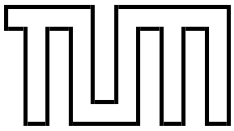
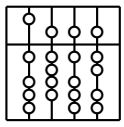


Technische Universität München
Fakultät für Informatik

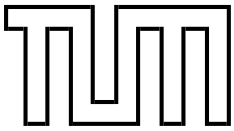


Diplomarbeit

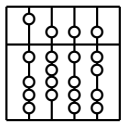
Techniques for Accelerating Intensity-based Rigid Image Registration

In collaboration with Siemens Corporate Research Inc., Princeton, USA

Razvan Chisu



Technische Universität München
Fakultät für Informatik



Diplomarbeit

Techniques for Accelerating Intensity-based Rigid Image Registration

In collaboration with Siemens Corporate Research Inc., Princeton, USA

Razvan Chisu

Aufgabensteller:	Prof. Nassir Navab, Ph.D. (TUM)
Betreuer:	Dipl.Inf. Wolfgang Wein (TUM) Ali Khamene, Ph.D. (SCR)
Abgabedatum:	15. Januar 2005

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Januar 2005 Razvan Chisu

Abstract

One important application of computers in medicine is the merging of various sets of patient data coming from different sources. In the case of image registration, the aim is to determine the translation and rotation by which one data set can be spatially aligned with a second one. Volumetric data (e.g. CT data) can be registered to another volume or to a two-dimensional image (e.g. an X-Ray image).
An essential part of this process is the ability to measure the similarity between two images. This paper focuses on various techniques for speeding up assessing the similarity of two-dimensional images. In most cases, the registration process requires 2D images to be generated from volumetric data, which can be accomplished much faster by using the capabilities of modern graphics hardware. One acceleration approach therefore consists in also moving the similarity measure computation to the graphics board. The second approach is taking advantage of parallel processing capability (SIMD - "single intruction, multiple data") of modern processors.
The various similarity measures and implementations are benchmarked for a 3D-3D registration process on different data sets. The results show that we can register two volumes very accurately in 20% to 25% of the time needed for a non-accelerated approach.

Zusammenfassung

Eine wichtige medizinische Anwendung von Computern ist das Verschmelzen von Patientendaten aus verschiedenen Quellen. Im Fall der Bildregistrierung besteht das Ziel darin, die Translation und Rotation zu bestimmen, durch die ein Datensatz in die selbe räumliche Ausrichtung gebracht wird, wie ein Zweiter. Volumetrische Daten (z.B. CT Daten) können mit einem anderen Volumen oder mit einem zwei-dimensionalen Bild (z.B. Röntgenbild) registriert werden.
Ein essenzieller Bestandteil dieser Anwendung ist die Möglichkeit, die Ähnlichkeit zweier Bilder zu berechnen. Ziel dieser Diplomarbeit ist die Untersuchung verschiedener Techniken um die Auswertung der Ähnlichkeit zwei-dimensionaler Bilder zu beschleunigen. In den meisten Fällen müssen, zum Zwecke der Registrierung, zwei-dimensionale Bilder aus Volumendaten generiert werden, was durch Nutzung moderner Grafikhardware viel schneller erfolgen kann. Ein Beschleunigungsansatz besteht demnach darin, die Berechnung der Ähnlichkeitsmaße ebenfalls auf die Grafikkarte auszulagernn. Der zweite Ansatz verwendet die Fähigkeiten zur parallelen Berechnung (SIMD - ŝingle intruction, multiple data") der modernen Prozessoren. Die verschiedenen Ähnlichkeitsmaße und Implementierungen werden mit einer 3D-3D Registrierungsanwendung auf verschiedenen Datensätzen untersucht. Die Ergebnisse zeigen, dass wir zwei Volumen in 20% bis 25% der Zeit für einen nicht-beschleunigten Ansatz mit hoher Genauigkeit registrieren können.

Preface

About this work This document is my diploma thesis for the computer science degree at the Technische Universität München (TUM). A part of it originates from my preliminary work at the TUM, where I acquired the basic knowledge and completed the implementation of the GPU accelerated algorithms, while the rest arises from an internship with Siemens Corporate Reasearch (SCR) in Princeton/NJ, USA, lasting from September to December 2004. I was working on the project *Reg3D*, a prototypical application for volumentric registration.

Acknowledgements First of all I would like to thank Prof. Nassir Navab for recommending me to SCR and conducting this thesis with me. I am also very grateful to Wolfgang Wein, my supervisor at the TUM, who has given me very valuable advice concerning both the general issues of the image registration process as well as detailed technical problems.
Last but not least, I am especially indebted to Ali Khamene and Frank Sauer for finding the time to support me in any necessary way, in spite of so many other SCR interns.

Contents

1 Introduction

1.1 Motivation

Medical imaging has become a vital component of a large number of medical applications, ranging from diagnostics and planning to consummation and evaluation of surgical and radiotherapeutical procedures. A vast number of imaging modalities is being used in today's clinics, including two-dimensional ones like X-ray, Angiography and Ultrasound (US) and three-dimensional ones like Computed Tomography (CT), Magnetic Resonance Tomography (MRT) and Positron Emission Tomography (PET). Especially volumetric data acquired by CT or MRT scans proves highly valuable in daily medical practice since it provides physicians with a high amount of detailed information about the patient. Medical images can be preprocessed and segmented into objects of interest, like single organs or bone structures, allowing for different visualization techniques and thus even more efficient presentation of data to the medic. The visualization and processing of such medical images represents one of the most important applications of computers in the medical domain, which profits very much from the steadily increasing performance of today's hardware [45].

It is common practice to acquire several images of a patient at different points of time, or by means of different types of sensors. Such images often provide complementary information, so it is highly desired to merge data coming from separate data sets. Besides from providing the physician with more information than single images could, this process also helps in exploiting the available data to a maximum degree, thus reducing the overall number of images that need to be acquired, which helps lowering clinics' operating costs and inconveniences (eg. exposure to radiation) for the patient. Since medical imaging is about establishing shape, size, structure and spatial relationships of anatomical structures within the patient's body, bringing different images into spatial alignment usually represents the first step in this process of data fusion and is referred to as *registration*.

Digital medical images, especially three-dimensional volumetric data, can easily reach hundreds of megabytes in size (say 500 slices of 512 x 512 images with 16-bit pixels equals 260 MB). Therefore, dealing with them is computationally quite expensive. Although today's commodity micro-processors can perform rigid image registration in less than a minute, some programming techniques might accelerate this process even further, maybe even reaching the point of real-time registration. The aim of this paper is to analyze and benchmark such techniques for accelerated registration.

Figure 1.1: Visualization of volumetric data, from [42]

1.2 Medical Image Registration

Maintz and Viergever have proposed a comprehensible classification of the whole registration problem based on nine main criteria [21], which include image dimensionality, the nature of the registration basis, the nature and domain of the transformation, user interaction, optimization procedures, the involved imaging modalities, the subject and the object.

All applications of image registration require the establishment of a spatial correspondence between the respective data sets. In other words, registration involves finding a spatial transformation that relates structures in one image to the corresponding structures in another image. The nature of these transformations can be classified by the dimensionality of the images to be registered and the number of parameters, or degrees of freedom, to be computed. Depending on the imaging sensor, the acquired data can be two, three or four-dimensional (three dimensions in space plus time as the fourth), so we can distinguish between:

- **2D-to-2D registration**
  In the most simple of cases, two 2D images can be registered by determining the translation along the two axes and the rotation angle. However, clinically relevant applications of this case are rare, since changes in the geometry of image acquisition over time or from one imaging sensor to another introduce additional degrees of freedom.

- **2D-to-3D registration**
  This case arises when a correspondence between a projection image like X-ray and a volumetric data set like CT must be established. A typical application of this would be relating an intra-operative 2D image to a preoperative 3D image for the purpose

of image-aided surgery or radiotherapy. The matching of such data sets usually yields ten degrees of freedom (whereas the case of a general affine transformation would yield fifteen degrees of freedom). Four of them determine the projective transformation for the 2D image and can therefore be determined in advance by correct calibration, leaving six parameters for the volume's translation and rotation along the three axes. Since most of the 2D-to-3D applications concern intra-operative scenarios, they are heavily time-constrained and have a strong focus on speed issues concerning the computation of the spatial pose.

- **3D-to-3D registration**
  The registration of two volumes, is the most well developed process and a widely used method. The positions of two rigid bodies can always be related to one another by means of three translations and three rotations, but the particular conditions of imaging might impose further degrees of freedom to account for different scaling of voxels, a tilt in the gantry or a general affine transformation which is determined by twelve paramters.

- **Registration of time series**
  Time series of images are acquired for various reasons and at various intervals. Applications include monitoring of tumor growth, post-operative monitoring of the healing process or observing the passing of injected bolus through a vessel tree. Subsequent registration of such an image series can be used to study dynamic processes such as blood flow or metabolic processes.

Depending on the organs and anatomical structures which are of interest, registration can be divided into *rigid registration* (ie. the registration of rigid structures like bone) and *deformable registration* (ie. the registration of deformable structures like soft tissue). Deformable registration adds a theoretically unlimited number of degrees of freedom to the ones named above. In order to make registration possible, several restrictions can be applied to the deformation based upon the physics of tissue transformation or general mathematical models[8].

This work will only investigate the area of rigid registration.

1.3 Medical Applications

1.3.1 Image-Guided Surgery

Especially the domain of neurosurgery has made extensive use of intraoperative image guidance. A preoperatively acquired and pre-segmented data set providing detailed information about the patient's anatomy can be combined with intraoperative images, possibly coming from different modalities, to help in better locating the lesions in relation to surrounding tissue and in identifying the position of used instruments or displaying any other information previously acquired.
For registration, fiducial markers or a laser-scan of the skin are commonly used. Another very popular method, and also the most accurate one [1], is the stereotactic frame. Here, a lightweight frame is tightly attached to the patient's head and, being visible in both the preoperative and intraoperative images, allows for a very accurate tracking of the patient's

position. Additionally, surgical instruments can be attached to the frame. However, as the stereotactic frame must remain attached during the whole procedure, it cannot be used for multiple treatments and can also be hampering in some interventions.

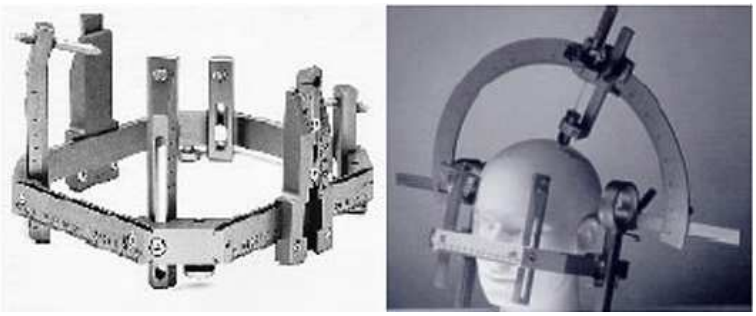


Figure 1.2: Stereotactic frame

1.3.2 Radiation Therapy

Radiation therapy, or radiotherapy, is commonly used for cancer treatment. In order to stop or slow down the reproduction of cancerous cells, the tumor is irradiated using a linear accelerator (LINAC), thus damaging the target cells' DNA, hampering their ability to reproduce. The DNA is either damaged directly by the radiation or indirectly by free radicals, a result of the ionization of oxygen atoms. While all cells have mechanisms for repairing damaged DNA strands, this ability is diminished in cancer cells, making them more vulnerable to radiation, as all damage to the DNA is inherited through the process of cell division [47].
Most modern linear accelerators, like the Siemens Primus, have a treatment couch that can be translated in all spatial directions and rotated along one axis, thus allowing the patient to be positioned in such a way that the target tissue is situated in the radiation beam's point of focus, or isocenter, which remains fixed at all times. While the patient is being immobilized, the gantry is usually rotated around one axis, thus making sure that only the target area is constantly irradiated, while the neighboring tissue is only affected by a minimal dose [45].

Before the beginning of the actual therapy, a CT scan of the patient is acquired which serves as the basis for radiotherapy planning. In order to maximize the amount of radiation delivered to the tumor and minimize the irradiation of healthy tissue, the exact location and size of the tumor and the necessary amount of radiation have to be determined. The radiation therapy itself is usually made up of multiple sessions, so the patient's position must be realigned to the LINAC's isocenter. For this purpose, two-dimensional portal images or three-dimensional CT scans can be acquired at the new position and then registered to the data set from the planning phase. Often, fiducial markers are implanted into the patient's body to facilitate registration. Depending on the actual medical situation, single-session therapies are also being performed. This will be especially the case where a stereotactic frame is being used for registration.

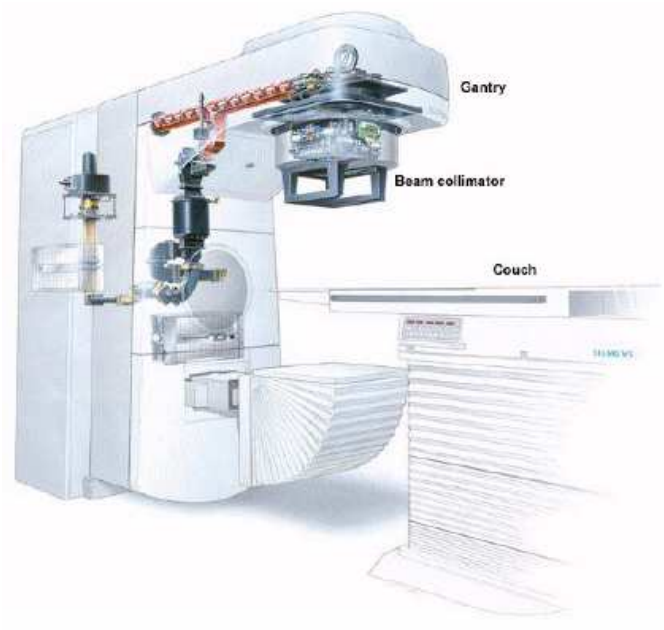


Figure 1.3: Schematic view of a Siemens Primus linear accelerator

1.3.3 Longitudinal Studies

Another application where rigid image registration is necessary are longitudinal studies, where the evolution of a certain anatomical feature in a patient is studied over a period of time. Though very often used for monitoring the development and response of tumors to radiotherapy, longitudinal studies are carried out in other medical domains as well, eg. to monitor the development of tooth lesions [6] or analyze the healing process after a surgical intervention. The data sets to be registered are obtained using the same imaging modality and after aligning rigid anatomical features, like bones, changes in interesting structures can be analyzed by a physician.

1.4 The Intensity-based Registration Algorithm

In order to spatially align two images, registration algorithms basically try out different poses, starting from an initial estimate, and compute some kind of image similarity measure for each pose, repeating this process until an optimal alignment is found.
In the case where the images to be registered are of different dimensionality, ie. 2D-3D registration, there also appears the need for generating a two-dimensional image from the volume in order to allow the assessment of image similarity. The points in two-dimensional X-ray images correspond to the line integral of the gamma radiation attenuation along the corresponding line from radiation source to detector. As such, X-ray images represent a projection of the three-dimensional physical space, which can be simulated from a corresponding CT volume by applying a perspective projective transformation to it, using the intrinsic parameters of the X-ray imaging device. This simulation of X-ray images from CT volumes yields a so-called *Digitally Reconstructed Radiograph (DRR)*, which can then be compared to the

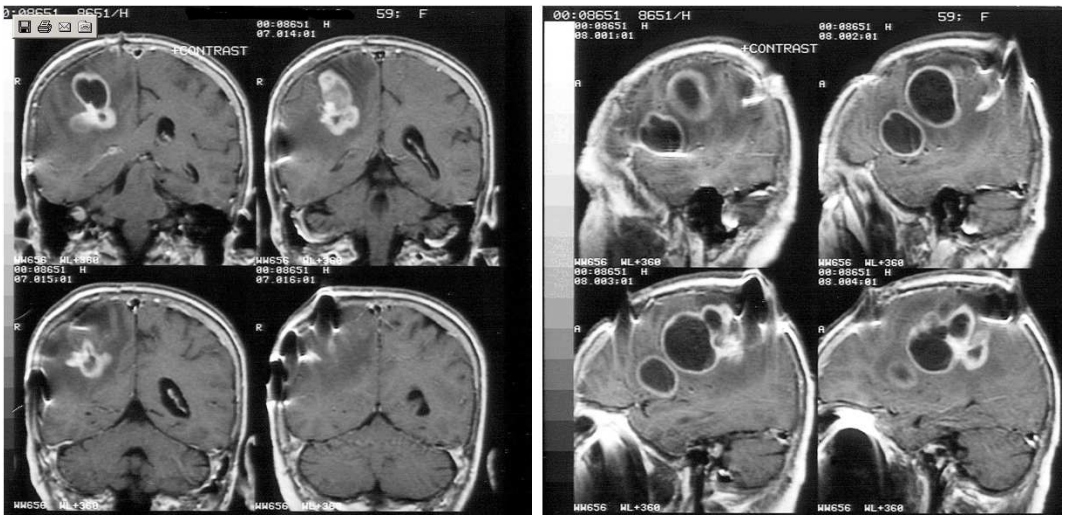


Figure 1.4: CT scan of a patient before therapy and after two months after the operation and chemo- and radiation therapy. From [30]

original X-ray image. Other imaging modalities, eg. ultrasound, do not represent a projective transformation of the physical reality, so the process of generating a corresponding 2D image from the 3D data set must be adapted accordingly. The similarity of the two 2D images can then be used to iteratively determine better estimates for the parameters describing the 3D-to-2D transformation applied to the volume data, usually the volume's rotation and translation along the three axes in the case of rigid transformations.

3D-3D registration can be performed in two ways. The volumes can be aligned directly by computing a volumetric similarity measure that examines the voxel intensities. Although this approach may be very straightforward, the number of computations for each pose is very high, resulting in a very slow registration process.
A new possibility, and also the one we used in our application (see 5.1), is to use orthographic projections along the three coordinate axes of the volume data and register the volumes by repeatedly align these two-dimensional images. Although this introduces the cost of generating projections from the volumes, the significantly reduced number of intensity values analyzed for assessing the similarity measure results in a much better peformance. One may choose to perform a predefined number of 2D-2D registrations or repeat the process until the images' similarity has reached a certain threshold.

In effect, the registration algorithm works as follows:

1. The algorithm starts with the two 2D/3D or 3D/3D images, an initial estimate for the spatial alignment, which is set by the user, and, if required, the parameters needed for generating two-dimensional images from the volume data, eg. the parameters for the perspective projection in the case of X-ray-to-CT registration or those for the orthographic projection in the case of 3D-3D registration.

2. The one or two required projections are computed from the columetric data set(s). This process is known as *volume rendering* and can be significantly accelerated by using modern graphics processors, which implement volume rendering as standard functionality. However, this type of acceleration is not part of this thesis.

3. The alignment of the two images (of same dimensionality) can now be estimated by computing a similarity measure. A large variety of such measures exist, which I will summarize in the next section.

4. Finding the best pose represents an unconstrained non-linear optimization problem. The similarity measure thus plays the role of a cost function for a non-linear optimizer algorithm that tries to find the pose that will minimize (or maximize) the image similarity. Several schemes can be applied for searching the parameter space in order to keep the number of required cost function evaluations as low as possible. By repeating steps 2-4, the optimizer is attempting to find the global optimum in the parameter space, thus determining the best alignment of the two images. Various abortion parameters (eg. number of iterations or the size of the steps taken in the parameter space) can be used to define when the algorithm terminates.

1.5 Acceleration Techniques

Driven by the need for more and more computational power on consumer-level computers, hardware manufacturers have steadily developed microprocessor architecture to work in a more efficient way. But apart from speeding up general code execution through higher clock rates and pipelined architectures, modern CPUs also implement special instructions which operate on multiple data elements simultaneously. This concept is known as SIMD (single instruction, multiple data), as opposed to SISD (single instruction, single data), where each operation has only one or a pair of parameters.
Aiming especially at multimedia and communications applications, Intel introduced the MMX instruction set back in 1995 on the Pentium MMX processors[11]. With several new data types, instructions and 64-bit MMX registers, this was the first step in bringing parallel computation capabilities to desktop computers. Later on, SIMD functionality was extended with the introduction of SSE (Streaming SIMD Extensions)[16] with the Pentium III processor. SSE is made up of instructions working with 128 bits of data regardless of the actual data type and the number of data elements therein, ie. SSE instructions can operate on anything from two 64 bit double-precision floating-point values to sixteen byte values. SSE was later extended by SSE2 and SSE3, adding several new instructions but no conceptual difference.
These SIMD instruction sets are explicitly meant for certain applications and therefore offer only a reduced functionality when compared to the CPUs general-purpose instructions. Furthermore, the compilers' ability to automatically optimize code for using MMX and SSE instructions is rather reduced, so the algorithms must be rewritten to use certain dedicated functions and store variables in a special manner in the system's RAM. Also, the programmer must figure out how to best take advantage of this parallel computation capability, which is not always straightforward.

As of late, a completely new possibility for carrying out computations on a desktop computer has arisen in the form of programmable graphics hardware. Modern GPUs[1] offer the possibility of replacing the otherwise fixed functionality required for rendering and displaying

[1] Graphics Processing Unit

geometric shapes with custom program code, allowing the GPU to be used as a general-purpose processor, within the hardware architecture's respective limitations.
The computation performance offered by today's video boards by far surpasses that of currently available CPUs - while a Pentium 4 3GHz CPU can theoretically reach 6 GFLOPS[2], synthetic benchmarks have shown the NVIDIA Geforce 6800 Ultra GPU to reach 40 GFLOPS [9]. This fact, together with the inherently parallel architecture of graphics processors, has made this approach of GPU programming highly attractive for accelerating algorithms in different domains. However, since graphics processors are designed for carrying out very special kinds of calculations, as opposed to the multi-purpose intended CPUs, numerous restrictions and other issues appear when trying to use their capabilities for a purpose that developers don't primarily have in mind.
In the domain of medical image registration, GPUs have been mainly used to speed up the generation of DRRs using hardware-accelerated volume rendering techniques. Our aim is now to move even more of the computations to the GPU in order to eliminate the slow data transfers between the graphics memory and the system memory.

[2]Billions of floating-point operations per second

2 Intensity-based Similarity Measures

Determining how similar two images are is a key problem in the registration process, since it indicates to what degree the images are aligned. Two main approaches for determining a measure of similarity exist, namely intensity-based and feature-based measures [32][45].

Feature-based measures take into account different shapes or structures in the two images, like points, curves or surfaces. These shapes can be markers placed inside or on the patient's body as well as anatomical elements like bones. Obviously, those objects must be identified before the actual similarity can be computed. This feature extraction, or *segmentation*, usually requires some user interaction and is therefore not a good choice for fully automatic registration. Still, it should be noted that these algorithms are usually very fast, once the segmentation has been completed, since they operate on a reduced set of information.

Intensity-based measures, on the other hand, only operate on the pixel or voxel values. Without the need for user interaction in identifying any landmarks or supervising a segmentation algorithm, this class of similarity measures has become quite popular. Although this approach is slower than the feature-based one because it takes the complete image data into account (which can reach hundreds of megabytes for high-resolution volumes), it can be sped up by defining a region of interest (ROI), disregarding the pixels (or voxels) outside of this area when computing the similarity measure and when generating the DRRs. Apart from speeding up the registration, the use of a ROI can also lead to a more accurate registration of the anatomical features the physician is interested in, since other, possibly deformable and thus non-matching, elements can be ignored.
Intensity-based similarity measures can further be classified as measures using only image intensities, measures using spatial information (ie. intensities in a voxel's neighborhood) and histogram-based measures.

2.1 Measures Using only Image Intensities

The most simple type of similarity measures only regards pairs of intensity values at the same pixel positions in the two images. The mathematical scheme by which this per-pixel similarity is being computed can be chosen with respect to the application's characteristics.

2.1.1 Minimizing Intensity Difference

One of the simplest measures is the sum of squared intensity differences (SSD)[8] between two images:

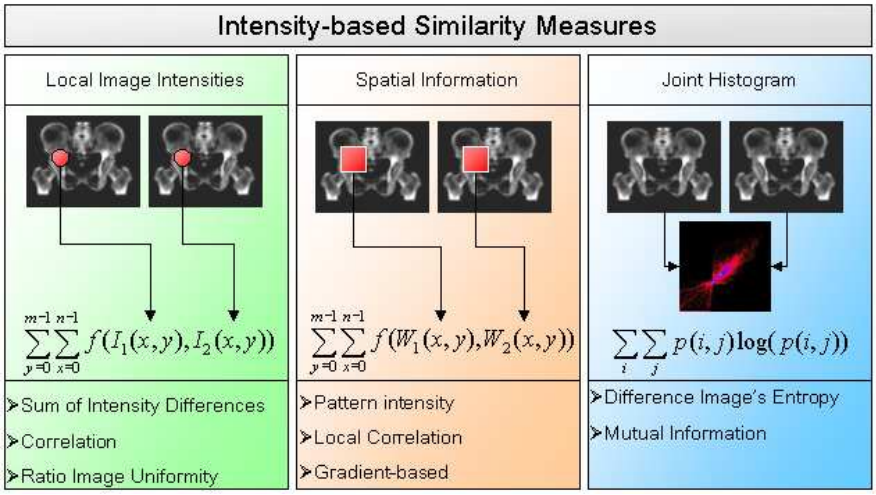


Figure 2.1: Classification of intensity-based image similarity measures

$$SSD = \frac{1}{N}\sum_{x,y\in T}(I_1(x,y) - I_2(x,y))^2 \tag{2.1}$$

where T is the images' overlapping domain consisting of N voxels and $I_1(x,y)$ and $I_2(x,y)$ are the voxel intensities at the (x,y) coordinate in each image.
It can be shown that this is the optimum measure when the two image differ only in gaussian noise[40]. This assumption never applies for inter-modality registration and almost never applies for intra-modality registration either, since occurring noise is rarely gaussian. Furthermore, changes in the imaged objects from one scan to the other further reduce the applicability of this assumption. SSD is also very sensitive to a small number of pixel pairs with high intensity differences, which can be caused by instruments or contrast visible in only one of the images. The sum of absolute intensity differences (SAD) is less sensitive to such outliers:

$$SAD = \frac{1}{N}\sum_{x,y\in T} \mid I_1(x,y) - I_2(x,y) \mid \tag{2.2}$$

2.1.2 Correlation Coefficient

If the assumption that registered images differ only by gaussian noise is replaced with a less restrictive one, namely that there is a linear relationship between the two images, the optimum measure is the correlation coefficient CC[8], which is sometimes referred to as normalized cross correlation (NCC):

$$NCC = \frac{\sum_{x,y\in T}(I_1(x,y) - \overline{I_1}) \cdot (I_2(x,y) - \overline{I_2})}{\sqrt{\sum_{x,y\in T}(I_1(x,y) - \overline{I_1})^2} \cdot \sqrt{\sum_{x,y\in T}(I_2(x,y) - \overline{I_2})^2}} \tag{2.3}$$

where $\overline{I_1}$ and $\overline{I_2}$ are the mean intensities of the two images.
Using this scheme, the algorithm will have to run through the image data twice - once for computing the mean intensity and then for computing the correlation coefficient itself. The

formula can be rewritten in such way as to require only one reading pass [45]. This expression is also faster to compute, as several differences are moved out from within the sums into the final formula. However, this approach is subject to more numerical problems [34].

$$NCC = \frac{\sum_{x,y \in T} I_1(x,y) \cdot I_2(x,y) - n\overline{I_1 I_2}}{\sqrt{\sum_{x,y \in T} I_1(x,y)^2 - n\overline{I_1}^2} \cdot \sqrt{\sum_{x,y \in T} I_2(x,y)^2 - n\overline{I_2}^2}} \tag{2.4}$$

2.1.3 Ratio Image Uniformity

The RIU measure[48] (also known as variance of intensity ratios, or VIR) first generates a ratio image from the two medical images by dividing pixel intensities at corresponding coordinates. The uniformity of this ratio image is then computed as being its standard deviation divided by its mean intensity, thus providing a normalized cost function which is independent of global intensity scaling of the original images.
To make the algorithm unbiased with regard to which image intensities are chosen as divident and which as divisor, the ratio is inverted, and the measure is computed again. The final result is then the average of the two steps.

$$RIU = \frac{1}{2} \cdot \left(\frac{\sqrt{\frac{1}{N}\sum (R_1(x,y) - \overline{R_1})^2}}{\overline{R_1}} + \frac{\sqrt{\frac{1}{N}\sum (R_2(x,y) - \overline{R_2})^2}}{\overline{R_2}}\right) \tag{2.5}$$

with

$$R_1(x,y) = \frac{I_1(x,y)}{I_2(x,y)}, R_2(x,y) = \frac{I_2(x,y)}{I_1(x,y)}$$

As for the Correlation Coefficient, this formula can also be expanded in order to allow the measure to be computed without having to run through the images twice.

$$RIU = \frac{1}{2} \cdot \left(\frac{\sqrt{\frac{1}{N}\sum (R_1(x,y))^2 - N\overline{R_1}^2}}{\overline{R_1}} + \frac{\sqrt{\frac{1}{N}\sum (R_2(x,y))^2 - N\overline{R_2}^2}}{\overline{R_2}}\right) \tag{2.6}$$

Obviously, one must decide how to handle values of zero. The most simple way of doing so, which also is the approach I took for both the GPU and the CPU implementations, is to map the input intensity range to an interval excluding zero values. I have chosen to transform the intensities from the original $[0\% - 100\%]$ range to $[5\% - 95\%]$.

2.2 Measures Using Spatial Information

Instead of regarding only the intensity values at a certain coordinate position, this type of measures also considers each pixel's neighborhood when computing the per-pixel similarity. The algorithm can thus operate on either the original image intensities and sum up the differences in a certain radius around the current position or first compute the gradient images and then compare them either with a pixel-based or a neighborhood-based scheme.
Gradient-based measures have the advantage of being quite insensitive to low spatial frequency differences, which can be caused by soft-tissue deformation. Taking into account only edge information intuitively also seems reasonable when trying to register two images.

2.2.1 Pattern Intensity

This measure [44][32] examines the contents of a difference image, starting from the assumption that the number of patterns, or structures, in this difference image will be the lowest when the optimal alignment has been found. A voxel is considered to belong to a structure if it has a significantly different intensity value from its neighboring voxels. With r being the radius of the neighborhood and σ a weighting constant, the measure is computed as follows:

$$PI_{r,\sigma} = \sum_{x,y \in T} \sum_{d^2 < r^2} \frac{\sigma^2}{\sigma^2 + (I_{diff}(x,y) - I_{diff}(v,w))^2} \tag{2.7}$$

$$d^2 = (x-v)^2 + (y-w)^2$$

$$I_{diff} = I1 - s \cdot I_2$$

For I_{diff}, a scaling constant s can be used to reduce the contrast of the resulting difference image. A good choice for r and σ seem to be 3 and 10, although one might want to increase the neighborhood radius to 5 to increase the measure's robustness.
Since PI evaluates differences in the difference image, a constant intensity shift in the original data sets does not influence the result.
As it is right now, PI has the drawback of computing $fraco\sigma^2\sigma^2 + (I(x,y) - I(u,v))2$ twice for each pair of pixels $P_1(x,y)$ and $P_2(u,v)$ - once when P_1 is in the center and P_2 belongs to its neighborhood, and once again when P_2 becomes the current location and has P_1 as neighbor. Thus we can disregard half of the computations, as described in [45], by running only through one part of the neighborhood.

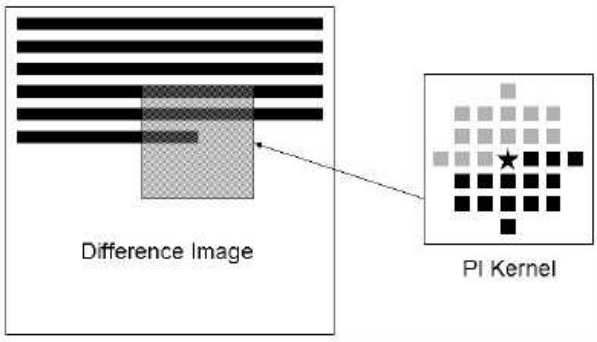


Figure 2.2: Kernel for computation of neighborhood differences, from [45]

2.2.2 Gradient Correlation

This method computes the correlation coefficients 2.3 for the horizontal and vertical gradient image pairs. The final measure of similarity is then the average of these two values.

$$GC = \frac{1}{2} \cdot \frac{\sum_{x,y \in T}(I_{1H}(x,y) - \overline{I_{1H}}) \cdot (I_{2H}(x,y) - \overline{I_{2H}})}{\sqrt{\sum_{x,y \in T}(I_{1H}(x,y) - \overline{I_{1H}})^2 \cdot \sum_{x,y \in T}(I_{2H}(x,y) - \overline{I_{2H}})^2}} + \frac{\sum_{x,y \in T}(I_{1V}(x,y) - \overline{I_{1V}}) \cdot (I_{2V}(x,y) - \overline{I_{2V}})}{\sqrt{\sum_{x,y \in T}(I_{1V}(x,y) - \overline{I_{1V}})^2 \cdot \sum_{x,y \in T}(I_{2V}(x,y) - \overline{I_{2V}})^2}} \tag{2.8}$$

2.2.3 Gradient Difference

Proposed in [32], Gradient Difference (GD) examines two difference images obtained from substracting the vertical and horizontal gradients of the two original images. By applying a $1/(1+x^2)$ scheme to the result, this measure should be quite robust to outliers. The fact that two images are being analyzed allows this measure to compare both direction and magnitude of the gradients.

$$GD = \sum_{x,y \in T} \frac{A_v}{A_v + (I_{diffV}(x,y))^2} + \sum_{x,y \in T} \frac{A_h}{A_h + (I_{diffH}(x,y))^2} \tag{2.9}$$

with

$$I_{diffV}(x,y) = \frac{dI_1}{dx} - s \cdot \frac{dI_2}{dx}, I_{diffH}(x,y) = \frac{dI_1}{dy} - s \cdot \frac{dI_2}{dy}$$

and A_h and A_v normalization constants, which seem to work well if set to the variance of the respective image.

