A Highly Parallelized Bloch Simulator for Magnetic Resonance Imaging

Michael Honke
Undergraduate Honours Thesis

Supervisor
Dr. Christopher Bidinosti

Committee Members
Dr. Murray Alexander
Dr. Christopher Henry

April 17, 2017

Abstract

A full low field, diffusion capable, MRI simulator is designed and implemented. The simulator starts with an input sample and outputs a simulated MR image. The simulator supports any general MRI sequence followed by a full image resconstruction process. Gradient recalled echo (GRE) and pulse gradient spin echo (PGSE) sequences were implemented with sample results. The PGSE sequence had diffusion sensitizing gradients added and demonstrated on samples with non zero diffusion coefficients. The simulator was also shown to faithfully recreate real MRI artifacts. The entire process is programmed using a computer's graphics processing unit (GPU) to keep run times short. Compared to the CPU the GPU was able to obtain simulation times shorter by up to a factor of fifty. GPU specific and general optimizations made to the simulator algorithms are discussed throughout this thesis. Having demonstrated the successful generation of simulated images the presented simulator is now ready to be used for low field MRI research.

Acknowledgements

I want to express my gratitude to my supervisor Dr. Chris Bidinosti for his support and guidance of this interdisciplinary project. I am thankful to my Honours Committee members Dr. Chris Henry and Dr. Murray Alexander who have brought together their unique backgrounds to guide the development of this project.

Contents

List of Figures

1 Introduction

Simulators of magnetic resonance imaging (MRI) serve as both teaching and research tools. Different simulators are required for each area of MRI research. Diffusion MRI (DMRI) is specific to generating images that indicate the level of diffusion present in a sample. Therefore to simulate DMRI requires modelling basic MRI and the diffusive motion of particles. MRI can be split into two separate regimes. Conventional and high field MRI uses magnetic fields between 0.1-10T [1]. However, MRI can use much lower fields, on the order of a milli-Tesla and is called low field MRI.

DMRI simulators that model MRI at conventional field strengths have received significant development. The Diffusion Microscopist Simulator (DMS) models a full MRI pipeline from sample generation to image export [2]. Some simulators have been written to specifically run on computer graphics processing units (GPUs) rather than central processing units (CPUs), reducing simulation times. The Waudby Christopher Diffusion Monte Carlo (WCDMC) simulator was shown to run 300 times faster on a GPU [3]. However when compared to the DMS, the WCDMC simulator did not generate a final image.

A new DMRI simulator, improving on the WCDMC simulator, was written by Trevor Vincent in 2013 [4]. This simulator used new features introduced for GPU programming to further reduce simulation times. It was also no longer restricted to the high field MRI regime. By using a Bloch equation solver on the GPU the new simulator became capable to model low field MRI. The low field regime requires avoiding time saving approximations used in the high field regime, the use of a GPU allowed for the simulator to run in reasonable times. This simulator did not model the full MRI process, producing no simulated image.

A complete low field diffusion MRI simulator was written for this thesis. This simulator uses the diffusion simulation done by Trevor Vincent placed within a full MRI pipeline. Like the DMS simulator, this simulator starts with a user defined sample and exports a final image. The simulator continues to use a GPU as the primary processing resource. With the additional computational burden of a full MRI pipeline the GPU allows the simulator to remain a tool that can be run on personal computers.

This simulator is intended to be used in the research of low field MRI. The ability to model diffusion allows for the effect of diffusion on low field MR images to be quantified. The simulator is designed to remain general, supporting multiple types of low field MRI. Simulating the full MRI pipeline allows for an early indicator of the results from experiments. Since running the simulation is much cheaper and time effective than a real scan, experiments can be optimized using this simulator before running on real hardware. Additionally, the simulator can be used to troubleshoot real MRI experiments by simulating physically unrealizable scenarios. Troubleshooting in a simulated environment is also assisted with data not possible to obtain in a lab environment.

The thesis starts by describing the main elements of a MRI scan. Each of these must be adequately modelled if the simulator is to produce a realistic final image. Apart from the physical theory, the computational theory required for running the simulator is documented. The algorithms developed are explained alongside with optimizations explored throughout the simulator development. To show the proper functioning of the simulator a series of standard MRI tests are performed. Lastly, the performance of the simulator is discussed.

2 Physical Theory

The simulator aims to model as many aspects of a real MRI scanner as possible. To model each scan step it is necessary to understand the physical processes for that step. The basic spin physics that provides the operational basis for MRI will be presented. Then the process of generating an MR image will be explained.

2.1 Nuclear Magnetization

Quantum mechanically particles have quantitized levels of spin. Particles with half integer spin are called fermions. Protons and electrons are examples of fermions with spin-1/2. The spin-1/2 property results in these particles having a magnetic moment $\vec{\mu_S}$, where $\vec{\mu_S} = \gamma\vec{S}$. Here γ is the gyromagnetic ratio, and $\vec{S}$ is the spin operator. If multiple particles are present then a net magnetization vector $\vec{M}$ can be defined as $\vec{M} = \sum_i \vec{\mu_S}_i$. When $\vec{M}$ is placed in a magnetic field $\vec{B}$ a torque is produced according to Equation 1 [5].

$$\frac{d\vec{M}}{dt} = \gamma[\vec{M} \times \vec{B}] \tag{1}$$

A resonance frequency ω_0 is produced by the torque defined by

$$\omega_0 = \gamma B \tag{2}$$

and is known as Larmor precession. Energetically it is favourable for $\vec{\mu_S}$ and hence $\vec{M}$ to align with $\vec{B}$. This introduces a longitudinal relaxation time T_1 which affects M_z according to

$$\frac{dM_z(t)}{dt} = \gamma\left(M_x(t)B_y(t) - M_y(t)B_x(t)\right) - \frac{M_z(t) - M_0}{T_1} \tag{3}$$

where M_0 is the stationary value for $|\vec{M}|$. M_x and M_y are also modified in Equation 4 by a time constant T_2 known as the transverse relaxation time. This occurs due to internuclear interactions between the particles.

$$\begin{aligned} \frac{dM_x(t)}{dt} &= \gamma\left(M_y(t)B_z(t) - M_z(t)B_y(t)\right) - \frac{M_x(t)}{T_2} \\ \frac{dM_y(t)}{dt} &= \gamma\left(M_z(t)B_x(t) - M_x(t)B_z(t)\right) - \frac{M_y(t)}{T_2} \end{aligned} \tag{4}$$

Equations 3 and 4 are known as the Bloch equations, and were derived by Bloch in 1946 [5]. These allow for $\vec{M(t)}$ to be solved for any $\vec{B(t)}$ either symbolically or numerically.

2.2 Pulse Sequences

2.2.1 Pulses

MRI uses magnetic excitation pulses to tip $\vec{M}$ against $\vec{B}$ to allow for Larmor precession. The precession of $\vec{M}$ can be used to induce current into a properly oriented coil according to Faraday's law. If a static magnetic field B_0, defining the z-axis, is on for a time $t >> T_1$ it would be expected that $\vec{M} = M_0\hat{z}$. The simplest method to tip $\vec{M}$ against B_0 is to apply a hard rectangular pulse B_1 along the x-axis in the rotating frame. The rotating frame is chosen to be a cartesian coordinate system that rotates at the frequency ω_0. According to Equation 1 the resulting torque is along the y-axis, which starts tipping $\vec{M}$. As $\vec{M}$ forms a non-zero angle α with $\vec{B_0}$ Larmor precession in the laboratory's frame begins. Since $\vec{M}$ is precessing it is necessary to match the frequency of the pulse to ω_0 to keep $\vec{B_1}$ stationary in the rotating frame. Therefore, from the rotating frame $\vec{B_1} = [B_1, 0, 0]$ and in the lab frame $\vec{B_1} = B_1[cos(\omega_0 t), -sin(\omega_0 t), 0]$. The resulting angle α is given by

$$\alpha = \gamma B_1 T \tag{5}$$

where T is the length of the pulse. This equation is only valid for small tip angles or on-resonance pulses [6].

The described excitation pulse along the rotating x-axis results in $\vec{M} = M_0 sin(\alpha)\hat{y} + M_0 cos(\alpha)\hat{z}$ [6]. An ideal excitation pulse will result in $\alpha = \pi/2$ as this maximizes the signal produced in a coil with its normal laying in the x-y plane. Another common angle $\alpha = \pi$ results in a complete inversion of $\vec{M}$. This type of pulse is called an inversion pulse.

2.2.2 Sampling Signal

Having created transverse magnetization using an excitation pulse, the next phase of a MRI scan is to sample the signal. This requires introducing the Nyquist criterion. The sampling rate must be at least twice as high as the highest frequency in the signal to prevent aliasing.

The sampling interval Δt is thus required to be,

$$\Delta t = \frac{1}{2\Delta v} \tag{6}$$

where Δv is the half-bandwidth of the signal.

The signal to be read is that which is induced into the receive coil of the scanner by the precessing magnetization. The signal $S(t)$ is given by

$$S(t) = \omega_0 \int d^3 r e^{-t/T_2(\vec{r})} M_\perp(\vec{r}, 0) B_\perp(\vec{r}) sin(\omega_0 t + \theta_B(\vec{r}) - \phi_0(\vec{r})) \tag{7}$$

where ω_0 is the Larmor precession frequency, $M_\perp$ is the transverse magnetization, $B_\perp$ is the transverse component of the receive coil field, θ_B is the phase of the receive coil field in the frame of the laboratory and ϕ_0 is the initial phase of $M_\perp$ after excitation [7].

An important characteristic of signals is the signal to noise ratio (SNR). This ratio determines how much signal is being produced by a scan compared to random background noise. The SNR is described quantitatively by

$$SNR \propto \frac{FOV_{FE} \cdot FOV_{PE} \cdot \Delta z \cdot F_{sequence} \cdot \sqrt{NSA}}{\sqrt{BW \cdot N_{FE} \cdot N_{PE}}} \tag{8}$$

where FOV_{FE} is the field of view (FOV) in the frequency encoding direction, FOV_{PE} is the FOV in the phase encoding direction, Δz is the slice thickness being imaged, $F_{sequence}$ are sequence depedent factors, NSA is the number of signal averages, BW is the scan bandwidth, N_{FE} is the resolution in the frequency encoding direction, and N_{PE} is the resolution in the phase encoding direction [8]. The concepts of frequency and phase encoding are introduced in later sections.

2.2.3 K-Space

K-space is the MRI specific term for the Fourier transform of an image. The purpose of a pulse sequence is to use a series of pulses and gradients to guide the signal acquisition process filling k-space. As a result k-space contains the time domain signal from the sample spins. This time domain signal is the Fourier transform of the sample's transverse magnetization. To reconstruct the image an inverse Fourier transform is applied creating an image revealing the sample's magnetization.

K-space has its own coordinate system denoted by $\vec{k}$. K-space is navigated by using the relation,

$$\vec{k}(t) = \frac{\vec{\nabla}\phi(\vec{r}, t)}{2\pi} \tag{9}$$

where $\phi(\vec{r}, t)$ is the accumulated phase of the spins in the sample [6]. K-space can be filled using techniques such as cartesian raster, radial projection or spiral patterns. Cartesian raster is commonly used, filling one line of k-space per excitation.

