementation of the 2D averaging filter:

Listing 5-13: CUDA Accelerated Averaging Filter

```
Filter boxFilter = cuda::createBoxFilter(input_array_type, output_array_type, window_size);  
...  
boxFilter->apply(input_image);
```

To use a CUDA box filter a filter object must first be created specifying the element types of the input and output arrays and the window size to be used. For the FLORA implementation greyscale images are used which are 8 bit unsigned values and the window size is initialised to 2 X 2. The window size is increased after every iteration of the function. Figure 5-9 shows an example of the output of the CUDA accelerated Image Blurring function.

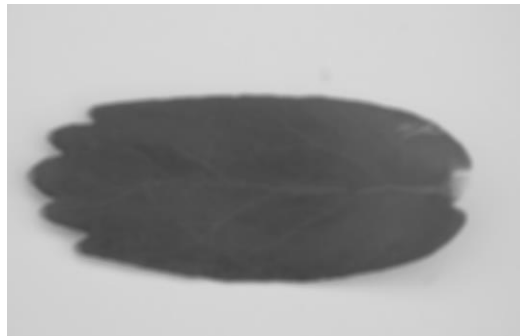


Figure 5-9 Example of Image blurred using CUDA Accelerated Blur Function

5.8 Termination of Iteration

Following iteration 5, with the exception of calculation of the image moments, the only major bottleneck that existed was the loading of the pre-trained classifier during KNN prediction (Shown in Table 11-6). This function could not be optimised in any way. All other processing steps in the FLORA implementation took less than 10ms to complete and therefore further GPU acceleration was deemed unnecessary as GPU overheads would prevent any speed up of these functions.

6. Integration with FLORA Front End

The C++ and CUDA accelerated implementations of the FLORA backend system are required to work with the current web based FLORA front end. The original FLORA system worked by creating and executing an Octave script each time an image was submitted via the FLORA html home page. This script was then run by calling Octave and passing it the Octave script, a results string would be returned which the front end system would then interpret. Listing 6-1 shows the original PHP code used to call the original FLORA backend.

Listing 6-1: PHP Script for Calling Original FLORA Backend

```

$octave_command = "Generated octave script for processing input image";

//Write the script to a file
file_put_contents($octave_file, $octave_command);
//Call Octave and pass the script file as an argument
$octave_command_output = exec("octave-cli.exe -q --path $octave_path --image-path $image_path $octave_file");

//Return the result string
return $octave_command_output;

```

To ensure compatibility with the original FLORA front end the C++ and CUDA accelerated implementations have been developed to be called in a similar manner. When the new FLORA backend code is compiled, an executable called 'KNN.exe' will be produced. This single executable can be called by the FLORA front end with the '-x' flag followed by the input image path in order to generate the desired return string. Listing 6-2 shows how the C++ and/or CUDA accelerated implementations of the FLORA backend is called by the existing FLORA frontend.

Listing 6-2: PHP Script for Calling New FLORA Backend

```

//Call KNN.exe and pass the input image path as an argument
$sknn_output = exec("KNN.exe -x $image_path");

//Return the result string
return $sknn_output;

```

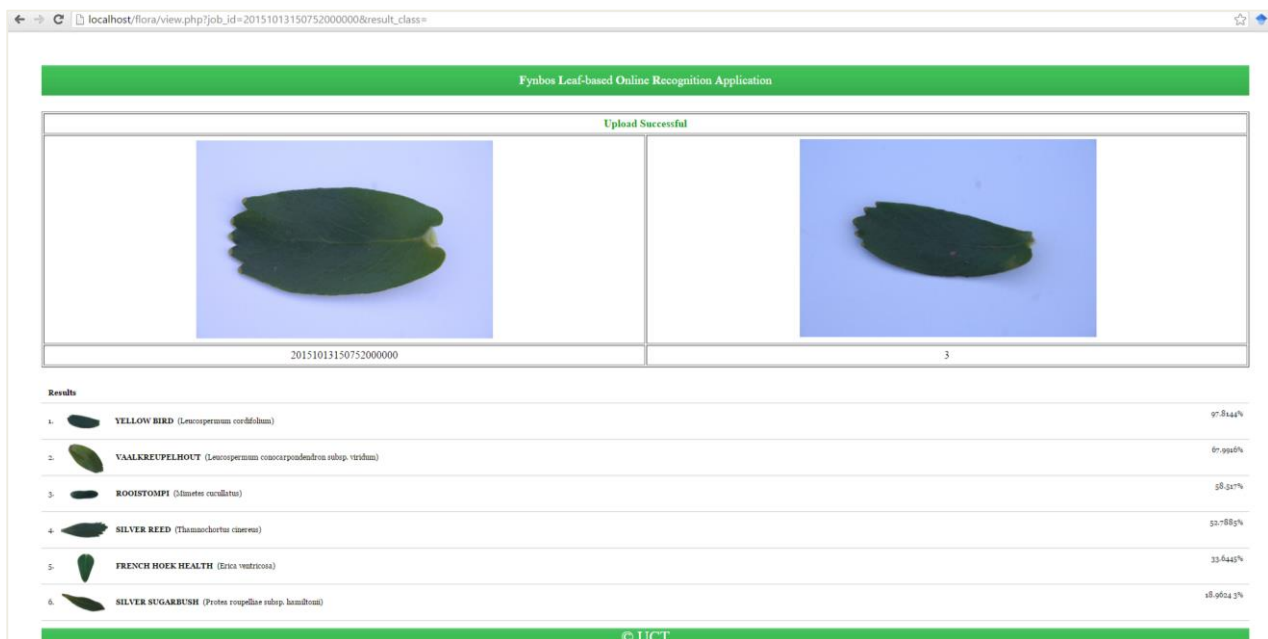


Figure 6-1 Example of FLORA Web App Running the New Backend System

Figure 6-1 shows a successful test of the FLORA web app running with the C++ backend. The same result is achieved with all of the CUDA accelerated implementations.

7. Performance Results

7.1 Overview

This section details the performance results obtained after successful implementation and performance testing of all FLORA implementations. Each implementation is tested for execution time and prediction accuracy. The execution time speedups are also calculated by comparing the new implementation execution times with that of the original FLORA implementation. The raw performance data can be found in Appendix A. Note only the performance results of the original FLORA Octave implementation, the un-accelerated C++ implementation and the 4 CUDA accelerated implementations created during GPU design iteration are shown in Appendix A. All performance data in Appendix A also excludes OpenCV initialisation overhead.

7.2 Analysis of Original FLORA Implementation

The execution timing of the original FLORA implementation was tested using Octave timing functions and prediction accuracy was tested using an Octave script implementing the testing methodology explained in section 3.7.1. Refer to Table 12-1 in Appendix A for the exact execution times recorded.