2.2.4 Gradient Proximity

For the purpose of aligning two gradient images, I propose this simple and fast measure. We can interpret the gradient vectors computed for each image pixel as position vectors, indicating the location of a point in a 2D plane. Thus we can use the proximity of the two points as a measure of how well the two gradients match, both in angle and in length. I define the proximity $P \in [0,1]$ of two points p_1 and p_2 as follows:

$$P(p_1,p_2) = 1 - \frac{distance(p_1,p_2)}{max\_distance}, \tag{2.10}$$

where $max\_distance$ represents the maximum possible distance between any two points, which depends on the vector components' data range, ie. [0..1], [0..255] and so on. As a distance measure, both the euclidian and the city-block distance may be used. Maximizing the sum of these values over all image pixels should then deliver the optimal gradient alignment. In effect, this is a rewriting of the SSD and SAD measures (using the euclidian and city-block distance, respectively) and applying them to the gradients rather than the raw image data.
However, the proximity alone is not enough to determine the gradients' alignment, because a good proximity (ie. a small distance) can be defined by two opposed, but small gradients or by two long gradients that are already well aligned. Hence, I suggest weighting the computed proximity with the product of the two gradients' length. This will also reduce the influence of non-matching structures like soft tissue to the overall measure, as the respective gradients tend to be much smaller than the ones representing the outlines of bones or other rigid structures like landmarks. Further, the cosine of the angle spanned by the two vectors is also a good indication of alignment.
Computing the product between the cosine of the angle and the product of lengths is actually cheaper than computing the individual terms, as it evaluates to the vectors' dot product:

$$\cos(\alpha) = \frac{\overrightarrow{u} \cdot \overrightarrow{v}}{|\overrightarrow{u}||\overrightarrow{v}|} \Longrightarrow \cos(\alpha)|\overrightarrow{u}||\overrightarrow{v}| = u_x v_x + u_y v_y,$$

where $\overrightarrow{u}$ and $\overrightarrow{v}$ are the gradients in the two images and $\alpha = \angle(\overrightarrow{u}, \overrightarrow{v})$.
Depending on the measure of distance we choose, we obtain the final expressions as:

$$GP_{euclidian} = \sum_{x,y \in T} (1 - \frac{(u_x - v_x)^2 + (u_y - v_y)^2}{max\_distance})(u_x v_x + u_y v_y) \tag{2.11}$$

$$GP_{cityblock} = \sum_{x,y \in T} (1 - \frac{|u_x - v_x| + |u_y - v_y|}{max\_distance})(u_x v_x + u_y v_y) \tag{2.12}$$

As $max\_distance$ is an a-priori known constant, we can write the division as a multiplication with $frac1max\_distance$, which is cheaper to compute than a division.

2.2.5 Sum of Local Normalized Correlation

Although normalized cross correlation is invariant to linear changes in image intensity, in practice there can also arise the case of spatially varying intensity distortions. This effect can appear due to vignetting and non-uniformity in the imager response and have a significant impact on the registration accuracy of the Correlation Coefficient method. LaRose et al. [18] have therefore proposed a modification of the original algorithm which overcomes this problem.
The sum of local normalized correlation evaluates the correlation coefficient of many small image regions. These image regions can be chosen to be non-overlapping or overlapping, up to the point where the NCC will be computed in each voxel for a small image area centered around the respective voxel.

$$SLNC(I_1, I_2) = \frac{1}{|Q|} \sum_{p \in Q} CC(I_0, I_1, P(p)), \tag{2.13}$$

where Q is a set of voxel locations which span the data set for which the SLNC is to be computed and $P(p)$ stands for the neighborhood of point p, ie. the sub-image for which the NCC is assessed. The sum is then divided by the number of points in set Q, ie. the number of sub-images, to obtain a result between -1 and 1. LaRose used sub-images of 7x7 and 11x11 pixels.
Some of these small images might have a constant intensity over all pixels, which makes the numerator of the CC formula zero, yielding an undefined result. If the images to be registered are of the same modality, the correlation coefficient can arbitrarily be assigned a value of zero. In the case of 2D-3D registration, however, this method cannot be used anymore, because the the number, size and position of the constant patches will usually change with each DRR generation. Assigning zero values in this case would lead to discontinuities in the similarity measure. This is avoided by adding small magnitude gaussian noise to each DRR.

2.2.6 Variance-Weighted Sum of Local Normalized Correlation

SLNC can further be improved by weighting the individual correlation coefficients with the variance of the respective sub-image in one of the two original images (which is termed the *control image*)[18]. VWC thus attributes more importance to high-frequency image areas, which

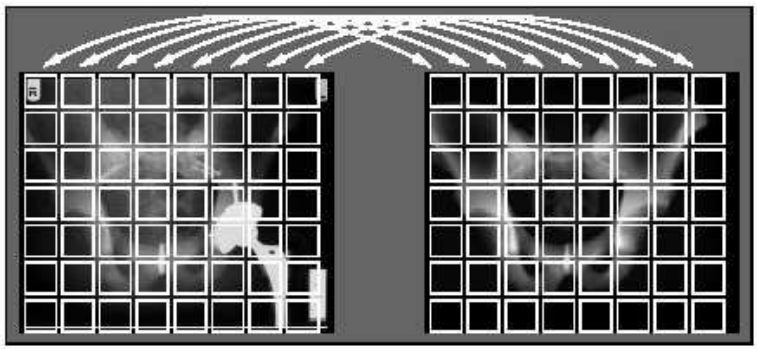


Figure 2.3: Sum of Local Normalized Correlation, from [18]

contain more information about the structures to be registered, and less to low-frequency areas.

$$\cdot VWC(I_1, I_2) = \frac{1}{|Q|} \sum_{p \in Q} Var(I_1) CC(I_0, I_1, P(p)) =$$

$$\frac{1}{|Q|} \sum_{p \in Q} \frac{\sum_{p \in Q} (I_1(x,y) - \overline{I_1})^2}{n} \cdot \frac{\sum_{x,y \in T} (I_1(x,y) - \overline{I_1}) \cdot (I_2(x,y) - \overline{I_2})}{\sqrt{\sum_{x,y \in T} (I_1(x,y) - \overline{I_1})^2 \cdot \sum_{x,y \in T} (I_2(x,y) - \overline{I_2})^2}} =$$

$$\frac{1}{|Q|} \sum_{p \in Q} \frac{\sqrt{\sum_{p \in Q} (I_1(x,y) - \overline{I_1})^2} \cdot \sum_{x,y \in T} (I_1(x,y) - \overline{I_1}) \cdot (I_2(x,y) - \overline{I_2})}{n \cdot \sum_{p \in Q} (I_2(x,y) - \overline{I_2})^2} \quad (2.14)$$

where $C(I_1, I_2, P(p))$ computes the variance of the control image in the neighborhood $P(p)$. Additionally, VWC does not simply compute the mean of the weighted sum of correlation coefficients, but a weighted average.

2.3 Information Theoretic Measures

We can think of registration as trying to maximize the amount of information shared between two images, or trying to minimize the amount of information present in the combined image. When the two images are perfectly aligned, all the corresponding structures will overlap, eliminating any duplicate elements that result from misalignment. Thus, registration works based upon a measure of *information*. The most commonly used measure of information is the Shannon-Wiener entropy [38], developed as part of communication theory (however, Wang et.al. [43] have proposed a different entropy concept which eliminates some drawbacks present in Shannon's approach):

$$H = -\sum_i p(i) \log p(i) \quad (2.15)$$

H represents the average amount of information supplied by a set of i symbols with their respective probabilities $p(1), p(2), ..., p(i)$. H reaches its maximum value when all probabilities are equal (ie. $p_i = \frac{1}{n}$), so any change in the data that tends to equalize the probabilities of the different symbols increases the entropy.

The distribution of the probabilities p_i can be represented as a histogram. In the case of digital images, histograms indicate the probability for a pixel to have a certain intensity, ie. the number of pixels having a certain intensity divided by the total number of pixels. Similarly, joint probability distributions are computed by counting the occurrences of pairs of intensities (a, b), ie. the number of pixel positions (x, y) where $I_1(x, y) = a \wedge I_2(x, y) = b$, which generates a two-dimensional histogram, or joint histogram.

2.3.1 Entropy of the Difference Image

The most straightforward information theoretic approach for assessing image similarity is computing the entropy of a difference image $I_{diff} = I_1 - s \cdot I_2$. If the images are matching perfectly, the difference image will have a constant intensity i, ie. $p(i) = 1$ and $p(j) = 0 \forall j \neq i$, which results in an entropy of zero.

2.3.2 Mutual Information

The joint entropy measures the amount of information present in a combination of two images:

$$H(I_1, I_2) = \sum_i \sum_j p(i, j) \log p(i, j), \tag{2.16}$$

with i and j belonging to the intensity range of the two data sets and $p(i, j)$ being the probability distribution function which can be visualized as a joint histogram as described above. If I_1 and I_2 are completely unrelated, $p(i, j) = p(i) \vee p(i, j) = p(j) \forall (i, j)$, so $H(I_1, I_2) = \sum_i p(i) \log p(i) + \sum_j p(j) \log p(j) = H(I_1) + H(I_2)$. With increasingly similar images, the amount information in the combined image, ie. the value of the joint entropy, decreases until, for identical images, it contains just as much information as each individual image, ie. $H(I_1, I_2) = H(I_1) = H(I_2$. The idea behind mutual information is now to combine the calculation of the individual and the joint entropies:

$$MI = H(I_1) + H(I_2) - H(I_1, I_2) = \sum_{i,j} p(i, j) \log \frac{p(i, j)}{p_1(i) p_2(j)} \tag{2.17}$$

In the case where I_1 and I_2 are completely unrelated, MI reaches its minimum value of 0. For identical images, $H(I_1, I_2) = H(I_1)$, so $MI = H(I_2) \leq \log n$, where n is the number of histogram bins. If normalization is desired, which usually is the case, the following scheme can be used[8]:

$$MI' = \frac{2MI}{H(I_1) + H(I_2)} = 2 - \frac{2H(I_1, I_2)}{H(I_1) + H(I_2)} \tag{2.18}$$

Alternatively, MI can be normalized by using the ratio between the individual and the joint entropies rather than their difference:

$$MI'' = \frac{H(I_1) + H(I_2)}{H(I_1, I_2)} = \frac{1}{MI' - 2} \tag{2.19}$$

Mutual Information assumes only a statistical dependency between the intensities in the two images, but no functional one. This property has made MI a very popular measure, especially

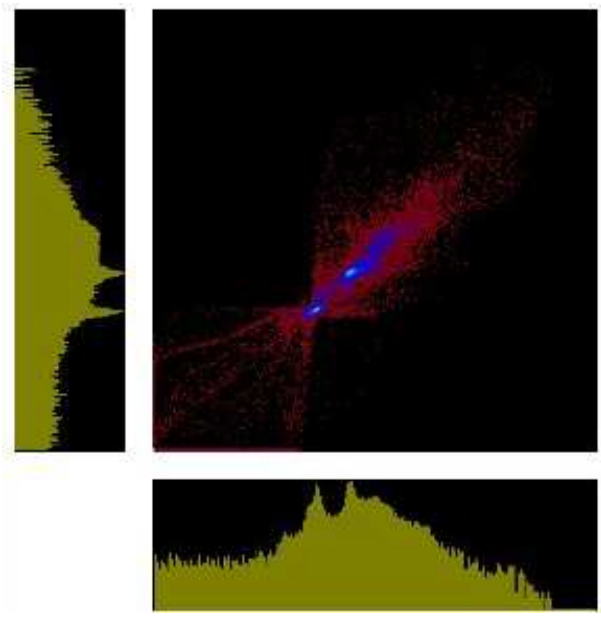


Figure 2.4: Joint and individual histograms of two images, from [45]

in the case of inter-modality registration, since images that provide the same information are found to be identical independently of the way in which they represent the information, ie. on the correspondance between tissue types and intensity ranges. More details about Mutual Information and its theoretic background can be found in [41] and [40].

3 GPU-based acceleration

3.1 Motivation

Since the advent of graphics boards with the ability to perform computations necessary for the rendering of 3D images (like shading, lighting and texturing) directly on the graphics processor (GPU), leaving the CPU available for other tasks, the market of 3D-accelerated consumer graphics boards has been one of the most dynamic segments of the semiconductor business. With a large 3D-gaming community demanding ever increasing frame rates and more sophisticated visual effects, off-the-shelf graphics hardware has evolved at a rate much higher than dictated by Moore's law over the past few years, reaching the point of GPUs having five or more times as many transistors as the fastest consumer-level CPU, executing almost seven times as many floating point operations per second and working with an internal memory bandwidth of over 35GB/s [9]. Recent graphics hardware even offers a programmable rendering pipeline, allowing an application developer to load and execute custom programs on the GPU. While primarily intended to allow for a larger range of visual effects, this feature also opened the gate for virtually any kind of application to take advantage of modern graphics hardware's capabilities, be it robot motion planning [19], flow visualization [46], segmentation [36], FFT computation [24], solving sets of algebraic equations [17], or even audio processing [3] and many more (see [10] for a good overview).

3.2 The Rendering Pipeline

The role of the GPU mainly consists in displaying geometrical shapes (together with appropriate lighting, shading and other effects) on the screen or a certain area of it, ie. a window. In order to achieve this, the data describing the visual scene runs through the so-called *rendering pipeline*. This pipeline has several stages dealing with certain aspects of the rendering process. John Purcell et al. [35] have even proposed an abstraction of the GPU, describing it as a general-purpose stream processor.

As a first step, the three-dimensional geometric data, which is usually made up of triangles, is converted into two-dimensional shapes using either an orthogonal or projective transformation, which the user can usually influence by setting different parameters like the field of view or display aspect ratio. Additionally, information needed for shading and lighting of objects, like normals, distances to light sources and texture coordinates, are also calculated in the geometry stage of a GPU's rendering pipeline. All these transformations and other attributes are computed for each vertex (corner point of a triangle or other shape) in the scene. These values are then interpolated for each pixel that is actually rendered to the screen, without being recomputed for each pixel.

Next, the two-dimensional geometry must be mapped to the discrete pixel positions on the screen, a process known as *rasterization*. Each point of the resulting image contains such information as color and depth. Thus, rasterizing a primitive consists of two parts. The first is to determine which rectangles of an integer grid in window coordinates are occupied by the primitive. The second is to assign each such discrete rectangle, or pixel (also called *fragment*), a color, a depth value and a texture coordinate. *Textures* are two-dimensional images that can be "wrapped around"the 3D geometry. *Texture mapping* thus corresponds to the process of applying wallpaper to a real object. The results of this process are then passed on to the next stage of the pipeline.
For each fragment, the final color value can now be computed from the available data and written to the appropriate location in the *frame buffer*, a piece of memory allocated to hold the color values for every pixel in one frame or picture. The most basic operations include shading, lighting and texturing but can be extended to very complex visual effects.

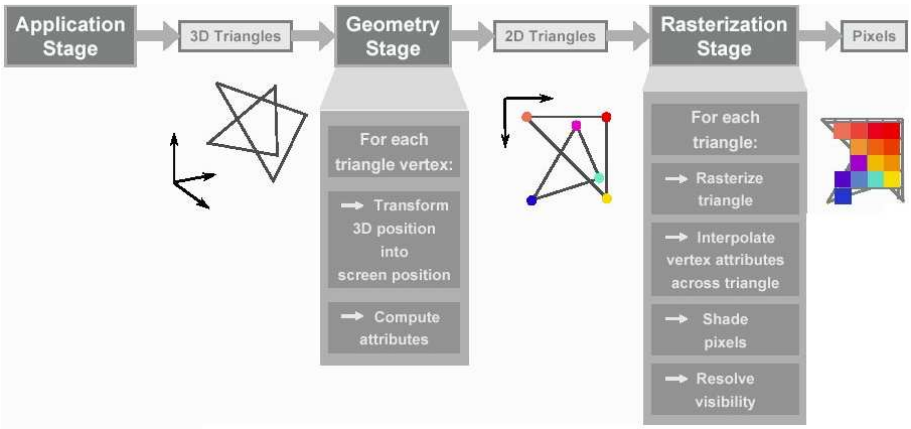


Figure 3.1: Rendering pipeline, from [7]

Classic rendering pipelines offered a fixed functionality, merely allowing the user to set various parameters and enable or disable certain effects, eg. lighting. With the high increase of performance and new capabilities of more modern GPUs, such an inflexible layout has become inadequate, not allowing graphics programmers to make full use of the hardware's features.

As a consequence, graphics card manufacturers have designed programmable rendering pipelines, allowing the user to replace the otherwise fixed functionality of the vertex and fragment shading engines with custom code (referred to as vertex shaders and pixel shaders or fragment programs, respectively), thus allowing for a virtually unlimited number of visual effects to be implemented.

3.3 Characteristics of GPU Computation

While CPUs are designed to execute program code made up of sequential instructions, graphics processors are used in a completely different environment. In computer graphics, the various elements like vertices or pixels are independent from one another, thus allowing for

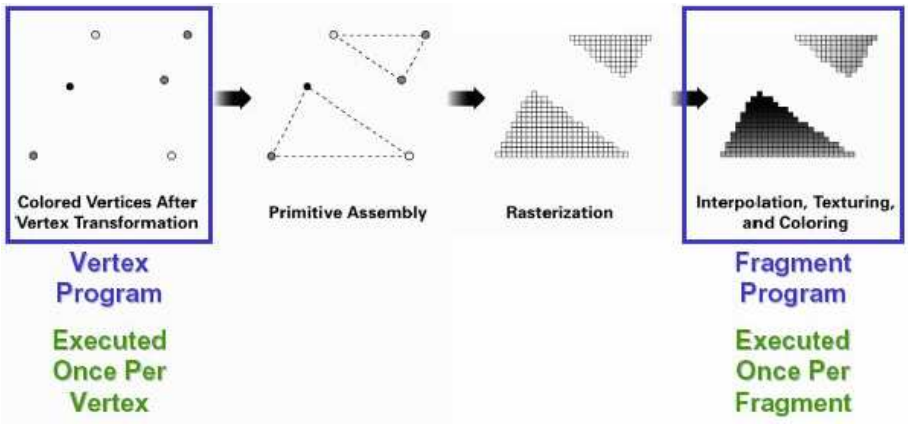


Figure 3.2: Programmable rendering pipeline, from [7]

highly parallelized computation. Hence, graphics boards have several pipelines for vertex and fragment processing, while the number of fragment units is usually larger due to the realistic assumption that the number of vertices in a scene will usually be much lower than the number of pixels. As an example, the NVIDIA Geforce 6 Series GPU has 16 separate pixel pipelines (but only 6 vertex pipelines), each one being capable of performing 8 pixel shading operations per clock cycle, leading to a total of 128 per-fragment operations that can be computed in a single clock [28].

While graphics processors offer many interesting features like parallel computation and very high memory bandwidth and gigaflop counts, it is important to note that they are specialized processors designed for graphics. Therefore the whole programming environment differs from that of classic software, not allowing for direct porting to the GPU of code written for the CPU. The programming model is tuned for graphics and not intended for general purpose applications, imposing several restrictions. These may vary across different programming languages and hardware platforms, however some of them result directly from the kind of application GPUs are designed for [9]. As a first, parameters passed to vertex and fragment shaders represent elements related to the rendering process (eg. transformation matrices or texture objects) which exist on the GPU. This means that the programmer cannot pass arbitrary parameters to a shader (except for the case were such a value is to remain the same for every invocation of the shader, ie. every fragment, were the application developer can pass a so-called *uniform* parameter), and especially that shaders cannot write data to a parameter they receive. Second, the CPU cannot directly access shaders' return values, since those are passed to the next step in the rendering pipeline, ie. the vertex unit passes the result on to the rasterizer which makes it available to the fragment unit and the fragment unit updates the output buffer. The user can merely choose to what stage of the pipeline the output is being written, eg. he can choose between the depth buffer and the frame buffer (or both) when computing a fragment. Third, the vertex and fragment engines are designed to work independently and in parallel, so there is no possibility for sharing data between units, either by using global variables or message passing. This also implies that a shader cannot write values to one of it's input parameters, eg. a vertex shader cannot modify a geometry transformation matrix and a fragment shader cannot write to a texture object. Furthermore,

restrictions may apply to the available data types (eg. there are no pointers available on the GPU), available language constructs (like conditional statements, loops or recursive function calls) or a maximum number of instructions per vertex or fragment program. However, these may vary a lot with the used hardware and programming language so it is hardly possible to provide a full list in this paper [26][14].

Computer graphics makes extensive use of vectors and matrices, so GPUs provide direct hardware support for such data types. In computer graphics, geometric points are represented using homogeneous coordinates, which add a fourth coordinate w to the three euclidian ones. Thus, the GPU can compute operations with vectors of up to four elements. This means that by properly vectorizing the code, a speedup of factor four can theoretically be achieved. It is also important to note that shading language compilers do perform vectorization (along with other optimizations).

Thus, before an application ca be ported from CPU to GPU, one must first identify some kind of analogy between the respective application and the graphics rendering process, matching input to geometry and texture elements, output to the various buffers and adapting the computational steps to the capabilities of the vertex and fragment shading units.

3.4 Computational Overhead

When considering the implementation of custom algorithms on graphics hardware, one should be aware of the fact that during rendering not only the programmed fragment and vertex shaders will be executed, but that there is a certain amount of overhead implied by the rendering pipeline architecture. While some of the unnecessary operations (ie. depth testing) can be disabled, others are an integral part of the rendering process. These include draw-buffer swapping, copying of data from the color-buffer to texture objects and switching rendering contexts when using auxiliary draw buffers (see 3.7.5). As all of these operations are independent of the actual computation carried out in the vertex and fragment shading unit, the relevance of the imposed overhead will strongly depend on the complexity of the implemented custom shader functionality. In the case of very simple shaders, most of the time will be spent on operations that are not of direct interest, while more complex shading programs will reduce the percentage of time spent on overhead operations, thus allowing for a much better exploitation of the GPU's capabilities.
Thus we can expect simple similarity measures (like SAD and SSD) to perform rather poorly when compared to the CPU-implementation, while more complex measures like PI or LNC have the potential for much better speed-up. But even if the computation of the similarity measure itself should not be faster on the GPU, the ability to assess image similarity on the graphics board eliminates the need for the very slow copying of data from the frame buffer to the main system RAM when generating the DRRs on the video board.

3.5 Data Transfer between GPU and RAM

Most modern graphics boards are connected to the computer's mainboard through an AGP (Advanced Graphics Port, or Accelerated Graphics Port) bus. AGP was introduced in 1997 by Intel and builds upon the PCI. The differences are the following:

- AGP implements a point-to-point architecture. This means that each AGP device has its own pathway to communicate with the CPU and the main memory, whereas all PCI devices must share the same bandwidth.
- The first version, AGP 1x had a clock speed twice as high as PCI (66Mhz as opposed to 33Mhz). The later versions, AGP 2x, AGP 4x and AGP 8x increased both the clock cycle (266Mhz for AGP 8x) and used a double-pumped design, which allows data transfer on both the rising and the falling edges of the clock cycles, thus doubling the data transfer rate while keeping the clock rate constant. AGP 8x has a bandwidth of 2.1GB/s (as opposed to 133MB/s possible for PCI).
- AGP has an asymmetric bandwidth. While up to 2.1GB/s can be uploaded to the GPU with AGP 8x, the download of data is limited to 133MB/s, ie. the speed of a PCI bus. This low performance is the reason why we would like to avoid copying of data from GPU to RAM.

The newest replacement for both PCI and AGP is PCI Express (also called 3GIO - 3rd Generation I/O). Like AGP, PCI Express is based to point-to-point connections so devices will not have to share the available bandwidth. Furthermore it allows devices to use multiple PCI Express lanes, where the next graphical bus interface will use 16 such lanes, which results in a bandwidth of 4.2GB/s. More important, unlike AGP, this bandwitdh is available for both data upload and download from the GPU. Thus, transferring data from the GPU to the system's main RAM might stop being an issues in the near future.

3.6 Shading Languages

At first, shading programs had to be written using hardware-specific assembler languages, thus requiring each shader to be separately implemented for each graphics card it was meant to run on, in the respective assembler language. As an alternative to the inflexible and often tedious assembler programming of the rendering pipeline's steps, several high level shading languages have been developed[22].

RenderMan was the first shading language to be developed by Pixar [33]. It has meanwhile become a standard in the domain of offline rendering. Although RenderMan was initially intended as a hardware API, it quickly became too complex and sophisticated for a hardware implementation. However, it strongly influenced the design of real-time shading languages, especially concerning the distinction between varying and uniform parameters.

The SGI Interactive Shading Language [31] is treating OpenGL as a general SIMD computer and translates a high-level description of shading effects into a multi-pass OpenGL rendering process. As such, it works on top of the OpenGL API and is therefore independent of the underlying GPU hardware. Furthermore, the basic techniques it uses can also be implemented on top of any other graphics API that offers the possibility of circulating data through the rendering pipeline.

Cg (C for Graphics)[25][26] was one of the earliest high-level shading languages designed to run on commodity graphics hardware and was developed by NVIDIA in collaboration with Microsoft. It was originally released in December 2002 and is, as the name implies, based on the ANSI C programming language, with a few restrictions and extensions demanded by the nature of GPU hardware. NVIDIA has tried hard to make Cg an industry-standard for shader programming, and most graphics hardware manufacturers offer support for this language. In its initial release, Cg code could be compiled using 14 different profile targets using DirectX 8, DirectX 9 and OpenGL in order to offer compatibility with multiple hardware platforms. More profiles continue to be added in order to offer support for the newest features offered by both newer hardware and graphics APIs. Cg is the only commercially available shading language offering support for both DirectX and OpenGL. As an addition to Cg, several other tools have been developed, including plug-ins for 3D- and CAD-programs, like Maya or 3dsmax, and CgFX[27]. CgFX allows programmers to describe a whole rendering effect instead of a single shader with fixed functionality. The specification of such an effect can include multiple rendering passes, use of varying shader profiles, assembly-language shaders or GPU fixed functionality and editable parameters. Furthermore, the CgFX file format allows for the encapsulation of multiple rendering techniques, or different ways to achieve a desired visual effect (eg. bump mapping), thus making fall-backs for level-of-detail, functionality and performance possible. Like Cg, CgFX also offers cross-API and cross-platform compatibility.

Microsoft's High Level Shading Language (or HLSL) was co-developed with NVIDIA, and has initially been separated from Cg mainly for branding purposes. Although NVIDIA and Microsoft have, at some point, continued to independently develop their respective shading languages, the main difference between Cg and HLSL remains the fact that HLSL can only be compiled to DirectX code. As a counter-part to CgFX, Microsoft offers .fx, which actually represents an identical file format with the same possibilities, but is again only compatible with Microsoft's proprietary graphics API DirectX.

The OpenGL Shading Language [15] (also referred to as GLSL or glslang) will be part of the upcoming OpenGL 2.0 specification and is intended to become the standard for programming GPUs under the OpenGL API. Just like Cg and HLSL, GLSL is based on ANSI C and offers the same set of features, but is optimized for use within an OpenGL environment, offering a seamless and transparent integration into current applications in order to achieve high acceptance among programmers. Similar to texture objects, the concept of program objects has been introduced. These may contain multiple shader objects, which encapsulate the actual source code meant to be executed by the GPU's vertex and fragment shading units or providing useful functions to those, thus acting as a shader library. The programmer can create, link and enable program objects using OpenGL API calls[4].

Brook for GPUs [5] is a scientific computing language treating the GPU as a stream processor, thus offering a way for reformulating the otherwise graphics-tied rendering process as a set of kernels acting upon data streams. Brook extends C to include simple data-parallel constructs, which are then processed using a Brook compiler and a Brook Runtime. The compiler maps Brook kernels to Cg shaders and data streams to floating-point textures and it also generates the respective C++ application code which uses the Brook Runtime, or BRT, to invoke those kernels. BRT is an architecture-independent software layer which provides a common interface for each of the supported target platforms, namely OpenGL, DirectX (running on both NVIDIA and ATI boards) and also a CPU reference implementation.

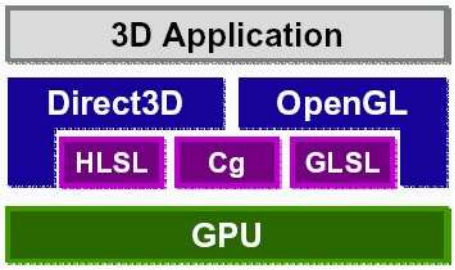


Figure 3.3: Compatibility between shading languages and graphics APIs, from [7]

3.7 Implementation Prerequisites

3.7.1 Vertex and Fragment Shaders

In order to display a 2D image on the screen using OpenGL, the most simple way is to render a view-aligned quad spanning over the entire viewport area and apply the image as a texture. As we are dealing with intensity-based registration, all of our operations will be performed on the values stored within the texture object, ie. in the fragment shading stage of the rendering pipeline, which means that every single operation implied by the respective similarity measure will be carried out for each individual pixel. In the case of measures using spatial information, this also implies that we compute the texture coordinates of each regarded fragment, relative to the current position, in each fragment shader. However, because this process of running through a neighborhood of an a-priori known, and fixed, size is identical for each pixel, it makes much more sense to complete it in the vertex shader.
Texture coordinates are computed within the vertex shading unit for each vertex (of which we have only four, as we are rendering a quad), and are then interpolated for each pixel within the respective geometric shape, ie. our view-aligned quad. We can then use the interpolated coordinates of the pixels within our neighborhood in the fragment program, without having to calculate them each time. However, as there are only eight texture coordinate units on the GPU where we can store the vertex shader's output, we can pre-compute at most eight coordinates in this way, leaving the rest to the fragment program.
In the case of measures that do not use spatial information, like SAD or NCC, one can load an empty vertex shader that simply passes on the four vertices' texture coordinates to the fragment unit, without actually computing anything else. In this way, one can make sure that the GPU will not waste any time carrying out any unnecessary computations for vertex shad-

ing, like normal or lighting computation. However, as we are dealing with only four vertices, the performance increase obtained through this method will not be noticeable in practice.
It should also be noted that accessing texture data is slower than performing mathematical operations. Therefore, shaders performing complex operations on a small number of texture pixels will perform better than those implementing a very simple functionality. As the presented similarity measures require rather few operations per pixel, the GPU's fragment shading unit will spend a considerable amount of time on fetching texture data rather than executing program code.

3.7.2 Handling Negative Values

Graphics hardware is designed to compute color intensities meant to be displayed on a screen, ie. light intensities for different color channels. These intensities are represented by values in the range [0...1]. This means that all values outside of these limits will be clamped to the respective bound, as they obviously do not represent meaningful intensities. This means especially that, while negative values can be used inside shader program, they cannot be written to the color buffer. Since many similarity measures do produce negative values, some workaround is required.
A first possibility would be to normalize all values to the [-1...1] interval and then shift them to [0...1] by applying a $0.5 \cdot (x+1)$ scheme. As traditional color buffers only offer 8-bit accuracy, ie. 256 values spanning the [0...1] range, this approach of interval compaction would only leave 7 bits of accuracy, which is definitely too low.
Another possibility is to use two color channels for storing values that might be negative as well as positive. One can store the unmodified value in the, let's say, red channel and the negated value in the green channel: $R = x, G = -x$. This means that one channel will always be zero (if $x > 0$, the green channel and if $x < 0$, the red channel) while the other one contains the absolute value. To reassemble our actual number inside a shading program or on the CPU, we can compute $x = R - G$. While this method does not impose any loss of accuracy, the obvious disadvantage is that it occupies two of the four available color channels, further reducing the number of output values that we can compute within a shader. Fortunately, in the case of the measures I have implemented, the four RGBA channels always prove sufficient for this method.
The last solution is to use a 16 (or 32) bit floating-point buffer for rendering instead of a classical 8-bit buffer. However, as floating-point values do not represent meaningful color intensities, the contents of such buffers cannot be directly displayed on the screen, which also implies that one cannot simply configure the standard frame-buffer to offer floating-point precision. I will describe the creation and usage of such off-screen rendering buffers in section 3.7.5.

3.7.3 Handling Gradient Values

Gradient values can be up to four times larger than the original image intensities. The fixed image's gradient will be stored as a texture that can contain only values between 0 and 1, which also is the range of the original intensities. The intuitively best possibility to accomodate the gradient's data range would be to divide it by 4, possibly multiplying it again by 4 when assessing the similarity between the precomputed fixed gradient and the moving one

(which is not stored within a texture object but computed within the fragment shading unit, where no such data range restrictions apply).
Although mathematically incorrect, another possibility is to simply ignore the problem. In this way, all values greater than the maximum representable intensity automatically get clamped to 1 when written to an 8-bit frame buffer. When computing the moving image's gradient, one would thus have to clamp the result to the same data range as well in order to make sure that the two images' gradient values correspond. However, during my experiments I have found that this almost always resulted in a slightly smaller registration accuracy, while clamping the fixed gradient but leaving the other one unchanged, seems to deliver better results.

