FAST GPU 3D DIFFEOMORPHIC IMAGE REGISTRATION

MALTE BRUNN*, NAVEEN HIMTHANI†, GEORGE BIROS†, MIRIAM MEHL*, AND ANDREAS MANG‡

Abstract. 3D image registration is one of the most fundamental and computationally expensive operations in medical image analysis. Here, we present a mixed-precision, Gauss–Newton–Krylov solver for diffeomorphic registration of two images. Our work extends the publicly available `CLAIRE` library to GPU architectures. Despite the importance of image registration, only a few implementations of large deformation diffeomorphic registration packages support GPUs. Our contributions are new algorithms to significantly reduce the run time of the two main computational kernels in `CLAIRE`: calculation of derivatives and scattered-data interpolation. We deploy (i) highly-optimized, mixed-precision GPU-kernels for the evaluation of scattered-data interpolation, (ii) replace Fast-Fourier-Transform (FFT)-based first-order derivatives with optimized 8th-order finite differences, and (iii) compare with state-of-the-art CPU and GPU implementations. As a highlight, we demonstrate that we can register 256^3 clinical images in less than 6 seconds on a single NVIDIA Tesla V100. This amounts to over 20× speed-up over the current version of `CLAIRE` and over 30× speed-up over existing GPU implementations.

1. INTRODUCTION. Image registration (also known as image alignment, warping, or matching) is an important task in medical image analysis [75]. It is used in computer aided diagnosis and clinical population studies. A comprehensive overview can be found in [31, 59, 60, 75]. The image registration problem is roughly this: Given two images $m_0(\boldsymbol{x})$ (the template image) and $m_1(\boldsymbol{x})$ (the reference image; here, $\boldsymbol{x} \in \Omega \subset \mathbb{R}^3$), we seek a spatial transformation $\boldsymbol{y}(\boldsymbol{x})$ such that the deformed template image $m_0(\boldsymbol{y}(\boldsymbol{x}))$ is similar to $m_1(\boldsymbol{x})$ [59]. Registration methods can be classified according to the parameterization for $\boldsymbol{y}$. In this paper, we consider methods that belong or are related to large-deformation diffeomorphic metric mapping (**LDDMM**) [11, 85]. Such mappings provide maximal flexibility [75]. LDDMM maps are expensive to compute since they are infinite-dimensional. Upon discretization, the number of unknowns for $\boldsymbol{y}$ is still in the millions. For example, registering two 256^3 images requires calculating a 256^3 resolution stationary velocity field $\boldsymbol{v}(\boldsymbol{x})\mathbb{R}^3$ with ≈50 M unknowns. Furthermore, LDDMM registration is a highly non-linear and ill-conditioned inverse problem [31]. As a result, image registration can take a few minutes on multi-core high-end CPUs. As large clinical, cross-center, population-study workflows require thousands of registrations, reducing the compute time of a single registration to seconds translates to a reduction of clinical study time from weeks to a few days. GPUs with their inherent parallelism and low energy consumption are an attractive choice to achieve this goal. However, despite the need for high-througput computational performance for registration, and the existence of several software libraries for LDDMM registration, there is little work on highly optimized GPU implementations (see §1.3 below).

1.1. Contributions. Based on the open source diffeomorphic image registration framework `CLAIRE` [51, 52, 54–56], we introduce a new, optimized, GPU implementation of LDDMM registration. The overall mathematical formulation and solution strategy remains unaltered from [56]. We propose several modifications of the ***differentiation*** and ***interpolation*** kernels, which are the main computational kernels in `CLAIRE`. More specifically, our contributions are:

- **Interpolation:** The first important computational kernel is scattered-data interpolation used for semi-Lagrangian advection. `CLAIRE` originally employed a Lagrange-basis cubic interpolation. We study several alternative methods on GPUs using a combination of pre-filtering, texture, and polynomial interpolation. We study their accuracy and performance using simple performance models and vendor performance profiling tools in §3.
- **Differentiation:** The second important computational kernel is computing derivatives (gradient and divergence) of 3D images (scalar fields). We introduce a mixed-precision implementation using 8^{th} order finite-difference (**FD8**) kernels to replace FFT-based spectral derivatives. In particular, we replace all first order derivatives that appear in the partial differential equations (**PDE**) of our optimality systems. Note that FFTs are still retained for higher-order derivatives and their inverse. We discuss this in detail in §3.

*Institute for Parallel and Distributed Systems, University of Stuttgart, Stuttgart 70569 DE, malte.brunn@ipvs.uni-stuttgart.de, miriam.mehl@ipvs.uni-stuttgart.de

†Oden Institute of Computational Engineering and Sciences, The University of Texas at Austin, TX 78712, USA, naveen@ices.utexas.edu, gbiros@acm.org

‡Department of Mathematics, University of Houston, TX 77204, USA, andreas@math.uh.edu

- **Evaluation:** We evaluate the new algorithm on four Magnetic Resonance Imaging **(MRI)** scans and for three different image resolutions. We compare the proposed method with the original `CLAIRE` in §4 as well as with the GPU packages `PyCA` [65] and `deformetrica` [16, 26]. We discuss these experiments in detail in §4. Overall, the method is over 20× faster than the original CPU-based `CLAIRE` and produces registration maps of similar quality. This speedup does not only reflect hardware differences but mostly algorithmic changes, some of which could also be implemented in a CPU version. Furthermore, reducing the accuracy of certain calculations to exploit hardware acceleration has no negative effects on the quality of the registration.

1.2. Limitations. The original implementation of `CLAIRE` was built to support the Message Passing Interface (**MPI**) for parallelism [36, 55, 56]. Our proposed adaption for GPUs has not been integrated with MPI yet. This will be subject to future work, in particular the integration of the high-speed GPU interface *NVLink* in a multi-node multi-GPU context. Thus, our solver does not scale to the image sizes that can be handled by `CLAIRE`. However, this is not an issue for clinical images since typical image sizes fit in a single GPU*.

1.3. Related Work. We refer to [31, 59, 60, 75] for recent developments in image registration. Surveys of GPU accelerated solvers can be found in [29, 33, 72]. As mentioned above, this work extends `CLAIRE` [36, 52, 54, 56]. Popular (in clinical studies) software packages for deformable registration are `IRTK` [66], `elastix` [46], `NiftyReg` [58], and `FAIR` [60]. GPU implementations of (low-dimensional) parametric approaches are described in [30, 58, 70, 71]. Fast GPU implementations of (high-dimensional) non-parametric formulations available in `FAIR` are presented in [18, 47]. Unlike `CLAIRE`, these methods do not guarantee that the computed map $\boldsymbol{y}$ is a diffeomorphism. One possibility to safeguard against non-diffeomorphic maps $\boldsymbol{y}$ is by augmenting the formulation by hard and/or soft constraints on $\boldsymbol{y}$ [19], which introduces significant algorithmic complications. Another approach to enable diffeomorphic registration is to parametrize $\boldsymbol{y}$ via a smooth velocity field $\boldsymbol{v}$ [25, 76]. This approach has been termed LDDMM. The formulation in `CLAIRE` is closely related to LDDMM. A key difference is that LDDMM is based on non-stationary (time-dependent) $\boldsymbol{v}$ but `CLAIRE` uses stationary $\boldsymbol{v}$. Other approaches that use stationary $\boldsymbol{v}$ are described in [2, 3, 42, 49, 50, 79]. There exists a large body of literature on LDDMM-type approaches that, in many cases, mostly focuses on theoretical considerations [25, 57, 84–86]. There is much less work on the design of efficient solvers; examples are [3, 4, 6, 7, 11, 64, 79, 87, 88]. Popular software packages for LDDMM arediffeomorphic `Demons` [79], `ANTs` [5, 6], `DARTEL` [3], `deformetrica` [15, 16, 26, 32], and `PyCA` [65]. A GPU implementation of the diffeomorphic `Demons` algorithm is described in [23, 38]. The runtime reported in [23] is in the order of 60 s on a Quadro FX 1400 for a dataset of size 128^3 [23] (2 s per iteration)†. A multi-GPU implementation of `DARTEL` is described in [77, 78]. The work in [88] introduces `FLASH`, a fast CPU implementation for LDDMM. It is based on a band-limited spectral discretization targeting low resolution images to speed up the computations. By truncating the problem to 16 frequencies along each spatial dimension, the runtime is reduced from 45 s to under 2 s per iteration, resulting in an overall execution time of ≈200 s for 100 gradient descent steps. In [39, 40], a (multi-)GPU implementation of the LDDMM approach described in [45] is presented; the runtime of this solver is in the order of 12 s on a single NVIDIA Quadro FX5600 for a dataset of size 256^3 [40]. In [37], a GPU accelerated LDDMM implementation called `FastReg` is introduced. The authors report results for neuroimaging data with an average DICE of ≈0.67 (much smaller than our results) and a runtime of ≈35 s on a GeForce RTX 2080Ti. A GPU implementation of an LDDMM formulation for point cloud matching (not images) is described in [74]. The software package `deformetrica` [16] parametrizes $\boldsymbol{y}$ by a finite set of control points [27]. The gradient is computed via automatic differentiation [63]. The timings reported in [16] for the registration of an image of size $181 \times 217 \times 181$, executing 50 iterations, are 102 s and 202 s (Nvidia Quadro M4000) for two variants of the GPU implementation, respectively. The execution time for the CPU version of `deformetrica` is ≈10 h (Intel Xeon E5-1630). The runtime for the GPU variant of `PyCA` [65] reported in [83] for a $229 \times 193 \times 193$ neuroimaging dataset is 648 s (Nvidia TitanX (Pascal)). Many of these methods reduce the unknowns by

*The GPU implementation is for a single GPU only and, therefore, limited by the memory available on the considered card (NVIDIA Tesla V100 in our case). The typical size for clinical images (magnetic resonance imaging) is approximately 256^3 and fits into memory of a single GPU for the current implementation.

†All timings here are for single-precision calculations, which is typically used in practice. Our results for the proposed method are for single-precision as well.

using coarser resolutions, and use algorithms that produce a registration quality that is not as good as CLAIRE in terms of Jacobians.

Another approach that can speed up image registration is deep learning [8,48,82,83]. As an example, the training in [82] is performed with PyCA; it takes ≈72 h. After training, the reported runtime for the registration of $229 \times 193 \times 193$ images is 18.43 s on a single Nvidia TitanX (Pascal) [83], which is significantly slower than our method. Most importantly, it is unclear how deep learning performs on unseen clinical datasets.

1.4. Outline. We summarize the overall formulation §2.1 and algorithms §2.2 in CLAIRE. All material in §2.1 and §2.2 is discussed in detail in the works [36, 52, 53, 55, 56]. In §2.3, we present the two main computational kernels, the scattered-data interpolation and the approximation of first-order spatial derivatives.

2. METHODS.

2.1. Formulation. CLAIRE uses an optimal control formulation. Instead of solving for the LDDMM $\boldsymbol{y}(x)$, it reformulates the problem for a ***velocity*** $\boldsymbol{v}(\boldsymbol{x})$ that generates $\boldsymbol{y}(x)$. Specifically, given two images $m_0(\boldsymbol{x})$ (template image; image to be registered to reference image) and $m_1(\boldsymbol{x})$ (reference image), we seek a *stationary* velocity field $\boldsymbol{v}(\boldsymbol{x})$ by solving

$$\underset{\boldsymbol{v}}{\text{minimize}}\ \frac{1}{2}\int_\Omega (m(\boldsymbol{x},1)-m_1(\boldsymbol{x}))^2\,\mathrm{d}\boldsymbol{x} + \frac{\beta}{2}\int_\Omega \langle \mathcal{A}\boldsymbol{v}(\boldsymbol{x}),\boldsymbol{v}(\boldsymbol{x})\rangle\,\mathrm{d}\boldsymbol{x} \tag{1a}$$

$$\begin{aligned}\text{subject to}\ \ \partial_t m(\boldsymbol{x},t) + \boldsymbol{v}(\boldsymbol{x})\cdot\nabla m(\boldsymbol{x},t) &= 0 && \text{in } \Omega\times(0,1],\\ m(\boldsymbol{x},t) &= m_0(\boldsymbol{x}) && \text{in } \Omega\times\{0\}\end{aligned} \tag{1b}$$

with periodic boundary conditions on $\partial\Omega$. The PDE constraint in (1b) is the forward problem of our formulation describing the deformation of the state variable $m(\boldsymbol{x},t)$. Given a candidate $\boldsymbol{v}(\boldsymbol{x})$, we model the geometric transformation of the template image $m_0(\boldsymbol{x})$ by transporting its intensities forward in time. The first term in (1a) is an image similarity term (without loss of generality, we use the squared L^2-distance). The second term in (1a) is a Tikhonov regularization functional with regularization parameter $\beta > 0$. It is introduced to ensure smoothness of $\boldsymbol{v}(\boldsymbol{x})$ so that the geometric transformation of $m_0(\boldsymbol{x})$ exists and is a diffeomorphism. We refer to [10, 11, 17, 21, 80, 85] for a theoretical discussion about uniqueness and well-posedness of the forward and inverse problem. We follow the default configuration of CLAIRE and select $\mathcal{A}$ to be a vector Laplacian combined with an additional penalty on the divergence of $\boldsymbol{v}$. We refer to [52, 56] for details.