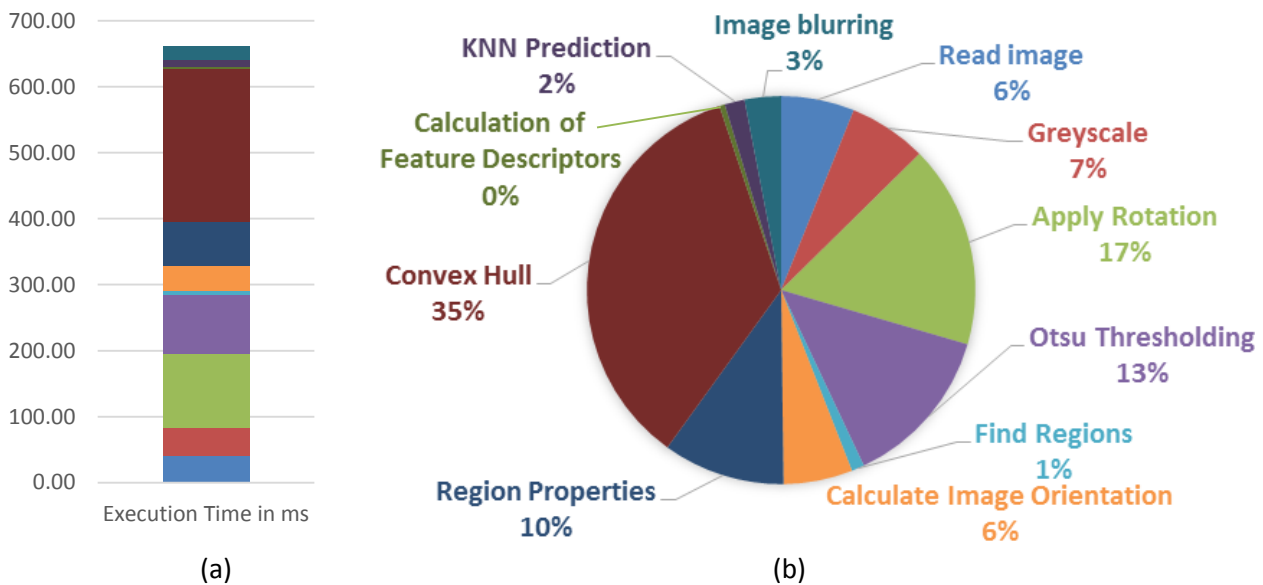


Figure 7-1 Execution Time of Original FLORA Implementation

Figure 7-1(a) shows the cumulative execution times of each processing function used in the original Octave FLORA implementation. The relative contribution of each processing step is also shown in Figure 7-1(b). The total execution time for this implementation was 660ms on average. The most time consuming function was also found to be the 'Convex Hull' function, this function was responsible for calculation of the region convex hull.

The average prediction accuracy was calculated to be 89.74%, the same result obtained by Katz in [6]. These results were used as the basis for speedup calculations and accuracy comparisons for the new FLORA implementations.

7.3 Analysis of Optimised C++ Implementation

The execution time of the C++ implementation of FLORA was testing using the code profiler built into Microsoft Visual Studio 2013. The exact execution times for this implementation can be found in Table

12-2 in Appendix A. Prediction accuracy was tested using the custom ‘-test’ function explained in section 4.16.3 and using the methodology explained in section 3.7.1.

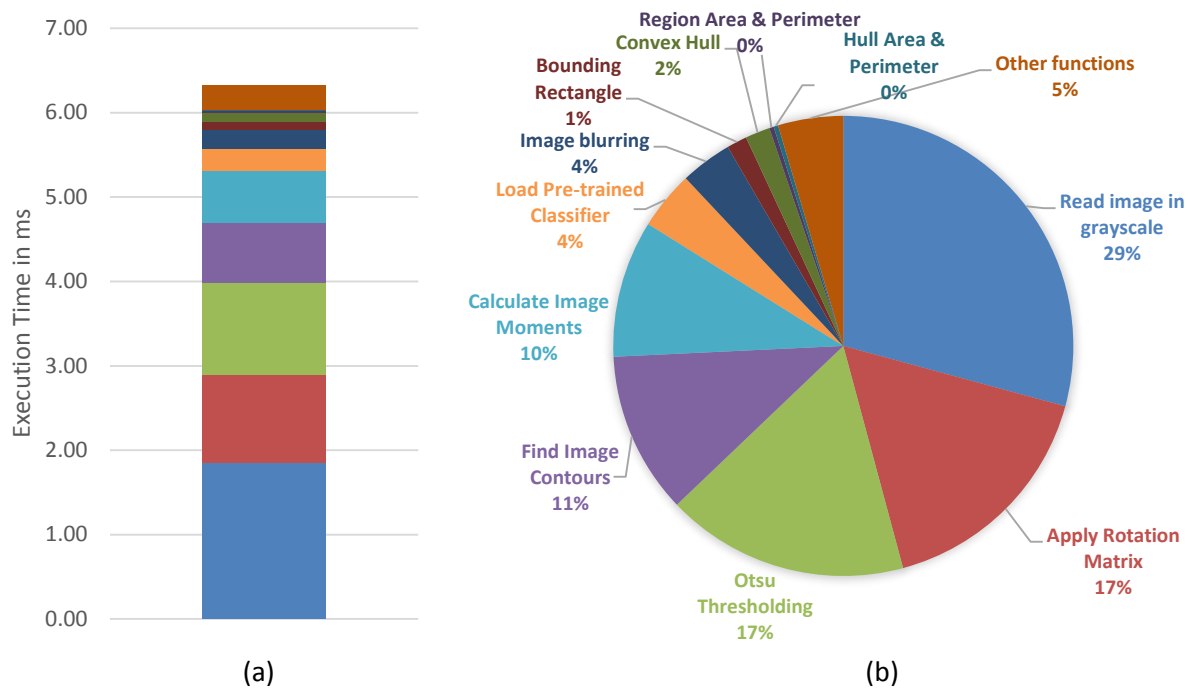


Figure 7-2 Execution Timing of C++ Implementation (Excluding Initialisation Overhead)

Figure 7-2(a) shows the cumulative execution times for the functions used in the C++ implementation. The total execution time was 6.33ms on average excluding the 250ms OpenCV initialisation overhead. Figure 7-2(b) shows the relative contribution of each function to the total execution time. As can be seen from the pie chart the reading of the image in greyscale had the biggest impact on execution time followed by the application of the rotation matrix and Otsu thresholding.

It should be noted that functions which had an execution time of less than 1ms could not be picked up by the profiler and therefore Table 12-2 does not show all the C++ functions that were actually implemented (Described in section 4). These functions along with other miscellaneous IO and C++ clean-up operations are grouped under ‘Other functions’.

Running the accuracy test on this implementation yielded an average prediction accuracy of 89.23%

Table 7-1 Function Equivalences between Octave and C++ Implementations

Octave Function	C++ Function
Greyscaling	Load Image in Greyscale
Read Image	
Calculate Image Orientation	
Otsu Thresholding	
Find Regions	Find Image Contours
Region Properties	
Apply Rotation	Calculate Rotation Matrix
	Apply Rotation Matrix
Convex Hull	Convex Hull
	Hull Area & Perimeter Length
Calculation of Feature Descriptors	
KNN Prediction	
Image Blurring	

7.3.1 Comparison with Original FLORA Implementation

The C++ implementation execution performance was compared with the original FLORA implementation's execution performance in order to determine the speedup achieved. Before a comparison could be made however, it was necessary to determine the functional equivalences between the Octave and C++ implementations.