2.2.4 Gradients

Allowing the main field of the MRI scanner to be along the z-axis, gradients are magnetic fields along the z-axis that augment that main field. The gradient vector $\vec{G}$ is given by

$$\vec{G} = \frac{\partial B_z}{\partial x}\hat{x} + \frac{\partial B_z}{\partial y}\hat{y} + \frac{\partial B_z}{\partial z}\hat{z} \tag{10}$$

where $\hat{x}, \hat{y}, \hat{z}$ are along each Cartesian axis [6]. An MRI scanner has separate coils for each direction of gradient. The length and amplitude of a gradient is specified by the pulse sequence guiding the population of k-space. As shown in Equation 9 the phase of the signal sets the k-space vector. The phase is in turn set by the gradients, where

$$\phi(\vec{r}, t) = \gamma \int_0^t \vec{r} \cdot \vec{G}(t')dt' \tag{11}$$

for a gradient that lasts t amount of time.

When the main scanner field is very strong, G_x, G_y or G_z gradients can be treated as existing solely on their own. This simplifies the navigation of k-space as $\vec{G}$ has only one component in Equation 11. However, due to Maxwell's equations it is not possible to produce a unidirectional gradient as $\vec{\nabla} \times \vec{B} = 0$. Under this requirement, to produce G_x an accompanying concomitant gradient with a field that points along the x-axis is required. The total magnetic field is then, $\vec{B}(x, z) = xG\hat{z} + zG\hat{x}$ [9]. If the main field is much greater than G the $\hat{x}$ term can be neglected. However, when the main field is comparable the concomitant term becomes significant.

Therefore, for a general description of MRI gradients it is necessary to include all the concomitant terms. As shown, $\vec{B}$ varies in more directions than just along the z-axis. The product $\vec{G} \cdot \vec{r}$ as in equaton 11 produces only B_z. Each term of $\vec{B}$ when produced by gradients features a particular gradient strength G for that direction, and a dependence on position r, where r is x, y, or z in Cartesian coordinates. Mathematically nine terms are required forming a tensor $\boldsymbol{G}$,

$$\boldsymbol{G} = \begin{bmatrix} G_{xx} & G_{xy} & G_{xz} \\ G_{yx} & G_{yy} & G_{yz} \\ G_{zx} & G_{zy} & G_{zz} \end{bmatrix}$$

where each element $G_{ij} = \dfrac{\partial B_i}{\partial j}\hat{i}$. The letter i denotes the direction of $\vec{B}$ the gradient term refers to, and j denotes the spatial dependence of $\vec{B}$. This tensor notation provides a convenient method of tracking gradients when concomitant terms must be kept.

The actual shape of a gradient can vary greatly. Time domain trapezoidal shaped pulses are one option. These consist of a ramp up, a plateau and a ramp down. The total area of the trapezoid is determined by the pulse sequence.

2.2.5 Frequency Encoding

Frequency encoding is one of the methods used in MRI to spatially encode spins as a function of position. A gradient is used to vary the Larmor precession frequency along its applied direction. The signal induced in the MRI's coil is now composed of a range of frequencies. An inverse Fourier transform can be done to map the quantity of signal in each frequency onto a shared axis of frequency and location. The resulting spectrum is a projection of the sample [6].

Frequency encoding gradients are used to navigate one direction of k-space. A gradient applied along the x-axis navigates k-space according to,

$$k_x = \frac{\gamma}{2\pi} \int_0^T G_x(t)dt \tag{12}$$

for a gradient of time T. The k-space coordinate is offset in the same direction as gradient polarity. The methods sequences use to achieve frequency encoding vary. Specific examples of frequency encoding are discussed later.

2.2.6 Phase Encoding

Phase encoding encodes spins by creating a linear variation in phase along a particular direction. Often phase encoding encodes information orthagonally to the direction of frequency encoding. Since a gradient was applied along the x-axis in the frequency encoding example, lets use a gradient along the y-axis for phase encoding. The k-space coordinate k_y is traversed according to

$$k_y = \frac{\gamma}{2\pi} \int_0^T G_y(t)dt \tag{13}$$

where T is the length of the phase encoding gradient [6]. Application of a zero gradient positions the k-space coordinate along the x-axis. Like frequency encoding, a positive or negative phase encoding gradient will move the k-space coordinate in the same direction. The phase encoding gradient often occurs before the frequency encoding gradient. However, the phase encoding gradient can be combined with other gradients such as the frequency pre-phasing gradient to minimize scan time. Using Cartesian raster k-space filling, the line to be filled is set by the phase encoding gradient. The k-space coordinate is translated along this line by the frequency encoding gradient. Therefore one line of k-space is filled per phase encoding step. The sequence repeats the frequency encoding steps, and any other preparatory steps for each phase encoding step. This repeat time is denoted by T_R. Therefore, if k-space contains N lines then N phase encoding steps are required. The required time for a complete scan can then be found as

$$T_{scan} = T_R \cdot N \cdot N_{avg} \tag{14}$$

where N_{avg} is the number of signal averages [6].

2.3 Gradient Recalled Echo (GRE) Sequence

A Gradient Recalled Echo (GRE) sequence combines all the previously mentioned sequence structures. Each part of the sequence is shown in Figure 1. An excitation pulse is initially

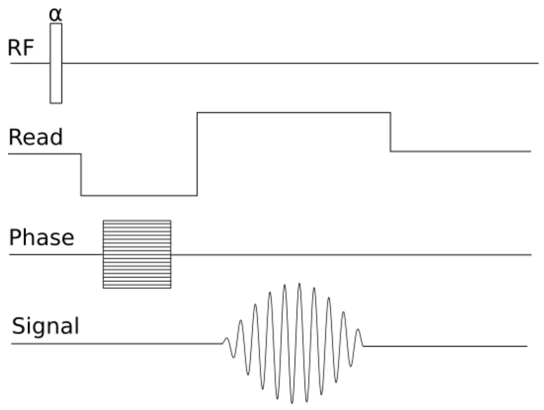


Figure 1: Pulse diagram of a typical GRE sequence. The α RF pulse is the initial excitation pulse. The entire sequence is repeated for each line of the phase encoding gradient.

applied. A $\pi/2$ pulse produces the maximum signal but any flip angle α can be used. Frequency encoding is done using gradient reversal. An initial pre-phasing pulse is applied, followed by a read-out pulse of opposite polarity [10]. This achieves the required k-space navigation as described in Section 2.2.5 through dephasing and rephasing spins. The initial negative pre-phasing gradient causes phase decoherence between spins along the gradient direction, moving negatively in k-space. The following positive read-out gradient rephases the spins, generating an echo of previously attenuated signal. This signal is collected during the read-out gradient pulse as k-space is traversed from left to right, eventually dephasing the spins again. Typically at the center of the echo T_E, $k_x = 0$. If filling k-space symmetrically the area of the pre-phasing gradient is half the area of the read-out gradient [6].

Phase encoding is done as described in Section 2.2.6. One strength of phase encoding gradient is used for each line of k-space. The phase encoding gradient in GRE is often combined wtih the frequency encoding pre-phasing gradient. Since frequency and phase encoding gradients are applied orthogonally k-space is traversed in two orthogonal directions.

2.4 Pulse Gradient Spin Echo (PGSE) Sequence

The Pulse Gradient Spin Echo (PGSE) sequence is presented as it provides a different method of frequency encoding. Also in the low field limit, when concomitant gradients become significant, the PGSE sequence has been shown to perform better than GRE [11]. The only difference in pulse diagram Figures 1 and 2 are the read and RF lines. While α need not be

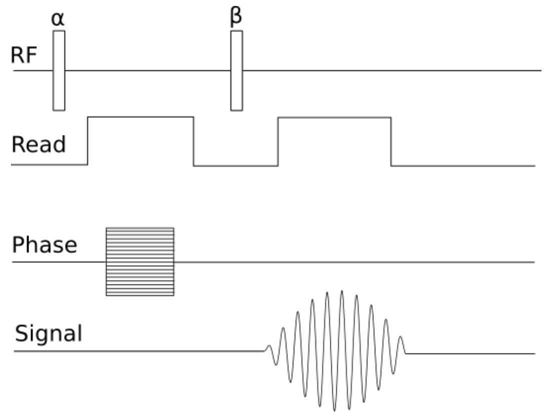


Figure 2: Pulse diagram of a typical PGSE sequence. The α RF pulse is the excitation pulse followed by a β inversion pulse. The entire sequence is repeated for each line of the phase encoding gradient.

$\pi/2$, the angle of the inversion pulse β should ideally be π. The first read gradient navigates k-space to the right. The inversion pulse flips $\vec{k}$ about the center of k-space positioning it on the leftwards most edge. The position of $\vec{k}$ is now identical as for GRE immediately before the second read gradient. The signal readout is done moving from left to right in k-space as before. The phase encoding strategy is the same as for GRE, one repeat of the sequence per phase encoding gradient line.

2.5 Diffusion

Even in thermal equilibrium the molecules composing a fluid are moving. As they move there is chance that two molecules may collide, altering their post-collision paths according to momentum conservation. For a large timespan containing many collisions the movement appears random. This phenomena is called Brownian motion after Robert Brown who made the initial observations in 1828. The mean displacement λ_x of a Brownian diffusing particle is

$$\lambda_x = \sqrt{2Dt} \tag{15}$$

where D is the coefficient of diffusion and t is the amount of time the particle has been travelling [12]. For general three dimensional motion Equation 15 becomes $\lambda = \sqrt{6Dt}$. Mathematically Brownian motion can be modelled as a series of random displacements of λ length. However, since the process is random the total mean sum of displacements must equal zero [13]. Therefore the macroscopic diffusion is zero.

A pulse sequence can be sensitized to the microscopic diffusion present in a sample. This requires linking the signal intensity to the the sample's diffusion coefficient as presented in Equation 15. Gradients have been used to achieve frequency and phase encoding, but can also be used to create sensitivity to diffusion. For GRE sequences two equal but opposite gradient pulses are added to the pre-existing sequence. If a spin remains fixed in position throughout the scan the total accumulated phase due to the two diffusion weighting gradients is zero. However, if the spin diffuses to a new location after the first diffusion weighting gradient is applied, a net phase is imparted. A phase difference results between the aforementioned spin, and neighbouring particles that have not moved during the scan. The signal experiences destructive interference and hence attenuation. The result is that sample regions with high diffusion produce less signal [14]. The signal S produced with diffusion is given by

$$S = S_0 e^{-bD} \tag{16}$$

where S_0 is the signal without diffusion, and b is the b-value which controls the amount of diffusion weighting imparted by the specific sequence.

This simplest type of MR image for diffusion imaging is called a diffusion weighted image. The intensity of each voxel is weighted expoentially by the diffusion coefficient [6]. This diffusion coefficient is no longer for a specific spin but is an average of all spins in that voxel. The ability to produce a diffusion weighted image allows for more quantitative images to be created. Two diffusion weighted images can be used to solve for the apparent diffusion coefficient (ADC) for each image voxel.