3.7.4 Computing the Similarity Measure from the Similarity Image

Using fragment shaders, the GPU computes a two-dimensional image holding a measure of similarity for each pixel location. Since we need one single value denoting the image similarity, we need to find a way of computing this single value from the individual pixel intensities. This always involves obtaining the average of intensities, or their sum, for the individual color channels, which can already be the required measure (eg. in the case of measures based on a difference image) or which can represent different factors in the final formula (eg. in the case of the correlation coefficient).

CPU averaging One approach for solving this problem is to simply copy the whole image from video memory into the system's RAM and compute the average value on the CPU. However, this solution, although delivering very accurate results, contradicts our intention of eliminating the slow copying of data between RAM and GPU.

Mipmaps A GPU-based solution for averaging an image's contents is provided by mipmaps. Mipmaps represent scaled down versions of a texture image, each mipmap having half the width and height of the previous one (note that the original image must have power-of-two values for width and height). Thus, from a 32x32 image, we can recursively generate 16x16, 8x8, 4x4, 2x2 and 1x1 pixel mipmaps by computing each pixel as the average of four input pixels.

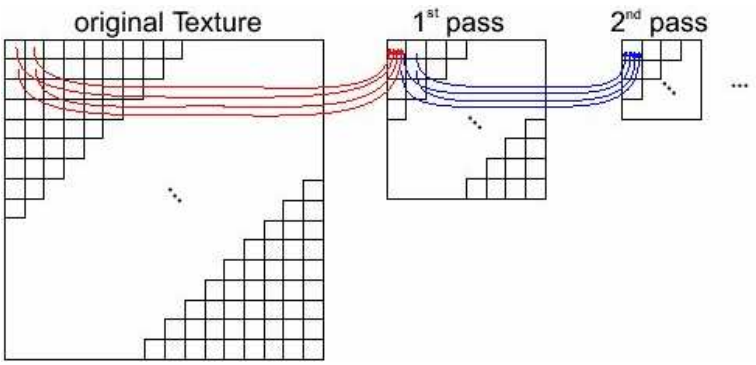


Figure 3.4: Mipmap generation, from [17]

Many modern video boards, like the NVIDIA Geforce GPUs, support the `SGIS_generate_mipmap` OpenGL extension which enables mipmaps to be automatically generated for a texture object as soon as its contents change (eg. when a `glCopyTexImage(...)` `is executed`. This method is much faster than the first approach (on a system containing an AMD Athlon XP 1800+ CPU and a Geforce FX 5200 GPU, the time for averaging a 256x256 image was 7.9ms on the CPU and 1.8ms using mipmaps), but is in turn less accurate.
This is due to the repeated rounding involved by mipmap generation: computing all mipmaps for a 256x256 image involves 8 steps, where in each step 4 pixel intensities are being averaged into one output value. The value will be rounded up to the next higher integer value (assuming 0...255 intensities). This repeated rounding, combined with the fact that we only have 8-bit accuracy, can lead to a result far from the actual average value. With the number of mipmapping steps being $n = log_2(imagesize)$, we have: $AVG_{SGIS} < n * 0.75 + AVG_{real}$.
On Geforce FX boards, this automatic mipmap generation is only available for 8-bit textures, since 16 or 32-bit textures are assumed to be of type `NV_TEXTURE_RECTANGLE` (instead of `GL_TEXTURE_2D`), which is not restricted to power-of-two dimensions. The newer Geforce 6 boards also support automatic mipmap generation for floating-point textures, where the error accumulated through repeated rounding will also be much smaller due to the higher 16 or even 32-bit accuracy. Unfortunately, the respective feature has not yet been implemented into current drivers, so that I could not evaluate it.
Also note that the 1x1 mipmap remains black (although this may also vary depending on hardware and driver version), so one has to use the 2x2 image. However, this only implies the copying of 3 additional pixel values (with 4 bytes for each of them, assuming 8-bit RGBA texture format) which has no noticeable impact on performance.

As an alternative to this method, we could also use a custom fragment shader to execute the averaging. However, this proves to be even slower than processing everything on the CPU, since it requires $log2(imagesize)$ processing steps (8 for the 256x256px images commonly used in our application), each one involving the rendering of the previously generated mipmap and copying the result to a texture object to be used for the next step. Also, this approach can merely reduce the maximum rounding error made in each mipmap generation step from 0.75 to 0.5 (since we can decide wether we will round up or down instead of always rounding up as is the case with the automatic mipmap generation). This, combined with the slow performance, makes this option for average computation an unattractive one.

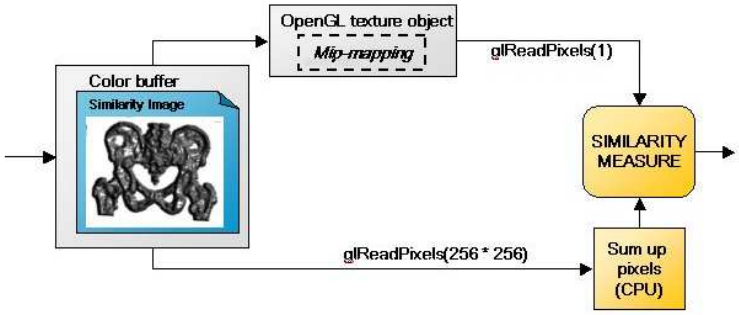


Figure 3.5: Computing the Similarity Measure from the Similarity Image

None of the methods is very attractive by itself, as choosing accuracy or speed as the

prioritary factor will have a very heavy impact on the other one.
Therefore, we can try to combine the advantages of the two methods in a hybrid approach that offers both a high speed and a good accuracy. Reducing the texture size by using mipmaps on the GPU obviously decreases the amount of data that we will have to transfer back to the CPU. While the width and height of the mipmaps is halved with each step, it is important to keep in mind that the texture's area, or the number of pixels it contains, is a quadratic function of width and height. Hence, only the first few mipmapping steps will considerably reduce the data amount to be transferred (and thus the time to transfer it), while the accumulated rounding error increases linearly with each step. Therefore it should prove much more efficient to carry out the first 2 to 4 averaging steps using mipmaps and, after copying the now much smaller texture image, complete the averaging on the CPU without introducing more rounding errors.

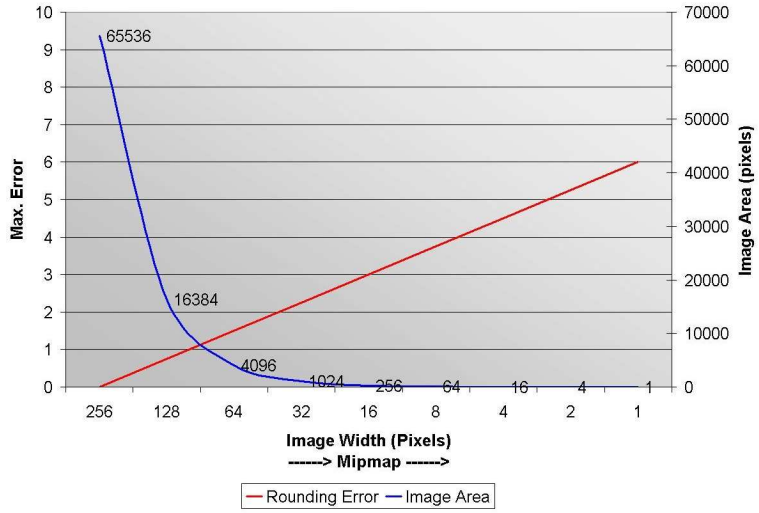


Figure 3.6: Effect of mipmaps on texture size and rounding error

3.7.5 Pbuffers

Pbuffers (pixel buffers) are off-screen render buffers that support 8, 16 or 32 bits per color channel and that can also be bound to texture objects and used like a normal texture without the need for copying any data. Each pbuffer is a separate device context and OpenGL rendering context, so switching between the frame buffer and a pbuffer implies a certain overhead. It should be noted here that in Microsoft DirectX, a pbuffer is simply a color buffer, so switching the rendering target will be much faster there.
Creation and usage of such buffers require various hardware and OS-specific extensions:
For Windows: `WGL_ARB_pbuffer, WGL_ARB_pixel_format, WGL_ARB_render_texture`
For Unix: `GLX_ARB_pbuffer, GLX_ARB_pixel_format, GLX_ARB_render_texture`

Usage Using a pbuffer for rendering is quite simple, once it has been created and initialized.

- Set the pbuffer as current rendering context

Windows OS	Unix OS
`wglMakeCurrent(pbuf_hDC, pbuf_hRC)`	`glXMakeCurrent(m_pDisplay, m_glxPbuffer, m_glxContext)`

- Render

- Go back to the main rendering context

Windows OS	Unix OS
`wglMakeCurrent(main_hDC, main_hRC)`	`glXMakeCurrent(m_pDisplay, m_glxPbuffer, m_glxContext)`

and then bind the pbuffer to a texture object (only possible under Windows):
`glBindTexture(GL_TEXTURE_2D, texobj);`
`wglBindTexImageARB(hBuf, WGL_FRONT_LEFT_ARB);`
OR

- Copy the pbuffer's content to a texture object, then switch back to the main context:

Windows OS	Unix OS
`glBindTexture(GL_TEXTURE_2D, texobj)` `glCopyTexSubImage2D(GL_TEXTURE_2D, ...)`	
`wglMakeCurrent(main_hDC, main_hRC)`	`glXMakeCurrent(m_pDisplay, m_glxPbuffer, m_glxContext)`

Creating a pbuffer Creating a pbuffer can be rather tedious, so building a reusable class surely is a good idea.
First of all, one should save the main device and rendering contexts, so one can switch back to them after rendering is complete.

Windows OS	Unix OS
`HDC mainDC = wglGetCurrentDC();` `HGLRC mainRC = wglGetCurrentContext();`	`Display *m_pMainDisplay = glXGetCurrentDisplay();` `GLXPbuffer m_glxMainDrawable = glXGetCurrentDrawable();` `GLXContext m_glxMainContext = glXGetCurrentContext();`

Then, two lists of pbuffer attributes must be set: one list for the buffer's pixel format and one for the pbuffer itself. This will be explained in more detail in the next section. Once those attributes have been set, an appropriate pixel format can be obtained by calling:

Windows OS	Unix OS
	`Display *pDisplay = glXGetCurrentDisplay(); int iScreen = DefaultScreen(pDisplay);`
`wglChoosePixelFormatARB(mainDC, &pfAttribList[0], NULL, MAX_PFORMATS, pformats, &iformats); //pfAttrbisList contains the pixel format attributes //MAX_PFORMATS gives the length of the pformats array (1 is enough) //pformats is an integer array in which the available pixel formats will be stored //iformats is the number of available pixel formats`	`GLXFBConfig * glxConfig = glXChooseFBConfigSGIX(pDisplay, iScreen, &pfAttribList[0], &iformats); //pfAttrbisList contains the pixel format attributes //glxConfig is a pointer to an array containing iformats pixel formats`

If there are any pixel formats available, you can create your pbuffer and get the other necessary handles.

Windows OS	Unix OS
`HPBUFFERARB hBuf = wglCreatePbufferARB(mainDC, pformats[0], m_width, m_height, &pbAttribList[0]);)`	`glxPbuffer m_glxPbuffer = glXCreateGLXPbufferSGIX( pDisplay, glxConfig[0], m_width, m_height, &pbAttribList[0]);`
`HDC m_hDC = wglGetPbufferDCARB(hBuf);`	`GLXContext m_glxContext = glXCreateContextWithConfigSGIX( glXGetCurrentDisplay(), glxConfig[0], GLX_RGBA_TYPE, glXGetCurrentContext(), true);`
`HGLRC m_hRC = wglCreateContext(hDC);`	`Display *m_pDisplay = glXGetCurrentDisplay();`

Now, since pbuffers represent a separate rendering context, the texture objects, display lists and shader programs from the main context are not available while rendering to the pbuffer. However, all those things can be shared across the two contexts. On Windows OS, this can be achieved by calling `wglShareLists(mainRC, m_hRC)` immediately after creating the pbuffer. Under Unix, the function `glXCreateContextWithConfigSGIX(...)` takes care of sharing these elements with the context it gets as parameter (`glXGetCurrentContext()` in the upper example).

Choosing a pixel format Note: `pfAttribList` represents the pixel **f**ormat attributes and `pbAttribList` the **b**uffer's. Both attribute arrays consist of pairs of values, where the

first one (a WGL or GLX constant) indicates the respective attribute, while the second represents the desired value for the specified attribute. A value of zero indicates the end of the list.
The following are the settings I used for my implementation and are supposed to offer a general guideline on setting up pbuffers. It is important to keep in mind that some attributes explicitly require, or exclude, the setting of other attributes. Detailed information on such restrictions can be found in SGI's OpenGL Extension Registry[37] and in NVIDIA's[29] and ATI's[2] extensions specifications. Apart from this well-specified relationship between attributes, I have also noticed that certain attributes may have a severe impact on the pbuffer's performance regarding functionality not directly related to the respective attribute. I describe one such issue at the end of this section. These problems might be related to driver issues, so one should best carry out some experiments with different pbuffer configurations to determine its behavior on the actual target system.

Basic attributes

Windows OS	Unix OS
`pfAttribList.push_back (WGL_DRAW_TO_PBUFFER_ARB); pfAttribList.push_back(true);`	`pfAttribList.push_back (GLX_DRAWABLE_TYPE); pfAttribList.push_back (GLX_PBUFFER_BIT);`
`pfAttribList.push_back (WGL_SUPPORT_OPENGL_ARB); pfAttribList.push_back(true);`	`pfAttribList.push_back (GLX_RENDER_TYPE); pfAttribList.push_back (GLX_RGBA_BIT);`
`pbAttribList.push_back (WGL_PBUFFER_LARGEST_ARB); pbAttribList.push_back(false);`	`pbAttribList.push_back (GLX_LARGEST_PBUFFER); pbAttribList.push_back(false);`
`//if set to 'true', you will get the largest pbuffer available, which might be smaller than the size required`	
	`pbAttribList.push_back (GLX_PRESERVED_CONTENTS); pbAttribList.push_back(true);`

Precision attributes

Windows OS	Unix OS
pfAttribList.push_back (WGL_RED_BITS_ARB); pfAttribList.push_back (color_bits); pfAttribList.push_back (WGL_GREEN_BITS_ARB); pfAttribList.push_back (color_bits); pfAttribList.push_back (WGL_BLUE_BITS_ARB); pfAttribList.push_back (color_bits); pfAttribList.push_back (WGL_ALPHA_BITS_ARB); pfAttribList.push_back (alpha_bits); pfAttribList.push_back (WGL_DEPTH_BITS_ARB); pfAttribList.push_back (depth_bits);	pfAttribList.push_back (GLX_RED_SIZE); pfAttribList.push_back (color_bits); pfAttribList.push_back (GLX_GREEN_SIZE); pfAttribList.push_back (color_bits); pfAttribList.push_back (GLX_BLUE_SIZE); pfAttribList.push_back (color_bits); pfAttribList.push_back (GLX_ALPHA_SIZE); pfAttribList.push_back (alpha_bits); pfAttribList.push_back (GLX_DEPTH_SIZE); pfAttribList.push_back (depth_bits);

If one needs more than 8 color bits per channel and uses an NVIDIA GPU, the pbuffer's components must be of floating-point type.

Windows OS	Unix OS
pfAttribList.push_back (WGL_FLOAT_COMPONENTS_NV); pfAttribList.push_back(true);	pfAttribList.push_back (GLX_FLOAT_COMPONENTS_NV); pfAttribList.push_back(true);

ATI GPUs also support 16-bit fixed-point values, but in case one requires floats, the following attributes must be set instead of the above:

Windows OS	Unix OS
pfAttribList.push_back (WGL_PIXEL_TYPE_ARB); pfAttribList.push_back (WGL_TYPE_RGBA_FLOAT_ATI);	not supported not supported

Texture binding attributes Note that on Unix-based operating systems, the required extensions for binding pbuffers to texture objects are currently not supported.

```
//8 bit pbuffer
  pfAttribList.push_back(WGL_BIND_TO_TEXTURE_RGBA_ARB);
  pfAttribList.push_back(true);
  pbAttribList.push_back(WGL_TEXTURE_TARGET_ARB);
  pbAttribList.push_back(WGL_TEXTURE_2D_ARB);
  pbAttribList.push_back(WGL_TEXTURE_FORMAT_ARB);
  pbAttribList.push_back(WGL_TEXTURE_RGBA_ARB);
//16 or 32-bit pbuffer
```

```
pfAttribList.push_back(WGL_BIND_TO_TEXTURE_RECTANGLE_FLOAT_RGBA_NV);
pfAttribList.push_back(true);
pbAttribList.push_back(WGL_TEXTURE_TARGET_ARB);
pbAttribList.push_back(WGL_TEXTURE_RECTANGLE_NV);
pbAttribList.push_back(WGL_TEXTURE_FORMAT_ARB);
pbAttribList.push_back(WGL_TEXTURE_FLOAT_RGBA_NV);
```

Binding a pbuffer to a texture object Pbuffers have the ability to be bound to a previously created texture object and be used directly as textures, without the need for copying any data from rendering buffer to texture.
However, this method is not a good choice due to the following issues:

1. It is slower than copying the pbuffer contents to the texture object using `glCopyTexSubImage2D(...)`
2. It makes the pbuffer *extremely* slow if you want to use `glCopyTexSubImage2D(...)` with it (besides binding)
3. It does not allow you to bind the pbuffer to a texture object with automatic mipmap computation turned on (`glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP_SGIS, GL_TRUE)`) . According to NVIDIA whitepapers, using
   `pfAttribList.push_back(WGL_MIPMAP_TEXTURE_ARB);`
   `pfAttribList.push_back(true);`
   as pixel format attributes should allow for such binding, however the system doesn't provide a suitable pixel format for such a pbuffer, at least not on Geforce FX graphics boards.
   Therefore, if you need to compute mipmaps for the image in a pbuffer, you have to `glCopyTexSubImage2D()` from the pbuffer to a texture object, which can result in a significant performance drop.

3.8 Implementation

3.8.1 The GPU-based algorithm

During the registration process, we successively apply new transformations to one of the images, leaving the other one unchanged, until a best match is found. I will term the images *moving image* and *fixed image*, respectively.
The basic GPU-based algorithm for comparing two images works in the following way:

1. Store the fixed image on the GPU using a texture object with four color channels (ie. RGBA texture). Since we are registering grayscale images, only the red channel is actually required for the image data, allowing the other three channels to be used otherwise.
2. If any values that go into the final similarity measure formula can be pre-computed from the fixed image alone, as is the case for gradients or the mean intensity, then:

- load the corresponding fragment shader and set its parameters (these contain the fixed texture object and any other necessary values),
- render the image to a pbuffer,
- copy the result back into the texture object.

Note that this step has to be performed only once for the whole registration process so all operations that are independent of the moving image should be performed here.
Pbuffers are off-screen render buffers with special properties that the regular frame-buffer does not have and are explained in detail in 3.7.5.

3. Store the moving image as a RGBA texture object. The image data can either be uploaded to the GPU from the system RAM or can be directly generated on the video card. The latter will usually be the case when the volume rendering methods for the DRR generation are also implemented in hardware. Again, only the red channel initially contains image data.

4. To evaluate the similarity measure:

   - load the fragment shader that will assess the desired similarity measure in each pixel location,
   - render to the frame-buffer or a pbuffer,
   - from the resulting image, which i will term *similarity image*, compute the actual measure, which is a single value.

   While all other steps perfectly fit the architecture and design of the rendering pipeline (ie. feed it geometry and textures and obtain a two-dimensional array of RGBA pixels), the last step is of different nature, as summing up the pixel values stored in a texture is not a typical graphics application. I describe two solutions to this issue in 3.7.4.

3.8.2 Measures Using only Image Intensities

Measures like SAD and SSD, which do not require any data except the original image intensities at the respective location, are very easily implemented as a fragment shader.
The following fragment shader computes the squared difference between the intensities at the current pixel location, or texture coordinate, in the fixed and moving image. The code and the subsequent explanation is intended as a very short introduction to the Cg shading language, illustrating the most common constructs and data types.

```
half4 main(half2 texcoord:  TEXCOORD0),
           uniform sampler2D image1,
           uniform sampler2D image2):  COLOR
{
           half val1 = h4tex2D(image1, texcoord).r;
           half val2 = h4tex2D(image2, texcoord).r;
           return ((val1 - val2) * (val1 - val2)).xxxx;
}
```

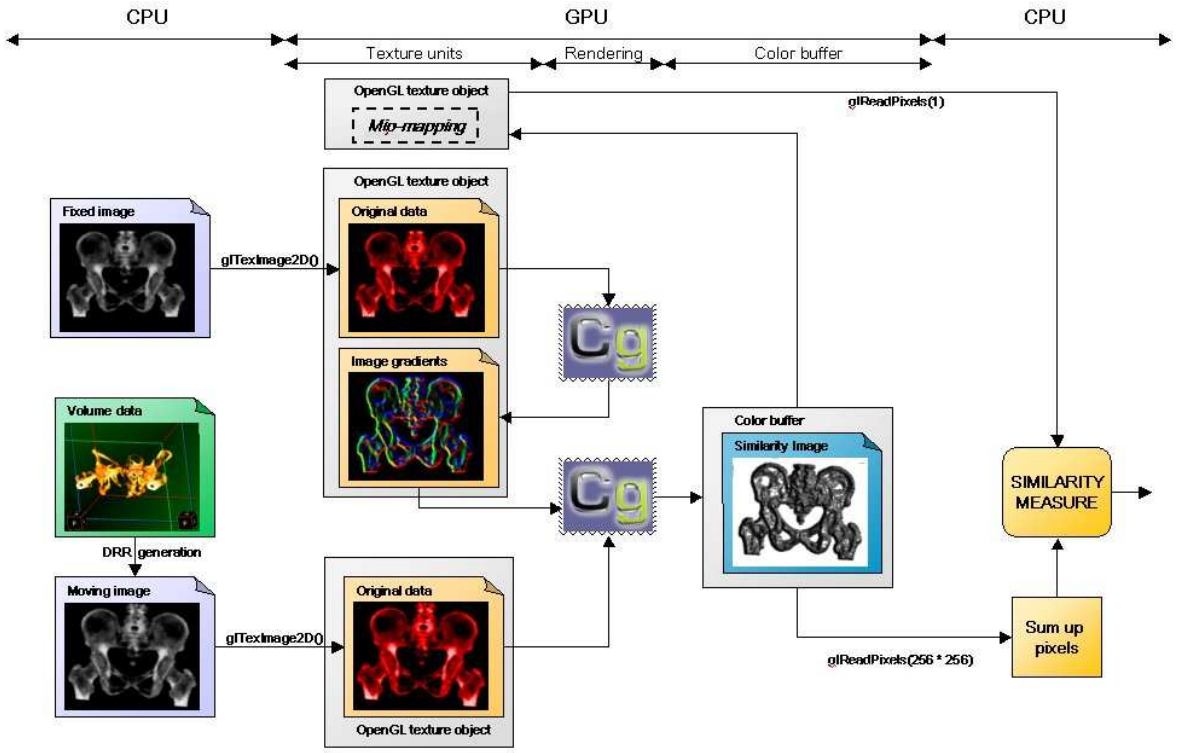


Figure 3.7: The GPU-based algorithm for 2D/3D registration (including CPU-based DRR generation) with a gradient-based similarity measure.

The parameter `texcoord` changes for each drawn fragment and is supplied by the rendering GPU's pipeline. The `half` data type represents a half-precision (ie. 16-bit) floating-point value, while the suffix `2` indicates that we are dealing with a two-component vector. The texture parameters `image1` and `image2` must be set by the user and remain the same for each shaded fragment, ie. they are `uniform` parameters. The result will be written to the `COLOR`-buffer, which stores four values for each pixel, one for each color channel. `h4tex2D(...)` returns a `half4` value, ie. a vector with four `half` components, containing the RGBA intensities in the given texture object and at the specified coordinate. The components in a `half4` can be individually addressed with `.r`, `.g`, `.b` and `.a` respectively (or `.x`, `.y`, `.z` and `.w`). As we are working with grayscale images, all the image information is contained in the red channel. After thus reading the grayscale intensities from the texture objects, we compute the squared sum, which is a single value. Now, since our `main` function's return type is `half4` (as imposed by the RGBA color-buffer), we can duplicate, or *smear*, the computed difference into the G, B and A channels at no computational cost.
To further speed up these very simple measures, one could also use all four color channels to store the image intensities, ie. the first gray-scale image value goes into the first pixel's red channel, the second gray-scale value into the green channel, the third goes into the blue channel, the fourth into the alpha channel, the fifth again into the red channel of the second pixel and so on, which results in a texture only one quarter the size of the original image. Since GPUs are designed to work with 4-component vectors, the fragment shader would compute the 4 squared differences just as fast as the one difference in the example above, which would finally result in a significant performance boost. However, this approach cannot be used for

any of the other measures and with SAD/SSD being already very fast, I have not examined this acceleration method any further.

The Correlation Coefficient (see 2.3) also operates on the original image data, but is nevertheless different from the measures described above as it requires the images' mean intensity values. I described two approaches to image averaging in section 3.7.4.
Pre-computing the fixed image's mean is best done using the CPU-based approach, as it is more accurate. As this is performed only once for the whole registration process, the lower speed is also not an issue.
If we were to use the first NCC formula, we would have to repeat this process for each DRR in a separate pass, before computing the Correlation Coefficient. However, using formula 2.4 we can assess the similarity in a single rendering pass, storing $I_1(x,y) \cdot I_2(x,y)$, $I_1(x,y)^2$, $I_2(x,y)^2$ and $I_2(x,y)$ in the RGBA color channels. In order to take advantage of the GPU's vector processing capability, within the fragment shader we can initialize two vectors containing $[I_1(x,y), I_1(x,y), I_2(x,y), 1]$ and $[I_2(x,y), I_1(x,y), I_2(x,y), I_2(x,y)]$. By multiplying these, we can compute all the four terms with a single operation. This also means that pre-computing the term which depends only on the fixed image ($\sum I_1(x,y)^2 - n \cdot \overline{I_1}^2$) will not yield any performance improvement as both the product and the summation (ie. mip-mapping of the respective color channel) will be carried out anyway.

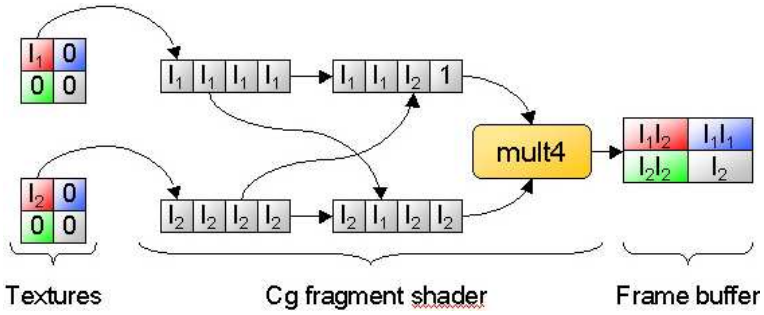


Figure 3.8: Normalized Cross Correlation on the GPU

After obtaining the average intensity of each channel, we can assemble the NCC as follows (where $\overline{I_1}$ is the pre-computed fixed image mean intensity).

$$NCC = \frac{\overline{R} - \overline{I_1} \cdot \overline{A}}{\sqrt{\overline{G} - \overline{I_1}^2} \cdot \sqrt{\overline{B} - \overline{A}^2}} \tag{3.1}$$

3.8.3 Measures Using Spatial Information

Although it regards only the original image intensities, like SAD and SSD, the Pattern Intensity measure (see 2.7) operates on a whole neighborhood rather than an individual fragment. Because the neighborhood is not square (see 2.2.1), running through it by means of two nested for-loops would force us to check each time whether the respective coordinate is inside the region of interest or not. Hence, it is the more efficient to address the neighboring pixels

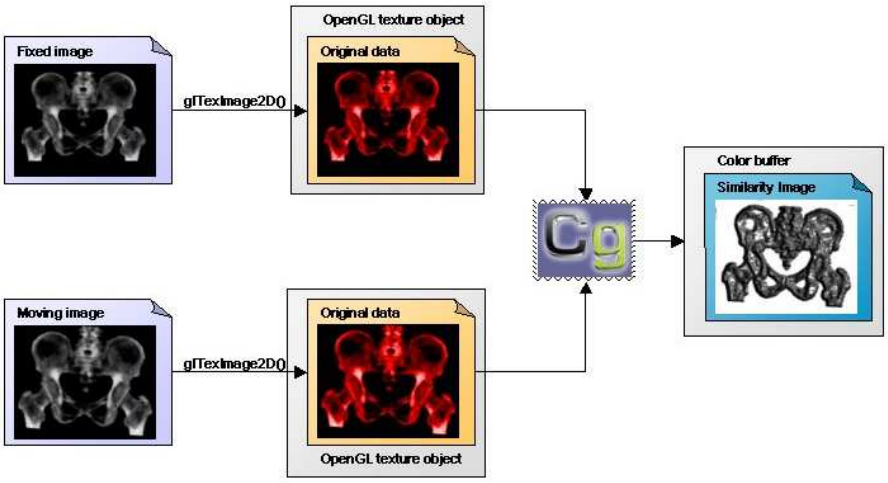


Figure 3.9: Measures Using only Image Intensities

individually, without using any loop constructs, which would be unrolled by the Cg compiler anyway.
As described in section 3.7.1, we can pre-compute the texture coordinates of eight neighboring pixels in the vertex shading unit, thus reducing the complexity of our fragment shader. By writing explicit code for each pixel, there's also no need to check whether the respective texture coordinate has already been computed in the vertex stage or not.

Gradient-based methods consist of two main steps - computing the gradient of the original image and then assess the similarity between the two gradient images. Obviously, it is enough if we compute the fixed image's gradient only once and use that information for the whole registration process. In the case of the DRR however, the gradient must be computed again with each new pose.
As gradients can also have negative values, two color channels will be needed for each pixel, as described in 3.7.2. With all the three gradient-based measures requiring both the vertical and horizontal image gradients, all the four color channels available in a RGBA texture are necessary to store all the data.
As the presented gradient-based measures only examine the gradient value at the respective pixel position, it is possible to pack the DRR gradient computation and the actual measure into a single fragment shader that takes the fixed image's gradient and the original DRR as parameters. This approach is faster than computing the moving image's gradient separately, as it eliminates the overhead of a second rendering pass, with the necessary OpenGL context switch and copying of data from the color buffer to a texture object.

After pre-computing the fixed image's gradient and its mean intensity, the Gradient Correlation Coefficient can be obtained by using a fragment shader that computes both the gradient value in the DRR at the respective position, and the other three terms making up the NCC formula. However, the GC is obtained as the average of two NCC's, namely between the horizontal gradients and the vertical ones. Thus, GC cannot be computed in a single rendering pass, because there are not enough output locations available. In a second pass, one can use

a modified fragment shader that will compute the vertical gradient instead of the horizontal one and correlate it with the corresponding fixed gradient values.
The most modern GPUs (like the NVIDIA Geforce 6) allow one to write to multiple color buffers at the same time. This feature is known as MRT, or multiple render targets. This means that by using two color buffers we can complete both vertical and horizontal gradient correlation computations in a single rendering pass. But due to the fact that the main two time-consuming factors, ie. the fragment shading operations and the similarity image averaging, are not reduced by using MRT, in this case both approaches resulted in the same execution speed. Hence, in order to maintain compatibility with older hardware, like the Geforce FX series, I have chosen not to use multiple render targets in my implementation.

Unlike Gradient Correlation, the Gradient Difference and Gradient Proximity measures can be easily computed in a single rendering pass. Both measures are obtained as a single sum over all pixels, so only one color channel will contain useful information at the end of the computation.

