SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz

The AGILE library for image reconstruction in biomedical sciences using graphics card hardware acceleration

M. Freiberger F. Knoll K. Bredies
H. Scharfetter R. Stollberger

SFB-Report No. 2012-009 April 2012

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

Das Land Steiermark

Stadt GRAZ Wissenschaft

The AGILE library for image reconstruction in biomedical sciences using graphics card hardware acceleration

Manuel Freiberger, Florian Knoll, Kristian Bredies, Hermann Scharfetter and Rudolf Stollberger

Abstract

Fast image reconstruction is a critical requirement for an imaging modality to be adopted in the field of clinical and pre-clinical sciences. While programs become faster due to more powerful hardware, at the same time data size increases and the need for advanced—and often computational more demanding—reconstruction algorithms arises. A cheap way to achieve a major speed-up is to utilize modern graphics hardware capable of executing algorithms in a massively parallel manner. In this article, the open source library *AGILE* designed for image reconstruction in biomedical sciences is introduced. Its modular, object-oriented and templated design eases the integration of the library into user code. Furthermore, applications from the field of magnetic resonance imaging and fluorescence tomography are presented. As demonstrated, a speed-up of factor 10–30 is achievable with commodity graphics hardware for different reconstruction tasks.

I. Introduction

Image reconstruction from different kinds of imaging modalities is a major research area in biomedical sciences. The recent development shows a trend towards more advanced and computationally more demanding reconstruction algorithms. On the one hand, this progression is due to the introduction of new imaging techniques like fluorescence tomography, for example. On the other hand, the development of fast measurement techniques for well established modalities like the various sub-sampling methods for magnetic resonance imaging, for example, pose additional requirements on the reconstruction algorithms and hardware.

Another trend in high-performance computing is the use of modern graphics processing units (GPU) for various applications. Although the computation capability of graphics hardware is limited, it can run a large number of threads in parallel. This makes a GPU very well suited for most matrix-vector and vector-vector operations as they mostly rely on the multiplication and summation of elements.

The development of code suitable for GPU was eased by the release of NVIDIA's CUDA toolkit. This is an extension to standard C code by directives for starting threads on the GPU. However, not every code running on an ordinary PC is suitable for graphics hardware. First of all, the GPU uses a massive amount of threads which means that only parallel algorithms will result in a speed-up. Serial algorithms should still be run on the host core to avoid overhead like the memory transfer from or to the graphics card. Second, care must be taken when accessing elements in GPU memory. Although this requirement is mitigated a bit when using latest graphics hardware it still applies that when writing efficient code a block of threads should access an aligned block of memory.

This publication presents the *AGILE* library, an environment for linear and non-linear image reconstruction using GPU acceleration, which was designed for different computationally demanding imaging methodologies in biomedical sciences, especially magnetic resonance imaging (MRI) and (fluorescence) diffuse optical tomography.

II. Library design

The *AGILE* library is a collection of basic matrix-vector and vector-vector operations as well as algorithms often needed for image reconstruction with focus on biomedical sciences. The code of the library is available as open source under the GNU General Public License version 3 (GPLv3) at http://www.imt.tugraz.at/research/agile-gpu-image-reconstruction-library/.

This work was supported by the Austrian Science Fund (FWF) under grant SFB F32 (SFB "Mathematical Optimization and Applications in Biomedical Sciences").

M. Freiberger is with the Institute of Medical Engineering, Graz University of Technology, Kronesgasse 5, A-8010 Graz, Austria, (email: manuel.freiberger@tugraz.at)

F. Knoll is with the Institute of Medical Engineering, Graz University of Technology, Kronesgasse 5, A-8010 Graz, Austria, (email: florian.knoll@tugraz.at)

K. Bredies is with the Institute for Mathematics and Scientific Computing, University of Graz, Heinrichstrasse 36, A-8010 Graz, Austria, (email: kristian.bredies@uni-graz.at).

H. Scharfetter is with the Institute of Medical Engineering, Graz University of Technology, Kronesgasse 5, A-8010 Graz, Austria, (email: hermann.scharfetter@tugraz.at)

R. Stollberger is with the Institute of Medical Engineering, Graz University of Technology, Kronesgasse 5, A-8010 Graz, Austria, (email: rudolf.stollberger@tugraz.at)

It can also be found in the open source section of MRI unbound, the collaborative Forum for MRI data acquisition and image reconstruction of the ISMRM (International Society of Magnetic Resonance in Medicine http://www.ismrm.org/mri_unbound/.

AGILE builds on NVIDIA's CUDA library [1] for the communication with the graphics card. The library itself is written in C++ and makes heavy use of the object-oriented design pattern and template classes both of which facilitate the rapid development of applications. Additionally, the class design follows established standards like the standard template library (STL) [2].

A. Data structures

AGILE provides a convenient way to access matrix and vector data structures on the GPU. The most basic entity is the `GPUVector`, which is a dynamically resizeable vector in graphics card memory. For compatibility, it provides most methods of the `std::vector` class from the STL, for example, `assign`, `capacity`, `resize`, `size` etc.

The exception to this rule is that there are no methods to access a single vector element like `at` or `operator[]`, for example. The reason is that reading or writing single elements does not fit well into the approach of achieving faster algorithms by a massive amount of threads running in parallel. Low-level access to a `GPUVector`'s elements is possible via the method `data` which returns a pointer to the first vector element which could be achieved via `&operator[0]` when operating on a `std::vector`.

Additionally, *AGILE* provides means to ease the exchange of data between the host and the GPU memory. For the `GPUVector` these are methods as `assignFromHost` and its counterpart `copyToHost` which copy a vector from host (i.e. CPU) memory to the graphics card's memory and vice versa.

Similar to the STL, `GPUVector` is templated and can be instantiated for several data types as real- and complex-valued floating point numbers in either single- or double-precision.

While there is only a single data type `GPUVector` for a vector in the graphics card's memory, there are currently three data structures to store matrices. In order to store a dense matrix, `GPUMatrix` is the naive choice. It stores the matrix data in a linear layout with column-major orientation, which is the conventional MATLAB and Fortran memory layout but contrary to the C/C++ layout. Again, the `GPUMatrix` is templated to be usable with different data types. Under the hood, `GPUMatrix` uses `CUBLAS` [3] for matrix-matrix- (e.g. sum and difference of matrices) and matrix-vector-operations (e.g. the matrix-vector multiplication). Again, convenience functions exist to ease the transfer of a matrix between CPU and graphics card memory.

An alternative to `GPUMatrix` is the hand-crafted class `GPUMatrixPitched`. As the name suggests, it pads the matrix rows with zeros in order to align the data to 32 byte boundaries. This padding is transparent to users of *AGILE*, who can copy the matrix as if it were stored in a linear fashion similar to `GPUMatrix`. Only developers must take care, when low-level access to the elements is required.

