Efficient Implementation of Nonrigid Registration Methods on commodity Hardware with CUDA

Diplomarbeit in Technomathematics

submitted
by

Christian Ledig

born 04.06.1984 in Nürnberg

Written at

[1] Chair of Applied Mathematics 2
Department of Mathematics

[2] Pattern Recognition Lab
Department of Computer Science

Friedrich-Alexander University Erlangen-Nuremberg, Germany

in Cooperation with

[3] Siemens Corporate Research, Princeton, NJ, U.S.A.

Advisors:
Christophe Chefd'hotel[3], Ph.D.,
Dr.-Ing. Dieter Hahn[2],
Prof. Dr.-Ing. Joachim Hornegger[2],
Prof. Dr. Günter Leugering[1]

Started: 10 July 2010

Finished: 21 December 2010

Erlangen, den 21. Dezember 2010

Übersicht

Die Vereinigung von medizinischen Bildern, die durch verschiedene Modalitäten oder zu unterschiedlichen Zeitpunkten akquiriert wurden, ist eine weit verbreitete Anwendung in der interventionellen und diagnostischen medizinischen Bildverarbeitung. Der Vorgang diese Bilder in einem gemeinsamen Koordinatensystem anzugleichen wird als Registrierung bezeichnet und ist eine anspruchsvolle Aufgabe. Insbesondere die rechenintensive Rückgewinnung von nicht-rigiden Weichteil-Deformationen ist von besonderem Interesse. Gesteigerte Datenmengen sowie Echtzeitanforderungen, wie sie in interventionellen Applikationen auftreten, erfordern hochperformante Registrierungsmethoden.

In der vorliegenden Arbeit präsentieren wir eine Möglichkeit diesen Anforderungen gerecht zu werden. Unser Ansatz basiert auf der Umsetzung eines gradientenbasierten nicht-rigiden Registrierungsalgorithmus auf der parallelen Programmierarchitektur NVIDIA CUDA. Im Rahmen der Variationsrechnung präsentieren wir ein Registrierungsverfahren, das auf dem Fluss von Diffeomorphismen basiert, in Verbindung mit verschiedenen Abstandsmaßen. Wir zeigen wie dominierende Bausteine dieses Registrierungsvorgangs, Gauß-Filter und Histogramm-Berechnung, effizient auf der GPU realisiert werden können.

Experimentelle Ergebnisse bestätigen eine signifikante Reduktion der Rechenzeiten um bis zu eine Größenordnung verglichen mit existierenden parallelen CPU Implementierungen. Schließlich schlagen wir, mit ersten vielversprechenden Ergebnissen von komplexeren Interpolationsschemata und Regularisierungstechniken als auch von einem neuartigen Abstandsmaß, Ausgangspunkte für zukünftige Forschungsarbeiten vor.

Die Registrierungszeiten, die mit einer effizienten parallelen Implementierung auf NVIDIA CUDA fähigen Grafikkarten erzielt werden können, öffnen die Tür für die Anwendung nicht-rigider Registrierung in zeitkritischen interventionellen Applikationen.

Abstract

The fusion of medical images obtained by different modalities or at different acquisition times is a common task in interventional and diagnostic medical image processing. The process of aligning these images in a common coordinate system is called registration and a challenging task. Especially the computationally expensive recovery of nonrigid soft tissue deformations is of particular interest. Increased data as well as real-time constraints as they arise in interventional applications call for high performance registration techniques.

In this thesis we present an attempt to cope with these challenges by employing a gradient based nonrigid registration algorithm implemented on the parallel programming architecture NVIDIA CUDA. We present a registration approach that is based on flows of diffeomorphisms in combination with different similarity measures in a variational framework. We show how dominating building blocks of this registration pipeline, Gaussian filtering and histogram computation, can be efficiently realized on the GPU.

Experimental results reveal significant decreases in computation times of up to one order compared to an existing parallel CPU implementation. Finally we provide with promising preliminary results of more complex interpolation schemes and regularization techniques as well as of a new similarity measure starting points for future research.

The registration times obtained by an efficient implementation on NVIDIA CUDA capable devices open the door to the application of nonrigid registration methods in a time-critical interventional setting.

Contents

Chapter 1

Introduction

The alignment of two images collected at different times or acquired by different modalities is a fundamental task in a variety of applications in interventional and diagnostic imaging. Often it is the combination of the information contained in different datasets that allows for a reliable diagnosis or an efficient treatment. For example, fusion methods are used to combine functional positron emission tomography (PET) data with morphological computed tomography (CT) images in oncology and cancer staging.

In general, registration is the process of aligning two or more images in a common coordinate system and a challenging task. Dependent on the application the given time to register two given images is short. Very challenging real-time constraints may arise in interventional imaging to provide the surgeon with additional information acquired preoperatively [MO08]. When brain diseases are treated by neurosurgical resection, fast nonrigid registration might be the key to employ intraoperative image guidance [Rui10].

The nonrigid registration process is especially due to its high dimensional parameter space given by the many degrees of freedom a very time consuming, computationally expensive task. However, these costly registration techniques are necessary when results obtained with an efficient rigid registration approach are not satisfying. For example, rigid techniques might not be sufficient to recover brain deformations of up to 20 mm caused by the so-called brain shift phenomenon [Soz02], [Rui10]. As mentioned in [MO08], an accelerated nonrigid registration could also allow for the application of adaptive radiation therapy (ART) [Jos04], where radiation therapy is improved by adapting the treatment plan on daily changes of the patients anatomy.

It is obvious to employ graphics hardware to address the challenge of a fast nonrigid registration efficiently in parallel.

Nevertheless, former approaches based on shader programming were often not satisfying. Computations that were not suitable for shader programming, as for example histogram computation, were often performed on the CPU. This required expensive data transfers between host and device memory, which ate up the performance gained through parallelism.

With NVIDIA CUDA [NVI10c] a new general purpose parallel computing architecture was introduced in 2006. Its shared memory capability enables efficient solutions on the GPU for more complex problems.

Up to now, only a small number of studies have been conducted on the realization of deformable registration techniques on GPUs [MO08]. In [Rui09] NVIDIA CUDA is used to tackle rigid registration problems arising in minimally invasive neuroangiography interventions. Various demons deformable image registration algorithms were implemented using CUDA and evaluated on ART relevant registration tasks in [Gu10]. Other recent approaches to recover nonrigid deformations modeled by B-Splines can be found in [Ans09] and [Rui10].

In this work we will investigate how a complete nonrigid registration pipeline that is based on the *flows of diffeomorphisms* [Che02] can be accelerated by using NVIDIA CUDA.

We will first provide a rough introduction to the problem of nonrigid registration and the programming architecture of NVIDIA CUDA. In Chapter 2 we will give a short overview over existing nonrigid registration approaches. We will present a certain gradient based matching algorithm together with different similarity criterions within a variational framework in Chapter 3. We present theoretical background and conduct research on recursive filtering and joint histogram computation on GPUs in Chapter 4. Both tasks turn out to be particularly relevant for the performance of the presented registration pipeline. After providing a detailed evaluation of the performance gain obtained by our implementations in Chapter 5, we give perspectives on how to extend this work in different directions in Chapter 6. We conclude in Chapter 7.

1.1 The Nonrigid Registration Problem

Image registration can be described as the process of estimating a geometric transformation that aligns a *moving* image to a fixed *reference* image. This task can be formulated as an optimization problem with respect to a given cost function which measures the quality of the alignment. The selection of the transformation model and the cost function is application dependent and has a

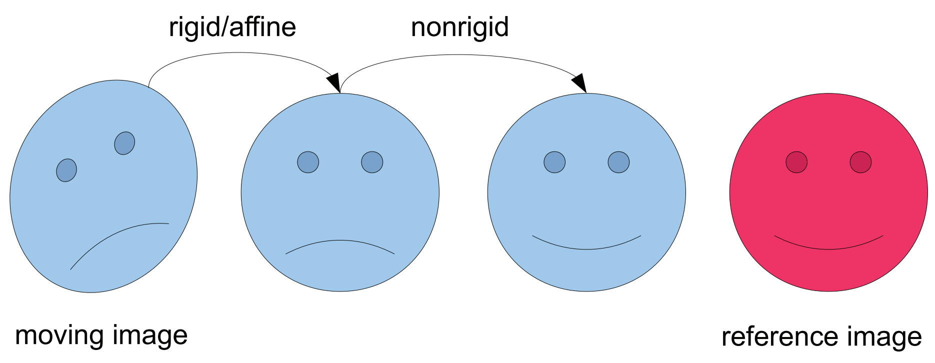


Figure 1.1: Schematic registration of a sad face (moving) to a happy face (reference). Rigid and affine registration (rotation, shear) is followed by nonrigid registration.

strong impact on the registration results as well as the computation time. Historically image registration is viewed as being *rigid*, meaning that images can simply be rotated and translated with respect to one another to obtain a good matching, or *nonrigid* [Cru04]. In the latter case a deformation or stretching of local structures is possible to recover misalignments caused by biological differences or non-uniform image acquisitions. Rigid transformations can be extended to arbitrary affine transformations by allowing scaling and shearing in addition to rotations and translations. An idea of what kind of transformations can be recovered with either rigid or nonrigid registration methods is given with Figure 1.1. In this artificial example the sad face (moving image) is rotated by a rigid and sheared by an affine transformation to recover the orientation and the circular shape of the face in a first step. The final match with the happy face (reference image) is obtained by a nonrigid transformation, which is recovering the bending of the mouth. Rigid approaches are often suitable and sufficient for the registration of images containing rigid structures as bones, where the preservation of distances and angles might be a desired property. However, most of the human body and especially soft-tissue movement does not conform to rigid transformations [Haw05]. The registration of moving organs, which can be found in heart or lung scans, call for more complex nonrigid transformations. Also the most challenging registration tasks, which arise during interventional imaging in surgery [Fer02] or modeling changes in neuroanatomy, crucially depend on nonrigid registration.

In this work we focus on nonrigid transformations, which allow for arbitrary deformations of the moving image. A further overview and introduction to nonrigid registration methods is given in Chapter 2.

1.2 GPU Programming using NVIDIA CUDA

The programmable graphics processing unit (GPU) is a highly parallel, multithreaded, many-core processor, which is specialized for compute intensive computations [NVI10c]. GPUs are especially well-suited for problems that can be solved in parallel, by executing the same program on different data elements. By trading caching and flow control for data processing, GPUs are better suited for compute intensive tasks than CPUs are. Whenever the same program can be executed independently on each data element, there is no major need for flow control. Also the increased memory access latency because of missing data caches can be hidden with high arithmetic intensity. By mapping all data elements to parallel processing threads, a huge variety of applications that require to process large data sets can take advantages of this data-parallel programming model.

With the introduction of CUDA, NVIDIA presented a general purpose parallel computing architecture, with a new parallel programming model as well as instruction set architecture. CUDA enables the efficient solution of many complex problems by employing NVIDIA GPUs.

With the availability of *C for CUDA* there exists a set of language extensions to the high-level programming language C that makes the development of CUDA programs very accessible.

In [NVI10c] NVIDIA describes three key abstractions that are available to the programmer as a set of language extensions. The hierarchy of thread groups, shared memories and barrier synchronization.

These abstractions enable the programmer to partition the problem in coarse-grained sub-problems which can be solved independently by blocks of threads. These sub-problems can now be solved by a number of threads within each block. The availability of 16 kB shared memory, which is shared among the threads within a certain block, plays a major role. Also the availability of synchronization routines within each block allows for more flexibility. The challenge to develop software that reasonably scales with an increasing number of processor cores is addressed by organizing a group of threads in so-called blocks. These blocks can be executed completely independent. Merely the runtime system needs to know the physical processor count to schedule the blocks according to the available cores. This concept is illustrated in Figure 1.2.

The language extensions provided with *C for CUDA* allow the definition of so-called kernels, that are similar to regular C functions but executed in parallel by different CUDA threads. Through the built-in *threadIdx* and *threadDim* variables every thread is labeled with a certain

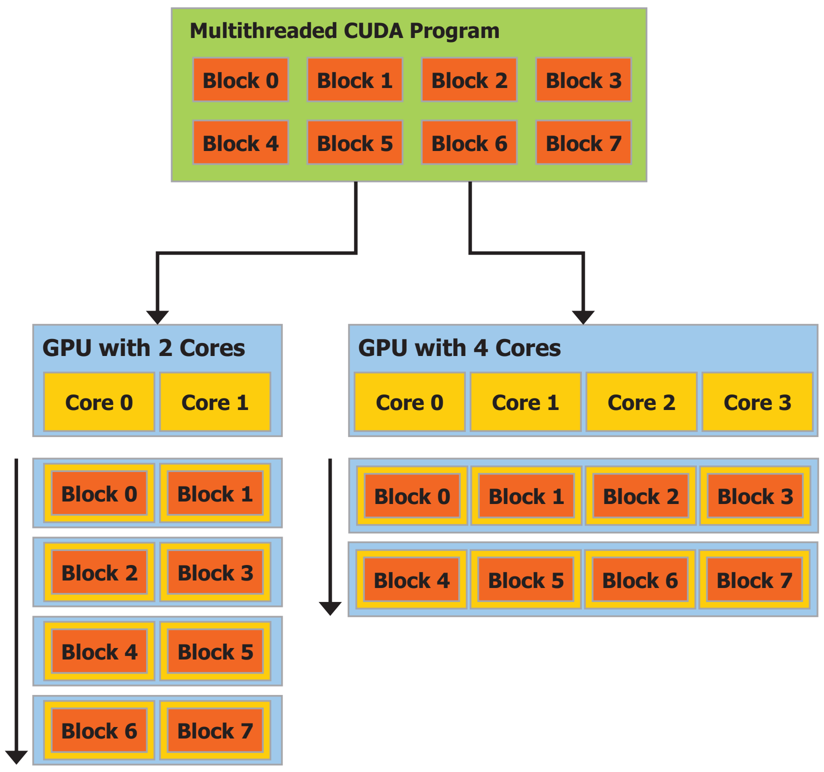


Figure 1.2: Multithreaded programs are partitioned into blocks of threads. These blocks are executed independently of each other. Thus the execution time scales theoretically with the number of available cores on the GPU. [NVI10c]

thread ID within each thread block and provided with the number of threads per dimension. This allows to distinguish between threads and therefore define the data elements, that will be processed by one certain thread. The fact that these variables are three component vectors makes programmer's life easy, since it allows for a natural way to invoke computation across different data elements. Caused by hardware limitations, current GPUs allow for up to 512 threads per block. Nevertheless, by executing the kernel with several thread blocks, the number of total threads can greatly exceed this number. The blocks themselves are organized into a up to two-dimensional *grid* of thread blocks, which is shown in Figure 1.3. In practice the block (number of threads) as well as the grid (number of thread blocks) configuration is usually derived from the size of the processed data.

The number of total blocks and the index of the block a certain thread belongs to is provided by the built-in variables *blockIdx* and *blockDim*. The fact that thread blocks need to be executed independently allows to schedule them in any order and to write code that scales with the number

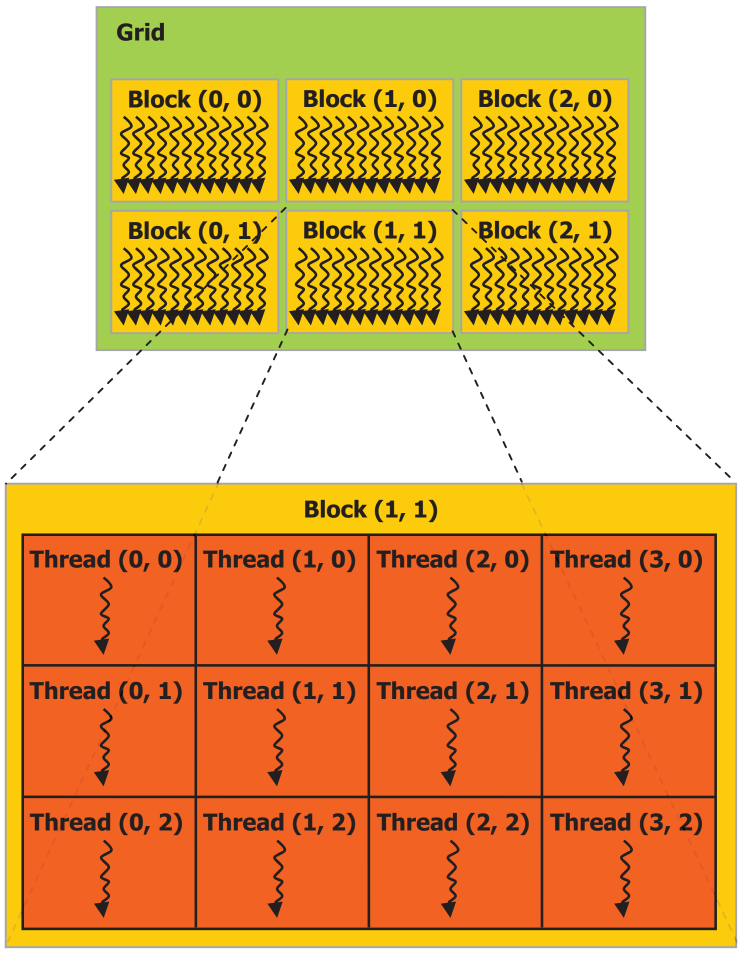


Figure 1.3: Schematical layout of a grid containing different thread blocks and a certain block containing a number of threads. [NVI10c]

of available cores. Threads within one block can be synchronized to coordinate memory accesses and interact by sharing up to 16 kB shared memory. It is emphasized that especially the availability of shared memory and synchronization routines are the novelties, which allow for general purpose programming on GPUs.

Threads themselves can access memory in different ways. While up to several GB of rather slow global memory is available to all threads, shared memory, shared among threads belonging to the same block, is fast but with 16 kB per block limited. Next to this every thread has its own but also quite slow local memory. In terms of performance global or local memory accesses should be replaced by fast shared memory whenever possible. The different memory spaces are

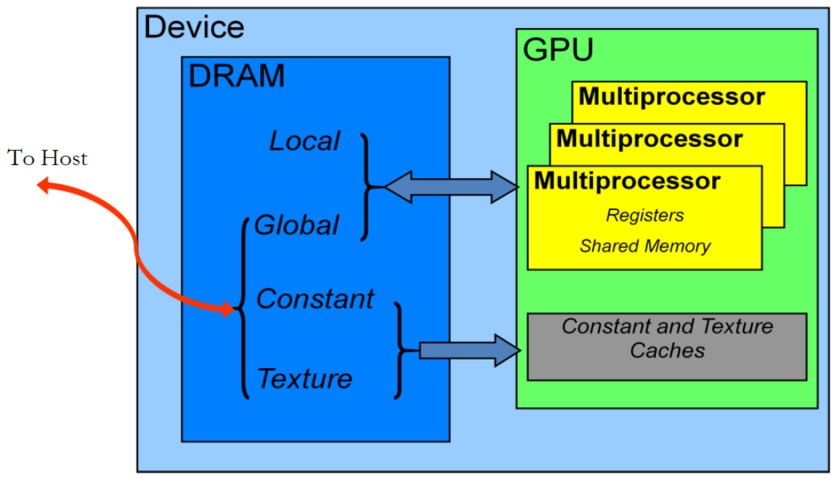


Figure 1.4: Different memory spaces on a CUDA device. [NVI10b]

illustrated in Figure 1.4, where comparatively slow spaces are located on the DRAM while fast registers and shared memory are located on-chip.

With texture and constant memory there are two additional read-only memory spaces available, which are accessed by all threads and located on the DRAM. However, since the texture memory space is cached and allows for hard-wired linear interpolation it is often preferred to global memory.

CUDA programs run on two different physical devices. On the one hand sequential computations are performed on the CPU (host system). On the other hand computationally expensive and data intensive calculations can be done efficiently in parallel on the GPU (device system). With the host and device memory, both systems have separate memory spaces which are managed by the program. This means that a program is required to (de-)allocate device memory as well as perform data transfers between host and device memory. This concept is shown schematically in Figure 1.5.

For a more detailed introduction as well as technical details we refer to the NVIDIA CUDA Programming Guide [NVI10c] and the NVIDIA CUDA C Programming, Best Practices Guide [NVI10b].

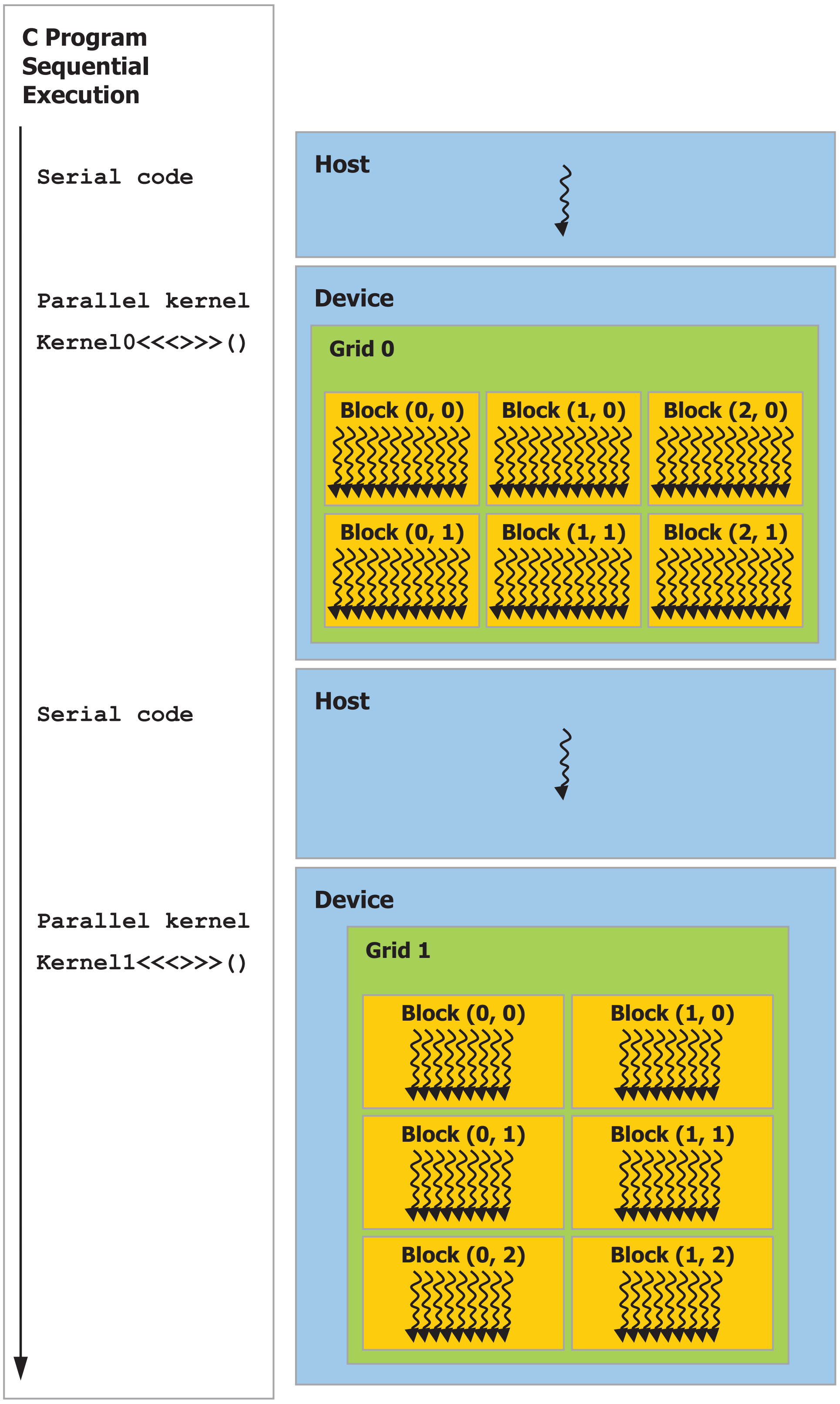


Figure 1.5: Heterogeneous programming using CUDA: Serial code executes on the host while parallel code executes on the device. [NVI10c]

Chapter 2

Overview of Existing Nonrigid Registration Methods

As already mentioned in Section 1.1 a main classification of registration methods can be done by distinguishing between *rigid* or *affine* and *nonrigid* transformations.
In our work, we are interested in nonrigid registration and will thus provide primarily an overview of commonly used nonrigid registration approaches in this Chapter. However, we mention that nonrigid registration is usually performed on rigidly preregistered images [Hil01].

According to [Cru04] the nonrigid registration process can be divided into three blocks:

1. the *similarity measures* which are criterions for the quality of the alignment and can be classified in intensity and geometric approaches;
2. the *transformation model* which specifies feasible changes to the moving image in order to match the reference image;
3. the *optimization process* that finds a valid transformation that optimizes the matching criterion (similarity measure).

A schematical overview of the whole registration pipeline and its components is shown in Figure 2.1. Determined by a certain energy and transformation model, the optimization algorithm usually iteratively updates the transformation. In every step this updated transformation is then applied to the moving image. For this warped (transformed) moving image and the reference image a similarity criterion is evaluated and another iteration can be performed. This basic scheme is repeated until a defined condition, as for example an iteration count, is met.

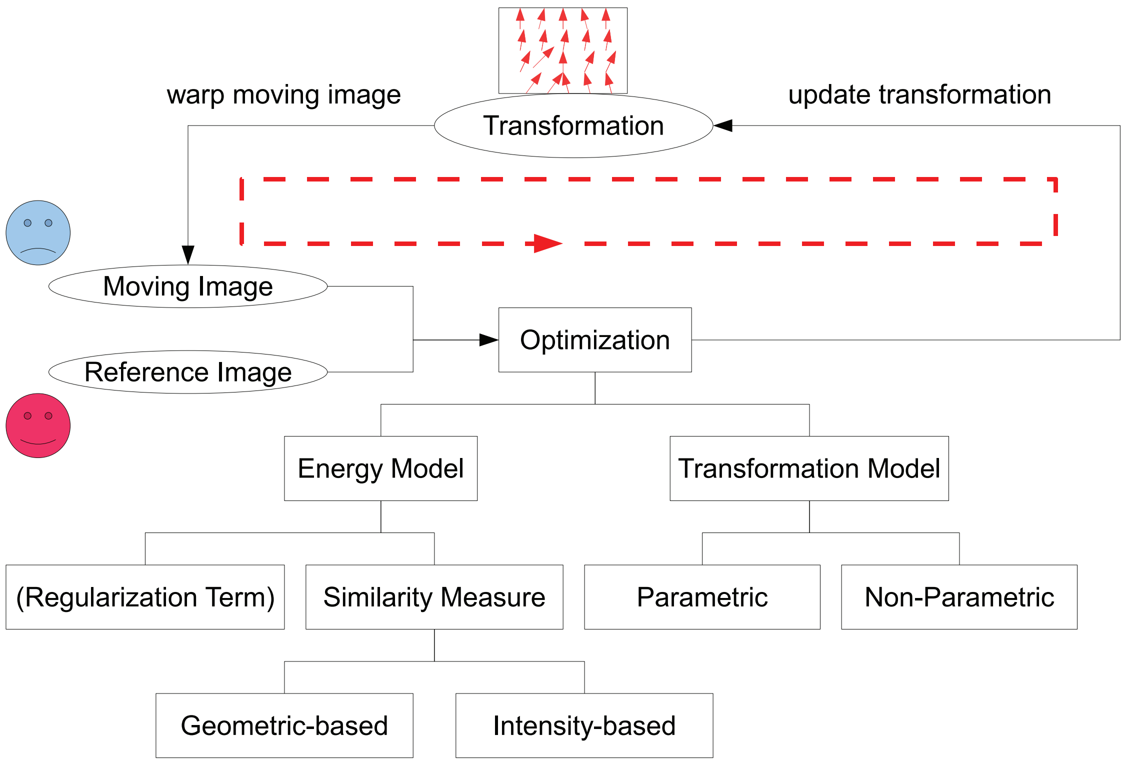


Figure 2.1: Schematic overview of a nonrigid registration pipeline and its components.

Research in the field of nonrigid registration is conducted for more than two decades. The fact that there is no universal choice of similarity measure, transformation model and optimizer that provides satisfying results for any application lead to a huge variety of approaches. Which strategy is best is dramatically dependent on the application and needs to be considered for every individual case. The demand of presenting a complete and technically detailed overview of all different concepts would be condemned to fail within the scope of this document. Therefore we restrict this section to be a rough snapshot of commonly used approaches, while providing references to literature containing further and technically more extensive descriptions.

In the following we will focus on different similarity measures, transformation models and optimization methods.

2.1 Geometry- and Intensity-based Methods

As already mentioned the registration process consists of finding a geometric mapping which transforms the moving image to be most similar to the reference image. This similarity is measured by cost functions. A proper design of these similarity measures is crucial for an efficient and accurate registration. For example the shape (e.g. smoothness, existence of local extrema) of the cost function is substantial for an efficient optimization. Dependent on the application there were proposed a variety of measurements to determine the similarity of two images.

We can distinguish two main concepts, which are usually called landmark- or geometry-based and intensity-based methods [Cru04], [Mod04].

Geometry-based

Geometry-based approaches are based on identifiable anatomical elements or fiducial markers [Mod04] in each image. Usually functionally important surfaces, curves or point landmarks are of interest. Once these features are selected, either automatically or by an expert [Mod04], the goal of the registration process is to find a transformation so that their counterparts in the reference image are optimally matched [Cru04]. The use of landmarks, based on structural information, ensures the anatomical relevance of the found transformation. An overview of surface registration techniques applied to anatomical surfaces can be found in [Aud00]. Even though there are approaches to identify the landmarks automatically this process is in general not automated [Mod04]. In fact the reproducible and consistent location of landmarks might be even for experts a challenging task.

Intensity-based

Intensity-based approaches do not depend on previously located geometric structures. Instead of matching features, intensity patterns are matched by employing mathematical or statistical criteria [Cru04]. However, no anatomical knowledge is taken into account.

The simplest representative of this class of measurements is the sum of squared differences (SSD). It is applicable if the registered images are except Gaussian noise identical [Cru04]. To avoid the quadratic influence of outliers the sum of absolute differences (SAD) was proposed. Next to SSD and SAD the correlation coefficient or also called cross correlation (CC) is suitable for mono-modal image registration. This more complex similarity measure allows for a linear relationship of corresponding intensities.
For multi-modal registration tasks the correlation ratio (CR) [Roc98] and mutual information (MI) [Mae97],[Vio97] was proposed. While CR weakens the linear relationship assumed with CC to a functional relationship, MI is a concept borrowed from information theory which is

based on a probabilistic relationship of corresponding voxel intensities. A normalized version of MI, called normalized mutual information (NMI), was introduced in [Stu99]. It tends to be more robust to variations of the overlap region between images.

Our registration method should be automatic and merely depend on information acquired by the images themselves. Therefore we will solely employ intensity-based similarity measures. A more detailed and formal introduction of SSD, MI and CC can be found in Section 3.2.

2.2 Transformation Models

We follow a common differentiation, also used in [Mod04], by discriminating between parametric and nonparametric transformation models.

Parametric Registration

Rigid as well as affine transformations can obviously be described for the whole image by parameters. For a 3D image any rigid transformation is described by 6 parameters (3 translations + 3 rotations). Taking also shearing and scaling into account, an affine transformation is described by 12 parameters (6 rigid + 3 scalings + 3 shears) respectively.
Next to this also nonrigid transformations can be described in a parametric way by representing the transformation with a set of basis functions and the corresponding coefficients. The registration problem is then tackled by optimizing these coefficients. Commonly used basis functions are the so-called “thin-plate” splines (TPS) [Boo89] and B-splines introduced by Rueckert et al. [Rue99], [Rue06]. If TPS are used, the transformation depends globally on every control point that belongs to the spline. This has the significant disadvantage that the computational cost for changing a single control point increases with the number of control points [Cru04]. In order to cope with likely and frequent occurring landmark extraction errors, approximating TPS were proposed in [Roh01]. Compared to TPS, B-splines have local support which means that the transformation is only affected in the neighborhood of a perturbed point. Especially because of the resulting computational efficiency and the general applicability B-spline based registration methods got popular [Cru04]. Transformations based on B-splines can be modeled by an equidistant grid of control points which are the spline coefficients. A reduced control point spacing (increased number of control points) allows for a better description of local deformations [Rue99]. Applications of B-spline registration are for example atlas construction [Bha04] or the registration of breast images [Rue99].

