

UNIVERSITY OF CALGARY

Accelerated Medical Image Registration using the Graphics Processing Unit

by

Daniel Henrik Adler

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JANUARY, 2011

© Daniel Henrik Adler 2011

Abstract

Registration of three-dimensional images is an important task in biomedical science. However, computational costs of 3D registration algorithms have hindered their widespread use in many clinical and research workflows. We describe an automated medical image registration framework, including a novel implementation of the mutual information similarity metric, that executes entirely on the commodity graphics processing unit (GPU). Our methods take advantage of the graphics hardware's high computational parallelism and memory bandwidth to perform affine, intensity-based registration of multi-modal 3D medical images at near interactive rates. We also accelerate the Demons algorithm for deformable registration on the GPU. Registration results generated using our GPU-based methods are equivalent to those generated by conventional software-based methods, but with an order of magnitude reduction in computation time.

Acknowledgements

I thank my lab mates, supervisor, and close friends and family for their support throughout my graduate studies. In particular, I thank Sonny Chan and Eric Penner for their mentorship and innovative ideas in image registration and computer graphics.

Table of Contents

Abstract	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
1 Introduction	1
1.1 Motivation	1
1.2 Background on Image Registration	1
1.2.1 Extrinsic vs. Intrinsic Methods	2
1.2.2 Parametrizable vs. Non-Parametrizable Transformations	2
1.2.3 Linear vs. Non-Linear Transformations	3
1.2.4 Uni-Modality vs. Multi-Modality Similarity Metrics	4
1.2.5 Intra-Subject vs. Inter-Subject Registration	5
1.2.6 Dimensionality	6
1.3 Methodology	6
1.4 Contributions	8
1.5 Overview of Thesis	8
2 Graphics Hardware for General Purpose Computation	10
2.1 Modern Graphics Hardware	10
2.2 Graphics Rendering Pipeline	12
2.3 Programmable Graphics Processors	13
2.3.1 GPGPU Shader Example	14
2.4 Unified Shader Architecture	16
3 Background on Accelerated Image Registration	21
3.1 Introduction	21
3.2 Accelerated Image Registration	21
3.2.1 Registration on Parallel Hardware	22
3.2.2 Registration on the GPU	24
3.3 Registration using Mutual Information	26
3.3.1 Image Histograms	28
3.3.2 Mutual Information on the GPU	29
4 GPU-Accelerated Image Registration Methods	34
4.1 Accelerated Affine Image Registration	34
4.1.1 Volume Rendering	34
4.1.2 Affine Image Transformation	36
4.1.3 Image Interpolation	37
4.1.4 Difference- and Correlation-Based Similarity Metrics	38
4.1.5 Mutual Information Similarity Metric	44
4.1.6 Metric Function Optimization	50
4.1.7 Hierarchical Search	51
4.2 GPU-Accelerated Deformable Image Registration	52
4.2.1 Optical Flow	52

4.2.2	Demons Update Iteration	53
4.2.3	Non-linear Image Transformation	54
4.2.4	Deformation Update	54
5	Validation of GPU-Accelerated Medical Image Registration	57
5.1	Experimental Methods	58
5.1.1	Artificial Affine Transformations	58
5.1.2	Artificial Non-linear Deformations	60
5.1.3	Retrospective Image Registration Evaluation	62
5.1.4	Experimental Equipment	66
5.2	Results	66
5.2.1	Affine Registration Iteration Speed	66
5.2.2	Artificially Transformed Images	67
5.2.3	Retrospective Image Registration Evaluation	71
5.3	Discussion	73
5.3.1	Speed and Accuracy	73
5.3.2	Normalized Mutual Information Metric	74
5.3.3	Real-Time Visualization	75
6	Tumor Spatial Distribution Analysis using Image Registration	78
6.1	Preface	78
6.2	Introduction	79
6.3	Methods	80
6.3.1	Patient Selection	80
6.3.2	DNA Samples	80
6.3.3	Image Processing	80
6.3.4	Tumor Distribution	81
6.3.5	Tumor Volume	82
6.3.6	Tumor Centroids	83
6.3.7	Case Selection and Imaging Parameters	83
6.4	Results	83
6.4.1	Tumor Distribution	83
6.4.2	Tumor Volume	85
6.4.3	Tumor Position	88
6.5	Discussion	89
6.5.1	Registration to Normalized Space	89
6.5.2	Tumor Location	89
6.5.3	Relevance of the Analysis	90
7	Conclusion	91
7.1	Limitations and Future Work	91
7.1.1	Alternative Multi-Modality Similarity Metrics	91
7.1.2	Partial Volume Interpolation	91
7.1.3	Parzen Windowing	92
7.1.4	Registration using Raycasting	92
7.1.5	Symmetric and Diffeomorphic Transformations	93
7.2	Concluding Remarks	94
A	Graphics Rendering Pipeline	96

A.1	Geometry Specification	97
A.2	Vertex Transformation and Lighting	98
A.3	Fragment Operations and Texturing	100
A.4	Frame Buffer Rendering	101
B	Shader Programming	103
B.1	Vertex Shaders	103
B.2	Fragment Shaders	104
B.3	Custom Graphics Shader Examples	105
	Bibliography	108

List of Tables

5.1	Size and spacing of images in the RIRE database	63
5.2	Mean run times for the transform-resample-metric cycle on the GPU and CPU as a function of image size and similarity metric	67
5.3	Errors and run times on the GPU and CPU for nine-parameter, affine registration of the MNI data	68
5.4	RMS non-linear registration errors before and after GPU Demons registration of the deformed MNI data	68
5.5	Registration errors and timing for GPU alignment of PET to MR images using the NCC metric	71
5.6	Registration errors and timing for GPU alignment of CT to MR images using the NGF metric	71
5.7	Registration errors and timing for GPU alignment of PET and CT to MR images using the NMI metric	72
6.1	Number of GBM tumors occupying defined anatomical regions [n (%)]	85
6.2	Number of GBM tumors occupying defined brain sectors [n (%)] . . .	86
6.3	Number of GBM tumors occupying defined brain sectors [n (%)] . . .	87

List of Figures

1.1	Illustration of affine and deformable registration: fixed image (a); moving image before (b) and after affine (c) and deformable (d) registration to the fixed image	4
1.2	T1-weighted MRI of an elderly patient with Alzheimer’s disease: baseline scan (a), two year follow-up scan affinely registered to baseline (b), Jacobian of non-linear deformation between baseline and follow-up (c): Red/blue correspond to $\pm 10\%$ volume expansion/loss over the two year period	5
1.3	Segmentations of cortex and deep gray matter in an atlas of elderly subjects (a); segmentations transferred to an individual subject after affine and non-linear transformation (b)	6
1.4	Core components of the iterative registration cycle, with shaded components executed on the graphics processing unit (GPU) in our framework	7
2.1	Comparison of GPU and CPU performance trends over most of the last decade	11
2.2	Schematic diagram of the GPU pipeline for graphics rendering	12
2.3	Example of parallel computation mapped to fragment processors in the GPU rendering pipeline	15
2.4	Schematic of memory gather and scatter operations	16
2.5	Memory processing hierarchy of thread blocks within CUDA	18
2.6	Block diagram of the NVIDIA Fermi architecture’s streaming multiprocessor containing 32 CUDA cores	19
3.1	Joint histograms of a T1-weighted image with itself for varying degrees of misalignment by pure axial rotation (shown natural logarithmic scale)	27
3.2	Pseudocode to compute the joint histogram of two images by scattering intensity values	29
3.3	Pseudocode to compute the histogram of an image by gathering intensity values	30
3.4	Summation of partial histograms (computed in parallel) into a global histogram using either atomic or sequential operations	31
4.1	Volume rendering of a CT head dataset (inferior view) using texture mapping and view-aligned proxy geometry	35
4.2	Mapping the fixed and moving images to quads using texture coordinates	36
4.3	Interpolation of the moving image during transformation	37
4.4	T1-weighted image (a), T2-weighted image (b); subtraction (c) of the T1 image and a rigidly transformed version of itself	39
4.5	Gradient images of the T1 (a) and T2 (b) MRIs	40
4.6	Computation of difference- and correlation-based metrics using the rendering pipeline	41

4.7	Parallel reduction to accumulate the final metric value by shader down-sampling passes	44
4.8	Computation of 1D image histograms on the using vertex scattering in the rendering pipeline	46
4.9	Partial volume interpolation weights (2D example)	49
4.10	Recursive Gaussian blurring and downsampling scheme to generate image pyramids	52
4.11	Non-linear transformation using coordinate look-up in a displacement field texture on the GPU	55
4.12	Ping-pong iterative updates of the Demons displacement fields by swapping source and render target textures	55
4.13	Applying a Gaussian blur using the separability of the convolution kernel	56
5.1	Slices of simulated T1- (a,b) and T2-weighted (c,d) MNI images . . .	59
5.2	Slice of original T1-weighted MNI image before (a) and after linear transformations with small (b) and large (c) magnitude 3D translation, rotation, and scaling applied	59
5.3	Slice and corresponding displacement field of original MNI image (a) and after local warping with sine waves (b), global warping with a pinch and a bulge (c), and combined local/global warping (d)	62
5.4	Sample unregistered PET (a) and CT (b) images of a patient in Group A of the RIRE database	64
5.5	Sample MR images of a patient in Group A of the RIRE database: proton-density (a), T1-weighted (b), and T2-weighted (c)	64
5.6	Sample PET, CT, and MR images from the RIRE study before (a, b, c) and after (d, e, f) removal of the fiducial markers (circled) and stereotactic frame	65
5.7	Recovered components of local displacement field using GPU Demons	69
5.8	Recovered magnitude (a) and Jacobian (b) of global displacement restorative field using GPU Demons	69
5.9	Renderings of recovered local displacement field (a) and RMS error (b), and recovered global displacement field (c) and RMS error (d) . .	70
5.10	Overlaid PET and T2-weighted MR image slices and their joint histograms (PET vs. T2) before (a, c) and after (b, d) rigid-body alignment	72
5.11	Overlaid CT and T1-weighted MR image slices and their joint histograms (CT vs. T1) before (a, c) and after (b, d) rigid-body alignment	72
5.12	Screenshots of our GPU accelerated image registration tool showing MR images of the same subject at two time points before and after alignment in 2D (a, b) and rendered in 3D (c, d)]	76
6.1	Segmentation and registration workflow demonstrated on GBM patient image	81
6.2	MNI atlas segmented into gross neuroanatomical regions shown in 3D rendering (a), axial slice (b), and coronal slice (c)	82

6.3	Workflow to create tumor volume map for four example GBM patient images	82
6.4	Sample segmentation of GBM tumor from our study	84
6.5	Occupancy maps depicting number of overlapping tumors after registration to atlas space	84
6.6	Axial, coronal, and sagittal projections of volume density maps (in units of 1 mm^3 per 1 mm^2 area) in atlas space for tumors with (36 cases, cyan) and without (36 cases, magenta) <i>MGMT</i> promoter methylation status	86
6.7	Interquartile plot of tumor volumes ($P = 0.55$)	87
6.8	Plots of tumor centroids in atlas space for tumors with (36 cases, blue) and without (36 cases, red) <i>MGMT</i> promoter methylation status	88
6.9	Axial, coronal, and sagittal plots of tumor centroids in atlas space for tumors with (36 cases, blue) and without (36 cases, red) <i>MGMT</i> promoter methylation status	88
7.1	Two-dimensional schematic of using volume raycasting to sample the fixed and moving images in registration	93
A.1	Schematic of the GPU pipeline, with programmable elements shown shaded	96
A.2	The vertex processing stage of the graphics pipeline applies transformation and lighting to vertices	98
A.3	Rasterization of geometric primitives into screen fragments	99
A.4	The fragment processing stage of the graphics pipeline applies colours to fragments	100
A.5	Mapping of textures onto polygons is defined by texture coordinates at the polygon vertices	101
B.1	Vectors used in the Phong lighting model	105

Chapter 1

Introduction

1.1 Motivation

Medical images acquired at different times, using different modalities, and of different subjects contain complementary information that often needs to be integrated in order to guide clinical procedures and analysis. This is accomplished with image registration: *the task of transforming two images into a common anatomical coordinate space*. Registration of images is extremely useful in medical science, where it is often required for clinical applications, such as to guide stereotactic surgery [1], and as a preprocessing step to answer research problems, such as finding group differences between imaged subjects [2]. Among many other applications, registration is used to characterize anatomical variability, to detect changes in disease state over time, and to map functional information into anatomical space [3].

Medical image registration has been extensively researched by groups around the world, each contributing their expertise to the field. This is partly evidenced by the over eight thousand hits returned on Google Scholar for the search terms “image” and “registration” in article titles. As we shall see in this thesis, medical image registration is quite literally a field that fuses knowledge from many disciplines, including physics, computer vision, numerical methods, and computer graphics.

Fast and accurate automated, intensity-based medical image registration is of great utility to clinicians and researchers [3]. However, the high computational demand of registration can lead to prohibitively lengthy execution times in imaging workflows. For instance, registration must be performed within minutes for applications in intra-operative imaging and image-guided surgery, so as not to delay procedures [1, 4]. Also, brain atlas creation [5] and clinical studies involving large cohorts of subjects [6] often require the accurate and reliable registration of hundreds or thousands of image pairs.

To date, the enormous computational requirements of registration methods have largely precluded their use at interactive or near real-time speeds on desktop computers. In order to become an accepted tool in day-to-day practice, our registration algorithms must execute on widely available hardware and generate accurate results within the acceptable timeframe of minutes or less.

1.2 Background on Image Registration

In order to familiarize the reader with the field of image registration, we briefly discuss common classifications of registration methods, highlighting aspects pertinent to our work.

1.2.1 Extrinsic vs. Intrinsic Methods

Registration methods can be broadly categorized in terms of the image features used to evaluate correspondence and to drive matching. *Extrinsic* methods rely on landmarks of artificial objects used specifically for registration purposes. Examples include fiducial markers and stereotactic frames [7]. These features can be used to align images between multiple imaging sessions if they remain fixed on the subject and are visible in all modalities. Registration based on matching of extrinsic markers is relatively straightforward, since the solution can usually be computed explicitly. A significant disadvantage of these methods is their prospective nature: fiducial markers must be introduced prior to registration. The methods may also be invasive. A stereotactic frame, for instance, must be rigidly screwed to the outer skull table. However, extrinsic methods remain the gold standard for rigid-body registration [8].

Intrinsic methods, such as ours, are driven by image features from the subject's anatomy alone. These features may be derived from the images in a number of ways. They can be salient anatomical landmarks, such as the anterior and posterior commissures, which are used to initially align subjects with the Talairach brain atlas [8]. Landmarks are particularly versatile if readily identifiable, since they can be used to match images from varying subjects in arbitrary poses and across modalities. Features can also consist of sets of corresponding structures identified through segmentation or geometrical features, like extracted surfaces and lines.

Once features have been identified or extracted, intrinsic methods are generally very fast, since they operate on reduced sets of image information. However, because feature extraction usually requires user interaction or substantial preprocessing (e.g. identification of landmarks or segmentation), it is generally not well suited for fully automatic registration. One must also keep in mind that registration accuracy is limited by the accuracy of the feature extraction protocol.

Alternatively, the intrinsic image intensities themselves can drive registration without any prior feature extraction or segmentation. *Our work in this thesis is on intensity-based registration methods that operate on pixel values only.* Major advantages of intrinsic over extrinsic and feature-based methods is that they are non-invasive, they can be used to register images retrospectively, and they do not require user interaction to identify landmarks or to supervise segmentation. In addition, they permit the evaluation of non-linear transformations between images, which can be used to match anatomy of different individuals. Intrinsic methods tend to be relatively computationally intensive, however, since they operate on the full image content throughout the registration process.

1.2.2 Parametrizable vs. Non-Parametrizable Transformations

In our work, *registration is cast as an iterative process in which a “moving” image is transformed to match a “fixed” image.* The nature of permissible transformation functions applied to the moving image distinguishes two major forms of intensity-based registration. If transformations are *parametrizable*, such as for our affine registration work, then registration is cast as an optimization problem over a predetermined parameter space. The optimization cost function is a metric of similarity between

the images that quantifies their degree of registration. This metric is improved by a numerical optimizer that iteratively steps through the parameter space.

Optimization is effectively an intelligent search through transformation parameter space for the best possible match, as quantified by the metric. Due to the potentially high dimension and large extent of the space, this search must be performed in an efficient, non-heuristic manner. To this end, many algorithms have been described that are suited to different numerical problems [9]. They can be broadly categorized as local or global. Local methods find an optimum within a neighbouring vicinity of their initialization point. Global methods search for the optimum over a given range of parameters. These are more robust with respect to initialization, but at the cost of slower convergence.

Another form of intensity-based registration uses *non-parametric* transformations based on free-form deformations. These transformations are generally expressed as vector fields over image space. Matching proceeds by iteratively updating the field using forces derived from local, pixel-wise measures of similarity.

1.2.3 Linear vs. Non-Linear Transformations

The particular class of transformations used depends on the nature of the registration task. It is generally chosen to model the geometric distortions that we would like to recover between the images. Transformations can be classed as *linear* and *non-linear*, depending on whether they preserve straight lines. In three dimensions (3D), linear transformations are parametrized by 4-by-4 matrices acting on vectors of homogeneous coordinates:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} & \mathbf{R} & & \mathbf{t} \\ & & & \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}, \quad (1.1)$$

where $\mathbf{t}$ is a translation vector and $\mathbf{R}$ is a 3-by-3 matrix that accounts for rotations, scaling, and shearing. For rigid transformations, $\mathbf{R}$ is a rotation matrix.

Rigid-body linear transformations are constrained to translation and rotation only (six parameters), whereas affine transformations can recover translation, rotation, scaling, and shearing (twelve parameters). Affine transformations have the property of preserving the collinearity of lines. Following affine registration between images, the remaining mismatches are due to local shape differences. If necessary for the application, these are recovered using higher order non-linear (deformable) transformations (see Fig. 1.1).

Deformable transformations can be categorized as parametric and non-parametric. It is common to model the displacement field parametrically with piecewise polynomials defined over a grid of control points [10]. Spline functions are favoured, since the movement of one control point only affects a limited region of the overall deformation. This property of local support enhances their ability to model complex deformations in images [11].

In non-parametric deformable methods, the transformation is represented by a

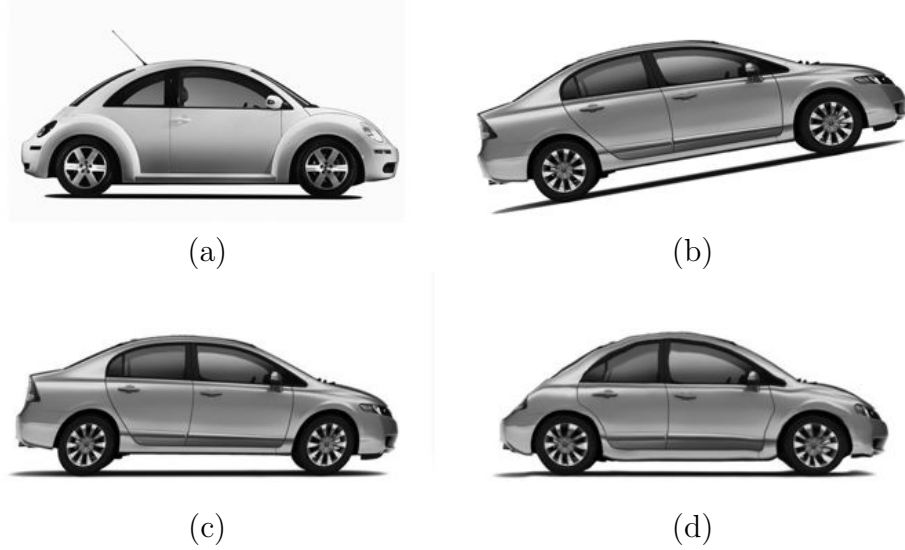


Figure 1.1: Illustration of affine and deformable registration: fixed image (a); moving image before (b) and after affine (c) and deformable (d) registration to the fixed image

dense vector field over the image samples. Each vector can move independently, though the field is generally regularized by some smoothing criteria. These methods tend to be based on physical models, whereby the image is subjected to forces that drive deformation [12]. For example, elastic models treat the image as an elastic body with external forces applied to guide matching and internal forces applied to impose smoothness on the overall transformation. Internal forces generally increase proportional to deformation magnitude, thereby penalizing extensive distortions. This may or may not be desirable, depending on the nature of the registration problem. An alternative formulation of deformable registration models the image as a viscous fluid, in which the internal stresses relax over time. This permits more extreme focal warping and the growth of new regions.

We implement two distinct registration methods. The first uses 12-parameter affine (linear) transformations; the second, called the *Demons* method, uses a vector field to store non-linear, non-parametric transformations.

1.2.4 Uni-Modality vs. Multi-Modality Similarity Metrics

The similarity metric function is used to drive registration. The function should achieve an optimum (either minimum or maximum) when the images are in perfect spatial correspondence, varying smoothly over the transformation parameter domain. The choice of metric depends on the intensity distributions of the input images. *Uni-modality* metrics are fit for registering images of the same modality. They can usually be evaluated by independent computations at each spatial coordinate.

So called *multi-modality* metrics are suitable for registering images that have different intensity characteristics. They are usually based on determining statistical or functional dependancies between images. Even if acquired using the same modality, intensity differences may arise due to varying acquisition parameters. This is particu-

larly true for MRIs acquired using slightly different sequences at different institutions. Multi-modality metrics are generally based on establishing either functional or statistical dependences between images. The former class comprises metrics that measure correlation of intensities and gradients. Statistical dependence is usually quantified by measures of information entropy [13].

For affine registration, we implement both uni- and multi-modality similarity metrics. Our implementation of non-linear transformation using the Demons method only registers images of the same modality.

1.2.5 Intra-Subject vs. Inter-Subject Registration

It is also helpful to categorize registration as being between images of the same subject (*intra-subject*) or of different subjects (*inter-subject*). In intra-subject registration, images that were acquired at different times or with different modalities are brought into correspondence. If the subject anatomy has not changed between scans, then registration consists of rigid-body alignment. Common instances of intra-subject registration include the alignment of baseline and follow-up scans for assessment of disease progression (see Fig. 1.2), the alignment of pre- and post-contrast images, and the correction of patient movement in serial functional studies. It is also commonly used to fuse data acquired on different machines.

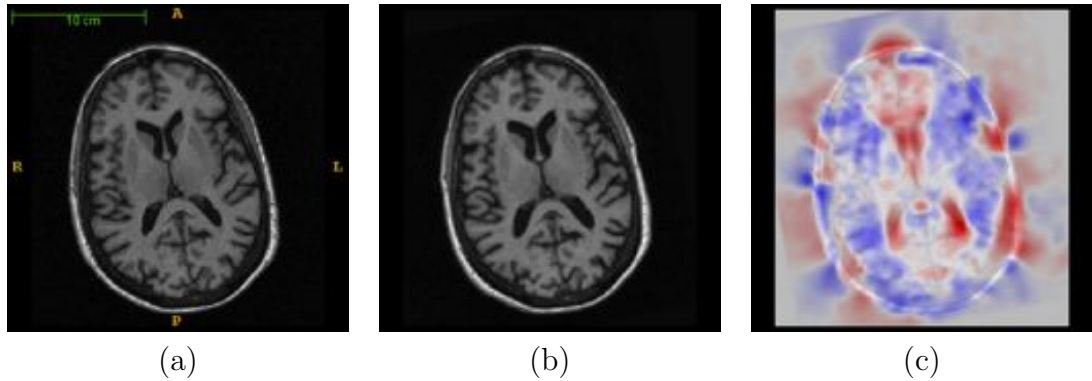


Figure 1.2: T1-weighted MRI of an elderly patient with Alzheimer’s disease: baseline scan (a), two year follow-up scan affinely registered to baseline (b), Jacobian of non-linear deformation between baseline and follow-up (c): Red/blue correspond to $\pm 10\%$ volume expansion/loss over the two year period

Inter-subject registration is used to combine scans of different subjects. An important application is the study of structural or functional differences between populations. Such studies are usually conducted in a normalized anatomical *atlas* space, permitting meaningful comparison between subjects across populations. Normalization is useful for comparing differences among subjects, since it removes individual anatomical shape variations. Atlas-based segmentation is essentially the reverse procedure. First, structures of interest are segmented and labeled in an atlas image. Next, the atlas and input subject images are registered, thereby transferring the atlas segmentation onto the subject (see Fig. 1.3).

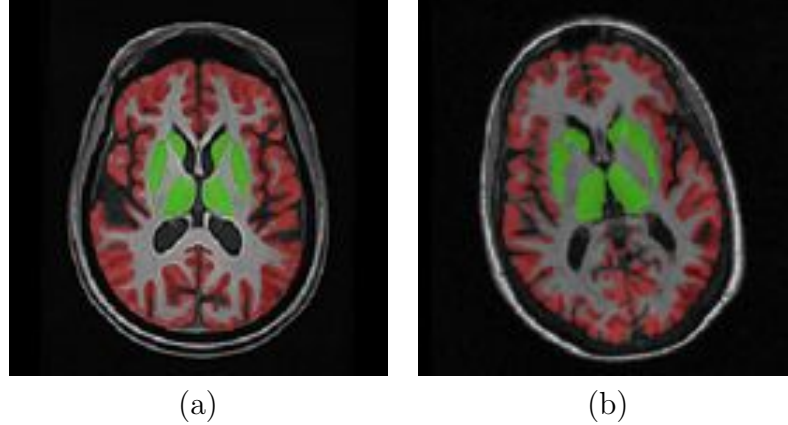


Figure 1.3: Segmentations of cortex and deep gray matter in an atlas of elderly subjects (a); segmentations transferred to an individual subject after affine and non-linear transformation (b)

Even though the images generally contain the same anatomic structures, rigid-body alignment is usually not sufficient. This is because the shape and relative locations of structures can vary between subjects. Registering different subjects is usually a two step process. One initially captures global scale, rotation, and translation differences using a nine-parameter affine transformation. This is followed by some form of *deformable registration* in order to capture more local differences.

1.2.6 Dimensionality

We treat the fixed and moving images as 3D volumes, since this is the most commonly encountered instance of the registration problem in medicine. Many of our methods can be applied to cases of lower or higher dimensionality, though we do not explicitly discuss their implementations. Registration of 2D images is sometimes used in reconstructing volumes from serially acquired tomographic data, such as histology slices. The problem of establishing correspondence between 2D and 3D images commonly arises in image guided surgery and radiotherapy treatment [14]. These procedures integrate both projection x-ray and volumetric CT or MR images. Higher dimensionality problems may arise when registering 4D time series or 6D diffusion tensor image datasets, for example.

1.3 Methodology

In this thesis, we focus on three-dimensional *automated, intensity-based* registration, which is an iterative procedure that transforms a *moving image* onto a stationary, *fixed image* by maximizing a measure of similarity between the images. This form of registration is cast as an optimization problem that requires minimal user interaction, with the goal of providing reliable and repeatable results [15].

Each iteration of automated, intensity-based registration involves four steps, as shown in Figure 1.4. First, a parametrized transformation is applied to the spatial

coordinates of the moving image (1). The transformed moving image values are obtained following resampling of the moving image into the coordinate space of the fixed image (2). Next, a similarity metric is computed to quantify correspondence between the fixed and transformed images (3). The metric is passed to the optimizer, which modifies the transform parameters to ultimately improve registration (4). The specifications and constraints of the registration problem at hand dictate the particular methods chosen to implement these four generic components.

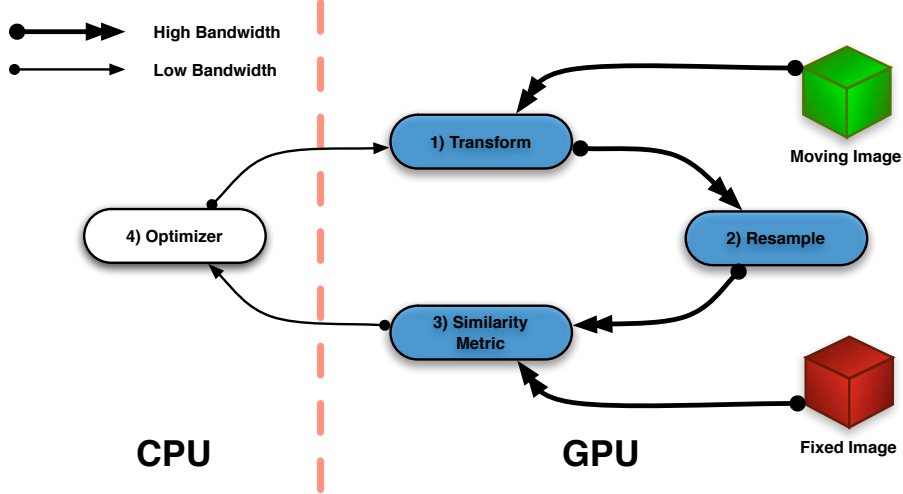


Figure 1.4: Core components of the iterative registration cycle, with shaded components executed on the graphics processing unit (GPU) in our framework

We emphasize that for intensity-based registration, only intrinsic image features are used to drive registration, without the need for externally placed landmarks or fiducials. The similarity measures that we optimize are derived directly from image intensity values, whereas geometric methods use shape matching to drive registration. The flexibility of intensity-based registration over geometric methods is well recognized in the literature [8].

Automated registration of 3D medical images is computationally intensive for two main reasons [10]. First, medical images may be very large. Standard magnetic resonance (MR) and computed tomography (CT) imaging studies can have voxel resolutions of a millimeter or less. With hundreds of samples per direction, they may contain millions of voxels, each of which must be processed on every iteration of the registration cycle. Second, automated image registration is a non-linear, multi-dimensional optimization problem. The dimension and complexity of the parameter space being searched partly dictate the number of iterations required for convergence to an optimum. Rigid-body transformations are parametrized by six values, while non-linear transformations based on control point lattices may have objective spaces with thousands of dimensions [16].

1.4 Contributions

To bring down computation times, a number of groups have implemented medical image registration on supercomputers and other high performance computing environments. Significant disadvantages of these approaches are certainly high costs and limited accessibility to the necessary machines in the clinical environment.

Recent advances in the speed, data capacity, and programmability of graphics hardware have enabled its use in the acceleration of image registration. We present a novel framework that leverages recent advances in the power and flexibility of the commodity graphics processing unit (GPU) to accelerate affine and deformable registration of 3D medical images.

Over the past decade, the demands of the video gaming and entertainment industries have driven tremendous advancement in GPU technology [17]. Modern GPUs are massively multi-threaded devices optimized for processing large volumes of data with high throughput. Compared to traditional central processing unit (CPU) architecture, the GPU’s data-caching and flow control logic are minimized in order to make room for more arithmetic logic units.

Our framework places all computationally intensive components of the registration cycle on commodity desktop graphics hardware. Also, all images remain on the GPU in our framework. This eliminates bandwidth-intensive copies of data between the GPU and the host application’s main memory on the CPU.

We accelerate several of the most commonly used uni- and multi-modality similarity metrics in our framework, including normalized mutual information [18, 19]. Normalized mutual information, which is derived from the field of information theory, is widely regarded as the most reliable and accurate intensity-based metric for registering images acquired using different modalities [13].

In addition to the parametric framework that we have outlined, we also accelerate the well-established Demons method for non-linear (deformable) image registration [20]. Demons is a non-parametric method that uses optical flow to generate a free-form deformation field to match the images [21]. Unlike parameterized affine registration, Demons does not explicitly optimize a similarity metric.

1.5 Overview of Thesis

In order for registration to become widely used in clinical practice, it must execute in a clinically compatible timeframe on ubiquitous hardware [22]. In this thesis, we present methods for significantly accelerating intensity-based, automatic registration of 3D medical images using commodity graphics processing units. This work is made possible by recent advancements that allow us to replace the GPU’s once fixed functionality graphics pipeline with custom code that executes general purpose computations.

In Chapter 2, we provide an overview of the GPU and advancements in its programmability. In Chapter 3, we give background on accelerated image registration and conduct a literature review of related work. In Chapter 4, we describe our implementations of affine and deformable registration methods on the GPU. In Chapter 5,

we thoroughly evaluate the accuracy and performance of our framework, proving its clinical viability for registering multi-modal, 3D images. A clinical application of registration to brain tumor mapping is presented in Chapter 6. We close the thesis in Chapter 7 with limitations of our work and recommendations for future improvements. Supplementary information is provided in the appendices. Appendix A discusses the graphics pipeline, which forms the basis of our framework, and Appendix B discusses shader programming, which we use to develop our methods.

Chapter 2

Graphics Hardware for General Purpose Computation

The graphics processing unit offers the highest ratio of computational power to cost of any hardware [23]. GPUs were initially designed with fixed-function rendering pipelines, intended solely for graphics applications. Their emergence as standard hardware in most computer systems by the year 2000 was driven largely by demand for interactive 3D graphics in consumer applications [24]. However, research and commercial developers have expended considerable effort to harness the GPU's parallelism and power for both enhanced graphics realism [25] and for general-purpose computational tasks [23]. Indeed, the GPU's architecture is very different from that of other single-chip processors and deserves particular consideration. The emerging field of applying the GPU to accelerate non-graphics computations is known as general-purpose GPU computing, or simply *GPGPU*.

In this chapter, we provide an overview of graphics hardware, the rendering pipeline, and discuss how they can be adapted to general-purpose tasks. This material serves as background to Chapters 3 and 4, in which we describe GPGPU methods for image registration.

2.1 Modern Graphics Hardware

Graphics hardware was originally designed for running real-time graphics applications that process large amounts of data. However, this hardware has evolved from being a devoted graphics processor to being a programmable, multipurpose engine. General-purpose applications can now achieve tremendous speedup on graphics hardware, so long as they are successfully mapped to its architecture. In particular, applications are well-suited for the GPU if they have large computational requirements, exhibit a high degree of parallelism, and demand high throughput over low latency [26]. These characteristics stem from the requirements of real-time rendering applications, which may output billions of pixels per second, each undergoing hundreds of processing operations. And since these operations are independent of each other, many pixels can be processed in parallel.

The latency of pixel processing operations is not critical, since graphics applications can generally tolerate relatively large time delays between command initiation and perceived effects. Indeed, each pixel operation is performed within nanoseconds, while the human visual system only perceives changes at the millisecond scale [26]. On the other hand, the pixel throughput per second in computer graphics applications is less subject to compromise.

Modern GPU architecture provides tremendous raw computational power and memory bandwidth compared to the CPU, and the gap is steadily widening [27]. The hardware's engine consists of an array of processors that execute the same set of operations on parallel streams of data. This computational model is referred to

as *single-instruction, multiple-data* (SIMD). In computer graphics applications, the streams consist of vertices and pixels. Additional performance gain is achieved on the GPU by efficient hardware implementations of many numerical operations [24]. Figure 2.1 (from Phillips, et al. [17]) shows plots of GPU and CPU performance trends over most of the last decade [17]. In terms of computing speed alone, GPU performance doubles roughly every six months, while CPU performance doubles roughly every 18 months. As a result, GPUs today offer processing power an order of magnitude greater than CPUs for about the same cost.

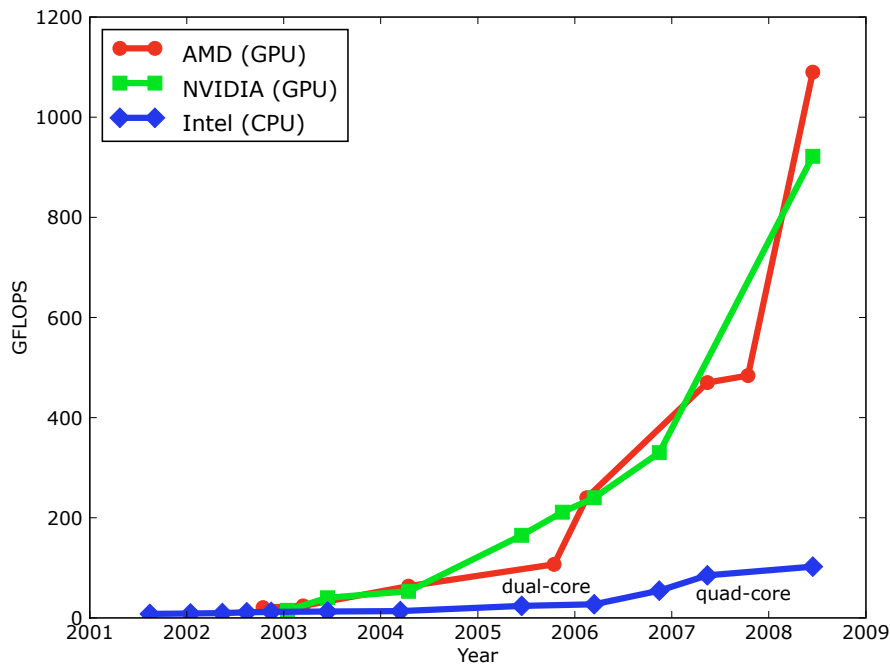


Figure 2.1: Comparison of GPU and CPU performance trends over most of the last decade

For example, the NVIDIA GeForce GTX 480 consumer graphics card¹ (released in March 2010) has 480 processor cores, each with a clock speed of 1.4 GHz. It achieves a sustained computational power of 1345 GFLOPS (billions of floating-point operations per second) at single precision and 673 GFLOPS at double precision. The GPU can process 33.6 billion pixels per second, as measured by the benchmarking tool 3DMark Vantage². In addition, it has 1.5 GB of dynamic random access memory (DRAM), with internal transfer bandwidth of up to 177.4 GB/second. The GeForce GTX 480 currently costs under \$500 US.

The preceding values far exceeds the peak raw computational performance of the

¹The latest NVIDIA GeForce graphics card product information can be downloaded at http://www.nvidia.com/object/geforce_family.html

²3DMark Vantage can be downloaded at <http://www.futuremark.com/3dmarkvantage/>

high-end Intel Core i7-930 processor³, with four cores clocked at 2.8 GHz. The Core i7 CPU has a maximum memory bandwidth of 25.6 GB/second and achieves 66 GFLOP in double precision, as measured by the benchmarking tool SiSoftware Sandra⁴. In the latest hardware, data transfer between the GPU and CPU main memory occurs across the 16 lane PCI-Express 2.0 bus at rates of up to 8 GB/second. This is an order of magnitude slower than transfer rates internal to the GPU.

2.2 Graphics Rendering Pipeline

Graphics hardware was originally designed to render images using a fixed-functionality pipeline. Input to the graphics pipeline consists of vertices of geometric primitives (points, lines, and planar polygons) that define a 3D scene. It outputs a 2D image of the scene as coloured, rasterized pixels (called fragments) into a frame buffer in video memory. Figure 2.2 shows a brief schematic of the pipeline.

