Image processing using GPUs

Daniel Connolly
AC40001 Honours Project
BSc (Hons) Applied Computing
University of Dundee, 2016-2017
Supervisor: Dr Iain Martin

Abstract – *This project focuses on the development of a software solution for evaluating relative the speed of image processing of central processing units versus graphical processing units.*

For this comparison, an application was created which receives several series of images generated by an external source. These images were then processed on both pieces of hardware using the same algorithm.

The measured results show that the graphical processing unit is significantly faster than the central processing unit at tracking features between images in the scenarios tested.

1 Introduction

The aim of the project is to evaluate the usefulness of exploiting the architecture of GPUs (Graphical Processing Units) for completing certain types of tasks compared to those same tasks being performed on a CPU (Central Processing Unit).

This will be done through creating a software application which allows processing to be done on both the CPU and the GPU inputs and outputs which are kept as similar as possible. The algorithms and techniques used will also be similar, but will be adapted to take advantage of the architecture of the device being used. The total processing time taken for the CPU and GPU will be measured, compared, and displayed in a readable format which will highlight the differences in speed between the two.

2 Background

2.1 CPU

In the early days of computing, processors had a single core which could execute only one thread at any point in time. This meant that the multithreading ability of these early computers was limited to interleaving by the operating system and no parallel processing was possible at the hardware level.

Modern processors however have multiple cores and each core can process several threads at once. Programmers and compilers can exploit this functionality to allow multiple instructions or sets of instructions which are not dependant on one-another to be executed simultaneously on the processor.

2.2 GPU

GPUs are normally used for rendering of graphics on a display device. They are designed in such a way as to be able to manipulate memory to create images in a framebuffer [1], ready for display, far faster than a CPU would be able to. GPUs are present in most computers and devices, either in the form of a *dedicated graphics card* or *integrated graphics* (integrated as part of the motherboard or on the same die as the CPU).

The idea of parallel processing is taken to far greater levels by GPUs which consist of thousands of small cores. This allows for greater parallelisation potential as far more instructions can be executed simultaneously at the cost of slower individual execution. This clearly useful for graphical applications as it is common to do computations for each pixel on the screen or for each vertex in an object and these individual operations are usually not dependant on each-other.

2.3 OpenCL and CUDA

Khronos' OpenCL [2] and NVIDIA's CUDA [3] framework provide APIs which allow programmers the ability to exploit the GPU's parallelism for processing which is not necessarily graphics-related. In general, this means that most the processing for an application using one of these OpenCL or CUDA will occur on the CPU, while a small amount of intensive and parallelizable work will be offloaded to the GPU.

One potential pitfall associated with this usage of the GPU is that there is a substantial cost associated with copying data between the CPU and GPU's respective memory. This means that the workload done by the GPU on the data copied must be substantial enough that the speed increase in processing it offsets the time taken to copy it back and forth. In addition, not all processing is equally suitable for parallelisation, such as instructions which rely on the outcomes of each other or use many branches. The time taken to process a set of data on the GPU takes as long as the last core does to finish processing its workload and so it is optimal to keep all cores doing useful work for as much of the time as possible.

One task which is often useful to parallelise is image processing. This is because many of the algorithms and techniques which are applied to images operate upon each pixel in the image or groups of pixels. For these tasks, it may be beneficial to exploit the GPU's parallel architecture to speed up the work done through doing the processing required for each pixel in the image simultaneously. For this reason, it was decided that aim of the project is to create a software application which will offer the ability to compare the speed at which the CPU and GPU complete the same image processing tasks to evaluate the usefulness of exploiting a GPU for tasks of this sort.

2.4 Options for Implementation

Different options will be evaluated for several aspects of the application to be produced, such as which programming languages and frameworks are to be used for controlling processing done on both the CPU and GPU. In addition, it must be decided which algorithms and techniques would be implemented for the comparison, as well as the source of the images to be processed. Finally, for the final application, a GUI (Graphical User Interface) library should be decided upon to implement visual elements allowing the comparison results and any parameter options to be presented in a readable format.

2.5 PANGU

For a source of images, the PANGU (Planet and Asteroid Natural scene Generation Utility) [4] simulation software would be useful tool. PANGU is software developed by the Space Technology Centre within the University of Dundee and aims to provide realistic images of the surface of planetary bodies using a combination synthetic and real data.

The images produced by PANGU are useful for simulations of planetary landing and testing guidance systems. PANGU can be run as either a desktop application, where the user can interact with the scene and control the camera through a user-interface; or as a server where the application runs in the background and takes input commands over a socket [5] using an API (Application Programming Language) in C. In non-server mode, PANGU can process flight files (.fli) which contain sequential instructions for camera movement and automatically execute these commands. This is useful for repeatedly simulating and viewing the same complex series of movements with little user interaction required. PANGU has two modes for controlling the camera: *craft* and *model*. In *craft* mode, the camera controls as if it were a spacecraft which can move independently in different directions and the camera view can be adjusted separately. In *model* mode, the camera moves around a point in space, with the view centred upon that point at all times.