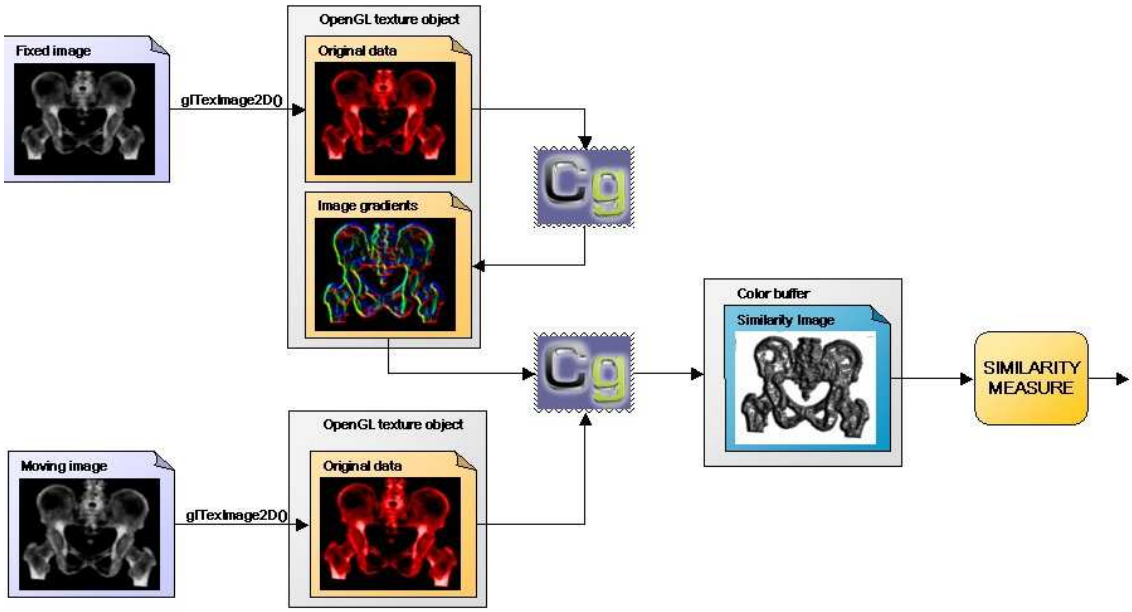


Figure 3.10: Gradient Difference and Gradient Proximity

The Sum of Local Normalized Correlation can be computed in two ways: for overlapping sub-images or for non-overlapping subimages. The second case would imply that we are computing the measure at some pixel locations (usually the center or a corner of each sub-image), but not at others. However, as the Shading Standard 2.0 supported by NVIDIA Geforce FX boards does not allow fragment programs to contain conditional return statements, one cannot decide within the fragment shading step whether the LNC will be computed or not. Therefore, the only straightforward possibility is to compute the local correlation coefficient for sub-images centered around each pixel, which is obviously quite expensive, resulting in a slow measure. A way of computing the correlation coefficients for non-overlapping images would be to store a mask inside the depth-, alpha- or stencil-buffer in such a way that only

one pixel out of each sub-image would pass the respective test and thus be drawn.
The newer Shading Standard 3.0 which is currently supported only by the NVIDIA Geforce 6 GPUs does provide much better support for conditional statements, so LNC can be implemented to examine non-overlapping sub-images as well. However, conditional statements are still quite expensive, so the performance gain I obtained by using this approach was by far not as large as expected, as the computation was just about 30% faster.
Fortunately, a considerable part of the final measure can be pre-computed and stored in the green, blue and alpha channels when one sets the fixed image. In this way, we can pre-compute the mean of the sub-images in the fixed image and also $\sqrt{\sum(I(x,y)-\overline{I})^2}$. For the Variance-Weighted Sum of Local Normalized Correlation, can also pre-compute the sub-images' variances and store it in the remaining color channel.

3.8.4 Information Theoretic Measures

Although this type of similarity measures, especially MI, are very popular in real life, I have found no way to implement them on the GPU.
In order to generate the required joint histogram, one needs access to a two-dimensional array which holds the number of pairs of intensities in two images at the same pixel position. This means that, whenever a fragment shader examines the texture coordinate (a, b) in the two input texture objects, it would have to increment the respective value at the coordinates $(I_1(a,b), I_2(a,b))$ inside the joint histogram, represented by another 2D image like a texture object or the color buffer. But, since a shading unit's output cannot be directed to an arbitrary location, being tied to a certain pixel in the color buffer, the generation of joint histograms is impossible on GPUs up to NVIDIA's Geforce FX series.
However, the Geforce 6 series GPU, which supports the Shading Standard 3.0 and provides access to textures inside the vertex shading unit, which is situated before the fragment unit inside the rendering pipeline, might offer a possibility to overcome this problem. Using vertex textures, one can generate a joint histogram in the vertex shading unit, by placing vertices in a spot indicated by the texture values. Still, this is expected not to be very fast due to the rather slow texture access within the vertex shading unit.
Due to the fact that I received a Geforce 6 GPU at a very late point of time in the course of this thesis, and a number of issues are to be solved in order to complete the implementation of the basic algorithm just described, I have not implemented MI on the GPU.

4 CPU-based acceleration

4.1 Motivation

Apart from ever-increasing clock speeds and more efficient micro-architectures, modern microprocessors also offer "single instruction, multiple data"(or SIMD) instruction sets, that allow a CPU to carry out a specific operation on multiple data elements at once. This approach can yield a significant performance increase for applications that handle large quantities of data in parallel, as is the case of multimedia applications like image and video processing[12].

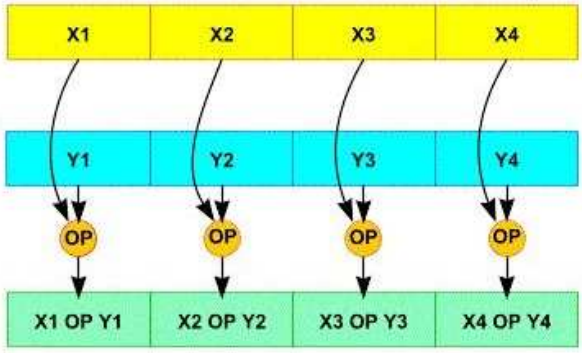


Figure 4.1: SIMD instruction working on four data elements in parallel

Intel has introduced two such instruction sets. The first one was MMX, supported by the Pentium MMX (and later) processors, followed by SSE (Streaming SIMD Extensions), supported by the Pentium III CPU. SSE has been extended with the introduction of the Pentium 4, giving birth to SSE2. The most recent Pentium 4 models also offer SSE3 functionality. Other CPU manufacturers also offer their own SIMD instruction sets, while trying to keep them compatible with Intel's, as is the case of AMD's 3DNow! technology.

These instruction sets also represent an attractive alternative for speeding up our registration process, and additionally eliminate the need for a programmable graphics board and knowledge of graphics API's. However, while compilers can automatically perform vectorization and generate SSE code, in order to obtain the best result, one still has to take up the task of manually writing SSE programs.

4.2 The SSE/SSE2 Instruction Set

Intel's first SIMD implementation, MMX, suffered from two main problems. It did not physically implement the eight 64-bit MMX registers, but reused the existing FPU registers, making

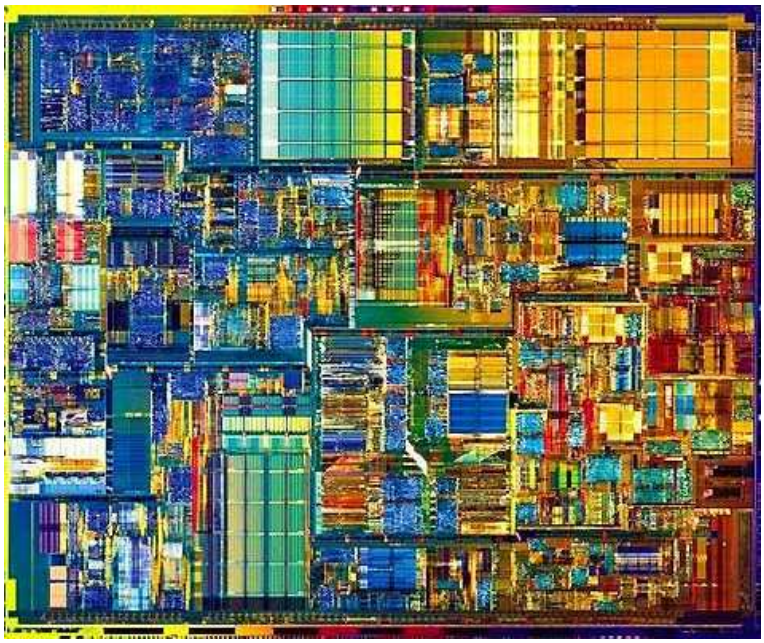


Figure 4.2: Intel Pentium 4 core

the CPU unable to work with floating-point and SIMD data at the same time, and it only supported integer operations.
SSE solves these problems by adding eight new 128-bit registers (XMM0 through XMM7), each of them packing four single-precision floating-point values. The later SSE2 version then eliminated this data-type restriction, adding support for 64-bit double-precision floating-point numbers and integer operands with lengths of 32, 16 and 8 bit. Thus, SSE2 allows SIMD operations to be carried out on anything from 2 double-precision floats to 16 half-word integers, rendering the MMX instruction set obsolete. However, mainly for the sake of backward compatibility, MMX is still supported by the latest Intel CPU's.
As image data is usually stored as integer intensity values, I have made extensive use of SSE2 instructions.

As already mentioned, the SIMD code automatically generated by the compiler is not optimal, so one has to rewrite any existing algorithms using explicit SSE instructions in order to take full advantage of the processor's capabilities.
The Intel C++ Compiler software and newer additions to Microsoft's Visual Studio come with C header files that define so-called SSE *intrinsics*. An intrinsic is a function known by the compiler that directly maps to a sequence of one or more assembly language instructions. Intrinsics greatly simplify SSE programming, as they not only provide a C/C++ language interface to the assembly instructions, but also leave a number of tasks to the compiler. In doing so, the programmer does not have to deal with register names, register allocations and memory locations of data, but can use C/C++ variables of a certain 128-bit data type just as in the case of classic programming [23]. However, SSE programming in effect still comes down to assembly programming, so developers can only use the functions directly implemented in hardware (apart from a few macro-like intrinsics) like register loading, storing, shifting and

the basic logical and arithmetical operations.

One issue arises from memory-alignment. Data alignment means that a data element, as a word or double-word, is stored in the system's RAM at an address that is a multiple of the respective data type's size, ie. an aligned word is stored at an even address, an aligned double-word at an address divisible by four and so on. SSE instructions, which work with 128-bit blocks, are designed to access data from 16-byte aligned memory addresses. Instructions for unaligned memory access also exist, but they are much slower than the aligned access. Instructions expecting aligned data will cause a general protection fault when trying to access unaligned data [13], so the programmer must take care to use the right instruction or, in order to obtain the optimum performance, make sure that all the data, that SSE operates on, is 16-byte aligned.
The C++ compiler can be forced to allocate static data at a 16-byte aligned address by putting the storage-class specifier `__declspec(align(16))` in front of a variable's declaration. In order to dynamically allocate and free aligned memory blocks, one can use the `void* _mm_malloc (int size, int align)` and `void _mm_free (void *p)` functions provided by the Intel libraries.

4.3 Implementation Prerequisites

4.3.1 Running through Pixel Neighborhoods

SSE is intended to manipulate data stored in 16-byte blocks, that are expected to be stored in memory at 16-byte aligned addresses. While this concept works very well if we want to apply exactly the same operation to all data elements in the loaded block (eg. add a constant value), running through a pixel's neighborhood is not so straightforward. This is mainly due to two reasons: one cannot randomly access and operate on data elements within a block (ie. add the 1st element in register XMM0 to element 2 in XMM1) and the neighborhood of the pixel stored at the end (or beginning) of a data blocks spans over two SSE registers.
This problem can be solved by shifting SSE register contents. In this way, one can successively align the data elements that are needed for a certain operation. The values at the beginning and end of the data block require additional operations in order to account for the values stored in the previous or next data block. Depending on the specific algorithm, one can either choose to run through multiple neighborhoods of adjacent pixels at the same time, or to speed up the process of regarding a single image area.

Note that register shifting operations are only available for integer data types (of any length). If one is using 32-bit or 64-bit floating-point values, the same effect can be achieved by using the `MM_SHUFFLE_PS)` instruction. Rather than shifting, this instruction allows the programmer to shuffle individual floating-point values coming from one or two SSE register in any desired order.

4.3.2 Data Type Conversions

Depending on the data type one intends to use, not all basic arithmetical operations might be supported by SSE. 8-bit integers can only be summed up and substracted, 16-bit and 32-bit

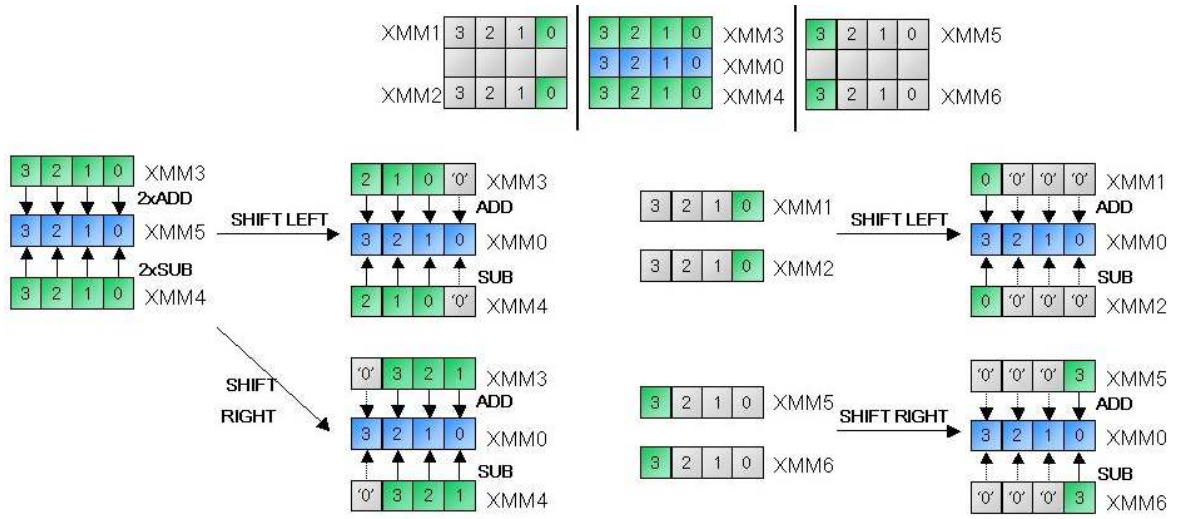


Figure 4.3: Computing the vertical gradient of a two-dimensional image using SSE, where image intensities are stored as 32-bit integers. The gradients at all pixel locations stored in XMM0 are computed simultaneously.

integers can also be multiplied and floating-point values can also be divided. Assuming the case that one is dealing with 16-bit integer data and wants to divide these numbers by those stored in another SSE register, the eight 16-bit integer values must first be converted to eight 32-bit floating-point values (which need two SSE registers to fit in), carry out the division, and then convert the floating-point values back to short integers. While this is achieved very easily in traditional C/C++ programming by means of the cast operators, on the SSE assembly level the programmer must implement this process himself.
In order to get from 16-bit to 32-bit integers, one can use the `MM_UNPACKLO_EPI16` and `MM_UNPACKHI_EPI16` instructions, which interleave the lower or higher four 16-bit elements in one SSE register with those in a second register, which for this purpose will be initialized to zero. One can thus easily obtain two SSE registers holding eight 32-bit integers, which contain zeroes in the high-order word and our actual 16-bit integer in the low-order word. However, special care must be taken when one is dealing with signed 16-bit values. In that case unpacking, or interleaving, of negative values with zeroes, as described above, will cause the sign bit, which is the highest-order bit in the word block, to end up in the middle of the double-word block, thus yielding an unsigned integer with an absolute value twice as large as the original one. To prevent this problem, one has to interleave the two registers holding 16-bit values in such a way as to obtain 32-bit integers that hold the actual value in the high word and the zeroes in the low word. An arithmetic shift (as opposed to the logical shift, this will fill up the register with sign bits instead of zeroes) to the right by 16 bits for each double-word in the SSE register then gives the expected result. The double-word integers are then easily converted to floating-points using `MM_CVTEPI32_PS`, divided and converted back to integers again. Using `MM_PACKS_EPI32`, the eight 32-bit integers stored in two registers can then be packed into eight 16-bit integers within a single register. This instruction preserves the values' sign and also clamps all numbers to the maximum value that can be represented by the shorter 16-bit data type.

Working on 8-bit data elements implies an additional (un)packing operation to convert

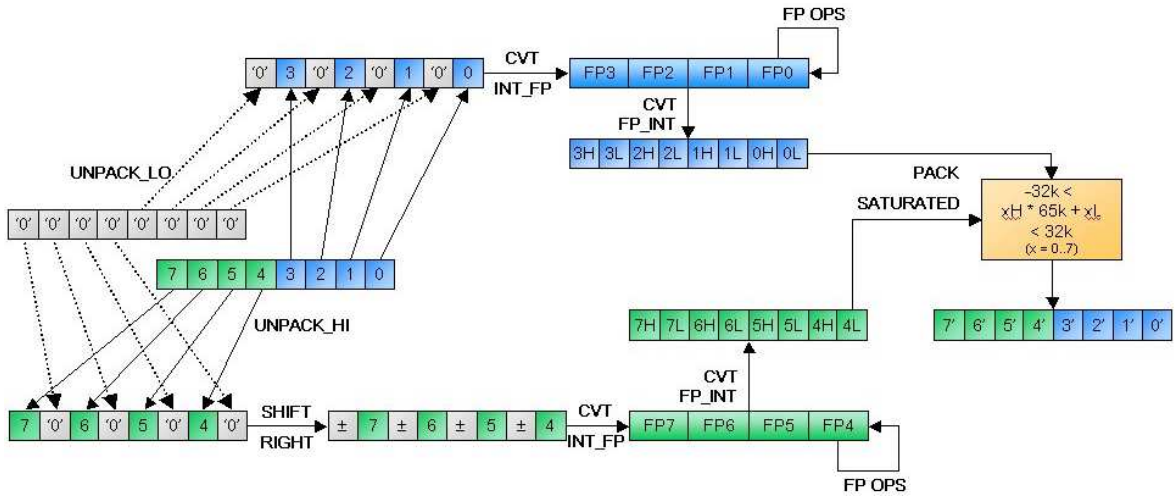


Figure 4.4: Converting from 16-bit integers to 32-bit floating-point and back. The upper (blue) path shows the case of unsigned integers, the lower one (green) shows the case of signed integers.

between sixteen 8-bit values in one SSE register and sixteen 16-bit values in two registers. The rest of the process remains unchanged.

4.3.3 Computing the Similarity Measure

While the GPU implementation forces one to first compute a similarity image and sum up the individual pixel values in a separate step at the end, the CPU-based SIMD implementation does not impose such a restriction. Therefore, it is the most convenient to sum up the per-pixel similarity values, once they are computed, in an SSE register containing four floating-point values.
In the case where the similarity measure requires floating-point operations in order to be computed (as Gradient Difference or any other measure containing a division, which can only be performed on floats), this also removes the need for converting the similarity values back to the data type of our actual images, which will usually be 8-bit or 16-bit integers.

4.4 Implementation

4.4.1 Measures Using only Image Intensities

After solving the problem of data conversion, the Sum of Squared Differences is very easily implemented:

```
__m128i zero = _mm_setzero_si128(); //an integer register full of zeroes
__m128 sum = _mm_setzero_ps(); //a floating-point register where we
//sum up the squared differences
```

```
//Run through the image data:
for(int i=0; i<packets128bit; i++)
{
        //Load an 8-pixel block from the images (16-bit integers)
        __m128i fixed16 = _mm_load_si128(ptrfixed);
        __m128i moving16 = _mm_load_si128(ptrmoving);

        //Convert the first four 16-bit integers to 32-bit integers
        __m128i fixed32 = _mm_unpacklo_epi16(fixed16,zero);
        __m128i moving32 =_mm_unpacklo_epi16(moving16,zero);

        //Compute the diference and convert to floating-point
        __m128i diff32 = _mm_sub_epi32(fixed32,moving32);
        __m128 result32f = _mm_cvtepi32_ps(diff32);

        //Compute the squared difference and add to the sum
        result32f = _mm_mul_ps(result32f,result32f);
        sum = _mm_add_ps(sum,result32f);

        //Convert the other 16-bit integers to 32 bit integers
        fixed32 = _mm_unpackhi_epi16(fixed16,zero);
        moving32 =_mm_unpackhi_epi16(moving16,zero);
        //Repeat the above steps
        [...]

        //Increment the pointers (they are of type __m128i*, so incrementing them
        //jumps to the next 16-byte aligned address)
        ptrfixed++;
        ptrmoving++;
}

//Create a 16-byte aligned array in which to store the 'sum'
__declspec(align(16)) float ssd[4];
//Store the SSE register to the array
_mm_store_ps(ssd,sum);
//Sum up the four floats and return the SSD
return (ssd[0]+ssd[1]+ssd[2]+ssd[3])/(ImageWidth*ImageHeight);
```

The Sum od Absolute Differences, although it might intuitively seem very similar to SSD, adds the issue of computing a number's absolute value, while there is no SSE instruction for that. One solution would be to first compute the squared difference as above, and then return its square root. However, this method will obviously be very slow, as floating-point multiplication, and especially the square root, are complex operations requiring many clock cycles.
A much better solution is the following:

1. Compute the difference, using 32-bit integers.

2. Use `_mm_cmplt_epi32(dif,zero)` to determine which of the four integers in register 'dif' is negative. The function's result is a register that contains four 32-bit blocks of zeroes and ones. A sequence of 32 zeroes indicates that the respective integer did not pass the comparison (ie. it was positive or zero), while a negative integer value generates a sequence of 32 ones. Or, in C/C++ notation: `result[i] = (dif[i] < zero[i]) ? 0xffff : 0x0000`, where i ranges from 0 to 3.

3. Compute the bitwise XOR between the result of the comparison and the register containing the difference. This will invert all the bits belonging to negative integers (a XOR 1 = ¬a), while leaving the bits of positive integers unmodified (a XOR 0 = a).

4. A bitwise inversion of negatives results in the numbers' absolute values, minus one. In order to obtain the correct result, we can now compute the XOR between the result of our previous comparison and a register containing four integer values of 1 and add this to our result. However, especially in the case of 16-bit images, we can actually skip this step as the resulting inaccuracy is negligible.

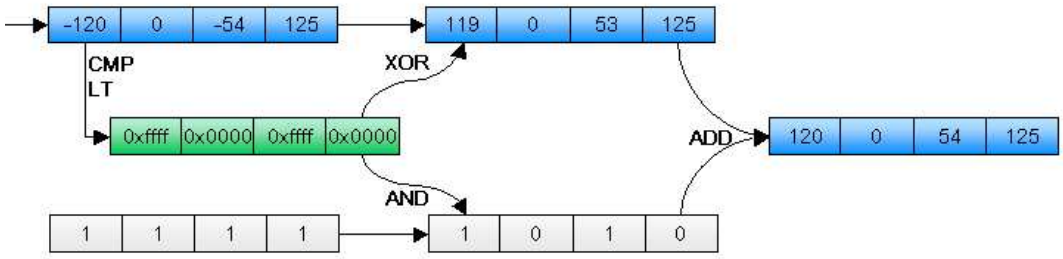


Figure 4.5: Computing the absolute values of four 32-bit signed integers.

As with the GPU implementation, implementing the Correlation Coefficient using the second formula (2.4), which requires us to run through the images only once, is much faster. Also, having pre-computed $\sqrt{\sum I(x,y)^2 - n\overline{I}}$, the resulting SSE program will be very simple.

4.4.2 Measures Using Spatial Information

This class of measures require us to deal with pixel neighborhoods, process which I have described in section 4.3.1. As stated, there are two ways of speeding up the computation by means of SIMD instructions: examine multiple neighborhoods or examine multiple values from the same neighborhood.

Pattern Intensity (see 2.7) contains multiplications and divisions, so the computations have to be carried out on floating-point values, ie. the SSE registers will contain only four pixel intensities. As the regarded neighborhood, as shown in 2.2.1 contains rows with five values, we will thus require two SSE registers for the second and third row, while one register is enough for the first and fourth row. However, this implies that, for the second and third row, we will carry out SIMD instructions on registers holding a scalar value. Hence, I decided

to extend the PI neighborhood from a circle to a square. This will supposedly enhance the measure's stability, as more spatial information is taken into account, while only the fourth row adds computational cost, as it now also requires two SSE registers instead of one. I have noticed that the performance drop is minimal.

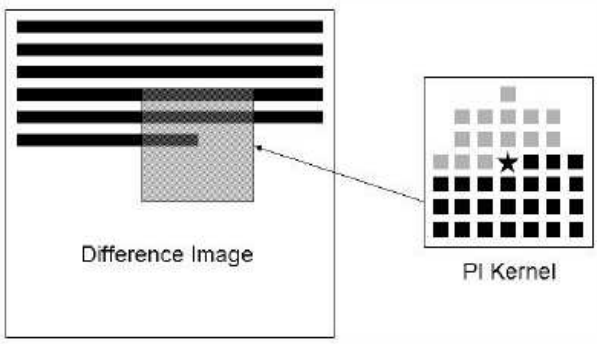


Figure 4.6: Kernel for computation of neighborhood differences with SSE

As the size of our neighborhood is quite large, with a pixel row spanning over up to three SSE blocks, in this case I have decided to regard a single neighborhood at a time, rather than multiple ones. The Pattern Intensity algorithm works as follows:

1. Convert image data from integer to floating-point (see 4.3.2). In total, we have 25 pixels in our neighborhood, so we will need 7 SSE registers for our initial values.

2. In three of four cases, the four values in the first neighborhood row will be spread over two SSE registers. Use a pre-initialized bit-mask to AND away the unneeded values and then OR the two blocks bit-wise to pack the four values in a single register. This means that the order, in which the pixel intensities are stored in the SSE register, does not match the order in the image anymore, but this makes no difference to our algorithm.

3. The SSE blocks situated directly 'below' the block containing our neighborhood center directly go into the final computation, as all four pixels belong to our region of interest. The other three pixels are situated in the previous, next, or both, SSE block(s), depending on the position of our center value. If they are scattered across two registers, use the same method as above to pack them into a single register.

4. Now we have four registers containing four neighboring pixel intensities and three registers containing three intensities and one zero. Using shuffling, create a register that contains four copies of the current center value. Using a bit-mask to AND away one value, create another register containing three copies of the center value and one zero. The zero must be at the same position as the ones in the other registers.

5. Compute $\sigma^2/(\sigma^2 + (I_{diff}(x,y) - I_{diff}(v,w))^2)$.

6. Due to three pairs of zeroes and the fact that there is a copy of our center value in the register storing our neighborhood's first row, we have four unwanted differences with values of zero, which thus add 4 to our final PI measure. In a final step, subtract this

constant value from our previous result. As in the end we will sum up the four floating-point values inside the SSE register into a single number, it makes no difference from which of the four elements we will subtract four.

Note that at the beginning and at the end of an image row, not the whole neighborhood is available. The above steps must be adapted to deal with those situations accordingly.

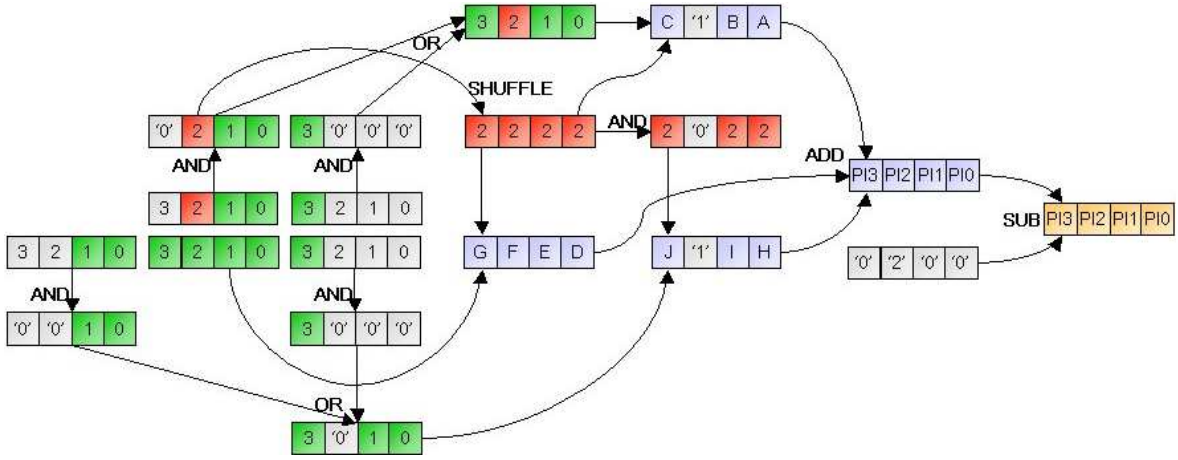


Figure 4.7: Pattern Intensity with SSE. The neighborhood's center is colored in red, the pixels belonging to the neighborhood in green and intermediate results in blue. For simplicity, the figure only shows the first and second row of the neighborhood (therefore, a constant value of 2 is subtracted instead of 4). Rows three and four are handled exactly like the second one. Data conversion from integer to floating-point has also been left away.

The Sobel filter used for gradient computation only requires additions and subtractions, so it can be applied directly to the integer image intensities, without any data type conversion. However, we must not forget that the gradient values are signed (as opposed to the unsigned image data) and can get up to four times larger than the initial values.
There are three ways of providing the required three bits: store the gradient image using the next-larger data type (ie. 32-bit gradients for 16-bit images, 16-bit gradients for 8-bit images), divide the image data by eight before computing the gradient or carry out the gradient computation using the next-larger data type and divide the result by eight.
16-bit images offer a very good intensity resolution, so I consider that, for our application, dividing the values up front by eight (ie. losing three bits of accuracy) will not have a noticeable impact on our similarity measure's performance. Actually, medical image data is usually stored using a 16-bit format, but only 12 of these bits contain intensity information, as X-ray intensities range from 0 to 4095, so we can completely skip this division in our case. The advantage of this approach is that it does not impose the overhead needed for conversion from 16-bit to 32-bit integers, which would also result in our SIMD operations computing only four gradients at a time instead of eight.
8-bit images, on the other hand, provide a rather low intensity resolution, so carrying out the division before computing the gradient will leave us with five bits of actual data and thus a very inaccurate result. Therefore, we load 8-bit numbers, convert the data to 16-bit and com-

pute eight gradients at a time (instead of sixteen, as we could do for 8-bit pixel intensities). If memory considerations are not a critical issue, it is best to store the image gradient as signed 16-bit integers. Otherwise, we would have to convert the results of our gradient computation back to 8-bit and then, when computing the actual similarity measure, again to 16-bit, as all our gradient-based measures operate on floating-point values.
When converting the 16-bit gradients to floating-point in order to assess the similarity measure, one must again not forget that he is dealing with signed values, ie. follow the 'green path' in figure 4.4.

As on the GPU, it is faster to compute the moving gradient and the similarity measure at the same time. Although the number of arithmetic operations is the same, whether the gradients are separately pre-computed or not, the speed gain is achieved by saving one transfer of data from the main RAM into the processor's cache and back.

The Sum of Local Normalized Correlation is most easily implemented when regarding 8x8 sub-images that do not overlap. When the image data is stored as 16-bit integers, eight pixels fit into each SSE register, so we can read one sub-image row at a time.
With 8-bit images, the sixteen intensity values can be split into two SSE registers and converted to short integers. After this, the same operations as in the aforementioned case can be executed, so that two neighboring 8x8 sub-images can be addressed with each register loading step.
If one chooses to compute the SLNC for overlapping images, he will face the problem of sub-images spanning over multiple SSE packets. As described in 4.3.1, handling this situation involves a significant overhead for re-arranging the data, which will obviously be detrimental to performance and also imply a much higher implementation effort. I have implemented both SLNC versions, but used only the second one for validation, in order to be able to compare the results to those given by the GPU-based approach.

4.4.3 Information Theoretic Measures

The SSE instruction set does not support indirect operand addressing and neither can SSE register be used for memory access in any other way. This means that image histogram generation cannot be implemented with SSE, as the a pixel's intensity value would have to serve as the memory address of the histogram bin it belongs to. Apart from that, the instruction set does not provide a SIMD logarithm function.
Therefore, Mutual Information cannot be implemented with SSE.

5 Validation

5.1 The Application

The presented algorithms were built into an existing application prototype developed by Siemens Corporate Research. The program is performing 3D-3D registration of volumetric data by iteratively aligning orthographic projections along the three coordinate axes. The current setup completes a total of nine optimizer runs, aligning the projections (ie. two-dimensional images) three times for each of the X, Y and Z axes. The employed optimization method is a closest neighbor approach optimizing three parameters (two translations and one rotation), where the step size within the parameter search space and the maximum number of performed evaluations is decreased with each of the three rounds. In this way we can register volume data without having to take a slow approach of running through a very high number of voxels.

The projections are rendered only once for each of the nine optimizer runs, the result being then stored as a RGBA 8-bit 2D texture object and/or as an array of 8-bit unsigned integer values. This 2D image is then rotated and translated iteratively in order to determine the best alignment of the respective projections.
When using the GPU-based similarity measure computation, the image transformation is performed on the GPU by drawing a rotated and translated textured quad into a pbuffer, with the resulting image being then copied back into a texture object and then used as input for the respective fragment shading programs. The similarity image is also rendered to a pbuffer, from where it can be summed up using either automatically generated mipmaps or fully copied to the main RAM for the CPU summation.

