Imperial College London

Department of Computing

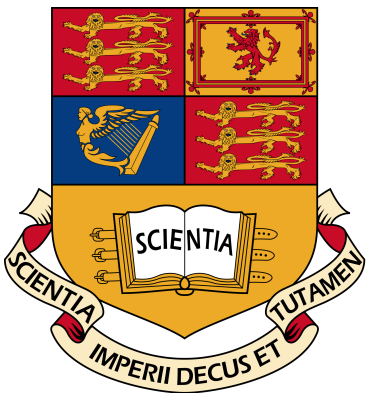
BEng Individual Project

Real-time 3D Ultrasound Stitching

Author:
Ivan Diaz Rios

Supervisor:
Dr. Bernhard Kainz *Second Marker:*
Dr. Abhijeet Ghosh

June 13, 2016



Abstract

The increasing adoption of 3D ultrasound by the medical community has drawn a focus on the technical improvements that can be made to increase its quality as a diagnostic imaging technique.

This project proposes an implementation for real-time ultrasound image fusion and registration in order to rectify small alignment errors in ultrasound acquisition systems. The required levels of parallelization for real-time performance are achieved through the use of the CUDA parallel computing platform.

The work accomplished in this project forms part of a larger research initiative with King's College London which hopes to increase the usefulness of medical ultrasound.

Acknowledgements

I would like to thank my supervisor Dr. Bernhard Kainz for his guidance throughout this project. I would also like to thank Alberto Gomez for his patience with all my questions. Lastly I would like to thank Danielle and my mother Marta for their support during my time at Imperial.

Contents

Chapter 1

Introduction

1.1 Motivation

Ultrasound offers a safe way for doctors to view successive "snapshots" of a patient's internals. It is a non-invasive procedure for diagnosing patients that has increased in popularity over the years and now, with the greater wide-spread availability of 2D array transducers, 3D ultrasound is becoming more common-place too. The concept is the same but snapshots are now replaced with volumes allowing for a better spatial understanding of structures being viewed.

By stitching these volumes in real-time, as they are acquired, we can expand the field-of-view and minimize artefacts, allowing sonographers to perform more conclusive examinations. Such a stitching implementation can also be used offline to improve the quality of training volumes meaning sonographers can improve their skills and further improve the care they provide to thousands of woman each year. As the quality of 3D ultrasound progresses other medical professionals will also be able provide valuable input to patient diagnoses as the additional skills required to understand ultrasound images will be reduced and features will be more easily identified.

1.2 Problem Statement

The stitching of 3D ultrasound images presents a greater challenge than with other medical modalities. Structures in ultrasound images can vary in appearance when viewed from different angles. Due to this gradient based nature of ultrasound, information needed to assist the alignment of two images may often be distorted. Small degrees of displacement can also occur due to organ, tissue and foetal movement, which further complicates the search for the optimal alignment between ultrasound images.

Modern acquisition systems contain trackers which can be used to align images captured within the same session. Unfortunately, these trackers are susceptible to magnetic interference causing the alignments to be inaccurate. The images therefore need to be

registered to fix the errors in the tracker data before they are fused to produce a final image, which would otherwise appear distorted.

An implementation which provides real-time performance is a key challenge which limits what type of registration method is usable. An approach must not sacrifice alignment quality for speed as an implementation that produces incorrect results is far less preferable to one that lacks fast enough execution speed for real-time image processing. The latter can still be used for offline volume processing while the former cannot.

1.3 Objectives and Contributions

This project is part of a collaborative research initiative with King's College London. It will contribute the following:

- A real-time ultrasound image fusion implementation. This will be able to form part of acquisition systems used by researchers at St Thomas' hospital and will also enable the offline processing of ultrasound data to be sped up.
- A proof of concept registration method for aligning ultrasound volumes in real-time.
- Two groundwork GPU implementations for ultrasound image processing which can be extended by future research.

Chapter 2

Background

2.1 Ultrasound Imaging

2.1.1 Basics

Ultrasound imaging is a non-invasive diagnostic imaging technique which utilizes ultrasound: sound waves with a frequency greater than 20,000 Hertz which the human ear cannot hear. The acoustical impedance of these waves can be used to visualize internal body structures by emphasizing their interfaces. The frequencies of sound used in medical ultrasound range between 2 MHz to 20 MHz.

Ultrasound transducers use a material called piezo electric crystal which expands when a voltage is applied to it and contracts back to its original size when the voltage is removed. By quickly applying and removing a voltage, the crystal's rapid expansion and contraction creates ultrasound waves. Additionally, when the crystal is hit by an ultrasound wave it is compressed, which also generates a voltage corresponding to the intensity of the wave. These crystals can therefore both transmit ultrasound waves and listen for them. Medical imaging transducers use these crystals to generate ultrasound waves in pulses. Enough time between pulses allows for the produced sound waves to reach their target and be reflected back as echo which is then detected by the transducer.

Waves are reflected according to a property called acoustical impedance. This can be thought of as the difficulty for a wave to travel through a medium. Echoes are produced at the interfaces of structures with different acoustical impedance. The intensity of the echo depends on the difference between the two tissue's acoustical impedance. This means that if a wave meets the interface between structures with similar acoustical impedance there will be little echo reflected back. On the other hand, if it meets an interface between structures with a vastly different acoustical impedance such as bone and air, a large portion of the wave will be reflected back. By measuring the time interval between when a transducer emits an ultrasound wave, and when the transducer receives the echo, the distance to the interface can be calculated and displayed as a bright point in an image. This is what B-mode (brightness mode) ultrasound does and it is the most widely

employed mode for ultrasound imaging.

Body tissue	Acoustic impedance (10^6 Rayls)
Air	0.0004
Lung	0.18
Fat	1.34
Liver	1.65
Blood	1.65
Kidney	1.63
Muscle	1.71
Bone	7.8

Figure 2.1: Acoustical Impedance values for types of body tissue. Source: [1]

If we group multiple transducers into an array we can generate a 2D image. This process is depicted in Fig 2.2. The same concept can then be extended by using a two dimensional array of transducers to produce a 3D volume.

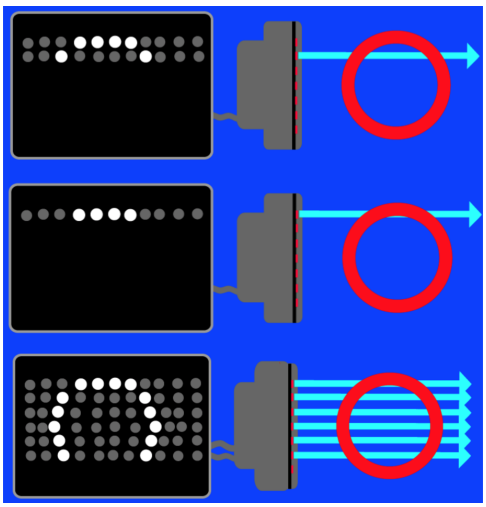


Figure 2.2: Demonstration of how 2D ultrasound images are produced. Source [2]

2.1.2 Ultrasound Artefacts

Ultrasound images often contain artefacts, the most common of which are shadowing, speckle and duplication artefacts. Shadowing occurs when ultrasound waves encounter mediums with very high or very low acoustical impedance such as bone or air, respectively. When this happens, most of the wave is reflected back resulting in a shadow being produced as further structures cannot be reached. This is also the reason why a water based gel is placed between the probe and the patients skin before conducting the medical procedure. It avoids any air surrounding the skin from reflecting the sound waves away and preventing passage into the patient's body.

Duplication artefacts are the consequence of refraction of the sound wave. When an ultrasound wave hits an interface of tissues with different acoustical impedance the waves

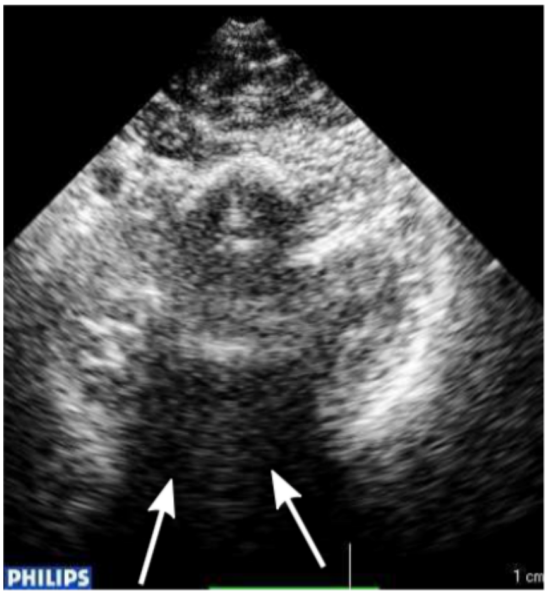


Figure 2.3: Shadowing artefact caused by bone. Source: [7]

which are not reflected back and make it through will change direction. This is due to the waves changing wavelength so as to accommodate for the decrease in speed (this follows as frequency must remain constant). In certain instances this direction change can cause ultrasound waves to detect structures not in the original path of the wave, leading to duplication artefacts as can be seen in Fig 2.4.

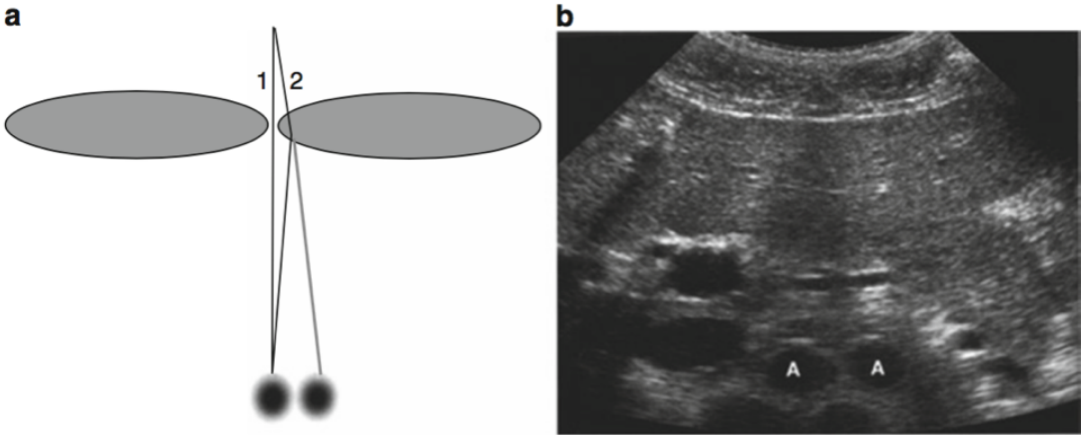


Figure 2.4: Example of a scenario which leads to a duplication artefact in an ultrasound image. Source: [7]

Speckle artefacts are caused by ultrasound waves being scattered when encountering small structures within tissue. The scattering produces a speckled pattern as is visible in Fig 2.5. Using an array of transducers usually results in speckle noise reduction since the use of multiple beams can average out the speckling. Gaussian filters are also often used to reduce the speckle noise.

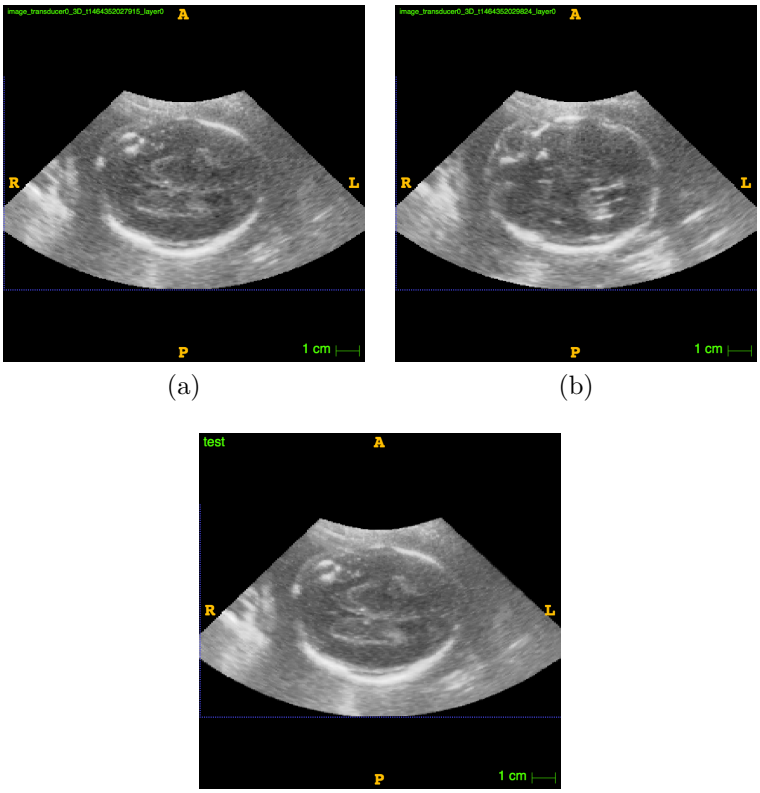
Figure 2.5: Example of speckle noise in an ultrasound image. Source: [7]

2.1.3 Ultrasound Stitching

Ultrasound stitching is the process of combining the information from individual ultrasound images into one image. There are two stages involved in ultrasound stitching. The first is registering the images, that is figuring out their correct alignment, and the second is the fusion of the registered images to a reconstructed final image.

Registration can only happen if the initial volumes have a minimum amount of overlap. The main goal of this process can have different focuses. One focus can be extended field-of-view imaging where the resulting volume is larger giving a more expansive view of the structures contained in the volumes. Minimal overlap is desired in these cases. On the other hand compounding is the process of creating a better quality volume. This requires using volumes from different angles which contain a high degree of overlap to gain more information of the same structure contained in the image. An example of compounding is shown in Fig 2.6.

As ultrasound emphasizes the the interfaces of structures the resulting image is dependent on the angle the structures are viewed. This gradient based dependency means there is no guarantee the same structures will look the same when viewed from different angles. This can make registration a lot more challenging and is why some registration methods can produce worse results when applied to ultrasound instead of data from a different modality. Image fusion generally uses the predicted alignment of a registration method to combine information from images into a single image. The fusion method can aid in emphasizing the relevant structural details while reducing the noise in the final reconstructed volume.



(a) (b)

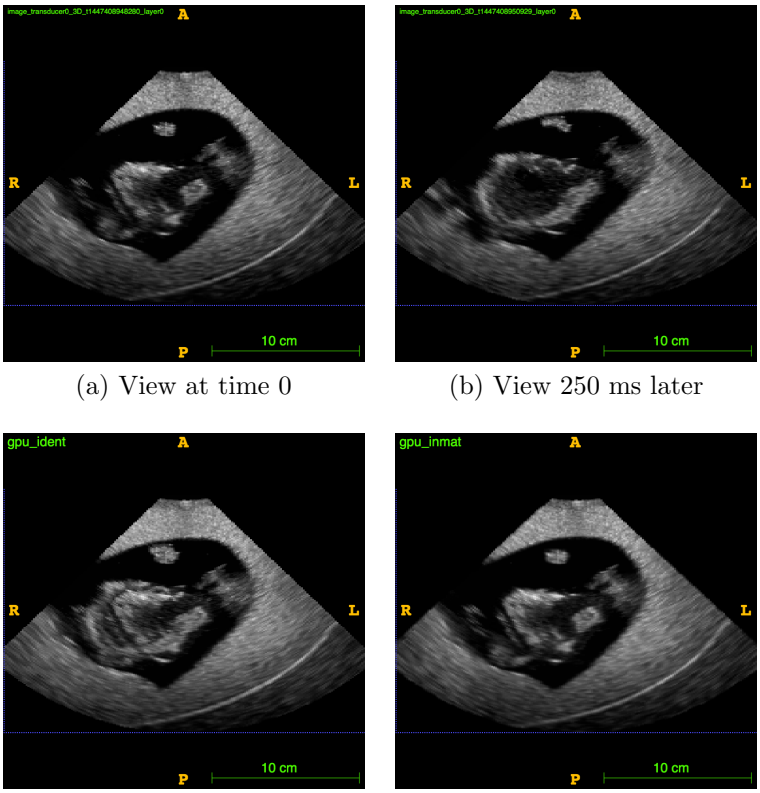
(c)

Figure 2.6: Using different angles (a), (b), of the same structure to create a more informative volume (c).

