UNIVERSITÄT ZU LÜBECK
INSTITUTE OF MATHEMATICS AND
IMAGE COMPUTING

GPU Based Affine Linear Image Registration

GPU basierte affin-lineare Bildregistrierung

Masterarbeit

im Rahmen des Studiengangs
Medizinische Ingenieurwissenschaft
der Universität zu Lübeck

vorgelegt von
Florian Tramnitzke, B.Sc.

ausgegeben und betreut von
Prof. Dr. Jan Modersitzki
Institute of Mathematics and Image Computing

mit Unterstützung von
Dipl.-Math. Jan Rühaak und Lars König, M.Sc.
Fraunhofer MEVIS Projektgruppe Bildregistrierung

Lübeck, den 03. September 2014

MIC

IM FOCUS DAS LEBEN

Eidesstattliche Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Lübeck,
03. September 2014

Florian Tramnitzke

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources and resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Lübeck,
03. September 2014

Florian Tramnitzke

Abstract

In this thesis, a CUDA implementation of a complete registration algorithm, which is capable of aligning images using affine linear transformations and the Sum of Squared Differences (SSD) or Normalized Gradient Fields (NGF) distance measure is presented. Through close analysis of the mathematical composition of the distance measures, pixel-wise independent, explicit calculation rules for the function values and derivatives were derived. These are fully parallelizable and by the extensive use of different memory types, well handled thread management and efficient hardware interpolation extreme fast executing code was gained. Contrary to the common technique of reducing kernel calls, the performance was significantly increased by rearranging a single kernel into multiple smaller ones. A speedup of the GPU implementation compared to parallelized CPU code of up to 11.4 for a multilevel image registration using SSD and 18.8 using NGF was achieved. Matching two 512×512 pixel images is performed in 18 milliseconds using SSD and in 28 milliseconds using NGF, thus making state-of-the-art multimodal image registration available in real time scenarios.

Zusammenfassung

In dieser Arbeit wird eine CUDA Implementierung eines Registrierungsverfahren, welches Bilder mittels affin-linearen Transformationen ausrichten kann, präsentiert. Als Distanzmaße wurden die Summe der quadrierten Differenzen (SSD) und die normalisierten Gradientenfelder (NGF) gewählt. Durch eine genaue Analyse der mathematischen Zusammensetzung der Distanzmaße, wurde eine pixelweise unabhängige, explizite Rechenvorschrift für die Funktionswerte und Ableitungen hergeleitet. Diese sind voll parallelisierbar und mit ausgiebiger Nutzung verschiedener Speichertypen, genau durchdachten Kernelaufrufen und effizienter Hardwareinterpolation wurde Code mit extrem schnellen Laufzeiten generiert. Im Gegensatz zur herkömmlichen Technik Kernelaufrufe zu minimieren, konnte die Leistung signifikant gesteigert werden, indem ein großer Kernel in mehrere kleine Kernel aufgeteilt wurde. Ein Geschwindigkeitszuwachs der CUDA Implementierung verglichen mit parallelisierten CPU Code von bis zu 11.4 für eine multilevel Registrierung mit SSD und 18.8 mit NGF konnte erreicht werden. Das Registrieren zweier Bilder der Größe 512×512 Pixel wird in 18 Millisekunden mit SSD und in 28 Millisekunden mit NGF durchgeführt und ermöglicht den Einsatz von modernster multimodaler Bildregistrierung in Echtzeitszenarios.

Acknowledgements

This thesis is the result of hard work and would not have been possible without the help and support of colleagues, friends and family. Therefore, I would like to take the time to appreciate these people as they have my highest gratitude.

First and foremost, I would like to thank my parents and Ulf for their everlasting support and love, not only during the time I wrote this thesis, but throughout my whole studies. Whether I needed a time-out at home or a little refill for my wallet, I could always count on you. You made it possible that I could enjoy these past years to the fullest and for that I will always be grateful.

I would like to express my deep gratitude to Prof. Dr. Jan Modersitzki. You helped me with many problems, supported my internship at the University of Texas in Austin and are now the supervisor of this thesis. You gave me guidance, support and constructive criticism during this work, which was helpful in so many ways. It was a pleasure to work with you these past years and I am looking forward to working with you in the future.

I am particularly grateful for the assistance given by Jan Rühaak and Lars König. I bombarded you with questions and you guys always knew the answer. Incredible. Thank you for all the constructive talks and for every bit of help with this thesis. Also, it was you who encouraged me to write a paper about my thesis for a workshop at the MICCAI conference in Boston this year and now I will present this paper in a mere two weeks. Thank you guys for everything.

I would also like to thank Carina for her help with those nasty Excel tables and her delicious sweets. Without you, I would probably have starved. Thanks to all my friends and all the people I have got to know during these last years. You made me the person I am today. Many thanks to my brother Danny for being the best brother in the world and to his wife Ise for additional proofreading. I know you are always there for me and I want you to know that I am always there for you and my little niece Ida.

Finally, my special thanks are extended to Falk. You are the best friend a guy can have during both good and bad times. We enjoyed so many unforgettable moments together and laughed so many times, it could last for a lifetime. And I know, there is only more to come.

Contents

Chapter 1: Introduction

Image registration is an important task in various areas of applications including medical image computing, see e.g. [20, 32] and the references therein. Roughly speaking, the objective of image registration is to automatically establish correspondences of structures that are measured in different poses, perspectives or modalities, see e.g. [9, 57]. This objective is achieved by transforming one image (the template) to another image (the reference). Images can be of different spatial dimension and image registration is possible in 2D-2D, 2D-3D, 3D-3D settings. Time can be included as another dimension leading to 3D-4D or 4D-4D registration [20, 32, 57].
Transformations are generally divided into two classes, linear and non-linear transformations [27]. Determining the appropriate class of transformations is generally a difficult task [65]. The class of non-linear transformations includes transformations based on physical or statistical models, such as elastic or viscous fluid models, or spline-based algorithms [9, 27]. They usually have a large degree of freedom for the transformation parameters, which results in high computational demand and long execution times [57]. The class of affine linear transformations, which includes rigid transformations, is used in numerous applications [18, 31, 49, 64] and as a pre-processing step in almost all medical image registrations [32, 37]. Furthermore, rigid transformations are broadly established in clinical practice as they preserve geometry and can be performed in comparably short time.
Following [37], an optimization framework can be used to formalize the registration problem mathematically. Here, the transformation is characterized as a minimizer of a carefully chosen distance measure D between the reference and transformed template. For images of the same modality, the Sum of Squared Differences (SSD) [3] is a simple, yet powerful distance measure. In a multimodal setting, the choice of D is non-trivial. Mutual Information (MI) [61, 7] is widely used but results in a highly non-convex optimization problem [17]. The Normalized Gradient Fields (NGF) distance measure proposed by Haber and Modersitzki [17] provides a fair compromise between flexibility with respect to modalities and convexity. The computation of a solution of the registration problem, i.e. minimizing the distance measure, can be achieved by numerical optimization. As this is an iterative process, image registration can be a time consuming task and speeding up the computation is an ongoing effort [54]. Clinical applications like image guided surgery [45] or ultrasound tracking [24] can greatly benefit from accelerating registration algorithms.
Over the course of the last 20 years, parallel programming gained more and more importance for writing code that uses the capabilities of modern computers. The introduction of multi-core Central Processing Units (CPUs) and the development of user-friendly frameworks for programming CPUs and Graphics Processing Units (GPUs) made it

possible to write parallelized code without great effort [22]. In contrast to the few cores of CPUs, GPUs provide thousands of threads that can work in parallel and are well suited for speeding up problems where the same program is executed on many data elements in parallel [43]. The growing capabilities and use of GPUs in a scientific environment is summarized in [46]. The authors state that current GPUs are highly parallel programmable processors that outpace their CPU counterpart and are therefore an interesting choice to accelerate algorithms. Many research groups implemented different parts of a registration algorithm on the GPU using NVIDIAs Compute Unified Device Architecture (CUDA) [12, 48, 54]. For instance, in [21] a statistical parametric mapping system was accelerated by parallelizing the computation of the rigid transformation, bilinear interpolation and joint histogram on the GPU, achieving a 14-fold speedup compared to a single-threaded CPU version. Deformable image registration was accelerated resulting in a 55-fold speedup compared to single-threaded CPU code in [39]. To gain this performance, the authors point out the importance of memory coalescing and low-level implementation. Comparable speedups were generated in [53] by running MI on the GPU.

In this thesis, a CUDA implementation of an affine linear 2D registration algorithm using SSD or NGF is presented. CUDA was chosen, since it is steadily updated, well documented and easy to learn. To my knowledge, this is the first contribution implementing the NGF distance measure on a GPU. In contrary to many contributions [12, 48, 54], the entire algorithm is executed on the GPU, thus minimizing costly data transfer and CPU load. The implementation is based on a pixelwise independent, explicit calculation rule for NGF based registration derived by Rühaak et al. [49]. This scheme was extended by using the CUDA framework and exploiting several techniques like optimized memory handling, specialized kernel invocation and the efficient use of hardware interpolation. In contrast to common practice, the computationally most demanding kernel was divided into two separate kernels, resulting in faster executions. Therefore, only by thoroughly analyzing the mathematical foundation, reconstructing the registration algorithm for parallel execution and fully utilizing the high and low-level features of the GPU, fast GPU code was gained that outperforms parallelized CPU code by far.

The thesis is organized as follows. Chapter 2 provides a brief overview of the registration framework, including a short summary of the used registration algorithm and its components. Further, the derivation of the explicit calculation rules, which form the foundation for massively parallel computation, is explained in detail. In Chapter 3, the CUDA programming model will be briefly discussed and the most important CUDA techniques that were used in order to gain very fast code are highlighted. After this, the GPU code is compared to serial and parallelized CPU code in terms of performance in Chapter 4. Finally, in Chapter 5, a conclusion and an outlook on future tasks is given.

Chapter 2: Mathematical Foundation

In this chapter an overview of the mathematical foundation of which the registration algorithm is built is given. In Section 2.1, image registration will be generally introduced, explaining the basic idea as well as the single components that compose the registration algorithm that was implemented for this work.Detailed explanations of how the SSD and NGF distance measure calculations can effectively be rearranged into a memory efficient and pixelwise independent explicit calculation rule are given in Section 2.2. This chapter is concluded with a summary in Section 2.3.

2.1 Image Registration

Medical image registration is one of the challenging problems in image computing. It describes the process of aligning two images, the reference and the template image, so that corresponding features can be matched. Corresponding features can either be structural, e.g. bones or organs, or functional, e.g. functionally equivalent regions in different brain images. Images are typically of the same context, e.g. a scene taken at different time or view points or with an emphasis on different key features. In a medical sense, applications could relate to aligning a series of computed tomography images of a lung taken at different times or aligning computed tomography images showing the anatomy with functional magnetic resonance images showing the brain activity. Image registration is needed for various tasks in medicine such as monitoring disease progression, planning and guiding surgery or dose delivery verification [20].
There are many different registration algorithms that use either rigid and affine or non-rigid transformations [33] to match images. Non-rigid transformations usually allow a high degree of freedom for the transformation parameters to account for non-linear transformations. Therefore, registration algorithms using non-linear transformations often have high computation times. Since the aim is to write an extremely fast registration algorithm, affine transformations are used. 2D affine transformations are parameterized by six transformation parameters that can account for object translations and rotations as well as scaling and shearing. Only six parameters are used and the transformations can be computed faster than non-rigid transformations. Affine transformations have been successfully used in many different medical scenarios [32].
To measure the correspondence between the transformed template and reference image, different methods have been proposed [9]. Assuming that pixel values in the reference and template are comparable, i.e. the images are of the same modality and similar pixel values correlate to the same object in both images, the Sum of Squared Differences (SSD) is a simple, yet powerful distance measure [20] and used in different applications [23, 18]. Pixel intensities of images with different modalities may not correlate with the same ob-

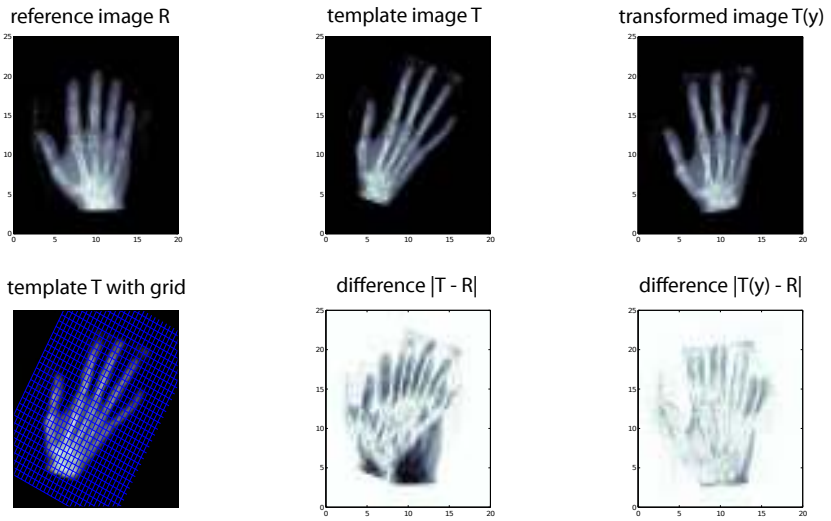

Figure 1: Example registration of two hand images.

ject and other distance measures have to be used. The Normalized Gradient Fields [17] distance measure evaluates the alignment of image gradients, which are persistent and do not change with the modality. It is well suited for optimization and fast computation. Other distance measures, like Normalized Cross Correlation [37] or Mutual Information [7, 61], are not discussed, since they either do not produce better results or are computationally more demanding than SSD or NGF.
Since both distance measures can be used to express the registration problem as a least-squares problem, Newton-type optimization is used to solve it. More specific, a Gauss-Newton optimization scheme [40] is used to compute a solution as it offers good numerical stability, fast convergence and can be implemented in a computationally efficient way [37].
In the following sections an overview of the different components of image registration will be given, staying close to the registration framework used in [37]. First, the general registration framework is discussed in Section 2.1.1. In Section 2.1.2 parametric transformations are explained, followed by linear interpolation in Section 2.1.3. Hereafter, insight on the two distance measures SSD and NGF is given in Sections 2.1.4 and 2.1.5, respectively. The multilevel approach is explained in Section 2.1.6. This part concludes with an overview of the complete registration algorithm in Section 2.1.7.

2.1.1 Registration Framework

Image registration is the process of finding a reasonable transformation y, that describes the spatial correspondence between two images, a reference image $\mathcal{R}$ and a template image $\mathcal{T}$. This is typically done by minimizing a distance measure $\mathcal{D}$ that depends on the transformation y [37].
The images are defined as continuous mappings $\mathcal{R} : \Omega_{\mathcal{R}} \to \mathbb{R}$ and $\mathcal{T} : \Omega_{\mathcal{T}} \to \mathbb{R}$ with compact support in the domains $\Omega_{\mathcal{R}} \subset \mathbb{R}^2$ and $\Omega_{\mathcal{T}} \subset \mathbb{R}^2$ in 2D, respectively. Note that calligraphic letters are used for continuous functionals.
The transformation $y : \Omega_{\mathcal{R}} \to \mathbb{R}^2$ is a-priori unknown and can be characterized as a minimizer of an optimization problem. This problem often describes the correspondence of the images by evaluating the image distance. Images showing similar structures in similar locations are considered to have high correspondence. This similarity is frequently depicted by a distance measure depending on the reference $\mathcal{R}$, the template $\mathcal{T}$, the transformation y and function ϕ and can be stated as

$$\mathcal{D}\left[\mathcal{T}[y], \mathcal{R}\right] = \int_{\Omega_{\mathcal{R}}} \phi(\mathcal{T}[y], \mathcal{R})\, dx.$$

Over the course of years, many different distance measures have been proposed, including *Mutual Information* [7, 61], *Normalized Cross Correlation* [37] or *Normalized Gradient Fields* [17] for multimodal images as well as *Sum of Squared Differences* [3] for monomodal images. Now, finding a reasonable correspondence between the images is done by solving the optimization problem through minimizing the objective function

$$\mathcal{D}(\mathcal{T}(y), \mathcal{R}) =: \mathcal{D}(y) \xrightarrow{y} \min, \tag{1}$$

where $\mathcal{T}(y)$ denotes the transformed template image. The transformations $y : \Omega_{\mathcal{R}} \to \mathbb{R}^2$ maps the reference domain to the template domain. The minimization of (1) is done by using the discretize-then-optimize ansatz [37]. The transformation and distance measure are discretized, resulting in a continuous, yet n-dimensional optimization problem, where $n \in \mathbb{N}$. The images are discretized using a grid of size $M \times N$, $M, N \in \mathbb{N}$ and pixel values are interpolated at the center of the grid cells. Therefore, rather than transforming the template image itself, the grid is deformed and the template image is interpolated at the transformed grid points.
This is the so called Eulerian approach. The Lagrangian framework, which follows tissue points, is not used and therefore not explained.
Figure 1 shows exemplary images of aligning two hands using affine-linear transformations. The top row displays the reference and template image before and after the registration, respectively. The bottom row shows the template image with the transformed grid and the differences before and after the registration, respectively. The bottom left image shows that rather transforming the image itself, a grid is transformed

and the template image is interpolated at the transformed grid. It also shows, that the transformation is not intuitive. The counterclockwise rotation of the template requires a clockwise rotation of the grid.

2.1.2 Affine Linear Transformations

In order to solve the optimization problem (1), we concentrate on affine linear transformations, which belong to the parametrized transformations. They have a fixed, often small, number of transformation parameters w as opposed to high degree of freedom for the transformation parameters of non-rigid transformations. Parametrized transformations preserve straight lines and planes as well as the parallelism of lines and can be defined as functions $y_w : \Omega_{\mathcal{R}} \to \mathbb{R}^2$.
For example, one of the easiest transformations is a translation of an image which can be written as

$$\begin{pmatrix} y^1 \\ y^2 \end{pmatrix} = \begin{pmatrix} x^1 \\ x^2 \end{pmatrix} + \begin{pmatrix} w_1 \\ w_2 \end{pmatrix},$$

where x^1, x^2 are the components of a single point $\mathbf{x} = (x^1, x^2)^\top \in \mathbb{R}^2$. Note that superscript indices identify components of vectors and subscript indices are used for numbering.
Introducing additional rotation of the image gives rigid transformations. Here, y_w maps a single point $\mathbf{x}$ with $y_w : \mathbf{x} \mapsto A\mathbf{x} + \mathbf{t}, \mathbf{t} = (w_2, w_3)^\top$, and $A := A(w_1)$ is a rotation matrix

$$A(w_1) = \begin{pmatrix} \cos(w_1) & -\sin(w_1) \\ \sin(w_1) & \cos(w_1) \end{pmatrix},$$

where w_1 denotes a rotation angle and $\mathbf{t}$ describes a translation in x^1 and x^2 direction. The transformation $y_w : \Omega_{\mathcal{R}} \to \mathbb{R}^2$ enables a comparison of the reference image $\mathcal{R}$ and the transformed image $\mathcal{T}(y_w) := \mathcal{T} \circ y_w$ by mapping the reference image domain to the template image domain depending on the parameters w. The components are given by

$$\begin{aligned} y^1 &= \cos(w_1)x^1 - \sin(w_1)x^2 + w_2, \\ y^2 &= \sin(w_1)x^1 + \cos(w_1)x^2 + w_3, \end{aligned}$$

where $w = [w_1; w_2; w_3] \in \mathbb{R}^3$ parameterizes the transformation.
Affine linear transformations can be realized by choosing an arbitrary transformation matrix A. Here, $y_w : \mathbf{x} \mapsto A\mathbf{x} + \mathbf{t}$, where A is a matrix and $\mathbf{t}$ is a vector defined as

$$A = \begin{pmatrix} w_1 & w_2 \\ w_4 & w_5 \end{pmatrix}, \ \mathbf{t} = (w_3, w_6)^\top, \tag{2}$$

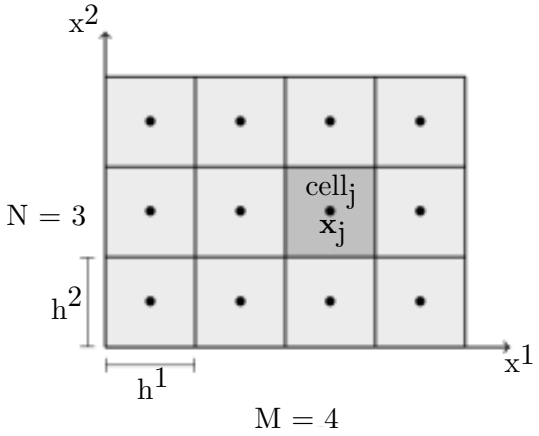

Figure 2: Discretization of a 2D domain Ω [37].

where $w = (w_1, \ldots, w_6)^\top$ can be chosen completely arbitrary, as long as $\det A \neq 0$ is satisfied. Doing so allows for additional scaling and shearing and the components are given by

$$\begin{aligned} y^1 &= w_1 x^1 + w_2 x^2 + w_3, \\ y^2 &= w_4 x^1 + w_5 x^2 + w_6, \end{aligned} \tag{3}$$

where $w = [w_1; \ldots; w_6] \in \mathbb{R}^6$ parameterizes the transformation.

2.1.3 Linear Interpolation

In order to find a solution to the optimization problem (1), the continuous model is discretized, thus enabling the use of numerical optimization. Therefore the reference domain $\Omega_{\mathcal{R}}$ is discretized into $M \times N$ cells. These cells are equally sized in x^1 and x^2 direction, where h^1 and h^2 define the step width in x^1 and x^2 direction, respectively. Data is given at the cell-centered points $\mathbf{x}_j = \left(x_j^1, x_j^2\right) \in \mathbb{R}^2$, $j = 1, \ldots, MN$, as illustrated in Figure 2. Using this, we define

$$R_j = \mathcal{R}(\mathbf{x}_j) \quad \text{and} \quad T_j = \mathcal{T}(\mathbf{x}_j),$$

where $j = 1, \ldots, MN$, $\mathcal{R}(\mathbf{x}_j) = 0 \; \forall \mathbf{x}_j \notin \Omega_{\mathcal{R}}$ and $\mathcal{T}(\mathbf{x}_j) = 0 \; \forall \mathbf{x}_j \notin \Omega_{\mathcal{T}}$.
The transformed grid usually does not coincide with these points. Thus in order to compute the transformed template image $T(y_w) := \mathcal{T}(y_w(\mathbf{x}))$ interpolation is needed. Though spline interpolation is smoother and differentiable at all points, linear interpolation is used for several reasons. The main object of this work is to accelerate an established registration algorithm and linear interpolation benefits from low computational costs.

