

BSc (Honours) Degree in Computer Science

Final Year Project

**High Efficiency Image Alignment for use in
multi-exposure HDR image creation**

By

Dylan Tomkins

Project unit: PJE40

Supervisor: Dr. Bryan Carpenter

May 2019

Title: High Efficiency Image Alignment for use in multi-exposure HDR image creation

Author: Dylan Tomkins

Supervisor: Dr. Bryan Carpenter, School of Computing

Abstract:

While there is many commercial software available for image stitching (alignment/registration), there is a lack of reliable or high performance software for the alignment of images with varying exposure, for use in multiple exposure High Dynamic Range image composition. This project will look at existing software for this task, or that could be adapted for this task, and existing algorithms in academia or design that could be adapted for a high performance implementation via use of a GPU, by using a framework such as OpenCL or CUDA. Results show in more cases than not, the new algorithm is able to align images faster by using the GPU, depending on number of images, and image size.

Key Words: OpenCL, GPGPU, Image Stitching, Image alignment, Image Registration

Word count: 10759

	0
1. Introduction	4
1.1. Background, Motivation and Aim of Project	4
1.2. Deliverables	5
1.3. Constraints	5
1.4. Project Approach, Plan and Risk	5
2. Literature Review	7
2.1. A brief overview of algorithm types	7
2.1.1 Single/Multi-modality methods	7
2.1.2 Automatic/interactive methods	7
2.1.2 Intensity/feature-based algorithms	7
2.2. Basic CPU VS GPU Architecture and Memory system	8
2.3 The Image Registration Recipe	8
2.3.1 The (Alignment/Similarity) Measure	9
2.3.2 The Transformer	10
2.3.2 The Optimiser	12
3. Requirements and Testing Methodology	14
3.1 Performance Requirements	14
3.2 Functional and Non-functional Requirements	16
4. Design	19
4.1. A Quick note on Optimiser Speed	19
4.2. Marriage of the Transform and Measure functions	19
4.3. Pre-Compensating for Exposure	19
4.4. The Measure Function	20
4.5. Image Subsampling	20
4.6. OpenCL's image2d_t location-based cacheable data type, and interpolation	20
4.7. New Design	20
5. Implementation	23
5.1. The Development Environment, Language Choice...	23
5.2. A note on the development lifecycle	23
5.3. Initial Implementation	23
5.3.1 Load images using OpenCV and display	24
5.3.2 Convert Images to grayscale (using an OpenCV function)	24
5.3.3 Build image the brightness/compensation function	24
5.3.4 Build Similarity Measure	24
5.3.5 Build Transform logic in to the Similarity Measure function	25

5.3.6 Build function to merge two images based on the translation	25
5.3.7 Build Optimiser (to calculate a list of new misalignment vectors to measure)	26
5.4. Porting to GPU using OpenCL and basic optimisation	27
5.5. Further Optimisation for OpenCL	28
5.6. Implementation Issues and Challenges	30
5.6.1 Performance using the C-style Array	30
5.6.2 Performance using the Image2D data type	32
5.6.2 Other Assorted Issues	33
6. Testing and Evaluation	34
6.1 Performance Requirements	34
6.3 Requirements Testing	36
7. Conclusion	39
8. References	40
9. List of figures	42
10. List of tables	43
11. Appendices	44

1. Introduction

1.1. Background, Motivation and Aim of Project

Modern cameras capture an image by exposing a digital sensor to the light from a scene through a lens. During an exposure, a sensor is unable to capture all information from some scenes because the levels of light hitting the sensor can be so varied from part of the image to another. Usually this is caused by sources of light directly in the scene, indirect sources such as windows and reflections, or by very dark parts of a scene such as shadows being cast over some objects. Some sensors have a greater range of light they are able to capture. An extreme example is provided in Figure 1 below.



Figure 1. Under and Over Exposed Image.

Some hobbyists or professionals overcome this by taking multiple photos of the same scene with different exposure levels, to capture the darker and lightest parts of a scene. These photos can then be combined/overlayed in software to produce a final image with all of the detail of the scene as what would be visible to the human eye observing the scene.

Often between the exposures, there will be some movement in the camera position, causing the images to be misaligned slightly such that they can't be overlayed to produce an image. The images then have to be aligned before they can be overlayed. There is existing software for this purpose, however it often fails, or is very slow in its computation. (See section 3.1 for

details). In this project, I propose a new algorithm (and software) to achieve the alignment faster and more reliably than existing software, by using the power of GPUs which can be used as massively parallel SIMD processing units that can far exceed the computational performance than today's CPUs and are present in almost all computers at time of writing.

1.2. Deliverables

The algorithm and software for the task of aligning multiple images will be the artefact for this project. Other deliverables will be listed below.

- Software and Algorithm requirements
- Research on existing commercial software
- Research on existing academic image registration algorithms
- Testing against requirements including detailed performance analysis

1.3. Constraints

- Limited access to a variety of hardware to test the GPU accelerated program on.
 - While this project is intended to be ran on standard consumer-grade hardware, wider testing should be done to ensure solid performance and compatibility over a wide range of systems.
- Time will be a large constraint on this project. Many sections will take a lot of research and experimentation
 - Learning C++
 - Learning the GPU framework (Likely OpenCL or CUDA)
 - Assessment of existing academic image alignment/registration algorithms
 - Testing and iterative performance improvements to the software.
- Gathering user requirements will be a challenge due to problems in finding other hobbyists to gather requirements and test the software.

1.4. Project Approach, Plan and Risk

The project plan can be seen with times in Appendix 4. Tasks will be detailed below.

1. Investigate current software and gather performance and usability metrics.
2. Contact hobbyists (target audience) to gather a list of usability requirements.
 - a. During this and until task 5, experimentation of C++ and the GPU framework will take place so that I can gain experience in the technologies, since I have no current experience.
3. Investigate current algorithms for image stitching and tone mapping, implement the stronger ones for performance testing and quality testing of the final image.
4. Using requirements and performance metrics from other software, generate a testing plan to ensure requirements has been met.
5. Design the new image stitching and tone mapping algorithms, based around OpenCL
6. Implement the above algorithm and iteratively optimise for performance as time allows. Testing for every version with improved performance or test success rate.
7. Validate the project over the user requirements
8. Review the new solution against existing solutions, and retrieve end user feedback. Analyse the project and the solution separately as to what could be improved, and what.

Risks table can be seen below.

Risk Description	Mitigation/Control	First Indicator	Severity and Likelihood /3
Collection of User requirements, Unable to gather sufficient requirements from users	Gather more potential users using online forums.	Requirements are bare and have little coverage	2-1
Falling behind due to Illness, Unable to work	Use contingency time to recover	Illness	1-1
Difficulty in understanding stitching or tone mapping algorithms	Seek help from supervisor and lecturers	Exceeding allotted time from the project plan for research of algorithms	2-2
Misunderstood C++ and OpenCL processes	Seek help from online tutorials	Unexpected results from program, falling behind deadlines	3-1
C++ Compiler troubleshooting issues	Seek help from supervisor	Unable to add new libraries, use functions due to C++ build complexity	2-2
In-efficient algorithm design	Use contingency time to develop a new algorithm.	Performance improvements from optimisation at implementation stage becomes stagnant in an unsatisfactory state	2-2
Loss of project due to hardware, software or environmental disaster	Keep incremental backups of development environment using off-site clone	Can have no indicator	3-1

Table 1. Risks table.

2. Literature Review

2.1. A brief overview of algorithm types

2.1.1 Single/Multi-modality methods

The topic of image registration is a very well studied, and many types of algorithms exist to align images, even when the images may not be from the same source, such as overlaying CT and ultrasound images. This is called multimodality image registration. (Apicella, Nagel, Duara, 1988, p. 414). This project will only be using data from one sensor, so only single-modality algorithms will need to be considered, however since the sensor parameters will be different for each exposure, some techniques may need to be borrowed when building the new algorithm.

2.1.2 Automatic/interactive methods

Automatic methods can align images based on only the image set itself and do not need human aid in aligning. Interactive methods usually require the user to find some common points in the image, which guide the algorithm through iterations to get a good alignment. (Gering, 2006) All of the software evaluated in the requirements did not require human interaction to get an alignment (although some software had the option of manually specifying points to aid the alignment), so another requirement of the software will be that the algorithm will be fully automatic. This is extra important for cases where you have many multiple images and manual alignment would be very time consuming.

2.1.2 Intensity/feature-based algorithms

The two main types of algorithms for image alignment are defined as Intensity or feature-based. Intensity based matches work by applying different transformations to an entire image, and measuring its alignment (also called fitness) at each translation, and applying new translations based on the previous to find a perfect translation. (Kappor, n.d.) Feature based matches work by finding distinct/unique sections of one image and then trying to find the same part in the second image. More detail on these in the next section.

2.2. Basic CPU VS GPU Architecture and Memory system

Modern CPUs have a small number of large, very fast, complex cores and a very complex memory caching system, comprising of multiple levels, usually 3, and sometimes up to 4 levels between main memory and the CPU's registers. (Kirsch, 2015). This is very good for single, or very lightly threaded sequential programs. This doesn't usually make a difference in to how programs are written for these platforms.

Alternatively, GPUs are comprised of hundreds or thousands of smaller and simpler ALU's with a much less cached memory layout, instead leaving this task to the programmers to pin important data to the faster on-die memory oppose to the slower (usually GDDR based) memory. There are some notable exceptions to this such as CPUs with integrated Graphics processors that use normal system DDR as its main memory instead of GDDR. Another notable exception to this is GPUs that use HBM (High Bandwidth Memory) where the memory chips have a much wider data bus than standard GDDR and are on the same substrate or interposer as the GPU die (Smith, 2015). These differences do not usually translate in to differences in code for the different platforms, however, but do change the characteristics in latency and throughput of the main memory available to the GPU.

For image registration, GPUs give promise of higher performance due to their massive increase in compute performance and memory bandwidth, and the very parrelisable nature of some of the image registration algorithms available today. Many researchers have ported existing algorithms and designed new ones for their specific use case and image properties (ie, single/multi-modality, Rigid/non-rigid, 2D/3D etc) (Fluck, Vetter, Wein, Kamen, Preim & Westermann, 2011). The rest of this review will be focused on creating a new one optimised for the specific aim of aligning multi-exposure images with small misalignment for use in multi-exposure HDR photography.

2.3 The Image Registration Recipe

Image Registration algorithms are built using 3 main sub-algorithms, and some optionally some preprocessing. (Shams, Sadeghi, Kennedy & Hartley, 2010) and (Crum, Hartkens, Hill, 2004) The 3 main sub-algorithms are the alignment measure, the translation function, and the optimiser. The roles of these are described in the image below.

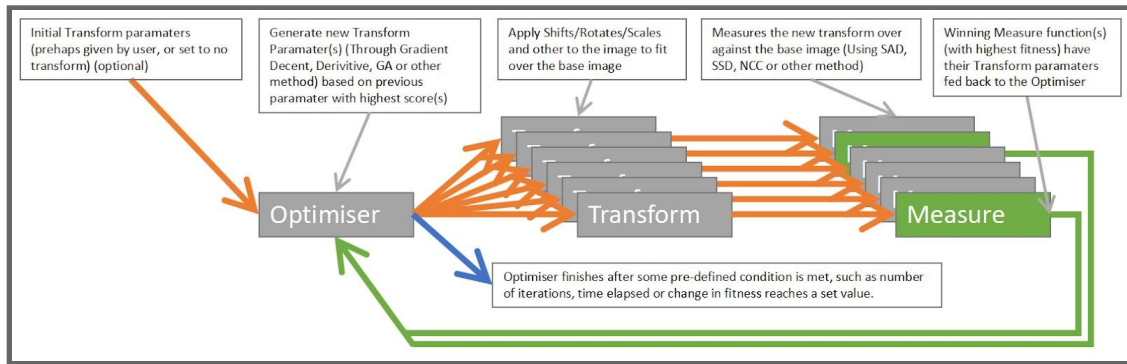


Figure 2. Basic Image Registration Algorithm

2.3.1 The (Alignment/Similarity) Measure

The (Alignment/Similarity) Measure is responsible for giving a score to a certain image transformation, indicating how aligned the current transform is. This can be called the fitness of the transformation. As discussed above, there are intensity and feature based methods of measuring the alignment. (Fluck, Vetter, Wein, Kamen, Preim & Westermann, 2011) found that (as of 2011, there are likely more as of writing) there was only one paper that implemented feature matching (Ruiz, Ujaldon, Cooper & Huang, 2009). This was done by taking sections of an image and using intensity measures to find local alignment of these sections. Other papers use intensity measures over the whole image to get better accuracy, and the increased parallelism this gives.

