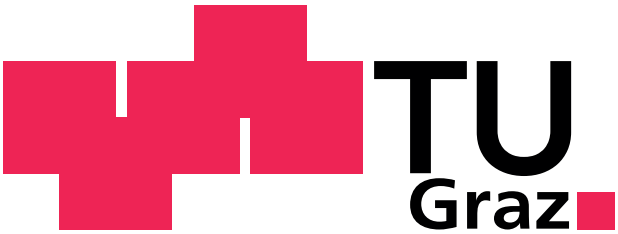
Markus Bödenler

Parallel Computing for Optimal Control

RF Pulse Design

Master Thesis



Institute for Medical Engineering
Technical University of Graz
Kronesgasse 5/II
A-8010 Graz
Head of the Institute: Univ.-Prof. Dr.techn. Dipl.-Ing. Rudolf Stollberger

Supervisor: Dipl.-Ing. Christoph Aigner
Evaluator: Univ.-Prof. Dipl.-Ing. Dr.techn. Rudolf Stollberger

Graz, (January 15, 2016)

Danke, Mama und Papa

Danke, Nidi

Danke, Christoph und Prof. Stollberger

Danke, meinen Kollegen vom Institut für Medizintechnik der TU Graz

Statutory Declaration

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

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Abstract

In this work the inherent spatial parallelism of the full time dependent Bloch equation for optimal control based RF pulse design is exploited to accelerate the optimization by means of parallel computing. The major bottlenecks in the provided MATLAB framework are implemented utilizing the MATLAB executable (MEX) interface using sequential C/C++ code, OpenMP CPU multi-threading and CUDA GPU-computing. The demonstrated implementations lead to a significant reduction in computing time while maintaining the high flexibility of the MATLAB environment. In particular, the CUDA implementation allows for optimization times in the order of a few seconds making real-time optimization and patient-specific design feasible. An evaluation of the generated RF pulses indicates no significant loss in accuracy with respect to the MATLAB implementation.

Keywords: RF pulse design, optimal control, simultaneous multi-slice excitation, parallel computing, general-purpose computing on graphics processing unit

Kurzfassung

Im Rahmen dieser Arbeit wird die örtliche Unabhängigkeit der Bloch Gleichung, zum Design von HF-Pulsen mittels Optimal Control Ansatz, ausgenützt, um mit Hilfe von Parallel Computing Methoden eine Beschleunigung der Optimierung zu erreichen. Die berechungsintensiven Funktionen des zur Verfügung gestellten MATLAB Frameworks werden unter Verwendung der MATLAB executable (MEX) Schnittstelle als sequenzielle C/C++, OpenMP Multi-Threading und CUDA GPU Computing Versionen implementiert. Dies führt zu einer signifikanten Reduktion der Optimierungsdauer unter Aufrechterhaltung der hohen Flexibilität der MATLAB-Umgebung. Insbesondere reduziert die CUDA Implementierung die Optimierung auf einige Sekunden, wodurch patientenspezifisches Design, in Echtzeit, ermöglicht wird. Eine Evaluierung der berechneten HF-Pulse zeigt keinen signifikanten Verlust an Genauigkeit gegenüber der Implementierung in MATLAB.

Schlüsselwörter: HF-Pulsdesign, Optimal Control, Simultaneous Multi-Slice Excitation, Parallel Computing, General-Purpose Computing on Graphics Processing Unit

Contents

1. Introduction **2**

1.1. Objective 4

2. Background **5**

2.1. Excitation problem and RF Pulse Design 5

2.1.1. Bloch Equation 5

2.1.2. Single Slice Excitation 6

2.1.3. Rectangular Pulses 9

2.1.4. SINC Pulses 10

2.1.5. SLR Pulses 11

2.1.6. Simultaneous Excitation of Multiple Slices 13

2.1.7. OC Pulses 18

2.2. Parallel Computing 24

2.2.1. NVIDIA CUDA 24

2.2.2. OpenMP 31

3. Methods **33**

3.1. Matlab Framework 33

3.2. CMake Project Structure 37

3.3. Thrust Library 39

3.4. Sequential C/C++ Implementation 40

3.5. Parallel CUDA C/C++ Implementation 48
3.6. Parallel OpenMP Implementation 53

4. Results **55**
4.1. Optimization Time 55
4.2. Problem Size Dependency 58
4.3. Single vs. Double 60
4.3.1. Speed 60
4.3.2. Accuracy 62
4.4. CUDA Implementation Profiler Results 67
4.5. CUDA Kernel Execution Time 68

5. Discussion and Conclusion **70**

Bibliography **78**

Appendix **84**
A. GPU Specifications 84
B. CUDA Kernels 85
B.1. `applyHessGPU` 85
B.2. `cn_blochGPU` 89
B.3. `cn_adjointGPU` 91

Index of Abbreviations

MRI Magnetic Resonance Imaging

RF radio frequency

SLR Shinnar-Le Roux

PM Parks-McClellan

FIR finite impulse response

TR repetition time

TE echo time

SMS simultaneous multi-slice

SNR signal to noise ratio

SENSE sensitivity encoding

CAIPIRINHA controlled aliasing in parallel imaging results in higher acceleration

EPI echo planar imaging

SAR specific absorption rate

PINS power independent number of slices

VERSE variable rate selective excitation

pTx parallel transmission

TSE turbo spin echo

OC optimal control

CG conjugate gradient

API application interface

MPI Message Passing Interface

OpenMP Open Multi-Processing

GPGPU general-purpose computing on graphics processing unit

CUDA Compute Unified Device Architecture

OpenCL Open Computing Language

SM streaming multiprocessor

SIMT Single-Instruction, Multiple-Thread

STL Standard Template Library

BLAS Basic Linear Algebra Subroutines

RMSE root-mean-square error

MAE mean absolute error

FLASH fast low angle shot

1. Introduction

We live in a highly parallel world. Numerous complex events are happening at the same time, in parallel. The attempt to understand and to model real world phenomena is as old as science itself. Realistic modelling and simulation of physics effects poses innumerous problems, whose computational requirements are so demanding that it is infeasible to solve them in reasonable time on a single computer [1]. In the age of information the increase in computing performance is crucial. Over the last decades the vast majority of applications are written as sequential programs expecting that each new generation of microprocessors comes with an increase in computing performance. We now have reached a point where this expectation is no longer strictly valid and parallelization techniques are essential to maintain this increase [2]. Many problems provide an inherent parallelism one can exploit to reduce computing time and thus parallel computing has become increasingly important in the last decade. Parallel computing utilizes the divide and conquer principle and splits a large-scale problem into smaller independent sub-problems that can be solved concurrently [3]. There exist a wide variety of available programming frameworks. The Message Passing Interface (MPI) standard is a model used for distributed memory systems, e.g., computing cluster, where all interaction and data sharing must be done via message passing [2]. The OpenMP (Open Multi-Processing) framework is designed for shared memory systems and consists of a set of compiler directives, environment variables and library functions to express and control parallelism [4]. General-purpose computing on graphic processing units (GPGPU) is the use of the graphics processing unit (GPU) for comput-

ing problems originally performed on the central processing unit (CPU). Traditionally, the hardware architecture of a GPU is designed to perform a massive amount of floating point operations per video frame as required in video processing and advanced video games. Therefore, the GPU architecture is highly specialized for parallel processing and even a single heterogeneous platform (CPU-GPU system) provides remarkable enhancements in computing performance, when the computational intensive parts are moved to the GPU for execution [2]. One of the first applications of general-purpose computations on a GPU was proposed to calculate large matrix-matrix products [5]. The Compute Unified Device Architecture (CUDA) programming model enables the use of general-purpose computing on NVIDIA GPUs [6]. A vendor independent platform is the OpenCL (Open Computing Language) framework. GPU computing can be utilized in a broad range of applications such as in Magnetic Resonance Imaging (MRI) [7, 8].

MRI is a noninvasive medical imaging modality and has a wide range of applications in medical diagnosis. Selective radio frequency (RF) pulses, together with a slice selection gradient, are used for spatial excitation of a distinct slice in the subject. During the design process of RF pulses a main question arises: What RF pulse should be applied to reach the desired magnetization profile? Due to the bilinearity of the Bloch equation it is very difficult to obtain a closed-form solution for an arbitrary pulse excitation [9]. For small tip angle excitation the RF envelope can be approximated sufficiently by the inverse Fourier transform of the desired slice profile. As a result of the bilinearity of Bloch's equation this approximation only holds for small tip angles ($< 30°$) but remains reasonable up to excitations of $90°$ [10]. Therefore, the use of special techniques to reduce the resulting excitation error becomes indispensable when an accurate slice profile, in combination with large flip angles, is needed. Several methods have been proposed in the literature, e.g., the Shinnar-Le Roux (SLR) selective excitation pulse design algorithm [11] or optimization methods based on optimal control (OC) theory [12]. The SLR algorithm transforms the problem of RF pulse design to the problem of digital filter design, which can be solved using sophisticated digital filter design algorithms (e.g. Parks-McClellan algorithm [11]). Optimal control methods optimize a proper modelled objective function with subject to the

Bloch equation describing the evolution of the magnetization vector in an external magnetic field [12]. However, OC approaches usually suffer from high computational effort. Recently, Aigner et al. proposed an efficient implementation of the OC approach applied to simultaneous multi-slice pulse design [13]. In particular, they introduced a matrix-free Newton-Krylov method using exact first- and second-order derivatives obtained by means of adjoint calculus. In addition, they embedded the matrix-free Newton-Krylov method in a Steihaug trust-region framework achieving global convergence to a local minimizer [14]. The numerical solution of this optimal control problem requires a discretization of the Bloch equation, adjoint equation and objective function in the time and spatial domain. In each spatial discretization point the Bloch and adjoint equation can be solved independently and in parallel. This underlying parallelism makes the algorithm highly suited for a parallel implementation.

1.1. Objective

The aim of this work is to exploit the intrinsic parallelism of the matrix-free trust-region Newton-Krylov algorithm presented in the work, *"Efficient high-resolution RF pulse design applied to simultaneous multi-slice excitation"* [13], by means of parallel computing. The major bottlenecks in the provided MATLAB framework are supposed to be identified and implemented utilizing the MATLAB executable interface, while maintaining the high flexibility of the MATLAB environment. Different parallel programming frameworks should be applied, whereas the focus lies on general-purpose computing on graphic processing units. These implementation methods are to be evaluated and compared with regard to the achievable speedup in computing performance and accuracy of the obtained optimization results depending on the underlying floating point precision.

2. Background

2.1. Excitation problem and RF Pulse Design

Selective RF pulses in conjunction with a slice selective gradient are used for excitation of a distinct slice in the subject. During the design process of RF pulses a main question arises: What RF pulse should be applied to reach the desired magnetization profile? The impact of a general RF pulse on the magnetization profile can be calculated by solving the Bloch equation and is called the *forward problem* of RF pulse design. The *inverse problem* denotes the computation of a RF pulse shape $B_1(t)$ for a given target magnetization profile. However, it is very difficult to obtain a closed-form solution for an arbitrary pulse excitation due to the bilinearity of the Bloch equation and thus numerical techniques are used to find an approximate solution [9].

2.1.1. Bloch Equation

The general behaviour of the magnetization vector $\mathbf{M}(t)$ in presence of an external magnetic field $\mathbf{B}(t)$ is governed by the Bloch equation

$$\frac{d\mathbf{M}(t)}{dt} = \gamma \mathbf{B}(t) \times \mathbf{M}(t) + \mathbf{R}(\mathbf{M}(t)) \ , \tag{2.1}$$

where γ denotes the gyromagnetic ratio. The relaxation term $\mathbf{R}(\mathbf{M}(t))$ is given by

$$\mathbf{R}(\mathbf{M}(t)) = -\frac{M_x(t)\hat{\mathbf{e}}_\mathbf{x} + M_y(t)\hat{\mathbf{e}}_\mathbf{y}}{T_2} - \frac{(M_z - M_0)\hat{\mathbf{e}}_\mathbf{z}}{T_1} \ , \tag{2.2}$$

with equilibrium magnetization M_0 and relaxation times T_1 and T_2. The vectors $\hat{\mathbf{e}}_\mathbf{x}$, $\hat{\mathbf{e}}_\mathbf{y}$ and $\hat{\mathbf{e}}_\mathbf{z}$ are unit vectors in x, y and z directions, respectively. The external magnetic field $\mathbf{B}(t)$ is composed of three components: the main static field B_0, the time-dependent radio frequency field $B_1(t)$ and the time-dependent gradient fields $G(t)$. With only a static field B_0 present in longitudinal direction (z-axis) the net magnetization vector $\mathbf{M}$ is aligned along this axis. The resonance frequency of the spins is the Larmor frequency of hydrogen atoms. To create a signal in MRI a RF field $B_1(t)$ (at Larmor frequency) is applied for a short time period to rotate $\mathbf{M}$ away from the z-axis and the transverse projection can be measured. The shape of such a magnetic field is denoted as RF pulse. In the absence of applied gradient fields all spins show the same resonance frequency and are tipped *nonselectively*. A *selective* excitation can be attained when a gradient field is played concurrently with the RF pulse. [10, 15]

2.1.2. Single Slice Excitation

A basic approach for selective excitation of a plane is to apply a constant magnetic field gradient along the z-direction together with a RF pulse modulated by a function $B_1(t)$ (RF pulse shape). This gradient field is commonly referred to as slice selection gradient G_z, and varies the Larmor frequency linearly along the longitudinal direction resulting in a different effective magnetic field $\mathbf{B}_{eff}$ in each spatial point. Thus the Larmor frequency is a function of z location $\omega(z)$. A RF pulse, tuned to a certain resonance frequency ω_0, excites those spins located in z direction with resonance frequency matching the frequencies of $B_1(t)$. Spins with resonance frequencies outside the RF bandwidth remain unexcited and the resulting excited slice is perpendicular to G_z. The slice thickness Δz is proportional to the RF bandwidth and reciprocally proportional to the amplitude of the slice selection gradient. The location z_0 of the slice is determined by the carrier frequency ω_0 of the RF pulse (see figure 2.1).

For non-adiabatic RF pulses and in the on-resonance case the flip angle θ at time point τ (pulse duration) is given by the area under the RF envelope $B_1(t)$ (equation 2.3).

$$\theta(\tau) = \gamma \int_{t=0}^{\tau} B_1(t)dt \tag{2.3}$$

An ideal selective excitation pulse will lead to a slice profile with uniform flip angle in-slice and zero flip angle out-of-slice, i.e., a rectangular profile. Such an ideal slice profile cannot be reached in practice but effective approximations are available. [10, 15]

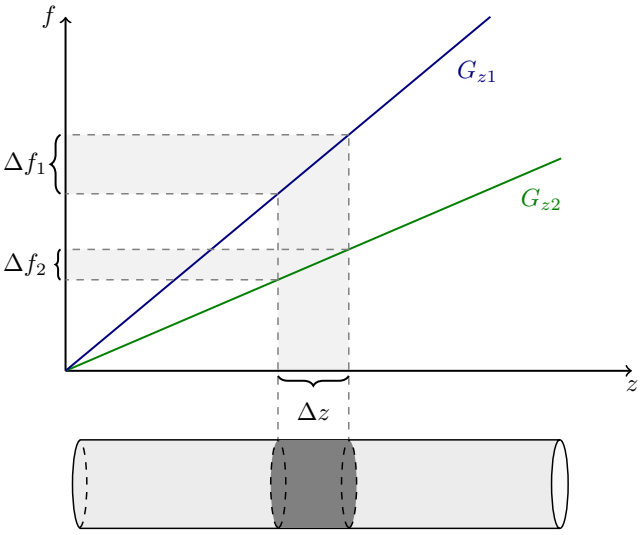


Figure 2.1.: The slice thickness Δz is proportional to the RF bandwidth Δf and inversely proportional to the slice selection gradient G_z.

Small Tip Angle Approximation In the small tip angle case the approximations $M_z \approx M_0$ and $dMz/dt \approx 0$ are used to simplify the Bloch equation (Eq. 2.1). Furthermore, relaxation effects are neglected because the RF pulse is short (i.e., $T_1,\ T_2 >> \tau$) and a constant z-gradient G_z is applied. Defining the complex transverse magnetization

$$M_{xy} = M_x + iM_y \ , \tag{2.4}$$

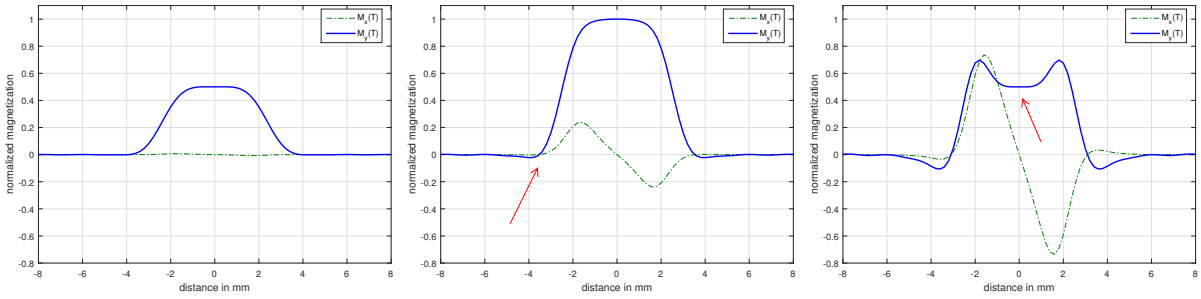
the decoupled transverse components merge into a first-order linear differential equation

$$\frac{dM_{xy}}{dt} = -i\gamma G_z z M_{xy} + i\gamma B_1(t) M_0 \ . \tag{2.5}$$

Solving equation 2.5 the magnitude of the transverse magnetization is given by

$$|M_{xy}(\tau, z)| = \gamma M_0 \mathcal{F}_{1D} \left\{ B_1 \left(t + \frac{\tau}{2} \right) \right\} \Big|_{f=-(\gamma/2\pi)G_z z} \ , \tag{2.6}$$

where $B_1(t)$ has a pulse duration from $t = 0$ to τ and $f = -(\gamma/2\pi)G_z z$ denotes the frequency at each z-position. Equation 2.6 states the Fourier transform relationship between $B_1(t)$ and the slice profile. For small tip angles the Fourier transform is a sufficient description and $B_1(t)$ can be designed using the inverse Fourier transform of the desired slice profile. For a detailed derivation of this relation the reader is referred to [10, Chapter 6.2.2]. The RF pulse of choice, for a rectangular slice profile, would be SINC-shaped and of infinite duration, but this is not feasible in practice. This approximation only holds for small tip angles ($< 30°$) but remains reasonable up to excitations of 90°. Above 90° severe excitation error emerges. Figure 2.2 depicts the transverse magnetization response to a Hamming windowed SINC-pulse with tip angles of 30°, 90° and 150°. [10, 15]



(a) $\theta = 30°$ (b) $\theta = 90°$ (c) $\theta = 150°$

Figure 2.2.: Validity of the small tip angle approximation. Shows the transverse magnetization components M_y (blue solid) and M_x (green dashed) in response to a Hamming windowed SINC-pulse $B_1(t)$ at tip angles of $\theta = 30°$, 90° and 150°. The Fourier approximation holds for $\theta = 30°$. At $\theta = 90°$ and 150° excitation errors are noticeable (red arrows).

2.1.3. Rectangular Pulses

A rectangular or hard pulse is defined by

$$B_1(t) = B_1 \; RECT\left(\frac{t}{\frac{\tau}{2}}\right) = \begin{cases} 1 & \text{if } |t| < \frac{\tau}{2} \\ 0 & \text{if } |t| > \frac{\tau}{2} \end{cases} , \tag{2.7}$$

where τ is the pulse duration (width) and B_1 the amplitude. The flip angle θ is proportional to the product of the width and amplitude of the pulse resulting in

$$\theta = \gamma B_1 \tau \; . \tag{2.8}$$

As a result from the small tip angle approximation the slice profile of a hard pulse is the corresponding SINC function. A short pulse width results in a wide bandwidth of excited spins. Therefore, hard pulses are used in the case of nonselective excitation. [15]

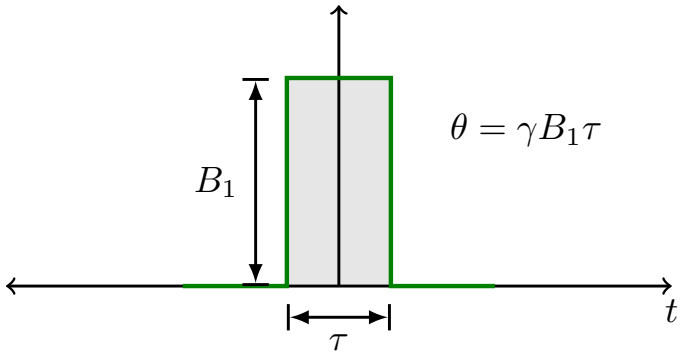


Figure 2.3.: Hard pulse with amplitude B_1 and pulse width τ. The flip angle is given by $\theta = \gamma B_1 \tau$

2.1.4. SINC Pulses

SINC pulses are widely used for selective excitation, saturation and refocusing. The mathematical description of the pulse shape is given by

$$B_1(t) = \begin{cases} B_1 \; SINC\left(\frac{\pi t}{t_0}\right) \equiv B_1 \; t_0 \frac{\sin\left(\frac{\pi t}{t_0}\right)}{\pi t} & \text{if } -N_L t_0 \leq t \leq N_R t_0 \\ 0 & \text{elsewhere ,} \end{cases} \tag{2.9}$$

where B_1 is the RF peak amplitude (at $t = 0$), N_R and N_L are the number of zero crossings to the right and left, respectively, and t_0 is one half the width of the central lobe. The Fourier transform of an infinitely long SINC pulse is the RECT function (eqiation 2.6). In practice, the SINC pulse is truncated after a few side lobes. The greater the number of side lobes, the better the approximation of the ideal magnetization profile. The truncation of the SINC pulse leads to ringing effects in the slice profile. A window function (e.g Hamming or Hanning window) is usually applied to the SINC pulse to smooth the slice profile. Figure 2.4 shows a symmetric SINC pulse with and without Hamming window. [15]

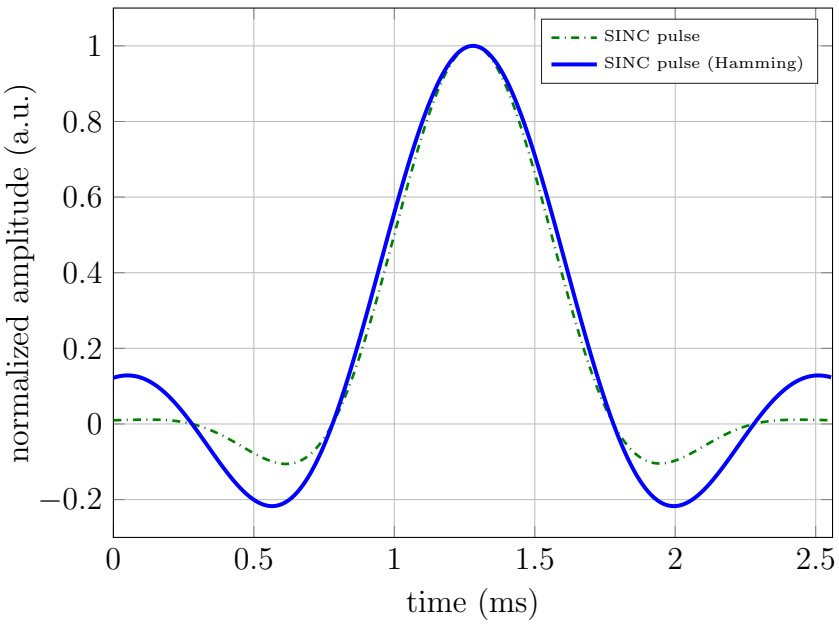


Figure 2.4.: SINC pulse with (blue solid) and without (green dashed) Hamming window.

2.1.5. SLR Pulses

In the case of large flip angles the small tip angle approximation does not provide a sufficiently accurate slice profile (figure 2.2) and more sophisticated methods such as the Shinnar-Le Roux transformation are required [11]. Originally, the SLR algorithm was limited to flip angles of 90° and 180°, but Lee [16] generalized this approach to arbitrary flip angles using an exact parameter relation. The SLR algorithm uses two key concepts: the hard pulse approximation and the spin-domain representation of rotation (spinor notation). The RF pulse $B_1(t)$ is approximated as a sequence of hard pulses with pulse width Δt, each resulting in a small rotation of the magnetization vector. Based on the spinor notation the total rotation of the magnetization vector can be described by two z-transform polynomials $A_N(z)$ and $B_N(z)$. This is denoted as *forward SLR transform*. The polynomials can be obtained by using filter design algorithms such as the Parks-McClellan (PM) algorithm. The computation of the RF pulse from the filter polynomials is known as the *inverse SLR transform*. Therefore, the SLR technique maps the RF pulse design problem to a digital filter design problem. The basic workflow of RF pulse design via SLR transform is illustrated in Figure 2.5.

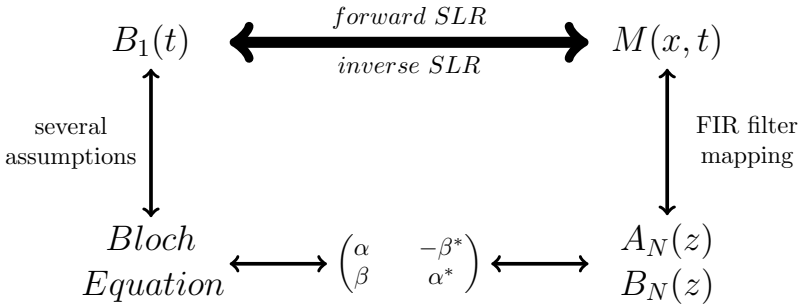


Figure 2.5.: Basic SLR workflow for optimized RF pulse design. A given RF pulse is mapped into the corresponding filter polynomials $A_N(z)$ and $B_N(z)$ via the forward SLR transform. The inverse SLR transform computes the RF pulse from the given polynomials.