2.6 Feature Detection

A common feature of planetary landers is the ability to track the movement of features around them using the series of images provided from their camera feeds. Feature tracking a useful tool which allows landers to gain information about their surroundings, position, and movement relative to the objects around them; this is necessary because the distances involved between Earth and planetary landers are usually great enough that they cannot be directly human controlled in real-time due to the high signal latency. This processing must be done in real-time so that the lander can react to the changes around it and adjust its movement accordingly; therefore, it is important that images are processed quickly, yet with enough accuracy that mistakes are not made. For the purpose of feature tracking, the Harris Corner Detector algorithm [6] is commonly used, often with modifications to optimise its performance and accuracy in the particular use-case [7].

3 Specification

Initial meetings with the project supervisor centred around deciding upon the general project scope and requirements. The requirements which were decided upon were as follows: -

R1. The created application shall run on a Windows machine.
R2. The application shall receive input images from an external source.
R3. Input images shall be processed using some algorithm.
R4. This algorithm shall be executed on both CPU and GPU.
R5. The times for this processing to take place shall be stored.
R6. Results showing differences in processing time shall be calculated.
R7. A graphical user interface shall be created.
R8. The GUI should allow the user to modify algorithm parameters.
R9. The GUI shall allow the user to execute a test.
R10. The GUI should display the results of a test in a graphical format.

It was also decided that the priority would be for the author to research techniques for GPU programming and learn how to use one of these frameworks, and then decide upon which algorithm for image processing would be used in the comparison.

4 Design

4.1 Processing Done

It was decided that use of the Harris Corner Detector algorithm [6] for feature tracking would make for a good comparison between the performance of the GPU and CPU. This is because it is a relatively complex algorithm with many steps and possible customisations and although it was first presented in 1988, it is still often used today. In addition, the algorithm was not originally designed to be run in parallel, but most steps throughout make for good candidates for parallelisation.

Several modifications and additions were made to the original algorithm to improve its performance in the scenario in which it is being used, namely tracking feature points in images from the PANGU simulation software.

4.2 Image Source

The feature tracking algorithm would be applied to a series of images generated by PANGU. This would allow images to be created and processed in real-time, as if they were coming from an actual lander. In addition, flight files could be used to specify camera movement and flight paths which can be used to repeat tests on CPU and GPU over multiple iterations with great accuracy.

4.3 GPU Programming Framework

For the GPU programming, CUDA would be used rather than OpenCL. This decision was made partly due to the availability of quality documentation and examples for CUDA compared to those available for OpenCL as this would be useful during the learning process for graphics card programming. In addition, although CUDA is a proprietary framework, an NVIDIA graphics card would be available and so there would be no issue in executing CUDA code.

Due to the availability of CUDA debugging tools (namely Nsight [8]) for Visual Studio [9], as well as the ease of integrating CUDA code into Visual Studio projects, the project was written primarily using Visual Studio 2017 as an IDE (Integrated Development Environment).

4.4 Overview and Hardware

Overall, the system would consist of PANGU running in server mode, generating images using a GeForce GTX 1080. The same series of images would be generated then and then processed using a CPU and GPU in turn. The CPU used would be an Intel® Core™ i7-6700K and the GPU used for processing would be a GeForce GTX 970. This choice of hardware was chosen (as opposed to processing the images on the GTX 1080) because the price of the CPU and GPU pair is closer than it would be otherwise. In addition, two graphics cards are used so that the GPU processing time is not hindered by the need to generate new images.

4.5 GUI Library

The GUI for the final application will be created using the Qt framework [10]. Qt will be used because it is a well-known and stable framework with a great degree of customization for themes and functionality, allowing for an attractive and functional UI to be created. In addition, the popularity of Qt means that there are many quality resources to reference when creating applications using the framework which will be helpful during the implementation process.

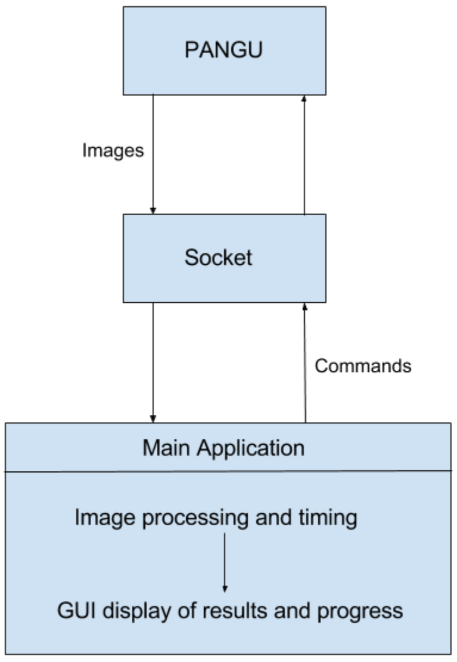

PANGU
Images
Socket
Commands
Main Application
Image processing and timing
GUI display of results and progress

Figure 1 - System Block Diagram

5 Implementation and Testing

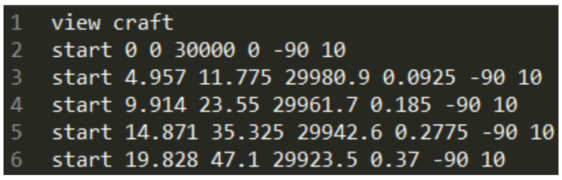
The project was written mainly in C++ using Visual Studio as an IDE. Visual Studio was used for debugging host-side code, while the Nsight extension was used to assist with debugging of device-side CUDA code. Using these tools, it possible to check that each stage of the image processing was completed as expected and made it easier to locate and remedy any problems which were encountered throughout the development process.