2.2 Image Registration

Registration is the process of finding the optimal alignment between images. This alignment is represented by a transformation from one image to the other. If the volumes 2.7a and 2.7b in Fig 2.7 were simply fused on top of each other (using the identity transformation) the result would look like the Fig 2.7c, while if the optimal alignment was found before hand it would correctly look like the one in Fig 2.7d.

There are different approaches to registering images. Some approaches are more favorable than others depending on the situation and type of images. Following are different classifications for registration methods.



(a) View at time 0

(b) View 250 ms later

(c) Fusion with the identity transform

(d) Fusion with the optimal alignment

Figure 2.7

2.2.1 Rigid vs Non-Rigid

Rigid registration methods are used when the differences between volumes can be modeled as rigid transformations. A rigid transformation is any transformation in vector space where the distances between every pair of points is preserved, for example translation, rotation and reflection. Non-rigid registration methods in contrast should be used when elastic (non-rigid) deformations can occur between volumes. Due to the greater degrees of freedom of non-rigid transformations non-rigid registration algorithms tend to be computationally more expensive than rigid ones [11] [12]. The choice of rigid vs non-rigid registration when working with ultrasound volumes is dependent on the acquisition rate of the transducer. The lower the frame rate the more time a fetus and organs have to move, allowing more non-rigid deformation to appear between volumes.

2.2.2 Tracker Based vs Image Based

Tracker based registration methods use the position and orientation of the transducer to supply the transformations between the volumes. They are simple as little computation is needed. Unfortunately some trackers are susceptible to magnetic interference which can distort the data used to create the transformations between volumes. Fusing volumes 2.7a and 2.7b using the information from the tracker gives 2.9.

Figure 2.8: Non rigid deformation between ultrasound images due to foetal movement. Source: [10]

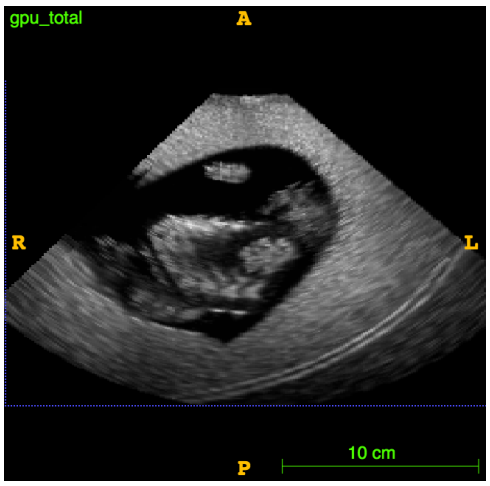


Figure 2.9: Fusion with tracker transform information.

The alternative to tracker based methods are image based methods. These attempt to track displacements in the volume and can be further categorized into intensity based and feature based methods.

Intensity based, or voxel based registration, can be thought as an iterative registration technique which tries to find the transformation between volumes by minimizing or maximizing a cost function (section 2.2.4). Feature based methods attempt to track the displacement of feature sets between volumes. They mainly use feature descriptors to achieve this and past implementations have shown that they can only handle small displacements between volumes [14].

2.2.3 Pairwise vs Simultaneous

One of the dominant factors when deciding which registration algorithm to use with a set of volumes is the amount of overlap that exists amongst them. The 3 main scenarios are illustrated in Fig 2.10. The first, left-most, scenario occurs when only neighbor volumes overlap and the second, middle, scenario is when there is a small overlap between non-

neighbors but the neighbor overlap remains dominant. The last right-most scenario involves all pairs of volumes having a high degree of overlap.

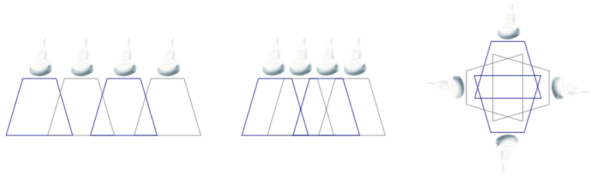


Figure 2.10: Illustration of 3 different registration scenarios. Source: [3]

In pairwise registration methods, all overlapping pairs of volumes undergo registration and the transformations produced are then used to deduce the correct global alignment for all the volumes. The main issue with this approach is that registration errors are accumulated which can lead to a bad overall global alignment as depicted in Fig 2.11. Simultaneous strategies instead use the available information from all the volumes at the same time when performing the registration. Wachinger summarizes their advantage in his thesis [3] by stating that they generally present a "better conditioned optimization problem on cost functions created by multivariate similarity measures".

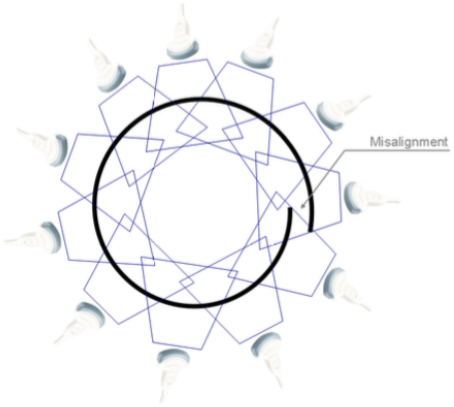


Figure 2.11: Illustration of 3 different registration scenarios. Source: [3]

Simultaneous strategies come with the downside of a higher computational cost so they should be used in situations where pairwise registration does not achieve the level of accuracy desired. Fig 2.12 shows a comparison of the accuracy of pairwise strategies vs simultaneous in one study [6].

Trajectory	Registration Strategy	Error	
		Translation (mm)	Rotation (degrees)
1	RTG	0.36	0.39
	RTP	0.56	1.06
2	RTG	0.28	0.36
	RTP	1.73	2.81
3	RTG	0.27	0.31
	RTP	4.70	4.15
Avg. ± Std. Dev.	RTG	0.30 ± 0.05	0.35 ± 0.04
	RTP	2.33 ± 2.13	2.67 ± 1.55

Figure 2.12: Comparison of pairwise vs global registration technique for a registration strategy R.

2.2.4 Cost Functions

The cost function is used to evaluate how accurately images are aligned. Different cost functions assume different relationships, and generally multivariate measures are preferred as bivariate measures are extremely overlap dependent [3]. The multivariate similarity metrics we will briefly discuss are analyzed in detail by Wachinger in his theses [3].

Two cost functions commonly used are the mean squared error (2.1) and normalized cross-correlation (2.2). The first is an identity metric that is used to calculate the similarity between volumes and normalized cross correlation is used to measure affine relationships between volumes [3]. Also commonly applied is mutual information, which focuses on statistical relationships between volumes. It is based on information theory and measures how much uncertainty about one set is reduced by the knowledge of the second set (equation 2.3 where H is entropy and A and B are two sets). Mutual information performs well with images or volumes from different modalities [15].

$$\text{Mean Squared Error} = \frac{1}{n}\sum(Y - X)^2 \tag{2.1}$$

$$\text{Normalized Cross Correlation} = \frac{1}{n}\frac{\sum(X * Y)}{\sqrt{\sum X^2}\sqrt{\sum Y^2}} \tag{2.2}$$

$$\text{Mutual Information} = H(A, B) - H(A) - H(B) \tag{2.3}$$

2.3 Image Fusion

The next step after registering volumes is fusing them together. Different criteria for fusing the intensities of volumes can be applied. Very general methods produce a target voxel intensity by calculating the mean, max or median of mapped voxels of the registered volumes.

There also exists more advanced fusion methods. Morphological methods focus on identifying spatially relevant data from the medical images using operators like K-L transforms or morphology pyramids. Unfortunately these methods are inaccurate with images containing noise and high amounts of variability, which is often the case with ultrasound. When information about a patient is known, medical experts can use knowledge based methods, but these require external input so are often hard to automate [8]. We also have neural network based image fusion methods which use inputted training data to determine the networks weights. The advantage of these methods is that they do not require computationally expensive mathematical solutions to infer information from data once trained [8].

2.4 Related Work

There are a number of research papers and existing implementations for 3D ultrasound registration and fusion using graphical processing units. I will give a brief outline of them here.

2.4.1 Research Papers

- *Real-time image-based rigid registration of three-dimensional ultrasound* [6]

  This paper presents a featured-based 3D ultrasound rigid registration algorithm related to SIFT. SIFT-based methods use SIFT keypoints of objects. They are stored in a database and used to recognize objects in new volumes or images by individually comparing each feature and using the Euclidean distance of their feature vectors to find matches [18]. The methods devised in this paper are used to perform real-time 3D volume stitching in ultrasound acquisition systems. Using GPU acceleration frameworks they were able to register two volumes in under 100ms. It concluded visually accurate results for acquisition rates of at least 25 Hz suggesting any non-rigid deformation is negligible between frames with around 40ms in delay. The registrations tests were run on NVIDIA GTX 260 (192 cores) and NVIDIA GeForce 8600M GT (32 cores) and a linear relationship between registration time and number of GPU cores was observed. The paper does not address the exact dimensions of all the data used but suggests each volume consisted of less than 2.5 millions voxels.

- *Multi-modal Registration Based Ultrasound Mosaicing* [13]

  In this paper an intensity based registration algorithm is presented to compute global affine transformations using a Downhill Simplex optimizer. It's runtime is evaluated using different similarity metrics on both the CPU and GPU and results are evaluated by visually inspecting alignments of major anatomical regions. The CPU was an Intel Core Duo processor (2.6Ghz) with 4GB of RAM and the GPU a NVIDIA Geforce GTX 280 with 1GB of VRAM. Fig 2.13 is a chart depicting the runtime results of the implementation.

- *Real-Time 4D Ultrasound Mosaicing and Visualization* [4]

	GPU [sec]	CPU [sec]
4 x 256^3	11.19231	> 36.000
4 x 128^3	9.71301	4006.34
4 x 64^3	7.10882	397.21

Figure 2.13: Results from study [13]

In this paper only fusion is performed. The implementation is written in C/C++ and CUDA. The transformation matrix is produced by the tracker system. Both volumes are sent to GPU device memory and wrapped in CUDA textures. 3D linear interpolation is then performed to map one volume onto the other and the maximum intensity is selected for overlapping locations. The data used to evaluate performance had dimensions of 320 x 144 x 210 and the average fusion time was of 25 ms between volumes when running on NVIDIA GTX260 graphics cards.

- *Real-Time GPU-Based 3D Ultrasound Reconstruction and Visualization* [5]

  This thesis describes pixel based and voxel based registration methods which are implemented with OpenCL. Results were measured on 3 different GPUs with 4 x 2 GB DDR3 memory: Nvidia Tesla C2050, AMD ATI Radeon HD5870 and Nvidia Quadro FX5800. Stitching results averaged 0.9 seconds and 0.6 seconds for pixel based methods and voxel based methods respectively on data with dimensions of 512 x 256 x 512. CPU results were also reported (Intel Core 2 Quad Q9550) with pixel based methods running up to 14 times slower than on the GPU, compared to voxel based methods running as high as 51 times slower.

2.4.2 NiftiReg

NiftiReg is an open-source medical image registration software developed at UCL primarly by members of the Translational Imaging Group and the Centre for Medical Image Computing. It provides rigid/affine and non-rigid registration implementations. The rigid/affine implementation uses a block matching algorithm and the non-rigid implementation uses a Fast Free-Form Deformation algorithm. The block matching algorithm is based on the past studies [20][21]. The repo's description of the algorithm is as follows:

"Firstly, the block matching provides a set of corresponding points between a target and a source image. Secondly, using this set of corresponding points, the best rigid or affine transformation is evaluated. This two-step loop is repeated until convergence to the best transformation. In our implementation, we used the normalised cross-correlation between the target and source blocks to extract the best correspondence. The block width is constant and has been set to 4 voxels. A coarse-to-fine approach is used, where the registration is first performed on down-sampled images (using a Gaussian filter to resample images) and finally performed on full resolution images."

The Fast Free-Form Deformation algorithm is based [22], and again it is described in the repo:

"The deformation of the source image is performed using cubic B-splines to generate the deformation field. Concretely, a lattice of equally spaced control points is defined over the target image and moving each point allows to locally modify the mapping to the source image. In order to assess the quality of the warping between both input images, an objective function composed from the Normalised Mutual Information (NMI) and the Bending-Energy (BE) is used. The objective function value is optimised using the analytical derivative of both, the NMI and the BE within a conjugate gradient scheme."

Table 2.1 outlines the execution times I obtained for registering and fusing two volumes both with dimensions of 192x253x135 using v1.3 of NiftiReg on CPU (Intel(R) Xeon(R) CPU E5606 @ 2.13GHz with 8GB RAM) and on a GPU (GeForce GTX 780):

Registration Type	Location	Execution Time
Affine	CPU	1 min 37 sec
Rigid	CPU	1 min 13 sec
Non-rigid	CPU	3 min 27 sec
Rigid	GPU	0 min 8 sec
Affine	GPU	0 min 17 sec
Non-rigid	GPU	0 min 38 sec

Table 2.1

2.4.3 ITK

ITK (Insight Segmentation and Registration Toolkit) is an open-source image analysis software package. It's extensive use of templating and generic programming has lead to support for a variety of medical data formats and the ability to add unsupported ones quickly by just defining appropriate template types as needed. ITK's seamless integration with the Visualization Toolkit (VTK) results in not just a powerful image processing library but combined also gives powerful medical visualising software [17].

ITK provides different registration implementations depending on the desired transform: translation, rigid, affine and non-rigid. The non-rigid methods can be further differentiated into differential and variational methods. ITK allows the user to select the desired similarity metric, which is advantageous as different metrics fit different modalities. The available metrics are mean squares, normalized cross correlation, pattern intensity and mutual information. The following are a few examples of optimizers that are available to the user: Gradient Descent, Regular Step Gradient Descent, Conjugate Gradient and Levenberg Marquardt (Nonlinear Least Squares Minimization). Unfortunately very few of the registration implementations have GPU support.

2.5 CUDA

CUDA was released in 2007 by Nvidia in an attempt to facilitate parralel computing and as such soon became the face of GPGPU programming (using a GPU for general

purpose processing). It is a platform built on top of Nvidia GPUs which provides an API such that languages like C, C++ and Fortran can make use of the GPU. CUDA has helped reduce the learning curve for GPU programming and allows programmers to port tasks normally implemented on the CPU onto the GPU. It has evolved greatly since its release and many libraries which use CUDA are now readily available offering GPGPU programming at a higher level meaning effective parallel programs no longer come at the cost of development time.