FIR filter design of $A_N(z)$ and $B_N(z)$ The transformation of RF pulse design to digital filter design gives access to a wide range of sophisticated tools, e.g., the PM algorithm (proposed in [11]) or finite impulse response (FIR) filter of least-square type. The desired slice profile is approximated by a filter polynomial $B_N(z)$. The PM algorithm requires the specification of the passband (in-slice) edge F_p and stopband (out-of-slice) edge F_s. Additionally, the passband and stopband ripple amplitudes needs to be defined by δ_1 and δ_2, respectively. Figure 2.6 depicts the key FIR filter parameters.

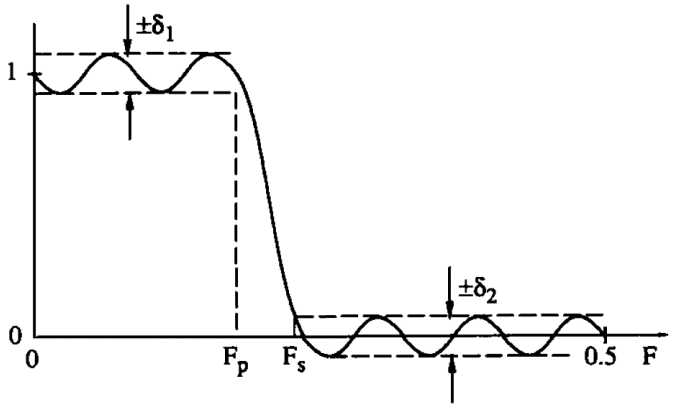


Figure 2.6.: Approximation of the target magnetization profile via FIR filter design with filter parameters: in-slice ripple δ_1, out-of-slice ripple δ_2, passband edge F_p and stopband edge F_s.[11]

Subsequently, the corresponding polynomial $A_N(z)$ is calculated from $B_N(z)$. The magnitude of $A_N(z)$ is given by

$$|A_N(z)| = \sqrt{1 - B_N(z)B_N^*(z)} \; . \tag{2.10}$$

Pauly et al. [11] constrains $A_N(z)$ to be a minimum-phase polynomial and therefore an analytic signal. Consequently, the minimum-phase $A_N(z)$ can be derived from

$$A_N(z) = |A_N(z)| \;\; exp(i \; \mathcal{H} \{\log |A_N(z)|\}) \; , \tag{2.11}$$

where $\mathcal{H}\{\cdot\}$ denotes the Hilbert transform operator. Given the polynomials $A_N(z)$ and $B_N(z)$ the RF pulse is obtained by means of the inverse SLR transform. Pulses designed with the SLR algorithm account for the nonlinearity of the bloch equation, but relaxation effects are neglected. In addition, SLR pulses are sensitive to B_1 inhomogeneities. Despite its limitations the SLR algorithm found widespread use in large flip angle pulse design [16–19].

2.1.6. Simultaneous Excitation of Multiple Slices

Multislice imaging is a standard technique for accelerating image acquisition. In traditional multislice imaging, additional slices are acquired in one repetition period (TR) using single slice excitation. A strategy to achieve further acceleration is to excite multiple slices simultaneously. This technique is called simultaneous multi-slice (SMS) imaging. This section gives a brief overview of the developments in SMS imaging. For a detailed review the reader is referred to Uğurbil et al. [20] and Feinberg et al. [21].

Early work ...

The use of simultaneous multi-slice acquisition dates back to 1988. Müller proposed a method that utilizes multifrequency selective RF pulses [22]. Such pulses are synthesized in a way that each frequency component has an individual phase, and hence tagging each slice by a unique phase information making the signals from each slice separable. Also in 1988 Souza et al. published a method for SMS acquisition using binary-encoded excitation [23]. Encoding in the slice dimension is achieved by modulating the phase of each slice in a binary pattern given by the Hadamard matrix.

... the step to parallel imaging ...

In 2001 Larkman and his co-workers proposed SMS imaging utilizing multicoil arrays,

in conjunction with sensitivity encoding (SENSE) image reconstruction, to separate the simultaneously acquired slices [24]. This technique is strongly dependent on the geometry of the multicoil array, and suffers from reduced signal to noise ratio (SNR), whenever the coil array sensitivities are similar in the excited slices. Breuer et al. presented the CAIPIRINHA (controlled aliasing in parallel imaging resutls in higher acceleration) technique to reduce the dependency of the coil array geometry by modifying the appearance of aliasing artefacts during acquisition [25]. The phase of the individual slices in the RF excitation pulse is modulated for each k-space line resulting in a shift of the simultaneously acquired slices relative to each other (in phase encoding direction). In Figure 2.7 a shematic description of a two-slice experiment is shown.

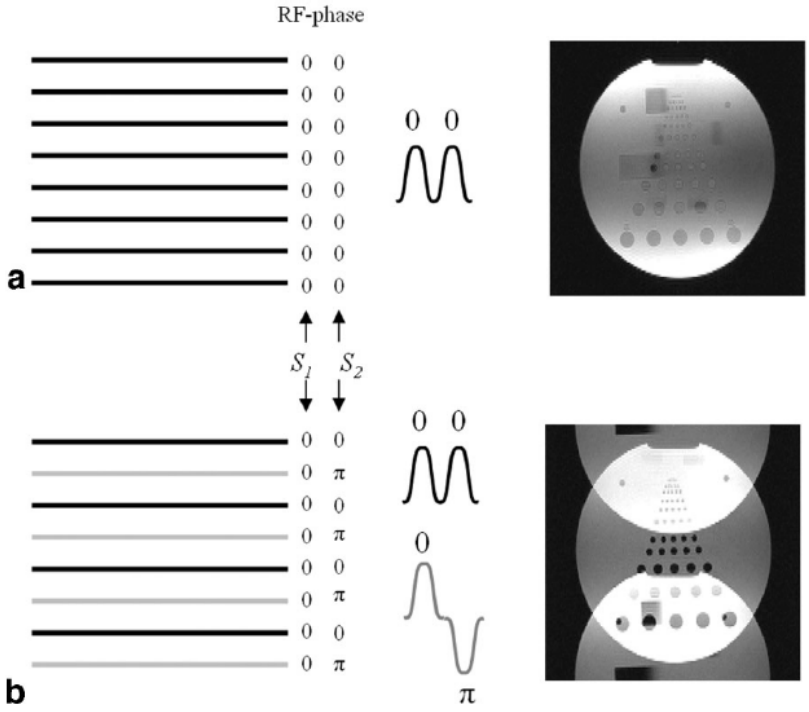


Figure 2.7.: Schematic description of a simultaneous two-slice CAIPIRINHA experiment without (a) and with (b) phase modulation. Odd k-space lines (black lines) are excited with a dual-band RF pulse with same phase (0,0) for both slices. Even k-space lines (grey lines) are the result of excitation with different phases (0,π). The individual slices are shifted with respect to each other in the superimposed image.[25]

... the step to human brain imaging ...

The CAIPIRINHA technique is not directly applicable to single shot echo planar imaging (EPI) due to the use of only one single excitation pulse. The use of SMS in single shot EPI for human brain imaging was first proposed by Nunes et al. [26]. The in plane shift in phase encoding direction is achieved by applying slice selection gradient blips concurrent with the EPI phase encoding blips; this leads to undesirable voxel tilting artifacts. To overcome the voxel tilt problem Setsompop et al. extended the method of Nunes et al. [26] to the blipped-CAIPI technique using sign and amplitude modulated slice-select gradient blips [27]. Figure 2.8 shows the pulse sequence diagram of a SMS single shot EPI sequence.

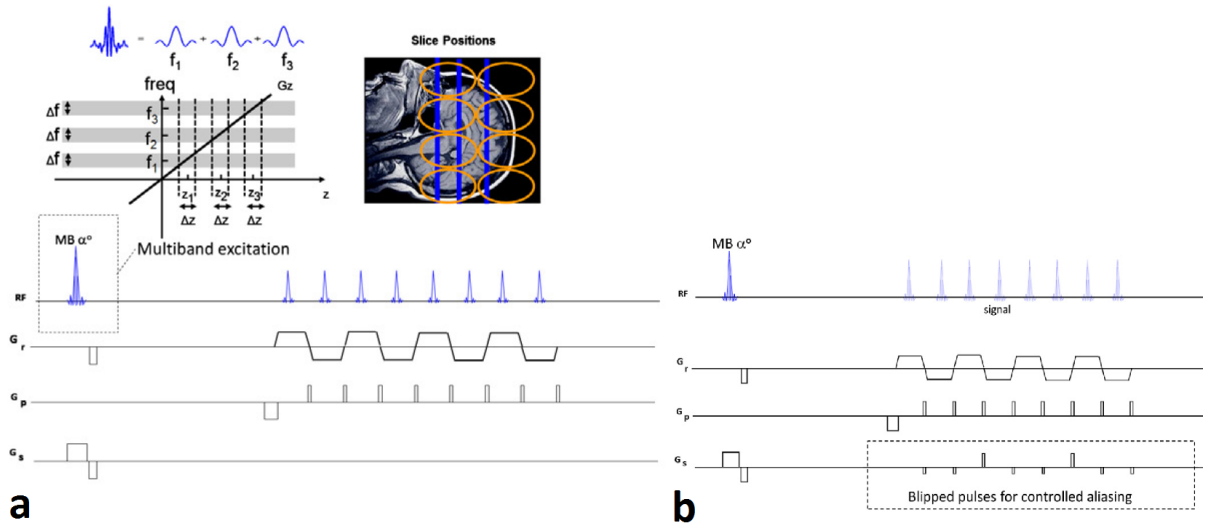


Figure 2.8.: Pulse sequence diagram for SMS single shot EPI. SMS excitation is achieved with a multiband pulse consisting of 3 frequency bands and therefore exciting 3 slices along the slice selection gradient axis. After the excitation a EPI readout follows (a). The use of sign and amplitude modulated slice select gradient blips in blipped-CAIPI is shown in (b).[21]

... image reconstruction ...

Image domain SENSE reconstruction is directly applicable to SMS imaging [24]. However, the k-space based GRAPPA reconstruction method is less straightforward to adapt. A SENSE/GRAPPA combination technique can be used to reconstruct slice-aliased data, but this method suffers from aliasing artifacts for CAIPIRINHA based data acquisition [28]. Therefore, Setsompop et al. proposed the slice-GRAPPA algorithm in combination with the blipped-CAIPI method [27]. The k-space data for each slice is estimated by applying a set of GRAPPA kernels to the slice-aliased k-space data. For each slice one GRAPPA kernel set is needed, and the kernels are fitted from a prescan calibration dataset acquired one slice at a time (see Figure 2.9).

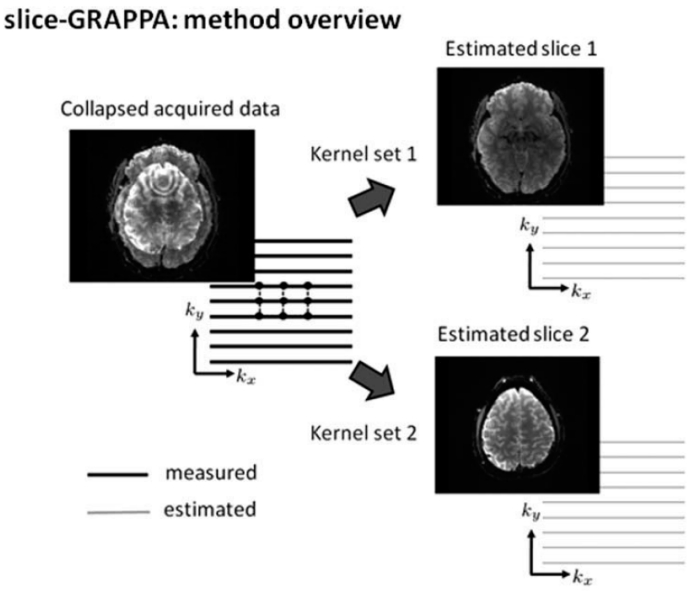


Figure 2.9.: Basic overview of the slice-GRAPPA algorithm. The k-space data of the unaliased slices is estimated by applying GRAPPA-like kernel sets to the k-space of the collapsed slices. The kernels are fitted from a prescan calibration dataset acquired one slice at a time. Image taken from [27].

... RF power limitations ...

Conventional design of SMS pulses is based on the superposition of single slice sub-pulses to a multifrequency RF pulse. Increasing the number of slices (frequency bands) result in a linear increase in B_1 peak amplitude and hence a quadratic scaling of RF peak power and a linear overall RF power increase. For greater numbers of slices the B_1 peak amplitude will exceed the hardware specifications of RF coils and power amplifiers. Furthermore, restrictions to the specific absorption rate (SAR) limit the use of such RF pulses. One way to address the increase of RF power is the power independent number of slices (PINS) method [29]. PINS pulses are generated by multiplication of an existing RF pulse by a Dirac comb function or alternatively, by Fourier series expansion. The PINS technique makes the RF power deposition independent of the number of slices. Likewise, variable rate selective excitation (VERSE) reduces the SAR in SMS applications [30]. An alternative approach is the use of parallel transmission (pTx) in combination with SMS excitation [31].

... recent developments

In 2015, Bilgic et al. introduced the wave-CAIPI 3D acquisition technique, using sinusoidal gradients (wave gradients) during each k_x encoding line [32]. This spreads the aliasing evenly in all spatial directions, thus benefiting from 3D coil sensitivity distribution. In addition, Gagoski et al. used the SMS wave-CAIPI acquisition method for turbo spin echo (TSE) imaging [33]. Guerin et al. demonstrated a SMS-pTx method with explicit control of local and global SAR [34]. Recently, Aigner et al. applied OC based pulse design for efficient SMS pulse optimization [13]. The use of a penalty term, modelling the SAR of the pulse, allows a trade-off between slice profile accuracy and RF power deposition, yielding RF pulses with reduced B_1 peak amplitude.

2.1.7. OC Pulses

The small tip angle approximation is not a valid approximation of the slice profile for large tip angles. Therefore, the use of optimization techniques, to increase slice profile accuracy, is gaining importance. Conolly et al. formulated the selective excitation problem as a dynamic optimization problem, with differential equations as constraints, i.e., as an optimal control (OC) problem [12]. Optimal control approaches involve the solution of the Bloch equation to model the evolution of the magnetization vector in presence of an external magnetic field. Additionally, relaxation effects or other desirable constraints (e.g., RF power) can be incorporated in the design model. Due to its flexible formulation the OC approach is increasingly used in MRI, in [12, 13, 35, 36].

2.1.7.1. Optimal Control Framework

In general, an optimal control is a minimizer to an objective function (i.e., a function of state and control variables) with subject to the state equation. The sate equation is a set of differential equations describing the paths of the control variables. In the context of RF pulse design the state equation is governed by the Bloch equation (see Section 2.1.1). Ignoring spatial field inhomogeneities, and for the on-resonance case, the matrix notation of equation (2.1) in the rotating reference frame is given by

$$\begin{cases} \dot{\mathbf{M}}(t,z) = \mathbf{A}(\mathbf{u}(t),z)\mathbf{M}(t,z) + \mathbf{b}(z), \\ \mathbf{M}(0,z) = \mathbf{M^0}(z), \end{cases} \tag{2.12}$$

with

$$\mathbf{A}(\mathbf{u}(t),z) = \begin{pmatrix} -\frac{1}{T_2} & \gamma G_z(t,z) & \gamma u_y(t) B_1 \\ -\gamma G_z(t,z) & -\frac{1}{T_2} & \gamma u_x(t) B_1 \\ -\gamma u_y(t) B_1 & -\gamma u_x(t) B_1 & -\frac{1}{T_1} \end{pmatrix} \quad \text{and} \quad \mathbf{b}(z) = \begin{pmatrix} 0 \\ 0 \\ \frac{M_0}{T_1} \end{pmatrix}. \tag{2.13}$$

The control $\mathbf{u}(t) = (u_x(t), u_y(t))^T$ denotes the RF pulse, G_z the slice selection gradient and $\mathbf{M^0}$ the initial magnetization vector. The OC approach in [13] computes the optimal control $\mathbf{u}(t)$, $t \in [0, T_u]$, that is a minimizer to

$$\underset{(\mathbf{u},\mathbf{M}) \text{ satisfying Eq. 2.12}}{min} J(\mathbf{u}, \mathbf{M}) = \frac{1}{2} \int_{-a}^{a} |\mathbf{M}(T, z) - \mathbf{M_d}(z)|_2^2 \, \mathrm{d}z + \frac{\alpha}{2} \int_0^{T_u} |\mathbf{u}(t)|_2^2 \, \mathrm{d}t \ . \quad (2.14)$$

To get a close fit to the desired slice profile the error function is modelled with the L^2 norm of the error, at read-out time $T > T_u$, between the solution to the state equation $\mathbf{M}(T, z)$ and the target magnetization $\mathbf{M_d}(z)$ for all $z \in [-a, a]$. A quadratic penalty term incorporates the SAR of the RF pulse in the design model, and the weighting parameter $\alpha > 0$ balances the trade-off between slice profile accuracy and SAR restriction.

2.1.7.2. Numerical Solution

Optimization problems can be solved iteratively using various numerical algorithms, and a wide range of literature is available on this topic, e.g., [37] and [38]. In gradient based optimization, there are two fundamental iterative approaches for moving from the actual point $\mathbf{u}^k$ to the iterate $\mathbf{u}^{k+1}$. One is the *line search* approach and the other is the *trust region* strategy. The basic iteration step is given by

$$\mathbf{u}^{k+1} = \mathbf{u}^k + s^k \mathbf{p}^k \ , \quad (2.15)$$

where $\mathbf{p}^k$ is the search direction and s^k is the step size. Hence, two major issues arise: the determination of the search direction $\mathbf{p}^k$ and the choice of a suitable step size s^k. Trust region and line search methods differ from each other in the way they choose the direction and step size from the actual iterate to the next.

The trust region approach first chooses a maximum distance, the *trust region radius*, which defines a region around the current iterate, the *trust region*. In this region the model is trusted to be an acceptable representation of the objective. Then the step is chosen to be a suitable minimizer to the model in this region. If a step is acceptable, the trust region will be extended, and for non-acceptable steps the trust region will be reduced.
On the other hand, line search methods first generate a search direction and then they attempt to find a suitable step size along this direction. The search direction can be computed by various methods, such as gradient methods, Newton methods, Quasi-Newton methods and Krylov methods (e.g. conjugate gradient (CG)).

Gradient Methods Gradient methods make use of first derivative information. They compute for a given $\mathbf{u}^k$ the gradient $\mathbf{g}(\mathbf{u}^k)$ of $j(\mathbf{u}) = J(\mathbf{u}, \mathbf{M})$ and set the iterate $\mathbf{u}^{k+1} = \mathbf{u}^k - s^k\mathbf{g}(\mathbf{u}^k)$. One example for such a method is the steepest descent algorithm. It requires just the calculation of the gradient $\mathbf{g}(\mathbf{u}^k)$ and no second derivative information is needed. However, they usually suffer from slow convergence close to a minimizer.

Newton Methods In mathematical optimization, Newton methods are second-order methods and make use of second order derivative information, i.e., they compute the Hessian $\mathbf{H}(\mathbf{u}^k)$ of $j(\mathbf{u})$ at the actual point $\mathbf{u}^k$. Subsequently, one solve for the Newton step $\delta\mathbf{u}$ in

$$\mathbf{H}(\mathbf{u}^k)\delta\mathbf{u} = -\mathbf{g}(\mathbf{u}^k) \tag{2.16}$$

and update the iterate by $\mathbf{u}^{k+1} = \mathbf{u}^k + \delta\mathbf{u}$. Methods that use the Newton direction have fast quadratic convergence to a local minimizer. The major drawback is the expensive computation of the full Hessian $\mathbf{H}(\mathbf{u}^k)$ for practical applications. Additionally, when the Hessian is not positive definite, $\mathbf{H}(\mathbf{u}^k)$ may not be invertible, i.e., the Newton direction may not be defined.

Quasi-Newton Methods This methods avoid the direct computation of the Hessian. Instead they use an approximation of the Hessian, which is updated in each iteration, and one method to calculate the updates is the BFGS algorithm. Due to the lack of exact information Quasi-Newton methods lose the quadratic convergence but yet attain superlinear rate of convergence.

Krylov Methods Krylov subspace methods, such as the CG method, are approaches for solving large linear systems $\mathbf{Ax} = \mathbf{b}$. These methods require only the computation of matrix-vector products per iteration. In Newton's method a CG algorithm can be applied to solve for the Newton step $\delta\mathbf{u}$ in equation 2.16. Consequently, only the computation of the action of the Hessian $\mathbf{H}(\mathbf{u}^k)\mathbf{h}$ for a given direction $\mathbf{h}$ per iteration is required and not the individual elements of $\mathbf{H}(\mathbf{u}^k)$. This so-called *matrix-free* approach induces substantial savings in computational effort and memory requirements.

2.1.7.3. Trust-Region Newton-CG Method

The optimal control problem in equation 2.14 can be solved iteratively using various numerical algorithms. Aigner et al. present an efficient numerical algorithm to solve this OC problem with subject to the full time-dependent Bloch equation [13]. The Newton step in equation 2.16 is solved by applying the CG algorithm, thus, only the calculation of the action of the Hessian, for a given direction, is needed. A direct application of the Newton-CG method is not possible, as the Bloch equation is bilinear in the unknowns $\mathbf{u}$ and $\mathbf{M}$ and therefore the optimization problem is not convex. Consequently, the Hessian $\mathbf{H}(\mathbf{u}^k)$ may not be positive definite. To overcome this problem Aigner et al. embedded the matrix-free Newton-CG method in a trust-region framework achieving global convergence to a local minimizer. In addition, exact first and second order information is obtained by using adjoint calculus.

Adjoint Approach The gradient $\mathbf{g}(\mathbf{u}^k)$ and the action of the Hessian $\mathbf{H}(\mathbf{u}^k)\mathbf{h}$ can be determined exactly by means of adjoint calculus. The adjoint model, to the sate equation, is given by

$$\begin{cases} -\dot{\mathbf{P}}(t,z) = \mathbf{A}(\mathbf{u}(t),z)^T\mathbf{P}(t,z), & 0 \leq t < T \\ \mathbf{P}(T,z) = \mathbf{M}(T,z) - \mathbf{M_d}(z), \end{cases} \tag{2.17}$$

where $\mathbf{A}(\mathbf{u}(t),z)$ is defined in equation 2.13, and the initial value $\mathbf{P}(T,z)$ is the difference between the magnetization profile at $\mathbf{M}(T,z)$ and the target profile $\mathbf{M_d}(z)$ at read-out time T. The gradient of J is calculated exactly via forward integration and backward integration of the state and the adjoint equation, respectively. This yields

$$\mathbf{g}(\mathbf{u}^k) = \alpha\mathbf{u}(t) + \begin{pmatrix} \int_{-a}^{a}\mathbf{M}(t,z)\mathbf{A_1}\mathbf{P}(t,z)\,\mathrm{d}z \\ \int_{-a}^{a}\mathbf{M}(t,z)\mathbf{A_2}\mathbf{P}(t,z)\,\mathrm{d}z \end{pmatrix}, \quad 0 \leq t \ \leq T_u \tag{2.18}$$

with

$$\mathbf{A_1} = \gamma B_1 \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad \mathbf{A_2} = \gamma B_1 \begin{pmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \tag{2.19}$$

where $\mathbf{M}(t,z) = (M_x(t,z), M_y(t,z), M_z(t,z)^T$ is the solution to the state equation (forward in time) for $\mathbf{u} = \mathbf{u}^k$ and $\mathbf{P}(t,z) = (P_x(t,z), P_y(t,z), P_z(t,z)^T$ is the solution to the adjoint equation (backward in time). In order to compute the action of the Hessian one needs to solve the linearized state and adjoint equation. The linearized state equation is given by

$$\begin{cases} \delta\dot{\mathbf{M}}(t,z) = \mathbf{A}(\mathbf{u}^k,z)\delta\mathbf{M}(t,z) + \mathbf{A}'(\mathbf{h})\mathbf{M}, \\ \delta\mathbf{M}(0,z) = (0,0,0)^T, \end{cases} \tag{2.20}$$

with

$$\mathbf{A}'(\mathbf{h}) = \begin{pmatrix} 0 & 0 & h_y(t) \\ 0 & 0 & h_x(t) \\ -h_y(t) & -h_x(t) & 0 \end{pmatrix}, \tag{2.21}$$

and the corresponding linearized adjoint model is given by

$$\begin{cases} -\delta\dot{\mathbf{P}}(t,z) = \mathbf{A}(\mathbf{u}^k,z)^T\delta\mathbf{P}(t,z) + \mathbf{A}'(\mathbf{h})^T\mathbf{P}, \\ \delta\mathbf{P}(T,z) = \delta\mathbf{M}(T,z). \end{cases} \tag{2.22}$$

Integrating equation 2.20 yields $\delta\mathbf{M}$, the directional derivative of $\mathbf{M}$ with respect to $\mathbf{u}$. Correspondingly, $\delta\mathbf{P}$, the directional derivative of $\mathbf{P}$ with respect to $\mathbf{u}$, can be obtained by solving equation 2.22. Finally, the exact action of the Hessian can be obtained with

$$[\mathbf{H}(\mathbf{u}^k)\mathbf{h}](t) = \alpha\mathbf{h}(t) + \begin{pmatrix} \int_{-a}^{a}\delta\mathbf{M}(t,z)\mathbf{A_1}\mathbf{P}(t,z) + \mathbf{M}(t,z)\mathbf{A_1}\delta\mathbf{P}(t,z)\,\mathrm{d}z \\ \int_{-a}^{a}\delta\mathbf{M}(t,z)\mathbf{A_2}\mathbf{P}(t,z) + \mathbf{M}(t,z)\mathbf{A_2}\delta\mathbf{P}(t,z)\,\mathrm{d}z \end{pmatrix}. \quad 0 \leq t \ \leq T_u \tag{2.23}$$

Discretization The numerical solution of the optimal control problem requires a discretization of the Bloch equation and objective function in time and spatial domain. In [13] the time interval $[0,T]$ is discretized by the time grid $0 = t_0 < ... < t_{N_t} = T$ with spacing $\Delta t_m := t_m - t_{m-1}$. The spatial domain $[-a,a]$ is discretized by a spatial grid $-a = z_1 < ... < z_{N_z} = a$ with spacing $\Delta z_m := z_m - z_{m-1}$. A Crank-Nicolson method (finite difference method) is used for discretization of the Bloch equation. For a detailed description of the discretization the reader is referred to [13] and the appendix therein.
An important consideration is that, in each point z_i, the state and adjoint equation can be solved independently. Consequently, a considerable enhancement in computing performance should be achievable by means of this underlying parallelism. This is the point where parallel computing comes into play.

