GPU Based Affine Linear Image Registration using Normalized Gradient Fields

Florian Tramnitzke[1], Jan Rühaak[1,2], Lars König[1,2], Jan Modersitzki[1,2], and Harald Köstler[3]

[1] Universität zu Lübeck, Ratzeburger Allee 160, 23562 Lübeck, Germany
tramnitzke@mic.uni-luebeck.de
[2] Fraunhofer MEVIS, Project Group Image Registration, Maria-Goeppert-Straße 3, 23562 Lübeck, Germany
[3] Friedrich-Alexander-Universität Erlangen-Nürnberg, Schloßplatz 4, 91054 Erlangen, Germany

Abstract. We present a CUDA implementation of a complete registration algorithm, which is capable of aligning two multimodal images, using affine linear transformations and normalized gradient fields. Through the extensive use of different memory types, well handled thread management and efficient hardware interpolation we gained fast executing code. Contrary to the common technique of reducing kernel calls, we significantly increased performance by rearranging a single kernel into multiple smaller ones. Our GPU implementation achieved a speedup of up to 11 compared to parallelized CPU code. Matching two 512×512 pixel images is performed in 37 milliseconds, thus making state-of-the-art multimodal image registration available in real time scenarios.

Keywords: GPU, image registration, high performance computing, NGF

1 Introduction

Image registration is an important task in various areas of applications including medical image computing, see e.g. [8, 10] 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. [3, 19]. This objective is achieved by transforming one image (the template) to another image (the reference). Determining the appropriate class of transformations is generally a difficult task [23]. The class of affine linear transformations, which includes rigid transformations, is important for numerous applications and is used as a pre-processing step in almost all medical image registrations [11]. Furthermore, rigid transformations are broadly established in clinical practice as they preserve geometry and can be performed in comparably short time.
Following [11], an optimization framework can be used to formalize the registration problem mathematically. Here, the transformation is characterized as a

This work was partly funded by the European Regional Development Fund (EFRE).

minimizer of a carefully chosen distance measure D between the reference and transformed template. In particular in a multimodal setting, the choice of D is non-trivial. Mutual Information (MI) [20, 2] is widely used but results in a highly non-convex optimization problem [7]. The Normalized Gradient Fields (NGF) distance measure proposed by Haber and Modersitzki [7] provides a fair compromise between flexibility with respect to modalities and convexity. The numerical computation of a solution of the registration problem, i.e. minimizing the distance measure, can be achieved by numerical optimization. As image registration can be a time consuming task, speeding up the computation is an ongoing effort [18]. Clinical applications like image guided surgery can greatly benefit from accelerating registration algorithms [14].
The Graphics Processing Unit (GPU) is well suited for speeding up problems where the same program is executed on many data elements in parallel [13]. The growing capabilities and use of GPUs in a scientific environment is summarized in [15]. 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) [5]. For instance, in [9] 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 [12]. To gain this performance, the authors point out the importance of memory coalescing and low-level implementation. Comparable speedups were generated in [17] by running MI on the GPU.
In this paper, we present a CUDA implementation of an affine linear registration algorithm using the NGF distance measure. We chose CUDA due to the fact that it is steadily updated, well documented and easy to learn. To our knowledge, this is the first contribution implementing the NGF distance measure on the GPU. In contrary to many contributions [18] 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. [16]. We extend this scheme by using the CUDA framework and exploit several techniques like optimized memory handling, specialized kernel invocation and the efficient use of hardware interpolation. In contrast to common practice, we divided the computationally most demanding kernel into four separate kernels. This resulted in a speed up of up to 60% for the entire algorithm. By taking these aspects into account, we gained fast GPU code that outperforms parallelized CPU code by far.
The paper is organized as follows. Section 2 provides a brief overview of the registration framework, including a short summary of the used registration algorithm and an approach to derive the explicit calculation rule. This is followed by a short overview of the CUDA programming model in Section 3. Next, in Section 4 we present the most important techniques of our CUDA implementa-

tion we used to gain our very fast code. After this, our CUDA implementation is compared to serial and parallelized CPU code in terms of performance and accuracy in Section 5. Finally, in Section 6 the results will be discussed.

2 Affine Linear Image Registration

In this section, we give a brief overview of the general registration framework and the NGF distance measure. More detailed descriptions can be found in [7, 16].