Intensity measure is the most common way of calculating the fitness of the transform, but can be split up in to different methods again. (Fluck, Vetter, Wein, Kamen, Preim & Westermann, 2011) briefly details the methods. SSD (sum of squared differences) and SAD (sum of absolute differences) is fairly self explanatory, and its cost can be easily calculated, see table below. These are good due to their low complexity, but suffer from not compensating for a change in brightness, contrast or exposure, so these may have to be ruled out if we can't compensate for the change in exposure before the alignment process is started. See 4.3 for details on this. NCC is the other method of interest, as it can compensate for the change in contrast in the images we will have to process. The final method is MI (Mutual information), which can accomodate for multi-modal registration. While this would also be able to cope with the non-fixed exposure of our images, because of its advances capabilities, it becomes more complex and slower to process, so it will be ignored from here onwards.

Key:

- n = number of pixels (in a 2 dimensional image)
- $x[n]$ = image1 pixel number 'n'
- $y[n]$ = image2 pixel number 'n'

Method	Formule	Rough Complexity	Θ Complexity
SSD	$\Sigma(x[n]-y[n])^2$	n minus, n indices (square), n addition	$\Theta(n)$
SAD	$\Sigma\text{abs}(x[n]-y[n])$	n minus, n absolutes, n addition	$\Theta(n)$
NCC	$\frac{\sum_{n=0}^{n-1} x[n] * y[n]}{\sqrt{\sum_{n=0}^{n-1} x[n]^2 * \sum_{n=0}^{n-1} y[n]^2}}$ <p>NCC formule: (Understanding Cross-Correlation, Auto-Correlation, Normalization and Time Shift, 2016)</p>	2n squares, n+1 multiplications, 3n additions, 1 indices (sqrt), 1 division	$\Theta(n)$

Table 2. Similarity Measure performance table

A quick note on NCC:

NCC is a way to compare the similarity between two time series (Understanding Cross-Correlation, Auto-Correlation, Normalization and Time Shift, 2016) and can work across multiple dimensions (Hii, Hann, Chase & Van Houten, n.d.). It has various different implementations due to its complexity, such as by using a fast fourier transform. (Hii, Hann, Chase & Van Houten, n.d.) showed that it's possible for different methods such as their "sum tables" to give over an order of magnitude quicker runtime as compared to using an FFT on their specific hardware.

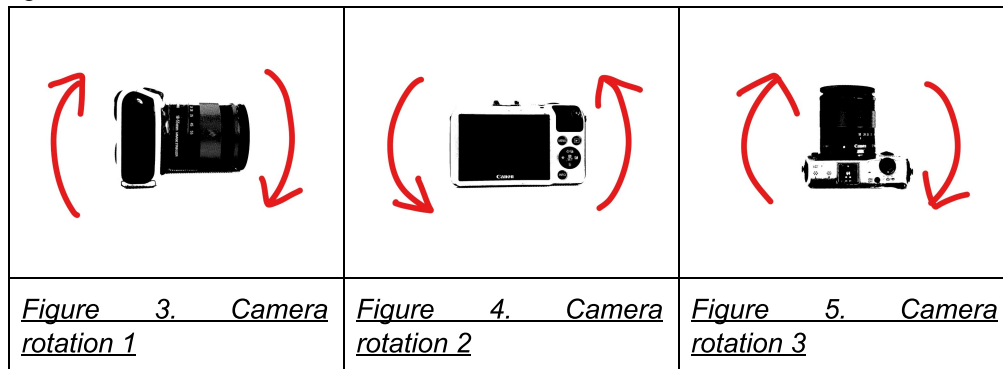
We can see from the above that SAD is the simplest and fastest operation, and NCC is the most complex. However this does not take in to account the time penalty for memory access. For SAD and SSD, the amount memory accesses is easy to see, but for NCC, since some values can be cached between calls, this is much harder to calculate, and varies depending on how its implemented. Thankfully, all of these algorithms are done in linear time complexity, so (assuming they all good at generalising the translation score) none of these are a bad idea to use, although maximum speed would be achieved using SAD if exposure could be compensated for prior ot alignment. See 4.4 for details on this.

2.3.2 The Transformer

The Transformer is responsible for applying a transform (rigid or non-rigid) to the an image over a static base image such that the similarity measure detailed above can calculate the similarity of the two images. The transformer may apply

different translations and warps to the image on any axis, depending on what's needed for correct alignment of the image. For example, it may move image one, 5 pixels to the left, 4 pixels down, rotate by 1 degree over the static base image. Non-rigid transformations can also contain scaling (sometimes called zoom) of an image size and shear of an image (Ashburner & Friston, n.d.), section 2.3. There are many techniques for non-rigid translations, and the parameters can get much more complex. Usually, they rely on multiple data points over an image, with its own independent translation which best describes the image deformation at its location. These are known as control points. Very simply put, this list of translations at the control points are then interpolated using various different methods such as B-splines (as used in computer graphics) to get a translation for each pixel in the image, at which point the image can then be translated and the intensity of the new pixels generated. Some methods change the translation at the control points based on the values of control points, and some methods such as the Demons method, try to emulate how a viscous fluid would deform, and the methods used can vary depending on the content of the actual image. (Crum, Hartkens, Hill, 2004), S143. The translations for this project will be very minor, and will be fully rigid, because the movement of the camera between exposures will be minimal.

It's likely that the user will not be able to keep a camera at a set rotation, as seen in figure 3, 4 and 5.



A user will also be unable to hold a camera in a constant x/y/z plane in 3D space, however this movement will not create any visible change in image, especially at longer distances between camera and subject, such as landscape photography, (a very common use case for multiple exposure HDR photography). This means the only translations we will have to be accounted for in the program are the three rotations mentioned above.

To see how these rotations change the output image, some sample images have been taken against a grid so that the distortion between exposures is clearly visible. By holding the camera of a wide (full frame equivalent) 24mm focal length (where the effects of lens distortion would be most prominent), and by adjusting its angle by around 2% (noticeable for the user, and unlikely that a user would knowingly change the angle between shots (using an exposure-bracketing mode) greater than this), we attempt to overlay these two images by only using x-y

translation and rotation to see if there is a sheer component needed for alignment. For clarity, the second image has had its colours inverted so any misalignment will be bright and obvious.

Results: Full Size images are available in the Appendices 1,2,3

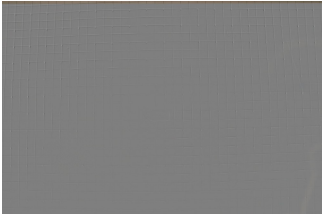


		
Settings: 18mm Translation: 9x, 50y, 0r See Appendix 1	Settings: 35mm Translation: 6x, 142y, -0.35r See Appendix 2	Settings: 55mm Translation: 67x, 243y, -0.37r See Appendix 3

Table 3. Image transform table

As you can see, by the basic x-y-r alignment noted above, most of the images are able to be aligned. The effects of lens distortion can be seen at 18mm, that introduce a non-uniform transformation across the image. This would require a non-rigid transformation. However, with the longer focal lengths, the lens has no distortion, and the images (35mm and 55mm) are aligned almost perfectly. It must be noted however that a sheer component is introduced in the alignment when trying to align images taken with a camera using Sensor Shift image stabilization. In most cameras however, this can be disabled.

2.3.2 The Optimiser

The Optimisers purpose is to minimise the amount calls to the transformation and measure parts of the algorithm, in order to find the optimal (or heuristic) solution (transform parameters) to the problem. The transform and measure can be thought of as a bounded function and the optimisers job is to find the global minimum/maximum of the function (Introduction to Optimizers, 2018), while doing as little calls to the function (transform and measure) as possible. This is a well-known problem that has been researched extensively, and is not inside the scope of the paper, but we will briefly give an overview of Gradient Descent due to its simplicity.

An exhaustive search for all except the most basic problems is almost impossible, due to either the size of the search space, or the complexity (and hence computation time) of the function (in this case, the transform and measure function).

Gradient Descent is one of the most basic algorithms, and one of the easiest to implement (except for an exhaustive search). It calculates the gradient of function, and uses that to estimate where function reaches its max/min value, and applies this

iteratively until it reaches either a perfect solution, or a minimum gradient change, a set number of iterations, or some other heuristic like time.

Genetic Algorithms applied as an optimiser are also worthy of a note here, as it lends itself to a large amount of function calls in each iteration (known as generation). However to limit the number of generations, the Genetic Algorithm would have to be customised a lot for this specific application.

3. Requirements and Testing Methodology

This section entails how success of the final product will be measured, and compared against current existing solutions. The role of the customer will be taken on by me, and several anonymous colleagues with an interest in photography. The aim of this project is to create a high performance, high efficiency algorithm for image registration, so the emphasis will be on performance requirements.

3.1 Performance Requirements

Multiple Exposure HDR photography follows the basic premise (as detailed in the introduction) of combining several standard dynamic range images and overlaying them. Therefore the only way to test a successful result from software was to create some sample images that would be used to test them. The sample images are available in appendix.

The next step in creating requirements for this project was to analyse existing software on our test images. The table below shows the times for the different sets of images using different software. I will be evaluating most popular existing software. Comparison of photo stitching software. Retrieved from (Comparison of photo stitching software, n.d.), along aside the opencv implementation for image alignment. This will help build some quantitative performance requirements for the project.

Notes:

- i2Align Quickcharge was excluded from the testing due to no trial option being available.
- Stitcher Unlimited was excluded from the testing due to no longer being sold or available for download.
- The OpenCV implementations was using the details as follows:
 - `int number_of_iterations = 5000;`
 - `double termination_eps = 1e-6;`
 - Using faster values would result in incorrectly aligned images. These were the fastest values that produced correct alignment.
- PTGui software has the potential to correctly compile a HDR image, however this only worked on image sets 3,4,5 and 6. On other test sets, it would instead would give the following error. "All source images were taken with the same exposure settings..."
- Values provided below up to 3 significant figures where available.
- Cells in red and underlined indicate a incorrectly stitched image or an image with a significant artefacts or warping, making the result unsatisfactory.

Device Hardware	# of images	Autopano Pro	PTGui ***	Easypano Panoweaver	Hugin for Panorama Tools	Desktop (1700x)	Tablet (m5-6Y75)	Samsung S7e ***	Canon EOS M1

Method	# of images	Tone mapping and image alignment	Tone mapping and image alignment	Tone mapping and image alignment Cubic Panorama	IMAGE ALIGNMENT ONLY Feature Matching 'Hugin's CPFind' Output/Stitch	IMAGE ALIGNMENT ONLY		In-phone HDR TONE MAPPING ONLY	In-camera HDR TONE MAPPING ONLY
						OpenCV (findTransformECC, MOTION_EUCLIDEAN)			
Image Set 1	3	7.87s	<u>~1s</u>	<u>~2s</u>	11.5s	27.3s	63.0s	<0.25s	8.40s
Image Set 2	4	22.18s	<u>~1s</u>	<u>~5s</u>	12.6s	51.3s	99.9s	<0.25s	8.30s
Image Set 3	4	<u>22.68s</u>	<u>~1s</u>	<u>~5s</u>	14.6s	254s*	401s*	<0.25s	8.50s
Image Set 4	3	2.33s	<u>~1s****</u>	<u>~3s</u>	5.2s	3.52s	5.05s	<0.25s	8.20s
Image Set 5 (No Exposure Change)	50	277s	~23s	<u>Failed</u>	819s	N/A**	N/A**	N/A	N/A
Image Set 6 (No Image misalignment)	8	34.0s	~3s	15.1s	N/A	N/A**	N/A**	N/A	N/A

Table 4. Software Performance Results Table

1. *Possible memory thrashing here, resulting in the excessive times.
2. **These tests were not run due to time constraints.
3. ***While the 'HDR' function on the Samsung S7e, its performance indicates that it is unlikely taking multiple exposures to generate a HDR image, when taking into account the relatively low computational power in comparison to a standard computer. I am also unable to find any technical documentation on what this specific 'HDR' function does. For these reasons I will be discounting its results.
4. ****PTGui for this test just set the output image to the first input image, since it could not get an alignment with the other images in the set.