Table 6-1 shows the relevant functions of the original Octave implementation and the C++ implementation of FLORA, functions in the same row are considered functionally equivalent. As can be seen from the table, the 'Otsu Thresholding', 'Calculate Image Orientation', 'KNN Prediction' and 'Image Blurring' functions are implemented similarly in both the Octave and C++ implementations. The Octave 'Greyscaling' and 'Image Reading' functions have been combined into one function in the C++ implementation, so too have the 'Find Regions' and 'Region Properties' functions. Conversely the 'Apply Rotation' and 'Convex Hull' Octave functions have been split into four separate functions in the C++ implementation.

The equivalences shown in Table 6-1 are not 100% accurate however they are similar enough to make performance comparison valid. Note that some C++ functions shown in Table 6-1 do not appear in the execution timing data in Table 11-2, as stated previously this means that these functions had an execution time of less than 1ms. Since these functions have negligible execution time their effective execution times will be treated as if they are 0. Only the functions listed in bold will be referred to for both the C++ and Octave implementations when showing comparisons.

Table 7-2 Speedup Achieved by C++ Implementation compared to Octave Implementation

Processing Steps	Speedup Achieved
Feature Descriptor Calculation	3000.00+
Convex Hull	2096.36
Apply Rotation	106.30
Find Image Contours	102.88
Image Blurring	86.96
Otsu Thresholding	82.50
Image Orientation	62.36
Load image in greyscale	45.08
KNN Prediction	42.38
Total Functional Speedup (Excluding OpenCV overhead)	104.27
Total Program Speedup (Including OpenCV Overhead)	2.57

Table 6-2 shows the speedup achieved by the FLORA C++ implementation compared to the original Octave implementation. Each equivalent function implementation is compared individually and the total program execution times are also compared. The functions listed in Table 6-2 are combinations of the actual Octave and C++ functions implemented as shown in Table 6-1.

Comparing the function execution times alone shows that a speedup of over 100 times had been achieved, however when the OpenCV initialisation overhead is included the speedup is reduced to about 2.5.

In terms of prediction accuracy the C++ implementation reduced prediction accuracy by an average of 0.51% compared to the Octave implementation. This is an extremely minor loss of prediction accuracy and therefore the prediction accuracy of the C++ implementation can effectively be considered to be the same as the Octave implementation.

7.4 Analysis of CUDA Accelerated Functions

GPU acceleration of the FLORA system was achieved by accelerating specific functions in the C++ implementation that were identified to be bottlenecks. The timing data for the FLORA implementations with CUDA acceleration can be found in section 12.1.3 in Appendix A.

Table 7-3 Specific Function Speedup of CUDA Accelerated Functions (Excluding Overheads)

CUDA Function	Execution Time (CUDA Accelerated)	Execution Time (Octave)	Function Speedup
1. Apply Rotation	0.59ms	111.62ms	189.19
2. Otsu Thresholding	0.98ms	89.10ms	90.92
3. Find Image Contours	12.06ms	74.07ms	6.14
4. Image Blurring	0.56ms	20.00ms	35.71

Table 6-3 shows the functional speedups achieved by each CUDA accelerated function compared to their Octave equivalents, these speedups were calculated ignoring GPU memory overheads and OpenCV library initialisation delay. All the CUDA functions except the 'Find Image Contours' function achieved a speedup that was better than or similar to the C++ speedup achieved (excluding overheads). The CUDA accelerated 'Find Image Contours' function was significantly slower than it's C++ counterpart.

Table 7-4 Performance Optimisation for CUDA Implementations

CUDA Function				Prediction Accuracy	Overall speedup including GPU memory overhead	Overall Speedup including memory and initialization overhead
1	2	3	4			
X				89.23%	87.88	0.66
X	X			90.77%	77.74	0.65
X		X		84.00%	24.69	0.64
X			X	89.23%	86.15	0.66
X	X		X	89.23%	73.01	0.65
X		X	X	84.00%	22.14	0.64
X	X	X		85.21%	23.91	0.64
X	X	X	X	84.00%	20.62	0.64
	X			90.77%	80.64	0.65
	X	X		84.00%	21.76	0.64
	X		X	89.23%	78.46	0.65
	X	X	X	84.00%	21.15	0.64
		X		84.00%	22.16	0.64
		X	X	84.00%	20.84	0.64
			X	89.23%	85.69	0.66

In order to determine the best CUDA accelerated implementation for the final FLORA system it was required to determine which set of CUDA accelerated functions would produce the best performance in

terms of execution time and prediction accuracy. To achieve this all possible combinations of CUDA accelerated functions were tested as shown in Table 6-4. The CUDA functions are numbered from 1 to 4 as shown in Table 6-3. Each combination of CUDA functions was implemented and tested against the original Octave implementation in order to determine what speedup could be achieved.

Two speedup values were calculated, one including GPU memory overheads and one including GPU memory overheads and CUDA library initialisation. The GPU memory overheads include time taken to upload and download data from the GPU and time taken to allocate and deallocate data structures on the GPU. The CUDA library initialisation overhead is explained in section 5.2, it is always a constant value of about 1 second.

The results of the CUDA performance optimisation show that no implementation could match the standard C++ implementation in terms of execution time. Even though the ‘Apply Rotation’ and ‘Otsu Thresholding’ CUDA accelerated functions were faster on their own, the GPU memory overhead cancelled out any performance gains. The CUDA accelerated implementation with just the ‘Otsu Thresholding’ function accelerated was considered the best performing CUDA implementation since it improved prediction accuracy and was only slightly slower than the fastest performing CUDA implementation.

7.5 Response to Variation of Image Resolution

As an additional experiment each FLORA implementation was tested using the original high resolution input images from the testing database. These images are about 4 times larger than the images used previously. To ensure a fair accuracy comparison the KNN classifier was retrained using the larger sized images before testing was initiated. The execution timing data for high resolution input can be found in section 12.2 in Appendix A. Note the timings used in this section are excluding OpenCV initialisation overhead only.

Only two CUDA accelerated implementations were chosen for comparison referred to as ‘CUDA 1’ and ‘CUDA 2’ respectively. For CUDA 1 just ‘Otsu Thresholding’ function was accelerated because determined as the most effective solution for low resolution images and in CUDA 2 all CUDA accelerated functions were implemented except for ‘Find Image Contours’ which was the worst performing CUDA function and did not provide a performance improvement even with high resolution image inputs.