2.2. Discretization and Numerical Algorithms. We use a second-order gradient based method to solve the PDE-constrained optimization problem (1). The gradient is given by the first-order optimality conditions. We use the method of Lagrange multipliers, and take variations with respect to m, λ (adjoint variable introduced below), and $\boldsymbol{v}$. The first-order optimality conditions amount to a set of coupled, nonlinear, hyperbolic-elliptic PDEs in 4D (space-time). The Lagrangian is given by

$$\begin{aligned}\mathcal{L}[m,\lambda,\boldsymbol{v}] &= \frac{1}{2}\int_\Omega (m(\boldsymbol{x},1)-m_1(\boldsymbol{x}))^2\,\mathrm{d}\boldsymbol{x} + \frac{\beta}{2}\int_\Omega \langle \mathcal{A}\boldsymbol{v}(\boldsymbol{x}),\boldsymbol{v}(\boldsymbol{x})\rangle\,\mathrm{d}\boldsymbol{x}\\ &+ \int_0^1\int_\Omega \lambda(\boldsymbol{x},t)(\partial_t m + \boldsymbol{v}\cdot\nabla m)\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}t\end{aligned}$$

2.2.1. Optimality Conditions & Reduced Space Approach. The first order optimality conditions of (1) consist of three equations. First, the forward problem (1b) (variation of $\mathcal{L}$ with respect to λ). Second, the backward in time adjoint problem (variation of $\mathcal{L}$ with respect to m):

$$\begin{aligned}-\partial_t\lambda(\boldsymbol{x},t) - \nabla\cdot\lambda(\boldsymbol{x},t)\boldsymbol{v}(\boldsymbol{x}) &= 0 && \text{in } \Omega\times[0,1),\\ \lambda(\boldsymbol{x},t) &= m_1(\boldsymbol{x}) - m(\boldsymbol{x},t) && \text{in } \Omega\times\{1\}\end{aligned} \tag{2}$$

with periodic boundary conditions on $\partial\Omega$. Third, the so-called reduced gradient system (variation of $\mathcal{L}$ with respect to $\boldsymbol{v}$) $\boldsymbol{g}(\boldsymbol{v}) = \boldsymbol{0}$, where

$$\boldsymbol{g}(\boldsymbol{v}) := \beta\mathcal{A}\boldsymbol{v}(\boldsymbol{x}) + \int_0^1 \lambda(\boldsymbol{x},t)\nabla m(\boldsymbol{x},t)\,\mathrm{d}t \quad \text{in } \Omega. \tag{3}$$

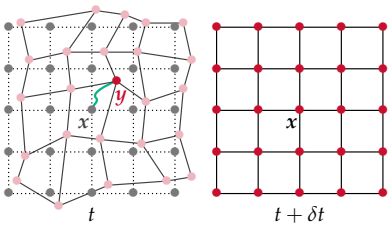

Fig. 1: *Illustration of the computation of the characteristic in the semi-Lagrangian scheme. We start with a regular grid at time $t + \delta t$ and solve for the characteristic $\boldsymbol{y}$ at a given point $\boldsymbol{x}$ backward in time (green line in the graphic on the left). The deformed grid configuration is overlaid onto the initial regular grid at time t. (Figure modified from [53].)*

CLAIRE uses a reduced-space approach, i.e., it iterates on the reduced-space of $\boldsymbol{v}$: given the current iterate $\boldsymbol{v}(\boldsymbol{x})$, it solves for $m(\boldsymbol{x}, t)$ and $\lambda(\boldsymbol{x}, t)$ using (1b) and (2), and substitutes m and λ to evaluate the gradient $\boldsymbol{g}(\boldsymbol{v})$. CLAIRE uses a Newton–Krylov method to solve the reduced gradient system $\boldsymbol{g}(\boldsymbol{v}) = \boldsymbol{0}$ for $\boldsymbol{v}$. We provide more details in §2.2.3.

2.2.2. Discretization. In CLAIRE, the forward and the adjoint systems of PDEs (1b) and (2) are discretized in the space-time interval $\Omega \times [0, 1]$, $\Omega := (0, 2\pi)^3 \subset \mathbb{R}^3$. All spatial fields are periodic in space and discretized using $N = N_1 N_2 N_3$ equispaced grid points x_{ijk}. CLAIRE uses N_t time steps for the forward and adjoint problems and a semi-Lagrangian scheme (see Figure 1) for the transport equations (see [53, 55]). It is implemented in two steps: (*i*) the solution of an ODE $\partial_t \boldsymbol{y}(t) = \boldsymbol{v}(\boldsymbol{y}(t))$ in $[t, t + \delta t)$ with final condition $\boldsymbol{y}(t + \delta t) = \boldsymbol{x}$ backward in time to compute the characteristic $\boldsymbol{y}$ along which points move; (*ii*) the solution of an ODE along this characteristic $\boldsymbol{y}$ is used to compute the change of a transported quantity of interest.

Furthermore, CLAIRE uses FFT-based spectral differentiation in several places. The linearized forward problem requires the gradient operator. The adjoint problem requires the computation of the divergence operator. The reduced gradient (3) involves $\mathcal{A}$ (which is a vector Laplacian), a Leray projection, and a gradient operator (see [53] for details on the formulation). Spectral differentiation was chosen because it diagonalizes $\mathcal{A}$. Using a different scheme would introduce significant complications. But the divergence and gradient operators, which are *applied for each time point*, do not need to be done with FFTs, and this is what we exploit in §2.3 to accelerate CLAIRE.

2.2.3. Newton–Krylov Solver. CLAIRE uses a Gauss–Newton–Krylov method globalized with an Armijo line search to find the root of (3) for $\boldsymbol{v}$. This separates CLAIRE from many of the existing registration packages for velocity-based diffeomorphic image registration (see §1.3 for a discussion). Developing second-order methods for large-scale, nonlinear control problems presents us with numerous challenges [12–14,44]. If implemented naively, these methods can become computationally prohibitive, despite their improved rate of convergence.

We iterate on the discretized velocity $\mathbf{v} \in \mathbb{R}^{3N}$ according to

$$\mathbf{v}_{k+1} = \mathbf{v}_k + \alpha_k \tilde{\mathbf{v}}_k, \quad \mathbf{H}\tilde{\mathbf{v}}_k = -\mathbf{g}_k, \quad k = 0, 1, 2, \ldots \tag{4}$$

where $\mathbf{H} \in \mathbb{R}^{3N,3N}$ is the discretized Gauss-Newton Hessian operator (or simply Hessian for the rest of the paper), $\tilde{\mathbf{v}}_k \in \mathbb{R}^{3N}$ is the search direction, $\mathbf{g}_k \in \mathbb{R}^{3N}$ is the discretized gradient given by (3), $\alpha_k > 0$ is a line search parameter, and $k \in \mathbb{N}$ is the Gauss–Newton iteration count. To compute $\tilde{\mathbf{v}}_k$ we have to solve the linear system in (4) at each Gauss–Newton step. We cannot form or store $\mathbf{H}$ since it is a $3(N_1 N_2 N_3)$-by-$3(N_1 N_2 N_3)$ matrix. We invert $\mathbf{H}$ iteratively using the preconditioned conjugate gradient method (**PCG**) [43]. Applying the Hessian to a vector (we refer to this operation as the *Hessian matvec*) is similar to evaluating the gradient in (3); it requires the solution of two PDEs, one forward in time, and one backward in time. We can split the Hessian operator into two terms, $\mathbf{H} = \mathbf{A} + \tilde{\mathbf{H}}$, where $\mathbf{A} \in \mathbb{R}^{3N,3N}$ is the discretized regularization operator $\mathcal{A}$; $\tilde{\mathbf{H}} \in \mathbb{R}^{3N,3N}$ involves inverses of the state and adjoint operators computed by solving two transport equations. Solving these two PDEs is costly; approximating $\mathbf{H}^{-1}$ using PCG at every Gauss-Newton step takes over 90% of the runtime of CLAIRE for clinical images [56].

2.3. Computational Kernels. Let us first summarize the overall algorithm. As we just discussed, we use a Gauss–Newton–Krylov method (4) to solve the reduced gradient system $\boldsymbol{g}(\boldsymbol{v}) = \boldsymbol{0}$ for $\boldsymbol{v}$. The matrix-free Gauss–Newton Hessian involves solving forward and adjoint hyperbolic PDEs for the linearized (1b) and (2). If we use N_t time steps, each Hessian matvec requires $2N_t$ semi-Lagrangian steps, $2N_t$ gradient

Table 1: *We report the complexity of our solver for the compressible case. We report the number of FFT operators (#FFTs, split into first order derivatives and other, i.e., higher order or inverse operators) and the number of scattered data interpolations (#IPs) that need to be performed for evaluating the objective functional, the gradient (notice, that the evaluation of the gradient requires forward and adjoint PDE solves), and the Hessian matvec (Gauss–Newton approximation; requires the evaluation of the incremental adjoint and state equations as subfunctions). The first order operators are either implemented as FFT or finite differences (#FD). We report generic numbers; $d \in \{2,3\}$ denotes the dimension of the ambient space ($d = 3$ in our case) and N_t is the number of time steps (we set $N_t = 4$). Each Newton iteration requires the evaluation of the objective and the evaluation of the gradient. Each line search step requires the evaluation of the objective function. We demonstrated in [36,55,56] (CPU implementation of CLAIRE) that about 90% of the runtime is spent on evaluating FFTs and the IP model. To reduce the memory footprint of our solver, we evaluate parts of the gradient and Hessian matvec during the solution of the adjoint operators. The memory pressure is $\mathcal{O}((N_t+7)N_1N_2N_3)$ for the gradient and $\mathcal{O}((N_t+10)N_1N_2N_3)$ for the Hessian matvec, respectively.*

function	subfunction	symbol symbol	#FFTs / #FD (1st order)	#FFTs (other)	#IPs
objective functional		—	—	d	$d+N_t$
	state equation (SE)	m	—	—	$d+N_t$
gradient		$\boldsymbol{g}$	$d(N_t+2)$	d	$d+N_t+1$
	adjoint equation (AE)	λ	d	—	$d+N_t+1$
Hessian matvec		$\mathbf{H}\tilde{v}$	$d(2N_t+3)$	d	$d+(d+2)N_t+1$
	incremental SE	$\tilde{m}$	$d(N_t+1)$	—	$d+(d+1)N_t$
	incremental AE	$\tilde{\lambda}$	d	—	N_t+1

operators, and N_t divergence operators. In addition, the Hessian matvec needs $\mathbf{A}$ and its inverse, which are computed as spectral operators using FFTs. All these operators have $\mathcal{O}(N)$ complexity per time step, up to a logarithmic prefactor. The total number of Hessian matvecs is the sum of PCG iterations across Newton steps. Table 1 lists the number of FFTs and interpolations in more detail. The overall method is outlined in Algorithm 2.1. The original `CLAIRE` implementation for CPUs used FFTs for gradients, divergences, $\mathbf{A}$ and $\mathbf{A}^{-1}$, and a highly optimized cubic Lagrange interpolation for the semi-Lagrangian method [56]. We transformed all computational kernels to GPU architectures, and most importantly, we introduced several algorithmic innovations to speed-up both derivatives and interpolations. First, we discuss several options for the interpolation. Second, we replace all gradient and divergence operators with high-order finite-difference (**FD**) operators.[‡] Notice that we keep the spectral differentiation for high-order differential operators, since we need to evaluate their inverses in our solver (spectral preconditioner and Leray projection). Computing their inverses can be done efficiently in the spectral domain; for FD it would require linear solves. We show that, for the given image resolution and floating point accuracy, replacing the spectral methods with high-order FD discretizations allows us to maintain accuracy but significantly increase efficiency on GPUs. To the best of our knowledge, we are the first group to implement this type of mixed-precision code in a hardware and resolution adaptive way. Again, the spectral differentiation is kept for evaluating $\mathbf{A}$ (and its inverse to avoid an additional need for linear solvers); the GPU implementation of the proposed method employs a hybrid differentiation scheme that uses both FFTs and finite differences.

2.3.1. GPU Interpolation. The semi-Lagrangian scheme requires costly interpolation of velocities and scalar image fields along backward characteristics as shown in Figure 1. `CLAIRE` uses Lagrange-based cubic interpolation. GPUs provide two technologies that we exploit in our schemes: texture fetches and hardware support for trilinear interpolation (although not fully single-precision). In addition to these modifications, we also consider another change: switching from Lagrange cubic to B-spline cubic interpolation. The generic formula for interpolating at an off-grid point $\boldsymbol{x} := (x_1, x_2, x_3) \in \mathbb{R}^3$ is given by

$$f(x_1,x_2,x_3) = \sum_{i,j,k=0}^{d} c_{ijk}\phi_i(x_1)\phi_j(x_2)\phi_k(x_3), \tag{5}$$

where $c_{ijk} \in \mathbb{R}$ are scalar coefficients associated with each grid point, $d \in \mathbb{N}$ is the polynomial order, and

[‡]We note that low-order (first and second order) FD (and finite volume) operators are a common choice in image registration [59, 60].