2.5.1 CUDA vs OpenCL

OpenCL is an open-source GPGPU framework which was released in 2009. When parallel computing is discussed CUDA and OpenCL are at the forefront. Both use the same compilation pipeline and share many concepts but a big difference is that the OpenCL framework can be used on a wider selection of GPU's. In order to use CUDA you must have a CUDA capable GPU supplied by Nvidia. The downside to this with respect to OpenCL is a slower evolvement of the framework to account for cross-platform support. Nvidia on the other hand has full control of the progression of CUDA and the graphics cards required to run it, which allow them to focus more efficiently on performance acceleration. Nvidia's GPUs support OpenCL giving users with CUDA capable GPUs an option to select from the two frameworks. When presented with both options it has generally been recommended to choose CUDA for performance reasons but this is slowly changing with the increasing adoption of OpenCL.

2.5.2 Programming in CUDA

The most dominant aspect of the CUDA programming model is the distinction it makes between *host* and *device*. The former refers to the CPU and the memory available to it while the latter refers to the GPU and its memory. A kernel is a function which is executed in parallel on the GPU. Kernel's only have access to device functions and device memory. The host is responsible for launching a kernel and moving data to the GPU that the kernel will need. A kernel launch takes an execution configuration, that is a set of parameters which are used to decide how the the GPU will distribute the workload.

When a kernel is invoked the threads created are split into blocks. These blocks are then allocated to a streaming multiprocessor (SM) which is made up of 8 CUDA cores. At any one time 32 threads are executed on these 8 CUDA cores (takes 4 clock cycles) which are referred to as a warp. For this reason the execution of warps are referred to as Single instruction, multiple thread (SIMT) since all threads in a warp execute the same instruction at the same time. A block of threads is split into groups of warps which take turn executing on the SM until all the threads of a block have completed execution, freeing up the SM for another block to be allocated to it. The number of threads per block and the number of blocks are the parameters that the programmer has to select when invoking a kernel. This will depend on the desired mapping between the threads and the data they are using and as a whole is a complicated discussion where a lot of research has been dedicated.

Every thread has its own set of registers and apart from that has access to different portions of the GPU memory as can be seen in 2.14. The differet areas of memory are:

- **Shared Memory**: on-chip, read/write memory shared by all threads in a block, very fast access.
- **Global Memory**: read/write and cached memory which is available to all threads. Still very slow relative to shared memory.
- **Constant Memory**: read-only memory region used to store constant data such as kernel arguments and constants.
- **Texture Memory**: read-only and cached on chip. Cache optimized for spatial locality so advantageous when all threads in a warp read from physically close addresses.

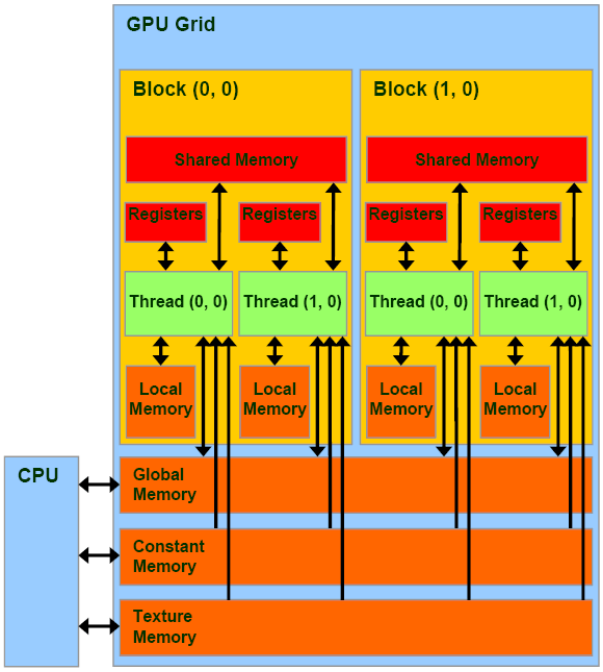


Figure 2.14: CUDA memory model. Source: [**?**]

Chapter 3

Implementation

3.1 General Approach

In the Background chapter I introduced the difference between tracker and image based registration strategies. The acquisition system being targeted in this project contains a tracker so we can use its data as the input to the registration method, limiting the parameter space.

Modern ultrasound acquisition system can acquire multiple frames per second. The acquisition system being targeted currently acquires 4 frames per second. Also discussed in the Background chapter is the speed benefit of rigid registration methods compared to non-rigid ones. The latter has less degrees so requires the search of a smaller parameter space. Since little movement from the organs and fetus can be expected within such small time intervals, a negligible amount of non-rigid deformation can be expected. Therefore the registration method employed will only account for rigid transformations in the images so as to reap the speed benefits of a reduced parameter space.

In order to evaluate the accuracy of an alignment a cost function is also necessary. It is a measure of how properly aligned two images are by using a transformation mapping one image onto another. Depending on the selected cost function the registration method's goal will be to find its minimum or its maximum, the point interpreted as the optimal alignment between the images. Since an implementation capable of performing in a real-time acquisition system is necessary an optimizer will also need to be implemented to speed up the search for the minimum of the cost function.

The image fusion implementation will be responsible for combining two volumes given a transformation matrix. It will be decoupled from the registration implementation to allow its use in the acquisition system alongside other registration implementations. The implementation will need to decide on how voxel intensities will be combined to form a voxel in the fused volume.

This chapter will first discuss how we can map voxels from one volume to another, followed by introducing a registration and fusion implementation on the CPU using C++.

Afterwards we will port these implementations to the GPU using CUDA.

Coordinate Systems

Voxels must be mapped between a target volume and a source volume to perform image registration and image fusion. This mapping can be identified in two directions, either going from the source volume to the target volume, known as forward mapping, or going from that target volume to the source volume, known as inverse mapping. The mapping can be performed by considering the coordinate systems the voxels exist in. The data in this project can be mapped to two coordinate systems: the tracker coordinate system and the world coordinate system.

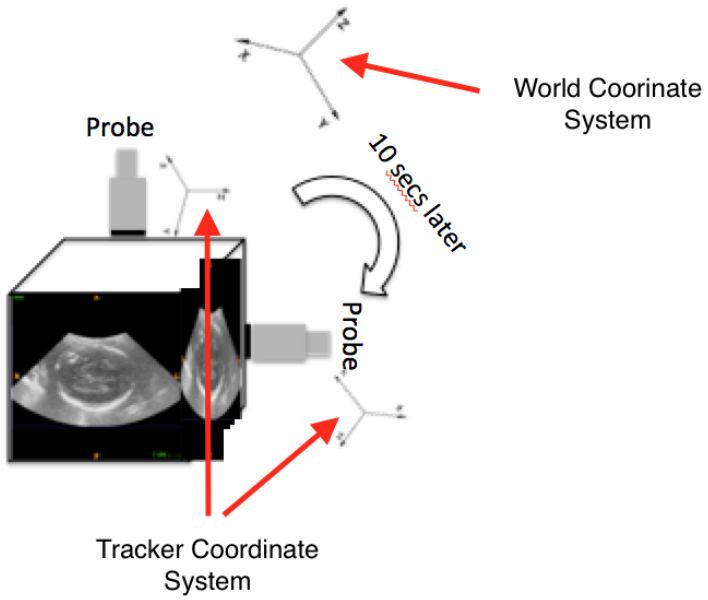


Figure 3.1: Distinction between tracker coordinate system and world coordinate system.

A voxel is mapped to the tracker's coordinate system by a transformation matrix M. M is calculated by using the spacing (size in mm) of the voxels and the dimensions of the data. Knowing the tracker's coordinate system is (0,0,0) at the tip of the frustum (Fig 3.2), we can calculate the **origin** of a volume, voxel (0,0,0) using equation (3.1). We can then use the origin in conjunction with the voxel spacing to find the transformation matrix, M (equation (3.2)), going from a voxel to the tracker's coordinate system.

$$\begin{aligned} xOrigin &= -(xDimensions - 1)/2 * xSpacing \\ yOrigin &= 0 \\ zOrigin &= -(zDimensions - 1)/2 * zSpacing \end{aligned} \tag{3.1}$$

$$M = \begin{pmatrix} xSpacing, & 0, & 0, & xOrigin \\ 0, & ySpacing, & 0, & yOrigin \\ 0, & 0, & zSpacing, & zOrigin \\ 0, & 0, & 0, & 1 \end{pmatrix} \tag{3.2}$$

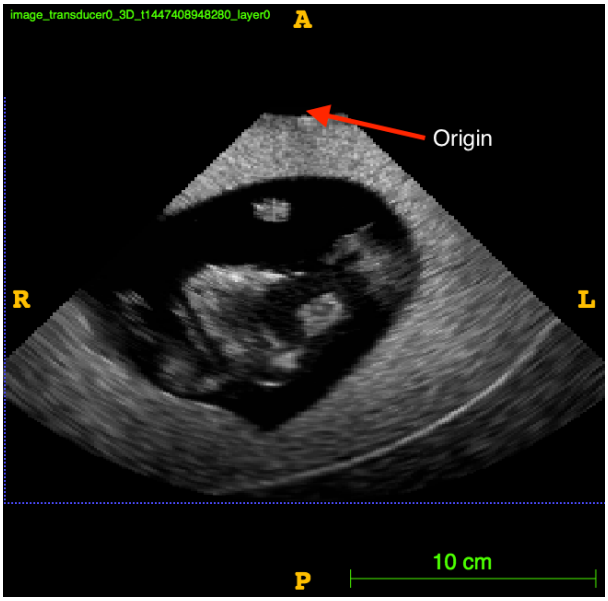


Figure 3.2: Position of origin in the ultrasound data.

If we represent a voxel as a homogeneous column vector then pre-multiplying it by M gives us the coordinates of its centre in the tracker's coordinate system. The next step is to go from the tracker's coordinate system to the world coordinate system; the global coordinate system where all volumes are acquired in. This is provided by the acquisition system. If we refer to this matrix as $M2$, then calculation of the world coordinates of a voxel can be done using equation (3.3). From the world coordinate system we can map any point back to the voxel indices of a volume given we have that volume's M and M2.

We need four matrices to go from a voxel in the target volume to a voxel in the source volume (equation (3.6)). It is matrices $M2_{source}$ and $M2_{target}$ which are supplied by the tracker and contain noise due to the tracker's sensitivity to surrounding magnetic forces. An example of the effect of noise in the tracker's transformation matrices is shown in Fig 3.3. For the remainder of this report, I will refer to $M2^{-1}_{source} * M2_{target}$ as the world transform matrix, and M^{-1}_{source} and M_{target} as the world to source matrix and the target to world matrix, respectively.

$$\begin{pmatrix} worldX \\ worldY \\ worldZ \\ 1 \end{pmatrix} = M2 * M * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.3}$$

$$\begin{pmatrix} sourceX \\ sourceY \\ sourceZ \\ 1 \end{pmatrix} = M^{-1}_{source} * M2^{-1}_{source} * M2_{target} * M_{target} * \begin{pmatrix} targetX \\ targetY \\ targetZ \\ 1 \end{pmatrix} \tag{3.4}$$

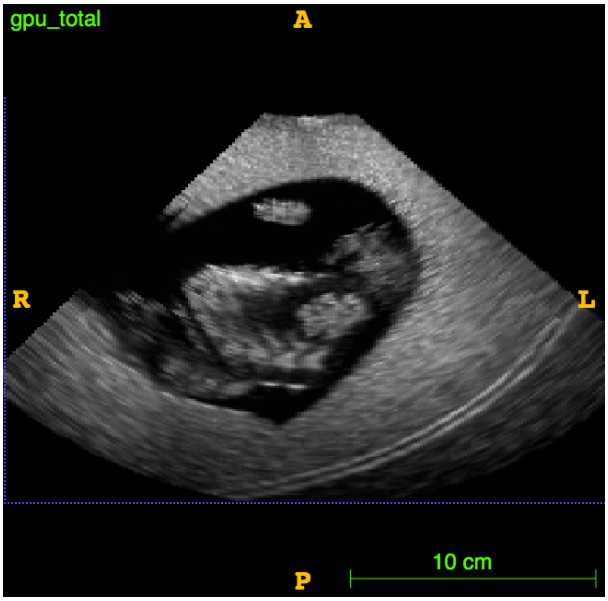


Figure 3.3: Misalignment due to magnetic interference of tracker's sorroundings

3.2 Registration Method

3.2.1 Calculating the Cost Function

The cost function will need to be evaluated repeatedly so using one with low computation complexity is desirable. I chose the mean squared error (equation (3.5)) as the cost function for this reason as it can be evaluated with only a single pass through all the image voxels. The input of the cost function will be a world transformation matrix and the output will be a measure of how well the images are aligned.

$$\text{MSE} = \frac{1}{n}\sum(I(targetX, targetY, targetZ) - I(sourceX, sourceY, sourceZ))^2 \quad (3.5)$$

Computing the squared error for every voxel of the target volume can be broken down into the following steps:

1. Find the corresponding voxel V_s in the source volume.
2. Compute the intensity of V_s by performing trilinear interpolation to account for non-grid indices.
3. Calculating the squared difference between the target and source voxel intenisties.

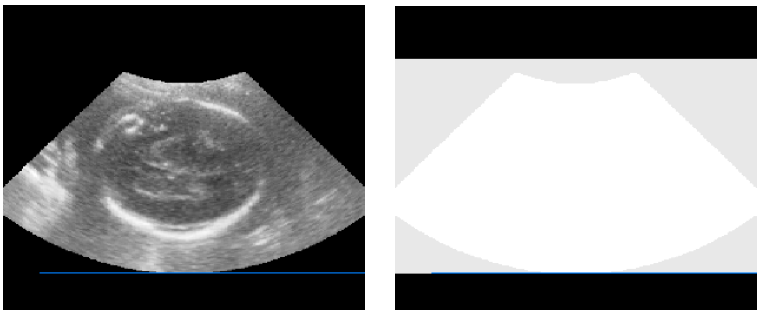
Implementing this on the CPU can be done with for loops to iterate through all the target volume's voxels. The transformation matrix will be the result of multiplying the world to source, world transform and target to world matrices together as explained in section 3.1. A CPU implementation therefore looks like:

```
float compute_mse() {
    mat4x4<float> transform = world_to_source *
                              world_transform *
                              target_to_world;

    float sum = 0;

    for (int i = 0; i < depth; i++)
    for (int j = 0; j < height; j++)
    for (int k = 0; k < width; k++) {

        index = k + width * (j + height * i); // Data is 1D, so flatten
    index
        out_vec = transform * vec4<float>(i,j,k,1.0f);

        float arg1 = target_data[index];
        float arg2 = interpolate(source_data, depth, height, width, out_vec
    );
        sum += (arg1 - arg2) * (arg1 - arg2);

    return sum / (depth * height * width);
}
```

The cost function must only evaluate the quality of alignment for overlapping voxels. If we look at an ultrasound image like the one in Fig 3.4a we can see that the voxels outside of the frustrum contain no data. It is therefore desirable to discard these voxels when computing the cost function. If we look at the same image with different contrast settings (Fig 3.4b) we can note that voxels in the frustrum are almost all non-zero while those outside are all zero valued. We can therefore check the values of the target and source voxels are both non-zero before calculating the squared difference. We will also have to keep track of the number of overlapping voxels. All of this can be done with minimal modifications:

```
float compute_mse() {

    mat4x4<float> transform = world_to_source *
                              world_transform *
                              target_to_world;

    float sum = 0;
    int n = 0;

    for (int i = 0; i < depth; i++)
    for (int j = 0; j < height; j++)
    for (int k = 0; k < width; k++) {

        index = k + width * (j + height * i); // Data is 1D, so flatten
    index
        out_vec = transform * vec4<float>(i,j,k,1.0f);

        if (arg1 && arg2) {
            sum += (arg1 - arg2) * (arg1 - arg2);
            n++;
        }
    }

```

```
23      return sum / n;
24  }
```



(a) (b)

Figure 3.4: Use of contrast to identify difference between voxels in frustrum and out.