3 Computational Theory

The simulator presented in this thesis was designed for a computer's graphics processing unit (GPU). Programming for a computer's central processing unit (CPU) is commonplace and would not warrant a significant discussion on the hardware and software used. However many of the programming design choices were heavily influenced due to the use of the GPU. These can be better understood in the context of the hardware and software used by a GPU.

3.1 GPU Hardware

The GPU approaches processing large workloads very differently than a CPU. CPUs tend to have a few cores, where each core is one processing unit, that runs at high clock frequencies. The clock frequency determines how quickly the core can process program instructions.

Over the development of processors the clock frequency has steadily been increased with improvements in manufacturering technology and chip design. However in the last few years the advancement of CPU clock frequency has slowed reaching a clockspeed limit of about 4 GHz [15]. This limits the maximal amount of FLoating point Operators Per Second (FLOPS). The true computational power of a device is represented by FLOPS.

GPUs use hundreds of cores clocked at under half the speed of a CPU core to execute instructions. This design choice was originally done due to the original purpose of GPUs. GPUs have been traditionally used to generate the images displayed on computer monitors. In this application it was sensible to split the computational burden of an entire image over many subprocessors.

The GPU consists of three types of memory (global, constant, shared), streaming multiprocessors (SMs), and streaming processors (SPs). The execution unit of the GPU is composed of multiple SMs. Each SM in turn has multiple SPs. A GTX1080, a GPU released by NVIDIA in 2016, has 20 SM's each with 128 SPs totally 2560 SPs for the device. Each core has a clock frequency of 1607 MHz [16]. Executions are carried out on the individual SP. The memory is shared between the execution units at different levels. Each SM has a section of memory called a register file. This is used for storing registers (variables) from each SP on the SM. The register file is the fastest level of memory available [15]. However it is the smallest in size at 256 KB on a GTX1080 [16]. The next level of memory is called shared memory. This is accessible to the entire SM, and can be used as cache to reduce access to the slower global memory on a GPU. The GTX1080 has 96KB of shared memory per SM [16]. Unlike cache on a CPU, the shared memory on a GPU is operated by the user program. Global memory is the largest and slowest available memory for a GPU. A GTX1080 has 8GB of global memory [16]. Constant and texture memory are both views into global memory, optimized for their particular application. Texture memory features hardware based interpretation, and constant memory is for read-only data [15]. User programs must consider the GPU hardware format to reach their peak performance.

3.2 GPU Software

To program the GPU hardware various application programming interfaces (APIs) exist. The one used for NVIDIA GPUs is the Compute Unified Device Architecture (CUDA). The hardware structure for CUDA was described in Section 3.1. The CUDA API abstracts the user from the hardware enabling easier programming. The API is for the C/C++ language.

All execution in CUDA is done with threads. Threads exist for CPUs, where each thread contains the execution of a sequential program. Each thread encapsulates the memory and code execution for a program. To the user each thread appears to be executed sequentially. This philosophy is shared by CUDA, but CUDA launches many threads at once to process data [17]. This is the basis of parallel programming in CUDA.

When a CUDA application is launched CUDA creates a grid of threads. This grid breaks the threads into groups of blocks. Each block can have up to 1024 threads [17]. The dimensionality of a block can be one, two, or three dimensions. This allows for the thread structure to mirror a program's data structure. Each block has access to one SM worth of shared memory.

Each SM can process 32 threads, called a warp, at a time. Warps are scheduled to be run on the SM when it is available. Warps are executed according to the Single Instruction Multiple Data (SIMD) model of parallel execution [17]. This means the numerical values of variables (the data) can vary, but the logical actions done to the data should remain identical. For example, the SIMD model is followed if each thread adds two possibly different numbers together. The current warp remains active only while it is being executed. If some or all of the threads are blocking, waiting for external data, the SM switches active warps. This helps to ensure that the GPU remains occupied [17]. For maximal performance all threads in a warp should follow the SIMD model, however CUDA can handle cases when individual threads in a warp require different logical instructions. This scenario is called thread divergence and should be avoided when programming with CUDA. The threads that don't require execution for the particular set of operations are deactivated. Therefore CUDA must process the same warp multiple times to cover all divergent paths resulting in a performance decrease.

A program written with CUDA has two sections. The first section is code for the CPU. The CPU is referred to as the host. The second section is code specifically for the GPU, which is referred to as the device. Host specific functions are prefixed with a `__host__` tag while device specific functions have a `__device__` tag. Functions executable by either device or host have both tags. Part of the device specific code are functions called kernels. These are prefixed with a `__global__` tag. A kernel is called using the name of the kernel followed by a set arguments inbetween `<<<` and `>>>`. These arguments define how the kernel runs on the device. The first argument defines the number of blocks the kernel uses and the second argument the number of threads in each block [15].

3.3 Numerical Integration

Previous work on the simulator focused on developing the diffusion algorithms[4]. This version focuses on simulating the magnetization of the spins under full MR imaging sequences. A Runge-Kutta 4th order solver has been used to solve the Bloch equations listed in Section 2.1.

The Euler method is a simple method of numerically solving a differential equation. It estimates the solution by using its first derivative, which is known. Let the differential equation be $f(x_n, y_n)$ where y_n varies with x_n. Here the subscript n keeps track of the current step of the integrator, where $x_{n+1} = x_n + h$. The variable h is the step size of x_n. Therefore

$$y_{n+1} = y_n + hf(x_n, y_n) \tag{17}$$

is the defining equation for the Euler method [18].

The Runge-Kutte 4th order integrator improves upon the Euler method by sampling the step interval at multiple points. There are four points k_1, k_2, k_3 and k_4 given by

$$k_1 = hf(x_n, y_n) \tag{18}$$

$$k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \tag{19}$$

$$k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \tag{20}$$

$$k_4 = hf(x_n + h, y_n + k_3) \tag{21}$$

$$y_{n+1} = y_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(h^5) \tag{22}$$

where $O(h^5)$ is a fifth order error term in the step-size [18].

4 Method

Here the implementation of the simulator is described. Programming the simulator presented two key challenges. First and foremost the simulator must be physically accurate above all else. However, optimizations were made as long as they did not compromise the numerical accuracy of the simulator.

The structure of the simulator was designed to mimic a real MRI scanner. This provides a convenient structure for encapsulating class responsibilities. Additionally, it clarifies the physical processes each class is modelling. Figure 3 diagrams the simulator modules and the relations between them. To create a scan the user instantiates an instance of each of the simulator classes. The main class is the scanner class. This class accepts references to all other classes and manages communication between them and the simulator backend. The sequence superclass defines the general characteristics of a simulator sequence. Sequence subclasses define specific implementations such as for the GRE and PGSE sequences discussed in Sections 2.3 and 2.4 respectively. The primitives class defines the basic functions required of the sample to be scanned. An included sample is a finite length cylinder that sits in the X-Y plane. The coil class represents the behaviour of a coil in a real scanner. Specific

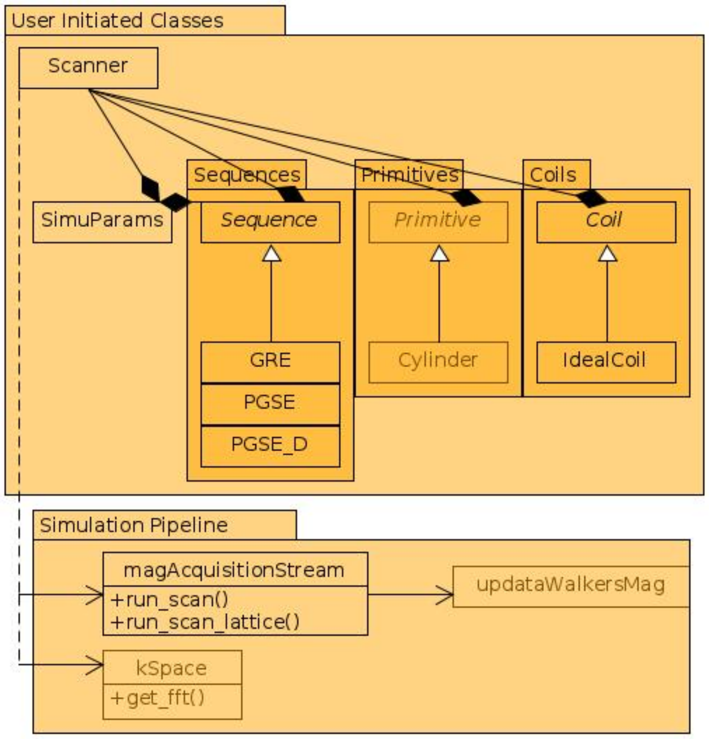


Figure 3: A high level structure of the simulator. The user initializes copies of the SimuParams, Sequence, Primitive and Coil classes. These classes are then passed to the Simulation Pipeline to direct the scan.

implementations can be made to alter the RF and gradient fields produced simulating non-ideal output from a real coil. Lastly the simulator parameter (SimuParams) class stores all simulation parameters, allowing for any class to access parameters it may need.

4.1 Scanner

The scanner class is responsible for collecting the parts of the virtual scanner, and then running the scan once all requirements are satisfied. Instantiating a scanner object requires providing the class constructor with a sequence, a coil, and a list of parameters for the scan. The class also features an `add_primitive()` method which allows the user to add a primitive. This simulates loading a scanner with a sample. Afterwhich the user calls the `scan()` or `scanCPU()` method to launch a scan on either the device or the host respectively. The scan consists of the following steps

1. Create a stream for the scan using `cudaStreamCreate()`.
2. Create a scan, which includes adding a magAcquisition object to save the scan results, such as the signal.
3. Run the scan by calling `runScan()`.
4. Call `cudaDeviceSynchronize()` to ensure all threads are done running on the device before continuing.
5. Save the scan results, this includes performing a device wide summation further discussed in Section 4.4. The results are copied from the device to the host memory.
6. Fill the collected signal into k-space.
7. Perform an inverse fast fourier transform to generate the final image.

afterwhich the image, and any other requested data is saved to the disk.

4.2 Sequence

The GRE and PGSE sequences as dicussed in Sections 2.3 and 2.4 were programmed for the simulator. The sequence is responsible for declaring sequence pulses, and for defining how k-space is to be filled. Code Listing 1 shows how pulses are defined for the GRE sequence.