Algorithm 2.1 Basic algorithm for a Gauss–Newton-Krylov step (4) in CLAIRE to solve the reduced gradient system $\boldsymbol{g}(\boldsymbol{v}) = \boldsymbol{0}$ for $\boldsymbol{v}$.

loop *root of* (3) ▷ Newton method ($\mathbf{g}(v) = 0$)
  ObjectiveFunctional($\boldsymbol{v}$) ▷ *as defined in* (1a)
    $m \leftarrow$ StateEquation($\boldsymbol{v}, m_0$) ▷ (1b)
  Gradient($\boldsymbol{v}$) ▷ (3)
    $\lambda \leftarrow$ AdjointEquation($\boldsymbol{v}, m, m_R$) ▷ (2)
  loop KrylovSolver($\tilde{\boldsymbol{v}}, \epsilon_K$) ▷ solve (4)
    HessianMatVec($\tilde{\boldsymbol{v}}$) ▷ $\beta\mathcal{A}\tilde{\boldsymbol{v}} + \int \tilde{\lambda}\nabla m \, \mathrm{d}t$
      $\tilde{m} \leftarrow$ IncStateEquation($\boldsymbol{v}, \tilde{\boldsymbol{v}}$) ▷ $\partial_t \tilde{m} + \boldsymbol{v} \cdot \nabla \tilde{m} + \tilde{\boldsymbol{v}} \cdot \nabla m = 0$
      $\tilde{\lambda} \leftarrow$ IncAdjointEquation($\boldsymbol{v}, \tilde{m}$) ▷ $-\partial_t \tilde{\lambda} - \nabla \cdot \boldsymbol{v}\tilde{\lambda} = 0$
    Preconditioner($\mathbf{r}$)
      $\beta^{-1}\mathcal{A}^{-1}\mathbf{r}$
  loop LineSearch(α)
    ObjectiveFunctional($\boldsymbol{v} + \alpha\tilde{\boldsymbol{v}}$)
      $m \leftarrow$ StateEquation($\boldsymbol{v} + \alpha\tilde{\boldsymbol{v}}, m_0$)
  $\boldsymbol{v} \leftarrow \boldsymbol{v} + \alpha\tilde{\boldsymbol{v}}$ ▷ Newton step

$\phi_i(x_1)$, $\phi_j(x_2)$, $\phi_k(x_3)$ are the basis functions. For Lagrange interpolation, the coefficients equal the grid values ($c_{ijk} = f_{ijk}$), and the ϕ's are the Lagrange polynomials. We use third order cubic ($d = 3$) but we also consider first-order trilinear interpolation ($d = 1$) since GPUs offer hardware acceleration for it. So, we need to evaluate a set of 64 (cubic) or 8 (linear) grid values f_{ijk}. However, there are other options. For example, we can use uniform B-splines for ϕ. In that case, the coefficients c_{ijk} are non-local—they depend on all grid values f_{ijk} unlike the Lagrange case [69]. Below we give the implementation details for the different schemes.

- **GPU-TXTLIN:** Here we use NVIDIA's libraries for trilinear interpolation [1, 73]. It is efficiently performed using NVIDIA's hardware-accelerated texture units (using the `tex3D()` function). The texture units store the coefficients of the trilinear interpolation in 9-bit precision and return the result in single precision. We observed some effects in the registration quality in terms of smoothness of the deformation and the overall mismatch—especially in lower-resolutions or when the image has high frequency components.
- **GPU-LAG:** This is our baseline since it represents a direct translation of the existing algorithm in CLAIRE to GPUs. The c_{ijk} values required to evaluate f are ordered lexicographically. This ordering results in non-coalesced memory accesses that reduce performance. To partially improve this, we use the texture function `tex3D()` as a table lookup to access c_{ijk} and evaluate (5). We remark that we use the texture memory only for look ups and not for trilinear interpolation.
- **GPU-TXTLAG:** This is also a cubic Lagrange interpolation but now we use texture-based interpolation (as opposed to using textures as a table lookup), and thus the accuracy is reduced compared to GPU-LAG. However, in our experiments we don't observe any significant difference in the accuracy. The algorithm is based on the same principle as presented in [68]. Instead of doing *eight* weighted trilinear interpolations, we do *27 weighted trilinear* interpolations at off-grid points. The different number of trilinear interpolations arises due to differences in the Lagrange and B-spline polynomials. Nevertheless, because of hardware acceleration, GPU-TXTLAG significantly outperforms GPU-LAG.
- **GPU-TXTSPL:** The algorithm we use is exactly the one presented in [68]. The implementation is based on the open source library [67], with a major modification related to pre-filtering. We replaced the pre-filter in [67] with a finite convolution inspired by [20]. The pre-filtering to compute the coefficients c_{ijk} then becomes a 15-point axis aligned stencil operation on f_{ijk} and is implemented using the FD scheme used in the CUDA SDK example [41]. We also modified the

code to support periodic boundary conditions. Then, following [68], we use *eight* weighted trilinear ($8 \times 8 f_{ijk}$) interpolations to compose the cubic B-spline interpolation. These interpolations require *eight* texture fetches at off-grid points. Overall, GPU-TXTSPL significantly outperforms GPU-TXTLAG.

2.3.2. GPU Derivatives. The CPU `CLAIRE` uses FFTs to perform spatial differentiation [36]. Since our functions are periodic, all such operators are diagonal in the spectral domain. But in the proposed GPU implementation, we use an FD scheme that is more accurate (only for the given resolutions—not asymptotically) and faster than FFTs (see §3).

- **Finite Difference Scheme:** In particular, we use an 8^{th} order central difference scheme to evaluate first-order partial derivatives for the gradient and divergence operators. To evaluate the partial derivative at a regular grid point, we require nine axis-aligned function evaluations f_{ijk}. We load the grid values f_{ijk} from global memory to a shared memory tile and then evaluate the finite difference stencil. The derivative evaluations in the x_1, x_2 and x_3 spatial dimensions are independent of each other. Our implementation is the same as the CUDA SDK finite difference code [41] except that our implementation works for general grid sizes and supports periodic boundary conditions.
- **FFT (Spectral Differentiation):** `CLAIRE` uses `AccFFT` [34,35], which supports MPI for both CPU and GPUs. Here, we just use `cuFFT` [62] as we focus on a single GPU implementation. When we use FFTs for gradient and divergence operations we compute 3D FFTs. This avoids an explicit transpose operation on the data and misaligned memory accesses. Additionally, 3D FFTs reduce the number of memory accesses of the spectral data from global device memory. For the gradient all partial derivatives can be computed with only a single read and three write operation per element (instead of $3 + 3$ as for one-dimensional FFTs). Similarly, the divergence operator only needs a single store operation after summing all partial derivatives.

3. KERNEL PERFORMANCE ANALYSIS. In this section, we evaluate the performance of interpolation (**IP**) and finite difference (**FD**) kernels. We calculate their arithmetic intensity (or simply ''*intensity*'') defined as the ratio of FLOPS (total floating point operations) to MOPS (total memory operations). We compare the kernel intensity to the device intensity. If the kernel intensity is less than the device intensity (peak floating point performance divided by peak device memory bandwidth), then the kernel is memory bound, otherwise it is compute bound. This is a simplification of the roofline model [81] since here we do not account for the cache hierarchy and latency effects. We also perform benchmark experiments to identify performance ceilings for our kernels.

As reference system for the CPU code, we used a two-socket Intel Skylake system. It is equipped with two Xeon Gold 5120 with a maximum frequency of 2.20 GHz and a maximum bandwidth of 107.30 GB s^{-1} with a TDP of 105 W per socket. We used a 32GB NVidia Tesla V100 with a memory bandwidth $B_{\max}$ of 900 GB s^{-1} and a TDP of 300 W for GPU experiments. The V100 is part of a two socket IBM Power9 system featuring NVLink as inter-device bus. Our implementation is in C++ and CUDA, and uses the PETSc library [9] for the Gauss–Newton–Krylov solvers.

3.1. Cubic Interpolation Kernel. Both cubic and linear IP are memory bound. The IP kernel has two main inputs: the target point coordinates ($3N$ floats), and the grid point scalar values (N floats). The output is the scalar field at the target points (N floats). Thus, the total MOPS is five floats (20 B) per target point. Formula (5) applies to both B-spline and Lagrange interpolation: the value at each target point depends on 64 regular grid values for cubic and 8 for trilinear interpolation, and these are not contiguous in memory.

Assuming an infinite amount of fast memory and ignoring latency cost, an analytic calculation of the FLOPS for each kernel gives the arithmetic intensity that shows that the kernels are memory bound. We overestimate the analytic intensity because we assume that all c_{ijk} values in (5) are loaded exactly once from device memory, which will typically not be the case, unless the memory accesses are fully coalesced. We evaluate performance using an effective bandwidth in GB/s defined as $\frac{(b_w+b_r)}{t\,x\,10^9}$, where b_r and b_w are the kernel loads/stores in bytes and t is the kernel total run time. We tuned the threadblock configuration to obtain optimal performance for the interpolation kernel. We used a one dimensional threadblock configuration with 256 threads for all our experiments. We perform two experiments for a

Table 2: *Experiment 2: Comparison of arithmetic ''intensity'' for two interpolations with $N = 256^3$ on an NVIDIA Tesla V100. For the analytic ''FLOPS'' value, we assume that each FPADD (add), FPMUL (multiply), FPSP (other ops like division) is one FLOP, and an FMA (multiply add) is two FLOPS. For GPU-TXTSPL, GPU-TXTLIN and GPU-TXTLAG, the FLOP count includes the operations required to compute the trilinear interpolation done internally by the texture unit. For the analytic ''MOPS'', we assume that each f_{ijk} value is loaded only once from the device memory. (Thus, all kernels have the same MOPS since the fact that we use linear versus cubic doesn't matter for this simple model.) The intensity value is computed as the ratio of FLOPS/MOPS. For the experimental "FLOPS", we make the same assumption as for the analytic "FLOPS", but here the FLOP count is obtained from the NVidia Visual Profiler. The experimental "MOPS" are also obtained from the visual profiler and are the sum of the total number of bytes read from and written to the GPU device memory by the L2 cache. GPU-TXTSPL⋆ corresponds to GPU-TXTSPL w/o prefilter.*

	Analytic			Experimental			
Kernel	**FLOPS**	**MOPS**	**intensity**	**GFLOPS**	**GMOPS**	**intensity**	**bound by**
PRE-FILTER	22	8	2.75	0.37	0.14	2.64	memory
GPU-TXTLIN	30	20	1.50	0.10	0.34	0.30	memory
GPU-LAG	221	20	11.05	3.66	1.55	2.36	memory
GPU-TXTLAG	482	20	24.10	3.00	0.34	8.94	memory
GPU-TXTSPL⋆	294	20	14.70	2.97	0.27	10.86	memory
NVIDIA Tesla V100				14 000GFLOPS/s	900GB/s	15.56	

Table 3: *Performance of the overall semi-Lagrangian transport using different interpolation kernels on the V100. We report runtimes (in seconds) for applying an LDDMM transformation on a real 3D brain MR image using a semi-Lagrangian scheme. We deform the brain image using a velocity field (generated by registering two images from a clinical dataset) forward in time, followed by deforming the resulting image backward in time. We then compare the original image to the resulting image and compute the relative mismatch between the two. CPU-LAG, GPU-LAG and GPU-TXTLAG have a relative error of* 5.3e−2 *and* 2.4e−2 *for* $N = 64^3$ *and* 256^3*, respectively. GPU-TXTSPL is* 2× *more accurate, and has a relative error* 2.5e−2 *and* 1.7e−2*, respectively. GPU-TXTLIN has a relative error of* 1.2e−1 *and* 5.5e−2*, respectively. We also report wall-clock time for two advection solves, which incurs 14 interpolation kernel calls in total. The corresponding effective global memory bandwidth is also reported. The run time and bandwidth reported for GPU-TXTSPL include the overhead of the pre-filter operation. The CPU Lagrange (CPU-LAG) interpolation kernel is executed on a single intel-skylake node with 24 MPI tasks.*

	CPU-LAG	GPU-LAG		GPU-TXTLAG		GPU-TXTSPL (w/pre-filter)		GPU-TXTLIN	
N	**time**	**time**	**BW**	**time**	**BW**	**time**	**BW**	**time**	**BW**
64^3	16	1.5	50	6.4e−1	115	6.7e−1	240	1.3e−1	552
128^3	124	1.1e1	54	4.0	146	2.9	442	8.3e−1	705
256^3	1000	8.4e1	56	3.5e1	136	2.2e1	461	6.0	790

localized and for a scattered target point distribution.