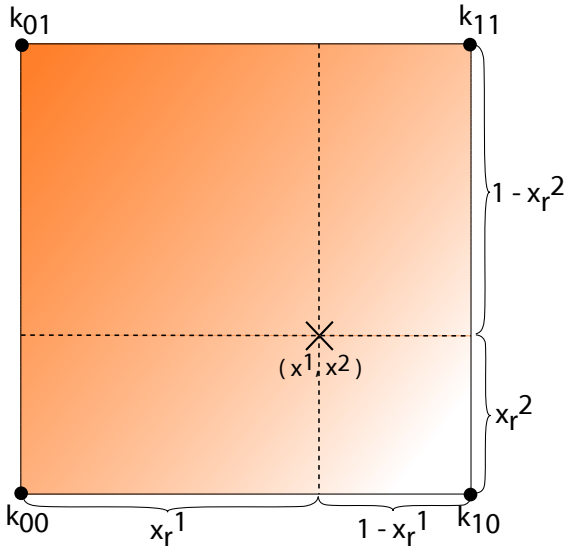

Figure 3: Simple representation of the bilinear pixel interpolation at coordinates (x^1, x^2).

Considering a unit square, the bilinear interpolation $p : \mathbb{R}^2 \to \mathbb{R}$ of a pixel value at the coordinates (x^1, x^2) can be written as

$$p = (1 - x_r^1)((1 - x_r^2)k_{00} + x_r^2 k_{01}) + x_r^1((1 - x_r^2)k_{10} + x_r^2 k_{11}), \tag{4}$$

where $k_{00} \cdots k_{11}$ are known pixel values and $x_r^i = x^i - \lfloor x^i \rfloor$, $i = 1, 2$ are remainders, as illustrated in Figure 3. The remainders are used to normalize the range of the square. The analytical derivative of (4), needed in the optimization process, is defined as

$$\begin{aligned} \frac{\partial p}{\partial x^1} &= (1 - x_r^2)(k_{10} - k_{00}) + x_r^2(k_{11} - k_{01}), \\ \frac{\partial p}{\partial x^2} &= (1 - x_r^1)(k_{01} - k_{00}) + x_r^1(k_{11} - k_{10}). \end{aligned} \tag{5}$$

2.1.4 Sum of Squared Differences Distance Measure

Once the template is transformed, a distance between it and the reference image needs to be measured as stated in the optimization problem (1). The Sum of Squared Differences distance measure is a simple, yet powerful distance measure and can be written as

$$\mathcal{D}_{\mathrm{SSD}}(w) = \frac{1}{2} \int_{\Omega_{\mathcal{R}}} (\mathcal{T}(y_w(\mathbf{x})) - \mathcal{R}(\mathbf{x}))^2 \, d\mathbf{x}. \tag{6}$$

In order to compute the integral, we discretize the continuous formulation using the mid-point quadrature rule. This leads to the discretized version D of the distance measure (6)

and can be written as

$$D_{\mathrm{SSD}}(w) = \frac{\bar{h}}{2} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j))^2,$$

where $\bar{h} = h^1 h^2$ defines the area of each cell. By discretizing the images as well the formulation can be reduced to

$$D_{\mathrm{SSD}}(w) = \frac{\bar{h}}{2} \sum_{j=1}^{MN} (T_j(y_w) - R_j)^2, \tag{7}$$

Eq. (7) discloses the simplicity of the SSD distance measure. The differences for each pixel of the reference and transformed template images are squared and summed up. This is perfectly sufficient in a monomodal setting, where pixel values are directly comparable, but does not perform well in a multimodal setting. To simplify further calculation steps, the formulation is shortened by the use of a vector-valued residual function r and reads

$$D_{\mathrm{SSD}}(w) =: \frac{\bar{h}}{2} \sum_{j=1}^{MN} r_j^2 =: \psi_{\mathrm{SSD}}(r(T(y_w))), \tag{8}$$

where $r_j := (T_j(y_w) - R_j)$ and $\psi_{\mathrm{SSD}} : \mathbb{R}^{MN} \to \mathbb{R}$.

2.1.5 Normalized Gradients Field Distance Measure

SSD directly compares pixel values of the images $\mathcal{R}(\mathbf{x})$ and $\mathcal{T}(y_w(\mathbf{x}))$, assuming that intensities correspond. This is disadvantageous in a multimodal setting such as shown in Figure 4. Mutual Information is widely used but due to its very general approach it is very highly non-convex [37]. This leads to many local minima, which is not preferable since the goal of minimizing (1) is to find a global minimum.

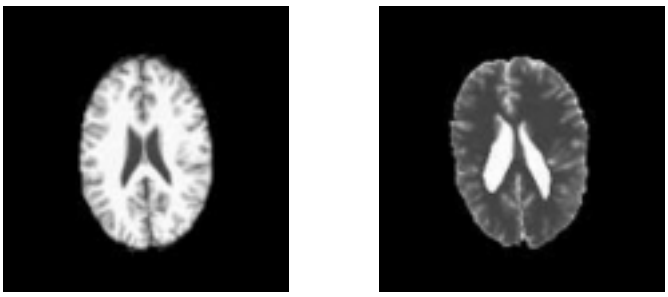

Figure 4: MRI sections of a head with T1 (left) and T2 (right) weighting. Pixel intensities of the same objects do not correspond, making a direct comparison difficult.

As proposed in [17] a distance measure based on Normalized Gradient Fields is used, which is both well suited for optimization and fast computation and can also handle multimodal images. Image gradients refer to a change of intensity and in many scenarios these intensity changes are assumably spatially correspondent. Because the gradient also indicates the magnitude of the change, which is an unwanted quantity, the gradient $\nabla\mathcal{T}$ is replaced by the normalized gradient $\nabla\mathcal{T}/|\nabla\mathcal{T}|$. The normalized gradient field is defined by

$$\mathrm{n}[\mathcal{T}] = \mathrm{n}[\mathcal{T}, \eta] = \frac{\nabla\mathcal{T}}{\sqrt{|\nabla\mathcal{T}|^2 + \eta^2}},$$

where η is an edge parameter that differentiates between edges $|\nabla\mathcal{T}| > \eta$ and noise $|\nabla\mathcal{T}| \leq \eta$. Using this, the correspondence of the gradient fields for the reference and template image can be measured by examine the orientation of the gradients. The main idea is to pointwise measure the angle between the two image gradients and to align these in either parallel or antiparallel fashion. Considering that the scalar product of two vectors is zero if they are orthogonal and maximal if they are parallel the continuous NGF distance measure is defined as

$$\mathcal{D}_{\mathrm{NGF}}(w) = \int_{\Omega_{\mathcal{R}}} 1 - \left(\frac{\langle \nabla\mathcal{T}(y_w(\mathbf{x})), \nabla\mathcal{R}(\mathbf{x}) \rangle}{\|\nabla\mathcal{T}(y_w(\mathbf{x}))\|_\eta \|\nabla\mathcal{R}(\mathbf{x})\|_\eta} \right)^2 d\mathbf{x}, \tag{9}$$

where $\|\cdot\|_\eta = \sqrt{|\cdot|^2 + \eta^2}$. Similar to discretizing the SSD distance measure, the continuous NGF distance measure (9) is discretized by also applying the midpoint quadrature rule. Additionally, finite differences are used to compute the various image gradients yielding

$$D_{\mathrm{NGF}}(w) = \bar{h} \sum_{j=1}^{MN} \left(1 - \left(\frac{\langle g_j(\mathcal{T}(y_w(\mathbf{x}))), g_j(\mathcal{R}(\mathbf{x})) \rangle}{\|g_j(\mathcal{T}(y_w(\mathbf{x})))\|_\eta \|g_j(\mathcal{R}(\mathbf{x}))\|_\eta} \right)^2 \right),$$

where g_j is the approximation to the image gradient through central finite differences defined as

$$g_j(\mathcal{T}(\mathbf{x})) := \begin{pmatrix} \frac{1}{2h^1}(-\mathcal{T}(\mathbf{x}_{j-1}) + \mathcal{T}(\mathbf{x}_{j+1})) \\ \frac{1}{2h^2}(-\mathcal{T}(\mathbf{x}_{j-M}) + \mathcal{T}(\mathbf{x}_{j+M})) \end{pmatrix}$$

Finally, using the discretized formulation for the images the NGF distance measure reads

$$D_{\mathrm{NGF}}(w) = \bar{h} \sum_{j=1}^{MN} \left(1 - \left(\frac{\langle g_j(T(y_w)), g_j(R) \rangle}{\|g_j(T(y_w))\|_\eta \|g_j(R)\|_\eta} \right)^2 \right). \tag{10}$$

For each pixel of the reference and template image, the scalar product of the approximated gradients is computed and normalized by dividing through the magnitudes of the

gradients. If the gradients at point j are parallel, the division yields 1 and thus the contribution to the distance is zero. Similar to the SSD distance measure the formulation can be shortened through the use of a residual function and can finally be written as

$$D_{\mathrm{NGF}}(w) =: \bar{h} \sum_{j=1}^{MN} \left(1 - r_j^2\right) =: \psi_{\mathrm{NGF}}(r(T(y_w))), \tag{11}$$

where $r_j := \frac{\langle g_j(T(y_w)), g_j(R) \rangle}{\|g_j(T(y_w))\|_\eta \|g_j(R)\|_\eta}$ and $\psi_{\mathrm{NGF}} : \mathbb{R}^{MN} \to \mathbb{R}$.

2.1.6 Multilevel Approach

In order to speed up the optimization process, the images are sampled from a very coarse to a very fine resolution. By averaging adjacent cells into one cell, the intensity values are smoothed and the resolution is decreased. Now, the registration starts on the coarsest resolution and the problem is solved with relatively low computational costs. The transformation parameters are then used as an initial guess for finer resolutions, thus reducing the amount of iterations until a minimum if found.
Assuming that an image is of a size that is a power of two, i.e.

$$m_l = 2^l, l \in [l_{min}, l_{min} + 1, l_{min} + 2, \ldots, l_{max}], l_{min} < l_{max} \in \mathbb{N},$$

and $T^l \in \mathbb{R}^{m_l \times m_l}$. Here, l_{max} is the finest and also initial level and l_{min} is the coarsest level. A multilevel representation is obtained by

$$\begin{gathered} T^{l-1}[i,j] = \\ \left(T^l[2i,2j] + T^l[2i+1,2j] + T^l[2i,2j+1] + T^l[2i+1,2j+1]\right)/4, \end{gathered} \tag{12}$$

where $i, j \in [0, 1, \ldots, m_{l-1}]$. The image size is chosen, so that a multilevel representation down to a resolution of 2×2 can be guaranteed. Figure 5 illustrates this process.
It is noticeable from a computational point of view, that the total memory allocation for the multilevel data in 2D is never greater than one third of the original data. The size of

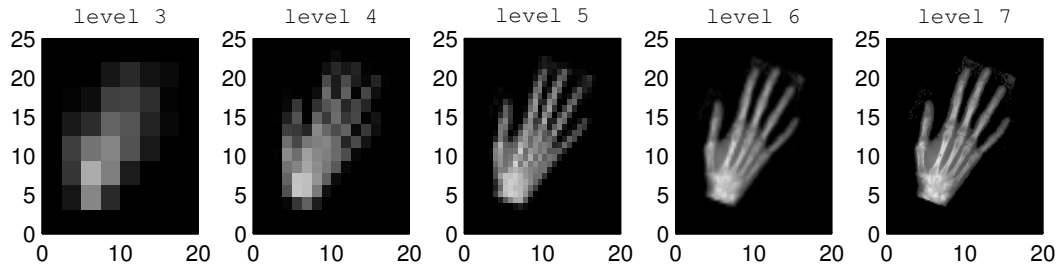

Figure 5: Multilevel representation of a hand.

each level is only one quarter of the preceding level, thus the memory requirement can be formulated as a geometric series

$$\sum_{n=0}^{\infty} \frac{1}{4}^n.$$

The limit of a geometric series is given by

$$\sum_{n=0}^{\infty} a_0 q^n = \lim_{n\to\infty} a_0 \frac{1-q^{n+1}}{1-q} = a_0 \frac{1}{1-q}.$$

Now, setting $a_0 = 1$ and $q = 1/4$ the limit of the series is

$$\frac{1}{1-\frac{1}{4}} = \frac{4}{3},$$

showing that no matter how many levels the representation has, the total size is limited to one third.

2.1.7 Gauss-Newton Optimization

Now that all the needed ingredients have been explained, a minimizer of the optimization problem (1) can be computed using numerical optimization. More specific a multilevel Gauss-Newton approach [40] was chosen. Unlike the Newton's method [40] for finding a minimum of a function, the Gauss-Newton method does not require second derivatives, which are often hard to compute and susceptible to noise. Further, it offers good numerical stability, fast convergence and can be implemented in a computationally efficient way [37].
A pseudo code describing the multilevel Gauss-Newton optimization is shown in Algorithm 1. In Step 1, a multilevel representation of the images in different resolutions is generated. Starting on the coarsest level, the images are set to their multilevel representation and the transformation parameters are initialized, as seen in Steps 2 to 4. Steps 5 to 11 describe the actual Gauss-Newton algorithm. First, the objective function is evaluated. The objective function reads $\mathcal{J}(w) = \mathcal{D}\left[\mathcal{T}(y_w(\mathbf{x})), \mathcal{R}(\mathbf{x})\right]$ and is defined as the distance between the reference and transformed template image. The evaluation of the objective function at the current transformation parameters w is the computation of the function value, i.e. $D(w) = \psi(r(T(y_w)))$, the gradient ∇D and the approximation to the Hessian H. In Step 7 common stopping rules [16] are checked. For the current iteration n, these stopping rules read

Stop 1: $|J_{n-1} - J_n| \quad \leq \text{tolJ}(1 + |J_0|)$

Stop 2: $\|(w_{n-1} - w_n)\| \quad \leq \text{tolW}(1 + \|w_n\|)$

Stop 3: $\bar{h} \cdot \|dJ_n\| \quad \leq \text{tolG}(1 + |J_0|)$

Stop 4: $\|dJ_n\| \quad \leq 10^{-7}$

Stop 5: iter $\quad > \text{maxIter}$

where J_0 is the initial function value and tolJ, tolW and tolG are tolerance values chosen according to the optimization problem. The first three rules measure the variation in the function value, parameters and gradient, respectively. If all three numbers are small, the iteration terminates. As safeguards, rule 4 and 5 are implemented. If the variation in the gradient is very small, i.e. in the area of machine precision, or if a maximum number of iterations maxIter is reached, the iteration terminates as well. The Quasi-Newton system is solved and the descent direction is computed in Steps 8 and 9. In the setting of affine linear transformations this system is small, i.e. $H \in \mathbb{R}^{6\times 6}$ and $\nabla D, dw \in \mathbb{R}^6$, and can be solved with the aid of pivoting and LU decomposition [60]. A standard Armijo line-search [40] is performed in Step 10 in order to guarantee a sufficient decrease in the objective function. After the Gauss- Newton optimization terminates for any reason, the final parameters are saved for the next level, as seen in Step 12. Steps 2 to 12 are repeated until the finest level is reached and the final transformation parameters are calculated.

Algorithm 1 Pseudo code for the multilevel Gauss-Newton optimization

1: get $T_{\text{level}}, R_{\text{level}} \; \forall$ level ▷ Compute multilevel representation
2: **for** level $\leftarrow$ minLevel, level $\leq$ maxLevel **do** ▷ Start multilevel loop
3: $T \leftarrow T_{\text{level}}, R \leftarrow R_{\text{level}}$ ▷ Set images to current level
4: $w \leftarrow w_0$ ▷ Set initial transformation parameters
5: **while** stopping rules not active **do** ▷ Start Gauss-Newton optimization
6: $[D, \nabla D, H] \leftarrow \text{evalObjFctn}(w, T, R)$ ▷ Evaluate objective function
7: $\text{stoppingRules} \leftarrow \text{stoppingRules}(w)$ ▷ Evaluate stopping rules
8: $H \cdot dw = -\nabla D$ ▷ Solve for dw
9: $v_{\text{descent}} \leftarrow \nabla D \cdot dw$ ▷ Compute descent direction
10: $w \leftarrow \text{lineSearch}(\text{ObjFctn}, w, D, v_{\text{descent}})$ ▷ Perform Armijo line-search
11: **end while**
12: $w_0 \leftarrow w$, level $\leftarrow$ level $+ 1$ ▷ Save parameters for next level
13: **end for**

2.2 Explicit Calculation Rules

The evaluation of the objective function as needed in the Gauss-Newton optimization scheme can be done in several ways. In [37] a modular, matrix-based approach is chosen. Here, the transformation, interpolation and distance measure calculation are expressed as matrix operations. Using this ansatz, the class of transformations, interpolation or distance measure can easily be interchanged. Therefore, this approach is very versatile, yet slow to compute, since it is not optimized for a certain kind of transformation, interpolation and distance measure.
Rühaak et al. [49] proposed a fully parallelizable, matrix-free computation of the 3D NGF distance measure. This enabled an extremely fast, on the fly evaluation of the objective function with minimal memory requirements. Based on their concepts and ideas, fully parallelizable and matrix-free 2D *explicit calculation rules* for the SSD and NGF computation were derived.
In this section insight on the derivation of these rules is provided, starting with some general remarks in Section 2.2.1. The calculation of the function value, gradient and approximation to the Hessian for the SSD distance measure is shown in Section 2.2.2. In Section 2.2.3 the differences for the NGF distance measure are discussed and the calculation rule for NGF is presented.

2.2.1 General Concept

In order to derive an efficient and pointwise independent explicit calculation rule for the function value D, the gradient ∇D and the approximation to the Hessian H, a close analysis of their mathematical composition is inevitable. The straightforward matrix-based approach, as used in the image registration toolbox FAIR [37], is hard to implement efficiently on the GPU with respect to memory usage and execution time. Therefore, the internal structures of the single matrices are exposed and exploited. Many of these matrices exhibit a sparse structure. By carefully dissecting the interdependency of the single entries, a very memory and time conserving problem specific algorithm can be derived. The same principles are used to derive closed form formulas for the gradient and Hessian computation as well.

2.2.2 Sum of Squared Differences

In this section, the derivation of explicit calculation rules for the SSD function value and derivatives are explained in detail.

Preliminary Considerations The cascaded formulation of the SSD distance measure in (8) reveals that the optimization problem can be stated as a minimization of a function

$D_{\text{SSD}} : \mathbb{R}^6 \to \mathbb{R}$. The six parameters of the affine transformation are mapped to one function value that represents the distance between the transformed template image and the reference image. The cascaded formulation (8) can be decomposed into vector-valued functions which are defined as

$$y : \mathbb{R}^6 \to \mathbb{R}^{2MN}, \quad \begin{pmatrix} w_1 \\ \vdots \\ w_6 \end{pmatrix} \mapsto \begin{pmatrix} (A\mathbf{x}_1 + \mathbf{t})_1 \\ (A\mathbf{x}_1 + \mathbf{t})_2 \\ \vdots \\ (A\mathbf{x}_{MN} + \mathbf{t})_1 \\ (A\mathbf{x}_{MN} + \mathbf{t})_2 \end{pmatrix}, \tag{13}$$

where $A, \mathbf{t}$ are defined as in Equation (2). Here, the transformation y maps six parameters w to a vector of $2MN$ transformed points. By using $\mathbf{y} = \left(y^1, y^2\right)^\top$ the evaluation of the template image at the transformed points can be written as

$$T : \mathbb{R}^{2MN} \to \mathbb{R}^{MN}, \quad \begin{pmatrix} \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_{MN} \end{pmatrix} \mapsto \begin{pmatrix} \mathcal{T}(\mathbf{y}_1) \\ \vdots \\ \mathcal{T}(\mathbf{y}_{MN}) \end{pmatrix}. \tag{14}$$

The residual is then formulated as

$$r : \mathbb{R}^{MN} \to \mathbb{R}^{MN}, \quad \begin{pmatrix} T_1 \\ \vdots \\ T_{MN} \end{pmatrix} \mapsto \begin{pmatrix} T_1 - R_1 \\ \vdots \\ T_{MN} - R_{MN} \end{pmatrix}, \tag{15}$$

leading to the definition of the outer function as the sum of the squared residual elements

$$\psi : \mathbb{R}^{MN} \to \mathbb{R}, \quad \begin{pmatrix} r_1 \\ \vdots \\ r_{MN} \end{pmatrix} \mapsto \frac{\bar{h}}{2} \sum_{j=1}^{MN} r_j^2. \tag{16}$$

Thus, the discretized objective function of the SSD distance measure (7) can be seen as a concatenation of the functions

$$D_{\text{SSD}} : \mathbb{R}^6 \xrightarrow{y} \mathbb{R}^{2MN} \xrightarrow{T} \mathbb{R}^{MN} \xrightarrow{r} \mathbb{R}^{MN} \xrightarrow{\psi} \mathbb{R}.$$

Using the above formulation and the chain rule, the analytical gradient is given by

$$\nabla D_{\text{SSD}}(w) = \frac{\partial \psi}{\partial r}[r(T(y(w)))] \cdot \frac{\partial r}{\partial T}[T(y(w))] \cdot \frac{\partial T}{\partial y}[y(w)] \cdot \frac{\partial y}{\partial w}[w]. \tag{17}$$

Again, we examine the components individually. The derivative of the first factor is given by

$$\frac{\partial \psi}{\partial r} = \bar{h}(r_1, \ldots, r_{MN}) \in \mathbb{R}^{1 \times MN}.$$

The derivative of the residual function is defined as

$$\frac{\partial r}{\partial T} = I_{MN} \in \mathbb{R}^{MN \times MN},$$

where I_{MN} is the identity matrix. Therefore, the derivative of the residual function has no influence on the computation of the analytical gradient and can be omitted. The derivative of the template image at point $\mathbf{x} = \left(x^1, x^2\right)$ is obtained by using the notation of Equation (5) and can be written as

$$\begin{aligned}
\partial_1 \mathcal{T}(\mathbf{x}) = \frac{\partial p}{\partial x_r^1} &= (1 - x_r^2)(k_{10} - k_{00}) + x_r^2(k_{11} - k_{01}), \\
\partial_2 \mathcal{T}(\mathbf{x}) = \frac{\partial p}{\partial x_r^2} &= (1 - x_r^1)(k_{01} - k_{00}) + x_r^1(k_{11} - k_{10}),
\end{aligned}$$

with the corresponding Jacobian

$$\frac{\partial T}{\partial y} = \begin{pmatrix}
\partial_1 \mathcal{T}(\mathbf{y}_1) & \partial_2 \mathcal{T}(\mathbf{y}_1) & & & & & & \\
& & \partial_1 \mathcal{T}(\mathbf{y}_2) & \partial_2 \mathcal{T}(\mathbf{y}_2) & & & & \\
& & & & \ddots & \ddots & & \\
& & & & & & \partial_1 \mathcal{T}(\mathbf{y}_{MN}) & \partial_2 \mathcal{T}(\mathbf{y}_{MN})
\end{pmatrix},$$

$$\frac{\partial T}{\partial y} \in \mathbb{R}^{MN \times 2MN}. \tag{18}$$

Following Equation (3) the derivative of the transformation y with respect to the parameters w reads

$$\frac{\partial y}{\partial w} = \begin{pmatrix}
x_1^1 & x_1^2 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & x_1^1 & x_1^2 & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
x_{MN}^1 & x_{MN}^2 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & x_{MN}^1 & x_{MN}^2 & 1
\end{pmatrix} \in \mathbb{R}^{2MN \times 6}, \tag{19}$$

completing the analysis of the single components of the analytical gradient $\nabla D_{\mathrm{SSD}}(w)$ from (17). As mentioned in Section 2.1.7, through the use of Gauss-Newton optimization the computation of second order image derivatives, which are highly susceptible to noise, is not needed. Instead the Hessian is replaced by an approximation that is given by

$$\nabla^2 D_{\mathrm{SSD}}(w) \approx H_{\mathrm{SSD}}(w) := \bar{h}\ dr^\top dr, \tag{20}$$

where $dr := \frac{\partial r}{\partial T}[T(y(w))] \cdot \frac{\partial T}{\partial y}[y(w)] \cdot \frac{\partial y}{\partial w}[w]$.

