High-Performance Image Registration Algorithms for Multi-Core Processors

A Thesis

Submitted to the Faculty

of

Drexel University

by

James Anthony Shackleford

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

December 2011

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

High-Performance Image Registration Algorithms for Multi-Core Processors

James Anthony Shackleford

Deformable registration consists of aligning two or more 3D images into a common coordinate frame. Fusing multiple images in this fashion quantifies changes in organ shape, size, and position as described by the image set, thus providing physicians with a more complete understanding of patient anatomy and function. In the field of image-guided surgery, for example, neurosurgeons can track localized deformations within the brain during surgical procedures, thereby reducing the amount of unresected tumor.

Though deformable registration has the potential to improve the geometric precision for a variety of medical procedures, most modern algorithms are time consuming and, therefore, go unused for routine clinical procedures. This thesis develops highly data-parallel registration algorithms suitable for use on modern multi-core architectures, including graphics processing units (GPUs). Specific contributions include the following:

Parallel versions of both unimodal and multi-modal B-spline registration algorithms where the deformation is described in terms of uniform cubic B-spline coefficients. The unimodal case involves aligning images obtained using the same imaging technique whereas multi-modal registration aligns images obtained via differing imaging techniques by employing the concept of statistical mutual information.

Multi-core versions of an analytical regularization method that imposes smoothness constraints on the deformation derived by both unimodal and multi-modal registration.

The proposed method operates entirely on the B-spline coefficients which parameterize the deformation and, therefore, exhibits superior performance, in terms of execution-time overhead, over numerical methods that use central differencing.

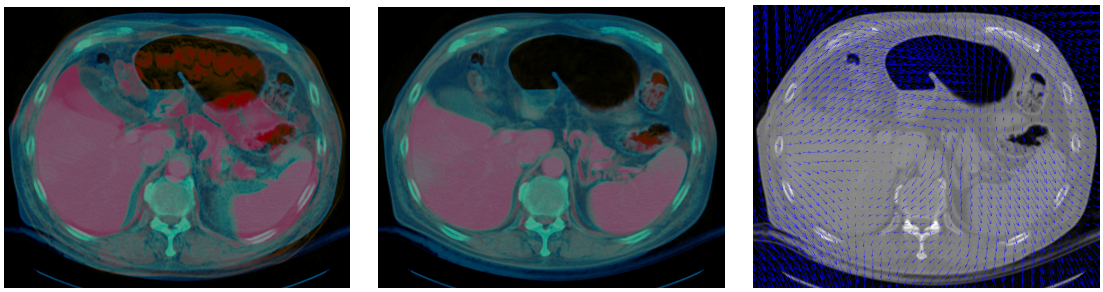
The above contributions have been implemented as part of the high-performance medical image registration software package Plastimatch, which can be downloaded under an open source license from www.plastimatch.org. Plastimatch significantly reduces the execution time incurred by B-spline based registration algorithms: compared to highly optimized sequential implementations on the CPU, we achieve a speedup of approximately 21 times for GPU-based multi-modal deformable registration while maintaining near-identical registration quality and a speedup of approximately 600 times for multi-core CPU-based regularization. It is hoped that these improvements in processing speed will allow deformable registration to be routinely used in time-sensitive procedures such as image-guided surgery and image-guided radiotherapy which require low latency from imaging to analysis.

CHAPTER 1: INTRODUCTION

Modern imaging techniques such as computed tomography (CT), positron emission tomography (PET), and magnetic resonance imaging (MRI) provide physicians with 3D image volumes of patient anatomy which convey information instrumental in treating a wide range of afflictions. It is often useful to register one image volume to another to understand how patient anatomy has changed over time or to relate image volumes obtained via different imaging techniques. For example, MRI provides a means of distinguishing soft tissues that are otherwise indiscernible in a transmission-based CT scan. The recent availability of portable CT scanners inside the operating room has led to the development of new methods of localizing cancerous soft tissue by registering intra-operative CT scans to a pre-operative MRI as shown in Fig. 1.1, thus allowing for precise tumor localization during the resection procedure.

1.1 Applications of Deformable Image Registration

The volumetric registration process consists of aligning two or more 3D images into a common coordinate frame via a deformation vector field. Fusing multiple images in this fashion provides physicians with a more complete understanding of patient anatomy and function. A registration is called *rigid* if the motion or change is limited to global rotations and translations, and is called *deformable* when the registration includes complex local variations.



(a) (b) (c)

Figure 1.1: Computing organ motion via deformable registration. (a) A pre-operative MRI image (in red) superimposed on an intra-operative CT image (in blue) before deformable registration; (b) the pre-operative MRI superimposed on the intra-operative CT after deformable registration; (c) the deformation vector field (in blue) derived by the registration process superimposed on the intra-operative CT scan wherein the vector field quantitatively describes the organ motion between the CT and MRI scans.

Rigid matching is adequate for serial imaging of the skull, brain, or other rigidly immobilized sites. Deformable registration is appropriate for almost all other scenarios and is useful for many applications within medical research, medical diagnosis, and interventional treatments.

The use of deformable registration has already begun to change medical research practices, especially in the fields of neuroanatomy and brain science. Deformable registration plays an important role in studying a variety of diseases including Alzheimer's disease [1, 2, 3], schizophrenia [4, 5], and generalized brain development [6]. Many of these studies make use of a powerful concept known as brain functional localization [7], which provides a method of mapping functional information to corresponding anatomic locations within the brain. This allows researchers to correlate patient MRI scans with a brain atlas to improve our understanding of how the brain is damaged by disease.

Deformable registration is also beginning to impact the field of image-guided surgery.

For example, neurosurgeons can now track localized deformations within the brain during surgical procedures, thus reducing the amount of unresected tumor [8, 9]. Similar benefits may be observed in surgical operations involving the prostate [10, 11], heart [12], and the liver [13, 14] where local complex organ deformation are a common impediment to procedural success. The application of deformable registration to such interventional surgical procedures does, however, carry with it unique challenges. Often multi-modal imaging is required, such as matching an intra-operative ultrasound with pre-operative MRI or a pre-operative MRI with an intra-operative CT scan. Since such registrations must be performed during active surgical procedures, the time to acquire an accurate solution must be reasonably fast. Additionally, surgical incisions and resections performed prior to intra-operative imaging analysis result in additional deformations that may be difficult to recover algorithmically.

In image-guided radiotherapy, deformable registration is used to improve the geometric and dosimetric accuracy of radiation treatments. Motion due to respiration has a "dose-blurring" effect, which is important for treatments in the lung [15, 16, 17] and liver [18, 19, 20]. Day-to-day changes in organ position and shape also affect radiological treatments to the prostate [21] and head and neck regions [22]. In addition to improving treatment delivery, deformable registration is also used in treatment verification and treatment response assessment [23]. Furthermore, deformable registration can be used to construct time-continuous 4D fields that provide a basis for motion estimation [24, 25] and time-evolution visualization [26], which aids in improving the dosimetric accuracy to tumors within the lung.

1.2 Algorithmic Approaches to Deformable Registration

The choice of an image registration method for a particular application is still largely unsettled. There are a variety of deformable image registration algorithms, distinguished by choice of similarity measure, transformation model, and optimization process [27, 28, 29, 30]. The most popular and successful methods seem to be based on surface matching [31], optical flow equations [32], fluid registration [33], thin-plate splines [34, 35], finite-element models (FEMs) [36], and B-splines [37]. The involvement of academic researchers in the development of deformable registration methods has resulted in several high-quality open source software packages. Notable examples include the Statistical Parametric Mapping software (SPM) [38], the Insight Segmentation and Registration Toolkit (ITK) [39], AIR [40], Freesurfer [41], and vtkCISG [42].

Though deformable registration has the potential to greatly improve the geometric precision for a variety of medical procedures, modern algorithms are computationally intensive. Consequently, deformable registration algorithms are not commonly accepted into general clinical practice due to their excessive processing time requirements. The fastest family of deformable registration algorithms are based on optical flow methods typically requiring several minutes to compute [16], and it is not unusual to hear of B-spline registration algorithms requiring hours to compute [43, 44] depending on the specific algorithm implementation, image resolution, and clinical application requirements. However, despite its computational complexity, B-spline based registration remains popular due to its flexibility and robustness in providing the ability to perform both uni-modal and multi-modal registrations. In other words, B-spline based registration is capable of registering two images obtained via

the same imaging method (unimodal registration) as well as images obtained via differing imaging methods (multi-modal registration). Consequently, surgical operations benefiting from CT to MRI registration may be routinely performed once multi-modal B-spline based registration can be performed with adequate speed.

A key element in accelerating medical imaging algorithms, including deformable registration, is the use of parallel processing. In many cases, images may be partitioned into computationally independent sub-regions and subsequently farmed out to be processed in parallel. The most prominent example of this approach is the use of a parallel solver such as PETSc [45]. The PETsc library is a suite of solvers for partial differential equations (PDEs) that employs the Message Passing Interface (MPI) standard to communicate data between multiple computers. Parallel MPI-based implementations of the FEM-based registration method using PETsc have been demonstrated and benchmarked by Warfield et al. [46, 47] and Semresant et al. [48]. The overall approach is to first parallelize the appropriate algorithmic steps (e.g., the displacement field estimation), partition the image data into small sets, and then process each set independently on a computer within the cluster.

1.3 Thesis Contributions

While cluster computing is a well-established technique for accelerating image fusion algorithms, recent advances in multi-core processor design offer new opportunities for truly large-scale and cost-effective parallel computing on a single chip. The Cell processor [49] and recent series of Nvidia graphics processing units (GPUs) are two examples of commodity stream processors designed specifically to support the single chip parallel computing paradigm. These processors have a large number of arithmetic units on chip, far more than

any general-purpose microprocessor, making them well suited for high-performance parallel-processing tasks. For example, the NVidia GTX 285 GPU has 240 processing cores with dynamic branching support. The architecture provides 32-bit floating-point arithmetic and offers 1063 GLOPs of computing power whereas, by comparison, a modern 3 GHz quad-core Pentium i7 processor may provide up to roughly 50 GFLOPs.

This thesis focuses on B-spline deformable registration algorithms and the development of multi-core accelerated methods for performing both unimodal and multi-modal registration with high precision and low latency. Specific contributions include:

- A highly parallel and thread-safe method of parameterizing the densely-defined deformation vector field in terms of a sparsely-defined set of uniform cubic B-spline basis function coefficients. This algorithmic building block forms the basis that enables B-spline based algorithms to be effectively implemented in a data-parallel fashion.

- Development of a highly data-parallel unimodal B-spline registration algorithm based on the mean squared error (MSE) similarity metric with implementations for both multi-core CPUs and many-core GPUs.

- Development of a data parallel multi-modal B-spline registration algorithm based on the statistical mutual information (MI) similarity metric with implementations for both multi-core CPUs and many-core GPUs.

- Parallel methods of constructing both marginal and joint histograms for floating-point quantities. This is a necessary algorithmic building block for data-parallel MI-based registration implementations.

- A method for quickly computing so-called "variance-optimal" or V-opt histograms. We show that employing such histograms increases the accuracy of MI-based registration while also serving to decrease the number of necessary histogram bins.

- Development of an analytic regularization method that imposes smoothness constraints on the deformation vector fields derived by both the unimodal and multimodal B-spline implementations. This method operates entirely on the B-spline coefficients that parameterize the deformation field and exhibits superior performance, in terms of execution-time overhead, over traditional central differencing methods. Both single core and multi-core accelerated implementations are developed.

The above-described contributions have been implemented for both multi-core CPUs as well as GPUs, and comprise the B-spline registration algorithms used by the high-performance medical image registration software package Plastimatch [50], which can be freely download under a BSD-style license from `www.plastimatch.org`.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 provides an overview of the unimodal B-spline registration algorithm and subsequently introduces a grid-alignment scheme for improving the algorithm's computation speed for both single and multi-core architectures. Using the grid-alignment scheme as a foundation, a high-performance multi-core algorithm is developed and described in detail. The fundamental concepts of image-similarity scoring, vector field evolution, and B-spline parameterization are covered in depth. Additionally, aspects of the CUDA programming model relevant to implementing the B-spline deformable reg-

istration algorithm on modern GPU hardware are introduced and discussed, and a highly parallel GPU implementation is developed. Finally, the single-core CPU, multi-core CPU, and many-core GPU based implementations are benchmarked for performance and registration quality using synthetic CT images as well as thoracic CT image volumes.

Chapter 3 describes how the B-spline registration algorithm may be extended to perform multi-modal image registration by utilizing the mutual information (MI) similarity metric. Modifications to the algorithm structure and the data flow presented in Chapter 2 are discussed in detail, and strategies for accelerating these new algorithmic additions are explored. Specific attention is directed towards developing memory-efficient and data-parallel methods of constructing the marginal and joint image-intensity histograms, since these data structures are key to successfully performing the MI-based registration. The impact of the MI similarity metric on the analytic formalism driving the vector field evolution is covered in depth and the partial-volume interpolation method is introduced that dictate how the intensity histograms evolve as the vector field evolves. Multi-core implementations are benchmarked for performance using synthetic image volumes. Finally, registration quality is assessed using examples of multi-modal thoracic MRI to CT deformable registration.

Chapter 4 focuses on improving the performance of the MI-based registration procedure presented in Chapter 3 by optimizing the estimation of information content within the marginal and joint intensity histograms. The statistical approach of MI scores potential registration solutions based on the amount of information mutually contained within both images—a metric derived directly from entropy measures of the marginal and joint histograms. Consequently, sub-optimal histogram binning schemes result in the misrepresentation of image entropy, introducing error into the assessment of the registration quality.

This chapter introduces the application of variance-optimal (or V-opt) histograms to MI-based registration. The technique computes non-uniform bin boundaries such that the sum of the variance within all the individual bins is minimal. The desirability of such a binning scheme is discussed in terms of tissue density separation within the histogram space. Furthermore, a method of approximating the V-opt histogram configuration is introduced that greatly reduces the associated computational complexity, thereby enabling routine use within the clinical environment. Improvements to MI-based registration using this binning technique are investigated via full-body PET to CT and thoracic MRI to CT cases.

Chapter 5 develops an analytic method for constraining the evolution of the deformation vector field that seamlessly integrates into both unimodal and multi-modal B-spline based registration algorithms. Although the registration methods presented in Chapters 2 and 3 generate vector fields describing how best to transform one image to match the other, there is no guarantee that these transformations will be physically valid. Image registration is an ill-posed problem in that it lacks a unique solution to the vector deformation field and consequently, the solution may describe a physical deformation that did not or could not have occurred. However, by imposing constraints on the character of the vector field, it is possible to guide its evolution towards physically meaningful solutions; in other words, the ill-posed problem is regularized. This chapter provides the analytic mathematical formalism required to impose second-order smoothness upon the deformation vector field in a faster and more efficient fashion than numerically-based central differencing methods. Furthermore, we show that such analytically-derived matrix operators may be applied directly to the B-spline parameterization of the vector field to achieve the desired physically meaningful solutions. Single and multi-core CPU implementations are developed and

discussed and the performance for both implementations is investigated with respect to the numerical method in terms of execution-time overhead, and the quality of the analytic implementations is investigated via a thoracic MRI to CT case study.

Finally, Chapter 6 concludes the thesis. Overarching concepts presented within the preceding chapters are concisely reviewed with a focus placed on key ideas and contributions to the field of high-performance image processing.

CHAPTER 2: UNIMODAL B-SPLINE REGISTRATION

This chapter provides an overview of the B-spline registration algorithm and subsequently introduces a grid-alignment scheme for improving the algorithm's computation speed for both single and multi-core computing architectures. Using this grid-alignment scheme as a foundation, an optimal multi-core algorithm is then derived and subsequently described in detail. Finally, single-core CPU, multi-core CPU, and many-core GPU based implementations are benchmarked for performance and quality comparison. The work presented in this chapter has been previously published as a chapter in the book GPU Computing Gems [51] and as a featured article in Physics in Medicine and Biology [52].

2.1 Overview of B-spline Registration

B-spline registration is a method of deformable registration that uses B-spline curves to define a continuous displacement field that maps the voxels in one image to those in another image. Fig. 2.1 shows an example of deformable registration of 3D CT images using B-splines, where registration is performed between an inhaled-lung image taken at time t_1 and an exhale image taken at t_2. Prior to registration, the image difference shown is quite large, highlighting the motion of the diaphragm and pulmonary vessels during breathing. Registration is performed to generate the vector or displacement field. After registration, the image difference is much smaller, demonstrating that the registration has successfully

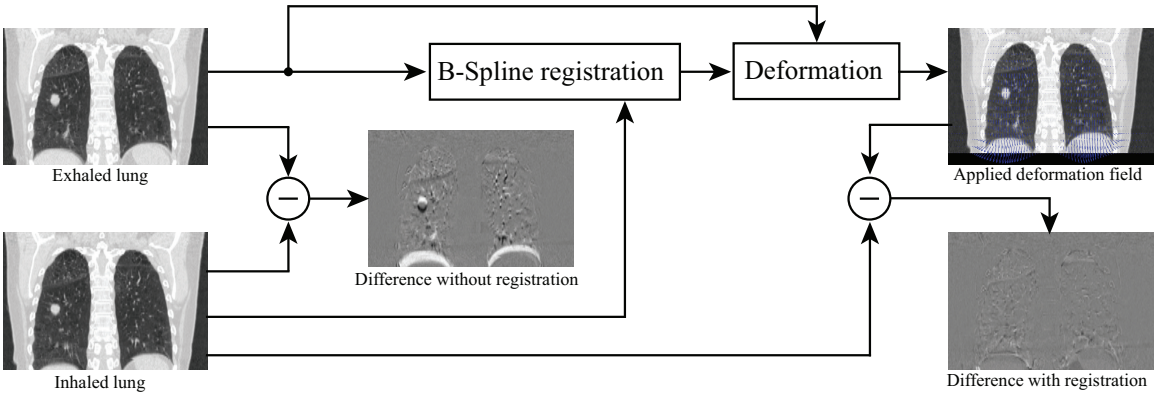
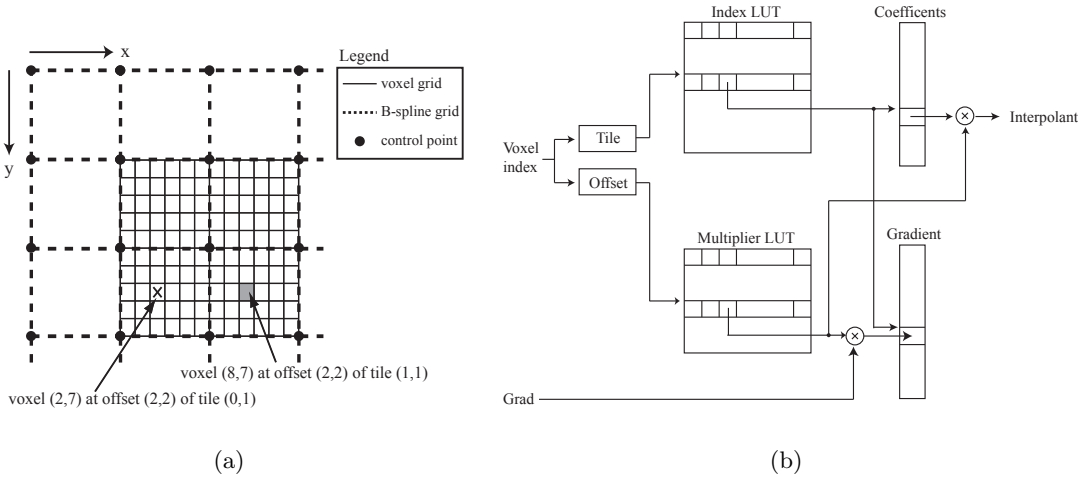


Figure 2.1: Difference with and without registration. Deformable registration of two 3D CT volumes. Images of an inhaled lung taken at time t_1 and an exhaled lung taken at time t_2 serve as the static and moving images, respectively. The registration algorithm iteratively deforms the moving image in an attempt to minimize the intensity difference between the static and moving images. The final result is a vector field describing how voxels in the moving image should be shifted in order to make it match the static image. The difference between the static and moving images with and without registration is also shown.

matched tissues of similar density.

Since we can define the vector field in a parametric fashion (i.e., in terms of coefficients provided by a finite number of control points), a cost function that quantifies the similarity between the static and moving images can be specified and registration can be posed as an optimization problem. This requires that we evaluate: (1) C, the cost function corresponding to a given set of spline coefficients, and (2) $\partial C/\partial P$, the change in the cost function with respect to the coefficient values P at each individual control point. We will refer to $\partial C/\partial P$ as the *cost function gradient* throughout the paper. The registration process then becomes one of iteratively defining coefficients P, performing B-spline interpolation, evaluating the cost function C, calculating the cost function gradient $\partial C/\partial P$ for each control point, and performing gradient-descent optimization to generate the next set of coefficients.

B-spline interpolation and gradient computation are the two most time-consuming stages



(a) (b)

Figure 2.2: Design structure of grid aligned scheme. (a) A portion of a 2D image showing a B-spline control-point grid superimposed upon an aligned voxel grid. Since both the marked voxel and the grayed voxel are located at the same relative offset within their respective tiles, both voxels will use the same $\beta_l(u)\beta_m(v)$ value. (b) For aligned grids, lookup tables can accelerate deformable registration computations by eliminating redundant computations.

within the overall registration process, and so we focus on accelerating these stages using a grid-alignment scheme and accompanying data structures.

2.1.1 B-spline Interpolation

By aligning the voxel grid with a uniformly-spaced control grid, as shown in Fig. 2.2, the image volume becomes partitioned into many equally sized tiles. In the shown 2D example, the control grid partitions the voxel grid into 6×5 tiles. The vector field at any given voxel within a tile is influenced by the 16 control points in the tile's immediate vicinity and the value of the B-spline basis function product evaluated at the voxel, which is dependant only on the voxel's local coordinates (i.e. offset) within the tile. Notice that the two marked voxels in Fig. 2.2, while residing at different locations within the image, both possess the same offsets within their respective tiles. This results in the B-spline basis function product yielding identical values when evaluated at these two voxels. This property allows us to

pre-compute all relevant B-spline basis function products once instead of recomputing the evaluation for each individual tile.

In the 3D case, the vector field at any given voxel is determined by the 64 control points in the immediate vicinity of the voxel's housing tile. This configuration forms the basis of an analytic expression for the continuous vector field $\vec{\nu}$. B-spline interpolation is performed for each voxel within a tile with respect to the 64 control point coefficients that form the local support for the operation. For example, the B-spline interpolation yielding the x-component of the displacement vector for a voxel located at coordinate (x, y, z) is

$$\nu_x(x, y, z) = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} \beta_l(u)\beta_m(v)\beta_n(w)P_{x,l,m,n}, \tag{2.1}$$

where P_x is the spline coefficient defining the x component of the displacement vector for one of the 64 control points that influence the voxel. We obtain the spline basis functions β as follows. Let N_x, N_y, and N_z denote the distance between control points, in terms of voxels, in the x, y, and z directions, respectively. The volume is therefore segmented by the B-spline control point grid into many equal-sized tiles of dimensions $N_x \times N_y \times N_z$. The three dimensional indices x_t, y_t, and z_t of the tile within which the voxel at (x, y, z) falls is given by

$$x_t = \left\lfloor \frac{x}{N_x} \right\rfloor - 1, y_t = \left\lfloor \frac{y}{N_y} \right\rfloor - 1, z_t = \left\lfloor \frac{z}{N_z} \right\rfloor - 1. \tag{2.2}$$

The local coordinates of the voxel within its tile, normalized between $[0, 1]$, are

$$u = \frac{x}{N_x} - \left\lfloor \frac{x}{N_x} \right\rfloor, v = \frac{y}{N_y} - \left\lfloor \frac{y}{N_y} \right\rfloor, w = \frac{z}{N_z} - \left\lfloor \frac{z}{N_z} \right\rfloor. \tag{2.3}$$

Finally, the uniform cubic B-spline basis function β_l along the x direction is given by

$$\beta_l(u) = \begin{cases} \frac{(1-u)^3}{6} & : l = 0 \\ \frac{3u^3-6u^2+4}{6} & : l = 1 \\ \frac{-3u^3+3u^2+3u+1}{6} & : l = 2 \\ \frac{u^3}{6} & : l = 3, \end{cases} \tag{2.4}$$

and similarly for β_m and β_n along the y and z directions, respectively.

A straightforward implementation of (2.1) to compute the displacement vector $\vec{\nu}$ for a single voxel requires 192 computations of the cubic polynomial B-spline basis function β as well as 192 multiplications and 63 additions. However, many of these calculations are redundant and can be eliminated by implementing a data structure that exploits symmetrical features that emerge as a result of the grid alignment, making the implementation of (2.1) much faster.

- All voxels residing within a single tile use the same set of 64 control points to compute their respective displacement vectors. So, for each tile in the volume, the corresponding set of control point indices can be pre-computed and stored in a lookup table (LUT), called the *Index LUT*. These indices serve as pointers to a coefficient table.

- Eq. (2.3) indicates that for a tile of dimension $N_w = N_x \times N_y \times N_z$, the number of $\beta(u)\beta(v)\beta(w)$ combinations is limited to N_w values. Furthermore, as Fig. 2.2 shows, two voxels belonging to different tiles but possessing the same normalized coordinates (u, v, w) within their respective tiles will be subject to identical $\beta(u)\beta(v)\beta(w)$ products. Therefore, we pre-compute the $\beta_l(u)\beta_m(v)\beta_n(w)$ product for all valid normalized coordinate combinations (u, v, and w) and store the results into a LUT called the *Multiplier LUT*.

Fig. 2.2 shows the complete data structure required to support the above-described optimizations. For each voxel, its absolute coordinate (x, y, z) within the volume is used to calculate the tile number that the voxel falls within as well as the voxel's relative coordinates within that tile using (2.2) and (2.3), respectively. The tile number is used to query the *Index LUT*, which provides the coefficient values associated with the 64 control points influencing the voxel's interpolation calculation. The voxel's relative coordinates (u, v, w) within the tile determine its index within $[0, N_w]$, which is used to retrieve the appropriate pre-computed $\beta(u)\beta(v)\beta(w)$ product from the *Multiplier LUT*. Computing ν_x, the x component of the displacement vector for the voxel, therefore, requires looping through the 64 entries of each LUT, fetching the associated values, multiplying, and accumulating. Similar computations are required to obtain ν_y and ν_z. The LUTs are stored in CPU cache, thereby achieving extremely fast lookup times.

Once the displacement vector field is generated, it is used to deform each voxel in the moving image. Tri-linear interpolation is used to determine the value of voxels mapping to non-integer grid coordinates. Once deformed, the moving image is compared to the static image in terms of a similarity metric or cost function. This paper focuses on matching images using the mean squared error (MSE) cost function. The MSE is computed once per iteration by accumulating the square of the intensity difference between the static image S and the deformed moving image M as

$$C = \frac{1}{N} \sum_{z} \sum_{y} \sum_{x} (S(x, y, z) - M(x + \nu_x, y + \nu_y, z + \nu_z))^2, \tag{2.5}$$

where C is the cost function and N is the total number of voxels in the volume.

2.1.2 Gradient Computation and Optimization

Gradient descent optimization requires computing the partial derivatives of the cost function with respect to each control-point coefficient value. In addition to accelerating B-spline interpolation, the grid-alignment scheme also accelerates cost function gradient computation using the same data structure shown in Fig. 2.2. Recall that the cost function gradient $\partial C/\partial P$ quantifies the change in the cost function with respect to the coefficient values P at each individual control point. So, we decompose the cost function gradient for a given control point at control grid coordinates (κ, λ, μ) as

$$\frac{\partial C}{\partial P}_{\kappa,\lambda,\mu} = \frac{1}{N} \sum_{(x,y,z)}^{64\ tiles} \frac{\partial C}{\partial \vec{\nu}(x,y,z)} \frac{\partial \vec{\nu}(x,y,z)}{\partial P}. \tag{2.6}$$

where the summation is performed for all voxels (x, y, z) contained within the 64 tiles found in the control point's local support region [53]. This decomposition allows us to evaluate the cost function gradient's dependencies on the cost function and spline coefficients independently. The first term, $\partial C/\partial \vec{\nu}$, depends only on the cost function and is independent of the type of spline parameterization employed. The second term describes how the deformation field changes with respect to the control-point coefficients and can be found by simply taking the derivative of (2.1) with respect to P. This term is dependent only on the B-spline parameterization and is computed as

$$\frac{\partial \vec{\nu}(x,y,z)}{\partial P} = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} \beta_l(u)\beta_m(v)\beta_n(w). \tag{2.7}$$

Here, (2.7) depends only on a voxel's location and the B-spline parameterization. So, it will remain constant over all optimization iterations. This allows us to pre-compute and store

(2.7) for each voxel coordinate prior to the optimization process. Note also, that the values generated by (2.7) are already available via the Multiplier LUT.

If MSE is used as the cost function, the first term in (2.6) can be re-written in terms of the moving image's spatial gradient $\nabla M(x, y, z)$ as

$$\frac{\partial C}{\partial \vec{\nu}(x, y, z)} = 2 \times [S(x, y, z) - M(x + \nu_x, y + \nu_y, z + \nu_z)] \nabla M(x, y, z). \tag{2.8}$$

The above expression depends on the intensity values of the static and moving images, S and M, respectively, as well as the current value of the vector field $\vec{\nu}$. During each iteration, the vector field will change, modifying the correspondences between the static and moving images. Therefore, unlike $\partial\vec{\nu}/\partial P$, $\partial C/\partial\vec{\nu}$ must be recomputed during each iteration of the optimization problem. Once both terms are computed, they are combined using the chain rule in (2.6). The resulting expression can be rewritten in terms of the control point coordinates (κ, λ, μ) and the summantion indicies thusly:

$$\begin{aligned} \frac{\partial C}{\partial P_{\kappa,\lambda,\mu}} = \frac{1}{N} \sum_{c}^{N_z} \sum_{b}^{N_y} \sum_{a}^{N_x} \frac{\partial C}{\partial \vec{v}(x, y, z)} \\ \times \sum_{n}^{3} \sum_{m}^{3} \sum_{l}^{3} \beta_l \left(\frac{a}{N_x} \right) \beta_m \left(\frac{b}{N_y} \right) \beta_n \left(\frac{c}{N_z} \right) \end{aligned} \tag{2.9}$$

where (a, b, c) are the un-normalized local coordinates of a voxel within its housing tile. Here (x, y, z) still represent the absolute coordinates of a voxel within the entire volume, which can now be defined in terms of the control point coordinates and summation indicies as:

$$x = N_x(\kappa - l) + a, \quad y = N_y(\lambda - m) + b, \quad z = N_z(\mu - n) + c \tag{2.10}$$

The coefficient values P that minimize the registration cost function are found using L-BFGS-B, a quasi-Newton optimizer suitable for either bounded or unbounded problems [54]. During each iteration, the optimizer chooses a set of coefficient values; for these coefficient values (2.1)-(2.5) and (2.6)-(2.8) are used to compute the cost and gradient, respectively. The cost and gradient values are transmitted back to the optimizer and the process is repeated for a set number of iterations or until the cost function converges to a local (or global) minimum.

2.2 Fast B-spline Registration for the GPU

The GPU is an attractive platform to accelerate compute-intensive algorithms (such as image registration) due to its ability to perform many arithmetic operations in parallel. Our GPU implementations use NVidia's Compute Unified Device Architecture (CUDA), a parallel computing interface accessible to software developers via a set of C programming language extensions. Algorithms written using CUDA can be executed on GPUs such as the Tesla C1060, which consists of 30 Streaming Multiprocessors (SMs) each containing 8 cores clocked at 1.5 GHz for a total of 240 cores. The CUDA architecture simplifies thread management by logically partitioning threads into equally sized groups called *thread blocks*. Up to eight thread blocks can be scheduled for execution on a single SM. In the context of image registration, a single thread is responsible for processing one voxel, and thus, a thread block is responsible for processing a group of voxels.

This section first briefly discusses the SIMD programming model using a simple example and then develops two GPU-based designs for B-spline registration.

