Fast 3D Deformable Image Registration on a GPU Computing Platform

FAST 3D DEFORMABLE IMAGE REGISTRATION ON A GPU COMPUTING PLATFORM

BY

MOHAMMAD HAMED MOUSAZADEH, B.Sc.

A THESIS

SUBMITTED TO THE SCHOOL OF BIOMEDICAL ENGINEERING

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

Master of Applied Science (2011) McMaster University

(Biomedical Engineering) Hamilton, Ontario, Canada

TITLE:	Fast 3D Deformable Image Registration on a GPU Computing Platform
AUTHOR:	Mohammad Hamed Mousazadeh B.Sc., (Biomedical Engineering-bioelectrical) Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran
SUPERVISOR:	Dr. Shahin Sirouspour
NUMBER OF PAGES:	xi, 103

To my beloved family and Saba

In loving memory of my dear father

Abstract

Image registration has become an indispensable tool in medical diagnosis and intervention. The increasing need for speed and accuracy in clinical applications have motivated researchers to focus on developing fast and reliable registration algorithms. In particular, advanced deformable registration routines are emerging for medical applications involving soft-tissue organs such as brain, breast, kidney, liver, prostate, etc. Computational complexity of such algorithms are significantly higher than those of conventional rigid and affine methods, leading to substantial increases in execution time.

In this thesis, we present a parallel implementation of a newly developed deformable image registration algorithm by Marami et al. [1] using the Computer Unified Device Architecture (CUDA). The focus of this study is on acceleration of the computations on a Graphics Processing Unit (GPU) to reduce the execution time to nearly real-time for diagnostic and interventional applications. The algorithm co-registers preoperative and intraoperative 3-dimensional magnetic resonance (MR) images of a deforming organ. It employs a linear elastic dynamic finite-element model of the deformation and distance measures such as mutual information and sum of squared difference to align volumetric image data sets. In this study, we report a parallel implementation of the algorithm for 3D-3D MR

registration based on SSD on a CUDA capable NVIDIA GTX 480 GPU. Computationally expensive tasks such as interpolation, displacement and force calculation are significantly accelerated using the GPU. The result of the experiments carried out with a realistic breast phantom tissue shows a 37-fold speedup for the GPU-based implementation compared with an optimized CPU-based implementation in high resolution MR image registration. The CPU is a 3.20 GHz Intel core i5 650 processor with 4GB RAM that also hosts the GTX 480 GPU. This GPU has 15 streaming multiprocessors, each with 32 streaming processors, i.e. a total of 480 cores. The GPU implementation registers 3D-3D high resolution ($512\times512\times136$) image sets in just over 2 seconds, compared to 1.38 and 23.25 minutes for CPU and MATLAB-based implementations, respectively. Most GPU kernels which are employed in 3D-3D registration algorithm also can be employed to accelerate the 2D-3D registration algorithm in [1].

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Dr. Shahin Sirouspour for his invaluable comments, patience, support and encouragement through my masters at McMaster University. I wish to express my warm thanks to Dr. Alexandru Patriciu for his constructive comments and suggestions during the GPU parallel implementation in this study. I warmly thank Dr. Micheal Noseworthy for his tremendous support and help for providing us with access to MRI machine. In addition I would like to thank my committee members, Dr. Shahin Sirouspour, Dr. Alexandru Patriciu, and Dr. Micheal Noseworthy for their interest in my work.

My warmest gratitude goes to my parents and brothers for their encouragement and endless support through my life.

I am indebted to my friends and lab mates, Ramin Mafi and Pawel Malysz for all the fun we have had in the last two years. In addition, I would like to thank Bahram Marami, my friend and lab mate who was involved in the medical image registration project, for his invaluable and supportive comments.

Last but not the least, I would like to express my deepest gratitude to Saba Mohtashami for her kindness, help and support. Without her help, this study would not have been completed.

Contents

List of Figures

Chapter 1

Introduction

Image registration is the process of aligning multiple images of the same subject to establish correspondence between their features. It has a wide variety of applications in weather forecasting, medical diagnoses and treatments, computer vision, etc. In this thesis, we will focus on medical applications of image registration. Medical image registration plays a significant role in clinical interventions such as biopsy, image-guided surgery and radiotherapy planing. In many medical procedures, a plan of action is constructed based on preoperative medical images obtained from the patient. Such plans often have to be updated in real-time to account for the changes that might have occurred since the acquisition of the preoperative images. A surgeon can take advantage of a fast and reliable registration algorithm to update the surgical plan during the operation to compensate for patient and tissue movements based on the information obtained from intraoperative images.

Medical image registration can be employed in diagnoses and treatment of breast cancer which is the most common type of cancer in women worldwide.

In this context, X-ray, Magnetic Resonance Imaging (MRI) and Ultrasound (US) are the primary focuses of interest for both single-modality and multi-modality registration. While X-ray only provides a 2-D projection of breast anatomy, MRI reveals more information about the shape, size and spatial relationships. In MRI, a contrast agent such as gadolinium DTPA (diethylenetriamine pentaacetic acid) is injected to enhance the signal intensity of vascular structures and suspicious areas. Fatty and connective tissue do not enhance after the injection. By registering pre- and post-contrast MR images, valuable information about the lesions are obtained [2]. The output of a registration algorithm that employs image sets at different time points can reveal anatomical difference (e.g., brain tumor growth), therefore yielding more information regarding the disease progression [3].

Registration algorithms can be categorized into rigid, affine and non-rigid methods. Non-rigid registration, which includes deformable registration, can accommodate both rigid and non-rigid tissue transformation, hence is suitable for applications involving biological soft tissue. This thesis is concerned with deformable image registration. The goal of registration is to find an optimal geometric transformation to maximize similarity among the images. The simplest transformation is a rigid transformation which only has six degree of freedom(DoF), three rotations and three translations. An affine formation has twelve DoFs adds scaling along different coordinate axes and shearing in different planes to translation and rotation. Rigid and affine registration methods are only suitable for non-deformable tissue such as bone; Non-rigid transformations must be employed when dealing with soft tissue such as breast, liver, brain, prostate and kidney.

Registration methods provide visual comparison of images taken at different

Figure 1.1: An Image-guided system for brain surgery with an optical tracker to monitor the position of the patient and surgical instruments (Photo courtesy of Dr.Richard Bucholz, St.Louis University School of Medicine).

times. The time difference can range from seconds (e.g., cardiac motion) to years (e.g., brain tumor growth). Fig. 1.1 illustrates an image-guided system for brain surgery. An optical tracker is used to estimate the current location of the target (e.g. brain). The images in the monitor are preoperative images and the objective is to identify the location of a surgical instruments relative to the patient's tissue. Once a surgical instrument (e.g. a needle) inserts into an organ, the location of the internal tissue (e.g. a tumor tissue) changes. An image-guided system which is also capable of performing image registration can provide real-time (or near real-time) information about the new location of the internal tissue with respect to the surgical instrument by updating the preoperative images.

Fig. 1.2 illustrates three main components of a generic deformable algorithm. In general, a deformable registration algorithm consists of the following elements [4]:

- **Transformation Model**: The transformation model identifies the geometric transformation between images. Deformable transformations are classified

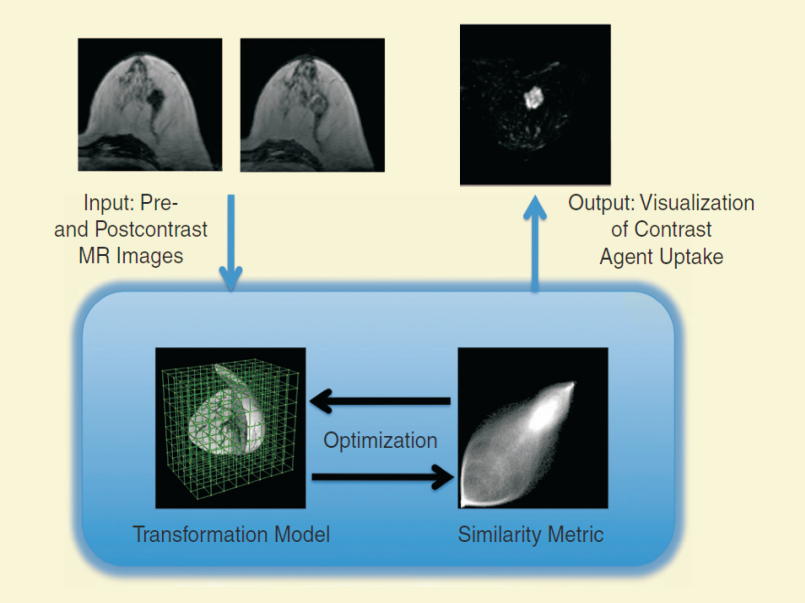


Figure 1.2: An algorithm for nonrigid image registration involving three components: The transformation model, similarity metric and optimizer (Photo courtesy of Dr. Daniel Rueckert [4]).

as *non-physics-based*, e.g. B-Spline and Moving Least Square (MLS) or *physics-based*, e.g. continuum mechanics deformation models. In this thesis, the focus will be on physics-based models.

- **Similarity/Distance Metric**: A similarity metric measures the degree of alignment among the images to be co-registered. Some methods use the alignment of features such as landmarks and edges to measure the similarity [5]. Other algorithms employ image intensity-based measures such as mutual information or image correlation [6].

- **Optimization**: Deformable registration can be formulated as an optimization problem in which the goal is to find a deformation that would maximize(minimize) the similarity(distance) measure among the images. There

are usually some constraints that regularize this optimization problem and would ensure that the resulting deformation is "reasonable".

- **Validation**: In order to have an accurate and robust algorithm, a careful validation is required prior to clinical use.

1.1 Problem Statement

Recently, a new deformable image registration algorithm that employs a dynamic linear elastic deformation model in conjunction with image similarity measure such as sum of squared difference (SSD), mutual information (MI), and correlation ratio (CR) has been developed by Marami et al [1] . The algorithm involves computationally expensive tasks such as trilinear interpolation, solving a system of second order differential equations, finding the 3D deformation using linear shape function on tetrahedral finite elements, and solving a large linear system of equations based on the conjugate gradient method. The improved accuracy and robustness of this method over conventional rigid and affine registration techniques are gained at the expense of its computation time. In addition to accuracy and robustness, speed is a critical factor in many clinical applications such as biopsy, image-guided surgery and radiotherapy in which real-time or near real-time performance is expected. In this thesis, we design, optimize and evaluate a highly parallel GPU-based implementation of the deformable registration method by Marami et al. [1] in order to significantly reduce its computation time.

Recently Graphics Processing Units (GPUs), Field-Programmable-Gate-Array (FPGA) devices and multiprocessor systems have been used to accelerate image

registration algorithms. In [7], multiprocessor systems have been employed to speed up image registration. Maintenance and acquisition costs are the main drawbacks of such systems [8].

FPGA-based computing architectures are highly customizable and therefore can significantly speed up computations, if properly designed. A study of rigid registration using FPGAs is presented in [9]. GPUs are less expensive, and easier to program than FPGAs, making them widely popular in scientific computing applications in recent years. Near real-time registration can be achieved with recent advances in GPU technology [4, 10].

The gaming industry demand for sophisticated graphics has spurred great advancement in the GPU technology. Today, standard GPUs in personal computers have higher computational performance and memory bandwidth than their host CPUs [11]. In fact, there is a large performance gap between the GPUs and CPUs due to fundamental difference in their design. CPUs have larger control logic unit and cache memories whereas GPUs dedicate more chip areas to floating point calculations , i.e. see Fig. 1.3. The massively parallel GPU architecture is organized into highly-threaded multiprocessors that can run thousands of threads per application.

It should be noted that GPUs are not good in every tasks, especially those on which CPUs are designed to perform well. Generally, CPUs can provide better performance in sequential tasks that require a serial implementation. The performance can be significantly improved by GPUs in tasks which can be implemented in parallel (e.g. Matrix by Matrix Multiplication). A good knowledge of the strengths and weaknesses of GPUs and CPUs in performing parallel and

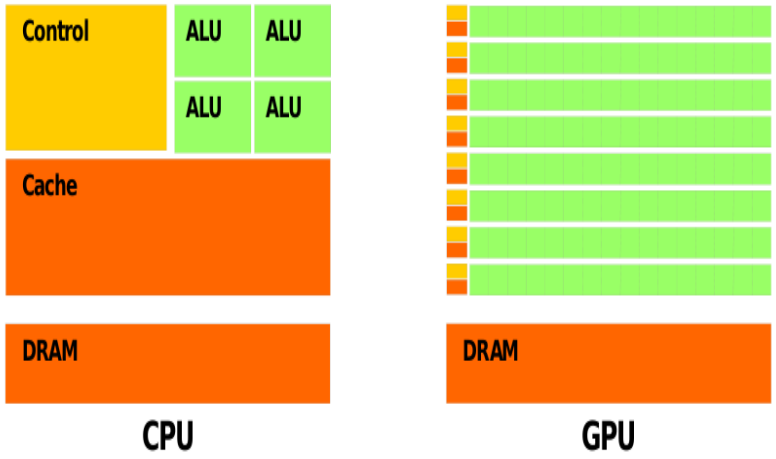


Figure 1.3: The differences between CPU and GPU architecture design. CPUs have larger control logic unit and cache memories whereas GPUs dedicate more chip areas to floating point calculations; From NVIDIA's document [13]

serial tasks is instrumental in achieving high performance in computing applications. A significant acceleration of computations is possible through a proper utilization of both CPU and GPU resources. Compute Unified Device Architecture (CUDA) is a programming model which supports joint CPU/GPU execution (NVIDIA, 2007) [12].

As mentioned before, GPUs are capable of running thousands of threads at a same time (parallel to each other), therefore one can take advantage of this feature to accelerate the non-sequential tasks. Image registration algorithms are a great candidate for parallel implementation on GPUs. For instance, GPUs excel in 3D texture mapping which can be used in trilinear interpolation, one of the most time consuming elements of registration algorithms. This and other capabilities of GPUs in the context of parallel computing for image registration will be discussed in details throughout the thesis.

1.2 Thesis Contributions and Outline

CUDA provides a powerful and user-friendly programming environment for GPU-based scientific computing applications including those in medical imaging. In this work, we report an implementation of the algorithm by Marami et al. [1] for 3D-3D MR deformable image registration based on SSD on a CUDA capable NVIDIA GTX 480 GPU. We have carried out experiments with a realistic breast phantom in order to volumetrically register high-resolution MR images on a single GPU. The results of these experiments show a 37-fold speedup for the GPU-based implementation compared with an optimized CPU-based implementation (written in C) in high resolution MR image registration. The GPU implementation is capable of registering 512 x 512 x 136 image sets in just over 2 seconds, making it suitable for clinical applications requiring fast and accurate processing of medical images. Furthermore, we briefly discuss how GPUs can accelerate the 2D-3D version of Marami's image registration algorithm. In summary the main contributions of the thesis are:

- CUDA implementation and optimization of trilinear interpolation of the image and its directional gradients using 3D texture memory. We have taken advantage of spacial locality feature of texture memory, designed for graphics applications, to perform trilinear interpolation. Assigning threads to the 3D regular grid points (one thread for each point), their grey value are measured independently.

- CUDA implementation, verification and optimization of force computation. A system of equations in the form of $Af=b$ is solved using the least squares

and conjugate gradient method to compute the vector of nodal forces based on the difference between the template and reference image using the SSD metric. The main computationally intensive elements of the conjugate gradient algorithm such as sparse matrix by vector multiplication and dot products are accelerated in this step.

- CUDA implementation, verification and optimization of dynamic finite element linear elastic deformation model. The Newmark method has been used for numerical integration. Time consuming elements such as matrix by vector multiplications including both dense and sparse, and vector addition are accelerated in this part.

- CUDA implementation, verification and optimization of the linear shape function of the tetrahedral elements in order to find the displacement of a finer 3D grids inside the image volume. Assigning threads to the 3D regular points (one thread for each point), their displacements are calculated independently by designing a parallel approach.

- Memory optimization for GPU to CPU and CPU to GPU data transfers and analysis of its influence on the performance of the algorithm.

- Implementation of GPU kernels to link different steps of the algorithms. Implementation of GPU kernels for other parts of the algorithm in which high performance can be achieved with parallel computing(e.g., finding the coefficients of each element's shape function).

The rest of this thesis is organized as follow. In Chapter 2, we briefly review registration algorithms with emphasizes on parallel implementation for acceleration

of computations. Chapter 3 presents the mathematical formulation of the registration algorithm in [1]. This chapter illustrates how the dynamic finite element linear elastic deformation model is employed in the registration method. Chapter 4 reviews the GPU hardware capabilities and introduces CUDA programming environment for joint CPU/GPU execution. This chapter clarifies how each part of the algorithm is accelerated using parallel implementations. Chapter 5 is dedicated to our experimental results using a realistic breast phantom (CIRS model 051). The GPU performance for different parts of the algorithm is compared with those from C and the runtime of the whole algorithm is compared with both C and Matlab execution times. The thesis is summarized in Chapter 6 with a discussion of possible future extensions of the work.

1.3 Related Publication

H. Mousazadeh, B. Marami, S. Sirouspour and A. Patriciu, " GPU Implementation of a Deformable 3D Image Registration Algorithm", Accepted for presentation at *33rd International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 11)*, Boston, MA, USA, pp.4897–4900, 2011.

Chapter 2

Literature Review

Over the last decade, image registration has played a significant role in medical image computing. The goal of image registration is to establish special correspondence between different images taken of the same patient at different times and/or using different modalities. Registration algorithms always involve a trade-off between accuracy and speed.

Registration algorithm can be classified according to the following three criteria: **Dimensionality**, **Modality**, and **Transformation**.

2.1 Image Dimension

In general, medical image registration methods can be divided into three groups based on their dimensionality; 2D-2D, 3D-3D, and 2D-3D. In 2D-2D registration, two-dimensional images of the same or different modalities are matched. An example is the registration of x-ray radiographs of the hand with Tc methyl-diphosphonate planar nuclear medicine images for the diagnosis of scaphoid injury. A 2D-2D

intensity-based image registration algorithm with application in endoscopic systems for image distortion correction has been developed in [14]. Two different black and white patterns, i.e. a chessboard and a concentric circles board, have been used to test the algorithm. In [15], an automated 2D-2D pixel-based registration method has been developed for patient set-up in radiotherapy.

In 3D-3D registration, volumetric images of the same or different modalities are compared. In [16], a 3D-3D registration methods has been developed for the localization of implanted subdural electrodes in planning epilepsy surgery. Anatomical fiducial markers have been employed for co-registration of volumetric pre-implant brain MRI and post-implant head CT. In [17], an automated algorithm has been developed to register volumetric MR brain images taken at different times.

In 2D-3D registration, correspondence is established between 2D and 3D images of the same or different modalities. Examples are registration of volumetric MR and 2D ultrasound and X-ray images [18]. In [19], a method for registering 2D X-ray images to 3D MRI has been developed. The study proposes a technique that generates pseudo-computed tomography from multi-spectral MRI acquisitions which enables 2D-3D registration of X-ray to MRI. The algorithm has been tested on *ex vivo* animal data.

2.2 Image Modality

Medical registration algorithm can employ data sets from the same modality (single-modality) or data sets from multiple modalities (multi-modality). An example of single-modality image registration can be found in [17] in which MR data sets have been used. In [16], a multi-modality registration algorithm has been developed for

co-registration of MRI and CT image sets.

2.3 Image Transformation

In general, medical image registration methods can be classified in three categories based on the transformation that is employed in the registration algorithm. They can employ rigid [20], affine [21], or non-rigid [22] (parametric or non-parametric) algorithms.