5.1 PANGU Configuration

PANGU was configured to run in server mode and to utilise the GTX 1080 for generating images of size 1024x768 pixels in single channel PPM format [11] with one unsigned byte (0-255) used per pixel. This image size was selected as it would be large enough that significant processing would be required due to the number of pixels in the image, but it would also be small enough to display comfortably alongside a GUI on a standard monitor. Running in server mode, PANGU will execute independently and can be communicated to through a socket, using its C API.

5.2 PANGU Flight Files

When not running in server configuration, PANGU has the option to process flight files (.fli). These files contain a sequential series of instructions for camera movement and other parameters.



```
1  view craft
2  start 0 0 30000 0 -90 10
3  start 4.957 11.775 29980.9 0.0925 -90 10
4  start 9.914 23.55 29961.7 0.185 -90 10
5  start 14.871 35.325 29942.6 0.2775 -90 10
6  start 19.828 47.1 29923.5 0.37 -90 10
```

Figure 2 - PANGU Flight File Excerpt

In *Figure 1* the first six lines of a PANGU flight file are shown. Line 1 uses the *view* command with the parameter *craft* to instruct PANGU to use the *craft* camera control mode. Lines 2-6 use the *start* command to set the camera's current position and rotation. The floating-point values following the *start* command are parameters which set the camera's X position, Y position, Z position, yaw, pitch, and roll in that order.

When using a flight file, PANGU attempts to process each command and generate a new image to be shown in the viewer as quickly as possible.

However, when running in server mode as will be used for this project, the functionality for processing and executing commands from flight files is not available. Rather, each camera movement and setting must be individually changed through commands in the API. To work around this limitation, a parser was created which will read a PANGU flight file and extract the most commonly used commands. This series of commands is then stored and can be executed by sending each one through the socket to the PANGU server. After each command is executed, a new image is requested and will be processed. In this way, the same series of images can be generated and processed by both the CPU and GPU, providing an accurate testing environment.

In addition, giving the code controlling the PANGU server access to the full list of commands it will execute throughout the duration of the flight allows the possibility for images to be generated in advance and stored in a thread-safe queue [12]. The size of this queue is a configurable parameter which allows for the testing process to be possibly expedited as new images can be generated while the previous ones are still being processed; while still allowing for images to be generated in real-time using a model and flight file and without prior preparation.

5.3 Flight File Writer

Most of the existing flight files for PANGU were using the *model* camera mode, however, the server mode C API only supports execution of commands using *craft* mode. For this reason, a tool was created, separately from the main application, whose purpose is to generate PANGU flight files which can then be parsed. The tool supports two modes, the first mode creates a flight file with commands to move the camera from a starting position and view direction to a destination position and view direction. The tool writes commands which interpolate linearly between this start and end state with a configurable number of steps. The other option which the tool supports is to generate a flight file containing commands which, when executed, causes the camera to orbit the equator of a model from a starting azimuth for a configurable distance.

With these tools in place, it is possible to create flight files for a specific model which can then be processed and executed. Images can then be generated in a queue and these images can be popped from the queue and processed whenever required.

5.4 Feature Selection

For selecting features points to use for tracking, the Harris Corner Detector algorithm is used. The algorithm is based on computing the derivatives I_x and I_y for each pixel in the image through convolution with the Sobel kernels [13] (Figure 3)and the surrounding pixels in a 3×3 window. The Sobel kernels are used to create images which approximate the *X* and *Y* gradients of an input image (Figure 4). This is useful for image processing techniques which require the location of edges or corners.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 3 - Sobel Kernels

$$I_x(i,j) = \sum_{l=0}^{2}\sum_{m=0}^{2} I(i+l-1, j+m-1) \times \mathrm{G}_x(l,m)$$

$$I_y(i,j) = \sum_{l=0}^{2}\sum_{m=0}^{2} I(i+l-1, j+m-1) \times \mathrm{G}_y(l,m)$$

Figure 4 - Gradients

[7]

The products of the derivatives are then smoothed using a Gaussian window function [14] with size 7 × 7 pixels and a sigma value of 0.81 (Figure 5).

$$S_{x2} = G_\sigma \times {I_x}^2$$
$$S_{y2} = G_\sigma \times {I_y}^2$$
$$S_{xy} = G_\sigma \times \left(I_x I_y\right)$$

Figure 5 - Smoothed products of derivatives

For each pixel, a matrix (Figure 6) is created. The Harris response value at that pixel can then be computed by finding the Trace [15] and Determinant [16] of the matrix (Figure 7), where *k* is a configurable value which is generally in the range $[0.04 : 0.1]$.