2.2. Parallel Computing

Parallel computing is an extensive research area in computer science and has become increasingly important in the last decade. In parallel computing a large-scale problem is divided into smaller independent sub-problems, which can be solved simultaneously. Parallelism can exist in a variety of forms: bit-level, instruction level, data and task parallelism. This parallelism can be utilized either in a single computer, with multiple processing elements, or with multiple computers, e.g., cluster and grid computing systems. [3, 39]
There also exist a wide range of available programming frameworks. The Message Passing Interface (MPI) standard is a model for distributed memory systems (e.g. clusters), where all interaction and data sharing must be done via message passing [2]. OpenMP (Open Multi-Processing) is based on compiler directives and supports shared memory systems [40]. Both, MPI and OpenMP, are widley used in various applications of high performance parallel computing. General-purpose computing on graphics processing unit (GPGPU) is the use of the GPU for computing problems originally performed by the CPU. The GPU architecture is highly specialized for parallel processing and even a single heterogeneous platform (CPU-GPU system) provides remarkable computing performance. For NVIDIA GPUs the Compute Unified Device Architecture (CUDA) platform enables the use of general purpose computing [6]. A vendor-independent framework for heterogeneous platforms is OpenCL (Open Computing Language). Subsequently, a brief overview of the CUDA and OpenMP application programming interfaces (API) is given. For a detailed description the reader is referred to the NVIDIA CUDA programming guide [6] and to the OpenMP application interface documentation [4], respectively.

2.2.1. NVIDIA CUDA

CUDA is a programming model developed by NVIDIA and provides compatibility with programming languages such as C, C++ and Fortran. Given this accessibility programmers are able to exploit the highly parallelism-supporting architecture of the GPU for parallel computing tasks. The CUDA technology is proprietary to NVIDIA and limits applications to CUDA capable GPUs.

2.2.1.1. GPU Architecture

The hardware architecture of a multi-core CPU is designed to optimize sequential code execution. The goal is to optimize performance of a small number of heavy-weighted threads. The control unit allows the utilization of instruction level parallelism and out-of-order execution while maintaining the appearance of sequential execution. Additionally, a large proportion of chip area is dedicated to cache memory to reduce the data access and instruction latencies. However, GPUs are designed to perform a massive amount of floating point operations. Therefore, much more chip area is dedicated to floating point operations (arithmetic logic units or ALUs) as to cache memory and control units (see figure 2.10). The goal is to optimize performance for a thousands of threads by means of a throughput orientated hardware design principle.

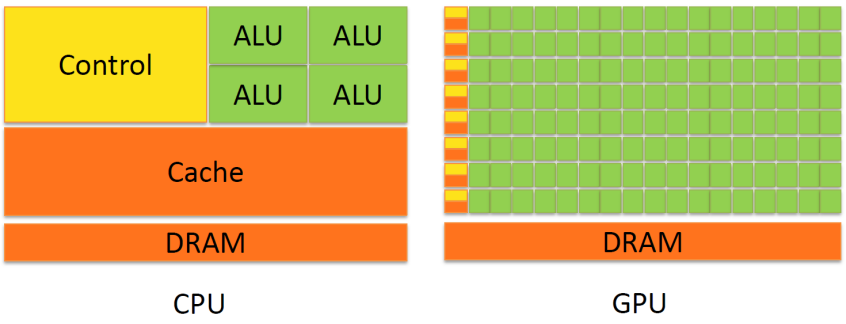


Figure 2.10.: CPU vs. GPU: The CPU is designed to minimize latency in a small number of heavy-weighted threads. Therefore, most of the chip area is dedicated to the control unit and cache memory. The GPU hides memory access latencies by focusing on computationally intensive tasks and much more chip area is dedicated to floating point calculations. [6]

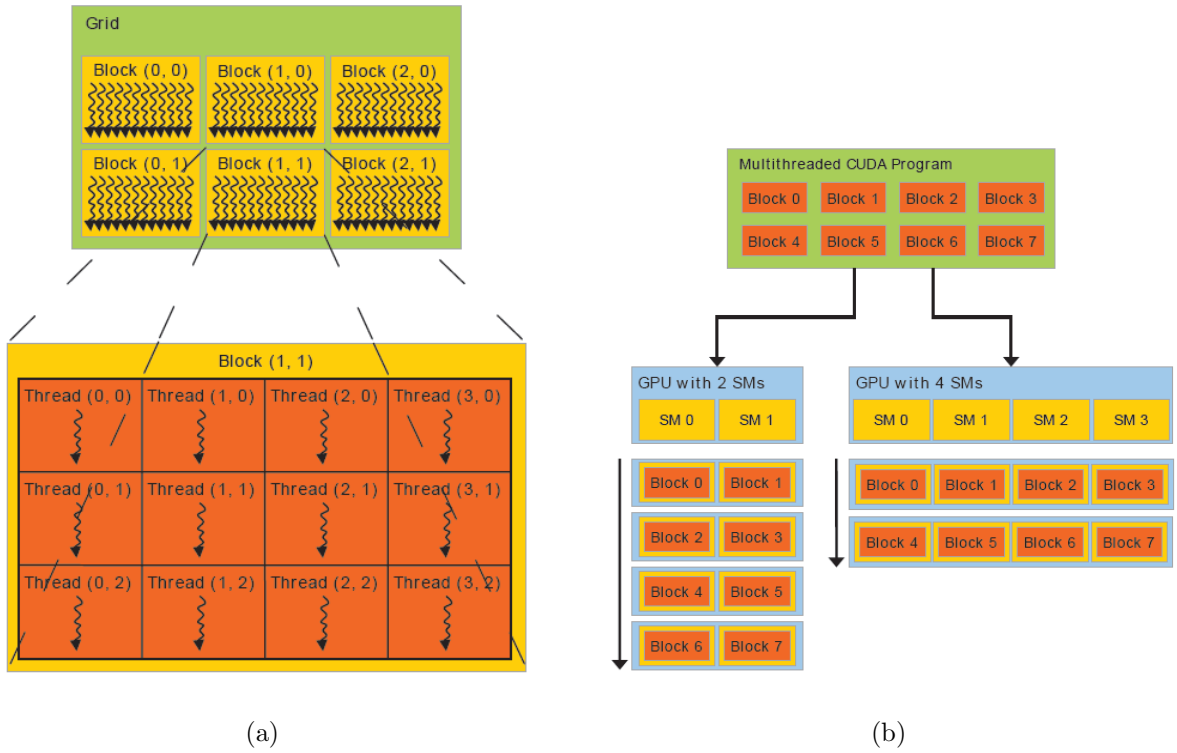
A CUDA capable GPU is organized into streaming multiprocessors (SMs) and each SM is composed of CUDA cores, which share the same control unit and instruction cache. The number of cores in a SM differ from one architecture generation to another. For

example, in NVIDIA's latest Maxwell architecture a SM consists of 128 CUDA cores, more specifically, a Geforce GTX 970 has 13 SMs with 128 CUDA cores each, resulting in a total of 1664 cores. The hardware implementation of a multiprocessor is based on the Single-Instruction, Multiple-Thread (SIMT) architecture, which was introduced by NVIDIA to execute hundreds of threads concurrently. In a SM, 32 threads are grouped together to form a warp and each thread within a warp must execute the same instruction (or otherwise is disabled). When a SM is given a thread block for execution, it splits them into warps and each warp is scheduled by a warp scheduler for execution.

One major issue in the past was the support for double-precision floating point calculations. With the release of compute capability 1.3, NVIDIA introduced double-precision supporting GPUs to overcome this issue and enhanced this support with future chip releases. In a heterogeneous system most applications will use the advantages of both CPU and GPU by executing the sequential parts on the CPU and computationally heavy parts on the GPU. To support this CPU/GPU interaction NVIDIA released the CUDA programming model with a low level Driver API and a higher level Runtime API.

2.2.1.2. CUDA Programming Model

The CUDA programming model is designed for scalability. CUDA applications scale their parallelism to GPUs with a variable number of cores. The programmer is guided to decompose the problem into sub-problems that can be solved in parallel by a block of threads. Each block is scheduled to an available SM, so that GPUs with a higher amount of SMs will automatically execute the program faster than the GPU with fewer SMs. This enables a scalability over a wide range of GPUs. For instance, suppose we have two GPUs, one with 2 and the second with 4 SMs, and the problem is split into 8 blocks. For this setup the corresponding block scheduling order is given in figure 2.11a.



(a) (b)

Figure 2.11.: A problem is decomposed into independent sub-problems that can be solved by a block of threads. All thread blocks are organized into a grid. In the case of (a) grid and block are two-dimensional. Each block is sheduled automatically to a SM for execution (b). This allows high scalability over a wide range of GPUs. [6]

In terms of CUDA programming the heterogeneous computing system consists of a *host*, the CPU, and one or more *devices*, the GPUs. Both, host and device, have separate memory spaces, which are referred to as *host memory* and *device memory*.

Kernel functions In CUDA C/C++ a kernel specifies a function that will be executed by N threads in parallel. A kernel is defined via the `__global__` declaration specifier and can be called from the host using the `<<<numBlocks, threadsPerBlock>>>` execution configuration syntax. This syntax sets the grid and block dimensions, which can either be one-, two- or three-dimensional. The `threadsPerBlock` variable defines the number

of threads that are grouped together in a block. The variable `numBlocks` describes the number of blocks organized into a grid (see figure 2.11b). In order to distinguish between the threads, each thread gets a unique index and can be addressed by predefined variables. In addition, the `__host__` keyword indicates a host function and the `__device__` specifier defines a device function. A host function is a traditional C/C++ function that can only be called and executed on the host. On the contrary, device functions execute on the GPU and can only be called from kernel or other device functions.

Thread hierarchy Another noteworthy extension to the C/C++ language is the predefined struct variable `threadIdx`. This struct consists of 3 elements `x`, `y` and `z`, depending on the block dimension, and identifies the threads within a block. The thread indices in different blocks are the same, therefore, each block within the grid can be addressed via `blockIdx` and the dimension of the block (number of threads) is given by the `blockDim` variable. The grid dimension can be accessed by the `gridDim` structure. In devices with compute capability 2.0 and higher the number of threads per block is limited to 1024. Nevertheless, the maximum number of threads per kernel launch is equal to the number of blocks times the number of threads per block, and the number of blocks in a grid is usually dependent on the data size. Assume we have the two-dimensional grid with two-dimensional blocks given in Figure 2.11a. The unique global thread index is calculated in two steps:

```
1  int blockId = blockIdx.x + blockIdx.y * gridDim.x;
2  int threadId = blockId*(blockDim.x * blockDim.y)
3          + (threadIdx.y * blockDim.x) + threadIdx.x;
```

First we have to map the two-dimensional block index into a linear index given by `blockID`. Second, the two-dimensional thread index is mapped into a global `threadId`. For example, thread $(3, 2)$ in block $(1, 1)$ has a `blockId` of 4, resulting in a global thread index `threadId` of 59. This is useful for mapping threads to tasks, as this provides a unique identifier for all the threads launched by the kernel.

2.2.1.3. CUDA Memory Model

In heterogeneous systems there are two separate memory spaces, the host and device memory. To perform computations on the GPU, the programmer needs to transfer the data from the host to the device and the results from the device back to the host. This data transfers are usually slow and therefore they may be bottlenecks in the application. In CUDA the programmer has access to various types of device memory: registers, global, local, shared, constant and texture memory.

Registers Are directly located on the chip and thus can be accessed at high speed. The memory is automatically allocated from the SM via register file (i.e. number of registers per block); each thread has only access to its individual registers. On the Maxwell architecture the register file size per SM is 256 KB.

Global memory Is the main GPU memory (e.g. 4 GB on the GeForce GTX 970) and accessible from all threads in a grid for read and write operations. The `__device__` keyword declares a variable that resides on the device. In order to store data on the GPU, that can communicate with the host, global memory can be allocated from the host using the API function `cudaMalloc()` and `cudaMemcpy()` manages the data transfer between host and device memory. Global memory is the slowest memory on the device, e.g., up to 1000 times slower than registers.

Local memory The compiler automatically places variables in local memory when they do not fit into registers. Local memory is part of global memory, thus, slow in comparison to registers, besides that, the behaviour is the same. The amount of local memory per thread is limited to 512 KB for GPUs with compute capability 2.0 or higher.

Shared memory Resides physically on the chip and therefore has much higher bandwidth and much lower latency than local or global memory. The size of shared memory per SM is limited to 96 KB in the Maxwell architecture and can be declared using the `__shared__` keyword. A shared variable is private for each block and provides shared access for all threads within a block, thus, allowing thread communication and cooperation in a block. In order to provide race conditions the `__syncthreads()` function ensures that all data from all threads is valid before threads read from shared memory, which can be written to by other threads.

Constant memory Is a read-only type of memory. It is used for data that will not change during a kernel execution and can be declared via the `__constant__` specifier. Constant variables are visible to all threads within a grid but only readable from the device. In order to copy data from the host to constant memory the `cudaMemcpyToSymbol` function is used. The available memory size is limited to 64 KB.

Texture memory Is another type of read-only memory and was originally designed for conventional graphics applications. Texture memory is optimized for spatial locality and therefore provides a performance benefit for nearby memory reads.

2.2.2. OpenMP

Open Multi-Processing is a standardized API for parallel computing on shared memory systems in C/C++ and Fortran. Basically, OpenMP consists of a set of compiler directives, environment variables and library functions to express and control parallelism. The portable and scalable OpenMP programming model is based on the fork-join parallel design pattern. An OpenMP application starts with a master thread that forks a specified number of slave threads by defining a parallel region. In this region the threads run concurrently and the work is shared among them. After execution of the parallel code section the threads join back into the master thread, which continues the execution of the sequential code. It is possible to define various parallel regions in an application, hence, the master thread can fork and join repeatedly (see figure 2.12).

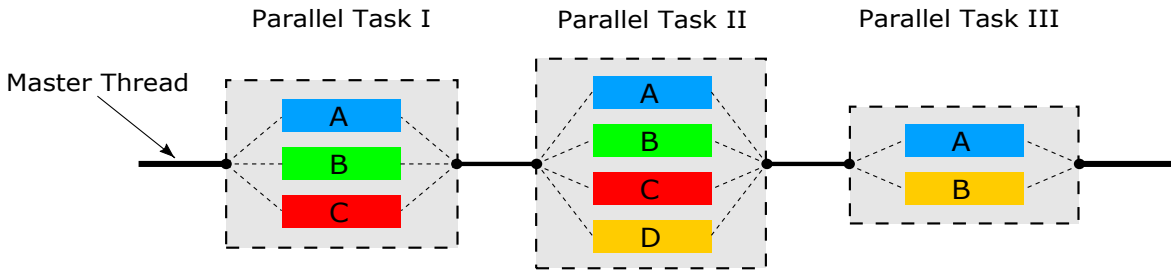


Figure 2.12.: A schematic illustration of the fork-join model. A master thread forks and joins sequent parallel regions with various threads.

OpenMP directives In the OpenMP programming model, parallel code sections are specified via preprocessor directives, also referred to as pragmas. In C/C++, OpenMP uses `#pragma`. The pragma `omp parallel` defines a parallel region and creates threads to perform the code enclosed in the construct in parallel. Each thread has an individual identifier, which can be obtained using runtime library functions. Additionally, independent work is assigned to one or all of the threads via work-sharing constructs. The loop constructs `omp for` or `omp do` are used to distribute the loop iterations across threads that already

exist in a parallel region. With the `section` construct it is possible to assign a set of structured blocks to different threads. Each thread executes its corresponding independent code block. The `single` construct specifies the enclosed code to be executed by one of the thread (not necessarily the master thread) and the `master` construct marks the code block to be executed only by the master thread.

OpenMP clauses The behaviour of a directive can be controlled by various clauses, and each directive has its own set of valid clauses. For instance, the visibility of variables to threads is specified using data sharing attribute clauses like `shared` or `private`. Shared variables are visible and accessible to all threads, whereas, in the case of a private variable each thread will have its own local copy that is used as a temporary variable.

Runtime routines The functions provided by the OpenMP runtime library are mainly used to check and set runtime parameters. For example, `omp_get_num_threads` returns the number of threads in a parallel region and `omp_set_num_threads` specifies the number of threads. Additionally, there are functions for thread synchronisation and timing purposes.

Environment variables Specifies the runtime settings of an OpenMP application prior to the execution. Modifications to environment variables are ignored after program launch, even if modified by the program itself. However, the configuration of some environment variables can be altered during execution by using suitable runtime routines or directive clauses.

3. Methods

3.1. Matlab Framework

The optimal control framework, described in section 2.1.7.1, was implemented in MATLAB (The MathWorks Inc., Natick, USA) using the Parallel Computing Toolbox. This Matlab framework for efficient high-resolution RF pulse design is available on GitHub[1] and contains two top-level m-files `test_single.m` and `test_multi.m` to compute the optimal RF pulse (i.e. the control $\mathbf{u}$) for the single-slice problem and the multi-slice problem, respectively. First, the problem parameters are initialized in a structure `d` and likewise the necessary parameters for the trust-region Newton-CG method in a structure `tr`. In the next step the target magnetization profile is defined. In the case of multi-slice excitation the center positions for all simultaneous slices are specified. These center positions depend on the slice number and are different for an even or odd number of slices. Additionally, an alternating phase shift may be incorporated to account for a CAIPIRINHA-based excitation pattern and to reduce Gibbs ringing artifacts the target magnetization profile is filtered with a Gaussian kernel before the optimization. Subsequently, the matrix-free trust-region Newton-CG algorithm, implemented in `tr_newton.m`, computes the optimal control $\mathbf{u}$ that is a minimizer to the objective function 2.14. A basic flowchart of this optimization algorithm is depicted in figure 3.1.

[1] https://github.com/chaigner/rfcontrol/releases/v1.2

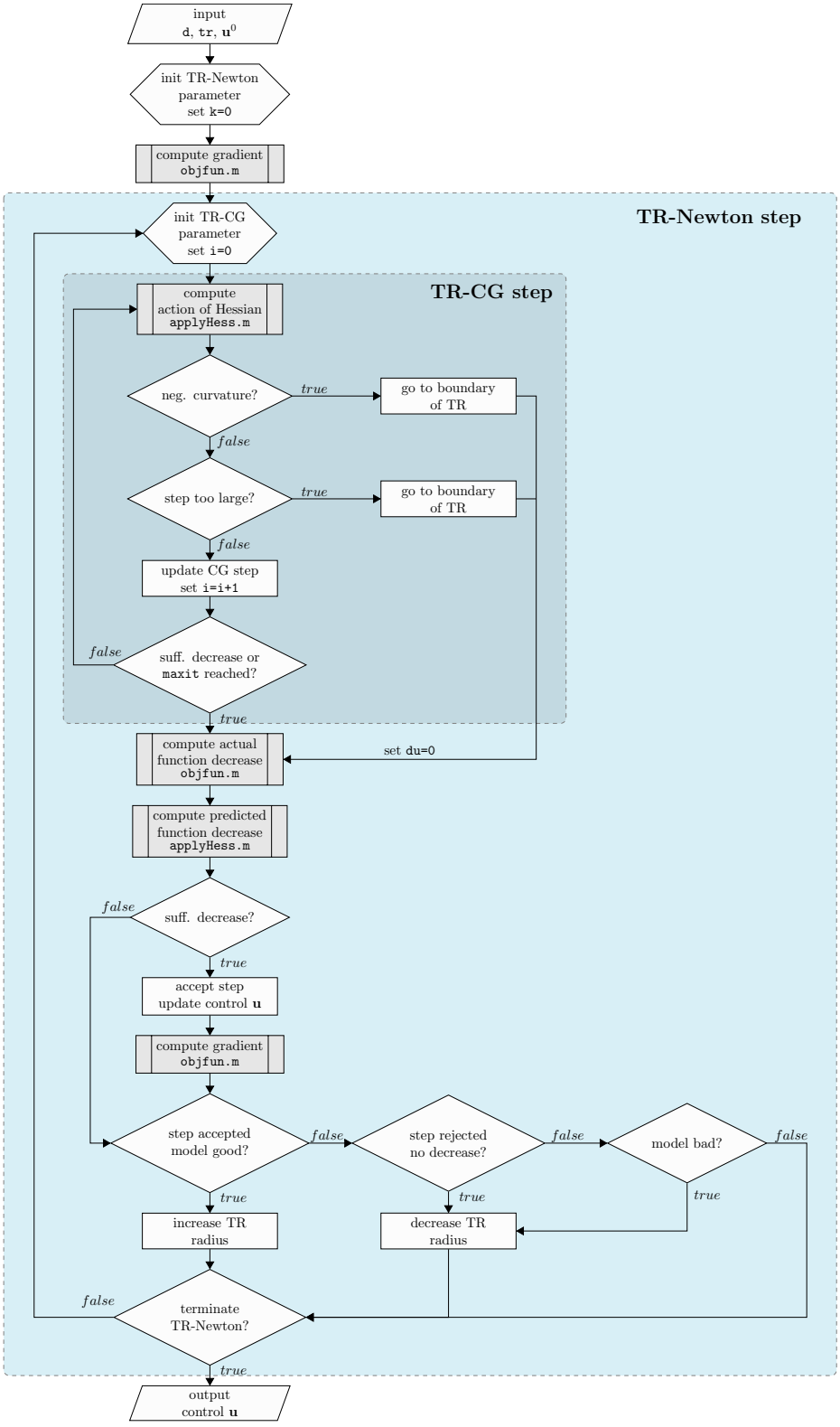


Figure 3.1.: Basic flowchart of the matrix-free trust-region Newton-CG optimization algorithm.

In each Newton step the algorithm computes the gradient at the actual point $\mathbf{g}(\mathbf{u}^k)$ (see equation 2.18) by a function call to `objfun.m`. Since the gradient evaluation is based on the adjoint calculus the solution to the forward and adjoint model are required. The forward integration of the Bloch equation is implemented in `cn_bloch.m` and the corresponding integration of the adjoint equation is performed by `cn_adjoint.m`. It is important to note that for each point z_i the forward and adjoint model can be solved independently and in parallel. Additionally, `objfun.m` computes the value $J(\mathbf{u}^k)$ of the objective function and the necessary information to evaluate the Hessian in the actual point. If one needs only the value of the functional, `objfun.m` can be called with just one output argument, thus, avoiding the integration of the adjoint model. Subsequently, the Newton update step $\delta\mathbf{u}$ is obtained by applying a Steihaug CG algorithm [14] to equation 2.16. Each CG iteration requires the computation of the action of the Hessian (equation 2.23) by invoking the `applyHess.m` function, where the linearized state and adjoint equation are solved. After completion of the CG algorithm the update step $\delta\mathbf{u}$ is verified to be an acceptable step, hence, providing a sufficient decrease of the functional. If $\delta\mathbf{u}$ is accepted and the model is good, the trust-region radius will be increased, and contrarily, if the step is rejected or the model is bad, the trust-region will be reduced. The function `tr_newton.m` terminates as soon as the gradient is smaller than the absolute tolerance tol_N or the maximum number of trust-region Newton iterations is reached, yielding the optimal control RF pulse.

For instance, the computation of an OC pulse, exciting a single slice of width $\Delta_w = 5mm$ and flip angle of $\theta = 90°$, terminates after 4 Newton iterations and a total number of 28 CG steps with gradient norm of $|\mathbf{g}| = 1.459 \times 10^{-7}$. This results in an overall optimization time of approximately $30min$ using Matlab `parfor` with 4 workers using the GTX 970 workstation[2]. The results of the Matlab profiler, listing the number of function calls and execution times of the individual functions, are shown in figure 3.2. The `tr_newton.m` function calls in each Newton iteration the CG algorithm `tr_cg.m` contributing to 75.1% of the total time.

[2] an overview of the hardware specifications is given in chapter 4. Results

tr_newton (1 call, 1660.126 sec)

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time	Time Plot
tr_cg (b)	function	4	1247.413 s	75.1%	
objfun (c)	function	9	243.741 s	14.7%	
tr_newton>@(du)applyHess(d,Xk,du)	anonymous function	4	168.860 s	10.2%	
num2str	function	31	0.023 s	0.0%	
@(x,y)d.dt*(x*y)	anonymous function	13	0.001 s	0.0%	
tr_newton>create@(du)applyHess(d,Xk,du)	anonymous function	4	0 s	0%	
Self time (built-ins, overhead, etc.)			0.088 s	0.0%	
Totals			1660.126 s	100%	

(a) `tr_newton.m`

tr_cg (4 calls, 1247.413 sec)

Children (called functions)

Function Name	Calls	Total Time	% Time
tr_newton>@(du)applyHess(d,Xk,du)	28	1247.399 s	100.0%
@(x,y)d.dt*(x*y)	87	0.003 s	0.0%
tr_cg>dist2bdy	1	0.001 s	0.0%
Self time (built-ins, overhead, etc.)		0.010 s	0.0%
Totals		1247.413 s	100%

(b) `tr_cg.m`

objfun (9 calls, 243.741 sec)

Children (called functions)

Function Name	Calls	Total Time	% Time
cn_bloch	9	153.768 s	63.1%
cn_adjoint	5	88.443 s	36.3%
squeeze	5	0.003 s	0.0%
Self time (built-ins, overhead, etc.)		1.526 s	0.6%
Totals		243.741 s	100%

(c) `objfun.m`

Figure 3.2.: Results of the Matlab profiler for the computation of a single-slice OC pulse. The `applyHess.m`, `cn_adjoint.m` and `cn_bloch.m` functions are the major bottlenecks in the application (marked in red), and therefore making them target for performance optimizations.

One step deeper, in the `tr_cg.m` function, for each of the 28 CG iterations the `applyHess.m` function is called, which is accountable for almost the entire execution time. Furthermore, in each Newton step `applyHess.m` is called, adding to a total number of 32 function calls. Consequently, the `applyHess.m` function is one of the major bottlenecks in the computation, with a share of about 85.3% of total execution time. Another considerable bottleneck

is `objfun.m` with a contribution of 14.7% to the execution time. This function invokes, for each gradient evaluation, the `cn_bloch.m` and `cn_adjoint.m` functions, which are responsible for almost the whole computation time. In addition, if `objfun.m` is called to evaluate the objective function in a specific point only a call to `cn_bloch.m` is performed. A gradient evaluation is performed once prior iteration start and once per Newton iteration and thus resulting in a total of 5 times. Additionally, the computation of the functional is performed 4 times contributing to a total number of 9 function calls. To summarize, the major bottlenecks in the implementation are the `applyHess.m`, `cn_adjoint.m` and `cn_bloch.m` functions, thereby making them target for performance optimizations.

Matlab Executable (MEX) The Matlab MEX interface provides a possibility to enhance performance by calling subroutines written in C, C++ or Fortran. When compiled, the binary MEX files are dynamically loaded, allowing the invokation of C, C++, or Fortran code as if it were a Matlab function. Since the major bottlenecks in the implementation are the `applyHess.m`, `cn_adjoint.m` and `cn_bloch.m` subroutines a considerable acceleration by means of MEX files may be possible. Thus, in a first step, a sequential C/C++ version of the three computationally intensive subroutines is implemented. In a further step, a parallel version is implemented in CUDA and OpenMP to exploit the intrinsic parallelism provided by the fact that for each spatial location the state and adjoint equation can be solved independently.

3.2. CMake Project Structure

All different MEX file implementations are included in the *RFcontrol* project and the build process is managed by the open-source software CMake to support cross-platform builds for Windows and Linux operating systems. CMake is designed to generate standard build files for the native build environment, such as Microsoft Visual Studio solution files on

Windows and makefiles on Unix. The RFcontrol project hierarchy is organized into header libraries and source files for the associated MEX functions. A list of these files is given in table 3.2. Depending on the implementation type the suffix `CPU`, `GPU` or `OMP` is attached to the source files. Furthermore, to specify whether the application uses single or double precision floating point format the `sprec_` prefix or no prefix is attached, respectively. In the following a brief description of the content of the header libraries is given.

Table 3.1.: List of files in the RFcontrol project.

header files	source files
RF_control_helper.cuh	applyHess.cu
basic_lin_algebra.cuh	applyHess.cpp
config.h	cn_bloch.cu
matlab_helper.h	cn_bloch.cpp
dpara.h	cn_adjoint.cu
cuda_utils.h	cn_adjoint.cpp
	checkRequiredDeviceMem

`config.h` Automatically generated configuration file by CMake. The RFcontrol project settings are configured per CMake environment variables. The `USE_DOUBLE_PRECISION` parameter defines the floating point precision per `typedef` of **`DType`** to `float` or `double`. Before the build process CMake detects if the system has a CUDA capable device and the `CUDA_GPU_FOUND` variable will be set accordingly. Additionally, CUDA error checking and an occupancy based kernel launch can be enabled or disabled.

`RF_control_helper.cuh` Several subroutines needed in the optimal control framework to solve the state and adjoint equation as well as the linearized state and adjoint model. All functions are realized as host and device functions making them callable from both.

basic_lin_algebra.cuh Implements basic matrix and vector operations on the host and device. Due to the three-dimensionality of the Bloch equation only 3×3 matrix and 3×1 vector operations are needed.

dpara.h Defines the structure `dpara` containing the model parameters.

matlab_helper.h Contains functions to manage the input and output of the MEX files and to redirect the `std::cout` to the Matlab command window.

cuda_utils.h Includes CUDA error handling, kernel performance metrics and functions to aid occupancy based kernel launch (CUDA 6.5 required).

3.3. Thrust Library

The RFcontrol project utilizes the Thrust library, a powerful library of parallel algorithms and data structures, to manage the data transfers between the host and device. Thrust is a C++ template library for CUDA imitating the Standard Template Library (STL) and provides a high-level interface that is interoperable with CUDA C/C++ [41]. Since CUDA Toolkit version 4.0 the Thrust library is already included and no separate installation is required. The key features of the Thrust library are data structures and parallel primitives such as `scan`, `sort` and `reduce`, which can be combined together to implement complex algorithms. Thrust provides two generic vector containers, `thrust::host_vector` and `thrust::device_vector`. The `host_vector` is allocated in host memory and the `device_vctor` resides in GPU device memory. These containers increase the readability and re-usability of code by hiding CUDA memory allocation methods such as `cudaMalloc`, `cudaMemcpy` and `cudaFree`. Thrust uses iterators, which can be thought of as pointers to

array elements, to access and operate on the vector containers. Unlike pointers, iterators provide additional information like type of memory space of the underlying container. With this Thrust tracks memory space and is able to determine whether to use a host or device implementation of the called function. This is known as *static dispatching*. Additionally, Thrust can be utilized with different *device backend systems* such as CUDA (default) or OpenMP. One can change the global device system by adding specific options to the compiler, requiring no changes to the source code. Instead of applying a global system change the thrust system can be accessed directly by defining a system-specific vector using the desired backend system. It is also possible to *re-tag* an existing thrust iterator to a different device system to operate on existent data structures.

3.4. Sequential C/C++ Implementation

In a first step a sequential version of the computational-heavy `cn_bloch`, `cn_adjoint` and `applyHess` subroutines in the optimal control framework are implemented in C/C++. For the sake of comparability, the sequential implementation is based on `thrust::host_vector` containers, although this is not mandatory and the C++ standard template library could be used as well.

Problem Dimensions The numerical computation of an OC pulse requires a discretization of the state equation and objective function in time and spatial domain (described in section 2.1.7.3). The spatial discretization results in a grid of size N_x and likewise the time discretization in a grid of size N_t. The number of control points is specified with N_u. Consequently, the size of the used data containers is a combination of this three problem dimensions.

Read Input Data Every C program has a `main()` function. In Matlab MEX files the routine `mexFunction` is used as entry point to the function. The following input parameters are passed to `mexFunction`: the number of output (left-hand side) arguments `nlhs`, the array of output arguments `plhs`, the number of input (right-hand side) arguments `nrhs` and the array of input arguments `prhs`. Listing 3.1 shows how to retrieve input data from the MEX interface.

Listing 3.1: Read input from MEX entry function.

```
1 void mexFunction(int nlhs, mxArray *plhs[],
2          int nrhs, const mxArray *prhs[]){
3 // ...
4 #define M0_IN prhs[1]
5 // ...
6
7   // init input parameter
8   // ...
9   DType* M0 = (DType*)mxGetData(M0_IN);
10  // ...
11 }
```

To provide better readability the pointer to the first element of the initial magnetization `prhs[1]` is given the name `M0_IN` via a preprocessor macro. In order to retrieve a pointer to real data from a `mxArray` the `mxGetData` function is used. This function returns a void pointer, and one needs to cast the return value to the pointer type that underlies the data type used by `M0_IN`. In this case `DType` represents a user defined data type, which can either be `double` or `float` (`typedef` in `config.h`).

Prepare Output Data In MEX files a `mxArray` output argument is created by using provided functions from Matlab. In listing 3.2 an $3 \times N_x \times N_t$ output matrix is generated to store the computation result. N_x represents the number of spatial discretization points and N_t the number of time points.

Listing 3.2: Prepare output from MEX entry function.

```
1  const mwSize mat_dims[3] = { 3, Nx, Nt };
2  plhs[0] = matCreateOutputArray(3, mat_dims);
3  DType* mat_output = (DType*)mxGetData(plhs[0]);
```

The `matCreateOutputArray` assigns a matrix with dimensions `mat_dims` to `plhs[0]`. In order to write to the output array by a function the pointer to the first data element `mat_output` is used.

Matlab vs. C/C++ Indexing It is important to note that Matlab and C/C++ allocate multidimensional array memory in a different way. MATLAB allocates the memory as a contiguous, one-dimensional block in column-major order, and contrarily, C/C++ stores the same array in row-major order. If interfacing with MATLAB the preferred method of accessing array elements is in column-major order by a linear index. If row-major indexing is used, one needs to transpose the input and output data to switch from one order to the other. Additionally, MATLAB array indexing starts with 1 and in C/C++ the first element of an array is indicated with 0. The RFcontrol project uses two-dimensional and three-dimensional arrays and the calculation of the linear index is realized with preprocessor macros (listing 3.3).

Listing 3.3: Convert MATLAB 2D/3D index to a linear index.

```
1 // Convert Matlab 2D/3D index to linear index:
2 // Matrix(I,J,K) = Matrix(LIN_IDX)
3 // r ... number of rows
4 // c ... number columns
5 // for C index notation use Matrix[LIN_IDX - 1]
6
7 #define LIN_IDX_3D(I,J,K,r,c) (I+r*(J-1)+r*c*(K-1))
8 #define LIN_IDX_2D(I,J,r) (I+r*(J-1))
```

`cn_blochCPU` The `cn_blochCPU` MEX subroutine solves the Bloch equation 2.12 using a Crank-Nicolson scheme. The numerical Crank-Nicolson method is a finite difference method originally proposed to solve the heat equation. The wrapper function `callCnBlochKernel` initializes the longitudinal relaxation term $\mathbf{b}(z)$ in equation 2.13 and calls `cnBlochKernel`. Note, that the function hierarchy is organized into a kernel structure to provide better comparability with the parallel implementations. The `cnBlochKernel` computes in an outer loop, iterating over each location `z_ind = 0, 1,..., Nx-1`, the temporal evolution of the

magnetization vector $\mathbf{M}$ starting from initial conditions `M0` with RF pulse `u`, `v` and gradient `w`. This temporal evolution is calculated in an inner loop iterating from time point `k=1` to `k=Nt-1`. In each time point the Bloch matrix `Ak` (equation 2.13) is updated using the `setAk` function contained in the `RF_control_helper.cuh` header. Subsequently, in `solveBloch` equation 2.12 is divided into basic linear algebra operations. The required Crank-Nicolson steps are performed in `cnStep`, depicted in listing 3.4. The function takes as input parameters the Bloch matrix in the actual time point `Ak`, the structure d containing the problem parameters and a boolean flag `direction`. The flag `direction` determines whether a step in forward or backward direction is performed. If the direction flag is set to 1 the forward step is calculated by adding a scaled version of `Ak` to the identity matrix, and if the flag is equal to 0 the backward step is obtained by subtracting `scale_Ak` from the identity matrix. The result of the Crank-Nicolson step is stored in `cn_Ak`.

Listing 3.4: Perform a Crank-Nicolson step (forward or backward in time).

```
template<typename TType>
__inline__ __host__ __device__
void cnStep(TType* Ak, TType* cn_Ak, dpara<TType>* d, bool direction){

  TType identity_matrix[3 * 3] = {1, 0, 0,
                                  0, 1, 0,
                                  0, 0, 1 };
  TType scale_Ak[3 * 3] = {};
  TType time_step = *(d->dt)*0.5;
  scalarMatrixMult3x3<TType>(Ak, &time_step, scale_Ak);
  //direction == 1 --> forward in time
  if (direction)
    addMatrix3x3<TType>(identity_matrix, scale_Ak, cn_Ak);
  // direction == 0 --> back in time
  else
    subtractMatrix3x3<TType>(identity_matrix, scale_Ak, cn_Ak);
}
```

`cn_adjointCPU` The solution to the adjoint equation 2.17 is obtained via the `cn_adjointCPU` MEX subroutine. In the outer loop the `cnAdjointKernel` computes the adjoint magnetization vector $\mathbf{P}$ starting from terminal conditions `PT` with the RF pulse `u`, `v` and gradient `w`. The major difference to the `cn_bloch` implementation is that the inner loop iterates over the time points in a rearward manner beginning with `k = Nt-2`. The magnetization

vector in time point `k = Nt-1` is obtained from the terminal conditions `PT` by performing a separate Crank-Nicolson step prior to the inner loop iterations. In each further time step the transpose of the Bloch matrix is set with the `setAkp1` function, and subsequently, equation 2.17 is solved using the `solveAdjoint` function. As in `solveBloch`, the adjoint equation is divided into basic linear algebra operations, and contrarily, no `addVector3x1` operation is required due to the non-existent longitudinal relaxation term $\mathbf{b}(z)$.

`applyHessCPU` The MEX subroutine `applyHessCPU` implements the computation of the action of the Hessian for a given direction. The wrapper function `callApplyHessKernel` allocates host memory to store the results of the Hessian action in `Hdu_host`. The direction of the Hessian action is obtained from the input arguments and stored to the `host_vector du_host`. Prior to invocation of the `applyHessKernel` the first column of `Hdu_host` is calculated by scalar multiplication of the cost control parameter `d.alpha` with the step `du_host`. This operation is known as a SSCAL or a DSCAL computation for single- or double-precision floating point format, respectively. SSCAL/DSCAL are functions in the standard Basic Linear Algebra Subroutines (BLAS) library for C and Fortran, and listing 3.5 shows an approach to realize a SSCAL/DSCAL operation using the Thrust library.

Listing 3.5: SSCAL/DSCAL operation using `thrust::transform` iterator.

```
1  // compute first column of Hdu
2  // perform SSCAL/DSCAL res = scalar*x
3  using namespace thrust::placeholders;
4  thrust::transform(du_host.begin(), du_host.end(), Hdu_host.begin(),
             (DType)*(d.alpha)*_1);
```

The `thrust::transform` applies the `(DType)*(d.alpha)*_1` operation to each element of the input vector defined by `du_host.begin()` and `du_host_end()` and stores the result to the first column of `Hdu_host`. The `thrust::placeholders` namespace is used to allow an inline implementation of an arithmetic function by using the `_1` notation. Since the size of `du_host` (`Nu`) is lower or equal to the number of time points `Nt` a zero-padding of `du_host` to readout time is performed (listing 3.6). The `thrust::fill` primitive assigns the value 0 to the specified range `du_host.begin()+ Nu, du_host.end()` in the input sequence.

Listing 3.6: Zero-padding of `du_host` to readout time.

```
1  //zero padding du to readout time
2  du_host.resize(Nt - 1);
3  thrust::fill(du_host.begin() + Nu, du_host.end(), 0);
```

Subsequently, the `applyHessKernel` is invoked. In order to obtain the required information for the computation of the application of the Hessian the linearized state equation 2.20 and adjoint equation 2.22 is solved using the same method as described in `cn_blochCPU` and `cn_adjointCPU`. In the following, the action of the Hessian is calculated by implementing equation 2.23 as shown in listing 3.7.

Listing 3.7: Basic structure of the `applyHessKernel`.

```
1  template<typename TType>
2  void applyHessKernel(dpara<TType>& d, TType* N, TType* P,
3                       TType* u, TType* v, TType* w, TType* du,
4                       int Nt, int Nx, int Nu, TType* output){
5
6    // for each spatial location
7    for (int z_ind = 0; z_ind < Nx; z_ind++){
8
9    // solve linearized Bloch equation
10   // solve linearized adjoint equation
11
12   // action of the Hessian with:
13   // dNz, dMz --> solution to linearized state and adjoint equation
14   // N, P     --> solution to state and adjoint equation
15     for (int ii = 0; ii < Nu; ii++){
16      output[(ii + Nu) + Nu*z_ind] = B1*(*d.dx)*
17       (dNz[LIN_IDX_2D(3,ii+1,3)-1]*P[LIN_IDX_3D(2,z_ind+1,ii+1,3,Nx)-1]-
18        dNz[LIN_IDX_2D(2,ii+1,3)-1]*P[LIN_IDX_3D(3,z_ind+1,ii+1,3,Nx)-1]+
19        N[LIN_IDX_3D(3,z_ind+1,ii+1,3,Nx)-1]*dMz[LIN_IDX_2D(2,ii+1,3)-1]-
20        N[LIN_IDX_3D(2,z_ind+1,ii+1,3,Nx)-1]*dMz[LIN_IDX_2D(3,ii+1,3)-1]);
21     }
22   }
23 }
```

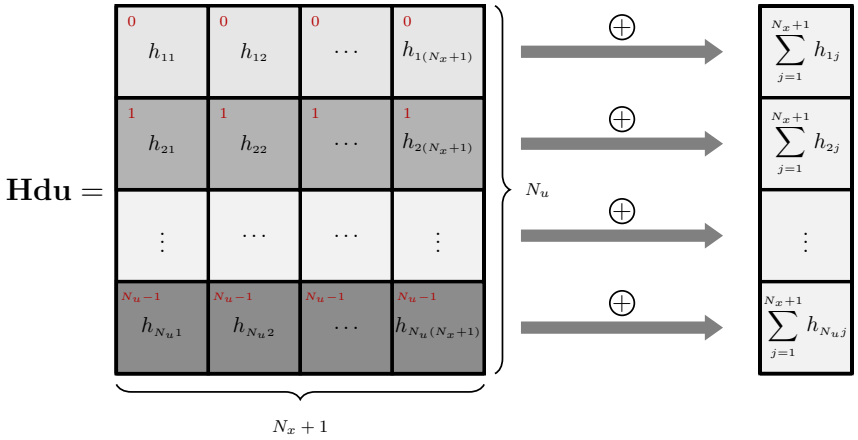
For each `z_ind=0, ..., Nx-1` the intermediate results are stored to the corresponding column of `output` (equal to `Hdu_host`) resulting in a $N_u \times (N_x + 1)$ matrix. In a last step, the temporary `Hdu_host` matrix is reduced to a single $N_u \times 1$ column by computing the sum of rows of the matrix. In parallel computing such an operation is known as *reduction*. The Thrust library provides powerful primitives to perform reduction operations. The sum

of rows is calculated using `thrust::reduce_by_key` in conjunction with *fancy iterators* as demonstrated in listing 3.8.

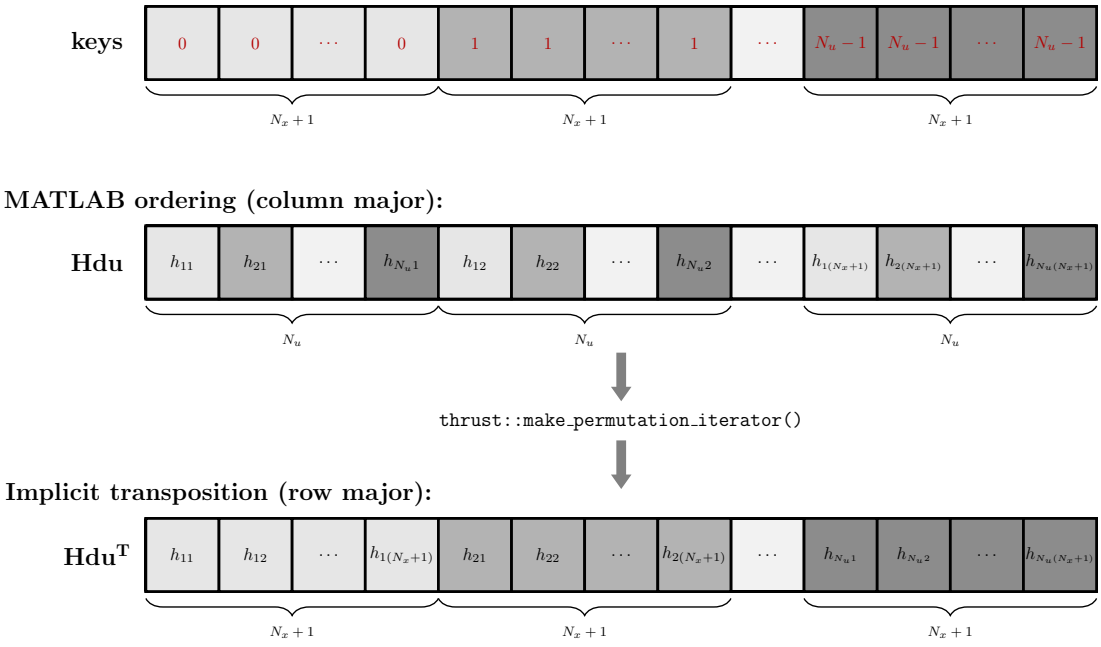
Listing 3.8: Calculate the sum of rows of `Hdu_host` using the Thrust library.

```
1  //row sum of Hdu using thrust reduce by key
2  thrust::host_vector<DType> row_sums(Nu);
3  thrust::reduce_by_key(
4    thrust::make_transform_iterator(
5        thrust::make_counting_iterator(0),
6        linear_index_to_row_index<int>(Nx+1)),
7    thrust::make_transform_iterator(
8        thrust::make_counting_iterator(0),
9        linear_index_to_row_index<int>(Nx+1))+((Nx+1)*Nu),
10   thrust::make_permutation_iterator(
11       Hdu_host.begin(),
12       thrust::make_transform_iterator(
13           thrust::make_counting_iterator(0),
14           (_1%(Nx+1))*(Nu)+_1/(Nx+1))),
15   thrust::make_discard_iterator(),
16   row_sums.begin(),
17   thrust::equal_to<int>(),
18   thrust::plus<DType>());
```

The `reduce_by_key` primitive sums values with equal keys (figure 3.3a). Each element in the same row of `Hdu_host` is marked with its individual row number as key using `make_transform_iterator`, `make_counting_iterator` and the `linear_index_to_row_index` functor. Since the values in `Hdu_host` are stored in column major order an implicit transposition is obtained with a `make_permutation_iterator`, iterating over the elements as they were stored in row major order (figure 3.3b). The binary function `plus<DType>` defines the operation which should be applied to the value with equal keys (checked with `equal_to<int>`). All values with equal row index key are reduced to a single value forming the sum of rows in the `host_vector<DType> row_sums(Nu)`. The `reduce_by_key` primitive returns the output keys, which are not required in this case. In order to ignore this output without wasting memory capacity or bandwidth the `make_discard_operator` is used.



(a) Sum along the rows of `Hdu`.



(b) Linear alignement of `keys` and elements of `Hdu` in memory.

Figure 3.3.: Illustration of the reduction (row sum) of the `Hdu` matrix using the Thrust primitive `reduce_by_key` (a). Each element, located in the same row, is marked with its row index as key (red numbers), and `reduce_by_key` sums all elements with equal keys. Since the values of `Hdu` are stored in column-major order (MATLAB ordering) an implicit transposition is obtained with a `permutation_iterator`, treating them as they were stored in row-major order (b). After this implicit transposition each key is assigned to the correct element.

3.5. Parallel CUDA C/C++ Implementation

The parallel CUDA C/C++ implementation exploits the parallelism provided by the fact that for each spatial location `z_ind=0,...,Nx-1` the state and adjoint equation can be solved independently. In the following the main CUDA C/C++ principles used in the RFcontrol project are outlined.

Data Transfer Host/Device An important part of each CUDA C/C++ application is the data transfer between host and device. Normally, the CUDA API functions `cudaMalloc`, `cudaFree` and `cudaMemcpy` are used for memory allocation and data transfer. The RFcontrol project uses the Thrust library to hide the use of CUDA API functions for memory allocation and data transfer. The code snippet in listing 3.9 demonstrates the allocation of device memory, and transfers the host data to the device.

Listing 3.9: Basic memory allocation and data transfer using Thrust containers.

```
1  // ...
2  //copy data to device
3  thrust::device_vector<DType> M0_dev(M0_host,M0_host+(3*Nx));
4  // get raw pointer
5  DType* M0 = thrust::raw_pointer_cast(M0_dev.data());
6
7  // CUDA C:
8  // DType* M0;
9  // cudaMalloc((void**)&M0,3*Nx*sizeof(DType));
10 // cudaMemcpy(M0,M0_host,3*Nx*sizeof(DType),cudaMemcpyHostToDevice);
11 // ...
```

The initial magnetization `M0_host`, residing in host memory space, is transferred to the allocated device memory space `M0_dev`. Thrust device containers are not compatible with user defined CUDA C kernels, but Thrust provides the option to obtain a raw pointer pointing to the first element of the device data. This raw pointer can be passed as input argument to user defined kernel functions. Note, that the suffixes `_host` and `_dev` denote Thrust container variables located on host and device memory, respectively, and for raw pointers no suffix is appended. In addition, to clarify the simplicity of the Thrust library the CUDA C code is shown in the comments.

Kernel Launch In CUDA C/C++ a kernel invokation requires the parameters of the execution configuration, the number of blocks per grid and the number of threads per block, to be set. These execution arguments depend on the problem size. The outer loop iterations in the `cn_bloch` `cn_adjoint` and `applyHess` routines , ranging from the locations `z_ind=0` to `Nx-1`, are fully independent from one another. Therefore, each loop iteration can be executed by a parallel thread resulting in a total number of `Nx` needed threads for parallelization. Listing 3.10 depicts the invokation of a CUDA kernel function.

Listing 3.10: Occupancy based kernel launch.

```
// ...
// determine execution config. to achieve high occupancy
int blocksPerGrid; int threadsPerBlock;
getKernelLaunchParameter(Nx,cnBlochKernel<DType>,
      &blocksPerGrid,&threadsPerBlock);
// launch kernel
cnBlochKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(...);
// ...

// cuda_utils.h
inline int getGridDim(int Nx, int threadsPerBlock){
  return ((Nx + threadsPerBlock - 1) / threadsPerBlock);
}
```

Since high device occupancy is crucial for good performance the launch configuration parameters, `blocksPerGrid` and `threadsPerBlock` are determined in a way that optimal occupancy is achieved. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at once [6]. With CUDA 6.5 Nvidia included API functions to aid in occupancy calculations. The function `getKernelLaunchParameter` calculates the number of threads per block depending on individual kernel characteristics and total number of spatial points. The number of blocks per grid is returned by `getGridDim`. The use of the occupancy based kernel launch feature is limited to CUDA 6.5 or higher. Downward compatibility is provided by means of a fixed number of threads per block, which is defined per preprocessor macro. CMake automatically checks for the installed CUDA version and sets the configuration variable `OCCUPANCY_BASED_LAUNCH` accordingly.