3.1.1. Experiment 1—Localized target points. As we discussed, each target point requires a set of c_{ijk} values. To isolate the memory issues related to streaming the target points, we conducted a run in which *all target points use the same 64 grid values* for interpolation. This ensures full reuse of regular grid values among targets and provides an upper limit for the performance of the kernel. We run this test on the GPU-LAG and GPU-TXTSPL kernels. The performance of GPU-TXTLAG is somewhere in between and we omitted it in these runs. In this model the MOPS change. We only read and write $4N$ floats, and read 64 grid values for all points. Since all thread blocks need to read these values, the number of total MOPS (in bytes) is equal to $4 * (4N + 64 * \#\text{threadblocks})$. We use this to estimate an upper performance bound. It is important to note here that the number of threadblocks only matters for a theoretical estimate without accounting for cache effects. In experimental runs, since all threadblocks are accessing the same set of 64 grid values, they will be cached. Hence, different threadblock configurations will not significantly affect the kernel performance, except for extremely small threadblocks where latency effects are dominant.

- **GPU-LAG Kernel (w/shared memory):** All CUDA thread-blocks load the same set of 64 c_{ijk} values from device memory and store them in the on-chip shared memory for reuse. All threads evaluate the result at their corresponding target points using the data available in shared memory and then apply (5). Using the MOPS estimate from above (and the observed timings), we achieve

Table 4: *Experiment 2: Runtime (in seconds) and error of different interpolation kernels on the NVIDIA Tesla V100. We report the relative interpolation error and the averaged run time for one kernel call in seconds. The relative interpolation error is given in the ℓ^2-norm with respect to an analytically known function. The evaluation is done on a grid with randomly perturbed grid points. The interpolated function is given by $(sin^2(8x_1) + sin^2(2x_2) + sin^2(4x_3))/3$. For this synthetic setup, the measured runtime t_{syn} (syn) is averaged over 100 interpolations. The faster variants GPU-TXTSPL and GPU-TXTLIN were also applied to the real data experiments shown in §4. For those, we also report the per-call duration t_{reg} for averaged over all Gauss-Newton iterations. The reported runtimes include all pre- and post-processing needed for the interpolation method.*

N	method	error	t_{syn}	t_{reg}
64^3	GPU-LAG	9.9e−3	1.2e−4	—
	GPU-TXTLAG	9.8e−3	7.5e−5	—
	GPU-TXTSPL	2.2e−3	1.1e−4	1.1e−4
	GPU-TXTLIN	2.6e−2	3.8e−5	2.7e−5
128^3	GPU-LAG	7.2e−4	7.4e−4	—
	GPU-TXTLAG	7.3e−4	4.1e−4	—
	GPU-TXTSPL	1.1e−4	3.6e−4	3.3e−4
	GPU-TXTLIN	6.8e−3	1.3e−4	1.4e−4
256^3	GPU-LAG	4.7e−5	5.2e−3	—
	GPU-TXTLAG	8.7e−5	3.0e−3	—
	GPU-TXTSPL	5.0e−5	2.3e−3	2.1e−3
	GPU-TXTLIN	1.7e−3	8.4e−4	1.0e−3

an effective bandwidth of 570 GB/s (63.3%$B_{\max}$).

- **GPU-TXTSPL Kernel:** To calculate the effective bandwidth, we assume that each of the 64 c_{ijk} are fetched by the texture exactly once from the device memory. Using this assumption (and the observed timings), the effective bandwidth for this method is 350 GB/s (39%$B_{\max}$). Note that the reported bandwidth here does not account for the prefilter operation. Also note that, in reality, textures cannot take significant advantage of the fact that the target points have exactly the same regular grid dependencies. As a result, there are more memory dependencies (than our MOPS estimate) and, thus, the observed performance drops—compared to the GPU-LAG kernel.

3.1.2. Experiment 2—Scattered target points. We consider a real distribution (generated via random perturbation of grid points or actual trajectory backward tracking) of target points (and switch to the original 20 B/point MOPS model). Here, unlike "Experiment 1", the implementation of GPU-LAG does not use shared memory to load target point dependencies. The implementation of GPU-LAG which uses shared memory to load target point dependencies for the scattered case is future work. However, the implementation of GPU-TXTSPL remains the same as in "Experiment 1".

The analytic observation that the interpolation is memory bound result is confirmed by measurements with the NVIDIA Visual Profiler summarized in Table 2.

For a random distribution of target points, GPU-TXTSPL achieves an effective global memory bandwidth of 335 GB/s (37.6%$B_{\max}$), which is nearly identical to "Experiment 1". Hence, GPU-TXTSPL is insensitive to target point dependencies. In contrast, GPU-LAGs performance drops by a factor of 10 to 56 GB/s because we are no longer making explicit use of shared memory to load and reuse the target point dependencies. Also note that, once we coupled the GPU-TXTSPL to the overall semi-Lagrangian scheme in Table 3, the effective bandwidth increases to 461 GB/s, which is slightly over 50% relative to the peak bandwidth. Finally, in Table 4, we compare the accuracy and time of the four different methods. The differences in accuracy are somewhat significant only in lower resolutions. Note that we get different accuracy results for real brain MR images in Table 3. This is expected since the cubic spline interpolation of GPU-TXTSPL gives better interpolation accuracy than third order Lagrange polynomials used in CPU-LAG or GPU-LAG in cases where the image resolution is not sufficiently high relative to the highest frequency in the image. For the synthetic low frequency image lower used in Table 4, Lagrange polynomials to perform better for higher image resolutions. Here, GPU-LAG gives more accurate results than GPU-TXTSPL for a 256^3 resolution. We compare our new GPU implementation to the original MPI based CPU version of CLAIRE [56]; the CPU version of CLAIRE does not support OpenMP.

As a byproduct, this analysis also addresses to some extent the following question: would it make sense to reorder (say in Morton order) the target and grid points in order to achieve better locality (but possible sacrifice texture memory)? As we show, an ideal ordering would result in 570 GB/s; we observe

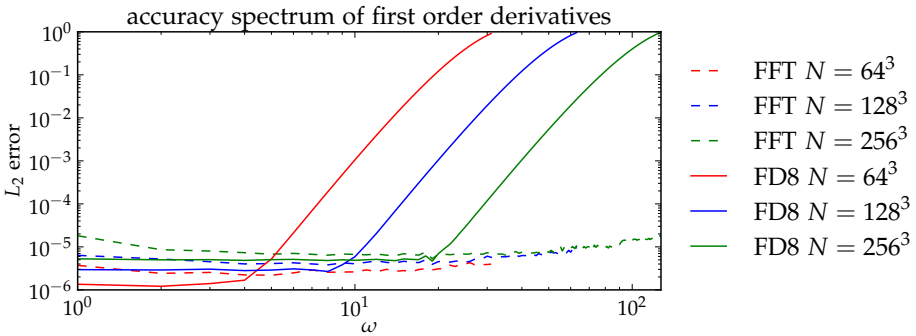

Fig. 2: *Accuracy of first order differential operators (gradient and divergence) using FFT and 8^{th} order finite differences on a Nvidia Tesla V100 for different problem sizes. We report the L_2 error of our operators. The error is measured using the computed partial derivative in x_3-direction of the function $\sin(\omega x_3) + \cos(\omega x_3)$ compared to the analytical derivative. The error is plotted over the frequency up to the Nyquist frequency. Finite differences are more accurate for low frequency modes and have an increasing error for higher modes. By replacing FFTs with finite differences, we trade faster computation (due to a higher data locality and a reduced algorithmic complexity) against lower accuracy for high frequency modes.*

about 460 GB/s for GPU-TXTSPL and conclude that our implementation is nearly optimal.

3.2. Finite Difference Kernel. In our implementation, each CUDA thread block evaluates the derivatives for a 2D tile of data. We refer to the points contained in this tile as *inner points*. To evaluate the derivatives at the edge of a tile, we load a set of neighboring points known as *halo points*. We load the set of inner points and halo points from device memory to a 2D shared memory tile, evaluate the derivatives, and store the result back to shared memory. The inner points of one thread-block are halo-points of the adjacent thread-block and are loaded twice. We quantify this experimentally. We first repeat the FLOPS-MOPS experiment for the FD kernel and observe that the kernel is memory bound.

We compare the bandwidth performance of our general kernel to the parent SDK example. The SDK code works only for a fixed grid size $N = 64^3$ and a 9-point stencil. CUDA SDK reports an effective bandwidth of 310 GB/s whereas our implementation achieves 212 GB/s. The reported bandwidth includes the cost of loading halo points. Both values are much smaller than B_{max} because the grid size is not large enough to hide latency. Unlike the SDK example, the CUDA threads on the boundary of the domain load halo points from global memory instead of shared memory. The observed performance drops due to the thread divergence caused by reading out-of-bound halo points. For large N, as we show later, this overhead is greatly reduced as a direct consequence of decreased latency caused by higher occupancy.

We perform a zero-overhead memory copy, i.e., copy within the HBM2 device memory to put an absolute upper bound on the performance of our implementation. We load each element of an array of size $N = 256^3$ from the global device memory and store it in another array. The peak performance we get for this copy routine is 780 GB/s. To quantify the halo points load overhead, we perform another experiment. Each thread-block loads its inner points and halo points into a 2D shared memory tile and copies only the inner points back to the output array. The effective bandwidth for this benchmark is 766 GB/s. The reported bandwidth includes the cost of loading halo points. We only lose 1.8% of the memory bandwidth in comparison to the zero-overhead memory copy experiment. This indicates that the overhead due to loading of out-of-bound halo points gets smaller as the kernel occupancy increases. We verify our claims by profiling the kernels using the NVIDIA Visual Profiler. For the smaller grid size of 64^3, the kernel is bound by instruction and memory latency, for the larger grids (128^3 and 256^3) by memory bandwidth.

4. IMAGE REGISTRATION RESULTS. We evaluate the overall algorithm using four 3D MRI images. We study convergence behavior, time-to-solution, and registration accuracy for several algorithmic variants of computational kernels available in our new GPU implementation of the CPU software `CLAIRE`. We compare with two popular GPU packages for LDDMM registration. The purpose of this section is to show that (a) our new (mixed-precision) GPU implementation yields the same registration accuracy as

Table 5: *Runtime (in seconds) of first order differential operators (gradient and divergence) using FFT and 8^{th} order finite differences (FD8) on a NVIDIA Tesla V100 for different problem sizes. We report the runtime in s per kernel call averaged over the whole registration run from experiments shown in §4 including all pre- and post-processing needed.*

N	Operator	FFT	FD8
64^3	grad	1.7e−4	3.6e−5
	div	1.7e−4	3.9e−5
128^3	grad	6.0e−4	1.4e−4
	div	5.7e−4	1.6e−4
256^3	grad	4.1e−3	9.4e−4
	div	3.8e−3	1.2e−3

Table 6: *Variants of combinations of computational kernels and the respective tag used in this work. IP stands for interpolation and FD8 for finite difference operators of 8^{th} order.*

Tag	Variant
cpu-fft-cubic	FP32, CPU, FFT, cubic IP
gpu-fft-cubic	FP32, GPU, FFT, cubic IP
gpu-fd8-cubic	FP32, GPU, FD8, cubic IP
gpu-fd8-linear	FP32, GPU, FD8, trilinear IP

our CPU implementation of `CLAIRE` [56] and (b) to compare our method against GPU implementations of other groups.

4.1. Data and Setup.

4.1.1. Images. We report results for the `NIREP` (Non-Rigid Image Registration Evaluation Project) data, a commonly used data set to evaluate the performance of deformable registration algorithms [22]. NIREP consists of 16 rigidly aligned T1-weighted magnetic resonance neuroimaging MR scans (`na01`–`na16`) of different individuals. The original resolution is $256 \times 300 \times 256$ voxels. Each scan is annotated with a label map that identifies 32 gray matter regions [22]. We select four scans from this data set, `na01` as reference image and `na02`, `na03`, and `na10` as template images, respectively. The initial DICE coefficient (spatial overlap index) for the union of the gray matter regions of the template images versus the reference image is 0.55, 0.50 and 0.48, respectively. A perfect matching would correspond to a value of 1.00. Currently, we only support image sizes $N_1 N_2 N_3$ dividable by 256. We resampled the data sets to grid sizes of 64^3, 128^3, 256^3, and 384^3, using a linear and a nearest-neighbor interpolation model for the image data and the label maps, respectively.

4.1.2. Numerical & Floating Point Accuracy Parameters. Unless specified otherwise, we use the default solver parameters from [54] for the Gauss–Newton–Krylov solver. For regularization we use the default of `CLAIRE`, H^1-div—an H^1-seminorm with an additional penalty on the divergence of the velocity. In all runs, we use a target regularization parameter $\beta = 5\text{e}{-4}$ selected based on experiments reported in [56]. We execute the proposed solver with a parameter continuation scheme for the regularization parameter β. This scheme is describe in detail in [51]. We set the parameter for the penalty for the divergence of v to 1e−4.