$$M = \begin{bmatrix} S_{x2} & S_{xy} \\ S_{xy} & S_{y2} \end{bmatrix}$$

Figure 6 - Smoothed derivative products matrix

$$R = Det(M) - k\left(\mathrm{Trace(M)}\right)^2$$

Figure 7 - Harris response calculation

The computation of the response value can be simplified, meaning that the matrix does not need to be constructed and the comparatively expensive determinant and trace calculations do not have to take place. This is done by instead calculating $Det(M)$ and $Trace(M)$ as shown in Figure 8. This is possible because the matrix size is only 2 × 2, so the Determinant can be calculated by subtracting the products of the diagonals of the matrix. The Trace of a 2 × 2 matrix can likewise be calculated simply by adding together the top left and bottom right values.

$$Det(M) = (S_{x2} \times S_{y2}) - ({G_{xy}}^2)$$
$$Trace(M) = G_{x2} + G_{y2}$$

Figure 8 - Determinant and Trace simplification

The response value calculation can then be performed as normal.

A configurable minimum threshold is then applied to *R* and non-maximum suppression is applied in a sliding 7 × 7 window throughout the matrix of response values.

To reduce the chance of undesirable points being tracked due to effects such as aliasing, this implementation throws away all the feature points in an area if multiple thresholded values are present in the same window. This means that fewer points are tracked, but the points which are tracked are more likely to be valid. The results of this change can be seen by comparing Figure 9 (original algorithm) with Figure 10 (after thresholding modification). After the change, fewer points are tracked around the outside of the object due to aliasing and so more valid features are tracked closer to the centre.

The resulting values are sorted and the locations of the highest of these are stored, up to a configurable limit.

The set of maximum values are compared against the list of currently tracked features, to ensure that the same feature point, or different parts of the same feature point are not tracked multiple times.

Lastly, for each new tracked feature, a 7 × 7 template is extracted from the original image and stored along with the feature's location in the image.

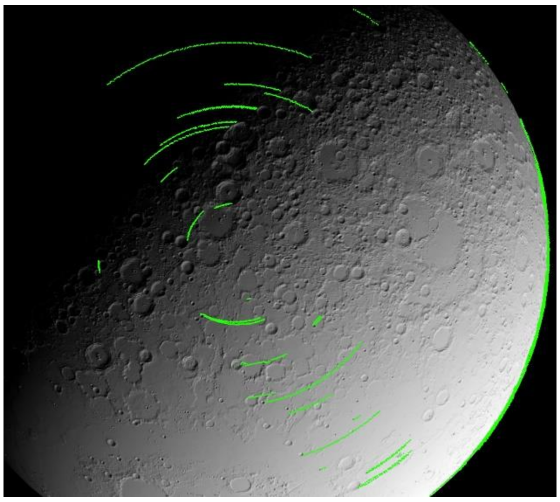


Figure 9 – Before thresholding modification

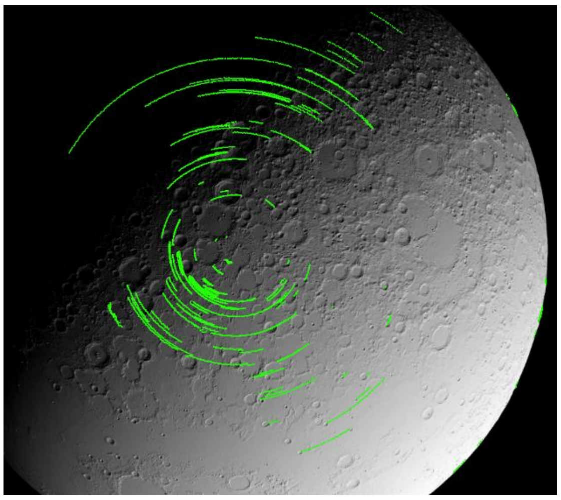


Figure 10 – After thresholding modification

5.5 Feature Tracking

The technique used for tracking the location of features throughout the series of images is done by comparing the extracted template for the tracked feature in the previous image to the corresponding template for the pixels in a search window around the feature location in the new image.

Computing a correlation value between two templates for use in comparison is done using the correlation statistic shown in Figure 11.

$$C(i,j) = \frac{\sigma_{IT}}{\sqrt{\sigma_I^2 \cdot \sigma_T^2}}$$

where

$$\sigma_{IT} = \sum_{l=0}^{6}\sum_{m=0}^{6}(I(i + \mathrm{l} - 3, \mathrm{j} + \mathrm{m} - 3) - (\mathrm{I})) \times (T(l,m) - (T))$$

$$\sigma_I^2 = \sum_{l=0}^{6}\sum_{m=0}^{6}(I(i + \mathrm{l} - 3, \mathrm{j} + \mathrm{m} - 3) - (\mathrm{I}))^2$$

$$\sigma_T^2 = \sum_{l=0}^{6}\sum_{m=0}^{6}\left(T(l,m) - (\mathrm{T})\right)^2$$

Figure 11 - Correlation statistic

[7]

(T) is the average of the pixel intensities in the template for the tracked feature on the previous image; (I) is the average of the pixel intensities in the window around the current search pixel in the next image; $T(x,y)$ is the pixel intensity value at the tracked pixel on the previous image and $I(x,y)$ is the pixel intensity value of the current search pixel in the new image.