Parallelization The kernel invocation shown in listing 3.10 launches a total number of threads of `blocksPerGrid` times `threadsPerBlock`, which usually results in a higher number of total threads than spatial discretization points `Nx`. The sole case for which the overall thread number do not exceed the amount of locations is when `Nx` is an exact multiple of `threadsPerBlock`. Listings 3.11 and 3.12 illustrate the difference between the parallel and sequential implementation of the `cnBlochKernel`.

Listing 3.11: Sequential C/C++ code.

```
// sequential code version
template<typename TType>

void cnBlochKernel(...){
// outer loop


for(int z_ind=0;z_ind<Nx;z_ind++){
   ...
   // inner loop
   for (int k=1;k<Nt;k++){
   // solve Bloch equation
   }
  }
}
```

Listing 3.12: Parallel CUDA code.

```
// parallel CUDA code version
template<typename TType>
__global__
void cnBlochKernel(...){
// outer loop
int z_ind = blockIdx.x*blockDim.x
          + threadIdx.x;
if (z_ind<Nx){
   ...
   // inner loop
   for (int k=1;k<Nt;k++){
   // solve Bloch equation
   }
  }
}
```

The parallel CUDA code is executed by each of the `Nx` threads replacing the sequential `for` loop iterations. Each thread is identifiable by means of a linear index `z_ind`. The `if` satement ensures that only `Nx` threads execute the enclosed code, which is the solution of the Bloch equation in this case. The remaining threads stay idle and no code is executed. Thus, when `z_ind` overshoots the `Nx` dimension of the data containers no memory is read or written, preventing the occurrence of false memory operations or, even worse, segmentation faults.

applyHessGPU In the GPU implementation the SSCAL/DSCAL and zero-padding operation, prior to the invocation of `applyHessKernel`, is performed on Thrust `device_vector` containers and, beside that, no change in code is required (see listing 3.13).

Listing 3.13: SSCAL/DSCAL and zero padding on the GPU.

```
// ...
// compute first column of Hdu
// perform SSCAL/DSCAL res = scalar*x
using namespace thrust::placeholders;
thrust::transform(du_dev.begin(), du_dev.end(), Hdu_dev.begin(),
      (DType)alpha[0] * _1);

//zero padding du to readout time
du_dev.resize(Nt - 1);
thrust::fill(du_dev.begin() + Nu, du_dev.end(), 0);
// ...
```

The Thrust library automatically distributes the computation to the GPU and executes the parallel GPU implementation of `transform` and `fill`. Therefore, Thrust allows to easily switch between CPU and GPU calculations depending on the underlying container type. One important consideration in the CUDA implementation of the `applyHessKernel` is the maximum kernel execution time. On Windows, the maximum run time of individual kernels is limited to approximately 5 seconds. Exceeding this time the Windows watchdog timer interferes and causes programs, using the primary GPU for computation and display, to time out. Thus, to provide short kernel runtimes the `applyHessKernel` is split into consecutive sub-kernels as shown in listing 3.14. Each sub-kernel requires the data of the previous kernel. CUDA kernel calls are asynchronous, meaning that the control is returned to the CPU as soon as the kernel is invoked. To ensure that all device operations have finished, the `cudaDeviceSynchronize` API function is called subsequential to each sub-kernel call. Due to limited device memory the container `dMz` is used to store the output data of, both, `solveLinStateEq` and `solveLinAdjointEq`. Hence, prior to solution of the linearized adjoint model `dMz_dev` is set to zero using `thrust::fill`.

Listing 3.14: Splitting of `applyHessKernel` into consecutive sub-kernels.

```
// ...
// solve linearized Bloch equation
solveLinStateEqKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(...);
cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
// calculate necessary information for Hessian action
setdNzKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(...);
cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
// set container to zero
thrust::fill(dMz_dev.begin(), dMz_dev.end(), 0);
// solve linearized adjoint equation
solveLinAdjointEqKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(...);
cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
// compute action of the Hessian
actionOfHessianKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(...);
cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
// ...
```

Subsequently, the sum of rows of the `Hdu_dev` matrix is needed to obtain the final result. This is accomplished by means of a GPU version of the code snippet described in listing 3.8. If one exchanges the host container `Hdu_host` by a device container `Hdu_dev` Thrust automatically invokes the parallel GPU version of the algorithm. Finally, the result is transferred from the device to the host yielding the output argument of the MEX file `applyHessGPU`.

CUDA Error Handling The `CUDA_CHECK_ERROR()` after kernel invocation checks if the previous kernel launch was valid or if a CUDA error occurred. In the case of the occurrence of an CUDA error, a Thrust `system_error` exception is thrown, defining the error category and error message. In order to provide better performance for release builds this error handling can be disabled via the CMake configuration variable `CUDA_ERROR_CHECK`. The enclosed code in the `CUDA_CHECK_ERROR()` macro is only performed when CUDA error handling is enabled, otherwise nothing is executed.

3.6. Parallel OpenMP Implementation

Alternatively, a vendor free method for a parallel implementation of the RFcontrol project is the use of the OpenMP application programming interface. The intrinsic parallelism is utilized by means of multithreading on the CPU. In what follows, a brief description of the underlying OpenMP principles in the RFcontrol project is given.

Thrust Device Backend The Thrust library provides a possibility to change between device backend systems. The default system is CUDA, but one can easily change the system to OpenMP without major modifications to the source code. A global change of the backend system can be achieved through compiler options. Instead of applying a global system change, the Thrust system can be accessed directly by re-tagging an existing Thrust iterator to the OpenMP backend system.

Listing 3.15: Change of the Thrust device backend system to OpenMP.

```
1 ...
2 // compute first column of Hdu
3 // perform SSCAL/DSCAL res = scalar*x
4 using namespace thrust::placeholders;
5 thrust::transform(
6     thrust::reinterpret_tag<thrust::omp::tag>(du_host.begin()),
7     thrust::reinterpret_tag<thrust::omp::tag>(du_host.end()),
8     thrust::reinterpret_tag<thrust::omp::tag>(Hdu_host.begin()),
9     (DType)*(d.alpha) * _1);
10 ...
```

The code snippet in listing 3.15 demonstrates the application of a Thrust backend system change to the scalar vector multiplication described in preceding sections. The `reinterpret_tag` routine returns a copy of an iterator and changes the corresponding system tag to `omp`. As a result the `transform` primitive handles the passed containers as OpenMP device vectors and therefore the OpenMP implementation is called.

Parallel Region OpenMP uses preprocessor directives (i.e. pragmas) to specify parallel code sections. The pragma `omp parallel for` defines a parallel region and creates a specific number of threads to distribute the enclosed `for` loop iterations among them. In listing 3.17 the use of a parallel region is shown in the case of the `cnBlochKernel`.

Listing 3.16: Sequential C/C++ code.

```
// sequential code version
template<typename TType>
void cnBlochKernel(...){
// outer loop



for(int z_ind=0;z_ind<Nx;z_ind++){
  ...
  // inner loop
  for (int k=1;k<Nt;k++){
  // solve Bloch equation
  }
 }
}
```

Listing 3.17: Parallel OpenMP code.

```
// parallel OpenMP code version
template<typename TType>
void cnBlochKernel(...){
// outer loop
int threads=omp_get_max_threads();
omp_set_num_threads(threads);
#pragma omp parallel for
for(int z_ind=0;z_ind<Nx;z_ind++){
  ...
  // inner loop
  for (int k=1;k<Nt;k++){
  // solve Bloch equation
  }
 }
}
```

The maximum number of available threads is obtained by calling the runtime API function `get_max_threads`. This maximum number is determined by the used multi-core CPU. Subsequently, the the number of forked threads is set to the obtained maximum by invocation of the `omp_set_num_threads` runtime function. The computation of the outer loop is shared between the forked CPU threads, whereby each individual thread identifier represents a loop iteration `z_ind`. After completion the forked threads join together in the master thread yielding the temporal evolution of the magnetization vector. Likewise, the outer loops in the `cnAdjointKernel` and `applyHessKernel` are parallelized. The calculation of the sum of rows of the `Hdu` matrix requires a re-tagging of the used iterators to locally change the Thrust backend system to OpenMP.

4. Results

The results in this section are obtained by using two different test systems named after their built-in GPU. The **GTX 970** workstation uses a four-core 64 bit processor (Intel i5-2500k) working at 3.3 GHz, 12 GB of RAM and a Nvidia GeForce GTX 970 with 4 GB device memory and 1664 CUDA cores. The **Tesla** workstation uses a six-core 64 bit processor (Intel i7-3930) working at 3.2 GHz, 64 of GB RAM and a Tesla C2075 GPU with 6 GB device memory and 448 CUDA cores. For a detailed list of the GPU specifications the reader is referred to the appendix and table A-1 therein.
The OC pulse used in this section was designed for an excitation of 6 simultaneous slices of width $\Delta w = 5mm$, a flip angle $\theta = 25°$ and a regularization parameter $\alpha = 1 \times 10^{-4}$. Relaxation effects were neglected in the optimization. The configuration parameters for the trust-region Newton-CG algorithm are described and listed in [13].

4.1. Optimization Time

The median $Q_{0.5}$ of the overall optimization time for the multi-slice OC pulse is shown in table 4.1. The timings are listed for different implementation methods: sequential MATLAB using `for`, MATLAB using `parfor` from the parallel computing toolbox, sequential C/C++ code, OpenMP multi-threading with multi-core CPU and CUDA GPU computing. The C/C++, OpenMP and CUDA implementations are realized by means of MEX files. For each implementation a total number of 10 runs (i.e. optimizations) was performed on the GTX 970 worksation. The number of locations N_x was set to 2048, 5001 and 10001.

The temporal grid size N_t was fixed to 697, and the underlying floating point format was set to double-precision.

Table 4.1.: Optimization times for different impelementation methods and varied N_x.

Implementation	**Optimization Time ($Q_{0.5}$) in s**		
	$N_x = 2048$	$N_x = 5001$	$N_x = 10001$
MATLAB sequential	634.2	1645.2	3020.8
MATLAB `parfor`	447.9	540.4	913.4
sequential C/C++	57.2	140.3	277.5
OpenMP	30.3	73.9	145.5
CUDA	1.6	3.2	5.9

The MATLAB `parfor` implementation shows significant fluctuations in the optimization time. In all other methods the execution time varies in the order of sub-seconds, therefore, only the MATLAB `parfor` implementation is further investigated. Figure 4.1 shows boxplots of the optimization times for different N_x values (10 runs per each value).

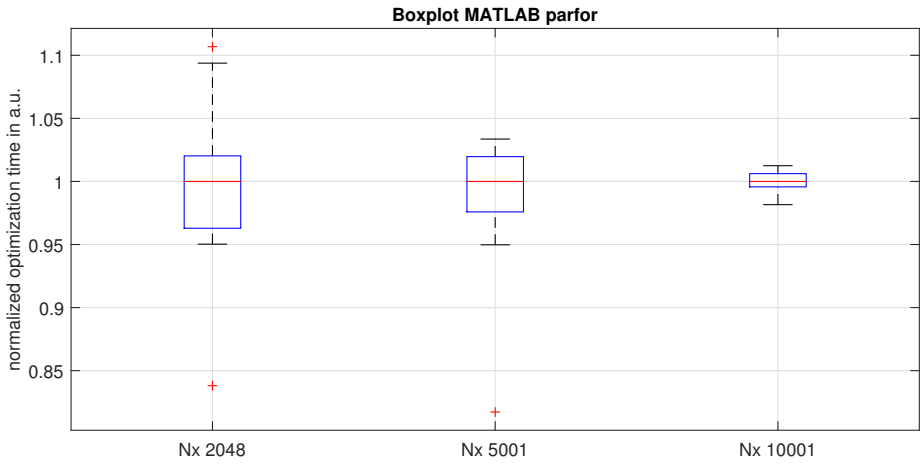


Figure 4.1.: Boxplots of the optimization times for three different spatial discretization points N_x. For each N_x value a total number of 10 runs was performed. The individual optimization times are normalized to their corresponding median.

Figure 4.2 illustrates the speedup of the different implementations with regard to the MATLAB sequential method. The speedup is obtained from the data in table 4.1. The median of the MATLAB sequential results is divided by the median of the corresponding implementation method. The speedup factors differ in the order of three magnitudes, thus, to provide large scale comparability the axis of ordinates is plotted on a logarithmic scale. In addition, the speedup of the CUDA implementation regarding to the other methods is shown in figure 4.3.

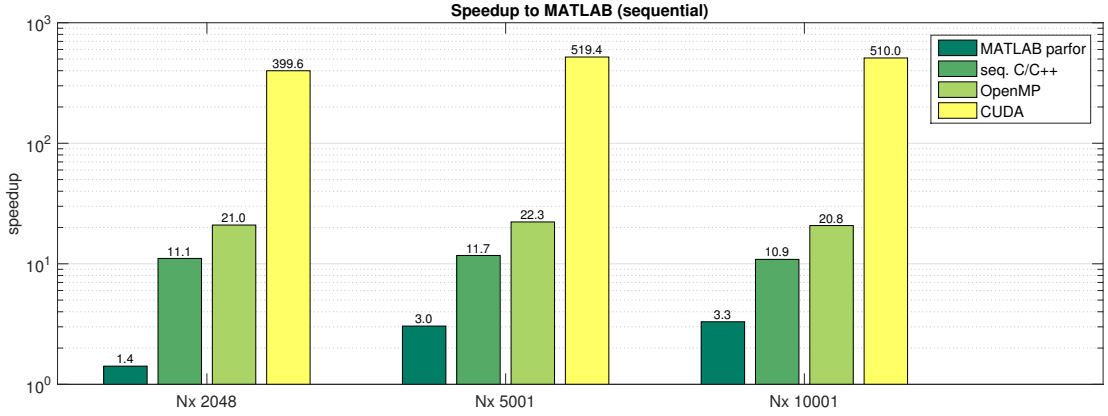


Figure 4.2.: Speedup of the different implementations with regard to the MATLAB single-core method for varying spatial discretization points N_x (using the GTX 970 workstation).

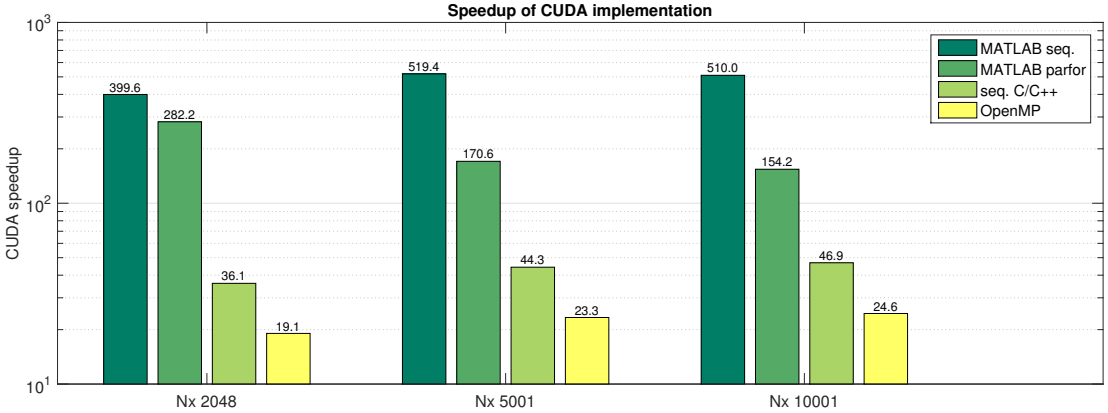
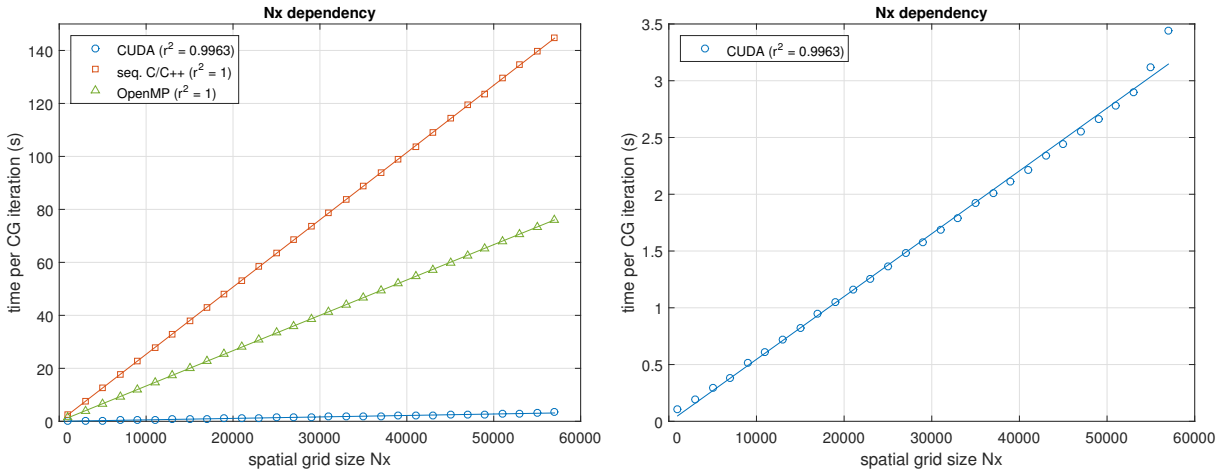


Figure 4.3.: Speedup of the CUDA implementation with regard to different methods for varying spatial discretization points N_x (using the GTX 970 workstation).

4.2. Problem Size Dependency

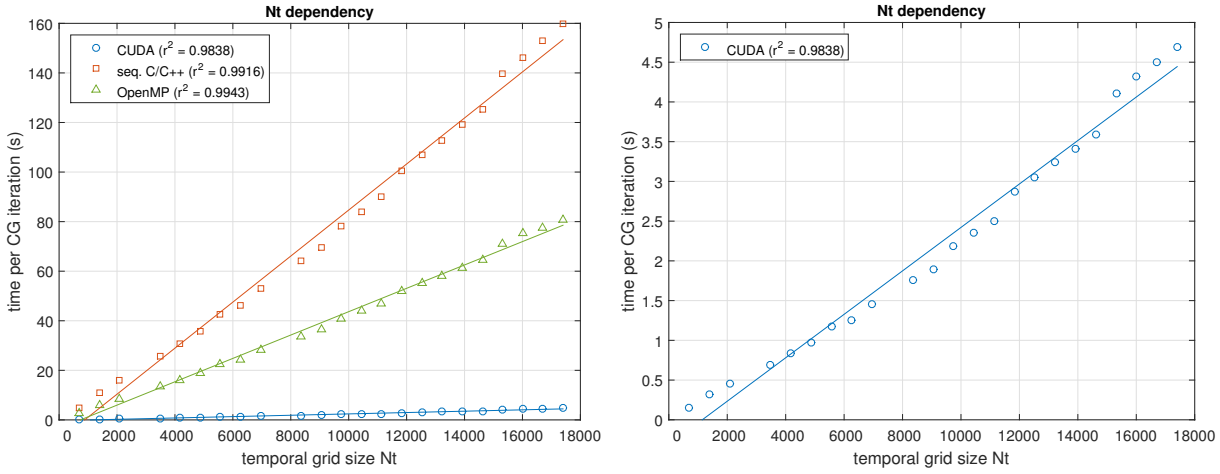
The dependency of the optimization time on the spatial grid size N_x for the sequential C/C++, OpenMP and CUDA implementation is shown in figure 4.2. The time values are normalized to the total number of CG iterations resulting in the time per CG iteration. The N_x values are increased in steps of 2000 until the device memory limit of the GTX 970 is reached. The N_x range of the CPU based methods is chosen equally to the GPU range, although, higher values would be possible (limited only by the host memory). The temporal grid size N_t is set to a constant value of 697 throughout the N_x alteration. The N_x dependency of the CUDA implementation is depicted separately in figure 4.2. In addition, a linear fit (least squares) is performed for each implementation and the r^2 values, rounded to four digits of precision, are listed.



(a) (b)

Figure 4.4.: Dependency of the optimization time on the spatial grid size N_x for the sequential C/C++, OpenMP and CUDA implementation (a). Additionally, the N_x dependency of the CUDA implementation is depicted in (b). The N_x parameter is incremented up to the device memory limit of the GTX 970. A linear fit (least squares) is performed and the r^2 values are listed (rounded to four digits of precision).

Considering the same OC pulse, the dependency of the optimization time (time per CG iteration) on the temporal grid size is demonstrated in figure 4.2. The N_t values are increased in steps of 697 until the device memory limit of the GTX 970 is reached and N_x is fixed to 2001 throughout the N_x alteration. The data points corresponding to grid sizes $4 \cdot N_t$ and $11 \cdot N_t$ were removed from the data set due to rejection of the last two Newton steps. Figure 4.2 shows an enlarged representation of the N_t dependency of the CUDA implementation. Likewise, a linear fit (least squares) is performed for each implementation and the r^2 values, rounded to four digits of precision, are listed.



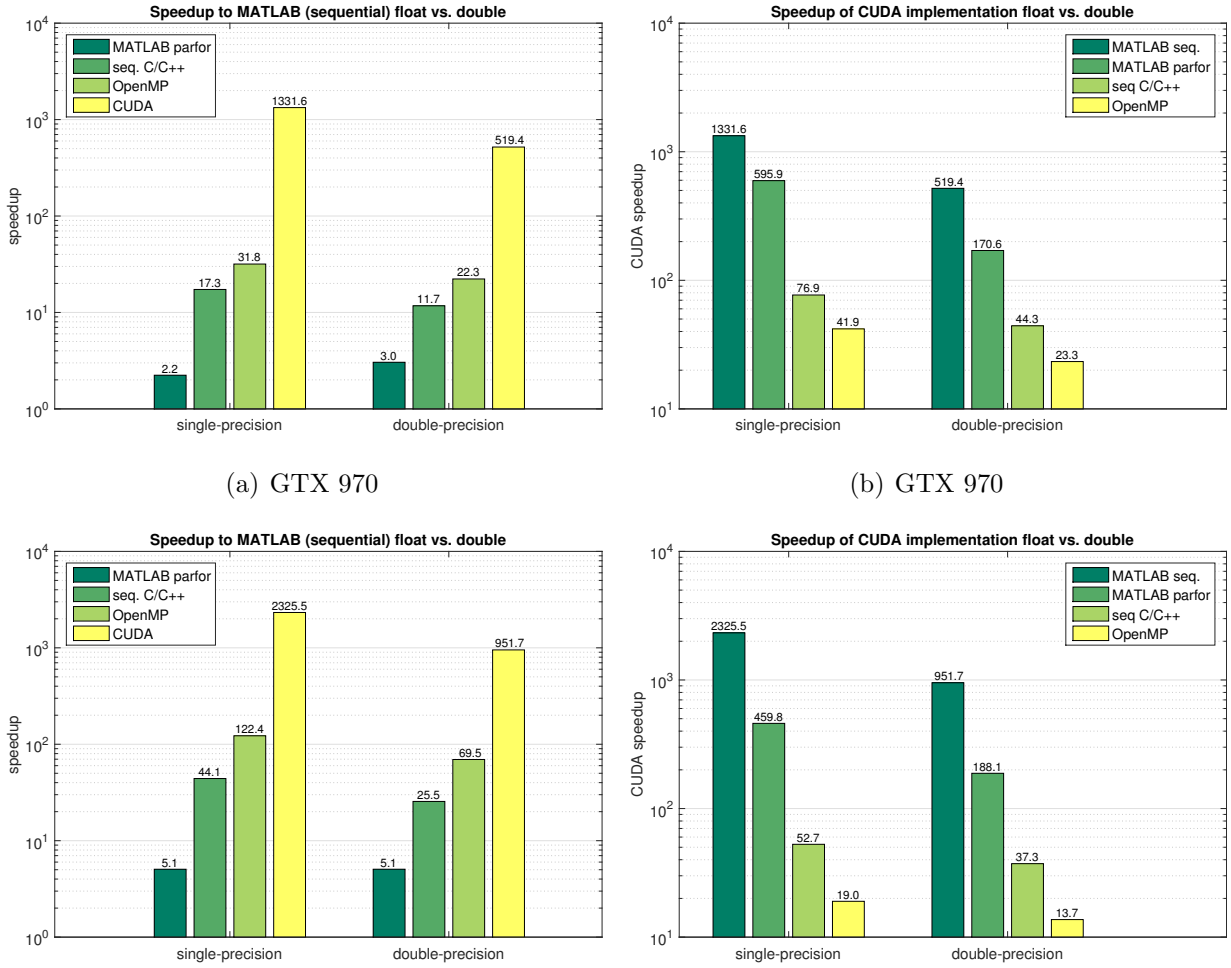
(a) (b)

Figure 4.5.: Dependency of the optimization time on the temporal grid size N_t for the sequential C/C++, OpenMP and CUDA MEX files (a). Additionally, the N_t dependency of the CUDA implementation is depicted in (b). The N_t parameter is incremented up to the device memory limit of the GTX 970. A linear fit (least squares) is performed and the r^2 values are listed (rounded to four digits of precision). The observations corresponding to grid sizes $4 \cdot N_t$ and $11 \cdot N_t$ were removed from the data set due to rejection of the last two Newton steps.

4.3. Single vs. Double

4.3.1. Speed

The impact of the underlying precision of the floating point format on the optimization time is illustrated in figure 4.6. A comparison of the achievable speedup with different implementation methods for single-precision (`float`) and double-precision (`double`) floating-point format is shown for the GTX 970 workstation in figure 4.6a and for the Tesla system in figure 4.6c. For single-precision data, the CUDA implementation offers an acceleration of about 1332 with regard to the MATLAB single-core implementation using the GTX 970 workstation. This is about 2.6 times faster than in the case of double-precision data. On the Tesla workstation the CUDA implementation provides a significant speedup of about 2336 with regard to the sequential MATLAB implementation using single-precision floating point format. All but the MATLAB `parfor` method, show an increase in speedup for single-precision compared to double-precision floating point format. Figures 4.6b and 4.6d demonstrate the achievable speedup with the CUDA implementation regarding to the other methods. In all cases the CUDA implementation offers a performance boost with underlying single-precision floating point format. On the Tesla system, utilizing a six-core i7-3930 CPU, the CUDA speedup regarding to the MATLAB `parfor`, sequential C/C++ and OpenMP implementation is not as high as on the GTX 970 workstation.



(a) GTX 970

(b) GTX 970

(c) Tesla

(d) Tesla

Figure 4.6.: Impact of the single-precision and double-precision floating point format on the achievable speedup in optimization time. A comparision of the different implementations with regard to the MATLAB single-core method is illustrated in (a) for the GTX 970 workstation and in (c) for the Tesla system. The speedup offered by the CUDA implementation is shown in (b) for the GTX 970 workstation and in (d) for the Tesla system. The grid sizes are fixed to $N_x = 5001$ and $N_t = 697$.

4.3.2. Accuracy

This section illustrates the influence of the underlying data type, i.e., single-precision or double-precision floating point, on the accuracy of the GPU optimization result. The root-mean-square error (RMSE) for single-precision and double-precision floating point is calculated for different temporal grid sizes N_t. The initial N_t value is 697 and the spatial grid size N_x is set to 5001. The RMSE is obtained between the desired magnetization profile $\mathbf{M_d}$ and the resulting profile of the OC pulse after Bloch simulation $\mathbf{M}$. Table 4.2 lists the results for single- and double-precision data on the GTX 970 and Tesla workstation. The RMSE values for single-precision and double-precision data on the GTX 970 system are about the same as on the Tesla workstation and only the difference values between $RMSE_s$ and $RMSE_d$ differ from each other marginally.

Table 4.2.: Root-mean-square error for single-precision data $RMSE_s$ and for double-precision data $RMSE_d$ for different multiples of the temporal grid size $N_t = 697$. The RMSE is obtained between the desired and simulated magnetization profile. The grid size N_x is set to 5001.

	Grid Size	$RMSE_s$ in a.u.	$RMSE_d$ in a.u.	$\lvert RMSE_s - RMSE_d \rvert$ in a.u.
GTX 970	N_t	11.6474×10^{-3}	11.6476×10^{-3}	1.4214×10^{-7}
	$4\ N_t$	10.7154×10^{-3}	10.7153×10^{-3}	0.3829×10^{-7}
	$8\ N_t$	10.6970×10^{-3}	10.6966×10^{-3}	4.2352×10^{-7}
Tesla	N_t	11.6474×10^{-3}	11.6476×10^{-3}	1.3841×10^{-7}
	$4\ N_t$	10.7154×10^{-3}	10.7153×10^{-3}	0.3642×10^{-7}
	$8\ N_t$	10.6970×10^{-3}	10.6966×10^{-3}	4.2538×10^{-7}

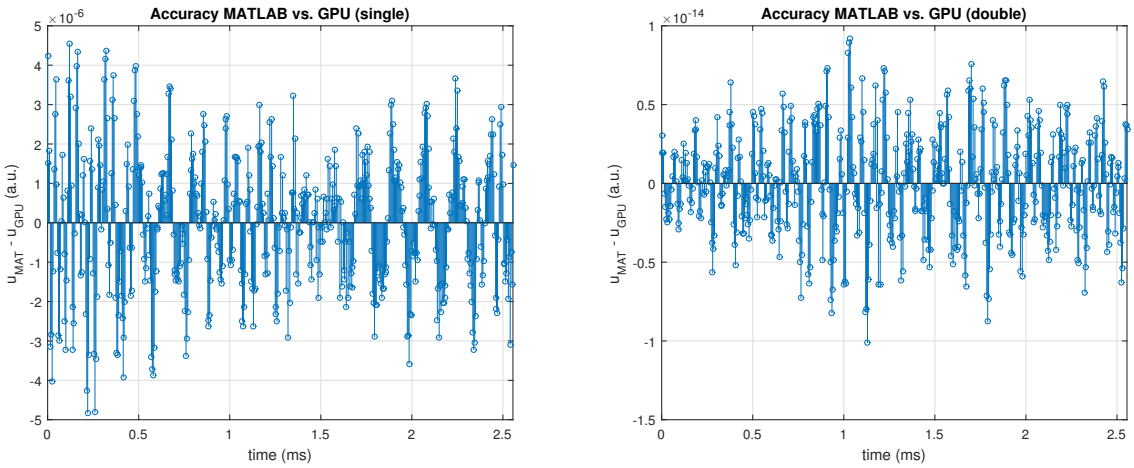
A comparison in accuracy of the GPU and MATLAB implementation is shown in table 4.3. The slice profile RMSE of the GPU and MATLAB results is listed for single- and double-precision data and for both workstations. Moreover, the mean absolute error (MAE) between the GPU and MATLAB pulse is stated. The temporal grid size N_t is set to 697 and the spatial grid size N_x to 5001. The difference between RMSE for the GPU and MATLAB result differs from each other marginally and lies in the order of machine precision for single- and double-precision floating point. A difference plot of the OC pulses obtained by means of the GPU and MATLAB implementation is shown in figure 4.7a for single-precision and in figure 4.7b for double-precision floating point format.

Table 4.3.: Root-mean square error for the GPU implementation $RMSE_{GPU}$ and MATLAB implementation $RMSE_{MAT}$ for single- and double-precision. The RMSE is obtained between the desired and simulated magnetization profile. The grid sizes N_t and N_x are set to 697 and 5001, respectively.

	Data Type	$RMSE_{GPU}$ in a.u.	$RMSE_{MAT}$ in a.u.	$\lvert RMSE_{GPU} - RMSE_{MAT} \rvert$ in a.u.
GTX 970	single	11.6474×10^{-3}	11.6475×10^{-3}	5.1223×10^{-8}
	double	11.6476×10^{-3}	11.6476×10^{-3}	3.4694×10^{-17}
Tesla	single	11.6474×10^{-3}	11.6475×10^{-3}	9.9652×10^{-8}
	double	11.6476×10^{-3}	11.6476×10^{-3}	3.4694×10^{-17}

Table 4.4.: Mean absolute error MAE_u and difference in J between the GPU pulse and MATLAB pulse. The grid sizes N_t and N_x are set to 697 and 5001, respectively.

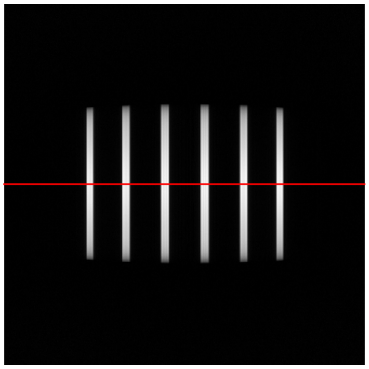
	Data Type	MAE_u in a.u.	$\lvert J(\mathbf{u_{MAT}},\mathbf{M}(T)) - J(\mathbf{u_{GPU}},\mathbf{M}(T))\rvert$ in a.u.
GTX 970	single	6.9827×10^{-7}	2.9104×10^{-11}
	double	1.3193×10^{-15}	5.4210×10^{-20}
Tesla	single	7.4633×10^{-7}	2.1828×10^{-11}
	double	1.3389×10^{-15}	1.3553×10^{-20}



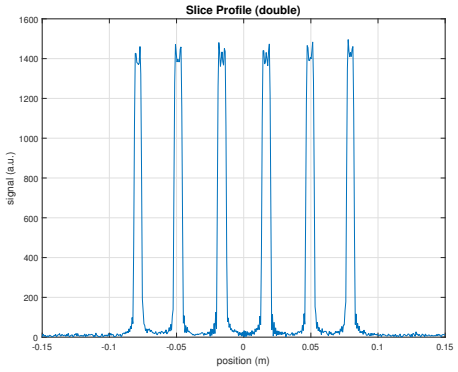
(a) single-precision

(b) double-precision

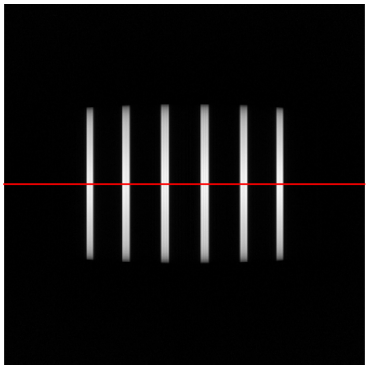
Figure 4.7.: Difference plot between the OC pulse obtained from the MATLAB $\mathbf{u_{MAT}}$ and GPU $\mathbf{u_{GPU}}$ implementation for single- and double-precision using the GTX 970 workstation. The grid sizes N_t and N_x are set to 697 and 5001, respectively.



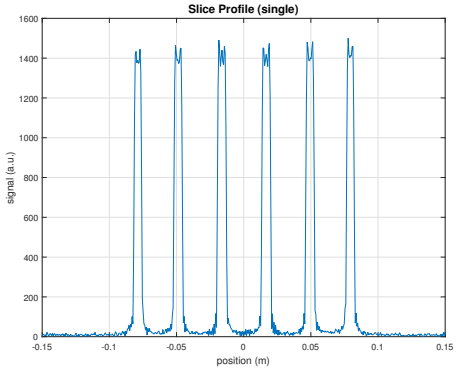
(a) reconstructed excitation (double)



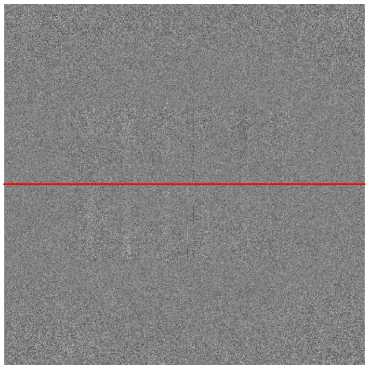
(b) measured slice profile (double)



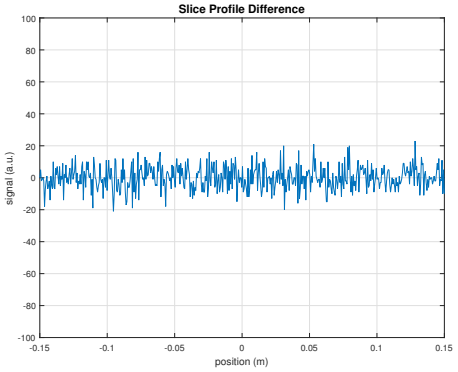
(c) reconstructed excitation (single)



(d) measured slice profile (single)



(e) difference (a) − (c)



(f) slice profile difference

Figure 4.8.: Reconstructed excitation (magnitude) obtained with a SMS pulse using (a) double-precision and (b) single-precision optimization. Additionally, the difference between (a) and (c) is shown in (e). The corresponding slice profile evaluations along the red lines are depicted in the right column.

The OC 6 SMS pulses optimized using single-precision and double-precision floating point format are validated by means of experimental measurements using a cylindrical phantom. The measurements were performed on a 3T MR scanner (Magnetom Skyra, Siemens Healthcare, Erlangen, Germany) and a modified FLASH sequence. In order to visualize the excitation pattern, the read-out gradient is changed from the frequency encoding axis to the slice direction. The sequence parameters are: repetition time $TR = 1000\ ms$, echo time $TE = 5\ ms$, field of view $FoV = 300 \times 300\ mm$, imaging matrix 512×512, slice thickness $THK = 5\ mm$ and flip angle $\theta = 25°$. In the left column of figure 4.8 the reconstructed excitations of the OC pulses optimized with double-precision and single-precision are shown, and in addition, the difference between both reconstructed images is depicted. The corresponding slice profile evaluations (along the red lines) are listed in the right column of figure 4.8.

4.4. CUDA Implementation Profiler Results

The computation of the single-slice OC pulse using the sequential MATLAB implementation leads to the profiler results summarized in figure 3.2. The former major bottlenecks are implemented as CUDA MEX files `applyHessGPU`, `cn_adjointGPU` and `cn_blochGPU` exploiting the underlying parallelism. The MATLAB profiler results of the CUDA implementation are listed in figure 4.9.

tr_newton (1 call, 5.655 sec)

Children (called functions)

Function Name	Function Type	Calls	Total Time	% Time
tr_cg (b)	function	4	3.115 s	55.1%
objfun (c)	function	9	2.025 s	35.8%
...pplyHessGPU(dpara,Xk,du,d.v,d.w,dims)	anonymous function	4	0.441 s	7.8%
num2str	function	31	0.010 s	0.2%
@(x,y)d.dt*(x'*y)	anonymous function	13	0.002 s	0.0%
...pplyHessGPU(dpara,Xk,du,d.v,d.w,dims)	anonymous function	4	0 s	0%
Self time (built-ins, overhead, etc.)			0.061 s	1.1%
Totals			5.655 s	100%

(a) `tr_newton.m`

tr_cg (4 calls, 3.115 sec)

Children (called functions)

Function Name	Calls	Total Time	% Time
...pplyHessGPU(dpara,Xk,du,d.v,d.w,dims)	28	3.109 s	99.8%
@(x,y)d.dt*(x'*y)	87	0.002 s	0.1%
tr_cg>dist2bdy	1	0 s	0%
Self time (built-ins, overhead, etc.)		0.004 s	0.1%
Totals		3.115 s	100%

(b) `tr_cg.m`

objfun (9 calls, 2.025 sec)

Children (called functions)

Function Name	Calls	Total Time	% Time
cn_blochGPU	9	0.530 s	26.2%
cn_adjointGPU	5	0.270 s	13.3%
squeeze	5	0.003 s	0.1%
Self time (built-ins, overhead, etc.)		1.222 s	60.3%
Totals		2.025 s	100%

(c) `objfun.m`

Figure 4.9.: Results of the Matlab profiler for the computation of a single-slice OC pulse. The former major bottlenecks (marked in red) are implemented as CUDA MEX files `applyHessGPU.m`, `cn_adjointGPU.m` and `cn_blochGPU.m`.

4.5. CUDA Kernel Execution Time

The execution time of the `applyHessGPU` kernel for the Geforce GTX 970 and Tesla C2075 GPU is depicted in figure 4.10. The demonstrated timings are obtained from 10 executions of the `applyHessGPU` kernel and the median values are listed. The spatial grid sizes N_x are set to 2048, 5001 and 10001. As can be seen in the median values, the Tesla system provides faster execution times for all three N_x sizes.

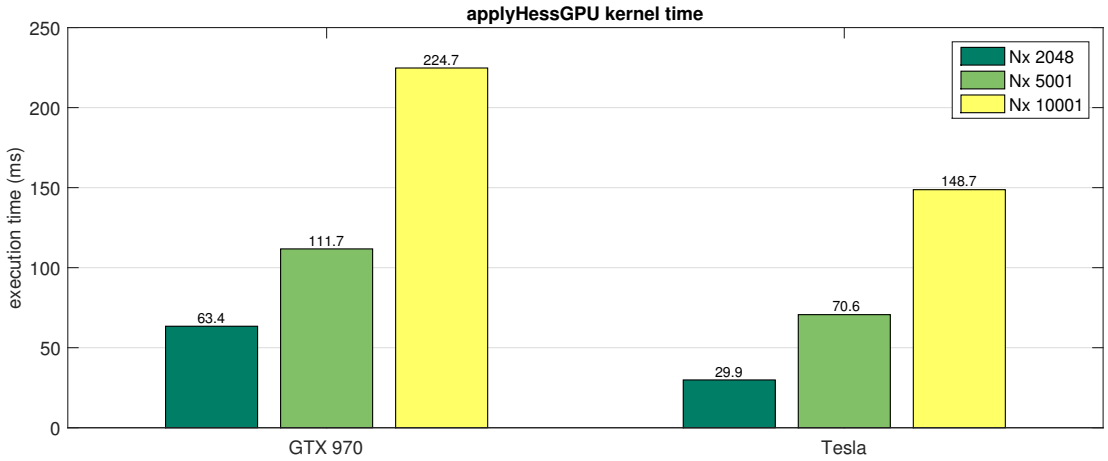
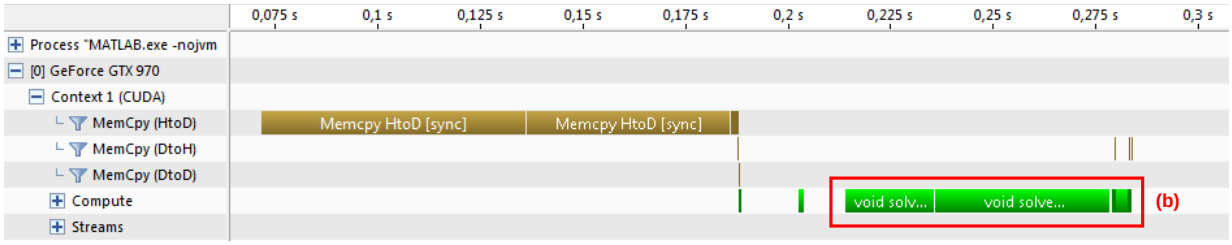


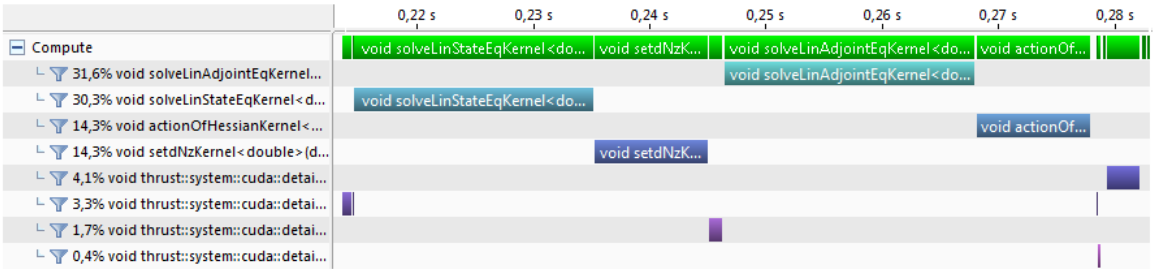
Figure 4.10.: Execution time of the `applyHess` CUDA kernel for different grid sizes N_x.

The Nvidia Visual Profiler provides the ability to examine CUDA kernel functions. Figure 4.11a shows the profiler results for the `applyHessGPU` MEX file utilizing the GTX 970 GPU. The spatial grid size N_x is set to 10001. The light brown blocks represent the timings required for CUDA memory transfer operations from the host to the device (HtoD) and back from the device to the host (DtoH). The green blocks illustrate the time spent on computation. The amount of time required for CUDA memory operations is about 114 ms, whereas, almost all of the time is spent on HtoD memory transfers. The time performing DtoH memory transfer operations is in the order of a few microseconds and is therefore negligible. The time performing computation amounts to 68 ms. This implies that the execution time of the `applyHessGPU` kernel is dominated by the HtoD memory transfers.

Figure 4.11b depicts a zoomed version of the computation part in the kernel, where the splitting of the `applyHessGPU` kernel in consecutive sub-kernels can be seen (described in section 3.5).



(a)



(b)

Figure 4.11.: Nvidia Visual Profiler result of the `applyHessGPU` MEX file (a) for a spatial grid size N_x of 10001. The light brown blocks show the time spent on memory transfer operations from the host to the device (HtoD) and from the device to the Host (DtoH). The green blocks represent the time spent on computation. A zoomed version of the compute block is illustrated in (b). The `applyHessGPU` kernel is split into consecutive sub-kernels.

5. Discussion and Conclusion

Implementation The numerical solution of the Bloch equation describes the evolution of the magnetization vector in presence of an external magnetic field and requires a discretization in time and spatial domain. The spatial domain $[-a, a]$ is discretized by a spatial grid $-a = z_1 < ... < z_{N_x} = a$ with spacing $\Delta z_m := z_m - z_{m-1}$. In each point z_i the state and adjoint equation can be solved independently. In standard serial code implementations the temporal evolution of the magnetization vector has to be computed one after another by iterating over each spatial point z_i. However, the spatial independence allows a direct application of parallel computing techniques to exploit the underlying parallelism.

The optimal control RF pulse design framework [13] was originally implemented entirely in MATLAB using the parallel computing toolbox (`parfor`). The MATLAB profiler results in figure 3.2 identifies the three subroutines `cn_bloch.m`, `cn_adjoint.m` and `applyHess.m` as the major bottlenecks in the application. Therefore, they are primary targets for performance optimization, which is achieved by means of the MATLAB executable interface. Different implementation methods are applied to generate MEX files of the three computational demanding functions, beginning with a sequential C/C++ version and followed by parallel versions implemented in OpenMP and CUDA.

The build process of the MEX files is embedded in the RFcontrol project utilizing CMake, an open-source software supporting cross-platform builds for various operating systems. CMake is designed to generate build files for the native build environment, like makefiles on Unix and Microsoft Visual Studio solution files on Windows. The CMake framework of the RFcontrol project provides a flexible option to configure the build process of the MEX files via CMake environment variables. It allows the configuration of the data type

to be used (single- or double-precision), CUDA error checking and occupancy based kernel launch. In addition, the installed CUDA toolkit version is identified and CUDA capable GPUs are detected automatically prior generation of the build file. If there is no CUDA capable GPU available the building of the CUDA MEX files is disabled and only the CPU based implementation methods are used.

Thrust Library The RFcontrol project is implemented by means of the Thrust library, a powerful library of parallel algorithms and data structures, to handle data transfers between host and device [41]. Since CUDA toolkit version 4.0 the Thrust library is already included and no separate installation is required. This straightforward accessibility is one major advantage of the Thrust library. In order to provide comparability between the implementation methods, the sequential C/C++ MEX files are working on Thrust host containers, although this is not mandatory and the C++ STL library could be used as well. Another advantage is the possibility to switch the device backend system (e.g. OpenMP) with only minor adaptations of the source code. Furthermore, Thrust increases the readability and usability of code by hiding CUDA specific memory allocation routines, such as `cudaMalloc`, `cudaMemcpy` and `cudaFree`. A remarkable demonstration of the capability of the Thrust library is shown in listing 3.8. The sum along the rows of the `Hdu` matrix is calculated using the `reduce_by_key` primitive in conjunction with fancy iterators like the `transform_iterator`. In order to overcome the column- vs. row-major memory alignment issue an implicit transposition of the `Hdu` matrix is achieved by means of a `permutation_iterator`. As a result of this implicit transposition the `reduce_by_key` primitive iterates over the elements in the `Hdu` matrix as they were stored in row-major order, without any allocation of additional memory or `memcpy` operations. The `reduce_by_key` primitive returns the used keys as output argument, which are not required in case of reduction of the `Hdu` matrix. The `discard_iterator` is used to ignore the output keys written to it without wasting memory capacity or bandwidth.

Optimization Time A measure for the overall execution time was obtained by determining the median $Q_{0.5}$ out of 10 optimization runs per implementation method (table 4.1). For instance, the sequential MATLAB implementation terminates approximately after 27 minutes on the GTX 970 workstation using a spatial grid size N_x of 5001 and double-precision floating point format. Just by implementing the subroutines with high computational effort as MEX files, using sequential C/C++ code, an optimization time of about 2 minutes and 20 seconds is achieved. This implies a notable speedup of 11.7 without any utilization of the underlying parallelism (figure 4.2). The CUDA implementation exploits the intrinsic parallelism and reduces the overall execution time to 3.2 seconds, which results in a remarkable speedup of 519.4. If compared to the OpenMP MEX file solution utilizing an Intel i5-2500k CPU (4 cores), the CUDA accelerated MEX files still offer a considerable speedup of 20.3 (CUDA speedup in figure 4.3). On the Tesla workstation, which uses an Intel i7-3930 CPU with 6 cores, the CUDA acceleration factor is still 13.7 (figure 4.6d).
As noted in section 4.1, the MATLAB `parfor` implementation shows significant fluctuations in the optimization time, whereas, all other methods vary in the order of sub-seconds. Therefore, only the MATLAB `parfor` implementation was further investigated and a boxplot is illustrated in figure 4.1. For the sake of comparability the individual optimization times are normalized to their corresponding median. As can be seen, the fluctuations decrease with increasing grid size N_x. A possible explanation is, that the operating system is not idle throughout the optimization and may cause interferences, when all CPU cores are utilized to full capacity. Therefore, one CPU core should always be dedicated to the operating system to provide a certain robustness of the execution times.

Problem Size Dependency In the optimal control RF pulse design framework the problem size is defined by the number of spatial discretization points N_x, temporal discretization points N_t and control points N_u. Consequently, the size of the used data structures is a combination of this problem dimensions. The exploited parallelism lies in the N_x dimension, i.e., for each spatial discretization point the state and adjoint equation can be solved

independently. On the contrary, the N_t dimension offers no parallelism, as for the temporal evolution of the magnetization vector the previous time point is mandatory for the computation. In section 4.2 the dependency of the optimization time on the grid sizes N_x and N_t was investigated for the sequential C/C++, OpenMP and CUDA implementation on the GTX 970 workstation. The N_x dependency is shown in figure 4.4, and the listed r^2 values close to 1 indicate a linear scaling of the optimization time with increasing grid size N_x. Note, that the r^2 values are not exactly 1, but rounded to four digits of precision, in case of the sequential C/C++ and OpenMP method. As can be seen in figure 4.4b, the last data point lies significantly above the linear least square fit. Since the N_x values are increased up to the device memory limit of the GeForce GTX 970 almost the total amount of device memory is occupied at this data point, and the impact of the memory issue[1] could become noticeable. Particularly, the GTX 970 device memory is segmented into a 3.5 GB section and a 0.5 GB section, whereas the 0.5 GB part is not accessed as efficiently as the main part. The N_t dependency is depicted in figure 4.5 and the r^2 values close to 1 also indicate a linear scaling of optimization time with increasing temporal resolution. However, this evaluation is not as straightforward as in case of the N_x dependency, because an additional sampling of the slice selection gradient by the same factor is required. The data points corresponding to a grid size of $4 \cdot N_t$ and $11 \cdot N_t$ were excluded from the data set due to rejection of the last two Newton steps and therefore making them not suitable for a comparison.

Single vs. Double: Speed The data type, to be used in the RFcontrol project, can be configured to single-precision (`float`) and double-precision (`double`) floating point format by setting the CMake variable `USE_DOUBLE_PRECISION` accordingly. Figure 4.6 shows the impact of single- and double-precision on the achievable speedup. All but the MATLAB `parfor` implementation show an increase in speedup when using the `float` data type. MATLAB naturally operates on double-precision data and is not optimized for single-

[1] statement of NVIDIA on http://www.pcper.com

precision performance, although it is possible to cast a variable of type `double` to a `single` variable. This also explains why the speedup of the MATLAB `parfor` implementation is reduced from 3.0 to 2.2 by switching from `double` to `single` on the GTX 970 workstation (see figure 4.6a). The GeForce GTX 970 is based on the second generation Maxwell hardware architecture (GM 204 chip) and supports CUDA compute capability 5.2. The double-precision performance of the Maxwell chip is 1/32 of single-precision performance. Although the Tesla C2075 is based on the older Fermi architecture (compute capability 2.0) the ratio of double-precision to single-precision performance is 1/2. Precisely, the GTX 970 and Tesla 2075 deliver 122 versus 515 GFLOPS for double-precision operations, respectively. The benefits of this additional GFLOPS of the Tesla GPU can be seen in figure 4.10, where timings of the `applyHessGPU` MEX file are listed without embedding in the algorithmic MATLAB framework. This and the six-core Intel i7-3930 CPU contribute to a better overall performance of the Tesla workstation, for both single- and double-precision floating point format.

Single vs. Double: Accuracy Table 4.2 lists the slice profile RMSE values for single- and double-precision data. The $RMSE_s$ and $RMSE_d$ values are about the same and the difference between them lies in the range of 4 times the numerical accuracy of single-precision floating point format ($\approx$ `1.19e-07` in MATLAB). For instance, using a temporal resolution of $8 \cdot N_t$ this difference is about 3.5 times above the numerical accuracy. If one takes a closer look to the output of the trust-region Newton-CG algorithm in listing 5.1, the norm of the gradient $|\mathbf{g}|$ is close to the numerical accuracy of single-precision. This results from an absolute tolerance of `1.2e-07` for the gradient norm in the Newton step, which is in the order of single-precision accuracy. Consequently, the algorithm may terminate at a slightly different local minimizer to the objective J for single- and double-precision. Therefore, the values of J after termination of the algorithm are compared in a further investigation. For single the value of J equals to `1.313229e-04` and in the case of double J equals to `1.313141e-04` resulting in a difference of `8.729808e-09`, which lies

in the order of numerical accuracy. Hence, it follows that the results of J are equal in the sense of numerical accuracy for single-precision floating point format. In a further step, the absolute tolerance for the gradient norm is set to a value of `1.2e-06` for a repeated evaluation of the RMSE differences. This change in absolute tolerance results in RMSE differences in the order of one magnitude smaller than the values in table 4.2. In this context, this implies no significant change in accuracy of the slice profile by switching from single-precision to double-precision.

Listing 5.1: Output of the TR-Newton-CG algorithm with `abstol=1.2e0-7`.

```
Computing minimizer with alpha = 0.0001
Using GPU acceleration (single)
it  J     |g|     flag  rho     dJa/dJm   cgits
0 2.764e-03 4.649e-03
1 1.617e-04 5.049e-04 2 2.000e+00 9.671e-01 1
2 1.313e-04 3.393e-06 0 2.000e+00 1.002e+00 2
3 1.313e-04 1.459e-07 0 2.000e+00 9.985e-01 8
Elapsed time is 11.798seconds.
-----------------------------------------------------------------
Computing minimizer with alpha = 0.0001
Using GPU acceleration (double)
it  J     |g|     flag  rho     dJa/dJm   cgits
0 2.764e-03 4.649e-03
1 1.617e-04 5.048e-04 2 2.000e+00 9.671e-01 1
2 1.313e-04 3.393e-06 0 2.000e+00 1.002e+00 2
3 1.313e-04 1.458e-07 0 2.000e+00 1.000e+00 8
Elapsed time is 23.2485 seconds.
-----------------------------------------------------------------
```

In order to investigate the results of the OC pulses, optimized with single- and double-precision floating point format, experimental measurements were performed using a cylindrical phantom. The reconstructed excitations and corresponding slice profile evaluations (along the red lines) in figure 4.8 show no significant visual difference. Therefore, the difference between the reconstructed excitation for double-precision and single-precision is computed in figure 4.8e. The evaluation of the slice profile difference in figure 4.8f shows a slight in-slice and out-off-slice error of less than 1%. Hence, no significant loss of accuracy in the measured excitation is noticeable by changing the floating point format to single-precision in the optimization.

The comparison in accuracy of the GPU and MATLAB implementation in table 4.3 states that the $RMSE_{GPU}$ and $RMSE_{MAT}$ values differ form each other marginally and the difference between them lies in the order of the corresponding numerical accuracy ($\approx$ `2.22e-16` for `double`). The MAE values in table 4.4 of the control $\mathbf{u}$ are above the range of numerical precision due to marginally differing termination criteria for single- and double-precision. An investigation of the objective values J, after termination of the algorithm, shows differences in the order of numerical accuracy (table 4.4). This implies no significant change in accuracy of the optimization results by use of GPU accelerated MEX files.

CUDA Implementation Figure 4.9 summarizes the MATALB profiler results of the GPU accelerated computation of the same single-slice OC pulse as used in figure 3.2. The former major bottlenecks are implemented as CUDA MEX files `applyHessGPU`, `cn_adjointGPU` and `cn_blochGPU` exploiting the provided parallelism. The total number of CG iterations and function calls remains unchanged. However, the overall execution time of the `tr_newton.m` function has been significantly reduced from former 1660.1 seconds to 5.7 seconds by means of GPU accelerated MEX files. As can be seen in figure 4.9c, the self time and MATLAB overhead becomes noticeable for optimization times in the order of a few seconds. In particular, in the `objfun.m` sub-routine the self time has a share of 60.3% of total execution time.

The Nvidia Visual Profiler result of the `applyHessGPU` kernel using a grid size N_x of 10001 is illustrated in figure 4.11. A major part of execution time is spent on memory transfer operations (114 ms), whereas the time spent on computation amounts to 68 ms resulting in a compute to memory transfer ratio of about 0.6. Therefore, the amount of time performing compute operations is low relative to the amount of time required for CUDA memory operations. As can be seen from the brown blocks in the kernel profile, the `applyHessGPU` kernel is dominated by memory transfers from the host to the device (HtoD). Hence, for best overall application performance it is important to minimize the data transfers between host and device. The RFcontrol framework is based on acceleration by means of

MEX file implementation of the major bottlenecks. This outsourcing of computational demanding sub-routines provides high flexibility as the main part of the algorithm remains in the MATLAB programming language, which offers a high level of abstraction. However, this flexibility comes at the cost of increased HtoD data transfers, because each MEX function call requires the results of previous computations making it difficult to minimize data transfers. In the `applyHessGPU` kernel it was possible to reduce the data transfers from the device back to the host (DtoH) to a minimum, because only the reduced `Hdu` matrix, a vector with N_u elements, needs to be copied in DtoH direction. Possible optimizations could be the use of constant and texture memory for read-only data like the target magnetization profile. In the former case, constant memory is limited to 64 KB, and for instance, the target magnetization profile for a typical grid size N_x of 5001 requires $3 \times 5001 \times$ `sizeof(double)` ($\approx$ 117 KB) and therefore excluding the use of constant memory. In the latter case, texture memory offers limited compatibility for `double` data. Another possibility would be the use of shared memory blocking, due to the fast nature of this memory type. In the RFcontrol project, with dynamic problem dimensions, this is rather difficult to achieve without loss of parallelism, because in each location the solution of the state and adjoint equation (inner loop) requires all time points N_t and the shared memory per thread block is limited to 48 KB (GTX 970). The CUDA programming model offers a wide variety of memory optimizations, but a detailed discussion of all the possibilities would be beyond the scope of this work, therefore, the reader is referred to the CUDA C Best Practices Guide [42]. A zoomed version of the compute block in figure 4.11b illustrates the splitting of the `applyHessGPU` kernel into consecutive sub-kernels in order to prevent the interference of the Windows watchdog timer on the GTX 970 workstation (see section 3.5). In order to overcome this problem, the use of two GPUs is recommended, where one is dedicated to the display driver and the second only to compute purposes as on the Tesla workstation.

The GPGPU application interface used in the RFcontrol project is NVIDIAs proprietary CUDA programming model, restricting the application to CUDA capable GPUs. A vendor-independent framework for GPGPU is OpenCL, an open standard from the Khronos Group

[2]. However, given the access to CUDA capable GPUs and the fact that the method of choice for GPU accelerated applications at our institute is CUDA C/C++ this programming model was choosen for the GPGPU implementation.

Conclusion The demonstrated multi-platform implementation of the OC RF pulse design framework by means of parallel computing accelerated MATLAB executable files leads to a significant reduction in computing time while maintaining the high flexibility of the MATLAB environment. Above all, the remarkable speedup of the CUDA implementation allows computation times in the order of a few seconds therefore making real-time optimization and patient-specific design feasible. Moreover, even a higher acceleration without significant loss in accuracy is possible by changing the underlying floating point format from double-precision to single-precision. While the CUDA implementation requires a CUDA capable NVIDIA GPU, the additional OpenMP implementation utilizes the intrinsic parallelism by means of CPU multi-threading and provides a considerable reduction in optimization time independent from the built-in GPU. The feasibility of real-time optimization by means of CUDA accelerated OC RF pulse design method allows a direct integration in a MR pulse sequence and may be part of future work.

Bibliography

[1] Foster I: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995, p. 430.

[2] Kirk D and Hwu W: *Programming massively parallel processors : a hands-on approach*. 2010, p. 514.

[3] Almasi GS and Gottlieb A: *Highly Parallel Computing*. 2nd ed. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994.

[4] *OpenMP Application Program Interface*. 2013.

[5] Larsen ES and McAllister D: "Fast matrix multiplies using graphics hardware". In: *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM. 2001, pp. 55–55.

[6] *CUDA C Programming Guide*. 2015.

[7] Freiberger M, Knoll F, Bredies K, Scharfetter H, and Stollberger R: The Agile Library for Biomedical Image Reconstruction Using GPU Acceleration. *Computing in Science and Engineering* 15 (2013), pp. 34–44.

[8] Knoll F, Bredies K, Pock T, and Stollberger R: Second Order Total Generalized Variation (TGV) for MRI. 491 (2011), pp. 480–491.

[9] Tahayori B, Johnston LA, Mareels IMY, and Farrell PM: Revisiting the Bloch Equation through Averaging (2008), pp. 4121–4126.

[10] Nishimura DG: *Principles of Magnetic Resonance Imaging*. Standford University, 1996.

[11] Pauly J, Le Roux P, Nishimura D, and Macovski A: Parameter Relations for the Shinnar-Le Roux Selective Excitation Pulse Design Algorithm. *IEEE Transactions on Medical Imaging* 10(1) (1991), pp. 53–65.

[12] Conolly S, Nishimura D, and Macovski A: Optimal Control Solutions to the Magnetic Resonance Selective Excitation Problem. *IEEE Transactions on Medical Imaging* MI-5(2) (1986), pp. 106–115.

[13] Aigner CS, Clason C, Rund A, and Stollberger R: Efficient high-resolution RF pulse design applied to simultaneous multi-slice excitation. *Journal of Magnetic Resonance* (2015).

[14] Steihaug T: The Conjugate Gradient Method and Trust Regions in Large Scale Optimization. *SIAM Journal on numerical analysis* 20 (1983), pp. 626–637.

[15] Bernstein MA, King KF, and Zhou XJ: *Handbook of MRI Pulse Sequences*. Elsevier Academic Press, 2004, p. 1040.

[16] Lee KJ: General parameter relations for the Shinnar-Le Roux pulse design algorithm. *Journal of Magnetic Resonance* 186 (2007), pp. 252–258.

[17] Balchandani P, Pauly J, and Spielman D: Designing adiabatic radio frequency pulses using the Shinnar–Le Roux algorithm. *Magnetic Resonance in Medicine* 64 (2010), pp. 843–851.

[18] Grissom WA, McKinnon GC, and Vogel MW: Nonuniform and multidimensional Shinnar-Le Roux RF pulse design method. *Magnetic Resonance in Medicine* 68 (2012), pp. 690–702.

[19] Ma C and Liang ZP: Design of multidimensional Shinnar–Le Roux radiofrequency pulses. *Magnetic Resonance in Medicine* 73 (2015), pp. 633–645.

[20] Ugurbil K, Xu J, et al.: Pushing spatial and temporal resolution for functional and diffusion MRI in the Human Connectome Project. *NeuroImage* 80 (2013), pp. 80–104.

[21] Feinberg DA and Setsompop K: Ultra-fast MRI of the human brain with simultaneous multi-slice imaging. *Journal of Magnetic Resonance* 229 (2013), pp. 90–100.

[22] Müller S: Multifrequency Selective rf Pulse for Multislice MR Imaging. *Magnetic Resonance in Medicine* 6 (1988), pp. 364–371.

[23] Souza SP, Szumowski J, Dumoulin C, Plewes DP, and Glover G: SIMA: Simultaneous Multislice Acquisition of MR Images by Hadamard-Encoded Excitation. *Journal of Computer Assisted Tomography* 6 (1988), pp. 1026–1030.

[24] Larkman D, Hajinal JV, Herlihy AH, Coutts GA, Young IR, and Ehnholm G: Use of Multicoil Arrays for Seperation of Signal from Multiple Slices SiSimultaneous Excited. *Journal of Magnetic Resonance Imaging* 12 (2001), pp. 313–317.

[25] Breuer F, Blaimer M, Heidemann RM, Mueller MF, Griswold MA, and Jakob PM: Controlled Aliasing in Parallel Imaging Results in Higher Acceleration (CAIPIRINHA) for Multi-Slice Imaging. *Magnetic Resonance in Medicine* 53 (2005), pp. 684–691.

[26] Nunes RG, Hajinal JV, Golay J, and Larkman DJ: "Simultaneous slice excitation and reconstruction for single shot EPI". In: *Proc. ISMRM 14*. 2006, p. 293.

[27] Setsompop K, Gagoski BA, Polimeni JR, Witzel T, Wedeen VJ, and Wald LL: Blipped-controlled aliasing in parallel imaging for simultaneous multislice echo planar imaging with reduced g-factor penalty. *Magnetic Resonance in Medicine* 67 (2012), pp. 1210–1224.

[28] Blaimer M, Breuer FA, Seiberlich N, Mueller MF, Heidemann RM, Jellus V, Wiggins G, Wald LL, Griswold MA, and Jakob PM: Accelerated volumetric MRI with a SENSE/GRAPPA combination. *Journal of Magnetic Resonance Imaging* 24 (2006), pp. 444–450.

[29] Norris DG, Koopmans PJ, Boyacioglu R, and Barth M: Power independent of number of slices (PINS) radiofrequency pulses for low-power simultaneous multislice excitation. *Magnetic Resonance in Medicine* 66 (2011), pp. 1234–1240.

[30] Hargreaves BA, Cunningham CH, Nishimura DG, and Conolly SM: Variable-rate selective excitation for rapid MRI sequences. *Magnetic Resonance in Medicine* 52 (2004), pp. 590–597.

[31] Poser BA, Anderson RJ, Guérin B, Setsompop K, Deng W, Mareyam A, Serano P, Wald LL, and Stenger VA: Simultaneous multislice excitation by parallel transmission. *Magnetic Resonance in Medicine* 71 (2014), pp. 1416–1427.

[32] Bilgic B, Gagoski BA, Cauley SF, Fan AP, Polimeni JR, Grant PE, Wald LL, and Setsompop K: Wave-CAIPI for highly accelerated 3D imaging. *Magnetic Resonance in Medicine* 73 (2015), pp. 2152–2162.

[33] Gagoski BA, Bilgic B, Eichner C, Bhat H, Grant PE, Wald LL, and Setsompop K: RARE/Turbo spin echo imaging with simultaneous multislice Wave-CAIPI. *Magnetic Resonance in Medicine* 73 (2015), pp. 929–938.

[34] Guérin B, Setsompop K, Ye H, Poser BA, Stenger AV, and Wald LL: Design of parallel transmission pulses for simultaneous multislice with explicit control for peak power and local specific absorption rate. *Magnetic Resonance in Medicine* 1953 (2014), pp. 1946–1953.

[35] Xu D, King KF, Zhu Y, McKinnon GC, and Liang ZP: Designing multichannel, multidimensional, arbitrary flip angle RF pulses using an optimal control approach. *Magnetic Resonance in Medicine* 59 (2008), pp. 547–560.

[36] Vinding MS, Maximov II, Tošner Z, and Nielsen NC: Fast numerical design of spatial-selective RF pulses in MRI using Krotov and quasi-Newton based optimal control methods. *Journal of Chemical Physics* 137 (2012).

[37] Nocedal J and Wright S: *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer New York, 2006.

[38] Ruszczynski A: *Nonlinear Optimization*. Princeton University Press, 2011.

[39] Grama A: *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003.

[40] Hoffmann S and Lienhart R: *OpenMP: Eine Einführung in die parallele Programmierung mit C/C++*. Informatik im Fokus. Springer Berlin Heidelberg, 2008.

[41] Hoberock J and Bell N: *Thrust: A Parallel Template Library*. 2010.

[42] *CUDA C Best Practices Guide*. 2015.

Appendix

A. GPU Specifications

Table A-1.: Summary of GPU specifications.

Specifications	**GTX 970**	**Tesla C2075**
CUDA cores	1664	448
Base Clock	1050 MHz	1150 MHz
Device Memory	4 GB (3.5 GB)[2]	6 GB
Memory Bandwidth	224 GB/s	144 GB/s
Memory Interface	GDDR5	GDDR5
FP32 (Peak)	3920 GFLOPS	1030 GFLOPS
FP64 (Peak)	122 GFLOPS	515 GFLOPS
Compute Capability	5.2	2.0

[2]The GTX 970 device memory is segmented into a 3.5 GB section and a 0.5 GB section, whereas the 0.5 GB part is not accessed as efficiently as the 4.5 GB part.

B. CUDA Kernels

B.1. `applyHessGPU`

Listing B-1: Invokation of the `applyHessKernel` in file `applyHessGPU.cu`.

```
extern "C" void callApplyHessKernel(dpara<DType>* dpara_host, DType* N_host, DType* P_host,
  DType* u_host, DType* v_host, DType* w_host, DType* du_host,
  int Nt, int Nx, int Nu,
  DType* mat_output){

  // allocate device vectors
  thrust::device_vector<DType> Hdu_dev(Nu*(Nx + 1));
  thrust::device_vector<DType> N_dev(N_host, N_host + (3 * Nx*(Nt - 1)));
  thrust::device_vector<DType> P_dev(P_host, P_host + (3 * Nx*(Nt - 1)));
  thrust::device_vector<DType> u_dev(u_host, u_host + Nt);
  thrust::device_vector<DType> v_dev(v_host, v_host + Nt);
  thrust::device_vector<DType> w_dev(w_host, w_host + Nt);
  thrust::device_vector<DType> du_dev(du_host, du_host + Nu);

  // copy dpara to device
  thrust::device_vector<DType> xdis(dpara_host->xdis, dpara_host->xdis + Nx);
  thrust::device_vector<DType> gamma(dpara_host->gamma, dpara_host->gamma + 1);
  thrust::device_vector<DType> T1(dpara_host->T1, dpara_host->T1 + 1);
  thrust::device_vector<DType> T2(dpara_host->T2, dpara_host->T2 + 1);
  //thrust::device_vector<DType> M0c(dpara_host->M0c, dpara_host->M0c + 1);
  thrust::device_vector<DType> B1c(dpara_host->B1c, dpara_host->B1c + 1);
  thrust::device_vector<DType> G3(dpara_host->G3, dpara_host->G3 + 1);
  thrust::device_vector<DType> relax(dpara_host->relax, dpara_host->relax + 1);
  thrust::device_vector<DType> dt(dpara_host->dt, dpara_host->dt + 1);
  thrust::device_vector<DType> alpha(dpara_host->alpha, dpara_host->alpha + 1);
  thrust::device_vector<DType> dx(dpara_host->dx, dpara_host->dx + 1);

  // compute first column of Hdu
  // perform SSCAL/DSCAL res = scalar*x
  using namespace thrust::placeholders;
  thrust::transform(du_dev.begin(), du_dev.end(), Hdu_dev.begin(), (DType)alpha[0] * _1);

  //zero padding du to readout time
  du_dev.resize(Nt - 1);
  thrust::fill(du_dev.begin() + Nu, du_dev.end(), 0);

  // get raw pointer
  DType* Hdu = thrust::raw_pointer_cast(Hdu_dev.data());
  DType* N = thrust::raw_pointer_cast(N_dev.data());
  DType* P = thrust::raw_pointer_cast(P_dev.data());

  DType* u = thrust::raw_pointer_cast(u_dev.data());
  DType* v = thrust::raw_pointer_cast(v_dev.data());
  DType* w = thrust::raw_pointer_cast(w_dev.data());
  DType* du = thrust::raw_pointer_cast(du_dev.data());

  //init dpara with raw pointer
  dpara<DType> d;
  d.xdis = thrust::raw_pointer_cast(xdis.data());
  d.gamma = thrust::raw_pointer_cast(gamma.data());
  d.T1 = thrust::raw_pointer_cast(T1.data());
  d.T2 = thrust::raw_pointer_cast(T2.data());
  d.B1c = thrust::raw_pointer_cast(B1c.data());
  d.G3 = thrust::raw_pointer_cast(G3.data());
  d.relax = thrust::raw_pointer_cast(relax.data());
  d.dt = thrust::raw_pointer_cast(dt.data());
  d.alpha = thrust::raw_pointer_cast(alpha.data());
  d.dx = thrust::raw_pointer_cast(dx.data());
```

```
59
60    thrust::device_vector<DType> dMz_dev(3*Nt*Nx);
61    thrust::device_vector<DType> dNz_dev(3*Nu*Nx);
62    thrust::device_vector<DType> dq_dev(3 * Nx);
63    DType* dMz = thrust::raw_pointer_cast(dMz_dev.data());
64    DType* dNz = thrust::raw_pointer_cast(dNz_dev.data());
65    DType* dq = thrust::raw_pointer_cast(dq_dev.data());
66
67    int numBlocks; int numThreads;
68    getKernelLaunchParameter(Nx, solveLinAdjointEqKernel<DType>, &numBlocks, &numThreads);
69    solveLinStateEqKernel<DType><<<numBlocks, numThreads >>>(d,N,u,v,w,du,Nt,Nx,Nu,dMz,dq);
70    cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
71
72    setdNzKernel<DType> <<<numBlocks, numThreads >>>(dMz,Nt,Nx,Nu,dNz);
73    cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
74
75    thrust::fill(dMz_dev.begin(), dMz_dev.end(), 0); CUDA_CHECK_ERROR();
76
77    solveLinAdjointEqKernel<DType><<<numBlocks, numThreads >>>(d,P,u,v,w,du,dq,Nt,Nx,Nu,dMz);
78    cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
79
80    actionOfHessianKernel<DType> <<<numBlocks, numThreads >>>(d,dMz,dNz,P,N,Nt,Nx,Nu,Hdu);
81    cudaDeviceSynchronize(); CUDA_CHECK_ERROR();
82
83    //row sum of Hdu using thrust reduce by key
84    thrust::device_vector<DType> row_sums_dev(Nu);
85    thrust::reduce_by_key(
86      thrust::make_transform_iterator(thrust::make_counting_iterator(0),
87                                      linear_index_to_row_index<int>(Nx+1)),
88      thrust::make_transform_iterator(thrust::make_counting_iterator(0),
89                                      linear_index_to_row_index<int>(Nx+1))+((Nx+1)*Nu),
90      thrust::make_permutation_iterator(Hdu_dev.begin(),
91        thrust::make_transform_iterator(thrust::make_counting_iterator(0),
92                                        (_1%(Nx+1))*(Nu)+_1/(Nx+1))),
93      thrust::make_discard_iterator(),
94      row_sums_dev.begin(),
95      thrust::equal_to<int>(),
96      thrust::plus<DType>());
97
98    // transfer data to host
99    cudaMemcpy(mat_output, thrust::raw_pointer_cast(row_sums_dev.data()),
100              Nu*sizeof(DType), cudaMemcpyDeviceToHost);
101
102   return;
103 }
```

Listing B-2: `setdNzKernel` in file `applyHessGPU.cu`.

```
1  template<typename TType>
2  __global__ void setdNzKernel(TType* dMz, const int Nt, const int Nx, const int Nu, TType*
       dNz){
3
4    int z_ind = blockIdx.x*blockDim.x + threadIdx.x;
5  if (z_ind < Nx){
6    for (int u_ind = 0; u_ind < Nu; u_ind++){
7      dNz[LIN_IDX_3D(1,u_ind+1,z_ind+1,3,Nu)-1] =
8        0.5*(dMz[LIN_IDX_3D(1,z_ind+1,u_ind+1,3,Nx)-1]+
9             dMz[LIN_IDX_3D(1,z_ind+1,u_ind+2,3,Nx)-1]);
10     dNz[LIN_IDX_3D(2,u_ind+1,z_ind+1,3,Nu)-1] =
11       0.5*(dMz[LIN_IDX_3D(2,z_ind+1,u_ind+1,3,Nx)-1]+
12            dMz[LIN_IDX_3D(2,z_ind+1,u_ind+2,3,Nx)-1]);
13     dNz[LIN_IDX_3D(3,u_ind+1,z_ind+1,3,Nu)-1] =
14       0.5*(dMz[LIN_IDX_3D(3,z_ind+1,u_ind+1,3,Nx)-1]+
15            dMz[LIN_IDX_3D(3,z_ind+1,u_ind+2,3,Nx)-1]);
16     }
17   }
18 }
```

Listing B-3: `solveLinStateEqKernel` in file `applyHessGPU.cu`.

```
template<typename TType>
__global__ void solveLinStateEqKernel(dpara<TType> d, TType* N,
  TType* u, TType* v, TType* w, TType* du,
  const int Nt, const int Nx, const int Nu, TType* dMz, TType* dq){

  int z_ind = blockIdx.x*blockDim.x + threadIdx.x;

  if (z_ind < Nx){
    TType B1 = (*d.gamma)*(*d.B1c);
    TType B3 = (*d.gamma)*(*d.G3)*d.xdis[z_ind];

    TType dMz_new[3] = {};
    TType dMz_old[3] = {};

    TType bk[3] = {};
    TType Ak[9] = {};
    //solve linearized state equation
    for (int k = 1; k < Nt; k++){
      bk[0] = 0;
      bk[1] = B1*N[LIN_IDX_3D(3,z_ind+1,k,3,Nx)-1]*du[k-1];
      bk[2] = -B1*N[LIN_IDX_3D(2,z_ind+1,k,3,Nx)-1]*du[k-1];
      setAk<TType>(Ak, &d, &u[k-1], &v[k-1], &w[k-1], &B1, &B3);

      dMz_old[0] = dMz[LIN_IDX_3D(1,z_ind+1,k,3,Nx)-1];
      dMz_old[1] = dMz[LIN_IDX_3D(2,z_ind+1,k,3,Nx)-1];
      dMz_old[2] = dMz[LIN_IDX_3D(3,z_ind+1,k,3,Nx)-1];
      solveBloch<TType>(Ak, &d, dMz_new, dMz_old, bk);

      dMz[LIN_IDX_3D(1,z_ind+1,k+1,3,Nx)-1] = dMz_new[0];
      dMz[LIN_IDX_3D(2,z_ind+1,k+1,3,Nx)-1] = dMz_new[1];
      dMz[LIN_IDX_3D(3,z_ind+1,k+1,3,Nx)-1] = dMz_new[2];
    }
    dq[LIN_IDX_2D(1,z_ind+1,3)-1] = dMz_new[0];
    dq[LIN_IDX_2D(2,z_ind+1,3)-1] = dMz_new[1];
    dq[LIN_IDX_2D(3,z_ind+1,3)-1] = dMz_new[2];
  }
}
```

Listing B-4: `actionOfHessianKernel` in file `applyHessGPU.cu`.

```
template<typename TType>
__global__ void actionOfHessianKernel(dpara<TType> d, TType* dMz, TType* dNz, TType* P,
    TType* N,
  const int Nt, const int Nx, const int Nu, TType* output){

  int z_ind = blockIdx.x*blockDim.x + threadIdx.x;

  if (z_ind < Nx){
    TType B1 = (*d.gamma)*(*d.B1c);
    for (int ii = 0; ii < Nu; ii++){
      output[(ii+Nu) + Nu*z_ind] =
      B1*(*d.dx)*(dNz[LIN_IDX_3D(3,ii+1,z_ind+1,3,Nu)-1]*P[LIN_IDX_3D(2,z_ind+1,ii+1,3,Nx)
          -1] -
        dNz[LIN_IDX_3D(2,ii+1,z_ind+1,3,Nu)-1]*P[LIN_IDX_3D(3,z_ind+1,ii+1,3,Nx)-1] +
        N[LIN_IDX_3D(3,z_ind+1,ii+1,3,Nx)-1]*dMz[LIN_IDX_3D(2,z_ind+1,ii+1,3,Nx)-1] -
        N[LIN_IDX_3D(2,z_ind+1,ii+1,3 Nx)-1]*dMz[LIN_IDX_3D(3,z_ind+1,ii+1,3,Nx)-1]);
    }
  }
}
```

Listing B-5: `solveLinAdjointEqKernel` in file `applyHessGPU.cu`.

```
template<typename TType>
__global__ void solveLinAdjointEqKernel(dpara<TType> d, TType* P,
  TType* u, TType* v, TType* w, TType* du, TType* dq_in,
  const int Nt, const int Nx, const int Nu, TType* dPz){

  int z_ind = blockIdx.x*blockDim.x + threadIdx.x;

  if (z_ind < Nx){
    TType B1 = (*d.gamma)*(*d.B1c);
    TType B3 = (*d.gamma)*(*d.G3)*d.xdis[z_ind];
    TType Akp1[9] = {};
    TType Ak[9] = {};
    TType bkp1[3] = {};
    TType bk[3] = {};
    TType buffer_matrix[9];
    TType dPz_new[3] = {};
    TType dPz_old[3] = {};

    TType dq[3] = {};
    dq[0] = dq_in[LIN_IDX_2D(1,z_ind+1,3)-1];
    dq[1] = dq_in[LIN_IDX_2D(2,z_ind+1,3)-1];
    dq[2] = dq_in[LIN_IDX_2D(3,z_ind+1,3)-1];

    setAk<TType>(Ak, &d, &u[Nt-2], &u[Nt-2], &w[Nt-2], &B1, &B3);
    transposeMatrix3x3(Ak, Akp1);

    bkp1[0] = 0;
    bkp1[1] = -B1*P[LIN_IDX_3D(3,z_ind+1,Nt-1,3,Nx)-1]*du[Nt-2];
    bkp1[2] = B1*P[LIN_IDX_3D(2,z_ind+1,Nt-1,3,Nx)-1]*du[Nt-2];

    for (int ii = 0; ii < 3; ii++)
      dPz_old[ii] = dq[ii] + (*d.dt)*0.5*bkp1[ii];

    cnStep<TType>(Akp1, Ak, &d, 0);
    inverseMatrix3x3<TType>(Ak, buffer_matrix);
    matrixVectorMult3x3<TType>(buffer_matrix, dPz_old, dPz_new);

    dPz[LIN_IDX_3D(1,z_ind+1,Nt-1,3,Nx)-1] = dPz_new[0];
    dPz[LIN_IDX_3D(2,z_ind+1,Nt-1,3,Nx)-1] = dPz_new[1];
    dPz[LIN_IDX_3D(3,z_ind+1,Nt-1,3,Nx)-1] = dPz_new[2];

    for (int k = Nt - 3; k >= 0; k--){
      bk[0] = 0;
      bk[1] = -B1*P[LIN_IDX_3D(3,z_ind+1,k+1,3,Nx)-1]*du[k];
      bk[2] =  B1*P[LIN_IDX_3D(2,z_ind+1,k+1,3,Nx)-1]*du[k];

      setAkp1<TType>(Ak, &d, &u[k], &v[k], &w[k], &B1, &B3);

      dPz_old[0] = dPz[LIN_IDX_3D(1,z_ind+1,k+2,3,Nx)-1];
      dPz_old[1] = dPz[LIN_IDX_3D(2,z_ind+1,k+2,3,Nx)-1];
      dPz_old[2] = dPz[LIN_IDX_3D(3,z_ind+1,k+2,3,Nx)-1];

      solveAdjoint<TType>(Ak, Akp1, &d, dPz_new, dPz_old, bk, bkp1);

      dPz[LIN_IDX_3D(1,z_ind+1,k+1,3,Nx)-1] = dPz_new[0];
      dPz[LIN_IDX_3D(2,z_ind+1,k+1,3,Nx)-1] = dPz_new[1];
      dPz[LIN_IDX_3D(3,z_ind+1,k+1,3,Nx)-1] = dPz_new[2];
      #pragma unroll
      for (int ii = 0; ii < 9; ii++)
        Akp1[ii] = Ak[ii];
      #pragma unroll
      for (int ii = 0; ii < 3; ii++)
        bkp1[ii] = bk[ii];
    }
  }
}
```

B.2. cn_blochGPU

Listing B-6: Invokation of the cnBlochKernel in file cn_blochGPU.cu.

```
extern "C" void callCnBlochKernel(dpara<DType>* dpara_host,
    DType* M0_host, DType* u_host, DType* v_host, DType* w_host,
    int Nt, int Nx, DType* mat_output){

 //copy data to device
 thrust::device_vector<DType> M0_dev(M0_host,M0_host+(3*Nx));
 thrust::device_vector<DType> u_dev(u_host, u_host+Nt);
 thrust::device_vector<DType> v_dev(v_host, v_host+Nt);
 thrust::device_vector<DType> w_dev(w_host, w_host+Nt);

 // get raw pointer
 DType* M0 = thrust::raw_pointer_cast(M0_dev.data());
 DType* u = thrust::raw_pointer_cast(u_dev.data());
 DType* v = thrust::raw_pointer_cast(v_dev.data());
 DType* w = thrust::raw_pointer_cast(w_dev.data());
 // copy dpara to device
 thrust::device_vector<DType>xdis(dpara_host->xdis,dpara_host->xdis+Nx);
 thrust::device_vector<DType>gamma(dpara_host->gamma,dpara_host->gamma+1);
 thrust::device_vector<DType>T1(dpara_host->T1, dpara_host->T1 + 1);
 thrust::device_vector<DType>T2(dpara_host->T2, dpara_host->T2 + 1);
 thrust::device_vector<DType>M0c(dpara_host->M0c, dpara_host->M0c + 1);
 thrust::device_vector<DType>B1c(dpara_host->B1c, dpara_host->B1c + 1);
 thrust::device_vector<DType>G3(dpara_host->G3, dpara_host->G3 + 1);
 thrust::device_vector<DType>relax(dpara_host->relax, dpara_host->relax+1);
 thrust::device_vector<DType>dt(dpara_host->dt, dpara_host->dt + 1);

 //init dpara
 dpara<DType> d;
 // get raw device pointer for passsing to kernel
 d.xdis = thrust::raw_pointer_cast(xdis.data());
 d.gamma = thrust::raw_pointer_cast(gamma.data());
 d.T1 = thrust::raw_pointer_cast(T1.data());
 d.T2 = thrust::raw_pointer_cast(T2.data());
 d.M0c = thrust::raw_pointer_cast(M0c.data());
 d.B1c = thrust::raw_pointer_cast(B1c.data());
 d.G3 = thrust::raw_pointer_cast(G3.data());
 d.relax = thrust::raw_pointer_cast(relax.data());
 d.dt = thrust::raw_pointer_cast(dt.data());

 // init long. relaxation term b
 thrust::device_vector<DType> b_dev(3);
 b_dev[0] = 0;  b_dev[1] = 0; b_dev[2] = M0c[0] / T1[0] * relax[0];
 DType* b = thrust::raw_pointer_cast(b_dev.data());

 // init output M
 thrust::device_vector<DType> M_dev(3*Nx*Nt);
 DType* M = thrust::raw_pointer_cast(M_dev.data());

 int blocksPerGrid; int threadsPerBlock;
 getKernelLaunchParameter(Nx,cnBlochKernel<DType>,&blocksPerGrid,&threadsPerBlock);

 cnBlochKernel<DType><<<blocksPerGrid,threadsPerBlock>>>(d M0,u,v,w,b,Nt,Nx,M);
 CUDA_CHECK_ERROR();

 cudaMemcpy(mat_output,M, 3*Nx*Nt*sizeof(DType), cudaMemcpyDeviceToHost);
 return;
}
```

Listing B-7: `cnBlochKernel` in file `cn_blochGPU.cu`.

```
template<typename TType>
__global__ void cnBlochKernel(dpara<TType> d, TType* M0,
    TType* u, TType* v, TType* w, TType* b,
    const int Nt, const int Nx, TType* output){

  int z_ind = blockIdx.x*blockDim.x + threadIdx.x;

  if (z_ind < Nx){
    TType B1 = (*d.gamma)*(*d.B1c);
    TType B3 = 0;
    TType Ak[9] = {};
    TType Mz_new[3] = {};
    TType Mz_old[3] = {};
    B3 = (*d.gamma)*(*d.G3)*d.xdis[z_ind];
    // set  Mz(t=0) to M0
    #pragma unroll
    for (int ii = 1; ii <= 3; ii++)
      output[LIN_IDX_3D(ii, z_ind + 1, 1, 3, Nx) - 1] =
        M0[LIN_IDX_2D(ii, z_ind + 1, 3) - 1];

    for (int k = 1; k < Nt; k++){
      setAk<TType>(Ak,&d,&u[k-1],&v[k-1],&w[k-1],&B1,&B3);
      #pragma unroll
      for (int ii = 0; ii < 3; ii++)
        Mz_old[ii] = output[LIN_IDX_3D(ii+1,z_ind+1,k,3,Nx)-1];
      solveBloch<TType>(Ak, &d, Mz_new, Mz_old, b);
      #pragma unroll
      for (int ii = 0; ii < 3; ii++)
        output[LIN_IDX_3D(ii+1,z_ind+1,k+1,3,Nx)-1] = Mz_new[ii];
    }
  }
}
```

B.3. cn_adjointGPU

Listing B-8: Invokation of the `cnAdjointKernel` in file `cn_adjointGPU.cu`.

```
1  extern "C" void callCnAdjointKernel(dpara<DType>* dpara_host, DType* PT_host,
2    DType* u_host, DType* v_host, DType* w_host,
3    int Nt, int Nx , DType* mat_output){
4
5    //copy data to device
6    thrust::device_vector<DType> PT_dev(PT_host, PT_host + (3*Nx));
7    thrust::device_vector<DType> u_dev(u_host, u_host + (Nt-1));
8    thrust::device_vector<DType> v_dev(v_host, v_host + (Nt-1));
9    thrust::device_vector<DType> w_dev(w_host, w_host + (Nt-1));
10   // get raw pointer
11   DType* PT = thrust::raw_pointer_cast(PT_dev.data());
12   DType* u = thrust::raw_pointer_cast(u_dev.data());
13   DType* v = thrust::raw_pointer_cast(v_dev.data());
14   DType* w = thrust::raw_pointer_cast(w_dev.data());
15   // copy dpara to device
16   thrust::device_vector<DType> xdis(dpara_host->xdis, dpara_host->xdis + (Nx ));
17   thrust::device_vector<DType> gamma(dpara_host->gamma, dpara_host->gamma+1);
18   thrust::device_vector<DType> T1(dpara_host->T1, dpara_host->T1 + 1);
19   thrust::device_vector<DType> T2(dpara_host->T2, dpara_host->T2 + 1);
20   thrust::device_vector<DType> M0c(dpara_host->M0c, dpara_host->M0c + 1);
21   thrust::device_vector<DType> B1c(dpara_host->B1c, dpara_host->B1c + 1);
22   thrust::device_vector<DType> G3(dpara_host->G3, dpara_host->G3 + 1);
23   thrust::device_vector<DType> relax(dpara_host->relax, dpara_host->relax + 1);
24   thrust::device_vector<DType> dt(dpara_host->dt, dpara_host->dt + 1);
25   //init dpara
26   dpara<DType> d;
27   // get raw device pointer for passsing to kernel
28   d.xdis = thrust::raw_pointer_cast(xdis.data());
29   d.gamma = thrust::raw_pointer_cast(gamma.data());
30   d.T1 = thrust::raw_pointer_cast(T1.data());
31   d.T2 = thrust::raw_pointer_cast(T2.data());
32   d.M0c = thrust::raw_pointer_cast(M0c.data());
33   d.B1c = thrust::raw_pointer_cast(B1c.data());
34   d.G3 = thrust::raw_pointer_cast(G3.data());
35   d.relax = thrust::raw_pointer_cast(relax.data());
36   d.dt = thrust::raw_pointer_cast(dt.data());
37
38   // init output P
39   thrust::device_vector<DType> P_dev(3*Nx*(Nt-1));
40   DType* P = thrust::raw_pointer_cast(P_dev.data());
41
42   int blocksPerGrid; int threadsPerBlock;
43   getKernelLaunchParameter(Nx,cnAdjointKernel<DType>,&blocksPerGrid,&threadsPerBlock);
44
45   cnAdjointKernel<DType><<<blocksPerGrid,threadsPerBlock >>>(d,PT,u,v,w,Nt,Nx,P);
46   CUDA_CHECK_ERROR();
47
48   //Copy from GPU to mxArray
49   cudaMemcpy(mat_output, P, 3 * Nx*(Nt-1)*sizeof(DType), cudaMemcpyDeviceToHost);
50
51   return;
52 }
```

Listing B-9: `cnAdjointKernel` in file `cn_adjointGPU.cu`.

```
template<typename TType>
__global__ void cnAdjointKernel(dpara<TType> d, TType* PT, TType* u, TType* v, TType* w,
  const int Nt, const int Nx, TType* output){

  int z_ind = blockIdx.x*blockDim.x + threadIdx.x;

  if (z_ind < Nx){
    TType B1 = (*d.gamma)*(*d.B1c);
    TType B3 = (*d.gamma)*(*d.G3)*d.xdis[z_ind];
    TType Akp1[9] = {};
    TType Ak[9] = {};
    TType buffer_matrix[9];
    TType Pz_new[3] = {};
    TType Pz_old[3] = {};

    setAkp1<TType>(Akp1, &d, &u[Nt-2], &u[Nt-2], &w[Nt-2], &B1, &B3);
    cnStep<TType>(Akp1, Ak, &d, 0);
    inverseMatrix3x3<TType>(Ak, buffer_matrix);
    Pz_old[0] = PT[0+3*z_ind];  Pz_old[1] = PT[1+3*z_ind];  Pz_old[2] = PT[2+3*z_ind];
    matrixVectorMult3x3<TType>(buffer_matrix, Pz_old, Pz_new);

    #pragma unroll
    for (int ii = 0; ii < 3; ii++)
      output[LIN_IDX_3D(ii+1,z_ind+1,Nt-1,3,Nx)-1] = Pz_new[ii];

    for (int k = Nt - 3; k >= 0; k--){
      setAkp1<TType>(Ak, &d, &u[k], &v[k], &w[k], &B1, &B3);
      #pragma unroll
      for (int ii = 0; ii < 3; ii++)
        Pz_new[ii] = output[LIN_IDX_3D(ii+1,z_ind+1,k+2,3,Nx)-1];
      solveAdjoint<TType>(Ak, Akp1, &d, Pz_new, Pz_old);
      #pragma unroll
      for (int ii = 0; ii < 3; ii++)
        output[LIN_IDX_3D(ii+1,z_ind+1,k+1,3,Nx)-1] = Pz_new[ii];
      #pragma unroll
      for (int ii = 0; ii < 9; ii++)
        Akp1[ii] = Ak[ii];
    }
  }
}
```

List of Figures

2.1. The slice thickness Δz is proportional to the RF bandwidth Δf and inversely proportional to the slice selection gradient G_z. 7

2.2. Validity of the small tip angle approximation. Shows the transverse magnetization components M_y (blue solid) and M_x (green dashed) in response to a Hamming windowed SINC-pulse $B_1(t)$ at tip angles of $\theta = 30°$, $90°$ and $150°$. The Fourier approximation holds for $\theta = 30°$. At $\theta = 90°$ and $150°$ excitation errors are noticeable (red arrows). 8

2.3. Hard pulse with amplitude B_1 and pulse width τ. The flip angle is given by $\theta = \gamma B_1 \tau$ 9

2.4. SINC pulse with (blue solid) and without (green dashed) Hamming window. 10

2.5. Basic SLR workflow for optimized RF pulse design. A given RF pulse is mapped into the corresponding filter polynomials $A_N(z)$ and $B_N(z)$ via the forward SLR transform. The inverse SLR transform computes the RF pulse from the given polynomials. 11

2.6. Approximation of the target magnetization profile via FIR filter design with filter parameters: in-slice ripple δ_1, out-of-slice ripple δ_2, passband edge F_p and stopband edge F_s.[11] 12

2.7. Schematic description of a simultaneous two-slice CAIPIRINHA experiment without (a) and with (b) phase modulation. Odd k-space lines (black lines) are excited with a dual-band RF pulse with same phase (0,0) for both slices. Even k-space lines (grey lines) are the result of excitation with different phases (0,π). The individual slices are shifted with respect to each other in the superimposed image.[25] 14

2.8. Pulse sequence diagram for SMS single shot EPI. SMS excitation is achieved with a multiband pulse consisting of 3 frequency bands and therefore exciting 3 slices along the slice selection gradient axis. After the excitation a EPI readout follows (a). The use of sign and amplitude modulated slice select gradient blips in blipped-CAIPI is shown in (b).[21] 15

2.9. Basic overview of the slice-GRAPPA algorithm. The k-space data of the unaliased slices is estimated by applying GRAPPA-like kernel sets to the k-space of the collapsed slices. The kernels are fitted from a prescan calibration dataset acquired one slice at a time. Image taken from [27]. 16

2.10. CPU vs. GPU: The CPU is designed to minimize latency in a small number of heavy-weighted threads. Therefore, most of the chip area is dedicated to the control unit and cache memory. The GPU hides memory access latencies by focusing on computationally intensive tasks and much more chip area is dedicated to floating point calculations. [6] 25

2.11. A problem is decomposed into independent sub-problems that can be solved by a block of threads. All thread blocks are organized into a grid. In the case of (a) grid and block are two-dimensional. Each block is sheduled automatically to a SM for execution (b). This allows high scalability over a wide range of GPUs. [6] 27

2.12. A schematic illustration of the fork-join model. A master thread forks and joins sequent parallel regions with various threads. 31

3.1. Basic flowchart of the matrix-free trust-region Newton-CG optimization algorithm. 34
3.2. Results of the Matlab profiler for the computation of a single-slice OC pulse. The `applyHess.m`, `cn_adjoint.m` and `cn_bloch.m` functions are the major bottlenecks in the application (marked in red), and therefore making them target for performance optimizations. 36
3.3. Illustration of the reduction (row sum) of the `Hdu` matrix using the Thrust primitive `reduce_by_key` (a). Each element, located in the same row, is marked with its row index as key (red numbers), and `reduce_by_key` sums all elements with equal keys. Since the values of `Hdu` are stored in column-major order (MATLAB ordering) an implicit transposition is obtained with a `permutation_iterator`, treating them as they were stored in row-major order (b). After this implicit transposition each key is assigned to the correct element. 47

4.1. Boxplots of the optimization times for three different spatial discretization points N_x. For each N_x value a total number of 10 runs was performed. The individual optimization times are normalized to their corresponding median. 56
4.2. Speedup of the different implementations with regard to the MATLAB single-core method for varying spatial discretization points N_x (using the GTX 970 workstation). 57
4.3. Speedup of the CUDA implementation with regard to different methods for varying spatial discretization points N_x (using the GTX 970 workstation). 57
4.4. Dependency of the optimization time on the spatial grid size N_x for the sequential C/C++, OpenMP and CUDA implementation (a). Additionally, the N_x dependency of the CUDA implementation is depicted in (b). The N_x parameter is incremented up to the device memory limit of the GTX 970. A linear fit (least squares) is performed and the r^2 values are listed (rounded to four digits of precision). 58

4.5. Dependency of the optimization time on the temporal grid size N_t for the sequential C/C++, OpenMP and CUDA MEX files (a). Additionally, the N_t dependency of the CUDA implementation is depicted in (b). The N_t parameter is incremented up to the device memory limit of the GTX 970. A linear fit (least squares) is performed and the r^2 values are listed (rounded to four digits of precision). The observations corresponding to grid sizes $4 \cdot N_t$ and $11 \cdot N_t$ were removed from the data set due to rejection of the last two Newton steps. 59

4.6. Impact of the single-precision and double-precision floating point format on the achievable speedup in optimization time. A comparision of the different implementations with regard to the MATLAB single-core method is illustrated in (a) for the GTX 970 workstation and in (c) for the Tesla system. The speedup offered by the CUDA implementation is shown in (b) for the GTX 970 workstation and in (d) for the Tesla system. The grid sizes are fixed to $N_x = 5001$ and $N_t = 697$. 61

4.7. Difference plot between the OC pulse obtained from the MATLAB $\mathbf{u_{MAT}}$ and GPU $\mathbf{u_{GPU}}$ implementation for single- and double-precision using the GTX 970 workstation. The grid sizes N_t and N_x are set to 697 and 5001, respectively. 64

4.8. Reconstructed excitation (magnitude) obtained with a SMS pulse using (a) double-precision and (b) single-precision optimization. Additionally, the difference between (a) and (c) is shown in (e). The corresponding slice profile evaluations along the red lines are depicted in the right column. . . 65

4.9. Results of the Matlab profiler for the computation of a single-slice OC pulse. The former major bottlenecks (marked in red) are implemented as CUDA MEX files `applyHessGPU.m`, `cn_adjointGPU.m` and `cn_blochGPU.m`. 67

4.10. Execution time of the `applyHess` CUDA kernel for different grid sizes N_x. . 68

4.11. Nvidia Visual Profiler result of the `applyHessGPU` MEX file (a) for a spatial grid size N_x of 10001. The light brown blocks show the time spent on memory transfer operations from the host to the device (HtoD) and from the device to the Host (DtoH). The green blocks represent the time spent on computation. A zoomed version of the compute block is illustrated in (b). The `applyHessGPU` kernel is split into consecutive sub-kernels. 69

List of Tables

3.1. List of files in the RFcontrol project. 38

4.1. Optimization times for different impelementation methods and varied N_x. . 56

4.2. Root-mean-square error for single-precision data $RMSE_s$ and for double-precision data $RMSE_d$ for different multiples of the temporal grid size $N_t = 697$. The RMSE is obtained between the desired and simulated magnetization profile. The grid size N_x is set to 5001. 62

4.3. Root-mean square error for the GPU implementation $RMSE_{GPU}$ and MATLAB implementation $RMSE_{MAT}$ for single- and double-precision. The RMSE is obtained between the desired and simulated magnetization profile. The grid sizes N_t and N_x are set to 697 and 5001, respectively. 63

4.4. Mean absolute error MAE_u and difference in J between the GPU pulse and MATLAB pulse. The grid sizes N_t and N_x are set to 697 and 5001, respectively. 64

A-1. Summary of GPU specifications. 84