Nonparametric Registration

Compared to the transformation models introduced so far, nonparametric models are not based on a certain set of basis functions, landmarks or control points. This allows for more freedom while modeling local soft tissue deformations. Instead of restricting the transformation by a certain number of parameters, nonparametric transformations are described by so-called deformation or displacement fields. These fields record vectors that describe the displacement for each voxel in the moving image to obtain a certain alignment with the corresponding location in the reference image [Hil01]. However, the freedom given by this transformation model comes with two major problems that need to be addressed. First of all, with this kind of transformation it is from scratch not guaranteed that no additional cracks or foldings of the tissue are introduced [Mod04]. Thus it is no longer sufficient to rate a found transformation by the obtained visual registration result. In fact the deformation itself needs to be validated to ensure a physically meaningful transformation [Hah09]. Furthermore the minimization of some suitable similarity measure is because of missing uniqueness and convexity an ill-posed problem [Mod04]. Especially the fact that the solution of a registration problem does not continuously depend on the image data makes numerical computation difficult [Cla06].

There are two main approaches to tackle these challenges. The first and more common method is to add a regularization term to the cost function or also called energy functional, which is described by the similarity measure. With this additional penalty function ("smoother") both previously stated problems (meaningfulness of the transformation, ill-posedness) can be addressed. It also allows to distinguish between more or less likely transformations [Mod04]. As described in [Cla06] there are different ways to incorporate this smoother. Next to the common way of adding a with some parameter $\alpha \in \mathbb{R}^+$ weighted smoother to the cost function, which corresponds to the Tikhonov regularization, there were iterative relaxation methods that are adapting α proposed in [Hen02]. Commonly used functionals employed for this regularization are Dirichlet, elastic, fluidal, curvature or higher-order functionals [Cla06], [Hah09]. While linear elastic models are restricted to small and local deformations, fluidal models allow for larger deformations while having an increased likelihood of misregistrations [Mod04], [Cru04]. Curvature based regularization does not penalize affine transformations. This makes rigid preregistration redundant and curvature based regularization particularly interesting for medical image registration [Fis03], [Hah09].

In [Mod04] and also [Cla06] a variety of smoothers are summarized and introduced in more detail.

Another possibility to incorporate regularization is based on gradient flow approaches [Her01a], [Cla06]. With this method the additive regularization is substituted by a spatial smoothing of the gradient fields which occur in a gradient descent optimization [Her01a]. This approach is related to the so-called demon's algorithm [Thi98]. We refer to [Her01a], [Cru04] and [Cla06] for further remarks on flow algorithms and alternative regularizers.

2.3 Optimization Process

The goal within the optimization process is to find a feasible transformation that maximizes the image similarity. Picking the right optimizer is a delicate challenge. It requires next to a profound knowledge of numerical analysis a good understanding of the registration problem [Cla06]. An often encountered problem of many optimizers is the "local minima" problem. This means that even if a transformation which results in a good similarity of the images is found, it is not ensured that it is really the best. As already mentioned also the physical meaningfulness of a found deformation might be relevant. These factors need to be ensured by regularization and the design of the optimizer. The solution of registration problems based on nonparametric models often breaks down to the numerical solution of partial differential equations (PDEs). Dependent on the resulting cost function, given computation time and required registration accuracy the most appropriate optimizer can be picked. A good overview of numerical methods to solve different nonrigid registration problems can be found in [Mod04] and [Cla06]. General considerations for the optimization of cost functions, that arise when nonparametric models are employed, are provided in [Cru04].

For a more detailed overview of registration methods we refer to [Hil01], [Zit03], [Cru04], [Mod04], [Cla06] and [Hah09].

In the following we will present a certain nonparametric nonrigid registration approach.

Chapter 3

A Matching Algorithm

In this chapter we introduce a variational framework and provide variational gradients for three different intensity based similarity criterions to employ a gradient based optimization method. As optimizer for our nonparametric nonrigid registration method we present a flow algorithm called *flows of diffeomorphisms* [Che02]. We chose this algorithm because it turns out that the arising computation tasks, especially the regularization, can be efficiently parallelized. Also the availability of an optimized parallel CPU implementation allows for a representative evaluation and classification of the results achieved by using NVIDIA CUDA.

3.1 Notation

We introduce some notations and assumptions which are used throughout this chapter. The notation is a combination, we felt most comfortable with, of the notation used in [Her01b], [Her02] and [Che02].
In general, we consider image matching problems for images with scalar intensities on a bounded domain $\Omega \subset \mathbb{R}^n$. Even if n is for the following developments arbitrary, we are mainly concerned about 3D matching problems where accordingly n will be equal to 3.

The challenge is to find a deformation $\phi : \Omega \mapsto \Omega$ which aligns a moving or template image $g : \Omega \mapsto \mathbb{R}$ to a reference or also called target image $f : \Omega \mapsto \mathbb{R}$. In the following we restrict ϕ to diffeomorphisms[1] that belong to a Hilbert space V. In this space the usual scalar product is defined as $\langle \phi, h \rangle_V = \int_\Omega \phi(x) \cdot h(x)\, dx$ for $h \in V$.

Diffeomorphisms represent smooth and invertible transformations, which ensure that every point in the reference image has a corresponding point in the moving image [Rue06]. We assume

[1]A bijective map ϕ is a *diffeomorphism* if ϕ and ϕ^{-1} are continuously differentiable [Che02].

that ϕ is a diffeomorphism since we expect similar structures in reference and moving image. The deformation ϕ acts on the moving image g by $g \circ \phi$. Next to ϕ, we will often consider the displacement $u : \Omega \mapsto \Omega$, which is related to ϕ via the decomposition $\phi = id + u$. Therefore we have pointwise $g(\phi(x)) = g(x + u(x)) = g_u(x)$.

By defining a cost functional $\mathcal{J} : V \mapsto \mathbb{R}$, which can be calculated for each similarity criterion, we can compare the alignment of f and g_u. In order to optimize the quality of this transformation, we are looking for a minimal solution $\hat{u}$ to the problem:

$$\hat{u} = \operatorname*{argmin}_{u \in V} \mathcal{J}[u] \tag{3.1}$$

In this sense $\mathcal{J}$ measures the dissimilarity of the images, which is minimized. It turns out that the gradient $\nabla_u \mathcal{J}$ of $\mathcal{J}$ with respect to u has the generic form:

$$\nabla_u \mathcal{J}(x) = L_u(f(x), g(x + u(x)), x) \nabla g(x + u(x)) \tag{3.2}$$

For a given displacement u, the function $L : \mathbb{R} \times \mathbb{R} \times \Omega \mapsto \mathbb{R}$ is called local intensity comparison function if $L(i_1, i_2, u, x)$ depends on x and global intensity comparison function otherwise.

At this point we recall the first variation, defined as the Gateaux derivative, of $\mathcal{J}$:

$$\delta_h \mathcal{J}[u] = \lim_{\epsilon \to 0} \frac{\mathcal{J}[u + \epsilon h] - \mathcal{J}[u]}{\epsilon} = \left. \frac{\partial \mathcal{J}[u + \epsilon h]}{\partial \epsilon} \right|_{\epsilon = 0} \tag{3.3}$$

If $\mathcal{J}$ has a local extremum for some $u \in V$, its first variation must vanish at u for all $h \in V$.

$$\forall h \in V, \delta_h \mathcal{J}[u] = 0 \tag{3.4}$$

Since the gradient $\nabla_u \mathcal{J}$ of the functional $\mathcal{J}$ is defined as $\delta_h \mathcal{J}[u] = \langle \nabla_u \mathcal{J}, h \rangle_V$, the necessary condition of optimality described in Eq. 3.4 is equivalent to

$$\nabla_u \mathcal{J} = 0 \tag{3.5}$$

and often called Euler-Lagrange equation.

In order to employ a similarity criterion for a gradient based matching algorithm, we need to calculate the intensity comparison function, which is related to the gradient by Eq. 3.2. In the following we will compute the variational gradients of three different similarity measures and provide the according intensity comparison functions together with the Euler-Lagrange equations.

3.2 Similarity Measures

In this section we will introduce different intensity-based similarity measures. With the Sum of Squared Difference (SSD), a quite simple but efficient similarity measure for mono-modal image registration is presented. However, when images can no longer be compared by the difference of their intensity values, we need to employ more complex similarity criterions. This is usually necessary whenever images are acquired by different modalities or corrupted by a non-uniform bias as it is often the case with magnetic resonance images. For this purpose we also present with Mutual Information (MI) and Cross Correlation (CC) two statistical similarity measures. These criterions are based on a density estimation of the probability density functions (pdfs) p^f, p_u^g and the joint probability density function (jpdf) $p_u^{f,g}$ of the two images f and g_u. We will now present a common approach to obtain a continuous estimation of these density functions.

Density Estimation

We associate the intensity values of the reference image f and the moving image g displaced by u with the two random variables X^f and X_u^g and the corresponding pdfs p^f and p_u^g. To be able to evaluate similarity measures based on these statistical parameters, we employ a nonparametric Parzen estimator to obtain a continuous estimation of the pdfs and jpdf [Her02, Par62]. More precisely, we picked an estimator based on a normalized Gaussian kernel with standard deviation $\gamma \in \mathbb{R}^+$. For a given displacement field u the jpdf of f and g_u is estimated as

$$p_u^{f,g}(i_1, i_2) = \frac{1}{|\Omega|} \int_\Omega G_\gamma(f(x) - i_1, g(x + u(x)) - i_2)dx \tag{3.6}$$

where the marginals p^f and p_u^g can be immediately expressed in terms of $p_u^{f,g}$ by

$$p^f(i_1) = \int_{\mathbb{R}} p_u^{f,g}(i_1, i_2)di_2 \tag{3.7}$$

and

$$p_u^g(i_2) = \int_{\mathbb{R}} p_u^{f,g}(i_1, i_2)di_1\ . \tag{3.8}$$

We calculate the first variation of the estimator $p_u^{f,g}$ with respect to u to

$$\begin{aligned}\delta_h p_u^{f,g}(i_1, i_2) =& \frac{1}{|\Omega|} \int_\Omega \partial_2 G_\gamma(f(x) - i_1, g(x + u(x)) - i_2) \\ & \nabla g(x + u(x)) \cdot h(x)\, dx\end{aligned} \tag{3.9}$$

where ∂_2 is the differential operator with respect to the second argument of G_γ.

We can also make this estimator local and allow for non-stationarities in the relation between intensities [Her02]. By employing another normalized Gaussian of standard deviation β we restrict this estimator to a neighborhood of each point $x_0 \in \Omega$. Our local estimator for a given point x_0 is then given as

$$p_u^{f,g}(i_1, i_2, x_0) = \frac{1}{\mathcal{G}_\beta(x_0)} \int_\Omega G_\gamma(f(x) - i_1, g(x + u(x)) - i_2) G_\beta(x - x_0)\, dx \tag{3.10}$$

where

$$\mathcal{G}_\beta(x_0) = \int_\Omega G_\beta(x - x_0)\, dx \tag{3.11}$$

with first variation

$$\begin{aligned} \delta_h p_u^{f,g}(i_1, i_2, x_0) =& \frac{1}{\mathcal{G}_\beta(x_0)} \int_\Omega \partial_2 G_\gamma(f(x) - i_1, g(x + u(x)) - i_2) G_\beta(x - x_0) \\ & \nabla g(x + u(x)) \cdot h(x)\, dx\ . \end{aligned} \tag{3.12}$$

For more details on the presented density estimation we refer to [Her02] and [Par62].

3.2.1 Sum of Squared Differences

For the registration of two datasets acquired by the same modality SSD which is defined as

$$\mathcal{J}_{\text{SSD}}[u] = \frac{1}{|\Omega|} \int_\Omega (f(x) - g(x + u(x)))^2\, dx \tag{3.13}$$

is often the similarity measure of choice.

First Variation

The variational gradient of $\mathcal{J}_{\text{SSD}}[u]$ is easily computed by

$$\begin{aligned} \frac{\partial \mathcal{J}_{\text{SSD}}[u + \epsilon h]}{\partial \epsilon} =& \frac{1}{|\Omega|} \int_\Omega \frac{\partial}{\partial \epsilon} \left[(f(x) - g(x + u(x)))^2 \right]\, dx \\ =& \frac{1}{|\Omega|} \int_\Omega 2(f(x) - g(x + u(x) + \epsilon h(x))) \\ & \nabla g(x + u(x) + \epsilon h(x)) \cdot h(x)\, dx\ . \end{aligned} \tag{3.14}$$

With $\epsilon = 0$ we calculate the first variation of $\mathcal{J}_{\text{SSD}}$ to

$$\delta_h \mathcal{J}_{\text{SSD}}[u] = \frac{2}{|\Omega|} \int_\Omega (f(x) - g(x + u(x)))\nabla g(x + u(x)) \cdot h(x)\, dx \,. \tag{3.15}$$

By defining a global intensity comparison function $L_u^{\text{SSD}} : \mathbb{R}^2 \mapsto \mathbb{R}$ as

$$L_u^{\text{SSD}}(i_1, i_2) = \frac{2}{|\Omega|}(i_1 - i_2) \tag{3.16}$$

we obtain the Euler-Lagrange Equation for SSD in the form

$$\nabla_u \mathcal{J}_{\text{SSD}} = 0 \quad \text{with} \quad \nabla_u \mathcal{J}_{\text{SSD}}(x) = L_u^{\text{SSD}}(f(x), g(x + u(x)), u)\nabla g(x + u(x)) \,. \tag{3.17}$$

3.2.2 Mutual Information

For multimodal image registration tasks, cost functions based on statistical similarity measures such as mutual information (MI) [Mae97] have been proposed. This metric is derived from information theory, and typically computed from joint histograms of image intensity pairs.

The choice of a suitable similarity measure to optimize is a critical part in image registration. Since many metrics suffer from data-dependence, robustness or parameter dependence, MI which overcomes these issues was proposed in [Mae97]. Concerning the data, no a priori knowledge is required when using MI. MI is typically defined as:

$$MI(X_u^g, X^f) = H(X_u^g) + H(X^f) - H(X_u^g, X^f) \tag{3.18}$$

with marginal entropy: $H(X) = -\int_\mathbb{R} p_X(x) \log p_X(x)\, dx$ and
joint entropy : $H(X,Y) = -\int_{\mathbb{R}^2} p_{XY}(x,y) \log p_{XY}(x,y)\, dx\, dy$

This metric captures the statistical dependence between the intensity values of an image pair and is maximal in the case of geometrically aligned images [Mae97].

MI may also be written as the Kullback-Leibler divergence [Kul51, Mae97]

$$MI_u^{f,g} = KL(p_u^{f,g}, p^f p_u^g) = \int_{\mathbb{R}^2} p_u^{f,g}(i_1, i_2) \underbrace{\log \frac{p_u^{f,g}(i_1, i_2)}{p^f(i_1) p_u^g(i_2)}}_{E^{\text{MI}}(i_1, i_2, u)} di_1\, di_2 \tag{3.19}$$

of the product $p^f p_u^g$ of the marginal distributions from the joint distribution $p_u^{f,g}$. This formulation describes the expected value of the function $E^{\text{MI}}(i_1, i_2, u)$ [Her02].

To evaluate Eq. 3.18 an estimation of the marginal probability distributions p^f and p^g_u and the joint probability distribution $p^{f,g}_u$ is necessary. In practice this will be done by computing a joint histogram of the intensity pairs of the reference and warped moving image. For our analytical calculations we will employ the Parzen estimators presented in Eq. 3.6, Eq. 3.7 and Eq. 3.8.

First Variation

In order to solve a minimization problem, we consider the cost functional $\mathcal{J}_{\text{MI}}[u] = -MI^{f,g}_u$.

In the following we will present an explicit calculation of the first variation of this functional $\mathcal{J}_{\text{MI}}[u]$, which was introduced in [Her01a] and [Her02].

We calculate

$$\begin{aligned}
\frac{\partial \mathcal{J}_{\text{MI}}[u+\epsilon h]}{\partial \epsilon} &= -\int_{\mathbb{R}^2} \frac{\partial}{\partial \epsilon}\left[p^{f,g}_{u+\epsilon h}(i_1,i_2) \log \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^f(i_1)p^g_{u+\epsilon h}(i_2)}\right] di_1\, di_2 \\
&= -\int_{\mathbb{R}^2} \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon} \log \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^f(i_1)p^g_{u+\epsilon h}(i_2)} + p^{f,g}_{u+\epsilon h}(i_1,i_2) \frac{p^f(i_1)p^g_{u+\epsilon h}(i_2)}{p^{f,g}_{u+\epsilon h}(i_1,i_2)} \\
&\quad \left(\frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon} \frac{p^f(i_1)p^g_{u+\epsilon h}(i_2)}{p^f(i_1)^2 p^g_{u+\epsilon h}(i_2)^2} - \frac{\partial p^g_{u+\epsilon h}(i_2)}{\partial \epsilon} \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)p^f(i_1)}{p^f(i_1)^2 p^g_{u+\epsilon h}(i_2)^2} \right) di_1\, di_2 \\
&= -\int_{\mathbb{R}^2} \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon} \log \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^f(i_1)p^g_{u+\epsilon h}(i_2)} \\
&\quad + \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon} - \frac{\partial p^g_{u+\epsilon h}(i_2)}{\partial \epsilon} \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^g_{u+\epsilon h}(i_2)} \, di_1\, di_2 \\
&= -\int_{\mathbb{R}^2} \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon} \left(\log \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^f(i_1)p^g_{u+\epsilon h}(i_2)} + 1 \right) di_1\, di_2 \\
&\quad + \underbrace{\int_{\mathbb{R}^2} \frac{\partial p^g_{u+\epsilon h}(i_2)}{\partial \epsilon} \frac{p^{f,g}_{u+\epsilon h}(i_1,i_2)}{p^g_{u+\epsilon h}(i_2)} \, di_1\, di_2}_{(*)}
\end{aligned} \tag{3.20}$$

where we can rewrite (*) as

$$\begin{aligned}
(*) &= \int_{\mathbb{R}} \frac{\partial p^g_{u+\epsilon h}(i_2)}{\partial \epsilon} \frac{1}{p^g_{u+\epsilon h}(i_2)} \underbrace{\left(\int_{\mathbb{R}} p^{f,g}_{u+\epsilon h}(i_1,i_2)\, di_1 \right)}_{p^g_{u+\epsilon h}(i_2)} di_2 \\
&= \frac{\partial}{\partial \epsilon} \underbrace{\left[\int_{\mathbb{R}} p^g_{u+\epsilon h}(i_2)\, di_2 \right]}_{1} = 0\,.
\end{aligned} \tag{3.21}$$

This result, together with the first variation of the estimator $p_u^{f,g}$ (Eq. 3.9) and $\epsilon = 0$, yields

$$\begin{aligned}
\delta_h \mathcal{J}_{\text{MI}}[u] =& -\frac{1}{|\Omega|}\int_{\mathbb{R}^2}\int_\Omega (E^{\text{MI}}(i_1,i_2,u)+1)\partial_2 G_\gamma(f(x)-i_1, g(x+u(x))-i_2) \\
& \nabla g(x+u(x))\cdot h(x)\,dx\,di_1\,di_2 \\
=& -\frac{1}{|\Omega|}\int_\Omega ((E^{\text{MI}}+1)\star\partial_2 G_\gamma)(f(x),g(x+u(x)),u) \\
& \nabla g(x+u(x))\cdot h(x)\,dx \\
=& -\frac{1}{|\Omega|}\int_\Omega (\partial_2(E^{\text{MI}}+1)\star G_\gamma)(f(x),g(x+u(x)),u) \\
& \nabla g(x+u(x))\cdot h(x)\,dx\;.
\end{aligned} \tag{3.22}$$

In this calculation a convolution $\star$ with respect to the intensity variables i_1 and i_2 appears, which commutes with the differential operator ∂_2.

With this consideration we finally obtain

$$\delta_h \mathcal{J}_{\text{MI}}[u] = -\frac{1}{|\Omega|}\int_\Omega (G_\gamma \star \partial_2 E^{\text{MI}})(f(x),g(x+u(x)),u)\nabla g(x+u(x))\cdot h(x)\,dx\;. \tag{3.23}$$

Again we define the so-called global intensity comparison function $L_{\text{MI}} : \mathbb{R}^2 \mapsto \mathbb{R}$ as

$$\begin{aligned}
L_u^{\text{MI}}(i_1,i_2) =& -\frac{1}{|\Omega|}G_\gamma \star \partial_2 E^{\text{MI}}(i_1,i_2,u) \\
=& -\frac{1}{|\Omega|}G_\gamma \star \partial_2 \log\frac{p_u^{f,g}(i_1,i_2)}{p^f(i_1)p_u^g(i_2)} \\
=& -\frac{1}{|\Omega|}G_\gamma \star \frac{p^f(i_1)p_u^g(i_2)}{p_u^{f,g}(i_1,i_2)}\,\frac{\partial_2 p_u^{f,g}(i_1,i_2)p^f(i_1)p_u^g(i_2) - p_u^{f,g}(i_1,i_2)p^f(i_1)p_u^{g\prime}(i_2)}{p^f(i_1)^2 p_u^g(i_2)^2} \\
=& -\frac{1}{|\Omega|}G_\gamma \star \left(\frac{\partial_2 p_u^{f,g}(i_1,i_2)}{p_u^{f,g}(i_1,i_2)} - \frac{p_u^{g\prime}(i_2)}{p_u^g(i_2)}\right)
\end{aligned} \tag{3.24}$$

to rewrite Eq. 3.23 as

$$\delta_h \mathcal{J}_{\text{MI}}[u] = \int_\Omega L_u^{\text{MI}}(f(x),g(x+u(x)))\nabla g(x+u(x))\cdot h(x)\,dx \tag{3.25}$$

and find the Euler-Lagrange Equation for $\mathcal{J}_{\text{MI}}[u]$ in the form

$$\nabla_u \mathcal{J}_{\text{MI}} = 0 \quad \text{with} \quad \nabla_u \mathcal{J}_{\text{MI}}(x) = L_u^{\text{MI}}(f(x),g(x+u(x)))\nabla g(x+u(x))\;. \tag{3.26}$$

3.2.3 Correlation Coefficients

Another widely used similarity criterion in image registration are the correlation coefficients, also called cross correlation (CC) [Cac00], [Her02]. This robust criterion measures the affine dependency of the intensity pairs of the images and is more restrictive than mutual information. We refer to [Her02] for nice illustrations of different similarity measures. As stated in [Cac00] CC is invariant by an affine rescaling of the intensity range of one of the images. In practice this means that it is insensitive to a uniform bias and a contrast change, as long as it is linear.
We will also introduce a local version of this similarity criterion. In that case the computation of the correlation coefficients is based on locally estimated statistical quantities. This has the advantage that bias is estimated locally around each point in the image. Even though this assumes a uniform bias within each window of the estimator, it allows for a varying non-uniform bias within the image from a global point of view [Cac00].

The definition of CC is based on the mean μ and variance v of the images. For a given pdf, which can be estimated by a one dimensional Parzen estimator according to Eq. 3.7 and Eq. 3.8, these quantities are given as:

$$\begin{aligned} \mu_1 &= \int_{\mathbb{R}} i_1 p^f(i_1)\, di_1, \quad & v_1 &= \int_{\mathbb{R}} i_1^2 p^f(i_1)\, di_1 - \mu_1^2 \\ \mu_2(u) &= \int_{\mathbb{R}} i_2 p_u^g(i_2)\, di_2, \quad & v_2(u) &= \int_{\mathbb{R}} i_2^2 p_u^g(i_2)\, di_2 - \mu_2(u)^2 \end{aligned} \tag{3.27}$$

With the covariance of X^f andX_u^g noted as $v_{1,2}(u)$

$$v_{1,2}(u) = \int_{\mathbb{R}^2} i_1 i_2 p_u^{f,g}(i_1, i_2)\, di_1\, di_2 - \mu_1 \mu_2(u) \tag{3.28}$$

the functional describing the dissimilarity based on the correlation coefficients is defined as:

$$\mathcal{J}_{\text{CC}}[u] = -\frac{v_{1,2}(u)^2}{v_1 v_2(u)} \tag{3.29}$$

First Variation

In order to employ this similarity measure in our variational framework we need to calculate the first variation of the functional $\mathcal{J}_{\text{CC}}[u]$. Even though the results are given in [Her02], we want to conduct a comprehensible calculation here explicitly.

First we compute:

$$\begin{aligned}
\frac{\partial \mu_2(u+\epsilon h)}{\partial \epsilon} &= \int_{\mathbb{R}} i_2 \frac{\partial p^g_{u+\epsilon h}(i_2)}{\partial \epsilon}\, di_2 \\
&= \int_{\mathbb{R}^2} i_2 \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2 \\
\frac{\partial \upsilon_2(u+\epsilon h)}{\partial \epsilon} &= \int_{\mathbb{R}^2} i_2^2 \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2 - 2\mu_2(u) \overbrace{\int_{\mathbb{R}^2} i_2 \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2}^{\frac{\partial \mu_2(u+\epsilon h)}{\partial \epsilon}} \\
&= \int_{\mathbb{R}^2} i_2(i_2 - 2\mu_2(u)) \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2 \\
\frac{\partial \upsilon_{1,2}(u+\epsilon h)}{\partial \epsilon} &= \int_{\mathbb{R}^2} i_1 i_2 \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2 - \mu_1 \overbrace{\int_{\mathbb{R}^2} i_2 \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2}^{\frac{\partial \mu_2(u+\epsilon h)}{\partial \epsilon}} \\
&= \int_{\mathbb{R}^2} i_2(i_1 - \mu_1) \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2
\end{aligned} \tag{3.30}$$

With these results we calculate $\frac{\partial \mathcal{J}_{\text{CC}}[u+\epsilon h]}{\partial \epsilon}$ to

$$\begin{aligned}
\frac{\partial \mathcal{J}_{\text{CC}}[u+\epsilon h]}{\partial \epsilon} &= -\frac{2\upsilon_{1,2}(u+\epsilon h)\frac{\partial \upsilon_{1,2}(u+\epsilon h)}{\partial \epsilon}}{\upsilon_1 \upsilon_2(u+\epsilon h)} + \frac{\upsilon_{1,2}(u+\epsilon h)^2 \frac{\partial \upsilon_2(u+\epsilon h)}{\partial \epsilon}}{\upsilon_1 \upsilon_2(u+\epsilon h)^2} \\
&= \int_{\mathbb{R}^2} E^{\text{CC}}(i_1, i_2, u+\epsilon h) \frac{\partial p^{f,g}_{u+\epsilon h}(i_1,i_2)}{\partial \epsilon}\, di_1\, di_2
\end{aligned} \tag{3.31}$$

where $E^{\text{CC}} : \mathbb{R}^2 \times V \mapsto \mathbb{R}$ is defined as:

$$E^{\text{CC}}(i_1, i_2, \cdot) = -\frac{1}{\upsilon_1 \upsilon_2(\cdot)}(2\upsilon_{1,2}(\cdot) i_2 (i_1 - \mu_1) - \mathcal{J}_{\text{CC}}[\cdot]\upsilon_1 i_2 (i_2 - 2\mu_2(\cdot))) \tag{3.32}$$

By setting $\epsilon = 0$, the first variation of the Parzen estimator $p^{f,g}_u$ (Eq. 3.9) and the same argumentation regarding the convolution as used for Eq. 3.23 in the calculation of MI, we obtain

$$\delta_h \mathcal{J}_{\text{CC}}[u] = \frac{1}{|\Omega|} \int_{\Omega} (G_\gamma \star \partial_2 E^{\text{CC}})(f(x), g(x+u(x)), u) \nabla g(x+u(x)) \cdot h(x)\, dx\ . \tag{3.33}$$

Again we define a global intensity comparison function $L_u^{\mathrm{CC}} : \mathbb{R}^2 \mapsto \mathbb{R}$ as:

$$\begin{aligned} L_u^{\mathrm{CC}}(i_1, i_2) =& \frac{1}{|\Omega|}(G_\gamma \star \partial_2 E^{\mathrm{CC}})(i_1, i_2, u) \\ =& -\frac{1}{|\Omega|} G_\gamma \star (\frac{1}{\upsilon_1 \upsilon_2(u)}(2\upsilon_{1,2}(u)(i_1 - \mu_1) + \mathcal{J}_{\mathrm{CC}}[u]\upsilon_1 2 i_2 - 2\mu_2(u))) \end{aligned} \tag{3.34}$$

Since $\partial_2 E^{\mathrm{CC}}$ is a sum of linear terms and G_γ is a symmetric and normalized kernel, the convolution with respect to the intensity variables has according to Theorem A.1 no effect and is thus redundant. In this case we have

$$(G_\gamma \star \partial_2 E^{\mathrm{CC}})(i_1, i_2, u) = \partial_2 E^{\mathrm{CC}}(i_1, i_2, u) \tag{3.35}$$

and can write

$$L_u^{\mathrm{CC}}(i_1, i_2) = -\frac{2}{|\Omega|}\left[\frac{\upsilon_{1,2}(u)}{\upsilon_2(u)}\left(\frac{i_1 - \mu_1}{\upsilon_1}\right) + \mathcal{J}_{\mathrm{CC}}[u]\left(\frac{i_2 - \mu_2(u)}{\upsilon_2(u)}\right)\right] . \tag{3.36}$$

In order to regard non-stationarities in the relation between intensities, we can make this estimator local. This can be done by employing Eq. 3.12 instead of Eq. 3.9 to estimate the jpdf. The local cross correlation (LCC) is then defined as:

$$\mathcal{J}_{\mathrm{LCC}}[u] = \int_\Omega \mathcal{J}_{\mathrm{CC}}[u](y)\, dy = \int_\Omega -\frac{\upsilon_{1,2}(u, y)^2}{\upsilon_1(y)\upsilon_2(u, y)}\, dy \tag{3.37}$$

We calculate the first variation as

$$\begin{aligned} \delta_h \mathcal{J}_{\mathrm{LCC}}[u] =& \int_\Omega \frac{1}{\mathcal{G}_\beta(y)} \int_{\mathbb{R}^2} \int_\Omega E^{\mathrm{LCC}}(i_1, i_2, u, y) \partial_2 G_\gamma(f(x) - i_1, g(x + u(x)) - i_2) \\ & G_\beta(x - y) \nabla g(x + u(x)) \cdot h(x)\, dx\, di_1\, di_2\, dy \\ =& \int_\Omega \int_\Omega \frac{1}{\mathcal{G}_\beta(y)} (G_\gamma \star \partial_2 E^{\mathrm{LCC}})(f(x), g(x + u(x)), u, y) \\ & G_\beta(x - y) \nabla g(x + u(x)) \cdot h(x)\, dy\, dx \\ =& \int_\Omega \frac{1}{\mathcal{G}_\beta(x)} (G_\gamma \star G_\beta \star \partial_2 E^{\mathrm{LCC}})(f(x), g(x + u(x)), u, x) \\ & \nabla g(x + u(x)) \cdot h(x)\, dx \end{aligned} \tag{3.38}$$

where $E^{\text{LCC}} : \mathbb{R}^2 \times V \times \Omega \mapsto \mathbb{R}$ is defined as:

$$\begin{aligned} &E^{\text{LCC}}(i_1, i_2, \cdot, x) = \\ &\quad -\frac{1}{\upsilon_1(x)\upsilon_2(\cdot, x)}(2\upsilon_{1,2}(\cdot, x)i_2(i_1 - \mu_1(x)) - \mathcal{J}_{\text{CC}}[\cdot](x)\upsilon_1(x)i_2(i_2 - 2\mu_2(\cdot, x)))\end{aligned} \tag{3.39}$$

Again we can neglect the convolution with G_γ and obtain for the correlation coefficients with locally estimated densities the intensity comparison function $L_u^{\text{LCC}} : \mathbb{R}^2 \times \Omega \mapsto \mathbb{R}$:

$$L_u^{\text{LCC}}(i_1, i_2, x) = G_\beta \star \frac{-2}{\mathcal{G}_\beta(x)} \left[\frac{\upsilon_{1,2}(u, x)}{\upsilon_2(u, x)} \left(\frac{i_1 - \mu_1(x)}{\upsilon_1(x)} \right) + \mathcal{J}_{\text{CC}}[u](x) \left(\frac{i_2 - \mu_2(u, x)}{\upsilon_2(u, x)} \right) \right] \tag{3.40}$$

Rewriting Eq. 3.38 to

$$\delta_h \mathcal{J}_{\text{LCC}}[u] = \int_\Omega L_u^{\text{LCC}}(f(x), g(x + u(x)), x) \nabla g(x + u(x)) \cdot h(x)\, dx \tag{3.41}$$

yields finally the Euler-Lagrange Equations, which are given for the global case by

$$\nabla_u \mathcal{J}_{\text{CC}} = 0 \quad \text{with} \quad \nabla_u \mathcal{J}_{\text{CC}}(x) = L_u^{\text{CC}}(f(x), g(x + u(x))) \nabla g(x + u(x)) \tag{3.42}$$

and for the local case by

$$\nabla_u \mathcal{J}_{\text{LCC}} = 0 \quad \text{with} \quad \nabla_u \mathcal{J}_{\text{LCC}}(x) = L_u^{\text{LCC}}(f(x), g(x + u(x)), x) \nabla g(x + u(x))\,. \tag{3.43}$$

3.3 Flows of Diffeomorphisms

The most natural way of optimizing a similarity criterion $\mathcal{J}$ and tackle the minimization problem posed in Eq. 3.1 is to come up with a sequence of transformations $\phi_k = id + u_k$ which follows iteratively the direction of the gradient $\nabla_{u_k}\mathcal{J}$. However, since we are looking for an optimal deformation ϕ which belongs to a function space, we need to employ a regularization technique to ensure the regularity of ϕ. As already mentioned in Section 2.2 this regularization is often done by adding a, with some scalar factor $\alpha \in \mathbb{R}^+$ weighted, penalty term $\mathcal{R}[u]$ to the cost functional $\mathcal{J}[u]$. This penalty term measures the irregularity of the transformation ϕ and is often used to introduce some physical knowledge to the model. This regularization originates in the Tikhonov functional [Her01a], where α controls the influence of the regularization term $\mathcal{R}$. Commonly used approaches based on this kind of regularization can be found in [Mod04] or [Her01a].
Another very intuitive approach of regularization was presented in [Che02], where the regular-

izer operator $\mathcal{R}$ is directly applied to the gradient $\nabla_{u_k}\mathcal{J}$ of the cost functional. We will employ and focus on the latter regularizer.