3.2.2 Optimisation For Registration

Registering two images involves searching the parameter space which constitutes a rigid transformation: rotation and translation. Instead of finding the minimum of the cost function by searching the whole parameter space, we can apply an optimization called gradient descent.

Gradient descent iteratively moves the rotation and translation parameters against the gradient of the cost function, with respect to each parameter, until converge to a local minima (Fig 3.5). This optimization provides quick convergence to a minimum but comes with the downside of the ability to become trapped in local minima (Fig 3.6).

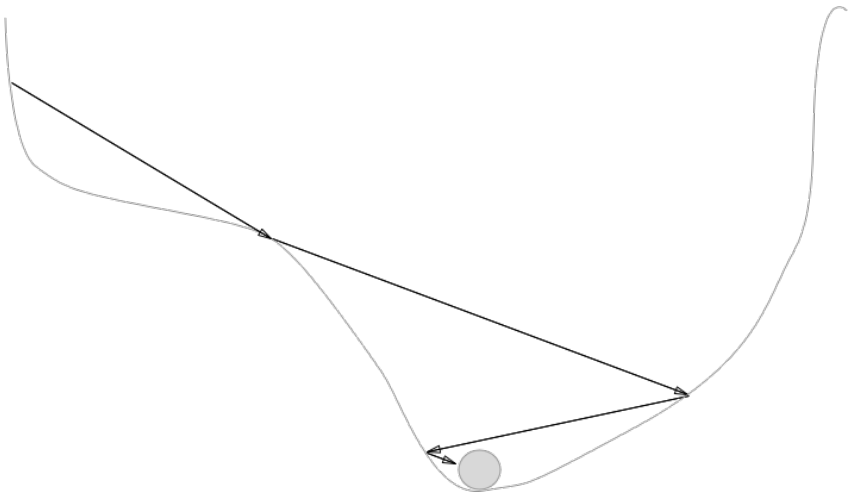


Figure 3.5: Iterative updates of gradient descent converging towards the minimum.

Since the tracker data for our images will place us near the global minimum of the parameter space, the probability of converging to a local minima lessens. Therefore we can use gradient descent as an optimizer to exploit its ability to greatly accelerate the registration method.

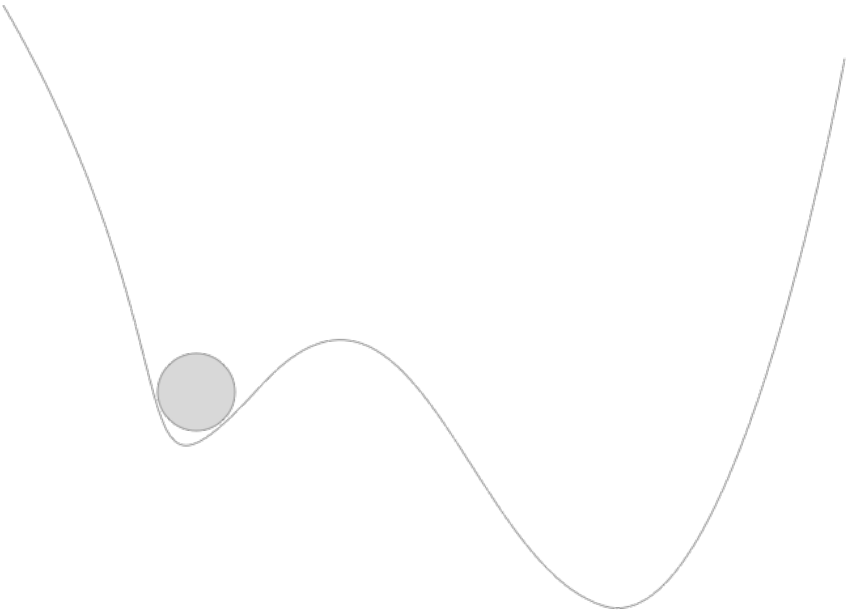


Figure 3.6: Convergence to a local minima instead of the global minimum.

The rotation and translation parameters will be used to update the world transformation matrix as demonstrated by equation (3.6) where W and W' are the old and new world transformation matrices, respectively.

$$W' = R * W + t \tag{3.6}$$

The initial W will be the world transform matrix presented in section 3.1. R and t are matrices created from the updated rotation and translation parameters, respectively. t can be generated using a simple transformation matrix from three translations dx, dy and dz as in equation (3.7) while generating R will involve the introduction of eular angles.

$$t = \begin{pmatrix} 1, & 0, & 0, & dx \\ 0, & 1, & 0, & dy \\ 0, & 0, & 1, & dz \\ 0, & 0, & 0, & 1 \end{pmatrix} \tag{3.7}$$

The Eular rotation theorem states that any 3D rotation can be represented by three rotation angles [25]. Eular angles can further be divided into two conventions: intrinsic and extrinsic rotations. With intrinsic rotations the coordinate system is also rotated, whereas in extrinic rotations it remains stationary. Bryant angles, the x-y-z convention of eular angles, perform intrinsic rotations along the x, y and z axes: first about the x axis by angle ϕ_1, then about the new y axis by angle ϕ_2 and finally about the new z axis by angle ϕ_3. This is depicted in Fig 3.7. These three angles can be used to generate the transformation matrix corresponding to these compounded rotations by multiplying the matrices from equation (3.8) as in equation (3.9). We now have three rotation parameters which we can use to generate R.

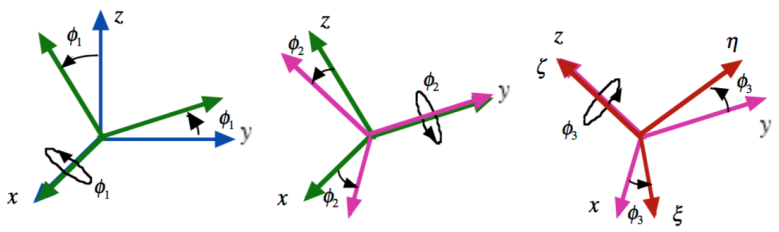


Figure 3.7: Rotations represented by bryant angles. Source: [26]

$$R_x = \begin{pmatrix} 1, & 0, & 0, & 0 \\ 0, & cos(\phi_1), & -sin(\phi_1), & 0 \\ 0, & sin(\phi_1), & cos(\phi_1), & 0 \\ 0, & 0, & 0, & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} cos(\phi_2), & 0, & sin(\phi_2), & 0 \\ 0, & 1, & 0, & 0 \\ -sin(\phi_2), & 0, & cos(\phi_2), & 0 \\ 0, & 0, & 0, & 1 \end{pmatrix} \tag{3.8}$$

$$R_z = \begin{pmatrix} cos(\phi_3), & -sin(\phi_3), & 0, & 0 \\ sin(\phi_3), & cos(\phi_3), & 0, & 0 \\ 0, & 0, & 1, & 0 \\ 0, & 0, & 0, & 1 \end{pmatrix}$$

$$R = R_x * R_y * R_z$$

$$R = \begin{pmatrix} c(\phi_2)c(\phi_3), & -c(\phi_2)s(\phi_3), & s(\phi_2), & 0 \\ c(\phi_1)s(\phi_3) + s(\phi_1)s(\phi_2)c(\phi_3), & c(\phi_1)c(\phi_3) - s(\phi_1)s(\phi_2)s(\phi_3), & -s(\phi_1)c(\phi 2), & 0 \\ s(\phi_1)s(\phi_3) - c(\phi_1)s(\phi_2)c(\phi_3), & s(\phi_1)c(\phi_3) + c(\phi_1)s(\phi_2)s(\phi_3), & c(\phi_1)c(\phi_2), & 0 \\ 0, & 0, & 0, & 1 \end{pmatrix} \tag{3.9}$$

where $c \equiv cos, s \equiv sin$

Now that we've defined our 6 parameters, ϕ_1, ϕ_2, ϕ_3, dx, dy, and dz, and we know how to use them to update the world transform matrix we can move onto the gradient descent implementation.

We can determine the gradient of the cost function with respect to each parameter, p, by considering only small changes, ∂p, of p while the other parameters remain fixed as in equation (3.10). We therefore evaluate the cost once with R and t generated for the 5 fixed parameters combined with $+\partial p$ and a second time combined with $-\partial p$ instead.

$$
\begin{aligned}
y1 &= f(p + \partial p) \\
y2 &= f(p - \partial p) \\
\\
\frac{\partial f}{\partial p} &= \frac{(y1 - y2)}{2 * \partial p}
\end{aligned}
\tag{3.10}
$$

If at every iteration we therefore update each rotation and translation parameter by using its gradient and a learning rate. Updating the translation parameters looks like the following code snippet where get_gradient is a function which computes the gradient for each parameter using the method just described. It takes a small step size corresponding to ∂p in equation (3.10).

```
xt_gradient = get_gradient<T, X_T>(t_step, world_transform, stitcher);
yt_gradient = get_gradient<T, Y_T>(t_step, world_transform, stitcher);
zt_gradient = get_gradient<T, Z_T>(t_step, world_transform, stitcher);

xt_diff = - t_learning_rate * xt_gradient;
yt_diff = - t_learning_rate * yt_gradient;
zt_diff = - t_learning_rate * zt_gradient;

world_transform = world_transform.translate(xt_diff, yt_diff, zt_diff);
```

For the rotation paramters we similarly have:

```
xr_gradient = get_gradient<T, X_R>(r_step, world_transform, stitcher);
yr_gradient = get_gradient<T, Y_R>(r_step, world_transform, stitcher);
zr_gradient = get_gradient<T, Z_R>(r_step, world_transform, stitcher);

xr_diff = - r_learning_rate * xr_gradient;
yr_diff = - r_learning_rate * yr_gradient;
zr_diff = - r_learning_rate * zr_gradient;

world_transform = mat4x4<float>::bryant_to_matrix(xr_diff, yr_diff, zr_diff
    ) * world_transform;
```

The rotation and translation parameters have different learning rates as the units for the rotation parameters (radians) are different to those for the translation parameters (mm). The rate of convergence is dependent on these two parameters. If the learning rates are too small this will lead to poor convergence, but if they are too big it is possible for divergence to occur as the minima will be jumped over. Trial and error must be performed to find the optimal learning rates for different data sets.

Convergence can be improved by using an adaptive learning rate. By letting the learning rates self-improve the accuracy of the registration method becomes less reliant on the initial values of the learning rates, although this will still not fully dispel the need to use trial and error for selecting the rough initial values.

The Bold Driver algorithm [19] increases the learning rates by a factor of 1 to 5 percent whenever an update of parameters has decreased the cost function, and sharply decreases the learning rate by 50 percent when an update has caused the cost function to increase as well as resetting the parameters to their previous values. This reset approach is an

extension of back propagation. This method only ever allows the cost function to decrease with every iteration, although it adds the overhead of having to perform more reductions per iteration as the cost function will need to be evaluated after the parameters have been updated to see if it has increased or decreased in value.

This effect of applying the Bold Driver algorithm will speed up convergence as we approach a minimum, but backtrack whenever we take a step too far to enable smaller steps to be taken in the hope of getting as close as possible to the minimum. I apply the Bold Driver method individually to each parameter set and their respective learning rate since updating both learning rates together would imply a linear relationship between them, which is inaccurate. After each parameter set update cycle I the store the world transform matrix with the applied parameter updates in a temporary matrix to perform a calculation of the cost function with the new values and adjust the learning rates as necessary:

```
final_metric = stitcher->reduce(temp_transform);

if (final_metric < error_value) {
  error_value = final_metric;
  world_transform = temp_transform;
  learning_rate *= 1.05;
} else {
  learning_rate *= 0.5;
}
```

3.3 Image Fusion Method

The method of fusion uses the world to source, world transform and target to world matrices to map from the target voxel to the source voxel. By premultiplying a target voxel by these matrices in the order mentioned we can produce the corresponding source vector which will need to be linearly interpolated to account for non-grid values. The fusion method will then need to combine both voxel values and save the resulting value to a new volume. This is done by just taking the maximum of the two voxel intensities. Since both volumes are loaded into RAM when read and the linear interpolation occurs only for the source volume, saving the new values over the old ones of the target volume will produce a new volume without overriding the source and target volumes on disk. This adds the benefit of avoiding having to create new arrays and can make use of the cache for better performance of reading and writing to the same memory locations. A CPU implementation will then look as follows:

```
mat4x4<float> transform = world_to_source *
                          world_transform *
                          target_to_world;

for (int i = 0; i < depth; i++)
for (int j = 0; j < height; j++)
for (int k = 0; k < width; k++) {

    index = k + width * (j + height * i);
    out_vec = transform * vec4<float>(i,j,k,1.0f);
```

```
11
12      unsigned char arg1 = target_data[index];
13      unsigned char arg2 = interpolate(source_data, depth, height, width,
        out_vec);
14
15      target_data[index] = arg1 > arg2 ? arg1 : arg2;
16  }
```

3.4 GPU Implementation

As the CPU image fusion and registration implementation both perform the same calculations for every voxel we can use this to our advantage to parallelize the implementations. Instead of looping over the same calculation for every voxel sequentially, we can perform these calculation in parallel over thousands of threads. Porting these two methods to the GPU in this manner will allow us to achieve real-time performance. In order to run these calculations on the GPU we must first move both volumes into GPU memory.

Moving Data To The GPU

The GPU has different memory regions with different caching behaviour and memory access latency. For both the target volume and the source volume consideration of these features must be taken into account to reduce the overall memory access overhead of the kernels.

The target volume will be stored in the GPU's global memory as the registration and image fusion kernels will need to read target voxel values. Unfortunately, volumes are in **column major order** so warps will not be able to coalesce memory accesses and this will decrease L1 and L2 caches performance relative to using data in **row major order**. This study [24] better highlights the performance differences of these two memory layouts. Moving the target data to global memory is done as following:

```
1 cudaMalloc(&device_data, num_voxels*sizeof(T));
2 cudaMemcpy(device_data1, target_data, num_voxels*sizeof(T),
    cudaMemcpyHostToDevice);
```

If we then pass in this device pointer to each kernel they will be able to access the target volume data. Doing the same for the source volume would then require every thread to perform trilinear interpolation when reading a source voxel. This would require reading a neighbourhood of 8 voxels, so 8 memory reads, and expensive floor and ceil operations in every thread. Interpolation performed in this manner would resemble:

```
1 int x0 = floor(x), x1 = ceil(y);
2 int y0 = floor(x), y1 = ceil(y);
3 int z0 = floor(x), z1 = ceil(y);
4 if (x0 < 0|| y0 < 0 || z0 < 0 || x1 >= depth || y1 >= height || z1 >= width
    )
5     return T();
6
```

```
T xd = a1 - x0;
T yd = a2 - y0;
T zd = a3 - z0;

T source_value =
data[ROW_INDEX(depth, height, width, x0, y0, z0)]*(T(1)-xd)*(T(1)-yd)*(T(1)
    -zd) +
data[ROW_INDEX(depth, height, width, x1, y0, z0)]*xd*(T(1)-yd)*(T(1)-zd) +
data[ROW_INDEX(depth, height, width, x0, y1, z0)]*(T(1)-xd)*yd*(T(1)-zd) +
data[ROW_INDEX(depth, height, width, x0, y0, z1)]*(T(1)-xd)*(T(1)-yd)*zd +
data[ROW_INDEX(depth, height, width, x1, y0, z1)]*xd*(T(1)-yd)*zd +
data[ROW_INDEX(depth, height, width, x0, y1, z1)]*(T(1)-xd)*yd*zd +
data[ROW_INDEX(depth, height, width, x1, y1, z0)]*xd*yd*(T(1)-zd) +
data[ROW_INDEX(depth, height, width, x1, y1, z1)]*xd*yd*zd;
```

To avoid having to manually perform the interpolation we can use a special memory space offered by CUDA called texture memory.