Concluding these results, it appears that none of the softwares tested were able to stitch and tonemap all of the test image sets without any defects. It also shows that algorithm time is roughly inversely proportional to the chance of getting a satisfactory output; Of Course there are outliers. This helps set the performance requirements for the implementation.

Image Set	Fastest Software for Image Stitching and Tone Mapping a successful result	Time target
Image Set 1	Autopano Pro	7.87s

Image Set 2	Autopano Pro	22.18s
Image Set 3	Hugin for Panorama Tools	14.6s (image alignment only)
Image Set 4	Autopano Pro	2.33s
Image Set 5 (No Exposure Change)	PTGui	~23s
Image Set 6 (No Image misalignment)	PTGui	~3s

Table 5. Software Performance Requirements Target Table

3.2 Functional and Non-functional Requirements

The overarching requirements for this project are fairly simple, as the aim is to create some image stitching software that takes use of a GPU compute device for performance improvements over standard sequential or parallel CPU solutions that exist today. Basic requirements will come from analysis of existing software, and further requirements will come from anonymous interviews.

Requirements from existing software.

- The software must be able to stitch as many photos as the user requires to create the HDR image. Usually this value is 3, as this is what is used by in-camera “exposure bracketing functions” to take multiple photos of varying exposure automatically. However there is no reason to limit this, and the software investigated above can work with many more, so the new software should be capable of this also. We will test up to 8 photos in the examples, but should have the option to do more.
- The software should be able to work with up to 50Mpx images and beyond, as some cameras today are capable of creating images of this resolution. (Canon EOS 5DS R. n.d.) The size of the images created by the program should not be artificially limited as image resolution from cameras is likely to continue to increase. Non-professional consumer cameras such as found in phones are able to take photos of around 16Mpx.
- The software should be able to run on any OpenCL capable system, and have a CPU fallback algorithm if OpenCL is unavailable at run-time. The CPU algorithm must also be performant, and not render the program useless or inconveniently slow in cases where OpenCL is not available. (CUDA would be a viable alternative to OpenCL, however it’s not supported on as many machines, due to it necessitating a GPU manufactured by Nvidia. This would limit the potential install-base for the software.)
- All the existing software has the basic functionality expected from such a program, such as selecting images, previewing the result, saving.

- Due to time constraints, no GUI will be provided (apart from the preview). The program will be ran using a config file or a command line argument to specify input images.
- The images accepted must follow the output for a standard consumer camera
 - JPEG, RGB/RGBA channel information, 8-bit depth per channel, (24/32 bits per pixel)
- Most software tested has a task indicator, showing how much of the processing has been completed, the new software should match this in functionality.
- Most software tested has a variation on blending mode, which dictates how the final image is computed based on the aligned input images. This is sometimes referred to as tone mapping, and will be referred to it as tone mapping in this document. Different algorithms produce different styles and looks. A basic “average pixel” method must be implemented at least, with more algorithms being desirable.

Requirements from anonymous user interviews, that have not already been captured by the requirements listed above.

- Software must work on Windows 10
- Software must be fast to startup (<10 seconds)
- A live preview of image with sliders for tone mapping parameters
- Save as 8-bit jpeg or 10-bit jpeg
- Support full-screen preview
- Support a no-alignment mode for only-tone mapping functionality.

The above sections was used to build the functional and non-functional requirements lists below.

“Must Have” requirements

1. Merge at least 3 images
2. Merge images with a resolution of at least 16Mpx
3. Run on OpenCL enabled systems (Windows 10 will be used in testing)
4. Save the output image
5. Ability to specify input images, for example, via command line argument, or in a config file.
6. A basic “average pixel” tone mapping algorithm to merge images once aligned.

“Should Have” requirements

1. Merge at least 8 images
2. Merge images with a resolution of at least 50Mpx
3. Run on all modern computers (2 x86-64 CPU cores, 8GB RAM, no dedicated-GPU, Windows 10 will be used in testing)
 - a. Performance should be no more than 10x slower than the OpenCL implementation.
4. Preview the output image before saving
5. Contain an indicator of processing time or processing complete percentage.
6. A more advanced tone mapping algorithm with parameters to change style and look

7. Save images as 10-bit JPEG

“Would Like” requirements.

1. Merge beyond 8 images
2. Full GUI with live previews, and sliders for adjusting the tone mapping settings
3. Support full screen previews of images
4. Support a no-alignment mode for where image registration/translation is not required.

4. Design

This section relies heavily on details outlined in the Literature Review (section 2). Overall design can be seen in section 4.7, more detailed design is noted below in section 4.1 to 4.6.

4.1. A Quick note on Optimiser Speed

Improvements to the Optimiser will result in less calls to the transform and measure. The same improvements to an optimiser from one image registration will almost always apply to another algorithm. As detailed in section 2.3.2, there are many different types of optimisation and is a large topic in itself, so improvements to the Optimiser will be out of scope for this project.

4.2. Marriage of the Transform and Measure functions

In (Shams, Sadeghi, Kennedy & Hartley, 2010), it's proposed that by merging of the Transform and Measure functions can reduce computation time. In this case, the Transform function would not have to save each image's translation to memory, so that the Measure function can load it to calculate its fitness. If these functions were merged, the measure could be calculated at the same time as the transform was done, with the benefit of much less memory usage, as the transform results for each pixel are never needed to be saved, instead each pixels value can be summed to the measure result.

4.3. Pre-Compensating for Exposure

In (Fluck, Vetter, Wein, Kamen, Preim & Westermann, 2011), it's noted that the advantages of using NCC was that it was able to find similarities in images with varying brightness and contrast, unlike the simpler, but faster SSD and SAD methods. In this project, we will attempt to correct the images to be have the same brightness and contrast by altering the gamma of the image. This means while we have an extra fixed computational cost for each image, we can use the much simpler and faster SSD or SAD methods. This reduces the computational complexity of the algorithm. To calculate and adjust the image for gamma, we first find the average brightness of each image by sampling a subset of the image pixel intensity value. By passing these values through the gamma function, we can get a value for gamma (γ).

$$\text{Gamma} = \gamma$$

$$\text{Average image pixel intensity (between 0 and 1)} = \text{img}(\text{base}) \text{ or } \text{img}(2)$$

Image pixel intensity (between 0 and 1) = base_img[i] or n_b_img[]

$$\gamma = \log(\text{img}(\text{base})) / \log(\text{img}(2))$$

The non-base image is then passed forwards through the function using the gamma value, to get a new image with similar brightness and contrast characteristics as the base image.

4.4. The Measure Function

In section 2.3.1, SAD was suggested as the fastest similarity measure. Because difference in exposures of the images is already taken into account of in section 4.3, SAD can be used, and it's very high performance characteristics can be taken advantage of.

4.5. Image Subsampling

As the image alignment is iterative, the first few iterations have quite large differences in the translation parameters, which get refined over time. This means the full image detail is not necessary to get a rough measure of the alignment in the first few iterations. Therefore, by reducing image size, by subsampling it, we can reduce the cost of sending the image to the GPU, reduce the cost of the transform and measure drastically, for the lower accuracy iterations.

4.6. OpenCL's image2d_t location-based cacheable data type, and interpolation

OpenCL provides the programmer with several data types that abstract less of the hardware, instead exploiting the low-level access granted by the interface to increase efficiency of the hardware. One of these data types is Image2D (Other Built-in Data Types, n.d.) Image2D's advantages are that it can use hardware texture mapping units that are already built in to GPU devices and that have been for many years. These texture mapping units are optimised heavily in hardware for the specific task of loading and transforming textures (images), and the physical memory die that supports them is usually faster than the standard global memory (Sellner, J. 2017). Therefore using these to store the images when being transformed and compared against each other (transform and measure functions) should be much faster than using the standard global memory, and doing the transformations in software using the general purpose ALUs present on the GPU.

4.7. New Design

The new design will incorporate the above optimisations applied to the standard image registration algorithm from Figure 2 in section 2.3. Figure 6 displays the new algorithm pictorially. Basic pseudocode for the program is given below.

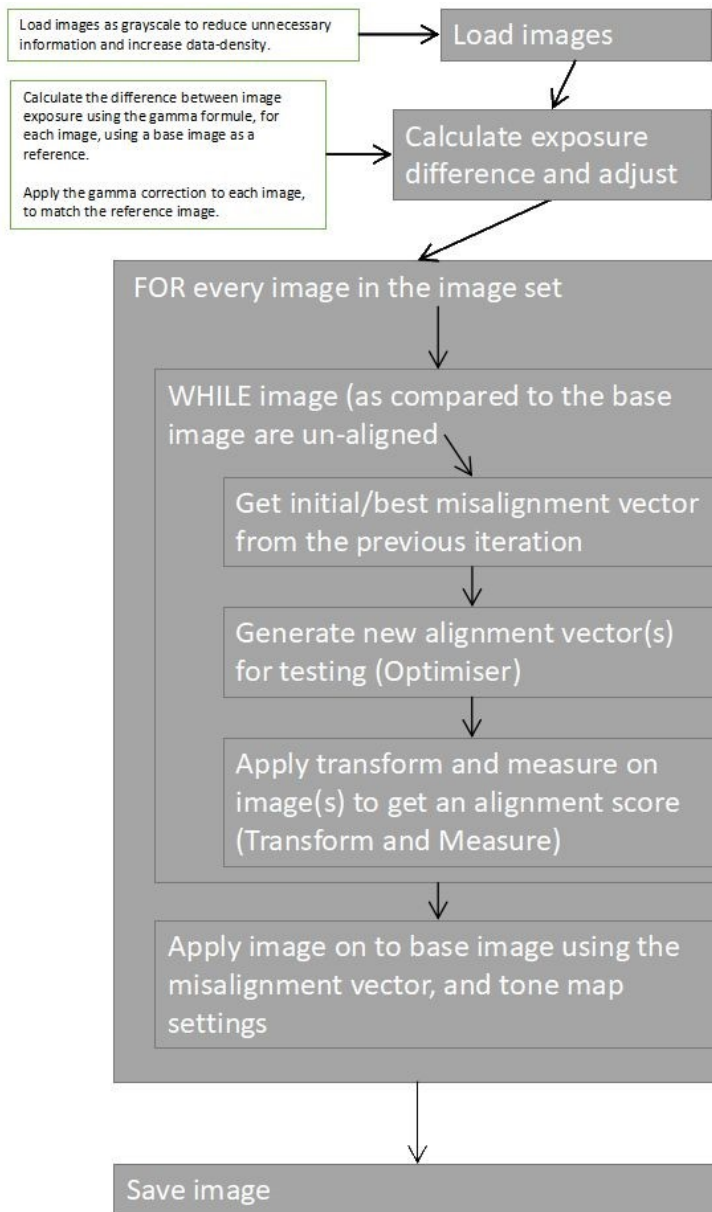


Figure 6. Full Program Flow

1. Load images as grayscale to reduce unnecessary information and increase data-density.
2. Calculate the difference between image exposure using the gamma formule, for each image, using a base image as a reference.
3. Apply the gamma correction to each image, to match the reference image.
4. LOOP (FOR every image compared to the base image)
 - a. LOOP (WHILE image as compared to the base image are unaligned)
 - i. Get misalignment vector (initially set at 0x, 0y, 0r)

- ii. Generate new alignment vectors to test, based on current alignment vector, and current image sample rate/resolution.
 - iii. Transfer grayscale images to the OpenCL device, then perform the transform and measure functions to generate an alignment score
 - iv. Pick the best alignment score, and get its corresponding misalignment vector.
 - b. Apply image on to base image using the misalignment vector and tone mapping algorithm.
- 5. Save Image

5. Implementation

5.1. The Development Environment, Language Choice...

For the implementation, C++ was used for its native OpenCL bindings and cross platform support. Python was considered as an option, but lacks in speed when compared to C++ and other 3rd Generation programming languages, and since the aim of this project is to create a fast and efficient HDR image alignment algorithm.