The intermediary values σ_{IT}, σ_I^2 and σ_T^2 are used to calculate the correlation value for that pixel in the search area.

The pixel in the search area with the highest correlation value is the most likely candidate for tracking the feature to. If this pixel's correlation value is above a configurable threshold then its location is selected as the feature's new location on the image. If the correlation value does not meet or exceed the required threshold, then the feature is considered to be lost.

Templates for tracked features are updated periodically, after the feature has been successfully tracked across a configurable number of frames. However, updating the template can cause issues because any minor variations or errors in the template such as those caused by aliasing may cause the tracking to begin following something which is not the original feature. In order to attempt to remedy this problem, rather than simply switching immediately to the new template, both the old and new templates are tracked in parallel for several frames. If, at the end of this set of frames, the distance between the location of the tracked location for both the old and new template is within a configurable threshold, then the new template becomes the tracked template. If the distance is above the threshold, then the feature is lost and it will not be tracked any longer.

5.6 Implementation Details

5.6.1 Image Normalization

The images received from PANGU are already in the ideal format for use with the feature detection part of the algorithm. However, the feature tracking section, including the values in the templates stored for each tracked feature and the calculation of correlation values requires that each pixel in the image is a normalized floating point value between $[0.0 : 1.0]$. Therefore, an extra step is added to the image processing stage in which, a version of the current image being processed is created with the values scaled from unsigned char $[0 : 255]$ to floating point $[0.0 : 1.0]$. This is stored along with the image in its original format and is used when required.

5.6.2 Non-Maximum Suppression

In the CPU implementation of the feature tracking algorithm, non-maximum suppression of Harris response values is achieved by sorting the list of values in descending order, then iterating through this list and removing response values in a window around the current point. This has the effect that only the highest response values in the image are retained and points too close together will not be tracked simultaneously.

The GPU implementation of the non-maximum suppression takes a slightly different approach to increase the potential parallelisation and thereby exploit the GPU’s architecture.

In the GPU implementation, matrix of boolean values matching the size of the Harris response value matrix is created and initialised with every value set to *true*. A CUDA kernel is launched for each value in the matrix of Harris response values. This kernel compares each value in an area around the current value’s location against the current value. If the current value is greater than any in the window, then the corresponding location in the boolean matrix is set to *false*. Although it is generally discouraged to have multiple threads writing to the same locations in memory, in CUDA this is acceptable if the values being written to the same location are guaranteed to be the same, as is this case in this instance. Once this operation is complete, all threads are synchronized and a counter is atomically incremented for each thread for which the boolean at its location is still set to *true*; this value is passed back to the host from the device.

The number of thresholded feature points is used by the host to allocate the correct amount of space on the GPU to store them. A second kernel is then launched for each value in the Harris response value matrix which checks whether the value for that thread is going to be kept using the boolean matrix, and adds it to the array which was allocated previously.

Once the second kernel returns, the array of feature points is sorted in descending order based on the Harris response values. The first elements of this array are taken up to a configurable limit and are used in the next part of the algorithm.

5.6.3 Graphical User Interface

A GUI was created for the final application using the Qt framework. The created interface is designed to allow for easy customization of algorithm parameters and execution of timing tests. The interface also provides the ability to select a PANGU flight file to be executed for the test.

The last image which was processed is also visible during test execution to allow visualisation of the algorithm, with an overlay showing the location history of feature points which have been tracked from frame to frame. A second tab allows the view to be switched from the processing output image to a view of the statistics which have been gained from the currently running or previously completed test. The processing time for each image and overall progress on both CPU and GPU is also displayed in a graphical format. This setup makes it simple to run tests with different configurations and see the results in real-time in a simple, visual format.

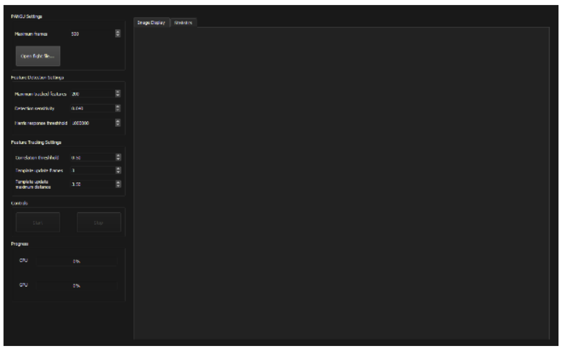
5.7 Meeting Specification

At the end of the implementation process, all project requirements which were decided upon in initial meetings with the tutor were met.