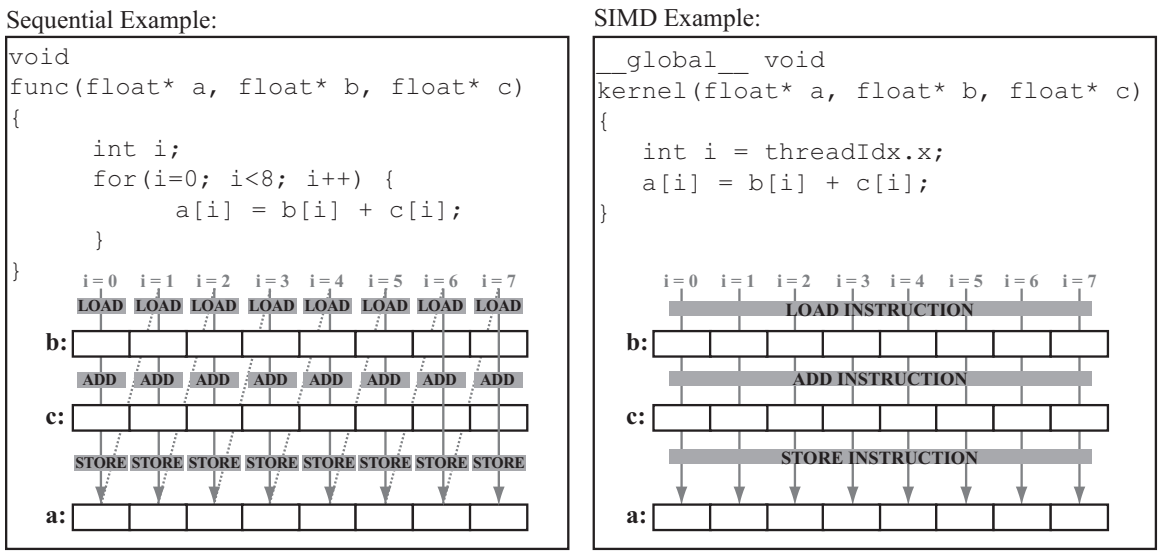


Figure 2.3: Comparison of sequential and SIMD. A simple example showing the differences between the sequential and SIMD programming models.

2.2.1 The SIMD Programming Model

The SIMD (single instruction multiple data) model structures programs for highly efficient computation on GPU cores. Here, all data is represented as an ordered set of the same type; for example, integers or floating-point numbers. Given an input data set we can develop a *kernel* that evaluates a function on each element of the input set so as to generate a corresponding output set. In the SIMD model all GPU cores execute the same instructions in lock-step, but they do so using different pieces of data.

Fig. 2.3 uses a simple example to show how SIMD differs from the sequential programming model used by the CPU. Both examples perform the same set of operations: they load elements from two input arrays, add them, and store the results into an output array. To generate elements of the output array, the sequential code in Fig. 2.3 steps through the input arrays, performing the same three operations for each set of elements, one set at a

time. The SIMD code in Fig. 2.3, however, operates on all array elements in parallel. A thread block with eight threads is dispatched to a core on the GPU. Though all of these threads execute the same kernel shown in Fig. 2.3, each thread obtains a unique identifer using `threadIdx.x`, an implicit or built-in variable provided by the CUDA run-time environment. Once the thread index is obtained, the operation is quite straightforward: thread i loads the i^{th} elements from arrays `b` and `c`, and generates the i^{th} element for the output array `a`.

Understanding the GPU memory model is crucial to developing high-performance SIMD programs. Modern GPUs provide three different memory types: global memory, texture memory, and shared memory. Communication between the GPU and the host CPU as well as communication between threads belonging to different thread blocks is done via the GPU's global memory. Though abundant—the Tesla C1060 has up to 4 GB of on-board RAM—global-memory accesses are quite slow compared to a core's clock speed. Loads/stores between GPU cores and global memory can be accelerated significantly if memory accesses are coalesced, meaning that sequentially numbered threads access sequential locations in global memory starting at an eight byte boundary (such as in the SIMD example in Fig. 2.3). If memory accesses are coalesced, the hardware can transfer data items requested by multiple threads in a single load/store operation. If threads access global memory in non-coalesced fashion, however, multiple load/store operations are needed to service these threads. In instances where coalesced operations are not possible, heavily referenced areas of global memory can be cached as read-only memory within the GPU's texture unit. However, due to the texture unit's 27-bit addressing scheme, arrays utilizing the texture until are limited to 134,217,728 elements, which is roughly equivalent to a 512^3 volume

containing intensity values or a 350^3 volume containing cost function gradient values.

Finally, for some problems, it may be necessary for threads within the same thread block to share intermediate results. Since individual threads with a thread block are assigned to and processed by a single SM, inter-thread communication within a block can be facilitated via a very fast memory channel known as shared memory, which is provided by the SM. However, shared memory must be used judiciously since only a modest amount is typically available; for example, even a high-end model such as the Tesla C1060 possesses only 16 KB of shared memory per SM. If, for example, each thread block requires more than 2KB of shared memory, then an SM will be unable to support its maximum of eight thread blocks due to insufficient shared memory resources. In this case, the number of thread bocks scheduled to an SM will be the maximum number allowed given the available shared memory resources.

2.2.2 Software Organization

Sections 2.2.3 and 2.2.4 develop two GPU implementations of B-spline registration using the software organization shown in Fig. 2.4. The spline interpolation as well as the cost function and gradient computations are performed on the GPU, while the optimization is performed on the CPU. During each iteration the optimizer, executing on the CPU, chooses a set of coefficient values to evaluate and transmits these to the GPU. The GPU then computes both the cost function and the cost function gradient, which are returned these to the optimizer. When a minima has been reached in the cost function gradient, the optimizer halts and invokes the interpolation routine on the GPU to compute the final deformation field.

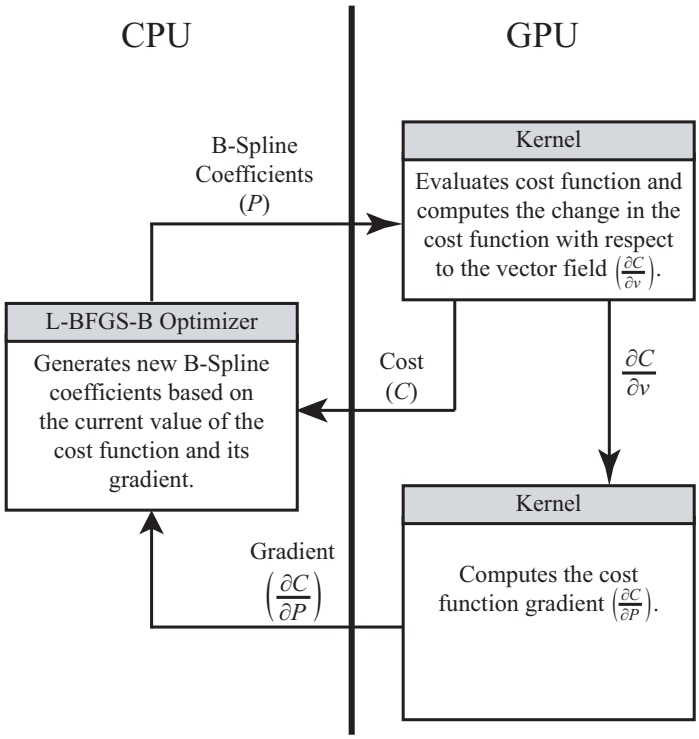


Figure 2.4: Software organization of GPU implementations. The software organization for B-spline registration in which the optimizer alone is executed on the CPU for greater flexibility.

Returning to Fig. 2.4, the value of the evaluated cost function C as well as its gradient $\partial C/\partial P$ must be transferred from the CPU to the GPU for every iteration of the registration process. Transfers between the CPU and GPU memories are the most costly in terms of time, and we have taken special care to minimize these types of transactions. The cost function is a single floating-point value and transferring it to the CPU incurs negligible overhead. The gradient, however, consists of three floating-point coefficient values for each control point in the grid. For example, registering two $256 \times 256 \times 256$ images with a control-grid spacing of $10 \times 10 \times 10$ voxels requires $73,167$ B-spline coefficients to be transferred between the GPU and the CPU per iteration, incurring about 0.35 ms over a PCIe 2.0 x16

bus[1]. Registering the same two volumes with a control-grid spacing of $30 \times 30 \times 30$ incurs 0.30 ms to transfer 5184 coefficients between the GPU and the CPU. Comparable transfer times are incurred in transferring the coefficients generated by the optimizer back to the GPU. Based on detailed profiling experiments, the CPU-GPU communication overhead demands roughly 0.14% of the total algorithm execution time. We therefore conclude these PCIe transfers deliver an insignificant impact on the overall algorithm performance even for high-resolution images with fine control grids.

Due to varying algorithmic requirements and the numerous hardware constraints involved in GPU programming, each algorithm must be re-targeted to the GPU manually, sometimes requiring multiple designs before an optimal design is obtained. We now discuss two specific implementations of the B-spline registration algorithm on the GPU. First, we detail a "naive" implementation, thus named due to its redundant operations and poor memory-access patterns when calculating the cost function gradient $\partial C/\partial P$. Then, we discuss in depth a highly optimized kernel to compute $\partial C/\partial P$.

2.2.3 The Naive GPU Implementation

The "naive" version performs straightforward parallelization with little regard for avoiding redundant computations and achieving coalesced memory-access patterns. The kernels comprising this implementation are very direct in their methodology, closely following the logic underlying the sequential algorithms. Loads from global memory are performed using the GPU's texture unit to reduce access latency. However, due to the somewhat random memory-access patterns exhibited by this "naive" implementation's cost function gradient

[1]The PCIe 2.0 x16 bus provides a maximum bandwidth of 8 gigabytes per second.

Kernel 1 Compute C and $\partial C/\partial \nu$ for a voxel.

```
1: /* Get T_G, the 1D index of the thread, and use it to obtain (x, y, z) for the voxel. */
2:
3: /* Normalized voxel coordinates (x, y, z) to within the range [0, 1] */
4: u = x/N_x − ⌊x/N_x⌋; v = y/N_y − ⌊y/N_y⌋; w = z/N_z − ⌊z/N_z⌋;
5:
6: /* Use (2.1) to obtain the displacement vectors for the voxel */
7: ν_x = ν_y = ν_z = 0;
8: for n = 0 to 3 step 1 do
9:    for m = 0 to 3 step 1 do
10:      for l = 0 to 3 step 1 do
11:         U = β_l(u)β_m(v)β_n(w);
12:         ν_x = ν_x + U × P_x(i + l);
13:         ν_y = ν_y + U × P_y(j + m);
14:         ν_z = ν_z + U × P_z(k + n);
15:      end for
16:   end for
17: end for
18:
19: /* Apply the deformation vector and compute the MSE score for the voxel */
20: D = S(x, y, z) − M(x + ν_x, y + ν_y, z + ν_z);
21: C(x, y, z) = D^2;
22:
23: /* Compute ∂C/∂ν⃗ for the voxel and store to GPU global memory */
24: ∂C/∂ν[3 × T_G + 0] = 2 × D × ∇M_x(x, y, z);
25: ∂C/∂ν[3 × T_G + 1] = 2 × D × ∇M_y(x, y, z);
26: ∂C/∂ν[3 × T_G + 2] = 2 × D × ∇M_z(x, y, z);
```

kernel, the speedup obtained by caching memory regions within the texture unit may vary widely and unpredictably for different volume sizes and control-grid resolutions. The GPU implementation follows the architecture described in Fig. 2.4: Kernel 1 calculates the cost function C and the $\partial C/\partial \vec{\nu}$ values, and Kernel 2 uses the $\partial C/\partial \vec{\nu}$ values to calculate the cost function gradient $\partial C/\partial P$.

Kernel 1 is launched with one thread per voxel in the static image S, and the variables (x, y, z) defining the coordinates of a voxel within the volume are derived from each thread's index T_G. As shown in the pseudo-code, the normalized coordinates (u, v, w) of the voxel within a tile are calculated using (2.3). Lines 7-17 of the kernel calculate the displacement vectors for the voxel using (2.1). Lines 20-21 apply the deformation vector $\vec{\nu}$ to the moving

image to calculate the intensity difference D between the static image S and moving image M for the voxel in question as well as the cost function C. Finally, lines 24-26 compute $\partial C/\partial \vec{\nu}$ using (2.8) and store the result to GPU global memory in an interleaved fashion. Calculating C and $\partial C/\partial \vec{\nu}$ exemplifies an algorithm that is easily parallelized on the GPU, and the strategy used by Kernel 1 performs very well. In fact, we reuse Kernel 1 in a slightly modified form for our optimized GPU implementation. The individual cost-function values computed for each voxel by Kernel 1 are accumulated using a sum reduction kernel to obtain the overall similarity metric C shown in (2.5).

Kernel 2 calculates the $\partial C/\partial P$ value for a control point as defined by (2.10) using the $\partial C/\partial \vec{\nu}$ values computed previously by Kernel 1. It is launched with as many threads as there are control points, where each thread computes $\partial C/\partial P$ for its assigned control point. Thus, the operations performed by a single thread to obtain $\partial C/\partial P$ for its control point are done serially, but $\partial C/\partial P$ is calculated in parallel for all control points in the grid. As shown in the pseudo-code, the variables (x, y, z) define the coordinates of a control-point within the volume and are derived from each thread's index T_G. We identify the 64 tiles influenced by the control point, and then for each tile perform the operations detailed in lines 4-26: (1) load the $\partial C/\partial \vec{\nu}$ value for each voxel from GPU memory and calculate the corresponding B-spline basis-function product, (2) compute $\partial C/\partial \vec{\nu} \times \beta_l(u)\beta_m(v)\beta_n(w)$, and accumulate the results for each spatial dimension as per the chain rule in (2.6). Once a thread has accumulated the results for all 64 tiles into registers A_x, A_y, and A_z, lines 29-31 interleave and insert these values into the $\partial C/\partial P$ array residing in GPU global memory.

Kernel 2 Compute the gradient $\partial C/\partial P$ for a control knot.

```
1: /* Use T_G, the 1D thread index, to obtain the (κ, λ, μ) coords of the control point */
2:
3: /* Iterate through the 64 tiles affecting this control point to calculate ∂C/∂P. */
4: A_x = A_y = A_z = 0;
5: for l = 0 to 3 step 1 do
6:    for m = 0 to 3 step 1 do
7:       for n = 0 to 3 step 1 do
8:          x_t = κ − n; y_t = λ − m; z_t = μ − l;
9:
10:         /* For each voxel in tile (x_t, y_t, z_t), compute ∂C/∂P using (2.6) and (2.7) */
11:         for k = 0 to N_z step 1 do
12:            for j = 0 to N_y step 1 do
13:               for i = 0 to N_x step 1 do
14:                  /* Get normalized local voxel coordinates (u, v, w) */
15:                  /*   and compute B-spline basis function product  */
16:                  U = β_l(u)β_m(v)β_n(w);
17:
18:                  /* Accumulate ∂C/∂P values. */
19:                  A_x = A_x + U × ∂C/∂ν_{x_t}(i);
20:                  A_y = A_y + U × ∂C/∂ν_{y_t}(j);
21:                  A_z = A_z + U × ∂C/∂ν_{z_t}(k);
22:               end for
23:            end for
24:         end for
25:
26:      end for
27:   end for
28: end for
29:
30: /* Store the ∂C/∂P solution for the control knot to GPU global memory. */
31: ∂C/∂P[3 * T_G + 0] = A_x;
32: ∂C/∂P[3 * T_G + 1] = A_y;
33: ∂C/∂P[3 * T_G + 2] = A_z;
```

2.2.4 The Optimized GPU Implementation

Though Kernel 2 details perhaps the most straightforward way of parallelizing $\partial C/\partial P$ calculations on the GPU, it has a serious performance deficiency in that the threads executing Kernel 2 perform a large number of redundant load operations from GPU global memory. We illustrate this problem using an example from Fig. 2.5. Consider the shaded tile shown in the top-left corner of the volume. The set of voxels within this tile are influenced by a set of 64 control points (of which eight are shown as black spheres). Conversely, voxels

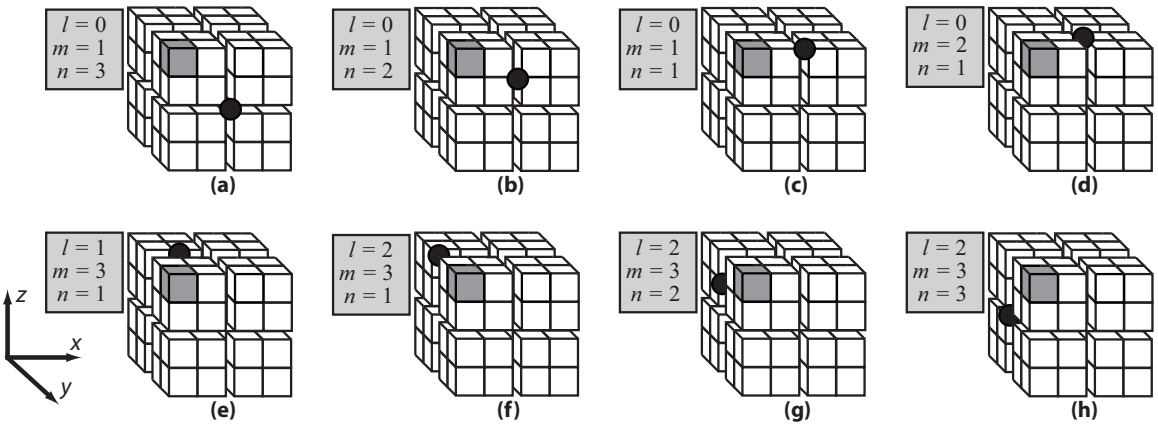


Figure 2.5: Visualization of tile influence on B-spline control points. Voxels within the shaded tile (in the top-left corner of the volume) are influenced by a set of 64 control points, of which eight are shown as black spheres. This tile partially contributes to the gradient values $\partial C/\partial P$ at each of these points. Figs. (a)–(h) show that the same tile is utilized in different relative positions with respect to each of the control points influencing it. So, each tile in the volume will be viewed in 64 unique ways by the corresponding 64 control points influencing it, which results in 64 unique (l, m, n) combinations being applied to each tile.

within this tile contribute a $\sum \partial C/\partial \vec{\nu} \times \partial \vec{\nu}/\partial P$ value to the gradient calculations of the respective 64 control points as per the chain rule in (2.6). Now, considering the control points shown in Fig. 2.5(b) and (c), the position of the tile relative to these two points is $(l = 0, m = 1, n = 2)$ and $(l = 0, m = 1, n = 1)$, respectively. This implies that though the two GPU threads computing the gradient for these control points use the same $\partial C/\partial \nu$ values from the tile, they must use *different* basis-function products when computing $\partial \vec{\nu}/\partial P$ to obtain their respective contributions to $\partial C/\partial P$ for the control points they are each workin on; the thread responsible for the control point in Fig. 2.5(b) will calculate the contribution of the highlighted tile to $\partial C/\partial P$ as

$$\sum_{N_w} \frac{\partial C}{\partial \vec{\nu}(x, y, z)} \beta_0(u)\beta_1(v)\beta_2(w), \tag{2.11}$$

whereas the thread processing the control point in Fig. 2.5(c) will compute the contribution

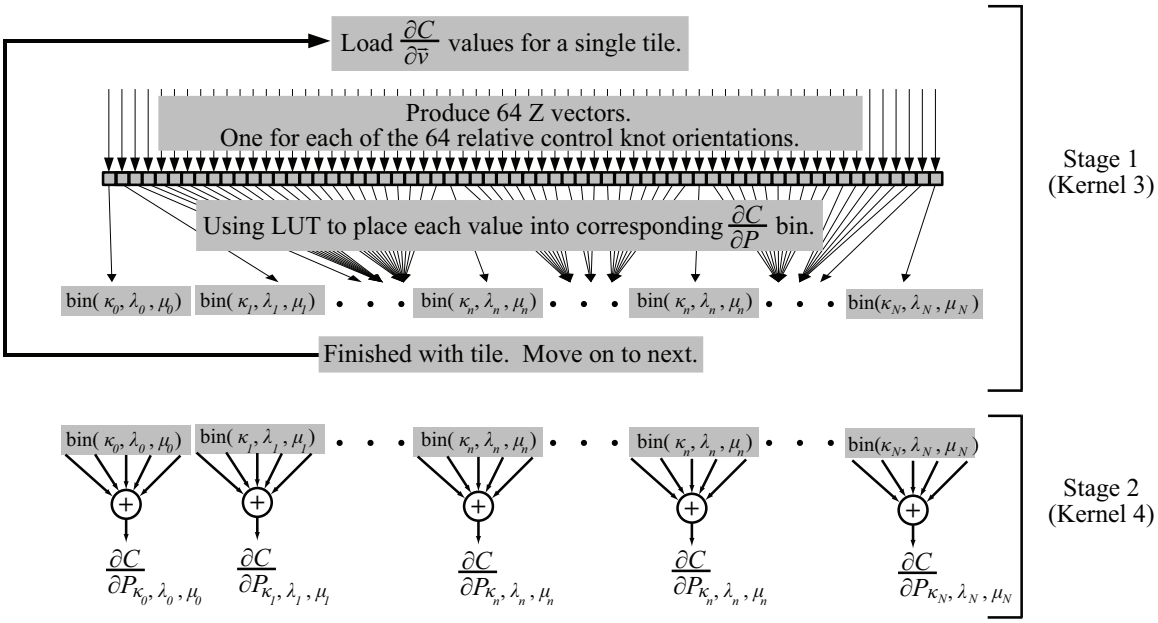


Figure 2.6: Highly parallel method of gradient computation. The flow corresponding to the "condense" process performed by the optimized GPU implementation. For each tile, we compute all 64 of its $\partial C/\partial P$ contributions to its surrounding control points. These partial contributions are then binned appropriately according to which control points are affected by the tile. We use (x_c, y_c, z_c) to denote the three dimensional coordinates of a control point within the volume. Notice how each control point is shown as having its own bin that stores all $\vec{Z}$ vectors that contribute to its cost function gradient.

of the highlighted tile to $\partial C/\partial P$ for its control point as

$$\sum_{N_w} \frac{\partial C}{\partial \vec{v}(x,y,z)} \beta_0(u)\beta_1(v)\beta_1(w). \tag{2.12}$$

Here, u, v, w represent the normalized position of a voxel within the tile. Since the two threads execute independently of each other and in parallel, each thread will end up loading $\partial C/\partial v$ values from the shaded tile separately. In general, given Kernel 2's design, every tile in the volume will be loaded 64 times by different threads during the process of computing $\partial C/\partial P$ values for the control points. Our goal, therefore, is to develop kernels that eliminate these redundant load operations.

The first step in developing kernels that compute $\partial C/\partial P$ efficiently is to reduce the large amount of $\partial C/\partial \vec{\nu}$ data generated by Kernel 1 residing in GPU global memory into smaller, more manageable units. Fig. 2.6 shows the overall flow comprising of two major stages. During the first stage, the $\partial C/\partial \vec{\nu}$ values corresponding to a tile are read from global memory in coalesced fashion. Since any given voxel tile is influenced by (and influences) 64 control points, it is subject to each of the 64 possible (l, m, n) configurations exactly once. This allows us to form intermediate solutions to (2.10) as follows, where for each tile, we obtain

$$\vec{Z}_{tile,l,m,n} = \sum_{z=0}^{N_z} \sum_{y=0}^{N_y} \sum_{x=0}^{N_x} \frac{\partial C}{\partial \vec{\nu}(x,y,z)} \beta_l(u)\beta_m(v)\beta_n(w). \tag{2.13}$$

The above operation is performed for the 64 possible (l, m, n) configurations, resulting in 64 $\vec{Z}$ values per tile where each $\vec{Z}$ is a partial solution to the gradient computation at a particular control point within the grid. Equation (2.13) can be implemented as a GPU kernel since multiple $\partial C/\partial \vec{\nu}$ tiles may be "condensed" in parallel due to the absence of any data dependencies between tiles. Moreover, once a $\partial C/\partial \vec{\nu}$ tile is read and condensed, it may be discarded since all relevant information required to compute $\partial C/\partial P$ is now represented by the $\vec{Z}$ values. Therefore, the optimized flow shown in Fig. 2.6 loads each $\partial C/\partial \vec{\nu}$ value from GPU global memory *only once*, unlike Kernel 2's design where each tile is loaded 64 times by different GPU threads.

Equation (2.13) is applied to each tile in the volume. Once the $\partial C/\partial \vec{v}$ values for a tile are condensed into 64 $\vec{Z}$ values, we consult a LUT that maps each $\vec{Z}$ value to one of the 64 control points influenced by the tile. Specifically, the output of this first stage is an

array of bins with each bin possessing 64 slots. Each control point in the grid has exactly one bin. For each of the 64 $\vec{Z}$ values computed by (2.13), the LUT provides not only the mapping to the appropriate control-point bin, but also the slot within that bin into which the $\vec{Z}$ value should be stored. Note that each of the 64 $\vec{Z}$ values generated from a single tile will not only be written to different control point bins, but to different slots within those bins as well – this property, in combination with each bin of 64 slots starting on an 8 byte boundary, allows us to adhere to the memory coalescence requirements imposed by the CUDA architecture. The second stage of the gradient computation simply sums the 64 $\vec{Z}$ values within each bin to calculate $\partial C/\partial P$ at each control point.

We now discuss the GPU kernels that implement the design flow shown in Fig. 2.6. As a first step, Kernel 1 is modified to store $\partial C/\partial \vec{\nu}$ values as three separate non-interleaved arrays whose values can be read in coalesced fashion. Kernel 3 is designed to be launched with 64 threads operating on a single tile. The outer-most loop iterates through the entire set of voxels within the tile in chunks of 64, and during each iteration of this loop, lines 5–6 load $\partial C/\partial \vec{\nu}$ for the current chunk of voxels into GPU shared memory. Each thread executes lines 14–16 to compute the $\partial C/\partial P$ value contributed by its voxel for the currently chosen basis-function product. These values are then accumulated into an array Q, indexed by the (l, m, n) combination, via a tree-style reduction in which all 64 threads contribute (lines 18–25). The inner-loops compute the next set of $\partial C/\partial P$ values corresponding to a different combination on the same batch of voxels.

The $\partial C/\partial P$ values, once computed, are placed into bins corresponding to the control points that influence the tile (lines 31–33). When executed on the NVidia Tesla C1060, approximately 15 tiles are processed in parallel at any given time.

Kernel 3 Optimized kernel design for computing the gradient $\partial C/\partial P$. Stage 1.

```
1: /* Get thread-block index B and the local thead index T. */
2:
3: /* Threads process a ∂C/∂v⃗ tile in groups of 64. All threads belonging to a thread block B
   work on the same tile whose index is denoted by O. This mapping is maintained in a lookup
   table LUT_Offset. */
4: for G = 0 to N_w/64 step 64 do
5:    O = LUT_Offset[B];
6:    α_x[T] = ∂C/∂ν_x[O + G + T]; α_y[T] = ∂C/∂v⃗_y[O + G + T]; α_z[T] = ∂C/∂ν⃗_z[O + G + T];
7:
8:    /* Obtain the normalized coordinates (u, v, w) for the voxel within the tile. Code is omitted.
      */
9:
10:   P = 0; // The (l, m, n) combination number, ranging from 0 to 63
11:   for n = 0 to 3 step 1 do
12:      for m = 0 to 3 step 1 do
13:         for l = 0 to 3 step 1 do
14:            U = β_l(u)β_m(v)β_n(w); // Evaluate the basis function product
15:            /* Store the ∂C⃗/∂P value contributed by this voxel. */
16:            R_x[T] = α⃗_x[T] × U; R_y[T] = α⃗_y[T] × U; R_z[T] = α⃗_z[T] × U;
17:
18:            /* Since there are 64 threads operating on different voxels, each thread will generate
               a ∂C/∂P value per voxel corresponding to the (l, m, n) combination. Reduce these
               values to a single value and store in R_x[0], R_y[0], and R_z[0]. Code is omitted. */
19:
20:            __syncthreads(); // Threads wait here until the reduction is complete
21:            /* Thread 0 accumulates the ∂C/∂P values corresponding to this (l, m, n) combina-
               tion. */
22:            if T = 0 then
23:               Q_x[P] = Q_x[P] + R_x[0]; Q_y[P] = Q_y[P] + R_y[0]; Q_z[P] = Q_z[P] + R_z[0];
24:            end if
25:            __syncthreads();
26:            P = P + 1; // Move on to the next combination
27:         end for
28:      end for
29:   end for
30: end for
31: /* Identify the 64 control points affecting the tile using LUT_CP and store ∂C/∂P values to the
   appropriate bins. */
32: K = LUT_CP[64 * B + T];
33: V_x[64 × K + T] = Q_x[T]; V_y[64 × K + T] = Q_y[T]; V_z[64 × K + T] = Q_z[T];
```

Kernel 4 implements the second stage of the flow in Fig. 2.6. It reduces the 64 $\partial C/\partial P$ values into a final gradient value for each control point. Lines 8–16 use shared memory to interleave the (x, y, z) components of the $\partial C/\partial P$ stream to improve the coalescence of write to GPU global memory in line 18. Kernel 4 is launched with as many threads as there are

Kernel 4 Optimized kernel design for computing the gradient $\partial C/\partial P$. Stage 2.

```
1: /* Get the thread index T and the thread-block index B for this thread. */
2: ξx[T] = Vx[64 × B + T]; ξy[T] = Vy[64 × B + T]; ξz[T] = Vz[64 × B + T];
3: __syncthreads();
4:
5: /* Reduce ξ⃗ and store results in ξ⃗[0]. Code is omitted. */
6:
7: /* Interleave gradient values in shared memory and store to GPU global memory. */
8: if T == 0 then
9:     ψ[0] = ξx[0];
10: end if
11: if T == 1 then
12:     ψ[1] = ξy[0];
13: end if
14: if T == 2 then
15:     ψ[2] = ξz[0];
16: end if
17: if T ≤ 2 then
18:     ∂C/∂P[3 × B + T] = ψ[T];
19: end if
```

control points.

To summarize, the optimized GPU implementation focuses primarily on restructuring the B-spline algorithm to use available GPU memory and processing resources as effectively as possible. We restructure the data flow of the algorithm so that loads from global memory are performed only once and in a coalesced fashion for optimal bus bandwidth utilization. Data fetched from global memory is placed into shared memory where threads within a thread block may quickly and effectively work together. Furthermore, for efficient parallel processing, we recognize the smallest independent unit of work is a tile. This leads to an interesting situation in which high-resolution control grids provide many smaller work units while lower-resolution ones provide fewer, but larger work units. So, high-resolution grids yield a greater amount of data parallelism than lower-resolution ones, leading to better performance on the GPU.

2.3 Performance Evaluation

We present experimental results obtained for the CPU and GPU implementations in terms of both execution speed and registration quality. We compare the performance achieved by six separate implementations: the single-threaded reference code, the multi-core OpenMP implementation on the CPU, and four GPU-based implementations. The GPU implementations are: the naive method comprising of Kernels 1 and 2, and three versions of the optimized implementation comprising of Kernels 1, 3 and 4—the first version uses a LUT of pre-computed basis-function products whereas the second version computes these values on the fly. The third version simply implements the standard code optimization technique of loop unrolling in an effort to maximize performance—the innermost loop (lines 13-27) of Kernel 3 is fully unrolled and the tree style sum reduction portrayed in line 18 is also fully unrolled. The reason for comparing the first two versions of the optimized GPU-based design is to experimentally determine if the GPU can evaluate the B-spline basis functions faster than the time taken to retrieve pre-computed values from the relatively slow global memory. We also quantify each implementation's sensitivity to both volume size as well as control-point spacing (i.e., the tile size). The tests reported in this section were performed on a machine with two Intel Xeon E5540 processors (a total of eight CPU cores), each clocked at 2.5GHz, 24 GB of RAM, and an NVidia Tesla C1060 GPU card. The Tesla GPU contains 240 cores, each clocked at 1.5 GHz, and 4 GB of onboard memory.