We can now consider the first step of a gradient algorithm with an initial deformation $\phi_0 = id$. By applying the regularizer directly to the so-called (driving) force field $\nabla_{u_k}\mathcal{J}$ we obtain a new transformation ϕ_1 as

$$\phi_1 = \phi_0 + \epsilon\mathcal{R}(\nabla_{u_0}\mathcal{J}) = id + \epsilon\mathcal{R}(\nabla_0\mathcal{J}) \; . \tag{3.44}$$

However, instead of applying this relation iteratively to obtain a sequence ϕ_k we stick to the template propagation method proposed in [Che02]. In this approach the deformation field ϕ_1 is immediately applied to the template image g. This results in a new, propagated image $g_1 = g \circ (id + \epsilon\mathcal{R}(\nabla_0\mathcal{J}))$. We can now consider a new matching problem of f and g_1 with the initial deformation $\phi_0 = id$. The deformation ϕ_k is then defined as the transformation that connects the original template image g to the propagated template image g_k after k iterations by $g_k = g \circ \phi_k$.

Overall this reduces to the following approach:

$$\begin{array}{rcll} \nu_k & \leftarrow & L(f, g \circ \phi_k)\nabla g_{u_k} & \text{(displacement field)} \\ \nu_k & \leftarrow & \mathcal{R}(\nu_k) & \text{(regularization)} \\ \phi_{k+1} & \leftarrow & \phi_k \circ (id + \epsilon_k \nu_k) & \text{(update)} \\ u_{k+1} & \leftarrow & \phi_{k+1} - id & \end{array} \tag{3.45}$$

The computation of ν_k is based on the evaluation of the intensity comparison function L of a certain similarity criterion. We calculated intensity comparison functions for SSD, MI and (L)CC in Section 3.2. The regularization in this method is done locally for each displacement field ν_k instead of the complete transformation ϕ_k. Compared to the mentioned Tikhonov regularizers this has the huge advantage that large deformations are not necessarily unreasonably penalized. As stated in [Che02] there is always an ϵ_k so that ϕ_{k+1} is a diffeomorphism, provided that ϕ_k is a diffeomorphism and ν_k is sufficiently smooth.

This algorithm was presented in [Che02], where a continuous formulation of Algorithm 3.45 and statements on the regularization are given. It is proposed to set the regularizer $\mathcal{R}(u) = \kappa \star u$ where κ is a generic linear filter. Since a Gaussian filter with standard deviation σ is suitable for this purpose, the regularization can be done very efficiently as we will investigate in Section 4.2. To recover local or global deformations the choice of σ is intuitive and application dependent.

We refer to [Che02] for more details on the continuous formulation and regularization.

Chapter 4

Implementation and Assembly of Key Building Blocks

In the preceding chapters we gave an introduction to NVIDIA CUDA, nonrigid registration methods and introduced a certain registration algorithm within a variational framework. To employ different similarity measures within this approach we provided variational gradients for three different cost functions based on SSD, MI and LCC. While the computation of MI relies on the availability of a jpdf estimation of moving and reference image, LCC depends on an efficient computation of local statistics. Furthermore efficient filtering is mandatory to provide a fast regularization of the deformation fields. The presented registration algorithm breaks down to two major building blocks: Gaussian filtering (regularization, estimation of local statistics) and joint histogram computation (estimation of jpdf). In this chapter we will describe how local statistics can be computed with a Gaussian filter and how this type of filter can be implemented in a recursive and very efficient way. Furthermore we will introduce novel optimization techniques to compute joint histograms efficiently on CUDA compatible GPUs. Finally we can concatenate these blocks and present a whole nonrigid registration pipeline which is based on the matching algorithm presented in Chapter 3.

4.1 Computation of Local Statistics

If LCC is employed as similarity measure, local variances and local means need to be estimated. As already mentioned in Section 3.2 this is done with a Parzen estimator. Even if we could use any symmetric window function for this estimation, we used a Gaussian G since it is the only isotropic kernel with the following two properties [Cac00, Kan92]:

1. The Gaussian convolution is separable. This means that a convolution of a 3D image with a 3D Gaussian kernel is equivalent to a successive convolution with three 1D Gaussian kernels.

2. As we will present in Section 4.2 the convolution with a 1D Gaussian kernel can be approximated by a high efficient recursive filter. It also turns out that this filter is independent of the standard deviation of the Gaussian.

With the following consideration we can easily see, that the local mean $\mu_f(x)$ of a function f is then estimated by a convolution with a Gaussian kernel G_β with standard deviation β. This approach originates in the local versions of Eq. 3.7 and Eq. 3.27.

$$\begin{aligned} \mu_f(x) &= \int_{\mathbb{R}} i \frac{1}{\mathcal{G}_\beta(x)} \int_\Omega G_\gamma(f(y) - i) G_\beta(y-x)\, dy\, di \\ &= \int_\Omega \underbrace{\frac{1}{\mathcal{G}_\beta(x)}}_{=1^{(*)}} (G_\gamma \star id)(f(y)) G_\beta(y-x)\, dy \\ &= \int_\Omega f(y) G_\beta(y-x)\, dy = (G_\beta \star f)(x) \end{aligned} \tag{4.1}$$

$(*)$: At the image boundaries this is only true if the convolution with respect to y is done with a complete Gaussian kernel by for example padding the image.

Again we used the fact that two convolutions, one with respect to the intensity variable i and one with respect to the space variable y emerge. We also employed Theorem A.1 which states that $(G_\gamma \star id)(f(y))$ is equal to $f(y)$.

The choice of β defines the neighborhood, which contributes to the computation of the local mean, and is intuitive.

For a given displacement field u we recap $g_u : \Omega \mapsto \mathbb{R}$ as $g_u(x) = g(x + u(x))$ and can now compute all local statistics around each voxel with Gaussian convolutions as follows:

$$\begin{aligned} \mu_f(x) = (G_\beta \star f)(x) \quad &, \quad \mu_g(x,u) = (G_\beta \star g_u)(x) \\ v_f(x) = (G_\beta \star f^2)(x) - (\mu_f(x))^2 \quad &, \quad v_g(x,u) = (G_\beta \star g_u^2)(x) - (\mu_g(x,u))^2 \end{aligned} \tag{4.2}$$

$$v_{f,g}(x,u) = (G_\beta \star f g_u)(x) - \mu_f(x)\mu_g(x,u) \tag{4.3}$$

4.2 Efficient Gaussian Filtering

4.2.1 Application

In every iteration of the presented registration process a Gaussian filter is applied to regularize the displacement field. If LCC is employed as similarity measure, also local means, local variances and local correlations are estimated using a Gaussian filter as shown in Section 4.1. Thus the overall runtime is significantly dominated by Gaussian filtering which is used as regularizer and estimator for local statistics.

One common approach to implement this filtering technique is to compute the convolution directly. This so-called *direct convolution*, implemented as separable filter with a finite-length approximation to a Gaussian, is already very efficient for filter kernels with a small standard deviation σ.

In [Hal06] two popular recursive infinite impulse response (IIR) filters, the method of Deriche [Der93] and the method of van Vliet et al. [vV98, You95], are compared to direct convolution. Both recursive methods are approximating a Gaussian filter. While the computation time of direct convolution linearly depends on σ, the cost of the recursive approaches is constant for any choice of σ. It turns out that especially for larger σ recursive methods outperform direct convolution. The independence of the computation time of σ as well as the good parallelizability of these approaches motivated the investigation of recursive Gaussian filters on the GPU. However, since Deriche's method yields more accurate results for $\sigma < 32$ than van Vliet's implementation [Hal06] and we will not choose a $\sigma \geq 32$, we will solely focus on Deriche's approach in the following.

4.2.2 Fast Recursive Deriche Filter

We will now present a theoretical introduction to an IIR filter, which is applicable for separable Gaussian filtering and was introduced by Deriche in [Der87, Der90, Der93]. Even if we most often follow the notation used by Deriche, we will, with ambition to provide better readability, slightly adapt it. Within the scope of this thesis we need to restrict the theoretical introduction to the basics, which are required to get a rough understanding of this approach. For a further understanding of filter design, we refer to Deriche's work and the article of Farnebäck [Far06].

Theoretical Background

We consider the following stable and non recursive convolution operation

$$y_i = \sum_{k=0}^{N-1} h(k) x_{i-k} \tag{4.4}$$

which relates some input sequence x_i to an output sequence y_i. This operation is, dependent on the length N, computationally more or less expensive. In the context of Gaussian filtering, Eq. 4.4 is equivalent to a half Gaussian filter and N defines the cut-off limit of the finite-length approximation to the Gaussian. According to [Der93] and [Hal06] a value of roughly 4σ is usually a good choice for N. This is why the computational cost of a direct convolution directly depends on σ.

With the application of the Z-transform [Mul99, Che01, Phi98] to the discrete impulse response $h(k)$, which will be the sampled Gaussian kernel, we can describe Eq. 4.4 with the transfer function

$$H(z^{-1}) = \sum_{k=0}^{N-1} h(k) z^{-k} \ . \tag{4.5}$$

The crucial idea behind recursive filter approximation is now, to exactly represent or approximate Eq. 4.5 with a rational transfer function of the form

$$H_{\text{appr}}(z^{-1}) = \frac{\sum_{k=0}^{n-1} b_k z^{-k}}{1 + \sum_{k=1}^{n} a_k z^{-k}} \ . \tag{4.6}$$

With this representation we could now highly benefit of the computationally cheaper solution of the n-th order recursive system

$$y_i = \sum_{k=0}^{n-1} b_k x(i-k) - \sum_{k=1}^{n} a_k y(i-k) \ , \tag{4.7}$$

which is characterized by Eq. 4.6 and requires just $2n$ instead of N operations per output element y_i. Since according to [Der93] an approximation of a Gaussian filter is sufficiently elaborate for most applications for $n \geq 3$, we consider and investigate a 4-th order approximation ($n = 4$) in the following.

As already mentioned, the causal sequence of Eq. 4.4 just describes a half Gaussian filter. The fact that a full Gaussian filter is non-causal, indeed the y_i's depend on x_i's ahead, requires an additional consideration to apply this recursive approach.

First we can rewrite the impulse response $g(k) = e^{-\frac{k^2}{2\sigma^2}}$ of a full Gaussian filter as the sum of the two halves of this filter as $g(k) = g^+(k) + g^-(k)$ with

$$g^+(k) = \begin{cases} g(k), & \text{if } k \geq 0, \\ 0, & \text{else} \end{cases} \tag{4.8}$$

$$g^-(k) = \begin{cases} 0, & \text{if } k \geq 0, \\ g(k), & \text{else .} \end{cases} \tag{4.9}$$

With this convention we can describe a 4-th order approximation $g_{\text{appr}}(k)$ of the impulse response $g(k)$ with the Z-transforms

$$G^+_{\text{appr}}(z^{-1}) = \sum_{k=0}^{\infty} g^+_{\text{appr}}(k) z^{-k} = \frac{n_0^+ + n_1^+ z^{-1} + n_2^+ z^{-2} + n_3^+ z^{-3}}{1 + d_1^+ z^{-1} + d_2^+ z^{-2} + d_3^+ z^{-3} + d_4^+ z^{-4}}\,, \tag{4.10}$$

$$G^-_{\text{appr}}(z) = \sum_{k=-\infty}^{0} g^-_{\text{appr}}(k) z^{k} = \frac{n_1^- z + n_2^- z^2 + n_3^- z^3 + n_4^- z^4}{1 + d_1^- z + d_2^- z^2 + d_3^- z^3 + d_4^- z^4}\,. \tag{4.11}$$

These systems describe the two independent recursive systems

$$\begin{aligned} y_k^+ = \;& n_0^+ x_k + n_1^+ x_{k-1} + n_2^+ x_{k-2} + n_3^+ x_{k-3} \\ & - d_1^+ y_{k-1}^+ - d_2^+ y_{k-2}^+ - d_3^+ y_{k-3}^+ - d_4^+ y_{k-4}^+ \quad k = 1, \ldots, N \end{aligned} \tag{4.12}$$

$$\begin{aligned} y_k^- = \;& n_1^- x_{k+1} + n_2^- x_{k+2} + n_3^- x_{k+3} + n_4^- x_{k+4} \\ & - d_1^- y_{k+1}^- - d_2^- y_{k+2}^- - d_3^- y_{k+3}^- - d_4^- y_{k+4}^- \quad k = N, \ldots, 1 \end{aligned} \tag{4.13}$$

Because of the relation $G_{\text{appr}}(z) = Z[g_{\text{appr}}(k)] = Z[g^+_{\text{appr}}(k) + g^-_{\text{appr}}(k)] = Z[g^+_{\text{appr}}(k)] + Z[g^-_{\text{appr}}(k)] = G^+_{\text{appr}}(z^{-1}) + G^-_{\text{appr}}(z)$ the result y_k is obtained by calculating $y_k = y_k^+ + y_k^-$ [Der93]. We are also interested in an even impulse response, the Gaussian is a symmetric filter, and can thus set

$$\begin{aligned} d_i^- &= d_i^+ & i &= 1, \ldots, 4 \\ n_i^- &= n_i^+ - d_i^+ n_0^+ & i &= 1, \ldots, 3 \\ n_4^- &= -d_4^+ n_0^+ & & . \end{aligned} \tag{4.14}$$

It remains to determine the filter coefficients d_i^+ for $i = 1, \ldots, 4$ and n_i^+ for $i = 0, \ldots, 3$.

In [Der93] and [Far06] different design techniques are presented to obtain these parameters. Even if this parameter design needs to be carried out only once and the results can be borrowed from papers as [Der93, Far06], we want to provide some more background.

A general 4-th order approximation of the operator $g^+(k)$ is given by

$$g^+_{\text{appr}}(k) = \sum_{i=1}^{4} \alpha_i e^{-\lambda_i \frac{k}{\sigma}} \ . \tag{4.15}$$

Since we want to optimize real coefficients, α_i and λ_i ($i = 1, \dots, 4$) may be complex but pairwise conjugate. This means specifically $\alpha_1 = \overline{\alpha_2}, \alpha_3 = \overline{\alpha_4}, \lambda_1 = \overline{\lambda_2}$ and $\lambda_3 = \overline{\lambda_4}$. After rewriting Eq. 4.15 according to Eq. A.2 in Section A as

$$g^+_{\text{appr}}(k) = \sum_{i=1}^{2} (a_i \cos \frac{\omega_i}{\sigma} k + b_i \sin \frac{\omega_i}{\sigma} k) e^{-\frac{\beta_i}{\sigma} k}, k \geq 0 \tag{4.16}$$

we can compute all 8 coefficients $a_1, a_2, b_1, b_2, \omega_1, \omega_2, \beta_1$ and β_2 by fitting Eq. 4.16 to a continuous Gaussian. This is done by solving the nonlinear optimization problem

$$\underset{a_1,a_2,b_1,b_2,\omega_1,\omega_2,\beta_1,\beta_2}{\text{argmin}} \sum_{i=0}^{i=10\sigma} (e^{-\frac{i^2}{2\sigma^2}} - g^+_{\text{appr}}(i))^2 \tag{4.17}$$

with for example Matlab. In order to obtain a good fit for any scale, Deriche suggests in [Der93] to use a σ of 100 and a set of 1000 sample points for this optimization.
The coefficients d_i^+ for $i = 1, \dots, 4$ and n_i^+ for $i = 0, \dots, 3$ can be calculated by

$$\begin{aligned}
n_0^+ &= a_1 + a_2 \\
n_1^+ &= e^{-\frac{\beta_1}{\sigma}}(-(a_1 + 2a_2)\cos(\tfrac{\omega_1}{\sigma}) + b_1 \sin(\tfrac{\omega_1}{\sigma})) \\
&\quad e^{-\frac{\beta_2}{\sigma}}(-(2a_1 + a_2)\cos(\tfrac{\omega_2}{\sigma}) + b_2 \sin(\tfrac{\omega_2}{\sigma})) \\
n_2^+ &= a_1 e^{-2\frac{\beta_2}{\sigma}} + a_2 e^{-2\frac{\beta_1}{\sigma}} + 2e^{-\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}((a_1 + a_2)\cos(\tfrac{\omega_1}{\sigma})\cos(\tfrac{\omega_2}{\sigma}) \\
&\quad -b_1 \cos(\tfrac{\omega_2}{\sigma})\sin(\tfrac{\omega_1}{\sigma}) - b_2 \cos(\tfrac{\omega_1}{\sigma})\sin(\tfrac{\omega_2}{\sigma})) \\
n_3^+ &= e^{-\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}(-a_1 \cos(\tfrac{\omega_1}{\sigma}) + b_1 \sin(\tfrac{\omega_1}{\sigma})) + e^{-2\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}(-a_2 \cos(\tfrac{\omega_2}{\sigma}) + b_2 \sin(\tfrac{\omega_2}{\sigma}))
\end{aligned} \tag{4.18}$$

$$\begin{aligned}
d_1^+ &= -2e^{-\frac{\beta_1}{\sigma}}\cos(\tfrac{\omega_1}{\sigma}) - 2e^{-\frac{\beta_2}{\sigma}}\cos(\tfrac{\omega_2}{\sigma}) \\
d_2^+ &= 4e^{-\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}\cos(\tfrac{\omega_1}{\sigma})\cos(\tfrac{\omega_2}{\sigma}) + e^{-2\frac{\beta_1}{\sigma}} + e^{-2\frac{\beta_2}{\sigma}} \\
d_3^+ &= -2e^{-\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}\cos(\tfrac{\omega_1}{\sigma}) - 2e^{-2\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}\cos(\tfrac{\omega_2}{\sigma}) \\
d_4^+ &= e^{-2\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}
\end{aligned}$$

This relation is obtained by applying a Z-transform to Eq. 4.16 [Che01, p. 210] and a comparism of the coefficients. The detailed calculation can be found in Eq. A.4 in Section A. The missing coefficients d_i^- and n_i^- $i = 1, \dots, 4$ are computed according to Eq. 4.14.

Implementation

Since it turns out that filtering is next to the joint histogram computation, only if MI is used, the dominating building block in terms of computation time, we want to investigate some aspects of the implementation more thoroughly.

Our recursive implementation for filtering an arbitrary 3D volume following Deriche's approach, consists of two steps per dimension. Since we allow to choose any σ_c, σ_r and σ_d for filtering in column, row or depth direction, we need to compute the filter parameters separately for each dimension. This step mainly breaks down to a computation of the coefficients according to Eq. 4.18 and a subsequent normalization. For this normalization step we create an array $x(i)$ which is with size $2\lfloor 10\sigma_i \rfloor + 1$ large enough to catch all filter responses and initialized as

$$x(i) = \begin{cases} 1, & \text{if } i = \lfloor 10\sigma_i \rfloor, \\ 0, & \text{else} \end{cases} \tag{4.19}$$

We filter this array by applying our recursive Deriche filter with the coefficients obtained by Eq. 4.18 and compute $\alpha = \sum_{i=0}^{2\lfloor 10\sigma_i \rfloor} x(i)$. In order to have a mass conserving filter this sum needs to be equal to 1. We fulfill this condition by scaling the coefficients n_i^+ with $\frac{1}{\alpha}$ for $i = 0, \dots, 3$.

With these coefficients we can now filter along the according dimension in a second step. Among different approaches, we observed lowest computation times by computing one complete line of the volume per thread. Common volumes are of sufficient size so that the GPU is occupied by a reasonable number of threads. As experimental results in Chapter 5 reveal, this approach allows due to coalesced memory access for very efficient filtering in row (same column, same slice) and depth (same column, same row) direction. Filtering in column direction suffers significantly of uncoalesced memory access. We tackled this problem by employing local memory to read and write intermediate results coalesced, which is an important optimization. Furthermore we used fast registers to cache the four x_k and four y_k in the computation of Eq. 4.12 and Eq. 4.13. With this optimization redundant access to global memory is avoided and the original volume is only read twice, once for the causal (Eq. 4.12) and once for the non-causal (Eq. 4.13) filtering step. Again, local memory allows to always read the image data for the non-causal filtering step coalesced.

Boundary conditions can be imposed by padding every line of the volume with $\max(5\lfloor \sigma_{c,r,d} + 0.5 \rfloor, 10)$ elements on either side. We set these elements to $x(0)$ at the start and $x(n-1)$ at the end of the line respectively, what corresponds to *clamping* as boundary condition.

4.3 Efficient Histogram Computation

Histograms, which show the frequency of occurrence of data elements, are a frequently employed analysis tool not just in image registration, but also in other fields as for example data mining. In multimodal image registration, joint histograms are the basis of a very powerful and robust criterion, MI. Since many clinical applications call for robust, high accurate, real-time registration based on MI, it is important to adapt and especially optimize existing histogram algorithms in this context.

Not least because improvements to joint histogram computation are not restricted to registration, we will in the following of this chapter present the results of the research that we conducted on optimizing existing histogram algorithms.

One widely pursued approach to efficient (joint) histogram computation is to employ GPUs. Although histogram computation is simple and trivial to implement on the CPU, it is a challenging task, to parallelize it efficiently on the GPU [Pod07].

The problem arising in the parallel calculation of a histogram with B bins distributed over N threads is illustrated in Figure 4.1. This example shows potential and unpredictable, data dependent collisions of different threads while updating the same histogram bin.

As stated in [Pod07], former approaches based on shader programming rely on expensive preprocessing steps, as sorting the pixels by intensity value, which are undesirable. With NVIDIA CUDA a new architecture for general purpose parallel computing is available. As already mentioned, especially the availability of shared memory and synchronization routines enable efficient solutions on the GPU for more complex problems. New algorithms for histogram computation on CUDA compatible devices were presented in [Pod07], [Sha07b], [Bro09], [Sha10], [Che09], [Sha07a] and [Lin08].

In [Pod07] a very efficient 64 bin histogram computation was proposed and extended to 256 bins by simulating atomic intra-warp updates to the shared memory.

In order to compute thousands of bins, this update mechanism was also used in the first method presented in [Sha07b] (labeled *Method1*). This algorithm lacks data independence, but it has the key advantage of low memory requirement and very good performance on real data. To overcome data dependence, the second method proposed in [Sha07b] (*Method2*) ensures collision free updates. This method requires the allocation of a full histogram for every thread in the global memory, which results in additional memory requirement. Dependent on the data, it

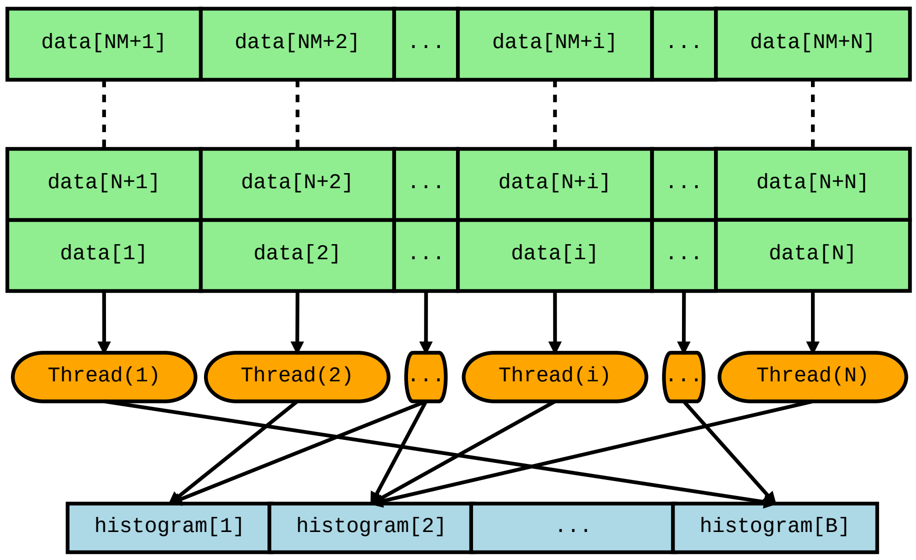


Figure 4.1: Parallel calculation of a histogram with B bins distributed over N threads. Histogram updates conflict and require synchronization of the threads or atomic updates to the histogram memory.

also underperforms *Method1* for high bin ranges [Sha07b], which often occur in joint histogram computations.

A new algorithm – named "sort and count" – was recently proposed in [Sha10]. This algorithm possesses very good scaling with high bin numbers. For reasonable joint histogram sizes around 75×75 bins this approach performs similar to *Method1*.

A self-optimizing histogram algorithm is presented in [Bro09]. Because of the very expensive preprocessing time of several seconds, this data dependent optimization does not meet our needs. In [Che09] an algorithm which sorts the image by intensity values was presented. Once the image is sorted, faster histogram computations are possible. Next to the high memory requirement because of additional stored coordinates, the expensive preprocessing (sorting) is a major drawback.

In addition to these exact computations, very fast algorithms were proposed in [Sha07a] and [Lin08] to approximate histograms and MI.

After we gave a rough overview over work related to this problem we will state the problem in Section 4.3.1. In Section 4.3.2 we will introduce an existing algorithm for histogram computation and present three novel optimization strategies for this approach in Section 4.3.3.

4.3.1 Problem Formulation

To evaluate Eq. 3.18 an estimation of the marginal probability distributions $p_X(x)$ and $p_Y(y)$ and the joint probability distribution $p_{XY}(x, y)$ is required. This is done by computing histograms or the joint histogram respectively. Since the similarity measure has to be evaluated in each iteration, it is crucial to have a very fast and efficient algorithm for histogram computation in place, when MI is used. The computation time of MI is dominated by the joint histogram computation.

For two normalized discrete images on $\Omega_D \subset \mathbb{Z}^n$, $I_{\mathrm{r}} : \Omega_D \mapsto [0.0, 1.0]$ (reference image) and $I_{\mathrm{m}} : \Omega_D \mapsto [0.0, 1.0]$ (moving image) the joint histogram with $\mathrm{B_r} \times \mathrm{B_m}$ bins can be described as:

$$J(i, j) = \sum_{x \in \Omega_D} \delta(x) \tag{4.20}$$

$$\text{with} \qquad \delta(x) = \begin{cases} 1, & \text{if } I_{\mathrm{r}}(x) \in [\frac{i}{\mathrm{B_r}}, \frac{i+1}{\mathrm{B_r}}] \text{ and } I_{\mathrm{m}}(\phi(x)) \in [\frac{j}{\mathrm{B_m}}, \frac{j+1}{\mathrm{B_m}}] \\ 0, & \text{otherwise} \end{cases} \tag{4.21}$$

where $\phi(x)$ is a geometric transformation applied to the moving image.

By employing interpolation both images I_{r} and I_{m} can be evaluated continuously. At this point we should also give a short remark to stay consistent in terms of notations: In the presented framework the continuous image f can be obtained by applying an interpolation scheme to the discrete image I_{r}, g is derived from I_{m} respectively.

As shown in [Sha07a], joint histogram computation of $\mathrm{B_r} \times \mathrm{B_m}$ bins is equivalent to the computation of a single histogram with $\mathrm{B_j} = \mathrm{B_r B_m}$ bins on I_{c} by combining the intensity values of the two original images such that,

$$I_{\mathrm{c}}(\phi(\cdot), x) = \frac{\mathrm{B_m}((\mathrm{B_r} - 1) I_{\mathrm{r}}(x) + I_{\mathrm{m}}(\phi(x)))}{\mathrm{B_j} - 1}, \mathrm{B_j} > 1 \; . \tag{4.22}$$

This new image, usually created in a preprocessing step, requires additional memory. This can be problematic for the registration of large datasets.

For histogram computation *Method1* is very efficient for common data. There are neither limitations of the number of histogram bins nor a loss of accuracy by approximations. We will now present the base algorithm (*Method1*) and three optimization techniques for this approach, which enable a high efficient calculation in terms of computation time as well as memory requirement.

4.3.2 Base Algorithm

When parallelizing histogram computations on GPU, the main bottleneck is that different threads might compete by updating the same histogram bin. Even if there is in general no “mutex” mechanism available in CUDA[1] a solution was presented in [Pod07] and employed in *Method1* for an efficient parallel histogram computation.

This approach is based on the fact that within one warp (32 threads) all threads execute the same instructions in parallel (“at the same time”). Within one warp an atomic update of a certain histogram bin can then be simulated by tagging data with the ID of the updating thread. If more than one thread updates this data, exactly one thread will succeed and data tagged with this thread ID will remain. Thus it is possible to check which thread actually succeeded and redo the update for all not successful threads [Pod07, NVI10c]. This is why this algorithm is data dependent and performs worst for images with constant intensity values, where all threads of a warp collide while updating the same bin and hence need to be serialized. How to simulate this atomic update within one warp can be found in Listing B.1 in Appendix B.

Since global memory is slow, shared memory is used to hold temporary histograms. In *Method1*, an unsigned integer is used in shared memory to hold the tag in the 5 ($log_2 32$) most significant bits and the bin counter in the remaining 27 bits for each bin. The limitations for this approach arise because shared memory is limited to 16 kB per block and in order to occupy the GPU, several warps need to be run within one execution block. As a result, a partial joint histogram has to be allocated in shared memory for every warp.