In general, the goal of image registration is to find a transformation $T : (x, y, z) \rightarrow (x', y', z')$ which maps any point in the template image into the corresponding point in the reference image. The simplest method is the rigid transformation:

$$T_{rigid}(x, y, z) = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

where t_x, t_y, t_z define the translations along axes of the coordinate systems, while r_{ij} are the result of the multiplication of three separate rotation matrices. In some cases, it may also be necessary to correct for scaling and shearing. An affine transformation combines rigid transformation with scaling and shearing matrices: $T_{affine}(x, y, z) = T_{shear}.T_{scale}.T_{rigid}.(x, y, z, 1)^T$

$$\text{where } T_{scale} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ and } T^{xy}_{shear} = \begin{pmatrix} 1 & 0 & h_x & 0 \\ 0 & 1 & h_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Rigid and affine registration methods are not applicable when the underlying tissue undergoes large deformations.

Rigid registration is suitable for rigid tissues such as pelvis, femur and might be also applicable to those tissues which their movement is constrained by a rigid tissue (e.g. brain motion which is constrained by skull). It should be noted that both rigid and non-rigid methods have been used for brain image registration. Non-rigid transformations, which include parametric and non-parametric models, can be used in the case where large deformation is expected, e.g. in soft tissues such as breast and liver.

Some nonrigid parametric registrations methods use a set of basis functions to present the deformation field. Bookstein [23] introduced the application of thin-plate splines for medical image registration. Registration methods that use splines assume that a set of landmarks points (referred as control points) can be identified in the template and reference images. Spline-based transformations are used to calculate displacements that map the location of the control points in the template image into their corresponding counterpart in the reference. The control points have a global influence on the transformation making it undesirable for cases in which local deformations are required. To solve this problem, control points must be distributed such that deformation is limited to desired regions [22].

An alternative approach for medical image registration based on free-from deformations (FFDs) has been proposed by Rueckert et al [24]. In spline-based FFDs, a 3D mesh of control points with uniform spacing is applied on the image domain. In such approaches, changing the control points only affects the transformation in the local neighborhood of those control points, therefore making them suitable for local deformations [18]. Simple changes may sometimes require adjustment of many control points, therefore precise modification of curves can be effortful. It should be noted that the complexity of these methods is increased by adding more landmarks as control points.

Thirion [25] proposed Demons, a deformable medical image registration algorithm. The algorithm is based on the optical flow method which can track small deformations in temporal sequences of the image volume; a displacement field is computed that would move the floating image so that it can match the static image. In this iterative method, the displacement vector is calculated based on the intensities of voxels in moving and static volumes and intensity gradient vector of the static image.

Physics-based image registration algorithms provide the greatest amount of flexibility compare to non-physics-based registration methods, however they are usually computationally expensive [3]. Mass-spring systems and finite element method (FEM) are two techniques that are widely employed to model deformable objects. A main advantage of models that use FEM over those that employ mass-spring systems is that they treat deformable objects as a continuum. In fact they provide more accuracy and reliability in modeling deformation of non-rigid objects [26].

2.4 Registration Validation

Registration algorithms should be validated prior to clinical use. The validation process is a complicated task due to lack of a gold standard [27]. This should be done both through visual assessment and also quantitative numerical analysis. Validation is not unique to image registration and has to be carried out for other medical image processing methods such as segmentation, visualization, and calibration [28]. A biomechanical-based algorithm which employs finite element method has been proposed in [29]. The algorithm assesses the accuracy of a previously developed non-rigid registration algorithm with application to MR mammography [24]. The accuracy of the registration algorithm in [24] were evaluated by simulating voxel displacements and considering the elastic behavior of the breast tissue. The authors claim that their proposed method can be adopted to patient-specific anatomy and has the ability to assess the accuracy of other algorithms. In [30], a validation process has been developed for retinal image registration. In diagnosis of ophthalmologic disorders, it is common to perform a registration process on multiple images to provide images with larger field of view. The validation method assesses the geometric misalignment of the overlapping regions with a coordinate system of a reference standard.

2.5 Image Similarity/Distance Measures

In medical image registration, a metric is required to compute the similarity of images being compared. This measure can evaluates how closely the images are

aligned. In general, registration algorithms employ corresponding points, corresponding surfaces [31], or image intensities in calculating image similarity [6,32].

2.5.1 Feature-based Measures

These measures employ corresponding points or/and surfaces to assess the similarity of images. A popular metric is the sum of the distances between each corresponding point in the images. These points can be artificial landmarks such as pins or markers affixed to the patient, e.g. attached to the skin or screwed into bone, or anatomical features [5,18]. The advantage of such measures is that the registration error only depends on the fiducial localization error (FLE), therefore, the clinical accuracy of registration can be assessed using realistic phantoms [33].

The distance between corresponding surfaces in the images can also be used as similarity measure. "Head and Hat" method is a widely used algorithm in which a set of points are identified in contours that are drawn on images [18]. An iterative algorithm is employed to compute the sum of squares of the distances between the corresponding points until this value reaches a minimum. Image registration of the brain in cortical region should deal with complex network of sulci and gyri. Purely intensity-based algorithms are unable to provide accurate results for this region [34]. A volumetric registration method has been developed in [34] for the cortical region that employs surface constraints. First, a map is computed to align the folding patterns of the sulci. Second, an intensity-based registration algorithm is carried out to complete the mapping and align the subcortical structures. The authors claim that their method has better accuracy than volumetric registration algorithms without surface constraints.

2.5.2 Intensity-based Measures

Voxel/Pixel intensity-based similarity measures employ all or a large portion of the image data and therefore usually yield more robust and accurate registration results than those achievable in feature-based methods. [18].

Sum of squared intensity difference (SSD) is a simple similarity measure that is widely used in single-modality image registration [35–37]. The SSD of two images is given by

$$SSD = \frac{1}{N} \sum_{x_A \in \Omega^T_{A,B}} \mid A(x_A) - B^T(x_A) \mid^2 \tag{2.1}$$

where x_A is the voxel location in image A within an overlap domain $\Omega^T_{A,B}$ that has N voxels. B^T represents the iteratively transformed image. SSD is sensitive to a small number of voxels that have very large intensity difference. For instance, when a contrast agent is injected to one of the images, the intensity differences of some voxels are increased. To diminish this sensitivity, sum of absolute difference (SAD) may be used.

$$SAD = \frac{1}{N} \sum_{x_A \in \Omega^T_{A,B}} \mid A(x_A) - B^T(x_A) \mid \tag{2.2}$$

Correlation coefficient (CC) is another similarity measure that can be used in medical registration algorithms, especially when the images are from the same imaging modality. The formulation, which is described in [18], involves multiplication of corresponding image intensities.

$$CC = \frac{\sum_{x_A \in \Omega^T_{A,B}} (A(x_A) - \bar{A}).(B^T(x_A) - \bar{B})}{\{\sum_{x_A \in \Omega^T_{A,B}} (A(x_A) - \bar{A})^2 . \sum_{x_A \in \Omega^T_{A,B}} (B^T(x_A) - \bar{B})^2\}^{1/2}} \tag{2.3}$$

where $\bar{A}$ and $\bar{B}$ are the mean voxel values in images A and B within the domain $\Omega^T_{A,B}$, respectively.

Recently, mutual information has been considered as a similarity measure in multi-modality image registration. The concept dates back to the work of Shannon's definition of entropy [38]. Since this measure makes no assumption about the functional relationship between image intensities, it is widely used in multi-modality image registration. Mutual information is defined as follows:

$$I(A,B) = H(A) + H(B) - H(A,B) \tag{2.4}$$

where $H(A)$ and $H(B)$ are the marginal entropy and $H(A,B)$ is the joint entropy of A and B.

2.6 High Performance Computing for Image Registration

In general, "degrees of freedom" of a registration model refer to the number of parameters of the registration transformation [18]. A rigid registration changes the position and orientation without changing the shape and size between the two scenes. The degrees of freedom of rigid registration algorithm is six, three for translation and three for rotation in 3D-3D. Affine registration adds scaling and shearing to translation/orientation, resulting in twelve degrees of freedom. Parametric non-rigid algorithms require more degrees of freedom. Physics-based methods have been developed to reduce the number of parameters [3, 18].

2.6.1 Multi-processor Systems

In [7], multiprocessor systems have been employed to speed up image registration. Parallel processing with 64 CPUs using a shared memory architecture, average runtime of 67 seconds and 89 seconds for nonrigid registration in intraoperative brain deformation analysis and contrast-enhanced MR mammography were reported, respectively. Multiprocessor systems although powerful, are expensive because of their high Maintenance and acquisition costs [8].

2.6.2 FPGA-based Systems

FPGA-based computing architectures are highly customizable and are capable of accelerating computations, if properly designed. An Altera Stratix EP1S10F780C5 FPGA has been employed in [9] to design and implement a mutual information-based affine registration algorithm. The authors stated that their architecture is reconfigurable to handle various image data sets. The speed-up results of the proposed design were not reported in this study. An FPGA-base implementation has been developed in [39]. A mutual information-based deformable registration algorithm has been employed in this study. The design reduced the registration time from hours to minutes for $256 \times 256 \times 256$ abdominal CT and PET images.

2.6.3 GPU-based Systems

Registration algorithms are often computationally intensive. In recent years GPUs have emerged as a powerful platform for high-performance computing applications. GPUs offer a flexible programming environment and compact and inexpensive hardware for massive parallel computing in scientific applications. We will briefly review the literature GPU-based implementations of various elements of image registration algorithms.

GPU-based Rigid and Affine Registration

To the best of our knowledge, GPU-based medical image registration was initially introduced in 1998 by Hastreiter et al [40]. A rigid transformation has been used in their studies for head data sets registration. The registration was based on Mutual Information and intuitive visualization of medical data sets. The hardware was mainly employed to accelerate the trilinear interpolation parts of the algorithm. Their proposed implementation was a factor of 2-3 faster compared to similar studies for 3D-3D rigid registration of head data sets running on a conventional CPU.

In [41], a rigid 2D-3D medical image registration algorithm has been implemented on a NVIDIA GeForce GTX 8800. The GPU implementation was developed under RapidMind software. The entire algorithm has been implemented on a single GPU and a runtime of about 3 seconds has been reported for $128 \times 128 \times 128$ CT data sets. A GPU-based implementation for a rigid medical image registration algorithm has been proposed and compared with cluster based methods

in [42]. Real spine and femur phantom data sets have been employed for experimental analysis. The authors mentioned that the maintenance and power consumption costs of the GPU implementations are much less than cluster computing approaches, therefore, GPUs can be used successfully in many clinical applications such as computer-aided surgery.

A GPU-based implementation of 2D and 3D rigid and nonrigid registration algorithms with focus on SSD was presented in [10]. The algorithm was coded by OpenGL and ran on a GeForce 6800 GPU. A CUDA implementation of a deformable registration algorithm based on SSD was proposed in [43]. The accuracy and speed of the same registration algorithm were compared by running it on a single CPU, single GTX 8800 GPU, and a cluster of GPUs. The authors stated that computation acceleration can be achieved at the expense of accuracy. This is due to the fact that at the time of implementation, GPUs were optimized for single precision floating point operations.

Researchers have investigated acceleration of MI-based multi-modal image registration methods on GPUs [21]. In [44], a CUDA capable GPU has been used to accelerate a mutual information-based registration algorithm. The authors implemented the registration algorithm in MATLAB and CUDA environments. The optimization of their registration cost function was carried out in MATLAB on the host CPU whereas transformation and bilinear interpolation were performed on GPU. For $256\times256\times176$ MR images, the authors reported an approximate runtime of 27 seconds for their GPU-based rigid registration algorithm which constitutes 14-fold speedup compared with a CPU-based implementation. In [20], a MI-based rigid registration algorithm was accelerated using a NVIDIA Geforce 7300 GPU.

The GPU implementation of the algorithm was tested for $256 \times 256 \times 100$ CT image sets and a 7-fold speed up was reported compared with a dual-core CPU implementation. The registration was performed in about 40 seconds with a single GPU.

A CUDA-based implementation for 3D-3D affine transformation based on Mutual Information has been proposed in [21]. A novel algorithm for efficient calculation of Mutual Information on GPU has been developed in this study. The main focus of their work has been on multi-modal image registration between CT and MR images, therefore Mutual Information has become the cost function of choice in their studies. Using a GTX 280 GPU with 30 multi-processors and 8 cores per multi-processor and taking advantage of texture memory for interpolation, their result shows that the proposed implementation is able to register multi-modal images in less than one second; This is about fifty times faster than similar works based on serial computing on conventional CPUs. The authors also compared their results using GTX 8800 and GTX 280 NVIDIA devices with different number of multiprocessors. This was done to show that with future advances in the GPU technology, even faster implementation of complex image registration algorithms will be feasible.

GPU-based Non-Rigid Registration

A 3D image registration program, denoted as Accelerated Image Registration with CUDA (AIRWC) has been developed based on the B-Spline method in [45]. Normalized correlation and normalized Mutual Information have been used as cost functions of the registration algorithm. The implementation was tested and analyzed by using human and mouse brain MR images.

Sharp et al. [46] implemented the Demons deformable registration algorithm in the Brook programming environment on an NVIDIA GeForce 8800 GPU. GPU times of 4 seconds and 13 seconds were reported for the registration of $256 \times 128 \times 128$ and $428 \times 180 \times 150$ images of swine lung.

In [47], Demons algorithm has been implemented on NVIDIA's Quadro FX 5600 GPU with CUDA environment. In this work, the algorithm has been divided into five parts that are suitable for parallel implementation and CUDA kernels have been developed for each part. These include gradient, interpolation, displacement, smoothing, and correlation computations. Since the hardware-based trilinear interpolation was not accessible at that time from CUDA interface, their interpolation kernel was developed without using the texture memory. This GPU implementation yielded a factor of 55 speed-up over an optimized CPU implementation for registration of 3D CT images of the lung. The GPU-based implementation of the algorithm takes about 13 seconds to complete 3D registration of high resolution $512 \times 512 \times 54$ images. The authors of this paper claim 10% faster runtime with CUDA compared to what was achieved in the Brook programming environment.

In [48], a 2D non-rigid registration algorithm for cardiac motion estimation based on FEM and SSD was implemented on NVIDIA 7700 and 7950 GTX. The code was developed in OpenGL. The authors reported 5-fold speed-up compared to a CPU-based implementation; the total time of registration was about 52 seconds.

Recently, Joldes et al. [49], proposed a GPU implementation of a biomechanical model to find the displacement field that can be applied to register preoperative

and intraoperative brain images to have a real-time prediction of brain shift. The algorithm uses a patient-specific brain mesh, and therefore requires segmentation of preoperative images. The implementation is capable of registering 3D brain MR images in about 4 seconds on a CUDA capable GPU. The resolution of the image and the model of the GPU were not reported in this study.

Chapter 3

Deformable Registration Algorithm

This chapter contains the mathematical formulation of the new 3D registration algorithm in [1]. We will focus on the SSD as the distance measure because it is computationally efficient and works relatively well in single modality registration. The registration algorithm uses a dynamic finite element continuum mechanics model of tissue deformation. The algorithm involves computationally intensive tasks such as interpolation, solving a system of second-order differential equations, finding the 3D deformation using linear shape functions on tetrahedral finite elements, and solving a large system of equations based on the conjugate gradient method. Therefore, it is impossible to achieve anything that comes close to real-time registration using conventional CPU-based implementations. In the next chapter we will discuss how many of these computations can be accelerated by a parallel computing kernels running on a GPU.

3.1 Physics-based Deformation Modeling

In this section we briefly discuss the fundamentals of tissue deformation modeling based on the continuum mechanics. The reader is referred to [50] for a full treatment of this problem. In continuum mechanics-based models, the object deformation depends on the external forces and object's material properties. The object is in its equilibrium state when Π, its potential energy, is at a minimum:

$$\Pi = \Lambda - W, \tag{3.1}$$

where Λ is the total strain energy of the object, the energy stored in the body as the material deforms and W denotes the work done by external loads on the deformable object [26]. The system potential energy reaches a minimum when the derivative of Π with respect to the displacement function is zero which leads to a continuous differential equilibrium equation that must be solved to find the displacement. FEM, divides the object into a set of elements to approximate the equilibrium equation over each element. The following steps describe how FEM computes the object deformation:

- Derive the equilibrium equation from the potential energy equation.
- Divide our objects into elements and choose an appropriate shape function.
- Re-define the equilibrium equation for each element based on the interpolation function and nodal displacement.
- Solve the system for the nodal displacement.

- Having the displacement of nodal points and considering the shape function, the displacement of any point within the element can be found.

To minimize 3.1, Λ and W are expressed in terms of the material displacement:

$$\Lambda = \frac{1}{2}\int_V \sigma^T \varepsilon \, dV = \frac{1}{2}\int_V \varepsilon^T D\varepsilon \, dV, \tag{3.2}$$

where $\sigma^T = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{yz}, \sigma_{zx}, \sigma_{xy})$ and $\varepsilon^T = (\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \varepsilon_{yz}, \varepsilon_{zx}, \varepsilon_{xy})$ are the vectors of the stress and strain. For a linear elastic material, D is a matrix which represents the stress-strain relationship according to the Hooke's law.

For small deformations, normal and shear strains are related to the displacement vector u=$(u, v, w)^T$ as:

$$\varepsilon_{xx} = \frac{\partial u}{\partial x} \quad \varepsilon_{yy} = \frac{\partial u}{\partial y} \quad \varepsilon_{zz} = \frac{\partial u}{\partial z}\,, \tag{3.3}$$

$$\varepsilon_{yz} = \frac{\partial u}{\partial z} + \frac{\partial u}{\partial y} \quad \varepsilon_{zx} = \frac{\partial u}{\partial x} + \frac{\partial u}{\partial z} \quad \varepsilon_{xy} = \frac{\partial u}{\partial y} + \frac{\partial u}{\partial x}\,, \tag{3.4}$$

The work done by the external force f(x,y,z) is equal to the dot product of the applied force and the material displacement u:

$$W = \int_V u.f \, dV, \tag{3.5}$$

The choice of elements and interpolation functions depends on the shape of the object and required accuracy. In fact, there is a trade-off between the accuracy and computation. Using shape function and having the displacement of nodal points, the 3D displacement vector u of a point (x, y, z) can be expressed as a linear

combination of interpolation functions applied to the nodal displacement:

$$u = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = HU \tag{3.6}$$

where $U = (u_1, v_1, w_1, u_2, v_2, w_2, ..., u_N, v_N, w_N)^T$. N is the number of nodal points in the element, H is a $3 \times 3N$ matrix denoting the interpolation function (which depends on the type of the element) and U is the vector of nodal displacement. Having 3.3, 3.4 and 3.6, the strain ε at (x, y, z) can be written as a function of nodal displacement:

$$\varepsilon = BU, \tag{3.7}$$

where B depends on the shape function. The strain energy in the element can be written as a function of the nodal displacement U:

$$\Lambda = \frac{1}{2} \int_V U^T B^T DBU \, dV = \frac{1}{2} U^T (\int_V B^T DB \, dV) U, \tag{3.8}$$

Similarly, the work done by an external force $f(x, y, z)$ can be written as a function of nodal displacement:

$$W = \int_V U^T H^T f \, dV = U^T (\int_V H^T f \, dV), \tag{3.9}$$

Substituting 3.8 and 3.9 into 3.1, the potential energy of our deformable object can be written as:

$$\Pi = \frac{1}{2}U^T(\int_V B^T D B \, dV)U + U^T(\int_V H^T f \, dV), \tag{3.10}$$

The above equations are for a single element. In order to minimize the potential energy the partial derivatives $\frac{\partial \Pi}{\partial U_i}$ is set to zero which yields a linear equation in the form of $K^{el}U^{el} = F^{el}$, K^{el} is the stiffness matrix and F^{el} is the vector of concentrated nodal forces for each element. The stiffness matrix and vector of nodal forces are computed for each element and are then assembled in a global system as $KU = F$.

The image registration methods in [1] is based on a dynamic model which expresses the object motion/deformation as it moves toward its equilibrium shape. A dynamic linear elastic deformation model comprised of a set of second-order differential equations for nodal displacements as:

$$M\ddot{U} + C\dot{U} + KU = F, \tag{3.11}$$

where M, C and K are the mass, damping, and stiffness matrices respectively, F is the vector of external forces and U represents the nodal displacement.