2.3.1 Registration Quality

Fig. 2.7 shows the registration of two $512 \times 512 \times 128$ CT images of a patient's thorax on the GPU. The image on the left is the reference image, captured as the patient was fully

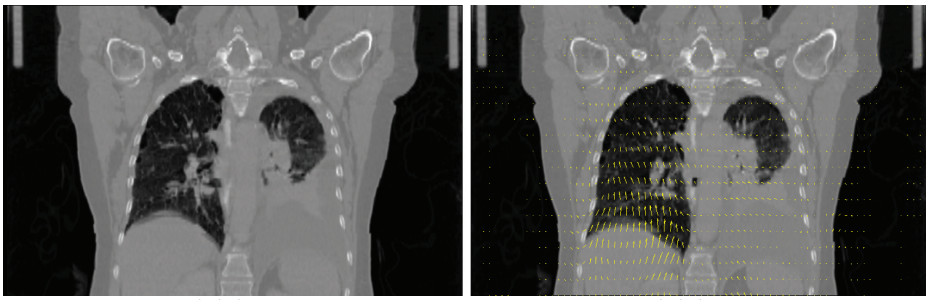


Figure 2.7: Unimodal inhaled to exhaled lung registration. Deformable registration result for two 3D CT images. Deformation vector field is shown superimpoed upon inhaled image. Performed using optimized GPU implementation.

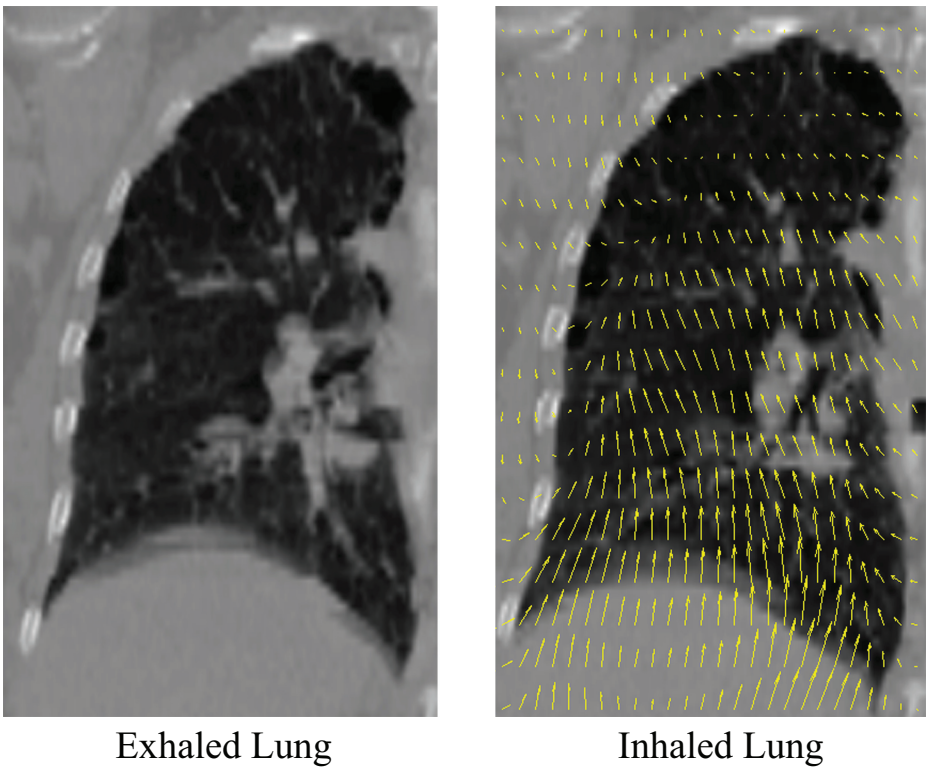


Figure 2.8: Unimodal inhaled to exhaled lung registration (Zoom View). An expanded view of the deformable registration result. The superimposed deformation field shows how the inhaled lung has been warped to register to the exhaled lung.

exhaled, and the image on the right is the moving image, captured after the patient had fully inhaled. The resulting vector field after registration is overlaid on the inhale image. Fig. 2.8 is a zoomed-in view of Fig. 2.7, focusing on just the left lung. To determine the

registration quality, we generate the deformation field by running the registration process for 50 iterations and then compare the results against the reference implementation. The multi-core versions generate near-identical vector fields with an RMS error of less than 0.014 with respect to the reference.

2.3.2 Sensitivity to Volume Size

We test each algorithm's sensitivity to increasing volume size by holding the control-point spacing constant at 10 voxels in each physical dimension while increasing the size of synthetically generated input volumes in steps of $10 \times 10 \times 10$ voxels. For each volume size, we record the execution time taken for a single registration iteration to complete. Fig. 2.9 shows the results for each of the five implementations. The plot on the left compares all five implementations, where we see that the execution time increases linearly with the number of voxels in a volume. The multi-core implementations provide an order of magnitude improvement in execution speed over the reference implementation. For large volume sizes around 350^3, the most highly optimized GPU implementation achieves a speedup of 15x compared to the reference code, whereas the muli-core CPU implementation achieves a speedup proportional to the number of CPU cores (8x when executed on our dual Xeon E5540 4-core processors). Furthermore, note that the naive GPU implementation cannot handle volumes having more than 4.3×10^7 voxels. Recall that Kernel 2 suffers from a serious performance flaw: redundant and coalesced loads of $\partial C/\partial \vec{v}$ values from GPU global memory. Using the texture unit as a cache provides a method of mitigation, but the resulting speedup varies unpredictably with control-grid resolution (see Fig. 2.10). Moreover, the texture unit cannot cache very large volumes, limiting the maximum size that the naive

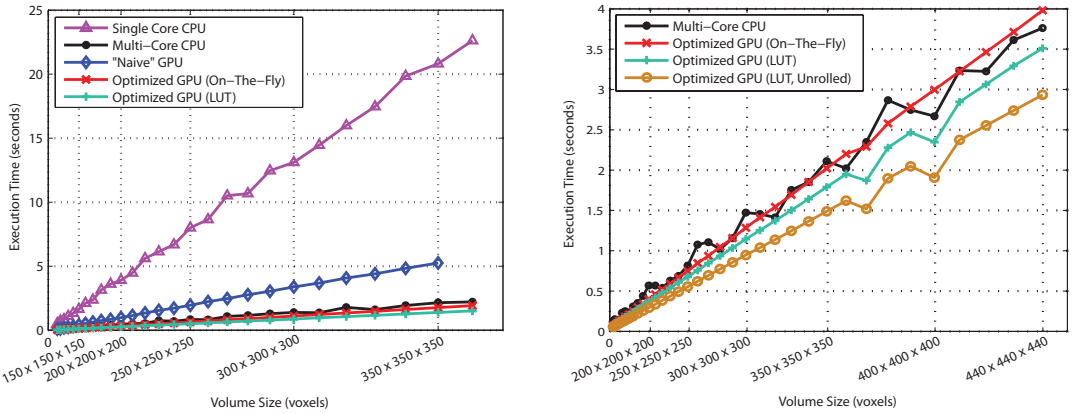


Figure 2.9: Unimodal algorithm execution times vs image volume size. (Left) The execution time for a single registration iteration is shown as a function of volume size. The control-point spacing is fixed at $10 \times 10 \times 10$ voxels. (Right) Execution time versus volume size for the various multi-core implementations.

implementation can correctly process to about 350^3 voxels.

The plot on the right focuses in on just the multi-core CPU and GPU designs. The optimized GPU implementation that uses a LUT to evaluate the B-spline basis-function product outperforms the version that computes these values on the fly by nearly 8% for large input volumes. Further optimization by unrolling the main gradient computation loop in Kernel 3 results in an additional 14% increase in speed. This unrolled LUT-based implementation achieves a speedup of approximately 1.5 times over the multi-core CPU implementation for large volumes, making it about 15 times faster than the reference implementation.

2.3.3 Sensitivity to Control-Point Spacing

The optimized GPU design achieves short iteration times by assigning individual volume tiles to processing cores as the basic work unit. Since tile size is determined by the spacing between control points, we investigated whether the execution time is sensitive to the

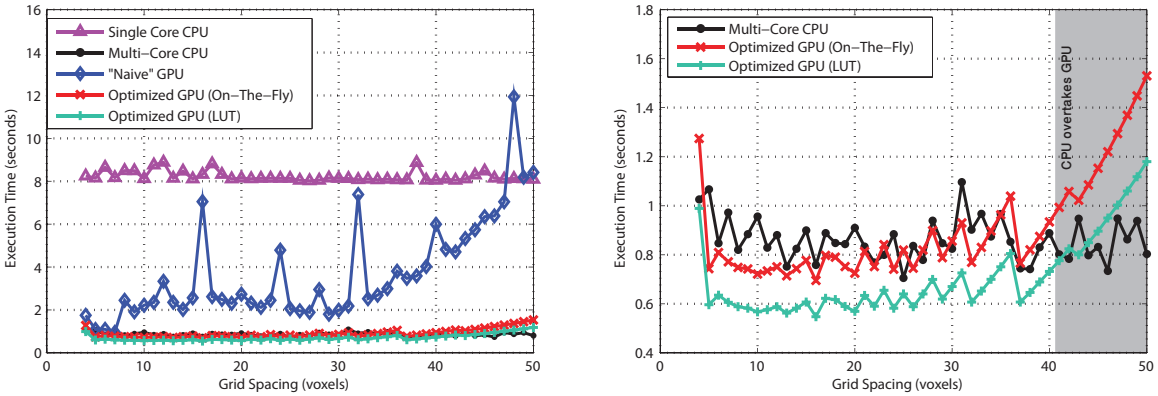


Figure 2.10: Unimodal algorithm execution times vs B-spline grid resolution. (Left) The execution time for a single registration iteration is shown as a function of the control-point spacing (same in all three dimensions). The volume size is held constant at $256 \times 256 \times 256$ voxels. (Right) The execution time versus control-point spacing for the multi-core implementations.

control-point spacing. The left plot in Fig. 2.10 shows the impact of different grid spacings on our B-spline implementations when the volume size is fixed at $256 \times 256 \times 256$ voxels. Notice that all implementations, except for the naive GPU version, are agnostic to spacing.

The right plot in Fig. 2.10 focuses just on the multi-core designs. Interestingly, the multi-core CPU implementation outperforms the optimized GPU implementations for coarse control grids, starting at a spacing of about 40 voxels. The higher-clocked CPU cores process these significantly larger tiles more rapidly than the lower-clocked GPU cores. So, for practitioners doing multi-resolution registration, the coarser control grids can be handled by the CPU whereas the GPU-based design can be invoked as the control-point spacing becomes finer.

2.4 Conclusions

We have developed a grid-alignment technique and associated data structures that greatly reduce the complexity of B-spline based registration. We have then used the main ideas

underlying the aligned-grid method to develop highly-parallel and scalable designs for computing the score and cost-function gradient on multi-core processors. We have demonstrated the speed and robustness of our parallelization strategy via experiments using both clinical and synthetic data. When compared to a highly optimized sequential implementation, the multi-core CPU version achieves a linear speedup when executed on eight cores. The GPU version, when executed on the Tesla 1060 GPU with 240 SIMD cores, achieves a speedup of 15 times over the sequential code. Our experiments also demonstrate a fairly strong independence between the B-spline grid resolution and execution time for our parallel algorithms.

The presented method for accelerating the cost function gradient computation on multi-core processors provides a solid foundation for developing fast parallel registration using mutual information as a similarity metric. Due to the cost function independent nature of Kernels 3 and 4, these highly efficient data parallel algorithms may easily be applied to any similarity for which the change in the cost function can be analytically expressed with respect to the vector field. Consequently, we are currently in the process of developing such a GPU-accelerated mutual information deformable registration technique. Additionally, the performance results obtained for the OpenMP based multi-core CPU implementation are nearly comparable to those provided by the Telsa C1060. We are currently in the processes of extending this multi-core implementation to use vectorized extensions such as SSE in order to provide an even more complete comparison with respect to state-of-the-art graphics processing units.

CHAPTER 3: MULTI-MODAL B-SPLINE REGISTRATION

This chapter describes how the B-spline registration algorithm may be extended to perform multi-modal image registration by utilizing the statistically based mutual information similarity metric. Modifications to the algorithm structure and data flow presented in Chapter 2 are discussed in detail, and strategies for accelerating these new algorithmic additions are explored. Specific attention is directed towards developing fast and memory efficient data parallel methods of constructing marginal and joint image intensity histograms, since these data structures are key to successfully performing this statistically based image registration method. The impact of the mutual information similarity metric on the analytic formalism driving the vector field evolution is covered in depth. Consequently, the partial volume interpolation method is also introduced; dictating how the image intensity histogram data structures evolve with the vector field evolution. Single-core CPU, multi-core CPU and many-core GPU based implementations are benchmarked for performance using synthetic image volumes. Finally, quality is assessed by example through a multi-modal thoracic MRI to CT deformable registration.

3.1 Overview of Multi-Modal B-spline Registration

The B-spline deformable registration algorithm maps each and every voxel in a static image S to a corresponding voxel in a moving image M as described by a deformation field $\vec{\nu}$ which

is defined at each and every voxel within the static image. An optimal deformation field accurately describes how the voxels in M have been displaced with respect to their original positions in S. The existence of such an optimal and physically meaningful deformation field assumes that the two images represent the same underlying physiology. If the images are obtained using the same imaging method, the registration is said to be uni-modal and the quality of the deformation field is assessed using the sum of squared differences between the intensity values of voxels in the static image and the corresponding voxels in the warped moving image. Alternatively, images obtained using differing imaging methods must be matched using multi-modal registration. The registration modality is important since assessing the quality of the deformation field for multi-modal registrations requires more complex methods than those required by uni-modal registration. This is due to the involved images having different color spaces which are not guaranteed to possess any type of linear or one-to-one mapping. Mutual information and normalized mutual information are widely-used similarity metrics when registering multi-modality images in which the mutual information quantifies the amount of information content common to the two images [55]. The images will be optimally aligned when the shared information content is maximized

Fig. 3.1 shows the overall process used for registering multi-modality images in this paper, comprising of the following major steps: 1) generating a deformation field using the B-spline coefficients, 2) applying the deformation field to the moving image, 3) generating voxel-intensity histograms for both the static and deformed moving images as well as their joint histogram, 4) computing the mutual information using the histograms to assess the registration quality, 5) computing the change in the mutual information with respect to the B-spline coefficients, and 6) generating a new set of B-spline coefficients. The above process

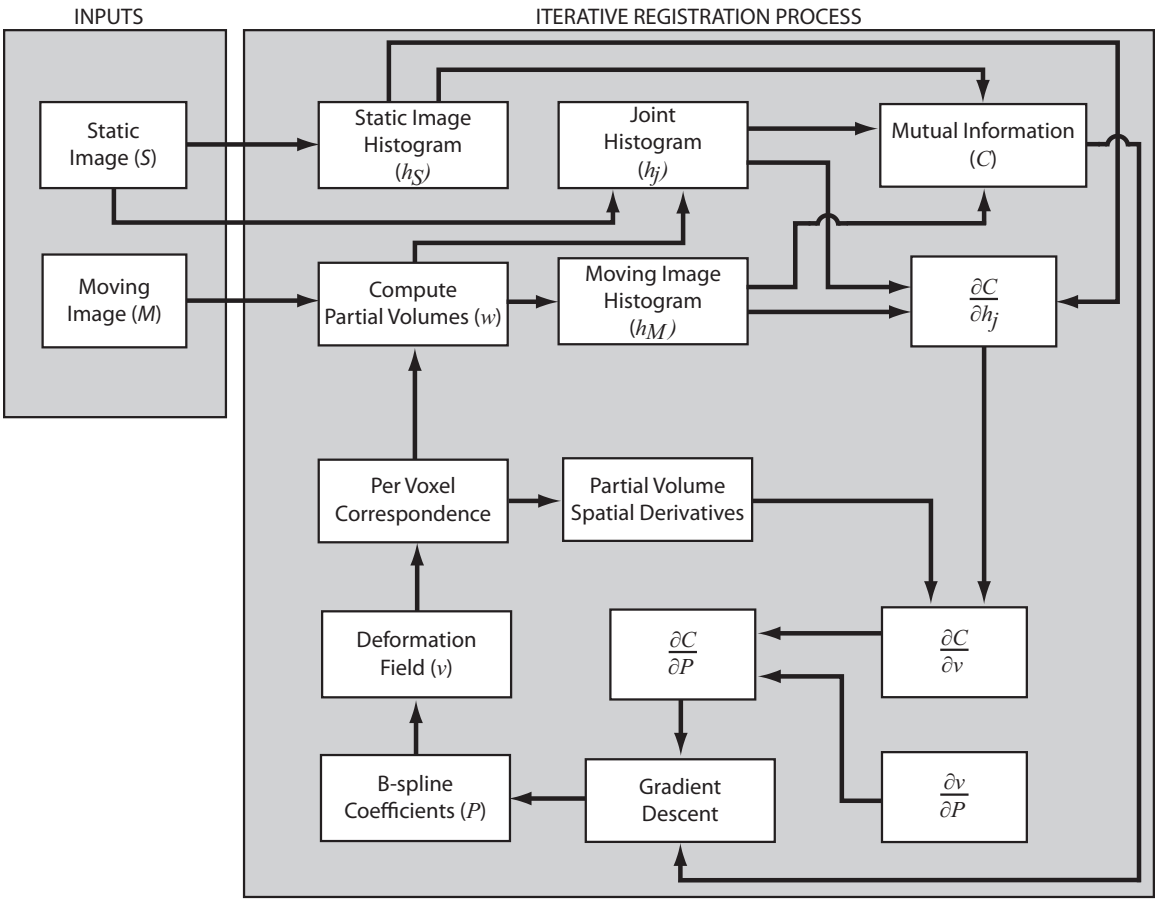


Figure 3.1: Flowdiagram of mutual information based registration. The overall process used to register multi-modality images using mutual information as the similarity metric. All major steps in this process (except for the gradient descent optimization) have been parallelized on multi-core processors including the GPU.

is repeated until an optimal deformation field is obtained that warps the moving image such that it is most similar to the static image. The number of iterations required depends on factors such as the severity of the initial misalignment, the complexity of local deformations in the patient's anatomy, and the level of accuracy the end-user deems necessary.

Each iteration of the process shown in Fig. 3.1 optimizes the deformation field $\vec{\nu}$, resulting in a more accurate mapping or correspondence of voxels in the static image to coordinates within the moving image. Any given voxel in the static image can map to

a point lying between multiple voxels in the moving image; in 3D images for example, a voxel in the static image can map to eight neighboring voxels in the moving image. The case of "one-to-many" correspondence is handled via a technique called partial volume interpolation, originally proposed by Maes et al. [56], and discussed in greater detail in Section 3.1.2. Once the correspondence has been performed for a voxel and the partial volumes have been computed, the intensity histograms for the static and moving images, as well as the joint histogram, are updated appropriately. These histograms capture the entropy in the individual images as well as the joint entropy describing the amount of uncertainty when considering both images as a joint system. For 3D images, this means updating one static-image histogram bin, eight moving-image histogram bins, and eight joint-histogram bins. The completed histograms are then used to compute the mutual information, which measures how similar the static image is to the moving image (after the moving image is subjected to $\vec{\nu}$).

The best registration is obtained by modifying the deformation field $\vec{\nu}$ so as to maximize the mutual information. This process can be posed as an optimization problem. However, since medical-image volumes can be quite large[1] and since the deformation field is defined at every voxel, operating on the vector field directly is a problem too large to handle even for modern computers. For faster computation, the deformation field $\vec{\nu}$ can be parameterized using a sparse number of B-spline coefficients which results in a compressed representation of the deformation field. The problem then becomes one of optimizing the B-spline coefficients $\vec{P}$ to maximize the mutual information cost function C. Performing this optimization via gradient descent (or quasi-Newtonian) methods requires that we know how the cost function

[1] A typical image volume has a resolution of $512 \times 512 \times 128$ voxels, or about 33 million voxels.

C changes with respect to the B-spline coefficients $\vec{P}$. The steps needed to obtain this derivative $\frac{\partial C}{\partial P}$ are also outlined in Fig. 3.1 and are described in greater detail in Section 3.2.4.

3.1.1 Using B-splines to Represent the Deformation Field

Given a number of uniformly-spaced discrete control points, a second-order continuous function involving these points can be described using uniform cubic B-spline basis functions. Describing a function in this fashion is advantageous when the desired function is unknown but we are required to maximize an optimization condition while maintaining second-order continuity. In deformable image registration, the deformation field $\vec{\nu}$ that maps voxels in the static image to voxels in the moving image must maintain this level of smoothness; yet the form of $\vec{\nu}$ is not known when starting the registration process since $\vec{\nu}$ depends on the geometry of the anatomy being registered. It is therefore advantageous to parameterize the dense deformation field $\vec{\nu}$ using a sparse set of control points, which are uniformly distributed throughout the fixed image's voxel grid. The placement of control points forms two grids that are aligned to one another: a dense voxel grid and a sparse control-point grid. As shown in Fig. 3.2, the control-point grid partitions the voxel grid into equally sized regions called tiles. The deformation field $\vec{\nu}$ can be found at any given voxel within a tile by performing B-spline interpolation using control points with local support regions that include the tile. Since the local support region for a cubic spline curve involves four control points in each of the three dimensions, computing a single point in the displacement field involves the 64 control points found in the immediate vicinity of a voxel's housing tile. Also, since three coefficients, p_x, p_y, and p_z, are associated with each control point, the interpolation uses 192 coefficients.

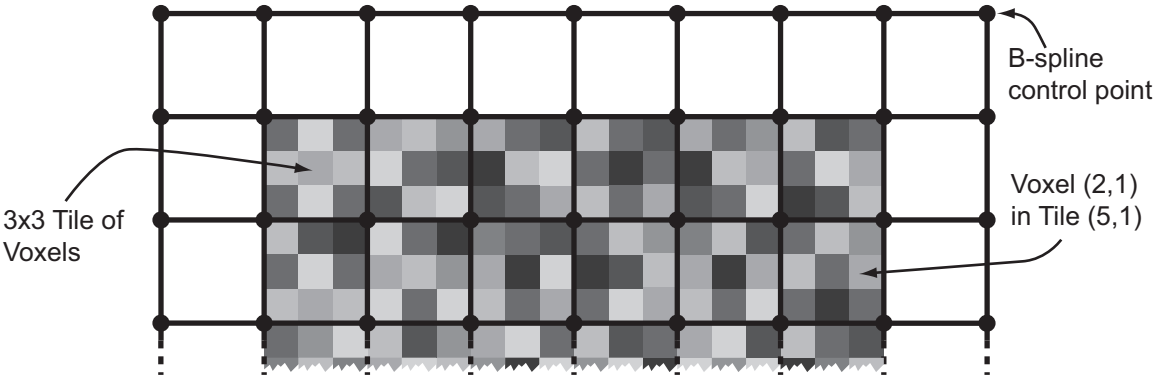


Figure 3.2: Superimposition of control-point and voxel grids. The grid of uniformly spaced B-spline control points partitions the voxel grid into equally sized tiles. In this example, each tile is three voxels wide and three voxels tall. The number of control points in the x-dimension is three greater than the number of tiles in the x-dimension. Although not shown, this is true for all dimensions.

Mathematically, the x-component of the deformation field for a voxel located at coordinates $\vec{x} = (x, y, z)$ in the fixed image can be described as

$$\nu_x(\vec{x}) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3}\beta_i(u)\beta_j(v)\beta_k(w)p_x(l,m,n). \tag{3.1}$$

The components in the y and z directions are defined similarly. The symbols l, m, and n are indices for control points in the neighborhood of the tile of interest, and in the 3D case there are 64 combinations for l, m, and n. If (N_x, N_y, N_z) are the dimensions, in voxels, of a tile, then $\left\lfloor \frac{x}{N_x} \right\rfloor - 1$, $\left\lfloor \frac{y}{N_y} \right\rfloor - 1$, and $\left\lfloor \frac{z}{N_z} \right\rfloor - 1$ denote the x, y, and z coordinates, respectively, of the tile in the volume within which a voxel $\vec{x}$ falls, and the set of control points indexed by l, m, and n is

$$l = \left\lfloor \frac{x}{N_x} \right\rfloor - 1 + i, \quad m = \left\lfloor \frac{y}{N_y} \right\rfloor - 1 + j, \quad n = \left\lfloor \frac{z}{N_z} \right\rfloor - 1 + k. \tag{3.2}$$

In (3.1), β_i is the B-spline basis function along the x-direction given as

$$\beta_i(u) = \begin{cases} \frac{(1-u)^3}{6} & : i = 0 \\ \frac{3u^3-6u^2+4}{6} & : i = 1 \\ \frac{-3u^3+3u^2+3u+1}{6} & : i = 2 \\ \frac{u^3}{6} & : i = 3, \end{cases} \tag{3.3}$$

with β_j and β_k defined similarly in the y and z directions, respectively. Finally, $\vec{q} = (u, v, w)$ denotes the local coordinates of voxel $\vec{x}$ within its housing tile where

$$u = \frac{x}{N_x} - \left\lfloor \frac{x}{N_x} \right\rfloor, \quad v = \frac{y}{N_y} - \left\lfloor \frac{y}{N_y} \right\rfloor, \quad w = \frac{z}{N_z} - \left\lfloor \frac{z}{N_z} \right\rfloor. \tag{3.4}$$

Since the basis function is only defined within the range between 0 and 1, the local coordinates are appropriately normalized to fall within this range.

Representing the dense deformation field as a sparse set of B-spline coefficients is akin to information compression and the deformation field is optimized by modifying only its compressed form. In other words, the deformation field is never directly modified but is always tuned via the B-spline coefficient values. Obtaining the deformation field from B-spline coefficients is akin to a decompression operation, which is needed to check the registration quality. Fig. 3.3 shows the process of obtaining a deformation vector at a single voxel. The vector's coordinate, $\vec{x}$, is specified in terms of the coordinate pair $(\vec{p}, \vec{q})$. The tile $\vec{p}$ within which the deformation vector is computed determines the set of 64 B-spline control points involved in the decompression operation. The local coordinate $\vec{q}$ is used to retrieve the pre-computed evaluation of the B-spline basis function stored within the lookup tables `LUT_Bspline_x`, `LUT_Bspline_y`, and `LUT_Bspline_z`. Line 20 uses the control-point indices (l, m, n) and the dimensions of the control-point grid to compute a one-dimensional index into the data structure used to store the B-spline coefficients (shown in Fig. 3.4).

```
1: function decompress_vector (c⃗, N⃗, p⃗, q⃗)
2:   /* vector c⃗ contains the control-point grid dimensions */
3:   /* vector N⃗ contains the tile dimensions */
4:   /* vector p⃗ contains the voxel's tile coordinates */
5:   /* vector q⃗ contains the voxel's local coordinates within the tile */
6:   /* Returns ν⃗, the displacement vector at voxel */
7:
8:   ν⃗ = 0.0
9:   for k = 0 to 3 step 1 do
10:    n = p_z + k
11:    β_n = LUT_Bspline_z[k × N_z + q_z]
12:    for j = 0 to 3 step 1 do
13:      m = p_y + j
14:      β_m = LUT_Bspline_y[j × N_y + q_y]
15:      for i = 0 to 3 step 1 do
16:        l = p_x + i
17:        β_l = LUT_Bspline_x[i × N_x + q_x]
18:
19:        /* Get index into coefficient look up table clut, given l, m, n */
20:        cidx = 3 × ((n × c_x × c_y) + (m × c_x) + l)
21:
22:        /* Add the control point's contribution to displacement vector */
23:        Q = β_l × β_m × β_n
24:        ν_x = ν_x + Q× clut[cidx + 0]
25:        ν_y = ν_y + Q× clut[cidx + 1]
26:        ν_z = ν_z + Q× clut[cidx + 2]
27:      end for
28:    end for
29:  end for
30:  return ν⃗
31: end function
```

Figure 3.3: Obtaining a deformation vector at a given voxel

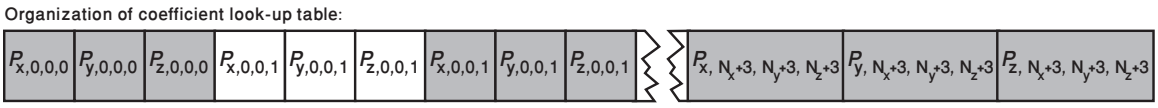


Figure 3.4: Organization and memory layout of the coefficient look-up table. The number of control points in the control grid is greater than the number of tiles by three in each dimension as shown in Fig. 3.2.

Finally, lines 23 through 26 accumulate the contribution of control point (l, m, n) to the three components of $\vec{\nu}(\vec{x})$.

Obtaining the entire deformation field is a simple matter of applying the technique

shown in Fig. 3.3 for every voxel $\vec{x}$ in the static image S. The most effective SIMD-style threading model on the GPU to obtain the correspondence between the static and moving images is to assign one thread per voxel in S. Given an execution grid of threads, each thread uses its unique identifier within the grid to locate the voxel and compute the voxel's tile and local coordinates, $\vec{p}$ and $\vec{q}$, respectively. Once these coordinates are obtained, each thread can decompress the vector at its voxel location in parallel using the operations listed in Fig. 3.3. Once a thread has obtained the deformation vector $\vec{\nu}$, it continues to work independently to find the correspondence in the moving image, which will consist of a group of eight voxels. The thread then computes the partial volumes associated with this neighborhood of voxels and accumulates them into the image histograms. The lookup tables `LUT_Bspline_x`, `LUT_Bspline_y`, `LUT_Bspline_z`, and `clut` are stored as textures to accelerate memory reads through the caching provided by the GPU's texture unit.

3.1.2 Mutual Information as a Cost Function

A cost function is used to determine the quality of the deformation field $\vec{\nu}$, which is equivalent to assessing the registration quality since $\vec{\nu}$ directly determines the voxel correspondence between the static and moving images. Since higher-quality deformation fields result in greater similarity between the static and moving images, the cost function is also referred to in the literature as the similarity metric. The cost function for assessing the quality of a unimodal registration simply accumulates the square of the intensity difference between the static image S and the moving image M subject to the deformation field $\vec{\nu}$ as

$$\frac{1}{N}\sum_{z}\sum_{y}\sum_{x}(S(x,y,z)-M(x+\nu_x,y+\nu_y,z+\nu_z))^2, \tag{3.5}$$

where N is the total number of voxels mapping from S to M. However, this cost function cannot be used to access the quality of a deformation field that is attempting to register images acquired using different imaging modalities since these images may have differing voxel intensity maps for identical anatomy. For such multi-modality registrations, the more sophisticated cost function of mutual information (MI) may be used which quantifies the amount of information content the two images share in common; the images will be optimally aligned when the shared information content is maximum [55]. To understand MI as a cost function, consider the intensity a of a voxel located at coordinates $\vec{x}$ within the static image, $a = S(\vec{x})$, and the intensity b of a voxel at coordinates $\vec{y}$ within the moving image, $b = M(\vec{y})$. The goal is to apply a coordinate transform $\mathbf{T}(\vec{y})$ to the moving image such that it registers best with the static image. The statistical MI is obtained as

$$I = \sum_{\mathbf{a},\mathbf{b}} p_j(a, \mathbf{T}(b)) \ln \frac{p_j(a, \mathbf{T}(b))}{p_S(a) p_M(\mathbf{T}(b))}, \tag{3.6}$$

which depends on the probability distributions of the voxel intensities in the static and moving images. So, we can view a and b as random variables with associated probability distribution functions $p_S(a)$ and $p_M(b)$, respectively, and joint probability $p_j(a, b)$. Applying the spatial transformation $\mathbf{T}(\vec{y})$ to M modifies $p_j(a, b)$ and this effect is implied using the notation $p_j(a, \mathbf{T}(b))$. Furthermore, if $\mathbf{T}$ results in voxels being displaced outside the moving image, $p_M(\mathbf{T}(b))$ will change, and if $\mathbf{T}$ results in a voxel being remapped to a location that falls between points on the voxel grid, some form of interpolation must be employed to obtain b, which will modify $p_M(b)$ as well. These effects are implied using the notation $p_M(\mathbf{T}(b))$.