In order to employ this algorithm for joint histogram computation, a new image is first created according to Eq. 4.22. This operation is expensive since it increases the memory requirement by a full image size. If no additional memory is available Eq. 4.22 has to be evaluated within each kernel call.

The computation of a reasonable joint histogram of size 80×80 with 4 warps (128 threads) per block would exceed the available shared memory by far (with a memory requirement of 100 kB $= 4$ warps $\times$ 6400 bins $\times$ 4 Bytes). For this reason, joint histograms usually cannot be computed with one kernel call. In fact, several kernel calls are necessary to compute step by step the complete joint histogram by computing a certain range of bins within each call.

In the following we propose to extend this algorithm with three major optimizations, which focus on execution time as well as on memory requirement.

[1]As described in [NVI10c], this functionality depends on the *compute capability* of the device. Potential improvements based on atomic operations remain to be investigated.

4.3.3 Contribution

Multiple Bin Encoding (MBE)

In order to use the shared memory more efficiently, we encode two bins in 26 of the 27 less significant bits of an unsigned integer. This concept is shown in Fig. 4.2. Of course this makes collisions more likely to happen, since even updates to two different bins may conflict because they are encoded in the same integer. Nevertheless *MBE* allows the handling of twice the number of bins per kernel call. This results in less overall kernel calls and therefore in less global memory access, which is especially important when it comes to expensive interpolation schemes and no spare memory is available. With this optimization the data of a bin is encoded with 13 bits. Overflows are avoided by regularly updating the block histograms in the global memory. This optimization is always applicable and has no influence on memory requirements.

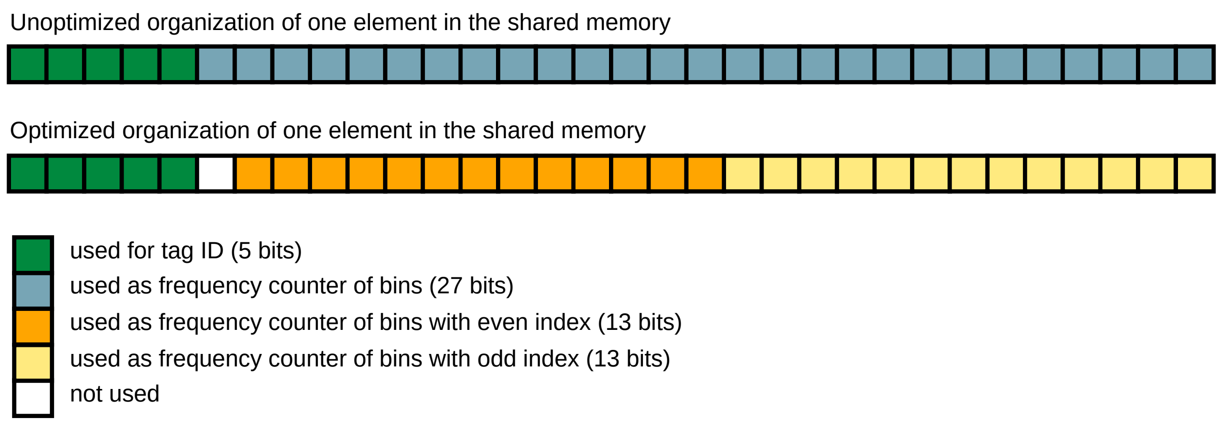

Unoptimized organization of one element in the shared memory
Optimized organization of one element in the shared memory
used for tag ID (5 bits)
used as frequency counter of bins (27 bits)
used as frequency counter of bins with even index (13 bits)
used as frequency counter of bins with odd index (13 bits)
not used

Figure 4.2: Example of MBE using an unsigned integer to encode two bins instead of one. Bin1 and Bin2 are encoded in the first word, Bin3 and Bin4 in the second word, etc. [Led10]

Bin Caching (BC)

If spare global memory is available, this optimization should be used in addition to *MBE*. Instead of performing a preprocessing step to create I_c, we use the first kernel call to compute the resulting bin number for a certain voxel pair. This is important since we avoid an unnecessary reread of the preprocessed image. It is also more efficient to store the resulting bin number instead of a combined intensity value. During the first kernel call the resulting bin number $[\mathrm{B_m}((\mathrm{B_r}-1)I_r(x)+I_m(\phi(x)))]$ is stored to additional global memory. Joint histograms of more than 255×255 bins are very seldom of interest. In this case a resulting bin will never exceed 16 bit. We propose to pack two bin numbers into an unsigned integer which saves 50% of the additional required memory compared to the commonly used preprocessing. This results not just in an optimization in terms of memory demand, but also in a reduced global memory access. Subsequent kernels just read from the global memory the bin of the joint histogram that needs to be incremented. Especially in cases where several kernel calls are necessary or expensive interpolation schemes are used, this is a very valuable optimization.

However, this approach requires additional memory of 50% of an image size. This is an improvement compared to the regular preprocessing step, but still too much to claim general applicability for joint histogram computation. See Figure 4.3 for a schematic flowchart of this optimization.

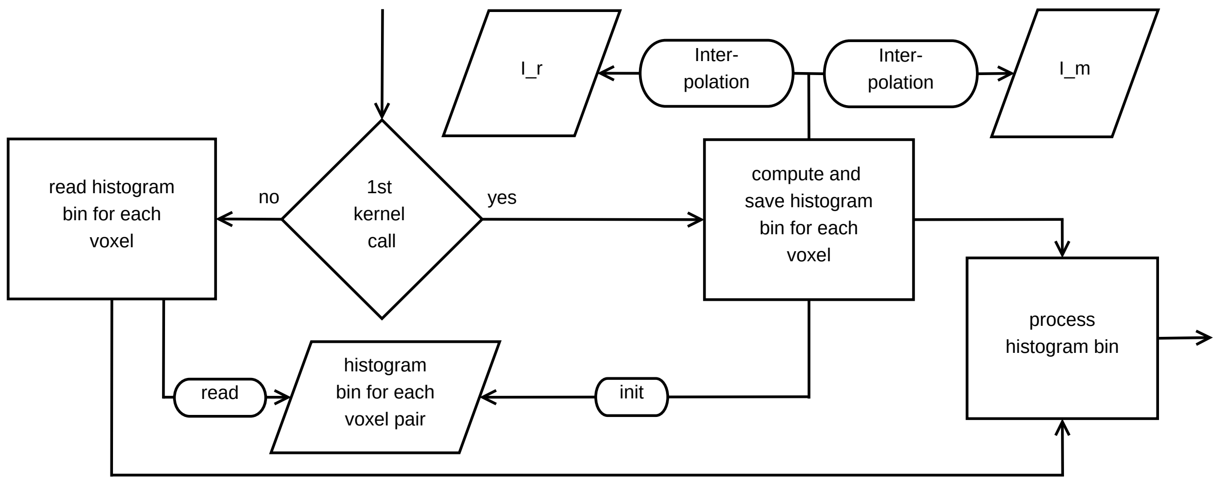

I_r
Inter-polation
Inter-polation
I_m
read histogram bin for each voxel
no
1st kernel call
yes
compute and save histogram bin for each voxel
process histogram bin
read
histogram bin for each voxel pair
init

Figure 4.3: Schematic flowchart of the kernel optimized with BC. [Led10]

Smart Texture Lookup (STL)

When there is not enough global memory temporarily available to apply *BC* or to use the regular preprocessing according to Eq. 4.22 the resulting bin number $[\mathrm{B_m}((\mathrm{B_r}-1)I_\mathrm{r}(x)+I_\mathrm{m}(\phi(x)))]$ needs to be computed within each kernel call. For this case we introduce an important optimization in addition to *MBE*. As already mentioned, the computation of a joint histogram with $\mathrm{B_r} \times \mathrm{B_m}$ bins is equivalent to the computation of a histogram of size $\mathrm{B_j} = \mathrm{B_r B_m}$.

Whether a kernel call processes a certain voxel pair, depends on the intensity values of the voxels in I_r and I_m. Often already the intensity value in I_r rules out a consideration for this voxel pair in this kernel call. Therefore the texture lookup in I_m needs to be skipped to avoid unnecessary and expensive texture lookups and interpolations in the global memory.

Within the k-th kernel call (each kernel computes $\mathrm{B_{kernel}}$ bins) only combinations of intensity values are considered for which the following equation holds:

$$(k-1)\mathrm{B}_{\text{kernel}} \leq \mathrm{B}_{\text{m}}((\mathrm{B}_{\text{r}}-1)I_{\text{r}}(x)+I_{\text{m}}(\phi(x))) < k\mathrm{B}_{\text{kernel}} \tag{4.23}$$

By reformulation Eq. 4.23 and the fact that $0.0 \leq I_{\text{m}}(\phi(x)) \leq 1.0$ we get

$$(k-1)\mathrm{B}_{\text{kernel}} - \mathrm{B}_{\text{m}} \leq \mathrm{B}_{\text{m}}(\mathrm{B}_{\text{r}}-1)I_{\text{r}}(x) < k\mathrm{B}_{\text{kernel}}\ . \tag{4.24}$$

If Eq. 4.24 is not true for a $I_{\text{r}}(x)$, the texture lookup of $I_{\text{m}}(\phi(x))$ is skipped.

Since *Method1* optimized by *MBE & BC* is showing the best performance it is most suitable for our purposes. Even if we will investigate the performance obtained with *MBE & STL* in Chapter 5, we will solely employ *MBE & BC* in our registration pipeline.

At this point we mention that we also published the results that were described in this section in [Led10].

4.4 Nonrigid Registration Pipeline

To this point of the thesis we have introduced all major methods and building blocks. Now, we can assemble these building blocks to a complete nonrigid registration pipeline, which will be presented in the following.

A rough overview over the essential parts of the registration algorithm, which were all implemented in CUDA, is given in Figure 4.4.

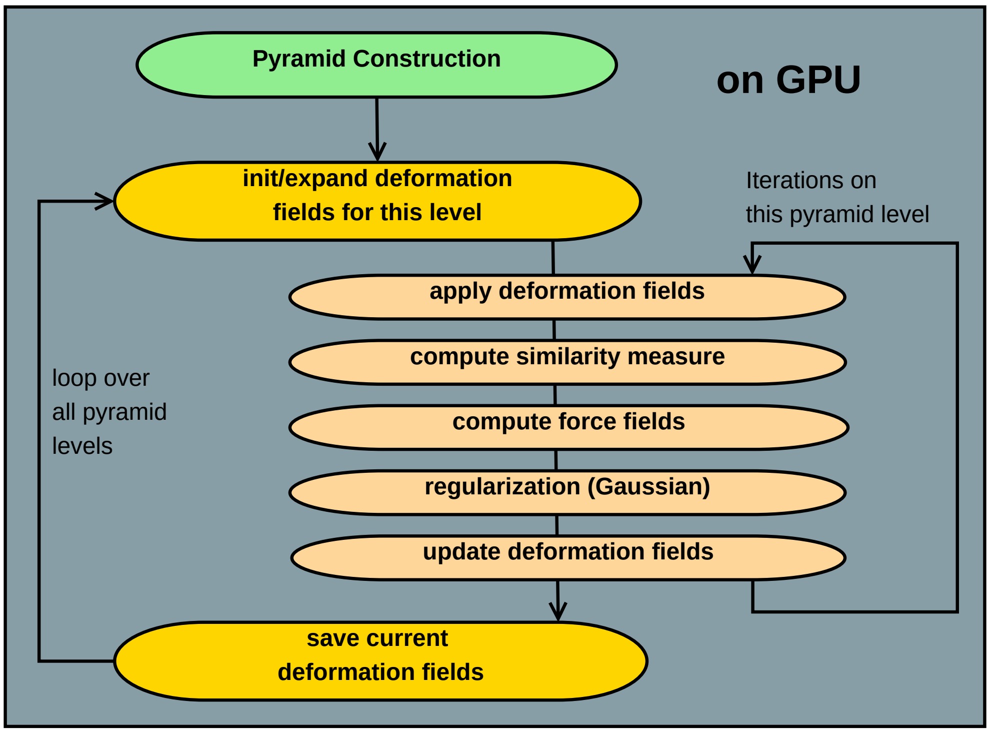

Pyramid Construction
on GPU
init/expand deformation fields for this level
Iterations on this pyramid level
apply deformation fields
compute similarity measure
loop over all pyramid levels
compute force fields
regularization (Gaussian)
update deformation fields
save current deformation fields

Figure 4.4: Schematic flowchart of nonrigid registration pipeline with important building blocks implemented on GPU.

In a first step the images are read from binary files, where each voxel is stored in 2 bytes with 2 bits precision. A corresponding xml configuration file provides information about the dataset and the acquisition geometry. This geometry is very important since the registration needs to be done in the world coordinate system (CSYS), while the data is stored in a data structure addressed in the voxel coordinate system.

Coordinate Systems

Image data resides in arrays that are either stored as cudaPitchedPtr which can be linearly addressed or stored as cudaArray which can be bound to a texture and allows for fast trilinear interpolation. For a given 3D coordinate (x, y, z) we find the linear array index as $i = z \cdot \#columns \cdot \#rows + y \cdot \#columns + x$, where x indexes the column, y the row and z the slice of the volume. This is considered as our *voxel coordinate system*. In order to consider the real geometry during the acquisition, we need a map between this coordinate system and the so-called *world coordinate system*. In the world coordinate system the euclidean distance between coordinate vectors is equivalent to the distance of these points during the acquisition in mm.

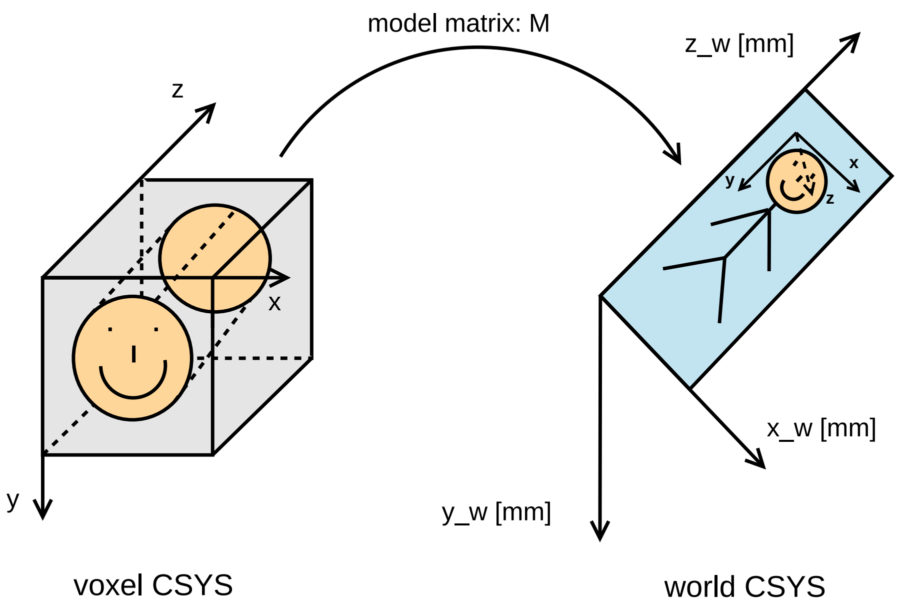


Figure 4.5: Coordinate Transformation between voxel CSYS and world CSYS.

We call the map between the voxel and world coordinate system *model matrix* M. Figure 4.5 shows a schematic presentation of the two coordinate systems. M is constructed by the information given in the configuration file. This file contains the translation of the origin in mm ($PatientPosition_{xyz}$ as t_{xyz}), the diameter of a pixel in mm ($PixelSpacing_{xyz}$ as Δ_{xyz}) and two normalized basis vectors of the world coordinate system ($PatientOrientation$ as v_{xyz}). By computing the third basis vector $v3$ of the two given orthonormal basis vectors as $v3 = v1 \times v2$

we can compute the model matrix M as

$$M = \underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{T} \underbrace{\begin{pmatrix} v1_x & v2_x & v3_x & 0 \\ v1_y & v2_y & v3_y & 0 \\ v1_z & v2_z & v3_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{R} \underbrace{\begin{pmatrix} \Delta_x & 0 & 0 & 0 \\ 0 & \Delta_y & 0 & 0 \\ 0 & 0 & \Delta_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{S} \tag{4.25}$$

where T is the transformation, R the rotation and S the scaling matrix. The whole registration process is performed in the common world coordinate system. This coordinate system provides meaningful distances, angles and orientations.

Multiresolution Pyramid

In a next step the image data is copied from host to device memory and a multiresolution pyramid is created for each image. We emphasize that this is the only time where data is transferred from host to device memory. The whole registration process is implemented on GPU using device memory.

The original image is the bottom of our pyramid. Higher pyramid levels are constructed by subsequently reducing the resolution of the levels. Since the in-plane resolution (in x,y direction) is often significantly higher than the number of slices we consider this ratio to decide whether we reduce the resolution by 2 in all dimensions or just in-plane (x,y). Dependent whether the resolution is reduced in-plane or not, we construct a so-called *mean-* or *average-pyramid* by averaging over four or eight voxels.

Multiresolution techniques come with two major advantages:

1. Building the average over several voxels behaves like a low-pass filter. It reduces noise and local extrema in the images. The resulting smoother cost function usually leads to more robust, however less accurate, registration results and higher capture ranges.

2. Registration on images with decreased resolution allows for significantly faster registration times. Often a few comparatively expensive iterations on higher resolution levels are sufficient to improve rough registration results obtained on lower resolutions to the desired accuracy.

Algorithm

Dependent on the configuration either the reference or the moving image is resampled by using trilinear interpolation to the resolution of the other image. The model matrix of the resampled volume needs to be adapted.

We start our algorithm on the highest pyramid level (lowest resolution) and initialize the displacement field with 0, what is equivalent to $\phi = id$. For subsequent pyramid levels we need to expand the last displacement field of the preceding level. We employed a trilinear interpolation scheme to evaluate the preceding displacement field of lower resolution.

According to the algorithm presented as Algorithm 3.45 we propagate the moving image g in the k-th iteration by the most recent displacement field u_k and obtain $g_k = g \circ \phi_k = g \circ (id + u_k)$.

Dependent on the employed similarity measure, we then evaluate the cost functional $\mathcal{J}_{SSD}, \mathcal{J}_{MI}$ or $\mathcal{J}_{LCC}$ for our reference image f and the propagated template image g_k. Within this step the intensity comparison function $L_{u_k}(f(x), g_{u_k}, x)$ is computed for every integral voxel $x \in \Omega$ according to the used metric. We refer to Eq. 3.16 ($L^{\text{SSD}}_{u_k}$), 3.24 ($L^{\text{MI}}_{u_k}$) and 3.40 ($L^{\text{LCC}}_{u_k}$). While the computation of $L^{\text{SSD}}_{u_k}$ is efficient and straight forward, the calculation of $L^{\text{MI}}_{u_k}$ and $L^{\text{LCC}}_{u_k}$ is computationally more expensive since it involves the computation of a joint histogram (MI) or several Gaussian filtering steps (LCC).

With this intensity comparison function L_{u_k} we can now easily compute the driving forces of the registration process, which are given by $\nabla \mathcal{J} = L_{u_k} \nabla g_{u_k}$. The differentiation of g_{u_k} is approximated by central differences with a step size equivalent to the pixel spacing in the corresponding direction of differentiation.

At the end of each iteration the displacement field needs to be updated. We can rewrite the last line of Algorithm 3.45 in the following way

$$u_{k+1} = \phi_{k+1} - id = (id + u_k) \circ (id + \epsilon_k \nu_k) - id = \epsilon_k \nu_k + u_k \circ (id + \epsilon_k \nu_k) \tag{4.26}$$

and obtain for the update of the displacement field the relation

$$u_{k+1}(x) = \epsilon_k \nu_k(x) + u_k(x + \epsilon_k \nu_k(x)) \tag{4.27}$$

where we used trilinear interpolation to evaluate u_k.

Chapter 5

Experimental Results

In order to classify this work, it is crucial to compare our implementation to existing solutions. Since we have access to a highly optimized parallel CPU implementation of the nonrigid registration algorithm presented in Chapter 3, we can provide this comparism. However, before we focus on the registration performance for practical nonrigid registration tasks in Section 5.3, we will present our test systems in Section 5.1 and investigate the individual performance of important building blocks in Section 5.2. We will close this chapter with statements on applied optimization techniques in Section 5.4 and general remarks on potential improvements and limitations.

5.1 Test Setup

For the experiments we used two different systems that are shown in Tab. 5.1. While System A is a wide-spread mid-range system with a dual core CPU and one of the first CUDA capable graphics card with compute capability 1.0, System B is a higher performant system. System B is equipped with 8 CPU cores and a Tesla C1060, which is a powerful graphics card dedicated to high performance computations on the GPU. This equipment allows for high performance registration on the CPU as well as on the GPU. By employing these two systems for practical registration tasks, we are able to get an impression of what can be achieved on affordable mid-range hardware as well as on more expensive systems that are in general not available.

We developed our CUDA registration framework on System A. This brings the huge advantage that our code runs on any CUDA capable device and is not dependent on a certain compute capability.

With one exception we used the same configuration (block and grid layout) for kernel launches

Table 5.1: Test Systems

	System A	System B
CPU	2 Intel Xeon @ 3,2 GHz	2 Intel Xeon E5420 QuadCore @ 2,5 GHz
Memory	3 GB	3 GB
GPU	NVIDIA GeForce 8800GTX 768 MB	NVIDIA Tesla C1060 4096 MB
	16 Multiprocessors	30 Multiprocessors
	Compute Capability 1.0	Compute Capability 1.3

on both systems. As described in Section 4.3, each thread block in the implemented algorithm for joint histogram computation utilizes the maximal 16 kB of available shared memory per multiprocessor. Thus not more than one block can run on a multiprocessor simultaneously. Because of this we configured the number of blocks according to the number of multiprocessors that are available on the corresponding test system.

We compiled our implementation with CUDA 3.1. As already noted, we developed our project on System A. This is important to keep in mind since minor performance and configuration optimizations might be useful on the GeForce 8800 GTX but useless or even disadvantageous on the Tesla C1060 of System B.

We used a CPU timer to measure the runtime of our GPU implementations. In this context we mention that many CUDA API functions are asynchronous. Thus the control might be returned to the calling CPU thread before the GPU kernels completed their tasks. To measure the elapsed time accurately we synchronized the CPU thread with the GPU immediately before starting and stopping the CPU timer. This is done using the CUDA function *cudaThreadSynchronize()*, which blocks the CPU thread until all GPU calls, which were started by this thread, are completed. For more details on the time measurement of GPU calls we refer to [NVI10b].

Since runtimes are often beyond several milliseconds, we averaged the computation times presented in Section 5.2 over 100 computations. Total registration times in Section 5.3 are averaged over 3 computations.

5.2 Performance of Certain Building Blocks

We will now present performance results of the most significant building blocks of our nonrigid registration pipeline which is based on the method introduced in Chapter 3 and described in Section 4.4.

As already mentioned, especially the Gaussian filtering is an often required tool. Not just for regularization by smoothing deformation fields but also for the computation of local statistics, provided that LCC is used. We will provide a detailed investigation of our parallel implementation of Deriche's approach (as presented in Section 4.2) in Section 5.2.2.

In Section 5.2.3 we will then determine the impact of the three novel optimization techniques on histogram computation, which we proposed in Section 4.3.

We mention that our CPU implementations, used in this section, run single threaded. However, the runtimes of the presented filtering techniques scale very well with the number of cores that are used. All benchmarks in this section were run on System A which is shown in Tab. 5.1.

The error analysis, conducted in Section 5.2.2, of the implementation of the recursive filter is based on the relative L1-error norm. For a scalar, non-zero reference value x and a scalar approximation $\tilde{x}$, of this as true considered value, this norm is defined as $\frac{|x-\tilde{x}|}{|x|}$. We calculate this error as $\frac{|x-\tilde{x}|}{|x|} \times 100\%$ in percent. In error calculations we will always provide the minimal, maximal and average L1-error.

5.2.1 Test Data

For the experiments in this section we generated test data of different resolutions, as follows:

HOMO_DATA:	A volume with a constant, homogeneous intensity value of 1.0.
RAND_DATA:	A volume with uniformly distributed random intensity values in $[0.0; 1.0]$.
EDGE_DATA:	A volume having repetitive 5 consequent slices with an intensity value of 1.0 and 5 consequent slices with an intensity value of 1000.0. This creates regions of constant intensity values separated by sharp edges.
SPIKED_DATA:	A volume created by modifying RAND_DATA with equidistant spikes of an intensity value of 1000.0. Spikes are created with a distance of 5 to each other in each dimension.

Volumes in host memory are always stored in a 1D array of floats. As data structure on the device we used either a *cudaPitchedPtr*, if no interpolation is needed (Filtering), or texture memory to allow for fast interpolation as it might be needed for the histogram computation.

In the experiments concerning joint histogram computation (Section 5.2.3) we also computed histograms on real datasets which are shown in Table 5.2.

Table 5.2: Real Registration tasks (REG), reference (R) and moving (M) images

Task	REG_BRAIN	
	R_BRAIN	M_BRAIN
Resolution	$44 \times 512 \times 512$	$52 \times 256 \times 256$
Modality	CT	MRI (T1)
Remark	images of Vanderbilt database	

5.2.2 Recursive Deriche Filter

In order to employ the recursive Gaussian filtering method presented in Section 4.2, we investigate the accuracy as well as the runtime of our GPU implementation.

Accuracy

In the following we compare the output of our recursive implementation to the numerical results obtained by a direct convolution with a Gaussian kernel with radius 10σ, where σ is the standard deviation of the Gaussian. We investigate the behavior on four different artificially created data sets of size 256^3, which are described in Section 5.2.1.

The minimal, maximal and average L1-errors of the performed experiments are shown in Figure 5.1. We observe that the accuracy of Deriche's recursive filter decreases with increasing σ on all data sets. This coincides with the results provided in [Hal06]. On HOMO_DATA as well as on RAND_DATA we determine very low average and maximal relative L1-errors. Even if these errors are increasing with σ, they stay in an order of 10^{-1} to $10^{-4}\%$. This is entirely sufficient for our applications. Errors on SPIKED_DATA are increased by roughly one order compared to the errors calculated on RAND_DATA. For higher σ the relative L1-error reaches several percent. Further experiments showed that this error is not directly influenced by the "height" of the spikes in the data. The high relative L1-error on EDGE_DATA for $\sigma = 1$ scales with the ratio of the intensity values in the region with higher intensity values and the region with lower intensity values. With an increased number of consequent slices with identical intensity values, this behavior is observed for higher choices of σ as well. We are not worried about these results since we will not filter data like this nor use $\sigma \geq 20$. However, we emphasize that a relative L1-error of several percent might be of importance in other applications. For the regularization there is also no theoretical need for an exact Gaussian filter.

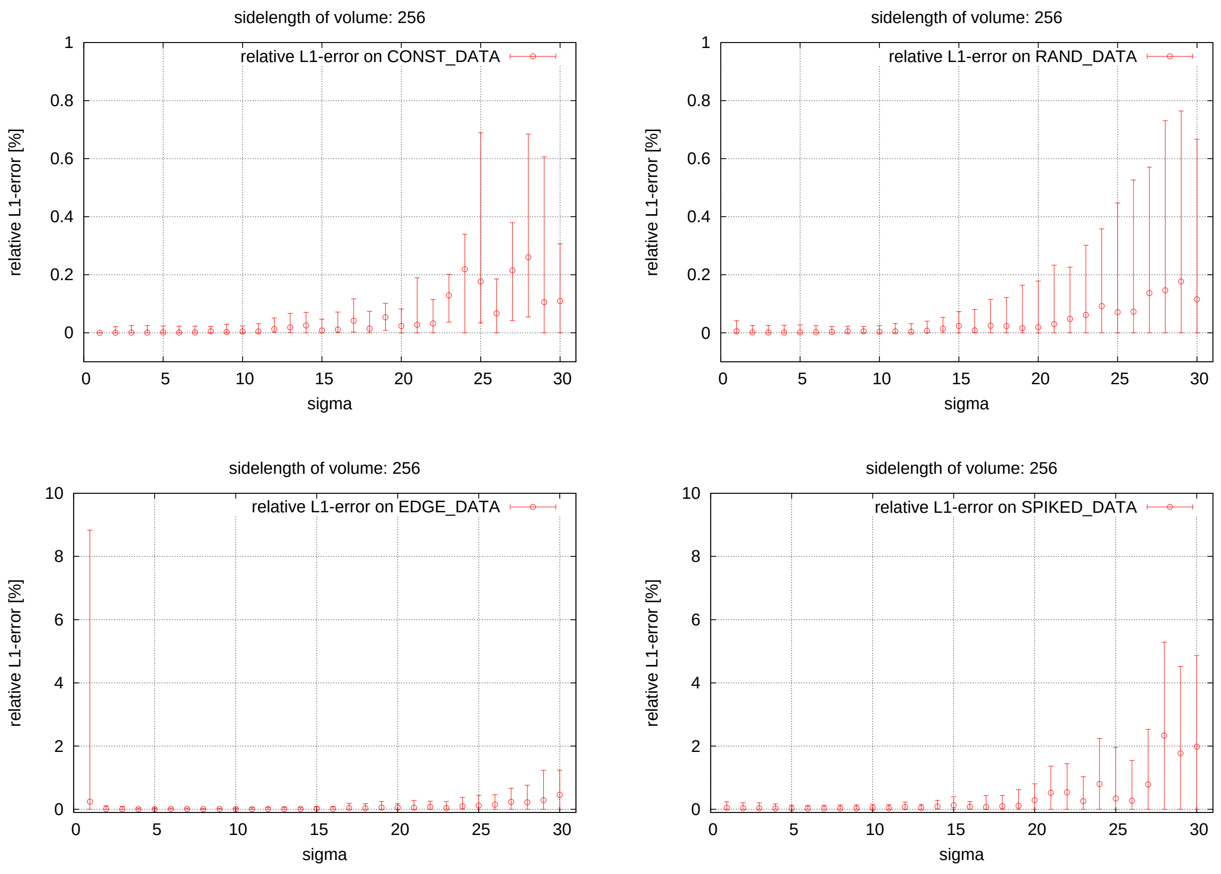


Figure 5.1: Relative L1-error of our parallel recursive Deriche filter compared to a direct convolution with a Gaussian kernel with radius 10σ.

A more severe problem for our application is that numerical tests revealed that new minima or maxima can be introduced to the image by applying the presented recursive filter. Even if the extent of this problem is covered by the low maximal relative L1-errors, we need to be aware of the possibility that this effect could result in a change of sign. Thus when filtering data with solely positive values, there is a chance that negative values emerge in the filtered data. This is usually nothing to worry about when the filter is employed as regularizer for the deformation field. But when local variances are estimated by Gaussian convolution as described in Section 4.1, it needs to be ensured that $\mathrm{Var}_{\mathrm{loc}}(X) \geq 0$. We encountered exactly this problem of negative variances, while using LCC as similarity measure, and present two solutions. One possibility is to employ a GPU implementation of a direct convolution, which obeys this MAX/MIN property, instead of the recursive filter. However, in order to benefit from the very fast and σ-independent recursive filter, we suggest to take the absolute values of the values resulting from the filtering. To avoid intra-warp branch divergence, caused by if-conditions, we realized this operation by computing $|x| = \sqrt{x^2}$. This has no observable influence on the performance of our GPU implementation. With the consideration that $|x - |\tilde{x}|| \leq |x - \tilde{x}| \ \forall x \geq 0$, it is also obvious that the relative L1-error is not increased by this operation. In the case $\tilde{x} < 0$ this error is even reduced.

Runtime

After investigating the accuracy of the recursive filter we will in this paragraph focus on the runtime of our implementation on System A.