3.2 Mathematical Formulation of the Registration Algorithm

The registration problem can be formulated as finding the displacement field u that minimizes the following cost function:

$$J(u) = D(T[u], R) + \alpha S(u); \qquad \alpha \in \Re_+ \tag{3.12}$$

where D is a distance measure between the reference R and the deformed template $T[u]$ images. In this work, SSD which is suitable for single modality registration has been employed as a distance measure. The second term, S is a regularization term based on a physical model for the deformation that would ensure that the resulting solution is "reasonable". α weights the importance of the regularization term compared with the distance measure.

In a linear elastic continuum with no initial stress or strain, the potential energy of a body subject to externally applied forces can be expressed as [51]:

$$E = \int_{\Omega} \sigma^T \epsilon d\Omega + \int_{\Omega} u^T f d\Omega; \tag{3.13}$$

where f is the vector of forces applied to the elastic body, u the displacement field, and Ω is the body of the elastic object. ϵ and σ are the strain and stress vectors respectively.

Considering 3.8 and 3.13, our registration problem can be written as:

$$J(u) = D(T[u], R) + \frac{\alpha}{2} u^T K u; \qquad \alpha \in \Re_+; \tag{3.14}$$

where $K = \int_{\Omega} B^T D B d\Omega$ is the global stiffness matrix associated with the volumetric mesh, D is the elasticity matrix characterizing the material's property and B depends on the shape function of the finite element. It should be noted that based on the concept of finite element discretization, a volume of elastic body is approximated as an assemblage of discrete finite elements interconnected at nodal points.

The nodal displacements are propagated to any point x of the template image

volume using the shape function of the element containing the point:

$$u_{in}(x) = \sum_{i=1}^{4} N_i^{el}(x) u_i^{el}(x) \tag{3.15}$$

where $u_{in}(x)$ is the displacement for the internal point and $u_i^e l(x)$ is the displacement of the nodal points of the element. $N_i^{el}(x)$ is the shape function of the elements which will be explained more in this chapter. If J in 3.14 has a minimum at u, its first derivative mush vanish, i.e.

$$\frac{\partial J(u)}{\partial u} = \frac{\partial D(T[u], R)}{\partial u} + \alpha K u = 0 \tag{3.16}$$

This equation can be written as a set of nonlinear equilibrium equations for static analysis

$$Ku = f(u) \tag{3.17}$$

where $f(u) = -\frac{1}{\alpha}\frac{\partial D(T[u],R)}{\partial u}$ is the vector of concentrated nodal forces applied to the volumetric mesh. The solution to 3.17 will provide the displacement field corresponding to the global minimum of the objective function 3.14.

3.3 Incorporating Dynamic Deformation Model into Registration

The force vector $f(u)$ in 3.17, is a nonlinear function of the displacement field u. An iterative numerical method has to be employed to solve the nonlinear system of equations in 3.17. To this end, we consider the second-order dynamic model for

the motion/deformation of deformable object in Eq. 3.11 with $U = u$ and $F = f(u)$. The steady-state equilibrium of this dynamic system is the solution to the static system of equation 3.17. In 3.11, the dynamic forces $M\ddot{u}$ and $C\dot{u}$ tend to smoothly drive the system towards this equilibrium. The dynamic system of equations can be solved using existing explicit or implicit numerical integration routines such as the Newmark method [50].

A dynamic model also allows for real-time intraoperative registration of a dynamically changing organ. Real-time MR based biopsy interventions, for instance, can take advantage of this feature to provide correlation model for deformation of soft tissue due to the force of needle insertion [1].

3.4 How Does the Algorithm Work?

In each iteration of the algorithm, the nodal force $f(u)$ have to be computed and then used to drive the dynamical system in 3.11. The registration solution is obtained when the system reaches its steady state. In this study we use the SSD as the distance measure because it is computationally efficient and works relatively well in single modality registration. To compute the force applied at nodal points in each iteration, we need to compute the derivative of the distance measure at those points. Therefore, the force vector at any nodal point can be computed as:

$$f(u_i) = -\frac{1}{\alpha}(T(p_i + u_i) - R(p_i))\nabla T(p_i + u_i) \tag{3.18}$$

where $f(u_i)$ is a 3×1 vector, $\nabla T(p_i + u_i)$ is the gradient of the deformed template image at the deformed nodal point $p_i + u_i$. $T(p_i + u_i)$, $R(p_i)$ and $\nabla T(p_i + u_i)$ are

computed using the trilinear interpolation algorithm at each iteration. When 3.11 reaches to its equilibrium point, the displacement of the finite element nodal points u can be used to calculate the displacement of the regular 3D high-resolution grid of the template image based on the shape function based on Eq. 3.15.

Eq. 3.18 shows that the force vector is computed based on the pixel values and the gradient of the template image only at the nodal points. Obviously, the registration accuracy is improved by increasing the resolution of the finite element mesh at the expense of a greater computational load.

In order to use the whole information of the reference and template images, displacements which are computed in a finer 3D regular grid x_p inside the image volume are used to find the nodal point forces. The displacement of the regular grid are computed according to 3.18, i.e. $\triangle x = -\kappa(T(x_{p+u}) - R(x_p))\nabla T(x_{p+u})$, where $\kappa \in \Re_+$, x_p and x_{p+u} represent the original and the deformed grid points in each iteration. Nodal forces can be approximated based on $\triangle x$ using the inverse of the shape function. This requires solving a system of linear equation, i.e. $Af = b$, where A is a long matrix derived from the linear shape function of tetrahedral elements and vector b results from $\triangle x$ of the regular grid.

In the next chapter, we will explain each step of the algorithm in more details and and present parallel computing kernels that can significantly speed up the registration process.

Chapter 4

GPU-based Parallel Implementation of the Image Registration Algorithm

In this chapter, we discuss how to accelerate the deformable image registration algorithm in Chapter 3 by designing parallel kernels and taking advantage of execution resources and high-bandwidth memories of a CUDA capable GPU. First, we review the GPU architecture and CUDA programming model. Thread organization, thread assignment, and GPU memories are discussed in this context. Second, we divide the image registration algorithm into five steps and explain how each step is implemented and optimized (e.g., shared memory vs. global memory) using the CUDA programming model and the GPU resources.

4.1 GPU Architecture

A CUDA capable GPU consists of highly threaded streaming multiprocessors, each with a number of streaming processors that share some units such as control logic

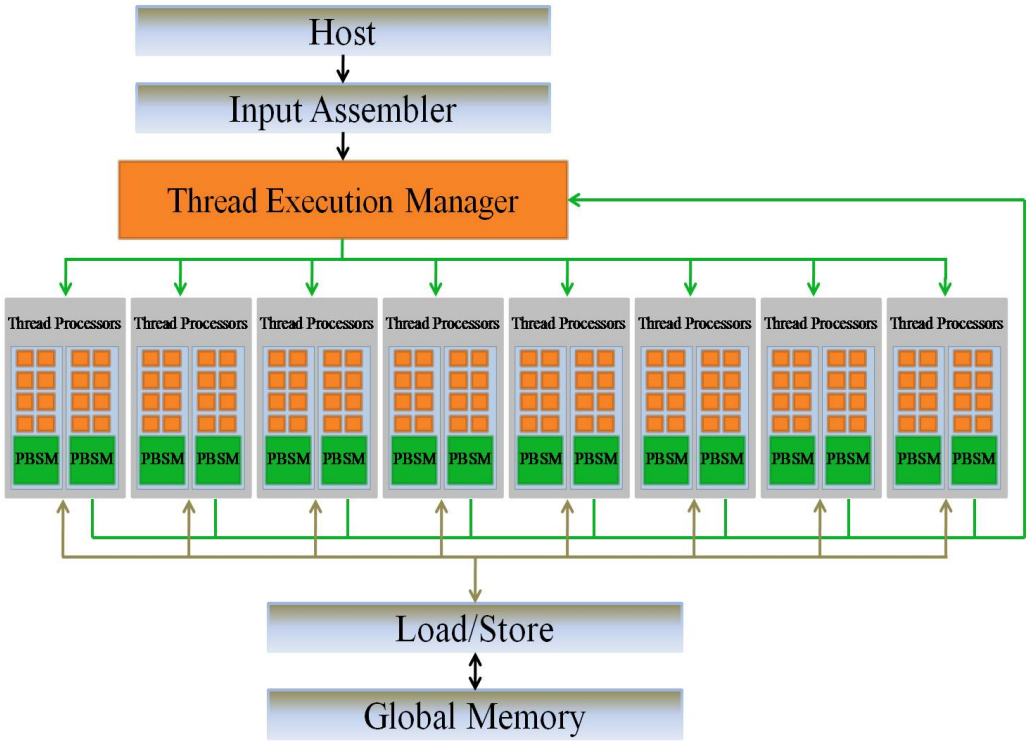


Figure 4.1: CUDA capable GPU architecture, G80; from [12].

and instruction cache, e.g. see Fig. 4.1. The number of processors depends on the generation of the GPU. For instance, the G80 that introduced the CUDA architecture in 2006, has 16 streaming multiprocessor each with 8 streaming processor, yielding a total of 128 cores. In CUDA, the CPU (host) and GPU (device) have separate memory spaces. In fact, GPUs have their own graphics double data rate (GDDR) DRAM, known as global memory. These memories are capable of holding graphics information such as video images and texture. They can be used as high-bandwidth off-chip memories in massively parallel non-graphics application, known as General-Purpose Computing on Graphics Processing Unit (GPGPU).

The memory bandwidth of CUDA capable GPUs has grown by a factor of two

since their early generations. G80 (Nov 2006) had 86.4 GB/s of memory bandwidth compared to 177.4 GB/s for GTX 480 (March 2010). Each core in GPU is equipped with a multiply-add (MAD) unit and an additional multiply unit. Some floating point functions such as square root have their own dedicated units. PBMS in Fig. 4.1 refers to per-block shared memory which will be discussed shortly. Joint CPU/GPU execution is one of the main advantages of CUDA programming model. As mentioned before, CPUs are optimized for sequential tasks. In another words, not all the tasks can be accelerated in GPUs. One can achieve high level of computing performance by assigning serial tasks to CPU and parallel tasks to GPU [12].

The GPU hardware provides support for the execution of thousands of threads in parallel. The threads are grouped in blocks and the blocks are organized in a grid. A typical CUDA program starts with the host code and when a GPU function (kernel) is invoked, thousands of threads will take advantage of GPU execution resources to perform the parallel task. In this model, serial code executes on the host while parallel code runs on the device, i.e. see Fig. 4.2.

4.2 Thread Organization

Once a CUDA kernel is launched, a grid of threads are created to execute the corresponding kernel. As stated before, these threads are grouped into a two-level hierarchy. A thread can be addressed using its unique coordinate. Two built-in variables, ”blockIdx” and ”threadIdx” identify the thread coordinate. A grid (top level) can form a 1D or 2D array of blocks and each block (bottom level) can shape a

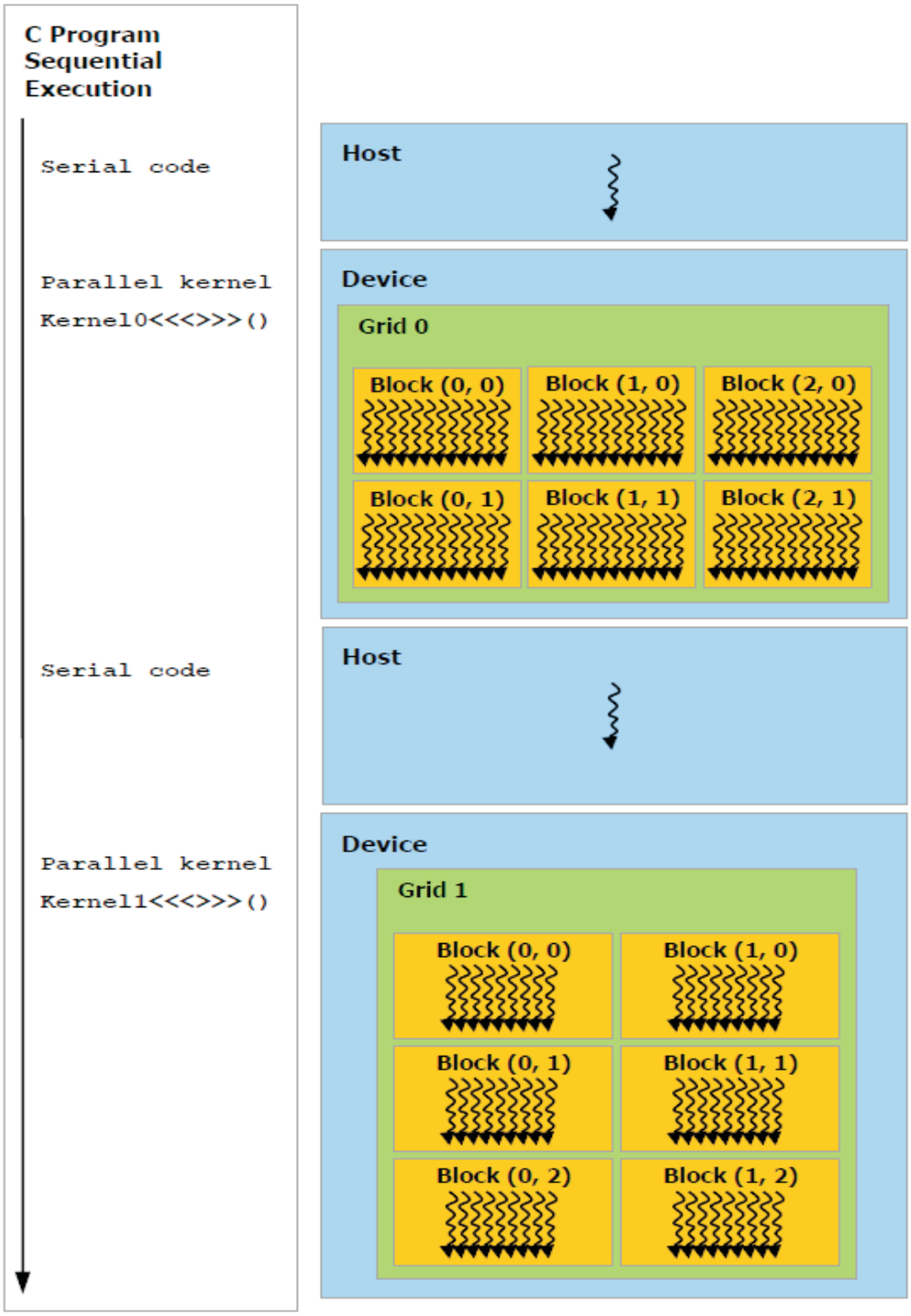


Figure 4.2: CUDA programming model; from NVIDIA's document [13].

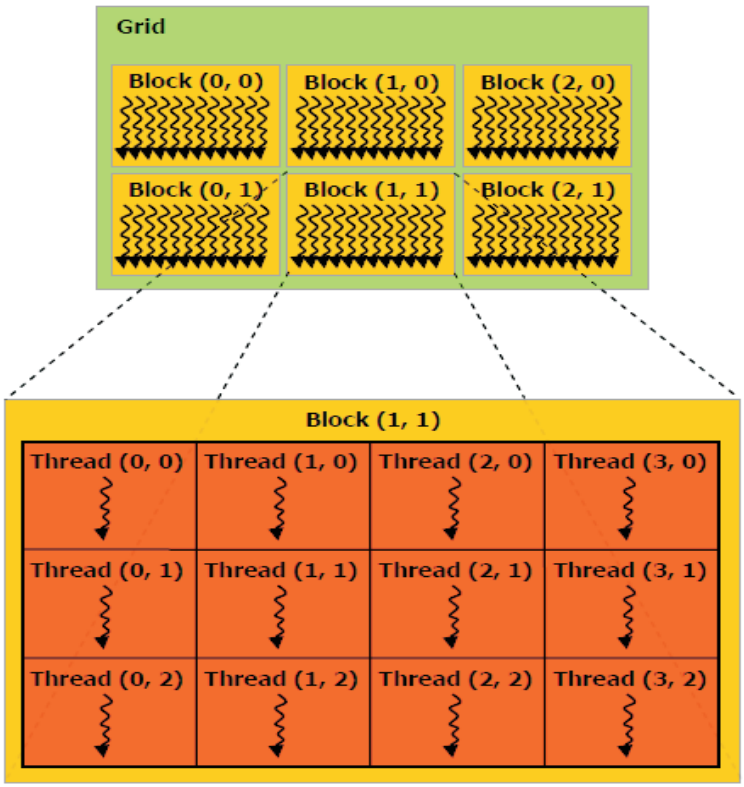


Figure 4.3: Thread organization; from NVIDIA's document [13].

1D, 2D or 3D array of threads. Two more built-in variable, "gridDim" and "blockDim", provide the block and grid dimensions.

There are some constraints for the maximum dimension of a block/grid. These constraints vary from one device into another. Fig. 4.3 shows a 2D grid of thread blocks where each block is organized in 2D threads. For GTX 480, the maximum x (gridDim.x) or y (gridDim.y) dimension of a grid of thread blocks is 65536. The maximum x (blockDim.x) or y (blockDim.y) dimension of a block is 1024. The maximum z (blockDim.z) dimension of a block is 64. Moreover, the number of threads per block cannot be more than 1024. Therefore, $blockDim.x \times blockDim.y \times$

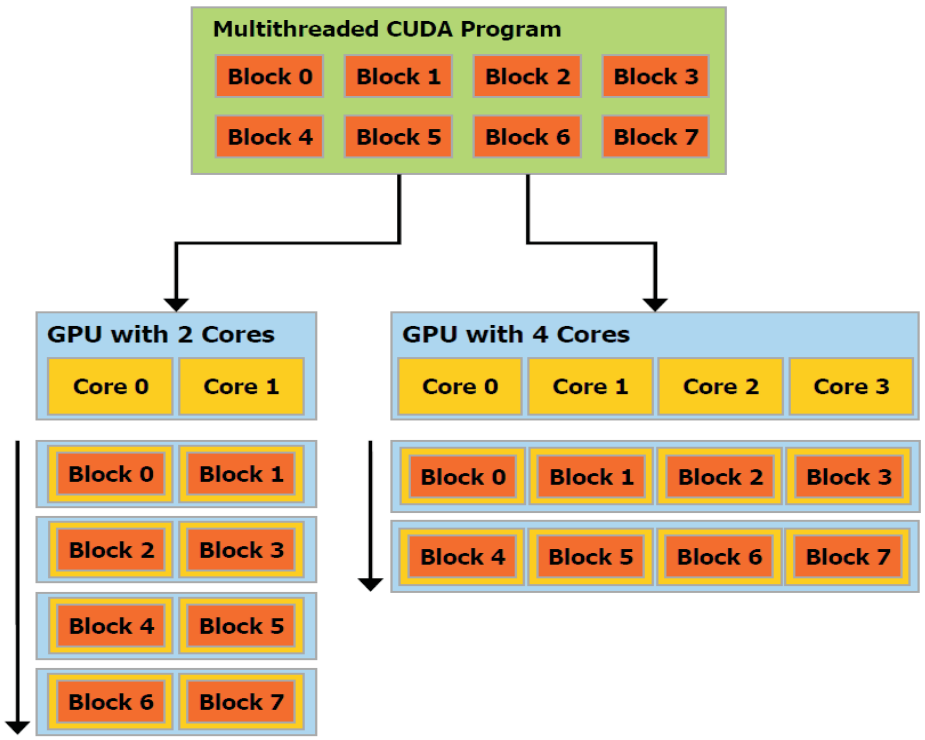


Figure 4.4: CUDA transparent scalability ; from NVIDIA's document [13].

$blockDim.z$ must be less than 1024 in a GTX 480.