The development environment was Windows 10 using Visual Studio 2017 and its inbuilt compiler, using -O2 optimization flag. OpenCL version 2.0 was used.

5.2. A note on the development lifecycle

The intent for this project was to use Iterative development, because once a working project is established, each iterations focus can be on improvements to the algorithm. With a more classical/standard development model such as waterfall, which makes alterations to the code (such as efficiency improvements) a much more laborious. In practice however, a waterfall cycle was needed for the first section, as large sections of the program could not be tested properly until most of the program was built (for example, the optimiser can't be tested until measure function is built). Once the project had reached a state where its performance could be tested (and hence requirements could be tested), a iterative development cycle was used to port the existing transform and merge function to OpenCL, and used until the performance requirements had been met.

5.3. Initial Implementation

For the first version of the program, the initial transform and measure algorithms are written as CPU functions, to test that the logic works as intended. OpenCL programming requires much more time, and so CPU algorithms also allow testing of this basic logic quicker.

Since my experience with C++ and OpenCL before starting this project was non-existent, a large of the time allocated (See Appendix 4) was designated to learning of the language and tools. Initial experiments were as follows

1. Loading an array to the OpenCL device, and retrieving it
2. Converting an Image to an array, copy and retrieve from the OpenCL device.
3. Above but with basic distortion using the OpenCL kernel.

Once confident with C++ and OpenCL, initial implementation was started, and progressed in the following order, since the later parts of this list were unable to be tested

until the the supporting code was in place. Small details on each sections implementation are detailed below.

5.3.1 Load images using OpenCV and display

- a. Wrapper functions for converting between OpenCVs Mat object and a standard C++17 vector were created very early on to help keep the codebase clean. Several of these were implemented depending on the type of image.

5.3.2 Convert Images to grayscale (using an OpenCV function)

- b. This is a pre-existing function within OpenCV so no detail on this will be given here

5.3.3 Build image the brightness/compensation function

- c. This was comprised of two parts, by *using pseudocode from section 4.3*
 - i. get_image_average_brightness() Getting the average brightness, done by averaging a sub-sample of the images grayscale pixel intensity values. (Sample rate used is 1%, yielding a theoretical performance improvement of 100x as compared to a sample rate of 100%)
 - ii. normalize_brightness() Takes each image, uses the above function to get average brightnesses, then using the pseudocode below for calculating gamma corrections for brightness, assuming image bit-depth of 8 (0-255 intensity values).

```
GAMMA = log(pixel_out / 255) / log(pixel_in / 255)
pixel_out = 255 * ( pixel_in / 255 ) ^ GAMMA
```

5.3.4 Build Similarity Measure

- d. *Using SAD (sum of absolute differences) function.*
- e. This involves finding the magnitude of the difference between the intensity of each pixel, and calculating this as a percentage of possible difference. This is calculated using the pseudocode below, assuming image bit-depth of 8 (0-255 intensity values)

```
for pixel-index in image
    i = pixel-index
    diff = diff + positive(image2[i] - image1[i])
tme = diff #total measured error
tpe = image_size * 255 #total possible error
similarity-score = 100 * (tpe - tme) / tpe
```

5.3.5 Build Transform logic in to the Similarity Measure function

- f. Instead of transforming the whole image on to a new canvas, and storing the result back to texture memory, it should be quicker to transform each pixel and save the result of the faster register memory.
- g. Each pixel of one image is to be associated with a pixel from the other, given the translation vector. The translation vector is given as a shift of x and y along the x and y coordinates, and a rotation r.
- h. Since the image is stored as a one dimensional array (C++ std::vector), and translations will be occurring in a 2d space, some translation will need to be done between these two. Pseudocode is provided below with some example values for reference. The design incorporates the formule from (Rotation Matrix, n.d.) in section 1, where it is stated that $x' = x*\cos(\theta) - y*\sin(\theta)$ and $y' = x*\sin(\theta) + y*\cos(\theta)$.

```
int no_of_rows = 1024 #number of rows in image
int no_of_cols = 1024 #number of columns in image
int x = 5 #shift 5 pixels left
int y = -10 #shift 5 pixels up
int r = 3.5 #rotate 3.5 degree clockwise

for row in range(0 to no_of_rows-1) step 1
    for col in range(0 to no_of_cols-1) step 1
        image2x = col + x - (no_of_cols / 2)
        image2y = row + y - (no_of_rows / 2)
        image2x = image2x * cos(degree_2_radians(r)) - image2y *
sin(degree_2_radians(r)) + (no_of_cols / 2)
        image2y = image2x * sin(degree_2_radians(r)) + image2y *
cos(degree_2_radians(r)) + (no_of_rows / 2)

        if image2x and image2y are inside canvas
            image1_pixel_index = col + row * no_of_cols
            image2_pixel_index = image2x + image2y * image_cols

    finally: apply Similarity Measure using SAD
```

5.3.6 Build function to merge two images based on the translation

- i. Since the function above already translates and compares the two images, it can be easily adapted to merge full colour images.
- j. The current function would only be able to add two images with equal weighting. If this function were ran in a loop, with each subsequent image (image2) being added to the previously merged image (image1), each subsequent image will have more weighting on the resulting final merged image. To fix this, the new image (image2) needs to have its intensity scaled based on how many images have been used to compose the base image (image1). See Figure 7. Pseudocode is provided below with some example values for reference.

```

iir = #image_intencity_ratio, which is the number of images used in the
creation of imagel
for each pixel in image
    for each colour in pixel
        Image[colour] = (imagel[colour] * iir + image2[colour]) / (iir + 1)

```

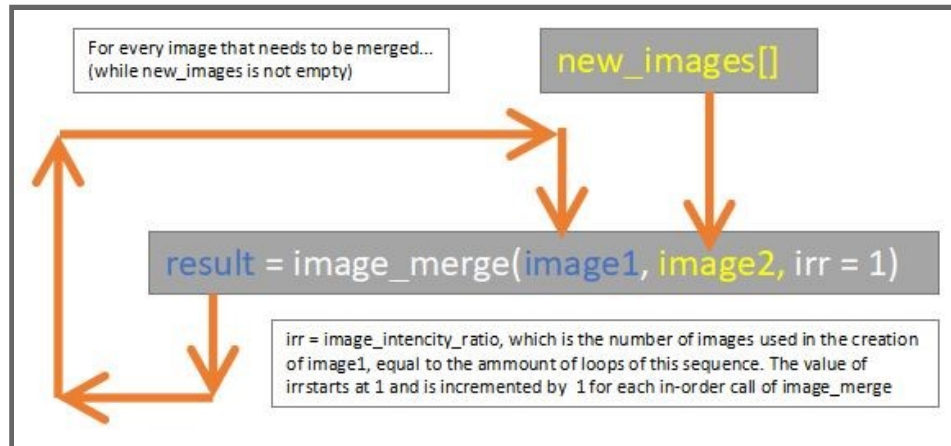


Figure 7. The image merge function (*render_image()*)

5.3.7 Build Optimiser (to calculate a list of new misalignment vectors to measure)

- k. As stated previously, details and optimisation of the optimiser is out of scope for this project, but some light details are described below for clarity.
- l. The optimiser takes the current best (or starting translation) along with the amount of allowed new translations to try, and creates an array (C++ `std::vector`) of new translations. It does this by creating as many as allowed evenly distributed between the two boundaries, which are determined by the previous translation, and the previous iterations resolution/precision (resolution in this context meaning granularity, not size). For example, if the previous iterations translation was $x=10$, $y=-25$, $r=0$, with a resolution/precision of 5px, and 1 degree, then the new bounds for testing are in from $x=5$, $y=-20$, $r=-1$ to $x=15$, $y=-30$, $r=1$. This is applied until the resolution/precision reach 1px and 0.05 degree, although these can be changed to optimise for accuracy of the alignment vs speed of the alignment.

Although the optimiser for this project has three variables to SAD measure algorithm for, it is explained below in a 1 dimensional space for simplicity.

In Figure 8, the starting optimiser algorithm starts between the bounds 0-5, with a resolution/precision of 1 (the blue lines). It finds the best result with its resolution/precision of 1 which is the orange line, at 2. It then creates it's new bounds, with 2 in the middle and using the previous iterations resolution/precision of 1 to create the new bounds at 1-3 (in green). The

algorithm can then be called iteratively until a maximum resolution/precision is reached. In this example, each iteration is allowed to have 10x the resolution/precision of the previous iteration, which is why the next best result (yellow) reaches the value of 2.7 with a resolution/precision of 0.1.

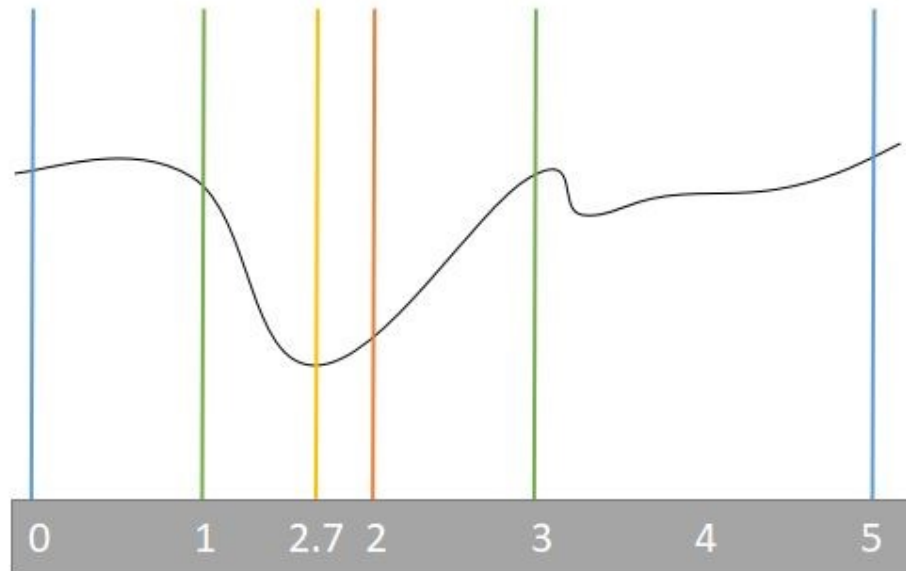


Figure 8. The Optimiser viewed

5.4. Porting to GPU using OpenCL and basic optimisation

C++ has native bindings for OpenCL, so once the programmer is familiar with the environment, adding OpenCL support should be simple, however in practice, the lack of tutorials and examples made this harder than first anticipated. More on this in section 5.6. Since OpenCL is designed to run on all types of SIMD hardware, the majority of which don't share memory with the host CPU, memory management is a little tedious. OpenCL code is based on a subset of the C language, meaning a lot of the data types and data structures available in C++17 are unavailable on the OpenCL. OpenCL was designed in this way because targeting a large feature set with more complex instructions will limit the amount of devices capable of running OpenCL code. OpenCL has a base set of requirements that are required from a hardware device for it to be called OpenCL compliant, (sometimes referred to as in-specification or in-spec). OpenCL allows device vendors to add capability to their devices without going out of spec, by adding optional extensions to the OpenCL language, such as double precision floating point operations. (Extension, n.d.) Because this project is intended on targeting most/all OpenCL capable systems, the amount of extensions used will be kept minimal.

The function in this project that is being targeted for OpenCL as discussed in section 5.3.5 is the transform and measure part of the registration algorithm. In my implementation,

this is called `get_greyscale_image_difference_CPU()`, which will be renamed as `get_greyscale_image_difference_GPU_kernel()`.