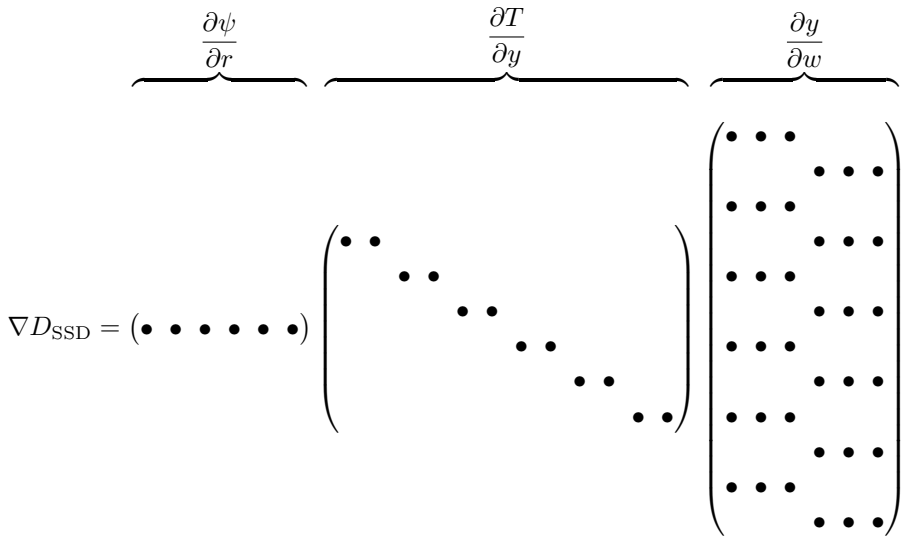

Figure 6: Schematic view of the sparse matrix structure in the computation of ∇D_{SSD}.

Problem Specific Derivative Calculation In the previous segment the derivation for the computation of the function value D_{SSD}, the gradient ∇D_{SSD} and the approximation to the Hessian $\nabla^2 D_{\mathrm{SSD}}$ was shown. However, the computations are matrix-based and further analysis is required in order to obtain pointwise independent calculation rules. Therefore, the sparse structure of the matrices, shown in Figure 6, that compose the gradient (17) are exploited. The complete analytical gradient is defined as

$$\nabla D_{\mathrm{SSD}} = \left(\partial_{w_1} D_{\mathrm{SSD}}, \partial_{w_2} D_{\mathrm{SSD}}, \partial_{w_3} D_{\mathrm{SSD}}, \partial_{w_4} D_{\mathrm{SSD}}, \partial_{w_5} D_{\mathrm{SSD}}, \partial_{w_6} D_{\mathrm{SSD}}\right).$$

By knowing the interdependency of the matrices, the entries of the gradient can then be written as

$$\begin{aligned}
\partial_{w_1} D_{\mathrm{SSD}}(w) &= \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_1 \mathcal{T}(y_w(\mathbf{x}_j)) \cdot x_j^1 \\
\partial_{w_2} D_{\mathrm{SSD}}(w) &= \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_1 \mathcal{T}(y_w(\mathbf{x}_j)) \cdot x_j^2 \\
\partial_{w_3} D_{\mathrm{SSD}}(w) &= \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_1 \mathcal{T}(y_w(\mathbf{x}_j)) \\
\partial_{w_4} D_{\mathrm{SSD}}(w) &= \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_2 \mathcal{T}(y_w(\mathbf{x}_j)) \cdot x_j^1
\end{aligned} \tag{21}$$

$$\partial_{w_5} D_{\mathrm{SSD}}(w) = \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_2 \mathcal{T}(y_w(\mathbf{x}_j)) \cdot x_j^2$$

$$\partial_{w_6} D_{\mathrm{SSD}}(w) = \bar{h} \sum_{j=1}^{MN} (\mathcal{T}(y_w(\mathbf{x}_j)) - \mathcal{R}(\mathbf{x}_j)) \cdot \partial_2 \mathcal{T}(y_w(\mathbf{x}_j)).$$

The same approach is used to compute the entries of the approximation H to the Hessian $\nabla^2 D_{\mathrm{SSD}}$. First, we define $dx_j^1 = \partial_1 \mathcal{T}(y_w(\mathbf{x}_j))$, $dx^1 x_j^1 = dx_j^1 \cdot x_j^1$, $dx^1 x_j^2 = dx_j^1 \cdot x_j^2$ and $dx_j^2 = \partial_2 \mathcal{T}(y_w(\mathbf{x}_j))$, $dx^2 x_j^1 = dx_j^2 \cdot x_j^1$, $dx^2 x_j^2 = dx_j^2 \cdot x_j^2$. Finally, we phrase the approximation as

$$H_{\mathrm{SSD}}(w) = \bar{h} \sum_{j=1}^{MN} l_j, \tag{22}$$

with

$$l_j := \begin{pmatrix} dx^1 x_j^1 \cdot dx^1 x_j^1 & dx^1 x_j^1 \cdot dx^1 x_j^2 & dx^1 x_j^1 \cdot dx_j^1 & dx^1 x_j^1 \cdot dx^2 x_j^1 & dx^1 x_j^1 \cdot dx^2 x_j^2 & dx^1 x_j^1 \cdot dx_j^2 \\ \bullet & dx^1 x_j^2 \cdot dx^1 x_j^2 & dx^1 x_j^2 \cdot dx_j^1 & dx^1 x_j^2 \cdot dx^2 x_j^1 & dx^1 x_j^2 \cdot dx^2 x_j^2 & dx^1 x_j^2 \cdot dx_j^2 \\ \bullet & \bullet & dx_j^1 \cdot dx_j^1 & dx_j^1 \cdot dx^2 x_j^1 & dx_j^1 \cdot dx^2 x_j^2 & dx_j^1 \cdot dx_j^2 \\ \bullet & \bullet & \bullet & dx^2 x_j^1 \cdot dx^2 x_j^1 & dx^2 x_j^1 \cdot dx^2 x_j^2 & dx^2 x_j^1 \cdot dx_j^2 \\ \bullet & \bullet & \bullet & \bullet & dx^2 x_j^2 \cdot dx^2 x_j^2 & dx^2 x_j^2 \cdot dx_j^2 \\ \bullet & \bullet & \bullet & \bullet & \bullet & dx_j^2 \cdot dx_j^2 \end{pmatrix}.$$

For clarification only the upper triangular part is shown in Equation (22). Since H is symmetric by definition the lower triangular part is easy to fill by mirroring the upper part. Both, the explicit calculation rule for the gradient and the approximation to the Hessian, only depend on the j-th pixel and can be evaluated independently. In addition, there are two other handy advantages. First, the memory costs are kept at minimum as everything is computed on the fly, whereas in the matrix-based approach the deformed template and various sparse matrices among other data need to be stored. Second, equations for the function value D_{SSD} (7), the gradient ∇D_{SSD} (21) and the approximation to the Hessian H_{SSD} (22) can be directly parallelized pixelwise. There are no interdependencies between the pixels, thus allowing the computation of the values for every pixel at once.

2.2.3 Normalized Gradient Fields

In the previous section, explicit calculation rules for the SSD function value and derivatives were derived. The same ideas are applied to derive calculation rules for the NGF function value and derivatives. For the effective computation of the function value the discretized formulation from Equation (10) can straightforwardly be used. To derive the calculation of the gradient we follow the example given in Section 2.2.2.

Preliminary Considerations The cascaded formulation of the NGF distance measure (11) can also be decomposed, yet there are several differences to the SSD distance measure. Compared to the definitions of the residual (15) and outer (16) function of the SSD, the outer function for the NGF is defined as

$$\psi : \mathbb{R}^{MN} \to \mathbb{R}, \begin{pmatrix} r_1 \\ \vdots \\ r_{MN} \end{pmatrix} \mapsto \bar{h} \sum_{j=1}^{MN} \left(1 - r_j^2\right),$$

and the residual function is written as

$$r : \mathbb{R}^{MN} \to \mathbb{R}^{MN}, \begin{pmatrix} T_1 \\ \vdots \\ T_{MN} \end{pmatrix} \mapsto \begin{pmatrix} \left(\frac{s(g_1(T))}{\|g_1(T)\|_\eta \|g_1(R)\|_\eta}\right)^2 \\ \vdots \\ \left(\frac{s(g_{MN}(T))}{\|g_{MN}(T)\|_\eta \|g_{MN}(R)\|_\eta}\right)^2 \end{pmatrix},$$

where s is the scalar product of two vectors define as

$$s : \mathbb{R}^2 \to \mathbb{R},\ a \mapsto \langle a, g_j(R) \rangle,$$

with $a \in \mathbb{R}^2$ and g_j is an approximation to the image gradient using central finite differences

$$g_j : \mathbb{R}^{MN} \to \mathbb{R}^2,\ T \mapsto \begin{pmatrix} \frac{1}{2h^1}\left(-T_{j-1} + T_{j+1}\right) \\ \frac{1}{2h^2}\left(-T_{j-M} + T_{j+M}\right) \end{pmatrix},$$

The other two terms are defined just as in Equations (13) and (14) and the discretized objective function of the NGF distance measure (10) can also be seen as a concatenation of the functions

$$D_{\mathrm{NGF}} : \mathbb{R}^6 \xrightarrow{y} \mathbb{R}^{2MN} \xrightarrow{T} \mathbb{R}^{MN} \xrightarrow{r} \mathbb{R}^{MN} \xrightarrow{\psi} \mathbb{R}.$$

Identical to (17), the analytical gradient can be written as

$$\nabla D_{\mathrm{NGF}}(w) = \frac{\partial \psi}{\partial r}[r(T(y(w)))] \cdot \frac{\partial r}{\partial T}[T(y(w))] \cdot \frac{\partial T}{\partial y}[y(w)] \cdot \frac{\partial y}{\partial w}[w]. \tag{23}$$

Here, the partial derivatives for the first two factors differ from the SSD distance measure. The derivative of the first factor is defined as

$$\frac{\partial \psi}{\partial r} = -2\bar{h}(r_1, \ldots, r_{MN}) \in \mathbb{R}^{1 \times MN}.$$

We define

$$r_j : \mathbb{R} \to \mathbb{R},\ T_j \mapsto \left(\frac{s(g_j(T))}{\|g_j(T)\|_\eta \|g_j(R)\|_\eta}\right)^2$$

as the j-entry of r. Now, using the quotient rule and $r_j := \frac{r_{j_1}}{r_{j_2}}$ we can write the derivative of r_j as

$$\frac{\partial r_j}{\partial T} = \frac{1}{r_{j_2}^2}\left(\frac{\partial r_{j_1}}{\partial T} r_{j_2} - r_{j_1}\frac{\partial r_{j_2}}{\partial T}\right). \tag{24}$$

Further, the derivative of the dividend is given by

$$\frac{\partial r_{j_1}}{\partial T} = \frac{\partial s(g_j(T))}{\partial T} = \frac{\partial s}{\partial g_j(T)}\frac{\partial g_j(T)}{\partial T}.$$

This leads to

$$\frac{\partial s}{\partial g_j(T)} = \left(\frac{1}{2h^1}\left(-R_{j-1} + R_{j+1}\right), \frac{1}{2h^2}\left(-R_{j-M} + R_{j+M}\right)\right) \in \mathbb{R}^{1\times 2}$$

and

$$\frac{\partial g_j(T)}{\partial T} = \begin{pmatrix} \cdots & 0 & \cdots & \overbrace{-\frac{1}{2h^1}}^{j-1} & 0 & \overbrace{\frac{1}{2h^1}}^{j+1} & \cdots & 0 & \cdots \\ \cdots & \underbrace{-\frac{1}{2h^2}}_{j-M} & \cdots & 0 & 0 & 0 & \cdots & \underbrace{\frac{1}{2h^2}}_{j+M} & \cdots \end{pmatrix} \in \mathbb{R}^{2\times MN}, \tag{25}$$

where the entries of the matrix are zero except at the positions $j-M$, $j-1$, $j+1$, $j+M$. We are using zero Neumann boundary conditions and specify the values of the derivative on the boundary as zero. Finally we get

$$\frac{\partial r_{j_1}}{\partial T} = \begin{pmatrix} \vdots \\ \frac{1}{4(h^2)^2}\left(R_{j-M} - R_{j+M}\right) \\ \vdots \\ \frac{1}{4(h^1)^2}\left(R_{j-1} - R_{j+1}\right) \\ 0 \\ \frac{1}{4(h^1)^2}\left(R_{j+1} - R_{j-1}\right) \\ \vdots \\ \frac{1}{4(h^2)^2}\left(R_{j+M} - R_{j-M}\right) \\ \vdots \end{pmatrix}^\top \in \mathbb{R}^{1\times MN}. \tag{26}$$

Here, the index of the first operand denotes the position of the entry in the vector, e.g. the entry $R_{j-M} - R_{j+M}$ is found on the position $j-M$. The rest of the entries are again zero.

The derivative of the divisor is given by

$$\frac{\partial r_{j_2}}{\partial T} = \frac{\partial \|g_j(T)\|_\eta \|g_j(R)\|_\eta}{\partial T} = \|g_j(R)\|_\eta \frac{\partial \|\cdot\|_\eta}{\partial g_j(T)}\frac{\partial g_j(T)}{\partial T}.$$

Further is

$$\frac{\partial \|\cdot\|_\eta}{\partial g_j(T)} = \frac{1}{\|g_j(T)\|_\eta} g_j(T)^\top,$$

so that with the aid of (25) follows

$$\begin{aligned}\frac{\partial r_{j_2}}{\partial T} &= \frac{\|g_j(R)\|_\eta}{\|g_j(T)\|_\eta} g_j(T)^\top \frac{\partial g_j(T)}{\partial T} \\ &= \frac{\|g_j(R)\|_\eta}{\|g_j(T)\|_\eta} \begin{pmatrix} \vdots \\ \frac{1}{4(h^2)^2}\left(T_{j-M} - T_{j+M}\right) \\ \vdots \\ \frac{1}{4(h^1)^2}\left(T_{j-1} - T_{j+1}\right) \\ 0 \\ \frac{1}{4(h^1)^2}\left(T_{j+1} - T_{j-1}\right) \\ \vdots \\ \frac{1}{4(h^2)^2}\left(T_{j+M} - T_{j-M}\right) \\ \vdots \end{pmatrix}^\top \in \mathbb{R}^{1\times MN}. \end{aligned} \tag{27}$$

Combining (26) and (27) we can write (24) as

$$\frac{\partial r_j}{\partial T} = \begin{pmatrix} \vdots \\ \frac{1}{4(h^2)^2}\left[\frac{R_{j-M}-R_{j+M}}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta} - \frac{\langle g_j(T), g_j(R)\rangle}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta^3}\left(T_{j-M} - T_{j+M}\right)\right] \\ \vdots \\ \frac{1}{4(h^1)^2}\left[\frac{R_{j-1}-R_{j+1}}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta} - \frac{\langle g_j(T), g_j(R)\rangle}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta^3}\left(T_{j-1} - T_{j+1}\right)\right] \\ 0 \\ \frac{1}{4(h^1)^2}\left[\frac{R_{j+1}-R_{j-1}}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta} - \frac{\langle g_j(T), g_j(R)\rangle}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta^3}\left(T_{j+1} - T_{j-1}\right)\right] \\ \vdots \\ \frac{1}{4(h^2)^2}\left[\frac{R_{j+M}-R_{j-M}}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta} - \frac{\langle g_j(T), g_j(R)\rangle}{\|g_j(R)\|_\eta \|g_j(T)\|_\eta^3}\left(T_{j+M} - T_{j-M}\right)\right] \\ \vdots \end{pmatrix}^\top \tag{28}$$

or

$$\frac{\partial r_j}{\partial T} =: \begin{pmatrix} \partial r_{j_1} \\ \vdots \\ \partial r_{j_{MN}} \end{pmatrix}^\top \in \mathbb{R}^{1\times MN}.$$

This finally leads to

$$\frac{\partial r}{\partial T} = \begin{pmatrix} \frac{\partial r_1}{\partial T} \\ \frac{\partial r_2}{\partial T} \\ \vdots \\ \frac{\partial r_{MN}}{\partial T} \end{pmatrix} \in \mathbb{R}^{MN \times MN}. \tag{29}$$

The other two factors that compose $\nabla D_{\mathrm{NGF}}(w)$ as in Equation (23) are the same as in Equations (18) and (19). Now, using $\frac{\partial \psi}{\partial r} = -2\bar{h}r^\top$, the gradient of the NGF distance measure results in

$$\nabla D_{\mathrm{NGF}}(w) = -2\bar{h} \sum_{j=1}^{MN} r_j dr_j,$$

where $dr_j = \frac{\partial r_j}{\partial T} \frac{\partial T}{\partial y} \frac{\partial y}{\partial w}$. The approximation to the Hessian is given by

$$\nabla^2 D_{\mathrm{NGF}}(w) \approx H_{\mathrm{NGF}} := 2\bar{h} \sum_{j=1}^{MN} dr_j^\top dr_j.$$

$$\frac{\partial r}{\partial T} =$$

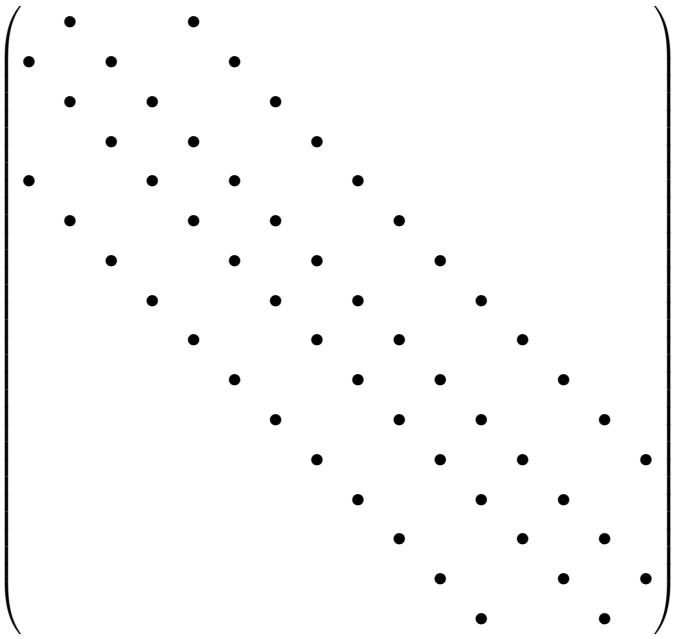

Figure 7: Schematic view of the sparse matrix structure of $\frac{\partial r}{\partial T}$ for the computation of gradient ∇D_{NGF}.