All threads in a block can synchronize their tasks using "__syncthreads( )" function. This function is invoked inside the kernel and coordinates parallel execution of the threads. It does not allow any threads in a block to move on to the next step unless all threads in the block have completed the current step. This process is called "Barrier Synchronization".

Due to lack of synchronization between blocks, CUDA runtime system can execute blocks in any order. In fact a program that is partitioned in to multiple blocks can be scaled in various ways. GPUs with different number of cores (Fig. 4.4) can take advantage of this "Transparent Scalability" feature by running a number of

blocks at the same time according to their execution resources. Therefore a GPU with a greater number of cores can execute the program faster than a GPU with less cores [52].

4.2.1 Thread Assignment

After kernel invocation, threads are assigned to execution resources. Our GPU, GTX 480, has 15 multiprocessor and each multiprocessor can accomodate up to 8 blocks at a same time. These blocks are called ”resident blocks”. Therefore, we can have up to 120 resident blocks. The maximum number of resident threads per multiprocessor is 1536. This is twice as the number of threads that an early generation device such as G80 can handle. These threads can be organized in 6 blocks of 256, 3 blocks of 512, etc. A good understanding of the limitations of the execution resources is needed in order to optimize the computing performance. In our implementation, we have tried to improve the performance by changing the grid size and block size. This ”Dynamic Partitioning” feature, allows us to optimize our implementation by appropriate utilization of recourses. In fact, we have chosen the configuration which can efficiently occupy the thread slots and block slots in each multiprocessor, as illustrated in Chapter 5. Although constraints are different in various GPUs, the same application code can be run in all CUDA capable GPUs without any change. For instance, our implementation can be run in any CUDA capable GPU with compute capability 2.0 or more.

When a kernel is called, thousands of threads are queued up for work. All threads are divided in 32-units called ”warps”. In fact, threads are scheduled in warp units. Having so many warps, CUDA runtime system does not need to wait

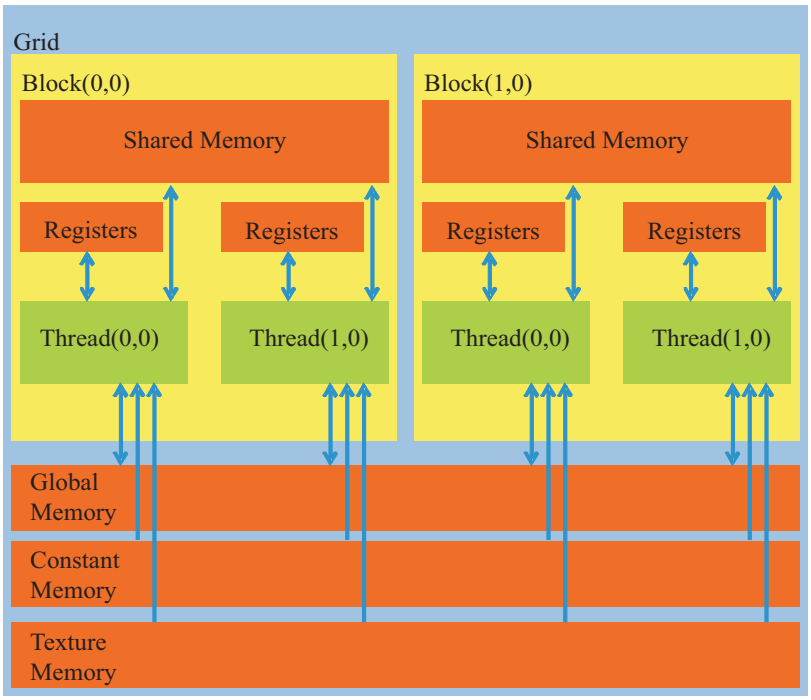


Figure 4.5: CUDA device memory model ; from NVIDIA's document [13].

for threads that are busy with a time consuming operation and can assign work to warps that are on standby. This mechanism is called "latency hiding".

4.3 Memory Types

Several types of memory are supported in CUDA, as shown in Fig. 4.5 and briefly discussed below.

- **Registers**: These are on-chip high-speed memories. Each thread in a block has access to its own register for placing automatic variables.
- **Shared Memory**: All threads within a block can share information through the shared memory. Shared memory can be used in many applications to

reduce the global memory access and increase the performance. In Section 4.4.2, we clarify how this type of memory is used in our implementation.

- **Constant Memory**: This memory is cached as read-only memory and would be suitable for constant variables.

- **Texture Memory**: This is a read-only type memory but can be used efficiently to increase the performance of computing applications. Later in the thesis, we will demonstrate how texture memory is employed in our algorithm to implement the interpolation step.

- **Global Memory**: In contrast with constant and texture memories, threads can read from and write into global memory. Too much global memory access can deteriorate the performance because the computation throughput is constraint by the loading speed from the global memory [12]. Since the global memory of the GPU is not cached, the performance of an application can be decreased due to memory contention. There are techniques available for reducing global memory access to increase the overall computing performance. We will clarify how shared memory can be employed to reduce the global memory access in Section 4.4.2 of the thesis.

4.4 GPU Implementation of Single-Modality 3D-3D Deformable Registration Algorithm

The 3D-3D image registration algorithm consists of the following five steps (see also Fig. 4.6):

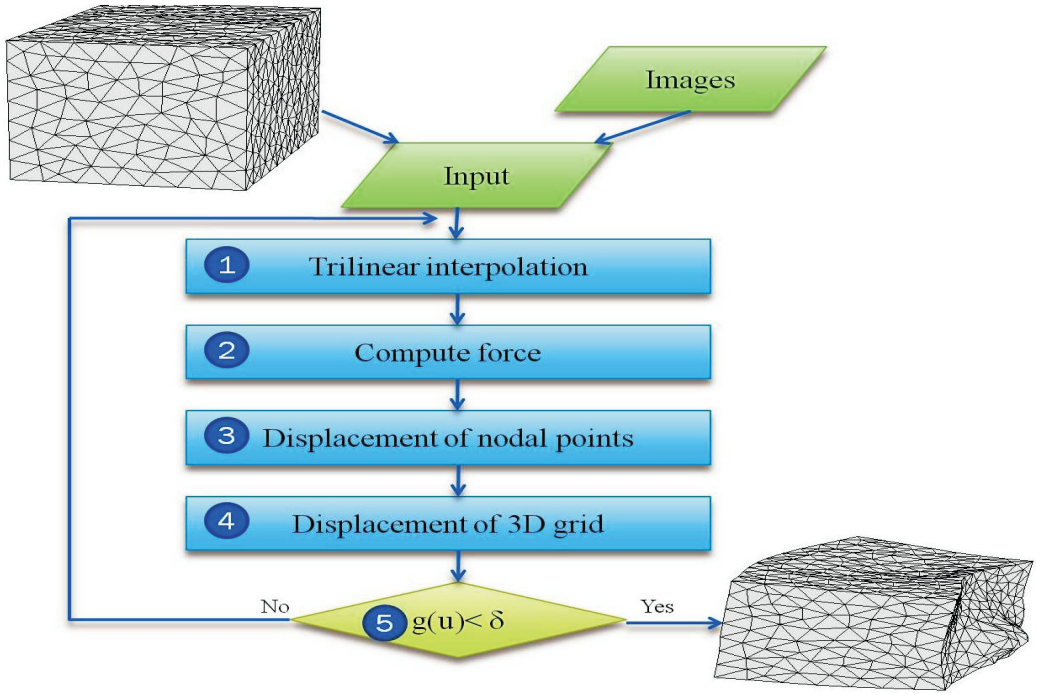


Figure 4.6: Registration algorithm flowchart.

1. Trilinear interpolation
2. Compute force
3. Displacement of nodal points
4. Displacement of 3D grid
5. Distance measure

In what follows, we will discuss how a CUDA capable GPU has been employed to accelerate the registration algorithm. CUDA kernels have been written for each step of the algorithm and then linked together to implement the whole algorithm.

4.4.1 Step 1: Trilinear Interpolation

As previously discussed about Eq. 3.18, the algorithm requires to compute the trilinear interpolation of the deformed template image (one interpolation of the gray value) and its directional gradients at x_{p+u} (one interpolation for each x, y and z direction). Therefore a total of four trilinear interpolations are required in each iteration of the algorithm. In this step the 3D regular grid points (see Section 3.3) are interpolated based on the $512 \times 512 \times 136$ MR voxels. As mentioned in Section 3.3, the accuracy of the algorithm increases by increasing the number of regular grid points. We have employed a 3D regular grid of 6000 points (20×30×10) for the 3D-3D single modality registration algorithm.

Texture memory is a cached on-chip read-only memory which is designed by NVIDIA for the OpenGL and DirectX rendering pipelines. Texure memory is designed for graphics application; however one can take advantage of this sophisticated feature of GPU for those tasks in which "spatial locality" access is a necessity for memory access [53].

In this step we exploit the linear filtering mode of the texture memory to accelerate the trilinear interpolations. The trilinear interpolation (Fig. 4.7) can be formulated as a weighted sum of the values of the neighboring points [13]. The formulation can be described as follows [54]:

$$\begin{aligned} G(x,y,z) &= c_0 + c_1\Delta x + c_2\Delta y + c_3\Delta z + c_4\Delta x\Delta y + \\ & C_5\Delta y\Delta z + c_6\Delta z\Delta x + c_7\Delta x\Delta y\Delta z; \end{aligned} \tag{4.1}$$

where Δx, Δy, and Δz are the relative distance of the 3D regular grid point G with

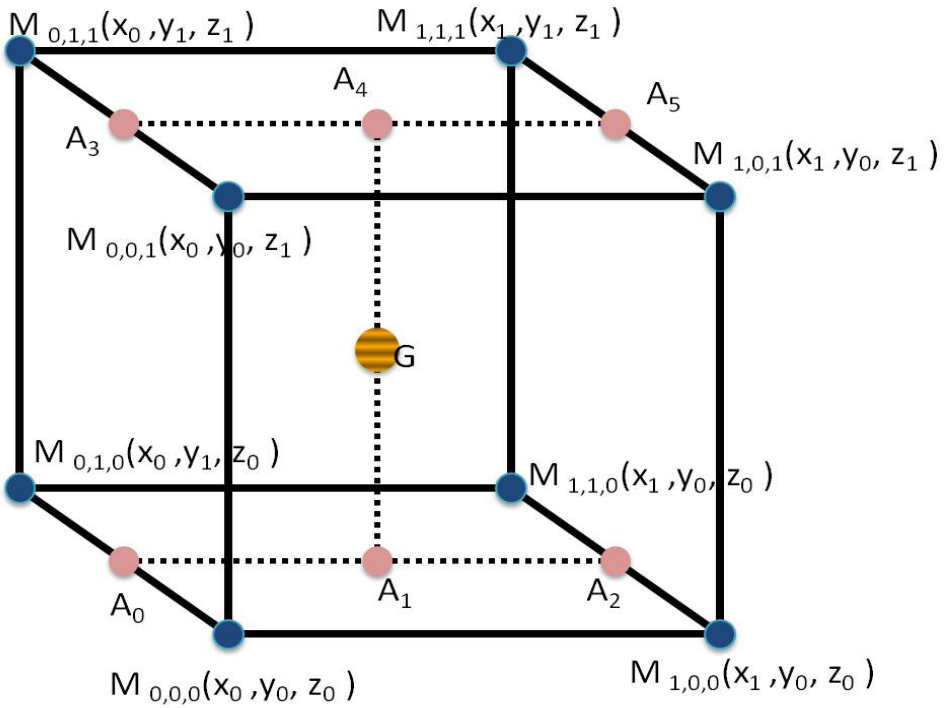


Figure 4.7: 3D regular grid point, G, is interpolated based on the MR voxels, $M_{0,0,0} - M_{1,1,1}$. Using auxiliary points $A_0 - A_5$, the trilinear interpolation can be illustrated as the combination of seven linear interpolations.

respect to the MR voxel point $M_{0,0,0}$ along the x, y, and z directions in Fig. 4.7. c_i are constants and are defined based on the gray values of MR voxels [54]:

$$
\begin{aligned}
&\Delta x = (x - x_0)/(x_1 - x_0), \Delta y = (y - y_0)/(y_1 - y_0), \Delta z = (z - z_0)/(z_1 - z_0) \\
&c_0 = M_{0,0,0}, c_1 = (M_{1,0,0} - M_{0,0,0}), c_2 = (M_{0,1,0} - M_{0,0,0}) \\
&c_3 = (M_{0,0,1} - M_{0,0,0}), c_4 = (M_{1,1,0} - M_{0,1,0} - M_{1,0,0} + M_{0,0,0}) \\
&c_5 = (M_{0,1,1} - M_{0,0,1} - M_{0,1,0} + M_{0,0,0}) \\
&c_6 = (M_{1,0,1} - M_{0,0,1} - M_{1,0,0} + M_{0,0,0}) \\
&c_7 = (M_{1,1,1} - M_{0,1,1} - M_{1,0,1} - M_{1,1,0} + M_{1,0,0} + M_{0,0,1} + M_{0,1,0} - M_{0,0,0})
\end{aligned}
\tag{4.2}
$$

The texture bound to the texture refrence is represented as an array T of $N \times M \times L$ texture pixels (texels) that is fetched using texture coordinates x, y and z. It should be noted that the maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array is $4096 \times 4096 \times 4096$ for the GTX 480. The texture Reference structure is defined as follow [13]:

```
struct textureReference{
int normalized;
enum cudaTextureFilterMode filterMode;
enum cudaTextureAddressMode addressMode[3];
struct cudaChannelFormatDesc channelDesc;
}
```

All above parameters and structures will be discussed shortly. A CUDA kernel is written in order to call *texture fetches*. In this kernel, we have adjusted the 3D grid point coordinates according to the texture coordinates:

$$new_x = \frac{(x + abs(w_1))N}{abs(w_1) + abs(w_2)} \tag{4.3}$$

where new_x is the new x coordinate of the 3D grid point and $\Omega_x = (w_1, w_2)$ is the domain of the MR image along x. new_y and new_z can be calculated similarly.

The texture reference must be initialized before being used in the kernel. In the initialization, output data type (e.g. integer, float), dimensionality (1D, 2D, 3D), and Read mode (output with/without conversion) are three features of texture reference that must be determined."cudaReadModeElementType" returns output without any conversion. These following texture parameters are employed in our implementation:

```
texture<float, 3, cudaReadModeElementType> texRef;
```

Assuming a texture of $N \times M \times L$, all texture coordinates can be either mapped in the range of [0,1] which is the normalized mode or in the range [0, N-1], [0,M-1], and [0, L-1] which is the non-normalized mode. In our implementation, we used the non-normalized mode, although the normalized mode also could be used in this step. Two filtering modes are available in the current version of CUDA, "nearest-neighbour interpolation" and "linear interpolation". In contrast with the linear interpolation which employs all eight neighbour voxels (see Fig. 4.7), the returned value in the nearest-neighbour mode is the voxel whose coordinates are the closest to the 3D grid point. In our implementation, the "linear interpolation" mode has been used in 3D to perform multiple trilinear interpolations in each iteration of the algorithm. The out-of-range data are clamped to a valid range using the "adressMode" in three dimensions. "channelDesc" specifies the format of returning data which is floating point type in our implementation. These additional parameters are set as:

```
cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<float>();
texRef.addressMode[0] = cudaAddressModeClamp;
texRef.addressMode[1] = cudaAddressModeClamp;
texRef.addressMode[2] = cudaAddressModeClamp;
texRef.filterMode = cudaFilterModeLinear;
texRef.normalized = false;
```

We have used `cudaMalloc3DArray` to allocate a 3D memory arrays to our template image and its gradients along x, y, and z coordinates (see Eq. 3.18). Our image sets which are stored in 1D arrays must be mapped into 3D array. `make_cudaExtent(512,`

`512, 136)` creates 3D volume that specifies the size of our 3D array:

```
cudaExtent volumeSize = make_cudaExtent(512, 512, 136)
cudaArray* cuArrayx=NULL;
cudaMalloc3DArray(& cuArraygx, & channelDesc, volumeSize);
```

Using `cudaMemcpy3DParams`, several parameters can be initialized in order to make this 3D mapping. First, source data which is a 1D array and destination data which is a 3D array are specified. It is also necessary to determine that the mapping is from host to device. Finally, cudaMemcpy3D performs the data transfer. The process is as follows:

```
// copy data to 3D array
cudaMemcpy3DParms copyParams = 0;
copyParams.srcPtr = make_cudaPitchedPtr((void*)h_data,
volumeSize.width*sizeof(float), volumeSize.width, volumeSize.height);
copyParams.dstArray = cuArraygx;
copyParams.extent = volumeSize;
copyParams.kind = cudaMemcpyHostToDevice;
cudaMemcpy3D(& copyParams);
```

Before reading from texture and writing to global memory, the texture reference must be bound to a texture using the following command:

`cudaBindTextureToArray( texRef, cuArraygx, channelDesc);` After the binding, the "tex3D" function is called inside the kernel to perform the trilinear interpolation. The returned value is the linear interpolation of eight texels whose texture coordinates are the closest to our 3D grid point coordinate (input).

As mentioned before, texture memory is generally designed for graphics application. It is cached on chip memory and its "spatial locality" feature significantly accelerates the interpolation step. Arithmetically, in trilinear interpolation eight voxels are read per each 3D grid value. In the serial implementation (using a CPU) and even global memory implementation (using a GPU) these eight voxels are not stored consecutively. Therefore, this non-consecutive access pattern decreases the performance of the interpolation kernel. The more 3D grid size, the more speed-up is provided by the texture memory. Speed-up analysis will be provided in Chapter 5 of this thesis.

4.4.2 Step 2: Compute force

The displacement of the regular grid are computed according to Eq. 3.18, i.e. $\triangle x = -\kappa(T(x_{p+u}) - R(x_p))\nabla T(x_{p+u})$, where $\kappa \epsilon \Re_+$, and x_p and x_{p+u} represent the original and the deformed grid points in each iteration. Nodal forces can be approximated based on $\triangle x$ using the inverse of the shape function [1,55]. This requires solving a system of linear equations in the form of $Af = b$, where A is a long matrix derived from the linear function of tetrahedral elements and vector b results from $\triangle x$ of the regular grid points. The system of equations must be solved at each iteration. The linear tetrahedral shape function is discussed in detail in Step 4.

Although the structure (location of zero and non-zero elements) of the sparse matrix A is constant in our registration algorithm, the values change in each iteration. Therefore direct solvers based on matrix factorization may not be very efficient in our application. They produce large matrices which are not as sparse as our A matrix and require larger memory storage space. The conjugate gradient

method is a powerful iterative algorithm for solving optimization problems as well as systems of equations. The main computations of this algorithm are one sparse matrix by vector multiplication and two dot products of dense vectors. Before the start of the iterations, an initial guess for the solution (could be a zero vector) is specified and residue and initial search direction are calculated. In the iterative part, the search direction and step length are updated and used in the calculation of a new approximate solution. This process continues until the residual error is small.

Considering A a real, symmetric, positive-definite matrix, the algorithm for solving $Af = b$ is given in the following pseudo-code:

$f = f_0$ **//initial guess**

$r_0 = b - Af_0$ **//residue**

$d_0 = r_0$ **//initial search direction**

$i = 0$

while ($\|r_{i+1}\|_2^2 > \ threshold$ and $i < MAX$)

$\alpha_i = \frac{\mathbf{r_i^T r_i}}{\mathbf{d_i^T A d_i}}$ **//step length**

$f_{i+1} = f_i + \alpha_i d_i$ **//approximate solution**

$r_{i+1} = r_i - \alpha_i A d_i$ **//residue**

$\beta_i = \frac{r_{i+1}^T r_{i+1}}{r_i^T r_i}$ **//improvement**

$d_{i+1} = r_{i+1} + \beta_i d_i$ **//new search direction**

$i = i + 1$

end

As mentioned before, this algorithm requires one sparse matrix-vector multiplication and two dot products of dense vectors, as highlighted in the pseudo-code. It

should be noted that other computations such as scalar multiplication and vector addition are also written in CUDA to take advantage of parallel computations.