This function takes in a translation vector, and a read-only memory reference the two images. Since tuples are not supported by OpenCL, three separate arrays are required for the translation parameters. (A two dimensional array could also work). The only output required from the transform/measure function is the measure score itself, as the optimiser will keep a copy of the translation vector used to get each measure score. As shown in section 5.3.5, the inner loop of this function consists of many integer operations, and a few floating point operations. Getting the smallest data type capable of computing and storing the result is important, especially on GPU devices, because the difference in performance between the datatypes is greater than that of a CPU. This is especially visible when looking at single vs double precision operations on most consumer GPUs. (Comparison of NVIDIA Tesla/Quadro and NVIDIA GeForce GPUs, n.d.) Since float is the smallest standard data type available for OpenCL that supports decimal parts, this will be used for computation of the transformation. `uchar` will be used for the storage of each image pixel, and compute of the difference, since the max value of the possible result is 255, which matches the max value that can be stored in a `uchar`. The result will be stored as a unsigned long, since the next smallest standard data_type is an unsigned int, which is incapable of handling images over roughly 16Mpx. This is because integer can store numbers up to 4,294,967,294. Divide this number by a max possible alignment score per pixel of 255, which gives 16,843,008 pixels. The requirements dictate images of over 50Mpx to be computed, so a larger data type is needed (unsigned long).

OpenCL exposes different types of memory for variables to be stored in, these are called global (and a variation of this called constant), local and private. (DarkZeros 2017, August, 2). As a follow on from section 2.2, “Global” memory is visible by all threads, and can be read and written to. The drawback is that access is very slow, as this memory is usually on physical GDDR or HBM type memory for a GPU. The “Constant” memory is a small portion of read only memory that is cached from global memory, and like “global”, it is available to all threads. “Local” memory is faster than “Global”, is read/write capable, but only accessible from threads inside the same work group (block) as it was declared in. “Private” memory is only available to the thread it was declared inside, is read/write capable, but much faster than every other memory type. In the transform and measure function, the only memory to be shared across all threads is the two image arrays, the list of translations, and the output array. All other variables can be declared as private.

5.5. Further Optimisation for OpenCL

The most obvious step in optimisation is to remove calculations from the inner loop which remain constant through each iteration of the loop. The most obvious example of this in the transform/measure function is the calculation of the new pixel index when a rotation is applied. As the in 5.3.5 shows, the statements `sin(degree_2_radians(r))` and `cos(degree_2_radians(r))` are used twice each in a single loop iteration. This means they can be calculated once out of the loop, and stored in fast private memory, and called whenever

needed. This saves 4 sin/cos operations, 4 global memory lookups (r) and a 4 `degree_2_radians()` calls per iteration. This improvement nearly doubles performance, since the lookup and sin/cos operations are very time costly.

When attempting to generalise an image, a full sample of the entire image is often not needed. As explained in the optimiser in section 5.3.7, each iterations will have a different precision. For smaller precision, smaller resolution samples are needed. This has the advantage of not having to transfer full size images to the OpenCL device, which is often very costly, and the advantage of reducing the amount of computations needed from the OpenCL device. At a precision of $\pm 10px$, the image required for storage/computation is $1/10^2$ the size (1%). This performance improvement delines to nothing as precision reaches $1px$. However, it may still be possible to sample only a subset of the full image, as long as the translation function could still map these pixels to the exact location required on the base image. A sample rate of 50% could yield up to a 2x theoretical performance improvement. In testing, the sample rate was set to 25%, to give a theoretical 4x performance improvement, although actual performance improvements seen were around 3x. The table below shows performance improvements per sample rate. The performance improvement isn't linear due to overhead from creating the GPU threads, transferring of the memory, setup of the main loop in the GPU kernel, and calculations that take place outside of the main loop in the GPU kernel.

Sample Rate (%)	Theoretical Performance	Observed Performance Increase
100	1x	1x
25	4x	3.0x
11%	9x	4.3x
6%	16x	5.5x
4	25x	6.1x

Table 6. OpenCL image sample rate table

Also visible in the pseudocode in section 5.3.5 is the calls to the translation vectors. These are implemented in Constant integer type arrays, and are accessed once per pixel in each thread (in the innermost nested for loop). By declaring a new temporary private integer type variable that can be used to store the specific translation vector for the thread, access should be faster, since each thread now only has to access its private memory once per loop, instead of shared constant region of memory once per loop. However when tested, performance improvements were not seen, likely due to the OpenCL optimiser doing this work already by caching these variables in a private memory.

Ideally, iterative development would have stopped once performance requirements were met, however in practice, development was forced to end early due to time constraints.

5.6. Implementation Issues and Challenges

5.6.1 Performance using the C-style Array

As detailed in section 4.6, the `image2d_t` can be used for storing the two images, and providing the performance increase. However, there were some troubles during implementation, which means it wasn't implemented until very late in to the development process. This is mainly due to a lack of understanding of the OpenCL `image2d_t` type, and a lack of examples available for C++. More on this in section 5.6.2.

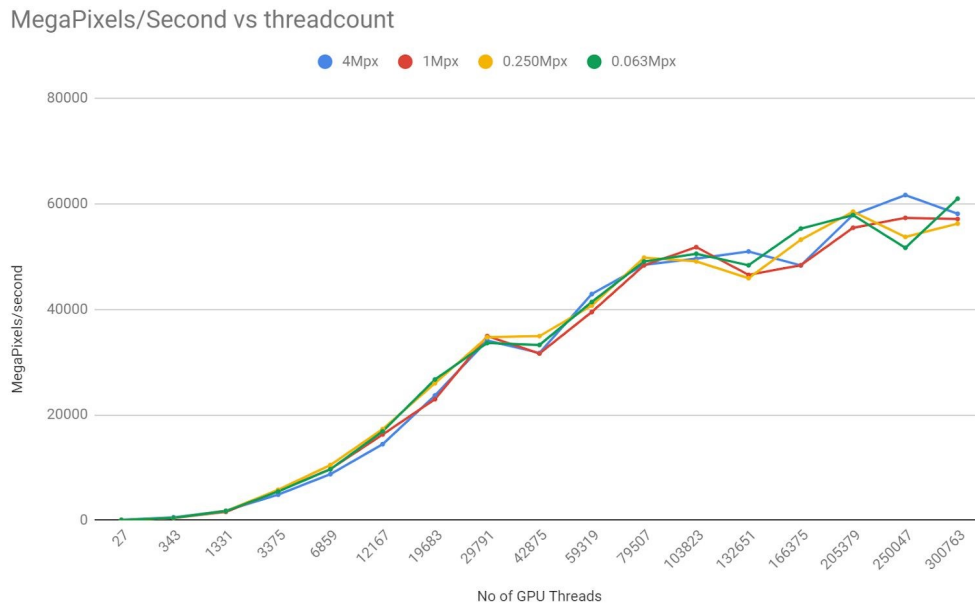


Figure 9. Performance of OpenCL kernel on Radeon R9 290, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays

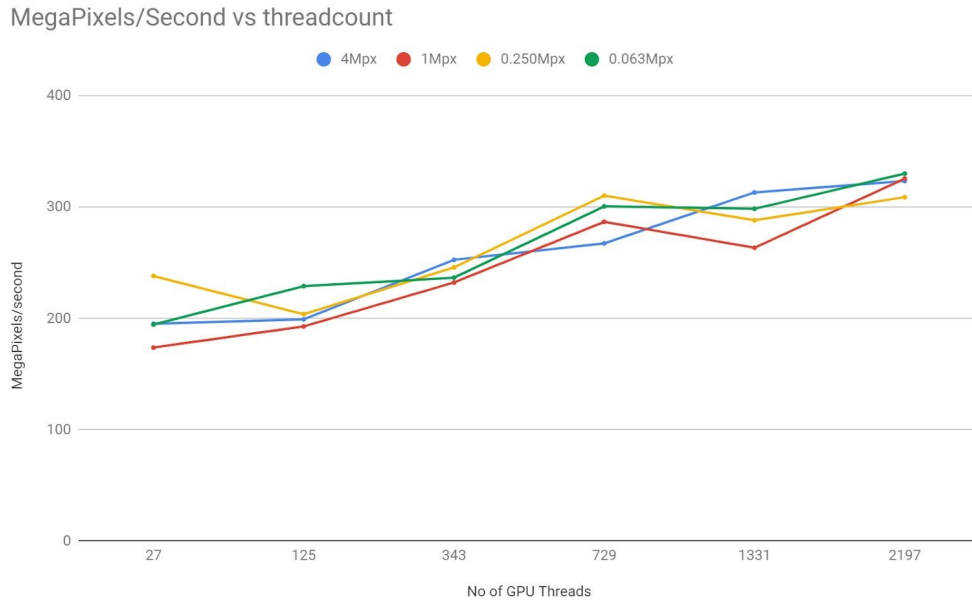


Figure 10. Performance of OpenCL kernel on Intel HD Graphics 515, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays

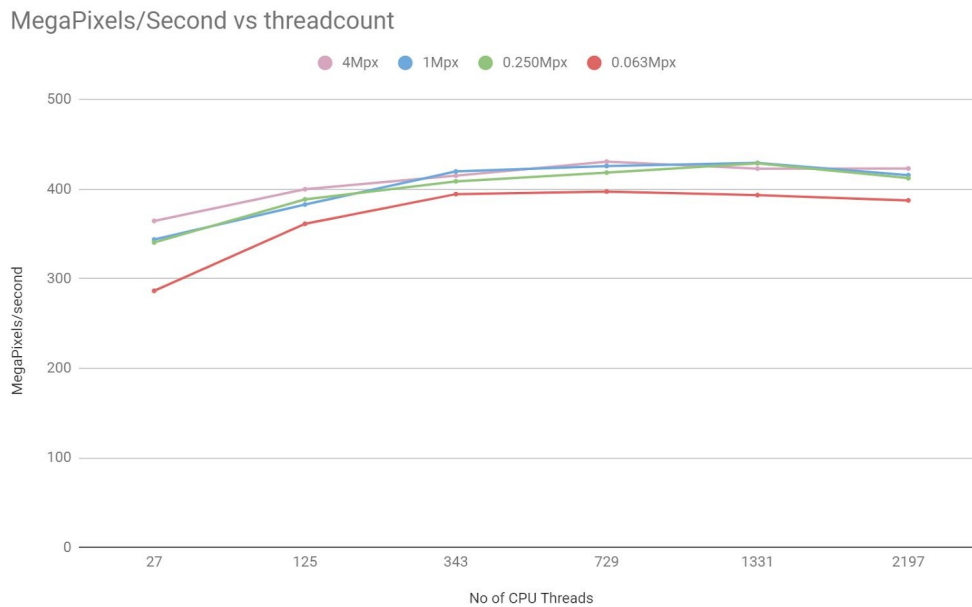


Figure 11. Performance of OpenCL kernel on AMD Ryzen 1700x, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays

An interesting challenge when using large (newer/faster many-core) GPUs, was the amount of parallelism needed to exploit the full potential of the GPU. In testing, figure 9, 200,000 threads for this specific workload was required to achieve maximum performance from the Radeon R9 290 GPU, with 2/3rds of that

performance being available at about 50,000 threads. Each line represents a different image size that the device is working with. This is to show possible memory bottlenecks on larger datasets (images). On a much smaller device, such as the Intel HD Graphics 515 (see figure 10), peak performance was observed at around 1000 threads, with 2/3rds of that performance being available at just 27 threads. This demonstrates the large difference in performance characteristics between OpenCL programs, and the importance of designing the algorithm to be parallel enough for the largest GPU devices. Since OpenCL is designed to target a wide variety of SIMD devices, a CPU device is able to be targeted also. For experimentation, the AMD Ryzen 1700x CPU was tested in the same way as the two GPU devices. Results below in figure 11 show that many less threads are required to achieve peak performance, because the CPU has many fewer SIMD processing units than a typical GPU, even when taking in to account modern AVX instruction sets; Peak performance is observed at just 343 threads, while 2/3rds of that performance is available at just (and looking at the trend of the curve, possibly before) 27 threads.

5.6.2 Performance using the Image2D data type

Using the image2d_t data type results in a small speedup over using the standard C array in testing, see figure 12. This was a surprisingly small performance improvement from its theoretical advantages. This is because the image data type can use the fast texture memory and dedicated parts of the GPU core for this task.