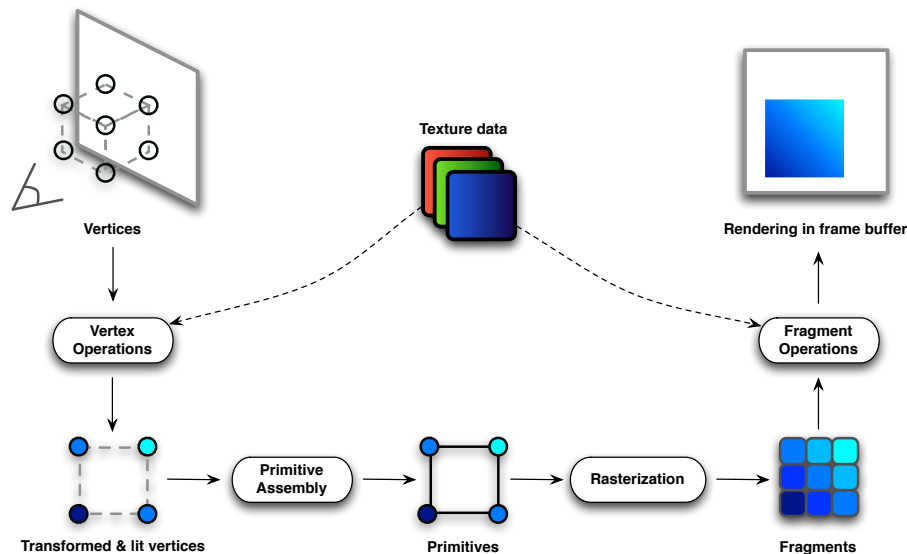


Figure 2.2: Schematic diagram of the GPU pipeline for graphics rendering

Two stages of the pipeline are especially relevant to our work: vertex and fragment processing. The *vertex processing* stage modifies attributes of the scene vertices, such as position and colour. Following this stage, all vertices are transformed into screen coordinates and lighting effects are applied to their colours. Next, the vertices are assembled into geometric primitives, which are subsequently rasterized into screen fragments. The *fragment processing* stage applies colours to the fragments, often using interpolated vertex attributes as input. Appendix A describes these pipeline stages in greater detail.

³The latest Intel Core i7 CPU product information can be downloaded at <http://www.intel.com/products/processor/corei7/>

⁴SiSoftware Sandra can be downloaded at <http://www.sisoftware.net/>

Prior to the current generation of GPUs, all vertices and fragments were sent through the pipeline using a fixed set of operations. Today, the vertex and fragment processing stages can be programmed to generate custom, advanced rendering effects, as demonstrated in the examples of Appendix B.

The introduction of programmable vertex and fragment processing stages in the graphics pipeline has also enabled graphics hardware to perform tasks other than traditional graphics rendering. General-purpose GPU computing harnesses the GPU as a highly parallel stream processor for which the shaders function as nodes in a cluster or supercomputer. The GPU is inherently well suited to accelerating image processing, since it is optimized for the storage, transfer, and parallel processing of vertices and pixels. However, it is not limited to this domain. Modern graphics hardware supports programming flow control, many scientific functions, and high precision floating-point data types that conform to IEEE standards [23].

Computations on the GPU are traditionally expressed as *kernels* that execute in parallel on multiple data streams, which are stored in video memory. The kernels are programmed using high-level languages, such as OpenGL Shading Language (GLSL) [28]. A considerable level of graphics programming expertise is required to design and debug general-purpose GPU algorithms implemented using the graphics pipeline. There exist several projects to address these difficulties and abstract the GPU as a general-purpose computation engine [29, 30]. Most significantly, the Compute Unified Device Architecture (CUDA) enables modern GPUs to be programmed as generic co-processors, rather than as graphics engines [31]. CUDA was introduced in 2007 with the NVIDIA GeForce 8800 series of GPUs.

2.3 Programmable Graphics Processors

Original graphics hardware was designed expressly to render using the fixed-functionality pipeline. The vertex and fragment processing stages could be configured using state variables, but they were not programmable [25]. For example, one could modify the positions and colours of lights, but not the equations used to model their interactions with the scene.

Graphics hardware has evolved to become much more flexible. NVIDIA introduced the first programmable GPU stage in 1999 [23]. They enabled register combiner operations in fragment processing, allowing limited custom combinations of textures and colours. NVIDIA and ATI, another major GPU manufacturer, both released hardware with the first fully programmable vertex and fragment stages in 2004 [29]. Indeed, the GPU can now be considered to be a collection of processors that execute the same kernel on parallel streams of data. The data elements are independent and cannot communicate with each other. In graphics, these basic elements are vertices and fragments.

We use OpenGL Shading Language [28] (GLSL) to implement the programmable graphics processing components of our registration application. This high-level language was approved as an extension to the OpenGL API in 2003. It enables the developer to create custom *shader programs* that override the fixed vertex and frag-

ment processing stages. The term *shader* is a reference to the ultimate task of the graphics pipeline: applying shading and colour to fragments. GLSL is based on the C programming language syntax and flow control, with the addition of various vector and matrix types that simplify graphics programming. Shader programs can be compiled and linked at application run-time into executable programs.

The ability to use high-level code on graphics hardware considerably simplifies GPU programming. However, shader programming still requires a thorough understanding of the features and limitations of graphics APIs and hardware. For instance, the programmer must handle the allocation and transfer of textures, the construction of graphics primitives, and the downloading of data from the GPU.

The intended purpose of custom vertex and fragment shader programs is to permit more advanced and varied rendering of scenes than was possible with the fixed pipeline. One major development, for instance, is the ability to perform lighting on a per-fragment basis, in contrast to the default per-vertex lighting implementation of the fixed pipeline, which we describe in Appendix A.2. Shader programming also enables the GPU to perform tasks that are out of the traditional scope of computer graphics. This is the crux of general-purpose GPU programming. The speed and ever increasing flexibility of shaders have led to tremendous development efforts in this field.

We note that alternatives to the shader programming environment have been available for some time. The Brook programming environment, for instance, was an early GPGPU project that abstracted the GPU as a streaming processor [29]. In Brook, kernels were programmed to execute on generic streams, without the graphics notions of vertices, fragments, and textures. Similarly, GPU abstractions of many common data structures have been provided to developers by the Glift template library [30].

2.3.1 GPGPU Shader Example

The GPU’s SIMD programming paradigm encourages the creation of applications with high ratios of arithmetic intensity to memory bandwidth. In this section, we give an example of performing a general-purpose computation using GLSL [32]. This program uses the GPU rendering pipeline (see Appendix A) in the same manner as any other graphics application, with computations expressed as a sequence of rendering operations on graphics primitives in a fragment shader (see Appendix B).

The following steps, illustrated in Figure 2.3, compute the squared difference between two input arrays. Our registration application uses a very similar sequence of operations to compute similarity metrics between images.

1. Create textures in GPU global memory that hold the input array values.
2. Specify a quadrilateral geometric primitive and map these textures to its surface. When the primitive is rasterized, its fragments should cover the computational domain of interest.
3. Define a fragment shader program (i.e. kernel) for computing the squared difference between two texture elements. Listing 2.1 gives an example of such a

shader. It performs basic mathematical operations on values accessed from the textures.

4. Render the quadrilateral to the frame buffer, thereby executing the kernel on all of its fragments in SIMD fashion.

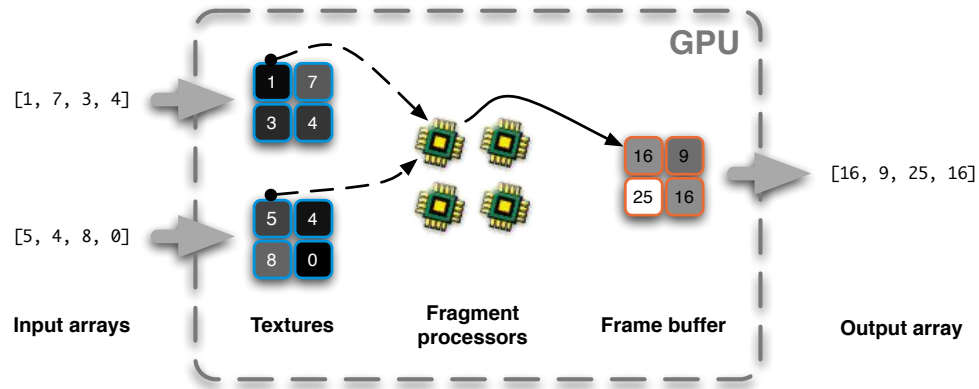


Figure 2.3: Example of parallel computation mapped to fragment processors in the GPU rendering pipeline

```

0 uniform sampler2D inputTexture0;
  uniform sampler2D inputTexture1;
2
3 void main()
4 {
5     vec4 A = texture2D(inputTexture0, gl_TexCoord[0].st);
6     vec4 B = texture2D(inputTexture1, gl_TexCoord[1].st);
7     gl_FragColor = pow(A - B, 2.0);
8 }

```

Listing 2.1: Example GLSL fragment shader for computing the square difference between values in two textures

Rendering the primitive in step 4 above executes the shader kernel of Listing 2.1 over all elements of the input streams. In this shader, objects are declared to sample the input streams stored as 2D textures (lines 0-1). Next, the shader reads data values at the interpolated texture coordinates (lines 5-6). The output fragment colour is set to the squared difference of the values (line 7). We note that no customized vertex processing is required in this example.

The output of our calculations reside in the frame buffer. These values can be used as input for successive rendering passes or transferred to the host system's memory.

In the GPGPU programming model that we have presented, fragment shaders can *gather* data from texture memory and use it in setting the output fragment colours (see Fig. 2.4). However, fragment shaders are incapable of performing arbitrary writes

(also called *scatter*) to texture memory, because the output position of each fragment is equal to its predetermined location in the frame buffer.

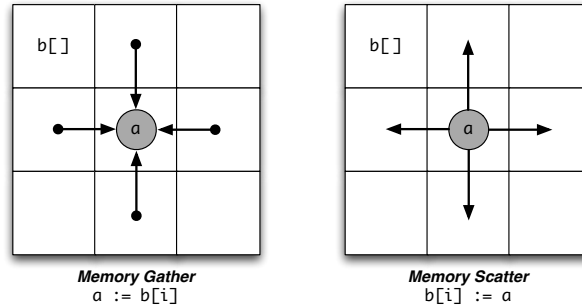


Figure 2.4: Schematic of memory gather and scatter operations

It is, however, possible to perform limited memory scattering operations during vertex processing by setting vertex coordinates to values fetched from textures [33]. The new unified shader architecture discussed in the next section enables arbitrary memory scattering, in addition to much more flexibility and power in GPU programming.

2.4 Unified Shader Architecture

We conclude with a discussion of the unified shader architecture of the most recent lineup of GPUs. As we shall see, the notion of processors devoted to fixed vertex and fragment tasks no longer exists in this architecture, where generic processors are allocated as needed at different stages of the pipeline. This essentially enables the programmer to completely abstract the GPU as a parallel processing engine, removing all semblance to traditional graphics rendering.

Using graphics hardware for general-purpose computation poses several challenges to the programmer. For instance, the instruction set available in shader programs is limited to basic logical and mathematical operations, most of which are graphics-specific. The graphics pipeline model also does not support certain fundamental programming constructs. There is no memory stack or heap; therefore, recursive procedures and function pointers do not exist. Also, support for conditional flow branching is limited. It is implemented inefficiently in GLSL by evaluating both sides of the branch, then discarding the results corresponding to the branch not taken [34]. Divergent branching reduces performance by stalling processors running threads that do not branch.

Limited data storage and handling must be considered in shader programming. As we have seen, data to be processed by the graphics pipeline is formatted as either arrays of vertex attributes or as textures. Transfers of large amounts of data to and from the GPU should be minimized. They can cause large bottlenecks in processing, since transfer rates across the bus are an order of magnitude slower than internal GPU

transfer rates. Also, algorithms that exhibit unpredictable memory access patterns do not conform well to the GPU rendering model [30].

Prior to NVIDIA’s introduction of the GeForce 8 series GPUs in 2006, each stage of the graphics pipeline was accelerated using special-purpose hardware. This type of configuration made it difficult to achieve optimal load balancing of processors in both graphics and general-purpose applications. For instance, geometry-heavy applications could saturate the vertex processors, leaving many idle pixel shaders. Likewise, applications with many pixels or complex fragment shading programs could result in idle vertex processors.

Over the last decade, there has been a major shift towards eliminating these complexities from GPGPU programming. The major development towards this goal was the unification of graphics hardware components: With increased programmability, the instruction sets of the vertex and fragment stages have become more similar, ultimately enabling manufacturers to use a single type of hardware unit for all vertex and fragment shader processing [26]. In this unified architecture, all programmable pipeline computations are distributed among a grid of identical *shader units*. The shader units are processors that are allocated as needed at vertex, fragment, and geometry assembly stages of the pipeline, yielding more optimal hardware utilization. Shader units are not devoted to a fixed task, preventing GPU performance from depending on the slowest pipeline stage. NVIDIA refers to this hardware design and its associated programming framework as the Computed Unified Device Architecture (CUDA) [31]. The individual programmable shader units are called CUDA cores.

The CUDA architecture and programming tools have greatly simplified the use of GPUs for general-purpose computations. Software libraries and hardware abstraction layers are provided to allow developers to program the cores in languages such as C and C++, without needing to navigate through traditional graphics-oriented structures with graphics APIs. The processing cores can be accessed directly, instead of only at intermediate steps within the graphics pipeline. Many highly optimized scientific packages have been written for CUDA [26, 35]. Another well established framework for parallel programming on modern processors is the Open Computing Language (OpenCL) [36]. It was released in late 2009 and is managed by a non-profit consortium. OpenCL works on multicore CPUs, GPUs, and other modern parallel processors from multiple vendors, including NVIDIA.

In CUDA, computations are expressed as kernels that are instantiated over a grid of thread blocks [31]. This grid, illustrated schematically in Figure 2.5, is a logical representation of the GPU itself. A block contains parallel, synchronized threads that each execute their own instance of the same kernel. Because of this structure, CUDA hardware is said to execute in a single-instruction, multiple-thread manner. The minimum unit of thread granularity is given by a *warp* of 32 threads. If threads of a warp diverge due to a data-dependent conditional branch, the hardware serially executes the entire warp on both paths of the branch.

CUDA exposes multiple levels of the memory hierarchy, including registers and private memory within threads, shared memory among threads in a block, global memory between blocks, and the host application’s CPU memory. The dimensions of the grid and blocks can be set at run-time based on the input data size. Each thread is

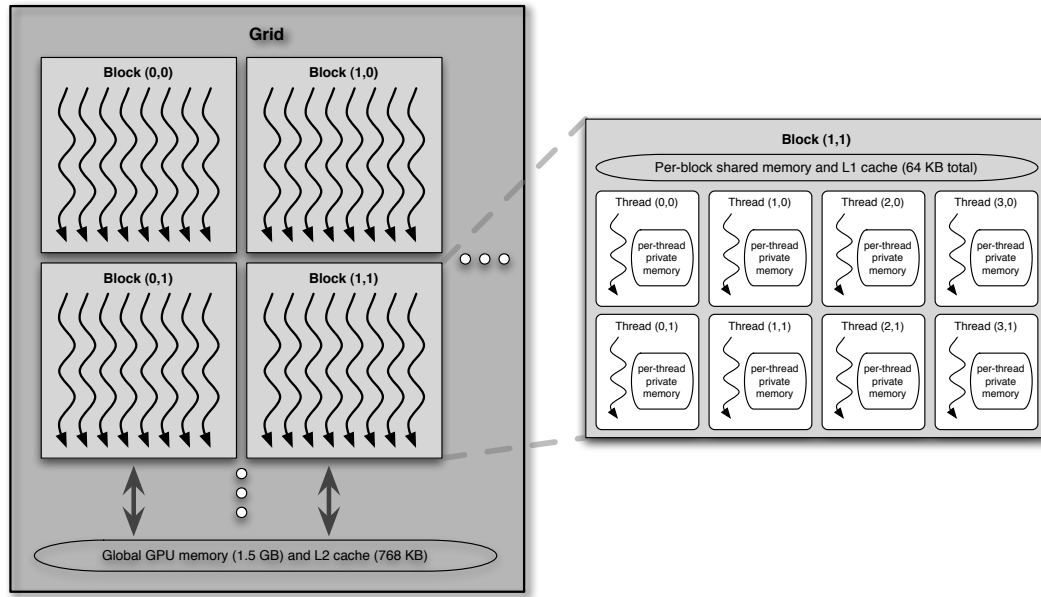


Figure 2.5: Memory processing hierarchy of thread blocks within CUDA

assigned an identifier within its block, and each block is assigned an identifier within the grid. These identifiers can have up to three dimensions, which facilitates indexing of volumetric data.

NVIDIA's latest GPU architecture built on CUDA is codenamed Fermi. In Fermi, the GPU is organized into an array of streaming multiprocessors (SMs) that execute in parallel and asynchronously. Each SM contains 32 CUDA cores, 16 memory load/store (LD/ST) units, and four special function units (SFUs) [35]. The SFUs compute transcendental functions, such as reciprocal, square root, and sine.

Figure 2.6 (from Glaskowsky [37]) shows a diagram of the Fermi streaming multiprocessor with these elements organized into four columns. One can think of the thread blocks as logical representations of these multiprocessors. Likewise, threads are logical representations of cores within a multiprocessor.

The most recent Fermi GPU, the GeForce GTX 480, has 15 SMs, for a total of 480 CUDA cores. (We noted performance times for this GPU in section 2.1.) Each SM executes a kernel on a warp of 32 threads during one clock cycle, sending instructions that saturate any two of the four columns of elements in Figure 2.6 [38]. All threads of a warp execute the same instruction, but operate on different data. It is possible to run multiple kernels on different SMs at the same time; however, some traditional parallel programming methods are not supported in CUDA. For instance, threads cannot spawn new threads, nor can threads send data between multiprocessors.

Next, we move down to the level of the CUDA core itself, which is shown on the left of Figure 2.6. Each core consists of a floating-point (FP) unit, an integer (INT) unit, dispatch logic for instructions and operands, and registers for holding results

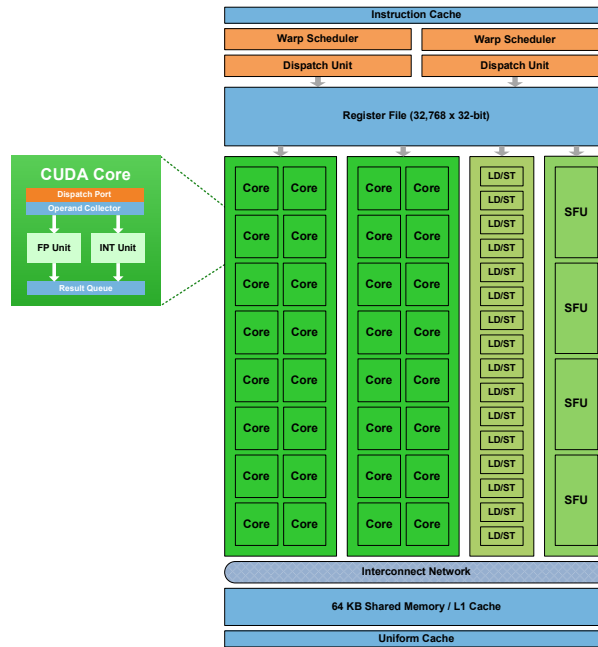


Figure 2.6: Block diagram of the NVIDIA Fermi architecture’s streaming multiprocessor containing 32 CUDA cores

and context switching between threads. The cores of a streaming multiprocessor share common resources, such as registers, local memory, L1 cache, and an interface to the GPU’s common L2 cache.

Each CUDA core executes one single precision floating-point or one integer operation per clock cycle. A double-precision operation requires two cores. Thus, an SM can perform 16 double-precision operations or 32 for single precision operations in one clock cycle. All floating-point operations are implemented following the most recent IEEE standard specifications [37].

The Fermi GPU moves data to and from the host application across the PCI Express 2.0 bus. This interface allows it to directly address and cache individual items in host memory. And with 64 bits of address spacing, all host memory can be mapped into the GPU’s address space. Addressing within Fermi is also unified, meaning that GPU global memory and local core memory are mapped to the same space. This makes memory pointers and full object-oriented C++ programming possible within CUDA.

As GPU technology advances, we expect that it will be applied to an ever-expanding computational domain. Major development and research efforts have already led to a loss of distinction between the traditional roles of the CPU and GPU. For instance, shader and CUDA implementations of methods for advanced graphics rendering, image processing, physical simulation, data structure manipulation, and scientific computing abound [39]. The technology has also been applied extensively in medicine,

such as for tomographic reconstruction [40], image segmentation [41], and volume rendering use raycasting [42].

Despite the advantages outlined above of using CUDA to accelerate computations, we opt for the graphics rendering pipeline in our application. We shall show that image registration conforms well to the traditional shader-centric model of GPGPU as implemented using OpenGL and GLSL.

Chapter 3

Background on Accelerated Image Registration

3.1 Introduction

Image registration—the task of mapping images into a common coordinate space—is an important investigative tool in science [8, 12]. In medicine, image registration is necessary when comparing or integrating diagnostic scans acquired at different times, using different devices, or of different subjects. It is essential in many investigative and diagnostic medical workflows, such as to correct for patient motion [43], to evaluate longitudinal changes in subjects [6], and to combine images of multiple individuals to create atlases of normal and abnormal anatomy [44], physiology [45], and function [46]. In clinical practice, registration is used to improve surgical planning and guidance [47], electrode localization in the brain [48], radiotherapy treatment planning [14], identification of disease response to therapy [49], and many other medical procedures [3].

Other applications of three-dimensional data registration abound. It is used in computer vision to compute depth from pairs of stereo images [50]. It aids in the identification and comparison of the 3D structures of proteins [51] and other molecules [52], and it improves the reconstruction of data acquired from electron tomography [53] and laser microscopy [54]. By allowing integration of molecular and imaging data from multiple research groups, registration is critical in enabling progress on the grand problems of neuroscience [55].

Registration is used to monitor changes in data over time, as applied extensively in geophysics for prospecting [56] and for seismic monitoring [57]. It is an important tool for monitoring climate change during the assessment of remotely sensed, multi-spectral satellite data of the earth [58]. Registration also helps in comparing changes in the spatial distribution of activated genes within multicellular organisms [59].

3.2 Accelerated Image Registration

On each iteration of automated image registration, the number of operations executed by the transformation, resampling, and similarity metric routines is on the order of the fixed image size. This is because these operations are performed in the coordinate space of the fixed image. This leads to high computational demand, since medical image datasets routinely contain millions of voxels. Factors shown to limit intensity-based registration performance include the number of interpolations following coordinate transformation [4, 60] and similarity metric evaluation [61]. The logic of the optimizer typically requires relatively few operations compared to the three other components shown in Figure 1.4.

Several researchers have leveraged parallel high performance computing (HPC) architectures in order to alleviate the execution time and memory constraints of reg-

istration on desktop workstations [22]. In the following sections, we give a review of parallelized and accelerated registration methods. Although we present speedups as stated by the researchers, these results should be interpreted with some caution. There is often no guarantee that the “sped-up” and “control” applications were implemented using equivalent algorithms. Also, full details regarding optimizations of CPU implementations are not always given (e.g. architecture streaming extensions, compiler optimizations, floating-point precision).

3.2.1 Registration on Parallel Hardware

Christensen et al. compared implementations of elastic non-linear registration on both multiple-instruction, multiple-data (MIMD) and single-instruction, multiple-data (SIMD) parallel processing computers [62]. The MIMD computer, a Silicon Graphics (SGI) Challenge, had a shared memory configuration. The registration algorithm was divided into sub-procedures distributed among its 16 processors, each of which had access to all memory. They demonstrated that performance scales linearly with the number of concurrently running processors. Their SIMD implementation executed on a MasPar computer with a massively parallel mesh of 128×128 processors. This was a distributed data system, meaning that the data was explicitly partitioned among the processors. The authors showed that the shared-memory MIMD implementation ran a minimum of four times faster than the distributed-memory SIMD implementation, though both the MIMD and SIMD computers had comparable peak theoretical performances. It was concluded that the SIMD algorithm was less efficient due to non-linear deformation computations requiring random access to data distributed across multiple processors.

Warfield et al. created a parallel implementation of inter- and intra-subject rigid-body registration driven by the minimization of tissue label mismatch [22]. They distributed metric computation and resampling over the nodes of two Sun Enterprise Server 5000 systems, each with eight CPUs at 167 MHz. For $256 \times 256 \times 52$ images, the authors achieved execution times of five to ten minutes, which they deemed to be clinically acceptable. Ourselin et al. parallelized affine registration on a cluster of ten dual-Pentium III 933 MHz machines connected by fast ethernet [63]. Their implementation used a combination of shared memory and message passing between nodes to achieve a maximum eight-fold speedup factor.

Wachowiak and Peters presented two new derivative-free optimization methods that take advantage of distributed similarity metric evaluation [64]. Their methods combined global and local search strategies via sampling and recursive subdivision of the parameter space. Their rigid-body registration experiments showed performance gains of up to five times running on an SGI Altix 3000 with 12 CPUs at 1.3 GHz.

Using a shared-memory architecture, Rohlfing and Maurer were able to speed up deformable registration of pre- to intra-operative MR scans by a factor of one hundred times over a serial implementation [10]. Their transformation model was based on free-form deformations with B-spline interpolation over a control point grid. They distributed the computation of the mutual information similarity metric and its spatial gradient over 64 CPUs running at 400MHz on an SGI Origin 3800 supercomputer.

The algorithm partitioned image data among independent threads, then combined resulting calculations in shared memory. Total execution time for $256 \times 256 \times 100$ images was about one minute.

Ino et al. also created a shared-memory, parallel implementation of deformable registration using adaptive mesh refinement and mutual information [16]. Their program ran on a cluster of 128 CPUs at 1 GHz interconnected by a fast local area network. They brought registration times for $512 \times 512 \times 295$ CT volumes from about 15 hours (on a single CPU) to clinically compatible times—by their standards—of about ten minutes. The main performance bottlenecks in their implementation were computation of the similarity metric and its spatial gradient.

More recently, Plishker et al. achieved real-time rigid and elastic registration by combining several different hardware platforms [65]. The platforms operated independently of one another and were responsible for different components of the registration cycle. Rigid-body transformations were computed using a GPU (NVIDIA Quadro FX 1400); image similarity metric and elastic transformations were computed on a field programmable gate array (Altera Stratix EP1S40); and metric gradients were distributed across an eight node cluster of CPUs (3 GHz Intel Xeon). The authors claimed accelerations of one hundred times compared to a uniprocessor implementation. They believed that their heterogeneous platform for registration was robust, scalable, and cost-effective.

Image registration using high performance computing has not achieved widespread clinical use due to the low availability and compatibility of such resources in clinical settings, such as the operating room. For example, ideal computer equipment for image-guided therapy should be physically compact, easy to maintain, efficient on power, and cost-effective. Nevertheless, researchers have expressed hope that HPC systems would become available to clinicians through the use of high-speed networks with guaranteed quality of service [10]. They believed that low latency could be assured for critical registration tasks on a shared machine using high priority job status.

Hastings et al. took the notion of HPC for image analysis applications to the extreme by proposing a grid environment [66]. In their proposed infrastructure, HPC resources would be dynamically shared among groups over the internet. They showed that this grid could execute medical image segmentation, registration, and visualization algorithms from the Insight Toolkit (ITK) [67] and the Visualization Toolkit (VTK) [68] using a stream programming model.

Since the advent of the customizable graphics pipeline, the attention of most accelerated registration research has shifted to GPGPU from traditional cluster and parallel computing approaches. Indeed, as a result of their increasing power and ubiquity, GPUs are being applied to a broad range of computational domains [26]. We summarize prior work related to linear and non-linear medical image registration using graphics hardware.

3.2.2 Registration on the GPU

The computational capabilities of GPUs per unit cost have compounded at a rate significantly outpacing that of traditional CPUs [23]. By devoting the majority of their transistors directly to computation, GPUs provide high arithmetic intensity. Top-end commodity graphics cards (currently selling for less than \$500 US) contain hundreds of parallel processors called shaders, each clocked at over 1 GHz¹, yielding over a tera-FLOP of computing performance on the desktop. This exceeds by an order of magnitude the performance of modern CPUs, which have a significant portion of transistors devoted to branch prediction and out-of-order execution, for example.

Prior to the introduction of programmable GPUs, Hastreiter and Ertl demonstrated a method using 3D texture mapping to accelerate interpolation calculations in multi-modal, rigid-body registration [60]. They had identified the large number of moving image interpolations to be the most significant factor contributing to overall registration time. Using specialized Indigo² Maximum Impact and SGI Onyx Reality Engine II graphics workstations, they achieved a two- to three-fold increase in performance for trilinear interpolation. Their method also enabled interactive visualization of the fused image volumes, which were stored in texture memory. Hastreiter et al. subsequently extended this work by using 3D texture mapping to apply non-linear deformations to subdivided image patches [69].

The interpolation of non-linear transformations on the GPU was first presented by Rezk-Salama et al. [70]. Their work was based on the earlier algorithms of Hastreiter et al. [69]. They subdivided image volumes into piecewise linear patches and used 3D texturing hardware to densely interpolate transformations across them. Volumetric deformations were computed with dependent texture look-ups in a custom fragment shader program. Using specialized graphics workstations (SGI Onyx2 with BaseReality GPU), the authors achieved a speed increase of 50 times compared to a CPU-based version of the same software. This allowed their tool's use in a clinical study to compensate for brain shift during neurosurgery [4]. The shift of brain tissue following craniotomy and brain resection often prohibits intra-operative navigation using pre-operatively acquired images. Registration of pre- and intra-operatively acquired images can correct these anatomical shifts [47].

Soza et al. extended this non-linear deformation method to use 3D Bézier functions in order to provide inherent smoothness and elasticity for the transformations [71]. The Bézier functions were parameterized by a sparse lattice of 3D control points. Using a commodity NVIDIA GeForce3 graphics card, they achieved average timings of about seven minutes for registering $256 \times 256 \times 112$ pre- and intra-operative MR scans. Similarly, Levin et al. presented a GPU algorithm to approximate thin-plate spline (TPS) transformations using piecewise linear transformations [72]. Their deformation algorithm achieved 7- to 65-fold acceleration with NVIDIA GeForce FX 5600 and ATI Radeon 9700 Pro GPUs over a CPU-based implementation; however, this algorithm was not included in a complete image registration application.

Strzodka et al. were also among the first to report acceleration of deformable reg-

¹Product specifications for the latest NVIDIA GPUs are available online at <http://www.nvidia.com/page/products.html>.

istration on the GPU [73, 74]. Their method was based on the minimization of an energy functional using gradient flow [75]. They solved the resulting partial differential equations with a finite element system on the GPU. Since their energy functional was derived from differences of image intensities, their method only applied to single-modality registration. Compared to a CPU-based implementation, they reported four-fold acceleration for 2D images on an NVIDIA GeForceFX 5800 Ultra GPU. Köhn et al. presented a straightforward extension of this gradient flow method to three dimensions [76]. However, they did not achieve substantial acceleration due to memory copy bottlenecks. The study reported 12-fold acceleration for 3D, single-modality rigid registration.

Chisu published an extensive study on the use of programmable fragment shaders to evaluate similarity metrics on the GPU [77]. This strategy eliminated very costly copies of transformed images from video memory to main memory for metric evaluation on the CPU, yielding up to five-fold acceleration for 3D registration. Such image copies from GPU to CPU over limited-bandwidth data buses significantly slowed down earlier 3D registration applications.

Ino et al. presented a method for aligning 3D computed tomography volumes to 2D fluoroscopy images on the GPU [78]. The method was based on the alignment of 2D projections of the CT volume, called digitally reconstructed radiographs (DRRs), to the fluoroscopy image. They used fragment shaders to compute the DRRs and image correlation metrics. They achieved ten-fold faster execution times running their method on a GeForce 7800 GTX GPU compared to a 32-node cluster of Pentium III CPUs. Khamene et al. reported between four- to five-fold speedup for another affine registration method based on image projections [79]. Their method reduced the registration of 3D images to a series of simpler, 2D registrations of the images' orthogonal projections.

The registration methods that we present in this chapter are built on a framework originally presented by Chan [80]. He was the first to report acceleration of both affine and free-form non-linear registration by implementing the image transformation, interpolation, and similarity metric modules on the GPU. His non-linear transformations were parameterized by a regular 3D lattice of control points and implemented via dependent texture look-up. Chan created shader implementations of several popular similarity metrics suited for mono- and multi-modality registration, with the exception of mutual information. For rigid-body registration, he showed speedups of about 20 and 13 times compared to a CPU implementation for mono- and multi-modal cases, respectively. Non-linear registration times were on the order of a few minutes for several representative clinical cases. The graphics hardware used was an NVIDIA Quadro FX 4500 GPU with 512 MB of video memory.

Since the initial presentation of our work in May 2008 [81], several researchers have published related results. A fairly comprehensive review of medical image registration on the GPU from 2007 to the present is given by Shams et al. [82]. This review presents the transformation models, similarity metrics, optimization schemes, and hardware that were employed for GPU registration by a number of research groups. The review also compares the relative performances of the registration applications. We believe that the most relevant work to our own was conducted by Muyan-Özçelik

et al. [83], who most likely conducted their research concurrently with us. They implemented the Demons algorithm for 3D deformable registration entirely using CUDA on the GPU. Regardless of the input dataset size, they achieved speedups of about 55 times over an optimized CPU version of the same algorithm.

3.3 Registration using Mutual Information

Suppose that we are given two misaligned images of a subject’s anatomy that may have been captured using different modalities. The images therefore contain overlapping information content with respect to the shared anatomical structures. When perfectly registered, all corresponding structures overlap and the amount of shared information is higher. In this way, we can think of registration as the maximization of the images’ *mutual information* content. The discussion below formalizes this concept.

In image registration, the principle of information content is quantified by the Shannon entropy, which was originally introduced in the context of communication theory in 1948 [13]. Let us identify an image’s intensity values with the random variable X and the probability density function (PDF) $p_X : X \rightarrow [0, 1]$. This function is estimated by counting the number of occurrences of each gray value intensity and then normalizing.

Shannon defined the information content associated with image intensity $x \in X$ as $I(x) = \log \frac{1}{p_X(x)}$. According to this definition, the more likely the occurrence of intensity x , the lower its associated information². The information attributed to an intensity is therefore akin to uncertainty in its occurrence.

The entropy of X is defined as the expected value of its information content:

$$H(X) = \sum_{x \in X} p_X(x) I(x) = - \sum_{x \in X} p_X(x) \log p_X(x). \quad (3.1)$$

For an image, X here denotes the set of all discrete image samples. We note that a completely random image, in which every intensity occurs with equal frequency, has maximal entropy for its size, whereas an image with a single peak intensity will have a low entropy. In this sense, entropy is also a measure of dispersion of the image’s probability density function.

The joint Shannon entropy between two discrete random variables X and Y with joint PDF p_{XY} is defined similarly:

$$H(X, Y) = - \sum_{x \in X} \sum_{y \in Y} p_{XY}(x, y) \log p_{XY}(x, y). \quad (3.2)$$

Analogous to the univariate case, this measure can be interpreted as the combined information content of the two variables (or images).

²The logarithm function makes the definition of information content additive for independent events in a distribution, since for $x_1, x_2 \in X$, $I(x_1 \text{ AND } x_2) = I(x_1) + I(x_2)$.

The joint image PDF changes as the degree of registration changes. With increased overlap of anatomical structures, clusters begin to appear in the PDF that correspond to gray values of aligned structures. Figure 3.1 shows the joint PDFs (estimated as the joint histogram) of a three-dimensional T1-weighted MR image with rotated versions of itself. Histogram bin counts are shown on a natural logarithmic scale. From top left to bottom right, the rotations are 0, 1, 2, 3, 4, 5, 10, and 20 degrees in the axial plane. With increasing misalignment, the clusters become more dispersed as new combinations of co-occurring gray value pairs emerge.

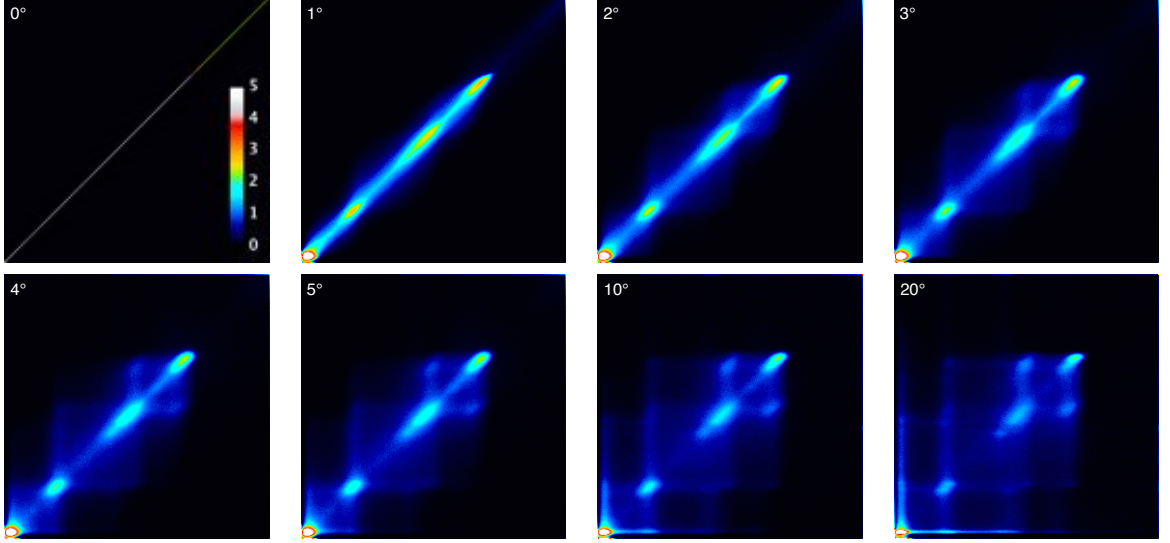


Figure 3.1: Joint histograms of a T1-weighted image with itself for varying degrees of misalignment by pure axial rotation (shown natural logarithmic scale)

The joint entropy is a measure of this dispersion and it decreases with improved registration. However, it alone should not be used to drive image registration. This is because it is computed on the overlapping domain of the images and is therefore sensitive to the overlap size. In practice, the joint entropy may be low for complete misregistration [19]. This is because minimizing joint entropy is analogous to minimizing information content of the overlapping region, which can lead to zero overlap. Instead, we require a metric that maximizes the shared image information content over the overlap.

We solve this problem by instead maximizing mutual information (MI):

$$\text{MI}(X, Y) = H(X) - H(X|Y) = H(X) + H(Y) - H(X, Y), \quad (3.3)$$

Mutual information is non-negative, since the joint entropy of two random variables is always less than or equal to the sum of their individual entropies: $H(X, Y) \leq H(X) + H(Y)$ [84]. By taking the marginal entropies into account, MI avoids the overlap problem of using joint entropy alone. This metric was first used for image registration in 1996 by Viola and Wells [18] and Maes, et al. [84]. It has been applied to the registration of most imaging modalities, including MR, CT, and positron emission tomography (PET).

In summary, *mutual information is a measure of the shared information content and statistical dependence between images*. We register two images by finding a transformation that maximizes MI. The measure is maximal when a one-to-one mapping exists between both images (i.e. one image can be completely predicted from the other). In this case, $H(X) = H(Y) = H(X, Y)$, so the metric equals $H(X)$. It is minimal when the images are statistically independent (i.e. contain no redundant information): $H(X, Y) = H(X) + H(Y)$, and the metric is zero.

A pitfall of mutual information is that the overlapping domain of medical images can vary considerably with their degree of alignment. In cases where the relative background and foreground areas even out, MI may incorrectly increase with greater misalignment [13]. A closely related variant called *normalized* mutual information (NMI) has been shown to be a more robust similarity metric than MI [19]. It was shown experimentally for MR-CT and MR-PET registration cases that the NMI metric is invariant to changes in both the image overlap and field of view. These properties make it significantly more robust than MI for automated registration, since medical images commonly have truncated fields of view and may have overlaps that vary considerably with alignment. The normalized metric is defined as

$$\text{NMI}(X, Y) = \frac{H(X) + H(Y)}{H(X, Y)}. \quad (3.4)$$

Mutual information and normalized MI are widely accepted as two of the most robust and accurate similarity metrics for multi-modal affine and deformable registration [13, 19].

3.3.1 Image Histograms

For numerical computation of entropies in Equations 3.1 and 3.2 for images, we estimate the marginal and joint PDFs by the marginal and joint image intensity histograms. The marginal histogram of a gray-scale image is a discrete array of bins, each of which represents a unique intensity range. The value of a bin is the number of image pixels whose intensities fall within the bin's range.