For the dot product of two vectors, one kernel is required to calculate the element-wise product of the vectors. Another kernel then computes the sum of the elements of the result vector. A parallel reduction algorithm, discussed in Step 5, is employed to accelerate the dot product computation.

Sparse-matrix multiplication is discussed in Step 3. Our sparse matrix which is derived from the inverse of the shape function must be updated in each iteration. Since the structure of the matrix is constant, a parallel kernel is designed to update the values. We will present a solution to this problem can be handled in Step 3. In this section, we explain how CUDA carries out dense matrix-by-matrix multiplication and the critical role of shared memory in this process. The shared memory will also be employed in other sparse matrix operations.

A serial matrix multiplication in host requires three loops to perform $AB = C$, whereas a parallel multiplication in a CUDA capable device needs only one loop, i.e. the innermost loop of the serial code (see Fig. 4.8):

Instead of having two outer loops, one over row and another over the column of the matrix, a two-dimensional grid of threads can be used in CUDA to perform the same operation. When the multiplication kernel is invoked, each thread performs a dot product of a row of A and a column of B (arrows in Fig. 4.9) to calculate one element of the C matrix. This process is parallel for all the threads.

Since the multiplication kernel in Fig. 4.9 does not employ the "blockIdx" (see Section 4.2), the largest matrix size that can be handled by this kernel for our device is $32 \times 32 = 1024$ elements. The reason is that the maximum number of threads in

Serial Pseudo-Code	Parallel Pseudo-Code
1 Loop over the row of A 2 Loop over the column of B 3 Dot-product of i^{th} row of A and j^{th} column of B to calculate c_{ij}	1 Each thread calculates the dot-product of i^{th} row of A and j^{th} column of B to calculate all c_{ij}

Figure 4.8: Matrix multiplication serial and parallel pseudo-code.

a block is 1024 for a GTX 480.

Larger matrices can be handled using multiple blocks. For a GTX 480, the maximum x or y dimension of a grid of thread blocks is 65535. This means that we can have a grid of 65535×65535 thread blocks and each block can accommodate up to 1024 threads. It should be noted that this does not mean that all of these threads can run in parallel. Obviously memory access and limited execution resources restrict the number of threads running in parallel. In GTX 480, the maximum number of resident threads and blocks per multiprocessor are 1536 and 8 respectively. Having a knowledge of these limitations, one can properly employ CUDA resources to achieve the best performance possible in a GPU implementation.

In order to use "blockIdx", our example in Fig. 4.9 can be divided into 4 blocks such that each block has 4 threads (Fig. 4.10). In this case, thread $(0, 1)$ of block $(1, 0)$ performs the dot product between Row 1 of A and Column 2 of B to calculate $C_{2,1}$. Although large matrices can be handled with this technique, global memory access contentions can significantly degrade the performance of such implementation.

Using only global memory in Fig. 4.10, thread (0,1) and thread (1,1) of block

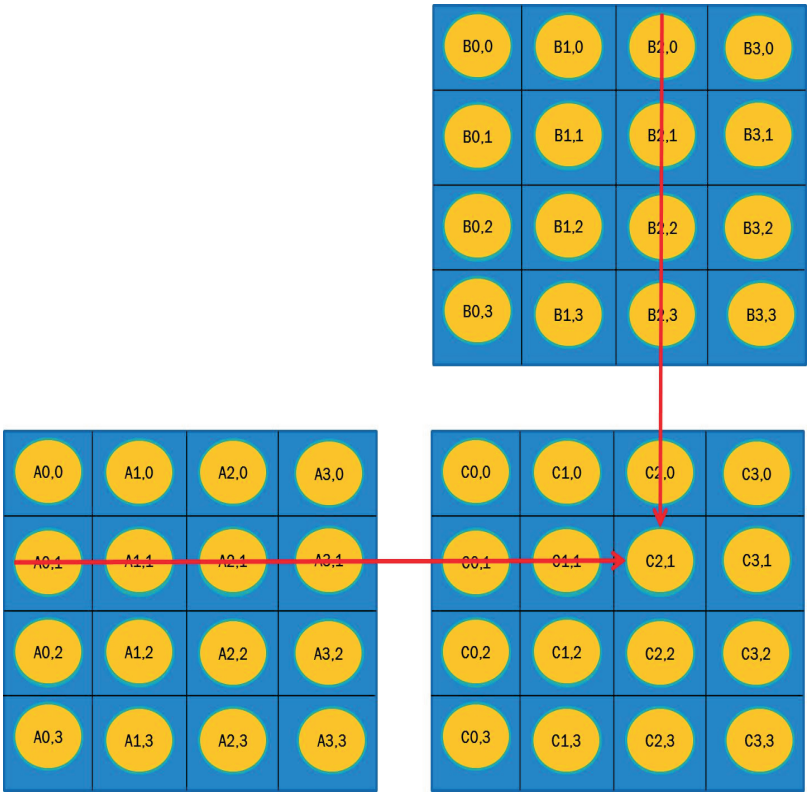


Figure 4.9: Matrix multiplication without using blocks.

(1,0) both access the first row of A possibly at the same time, causing global memory access conflicts. The larger the matrix, the more global memory access and the bigger the possibility of memory contention. Since the tile in our example is 2×2, each block loads a row of A matrix twice; the same argument applies to the column of B.

Using shared memory, all threads in a block can share data, reducing global memory access and improving the overall computing performance. In this case, thread (0,1) and thread (1,1) of block (1,0) load the first row of A from the global memory only once, reducing memory access by a factor of two.

In general, the matrix by matrix multiplication involves the following steps:

- First, threads of a block load a subset of elements from global memory into shared memory.
- Second, __synchthreads() (see Section 4.2) ensures that data is loaded for all threads that are in a block.
- Third, the threads perform a dot product between the loaded subset row of A and column of B (inner loop); the size of this inner loop is equal to the tile width.
- Lastly, __synchthreads() is called to make sure that the previous step is completed for all threads of a block.

Since only threads that are in a block can communicate with each other, a loop is required to load the next subset of elements [13]. This outer loop encompasses all the above steps. The size of this outer loop is equal to $\frac{M}{N}$, where M is the matrix size and N is the tile size.

As illustrated in the previous matrix multiplication with 2×2 tiles, the global memory access was reduced by a factor of 2. Therefore, having $N \times N$ tiles, the global memory access can be reduced by $\frac{1}{N}$. Shared memory is small, high speed, and on chip. Increasing N leads to reducing the global memory access, however, the size of shared memory is very limited. Excessive use of shared memory limits the number of threads assigned to each streaming multiprocessor [12]. In GTX 480, the maximum amount of shared memory per multiprocessor is 48 kilobytes (KB). In order to be able to assign the maximum number of blocks to each multiprocessor, each block can use up to 4KB of the shared memory. If each block uses more than this number, the total number of resident blocks in each multiprocessor

is reduced.

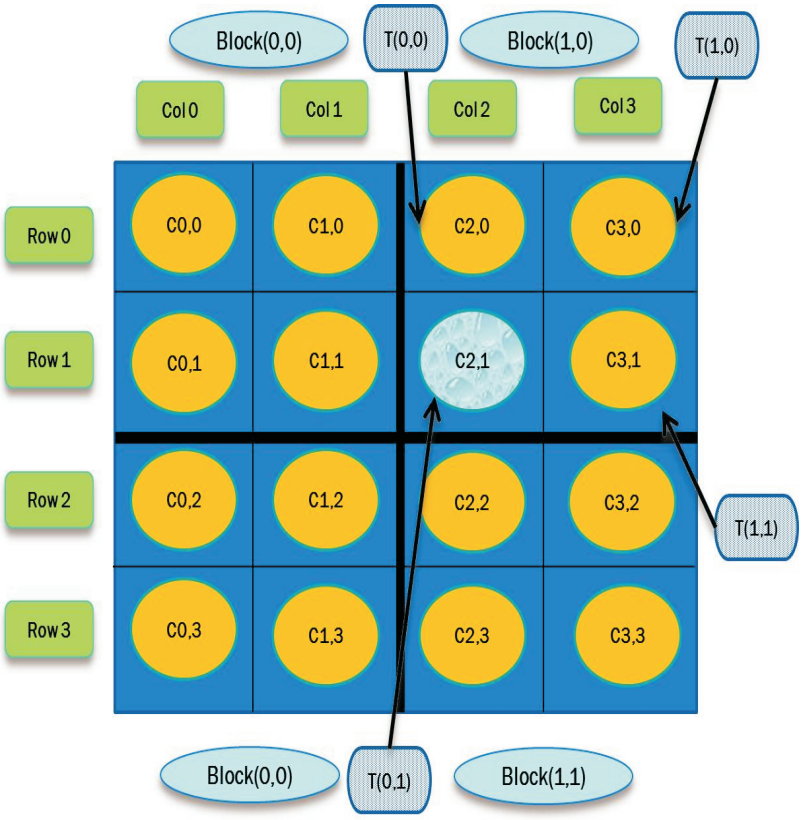


Figure 4.10: Matrix multiplication using multiple blocks.

The NVIDIA CUBLAS library which is provided by NVIDIA for basic linear algebra operations [56] has been employed in parts of our implementation for matrix and vector operations. CUBLAS provides some "helper functions" to initialize the library and to transfer vectors and matrices between CPU and GPU. It also provides functions to allocate memory in GPU (similar to cudaMalloc()). This part of the memory is a device memory and can also be used in other GPU kernels. In general, the library has three types of functions for vector-vector (BLAS1), matrix-vector (BLAS2), and matrix-matrix (BLAS3) operations.

In Section 4.4.3, we will discuss sparse matrix and vector operations used in Step 2 and 3 of our registration algorithm.

4.4.3 Step 3: Displacement of Nodal Points

Displacement of nodal points are calculated in this step of the registration algorithm. The dynamic finite element model is solved using the Newmark integration scheme [50]. We briefly overview this numerical integration algorithm.

The deformation/motion dynamics equations at time $t + \Delta t$ can be written as:

$$M\ddot{U}_{t+\Delta t} + C\dot{U}_{t+\Delta t} + KU_{t+\Delta t} = f_{t+\Delta t} \tag{4.4}$$

The position and velocity at time $t + \Delta t$ can be approximated as:

$$\begin{aligned} \dot{U}_{t+\Delta t} &= \dot{U}_t + [(1-\delta)\ddot{U}_t + \delta\ddot{U}_{t+\Delta t}]\Delta t \\ U_{t+\Delta t} &= U_t + \dot{U}_t\Delta t + [(\frac{1}{2} - \alpha)\ddot{U}_t + \alpha\ddot{U}_{t+\Delta t}]\Delta t^2 \end{aligned} \tag{4.5}$$

where $\delta \geq 0.50$ and $\alpha \geq 0.25(0.5+\delta)^2$ guarantee the numerical stability of the numerical integration [50]. Constants can be calculated as follow:

$$\begin{aligned} &a_0 = \frac{1}{\alpha\Delta t^2};\ \ a_1 = \frac{\delta}{\alpha\Delta t};\ \ a_2 = \frac{1}{\alpha\Delta t};\ \ a_3 = \frac{1}{2\alpha} - 1; \\ &a_4 = \frac{\delta}{\alpha} - 1;\ \ a_5 = \frac{\Delta t}{2}(\frac{\delta}{\alpha} - 2);\ \ a_6 = \Delta t(1-\delta);\ \ a_7 = \delta\Delta t \end{aligned} \tag{4.6}$$

The equations in 4.5 can be solved for $\dot{U}_{t+\Delta t}$ and $\ddot{U}_{t+\Delta t}$ as follows:

$$\begin{aligned} \ddot{U}_{t+\Delta t} &= a_0(U_{t+\Delta t}) - a_2\dot{U}_t - a_3\ddot{U}_t \\ \dot{U}_{t+\Delta t} &= \dot{U}_t + a_6\ddot{U}_t + a_7\ddot{U}_{t+\Delta t} \end{aligned} \tag{4.7}$$

Substituting 4.7 into 4.4, and considering $\hat{K} = K + a_0M + a_1C$ and $\hat{f}_{t+\Delta t} = f_{t+\Delta t} + M(a_0U_t + a_2\dot{U}_t + a_3\ddot{U}_t) + C(a_1U_t + a_4\dot{U}_t + a_5\ddot{U}_t)$, it can be shown that:

$$\hat{K}U_{t+\Delta t} = \hat{f}_{t+\Delta t} \tag{4.8}$$

Step 2 and 3 involve large sparse matrix by vector (SpMV) multiplications. SpMV is known as a memory-bounded problem [57]. In dense operations, a CUDA kernel can be designed such that the memory divergence is decreased. This will happen when threads within a warp load(store) from(to) memory locations that are not far from each other. Scattered access can be avoided in dense matrix by vector multiplication. For instance, when all threads in a warp (32 threads) access the first 32 elements of an array in order, the number of memory transactions would be equal to 1. This would result in a fully coalesced access, with a data transfer rate that would be close to the peak global memory bandwidth. Unlike dense operations, SpMV operation has non-coalesced memory access patterns. In this section, we compare well-known compact storage formats for sparse matrices. These storage formats are designed to store non-zeros and their locations in a compact format, avoiding non-coalesced memory access in sparse matrix operations.

Diagonal Format: This format is suitable for sparse matrix representation when the nonzero elements are matrix diagonals (may include other diagonals than the main). Instead of storing the whole sparse matrix, only two arrays, one for nonzero

elements (values) and one for the offset of each diagonal from the main diagonal (offsets) are stored. This representation is not capable of handling matrices that do not have sparse diagonal pattern.

ELLPACK Format: A more general representation for an $i \times j$ sparse matrix is the ELLPACK (ELL) in which two $i \times k$ matrices (values and indices) are stored, where k is the maximum nonzero elements over all rows. In order to construct the value matrix, all zeros are removed and the nonzero elements are shifted to the leftmost location so the width of the value matrix is equal to the maximum number of nonzero elements per row. The indices matrix has an identical structure to the value matrix and its elements are the corresponding column index of the value matrix elements. This sparse representation format is not suitable for cases in which the ratio of k to the average nonzero elements per row is a large number.

Compressed Sparse Row (CSR) Format: This is a popular efficient storage scheme for sparse matrices. Assuming an $i \times j$ sparse matrix with nnz nonzero elements, one value array and one indices array of length nnz are stored. The indices array indicates the columns of nonzero elements. In addition to these two arrays, a third array of row pointers that has length $i + 1$ is used. The first i elements of this array hold the indices of the first nonzero elements row and the last element of this array is equal to nnz. The example below illustrates this sparse matrix representation.

$$A = \begin{bmatrix} 0 & 2 & 0 & 3 \\ 1 & 0 & 0 & 0 \\ 0 & 7 & 4 & 0 \\ 0 & 0 & 0 & 5 \end{bmatrix}$$

$$data = [2 \quad 3 \quad 1 \quad 7 \quad 4 \quad 5]$$

$$col = [1 \quad 3 \quad 0 \quad 1 \quad 2 \quad 3]$$

$$ptr = [0 \quad 2 \quad 3 \quad 5 \quad 6]$$

Using ptr array, the number of nonzero elements in i^{th} row is equal to $ptr[i+1] - ptr[i]$. We have used the CSR format for sparse matrix-vector multiplication because it allocates less memory to store our sparse matrices.

Consider $Ax = y$ where A is a sparse matrix and x and y are in general dense vectors. The serial algorithm [57], which requires the number of rows of A matrix and the three CSR arrays plus x and y vectors, can be summarized as follows:

- Loop over the rows of the sparse matrix (A matrix).
- Find the number of nonzero elements per row by using the ptr array to indicate the start and end points for an inner loop.
- Use an inner loop to calculate the dot product of nonzero elements per row and their corresponding elements in x vector.

Since all rows can independently perform the dot product, a thread can be assigned to each row so that the CUDA kernel performs the serial dot product loop

(inner loop) in parallel for all rows of the A matrix; therefore the outer loop is removed from the algorithm. This kernel is known as "CSR scalar kernel". When the numbers of nonzero elements per row are highly variable, it is probable that some threads in a warp (which are assigned to rows with less nonzero elements) are idle and waiting for other threads in that warp (which are assigned to rows with more nonzero elements) to finish their inner product.

In order to resolve the CSR scalar kernel problem, "CSR vector kernel" has been developed [57]. In the CSR vector kernel, one warp is assigned to each matrix row. In the scalar kernel only one thread per row performs the inner product, whereas, in the vector kernel multiple threads in a warp perform the inner product and then a parallel reduction is carried out to sum the result of each thread in a warp. Parallel reduction in CUDA is discussed in Step 5 of this chapter. Although the CSR vector kernel alleviates the thread divergence problem of the scalar kernel, it has been shown that the scalar kernel provides better performance when the number of nonzero elements per row is less than 32 [57]. This might be due to the fact that each warp consists of 32 threads. Therefore in the vector kernel, some threads follow another instruction path resulting in thread divergence. Hence the performance might be decreased in the case where the number of nonzero elements per row is less than 32.

CUSPARSE library has been used for sparse operations in Step 2 and 3 of our registration problem. In Step 3, the sparse matrices are constant in all iterations. These matrices are derived from the finite element mesh (such as M, C, and K in 3.11) and can be converted to CSR format off-line; therefore the conversion time does not affect real-time performance of our registration process. In Step 2, the

structure of the sparse matrix is constant in all iterations but its values are changing. This matrix is derived from the linear shape function of tetrahedral elements (see 3.15). Since the coefficients of this function change per iteration, the values of the matrix also change from one iteration to another. The two arrays of the CSR format (ptr and col) can be calculated off-line. Based on these two arrays, a CUDA kernel has been written to fill and update the nonzero elements of the sparse matrix in parallel. The values of our sparse matrix are updated on the fly as the algorithm progresses.

From a computational perspective, the Newmark method can be divided into "initial calculations" and an "iterative" part. The initial calculations which are carried out off-line include U_o , $\dot{U}_0$, $\ddot{U}_0$ and the parameters in 4.6. Also the effective stiffness matrix $\hat{K}$ and its inverse are computed off-line.

The online computations in the iterative steps of the algorithm are summarized as follows:

- Calculation of $\hat{f}$ which requires two SpMV and multiple vector additions (see $\hat{f}_{t+\Delta t}$ in Section 4.4.3). In addition to SpMV, a kernel has been written to add all vectors in parallel.

- Computation of the nodal displacement which requires a dense matrix by vector multiplication. Equation 4.8 has been used to calculate $U_{t+\Delta t}$, the inverse of $\hat{K}$ is computed off-line and remains constant in all iterations.

- Calculation of the velocity and acceleration in 4.7 which requires multiplication of scalar coefficients and vectors, and vector addition/subtraction operations.

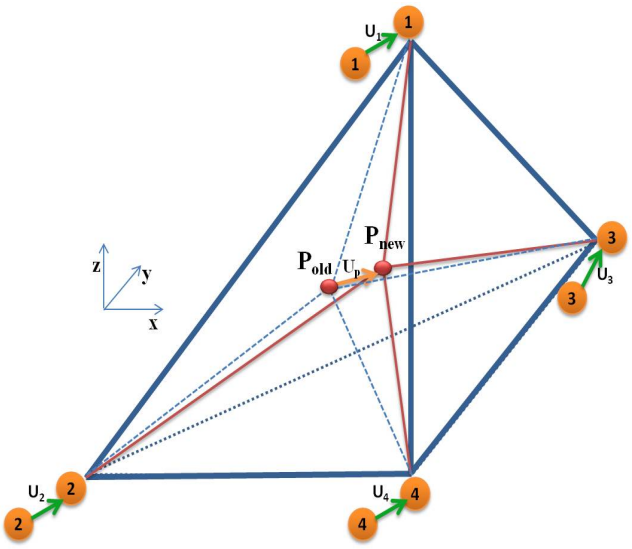


Figure 4.11: Linear shape function on tetrahedral elements.