Texture Memory

Texture memory is a segment of read-only global memory with the special characteristic of spatial caching on every one of the GPU's processors chip. Texture reads can be setup so that interpolation occurs directly in the hardware, reducing a read of the source voxel's intensity to the much quicker one line of code:

```
tex3D(tex, x+0.5f, y+0.5f, z+0.5f)
```

Every index needs 0.5 added to it before reads occur so that the centre of the voxel is fetched. To wrap the source volume into a 3D texture we first create a CudaArray which contains the volume data. CudaArray's are memory blocks which are optimized for binding to textures. These arrays are read-only and are structured to fit a **space filling curve** so that texture fetches can exhibit the desired spatial locality. CudaArrays may use pitched memory to pad the array to optimize memory accesses and fulfill coalescing alignment requirements. Creating a CudaArray from volume data is performed as following:

```
cudaArray *d_volumeArray = 0;

const cudaExtent volumeSize = make_cudaExtent(width, height, depth);
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<T>();
checkCudaErrors(cudaMalloc3DArray(&d_volumeArray, &channelDesc, volumeSize)
    );

// copy data to 3D array
cudaMemcpy3DParms copyParams = {0};
copyParams.srcPtr   = make_cudaPitchedPtr((void *)data, volumeSize.width *
    sizeof(T), volumeSize.width, volumeSize.height);
copyParams.dstArray = d_volumeArray;
copyParams.extent   = volumeSize;
copyParams.kind     = cudaMemcpyHostToDevice;
checkCudaErrors(cudaMemcpy3D(&copyParams));
```

Textures which require interpolated fetches must have its read mode set as a normalized float if the volume's data type is not single-precision floating point. This means that fetches will return a float in the range [0,1] for signed data and [-1,1]. Since we want texture accesses outside of the set volume dimensions to return 0 we also have to set the address mode for each dimension to cudaAddressModeBorder. Create and binding the above array to a texture with these requirements is performed as follows:

```
texture<unsigned char, 3, cudaReadModeNormalizedFloat> tex;  // 3D texture

// set texture parameters
tex.normalized = false;                      // access with normalized
    texture coordinates
tex.filterMode = cudaFilterModeLinear;       // linear interpolation
tex.addressMode[0] = cudaAddressModeBorder; // wrap texture coordinates
tex.addressMode[1] = cudaAddressModeBorder;
tex.addressMode[2] = cudaAddressModeBorder;

checkCudaErrors(cudaBindTextureToArray(tex, d_volumeArray, channelDesc));
```

Since the returned value of a texture fetch will be a normalized float we must multiply it by the maximum value of our data type:

```
value = tex3D(tex, x+0.5f, y+0.5f, z+0.5f) * std::numeric_limits<T>::max()
```

Kernels can now access the source data through a texture reference.

3.4.1 Registration Kernels

The registration method computes a similarity metric between two volumes. A computation is performed for every voxel which then needs to be reduced to one value (through addition in our case). The GPU reduction implementation can therefore be broken down into two kernel calls, one for calculating the squared difference between a voxel in the target volume and its corresponding voxel in the source volume, and one for summing the squared differences.

Squared Difference Kernel

This kernel will have to compute the squared difference between a target voxel and the corresponding source voxel. We only need to compute this for voxels inside the frustrum (values greater than 0) so we will also need to keep track of how many valid voxels there are to later calculate the mean. To do this we need to allocate two arrays on the device. The first one will store the computed squared difference for every voxel and the second one will simply store a 1 for valid voxels (pairs of voxels both in their volumes frustum), and a 0 for invalid ones. Summing the first array will then give the sum of squared differences, and summing the second array will give the number of voxels. The allocation is performed as following:

```
cudaMalloc(&device_sum_diff, num_voxels*sizeof(float));
cudaMalloc(&device_sum_count, num_voxels*sizeof(float));
```

When invoking a kernel the number of threads per block and number of blocks can be given in three dimensions. Doing so allows us to obtain the 3D indices of the GPU thread executing the kernel, and use them to map each thread to a particular voxel of the target data:

```
unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
unsigned int z = blockDim.z * blockIdx.z + threadIdx.z;

unsigned int index = z + width * (y + height * x);
```

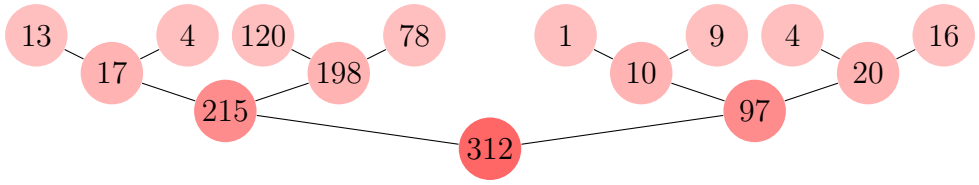
Using this approach instead of making the thread blocks and grid one dimensional is preferable as the texture fetches require the indices to be 3D. Working backwards from a flat index requires the use modulo arithmetic which is expensive on the GPU, so its avoidance is desirable.

As discussed in the Background chapter every block is assigned a number of threads which are then split into groups of 32 threads. These take turns executing as Single instruction, multiple data (SIMD) on their assigned SM. If the block size is not divisible by 32 empty "ghost" threads will be added onto a warp when it is executing. This means that in cases where the volume data is not divisible by 32 we must check the indices to be within the dimension of the volume. Otherwise ghost threads could contribute to the calculation of the cost function. We must also check that both the target voxel's intensity and the source voxel's intensity are non-zero. We then proceed to store the squared difference between their values in the device_sum_diff and a 1 in device_sum_count array (both arrays are passed in to the kernel as pointers). If any of the conditions are not met we instead store 0s in both arrays. We compute an offset, tid, to index the device arrays which is unique for every thread. The remainder of the kernel therefore looks like:

```
float arg1;

if (z < depth && y < height && x < width && (arg1=data1[index])) {
  vec4<float> v = m * vec4<float>(x,y,z,1.0f);
  float arg2 = (tex3D(tex, v.a1+0.5f, v.a2+0.5f, v.a3+0.5f) * n_max);

  device_sum_diff[tid] = arg2 ? (arg1 - arg2) * (arg1 - arg2) : 0.0f;
  device_sum_count[tid + threads_per_block] = arg2 ? 1 : 0.0f;
} else {
  device_sum_diff[tid] = 0;
  device_sum_count[tid + threads_per_block] = 0;
}
```

Reduction Kernel

After the squared difference kernel executes the next step is to add up values in the two device arrays. Reducing data with CUDA kernels involves recursively combing groups (blocks on the GPU) of the data so that every kernel execution reduces the amount of elements left to reduce:



All threads in the same block have access to a small memory region, shared memory, which resides directly on the chip of the SM the block is assigned to. Each thread within the block can copy the array value from the GPU global memory into shared memory and the reduction can then be performed in this much lower latency memory. Since every block reduces a group of threads to one value, after a kernel is executed the array can be considered to shrink in size to the number of blocks that the kernel was launched with. We can therefore keep scaling the size of the array down this way until only a single value is left, or the amount of values left is below a certain threshold where launching another iteration of the kernel requires a greater overhead then performing the last few additions on the CPU.

If each thread in a block is assigned a shared memory location in contiguous order global memory accesses can be coalesced. We do this by specifying the thread blocks and grid as one dimensional when we launch the kernel and assign each thread an index using: $blockIdx.x * blockDim.x + threadIdx.x$. Now that all the required data is in shared memory we can use a for loop that iteratively folds the values half, adding the second half of memory to the first. We do this until only one value remains at the first shared memory address holding the total sum of the block. It is important to synchronize all the block's threads after each fold is performed. This means all threads in the block wait until they have reached the same execution point before being able to continue. This is needed here since threads will be reading from shared memory addresses that other threads must write to first. This can be done with a call to __syncthreads():

```
__global__ void getsum(int n, float *d_sum) {
  extern __shared__ int shared_data[];

  // each thread loads one element from global to shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  shared_data[tid] = (i < n) ? d_sum[i] : 0;
  __syncthreads();

  for (unsigned int s=THREADS_PER_BLOCK; s>0; s>>=1) {
    if (tid < s) {
      shared_data[tid] += shared_data[tid + s];
    }
    __syncthreads();
  }

  if (tid == 0) {
    d_sum[blockIdx.x] = shared_data[0];
  }
}
```

This kernel can be further optimized by unrolling the loop. If we know the number of threads per block at compile time it can be passed as a template argument. Since warps perform instructions as SIMD, the first warp of the thread block, (first 32 threads), will

not need to call __syncthreads() after each fold. This is the main benefit of unrolling the loop as synchronizing the threads is expensive. Also to avoid half of the threads being idle in the first unraveled loop iteratoin we can perform the first addition between two values loaded from global memory instead of just one. This allows all threads to be active for at least one unraveled loop iteration. The number of blocks for the kernel launch will have to be halved to allow for this. After these optimizations are applied we are left with the following kernel:

```
template <unsigned int block_size>
__global__ void getsum(unsigned int n, float *device_sum) {

    volatile extern __shared__ float shared_data[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(block_size*2) + tid;

    float sum = (i < n) ? device_sum[i] : 0;

    if (i + blockDim.x < n)
        sum += device_sum[i+blockDim.x];

    shared_data[tid] = sum;

    __syncthreads();

    if(block_size >=1024) { if(tid < 512) { shared_data[tid] = sum = sum + shared_data[tid + 512]; } __syncthreads(); }
    if(block_size >= 512) { if(tid < 256) { shared_data[tid] = sum = sum + shared_data[tid + 256]; } __syncthreads(); }
    if(block_size >= 256) { if(tid < 128) { shared_data[tid] = sum = sum + shared_data[tid + 128]; } __syncthreads(); }
    if(block_size >= 128) { if(tid <  64) { shared_data[tid] = sum = sum + shared_data[tid +  64]; } __syncthreads(); }

    if (tid < 32) {
        if (block_size >= 64) shared_data[tid] = sum = sum + shared_data[tid + 32];
        if (block_size >= 32) shared_data[tid] = sum = sum + shared_data[tid + 16];
        if (block_size >= 16) shared_data[tid] = sum = sum + shared_data[tid +  8];
        if (block_size >=  8) shared_data[tid] = sum = sum + shared_data[tid +  4];
        if (block_size >=  4) shared_data[tid] = sum = sum + shared_data[tid +  2];
        if (block_size >=  2) shared_data[tid] = sum = sum + shared_data[tid +  1];
    }

    if (tid == 0) {
        device_sum[blockIdx.x] = shared_data[0];
    }
}
```

Now that we have our optimized kernel we iteratively launch it until the number of elements that need to be added are below a certain threshold:

```
float reduce_sum(unsigned int n, float *device_sum)

  // n is number of elements which need to be added

  unsigned int num_blocks, num_threads;

    get_num_blocks_threads(n, num_blocks, num_threads);

  while (n > CPU_SUM_THRESHOLD) {

        unsigned int shared_mem_size = (num_threads <= 32) ?
        2 * num_threads * sizeof(float) :
        num_threads * sizeof(float);

        getsum<num_threads> <<<num_blocks, num_threads, shared_mem_size>>>(
    n, device_sum);


        // Calculate blocks, threads per block and number of elements left
    to sum
        get_num_blocks_threads(n, num_blocks, num_threads);
        n = (n + (num_threads*2-1)) / (num_threads*2);
     }
```

Before every iteration the get_num_blocks_threads updates the number of threads and blocks needed for the next kernel launch using arithmetic based on the number of elements left to add:

```
void get_num_blocks_threads(unsigned int n, unsigned int &num_blocks,
    unsigned int &num_threads) {
     num_threads = (n < MAX_THREADS*2) ? nextPow2((n + 1)/ 2) : MAX_THREADS;
     num_blocks  = (n + (num_threads * 2 - 1)) / (num_threads * 2);
}
```

Reduction in Squared Difference Kernel

Since we have seen how additions can be performed within a block we can perform the first iteration of block additions directly in the Squared Difference Kernel. This saves us the overhead of one extra execution of the reduction kernel. Since the launch configuration for this kernel is three dimensional, the flattened indices in a warp do not access contiguous memory locations so unraveling the loop is less beneficial as __synchthreads() will still have to be called after every fold in the first warp. By adding the following definitions at the beginning of the Squared Difference Kernel:

```
extern __shared__ float s_data[];

unsigned int block_id = blockIdx.z + gridDim.z * (blockIdx.y + gridDim.y *
    blockIdx.x);
unsigned int threads_per_block = blockDim.x * blockDim.y * blockDim.z;
```

and the following at the end:

```
__syncthreads();

```

```
for (unsigned int s=THREADS_PER_BLOCK/2; s>0; s>>=1) {
    if (tid < s) {
        s_data[tid] += s_data[tid + s];
    s_data[tid + threads_per_block] += s_data[tid + threads_per_block + s];
    }
    __syncthreads();
}

if (tid == 0) {
    device_sum_diff[block_id] = s_data[0];
    device_sum_count[block_id] = s_data[threads_per_block];
}
```

the kernel now performs a block reduction for both the overlapping voxel count and the similarity metric. The final reduced value for every block of threads is then stored in their respective device array.

Piecing The Kernels Together

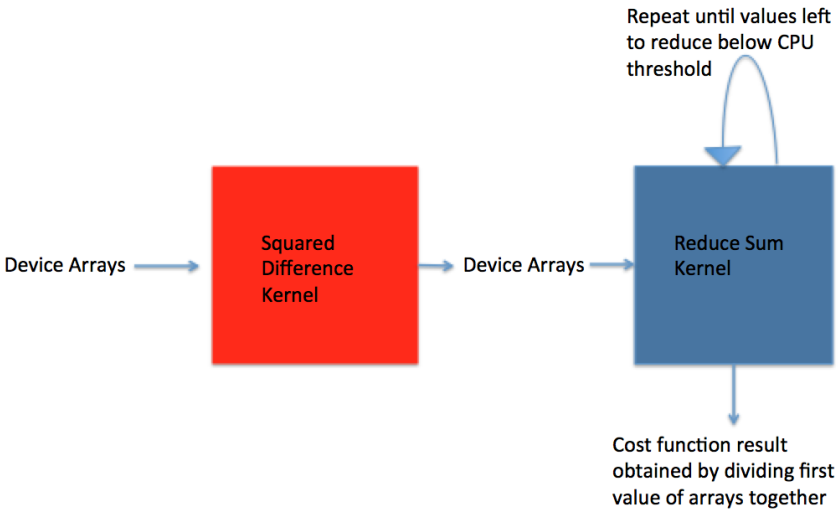


Figure 3.8: Execution order of kernels.