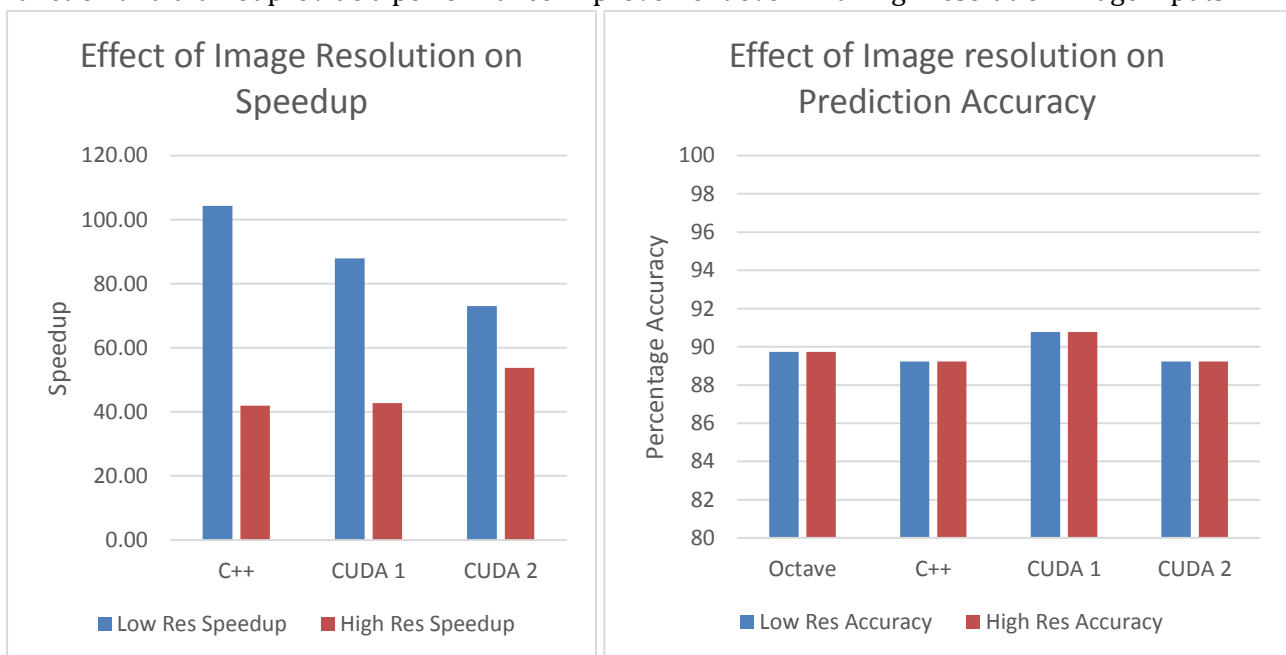


Figure 7-3 Effect of Image Resolution on Speedup & Prediction Accuracy

Figure 7-3 shows how increasing image resolution affects implementation speedup and prediction accuracy, Table 7-5 additionally shows the exact overall execution times that were observed during testing.

Table 7-5 Effect of Image Resolution on Speedup & Prediction Accuracy

FLORA implementation	Low Res Execution Time	Low Res Speedup	High Res Execution time	High Res Speedup	High Res Accuracy
Octave	660.00	1.00	8182.90	1.00	89.74
C++	6.33	104.27	195.05	41.95	89.23
CUDA 1	7.51	87.88	191.51	42.73	90.77
CUDA 2	9.04	73.01	152.27	53.74	89.23

The results of the image resolution variation show that for lower resolution images any GPU acceleration results in an increase in overall execution time, this can be seen in the fact that the unaccelerated C++ implementation performed the best while the ‘CUDA 2’ implementation which had 4 functions accelerated on the GPU performed the slowest. ‘CUDA 1’ with 1 accelerated function achieving results in between these 2. On the other hand the high resolution test results showed that the more functions accelerated effectively on the GPU the greater the speedup achieved. In the high resolution test ‘CUDA 2’ had the greatest speedup while the accelerated C++ implementation performed the worst. The prediction accuracy test showed that increased image resolution has no effect on prediction accuracy.

8. Discussion

8.1 Execution Time Performance

The performance results show that the best overall performance was achieved by the unaccelerated C++ implementation. With all overheads included this implementation managed to achieve a speedup of 2.57 times. All CUDA accelerated functions on the other hand achieved a slowdown of about 1.5 with all overheads taken into account. These speedups may seem underwhelming but the poor performance is mainly due to the OpenCV initialisation overhead which increases execution time by around 250ms for the C++ implementation and around 1.25s for the CUDA accelerated implementations. This is quite a significant delay, amounting to about 96% of the total execution time for the C++ implementation and 99% of the total execution time for all the CUDA accelerated implementations. This initialisation delay can however be avoided by modifying the new FLORA implementations into persistent system processes or services that run continuously in the background of the server. The OpenCV initialisation could take place upon initiation of the service and the service could then be used repeatedly to do multiple predictions without needing reinitialisation. Unfortunately due to time constraints these modifications could not be made but research shows that there are several possible ways of implementing this, including Windows Services [53], Linux Daemons [54] and persistent processes with interprocess communication [55].

By excluding the OpenCV initialisation overhead the true performance potential of the new FLORA implementations could be analysed. The unaccelerated C++ implementation achieved a speedup of 104 with OpenCV initialisation overhead excluded, again this implementation proved to perform the best compared to all other implementations with CUDA acceleration. All 4 functions that were given CUDA accelerated implementations performed slower than their standard C++ counterparts even with OpenCV initialisation excluded. This reinforces the point made by Vuduc et al in [24] that CPU optimisation is often more effective than GPU acceleration. The speedups achieved are however inline

or greater than with the speedups achieved by K. Chen in [11], Singh et al in [8] and R. Fernando in [33] for the 'Apply Rotation', 'Otsu Thresholding' and 'Image Blurring' functions respectively. Only the 'Find Regions' did not achieve a similar speedup to O. Stava in [34], this function only managed to achieve half the expected speedup. Unfortunately unlike the OpenCV initialisation overhead the GPU memory overheads cannot be avoided easily, attempting doing so would effectively mean developing entirely new function implementations. Analysing the CUDA accelerated functions with memory overhead excluded did however show that the 'Otsu Thresholding' and 'Apply Rotation' functions did perform much better than their unaccelerated counterparts. The CUDA 'Image Blurring' function had similar performance to the unaccelerated version while the 'Find Regions' function was the only CUDA accelerated function that showed significant slowdown in all aspects.

Despite the poor performance of the CUDA accelerated implementations compared to their standard C++ counterparts it was found that if high resolution input images are used rather than the expected low resolution input images, the CUDA accelerated implementations actually proved to outperform the C++ implementation. With a high resolution input all CUDA accelerated functions except for the 'Find Regions' function outperformed their C++ counterparts. With the 3 best performing CUDA accelerated functions implemented a speedup of 53.74 was achieved compared to 41.95 for the C++ implementation. This is expected due to the highly parallel nature of the GPU which can process more of the image at a time across its thousands of CUDA cores compared to the CPU which can only has 4 cores to split the workload. Although the current FLORA implementations do not benefit from better high resolution performance the CUDA accelerated may be useful for future work if for example one wants to add additional feature descriptors that require higher resolution images.

8.2 Prediction Accuracy

The prediction accuracy of all the FLORA implementations including the original Octave implementation were very similar, each having an average accuracy of around 90% which can be considered acceptable. It was found however that the CUDA accelerated implementation of 'Otsu Thresholding' seemed to improve the prediction accuracy by about 1.50% in most cases while the CUDA accelerated 'Find Regions' reduced the prediction accuracy by about 5%. It was also found that larger input image resolution had no effect on accuracy.

8.3 Other Advantages of New FLORA implementations

Apart from improving the execution time of the FLORA backend, the new C++ and CUDA accelerated FLORA implementations have several other advantages compared to the original FLORA system. One of the main advantages is that all code has been developed to be easy to understand and modify. The code has been split into modules and commented thoroughly to ensure future developers do not have trouble attempting to build on the implementations developed in this project.