- **Convergence Criteria:** As a stopping criterion for the optimizer, we use a tolerance of 5e−2 for the relative reduced gradient (3) together with a maximal number of Gauss–Newton iterations of 50 (never reached in our experiments). We use a superlinear forcing sequence for the Newton-Krylov solver (inexact Newton solve; see [24, 28] for details) and set the maximum number of iterations for the PCG (used to compute the search direction; see §2) to 500 (never reached in our experiments). We globalize our Gauss–Newton–Krylov method using an Armijo line search [61].
- **Interpolation:** We consider different interpolation methods to evaluate the value of variables at off grid locations within our semi-Lagrangian scheme (see §2). In particular, we select either a linear or a cubic interpolation scheme. For cubic interpolation, we use GPU-TXTSPL as proposed in §2.3.
- **First Order Derivatives:** For the calculation of first order derivatives, we compare the FFT-based scheme and the 8^{th} order finite difference (**FD8**) scheme as proposed in §2.
- **Floating Point Accuracy:** Our new implementation uses single precision (**FP32**). For validation, we compare against results achieved with the `CLAIRE` CPU implementation in single precision. We summarize our settings in Table 6.

4.1.3. Performance Metrics. We report two groups of metrics: To assess computational performance, we report runtimes. To assess accuracy of the results, we report the relative mismatch $\|m(\bullet, 1) - m_1\|_2 / \|m_1 - m_0\|_2$ of the template image $m_0(\boldsymbol{x})$, the reference image $m_1(\boldsymbol{x})$, and the transformed tem-

plate image $m(\boldsymbol{x}, 1)$ given by the forward problem (1b) as well as the DICE coefficient (overlap) between the union of the gray matter labels associated with the data sets. This enables an assessment of how well anatomical structures identified by expert observers are aligned after registration. For a perfect matching the value is 1.00[§]. To measure the quality of the computed deformation map, we report min, mean and max values of the determinant of the deformation gradient $\det \boldsymbol{F}$, $\boldsymbol{F} \in \mathbb{R}^{3,3}$. The mapping is locally non-diffeomorphic if the determinant of the deformation gradient changes sign or is zero. In general, if $\det \boldsymbol{F}$ is either very small (but still positive) or very big, the LDDMM mapping is of poor quality. In our case, $\det \boldsymbol{F}$ is between 0.5 and 10, which indicates excellent registration quality.

To assess the (rate of) convergence of our solver, we report the relative gradient norm $\|\boldsymbol{g}\|_{\text{rel}} := \|\boldsymbol{g}^\star\|_2 / \|\boldsymbol{g}^0\|_2$, where $\boldsymbol{g}^\star$ is the gradient of the optimization problem after convergence and $\boldsymbol{g}^0$ is the gradient for the initial guess $\boldsymbol{v} = \boldsymbol{0}$. We also report the number of iterations for the Newton-Krylov solver and the total number of Hessian matvecs (application of the Hessian to a vector; the smaller the better; see §2).

4.2. Results. Next, we report results for our improved implementation of `CLAIRE`. We use the same experimental setup as for the kernel performance analysis in §3.

4.2.1. Performance Analysis of the Proposed Method. ***Purpose:*** We study the performance of different variants of our solver, i.e., for different combinations of computational kernels.

Results: The results for the experiments described above are reported in Table 7 for image sizes of 64^3, 128^3, 256^3, and 384^3, respectively. The breakdown of the execution time with respect to the individual kernels is shown in Figure 3 and Figure 4. Figure 3 compares runtimes between the baseline CPU implementation with the equivalent GPU implementation using FFT for first order derivatives and cubic interpolation for the semi-Lagrangian scheme. We compare different GPU implementations in Figure 4 (for `na02`). The maximum allocated memory on the GPU during the experiments was 0.60 GB, 1.30 GB, 6.10 GB, and 20.00 GB for image sizes of 64^3, 128^3, 256^3, and 384^3, respectively. The maximum allocated memory on the host CPU was below 2 GB for all GPU experiments and only used for management and IO purposes.

Observations: The critical result is that we can accurately solve 3D image registration problems for clinically relevant sizes (256^3) on a single GPU in less than 10 seconds (Run #28, Run #32 and Run #36 in Table 7) for the variant gpu-fd8-linear. The gpu-fd8-cubic approximation is almost as fast while resulting in lower reduced gradient and similarity than gpu-fd8-linear. We also found that the iteration counts, registration quality and number of Hessian matvecs remains almost constant as we switch to lower accuracy regimes. The values for the DICE, the relative mismatch between the deformed template image and the reference image, and the Gauss-Newton iteration counts are almost identical. We observe slight differences in the number of Hessian matvecs between implementations, with fewer matvecs typically observed for gpu-fd8-linear. For all implementations we reach the set tolerance of 5e$-$2 for the relative reduction of the gradient.

All implementations produce well-behaved determinants of the deformation gradients. The highest DICE score is achieved for `na02` (0.86, Run #25 and Run #28). For gpu-fd8-linear, we see an increase in the maximum determinant of the deformation gradient, indicating a slightly more irregular mapping. For example, for Run #12 or Run #32 in Table 7, the maximum of the determinant of the deformation gradient increases from 7.54 to 10.52 (14%) and from 7.18 to 7.92 (11%). The speedup between the baseline method cpu-fft-cubic and gpu-fd8-linear is 8–11 for 64^3, 16–18 for 128^3, and 23–25 for 256^3. The gpu-fd8-cubic variant also performs very well with similar run times and slightly better $\det \boldsymbol{F}$.

For the considered test problems with image sizes 64^3, 128^3, and 256^3, the number of Gauss–Newton iterations remains constant per resolution level with a minimum of 12 and a maximum of 18 Gauss–Newton iterations. The number of Hessian matvecs increases up to a factor of two as we change resolution levels, with a minimum of 42 (Run #8) and a maximum of 104 (Run #34 and Run #35). There are several reasons for the increase in the number of matvecs. First, we can resolve finer details in the velocity and

[§]The DICE coefficient is a metric that has been widely adopted by the registration community to assess registration accuracy. We provide a more detailed study in [56]. We note that DICE and mismatch values do not provide a complete picture about registration accuracy. Other metrics include the Haussdorff distance between the contours of label maps or landmark errors (an example for a database that considers landmarks to evaluate registration performance is DIRLAB; see `www.dir-lab.com`). We note that the focus of the manuscript is on computational performance and not registration accuracy. The accuracy results included in this study serve as a baseline to compare our improved solver to our past work [56].

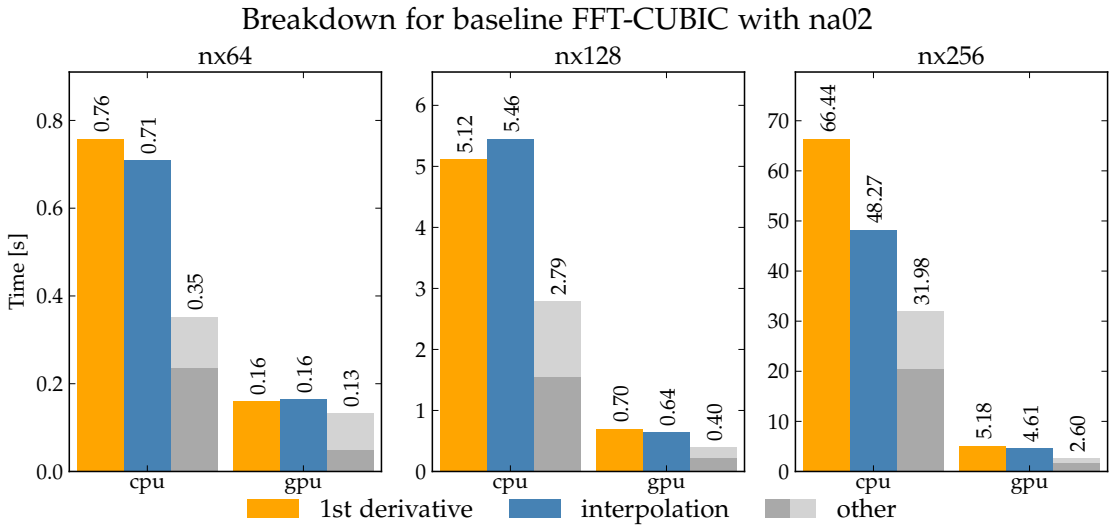

Fig. 3: *Runtime breakdown for the main kernels of the proposed method and the baseline CPU implementation in* `CLAIRE` *(first order derivatives via FFT, cubic interpolation). The dark gray parts indicate the FFTs used for the regularization terms. We consider the registration of the* `na02` *image to the* `na01` *image at a resolution of* 64^3*,* 128^3*, and* 256^3*, respectively. Note that the speed-up when moving to the GPU is a combination of algorithmic improvements and the higher memory bandwidth.*

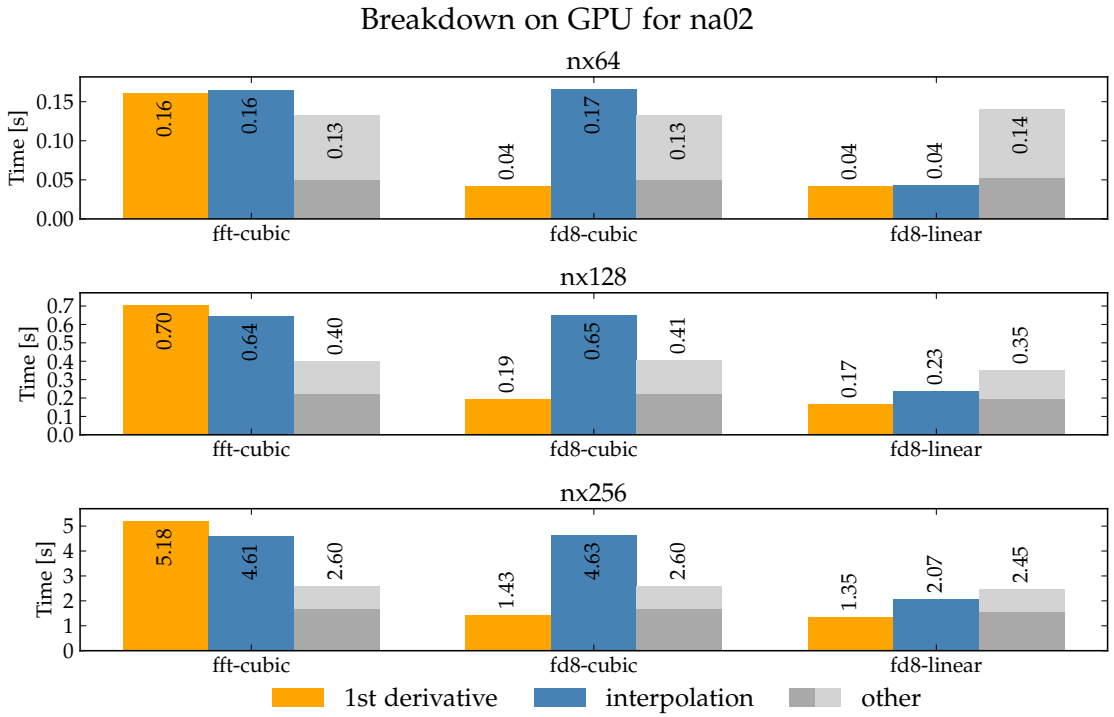

Fig. 4: *Runtime breakdown for the main kernels of the proposed method for all GPU implementations (first order derivatives via FFT or FD8, cubic or linear interpolation). The dark gray parts indicate the contribution of higher order operators in spectral space to the overall execution time of the solver. We consider the registration of* `na02` *to* `na01` *at a resolution of* 64^3*,* 128^3*, and* 256^3*, respectively.*