Now all that is left to do is calculate the launch parameters for the squared difference kernel, called getdiff, and launch it followed by calls to the reduction kernel for both arrays. This can be visualised in Fig 3.8. The dimensions are computed depending on how many threads we want in a block and how many blocks we want. If for example we want 256 threads per block then the dimensions of the grid and block are computed as following:

```
int X = (depth+8-1)/8;
int Y = (height+8-1)/8;
int Z = (width+4-1)/4;

dim3 Dg(X, Y, Z);
dim3 Db(8, 8, 4);
```

Next the amount of shared memory per block needed is computed. Each block adds up

two values per thread so the amount needed is two times the number of threads per block. The kernel can be launched with the following arguments:

```
unsigned int shared_mem_size = 2*Db.x*Db.y*Db.z*sizeof(float);

getdiff<T><<<Dg, Db, shared_mem_size>>>(device_sum_diff, device_sum_count,
    device_target_data,
                                        depth, height, width,
                                        std::numeric_limits<T>::max(),
    transformation_matrix);
```

The number of elements that will need to be summed is equal to the amount of blocks the getdiff kernel was invoked with, so we pass that value into the previously defined reduce_sum function to reduce the two arrays and compute the value of the cost function:

```
unsigned int n = X * Y * Z;

float result = reduce_sum(n, device_sum_diff);
float count = reduce_sum(n, device_sum_count);

if (count == 0) {
  return 0;
} else {
  return result/count;
}
```

3.4.2 Image Fusion Kernel

An image fusion implementation requires a method for mapping two volumes into one coordinate system and a method of calculating the intensity of a final voxel from the intensities of a target and source voxel. My image fusion kernel takes a transformation matrix, such as the one outputted by my registration method, and a functor encapsulating how to combine the intensities. Using a functor allows the member function call to be inlined since it will be known at compile time, whereas a memory access would be needed if a runtime function pointer was used instead.

My implementation is parallelized on the GPU so that the computation required for every voxel is performed in its own thread, the same way the Squared Difference Kernel works in my registration implementation. The source voxel is computed by applying the transformation to the target voxel (inverse mapping) and the intensity of the target voxel is combined with the interpolated intensity of the source voxel using a functor:

```
template <typename T, typename Fuse_method>
__global__ void getmax(int n, T *data1, int depth, int height, int width,
    int n_max, mat4x4<float> m, Fuse_method fuser)
{
    unsigned int x = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int y = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int z = blockDim.z * blockIdx.z + threadIdx.z;

    if (z < depth && y < height && x < width) {

        unsigned int index = z + width * (y + height * x);
```

```
11
12          vec4<float> v = m * vec4<float>(x,y,z,1.0f);
13
14          data1[index] = fuser((tex3D(tex, v.a1+0.5f, v.a2+0.5f, v.a3+0.5f) *
        n_max), data1[index]);
15      }
16  }
```

The kernel follows the same approach of the squared difference kernel in section 3.4.1. It first calculates the x, y and z indices using the 3 dimension blockDim, blockIdx and threadIdx provided to each thread on the GPU. Since all blocks divide its threads into warps of size 32, adding "ghost" threads if the block size specified is not a multiple of 32, we must check that x, y and z are within the dimensions of the volume. This avoids these "ghost" threads accessing invalid memory addresses resulting in segmentation faults when the volume dimensions are not multiples of 32.

The source voxel is computed on line 12 by wrapping the target voxel indices in a vector and premultiplying it by the transformation matrix. The indices of the resulting vector are then used to retrieve the intensity of the source voxel from texture reference that is binded to the source volume as explained in 3.4. On line 14 the target voxel is then updated with the result returned by passing the two values through the inputted functor. As with the CPU implementation the target volume is overwritten with the new volume.

A functor implementation for combining data with the maximum intensity would look like:

```
1  template <typename T>
2  class F_max {
3  public:
4    __device__
5    T operator()(float arg1, float arg2) {
6        return (T) (arg1 > arg2 ? arg1 : arg2);
7    }
8  };
```

Executing the image fusion kernel is then as follows for a 3D grid and 512 threads per block:

```
1  fuse(mat4x4<float> transform) {
2
3      int X = (depth+8-1)/8;
4      int Y = (height+8-1)/8;
5      int Z = (width+8-1)/8;
6
7      dim3 Dg(X, Y, Z);
8      dim3 Db(8, 8, 8);
9
10     getmax<T, F_max<T> ><<<Dg, Db>>>(device_data1, depth, height, width,
       std::numeric_limits<T>::max(), transform, F_max<T>());
11 }
```

Chapter 4

Evaluation

The evaluation of this project can be divided into two parts: evaluation of the speed of the implementation and evaluation of the alignment quality produced by the registration implementation. The goal of this project has been to create a stitching implementation capable of running in real-time, but the quality of the alignment is even more important. A registration implementation that produces incorrect alignments is far less preferable to one that lacks fast enough execution speed. The latter can still be used for offline volume processing while the former cannot. Also execution speeds can be scaled by improving the hardware whilst the quality of alignment is implementation dependent.

The execution speed depends on the hardware being used, the size of the volumes being fused and the number of iterations of gradient descent performed. The quality of alignment depends on the initial transformation matrices supplied by the acquisition system and the number of iterations performed. For a given amount of iterations a faster runtime does not correspond to a better alignment quality.

4.1 Measuring Alignment Quality

The quality of alignment is measured by seeing how far the alignment produced by the registration method deviates form the correct alignment. To do this we can track the euclidean distance of a group of n voxels to the optimal alignment before the registration and then after. As the primary features being observed in ultrasound images are situated in the middle of the image, we can select the voxels forming a central sphere in the volume as depicted in Fig 4.1. We can then compare the error (distance) after the registration with respect to the error before it to assess to how well it performed. These calculations are shown with equations (4.1) and (4.2) where d is the euclidean distance.

$$\begin{aligned} \{vs\_noreg\} &= M_{truth}^{-1} * M_{total} * \{vs\} \\ \text{error before reg} &= \frac{1}{n} \sum_{i=1}^{n} d(\{vs\_noreg\}_i, \{vs\}_i) \end{aligned} \tag{4.1}$$

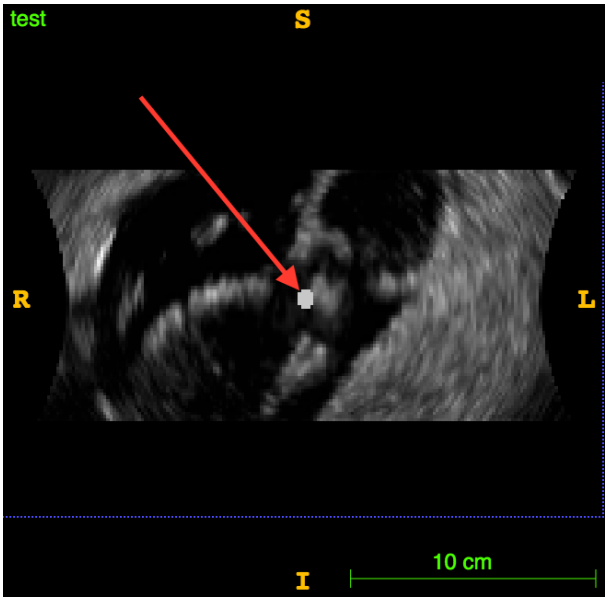


Figure 4.1: Spherical set of voxels used to measure alignment quality

$$\begin{aligned} \{vs\_reg\} &= M_{truth}^{-1} * M_{reg} * \{vs\} \\ \text{error after reg} &= \frac{1}{n}\sum_{i=1}^{n} d(\{vs\_noreg\}_i, \{vs\}_i) \end{aligned} \tag{4.2}$$

Using a sample of 121 volumes acquired in real-time supplied with their ground truth matrices enables the accuracy of my registration implementation to be analyzed in the next few sections. If we sort by pairs of volumes acquired with the same time difference in our sample we can further categorize our analysis by frame rate.

4.1.1 Alignment Quality vs Number of Iterations

Every iteration of gradient descent aims to update the world transform's rotation and translation parameters such that the cost function decreases, bringing us closer to the optimal alignment. As the algorithm performs more iterations, iteratively bringing the cost function to a minimum, it will experience smaller relative changes per successive iteration seen as in Fig 4.2. In order to measure the necessary amount of iterations required for convergence to an alignment error value, we need to track the relative difference of this error between iterations. We define convergence to be when the relative change is below 10^{-5} between successive iterations. Using this approach it is important to ignore successive iterations where back-propagation occurs, that is when the parameters have not moved meaning the relative change of the alignment error will be 0.

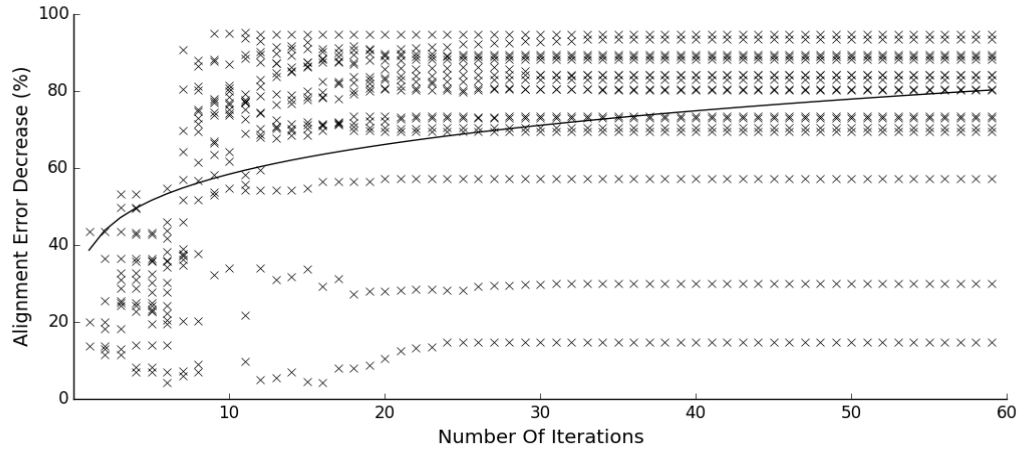


Figure 4.2: Visualising the error decrease with respect to the number of iterations of gradient descent performed.

Using our sample set of volumes acquired in real-time we can then visualise the relative frequencies of the iterations for convergence over our sample data in Fig 4.3 for a frame rate of 3.7 Hz. The average number of iterations needed is **29.4**. Viewing the mean alignment error before registration (Fig 4.4) for volumes grouped by the number of iterations for convergence does not show any meaningful correlation.

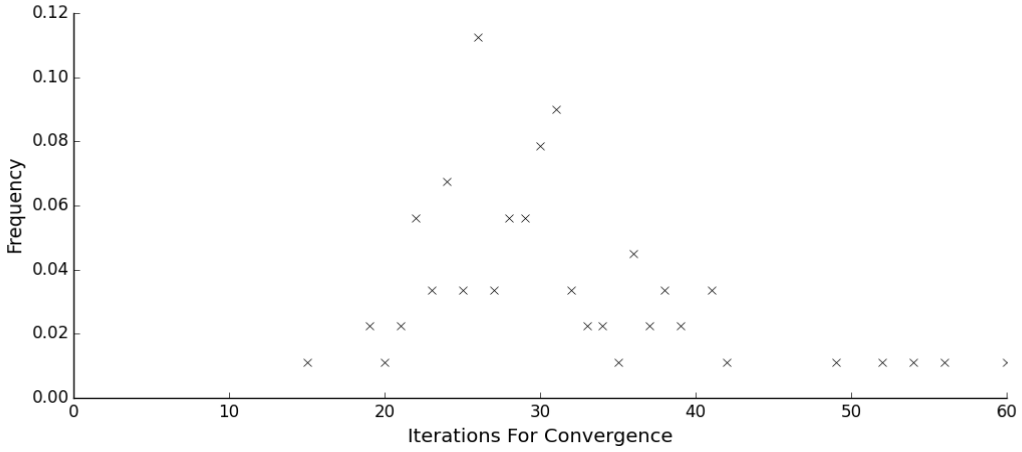


Figure 4.3

If we repeat the above search for the minimum iterations for convergence for different frame rates we obtain the results in 4.1. If the results are then plotted (Fig 4.5) a small positive correlation between the frame rate and the number of iterations needed for convergence can be seen.

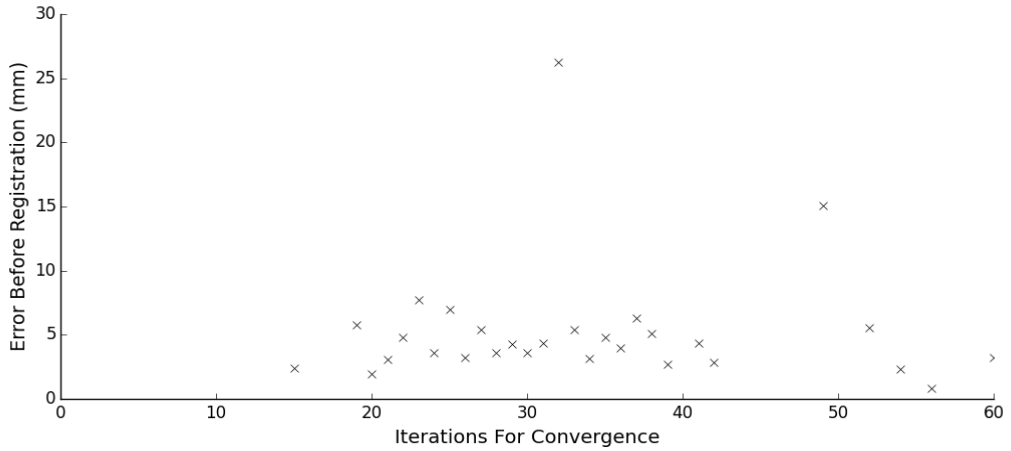


Figure 4.4: Mean Alignment Error vs Number of Iterations for Convergence

Frame Rate (Hz)	Avg Iterations For Convergence
3.70	29.4
1.85	29.1
1.22	29.3
0.92	30.3
0.74	30.4
0.61	31.5
0.52	30.8
0.46	31.5
0.41	30.1
0.37	31.5

Table 4.1

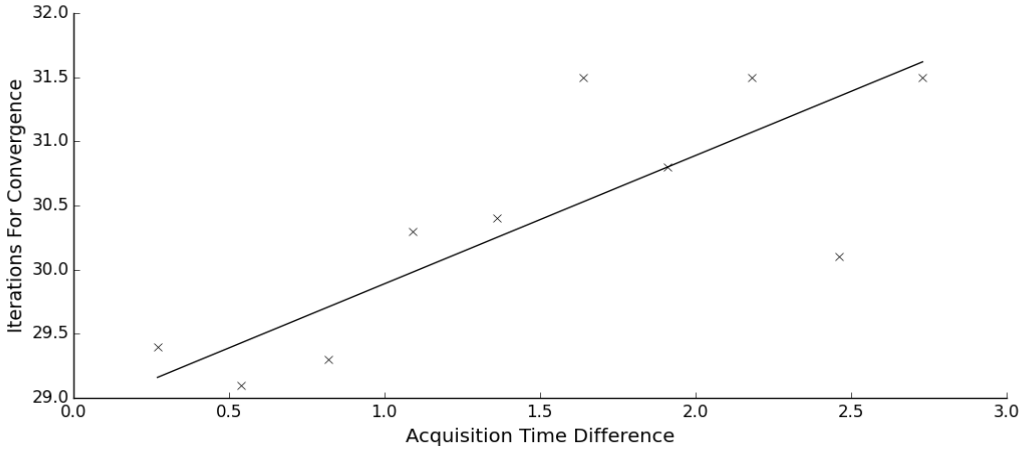


Figure 4.5: Plot of information from table 4.1

4.1.2 Relationship Between Alignment Error and MSE

As the cost function is what guides the improvement of the alignment it is useful to see how it moves with respect to the alignment error in millimetres. By plotting decreases in the MSE with decreases in alignment error before and after registering volumes in our sample set we can visualise the relationship between MSE improvements and alignment improvements. Figure 4.6 shows the results of doing volumes acquired at a frame rate of 0.92 Hz.