For the SSE-based measures, the transformed image is computed on the CPU. When not using SIMD optimizations, it would be possible to perform both the image transformation and the similarity evaluation in a single pass by comparing the values at the original pixel coordinates and at the rotated ones, ie. $f(I_1(x,y), I_2(x\cos(\alpha) - y\sin(\alpha) + t_x, x\sin(\alpha) + y\cos(\alpha) + t_y))$. In this way, we wouldn't have to read the input image twice and write a rotated version, which would surely improve performance in the case of measures that do not evaluate spatial information. However, as SSE can only read blocks of 128 bits at a time, the transformed image must be computed before assessing the similarity measure, because otherwise the blocks of intensity values from the fixed image would correspond to a set of values from the moving image that are not stored one after the other in memory, thus being inaccessible via SSE calls. Although it might be possible to rotate an image using Intel SIMD acceleration, for this application the image transformation was computed without any hardware-based acceleration.
The image can be rotated using either the nearest-neighbor approach or bilinear interpolation,

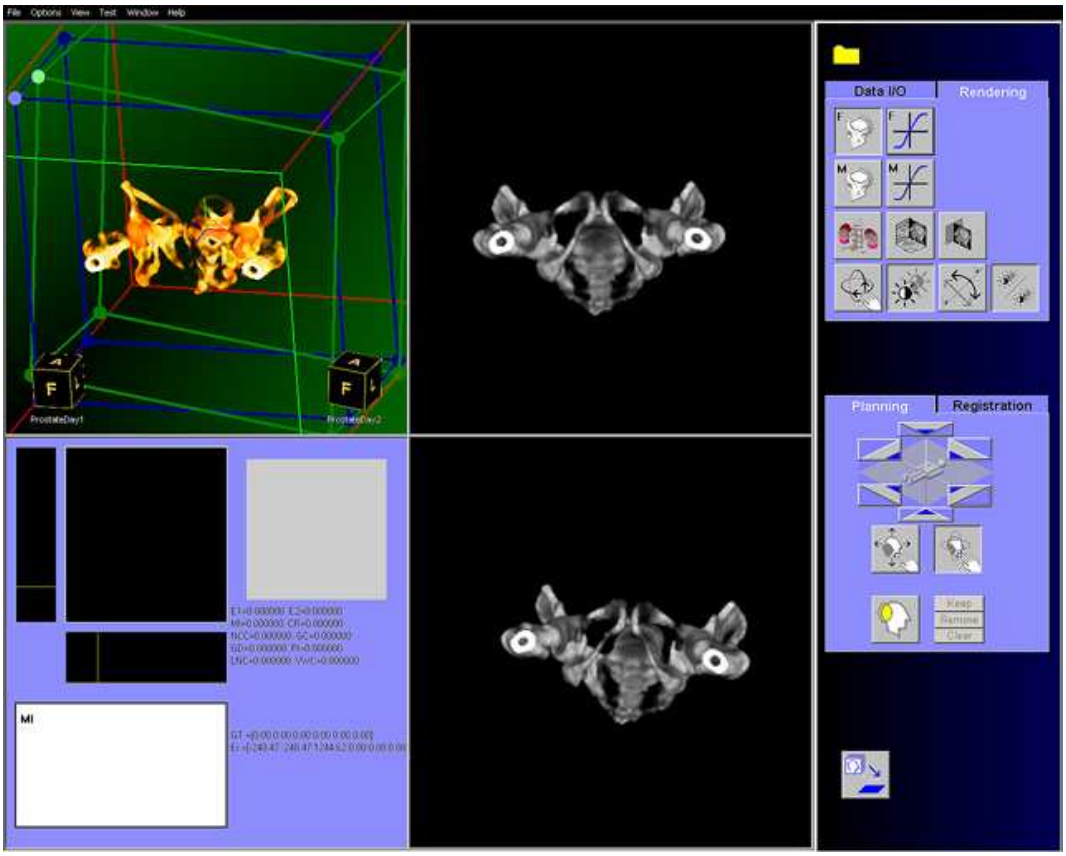


Figure 5.1: The Reg3D application prototype. The upper left view shows a volumetric representation of a CT scan of a human pelvis (the soft tissue has been removed using a transfer function), while the two views on the right show the projections along one of the axes of the fixed data set (top) and the moving data set (bottom). The lower left view display intermediate results during registration.

which is obviously faster but more accurate. For validation purposes, both methods have been evaluated.

5.2 The Validation Process

For validating the similarity measures for the five different implementations, we have registered four pairs of CT volumes - a scan of a pelvis with itself (to determine the best-case performance), two scans of the same pelvis taken at different times (as would be the case for longitudinal studies), two scans of another pelvis that are affected bt image noise and two scans of a Rando head phantom, one of which was acquired using a regular CT and the other using a megavolt cone-beam CT.
Starting from the ground truth pose, the moving volume is then being displaced by predefined random amounts, that are being reused for each experiment to make sure that the various implementations are benchmarked under equal circumstances. The average and the standard deviation of the registration error and the duration of the registration process of one hun-

dred such registration runs for each of the ten similarity measures and five implementations, resulting in five thousand registration runs for each data set (taking approximatively nine to ten hours).
The registration error and running time is also compared to a non-accelerated registration using Mutual Information as a similarity measure for the two-dimensional projections.
The validation was run on a machine with an Intel Pentium4 2.4GHz processor, 1024MB RAM and an NVIDIA Geforce 6800GT AGP8x GPU with the generic drivers (release 66.93). The operating system was Microsoft Windows 2000 SP4.

5.3 Obtaining Ground Truth

As we could not acquire new CT scans for these experiments, we had to use existing data sets, where the actual ground truth pose was not known. As the data sets also do not contain any fiducial markers, we have used the registered pose obtained by using a volumetric, ie. voxel-based, Mutual Information similarity measure, as ground truth. By adjusting the window/level settings in order to isolate only the rigid bony structures, we could thus obtain a very accurate alignment. It should be noted that this registration process took several minutes, as opposed to the other methods taking several seconds.

5.4 Expressing Alignment Errors

In order to assess the alignment error, given the ground truth pose, we can use and quantify an error transformation, ie. the transformation that must be applied to the registered pose in order to obtain the ground truth pose [49].

$$T_{GT} = T_{error} \cdot T_{result} \Rightarrow T_{error} = T_{GT} \cdot T_{result}^{-1} \tag{5.1}$$

If this transformation is expressed as translations and rotations with respect to the three axes, $(t_x, t_y, t_z, r_\alpha, r_\beta, r_\gamma)$, a single error value can be obtained from this representation as RMS sum, $\sqrt{t_x^2 + t_y^2 + t_z^2 + r_\alpha^2 + r_\beta^2 + r_\gamma^2}$. However, from a physical point of view, this is not a correct expression as it sums up values representing millemeters and others representing rotational angles.

Another common possibility to express the alignment error is the Target Reigstration Error, or TRE, which represents the euclidian distance between one or a set of corresponding points in the two volumes. One possibility for choosing these points is to identify anatomical features of interest, eg. a tumor. For our validation process, we chose the corner points of a box having the side's length of 100 and being centered around the volume center. The TRE is then computed as the average euclidian distance between the eight pairs of points of the two boxes being transformed with T_{GT} and T_{result}, respectively.

$$TRE = \frac{1}{8} \sum_{i=1}^{8} d(P_{GT}^i, P_{result}^i) \tag{5.2}$$

The average alignment error, or *bias*, indicates how close the estimated poses are, on average, to the ground truth. Besides this, the likeliness of the registration to yield the same end pose while starting from different initial estimates, is also required for determining the accuracy of a registration method. A good indicator of this property which is known as *precision* is the standard deviation of the TREs. According to [20], both a low bias and a high precision are necessary for a high accuracy.

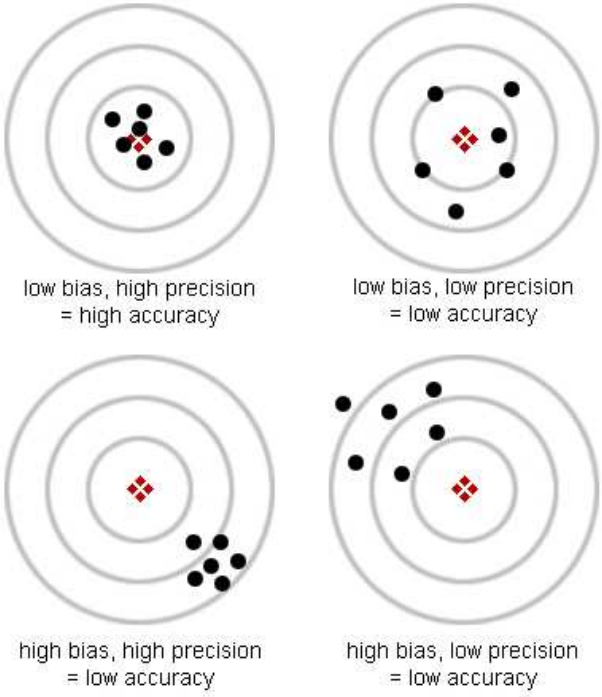


Figure 5.2: The effect of bias and precision on accuracy

5.5 Experiments

Using four pairs of CT data sets, ten similarity measures in five hardware-accelerated implementations have been benchmarked with respect to registration time (in seconds) and target registration error (in millimeters).
In this section, I will list the TRE and registration time of the most accurate three measures for each of the five implementations. Additionally, the TRE and time for registering the volumes using a non-accelerated, Mutual Information similarity measure is also shown for better comparison. The complete results for all the measures can be found in the appendix.

The following table contains the abbreviations used to denote the similarity measures.

SAD	Sum of Absolute Differences
SSD	Sum of Squared Differences
RIU	Ratio Image Uniformity
NCC	Normalized Cross Correlation
GC	Gradient Correlation
GD	Gradient Difference
GP	Gradient Proximity
PI	Pattern Intensity
LNC	Sum of Local Normalized Correlation (on 7x7px overlapping sub-images)
VWC	Variance-Weighted LNC (on 7x7px overlapping sub-images)
MI nn	Mutual Information (no hardware acceleration), nearest neighbor
MI bi	Mutual Information (no hardware acceleration), bilinear interpolation

The following table contains the abbreviations used to denote the various implementations.

GPU, MM	GPU implementation, image averaging with mipmaps
GPU, CPU	GPU implementation, image averaging on the CPU
GPU, Hyb	GPU implementation, using mipmaps to reduce the similarity image from 256x256 pixels to 16x16 (4th mipmap) and averaging the remaining 256 pixels on the CPU
SSE, nn	SSE implementation, image transformation using nearest-neighbor interpolation
SSE, bi	SSE implementation, image transformation using bilinear interpolation

5.5.1 Experiment 1 - Identical CT scans

In this experiment, I used the same data set with identical window/level settings as both moving and fixed volume. The experiment's outcome should give a good indication of the best-possible results when using a certain measure.

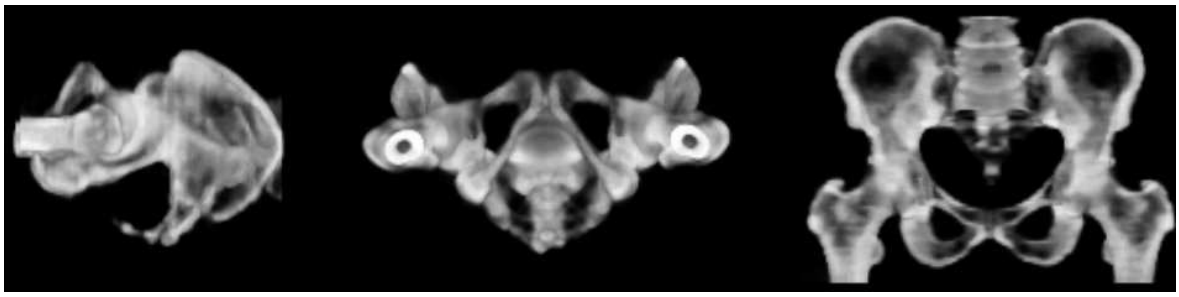


Figure 5.3: Experiment 1 - Identical CT scans

From the following charts we can see very well the impact of the mipmap-based image averaging. Although the computed similarity values at each pixel location are the same in all cases, the repeated rounding errors introduced by the generation of mipmaps with only 8 bit of accuracy per color channel considerably affect the accuracy of the registration. This effect is especially strong for measures where the final formula consists of more than one element, which is not very surprising. Because the intensity values in the different color channels are rounded up to the next higher value representable by 8 bits, the final rounding errors affecting each of the R, G, B and A channels will vary a lot from one channel to another and one image to another. Thus we will observe a very low precision in measures like NCC, GC, and VWC. When using the CPU for averaging the individual RGBA intensities of the computed similarity image, we can see a significant improvement in both TRE and precision. However, the impact on performance is also noticeable, as the slow readback of data from the GPU to the system's main RAM approximatively doubles the registration time. This will be especially noticeable in the case of Gradient Correlation, which, being a two-pass measure, requires twice as many texture downloads.
We can see that the hybrid approach for averaging the images, using both mipmaps and the CPU, yields TREs comparable to those obtained by full CPU averaging, without needing much more time than for the full mipmap-based averaging.

When comparing the two SSE-based implementations, we can again observe a strong improvement in the measures' accuracy if using bilinear interpolation instead of nearest neighbor when transforming the two-dimensional projection of the volumetric image. But we can also observe the significant impact of using bilinear interpolation on the registration's running time.

When compared to Mutual Information, we can see that many measures offer more accurate results, (except for the two low-quality implementations, ie. GPU with complete mipmap-based averaging and SSE with neares-neighbor interpolation), in as much as 20% of the time.

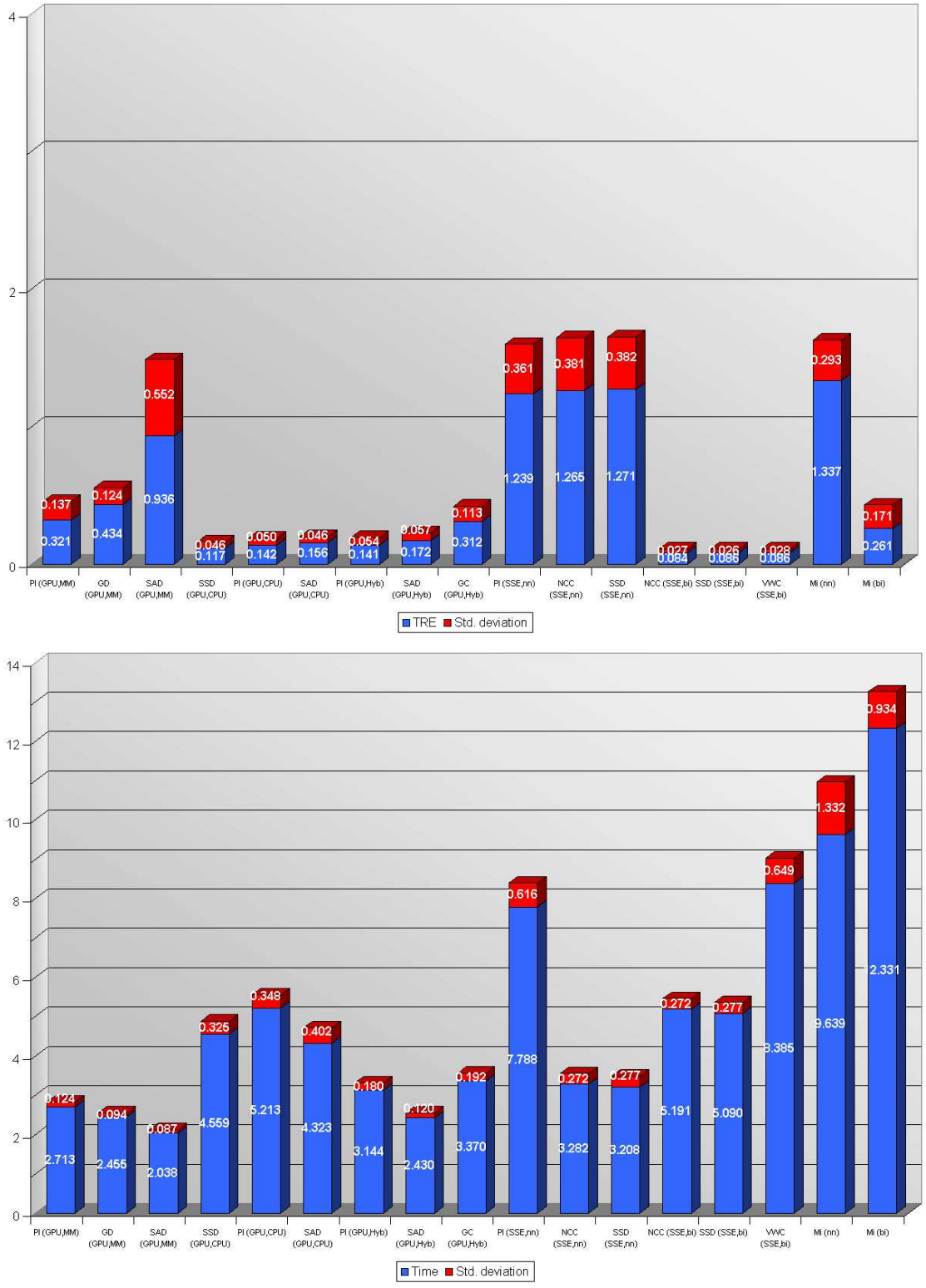


Figure 5.4: Identical CT scans - TRE (mm) and time (sec) for the most accurate three measures for each implementation

5.5.2 Experiment 2 - CT Scans at Different Points of Time

In this experiment, I used two CT scans of the same patient at different points of time. The data has been acquired using the same imaging modality and uses the same window/level settings.

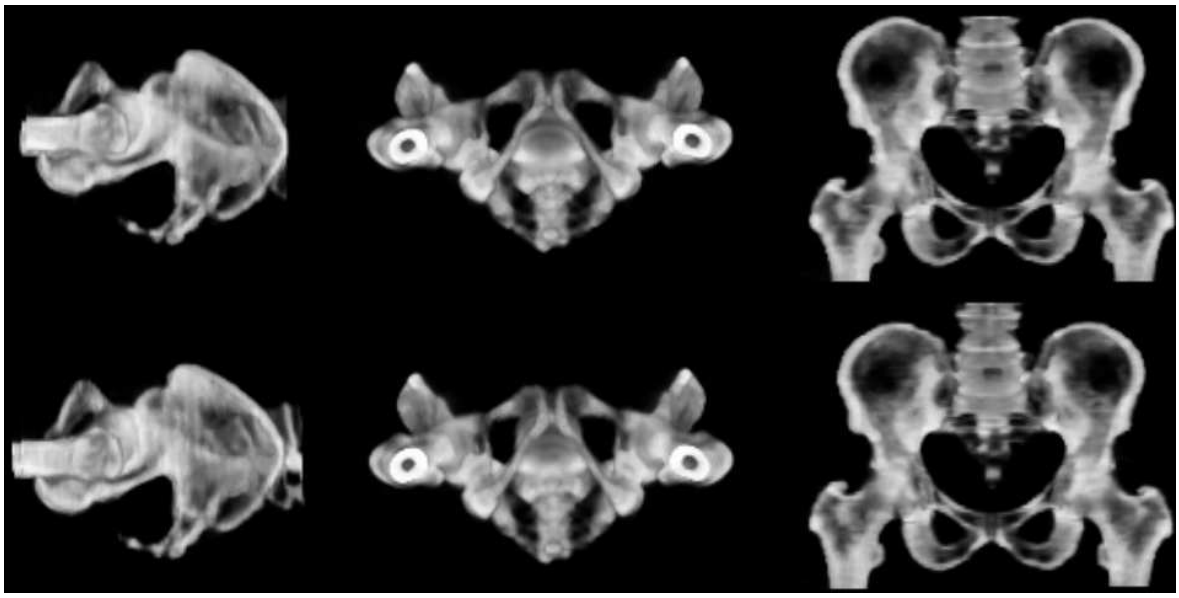


Figure 5.5: Pelvis CT scans. The fixed volume is on the upper side, the moving one on the lower side. Leg bones are positioned slightly differently and the moving volume contains more slices.

Basically the same issues as in the previous test can be observed in this case.
As before, Pattern Intensity yields very good results even when using the mipmap-based approach, although other measures offer better alignment in less time when using a combination of the two possible averaging methods.
Although the GPU does not reach TREs as small as the ones which result from the SSE implementation using bilinear interpolation, the still very good results (TRE below 0.5mm) can be obtained much faster.

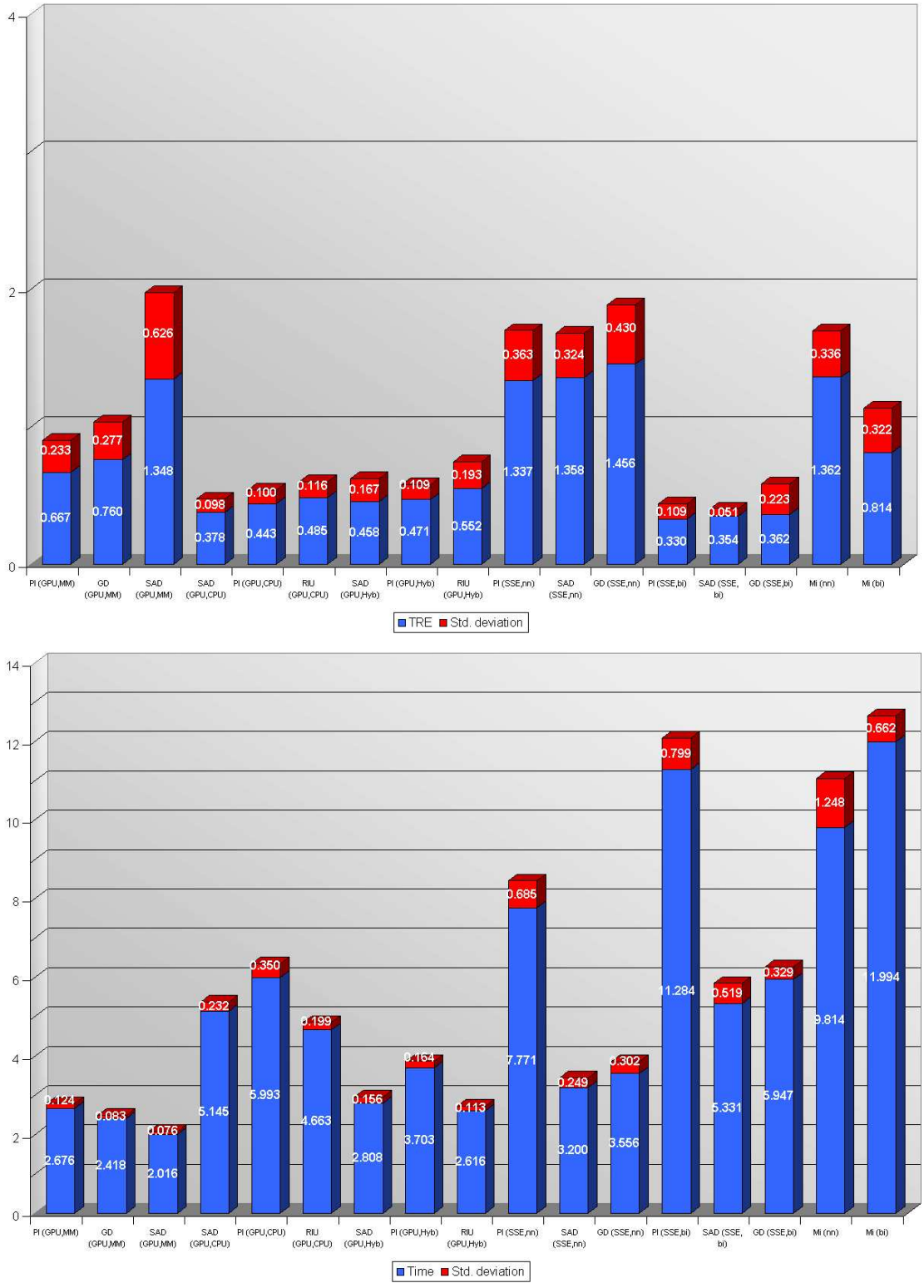


Figure 5.6: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the most accurate three measures for each implementation

5.5.3 Experiment 3 - Noisy CT Scans

In this experiment, I again used two CT scans of the same patient at different points of time. However, this time the image intensity and contrast differ slightly, with some features being visible in only one of the volumes. The data is also affected by a certain amount of noise.

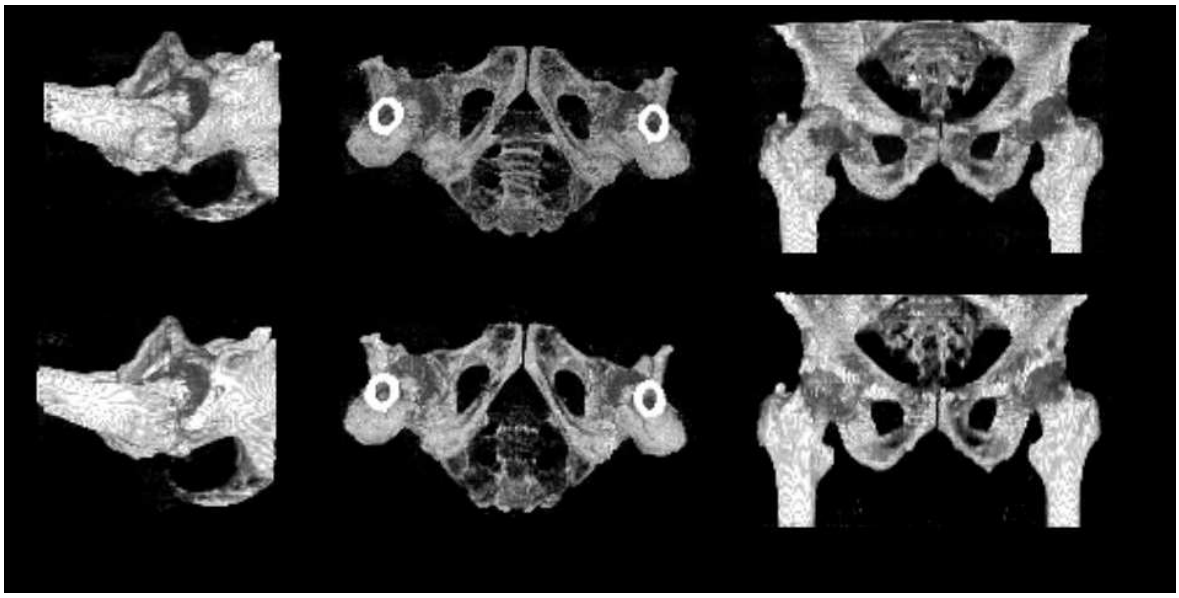


Figure 5.7: Noisy pelvis CT scans. The fixed volume is on the upper side, the moving one on the lower side. Image contrast and brightness are different, making some anatomical features visible in only the fixed volume.

The registration error is slightly larger than in the previous case, where the image intensities perfectly matched. Still, Gradient Difference and Pattern Intensity again offer the most accurate results, although PI seems to be rather imprecise in this case. This might be caused by the image noise, as the difference images do not yield smooth areas anymore.

It should be noted that especially Pattern Intensity runs much faster in the GPU-based implementation. As PI is the most complex measure, the impact of the overhead implied by the rendering pipeline is less noticeable than in other cases. A detailed analysis of the computation times for the various measures will be given in section 5.5.5.

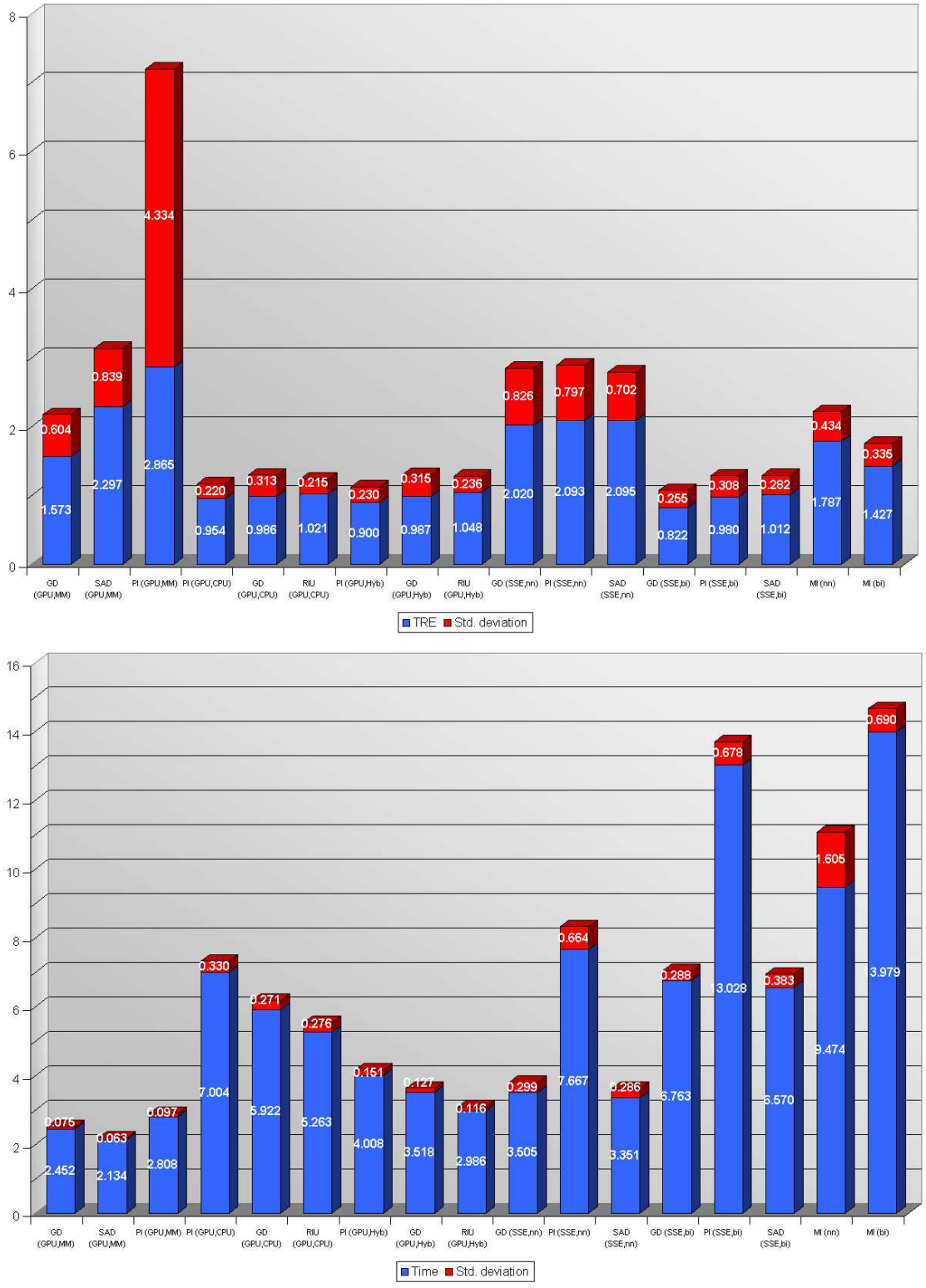


Figure 5.8: Noisy CT Scans - TRE (mm) and time (sec) for the most accurate three measures for each implementation

5.5.4 Experiment 4 - Different Types of CT Scanners

The fixed volume was acquired using a regular CT scanner, while a megavolt cone-beam CT was used for the moving volume. Thus, the two volumes differ quite strongly with respect to brightness and contrast, and some features, ie. skin, are visible in one volume but not in the other.

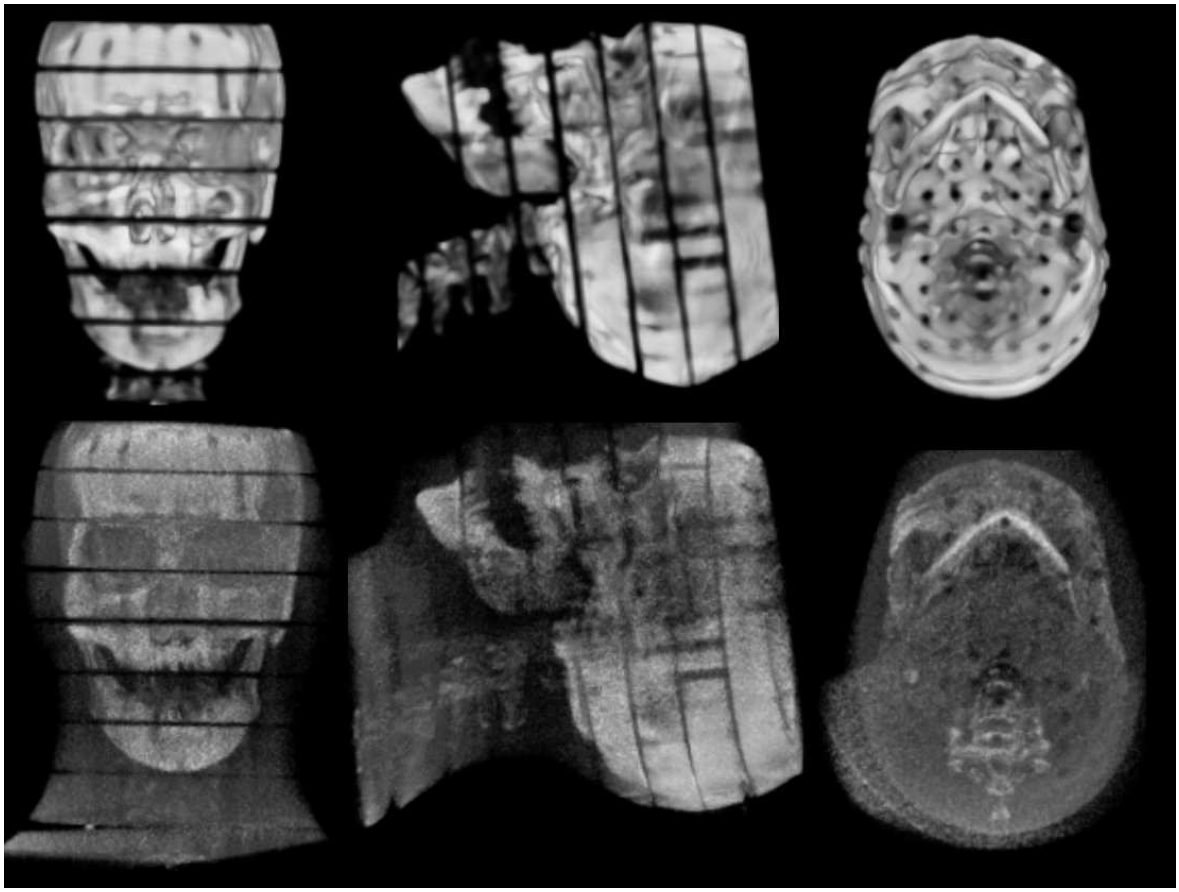


Figure 5.9: CT scans of a Rando head phantom. The fixed volume is on the upper side, the moving one on the lower side. Due to the different imaging modality, brightness, contrast and image noise vary significantly. Also, the moving volume contains more slices.