The *joint histogram* is a 2D matrix whose indices also correspond to image intensity value ranges. For two images A and B , it is computed by counting co-occurrences of all intensity pairs $a \in A$ and $b \in B$. That is, the number of pixel locations (i, j) where $A(i, j) = a$ and $B(i, j) = b$.

As an example, suppose that a joint histogram h has dimensions $N \times N$ and that images A and B have intensities in $[0, A_{\max}]$ and $[0, B_{\max}]$. If we equally distribute the image intensities over the histogram bins, then bin indices $(i, j) \in [1, N] \times [1, N]$ correspond to the intensity ranges $[\frac{i-1}{N}A_{\max}, \frac{i}{N}A_{\max}]$ and $[\frac{j-1}{N}B_{\max}, \frac{j}{N}B_{\max}]$.

Bin entry $h(i, j)$ of the joint histogram denotes the number of times that intensities $a \in [\frac{i-1}{N}A_{\max}, \frac{i}{N}A_{\max}]$ of image A coincide spatially with intensities $b \in [\frac{j-1}{N}B_{\max}, \frac{j}{N}B_{\max}]$ of image B . The most straightforward approximation of the joint PDF is then $p(a, b) = \frac{1}{\sum_{i,j} h(i, j)} h(i, j)$.

The marginal PDFs are obtained by summing along the rows and columns of the joint PDF. Most implementations of MI for medical image registration use on

the order of 100 bins per histogram dimension, yielding upwards of 10,000 bins. Intuitively, the dispersion of clusters in the joint image histogram decreases as their degree of anatomical alignment and mutual information increase.

Forming image histograms requires looping over every image voxel. This is computationally demanding in automated registration, since it must be repeated on every cycle iteration. As shown in the algorithm of Figure 3.2, it is straightforward to compute the joint histogram (stored in the 2D, $N \times N$ array *bins*) for two images *A* and *B* on a general-purpose computer. This method scatters image intensities into the histogram. Access to the input images is sequential, whereas access to the histogram *bins* is data-dependent. It is assumed here that *A* and *B* have the same dimensions.

```

let bins be an integer array with dimensions ( $N,N$ )
initialize all elements of bins to 0

for  $i = 1$  to width(A)
  for  $j = 1$  to height(A)
     $a = A[i, j]$ 
     $b = B[i, j]$ 
    rescale  $a$  and  $b$  to the integer range  $1 \dots N$ 
     $bins[a, b] = bins[a, b] + 1$ 
  end
end
end

```

Figure 3.2: Pseudocode to compute the joint histogram of two images by scattering intensity values

3.3.2 Mutual Information on the GPU

It was previously reported that the GPU could not achieve significant performance gains for the computation of mutual information [4, 79]. This is because on original GPU hardware, it was not possible to write to arbitrary memory locations while constructing the histogram. Studies have also shown that GPU performance gains diminish if data is processed in an unpredictable manner [85], as is the case for the histogram bin updates in this method.

Due to their inability to scatter data, early GPU implementations computed histograms using an extremely inefficient gathering-based approach. The algorithm given in Figure 3.3 uses this approach to compute the 1D histogram of image *A*. It iterates through all image samples to set each of the N elements in the histogram array *bins*. A given histogram bin is incremented if an image intensity is found that falls within the bin's intensity range.

An implementation of this gather-based method for 1D histograms on the GPU was given by Green [86]. In this method, a fragment shader loops through every image pixel to construct each histogram bin. The shader renders an output fragment

```

let bins be an integer array with length  $N$ 
initialize all elements of bins to 0

for  $n = 1$  to  $N$ 
  for  $i = 1$  to width( $A$ )
    for  $j = 1$  to height( $A$ )
      if  $A[i, j]$  falls in the range of  $bins[n]$  then
         $bins[n] = bins[n] + 1$ 
      end
    end
  end
end

```

Figure 3.3: Pseudocode to compute the histogram of an image by gathering intensity values

only if the incoming pixel intensity is within the current bin’s intensity range. The number of rendered fragments (i.e. the bin value) is read back to the CPU using an *occlusion query* function from the graphics API. Occlusion queries are used to count the number of fragments that are not rendered during a pass. A limitation of occlusion queries is that they require the CPU and GPU to synchronize, potentially stalling the rendering pipeline. This leads to poor performance in Green’s method.

The gathering method of Fluck is similar [87]. It divides the input image into tiles and renders a local 1D histogram for each tile. Like Green’s method, a fragment shader gathers all pixel intensities of a tile in order to compute each local histogram bin. The local histograms are summed into a global histogram using a sequence of rendering passes.

The methods of Green and Fluck are inherently inefficient, since each bin value is computed using a gather operation over all input pixels. The methods have asymptotic complexity $O(PN)$, where P and N are the number of image pixels and histogram bins, respectively. Recent advances in graphics hardware capabilities have enabled much more efficient histogram computation on the GPU using memory scatter operations in the vertex shader.

Scheuermann and Hensley presented an efficient GPU implementation of 1D histogram computation using vertex scattering [88]. They create histograms with arbitrary numbers of bins in a single pass over the input image, requiring $O(P)$ operations. For each input pixel, a vertex is created that is rendered as a point primitive to its histogram bin location in the frame buffer. The authors reported significant accelerations compared to the gathering methods of Green and Fluck. For example, computation of a 256×256 histogram for a 256×256 image took 1.09 ms on a GeForce 7800 GTX. On prior hardware, vertex scattering could lead to memory coherence issues and much lower performance [23]. We further discuss the method of Scheuermann and Hensley in section 4.1.5, since it provides us the means to implement mutual information on the GPU.

The Compute Unified Device Architecture from NVIDIA has also been used to accelerate histogram computation. CUDA gives direct access to the instruction set and memory of the GPU's parallel processors, allowing arbitrary memory scattering. The general form of the latest CUDA histogram methods is illustrated in Figure 3.4. First, the input image is broken up into equally sized partitions that fit into shared memory. A separate thread or group of threads is assigned to each partition and computes its histogram. Next, these local histograms are summed in global memory in order to create the global image histogram. If global atomic operations are available, then this merger step can be performed by multiple, concurrent threads with exclusive write access to global memory [10]. Otherwise, each local histogram must be added sequentially to the global histogram.

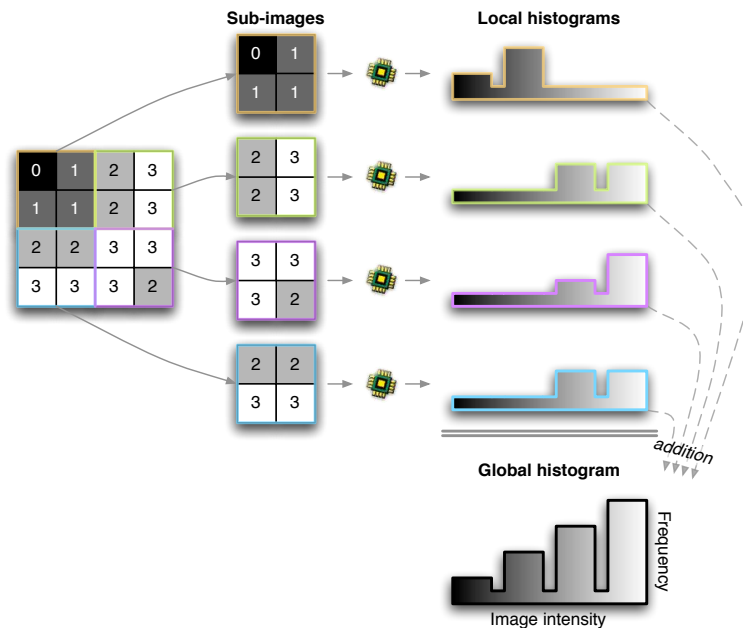


Figure 3.4: Summation of partial histograms (computed in parallel) into a global histogram using either atomic or sequential operations

However, there are two major limitations in CUDA that histogram generation algorithms must circumvent. The first is that shared memory among threads is limited to 16 KB. Shared memory is needed to store image histograms that are updated in parallel. Secondly, CUDA graphics hardware did not support *atomic operations* on shared or global memory until 2009. Atomic operations prevent interference between concurrent threads accessing the same memory. During an atomic operation on a particular memory address by one thread, it is guaranteed that no other thread can access the address. This functionality is necessary to deal with simultaneous updates to the same histogram bin. The latest hardware now supports atomic integer functions on 32-bit integers in shared memory between threads and global GPU memory.

Methods for computing histograms with 64 and 256 bins using CUDA are de-

scribed by NVIDIA [89]. However, these methods cannot be applied to MI computation for registration, since they do not scale to higher histogram resolutions. Both methods divide the input image among GPU execution threads and compute a local histogram for each sub-image. The 64-bin method assigns a local histogram to each thread; the 256-bin method shares a local histogram between a warp of 32 threads. The local histogram and sub-image sizes are limited by the total size of shared memory. Also, without atomic updates, the 256-bin method requires explicit resolution of shared memory write collisions. Atomic updates are simulated using a method that tags the target bin with the writing thread’s ID. Performance rates of 10 GB/s and 5.5 GB/s were reported for the 64-bin and 256-bin methods, respectively.

Shams and Kennedy presented a method for approximating histograms with thousands of bins for large images using CUDA, achieving up to 30 times acceleration compared to a CPU-based implementation [90]. The method maintains one partial histogram per warp, with all threads of a warp sharing access to the histogram using simulated (software-based) atomic memory writes. The partial histograms are summed using a parallel reduction technique. While histograms of arbitrary size can be created, the method’s efficiency greatly decreases with the number of bins. The highest performance is achieved for 1000 bins or less. The authors reported only two- to four-fold acceleration over the CPU for 10,000 bins. They attributed this poor performance to the limited size of cached shared memory (4K of 32-bit words), which is used to hold the intermediate partial histograms. Complexity of the image data also affects speed, with degenerate distributions resulting in more collisions and slower throughput.

Shams and Barnes applied this method to accelerating mutual information on the GPU [85]. They compared performance using “approximated” histograms (calculated using a subset of the input samples) and “exact” histograms (calculated using all input samples). For 100×100 approximated and joint histograms, they reported metric throughputs of approximately 5 GB/s and 0.9 GB/s, respectively. The approximated histogram method gave 21- to 25-fold performance gain over the CPU.

Ohara et al. were also able to accelerate MI on ubiquitous consumer graphics hardware. They implemented 3D linear registration using MI on the Cell Broadband Engine multi-core processor [91]. The Cell processor was jointly developed by Sony Corporation, IBM, and Toshiba. The authors reported a registration time of approximately one second for a pair of $256 \times 256 \times 30$ images, though no accuracies were given. Their implementation, which is highly optimized for the Cell architecture, uses only 1% of the original image samples to compute the MI metric.

Most recently, Jung and Wesarg were able to take advantage of atomic addition operations in order to compute NMI on a GeForce GTX 260 GPU (circa 2009) [92]. They tested their method on clinical images from the Retrospective Image Registration Evaluation (RIRE) Project³ [93], achieving times of about ten seconds for CT-MR, CT-PET, and PET-MR registration cases. The authors reported speedup factors between five and seven times compared to a single-threaded implementation

³The RIRE Project images and registration method evaluations are available online at <http://www.insight-journal.org/rire/>.

on an Intel Quad Core Q6600 CPU with 2.4 GHz.

In the next chapter, we describe our GPGPU approach to automated medical image registration, which we implement GLSL. Despite the inherent limitations of traditional shader programming that we have discussed, our registration application conforms well to the rendering paradigm, and we do not use CUDA or any other generic parallel computing frameworks.

Chapter 4

GPU-Accelerated Image Registration Methods

4.1 Accelerated Affine Image Registration

Automated, intensity-based affine registration is cast as the iterative optimization of a similarity metric objective function. This is a single number that evaluates the quality of registration (i.e. similarity) between a moving and a fixed image. Each iteration of the optimization cycle consists of applying a parameterized transformation to the moving image, resampling the transformed moving image into the space of the fixed image, computing the similarity metric between the two images, and generating new transformation parameters for the following iteration.

We implement the computationally intensive components of this cycle on the GPU. These components are the first three steps shown in Figure 1.4: image transformation, interpolation, and metric evaluation. We also store all 3D images in GPU video memory. This eliminates the need for repeated transfers of up to hundreds of megabytes of data over the CPU-GPU bus during each iteration. The transfer of image data has proven to be the primary bottleneck in other GPU registration programs [4]. In our implementation, the amount of data sent between the CPU and GPU is negligible, consisting of the metric value and a set of transformation parameters.

Our software is implemented in C++ and uses the OpenGL 2.0 graphics API [25] to access graphics hardware functionality. We write our GPU shader programs in OpenGL Shading Language (GLSL) [28]. Our framework is implemented modularly, with data and processes encapsulated as objects. Its structure thus generally resembles that of the Insight Toolkit (ITK) registration architecture [67, 94]. ITK is an open-source, cross-platform toolkit for performing registration, segmentation, and other image processing tasks on the CPU.

4.1.1 Volume Rendering

The registration methods that we present are formulated in terms of *3D texture-based volume rendering* and are thus naturally suited to implementation on the GPU. The purpose of volume rendering is to synthesize a virtual 2D view of volumetric data, where the rendered colour values represent physical interactions of light with the data [95]. This is usually achieved by a transfer function that maps scalar data intensities to light emission and absorption properties.

When performing texture-based rendering, the volumetric dataset is first loaded into the GPU's 3D texture memory. The next step is to map the texture's slices onto *proxy geometry*. The proxy geometry consists of a stack of equally-spaced quadrilateral polygons (or *quads*) oriented parallel to the viewing plane, as shown in Figure 4.1. The number of quads generated equals the number of slices in the texture.

Finally, the quads are rendered into the frame buffer one-by-one. This is done without depth testing, so that the quads do not occlude each other. If desired,

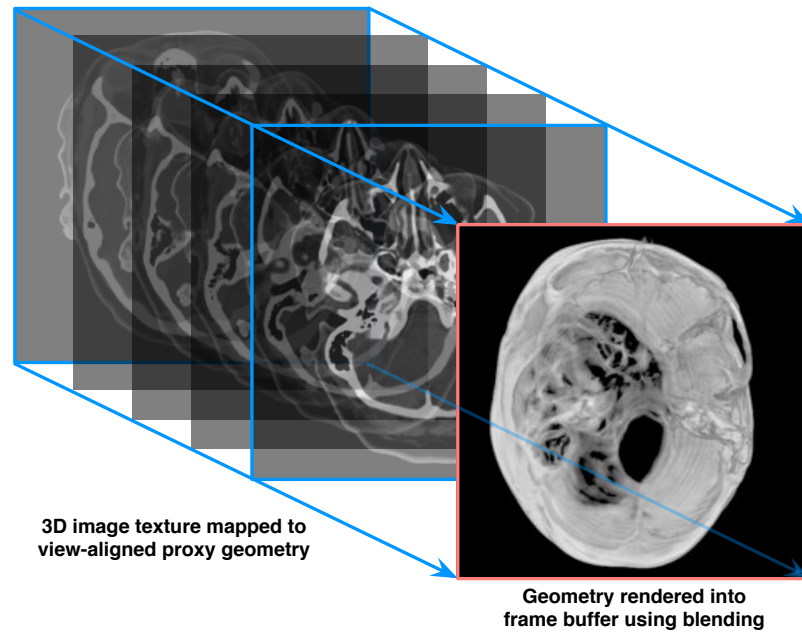


Figure 4.1: Volume rendering of a CT head dataset (inferior view) using texture mapping and view-aligned proxy geometry

lighting effects are simulated during this stage using blending functions and additional data, such as surface normals, colours, and transparencies.

If the user changes the viewing direction, then the dataset is geometrically transformed and retextured onto the proxy geometry. If the view direction is not parallel to an axis of the image, then 3D interpolation is required during the texture mapping procedure.

Every iteration of our registration algorithm is cast as the rendering of one frame of a 3D volume—with some modifications. To start, we load the fixed and moving volumetric medical datasets into two textures on the GPU. Next, we map both textures onto common quadrilateral proxy geometry. As discussed in the following section, the current registration transformation is applied to the moving image during the mapping. The image metric between the fixed and moving images is computed by shaders as the textured quads are rendered to the frame buffer.

Most raw CT, MR, and PET images have integer intensity values that span no more than a 12-bit range, whereas processed images often have floating-point values. We use GPU texture formats that match the input data format. Thus, we typically load raw images into 16-bit integer textures; processed images are typically loaded into 32-bit floating-point textures, which are available on most graphics hardware since 2008. Since the image data is scalar, we use textures that hold a single luminance intensity channel, as opposed to the standard RGBA channels.

Images are transferred from main memory to the GPU using the OpenGL function `glTexImage3D` and textures are set to exactly match the input image size. On hardware

prior to 2006, it was necessary to pad texture dimensions to powers-of-two. The dimensions of the quads that we render match those of the fixed image cross-sectional slices, since the moving image is always resampled into the space of the fixed image.

4.1.2 Affine Image Transformation

Our registration framework performs image transformations using 3D texture mapping on the GPU. Indeed, the GPU is ideally suited to performing geometric transformations, since they are called for frequently in graphics applications.

The first step in the texture mapping of a polygon is to define texture coordinates at each of its vertices. Each coordinate serves as an index into the texture. Coordinates inside the polygon are computed using 2D interpolation during rasterization. The effect of interpolation is to create a smooth map of the texture over the entire shape. Figure 4.2 shows image slices mapped to a quad using texture coordinates at the four vertices.

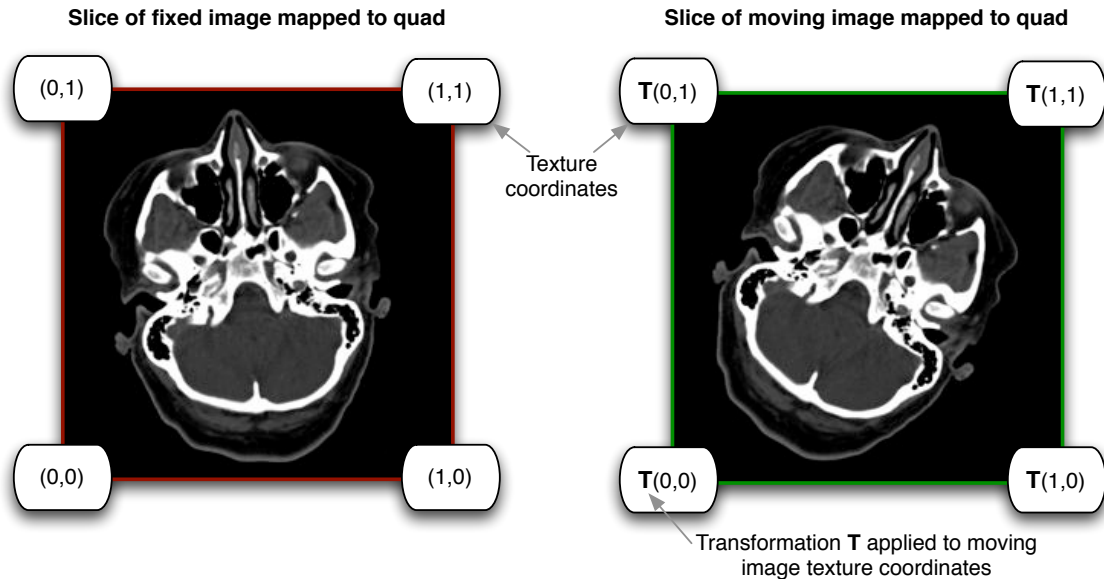


Figure 4.2: Mapping the fixed and moving images to quads using texture coordinates

We transform the moving image by modifying its mapping over the quadrilateral proxy geometry. This is done by multiplying its texture coordinates by a 4×4 homogeneous matrix. The OpenGL functions `glTranslatef`, `glRotatef`, and `glScalef` can be used to apply 3D translations, rotations, and scalings to the matrix. Alternatively, the components of the matrix can be set directly via `glMultMatrixf`. Using this method, we are able to specify arbitrary 12-parameter affine transformations. Figure 4.2 depicts an affine transformation $\mathbf{T}$ applied to the moving texture coordinates.

Listing 4.1 shows how we implement image transformations in OpenGL. In this code, the fixed and moving images are stored in textures labeled 0 and 1, respectively.

We apply an identity transformation to the fixed image (line 2) and the current OpenGL matrix transformation to the moving image (line 6).

```

0 glMatrixMode(GL_TEXTURE);
  glActiveTexture(GL_TEXTURE0); // fixed image
2 glLoadIdentity();

4 glActiveTexture(GL_TEXTURE1); // moving image
  glLoadIdentity();
6 glMultMatrixf(transformationMatrix);

```

Listing 4.1: Applying transformations to fixed and moving image texture coordinates for registration

4.1.3 Image Interpolation

Image transformations in registration require interpolation of the moving image intensities. To see this, consider the image transformation, which is defined mathematically as a function $\mathbf{T} : M \rightarrow F$ that maps spatial coordinates from the moving image domain M to the fixed image domain F . The map must cover the entire fixed image domain, since it is the target of registration. However, naively transforming the moving image using $\mathbf{T}$ results in a map that is not dense *onto* F .

We solve this problem by instead defining the inverse transformation $\mathbf{T}^{-1}$ that operates on fixed image coordinates. The transformation is then computationally evaluated by looping over all fixed image points. Given one point $\mathbf{x} \in F$, the value of the corresponding transformed image intensity $M(\mathbf{T}^{-1}(\mathbf{x}))$ is determined by interpolation of the moving image. Figure 4.3 shows a schematic of evaluating the inverse transformation at a fixed image coordinate by interpolating the moving image.

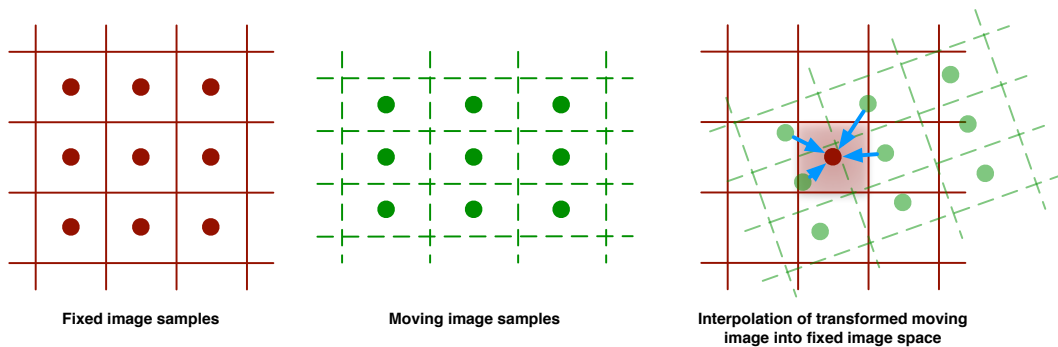


Figure 4.3: Interpolation of the moving image during transformation

The simplest and fastest interpolation method uses the intensity of the nearest neighbouring sample. However, this is known to result in aliasing and severe partial voluming effects, yielding poor registration [3]. Trilinear interpolation is among the most commonly used methods, since it provides a good tradeoff between accuracy and computation time and it has been shown to yield acceptable results for image

registration [72]. Methods using quadratic, cubic, cubic B-spline, Gaussian, and *sinc*-based interpolation kernels of different sizes have been studied in detail [96, 97].

For affine registration on the CPU, it has been shown that up to about 90% of the computational time is spent doing image transformation and (trilinear) interpolation [82]. To accelerate transformations in our GPU framework, we configure the hardware to automatically perform trilinear interpolation when reading from the moving texture. The hardware is optimized to perform fast and accurate trilinear interpolation, which is critical in real-time graphics applications, where scenes often contain multiple textures that are mapped to dynamically resizing polygons.

The GPU currently has no native support for higher-order interpolation schemes. However, efficient third-order B-spline interpolation is possible within fragment shaders using multiple texture look-ups [98]. This is done by taking advantage of automatic hardware trilinear interpolation. To complete a third-order B-spline interpolation, the method makes eight trilinear texture look-ups that implicitly combine the 64 neighbouring intensities. The overhead compared to trilinear interpolation is thus a factor of eight.

4.1.4 Difference- and Correlation-Based Similarity Metrics

Computation of the image similarity metric forms the core of our registration framework. We take advantage of the GPU’s SIMD architecture to compute point similarity metrics, which measure correspondence between individual image samples [99]. We implement several similarity metrics based on intensity difference and correlation: *mean squared error* (MSE), *mean absolute error* (MAE), and *normalized cross-correlation* (NCC). We also implement the *normalized gradient field* (NGF) metric [100], which has been reported to yield a good objective space for parameter optimization in multi-modal registration. Information-theoretic metrics are treated in the next section of this chapter.

The mean squared error (MSE) and mean absolute error (MAE) metrics are defined between an image pair A and B as follows:

$$\text{MSE}(A, B) = \frac{1}{N} \sum_{\mathbf{x} \in \Omega} (A(\mathbf{x}) - B(\mathbf{x}))^2, \quad (4.1)$$

$$\text{MAE}(A, B) = \frac{1}{N} \sum_{\mathbf{x} \in \Omega} |A(\mathbf{x}) - B(\mathbf{x})|, \quad (4.2)$$

where Ω is the overlapping sample domain of A and B , and N is the number of samples in the domain. It has been shown that MSE is the optimal choice of metric when A and B differ only by Gaussian noise [3]. (The assumption of pure Gaussian noise rarely holds in practice for intra-modality registration.) The MSE metric is sensitive to a small number of high intensity voxels, which can be caused by injected contrast or surgical instruments in the imaging field of view [77]. The MAE metric reduces the effect of such outliers.

The difference metrics defined above assume direct correspondence between the intensity values of the images to be registered. This assumption does not always hold

even within the same modality, especially for MR images. For example, the T1- and T2-weighted images in Figure 4.4 (a) and (b) cannot be aligned using MSE or MAE, since the same tissue assumes different values in the two images. The metrics can only be used for registering a pair of the same modality. For example, Figure 4.4 shows the difference of the T1 image and a version of itself rotated and translated by 2° and 2 mm. Two perfectly aligned images would yield a zero-valued difference.

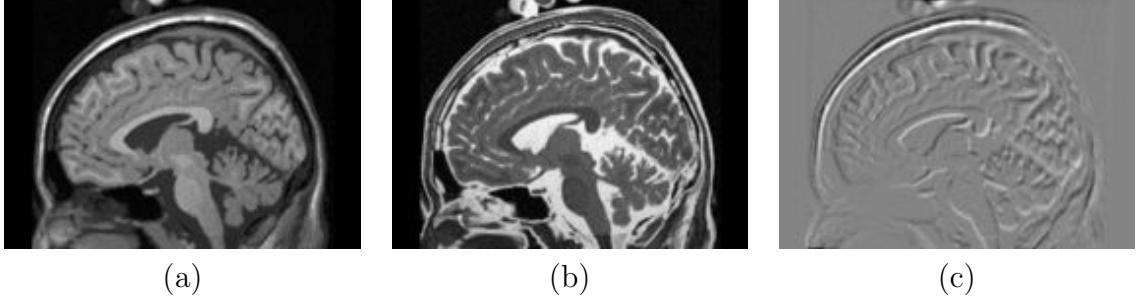


Figure 4.4: T1-weighted image (a), T2-weighted image (b); subtraction (c) of the T1 image and a rigidly transformed version of itself

The normalized cross-correlation metric is used when a linear relationship exists between fixed and moving image intensities—a less restrictive assumption. The metric increases with better image correspondence:

$$\text{NCC}(A, B) = \frac{\sum_{\mathbf{x} \in \Omega} (A(\mathbf{x}) - \bar{A}) \cdot (B(\mathbf{x}) - \bar{B})}{\sqrt{\sum_{\mathbf{x} \in \Omega} (A(\mathbf{x}) - \bar{A})^2 \cdot \sum_{\mathbf{x} \in \Omega} (B(\mathbf{x}) - \bar{B})^2}}, \quad (4.3)$$

where $\bar{A}$ and $\bar{B}$ are the mean image intensities.

Gradient-based metrics assume that anatomical structures between the images have common boundaries, though they may exhibit different intensity and contrast characteristics. The normalized gradient field (NGF) metric, introduced by Haber and Modersitzki, is based on the assumption that intensity changes spatially co-occur in similar images [100]. The metric is constructed from normalized image gradients:

$$\mathbf{n}(A, \mathbf{x}) = \frac{\nabla A(\mathbf{x})}{\sqrt{\|\nabla A(\mathbf{x})\|^2 + \epsilon^2}}, \quad (4.4)$$

where ϵ is a regularization parameter that controls the metric's sensitivity to edges. We set it to be proportional to the estimated image noise level, which is computed as the background intensity standard deviation. Our implementation uses two-point central differences to compute the gradients.

The normalized gradient field metric accounts for structural boundaries with the same or opposing directions by maximizing the square of the normalized gradient inner products:

$$\text{NGF}(A, B) = \frac{1}{N} \sum_{\mathbf{x} \in \Omega} \langle \mathbf{n}(A, \mathbf{x}), \mathbf{n}(B, \mathbf{x}) \rangle^2. \quad (4.5)$$

Figure 4.5 shows gradient images of the T1 and T2 MRIs of Figure 4.4 above. We see that correlation of the T1 and T2 images’ normalized gradients makes sense for registration, while correlation of their intensities does not.

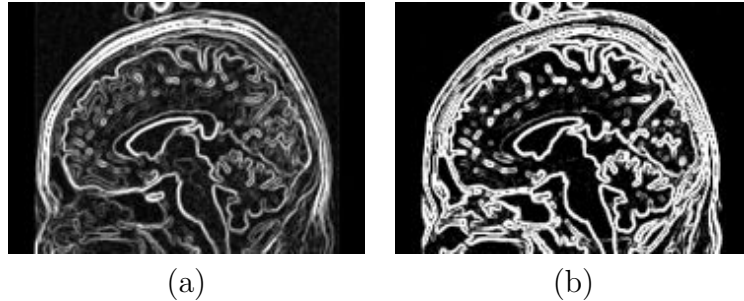


Figure 4.5: Gradient images of the T1 (a) and T2 (b) MRIs

Haber and Modersitzki describe NGF as an alternative to mutual information that is easier to compute, more straightforward to implement, and more readily suitable to numerical optimization due to greater convexity over the transformation parameter space.

Metric Image Rendering

We formulate metric evaluation as a rendering operation. As we have discussed, the fixed and moving images are textured onto view-aligned quads using the graphics hardware’s fast interpolation capability. The metric evaluation takes place when we render this textured geometry using a custom fragment shader program that replaces default graphics pipeline functionality. Figure 4.6 (modified with permission from Chan [80]) depicts the slice-by-slice metric computation process. The process is summarized below.

1. *Texture and Geometry Initialization.* The fixed and transformed moving image cross-sectional slices are textured-mapped onto a stack of quadrilaterals aligned parallel to the view plane.
2. *Rendering Metric Images.* Each quad is rendered using a custom fragment shader that outputs an *intermediate* metric image. The shader is able to access and perform arithmetic operations on arbitrary samples of the input fixed and moving textures.

For the MSE, MAE, and NGF similarity metrics (Eqs. 4.1, 4.2, and 4.5), the output fragment intensity is set to the metric summand. For the NCC metric (Eq. 4.3), the three different summands are output to the fragment’s red, green, and blue components.

3. *Summation of Metric Images.* As they are rendered, additive alpha blending automatically sums the intermediate metric images along the slice axis into a composited metric image.

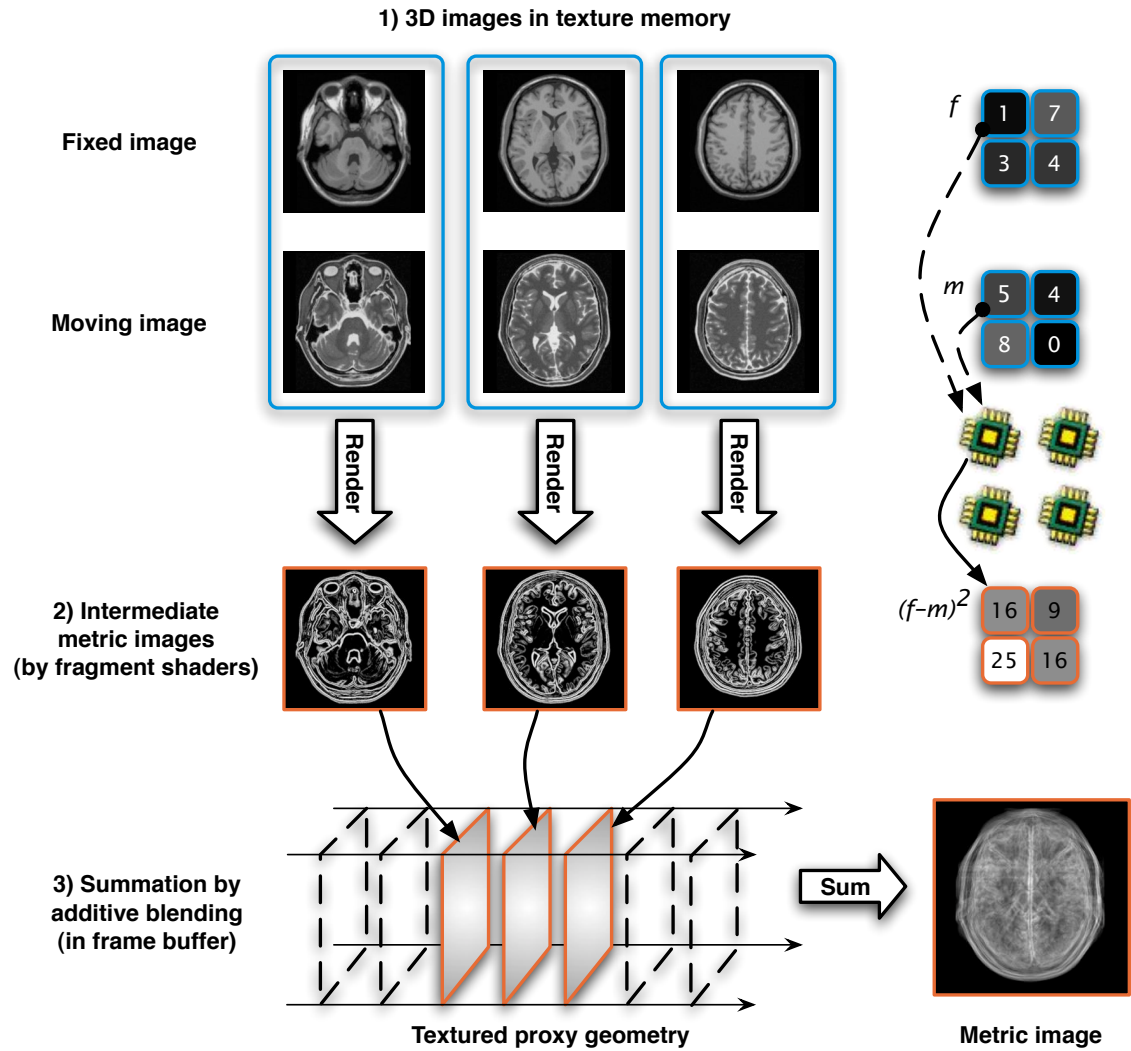


Figure 4.6: Computation of difference- and correlation-based metrics using the rendering pipeline

Following these steps, there remains to compute a single metric value by averaging the intensities of the composited metric image from step 3 above.

Listing 4.2 shows the setup of parameters in OpenGL prior to rendering. An orthographic view projection is used (lines 0-2) to ensure that all metric slices have the same dimensions in the frame buffer. The clipping volume is a cube with normalized device coordinates $[-1, 1]^3$. Depth testing is disabled (line 4) so that all fragments get rendered to the screen without occlusion. Frame buffer blending is enabled, with the blend function set to additive compositing (lines 5-7). We always render to a 32-bit floating-point frame buffer in order to preserve precision and to avoid overflow during blending.

```

0  glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
2  glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0); // orthographic projection

4  glDisable(GL_DEPTH_TEST);
   glEnable(GL_BLEND);
6  glBlendFunc(GL_ONE, GL_ONE); // blending coefficients
   glBlendEquation(GL_FUNC_ADD); // additive blending

```

Listing 4.2: Projection and frame buffer setup for registration rendering

As an example, Listing 4.3 shows the GLSL fragment shader program used to compute the metric images for the normalized cross-correlation metric of Equation 4.3. First, the fixed (F) and moving (M) image textures and their mean values ($\bar{F}$, $\bar{M}$) are declared (lines 0-1). The means are precomputed, as they do not change.

Suppose that $\mathbf{T} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is the current moving image transformation. At each voxel $\mathbf{x} \in \mathbb{R}^3$, the fixed image intensity $F(\mathbf{x})$ and moving intensity $M(\mathbf{T}(\mathbf{x}))$ are retrieved by sampling the textures at the original and transformed 3D textures coordinates (lines 5-6). The output fragment's colour red, green, and blue intensities are set to the metric summands (line 11): $(F(\mathbf{x}) - \bar{F}) \cdot (M(\mathbf{T}(\mathbf{x})) - \bar{M})$, $(F(\mathbf{x}) - \bar{F})^2$, and $(M(\mathbf{T}(\mathbf{x})) - \bar{M})^2$. The alpha value (set to 1.0) is inconsequential.

The other metrics are computed similarly. For example, changing line 11 in the above listing to `gl_FragColor = pow(fixed - moving, 2.0)` yields a shader program that computes the MSE metric instead of the NCC metric.

The code used to render the proxy geometry is given in Listing 4.4. Each loop iteration creates a quad over the $[-1, 1]^2$ view plane by defining its vertices with the function `glVertex3f`. Coordinates into the fixed (`GL_TEXTURE0`) and moving (`GL_TEXTURE1`) textures are defined using `glMultiTexCoord3f`. Texture coordinates are normalized to the range $[0, 1]^3$, with sampling done at voxel centres in the z (slice) direction.

By default, we render directly into texture memory rather than to the on-screen frame buffer. This is done by binding a 2D texture to the frame buffer using OpenGL's *frame buffer object* specification extension [86]. We do this for two reasons. First, it avoids copying from the screen into texture memory for subsequent computation of the final metric. Second, performance is improved by not rendering to the display hardware. However, the user is given the option to render to the display permitting

```

0 uniform sampler3D fixedTexture, movingTexture;
  uniform float fixedMean, movingMean;

2

4 void main(void)
  {
    vec4 fixed = texture3D(fixedTexture, gl_TexCoord[0].stp);
    vec4 moving = texture3D(movingTexture, gl_TexCoord[1].stp);

6     float f = fixed.r - fixedMean;
    float m = moving.r - movingMean;

8     gl_FragColor = vec4(f*m, f*f, m*m, 1.0); // NCC metric summands

10
12 }

```

Listing 4.3: Computing the normalized cross-correlation metric images using a GLSL fragment shader program

```

0 glBegin(GL_QUADS);
  // submit one quad per image slice
2   for (int i = 0; i < numSlices; i++)
    {
4       float z = (0.5 + i) / numSlices; // slice depth

6       glMultiTexCoord3f(GL_TEXTURE0, 0.0, 0.0, z);
       glMultiTexCoord3f(GL_TEXTURE1, 0.0, 0.0, z);
       glVertex3f(-1.0, -1.0, z); // 1st vertex

8       glMultiTexCoord3f(GL_TEXTURE0, 1.0, 0.0, z);
       glMultiTexCoord3f(GL_TEXTURE1, 1.0, 0.0, z);
       glVertex3f( 1.0, -1.0, z); // 2nd vertex

10      glMultiTexCoord3f(GL_TEXTURE0, 1.0, 1.0, z);
       glMultiTexCoord3f(GL_TEXTURE1, 1.0, 1.0, z);
       glVertex3f( 1.0,  1.0, z); // 3rd vertex

12      glMultiTexCoord3f(GL_TEXTURE0, 0.0, 1.0, z);
       glMultiTexCoord3f(GL_TEXTURE1, 0.0, 1.0, z);
       glVertex3f(-1.0, 1.0, z); // 4th vertex

14      }
16      glEnd();
18
20
22

```

Listing 4.4: Applying transformations using texture coordinates and rendering the fixed and moving images textures in OpenGL

visualization of intermediate metric images during registration.