6 Final Product

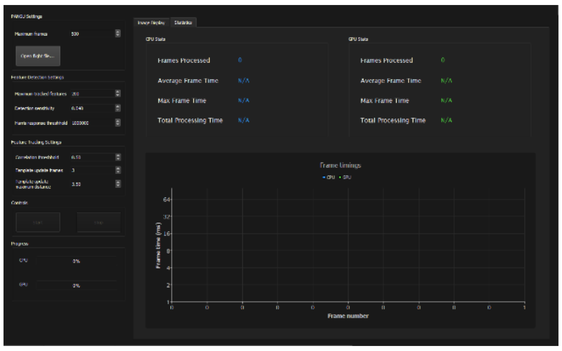
The product consists of a desktop application designed to run on the Windows 10 operating system.

Prior to a test being run, algorithm parameter and test setting options are set to their defaults, the test control buttons are not available until the user selects a flight file to use, and the processed image display and statistics tabs are empty of any information.



Maximum frames 500
Maximum tracked features 200
Detection sensitivity 0.040
Harris response threshold 1000000
Correlation threshold 0.50
Template update frames 3
Template update maximum distance 3.50
CPU 0%
GPU 0%

Figure 12 - Pre-test tab 1



CPU Stats
Frames Processed 0
Average Frame Time N/A
Max Frame Time N/A
Total Processing Time N/A

GPU Stats
Frames Processed 0
Average Frame Time N/A
Max Frame Time N/A
Total Processing Time N/A

Figure 13 - Pre-test tab 2

The *Open Flight File* button allows the user to select a PANGU flight file (.fli) to use for the test.

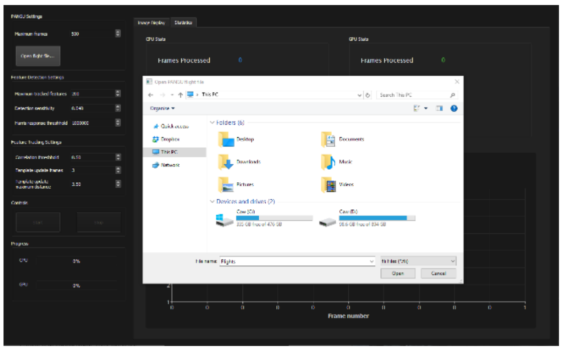
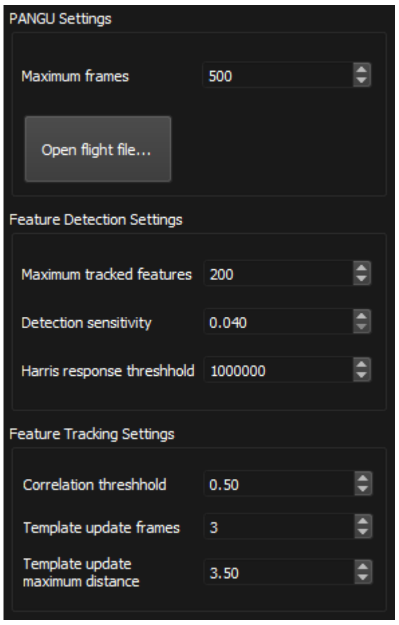


Figure 14 - Open Flight File

The user can configure algorithm parameters and test settings using the options on the left side of the GUI.



PANGU Settings

Maximum frames	500

Feature Detection Settings

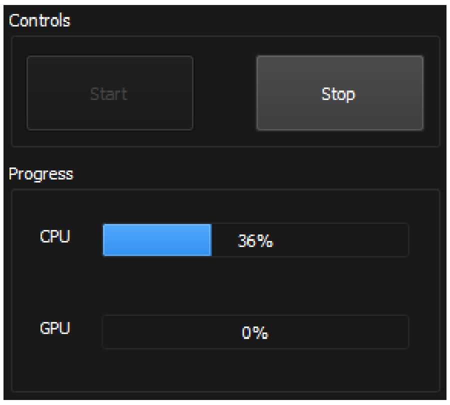
Maximum tracked features	200
Detection sensitivity	0.040
Harris response threshhold	1000000

Feature Tracking Settings

Correlation threshhold	0.50
Template update frames	3
Template update maximum distance	3.50

Figure 15 - Algorithm Parameters and Test Settings

Assuming PANGU is running locally in server-mode, displaying a model. Once the user selects a PANGU flight file to use through the GUI, the *start* button will become available and the user can press it to begin a test. The test progress can be monitored through the progress bars visible nearby. Once the test has been started, the user can stop the test part-way using the *stop* button.



CPU	36%
GPU	0%

Figure 16 - Test Controls and Progress

During test execution, the last processed image is displayed on the GUI in the *image display* tab.