4.4.4 Step 4: Displacement of 3D Grid

Having computed the displacement of the nodal points (u), the deformed 3D grid in the template image can be obtained from the deformed finite element mesh using the linear shape function of the tetrahedral elements. In this section, we discuss how the linear shape function on tetrahedral elements (3.15) can be implemented on GPU.

The displacement of point p in Fig. 4.11, u_p, can be approximated as a linear shape function [58]:

$$u_p(x_p, y_p, z_p) = a + bx_p + cy_p + dz_p \tag{4.9}$$

Assuming u_1, u_2, u_3, and u_4 are the displacement vectors of four vertices of the tetrahedral element, the following system of equations can be constructed:

$$\begin{aligned} u_1 &= a + bx_1 + cy_1 + dz_1 \\ u_2 &= a + bx_2 + cy_2 + dz_2 \\ u_3 &= a + bx_3 + cy_3 + dz_3 \\ u_4 &= a + bx_4 + cy_4 + dz_4 \end{aligned} \tag{4.10}$$

Solving 4.10 for the four coefficients a, b, c, and d yields:

$$\begin{aligned} a = \quad & \frac{1}{6V}\begin{vmatrix} x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{vmatrix} u_1 - \frac{1}{6V}\begin{vmatrix} x_1 & y_1 & z_1 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \end{vmatrix} u_2 \\ & + \frac{1}{6V}\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_4 & y_4 & z_4 \end{vmatrix} u_3 - \frac{1}{6V}\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} u_4 \\ & = a_1u_1 + a_2u_2 + a_3u_3 + a_4u_4 \end{aligned} \tag{4.11}$$

$$
\begin{aligned}
b = &-\frac{1}{6V}\begin{vmatrix} 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} u_1 + \frac{1}{6V}\begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_3 & z_3 \\ 1 & y_4 & z_4 \end{vmatrix} u_2 \\
&-\frac{1}{6V}\begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_4 & z_4 \end{vmatrix} u_3 + \frac{1}{6V}\begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} u_4 \\
&= b_1u_1 + b_2u_2 + b_3u_3 + b_4u_4
\end{aligned}
\tag{4.12}
$$

$$
\begin{aligned}
c = &\frac{1}{6V}\begin{vmatrix} 1 & x_2 & z_2 \\ 1 & x_3 & z_3 \\ 1 & x_4 & z_4 \end{vmatrix} u_1 - \frac{1}{6V}\begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_3 & z_3 \\ 1 & x_4 & z_4 \end{vmatrix} u_2 \\
&+\frac{1}{6V}\begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_2 & z_2 \\ 1 & x_4 & z_4 \end{vmatrix} u_3 - \frac{1}{6V}\begin{vmatrix} 1 & x_1 & z_1 \\ 1 & x_2 & z_2 \\ 1 & x_3 & z_3 \end{vmatrix} u_4 \\
&= c_1u_1 + c_2u_2 + c_3u_3 + c_4u_4
\end{aligned}
\tag{4.13}
$$

$$
\begin{aligned}
d = &-\frac{1}{6V}\begin{vmatrix} 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} u_1 + \frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_3 & y_3 \\ 1 & x_4 & y_4 \end{vmatrix} u_2 \\
&-\frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_4 & y_4 \end{vmatrix} u_3 + \frac{1}{6V}\begin{vmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{vmatrix} u_4 \\
&= d_1u_1 + d_2u_2 + d_3u_3 + d_4u_4
\end{aligned} \tag{4.14}
$$

and V is the volume of the tetrahedral given by:

$$
V = \frac{1}{6}\begin{vmatrix} 1 & x_1 & y_1 & z_1 \\ 1 & x_2 & y_2 & z_2 \\ 1 & x_3 & y_3 & z_3 \\ 1 & x_4 & y_4 & z_4 \end{vmatrix} \tag{4.15}
$$

Considering the linear shape function on tetrahedral elements 4.9, and Equations 4.11 to 4.14, the displacement vector of point p inside the tetrahedron can be written as:

$$
u_p(x_p, y_p, z_p) = \begin{bmatrix} u_{1x} & u_{2x} & u_{3x} & u_{4x} \\ u_{1y} & u_{2y} & u_{3y} & u_{4y} \\ u_{1z} & u_{2z} & u_{3z} & u_{4z} \end{bmatrix} \begin{bmatrix} a_1 & b_1 & c_1 & d_1 \\ a_2 & b_2 & c_2 & d_2 \\ a_3 & b_3 & c_3 & d_3 \\ a_4 & b_4 & c_4 & d_4 \end{bmatrix} \begin{bmatrix} 1 \\ x_p \\ y_p \\ z_p \end{bmatrix} \tag{4.16}
$$

The order of multiplication in 4.16 is an important factor in increasing the speed

of computation. $u_p(x_p, y_p, z_p)$ can be computed in one of the following two ways:

1. A matrix by matrix multiplication and then one matrix by vector multiplication. This approach requires 60 multiplications and 45 additions for each 3D grid point.

2. Two matrix by vector multiplications. This approach requires 28 multiplications and 21 additions for each 3D grid point.

The second approach has been chosen for Step 4. Two CUDA kernels perform the matrix by vector multiplications. For each 3D grid point, the corresponding tetrahedron and its four nodal points must be extracted. These nodal points have been used multiple times in the kernel to load the coefficients and directional displacements from the global memory. It should be noted that the nodal points have been stored in registers to minimize global memory access and increase the performance of the algorithm. At the end, the final result, i.e. displacements of 3D grid points, are added to the positions of the 3D grid points to calculate their new positions. The process can be summarized as follows:

Kernel 1

1. The position of each 3D grid point and its corresponding coefficients are loaded from global memory into registers.

2. Four threads are assigned to each 3D grid point to compute a 4×4 matrix by a 4×1 vector multiplication and store the result in C, a 4×1 vector (see 4.16).

Kernel 2

1. For each 3D grid point, the corresponding elements of the C vector and nodal displacements are loaded into registers.

2. Three threads are assigned to each 3D grid point to compute a 3×4 matrix by a 4×1 vector multiplication. The result represents the displacement of the 3D grid point.

3. The new position of each 3D grid point is obtained by adding the old position of each 3D point to the result that is obtained in Step 2 of Kernel 2 (i.e. previous step).

In contrast with the GPU implementation which is capable of computing all above steps in parallel, the CPU implementation computes the 3D displacement of grid points in serial. The displacement of each grid point is a 3D vector representing the displacement in three separate directions. Therefore not only the calculations of displacement vector for each grid point can be parallelized, but also the directional displacement can be computed independently. This will take the maximum advantage of GPU execution resources.

4.4.5 Step 5: Distance Measure

The SSD between the deformed template and the reference image is computed in this step. The iterations stop when the relative error in the SSD falls below a threshold, i.e.

$$g(u) = \frac{|SSD(u_t) - SSD(u_{t-1})|}{SSD(u_{t-1})} < \delta \tag{4.17}$$

where u_t and u_{t-1} are the displacement vectors at the current and previous iterations, respectively. First, a CUDA kernel computes the squared difference between the two vectors and stores the result into a third vector. Then another kernel calculates the sum of the elements of the third vector. In the first kernel, one thread is assigned to each element of the 3D data which are projected into a 1D array. This thread calculates the squared difference between one element in the deformed template and its corresponding element in the reference image. The tasks consists of the following steps:

- The corresponding elements from the deformed template and reference images are loaded from global memory in to registers.
- One thread computes the squared difference of the two elements and stores the result in a third vector.

A reduction algorithm then is used in the second kernel to calculate the sum of the elements in the result vector. The algorithm can be summarized as follows:

- One thread block is assigned to one section of the array (in GTX 480, up to 1024 threads can be placed in a block)
  - Each thread loads one element from global memory into shared memory.
  - Barrier synchronization statement (__syncthreads()) ensures that the loading task is completed for all threads within a block.
  - Parallel reduction task is performed using a tree-based approach (see Fig. 4.12) in shared memory. An array with eight elements is shown in

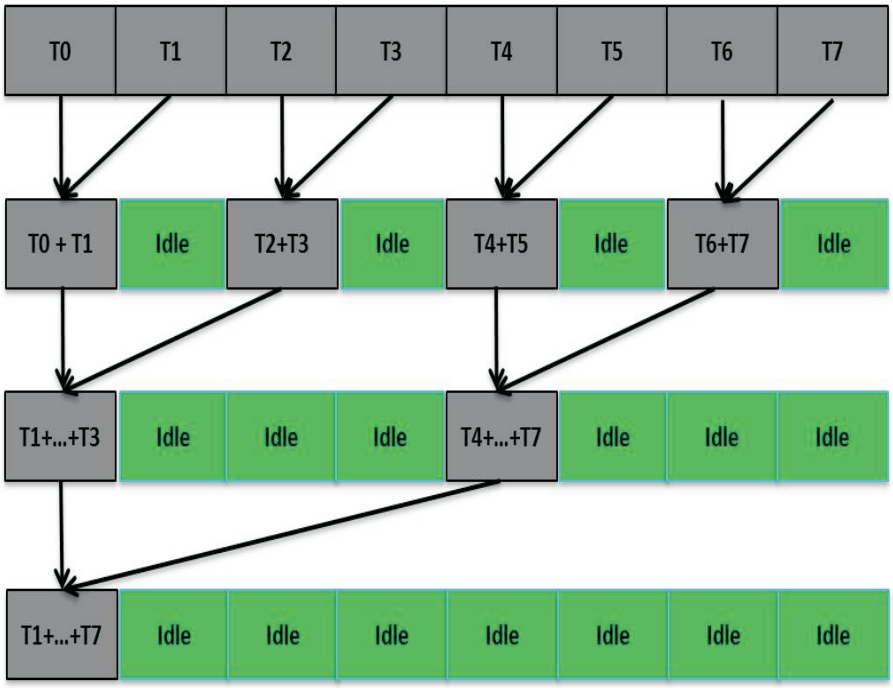


Figure 4.12: Parallel sum reduction.

Fig. 4.12; three steps are required to calculate the sum of its elements. In general, the algorithm requires $\log_2(n)$ steps to perform this task for an array with $n = 2^k$ elements, $k = 0, 1, 2, ...$ Active threads compute partial sums in each step and the number of active threads is reduced from one step to another.

- __syncthreads() is used again to ensure that the partial sums are completed in one step before running the next step.

- The results from multiple blocks are added together to calculate the final result. If the array is very large, the same parallel sum reduction algorithm can be used to find the final result.

Fig. 4.12 illustrates the parallel sum reduction kernel. Since the values are stored in shared memory, threads that are not idle have access to the previous results. When all non-idle threads complete one step, the stride is multiplied by 2 for the next iteration.

As mentioned in Section 4.2.1, the GPU executes an instruction for all threads that are grouped in a warp. In Fig. 4.12, we have 8 threads that are grouped in a warp. Obviously the kernel has an if statement and that is why some of the threads execute one instruction while others are idle. This causes a divergence in the execution paths. In fact some threads follow one path, while others follow a different one. Since the GPU handles these paths in a sequential form, the "thread divergence" can add to the execution time.

The above situation decreases the performance especially when the array is large. A change in the position of threads, which perform the reduction task, can alleviate the thread divergence problem. Fig. 4.13, shows a revised parallel sum reduction method [12]. Instead of considering two threads that are neighbors (most likely in one warp), two threads that are far from each other (most likely in two separate warps) have been chosen to calculate the sum in each iteration. Therefore, threads that are in a warp follow the same instruction and this will reduce the thread divergence in most steps. Although in most iterations we do not have thread divergence, this problem still exists in the final steps when threads in a warp do not follow the same path. "cublasSasum" function can also be used to perform the summation task.

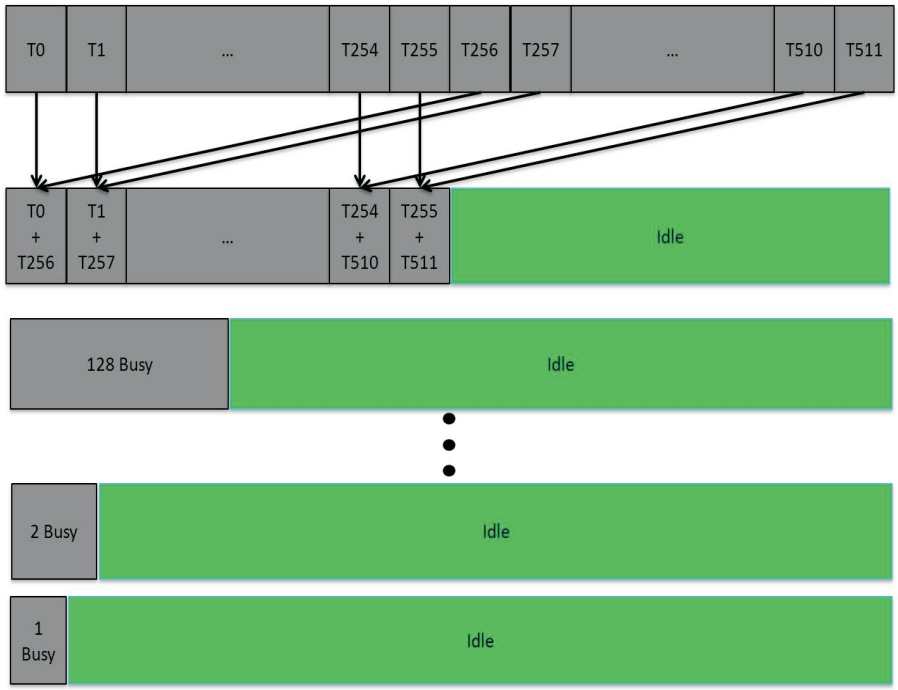


Figure 4.13: Revised parallel sum reduction.

4.4.6 Other Computations

In addition to the above steps, there are some other computations that require matrix and vector operations. Although the runtime of these tasks is much less than the runtime of the five main steps, their time has been considered in the results presented in Chapter 5.

One time consuming task that must be completed before the conjugate gradient in Step 2 is computing the coefficients that are used to find the inverse of the shape function. The coefficients are needed in the calculation of the matrix A in the conjugate gradient algorithm. They are computed according to Equations 4.11 to 4.14. Since the position of nodal points change per iteration, these coefficients

must be updated in each iteration. The task has been parallelized for each tetrahedral elements. A "device function" has been written for 3×3 determinant and is called inside a kernel to calculate sixteen coefficients per element $a_1, a_2, ..., d_4$. The positions of the nodal points are placed in registers to decrease the global memory access. The process of computing $a_1, a_2, ..., d_4$ coefficients can be shown as follows:

- Nodal points' positions are loaded from global memory into registers for each finite element element.
- A device function which is written to perform the 3×3 determinant is called to compute the coefficients.
- The results are written in a 1D array in the device.

Chapter 5

Performance Analysis, Optimization and Experimental Results

In this chapter, we present the performance results of the GPU kernels that are implemented in different steps of the registration algorithm. We also study the performance of other kernels such as matrix by matrix multiplication to explain the techniques that can be used for design optimization. A comparison between GPU and CPU-based registration also has been conducted to demonstrate the GPU capability in accelerating the image registration algorithm.

We have used the MR image sets of abdomen from [59] shown in Fig. 5.1 to illustrate the performance of the GPU-based implementation and how such performance can be optimized across different GPU kernels. The GPU execution resources, high-bandwidth memories, and CUDA flexible programming techniques have helped us significantly accelerate our computationally intensive image registration algorithm.

We studied the performance of the proposed implementation in accelerating

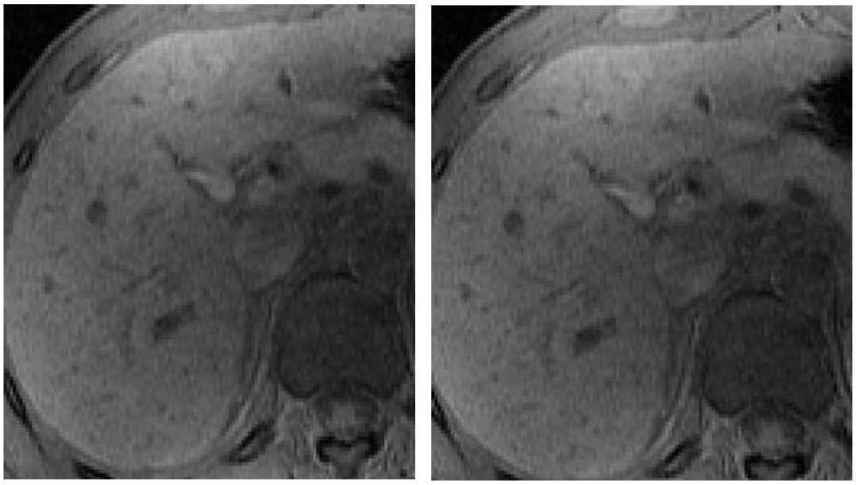


Figure 5.1: MR images of abdomen showing the liver; the reference (right) and template (left) images.

the whole registration algorithm using a realistic breast phantom tissue. The main advantage of this data sets over the abdominal MR images is that it involves large deformation. Therefore, both the algorithm and its GPU implementation can be examined in the case of such large tissue deformations are relevant in applications such as breast biopsy.

5.1 Computing Performance as a Function of 3D Grid Size

As discussed in Sections 3.1 and 3.3, a regular 3D grid has been used in the registration algorithm. The displacement and gray value of the grid points must be calculated at each iteration of the algorithm. The size of the 3D grid and the resolution of the finite element mesh affect the outcome of the registration. In general, the accuracy can be improved by increasing the resolution of the finite element

mesh and number of 3D grid points since more image information is available in a finer grid. Obviously there is a trade-off between the runtime of our GPU kernels and the size of the 3D grid. The appropriate choice of the 3D regular grid size depends on the application and available computational resources.

In this section, we explain how the size of the 3D regular grid can directly affect the interpolation (Step 1) and displacement (Step 4) kernels. It should be noted that the grid size also affects the size of some other matrices and vectors in other steps of the algorithm. We compare the performance of two GPU 3D interpolation kernels, with and without texture memory, with the performance of a serial implementation in the trilinear interpolation of a $512 \times 512 \times 136$ image. The serial code is written in standard C and runs on a 3.20 GHz Intel core i5 650 processor. The results are given in Table 5.1. In all three implementations, the computation time

3D grid size	GPU (Texture)		GPU (without Texture)		CPU	
	Time (ms)	GFLOP/s	Time (ms)	GFLOP/s	Time (ms)	GFLOP/s
64×64×25	0.15	29.35	0.49	8.98	47	0.09
128×128×50	0.28	125.80	0.92	38.28	251	0.14
256×256×50	0.52	270.96	1.68	83.87	547	0.25

Table 5.1: The performance of trilinear interpolation using GPU with texture memory, GPU without texture memory, and CPU for different 3D grid sizes.

and GFLOP/s increases by increasing the 3D grid size. Although GPUs are able to run thousands of threads in parallel, execution resources are limited. Therefore in a large 3D grid size, not all the tasks can be performed in parallel. In fact, some

tasks should wait until execution recourses become available. It should be noted that the relationship between the 3D grid size and the execution time is not linear in all three implementation. For instance when the grid size becomes four times larger, the execution time does not increase by a factor of 4. One may argue that the computation time should remain unchanged for a parallel implementation. This is not true since the execution resources, e.g. number of resident threads per multiprocessor, are limited in GPU hardware. In fact, the interpolation task cannot be performed for all grid points at a same time when we have a large 3D grid which requires more than the maximum amount of execution resources. It is evident that one can achieve significant speed up by using the hardware built-in 3D texture trilinear interpolation feature of the GPU. In contrast with the CPU implementation which computes the values of the regular 3D points sequentially, both GPU implementations provide a parallel approach to calculate the trilinear interpolation for all regular points.