Many fragment shaders compute the metric image in parallel, as illustrated schematically in Figure 4.6. In practice, however, there are more fragments to be shaded for each quad than there are available shader processors. The shaders therefore execute in parallel on sub-blocks of the images. As an example, a typical medical image may have $256^2 = 65,536$ pixels per slice, whereas the NVIDIA GeForce 8800 GPU has 128 shaders.

Metric Value Accumulation

At this point, the metric image has been computed and resides on the GPU. This corresponds with the last step in Figure 4.6. There remains to average the pixels of this image in order to compute the final similarity metric value. We do this using a sequence of parallel downsampling rendering passes [101] on the GPU, as shown in Figure 4.7.

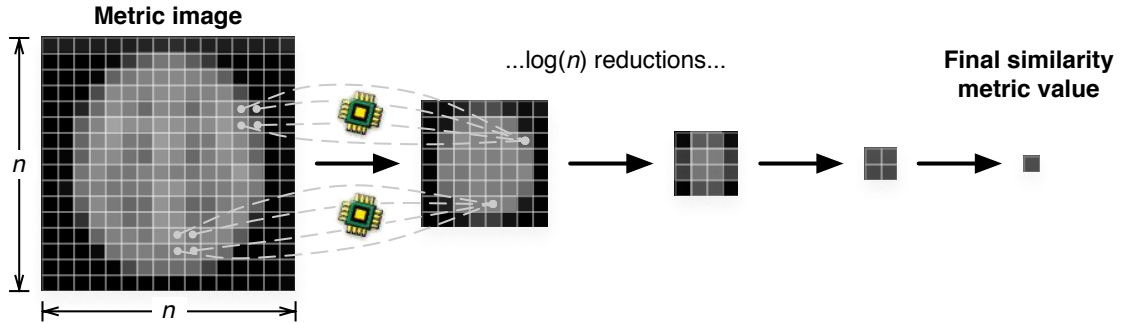


Figure 4.7: Parallel reduction to accumulate the final metric value by shader downsampling passes

If the metric image has dimensions $n \times n$, then $\log n$ downsampling passes in a fragment shader yield the final similarity metric. The final metric value is downloaded from the GPU to the CPU optimizer using the command `glReadPixels`.

4.1.5 Mutual Information Similarity Metric

In this section, we describe our GPU implementation of mutual information (Eq. 3.3) and its normalized variant, NMI (Eq. 3.4). These are the most widely used similarity metrics for automatic registration of multi-modal medical images [13, 84, 102], being among the most accurate and robust metrics for retrospective studies of CT, MR, and PET images of the brain [93, 103].

Mutual information measures the mutual dependence between random variables, as described in section 3.3.2. It is based on the information-theoretic quantity of entropy, which is the expected information content of a probabilistic event. As such, computation of mutual information requires estimation of the marginal and joint PDFs of images. This is done using the marginal and joint image histograms.

Mutual information is computed using fundamentally different GPU methods than the difference- and correlation-based metrics (MSE, MAE, NCC, and NGF) of section 4.1.4. The most computationally involved component of mutual information evaluation is construction of the joint image histogram, since this requires iteration through the image volumes and non-sequential access to shared memory. We compute image histograms entirely on the GPU, thereby greatly accelerating registration.

Accelerated Histogram Computation

We extend a method described by Scheuermann and Hensley [88] for 1D histogram computation on the GPU to the case of joint (2D) histograms. The method permits creation of histograms with arbitrary size in a single rendering pass. It uses a recent extension of NVIDIA graphics hardware functionality called *vertex texture fetch*, which allows the vertex shader to read from textures in video memory [33]. The vertex shader then uses these fetched values to *scatter* vertices to arbitrary output locations in the frame buffer.

Figure 4.8 illustrates the method used to generate a 1D histogram of an image. Corresponding GLSL vertex and fragment shaders are given in Listings 4.5 and 4.6. In this method, the histogram bins are stored as pixel intensities in a row of the frame buffer.

The method is initialized with a vertex array. One vertex is generated for each image sample, and the vertex locations are set to the image sampling coordinates (Fig. 4.8, step 1). Next, the vertex shader fetches the image intensity at each vertex position (Fig. 4.8, step 2; Listing 4.5, line 4). It sets the vertex output position to equal the fetched image intensity (Fig. 4.8, step 3; Listing 4.5, line 5). The histogram bin counts are incremented in the frame buffer by rendering the vertices as point primitives with a colour intensity of 1.0 (Fig. 4.8, step 4; Listing 4.6, line 2). Additive blending is enabled.

Joint Histogram Rendering

Our method for computing joint histograms is given in detail below. It is analogous to the method illustrated in Figure 4.8, except that two volumetric images are used as input and the histogram is two-dimensional. Similar to the algorithm presented in section 4.1.4, histograms of 3D images are computed on a *per-slice* basis, then summed in the frame buffer for all slices using additive blending.

1. *Geometry Initialization.* One vertex is created for each sample in a slice of the fixed image. The vertices are stored in an array on the GPU called a *vertex buffer object* [104]. The vertices will be rendered as point primitives, with each vertex defining one point.

The positions of the vertices are initialized to the fixed image sampling coordinates (i, j, k) , where k is the depth of the current slice being processed. (The value of k is passed to the vertex shader as a uniform variable.) We refer to each vertex by the coordinates $\mathbf{x} = (i, j, k)$. The fixed image intensities $F(\mathbf{x})$ are stored on the GPU as an array of vertex attributes.

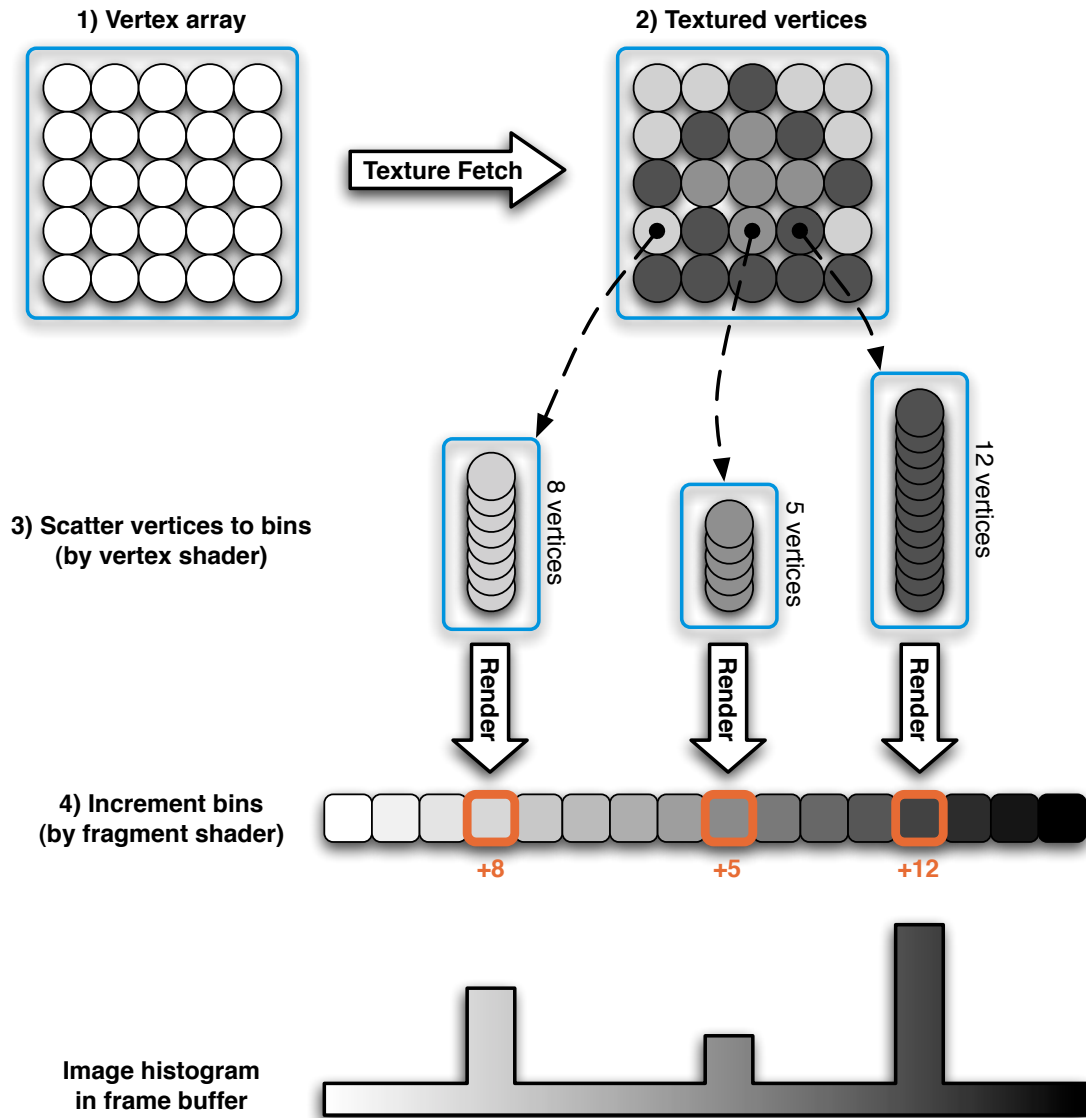


Figure 4.8: Computation of 1D image histograms on the using vertex scattering in the rendering pipeline

```

0 uniform sampler3D imageTexture;
2 void main(void)
{
4     float intensity = texture3D(imageTexture, gl_Vertex.xy).r;
    gl_Position = vec4(intensity, 0.0, 0.0, 1.0);
6 }

```

Listing 4.5: Vertex shader using *vertex scattering* to generate an image histogram

```

0 void main(void)
{
2     gl_FragColor = vec4(1.0);
}

```

Listing 4.6: Trivial fragment shader for incrementing histogram bins

2. *Vertex Processing.* Suppose that $\mathbf{T}$ is the current affine transformation matrix. Given an input vertex $\mathbf{x}$, a custom vertex shader fetches the corresponding moving texture intensity $M(\mathbf{T}(\mathbf{x}))$. The resulting vertex is said to be *textured*. Processing of the vertex is halted if the transformed coordinates $\mathbf{T}(\mathbf{x})$ are outside of the moving image domain. This ensures that the histogram is computed only using intensities in the overlap of the fixed and moving images.

The output position of the vertex is set to 2D coordinates $(F(\mathbf{x}), M(\mathbf{T}(\mathbf{x})))$, normalized to the range $[-1, 1] \times [-1, 1]$. These coordinates are equal to the fixed and moving image intensities.

3. *Fragment Processing.* Following rasterization, a custom fragment shader sets the output intensity of each fragment to 1.0. With additive blending enabled, this results in bin $(F(\mathbf{x}), M(\mathbf{T}(\mathbf{x})))$ being incremented every time that a vertex is scattered into it during rendering.
4. *Rendering.* The vertex array corresponding to slice k is rendered. Since vertices $\mathbf{x}$ are rendered to point primitives, the resulting “image” consists of fragments at positions $(F(\mathbf{x}), M(\mathbf{T}(\mathbf{x})))$. In other words, the vertex shader *scatters* vertices into their joint histogram bin locations.

To prevent bin saturation, the joint histogram is rendered to a 32-bit floating-point buffer. Vertex array rendering initiated using the OpenGL command `glDrawArrays`. Our metric computations use joint histograms with 256×256 bins. This size is common among registration implementations and has been found empirically to be a good choice for most cases [105].

Histogram Rendering Optimizations

Several optimizations employed in our implementation of the method above are worth highlighting. First, we store all vertices in a buffer on the GPU (step 1 above). This eliminates costly transfers of vertices from CPU to GPU on each rendering pass. Also,

we render vertices for one slice of the image at a time, as we found it prohibitive to render an entire volume’s worth of vertices in one pass. Since each image slice is identical in terms of sample spacing, we render the same vertex array on each pass, thereby eliminating redundancy. The only variable updated between rendering passes is a uniform corresponding to the slice number.

Because the fixed image values remain constant, they are stored in a GPU vertex buffer for fast access in the vertex shader. Prior to vertex processing in step 2, each vertex is assigned its corresponding fixed image value as an attribute variable. This increases vertex throughput, since the vertex shader only needs to fetch the moving image value.¹ Accessing vertex attributes is reported to incur less overhead than texture reads [33].

Medical images often contain a large percentage of background voxels that are located outside of the subject’s anatomy. It is common for up to 25% of pixels to belong to the background. In MR and PET, these voxels usually have zero intensity, since they do not contribute signal to the image. In CT, background voxels have intensities near -1000 Hounsfield units, corresponding to air.

Thus, another optimization that we implement is to discard vertices destined for histogram bin (F_b, M_b) , where F_b and M_b are the fixed and moving image background intensities. For MR to MR registration, we therefore usually discard vertices destined for bin $(0, 0)$. An *occlusion query* [106] is issued following rendering in order to determine the number of fragments that were discarded, and this value is explicitly copied into bin (F_b, M_b) . (In traditional graphics applications, occlusion queries are used to determine the objects obstructed from view in a scene, so as not to render them.) By eliminating the need to render a large number of vertices, this optimization significantly reduces load on the GPU without affecting the resulting histogram, yielding time savings of up to 40% in our tests on typical medical images.

Entropy Calculation

The two marginal image histograms required for MI and NMI are computed by integrating the 2D joint histogram along the fixed and moving intensity axes. This is done in a fragment shader program that uses parallel reduction along one dimension. (Figure 4.7 demonstrates parallel reduction along two dimensions.) Since our joint histograms have 256×256 bins, eight 1D downsampling passes are required.

The marginal and joint PDFs of Equations 3.1 and 3.2 are estimated by normalizing the histogram bins by the total histogram bin counts. Next, the summands of the entropies are calculated, then accumulated using a sequence of downsampling passes to generate the marginal and joint entropy values. The downsampling passes are also executed using the parallel reduction technique demonstrated in Figure 4.7.

Partial Volume Interpolation

By default, the moving image intensity $M(\mathbf{T}(\mathbf{x}))$ used to update the histogram is found using trilinear interpolation (see section 4.1.3). As an alternative, we also im-

¹At the time of programming, this *vertex texture fetch* feature was available only on NVIDIA hardware. On ATI hardware, a related feature called *render to vertex buffer* is used to place image intensities directly into the vertex shader as vertex attributes.

plement the method of partial volume interpolation (PVI), which has been shown to increase the robustness of NMI objective function optimization [84]. The PVI method uses fractional weights to update the joint histogram, establishing a gradual change in the bin values as the moving image is transformed. This removes undesired local optima in the similarity metric function. Compared to standard trilinear interpolation, PVI yields superior registration accuracy [107, 108].

Suppose that we are updating the histogram with intensity pair $(F(\mathbf{x}), M(\mathbf{T}(\mathbf{x})))$, where $\mathbf{T}(\mathbf{x})$ are the transformed image coordinates. Let $\mathbf{y}_i$ ($0 \leq i \leq 7$) be the eight sample coordinates that neighbour $\mathbf{T}(\mathbf{x})$. Recall that the trilinearly interpolated estimate of $M(\mathbf{T}(\mathbf{x}))$ is a weighted average of the neighbouring intensities:

$$\tilde{M}(\mathbf{T}(\mathbf{x})) = \sum_i w_i \cdot M(\mathbf{y}_i), \quad (4.6)$$

where the weights w_i are computed as normalized rectilinear volumes between the sample locations $\mathbf{y}_i$ and $\mathbf{T}(\mathbf{x})$. Figure 4.9 illustrates the weights as areas for the 2D case corresponding to bilinear interpolation.

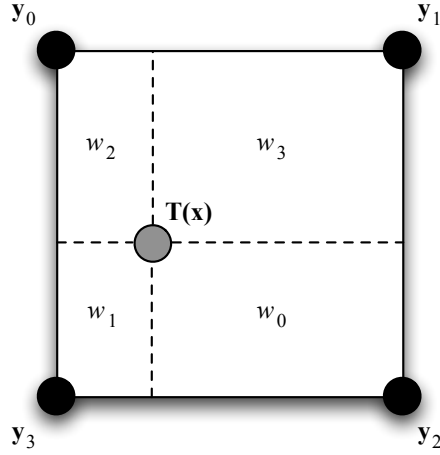


Figure 4.9: Partial volume interpolation weights (2D example)

Rather than updating one bin, as in the standard approach described in section 4.1.5, the PVI method updates the eight histogram bins that correspond to the intensity pairs $(F(\mathbf{x}), w_i M(\mathbf{y}_i))$, $0 \leq i \leq 7$.

Our implementation of PVI updates the joint histogram using eight rendering passes over all image samples. The first pass updates bin $(F(\mathbf{x}), M(\mathbf{y}_0))$ by the fractional amount w_0 , the second pass updates bin $(F(\mathbf{x}), M(\mathbf{y}_1))$ by w_1 , and so on. Thus, we scatter each vertex eight times, computing the weights explicitly in the vertex shader and sending them to the fragment shader as *varying* variables.

4.1.6 Metric Function Optimization

The goal of optimization in registration is to search for the transformation parameters that achieve the best possible similarity match between the fixed and moving images. The optimization component of the registration framework is implemented on the CPU, since it does not involve parallel operations on large data.

The objective of intensity-based registration is expressed generally as

$$\mathbf{p}^* = \arg \min [S(F(\mathbf{x}), M(\mathbf{T}_{\mathbf{p}}(\mathbf{x})))], \quad (4.7)$$

where F and M are the fixed and moving images, S is the similarity metric function, and $\mathbf{T}_{\mathbf{p}}(\mathbf{x})$ is the transformation with parameters $\mathbf{p}$ applied to the fixed image coordinates $\mathbf{x}$. Since our convention is to minimize the metric function, we multiply the NCC, NGF, and NMI metrics by -1 . As described in the next section, we use a multi-resolution optimization strategy in order to increase the likelihood and rate of convergence to the global optimum.

Full affine transformations have 12 independent parameters to optimize simultaneously: three for each of rotation, translation, scaling, and shear. Rigid-body and scaling transformations have six and nine parameters, respectively. We normalize all parameters in objective space such that a unit step along any parameter axis results in approximately the same displacement of the image in physical space [102]. (Displacements associated with rotation, scaling, and shearing are estimated by the movement of the image corners.)

We implement both *Powell's method* and the *Nelder-Mead simplex method* for optimization [9]. These methods perform multidirectional optimization without evaluating gradients. Both have been extensively applied to registration on the CPU and GPU [82]. A comprehensive study by Maes et al. found that compared to several common optimization strategies, Powell's method often yields the best results for multi-modal image registration [109]. They also found that the simplex method is among the fastest of the strategies studied. They recommended using the simplex method for multi-resolution optimization and Powell's method for single level optimization.

For an n -dimensional objective space, Powell's method repeatedly minimizes along a set of n directions in turn. It uses one-dimensional line minimizations, initializing each search with the minimum found from the last direction. Powell's method ensures conjugacy of the direction set by replacing the direction of largest functional decrease with the vector between the start and end points after n minimizations.

The Nelder-Mead simplex method considers all n degrees of freedom simultaneously by updating the $n+1$ vertices of a non-degenerate simplex. The simplex follows the downhill gradient of the objective function using amoeboid-like movements until it reaches a minimum. The simplex deforms using geometric reflection, expansion, and contraction steps.

The convergence criteria for the two minimization methods are set to be as similar as possible. We stop the optimizer if $\frac{|f_m - f|}{(|f_m| + |f|)/2} \leq f_{\text{tol}}$, where f denotes the current minimum function value, $\bar{f}_m$ denotes the moving average of the last m (generally set

to 10) smallest function values, and f_{tol} is a specified function tolerance. The Nelder-Mead optimizer has two additional (optional) convergence criteria. Convergence can be declared if the simplex volume is below a threshold, or if the relative difference between the highest and lowest simplex vertices (in terms of function value) is below a threshold. A hard limit is set for the maximum number of function evaluations in both the Powell and Nelder-Mead optimizers.

Powell and Nelder-Mead are categorized as local optimization strategies [9]. This means that they search for local minima within a certain capture range of their starting point. Global optimization methods, such as *dividing rectangles* [64] and some genetic algorithms, search for the global minimum within a given parameter range. We employ a hierarchical search strategy in order to increase the optimization capture range.

4.1.7 Hierarchical Search

Hierarchical optimization strategies are commonly used in automated registration [12]. Upon initialization, the input fixed and moving images are downsampled and smoothed to generate multi-resolution image pyramids. Registration progresses from the lowest resolution image pair of the pyramid to successively higher resolution levels. Registration iterations at lower resolutions take less time, since the number of computations per iteration scales linearly with image dimension. Larger registration mismatches are recovered initially using coarser strides through search space, whereas finer details are matched as they are introduced at higher resolutions using smaller strides.

This strategy helps avoid convergence to local optima in the objective function and increases the likelihood of a global optimum match, thereby improving registration accuracy. It also accelerates optimizer convergence and increases the parameter search capture range, since relatively larger image mismatches tend to be recovered at lower resolutions [109]. Multi-resolution optimization thus means fewer iterations are performed the finest pyramid level, as compared to a single-resolution strategy. We empirically found that using two to four resolution levels works best for most clinical registrations cases.

A series of fragment shaders generate the pyramid levels by recursively blurring and downsampling the 3D textures. Blurring is done separately along the spatial dimensions using a 1D Gaussian kernel. We generally use a Gaussian with standard deviation 0.5 and a width of five pixels: (0.06, 0.24, 0.40, 0.24, 0.06). Filtering in the slice direction is not performed for images with relatively large slice spacing compared to in-plane spacing [109]. A downsampling factor of two is used between pyramid levels. This scheme is depicted in Figure 4.10.

The optimizer is initialized with the identity transformation at the lowest pyramid level. The registration parameters estimated at this resolution are used as the starting point for optimization at the next highest level. This process is repeated until the final registration is computed at the highest pyramid level. Other initialization choices have been reported in the literature, such as matching image centroids and principal component analysis to estimate initial translation and rotation. However, these methods often fail for images acquired with different fields of view [22].

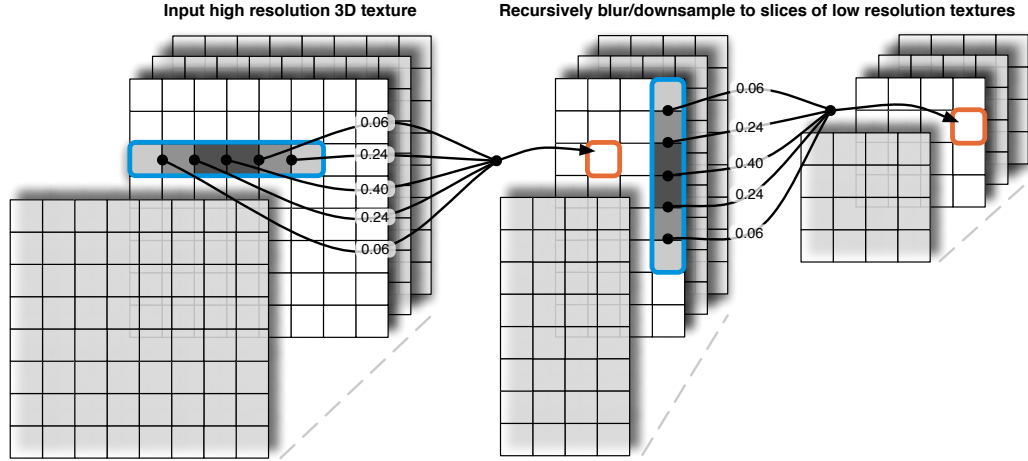


Figure 4.10: Recursive Gaussian blurring and downsampling scheme to generate image pyramids

4.2 GPU-Accelerated Deformable Image Registration

In this section, we describe our GPU implementation of the “Demons” method for deformable image registration. The method was proposed by Thirion in 1996 and has since been widely used in medical imaging [20]. It is based on the concept of *optical flow*, which is used to determine apparent velocities of intensity patterns between temporal sequences of images [110].

4.2.1 Optical Flow

Optical flow is based on the assumption that the intensity of a point on an object does not change as the object moves over time. Let $I(\mathbf{x}, t)$ denote the intensity of a sequence of images that vary over time t and have spatial coordinates $\mathbf{x}$. Suppose that the trajectory of a point on an object is given by $\mathbf{p}(t)$. As the object moves, the intensity $I(\mathbf{p}(t), t)$ is assumed to remain constant with time: $dI(\mathbf{p}(t), t)/dt = 0$. Expanding this differential equation yields the optical flow constraint:

$$\nabla I(\mathbf{p}, t) \cdot \dot{\mathbf{p}} + \frac{\partial I(\mathbf{p}, t)}{\partial t} = 0, \quad (4.8)$$

where $\nabla I(\mathbf{p}, t)$ is the image’s spatial gradient at time t .

This is an under-constrained linear equation in the object’s velocity vector $\dot{\mathbf{p}}(t)$. In Demons, the orthogonal projection of $\dot{\mathbf{p}}(t)$ on $\nabla I(\mathbf{p}, t)$ is determined²:

$$\dot{\mathbf{p}}(t) = -\frac{\partial I(\mathbf{p}, t)/\partial t}{\|\nabla I(\mathbf{p}, t)\|^2} \nabla I(\mathbf{p}, t). \quad (4.9)$$

²Constraining the velocity to lie along the gradient vector is equivalent to constraining the velocity’s magnitude to be as small as possible.

The Demons method approximates the time derivative in the numerator as the difference of the fixed and moving image intensities. This assumes small object displacements between the images, which may not hold in general. For this reason, Demons updates the velocity vector estimate iteratively.

4.2.2 Demons Update Iteration

As we have discussed, a registration algorithm is primarily defined by a transformation model and an image similarity metric function. The Demons method uses dense free-form deformations, where the transformation is specified independently at each 3D coordinate in the image domain by $\mathbf{T}(\mathbf{x}) = \mathbf{x} + \mathbf{d}(\mathbf{x})$. The vector function $\mathbf{d} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ is called the *displacement field*.

It can be shown that the Demons method minimizes the sum of squared intensity differences between the fixed (F) and moving (M) images [21]: $\int (F(\mathbf{x}) - M(\mathbf{T}(\mathbf{x})))^2 d\mathbf{x}$. However, this metric is not optimized explicitly as was the case for affine registration. Instead, *Demons uses the optical flow equation to iteratively solve for the displacement field*.

Let the displacement field estimate following the i^{th} iteration be $\mathbf{d}_i$. The incremental update to the field at position $\mathbf{x}$ is driven by a force that is parallel to the fixed image gradient:

$$\delta \mathbf{d}_i(\mathbf{x}) = -\frac{m - f}{\|\nabla f\|^2 + \alpha^2(m - f)^2} \nabla f, \quad (4.10)$$

where $f = F(\mathbf{x})$ and $m = M(\mathbf{x} + \mathbf{d}_i(\mathbf{x}))$ is the iteratively deformed moving image. This update is analogous to Equation 4.9 with the exception of the additional term $\alpha^2(m - f)^2$ in the denominator, which is needed to stabilize the update for small magnitude gradients. The parameter α adjusts the update “force” strength. Smaller values of α can be used initially to recover large deformations [21].

Starting with an initial displacement field $\mathbf{d}_0(\mathbf{x}) = \mathbf{0}$, $\forall \mathbf{x}$, the displacement estimate at iteration $i + 1$ is given by

$$\mathbf{d}_{i+1}(\mathbf{x}) = G_\sigma \otimes (\mathbf{d}_i(\mathbf{x}) + \delta \mathbf{d}_i(\mathbf{x})), \quad (4.11)$$

where G_σ is a Gaussian of standard deviation σ that is convolved with the total displacement field at each iteration. Smoothing the displacement field in this manner roughly simulates an elastic transformation model.³ The standard deviation is an adjustable parameter that suppresses noise and controls smoothness of the transformation.

One pitfall of free-form deformations is that there is no intrinsic regularization of the fields. It is theoretically possible, for example, to generate a discontinuous transformation that minimizes the similarity metric by mapping all points in M to points of the same intensity in F [111]. A sufficient degree of smoothing by G_σ guarantees that the transformation is bijective, though not necessarily diffeomorphic.

³Smoothing only the update term ($\delta \mathbf{d}_i(\mathbf{x})$) instead results in a variation of Demons that roughly simulates a viscous fluid model.

The incremental displacement updates in Equation 4.11 become progressively smaller as this scheme converges. The degree of smoothing affects the convergence rate and the level of detail being matched. In general, larger values of σ slow the rate of convergence and tend to capture larger details. In our implementation, we quantify the level of convergence at each iteration using the normalized cross-correlation metric (Eq. 4.3). Image gradients are computed using a two-point numerical approximation of the derivative.

An assumption in deriving the Demons update force in Equation 4.10 is that deformations should be reasonably small [112]. Since this is not always true in clinical scenarios, we use a coarse-to-fine multi-resolution scheme, as we did for affine registration (see section 4.1.7). The images are downsampled by factors of two to generate the image pyramids. Following registration convergence at a low resolution level, the displacement field estimate is up-sampled and used to initialize the field at the next higher resolution level.

4.2.3 Non-linear Image Transformation

The displacement field updates in Equation 4.10 are independent for each voxel. Thus, we parallelize these computations in a fragment shader that operates over the image domain. The displacement field is represented on the GPU as an RGB texture, with its x , y , and z components stored in the red, green, and blue channels.

Although the fields are inherently 3D, they are stored in 2D textures with slices laid out as flat tiles. This is because not all standard hardware currently supports rendering to 3D textures in GLSL. The textures have either 16 or 32 bits per channel, depending on the amount of available GPU video memory. Using higher precision textures provides superior accuracy; however, their use often requires a prohibitive amount of memory to store the images and deformation fields.

We implement non-linear image transformation of the moving image in Equation 4.10 entirely on the GPU by programming the fragment processor to use *dependent texture look-ups*. In order to compute $M(\mathbf{T}(\mathbf{x}))$, the coordinates $\mathbf{x}$ are first used to index the current displacement field texture, yielding the transformed coordinates $\mathbf{T}(\mathbf{x})$. These new coordinates are then used to index the moving image texture, as depicted in Figure 4.11. The hardware is configured to use trilinear interpolation of textures.

4.2.4 Deformation Update

We use multiple rendering passes to update the displacement field estimate according to Equation 4.11. In order to avoid reading and writing the same texture buffer simultaneously, we use two different textures to store results of the previous and current iterations. We alternately feed the output of a given rendering pass as input to the next one using a technique called *texture ping-pong*, as depicted in Figure 4.12.

Suppose that the displacement field estimate $\mathbf{d}_{i-1}$ from iteration $i - 1$ is stored in texture 0. On iteration i , we read texture 0 and write the next estimate $\mathbf{d}_i$ to texture 1. The roles of the textures are then swapped: texture 1 is bound as the source and

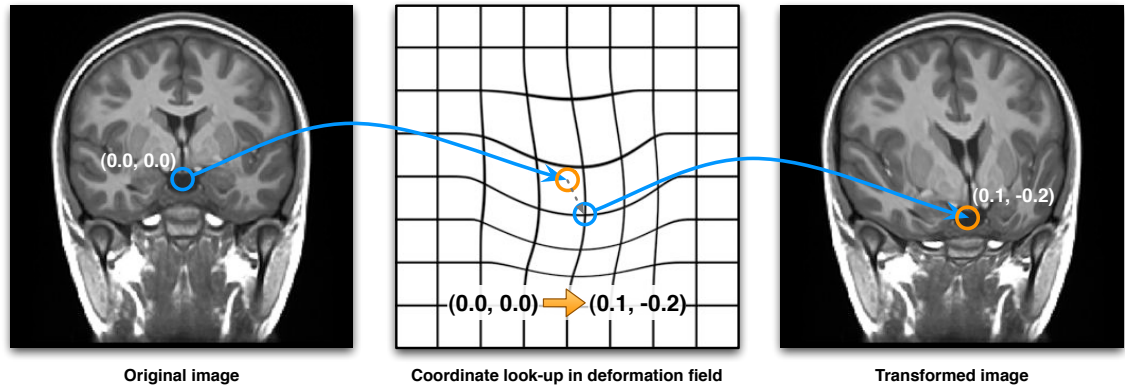


Figure 4.11: Non-linear transformation using coordinate look-up in a displacement field texture on the GPU

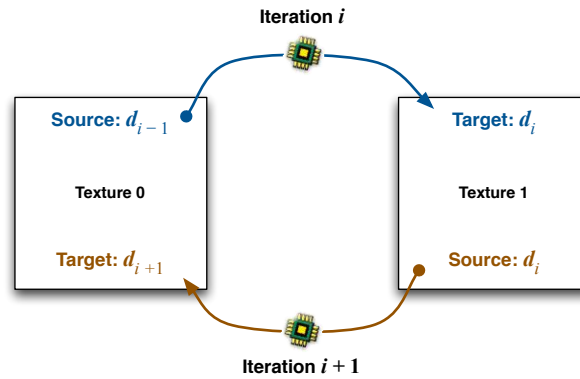


Figure 4.12: Ping-pong iterative updates of the Demons displacement fields by swapping source and render target textures

texture 0 is bound as the target. The subsequent estimate $\mathbf{d}_{i+1}$ is then written to texture 0.

We take advantage of the separability property of the Gaussian kernel when smoothing the field at each iteration. In particular, the 3D convolution operation is decomposed into a set of smoothing steps with a 1D Gaussian kernel along each image dimension, as demonstrated in Figure 4.13.

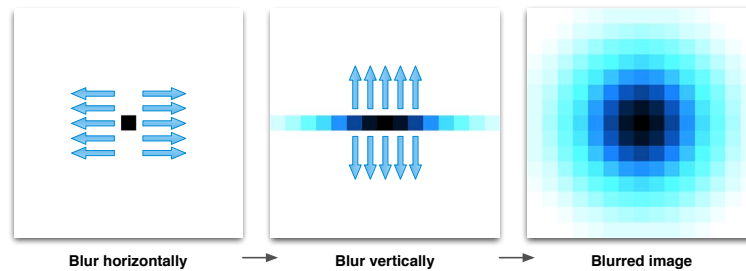


Figure 4.13: Applying a Gaussian blur using the separability of the convolution kernel

Chapter 5

Validation of GPU-Accelerated Medical Image Registration

In this chapter we report several evaluations on the speed and accuracy of our GPU registration framework. Gold standard assessment of registration accuracy is based on the correspondence of homologous anatomic features between images [113]. However, such quantitative evaluation of accuracy is often difficult to perform, since ground truth mappings between images are typically not known and cannot be determined exactly [103]. In addition, unique anatomic correspondences may not exist due to differing subject anatomy or image acquisition parameters. In deformable registration studies, there may exist multiple—equally valid—solutions that match image intensities between homologous structures [112].

Many registration validation studies are based on the identification and correlation of shared anatomy between images [114]. They are usually performed by expertly identifying anatomical point landmarks or segmenting regions of interest (ROIs) in images. Landmarks and regions are defined for structures known to be shared across individuals in the test population. Klein et al. used ROI segmentations to perform the most extensive evaluation of non-linear registration algorithms to date [113]. In their study, source and target image pairs were manually segmented into over fifty regions. Following registration, the recovered deformations were applied to the source image segmentation labels and compared to the target image segmentation labels. They evaluated registration accuracy using various measures of source and target label overlap, similarity, and distance.

Ardekani et al. compared the accuracy of registration algorithms using a method based on landmarks [115]. The landmarks were manually placed on homologous structures throughout the source image brains by an operator trained in neuroanatomy. Following registration to a common reference space, accuracy was measured by the dispersion of the warped landmarks. Hellier et al. also used landmarks to measure registration accuracy [116]. However, they focused on cortical structures, which are particularly relevant in the context of functional imaging.

Evaluation techniques based on landmarks and ROIs are often limited by the accuracy with which these features can be identified. Human judgement can be subjective and often yields results that are difficult to reproduce [114]. In addition, such evaluation cannot detect registration mismatches that fall between landmarks or within segmented regions. It is also difficult to distinguish between registration errors and true morphological variability among a study population. Other evaluation methods attempt to circumvent these problems by automatically extracting image features for comparison. For example, the overlap of classified tissue types (e.g. gray matter and white matter) is a frequently used measure of registration accuracy in neuroimaging [116]. Correlation between dense feature sets, such as the local curvatures of extracted surfaces, is also used.

An alternative registration evaluation method is to apply artificial transformations to the test images [103, 114]. Following registration, the recovered transformations

are compared to the artificial (ground truth) transformations. By varying the applied transformations, registration performance can be evaluated for different kinds of image mismatch. This gives the tester exquisite control for validation, though the simulated deformations may lack sufficient realism. Also, results may be biased by the particular class of deformations chosen.

We evaluate our registration framework in two ways. Our first set of tests consists of applying known artificial linear and non-linear transformations to images of the human head. Our second set of tests is performed on images of brain tumor patients from a clinical database. This database is noteworthy, since gold standard rigid-body transformations are known for all patients. The true alignments were obtained from a method based on the fixation of fiducial markers to the patient skulls.

The goals of this section are to evaluate the speed and accuracy of our GPU framework for routine clinical registrations of the human head. We do not, however, aim to compare our work against other registration methods or software packages. Our affine and Demons registration applications were constructed from well recognized methods that have been published and thoroughly evaluated in the literature [8, 113, 116]. It is thus not necessary to reproduce prior validation work. *Rather, we aim to show that our novel implementations of methods on the GPU can yield great improvements in terms of speed without sacrificing accuracy over equivalent methods implemented on the CPU.*

5.1 Experimental Methods

We apply artificial affine and non-linear transformations to synthetically constructed, but realistic images of the human head. These images were obtained from the Montreal Neurological Institute (MNI) Simulated Normal Brain Database [117]. In this section, we follow the methods of Chan, who also tested his GPU registration framework by applying artificial transformations to the MNI data [80]. A number of other registration methods have been evaluated using this data [10, 64, 99, 100, 118].

The test images were created by averaging the co-registered, intensity-normalized MRIs of 305 young, normal right-handed subjects in a common anatomical space [119]. We use T1- and T2-weighted test images, which are shown in Figure 5.1. These two images are in perfect anatomic alignment and were created by averaging the same set of subjects. They both have $181 \times 217 \times 181$, 1 mm isotropic voxels, 3% noise relative to the brightest tissue, and 20% simulated radio-frequency non-uniformity. The images were converted to 16-bit unsigned integer format prior to processing.