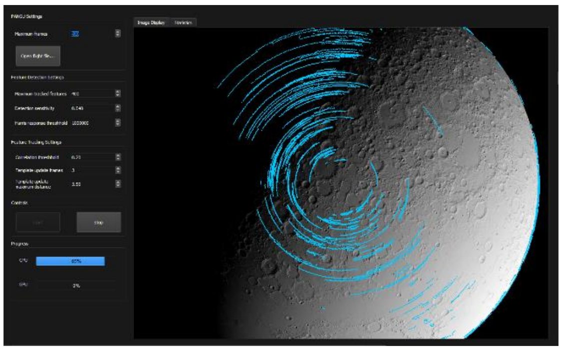
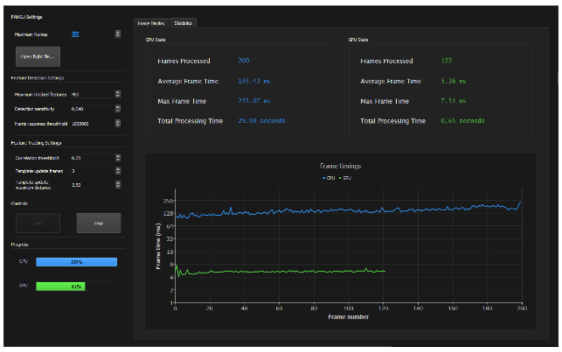


Figure 17 - Mid-test image display

Additionally, statistics from the currently running test are updated in real-time in the *statistics* tab. These results persist after the test finishes until a new test is started, at which point they are reset.



CPU Stats

Frames Processed	200
Average Frame Time	145.43 ms
Max Frame Time	235.07 ms
Total Processing Time	29.09 seconds

GPU Stats

Frames Processed	122
Average Frame Time	5.36 ms
Max Frame Time	7.51 ms
Total Processing Time	0.65 seconds

Figure 18 - Mid-test statistics

Once either the test is completed or the user has stopped the test manually, a new test can be started by following the same procedure. A new model can be opened in PANGU, a new flight file can be selected and algorithm parameters chosen; once the new test is started, the results and processed image display will replace those from the previously run test.

7 Evaluation

7.1 Optimisations

7.1.1 Pixel Intensity Normalisation

For the step of normalizing each pixel in the input image during processing, originally, each pixel value in the range $[0 : 255]$ was divided by 255 to scale it into the range $[0.0 : 1.0]$ in the same location in a new image. However, this step was sped up by using a pre-declared 256-value lookup table to map each unsigned byte value to its corresponding normalised floating point equivalent, without doing repetitive and relatively expensive divisions for each pixel of each new image. Although this optimisation was implemented on both the CPU and GPU versions of the algorithm, the improvement to the GPU version is very minor as every pixel is processed at the same time and so the time saved from each operation does not add up.

7.1.2 Automatic Parallelisation

Although the CPU is far faster than the GPU at executing instructions in a single thread, the multiple cores and multithreading available on modern processors allow for many threads to be executing simultaneously. For a fair comparison to take place, the CPU must be used to its full potential. Visual Studio's proprietary C++ Automatic Parallelisation [17] feature is therefore used to exploit the CPU fully when loops are executed for which the operation within does not require the results of previous iterations of the loop to function. This is the case in most stages of the algorithm which operate on each pixel in an image or matrix of values.

7.1.3 Compiler Settings

The following compiler options are used in Visual Studio for compilation of host-side code to produce a binary which executes as quickly as possible: -

- Enable full optimization (/Ox)
- Inline any suitable functions (/Ob2)
- Enable intrinsic functions (Oi)
- Favour fast code (/Ot)
- Omit frame pointers (/Oy)
- Enable whole program optimization (/GL)
- Enable function level linking (/Gy)
- Use fast floating point model (/fp:fast)
- Enable Advanced Vector Extensions 2 enhanced instruction set (/arch:AVX2)

Compilation of device-side (CUDA) code is configured to use the same optimisation settings as the host-side compiler where possible, and so similar optimisations will be made for code running on the GPU and CPU.

These optimisations are important and make a large different to the speed of processing. For example, when the algorithm is executed on the CPU with none of these optimisations enabled, the processing time for each image takes more than 4.6 times as long (from ~141ms to ~654ms).

7.2. Results

The results gained from testing the processing speed of the CPU and GPU on series of images gained from executing flight paths on different models are shown below.

7.2.1 Moon - Roll

Algorithm Parameters and Input	
Frames Processed	200
Max Tracked Features	400
Detection Sensitivity	0.040
Harris Response Threshold	1000000
Correlation Threshold	0.20
Template Update Frames	3
Template Update Max Distance	3.50

Test Results		
	CPU	**GPU**
Average Frame Time (ms)	141.92	5.24
Max Frame Time (ms)	189.41	6.95
Total Processing Time (s)	28.38	1.05

7.2.2 Itokawa - Approach

Algorithm Parameters and Input	
Frames Processed	300
Max Tracked Features	200
Detection Sensitivity	0.040
Harris Response Threshold	1000000
Correlation Threshold	0.60
Template Update Frames	3
Template Update Max Distance	3.50

Test Results		
	CPU	**GPU**
Average Frame Time (ms)	116.27	3.87
Max Frame Time (ms)	142.11	4.44
Total Processing Time (s)	34.88	1.16

7.2.3 Phobos – Pan Surface

Algorithm Parameters and Input	
Frames Processed	500
Max Tracked Features	200
Detection Sensitivity	0.040
Harris Response Threshold	1000000
Correlation Threshold	0.50
Template Update Frames	3
Template Update Max Distance	3.50

Test Results		
	CPU	**GPU**
Average Frame Time (ms)	102.20	3.85
Max Frame Time (ms)	141.54	5.26
Total Processing Time (s)	51.10	1.92

7.2.4 Phobos - Orbit Equator

Algorithm Parameters and Input	
Frames Processed	500
Max Tracked Features	200
Detection Sensitivity	0.040
Harris Response Threshold	1000000
Correlation Threshold	0.50
Template Update Frames	3
Template Update Max Distance	4.50

Test Results		
	CPU	**GPU**
Average Frame Time (ms)	108.67	3.75
Max Frame Time (ms)	142.71	6.01
Total Processing Time (s)	54.34	1.87

7.3 Conclusion

As can be seen from the test results, the total GPU processing time is generally around 26-30 times faster than the CPU implementation, with an average of 28.19 times taken from the results shown in this report.