Problem Specific Derivative Calculation Analog to the problem specific derivative calculation for the SSD distance measure, compact formulations for the NGF distance measure are derived. The layout of the matrices is the same as in Figure 6. The difference is that $\frac{\partial r}{\partial T}$ is not the identity matrix anymore and thus plays a role in the gradient computation. In this case, $\frac{\partial r}{\partial T}$ is a sparse matrix with a maximum of only four entries per row, as seen in Equations (28) and (29) and Figure 7. Exploiting the sparse matrix structure, dr_j can be written as

$$dr_j = \begin{pmatrix} \partial r_{j_{j-M}} \partial_1 T_{j-M} x^1_{j-M} + r_{j_{j-1}} \partial_1 T_{j-1} x^1_{j-1} \\ +\partial r_{j_{j+1}} \partial_1 T_{j+1} x^1_{j+1} + \partial r_{j_{j+M}} \partial_1 T_{j+M} x^1_{j+M} \\ \\ \partial r_{j_{j-M}} \partial_1 T_{j-M} x^2_{j-M} + r_{j_{j-1}} \partial_1 T_{j-1} x^2_{j-1} \\ +\partial r_{j_{j+1}} \partial_1 T_{j+1} x^2_{j+1} + \partial r_{j_{j+M}} \partial_1 T_{j+M} x^2_{j+M} \\ \\ \partial r_{j_{j-M}} \partial_1 T_{j-M} + r_{j_{j-1}} \partial_1 T_{j-1} \\ +\partial r_{j_{j+1}} \partial_1 T_{j+1} + \partial r_{j_{j+M}} \partial_1 T_{j+M} \\ \\ \partial r_{j_{j-M}} \partial_2 T_{j-M} x^1_{j-M} + r_{j_{j-1}} \partial_2 T_{j-1} x^2_{j-1} \\ +\partial r_{j_{j+1}} \partial_2 T_{j+1} x^1_{j+1} + \partial r_{j_{j+M}} \partial_2 T_{j+M} x^1_{j+M} \\ \\ \partial r_{j_{j-M}} \partial_2 T_{j-M} x^2_{j-M} + r_{j_{j-1}} \partial_2 T_{j-1} x^2_{j-1} \\ +\partial r_{j_{j+1}} \partial_2 T_{j+1} x^2_{j+1} + \partial r_{j_{j+M}} \partial_2 T_{j+M} x^2_{j+M} \\ \\ \partial r_{j_{j-M}} \partial_2 T_{j-M} + r_{j_{j-1}} \partial_2 T_{j-1} \\ +\partial r_{j_{j+1}} \partial_2 T_{j+1} + \partial r_{j_{j+M}} \partial_2 T_{j+M} \end{pmatrix}.$$

Using this, the components which compose the gradient are defined as

$$\begin{aligned} \partial_{w_1} D_{\mathrm{NGF}}(w) &= -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[1] \\ \partial_{w_2} D_{\mathrm{NGF}}(w) &= -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[2] \\ \partial_{w_3} D_{\mathrm{NGF}}(w) &= -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[3] \\ \partial_{w_4} D_{\mathrm{NGF}}(w) &= -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[4] \\ \partial_{w_5} D_{\mathrm{NGF}}(w) &= -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[5] \end{aligned} \tag{30}$$

$$\partial_{w_6} D_{\mathrm{NGF}}(w) = -2\bar{h} \sum_{j=1}^{MN} r_j \cdot dr_j[6],$$

where $dr_j[i]$, $i = 1, \ldots, 6$ is the i-th entry of the vector and $r_j = \frac{\langle g(T_j(y_w)), g_j(R) \rangle}{\|g(T_j(y_w))\|_\eta \|g_j(R)\|_\eta}$, see Equation (10) and (11). The complete analytical gradient reads

$$\nabla D_{\mathrm{NGF}} = \left(\partial_{w_1} D_{\mathrm{NGF}}, \partial_{w_2} D_{\mathrm{NGF}}, \partial_{w_3} D_{\mathrm{NGF}}, \partial_{w_4} D_{\mathrm{NGF}}, \partial_{w_5} D_{\mathrm{NGF}}, \partial_{w_6} D_{\mathrm{NGF}}\right).$$

The approximation to the Hessian can then be phrased as

$$H_{\mathrm{NGF}}(w) = 2\bar{h} \sum_{j=1}^{MN} l_j, \tag{31}$$

with

$$l_j := \begin{pmatrix} dr_j[1] \cdot dr_j[1] & dr_j[1] \cdot dr_j[2] & dr_j[1] \cdot dr_j[3] & dr_j[1] \cdot dr_j[4] & dr_j[1] \cdot dr_j[5] & dr_j[1] \cdot dr_j[6] \\ \bullet & dr_j[2] \cdot dr_j[2] & dr_j[2] \cdot dr_j[3] & dr_j[2] \cdot dr_j[4] & dr_j[2] \cdot dr_j[5] & dr_j[2] \cdot dr_j[6] \\ \bullet & \bullet & dr_j[3] \cdot dr_j[3] & dr_j[3] \cdot dr_j[4] & dr_j[3] \cdot dr_j[5] & dr_j[3] \cdot dr_j[6] \\ \bullet & \bullet & \bullet & dr_j[4] \cdot dr_j[4] & dr_j[4] \cdot dr_j[5] & dr_j[4] \cdot dr_j[6] \\ \bullet & \bullet & \bullet & \bullet & dr_j[5] \cdot dr_j[5] & dr_j[5] \cdot dr_j[6] \\ \bullet & \bullet & \bullet & \bullet & \bullet & dr_j[6] \cdot dr_j[6] \end{pmatrix}.$$

For clarification only the upper triangular part is shown in Equation (31) as the lower triangular part can easily be filled by mirroring the upper part. Also, the two advantages presented at the end of Section 2.2.2 apply here as well, thus enabling a fast and memory efficient computation of the NGF distance.
This concludes the lengthy derivation of the explicit calculation rules for the SSD and NGF distance measures.

2.3 Summary

In this chapter, the mathematical foundation of the registration algorithm was explained. First, an overview of all the needed components for the registration algorithm was given. Using these components and with the help of pseudo code in Algorithm 1, the affine linear multilevel registration algorithm was then discussed in detail. In the second part of this chapter, explicit calculation rules for the SSD and NGF distance measures were derived. This was achieved by carefully analyzing the sparse matrix structures that compose the distance measure calculations and examine the interdependencies of the single entries. Thus, it was possible to derive pixelwise independent, memory efficient explicit calculation rules that can be effectively parallelized and computed using GPUs and NVIDIA's CUDA.
In the next chapter, the key features of CUDA will be explained and the GPU implementation will be discussed in detail.

Chapter 3: General-Purpose Computing on Graphics Processing Units

In this chapter, NVIDIA's Compute Unified Device Architecture (CUDA) is introduced. It is used to implement the algorithm and a detailed explanation about the most important features will be given. Starting with a brief introduction to CUDA in Section 3.1, the reasons why CUDA was chosen are stated and the programming model and memory layout are explained. An overview of the workstation on which all experiments were conducted is given in Section 3.2. In Section 3.3 the most important CUDA techniques in order to gain high performing code are analyzed. This chapter concludes with a summary in Section 3.4.

3.1 An Introduction to GPGPU

Starting with the turn of the century, graphics processing units (GPUs) became an interesting choice for computations other than graphics related. Programmable shaders and floating point support on GPUs made them a target for highly parallel numerical workloads and the term general-purpose computing on graphics processing units (GPGPU) was introduced.
Implementations of matrix-matrix multiplications on the GPU date back to 2001 [25] but suffered from slow memory bandwidth and were not faster than optimized CPU code. One of the first applications that actually ran faster on a GPU than on a CPU was a LU decomposition [14].
However, writing code for the GPU was difficult, since the two major application programming interfaces (APIs) were Silicon Graphics Inc. OpenGL [56] and Microsoft DirectX [30]. They are shading languages and programmers must express numerical computations in graphics terms, e.g. textures or vertices [46]. The introduction of high level languages like BrookGPU [4] allowed for stream programming [47]. Streams are similar to arrays, but it is possible to work on every element in parallel.
The advent of NVIDIA's CUDA [50] in 2007 improved the performance for many applications and simplified GPU programming [11]. Much like Khronos Group's Open Computing Language (OpenCL) [58], it features a C-like syntax, but offers more low-level features to fully utilize the capabilities of the GPU.
These and ongoing developments allowed programmers to write GPU code more easily and for more complex applications. The importance of GPGPU is known to hardware companies and a new focus is laid on novel technological features that further simplify GPU programming. For example, CUDA's latest major release of version 6.0 features *Unified Memory*, memory that is accessible to CPU and GPU and is managed by the

system rather than the developer.
These and past developments made GPGPU an interesting choice to accelerate different applications in medical imaging [34, 54, 55] as well as many other applications in many other fields of research [47, 46].
Section 3.1.1 explains why a GPU implementation was chosen over a parallelized CPU implementation. More insight into the CUDA framework will be given in Section 3.1.2 by starting with an discussion about why CUDA was selected over other computing languages. Thereafter, the programming model will be explained in Sections 3.1.3 and illustrated with the help of a simple example in Section 3.1.4. The different types of memory including their features are discussed in Section 3.1.5.

3.1.1 Why GPGPU?

There are many options in order to improve the computational performance of an algorithm. On the one hand the algorithm itself can be analyzed and optimized in terms of avoiding unnecessary computations and operations. On the other hand the implementation of the algorithm has a big impact on the performance as well. The first option was dealt with in Section 2.2 and problem specific, memory efficient calculation rules for computationally expensive operations of the registration algorithm were derived. In this chapter, the implementation of the algorithm itself is dealt with.
The explicit calculation rules derived in Section 2.2, expose pixelwise parallelism, which makes them a perfect choice for massively parallel computing. Modern computing devices, starting from ordinary personal computers to huge computer clusters or even mobile phones, offer multi-core processors or GPUs that enable parallel computing.

Parallelizing CPU code can be done by using the Open Multi-Processing (OpenMP) API [10]. The programmer can run C/C++ code in parallel through the use of compiler

Figure 8: Schematic design of a CPU and GPU. Whereas the CPU devotes a large portion of transistors to flow control and data caching, the GPU provides more arithmetic logic units (ALUs) needed for data processing [43].

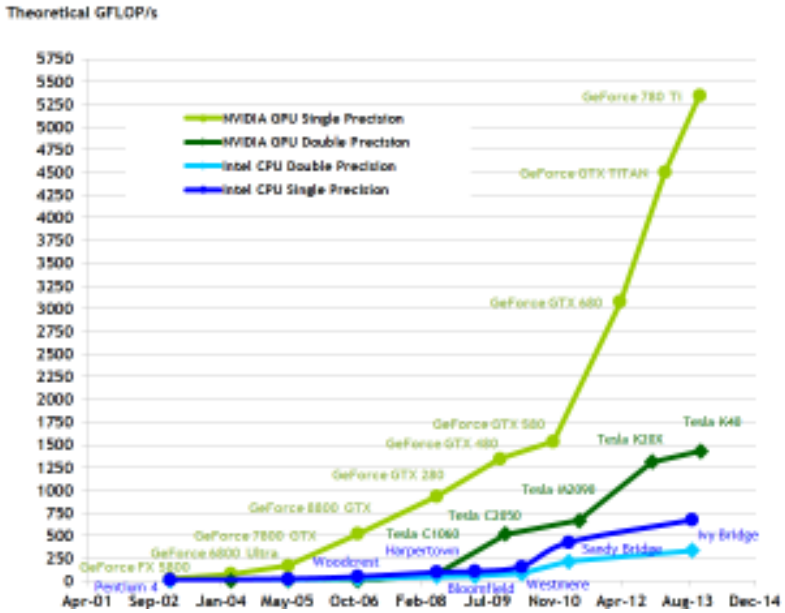

Figure 9: Comparison of the theoretical peak performance of various CPUs and GPUs [43].

directives.

GPU code can be created through the use of high level programming languages like OpenCL or CUDA. Current GPUs provide thousands of small cores designed to handle multiple tasks simultaneously, whereas CPUs only consist of up to 16 cores (AMD Opteron) with many transistors devoted to different tasks such as data caching and flow control [43], schematically illustrated by Figure 8.

The number of single precision floating point operations per second on current GPUs is nearly ten times higher compared to CPUs, as seen in Figure 9. This shows, that the computational much more powerful device is the GPU.

So, why is not every computation done on a GPU? One downside of graphics cards is the data transfer. Whereas the transfer does not matter for CPU computations, since the data is already available in main memory, it needs to be transferred from the main RAM to the GPU memory before it can be accessed. This results in a non-negligible overhead which impacts the overall runtime of the code.

Another fact to be considered is the ratio of serial and parallel code. If a program runs mainly serial and only a little portion of the code runs parallel, the performance benefit of using a GPU will be small. This is described by Amdahl's law [1], which reads

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}, \tag{32}$$

where S is the theoretical speedup, P is the portion of the code that can be parallelized and N is the number of processors. As an illustrative example, let 90% of the code be parallelizable. Amdahl's law then states that no matter how many cores are used, the maximum speedup is capped at 10. Therefore, GPUs require a very high amount of parallelism to outperform CPUs.
Since the computational most expensive part of the algorithm is pixelwise parallelizable and data transfer between the CPU and GPU is kept at a minimum, see Section 3.3.1, implementing a GPU version of the algorithm was very promising. That said, there is no general opinion on whether to implement parallelizable code on CPUs or GPUs. The programmer has to consider many factors to find an optimal solution and in many cases highly optimized CPU code can compete with GPU code [62, 26].

3.1.2 Why CUDA?

The implementation of the registration algorithm was done using CUDA for numerous reasons. The main reason is that CUDA offers a good trade-off between low-level and high-level features. The high-level features and the C-like syntax make it easy to learn and use. Programmers do not need to understand OpenGL or DirectX APIs or restructure the problems in terms of graphics primitives [50]. Yet, the low-level features enable the developer to unleash the full power of the GPU.
The use of different memory spaces and problem specific kernel layout and launch are mandatory in obtaining high performing code, as will become clear in Section 3.3. Moreover, CUDA is steadily updated and very well documented. Compared to OpenCL CUDA offers better memory and thread handling and faster transfer rates [55, 29].

These reasons make CUDA a widely used framework by researchers across the world and for different tasks like weather prediction [35], molecular dynamics [28] or medical image registration [5, 39, 52, 59].
Disadvantages of using CUDA are the restriction to NVIDIA GPUs and the low performance when using double precision floating point operations. The latter requires the programmer to find a reasonable trade-off between runtime and accuracy.

3.1.3 Programming Model

In order to briefly explain the principles behind the CUDA framework, the CPU and the system's memory is referred to as the *host* and the GPU and its memory as the *device*, staying close to common literature [43, 50, 63].
CUDA uses a single-program multiple-data programming model, allowing the user to pass a program, called *kernel*, to the device. The kernels will execute N times in parallel

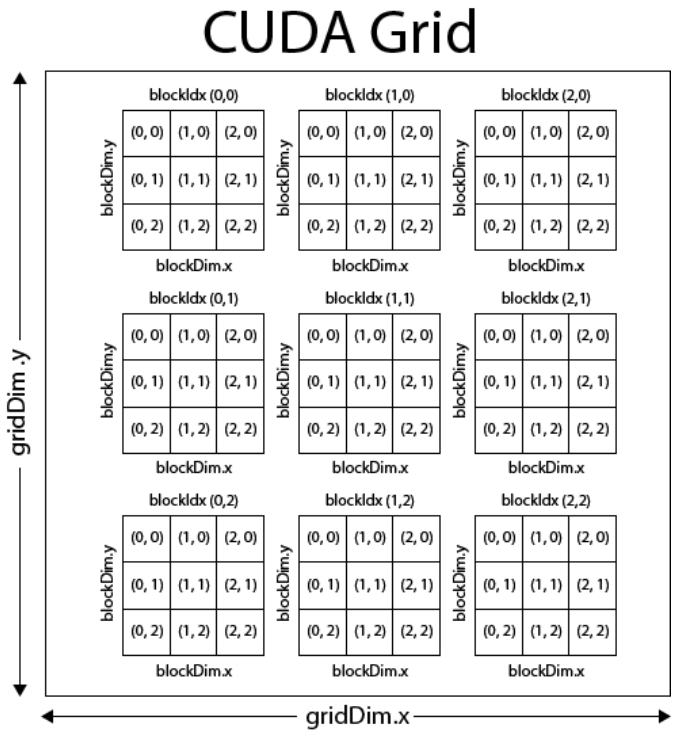

Figure 10: Thread, block and grid hierarchy [44].

by N different CUDA *threads*[1]. This makes it similar to the single-instruction multiple-data model, yet CUDA allows branching within the kernel, providing a much broader instruction set.
Threads are identified by a d-dimensional *thread index* forming d-dimensional *thread blocks*, with $d = 1, 2, 3$. Again, these blocks are identified by a *block index* and grouped into a d-dimensional *grid*, as shown in Figure 10.
Threads are executed in groups of 32 parallel threads called *warps*[1]. In order to gain full efficiency, all 32 threads of the warp should execute the same instruction. Otherwise, each instruction branch will be executed serially resulting in increased execution time [43].
The exact layout of the thread blocks and grid is defined by the user when calling a kernel. The thread blocks of the grid are then distributed to available *streaming multiprocessors* (SMs). A SM can concurrently execute threads of a thread block as well as multiple thread blocks. After the termination of a thread block, the vacancy is filled by launching new thread blocks.

[1]The terms *thread* and *warp* are an analogy to the weaving industry, one of the first technologies using parallel threads.

```
1  // define kernel
2  __global__ void add(int *a, int *b, int *result)
3  {
4      // compute global thread index
5      const int tid = blockIdx.x*blockDim.x + threadIdx.x;
6
7      // elementwise addition of the vectors
8      if(tid<N)
9          result[tid] = a[tid] + b[tid];
10 }
11
12 // main routine, analog to every other C main routine
13 int main(void)
14 {
15     // initialize arrays on host
16     int h_a[N], h_b[N], h_result[N];
17
18     // fill arrays with values
19     ...
20
21     // initialize arrays on device
22     int *d_a, *d_b, *d_result;
23
24     // allocate memory on device
25     cudaMalloc( (void**)&d_a, N*sizeof(int) );
26     ...
27
28     // transfer data from host to device
29     cudaMemcpy(d_a, h_a, sizeof(int)*N, cudaMemcpyHostToDevice);
30     ...
31
32     // define grid and block layout
33     dim3 gridDim( 1, 1, 1 );
34     dim3 blockDim( N, 1, 1 );
35
36     // call kernel
37     add<<<gridDim, blockDim>>>(d_a, d_b, d_result);
38
39     // transfer data from device to host
40     cudaMemcpy(h_result, d_result, sizeof(int)*N, cudaMemcpyDeviceToHost);
41
42     // do whatever you want with the result ...
43
44     // free allocated memory on the GPU
45     cudaFree(d_a);
46     ...
47
48     return 0;
49 }
```

Figure 11: Pseudo C code showing a simple CUDA program that performs the elementwise addition of two vectors and saves the result to a third vector.

3.1.4 CUDA Example

To illustrate a simple kernel call, Figure 11 shows pseudo-code for a very basic example of adding two vectors and storing the result in a third vector. Lines $1 - 10$ define the kernel. The `__global__` qualifier is an indicator for the compiler that the function should run on the device rather than on the host. Other than that, the function looks the same as a simple C function. The global thread index is computed in line 5 depending on the current block index, the block dimension and the local thread index.
Additional to the kernel definition, some more CUDA specific commands in the main function are needed. Arrays on the GPU need to be initialized and allocated. The latter is done with the `cudaMalloc()` function, see line 25. It behaves much like the standard C function `malloc()` but it tells the compiler to allocate memory on the device. This shows how thin the line between standard C and CUDA really is. It leaves the responsibility to the programmer to differentiate between memory allocated on the device and on the host. This is typically done by naming variables in the Hungarian notation.
After reserving enough space, data needs to be copied unto the device. This is done by calling `cudaMemcpy()`. Again, it behaves exactly like the standard C `memcpy()` with the difference that a *direction* is specified in the last argument. Simple enough, memory can be copied from the host to the device or vice versa, as shown in lines 29 and 40. If both pointers remain on the device `cudaMemcpyDeviceToDevice` is passed.
Now, the data is on the device and everything is set to call the kernel and to do some work. The kernel call is shown in line 37 and looks much like a standard C function call, except for the angle brackets. Inside the angle brackets the grid and thread block layout is defined. These can be special variables of the type `dim3` as seen in lines 33 and 34. Both, grid and thread blocks, are three dimensional structures and in this case the grid contains one thread block which holds N threads in form of a long vector. When calling the kernel each of the threads will perform one pair-wise addition and store the result in the result vector.
The threads are not launched in sequence. Thus, it is important to ensure that the computation can be done in any order. This ensures that the thread blocks can be scheduled in any order across all cores, so that the code scales with the number of cores. Since all of the threads of one thread block reside on the same processor core, the number of threads per thread block is limited [43]. Current GPUs support up to 1024 threads per thread block. So, if the size N of the vectors exceeds 1024, additional blocks need to be launched. It is common practice to have a fixed number of threads per block and to launch $\lceil \frac{N}{\text{threads}} \rceil$ blocks.
After the kernel was executed the result is copied back to the host. In order to free the memory space that was allocated on the device, `cudaFree()` needs to be called, which also behaves exactly as `free()` does.
Before the program can be run, it needs to be compiled. For that, CUDA comes with

its own NVIDIA CUDA compiler driver `nvcc`. The device functions are compiled using NVIDIA compilers/assemblers, the host functions are handed-off to a supported C compiler. The compiled GPU functions are then embedded in the host object file [42].
This concludes the illustrative example in order to get a grasp of the principle behavior of a CUDA program. For a more detailed introduction, the reader is referred to [50].

3.1.5 Memory Layout

CUDA threads can access data from different memory spaces, as shown in Figure 12. Device memory is located off-chip. Therefore, it is visible to all threads, but very slow to access directly. *Global memory*, *constant memory* and *texture memory* reside in the device memory space. *Global memory* offers the largest space with up to 12 GB on current GPUs, but cannot be cached.
In contrary, *constant memory* is cached in the constant memory cache. Therefore, only cache misses result in reads from the device memory, otherwise data will be read from the constant cache. A read from constant memory can be broadcasted to nearby threads within a warp and consecutive reads do not incur additional memory traffic.
Similarly to constant memory, *texture memory* resides in device memory and is cached in texture cache. Only cache misses result in reading from device memory, otherwise data will be *fetched* from texture cache. Textures are optimized for 2D/3D spatial read-out patterns, by using a specialized address calculation. High performance is achieved

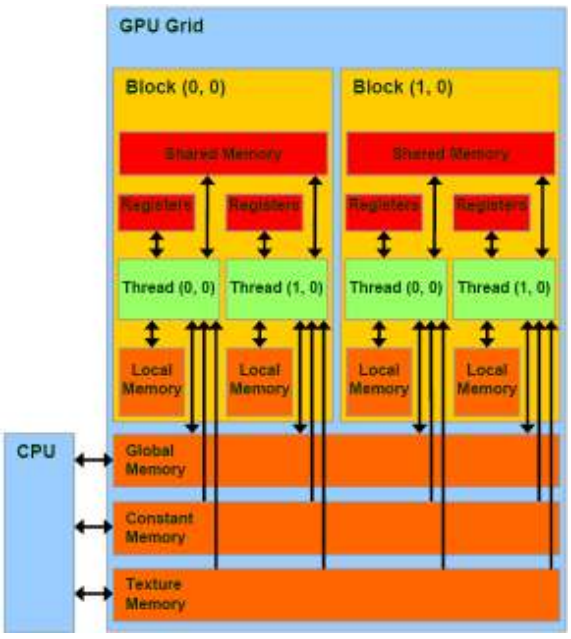

Figure 12: Schematic view of the memory hierarchy [43].

Figure 13: Different addressing for global (left) and texture memory (right).

00	01	02	03
04	05	06	07
08	09	0A	0B
0C	0D	0E	0F

serial addressing

00	01	04	05
02	03	06	07
08	09	0C	0D
0A	0B	0E	0F

texture addressing

when reading from addresses that are close together in 2D/3D, as illustrated for 2D in Figure 13. Additional to the 2D/3D read-out pattern, textures offer hardware interpolation and boundary handling, by defining filtering and address modes, respectively. The filtering mode allows for nearest neighbor or linear interpolation of data. The address mode defines how to handle out of range texture fetches and provides different options like mirroring or wrapping pixel values. The downside of using texture memory is the limitation to single precision. Fetching data from the texture cache returns only single precision values. This can have an impact on the accuracy of the algorithm and has to be considered by the programmer, see [43] for more details.

Shared memory resides in cached memory and is visible to all threads of a block. Being on-chip, it provides higher bandwidth and less latency than local or global memory. The access is about 100 times faster than global memory, but the space is limited to 48 kB. Shared memory is divided into equally sized memory portions, called *banks*. If n threads address n different memory banks, they can be serviced at once, resulting in a n times higher bandwidth. Yet, if two ore more threads access the same bank, it results in a *bank conflict* and the access is serialized. The throughput is then reduced by a factor equal to the number of separate requests [41].

Other memory types, which are not discussed, are *registers* and *local memory*. They did not have a significant role in the implementation and the reader is referred to [41, 63] for additional information. A summary of all memory types is given in Table 1.

Table 1: Summary of the different memory spaces accessed by threads [41].

Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	n/a	R/W	1 thread	Thread
Local	Off-chip	†	R/W	1 thread	Thread
Shared	On-chip	n/a	R/W	All threads in block	Block
Global	Off-chip	†	R/W	All threads + host	Host allocation
Constant	Off-chip	Yes	R	All threads + host	Host allocation
Texture	Off-chip	Yes	R	All threads + host	Host allocation
†: Cached only on devices of compute capability 2.x.					

3.2 Test Environment

Unless otherwise stated, all tests and results where generated on a workstation as shown in Table 2. The CPU features four cores with a clock speed of 3.40 GHz and supports multi threading. The instruction set is 64-bit and it offers 8 MB cache. The GPU provides 1536 CUDA cores with a base clock speed of 1.046 GHz and 2 GB GDDR5 memory with a theoretical bandwidth of 224.3 GB/s. It is connected to the motherboard via a PCIe 3.0 16x slot and can achieve a theoretical bandwidth of 15.75 GB/s. The GPU offers compute capability 3.0 based on the Kepler architecture which features more CUDA cores for arithmetic operations than previous versions [43]. The DDR3 RAM has a memory clock speed of $166\frac{2}{3}$ MHz and therefore a theoretical bandwidth of 10.42 GB/s. Ubuntu 12.04 is installed as an operating system.

Table 2: Overview of the workstation used to analyze the code and generate results.

Motherboard	Asus P8P67 Evo
CPU	Intel Core i7-2600S
GPU	NVIDIA GeForce GTX 770
Memory	16 GB DDR3
OS	Ubuntu 12.04

3.3 Accelerating Affine Linear Image Registration

Writing high performing GPU code requires the developer to not only derive parallel algorithms, but also to fully utilize the capabilities of the graphics card. As mentioned in Section 3.1.2, CUDA enables programmers to unleash the full power of the GPU, but in order to do so, many features must be analyzed and used in an efficient way.
The first and naïve GPU implementation was four times slower than the competing OpenMP code. Only through the use of specialized kernel invocation, optimized memory handling and the efficient use of hardware interpolation, high performing and fast executing code was gained.
Starting with an computational analysis of the registration algorithm in Section 3.3.1, the most important techniques used to implement the algorithm are explained. With this analysis we determine which parts of the algorithm are computationally most expensive and would benefit most from parallelizing. In Section 3.3.2 specialized kernel invocation is explained and insight into the optimal grid and layout is given. The efficient use of the different memory types and the thereby enabled hardware interpolation is examined in Section 3.3.3 and 3.3.4, respectively. Thereafter, the benefits of minimizing data transfer

by running the complete algorithm on the GPU are shown in Section 3.3.5. In the final Section 3.3.6 the limitation to single precision floating point operations are discussed.

3.3.1 Parallelizable Operations

Not all parts of an algorithm are computationally so demanding that they benefit from a CUDA implementation. And those which are, are not necessarily effectively parallelizable. Additionally, there are costly operations which are called only a few times opposed to less demanding operations which are called countless times. Both may contribute the same amount of time to the overall runtime.
Thus, a thorough analysis of the key parts of the registration algorithm is needed in order to find the most time-consuming tasks. The implemented registration algorithm consists of six main tasks:

- Computing the multilevel representations for the data
- Evaluating the objective function
- Evaluating the stopping rules
- Solving the Quasi-Newton system
- Computing the descent direction
- Performing the Armijo line-search

The creation of the multilevel representations for the image data, as shown in Section 2.1.6, can be a costly task depending on the size of the input images. In general, the minimum level l_{min} was set to 3 and therefore the number of evaluations of Equation 12 can be written as

$$n = \sum_{l=3}^{l_{max}-1} 2^{2l}.$$

Considering a typical image size of 1024×1024 pixels, i.e. $l_{max} = 10$, nearly 350000 sums need to be computed. These computations are completely independent for each level and can be parallelized in a very efficient way, therefore making it a perfect task to be computed on the GPU.
The computational cost of the evaluation of the objective function depends linearly on the image size as can be seen by analyzing the Equations (7), (21), (22), (10), (30) and (31). The equations state, that sums of a constant number of operations across the number of pixels are needed. Thus, with increasing image size, the number of operations increases linearly.
The function value D, the gradient ∇D and the approximation to the Hessian H need

to be computed in every iteration, plus the function value needs to be computed at least once per Armijo line search iteration. Therefore, the main focus is laid on finding an explicit calculation rule for the SSD and NGF distance measure, as shown in Section 2.2. It was also pointed out, that these calculation rules can be directly parallelized pixelwise and are ideal for GPU computation.
The next three tasks, namely evaluating the stopping rules, solving the Quasi-Newton system and computing the descent direction, are independent of the image size and merely require a low constant number of floating point operations.
The evaluation of the stopping rules is done by a comparison of ten numbers. Solving the Quasi-Newton system for affine linear registration is done by solving a 6×6 system of linear equations. To solve $H \cdot dw = -\nabla D$ for dw a parallelized implementation of a pivotized LU decomposition is used. The descent direction is calculated by a simple scalar multiplication of two vectors $\nabla D, dw \in \mathbb{R}^6$. Hence, all three tasks have little impact on the overall performance of the code.
Other than the calculation of the function value, the Armijo line-search demands only minor computations with a constant number of floating point operations. An overview of the tasks and their computational relevance is given in Table 3.
Now, that the computational most demanding operations have been identified, the most important CUDA features used to write the CUDA code are explained. This includes specialized kernel invocation, the use of different memory types and the benefits of hardware interpolation.

Table 3: Summary of the six main tasks of the registration algorithm. In the column **Complexity**, $n \in \mathbb{N}$ is the number of image pixels, i.e. $n = MN$, and $c \in \mathbb{N}$ is a low constant number. The column **Ranking** ranks the operations according to their influence on the overall runtime. Since no clear differences could be made for the operations of complexity $\mathcal{O}(c)$ they all rank lowest.

Task	Occurrence	Complexity	Ranking
Compute multilevel data	once	$\mathcal{O}(n \log n)$	3.
Evaluate objective function	every iteration	$\mathcal{O}(n)$	1.
Evaluate stopping rules	every iteration	$\mathcal{O}(c)$	4.
Solve Quasi-Newton system	every iteration	$\mathcal{O}(c)$	4.
Compute descent direction	every iteration	$\mathcal{O}(c)$	4.
Perform Armijo line-search	every iteration	$\mathcal{O}(n)$	2.

3.3.2 Kernel Invocation

As stated in Section 3.3.1, the computationally most demanding task is evaluating the objective function and computing the scalar values that compose the function value D,

the gradient ∇D and the approximation to the Hessian H. These have to be computed at least once per optimization level, see Table 3.
Since the Gauss-Newton approximation to the Hessian is symmetric [40], it is sufficient to compute the upper triangular part of the matrix and mirror the entries to the lower triangular part. Thus, only 21 instead of 36 entries of the Hessian need to be computed. Adding the six components of the gradient and the function value itself, we get 28 scalar values in total.
Different setups for the grid layout were evaluated by changing the number of threads per block. Based on the image size and the total amount of possible active threads, blocks were allocated. Furthermore the amount of scalar values that were computed per kernel call were changed ranging from 1, 2, 4, 7, 14 and 28, respectively. These numbers were chosen in order to get an integer amount of kernel calls.
To ensure that the workload of the kernel was high, testing images of the size 4096×4096 pixels were used. The high workload minimizes the impact of the overhead that arises from multiple kernel calls. Further, single and double precision was employed when adding up the values to compute the sums in order to evaluate the impact on the overall timing of the algorithm. Tables 4 and 5 show the execution times for different setups using single and double precision, respectively.
Table 4 states that the best setup consists of two different kernels which calculate 14 scalar values each and have 64 threads per block when using single precision. Whereas the benefit of rearranging the kernel calls for single precision calculations is not great, Table 5 indicates that the execution time for double precision computations can be improved by roughly 40%. While one kernel call needs 162.4 ms at best, two kernel calls computing 14 scalar values each and having 32 threads per block need only 97.9 ms. This may be contrary to the belief that one kernel doing all the work is the best choice [5]. The used setup results in double overhead, generated by initializing kernel calls and computing needed parameters, yet the performance increases immensely.
Interestingly, the overhead is also negligible when the workload of the kernels is smaller. For images of the size 512×512 pixels the computation time can be reduced by 14% for single and 36% for double precision floating point operations compared to a single kernel launch, as indicated in Tables 6 and 7, respectively.
Developers should avoid launching only 16 threads per thread block, due to the fact that CUDA operates best with a thread number that is a multiple of 32. The tables also show that the optimal layout changes depending on the data type, making general conclusions about an optimal grid layout hard to derive. Further analysis of the impact of threads per block and the kernel layout is a topic of future work.
The use of concurrent kernel calls to improve the performance was also analyzed. Since the kernels have full workload and there is only little data to be copied, concurrent kernel calls did not enhance the overall runtime. Therefore, all kernels are called serially.

Table 4: Performance overview for different kernel setups using single precision. The table shows the computation time for calculating D, ∇D and H for an image of size 4096×4096 pixels in milliseconds with single precision. For some cases it was not possible to allocate enough shared memory, these cases are marked "out of memory" (o.o.m.). The bold values highlight the fastest execution times for each layout. The red value indicates the overall fastest execution time.

Kernel Setups, Single Precision, 4096×4096								
Number of kernel calls	Number of scalar values per kernel	Threads per block						
		16	32	64	128	256	512	1024
		Timings in milliseconds						
1	28	74.7	**45.9**	51.0	55.1	84.7	o.o.m.	o.o.m.
2	14	109.3	56.6	**44.4**	45.9	48.4	75.4	o.o.m.
4	7	189.1	96.8	80.1	**79.2**	83.7	88.6	100.8
7	4	344.2	176.8	139.3	**138.2**	148.2	161.5	173.0
14	2	657.7	334.1	**274.7**	274.9	284.9	306.2	332.7
28	1	1291.8	652.6	**547.1**	547.9	570.1	641.2	677.7

Table 5: Performance overview for different kernel setups using double precision. The table shows the computation time for calculating D, ∇D and H for an image of size 4096×4096 pixels in milliseconds with double precision. For some cases it was not possible to allocate enough shared memory, these cases are marked "out of memory" (o.o.m.). The bold values highlight the fastest execution times for each layout. The red value indicates the overall fastest execution time.

Kernel Setups, Double Precision, 4096×4096								
Number of kernel calls	Number of scalar values per kernel	Threads per block						
		16	32	64	128	256	512	1024
		Timings in milliseconds						
1	28	221.6	167.4	**162.4**	229.5	o.o.m.	o.o.m.	o.o.m.
2	14	167.7	**97.9**	102.7	101.1	153.4	o.o.m.	o.o.m.
4	7	242.4	126.2	117.3	**106.0**	109.9	147.2	o.o.m.
7	4	395.3	203.5	176.3	**164.2**	165.9	180.5	191.7
14	2	716.9	365.3	**313.2**	317.1	331.6	346.1	364.1
28	1	1351.8	684.5	**583.4**	584.3	606.1	649.3	733.4

Table 6: Performance overview for different kernel setups using single precision. The table shows the computation time for calculating D, ∇D and H for an image of size 512×512 pixels in milliseconds with single precision. For some cases it was not possible to allocate enough shared memory, these cases are marked "out of memory" (o.o.m.). The bold values highlight the fastest execution times for each layout. The red value indicates the overall fastest execution time.

Kernel Setups, Single Precision, 512×512								
Number of kernel calls	Number of scalar values per kernel	Threads per block						
		16	32	64	128	256	512	1024
		Timings in milliseconds						
1	28	1.97	**1.45**	1.58	1.67	2.07	o.o.m.	o.o.m.
2	14	2.29	1.42	**1.25**	1.33	1.39	1.62	o.o.m.
4	7	3.65	2.13	**1.87**	1.88	2.00	1.99	2.05
7	4	6.28	3.50	**2.94**	2.99	3.37	3.20	3.27
14	2	11.79	6.39	**5.45**	5.54	5.94	5.94	6.10
28	1	22.52	11.96	**10.36**	10.48	11.08	11.56	11.92

Table 7: Performance overview for different kernel setups using double precision. The table shows the computation time for calculating D, ∇D and H for an image of size 512×512 pixels in milliseconds with double precision. For some cases it was not possible to allocate enough shared memory, these cases are marked "out of memory" (o.o.m.). The bold values highlight the fastest execution times for each layout. The red value indicates the overall fastest execution time.

Kernel Setups, Double Precision, 512×512								
Number of kernel calls	Number of scalar values per kernel	Threads per block						
		16	32	64	128	256	512	1024
		Timings in milliseconds						
1	28	4.28	3.37	**3.25**	4.36	o.o.m.	o.o.m.	o.o.m.
2	14	3.48	**2.08**	2.24	2.27	3.05	o.o.m.	o.o.m.
4	7	4.79	2.63	2.50	**2.41**	2.46	2.84	o.o.m.
7	4	7.30	3.96	3.55	**3.42**	3.45	3.46	3.62
14	2	13.01	6.95	**6.09**	6.36	6.26	6.32	6.65
28	1	23.69	12.60	**10.85**	11.07	11.36	11.94	12.94

```
1  // initialize variable in constant memory space
2  __constant__ int const_two = 2;
3  // simple elementwise vector multiplication
4  // the number 2 is fetched from constant memory space
5  __global__ void kernel_mul2constant(float *inputA,  // input vector
6                                      const int N)    // number of elements
7  {
8      // compute global index
9      const unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
10
11     // execute multiplication
12     if(i < N)
13         inputA[i] = const_two*inputA[i];
14 }
15 // simple elementwise vector multiplication
16 // the number 2 is directly multiplied
17 __global__ void kernel_mul2(float *inputA,          // input vector
18                             const int N)            // number of elements
19 {
20     // compute global index
21     const unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
22
23     // execute multiplication
24     if(i < N)
25         inputA[i] = 2*inputA[i];
26 }
```

Figure 14: Pseudo C code showing two simple kernels performing an elementwise multiplication of a vector with the number 2. The first kernel fetches the number from constant memory space, the second fetches it from register space.

3.3.3 Memory Types

The use of the different memory types is crucial for writing a fast CUDA kernel. The use of shared memory for computing sums by reduction is very well explained in [19]. In this white paper, different implementations are presented and discussed in regards to execution time and bandwidth. Tests with the different kernel variants were conducted and a implementation without unrolling of loops was decided on. Though, the paper presents better optimized kernels than the one that was adapted for the numerous reductions needed for the evaluation of the objective functions, no increase in performance was found when adapting the best kernel of the paper. For more information on the different kernels and their characteristics see [19].
Due to the heavy use of shared memory, larger shared memory was preferred by setting `cudaDeviceSetCacheConfig()` to `cudaFuncCachePreferShared`. This tells the compiler to minimize L1 cache and maximize shared memory. Another fine tuning option is to set the memory bank size of the shared memory to four byte. Since single precision floating point variables of four bytes size are used, cache misses are reduced. This is done

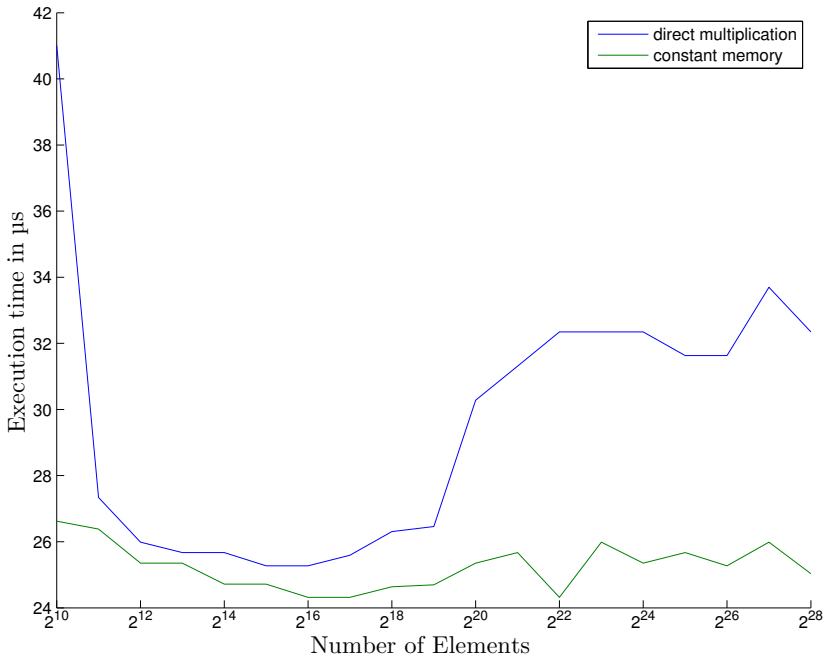

Figure 15: Execution times for elementwise vector multiplication in μs. Using constant memory results in faster execution times.

by setting `cudaDeviceSetSharedMemConfig()` to `cudaSharedMemBankSizeFourByte`. Another improvement was achieved by writing fixed and often read variables into the constant memory space of the GPU. Values that where often read include the reference and template image sizes and domains, the cell sizes h^1, h^2 as well as their reciprocals $1/h^1, 1/h^2$ and tolerances for the evaluation of the stopping rules. Especially the cell sizes and their reciprocals are needed numerous times. To illustrate the impact of constant memory, a little test scenario was constructed. Consider a simple elementwise multiplication of a vector with the number 2. In the first case, each element of the vector is directly multiplied with 2 and in the second case the number is fetched from constant memory space prior to the multiplication, as shown in the pseudo code of Figure 14. Figure 15 displays the execution times in μs for different vector sizes. Using constant memory results in faster execution times compared to the direct method. Unfortunately, the impact on the overall performance is little, due to very small differences of a few microseconds.

The most performance, however, was gained by binding both the template and reference image to texture memory. Pixel values of both images are needed several times for the computation of the function value, gradient and approximation to the Hessian. Through specialized 2D access patterns, the read-out is optimized for texture memory and results

in fewer cache misses than reading from global memory using the same pattern [43]. Further performance is gained by the use of the right address mode. The address modes are summarized in Table 8. Since zero Neumann boundary conditions are used, the address mode is set to `cudaAddressModeBorder` and free boundary handling is gained. This saves additional if-conditions or the use of zero padding as done by Rühaak et. al. [49].

Table 8: Summary of the different address modes for texture memory.

Address mode	Out of bounds index handling
Border	Fetched values are set to zero
Clamp	Fetched values are set to closest boundary
Wrap	Fetched values are interpreted as texture is continuous
Mirror	Fetched values are interpreted as texture is mirrored

3.3.4 Hardware Interpolation

The use of texture memory also enables hardware interpolation. Instead of calculating a bilinear interpolation from given pixel values, an interpolated value is fetched from the texture cache, which improves performance. Linear interpolation can be activated by setting the filter mode of the texture to `cudaFilterModeLinear`. As a reminder, the interpolation of a pixel value at the coordinates (x^1, x^2) can be written as

$$p = (1 - x_r^1)((1 - x_r^2)k_{00} + x_r^2 k_{01}) + x_r^1((1 - x_r^2)k_{10} + x_r^2 k_{11}) \tag{33}$$

where $k_{00} \cdots k_{11}$ are known pixel values and $x_r^i = x^i - \lfloor x^i \rfloor$, $i = 1, 2$ are remainders, as illustrated in Figure 3. The analytical derivative of (33) is defined as

$$\begin{aligned}\frac{\partial p}{\partial x^1} &= (1 - x_r^2)(k_{10} - k_{00}) + x_r^2(k_{11} - k_{01}),\\ \frac{\partial p}{\partial x^2} &= (1 - x_r^1)(k_{01} - k_{00}) + x_r^1(k_{11} - k_{10}).\end{aligned}$$

Using textures, the linear interpolation of a pixel value and its analytical derivative can directly be computed by only five texture fetches

$$\begin{aligned}p &= f(x^1, x^2),\\ \frac{\partial p}{\partial x^1} &= f(1, x^2) - f(0, x^2),\\ \frac{\partial p}{\partial x^2} &= f(x^1, 1) - f(x^1, 0),\end{aligned}$$

where $f(x^1, x^2)$ denotes a texture fetch at the given coordinates. The four texture fetches needed for the derivative computation also benefit from a specialized 2D read-out pattern optimized for texture memory, since they are close together in 2D.

3.3.5 Data Transfer Minimization

Besides the aforementioned techniques, reducing data transfers and storage is one of the most important tasks in order to gain high performing code [13, 41].
The first step was to analyze the registration algorithm and derive formulations that save memory. As shown in Section 2.2, the derived calculation rules are very memory efficient as they compute the relevant values on the fly. The data is limited to the reference and template images, the transformation parameters and some additional constants like cell size or image domains. No matrices or temporal data needs to be stored, thus minimizing the overall memory requirements.
The second step was to reduce the device-host-communication to a minimum. Using PCIe 3.0 (Peripheral Component Interconnect Express) to connect the GPU to the host allows for a theoretical bandwidth of $\approx$ 16 GB/s for host-device transfers. The theoretical bandwidth of device-device transfers is $\approx$ 224 GB/s making it 14 times faster. Intensive bandwidth tests with tools provided by NVIDIA showed a practical peak bandwidth rate of 6.659 GB/s for host-device transfers and 175.781 GB/s for device-device transfers making it over 26 times faster. An overview of the theoretical and practical bandwidths can be found in Table 9.
Therefore, the whole registration algorithm was implemented using CUDA, even parts that got little to no speedup. Only the template and reference images, as well as the transformation parameters and some additional small data is copied to the GPU. The registration then runs completely independent of the host and leaves the CPU open for further computations. When the algorithm is done, the final transformation parameters are copied back to the host.
For the Gauss-Newton optimization, many kernels are needed for operations other than evaluating the objective function, like solving the Quasi-Newton system or calculating L^2-norms for the stopping criteria. This requires the storage and transfer of additional data. Unfortunately, the algorithm leaves little room to make use of asynchronous memory transfers in order to improve performance.
Every step of the Gauss-Newton optimization, relies on the results of the prior step. Therefore, these results need to be computed and transfered in total in order to start

Table 9: Overview of the bandwidth rates for different memory transfers. The theoretical peak bandwidths are hardware specified. The practical peak bandwidths were determined with NVIDIA tools.

transfer direction	theoretical peak bandwidth	practical peak bandwidth
host-device	16 GB/s	6.659 GB/s
device-host	16 GB/s	6.659 GB/s
device-device	224 GB/s	175.781 GB/s

the next step.
Only the evaluation of the stopping rules could be handled parallel to some other computation, but the computation time is so little, that it is irrelevant for the entire algorithm. Thus, only synchronous memory transfers are performed.