2.1 Registration Framework

Image registration is the process of finding a reasonable transformation y, that describes the spatial correspondence between two images, a reference image R and a template image T. This is typically done by minimizing a distance measure D [11]. The images are defined as continuous mappings $T, R : \Omega \to \mathbb{R}$ with the domain $\Omega \subset \mathbb{R}^2$ in 2D. In this paper, we consider affine linear transformations, i.e. $y_w : \mathbb{R}^2 \to \mathbb{R}^2, y_w(\mathbf{x}) = A\mathbf{x} + b, A \in \mathbb{R}^{2\times 2}, b \in \mathbb{R}^2$, where the six entries in A and b compose the parameters w to be optimized. The transformation $y_w : \Omega \to \mathbb{R}^2$ enables a comparison of the reference image R and the transformed image $T(y_w) := T \circ y_w$ by mapping the reference image domain to the template image domain depending on the parameters w. Rigid transformations can easily be modeled by restricting A to rotations, which yields a total of three degrees of freedom. Now, finding a reasonable correspondence between the images is done by solving the optimization problem through minimizing the objective function

$$D(T(y_w), R) =: D(w) \overset{w}{\to} \min . \tag{1}$$

2.2 Normalized Gradient Fields Distance Measure

As proposed in [7] we use a distance measure based on Normalized Gradient Fields which is both well suited for optimization and fast computation. The main idea is to pointwise measure the angle between two image gradients and to align these in either parallel or antiparallel fashion. A discretized version of the originally continuous distance measure can be written as

$$D_{\mathrm{NGF}}(w) \approx \frac{h_1 h_2}{2} \sum_{i=1}^{MN} \left(1 - \left(\frac{\langle \nabla T_i(y_w), \nabla R_i \rangle}{\| \nabla T_i(y_w) \|_\eta \| \nabla R_i \|_\eta} \right)^2 \right) \tag{2}$$

$$=: \frac{h_1 h_2}{2} \sum_{i=1}^{MN} \left(1 - r_i^2 \right) =: \psi(r(T(y_w))), \tag{3}$$

where $\| \cdot \|_\eta = \langle \cdot, \cdot \rangle + \eta^2$. The parameter η represents a modality dependent parameter which allows filtering of noise. To discretize the image domain Ω we define $[0, M], [0, N]$ as the ranges of indices and h_1, h_2 as the stepwidth in x_1-, and x_2-direction, respectively. We then define $T_i(y_w)$ as the deformed template, interpolated on the reference image domain at point $i = x_1 + M x_2$ using bilinear interpolation and lexicographical ordering.

2.3 Problem Specific Derivative Computation

In order to solve the optimization problem described in (1), we use a multilevel Gauss-Newton optimization scheme [6]. In each optimization step the computation of the function value D, gradient ∇D and the Gauss-Newton approximation [22] of the Hessian $\nabla^2 D$ is required. Using the cascaded formulation in (3) Rühaak et al. [16] derived an explicit calculation rule for NGF based registration. This calculation rule allows the pointwise independent computation of D, ∇D and $\nabla^2 D$. Equation (2) states that the function value can simply be parallelized over all pixels. The efficient parallelization of the gradient and Hessian is done by carefully analyzing the cascaded formulation in (3). Using the chain rule, the formulation can be differentiated as $\nabla D = \frac{\partial \psi}{\partial r} \frac{\partial r}{\partial T} \frac{\partial T}{\partial y} \frac{\partial y}{\partial w}$, where $\frac{\partial y}{\partial w}$ denotes the derivative of the transformation y with respect to the parameters w. The individual factors exhibit a fixed sparse matrix structure [16] which can be exploited. Further, the entries of the matrices only depend on the parameters, reference image and template image and can be calculated independently. Utilizing both features, the multiplication of the four factors can be rewritten into a single matrix-free closed-form formula that can be parallelized for each pixel.

2.4 Optimization Scheme

A pseudo code describing the complete registration algorithm is shown in Algorithm 1. A multilevel representation of the images in different resolutions is generated. Starting on the coarsest level a Gauss-Newton optimization using Armijo line-search [22] is performed. The optimization stops if common stopping criteria [11] are met. The current transformation parameters are then used as a starting guess in the next multilevel iteration until the finest level is reached. While evaluating the stopping rules, solving the 6×6 Gauss-Newton system and updating values requires a low, constant number of floating point operations, evaluating the objective function depends linearly on the number of pixels, making it the computationally most demanding task.

Algorithm 1 Pseudo code for the multilevel Gauss-Newton optimization

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

3 General-Purpose Computing on Graphics Processing Units using CUDA

Before the implementation is explained in detail, a short overview of the CUDA programming model will be given, following the descriptions in [21, 13].
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 by N different CUDA *threads*. Threads are identified by a d-dimensional *thread index* forming d-dimensional *thread blocks*, with $d = 1, 2, 3$. Again, these blocks are grouped into a d-dimensional *grid*. The exact layout of thread blocks and grids is defined by the user when calling a kernel.
CUDA threads can access data from two memory spaces: device memory and cached memory, being off-chip and on-chip, respectively. Device memory is very slow to access but visible to all threads. Global, constant and texture memory resides in device memory. 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 and consecutive reads do not incur additional memory traffic. Shared memory resides in cached memory and is visible to all threads of a block. The access is about 100 times faster than global memory, but the space is limited to 48 kB. 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. High performance is achieved when reading from addresses that are close together in 2D. Additionally, textures offer hardware interpolation and boundary handling. Hardware interpolation enables nearest neighbor or linear interpolation of data and different options how to handle out of range texture fetches, see [13] for more details. Fully exploiting these thread and memory structures was mandatory in obtaining fast GPU code.