In order to rate the results achieved with parallel solutions on the GPU, we compare our GPU implementations of the recursive and direct convolution filter to equivalent CPU implementations in a first test. In this test a kernel of half width 4σ is used for the direct convolution and the GPU computation times include the time required to copy the test data from host to device memory and vice versa. If Deriche's approach is compared to a direct convolution, we can easily count the theoretical necessary calculations to compute an output sample. As can be seen from Eq. 4.12 and Eq. 4.13, the computation of an output element by an 4-th order recursive filter requires, independent of σ, 16 multiplications and 15 additions for each dimension. In contrast to this $1+4\sigma$ multiplications and 8σ additions are necessary, if a filter kernel of half width 4σ is used for direct convolution. This relation can be roughly observed in Figure 5.2 for the CPU implementations, where recursive filtering outperforms the direct convolution for $\sigma \geq 2$ and performs almost independent of σ. That even the computation time of the recursive filter increases slightly with σ, is caused by the fact that every line of the volume is padded with $\max(5\lfloor\sigma + 0.5\rfloor, 10)$ elements to impose boundary conditions, as described in Section 4.2.2. However, the computation times on the GPU are influenced by other factors as well. For example the parallelizability of an algorithm and possibility of efficient memory access is more important than the sole count of arithmetic operations. Therefore these arithmetic considerations can not be applied to GPU programming in a straightforward way. As we see in Figure 5.2 the GPU implementations outperform both CPU implementations by far. The recursive GPU implementation performs, independent of σ, around 40 times faster than its counterpart on the CPU. As also shown, the computational cost of the parallel direct convolution on the GPU obviously depends on σ as well. However, compared to the CPU version, we observe a much better scaling of the runtime for direct convolution with

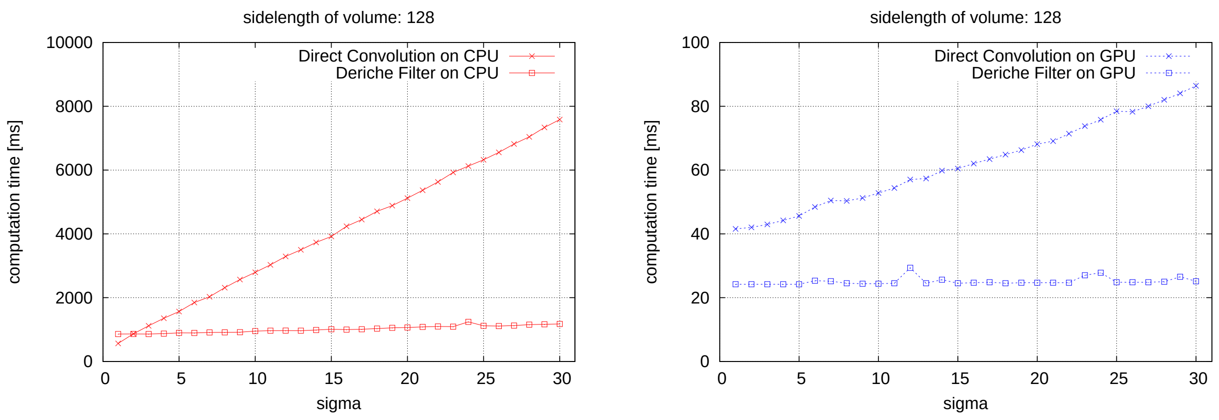
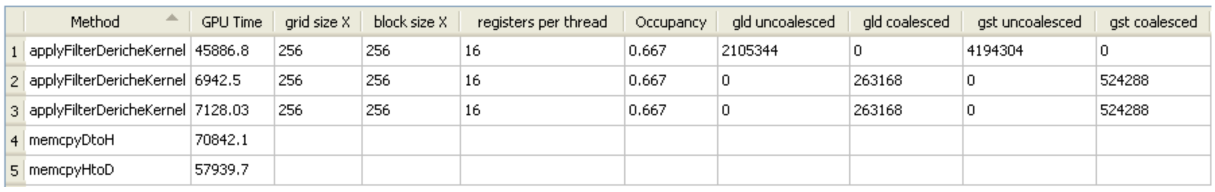


Figure 5.2: Comparism of runtimes for Direct Convolution and Deriche Filtering on System A. Volume size: 128^3



	Method	GPU Time	grid size X	block size X	registers per thread	Occupancy	gld uncoalesced	gld coalesced	gst uncoalesced	gst coalesced
1	applyFilterDericheKernel	45886.8	256	256	16	0.667	2105344	0	4194304	0
2	applyFilterDericheKernel	6942.5	256	256	16	0.667	0	263168	0	524288
3	applyFilterDericheKernel	7128.03	256	256	16	0.667	0	263168	0	524288
4	memcpyDtoH	70842.1								
5	memcpyHtoD	57939.7								

Figure 5.3: CUDA Visual Profiler Analysis for Deriche Filtering on System A. Volume size: 256^3

increasing σ on the GPU. We notice a 14 ($\sigma = 1$) to 88 ($\sigma = 30$) times faster direct convolution on the GPU. The parallel recursive filtering approach on GPU outperforms its counterpart on the CPU by a factor of roughly 40. The changing speed up factor for the comparison of direct convolution is explained by the fact that the two expensive tasks of the GPU implementation are constant and independent of σ. These tasks are to copy data between host and device memory and between global and shared memory. To copy $128^3 \times 4$ bytes data from host to device and vice versa takes for example approximately 16 ms. This is especially for the very fast recursive filtering significant, since the overall time for the recursive GPU approach was 24 ms, which is just slightly more. Neglecting the time needed to transfer data between host and device memory, the GPU implementation of the recursive filter outperforms its equivalent CPU implementation by a factor of at least 100.

As shown in Figure 5.3, which is an output diagram of the CUDA Visual Profiler for the recursive filtering, transferring data between host and device memory is very expensive in terms of runtime. In the illustrated example the time to copy $256^3 \times 4$ bytes data is with 128,7 ms more than 2 times higher than the required time for the actual filtering with 59,9 ms. We also see the importance of coalesced global loads (gld) and global stores (gst). As described in Section 4.2.2, we can realize coalesced memory access just for two filtering directions. Filtering in column direction requires expensive uncoalesced access to global memory, what results in a slow down factor of roughly 8 in this example.

Compiled with CUDA 3.1 our kernel is using 16 registers which yields a good occupancy of 0.666/1 on System A/B. We refer to [NVI10b] for more information about these characteristic parameters. As shown in the kernel summaries in Appendix B, our implementation yields an overall memory throughput of around 20 GB/s on System B which equals 20% of the theoretical possible (102 GB/s) throughput of the Tesla C1060 GPU. However, we recap that we optimized our implementation on System A. Employing optimizations which are specific for System B should allow for a more efficient filtering on System B. Apparently the higher occupancy on System B is disadvantageous for this kernel. Different general optimization approaches as employing shared memory failed, since all of them resulted in an implementation which was inflated by algorithmic overhead.

In Figure 5.4 the performance of both GPU implementations are opposed for different volume sizes and different choices of σ. To avoid comparing the computational overhead introduced by transferring data, we will now benchmark the filtering time for data residing in device memory. This is also reasonable, since there is no need to copy data between host and device memory in the presented registration pipeline.

Compared to the direct convolution, the recursive implementation performs faster for any choice of σ and any volume size. Nevertheless, it scales much better with increasing sidelenghts and especially σ, where the computation time is constant. For small sizes (sidelength < 64) and $\sigma = 1.0$ both approaches perform similar. The recursive approach performs up to 13 times faster for the largest tested volume (256^3) and σ (30.0). Even for small choices of σ it performs in general noticeably faster.

The rapid increase of computation time using direct convolution for certain image sizes is caused by the fact, that the block and thread configuration of the CUDA kernel depends on the image size. Since we focus on and employ the recursive implementation, which behaves well for increasing image sizes, we are not worried about this observation. However, it shows the potential dramatic influence of the kernel configuration on the computation time.

All conducted experiments lead to the conclusion that the in Section 4.2.2 presented 4-th

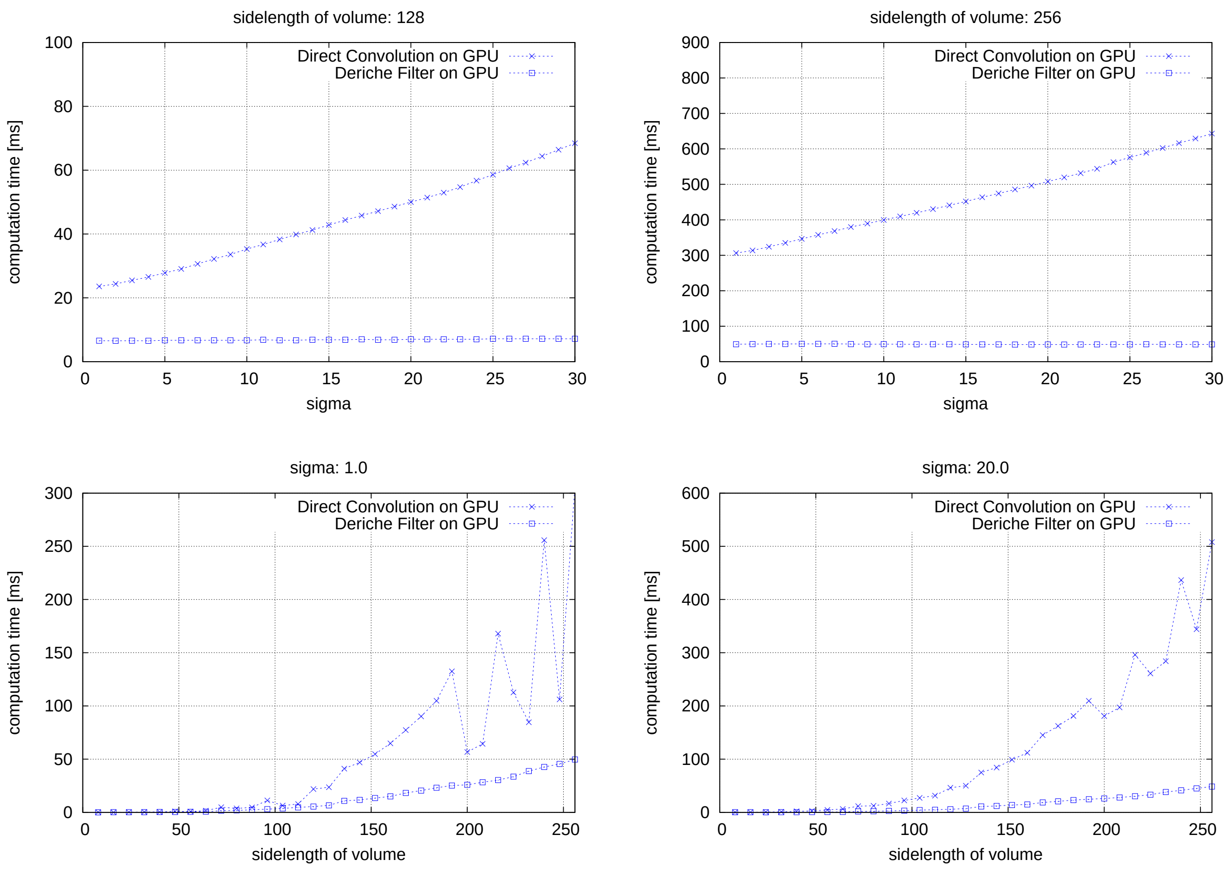


Figure 5.4: Comparism of runtimes for Direct Convolution and Deriche Filter on System A.

order recursive filter according to Deriche [Der93] is especially when implemented in CUDA a very powerful tool. We showed that this filter is, with some modifications for the computation of running averages, suitable for all our filtering purposes.

5.2.3 Joint Histogram Computation

We will conduct performance experiments from two different view points in this section. In the first part we will focus on the comparison of joint histogram computation on the CPU and on the GPU. Later on we will investigate the relevance of the novel optimization techniques, which we introduced in Section 4.3.

First of all we will compare joint histogram computation on the CPU to our highly optimized implementation on GPU. For the GPU implementation we employed the histogram algo-

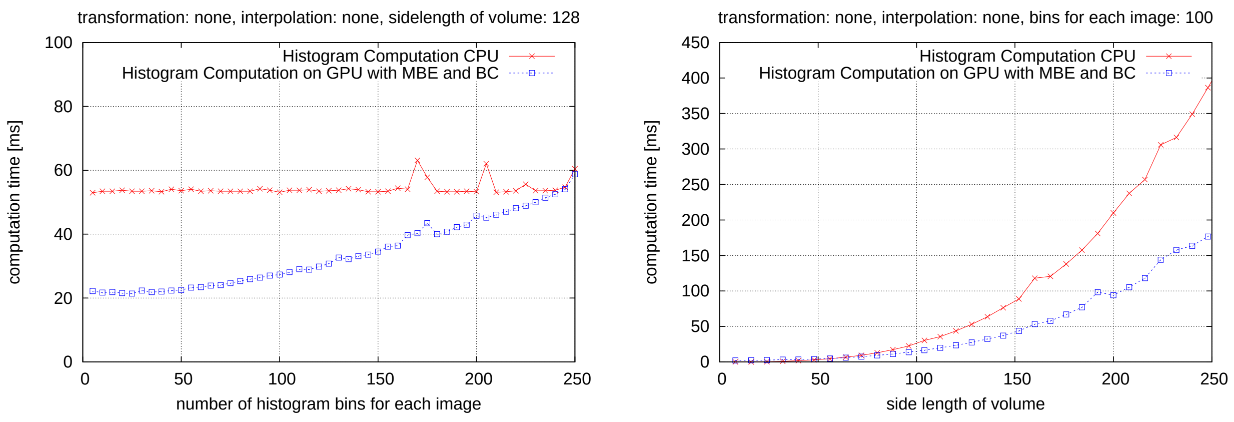


Figure 5.5: Comparism of runtimes for Histogram Computation on System A.

rithm, which was presented in Section 4.3.2 and enhanced by two complementary optimization techniques. These optimizations were presented as *MBE* and *BC* in Section 4.3.3. The CPU implementation is straightforward.

We compared computation times for a joint histogram calculation on two volumes of size 128^3 initialized according to RAND_DATA. Neither an interpolation scheme nor any transformation to the moving image was applied in these calculations. Since in our pipeline the moving volume is warped before the joint histogram is actually computed, this is exactly how we employed this building block for the computation of MI. The CPU and GPU computation times are compared in Figure 5.5, whereby the measured GPU time includes the required transfer time to copy both images from host to device memory. Even if the GPU implementation has an overhead of 17.0/131.9 ms to copy two volumes of size 128^3/256^3 from host to device memory, we observe a significant speedup of factor 2 for reasonable histogram sizes of 100×100. In our registration process the overhead caused by data transfer is redundant. Thus we will notice a speed up factor of roughly 5, for reasonable joint histogram sizes, in our application for this building block. Also for large bin numbers of 250 bins for each image the GPU implementation is faster.

We point out, that in all performed experiments the parallel GPU implementation yields within the limits of rounding errors the same joint histograms as the CPU implementation.

In the following we focus on the impact of the optimization techniques which we presented in Section 4.3.3. For this purpose we slightly broaden the scope. We will now employ interpolation schemes and apply affine transformations to the moving image. Even if this is in general not necessary in the presented nonrigid registration pipeline, these results are for example within the process of rigid preregistration of high interest. We will provide a comparison based on the runtimes of one of the fastest published joint histograms algorithms, which was presented as *Method1* in [Sha07b]. On the one hand side we will measure runtimes of this algorithm itself and on the other hand we will time this algorithm enhanced by the presented optimizations. We will also differentiate whether spare global memory is available or not.

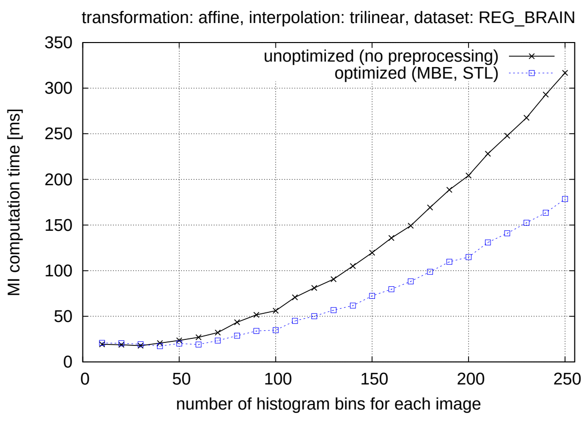
If global memory is not a limiting factor, we can facilitate a fast histogram computation by applying *Method1* to a new temporary image $I_c(\phi(\cdot), x)$ (Eq. 4.22). This image is generated in a preprocessing step according to Eq. 4.22. In this case (I) we will compare *Method1* to *Method1* enhanced by applying the optimizations *MBE* and *BC*, where both approaches require additional memory. In Section 4.3.3 we already pointed out that for joint histograms of sizes $\leq 256 \times 256$ bins, our proposed optimization *BC* saves 50% of the additionally required memory compared to the regular preprocessing step according to Eq. 4.22.

If global memory is short, which might occur when dealing with large volume sizes ($256^3 \times 4$ Byte $= 64$ MB, $512^3 \times 4$ Byte $= 512$ MB), histogram computation can neither be accelerated by creating a temporary image as shown in Eq. 4.22 nor by applying *BC*. Even 50% of an image size might be too much to ask for additional memory. In this case (II) we compare the unoptimized implementation of *Method1*, which is evaluating both images to obtain I_c within each kernel call, to *Method1* optimized by *MBE* and *STL*.

In general this entails, in order to get the resulting joint histogram bin of a voxel pair, that two images need to be evaluated within each kernel call. This includes the application of a transformation to the moving image as well as the employment of more or less expensive interpolation schemes. All these tasks need to be performed only once, if the result can be cached in additional memory. Therefore additional memory is highly desirable for a fast histogram computation.

We calculate MI in three steps: joint histogram computation, normalization of the histogram ($\sum_{i,j} J(i,j) = 1$) and the actual computation of MI. For the computation of the marginal entropies $H(X)$ and $H(Y)$ an adapted implementation of NVIDIA's high efficient reduction algorithm [Har07] is employed. The normalization together with the computation of MI takes 0.19/0.14 ms (on System A/B resp.) for a joint histogram with 100×100 bins. This cost is negligible compared to the joint histogram calculation.

Experiments on REG_DATA revealed that for bin ranges above 140×140 the optimized algorithm with *tricubic* interpolation is faster than the unoptimized algorithm with *trilinear* interpolation (if additional memory is available). This could allow for a more accurate registration by using higher-order interpolation schemes *while* keeping computation times reasonable.



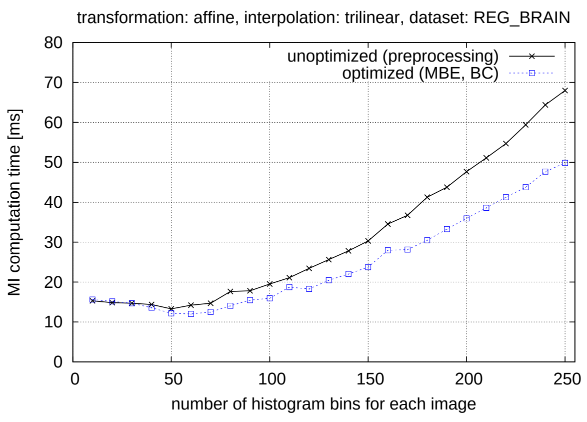


Figure 5.6: Mutual Information computation times using different methods on REG_BRAIN without (left) and with (right) additional memory available on System A.

Figure 5.6 illustrates the improvements achieved with the presented optimizations for different bin numbers on the REG_BRAIN dataset.

Both implementations, which are not using additional memory and are illustrated in the left diagram, perform clearly slower than the ones using additional memory. Nevertheless our presented optimizations contribute significantly to a fast MI computation. *MBE* combined with, dependent on the availability of global memory, *STL* or *BC* outperforms the unoptimized *Method1* for all investigated scenarios considerably.

See also Tab. 5.3 for an overview over the resulting average reduction in computation times on System A and B.

Table 5.3: Comparison of MI computation times with optimized/unoptimized algorithms on RAND_DATA/HOMO_DATA and realistic REG_BRAIN datasets. Average computation time reduction on System A/B. Joint histogram size: bins$\times$bins, Transformation: affine

	MBE & BC, (I)		MBE & STL, (II)	
	(trilinear)	(tricubic)	(trilinear)	(tricubic)
side length of volumes: 256 bins: 10 to 250, RAND_DATA	19.2/24.4%	15.4/28.5%	28.8/28.6%	40.9/41.9%
side length of volumes: 16 to 256 bins: 100, RAND_DATA	19.8/21.7%	14.1/24.1%	35.1/32.8%	44.7/43.5%
side length of volumes: 16 to 256 bins: 100, HOMO_DATA	10.9/13.6%	10.7/19.6%	33.0/31.6%	53.0/52.1%
REG_BRAIN datasets bins: 10 to 250	17.3/22.7%	16.0/28.3%	31.4/30.9%	45.9/46.6%

We point out, that we observed a significant reduction in runtime not just for generated artificial data, but especially for the realistic REG_BRAIN datasets of the Vanderbilt database [Wes97].

Regarding *MBE*, due to algorithmic overhead and more frequent global memory updates to avoid overflows, no further performance improvements were observed by encoding more than two bins in the 27 less significant bits.

The conducted performance experiments reveal that CUDA allows for a significant faster histogram computation on the GPU than on the CPU. We demonstrated that for our application a speed up of factor 5 for joint histogram computation can be expected. This is achieved by optimizing a recent joint histogram algorithm (*Method1* in [Sha07b]) by applying novel optimization techniques (*MBE* & *BC*), which we introduced in Section 4.3.3.

With this highly optimized joint histogram computation, we have an accurate and very fast implementation in place to compute joint histograms. Histogram computation is a crucial and time consuming building block, if we employ MI as similarity measure.

5.3 Registration Performance

In this section we will investigate the overall performance of the presented registration pipeline realized in parallel on the GPU compared to a parallel CPU implementation using all available cores on the respective test system. For the conducted performance experiments we employed common and identical (on CPU and GPU) iteration schemes to tackle different registration tasks, which are shown in Tab. 5.4.

Table 5.4: Registration tasks (REG), reference (R) and moving (M) images

Task	REG_HEAD		REG_LUNG128	REG_LUNG256
	R_HEAD	M_HEAD	R/M_LUNG128	R/M_LUNG256
Resolution	$49 \times 512 \times 512$	$52 \times 256 \times 256$	$128 \times 128 \times 128$	$256 \times 256 \times 256$
Modality	CT	MRI (T2)	CT	CT
Patient	different		same	
Remark	images of Vanderbilt database		lung in inspiration/expiration (R/M)	

Since objective error evaluation is a very challenging task for nonrigid registration problems [Cru03], we restrict the scope of this thesis to the analysis of computation times and visual registration results.

We will oppose overall registration times on the GPU/CPU obtained on our test systems (Tab. 5.1) for different configurations. We employed the recursive Deriche filter as presented in Section 4.2.2 for the regularization and the estimation of local statistics. Joint histograms, required for the calculation of MI, are computed by an improved implementation of an algorithm (*Method1*) of Shams et al. [Sha07b], which we presented in Section 4.3.2. We optimized this algorithm by two essential optimization techniques, which were introduced as *MBE* and *BC* in Section 4.3.3.

First of all we will compare overall registration times and average iteration times on different pyramid levels in Section 5.3.1. Next to this we will investigate the contribution of different building blocks to the overall process by analyzing the runtime of different CUDA kernels in Section 5.3.2. How many iterations are actually required to obtain desirable registration results is data and application dependent. We will show in Section 5.3.3 that often just a few iterations on lower pyramid levels can be sufficient to recover significant misalignments. A memory footprint of our algorithm is given in Section 5.3.4.

5.3.1 Performance on Real Data

In order to compare computation times of all presented similarity measures (SSD, MI and LCC) we investigate runtimes on the monomodal REG_LUNG256 registration task, which can be found in Tab. 5.4, where all similarity measures lead to reasonable registration results. We will present results for both test systems. As investigated in Section 5.3.4, the memory requirement of the algorithm depends on the input data and the iteration scheme. Because of this limitation no iterations can be performed on the highest resolution level for REG_LUNG256 on System A. We registered this dataset by using iteration scheme [64,32,16,8,0] on System A and [64,32,16,8,4] on System B. This means 64 iterations on resolution 16^3, 32 iterations on resolution 32^3, 16 iterations on 64^3, 8 iterations on 128^3 and 0 or 4 iterations on the full resolution 256^3. The registration results highly depend on the iteration schemes and on the characteristics of the employed filters. The standard deviations of the Gaussian filter used for the regularization (σ) and for the estimation of local statistics (β) are crucial parameters. However, since the runtime of our recursive implementation according to Deriche is independent of the standard deviation, the overall registration time with a certain iteration scheme does not depend on the choice of these parameters.

Figure 5.7 shows the visual registration results achieved with LCC and $\beta = 1.0$ for the estimation of local statistics. While the alignment of tissue and the ribs looks quite reasonable we notice an extension of the spine. This can be tackled by for example shifting iterations from lower to higher resolution levels. However, the registration of other significant landmarks might suffer with a changing iteration scheme. The choice of parameters is usually done intuitively and by trying different settings. We did not optimize these parameters explicitly for the presented registration tasks. We set the regularization parameter $\sigma = 2.0$ for all similarity measures. The registration results on this monomodal dataset look similar for all three similarity measures. For the computation of MI we used 80×80 bins.

In Tab. 5.5 we compared the CPU to the GPU implementation on System A and in Tab. 5.6 respectively for System B.

The time noted as “pyramid construction” includes the time needed to transfer the reference and moving image to the device memory. However, the time to load the images from the hard drive to host memory and save the deformation fields on it is not included. We emphasize that we observed for any setting significant reductions in registration times by a factor from approximately 5 on System B to 10 on System A. The lower speed up factor observed on System B is mainly caused by the fact that the CPU of System B has 4 times as many cores than System A, while the Tesla C1060 has “only” twice as many multiprocessors as the GeForce 8800 GTX and might also suffer from memory limitations. Again we point out that these total registration times include all data transfers between host and device memory and represent thus the actual computation times needed to obtain the three resulting deformation fields in host memory.

We regard the encountered speed up factors of up to one order compared to high performant parallel CPU implementations as vast. Especially the fact that even the low-end GeForce 8800 GTX outperforms the 8 core CPU of System B clearly by a factor of roughly 3 needs to be emphasized.

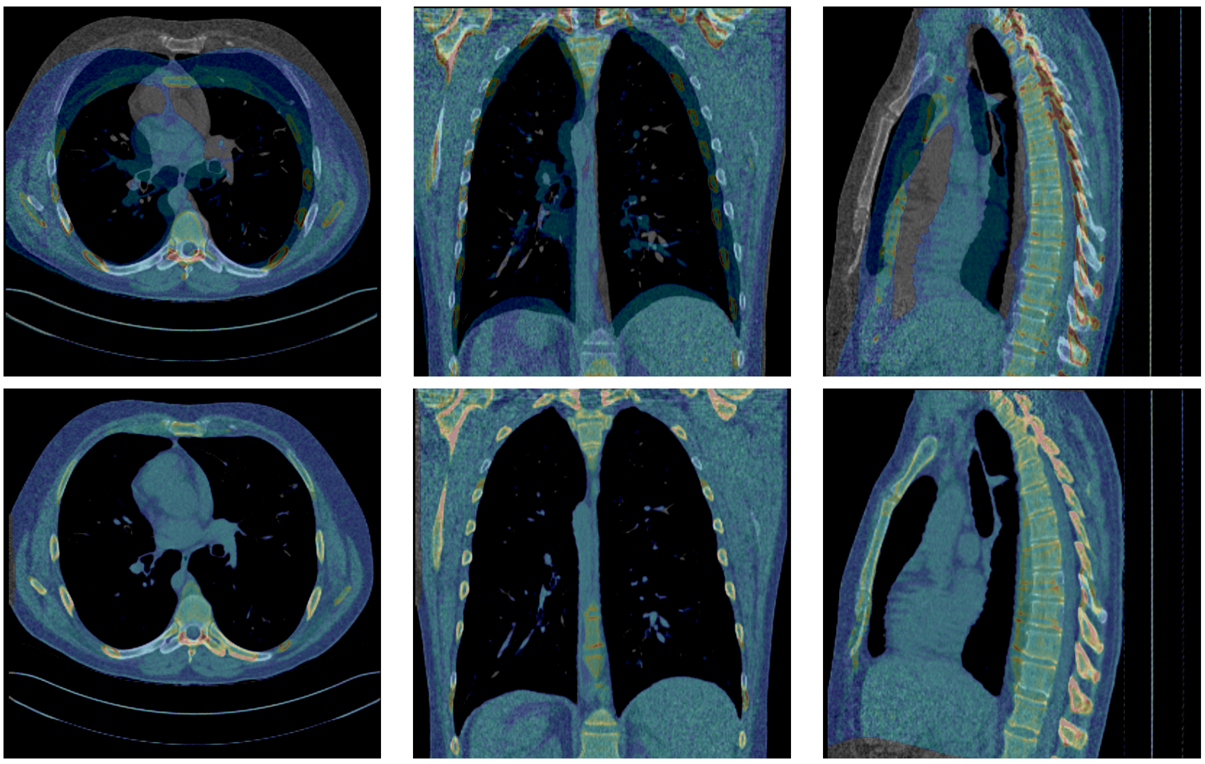


Figure 5.7: Nonrigid Registration Results achieved with LCC on REG_LUNG256 and iteration scheme [64,32,16,8,4] on System B. Upper row: unregistered. Lower row: registered. From left to right: view on axial, coronal, sagittal plane.

Table 5.5: Comparison of nonrigid registration times for REG_LUNG256 on System A. Time for pyramid construction for both images [ms]. Average iteration times on each pyramid level [ms]. Time to transfer 3 deformation fields of size 256^3 from device to host memory.

	CPU A			GPU A		
	SSD	MI	LCC	SSD	MI	LCC
pyramid construction [ms]	108			132		
avg. iteration time on 16^3 [ms]	3	5	6	1	2	2
avg. iteration time on 32^3 [ms]	21	23	35	2	3	4
avg. iteration time on 64^3 [ms]	152	182	235	8	11	16
avg. iteration time on 128^3 [ms]	1154	1378	1837	46	52	103
avg. iteration time on 256^3 [ms]	–	–	–	–	–	–
transfer of 3 def. fields (256^3) [ms]	0			292		
total reg. time with [64,32,16,8,0]	17.02s	19.77s	24.41s	1.69s	1.78s	2.33s

Table 5.6: Comparison of nonrigid registration times for REG_LUNG256 on System B. Time for pyramid construction for both images [ms]. Average iteration times on each pyramid level [ms]. Time to transfer 3 deformation fields of size 256^3 from device to host memory.

	CPU B			GPU B		
	SSD	MI	LCC	SSD	MI	LCC
pyramid construction [ms]	32			76		
avg. iteration time on 16^3 [ms]	1	2	3	1	1	1
avg. iteration time on 32^3 [ms]	6	8	13	1	3	3
avg. iteration time on 64^3 [ms]	40	47	73	6	7	12
avg. iteration time on 128^3 [ms]	282	351	532	35	43	87
avg. iteration time on 256^3 [ms]	2211	2828	4185	367	399	1042
transfer of 3 def. fields (256^3) [ms]	0			182		
total reg. time with [64,32,16,8,4]	13.77s	17.10s	24.58s	2.57s	2.88s	5.92s
total reg. time with [64,32,16,8,0]	4.90s	5.65s	7.89s	1.13s	1.24s	1.79s

5.3.2 CUDA Visual Profiler Analysis

The visual profiler tool which comes with the CUDA Toolkit is a very powerful tool to investigate the contribution of single CUDA kernels to the overall runtime. We show these profiler results for the most contributing kernels for SSD, MI and LCC in Figure 5.8. The applied iteration scheme [64,32,16,8,4] results in 124 iterations in this experiment on System B.

Independent of the employed metric the deformation fields need to be regularized by applying the in Section 4.2 presented recursive filter. This filtering is with a contribution of more than 50% clearly the dominating factor in terms of computation time. If LCC is employed as similarity criterion, a Gaussian is used to estimate local statistics and the contribution of the Deriche filter is with 80% even higher.