3.3.6 Single Precision Accuracy

The downside of using texture memory is the limitation to single precision. Fetching data from a texture, returns a single precision value, impacting the overall accuracy of the algorithm.
In all conducted tests, the approximation to the Hessian had a condition number greater than 100 and was therefore ill-conditioned. A small error in the gradient falsified the calculation of the descent direction when solving the Quasi-Newton system $H \cdot dw = -\nabla D$ and $v_{\text{descent}} = \nabla D \cdot dw$.
Further, the many summations of one hundred thousand and more elements introduces a probable error of order $n^{3/2}10^{-t}x$ [36], where n is the number of addends, t is the number of places in the mantissa of the machine accuracy and x is the magnitude of the addends. Though the error is small, it is non-negligible considering the ill-conditioned Hessian.

3.4 Summary

In this chapter, an introduction to CUDA was given and the most important techniques of implementing the GPU code were explained. First, the reasons for the choice of a GPU implementation using CUDA were discussed. Hereafter, the programming model and memory layout were summarized and an illustrative example showing a basic CUDA program was given.
In the second part of the chapter, the CUDA implementation of the registration algorithm was explained in detail. The most important techniques, including kernel invocation, the use of different memory types and hardware interpolation, are exposed. Extensive tests of the right kernel invocation showed, that a setup consisting of two kernels, each computing one half of the solution, offers the most performance.
In the next chapter, the CUDA implementation is tested in many scenarios. The results are compared to other implementations with a focus on speedup compared to optimized and parallelized CPU code.

Chapter 4: Results and Discussion

The aim of this thesis was to write a very fast executing GPU implementation of an established registration algorithm. Not only should the computation time be short, but the CUDA code should also outperform optimized and parallel CPU code in order to prove that GPU computing is indeed an interesting choice in medical image registration. Therefore, the performance of the GPU code is measured by conducting numerous experiments, which are presented and discussed in this chapter. In order to classify the performance within the context of execution time, the results are compared to four other implementations. All tests were conducted on hardware as described in Section 3.2 and Table 2.

Different abbreviations are used to identify the different implementations of the image registration algorithm. Here, *FAIR* stands for MATLAB code written with the aid of the image registration toolbox FAIR [37]. FAIR also provides the option of using C++ code to accelerate the computation. These files are part of the toolbox and can be used through MATLAB's MEX-interface [15]. This method is identified by *FAIR-MEX*. FAIR is mainly used for research and educational purposes and is optimized in terms of usability and flexibility rather than performance. Nevertheless, for a demonstration of how much research code can be sped up by writing specialized production code, the FAIR runtimes are listed as well. The magnitude of speedup compared to FAIR code, however, should not be overrated by the reader.

Rühaak et al. wrote optimized C++ versions for the evaluation of the objective function, referenced by *C++*, with the option to enable the use of OpenMP in order to speed up the calculation, referred to as *OpenMP*. More details on their implementation of the algorithm using the NGF distance measure can be found in [49]. C++ code for the multilevel generation with the optional use of OpenMP was written for this thesis. Due to the fact that the multilevel generation and the evaluation of the objective function are the computationally most demanding operations by far, the Gauss-Newton optimization was not implemented using C++ and functions from the FAIR toolbox were used for this purposes. This adds a little overhead to the overall runtime of the C++ and OpenMP code.

Two different approaches were followed for the CUDA implementation. The first idea is analog to the handling of the optimized C++ code. We only ran the computational demanding evaluation of the objective function on the GPU and copy back the results for each iteration. The rest of the registration algorithm is managed in MATLAB. This method is identified by *CUDA-Obj*. The second approach was to run the whole algorithm on the GPU and thereby reduce the memory traffic. This required extra CUDA code in order to run the Gauss-Newton optimization on the GPU, which was also written as part of this thesis. This method is named *CUDA*.

For both, the SSD and NGF distance measure, three different tests were conducted. First, the performance of the evaluation of the objective function is measured outside an image registration context. This is the computational most demanding task and the most work was put into optimizing and parallelizing its computation. Secondly, the performance of the affine linear registration algorithm without the multilevel approach was measured. The percentage of time spent evaluating the objective function is very high compared to the percentage of time spent for memory transfers or initializations, which is advantageous for a GPU computation. Thirdly, the performance of the multilevel registration was measured as this is the preferred algorithm in many applications. Here, other operations such as generating the multilevel data and memory transfers have a much larger influence on the overall runtime. Since a GPU is optimized for massively parallel computations and not for memory transfers it is much harder to compete with optimized OpenMP code.
This chapter is organized as follows. A general overview of the image data used in the experiments is given in in Section 4.1. The multilevel implementation is compared to MATLAB code based on the FAIR framework and optimized C++ code with and without OpenMP in Section 4.2. The detailed performance analysis of the GPU implementation of the registration algorithm using both the SSD and the NGF distance measure can be found in Sections 4.3 and 4.4, respectively. Finally, this chapter ends with a short statement on the accuracy of the GPU code in Section 4.5. The general quality of the algorithms is shown in [37] and will not be discussed in this thesis.

4.1 Experimental Data

To conduct the experiments, five different registration scenarios were chosen. There are three monomodal and two multimodal settings, as shown in Figure 16.
A relatively simple image registration two brain slices, referenced as *HNSP*, as shown in the first row of Figure 16. They were acquired for histological serial sectioning for the Human NeuroScanning Project (HNSP) and are courtesy of Oliver Schmitt, Institute of Anatomy, University of Rostock, Germany [51]. The shape of the brain stays mostly intact and primarily rotations and translations occur. A more complicated test scenario is the registration of two hand images, referenced as *HANDS*. They are courtesy of Stefan Heldmann, Institute of Mathematics and Image Computing, University of Lübeck, Germany, and Yali Amit [2]. Here, the template image needs to be rotated as well as scaled and sheared, as seen in the second row of Figure 16. Additionally, the solution to this problem is hard to find, due to local minima. These occur when the template image is rotated and the fingers of the images overlap. Similar to the HNSP scenario is the registration of two histological tissue slices [38], referenced as *CELLS*, as presented in the third row of Figure 16.

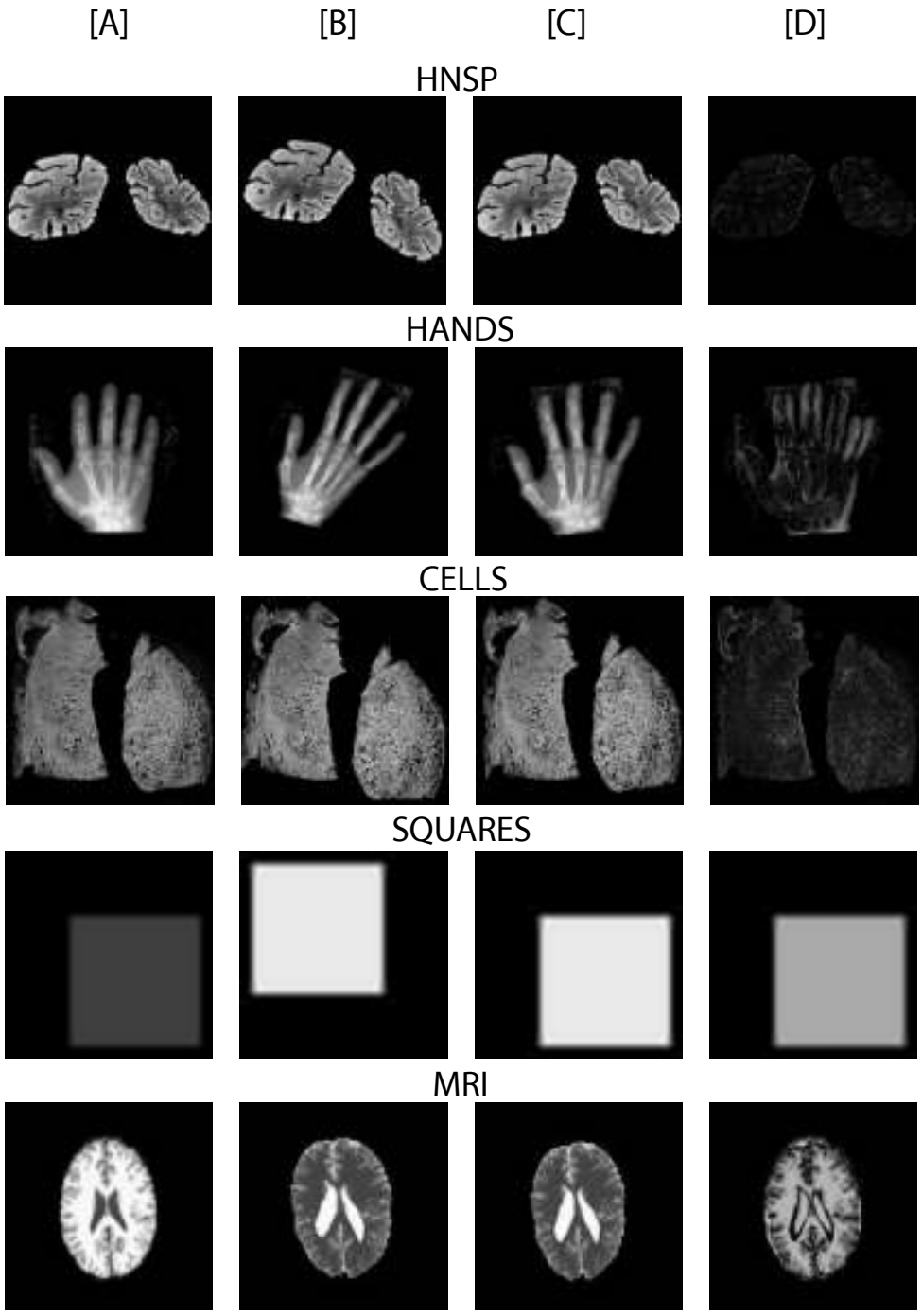

Figure 16: Overview of the five different image pairs for the different test scenarios. Column [A]: reference image R, column [B]: template image T, column [C]: final transformed image $T(y_w)$, column [D] final difference: $|R-T(y_w)|$. Generated with the FAIR toolbox

A simple multimodal test scenario is the registration of two squares with different intensities, referenced as *SQUARES*, fourth row of Figure 16. The template image was generated by translating the reference image and increasing the intensity. Lastly, a clinical more relevant test case is the registration of two MRI images, referenced as *MRI*, of the same brain with T1 and T2 weighting, as shown in last row Figure 16. They are courtesy of BrainWeb [6]. These images have many local differences and are therefore hard to align using affine linear transformations.
The columns [C] and [D] of Figure 16 show the final transformed template image and final difference to the reference image as generated with the FAIR toolbox. The results of the other implementations bear no visual difference to these results.

Image Attributes Since the images are stored in the texture memory space, the image data cannot be double precision floating point and single precision floats have to be used instead. The images must be quadratic and can be of any size when using the registration without the multilevel representations. The multilevel ansatz (12) that was implemented works only for quadratic images of size $2^n \times 2^n$ pixels, where $n \in \mathbb{N}^+$. Thus, the images must be of this size when using the multilevel approach. Implementing another multilevel method that can work with image sizes other than a power of two is a task for future work. With this new method, the multilevel image registration should also work with quadratic images of any size.
Though, many functions of the code are optimized to work with data sizes that are a power of two, using different sized images had only little to no impact on the overall runtime, as shown in Table 10. We registered the HNSP images as shown in images [A] and [B] of the first row of Figure 16. The algorithm computed a solution after 133 iterations. The differences in the runtime are just a few milliseconds and can also arise due to random influences from the system. It is not possible to single out the GPU for computation purposes only. Every running system process that needs the GPU in any way impacts the computations and runtimes vary in the scope of milliseconds.

Table 10: Runtime for the registration of two brain images using affine transformations and the SSD distance measure without multilevel. The registration algorithm terminated after 133 iterations. Image sizes other than a power of two have only little to no impact on the overall runtime.

Image size in pixel	Runtime of CUDA algorithm
510×510	0.1199 s
512×512	0.1200 s
514×514	0.1212 s

4.2 Multilevel Data Generation

As one of the more time consuming components of the registration algorithm, the multilevel data generation was parallelized. The runtime of different implementations can be found in Table 11. For this experiment, multilevel data down to the resolution of 8×8 pixels was generated. The runtimes are averaged over 20 runs.
As expected, the FAIR code runs slowest while the CUDA code runs fastest. Most noticeable is the different runtime increase for the OpenMP and CUDA version. The number of pixels quadruples for each image, whereas the runtime for the CUDA methods increases by 3.33, 3.45 and 3.86, respectively, and for the OpenMP method by 2.05, 3.02 and 2.83, respectively.
This difference can be explained through more memory management on the GPU for the greater images. Each increase in image size results in one additional level of data that needs to be generated and stored. Furthermore, the initialization of OpenMP takes some time as well. The influence of this overhead decreases as more computations are made. This leads to a speedup of 3 for the CUDA version compared to the optimized OpenMP version for small images that reduces to 1.5 for large images. It is likely, that the OpenMP implementation outperforms the CUDA implementation for images greater than 4096×4096 or when using more than four cores.
We could not test this hypothesis due to hardware limitations of the workstation.

Table 11: Runtimes and speedup factors for the generation of multilevel data for different image sizes.

Multilevel Data Generation			
Image size in Pixel	Version	Runtime in s	CUDA speedup
512×512	FAIR	0.0892	129.8
	FAIR-MEX	0.0245	35.6
	C++	0.0032	4.6
	OMP	0.0022	3.1
	CUDA	0.0006	1.0
1024×1024	FAIR	0.3172	158.8
	FAIR-MEX	0.0649	32.5
	C++	0.0080	4.0
	OMP	0.0045	2.3
	CUDA	0.0020	1.0
2048×2048	FAIR	1.7598	256.7
	FAIR-MEX	0.3022	44.1
	C++	0.0273	4.0
	OMP	0.0136	2.0
	CUDA	0.0069	1.0
4096×4096	FAIR	6.6850	266.8
	FAIR-MEX	1.3087	51.0
	C++	0.1133	4.4
	OMP	0.0386	1.5
	CUDA	0.0257	1.0

4.3 Image Registration using SSD

In this section, the test results of the evaluation of the objective function and the affine linear registration routine with and without multilevel data using the SSD distance measure are presented and discussed.

4.3.1 Objective Function Evaluation

The main work of this thesis was to optimize the evaluation of the objective function as its computation is the most demanding task of the registration algorithm. Therefore, different implementations to evaluate the objective function using the SSD distance measure are analyzed. The time it takes to transform a template image with given transformation parameters, interpolate the new pixel values and compute the distance D between the transformed template and the reference image as well as the gradient ∇D and the approximation to the Hessian H was measured for the HNSP images. Tests for the other images are not presented as the results were identical. Table 13 summarizes the execution times for all implementations and different image sizes. The timings are averaged over ten test runs to minimize outliers. Since the C++ and CUDA implementations are managed through MATLAB, a differentiation between the full routine, including the MATLAB and memory transfer overhead (column **With overhead**) and the sole computation times (column **Without overhead**) was made. The CUDA implementation will be referenced as *evalObjSSD*-kernels.
The timings show that the CUDA version is faster than all other versions, yet the speedup compared to the optimized OpenMP code including the overhead is less than 2. For large images, the speedup of the CUDA version compared to the OpenMP version for the sole computation is 3.5.
The table reveals the big impact of the overhead on the overall execution time. The evaluation of the objective function requires only a few kernel launches, yet the data transfer is the same as for the whole registration algorithm. In both cases, the template and reference images and the transformation parameters need to be copied onto the GPU. Therefore, the memory transfer and CUDA initialization account for over half the runtime of the CUDA implementation. The overhead is also noticeable in the OpenMP implementation as it makes up roughly 25 % of the runtime. Here, the overhead is composed of C++ and OpenMP initializations and MATLAB and MEX management.

Analysis with the NVIDIA Visual Profiler [8] revealed that the CUDA implementation achieved an average branch efficiency of 100 %, shared memory efficiency of 100 %, multiprocessor activity of 99.9 % and texture cache hit rate of 81.4 %. A description of these values can be found in Table 12. Additionally, the texture cache throughput for images of size 512×512 pixels was 102.4 GB/s and increased to 127.6 GB/s for images of size 4096×4096 pixels. These rates are lower than the theoretical peak bandwidth

of 175.8 GB/s and could be raised by further specialize the 2D indexing and reduce the texture cache misses. The increased bandwidth would also further reduce the runtime.

Table 12: Description of different kernel metrics [8].

Metric name	Description
Branch efficiency	Ratio of non-divergent branches to total branches expressed as percentage
Shared memory efficiency	Ratio of requested shared memory throughput to required shared memory throughput expressed as percentage
multiprocessor activity	The percentage of time at least one warp is active on a multiprocessor averaged over all multiprocessors on the GPU
Texture cache hit rate	Texture cache hit rate in percent
Texture cache throughput	Texture memory throughput in GB/s

Table 13: Runtimes and speedup factors for the evaluation of the objective function using affine transformations and SSD. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$. The columns **With overhead** include the MATLAB and memory transfer overhead in their timings, whereas the columns **Without overhead** show the sole computation times of the C++, OMP and CUDA implementations.

SSD: Objective Function Evaluation					
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		With overhead		**Without overhead**	
512×512	FAIR	0.136601	35.5	-	-
	FAIR-MEX	0.071462	18.6	-	-
	C++	0.011575	3.0	0.010519	6.7
	OMP	0.004762	1.2	0.003577	2.3
	CUDA	0.003846	1.0	0.001576	1.0
1024×1024	FAIR	0.689126	78.6	-	-
	FAIR-MEX	0.365954	41.8	-	-
	C++	0.047361	5.4	0.045114	12.7
	OMP	0.014161	1.6	0.011762	3.3
	CUDA	0.008763	1.0	0.003557	1.0
2048×2048	FAIR	2.786645	99.0	-	-
	FAIR-MEX	1.496077	53.2	-	-
	C++	0.155262	5.5	0.147031	12.2
	OMP	0.049627	1.8	0.041267	3.4
	CUDA	0.028145	1.0	0.012058	1.0
4096×4096	FAIR	11.557897	103.5	-	-
	FAIR-MEX	5.718299	51.2	-	-
	C++	0.666001	6.0	0.626058	13.7
	OMP	0.201267	1.8	0.158932	3.5
	CUDA	0.111669	1.0	0.045667	1.0

4.3.2 Affine Linear Registration without Multilevel

The algorithm was tested without the multilevel approach and registered the HNSP, HANDS and CELLS images. The SSD works best with monomodal images and thus the multimodal images were not registered. As a safeguard, a maximum of 200 iterations was allowed. All registrations reached a solution without hitting the iteration limit. The HNSP images needed about 140 iterations, the HANDS images required roughly 180 iterations and the registration of the CELLS images terminated after about 60 iterations. The exact number of iterations changed based on the image resolution. To minimize outliers, the timings are averaged over ten test runs. An overview of the runtimes for all implementations can be found in Table 15.
The CUDA implementation outperforms every other method and the performance increases with increasing image size. For smaller images, the speedup of the CUDA version compared to the OpenMP version is at least 6, while for larger images the speedup increases to 16. This was expected, since increasing image size relates to more computational work and the overhead of transferring data to and from the GPU diminishes.

The registration of the images using the CUDA-Obj method results in a speedup of 1.5 to 2 compared to the OpenMP version, which correlates to the findings in Table 13. The overhead of data transfer in every iteration is too large compared to the fast computations resulting in a low speedup compared to the OpenMP code. Implementing the whole registration algorithm using CUDA minimizes the host-device communication and most of the memory transfers are made on the device with much higher bandwidth. Table 14 shows the ratio of computation and memory transfers for the registration of the HNSP images. In the case of the 4096×4096 pixels image, 99.3 % of the execution time are spent for evaluating the objective function. Since the time spent executing kernels is very high, optimizations of the memory handling will not improve the overall runtime by much.
The registration of the CELLS images shows that specialized and optimized production code is enormously faster than research code. A speedup of the CUDA implementation compared to the MATLAB implementation of over 800 was reached for the largest image. The execution time was reduced from more than 18 minutes to 1.4 seconds.

Table 14: Relation of computation and memory transfers for the registration of the HNSP images without multilevel.

Image size in pixel	Computation	Memory transfers
512×512	83.95 %	16.05 %
4096×4096	99.40 %	0.6 %

Table 15: Runtimes and speedup factors for the registration of different monomodal scenarios as shown in the first three rows of Figure 16 using affine transformations and SSD without multilevel. The HNSP images needed about 140 iterations, the HANDS images required roughly 180 iterations and the registration of the CELLS images terminated after about 60 iterations. The exact number of iterations changed based on the image resolution. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

SSD: Affine Linear Image Registration without Multilevel							
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		HNSP		**HANDS**		**CELLS**	
512×512	FAIR	20.88	163.4	17.86	168.8	12.97	216.1
	FAIR-MEX	10.06	78.7	7.73	73.1	4.94	82.3
	C++	2.37	18.6	1.92	18.1	1.22	20.3
	OMP	0.82	6.4	0.68	6.5	0.76	12.6
	CUDA-Obj	0.81	6.4	0.78	7.4	0.46	7.6
	CUDA	0.13	1.0	0.11	1.0	0.06	1.0
1024×1024	FAIR	96.96	395.4	110.84	444.7	76.00	538.3
	FAIR-MEX	44.44	181.2	46.72	187.5	29.44	208.5
	C++	8.54	34.8	8.97	36.0	4.95	35.1
	OMP	2.56	10.4	2.71	10.9	1.93	13.6
	CUDA-Obj	1.77	7.2	2.29	9.2	1.26	8.9
	CUDA	0.25	1.0	0.25	1.0	0.14	1.0
2048×2048	FAIR	393.60	522.0	490.12	587.8	297.03	749.4
	FAIR-MEX	183.77	243.7	211.78	254.0	117.89	297.4
	C++	33.57	44.5	39.42	47.3	19.68	49.7
	OMP	9.40	12.5	10.99	13.2	6.23	15.7
	CUDA-Obj	5.14	6.8	7.74	9.3	3.87	9.8
	CUDA	0.75	1.0	0.83	1.0	0.40	1.0
4096×4096	FAIR	1669.71	596.5	2158.08	673.2	1118.81	806.6
	FAIR-MEX	726.64	259.6	877.77	273.8	414.02	298.5
	C++	138.17	49.4	164.16	51.2	70.66	50.9
	OMP	38.66	13.8	46.71	14.6	22.74	16.4
	CUDA-Obj	20.01	7.2	31.62	9.9	13.76	9.9
	CUDA	2.80	1.0	3.21	1.0	1.39	1.0

4.3.3 Affine Linear Registration with Multilevel

The multilevel routine was tested similar to the routine without multilevel. Experiments were run with the HNSP, HANDS and CELLS images and as a safeguard a maximum of 10 iterations per optimization level was set. If these were reached at all, it was on the coarse levels. Computations on the finest level usually just required one correction step to satisfy the stopping criteria. Thus, the main part of the work is done on the coarse levels which are computationally less intensive than the fine levels. To minimize outliers, the timings are averaged over 10 runs. An overview of the runtimes for all implementations can be found in Table 17.
Analog to the former test case, the performance of the CUDA code improves with the image size. Starting with a speedup of 6.5 of the CUDA implementation compared to the OpenMP implementation for small images, a speedup of over 11 is reached for the largest images.
Due to the generation and transfer of the multilevel data the influence of memory transfers is a lot higher than in the routine without multilevel. Additionally, the kernels have little work to do on the coarser levels, which is something GPU programmers usually try to avoid. Therefore, the ratio of computation and memory transfers is nearly even, as indicated in Table 16. For images of size 512×512 pixels, the kernels to compute the function value and derivatives have little workload and other factors, such as increased memory transfers, come into play, as illustrated in Figure 17. Most of the work is done on a coarse level resulting in a decreased averaged multiprocessor activity of 45.8 % for the evaluation of the objective function, leaving multiprocessors idle.
Further, only 35 % of the overall runtime is spent in these kernels, while 20 % are needed for other computations of the Gauss-Newton optimization. These kernels are not as optimized as the kernels to evaluate the objective function and have a generally low workload. Examples are the computation of the norm of the gradient $\|\nabla D_{\text{SSD}}\|$ or descent direction $v_{\text{descent}} = \nabla D \cdot dw$, where $\nabla D \in \mathbb{R}^{1\times 6}$ and $dw \in \mathbb{R}^6$. These operations do not fully utilize the capabilities of the GPU.
Another 29.5 % of the time is needed for copying data on the device. These copies are mostly in the scope of a few kilobytes and the average throughput is only 413 MB/s. This is far from the practical peak performance and throughput could be increased by

Table 16: Ratio of calculations and memory transfers for the multilevel registration of the HNSP images using the SSD distance measure.