Table 7: *Results for registration runs using the proposed method. The experiments for the baseline (fft-cubic) implementation are highlighted in gray. We report for each dataset (from left to right): minimum, mean and maximum value of the determinant of the deformation gradient* $\det \boldsymbol{F}$*, the DICE coefficient before and after registration, the relative mismatch, the relative* ℓ^2*-norm of the gradient, the number of Gauss–Newton iterations until convergence (#iter), the number of Hessian matvecs (#MV), and the total runtime in seconds. We report results for data grid sizes of* 64^3*,* 128^3*,* 256^3*, and* 384^3*.*

				det $\boldsymbol{F}$		DICE						
run	variant	data	min	mean	max	before	after	mism.	$\|g\|_{\text{rel}}$	#iter	#MV	time
$N = 64^3$												
#1	cpu-fft-cubic	na02	0.64	1.01	4.14	0.56	0.62	1.1e−2	7.7e−3	12	58	1.82
#2	gpu-fft-cubic		0.63	1.01	3.99		0.62	1.1e−2	9.0e−3	12	58	0.46
#3	gpu-fd8-cubic		0.63	1.01	3.96		0.62	1.1e−2	8.9e−3	12	58	0.34
#4	gpu-fd8-linear		0.64	1.01	5.06		0.63	1.7e−2	1.1e−2	12	54	0.23
#5	cpu-fft-cubic	na03	0.63	1.01	8.50	0.50	0.61	8.7e−3	8.0e−3	13	64	1.97
#6	gpu-fft-cubic		0.63	1.02	8.04		0.61	8.6e−3	8.3e−3	13	63	0.54
#7	gpu-fd8-cubic		0.63	1.02	8.01		0.61	8.6e−3	8.2e−3	13	63	0.39
#8	gpu-fd8-linear		0.59	1.02	9.06		0.61	1.4e−2	1.6e−2	12	42	0.18
#9	cpu-fft-cubic	na10	0.56	1.03	7.88	0.48	0.68	7.2e−3	1.2e−2	12	48	1.61
#10	gpu-fft-cubic		0.56	1.03	7.48		0.68	7.1e−3	1.3e−2	12	48	0.41
#11	gpu-fd8-cubic		0.56	1.03	7.54		0.68	7.1e−3	1.3e−2	12	48	0.31
#12	gpu-fd8-linear		0.59	1.03	10.52		0.68	9.6e−3	1.3e−2	12	44	0.18
$N = 128^3$												
#13	cpu-fft-cubic	na02	0.54	1.01	3.99	0.55	0.79	1.7e−2	1.8e−2	14	70	13.36
#14	gpu-fft-cubic		0.54	1.01	3.92		0.79	1.7e−2	1.8e−2	14	73	1.75
#15	gpu-fd8-cubic		0.54	1.01	3.92		0.79	1.7e−2	1.8e−2	14	73	1.25
#16	gpu-fd8-linear		0.58	1.01	4.79		0.80	2.0e−2	1.7e−2	12	63	0.75
#17	cpu-fft-cubic	na03	0.48	1.02	8.10	0.51	0.79	1.5e−2	1.8e−2	15	77	14.62
#18	gpu-fft-cubic		0.48	1.02	7.93		0.79	1.6e−2	1.9e−2	15	78	1.86
#19	gpu-fd8-cubic		0.48	1.02	7.94		0.79	1.6e−2	1.9e−2	15	78	1.33
#20	gpu-fd8-linear		0.48	1.02	10.14		0.79	1.6e−2	1.7e−2	13	68	0.81
#21	cpu-fft-cubic	na10	0.55	1.04	8.78	0.48	0.78	1.2e−2	1.7e−2	15	84	15.93
#22	gpu-fft-cubic		0.57	1.04	8.86		0.78	1.2e−2	1.6e−2	14	82	1.92
#23	gpu-fd8-cubic		0.57	1.04	8.84		0.78	1.2e−2	1.6e−2	14	82	1.36
#24	gpu-fd8-linear		0.58	1.03	9.98		0.78	1.3e−2	1.7e−2	15	82	0.96
$N = 256^3$												
#25	cpu-fft-cubic	na02	0.41	1.01	3.62	0.55	0.86	2.9e−2	3.7e−2	14	81	146.69
#26	gpu-fft-cubic		0.41	1.01	3.57		0.85	3.0e−2	3.8e−2	14	81	12.38
#27	gpu-fd8-cubic		0.41	1.01	3.57		0.85	3.0e−2	3.7e−2	14	81	8.66
#28	gpu-fd8-linear		0.43	1.01	3.83		0.86	2.7e−2	3.1e−2	14	75	5.87
#29	cpu-fft-cubic	na03	0.47	1.02	6.83	0.50	0.83	2.8e−2	3.6e−2	17	95	169.46
#30	gpu-fft-cubic		0.47	1.00	6.81		0.83	2.9e−2	3.8e−2	17	99	15.09
#31	gpu-fd8-cubic		0.47	1.00	6.79		0.83	2.9e−2	3.7e−2	17	98	10.44
#32	gpu-fd8-linear		0.48	1.00	7.51		0.83	2.6e−2	3.1e−2	17	93	7.22
#33	cpu-fft-cubic	na10	0.58	1.04	7.18	0.48	0.82	2.1e−2	3.5e−2	18	103	184.78
#34	gpu-fft-cubic		0.58	1.01	7.08		0.82	2.2e−2	3.8e−2	18	104	16.05
#35	gpu-fd8-cubic		0.58	1.01	7.18		0.82	2.1e−2	3.4e−2	18	104	11.05
#36	gpu-fd8-linear		0.61	1.01	7.92		0.82	2.0e−2	2.9e−2	17	94	7.29
$N = 384^3$												
#37	gpu-fft-cubic	na02	0.37	0.59	3.78	0.55	0.86	2.6e−2	3.4e−2	16	152	72.82
#38	gpu-fd8-cubic		0.40	0.59	3.55		0.85	3.4e−2	4.3e−2	15	91	31.59
#39	gpu-fd8-linear		0.41	0.59	3.71		0.85	3.1e−2	3.8e−2	15	85	21.69
#40	gpu-fft-cubic	na03	0.46	0.60	7.52	0.50	0.84	2.7e−2	4.3e−2	22	201	96.59
#41	gpu-fd8-cubic		0.44	0.60	6.63		0.83	3.3e−2	4.1e−2	18	112	38.72
#42	gpu-fd8-linear		0.45	0.60	6.99		0.83	3.0e−2	3.8e−2	17	98	24.90
#43	gpu-fft-cubic	na10	0.59	0.61	7.98	0.48	0.81	2.2e−2	3.8e−2	25	233	111.55
#44	gpu-fd8-cubic		0.55	0.61	7.20		0.80	2.6e−2	4.2e−2	20	117	40.82
#45	gpu-fd8-linear		0.58	0.61	7.49		0.81	2.4e−2	3.7e−2	18	104	26.35

the images, which results in more complicated deformation patterns and by that longer runtimes. Second, we use a regularization parameter of $\beta = 1\text{e}{-4}$ for all resolutions, to be consistent. Given the observed change of information content, one should in general adapt the regularization parameter according to the resolution level in real application cases. Our experiments for the image size 384^3 have a higher variation in the number of Newton steps and matvecs. Notice that we use relative tolerances in our algorithm (as opposed to a fixed number of iterations). Consequently, we expect that differences in numerical accuracy and changes in the resolution (more frequencies can be resolved) have an effect on the number of iterations required until convergence.

Table 8: *Registration performance for* `PyCA` *[65],* `deformetrica` *[26], and the proposed method executed on a V100 and a P100 for three neuroimaging data sets (grid size:* 256^3*). We were not able to execute* `deformetrica` *on a V100 due to issues with the installation. We expect the speedup to be* $2\times$ *(in accordance with the observations we have made for the other software packages);* `deformetrica` *would still be slower than* `PyCA`*. The solvers are executed with default parameters. We only alter the maximum number of iterations. The defaults are 300 iterations per level for* `PyCA` *(using a multi-resolution strategy with two levels) and 50 iterations for* `deformetrica`*. We execute the proposed method with a parameter continuation scheme for the regularization parameter (the default method used in the CPU version of* `CLAIRE`*); we report results for the proposed method corresponding to Run #28, Run #32, and Run #36, in Table 7. We report iterations per level ("100,50" for PyCA means 100 iterations on the first level and 50 iterations on the second level), the relative mismatch after registration (mism.), and the runtime (in seconds). We see that* **our GPU implementation of** `CLAIRE` **is about an order of magnitude more accurate (mismatch) and, at the same time, up to** $30\times$ **faster** *(fastest result for* `PyCA` *on a V100). The runs #3/14/19 for CLAIRE correspond to the runs #28/32/36 in Table 7 (same experiment).*

	`PyCA` [65]					`deformetrica` [26]					proposed method				
data	run	**#iter**	**mism.**	**time**		run	**#iter**	**mism.**	**time**		run	**#iter**	**mism.**	**time**	
				P100	V100				P100	V100				P100	V100
na02	#1	100,50	4.2e−1	1.9e1	1.1e1	#2	10	4.8e−1	1.4e2	–	#3	14	2.7e−2	9.0	5.9
	#4	100,100	3.4e−1	3.4e1	1.8e1	#5	25	4.0e−1	2.5e2	–					
	#6	300,300	2.4e−1	1.0e2	5.3e1	#7	50	3.5e−1	4.4e2	–					
	#8	500,500	2.1e−1	1.7e2	8.9e1	#9	100	3.2e−1	8.2e2	–					
	#10	1000,1000	1.9e−1	3.4e2	1.8e2	#11	300	2.8e−1	2.4e3	–					
na03	#12	300,300	2.5e−1	1.0e2	5.4e1	#13	50	3.1e−1	8.4e2	–	#14	17	2.6e−2	1.1e1	7.22
	#15	500,500	2.5e−1	1.7e2	9.0e1	#16	300	2.5e−1	2.4e3	–					
na10	#17	300,300	2.5e−1	1.0e2	5.4e1	#18	50	3.0e−1	8.3e2	–	#19	17	2.0e−2	1.1e1	7.29
	#20	500,500	2.2e−1	1.7e2	9.0e1	#21	300	2.5e−1	2.4e3	–					

Looking at the breakdown of the CPU baseline in Figure 3, we observe that its runtime is dominated by the application of first-order derivatives and interpolation operations. If we add the execution time of high-order spectral derivatives (bars in dark gray in the "other" category), we see that almost all runtime goes to differentiation and interpolation. We spend $66.44\,\text{s} + 48.27\,\text{s} = 114.71\,\text{s}$ out of $146.69\,\text{s}$ (78% of the runtime) on computing first-order derivatives and evaluating the interpolation kernel (right plot in Figure 3; CPU; grid size: 256^3). We observe a similar behavior for the GPU implementation. For example, we spend $5.18\,\text{s} + 4.61 = 9.79\,\text{s}$ of $12.39\,\text{s}$ (80% of the runtime) on these kernels (right plot in Figure 3; GPU; grid size: 256^3). Consequently, we expect a significant reduction in the runtime of our GPU accelerated version of `CLAIRE` compared to the CPU implementation of `CLAIRE` if we can speed up the evaluation of these kernels. This is precisely what we observe in Table 7.

The breakdown in Figure 4 provides additional insight. We can see that the execution time for the first-order derivatives reduces from $5.18\,\text{s}$ to $1.43\,\text{s}$ (speed up of ≈ 3.5) when switching from spectral methods to an optimized FD8 approximation (Figure 4, bottom block; yellow bars for the 1st derivative). If we switch from cubic to linear interpolation, we see a reduction in the execution time from $4.63\,\text{s}$ to $2.07\,\text{s}$ (speed up of ≈ 2). The runtime of the other operations remains almost constant. So, overall we went from a solver that is bound by the through-put of first order derivatives and interpolation operations, to a solver that is now bound by the execution time of high-order derivatives.

4.2.2. Comparison with other GPU Implementations. ***Purpose:*** We compare the performance of our new, improved GPU version of `CLAIRE` to other GPU implementations of LDDMM-type methods.

Setup: We compare the performance of the proposed method to publicly available GPU implementations of LDDMM approaches that have recently been considered by several groups [15, 16, 32, 82, 83]. The first software package is `PyCA` [65]. `PyCA` uses gradient descent for optimization. Its interface is written in `python`. The libraries and modules used for the compilation of `PyCA` and `deformetrica` are listed in the citations [65] and [26], respectively. The second software package is `deformetrica` [26]; `deformetrica` uses a limited-memory Broyden-Fletcher-Goldfarb-Shanno method for optimization. The gradient of the optimization problem is computed based on automatic differentiation [16]. We execute both registration packages for the three neuroimaging data sets we used to assess the performance of the proposed method (`na02`, `na03`, and `na10` as template images and `na02` as reference image). The runs are performed using the full resolution of our data (256^3). We slightly modify scripts available in the repositories of these two