The padding allows for more efficient matrix-matrix and matrix-vector operations as will be demonstrated in section III. A further advantage of this custom implementation is that the matrix-vector and vector-vector operations are all templated for real and complex data types. If operations with mixed types are performed—which could be the inner product of a real and complex valued vector, for example—the result is promoted to the longer data type which would be a complex for the previous example. Thus, for the user there is no difference from a coding point of view when multiplying a real or complex matrix with a real or complex vector. The only restriction is that the object to which the operation's result will be assigned has to have a suitable type. For the matrix-vector product that would require that if either the matrix or the input vector is complex, the output vector has to be complex valued too.

Finally, the third matrix data structure is `GPUCSMatrix`, which can be used to store sparse matrices in a compressed storage format which can be compressed row-storage or compressed column-storage. This is a very popular format in the context of finite element simulations. The memory layout used in *AGILE* is depicted in Figure 1(c). In order to efficiently access the data from different threads in parallel, the elements are stored interleaved, i.e. the array holds the first element of every row, followed by the second element and so on. If necessary, rows are padded with zero elements such that all have the same amount of non-zero entries. To avoid wasting too much space with this padding, the rows are processed in blocks and the maximum number of non-zero elements is computed per block only. This layout has also been used in the work [4].

Again, the type used for storing elements in `GPUCSMatrix` is specified by a template argument. Also the "compression dimension" (i.e. compressed row vs. compressed column) can be adapted with a template parameter. As it is the case for `GPUMatrixPitched`, a sparse matrix and a vector do not have to be of the same type for e.g. multiplication as long as the output vector can store the result.

B. Algorithms

The *AGILE* library already provides a collection of solvers for linear equation systems. Currently these include the minimum residual method (MINRES), the conjugate gradient method (CG), the preconditioned CG (PCG), the complex orthogonal CG (COCG), the generalized minimal residual method (GMRES) and the least squares QR decomposition method (LSQR).

These algorithms are implemented on a high level whose execution is controlled by the CPU. Only modules carrying out the real work (e.g. matrix vector multiplications or the computation of inner products) is done by the graphics card. This allows a

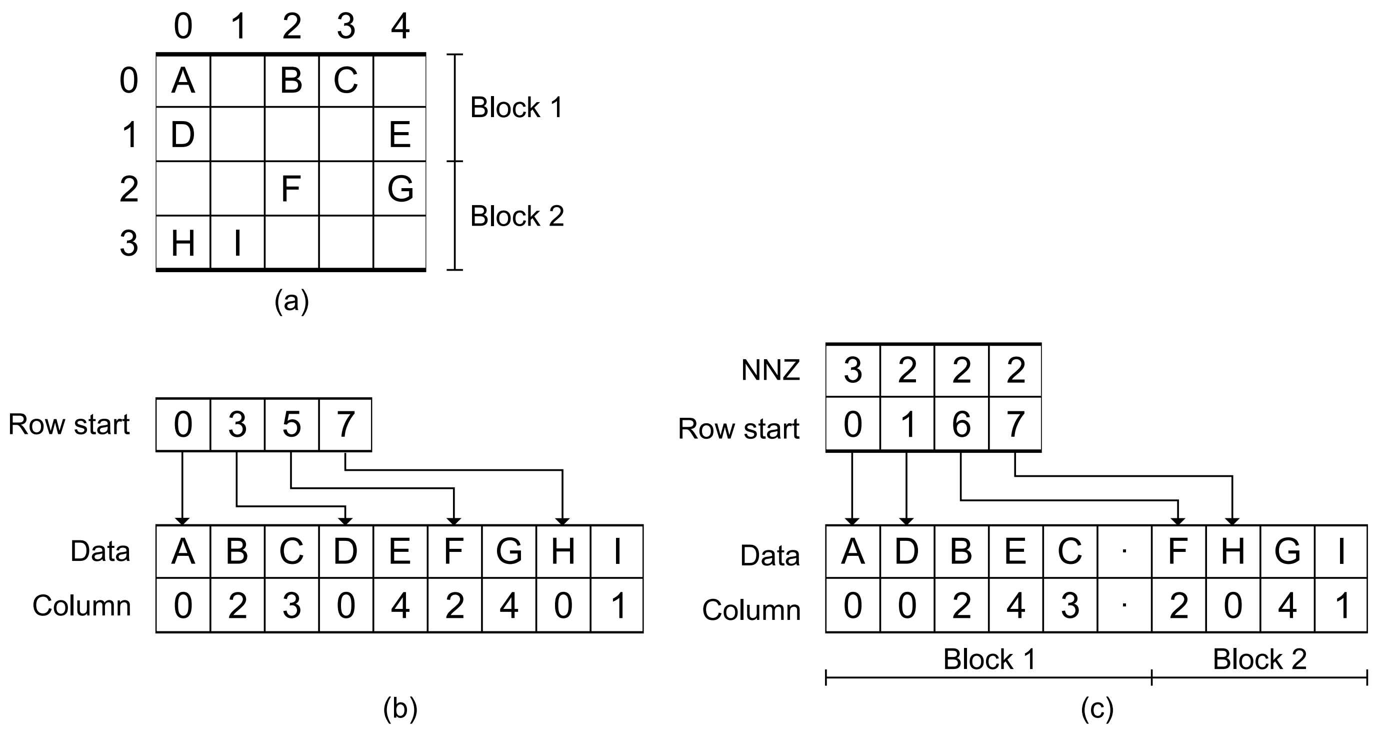

Fig. 1. Memory layout for a sparse matrix shown in (a) in compressed row storage format (CRS). The conventional CRS format is given in (b). It holds the linearized data elements, the corresponding column indices as well as a vector which gives the index of the first data element of every row in the linearized storage. In (c) the adapted CRS format used for `GPUCSMatrix` in *AGILE* is depicted. Data elements are stored interleaved and an additional vector holding the number of non-zero entries per row (NNZ) is needed.

rapid implementation of new algorithms as only time-critical parts have to be tailored for the GPU. Furthermore, one can run the same algorithm on the CPU and the GPU, which is convenient during debugging.

For maximum code re-usability, the matrix-vector and vector-vector operations can be wrapped into command classes which provide a consistent interface to the caller. This interface resembles the mathematical notation as close as possible. For example, the matrix vector product $y \leftarrow Ax$ would be written as

```
// wrap the GPU matrix A in an operator
GPUMatrix<float> A(...);
ForwardMatrix<communicator, GPUMatrix<float> > A_op(com, A);
// compute y = A*x
A_op(x, y);
```

`ForwardMatrix` is a class which stores a reference to a communicator object needed internally to "talk" to the graphics card and a reference to the matrix with which a vector shall be multiplied. As shown above, the resultant object `A_op` can be called directly to perform a matrix-vector multiplication. However, this object is much more powerful as it can be passed to one of the solvers mentioned before. By subclassing `ForwardMatrix`, the matrix-vector operation used in e.g. the PCG method can be adapted to the users needs very rapidly. A usage case that already has been used in *AGILE*, is the multiplication with a matrix A which is much too large to fit into the memory of current commodity graphics hardware. By creating a subclass which calculates the matrix-vector product in a block-wise manner and handles the transfer of the matrix data to the GPU on its own, the computation of the result is hidden from the caller. Thus, all algorithms can immediately operate on the large matrix A without adapting any of their sources.