Listing 1: Pulses defined for the GRE sequence. The pulses are included from a pulse library for the simulator.

```
pulse[0] = new RFflip(0.0, 0.1, PI/4.0, par->B0);
pulse[1] = new PulseTrap(pulse[0]->end, T_phase_dur, G_max,
         Vector3(0,0,1), Vector3(0,1,0), ramp_time, delta_G);
pulse[2] = new PulseTrap(pulse[1]->end, T_read_duration/2.0, -Gf,
         Vector3(0,0,1), Vector3(1,0,0), ramp_time, 0);
pulse[3] = new PulseTrap(pulse[2]->end, T_read_duration, Gf,
         Vector3(0,0,1), Vector3(1,0,0), ramp_time, 0);
pulse[4] = new PulseTrap(pulse[3]->end, T_phase_dur, -G_max,
         Vector3(0,0,1), Vector3(0,1,0), ramp_time, -delta_G);
```

The pulses shown are included from a pulse library which contains RF pulses and trapezoidal pulses. The library provides a pulse superclass which can be subclassed to easily add additional pulse types. The user needs to define the start time, duration, strength, field direction, gradient direction, ramp-time, and gradient strength shift for a gradient pulse. While most gradients have their field along the z-axis, the field direction allows for concomitant gradient support. The gradient strength shift specifies how much the pulse strength changes for each repeat of the sequence. This is to primarily support phase gradients. The sequence timings are all referenced to a zero time defined as the start of a repeat of the sequence (a phase encoding step). This is called the sub-sequence clock. Another clock keeps track of total simulation time. Therefore the user does not need to rewrite the pulses for each phase encoding step. The user is responsible for calculating the duration of the pulses as this is sequence specific. However, the scan parameters required for these calculations are made available in the previously discussed `SimuParams` class.

The RF pulse, `RFflip`, requires a start time, an end time, a flip angle, and the strength of the applied field at the moment it is active. The pulse uses equation 5 to calculate the required field strength to achieve the desired flip angle.

Besides pulses, the sequence is responsible for defining what times signal should be read from the simulator. In terms of k-space, the sequence defines the number of k-space points, and provides a function that links the k-space vector to signal time-step.

If the user uses the pulse library as shown in Listing 1 they do not need to implement the `getG()` function. This function takes the time and position of a particle and returns the $\vec{B}_1$ field. The name is slightly misleading as it also includes fields from RF pulses. This function is called by every thread (particle) for every simulation timestep.

4.2.1 Optimization

The `getG()` function was important to optimize due to the number of times it is called. The first version of the function loops through each pulse of the sequence. If the pulse is active for the current timestep `getG()` adds the output of that pulse to a total local magnetic field vector. This implementation is robust as it can work for any number of pulses, and pulse type. However as the number of pulses increases `getG()`'s performance will decrease. This is because the number of loop iterations is proportional to the number of pulses as seen in Listing 2.

Listing 2: The function body of the simple but slow version of **getG()**.

```
Vector3 G(0.0,0.0,0.0);
for (int i = 0; i<num_pulses; i++){
        if (pulse[i]->on(seg_time))
                G += pulse[i]->out(time, seg_time, r, seg);
}
return G;
```

While the concern is minimal for sequences such as GRE and PGSE which only have five to six pulses the simulator should be able to support complex sequences without significant performance loss. The local magnetic field experienced by a simulated particle is given by

$$\begin{bmatrix} B_x(\vec{r},t) \\ B_y(\vec{r},t) \\ B_z(\vec{r},t) \end{bmatrix} = \begin{bmatrix} G_{xx}(t) & G_{xy}(t) & G_{xz}(t) \\ G_{yx}(t) & G_{yy}(t) & G_{yz}(t) \\ G_{zx}(t) & G_{zy}(t) & G_{zz}(t) \end{bmatrix}_{pre} \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} + \begin{bmatrix} B_{RFx}(t) \\ B_{RFy}(t) \\ B_{RFz}(t) \end{bmatrix}_{pre} \tag{23}$$

where B_x, B_y, B_z are the components of the local magnetic field. The terms $B_{RF_x}, B_{RF_y}, B_{RF_z}$ are for spatially non dependent RF pulses. For simulations with ideal coils the gradients and RF pulses should be uniform. Therefore their values only depend on time and not particle position. An optimization can be made which only calculates the G and B_{RF} terms in Equation 23 once per timestep, instead of for each individual particle's timestep. This reduces the number of calculations for G and B_{RF} by a factor equal to the number of particles in the simulation. Also, only G and B_{RF} are affected when sequence complexity is increased. Therefore, the main simulation performance should no longer be linked with the number of pulses as long as the number of timesteps remains the same.

To implement the optimization the G and B_{RF} terms are pre-allocated on the device for each timestep of the simulation. This occurs before the main body of the simulation kernel starts. To minimize the time required for pre-allocation the device is used as this is a parallelizable problem. A thread on the device is assigned for each timestep of the simulation. This contrasts with the thread assignment pattern for the main kernel where each thread corresponds to a particle. The thread uses code similar to Listing 2 to calculate the $\vec{B}_1$ fields for its respective time-step. The overhead is minimal when compared to the original method. An additional benefit to using the device for pre-allocation is that no memory transfer of the G and B_{RF} arrays are required from host to device.

While the optimization saves the device from recalculating constant terms, a memory requirement is introduced. The amount of memory required is proportional to the number of time-steps in the simulation. This is related to the image resolution as more read, and phase encoding steps are required.

4.3 Samples

The samples called `primitives` are what the user loads into the virtual scanner using `add_primitive()`. The focus of the project was not to develop new samples, as this primarily deals with the diffusion aspect of the simulation. However, a sample called `CylinderXY`, which is a finite length cylinder perpendicularly bisected by the XY plane, was added. A cylinder of water is a standard MRI phantom to use. A phantom provides a basic test of a MRI scanner. If the scanner is operating properly an image of a circle should result.

The samples extend the `Primitives` class. The sample indicates to the simulator where its walls are through a set of vector intersection methods. The permeability of a sample can be set, setting a random chance that a particle leaves the sample. The diffusion coefficient is also set by the user. Lastly, the T_1 and T_2 times are required which allow the simulator to correctly evolve the magnetization. Of course each sample will have its own specific paramaters, such as a radius for a cylinder.

4.4 Signal Acquisition

The signal acquisition portion of the virtual scanner exists within the device kernel. The sequence class guides the signal acquisition process. The sequence determines when signal needs to be read. The virtual scanner then summates the magnetization values for each particle (thread) in the simulator. Mathematically the operation is

$$\vec{S} = \sum_{1}^{N} \vec{M}_i \tag{24}$$

where N is the number of particles in the simulation. Equation 24 which generates the simulator signal is different than Equation 7, the theoretical equation for signal induced in a coil. The most important difference is that the simulator signal includes contributions from both tranverse and longitudinal magnetization. Real signal is solely due to transverse magnetization. The virtual scanner only uses the transverse portion of the signal, $\vec{S_\perp} = S_x\hat{x} + S_y\hat{y}$, in the image reconstruction process. However the longitudinal portion is saved to assist in the debugging of sequences.

During the read portion of the sequence the signal is read every read time-step. This time-step is not the same as the time-step for the numerical integrator discussed in Section 3.3. The numerical integrator time-step T_{int} is set to the largest possible value that maintains numerical accuracy throughout the simulation. The read time-step is set by the distance between adjacent points in k-space, which is in turn set by the Nyquist theorem and FOV of the scan. Multiple numerical integrator time-steps occur between each read time-step evolving the magnetization. Since the numerical integrator time-step is the smallest discrete unit of time in the simulator, the read time-step length is $T_{read} = N_{int}T_{int}$. Here N_{int} is the number of integrator steps per read step. The management of simulator time-steps is show in Figure 4.

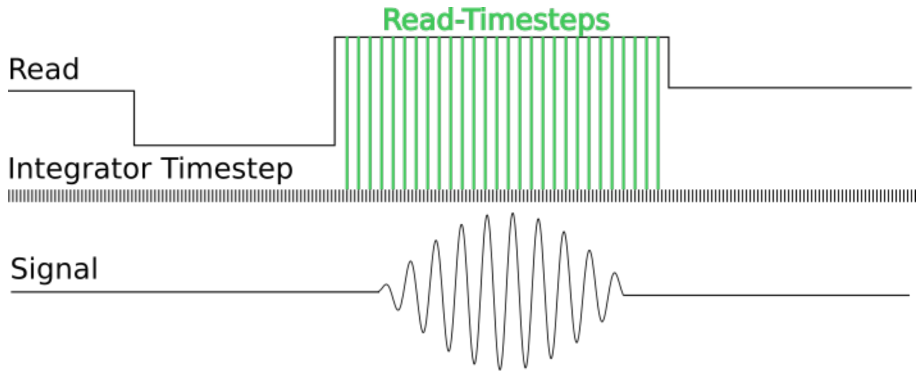


Figure 4: Integrator and read time-steps for a simulated GRE sequence. Read time-steps are in green, and occur during the read gradient. 28 read steps are present allowing for a resolution of up to 28 voxels along the frequency encoding direction in k-space. In this figure $N_{int} = 3$.

4.4.1 Optimization

Multiple implementations of the read-step operation were considered. It was necessary to balance processing time with GPU memory limitations. For a 256x256 resolution scan it takes 1.57MB to save all the read-step magnetization values per particle. At the end of the scan the GPU would perform the computation in Equation 24 for each read-step. This approach is efficient for the GPU processor as there are no interruptions during the scan to summate magnetization values. However it takes over 1.5TB of memory to store all the magnetization values for a million particle scan. The alternative option, to summate magnetization values after every time-step, minimizes memory requirements but routinely interrupts the scanning kernel. If one warp is delayed reaching a read-step the rest of the simulation is also delayed reducing occupation of GPU resources. The resulting design is a mix of both summation strategies.

Each block of threads (particles) has block-specific shared memory. This memory is quicker to access than the global memory. On each read-step the magnetization is summated to one value in shared memory. To do this all threads in the block need to reach the read step at the same time. If a single warp surpasses the read-step before having its value summated its magnetization value is lost forever. Therefore the command `__syncthreads()` is called which blocks a thread's execution until all threads reach that execution line. Once the summed magnetization vector is complete for the read-step it is copied to global memory. At the end of the simulation global memory contains N_{blocks} of magnetization vectors per read-step. Now that all read-steps are complete a dedicated summation kernel is executed which sums all the magnetization values for each read-step. These summation steps are shown graphically in Figure 5 for one read-step of the simulation.