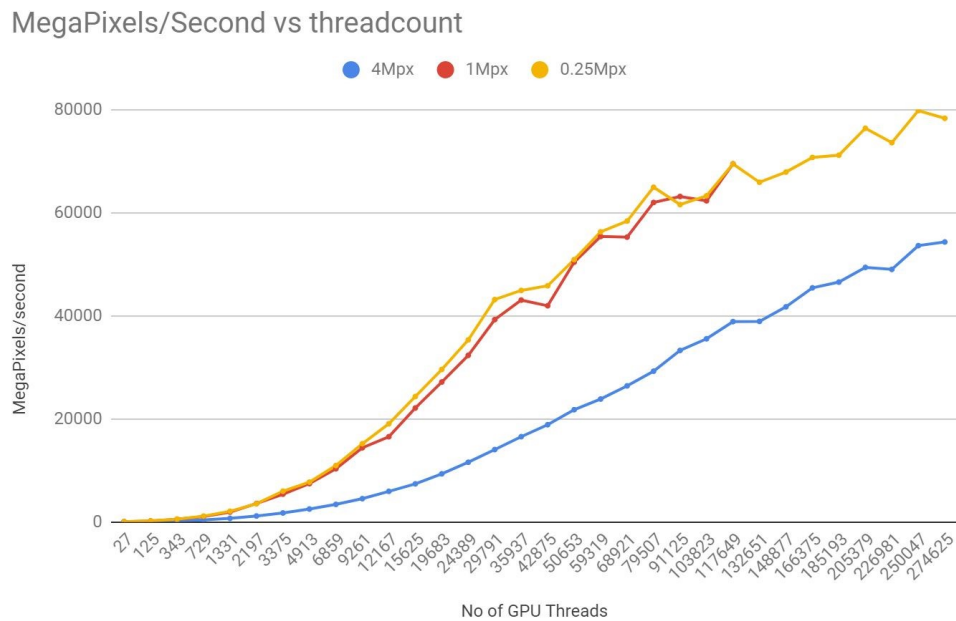


Figure 12. Performance of OpenCL kernel on Radeon R9 290, measured in Megapixels/second vs number of simultaneous GPU threads, using image2d

5.6.2 Other Assorted Issues

Another major issue in development was an intermittent hard crash that was experienced only in systems with dedicated GPUs (intel integrated graphics systems are immune in testing), and when loading images over around 4Mpx to the GPU, when running Windows 10. (Windows 7 systems are immune in testing) This hard crash would vary in severity, however would most commonly result in a crash of the display driver, which was sometimes recoverable, and on other systems, required a power cycle. This remained a large problem throughout the development project because it's near impossible to debug with the nature of the crash, the limited time and resources available on this project, and the lack of guidance and documentation on such issues available in the OpenCL programming guides, and the wider general internet. Initial experimentation showed that if the GPU kernel code is set to not return any results resulting from the computation of any part of the large C style array, the issue becomes no longer present. As soon as the GPU kernel has to send data back to the host where the data was created in part by the large C style array, the issue returned. This problem was partially solved in the last iteration on development, by sampling only a small section of the image, (leading to less work in each thread, see Table 6 in section 5.5). This could be due to a timeout happening on long-running GPU threads on some GPUs, and would explain why this doesn't happen with multiple smaller threads instead.

More advanced tone mapping algorithms were not implemented due to time constraints. A basic "average" based implementation is included, as this is one of the more common algorithms, easy to implement, and shows the image alignment function clearly in the rendered images.

6. Testing and Evaluation

6.1 Performance Requirements

Performance requirements were tested during iterative development of the project, as explained in section 5.2. Therefore, by the end of iterative development, all of the performance requirements should have been met, however due the deadline for this project, and a too-small contingency time buffer, performance requirements fall short in some tests.

Notes:

- Values provided below up to 3 significant figures where available.
- Cells in red and underlined indicate a incorrectly stitched image or an image with a significant artefacts or warping, making the result unsatisfactory.
- Program Outputs can be seen in Appendix 5-9

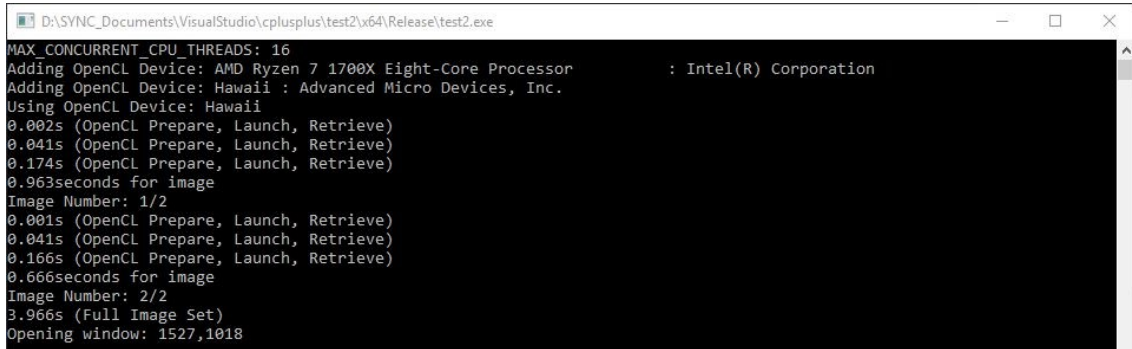
Image Set	Fastest Software for Image Stitching and Tone Mapping a successful result	Time target	Time result	PASS/FAIL (% faster than target)
Image Set 1	Autopano Pro	7.87s	3.93s	PASS (200%)
Image Set 2	Autopano Pro	22.18s	17.7s	PASS (125%)
Image Set 3	Hugin for Panorama Tools	14.6s (image alignment only)	<u>18.8s*</u>	FAIL (77.6%)
Image Set 4	Autopano Pro	2.33s	1.49s	PASS(156%)
Image Set 5 (No Exposure Change)	PTGui	~23s	255s	FAIL (9.02%)
Image Set 6 (No Image misalignment)	PTGui	~3s	N/A**	N/A*

Table 7. Software Performance Requirements Results Table

1. *Image was not aligned correctly. From analysis of the output image, it seems that there is another type of translation in the image set that was not accounted for in this implementation, such as sheer. A preview of this is available in appendix 7.
2. **Since this implementation has no advanced tone mapping functions, this feature

can't be implemented.

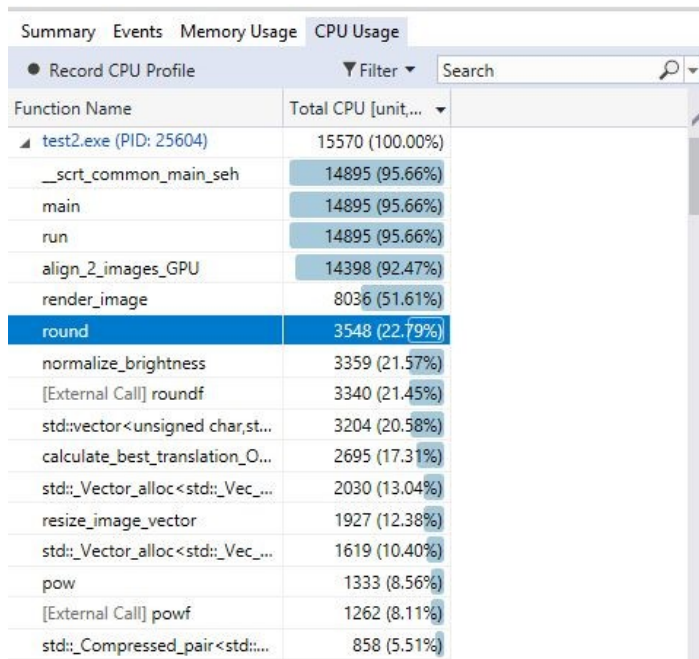
In all of the tests run, performance is better in three of the tests, and worse in the other two tests, one of which failed. In comparison to any single software from section 3, the new software is able to match the best software for most image sets correctly aligned from the test sets, and is able to do this in a faster time on more of the image sets than the best software. For alignment, these results show promising options for GPU acceleration of image registration in this use case.



```
D:\SYNC_Documents\VisualStudio\cplusplus\test2\x64\Release\test2.exe
MAX_CONCURRENT_CPU_THREADS: 16
Adding OpenCL Device: AMD Ryzen 7 1700X Eight-Core Processor : Intel(R) Corporation
Adding OpenCL Device: Hawaii : Advanced Micro Devices, Inc.
Using OpenCL Device: Hawaii
0.002s (OpenCL Prepare, Launch, Retrieve)
0.041s (OpenCL Prepare, Launch, Retrieve)
0.174s (OpenCL Prepare, Launch, Retrieve)
0.963seconds for image
Image Number: 1/2
0.001s (OpenCL Prepare, Launch, Retrieve)
0.041s (OpenCL Prepare, Launch, Retrieve)
0.166s (OpenCL Prepare, Launch, Retrieve)
0.666seconds for image
Image Number: 2/2
3.966s (Full Image Set)
Opening window: 1527,1018
```

Figure 13. Example Software Output (Image set 1)

By further analysis of the console output from running the program (see figure 13), we can see that only a small section of the execution time was used by OpenCL in execution alignment in the image (Only 51%).



Function Name	Total CPU [unit,...
test2.exe (PID: 25604)	15570 (100.00%)
_scrn_common_main_seh	14895 (95.66%)
main	14895 (95.66%)
run	14895 (95.66%)
align_2_images_GPU	14398 (92.47%)
render_image	8036 (51.61%)
round	3548 (22.79%)
normalize_brightness	3359 (21.57%)
[External Call] roundf	3340 (21.45%)
std::vector<unsigned char,st...	3204 (20.58%)
calculate_best_translation_O...	2695 (17.31%)
std::_Vector_alloc<std::_Vec_...	2030 (13.04%)
resize_image_vector	1927 (12.38%)
std::_Vector_alloc<std::_Vec_...	1619 (10.40%)
pow	1333 (8.56%)
[External Call] powf	1262 (8.11%)
std::_Compressed_pair<std::i...	858 (5.51%)

Figure 14. CPU Profiling of new software (Image set 1)

As figure 14 shows, a lot of the time was taken up by the `render_image()` function, which takes in the two image and its alignment vector, and renders a new image with tone-mapping. The round function name is the part of the `render_image()` function which results in the poor performance. This is responsible for rounding the coordinates that the translation vector gives when applied to an image. Since GPUs have texture units which can handle interpolation like this natively, this would be a good candidate for OpenCL implementation in future versions of this software. Another function which consumes a lot of CPU time is the `normalize_brightness()` function, which is explained in section 5.3.3. This is because each pixel needs to have the computational expensive formula applied to it to adjust the gamma of each pixel. Since this is another embarrassingly parallel task, it becomes a good candidate for either a multithreaded CPU or OpenCL based re-write.

6.3 Requirements Testing

Requirements from section 3.2 give this table, with expected results.