5.1.1 Artificial Affine Transformations

Affine transformations are generated by composing random 3D translations, rotations, and scalings. The maximum translation, rotation, and scaling magnitudes along the coordinate axes are limited to 30 mm, 20 degrees, and $\pm 10\%$, respectively. Ten artificially transformed versions of the T1-weighted image are created. Prior to transformation, the image is zero-padded to 256^3 voxels. This is done in order to keep

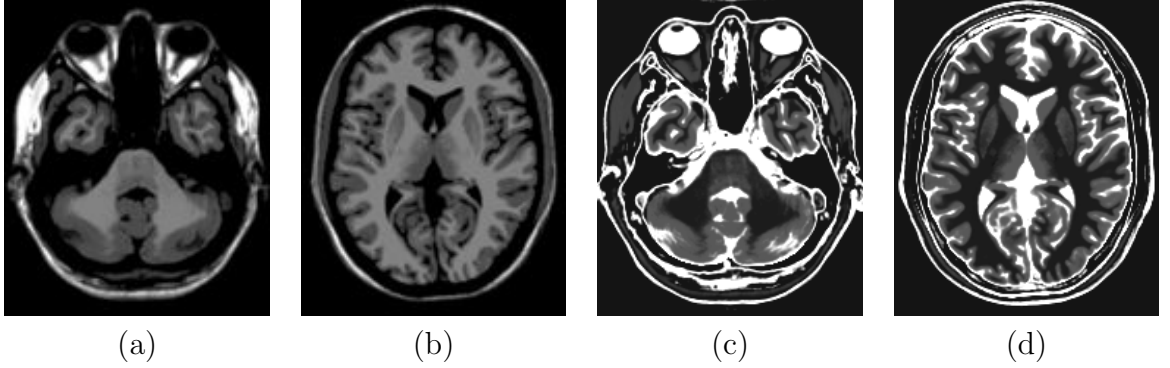


Figure 5.1: Slices of simulated T1- (a,b) and T2-weighted (c,d) MNI images

the image content in the field of view following transformation. Figure 5.2 shows two example affine transformations applied to the image.

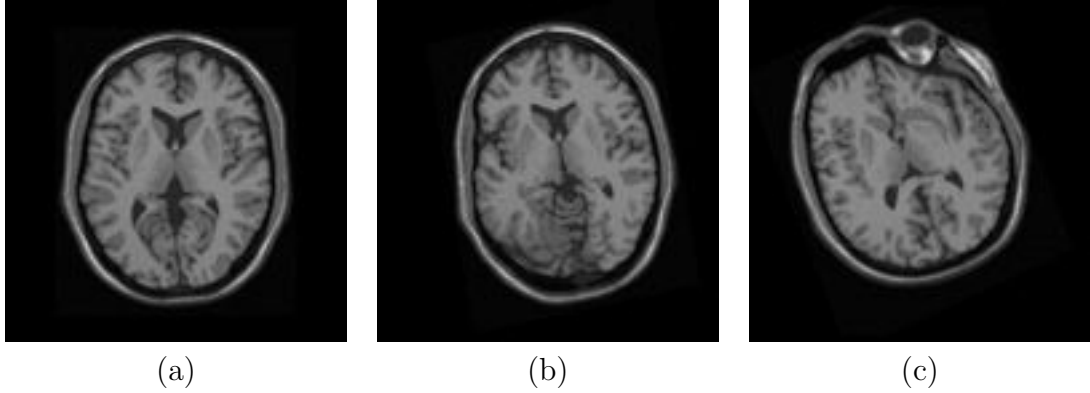


Figure 5.2: Slice of original T1-weighted MNI image before (a) and after linear transformations with small (b) and large (c) magnitude 3D translation, rotation, and scaling applied

We evaluate our registration software by attempting to recover these artificial transformations. The mono-modality MSE metric is evaluated by registering the transformed T1 image to the original T1 image. The multi-modality metrics NCC, NGF, and NMI are evaluated by registering the transformed T1 image to the original T2 image. We constrain our image transformation model to nine parameters, accounting for 3D translation, rotation, and scaling.

Trilinear interpolation is used to resample the moving images, since it provides acceptable accuracy for registration and requires fewer computations than higher-order methods [72]. The Nelder-Mead simplex optimizer is used for all experiments, with convergence criteria defined as follows: The function tolerance is set to $f_{\text{tol}} = 10^{-4}$, the minimum relative difference between the lowest and highest simplex vertices is set to 10^{-4} , and the maximum number of function evaluations per iteration is set to one thousand. A multi-resolution optimization strategy with three pyramid levels is used.

We perform all affine registrations twice. One set of runs is done using our GPU-

accelerated framework. The other set of runs is done using CPU implementations of the same registration methods. The GPU and CPU registration methods perform equivalent sets of computations given the same data, though the CPU methods were written in C++ following a traditional software-based approach. The CPU methods are not multi-threaded. The purpose of running all registrations twice is to compare timings between the GPU and CPU implementations.

We compute the accuracy of our affine registrations in two ways. First, we report mean errors between the applied and recovered translation, rotation, and scaling components of the artificial transformations. Translation error is measured as the vector length between the applied and recovered translations. Rotation and scaling errors are measured as the mean absolute difference between their respective applied and recovered components.

Second, we report the root mean square (RMS) error between the applied and recovered affine transformations over a volume of interest (VOI) in the test image. The RMS error between two affine transformations $\mathbf{T}_1$ and $\mathbf{T}_2$ from the source to the target image is defined by $\frac{1}{V} \sqrt{\int_{\mathbf{x} \in \text{VOI}} (\mathbf{T}_2 \mathbf{T}_1^{-1} - \mathbf{I})(\mathbf{x}))^2}$, where $\mathbf{I}$ is the identity transformation and V is the volume of the VOI. If we take the VOI to be a sphere of radius R and centre $\mathbf{x}_c$, then the RMS error between $\mathbf{T}_1$ and $\mathbf{T}_2$ simplifies [120]:

$$E_{\text{RMS}}^{\text{affine}}(\mathbf{T}_1, \mathbf{T}_2) = \sqrt{\frac{R^2}{5} \text{trace}(\mathbf{A}^T \mathbf{A}) + (\mathbf{t} + \mathbf{A} \mathbf{x}_c)^T (\mathbf{t} + \mathbf{A} \mathbf{x}_c)}, \quad (5.1)$$

where the 3×3 matrix $\mathbf{A}$ and the 3×1 vector $\mathbf{t}$ are components of the 4×4 matrix

$$\mathbf{T}_2 \mathbf{T}_1^{-1} - \mathbf{I} = \begin{bmatrix} \mathbf{A} & \mathbf{t} \\ 0 & 0 \end{bmatrix}. \quad (5.2)$$

We choose the VOI to be a sphere of radius 80 mm centred at the middle of the third ventricle.

We also evaluate the performance gain achieved by the specific registration components that were implemented on the GPU. To do this, we time the execution of one thousand repeated iterations of the affine *transform-resample-metric* cycle on both the GPU and CPU implementations of our application. These experiments are performed using the 16-bit MNI data as the moving and fixed images. In order to test cycle speed as a function of image size, we create versions of the images with $128^2 \times 128$, $128^2 \times 256$, $256^2 \times 128$, and $256^2 \times 256$ voxels. We apply an arbitrary affine transformation to the moving image, resample it into the space of a fixed image, then compute the similarity metric between the two. Since no registration is performed, these experiments are independent of accuracy.

5.1.2 Artificial Non-linear Deformations

We simulate non-linear transformations using several mathematical functions presented by Zagorchev and Gotashby [121]. In order to model local deformations that are diffuse through the image, we warp the test image along three orthogonal axes using modulated sine waves. Spatial coordinates $\mathbf{x} = (x \ y \ z)^T$ in the image are deformed

to coordinates $\mathbf{x}' = (x' y' z')^T$ according to

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x + A \sin f y \\ y + A \sin f x \\ z + A \sin f(x + y) \end{pmatrix}, \quad (5.3)$$

where A and f are parameters that control the wave amplitude and frequency.

Deformations of a more global and focal nature are modeled by applying radial compression (“pinch”) and expansion (“bulge”) warps to regions of the test image. These deformations are parameterized by a centre of influence $\mathbf{x}_c$, radius of influence R , and distortion strength S . The magnitude of the synthetic displacement at radial distance $r = \|\mathbf{x} - \mathbf{x}_c\|$ from the centre is

$$\rho = \begin{cases} S(r/R)(1 - r/R), & r \leq R \\ 0, & r > R \end{cases}. \quad (5.4)$$

The deformed spatial coordinates are computed as

$$\mathbf{x}' = \mathbf{x} \pm \rho \cdot \frac{\mathbf{x} - \mathbf{x}_c}{\|\mathbf{x} - \mathbf{x}_c\|}, \quad (5.5)$$

with the choice of sign dependent on whether compression or expansion is applied.

We apply these deformations to the 256³-voxel T1-weighted MNI image in order to evaluate our Demons registration method. A *locally deformed* image was generated from the original T1 image by applying a sine wave deformation according to Equation 5.3, using an amplitude of 2 mm and a spatial frequency of 0.15 mm⁻¹. A *globally deformed* image was generated according to Equation 5.5 by applying both a compression and an expansion. The compression had centre (30, 30, 10) mm, radius 60 mm, and strength 7; the expansion had centre (−40, −40, −10) mm, radius 60 mm, and strength 7. A third test image was created by summing these local and global displacement fields. Figure 5.3 shows representative slices of the three synthetically deformed images and their corresponding displacement fields.

The non-linearly deformed T1 images are registered to the original T1 image using our implementation of the Demons algorithm. Three multi-resolution levels are used for the test registrations, with one hundred iterations performed at each level. Like Thirion, we use a fixed value of $\alpha = 1.0$ to control the force strength in Equation 4.10 and we use a Gaussian kernel of standard deviation $\sigma = 1.0$ (nine pixels wide) to smooth the displacement fields in Equation 4.11 [20]. Fields are stored on the GPU in RGB textures with 16-bits of precision per channel. As with the affine registration experiments, we compare the accuracy and speed of our GPU implementation of Demons with an equivalent CPU implementation.

The quality of registration was measured by the RMS residual difference between the applied and recovered deformation fields, $\mathbf{D}_1$ and $\mathbf{D}_2$:

$$E_{\text{RMS}}^{\text{non-linear}}(\mathbf{D}_1, \mathbf{D}_2) = \sqrt{\frac{1}{|\Omega|} \sum_{\mathbf{x} \in \Omega} \|\mathbf{D}_2(\mathbf{D}_1^{-1}(\mathbf{x})) - \mathbf{x}\|^2}, \quad (5.6)$$

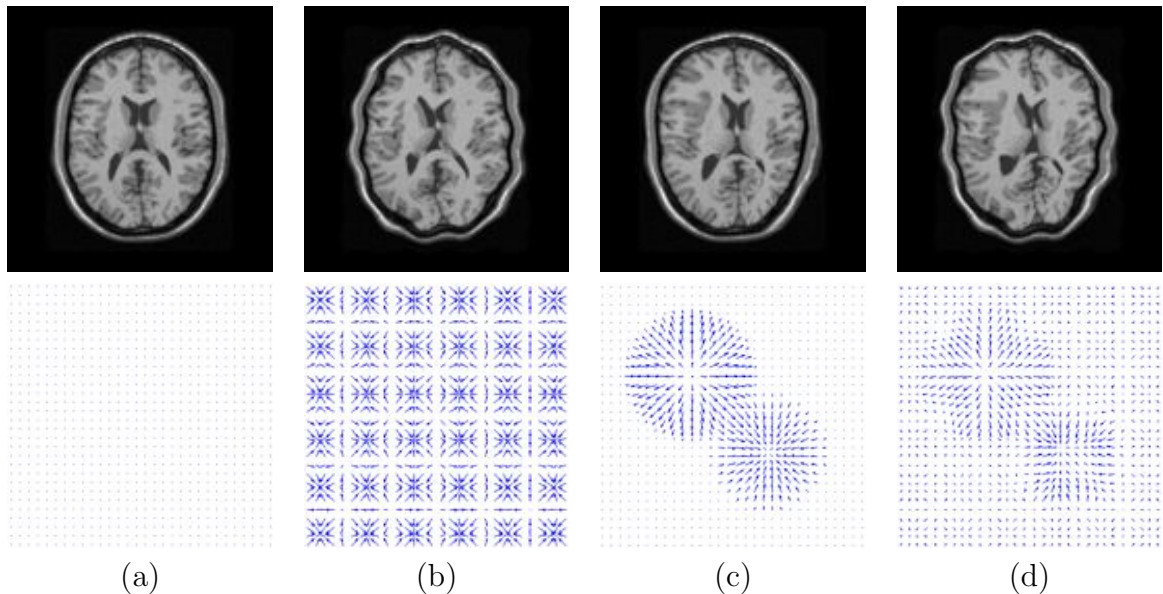


Figure 5.3: Slice and corresponding displacement field of original MNI image (a) and after local warping with sine waves (b), global warping with a pinch and a bulge (c), and combined local/global warping (d)

where the volume of interest Ω is chosen to be the head. Voxels outside the head are excluded from computation of Equation 5.6 using a binary mask.

5.1.3 Retrospective Image Registration Evaluation

We evaluate the accuracy and speed of our GPU registration methods on a set of clinical images. The images are of brain tumor patients from Vanderbilt University’s Retrospective Image Registration Evaluation (RIRE) Project [93]. The primary objective of the RIRE Project is to evaluate the clinical accuracy of retrospective techniques for registering inter-modality images of the head. Retrospective techniques, such as ours, are based on the analysis of image intensities related to anatomical features. In this study, registration results from various retrospective techniques are compared against results from a gold standard, prospective registration technique based on physical fiducial markers. The gold standard transformations remain sequestered from us and other study participants in order to ensure blinding in the study. Following registration of the standard dataset, transformation parameters are sent to Vanderbilt, where they are compared against the gold standard. Results are reported back in terms of geometric error in millimeters, allowing for ranking of competing retrospective registration algorithms.

The authors of the RIRE study made publicly available their image database¹ of 18 patients, whose heads had been scanned using positron emission tomography, computed tomography, and magnetic resonance imaging modalities. Table 5.1 gives the database image sizes and voxel spacings. As shown in the table, the patients

¹The Vanderbilt RIRE database can be downloaded at <http://insight-journal.org/rire/>. The site includes results of alignment methods from numerous research groups.

are divided into two groups (A and B) based on the imaging parameters used. Each group has nine patients. One PET image, which is not represented in the table, has a voxel size of $1.94 \times 1.94 \times 8.00$ mm. Images from all modalities were acquired with contiguous slices (i.e. zero spacing between slices).

The PET images were acquired following injection of the radioactive tracer ^{18}F -fluorodeoxyglucose (^{18}F -FDG), which is used to assess glucose metabolism. It is characterized by elevated uptake in tissues with high metabolic activity, such as malignant brain tumors. The CT images were acquired without intravenous contrast agent; they are helpful in visualizing relatively dense structures, such as bone. The MR images were acquired for each patient using T1-weighted, T2-weighted, and proton-density (PD) spin-echo imaging sequences on a 1.5 T scanner. The MR images demonstrate soft tissue contrast, such as between tumor and normal brain parenchyma. Figures 5.4 and 5.5 show slices of the raw PET, CT, and MR images of a sample patient in Group A of the RIRE database.

Modality		Image Dimensions			Voxel Size (mm)		
		x	y	z	x	y	z
Group A	PET	128	128	15	2.59	2.59	8.00
	MR	256	256	20, 26	1.25–1.28	1.25–1.28	4.00–4.16
	CT	512	512	27–34	0.65	0.65	4.00
Group B	MR	256	256	52	0.78–0.86	0.78–0.86	3.00
	CT	512	512	40–49	0.40–0.45	0.40–0.45	3.00

Table 5.1: Size and spacing of images in the RIRE database

Not all patients were imaged using all modalities. Only patients of Group A were imaged using PET. In Group A, PET images were not available for two patients and CT images were not available for two (different) patients. In Group B, proton-density MRI was not available for five patients and there was no T2-weighted MRI for one patient.

A second set of T1, T2, and proton-density MR images was also included for seven patients of Group A in the database. The images in the second set were numerically corrected for geometrical distortions, which are known to decrease registration accuracy. The distortions were due to magnetic susceptibility changes induced by the patient tissues within the scanner. No images in Group B were corrected for geometrical distortion.

Registrations consist of rigidly aligning the PET and CT images to the MR images of each patient. Gold standard results were obtained prospectively at Vanderbilt by attaching fiducial markers to the patients prior to imaging. Two types of markers were used: one bright on CT and MRI; the other bright on PET. Four fiducial markers were attached to each patient’s skull by means of implanted binding posts. The study authors justified the use of this invasive procedure on the patients, stating that it was also used to aid in intra-operative guidance during subsequent neurosurgery. They used the registration results to align the pre-operative images to the patients during surgery.

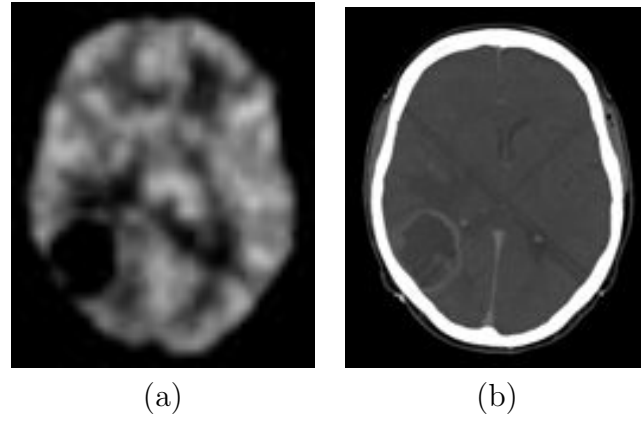


Figure 5.4: Sample unregistered PET (a) and CT (b) images of a patient in Group A of the RIRE database

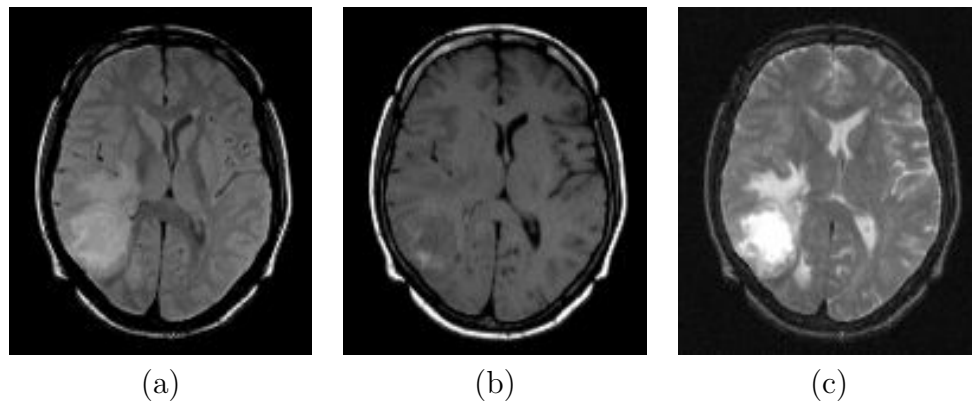


Figure 5.5: Sample MR images of a patient in Group A of the RIRE database: proton-density (a), T1-weighted (b), and T2-weighted (c)

The gold standard rigid-body transformations were found by matching fiducial markers between image pairs. The study authors used a least-squares approach to minimize Euclidean distances between corresponding fiducials. The coordinates used for registration were defined as the centroids of the marker intensities in the images. After finding the gold standard transformations, the images were altered to remove all traces of the fiducials. These altered images were then uploaded to the RIRE database for us to download. Figure 5.6 (modified from West, et al. [93]) demonstrates the removal of the fiducial markers and points on a stereotactic frame that were visible on the original images. The fiducials are circled on the original images in Figure 5.6 (a), (b), and (c).

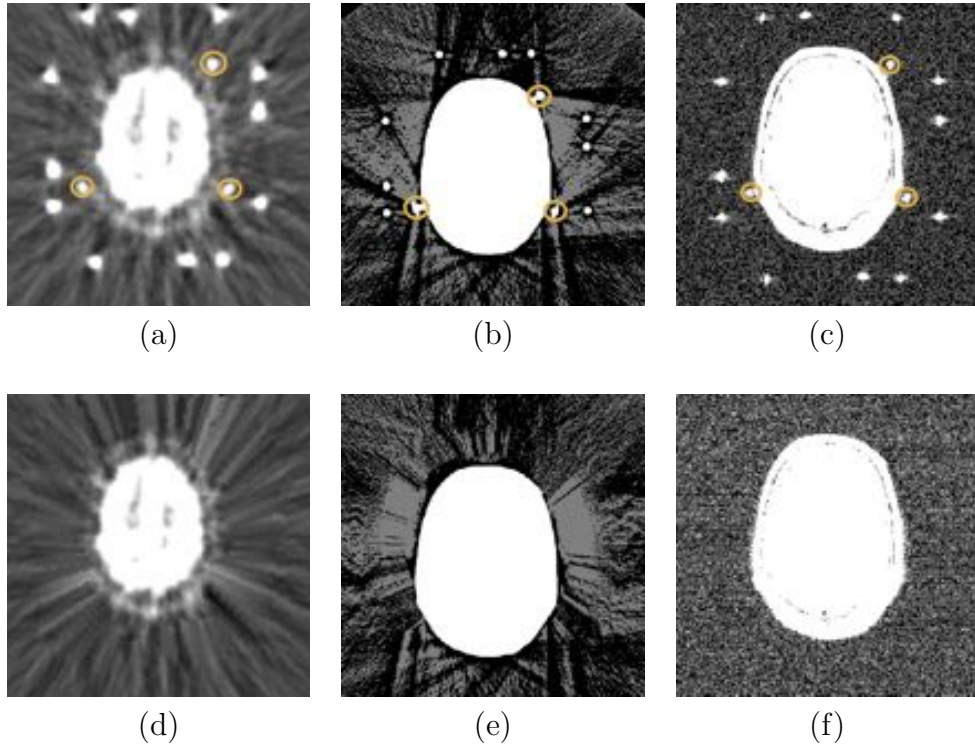


Figure 5.6: Sample PET, CT, and MR images from the RIRE study before (a, b, c) and after (d, e, f) removal of the fiducial markers (circled) and stereotactic frame

Our PET-to-MR and CT-to-MR registration methods are evaluated against these prospectively determined, gold standard transformations. Registration error between the retrospective and prospective methods is defined as mean registration disparity between ten target points in the images. The authors of the study defined these target points to lie within regions of surgical and diagnostic importance.

Registration error is computed as follows. Let $\mathbf{x}_{\text{MR}}$ denote the coordinates of a target point in a patient's MR image. The gold standard transformation is first used to find the corresponding point $\mathbf{x}$ in the patient's CT or PET image. Next, the retrospectively determined transformation (i.e. from our software) is applied to $\mathbf{x}$, yielding the point $\mathbf{x}'_{\text{MR}}$ in the MR image. The target registration disparity for the target point is defined as $\|\mathbf{x}_{\text{MR}} - \mathbf{x}'_{\text{MR}}\|$. The overall registration error of a given

method is the mean disparity computed over all target points.

We register the images using six-parameter, rigid-body transformations. Trilinear interpolation is used to resample the moving images. We evaluate the normalized cross-correlation (NCC) metric for alignment of PET to MR, the normalized gradient field (NGF) metric for alignment of CT to MR, and the normalized mutual information (NMI) metric for alignment of both PET to MR and CT to MR. The NMI, NCC, and NGF metrics are defined in Equations 3.4, 4.3, and 4.5, respectively. All RIRE images are stored using 16-bit signed integer intensities.

The Nelder-Mead simplex optimizer is used for all experiments, with $f_{\text{tol}} = 10^{-4}$, the minimum relative difference between the lowest and highest simplex vertices set to 10^{-4} , and the maximum number of function evaluations per iteration set to one thousand. A multi-resolution optimization strategy is employed, using two image resolution levels for PET to MR registration and three levels for CT to MR registration. Geometrically corrected versions of the MR images are used when available.

5.1.4 Experimental Equipment

All experiments were performed on an Apple MacPro desktop computer, circa 2008. The system ran Mac OS X “Leopard” (version 10.5.3) and was equipped with two 3.2 GHz Quad-Core Intel Xeon processors and 2 GB of main memory. The video card used was an NVIDIA GeForce 8800 GT with 512 MB of video memory.

5.2 Results

In this section, we present the results of testing our GPU framework’s accuracy and speed for registering the synthetic and clinical datasets. It is important to note that, to some extent, high registration accuracy can be achieved at the expense of convergence speed, and vice versa. The precise nature of the relationship between accuracy and speed is complex, as it depends on the choice of methods and the tuning of optimization parameters. In configuring our software for these experiments, we generally opted for accuracy over speed.

5.2.1 Affine Registration Iteration Speed

The mean execution timings of the *transform-resample-metric* iteration cycle are presented in Table 5.2. Results were obtained using the MSE, NCC, NGF, and NMI similarity metrics. The experiments were run on the same workstation using both GPU- and CPU-based implementations of the methods. Each timing is reported as the mean of one thousand cycle iteration timings. The cycle was run with the MNI data as input, resampled to four different sizes.

Average cycle time performance gains using the GPU versus the CPU are 141, 143, 43, and 38 times for the MSE, NCC, NGF, and NMI metrics, respectively. Tests of the NMI metric were conducted using a 256^2 joint histogram. We also tested the NMI metric with joint histograms of size 32^2 , 64^2 , 128^2 , 512^2 , and 1024^2 . No correlation was found between joint histogram computation time and histogram size.

Method		Cycle Timing (msec)			
		size= $128^2 \times 128$	$128^2 \times 256$	$256^2 \times 128$	$256^2 \times 256$
MSE	GPU	2.5	4.3	8.9	18.0
	CPU	322	642	1281	2572
NCC	GPU	2.9	4.5	9.1	18.8
	CPU	338	678	1394	2821
NGF	GPU	9.1	17.4	36.6	64.3
	CPU	373	687	1542	3109
NMI	GPU	11.5	18.4	36.9	83.0
	CPU	378	746	1504	3040

Table 5.2: Mean run times for the transform-resample-metric cycle on the GPU and CPU as a function of image size and similarity metric

For computing the NMI metric results in Table 5.2, we employed an optimization that was discussed in section 4.1.5: Joint histogram bins associated with background values from the two images are not incremented using the vertex scattering mechanism. For the MNI T1- and T2-weighted images used in these tests, our algorithm discarded all vertices destined for scattering to bin $(0, 0)$ of the 256×256 histogram. This accounted for 59% of the total vertices and resulted in an acceleration of 2.15 times over a non-optimized version.

5.2.2 Artificially Transformed Images

In this section, we present the results of recovering transformations artificially applied to the MNI data, which has 256^3 samples and a voxel size of 1 mm^3 . Table 5.3 shows the registration accuracies and timings associated with recovering nine-parameter affine transformations. These are the mean values computed after complete registrations of ten randomly transformed volumes. Accuracy is presented as the mean error between the applied and recovered transformation components, as well as the overall RMS error (Eq. 5.1) over a spherical VOI of radius 80 mm contained within the head. The RMS error is a measure of overall registration accuracy.

The MSE metric was used to register the transformed T1 image to the original T1 image. The other metrics were used for the multi-modal registration of the transformed T1 image to the original T2 image. Registration was performed sequentially at three resolution levels, with an average of approximately 200 iterations used at the lowest resolution (64^3 samples), 150 iterations at the middle resolution (128^3), and 50 iterations at the highest resolution (256^3).

Table 5.4 shows the registration accuracy and timings associated with recovering non-linear transformations using the Demons method on the GPU. The image data was artificially transformed using models of local (Eq. 5.3) and global (Eq. 5.5) deformation. Accuracy is presented as the RMS error between the applied and recovered displacement fields (Eq. 5.6).

The number of iterations was fixed at one hundred for each of the three resolution levels of registration. Mean execution times at the low (64^3), medium (128^3), and

Method		Registration Errors				Timing (sec)
		Translation (mm)	Rotation (°)	Scaling (%)	RMS (mm)	
MSE	GPU	0.139	0.005	0.23	0.200	4.22
	CPU	0.141	0.005	0.22	0.197	60.70
NCC	GPU	0.321	0.016	0.42	0.414	6.38
	CPU	0.299	0.016	0.41	0.393	90.63
NGF	GPU	0.200	0.008	0.39	0.327	11.91
	CPU	0.258	0.008	0.43	0.385	129.75
NMI	GPU	0.175	0.011	0.28	0.246	15.28
	CPU	0.173	0.012	0.27	0.243	136.52

Table 5.3: Errors and run times on the GPU and CPU for nine-parameter, affine registration of the MNI data

Deformation Model	RMS Registration Error (mm)	
	Before	After
Local (sine waves)	2.375	0.299 ± 0.462
Global (pinch and bulge)	2.039	0.394 ± 0.177
Combined (local and global)	3.617	0.510 ± 0.525

Table 5.4: RMS non-linear registration errors before and after GPU Demons registration of the deformed MNI data

high (256^3) resolutions were 1.24, 3.66, and 30.53 seconds, respectively. The overall acceleration of the GPU version compared to the equivalent CPU version was 16 times. The GPU- and CPU-based methods recovered nearly identical deformation fields in all cases. Root mean square errors between the field recovered using the GPU- and CPU-based methods were under 0.04 mm.

Below, we visually depict some of the deformable registration results using the GPU Demons algorithm. All images correspond to the central slice of the 256^3 MNI data. Figure 5.7 shows the recovered x , y , and z components of the local displacement field. Each component was originally warped using a sinusoidal function.

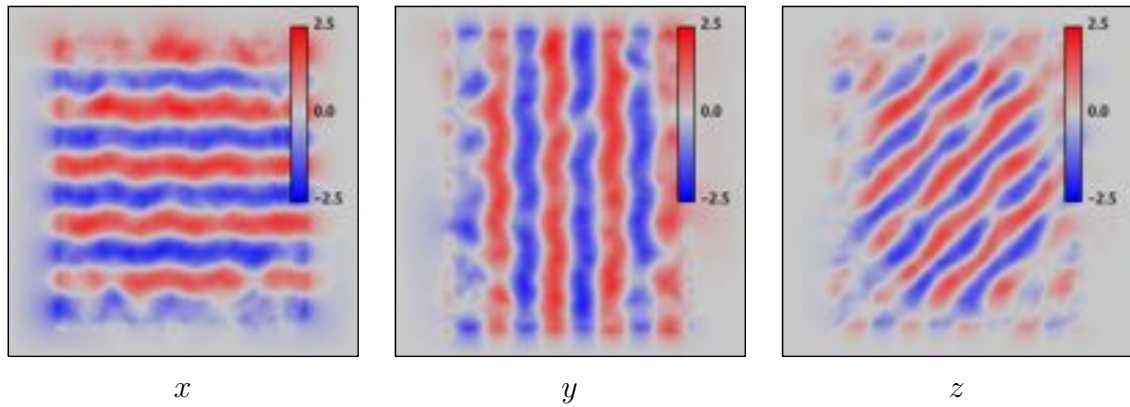


Figure 5.7: Recovered components of local displacement field using GPU Demons

The magnitude of the recovered (restorative) global displacement field and its Jacobian are given in Figure 5.8. Jacobian values less than 1.0 correspond to regions where the restorative deformation causes volume contraction, whereas values greater than 1.0 correspond to volume expansion. Figure 5.9 shows renderings of the recovered deformations and their RMS errors compared to the ground truth. The RMS errors are masked against the MNI image brain.

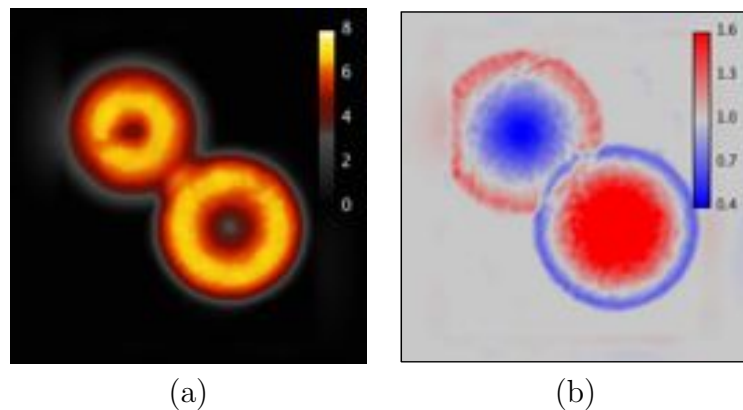


Figure 5.8: Recovered magnitude (a) and Jacobian (b) of global displacement restorative field using GPU Demons

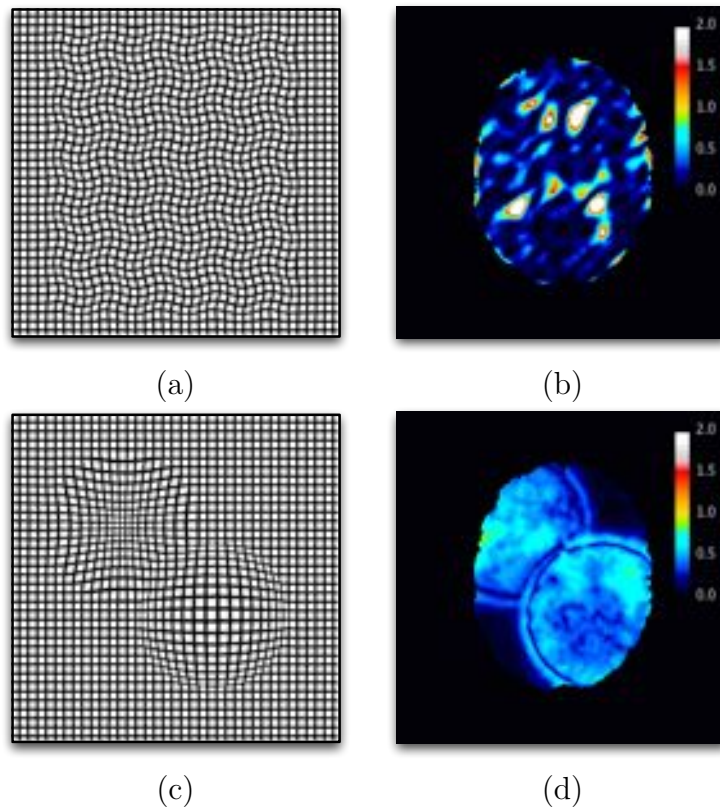


Figure 5.9: Renderings of recovered local displacement field (a) and RMS error (b), and recovered global displacement field (c) and RMS error (d)

5.2.3 Retrospective Image Registration Evaluation

The results of aligning the PET and CT images to the MR images of the RIRE Project database are presented in this section. Accuracy results are given as mean and median errors with respect to the gold standard transformations. Timings for the complete registrations in our GPU-accelerated framework are also given. Standard errors accompany all results.

Table 5.5 summarizes results of the NCC metric, which was used to align PET to MR images. Table 5.6 summarizes results of the NGF metric, which was used to align CT to MR images. The Table 5.7 summarizes results of the NMI metric, which was used to align both PET and CT to MR images.

Group	Modality		Mean Error (mm)	Median Error (mm)	Time (sec)	No. Cases
	From	To				
A	PET	PD	2.986 ± 0.043	2.821	3.49 ± 0.77	5
	PET	T1	3.824 ± 0.070	2.574	3.81 ± 0.77	4
	PET	T2	4.329 ± 0.059	3.515	3.41 ± 0.57	5

Table 5.5: Registration errors and timing for GPU alignment of PET to MR images using the NCC metric

Group	Modality		Mean Error (mm)	Median Error (mm)	Time (sec)	No. Cases
	From	To				
A	CT	PD	1.391 ± 0.012	1.327	10.19 ± 0.44	7
	CT	T1	1.424 ± 0.018	1.316	11.77 ± 0.89	7
	CT	T2	1.435 ± 0.026	0.950	11.85 ± 1.02	7
B	CT	PD	2.763 ± 0.100	2.338	12.39 ± 0.36	4
	CT	T1	2.297 ± 0.039	1.895	12.24 ± 0.25	9
	CT	T2	3.044 ± 0.136	2.508	14.11 ± 0.42	8

Table 5.6: Registration errors and timing for GPU alignment of CT to MR images using the NGF metric

Figures 5.10 and 5.11 show overlaid PET, CT, and MR images of a sample brain tumor patient from Group A of the RIRE study. The representative slices are shown with their corresponding joint histograms before and after rigid-body alignment using the NMI metric. The horizontal axes of the histograms correspond to the MRI intensities. The bin values are represented on a natural logarithm scale. For this patient, the 3D alignment time for PET to T2 was 3.4 seconds, with a median error less than one half of the PET slice thickness. The 3D alignment time for CT to T1 was 11.8 seconds, with a median error equal to the in-plane voxel spacing of the MR dataset, or approximately one third of the CT and MR slice thicknesses.

Group	Modality		Mean Error (mm)	Median Error (mm)	Time (sec)	No. Cases
	From	To				
A	PET	PD	3.021 ± 0.337	2.369	3.45 ± 0.37	5
	PET	T1	2.053 ± 0.163	1.811	3.61 ± 0.39	4
	PET	T2	2.713 ± 0.223	2.349	3.83 ± 0.17	5
	CT	PD	1.228 ± 0.084	1.058	11.19 ± 0.40	7
	CT	T1	1.040 ± 0.085	0.870	9.54 ± 0.39	7
	CT	T2	1.319 ± 0.139	1.083	8.97 ± 0.95	7
B	CT	PD	2.443 ± 0.102	2.388	12.39 ± 0.36	4
	CT	T1	1.806 ± 0.088	1.699	12.24 ± 0.25	9
	CT	T2	2.016 ± 0.048	1.965	14.11 ± 0.42	8

Table 5.7: Registration errors and timing for GPU alignment of PET and CT to MR images using the NMI metric

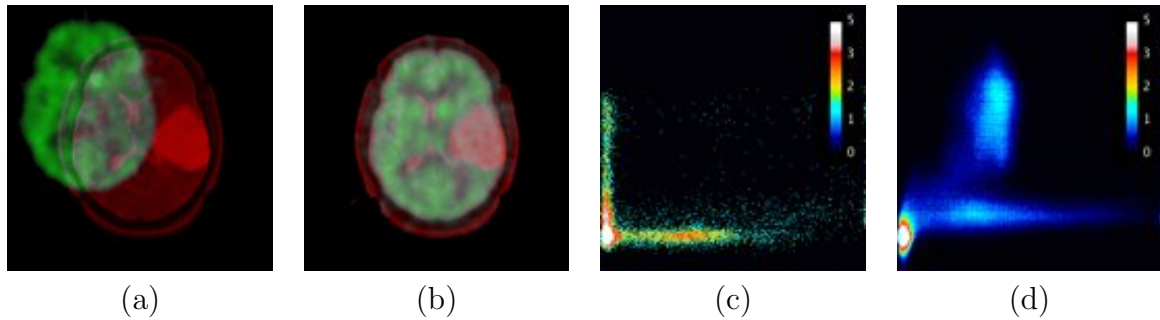


Figure 5.10: Overlaid PET and T2-weighted MR image slices and their joint histograms (PET vs. T2) before (a, c) and after (b, d) rigid-body alignment

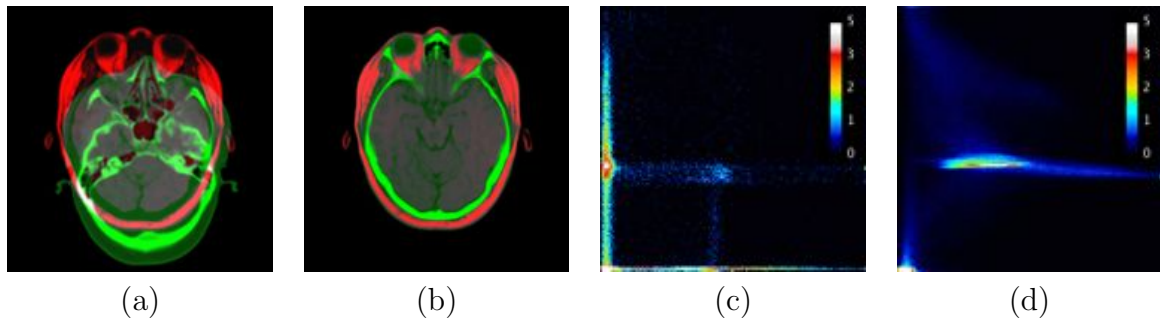


Figure 5.11: Overlaid CT and T1-weighted MR image slices and their joint histograms (CT vs. T1) before (a, c) and after (b, d) rigid-body alignment

5.3 Discussion

In the medical imaging community, certain attributes are generally demanded of a good registration tool. These include speed, accuracy, robustness, and the need for minimal user intervention [102]. Robustness is a broad term that we define as the tool’s ability to register images of varying quality and modality, even under large initial mis-registrations. As we discuss below, the results of our experiments show that our software has these desired attributes.