4 Implementation

In this section we present the most important techniques of our work to gain high performance, including specialized kernel invocation, optimized memory handling and the efficient use of hardware interpolation.

4.1 Kernel Invocation

As stated before in Section 2.4, 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 $\nabla^2 D$. These have to be computed at least once per optimization level, see Algorithm 1. Since the Gauss-Newton approximation of the Hessian is symmetric

[22], it is sufficient to compute the upper triangular part of the matrix and mirror the entries to the lower triangular part. Thus, we only need to calculate 21 instead of 36 entries of the Hessian. Adding the six components of the gradient and the function value itself, we get 28 scalar values in total. We evaluated different setups for the grid layout 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 we changed the amount of scalar values that were computed per kernel call ranging from 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 kernels was high, we chose testing images of the size 2048×2048 pixels. Therefore, concurrent kernel calls did not improve the performance and kernels were launched sequentially. Table 1 shows the execution times for different setups, stating that the best setup consists of four different kernels which calculate seven scalar values each and have 128 threads per block. This may be contrary to the belief that one kernel doing all the work is the best choice [1]. The setup we use results in quadruple overhead, yet we almost gained a two-fold speedup. Further analysis of the impact of threads per block and the kernel layout is a topic of future work.

Table 1. Performance overview for different kernel setups. The table shows the computation time for calculating D, ∇D and $\nabla^2 D$ for an image of size 2048×2048 pixels in milliseconds. 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 using multiple rearranged kernels and a single kernel, respectively.

Number of kernel calls	Number of scalar values per kernel	Threads per block						
		16	32	64	128	256	512	1024
		Timings in milliseconds						
7	4	102.1	56.5	35.0	30.5	30.7	32.8	36.4
4	7	72.0	40.6	25.0	**18.6**	20.7	23.0	43.8
2	14	53.4	31.0	22.7	24.9	27.1	47.2	o.o.m.
1	28	45.1	**32.2**	37.2	40.4	66.8	o.o.m.	o.o.m.

4.2 Use of Different Memory Types

The use of the different memory types is crucial for writing a fast CUDA kernel. Our first naive approach, using shared memory only to compute vector sums by parallel reduction and store all other data in global memory, resulted in a code that was four times slower than our OpenMP [4] implementation. A first improvement was achieved by writing fixed and often read data into constant memory. The most performance, however, we gained by binding both the template and reference image to texture memory. This greatly enhanced the performance due to the many reads of pixel values. The downside of using texture memory is the limitation to single precision. Fetching data from a texture, thus returns a single precision value impacting the overall accuracy of the algorithm. In our case, the use of a multilevel approach and Gauss-Newton type optimization provides a stable and robust algorithm and the loss of accuracy was generally of no concern in finding reasonable transformation parameters.

4.3 Hardware Interpolation

Another interesting feature of texture memory is hardware interpolation. Instead of calculating a bilinear interpolation from given pixel values, an interpolated value is fetched from the texture cache, which greatly improves performance. The interpolation of a pixel value at the coordinates (x_1, x_2) can be written as

$$p = (1 - dx_1)((1 - dx_2)w_{00} + dx_2 w_{01}) + dx_1((1 - dx_2)w_{10} + dx_2 w_{11}), \qquad (4)$$

where $w_{00} \cdots w_{11}$ are known pixel values and $dx_i = x_i - \lfloor x_i \rfloor, i = 1, 2$ are remainders. The analytical derivative of (4) is defined as

$$\begin{aligned} \frac{\partial p}{\partial x_1} &= (1 - dx_2)(w_{10} - w_{00}) + dx_2(w_{11} - w01), \\ \frac{\partial p}{\partial x_2} &= (1 - dx_1)(w_{01} - w_{00}) + dx_1(w_{11} - w10). \end{aligned}$$

Using textures we obtain

$$\begin{aligned} p &= f(x_1, x_2), \\ \frac{\partial p}{\partial x_1} &= f(1, x_2) - f(0, x_2), \\ \frac{\partial p}{\partial x_2} &= f(x_1, 1) - f(x_1, 0). \end{aligned}$$

This implies that not only the function value but also the analytical derivative can directly be computed by four calls to hardware interpolation, again increasing the performance. Additionally, we can omit zero Dirichlet boundary handling by setting the address mode to *cudaAddressModeBorder*.