Test Num and Importance (Must/ Should/ Would-like requirements)	Test Description	Test Input(s)	Expected Output(s)	Actual Outputs	PASS/ FAIL
1m, Must	Merge at least 3 images	Image Set 1	Merged image	Appendix 5	PASS
2m, Must	Merge images with a resolution of at least 16Mpx	Image Set 2	Merged image	Appendix 6	PASS
3m, Must	Run on OpenCL enabled systems (Windows 10 will be used in testing)	OpenCL capable system	Program uses OpenCL functions over CPU fallback functions	Program Uses OpenCL functions as indicated in the console output	PASS
4m, Must	Save the output image	Image Set 1	Saved image as "New Software Output.jpg"	Saved image as "New Software Output.jpg" see Appendix 5	PASS
5m, Must	Ability to specify input images, for example, via command line argument, or in a config file.	Move images to be merged in to folder called "source"	Merged image	Saved image as "New Software Output.jpg" see Appendix 5	PASS
6m, Must	A basic "average pixel" tone mapping algorithm to merge images once aligned.	Image Set 1	Merged image	Saved image as "New Software Output.jpg" see Appendix 5	PASS
1s, Should	Merge at least 8 images	Image set 5	Merged image	Saved image as "New Software Output.jpg" see Appendix 9	PASS
2s, Should	Merge images with a	Image set 7	Merged	Saved image as	PASS

	resolution of at least 50Mpx	(New set)	image	“New Software Output.jpg” see Appendix 10 (Compute time of around 6.3s, 23 seconds required per image for tone-mapping and saving)	
3s, Should	Run on all modern computers (2 x86-64 CPU cores, 8GB RAM, no dedicated-GPU, Windows 10 will be used in testing)	Windows 10 VM with details as described was created, and Image Set 1 was ran as input	Merged Image using CPU algorithm	Crash due to missing OpenCL dll file.	FAIL*
3s.a, Should	Non-OpenCL performance should be no more than 10x slower than the OpenCL implementation.	Manually specifying CPU-Only functions, Image set 1	Merged Image using CPU algorithm within 39.3s	Saved image as “New Software Output.jpg” Image is bitwise identical to Appendix 5, execution time of 9.05s (4.3x slower)	PASS
4s, Should	Preview the output image before saving	Image set 1	Preview of Appendix 5	Preview of Appendix 5	PASS
5s, Should	Contain an indicator of processing time or processing complete percentage.	Image set 1	GUI or Command line indicator of progress.	Message for each image passed, with time to process each image included	PASS
6s, Should	A more advanced tone mapping algorithm with parameters to change style and look	N/A**	N/A**	N/A**	FAIL
7s, Should	Save images as 10-bit JPEG	Image Set 1	Copy of output image saved as 10-bit JPEG	N/A**	FAIL
1w, Would like	Merge beyond 8 images	Image set 5	Merged image	Saved image as “New Software Output.jpg” see Appendix 9	PASS
2w, Would like	Full GUI with live previews, and sliders for adjusting the tone mapping settings	N/A**	N/A**	N/A**	FAIL
3w, Would like	Support full screen previews of images	N/A**	N/A**	N/A**	FAIL
4w, Would like	Support a no-alignment mode for where image registration/translation is not required.	N/A**	N/A**	N/A**	FAIL
Total: 18					Total PASS:

					12/18
--	--	--	--	--	-------

Table 8. Software Functional/Non-Functional Requirements Results Table

1. * To fix, a separate executable version of the program could be included with none of the OpenCL code as requirements. This would allow users to select OpenCL or non-OpenCL code when running the program manually. There are more intuitive ways to do this.
2. **Feature(s) not implemented (see below)

As shown above in table 7, only 12/18 of these requirements have been met. This was due to time constraints on the project given my starting experience in C++ and OpenCL, and the limited contingency time in my project plan (see appendix 4). For example, processing of 10-bit images for test 7s would require alterations to every function that computes the transform, measure, tone-mapping and render functions, since all of the compute is only calculated to a precision of 8 bits for efficiency. Changing this would require functions to handle multiple data types, to keep optimum efficiency when working with 8-bit depth (per channel) images. A GUI would have required more time spent learning C++ GUI frameworks, and would have meant other more important features would have been missed. Tone mapping features would have required less time to implement than GUI or multi-bit depth images (such as the 10-bit images required for test 7s), due to the smaller complexity in this task, however the architectural design of the software becomes an issue. As these features were not taken in to account properly in the planning stage, a possible implementation becomes harder and messier than necessary.

Deadlines set by the project plan were missed frequently throughout the project, due to miss-estimations in the amount of time required for learning C++ and OpenCL. This can be seen clearly in appendix 12. Experimentation with OpenCL was started much earlier in the process via recommendation of the project supervisor. This because extremely important since this took much longer than expected, and stopped the project from falling too far behind. This in turn caused the literature review to start much later in the project timeline, which in turn caused development to take longer since these tasks were done simultaneously. Due to other commitments from other units at university, in early April, there was no progress on this project. This caused the Validation and Evaluation to be rushed at the end of the project. In future, more time will need to be set aside for learning new tools and technologies when undertaking a project of this size. The included contingency time helped me to catch up on the project towards the end when progress was falling far behind. In future, some contingency time in the middle of the project timeline could be beneficial.

7. Conclusion

In this document, a new software was proposed that would use OpenCL to accelerate the process of aligning and merging images for use in multi-exposure HDR image composition. Existing software was analysed to create a benchmark for the new software to be compared against. In testing, the software was able to achieve a correct alignment in 4 out of 5 tests, faster than any other software tested in 3 out of 5 tests by using OpenCL to exploit the power of GPGPU compute. It's shown here that by designing a new algorithm specific for this particular to this use case (non-exposure matched images, translate and rotate transformations only), large performance improvements can be seen by exploitation of OpenCL, for the image alignment portion of the problem.

As previously noted, due to time constraints, tone-mapping was not implemented. Software for this already exists that takes advantage of GPUs, with very impressive performance characteristics (Fastvideo, 2019), showing that this is an excellent candidate for an OpenCL implementation. This along with some of the features/improvements discussed at the end of section 6.1 would make for good further work for this project. Other avenues to research would include adding more translation options to the transform function, such as sheer and scale translations.

8. References

1. Apicella, A., Nagel, J. H., & Duara, R. (1988, November 4-7). Fast multimodality image matching. *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, 1, 414-415.
<https://dx.doi.org/10.1109/IEMBS.1988.94584>
2. Gering, D, T. (2006). *U.S. Patent No. US7693349B2*. Washington, DC: U.S. Patent and Trademark Office.
3. Kapoor, A. (n.d.). *Registration Methods In Multi-modality Imaging*. [Presentation Slides]. Retrieved from
<https://www.aapm.org/meetings/amos2/pdf/49-14439-38613-588.pdf>
4. Kirsch, N. (2015). Intel Core i7-5775C Broadwell Processor Review. [Review of the central processor *Intel Core® i7-5775C*, Intel®]. Retrieved from
https://www.legitreviews.com/intel-core-i7-5775c-broadwell-processor-review_166875
5. Smith R, (2015). The AMD Radeon R9 FuryX Review: Aiming For the Top. [Review of the graphics processor *AMD Radeon® R9 FuryX*, AMD® Radeon Technologies Group]. Retrieved from
<https://www.anandtech.com/show/9390/the-amd-radeon-r9-fury-x-review/6>
6. Fluck, O., Vetter, C., Wein W., Kamen, A., Preim, B., & Westermann, R. (2011). A survey of medical image registration on graphics hardware. *Computer Methods and Programs in Biomedicine*. 104(3). e45-e57.
<https://doi.org/10.1016/j.cmpb.2010.10.009>
7. Shams, R., Sadeghi, P., Kennedy, R. A., Hartley, R. I. (2010). A Survey of Medical Image Registration on Multicore and the GPU. *IEEE Signal Processing Magazine*, 50-60, Retrieved from
<https://ieeexplore.ieee.org/abstract/document/5438962/authors#authors>
8. Crum, W. R., Hartkens, T., & Hill, D. L. G. (2004). Non-rigid image registration: theory and practice. *The British Journal of Radiology*, 77, S140-153.
<https://dx.doi.org/10.1259/bjr/25329214>
9. Ruiz, A., Ujaldon, M., Cooper, L., & Huang, K. (2009). Non-rigid Registration for Large Sets of Microscopic Images on Graphics Processors. *Journal of Signal Processing Systems*, 55(1-3), 229-250. <https://doi.org/10.1007/s11265-008-0208-4>
10. *Understanding Cross-Correlation, Auto-Correlation, Normalization and Time Shift*. (2016). Retrieved from
<https://anomaly.io/understand-auto-cross-correlation-normalized-shift/>
11. Hii, A. J. H., Hann, C. E., Chase, J. G., & Van Houten, E. E. W. (n.d.). *Fast Normalized Cross Correlation for Motion Tracking using Basis Functions*. Department of Mechanical Engineering University of Canterbury, Christchurch, New Zealand. Retrieved from
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.108.5048&rep=rep1&type=pdf>

12. Ashburner, J., & Friston, K. J. (n.d.) *Rigid Body Registration*. The Wellcome Dept. of Imaging Neuroscience, 12 Queen Square, London, UK. Retrieved from <https://www.fil.ion.ucl.ac.uk/spm/doc/books/hbf2/pdfs/Ch2.pdf>
13. *Introduction to Optimizers*. (2018). Retrieved from <https://blog.algorithmia.com/introduction-to-optimizers/>
14. Comparison of photo stitching software. (n.d.) In *Wikipedia*. Retrieved November 6th, 2018, from https://en.wikipedia.org/wiki/Comparison_of_photo_stitching_software#HDR_tonemapping_and_exposure_fusion
15. *Canon EOS 5DS R*. (n.d.). Canon EOS 5DS R product page. Retrieved from https://www.canon.co.uk/for_home/product_finder/cameras/digital_slr/eos_5ds_r/
16. *Other Built-in Data Types*. (n.d.). Retrieved from <https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/otherDataTypes.html>
17. Sellner, J. (2017). *Buffer vs. image performance for applying filters to an image pyramid in OpenCL*. Retrieved from https://milania.de/blog/Buffer_vs_image_performance_for_applying_filters_to_an_image_pyramid_in_OpenCL
18. Rotation Matrix. (n.d.) In *Wikipedia*. Retrieved February 5th, 2019 from https://en.wikipedia.org/wiki/Rotation_matrix
19. *EXTENSION*. (n.d.). Retrieved from <https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/EXTENSION.html>
20. Comparison of NVIDIA Tesla/Quadro and NVIDIA GeForce GPUs, (n.d.). Retrieved from <https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/>
21. DarkZeros (2017, August, 2). Re: OpenCL When to use global, private, local, constant address spaces [Online forum comment]. Retrieved from <https://stackoverflow.com/questions/45426212/opengl-when-to-use-global-private-local-constant-address-spaces>
22. Fastvideo. (2019) 3D LUT on NVIDIA GPU [Computer software]. Retrieved from <https://www.fastcompression.com/solutions/gpu-3dlut.htm>

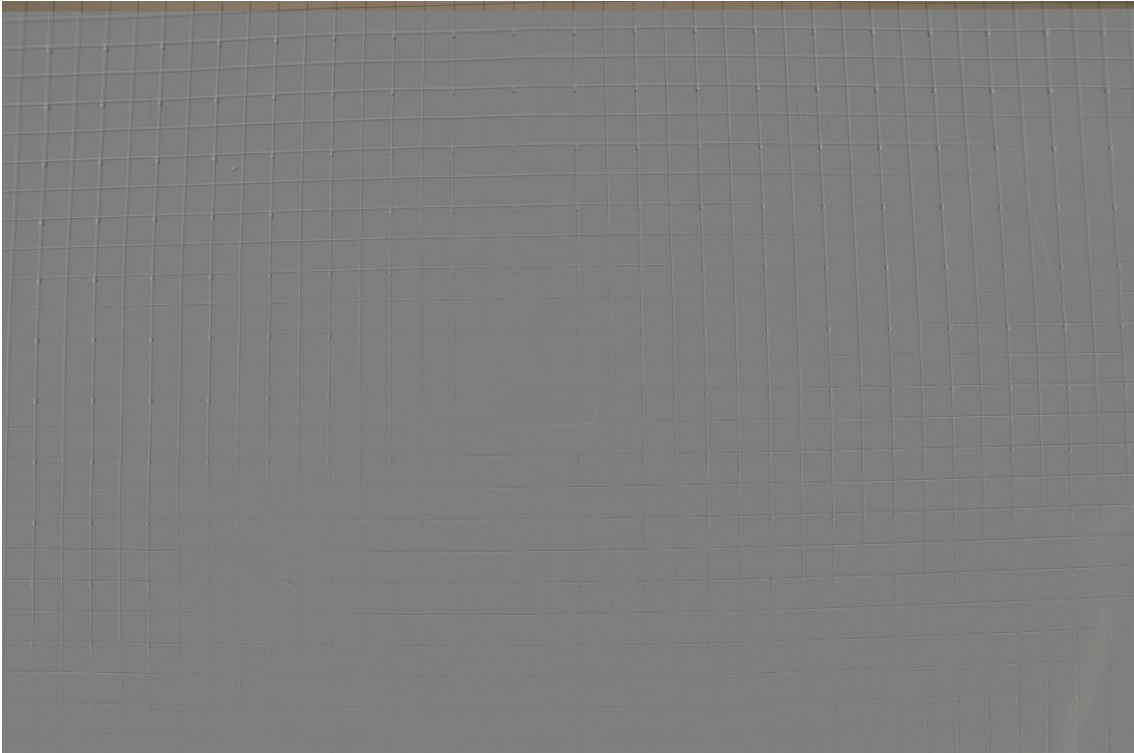
9. List of figures

1. Figure 1. Under and Over Exposed Image.
2. Figure 2. Basic Image Registration Algorithm
3. Figure 3. Camera rotation 1
4. Figure 4. Camera rotation 2
5. Figure 5. Camera rotation 3
6. Figure 6. Full Program Flow
7. Figure 7. The image merge function (`render_image()`)
8. Figure 8. The Optimiser viewed
9. Figure 9. Performance of OpenCL kernel on Radeon R9 290, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays
10. Figure 10. Performance of OpenCL kernel on Intel HD Graphics 515, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays
11. Figure 11. Performance of OpenCL kernel on AMD Ryzen 1700x, measured in Megapixels/second vs number of simultaneous GPU threads, using C-style arrays
12. Figure 12. Performance of OpenCL kernel on Radeon R9 290, measured in Megapixels/second vs number of simultaneous GPU threads, using `image2d`
13. Figure 13. Example Software Output (Image set 1)
14. Figure 14. CPU Profiling of new software (Image set 1)