In addition, it can be seen in the frame-to-frame timing graphs that the processing time for each image on the CPU varies far more than when processed on the GPU. While the CPU processing time for a single image can vary by up to 50ms between frames, the variation seen when processing is done on the graphics card is rarely more than 4ms showing far more consistency. This is mainly because the CPU must work on other tasks such as operating system processes at the same time while the graphics card does not share this burden.

Overall it is clear that it is far faster to do image processing of this type on the GPU rather than the CPU, especially in cases where the algorithm being used is optimal for exploiting the GPU's parallel architecture or can be modified in order to meet this requirement. In addition, it must be ensured that the time saved from doing a processing step on the GPU is substantial enough that it overshadows the cost in copying data back and forward between host and device.

8 Future Work

This project centres around two of the three parts of a planetary lander guidance system. The two parts in place are the receiving of images in real-time and the processing of these images to track features between frames. The third part which was not implemented is for the system to alter the movement of the craft in response to information gathered during the image processing stage. Due to the design of the application, this extension could be easily extended to include this feature if it were implemented. If this feature were added, the application would take information gathered during the image processing step and could guide the craft down towards the planetary body.

9 Acknowledgments

The author would like to thank Dr Iain Martin for his great support and guidance; as well being consistently enthusiastic and optimistic throughout the duration of the project.

References

[1] The Khronos Group, Inc., “Framebuffer,” 18 September 2015. [Online]. Available: https://www.khronos.org/opengl/wiki/Framebuffer. [Accessed 25 April 2017].

[2] The Khronos Group, Inc., “The open standard for parallel programming of heterogeneous systems,” Khronos, [Online]. Available: https://www.khronos.org/opencl/. [Accessed 9 April 2017].

[3] Nvidia, “CUDA Parallel Computing Platform,” NVIDIA, [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html. [Accessed 9 April 2017].

[4] STAR-Dundee, “PANGU - Planet and Asteroid Natural scene Generation Utility,” STAR-Dundee, [Online]. Available: https://www.star-dundee.com/products/pangu-planet-and-asteroid-natural-scene-generation-utility. [Accessed 9 April 2017].

[5] Microsoft Corporation, “Windows Sockets 2,” [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms740673(v=vs.85).aspx. [Accessed 26 April 2017].

[6] C. Harris and M. Stephens, “A Combined Corner and Edge Detector,” in *Proceedings of The Fourth Alvey Vision Conference,* Manchester, UK, 1988.

[7] N. Rowell, S. Parkes and M. Dunstan, “Image Processing for Near Eath Object Optical Guidance Systems,” 57th International Astronautical Congress, University of Dundee, October 2006.

[8] Nvidia, “NVIDIA Nsight Visual Studio Edition,” NVIDIA, [Online]. Available: https://developer.nvidia.com/nvidia-nsight-visual-studio-edition. [Accessed 11 April 2017].

[9] Microsoft Corporation, Microsoft Corporation, [Online]. Available: https://www.visualstudio.com/. [Accessed 11 April 2017].

[10] The Qt Company, “Qt | Cross-platform software development for embedded & desktop,” The Qt Company, [Online]. Available: https://www.qt.io/. [Accessed 11 April 2017].

[11] Netpbm, “PPM Format Specification,” 09 October 2016 . [Online]. Available: http://netpbm.sourceforge.net/doc/ppm.html. [Accessed 19 April 2017].

[12] C. Desrochers, “A Fast Lock-Free Queue for C++,” 2013. [Online]. Available: http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++. [Accessed 13 April 2017].

[13] I. Sobel, “History and Definition of the so-called "Sobel Operator", more appropriately named the Sobel-Feldman Operator,” 2 February 2014.

[14] L. Shapiro and G. Stockman, in *Computer Vision,* Prentice Hall, 2001, pp. 137-150.

[15] Encyclopedia of Mathematics, “Trace of a square matrix,” [Online]. Available: https://www.encyclopediaofmath.org/index.php/Trace_of_a_square_matrix. [Accessed 26 April 2017].

[16] Encyclopedia of Mathematics, “Determinant,” [Online]. Available: https://www.encyclopediaofmath.org/index.php/Determinant. [Accessed 26 April 2017].

[17] Microsoft Corporation, “Auto-Parallelization and Auto-Vectorization,” Microsoft, [Online]. Available: https://msdn.microsoft.com/en-us/library/hh872235.aspx. [Accessed 18 April 2017].

Appendices