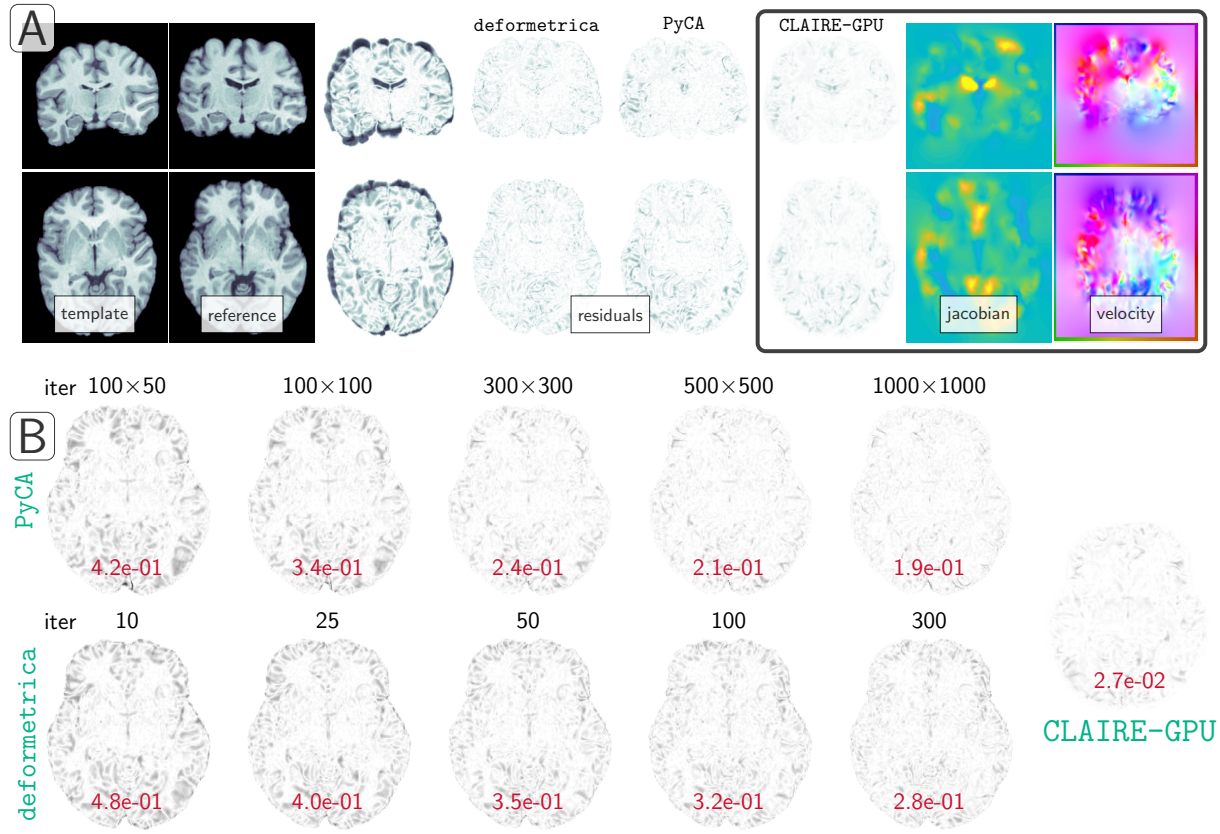

Fig. 5: *Registration results. (A) We visualize the registration results for image* `na03` *to* `na01`*. Top row: Coronal view. Bottom row: Axial view. We show (from left to right) the template image $m_0(\boldsymbol{x})$, the reference image $m_1(\boldsymbol{x})$, the mismatch before registration, the mismatch after registration (for* `deformetrica`*,* `PyCA`*, and our improved implementation of* `CLAIRE`*, respectively), and the determinant of the deformation gradient as well as the scalar map for the orientation of the computed velocity vectors. The color bar for the values for the determinant of the deformation gradient is limited to $[0,2]$ with blue/green/yellow corresponding to $0/\approx 1/\geq 2$ (values ≥ 2 are set to 2 for visualization purposes). The computed deformation map is locally diffeomorphic as judged by the determinant of the deformation gradient (up to numerical accuracy; min:* $4.8\text{e}{-1}$*; max:* 7.5*; mean:* 1.0*). The results reported in this figure are the best-performing runs of those reported in Table 8 for each software. (B) Registration results for the image* `na02` *to* `na01`*. We show results for different iteration settings for* `PyCA` *(top row) and* `deformetrica` *(bottom row). Results for* `CLAIRE` *are shown on the right. The numbers in red are the obtained mismatch values for the respective settings.*

software packages to execute these runs (using the default parameters available in the scripts). We vary the number of iterations for `PyCA` and `deformetrica` to make sure we (i) do not terminate early, (ii) do not perform unnecessary iterations, and (iii) (possibly) generate the most accurate results attainable for the default settings (subject to a reasonable iteration count/runtime). We compare these results to our fastest implementation of the proposed method (gpu-fd8-linear; see results reported in Table 7).

Results: In Table 8, we report runtimes and relative mismatch values for all methods. We compare these results to the best performance achieved for the proposed method for the experiments reported in Table 7 (Run #28, Run #32, and Run #36). We showcase exemplary registration results as well as the imaging data to be registered in Figure 5 and Figure 6. In Figure 5, we show (from left to right; coronal views: top row; axial views: bottom row) the reference image, the template image, the initial mismatch before registration, and the mismatch after registration for `deformetrica`, `PyCA`, and the proposed method, respectively. We also provide point wise maps for the determinant of the deformation gradient and a map of the orientation of the velocity field for the proposed method. Figure 6 shows image data overlaid with the 32 gray matter labels, contours of the union of these labels overlaid onto the reference

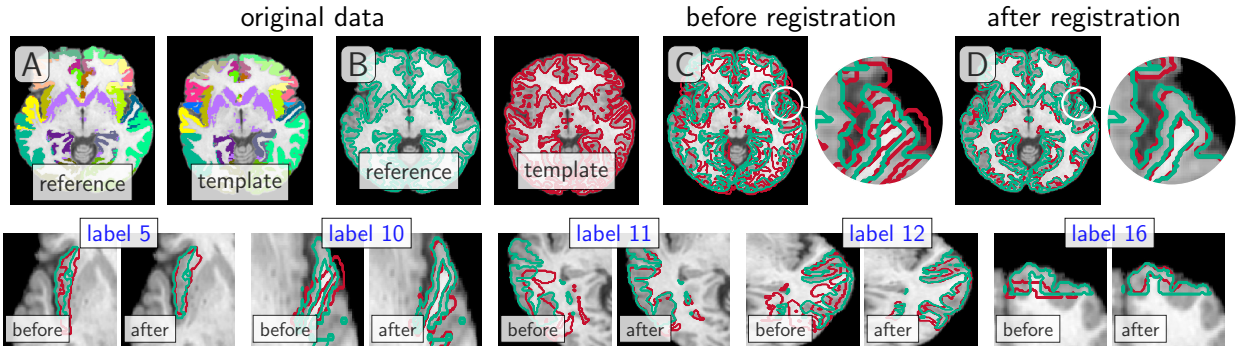

Fig. 6: *Registration results for CLAIRE. Top row: In (A) we show the image data overlaid with the 32 gray matter labels (datasets* `na03` *and* `na01`*). In (B) we show the contours of the union of these labels overlaid onto the reference and template image, respectively. In (C) we show the two contours overlaid onto the reference image before registration and in (D) after registration (red contour: template image; green contour: reference image). The circles show a closeup. In the bottom row we show contours before and after registration (left and right, respectively) for five of the 32 gray matter labels visualized in (A) (top row).*

and template image, respectively, and overlaid contours before and after registration. We have reported extensive experiments in our past work [56]. In the present work, we are only interested in demonstrating that switching to our GPU implementation (with mixed-precision accuracy) does not deteriorate the results we get.

Observations: The most important observation is that the proposed method delivers a mismatch that is about one order of magnitude better than `PyCA` and `deformetrica` for the default settings, with more than one order of magnitude decrease in runtime. For the peak performance of the proposed method, we see that our approach is 30× faster with a 6× better mismatch (comparison of Run #9 in Table 8 with the best result obtained for the proposed method; Run #28 in Table 7). Note that PyCA uses first order methods for optimization. Therefore, each iteration is much cheaper. In CLAIRE, we use second order information (Newton). Our method makes more progress per iteration but also requires more work; we need to iteratively invert the Hessian matrix to compute the search direction (i.e., solve a linear system). Thus, time per iteration is not a good measure on its own. We need to compare how much work (runtime) it requires to reach a certain accuracy (mismatch between the data). For the proposed method, we use convergence criteria based on the relative reduction of the gradient norm. The two other methods considered here terminate when they reach the set upper bound for the iterations. The best result is obtained for `PyCA` with 1,000 gradient descent steps per level. If we would further increase the runtime (number of iterations) we would probably obtain results that are closer to those obtained for the proposed method (in terms of mismatch). We observe a linear increase in the runtime with respect to the number of iterations for both considered methods. We note that the differences in accuracy between the methods can be attributed to various factors (e.g., different optimization methods; convergence criteria; different regularization weights and norms; different parameters for the algorithm; or different mathematical formulations). The findings reported here are in accordance with timings reported in the literature [15,82,83]. Figure 6 shows that not only the DICE coefficients indicate good quality od registration results, but also the label contours match very well after registration.

5. CONCLUSIONS. We presented algorithms, analysis, and numerical experiments for an improved GPU implementation of the CPU registration package `CLAIRE` for large deformation diffeomorphic image registration. This problem is resource constrained because clinical workflows require high-throughput, with one or more registration tasks per node. Typical image sizes fit into the memory of a single GPU in our optimized implementation. MPI parallelism cannot help since multiple registration tasks can take place in an embarrassingly parallel way. Therefore, our focus is on single node and, in particular, on single device optimizations. We demonstrated over 10× speedup over state-of-the-art GPU implementations of LDDMM registration. We showed that the problem is memory-bound but it utilizes over 50% of the peak bandwidth and has sufficient arithmetic intensity to deliver multi TFLOP/s performance.

REFERENCES

[1] *Cuda toolkit dcoumentation*. 6

[2] V. Arsigny, O. Commowick, X. Pennec, and N. Ayache, *A Log-Euclidean framework for statistics on diffeomorphisms*, in Proc Medical Image Computing and Computer-Assisted Intervention, vol. LNCS 4190, 2006, pp. 924–931. 2

[3] J. Ashburner, *A fast diffeomorphic image registration algorithm*, NeuroImage, 38 (2007), pp. 95–113. 2

[4] J. Ashburner and K. J. Friston, *Diffeomorphic registration using geodesic shooting and Gauss-Newton optimisation*, NeuroImage, 55 (2011), pp. 954–967. 2

[5] B. B. Avants, C. L. Epstein, M. Brossman, and J. C. Gee, *Symmetric diffeomorphic image registration with cross-correlation: Evaluating automated labeling of elderly and neurodegenerative brain*, Medical Image Analysis, 12 (2008), pp. 26–41. 2

[6] B. B. Avants, N. J. Tustison, G. Song, P. A. Cook, A. Klein, and J. C. Gee, *A reproducible evaluation of ANTs similarity metric performance in brain image registration*, NeuroImage, 54 (2011), pp. 2033–2044. 2

[7] R. Azencott, R. Glowinski, J. He, A. Jajoo, Y. Li, A. Martynenko, R. H. W. Hoppe, S. Benzekry, and S. H. Little, *Diffeomorphic matching and dynamic deformable surfaces in 3D medical imaging*, Computational Methods in Applied Mathematics, 10 (2010), pp. 235–274. 2

[8] G. Balakrishnan, A. Zhao, M. R. Sabuncu, J. Guttag, and A. V. Dalca, *VoxelMorph: A learning framework for deformable medical image registration*, IEEE Transactions on Medical Imaging, (2019). (in press) DOI: 10.1109/TMI.2019.2897538. 3

[9] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory, 2016. 7

[10] V. Barbu and G. Marinoschi, *An optimal control approach to the optical flow problem*, Systems & Control Letters, 87 (2016), pp. 1–9. 3

[11] M. F. Beg, M. I. Miller, A. Trouvé, and L. Younes, *Computing large deformation metric mappings via geodesic flows of diffeomorphisms*, International Journal of Computer Vision, 61 (2005), pp. 139–157. 1, 2, 3

[12] G. Biros and O. Ghattas, *Parallel lagrange–newton–krylov–schur methods for pde-constrained optimization. part i: The krylov–schur solver*, SIAM Journal on Scientific Computing, 27 (2005), pp. 687–713. 4

[13] ———, *Parallel lagrange–newton–krylov–schur methods for pde-constrained optimization. part ii: The lagrange–newton solver and its application to optimal control of steady viscous flows*, SIAM Journal on Scientific Computing, 27 (2005), pp. 714–739. 4

[14] P. T. Boggs and J. W. Tolle, *Sequential quadratic programming*, Acta Numerica, 4 (1995), pp. 1–51. 4

[15] A. Bone, O. Colliot, and S. Durrleman, *Learning distributions of shape trajectories from longitudinal datasets: A hierarchical model on a manifold of diffeomorphisms*, arXiv e-prints, (2019). 2, 15, 17

[16] A. Bone, M. Louis, B. Martin, and S. Durrleman, *Deformetrica 4: An open-source software for statistical shape analysis*, in Proc International Workshop on Shape in Medical Imaging, vol. LNCS 11167, 2018, pp. 3–13. 2, 15

[17] A. Borzì, K. Ito, and K. Kunisch, *Optimal control formulation for determining optical flow*, SIAM Journal on Scientific Computing, 24 (2002), pp. 818–847. 3

[18] D. Budelmann, L. Koenig, N. Papenberg, and J. Lellmann, *Fully-deformable 3D image registration in two seconds*, in Bildverarbeitung für die Medizin, 2019, pp. 302–307. 2

[19] M. Burger, J. Modersitzki, and L. Ruthotto, *A hyperelastic regularization energy for image registration*, SIAM Journal on Scientific Computing, 35 (2013), pp. B132–B148. 2

[20] F. Champagnat and Y. Le Sant, *Efficient cubic B-spline image interpolation on a GPU*, Journal of Graphics Tools, 16 (2012), pp. 218–232. 6

[21] K. Chen and D. A. Lorenz, *Image sequence interpolation using optimal control*, Journal of Mathematical Imaging and Vision, 41 (2011), pp. 222–238. 3

[22] G. E. Christensen, X. Geng, J. G. Kuhl, J. Bruss, T. J. Grabowski, I. A. Pirwani, M. W. Vannier, J. S. Allen, and H. Damasio, *Introduction to the non-rigid image registration evaluation project*, in Proc Biomedical Image Registration, vol. LNCS 4057, 2006, pp. 128–135. 11

[23] N. Courty and P. Hellier, *Accelerating 3D non-rigid registration using graphics hardware*, International Journal of Image and Graphics, 8 (2008), pp. 81–98. 2

[24] R. S. Dembo, S. C. Eisenstat, and T. Steihaug, *Inexact Newton methods*, SIAM Journal on Numerical Analysis, 19 (1982), pp. 400–408. 11