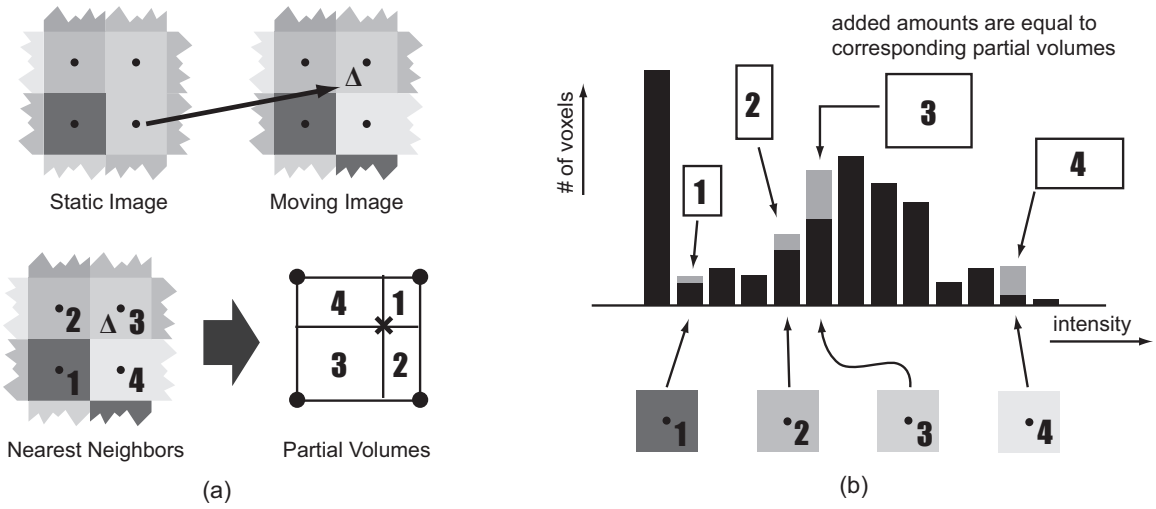


Figure 3.5: Partial volume interpolation. (a) Partial volume interpolation for 2D images using a neighborhood of four voxels. Notice that the first partial volume is the smallest since the first neighbor is the furthest away from the interpolation point Δ. (b) The nearest-neighbor voxels are binned according to their intensity values. The amount added to each bin is determined by each voxel's corresponding partial volume which is equivalent to adding fractional voxels to each involved histogram bin.

The interpolation method used to obtain b, given $\mathbf{T}(\mathbf{y})$, is important both in terms of execution speed and solution convergence. Our implementation uses the partial volume interpolation (PVI) method proposed by Maes et al. [56]. Fig. 3.5 shows an example of computing partial volumes for 2D images in which the deformation vector has mapped a pixel in the static image to a point falling within a neighborhood of four pixels in the moving image. The pixel centers are shown as black circles and the interpolation point is denoted by Δ. The interpolation method divides the volume defined by the four neighboring voxels into corresponding partial volumes that share the interpolation point as a common point. Once the partial volumes are computed, they are placed into the histogram bins of the corresponding voxels as shown in Fig. 3.5(b).

For 3D images, PVI is performed using a neighborhood of eight voxels where the partial

```
1: /* Compute partial volumes */
2: function compute_pv (Δ⃗)
3: /* Here { } is the sawtooth function (i.e. {x} = x − ⌊x⌋) */
4:    w_0 = (1 − {Δ_x}) × (1 − {Δ_y}) × (1 − {Δ_z})
5:    w_1 = (0 + {Δ_x}) × (1 − {Δ_y}) × (1 − {Δ_z})
6:    w_2 = (1 − {Δ_x}) × (0 + {Δ_y}) × (1 − {Δ_z})
7:    w_3 = (0 + {Δ_x}) × (0 + {Δ_y}) × (1 − {Δ_z})
8:    w_4 = (1 − {Δ_x}) × (1 − {Δ_y}) × (0 + {Δ_z})
9:    w_5 = (0 + {Δ_x}) × (1 − {Δ_y}) × (0 + {Δ_z})
10:   w_6 = (1 − {Δ_x}) × (0 + {Δ_y}) × (0 + {Δ_z})
11:   w_7 = (0 + {Δ_x}) × (0 + {Δ_y}) × (0 + {Δ_z})
12:   return w⃗
13: end function
14:
15: /* Computes indices of the eight nearest neighbors */
16: function find_nearest_neighbors (Δ⃗, M_X, M_Y)
17: /* M_X and M_Y are the dimensions of the moving image in the x and y
    directions, respectively */
18:   n_0 = (⌊Δ_z⌋ × M_X × M_Y) + (⌊Δ_x⌋ × M_X) + ⌊Δ_x⌋
19:   n_1 = n_0 + 1
20:   n_2 = n_0 + M_X
21:   n_3 = n_2 + 1
22:   n_4 = n_0 + M_X × M_Y
23:   n_5 = n_4 + 1
24:   n_6 = n_4 + M_X
25:   n_7 = n_6 + 1
26:   return n⃗
27: end function
```

Figure 3.6: Computation of partial volumes and nearest neighbors

volumes, w_0 through w_7, are defined in terms of the interpolation point $\mathbf{\Delta}$ as shown by the `compute_pv` function in Fig. 3.6. Note that $\sum_{i=0}^{7} w_i = 1$. Once the partial volumes have been computed, they are placed into the histogram bins of the corresponding voxels: partial volume w_0 is placed into the histogram bin associated with neighboring voxel n_0, w_1 with n_1, and so on. The indices of the bins are computed using the `find_nearest_neighbors` function, also listed in Fig. 3.6. The PVI technique is used to compute both $p_M(\mathbf{T}(b))$ and $p_j(a, \mathbf{T}(b))$. Since the static image is not subject to the coordinate transform $\mathbf{T}$, PVI does not apply when generating $p_S(a)$. However, if $\mathbf{T}$ results in a voxel $\vec{x}$ within S mapping outside of M, then that voxel is not included in the distribution $p_S(a)$. Such voxels cannot

be registered, and so are excluded when computing the cost function and related items such as intensity distributions.

Given that the coordinate transformation $\mathbf{T}$ is defined by the deformation field $\vec{\nu}$ such that $\mathbf{T}(b) = M(\mathbf{T(y)}) = M(\vec{x} + \vec{\nu}) = M(\vec{\Delta})$, an algorithm to compute the mutual information cost function C is best implemented by modifying (3.6) as

$$C = \frac{1}{N} \sum_{j=0}^{K_S} \sum_{i=0}^{K_M} h_j(i,j) \ln \frac{N \times h_j(i,j)}{h_S(j) \times h_M(i)} \tag{3.7}$$

where the probability distributions $p_S(a)$, $p_M(\mathbf{T}(b))$, and $p_j(a, \mathbf{T}(b))$ are constructed as image histograms h_S, h_M, and h_j consisting of K_S, K_M, and $K_S \times K_M$ bins, respectively. Also, (3.7) incorporates N, the number of voxels being registered, thereby allowing the use of unnormalized histograms (which reduces the number of division operations during histogram generation).

3.2 Efficient Computation of Mutual Information

Evaluating the MI-based cost function in (3.7) requires constructing the image histograms h_S, h_M, and h_j. Generating these histograms using a serial (or single-threaded) program, as shown in Fig. 3.7, is straightforward. First, the voxel $a = S(\vec{x})$ found at the tail of the deformation vector located at $\vec{x}$ is processed for inclusion in the static-image histogram $h_S(a)$. This is a simple matter of determining which bin the intensity value a falls within, and incrementing it by one (lines 6 and 7). The second operation is to compute the coordinates of the eight corresponding voxels, n_0 through n_7, associated with $\mathbf{T}(\vec{y})$ within the moving image by looking at the head of the deformation vector $\vec{\nu}$ with the tail placed at $\vec{x}$ (line 10). Similarly, the partial volumes, w_0 through w_7, are obtained in line 11 for PVI.

1: /* Calculate the appropriate bin in the static-image histogram and increment it */
2: /* $h_S[\ ]$ is an array containing histogram values */
3: /* B_S is the destination bin for voxel $a = S(\vec{x})$ */
4: /* O_S is the minimum static-image voxel value */
5: /* D_S is the histogram bin spacing */
6: $B_S = \lfloor (S(\vec{x}) - O_S)/D_S \rfloor$
7: $h_S[B_S] = h_S[B_S] + 1$
8:
9: /* Use the deformation vector $\vec{\nu}$ to find nearest neighbors and partial volumes */
10: $\vec{n} =$ find_nearest_neighbors $(\vec{x} + \vec{\nu}, M_X, M_Y)$
11: $\vec{w} =$ compute_pv $(\vec{x} + \vec{\nu})$
12:
13: /* Add partial volumes to the moving-image histogram and the joint histogram */
14: /* $h_M[\ ]$ is an array containing the moving-image histogram values */
15: /* $h_J[\ ]$ is a 2D array of values in the joint histogram */
16: /* B_M is the destination bin for voxel $b = S(\vec{n_x})$*/
17: /* O_M is the minimum moving-image voxel value */
18: /* D_M is the histogram bin spacing */
19: **for** $i = 0$ to 7 step 1 **do**
20: $B_M = \lfloor (M(n_i) - O_M)/D_M \rfloor$
21: $h_M[B_M] = h_M[B_M] + w_i$
22: $h_j[B_M][B_S] = h_j[B_M][B_S] + w_i$
23: **end for**

Figure 3.7: Serial method of histogram construction

(The expanded definitions of functions find_nearest_neighbors and compute_pv can be found in Fig. 3.6.) For each of the eight voxels, the associated bin within the moving-image histogram h_M is incremented by the corresponding partial volume (lines 20 and 21). Additionally, the joint-histogram bins of interest are easily found using the appropriate bin within h_S and the eight bins within h_M. Each of these bins within the joint histogram h_j is incremented by the appropriate partial volume (line 22). After the above-described process is performed for every voxel in the static image possessing a correspondence, the image histograms are complete[1].

The algorithm listed in Fig. 3.7 is invoked for each vector $\vec{\nu}$ in the deformation field

[1] Voxels mapping to coordinates outside the moving image have no correspondence.

and since the number of vectors equals the number of voxels found in the static image, this algorithm must be invoked N times. When trying to improve computational efficiency, the algorithm cannot be simply invoked in parallel across N threads due to write hazards associated with histogram construction wherein two or more threads attempt to increment the same histogram bin simultaneously. We use two separate thread-safe techniques: one targeting h_S and h_M, and the other targeting h_j to construct the image histograms in parallel on the GPU. Since both methods make effective use of the memory hierarchy available within the GPU, we familiarize the reader with this topic via the following brief discussion. The interested reader is referred to Kirk and Hwu for more details [57].

The memory hierarchy within a GPU comprises of registers, shared memory, and global memory. Registers provide the fastest access but are also the most scarce. They exhibit thread-level scope, meaning every thread is assigned a set of registers that store data that is private to that thread. Shared memory is the fastest memory type accessible to multiple threads; it exhibits what is known as thread-block scope. Since GPU kernels can comprise of thousands of threads, these threads are grouped into many smaller sets called thread blocks of up to 512 threads each, and each thread block is assigned a modest amount of shared memory that allows the threads within the block to communicate quickly with each other. The size of the shared memory assigned to a thread block ranges from 16KB to 48KB on various GPU platforms[1]. Finally, ranging on the order of gigabytes, global memory is the largest yet slowest memory available. It is accessible to every thread, which provides a means for threads blocks to communicate with each other. Furthermore, global memory is

[1]The Tesla C1060 GPU limits the size of shared memory available to each thread block to 16KB whereas the C2050 GPU provides up to 48KB to each thread block.

how the CPU and GPU exchange data and it remains persistent between multiple kernel invocations. Consequently, kernels generally begin with a read from global memory and end with a write to global memory.

3.2.1 Constructing Histograms for the Static and Moving Images

This technique partitions an image into many non-overlapping subregions. Each subregion is assigned to a unique thread block which then generates a histogram for its assigned subregion of the image, where the size of a subregion (in voxels) equals the number of threads within a thread block. Fig. 3.8 describes the operations performed by a thread block computing the moving-image histogram, beginning with each thread obtaining the deformation vector $\vec{\nu}$ corresponding to its assigned voxel $\vec{x}$ within the subregion delegated to the thread block. Given $\vec{\nu}$, each thread computes the eight nearest neighbors in the moving image corresponding to the static-image voxel $\vec{x}$ and the weights associated with each of these neighbors by computing the partial volumes (lines 4 and 5). Upon reaching line 8, each thread has local copies of 16 items: the indices of the nearest neighbors and the associated weights. At this point, all threads simultaneously place each of the eight weights into the moving-image histogram bins associated with the intensity values of the nearest neighbors (lines 9 through 13).

Since all threads within a thread block perform the operations listed in Fig. 3.8 concurrently, we ensure that threads wishing to modify the same histogram bin do not modify the same memory location simultaneously. If a thread block has N_B threads, we divide each bin into N_B partitions as shown in Fig. 3.9. This data structure, `s_partitions`, resides within the GPU's shared memory and allows each thread to have its own copy of each histogram

```
1: /* Note:  Each thread is assigned a deformation vector ν⃗ */
2:
3: /* Each thread finds nearest neighbors and partial volumes */
4: n⃗ = find_nearest_neighbors (x⃗ + ν⃗, M_X, M_Y)
5: w⃗ = compute_pv (x⃗ + ν⃗)
6:
7: /* Accumulate weights into shared memory array s_partitions */
8: /* Here threadIdx is the index of the thread within the thread block */
9: for i = 0 to 7 step 1 do
10:   B_M = ⌊(M(n_i) − O_M)/D_M⌋
11:   idx = threadIdx + B_M × threadsPerBlock
12:   s_partitions[idx] = s_partitions[idx] + w_i
13: end for
14:
15: /* Synchronize threads to this point */
16: __syncthreads()
17:
18: /* Assign each thread to a single sub-histogram bin */
19: if threadIdx < num_bins then
20:   sum = 0.0
21:   element = (threadIdx) AND (0x0F)
22:   offset = threadIdx × threadsPerBlock
23:
24:   /* Merge bin partitions */
25:   for i = 0 to (threadsPerBlock − 1) step 1 do
26:     sum = sum + s_partitions[offset + element]
27:     element = element + 1
28:     if element = threadsPerBlock then
29:       element = 0
30:     end if
31:   end for
32:   /* Each bin has now been merged */
33:
34:   /* Write merged bins to sub-histogram for this thread block */
35:   /* Here, blockIdxInGrid denotes the index of a thread block within the grid
      of thread blocks */
36:   sub_hist[blockIdxInGrid * num_bins + threadIdx] = sum
37: end if
```

Figure 3.8: Parallel histogram construction using sub-histograms

bin, which prevents write collisions. On line 16, all threads within the thread block are synchronized to ensure that each has incremented its personal copy of the moving-image histogram before moving on to the next step: merging the partitions within each bin. This operation, shown in lines 19 through 36, assigns one thread to each bin and since the num-

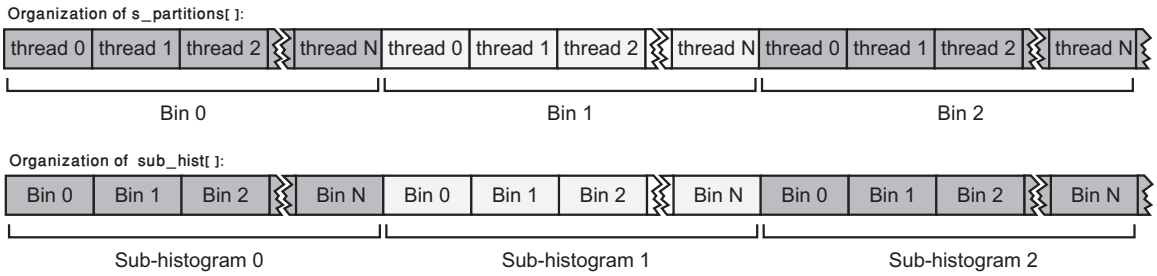


Figure 3.9: Memory organization of sub-histogram method. Graphical representation of the memory layout for arrays `s_partitions[ ]` and `sub_hist[ ]` used in Fig. 3.8.

ber of partitions equals the number of threads within the thread block, each thread merging a partition performs N_B accumulation operations to complete the process. All bins can be processed in parallel in this fashion since there are no dependencies between bins. However, some special considerations must be taken due to the way shared memory is organized. Shared memory is organized as 16 banks of 1KB memory each. If two threads attempt to read from the same memory bank simultaneously, the reads become serialized, which negatively impacts performance. We, therefore, aim to minimize these bank conflicts by starting each thread off on a different bank as shown in line 21. Thread 0 sums its N_B partitions starting with partition 0, residing in the first bank; thread 1 starts with partition 1, residing in the second bank, sums through to N_B, and then "wraps around" to end on partition 0, and so on. This way, each thread responsible for merging the partitions within a bin in `s_partitions` will always read from a different shared-memory bank when generating a histogram with up to 16 bins. For histograms with more than 16 bins, bank conflicts will occur when reading, but they will be minimal; to construct an 18-bin histogram, for example, threads 0 and 16 will read bank 1 simultaneously, and threads 1 and 17 will read bank 2 simultaneously, while the reads issued by threads 2 through 15 will remain free of

conflicts.

Once each thread has merged the partitions of a bin down to a single value (lines 25 through 31), the threads copy the histogram bins to the `sub_hist` array residing in the GPU's global memory (line 36), where thread block 0 writes out the sub-histogram 0, thread block 1 writes out sub-histogram 1, and so on. Once all thread blocks have written their sub-histograms to `sub_hist`, a simple tree-style sum reduction kernel is used to merge these sub-histograms into one histogram that is representative of the moving image M.

3.2.2 Constructing the Joint Histogram

The above-described method cannot be used to generate the joint histogram h_j since GPUs typically do not have enough shared memory to maintain individual copies of the joint histogram for each thread within a thread block[1]. Therefore, the proposed method, detailed in Fig. 3.10, relies on atomic operations that have become available in recent GPU models to guarantee mutually exclusive access to histogram bins, and requires considerably less shared memory.

The most popular GPU models in use today (the Tesla C1060 and the GTX 200 series of GPUs) do not support atomic addition operations on floating-point values residing in global or shared memories. However, these models support atomic exchange operations on floating-point values residing in shared memory; one can safely swap a value in a shared-memory location with a register value that is private to the thread. The technique discussed

[1]Consider generating a 20×20 joint histogram on the Tesla C1060 where each bin value is a four-byte floating-point data type. If we wish to apply the method detailed in Fig. 3.8, each thread would require a dedicated data structure of 1.6KB. Since shared memory per thread block is limited to 16KB, this limits the number of threads per block to about 10, which is impractical from a computational viewpoint.

```
1: /* Note:  Each thread is assigned a deformation vector ν⃗ */
2:
3: /* Each thread finds nearest neighbors and partial volumes */
4: n⃗ = find_nearest_neighbors (x⃗ + ν⃗, M_X, M_Y)
5: w⃗ = compute_pv (x⃗ + ν⃗)
6:
7: /* Compute the fixed histogram bin and joint offset */
8: B_S = ⌊(S(x⃗) − O_S)/D_S⌋
9: offset = B_S × K_M
10:
11: /* Add partial volumes to joint histogram */
12: for i = 0 to 7 step 1 do
13:    B_M = ⌊(M(n_i) − O_M)/D_M⌋
14:    idx = offset + B_M
15:    if idx != inferred_bin then
16:       success = FALSE
17:       while success == FALSE do
18:          val =atomicExch(s_joint[idx], −1)
19:          if val! = −1 then
20:             success = TRUE
21:             val = val + w_i
22:             atomicExch(s_joint[idx], val)
23:          end if
24:       end while
25:    end if
26: end for
27:
28: /* Copy sub-histogram from shared to global memory */
29: chunks = (K_J × block_size − 1) ÷ block_size
30: for i = 0 to chunks step 1 do
31:    idx = threadIdx + i × block_size
32:    if idx < K_J then
33:       j_hist[j_stride + idx] = s_joint[idx]
34:    end if
35: end for
```

Figure 3.10: Parallel histogram construction using atomic exchange

below uses the concept of atomic exchange to regulate access to histogram bins and avoid write conflicts between multiple threads[1].

The image is once again divided into subregions, each of which is assigned to individual thread blocks to generate the corresponding sub-histograms. However, instead of maintain-

[1] Atomic arithmetic operations on floating-point values are available on the recently released Fermi series of GPUs, for example, the GTX 400 series and the Tesla C2050 family.

ing a separate partition for each thread within a bin, all threads write to the same bin in shared memory. So, for a joint histogram with $K_J = K_S \times K_M$ bins, K_J elements of shared memory are allocated per thread block. This array, `s_joint`, holds the entire sub-histogram for the thread block. Write conflicts to the same bin are handled using the atomic-exchange instruction, as shown in lines 16 through 24. The GPU instruction `atomicExch(`x`, `y`)` allows a thread to swap a value x in a shared-memory location with a register value y that is private to the thread, while returning the previous value of x. If multiple threads attempt to exchange their private values with the same memory location simultaneously, it is guaranteed that only one will succeed. Returning to line 18, the successful thread obtains a private copy of the histogram value in `val`, leaves the value -1 in the shared-memory location `s_joint[idx]`, and proceeds to lines 20 through 23; other threads attempting to access the same bin simultaneously will obtain the -1 previously placed into shared memory. This technique, therefore, provides an efficient mechanism of serializing threads attempting to write to the same memory location simultaneously. Note that the threads proceeding to line 20 increment the histogram value `val` obtained from shared memory by the appropriate partial volume weight (line 21), exchange the incremented value back into shared memory (thus removing the -1 placed earlier), and set their `success` flag to `TRUE` to indicate that their contributions to the joint histogram have been committed. Finally, lines 29 through 35 copy the sub-histogram from shared memory to the GPU's global memory.

When generating the joint histogram, we also perform a simple but key optimization step to improve computational efficiency. Since medical images generally contain a predominant intensity—for example, black, which is the intensity value of air, is abundant in most CT scans—the technique presented here can result in the serialization of many threads if they all

update histogram bins involving this color. We prevent this situation, however, by inferring the value of the bin corresponding to the predominant color since this bin is expected to cause the most write conflicts. Since the sum of all unnormalized histogram bins must equal the total number of voxels in the static image having a correspondence within the moving image, one bin may be omitted during the histogram construction phase and filled in later using the simple relationship

$$h_j(\texttt{inferred\_bin}) = N - \sum_i h_j(i) \tag{3.8}$$

where `inferred_bin` is the bin that is skipped in line 15 of Fig. 3.10. Initially, an educated guess is made for `inferred_bin` based on the imaging modality, but as the registration is performed over multiple iterations, the largest bin is tracked and skipped for the next iteration. Experimental results using CT images indicate a noticeable speedup, since on average, 80% of GPU threads attempt to bin values correlating to air which would otherwise be serialized.

3.2.3 Evaluating the Cost Function

Once the histograms are generated, evaluating the MI-based cost function is straightforward, consisting of simply cycling through these histograms while accumulating the results of the computation into C as in (3.7). (Care must be taken, however, to avoid evaluating the natural logarithm of zero in instances where a joint-histogram bin is empty.) Since the operation does not substantially benefit from parallelization, it is performed on the CPU. Moving the histogram data from the GPU to the CPU requires negligible time since even

large histograms incur very small transfer times on a modern PCI bus. Once evaluated, a single cost value is copied back to the GPU for use in subsequent operations.

3.2.4 Optimizing the B-spline Coefficients

Since we have chosen the coordinate transformation $\mathbf{T}(\vec{y}) = \vec{x} + \vec{\nu}$, where $\vec{\nu}$ is parameterized in terms of the sparse B-spline coefficients $\vec{P}$, it follows that the mutual information can be maximized by optimizing these coefficients. We choose to perform this optimization via the method of gradient descent for which an analytic expression for the gradient $\partial C/\partial \vec{P}$ is required at every control point $\vec{P}$. The expression $\partial C/\partial \vec{P}$ can be separated into partial derivatives using the chain rule:

$$\frac{\partial C}{\partial \vec{P}} = \frac{\partial C}{\partial \vec{\nu}} \times \frac{\partial \vec{\nu}}{\partial \vec{P}} \tag{3.9}$$

where the first term depends on the similarity metric. The second term depends on the parameterization of the deformation field $\vec{\nu}$ and is easily obtained by taking the derivative of (3.1) with respect to $\vec{P}$ as

$$\frac{\partial \vec{\nu}}{\partial \vec{P}} = \sum_{l=0}^{3} \sum_{m=0}^{3} \sum_{n=0}^{3} \beta_l(u)\beta_m(v)\beta_n(w). \tag{3.10}$$

In the first term of (3.9), C and $\vec{\nu}$ are coupled through the probability distribution p_j and are therefore directly affected by the partial volume interpolation. This becomes clearer when $\partial C/\partial \vec{\nu}$ is further decomposed as

$$\frac{\partial C}{\partial \vec{\nu}} = \frac{\partial C}{\partial p_j(a, M(\vec{\Delta}))} \times \frac{\partial p_j(a, M(\vec{\Delta}))}{\partial \vec{\nu}}$$
$$= \sum_{x=0}^{7} \left(\frac{\partial C}{\partial p_j(a, M(n_x))} \times \frac{\partial w_x}{\partial \vec{\nu}} \right), \tag{3.11}$$

where $M(\vec{\Delta})$ is the value of the voxel in the moving image that corresponds to the static image voxel $a = S(\vec{x})$. However, since $\vec{\Delta}$ falls between voxels in the moving image, eight moving-image voxels of varying weights are taken to correspond to $\vec{x}$ due to the partial volume interpolation, resulting in the simplification shown in (3.11). The first term of (3.11) is obtained using the derivative of (3.6) with respect to the joint distribution p_j as

$$\frac{\partial C}{\partial p_j(a, M(n_x))} = \ln \frac{p_j(a, M(n_x))}{p_S(a) p_M(M(n_x))} - C. \tag{3.12}$$

The second term describes how the joint distribution changes with the vector field. Recall that the displacement vector locally transforms the coordinates of a voxel in the moving image M such that $\vec{\Delta} = \vec{x} + \vec{\nu}$. As the vector field is modified, the partial volumes, w_0 through w_7, to be inserted into the moving-image and joint histograms h_M and h_j will change in size. Therefore, $\partial p_j(a, M(\vec{\Delta}))/\partial \vec{\nu}$ is determined by changes exhibited in the partial volumes w_0 through w_7 as $\vec{\Delta}$ evolves with the governing deformation field $\vec{\nu}$. These changes in the partial volumes with respect to the deformation field, $\partial w_x/\partial \vec{\nu}, x \in \{0, 7\}$, with respect to each of the Cartesian directions are easily obtained, thus resulting in 24 expressions. (The mathematical expressions for w_0 through w_7 can be found in the `compute_pv` function shown in Fig. 3.6.) So, for partial volume w_0:

$$\frac{\partial w_0}{\partial \nu_x} = (-1) \times (1 - \{\Delta_y)\} \times (1 - \{\Delta_z)\}, \tag{3.13}$$

$$\frac{\partial w_0}{\partial \nu_y} = (-1) \times (1 - \{\Delta_x)\} \times (1 - \{\Delta_z)\}, \tag{3.14}$$

$$\frac{\partial w_0}{\partial \nu_z} = (-1) \times (1 - \{\Delta_x)\} \times (1 - \{\Delta_y)\}, \tag{3.15}$$

and similarly for w_1 through w_7. Therefore, as prescribed by (3.11), computing $\partial C/\partial v$ at a given voxel $\vec{x}$ in S involves cycling through the eight bins corresponding to the neighbors described by $\vec{\Delta}$. So, for the first neighbor n_0, we determine which bin B_{M_0} within histogram h_M the voxel value n_0 belongs. This gives $h_M(B_{M_0})$. Similarly, the bin B_S within the static image histogram h_S associated with the static image voxel $a = S(\vec{x})$ is easily obtained, thus giving $h_S(B_S)$. Knowing B_S and B_{M_0} gives the associated joint histogram value $h_j(B_S, B_{M_0})$. Now, $\partial C/\partial p_j$ for neighbor n_1 is obtained as

$$\frac{\partial C}{\partial p_j(a, M(n_0))} = \ln \frac{h_j(B_S, B_{M_0})}{h_S(B_S) h_M(B_{M_0})} - C. \tag{3.16}$$

As prescribed by (3.11), the contribution of nearest neighbor n_0 and its associated partial volume w_0 on $\partial C/\partial \vec{\nu}$ is found by first computing $\partial w_0/\partial \vec{x}$ as in (3.15). Each of the three components of $\partial w_0/\partial \vec{x}$ are weighted by (3.16), leading to

$$\frac{\partial C}{\partial \nu_x} = \left(\frac{\partial w_0}{\nu_x} \times \frac{\partial C}{\partial p_j}\bigg|_{n_0} \right) + \left(\frac{\partial w_1}{\nu_x} \times \frac{\partial C}{\partial p_j}\bigg|_{n_1} \right) + \ldots \tag{3.17}$$

$$\frac{\partial C}{\partial \nu_y} = \left(\frac{\partial w_0}{\nu_y} \times \frac{\partial C}{\partial p_j}\bigg|_{n_0} \right) + \left(\frac{\partial w_1}{\nu_y} \times \frac{\partial C}{\partial p_j}\bigg|_{n_1} \right) + \ldots \tag{3.18}$$

$$\frac{\partial C}{\partial \nu_z} = \left(\frac{\partial w_0}{\nu_z} \times \frac{\partial C}{\partial p_j}\bigg|_{n_0} \right) + \left(\frac{\partial w_1}{\nu_z} \times \frac{\partial C}{\partial p_j}\bigg|_{n_1} \right) + \ldots \tag{3.19}$$

1: $\vec{n}$ = `find_nearest_neighbors` $(\vec{\Delta}, M_X, M_Y)$
2:
3: `/* Compute partial volumes spatial derivatives */`
4: $\partial\vec{w}/\partial x$ = `compute_pv_derivatives_x` $(\vec{\Delta})$
5: $\partial\vec{w}/\partial y$ = `compute_pv_derivatives_y` $(\vec{\Delta})$
6: $\partial\vec{w}/\partial z$ = `compute_pv_derivatives_z` $(\vec{\Delta})$
7:
8: `/* Calculate static image histogram bin */`
9: $B_S = \lfloor (S(\vec{x}) - O_S)/D_S \rfloor$
10:
11: `/* Compute` $\partial C/\partial\vec{\nu}$ `at voxel coordinate` $\vec{x}$ `*/`
12: **for** $i = 0$ to 7 step 1 **do**
13: $B_M = \lfloor (M(n_i) - O_M)/D_M \rfloor$
14: $\partial C/\partial p_j = \ln((N \times h_j[B_M][B_S])/(h_S[B_S] \times h_M[B_M])) - C$
15: $\partial C/\partial\nu_x = \partial C/\partial\nu_x + \partial w_i/\partial x \times \partial C/\partial p_j$
16: $\partial C/\partial\nu_y = \partial C/\partial\nu_y + \partial w_i/\partial y \times \partial C/\partial p_j$
17: $\partial C/\partial\nu_z = \partial C/\partial\nu_z + \partial w_i/\partial z \times \partial C/\partial p_j$
18: **end for**

Figure 3.11: Computation of cost derivative with respect to vector field

which gives $\partial C/\partial\vec{\nu}$ at the static-image voxel coordinate $\vec{x}$. This operation is performed for all N voxels in S.