5 Results

To test our code, we registered images ranging from 512×512 to 4096×4096 in pixel size using the algorithm described in Section 2.4. As a safeguard, a maximum of ten iterations per optimization level was used. 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. The timings of our CUDA code are compared to MatLab code based on the FAIR framework [11] and optimized C++ code with and without OpenMP. The general quality of the algorithm is shown in [11] and will not be discussed in this paper. The images show two slices of a brain acquired for histological serial sectioning and are courtesy of Oliver Schmitt, Institute of Anatomy, University of Rostock, Germany. The reference and template image are shown in images [A] and [B] of Figure 1. The tests were executed on a desktop PC with an Intel Core i7-2600 CPU @ 3.40 GHz and a NVIDIA GeForce 680 GTX with 2 GB GDDR5 memory allowing a bandwidth of 192.2 Gb/sec. It has 1536 CUDA cores @ 1.008 GHz and supports compute capability 3.0. The evaluation was done in MatLab via *mex-files*. The use of MatLab impacts the overall performance, but we found the performance loss to be negligible. The tests were run 15 times to minimize outliers.

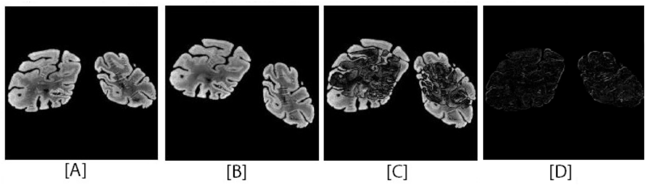

Fig. 1. Brain images used for testing the registration algorithm. [A] reference image R, [B] template image T, [C] difference: $|R - T|$, [D] difference: $|R - T(y)|$

5.1 Performance

Table 2 summarizes the timing results for different image sizes for the complete registration algorithm. It is clear from this table, that the GPU implementation is at least six times faster than competing parallelized CPU code. Registering two 512×512 pixel images is performed in 37 ms. The speedup increases with increasing image size, showing that it is desirable to have many computations on the GPU and to minimize the CPU-GPU-communication.

Table 2. Summary of runtimes in seconds of the complete registration for different images. The speedup of the routines are compared to OpenMP code, showing that our CUDA code outperforms even fast parallelized CPU code. The tests were done using an Intel Core i7-2600 and an NVIDIA GeForce 680 GTX, see Section 5 for details.

image size in pixel	method	runtime (s)	speedup
512×512	FAIR	3.090	0.08
	C++	0.475	0.50
	OpenMP	0.239	1.00
	CUDA	**0.037**	6.64
1024×1024	FAIR	5.355	0.09
	C++	1.437	0.33
	OpenMP	0.481	1.00
	CUDA	0.062	7.76
4096×4096	FAIR	89.314	0.06
	C++	20.596	0.25
	OpenMP	5.104	1.00
	CUDA	0.464	**11.00**

5.2 Accuracy

Images [C] and [D] of Figure 1 show the initial difference and the final difference after registration, respectively. The relative error of the final transformation parameters compared to the FAIR code is in the magnitude of $1 \cdot 10^{-3}$, but final images of the different implementations show no difference in visual inspection. Hence, errors made due to single precision accuracy are negligible.

6 Discussion

In this paper, we presented a fast GPU implementation of a multimodal image registration algorithm using the NGF distance measure. We exploited the inner structure of the underlying algorithm to enable full pixelwise parallelization.
We evaluated the code on brain images of different size on a PC with standard hardware. A speedup of 11 compared to our OpenMP implementation was achieved for the largest images. This shows that for conventional systems with one CPU and one GPU, implementing algorithms on the GPU has a lot performance to offer. This can be especially interesting in medical applications, where results are needed as fast as possible and PC clusters are not an option.
Further, we analyzed the impact of optimized memory handling, specialized kernel invocation and the efficient use of hardware interpolation. By fully utilizing these techniques high performance from GPU code was gained and parallelized OpenMP code was outperformed. Compared to research code of the FAIR framework a speedup of almost 200 was gained, showing the immense potential of high performance computing techniques applied to research code in medical imaging.

Bibliography

[1] 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)

[2] 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)

[3] 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)

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

[5] 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)

[6] Haber, E., Modersitzki, J.: A multilevel method for image registration. SIAM Journal on Scientific Computing 27(5), 1594–1607 (2006)

[7] 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)

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

[9] 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)

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

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

[12] 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)

[13] NVIDIA Corporation: NVIDIA CUDA C Programming Guide (2014)

[14] 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)

[15] 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)

[16] 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)

[17] 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)

[18] 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)

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

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

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

[22] Wright, S., Nocedal, J.: Numerical optimization, vol. 2. Springer New York (1999)

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