[25] P. Dupuis, U. Gernander, and M. I. Miller, *Variational problems on flows of diffeomorphisms for image matching*, Quarterly of Applied Mathematics, 56 (1998), pp. 587–600. 2

[26] A. S. Durrleman, A. Bone, M. Louis, B. Martin, P. Gori, A. Routier, M. Bacci, A. Fougier, B. Charlier, J. Glaunes, J. Fishbaugh, M. Prastawa, M. Diaz, and C. Doucet, *deformetrica [commit: v4.0.0-390-ged9c1f9; libraries: python3.6; cuda9.2.88]*, 2019. 2, 15

[27] S. Durrleman, M. Prastawa, N. Charon, J. R. Korenberg, S. Joshi, G. Gerig, and A. Trouve, *Morphometry of anatomical shape complexes with dense deformations and sparse parameters*, NeuroImage, 101 (2014), pp. 35–49. 2

[28] S. C. Eisentat and H. F. Walker, *Choosing the forcing terms in an inexact Newton method*, SIAM Journal on Scientific Computing, 17 (1996), pp. 16–32. 11

[29] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, *Medical image processing on the GPU–past, present and future*, Medical Image Analysis, 17 (2013), pp. 1073–1094. 2

[30] N. D. Ellingwood, Y. Yin, M. Smith, and C.-L. Lin, *Efficient methods for implementation of multi-level nonrigid mass-preserving image registration on GPUs and multi-threaded CPUs*, Computer Methods and Programs in Biomedicine, 127 (2016), pp. 290–300. 2

[31] B. Fischer and J. Modersitzki, *Ill-posed medicine – an introduction to image registration*, Inverse Problems, 24 (2008), pp. 1–16. 1, 2

[32] J. Fishbaugh, S. Durrleman, M. Prastawa, and G. Gerig, *Geodesic shape regression with multiple geometries and sparse parameters*, Medical Image Analysis, 39 (2017), pp. 1–17. 2, 15

[33] O. Fluck, C. Vetter, W. Wein, A. Kamen, B. Preim, and R. Westermann, *A survey of medical image registration on graphics hardware*, Computer Methods and Programs in Biomedicine, 104 (2011), pp. e45–e57. 2

[34] A. Gholami and G. Biros, *AccFFT*, 2017. 7

[35] A. Gholami and G. Biros, *AccFFT home page*, 2017. 7

[36] A. Gholami, A. Mang, K. Scheufele, C. Davatzikos, M. Mehl, and G. Biros, *A framework for scalable biophysics-based image analysis*, in Proc ACM/IEEE Conference on Supercomputing, 2017, pp. 1–13. 2, 3, 5, 7

[37] D. Grzech, L. Folgoc, M. P. Heinrich, B. Khanal, J. Moll, J. A. Schnabel, B. Glocker, and B. Kainz, *FastReg: Fast non-rigid registration via accelerated optimisation on the manifold of diffeomorphisms*, arXiv e-prints, (2019). 2

[38] X. Gu, H. Pan, Y. Liang, R. Castillo, D. Yang, D. Choi, E. Castillo, A. Majumdar, T. Guerrero, and S. B. Jiang, *Implementation and evaluation of various demons deformable image registration algorithms on a GPU*, Physics in Medicine and Biology, 55 (2009), pp. 207–219. 2

[39] L. Ha, J. Krüger, S. Joshi, and C. T. Silva, *Multiscale unbiased diffeomorphic atlas construction on multi-GPUs*, in CPU Computing Gems Emerald Edition, Elsevier Inc, 2011, ch. 48, pp. 771–791. 2

[40] L. K. Ha, J. Krüger, P. T. Fletcher, S. Joshi, and C. T. Silva, *Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units*, in Proc Eurographics Conference on Parallel Grphics and Visualization, 2009, pp. 41–48. 2

[41] M. Harris, *Nvidia developer blog*, 2019. 6, 7

[42] M. Hernandez, M. N. Bossa, and S. Olmos, *Registration of anatomical images using paths of diffeomorphisms parameterized with stationary vector field flows*, International Journal of Computer Vision, 85 (2009), pp. 291–306. 2

[43] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards, 49 (1952), pp. 409–436. 4

[44] M. Hinze, R. Pinnau, M. Ulbrich, and S. Ulbrich, *Optimization with PDE constraints*, Springer, Berlin, DE, 2009. 4

[45] S. Joshi, B. Davis, M. Jornier, and G. Gerig, *Unbiased diffeomorphic atlas construction for computational anatomy*, NeuroImage, 23 (2005), pp. S151–S160. 2

[46] S. Klein, M. Staring, K. Murphy, M. A. Viergever, and J. P. W. Pluim, *ELASTIX: A tollbox for intensity-based medical image registration*, Medical Imaging, IEEE Transactions on, 29 (2010), pp. 196–205. 2

[47] L. Koenig, J. Ruehaak, A. Derksen, and J. Lellmann, *A matrix-free approach to parallel and memory-efficient deformable image registration*, SIAM Journal on Scientific Computing, 40 (2018), pp. B858–B888. 2

[48] J. Krebs, H. Delingette, B. Mailhé, N. Ayache, and T. Mansi, *Learning a probabilistic model for diffeomorphic registration*, IEEE Transactions on Medical Imaging, (2019). (in press) DOI: 10.1109/TMI.2019.2897112. 3

[49] M. Lorenzi, N. Ayache, G. B. Frisoni, and X. Pennec, *LCC-demons: a robust and accurate symmetric diffeomorphic registration algorithm*, NeuroImage, 81 (2013), pp. 470–483. 2

[50] M. Lorenzi and X. Pennec, *Geodesics, parallel transport and one-parameter subgroups for diffeomorphic image registration*, International Journal of Computer Vision, 105 (2013), pp. 111–127. 2

[51] A. Mang and G. Biros, *An inexact Newton–Krylov algorithm for constrained diffeomorphic image registration*, SIAM Journal on Imaging Sciences, 8 (2015), pp. 1030–1069. 1, 11

[52] ———, *Constrained H^1-regularization schemes for diffeomorphic image registration*, SIAM Journal on Imaging Sciences, 9 (2016), pp. 1154–1194. 1, 2, 3

[53] ———, *A Semi-Lagrangian two-level preconditioned Newton–Krylov solver for constrained diffeomorphic image registration*, SIAM Journal on Scientific Computing, 39 (2017), pp. B1064–B1101. 3, 4

[54] A. Mang and G. Biros, *Constrained large deformation diffeomorphic image registration (CLAIRE)*, 2019. [Commit: v0.07-131-gbb7619e]. 1, 2, 11

[55] A. Mang, A. Gholami, and G. Biros, *Distributed-memory large-deformation diffeomorphic 3D image registration*, in Proc ACM/IEEE Conference on Supercomputing, 2016. 1, 2, 3, 4, 5

[56] A. Mang, A. Gholami, C. Davatzikos, and G. Biros, *CLAIRE: a distributed-memory solver for constrained large deformation diffeomorphic image registration*, SIAM Journal on Scientific Computing, 41 (2019), pp. C548–C584. 1, 2, 3, 4, 5, 9, 11, 12, 17

[57] M. I. Miller and L. Younes, *Group actions, homeomorphism, and matching: A general framework*, International Journal of Computer Vision, 41 (2001), pp. 61–81. 2

[58] M. Modat, G. R. Ridgway, Z. A. Taylor, M. Lehmann, J. Barnes, D. J. Hawkes, N. C. Fox, and S. Ourselin, *Fast free-form deformation using graphics processing units*, Computer Methods and Programs in Biomedicine, 98 (2010), pp. 278–284. 2

[59] J. Modersitzki, *Numerical methods for image registration*, Oxford University Press, New York, 2004. 1, 2, 5

[60] ———, *FAIR: Flexible algorithms for image registration*, SIAM, Philadelphia, Pennsylvania, US, 2009. 1, 2, 5

[61] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer, New York, New York, US, 2006. 11

[62] Nvidia, *CUDA CUFFT Library*, 2007. 7

[63] A. Paszke, S. Gross, S. Chintala, and G. Chanan, *Tensors and dynamic neural networks in python with strong GPU acceleration*, 2019. 2

[64] T. Polzin, M. Niethammer, M. P. Heinrich, H. Handels, and J. Modersitzki, *Memory efficient LDDMM for lung CT*, in Proc Medical Image Computing and Computer-Assisted Intervention, vol. LNCS 9902, 2016, pp. 28–36. 2

[65] J. S. Preston, *Python for computational anatomy*, 2019. [Commit: v0.01-434-gf31ab43; Libraries: ITK4.13.2; boost1.69; FFTW3.3.6-pl2; python2.7; CUDA9.2.88]. 2, 15

[66] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes, *Non-rigid registration using free-form deformations: Application to breast MR images*, Medical Imaging, IEEE Transactions on, 18 (1999), pp. 712–721. 2

[67] D. Ruijters, *GPU accelerated pre-filtered cubic B-spline interpolation using CUDA*, 2019. 6

[68] D. Ruijters, B. ter Haar Romeny, and P. Suetens, *Efficient gpu-based texture interpolation using uniform b-splines*, Journal of Graphics Tools, 13 (2008), pp. 61–69. 6, 7

[69] D. Ruijters and P. Thévenaz, *GPU prefilter for accurate cubic B-spline interpolation*, The Computer Journal, 55 (2012), pp. 15–20. 6

[70] J. Shackleford, N. Kandasamy, and G. Sharp, *On developing B-spline registration algorithms for multi-core processors*, Physics in Medicine and Biology, 55 (2010), pp. 6329–6351. 2

[71] D. P. Shamonin, E. E. Bron, B. P. F. Lelieveldt, M. Smits, S. Klein, and M. Staring, *Fast parallel image registration on CPU and GPU for diagnostic classification of Alzheimer's disease*, Frontiers in Neuroinformatics, 7 (2014), pp. 1–15. 2
[72] R. Shams, P. Sadeghi, R. A. Kennedy, and R. I. Hartley, *A survey of medical image registration on multicore and the GPU*, Signal Processing Magazine, IEEE, 27 (2010), pp. 50–60. 2
[73] C. Sigg and M. Hadwiger, *Fast third-order texture filtering*, (2005), pp. 313–329. 6
[74] S. Sommer, *Accelerating multi-scale flows for LDDKBM diffeomorphic registration*, in Proc IEEE International Conference on Computer Visions Workshops, 2011, pp. 499–505. 2
[75] A. Sotiras, C. Davatzikos, and N. Paragios, *Deformable medical image registration: A survey*, Medical Imaging, IEEE Transactions on, 32 (2013), pp. 1153–1190. 1, 2
[76] A. Trouvé, *Diffeomorphism groups and pattern matching in image analysis*, International Journal of Computer Vision, 28 (1998), pp. 213–221. 2
[77] P. Valero-Lara, *A GPU approach for accelerating 3D deformable registration (DARTEL) on brain biomedical images*, in Proc European MPI Users' Group Meeting, 2013, pp. 187–192. 2
[78] P. Valero-Lara, *Multi-GPU acceleration of DARTEL (early detection of Alzheimer)*, in Proc IEEE International Conference on Cluster Computing, 2014, pp. 346–354. 2
[79] T. Vercauteren, X. Pennec, A. Perchant, and N. Ayache, *Diffeomorphic demons: Efficient non-parametric image registration*, NeuroImage, 45 (2009), pp. S61–S72. 2
[80] F.-X. Vialard, L. Risser, D. Rueckert, and C. J. Cotter, *Diffeomorphic 3D image registration via geodesic shooting using an efficient adjoint calculation*, International Journal of Computer Vision, 97 (2012), pp. 229–241. 3
[81] S. Williams, A. Waterman, and D. Patterson, *Roofline: An insightful visual performance model for multicore architectures*, Commun. ACM, 52 (2009), pp. 65–76. 7
[82] X. Yang, R. Kwitt, and M. Niethammer, *Fast predictive image registration*, in Proc International Workshop on Deep Learning in Medical Image Analysis, 48-57, ed., vol. LNCS 10008, 2016, pp. 48–57. 3, 15, 17
[83] X. Yang, R. Kwitt, M. Styner, and M. Niethammer, *Quicksilver: Fast predictive image registration—A deep learning approach*, NeuroImage, 158 (2017), pp. 378–396. 2, 3, 15, 17
[84] L. Younes, *Jacobi fields in groups of diffeomorphisms and applications*, Quarterly of Applied Mathematics, 650 (2007), pp. 113–134. 2
[85] ———, *Shapes and diffeomorphisms*, Springer, 2010. 1, 2, 3
[86] L. Younes, F. Arrate, and M. I. Miller, *Evolutions equations in computational anatomy*, NeuroImage, 45 (2009), pp. S40–S50. 2
[87] M. Zhang and P. Fletcher, *Finite-dimensional lie algebras for fast diffeomorphic image registration*, in Proc Information Processing in Medical Imaging, Springer International Publishing, 2015, pp. 249–260. 2
[88] M. Zhang and P. T. Fletcher, *Fast diffeomorphic image registration via Fourier-approximated Lie algebras*, International Journal of Computer Vision, (2018), pp. 1–13. 2