The operations needed to compute $\partial C/\partial\vec{\nu}$ are performed in parallel by assigning a GPU thread to each voxel in the static image that has a correspondence in the moving image. Fig. 3.11 shows the operations performed by each thread. Once $\partial C/\partial\vec{\nu}$ has been computed at every voxel, we can now use (3.9) to describe how the cost function changes with the B-spline coefficients $\vec{P}$ associated with each control point. Fig. 3.12 shows an example of how the cost-function gradient is obtained at a single control point, highlighted in white, in a 2D image. Here, $\partial C/\partial\vec{\nu}$ has been computed at all voxels, including the hatched voxel shown in the zoomed-in view at local coordinates (1,1) within tile (0,0). The location of this hatched voxel's tile with respect to the highlighted control point results in the evaluation of the B-spline basis function with $l = 0$ and $m = 0$ in the x and y dimensions, respectively. Moreover, these evaluations are performed using the normalized

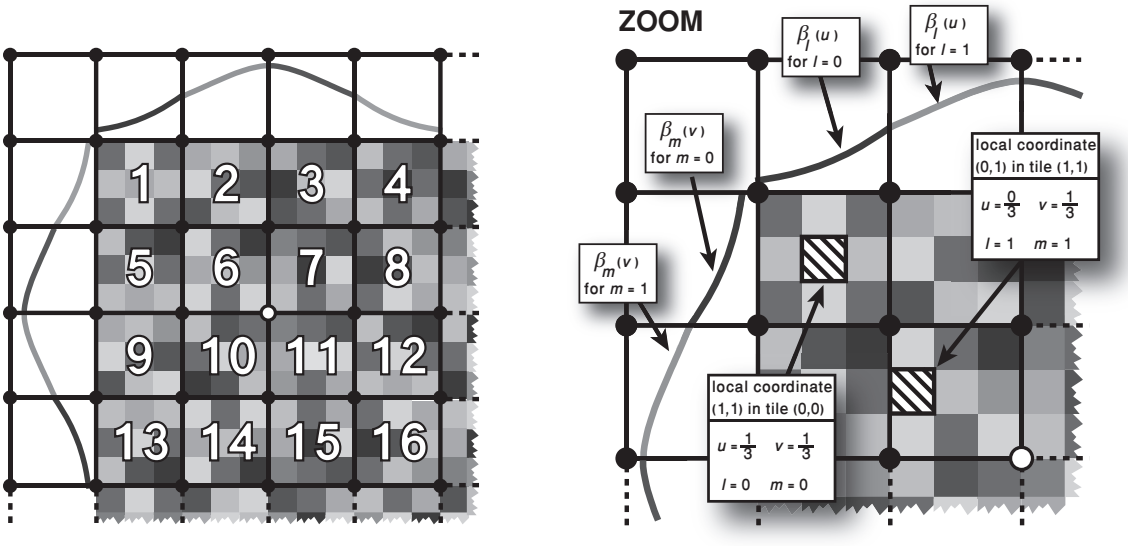


Figure 3.12: 2D example of cost function gradient computation. The process of re-expressing $\partial C/\partial \vec{\nu}$ in terms of the B-spline control point coefficients, thereby yielding the cost function gradient $\partial C/\partial P$ at each control point, is partially demonstrated for the control point highlighted in white.

coordinates of the voxel within the tile, therefore evaluating $\beta_0(1/3)$ and $\beta_0(1/3)$ in the x and y dimensions, respectively. These two results and the value of $\partial C/\partial \vec{\nu}$ at the voxel in question are multiplied together and the product is stored away for later. Once this procedure is performed at every voxel for each tile in the vicinity of the control point, all of the resulting products are accumulated, resulting in the value of the cost function gradient $\partial C/\partial \vec{P}$ at the control point.

Since the example in Fig. 3.12 uses a 2D image, 16 control points are needed to parameterize how the cost function changes at any given voxel with respect to the deformation field. Therefore, when computing the value of the cost-function gradient at a given control point, the 16 tiles affected by the control point must be included in the computation; these tiles are numbered one through sixteen in the figure. Each tile number represents a specific

combination of the B-spline basis-function pieces used to compute a tile's contribution to the gradient at the highlighted control point. For example, voxels within tile number 1 use basis functions with $l = 0$ and $m = 0$ in the x and y directions, respectively; voxels within tile 2 use basis functions with $l = 1$ and $m = 0$, and so on. Furthermore, in the 2D case, each tile of $\frac{\partial C}{\partial \nu}$ affects exactly 16 control points and is therefore, subjected to each of the 16 possible B-spline combinations exactly once. In the 3D case, each tile affects 64 control points. This is an important property that forms the basis for our parallel implementation of this algorithm.

The GPU-based algorithm that computes $\partial C/\partial \vec{P}$ operates on tiles instead of individual voxels. Here, one thread-block of 64 threads is assigned to each tile in the static image. Given a tile in which $\partial C/\partial \vec{\nu}$ values are defined at each voxel location, the 64 threads work together to parameterize these derivative values in terms of B-spline control-point coefficients, namely a set of $\partial C/\partial \vec{P}$ values. Since 64 control points are needed to parameterize a tile's contents using cubic B-splines, the thread-block will contribute to the gradient values defined at the 64 control points in the tile's immediate vicinity. In fact, each control point in the grid will receive such gradient value contributions from exactly 64 tiles (or thread blocks). The final value of the cost function gradient at a given control point is the sum of the 64 contributions received from its surrounding tiles.

Fig. 3.13 shows the multi-stage process of computing the x component of $\partial C/\partial \vec{P}$ in parallel on the GPU. This process takes as input, the starting address of the tile within the $\partial C/\partial \vec{\nu}$ array that is associated with the thread block. During stage 1, the 64 threads work in unison to fetch a cluster of 64 contiguous $\partial C/\partial \vec{\nu}$ values from global memory, which are then stored into registers. Once the cluster has been loaded, each thread computes the local

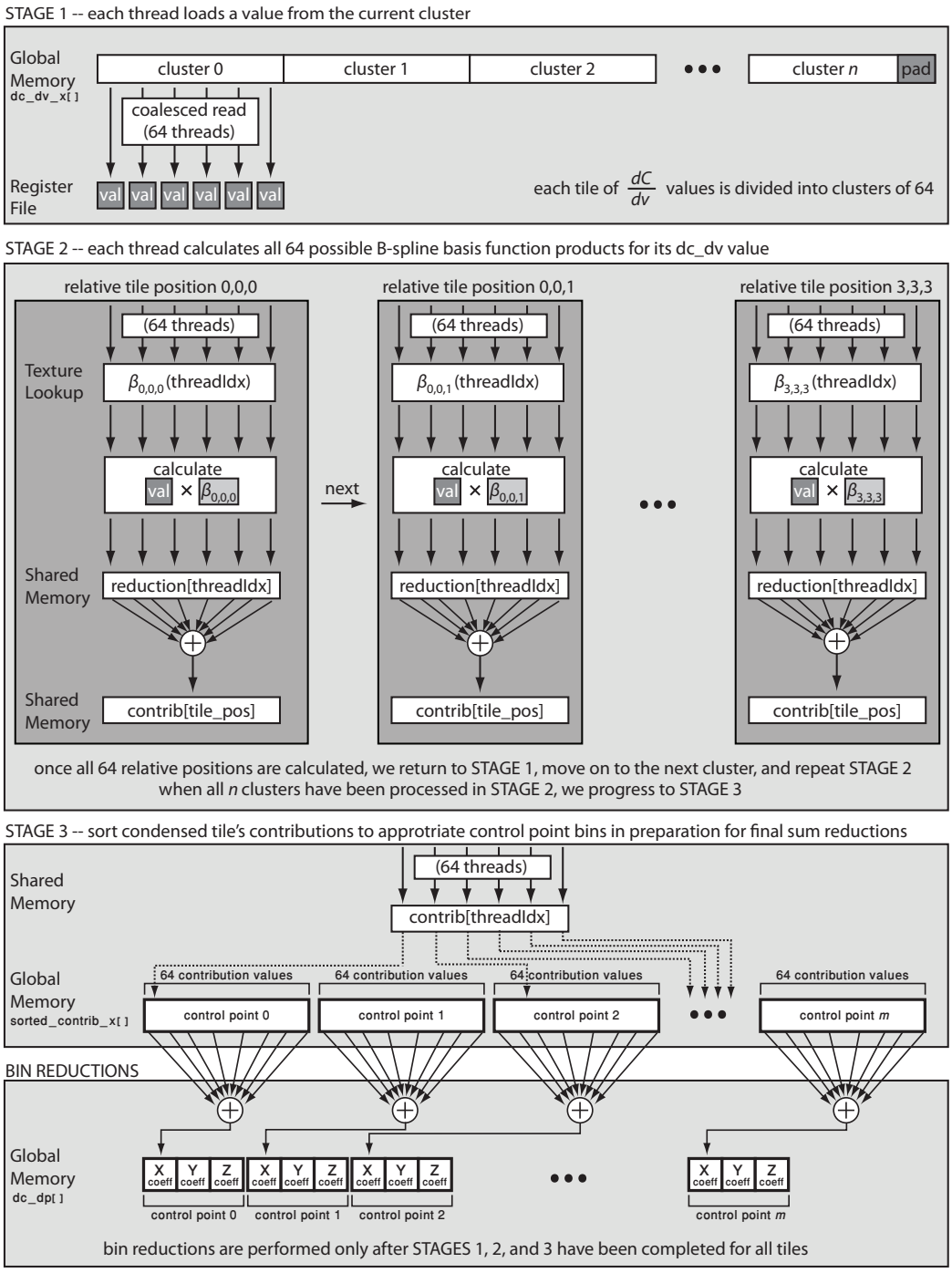


Figure 3.13: Parallel gradient computation workflow. The multi-stage process of computing the cost-function gradient $\partial C/\partial \vec{P}$ in parallel. Computation of the x-component of $\partial C/\partial \vec{P}$ is depicted. Components in the y and z directions are calculated similarly.

coordinates within the tile for the $\partial C/\partial \vec{\nu}$ value that it is responsible for. Also, as shown in Fig. 3.13, the input values are zero padded to 64, which was chosen to make the tile size a multiple of thread-block size. The padding prevents a cluster from reading into the next tile when the control-point configuration results in tiles that are not naturally a multiple of the cluster size. Stage 2 sequentially cycles through each of the 64 possible B-spline piecewise function combinations, using the local coordinates of the $\partial C/\partial \vec{\nu}$ values previously computed in Stage 1. Each of the 64 function combinations is applied to each element in the cluster in parallel; the results are stored in a temporary array located within the GPU's shared memory which is then reduced to a single value and accumulated into `sorted_contrib_x`, a region of shared memory indexed by the piecewise B-spline function combination. This stage ends once these operations have been performed for the 64 piecewise combinations. Upon completion, control returns back to stage 1, beginning another cycle by loading the next cluster of 64 values.

Once stage 2 has processed all the clusters within a tile, we will have 64 gradient contributions stored within shared memory that must be distributed to the control points they influence. Stage 3 assigns one contribution, to be distributed appropriately based on the combination number, to each of the 64 threads in the thread-block. To avoid race conditions when multiple threads belonging to different thread blocks write to the same memory location, each control point is given 64 "slots" in which to store these contributions. Once all contribution values have been distributed, each set of 64 slots is reduced to a single value, resulting in the gradient $\partial C/\partial \vec{P}$ at each control point. As shown in Fig. 3.13, the array `dc_dp` that holds the gradient is organized in an interleaved fashion as opposed to using separate arrays for each of the x, y, and z components. This provides better cache

locality when these values are read back by the optimizer, which is executed on the CPU. The time needed to copy the `dc_dp` array from the GPU to the CPU over the PCIe bus is negligible; for example, registering two $256 \times 256 \times 256$ images with a control-point spacing of $10 \times 10 \times 10$ voxels requires 73,167 B-spline coefficients to be transferred between the GPU and the CPU per iteration, which incurs a transfer overhead of 0.35 milliseconds over a PCIe 2.0 x16 bus.

3.3 Performance Evaluation

This section presents experimental results obtained for the CPU and GPU implementations in terms of both execution speed and registration quality. We compare the performance achieved by the following different implementations:

- *Single-threaded CPU implementation.* This reference implementation serves as a baseline for comparing the performance of the multi-core CPU and GPU implementations. It is highly optimized, uses the SSE instruction set, and characterizing its performance using Valgrind, a profiling system for Linux programs [58], indicates a very low miss rate of about 0.1% in both the L1 and L2 data caches.

- *Multi-core implementation on the CPU using OpenMP.* This implementation uses OpenMP, a portable programming interface for shared-memory parallel computers [59], to parallelize the steps involving histogram generation, cost-function evaluation, and gradient computation.

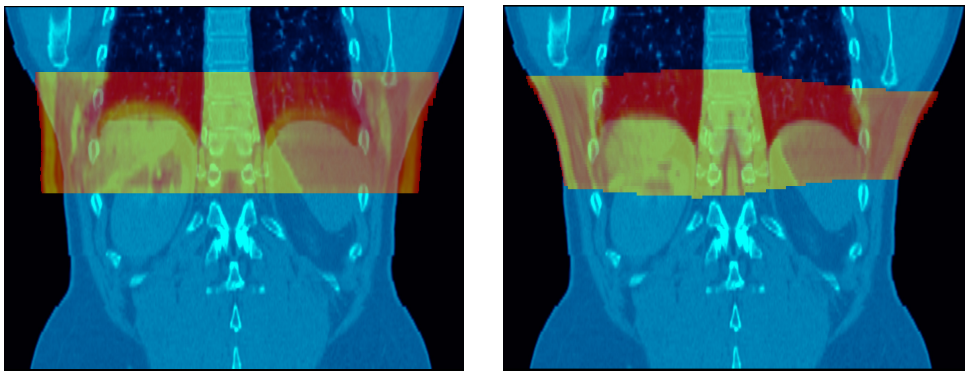
- *GPU-based implementation.* This implementation uses the compute unified device architecture or CUDA programming interface to perform both histogram generation

and gradient computation on the GPU. The cost-function evaluation and the gradient descent optimization is performed on the CPU.

In addition to the registration quality, we quantify the impact of the volume size and control-point spacing on the execution times incurred by each of the above implementations. The tests reported here use a machine equipped with an Intel quad-core i7 920 processor with each core clocked at 2.6GHz, and 12GB of RAM. The GPUs used are: the NVidia Tesla C1060 model with 240 cores, each clocked at 1.5GHz, and 4GB of onboard memory; and the Tesla C2050 model with 448 cores, each clocked at 1.1GHz, and 2.6GB of onboard memory.

3.3.1 Registration Quality

Fig. 3.14 shows the registration results obtained by the GPU-based implementation, given two multi-modality images of a patient's thorax: a $512 \times 384 \times 16$ MR volume (shown in red) and a $512 \times 512 \times 115$ CT volume (shown in blue). Fig. 3.14(a) shows the two images superimposed on each other prior to registration. The images were registered using the CT volume as the static image and the MR volume as the moving image with a control-point grid spacing of $100 \times 100 \times 100$ voxels. When computing the mutual information, the static- and moving-image histograms were each constructed using 32 equally-wide bins, and the joint histogram was constructed using 1024 (or 32^2 bins). The registration process was allowed to evolve over 20 iterations and the resulting deformation field was used to warp the MR image. Fig. 3.14(b) shows this warped image (shown in red) superimposed on the CT volume (shown in blue).



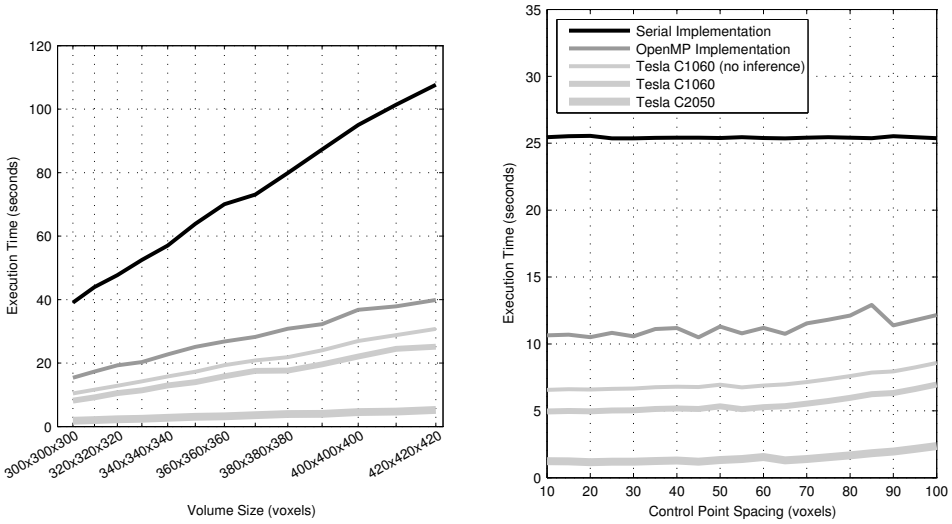
(a) (b)

Figure 3.14: Thoracic MRI to CT registration using mutual information. (a) A $512 \times 384 \times 16$ MRI volume (shown in red) is superimposed on a $512 \times 512 \times 115$ CT volume (shown in blue) prior to deformable registration. (b) The same MRI and CT volumes superimposed on each other after 20 iterations of the deformable registration process on the GPU. The control-point grid spacing was set to 100^3 voxels.

3.3.2 Sensitivity to Volume Size

This series of tests characterizes each implementation's sensitivity, in terms of execution time, to increasing volume size where the volumes are synthetically generated. We fix the control-point spacing at 15 voxels in each physical dimension and increase the volume size in steps of $10 \times 10 \times 10$ voxels. For each volume size, we record the execution time incurred by a single iteration of the registration process. Fig. 3.15(a) summarizes the results. As expected, the execution time increases linearly with the number of voxels involved. The OpenMP version offers slightly better performance with respect to the reference implementation, with a speedup of 2.5 times. We attribute this result to the high level of serialization, imposed by the locking mechanisms, during histogram construction.

The GPU achieves a speedup of 21 times with respect to the reference implementation and 8.5 times with respect to the OpenMP implementation. Also, the relatively simple



(a) (b)

Figure 3.15: Mutual information registration performance. (a) The impact of volume size on the execution times incurred by the single-threaded and multi-threaded CPU implementations and the GPU implementations using the Tesla C1060 and C2050 models. (b) The impact of control-point spacing on the execution times incurred by the single-threaded and multi-threaded CPU implementations and the GPU implementations using the Tesla C1060 and C2050 models. The volume size is fixed at $260 \times 260 \times 260$ voxels.

optimization step of inferring the value of the histogram bin expected to incur the most write conflict (in other words, the most number of atomic-exchange operations) significantly improves performance on the Tesla 1060C GPU; for actual CT data, this optimization step speeds up the histogram generation phase by five times. Finally, though not supported on the Tesla C1060, the `atomicAdd` instruction available on newer models such as the C2050 allows bins to be incremented in a thread-safe manner without having to performing the complex atomic-exchange logic shown in lines 15 through 25 of the histogram generation algorithm listed in Fig. 3.10. Therefore, the Tesla C2050 spends only 60% of its processing time generating histograms when compared to the C1060 which spends about 70% of its time, and is therefore, much faster.

3.3.3 Sensitivity to Control-Point Spacing

As discussed in Section 3.2.4, the parallelized gradient computation exploits key attributes of the uniform control-point spacing scheme to achieve fast execution times. We consider each tile in the volume as a work unit where each unit is farmed out to individual cores that are single threaded in the case of a CPU core or multi-threaded in the case of the GPU. If the volume size is fixed, then increasing the control-point spacing results in fewer, yet larger, work units. Fig. 3.15(b) shows the impact of varying the control-point spacing in increments of $5 \times 5 \times 5$ voxels with the volume size fixed at $260 \times 260 \times 260$ voxels. Note that the execution times are relatively unaffected by the control-point spacing for all implementations.

The GPU-based versions show a slight sub-linear increase in execution time starting at a control-point spacing of approximately 65 voxels. For larger spacings, the work-unit size becomes adequately large such that the processing time dominates the time required to swap work units in and out. When the time needed to process a work unit is significantly less than the overhead associated with swapping it in and out of a GPU core, the execution time is essentially constant since the overhead incurred by the swapping is constant. When the processing time begins to dominate, we expect the execution time to increase as the number of elements within the work units increase.

3.4 Conclusions

We have developed a B-spline based deformable registration process for aligning multi-modality images, suitable for use on multi-core processors and GPUs. Using mutual information as the similarity metric, the goal is to obtain a deformation field that warps the

moving image such that it is most similar to the static image. We developed and implemented parallel algorithms to realize the following major steps of the registration process: generating a deformation field using the B-spline control-point grid, calculating the image histograms needed to compute the mutual information, and calculating the change in the mutual information with respect to the B-spline coefficients for the gradient-descent optimizer. We have evaluated the multi-core CPU and GPU implementations in terms of both execution speed and registration quality. Our results indicate that the speedup varies with volume size and the voxel-intensity distribution within the images, but is relatively insensitive to the control-point spacing. Our GPU-based implementations achieve, on average, a speedup of 21 times with respect to the reference implementation and 7.5 times with respect to a multi-core CPU implementation using four cores, with near-identical registration quality. We hope that such improvements in processing speed will mean that deformable registration methods can be routinely in interventional procedures such as image-guided surgery and image-guided radiotherapy.

CHAPTER 4: IMPROVING MI WITH VARIANCE OPTIMAL HISTOGRAMS

This chapter builds upon the preceding chapter covering multi-modal deformable image registration. Here, we develop a fast method of constructing histograms that better estimate the marginal and joint image probability density functions. Because the mutual information similarity metric is directly derived from these distributions, it is vital that they accurately measure the information content of the images undergoing registration. We propose that the optimal histogram configuration contains one tissue density per histogram bin – a configuration that we show may obtained by employing variance optimal histograms. Specifically, this chapter develops a method for approximating such histograms so that they may be computed within the time constrains imposed by standard clinical work flow. Image tissue distribution within the variance optimal histogram binning scheme is demonstrated for a thoracic CT scan. Furthermore, we demonstrate with PET to CT and MRI to CT cases that registration accuracy may be greatly improved while using fewer bins than previously required by the equally spaced histograms presented in the preceding chapter – thereby enabling superior performance while imposing less restrictive memory requirements.

4.1 Overview of Variance Optimal Histograms

Multi-modal image registration aligns two images obtained via differing imaging modalities into a common coordinate system. Two common modalities that often undergo multi-modal registration are magnetic resonance imaging (MRI) and computed tomography (CT) imaging. The high-contrast resolution of an MRI image helps distinguish between various soft tissues that otherwise appear similar in conventional CT imaging. CT images, on the other hand, have high spatial resolution and are more readily obtained in intra-operative settings due to the availability of mobile CT scanner technology. Consequently, it is not uncommon to obtain and register an intra-operative CT image to a pre-operative MRI to assist in malignant tissue localization and its subsequent resection. In such cases, the spatial correlation provided by the registration allows, for example, cancerous tissue highly visible in the pre-operative MRI to be accurately superimposed upon the intra-operative CT where the cancerous tissue is otherwise indistinguishable from surrounding healthy tissue.

Since MRI and CT images are obtained via fundamentally different imaging methods, with each revealing features invisible to the other, obtaining an accurate spatial correlation through registration requires the use of statistical methods rooted in information theory. Therefore, multi-modal registration is often implemented using mutual information (MI) as the similarity metric that determines the correctness of a given spatial mapping [55]. Modern registration algorithms ultimately obtain accurate mappings by iteratively modifying a set of warping parameters that are then applied to one of the two images involved in the registration process. This warped image, referred to the *moving image*, is then compared to the other image involved in the process, referred to as the *static image*, using MI as the

similarity metric. If the metric indicates that the images possess a good spatial correlation, then the warping parameters describing the correlation are provided to the physician. If the metric indicates a suboptimal correlation, the registration algorithm may use suitable optimization techniques to intelligently search for the set of warping parameters that maximizes the metric.

The MI metric measures the information content shared between the static and moving images, and the most straightforward way to compute it is to use the marginal histograms of the static and moving image intensities, h_S and h_M, respectively, and the joint histogram h_j as

$$MI = \frac{1}{N} \sum_{j=0}^{K_S} \sum_{i=0}^{K_M} h_j(i,j) \ln \frac{N \times h_j(i,j)}{h_S(j) \times h_M(i)} \tag{4.1}$$

where K_S and K_M denote the number of bins in the static and moving image histograms, h_S and h_M, respectively, and N is the number of voxels in S with corresponding voxels in M after application of the warping parameters. The use of histograms to estimate the probability distributions of the image intensities has been adopted in numerous previous studies [60] due to the relative ease of computing these histograms. The registration algorithm's goal then is to determine which bins in the static image histogram correlate to which bins in the moving image's histogram. Ideally (4.1) will be maximized when each bin in the moving-image histogram has only one corresponding bin in the static-image histogram. In other words, if the pixel intensities within a given bin of h_S represent fatty tissue, (4.1) will benefit the most when the warping maps each of those pixels in S to pixels in M that also represent fatty tissue, which are similarly lumped together within a single bin within h_M.

As the foregoing discussion suggests, the width of the histogram bins is an important parameter affecting both the speed and quality of the registration process. Unfortunately, the current state-of-the-art requires the operator to manually choose bin widths (specific to the underlying data set) to achieve good registration [61, 62]. Alternatively, equally sized bins may be used to construct the histogram. However, this simple binning strategy incurs the risk of spreading a single tissue type over several bins or, even worse, grouping multiple tissues together within a single bin, resulting in poor results. In this paper, we develop a fast *variance-optimal* or *V-opt* histogram binning technique suitable for high resolution medical images that aims to automatically place each distinct tissue type within exactly one histogram bin.

4.2 Theory of Operation and Implementation

To delegate bins in a way that segregates different tissue types, we define bin boundaries based on the variance of the pixel intensities falling within the bins. If a single bin were to contain only pixels for one type of tissue, the variance for that bin would be minimal. This is due to the tissues possessing a fairly uniform density throughout, thus resulting in an equally uniform pixel intensity distribution within the bin. So, when considering all tissues within an image together, the sum of the bin variances for the entire histogram should be minimal when optimally segregated. The histograms generated using this binning scheme are termed variance-optimal or V-opt histograms [63].

Generating a V-opt histogram wherein N pixels are partitioned into B bins incurs $O(N^2B)$ computation complexity, thus making the direct application of this binning scheme to most all medical images prohibitive. For example, it is not uncommon for a 3D CT scan to

```
1: /* Initialize lookup tables */
2: [s, ssq, cnt] = init_vopt(k, delta, offset, min_val, max_val)
3:
4: /* Compute one-bin scores */
5: for i = 0 to K step 1 do
6:   err[0][i] = bin_error(0, i, s, ssq, cnt)
7: end for
8:
9: /* Compute best multi-bin scores */
10: for j = 0 to B step 1 do
11:   for i = 0 to K step 1 do
12:     err[j][i] = ∞
13:     tracker[j][i] = 0
14:     for k = 0 to i step 1 do
15:       candidate = err[j-1][k]
16:           + bin_error(k+1, i, s, ssq, cnt)
17:       if candidate <= err[j][i] then
18:           err[j][i] = candidate
19:           tracker[j][i] = k
20:       endif
21:     end for
22:   end for
23: end for
24:
25: keys = make_key_table(tracker)
```

Figure 4.1: The V-opt histogram generation technique.

consist of $512 \times 512 \times 128$ voxels, that is $N = 33.6$ million voxels. Though the original data set can be sub-sampled, even sub-sampling by a factor of four—which is quite coarse—results in a $128 \times 128 \times 32$ volume where N would still be prohibitive at roughly half a million voxels. To improve computational efficiency, we generate a temporary histogram in the initialization stage consisting of a large number of uniformly spaced bins K. We then merge these bins such that the resulting histogram is an approximation of a true V-opt histogram. This operation incurs $O(K^2B)$ computation complexity where $K \ll N$, and yields good results with values as low as $K = 1000$.

Fig. 4.1 details the proposed V-opt histogram generation technique. Here, `tmp_hist[ ]` is the temporary histogram comprising K equally-sized bins, which will be selectively

```
1: function init_vopt (k, delta, offset, min_val, max_val)
2:   /* Temp histogram with bin averages */
3:   delta = (max_val − min_val)/(K − 1)
4:   offset = min_val − 0.5 × delta
5:   for i = 0 to npix step 1 do
6:     bin = ⌊(img[i] − offset)/delta⌋
7:     tmp_hist[bin] += 1
8:     tmp_sum[bin] += img[i] - offset
9:   end for
10:
11:  for i = 0 to K step 1 do
12:    if tmp_hist[i] > 0 then
13:      tmp_avg[i] = tmp_avg[i] / tmp_hist[i]
14:    else
15:      tmp_avg[i] = tmp_avg[i-1]
16:    end if
17:  end for
18:
19:  /* Generate lookup tables */
20:  s[0] = tmp_avg[0]
21:  ssq[0] = tmp_avg[0]^2
22:  cnt[0] = tmp_hist[0]
23:  for i = 1 to K step 1 do
24:    s[i] = s[i-1] + tmp_avg[i]
25:    ssq[i] = ssq[i-1] + tmp_avg[i]^2
26:    cnt[i] = cnt[i-1] + tmp_hist[i]
27:  end for
28:  return [s, ssq, cnt]
29:
30: end function
```

Figure 4.2: Computation of V-opt lookup tables

```
1: function bin_error (start, end, s, ssq, cnt)
2:   diff = s[end] − s[start]
3:   sq_diff = ssq[end] − ssq[start]
4:   Δ = end − start + 1
5:   n = cnt[end] − cnt[start]
6:   v = sq_diff − diff^2/Δ
7:
8:   /* Less than one voxel in bin */
9:   if (n < 1) then
10:    /* Penalize this solution */
11:    return ∞
12:  end if
13:  return v
14: end function
```

Figure 4.3: Computation of the bin error.

merged to obtain the final approximation of the V-opt histogram comprising $B \ll K$ bins. Similarly, `tmp_avg[ ]` is comprised of K values and contains the average values of the voxels within each bin of `tmp_hist[ ]`. Lines 1–9 generate lookup tables used to accelerate the computation

$$V_{sum}(i,j) = (SSQ(j) - SSQ(i)) - \frac{(S(j) - S(i))^2}{j - i + 1} \tag{4.2}$$

which gives the sum of the variance for elements i through j in `tmp_avg[ ]` where $0 < i < j < K$. Here $S(i)$ is the sum of all bin averages up to and including bin i. Similarly, $SSQ(i)$ is the sum of all squared bin averages up to and including i. The `s[ ]` and `ssq[ ]` arrays provide precomputed tables for these summation series. The `cnt[ ]` array is generated to provide fast computation of the number of voxels falling within the range $[i, j]$, that is `cnt[j]` − `cnt[i]`. Eq. (4.2) is implemented in lines 2–6 of the listing shown in Fig. 4.3; lines 9–12 prevent empty bins in the V-opt histogram.

The remainder of the listing shown in Fig. 4.1 uses a dynamic programming (DP) method to determine the optimal bin boundaries for a V-opt histogram comprising B bins. Broadly speaking, if $B = 32$, then 32 ranges within $[0, K]$ would be determined using (4.2) such that the sum of their variances is minimized. These 32 ranges then become the bin boundaries for the V-opt histogram. The DP algorithm uses the key property that adding an additional bin to the current V-opt histogram always decreases the histogram's overall variance. The procedure detailed in lines 11–16 can be summarized as follows: Consider the j^{th} DP step charged with adding one more bin to the V-opt histogram that currently has $j - 1$ bins. The previous $j - 1$ DP steps have computed optimal values for `err[j-1][i]`, $1 \leq i \leq K$.

During the j^{th} step, we use the variable i, ranging from 1 to K, and for each i, compute the variance when i elements are partitioned over $j-1$ bins (for which the optimal value is already known) and the remaining $i-K$ elements are placed in the j^{th} bin. We then select the partitioning choice that minimizes the variance over j bins. This process is repeated until all B bins are added to the V-opt histogram.

4.3 Results

Validation experiments were performed using both V-opt histograms constructed with $B = 32$ and $K = 1000$ bins and equally spaced histograms comprising 32 bins. All registrations are deformable and use MI as the similarity metric with the warping parameters expressed using uniform cubic B-spline basis coefficients.

Fig. 4.4 visualizes results from experiments aimed at determining the efficacy of the V-opt binning scheme in terms of automatically placing each distinct tissue type within exactly one histogram bin (or the fewest number of bins). Fig. 4.4(a) shows the thoracic CT volume used as the input data and Fig. 4.4(b-f) each show only those image voxels falling within a single bin, thereby visualizing how different tissue types are partitioned across 5 of the 32 bins. Bins not shown mostly contain voxels falling within the lung due to the organ's high range of contrast in the CT image. The time required to compute the optimal bin ranges was 659 ms.