The code has also been developed to be as portable and easy to use as possible. The C++ and CUDA accelerated implementations are all compiled to single executable files that can be run directly from the command line or can be plugged in directly to the FLORA front end. Once the 'KNN.exe' executable is compiled on the host system it requires no other dependency to run, unlike the original backend which requires Octave to be installed, this makes the new FLORA backend much easier to deploy. Also the original backend essentially cannot be used without the front end as the front end is required to generate the Octave script to pass to Octave. The new FLORA implementation also includes the functionality to easily retrain and test its KNN classifier using simple command line options, this makes updating the FLORA leaf database easy.

9. Conclusions

The objective of this project was to design and develop a GPU accelerated implementation of the FLORA web application developed by Shaun Katz in [6]. The purpose was to try and improve the execution time of the FLORA backend system which had unacceptable processing latency. To solve this problem the FLORA backend was analysed and re-implemented in C++ using a combination of OpenCV libraries and custom code.

The C++ implementation formed the foundation on which the GPU accelerated implementation could be built. CUDA was chosen as the GPU acceleration platform and research was done to determine how best to accelerate the processing steps used in the existing FLORA system. An iterative design approach was then used to identify functions that should be accelerated to improve performance.

Four functions were identified as major bottlenecks suitable for GPU acceleration: Image rotation, Otsu thresholding, connected component labelling and image average filtering. Each of these functions were then implemented in CUDA.

Following successful creation of the FLORA C++ implementation and CUDA accelerated functions, the system was tested and benchmarked to determine its execution speed performance as well as its prediction accuracy. The CUDA accelerated functions were tested in different combinations in order to determine what the best CUDA accelerated implementation would be. Results showed that the un-accelerated C++ implementation performed the best with a speedup of 2.5 compared to the original FLORA implementation. Implementation of any of the CUDA accelerated functions did not achieve any speedup, implementation of any combination of the CUDA accelerated functions resulted in a slowdown of 1.5 times.

It was proven however that with some modifications to the C++ implementation a speedup of 104 times could be achieved with the un-accelerated C++ implementation and a maximum speedup of 87 times could be achieved through CUDA acceleration using expected input. Analysis of the individual CUDA accelerated functions showed that only the 'connected component labelling' function could not achieve the expected speedup. The new FLORA implementations were also tested with high resolution input. It was found that the optimal CUDA implementation could achieve a speedup of 53.74 while the un-accelerated C++ implementation could only achieve a speedup of 42 compared to the original FLORA implementation when using high resolution input.

The prediction accuracy of all FLORA implementations including the original were similar at around 90% regardless of input image resolution. Only the CUDA accelerated 'connected component labelling' function was observed to significantly reduce the prediction accuracy.

Finally the C++ and optimised CUDA accelerated FLORA implementations were integrated with the FLORA front end and confirmed to work correctly giving the expected speedups.

In its current state the FLORA C++ implementation did not meet the objective of achieving a speedup of 10 times the original implementation, however this project can be considered a partial success since a speedup of 2.5 was achieved and it has been shown that with a modification the FLORA implementation developed could achieve a speedup of over 100 times.

10. Recommendations

In order to achieve the best possible execution time performance from the FLORA system in future it is recommended that the C++ implementation developed in this project be modified into a persistent service such as a Windows Service or Linux Daemon in order to remove the major performance bottleneck caused by the OpenCV image processing library initialisation. Once that is achieved the performance of the FLORA system should be good enough for real world deployment.

Due to time constraints a CUDA accelerated implementation of the 'Calculation of Image Moments' function could not be attempted. This function still remains a bottleneck in the system and therefore developing a GPU accelerated version of this function could further improve the GPU implementation performance. Additionally it was found that the CUDA accelerated 'Find Regions' function performed poorly, re-implementing this function using another approach may also improve the GPU implementation performance. This is only recommended if one wishes to use high resolution inputs, otherwise the un-accelerated C++ version should just be used.

Since the new FLORA implementation has been designed to work independently from the FLORA front end system one could try testing the backend system's predicative accuracy in other areas apart from leaf recognition. For example one could test the system's ability to identify flowers based on their shape.

The current FLORA front end has limited functionality in that it does not allow one to upload images of new and existing plant species. The front end could be modified to allow users to upload new images by either making use of the built in '-train' training functionality of the new FLORA backend system or by directly modifying the 'knn_classifier.yaml' classifier database. Including this functionality should help to grow the FLORA database.

One of the major limitations of the current FLORA implementation is that the leaf images must have a white background in order for the system to properly recognise them. One possible approach to solve this problem would be to write a mobile frontend app that could use a phones camera to Otsu threshold the input image in real time so that the user can see if the features will be extracted properly before they take a picture to upload. This way a user could simply use their hand as a background for the leaf and still ensure that a good prediction can be made.

11. List of References

- [1] M. Harris, "GPGPU: General -Purpose Computation on GPU," in *Game Developers Conference*, 2005.
- [2] J. P. Farrugia, P. Horain, E. Guehenneux, and Y. Alusse, "GPUCV: A framework for image processing acceleration with graphics processors," *2006 IEEE Int. Conf. Multimed. Expo, ICME 2006 - Proc.*, vol. 2006, pp. 585–588, 2006.
- [3] A. Stein, E. Geva, and J. El-Sana, "CudaHull: Fast parallel 3D convex hull on the GPU," *Comput. Graph.*, vol. 36, no. 4, pp. 265–271, 2012.
- [4] J. Kim, E. Park, X. Cui, W. a Gruver, and H. Kim, "A fast feature extraction in object recognition using parallel processing on CPU and GPU," *Syst. Man Cybern. 2009. SMC 2009. IEEE Int. Conf.*, no. October, pp. 3842–3847, 2009.

- [5] S. Tzeng and J. Owens, "Finding convex hulls using Quickhull on the GPU," *arXiv Prepr. arXiv1201.2936*, pp. 1–11, 2012.
- [6] S. Katz, "Fynbos Leaf-based Online Recognition Application," no. October, 2011.
- [7] J. Zhang and J. Hu, "Image Segmentation Based on 2D Otsu Method with Histogram Analysis," *2008 Int. Conf. Comput. Sci. Softw. Eng.*, no. 1, pp. 105–108, 2008.
- [8] B. M. Singh, R. Sharma, A. Mittal, and D. Ghosh, "Parallel Implementation of Otsu's Binarization Approach on GPU," *Int. J. Comput. Appl.*, vol. 32, no. 2, 2011.
- [9] University of Southern California, "The USC-SIPI Image Database," 2015. .
- [10] R. M. Haralick and L. G. Shapiro, *Computer and Robot Vision*, vol. 1. 1992.
- [11] K. Chen, "Efficient parallel algorithms for the computation of two-dimensional image moments," *Pattern Recognit.*, vol. 23, no. 1–2, pp. 109–119, Jan. 1990.
- [12] M. S. Nixon and A. S. Aguado, *Feature Extraction & Image Processing for Computer Vision*. Elsevier, 2012.
- [13] S. G. Wu, F. S. Bao, E. Y. Xu, Y. Wang, Y. Chang, and Q. Xiang, "A Leaf Recognition Algorithm for Plant Classification Using Probabilistic Neural Network," in *International Symposium on Signal Processing and Information Technology*, 2007, pp. 11–16.
- [14] D. Q. Nykamp, "Using Green's theorem to find area," 2015. .
- [15] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer Berlin Heidelberg, 2008.
- [16] M. Hayworth, "Filling concave regions on an image," *Matlab Newsgroup*, 2010. [Online]. Available: http://www.mathworks.com/matlabcentral/newsreader/view_thread/294624. [Accessed: 19-Sep-2015].
- [17] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Inf. Process. Lett.*, vol. 1, no. 4, pp. 132–133, 1972.
- [18] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software*, vol. 22, no. 4, pp. 469–483, 1996.
- [19] Octave-Forge, "Octave-Forge Image Documentation," 2015. [Online]. Available: <http://octave.sourceforge.net/image/overview.html>. [Accessed: 19-Sep-2015].
- [20] J.-X. Du, X.-F. Wang, and G.-J. Zhang, "Leaf shape based plant species recognition," *Appl. Math. Comput.*, vol. 185, no. 2, pp. 883–893, 2007.
- [21] A. G. Ghuneim, "Contour Tracing Algorithms," 2014. .
- [22] S. Sinha, "Leaf shape recognition via support vector machines with edit distance kernels," 2004.
- [23] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