Image size in pixel	Computation	Memory transfers
512×512	55.7 %	44.3 %
4096×4096	61.4 %	38.6 %

joining several small copies into one single larger copy. The texture cache throughput decreased to an average of 18.5 GB/s compared to the single level registration. Again, this is not close to the theoretical peak performance of 175.8 GB/s and is a result of the low workload on the coarse levels.
For images of size 4096×4096 pixels the computational workload is higher, as seen in Figure 18. Here, roughly 59 % of the time are used for computing the function value and derivatives. The fact that additional work is done on higher optimization levels raises the average multiprocessor activity to 57.6 %. Also, the influence of the little device-device memory transfers diminishes as another 35 % of the runtime are needed to transfer the reference and template image to the GPU. This transfer achieves a bandwidth of 5.811 GB/s, which is close to the practical peak bandwidth. The texture cache throughput increased to an average of 33.7 GB/ due computations on higher levels. We already discussed the improvement of the kernels that evaluate the objective function and since the host-device transfer bandwidth is nearly optimal, only the remaining 5 % of the code's runtime can be further accelerated.
Another interesting aspect is the runtime of the multilevel registration for images of size 512×512 pixels. The registration for the HNSP images was done in 18 ms while the other two registration problems took only a few milliseconds longer. These very fast execution times open the possibility to use this code in live settings where 15 or more images need to be processed within a second, e.g. image guided surgery [45] or live tracking [24].

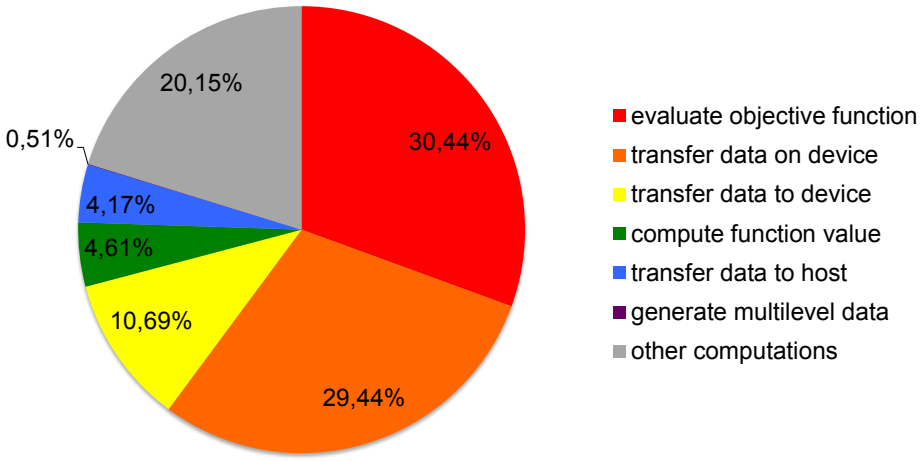

Figure 17: Runtime breakdown of the multilevel registration of the HNSP images using SSD and image size 512×512. **Evaluate objective function** is the time spent in the evalObjSSD-kernels and **compute function value** is the time spent in a kernel that solely computes the function value for the Armijo line-search.

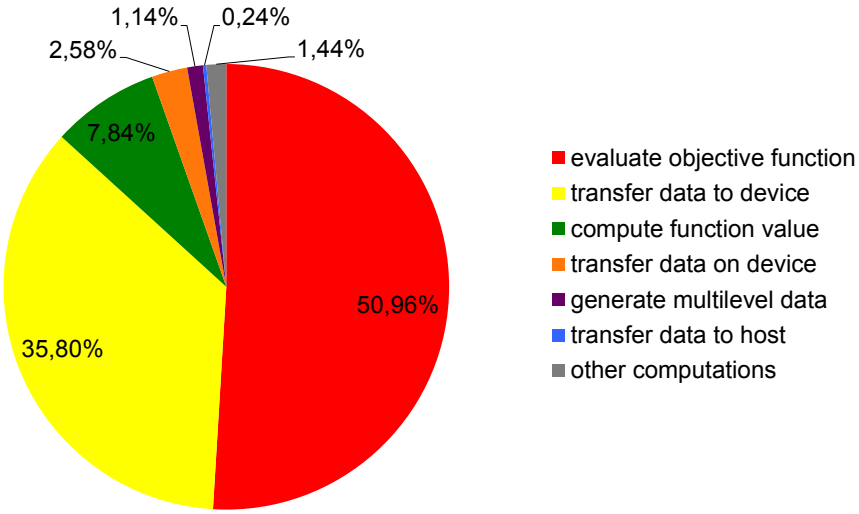

Figure 18: Runtime breakdown of the multilevel registration of the HNSP images using SSD and image size 4096×4096. **Evaluate objective function** is the time spent in the evalObjSSD-kernels and **compute function value** is the time spent in a kernel that solely computes the function value for the Armijo line-search.

Table 17: Runtimes and speedup factors for the multilevel registration of different monomodal scenarios as shown in the first three rows of Figure 16 using affine transformations and SSD. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

SSD: Affine Linear Image Registration with Multilevel							
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		HNSP		**HANDS**		**CELLS**	
512×512	FAIR	0.845	46.2	0.962	37.9	0.962	46.8
	FAIR-MEX	0.459	25.1	0.558	22.0	0.506	24.6
	C++	0.168	9.2	0.187	7.4	0.181	8.8
	OMP	0.147	8.0	0.165	6.5	0.150	7.3
	CUDA-Obj	0.172	9.4	0.188	7.4	0.196	9.5
	CUDA	0.018	1.0	0.025	1.0	0.021	1.0
1024×1024	FAIR	3.378	121.7	4.108	120.9	4.151	113.7
	FAIR-MEX	1.509	54.4	1.886	55.5	1.793	49.1
	C++	0.383	13.8	0.424	12.5	0.391	10.7
	OMP	0.231	8.3	0.310	9.1	0.230	6.3
	CUDA-Obj	0.255	9.2	0.313	9.2	0.269	7.4
	CUDA	0.028	1.0	0.034	1.0	0.037	1.0
2048×2048	FAIR	13.728	239.0	15.948	236.8	17.195	277.2
	FAIR-MEX	6.025	104.9	7.058	104.8	6.799	109.6
	C++	1.220	21.2	1.372	20.4	1.267	20.4
	OMP	0.537	9.3	0.800	11.9	0.561	9.0
	CUDA-Obj	0.494	8.6	0.563	8.4	0.513	8.3
	CUDA	0.057	1.0	0.067	1.0	0.062	1.0
4096×4096	FAIR	56.359	302.5	64.609	335.0	66.087	356.0
	FAIR-MEX	23.389	125.5	25.673	133.1	25.335	136.5
	C++	4.667	25.1	5.287	27.4	4.595	24.8
	OMP	1.660	8.9	2.190	11.4	1.656	8.9
	CUDA-Obj	1.395	7.5	1.627	8.4	1.449	7.8
	CUDA	0.186	1.0	0.193	1.0	0.186	1.0

4.4 Image Registration using NGF

Similar to the preceding section, in this section, the test results of the evaluation of the objective function and the affine linear registration routine with and without multilevel data, all using the NGF distance measure, are presented and discussed. The tests were performed with all images of Figure 16 and different image sizes in order to evaluate the behavior of the routines thoroughly. In general, the results and conclusions for the different experiments using the NGF distance measure are analog to the results and conclusions for the SSD distance measure. Hence, the following analysis is similar to the preceding.

4.4.1 Objective Function Evaluation

Again, the implementations of the different methods to evaluate the objective function and calculate the function value D, the gradient ∇D and the approximation to the Hessian H using the NGF distance measure will be analyzed initially. The CUDA implementation will be referenced as *evalObjNGF*-kernels. An overview of the runtime for all implementations with different image resolutions is given in Table 18. The timings between the complete routine including MATLAB and memory transfer overhead (column **With overhead**) and the sole execution time of the C++, OpenMP and CUDA functions (column **Without overhead**) are differentiated.
The CUDA code outperforms every other code and a speedup of 3 to 3.8 is achieved for the complete routine, while the sole computation is over 5 times faster than the competing OpenMP code. The influence of the overhead is lower than for the SSD distance measure since the NGF distance measure is computationally more demanding and more time is spent to compute the values. While the memory transfers and CUDA initializations as well as MATLAB administrations account for roughly 40 % of the CUDA runtime, only less than 10 % of the OpenMP runtime come from the overhead.
Similar to the SSD kernels, the evalObjNGF-kernels achieved an average branch efficiency of 100 %, shared memory efficiency of 100 %, multiprocessor activity of 99.9 % and texture cache hit rate of 89.6 %. The texture cache throughput for images of size 512×512 was 133.5 GB/s and increased to 153.8 GB/s for images of size 4096×4096, which is relatively close to the theoretical peak bandwidth of 175.8 GB/s. The increase compared to SSD can be explained by additional texture fetches for the NGF computation.

Table 18: Runtimes and speedup factors for the evaluation of the objective function using affine transformations and NGF. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$. The columns **With overhead** include the MATLAB and memory transfer overhead in their timings, whereas the columns **Without overhead** show the sole computation times of the C++, OMP and CUDA implementations.

NGF: Objective Function Evaluation					
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		With overhead		**Without overhead**	
512×512	FAIR	0.232675	49.0	-	-
	FAIR-MEX	0.113178	23.8	-	-
	C++	0.043095	9.1	0.041887	16.1
	OMP	0.014229	3.0	0.013123	5.1
	CUDA	0.004749	1.0	0.002597	1.0
1024×1024	FAIR	1.033982	75.6	-	-
	FAIR-MEX	0.414318	30.3	-	-
	C++	0.168096	12.3	0.165727	19.9
	OMP	0.046569	3.4	0.044127	5.3
	CUDA	0.013681	1.0	0.008333	1.0
2048×2048	FAIR	4.782144	105.6	-	-
	FAIR-MEX	1.899915	42.0	-	-
	C++	0.677042	15.0	0.668580	22.8
	OMP	0.170617	3.8	0.161458	5.5
	CUDA	0.045287	1.0	0.029280	1.0
4096×4096	FAIR	19.096109	105.9	-	-
	FAIR-MEX	7.427478	41.2	-	-
	C++	2.774101	15.4	2.733711	23.9
	OMP	0.673741	3.7	0.631621	5.5
	CUDA	0.180401	1.0	0.114210	1.0

4.4.2 Affine Linear Registration without Multilevel

The results of affine linear registration without multilevel for all test scenarios, as shown in Figure 16, are summarized in Table 20 for the monomodal and in Table 21 for the multimodal images. Since the runtimes were very high, especially for the MATLAB code, the algorithm was terminated after 50 iterations. To minimize outliers, the timings are averaged over ten test runs.
Once more, the CUDA implementation is better than the competing OpenMP code. For the small images, a speedup of the CUDA implementation compared to the OpenMP implementation of at least 7 was achieved which increases to ≈ 11 for the larger images. Analog to the SSD distance measure, the computational workload is desired to be high in order to fully utilize the capabilities of the GPU. Therefore, the speedup grows larger for the larger images.
The speedup of the CUDA-Obj version compared to the OpenMP code ranges from 2 to 3 and is only slightly lower than the speedup measured for the evalObjNGF-kernels. The difference can be explained by additional memory transfers. This leads to the same conclusions as for the SSD registration. The overhead of data transfer in every iteration is too large and host-device communication should be minimized. This was effectively done by implementing the whole registration algorithm using CUDA.
Even though computationally less demanding operations are performed using the GPU and many kernels do not achieve such efficiency as the evalObjNGF-kernels, the algorithm greatly benefits from the very high memory bandwidth on the device. Table 19 shows the ratio of computation and memory transfers for the registration of the HNSP images. In the case of the 4096×4096 pixels image, 98.6 % of the execution time are spent for evaluating the objective function. Again, the time spent executing kernels is dominating, thus optimizations of the memory handling will not have a big impact on the overall runtime.

Table 19: Ratio of calculations and memory transfers for the registration of the HNSP images without multilevel using the NGF distance measure.

Image size in pixel	Computation	Memory transfers
512×512	84.8 %	15.2 %
4096×4096	98.6 %	1.4 %

Table 20: Runtimes and speedup factors for the registration of different monomodal scenarios as shown in the first three rows of Figure 16 using affine transformations and NGF without multilevel. For timing reasons, all registrations were terminated after 50 iterations. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

NGF: Affine Linear Registration without Multilevel Monomodal Images							
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		HNSP		**HANDS**		**CELLS**	
512×512	FAIR	15.05	99.7	13.86	170.8	17.92	143.9
	FAIR-MEX	5.67	37.5	5.38	66.3	6.49	52.1
	C++	4.29	28.4	4.24	52.2	4.19	33.6
	OMP	1.06	7.0	1.02	12.6	1.00	8.1
	CUDA-Obj	0.53	3.5	0.53	6.6	0.51	4.1
	CUDA	0.15	1.0	0.08	1.0	0.12	1.0
1024×1024	FAIR	61.41	175.7	69.30	199.4	81.03	230.5
	FAIR-MEX	24.84	71.1	27.19	78.3	28.90	82.2
	C++	16.33	46.7	16.38	47.2	16.31	46.4
	OMP	3.58	10.2	3.60	10.4	3.67	10.4
	CUDA-Obj	1.45	4.2	1.40	4.0	1.39	4.0
	CUDA	0.35	1.0	0.35	1.0	0.35	1.0
2048×2048	FAIR	306.55	238.4	302.17	233.9	391.26	305.9
	FAIR-MEX	106.16	82.5	108.53	84.0	123.89	96.9
	C++	65.07	50.6	65.01	50.3	67.71	53.0
	OMP	13.83	10.8	14.14	10.9	14.15	11.1
	CUDA-Obj	4.63	3.6	4.71	3.6	4.57	3.6
	CUDA	1.29	1.0	1.29	1.0	1.28	1.0
4096×4096	FAIR	1187.34	234.8	1230.63	244.2	1506.79	300.4
	FAIR-MEX	416.44	82.4	428.96	85.1	490.85	97.9
	C++	259.92	51.4	257.50	51.1	283.27	56.4
	OMP	54.67	10.8	55.02	10.9	58.02	11.6
	CUDA-Obj	18.14	3.6	18.22	3.6	17.99	3.6
	CUDA	5.06	1.0	5.04	1.0	5.02	1.0

Table 21: Runtimes and speedup factors for the registration of different multimodal scenarios as shown in the last two rows of Figure 16 using affine transformations and NGF without multilevel. For timing reasons, all registrations were terminated after 50 iterations. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

NGF: Affine Linear Registration without Multilevel Multimodal Images					
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		SQUARES		**MRI**	
512×512	FAIR	16.10	118.0	15.31	133.2
	FAIR-MEX	7.45	54.6	6.04	52.6
	C++	6.03	44.2	4.23	36.8
	OMP	2.00	14.7	1.02	8.9
	CUDA-Obj	0.67	4.9	0.53	4.6
	CUDA	0.14	1.0	0.11	1.0
1024×1024	FAIR	80.62	187.3	64.59	184.9
	FAIR-MEX	31.28	72.7	26.60	76.1
	C++	24.26	56.4	16.61	47.5
	OMP	6.40	14.9	3.78	10.8
	CUDA-Obj	1.85	4.3	1.41	4.0
	CUDA	0.43	1.0	0.35	1.0
2048×2048	FAIR	340.31	213.2	301.84	234.8
	FAIR-MEX	131.90	82.6	112.18	87.3
	C++	93.41	58.5	65.08	50.6
	OMP	24.13	15.1	13.76	10.7
	CUDA-Obj	5.40	3.4	4.60	3.6
	CUDA	1.60	1.0	1.29	1.0
4096×4096	FAIR	1377.50	220.6	1221.16	242.8
	FAIR-MEX	538.16	86.2	433.14	86.1
	C++	320.52	51.3	256.84	51.1
	OMP	87.56	14.0	54.80	10.9
	CUDA-Obj	20.06	3.2	17.96	3.6
	CUDA	6.24	1.0	5.03	1.0

4.4.3 Affine Linear Registration with Multilevel

The results of the multilevel affine linear registration using the NGF distance measure for all images illustrated in Figure 16 are presented and discussed. As a safeguard, the maximum number of iterations per optimization level was set to ten. These were generally only reached on the coarser levels. On the finer levels, only one correction step was needed, thus minimizing the execution time. The only exception are the CELLS images. For unknown reasons, the behavior was the other way around. On the coarser levels, the registration was done after a few iterations while on the finer levels, the iteration maximum was reached, thereby increasing the runtime. A summary of all results and speedups are found in Table 23 for the monomodal images and in Table 24 for the multimodal images.
Here, too, the timings show the superiority of the CUDA code compared to all other implementations. A speedup of the CUDA version compared to the optimized OpenMP version of at least 7 was measured, while a maximum speedup of 18.8 for the SQUARES images of size 2048×2048 pixels was achieved. For the CELLS images a speedup of 21.1 was measured. Since most of the work is done on the finest levels, the overhead for data transfer diminishes and the kernels work with high multiprocessor activity most of the time.
Similar to the multilevel registration using the SSD distance measure, the influence of memory transfer on the device is a lot higher compared to the registration working only on one level. The ratio of calculations and memory transfer for the multilevel registration of the HNSP images is shown in Table 22. Compared to the single level registration, the portion of memory transfers is a lot higher but not as high as for the SSD distance measure, since the evaluation of the objective function using NGF is computationally more demanding than using SSD.
A more detailed breakdown of the runtime is found in Figure 19 for HNSP images of size 512×512 pixels. Most of the time is spent in the evalObjNGF-kernels, but they do not work as efficiently any more. The low workload on the coarser levels decrease the multiprocessor activity to a minimum of 11 % on the coarsest level and to an average of 47.6 % for complete registration. Figure 19 also states that 20 % of the time is needed for other kernel calculations. These kernels are not as optimized as the evalObjNGF-

Table 22: Ratio of calculations and memory transfers for the multilevel registration of the HNSP images using the NGF distance measure.

Image size in pixel	Computation	Memory transfers
512×512	65.3 %	34.7 %
4096×4096	75.9 %	24.1 %

kernels and have a generally low workload, further negatively impacting the runtime of the multilevel registration. Additionally, almost 25 % of the time is required for device-device memory transfers. Like in the SSD case, many of these transfers are only a few kilobytes yet the average throughput is even lower with only 263 MB/s. The throughput could also be increased if several of the small transfers are joined into one large transfer. The texture cache throughput decreased to 36.7 GB/s compared to the single level registration. Compared to the multilevel registration using SSD the rate doubled. This can be explained by additional texture fetches for computing the NGF function value and derivatives.
For images of size 4096×4096 pixels the runtime distribution is more suiting for GPU computation, as seen in Figure 20. Here, nearly two thirds of the whole runtime are spent in the evalObjNGF-kernels. Combined with the time to compute the function value for the Armijo line-search, almost three quarters of the time is spent in optimized kernels. Also, the average multiprocessor activity is increased slightly by 7 % to a total of 54.4 %. Since more work is done on higher levels, the kernels are executed more often with a high workload. Copying the images and other data to the device requires 22 % of the time. This transfer achieves a bandwidth of 5.835 GB/s, which is close to the practical peak bandwidth. The average texture cache throughput increased to 47.3 GB/s. This leaves only 4 % of the runtime which can be improved by fine tuning of the other kernels and better memory management.
Generally, like the multilevel registration using SSD, this code can benefit of a joint CPU-GPU computation. One idea is to run the code for the coarser levels on the host while copying data for the finer levels to the device. Once the data is on the device, the transformation parameters of the first optimization level can be transfered to the GPU and the remaining optimization is finished using the CUDA code.
The runtime of 28 ms for the multilevel registration of the HNSP images makes the CUDA implementation also an interesting option for real-time applications, where images need to be processed within 60 ms in order to be visually fluent.

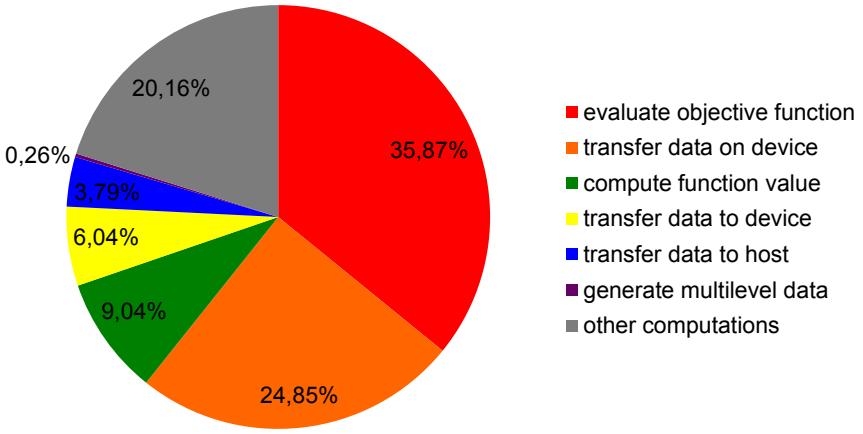

Figure 19: Runtime breakdown of the multilevel registration of the HNSP images using NGF and image size 512×512. **Evaluate objective function** is the time spent in the evalObjSSD-kernels and **compute function value** is the time spent in a kernel that solely computes the function value for the Armijo line-search.