Accuracy improvements when registering PET to CT images as well as MRI to CT images are also validated quantitatively. Fig. 4.5 shows the results for two PET to CT image registrations in which the static image shown in Fig. 4.5(a) is registered to the moving image shown in Fig. 4.5(b). The two registrations differ only by the type of histogram em-

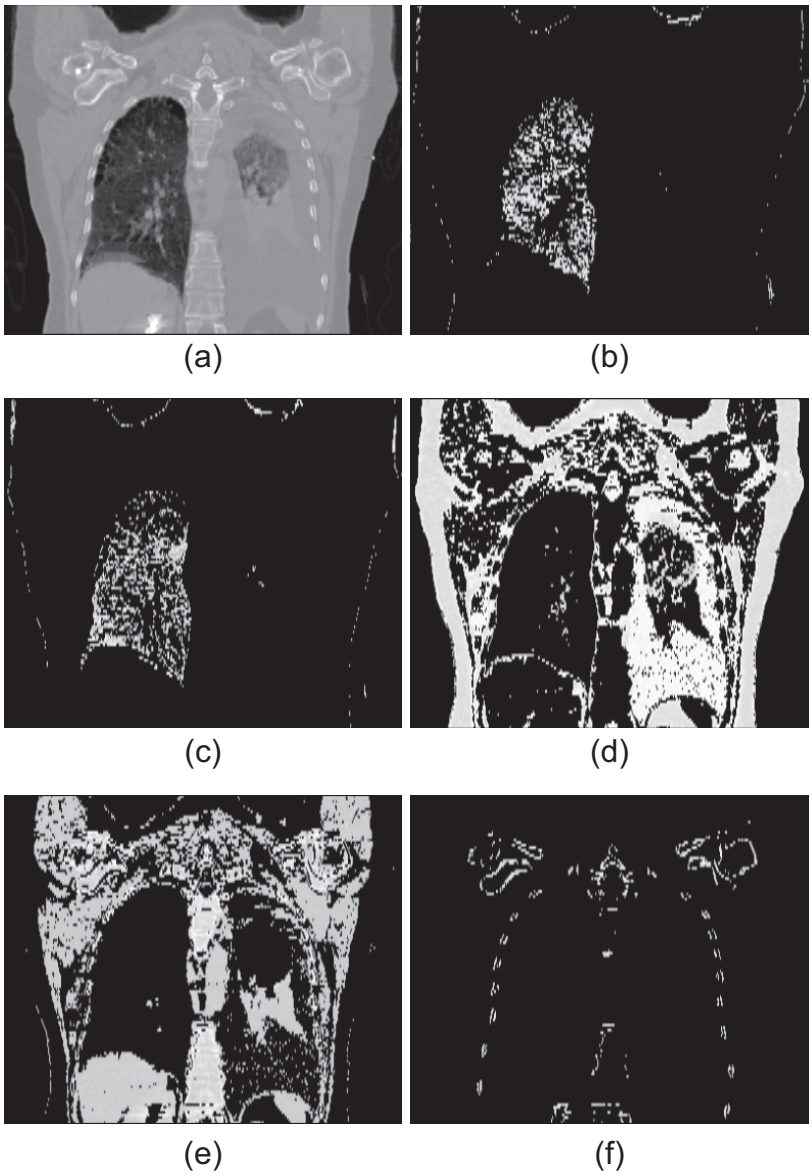


Figure 4.4: Example of variance optimal tissue division by bin. Panel (a) shows a CT volume of dimension $512 \times 512 \times 128$ voxels. Panels (b)–(f) each show only the voxels falling with a single given bin in the V-opt histogram generated using 32 bins.

ployed to compute the mutual information. Fig. 4.5(c) shows the resulting warped moving image superimposed upon the static image when using equally spaced 32-bin histograms. Fig. 4.5(d) shows the same superimposition but using V-opt histograms. The registration performed using V-opt histograms displays superior registration for an equal number of

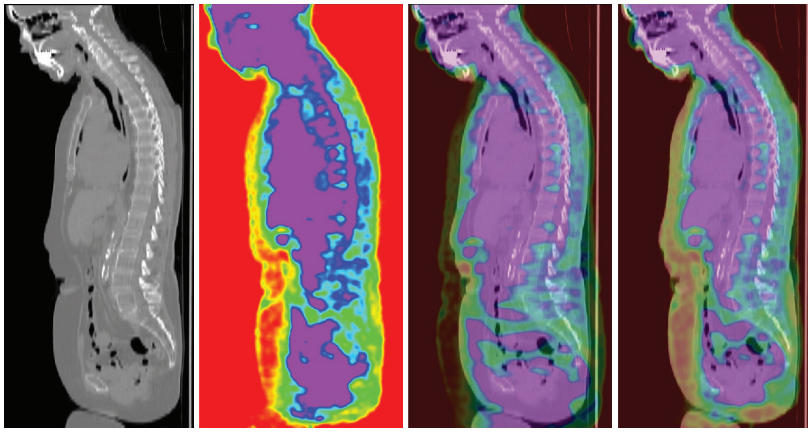


Figure 4.5: PET to CT registration using variance optimal histograms. Results of a CT to PET registration. The CT image shown in (a) is the static image and the PET image in (b) is the moving image. The post-registration PET image superimposed upon the CT image is shown in (c) when the registration is performed using histograms with 32 uniformly spaced bins. The same post-registration superimposition is shown in (d) when using V-opt histograms with 32 bins.

bins, particularly in the anterior regions of the scan.

Finally, to quantitatively assess the benefits of V-opt histograms, the liver was manually segmented in both a pre-operative MRI and an intra-operative CT image. The images were then registered using both equally spaced and V-opt histograms. Subsequently, the obtained warping parameters were used to warp the MRI liver segmentation, which was then compared to the CT liver segmentation. The equally spaced histogram resulted in the two registered livers having a Hausdorff distance of 65.805 mm whereas the V-opt histogram produced a Hausdorff distance of 43.278 mm. (The resulting vizualization is not shown in the paper due to space constraints.)

4.4 Conclusions

We have developed a fast method of estimating the probability distribution of intensities within medical images using variance-optimal histograms. The goal is to improve the quality of MI-based image registration by automatically placing each distinct tissue type in the image in exactly one bin within the intensity histogram. Experimental results indicate that for the same number of bins, V-opt histograms achieve better accuracy for both MRI to CT and PET to CT registrations when compared to histograms that use uniform bin spacing.

CHAPTER 5: ANALYTIC VECTOR FIELD REGULARIZATION

This chapter develops an analytic method for constraining the vector field evolution that seamlessly integrates into both uni-modal and multi-modal B-spline based registration algorithms. Because image registration is an ill-posed problem, multiple vector field solutions may equally satisfy the criteria imposed by either the employed similarity metric. Consequently, the registration solution produced may describe a physically impossible deformation. By imposing explicit constrains on the character of the vector field, it is possible to guide the registration process towards producing physically meaningful solutions; thereby regularizing the ill-posed problem. This chapter provides the analytic mathematical formalism required to impose second order smoothness upon the deformation vector field in a faster and more efficient fashion than traditional numerically based central differencing methods. Furthermore, it is shown that the analytically derived matrix operators may be applied directly to the B-spline parameterization of the vector field to achieve the desired physically meaningful solutions. Single and multi-core CPU implementations are developed and discussed. Performance for both implementations is investigated with respect to the traditional central differencing based numerical method, and the quality of the analytic implementations is investigated via a thoracic MRI to CT case study.

5.1 Theory and Mathematical Formalism

When it is desired to register one image to another in a deformable fashion, many attempts at incrementally improving the individual voxel-to-voxel mapping are algorithmically attempted. The "goodness" of a particular mapping, described by a vector deformation field, is determined by a similarity metric: the mean squared error (MSE) for unimodal registrations, and the mutual information (MI) for multi-modal registrations. Although these two metrics measure different features, they both resolve to their optimal values when given a deformation field that yields a perfect registration. However, it is also possible to provide alternative deformation field configurations that, when scored by the similarity metric, appear to provide a desirable registration despite actually providing worse or even anatomically impossible voxel mappings. Consequently, image registration is an ill-posed problem in the Hadamardian sense due to its lack of a unique solution. This chapter describes how the solution space may be pruned so as to consist only of physically meaningful deformation fields. In other words, the problem will be "regularized" by the imposition of physical constrains on the solution space.

We will now develop the mathematic formalism required to perform regularization analytically on 3D medical images in a fashion that is compatible and easy coupled with the unimodal and multi-modal registration techniques presented in earlier chapters.

The problem of registration may be regularized by implementing a penalty score that increases with the second derivative of the vector field:

$$
\begin{aligned}
S(\vec{\nu}) = \int\int\int & \left(\frac{\partial^2 \nu_x}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial z^2}\right)^2 \\
& + \left(\frac{\partial^2 \nu_y}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial z^2}\right)^2 \\
& + \left(\frac{\partial^2 \nu_z}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial z^2}\right)^2 \\
& + \left(\frac{\partial^2 \nu_x}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial y \partial z}\right)^2 \\
& + \left(\frac{\partial^2 \nu_y}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial y \partial z}\right)^2 \\
& + \left(\frac{\partial^2 \nu_z}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial y \partial z}\right)^2 dxdydz
\end{aligned}
\tag{5.1}
$$

where the integration is performed over the entire deformation field. This penalty term, which quantifies the bending energy or smoothness of the deformation field, is a weighted addition to the similarity metric C:

$$C_{total} = C + \lambda S. \tag{5.2}$$

which requires similar modification of the cost function gradient to reflect the influence of the regularization:

$$\frac{\partial C_{total}}{\partial P} = \frac{\partial C}{\partial P} + \lambda \frac{\partial S}{\partial P}. \tag{5.3}$$

Consequently, deformation field configurations prescribing non-linear changes in patient anatomy will be avoided by the optimizer due to possessing higher costs. The degree to which such solutions are avoided is dictated by tuning the parameter λ. Specific values for λ are application specific and depend on both image modality and the particular anatomical structures undergoing registration.

Traditionally, (5.1) would be implemented numerically – a slow process requiring second-order central differencing to be performed for each vector in the deformation field. However, for B-spline based registration algorithms, the deformation field is parameterized in terms of a sparse set of B-spline control-points. Since the deformation field $\vec{\nu}$ can be computed at any given point within a tile by means of the B-spline interpolation:

$$\nu_x(\vec{x}) = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3}\beta_i(u)\beta_j(v)\beta_k(w)P_{p_x,l,m,n}. \tag{5.4}$$

it is possible to express the squared derivative terms in (5.1) analytically. Not only does this forego the need to compute these derivatives via partial differencing, but it also allows for analytic integration. Additionally, since an integration over a large interval can be expressed as a sum of several integrations performed over smaller sub-intervals, the smoothness metric for the entire deformation field may be computed by simply computing and summing the smoothness metric for each individual tile defined by the B-spline control-grid:

$$S(\vec{\nu}) = \sum_{all\ \ tiles} S_{tile} \tag{5.5}$$

where S_{tile} is defined thusly given a control-point spacing of r_x, r_y, and r_z in the x, y, and z dimensions, respectively:

$$
\begin{aligned}
S_{tile} = \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} &\left(\frac{\partial^2 \nu_x}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial z^2}\right)^2 \\
&+\left(\frac{\partial^2 \nu_y}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial z^2}\right)^2 \\
&+\left(\frac{\partial^2 \nu_z}{\partial x^2}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial y^2}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial z^2}\right)^2 \\
&+\left(\frac{\partial^2 \nu_x}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_x}{\partial y \partial z}\right)^2 \\
&+\left(\frac{\partial^2 \nu_y}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_y}{\partial y \partial z}\right)^2 \\
&+\left(\frac{\partial^2 \nu_z}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 \nu_z}{\partial y \partial z}\right)^2 dxdydz.
\end{aligned} \tag{5.6}
$$

Furthermore, since the control grid is formed by uniformly spaced control-points, all tiles defined by the control-grid are of equal dimensions, which allows for the formulation of six standard matrix operators that can be rapidly applied to each tile's set of 64 control points in order to obtain each tile's smoothness. Specifically, these six operators are:

$$
\begin{aligned}
\mathbf{V_1} &\equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial x^2}\right)^2 dxdydz, \quad \mathbf{V_2} \equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial y^2}\right)^2 dxdydz, \\
\mathbf{V_3} &\equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial z^2}\right)^2 dxdydz, \quad \mathbf{V_4} \equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial x \partial y}\right)^2 dxdydz, \\
\mathbf{V_5} &\equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial x \partial z}\right)^2 dxdydz, \quad \mathbf{V_6} \equiv \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \left(\frac{\partial^2}{\partial y \partial z}\right)^2 dxdydz.
\end{aligned} \tag{5.7}
$$

It is possible to pre-compute matrices representing these six operators. This results in six matrix operations being performed on each tile's set of 64 control point coefficients $\vec{p}$ for each iteration of the registration process.

To this end, let us restate (5.4) in terms of matrices. It is convenient to first construct a matrix $\mathbf{Q}$ containing the B-spline basis function coefficients normalized to the grid dimensions. This is necessary because the B-spline basis may only be evaluated within the range [0,1]:

$$\mathbf{Q}_x = \mathbf{B}\mathbf{R}_x \quad \mathbf{Q}_y = \mathbf{B}\mathbf{R}_y \quad \mathbf{Q}_z = \mathbf{B}\mathbf{R}_z \tag{5.8}$$

The matrix $\mathbf{B}$ contains the coefficients of the cubic uniform B-spline basis function:

$$\mathbf{B} = \frac{1}{6}\begin{bmatrix} 1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.9}$$

and

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{r_x} & 0 & 0 \\ 0 & 0 & \frac{1}{r_x^2} & 0 \\ 0 & 0 & 0 & \frac{1}{r_x^3} \end{bmatrix}, \quad \mathbf{R}_y = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{r_y} & 0 & 0 \\ 0 & 0 & \frac{1}{r_y^2} & 0 \\ 0 & 0 & 0 & \frac{1}{r_y^3} \end{bmatrix}, \quad \mathbf{R}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{1}{r_z} & 0 & 0 \\ 0 & 0 & \frac{1}{r_z^2} & 0 \\ 0 & 0 & 0 & \frac{1}{r_z^3} \end{bmatrix}, \tag{5.10}$$

where r_x, r_y, and r_z are the B-spline control point spacings in millimeters for the x, y, and z dimensions, respectively. This allows us to restate (3.1) as:

$$\nu_x = \sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x(i,a)\vec{x}(a)\right)\left(\sum_b \mathbf{Q}_y(j,b)\vec{y}(b)\right)\left(\sum_c \mathbf{Q}_z(k,c)\vec{z}(c)\right) \tag{5.11}$$

where $\mathbf{Q}(0,0)$ refers to the first element of the first row of the $\mathbf{Q}$ matrix, $\mathbf{Q}(1,1)$ the second element of the second row, and so on. Additionally, $\vec{x}$, $\vec{y}$, and $\vec{z}$ are defined thusly to form the traditional Cartesian basis in the three orthogonal dimensions:

$$\vec{x} = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} 1 \\ y \\ y^2 \\ y^3 \end{bmatrix}, \quad \vec{z} = \begin{bmatrix} 1 \\ z \\ z^2 \\ z^3 \end{bmatrix}. \tag{5.12}$$

We can now construct matrices for the six operators listed in (5.7) by taking the second order derivatives of (5.11). To do this, we first redefine $\mathbf{Q}$ to include an additional matrix

which serves to take the first ($\delta = 1$) and second ($\delta = 2$) order derivatives of the B-spline basis function:

$$\mathbf{Q}_x^{(\delta)} = \mathbf{B}\mathbf{R}_x\boldsymbol{\Delta}^{(\delta)} \quad \mathbf{Q}_y^{(\delta)} = \mathbf{B}\mathbf{R}_y\boldsymbol{\Delta}^{(\delta)} \quad \mathbf{Q}_z^{(\delta)} = \mathbf{B}\mathbf{R}_z\boldsymbol{\Delta}^{(\delta)} \tag{5.13}$$

where $\boldsymbol{\Delta}^{(\delta)}$ is defined thusly for $\delta \in [0,2]$ as

$$\boldsymbol{\Delta}^{(0)} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\Delta}^{(1)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}, \quad \boldsymbol{\Delta}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}. \tag{5.14}$$

The second derivatives are now easily constructed:

$$\begin{aligned}
\frac{\partial^2 \nu_x}{\partial x^2} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(2)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(0)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(0)}(k,c)\vec{z}(c) \right) \\
\frac{\partial^2 \nu_x}{\partial y^2} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(0)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(2)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(0)}(k,c)\vec{z}(c) \right) \\
\frac{\partial^2 \nu_x}{\partial z^2} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(0)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(0)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(2)}(k,c)\vec{z}(c) \right) \\
\frac{\partial^2 \nu_x}{\partial x \partial y} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(1)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(1)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(0)}(k,c)\vec{z}(c) \right) \\
\frac{\partial^2 \nu_x}{\partial x \partial z} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(1)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(0)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(1)}(k,c)\vec{z}(c) \right) \\
\frac{\partial^2 \nu_x}{\partial y \partial z} &= \sum_{i,j,k} \vec{p}_{i,j,k} \left(\sum_a \mathbf{Q}_x^{(0)}(i,a)\vec{x}(a) \right) \left(\sum_b \mathbf{Q}_y^{(1)}(j,b)\vec{y}(b) \right) \left(\sum_c \mathbf{Q}_z^{(1)}(k,c)\vec{z}(c) \right).
\end{aligned} \tag{5.15}$$

To construct the squares of the derivatives, we first simplify notation in a way that eliminates the summation over indices i, j, and k:

$$\vec{\gamma}^{(\delta_x,\delta_y,\delta_z)} = \begin{bmatrix} \left(\sum_a \mathbf{Q}_x^{(\delta_x)}(0,a)\vec{x}(a)\right)\left(\sum_b \mathbf{Q}_y^{(\delta_y)}(0,b)\vec{y}(b)\right)\left(\sum_c \mathbf{Q}_z^{(\delta_z)}(0,c)\vec{z}(c)\right) \\ \left(\sum_a \mathbf{Q}_x^{(\delta_x)}(1,a)\vec{x}(a)\right)\left(\sum_b \mathbf{Q}_y^{(\delta_y)}(0,b)\vec{y}(b)\right)\left(\sum_c \mathbf{Q}_z^{(\delta_z)}(0,c)\vec{z}(c)\right) \\ \left(\sum_a \mathbf{Q}_x^{(\delta_x)}(1,a)\vec{x}(a)\right)\left(\sum_b \mathbf{Q}_y^{(\delta_y)}(1,b)\vec{y}(b)\right)\left(\sum_c \mathbf{Q}_z^{(\delta_z)}(0,c)\vec{z}(c)\right) \\ \vdots \\ \left(\sum_a \mathbf{Q}_x^{(\delta_x)}(3,a)\vec{x}(a)\right)\left(\sum_b \mathbf{Q}_y^{(\delta_y)}(3,b)\vec{y}(b)\right)\left(\sum_c \mathbf{Q}_z^{(\delta_z)}(3,c)\vec{z}(c)\right) \end{bmatrix} = \begin{bmatrix} \gamma_{0,0,0} \\ \gamma_{1,0,0} \\ \gamma_{1,1,0} \\ \\ \vdots \\ \gamma_{3,3,3} \end{bmatrix}^{(\delta_x,\delta_y,\delta_z)} \tag{5.16}$$

and, similarly:

$$\vec{p}_x = \begin{bmatrix} p_{x,0,0,0} \\ p_{x,1,0,0} \\ p_{x,1,1,0} \\ \\ \vdots \\ p_{x,3,3,3} \end{bmatrix} \tag{5.17}$$

such that the squared second derivatives may be expressed thusly:

$$\begin{aligned} \left(\frac{\partial^2 \nu_x}{\partial x^2}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(2,0,0)} \otimes \vec{\gamma}^{(2,0,0)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(2,0,0)}\right)\vec{p}_x \\ \left(\frac{\partial^2 \nu_x}{\partial y^2}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(0,2,0)} \otimes \vec{\gamma}^{(0,2,0)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(0,2,0)}\right)\vec{p}_x \\ \left(\frac{\partial^2 \nu_x}{\partial z^2}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(0,0,2)} \otimes \vec{\gamma}^{(0,0,2)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(0,0,2)}\right)\vec{p}_x \\ \left(\frac{\partial^2 \nu_x}{\partial x \partial y}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(1,1,0)} \otimes \vec{\gamma}^{(1,1,0)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(1,1,0)}\right)\vec{p}_x \\ \left(\frac{\partial^2 \nu_x}{\partial x \partial z}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(1,0,1)} \otimes \vec{\gamma}^{(1,0,1)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(1,0,1)}\right)\vec{p}_x \\ \left(\frac{\partial^2 \nu_x}{\partial y \partial z}\right)^2 &= \vec{p}_x^{\,\intercal}\left(\vec{\gamma}^{(0,1,1)} \otimes \vec{\gamma}^{(0,1,1)}\right)\vec{p}_x = \vec{p}_x^{\,\intercal}\left(\mathbf{\Gamma}^{(0,1,1)}\right)\vec{p}_x \end{aligned} \tag{5.18}$$

where, for future convenience, we define:

$$\mathbf{\Gamma}^{(\delta_x,\delta_y,\delta_z)} = \vec{\gamma}^{(\delta_x,\delta_y,\delta_z)} \otimes \vec{\gamma}^{(\delta_x,\delta_y,\delta_z)} \tag{5.19}$$

which brings us very near the final form of the $\mathbf{V}$ matrices. Considering, for example, the

operator $\mathbf{V}_1$ from (5.7):

$$\begin{aligned}\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial x^2}\right)^2 dxdydz &= \vec{p}_x^{\,\intercal}\left(\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left[\vec{\gamma}^{(2,0,0)}\otimes\vec{\gamma}^{(2,0,0)}\right]dxdydz\right)\vec{p}_x \\ &= \vec{p}_x^{\,\intercal}\left(\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left[\mathbf{\Gamma}^{(2,0,0)}\right]dxdydz\right)\vec{p}_x \\ &= \vec{p}_x^{\,\intercal}\left(\mathbf{V}_1\right)\vec{p}_x\end{aligned} \tag{5.20}$$

incorporating the triple integration over the 3D tile region will provide with desired operator matrix. However, before the integration can be performed, we must first compute the tensor product $\vec{\gamma}\otimes\vec{\gamma}$, which we will call $\mathbf{\Gamma}$, and group like-ordered terms. In the interest of computational simplicity, orthogonal B-spline basis functions composing the tensor product $\mathbf{\Gamma}$ will be evaluated separately and finally combined. Consider the computation of an arbitrary element (α,κ) within the 64×64 matrix $\mathbf{\Gamma}$ (i.e. $\mathbf{\Gamma}(\alpha,\kappa)$):

$$\begin{aligned}\mathbf{\Gamma}^{(\delta_x,\delta_y,\delta_z)}(\alpha,\kappa) &= \vec{\gamma}^{(\delta_x,\delta_y,\delta_z)}(\alpha)\times\vec{\gamma}^{(\delta_x,\delta_y,\delta_z)}(\kappa) \\ &= \gamma_{i_1,j_1,k_1}\times\gamma_{i_2,j_2,k_2} \\ &= \sum_a \mathbf{Q}_x^{(\delta_x)}(i_1,a)\vec{x}(a) \\ &\times \sum_b \mathbf{Q}_y^{(\delta_y)}(j_1,b)\vec{y}(b) \\ &\times \sum_c \mathbf{Q}_z^{(\delta_z)}(k_1,c)\vec{z}(c) \\ &\times \sum_d \mathbf{Q}_x^{(\delta_x)}(i_2,a)\vec{y}(a) \\ &\times \sum_e \mathbf{Q}_y^{(\delta_y)}(j_2,b)\vec{y}(b) \\ &\times \sum_f \mathbf{Q}_z^{(\delta_z)}(k_2,c)\vec{z}(c)\end{aligned} \tag{5.21}$$

and regrouping like terms:

$$\begin{aligned}\mathbf{\Gamma}^{(\delta_x,\delta_y,\delta_z)}(\alpha,\kappa) &= \gamma_{i_1,j_1,k_1} \times \gamma_{i_2,j_2,k_2} \\ &= \left(\sum_a \mathbf{Q}_x^{(\delta_x)}(i_1,a)\vec{x}(a) \times \sum_d \mathbf{Q}_x^{(\delta_x)}(i_2,d)\vec{x}(d)\right) \\ &\times \left(\sum_b \mathbf{Q}_y^{(\delta_y)}(j_1,b)\vec{y}(b) \times \sum_e \mathbf{Q}_y^{(\delta_y)}(j_2,e)\vec{y}(e)\right) \\ &\times \left(\sum_c \mathbf{Q}_z^{(\delta_z)}(k_1,c)\vec{z}(c) \times \sum_f \mathbf{Q}_z^{(\delta_z)}(k_2,f)\vec{z}(f)\right) \\ &= \mathbf{\Gamma}_x^{(\delta_x)}(\alpha,\kappa) \times \mathbf{\Gamma}_y^{(\delta_y)}(\alpha,\kappa) \times \mathbf{\Gamma}_z^{(\delta_z)}(\alpha,\kappa)\end{aligned} \tag{5.22}$$

which allows for the expression of $\mathbf{\Gamma}$ as the tensor product of the components $\mathbf{\Gamma}_x, \mathbf{\Gamma}_y$, and $\mathbf{\Gamma}_z$:

$$\mathbf{\Gamma}^{(\delta_x,\delta_y,\delta_z)} = \mathbf{\Gamma}_x^{(\delta_x)} \otimes \mathbf{\Gamma}_y^{(\delta_y)} \otimes \mathbf{\Gamma}_z^{(\delta_z)}. \tag{5.23}$$

from which, given (5.20), follows:

$$\begin{aligned}\mathbf{V}^{(\delta_x,\delta_y,\delta_z)} &= \int_0^{r_z}\int_0^{r_y}\int_0^{r_x} \mathbf{\Gamma}^{(\delta_x,\delta_y,\delta_z)} dxdydz \\ &= \int_0^{r_x} \mathbf{\Gamma}_x^{(\delta_x)} dx \otimes \int_0^{r_y} \mathbf{\Gamma}_y^{(\delta_y)} dy \otimes \int_0^{r_z} \mathbf{\Gamma}_z^{(\delta_z)} dz.\end{aligned} \tag{5.24}$$

The computation of $\mathbf{\Gamma}$ is now a process of computing the three 4×4 matrices $\mathbf{\Gamma}_x, \mathbf{\Gamma}_y$, and $\mathbf{\Gamma}_z$. This operation requires multiplying each of the four B-spline basis functions with every B-spline basis function. Because each of the four rows of $\mathbf{Q}$ is a B-spline basis function, the desired operation for finding $\mathbf{\Gamma}_x$ is taking the outer product of each row of $\mathbf{Q}_x$ with every row of $\mathbf{Q}_x$. To simplify the mathematical description, row λ of a matrix $\mathbf{Q}$ will be referred to as $\vec{q}(\lambda)$ such that:

$$\mathbf{Q}_x = \begin{bmatrix}\vec{q}_x^{\,\intercal}(0) \\ \vec{q}_x^{\,\intercal}(1) \\ \vec{q}_x^{\,\intercal}(2) \\ \vec{q}_x^{\,\intercal}(3)\end{bmatrix} \tag{5.25}$$

thus providing an expression for $\mathbf{\Gamma}_x$:

$$\begin{aligned}\mathbf{\Gamma}_x &= \left[\begin{array}{c|c|c|c} \vec{q}_x(0)\otimes\vec{q}_x(0) & \vec{q}_x(0)\otimes\vec{q}_x(1) & \vec{q}_x(0)\otimes\vec{q}_x(2) & \vec{q}_x(0)\otimes\vec{q}_x(3) \\ \hline \vec{q}_x(1)\otimes\vec{q}_x(0) & \vec{q}_x(1)\otimes\vec{q}_x(1) & \vec{q}_x(1)\otimes\vec{q}_x(2) & \vec{q}_x(1)\otimes\vec{q}_x(3) \\ \hline \vec{q}_x(2)\otimes\vec{q}_x(0) & \vec{q}_x(2)\otimes\vec{q}_x(1) & \vec{q}_x(2)\otimes\vec{q}_x(2) & \vec{q}_x(2)\otimes\vec{q}_x(3) \\ \hline \vec{q}_x(3)\otimes\vec{q}_x(0) & \vec{q}_x(3)\otimes\vec{q}_x(1) & \vec{q}_x(3)\otimes\vec{q}_x(2) & \vec{q}_x(3)\otimes\vec{q}_x(3) \end{array}\right] \\ &= \left[\begin{array}{c|c|c|c} \mathbf{\Xi}_{x,0,0} & \mathbf{\Xi}_{x,0,1} & \mathbf{\Xi}_{x,0,2} & \mathbf{\Xi}_{x,0,3} \\ \hline \mathbf{\Xi}_{x,1,0} & \mathbf{\Xi}_{x,1,1} & \mathbf{\Xi}_{x,1,2} & \mathbf{\Xi}_{x,1,3} \\ \hline \mathbf{\Xi}_{x,2,0} & \mathbf{\Xi}_{x,2,1} & \mathbf{\Xi}_{x,2,2} & \mathbf{\Xi}_{x,2,3} \\ \hline \mathbf{\Xi}_{x,3,0} & \mathbf{\Xi}_{x,3,1} & \mathbf{\Xi}_{x,3,2} & \mathbf{\Xi}_{x,3,3} \end{array}\right]\end{aligned} \tag{5.26}$$

where, for future convenience, we define the 4×4 matrix $\mathbf{\Xi}_{x,\lambda_1,\lambda_2}$ as:

$$\mathbf{\Xi}_{x,\lambda_1,\lambda_2} = \vec{q}_x(\lambda_1)\otimes\vec{q}_x(\lambda_2) \tag{5.27}$$

which possesses the form:

$$\mathbf{\Xi}_{x,\lambda_1,\lambda_2} = \begin{bmatrix} c_0x^0 & c_1x^1 & c_2x^2 & c_3x^3 \\ c_4x^1 & c_5x^2 & c_6x^3 & c_7x^4 \\ c_8x^2 & c_9x^3 & c_{10}x^4 & c_{11}x^5 \\ c_{12}x^3 & c_{13}x^4 & c_{14}x^5 & c_{15}x^6 \end{bmatrix} \tag{5.28}$$

in grouping like-order polynomial terms for the integration over the region r_x, we define the vector $\vec{\sigma}$:

$$\vec{\sigma}_{x,\lambda_1,\lambda_2} = \left[\begin{array}{l} \mathbf{\Xi}(0,0) \\ \mathbf{\Xi}(0,1)+\mathbf{\Xi}(1,0) \\ \mathbf{\Xi}(0,2)+\mathbf{\Xi}(1,1)+\mathbf{\Xi}(2,0) \\ \mathbf{\Xi}(0,3)+\mathbf{\Xi}(1,2)+\mathbf{\Xi}(2,1)+\mathbf{\Xi}(3,0) \\ \mathbf{\Xi}(1,3)+\mathbf{\Xi}(2,2)+\mathbf{\Xi}(3,1) \\ \mathbf{\Xi}(2,3)+\mathbf{\Xi}(3,2) \\ \mathbf{\Xi}(3,3) \end{array}\right]_{x,\lambda_1,\lambda_2} \tag{5.29}$$

and defining:

$$\vec{\psi}_x = \begin{bmatrix} r_x \\ \frac{1}{2}r_x^2 \\ \frac{1}{3}r_x^3 \\ \frac{1}{4}r_x^4 \\ \frac{1}{5}r_x^5 \\ \frac{1}{6}r_x^6 \\ \frac{1}{7}r_x^7 \end{bmatrix}, \quad \vec{\psi}_y = \begin{bmatrix} r_y \\ \frac{1}{2}r_y^2 \\ \frac{1}{3}r_y^3 \\ \frac{1}{4}r_y^4 \\ \frac{1}{5}r_y^5 \\ \frac{1}{6}r_y^6 \\ \frac{1}{7}r_y^7 \end{bmatrix}, \quad \vec{\psi}_z = \begin{bmatrix} r_z \\ \frac{1}{2}r_z^2 \\ \frac{1}{3}r_z^3 \\ \frac{1}{4}r_z^4 \\ \frac{1}{5}r_z^5 \\ \frac{1}{6}r_z^6 \\ \frac{1}{7}r_z^7 \end{bmatrix} \tag{5.30}$$