- [24] R. Vuduc, A. Chandramowliswaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of GPU acceleration," *Proc. 2nd USENIX Conf. Hot Top. parallelism*, p. 13, 2010.
- [25] J. Esselaar and S. Winberg, "Fynbos Leaf-based Online Recongition," University of Cape Town, 2012.
- [26] M. A. Ramone, "FLORA – Fynbos Leaf Online Recognition Application," University of Cape Town, 2013.
- [27] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - A case study," *ICASSP, IEEE Int. Conf. Acoust. Speech Signal Process. - Proc.*, pp. 2629–2633, 2013.
- [28] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. 2011.
- [29] S. Cook, "CUDA Programming," *CUDA Program.*, no. June, pp. 441–502, 2013.
- [30] AMD, "AMD APP SDK."
- [31] Itseez, "OpenCV." .
- [32] Itseez, "OpenCV 3.0.0 Documentation," 2015. [Online]. Available: <http://docs.opencv.org/master/>. [Accessed: 20-Sep-2015].
- [33] R. Fernando, *GPU Gems*, 1st ed. Addison-Wesley Professional, 2004.
- [34] O. Štava and B. Beneš, *GPU Computing Gems Emerald Edition*. Elsevier, 2011.
- [35] V. M. A. Oliveira and R. A. Lotufo, "A Study on Connected Components Labeling algorithms using GPUs."
- [36] C. E. Leiserson, R. L. Rivest, C. Stein, and T. H. Cormen, *Introduction to algorithms*, 2nd ed., vol. 5. Cambridge: The MIT press, 2001.
- [37] C. (University of V. Woolley, "GPU Gems 2," in *GPU Gems 2*, vol. 2, 2005.
- [38] R. Hochberg, "Matrix Multiplication with CUDA — A basic introduction to the CUDA programming model," 2012.
- [39] A. Chrzesczyk and J. Chrzesczyk, "Matrix computations on the GPU," 2013.
- [40] BLAS, "Basic Linear Algebra Subprograms (BLAS) FAQ," *Netlib Repository*, 2005. .
- [41] S. Srungarapu, D. P. Reddy, K. Kothapalli, and P. J. Narayanan, "Fast two dimensional convex hull on the GPU," *Proc. - 25th IEEE Int. Conf. Adv. Inf. Netw. Appl. Work. WAINA 2011*, pp. 7–12, 2011.
- [42] B. Barber and H. Huhdanpaa, "Qhull," *Geom. Center, Univ. Minnesota*, <http://www.geom.umn.edu/software/qhull>, 1995.
- [43] S. Liang, Y. Liu, C. Wang, and L. Jian, "Design and evaluation of a parallel K-nearest neighbor algorithm on CUDA-enabled GPU," *Proc. - 2010 IEEE 2nd Symp. Web Soc. SWS 2010*, pp. 53–60, 2010.

- [44] S. Liang, Y. Liu, C. Wang, and L. Jian, "A CUDA-based parallel implementation of K-nearest neighbor algorithm," *2009 Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discov.*, 2009.
- [45] S. Liang, C. Wang, Y. Liu, and L. Jian, "CUKNN: A parallel implementation of K-nearest neighbor on CUDA-enabled GPU," *Proc. - 2009 IEEE Youth Conf. Information, Comput. Telecommun. YC-ICT2009*, pp. 415–418, 2009.
- [46] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud, "K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching," *Proc. - Int. Conf. Image Process. ICIP*, pp. 3757–3760, 2010.
- [47] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using GPU," *2008 IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, no. 2, pp. 1–6, 2008.
- [48] B. Stroustrup, *The C++ Programming Language*, vol. 923. 1997.
- [49] Itseez, "Performance Measurement and Improvement Techniques," *OpenCV 3.0.0 Documentation*, 2015. [Online]. Available: http://docs.opencv.org/master/dc/d71/tutorial_py_optimization.html#gsc.tab=0. [Accessed: 07-Oct-2015].
- [50] S. Suzuki and K. be, "Topological structural analysis of digitized binary images by border following," *Comput. Vision, Graph. Image Process.*, vol. 30, no. 1, pp. 32–46, Apr. 1985.
- [51] J. Sklansky, "Finding the convex hull of a simple polygon," *Pattern Recognit. Lett.*, vol. 1, no. 2, pp. 79–83, Dec. 1982.
- [52] O. Ben-Kiki, C. Evans, and B. Ingerson, "YAML Ain't Markup Language (YAML™) Version 1.2," *Language (Baltim.)*, pp. 1–100, 2009.
- [53] Microsoft, "Introduction to Windows Service Applications," 2015. [Online]. Available: [https://msdn.microsoft.com/en-us/library/d56de412\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/d56de412(v=vs.110).aspx).
- [54] Linfo, "Daemon Definition," *The Linux Information Project*, 2005. [Online]. Available: <http://www.lininfo.org/daemon.html>.
- [55] Microsoft, "Interprocess Communications," 2015. [Online]. Available: [https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx).

12. Appendix A: Code Performance Data

This section details all the execution time profiling and accuracy testing results for all implementations of the FLORA backend used in this project. All implementations were tested using both high resolution input images and low resolution input images. Note that all results exclude OpenCV initialisation time.