Furthermore, this approach provides some "syntactic sugar". The function `adjoint()` can be used to create the adjoint of a given operator. As an example, the multiplication with the hermitian matrix $y \leftarrow A^H x$ can be written in an intuitive way as `adjoint(A_op)(x,y)`. The call to `adjoint(A_op)` simply returns a new object which stores a reference to the original matrix but its `operator()` computes $A^H x$ instead of Ax.

It is important to note that all algorithms mentioned at the beginning of this section adhere to the command class interface themselves such that a CG object can be used as input for another algorithm. This allows to create chains of operators which eases the implementation of complex preconditioners or Schur-complement systems, where in every outer PCG iteration a matrix expression has to be inverted with another PCG method, for example.

III. Comparison of AGILE matrix products to CUBLAS

As mentioned in section II-A, two different matrix storage formats are available in *AGILE*. The first is `GPUMatrix`, which is a convenience wrapper around CUBLAS. The second is `GPUMatrixPitched`, which pads the matrix rows with zeros to provide faster element access at the expense of additional memory requirements. Performance tests between these two implementations were executed for matrix-scalar ($y = \alpha \cdot A$), matrix-vector ($y = Ax$) and hermitian matrix-vector multiplication ($y = A^H x$, matrix A is transposed and the conjugate complex is taken). Every test was performed with three test matrices A where: i) the number of rows was equal to the number of columns ($R = C = n$), ii) the number of rows was much higher than the number of columns ($R = 10n$, $C = n$) and iii) the number of columns was much higher than the number of rows ($R = n$, $C = 10n$). For each $n = 1 \ldots 1000$, the test was executed 2000 times and the execution times were averaged. Results for the data type `std::complex<float>`, which is also used predominantly in the applications described in sections IV-A and IV-B, are presented here. The main trend of these results can also be found for the other data types `float`, `double` and `std::complex<double>`, and the corresponding tests are included in the library's source code.

Figure 2 shows the relative computation times of the corresponding multiplications for `GPUMatrix` and `GPUMatrixPitched`. The performance tests, as well as all other computations in sections IV-A and IV-B were carried out on a conventional desktop PC (Intel Core 2 DUO E6600 2.40 GHz, 2 GB RAM) using an NVIDIA GTX480 GPU with 1536 MB memory. *AGILE* was compiled with release 3.0 of the NVCC compiler. Computations with the hand-crafted class `GPUMatrixPitched` were faster for all matrix-scalar multiplications and in the cases $R = C$ and $C = 10 \cdot R$ for matrix-vector multiplications. In contrast, for hermitian matrix-vector computations, with the exception of small matrix sizes, the `GPUMatrix` using CUBLAS was computationally more efficient. This can be explained by the fact that CUBLAS stores matrices in column-major orientation which makes access to consecutive elements in the same column—which is required for operations using the transposed matrix—more cache efficient. As `GPUMatrixPitched` uses the C/C++ memory layout, access in the same matrix row is faster making this type more suitable for ordinary matrix-vector multiplications. Additionally, the padding of rows with zero elements cares for an aligned memory access (thread i always reads from a memory location aligned at i) which leads to an additional speed-up for this matrix type.

IV. Example applications

A. Iterative image reconstruction of undersampled MRI data from multiple coils

Increasing imaging speed of MRI has always been a major research area ever since the first experiments in the 1970s, and improvements in this field have contributed significantly to pave the way for routine clinical application. The main reason for the lengthy data acquisition is that only a limited number of data points can be encoded in one MR signal and the measurement has to be repeated for many times. This means that MR data is acquired sequentially (line by line) in frequency (k)-space which is a fundamental difference compared to other imaging modalities, which always acquire complete images in one measurement step. In the past, research to accelerate MR imaging speed was mainly directed toward faster collection of data. However, in current MR systems, these developments have reached a state where further advances are not limited by technical challenges, but by fundamental physiological limits due to patient safety. One way to circumvent this problem is to increase imaging speed by reducing the number of data points that are necessary to reconstruct an image with a defined resolution. However, as this approach violates the Nyquist-Shannon sampling theorem, aliasing artifacts are introduced which have to be eliminated during the image reconstruction process.

By reformulating the image reconstruction task as an inverse problem, with the use of numerical optimization methods and with integration of a-priori information in the reconstruction, undersampling artifacts can be eliminated. One example for a-priori information is the use of multiple receive channels and the exploitation of the information about the spatial position of the receive coils (parallel imaging, [5], [6], [7]). Another strategy is to constrain the optimization problem based on a-priori information about the structure of the solution [8], [9], which is often summarized under the name compressed sensing [10]. These strategies allow to reconstruct images with reduced artifacts or even artifact-free images from highly undersampled data sets. The main drawback of these methods is that the computational effort to solve an optimization problems is much higher than that of conventional MR image reconstruction which, in the easiest case, consists of a single multi-dimensional inverse Fourier transform. While this is not a major restriction for research applications, it prohibits the use of these approaches in daily clinical applications, where reconstructed images have to be available immediately after the scan is completed. However, nearly all image reconstruction problems have a very high parallelization potential. Therefore, parallelized implementations on GPUs have already been proposed for parallel imaging [11], image reconstruction from non-cartesian data sets [12] and Total Variation (TV) [13] filtering of undersampled radial data [14].