All three footprints also reveal a significant contribution of the DeviceToArray copy. This transfer between global and texture memory is necessary since there is no write access to texture memory. Each time texture memory needs to be modified this copy process is performed. We refer to Section 5.4 for a further explanation of this effect.

In case of MI, the importance of an efficient joint histogram computation shows up. In the corresponding diagram in Figure 5.8 we see that the kernel JointHistogramKernel is called 5 times in each iteration, resulting in 620 total invocations, to compute subsequently the full joint histogram of size 80×80 bins. Even if this task is computationally complex the applied optimizations push the contribution of this kernel below 10%.

Overall these profiler outputs emphasize the importance of optimizing Deriche filtering and joint histogram computation. In this thesis we investigated the efficient realization of both building blocks.

We point out that the contribution of other kernels to the overall registration time is of minor relevance while filtering is particularly dominant in terms of computation time. In order to reduce runtimes, the focus should first of all be on optimizing the recursive filter. Since our implementations and configurations were optimized for System A the filtering on System B could be further accelerated by applying different optimizations which are specific for System B.

A more detailed summary of all participating CUDA kernels, which is also showing the global memory throughput, can be found for each similarity measure in Appendix B.

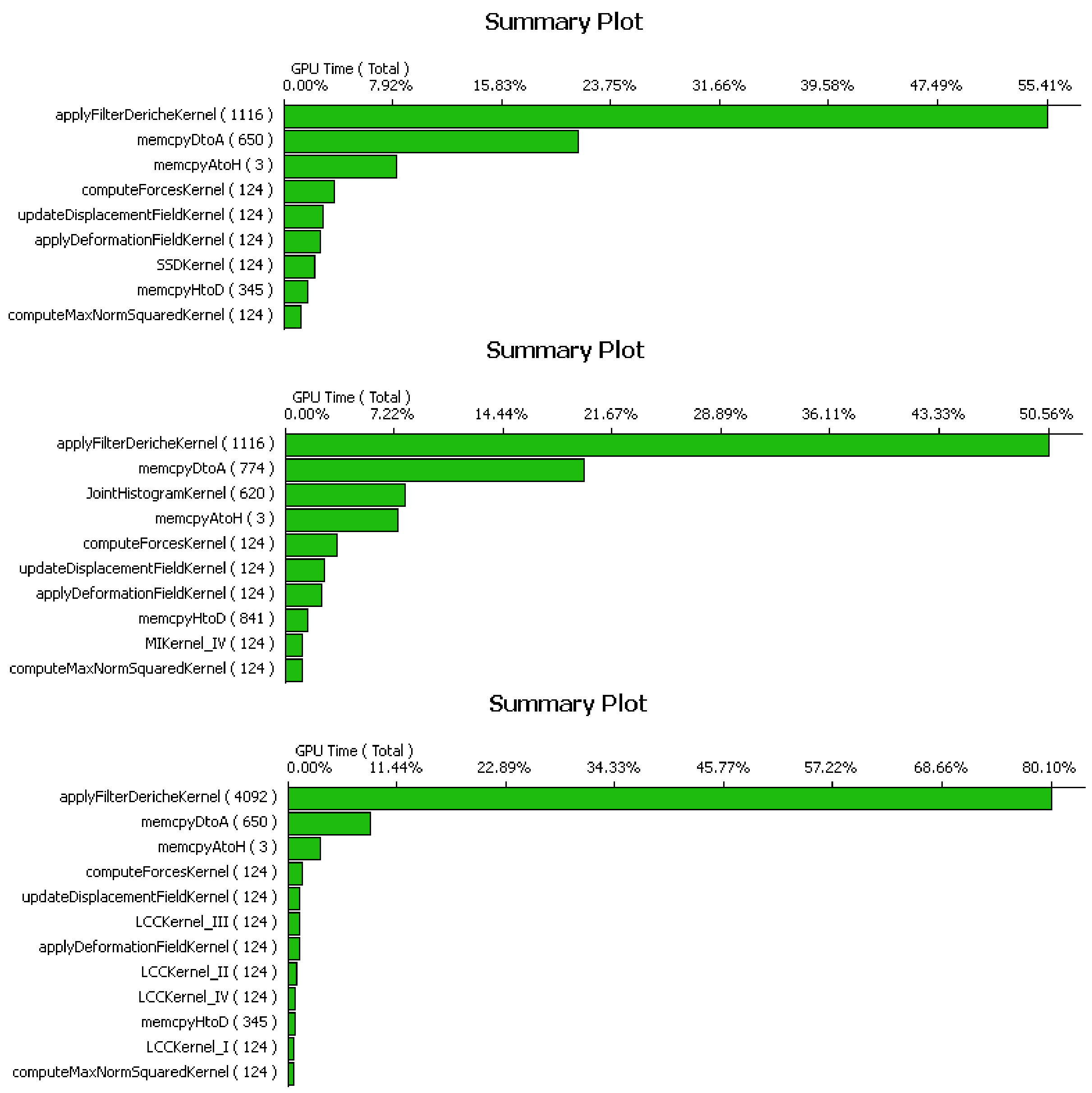


Figure 5.8: CUDA Visual Profiler Analysis for the nonrigid registration pipeline using (from top) SSD, MI and LCC on System B. Volume size: 256^3

5.3.3 Real Time Registration

Within real-time constraints, as they arise in interventional applications, registration time might be more relevant than good accuracy on a dataset of high resolution. Often truncated iteration schemes applied to data of lower resolution provide sufficient registration results.

We investigated the possibility of real-time registration by using the reduced iteration scheme [32,16,8,0] on the lung datasets REG_LUNG128 of a reduced resolution of 128^3.

The visual results, which look rather satisfying and were obtained with SSD as similarity measure, are shown in Figure 5.9. The computation times for different similarity criterions measured on both test systems are compared in Tab. 5.7. Again we encountered a significantly accelerated registration process on both systems. Interesting in this context is again the fact that the low-end GeForce 8800 GTX in System A outperforms the two QuadCore processors in System B clearly. We also mention that registration times are limited downward by the time needed to transfer data between host and device memory and initializing the CUDA drivers ($\approx$ 100 ms). Taking into account that these processes might not be necessary for subsequent registration tasks, online and real-time registration with a few frames per second is accessible.

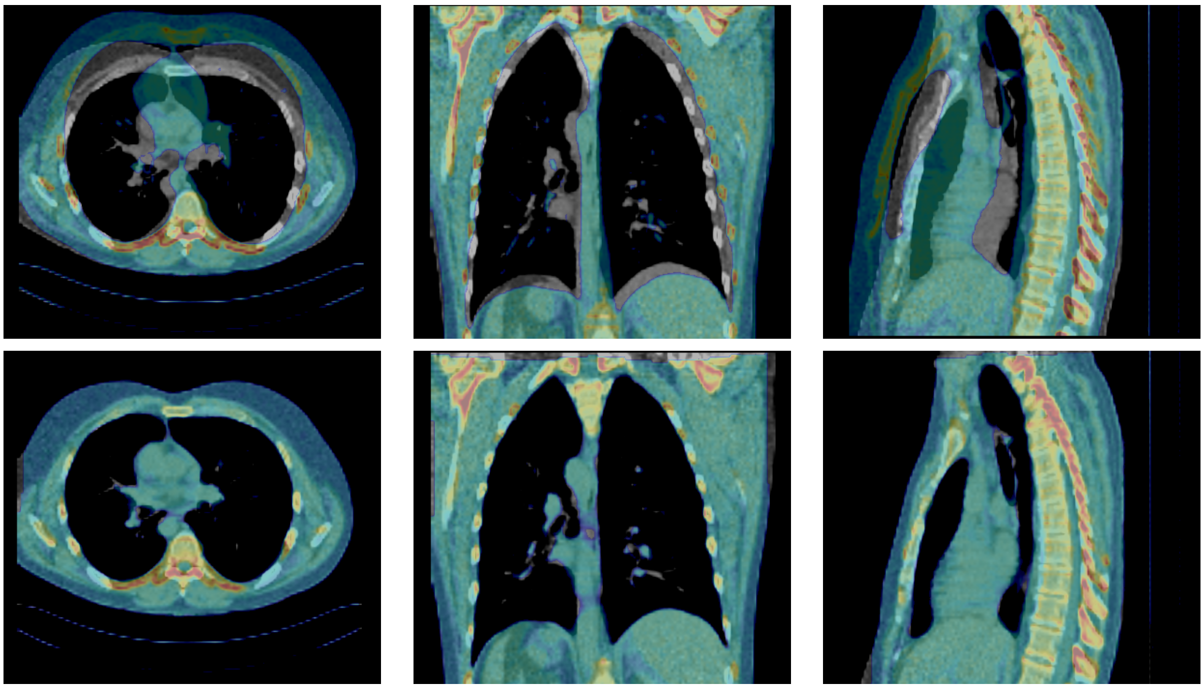


Figure 5.9: Nonrigid registration results achieved with SSD on REG_LUNG128 and iteration scheme [32,16,8,0] on System B. Upper row: unregistered. Lower row: registered. From left to right: view on axial, coronal, sagittal plane.

Table 5.7: Comparison of nonrigid registration times for REG_LUNG128 on System A and B.

	CPU			GPU		
	SSD	MI	LCC	SSD	MI	LCC
total reg. time with [32,16,8,0] on System A	2.42s	2.90s	3.47s	0.52s	0.63s	0.69s
total reg. time with [32,16,8,0] on System B	0.82s	0.92s	1.26s	0.33s	0.38s	0.47s

5.3.4 Memory Footprint

Available memory on the graphics card can be a limiting factor for our registration algorithm. Instead of providing a detailed analysis we restrict ourself to the practical relevant overall maximal memory requirement for certain registration tasks. The memory demand for different tasks of different resolutions is shown in Tab. 5.8. In this summary we can see immediately why REG_LUNG256 can not be registered on System A using iterations on the pyramid level of highest resolution. With 768 MB the memory of the GeForce 8800 GTX is insufficient. The actual demand depends on the resolution of the registered images and the employed similarity metric. We can register high-resolution data even if memory is short by neglecting iterations on high pyramid levels. This is shown in the last row of Tab. 5.8 where no iterations are performed on the 256^3 level. We mention that the actual memory demand can be roughly computed as 13 image sizes. At the highest pyramid level we need to store 2 images, 1 warped image, 3 deformation fields, 3 force fields, 1 intensity comparison function and another 3 temporary fields for intermediate computations in the device memory.

Table 5.8: Memory demand in MB for SSD, LCC and MI for registration tasks of different resolutions.

Size of reference image	SSD & MI	LCC
$128 \times 128 \times 128$ (8 MB)	108	116
$192 \times 192 \times 144$ (20 MB)	274	294
$256 \times 256 \times 107$ (27 MB)	375	402
$256 \times 256 \times 256$ (64 MB)	865	930
no iterations on highest level	481	481

5.3.5 Multimodal Registration

We registered the multimodal dataset REG_HEAD as shown in Tab. 5.4 to illustrate the difference, possibilities and also limitations of the presented similarity measures in a multimodal case. While SSD clearly fails, LCC and especially MI leads to visually better results. Again, the results depend directly on the registration parameters, which were not optimized for this dataset. The presented case is mainly constructed for validation purposes and to show the functionality of our registration pipeline. However, tasks like this might appear in a similar setup for atlas construction.

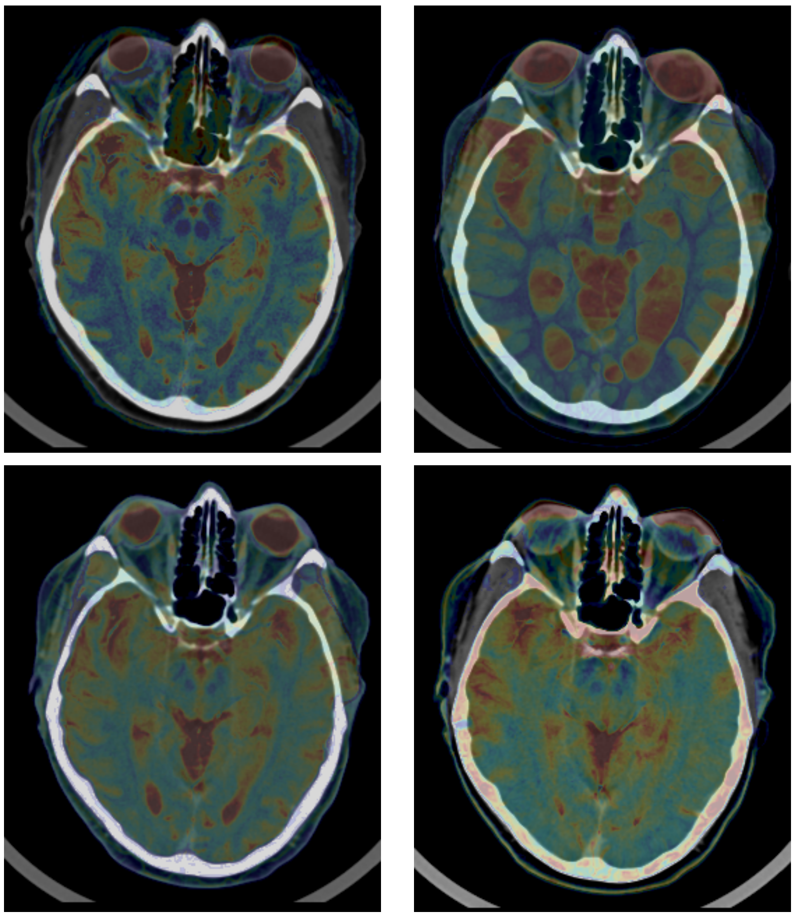


Figure 5.10: Nonrigid registration results achieved with SSD, MI and LCC on REG_HEAD and iteration scheme [30,30,30,30] on System A. From left to right and top to bottom: Rigid preregistered, nonrigid SSD, nonrigid MI, nonrigid LCC.

5.4 Discussion of Optimization and Performance Aspects

Throughout our implementations we obeyed important “high-priority” optimization strategies as presented in [NVI10b]. Whenever possible we attempted to employ the fast shared memory and reduced global memory access to a minimum. Furthermore we placed value on the occupation of the multiprocessors with sufficient threads and avoided branch divergence, whenever possible.

Before the actual registration process can be started, the reference and moving volume are copied from host to device memory. Since both images are stored as binary files having 2 bytes per voxel this procedure is rather fast compared to the transfer of the deformation fields, which are copied from device to host memory after the registration algorithm has finished. These three resulting scalar deformation fields (one for each dimension) are stored with 4 byte floating point precision and have the same dimension as the moving image. During the whole registration process, there is no significant data transfer between host and device memory. Exceptions are scalar values as for example the evaluation of a cost function during an iteration.

Figure 5.11 (left diagram) illustrates the required time to copy data between cudaArrays and host memory on System A. We emphasize that this host-device data transfer is a significantly contributing factor to the overall runtime of the registration process. This transfer takes for example for the three resulting deformation fields of size 256^3 more than 200 ms on System A.

Even if data transfer between a cudaPitchedPtr and a cudaArray on the device itself is less time consuming, it contributes as already mentioned in Section 5.3.1 to the overall registration time significantly. The required time for this data transfer is shown in the right diagram of Figure 5.11. Since it is not possible to perform write operations on texture memory, a reinitialization of this memory is performed each time texture memory needs to be modified. In our implementation we employ this copy at least 5 times in each iteration. By applying the current deformation fields to the moving volume we create the current warped volume in each iteration. Finally we modify the three deformation fields at the end of each iteration. Because of the fact that we want to benefit from a fast trilinear interpolation in the warped image and in the deformation fields the use of texture memory is essential at this point. Caused by the design of our pipeline, we also employed texture memory to store the values of the intensity comparison function.

We computed a throughput for the data transfer between device and texture memory of roughly 3.5 GB/s on System A, which is one order slower than using a CUDA kernel to copy between two device arrays. This is poor compared to the theoretical bandwidth of 86.4 GB/s and a known bottleneck. Improvements with upcoming CUDA versions are of high interest.

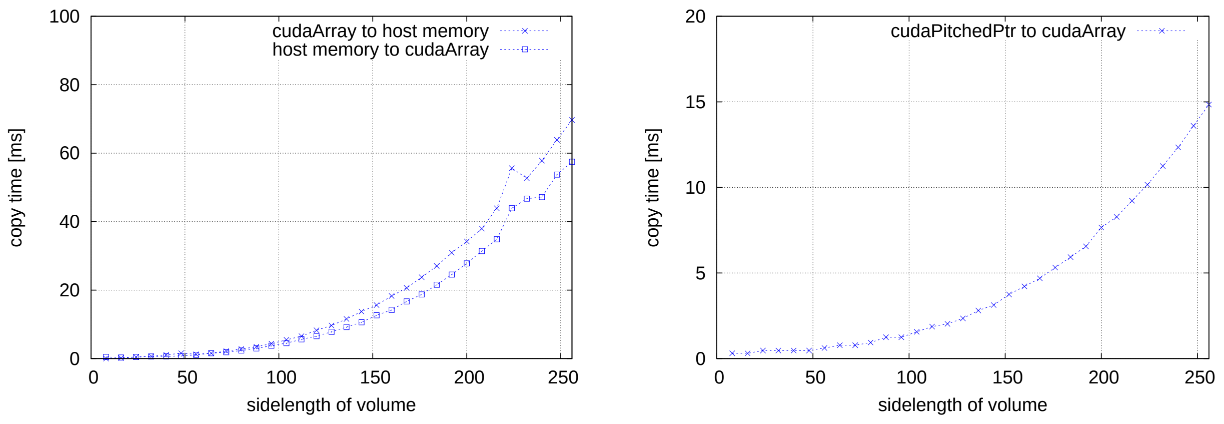


Figure 5.11: Required time to transfer data between host memory and cudaArrays (left) and cudaPitchedPtr and cudaArrays (right) on System A.

At this point we share another experience that we made during the development and the conducted experiments. We observed that the performance of a certain kernel is highly dependent on the configuration. How many blocks and threads can actual run in parallel on one multiprocessor depends on limitations imposed by a variety of parameters. Most important in this context are for example the compute capability of the device and kernel dependent parameters as the threads per block, the registers per thread and the amount of shared memory used by each block. These characteristics depend partly on the device (compute capability) and partly on the kernel launch configuration (threads per block). Taking into account that different CUDA compiler versions yield different register counts for one and the same kernel, the runtime behavior might also change significantly with different CUDA versions. A very helpful tool for this optimization process is available with the “CUDA Occupancy Calculator”, which can be found on the NVIDIA website [NVI10a]. For more technical background on the optimization of CUDA kernels we refer to the NVIDIA CUDA Programming [NVI10c] and Best Practices [NVI10b] Guide.

In Section 5.3 we have shown that filtering and joint histogram computation are next to data transfers the most time consuming tasks in the presented registration pipeline. This is why we presented efficient approaches for these operations in Section 4.2 and 4.3 and investigated the suitability of these building blocks thoroughly in Section 5.2. As shown in Section 5.3.2 the runtime of all other kernels is of minor importance in terms of overall runtime. By employing parallel commodity GPU hardware, we observed a vastly accelerated registration by a factor of up to 10 compared to parallel CPU implementations.

Chapter 6

Extended and Future Work

Within this work we also implemented and tested different filtering techniques and interpolation schemes. The scope of this document does not allow to present all these results in detail. However, we want to provide first and rough results for an edge presenting filtering approach and a tricubic interpolation scheme. Next to this we will present a new similarity criterion, which needs to be investigated in the future. We will start this section with general starting points for future research and engineering.

6.1 General Starting Points

As already mentioned we developed and optimized the employed kernels on System A. Optimizing our implementation for the higher compute capability on System B should result in a further decrease in computation time on the Tesla C1060 graphics card. We raised the subject of device (compute capability) and environment (CUDA version) specific kernel optimization already in Section 5.4. Optimizing the computationally most expensive kernels in this context is certainly useful in terms of runtime but might be tricky in practice. Since the kernel configuration often depends on the image size it is also important to ensure efficient computations for any given sizes. This effect was shown in Figure 5.4 where the unoptimized Direct filter performs unreasonably bad for certain image sizes. This also needs to be considered when kernels are optimized for other systems.

Another general starting point for future research is to further investigate the realization of the presented recursive filter according to Deriche. Even if the algorithmic instruction count could be reduced by employing a recursive approximation of lower order, we think that not the instruction count but first of all the uncoalesced memory access is something to focus on. Especially the unused shared memory, which is a powerful feature of CUDA capable GPUs, could enable innovative solutions. However, all of our approaches that were based on shared memory yielded comparatively slow computation times. All our implementations utilizing shared memory suffered from algorithmic overhead or other throughput decreasing factors.

Also further optimizations of the pipeline itself and novelties available with more recent or future GPUs are of interest. For example, for now local statistics of the reference image are estimated in every iteration. If memory is not a limiting factor this could be redundant by caching these results. Next to this, potential improvements for the histogram computation enabled by atomic updates, which are available with higher compute capabilities, need to be investigated. Possibilities to speed up the slow transfer between global and texture memory are of particular interest.

The profound validation of the registration accuracy is something that was missed out in this work. However, in order to employ the presented pipeline in a clinical environment this is an essential assignment which needs to be done in future work.

6.2 Tricubic B-Spline Interpolation

Since data can only be stored discretely on the computer system, the need of interpolation arises whenever an image or deformation field is continuously evaluated.

In the presented algorithm this can be for example a crucial step when the deformation field is applied to the moving volume or resampling is done. Since we can store the image and deformation field data in texture memory, we can in general choose whether to employ nearest-neighbor or trilinear interpolation. In contrast to the CPU approach trilinear interpolation is hardwired and therefore “free” on the GPU, which means that there is no notable performance difference between these two interpolation schemes. However, trilinear interpolation might not be sufficient to obtain a satisfying registration result. In these cases it is often a vast improvement to employ tricubic interpolation instead [Par83].

As presented in [Rui08] or [The00] interpolation can be described in general as the evaluation of the equation

$$f(x) = \sum_{k \in \mathbb{Z}} c(k)\beta_n(x-k) \tag{6.1}$$

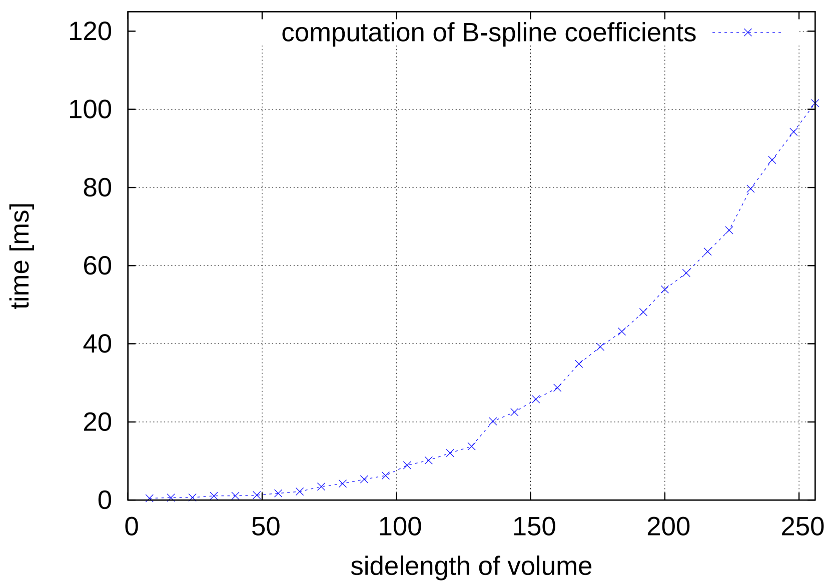


Figure 6.1: Computation time of B-spline coefficients for tricubic interpolation on System A.

where $c(k)$ are the B-spline coefficients and β_n is the B-spline basis of order n. We emphasize that only in the case of nearest-neighbor ($n = 0$) and linear interpolation ($n = 1$) the coefficients $c(k)$ coincide with the actual function values $f(k)$. It is often presented differently in literature as for example in [Sig05]. We refer to [The00] for more details on this issue. For tricubic interpolation ($n = 2$) these coefficients need to be computed in a preprocessing step. This calculation can be computed efficiently with a causal and non-causal filter similar to the presentation in [Uns99]. Figure 6.1 provides the runtimes for this task on System A. Especially for larger image sizes these additional computation times can be with over 100 ms significant.

Once these B-spline coefficients are calculated the tricubic interpolation itself can be done quite efficiently on the GPU. Since the support of cubic B-splines is quite local (the width equals 4), we can evaluate Eq. 6.1 with 4^N nearest-neighbor interpolations, where N is the dimension of the image. However, with the method presented in [Sig05] we can replace this by a weighted combination of 2^N linear interpolations. Following this consideration we can for $N = 3$ use 8 trilinear interpolations, which are "free" on the GPU, instead of 64 nearest-neighbor interpolations to evaluate Eq. 6.1. We reference to [Sig05] or [Rui08] for more details on this concept. Because of the efficient trilinear interpolation on GPUs, a CUDA implemented registration algorithm will outperform CPU implemented equivalents more significantly if tricubic interpolation is employed.

We point out that, if tricubic interpolation is used throughout the whole registration process, the prefiltering to obtain the B-spline coefficients needs to be done once for the reference but in every iteration for the warped moving volume and the deformation fields. Also the evaluation of Eq. 6.1 is computationally more expensive than a single texture lookup. The overall registration time might thus increase significantly compared to trilinear interpolation. However higher-order interpolation schemes could be the key to obtain higher registration accuracy or to avoid artifacts.

The investigation of registration results, a detailed runtime comparison and optimization of tricubic interpolation in this context should be part of future work.

6.3 Regularization with Nonlinear Diffusion Filtering

The regularizer of the presented registration method does not take structures or different types of tissue into account, since the regularization is done globally with a Gaussian filter. Thus rigid structures as for example bones may be transformed in a nonrigid manner, while growth to tumors is concealed [Sta07]. One approach to restrict deformations to regions of similar tissue type, and therefore usually similar intensity values, is to employ nonlinear diffusion filters. In contrast to linear diffusion filtering, which is equivalent to a convolution with a Gaussian kernel, nonlinear diffusion filters preserve or even enhance edges in an image. In practice this means that deformation fields can be locally regularized by a diffusion filter, which is related to the tissue stiffness. Therefore filtering is amplified in areas of similar tissue type as for example an organ, while filtering is reduced across borders of different regions. The main reason to not use these nonlinear filters is the high computationally cost. A very efficient and reliable scheme, based on additive operator splitting (AOS), is presented in [Wei98].

We implemented this AOS scheme parallelized in CUDA and will provide the achieved computation results as motivation for further work. However, we will restrict the presentation of theoretical background of this nonlinear filter to the most essential parts and refer to the literature provided in [Wei98] for more details.

Let Ω be the domain of our image $f : \Omega \mapsto \mathbb{R}$ with $x \in \Omega$. In the following we consider the m-dimensional filter introduced in [Cat92] which calculates a filtered image $u(x, t)$ of our initial image $f(x)$ as the solution of the diffusion equation

$$\partial_t u = div\left(g(|\nabla u_\sigma|^2)\nabla u\right) = \sum_{l=1}^{m} \partial_{x_l}\left(g(|\nabla u_\sigma|^2)\partial_{x_l} u\right) \qquad (6.2)$$

with initial state $u(x, 0) = f(x)$ and Neumann boundary conditions. We used the diffusion function $g : \mathbb{R} \mapsto \mathbb{R}$, which is found in [Wei98] and defined as

$$g(s) = \left\{ \begin{array}{cl} 1, & \text{if } s \leq 0 \\ 1 - exp(\frac{-3.315}{(s/\lambda)^4}), & \text{otherwise} \end{array} \right. . \qquad (6.3)$$

Considering t as the so-called scale parameter, the original image f is embedded into a scale-space [Wei98]. Compared to the well known Perona and Malik filter [Per90], this filter depends on the magnitude of the gradient $|\nabla u_\sigma|$ of a smoothed image $u_\sigma = G_\sigma \star u$, where G_σ is a Gaussian with standard deviation σ. This Gaussian filtering of u tackles the sensitivity to noise of the Perona and Malik filter. The behavior of the edge detector g is controlled by the diffusion parameter λ.

In order to solve Eq. 6.2 we can discretize it by central differences and obtain a semi-implicit scheme in the form

$$u^{k+1} = \left(i - \tau \sum_{l=1}^{m} A_l(u_k)\right)^{-1} u^k \qquad (6.4)$$

which can be modified to the so-called AOS scheme

$$u^{k+1} = \frac{1}{m} \sum_{l=1}^{m} \left(I - m\tau A_l(u_k)\right)^{-1} u^k . \qquad (6.5)$$

The coefficients in the matrix $A_l(u_k)$ follow from the discretization and can be found in [Wei98]. It turns out that the operators $B_l(u^k) = I - m\tau A_l(u_k)$ are tridiagonal and strictly diagonal dominant. Therefore each operator $B_l(u^k)$ can be inverted in a very efficient and stable way by computing the LR decomposition of this matrix followed by a forward and backward substitution. Again we refer to [Wei98] where this algorithm is described in detail and where is shown that the AOS scheme creates a discrete scale-space. This means that desirable properties as the maximum-minimum principle or convergence to a constant steady-state are provided.

To obtain similar regularization results in homogeneous regions (where $g \approx 1$) to the regularization obtained by Gaussian filtering, we recall the equivalence of a Gaussian convolution with

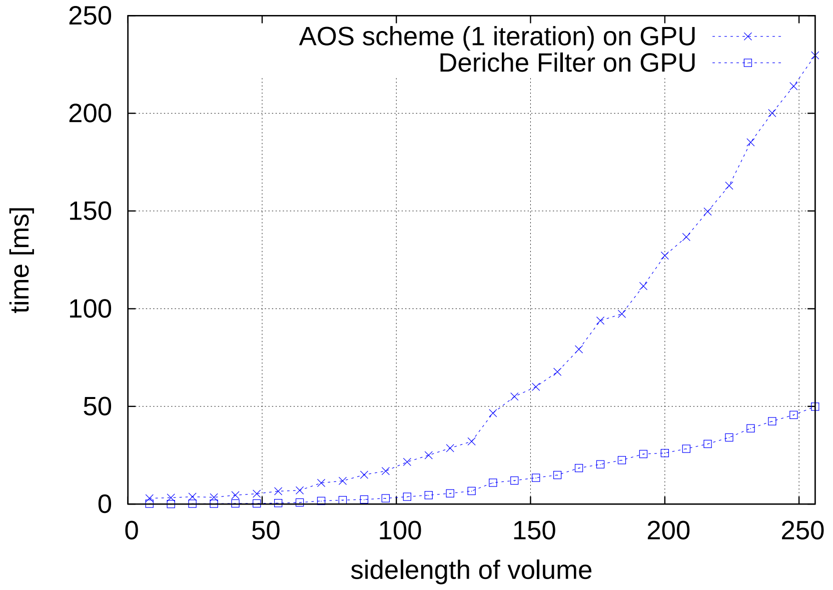


Figure 6.2: Computation time of nonlinear diffusion filtering solved with an AOS scheme (1 iteration) compared to Deriche filtering on System A.

standard deviation σ to linear diffusion filtering for a certain end time $T = \frac{\sigma^2}{2}$. It is investigated in [Wei98] that the accuracy of the AOS scheme depends on the step size τ. Because of this one will usually need more than one iteration to solve for a certain end time $T = n\tau$.

In Figure 6.2 we present how our CUDA implementation scales with different volume sizes. We compared the Deriche filter presented in Section 4.2 to a single iteration of the AOS scheme. As already mentioned, dependent on σ, significantly more iterations might be necessary in practice. Visual results where we filtered a MR scan of resolution $192 \times 192 \times 160$ are shown in Figure 6.3. We set $\sigma = 3$ and $\tau = 0.25$ what results in $n = 18$ iterations ($T = \frac{3^2}{2} \overset{!}{=} 0.25n$). For the computation of image $d)$ we set $g(s) = 1$ and compared the AOS scheme to the direct Gaussian convolution with a kernel radius of 4σ. We observed an average relative L1-error of 2.1%.