12.1 Execution Time Profiling Results

12.1.1 Original FLORA Octave Implementation

Table 12-1 Execution Time of Original FLORA Implementation for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
Convex Hull (Calculation, Area & Perimeter)	230.60	34.89%
Apply Rotation	111.62	16.89%
Otsu Thresholding	89.10 (2 iterations)	13.48%
Region Properties (Area & Perimeter)	67.07	10.15%
Greyscale	43.08	6.52%
Read image	40.31	6.10%
Calculate Image Orientation	38.04	5.76%
Image blurring	20.00	3.03%
KNN Prediction	11.02	1.67%
Find Regions (CCL)	7.00	1.06%
Calculation of Feature Descriptors	3.00	0.45%

Total execution time 660.00

Accuracy 89.74%

12.1.2 FLORA C++ Implementation

Table 12-2 Execution Time of FLORA before GPU Design Iteration 1 and 2 for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
Read image in greyscale	1.85	29.23%
Otsu Thresholding	1.08 (2 iterations)	17.06%
Apply Rotation Matrix	1.05	16.59%
Find Image Contours	0.72 (2 iterations)	11.37%
Calculate Image Moments	0.61	9.64%
Load Pre-trained Classifier (KNN Prediction)	0.26	4.11%
Image blurring	0.23	3.63%
Convex Hull	0.11	1.74%
Bounding Rectangle	0.09	1.42%
Region Area & Perimeter	0.02	0.32%

Hull Area & Perimeter	0.02	0.32%
Other functions < 1ms each	0.29	4.58%

Total execution time: 6.33ms excluding OpenCV initialisation

Accuracy 89.23%

12.1.3 FLORA CUDA Accelerated Implementations

Table 12-3 Execution Time of FLORA after GPU Design Iteration 2 for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
Read image in greyscale	1.83	24.37%
Otsu Thresholding	1.14 (2 iterations)	15.18%
Calculate Image Moments	1.13	15.04%
Find Image Contours	1.08 (2 iterations)	14.38%
CUDA memory upload & download overhead	0.62	8.26%
CUDA Apply Rotation Matrix	0.59	7.86%
Image blurring	0.56	7.46%
Load Pre-trained classifier	0.27	3.60%
Other functions <0.10ms each	0.29	3.85%

CUDA accelerated functions implemented in iteration 2:

- Apply Rotation

Total execution time: 7.51ms excluding OpenCV initialisation

Accuracy 89.23%

Table 12-4 Execution Time of FLORA after GPU Design Iteration 3 for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
Read image in greyscale	1.9	22.38%
Find Image Contours	1.44 (2 iterations)	16.96%
Find Image Moments	1.14	13.43%
CUDA Otsu Thresholding	0.98 (2 iterations)	11.54%
CUDA memory upload & download overhead	0.75	8.83%
CUDA Apply Rotation Matrix	0.54	6.36%
CUDA Matrix creation & destruction overhead	0.41	4.83%
Image blurring	0.4	4.71%
Load Pre-trained classifier	0.27	3.18%
Other functions <0.10ms each	0.66	7.77%

CUDA accelerated functions implemented in iteration 3:

- Apply Rotation

- Otsu Thresholding

Total execution time 8.49ms excluding OpenCV initialisation

Accuracy 90.77%

Table 12-5 Execution Time of FLORA after GPU Design Iteration 4 for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
CUDA Find Regions (CCL)	10.38 (3 iterations)	37.61%
Find Image Moments	2.72	9.86%
CUDA Otsu Thresholding	2.28 (3 iterations)	8.26%
Read image in greyscale	1.9	6.88%
Image blurring	1.8 (3 iterations)	6.52%
Find Image Contours	1.68	6.09%
CUDA memory upload & download overhead	1.06	3.84%
CUDA Matrix creation & destruction	0.86	3.12%
CUDA Apply Rotation Matrix	0.27	0.98%
Load Pre-trained classifier	0.27	0.98%
Other functions < 10.0ms each	4.38	15.87%

CUDA acceleration applied in iteration 4:

- Apply Rotation
- Otsu Thresholding
- Find Regions (CCL)

Total Execution Time 27.6ms excluding OpenCV initialisation

Accuracy 85.54%

Table 12-6 Execution Time of FLORA after GPU Design Iteration 5 for Low Resolution Input

Function	Execution Time ms	Percentage of total execution time
Read image in greyscale	1.9	21.02%
Find Image Contours	1.42 (2 iterations)	15.71%
Find Image Moments	1.15	12.72%
CUDA memory upload & download overhead	1.1	12.17%
CUDA Otsu Thresholding	0.84 (2 iterations)	9.29%
CUDA Apply Rotation Matrix	0.61	6.75%
CUDA Image blurring	0.56	6.19%
CUDA Matrix creation & destruction overhead	0.37	4.09%
Load Pre-trained classifier	0.25	2.77%
Other functions <0.10ms each	0.84	9.29%

CUDA acceleration applied in iteration 5:

- Apply Rotation
- Otsu Thresholding
- Image blurring

Total Execution Time 9.04ms excluding OpenCV initialisation

Accuracy 89.23%

12.2 Code Execution Time Profiling Results – High Resolution

12.2.1 Original FLORA Octave Implementation with High Resolution Input

Table 12-7 Execution Time of Original FLORA System

Function	Execution Time ms	Percentage of total execution time
Convex Hull Properties	2115.86	25.86%
Otsu Thresholding	2042.94 (4 iterations)	24.97%
Apply Rotation	1559.48	19.06%
Image blurring	948.9	11.60%
Region Properties (Area & Perimeter)	746.72	9.13%
Find Regions (CCL)	351.35 (3 iterations)	4.29%
Calculate Image Orientation	271.27	3.32%
Read image	60	0.73%
Greyscale	46.15	0.56%
Calculation of Feature Descriptors	31.21	0.38%
KNN Prediction	9.02	0.11%

Total Execution Time 8182.90ms

Accuracy 89.74%

12.2.2 FLORA C++ Implementation with High Resolution Input

Table 12-8 Execution Time of FLORA before GPU Design Iteration 1 & 2 with High Resolution Input

Function	Execution Time	Percentage of total execution time
Find Image Contours	45.55 (5 iterations)	23.35%
Image blurring	41.04 (4 iterations)	21.04%
Read image in greyscale	36.03	18.47%
Otsu Thresholding	33.35 (5 iterations)	17.10%
Calculate Image Moments	24.1	12.36%
Apply Rotation Matrix	12.23	6.27%
Bounding Rectangle	1.89	0.97%
Convex Hull	0.43	0.22%
Load Pre-trained Classifier	0.26	0.13%
Region Area & Perimeter	0.05	0.03%

Hull Area & Perimeter	0.05	0.03%
Other functions <0.01ms each	0.07	0.04%

Total Execution Time 195.05ms

Accuracy 89.23%

12.2.3 FLORA CUDA Accelerated Implementations with High Resolution Input

Table 12-9 Execution Time of FLORA after GPU Design Iteration 2 with High Resolution Input

Function	Execution Time	Percentage of total execution time
Find Image Contours	45.3 (5 iterations)	23.65%
Image blurring	40.28	21.03%
Read image in greyscale	35.75	18.67%
Otsu Thresholding	33.5 (5 iterations)	17.49%
Calculate Image Moments	24.83	12.97%
CUDA memory upload & download overhead	6.73	3.51%
CUDA Apply Rotation Matrix	2.57	1.34%
Load Pre-trained classifier	0.27	0.14%
Other functions <0.10ms each	2.28	1.19%

CUDA acceleration applied in iteration 2:

- Apply Rotation

Total Execution Time: 191.51ms

Accuracy 89.23%

Table 12-10 Execution Time of FLORA after GPU Design Iteration 3 with High Resolution Input