5.3.1 Speed and Accuracy

In Table 5.2, we compared timings between GPU- and CPU-based implementations of the image transformation, resampling, and similarity metric steps. For execution of the registration cycle using the MSE and NCC metrics, we achieved a mean acceleration of two orders of magnitude. These findings agree with those of Chan, who tested his software on a high-end graphics workstation in 2007 [80]. Accelerations of one order of magnitude were found using the NGF and NMI metrics. On both the GPU and CPU, cycle computation times roughly scaled linearly with the number of voxels processed and were constant with respect to histogram size, as expected [88].

On the GPU, the NGF metric executed approximately three times slower than the MSE and NCC metrics. This increased time complexity is mainly due to image gradient computations in the NGF routine. Such a marked slowdown was not found for the CPU implementation of NGF. This is presumably due to more sophisticated optimization of floating-point calculations by the CPU and C++ compiler than the GPU and OpenGL.

We note that accelerations determined for the transform-resample-metric cycle are not applicable to the entire image registration process. These isolated cycle timings discount relatively costly operations associated with running a complete registration, such as OpenGL environment setup, data transfers between main memory and video memory, and control logic in the optimizer. Speed-ups of between ten and twenty times are commonly observed with GPU medical image registration applications [82].

To evaluate our affine registration accuracy, we used our software to recover affine transformations applied to synthetic test images. These results were given in Table 5.3. The GPU and CPU versions of affine registration yielded nearly identical registration errors, all of which were sub-voxel in magnitude. The slight differences between the GPU and CPU results are most likely due to a non-standard implementation of floating-point calculations on the tested graphics hardware. These differences should not be present on later model hardware, such as the Fermi architecture, which implements IEEE standards for 32-bit floating-point operations [37]. Still, the near perfect registration results and concordance between the GPU and CPU versions prove the validity of our affine registration methods. For the affine registration tests, we achieved a speedup of approximately 13-fold using graphics hardware.

Artificial non-linear transformations of both a local and global nature were also applied to the test images. We recovered these deformations using our GPU and CPU implementations of the Demons method. Quantitative results for the GPU method

were presented in Table 5.4 as the residual root mean square error between the applied and recovered fields. We were able to recover the deformations with average errors below 1 mm while achieving GPU acceleration of an order of magnitude over the CPU. Visual inspection of the images after registration confirmed that the Demons method recovered the artificial warps.

We also showed that our affine registration methods are effective on multi-modal clinical data. We aligned PET and CT images to MR images of brain tumor patients in the Retrospective Image Registration Evaluation Project. Results were given in Tables 5.5, 5.6, and 5.7. With respect to the input image voxel sizes (see Table 5.1), all RIRE mean target registration errors were sub-voxel in magnitude. Larger errors were noted for images in Group B, since unlike the Group A images, they were not corrected for geometrical distortion. All errors were reported with respect to the gold standard fiducial marker technique. The RMS error of the gold standard fiducial-based registration is estimated to be approximately 0.39 mm for CT-to-MR and 1.65 mm for PET-to-MR [93].

5.3.2 Normalized Mutual Information Metric

Normalized mutual information has long been established as one of the most robust image similarity metrics in medical image registration. Optimization of the metric has been proven to allow fully automated affine registration of CT, MR, and PET images without the need for preprocessing, such as landmark or feature definition [19, 109]. Unlike measures based on differences and correlation, mutual information does not assume a predefined mathematical relationship between image intensities. Results of the RIRE study have also shown that MI-based methods are among the most accurate [93]. Indeed, a literature review reveals that either MI or NMI are used to drive linear registration in the vast majority of published neuroimaging studies [13].

The robustness and accuracy of NMI was shown in our experiments. The metric yielded superior registration results for both PET-MR and CT-MR cases compared to the normalized gradient field and normalized cross-correlation metrics. The NCC metric was only applied to PET-MR cases; it was not able to successfully align CT and MR images. This is due to the inherent lack of spatial correlation between CT and MR voxel intensities. The NGF metric could not align the PET and MR images, because the PET images have weak gradient values that lack spatial coherence with MR image gradients. Thus, the NGF metric was only applied to the CT-MR cases. The NMI metric is robust and had none of these limitations. It could be employed successfully in all cases.

We use vertex scattering to generate the joint histograms required for mutual information: Parallel threads of shaders read image intensities from texture memory and increment histogram bins in the frame buffer. With this method, we take advantage of modern hardware’s ability to effectively allocate either vertex or fragment processing tasks on the fly. Heavy use of vertex processing would have been disadvantageous prior to the unified shader architecture, since vertex shading units had a lower level of parallelism than fragment shading units. For instance, the NVIDIA GeForce 7800 GTX GPU, released mid-2005 (just prior to CUDA), had 24 fragment

shaders and 8 vertex shaders. Current GPUs essentially consists of a collection of flexible floating-point engines.

We also rely on graphics hardware to serialize the bin increment operations in the frame buffer, thus guaranteeing that no two threads can simultaneously read or write a value at the same bin address. The method by which the hardware ensures atomicity and prevents memory collisions has not, to our knowledge, been made public. Regardless, we can be assured that NVIDIA’s hardware implementation of memory locking is efficient. This was shown with our NMI metric cycle timings in section 5.2.1. We compared timings with vertex scattering to bin $(0, 0)$ both enabled and disabled. When enabled, bin $(0, 0)$ is incremented by 59% of the scattered voxels, or nearly ten million times per iteration. This certainly results in thousands upon thousands of memory collisions at this bin. When disabled, none of these collisions take place. However, the resulting time savings is two-fold, which is relatively low. This leads us to believe that the GPU very efficiently handles collisions at the histogram bins.

5.3.3 Real-Time Visualization

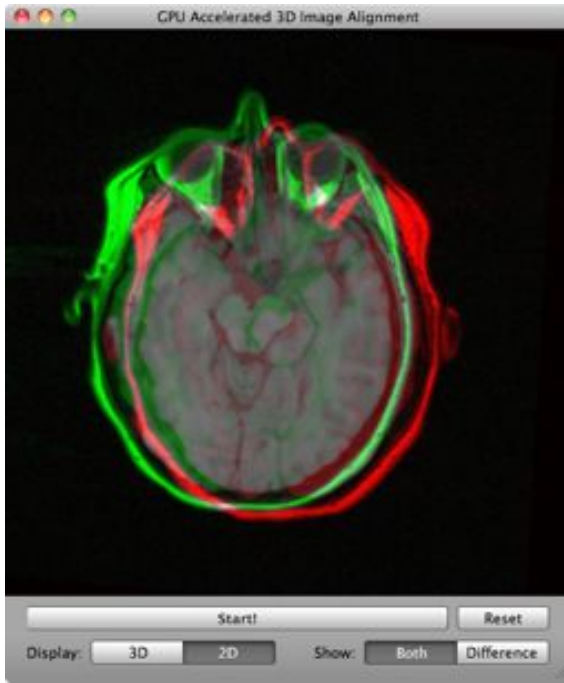
In practice, registration quality between two images is often assessed qualitatively by viewing their fused overlay or their difference image [114]. Integration of image registration and 3D visualization was pioneered by Hastreiter, et al., who applied their work to image-guided neurosurgery [60].

An advantage of our tool is that it provides visualization of the registration process in real-time [80]. Fused 2D or 3D image views, similarity metric images (as described in Fig. 4.6), and the joint image density distribution can be viewed as registration progresses. In addition, affine transformations can be initialized by manually aligning the moving and fixed images. In non-linear Demons-based registration, the user can view the current displacement field. By providing these options, our software allows the user to quickly detect and potentially correct failed registrations and to scrutinize the effects of adjusting registration parameters.

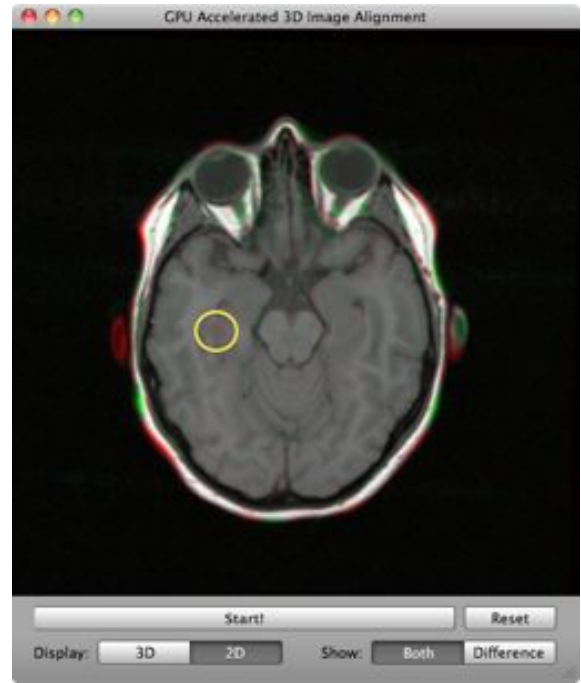
Integrated visualization was straightforward to accomplish without significant overhead, because image transformations and metrics are implemented in OpenGL. Our experiments with registration using the CUDA environment have shown that transfers of computed results to a graphics context for visualization constitute a significant bottleneck.

Figure 5.12 shows screen captures of our GPU alignment software developed with Chan. Views of the images in 2D or 3D are updated in real-time as registration proceeds. The T1-weighted MR images in this example were acquired of the same multiple sclerosis patient eight months apart. The baseline scan is loaded into the red channel and the followup scan is loaded into the green channel.

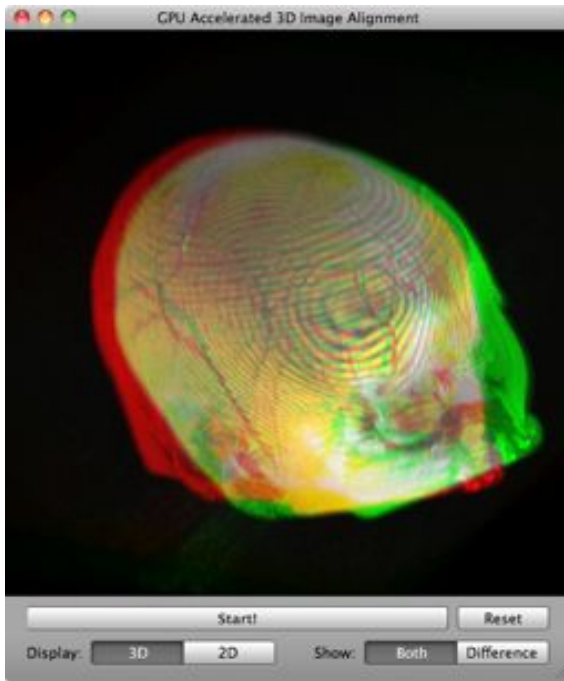
Prior to rigid-body registration in (a) and (c), it is difficult to ascertain longitudinal changes between the two scans. After registration in (b) and (d), internal brain structures are well aligned, though some flexible external soft tissues, such as the ears and ocular muscles remain unaligned. The circled red spot in (b) indicates a hypointense white matter lesion present at followup but not at baseline [122]. It would be difficult or nearly impossible to detect such changes by looking at the unaligned



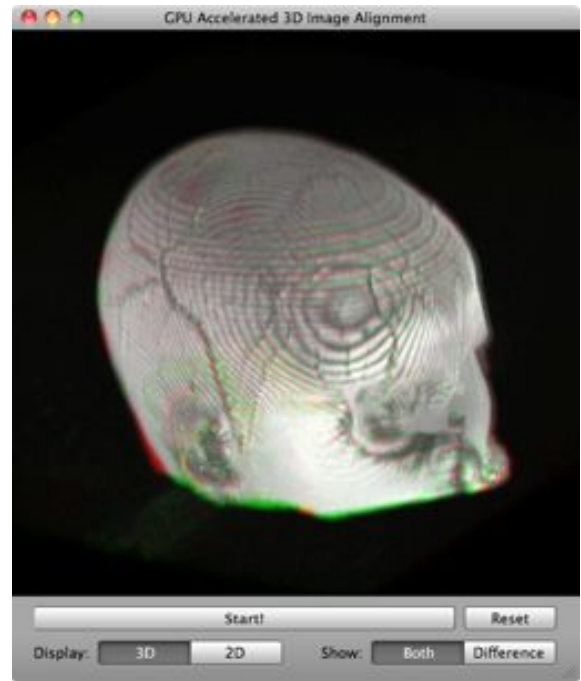
(a)



(b)



(c)



(d)

Figure 5.12: Screenshots of our GPU accelerated image registration tool showing MR images of the same subject at two time points before and after alignment in 2D (a, b) and rendered in 3D (c, d)]

images.

Other promising clinical scenarios include the detection of brain atrophy in patients with cognitive decline [6] and the assessment of tumor changes or metastatic growths in cancer patients [44]. Our tool could also prove valuable in time-critical settings, such as for rapidly correcting patient misalignment in emergency CT scans and for guiding radiotherapy treatment [123].

Chapter 6

Tumor Spatial Distribution Analysis using Image Registration

6.1 Preface

In this chapter, we analyze the 3D spatial distribution of *glioblastoma multiforme* (GBM) brain tumors in relation to a genetic trait known as O₆-methylguanine-DNA methyltransferase (*MGMT*) promoter methylation. In particular, we analyze the correlation between GBM tumor distribution and methylation status in patients. This work is included in the thesis as a clinical research example of image registration.

The content of this chapter was accepted for publication in September 2009 by the journal *NeuroImage* with the title “An analysis of image texture, tumor location, and *MGMT* promoter methylation in glioblastoma using magnetic resonance imaging” [124]. The authors, listed in order, are: Sylvia Drabycz, Gloria Roldán, Paula de Robles, Daniel Adler, John B. McIntyre, Anthony M. Magliocco, J. Gregory Cairncross, and J. Ross Mitchell.

This journal paper combines the work of two clinical studies. The first study investigated whether image texture features from MRI could be used to non-invasively predict *MGMT* promoter methylation status in glioblastoma multiforme tumors. Tumor texture was assessed using visual descriptors and space-frequency transform analysis. This work was primarily conducted by Sylvia Drabycz and Gloria Roldán, M.D.

The second study, which is reported in this chapter, was conducted primarily by myself and Paula de Robles, M.D., a neuro-oncology fellow from the Departments of Oncology and Clinical Neurosciences at the University of Calgary. The study was conceived by Ross Mitchell, Ph.D. and Gregory Cairncross, M.D., chair of the Department of Clinical Neurosciences at the University of Calgary. The tumor distribution study methodology was devised by myself and Dr. Mitchell. I conducted the analysis of tumor distribution and wrote the corresponding methods and results sections in the manuscript. Dr. de Robles segmented the tumors from all patient images and aided in the verification of registration results. Drs. de Robles, Roldán, and Cairncross were responsible for reviewing patient charts to create the database of clinical and imaging data and for writing our article’s introduction and clinical discussion sections. Determination of *MGMT* promoter methylation status was done by Dr. Magliocco from the Departments of Oncology, and Pathology and Laboratory Medicine, University of Calgary.

Automated image registration was an important step in the analysis workflow of this study. Automated registration was used to align all patient images into a common anatomical coordinate system, which was defined by an atlas template image. We mapped and analyzed the spatial distribution of tumors in this common atlas space.

We used our GPU-accelerated software for all registrations in this study. Registrations took on the order of seconds to complete using an NVIDIA GeForce 8600 GT video card with 256 MB of video memory. This is nearly an order of magnitude faster

than performing the registrations using CPU-based software that is conventionally used in similar research workflows [80]. GPU acceleration of registration significantly cut down on processing time.

6.2 Introduction

Glioblastoma multiforme is the most common primary brain tumor in adults. The DNA alkylating agent temozolomide (TMZ) is the only chemotherapy that when added to radiotherapy significantly prolongs patient survival [125]. For unknown reasons, a DNA repair gene known as O₆-methylguanine-DNA methyltransferase (*MGMT*) is epigenetically silenced by promoter methylation in about 50% of newly diagnosed cases of GBM [126]. Silencing of *MGMT* gene promoter has clinical importance, because this genetic alteration predicts benefit from TMZ chemotherapy [126]. Because *MGMT* repairs the therapeutic DNA damage caused by TMZ, silencing of *MGMT* promoter (by methylation) renders the otherwise drug resistant cancer GBM sensitive to chemotherapy [127]. To date, the test for *MGMT* promoter methylation status by methylation-specific polymerase chain reaction is the only tool available to help guide clinicians to identify GBM patients who may benefit from TMZ therapy [127].

In this study, we examined the topographical distribution of newly diagnosed GBM tumors in relation to their *MGMT* promoter methylation status in an effort to understand the biological basis of *MGMT* promoter methylation. We wanted to learn whether chemo-sensitive methylated tumors are preferentially located in specific brain regions and whether they have the same topographical distribution as unmethylated ones. For example, it has been demonstrated in oligodendrogliomas that tumor location within the brain correlates with genetic signature [128, 129].

If methylation of *MGMT* promoter is determined stochastically in evolving GBMs, then methylated and unmethylated tumors are likely to have similar distributions in the brain. If, on the other hand, methylation of *MGMT* promoter is an early event in tumorigenesis or a characteristic of a subset of glial cells from which some GBMs arise, or it is an epigenetic event related to a particular cellular micro-environment, then important differences in the locations of methylated and unmethylated tumors may be observed. For example, different patterns of *MGMT* promoter methylation in oligodendrocyte lineage versus astrocyte lineage cells or in glial stem cells versus committed progenitor cells might result in different locations for methylated versus unmethylated tumors.

Recent experimental data suggest that GBM tumors arise from pluripotent glial progenitors or neural stem cells [130–132]. Also, patterns of methylation appear to be uniform within each GBM [133]. These observations suggest that *MGMT* promoter methylation status is either a characteristic of the cell of origin, or is an early event in GBM tumorigenesis [134]. Eoli et al. have demonstrated a correlation between *MGMT* promoter methylation status and tumor location assessed visually by expert judgment [135]. They reported that tumors with methylation status were preferentially located in the parietal and occipital lobes, whereas tumors without

methylation status were more frequently located in the temporal lobes ($P = 0.005$).

We hypothesize that distinct subtypes of GBM may arise from different precursor cells, have different patterns of MGMT promoter methylation, and therefore behave in biologically distinct ways and arise within different regions of the brain.

6.3 Methods

6.3.1 Patient Selection

This study included patients (age ≥ 18 years) with newly diagnosed GBM (astrocytoma grade IV, WHO classification), as identified through their pathology report based on first surgery, who were treated at the Tom Baker Cancer Centre (TBCC) in Calgary, Alberta between January 1, 2004 and December 31, 2006. Available for all included patients were a paraffin-embedded tumor tissue sample taken from the first surgery and a preoperative, axial T1-weighted post-gadolinium MRI archived in a Picture Archiving and Communication System (PACS). Exclusion criteria for this study included patients with non-GBM pathology or inability to determine *MGMT* status. Anonymized case information was collected through chart review and collated into a single database for analysis of clinical and imaging data.

6.3.2 DNA Samples

The *MGMT* promoter methylation status was assessed by methylation-specific polymerase chain reaction (MS-PCR). Genomic DNA was isolated from paraffin sections of the tumor tissue. For each sample, 1 μ g of DNA was subjected to bisulphate conversion according to manufacturer's protocol (EZ DNA Methylation-Gold Kit, Zymo Research). MS-PCR products were performed as previously described using a two-step approach [126]. PCR products were separated on agarose gels and visualized by ethidium bromide staining, and were then analyzed by a pathologist [JBM] unaware of the clinical results.

6.3.3 Image Processing

The tumor location analysis was performed by retrospectively reviewing the pre-operative axial T1-weighted post-gadolinium MRI brain exams. The brain tumors were manually segmented on the axial MRI slices of each patient by a neuro-oncologist [PdR]. Tumor margins were defined with the aid of appropriate windowing and leveling settings.

Next, all MRIs were registered to a common normalized (stereotaxic) space defined by the T1-weighted MRI atlas ($181 \times 217 \times 181$ samples, $1 \times 1 \times 1$ mm voxel size, 0% noise relative to brightest tissue, 0% non-uniformity) of the Montreal Neurological Institute (MNI) Simulated Normal Brain Database [117]. This simulated brain atlas was created by averaging the co-registered, intensity-normalized MRIs of 305 young, normal right-handed subjects in atlas space [119]. The GBM patient images were registered to the target MNI atlas using automated registration software [15] by maximization of the normalized mutual information similarity metric [19, 84] and

linear scaling spatial transformations. The scaling transformations had nine degrees of freedom, permitting 3D translation, rotation, and scaling. We used a three-level multi-resolution metric optimization strategy. The parameters derived from the MRI registration process were saved and used to transform the tumor segmentation from each patient into atlas space. Figure 6.1 illustrates the image segmentation and registration workflow.

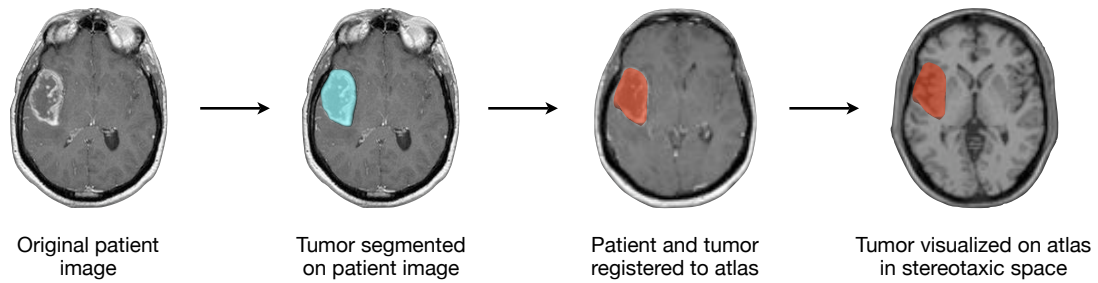


Figure 6.1: Segmentation and registration workflow demonstrated on GBM patient image

6.3.4 Tumor Distribution

To define tumor distribution, we partitioned the brain using two methods: regions and sectors. The following anatomical regions were manually defined on all slices of the MNI simulated brain atlas in normalized space: frontal, temporal, parietal, and occipital lobes (left and right); basal ganglia and cerebellum (left and right); and brain stem. Figure 6.2 shows three views of the neuroanatomical regions used in this study. The atlas was also divided into sectors defined by three orthogonal cut planes through a point 10 mm superior to the centre of the third ventricle. The sectors were labeled as inferior/superior, anterior/posterior, and left/right.

We then measured the tumor volume and the number of tumors in each region and sector. The “primary region” and “primary sector” was defined for each case as the anatomical region or sector containing the largest percentage of the tumor’s volume. The volume of a tumor in a given region/sector was computed as the volume of intersection between the tumor segmentation and the region/sector in atlas space. Tumors visually judged to arise in several distinct locations were deemed multifocal, and those involving both cerebral hemispheres were deemed bilateral. The significance of association between tumor occupancy and methylation status was assessed with the generalized Fisher exact test [136]. Tests were conducted to determine whether the tumors were preferentially distributed in certain regions of the brain. This was done by comparing the observed tumor occupancy distributions with a uniform random distribution using the non-parametric Wilcoxon rank-sum test.

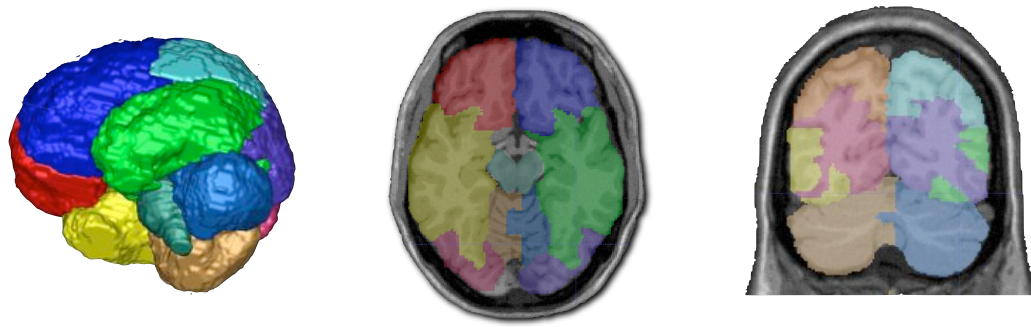


Figure 6.2: MNI atlas segmented into gross neuroanatomical regions shown in 3D rendering (a), axial slice (b), and coronal slice (c)

6.3.5 Tumor Volume

Tumor segmentations from all patients were summed in atlas space to form a map of tumor distribution within the brains of the study population. Two maps were created: one for tumors in which the *MGMT* promoter was methylated and one for those in which the *MGMT* promoter was unmethylated. Figure 6.3 illustrates the workflow used to superimpose the tumors in atlas space for four sample patients. The aggregate methylated and unmethylated tumor volumes within each anatomical region were computed from the tumor maps. The aggregate tumor volume within each region was compared between the two methylation status groups. The non-parametric Wilcoxon rank-sum test was used to compare volume distributions, which were non-normally distributed by the Lilliefors test.

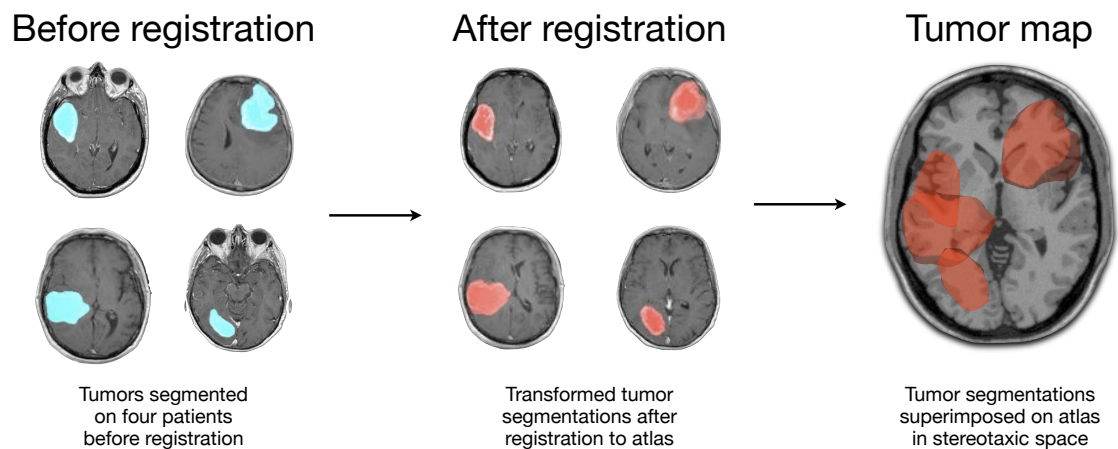


Figure 6.3: Workflow to create tumor volume map for four example GBM patient images

6.3.6 Tumor Centroids

The three-dimensional position of each tumor, as defined by the centroid, was measured. Tumor centroid was computed as the arithmetic mean of each coordinate of the voxels in the segmentation. Two-dimensional plots were created by projecting centroids onto planes oriented in the axial, coronal and sagittal directions. Centroid plots corresponding to methylated and unmethylated GBMs were compared using the 2D Kolmogorov-Smirnov (K-S) test [9].

The distance from each centroid to the centre of the brain was also calculated. We used three points in the third ventricle as alternative definitions of the brain centre: middle of third ventricle, 10 mm superior to the middle, and 20 mm superior to the middle. The non-parametric Wilcoxon rank-sum test was used to compare distance, which were non-normally distributed by the Lilliefors test.

6.3.7 Case Selection and Imaging Parameters

The MR images of 103 patients with newly diagnosed GBM were initially retrieved. Patients were excluded from the study for the following reasons: tumor was reclassified as a non-GBM (2 cases); *MGMT* methylation status was not assessable (21 cases); MR images were not obtained on a 1.5-T scanner (3 cases); MR images were acquired postoperatively (1 case); MR images had severe motion artifacts or were of very poor quality (4 cases).

Seventy-two patients (48 men; 24 women) were included in the location analysis. The median age at diagnosis was 59 years (range = 29-82) and their median Karnofsky Performance Score was 80 (range = 50-100). Median imaging parameters of the T1 post-contrast images used were as follows: TR = 552 ms, TE = 16 ms, $22.0 \times 22.0 \times 15.0$ cm field of view, 256×256 matrix size, 20 slices. Of the 72 cases of GBM analyzed, 36 (50 %) were determined to have *MGMT* promoter methylation status and 36 were determined to not have *MGMT* promoter methylation status.

6.4 Results

Figure 6.4 shows the segmentation of a tumor in the contrast-enhanced T1 image ($448 \times 512 \times 59$ samples) of a patient with *MGMT* methylation status. This particular sample displays an enhancing ring pattern on its margins with higher intensity than the tumor bulk.

6.4.1 Tumor Distribution

Figure 6.5 shows the geometrical extents of the tumors in atlas space, classified by genotype. These maps were obtained by superimposing all tumor segmentation boundaries following registration to the MNI atlas. Map intensities indicate the number of overlapping tumors in a particular area. Halos outside of the skull are due to the inability of scaling transforms to perfectly account for differences between patient and target brain shape.

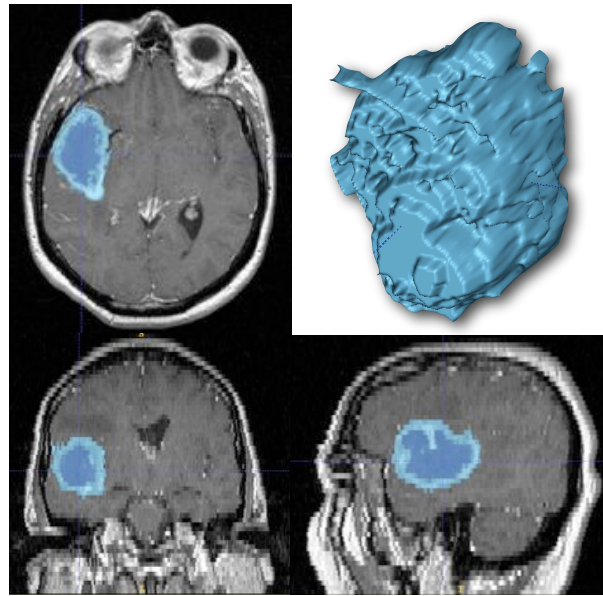


Figure 6.4: Sample segmentation of GBM tumor from our study

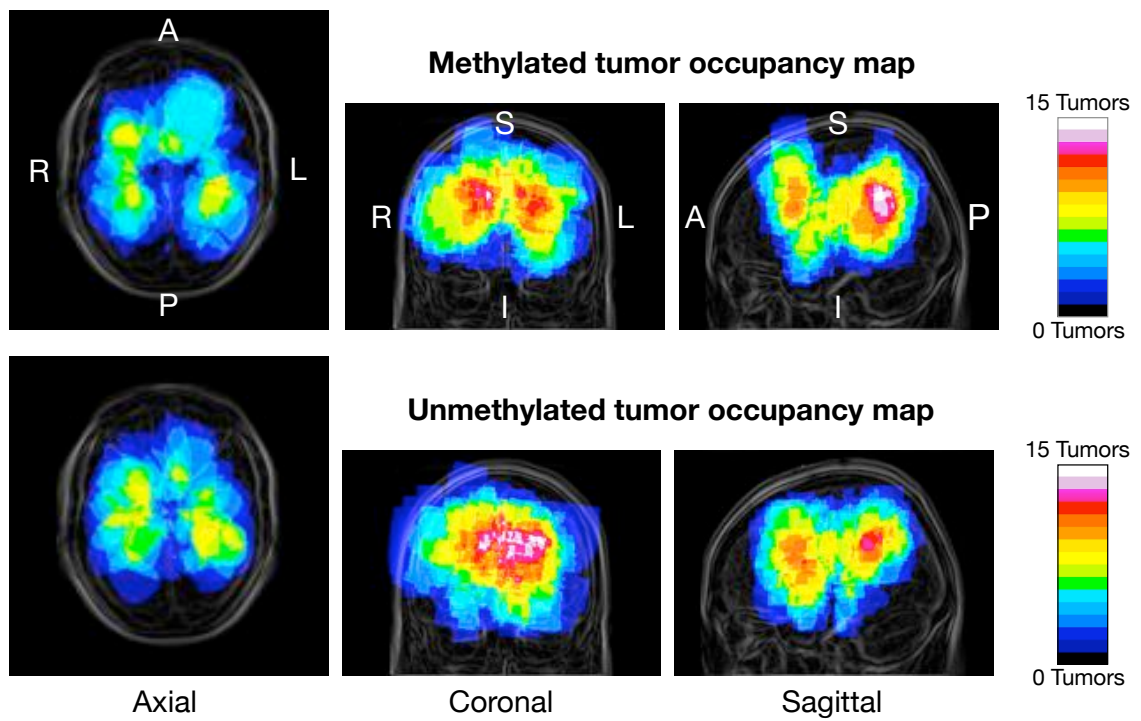


Figure 6.5: Occupancy maps depicting number of overlapping tumors after registration to atlas space

The primary regions of occupancy of the methylated ($n = 36$) and the unmethylated ($n = 36$) tumors are summarized in Table 6.1. One methylated tumor was bilateral and two were multifocal; two unmethylated tumors were bilateral and three were multifocal. Two other tumors could not be assigned to a unique region of primary occupancy. Using the generalized Fisher exact test, there was no evidence for different distributions between the two groups of tumors. Tests were done both accounting for differences between left and right sides ($P = 0.88$) and grouping sides ($P = 0.85$).

Region	Methylated		Unmethylated		Total	
Frontal L	5	8/36 (22)	6	11/36 (31)	11	19/72 (26)
Frontal R	3		5		8	
Temporal L	8	16/36 (44)	11	16/36 (44)	19	32/72 (44)
Temporal R	8		5		13	
Parietal L	4	10/36 (28)	2	7/36 (19)	6	17/72 (24)
Parietal R	6		5		11	
Occipital L	1	1/36 (3)	0	1/36 (3)	1	2/72 (3)
Occipital R	0		1		1	
Other		1/36 (3)		1/36 (3)		2/72 (3)

Table 6.1: Number of GBM tumors occupying defined anatomical regions [n (%)]

The methylated and unmethylated tumor counts in the frontal, temporal, parietal, and occipital lobes were compared; the methylated and unmethylated tumors were preferentially distributed in the temporal lobe compared to a uniform random distribution ($P = 0.029$).

The primary sectors of occupancy are summarized in Table 6.2. There was no evidence for different distributions of methylated and unmethylated cases ($P \geq 0.64$).

6.4.2 Tumor Volume

Aggregate tumor volume maps in atlas space are shown in Figure 6.6. Projections of the 3D maps are shown on the axial, coronal, and sagittal planes, superimposed on the outline of the MNI atlas. The map scale has units of 1 mm^3 volume per 1 mm^2 area. The intensity at a given point is directly proportional to the total tumor volume at that point. The maps have the same resolution as the MNI atlas.

Table 6.3 shows the results of comparing the distributions of tumor volume in the neuroanatomical regions between methylated and unmethylated cases. The table reports the ratio of total tumor volume within a region to the region volume as “normalized tumor volume”. There is no evidence that tumor volume in the regions is differentiated by methylation status ($P \geq 0.34$). A similar analysis showed no evidence for different volume distributions among the brain sectors ($P \geq 0.49$). There was also no difference between the median methylated and unmethylated tumor volumes (45.4 cm^3 and 45.1 cm^3 , respectively; $P = 0.55$), as shown in Figure 6.7.

Sector	Methylated	Unmethylated	Total
Inf Ant Left	3/36 (8)	5/36 (14)	8/72 (6)
Inf Ant Right	5/36 (14)	4/36 (11)	9/72 (13)
Inf Post Left	5/36 (14)	5/36 (14)	10/72 (14)
Inf Post Right	4/36 (11)	2/36 (6)	6/72 (8)
Sup Ant Left	4/36 (11)	4/36 (11)	8/72 (11)
Sup Ant Right	3/36 (8)	5/36 (14)	8/72 (11)
Sup Post Left	6/36 (17)	6/36 (17)	12/72 (17)
Sup Post Right	6/36 (17)	5/36 (14)	11/72 (15)
Inferior	17/36 (47)	16/36 (44)	33/72 (46)
Superior	19/36 (53)	20/36 (56)	39/72 (54)
Anterior	15/36 (42)	18/36 (50)	33/72 (46)
Posterior	21/36 (58)	18/36 (50)	39/72 (54)
Left	18/36 (50)	20/36 (56)	38/72 (53)
Right	18/36 (50)	16/36 (44)	34/72 (47)

Table 6.2: Number of GBM tumors occupying defined brain sectors [n (%)]

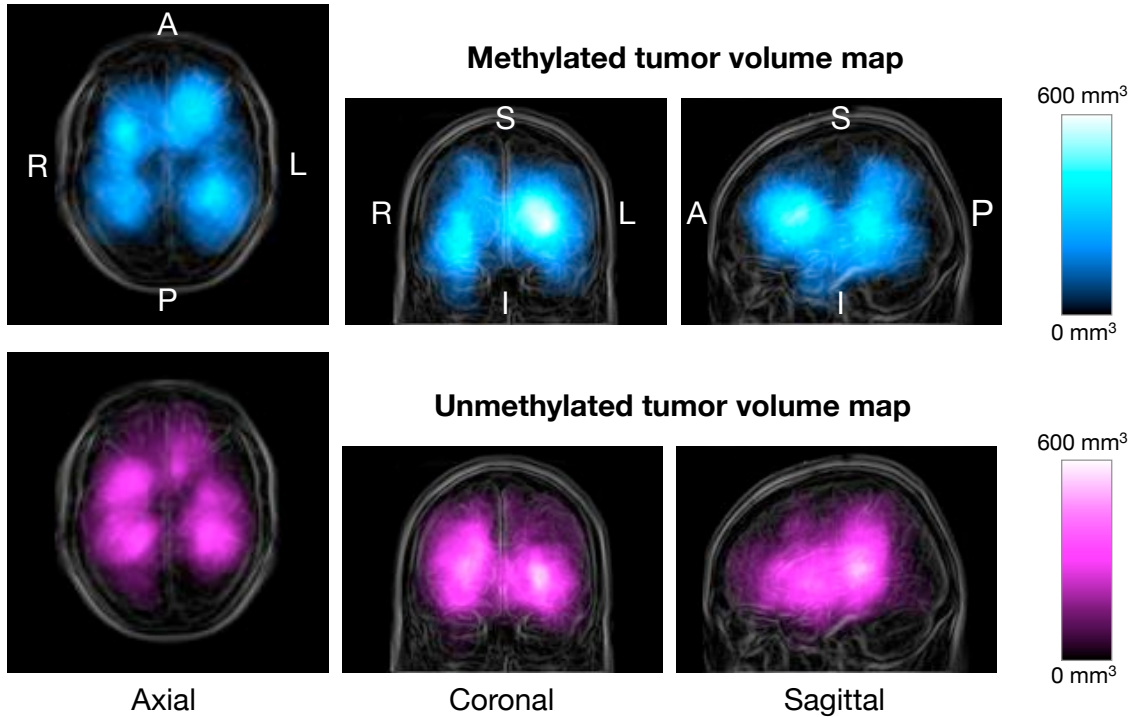


Figure 6.6: Axial, coronal, and sagittal projections of volume density maps (in units of 1 mm^3 per 1 mm^2 area) in atlas space for tumors with (36 cases, cyan) and without (36 cases, magenta) *MGMT* promoter methylation status

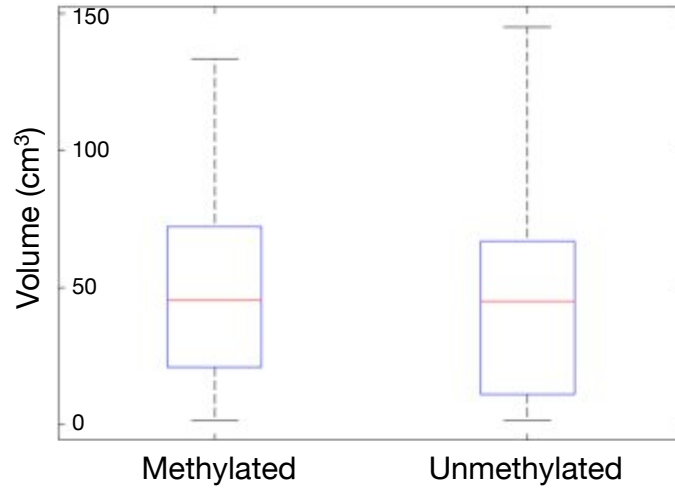


Figure 6.7: Interquartile plot of tumor volumes ($P = 0.55$)

Region	Normalized Tumor Volume	
	Methylated	Unmethylated
Frontal L	1.53	0.96
Frontal R	0.92	1.22
Temporal L	1.96	1.99
Temporal R	1.87	1.64
Parietal L	1.13	0.99
Parietal R	0.79	1.75
Occipital L	0.69	0.26
Occipital R	0.50	0.65
Basal Ganglia L	3.24	2.46
Basal Ganglia R	2.26	2.68
Cerebellum L	0.25	0.13
Cerebellum R	0.32	0.09
Brain Stem	0.36	0.29

Table 6.3: Number of GBM tumors occupying defined brain sectors [n (%)]

6.4.3 Tumor Position

Three-dimensional plots of the tumor centroids in atlas space are shown in Figure 6.8. The orbits and ventricles have been rendered to serve as visual landmarks. Projections of the centroids onto the axial, coronal, and sagittal planes are shown in Figure 6.9. No significant differences were found between the 2D distributions in relation to methylation status ($P = 0.60, 0.42, 0.78$, respectively).