Especially the gradient-based measures, PI, LNC and VWC perform quite poorly under these conditions, where image gradients do not match and where the difference image contains a significant amount of patterns even for optimal alignment. Still, the more simple approaches like the Sums of Intensity Differences, Normalized Cross Correlation and Ratio Image Uniformity yield TREs smaller than 2mm, thus being more accurate than Mutual Information, in all implementations except for the GPU using complete mipmap-based image averaging. But even there, SAD offers a TRE very similar to that obtained using MI in less than 2.5 seconds (compared to almost 10 seconds for MI).

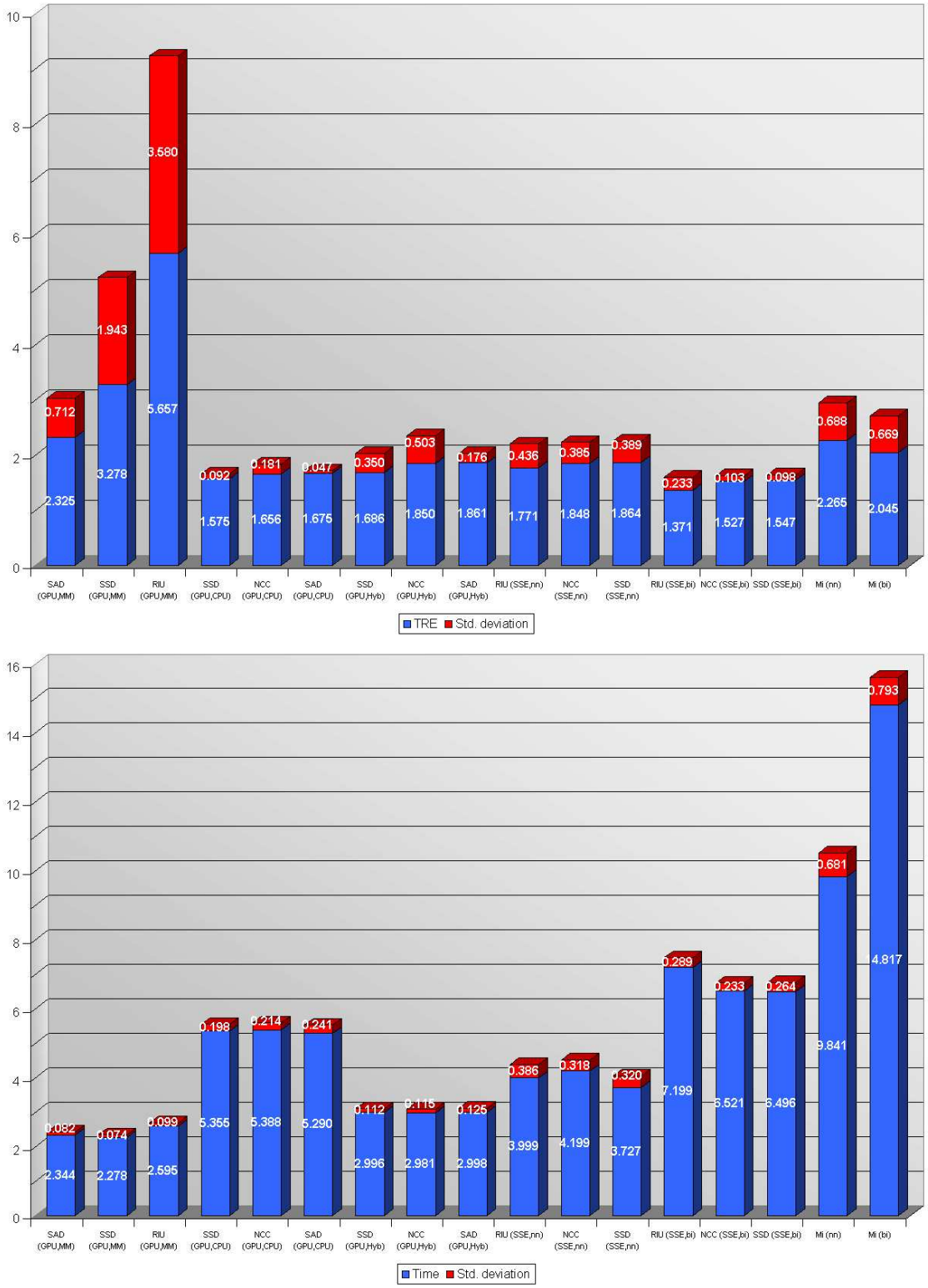


Figure 5.10: Different Types of CT Scanners - TRE (mm) and time (sec) for the most accurate three measures for each implementation

5.5.5 Similarity Measure Computation

The following table provides a short overview of only the time necessary for computing the respective similarity measures for 256x256px 8-bit greyscale images on the CPU without and with using SIMD and on the GPU with mipmap and/or CPU-based averaging. For this, we are assuming that both the images are available in the system's RAM or as texture objects on the GPU, respectively, with all values depending on the fixed image alone (eg. gradient, mean intensity, ...) having been precomputed in advance.
The values have been obtained by computing each measure 1000 times and then taking the average running time. As for the previous experiments, the testing system was the same commodity PC with an Intel Pentium4 2.4GHz processor, 1024MB RAM, an NVIDIA Geforce 6800GT AGP8x GPU with the generic drivers (release 66.93) and Microsoft Windows 2000 SP4.

Measure	CPU	CPU, SSE	GPU, mipmaps	GPU, CPU	GPU, hybrid
SAD	0.17ms	0.05ms	0.54ms	3.52ms	0.60ms
SSD	0.17ms	0.05ms	0.54ms	3.54ms	0.59ms
RIU	1.20ms	0.32ms	0.55ms	3.56ms	0.62ms
NCC	0.87ms	0.17ms	0.53ms	3.56ms	0.59ms
GC	2.46ms	0.55ms	1.15ms	7.12ms	1.29ms
GD	2.86ms	0.70ms	0.73ms	3.71ms	0.76ms
GP	1.82ms	0.45ms	0.72ms	3.70ms	0.75ms
PI	14.83ms	7.93ms	1.21ms	4.20ms	1.27ms
LNC	18.32ms	6.69ms	3.14ms	6.11ms	3.17ms
VWC	18.35ms	6.59ms	1.78ms	4.77ms	1.81ms

As we can see, downloading an image from the GPU and averaging on the CPU takes about 3ms more than when using mipmaps, which is 2 to 6 times as much as it takes to render the similarity image iself. Using the hybrid approach, we only need about 0.05ms more than for the fastest method, while keeping the rounding errors very small, as can be seen from the results listed in the appendix.
Because the hybrid averaging method is only minimally slower than the mipmap-based one, at first it might seem confusing that the differences in the actual registration time are more noticeable. It should be kept in mind that the registration algorithm does not evaluate a fixed number of similarity measures, but that it will abort if the changes obtained in the measure drop below a certain threshold when the pose parameters are adjusted. As the mipmap-based averaging introduces errors, minor changes in the similarity images might be lost during the rounding process, not affecting the final result. This means that the abortion threshold of the optimizer is reached much faster, which is not the case for the much more accurate hybrid averaging, which will thus cause the optimizer to evaluate a larger number of poses until it terminates.

Also, the assumption that simple measures will perform rather poorly in comparison with the CPU and that more complex ones have a much better potential for being accelerated is strongly confirmed by these results. While the sum of intensity differences is three times

slower than a classical CPU implementation, measures operating on large neighborhoods, ie. PI, LNC and VWC execute 2 to 5 times faster on the GPU than on any CPU-based implementation. For the other measures, having a medium complexity, the GPU implementation remains between the SISD and SIMD approaches of the CPU version.

6 Conclusion

6.1 Results

6.1.1 GPU-based acceleration

The results previously presented illustrate that using modern graphics hardware for similarity measure computation can significantly accelerate the process of medical image registration. While assessing the similarity at the individual pixel locations is very accurate even when using an 8-bit color buffer for output, the result can be significantly distorted by using the hardware-accelerated generation of recursively smaller texture images, or mipmaps, for summing up the individual values. This can be avoided by copying the similarity image from the video board to the system's RAM for completing the summation on the CPU, but this approach is much slower due to the slow data transfer when using an AGP interface. However, a hybrid approach exploiting the advantages of both averaging methods yields results that are as accurate as a CPU-based implementation, but in up to five times less time.
When we look at the times of registration when using automatically generated mipmaps, we can notice that the differences in the running time across the different measures are rather small, as most of the time is used for generating the orthogonal projections necessary for registration. Hence, the user can choose the similarity measure best suited for the respective data set without having to accord too much attention to speed issues.
The experiments also show that some similarity measures, namely Pattern Intensity and Gradient Difference, deliver very accurate registration even when mipmaps are used for completely summing up the resulting pixels. When registering data sets acquired by using different CT energy levels, similarity measures examining spatial information instead of just the intensity at the individual pixel location deliver very poor results. Still, measures examining pixel intensities alone, especially the Sum of Absolute Differences, yield a very accurate registration. Using a high-level shading language, the respective measures are implemented with ease and using a familiar, C-like syntax. However, some basic knowledge of graphics APIs and the GPU's rendering pipeline is required.

6.1.2 SSE-based acceleration

Single instruction, multiple data instruction sets supported by modern micro-processors (like Intel's SSE/SSE2) also offer a very good possibility for accelerating the computation of similarity measures. However, this implies that the respective two-dimensional images are available in the system's main RAM. Most registration applications imply two-dimensional images to be generated from volumetric data, either for 2D-3D or 3D-3D registration, or at least a translation and rotation of a 2D image. These tasks can be performed much faster by modern video boards, as the necessary functionality is supported directly in hardware, which is not the

case for general-purpose microprocessors. The experimental results illustrate very well how an operation that comes at virtually no cost on GPUs, namely bilinear image interpolation, can have a significant impact on the overall registration time. Thus, the issue of supplying the necessary image data to the SIMD-capable CPU necessitates the moving image to be either generated by means of software-based methods or to be transferred from the graphics board to the main RAM, both of which are rather slow. However, as the image download from the GPU will be much faster when using the upcoming PCI Express bus and most of the similarity measures run faster on the CPU, SIMD-based acceleration can also heavily contribute to speeding up image registration. Furthermore, this acceleration approach can also be used for distributed registration applications running on multiple processors.
Rewriting algorithms for SSE can be rather tedious, as one is limited to using simple assembler-like instructions. Additionally, one has to solve the problems implied by the fact that all instructions are operating on data blocks of a fixed length (128 bits for SSE), independently of the data type, and thus the number, of actual data elements stored therein. Together with the issue of memory alignment, these two factors can result in a significant cost, with respect to both implementation effort and algorithm execution time, in order to rearrange the input data so that it can be processed using SSE.

6.2 Problems/Discussion

The presented algorithms were tested using a 3D-3D registration application which attempts to align two volumes by succesively aligning two-dimensional orthographic projections of the volumetric data. For the whole registration process only nine such projections need to be computed (three times for each of the three coordinate axes). As the volume rendering is the most expensive operation during the registration process, be it 2D-3D or 3D-3D, it will normally be carried out on the GPU, as most modern boards offer the respective functionality. In the case of 2D-3D registration, where the number of necessary DRRs can be very high, transferring the DRRs from the GPU to the main RAM can have a serious impact on performance when using a PCI or AGP port. Especially this kind of application would hence benefit from having a GPU-based implementation of similarity measures, assuming that the video board offers an accurate and fast means of summing up the similarity image.

Hardware-specific features and performance, especially in the graphics domain, change and evolve very fast. Thus, the comparison between the CPU-based and GPU-based algorithms is of limited use when attempting to predict which of the methods might prove more attractive in the future. Although the current trend shows a much faster evolution of graphics hardware (a performance increase of factor 2 per year for GPUs versus a factor of 1.5 for CPUs [9]), this paper is intended to give an overview of the main issues of accelerating intensity-based rigid registration.

6.3 Future Work

The presented approaches for accelerating image registration can further be examined and improved.

- The main problem when using the GPU for the similarity measure computation is the summation of the individual values, which can currently be accomplished by a fast, but inaccurate approach or by an accurate but very slow one. Although a hybrid approach can yield good results, upcoming graphics hardware and drivers for currently available GPUs promise to solve this issue by supporting automatic mipmap generation for textures with 16 or even 32 bits per color channel, were the rounding errors should be much less important. Also, on the new PCI Express bus, intended to replace AGP, downloading data from the GPU to the host machine will be just as fast as uploading data, so even complete CPU-based averaging might provide a possible alternative on newer hardware platforms.

- The current GPU implementation assesses the similarity between one fixed and one moving image in each step. However, it would also be possible to combine several moving images into one larger texture and thus compute several similarity images in a single rendering pass. This would reduce the number of rendering passes and also the total number of texture accesses within the fragment shader (as the fixed image has to be accessed only once).

- Graphics boards are used more and more for general purpose computation and GPU manufacturers have been forced to take this fact into consideration when designing new chips. Thus, some of the restrictions currently imposed by the GPU environment (ie. expensive conditional statements, no pointer data types, etc.) are very likely to disappear in the near future. Over the last years, GPU performance has increased at much higher rates than that of CPUs. As this evolution is expected to continue in the future, the performance difference between CPU and GPU implementations will get even bigger. Video boards might soon reach performance levels where real-time rigid registration becomes possible.

- While this work has examined intensity-based registration, fast GPU-based segmentation approaches have also been proposed [39], thus making a GPU implementation of feature-based registration algorithms an interesting possibility for applications where intensity-based methods cannot be applied.

- Being a very popular measure, especially for inter-modal registration, it would be very interesting to implement and validate Mutual Information on a GPU supporting vertex texture access.

- The presented measures assess the similarity between two-dimensional images only. But besides 2D textures, modern graphics hardware also offers support for 3D textures (frequently used for volume rendering applications) so it should be possible to implement volumetric similarity measures as well.

- To improve the performance of the SSE-based approaches, it would definitely make very much sense to find a way of transforming the two-dimensional images using SIMD calls as well, optimally without having to read the respective image twice, ie. once for transforming it and then again to assess the similarity.

- More experiments are necessary to better determine the accuracy of the presented methods and of the individual similarity measures under different conditions and for different

target objects. It would definitely make much sense to determine the ground truth by means of a more accurate approach (ie. fiducial markers or a stereotactc frame) instead of another intensity-based measure of similarity.

A Detailed Experimental Results

A.1 Experiment 1 - Identical CT scans

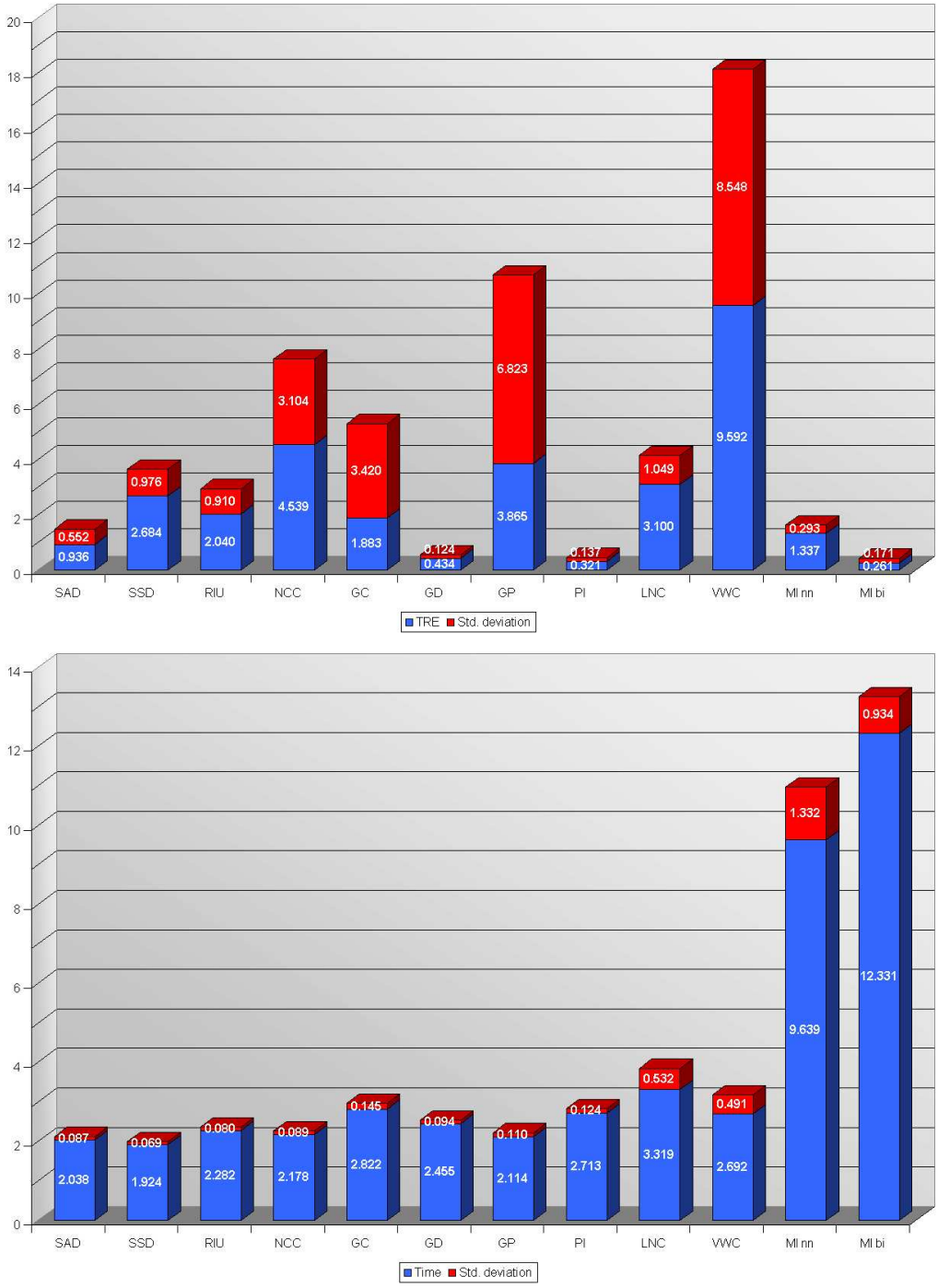


Figure A.1: Identical CT scans - TRE (mm) and time (sec) for the GPU-based registration, using mipmaps for averaging

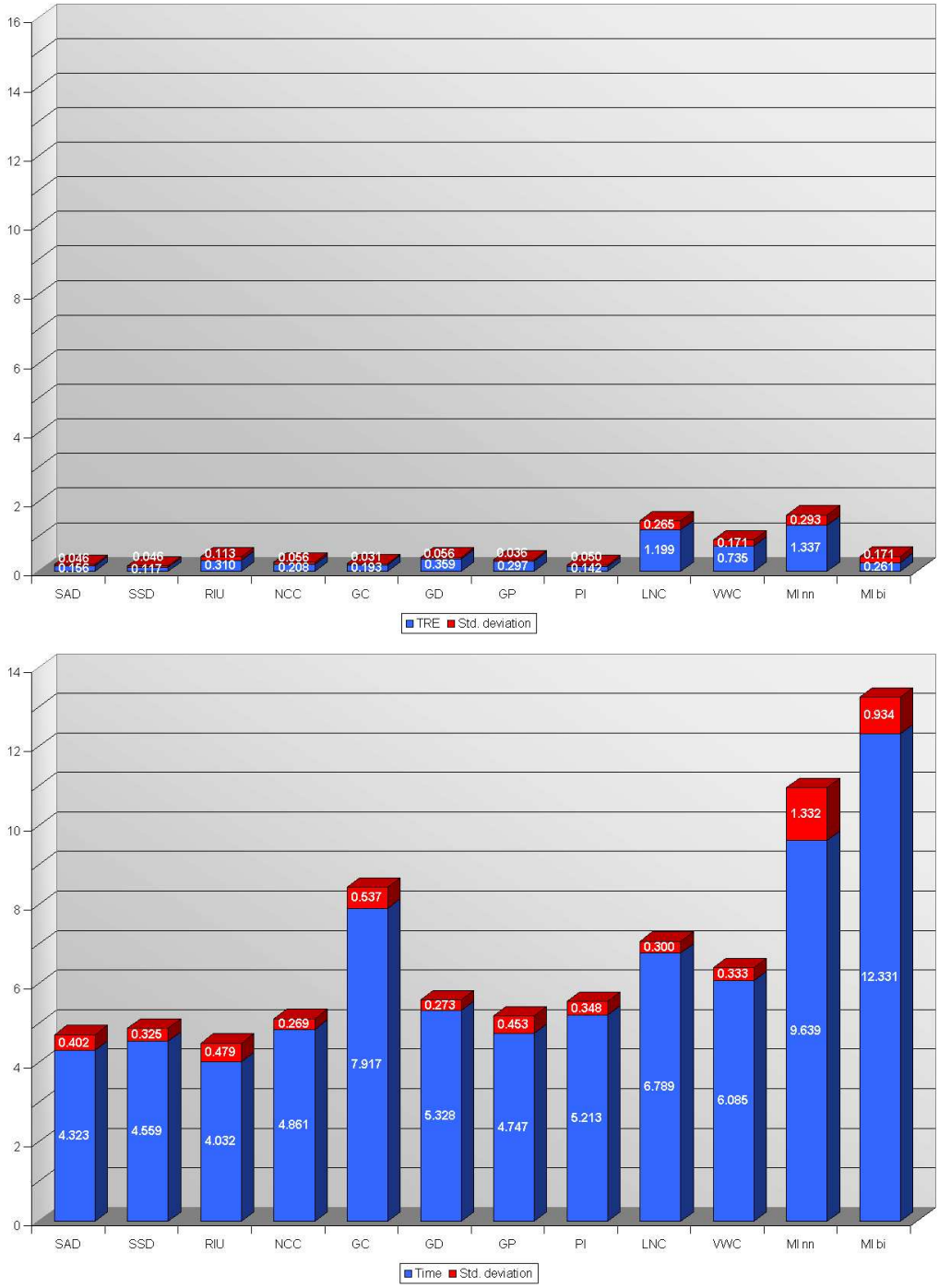


Figure A.2: Identical CT scans - TRE (mm) and time (sec) for the GPU-based registration, using the CPU for averaging

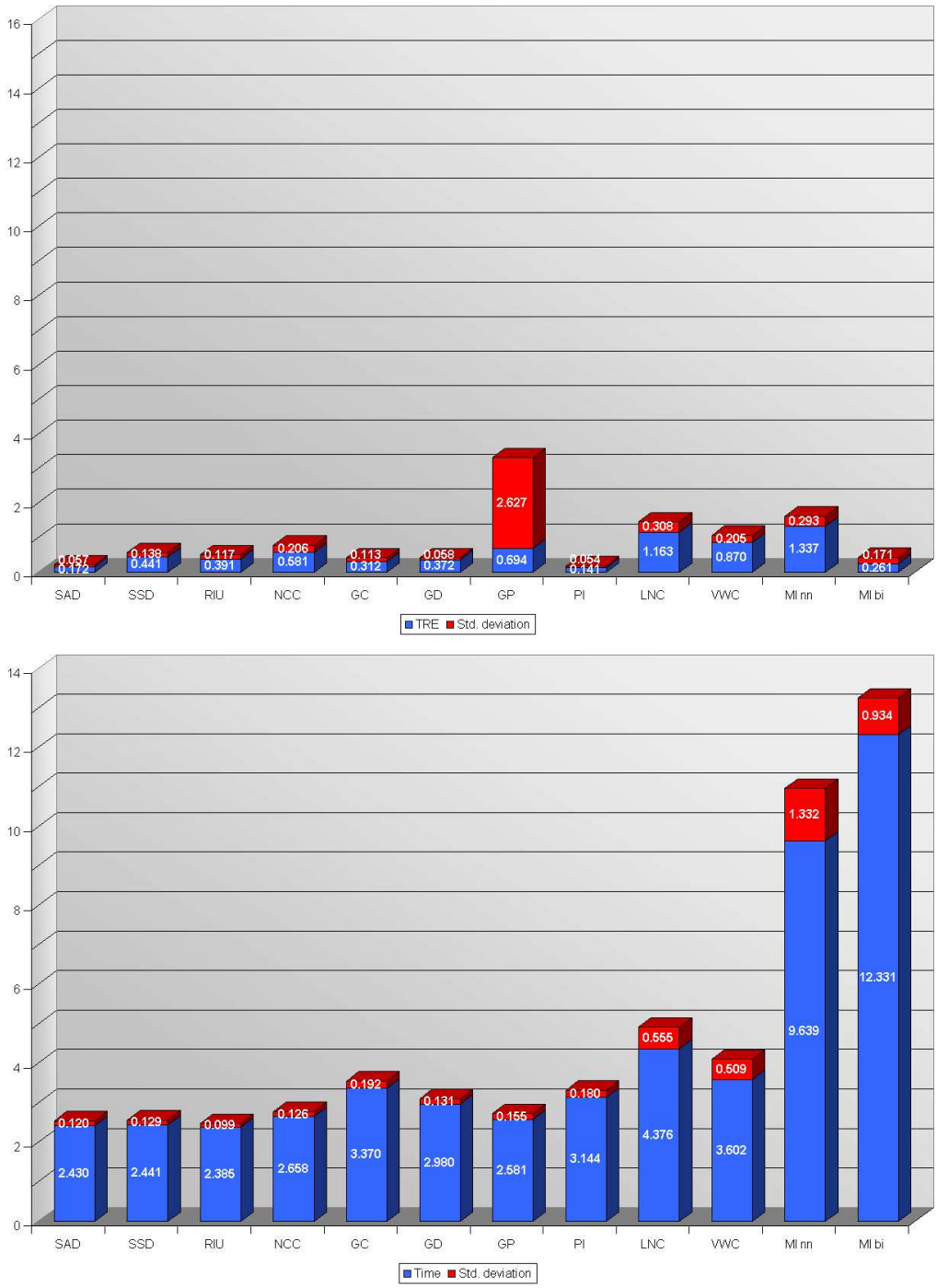


Figure A.3: Identical CT scans - TRE (mm) and time (sec) for the GPU-based registration, using the hybrid approach for averaging

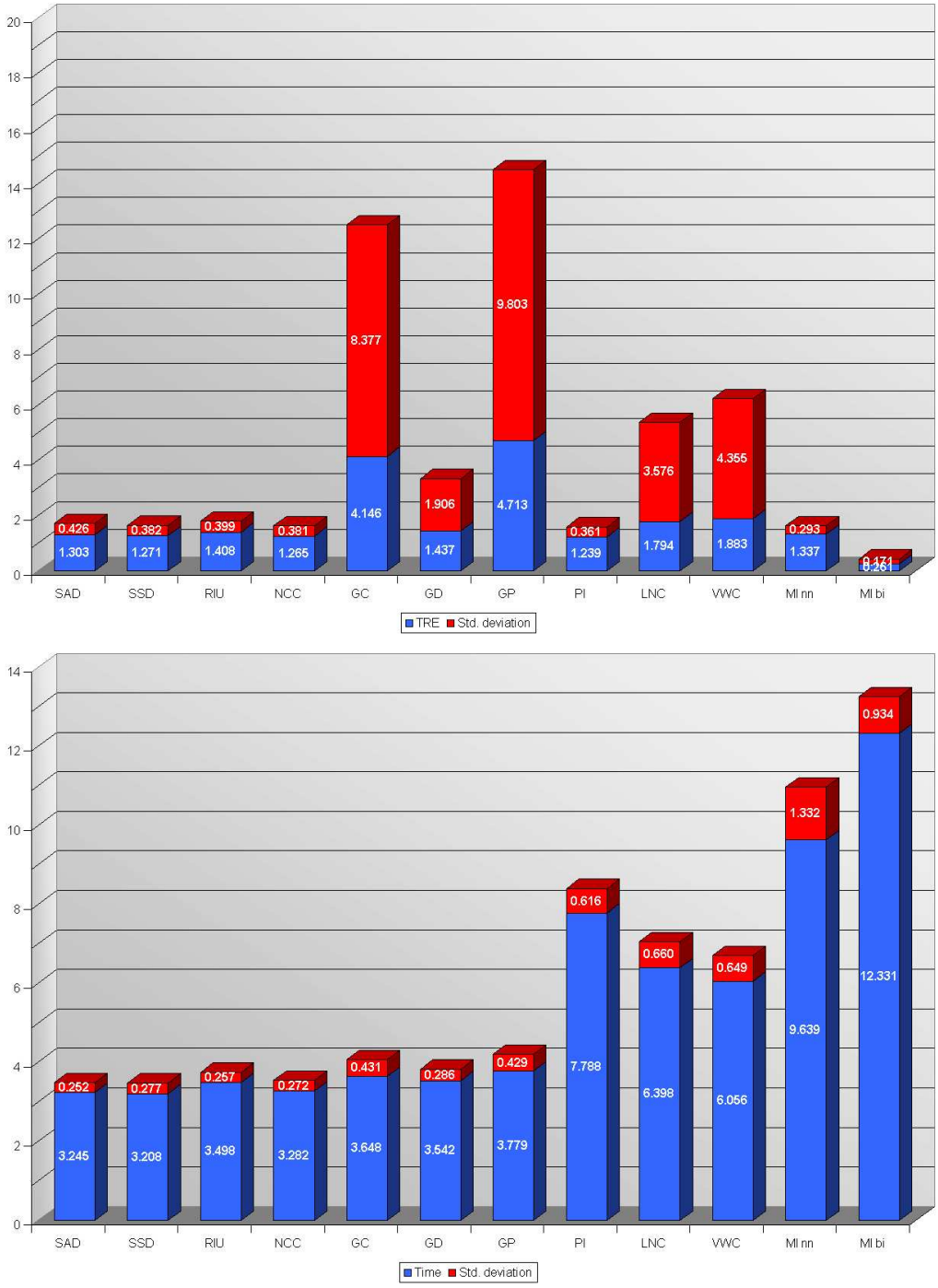


Figure A.4: TRE (mm) and time (sec) for the SSE-based registration, using nearest-neighbor interpolation when rotating the 2D image

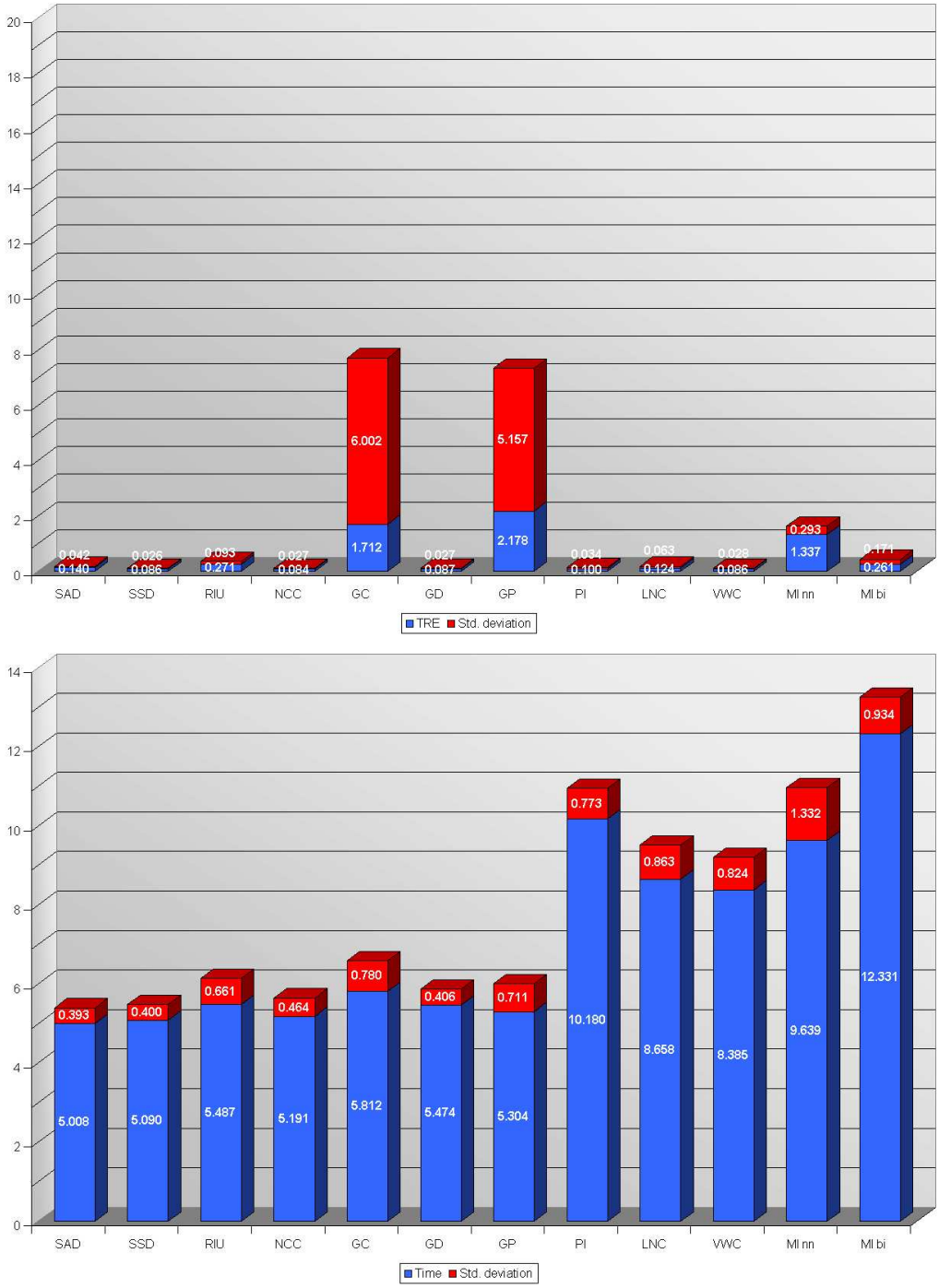


Figure A.5: Identical CT scans - TRE (mm) and time (sec) for the SSE-based registration, using bilinear interpolation when rotating the 2D image

A.2 Experiment 2 - CT Scans at Different Points of Time

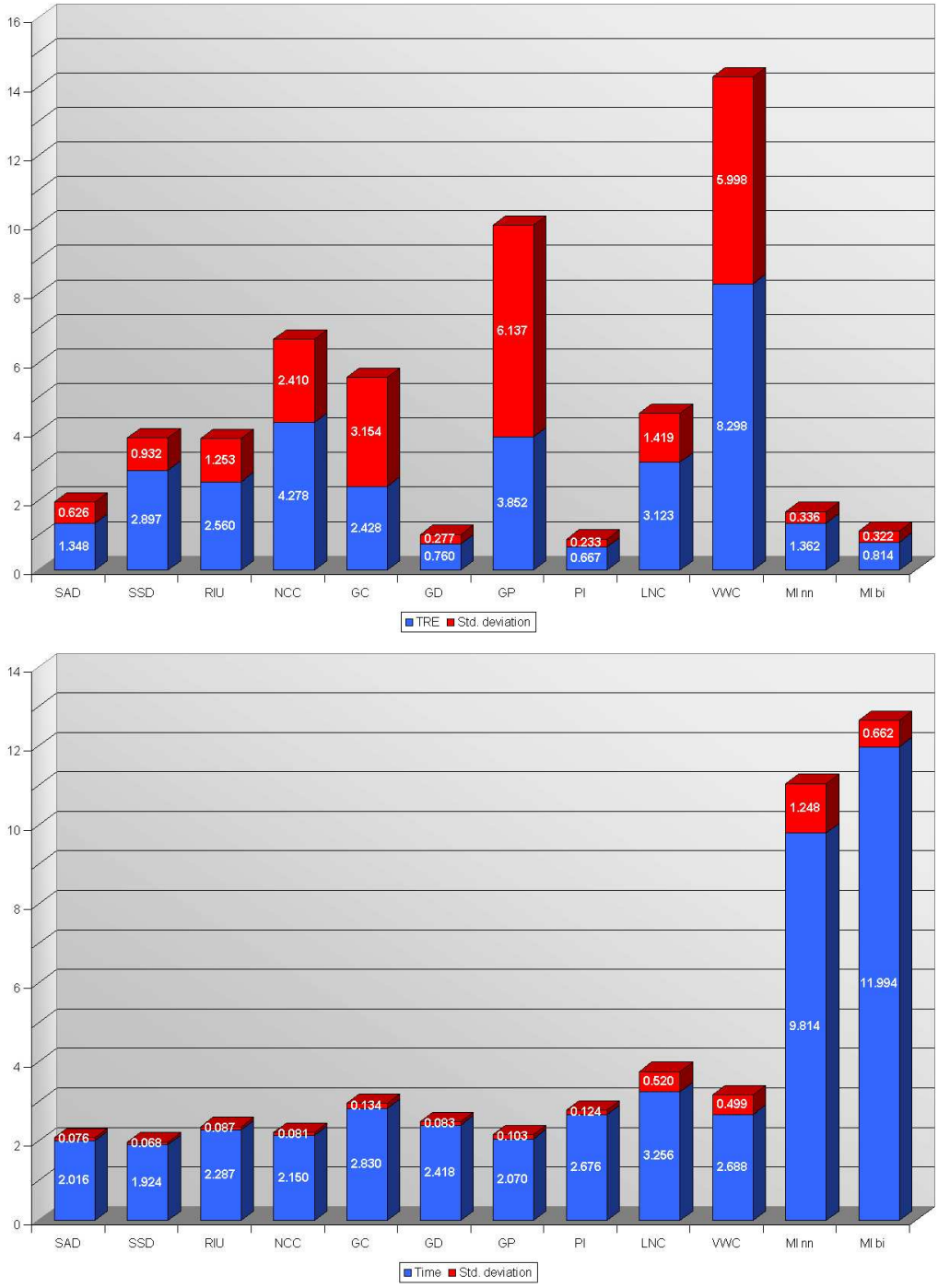


Figure A.6: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the GPU-based registration, using mipmaps for averaging

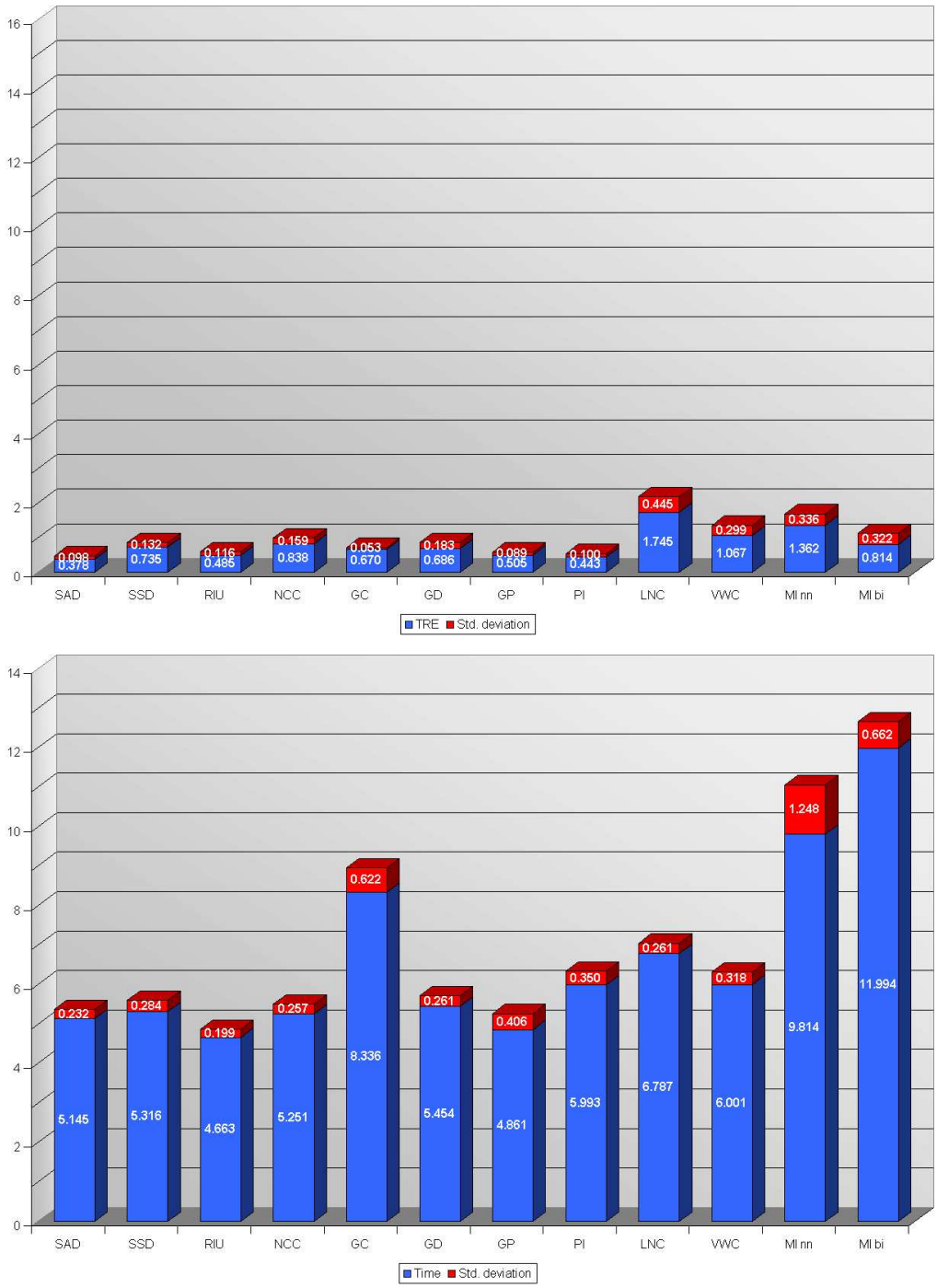


Figure A.7: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the GPU-based registration, using the CPU for averaging

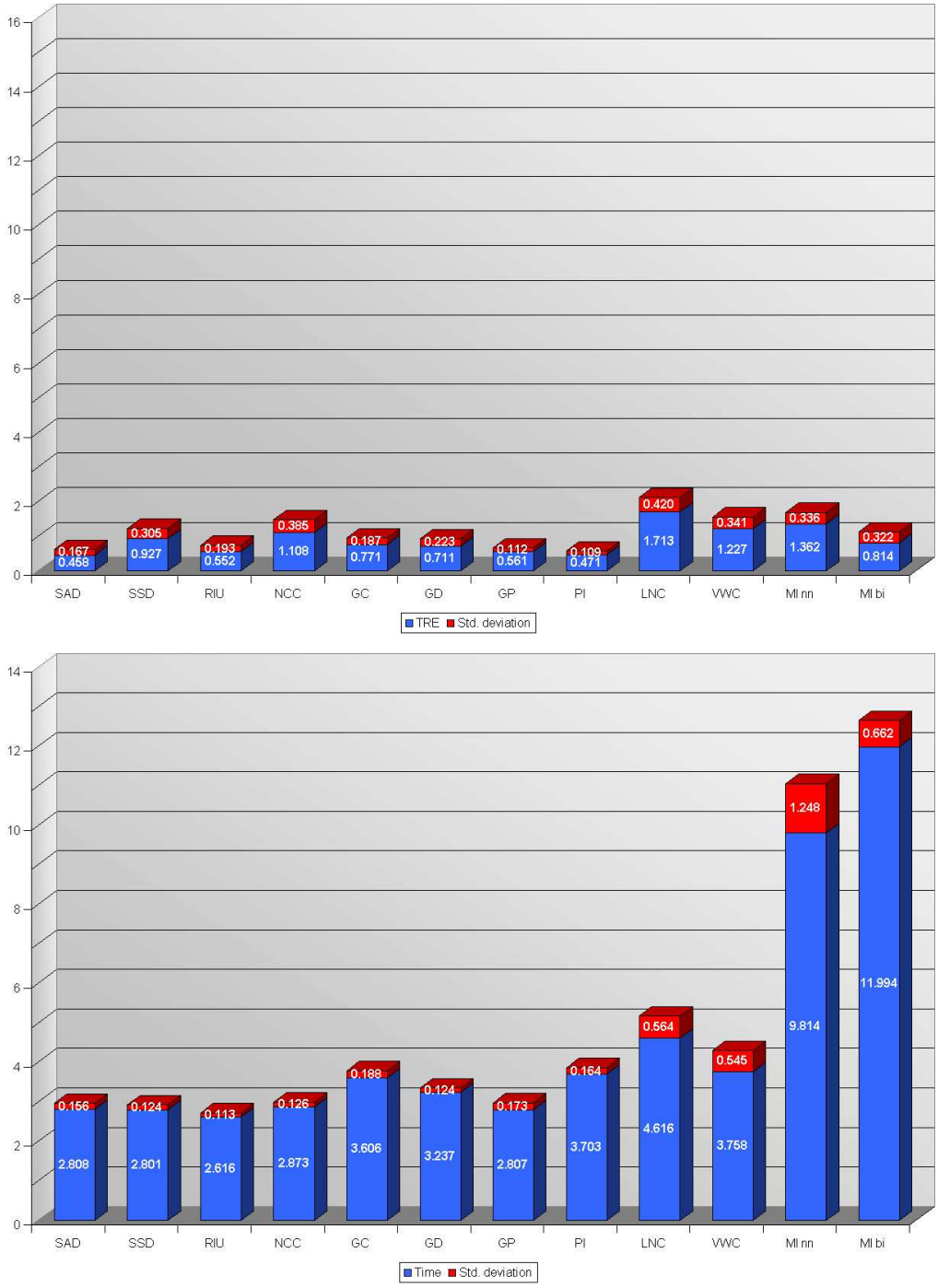


Figure A.8: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the GPU-based registration, using the hybrid approach for averaging

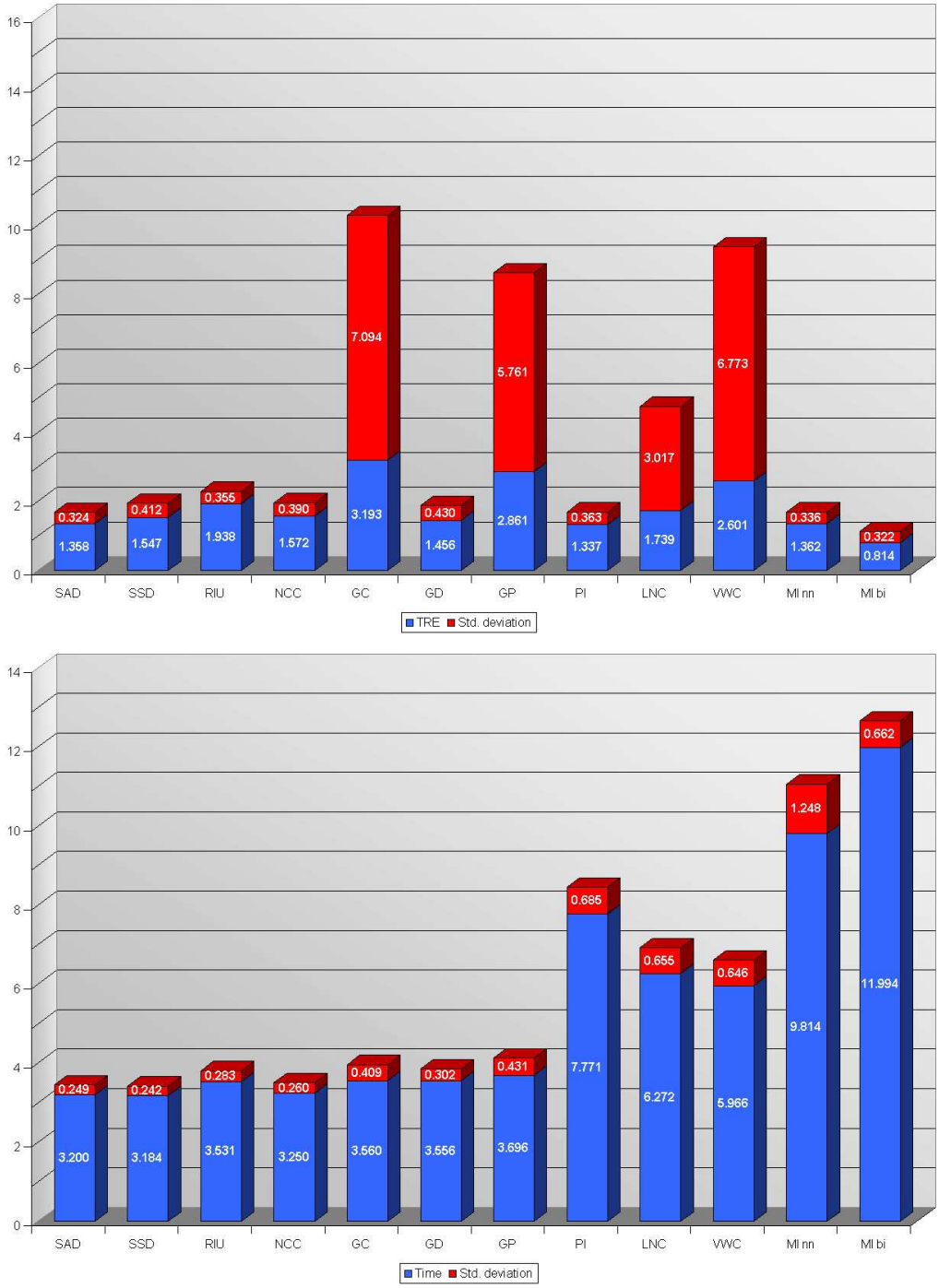


Figure A.9: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the SSE-based registration, using nearest-neighbor interpolation when rotating the 2D image

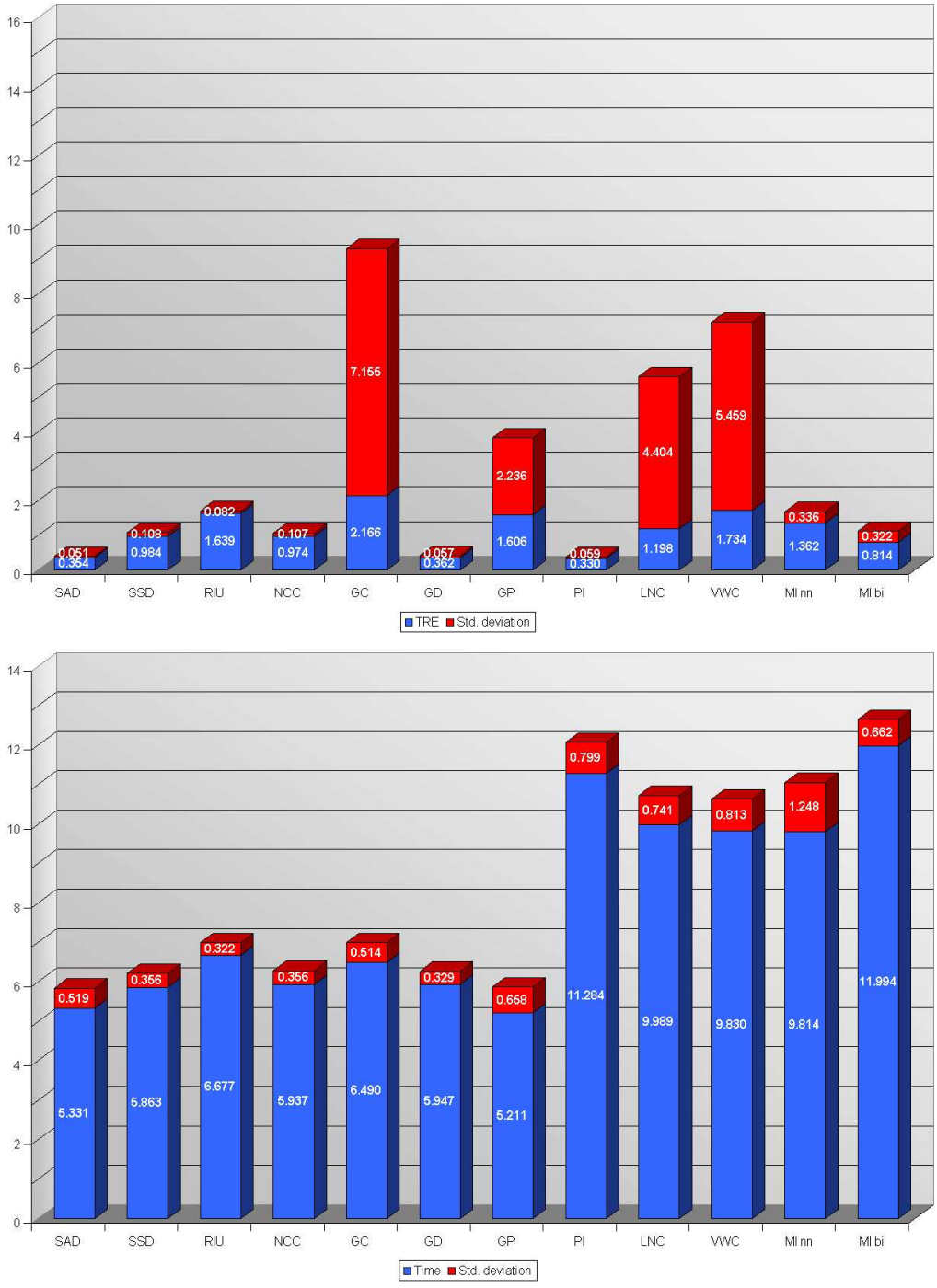


Figure A.10: CT Scans at Different Points of Time - TRE (mm) and time (sec) for the SSE-based registration, using bilinear interpolation when rotating the 2D image

A.3 Experiment 3 - Noisy CT Scans

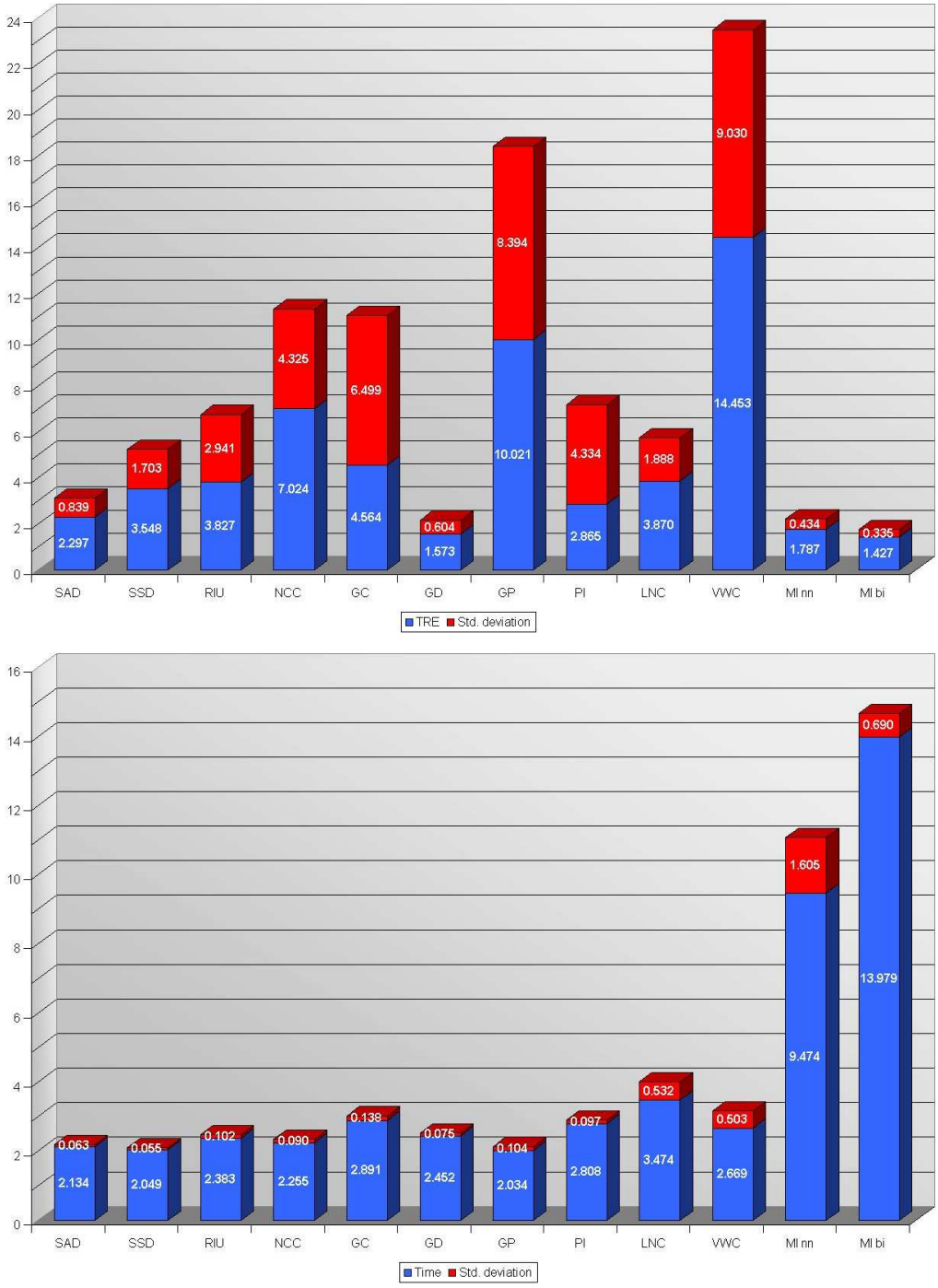


Figure A.11: Noisy CT Scans - TRE (mm) and time (sec) for the GPU-based registration, using mipmaps for averaging

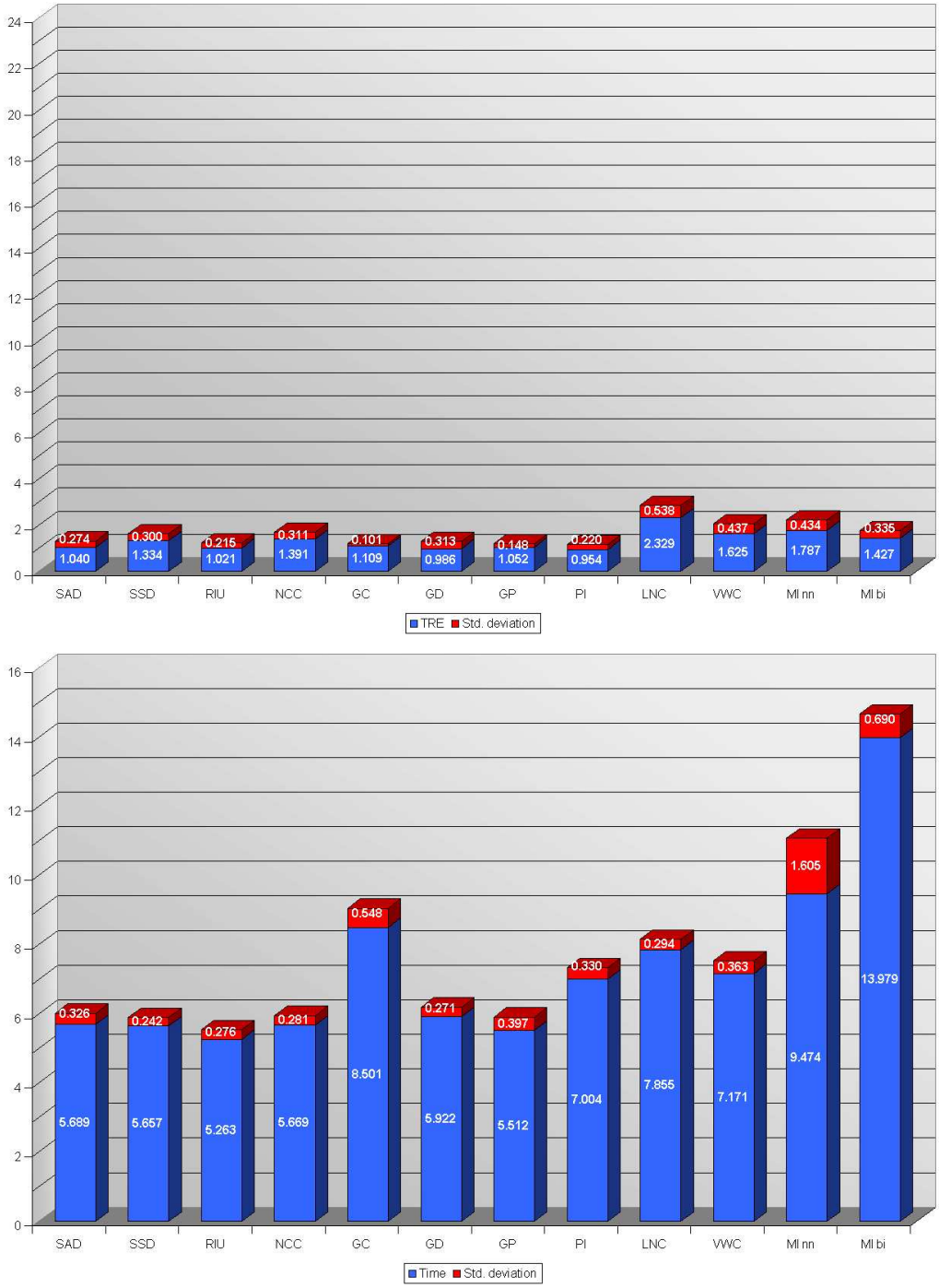


Figure A.12: Noisy CT Scans - TRE (mm) and time (sec) for the GPU-based registration, using the CPU for averaging

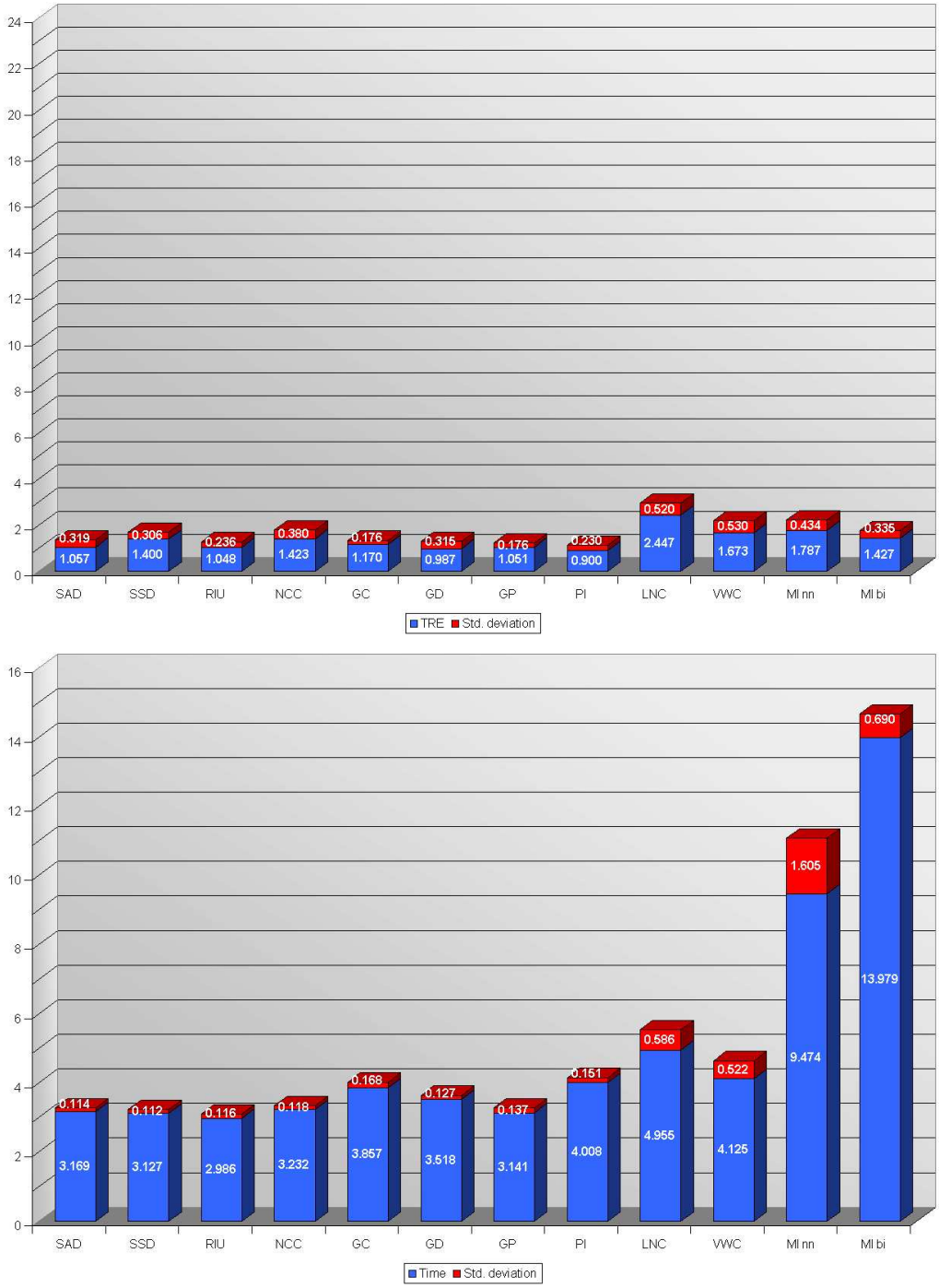


Figure A.13: Noisy CT Scans - TRE (mm) and time (sec) for the GPU-based registration, using the hybrid approach for averaging