Overall we think, that these results look promising and it is worth to investigate more thoroughly how nonlinear diffusion filtering could improve registration results. Most likely registration results could already be significantly improved if nonlinear diffusion filters are only employed on pyramid levels with lower resolutions where computation times are still reasonable.

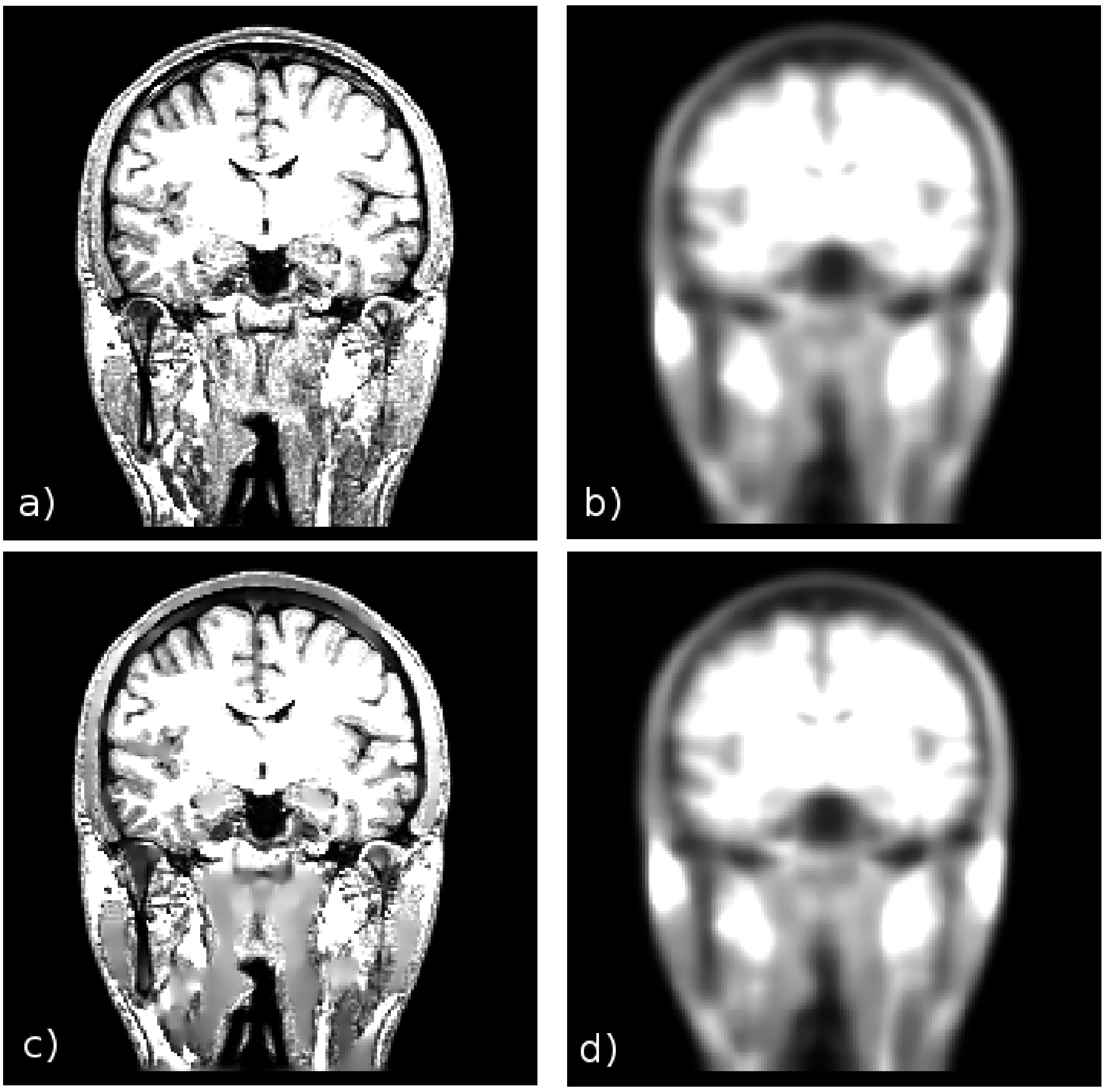


Figure 6.3: Example of different filtering techniques with $\sigma = 3.0$. a) Original Image, b) Direct Convolution ($kernelRadius = 12$), c) Nonlinear Diffusion Filter $\lambda = 10.0, \tau = 0.25$, d) Nonlinear Diffusion Filter $g = 1, \tau = 0.25$.

6.4 Polynomial Intensity Correction

During this work we took notice of a novel similarity measure for multimodal registration.

The investigation of this new metric is beyond the scope of this document and part of future research. However, since first experimental results are quite promising, we want to provide the idea and the first variation. This allows to employ this metric in the framework which was presented in Chapter 3.

As illustrated in [Her02] cross correlation measures the affine dependence of the two random variables X^f and X^g_u given by the reference image f and moving image g under the displacement u. This basically results in finding an affine function which is best fitting the jpdf of both images.

Instead of looking for an affine function it was proposed in [Ou09] to find a polynomial which is best fitting, in a least square sense, the joint pdf described by f and g_u.

In other words, in a first step we try to model the intensity values $g_u(x)$ as polynomial function of the intensity values $f(x)$. For this model, which is a polynomial intensity correction (PIC) of the intensity values of f, we will then calculate the cost function as the SSD in a second step. This idea was already described in [Ou09].

We will now present this similarity measure in a continuous setting and make it applicable for nonrigid registration by providing the variational gradient.

For a given vector $a \in \mathbb{R}^m$ we define the polynomial function $p : \mathbb{R}^m \times \mathbb{R} \mapsto \mathbb{R}$ of degree $m-1$ as:

$$p(a, f(x)) = \sum_{i=0}^{m-1} a_i f(x)^i \tag{6.6}$$

For a discrete subset of n samples $\{x_1, \dots, x_n\} \subset \Omega$ we also define:

$$y_u := \begin{pmatrix} g_u(x_1) \\ \vdots \\ g_u(x_n) \end{pmatrix} \quad , \quad X := \begin{pmatrix} 1 & f(x_1) & \dots & f(x_1)^{m-1} \\ 1 & f(x_2) & \dots & f(x_2)^{m-1} \\ \vdots & \vdots & & \vdots \\ 1 & f(x_n) & \dots & f(x_n)^{m-1} \end{pmatrix} \tag{6.7}$$

We can now obtain the coefficients $\hat{a}_u$ of the polynomial of degree $m-1$, which is best approximating the joint pdf of g_u and f, by solving the minimization problem

$$\hat{a}_u = \underset{a \in \mathbb{R}^m}{\operatorname{argmin}} \, ||y_u - Xa||^2 \, . \tag{6.8}$$

We assume that $n > m$ and that X has no rank deficiency. These are mild assumptions since we can always pick more samples and know that the Vandermonde matrix X has full rank if at least m image values $f(x_i)$ are pairwise distinct. In this case we can rewrite $\hat{a}_u$ as

$$\hat{a}_u = \underbrace{(X^T X)^{-1} X^T}_{=:C} y_u \tag{6.9}$$

where we find the i-th coefficient $\hat{a}_u^i$ by computing a scalar product as $\hat{a}_u^i = C_i y_u$.

With these preliminaries we can now consider the in [Ou09] presented cost function in the continuous formulation

$$\begin{aligned}\mathcal{J}_{PIC\_SSD}[u, \hat{a}_u] &= \frac{1}{2} \int_\Omega (g_u(x) - p(\hat{a}_u, f(x)))^2 \, dx \\ &= \frac{1}{2} \int_\Omega \left(g_u(x) - \sum_{i=0}^{m-1} C_i \begin{pmatrix} g_u(x_1) \\ \vdots \\ g_u(x_n) \end{pmatrix} f(x)^i \right)^2 dx\end{aligned} \tag{6.10}$$

where we calculate the first variation straight forward to

$$\begin{aligned}\delta_h \mathcal{J}_{PIC\_SSD}[u, \hat{a}_u] = \int_\Omega &(g_u(x) - p(\hat{a}_u, f(x))) \\ &\left(\nabla g_u(x) - \sum_{i=0}^{m-1} C_i \begin{pmatrix} \nabla g_u(x_1) \\ \vdots \\ \nabla g_u(x_n) \end{pmatrix} f(x)^i \right) \cdot h(x) \, dx\end{aligned} \tag{6.11}$$

and provide the resulting gradient of $\mathcal{J}_{PIC\_SSD}$ as

$$\nabla_u \mathcal{J}_{PIC\_SSD}(x) = (g_u(x) - p(\hat{a}_u, x)) \left(\underbrace{\nabla g_u(x)}_{\text{1st term}} - \underbrace{\sum_{i=0}^{m-1} C_i \begin{pmatrix} \nabla g_u(x_1) \\ \vdots \\ \nabla g_u(x_n) \end{pmatrix} f(x)^i}_{\text{2nd term}} \right) . \tag{6.12}$$

To evaluate this similarity measure we need to solve the minimization problem given by Eq. 6.8 in each iteration. We did this in first experiments by employing a free CUDA library for linear algebra problems named CULAtools. Since this optimization algorithm just solved for the

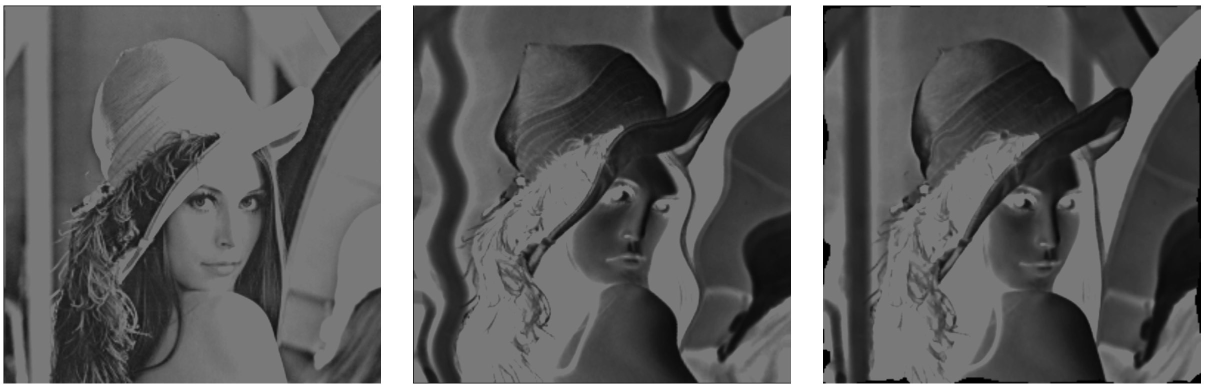


Figure 6.4: Registration results obtained with PICSSD: reference image (left), moving image (middle), registration result (right).

coefficients a, but not for the left inverse C, we neglected the second term in the computation of the gradient, which is shown in Eq. 6.12, in our first experiments.

For our first experiments we employed our CUDA implemented nonrigid registration pipeline. As test data we used a volume of size $256 \times 256 \times 64$ containing 64 slices of an image of "Lenna". We applied a generated deformation field, with displacements described by trigonometric functions, and a nonlinear transfer function to this volume and tried to recover the deformation.

First visual results, which are given in Figure 6.4, show that, even if we approximated the gradient by neglecting the 2nd term, we were able to recover the deformation.

After we have computed the variational gradient of the presented similarity measure, we showed in a first experiment that it allows us to recover nonrigid deformations on commonly used data. Further experiments which also take the second term of the differentiation into account need to be conducted. Also the influence of the parameters n and m is of high interest.

Within this Chapter we showed that our work can, next to general starting points, be extended in many directions. We provided a first idea of how registration results could be improved by more complex tricubic interpolation schemes and nonlinear filters used as regularizer. Especially the performance of a new similarity criterion called polynomial intensity correction of the sum of squared differences (PICSSD) applied on nonrigid registration tasks is of high interest for future research.

Chapter 7

Summary

We motivated the problem and illustrated the need of a fast nonrigid registration in Chapter 1, where we also gave an introduction to NVIDIA's CUDA programming model. Especially applications as image guidance in neurosurgical interventions or new opportunities in ART were mentioned in related works [MO08], [Rui10].

We provided an overview of nonrigid registration methods in Chapter 2. After describing a general nonrigid registration pipeline and its components we focused on commonly used similarity measures and transformation models. We also raised the challenge of picking the right optimizer to solve the evolving optimization problem.

In Chapter 3, we picked a gradient based nonrigid registration algorithm, called *flows of diffeomorphisms*, and provided variational gradients for the three different similarity measures SSD, MI and LCC. In this context we employed a Parzen estimator based on a normalized Gaussian kernel to estimate pdfs.

We presented a complete nonrigid registration pipeline implemented in parallel on the GPU using CUDA in Chapter 4. This pipeline resides completely on the GPU and does therefore not suffer of expensive data transfers between host and device memory. The performance of the introduced registration algorithm highly depends on a fast regularization of the deformation fields and a fast computation of similarity measures. Thus it turned out that Gaussian filtering and joint histogram calculation are the crucial building blocks in terms of computation time. With a recursive implemented filtering algorithm based on Deriche's approach [Der93] and an optimized joint histogram calculation [Led10] we presented high performant parallel solutions on CUDA compatible devices for both challenges.

More precisely, we provided theoretical background on approximating filters by IIR filters, which can be implemented recursively, in Section 4.2. We deduced a separable recursive filter, which is approximating a Gaussian and suitable for an efficient parallel implementation on the GPU. Performance experiments revealed convincing computation times for this building block. Especially the independence of the measured runtimes of the standard deviation σ is emphasized in this context. However, filtering along one dimension suffers dramatically of uncoalesced memory access, which is a bottleneck. Since the overall registration time is particularly dominated by filtering, further research on filtering techniques should be conducted.

In Section 4.3 we presented three novel optimization strategies which allow for a faster computation of joint histograms compared to a recent histogram algorithm, even if higher-order interpolation schemes are used or memory is limited. While *Bin Caching* requires temporary additional memory, optimizations based on *Multiple Bin Encoding* and *Smart Texture Lookup* are always possible. The combination of *MBE* and *BC* allows for an average decrease in computation time of around 20% on our test systems with a reduced memory overhead compared to an advanced histogram algorithm. Even if *STL* is significantly slower than *BC*, this optimization can still be useful when no additional memory is available. None of the proposed optimization techniques require preprocessing steps.

In Chapter 5 we conducted experiments for the most important building blocks, filtering and joint histogram computation, and benchmarked the complete registration pipeline against a parallel CPU implementation. Compared to this parallel and optimized CPU implementation we observed significant speed ups of up to one order in any experiment by employing commodity GPU hardware. Even with a GeForce 8800 GTX, a rather low-end GPU outperformed a high performant multicore system with 8×2.5 GHz Intel Xeon CPUs by an overall factor of roughly 3 in practical experiments. This makes high performant nonrigid registration not just technically possible but especially economically accessible.

Trading registration accuracy for computation time could allow with runtimes clearly below 1 second for real-time registration with several frames. Next to a detailed overview of the contribution of individual CUDA kernels we also provided, dependent on the similarity criterion, a memory footprint for different datasets. We completed this Chapter with a discussion of optimization and performance aspects. In this context we emphasize especially the rather expensive data transfers between global and texture memory.

In Chapter 6 we proposed to extend this work by focusing on more complex regularization techniques, as nonlinear diffusion filters which take structures into account. To what extent higher-order interpolation schemes as tricubic interpolation allow for a reasonable trade off of registration time to registration accuracy requires a detailed investigation in the future. We also integrated a new similarity measure based on polynomial intensity correction (PIC) in the presented framework. While preliminary results obtained with this metric look very promising, a further analysis remains open.

During the development, we obeyed important CUDA specific optimization guidelines. However, we point out that optimizing the configuration of important kernels dependent on the graphics card and CUDA version could allow for a further decrease in computation time. This is of particular interest for the presented recursive filter.

We conclude that a nonrigid registration pipeline based on the *flows of diffeomorphisms* can be implemented very efficiently on the GPU using NVIDIA CUDA. We think that these performance gains are a vast step towards the feasibility of nonrigid registration techniques in interventional applications. As for example stated in [Arc07], 30 seconds registration time might be sufficient in a practical intraoperative setting.

The presented pipeline can easily provide these runtimes even on large datasets. The nonrigid registration of datasets of sizes up to 256^3 took in any conducted experiment less than 6 seconds, even when complex similarity measures as MI and reasonable iteration schemes were employed. By investigating device dependent optimizations further decreases in computation times can be achieved. If registration time is not a too crucial factor, nonlinear regularization techniques or more complex but more expensive interpolation schemes should allow for a better accuracy when recovering nonrigid deformations.

Next to the performance we emphasize the importance of the maintainability of software projects, which is given with CUDA.

We recall that within the scope of this work we first of all focused on computation times. Nevertheless, also the validation of the quality of the recovered deformation is fundamental when it comes to clinical applications and remains open. The selection of similarity criterion, transformation model and optimizer is crucial in terms of runtime and registration accuracy and in practice dependent on the application.

Overall the results of this work are highly encouraging to further investigate nonrigid registration methods that are implemented on GPUs. With NVIDIA CUDA a powerful technique is available to realize GPU implementations efficiently. The presented results open the door to new applications for advanced registration methods, such as solving nonrigid alignment problems in interventional imaging under strict time constraints.

Appendix A

Theorems and Calculations

Theorems

Theorem A.1. *If G is symmetric and normalized and f is linear then* $(G \star f)(t) = f(t)$.

Proof.

$$\begin{aligned}
(G \star f)(t) =& \int_{-\infty}^{\infty} G(\tau) f(t-\tau)\, d\tau \\
=& \int_{-\infty}^{0} G(\tau) f(t-\tau)\, d\tau + \int_{0}^{\infty} G(\tau) f(t-\tau)\, d\tau \\
=& \int_{0}^{\infty} G(-\tau) f(t+\tau)\, d\tau + \int_{0}^{\infty} G(\tau) f(t-\tau)\, d\tau \\
=& \int_{0}^{\infty} G(\tau)(f(t)+f(\tau))\, d\tau + \int_{0}^{\infty} G(\tau)(f(t)-f(\tau))\, d\tau \\
=& \int_{0}^{\infty} G(\tau) 2f(t)\, d\tau \\
=& 2f(t) \int_{0}^{\infty} G(\tau)\, d\tau = 2f(t)\frac{1}{2} = f(t)
\end{aligned} \tag{A.1}$$

□

Representation of the Operator $g^+_{\text{appr}}(k)$

If α_i and λ_i ($i = 1, \dots, 4$) are complex but pairwise conjugate, we can do the following calculation to represent the operator $g^+_{\text{appr}}(k)$ for $k \geq 0$:

$$\begin{aligned}
g^+_{\text{appr}}(k) &= \sum_{i=1}^{4} \alpha_i e^{-\lambda_i \frac{k}{\sigma}} \\
&= (a_1 + ib_1)e^{-(\beta_1 + i\omega_1)\frac{k}{\sigma}} + (a_1 - ib_1)e^{-(\beta_1 - i\omega_1)\frac{k}{\sigma}} \\
&\quad + (a_2 + ib_2)e^{-(\beta_2 + i\omega_2)\frac{k}{\sigma}} + (a_2 - ib_2)e^{-(\beta_2 - i\omega_2)\frac{k}{\sigma}} \\
&= e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(-\omega_1\frac{k}{\sigma}) + ia_1\sin(-\omega_1\frac{k}{\sigma}) + ib_1\cos(-\omega_1\frac{k}{\sigma}) - b_1\sin(-\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(\omega_1\frac{k}{\sigma}) + ia_1\sin(\omega_1\frac{k}{\sigma}) - ib_1\cos(\omega_1\frac{k}{\sigma}) + b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(-\omega_2\frac{k}{\sigma}) + ia_2\sin(-\omega_2\frac{k}{\sigma}) + ib_2\cos(-\omega_2\frac{k}{\sigma}) - b_2\sin(-\omega_2\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(\omega_2\frac{k}{\sigma}) + ia_2\sin(\omega_2\frac{k}{\sigma}) - ib_2\cos(\omega_2\frac{k}{\sigma}) + b_2\sin(\omega_2\frac{k}{\sigma})) \\
&= e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(\omega_1\frac{k}{\sigma}) - ia_1\sin(\omega_1\frac{k}{\sigma}) + ib_1\cos(\omega_1\frac{k}{\sigma}) + b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(\omega_1\frac{k}{\sigma}) + ia_1\sin(\omega_1\frac{k}{\sigma}) - ib_1\cos(\omega_1\frac{k}{\sigma}) + b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(\omega_2\frac{k}{\sigma}) - ia_2\sin(\omega_2\frac{k}{\sigma}) + ib_2\cos(\omega_2\frac{k}{\sigma}) + b_2\sin(\omega_2\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(\omega_2\frac{k}{\sigma}) + ia_2\sin(\omega_2\frac{k}{\sigma}) - ib_2\cos(\omega_2\frac{k}{\sigma}) + b_2\sin(\omega_2\frac{k}{\sigma})) \\
&= e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(\omega_1\frac{k}{\sigma}) + b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_1\frac{k}{\sigma}}(a_1\cos(\omega_1\frac{k}{\sigma}) + b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(\omega_2\frac{k}{\sigma}) + b_2\sin(\omega_2\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(a_2\cos(\omega_2\frac{k}{\sigma}) + b_2\sin(\omega_2\frac{k}{\sigma})) \\
&= e^{-\beta_1\frac{k}{\sigma}}(2a_1\cos(\omega_1\frac{k}{\sigma}) + 2b_1\sin(\omega_1\frac{k}{\sigma})) \\
&\quad + e^{-\beta_2\frac{k}{\sigma}}(2a_2\cos(\omega_2\frac{k}{\sigma}) + 2b_2\sin(\omega_2\frac{k}{\sigma})) \\
&= \sum_{i=1}^{2} (a'_i\cos(\frac{\omega_i}{\sigma}k) + b'_i\sin(\frac{\omega_i}{\sigma}k))e^{\frac{-\beta_i}{\sigma}k}
\end{aligned} \tag{A.2}$$

Z-transformation of the Operator $g^+_{\text{appr}}(k)$

We use known Z-transform pairs as given in [Che01, p. 210] and the linearity law to compute the Z-transformation of

$$g^+_{\text{appr}}(k) = \sum_{i=1}^{2} (a_i \cos(\frac{\omega_i}{\sigma}k) + b_i \sin(\frac{\omega_i}{\sigma}k))e^{\frac{-\beta_i}{\sigma}k}, k \geq 0 \tag{A.3}$$

to

$$\begin{aligned}
&\sum_{k=0}^{\infty} g^+_{\text{appr}}(k) z^{-k} = \\
&= \underbrace{\frac{a_1 - a_1 e^{-\frac{\beta_1}{\sigma}} \cos(\frac{\omega_1}{\sigma}) z^{-1}}{1 - 2e^{-\frac{\beta_1}{\sigma}} \cos(\frac{\omega_1}{\sigma}) z^{-1} + e^{-2\frac{\beta_1}{\sigma}} z^{-2}}}_{Z[a_1(e^{-\frac{\beta_1}{\sigma}})^k \cos(\frac{\omega_1}{\sigma}k)]} + \underbrace{\frac{b_1 e^{-\frac{\beta_1}{\sigma}} \sin(\frac{\omega_1}{\sigma}) z^{-1}}{1 - 2e^{-\frac{\beta_1}{\sigma}} \cos(\frac{\omega_1}{\sigma}) z^{-1} + e^{-2\frac{\beta_1}{\sigma}} z^{-2}}}_{Z[b_1(e^{-\frac{\beta_1}{\sigma}})^k \sin(\frac{\omega_1}{\sigma}k)]} \\
&\quad + \underbrace{\frac{a_2 - a_2 e^{-\frac{\beta_2}{\sigma}} \cos(\frac{\omega_2}{\sigma}) z^{-1}}{1 - 2e^{-\frac{\beta_2}{\sigma}} \cos(\frac{\omega_2}{\sigma}) z^{-1} + e^{-2\frac{\beta_2}{\sigma}} z^{-2}}}_{Z[a_2(e^{-\frac{\beta_2}{\sigma}})^k \cos(\frac{\omega_2}{\sigma}k)]} + \underbrace{\frac{b_2 e^{-\frac{\beta_2}{\sigma}} \sin(\frac{\omega_2}{\sigma}) z^{-1}}{1 - 2e^{-\frac{\beta_2}{\sigma}} \cos(\frac{\omega_2}{\sigma}) z^{-1} + e^{-2\frac{\beta_2}{\sigma}} z^{-2}}}_{Z[b_2(e^{-\frac{\beta_2}{\sigma}})^k \sin(\frac{\omega_2}{\sigma}k)]} \\
&= \frac{a_1 + e^{-\frac{\beta_1}{\sigma}}(-a_1 \cos(\frac{\omega_1}{\sigma}) + b_1 \sin(\frac{\omega_1}{\sigma})) z^{-1}}{1 - 2e^{-\frac{\beta_1}{\sigma}} \cos(\frac{\omega_1}{\sigma}) z^{-1} + e^{-2\frac{\beta_1}{\sigma}} z^{-2}} \\
&\quad + \frac{a_2 + e^{-\frac{\beta_2}{\sigma}}(-a_2 \cos(\frac{\omega_2}{\sigma}) + b_2 \sin(\frac{\omega_2}{\sigma})) z^{-1}}{1 - 2e^{-\frac{\beta_2}{\sigma}} \cos(\frac{\omega_2}{\sigma}) z^{-1} + e^{-2\frac{\beta_2}{\sigma}} z^{-2}} \\
&= \frac{n_0^+ + n_1^+ z^{-1} + n_2^+ z^{-2} + n_3^+ z^{-3}}{1 + d_1^+ z^{-1} + d_2^+ z^{-2} + d_3^+ z^{-3} + d_4^+ z^{-4}}
\end{aligned} \tag{A.4}$$

with

$$\begin{aligned}
n_0^+ &= a_1 + a_2 \\
n_1^+ &= e^{-\frac{\beta_1}{\sigma}}(-(a_1 + 2a_2)\cos(\frac{\omega_1}{\sigma}) + b_1 \sin(\frac{\omega_1}{\sigma})) \\
&\quad e^{-\frac{\beta_2}{\sigma}}(-(2a_1 + a_2)\cos(\frac{\omega_2}{\sigma}) + b_2 \sin(\frac{\omega_2}{\sigma})) \\
n_2^+ &= a_1 e^{-2\frac{\beta_2}{\sigma}} + a_2 e^{-2\frac{\beta_1}{\sigma}} + 2e^{-\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}((a_1 + a_2)\cos(\frac{\omega_1}{\sigma})\cos(\frac{\omega_2}{\sigma}) \\
&\quad - b_1 \cos(\frac{\omega_2}{\sigma})\sin(\frac{\omega_1}{\sigma}) - b_2 \cos(\frac{\omega_1}{\sigma})\sin(\frac{\omega_2}{\sigma})) \\
n_3^+ &= e^{-\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}(-a_1 \cos(\frac{\omega_1}{\sigma}) + b_1 \sin(\frac{\omega_1}{\sigma})) + e^{-2\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}(-a_2 \cos(\frac{\omega_2}{\sigma}) + b_2 \sin(\frac{\omega_2}{\sigma}))
\end{aligned} \tag{A.5}$$

$$\begin{aligned}
d_1^+ &= -2e^{-\frac{\beta_1}{\sigma}}\cos(\frac{\omega_1}{\sigma}) - 2e^{-\frac{\beta_2}{\sigma}}\cos(\frac{\omega_2}{\sigma}) \\
d_2^+ &= 4e^{-\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}\cos(\frac{\omega_1}{\sigma})\cos(\frac{\omega_2}{\sigma}) + e^{-2\frac{\beta_1}{\sigma}} + e^{-2\frac{\beta_2}{\sigma}} \\
d_3^+ &= -2e^{-\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}\cos(\frac{\omega_1}{\sigma}) - 2e^{-2\frac{\beta_1}{\sigma}-\frac{\beta_2}{\sigma}}\cos(\frac{\omega_2}{\sigma}) \\
d_4^+ &= e^{-2\frac{\beta_1}{\sigma}-2\frac{\beta_2}{\sigma}}
\end{aligned}$$

Appendix B

Code Listings and Kernel Summaries

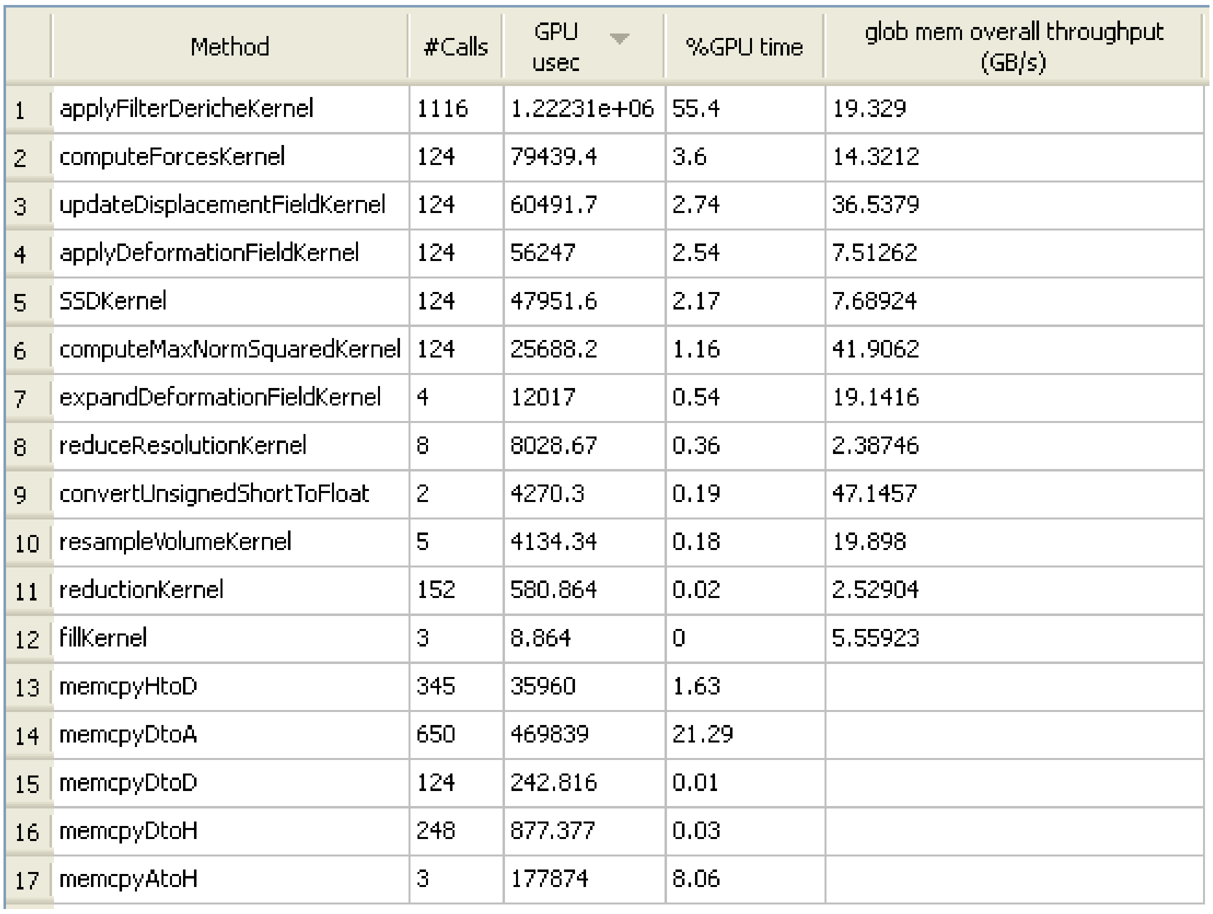
Histogram Computation

```
 1 /* 'bin' is defined as 'volatile' to prevent the compiler
      from optimizing away the comparison in line 12. */
   volatile unsigned int bin;
 3 unsigned int tagged;
   bin = (unsigned int) (data[i] * (bins - 1));
 5 do
   {
 7         unsigned int val = histogram[bin] & 0x07FFFFFF;
           /* The lower 5 bits of the thread id (tid) are
 9          used to tag the memory location. */
           tagged = (tid << 27) | (val + 1);
11         histogram[bin] = tagged;
   } while( histogram[bin] != tagged );
```

Listing B.1: Simulating atomic updates to shared memory for threads that belong to the same warp. [Sha07b]

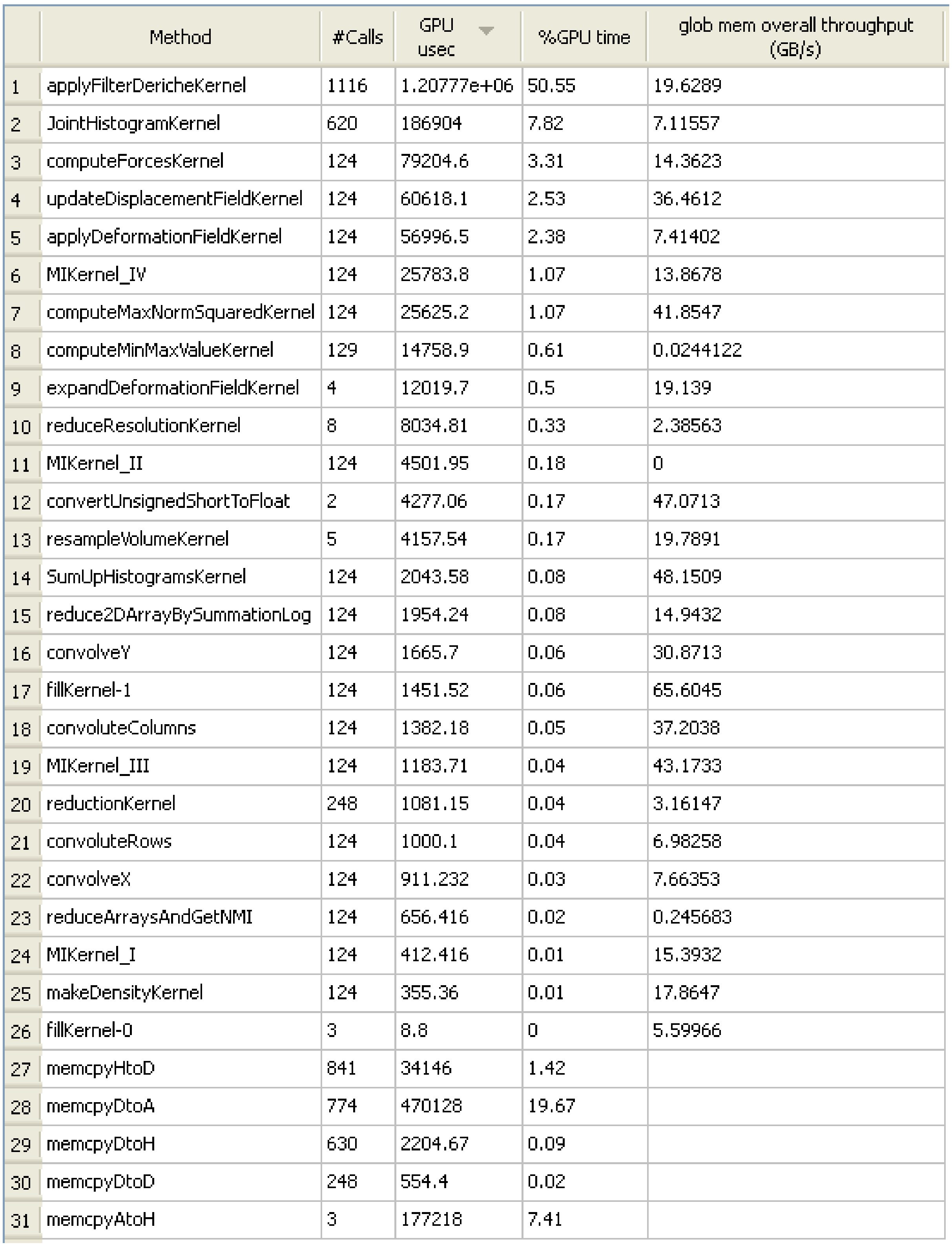
Kernel Summaries

In the following we provide additional CUDA visual profiler outputs of the performance experiments conducted in Section 5.3.2. These diagrams show the performance and contribution of all kernels of the presented pipeline for different similarity criterions. Especially the memory throughput is of interest, since it can be an indicator of the efficiency of a kernel.