Function	Execution Time	Percentage of total execution time
Find Image Contours	45.85 (5 iterations)	24.79%
Image blurring	40.12	21.69%
Read image in greyscale	35.25	19.06%
Find Image Moments	25.69	13.89%
CUDA memory upload & download overhead	19.33	10.45%
CUDA Otsu Thresholding	13.5 (5 iterations)	7.30%
CUDA Apply Rotation Matrix	2.5	1.35%
CUDA Matrix creation & destruction overhead	0.51	0.28%
Load Pre-trained classifier	0.27	0.15%
Other functions <0.10ms each	1.92	1.04%

CUDA acceleration applied in iteration 3:

- Apply Rotation
- Otsu Thresholding

Total Execution time 184.94ms

Accuracy 90.77%

Table 12-11 Execution Time of FLORA after GPU Design Iteration 4 with High Resolution Input

Function	Execution Time	Percentage of total execution time
CUDA Find Regions (CCL)	115.16 (4 iterations)	34.86%
Find Image Moments	46.15	13.97%
Image blurring	45.84 (3 iterations)	13.87%
Read image in greyscale	35.12	10.63%
CUDA Otsu Thresholding	23.72 (4 iterations)	7.18%
Find Image Contours	19.16	5.80%
CUDA memory upload & download overhead	12.32	3.73%
CUDA Apply Rotation Matrix	1.22	0.37%
CUDA Matrix creation & destruction overhead	0.85	0.26%
Load Pre-trained classifier	0.27	0.08%
Other functions <0.10ms each	30.57	9.25%

CUDA acceleration applied in iteration 4:

- Apply Rotation
- Otsu Thresholding
- Find Regions (CCL)

Total Execution Time 330.36ms

Accuracy: 85.54%

Table 12-12 Execution Time of FLORA after GPU Design Iteration 5 with High Resolution Input

Function	Execution Time	Percentage of total execution time
Find Image Contours	44.7 (5 iterations)	29.36%
Read image in greyscale	35.43	23.27%
Find Image Moments	20.1	13.20%
CUDA Image blurring	19.66 (4 iterations)	12.91%
CUDA memory upload & download overhead	14.02	9.21%
CUDA Otsu Thresholding	12.55 (5 iterations)	8.24%
CUDA Apply Rotation Matrix	2.47	1.62%
CUDA Matrix creation & destruction overhead	0.46	0.30%
Load Pre-trained classifier	0.27	0.18%
Other functions <0.10ms each	2.61	1.71%

CUDA acceleration applied in iteration 5:

- Apply Rotation
- Otsu Thresholding
- Image blurring

Total Execution time 152.27ms

Accuracy: 90.77%

EBE Faculty: Assessment of Ethics in Research Projects

Any person planning to undertake research in the Faculty of Engineering and the Built Environment at the University of Cape Town is required to complete this form before collecting or analysing data. When completed it should be submitted to the supervisor (where applicable) and from there to the Head of Department. If any of the questions below have been answered YES, and the applicant is NOT a fourth year student, the Head should forward this form for approval by the Faculty EIR committee: submit to Ms Zulpha Geyer (Zulpha.Geyer@uct.ac.za; Chem Eng Building, Ph 021 650 4791). Students must include a copy of the completed form with the final year project when it is submitted for examination.

Name of Principal

Researcher/Student: Khagendra Naidoo

Department: ELECTRICAL ENGINEERING

If a Student: YES **Degree:** Bsc Computer & Electrical Engineering

Supervisor: Simon Winberg

If a Research Contract indicate source of funding/sponsorship:

Research Project

Title: GPU Acceleration of the Fynbos Leaf-based Online Recognition Application

Overview of ethics issues in your research project:


Question 1: Is there a possibility that your research could cause harm to a third party (i.e. a person not involved in your project)?	YES	NO
Question 2: Is your research making use of human subjects as sources of data? If your answer is YES, please complete Addendum 2.	YES	NO
Question 3: Does your research involve the participation of or provision of services to communities? If your answer is YES, please complete Addendum 3.	YES	NO
Question 4: If your research is sponsored, is there any potential for conflicts of interest? If your answer is YES, please complete Addendum 4.	YES	NO

If you have answered YES to any of the above questions, please append a copy of your research proposal, as well as any interview schedules or questionnaires (Addendum 1) and please complete further addenda as appropriate.

I hereby undertake to carry out my research in such a way that

- there is no apparent legal objection to the nature or the method of research; and
- the research will not compromise staff or students or the other responsibilities of the University;
- the stated objective will be achieved, and the findings will have a high degree of validity;
- limitations and alternative interpretations will be considered;
- the findings could be subject to peer review and publicly available; and
- I will comply with the conventions of copyright and avoid any practice that would constitute plagiarism.

Signed by:

	Full name and signature	Date
Principal Researcher/Student:	Khagendra Naidoo 	14 October 2015

This application is approved by:

Supervisor (if applicable):		14 October 2015
HOD (or delegated nominee): Final authority for all assessments with NO to all questions and for all undergraduate research.		
Chair : Faculty EIR Committee For applicants other than undergraduate students who have answered YES to any of the above		

ADDENDUM 1:

Please append a copy of the research proposal here, as well as any interview schedules or questionnaires:

ADDENDUM 2: To be completed if you answered YES to Question 2:

It is assumed that you have read the UCT Code for Research involving Human Subjects (available at <http://web.uct.ac.za/depts/educate/download/uctcodeforresearchinvolvinghumansubjects.pdf>) in order to be able to answer the questions in this addendum.

2.1 Does the research discriminate against participation by individuals, or differentiate between participants, on the grounds of gender, race or ethnic group, age range, religion, income, handicap, illness or any similar classification?	YES	NO
2.2 Does the research require the participation of socially or physically vulnerable people (children, aged, disabled, etc) or legally restricted groups?	YES	NO
2.3 Will you not be able to secure the informed consent of all participants in the research? (In the case of children, will you not be able to obtain the consent of their guardians or parents?)	YES	NO
2.4 Will any confidential data be collected or will identifiable records of individuals be kept?	YES	NO
2.5 In reporting on this research is there any possibility that you will not be able to keep the identities of the individuals involved anonymous?	YES	NO
2.6 Are there any foreseeable risks of physical, psychological or social harm to participants that might occur in the course of the research?	YES	NO
2.7 Does the research include making payments or giving gifts to any participants?	YES	NO

If you have answered YES to any of these questions, please describe below how you plan to address these issues:

ADDENDUM 3: To be completed if you answered YES to Question 3:

3.1 Is the community expected to make decisions for, during or based on the research?	YES	NO
3.2 At the end of the research will any economic or social process be terminated or left unsupported, or equipment or facilities used in the research be recovered from the participants or community?	YES	NO
3.3 Will any service be provided at a level below the generally accepted standards?	YES	NO

If you have answered YES to any of these questions, please describe below how you plan to address these issues:

ADDENDUM 4: To be completed if you answered YES to Question 4

4.1 Is there any existing or potential conflict of interest between a research sponsor, academic supervisor, other researchers or participants?	YES	NO
4.2 Will information that reveals the identity of participants be supplied to a research sponsor, other than with the permission of the individuals?	YES	NO
4.3 Does the proposed research potentially conflict with the research of any other individual or group within the University?	YES	NO

If you have answered YES to any of these questions, please describe below how you plan to address these issues: