




# GPU-accelerated Bloch simulations and MR-STAT reconstructions using the Julia programming language

Oscar van der Heide<sup>1,2</sup>  | Cornelis A. T. van den Berg<sup>1,2</sup>  | Alessandro Sbrizzi<sup>1,2</sup> 

<sup>1</sup>Computational Imaging Group for MR Diagnostics and Therapy, Center for Image Sciences, University Medical Center Utrecht, Utrecht, The Netherlands

<sup>2</sup>Department of Radiotherapy, Division of Imaging and Oncology, University Medical Center Utrecht, Utrecht, The Netherlands

## Correspondence

Oscar van der Heide, Computational Imaging Group for MR Diagnostics and Therapy, Center for Image Sciences, University Medical Center Utrecht, Utrecht, The Netherlands.

Email: [o.vanderheide@umcutrecht.nl](mailto:o.vanderheide@umcutrecht.nl)

## Funding information

Dutch Technology Foundation, Grant/Award Number: 17986

## Abstract

**Purpose:** MR-STAT is a relatively new multiparametric quantitative MRI technique in which quantitative parameter maps are obtained by solving a large-scale nonlinear optimization problem. Managing reconstruction times is one of the main challenges of MR-STAT. In this work we leverage GPU hardware to reduce MR-STAT reconstruction times. A highly optimized, GPU-compatible Bloch simulation toolbox is developed as part of this work that can be utilized for other quantitative MRI techniques as well.

**Methods:** The Julia programming language was used to develop a flexible yet highly performant and GPU-compatible Bloch simulation toolbox called BlochSimulators.jl. The runtime performance of the toolbox is benchmarked against other Bloch simulation toolboxes. Furthermore, a (partially matrix-free) modification of a previously presented (matrix-free) MR-STAT reconstruction algorithm is proposed and implemented using the Julia language on GPU hardware. The proposed algorithm is combined with BlochSimulators.jl and the resulting MR-STAT reconstruction times on GPU hardware are compared to previously presented MR-STAT reconstruction times.

**Results:** The BlochSimulators.jl package demonstrates superior runtime performance on both CPU and GPU hardware when compared to other existing Bloch simulation toolboxes. The GPU-accelerated partially matrix-free MR-STAT reconstruction algorithm, which relies on BlochSimulators.jl, allows for reconstructions of 68 seconds per two-dimensional (2D slice).

**Conclusion:** By combining the proposed Bloch simulation toolbox and the partially matrix-free reconstruction algorithm, 2D MR-STAT reconstructions can be performed in the order of one minute on a modern GPU card. The Bloch simulation toolbox can be utilized for other quantitative MRI techniques as well, for example for online dictionary generation for MR Fingerprinting.

## KEYWORDS

Bloch simulations, CUDA, Julia, MR-STAT, quantitative MRI

## 1 | INTRODUCTION

In recent years, new transient-state quantitative MRI (“qMRI”) techniques like MR Fingerprinting<sup>1</sup> (“MRF”) and MR-STAT<sup>2</sup> have emerged that drastically reduce qMRI acquisition times. In MR-STAT, the qMRI problem is posed as a large-scale nonlinear inversion problem. Multiple quantitative tissue parameter maps (e.g.  $T_1, T_2$ ) are generated from a single-short scan by fitting a Bloch-equation-based forward model to the measured, transient-state k-space samples. A major challenge with this technique is managing the reconstruction times and memory requirements of the reconstruction algorithm. In previous work, MR-STAT reconstruction times have been accelerated by utilizing CPU-parallelization,<sup>3</sup> surrogate modeling<sup>4</sup> as well as algorithmic techniques that rely on the assumption of Cartesian-based gradient trajectories.<sup>4,5</sup>

Since GPU hardware has been demonstrated to result in runtime performance gains in many different areas of research, including the field of MRI reconstructions,<sup>6,7</sup> it could be beneficial to implement (the computationally demanding parts) of the MR-STAT reconstruction algorithm on GPU hardware. However, implementing such an algorithm in a low-level language like CUDA C/C++ is challenging and time consuming, especially for MRI researchers whose main area expertise typically does not include low-level programming.

In this work, we utilized the Julia language programming language<sup>8</sup> to implement a GPU-accelerated adaptation of the MR-STAT reconstruction algorithm. Julia is a relatively new programming language that is free and open source, and it is designed specifically for scientific programming purposes. Together with the CUDA.jl package,<sup>9</sup> one can use Julia to write functions that compile down to native instructions to be executed on GPU hardware. We will make use of these language features to implement custom kernels for the computationally demanding tasks of the MR-STAT reconstruction algorithm.

One of those computationally demanding tasks is the need to perform Bloch simulations in each voxel at each iteration of the iterative reconstruction algorithm. A new Julia Bloch simulation toolbox is developed that is highly flexible in the sense that it allows users to implement custom pulse sequences that can then be executed on multiple hardware architectures (single CPU, multithreaded CPU, distributed CPU as well as GPU) using multiple numbers types (e.g., single- and double precision) without code repetition. State-of-the-art runtime performance is achieved with the toolbox, as will be demonstrated through benchmarks against several existing MR physics-based toolboxes<sup>10–13</sup> written in

different programming languages as well as a neural network-based surrogate modeling toolbox.<sup>14</sup> Our Bloch simulation toolbox is released as a stand-alone Julia package called BlochSimulators.jl, available at <https://github.com/oscarvanderheide/BlochSimulators.jl>. The package could prove to be useful not only for MR-STAT but also for other computationally demanding qMRI techniques such as MRF. Another computationally demanding task is the need to perform matrix-vector products with the so called Jacobian matrix. In previous work,<sup>3</sup> a matrix-free algorithm was proposed and implemented on multi-CPU hardware to perform these operations. In the current work, a partially matrix-free variation of this algorithm is proposed and implemented on GPU hardware using custom kernel functions written in Julia. MR-STAT reconstructions are performed using BlochSimulators.jl together with the proposed algorithm and the runtimes are substantially reduced with respect to previously proposed MR-STAT reconstruction techniques.

## 2 | THEORY

In this section we first provide a high-level overview of the MR-STAT reconstruction algorithm presented in Reference 3 and we list the computational bottlenecks that can benefit from GPU acceleration.

Let  $\mathbf{d}_i \in \mathbb{C}^{N_t}$  be the vector of time-domain samples measured at  $N_t$  sampling times  $t_1, \dots, t_{N_t}$  during an MR experiment using a receive coil with index  $i$ . The time-dependent forward model  $s_i$  (after spatial discretization) that is used to synthesize time-domain samples is given by

$$s_i(t) = \sum_{j=1}^{N_v} c_{ij} m(\boldsymbol{\theta}_j, t) e^{-2\pi i \mathbf{k}(t) \cdot \mathbf{r}_j} \Delta_V, \quad (1)$$

where  $N_v$  is the number of voxels within the field-of-view,  $\Delta_V$  is the volume element for each voxel,  $\mathbf{r}_j$  is the vector of spatial coordinates for voxel  $j$ ,  $c_{ij}$  is the receive sensitivity of coil  $i$  in voxel  $j$ ,  $\boldsymbol{\theta}_j$  is the vector of MR-relevant biophysical tissue properties (e.g.,  $T_1, T_2, \rho, \dots$ ) for voxel associated with index  $j$ ,  $\mathbf{k}(t)$  is the k-space trajectory and  $m$  is the complex transverse magnetization whose time-varying behavior is obtained from Bloch simulations. Define the vector  $\mathbf{s}_i$  as

$$\mathbf{s}_i := [s_i(t_1), \dots, s_i(t_{N_t})] \in \mathbb{C}^{N_t}.$$

Note that  $\mathbf{s}_i$  depends on the tissue parameters  $\boldsymbol{\theta}_j$  for all voxels  $j$ . Concatenate all tissue parameters into a single vector  $\boldsymbol{\alpha}$  and concatenate the data vectors  $\mathbf{d}_i$  and signal

**Algorithm 1.** Minimize  $\alpha \rightarrow \frac{1}{2} \|\mathbf{d} - \mathbf{s}(\alpha)\|_2^2$

**Require:** Initial guess  $\alpha_0$

**while** not converged **do**

1. Evaluate forward model  $\mathbf{s}$  at  $\alpha$
2. Compute residual:  $\mathbf{r} = \mathbf{d} - \mathbf{s}$
3. Compute gradient:  $\mathbf{g} = \text{Re}(\mathbf{J}^H \mathbf{r})$ ,  
where  $\mathbf{J} := -\frac{\partial \mathbf{s}}{\partial \alpha}$
4. Solve linear system:  $\text{Re}(\mathbf{J}^H \mathbf{J}) \mathbf{p} = -\mathbf{g}$
5. Update parameters:  $\alpha = \alpha + \mathbf{p}$

**end while**

models  $\mathbf{s}_i$  for all coils  $i$  into vectors  $\mathbf{d}$  and  $\mathbf{s}$ , respectively. The parameter maps (contained in  $\alpha$ ) are then obtained by numerically solving the inverse problem

$$\alpha^* = \underset{\alpha}{\text{argmin}} \frac{1}{2} \|\mathbf{d} - \mathbf{s}(\alpha)\|_2^2. \quad (2)$$

using the Gauss–Newton iterative algorithm as outlined in Algorithm 1.

The computationally demanding steps of this iterative algorithm are Steps 1, 3, and 4. In Step 1, Bloch simulations must be performed in each voxel. In Step 3, the gradient of the objective function is computed, which requires a multiplication with the Jacobian matrix  $\mathbf{J}$  whose columns are given by partial derivatives of the forward model  $\mathbf{s}$  with respect to the tissue parameters in all voxels  $\alpha$ . Computing these partial derivatives of the forward model is typically even more demanding than the Bloch simulations themselves. In Step 4, a linear system involving the Gauss–Newton matrix  $\mathbf{J}^H \mathbf{J}$  is numerically solved by an iterative solver for linear least squares problems.<sup>15</sup> This solver requires repeated multiplications with  $\mathbf{J}$  and  $\mathbf{J}^H$ .

The main aim of this work will be to accelerate MR-STAT reconstructions by using Julia to perform each of the following computationally demanding tasks on GPU hardware:

- (A) Bloch simulations (Step 1);
- (B) Partial derivatives of the forward model  $\mathbf{s}$  with respect to  $\alpha$  (Step 3); and
- (C) Matrix-vector products with  $\mathbf{J}$  and  $\mathbf{J}^H$  (Step 4).

## 2.1 | Bloch simulations

### 2.1.1 | Numerical integration of Bloch equations

In MRI, individual spin isochromats are modeled as three-dimensional (3D) vectors  $\mathbf{M} = (M_x, M_y, M_z)$  whose

dynamics are described by the Bloch equations:<sup>16</sup>

$$\frac{d\mathbf{M}}{dt} = (\gamma \mathbf{B}(t) + \mathbf{D})\mathbf{M}(t) + \frac{\mathbf{M}_0}{T_1}. \quad (3)$$

Here  $\gamma$  is the gyromagnetic ratio,  $\mathbf{B}(t)$  is a  $3 \times 3$  matrix containing the pulse sequence-dependent magnetic field at each timepoint,  $\mathbf{D} = \text{diag}(-1/T_2, -1/T_2, -1/T_1)$  contains the longitudinal ( $T_1$ ) and transversal ( $T_2$ ) relaxation times of the spin isochromat, and  $\mathbf{M}_0 = (0, 0, M_{z,0})$  is its equilibrium magnetization.

A commonly used method to numerically integrate Equation (3) is to manually discretize the pulse sequence in time, and subsequently, for each time-step, applying a splitting method<sup>17</sup> where first a 3D rotation is applied, followed by decay and regrowth operations. That is, given  $\mathbf{M}$  at time  $t$ , the magnetization at time  $t + \Delta t$  is computed as

$$\mathbf{M}(t + \Delta t) = e^{\Delta t \mathbf{D}} e^{-\gamma \Delta t \mathbf{B}(t)} \mathbf{M}(t) + (1 - e^{-1/T_1}) \mathbf{M}_0. \quad (4)$$

Here  $e^{-\gamma \Delta t \mathbf{B}(t)}$  is the rotation operator,  $e^{\Delta t \mathbf{D}}$  the decay operator and  $(1 - e^{-1/T_1}) \mathbf{M}_0$  the regrowth term. We refer to Reference 17 for more details on the splitting approach.

In the presence of spoiling gradients, the extended phase graph (“EPG”) model<sup>18,19</sup> may be more appropriate signal model. In the EPG model, instead of keeping track of individual spin isochromats, one tracks the dynamics of so called configuration states. Like in the case of the isochromat model, Bloch simulations can be performed by repeatedly applying rotation, decay and regrowth operations but in addition, a spoiling operator is required that shifts configuration states (see Reference 20 for a review of the EPG framework).

### 2.1.2 | BlochSimulators.jl

In this section we provide a high-level overview of the design of BlochSimulators.jl: our Julia implementation of a Bloch simulation framework with a focus on speed and flexibility.

For both the isochromat and EPG models, functions are written that implement the above-mentioned rotation, decay, regrowth, and spoiling (EPG only) operations. The functions are designed to be both *type-stable* and *nonallocating*, and form the core of the BlochSimulators.jl package. Type-stability allows Julia’s just-in-time compiler to generate efficient machine code without having to hard-code, for example, whether single or double-precision number types are used. This is useful because, for example, on GPU one might want to perform computations using 32-bit (complex) floating point numbers for performance reasons. On the other

hand, when precision is more important than runtime performance, one might want to use 64-bit floating point numbers instead.

The operator functions are nonallocating in the sense that relatively expensive heap allocations are avoided during their execution. This is in general beneficial for runtime performance. An important ingredient for achieving non-allocating code is the `StaticArrays` package which introduces arrays whose sizes are known at compile time and thus allows them to be stack-allocated. In `BlochSimulators.jl`, the isochromats and configuration state matrices are stored using `StaticArrays.jl`.

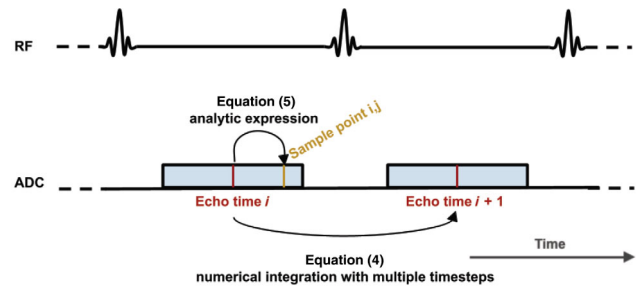
Simulators for entire pulse sequences can in principle be assembled by combining the individual operator functions. Because the individual operator functions are type-stable and non-allocating, a proper combination of these functions should result in type-stable and non-allocating sequence simulations as well.

In `BlochSimulators.jl`, we follow a convention where computing the magnetization at echo times (without taking into account gradient encoding) is separated from computing the magnetization at other readout times (with gradient encoding) and evaluating the volume integral in Equation (1). The reason for this separation is twofold. Firstly, for some applications such as computing MRF dictionaries, only the magnetization at echo times without gradient encoding is needed. Secondly, when performing 2D scans, slice profile correction mechanisms typically involve a summation over multiple simulations per voxel using either different locations of isochromats in the slice-select direction or different effective radiofrequency (RF) flip angles.<sup>21</sup> After the net transverse magnetization (i.e., after summation) in a voxel at some echo time  $t^*$  has been computed, a simplification of Equation (4) allows the net transverse magnetization at another time  $t$  during that readout to be computed analytically as

$$\mathbf{M}(t) = \mathbf{M}(t^*) e^{-\frac{t-t^*}{T_2}} e^{-2\pi i(t-t^*)\Delta B_0} e^{-2\pi i\mathbf{k}(t)\cdot\mathbf{r}}, \quad (5)$$

where  $\mathbf{r}$  is the spatial position of the voxel,  $\mathbf{k}$  is the k-space trajectory and  $\Delta B_0$  is the local off-resonance. The net magnetization in a voxel at other readout times can thus be computed *after* the summation necessary for the slice profile correction, therefore reducing the total number of computations. The two different timescales (i.e., between echo times and between sample points within a readout) are illustrated in Figure 1. Note that Equation (5) assumes there is no spin motion. Spin motion is currently not supported by `BlochSimulators.jl`.

In order to compute the magnetization at echo times for a particular sequence, a new custom Julia struct (subtype of either `IsochromatSimulator`

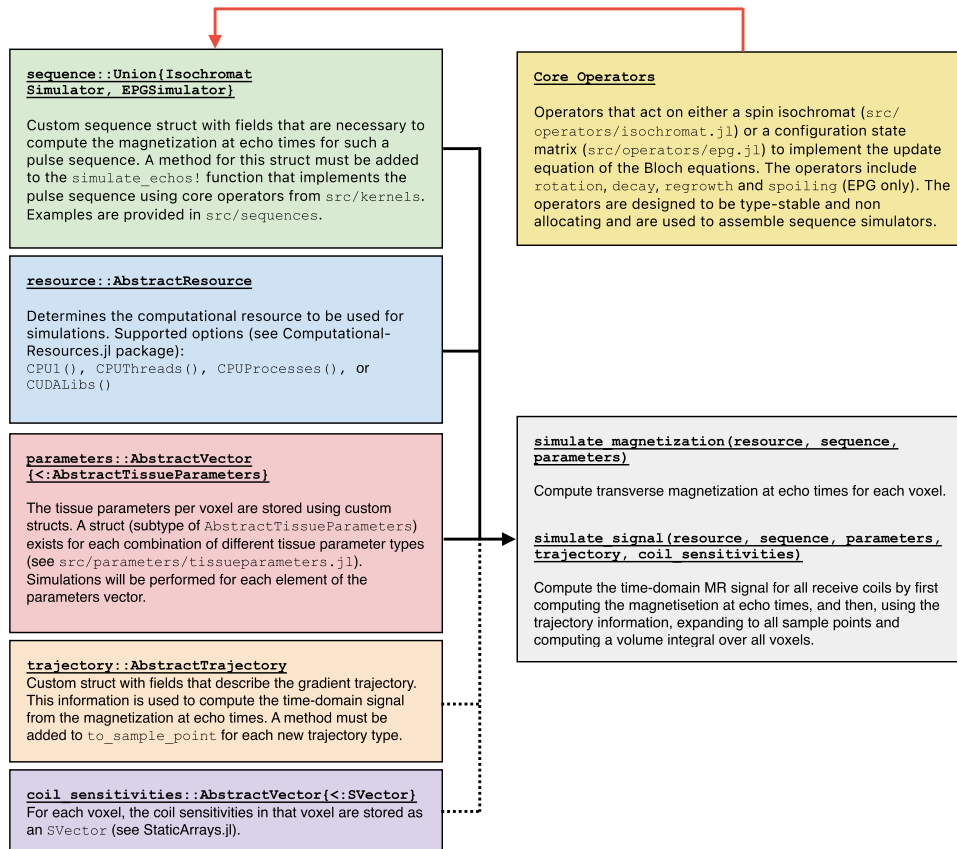


**FIGURE 1** In `BlochSimulators.jl` a convention is followed where at first only the magnetization at the different echo times is computed. Equation (4) is used to propagate the magnetization state from the  $i$ th echo time to the  $(i+1)$ th echo time. Typically there are RF excitations (indicated by the waveforms on the RF line) in between echo times which may require multiple timesteps to be properly simulated. Given the magnetization at the  $i$ th echo time, the magnetization at the  $j$ th sample point of that readout (the blue bar on the ADC line) is computed analytically using Equation (5).

or `EPGSimulator`) with fields that are necessary to describe the pulse sequence (e.g., pulse repetition time [TR], echo time [TE] and flip angles). Given the sequence struct, a new method must be added to the `simulate_magnetization!` function which uses the fields of the struct, together with the above-mentioned core operators, to implement the actual pulse sequence. Similarly, for each type of gradient trajectory, a custom Julia struct (subtype of `AbstractTrajectory`) must be introduced. A graphical overview of the code structure underlying `BlochSimulators.jl` is shown in Figure 2. Further details on the code structure, implementation details and examples are provided in the online documentation available at (<https://oscarvanderheide.github.io/BlochSimulators.jl/dev/>). Note that `BlochSimulators.jl` does not provide a default time discretization step that applies to all simulators. Each simulator comes with its own (user-defined) discretization scheme instead that, based on knowledge of the underlying pulse sequence, may alternate between using Equation (4) with small timesteps during RF excitation and the analytic update step Equation (5) when no RF is applied.

## 2.2 | Partial derivatives of the forward model

To compute partial derivatives of the forward model with respect to the tissue parameters  $\alpha$ , we use the finite difference technique<sup>22</sup> to compute the partial derivatives at echo times first. The finite difference technique directly translates the GPU acceleration from `BlochSimulators.jl` to the partial derivative computations. Given the partial



**FIGURE 2** Graphical overview of the code structure underlying the BlochSimulators.jl package. For further details, we refer to the online documentation available at <https://oscarvanderheide.github.io/BlochSimulators.jl/dev/>.

derivatives at echo times, the partial derivatives at other readout times are computed by manually differentiating the analytical expression from Equation (5) and, again, custom CUDA kernels are written in Julia to execute these partial derivative computations on GPU hardware.

### 2.3 | Matrix-vector products with $\mathbf{J}$ and $\mathbf{J}^H$

Computing the gradient in step 3 of Algorithm 1 involves a matrix-vector multiplication with  $\mathbf{J}^H$ . Numerically solving the linear system in step 4 of Algorithm 1 also involves repeated multiplications with  $\mathbf{J}$  and  $\mathbf{J}^H$ . In previous work,<sup>3</sup> it was argued that storing  $\mathbf{J}$  requires more computer memory than typically available and a matrix-free implementation was proposed that computes the matrix-vector products without having to store  $\mathbf{J}$  into computer memory. However, this method requires recomputation of the entries of  $\mathbf{J}$  for each matrix-vector multiplication. While memory-efficient, the method involves many redundant computations. In this work, we propose a *partially matrix-free* implementation instead, where at first only the partial derivatives of the transverse magnetization at echo times are computed and stored in computer memory. For 2D MR-STAT reconstructions at clinically relevant

resolutions it should be possible to store these arrays on modern GPU cards, see Supporting information S1.

Having stored arrays with partial derivatives at echo times in memory, matrix-vector multiplications with  $\mathbf{J}$  and  $\mathbf{J}^H$  can be then performed in parallel by following slight modifications of the algorithms presented in Reference 3: instead of recomputing the partial derivatives at all readout times, the processes read in the partial derivatives at echo times and then use Equation (5) (differentiated w.r.t. the tissue parameters at hand) to compute the values at the remaining readout times. Pseudo-code for these algorithms is presented in Algorithms 2 and 3. In this pseudo-code, we assume for simplicity that only a single tissue parameter per voxel is considered, that no coilmaps are used in the reconstruction and that the same number of samples is acquired during each readout.

CUDA kernels are written in Julia to perform the partially matrix-free evaluation of Equation (1) and multiplications with  $\mathbf{J}$  (Algorithm 2) and  $\mathbf{J}^H$  (Algorithm 3), respectively.

With tasks A, B, and C being made GPU compatible (with the help of BlochSimulators.jl), we arrive at a GPU-compatible, partially matrix-free MR-STAT reconstruction algorithm. Example Julia code to perform a partially matrix-free MR-STAT reconstruction on numerical phantom data with a Cartesian gradient

**Algorithm 2.** Partially matrix-free, parallelized algorithm for computing matrix-vector products  $\mathbf{J}\mathbf{x}$

**Assumptions:** Let  $\{N_r, N_s, N_v\}$  be the total number of {readouts, samples per readout, voxels}.

**Input:**

- Matrix with partial derivatives at echo times  $\tilde{\mathbf{J}} \in \mathbb{C}^{N_r \times N_v}$  computed with a `sequence` struct
- A `trajectory` struct describing the gradient trajectory
- Input vector  $\mathbf{x} \in \mathbb{R}^{N_v}$

**Output:**

- Matrix-vector product  $\mathbf{J}\mathbf{x}$  stored in  $\mathbf{y} \in \mathbb{C}^{N_r N_s}$

**Algorithm Kernel:**

Let  $1 \leq t \leq N_s N_r$  be the current process' global index.

# Compute readout and sample indices

$r, s = \text{quotient}(t, N_r), \text{remainder}(t, N_r)$

**for**  $v = 1, \dots, N_v$  **do**

  # Load partial derivative at  $r$ -th readout

$J_{r,v} = \tilde{\mathbf{J}}(r, v)$

  # Compute partial derivative at  $s$ -th sample point

$J_{s,v} = \text{dto\_sample\_point}(J_{r,v}, \text{trajectory}, s, \dots)$

  # Multiply with input vector and accumulate

$\mathbf{y}(t) += J_{s,v} * \mathbf{x}(v)$

**end for**

trajectory on GPU hardware is available at <https://github.com/oscarvanderheide/mrstat>.

### 3 | METHODS

We benchmarked BlochSimulators.jl in terms of runtime performance against several existing Bloch simulation packages available online. Benchmarks were performed on both CPU and GPU hardware, using both the isochromat and EPG model. Secondly, we benchmarked the runtime of the partially matrix-free MR-STAT reconstruction algorithm on GPU against previously presented MR-STAT reconstruction techniques. All CPU-based experiments were performed on an Intel(R) Xeon(R) W-2245. The GPU-based experiments were performed on an NVIDIA RTX A5000 with 24 GB memory.

#### 3.1 | Performance benchmarks: BlochSimulators.jl

Although users of BlochSimulators.jl are encouraged to assemble their own simulators, example simulators are provided as part of the source code (see the examples

**Algorithm 3.** Pseudo-code to compute  $\mathbf{J}^H \mathbf{y}$  in partially matrix-free, parallelized fashion

**Assumptions:** Let  $\{N_r, N_s, N_v\}$  be the total number of {readouts, samples per readout, voxels}.

**Input:**

- Matrix with partial derivatives at echo times  $\tilde{\mathbf{J}} \in \mathbb{C}^{N_r \times N_v}$  computed with a `sequence` struct
- A `trajectory` struct describing the gradient trajectory
- Input vector  $\mathbf{y} \in \mathbb{C}^{N_s}$

**Output:**

- Matrix-vector product  $\mathbf{J}^H \mathbf{y}$  stored in  $\mathbf{z} \in \mathbb{C}^{N_v}$

**Algorithm Kernel:**

Let  $1 \leq v \leq N_v$  be the current process' global index.

# Initialize accumulator

$tmp = 0$

**for**  $r = 1, \dots, N_r$  **do** # Load partial derivative at  $r$ -th readout

$J_{r,v} = \tilde{\mathbf{J}}(r, v)$

**for**  $s = 1, \dots, N_s$  **do**

    # Compute partial derivative

    # at  $s$ th sample point

$J_{s,v} = \text{dto\_sample\_point}(J_{r,v}, \text{trajectory}, s, \dots)$

    # Multiply with input vector and accumulate

$tmp += \text{conj}(J_{s,v}) * \mathbf{x}(v)$

**end for**

**end for**

$\mathbf{z}(v) = tmp$

subdirectory). For some of these examples we performed benchmarks against other online available Bloch simulations toolboxes in terms of runtime performance. For each of the benchmarked simulators we will describe their purpose, limitations and provide some details on the benchmark setup. The scripts used for benchmarking are part of the source code (see the benchmarks subdirectory). Within these scripts, the accuracy of the simulators is assessed using Julia's `isapprox` operator. We also note that BlochSimulators.jl contains unit tests for each of the core operators (see `test/runtests.jl`).

##### 3.1.1 | "Generic2D/3D" simulator

First, we compared BlochSimulators.jl against an isochromat-based Bloch simulator written in the C programming language written by Hargreaves,<sup>10</sup> available at <http://mrsrl.stanford.edu/~brian/blochsim/>. The Hargreaves Bloch simulator takes as input a vector with time intervals, together with RF and gradient values and for each time interval it updates the  $x, y$ , and  $z$  components

of a spin isochromat with given  $T_1$ ,  $T_2$  and off-resonance values using Equation (4). One might use such an isochromat simulator to simulate gradient-balanced pulse sequences. In BlochSimulators.jl, a sequence simulator called “Generic3D” was implemented that requires similar input vectors and performs the same update steps. For benchmarking purposes, simulations were performed to generate dictionaries for 1000 up to 10 000 different combinations of  $T_1$  and  $T_2$  values for a sequence with 1120 readouts. The simulations were performed on a single CPU using both single- and double precision floating point numbers. More details of the simulation setup are described in Supporting Information S2 and the variable flip angle train used in the simulations is depicted in Figure S1. For the Generic3D simulator, we generated dictionaries on the GPU as well using 10 000 up to 350 000 combinations of  $T_1$  and  $T_2$  values. The maximum number of 350 000 was chosen to be in the order of a typical MR Fingerprinting dictionary size and requires 2.9 GB of memory. The Generic3D simulator does not take into account slice profile effects. For this purpose, a simulator called Generic2D is also provided. During setup it requires a vector of different  $z$ -locations. Within each voxel, it performs multiple simulations, one for each different  $z$ -location, and accumulates the resulting magnetization. The Generic2D simulator is not benchmarked.

The Generic2D/3D (and the Hargreaves) simulators are “generic” in the sense that they allow arbitrary RF and gradient waveforms as inputs. The simulators, however, are not feature-complete and, for example, assume there is no motion during the sequence and also they do not support diffusion and magnetization transfer effects.

### 3.1.2 | “pSSFP2D/3D” simulator

Most pulse sequences are repetitive by nature. For example, in most conventional MR sequences, only the gradient encoding is different between different repetitions of the base sequence block. For transient-state sequences as used in MR Fingerprinting or MR-STAT the flip angle typically does change each TR but the nominal RF excitation waveform is fixed throughout the sequence. Whereas it would be possible to perform simulations for such a (transient-state) sequence with the simulator from Hargreaves or the Generic3D simulator from BlochSimulators.jl, runtime memory access could be reduced by incorporating knowledge of the repetitive nature of the sequence into the simulation code. For example, instead of reading in from memory at each timepoint of the simulation the actual RF field during that timepoint, the nominal RF waveform could be loaded once up front and scaled with the desired flip angle each

TR instead. We take this approach with the “pSSFP3D” (pseudo steady-state free precession<sup>23</sup>) sequence that is provided as an example in BlochSimulators.jl. For this particular sequence implementation, the TR and TE remain fixed throughout the sequence but the flip angle is allowed to change each TR. The runtime for this sequence-specific simulator was benchmarked using a similar setup as used for the Hargreaves simulator and the Generic3D simulator. The Generic3D simulator provides no support for motion, diffusion, magnetization transfer effects or slice profile effects. A “pSSFP2D” simulator is provided that supports slice profile effects by summing up simulations with different (user-provided) locations along the slice direction for within each voxel.

### 3.1.3 | “FISP2D/3D” simulator

For a third benchmark, we performed simulations for a transient-state FISP-type sequence with time-varying flip angles<sup>24</sup> based on the EPG model (additional sequence simulation details are provided in Supporting Information S3). In BlochSimulators.jl, a sequence simulator “FISP2D” was implemented for this purpose. The FISP2D simulator assumes instantaneous RF excitations and fixed TR and TE throughout the sequence. Slice profile effects are approximated using the partitioned EPG method,<sup>21</sup> where multiple simulations are performed within a voxel, each with (user-provided) flip angle scaling factors corresponding to different positions along the slice direction. We compared this against the FISP-type simulator from SnapMRF,<sup>12</sup> an MR Fingerprinting dictionary simulation framework that runs on GPU hardware and is written in native CUDA, available at <https://github.com/dongwang881107/snapMRF>.

It has been demonstrated that neural networks can be trained to act as a surrogate model for Bloch simulations with high accuracy and fast runtime performance.<sup>25</sup> For the fourth benchmark, we compare the FISP simulator from BlochSimulators.jl against against the recurrent neural network proposed in Reference 26 (“RNN-EPG”) that is trained to perform simulations for a FISP2D sequence. We note that EPG-RNN was trained on sequences for which the RF excitations always have the same phase. The calculation of the response of configuration states to an RF excitation then simplifies and the magnetization can also be described using real numbers instead of complex numbers. Using type-stability of BlochSimulators.jl’s kernel function, together with Julia’s multiple dispatch system, we could use the same FISP sequence implementation for both scenarios (RF excitations with constant and time-varying phases). Furthermore, RNN-EPG computes the partial derivatives of the magnetization with

respect to  $T_1$  and  $T_2$ , respectively, at the same time as the magnetization itself. For a fair runtime comparison, we used finite differences to compute the partial derivatives with the FISP2D simulator from BlochSimulators.jl. A FISP3D sequence that does not take into account slice profile effects is also included in BlochSimulators.jl (not benchmarked).

### 3.1.4 | “pSSFP2D” + “Radial” simulator

For the final benchmark, we measure runtimes for evaluating the forward model (Equation 1) using both BlochSimulators.jl and KomaMRI.jl. KomaMRI.jl is another Julia Bloch simulation package that allows for evaluation of the forward model (Equation 1) for arbitrary sequences and it supports GPU hardware as well. Both toolboxes were used to simulate the signal from a gradient-balanced, transient-state sequence with golden angle radial readouts. Within BlochSimulators.jl, the pSSFP2D sequence was used to simulate the magnetization at echo times first. Subsequently, a radial trajectory implementation was used to compute the magnetization at other readout times and simultaneously perform the volume integration in Equation (1). This radial trajectory stores the starting position in k-space for each readout together with the k-space step in between samples. As such, it provides no support for spin displacement effects during readouts. KomaMRI.jl does not offer sequence-specific simulators, but does provide support for spin displacement throughout the sequence as well as diffusion effects, for example. The main purpose of this benchmark is to demonstrate the potential runtime benefits of using sequence/trajectory specific information in the simulations in situations where these additional features are not considered necessary. We performed signal simulations for a sequence of 500 TRs with 10 000 up to 350 000 voxels using single precision computations on GPU hardware. Additional simulation details are provided in Supporting Information S4.

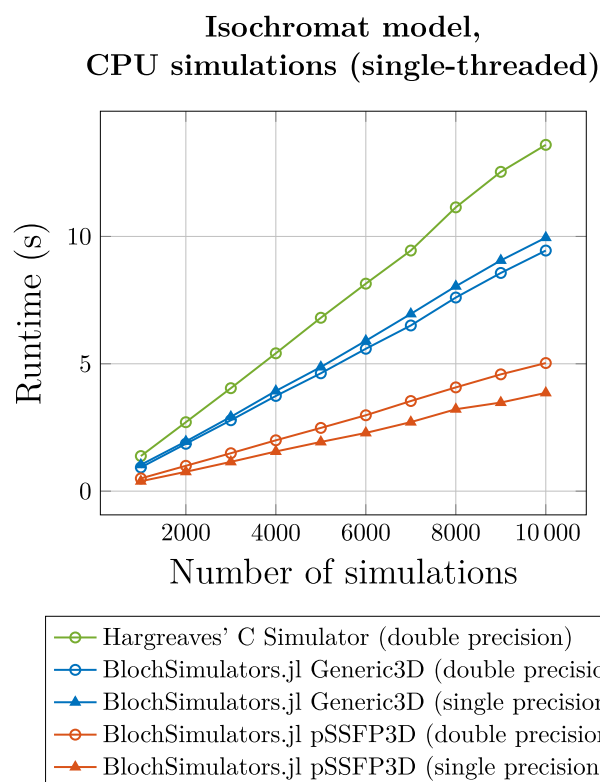
## 3.2 | Performance benchmarks: MR-STAT Reconstructions on GPU

The Julia GPU implementation of the partially matrix-free MR-STAT reconstruction algorithm was benchmarked in terms of runtime performance against the matrix-free distributed CPU implementation from Reference 3, a sparse Hessian distributed CPU implementation from Reference 5 and a neural-network and ADMM implementation from Reference 4. In all these previous reports, the same 2D in-vivo brain dataset obtained using a clinical 1.5T MR System (Ingenua, Philips Healthcare) was reconstructed.

For this dataset, 1120 Cartesian readouts were acquired with gradient-balanced, variable-flip angle sequence with  $TE/TR = 3.8 \text{ ms}/7.6 \text{ ms}$  and a total scan time of 8.5 s. The flip angle train and phase encoding order are depicted in Figure S1. The field-of-view was set to  $224 \text{ mm} \times 224 \text{ mm} \times 5 \text{ mm}$  and the reconstructed resolution was  $1 \text{ mm} \times 1 \text{ mm} \times 5 \text{ mm}$ . The partially matrix-free GPU implementation will be used on this same dataset, using 10 outer iterations of Algorithm 1 and a maximum of 20 inner iterations (Step 4 of Algorithm 1) to reconstruct  $T_1$ ,  $T_2$  and proton density (complex) maps. The quantitative maps will be compared against the maps obtained using the algorithm from Reference 3, which we consider to be the reference implementation for the purpose of this work.

## 4 | RESULTS

In Figure 3, CPU runtimes for Hargreaves' simulator and the Generic3D and pSSFP3D simulators from BlochSimulators.jl are shown. The Generic3D simulator outperforms Hargreaves' simulator by approximately 33%, demonstrating the effectiveness of Julia's Just-In-Time compiler to generate efficient machine code. Additionally,



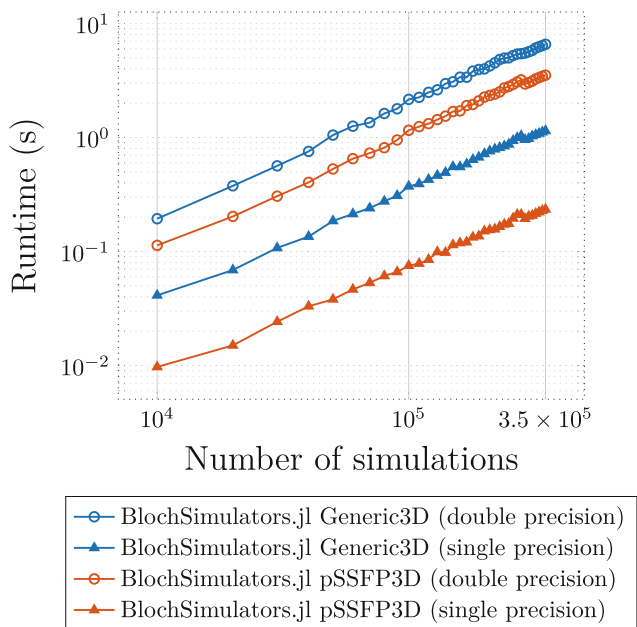
**FIGURE 3** Runtime performance comparison of Hargreaves' C simulator (double precision only) and the Generic3D and pSSFP3D simulators from BlochSimulators.jl (single and double precision) on CPU.



the pSSFP3D simulator is seen to be approximately 50% faster than the Generic3D simulator which showcases the benefit of exploiting the repetitive nature of a pulse sequence to reduce runtime memory access. In BlochSimulators.jl it is relatively easy to use its kernel functions to assemble a simulator corresponding to a specific pulse sequence. For the pSSFP3D simulator there is a performance gain of approximately 20% when single precision is used instead of double precision. For the Generic3D simulator, double precision is actually slightly faster, likely due to different compiler optimizations being performed. The Hargreaves' simulator is hardcoded to double precision and therefore no single precision simulations were performed.

The design of BlochSimulators.jl allows the Generic3D and pSSFP3D sequence implementations to be directly executed on the GPU and the runtime results are shown in Figure 4. On the GPU we observe significant differences between single and double precision with the single precision Generic3D and pSSFP3D simulators being approximately six times and 15 times faster compared to their double precision counterparts, respectively. Furthermore, the single precision pSSFP3D simulator is approximately five times faster than the single precision Generic3D simulator and performs simulations in 350 000 voxels in 0.22 s. Extrapolating the CPU runtimes for this simulator from

Isochromat model, GPU simulations



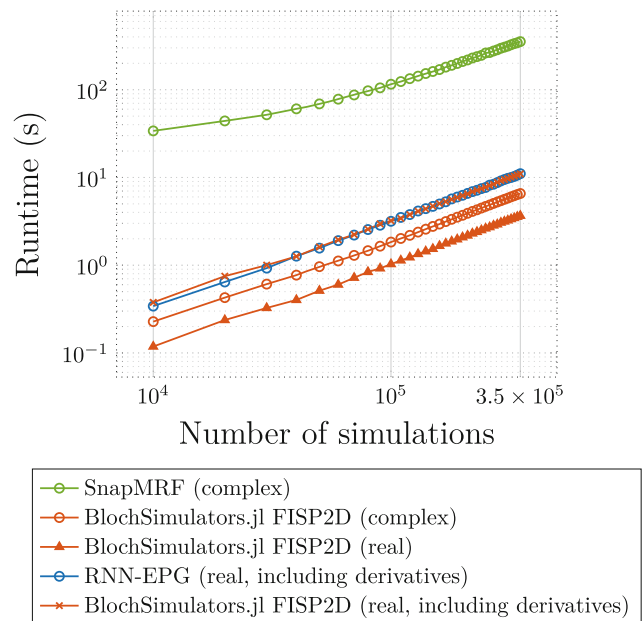
**FIGURE 4** Runtime performance comparison of the Generic3D and pSSFP3D simulators from BlochSimulators.jl (single and double precision), this time on GPU. The runtime performance benefit of using single precision floating point numbers is more pronounced on GPU hardware.

Figure 3 to 350 000 simulations would result in a runtime of approximately 140 s. That is, the GPU results in a 600× speedup in this scenario.

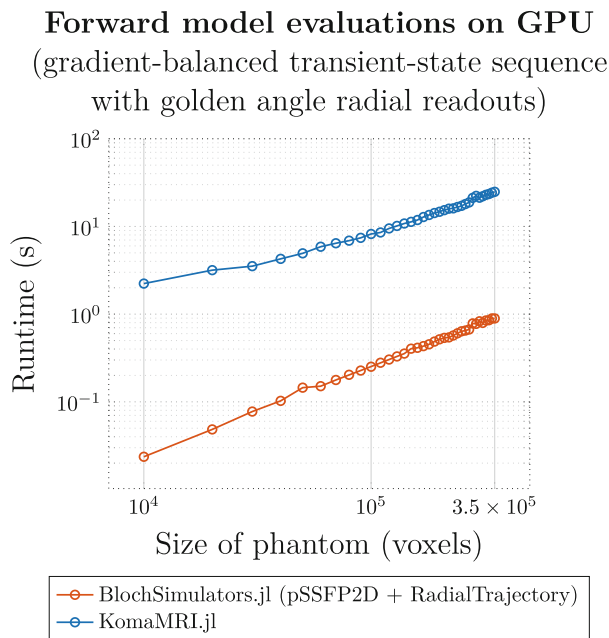
For the EPG benchmark we performed single precision simulations on GPU hardware using the FISP2D simulator from BlochSimulators.jl, SnapMRF<sup>12</sup> and RNN-EPG.<sup>26</sup> The results are displayed in Figure 5. Our FISP2D implementation (annotated with “complex” in Figure 5 to indicate that the RF excitations can have varying phases) is more than 50 times faster than SnapMRF. This difference may be explained by different design choices in the implementation such as using shared memory to store the configuration states, and further exemplifies the ability of Julia’s compiler to generate efficient machine code. When RF excitations are assumed to have constant phase (annotated with “real” in Figure 5), the FISP2D runtimes decrease by approximately a factor of two. If we use finite differences to compute partial derivatives with respect to  $T_1$  and  $T_2$ , the FISP2D simulator is as fast as RNN-EPG.

In Figure 6 the runtimes for evaluating the forward model Equation (1) using both KomaMRI.jl<sup>13</sup> and BlochSimulators.jl are displayed. We observe that BlochSimulators.jl is faster, with the speedup factor depending on the number of voxels used in

EPG model, single precision GPU simulations



**FIGURE 5** Runtime performance comparison of a variable flip-angle FISP-type sequence. The EPG model is used for signal simulations using an implementation from BlochSimulators.jl (“FISP2D”), the native CUDA implementation from SnapMRF<sup>12</sup> and RNN-EPG.<sup>26</sup>



**FIGURE 6** Runtimes for evaluating the forward model Equation (1) using both KomaMRI.jl and BlochSimulators.jl on a gradient-balanced sequence with golden angle radial readouts.

the simulations. For a typical 2D phantom size of  $256^2 = 65\,536$  voxels BlochSimulators.jl is approximately 37.5 times faster whereas for 350 000 voxels it is approximately 27.5 times faster.

Quantitative  $T_1$ ,  $T_2$  and proton density maps for the in-vivo dataset reconstructed using the proposed partially matrix-free MR-STAT reconstruction algorithm and the reference implementation from Reference 3 are displayed in Figure 7, together with absolute relative error maps.

The mean absolute relative errors in  $T_1$ ,  $T_2$  and proton density are 0.3%, 0.69%, and 0.4%, respectively. These small differences stem from finite differences being used for partial derivative computations as compared to (hand-written) forward mode automatic differentiation in the reference. Secondly, single precision computations are being performed in the proposed GPU implementation while the reference uses double precision.

In Table 1, reconstruction times for different MR-STAT reconstruction techniques on the same dataset are reported. The proposed partially matrix-free approach on GPU resulted in a reconstruction time of 68 s, almost twice as fast as the prior state-of-the-art.<sup>4</sup> This reconstruction required 621 MB of GPU memory was required to store the magnetization at echo times as well as the partial derivatives at each iteration. In Table 2 we further outline the computationally demanding tasks of the reconstruction algorithm and what percentage of the reconstruction time is spent on these tasks. Matrix-vector multiplications with  $\mathbf{J}$  form the primary bottleneck, account for approximately 65% of the total reconstruction time.

## 5 | DISCUSSION

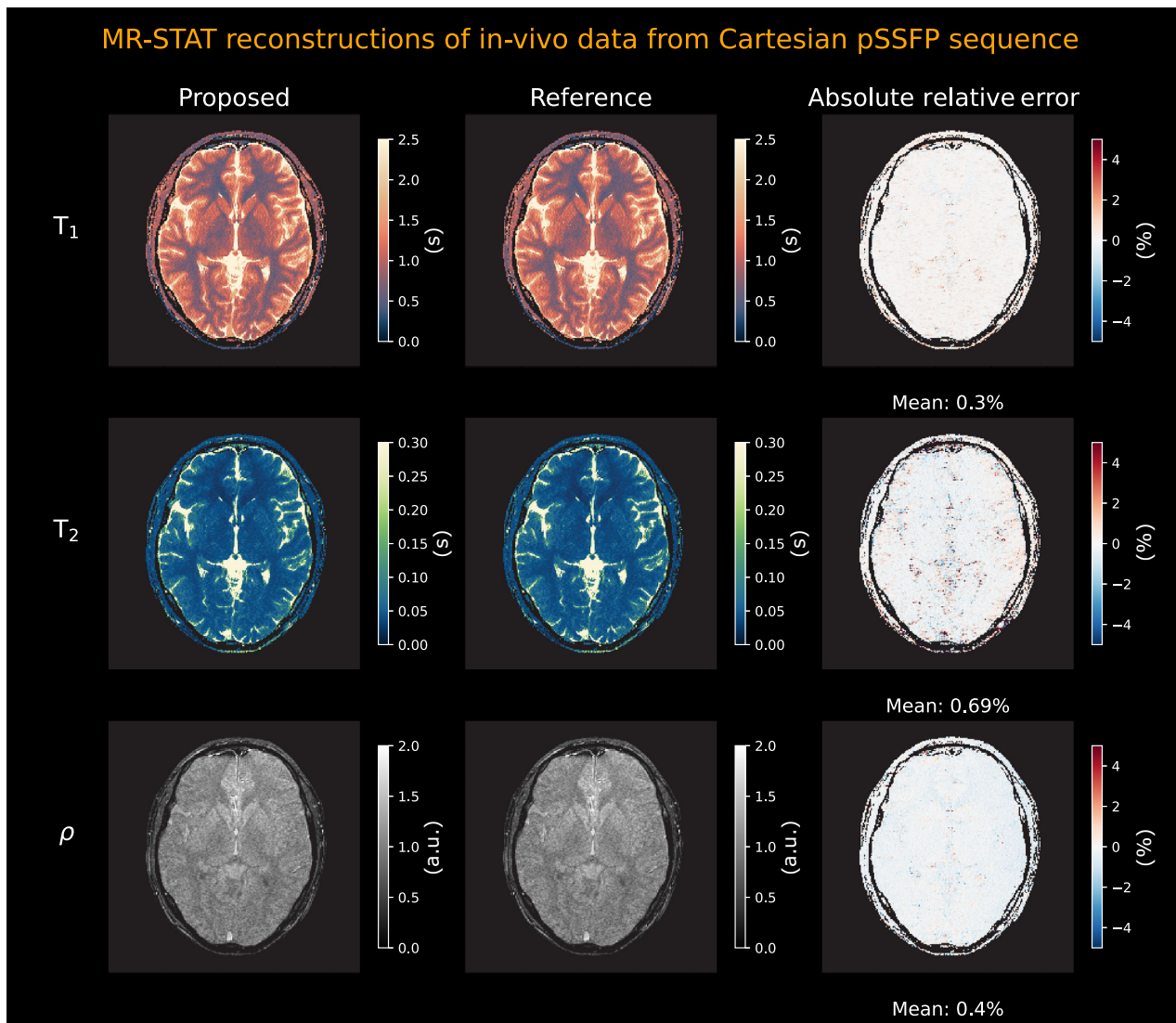
In this work, we explored the potential of GPU hardware for accelerating MR-STAT reconstructions. Rather than using labor intensive programming language like CUDA C/C++, the Julia programming language was used to write GPU kernel functions instead. The standalone software package BlochSimulators.jl that was developed as part of this work can be used to perform Bloch simulations using both the isochromat model and the EPG model. Sequence specific simulators can be assembled that, for example, exploit the repetitive nature of pulse sequences. The runtime benchmarks demonstrated that the performance of BlochSimulators.jl surpasses that of other Bloch simulation toolboxes developed in static, compiled languages. Besides using BlochSimulators.jl for forward model evaluations in MR-STAT, the package could be used, for example, for online generation of MR Fingerprinting dictionaries.

Although not demonstrated in the benchmarks, BlochSimulators.jl also supports multithreaded and distributed CPU computations. It also generalizes to different number types such as single- and double precision but also to more exotic number types such as stochastic variables from the Measurements.jl package that could be used to propagate uncertainties in certain parameters (e.g., flip angles) throughout the simulations.

We observed that the EPG-based FISP2D sequence simulator from BlochSimulators.jl has similar runtime performance as the recurrent neural network RNN-EPG. At the same time, it does not require a training phase and there is no loss of accuracy that is inherent to trained networks. However, we still believe neural networks can play an important role in MR-STAT reconstructions. The RNN-EPG network was trained to be able to predict the signal for a wide range of varying flip angle trains. For scenario's where the fixed flip angle trains are fixed, a different network architecture<sup>25</sup> with better runtime performance may be more suitable and have better performance.

We released BlochSimulators.jl as a new toolbox rather than modifying/extending existing toolboxes (e.g., BlochSim.jl,<sup>27</sup> DECAES.jl,<sup>28</sup> KomaMRI.jl,<sup>13</sup> MRIEPG.jl,<sup>29</sup> MRIReco.jl,<sup>30</sup> MRIgeneralizedBloch.jl,<sup>31</sup> and SpinDoctor.jl<sup>32</sup>) to focus purely on runtime performance through custom, sequence specific (GPU) kernels. As demonstrated, our approach results in significantly faster runtimes as compared to KomaMRI.jl, which uses a generic sequence format combined with array operations on CuArrays for GPU support (resulting in many kernel launches per simulation) instead.

BlochSimulators.jl currently lacks some features which are available in other toolboxes such as a graphical user interface,<sup>13</sup> spin displacements during the



**FIGURE 7** In vivo  $T_1$ ,  $T_2$  and proton density maps reconstructed using the proposed partially matrix-free MR-STAT reconstruction algorithm on GPU hardware (left column) and the reference MR-STAT reconstruction from Reference 3 (middle column). The absolute relative error maps together with mean absolute relative errors are shown in the right column. The in vivo data was acquired using a gradient-balanced sequence with varying flip angles and Cartesian readouts.

**TABLE 1** Two-dimensional in vivo MR-STAT reconstruction times.

Method	Hardware	Time
Matrix-free <sup>3</sup>	96 CPUs	193 min
Sparse-Hessian <sup>5</sup>	96 CPUs	16 min
ADMM, neural network <sup>4</sup>	8 CPUs	2 min
Partially matrix-free	1 GPU	68 s

sequence,<sup>13</sup>  $T_2^*$ ,<sup>28</sup> diffusion,<sup>32</sup> and magnetization transfer.<sup>29,33</sup> We envision that in the future the best features of the currently separate toolboxes will be merged to arrive at a feature-complete, yet highly performant and flexible Julia Bloch simulation toolbox.

**TABLE 2** Breakdown of computationally demanding tasks for a two-dimensional in vivo MR-STAT reconstruction with the partially matrix-free algorithm executed on GPU hardware.

Task	Time	% of total
Forward model evaluation	6.22 s	9.1%
Partial derivatives at echo times	2.84 s	4.2%
Matrix-vector products with $\mathbf{J}$	44.1 s	64.9%
Matrix-vector products with $\mathbf{J}^H$	9.1 s	13.5%

To evaluate the forward model Equation (1), we proposed to first compute the magnetization at echo times in all voxels, followed by computing the magnetization at other readout times and evaluating the volume integral.

For some intra-voxel effects such as simulating  $T_2^*$  decay using multiple isochromats with different off-resonances within a voxel, this separation cannot be applied. Another important limitation is that for high-resolution 3D simulations, storing the magnetization at echo times in all voxels may require more memory than what is currently available on modern GPU cards. To deal with this limitation, the total number of voxels could be partitioned and the computations could be performed in batches (in parallel if multiple GPU cards are available). This functionality, however, is not yet included in BlochSimulators.jl.

The proposed partially matrix-free algorithm on GPU results in reconstruction times that are faster than the prior state-of-the-art ADMM MR-STAT reconstruction technique<sup>4</sup> (68 s vs. 2 min). An additional benefit is that, unlike the ADMM method, no Cartesian-based gradient encoding scheme is required and the proposed could therefore also be applied for non-Cartesian MR-STAT reconstructions.<sup>34</sup>

The reported 2D in vivo reconstruction time of 68 s is specific for the currently used acquisition and reconstruction setup. Increasing, for example, the number of samples during the acquisition, the number of isochromats per voxel (for slice profile correction purposes), or the number of receive coils results in increased computation times.

Despite the speedup achieved in this work, reconstruction times are still too long for online reconstruction in clinical practice. To further reduce reconstruction times, matrix-vector products with the Jacobian  $\mathbf{J}$  must be further optimized, for example by incorporating the difference GPU memory layers into the computations, using reduced precision or adapting the sparse Hessian technique.<sup>5</sup> In addition, faster, sequence-specific neural network architectures could be explored for accelerated Bloch simulations.<sup>25</sup> Automatic differentiation tools (e.g., Enzyme.jl<sup>35,36</sup> and Zygote.jl<sup>37</sup>) could potentially be used to accelerate derivative computations.

## ACKNOWLEDGMENTS

The authors are thankful for fruitful discussions with Christian Hundt, Oliver Kutter and Boris Bonev from NVIDIA. The authors thank the ISMRM Reproducible Research Study Group for conducting a code review of the code (BlochSimulators v0.2.3) supplied in the Data Availability Statement. The scope of the code review covered only the code's ease of download, quality of documentation, and ability to run, but did not consider scientific accuracy or code efficiency.

## DATA AVAILABILITY STATEMENT


The BlochSimulators.jl package is written in the open source language Julia and is available online at <https://github.com/oscarvanderheide/BlochSimulators.jl>. The package is also registered in the General registry of Julia's

package manager. The benchmarking scripts that were used to generate Figures 3–6 are found in the benchmarks folder of the repository. We used v0.2.7 of BlochSimulators.jl (commit 737d40d) in all benchmarks. An example implementation of the partially matrix-free MR-STAT reconstruction algorithm on GPU hardware is available at <https://github.com/oscarvanderheide/mrstat>. This code uses simulated data from a numerical brain phantom and assumes Cartesian readouts trajectories are used.

## ORCID

Oscar van der Heide  <https://orcid.org/0000-0002-8451-525X>

Cornelis A. T. van den Berg  <https://orcid.org/0000-0002-5565-6889>

Alessandro Sbrizzi  <https://orcid.org/0000-0003-3276-4542>

## REFERENCES

1. Ma D, Gulani V, Seiberlich N, et al. Magnetic resonance fingerprinting. *Nature*. 2013;495:187-192.
2. Sbrizzi A, Heide O, Cloos M, et al. Fast quantitative MRI as a nonlinear tomography problem. *Magn Reson Imaging*. 2018;46:56-63.
3. Van der Heide O, Sbrizzi A, Luijten PR, Van den Berg CAT. High resolution in-vivo MR-STAT using a matrix-free and parallelized reconstruction algorithm. *NMR Biomed*. 2020;33:e4251.
4. Liu H, Van der Heide O, Mandija S, Van den Berg CAT, Sbrizzi A. Acceleration strategies for MR-STAT: achieving high-resolution reconstructions on a desktop pc within 3 minutes. *IEEE Trans Med Imaging*. 2022;41:2681-2692.
5. van der Heide O, Sbrizzi A, van den Berg CAT. Accelerated MR-STAT reconstructions using sparse hessian approximations. *IEEE Trans Med Imaging*. 2020;39(11):3737-3748.
6. Stone SS, Haldar JP, Tsao SC, Hwu W-MW, Liang Z-P, Sutton BP. Accelerating advanced MRI reconstructions on GPUs. Paper presented at: Proceedings of the 5th Conference on Computing Frontiers, Ischia, Italy; 2008:261-272.
7. Wang H, Peng H, Chang Y, Liang D. A survey of GPU-based acceleration techniques in MRI reconstructions. *Quant Imaging Med Surg*. 2018;8:196-208.
8. Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: a fresh approach to numerical computing. *SIAM Rev*. 2017;59:65-98.
9. Besard T, Foket C, De Sutter B. Effective extensible programming: unleashing Julia on GPUs. *IEEE Trans Parallel Distrib Syst*. 2018;30(4):827-841.
10. Brian H. Bloch equation simulator. Accessed January 25, 2023. <http://mrsrl.stanford.edu/brian/blochsim/>
11. Stöcker T, Vahedipour K, Pflugfelder D, Shah NJ. High-performance computing MRI simulations. *Magn Reson Med*. 2010;64:186-193.
12. Wang D, Ostenson J, Smith DS. snapMRF: GPU-accelerated magnetic resonance fingerprinting dictionary generation and matching using extended phase graphs. *Magn Reson Imaging*. 2020;66:248-256.
13. Castillo-Passi C, Coronado R, Varela-Mattatall G, Alberola-López C, Botnar R, Irarrazaval P. KomaMRI. jl: an

- open-source framework for general MRI simulations with GPU acceleration. *Magn Reson Med.* 2023;90:329-342.
14. Liu F, Velikina JV, Block WF, Kijowski R, Samsonov AA. Fast realistic MRI simulations based on generalized multi-pool exchange tissue model. *IEEE Trans Med Imaging.* 2017;36:527-537.
  15. Paige CC, Saunders MA. LSQR: an algorithm for sparse linear equations and sparse least squares. *ACM Trans Math Softw.* 1982;8:43-71.
  16. Jaynes ET. Matrix treatment of nuclear induction. *Phys Rev.* 1955;98:1099-1105.
  17. Willem V. MR Pulse Design through Optimal Control and Model Order Reduction of the Bloch Equations. Master Thesis. Utrecht University. 2015.
  18. Jürgen H. Echoes—how to generate, recognize, use or avoid them in MR-imaging sequences. Part I: fundamental and not so fundamental properties of spin echoes. *Concepts Magn Reson.* 1991;3:125-143.
  19. Jürgen H. Echoes—how to generate, recognize, use or avoid them in MR-imaging sequences. Part II: echoes in imaging sequences. *Concepts Magn Reson.* 1991;3:179-192.
  20. Matthias W. Extended phase graphs: dephasing, RF pulses, and echoes-pure and simple. *J Magn Reson Imaging.* 2015;41:266-295.
  21. Lebel RM, Wilman AH. Transverse relaxometry with stimulated echo compensation. *Magn Reson Med.* 2010;64:1005-1014.
  22. Sbrizzi A, Bruijnen T, van der Heide O, Luijten P, van den Berg CAT. Dictionary-free MR Fingerprinting reconstruction of balanced-GRE sequences. 2017.
  23. Assländer J, Glaser SJ, Hennig J. Pseudo steady-state free precession for MR-fingerprinting. *Magn Reson Med.* 2017;77:1151-1161.
  24. Jiang Y, Ma D, Seiberlich N, Gulani V, Griswold MA. MR fingerprinting using fast imaging with steady state precession (FISP) with spiral readout. *Magn Reson Med.* 2015;74:1621-1631.
  25. Yang M, Jiang Y, Ma D, Mehta BB, Griswold MA. Game of learning Bloch equation simulations for MR fingerprinting. *arXiv preprint arXiv:2004.02270.* 2020.
  26. Liu H, Van der Heide O, Van den Berg CAT, Sbrizzi A. Fast and accurate modeling of transient-state, gradient-spoiled sequences by recurrent neural networks. *NMR Biomed.* 2021;34:e4527.
  27. BlochSim.jl. Accessed December 4, 2023. <https://github.com/StevenWhitaker/BlochSim.jl>
  28. Jonathan D, Christian K, Alexander R. DECAES - DEcomposition and component analysis of exponential signals. *Zeit Med Phys.* 2020;30(4):271-278.
  29. MRIEPG.jl. Accessed December 4, 2023. <https://github.com/felixhorgor/MRIEPG.jl>
  30. Knopp T, Grosser M. MRIReco.Jl: an MRI reconstruction framework written in Julia. *Magn Reson Med.* 2021;86(3):1633-1646.
  31. MRIgeneralizedBloch.jl. Accessed December 4, 2023. <https://github.com/JakobAsslaender/MRIgeneralizedBloch.jl>
  32. Li JR, Nguyen VD, Tran TN, et al. SpinDoctor: a MATLAB toolbox for diffusion MRI simulation. *Neuroimage.* 2019;202:116120.
  33. Assländer J, Gultekin C, Flassbeck S, Glaser SJ, Sodickson DK. Generalized Bloch model: a theory for pulsed magnetization transfer. *Magn Reson Med.* 2022;87:2003-2017.
  34. Van der Heide O, Sbrizzi A, Van den Berg CAT. Cartesian vs radial MR-STAT: an efficiency and robustness study. *Magn Reson Imaging.* 2023;99:7-19.
  35. Moses W, Churavy V. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In: Larochelle H, Ranzato M, Hadsell R, Balcan MF, Lin H, eds. *Advances in Neural Information Processing Systems.* Curran Associates, Inc.; 2020:12472-12485.
  36. Moses WS, Churavy V, Paehler L, et al. Reverse-Mode Automatic Differentiation and optimization of GPU kernels via enzyme. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC 21.* Association for Computing Machinery; 2021.
  37. Innes M. Don't unroll adjoint: differentiating SSA-form programs. *CoRR.* 2018:abs/1810.07951.

## SUPPORTING INFORMATION

Additional supporting information may be found in the online version of the article at the publisher's website.

**S1:** Partially matrix-free 2D MR-STAT memory estimate

**S2:** Isochromat simulation details

**S3:** Extended Phase Graph simulation details

**S4:** Forward model simulation details

**Figure S1:** The time-varying flip angle train [top] and phase encoding indices for the Cartesian trajectory [bottom] used in the in-vivo MR-STAT data acquisition as well as the dictionary/signal simulations.

**How to cite this article:** van der Heide O, van den Berg CAT, Sbrizzi A. GPU-accelerated Bloch simulations and MR-STAT reconstructions using the Julia programming language. *Magn Reson Med.* 2024;92:618-630. doi: 10.1002/mrm.30074