In this work, image reconstruction from undersampled radial multiple coil data with a Total Generalized Variation ($\mathcal{TGV}$) penalty term is performed. $\mathcal{TGV}$ is an extension of the popular TV model with the additional advantage that it does not suffer from staircasing artifacts in image regions with smooth signal changes [15], [16], [17]. The following optimization problem has to be solved:

$$\min_x \|\mathcal{S}(x) - k\|_2^2 + \lambda \mathcal{TGV}(x) \tag{1}$$

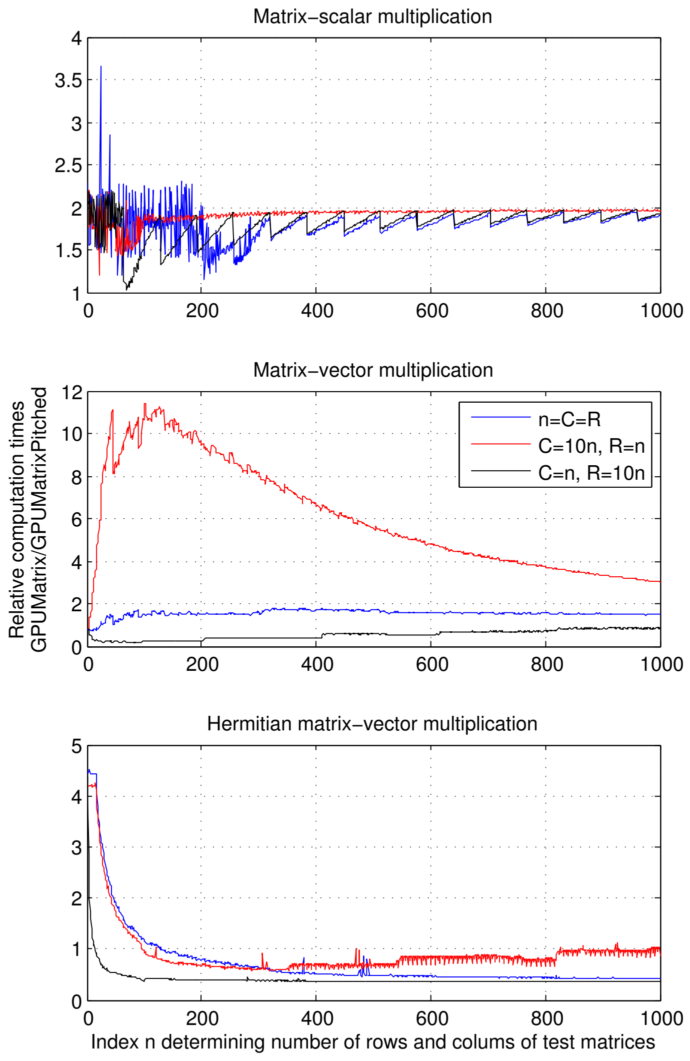

Fig. 2. Comparison of the relative computation times of matrix-scalar (top), matrix-vector (middle) and hermitian matrix-vector (bottom) multiplications. The test matrices were of size $R \times C$, where R and C. The graphs illustrate the cases $R = C$ (blue graphs, square matrices), $C = 10 \cdot R$ (red graphs) and $R = 10 \cdot C$ (black graphs). The ratios of the computation times of CUBLAS (using `GPUMatrix`) to those of `GPUMatrixPitched` are displayed. Values above 1 indicate faster computation with `GPUMatrixPitched`.

In equation (1), x is the reconstructed image $\mathcal{TGV}$ is the penalty functional, $\mathcal{S}$ is the sampling operator that maps an image to the corresponding k-space data and also includes the sensitivity information of receive coils, k is the subsampled k-space data and λ is the weighting parameter which allows to tune the trade-off between the data fidelity and the regularization term. Numerically, equation 1 is solved with the primal-dual algorithm of [18]. To construct the operator $\mathcal{S}$ in equation 1, an estimation of the coil sensitivity profiles is needed. In this work, a reconstruction with quadratic regularization of the derivative (usually called H1-regularization) is used on each, which is followed by the convolution with a smoothing kernel. Sensitivity profiles are then obtained by normalization with the sum of squares image [8], [16].

A fully sampled T2 weighted turbo spin echo scan of the brain of a healthy volunteer was performed with a clinical 3T scanner and a 4 channel head coil. Written informed consent was obtained prior to the examination. Sequence parameters were repetition time TR = 5000 ms, echo time TE = 99 ms, turbo factor 10, matrix size (x,y) = (256,256), 10 slices with a slice thickness of 4 mm and an in plane resolution of 0.86 mm $\times$ 0.86 mm. The raw data set was exported from the scanner and then subsampled, to simulate an accelerated acquisition. Images were reconstructed from 96, 48, 32 and 24 radial projections. According to the literature [19] $\frac{\pi}{2}n$, projections (402 for n = 256 in our case) have to be acquired for a data set in line with the sampling theorem. Therefore this corresponds to undersampling factors of approximately 4, 8, 12 and 16. Each projection consisted of 256 sample points. Results obtained with $\mathcal{TGV}$ are are compared to different reference reconstructions. The most simple reference used was a conventional NUFFT SOS reconstruction (NUFFT [20], [21] reconstruction of each individual receiver channel, followed by a sum of squares combination of the individual channels). Additionally, parallel imaging with conjugate gradient (CG)-SENSE [6], using coil sensitivities obtained by a normalization of the individual receiver channels with the SOS image followed by filtering with a Gaussian low-pass filter, was performed. Finally, CG-SENSE was used with the same H1 based sensitivity profiles used for $\mathcal{TGV}$. All results are shown in Fig. 3. Conventional NUFFT SOS and CG-SENSE reconstructions using the SOS based coil sensitivities suffer from pronounced streaking artifacts which get increasingly worse as acceleration is increased. Higher image quality can be obtained with CG-SENSE if the H1 sensitivity profiles are used. In this case, acceptable image quality can ob obtained in the case of 96 and 48 projections. In the case of higher acceleration, residual streaking artifacts appear in the reconstructions. In these cases of high acceleration, only the $\mathcal{TGV}$ method yields results without residual aliasing.