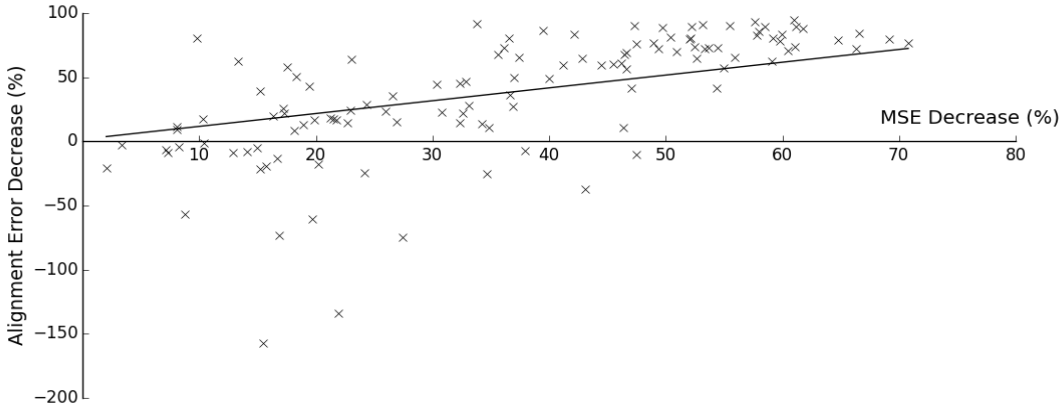


Figure 4.6

We can clearly see a positive linear relationship between the MSE improvement and alignment error improvement indicating its validity as a cost function when registering volumes acquired sequentially. It also has the advantage of only requiring a single pass of all the voxels in the target and source volume whereas more complicated similarity metrics may require multiple passes, increasing the execution time of the registration. The points below the x axis indicate when divergence from the optimal alignment has occurred, demonstrating a major pitfall of gradient descent using a simple cost function like MSE. It will contain local minima which can cause the algorithm to diverge from the global minimum and instead converge towards a local minima. The frequency of this is discussed in section 4.1.3.

4.1.3 Effect of Tracker Noise

Since the goal of the registration method is to suppress the alignment inaccuracy due to the tracker noise it is useful to try and quantify how well this can be accomplished. If we take the ground truth transformation matrices between the volumes in our sample set and artificially apply them with noise we can simulate the noise in the tracker. The noise will be applied in the form of rotations along the x, y and z axis by angles ϕ_1, ϕ_2, ϕ_3 respectively and translations along the x,y and z axis by dx, dy, dz millimeters respectively. For a noise range of $\phi_1, \phi_2, \phi_3 \in (-20°, 20°)$ for the rotation parameters and $dx, dy, dz \in (-10, 10)$ for the translation parameters we can use the minimum iterations for convergence computed in section 4.1.1 to measure the effect of noise on the different frame rates. 1000 noisy

samples were run for each frame rate. The results for a frame rate of 3.7 Hz has been plotted in Fig 4.7.

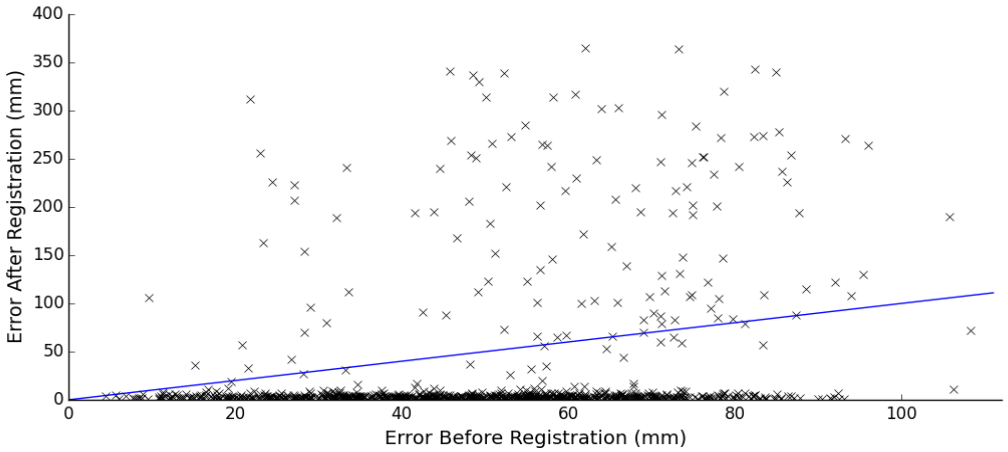


Figure 4.7: Results of 1000 random noise applications.

Points in Fig 4.7 below the line correspond to samples where the registration error converged towards the optimal alignment while points above correspond to points which diverged from it. If we consider the capture range to be the range of initial alignment errors where the registration method improves the error with a 95% from Fig 4.8 we can see this range is [0,75) for a frame rate of 3.7 Hz.

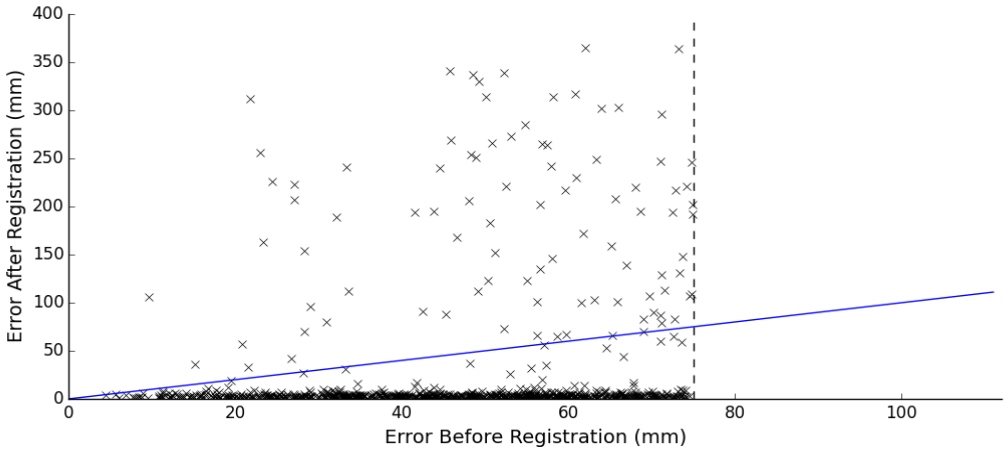


Figure 4.8: Capture range of the results.

Doing the same for volumes acquired at different frame rates using their respective minimum number of iterations for convergence from section 4.1.1 produces the results in table 4.2

Table 4.2 demonstrates how slower acquisition rates have, on averae, smaller capture ranges. This can be do to the increase amount of non-rigid deformation occuring in a larger time interval, meaning the optimal alignment no longer corresponds to the minimum of the cost function.

Frame Rate (Hz)	Avg Iterations For Convergence	Capture Range (mm)
3.70	30	[0,75)
1.85	30	[0,53)
1.22	30	[0,48)
0.92	31	[0,5)
0.74	31	[0,11)
0.61	32	[0,5)
0.52	31	[0,5)
0.46	32	[0,8)
0.41	31	[0,5)
0.37	31	[0,8)

Table 4.2: Capture range for different frame rates

Frame Rate (Hz)	Number of Iterations	Avg. Tracker Error (mm)	Avg. Reg Error (mm)	Avg. Increase (mm)	% Under 1mm
3.70	30	4.47	3.16	1.31	15.00
1.85	30	6.82	3.88	2.94	9.24
1.22	30	8.75	4.46	4.29	11.86
0.92	31	10.61	5.22	5.38	5.13
0.74	31	12.30	5.72	6.58	2.59
0.61	32	13.81	7.48	6.32	1.74
0.52	31	15.20	6.73	8.47	0.88
0.46	32	16.54	12.16	4.38	3.54
0.41	31	17.93	13.14	4.79	6.25
0.37	31	19.29	13.91	5.38	2.70

Table 4.3: Results registering volumes with tracker data

4.1.4 Alignment With Tracker Data

To conclude, if we use the optimal iteration number for the different frame rates we can calculate the average alignment error using the tracker data before and after registration (Avg. Reg Error) for our volume sample. The final results are shown in table 4.3. Also shown in the table is the percentage of error values after registration which were under 1mm. We can consider these to be alignment errors not visible to the human eye and therefore successfully registered.

4.1.5 Discussion

While all frame rates on average underwent an alignment improvement after registration, the final alignment error is slightly above desired when dealing with tracker data. Table 4.3 results also do not indicate that a significant percentage of the volumes were registered to a visual satisfaction as the percentages of post registration errors under 1 mm were low.

Weaknesses

The results conclude a weakness of using the MSE as the cost function. While simple to implement, it does not guarantee its minimum corresponds to that of the optimal alignment and also produces a large quantity of local minima that we have seen negatively affect the optimizer. Use of other cost functions could be investigated as an extension to this implementation as well as variations of gradient descent as the optimizer. Gradient descent is the simplest optimization technique available as its speed comes at the expense of not always being accurate. From the results we can conclude that the chosen optimizer and cost function do not combine well.

Another weakness that can be understood from the results is poor performance for lower frame rates. This can be understood to derive from the higher amounts of non-rigid deformation that can occur in image structures acquired with a larger delay. Modelling these transformations as rigid does not allow the cost function to guarantee that its global minimum corresponds to the optimal alignment of the images. Switching to a non-rigid one would sacrifice execution speed for accuracy. As there is a correlation between higher frame rates and increases in registration accuracy (4.3) we could expect registration results to increased if the frame rate was higher than 3.7 frames per second.

4.2 Execution Speed

The execution speeds reported in this section were acquired running on host architecture with an Intel(R) Xeon(R) CPU E5606 @ 2.13GHz with 8GB RAM and using the following GPU:

```
GeForce GTX 780: 3.5
  Global memory:    3071mb
  Shared memory:    48kb
  Constant memory: 64kb
  Block registers: 65536

  Warp size:         32
  Threads per block: 1024
  Max block dimensions: [ 1024, 1024, 64 ]
  Max grid dimensions:  [ 2147483647, 65535, 65535 ]
```

4.2.1 GPU Data Transfers

GPU data transfer times are dependent on the size of the volumes. Using sample data I measured the different execution times of 6 different sizes. The following operations are measured and reported in table 4.4.

1. Time to copy the target volume from the host to the GPU global memory
2. Time to copy the target volume back from the GPU global memory to the host
3. Time to copy the source volume into a CudaArray and bind it to a texture

Data Size (MB)	1	2	3
1.1	0.6ms	0.5ms	0.9ms
2.2	0.9ms	0.8ms	1.1ms
4.3	1.7ms	1.4ms	1.9ms
8.7	3.1ms	2.6ms	3.6ms
17.3	4.7ms	8.6ms	6.4ms
34.6	8.15ms	16.2ms	12.5ms

Table 4.4: Execution times of data transfers to and from the GPU.

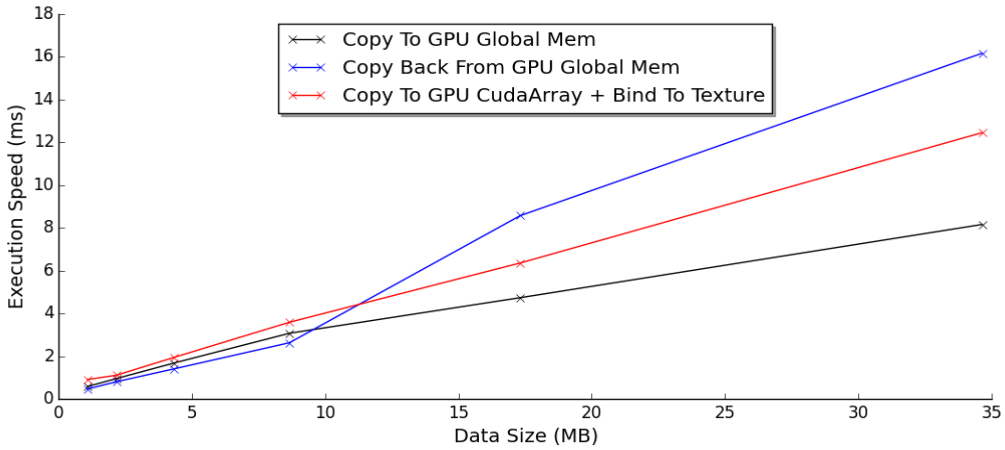


Figure 4.9: Relative speeds of the data transfers

4.2.2 Registration

The registration execution time also follows a linear relationship with the size of the data as the number of threads executed on the kernel is equal to the number of voxels. The times taken for one computation of the cost function on the CPU implementation and in contrast the GPU kernels are reported in table 4.6. The relative speedup is plotted in Fig **??** and shows a positive correlation between the data size and the speedup of using the GPU. This can be interpreted as the overhead of having to calculate the MSE in two steps on the GPU is reduced when compared to performing the additions in place in the CPU implementation as the numbers of voxels increase.

Data Size (MB)	Number Of Voxels	GPU (ms)	CPU (ms)	GPU Speedup
1.1	1108640	0.4	124	x308.7
2.1	2217280	0.8	342.8	x426.9
4.2	4434560	1.5	707.9	x462.0
8.5	8869120	2.9	1,421.3	x483.3
16.9	17738240	5.7	2,781.5	x487.1
33.8	35476480	11.4	5,445.3	x480.0

Table 4.5: Cost function execution time on the CPU compared to the GPU

4.2.3 Fusion

Due to the similarity between the first kernel of the registration method and the fusion kernel similar results to section 4.2.2 can be expected. The fusion kernel was measured using a functor which simply selects the maximum of the target voxel intensity and the source voxel intensity. While the GPU fusion implementation is incredibly quick, this approach to the fusion does not lend itself to more complex calculations where more information is needed such as the mean voxel intensity of the volumes.