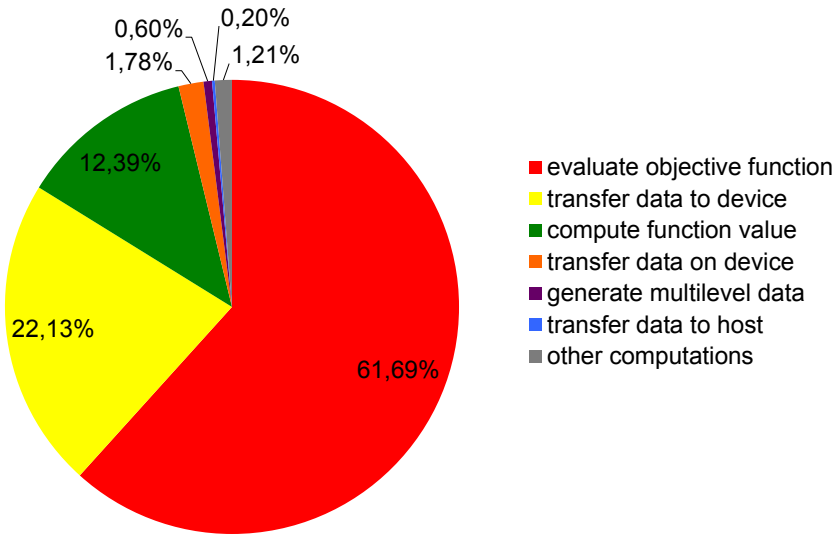

Figure 20: Runtime breakdown of the multilevel registration of the HNSP images using NGF and image size 4096×4096. **Evaluate objective function** is the time spent in the evalObjSSD-kernels and **compute function value** is the time spent in a kernel that solely computes the function value for the Armijo line-search.

Table 23: Runtimes and speedup factors for the multilevel registration of different monomodal scenarios as shown in the first three rows of Figure 16 using affine transformations and NGF. As a safeguard a maximum of 10 iterations per optimization level was allowed. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

NGF: Affine Linear Registration with Multilevel Monomodal Images							
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		HNSP		**HANDS**		**CELLS**	
512×512	FAIR	1.365	48.0	1.583	42.7	6.359	113.7
	FAIR-MEX	0.639	22.5	0.792	21.4	2.764	49.4
	C++	0.448	15.7	0.533	14.4	1.456	26.0
	OMP	0.224	7.9	0.280	7.6	0.529	9.5
	CUDA-Obj	0.256	9.0	0.371	10.0	0.327	5.9
	CUDA	0.028	1.0	0.037	1.0	0.056	1.0
1024×1024	FAIR	5.089	111.1	5.715	119.1	26.465	397.6
	FAIR-MEX	1.784	38.9	1.968	41.0	10.683	160.5
	C++	1.337	29.2	1.662	34.7	5.337	80.2
	OMP	0.439	9.6	0.575	12.0	1.340	20.1
	CUDA-Obj	0.372	8.1	0.491	10.2	0.512	7.7
	CUDA	0.046	1.0	0.048	1.0	0.067	1.0
2048×2048	FAIR	22.666	234.9	24.154	250.0	114.227	522.0
	FAIR-MEX	6.655	69.0	7.386	76.4	44.368	202.7
	C++	4.968	51.5	5.061	52.4	23.609	107.9
	OMP	1.270	13.2	1.338	13.8	4.626	21.1
	CUDA-Obj	0.673	7.0	0.737	7.6	2.331	10.7
	CUDA	0.096	1.0	0.097	1.0	0.219	1.0
4096×4096	FAIR	90.376	308.5	98.052	328.4	451.757	480.9
	FAIR-MEX	25.095	85.7	28.429	95.2	171.169	182.2
	C++	19.473	66.5	19.887	66.6	72.084	76.7
	OMP	4.528	15.5	4.797	16.1	12.691	13.5
	CUDA-Obj	1.973	6.7	2.051	6.9	3.503	3.7
	CUDA	0.293	1.0	0.299	1.0	0.939	1.0

Table 24: Runtimes and speedup factors for the multilevel registration of different multimodal scenarios as shown in the last two rows of Figure 16 using affine transformations and NGF. As a safeguard a maximum of 10 iterations per optimization level was allowed. The timings are averaged over 10 runs. The columns *CUDA speedup* display $\text{runtime}_{\text{method}}/\text{runtime}_{\text{CUDA}}$.

NGF: Affine Linear Registration with Multilevel Multimodal Images					
Image size in Pixel	Version	Runtime in s	CUDA speedup	Runtime in s	CUDA speedup
		SQUARES		**MRI**	
512×512	FAIR	2.623	76.9	1.542	45.4
	FAIR-MEX	1.497	43.9	0.664	19.5
	C++	0.820	24.0	0.485	14.3
	OMP	0.339	9.9	0.241	7.1
	CUDA-Obj	0.343	10.1	0.278	8.2
	CUDA	0.034	1.0	0.034	1.0
1024×1024	FAIR	7.932	110.6	5.466	143.6
	FAIR-MEX	4.176	58.2	1.796	47.2
	C++	2.391	33.3	1.335	35.1
	OMP	0.871	12.1	0.420	11.0
	CUDA-Obj	0.466	6.5	0.308	8.1
	CUDA	0.072	1.0	0.038	1.0
2048×2048	FAIR	33.087	305.2	23.891	262.8
	FAIR-MEX	16.665	153.7	7.448	81.9
	C++	6.704	61.8	5.045	55.5
	OMP	2.042	18.8	1.265	13.9
	CUDA-Obj	0.943	8.7	0.661	7.3
	CUDA	0.108	1.0	0.091	1.0
4096×4096	FAIR	117.161	321.7	96.137	330.9
	FAIR-MEX	59.286	162.8	27.092	93.2
	C++	28.456	78.1	19.533	67.2
	OMP	6.740	18.5	4.767	16.4
	CUDA-Obj	2.979	8.2	2.003	6.9
	CUDA	0.364	1.0	0.291	1.0

4.5 Accuracy

The use of single precision floating point data did not have any significant impact on the computation of a solution to the registration problems. While the approximation to the Hessian is ill-conditioned and had a condition number greater than 100 for all the conducted experiments, the influence of the single precision was not great enough to alter the solution of the Quasi-Newton system in a way, that the registration algorithm wrongly terminated.
Compared to the FAIR version of the code, which runs with double precision, the relative error of the evalObjSSD-kernels compared to the FAIR code was in the range of 10^{-7} for the function value, 10^{-4} for the norm of the gradient and 10^{-5} for the norm of the approximation to the Hessian. Since additional arithmetic operations for the gradient and even more for the approximation to the Hessian computation were needed, the error of these norms is higher than for the function value.
Similar behavior was measured for the evalObjNGF-kernels. Here, the error of the function value is in the range of 10^{-6}, 10^{-2} for the norm of the gradient and 10^{-3} for the norm of the approximation to the Hessian. Compared to the SSD, the errors are worse since more sums have to be computed.
A statement about the absolute final error of the single and multilevel registration is difficult to make, because the true transformation is generally unknown. The use of a robust Gauss-Newton optimization minimized the influences of these errors and no differences in the final transformed template images of the FAIR and CUDA registration were visible to the human eye. If NVIDIA introduces double precision for texture memory, the code can easily be altered to use double precision in order to increase accuracy.

Chapter 5: Conclusion and Outlook

Over the past decade, parallel programming gained more and more importance for writing new code that effectively uses the capabilities of state-of-the-art computers. While CPUs consist of multiple cores, GPUs provide several thousands of threads that can work in parallel. This requires programmers to often rethink their approach on how to solve problems, so that all these possible work units are used. The introduction of CUDA and general-purpose computing on GPUs made it possible to write GPU code without great effort. Yet, in order to unleash the full potential of GPGPU, algorithms need to be restructured and all the hardware features must be utilized.
This work presented a CUDA implementation of an affine linear multilevel registration algorithm using the SSD and NGF distance measures in order to reduce the computation time. To our knowledge, the CUDA implementation of a registration algorithm using the Normalized Gradient Fields distance measure is a novelty.
We identified the computationally most demanding operations and optimized them to be calculated on a GPU. The by far most expensive operation is the computation of the distance measure function value and its derivatives. Following the approach of Rühaak et al. [49], we extensively analyzed the mathematical foundation of the distance measure computation. A straight-forward matrix-based approach, as used in FAIR, is not optimized for parallel execution. A complete reconstruction of the calculation was needed. Hence, the internal structure of the matrices was exposed and exploited. Since these matrices are only sparsely filled, the interdependencies of the single entries were dissected and pixelwise independent, explicit calculation rules were derived.
This enabled the computation of the function value and derivatives for both distance measures on the fly without storing any temporal data. Secondly, other operations, e.g. the multilevel generation or a solver for the Quasi-Newton system, were restructured as well in order to be executed in parallel on the GPU. Without these preliminary considerations it would not have been possible to effectively parallelize the algorithm.
After the algorithm was parallelized, it was implemented using NVIDIAs CUDA framework. CUDA enables the use of high and low level features of the GPU and only by combining the use of these features, high performing code was gained. We started by analyzing the kernel invocation with different setups. Kernels can be launched with a user-defined grid and thread block layout and general statements on what layout is best can not be made [63]. For the kernels which compute the function value D, the gradient ∇D and the approximation to the Hessian H of the distance measures, now called *evalObj-kernels*, the best setup consisted of two kernels that separately computed one half of the solution. This is contrary to common practice, as kernel calls induce an overhead, which programmers usually avoid [5]. Through the rearrangement of the kernels, the time to evaluate the objective function was decreased by up to 14 %.

Storing different data in different memory spaces was another important step in writing the code. The GPU offers various memory spaces of different size and with different access times. Hence, we stored the template and reference images in the texture memory space and used shared memory to compute sums by parallel reduction. Shared memory can be accessed very fast but it is only visible to threads within a block and limited to 48 kB. It is not possible to write very large arrays completely into the shared memory space and kernels were called recursively to compute sums efficiently. The texture memory is optimized for 2D/3D spatial read-out patterns and features the use of linear hardware interpolation and boundary handling. All three features came in handy, since the registration algorithm works with 2D images, linear interpolation and zero Dirichlet and Neumann boundary conditions.
Besides using different memory types, the memory transfers between the GPU and the CPU should be kept at a minimum. By implementing the whole algorithm via CUDA, instead of just the distance measure computation, we ensured minimal data transfer as only the images and transformation parameters need to be copied to the device once. Though we reduced the host-device communication to a minimum, the device-device data transfers leave room for optimizations. A lot of the transfers are in the scope of a few kilobytes or even bytes and the bandwidth is far from close to the practical peak bandwidth.
Using NVIDIAs Visual Profiler [8] we analyzed the characteristics of the evalObj-kernels. They achieve very high branch and shared memory efficiency as well as multiprocessor activity. The texture cache hit rate is at least 79.4 % and the throughput comes close to the theoretical peak bandwidth when performing a single level image registration using NGF. Both could be improved by further specializing the 2D read-out patterns. Other kernels used in the Gauss-Newton optimization are underperforming since they have low workload and are not as optimized as the evalObj-kernels.
To test our CUDA implementation, we ran experiments with several image pairs, three monomodal and two multimodal. These experiments showed, that our CUDA code of the affine linear multilevel registration using SSD and NGF outperforms even optimized and parallelized C++ code. A speedup of the CUDA code compared to OpenMP code of up to 18.8 was gained when performing a multilevel registration using NGF.
In general, the performance of the CUDA implementation was best when images of size 4096×4096 pixels were used. Large images ensured high kernel workload and minimized the percentage of memory transfers compared to computations.
The multilevel registration of two 512×512 pixel images was completed in only 18 ms for the SSD and 28 ms for the NGF distance measure. These very fast execution times enable the use of this code in real-time settings, where images need to be processed in a fixed amount of time. One possible field of use could be image registration based ultrasound tracking as proposed by König et al. [24].

As part of future work, we will further optimize the algorithm. As a first idea, we will use the idle CPU for computations on coarse levels. The time is takes to transfer the reference and template images to the GPU can be used to begin the multilevel registration process on the coarsest level using the CPU. Once the image transfer is done, the current transformation parameters can be transfered as well and the algorithm can start with a finer level on the device. We will also extend the CUDA code to work with 3D images enabling the use of this code in more sophisticated applications. High performance is expected since 3D images are typical large enough, e.g. 3D CT or MRI images in the scope of $512 \times 512 \times 200$ pixels, to guarantee high workload of the GPU computations.

Accelerating an established image registration algorithm by implementing GPU code with the aid of CUDA generated very promising results. However, only by thoroughly analyzing the mathematical foundation, reconstructing the registration algorithm for massively parallel execution and fully utilizing the high and low-level features of the GPU, fast code was gained that even outperformed parallelized CPU code.

References

[1] Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. pp. 483–485. AFIPS '67 (Spring), ACM, New York, NY, USA (1967)

[2] Amit, Y.: A nonlinear variational problem for image matching. SIAM Journal on Scientific Computing 15, 207–224 (1994)

[3] Brown, L.G.: A survey of image registration techniques. ACM Computing Surveys (CSUR) 24(4), 325–376 (1992)

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

[5] Bui, P., Brockman, J.: Performance analysis of accelerated image registration using GPGPU. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. pp. 38–45. ACM (2009)

[6] Cocosco, C., Kollokian, V., Kwan, R.K.S., Evans, A.C.: BrainWeb: Simulated brain data base (2006), `http://brainweb.bic.mni.mcgill.ca/brainweb`

[7] Collignon, A., Maes, F., Delaere, D., Vandermeulen, D., Suetens, P., Marchal, G.: Automated multi-modality image registration based on information theory. In: Information Processing in Medical Imaging. vol. 3, pp. 263–274 (1995)

[8] Corporation, N.: Profiler user's guide (August 2014)

[9] Crum, W.R., Hartkens, T., Hill, D.L.G.: Non-rigid image registration: theory and practice. The British Journal of Radiology 77(2), S140–S153 (2004)

[10] Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5(1), 46–55 (1998)

[11] Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 38(8), 391–407 (2012)

[12] Fluck, O., Vetter, C., Wein, W., Kamen, A., Preim, B., Westermann, R.: A survey of medical image registration on graphics hardware. Computer Methods and Programs in Biomedicine 104(3), e45–e57 (2011)

[13] Fujii, Y., Azumi, T., Nishio, N., Kato, S., Edahiro, M.: Data transfer matters for GPU computing. In: Parallel and Distributed Systems (ICPADS), 2013 International Conference on. pp. 275–282. IEEE (2013)

[14] Galoppo, N., Govindaraju, N.K., Henson, M., Manocha, D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. p. 3. IEEE Computer Society (2005)

[15] Getreuer, P.: Writing MATLAB C/MEX code (April 2010)

[16] Gill, P.E., Murray, W., Wright, M.H.: Practical optimization (1981)

[17] Haber, E., Modersitzki, J.: Intensity gradient based registration and fusion of multi-modal images. In: Medical Image Computing and Computer-Assisted Intervention–MICCAI 2006, pp. 726–733. Springer (2006)

[18] Hajnal, J.V., Saeed, N., Soar, E.J., Oatridge, A., Young, I.R., Bydder, G.M.: A registration and interpolation procedure for subvoxel matching of serially acquired MR images. Journal of computer assisted tomography 19(2), 289–296 (1995)

[19] Harris, M., et al.: Optimizing parallel reduction in CUDA. NVIDIA Developer Technology 2(4) (2007)

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

[21] Huang, T.Y., Tang, Y.W., Ju, S.Y.: Accelerating image registration of MRI by GPU-based parallel computation. Magnetic Resonance Imaging 29(5), 712–716 (2011)

[22] Kasim, H., March, V., Zhang, R., See, S.: Survey on parallel programming model. In: Network and Parallel Computing, pp. 266–275. Springer (2008)

[23] Köhn, A., Drexl, J., Ritter, F., König, M., Peitgen, H.O.: GPU accelerated image registration in two and three dimensions. In: Bildverarbeitung für die Medizin 2006, pp. 261–265. Springer (2006)

[24] König, L., Kipshagen, T., Rühaak, J.: A Non-Linear Image Registration Scheme for Real-Time Liver Ultrasound Tracking using Normalized Gradient Fields. In: MICCAI Challenge on Liver Ultrasound Tracking (CLUST 2014). Boston, USA (September 2014)

[25] Larsen, E.S., McAllister, D.: Fast matrix multiplies using graphics hardware. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing. pp. 55–55. ACM (2001)

[26] Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., et al.: Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In: ACM SIGARCH Computer Architecture News. vol. 38, pp. 451–460. ACM (2010)

[27] Lester, H., Arridge, S.R.: A survey of hierarchical non-linear medical image registration. Pattern recognition 32(1), 129–149 (1999)

[28] Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. Computer Physics Communications 179(9), 634–641 (2008)

[29] Lombardi, F., Spigler, R.: The evolution of the approach to scientific computing: a survey. Journal of Parallel & Cloud computing (2014)

[30] Luna, F.D.: Introduction to 3D game programming with DirectX 10. Jones & Bartlett Publishers (2008)

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

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

[33] Mani, V., Arivazhagan, S.: Survey of medical image registration. Journal of Biomedical Engineering and Technology 1(2), 8–25 (2013)

[34] Membarth, R., Hannig, F., Teich, J., Korner, M., Eckert, W.: Frameworks for GPU accelerators: A comprehensive evaluation using 2D/3D image registration. In: Application Specific Processors (SASP), 2011 IEEE 9th Symposium on. pp. 78–81. IEEE (2011)

[35] Michalakes, J., Vachharajani, M.: GPU acceleration of numerical weather prediction. Parallel Processing Letters 18(04), 531–548 (2008)

[36] Mikov, A.I.: Large-scale addition of machine real numbers: Accuracy estimates. Theoretical computer science 162(1), 151–170 (1996)

[37] Modersitzki, J.: FAIR: flexible algorithms for image registration. SIAM (2009)

[38] Mueller, B., Olesch, J., Lotz, J., Barendt, S., Sedlaczek, O., Lahrmann, B., Grabe, N., Bestvater, F., Kauczor, U., Schnabel, P.A., Hoffmann, H., Fischer, B., Schirmacher, P., Warth, A., Breuhahn, K.: 3D reconstruction of lung adenocarcinomas - one module for the development of mathematical multiscale models of lung cancer (May 2013)

[39] Muyan-Ozcelik, P., Owens, J.D., Xia, J., Samant, S.S.: Fast deformable registration on the GPU: A CUDA implementation of demons. In: Computational Sciences and Its Applications, 2008. ICCSA'08. International Conference on. pp. 223–233. IEEE (2008)

[40] Nocedal, J., Wright, S.J.: Numerical Optimization. Springer (2006)

[41] NVIDIA Corporation: CUDA C best practices guide (February 2014)

[42] NVIDIA Corporation: CUDA compiler driver nvcc (February 2014)

[43] NVIDIA Corporation: NVIDIA CUDA C programming guide (February 2014)

[44] van Oosten, J.: CUDA hierarchy model (November 2011), `http://3dgep.com/?p=2012`

[45] Otake, Y., Armand, M., Armiger, R.S., Kutzer, M.D., Basafa, E., Kazanzides, P., Taylor, R.H.: Intraoperative image-based multiview 2D/3D registration for image-guided orthopaedic surgery: incorporation of fiducial-based C-arm tracking and GPU-acceleration. Medical Imaging, IEEE Transactions on 31(4), 948–962 (2012)

[46] Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU computing. Proceedings of the IEEE 96(5), 879–899 (2008)

[47] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. In: Computer Graphics Forum. vol. 26, pp. 80–113. Wiley Online Library (2007)

[48] Pratx, G., Xing, L.: GPU computing in medical physics: A review. Medical Physics 38(5), 2685–2697 (2011)

[49] Rühaak, J., König, L., Hallmann, M., Papenberg, N., Heldmann, S., Schumacher, H., Fischer, B.: A fully parallel algorithm for multimodal image registration using normalized gradient fields. In: Biomedical Imaging (ISBI), 2013 IEEE 10th International Symposium on. pp. 572–575. IEEE (2013)

[50] Sanders, J., Kandrot, E.: CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional (2010)

[51] Schmitt, O.: Die multimodale Architektonik des menschlichen Gehirns. No. 1 (2001)

[52] Shams, R., Barnes, N.: Speeding up mutual information computation using NVIDIA CUDA hardware. In: Digital Image Computing Techniques and Applications, 9th Biennial Conference of the Australian Pattern Recognition Society on. pp. 555–560. IEEE (2007)

[53] Shams, R., Sadeghi, P., Kennedy, R., Hartley, R.: 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 (2010)

[54] Shams, R., Sadeghi, P., Kennedy, R.A., Hartley, R.I.: A survey of medical image registration on multicore and the GPU. Signal Processing Magazine, IEEE 27(2), 50–60 (2010)

[55] Shi, L., Liu, W., Zhang, H., Xie, Y., Wang, D.: A survey of GPU-based medical image computing techniques. Quantitative Imaging in Medicine and Surgery 2(3), 188 (2012)

[56] Shreiner, D., Sellers, G., Kessenich, J.M., Licea-Kane, B.M.: OpenGL programming guide: The Official guide to learning OpenGL, version 4.3. Addison-Wesley Professional (2013)

[57] Sotiras, A., Davatzikos, C., Paragios, N.: Deformable medical image registration: a survey. Medical Imaging, IEEE Transactions on 32(7), 1153–1190 (July 2013)

[58] Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering 12(3), 66 (2010)

[59] Tramnitzke, F., Rühaak, J., König, L., Modersitzki, J., Köstler, H.: GPU based affine linear image registration using normalized gradient fields. In: Proc. Seventh International Workshop on High Performance Computing for Biomedical Image Analysis (HPC-MICCAI). Boston, MA, USA (September 2014)

[60] Turing, A.M.: Rounding-off errors in matrix processes. The Quarterly Journal of Mechanics and Applied Mathematics 1(1), 287–308 (1948)

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

[62] Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., Shringarpure, A.: On the limits of GPU acceleration. In: Proceedings of the 2nd USENIX conference on Hot topics in parallelism. pp. 13–13. USENIX Association (2010)

[63] Wilt, N.: The CUDA handbook: a comprehensive guide to GPU programming. Pearson Education (2013)

[64] Zikic, D., Glocker, B., Kutter, O., Groher, M., Komodakis, N., Kamen, A., Paragios, N., Navab, N.: Linear intensity-based image registration by Markov random fields and discrete optimization. Medical image analysis 14(4), 550–562 (2010)

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