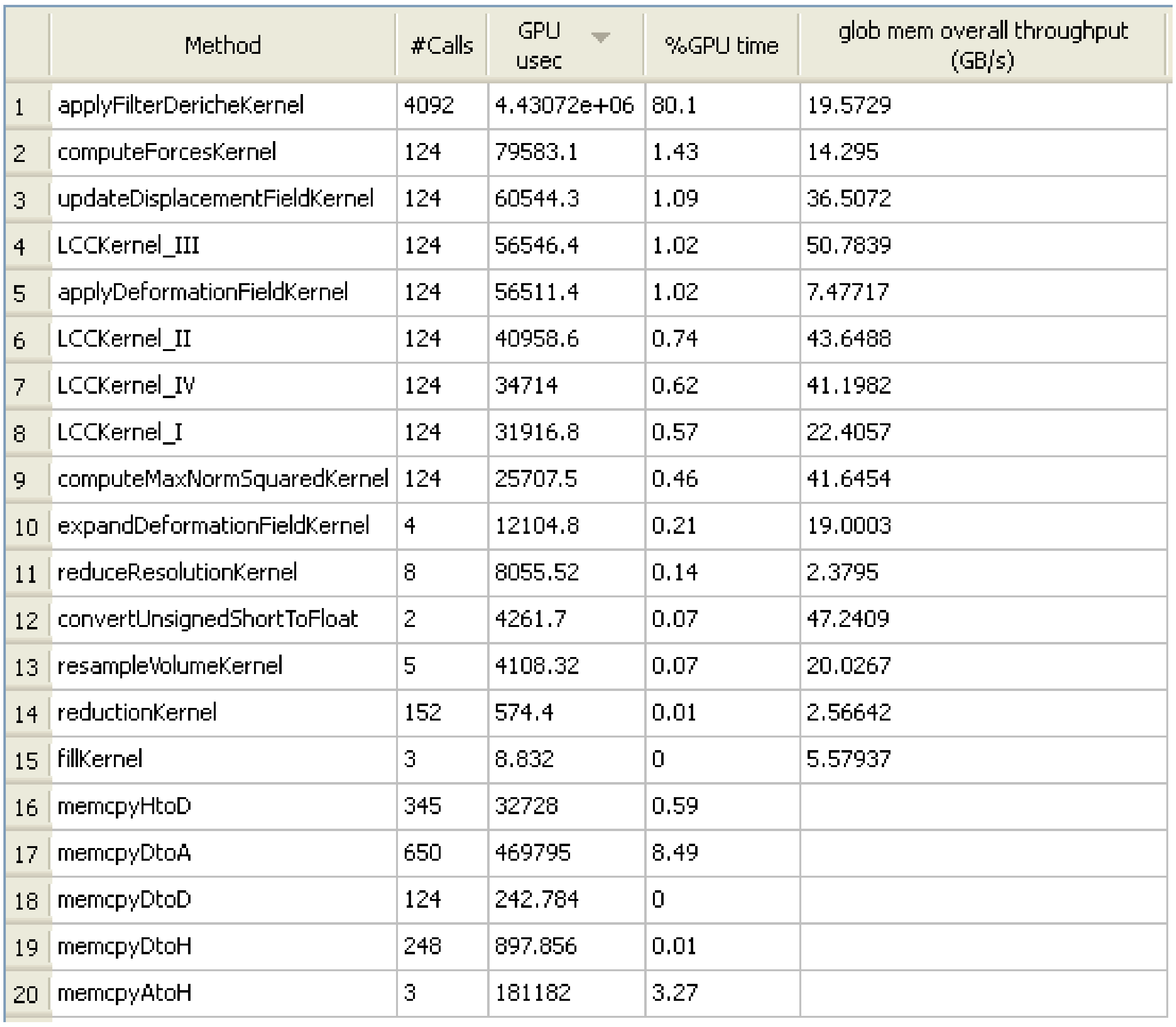
	Method	#Calls	GPU usec	%GPU time	glob mem overall throughput (GB/s)
1	applyFilterDericheKernel	1116	1.22231e+06	55.4	19.329
2	computeForcesKernel	124	79439.4	3.6	14.3212
3	updateDisplacementFieldKernel	124	60491.7	2.74	36.5379
4	applyDeformationFieldKernel	124	56247	2.54	7.51262
5	SSDKernel	124	47951.6	2.17	7.68924
6	computeMaxNormSquaredKernel	124	25688.2	1.16	41.9062
7	expandDeformationFieldKernel	4	12017	0.54	19.1416
8	reduceResolutionKernel	8	8028.67	0.36	2.38746
9	convertUnsignedShortToFloat	2	4270.3	0.19	47.1457
10	resampleVolumeKernel	5	4134.34	0.18	19.898
11	reductionKernel	152	580.864	0.02	2.52904
12	fillKernel	3	8.864	0	5.55923
13	memcpyHtoD	345	35960	1.63	
14	memcpyDtoA	650	469839	21.29	
15	memcpyDtoD	124	242.816	0.01	
16	memcpyDtoH	248	877.377	0.03	
17	memcpyAtoH	3	177874	8.06	

Figure B.1: Kernel Summary for Nonrigid Registration Pipeline using SSD.



	Method	#Calls	GPU usec	%GPU time	glob mem overall throughput (GB/s)
1	applyFilterDericheKernel	1116	1.20777e+06	50.55	19.6289
2	JointHistogramKernel	620	186904	7.82	7.11557
3	computeForcesKernel	124	79204.6	3.31	14.3623
4	updateDisplacementFieldKernel	124	60618.1	2.53	36.4612
5	applyDeformationFieldKernel	124	56996.5	2.38	7.41402
6	MIKernel_IV	124	25783.8	1.07	13.8678
7	computeMaxNormSquaredKernel	124	25625.2	1.07	41.8547
8	computeMinMaxValueKernel	129	14758.9	0.61	0.0244122
9	expandDeformationFieldKernel	4	12019.7	0.5	19.139
10	reduceResolutionKernel	8	8034.81	0.33	2.38563
11	MIKernel_II	124	4501.95	0.18	0
12	convertUnsignedShortToFloat	2	4277.06	0.17	47.0713
13	resampleVolumeKernel	5	4157.54	0.17	19.7891
14	SumUpHistogramsKernel	124	2043.58	0.08	48.1509
15	reduce2DArrayBySummationLog	124	1954.24	0.08	14.9432
16	convolveY	124	1665.7	0.06	30.8713
17	fillKernel-1	124	1451.52	0.06	65.6045
18	convoluteColumns	124	1382.18	0.05	37.2038
19	MIKernel_III	124	1183.71	0.04	43.1733
20	reductionKernel	248	1081.15	0.04	3.16147
21	convoluteRows	124	1000.1	0.04	6.98258
22	convolveX	124	911.232	0.03	7.66353
23	reduceArraysAndGetNMI	124	656.416	0.02	0.245683
24	MIKernel_I	124	412.416	0.01	15.3932
25	makeDensityKernel	124	355.36	0.01	17.8647
26	fillKernel-0	3	8.8	0	5.59966
27	memcpyHtoD	841	34146	1.42	
28	memcpyDtoA	774	470128	19.67	
29	memcpyDtoH	630	2204.67	0.09	
30	memcpyDtoD	248	554.4	0.02	
31	memcpyAtoH	3	177218	7.41	

Figure B.2: Kernel Summary for Nonrigid Registration Pipeline using MI.



	Method	#Calls	GPU usec	%GPU time	glob mem overall throughput (GB/s)
1	applyFilterDericheKernel	4092	4.43072e+06	80.1	19.5729
2	computeForcesKernel	124	79583.1	1.43	14.295
3	updateDisplacementFieldKernel	124	60544.3	1.09	36.5072
4	LCCKernel_III	124	56546.4	1.02	50.7839
5	applyDeformationFieldKernel	124	56511.4	1.02	7.47717
6	LCCKernel_II	124	40958.6	0.74	43.6488
7	LCCKernel_IV	124	34714	0.62	41.1982
8	LCCKernel_I	124	31916.8	0.57	22.4057
9	computeMaxNormSquaredKernel	124	25707.5	0.46	41.6454
10	expandDeformationFieldKernel	4	12104.8	0.21	19.0003
11	reduceResolutionKernel	8	8055.52	0.14	2.3795
12	convertUnsignedShortToFloat	2	4261.7	0.07	47.2409
13	resampleVolumeKernel	5	4108.32	0.07	20.0267
14	reductionKernel	152	574.4	0.01	2.56642
15	fillKernel	3	8.832	0	5.57937
16	memcpyHtoD	345	32728	0.59	
17	memcpyDtoA	650	469795	8.49	
18	memcpyDtoD	124	242.784	0	
19	memcpyDtoH	248	897.856	0.01	
20	memcpyAtoH	3	181182	3.27	

Figure B.3: Kernel Summary for Nonrigid Registration Pipeline using LCC.

Appendix C

Related Patents

In this section we mention patents that might be related to and patents that are originated in this work. Of course we do not claim that this list is complete from a legal viewpoint.

System and Method for GPU-based 3D nonrigid registration

Application number	11/062,962
Publication number	US 2005/0190189 A1
Publication date	1 September 2005
Inventor	Christophe Chefd'hotel Kinda Anna Saddi

Abstract:
A method of registering two images using a graphics processing unit includes providing a pair of images with a first and second image, calculating a gradient of the second image, initializing a displacement field on the grid point domain of the pair of images, generating textures for the first image, the second image, the gradient, and the displacement field, and loading said textures into the graphics processing unit. A pixel buffer is created and initialized with the texture containing the displacement field. The displacement field is updated from the first image, the second image, and the gradient for one or more iterations in one or more rendering passes performed by the graphics processing unit.

GPU-based image manipulation method for registration applications

Application number	11/109,126
Publication number	US 2005/0271302 A1
Publication date	8 December 2005
Inventor	Ali Khamene Christophe Chefd'hotel Jens Gühring Bernhard Geiger Sebastian Vogt

Abstract:
Exemplary systems and methods for performing registration applications are provided. An exemplary system includes a central processing unit (CPU) for transferring a plurality of images to a graphics processing unit (GPU); wherein the GPU performs a registration application on the plurality of images to produce a registration result, and wherein the GPU returns the registration result to the CPU. An exemplary method includes the steps of transferring a plurality of images from a central processing unit (CPU) to a graphics processing unit (GPU); performing a registration application on the plurality of images using the GPU; transferring the result of the step of performing from the GPU to CPU.

The following two pending patents were filed based on our work presented in Section 4.3 and Section 6.4.

Optimization of the Computation of Joint Histograms and Mutual Information for NVIDIA CUDA Compatible Devices

Application number	61/320,911
Publication date	05 April 2010
Status	Patent Pending
Inventor	Christian Ledig Guillaume Bousquet

Abstract:
We describe CUDA optimizations to solve the time consuming task of (Joint-) Histogram computation, especially in image registration using Mutual Information. Fast computation of Histograms is difficult since it is hard to parallelize on GPUs. By using CUDA together with improvements of a certain algorithm a fast computation gets possible.

Introduction of Polynomial Intensity Correction for Nonrigid Registration in a Variational Framework

Application number	61/365,521
Publication date	19 July 2010
Status	Patent Pending
Inventor	Christian Ledig Christophe Chefd'hotel

Abstract:
We present the derivation of a similarity measure, which is based on a Polynomial Intensity Correction of the Sum of Squared Difference (PICSSD), in a variational framework and make it thus applicable for nonrigid registration tasks.

List of Abbreviations

2D	Two Dimensional
3D	Three Dimensional
AOS	Additive Operator Splitting
ART	Adaptive Radiation Therapy
BC	Bin Caching
CC	Cross Correlation, Correlation Coefficients
CSYS	Coordinate System
CPU	Central Processing Unit
CR	Correlation Ratio
CT	Computed Tomography
CUDA	Compute Unified Device Architecture
fMRI	Functional Magnetic Resonance Imaging
GPU	Graphics Processing Unit
IIR	Infinite Impulse Response
jpdf	joint probability density function
LCC	Local Cross Correlation
MBE	Multiple Bin Encoding
MI	Mutual Information
MR	Magnetic Resonance
MRI	Magnetic Resonance Imaging
mutex	Mutual Exclusion
NMI	Normalized Mutual Information
PDE	Partial Differential Equation
pdf	probability density function
PET	Positron Emission Tomography
PIC	Polynomial Intensity Correction

PICSSD	Polynomial Intensity Correction of the Sum of Squared Differences
SAD	Sum of Absolute Differences
SSD	Sum of Squared Differences
STL	Smart Texture Lookup
TPS	Thin-Plate Splines

List of Figures

List of Tables

Bibliography

[Ans09] R. E. Ansorge, S. J. Sawiak, and G. B. Williams. Exceptionally fast non-linear 3D image registration using GPUs. In *IEEE Nuclear Science Symposium (NSS/MIC) Workshop on High Performance Medical Imaging (HPMI)*, pages 4088–4094, Orlando, FL, USA, October 2009.

[Arc07] N. Archip, O. Clatz, S. Whalen, D. Kacher, A. Fedorov, A. Kot, N. Chrisochoides, F. Jolesz, A. Golby, and S. K. Black, P. M. Warfield. Non-rigid alignment of pre-operative MRI, fMRI, and DT-MRI with intra-operative MRI for enhanced visualization and navigation in image-guided neurosurgery. *NeuroImage*, 35(2):609–624, April 2007.

[Aud00] M. A. Audette, F. P. Ferrie, and T. M. Peters. An algorithmic overview of surface registration techniques for medical imaging. *Medical Image Analysis*, 4(3):201–217, September 2000.

[Bha04] K. K. Bhatia, J. V. Hajnal, B. K. Puri, D. Edwards, and D. Rueckert. Consistent groupwise non-rigid registration for atlas construction. In *Proceedings of the IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*, volume 1, pages 908–911, Arlington, VA, USA, April 2004.

[Boo89] F. L. Bookstein. Principal warps: Thin-plate splines and the decomposition of deformations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(6):567–585, June 1989.

[Bro09] T. Brosch and R. Tam. A self-optimizing histogram algorithm for graphics card accelerated image registration. In *Proceedings of Medical Image Computing and Computer Assisted Intervention (MICCAI) Grid Workshop*, pages 35–44, London, United Kingdom, September 2009.

[Cac00] P. Cachier and X. Pennec. 3D non-rigid registration by gradient descent on a Gaussian-windowed similarity measure using convolutions. In *Proceedings of the IEEE Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA)*, pages 182–189, Hilton Head Island, SC, USA, June 2000. IEEE Computer society.

[Cat92] F. Catté, P. Lions, J. Morel, and T. Coll. Image selective smoothing and edge detection by nonlinear diffusion. *SIAM Journal on Numerical Analysis*, 29(1):182–193, February 1992.

[Che01] C. Chen. *Digital Signal Processing - Spectral Computation and Filter Design*. Oxford University Press, Inc., New York, 2001.

[Che02] C. Chefd'Hotel, G. Hermosillo, and O. Faugeras. Flows of diffeomorphisms for multimodal image registration. In *Proceedings of the IEEE International Symposium on Biomedical Imaging (ISBI)*, pages 21–28, Washington, DC, USA, July 2002.

[Che09] S. Chen, J. Qin, Y. Xie, W. Pang, and P. Heng. CUDA-based acceleration and algorithm refinement for volume image registration. In *Proceedings of the International Conference on Future BioMedical Information Engineering (FBIE)*, pages 544–547, Sanya, China, December 2009.

[Cla06] U. Clarenz, M. Droske, S. Henn, M. Rumpf, and K. Witsch. Computational methods for nonlinear image registration. In O. Scherzer, editor, *Mathematical Method for Registration and Applications to Medical Imaging, Mathematics in Industry*, volume 10, 2006.

[Cru03] W. R. Crum, L. D. Griffin, D. L. G. Hill, and D. J. Hawkes. Zen and the art of medical image registration: correspondence, homology, and quality. *NeuroImage*, 20(3):1425–1437, November 2003.

[Cru04] W. R. Crum, T. Hartkens, and D. L. G. Hill. Non-rigid image registration: theory and practice. *The British Journal of Radiology*, 77(2):140–153, 2004.

[Der87] R. Deriche. Separable recursive filtering for efficient multi-scale edge detection. In *Proceedings of the International Workshop Machine Vision and Machine Intelligence*, pages 18–23, Tokyo, Japan, February 1987.

[Der90] R. Deriche. Fast algorithms for low-level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–87, January 1990.

[Der93] R. Deriche. Recursively implementing the Gaussian and its derivatives. Technical report, INRIA, Unité de Recherche Sophia-Antipolis, 1993.

[Far06] G. Farneback and C. F. Westin. Improving Deriche-style recursive Gaussian filters. *Journal of Mathematical Imaging and Vision*, 26(3):293–299, December 2006.

[Fer02] M. Ferrant, A. Nabavi, B. Macq, P. McL. Black, F. A. Jolesz, R. Kikinis, and S. K. Warfield. Serial registration of intraoperative MR images of the brain. *Medical Image Analysis*, 6(4):337–359, December 2002.

[Fis03] B. Fischer and J. Modersitzki. Curvature based image registration. *Journal of Mathematical Imaging and Vision*, 18(1):81–85, January 2003.

[Gu10] X. Gu, H. Pan, Y. Liang, R. Castillo, D. Yang, D. Choi, E. Castillo, A. Majumdar, T. Guerrero, and J. B. Jiang. Implementation and evaluation of various demons deformable image registration algorithms on a GPU. *Physics in Medicine and Biology*, 55(1):207–219, January 2010.

[Hah09] D. A. Hahn. *Statistical Medical Image Registration with Applications in Epilepsy Diagnosis and Shape-Based Segmentation*. PhD thesis, Friedrich-Alexander University of Erlangen-Nuremberg, Erlangen, Germany, 2009.

[Hal06] D. Hale. Recursive Gaussian filters. Cwp report 546, Center for Wave Phenomena, 2006.

[Har07] M. Harris. Optimizing parallel reduction in CUDA. Technical report, NVIDIA, 2007.

[Haw05] D.J. Hawkes, D. Barratt, J.M. Blackall, C. Chan, P.J. Edwards, K. Rhode, G.P. Penney, J. McClelland, and D.L.G. Hill. Tissue deformation and shape models in image-guided interventions: a discussion paper. *Medical Image Analysis*, 9(2):163–175, April 2005.

[Hen02] S. Henn and K. Witsch. Iterative multigrid regularization techniques for image matching. *SIAM Journal on Scientific Computing*, 23(4):1077–1093, July 2002.

[Her01a] G. Hermosillo, C. Chefd'hotel, and O. Faugeras. A variational approach to multimodal image matching. Technical report, INRIA, February 2001.

[Her01b] G. Hermosillo and O. D. Faugeras. Dense image matching with global and local statistical criteria: A variational approach. In *Proceedings of Computer Vision and Pattern Recognition (CVPR)*, pages 73–78, Kauai, HI, USA, December 2001.

[Her02] G. Hermosillo, C. Chefd'Hotel, and O. D. Faugeras. Variational methods for multimodal image matching. *International Journal of Computer Vision*, 50(3):329–343, December 2002.

[Hil01] D. L. G. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes. Medical image registration. *Physics in Medicine and Biology*, 46(3):R1–R45, 2001.

[Jos04] S. C. Joshi, M. Foskey, and B. Davis. Automatic organ localization for adaptive radiation therapy for prostate cancer. Technical report ada428913, North Carolina University at Chapel Hill, May 2004.

[Kan92] P. Kannappan and P. K. Sahoo. Rotation invariant separable functions are Gaussian. *SIAM Journal on Mathematical Analysis*, 23(5):1342–1351, September 1992.

[Kul51] S. Kullback and R. A. Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

[Led10] C. Ledig and C. Chefd'Hotel. Efficient computation of joint histograms and normalized mutual information on CUDA compatible devices. In *Proceedings of a Medical Image Computing and Computer-Assisted Intervention (MICCAI) Workshop, High-Performance Medical Image Computing for Image-Assisted Clinical Intervention and Decision-Making Workshop (HP-MICCAI)*, pages 90–99, Beijing, China, September 2010.

[Lin08] Y. P. Lin and G. Medioni. Mutual information computation and maximization using GPU. In *Proceedings of Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1–6, Anchorage, Alaska, June 2008. IEEE Computer Society.

[Mae97] F. Maes, A. Collignon, D. Vandermeulen, G. Marchal, and P. Suetens. Multimodality image registration by maximization of mutual information. *IEEE Transactions on Medical Imaging*, 16(2):187–198, April 1997.

[MO08] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant. Fast deformable registration on the GPU: A CUDA implementation of demons. In *Proceedings of the 2008*

International Conference on Computational Science and Its Applications (ICCSA), Perugia , Italy, June/July 2008. IEEE Computer society.

[Mod04] J. Modersitzki. *Numerical Methods for Image Registration*. Oxford University Press, Inc., February 2004.

[Mul99] B. Mulgrew, P. Grant, and J. Thompson. *Digital Signal Processing*. Macmillan Press Ltd, 1999.

[NVI10a] NVIDIA. CUDA developer website, 2010. http://developer.nvidia.com/cuda/.

[NVI10b] NVIDIA. NVIDIA CUDA C programming, best practices guide 3.1. Technical report, NVIDIA, 2010.

[NVI10c] NVIDIA. NVIDIA CUDA programming guide. Technical report, NVIDIA, 2010.

[Ou09] W. Ou and C. Chefd'hotel. Polynomial intensity correction for multimodal image registration. In *Proceedings of the IEEE International Symposium on Biomedical Imaging: From Nano to Macro (ISBI)*, pages 932–942, Boston, MA, USA, June/July 2009.

[Par62] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, September 1962.

[Par83] J. A. Parker, R. V. Kenyon, and D. E. Troxel. Comparison of interpolation methods for image resampling. *IEEE Transactions on Medical Imaging*, 2(1):31–39, March 1983.

[Per90] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[Phi98] C. L. Phillips and J. M. Parr. *Signals, Systems, and Transforms*. Prentice Hall, August 1998.

[Pod07] V. Podlozhnyuk. Histogram calculation in CUDA. Technical report, NVIDIA, November 2007.

[Roc98] A. Roche, G. Malandain, X. Pennec, and N. Ayache. The correlation ratio as a new similarity measure for multimodal image registration. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 1496 of *Lecture Notes in Computer Science*, pages 1115–1124, Cambridge, MA, USA, September 1998.

[Roh01] K. Rohr, H. S. Stiehl, R. Sprengel, T. M. Buzug, J. Weese, and M. H. Kuhn. Landmark-based elastic registration using approximating thin-plate splines. *IEEE Transactions on Medical Imaging*, 20(6):526–534, June 2001.

[Rue99] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes. Nonrigid registration using free-form deformations: Application to breast MR images. *IEEE Transactions on Medical Imaging*, 18(8):712–721, August 1999.

[Rue06] D. Rueckert, P. Aljabar, R. A. Heckemann, J. V. Hajnal, and A. Hammers. Diffeomorphic registration using B-splines. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 4191 of *Lecture Notes in Computer Science*, pages 702–709, Copenhagen, Denmark, October 2006.

[Rui08] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens. Accuracy of GPU-based B-spline evaluation. In *Proceedings of Tenth IASTED International Conference on Computer Graphics and Imaging (CGIM)*, pages 117–122, Innsbruck, Austria, February 2008.

[Rui09] D. Ruijters, D. Babic, R. Homan, and P. Mielekamp. Real-time integration of 3-D multimodality data in interventional neuroangiography. *Journal of Electronic Imaging*, 18(3):033014, July–September 2009.

[Rui10] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens. GPU-accelerated elastic 3D image registration for intra-surgical applications. *Computer Methods and Programs in Biomedicine*, In Press, Corrected Proof, 2010.

[Sha07a] R. Shams and N. Barnes. Speeding up mutual information computation using NVIDIA CUDA hardware. In *Proceedings of the 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA)*, pages 555–560, Glenelg, Australia, December 2007. IEEE Computer Society.

[Sha07b] R. Shams and R. A. Kennedy. Efficient histogram algorithms for NVIDIA CUDA compatible devices. In *Proceedings of the International Conference on Signal Processing and Communications Systems (ICSPCS)*, pages 418–422, Gold Coast, Australia, December 2007.

[Sha10] R. Shams, P. Sadeghia, R. Kennedya, and R. Hartleya. Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images. *Computer Methods and Programs in Biomedicine*, 99(2):133–146, August 2010.

[Sig05] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In P Matt, editor, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 313–329. Addison-Wesley, 2005.

[Soz02] G. Soza, P. Hastreiter, M. Bauer, C. Rezk-Salama, C. Nimsky, and G. Greiner. Intraoperative registration on standard PC graphics hardware. In *Bildverarbeitung für Medizin*, pages 334–337, Leipzig, Germany, March 2002.

[Sta07] M Staring, S. Klein, and J. P. W. Pluim. Nonrigid registration with tissue-dependent filtering of the deformation field. *Physics in Medicine and Biology*, 52(23):6879–6892, 2007.

[Stu99] C. Studholme, D. L. G. Hill, and D. J. Hawkes. An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition*, 32(1):71–86, January 1999.

[The00] P. Thevenaz, T. Blu, and M. Unser. Interpolation revisited. *IEEE Transactions on Medical Imaging*, 19(7):739–758, July 2000.

[Thi98] J.-P. Thirion. Image matching as a diffusion process: an analogy with Maxwell's demons. *Medical Image Analysis*, 2(3):243–260, September 1998.

[Uns99] M. Unser. Splines: A perfect fit for signal and image processing. In *IEEE Signal Processing Magazine*, volume 16, pages 22–38, November 1999.

[Vio97] P. Viola and W. M. III Wells. Alignment by maximization of mutual information. *International Journal of Computer Vision*, 24(2):137–154, 1997.

[vV98] L. J. van Vliet, I. Young, and P. W. Verbeek. Recursive Gaussian derivative filters. In *Proceedings of International Conference on Pattern Recognition (ICPR)*, pages 509–514, Brisbane, Australia, August 1998.

[Wei98] J. Weickert, B. M. ter Haar Romeny, and M. A. Viergever. Efficient and reliable schemes for nonlinear diffusion filtering. *IEEE Transactions on Image Processing*, 7(3):398–410, March 1998.

[Wes97] J. West, J. M. Fitzpatrick, M. Y. Wang, B. M. Dawant, C. R. Maurer, Jr., R. M. Kessler, R. J. Maciunas, C. Barillot, D. Lemoine, A. Collignon, F. Maes, P. Suetens, D. Vandermeulen, P. A. van den Elsen, S. Napel, T. S. Sumanaweera, B. Harkness, P. F. Hemler, D. L. G. Hill, D. J. Hawkes, C. Studholme, J. B. A. Maintz, M. A. Viergever, G. Malandain, X. Pennec, M. E. Noz, G. Q. Maguire, Jr., M. Pollack, C. A. Pelizzari, R. A. Robb, D. Hanson, and R. P. Woods. Comparison and evaluation of retrospective intermodality brain image registration techniques. *Journal of Computer Assisted Tomography*, 21(4):554–566, July/August 1997.

[You95] I. T. Young and L. J. van Vliet. Recursive implementation of the Gaussian filter. *Signal Processing*, 44(2):139–151, June 1995.

[Zit03] B. Zitová and J. Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.