It should be noted that, the eight voxels that are involved in computing the value of a regular point (see Fig.4.7) are not placed consecutively in the global memory. GPU texture memory increases the performance of the interpolation kernel by providing a different access pattern which takes the spatial locality of those voxels into account. It should be also noted that by this time the texture interpolation feature is not available for double floating-point data.

The size of the 3D regular grid also affects the displacement kernel in Step 4. The performance of this kernel as function of the grid size for a volumetric mesh with 2335 tetrahedral elements and 515 nodes, encompassing a $512\times512\times136$ image (abdominal MRI in Fig. 5.1), is shown in Table. 5.2. As we can see in the table,

3D grid size	CPU (ms)	GPU (ms)	Speed Up
128×128×50	312	2.15	145.11
64×64×50	78	0.59	132.2
32×32×40	16	0.18	88.88
16×16×20	2	0.096	20.83

Table 5.2: The performance of the kernel that calculates the displacements of 3D grid points as a function of the grid size.

there is almost a linear relationship between the 3D grid size and the CPU serial implementation. For instance when the grid size becomes 8 times larger, i.e. from $16 \times 16 \times 20$ to $32 \times 32 \times 40$, the CPU times increases by a factor of 8. In this case, the time is changed by a factor of 1.8 in our GPU; meaning that there is not a linear relationship between the 3D grid size and the GPU runtime. As the 3D grid size becomes larger, the relationship between the GPU runtime and the 3D grid size becomes almost linear. For instance, when the grid size changes from $64 \times 64 \times 50$ to $128 \times 128 \times 50$, i.e. 4-fold increases in the grid size, the GPU runtime changes from 0.59 to 2.15 milliseconds, i.e. 3.64-fold time increases. This might be due to large amount of registers that were employed to implement the displacement kernel. Using additional registers can reduce kernel performance.

5.2 Shared Memory and Performance

In this section, we compare three approaches in matrix by matrix multiplication kernel for multiple dimensions. We report the performance and runtime of the

multiplication kernel with global memory, shared memory and cublasSgemm from the CUBLAS library. Fig. 5.2 shows that using shared memory we can achieve higher performance compared to the matrix multiplication kernel in which only global memory is used. As mentioned in the previous chapter, shared memory reduces the global memory access and improves the performance by reducing its runtime. In fact, the kernel is designed such that the number of loadings from global memory into shared memory for a given matrix element is much less than the number of floating point calculations performed on that matrix element. CUBLAS not only employs shared memory but also uses other optimization techniques to enhance the multiplication performance. CUBLAS outperforms other two GPU implementations at larger matrices (see Fig. 5.2 and Table. 5.3). In this section some possible techniques that might have been used in CUBLAS library are analyzed. We also discuss about possible reasons that may cause CUBLAS to have less performance than the two others in performing multiplication for small matrices.

In contrast with the shared and global kernels, CUBLAS employs non-square blocks; yielding in better usage of execution resources (e.g. thread slots and block slots). CUBLAS also provides better memory access pattern (this will be discussed shortly). These features might be the reasons that the global and shared kernels saturate earlier than CUBLAS; i.e. when the dimension becomes more than 512. In the case when the matrix dimension is 512×512, the block size is set to 16×16=256 threads in both the global and shared kernel. Since the maximum resident threads in a multiprocessor is 1536, $\frac{1536}{256} = 6$ blocks can be placed in each multiprocessor. Because there are 15 multiprocessor in a GTX 480, 15×6 = 90 blocks can reside in the GPU. It should be noted that the total number of blocks in this case is 1024 (i.e.

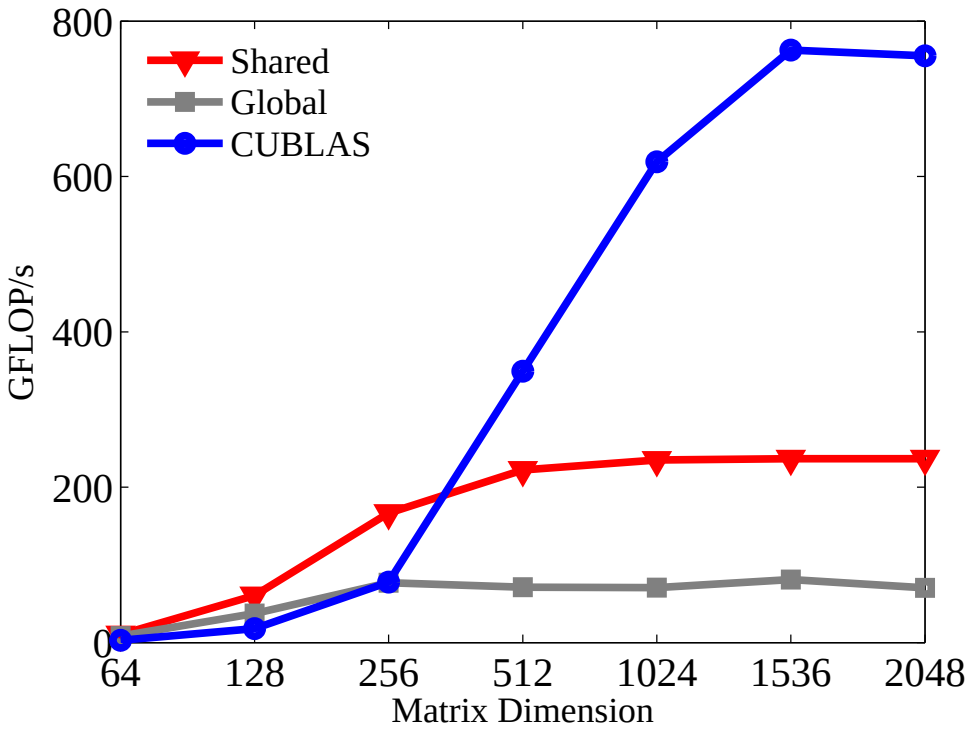


Figure 5.2: Performance results on multiplying two N×N matrices on GTX 480.

$\frac{512}{16} = 32$ in each dimension) but only 90 blocks can reside in the multiprocessors at a same time. Many blocks should wait until the resources become available. This waiting time is much more than the waiting time in a case when smaller matrices (e.g., 256×256) are considered. Assuming the same block size, the total number of blocks in the case when the matrix dimension is 256×256 is $\frac{256}{16} = 16$ in each dimension yielding in a total of 256 blocks.

In the matrix multiplication kernel that uses shared memory (see Section 4.4.2), there is a dot product for each thread in a block. This requires a loop to compute the dot product of a subset of matrix elements with the corresponding subset of another matrix. The loop adds extra instructions such as loop counter update

and performing conditional branch which increase the kernel runtime. The loop unrolling technique can be employed to eliminate those extra instruction and enhance the performance. Loop unrolling may result in more than 20% performance improvement [12].

Matrix Size	Global Memory		Shared Memory		CUBLAS	
	Time (ms)	GFLOP/s	Time (ms)	GFLOP/s	Time (ms)	GFLOP/s
64×64	0.063	8.257	0.051	10.199	0.188	2.766
128×128	0.112	37.302	0.069	60.549	0.233	17.930
256×256	0.435	76.985	0.201	166.611	0.430	77.881
512×512	3.763	71.265	1.207	222.181	0.768	349.184
1024×1024	30.321	70.790	9.137	234.916	3.469	618.747
1536×1536	89.352	81.088	30.614	236.669	9.501	762.593
2048×2048	243.491	70.539	72.552	236.736	22.742	755.24

Table 5.3: The Performance of matrix multiplication with different approaches in GPU.

Another technique is data prefetching in which additional automatic variables (registers) are used to load the next step data from the global memory into registers right before the dot product computation (see Section 4.4.2). These registers hold data that will be consumed in next iteration, thus the amount of time that is required for global memory access is reduced. The result of multiplication kernel

with CUBLAS has been shown in Fig. 5.2 and Table. 5.3. Although the cublasSgemm function employs shared memory and optimization techniques to gain performance close to peak performance (1.35 Tflops in GTX 480), the shared memory kernel provides better performance for smaller matrices. As mentioned before, registers decrease the amount of time that is required for global memory loading. On the other hand, additional registers can decrease the number of resident blocks on a multiprocessor [12]. Therefore there is a trade-off between the number of resident blocks and the time each thread waits for global memory loading. One possible reason that CUBLAS library has less performance than the two others in performing multiplication for small matrices might be due to insufficient use of block and thread slots. As discussed in Section 4.2.1, underutilization of these resources can decrease the performance of the kernels.

5.3 Memory Transfer and Performance

In most GPU implementations, it is required to exchange data between CPU (host) and GPU (device). In this section we show the host memory configuration can affect the overall computation performance. In our implementation, we have data transfers at the beginning and end of our algorithm. We have chosen the kernel for calculating the displacement of 3D grid point as an example to illustrate the dependency of the kernel runtime on the type of data transfer and memory configuration. The memory transfer time is included in the overall reported runtime of the GPU implementation.

Fig. 5.3 shows the execution time for our kernel based on different host memory configurations. The memory transfer portion, memcpy in the figure includes

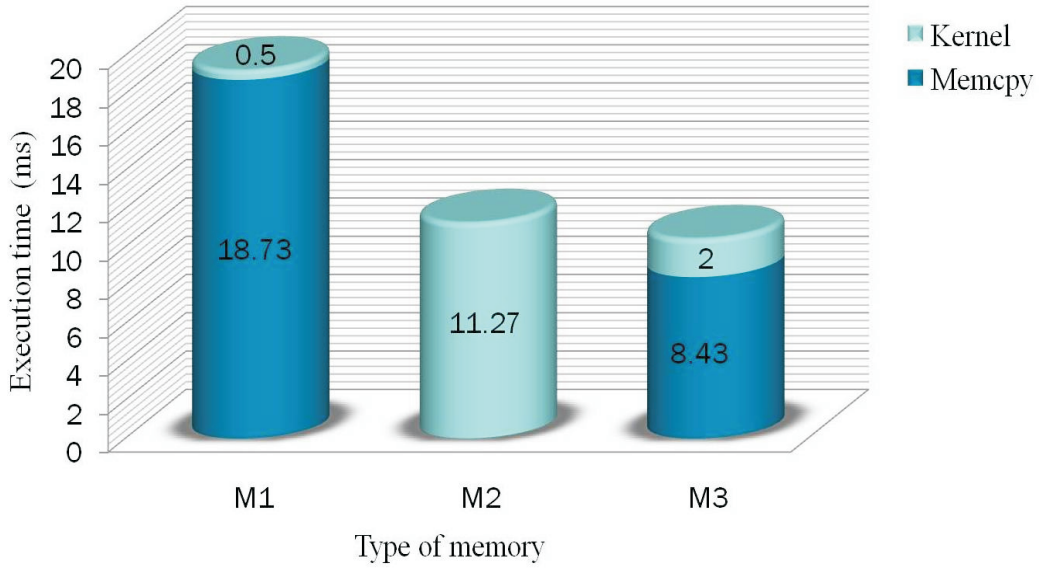


Figure 5.3: Total execution time (memcpy + kernel) is shown for the displacement. Three types of host memories have been used. M1 is the regular pageable memory, M2 is the mapped memory and M3 is the write-combined memory.

both CPU (host) to GPU (device) and GPU to CPU transfers. The regular pageable memory, M1, has the highest runtime of 19.32 ms which includes the kernel time plus the memory transfer time. The mapped memory, M2, maps a block of page-locked host memory into the device. This block has two addresses in host and device thus, in contrast with M1, the programmer does not need to explicitly allocate a block in GPU for memory transfer. In fact this data transfer is implicitly performed by the kernel and its time is included in the kernel time. The write-combined memory configuration, M3, exhibits the best performance of all three with a total runtime of 10.43 ms. M3 transfers data across the Peripheral Component Interconnect Express (PCIe) more quickly and increases the performance especially when GPU reads the buffer that is written by CPU.

5.4 Thread Organization and Performance

GPU provides support for massive parallel computing but the execution resources are obviously limited and shared between the execution threads. The block size and grid size (Section 4.1 and 4.2) can have a great influence on the execution performance. CUBLAS and CUSPARSE functions automatically set block size and grid size. The performance of other kernels in our implementation depend on their block and grid size. The performance of a few possible CUDA thread organizations in the same kernel of the algorithm as in Section 5.3 are compared in Table. 5.4.

Configuration 3 has the best performance in terms of kernel execution time. This is due to the fact that the kernel can fully occupy each multiprocessor in our

Config	Block Size (B), Grid Size (G)	GFLOP/s	time (ms)
1	B(1,64), G(1,38400)	25.84	1.87
2	B(1,128), G(1,19200)	28.94	1.67
3	B(1,256), G(1,9600)	29.65	1.63
4	B(1,512), G(1,4800)	26.55	1.82
5	B(1,1024), G(1,2400)	22.48	2.15

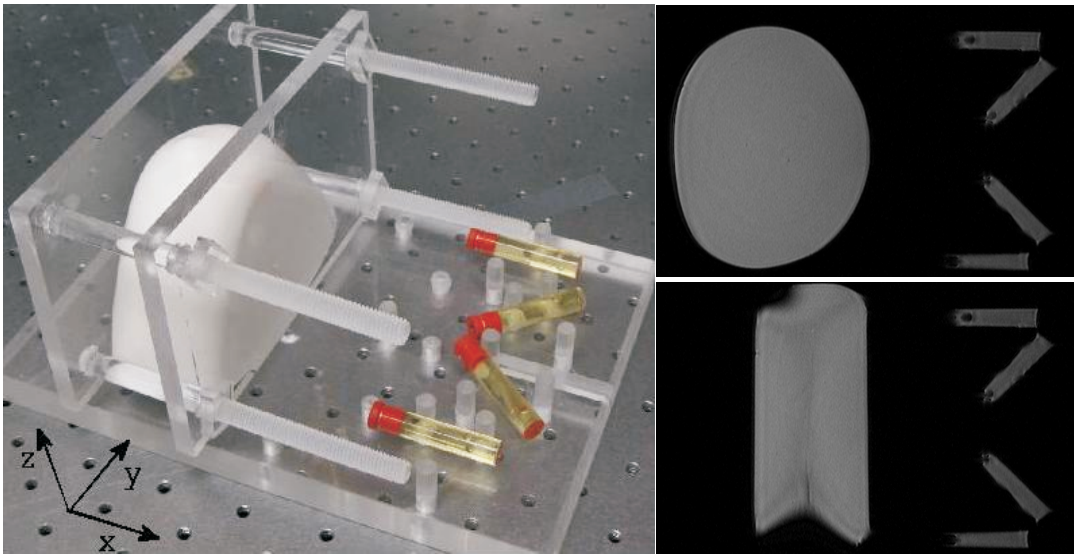
Table 5.4: GPU runtime and GFLOP/s of the CUDA kernel for 3D grid displacement field.

GPU. The maximum number of resident threads per multiprocessor is 1536 in a GTX 480. Each multiprocessor can have up to 8 resident blocks. In Configuration 3, we have 1536/256 = 6 blocks which is less than 8; each block has 256 threads, hence $6\times256=1536$ threads can fully occupy the thread slots of a multiprocessor.

Configurations 1 and 2 are less effective than Configuration 3 due to insufficient use of threads. For instance in Configuration 2, each block has 128 threads and 1536/128 = 12 blocks are needed to fully occupy a multiprocessor; but we can only have up to 8 blocks resulting in 8×128 = 1024 threads, less than the maximum number of resident threads which is 1536 in a GTX 480. Similarly Configurations 4 and 5 are not as efficient because of inefficient use of execution resources. It should be noted that these limits are device dependent.

5.5 3D-3D MR Registration of Breast Phantom

A triple modality biopsy training breast phantom (CIRS model 051) has been used for obtaining 3D volume high-resolution (512×512×136) MR images in both the un-deformed and the deformed states. Fig. 5.4.a shows an MR compatible plexiglass structure with four mounted capsules of vitamin E as landmarks to match the coordinates of the deformed and un-deformed images. Fig. 5.4.b illustrates the deformed and un-deformed images. The goal of the algorithm is to deform the un-deformed image so that the two image sets can be aligned. COMSOL Multiphysics and Simulation software is used just to create a cubic finite element mesh of 7502 linear tetrahedral elements with 1601 nodal points. The deformation model serves as a tunable constraint on the registration process and therefore exact material and geometrical properties of the tissue are not required in the algorithm. The cubic FE mesh encompasses the whole volume of deformed and un-deformed data and a 20×30×10 regular grid with no need for image segmentation.



(a) (b)

Figure 5.4: (a) the apparatus, (b) x-y views of un-deformed (top) and deformed (bottom) images.

5.5.1 Execution Time

The registration algorithm converges to a solution after 15 iterations. The CPU execution time for the 3D high resolution image registration is 82.83 sec. compared to only 2.19 sec. for the GPU, i.e. a 37-fold speedup is achieved. The CPU implementation which is written in standard C runs on a 3.20 GHz Intel core i5 650 processor with 4GB RAM. The compositions of the GPU and CPU execution times are given in Table 5.5. It should be noted that the algorithm runtime in MATLAB is about 23 minutes with the same CPU.

Step 1 includes GPU kernels which compute the coefficients of the tetrahedral shape function (Equations 4.11 to 4.14). This step also involves four trilinear interpolation, exploiting the hardware built-in 3D texture trilinear interpolation. Step 2, which computes the force vector of nodal points, involves the conjugate gradient

Steps	GPU (ms)/itr	CPU (ms)/itr	Speed Up
1	2.21	892	403.61
2	120	2797	23.3
3	12.7	1828	143.93
4	0.11	4	36.36
5	0.65	1	1.53
GPU others	10.54	–	–
Total	146.21	5522	37.76

Table 5.5: Execution times for various steps of 3D/3 registration for $512 \times 512 \times 136$ MR images.

method (takes 70 ms) and some sparse matrix operations. Step 3 includes the implementation of the displacement of nodal points. As mentioned in the previous chapter, the dynamic finite element model is solved using the Newmark integration scheme in this step. CUSPARSE library has been used in Steps 2 and 3 for sparse operations. Linear shape function on tetrahedral elements 3.15, has been implemented in Step 4. SSD is computed in Step 5 and "GPU others" refers to memory transfer and other computations.

Step 2 of the algorithm is the most time consuming part in both the CPU and the GPU implementation. The highest speed-ups were achieved in Step 1 and 3 of the registration algorithm, respectively.