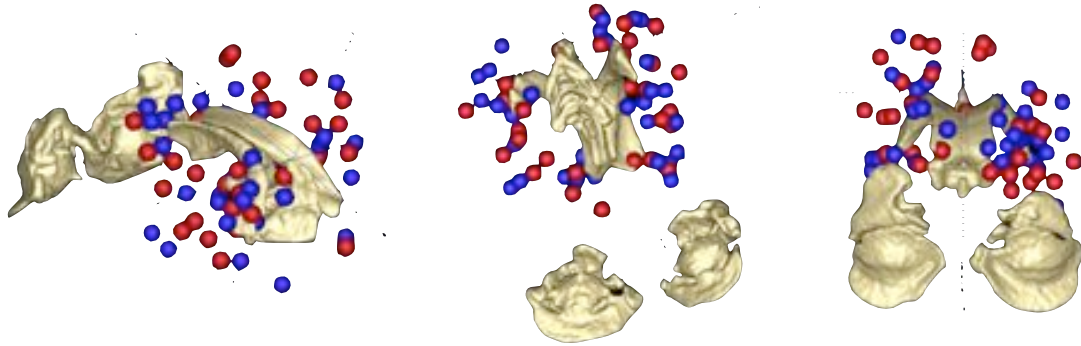


Figure 6.8: Plots of tumor centroids in atlas space for tumors with (36 cases, blue) and without (36 cases, red) *MGMT* promoter methylation status

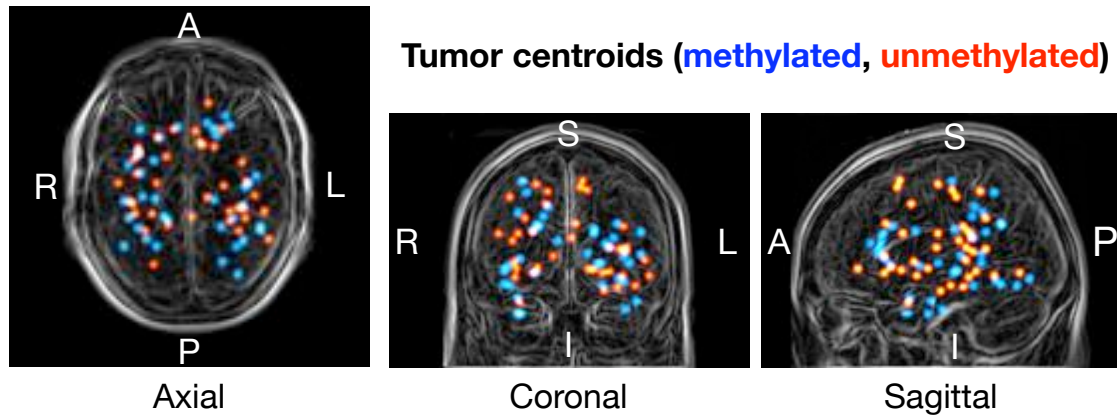


Figure 6.9: Axial, coronal, and sagittal plots of tumor centroids in atlas space for tumors with (36 cases, blue) and without (36 cases, red) *MGMT* promoter methylation status

There was no difference between the median distances from centroids to the middle of the third ventricle for the methylated and unmethylated cases (52.1 mm and 47.6 mm, respectively; $P = 0.43$). Computing distances to points 10 mm and 20 mm

superior to the middle of the third ventricle yielded similar findings ($P = 0.22$ and $P = 0.29$, respectively).

6.5 Discussion

6.5.1 Registration to Normalized Space

There has been substantial work done on the use of non-linear transformations to register images to atlases [8]. We justify our present use of nine-parameter linear scaling transformations, noting that non-linear registration may be considered for future studies. Registration to the MNI target was intended to correct for inter-patient positioning, orientation, and scaling variations. This was done primarily to facilitate analysis of tumor occupancy in gross regions of the brain. Visual inspection of all patient images overlaid on the target following registration confirmed that scaling transformations were sufficient to account for gross positioning and size variations between patients in this study. Had our goal been to precisely segment anatomical structures in the patient images using an atlas [5], non-linear registration would have been required.

Non-linear registration would ideally result in warped patient images that more closely match the anatomical structures of the MNI target. However, the validity of non-linear registration necessitates reliable measures of local image correspondence between the patient and atlas images [137], which was severely limited in this study due to image quality considerations. Approximately one fourth of the images had artifacts due to either patient motion, magnetic gradient inhomogeneity, or magnetic susceptibility. Also, the images were acquired using sequences with relatively low axial resolution (between 3.0 mm and 7.5 mm), which resulted in significant partial voluming in voxels. Due to these artifacts, we opted for a linear transformation model requiring only a global metrics of image correspondence.

6.5.2 Tumor Location

In the absence of a tumor growth model, the centroid was chosen to represent tumor location. Analysis of the tumor spatial distribution was conducted on the centroids in order to discount tumor size. We used the 2D K-S test to analyze location differences among the two tumor genotypes. Specifically, the test determines whether one can disprove the null hypothesis that the two sample datasets are drawn from the same population distribution function [9]. The test is distribution-free, meaning that it makes no assumptions regarding the distribution of the sample population. Rejecting the null hypothesis proves that the datasets are from different distributions. It is impossible to prove that two datasets come from the same distribution. Failure to reject the null hypothesis shows that the datasets can be consistent with the same distribution.

Since the 2D K-S test is only suitable for 2D data, we did not analyze the 3D centroid distributions directly. The centroids were orthogonally projected onto 2D planes prior to analysis. If there were a significant difference between the two centroid

distributions in 3D, we would expect the difference to manifest itself on a 2D marginal projection of the data; however, this is not certain.

Research has been conducted by other authors into multi-dimensional extensions of the K-S test [138, 139]. These methods were initially devised to analyze spatial distributions in astronomical data. These tests would be suitable to investigate the distribution of the centroids directly in 3D space. We expect to implement the 3D K-S test for use in future studies.

6.5.3 Relevance of the Analysis

We examined the topographical distribution of newly diagnosed GBM tumors in relation to *MGMT* status. We asked the questions of whether chemo-sensitive methylated tumors are preferentially located in specific brain regions and whether they have the same topographical distribution as unmethylated ones. If methylation of *MGMT* is an early event in gliomagenesis, a characteristic of a subset of glial cells from which some GBMs arise, or an epigenetic event related to a specific cellular microenvironment, then differences in the locations of methylated and unmethylated tumors might be observed. We further reasoned that methylated and unmethylated GBMs would have similar topographical distributions if *MGMT* methylation status was determined stochastically.

Recent data linking GBMs with neural stem cells [130, 132] and showing that patterns of *MGMT* methylation are uniform in GBMs [133] raised the possibility that methylation of *MGMT* is a characteristic of the cell of origin of a GBM or an early event in tumorigenesis [134]. This led us to hypothesize that GBMs in different regions of the brain could have different patterns of *MGMT* methylation. Instead, we found no association between anatomic location or radial distribution of GBMs and *MGMT* promoter methylation status. These data do not support the hypothesis that methylated and unmethylated GBMs arise from different types of glial cells with different *MGMT* methylation states. Nor could we confirm the report by Eoli et al. [135] that methylated GBMs were preferentially located in the parietal and occipital lobes and that unmethylated tumors were more frequently located in the temporal lobes when assessed visually.

Chapter 7

Conclusion

7.1 Limitations and Future Work

In this section, we highlight potential improvements to our registration framework that address several of its limitations.

7.1.1 Alternative Multi-Modality Similarity Metrics

We propose to include several additional commonly used metrics in future versions of our software. For example, a metric that combines mutual information and image gradients was suggested by Pluim et al. [140]. It is known that MI does not account for spatial information in images. Randomly reshuffling image voxels, for instance, yields an identical MI value. The authors proposed a metric equal to the product of MI and a measure of correlation between the fixed and moving image gradients (similar to NGF). By combining both statistical and spatial information, this metric has been shown to be smoother and to contain fewer incorrect local and global optima than MI. Registration robustness was improved overall for low resolution images with this metric. In our framework, we would compute it by issuing separate metric rendering passes for MI and gradient correlation.

The normalized cross-correlation metric (Eq. 4.3) is invariant to global linear changes in intensity; however, it cannot account for spatially varying intensity distributions. Such uniformity artifacts are common in practice and can affect the NCC metric’s accuracy [77]. A straightforward solution is to define a new metric as the sum of *local* NCC values over many small image neighbourhoods. Parameters for this metric would include the density and size of the neighbourhoods.

Another metric to implement is called the *ratio image uniformity* (RIU), which was originally applied to MRI-PET registration [141]. Given images A and B , we first compute their voxel-wise ratios: $R_1(\mathbf{x}) = A(\mathbf{x})/B(\mathbf{x})$ and $R_2(\mathbf{x}) = B(\mathbf{x})/A(\mathbf{x})$. The metric is based on the assumption that ratios are maximally uniform when A and B are aligned. We compute the metric as the standard deviation of the ratio images divided by their means: $\text{RIU}(A, B) = \sigma_{R_1}/\mu_{R_1} + \sigma_{R_2}/\mu_{R_2}$. For ideal alignment, it is assumed that the ratios for corresponding points vary little, thus minimizing the metric.

7.1.2 Partial Volume Interpolation

The choice of a high quality interpolation model is important, especially in computing mutual information. As discussed in section 4.1.5, our framework uses partial volume interpolation (PVI), which updates histogram bins using weights from a trilinear interpolation kernel. We propose to implement a more generalized form of PVI that uses bilinear weights in the image plane and cubic B-spline weights in the slice direction [142, 143]. Compared to the trilinear kernel, this method is reported

to further reduce interpolation artifacts in the MI metric, such as local optima and cusps. It has also been shown to significantly improve clinical registration accuracy and robustness.

7.1.3 Parzen Windowing

Mutual information is based on the joint image density distribution. However, the discrete nature of digital images impedes exact calculation of the joint density—it is estimated from the joint histogram [107]. In our implementation of MI, each co-occurring pair of image samples (a, b) is used to increment a single histogram bin, $h(a, b)$. An alternative approach for density estimation is called Parzen windowing, whereby a neighbourhood of bins centred at $h(a, b)$ is updated using a 2D windowing function. This leads to continuous estimates of the distribution, thus minimizing the effects of interpolation and discretization from binning [103]. If only a subset of image voxels are sampled, then Parzen windowing is crucial for generating reliable estimates of the joint probability density function [144]. In this case, the window width scales inversely with number of sampled voxels. Parzen windowing has been shown to improve both rigid and non-linear registration accuracy [107, 143].

We propose to efficiently implement Parzen windowing using *point sprite* rendering. Point sprites are textured, view-aligned quadrilaterals that are drawn to the frame buffer by issuing a single 3D point render call [25]. We would store the window function (commonly modeled as splines that partition unity [103]) in a 2D texture. During histogram generation, each scattered vertex would be rendered as a sprite with this texture, thereby superimposing the window on a neighbourhood of histogram bins.

7.1.4 Registration using Raycasting

In section 4.1.1, we discussed the semblance of our registration methods to a 3D visualization technique called *texture-based volume rendering*. On each iteration, we compute the metric by rendering a sequence of view-aligned quads to the frame buffer. These quads are textured with slices of the fixed and moving images, the latter being transformed via the texture coordinate matrix. Custom shader programs are enabled to compute the metric images as the geometry is rendered.

We have also implemented a prototype registration application that is based on a fundamentally different method of 3D visualization called *volume raycasting* [42, 145]. Figure 7.1 illustrates how the fixed and moving image volumes are sampled in the raycasting method. The algorithm is implemented as a loop over all frame buffer pixels in a custom fragment shader.

Two rays emanate from each pixel of the view plane (i.e. frame buffer) and simultaneously traverse the fixed and moving images. The images are sampled at their intersections with the rays at equidistantly spaced steps. The sampled intensities along the ray paths are used to compute the metric values, which are composited in the frame buffer. The fixed image is sampled front-to-back, whereas the moving image is sampled by rays oriented according to the current transformation parameters, as

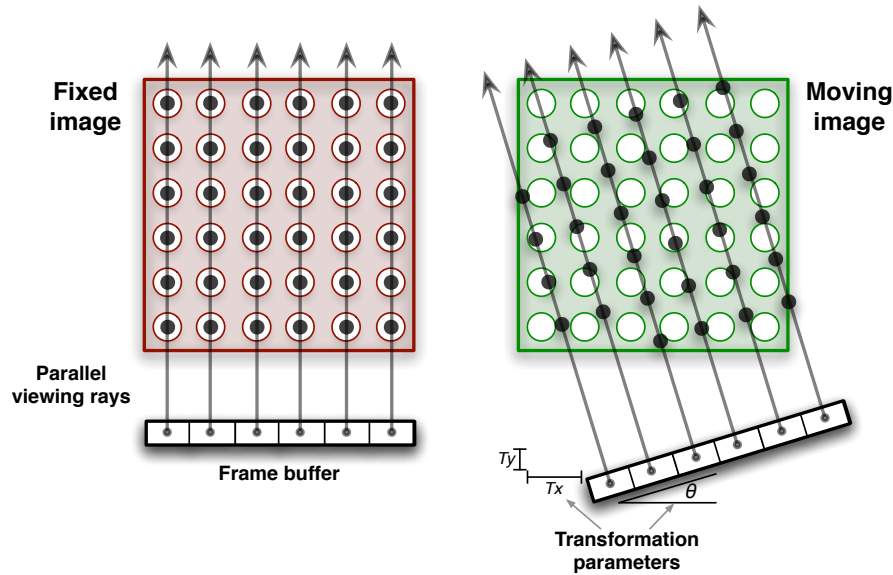


Figure 7.1: Two-dimensional schematic of using volume raycasting to sample the fixed and moving images in registration

shown in the figure above. Trilinear interpolation is used when sampling the moving image, which may not be aligned with the ray paths.

This new raycasting technique offers several advantages over *texture-based* registration. First, only one quadrilateral rendering call is needed, since the traversal of each ray is done in a loop that executes in the fragment shader. The texture-based method requires us to render one quad per image slice. Fewer rendering calls translates to less state setup and overhead by OpenGL.

Second, raycasting greatly simplifies affine image transformation. The ray's sampling frequency and path direction implicitly specify the transformation parameters. These values are set once per ray, whereas our other method required a multiplication with the moving texture coordinate matrix for each voxel.

More sophisticated traversal of the images is also simple to implement with raycasting. Empty image regions that do not contribute to the similarity metric can be efficiently excluded from processing. Such empty-space skipping can be implemented using an octree hierarchy of the images [145]. Preliminary experiments show that registration using raycasting for affine registration is approximately 15% faster than the current method when run on the GeForce 8800 GT GPU.

7.1.5 Symmetric and Diffeomorphic Transformations

We address limitations of the Demons algorithm presented in section 4.2 that concern symmetry and invertibility of the transformation. Intuitively, the problem of image registration is symmetric, since correspondences between images should not depend on the order in which they are compared. However, the displacement field update

force of Equation 4.10 is asymmetric in the two input images, because it is driven by gradient information from the fixed image alone. This leads to an inherent bias in registration, since one obtains different results depending on which image is arbitrarily specified as “fixed” and which is specified as “moving”. We propose to remove this bias using forces that are symmetric in the two image gradients ($\vec{\nabla}F(\mathbf{x})$ and $\vec{\nabla}M(\mathbf{x})$) and that also tend to zero when the gradients are dissimilar [112, 146].

The goal of deformable registration is ultimately to match corresponding anatomy between images. In order to guarantee that topology is preserved between homologous structures, the transformation functions must be diffeomorphic. That is, an admissible free-form transformation $\mathbf{T}$ must be differentiable and have a differentiable inverse $\mathbf{T}^{-1}$. Differentiability implies both continuity and a certain smoothness of the transformation. These properties ensure that connected structures remain connected and that structure boundary smoothness is preserved. The property of bijectivity (having a one-to-one inverse) ensures that disjoint structures remain disjoint. It is also required for the transformations to be truly symmetric.

In his original publication on Demons, Thirion gives a method of ensuring bijectivity that proves more robust at preserving homologous organ structure [20]. The method constructs the forward and inverse transformations simultaneously, ensuring their consistency by distributing the residual $\mathbf{T}_i^{-1} \circ \mathbf{T}_i$ over the iterative estimates. Thirion implicitly relies on Gaussian smoothing to maintain differentiability of the transforms. Rigorous studies of topology-preserving diffeomorphic registration have been studied more recently by researchers such as Beg, et al. [147] and Avants, et al. [148].

7.2 Concluding Remarks

Throughout the past decade, explosive growth in the video game and entertainment industries has driven dramatic advances in graphics processing hardware technology. Consequently, GPUs have become widely available, inexpensive, and powerful computational engines. Their rendering rates (in pixels per second) have approximately doubled every six months over the last decade [17, 149]. Scientific algorithms adapted to leverage this highly parallel processing environment can benefit from significant acceleration.

We have developed novel approaches to 3D image registration using the GPU. In our programs, all steps with high computational and memory bandwidth requirements run on the GPU. Thus, significant computations occur using its high bandwidth architecture. Our new methods are significantly faster than traditional CPU-based registration methods, yet are comparable in accuracy.

Our framework allows automatic, 3D alignment of multi-modal clinical data to sub-voxel accuracy within seconds. This is due to our novel implementation of the mutual information metric. We also implemented 3D deformable registration at near real-time rates. Since image transformation and similarity metric computation are implemented using the rendering pipeline, our tool provides visual feedback of the moving image and metric during the registration process. This allows interactive

inspection of the data and simplifies the fine-tuning of parameters often required to achieve successful registration. We believe that our intuitive interface and improved speed performance will make registration more accessible to clinicians and researchers.

Appendix A

Graphics Rendering Pipeline

The graphics rendering pipeline forms the backbone of our accelerated image registration application. In this section, we give an overview of the pipeline and highlight specific features used in our work.

Data enters the pipeline as a 3D scene representation in the form of a list of geometric primitives. The primitives are mapped to the screen and shaded by the pipeline, resulting in the synthesis of a 2D raster image of the scene. Figure A.1 shows a schematic diagram of the processing pipeline. In real-time computer graphics applications, images are commonly synthesized at rates of about 60 frames per second [27].

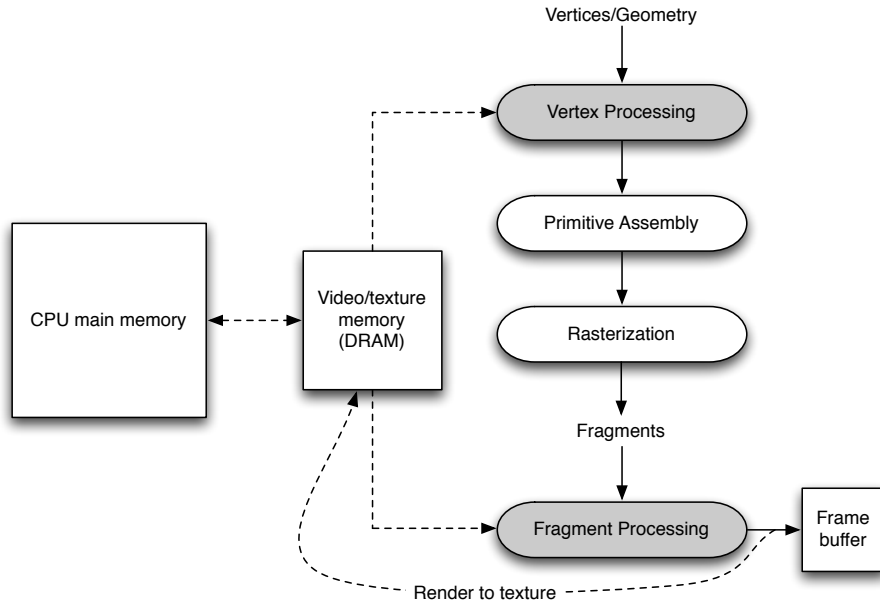


Figure A.1: Schematic of the GPU pipeline, with programmable elements shown shaded

All modern graphics hardware executes the same pipeline processing steps that we shall discuss. Scene setup and interaction are usually done through a graphics application programmer interface (API). Graphics APIs facilitate 3D programming by providing a standard interface for interacting with different types of GPUs. The two most widely supported graphics APIs are OpenGL [25] and Direct3D [150]. OpenGL is a cross-platform specification managed by a non-profit consortium; Direct3D is managed by Microsoft and is only available for Windows operation systems. Both have undergone multiple substantial revisions since they were introduced in 1992. These two APIs provide nearly identical functionality and performance, though their

syntaxes and nomenclatures differ. Since we implemented our application using the OpenGL API, our discussion follows OpenGL naming conventions.

The graphics pipeline is required to have the same functionality in every implementation of OpenGL. That is, given a set of inputs, the output must be consistent with API specification. The sequence of steps that we discuss is therefore also referred to as the *fixed-functionality* graphics pipeline.

A.1 Geometry Specification

Input to the graphics pipeline consists of geometric primitives that are constructed from vertices. Primitives can be either points, lines, or planar polygons, and they are used to model the structure of objects to be rendered. The pipeline’s final output consists of colour values of the primitives’ pixels, mapped to screen coordinates. Many parameters are accessed throughout the pipeline that affect the final rendering, such as light source characteristics, surface reflectance characteristics, surface textures, and camera positioning. Most OpenGL commands either configure state variables that hold these parameters or submit primitives to the pipeline.

Three-dimensional scenes often contains multiple objects, each of which may have a hierarchical structure of components. In order to conveniently describe a scene, the vertices of an object are therefore often defined in a coordinate system that is local to the object, called *model space*. The origin is usually assigned to a sensible reference location on the object, and the coordinate axis orientation and scale are defined to correspond with the object position and size. In OpenGL, the 3D vertex positions are expressed as four-tuples in homogenous coordinates. Homogeneous coordinates allow affine and projective transformations to be applied to points using matrix multiplication.

In addition to 3D position, vertices can be assigned arbitrary attribute variables. OpenGL’s built-in vertex attributes include the colour and normal vector, for example. This vertex data is created by the software application and resides in application-controlled memory on the CPU or GPU before being sent into the graphics pipeline. There are different ways of sending vertex data into the pipeline. The application can issue a separate function call to submit each vertex attribute. However, this method results in significant overhead for the submission of many vertices. When performance is critical, it is more convenient to organize a large number of vertex attributes into *vertex arrays*, then to submit pointers to these arrays for processing. Fewer function calls are necessary and graphics APIs can more efficiently process data organized in arrays [28].

If the same set of vertices is rendered multiple times, then the highest performance is obtained by storing the set directly in graphics hardware memory. This eliminates data transfers over the bus between the CPU and GPU for each rendering pass. Our registration application submits up to millions of vertices for each iteration of the mutual information similarity metric; therefore, we store vertex arrays on the GPU.

A.2 Vertex Transformation and Lighting

The first stage of the graphics pipeline is called *vertex processing*. In this stage, all vertices from the scene are processed independently by a parallel array of processors. Per-vertex operations include applying geometric transformations to positions and computing scene lighting interactions with colours. For this reason, the stage is also referred to as *transformation and lighting*. Figure A.2 shows a schematic example of vertex processing on a mesh with quadrilateral geometry. In this example, vertex processors apply lighting to the vertices and warp their positions.

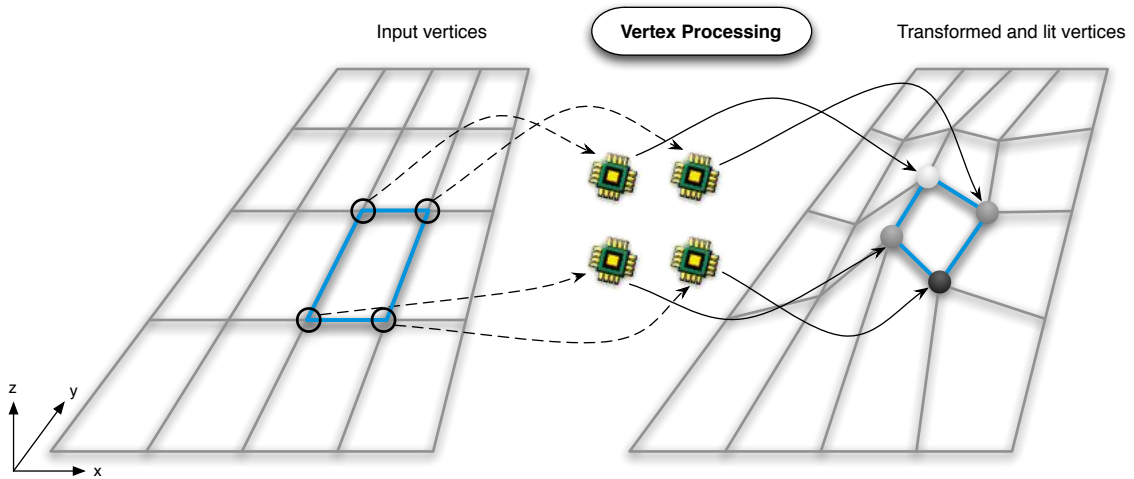


Figure A.2: The vertex processing stage of the graphics pipeline applies transformation and lighting to vertices

The GPU computes the colour of each vertex by integrating contributions from the scene's light sources. The Phong lighting equation, for instance, is commonly used to model the appearance of plastic surfaces in virtual scenes [25]. It models the vertex colour as a combination of an ambient scene colour, a diffuse material colour, and specular highlights from point light sources. Lighting and material parameters are set using OpenGL state variables by the software application.

Another critical step in vertex processing is the transformation of all vertices from their local model space coordinates into *world space*. World space coordinates are common to all elements of the scene, including objects, lights, and the view camera. This step is referred to as the *model transformation*, and it is accomplished by multiplying each set of model space coordinates with an affine matrix. We note that affine matrix transformations are commonly used in computer graphics to rotate, translate, scale, and shear objects. These transformations preserve proportional distances between points and the parallelism of lines.

Following model transformation, vertices must be transformed into *eye space* according to the camera's viewing parameters. This step is called the *view transformation* and is also represented by an affine matrix. Parameters of the transformation

include the camera position, orientation, and viewing direction. The model and view transformations are often concatenated into a single *model-view transformation* matrix, which takes coordinates directly from model space into eye space.

In order to view the 3D scene, it must be projected into the 2D plane of the view camera. This step is called the *projection transformation*. Perspective and orthogonal projections are commonly used in computer graphics for this purpose. Perspective projections account for the distance of points from the view plane by foreshortening parallel lines to produce vanishing points at infinity. As implied by the name, orthogonal projections map objects orthogonally onto the view plane, preserving parallel lines. Using homogeneous coordinates, the transformation of a vertex into the view plane is also accomplished by a single matrix multiplication. Following the projection, all coordinates are scaled to correspond with the display screen's integer sampling locations.

Aside from colour and position, any other vertex attribute can be modified during vertex processing. Two noteworthy attributes that usually require transformation are the vertex normal vector and vertex *texture coordinates*. Normal vectors orient object surfaces for lighting calculations, whereas texture coordinates are used to map images onto surfaces. Our GPU-accelerated registration application uses texture mapping in order to transform medical images stored as textures on the GPU.

Following vertex processing, vertex positions are in screen coordinates and all other vertex attributes are finalized. Geometric primitives are assembled at this point by applying user-defined connectivity information to the vertices. The primitives are then clipped against scene viewing planes and rasterized. *Rasterization* is the process of determining the set of screen pixels that fall within the boundaries of each primitive, as illustrated in Figure A.3. The screen pixels of a primitive are also referred to as its *fragments*. Each fragment output from the rasterization stage has a collection of attributes that are modified in the graphics pipeline's subsequent fragment processing stage.

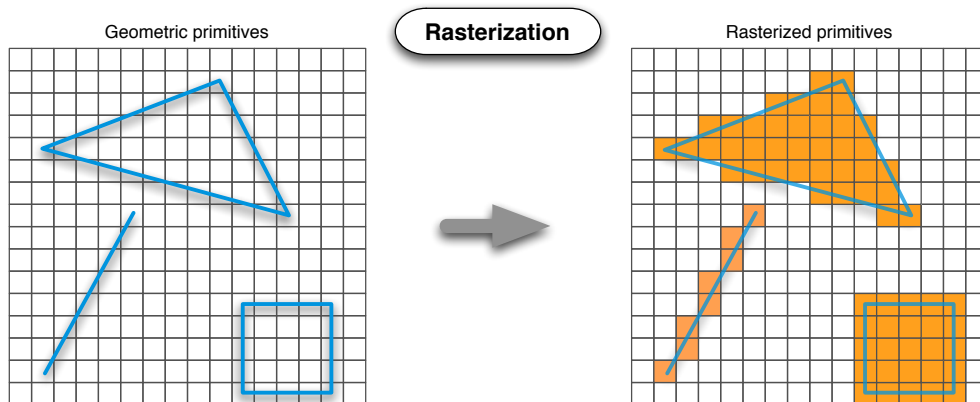


Figure A.3: Rasterization of geometric primitives into screen fragments

A.3 Fragment Operations and Texturing

The next stage of the pipeline is *fragment processing*. In this stage, all rasterized fragments are processed independently by a parallel array of GPU processors. Each fragment is assigned a colour, then sent to the screen's frame buffer for display.

Figure A.4 illustrates the shading of rasterized fragments over a polygon. As shown in this example, the fragment colours of a geometric primitive are usually determined by interpolating the primitive's vertex colours that were set by integrating scene lighting.

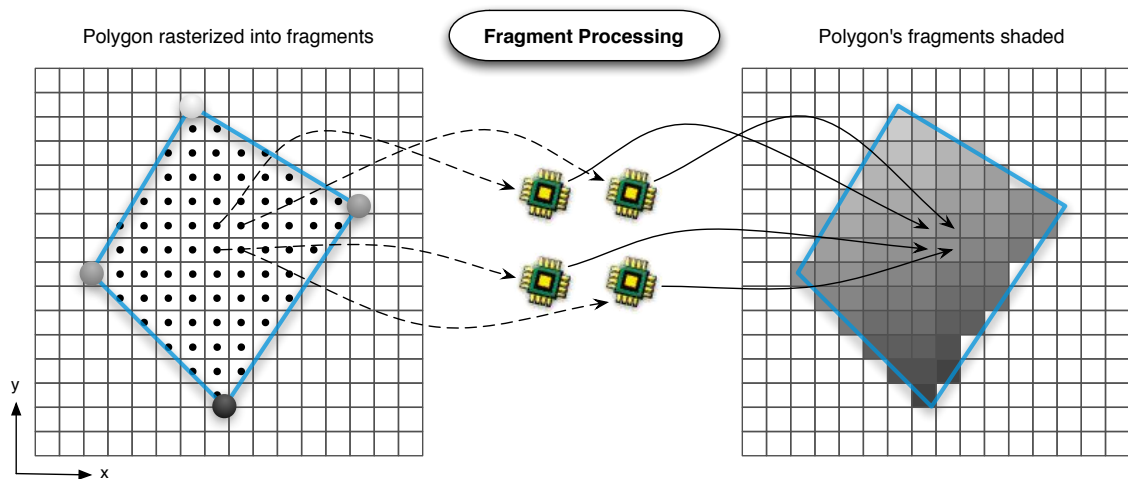


Figure A.4: The fragment processing stage of the graphics pipeline applies colours to fragments

In order to achieve added rendering realism, fragments can also be coloured by mapping images over primitives, as mentioned in the previous section. These images are called textures, since they can give the impression of added detail to a scene.

Textures are the main storage structure accessible in the GPU's rendering pipeline. They are arrays of colour values that can have up to three spatial dimensions. Each texture array element is a vector that stores red, green, and blue intensities, and (optionally) an alpha channel value. The alpha channel generally represents the fragment transparency. Two-dimensional textures are most often used in traditional rendering applications, since they naturally map onto the surfaces of objects. In our registration application, we use 3D textures to store the medical image datasets.

Mapping a texture over the rasterized fragments of a primitive is accomplished using texture coordinates: Each vertex is assigned a coordinate that serves as an index into the texture, as shown in Figure A.5. These coordinates are interpolated over the fragments following rasterization. The colour value assigned to a particular fragment is then obtained by a texture look-up at the fragment's interpolated texture coordinate.

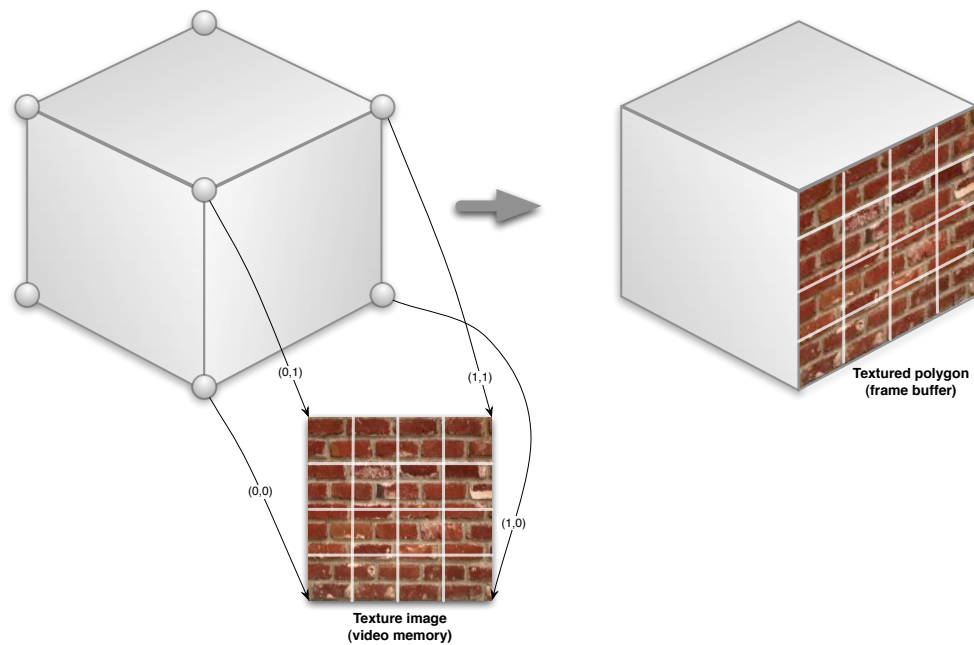


Figure A.5: Mapping of textures onto polygons is defined by texture coordinates at the polygon vertices

Efficient handling of textures by the hardware is critical, because millions of fragments may need to be textured at high frame rates in real-time applications. Fortunately, specialized caching of textures reduces access latency [27]. This caching is made possible by very regular texture access patterns. Indeed, the fragments of the frame buffer are processed in a raster pattern, and adjacent fragments generally only access adjacent texture elements.

Linear interpolation of textures is also implemented in the graphics hardware. Even though linear interpolation in 3D requires up to eight texture reads, it shows no measurable computational overhead compared to nearest-neighbor interpolation on the GPU [98]. Our registration application makes use of this efficient texture mapping and interpolation in order to quickly transform large image datasets.

A.4 Frame Buffer Rendering

The region of memory that is allocated for holding a frame of display contents is called the *frame buffer*. This memory is stored in the graphics hardware and it is updated when rendering to the display. The display hardware reads the entire frame buffer contents every time it requires refreshing.

Each element of the frame buffer is a vector that holds the colour channels of a pixel. Precision ranges from one byte per channel to double precision on the latest hardware and in the latest specification of OpenGL (version 4.0, released in March 2010). Double buffering is a method that uses an offscreen frame buffer in order to

achieve smooth animation. In this method, the application renders into an offscreen *back buffer* while displaying the visible *front buffer*. The two buffers are swapped following rendering, ensuring that graphics are never displayed while being rendered.

It is sometimes necessary to store frame buffer contents for input to subsequent rendering operations. For this reason, the buffer's contents can be transferred to GPU texture memory or downloaded to the software application on the CPU. The graphics pipeline's output can be also be rendered directly into texture memory, allowing it to be easily accessed in subsequent pipeline iterations. Our registration application renders into offscreen textures when computing image metrics and deformation fields. Because no image is physically displayed on screen, it takes less time to render to a texture than to the main frame buffer.

Blending of colours in the frame buffer is powerful operation that can be used to simulate translucency of materials. By default, incoming fragment colours overwrite previous colour values at the same screen coordinates. Blending combines the incoming fragment colour with the destination colour according to a predefined blending function. Our application uses colour blending operations in order to sum the image similarity metric.

In practice, it is usually unnecessary to render all fragments from a scene, since some are obscured from view by primitives. Immediately following fragment processing, each fragment is therefore put through several logical tests to determine whether it should be rendered. For instance, a depth buffer with equal dimensions to the frame buffer stores the distance from each fragment to the viewing plane. The GPU compares each incoming fragment's distance to the depth buffer value, and it only writes a fragment to the frame buffer if it is closer to the view plane than the previously written fragment at that same location.

Masking of fragments is also performed just prior to frame buffer rendering. It is possible to exclude arbitrary fragments from rendering by defining a binary mask in the *stencil buffer*. This method can be used to speed up registration iterations by excluding certain voxels from image metric computation in our application. For example, the stencil buffer could mask voxels that are within the brain of the subject being registered.

Appendix B

Shader Programming

In the sections that follow, we provide a brief overview of the structure of vertex and fragment shaders, giving introductory examples of using them for custom rendering and general-purpose computation. This is intended as prerequisite knowledge for our work on GPU-accelerated image registration.

B.1 Vertex Shaders

Vertex shaders are custom programs that execute on vertex processors at the lighting and transformation stage of the graphics pipeline. This stage of the pipeline is executed by processors that take in and modify vertex attributes, such as colour and position, by applying lighting and spatial transformations. Multiple vertex processors execute in parallel on unique vertices, and all vertices in the scene are processed once and independently of one another. Computations in this stage rely heavily on vector operations that are efficiently implemented in the graphics hardware.