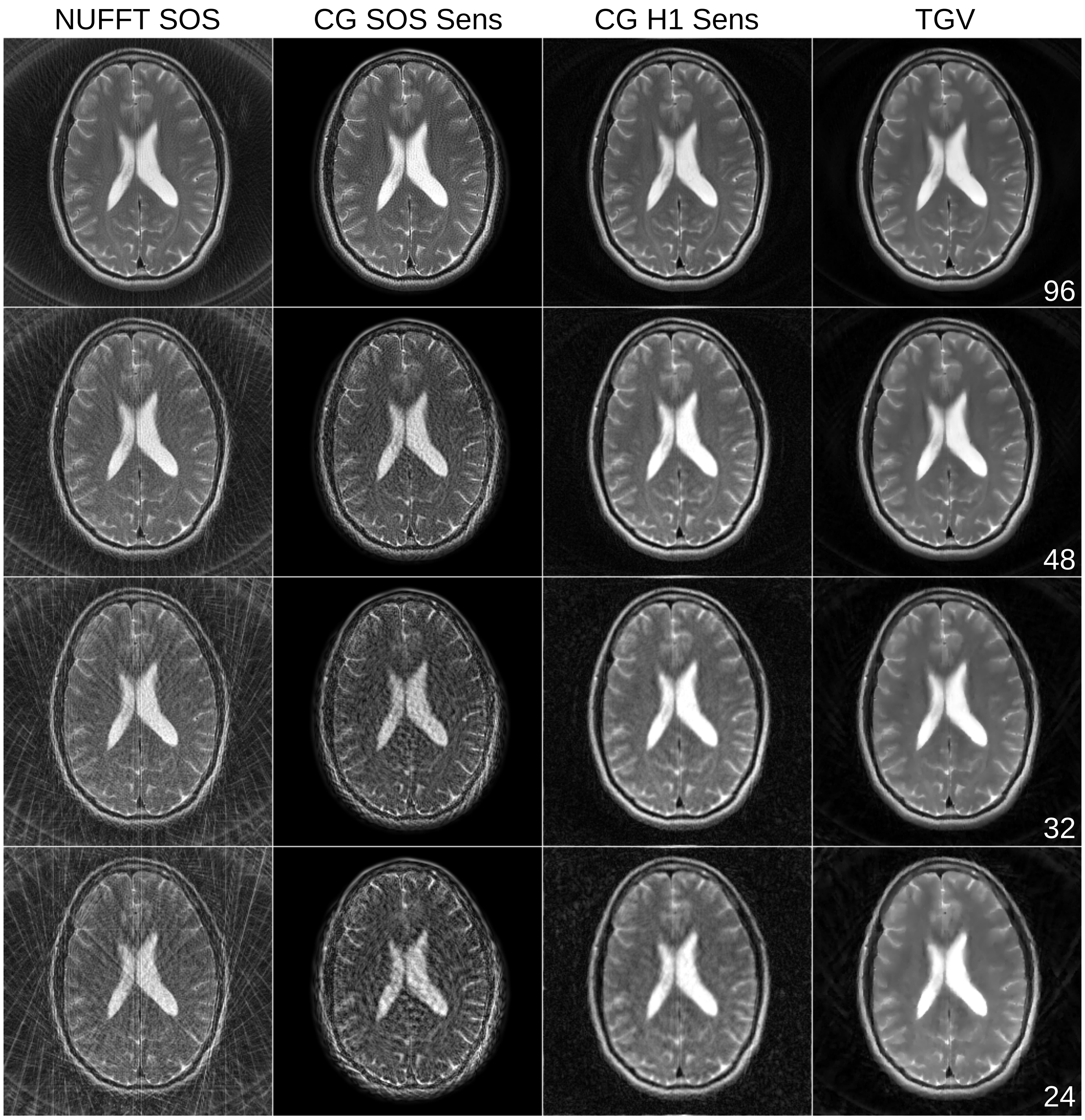

Fig. 3. Image reconstruction from undersampled radial data with 96, 48, 32 and 24 projections. Conventional minimum norm NUFFT reconstruction (first column), CGSENSE with sensitivities from normalization with the SOS reconstruction (second column), CGSENSE with H1 sensitivities (third column) and CUDA $\mathcal{TGV}$ constrained reconstructions (fourth column) are shown.

$\mathcal{TGV}$ was implemented for the *AGILE* library, and this GPU reconstruction was compared to a CPU implementation in C and an implementation in MATLAB (R2010b, 64 bit). *AGILE* was compiled with release 3.0 of the NVCC compiler. To illustrate the differences in computation performance, all computations were then repeated on a second system with the following configuration: Intel Core i7 2.93 GHz (8 cores), 12 GB RAM, NVIDIA GTS450 GPU with 1024 MB memory, release 4.0 of the NVCC compiler, MATLAB (R2011a, 64 bit). A fixed number of iterations (500) was used for all implementations. It should be noted that the reconstruction time of both strategies could be decreased by optimizing the number of iterations. In data sets with less corruption (e.g. the 96 projections data set), no visual changes of the reconstructed image occur after the first 150 iterations while more iterations are needed for data with higher corruption. However, the goal of this experiment was the comparison of the GPU and the CPU implementation. A reduction of iterations affects both implementations in the same way. In order to keep the experimental setup as simple as possible, the number of iterations was fixed to a rather high value which ensures that all artifacts are eliminated in all data sets. The corresponding reconstruction times are given in Table I and Table II shows a speed profile for the respective subtasks.

B. *High-performance fluorescence tomography reconstruction*

Fluorescence tomography (FT) is a relatively new imaging modality that injects light on the surface of an object or a small animal like a mouse, for example, to excite a fluorophore inside the object. From the knowledge of the source position and additional boundary measurements of the emitted fluorescent light, it is sought to reconstruct the distribution of fluorescent dye inside the tissue.

Using several assumptions [22], [23], light propagation in highly scattering media can be modeled by two coupled partial differential equations:

$$-\nabla \cdot (\kappa_x(c)\nabla\varphi_x) + \mu_{a,x}(c)\varphi_x = q, \qquad \text{in } \Omega, \tag{2}$$
$$\varrho\varphi_x + \kappa_x(c)\partial_n\varphi_x = 0, \qquad \text{on } \partial\Omega, \tag{3}$$
$$-\nabla \cdot (\kappa_m(c)\nabla\varphi_m) + \mu_{a,m}(c)\varphi_m = \gamma c\varphi_x, \qquad \text{in } \Omega, \tag{4}$$
$$\varrho\varphi_m + \kappa_m(c)\partial_n\varphi_m = 0, \qquad \text{on } \partial\Omega. \tag{5}$$

In these equations, κ and μ are the optical diffusion and absorption parameters of the tissue, φ is the photon density, ϱ is the reflection coefficient, γ is the conversion efficiency of the fluorophore and c is the concentration of the fluorophore which is the quantity of interest. The computation domain is denoted by Ω and its boundary by $\partial\Omega$. The sub-scripts x and m refer to the excitation and emission wavelength, respectively.

This problem can be discretized by finite elements, for example. This leads to a matrix equation system of the form

$$A_x(c)\varphi_x = q_x \tag{6}$$
$$A_m(c)\varphi_m = A_{x2m}(c)\varphi_x, \tag{7}$$

where A_x and A_m are the sparse system matrices describing the propagation of the excitation and emission light in the tissue and A_{x2m} is another sparse matrix converting light from the excitation to the emission wavelength as sites where a fluorophore

TABLE I
LEFT: COMPARISON OF THE RECONSTRUCTION TIMES AND CORRESPONDING SPEED-UPS OF MATLAB CPU, C CPU AND *AGILE* GPU IMPLEMENTATIONS OF ITERATIVE RADIAL IMAGE RECONSTRUCTION WITH A $\mathcal{TGV}$ CONSTRAINT. COMPUTATIONAL RESULTS ARE SHOWN FOR DATA WITH 96, 48, 32 AND 24 RADIAL PROJECTIONS (INTEL CORE 2DUO 2.40 GHZ, 2 GB RAM, NVIDIA GTX 480 WITH 1536 MB MEMORY). RIGHT: SAME COMPUTATIONS ON A DIFFERENT SYSTEM (INTEL CORE I7 2.93 GHZ, 12 GB RAM, NVIDIA GTS 450 WITH 1024 MB MEMORY).