5.5.2 Quality of Registration

The 3D preoperative image in Fig. 5.5 (top left) and the 3D intraoperative image in Fig. 5.5 (top right) are the inputs of the registration algorithm. Fig. 5.5 (bottom

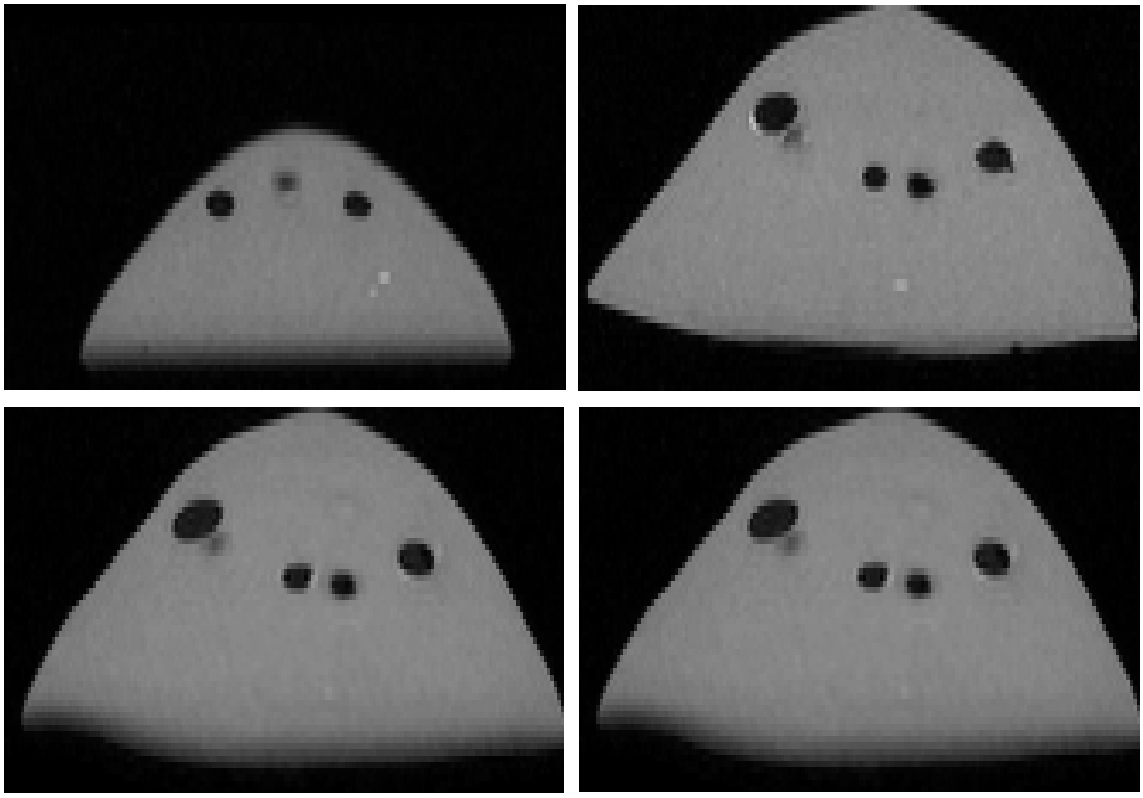


Figure 5.5: Visual comparison of GPU and CPU registration. 2D ($y-z$ view) slices of the preoperative (template) image (top left), intraoperative (reference) image (top right), deformed preoperative image after registration on GPU (bottom left) and deformed preoperative image after registration on CPU (bottom right). Note that the phantom has been compressed along x-axis (see Fig.5.4) which has caused image elongation in $y-z$ plane and also movements of image features in the image plane.

left) and Fig. 5.5 (bottom right) are the outputs of GPU and CPU-based implementations, respectively. The SSD between the intraoperative and the deformed preoperative data are 285.11 and 282.08 for the GPU and CPU implementation, respectively. These numbers are normalized by the image size.

The small difference between the GPU and CPU results is due to the double-precision floating point operations in CPU implementation versus the single-precision floating point operations in the GPU implementation. As mentioned before, some features such as linear filtering are only supported in single-precision operations on GPU. In this thesis, all kernels are designed for single-precision floating point operation.

5.6 Scalability of Solutions

CUDA programming environment provides flexible implementations. The grid size and block size and other execution and memory resources can be easily modified to compensate for various sizes of the input/output data. Our implementation can be run in any other CUDA capable GPU with compute capability 2.0 or more. In order to optimize the implementation to take full advantage of the recourses of newer GPUs, one should have a good knowledge of the GPU device.

Our implementation is capable of handling problems with larger sizes. For instance, the size of the finite element mesh, the resolutions of the reference and template images, and the 3D grid size can be easily increased by changing the memory sizes and the GPU block and grid sizes. This might be required when the accuracy of the image registration algorithm should be increased for a specific application.

Main Steps	3D-3D Registration algorithm	2D-3D Registration algorithm
On-line search algorithm	✗	✓
Trilinear interpolation	✓	✓
Force computation	✓	✓
Displacement of nodal points	✓	✓
Displacement of 3D grid	✓	✓
SSD	✓	✓

Figure 5.6: Main steps in 3D-3D and 2D-3D registration algorithms.

We believe that newer GPUs can be employed in the future to accelerate our implementation by adding more cores, providing more execution resources, and increasing the memory bandwidths.

5.7 GPU Implementation of the 2D-3D MR Registration

In this thesis, we designed, optimized and implemented GPU kernels to accelerate the 3D/3D deformable MR registration algorithm in [1]. An algorithm for 2D/3D deformable registration also is discussed in [1]. Fig. 5.6 illustrates the main steps of each registration algorithm. From a numerical and computational points of view, all the GPU kernels designed for the 3D/3D algorithm also applies to the 2D/3D method. As discussed in Chapter 3, the displacement of the 3D grid points must

be computed in each iteration of the 3D/3D algorithm. The 3D regular grid points are distributed within the finite element mesh and each point is located inside a tetrahedral element. An off-line search algorithm identifies the index of each tetrahedral element that encompass one or more 3D regular grid point/points. The indices then are stored in an array and are exploited by multiple GPU kernels in the iterative sections (i.e. online sections) of the algorithm.

In contrast with the 3D/3D registration algorithm, 2D/3D registration method in [1] requires designing a GPU kernel to compute the indices in each iteration. Since the focus of this thesis is on 3D/3D, the search kernel remails as a possible future work. The kernel must search all or some tetrahedrons (depends on the search algorithm) for each 3D regular grid point to see whether the point is inside or outside the tetrahedrons.

It should be noted that in 3D/3D registration method, an intraoperative 3D image is transferred from CPU to GPU before starting the registration process. On the contrary, in 2D/3D registration method, a set of 2D slices of the intraoperative images are continuously transferred to GPU which may increase the overall runtime.

Chapter 6

Conclusions and Future Work

Medical image registration has become an integral part of many diagnostic and interventional procedures. In applications such as image-guided surgeries, biopsy, and radiotherapy, the task must be completed within a relatively short period of time. However advanced registration methods, particularly those that account for tissue deformation, are computationally expensive, restricting their use. The goal of this research was to study GPU-based parallel computing for fast deformable image registration. We have developed and tested a parallel implementation of the model-based image deformable image algorithm in [1]. The algorithm can take into account large deformations and is merely based on voxel intensities; therefore it does not require feature extraction or image segmentation to accomplish the registration. A GPU-based parallel implementation of the algorithm yielded 37-fold speedup over an optimized CPU implementation for registration of 3D $512 \times 512 \times 136$ MR images. This was accomplished through massive parallelization of the computationally intensive elements of the registration algorithm such

as interpolation, displacement, and force calculation on a GTX 480 GPU. The execution and memory resources were also analyzed and optimized to improve the performance. While MATLAB and CPU implementations take 23.25 and 1.38 minutes respectively, the GPU platform is capable of registering the same MR image sets (512×512×136) in 2.19 seconds. It should be mentioned that the parallel implementation can run on any other CUDA capable GPU with compute capability of 2.0 or more.

The results of our work show that near real-time registration can be achieved with recent advances in the GPU technology. GPUs have unique features such as texture memory which can significantly accelerate operations such as trilinear interpolation. They are much less expensive and much more compact than large multi-processor systems which require a large number of CPUs communicating through shared memory and/or over a netowrk. GPUs are also easier to program than FPGAs, making them widely popular for scientific general-purpose computing on GPU (GPGPU). One can easily modify the registration algorithm in case it is necessary for a specific medical application. Due to high demand of gaming industry, it is expected that the memory bandwidth and computational peak performance of GPUs will continue to increase in future, making real-time applications of advanced medical registration algorithms even more feasible.

Although the results presented in this study are very encouraging, there are still various possibilities for further studies:

- Comprehensive analysis of the CPU and GPU implementation accuracy and precision of in-vivo image sets instead of phantom image sets.

- Extension of the work to allow 3D to 2D registration of single and multi-modality images.
- Using multiple-GPU to further accelerate the computations in medical image registration algorithms.
- A comparative study of the speed and accuracy of the implemented registration algorithm with other well-known methods such as Demons [25], etc.
- Extension of the current implementation to use other similarity measures such as mutual information (MI) and correlation ratio. The algorithm in [1] is capable of working with these metrics.

Bibliography

[1] B. Marami, S. Sirouspour, and D. W. Capson, "Model-based deformable registration of preoperative 3d to intraoperative low-resolution 3d and 2d sequences of mr images," in *accepted to MICCAI*, 2011.

[2] Y. Guo, R. Sivaramakrishna, C. Lu, J. S. Suri, and S. Laxminarayan, "Breast image registration techniques: a survey," *Medical and biological engineering and computing*, vol. 44, no. 1, pp. 15–26, 2006.

[3] D. Rueckert and J. A. Schnabel, *Medical Image Registration*. Springer Berlin Heidelberg, 2011.

[4] D. Rueckert and P. Aljabar, "Nonrigid registration of medical images: Theory, methods, and applications," *IEEE Signal Processing Magazine*, vol. 27, no. 4, p. 113.

[5] J. B. A. Maintz and M. A. Viergever, "A survey of medical image registration," *Medical Image Analysis*, vol. 2, no. 1, p. 136, 1998.

[6] W. M. Wells, P. Viola, H.Atsumi, S. Nakajima, and R. Kikinis, "Multi-modal volume registration by maximization of mutual information," *Medical Image Analysis*, vol. 1, no. 1, p. 3551, 1996.

[7] T. Rohlfing and C. R. Maurer, "Nonrigid image registration in shared-memory multiprocessor environment with application to brains, breast and bees," *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, no. 1, p. 16.

[8] V. Saxena, J. Rohrer, and L. Gong, "A parallel gpu algorithm for mutual information based 3d nonrigid image registration," in *Lecture Notes in Computer Science, 16th International Euro-Par Conference on Parallel Processing*, pp. 223–234, 2010.

[9] M. Sen, Y. Hemaraj, S. Bhattacharyya, and R. Shekhar, "Reconfigurable image registration on fpga platforms," in *IEEE Conference on Biomedical Circuits and Systems*, pp. 154–157, 2006.

[10] A. Köhn, J. Drexl, F. Ritter, M. Knig, and H. O. Peitgan, "Gpu accelerated image registration in two and three dimensions," *Bildverarbeitung fr die Medizin*, vol. 3, no. 3, pp. 261–265, 2006.

[11] O. Fluck, C. Vetter, W. Wein, A. Kamen, B. Preim, and R. Westermann, "A survey of medical image registration on graphics hardware," *Computer Method and Programs in Biomedicine, article in press*, 2010.

[12] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*. Elsevier Inc, 2010.

[13] *NVIDIA Cuda C Programming Guide*. NVIDIA, 2011.

[14] R. Mirundci-Lunci, W. Blondel, C. Daul, Y. Hernandez-Mier, R. Posada, and D. Wolf, "A simplified method of endoscopic image distortion correction

based on grey level registration," in *International Conference on Image Processing*, pp. 3383 – 3386, 2004.

[15] D.Plattard, M. Soret, J. Troccaz, P. Vassal, J. Giraud, G. Champleboux, A. X, and M. Bolla, "Patient set-up using portal images: 2d/2d image registration using mutual information," *Computer Aided Surgery*, vol. 5, no. 4, p. 246262, 2000.

[16] J. X. Tao, S. Hawes-Ebersolea, M. Baldwina, S. Shaha, R. K. Ericksona, and J. S. Ebersole, "The accuracy and reliability of 3d ct/mri co-registration in planning epilepsy surgery," *Clinical Neurophysiology*, vol. 120, no. 4, pp. 748–753, 2009.

[17] C. Nikoub, F. Heitzb, J. Armspacha, I. Namera, and D. Grucker, "Registration of mr/mr and mr/spect brain images by fast stochastic optimization of robust voxel similarity measures," *NeuroImage*, vol. 8, no. 1, pp. 30–43, 1998.

[18] J. V. Hajnal, D. L. G. Hill, and D. J. Hawkes, *Medical Image Registration*. CRC Press LLC, 2001.

[19] M. J. van der Bom, J. P. W. Pluim, M. J. Gounis, E. B. van de Kraats, S. M. Sprinkhuizen, J. T. amd R. Homan, and L. W. Bartels, "Registration of 2d x-ray images to 3d mri by generating pseudo-ct data," *Pyisics in Medicine and Biology*, vol. 56, no. 4, p. 10311043, 2011.

[20] G. Li, Z. Ou, T. Su, and J. Han, "Graphic processing unit-accelerated mutual information-based 3d image rigid registration," *Transactions of Tianjin University*, vol. 15, no. 5, pp. 375–380, 2009.

[21] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "Parallel computation of mutual information on the gpu with application to real-time registration of 3d medical images," *Computer Methods and Programs in Biomedicine*, vol. 99, no. 2, pp. 133–146, 2010.

[22] A. Masood, A. M. . Siddiqui, and M. Saleem in *EEE Pacific Rim Symposium on Image and Video Technology*, pp. 651–663, 2007.

[23] F. L. Bookstein, "Principle warps: thin-plate splines and the decomposition of deformations," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 1989.

[24] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes, "Nonrigid registration using free-form deformations: application to breast mr images," *IEEE Transaction on Medical Imaging*, vol. 18, no. 8, pp. 712 – 721, 1999.

[25] J. P. Thirion, "Image matching as a diffusion process: an analogy with maxvell's demons," *Medical Image Analysis*, vol. 2, no. 3, pp. 243–260, 1998.

[26] S. F. F. Gibson and M. B, "A survey of deformable modeling in computer graphics," in *http://www.ncbi.nlm.nih.gov*, 1997.

[27] D. Mattes, D. R. Haynor, H. Vesselle, T. K. Lewellen, and W. Eubank, "Pet-ct image registration in the chest using free-form deformations," *IEEE Transactions on Medical Imaging*, vol. 22, no. 1.

[28] P. Jannin, J. M. Fitzpatrick, D. J. Hawkes, X. Pennec, R. Shahidl, and M. W. Vannier, "Validation of medical image processing in image-guided therapy," *IEEE Transactions on Medical Imaging*, vol. 21, no. 12.

[29] J. A. Schnabel, C. Tanner, A. D. Castellano-Smith, A. Degenhard, M. O. Leach, D. R. Hose, D. L. G. Hill, and D. J. Hawkes, "Validation of nonrigid image registration using finite-element methods: Application to breast mr images," *IEEE Transactions on Medical Imaging*, vol. 22, no. 2.

[30] S. Leea, J. M. Reinhardt, P. C. Cattin, and M. D. Abrmoff, "Objective and expert-independent validation of retinal image registration algorithms by a projective imaging distortion model," *Medical Image Analysis*, vol. 14, no. 4, pp. 539–549, 2010.

[31] J. Bijhold, "Three-dimensional verification of patient placement during radiotherapy using portal images," *Medical Physics*, vol. 20, no. 2, pp. 347–357, 1992.

[32] G. K. Rohde, A. Aldroubi, and B. M. Dawant, "The adaptive bases algorithm for intensity-based nonrigid image registration," *IEEE Transactions on Medical Imaging*, vol. 12, no. 11, pp. 1470 – 1479, 2003.

[33] J. M. Fitzpatrick, J. B. West, and C. R. Maurer, "Predicting error in rigid-body point-based registration," *IEEE Transactions on Medical Imaging*, vol. 17, no. 5, pp. 694 – 702, 1998.

[34] N. Lepore, A. A. Joshi, R. M. Leahy, C. Brun, Y. Chou, X. Pennec, A. D. Lee, M. Barysheva, G. I. D. Zubicaray, M. J. Wright, K. L. McMahon, A. W. Toga,

and P. M. Thompson, "A new combined surface and volume registration," in *Proceedings of SPIE - The International Society for Optical Engineering*, 2010.

[35] J. K. S. Xu, B. Turkbey, N. Glossop, A. K. Singh, P. Choyke, P. Pinto, and B. J. Wood, "Real-time mri-trus fusion for guidance of targeted prostate biopsies," *Computer Aided Surgery*, vol. 13, no. 5, p. 255264, 2003.

[36] J. Woo, B. W. Hong, C. H. Hu, K. K. Shung, C. C. J. Kuo, and P. J. Slomka, "Non-rigid ultrasound image registration based on intensity and local phase information," *Journal of Signal Processing Systems*, vol. 54, no. 1, pp. 33–43, 2009.

[37] K. Ensa, S. Heldmanna, J. Modersitzkic, and B. Fischera, "Improving an affine and nonlinear image registration and/or segmentation task by incorporating characteristics of the displacement field," in *Medical Imaging: Image Processing*, 2009.

[38] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J*, vol. 27, pp. 379–423, 1948.

[39] O. Dandekar and R. Shekhar, "Fpga-accelerated deformable image registration for improved target-delineation during ct-guided interventions," *IEEE Transactions on Biomedical Circuits AND Systems*, vol. 1, no. 2, pp. 116–127, 2007.

[40] P. Hastreiter and T. Ertl, "Integrated registration and visualization of medical image data," in *Computer Graphics International*, pp. 78–85, 1998.

[41] L. Xu and J. W. L. Wan, "Real-time intensity-based rigid 2d-3d medical image registration using rapidmind multi-core development platform," in *Engineering in Medicine and Biology Society, EMBS*, pp. 5382 – 5385, 2008.

[42] F. Ino, J. Gomita, Y. Kawasaki, and K. Hagihara, "A gpgpu approach for accelerating 2-d/3-d rigid registration of medical images," *Parallel and Distributed Processing and Applications*, vol. 4330, pp. 939–950, 2006.

[43] W. Plishker, O. Dandekar, S. S. Bhattacharyya, and R. Shekhar, "Towards systematic exploration of tradeoffs for medical image registration on heterogeneous platforms," in *IEEE Biomedical Circuits and Systems Conference*, pp. 53–56, 2008.

[44] T. Y. Huang, Y. W. Tang, and S. Y. Ju, "Accelerating image registration of mri by gpu-based parallel computation," *Magnetic Resonance Imaging*, vol. 29, no. 5, pp. 712–716, 2011.

[45] R. E. Ansorge, S. J. Sawiak, and G. B. Williams, "Exceptionally fast non-linear 3d image registration using gpus," in *IEEE Nuclear Science Symposium Conference*, pp. 4088 – 4094, 2009.

[46] G. C. Sharpand, N. Kandasamy, H. Singh, and M. Folkert, "Gpu-based streaming architectures for fast cone-beam ct image reconstruction and demons deformable registration," *Physics in Medicine and Biology*, vol. 52, no. 19, pp. 5771–5783, 2007.

[47] P. M. Ozcelik, J. D. Owens, J. Xia, and S. S. Samant, "Fast deformable registration on the gpu: A cuda implementation of demons," in *International Conference on Computational Science and Its Application*, pp. 223–233, 2008.

[48] B. Li, A. A. Young, and B. R. Cowan, "Gpu accelerated non-rigid registration for the evaluation of cardiac function," in *Medical Image Computing AND Computer-Assisted Intervention (MICCAI)*, pp. 880–887, 2008.

[49] G. R. Joldes, A. Wittek, M. Couton, S. K. Warfield, and K. Miller, "Real-time prediction of brain shift using nonlinear finite element algorithms," in *Medical Image Computing AND Computer-Assisted Intervention (MICCAI)*, pp. 300–307, 2009.

[50] K. J. Bathe, *Finite Element Procedures in Engineering Analysis*. Prentice-Hall, 1982.

[51] O. C. Zienkewickz and R. L. Taylor, *The Finite Element method*. McGraw Hill, 1987.

[52] *Cuda C Best Practice Guide*. NVIDIA, 2011.

[53] J. Sanders and E. Kandrot, *CUDA by Example- An Introduction to General Purpose GPU Programming*. Addison Wesley, 2011.

[54] H. R. Kang, *Computational color technology*. SPIE Press Book, 2006.

[55] H. Mousazadeh, B. Marami, S. Sirouspour, and A. Patriciu, "Gpu implementation of a deformable 3d image registration algorithm," in *Accepted for presentation at 33rd International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC 11)*, pp. 4897–4900, 2011.

[56] *CUDA Cublas library*. NVIDIA, 2010.

[57] N. Bell and M. Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report, Dec.

[58] A. Nentchev, *Numerical Analysis and Simulation in Microelectronics by Vector Finite Elements*. PhD Thesis, Technische Universitt Wien, Institute for Microelectronics, 2008.

[59] N. Samavati, *Deformable Multi-Modality Image Registration Based on Finite Element Modeling and Moving Least Squares*. M.A.Sc Thesis, Electrical and Computer Engineering, McMaster University, 2009.