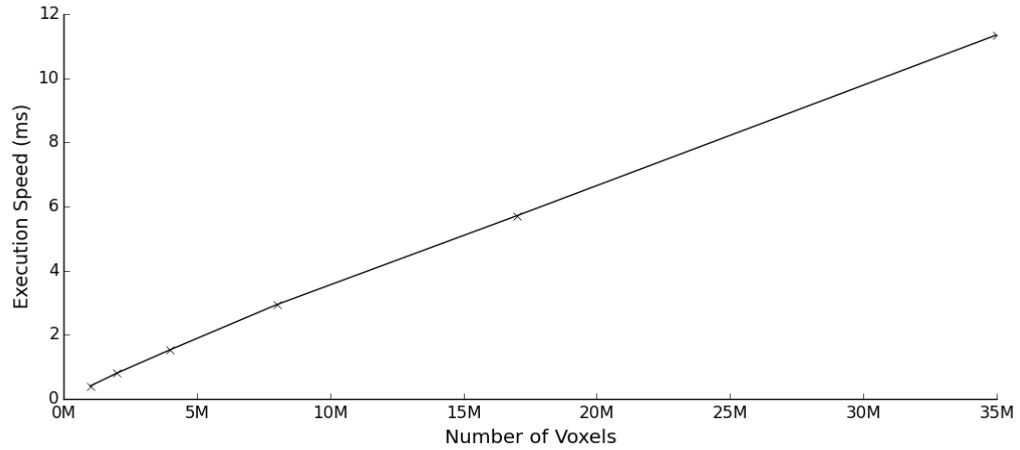


Figure 4.10: Registration time vs number of voxels

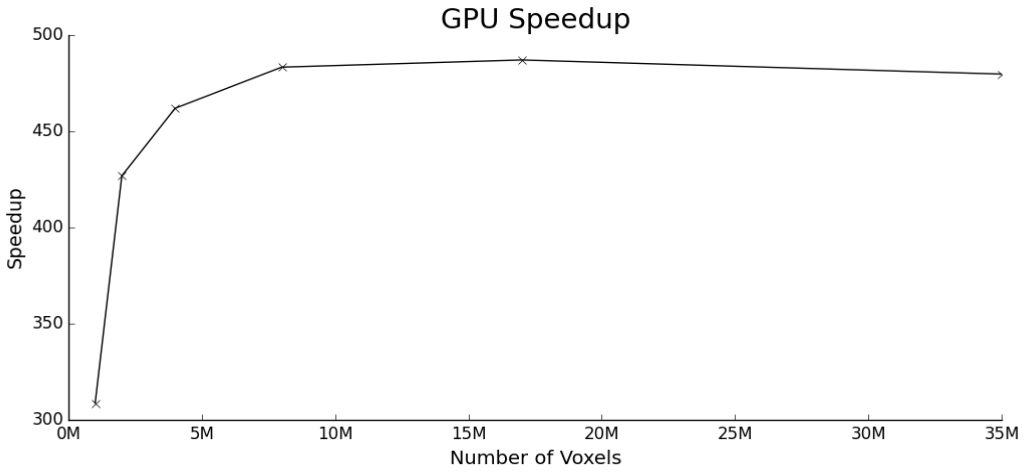


Figure 4.11: Speedup provided by the GPU relative to the number of voxels.

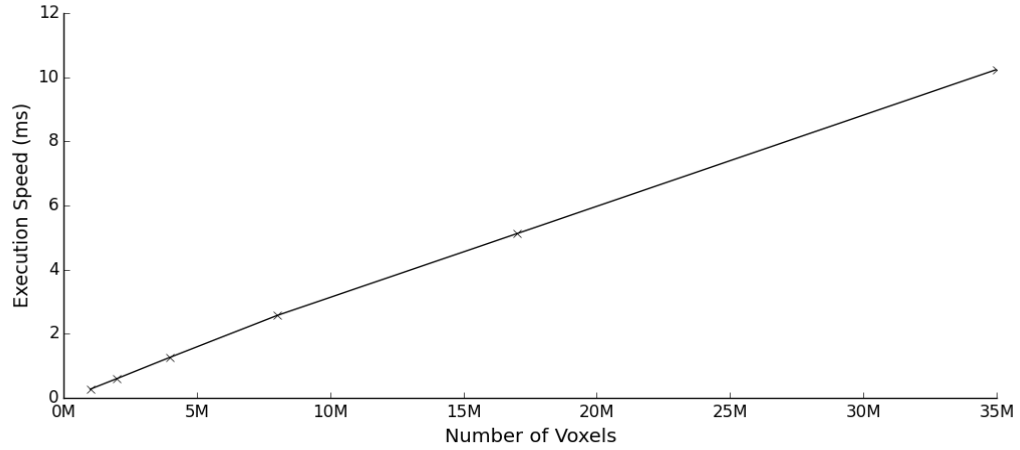


Figure 4.12: Fusion execution time on the CPU compared to the GPU

Data Size (MB)	Number Of Voxels	GPU (ms)	CPU (ms)	GPU Speedup
1.1	1108640	0.3	118.6	x437.5
2.1	2217280	0.6	336.4	x560.0
4.2	4434560	1.3	718.5	x567.0
8.5	8869120	2.6	1,543.8	x597.9
16.9	17738240	5.1	2,832.0	x553.0
33.8	35476480	10.2	5.479.8	x535.1

Table 4.6: Fusion execution time on the CPU compared to the GPU

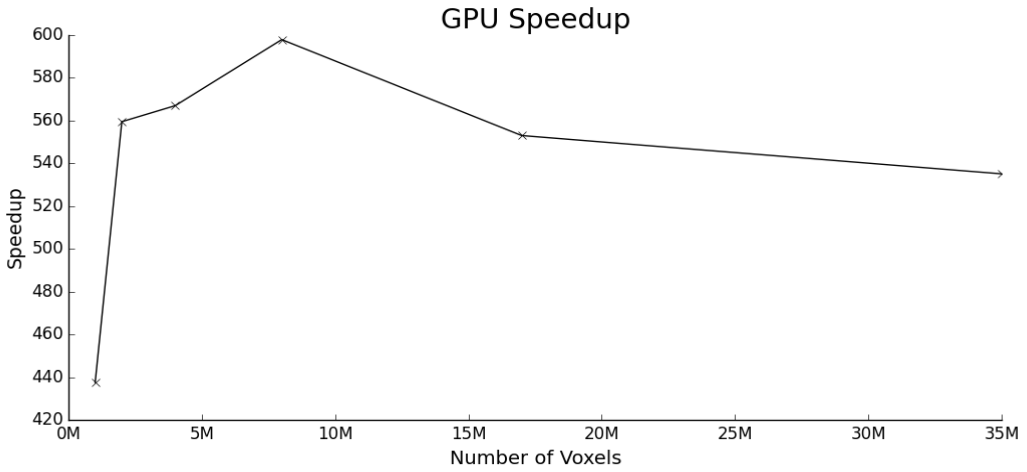


Figure 4.13: Speedup provided by the GPU relative to the number of voxels.

4.2.4 Discussion

In section 4.1.1 we found the optimal number of iterations for different acquisition rates averaged out to around 30. Every iteration requires 14 evaluations of the cost function, plus one initial one, so for real-time data of expected size 9MB combining the execution times for the three steps above we reach a total run time of about **2 seconds** for the specific hardware used. This is too slow to be used in an acquisition system obtaining frames at 4Hz. To reduce the total time to the desired 250ms modifications are discussed below.

Using a more powerful GPU can accelerate the runtime reported here by a magnitude of 5 to 10 times. The G3D mark (metric used to evaluate) of the GeForce GTX 780 is 8,014, significantly lower than new high-end CUDA capable GPUs release by Nvidia such as the GeForce GTX 1080 (12,613) or the GeForce GTX 980 Ti (11,556). Nvidia's upcoming Pascal GPU is also expected to top these other models with an expected 3840 CUDA cores.

The runtime can also be accelerated through the use of multiple GPUs by splitting the volume into subsets of voxels. Each subset can then be processed by their own GPU. This approach is depicted in Fig 4.14. Doing this would divide the runtime by the number of GPUs used. If we use 4 GeForce GTX 780 the runtime is reduced to 500ms while we furthermore use 8 GPUs we reach the desired threshold of 250ms for our acquisition system.

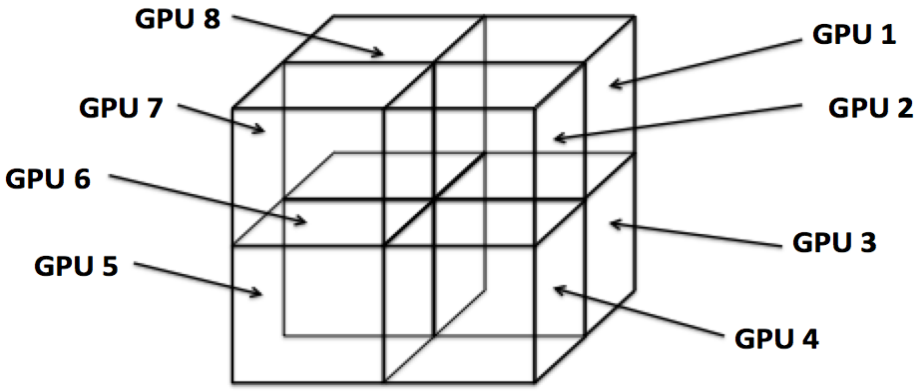


Figure 4.14: Allocating different parts of the volume to be processed by their own GPU.

By combining the two approaches of using more powerful GPU models and modifying our implementation to a multi-GPU implementation, it is easy to see how the implementation created in this project could by accelerated for uses in real-time systems. A run-time of **under 100ms** is a realistic expectation meaning it could run alongside an acquisition system with a frame rate of at least 10Hz.

Chapter 5

Conclusion and Future Work

This project has resulted in a successful image fusion implementation capable of running in real-time. It will be free for incorporation into researchers' acquisition systems at St Thomas' Hospital. This project has also explored an implementation for registering ultrasound images utilizing gradient descent as an optimizer and the mean squared error as the cost function. While results for this method were not entirely promising, it has still served as a proof of concept which has highlighted a few weakness to the approach.

On a personal level this project has exposed me to the rapidly evolving world of GPU performance acceleration. It has given me a strong foundation in GPGPU programming which I will be able to carry with me to future projects.

5.1 Future Work

- Modify the image fusion and registration implementations to be multi-GPU to achieve greater speedup.
- Explore the use of different cost functions such as normalized cross-correlation to improve the alignment accuracy of the registration method.
- Explore different forms of learning rate adaptation for the gradient descent method to try and improve its convergence.
- Modify the image fusion method to be able to extend the field of view by creating larger output volumes which include all of the source volume.
- Explore the accuracy of the registration method using data acquired with higher frame rates than 4 frames per second.
- Extend the current registration method to enable group-wise volume registrations.

Bibliography

[1] Vincent Chan and Anahi Perlas. "Basics of Ultrasond Imaging". 2011 URL:http://www.springer.com/978-1-4419-1679-2

[2] "How ultrasound imaging works explained simply". (visited on November 9th 2015) URL:http://www.howequipmentworks.com/ultrasound_basics/

[3] Christian Wachinger. "Three-Dimensional Ultrasound Mosaicing". In: Med Image Comput Comput Assist Interv 2007;10(Pt 2):327-35.

[4] Laura J. Brattain and Robert D. Howe. "Real-Time 4D Ultrasound Mosaicing and Visualization" Medical Image Computing and Computer-Assisted Intervention – MICCAI 2011 Volume 6891 of the series Lecture Notes in Computer Science pp 105-112

[5] Holger Ludvigsen. "Real-Time GPU-Based 3D Ultrasound Reconstruction and Visualization". 2012. Lambert Academic Publishing

[6] Schneider RJ1, Perrin DP, Vasilyev NV, Marx GR, Del Nido PJ, Howe RD. "Real-time image-based rigid registration of three-dimensional ultrasound" Med Image Anal. 2012 Feb;16(2):402-14.

[7] Alberto Gomez "Full 3D Blood Velocity Mapping and Flow Quantification from Doppler Echocardiographic Images" 2013 URL: https://kclpure.kcl.ac.uk/portal/en/theses/full-3d-blood-velocity-mapping-and-flow-quantification-from-doppler-echocardiographic-images(f6a945d7-3a8e-4458-91c8-377ce3202119).html

[8] A.P. James and B. V. Dasarathy. "Medical Image Fusion: A survey of the state of the art" 2013. DOI: 10.1016/j.inffus.2013.12.002

[9] S. T. Elliott. "A User Guide to Compound Imaging". 2005. DOI: 10.1179/174313405X40215

[10] Susan Perry. "Are 'grimacing' babies in ultrasound images really reacting to mom's smoking?" 2015. URL:https://www.minnpost.com/second-opinion/2015/03/are-grimacing-babies-ultrasound-images-really-reacting-moms-smoking

[11] Gu S, Wilson D, Tan J, Pu J. "Pulmonary nodule registration: rigid or nonrigid?" . Med Phys. 2011 Jul;38(7):4406-14

[12] Daniel Rueckert and Paul Aljabar "Nonrigid Registration of Medical Images: Theory, Methods, and Applications". 2010. Signal Processing Magazine, IEEE (Volume:27 , Issue: 4)

[13] Kutter, O., Wein, W., Navab, N., "Multi-modal registration based ultrasound mosaicing". 2009. Medical Image Computing and Computer-Assisted Intervention–MICCAI 2009 763–770.

[14] Ni, D., Qu, Y., Yang, X., Chui, Y., Wong, T., Ho, S., Heng, P., "Volumetric ultrasound panorama based on 3D SIFT. Medical Image Computing and Computer-Assisted Intervention – MICCAI" 2008. 52–60

[15] Registration Techniques, ITK documentation, (visited on Jun 11 2016), URL: http://www.itk.org/Doxygen/html/RegistrationPage.html

[16] S. Schuon, C. Theobalt, J. Davis, and S. Thrun, "LidarBoost: Depth Superresolution for ToF 3D Shape Scanning", In Proceedings of IEEE CVPR 2009

[17] Hans J. Johnson, Matthew M. McCormick, Luis Ibanez, and the Insight Software Consortium"The ITK Software Guide Book 1: Introduction and Development Guidelines Fourth Edition Updated for ITK version 4.8", URL: http://www.itk.org/ItkSoftwareGuide.pdf

[18] "Scale-invariant feature transform" Wikipedia: The Free Encyclopedia. Wikimedia Foundation, 12 June 2016. Web. (visited on 13 June 2016). URL:https://en.wikipedia.org/wiki/Scale-invariant_feature_transform#SIFT

[19] Roberto Battiti. "Accelerated Backpropagation Learning: Two Optimization Methods", Complex Systems 3 (1989) p331-342

[20] Sebastien Ourselin, A Roche, G Subsol, Xavier Pennec, and Nicholas Ayache. "Reconstructing a 3d structure from serial histological sections". Image and Vision Computing, 19(1-2):25–31, 2001.

[21] Sébastien Ourselin, Radu Stefanescu, Xavier Pennec. "Robust registration of multimodal images: Towards real-time clinical applications", Medical Image Computing and Computer-Assisted Intervention — MICCAI 2002 Volume 2489 of the series Lecture Notes in Computer Science pp 140-147

[22] D. Rueckert, L.I. Sonoda, C. Hayes, D.L.G. Hill, M.O. Leach, and D.J. Hawkes. "Nonrigid Registration Using Free-Form Deformations: Application to Breast MR Images". IEEE Trans. Med. Imag., 18:712–721, 1999.

[23] Sidharth N. Kashyap. "Performance Engineering: Part 1 – Introduction". https://sidkashyap.files.wordpress.com/2013/05/cuda_memory_classes.png

[24] Jeyarajan Thiyagalingam, Olav Beckmann, Paul H. J. Kelly. "An exhaustive evaluation of row-major, column-major and Morton layouts for large two-dimensional arrays"

[25] "Euler's theorem". Wikipedia: The Free Encyclopedia. Wikimedia Foundation, 22 May 2016. Web. (visited on 13 June 2016). URL:https://en.wikipedia.org/wiki/Scale-invariant_feature_transform#SIFThttps://en.wikipedia.org/wiki/Euler%27s_rotation_theorem

[26] Parviz E. Nikravesh. "Computer-Aided Analysis of Mechanical Systems" http://www.u.arizona.edu/~pen/ame553/Notes/Lesson%2008-B.pdf