Reconstruction time system 1					
Data set	MATLAB	C	*AGILE*	SU MATLAB	SU C
96 proj.	242.27 s	1400.47 s	20.47 s	11.84	68.42
48 proj.	236.81 s	784.21 s	10.52 s	22.51	74.55
32 proj.	222.49 s	577.41 s	7.25 s	30.69	79.64
24 proj.	218.73 s	475.76 s	5.47 s	39.99	86.98

Reconstruction time system 2					
Data set	MATLAB	C	*AGILE*	SU MATLAB	SU C
96 proj.	125.10 s	290.00 s	42.92 s	2.91	6.76
48 proj.	111.27 s	178.08 s	23.55 s	4.72	7.56
32 proj.	107.10 s	136.91 s	16.83 s	6.36	8.13
24 proj.	106.47 s	118.36 s	13.29 s	8.01	8.91

TABLE II
RIGHT: COMPUTATIONAL COST PROFILE OF THE SUBTASKS PERFORMED DURING THE *AGILE* IMAGE RECONSTRUCTION PROCEDURE AND COMPARISON TO THE MATLAB CPU IMPLEMENTATION. THESE RESULTS CORRESPOND TO THE CALCULATIONS ON SYSTEM 1 FOR THE 24 PROJECTIONS DATA SET IN TABLE I.

Task	% GPU time	% CPU time
NUFFT operator	79.27	68.25
$\mathcal{TGV}$ minimization	10.91	28.54
Sensitivity estimation	1.61	3.21
Data transfer	0.22	-
Idle	7.99	-

TABLE III
COMPARSION OF CPU AND GPU RECONSTRUCTION TIMES FOR A MOUSE PHANTOM.

Task	CPU		GPU		Speed-up
Assembly of system matrices	121.97	ms	13.83	ms	8.82
Solution of forward systems	5.45	s	461.60	ms	11.81
Solution of adjoint systems	6.02	s	508.29	ms	11.85
Assembly of sensitivity matrix	16.68	s	1.03	s	16.23
Computation of Gauß-Newton update	10.87	s	331.92	ms	32.75
Total reconstruction time	5.41	min	24.94	s	13.01

is present. Because all of the matrices are non-linearly dependent on the fluorophore concentration c, it is necessary to apply iterative methods to solve the inverse problem.

Using the *AGILE* library, we implemented the assembly and the solution of the equation system on the graphics card. To solve the forward and adjoint systems, the method of conjugate gradients together with an geometric multigrid preconditioner was applied. The reconstruction algorithm uses an iteratively regularized Gauß-Newton scheme. The comparison of the reconstruction time was performed on the Digimouse phantom [24] which is a tetrahedral grid with 34,677 nodes and 174,080 elements. Timings are listed in Table III and the outcome is presented in Figure 4. The speed-up is highest for tasks which use dense matrices (such as the computation of the Gauß-Newton update) and significantly reduces when operations require random access to the GPU memory (e.g. during the assembly of the system matrices). The overall speed-up is around 13, which enables non-linear fluorescence tomography reconstructions below 30 seconds. For an in-depth discussion of the algorithms and further performance comparisons we refer to [25]. The reconstructed concentration shown in Figure 4 exhibits the typical characteristics of diffusion optical tomography: the inclusions appear to have a lower concentration and are increased in size. This is due to the high scattering of light in biological tissue and inherent to the imaging modality.

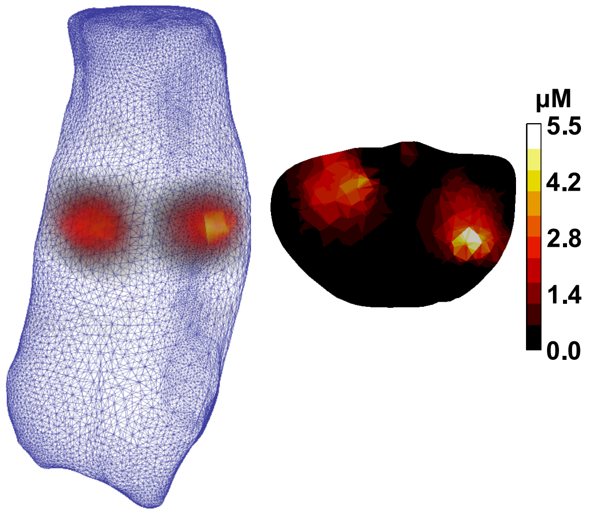

Fig. 4. Reconstruction of two fluorescent inclusions with a concentration of 10 μM inside a mouse torso. The cross-section is made through the center of the inclusions.

V. DISCUSSION

The intention behind the release of this open-source library is to alleviate the usage of the huge potential of graphics hardware in medical image reconstruction on the one hand, and to reduce the steep learning curve of GPU programming for medical applications on the other hand. As the toolbox is an open-source project, the algorithms can be used either right as a black box for the reconstruction of user data, or as a basis for own implementations for similar problems. The latter case is facilitated by an object-oriented and templated design which provides a well defined structure for extensions and allows for maximum code re-usability. For example, a user who is just interested in the reconstruction of own measurement data can use the provided implementation of e.g. the $\mathcal{TGV}$-MRI algorithm. However, it is also possible to use only the matrix-vector multiplication kernels, the linear algebra solvers or anything in between. Due to the integration of fluorescence tomography, the library also provides support for finite element simulations and the corresponding data structures. This distinguishes *AGILE* from other open source projects in the field of medical image reconstruction, which are usually more specialized implementations of a specific reconstruction algorithm. One example for such an approach is the IMPATIENT MRI project [26], developed at the

University of Illinois[1]. This library provides a GPU implementation for MRI compressed sensing and parallel imaging which also includes field inhomogeneity correction [27]. Another example is the l1-SPIRiT software library [28], [29], developed at the University of California, Berkeley[2]. This is an MRI reconstruction method that can be applied to arbitrary sampling patterns and in principle also allows the integration of multiple regularization functionals (the current implementation which is available for download focuses on wavelets). The method is based on a self-consistency criterion and was originally inspired by GRAPPA [7], which means that it does not need explicit knowledge of coil sensitivity maps for parallel imaging. Another GPU reconstruction package, which was developed for real-time MRI applications, especially for cardiac imaging [30], [31], [32], is provided by the Max-Planck Institute in Göttingen[3]. The core of this reconstruction algorithm is a nonlinear inversion approach that allows joint estimation of image content and coil sensitivity profiles, but it also includes specialized filtering and post-processing techniques tailored to dynamic imaging with radial trajectories. Whether *AGILE* or any of these more specialized implementations better suits the needs of a user depends on the specific application.