Vertex shader programs must adhere to the interface defined for vertex processing. They can modify the values of vertex attributes, but they cannot create or delete vertices. Vertex shaders do not have access to the geometric primitive containing the current vertex, since all connectivity is integrated in a subsequent pipeline stage. However, vertex shaders are permitted to implement an entirely custom set of operations than is defined by the fixed-function pipeline.

Vertex shaders work with three different variable types: attribute variables, uniform variables, and varying variables. *Attribute variables* are used to hold floating-point scalar, vector, and matrix data that pass from the software application to the vertex processor and that are modified at a relatively frequent rate—up to once per vertex. As implied by their name, these variables are specifically designed to hold vertex attribute values, such as the vertex colour (`gl.Color`), position (`gl.Vertex`), and normal vector (`gl.Normal`). Besides these examples of built-in attribute variables, the application is also able to declare arbitrary per-vertex data as attribute variables.

Values that change at most once per primitive are stored in *uniform variables*. These variables are typically used to pass parameters and current OpenGL state to the vertex shader, such as lighting (e.g. `gl.LightSource`), surface reflection characteristics (e.g. `gl.FrontMaterial`), and transformation matrices (e.g. `gl.ModelViewMatrix`, `gl.ProjectionMatrix`, `gl.TextureMatrix`). Uniform variables cannot be modified within a shader, since they are shared across vertex processors.

All data is transferred from vertex to fragment processing through *varying variables*. These variables are chosen for data values that vary from vertex to vertex in a primitive and that require interpolation across the primitive's fragments. There are several built-in varying variables that naturally require interpolation across fragments. Built-in varying variables include colours, normal vectors, and texture coordinates.

Interpolated normal vectors, for instance, are useful for generating advanced lighting effects that vary across fragments.

In addition to these data types, the vertex shader can read from texture memory. This is a powerful feature, because it greatly increases the number of parameters used to transform vertices [33]. Our registration application uses this functionality in order to generate joint intensity histograms of images stored as textures. This enables us to compute the mutual information similarity metric on the GPU. We discuss this application further in the next chapter.

The vertex shader must compute the output position of the vertex being processed. This is the only requirement of a valid vertex shader. The coordinates of the vertex are stored in the special output varying variable `gl_Position` as homogeneous coordinates in eye space.

Immediately following vertex shading there is another customizable stage called *geometry shading*. Geometry shader programs operate on primitives (i.e. vertices and their connectivity information). These shaders can modify geometry by adding and removing vertices from primitives. They can also remove primitives or create entirely new primitives on the fly, which are subsequently rasterized and processed by the fragment shader.

B.2 Fragment Shaders

We described fragment operations in section A.3. At this stage of the pipeline, fragment processors assign colours to fragments that were generated by the rasterization of points, lines, and polygons. *Fragment shaders* are custom programs that execute on fragment processors.

The main inputs to fragment shaders are varying variables from the preceding vertex processing stage. These variables are automatically interpolated during rasterization across all screen fragments. Fragment shaders also have read access to the current fragment's 2D screen coordinates (`gl_FragCoord`). They cannot modify the output coordinates of fragments, which contrasts with the ability of vertex shaders to modify the positions of vertices.

Texture memory can be accessed by fragment shaders any number of times. In traditional graphics applications, this functionality is used to map images over the screen fragments. In GPGPU applications, this feature is often used for *dependent texture look-ups*, whereby the result of one texture read is used as input for a subsequent reads. We use this functionality to implement complex algorithms, such as image warping using deformation fields [76].

As output, the fragment shader can assign to each fragment a colour (`gl_FragColor`) and a depth coordinate (`gl_FragDepth`), or it can discard the fragment so that it is not rendered.

B.3 Custom Graphics Shader Examples

In this section, we give an example pair of GLSL vertex and pixel shaders in order to familiarize the reader with concepts that we have discussed. These shaders demonstrate how to apply basic *per-fragment lighting* and *texture mapping* to a surface.

Light is modeled in this example using diffuse and specular components, as defined by the Phong lighting equation [25]. The diffuse light component is computed assuming that its perceived intensity is independent of the viewer's position: $I_d = L_d M_d \cos \theta$, where L_d is the light's diffuse colour, M_d is the material's diffuse reflectivity coefficient, and θ is the angle between the light direction and the surface normal. These vectors are illustrated in Figure B.1. The diffuse light is computed once per vertex, then it is interpolated across each fragment.

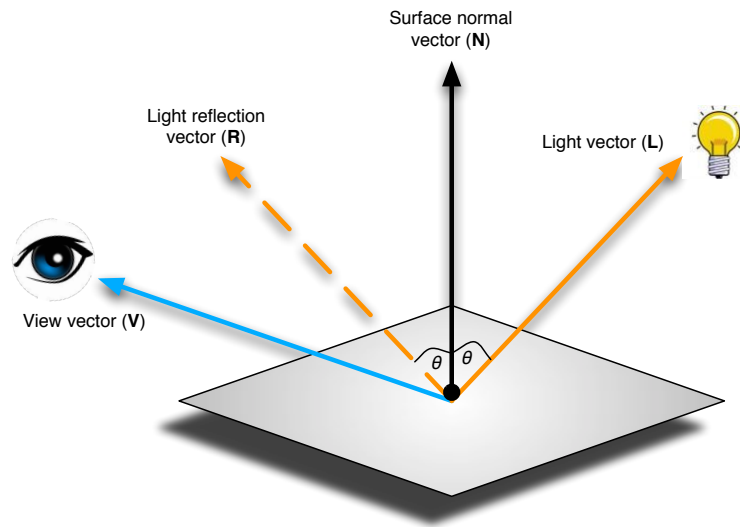


Figure B.1: Vectors used in the Phong lighting model

The specular light component is computed separately for each fragment, taking the view vector into account: $I_s = L_s M_s (\mathbf{R} \cdot \mathbf{V})^s$, where L_s is the light's specular colour, M_s is the material's specular reflection coefficient, $\mathbf{R}$ is the reflection vector of the incident light on the surface, $\mathbf{V}$ is the vector from the surface to the viewer, and s is a positive constant affecting surface shininess. The final fragment colour is computed by multiplying the total light intensity ($I_d + I_s$) by a stored texture value. We emphasize that *specular lighting computations are done separately for each fragment*. This would not be possible using the fixed pipeline, which only computes lighting interactions on a per-vertex basis.

The vertex shader is given in Listing B.1. The point light source direction and colours are passed into the shader as uniform vector variables (lines 0-1). The identifier `vecn` denotes a vector with n floating-point elements. The view vector (from vertex to origin), surface normal vector, light direction vector, and diffuse light component are declared as varying variables, since they will be passed from the vertex

shader to the fragment shader (lines 3-4).

```

0 uniform vec3 lightPosition;
  uniform vec4 lightDiffuseColour, lightSpecColour;
2
  varying vec3 viewVector, normalVector, lightDirection;
4 varying vec4 diffuseLight;

6 void main()
  {
8     vec3 vertexPosition = vec3(gl_ModelViewMatrix * gl_Vertex);
      normalVector = normalize(gl_NormalMatrix * gl_Normal);
10     viewVector = normalize(-vertexPosition);
      lightDirection = normalize(lightPosition - vertexPosition);
12
      diffuseLight = lightDiffuseColour * gl_FrontMaterial.diffuse
14                    * max(dot(lightDirection, normalVector), 0.0);

16     gl_TexCoord[0] = gl_MultiTexCoord0;
      gl_Position = gl_ModelViewMatrix * gl_Vertex;
18 }

```

Listing B.1: Sample GLSL vertex shader demonstrating per-fragment lighting and texture mapping

The shader begins with several vector operations. First, the vertex position and normal vector are transformed from model space into eye space by multiplying by the model-view matrix (line 8) and its transpose (line 9), respectively. All vectors are normalized to unit length (lines 9-11) before computing the diffuse light contribution (line 13). (Multiplication of vectors is carried out point-wise on their elements.) Next, the vertex's texture coordinate attribute (set by the calling application) is passed into the varying variable `gl_TexCoord[0]` (line 16). Finally, we transform the incoming vertex position into eye space in a manner that reproduces the fixed pipeline's functionality (line 17).

```

0 uniform sampler2D myTexture;
  varying vec3 viewVector, normalVector, lightDirection;
2 varying vec4 diffuseLight;
  const float s = 10.0;
4
  void main()
  {
6     vec3 N = normalize(normalVector);
8     vec3 V = normalize(viewVector);
      vec3 L = normalize(lightDirection);
10     vec3 R = reflect(-L, N);
      vec4 specularLight = lightSpecColour * gl_FrontMaterial.shininess
12                    * pow(dot(R, V), s);
      vec4 lightIntensity = diffuseLight + specularLight;
14

      vec3 textureColour = vec3(texture2D(myTexture, gl_TexCoord[0].st));
16     gl_FragColor = vec4(textureColour, 1.0) * lightIntensity;
  }

```

Listing B.2: Sample GLSL fragment shader demonstrating per-fragment lighting and texture mapping

The corresponding fragment shader is given in Listing B.2. First, an object is defined in order to sample the 2D texture image (line 0) and the varying variables from the vertex shader are redeclared (lines 1-2). The interpolated normal, view, and light direction vectors are renormalized to unit length in the fragment shader (lines 7-9). Next, the reflection vector of the light ray with respect to the normal is computed with a built-in GLSL function (line 10) and used to generate the specular light contribution (line 11). The texture is sampled using the interpolated 2D texture coordinates `gl_TexCoord[0].st` (line 15). (The selector `.st` instructs the shader to sample the x and y coordinates of the texture.) The output fragment colour is equal to the sampled texture colour modulated by the overall light intensity (line 16).

Bibliography

- [1] N. Archip, O. Clatz, S. Whalen, D. Kacher, A. Fedorov, A. Kot, N. Chrisochoides, F. Jolesz, A. Golby, P. M. Black, and S. K. Warfield. Non-rigid alignment of pre-operative MRI, fMRI, and DT-MRI with intra-operative MRI for enhanced visualization and navigation in image-guided neurosurgery. *NeuroImage*, 35(2), 2007.
- [2] M. Brett, I. S. Johnsrude, and A. M. Owen. The problem of functional localization in the human brain. *Nature Reviews Neuroscience*, 3:243–249, 2002.
- [3] J. V. Hajnal, D. L. G. Hill, and D. J. Hawkes. *Medical Image Registration*. CRC Press, Boca Raton, 2001.
- [4] P. Hastreiter, C. Rezk-Salama, G. Soza, M. Bauer, G. Greiner, R. Fahlbusch, O. Ganslandt, and C. Nimsy. Strategies for brain shift evaluation. *Medical Image Analysis*, 8(4):447–464, 2004.
- [5] A. C. Evans, D. L. Collins, S. R. Mills, E. D. Brown, R. L. Kelly, and T. M. Peters. 3D statistical neuroanatomical models from 305 MRI volumes. In *Nuclear Science Symposium and Medical Imaging Conference*, volume 3, pages 1813–1817, 1993.
- [6] J. H. Morraa, Z. Tu, L. G. Apostolova, A. E. Green, C. Avedissian, S. K. Madsen, N. Parikshak, A. W. Toga, C. R. Jack, N. Schuff, M. W. Weiner, and P. M. Thompson. Automated mapping of hippocampal atrophy in 1-year repeat MRI data from 490 subjects with alzheimers disease, mild cognitive impairment, and elderly controls. *Neuroimage*, 45(1S), 2009.
- [7] J. West, J. M. Fitzpatrick, M. Y. Wang, B. M. Dawant, C. R. Maurer, Jr., R. M. Kessler, and R. J. Maciunas. Retrospective intermodality registration techniques for images of the head: surface-based versus volume-based. *IEEE Transactions on Medical Imaging*, 18(2):144–150, 1999.
- [8] J. B. A. Maintz and M. A. Viergever. A survey of medical image registration. *Medical Image Analysis*, 2(1):1–36, 1998.
- [9] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1992.
- [10] T. Rohlfing, C. R. Maurer, Jr., D. A. Bluemke, and M. A. Jacobs. Volume-preserving nonrigid registration of MR breast images using free-form deformation with an incompressibility constraint. *IEEE Transactions on Medical Imaging*, 22(6):730–741, 2003.
- [11] 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 Transactions on Medical Imaging*, 18(8):712–721, 1999.
- [12] H. Lester and S. R. Arridge. A survey of hierarchical non-linear medical image registration. *Pattern Recognition*, 32(1):129–149, 1999.
 - [13] J. P. W. Pluim, J. B. A. Maintz, and M. A. Viergever. Mutual-information-based registration of medical images: a survey. *IEEE Transactions on Medical Imaging*, 22(8):986–1004, 2003.
 - [14] M. L. Kessler and M. Roberson. Image registration and data fusion for radiotherapy treatment planning. In W. Schlegel, T. Bortfeld, and A.-L. Grosu, editors, *New Technologies in Radiation Oncology*, Medical Radiology. Springer, Heidelberg, 2006.
 - [15] F. Maes, F. Vandermeulen, G. Marchal, and P. Suetens. Clinical relevance of fully automated multimodality image registration by maximization of mutual information. In *Proc. of the Image Registration Workshop*, pages 323–330, November 1997.
 - [16] F. Ino, K. Ooyama, and K. Hagihara. A data distributed parallel algorithm for nonrigid image registration. *Parallel Computing*, 31(1):19–43, 2005.
 - [17] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, pages 565–575, Jan 2009.
 - [18] P. Viola and W. M. Wells, III. Alignment by maximization of mutual information. *International Journal of Computer Vision*, 24(2), 1997.
 - [19] C. Studholme, D. L. G. Hill, and D. J. Hawkes. An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition*, 32(1):71–86, 1999.
 - [20] J.-P. Thirion. Image matching as a diffusion process: an analogy with maxwell’s demons. *Medical Image Analysis*, 2(3):243–260, 1998.
 - [21] P. Cachier, X. Pennec, and N. Ayache. Fast non rigid matching by gradient descent: study and improvements of the “demons” algorithm. Technical Report 3706, Institut National de Recherche en Informatique et en Automatique (INRIA), June 1999.
 - [22] S. K. Warfield, F. A. Jolesz, and R. Kikinis. A high performance computing approach to the registration of medical imaging data. *Parallel Computing*, 24(9-10):1345–1368, 1998.
 - [23] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

- [24] T. Dokken, T. R. Hagen, and J. M. Hjelmervik. The GPU as a high performance computational resource. In *Proc. of the 21st spring conference on Computer graphics (SCCG)*, pages 21–26. ACM, 2005.
- [25] D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. Addison-Wesley Professional, 2009.
- [26] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008.
- [27] D. Luebke and G. Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.
- [28] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley, Boston, 2004.
- [29] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004.
- [30] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1):60–99, 2006.
- [31] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture*, June 2008. Programming Guide 2.0.
- [32] M. Harris. Mapping computational concepts to GPUs. In M. Pharr and R. Fernando, editors, *GPU Gems 2*, pages 493–508. Addison-Wesley, 2005.
- [33] P. Gerasimov, R. Fernando, and S. Green. *Shader Model 3.0: Using vertex textures*. NVIDIA Corporation, June 2004. Whitepaper.
- [34] M. Harris and I. Buck. GPU flow-control idioms. In M. Pharr and R. Fernando, editors, *GPU Gems 2*, pages 547–556. Addison-Wesley, 2005.
- [35] T. R. Halfhill. *Looking Beyond Graphics*. NVIDIA Corporation, Sept 2009. Whitepaper.
- [36] Khronos Group. *OpenCL: Parallel Computing for Heterogeneous Devices*, Dec 2009. Whitepaper.
- [37] P. N. Glaskowsky. *NVIDIA’s Fermi: The First Complete GPU Computing Architecture*. NVIDIA Corporation, Sept 2009. Whitepaper.
- [38] N. Brookwood. *NVIDIA Solves the GPU Computing Puzzle*. NVIDIA Corporation, Sept 2009. Whitepaper.
- [39] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

- [40] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science*, 52(3):654–663, 2005.
- [41] S. Schenke and B. C. Wünsche. GPU-based volume segmentation. In *Image and Vision Computing New Zealand*, November 2005.
- [42] H. Scharsach. Advanced GPU raycasting. In *Central European Seminar on Computer Graphics*, 2005.
- [43] Y. Kamitani and F. Tong. Decoding the visual and subjective contents of the human brain. *Nature Neuroscience*, 8(5):679–685, 2005.
- [44] E. I. Zacharaki, D. Shen, S.-K. Lee, and C. Davatzikos. A multiresolution framework for deformable registration of brain tumor images. *IEEE Transactions on Medical Imaging*, 27(8), 2008.
- [45] N. J. Tustison, S. P. Awate, J. Cai, T. A. Altes, G. W. Miller, E. E. de Lange, J. P. Mugler, and J. C. Gee. Pulmonary kinematics from tagged hyperpolarized helium-3 MRI. *Journal of Magnetic Resonance Imaging*, 31(5), 2010.
- [46] A. Gholipour, N. Kehtarnavaz, R. Briggs, M. Devous, and K. Gopinath. Brain functional localization: A survey of image registration techniques. *IEEE Transactions on Medical Imaging*, 26(4), 2007.
- [47] S. P. DiMaio, N. Archip, N. Hata, I. F. Talos, S. K. Warfield, A. Majumdar, N. Mcdannold, K. Hynynen, P. R. Morrison, W. M. Wells, 3rd, D. F. Kacher, R. E. Ellis, A. J. Golby, P. M. Black, F. A. Jolesz, and R. Kikinis. Image-guided neurosurgery at brigham and women’s hospital. *IEEE Engineering in Medicine and Biology Magazine*, 25(5):67–73, 2006.
- [48] Y. Starreveld. *Fast Non-Linear Registration Applied to Stereotactic Functional Neurosurgery*. PhD thesis, University of Western Ontario, London, Ontario, 2002.
- [49] C. Jongen. *Interpatient Registration and Analysis in Clinical Neuroimaging*. PhD thesis, Utrecht University, The Netherlands, March 2006.
- [50] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1-3):7–42, 2002.
- [51] L. Holm and C. Sander. Mapping the protein universe. *Science*, 273:595–602, 1996.
- [52] D. Nicastro, C. Schwartz, J. Pierson, R. Gaudette, M. E. Porter, and J. R. McIntosh. The molecular architecture of axonemes revealed by cryoelectron tomography. *Science*, 313:944–948, 2006.

- [53] L. Bonetta. Zooming in on electron tomography. *Nature Methods*, 2(2):139–145, 2005.
- [54] A. Rodriguez, D. Ehlenberger, K. Kelliher, M. Einstein, S. C. Henderson, J. H. Morrison, P. R. Hof, and S. L. Wearne. Automated reconstruction of three-dimensional neuronal morphology from laser scanning microscopy images. *Methods*, 30:94–105, 2003.
- [55] M. E. Martone, A. Gupta, and M. H. Ellisman. e-Neuroscience: challenges and triumphs in integrating distributed data from molecules to brains. *Nature Neuroscience*, 7(5):467–472, 2004.
- [56] S. A. Hall, C. Macbeth, O. I. Barkved, and P. Wild. Cross-matching with interpreted warping of 3D streamer and 3D ocean-bottom-cable data at valhall for time-lapse assessment. *Geophysics Prospecting*, 53:283–297, 2005.
- [57] J. E. Rickett and D. E. Lumley. Cross-equalization data processing for time-lapse seismic reservoir monitoring: A case study from the gulf of mexico. *Geophysics*, 66(4):1015–1025, 2001.
- [58] V. Klemas. Remote sensing of landscape-level coastal environmental indicators. *Environmental Management*, 27(1):47–57, 2001.
- [59] J. Streicher, M. A. Donat, B. Strauss, R. Spörle, K. Schughart, and G. B. Mller. Computer-based three-dimensional visualization of developmental gene expression. *Nature Genetics*, 25:147–152, 2000.
- [60] P. Hastreiter and T. Ertl. Integrated registration and visualization of medical image data. In *Computer Graphics International (CGI)*, pages 78–85, 1998.
- [61] C. Rezk-Salama, P. Hastreiter, G. Greiner, and T. Ertl. Non-linear registration of pre- and intraoperative volume data based on piecewise linear transformations. In *Vision, Modelling, and Visualization*, pages 365–372, 1999.
- [62] G.E. Christensen. MIMD vs. SIMD parallel processing: A case study in 3D medical image registration. *Parallel Computing*, 24:1369–1383, 1998.
- [63] S. Ourselin, R. Stafanescu, and X. Pennec. Robust registration of multi-modal images: Towards real-time clinical applications. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 2489 of *Lecture Notes in Computer Science*, pages 140–147. Springer, Heidelberg, 2002.
- [64] M. P. Wachowiak and T. M. Peters. High-performance medical image registration using new optimization techniques. *IEEE Transactions on Information Technology in Biomedicine*, 10(2):344–353, 2006.
- [65] W. Plishker, O. Dandekar, S. Bhattacharyya, and R. Shekhar. Towards a heterogeneous medical image registration acceleration platform. In *Biomedical Circuits and Systems Conference*, pages 231–234, 2007.

- [66] S. Hastings, T. Kurc, S. Langella, U. Catalyurek, T. Pan, and J. Saltz. Image processing for the Grid: A toolkit for building grid-enabled image processing applications. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 1–8. IEEE, 2003.
- [67] L. Ibáñez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide 2.4*. Kitware, Inc., 2005.
- [68] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics (2nd edition)*. Prentice Hall, 1997.
- [69] P. Hastreiter, C. Rezk-Salama, C. Nimsy, C. Lürig, G. Greiner, and T. Ertl. Registration techniques for the analysis of the brain shift in neurosurgery. *Computers & Graphics*, 24(3):385–389, 2000.
- [70] C. Rezk-Salama, M. Scheuering, G. Soza, and G. Greiner. Fast volumetric deformation on general purpose hardware. In *ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 17–24. ACM Press, 2001.
- [71] G. Soza, M. Bauer, P. Hastreiter, C. Nimsy, and G. Greiner. Non-rigid registration with use of hardware-based 3D Bézier functions. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 2489 of *Lecture Notes in Computer Science*, pages 549–556. Springer, Heidelberg, 2002.
- [72] D. Levin, D. Dey, and P. J. Slomka. Acceleration of 3D, nonlinear warping using standard video graphics hardware: implementation and initial validation. *Computerized Medical Imaging and Graphics*, 28(8):471–483, 2004.
- [73] R. Strzodka, M. Droske, and M. Rumpf. Fast image registration in DX9 graphics hardware. *Journal of Medical Informatics and Technologies*, 6:43–49, 2003.
- [74] R. Strzodka, M. Droske, and M. Rumpf. Image registration by a regularized gradient flow - a streaming implementation in DX9 graphics hardware. *Computing*, 73(4):373–389, 2004.
- [75] U. Clarenz, M. Droske, and M. Rumpf. Towards fast non-rigid registration. *Inverse Problems, Image Analysis and Medical Imaging, AMS Special Session Interaction of Inverse Problems and Image Analysis*, 313:67–84, 2002.
- [76] A. Köhn, J. Drexler, F. Ritter, M. König, and H. O. Peitgen. GPU accelerated image registration in two and three dimensions. In *Bildverarbeitung für die Medizin 2006*, number 3 in *Informatik aktuell*, pages 261–265. Springer, Heidelberg, 2006.
- [77] R. Chisu. Techniques for accelerating intensity-based rigid image registration. Master’s thesis, Technische Universität München, München, 2005.

- [78] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara. A GPGPU approach for accelerating 2-D/3-D rigid registration of medical images. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 939–950, Sorrento, Italy, 2006.
- [79] A. Khamene, R. Chisu, W. Wein, N. Navab, and F. Sauer. A novel projection based approach for medical image registration. In *Biomedical Image Registration*, Lecture Notes in Computer Science, pages 247–256, 2006.
- [80] S. Chan. Three-dimensional medical image registration on modern graphics processors. Master’s thesis, University of Calgary, Calgary, Alberta, 2007.
- [81] D. H. Adler, S. Chan, E. S. Penner, and J. R. Mitchell. Accelerated 3D medical image registration using graphics hardware. International Society for Magnetic Resonance in Medicine: Poster presentation, May 2008.
- [82] R. Shams, P. Sadeghi, R. A. Kennedy, and R. I. Hartley. A survey of medical image registration on multicore and the GPU. *IEEE Signal Processing Magazine*, 50, March 2010.
- [83] P. Muyan-Özçelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the GPU: A CUDA implementation of demons. In *Proceedings of the 2008 International Conference on Computational Science and its Applications (ICCSA)*, page 11 pp. IEEE Computer Society Press, June 2008.
- [84] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens. Multimodality image registration by maximization of mutual information. *IEEE Transactions on Medical Imaging*, 16(2):187–198, 1997.
- [85] R. Shams and N. Barnes. Speeding up mutual information computation using NVIDIA CUDA hardware. In *Proc. of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA)*, pages 555–560, Washington, DC, 2007. IEEE Computer Society.
- [86] S. Green. The OpenGL framebuffer object extension. GameDevelopers Conference Presentation, 2005.
- [87] O. Fluck, A. Shmuel, D. Cremers, and M. Rousson. GPU histogram computation. In *International Conference on Computer Graphics and Interactive Techniques*, 2006.
- [88] T. Scheuermann and J. Hensley. Efficient histogram generation using scattering on GPUs. In *Proc. of the 2007 symposium on Interactive 3D graphics and games*, pages 33–37. ACM, 2007.
- [89] V. Podlozhnyuk. *Histogram calculation in CUDA*. NVIDIA Corporation, November 2007. Whitepaper.

- [90] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proc. of the International Conference on Signal Processing and Communication Systems (ICSPCS)*, page 5 pp, 2007.
- [91] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, and S. Daijavad. Accelerating mutual-information-based linear registration on the cell broadband engine processor. In *IEEE International Conference on Multimedia and Expo*, pages 272–275, 2007.
- [92] F. Jung and S. Wesarg. 3D registration based on normalized mutual information: Performance of CPU vs. GPU implementation. In *Bildverarbeitung für die Medizin 2010*, pages 325–329, 2010.
- [93] J. West, J. M. Fitzpatrick, M. Y. Wang, and B. M. Dawant et al. Comparison and evaluation of retrospective intermodality brain image registration techniques. *Journal of Computer Assisted Tomography*, 21(4):554–566, 1997.
- [94] T. S. Yoo, editor. *Insight into Images*. A K Peters, Wellesey, 2004.
- [95] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *ACM Symposium on Volume Visualization*, pages 91–98, 1994.
- [96] T. M. Lehmann, C. Gönnner, and K. Spitzer. Survey: Interpolation methods in medical image processing. *IEEE Transactions on Medical Imaging*, 18(11):1049–1075, 1999.
- [97] J. V. Hajnal, N. Saeed, E. J. Soar, A. Oatridge, I. R. Young, and G. M. Bydder. A registration and interpolation procedure for subvoxel matching of serially acquired MR images. *Journal of Computer Assisted Tomography*, 19(2), 1995.
- [98] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In M. Pharr, editor, *GPU Gems 2*, pages 313–329. Addison-Wesley, Upper Saddle River, 2005.
- [99] P. Rogelj, S. Kovačič, and J. C. Gee. Point similarity measures for non-rigid registration of multi-modal data. *Computer Vision and Image Understanding*, 92:112–140, 2003.
- [100] E. Haber and J. Modersitzki. Intensity gradient based registration and fusion of multi-modal images. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 4191 of *Lecture Notes in Computer Science*, pages 726–733. Springer, Heidelberg, 2006.
- [101] J. Owens. Data-parallel algorithms and data structures. SIGGRAPH 2007 Presentation, August 2007.
- [102] R. Shekhar and V. Zagrodsky. Mutual information-based rigid and nonrigid registration of ultrasound volumes. *IEEE Transactions on Medical Imaging*, 21(1):9–22, 2002.

- [103] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank. PET-CT image registration in the chest using free-form deformations. *IEEE Transactions on Medical Imaging*, 22(1), 2003.
- [104] NVIDIA Corporation. *Using Vertex Buffer Objects*, May 2004. Whitepaper.
- [105] M. Jenkinson and S. M. Smith. A global optimisation method for robust affine registration of brain images. *Medical Image Analysis*, 5(2):143–156, 2001.
- [106] A. Rege. Occlusion (HP and NV extensions). GameDevelopers Conference Presentation, 2002.
- [107] D. Loeckx, F. Maes, D. Vandermeulen, and P. Suetens. Comparison between parzen window interpolation and generalised partial volume estimation for non-rigid image registration using mutual information. In *Biomedical Image Registration*, volume 4057 of *Lecture Notes in Computer Science*, pages 206–213. Springer-Verlag, 2006.
- [108] N. Dowson and R. Bowden. A unifying framework for mutual information methods for use in non-linear optimisation. In *European Conference on Computer Vision*, volume 3951 of *Lecture Notes in Computer Science*, pages 365–378. Springer-Verlag, 2006.
- [109] F. Maes, D. Vandermeulen, and P. Suetens. Comparative evaluation of multiresolution optimization strategies for multimodality image registration by maximization of mutual information. *Medical Image Analysis*, 3(4):373–386, 1999.
- [110] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17, 1981.
- [111] X. Pennec, P. Cachier, and N. Ayache. Understanding the ”demon’s algorithm”: 3D non-rigid registration by gradient descent. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 1679 of *Lecture Notes in Computer Science*, pages 597–605. Springer, Heidelberg, 1999.
- [112] H. Wang, L. Dong, J. O’Daniel, R. Mohan, A. S. Garden, K. K. Ang, D. A. Kuban, M. Bonnen, J. Y. Chang, and R. Cheung. Validation of an accelerated ’demons’ algorithm for deformable image registration in radiation therapy. *Physics in Medicine and Biology*, 50(12), 2005.
- [113] A. Klein, J. Andersson, B. A. Ardekani, J. Ashburner, B. Avants, M.-C. Chiang, G. E. Christensen, D. L. Collins, J. Gee, P. Hellier, J. H. Song, M. Jenkinson, C. Lepage, D. Rueckert, P. Thompson, T. Vercauteren, R. P. Woods, J. J. Mann, and R. V. Parsey. Evaluation of 14 nonlinear deformation algorithms applied to human brain MRI registration. *NeuroImage*, 46(3):786–802, 2009.
- [114] P. Rogelj, S. Kovačič, and J. C. Gee. Validation of a non-rigid registration algorithm for multi-modal data. In *SPIE Medical Imaging: Image Processing*, pages 299–307, Feb 2002.

- [115] B. A. Ardekani, S. Guckemus, A. Bachman, M. J. Hoptman, M. Wojtaszek, and J. Nierenberg. Quantitative comparison of algorithms for inter-subject registration of 3D volumetric brain MRI scans. *Journal of Neuroscience Methods*, 142(1):67–76, 2005.
- [116] P. Hellier, C. Barillot, I. Corouge, B. Gibaud, G. Le Goualher, L. Collins, A. Evans, G. Malandain, N. Ayache, G. E. Christensen, and H. J. Johnson. Retrospective evaluation of intersubject brain registration. *IEEE Transactions on Medical Imaging*, 22(9):1120–1130, Sept 2003.
- [117] C. A. Cocosco, V. Kollokian, R. K.-S. Kwan, and A. C. Evans. BrainWeb: Online interface to a 3D MRI simulated brain database. *NeuroImage*, 5(4), 1997.
- [118] D. L. Collins, C. J. Holmes, T. M. Peters, and A. C. Evans. Automatic 3-D model-based neuroanatomical segmentation. *Human Brain Mapping*, 3(3):190–208, 1995.
- [119] D. L. Collins, A. P. Zijdenbos, V. Kollokian, J. G. Sled, N. J. Kabani, C. J. Holmes, and A. C. Evans. Design and construction of a realistic digital brain phantom. *IEEE Transactions on Medical Imaging*, 17(3):463–468, 1998.
- [120] M. Jenkinson. Measuring transformation error by RMS deviation. Technical Report TR99MJ1, Oxford Centre for Functional Magnetic Resonance Imaging of the Brain, 1999.
- [121] L. Zagorchev and A. Goshtasby. A comparative study of transformation functions for nonrigid image registration. *IEEE Transactions on Image Processing*, 15(3):529–538, 2006.
- [122] F. Barkhof, J.-H.T.M. van Waesberghe, M. Filippi, T. Yousry, D.H. Miller, D. Hahn, A.J. Thompson, L. Kappos, P. Brex, C. Pozzilli, and C.H. Polman. T1 hypointense lesions in secondary progressive multiple sclerosis: effect of interferon beta-1b treatment. *Brain*, 124:1396–1402, 2001.
- [123] S. S. Samant, J. Xia, P. Muyan-Özçelik, and J. D. Owens. High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy. *Medical Physics*, 35(8):3546–3553, 2008.
- [124] S. Drabycz, G. Roldán, P. de Robles, D. Adler, J. B. McIntyre, A. M. Magliocco, J. G. Cairncross, and J. R. Mitchell. Analysis of MGMT promoter methylation status in high grade glioma patients with long term and conventional survival times: a retrospective study. *NeuroImage*, 49(2), 2010.
- [125] R. Stupp, W. P. Mason, M. J. van den Bent, M. Weller, B. Fisher, M. J. Taphoorn, K. Belanger, A. A. Brandes, C. Marosi, U. Bogdahn, J. Curschmann, R. C. Janzer, S. K. Ludwin, T. Gorlia, A. Allgeier, D. Lacombe, J. G. Cairncross, E. Eisenhauer, and R. O. Mirimanoff. Radiotherapy plus concomitant and

- adjuvant temozolomide for glioblastoma. *New England Journal of Medicine*, 352(10), 2005.
- [126] M. E. Hegi, A. C. Diserens, T. Gorlia, M.-F. Hamou, N. de Tribolet, M. Weller, J. M. Kros, J. A. Hainfellner, W. Mason, L. Mariani, J. E. C. Bromberg, P. Hau, R. O. Mirimanoff, J. G. Cairncross, R. C. Janzer, and R. Stupp. MGMT gene silencing and benefit from temozolomide in glioblastoma. *New England Journal of Medicine*, 352(10), 2005.
 - [127] P. Hau, R. Stupp, and M. E. Hegi. MGMT methylation status: the advent of stratified therapy in glioblastoma? *Disease Markers*, 23(1), 2007.
 - [128] M. C. Zlatescu, A.-R. Tehrani-Yazdi, H. Sasaki, J. F. Megyesi, R. A. Betensky, D. N. Louis, and J. G. Cairncross. Tumor location and growth pattern correlate with genetic signature in oligodendroglial neoplasms. *Cancer Research*, 61, 2001.
 - [129] W. Mueller, C. Hartmann, A. Hoffmann, W. Lanksch, J. Kiwit, J. Tonn, J. Veelken, J. Schramm, M. Weller, O. D. Wiestler, D. N. Louis, and A. von Deimling. Genetic signature of oligoastrocytomas correlates with tumor location and denotes distinct molecular subsets. *American Journal of Pathology*, 161(1), 2002.
 - [130] D. A. Lim, S. Cha, M. C. Mayo, M.-H. Chen, E. Keles, S. VandenBerg, and M. S. Berger. Relationship of glioblastoma multiforme to neural stem cell regions predicts invasive and multifocal tumor phenotype. *Neuro-Oncology*, 9(4), 2007.
 - [131] P. B. Dierks. Stem cells and brain tumors. *Nature*, 444(7120), 2006.
 - [132] R. J. Gilbertson and D. H. Gutmann. Tumorigenesis in the brain: Location, location, location. *Cancer Research*, 67(12), 2007.
 - [133] E. M. Grasbon-Frodl, F. W. Kreth, M. Ruiter, O. Schnell, K. Bise, J. Felsberg, G. Reifenberger, J. C. Tonn, and H. A. Kretzschmar. Intratumoral homogeneity of MGMT promoter hypermethylation as demonstrated in serial stereotactic specimens from anaplastic astrocytomas and glioblastomas. *International Journal of Cancer*, 121(11), 2007.
 - [134] R. J. Schrot, J. H. Ma, C. M. Greco, A. D. Arias, and J. M. Angelastro. Organotypic distribution of stem cell markers in formalin-fixed brain harboring glioblastoma multiforme. *Journal of Neurooncology*, 85(2), 2007.
 - [135] M. Eoli, F. Menghi, M. G. Bruzzone, T. De Simone, L. Valletta, B. Pollo, L. Bisola, A. Silvani, D. Bianchessi, L. D’Incerti, G. Filippini, G. Broggi, A. Boiardi, and G. Finocchiaro. Methylation of O6-methylguanine DNA methyltransferase and loss of heterozygosity on 19q and/or 17p are overlapping features of secondary glioblastomas with prolonged survival. *Clinical Cancer Research*, 13(9):2606–2613, 2007.

- [136] F. Requena and N. Martín-Ciudad. A major improvement to the network algorithm for fishers exact test in 2xc contingency tables. *Computational Statistics and Data Analysis*, 51(2), 2006.
- [137] W. R. Crum, L. D. Griffin, D. L. G. Hill, and D. J. Hawkes. Zen and the art of medical image registration: correspondence, homology, and quality. *NeuroImage*, 20(3):1425–1437, 2003.
- [138] F. Fasano and A. Franceschini. A multidimensional version of the Kolmogorov-Smirnov test. *Monthly Notices of the Royal Astronomical Society*, 255, 1987.
- [139] E. Gosset. A three-dimensional extended Kolmogorov-Smirnov test as a useful tool in astronomy. *Astronomy and Astrophysics*, 188, 1987.
- [140] J. P. W. Pluim, J. B. A. Maintz, and M. A. Viergever. Image registration by maximization of combined mutual information and gradient information. *IEEE Transactions on Medical Imaging*, 19(8):809–814, 2000.
- [141] R. P. Woods, S. R. Cherry, and J. C. Mazziotta. Rapid automated algorithm for aligning and reslicing PET images. *Journal of Computer Assisted Tomography*, 16(4):620–633, 1992.
- [142] H.-M. Chen and P. K. Varshney. Mutual information-based CT-MR brain image registration using generalized partial volume joint histogram estimation. *IEEE Transactions on Medical Imaging*, 22(9):1111–1119, 2003.
- [143] P. Thévenaz and M. Unser. Optimization of mutual information for multiresolution image registration. *IEEE Transactions on Image Processing*, 9(12), 2000.
- [144] M. Unser and P. Thévenaz. Stochastic sampling for computing the mutual information of two images. In *Proceedings of the Fifth International Workshop on Sampling Theory and Applications*, pages 102–109, 2003.
- [145] J. Krüger and R. Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization 2003*, 2003.
- [146] B. Rodríguez-Vila, J. Pettersson, M. Borga, F. Garca-Vicente, E.J. Gómez, and H. Knutsson. 3D deformable registration for monitoring radiotherapy treatment in prostate cancer. In *Scandinavian Conference on Image Analysis (SCIA)*, volume 4522 of *Lecture Notes in Computer Science*, pages 750–759. Springer, Heidelberg, June 2007.
- [147] M. F. Beg, M. I. Miller, and A. Trouvé. Computing large deformation metric mappings via geodesic flows of diffeomorphisms. *International Journal of Computer Vision*, 61(2):139–157, 2005.
- [148] B. B. Avants, C. L. Epstein, M. Grossman, and J. C. Gee. Symmetric diffeomorphic image registration with cross-correlation: Evaluating automated labeling of elderly and neurodegenerative brain. *Medical Image Analysis*, 12(1):26–41, 2008.

- [149] R. Fernando, editor. *GPU Gems*. Addison-Wesley, Boston, 2004.
- [150] S. Zerbst. *Direct3D and 3D Engine Programming*. Lulu.com, 2006.