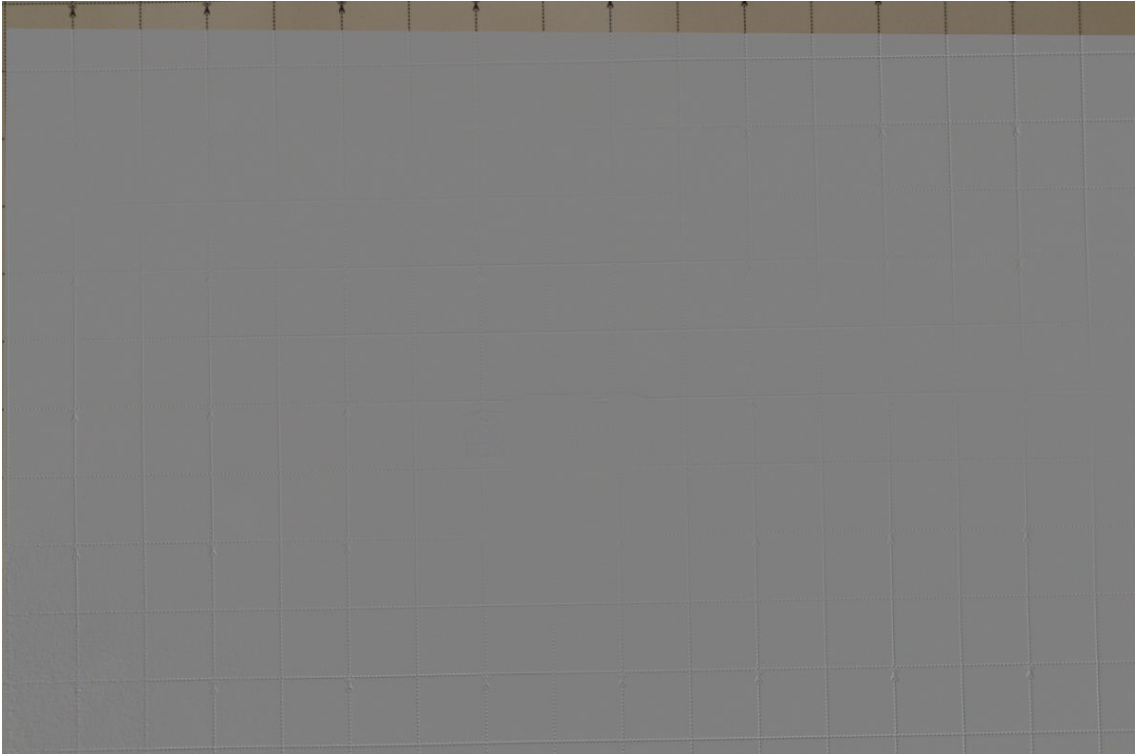
10. List of tables

1. Table 1. Risks table.
2. Table 2. Similarity Measure performance table
3. Table 3. Image transform table
4. Table 4. Software Performance Results Table
5. Table 5. Software Performance Requirements Target Table
6. Table 6. OpenCL image sample rate table
7. Table 7. Software Performance Requirements Results Table
8. Table 8. Software Functional/Non-Functional Requirements Results Table

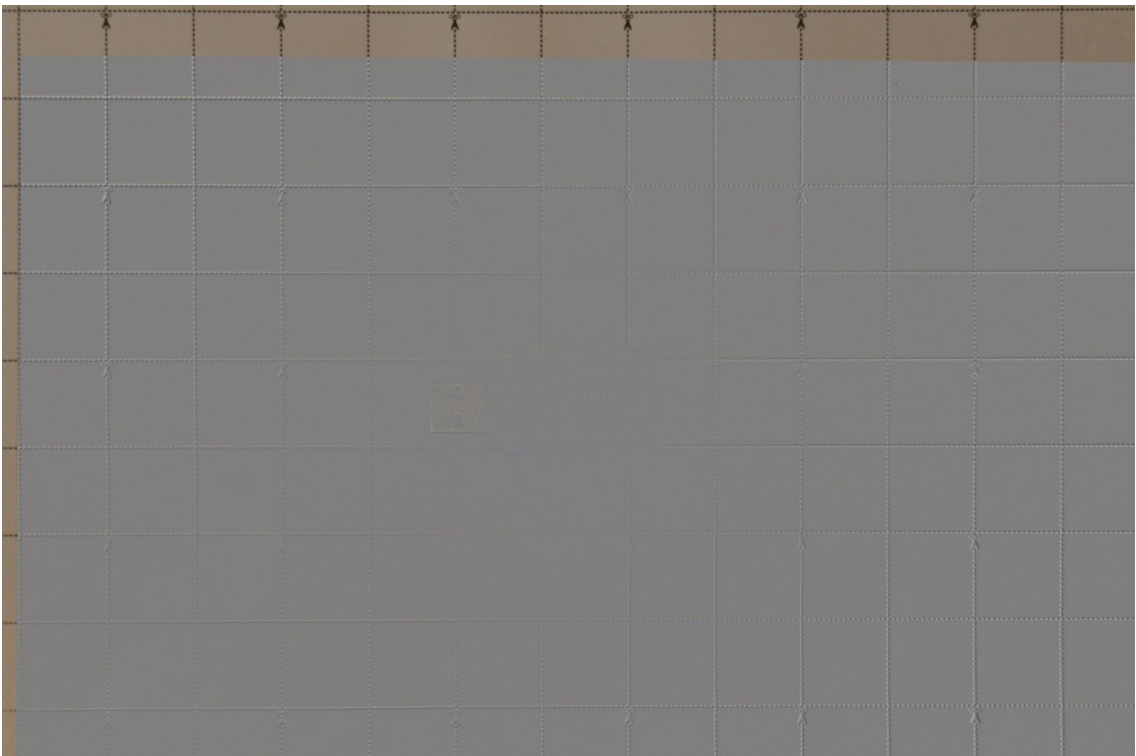
11. Appendices



Appendix 1. Transform Image Example 1



Appendix 2. Transform Image Example 2



Appendix 3. Transform Image Example 3

Date	Major Deadlines	Minor Deadlines						
22-Oct-2018	PID Submission	Risks Assessment						
29-Oct-2018								
5-Nov-2018			Initial Research and Gather Initial Requirements	Initial Requirements Specification				
12-Nov-2018								
19-Nov-2018								
26-Nov-2018								
3-Dec-2018					Learn and Experiment with OpenCL	Literature Review		
10-Dec-2018								
17-Dec-2018								
24-Dec-2018	Holiday							
31-Dec-2018								
7-Jan-2019					Learn and Experiment with OpenCL	Generate Testing Method		
14-Jan-2019								
21-Jan-2019								
28-Jan-2019							Algorithm Design	
4-Feb-2019								
11-Feb-2019	Satisfactory Progress Report							
18-Feb-2019								
25-Feb-2019								
4-Mar-2019								
11-Mar-2019								
18-Mar-2019								Requirements Validation
25-Mar-2019								
1-Apr-2019								Evaluation
8-Apr-2019	Contingency Time							
15-Apr-2019								
22-Apr-2019								
29-Apr-2019								
6-May-2019	Final Submission							

Appendix 4. Project Plan



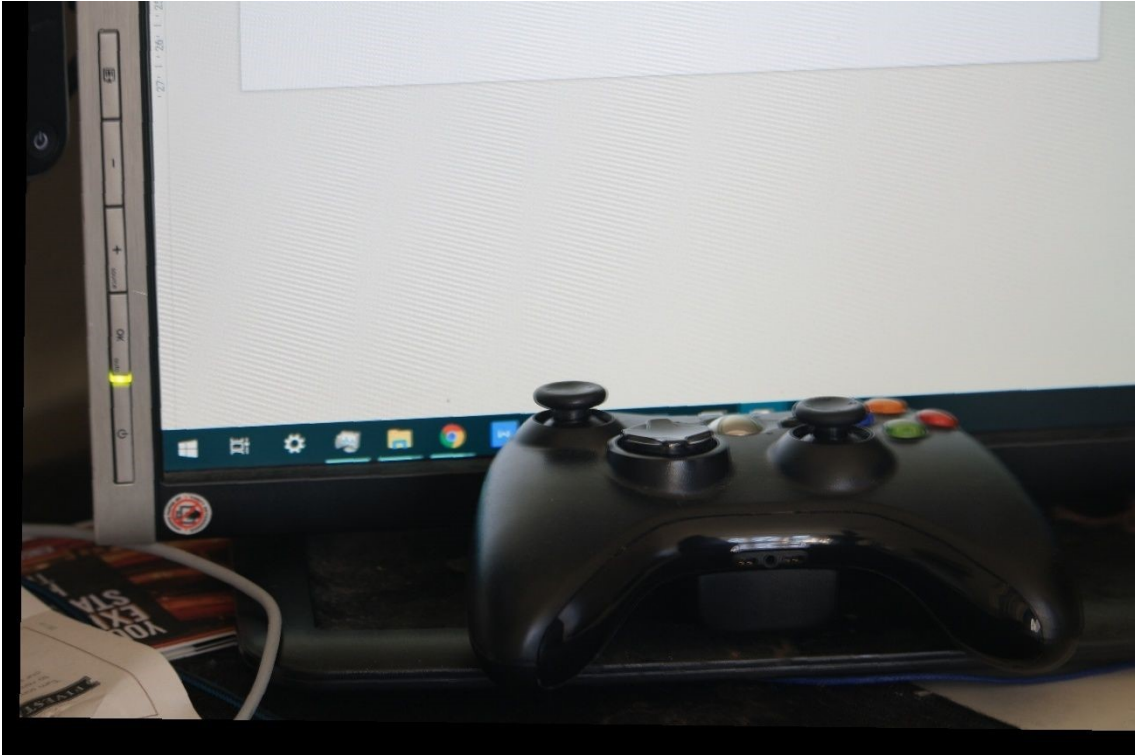
Appendix 5. Image Set 1 New Software Output



Appendix 6. Image Set 2 New Software Output



Appendix 7. Image Set 3 New Software Output



Appendix 8. Image Set 4 New Software Output



Appendix 9. Image Set 5 New Software Output



Appendix 10. Image Set 6 New Software Output



Appendix 11. Image Set 7 New Software Output

Date	Major Deadlines	Minor Deadlines									
22-Oct-2018	PID Submission	Risks Assessment									
29-Oct-2018											
5-Nov-2018											
12-Nov-2018			Initial Research and Gather Initial Requirements	Initial Requirements Specification							
19-Nov-2018											
26-Nov-2018											
3-Dec-2018					Learn and Experiment with OpenCL	Literature Review					
10-Dec-2018											
17-Dec-2018											
24-Dec-2018	Holiday										
31-Dec-2018	Holiday										
7-Jan-2019											
14-Jan-2019					Learn and Experiment with OpenCL		Generate Testing Method				
21-Jan-2019								Algorithm Design			
28-Jan-2019											
4-Feb-2019											
11-Feb-2019	Satisfactory Progress Report					Literature Review					
18-Feb-2019											
25-Feb-2019											
4-Mar-2019											
11-Mar-2019											
18-Mar-2019											
25-Mar-2019											
1-Apr-2019											
8-Apr-2019											
15-Apr-2019	Planned Contingency Time										
22-Apr-2019											
29-Apr-2019										Requirements Validation	
6-May-2019	Final Submission										Evaluation

Appendix 12. Project Timeline