allows us to express the integrals of $\mathbf{\Gamma}_x, \mathbf{\Gamma}_y$ and $\mathbf{\Gamma}_z$ thusly:

$$\begin{aligned}
\bar{\mathbf{\Gamma}}_x^{(\delta_x)} &= \int_0^{r_x} \mathbf{\Gamma}_x^{(\delta_x)} dx = \left[\begin{array}{c|c|c|c} \vec{\sigma}_{x,0,0}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,0,1}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,0,2}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,0,3}^{\intercal}\vec{\psi}_x \\ \hline \vec{\sigma}_{x,1,0}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,1,1}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,1,2}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,1,3}^{\intercal}\vec{\psi}_x \\ \hline \vec{\sigma}_{x,2,0}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,2,1}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,2,2}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,2,3}^{\intercal}\vec{\psi}_x \\ \hline \vec{\sigma}_{x,3,0}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,3,1}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,3,2}^{\intercal}\vec{\psi}_x & \vec{\sigma}_{x,3,3}^{\intercal}\vec{\psi}_x \end{array}\right] \\
\bar{\mathbf{\Gamma}}_y^{(\delta_y)} &= \int_0^{r_y} \mathbf{\Gamma}_y^{(\delta_y)} dy = \left[\begin{array}{c|c|c|c} \vec{\sigma}_{y,0,0}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,0,1}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,0,2}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,0,3}^{\intercal}\vec{\psi}_y \\ \hline \vec{\sigma}_{y,1,0}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,1,1}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,1,2}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,1,3}^{\intercal}\vec{\psi}_y \\ \hline \vec{\sigma}_{y,2,0}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,2,1}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,2,2}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,2,3}^{\intercal}\vec{\psi}_y \\ \hline \vec{\sigma}_{y,3,0}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,3,1}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,3,2}^{\intercal}\vec{\psi}_y & \vec{\sigma}_{y,3,3}^{\intercal}\vec{\psi}_y \end{array}\right] \\
\bar{\mathbf{\Gamma}}_z^{(\delta_z)} &= \int_0^{r_z} \mathbf{\Gamma}_z^{(\delta_z)} dz = \left[\begin{array}{c|c|c|c} \vec{\sigma}_{z,0,0}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,0,1}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,0,2}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,0,3}^{\intercal}\vec{\psi}_z \\ \hline \vec{\sigma}_{z,1,0}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,1,1}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,1,2}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,1,3}^{\intercal}\vec{\psi}_z \\ \hline \vec{\sigma}_{z,2,0}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,2,1}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,2,2}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,2,3}^{\intercal}\vec{\psi}_z \\ \hline \vec{\sigma}_{z,3,0}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,3,1}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,3,2}^{\intercal}\vec{\psi}_z & \vec{\sigma}_{z,3,3}^{\intercal}\vec{\psi}_z \end{array}\right]
\end{aligned} \tag{5.31}$$

which, when applied to (5.24), yields the expressions for the six desired operators:

$$\begin{aligned}
\mathbf{V}_1 &= \mathbf{V}^{(2,0,0)} = \bar{\mathbf{\Gamma}}_x^{(2)} \otimes \bar{\mathbf{\Gamma}}_y^{(0)} \otimes \bar{\mathbf{\Gamma}}_z^{(0)} \\
\mathbf{V}_2 &= \mathbf{V}^{(0,2,0)} = \bar{\mathbf{\Gamma}}_x^{(0)} \otimes \bar{\mathbf{\Gamma}}_y^{(2)} \otimes \bar{\mathbf{\Gamma}}_z^{(0)} \\
\mathbf{V}_3 &= \mathbf{V}^{(0,0,2)} = \bar{\mathbf{\Gamma}}_x^{(0)} \otimes \bar{\mathbf{\Gamma}}_y^{(0)} \otimes \bar{\mathbf{\Gamma}}_z^{(2)} \\
\mathbf{V}_4 &= \mathbf{V}^{(1,1,0)} = \bar{\mathbf{\Gamma}}_x^{(1)} \otimes \bar{\mathbf{\Gamma}}_y^{(1)} \otimes \bar{\mathbf{\Gamma}}_z^{(0)} \\
\mathbf{V}_5 &= \mathbf{V}^{(1,0,1)} = \bar{\mathbf{\Gamma}}_x^{(1)} \otimes \bar{\mathbf{\Gamma}}_y^{(0)} \otimes \bar{\mathbf{\Gamma}}_z^{(1)} \\
\mathbf{V}_6 &= \mathbf{V}^{(0,1,1)} = \bar{\mathbf{\Gamma}}_x^{(0)} \otimes \bar{\mathbf{\Gamma}}_y^{(1)} \otimes \bar{\mathbf{\Gamma}}_z^{(1)}
\end{aligned} \tag{5.32}$$

which can be used to express the terms in (5.1) thusly:

$$\begin{aligned}
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial x^2}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_1\right)\vec{p}_x \\
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial y^2}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_2\right)\vec{p}_x \\
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial z^2}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_3\right)\vec{p}_x \\
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial x\partial y}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_4\right)\vec{p}_x \\
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial x\partial z}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_5\right)\vec{p}_x \\
\int_0^{r_z}\int_0^{r_y}\int_0^{r_x}\left(\frac{\partial^2\nu_x}{\partial y\partial z}\right)^2 dxdydz &= \vec{p}_x^{\intercal}\left(\mathbf{V}_6\right)\vec{p}_x
\end{aligned} \tag{5.33}$$

which allows for the concise re-expression of the smoothness metric:

$$\begin{aligned}
S_{tile} = {} & \vec{p}_x^{\intercal}\left(\mathbf{V}_1\right)\vec{p}_x + \vec{p}_x^{\intercal}\left(\mathbf{V}_2\right)\vec{p}_x + \vec{p}_x^{\intercal}\left(\mathbf{V}_3\right)\vec{p}_x \\
& + \vec{p}_x^{\intercal}\left(\mathbf{V}_4\right)\vec{p}_x + \vec{p}_x^{\intercal}\left(\mathbf{V}_5\right)\vec{p}_x + \vec{p}_x^{\intercal}\left(\mathbf{V}_6\right)\vec{p}_x \\
& + \vec{p}_y^{\intercal}\left(\mathbf{V}_1\right)\vec{p}_y + \vec{p}_y^{\intercal}\left(\mathbf{V}_2\right)\vec{p}_y + \vec{p}_y^{\intercal}\left(\mathbf{V}_3\right)\vec{p}_y \\
& + \vec{p}_y^{\intercal}\left(\mathbf{V}_4\right)\vec{p}_y + \vec{p}_y^{\intercal}\left(\mathbf{V}_5\right)\vec{p}_y + \vec{p}_y^{\intercal}\left(\mathbf{V}_6\right)\vec{p}_y \\
& + \vec{p}_z^{\intercal}\left(\mathbf{V}_1\right)\vec{p}_z + \vec{p}_z^{\intercal}\left(\mathbf{V}_2\right)\vec{p}_z + \vec{p}_z^{\intercal}\left(\mathbf{V}_3\right)\vec{p}_z \\
& + \vec{p}_z^{\intercal}\left(\mathbf{V}_4\right)\vec{p}_z + \vec{p}_z^{\intercal}\left(\mathbf{V}_5\right)\vec{p}_z + \vec{p}_z^{\intercal}\left(\mathbf{V}_6\right)\vec{p}_z
\end{aligned} \tag{5.34}$$

whose derivative with respect to the B-spline control-point parameterization P may be expressed thusly:

$$\begin{aligned}\frac{\partial S_{tile}}{\partial P} = &\ (2 \times \mathbf{V}_1\vec{p}_x) + (2 \times \mathbf{V}_2\vec{p}_x) + (2 \times \mathbf{V}_3\vec{p}_x) \\ + &\ (2 \times \mathbf{V}_4\vec{p}_x) + (2 \times \mathbf{V}_5\vec{p}_x) + (2 \times \mathbf{V}_6\vec{p}_x) \\ + &\ (2 \times \mathbf{V}_1\vec{p}_y) + (2 \times \mathbf{V}_2\vec{p}_y) + (2 \times \mathbf{V}_3\vec{p}_y) \\ + &\ (2 \times \mathbf{V}_4\vec{p}_y) + (2 \times \mathbf{V}_5\vec{p}_y) + (2 \times \mathbf{V}_6\vec{p}_y) \\ + &\ (2 \times \mathbf{V}_1\vec{p}_z) + (2 \times \mathbf{V}_2\vec{p}_z) + (2 \times \mathbf{V}_3\vec{p}_z) \\ + &\ (2 \times \mathbf{V}_4\vec{p}_z) + (2 \times \mathbf{V}_5\vec{p}_z) + (2 \times \mathbf{V}_6\vec{p}_z)\end{aligned} \tag{5.35}$$

5.2 Algorithmic Implementation

Because the matrix operators $\mathbf{V}_1$ through $\mathbf{V}_6$ depend only on the B-spline grid configuration, they may be pre-computed before the registration begins and simply reused within each iteration. Therefore, the algorithmic implementation of the regularization process consists of two stages: an initialization stage and an update stage. During the initialization stage, the matrix operators $\mathbf{V}_1$ through $\mathbf{V}_6$ are simply constructed and stored. The update stage occurs at the end of each optimization iteration and consists of applying the pre-computed matrix operators to the B-spline coefficients in order to compute the vector field smoothness S and its derivative with respect to each control point $\partial S/\partial P$. This stage concludes by adding the smoothness S to the overall cost function C as in 5.2 and the smoothness derivative $\partial S/\partial P$ to the cost function gradient $\partial C/\partial P$ as per 5.3.

First, we consider the initialization process, which is performed along with all other B-spline initialization procedures. Only the B-spline control point spacing in each spatial dimension is required for generation of the matrix operators $\mathbf{V}_1$ through $\mathbf{V}_6$. This initialization process is described algorithmically in Fig. 5.1 and Fig. 5.2. Lines 1–4 normalize the

1: /* Generate the Q matrices from (5.8) */
2: $\mathbf{Q}_x^{(0)} = \mathbf{B}\mathbf{R}_x$
3: $\mathbf{Q}_y^{(0)} = \mathbf{B}\mathbf{R}_y$
4: $\mathbf{Q}_z^{(0)} = \mathbf{B}\mathbf{R}_z$
5:
6: /* Generate first and second derivatives as in (5.13) and (5.14) */
7: $\mathbf{Q}_x^{(1)} = \mathbf{Q}_x^{(0)}\boldsymbol{\Delta}^{(1)}$
8: $\mathbf{Q}_y^{(1)} = \mathbf{Q}_y^{(0)}\boldsymbol{\Delta}^{(1)}$
9: $\mathbf{Q}_z^{(1)} = \mathbf{Q}_z^{(0)}\boldsymbol{\Delta}^{(1)}$
10: $\mathbf{Q}_x^{(2)} = \mathbf{Q}_x^{(0)}\boldsymbol{\Delta}^{(2)}$
11: $\mathbf{Q}_y^{(2)} = \mathbf{Q}_y^{(0)}\boldsymbol{\Delta}^{(2)}$
12: $\mathbf{Q}_z^{(2)} = \mathbf{Q}_z^{(0)}\boldsymbol{\Delta}^{(2)}$
13:
14: /* Generate $\bar{\boldsymbol{\Gamma}}_x^{(0)}$, $\bar{\boldsymbol{\Gamma}}_x^{(1)}$, $\bar{\boldsymbol{\Gamma}}_x^{(2)}$ as per (5.26) - (5.31) */
15: $\bar{\boldsymbol{\Gamma}}_x^{(0)} = \texttt{eval\_integral}(\mathbf{Q}_x^{(0)}, r_x)$
16: $\bar{\boldsymbol{\Gamma}}_x^{(1)} = \texttt{eval\_integral}(\mathbf{Q}_x^{(1)}, r_x)$
17: $\bar{\boldsymbol{\Gamma}}_x^{(2)} = \texttt{eval\_integral}(\mathbf{Q}_x^{(2)}, r_x)$
18:
19: /* Generate $\bar{\boldsymbol{\Gamma}}_y^{(0)}$, $\bar{\boldsymbol{\Gamma}}_y^{(1)}$, $\bar{\boldsymbol{\Gamma}}_y^{(2)}$ as per (5.26) - (5.31) */
20: $\bar{\boldsymbol{\Gamma}}_y^{(0)} = \texttt{eval\_integral}(\mathbf{Q}_y^{(0)}, r_y)$
21: $\bar{\boldsymbol{\Gamma}}_y^{(1)} = \texttt{eval\_integral}(\mathbf{Q}_y^{(1)}, r_y)$
22: $\bar{\boldsymbol{\Gamma}}_y^{(2)} = \texttt{eval\_integral}(\mathbf{Q}_y^{(2)}, r_y)$
23:
24: /* Generate $\bar{\boldsymbol{\Gamma}}_z^{(0)}$, $\bar{\boldsymbol{\Gamma}}_z^{(1)}$, $\bar{\boldsymbol{\Gamma}}_z^{(2)}$ as per (5.26) - (5.31) */
25: $\bar{\boldsymbol{\Gamma}}_z^{(0)} = \texttt{eval\_integral}(\mathbf{Q}_z^{(0)}, r_z)$
26: $\bar{\boldsymbol{\Gamma}}_z^{(1)} = \texttt{eval\_integral}(\mathbf{Q}_z^{(1)}, r_z)$
27: $\bar{\boldsymbol{\Gamma}}_z^{(2)} = \texttt{eval\_integral}(\mathbf{Q}_z^{(2)}, r_z)$
28:
29: /* Generate V_1 through V_6 as per (5.32) */
30: $\mathbf{V}_1 = \bar{\boldsymbol{\Gamma}}_x^{(2)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(0)}$
31: $\mathbf{V}_2 = \bar{\boldsymbol{\Gamma}}_x^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(2)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(0)}$
32: $\mathbf{V}_3 = \bar{\boldsymbol{\Gamma}}_x^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(2)}$
33: $\mathbf{V}_4 = \bar{\boldsymbol{\Gamma}}_x^{(1)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(1)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(0)}$
34: $\mathbf{V}_5 = \bar{\boldsymbol{\Gamma}}_x^{(1)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(1)}$
35: $\mathbf{V}_6 = \bar{\boldsymbol{\Gamma}}_x^{(0)} \otimes \bar{\boldsymbol{\Gamma}}_y^{(1)} \otimes \bar{\boldsymbol{\Gamma}}_z^{(1)}$

Figure 5.1: Initialization of the regularizer

B-spline basis by the control point grid spacing for each Cartesian axis as per (5.8). This is necessary so that any given voxel coordinate within a tile will normalize within the required domain [0,1] within which the B-spline basis functions are defined. Lines 6–12 generate the first and second order spatial derivatives of the normalized B-spline basis functions as

1: `function eval_integral`(Q, r)
2: **for** $\lambda_2 = 0$ to 3 step 1
3: **for** $\lambda_1 = 0$ to 3 step 1
4: `/* As per (5.25) and (5.27) */`
5: $\boldsymbol{\Xi} = \vec{q}(\lambda_1) \otimes \vec{q}(\lambda_2)$
6:
7: `/* As per (5.28) and (5.31) */`
8: $\bar{\boldsymbol{\Gamma}}(\lambda_1, \lambda_2) = \frac{r^1}{1}(\boldsymbol{\Xi}(0,0))$
9: $+\frac{r^2}{2}(\boldsymbol{\Xi}(0,1) + \boldsymbol{\Xi}(1,0))$
10: $+\frac{r^3}{3}(\boldsymbol{\Xi}(0,2) + \boldsymbol{\Xi}(1,1) + \boldsymbol{\Xi}(2,0))$
11: $+\frac{r^4}{4}(\boldsymbol{\Xi}(0,3) + \boldsymbol{\Xi}(1,2) + \boldsymbol{\Xi}(2,1) + \boldsymbol{\Xi}(3,0))$
12: $+\frac{r^5}{5}(\boldsymbol{\Xi}(1,3) + \boldsymbol{\Xi}(2,2) + \boldsymbol{\Xi}(3,1))$
13: $+\frac{r^6}{6}(\boldsymbol{\Xi}(2,3) + \boldsymbol{\Xi}(3,2))$
14: $+\frac{r^7}{7}(\boldsymbol{\Xi}(3,3))$
15: **end for**
16: **end for**
17:
18: **return** $\bar{\boldsymbol{\Gamma}}$
19: `end function`

Figure 5.2: Generation of integrated sub-matrices $\bar{\boldsymbol{\Gamma}}$

per (5.13) and (5.14). Lines 14–17 generate the $\bar{\boldsymbol{\Gamma}}_{\mathbf{x}}^{(\mathbf{0})}$, $\bar{\boldsymbol{\Gamma}}_{\mathbf{x}}^{(\mathbf{1})}$, and $\bar{\boldsymbol{\Gamma}}_{\mathbf{x}}^{(\mathbf{2})}$ matrices by squaring (5.4) for the the zeroth, first, and second order normalized B-spline basis functions $\mathbf{Q}_{\mathbf{x}}^{(\mathbf{0})}$, $\mathbf{Q}_{\mathbf{x}}^{(\mathbf{1})}$, $\mathbf{Q}_{\mathbf{x}}^{(\mathbf{2})}$ and integrating the resulting 6^{th} order polynomials over the control-point spacing in the x-direction as per (5.26) - (5.31). Similarly , this operation is performed in the y- and z-directions to obtain $\bar{\boldsymbol{\Gamma}}_{\mathbf{y}}^{(\mathbf{0})}$, $\bar{\boldsymbol{\Gamma}}_{\mathbf{y}}^{(\mathbf{1})}$, $\bar{\boldsymbol{\Gamma}}_{\mathbf{y}}^{(\mathbf{2})}$, $\bar{\boldsymbol{\Gamma}}_{\mathbf{z}}^{(\mathbf{0})}$, $\bar{\boldsymbol{\Gamma}}_{\mathbf{z}}^{(\mathbf{1})}$, and $\bar{\boldsymbol{\Gamma}}_{\mathbf{z}}^{(\mathbf{2})}$. As shown, this process of squaring and integrating is performed within the function `eval_integral( )`, which is algorithmically described in Fig. 5.2. Finally, lines 29–35 complete the initialization process by computing the $\mathbf{V}_1$–$\mathbf{V}_6$ matrices via the tensor product as per 5.32.

Once the matrix operators $\mathbf{V}_1$ through $\mathbf{V}_6$ have been obtained, the smoothness S and its derivative $\partial S/\partial P$ may be quickly computed for any given tile via (5.34) and (5.35). Fig. 5.3 describes the process of computing the smoothness for the entire vector field by sequentially computing the vector field smoothness for each tile S_{tile} and accumulating the

```
1: S = 0
2: for tile_idx = 0 to NUM_TILES-1 step 1
3:     /* Generate array containing indices for tile's 64 control points */
4:     cp_lut = find_control_points(tile_idx)
5:
6:     /* Sum partial derivatives as per (5.34) and (5.35) */
7:     S += apply_operator(cp_lut,V_1)
8:     S += apply_operator(cp_lut,V_2)
9:     S += apply_operator(cp_lut,V_3)
10:    S += apply_operator(cp_lut,V_4)
11:    S += apply_operator(cp_lut,V_5)
12:    S += apply_operator(cp_lut,V_6)
13: end for
```

Figure 5.3: The update stage of the regularizer

results as in (5.5). For each iteration, we first use the tile's index `tile_index` to compute the indices of the 64 control points that are associated with that tile. These indices are stored into the 64 element array `cp_lut` as shown in line 4. The remainder of the iteration computes the smoothness of the individual tile's vector field by applying the six matrix operators as shown in lines 7–12 and summing the results. For subsequent iterations we continue to accumulate into S, thereby computing the smoothness for the entire vector field as prescribed by (5.5).

Inspection of Fig. 5.4 reveals the steps involved in the application of a given $\mathbf{V}$ matrix operator to a set of 64 control points. Here, lines 4–12 implement the straight-forward matrix multiplication required by the $\vec{p}^{\,\intercal}(\mathbf{V})\,\vec{p}$ operation found in (5.34). The array $V[\,]$ holds the 64×64 matrix operator being applied to the B-spline coefficients stored within P, which is x, y, z-interlaced as shown in Fig. 3.4 in Chapter 3. The control-point index lookup table `cp_lut` passed into the function contains the indices of the 64 control-points for the tile in question; thus its use in lines 5–7 and 10–12 serves as a means of converting from tile-centric control-point indexing (ranging from 0 to 63) to the absolute control-point

```
1: function apply_operator(cp_lut, V)
2:   for j = 0 to 63 step 1
3:     /* Compute tile smoothness as per (5.34).  */
4:     for i = 0 to 63 step 1
5:       tmp_x[j] += P[3*cp_lut[i]+0] * V[64 * j + i]
6:       tmp_y[j] += P[3*cp_lut[i]+1] * V[64 * j + i]
7:       tmp_z[j] += P[3*cp_lut[i]+2] * V[64 * j + i]
8:     end for
9:
10:    S_tile += tmp_x[j] * P[3*cp_lut[i]+0]
11:    S_tile += tmp_y[j] * P[3*cp_lut[i]+1]
12:    S_tile += tmp_z[j] * P[3*cp_lut[i]+2]
13:    /* ------------------------------------- */
14:
15:    /* Compute tiles smoothness derivative as per (5.35).  */
16:    ∂C/∂P[3*cp_lut[j]+0] += 2 * λ *  tmp_x[j]
17:    ∂C/∂P[3*cp_lut[j]+1] += 2 * λ *  tmp_y[j]
18:    ∂C/∂P[3*cp_lut[j]+2] += 2 * λ *  tmp_z[j]
19:  end for
20:
21:  return S_tile
22: end function
```

Figure 5.4: Application of the regularization operators to the B-spline coefficients

indexing used within the B-spline coefficient array P. Furthermore, due to the operational similarity found in the computation of the tile smoothness S_{tile} and its derivative $\partial S_{tile}/\partial P$, we are able to compute (5.35) in place using the partial solutions `tmp_x[ ], tmp_y[ ],` and `tmp_z[ ]` from the tile smoothness computation as shown in lines 16–18.

Finally, it should be noted that since the computation of an individual tile's smoothness is independent of all other tiles, it is possible to parallelize the algorithm by simply spreading the iterations of Fig. 5.3 across N cores and performing a sum reduction on the resulting N values of S. Additionally, because lines 16–18 of Fig. 5.4 attempt to update 64 control-point $\partial C/\partial P$ values by appending $\partial S/\partial P$ as in (5.3), this cost function gradient update operation must be modified to be thread safe. For this, the same thread safe, parallel method used by Chapters 2 and 3 to update a set of 64 $\partial C/\partial P$ values given a tile of $\partial C/\partial \nu$ values may

be employed since the data structure and operation is identical. The only difference being $\partial S/\partial P$ data replaces $\partial C/\partial \nu$ data, but is otherwise identical as the 64 solutions for the tile affect the 64 surrounding control points in the exact same fashion.

5.3 Performance Evaluation

This section presents experimental results obtained for single and multi-core CPU implementations in terms of both execution speed and registration quality. A numerically based central differencing implementation that computes the vector field smoothness by operating directly on the vector field is provided as a basis for comparison. All implementations are evaluated in terms of execution speed as a function of 1) volume size given a fixed control point grid and 2) control point spacing given a fixed volume size. Additionally, the processing time for a single tile as a function of the tile's size is also investigated. As previously described, the sequential analytic implementation computes the smoothness by applying the $\mathbf{V}$ matrix operators to the B-spline coefficients pertaining to each tile – one tile at a time until all tiles within the volume are processed. Because the computation of each individual tile's smoothness is independent of other tiles, the parallel analytic implementation may process the smoothness for N tiles in parallel given N cores. Additionally, this implementation uses the parallel gradient update method developed in chapters 2 and 3 to further accelerate the algorithm. These analytic implementations provide interesting contrast to the numerical method of smoothness computation, which is based on central differencing of raw vector field values at each individual voxel in the volume. Consequently, the numerical method differs from the analytic methods in that it is voxel resolution centric and not control grid resolution centric. This results in the two methods having not only

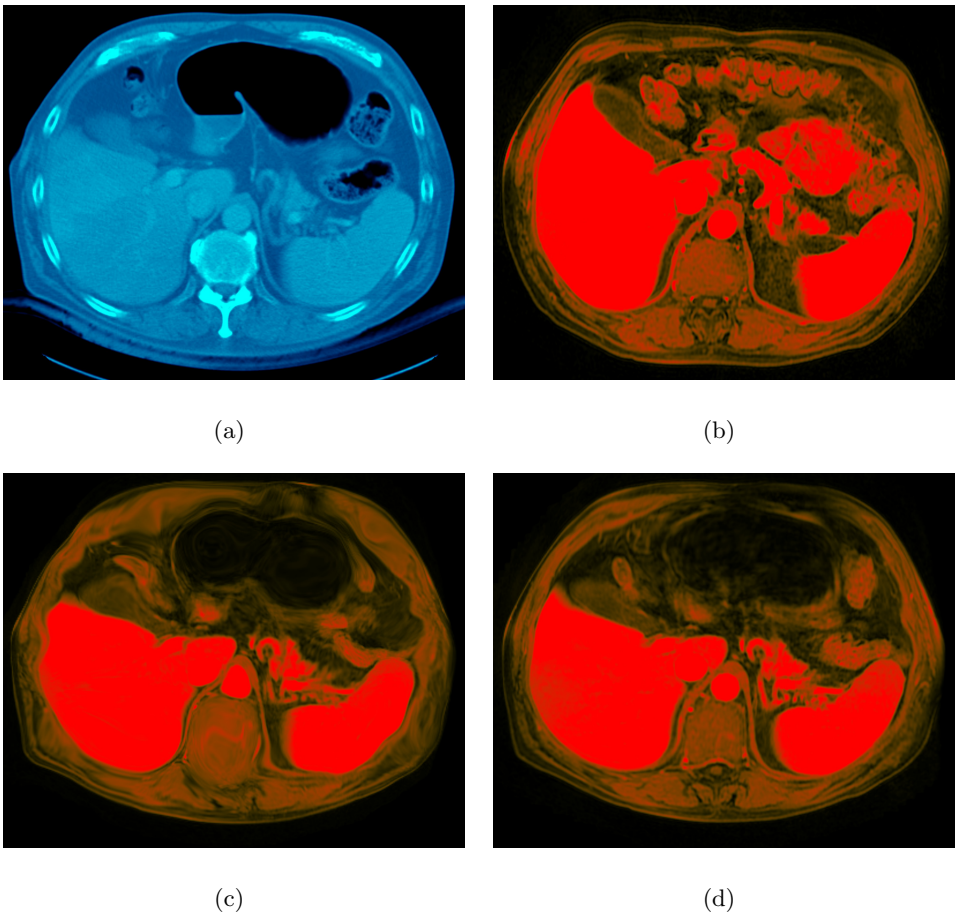
differing processing speeds but fundamentally different execution-time profiles with respect to the various input parameters.

Lastly, we demonstrate the effectiveness of regularization for a multi-modal case requiring the registration of an intra-operative CT to a pre-operative MRI. Warped moving images with are without regularization are shown as well as their associated deformation vector field transforms. Additionally, post-warp CT-MRI fusion images are provided in order to more clearly demonstrate the affects of regularization on registration solution convergence. All tests reported within this section were performed using a machine equipped with an Intel quad-core i7 920 processor with each core clocked at 2.6 GHz, and 12 GB of RAM.

5.3.1 Registration Quality

As previously mentioned, the need for regularization arises from the inverse nature of the image registration problem. In other words, the deformation vector field solution we seek is non-unique – there are many possible image mappings that will serve to satisfy our imposed score requirement dictating a "good" registration. Our imposition of a score penalty based on vector field smoothness constrains the solution space to a subset. Constraining the solution space in this fashion is particularly beneficial when performing registration using mutual information (MI) as a similarity metric.

Fig. 5.5 shows axial cross-sections of thoracic image volumes involved in an MRI to CT multi-modal registration using mutual information. CT images are shown in blue and MRI images are shown in red. Fig. 5.5(a) is a cross-section of the CT volume serving as the fixed image. Similarly, Fig. 5.5(b) is a cross-section of the MRI serving as the moving image. Fig. 5.5(c) shows the result of a carefully conducted five-stage multi-resolution B-spline

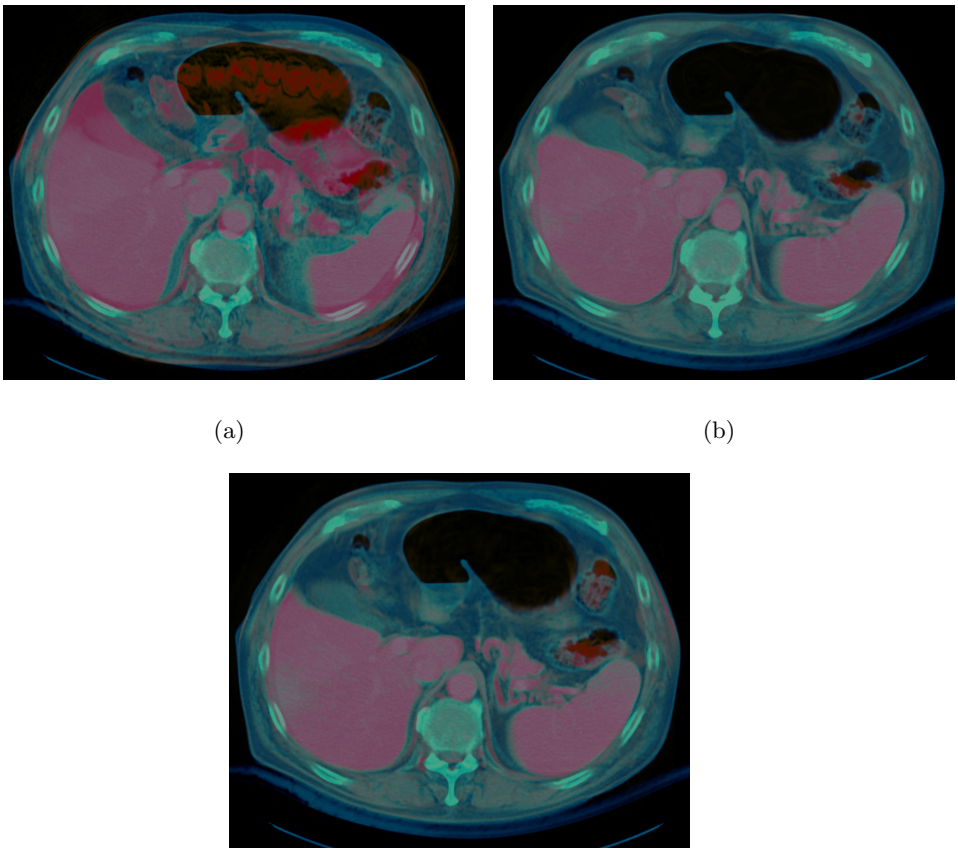


(a) (b)

(c) (d)

Figure 5.5: Warped thoracic images with and without regularization. (a) The static CT image, (b) the moving MRI image, (c) the warped MRI after registration without regularization, and (d) the warped MRI after registration with a regularization factor of $\lambda = 5 \times 10^{-6}$.

grid registration that does not impose any regularization on the deformation vector field. Fig. 5.5(d) shows the same 5-stage registration that imposes the smoothness penalty term with a weight of $\lambda = 5 \times 10^{-6}$. Fig. 5.6(a) shows the un-warped MRI image superimposed upon the CT image prior to deformable registration. As shown, the two images have been rigidly registered manually to one another such that the common vertebra are aligned. Notice the significant liver deformation on the left of the thorax and the spleen deformation found on the right posterior. The aim of the deformable registration is to recover the



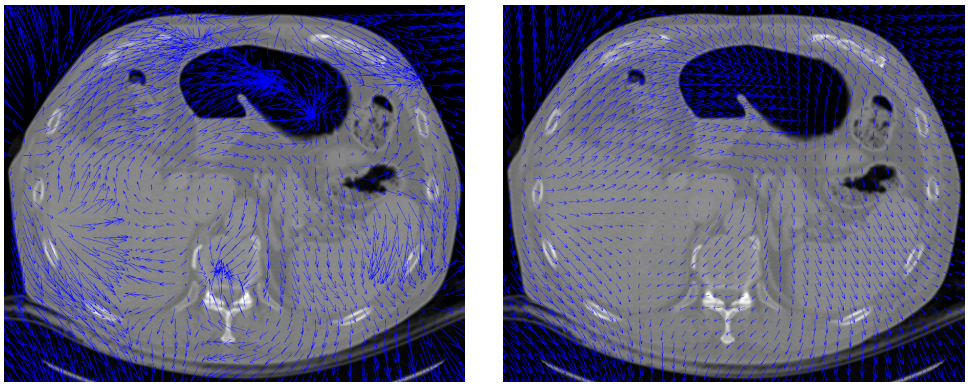
(a) (b)

(c)

Figure 5.6: Fusion of MRI and CT thoracic images with and without regularization. (a) The unwarped MRI image superimposed on CT image, (b) the MRI warped without regularization superimposed on CT, and (c) the MRI warped with regularization and superimposed on the CT.

deformation vector field accuracy describing the movement of these organs and surrounding dynamic anatomy.