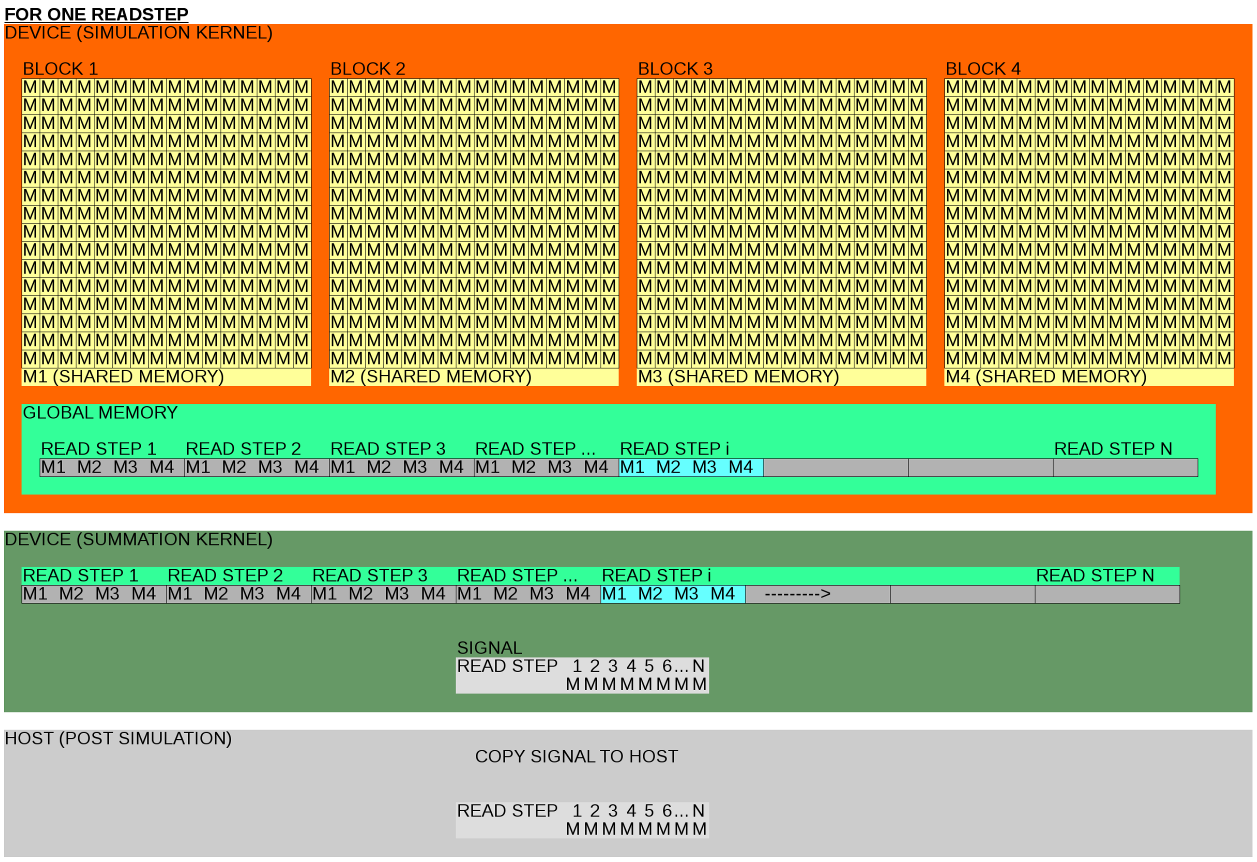


Figure 5: Thread diagram demonstrating how magnetization is summed on the device to create the simulator's signal.

Computationally, block reduction has been performed on the magnetization vectors followed by device reduction. These two types of reduction are common in parallel computing and have been previously implemented within the Cuda Unbound (CUB) library[1]. As these reduction algorithms are optimized for a range of hardware and are well documented they

[1]CUDA Unbound Library, https://nvlabs.github.io/cub/

were used in this simulator.

Now the signal acquisition process is complete. Compared to saving all magnetization vectors, the memory requirement is reduced by the number of threads in a block. Additionally the decrease in performance is specific to only the individual block. A slow warp only slows down the execution of one block of threads, not all threads on the device.

4.5 K-space

The scan is completed when k-space is filled and an inverse fourier transform is performed. K-space is defined as a two dimensional array on the host. This array is filled from elements in the signal, which has been copied from the device to the host. The sequence defines the relation between the signal time, and k-space vector. Once the k-space array is filled the CUDA Fast Fourier Transform (CUFFT) library is called to perform the inverse fourier transform of k-space on the device. Lastly, the result of the transform is copied back to the host, and saved as a grayscale image on the host's disk.

5 Results and Analysis

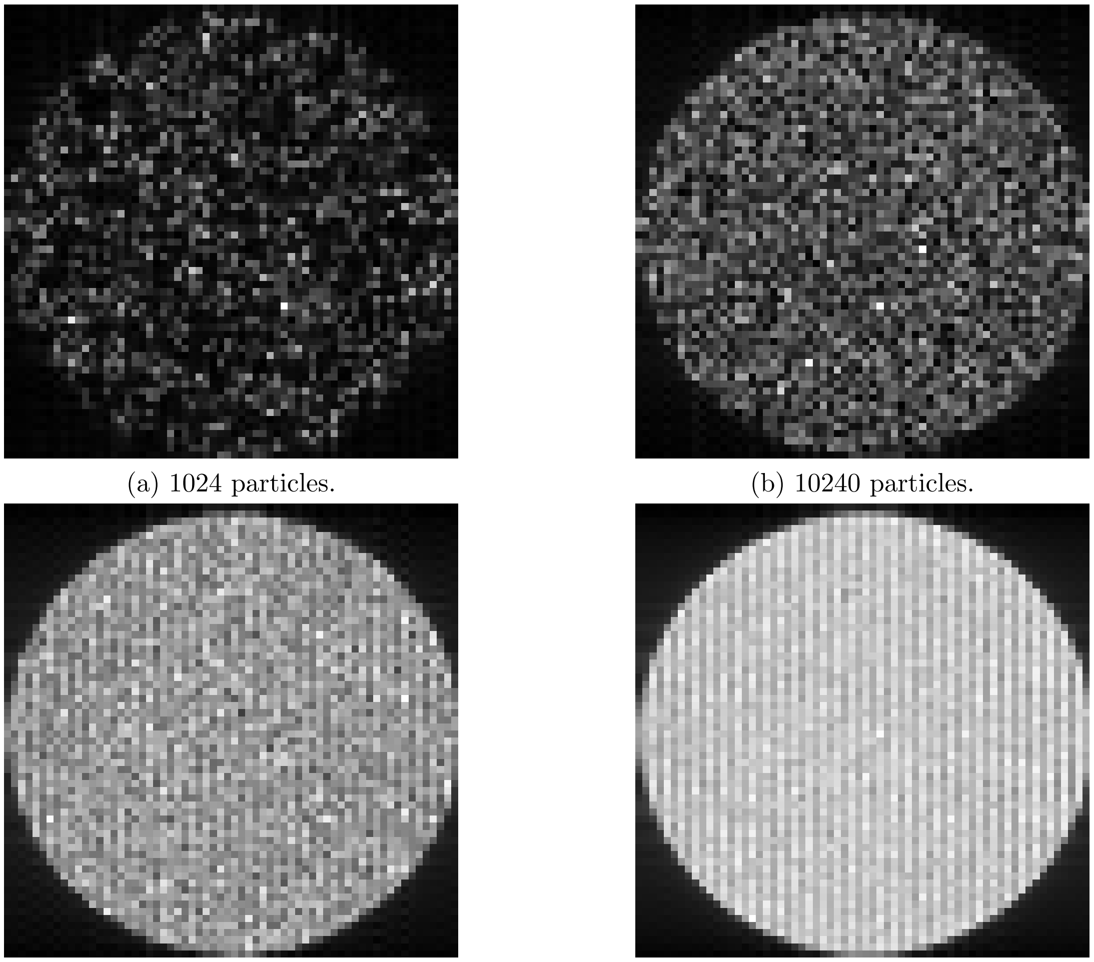
The results presented are to show the successful operation of the MRI simulator for various sequences and conditions. The performance of the simulator on GPU and CPU is also compared.

5.1 Zero Diffusion Images

One of the unique features of the simulator is that it supports diffusion. However, diffusion can mask image quality problems, and requires a more complicated sequence. Therefore the first images presented feature no diffusion.

5.1.1 Varying Particle Count

A traditional MRI phantom, the cross section of a cylinder, is used to generate the first test images seen in Figure 6. The number of particles are varied for each image. The images are all normalized to the same intensity scale. Therefore a brighter image indicates stronger signal. A GRE sequence was used to collect the images. The phantom was 4cm in diameter and 2 cm in length. It was centered at the origin of the simulated scanner bore, which is the center of the Cartesian coordinate system. No concomitant gradients were active for this scan. The main field was set to 0.001T. This allows for a slower Larmor precession, and longer integrator timestep.



(a) 1024 particles.

(b) 10240 particles.

(c) 102400 particles.

(d) 1024000 particles.

Figure 6: 64x64 resolution images of a cylinder phantom. Images were generated using a GRE sequence with varying numbers of particles.

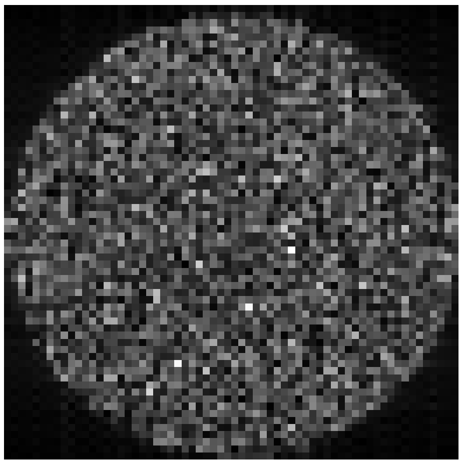
The effect of altering the number of particles is to alter the overall signal produced by the phantom as there are fewer magnetization vectors to sum. According to Equation 24 fewer magnetization vectors imply less signal. This is the observed result. The images with fewer particles have less intensity than images with many particles. This is an important capability for a MRI simulator. In real brain images the intensity of varying areas are dependent on the proton density of the particular tissues. This allows MRI scans to reveal the structure of organs. The virtual scanner is able to reproduce this effect.

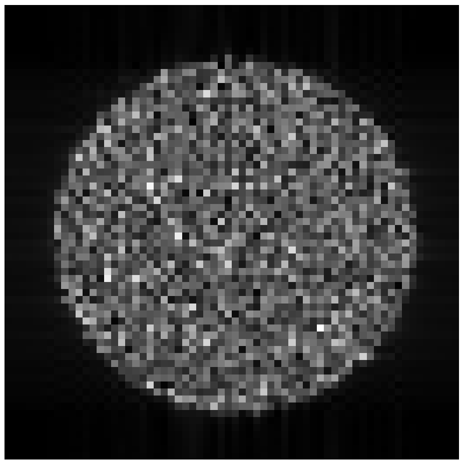
5.1.2 Setting Field of View (FOV)

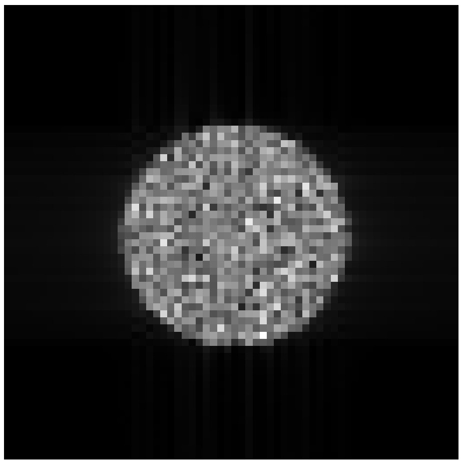
One of the important parameters of a MRI scanner is the field of view (FOV). This sets the area of imaging for a scanner. Using the correct FOV is important for image quality. If the FOV is too small artifacting occurs and the image quality is compromised. If the FOV is too large less information from the area of interest is captured resulting in reduced resolution of image details. The same phantom was used from Section 5.1.1 to demonstrate changing the field of view. 10240 particles were used in each scan shown in Figure 7. This number of particles creates enough signal for imaging, while keeping total scan times short.

The size of the phantom in each image from Figure 7 is correct with respect to the scan's FOV. The phantom in image (a) fills the entire FOV. This is as expected since the FOV and object are the same size. A more realistic scenario is presented in image (b) where some space is left around the phantom. This ensures that the phantom fits inside the entire FOV. Image (c) demonstrates correct scaling between phantom size and FOV setting. The phantom is half its original size when compared to image (a), since the FOV is now twice as large.

Image (d) makes evident a trend that is occuring as the FOV is increased. All other parameters are kept consistent between scans. The phantom is gradually becoming brighter as it is reduced in size. This occurs because the voxel size is becoming larger. Each voxel contains more spins, and thus creates more signal and SNR. The result is real, and expected


(a) 4cm by 4cm FOV.


(b) 5cm by 5cm FOV.


(c) 8cm by 8cm.

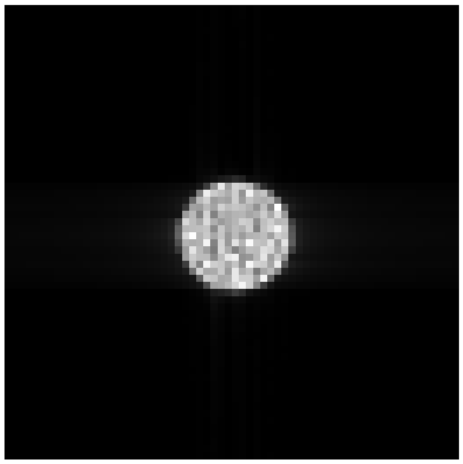

(d) 16cm by 16cm FOV.

Figure 7: 64x64 resolution images of a 4cm diameter cylinder phantom. Images were generated using a GRE sequence with a range of FOVs from 4cm by 4cm to 16cm by 16cm.

in actual MRI scans. Equation 8 shows that as the FOV is increased the SNR should also increase. Therefore the virtual scanner's FOV parameter does simulate a real scanner's FOV.

5.1.3 Scan Resolution

The resolution of a scan requires careful consideration. Ideally scans would maximize their resolution, but this comes at the cost of greater scan time and lower SNR. The effect of changing the scan resolution of the virtual scanner is demonstrated in Figure 8. The same phantom and sequence is used as in the previous section.

The resolution was doubled between each image in Figure 8. However this generates

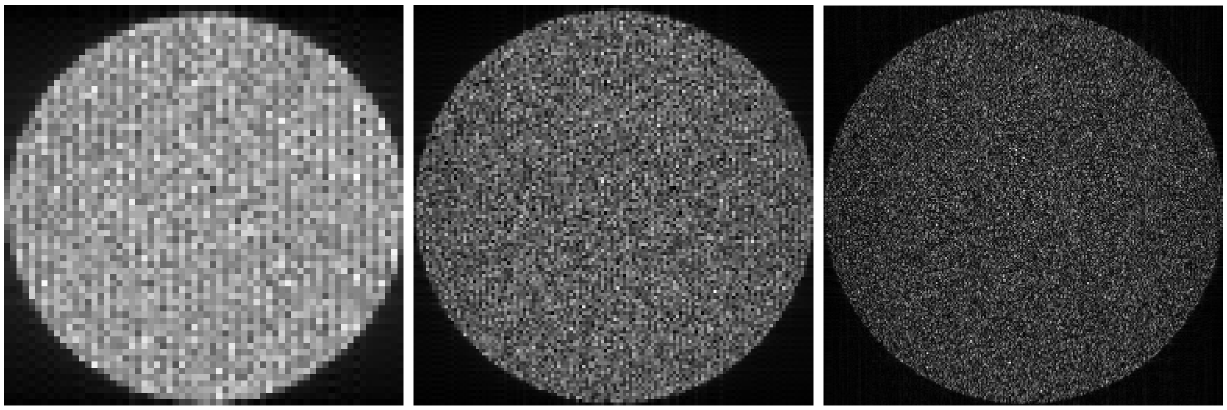

(a) 64x64 resolution (80s). (b) 128x128 resolution (240s). (c) 256x256 resolution (610s).

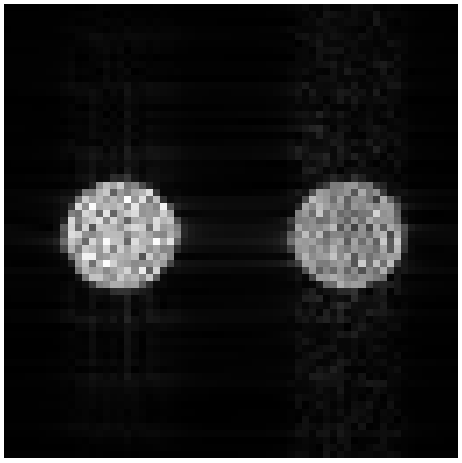
Figure 8: Images of varying resolution of cylinder phantom. Images were generated using a GRE sequence and 102400 particles.

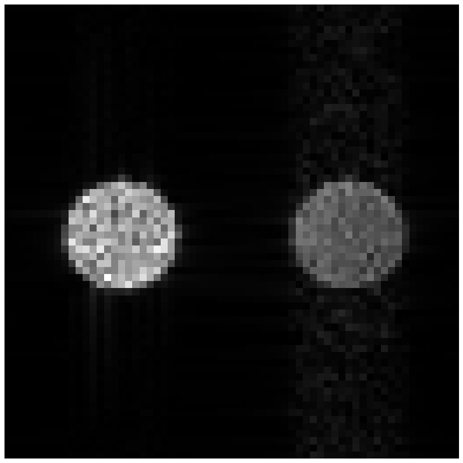
four times as many voxels. Therefore, analogous to a real MRI scanner the virtual scanner required more time to generate images of higher resolution. Image (b) took 3 times as long to generate as image (a). Image (c) took 2.5 times as long to generate as image (b). Despite the virtual scanner capturing four times as many voxels, the over sequence time was not greater by a factor of four. This is because the total number of simulation timesteps does not scale proportionally to resolution. The phase encoding steps are doubled for each doubling of resolution. This doubles the simulation time. The number of read encoding steps are also doubled, but this does not double the T_R of the sequence. The T_R is defined by multiple pulses, such as the excitation and phase encoding pulses. The read pulse is doubled but this does not double the T_R.

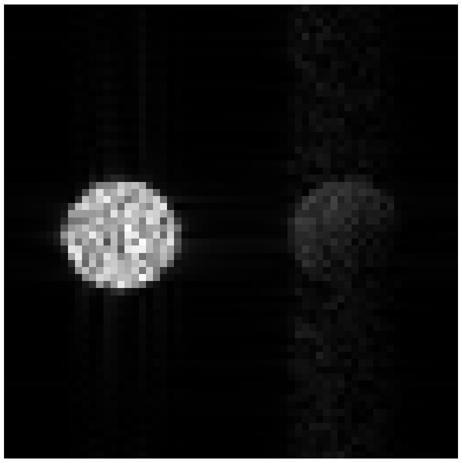
As the resolution of the image is increased the SNR decreases. This is due to Equation 8 which shows the SNR decreases with voxel size. The same number of particles are present in the high resolution scans, therefore there are less particles per voxel generating signal. The same effect was seen when changing the FOV of the scan. The virtual simulator faithfully reproduces this effect which is a concern when designing real scans.

5.2 Diffusion Images

Real MRI scanners take advantage of natural diffusion to create a special class of images. The simplest of these are called diffusion weighted (DW) images. As shown in Equation 16 the diffusion coefficient D attenuates the signal. The stronger D is the less signal will be generated. The other method to adjust signal attenuation is to alter the value b. This can be done by changing the strength of the diffusion sensitizing gradients. The stronger they are the greater b is further attenuating the signal. Both parameters b and D can be varied in the virtual scanner. The result is shown in Figure 9.


(a) Zero strength diffusion gradients.


(b) 1mT/cm diffusion gradients.


(c) 2mT/cm diffusion gradients.

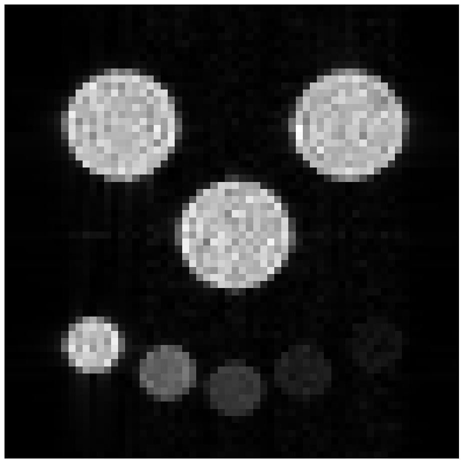

(d) Multiple diffusion coefficients.

Figure 9: Images (a), (b) and (c) contain 10240 particles and two phantoms. The left phantom has $D = 0.0$ cm^2/s, and the right phantom has $D = 0.0001$ cm^2/s. Image (d) contains 102400 particles and multiple phantoms with multiple strength diffusion coefficients. All images captured using a PGSE sequence with diffusion weighting gradients.

Images (a), (b), (c) show the effect of varing b by changing the strength of the diffusion sensitizing gradents from 0mT/cm to 2mT/cm. The PGSE sequence was used with added diffusion weighting gradients. As expected, when the diffusion weighting gradients were increased the phantom that had diffusion became darker. In the extreme case shown in image (c) the phantom with diffusion is almost completed faded. Although image (a) was generated with zero diffusion gradients the phatom with diffusion was slightly darker. This is because the normal imaging gradients are slightly sensitive to diffusion.

Image (d) contains multiple phantoms arranged in a face. The top three phantoms and bottom left phantom all have zero diffusion coefficients. The remaining bottom phantoms have increasing diffusion coefficients from left to right. As predicted by Equation 16 a gradient effect is created, with the right most phantom nearly faded to black. The image is also an example of how multiple primitives can be used to simulate more complex samples.

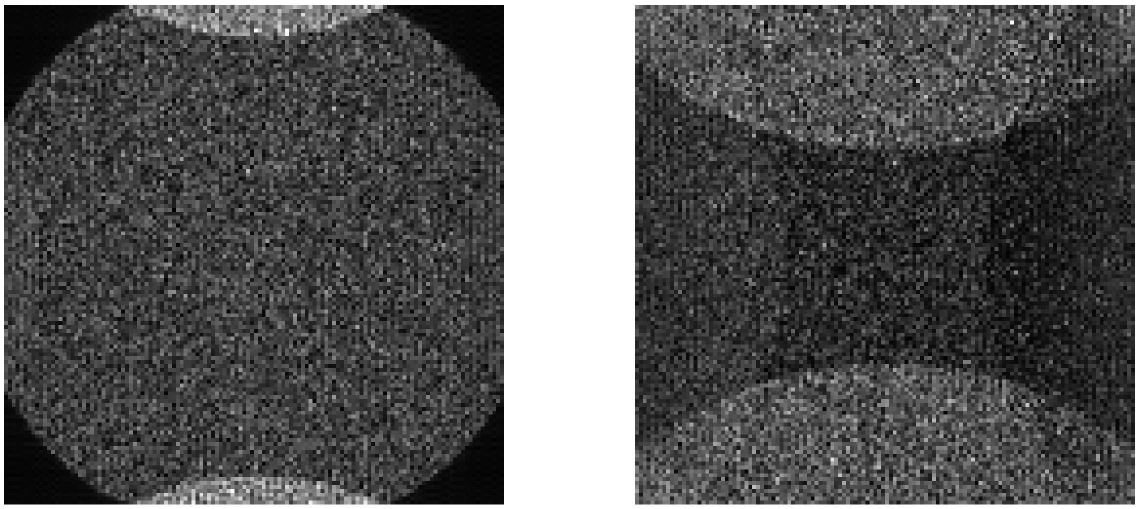
The images in Figure 9 are all examples of DW images. Since these were successfully produced the simulator should also be able to produce ADC images. As described in Section 2.5 an ADC image is a reconstruction from multiple DW images. Other more advanced types of diffusion imaging exist, but DW images form the base for these more advanced reconstruction techniques.