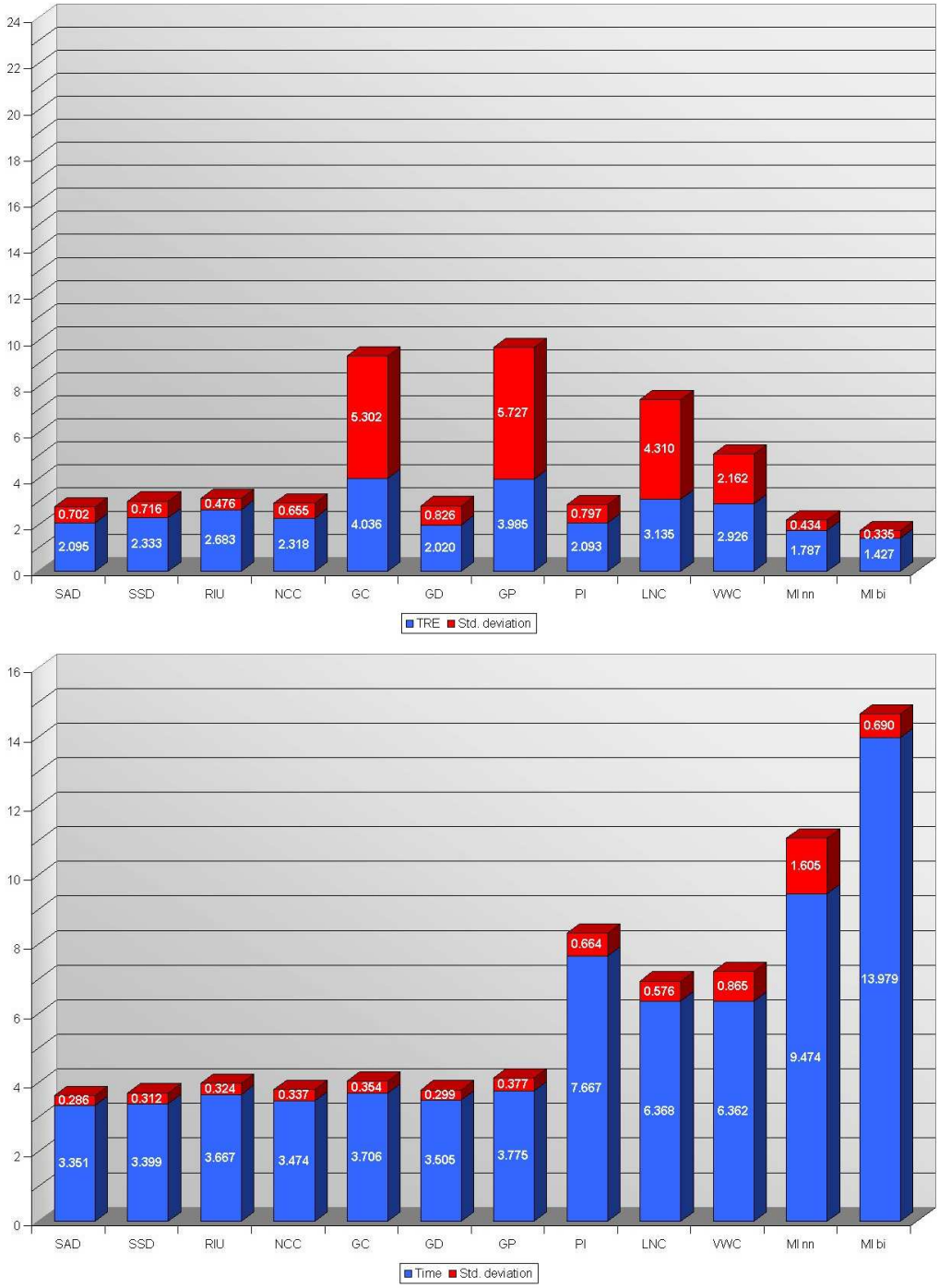


Figure A.14: Noisy CT Scans - TRE (mm) and time (sec) for the SSE-based registration, using nearest-neighbor interpolation when rotating the 2D image

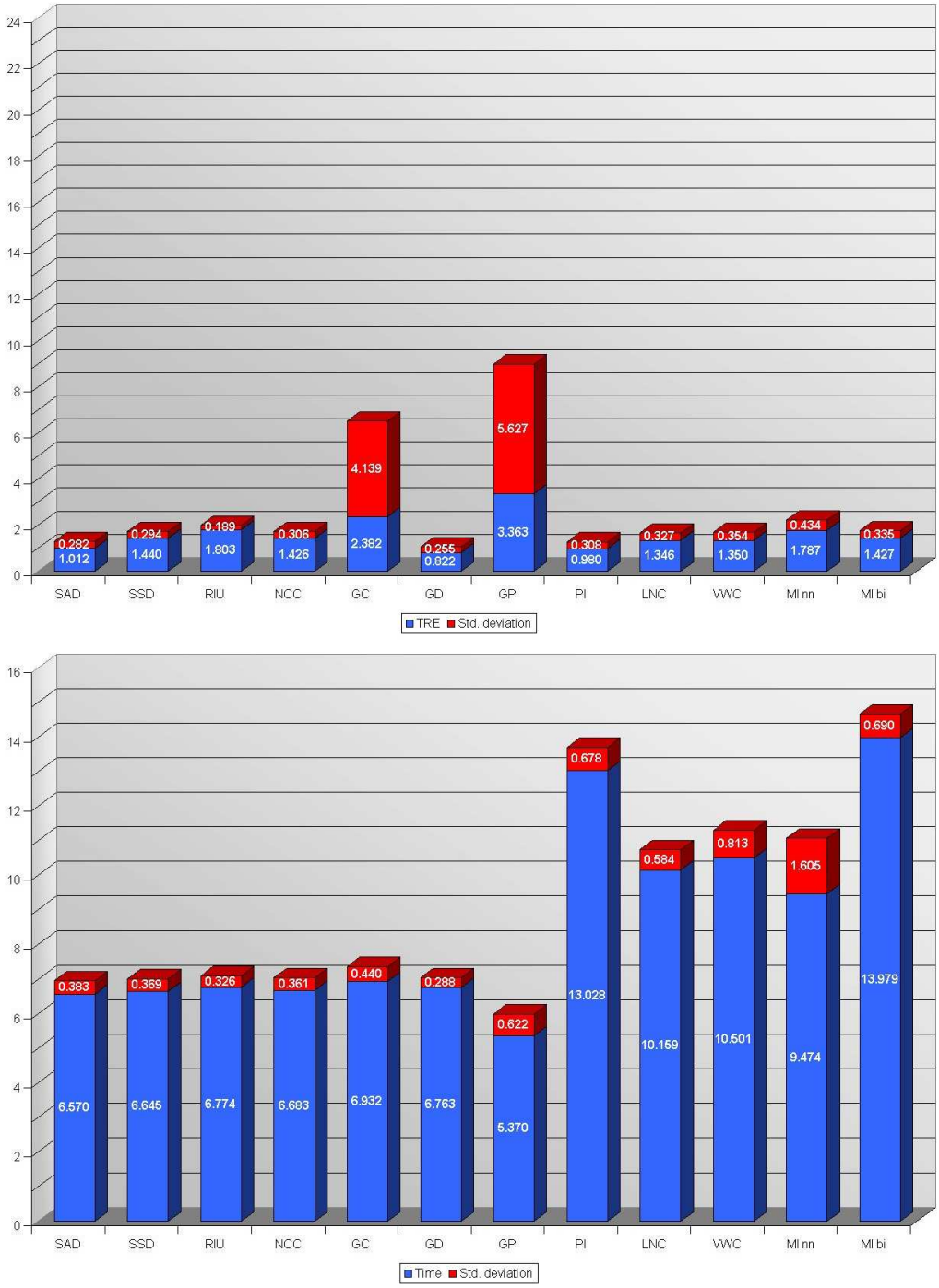


Figure A.15: Noisy CT Scans - TRE (mm) and time (sec) for the SSE-based registration, using bilinear interpolation when rotating the 2D image

A.4 Experiment 4 - Different Types of CT Scanners

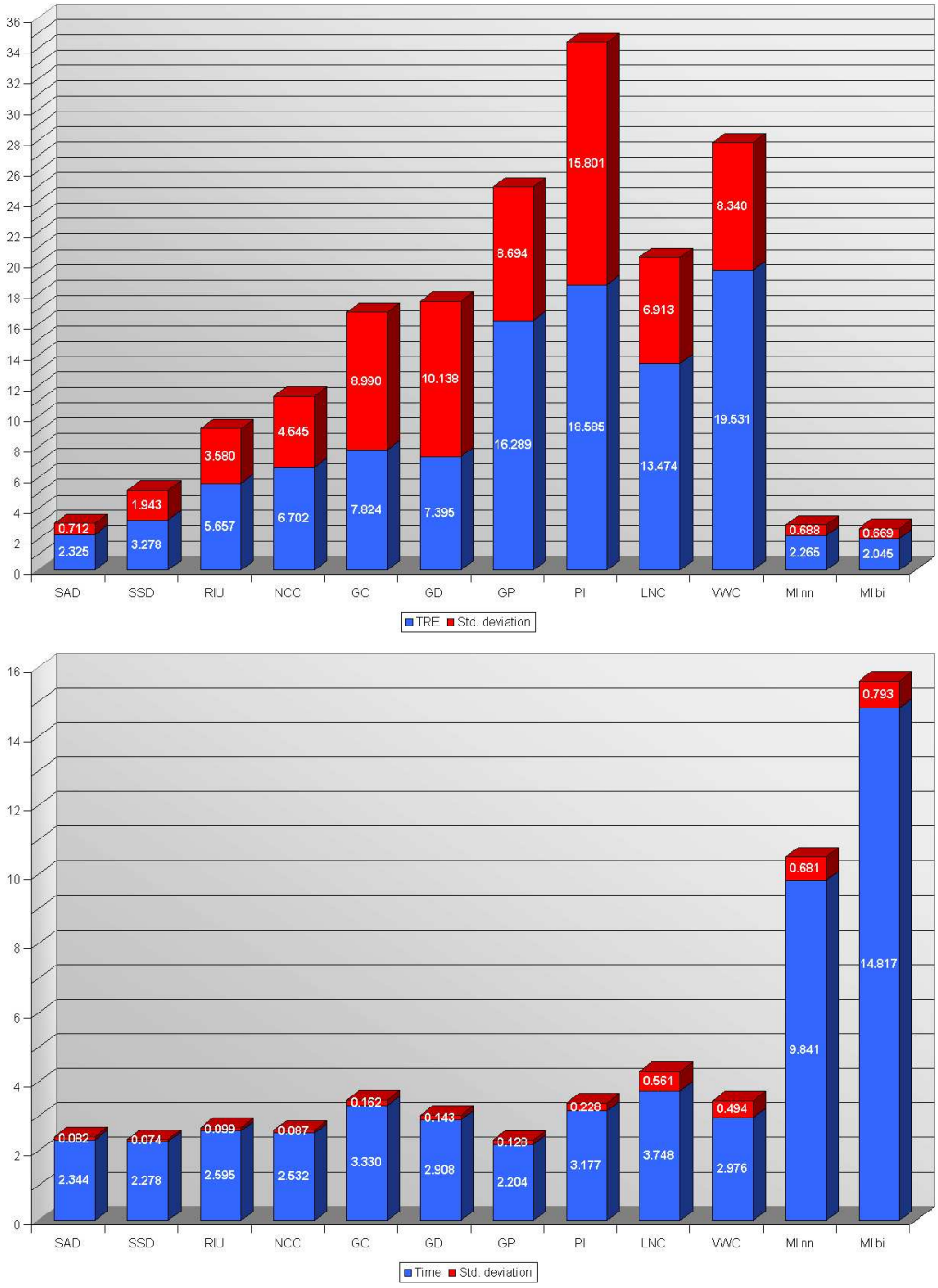


Figure A.16: Different Types of CT Scanners - TRE (mm) and time (sec) for the GPU-based registration, using mipmaps for averaging

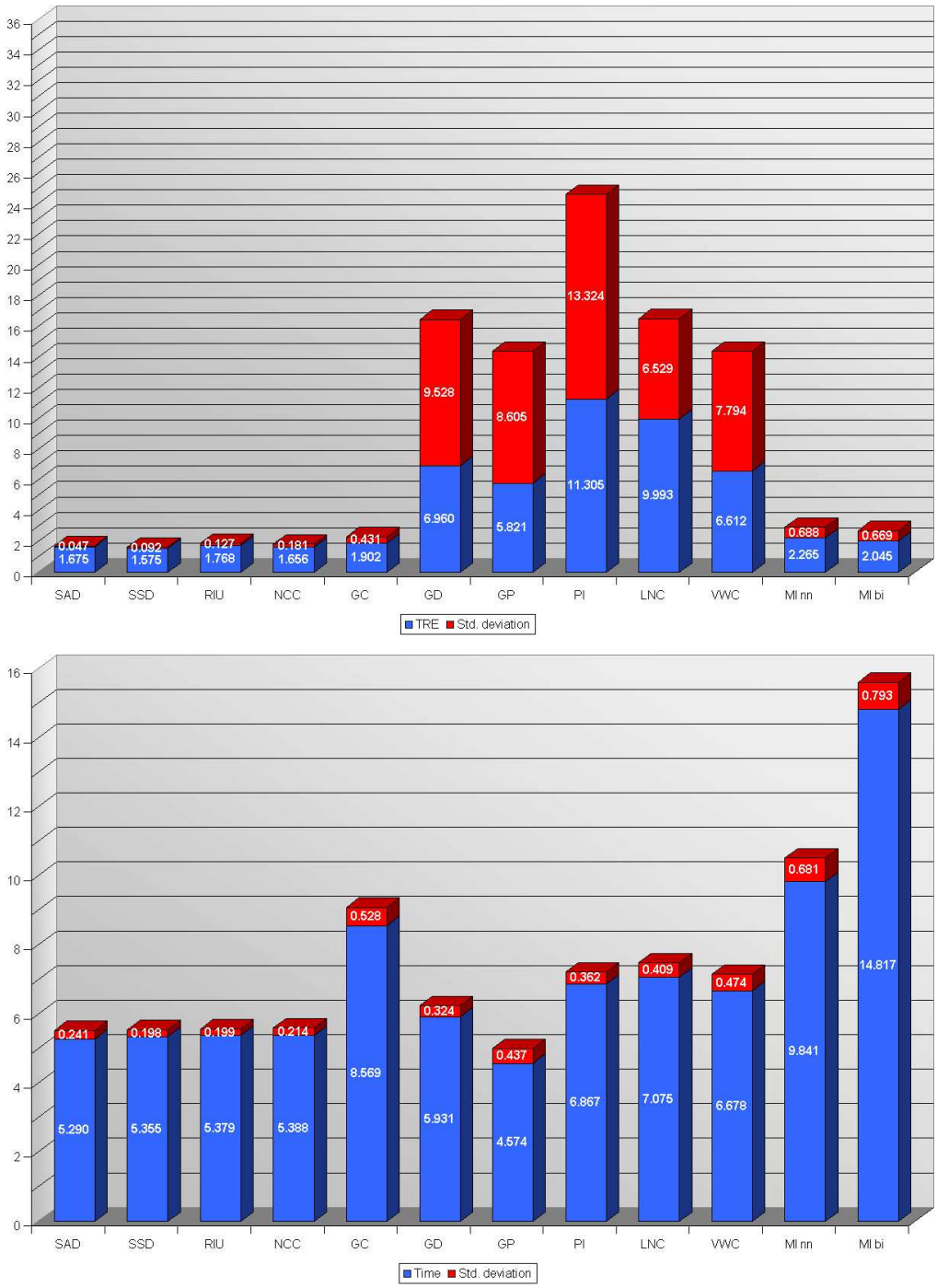


Figure A.17: Different Types of CT Scanners - TRE (mm) and time (sec) for the GPU-based registration, using the CPU for averaging

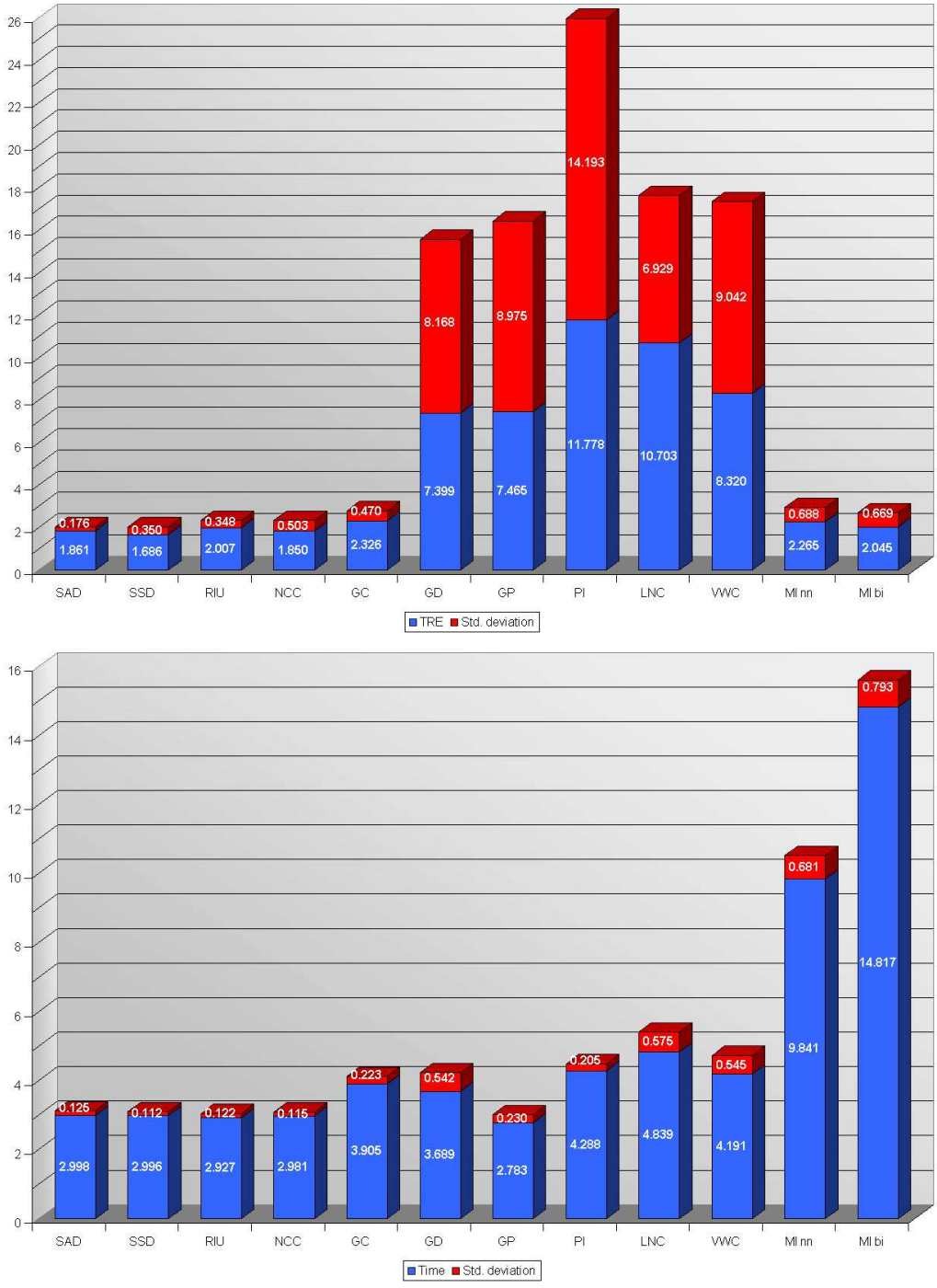


Figure A.18: Different Types of CT Scanners - TRE (mm) and time (sec) for the GPU-based registration, using the hybrid approach for averaging

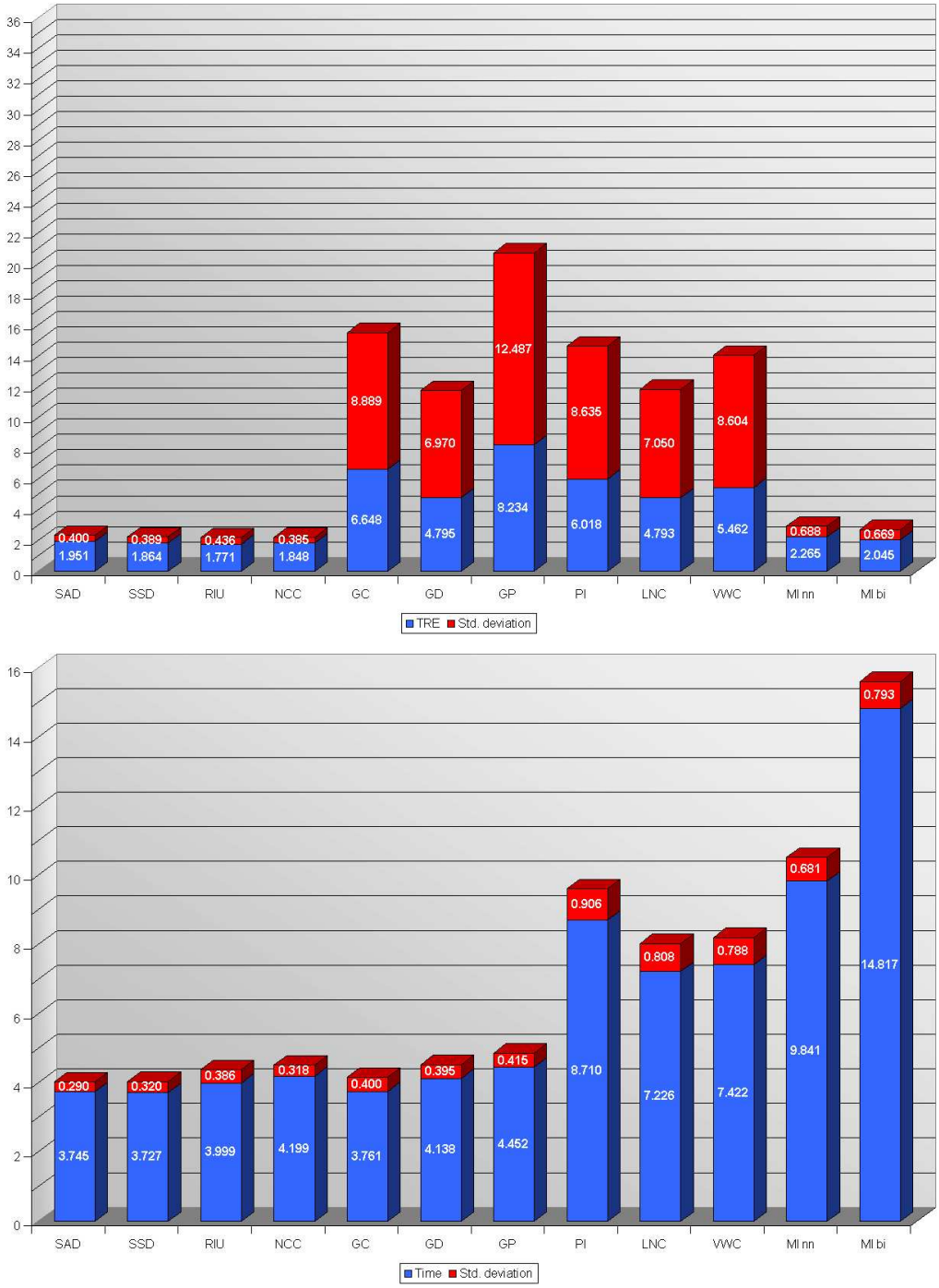


Figure A.19: Different Types of CT Scanners - TRE (mm) and time (sec) for the SSE-based registration, using nearest-neighbor interpolation when rotating the 2D image

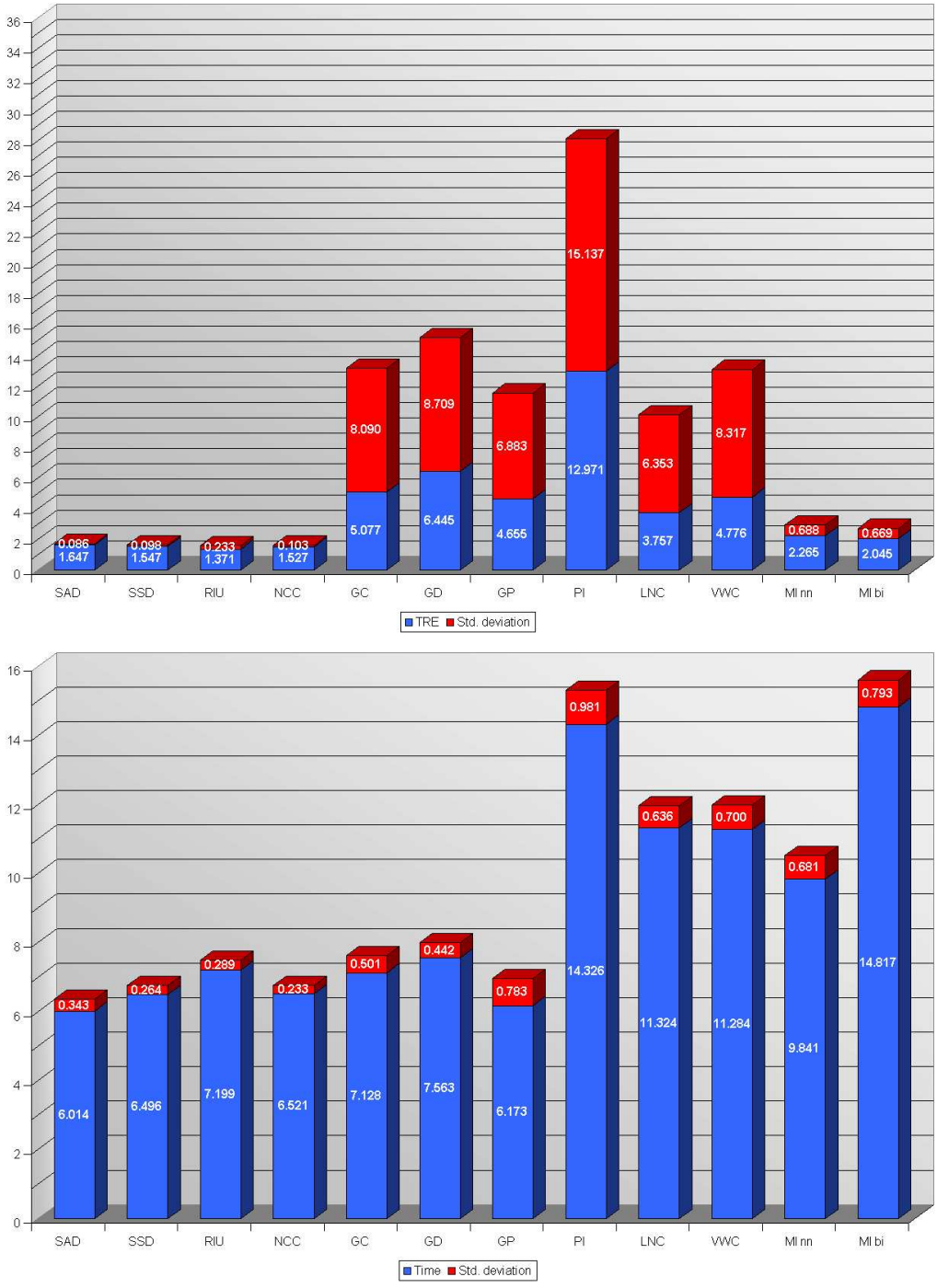


Figure A.20: Different Types of CT Scanners - TRE (mm) and time (sec) for the SSE-based registration, using bilinear interpolation when rotating the 2D image

Bibliography

[1] M. Alp, M. Dujovny, M. Misra, F. Charbel, and J. Ausman, *Head registration techniques for image-guided surgery*, PubMed, (1998).

[2] ATI Technologies Inc., *ATI OpenGL Extension Support*, 2002.

[3] BionicFX, *BionicFX Announces Audio Processing on NVIDIA GPU*. http://www.bionicfx.com/, 2004.

[4] S. Brumme, *The OpenGL Shading Language*, Hasso-Plattner-Institut, Universität Potsdam, 2003.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, *Brook for GPUs: stream computing on graphics hardware*, ACM Transactions on Graphics (TOG), 23 (2004), pp. 777–786.

[6] S. Dowker, J. Elliott, G. Davis, and H. Wassif, *Longitudinal Study of the Three-Dimensional Development of Subsurface Enamel Lesions during in vitro Demineralisation*, Caries Research, (2003).

[7] R. Fernando, M. Harris, M. Wloka, and C. Zeller, *Programming Graphics Hardware*, EuroGraphics, 2004.

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

[9] M. Harris, *GPGPU: General Purpose Computation on GPUs*, EuroGraphics, 2004.

[10] M. J. Harris, *General purpose computation using graphics hardware*. http://www.gpgpu.org.

[11] Intel, *MMX Technology Technical Overview*, 1996.

[12] Intel, *Intel Architecture Software Developer's Manual - Volume 1: Basic Architecture*, 1999.

[13] Intel, *Intel Architecture Software Developer's Manual - Volume 2: Instruction Set Reference*, 1999.

[14] D. B. John Kessenich and R. Rost, *The OpenGL Shading Language*, 3Dlabs Inc., 2003.

[15] D. B. John Kessenrich and R. Rost, *OpenGL 2.0 Shading Language*, 3Dlabs Inc., 1.051 ed., February 2003.

[16] A. Klimovitski, *Using SSE and SSE2: Misconceptions and Reality*, Intel Developer Update Magazine, (2001).

[17] J. Krüger and R. Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*, SIGGRAPH, 2003, pp. 1–9.

[18] D. LaRose, *Iterative X-ray/CT Registration Using Accelerated Volume Rendering*, PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, May 2001.

[19] J. Lengyel, M. Reichert, B. R. Donald, and D. Greenberg, *Real-time robot motion planning using rasterizing computer graphics hardware*, SIGGRAPH, 1990, pp. 327–335.

[20] A. Lowery, J. Strojny, and J. Puleo, *Equipment and Calibration,*, tech. rep., 1996.

[21] J. Maintz and M. Viergever, *A survey of medical image registration*, Medical Image Analysis, 2 (1998), pp. 1–36.

[22] M. McCool, *Shading Language Overview*, SIGGRAPH, SIGGRAPH, 2003.

[23] Microsoft, *MSDN Library*, 2004.

[24] K. Moreland and E. Angel, *The FFT on a GPU*, SIGGRAPH, 2003.

[25] NVIDIA Corporation, *Cg Toolkit Release Notes*, February 2003.

[26] NVIDIA Corporation, *Cg Toolkit User's Manual*, 2003.

[27] NVIDIA Corporation, *CgFX Overview*, February 2003.

[28] NVIDIA Corporation, *The GeForce 6 Series of GPUs: High Performance and Quality for Complex Image Effects*, 2004.

[29] NVIDIA Corporation, *NVIDIA OpenGL Extension Specifications*, May 2004.

[30] V. Oliushin, G. Tigliev, O. Ostreiko, and M. Filatov, *Immunotherapy in Patients with Growing Glioblastomas*, tech. rep., 2001.

[31] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar, *Interactive multi-pass programmable shading*, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, (2000), pp. 425–432.

[32] G. P. Penney, J. Weese, J. A. Little, P. Desmedt, D. L. G. Hill, and D. J. Hawkes, *A Comparison of Similarity Measures for Use in 2-D3-D Medical Image Registration*, in IEEE Transactions on Medical Imaging, IEEE, IEEE, August 1998, pp. 586–595.

[33] Pixar, *The RenderMan Interface*, July 2000.

[34] W. Press, W. Vetterling, and B. Flannery, *Numerical Recipes in C, Second Edition*, CRC Press, 1992, ch. 14.1.

[35] T. Purcell, *Ray Tracing on a Stream Processor*, phd, Stanford University, March 2004.

[36] M. Rumpf and R. Strzodka, *Level segmentation in graphics hardware*, vol. 3, ICIP, 2001, pp. 1103–1106.

[37] SGI, *OpenGL Extension Registry*, 2004.

[38] C. E. Shannon, *The Mathematical Theory of Communications*, Bell System Technical Journal, 27 (1948), pp. 379–423 and 623–656.

[39] A. Sherbondy, M. Houston, and S. Napel, *Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware*, tech. rep., Stanford University, 2003.

[40] P. Viola, *Alignment by Maximization of Mutual Information*, PhD thesis, Massachusetts Institute of Technology, 1995.

[41] P. A. Viola and W. M. W. III, *Alignment by Maximization of Mutual Information*, Tech. Rep. AITR-1548, 1995.

[42] Volume Graphics GmbH, *VGL Software Development Tool*, 2003.

[43] F. Wang, B. Vemuri, M. Rao, and Y. Chen, *Cumulative Residual Entropy, A new Measure of Information and its Application to Image Alignment*, in International Conference on Computer Vision, 2003.

[44] J. Weese, T. Buzug, C. Lorenz, and C. Fassnacht, *An approach to 2D/3D registration of a vertebra in 2D X-ray fluoroscopies with 3D CT images*, in Processding CVRMed/MRCAS, 1997, pp. 119–128.

[45] W. Wein, *Intensity Based Rigid 2D-3D Registration Algorithms for Radiation Therapy*, Master's thesis, TUM, 2003.

[46] D. Weiskopf, M. Hopf, and T. Ertl, *Hardwareaccelerated visualization of time-varying 2d and 3d vector fields by texture advection via programmable per-pixel operations*, VMV, 2001, pp. 439–446.

[47] *Wikipedia*. http://en.wikipedia.org/.

[48] R. Woods, S. Grafton, C. Holmes, S. Cherry, and J. Mazziotta, *Automated image registration: I. General methods and intrasubject, intramodality validation*, Journal of Computer Assisted Tomography, 16 (1992), pp. 620–633.

[49] L. Zollei, *2D-3D Rigid-Body Registration of X-Ray Fluoroscopy and CT Images*, Master's thesis, Massachusetts Institute of Technology, 2003.