We will first analyze the result without regularization. Despite this deformation being physically impossible, as we will show, it does meet the mutual information criteria for a good registration. Consequently, the fusion of this solution with the fixed CT image is visually favorable as shown in Fig. 5.6(b). However, notice how the MRI image warped by the unregularized vector field, when viewed by itself in Fig. 5.5(c), appears "wavy" and

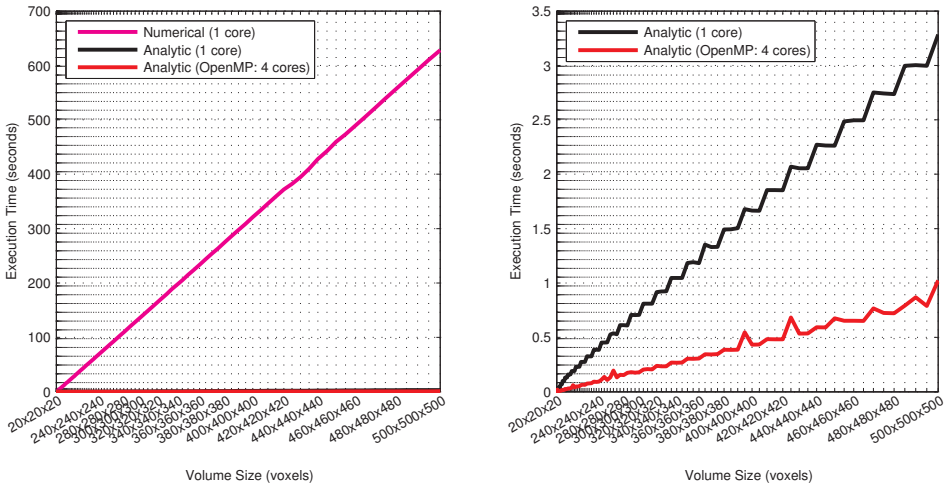


(a) (b)

Figure 5.7: Multi-modal vector fields with and without regularization. (a) Superimposition of a 2D slice of the 3D deformation vector field upon the corresponding axial thoracic CT slice. This vector field was generated from an MRI to CT registration that did not employ regularization. (b) Superimposition of a vector field upon a CT image that underwent the same registration but with a regularization penalty weight of $\lambda = 5 \times 10^{-6}$.

exhibits artifacting reminiscent of a thin film of oil – particularly pronounced within the spinal column and the anterior layer of fat around the abdomen. Naturally, the human body is incapable of deforming in this fashion and direct inspection of the deformation field shown in Fig. 5.7(a) confirms its implausibility.

By contrast, Fig. 5.5(d) shows the result for the same registration performed with a regularization penalty weight of $\lambda = 5 \times 10^{-6}$. Notice how the artifacting is no longer present – the deformation appears physically sane; accordingly, the deformation vector field shown in Fig. 5.7(b) confirms that the mapping is sane. Finally, the super-imposition of this warped MRI upon the reference CT image shown in Fig. 5.6(c) represents an accurate anatomical correlation between the intra-operative CT and pre-operative MRI images.

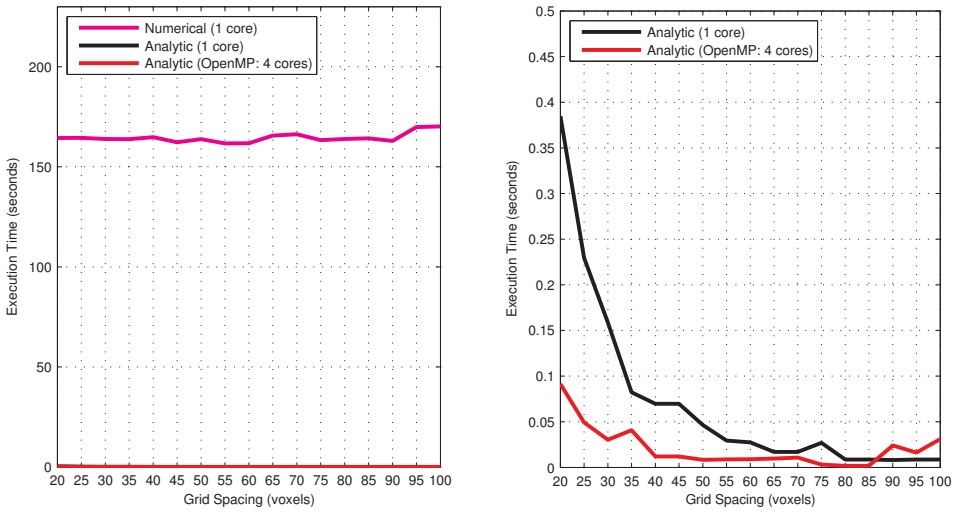


(a) (b)

Figure 5.8: Performance of the regularizer with respect to volume size. (a) Execution times for each regularization implementation as a function of input volume size. Control-point spacing is fixed at $5 \times 5 \times 5$ voxels. (b) Only showing analytic implementations.

5.3.2 Sensitivity to Volume Size

This set of tests characterizes each implementation's sensitivity, in terms of execution time, to increasing volume size where the volumes are synthetically generated. We fix the control-point spacing at 15 voxels in each physical dimension and increase the volume size in steps of $5 \times 5 \times 5$ voxels. For each volume size, we record the execution time incurred by a single iteration of the regularization process. Fig. 5.8(a) summarizes the results. As expected for the numerically derived solution, the execution time increases linearly with the number of voxels involved. Fig. 5.8(b) shows the same graph excluding the numerical method. Notice how the execution time for the analytic increases only when the volume size increases by a multiple of the control-point grid spacing. This is because the analytic algorithms operates directly on the control-point coefficients in order to compute the vector field smoothess for a tile. Therefore, the execution time depends only on the number of tiles within the volume.



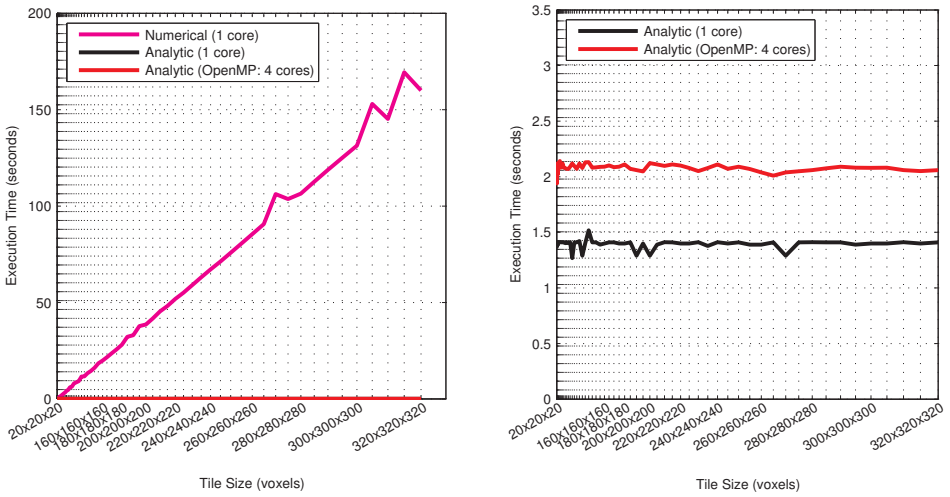
(a) (b)

Figure 5.9: Regularization performance with respet to control grid spacing. (a) Execution times for each regularization implementation as a function of control-point grid size. Input volume size is fixed at $320 \times 320 \times 320$ voxels. (b) Only showing analytic implementations.

Increasing the volume size by a multiple of the control-point spacing introduces addition tiles; therefore incurring additional overall processing time for the volume. In the case of a large test volume of $500 \times 500 \times 500$ voxels, the single core analytic implementation exhibits a speedup of 191 times over the numerical method. For the same volume size, the parallel analytic method achieves an additional $3.2x$ speedup – a speedup of $613x$ with respect to the numerical method.

5.3.3 Sensitivity to Control-Point Spacing

Fig. 5.9(a) shows the execution time for all three regularization implementations over a single registration iteration as a function of control-point grid spacing with the volume size held constant at $320 \times 320 \times 320$ voxels. As expected, the execution time for the numerical implementation is agnostic to the control-point spacing since it performs central differencing



(a) (b)

Figure 5.10: Regularization performance with respet to tile size. (a) Execution times for each regularization implementation as a function of tile size. For these tests, only the time to process a single tile is measured. (b) Only showing analytic implementations.

at every point in the vector field. Fig. 5.9(b) shows only the execution times for the analytically derived solutions. Interestingly, the analytic implementations exhibit inverse cubic decay in execution time with respect to control-point spacing, with finer spacings showing longer execution times. This character is most easily explained by examining the execution time required to compute the smoothness for a single tile with respect to tile size as shown in Fig. 5.10. As shown in Fig. 5.10(b), as the tile size increases, the time required for the analytic implementations to process that tile remains constant. This is because the computation operation is based only on the coefficients of the 64 control-points which define the tile. If the spacing between these 64 control-points increase, the number of elements required to perform the computation remains unchanged – only the B-spline normalization matrices $\mathbf{R}_x$, $\mathbf{R}_y$, and $\mathbf{R}_z$ from (5.10) are modified, which has no affect on processing time

as shown in Fig. 5.10(b). Consequently, the inverse cubic execution profile exhibited by the analytic algorithms in Fig. 5.9(b) is due only to the number of tiles decreasing in each of the three spatial volume dimensions as the control-point spacing is isometrically increased. Finally, Fig. 5.10(a) shows the execution time to increase with tile size for the central differencing based numerical algorithm. However, because this algorithm's processing time is solely dependent on the total number of voxels in the image volume, the overall execution time remains unchanged with control-point spacing as shown in Fig. 5.9(a). In other words, if the volume holds constant dimensionality, increasing the control-point spacing results in longer processing times per tile, however the number of tiles have proportionally decreased; thus resulting in an overall unchanged processing time for the image volume as a whole.

5.4 Conclusions

We have developed an analytic method for computing the smoothness of a vector deformation field parameterized by uniform cubic B-spline basis coefficients. Furthermore, we have demonstrated how to integrate this smoothness metric into the deformable registration workflow; thereby constraining the solution of the vector deformation field and regularizing the ill-posedness of the image registration problem. The effectiveness of this method of regularization has been validated by performing multi-modal MI-based deformable registration of a pre-operational thoracic MRI to an intra-operational thoracic CT scan. The warped images, fused images, and vector field visualizations show increased anatomical correctness for registration procedures incorporating regularization over otherwise identical procedures excluding regularization. Finally, performance analysis shows our analytic method for computing the smoothness metric to be independent of volume resolution and 191 times faster

than the traditional numerical method of computation based on central differencing – reducing the operation from hundreds of seconds to approximately 1 second for most registration configurations. By parallelizing the analytic algorithm via OpenMP, we achieved a speed up of over 3.2 times the single core analytic algorithm when executed on a four core Intel i7 920 processor; thus providing sub-second processing times for even the most demanding medical image registration problems.

CHAPTER 6: CONCLUSIONS

The ideas and techniques presented in this thesis form a fast and robust basis for deploying both unimodal and multi-modal medical image registration on multi-core computer architectures.

In Chapter 2, the grid-alignment scheme and its associated data structures were introduced. These data structures greatly reduce the complexity of the B-spline registration process and form a strong basis that is used extensively by the techniques presented throughout this thesis. Highly parallel and scalable designs for computing both the unimodal MSE similarity metric and its derivative with respect to the B-spline parameterization were developed for multi-core architectures. The speed and robustness of the proposed parallelization strategy was experimentally demonstrated using both synthetic and clinical CT data. For multi-core CPU systems, the acceleration over a highly optimized serial implementation was shown to be linearly proportional to the number of available cores up to the tested number of eight cores. The GPU implementation, when executed on a Tesla 1060 GPU, exhibited an acceleration of 15 times over the serial implementation. Experiments demonstrated a fairly strong independence between the B-spline grid resolution and the execution time for both multi-core CPU and GPU-based implementations. Additionally, the presented method for computing the MSE cost function in parallel formed a solid foundation for developing highly data-parallel methods for both MI-based multi-modal registration and deformation vector

field regularization by providing a fast and thread-safe method of working with B-spline parameterization.

In Chapter 3, the integration of the MI cost function with the B-spline registration algorithm was introduced; thereby enabling multi-modal registrations such as MRI to CT. The mathematical foundation of MI-based scoring was detailed, and the fundamental theory describing how such statistically based similarity scoring differs from the simpler intensity based MSE scoring was introduced. Furthermore, parallel methods for computing the following were developed mathematically and demonstrated algorithmically in pseudo-code: deformation vector field expansion from a sparse B-spline parameterization, generation of marginal and joint intensity histograms, partial volume interpolation, MI computation, and the computation of the MI derivative with respect to the B-spline parameterization. The relationship between these operations was presented within the framework of an iterative optimization workflow that constitutes the multi-modal registration process. Single-core CPU, multi-core CPU, and many-core GPU implementations based on the presented theory were developed and evaluated in terms of execution speed and registration quality. Results indicated that the speedup varied linearly with volume size and was relatively insensitive to the B-spline control-point spacing. The GPU-based implementations achieved, on average, a speedup of 21 times with respect to the serial CPU implementation and 7.5 times with respect to the multi-core CPU implementation when executed using four cores. All algorithms exhibited near-identical registration quality.

Chapter 4 developed a fast method for increasing the accuracy of estimating the probability distribution of image intensities by introducing V-opt histograms. It was demonstrated that such histograms improve the quality of MI-based image registration by auto-

matically placing each distinct tissue type in the image within segregated bins. Experimental results indicated that for the same number of bins, the V-opt histograms resulted in better registration accuracy for both MRI to CT and PET to CT registrations when compared to histograms that use the traditional method of uniform bin spacing.

Finally, Chapter 5 developed an analytic method for computing the smoothness of the vector deformation field by operating directly on the B-spline parameterization coefficients. Additionally, it was demonstrated how to integrate this smoothness metric into the deformable registration workflow; thereby constraining the solution of the vector deformation field and regularizing the ill-posedness of the registration problem. The effectiveness of this method was validated by performing MI-based deformable registration of a pre-operational thoracic MRI to an intra-operational thoracic CT scan. Warped images, fused images, and vector field visualizations were presented and indicated increased anatomical correctness for registration procedures incorporating regularization over otherwise identical procedures excluding regularization. Performance analysis showed the analytic method for computing the smoothness metric to be independent of volume resolution and 191 times faster than the traditional numerical computation based on central differencing – reducing the operation from hundreds of seconds to approximately 1 second for most clinical applications. By parallelizing the analytic algorithm via OpenMP, we achieved a further speedup of 3.2 times over the single core analytic algorithm when executed on a four core Intel i7 920 processor; thus providing sub-second processing times for even the most demanding medical image registration problems.

The topics and methods presented in this thesis have been implemented and comprise the B-spline registration engine used by the high-performance medical image registration

software package Plastimatch, which is freely available for download under a BSD-style license from www.plastimatch.org. With markedly improved execution speeds and integration with modern GPU processing platforms, it is hoped that deformable registration based medical image analysis will become more commonly adopted into routine clinical practices.

LIST OF REFERENCES

[1] P. Freeborough and N. Fox, "Modeling brain deformations in Alzheimer disease by fluid registration of serial 3D MR images," *J Comput Assist Tomogr*, vol. 22, 1998. 2

[2] P. Thompson, M. Mega, R. Woods, C. Zoumalan, C. Lindshield, R. Blanton, J. Moussai, C. Holmes, J. Cummings, and A. Toga, "Cortical change in Alzheimer's disease detected with a disease-specific population-based brain atlas," *Cerebral Cortex*, vol. 11, pp. 1–16, Jan 2001. 2

[3] R. Scahill, C. Frost, R. Jenkins, J. Whitwell, M. Rossor, and N. Fox, "A longitudinal study of brain volume changes in normal aging using serial registered magnetic resonance imaging," *Arch Neurol*, vol. 60, pp. 989–94, Jul 2003. 2

[4] W. Gharaibeh, F. Rohlf, D. Slice, and L. DeLisi, "A geometric morphometric assessment of change in midline brain structural shape following a first episode of schizophrenia," *Biol Psychiatry*, vol. 48, pp. 398–405, 2000. 2

[5] D. Job, H. Whalley, S. McConnell, M. Glabus, E. Johnstone, and S. Lawrie, "Voxel-based morphometry of grey matter densities in subjects at high risk of schizophrenia.," *Schizophr Res*, vol. 64, pp. 1–13, 2003. 2

[6] P. Thompson, J. Giedd, R. Woods, D. MacDonald, A. Evans, and A. Toga, "Growth patterns in the developing human brain detected using continuum-mechanical tensor mapping," *Nature*, vol. 404, pp. 190–3, 2000. 2

[7] A. Gholipour, N. Kehtarnavaz, R. Briggs, M. Devous, and K. Gopinath, "Brain functional localization: A survey of image registration techniques," *IEEE Trans Med Imaging*, vol. 26, pp. 427–451, Apr 2007. 2

[8] M. Ferrant, A. Nabavi, B. Macq, P. Black, F. Jolesz, R. Kikinis, and S. Warfield, "Serial registration of intraoperative MR images of the brain," *Med Image Anal*, vol. 6, pp. 337–59, Dec 2002. 3

[9] T. Hartkens, D. Hill, A. Castellano-Smith, D. Hawkes, C. Maurer, A. Martin, W. Hall, H. Liu, and C. Truwit, "Measurement and analysis of brain deformation during neurosurgery," *IEEE Trans Med Imaging*, vol. 22, pp. 82–92, 2003. 3

[10] A. Bharatha, M. Hirose, N. Hata, S. Warfield, M. Ferrant, K. Zou, E. Suarez-Santana, J. Ruiz-Alzola, A. D'Amico, R. Cormack, R. Kikinis, F. Jolesz, and C. Tempany, "Evaluation of three-dimensional finite element-based deformable registration of pre- and intraoperative prostate imaging," *Med Phys*, vol. 28, pp. 2551–60, Dec 2001. 3

[11] A. Mohamed, C. Davatzikos, and R. Taylor, "A combined statistical and biomechanical model for estimation of intra-operative prostate deformation," in *Proc MICCAI*, pp. 452–60, 2002. 3

[12] D. Stoyanov, G. Mylonas, F. Deligianni, A. Darzi, and G. Yang, "Soft-tissue motion tracking and structure estimation for robotic assisted MIS procedures," in *Proc MICCAI*, pp. 139–46, 2005. 3

[13] T. Lange, S. Eulenstein, M. Hünerbein, and P. Schlag, "Vessel-based non-rigid registration of MR/CT and 3D ultrasound for navigation in liver surgery," *Comput Aided Surg*, vol. 8, no. 5, pp. 228–40, 2003. 3

[14] E. Boctor, M. de Oliveira, M. Choti, R. Ghanem, R. Taylor, G. Hager, and G. Fichtinger, "Ultrasound monitoring of tissue ablation via deformation model and shape priors," in *Proc MICCAI*, pp. 405–12, 2006. 3

[15] W. Lu, M. Chen, G. Olivera, K. Ruchala, and T. Mackie, "Fast free-form deformable registration via calculus of variations," *Phys Med Biol*, vol. 49, no. 14, pp. 3067–87, 2004. 3

[16] H. Wang, L. Dong, J. O'Daniel, R. Mohan, A. Garden, K. Ang, D. Kuban, M. Bonnen, J. Chang, and R. Cheung, "Validation of an accelerated 'demons' algorithm for deformable image registration in radiation therapy," *Phys Med Biol*, vol. 50, pp. 2887–2905, 2005. 3, 4

[17] S. Flampouri, S. Jiang, G. Sharp, J. Wolfgang, A. Patel, and N. Choi, "Estimation of the delivered patient dose in lung IMRT treatment based on deformable registration of 4D-CT data and Monte Carlo simulations," *Phys Med Biol*, vol. 51, no. 11, pp. 2763–79, 2006. 3

[18] K. Brock, J. Balter, L. Dawson, M. Kessler, and C. Meyer, "Automated generation of a four-dimensional model of the liver using warping and mutual information," *Med Phys*, vol. 30, pp. 1128–33, 2003. 3

[19] T. Rohlfing, C. Maurer, W. O'Dell, and J. Zhong, "Modeling liver motion and deformation during the respiratory cycle using intensity-based nonrigid registration of gated MR images," *Med Phys*, vol. 31, pp. 427–32, 2004. 3

[20] E. Rietzel, G. Chen, N. Choi, and C. Willet, "Four-dimensional image-based treatment planning: Target volume segmentation and dose calculation in the presence of respiratory motion," *Int J Radiat Oncol Biol Phys*, vol. 61, no. 5, pp. 1535–50, 2005. 3

[21] M. Foskey, B. Davis, L. Goyal, S. Chang, E. Chaney, N. Strehl, S. Tomei, J. Rosenman, and S. Joshi, "Large deformation three-dimensional image registration in image-guided radiation therapy," *Phys Med Biol*, vol. 50, p. 24, 2005. 3

[22] T. Zhang, Y. Chi, E. Meldolesi, and D. Yan, "Automatic delineation of on-line head-and-neck computed tomography images: Toward on-line adaptive radiotherapy," *Int J Radiat Oncol Biol Phys*, vol. 68, no. 2, pp. 522–30, 2007. 3

[23] K. Brock, L. Dawson, M. Sharpe, D. Moseley, and D. Jaffray, "Feasibility of a novel deformable image registration technique to facilitate classification, targeting, and monitoring of tumor and normal tissue," *Int J Radiat Oncol Biol Phys*, vol. 64, no. 4, pp. 1245–54, 2006. 3

[24] J. R. McClelland, J. M. Blackall, S. Tarte, A. C. Chandler, S. Hughes, S. Ahmad, D. B. Landau, and D. J. Hawkes, "A continuous 4D motion model from multiple respiratory cycles for use in lung radiotherapy," *Medical Physics*, vol. 33, no. 9, p. 3348, 2006. 3

[25] C. Rohkohl, G. Lauritsch, L. Biller, M. Pr
"ummer, J. Boese, and J. Hornegger, "Interventional 4-D motion estimation and reconstruction of cardiac vasculature without motion periodicity assumption," *Medical Image Analysis*, 2010. 3

[26] T. Brunet, K. Nowak, and M. Gleicher, "Integrating dynamic deformations into interactive volume visualization," in *Eurographics/IEEE VGTC Symposium on Visualization 2006*, pp. 219–226, Citeseer, 2006. 3

[27] J. Maintz and M. Viergever, "A survey of medical image registration," *Med Image Anal*, vol. 2, no. 1, pp. 1–37, 1998. 4

[28] B. Zitová and J. Flusser, "Image registration methods: A survey," *Image Vis Comput*, vol. 21, pp. 977–1000, 2003. 4

[29] W. Crum, T. Hartkens, and D. Hill, "Non-rigid image registration: theory and practice," *The British Journal of Radiology*, vol. 77, pp. S140–S153, 2004. 4

[30] G. Sharp, M. Peroni, R. Li, J. Shackleford, and N. Kandasamy, "Evaluation of plastimatch b-spline registration on the empire10 data set," *Medical Image Analysis for the Clinic: A Grand Challenge*, pp. 99–108, 2010. 4

[31] P. Thompson and A. Toga, "A surface-based technique for warping three-dimensional images of the brain," *IEEE Trans Med Imaging*, vol. 15, no. 4, pp. 402–417, 1996. 4

[32] J. Thirion, "Image matching as a diffusion process: An analogy with Maxwell's demons," *Med Image Anal*, vol. 2, no. 3, pp. 243–260, 1998. 4

[33] G. Christensen, R. Rabbitt, and M. Miller, "Deformable templates using large deformation kinematics," *IEEE Trans Image Proc*, vol. 5, no. 10, pp. 1435–1447, 1996. 4

[34] F. Bookstein, "Principal warps: thin-plate splines and the decomposition of deformations," *IEEE Trans Pattern Anal Mach Intell*, vol. 11, no. 6, pp. 567–85, 1989. 4

[35] F. Bookstein and D. Green, "A feature space for derivatives of deformations," in *IPMI, LNCS*, vol. 687, pp. 1–16, 1993. 4

[36] D. Metaxas, *Physics-Based Deformable Models: Applications to Computer Vision, Graphics and Medical Imaging*. Kluwer Academic Publishers, 1997. 4

[37] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. Hill, M. O. Leach, and D. J. Hawkes, "Nonrigid registration using free-form deformations: application to breast MR images," *IEEE Trans Med Imaging*, vol. 18, no. 8, pp. 712–21., 1999. 4

[38] R. Frackowiak, K. Friston, C. Frith, R. Dolan, and J. Mazziotta, eds., *Human Brain Function*. Academic Press USA, 1997. 4

[39] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second ed., 2005. 4

[40] R. Woods, S. Cherry, and J. Mazziotta, "Rapid automated algorithm for aligning and reslicing PET images," *Journal of Computer Assisted Tomography*, vol. 16, pp. 620–633, 1992. 4

[41] B. Fischl, A. Liu, and A. Dale, "Automated manifold surgery: Constructing geometrically accurate and topologically correct models of the human cerebral cortex," *IEEE Trans Med Imaging*, vol. 20, no. 1, pp. 70–80, 2001. 4

[42] T. Hartkens, *Measuring, analysing, and visualizing brain deformation using non-rigid registration*. PhD thesis, King's College London, 1993. 4

[43] G. Rohde, A. Aldroubi, and B. Dawant, "The adaptive bases algorithm for intensity based nonrigid image registration," *IEEE Trans Med Imaging*, vol. 22, pp. 1470–1479, 2003. 4

[44] S. Aylward, J. Jomier, S. Barre, B. Davis, and L. Ibanez, "Optimizing ITK's registration methods for multi-processor, shared-memory systems," in *MICCAI Open Source and Open Data Workshop*, 2007. 4

[45] S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. McInnes, B. Smith, and H. Zhang, "PETSc Web page," 2001. http://www.mcs.anl.gov/petsc. 5

[46] S. Warfield, M. Ferrant, X. Gallez, A. Nabavi, F. Jolesz, and R. Kikinis, "Real-time biomechanical simulation of volumetric brain deformation for image guided neurosurgery," in *Proceedings of SC2000*, 2000. 5

[47] S. Warfield, S. Haker, I. Talos, C. Kemper, N. Weisenfeld, A. Mewes, D. Goldberg-Zimring, K. Zou, C. Westin, W. Wells, C. Tempany, A. Golby, P. Black, F. Jolesz, and R. Kikinis, "Capturing intraoperative deformations: Research experience at Brigham and Women's hospital," *Medical Image Analysis*, 2005. 5

[48] M. Sermesant, O. Clatz, Z. Li, S. Lanteri, H. Delingette, and H. Ayache, "A parallel implementation of non-rigid registration using a volumetric biomechanical model," in *WBIR'03*, pp. 398–407, Springer-Verlag, 2003. 5

[49] D. Pham *et al.*, "The design and implementation of a first-generation cell processor," *ISSCC Dig. Tech. Papers*, pp. 184–185, 2005. 5

[50] G. Sharp, R. Li, J. Wolfgang, G. Chen, M. Peroni, M. Spadea, S. Mori, J. Zhang, J. Shackleford, and N. Kandasamy, "Plastimatch-an open source software suite for radiotherapy image processing," in *Proceedings of the XVIth International Conference on the use of Computers in Radiotherapy (ICCR), Amsterdam, Netherlands*, 2010. 7

[51] J. Shackleford, N. Kandasamy, and G. Sharp, *GPU Computing Gems Emerald Edition, Chapter 47*. Morgan Kaufmann Publishers Inc., 2011. 11

[52] J. Shackleford, N. Kandasamy, and G. Sharp, "On developing b-spline registration algorithms for multi-core processors," *Physics in Medicine and Biology*, vol. 55, p. 6329, 2010. 11

[53] J. Kybic and M. Unser, "Fast parametric elastic image registration," *IEEE Trans. Medical Imaging*, vol. 12, no. 11, pp. 1427–1442, 2003. 17

[54] C. Zhu, R. Byrd, and J. Nocedal, "L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization," *ACM Transactions on Mathematical Software*, vol. 23, no. 4, pp. 550–60, 1997. 19

[55] P. Thevenaz and M. Unser, "Optimization of mutual information for multiresolution image registration," *IEEE Trans Image Proc*, vol. 9, pp. 2083–2099, Dec 2000. 41, 49, 77

[56] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Seutens, "Multimodality image registration by maximization of mutual information," *IEEE Transactions on Medical Imaging*, vol. 16, pp. 187–198, April 1997. 43, 50

[57] D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010. 54

[58] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007. 70

[59] B. Chapman, G. Jost, and R. V. D. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007. 70

[60] J. Pluim, J. Maintz, and M. Viergever, "Mutual-information-based registration of medical images: A survey," *IEEE Trans. Med. Imaging*, vol. 22, no. 8, pp. 986–1004, 2003. 78

[61] Y.-M. Zhu and S. Cochoff, "Influence of implementation parameters on registration of MR and SPECT brain images by maximization of mutual information," *Journal Nuclear Medicine*, vol. 43, no. 2, pp. 160–66, 2002. 79

[62] R. Shekhar and V. Zagrodsky, "Mutual information-based rigid and nonrigid registration of ultrasound volumes," *IEEE Trans. Med. Imaging*, vol. 21, no. 1, pp. 9–22, 2002. 79

[63] H. Jagadish *et al.*, "Optimal histograms with quality guarantees," *Proc. Very Large Database Conf.*, 1998. 79

VITA

James Anthony Shackleford was born in Memphis, TN in 1983. James earned a Bachlors of Electrical Engineering from Drexel University in 2006. The Senior Design Project he completed in partial fulfillment of the degree entitled *Design of a CMOS IC Nanowire Sensor Array* received the distinguished honor of first place in the Department of Electrical Engineering's Senior Design Competition. Upon completion of his undergraduate studies, James continued to attend Drexel in persuit of a Masters of Engineering degree in solid state device physics under the advisement of Dr. Bahram Nabet. While earning this degree, for the years 2006–2007, James received the honor of distinguished GAANN Fellow. The work earning his Masters Degree was published on February 2009 in Applied Physics Letters, Volume 94, Issue 8 under the title *Integrated Plasmonic Lens Photodetector*. Upon completion of his Master's degree, James continued to attend Drexel University in persuit of a Doctorate of Philosophy degree in Computer Engineering under the advisement of Dr. Nagarajan Kandasamy. During this time, his work on accelerating deformable image registration was published as both a featured article in Physics in Medicine and Biology, vol. 55 as well as a chapter in Morgan Kaufmann publication GPU Computing Gems: Emerald Edition. During his time as a graduate student, James has taught lab sections for courses in analog electronics, solid state devices, microcontrollers, Freshman design, Senior design, and advanced electronics. For the 2009 and 2010 summer quarters, James taught Embedded Systems as an Adjunct Processor for Drexel University.