5.3 Artifacts

Artifacts negatively affect image quality. They are present in MRI scans when conditions are not ideal. Artifacts can result from poor sequence choice or parameter selection. They can also be generated from outside sources, such as by stray EM waves. This is why MRI scanners are often well shielded and placed in locations away from extraneous interference. Therefore it is important to consider artifacts when designing MRI sequences. The virtual scanner should be able to simulate artifacts if it is following the same imaging and processing steps as a real scanner.

5.3.1 Wrap Around

A warp-around artifact is a common artifact seen in real MRI scans. It occurs along the phase encoding direction of the image. The phase encoding step is described in Section 2.2.6. Phase encoding can only encode phases over a total range of 2π radians. Mathematically a phase of $2\pi + \Delta\pi$ is equivalent to $\Delta\pi$ where Δ is a number between 0 and 2. The FOV of an image sets the gradient strengths necessary to capture the desired imaging area with this phase range [19]. However, the gradients do not cease to exist outside of the FOV. Any sample lying outside will be encoded. In this region the phase encoding gradient encodes more than 2π of phase into the spins since it is too strong. The signal from the outlying sample areas map to the opposite side of the image. This produces a wrap around artifact, where sample from outside the FOV maps to sample within the FOV [20]. The FOV was purposefully made smaller than the sample to generate this artifact for images in Figure 10.



(a) Slight wrap around artifact. FOV is 3.5cm square.

(b) Strong wrap around artifact. FOV is 2.5cm square.

Figure 10: Images affected by a wrap around artifact. Images were captured using a GRE sequence, with 102400 particles and a 128x128 resolution. A cylindrical phantom with a 4cm diameter was used.

The phase encoding gradient was along the vertical direction for images in Figure 10. The wrap around artifact can clearly be seen by the semi-circular areas of increased signal at the top and bottom of each image. These areas appear brighter as additional signal from outside the FOV has been mapped to these areas in the image. In image (a) the FOV was made only slightly smaller than the cylinder phantom. This resulted in a small wrap-around artifact. Image (b) had a FOV that was far smaller than the cylinder phantom creating an extreme wrap-around artifact. These artifacts indicate that the virtual scanner is simulating gradients correctly, and reconstructing the image from only the scan signal.

5.3.2 Phase Cycling

Phase cycling is not an artifact, but a method to avoid artifacts in MRI images. A sequence typically consists of multiple RF pulses. In the rotating frame these RF pulses are usually applied along one axis throughout the total imaging process. These RF pulses have zero phase relative to the rotating frame. However the direction of the pulse can be varied creating a relative phase difference between the pulse and the rotating frame. This creates a linear shift in the image due to the inverse Fourier transform of k-space done during reconstruction [6]. The simulator is capable of phase cycling, and the feature was discovered accidentally. Figure 11 shows the effect of phase cycling on the simulated image.

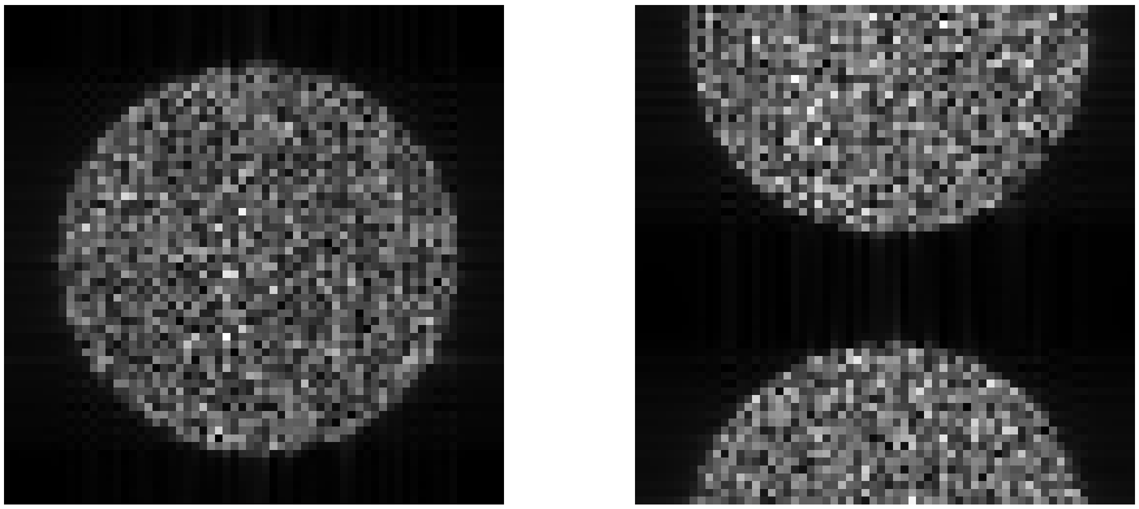

(a) Normal image without phase cycling. (b) Image with phase cycling.

Figure 11: Comparison of images taken without and with phase cycling enabled. A GRE sequence was used with a 4cm diameter cylindrical phantom and a 5cm square FOV. 10240 particles were simulated.

Phase cycling can be used to avoid artifacts by shifting the area of interest away from the artifact. Extraneous DC signals can generate bright spots at the center of MR images [6]. Phase cycling such as used in image (b) can shift the image away from this artifact. Later reconstruction can rejoin the split image and discard the artifact.

The ability to phase cycle RF pulses was accidentally discovered in the simulator when the reference times for RF pulses was miscoded. The RF pulse requires the current simulation time so that it can be applied along the same axis in the rotating frame. However, two clocks exist in the simulator. The previously referred to total time, and the sub-sequence time. The sub-sequence time is set to zero at the start of each phase encoding step. This allows for a simpler implementation of gradient pulses, as they only need to be coded once but can be repeated many times. While fine for a gradient that is stationary in the lab frame, the initial direction of the RF pulse is also set along the same axis in the lab frame for each phase encoding step. This works for the first pulse as the lab and rotating frames are initially aligned, but for future pulses the frames may not be aligned. This imparts an overall phase different resuling in phase cycling. The solution was to start and end RF pulses based on the sub-sequence time but to calculate their direction based on total simulation time.

5.4 Performance Impact of Optimizations

The physical performance of the simulator has been demonstrated. However, the usefulness of a simulator also depends on the total computational time required. The impact of using the GPU is quantified by running an equivalent CPU based simulator and comparing times. Lastly some specific optimizations are justified by showing the time savings they provided despite the additional code complexity required.

5.4.1 CPU and GPU Scanner Times

An equivalent CPU version of the simulator was written. Two key parameters strongly affect the computational time, number of particles and scan resolution. Increasing the scan resolution adds timesteps to the simulation. Increasing the number of particles adds threads to the GPU simulator and extends a loop in the CPU simulator. Figure 12 shows the comparative performance time of the CPU and GPU simulators as a function of particle count. See Appendix C for the specifications of the hardware used to conduct the test.

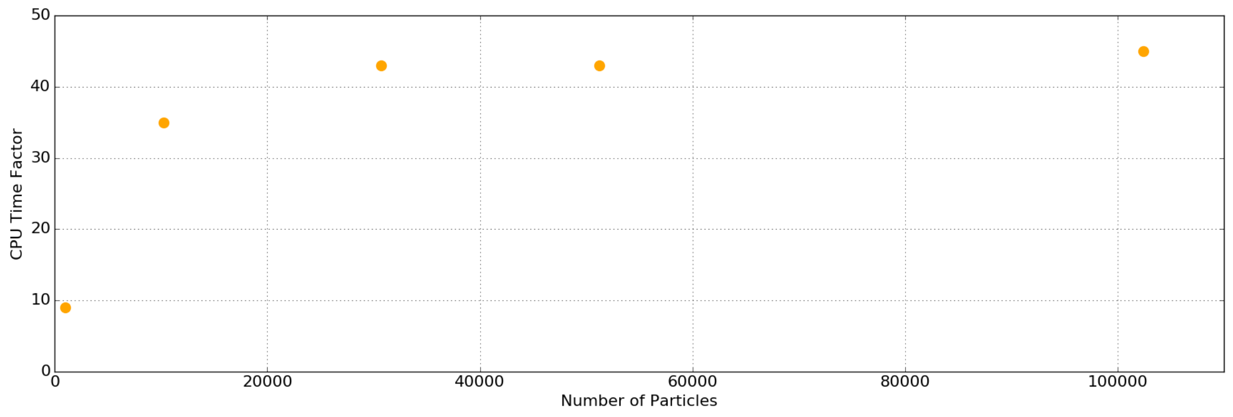


Figure 12: CPU simulation time shown as a multiple of GPU simulation time.

The CPU scanner did not have pre-allocated gradients implemented, therefore they were turned off in the GPU scanner. Overall the GPU scanner performed better than the CPU scanner. The GPU scanner was ten to fifty times faster than the CPU scanner. An interesting trend can be observed in Figure 12. Low particle counts resulted in smaller improvements in simulation time when using the GPU. This is because the GPU is not being fully utilized. In contrast, the CPU is always fully utilized regardless of the number of particles. Recall that each thread of the GPU is assigned a particle. The GPU used in the test can support 30720 threads (15 SMs with 2048 threads each). Therefore the GPU is not fully occupied until a simulation contains 30720 particles. In Figure 12 the CPU time factor slows its increase at this number of particles. As shown in Section 5.1.1 scans with particle numbers of 1024 have low SNR. Useful data is generated for higher numbers of particles and this is where the GPU performs well.

5.4.2 Effect of the Gradient Pre-Allocation

One of the optimizations made was to pre-allocate gradient values as described in 4.2.1. This avoids recalculating the gradient values for every thread. Instead, the only calculation is to multiply the position with the gradient value to find the local B_1 field. Therefore the computational time per scan should be reduced. Also the computational time should be independent of sequence selection. The effect of pre-allocated gradients on simulation time is tested for GRE and PGSE sequences. The results are shown in Figure 13.