Regarding the field of optical tomography, GPU-accelerated reconstructions for bioluminescence tomography [33] and diffuse optical tomography [34] have been reported previously, both using the proprietary CULA package [35]. In these applications, data is copied to the GPU which then performs a specific task such as a matrix-matrix multiplication or solving a system of linear equations and the result is copied back to the host. Our approach seeks to minimize data transfer by also using the graphics hardware for non-trivial tasks such as the assembly of the sparse finite-element matrices, for example, for which special CUDA kernels have been written. Only by avoiding costly memory copies, high speed-up factors as shown in Table III are achievable.

Thrust [36] is another exciting general purpose library for writing GPU-accelerated code in a high-level manner in C++. Similar to *AGILE*, it offers an STL-compliant vector class, which reside inside the graphic card's memory. However, the primary focus of *Thrust* and *AGILE* is different: *Thrust* provides means for transforming, scanning or reducing vectors together with STL-like iterators while *AGILE* implements linear algebra operations and solvers.

The results from Table I illustrate that depending on the used systems and data sets, a rather large variation of speed-ups can be achieved. In particular, speed-ups from 2.91 to 86.98 were observed for the same computational problem. While it is not surprising that the speed-ups are much larger for system 1, which is equipped with a superior GPU and a significantly slower CPU in comparison to system 2, a remarkable result is that the MATLAB implementation shows better scaling with respect to the size of the data than the C and the GPU solutions. For example, on system 1, the speed-up of the GPU against MATLAB is 3.38 times larger for the 24 projections data set in comparison to the 96 projections data. In contrast, the speed-ups are much more consistent when comparing the C implementation with the GPU (ratio of 1.27 between smallest and largest data set). However, it is important to remember that for practical applications, the important criterion is the relation of the reconstruction times of *AGILE* in comparison to the corresponding data acquisition times. With the data acquisition parameters of the used data set, data acquisition (TR $\times$ number of projections / turbo factor) takes approximately two times longer than image reconstruction. This means that the reconstruction process is not the time limiting factor in the imaging pipeline.

For comparison reasons, we have also ported our $\mathcal{TGV}$-code to C++. However, a first straightforward implementation was a factor of 2 slower than the MATLAB implementation. One problem is that for the calculation of gradients and derivatives, if-clauses are needed to avoid out-of-bounds array accesses (e.g. only compute the difference with the pixel to the left if the x-index is larger than 0). Such branches inside loops cause most likely a pipeline stall, which significantly reduces the speed of execution. The stalls could be eliminated by loop-unswitching, which moves the conditional outside the loop body. However, this leads to a dramatic increase in source code. Thus, ideally one would pad the images by a pixel with value zero on each side and remove the if-clause inside the loop body at all. However, this also requires to modify the NFFT code as the stride between rows in the image is no longer equal to the size of the image. Code optimizations for speeding up the C++ code to be at least on par with MATLAB is current work in progress.

As can be seen in Table II, the computational bottleneck for the *AGILE* MR image reconstruction on the GPU is clearly the evaluation of the NUFFT operator which roughly requires 80 % of the reconstruction time. This is due to the computationally demanding nature of the operator: In the reference implementation in MATLAB, which makes use of [20], approximately 70 % of the CPU time is spent for evaluating the NUFFT. In contrast, the computational overhead for the $\mathcal{TGV}$ minimization is smaller in the *AGILE* GPU implementation as for the MATLAB implementation. As the whole reconstruction is performed on the GPU, almost no data transfer to the CPU is necessary and GPU idle times are well below 10 %.

Currently, only 2D MR image reconstruction is implemented in *AGILE*. Extensions to 3D are currently under investigation. The main challenge is the increased memory demand when dealing with data from non-Cartesian 3D trajectories. Current graphics hardware is usually equipped with no more than 1.5 GB (NVIDIA GTX 480) of memory. An alternative are specialized boards for GPU computing, which are currently equipped with up to 6 GB of memory (e.g. the NVIDIA Tesla family) but are much less common in standard desktop computer systems. The applications built around the *AGILE* library currently use single-precision floating point operations mostly which is due to the fact that most CUDA capable hardware only supports

[1]http://impact.crhc.illinois.edu/mri.php
[2]http://www.cs.berkeley.edu/~mjmurphy/l1spirit.html
[3]http://www.gwdg.de/~muecker/noir_2010-07-14b.tgz

double-precision in a rudimentary manner. Preliminary investigations show that the difference in accuracy between single and double-precision is negligible in many cases. For example, the single- vs. double-precision error in the reconstructed image of fluorescence tomography is around 1 % which is much lower than any error introduced through the discretization of the geometry or imprecise optode positions. Furthermore, this limitation will eventually fall with the advent of new graphics hardware which is capable of performing single- and double-precision operations in the same time. The difference in programming code will be minor as only the template arguments have to be changed from `float` to `double` which is easy to achieve if programs are written using a `typedef`.

While reconstruction times in the order of several hours may be feasible in research, such delays currently limit the widespread application of novel imaging modalities like fluorescence tomography and advanced image reconstruction methods for already established ones like magnetic resonance imaging.

The decrease in reconstruction time paired with low-cost commodity hardware is seen as a major step forward to introducing fast MR reconstruction algorithms in clinical daily use where it is mandatory that reconstructed images are available immediately after the completion of a scan. Furthermore, it is now possible to fine-tune certain parameters in the reconstruction algorithm, for example the regularization parameter, online by the operator which has not been feasible up to now.

References

[1] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*. NVIDIA Cooperation, 2008.

[2] M. H. Austern, *Generic programming and the STL: using and extending the C++ Standard Template Library*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[3] NVIDIA, *NVIDIA CUBLAS library V2.0*. NVIDIA Cooperation, 2008.

[4] M. Liebmann, "Efficient PDE Solvers on Modern Hardware with Applications in Medical and Technical Sciences," Ph.D. dissertation, University of Graz, 2009.

[5] K. P. Pruessmann, M. Weiger, M. B. Scheidegger, and P. Boesiger, "SENSE: sensitivity encoding for fast MRI." *Magn Reson Med*, vol. 42, no. 5, pp. 952–962, Nov 1999.

[6] K. P. Pruessmann, M. Weiger, P. Börnert, and P. Boesiger, "Advances in sensitivity encoding with arbitrary k-space trajectories." *Magn Reson Med*, vol. 46, no. 4, pp. 638–651, Oct 2001.

[7] M. A. Griswold, P. M. Jakob, R. M. Heidemann, M. Nittka, V. Jellus, J. Wang, B. Kiefer, and A. Haase, "Generalized autocalibrating partially parallel acquisitions (GRAPPA)." *Magn Reson Med*, vol. 47, no. 6, pp. 1202–1210, Jun 2002.

[8] K. T. Block, M. Uecker, and J. Frahm, "Undersampled radial MRI with multiple coils. Iterative image reconstruction using a total variation constraint." *Magn Reson Med*, vol. 57, no. 6, pp. 1086–1098, Jun 2007.

[9] ——, "Model-based iterative reconstruction for radial fast spin-echo MRI." *IEEE Trans Med Imaging*, vol. 28, no. 11, pp. 1759–1769, Nov 2009.

[10] M. Lustig, D. L. Donoho, J. M. Santos, and J. M. Pauly, "Compressed sensing MRI," *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 72–82, March 2008.

[11] M. S. Hansen, D. Atkinson, and T. S. Sorensen, "Cartesian SENSE and k-t SENSE reconstruction using commodity graphics hardware." *Magn Reson Med*, vol. 59, no. 3, pp. 463–468, Mar 2008.

[12] T. S. Sorensen, T. Schaeffter, K. O. Noe, and M. S. Hansen, "Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware," *IEEE Transactions on Medical Imaging*, vol. 27, no. 4, pp. 538–547, April 2008.

[13] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Phys. D*, vol. 60, no. 1-4, pp. 259–268, 1992.

[14] F. Knoll, M. Unger, C. Diwoky, C. Clason, T. Pock, and R. Stollberger, "Fast reduction of undersampling artifacts in radial MR angiography with 3D total variation on graphics hardware." *MAGMA*, vol. 23, no. 2, pp. 103–114, Apr 2010.

[15] K. Bredies, K. Kunisch, and T. Pock, "Total generalized variation," *SIAM Journal on Imaging Sciences*, vol. 3, no. 3, pp. 492–526, 2010.

[16] F. Knoll, K. Bredies, T. Pock, and R. Stollberger, "Second order total generalized variation TGV for MRI." *Magn Reson Med*, vol. 65, no. 2, pp. 480–491, Feb 2011.

[17] F. Knoll, M. Freiberger, K. Bredies, and R. Stollberger, "AGILE: An open source library for image reconstruction using graphics card hardware acceleration," in *Proceedings of the 19th Scientific Meeting and Exhibition of ISMRM, Montreal, CA*, 2011, p. 2554.

[18] A. Chambolle and T. Pock, "A first-order primal-dual algorithm for convex problems with applications to imaging," *Journal of Mathematical Imaging and Vision*, vol. 40, no. 1, pp. 120–145, 2010.

[19] M. A. Bernstein, K. F. King, and X. J. Zhou, *Handbook of MRI Pulse Sequences*. Academic Press, September 2004.

[20] J. A. Fessler and B. P. Sutton, "Nonuniform fast Fourier transforms using min-max interpolation," *IEEE Transactions on Signal Processing*, vol. 51, no. 2, pp. 560–574, Feb. 2003.

[21] J. Keiner, S. Kunis, and D. Potts, "Using NFFT 3—a software library for various nonequispaced fast Fourier transforms," *ACM Trans. Math. Softw.*, vol. 36, no. 4, pp. 1–30, 2009.

[22] S. R. Arridge, "Optical tomography in medical imaging," *Inverse Problems*, vol. 15, pp. R41–R93, 1999.

[23] A. Joshi, W. Bangerth, and K. Hwang, "Fully adaptive FEM based fluorescence optical tomography from time-dependent measurements with area illumination and detection," *Med. Phys.*, vol. 33, pp. 1299–1310, 2006.

[24] B. Dogdas, D. Stout, A. F. Chatziioannou, and R. M. Leahy, "Digimouse: a 3D whole body mouse atlas from CT and cryosection data," *Physics in Medicine and Biology*, vol. 52, pp. 577–587, 2007.

[25] M. Freiberger, H. Egger, M. Liebmann, and H. Scharfetter, "High-performance image reconstruction in fluorescence tomography on desktop computers and graphics hardware," *Biomed. Opt. Express*, vol. 2, no. 11, pp. 3207–3222, Nov 2011.

[26] M. Fu, J. Gai, J. Haldar, W.-M. Hwu, F. Lam, Z.-P. Liang, B. Sutton, X.-L. Wu, and Y. Zhuo, "IMPATIENT MRI: Illinois Massively Parallel Acceleration Toolkit for Image Reconstruction with ENhanced Throughput in MRI," in *Proceedings of the 19th Scientific Meeting and Exhibition of ISMRM, Montreal, CA*, 2011, p. 4396.

[27] B. P. Sutton, D. C. Noll, and J. A. Fessler, "Fast, iterative image reconstruction for mri in the presence of field inhomogeneities." *IEEE Trans Med Imaging*, vol. 22, no. 2, pp. 178–188, Feb 2003.

[28] M. Murphy, K. Keutzer, S. Vasanawala, and M. Lustig, "Clinically feasible reconstruction time for L1-SPIRiT parallel imaging and compressed sensing MRI," in *Proceedings of the 18th Scientific Meeting and Exhibition of ISMRM, Stockholm, Sweden*, 2010, p. 4854.

[29] M. Lustig and J. M. Pauly, "Spirit: Iterative self-consistent parallel imaging reconstruction from arbitrary k-space." *Magn Reson Med*, vol. 64, no. 2, pp. 457–471, Aug 2010.

[30] S. Zhang, K. T. Block, and J. Frahm, "Magnetic resonance imaging in real time: advances using radial FLASH." *J Magn Reson Imaging*, vol. 31, no. 1, pp. 101–109, Jan 2010.

[31] M. Uecker, S. Zhang, and J. Frahm, "Nonlinear inverse reconstruction for real-time MRI of the human heart using undersampled radial FLASH." *Magn Reson Med*, vol. 63, no. 6, pp. 1456–1462, Jun 2010. [Online]. Available: http://dx.doi.org/10.1002/mrm.22453

[32] M. Uecker, S. Zhang, D. Voit, A. Karaus, K.-D. Merboldt, and J. Frahm, "Real-time MRI at a resolution of 20 ms." *NMR Biomed*, vol. 23, pp. 986–994, Aug 2010. [Online]. Available: http://dx.doi.org/10.1002/nbm.1585

[33] B. Zhang, X. Yang, F. Yang, X. Yang, C. Qin, D. Han, X. Ma, K. Liu, and J. Tian, "The CUBLAS and CULA based gpu acceleration of adaptive finite element framework for bioluminescence tomography," *Opt. Express*, vol. 18, pp. 20 201–20 214, 2010.

[34] J. Prakash, V. Chandrasekharan, V. Upendra, and P. K. Yalavarthy, "Accelerating frequency-domain diffuse optical tomographic image reconstruction using graphics processing units," *Journal of Biomedical Optics*, vol. 15, p. 066009, 2010.

[35] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis, "Cula: Hybrid gpu accelerated linear algebra routines," in *SPIE Defense and Security Symposium (DSS)*, April 2010.

[36] J. Hoberock and N. Bell, "Thrust: A parallel template library," 2010, version 1.3.0. [Online]. Available: http://code.google.com/p/thrust/