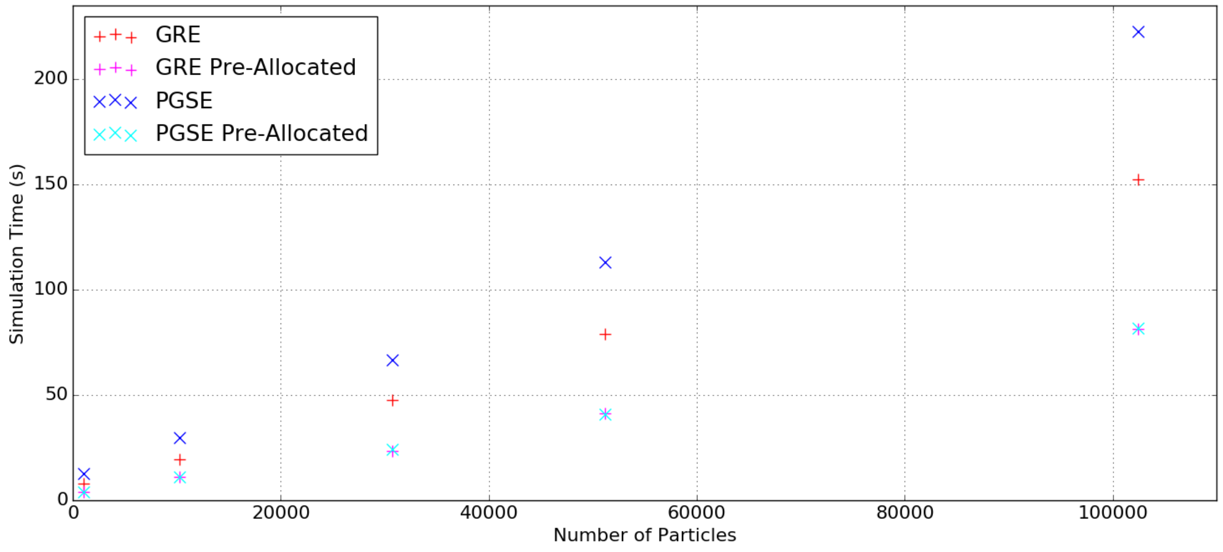


Figure 13: Computational time for GRE and PGSE sequences for a range of particles. The effect of gradient pre-allocation on simulation time was tested.

Using pre-allocated gradients decreased the simulation time for the GRE sequence by 1.9 times. The PGSE sequence experienced a greater performance benefit reducing simulation time by a factor of 2.9. The PGSE sequence included diffusion weighting pulses. When combined with normal imaging pulses there are a total of twelve pulses. The GRE sequence does not have diffusion weighting pulses requiring a total of five pulses only. Therefore using gradient pre-allocation has a stronger effect on sequences with a greater number of pulses. This was predicted in Section 4.2.1 as the `getG()` function has to spend more time checking if pulses are activated with higher pulse count sequences. Also pulse shapes have to be calculated more frequently as well if the pulse is active.

Without pre-allocated gradients the PGSE sequence took 1.5 times longer to run than the GRE sequence. This agrees with the current understanding of the `getG()` function. With pre-allocated gradients the runtimes of the GRE and PGSE sequences were identical. This indicates that with pre-allocation the simulation time has no dependence on sequence selection. The PGSE and GRE sequences are relatively simple MRI sequences. Using pre-allocated gradients the simulator is able to support complex sequences without an increase in simulation time.

6 General Discussion

The developed simulator is able to successfully simulate DMRI, while maintaining a large performance gain when run on a GPU. However there are limitations present as well. Since the simulator was written for low field MRI applications its performance is not optimized for high field MRI. When higher B_0 fields are used the Larmor precession frequency increases. The integrator timestep must be reduced to have enough solver points per Larmor rotation. Too few integrator timesteps per rotation of magnetization vector results in large integrator error. This error is included in the magnetization update. The next iteration of the solver then adds more error, and so on and the solution quickly diverges. Therefore it is necessary to increase the number of timesteps in a high field scan of the same total time. With a higher B_0 field stronger B_1 fields can be used, shortening the sequence time. Therefore it is possible to keep the same number of timesteps per scan if B_1 and B_0 are scaled together. However, this is not realistic of a high field scan.

High-field MRI simulators such as DMS or WCDMC use an alternative means of calculating magnetization. Since the z-component of a particle's magnetization vector is far less than the main magnetic field it is ignored. Only the inplane components of a magnetization vector are of interest in the high field limit. The phase of the magnetization vector, which contains only the inplane components, is tracked relative to the lab frame. Bloch's equations no longer require solving and an integrator is no longer required. This significantly reduces the computational complexity of the simulation allow for fast times in the high field regime. A similar algorithm can be implemented in the present simulator. A user defined parameter could switch the simulator into a high field approximation. This was not implemented in this simulator as a low field MRI, Bloch solving, diffusion simulator on GPU is more novel than a high field diffusion simulator that runs on GPU. However, to allow for easy comparison between high and low field MRI regimes the feature would be useful.

One undeveloped but important to note possible feature of the simulator is the ability to run on multiple devices. All scans are run in CUDA streams on the device. These streams can easily be assigned to different devices as part of the CUDA API. The simulation times should reduce linearly as devices are added. Two strategies exist for spreading the simulation load. Some MRI sequences can treat each phase encoding step as a separate simulation. If the T_1 and T_2 times are sufficiently short, or spoiling gradients are employed, little transverse magnetization remains at the end of a phase encoding step. Therefore the phase encoding steps are independent of each other. This feature is implemented in the simulator as it is a useful debugging tool. Each phase encoding step is launched in its own stream. If the scan issue is resolved with indepedent phase encoding steps than the issue was due to remaining transverse magnetization at the end of the previous phase encoding step. However for sequences where transverse magnetization does remain the number of particles can be split among devices. If a million particle simulation is required, two half million particle scans can be run each on a device. The signal from each device can be summed together to replicate a single million particle scan. This is allowed as the particles are independent of each other.

While the simulator uses a GPU to achieve usable run times, there is a cost to the portability of the software. The GPU was programmed using NVIDIA's CUDA API limiting the simulator to running only on NVIDIA GPUs. To achieve similar run times on CPU based machines a server with multiple CPUs would be required. This incurs a greater hardware cost than purchasing a GPU to run the simulator.

As is the case with software development there were a few bugs after initial completion of the simulator. One of these was related to the use of CUDA's random number generator and is documented in Appendix A. The bug was resolved and is not contained in any of the images presented in this thesis. Inevitably additional bugs will be found and resolved with continued use of the simulator.

7 Conclusion and Future Work

The purpose of this project was to create a novel MRI simulator. A complete simulator of low field diffusion MRI that supports GPU has not been previously developed. Therefore the developed simulator serves as a base for low field MRI research. The physical performance of the simulator was verified using a standard MRI phantom. The effects of particle density, image FOV, and image resolution were demonstrated. The resulting images coincided with expectations, confirming that the simulator was modelling basic MRI processes correctly.

The diffusion capability of the simulator was tested. As the diffusion algorithm was used from a previous simulator, it was not tested for numerical accuracy. When diffusion was activated the samples with diffusion experienced signal attenuation. The signal attenuation also varied with the strength of applied diffusion weighting gradients. These results were as expected, and show that the simulator is useful for modelling DMRI.

To demonstrate that the simulator was faithfully following a real MRI pipeline a couple image artifacts were shown. Phase wrapping occurred as on real MRI scanners when the FOV is smaller than the sample. This indicated that imaging gradients were properly extending outside of the FOV wrapping signal onto the image. It also showed that no a priori information was being used of the sample to simulate the MR image. Phase cycling occurred as a result of chosing the wrong simulator clock for setting the phase of excitation pulses.

However, this result is also seen in real MRI and often purposefully employed to avoid image artifacts.

The simulator showed a significant speed up of 50 times when executed on a GPU as opposed to a CPU. Some optimization has been done, such as using pre-allocated gradients and carefully summating signal throughout the scan. However shared and constant memory was not fully exploited. The CPU automatically manages its cache, where as the GPU cache is managed by the program. Increased use of the shared memory would provide a fairer comparison between the GPU and CPU. It is expected that the GPU performance would further improve relative to the CPU.

The simulator is now at a stage of development where more complex experimental sequences can be run. The simulator is able to indicate how these sequences would run on a real scanner. Scans with concomitant gradients were not performed as part of the results, but the resulting simulator supports their use. The effect of concomitant gradients on signal from samples with large diffusion coefficients can be explored. More complex samples can be added, although the simulator has not been intended to simulate biological tissue. The simulator could be used to explore medical applications of low field MRI if a more sophisticated method of modelling samples was implemented. Specifically, a tetrahedral mesh would allow for generically shaped samples to be created that could model tissue.

The presented simulator has been shown to successfully model its MRI regime while maintaining realistic run times. Both computational and physical performance can be further enhanced. The strength of this simulator is the flexibility of possibly modelled MRI regimes. Concomitant gradients and diffusion can be deactivated and activated as required. The simulator is therefore a powerful debugging tool for troubleshooting and designing a range of MRI experiments.

References

[1] D. A. Yablonskiy, A. L. Sukstanskii, and J. J. Ackerman, “Image artifacts in very low magnetic field mri: The role of concomitant gradients,” *Journal of Magnetic Resonance*, vol. 174, pp. 279–286, 2005.

[2] C.-H. Yeh, B. Schmitt, D. Le Bihan, J.-R. Li-Schlittgen, C.-P. Lin, and C. Poupon, “Diffusion microscopist simulator: A general monte carlo simulation system for diffusion magnetic resonance imaging,” *PLoS ONE*, vol. 8, pp. 1–12, 10 2013.

[3] C. A. Waudby and J. Christodoulou, “{GPU} accelerated monte carlo simulation of pulsed-field gradient {NMR} experiments,” *Journal of Magnetic Resonance*, vol. 211, no. 1, pp. 67 – 73, 2011.

[4] T. Vincent, “Monte carlo diffusion mri simulations on the gpu,” undergraduate honours thesis, University of Winnipeg, May 2013.

[5] F. Bloch, “Nuclear induction,” *Physical Review*, vol. 70, pp. 460–474, October 1946.

[6] M. A. Bernstein, K. F. King, and X. J. Zhou, *Handbook of MRI Pulse Sequences.* Elsevier, 2004.

[7] R. W. Brown, Y.-C. N. Cheng, E. M. Haacke, M. R. Thompson, and R. Venkatesan, *Magnetic Resonance Imaging, Physical Principles and Sequence Design.* Wiley-Blackwell.

[8] D. W. McRobbie, E. A. Moore, M. J. Graves, and M. R. Prince, *MRI from Picture to Proton.* Cambridge University Press.

[9] D. G. Norris and J. M. S. Hutchison, “Concomitant magnetic field gradients and their effects on imaging at low magnetic field strengths,” *Magnetic Resonance Imaging*, vol. 8, pp. 33–37, 1990.

[10] A. D. Elster, “Gradient-echo mr imaging: Techniques and acronyms,” *Radiology*, vol. 186, pp. 1–8, 1993.

[11] Z. Xian, C. Bidinosti, J. L. Hobson, and M. E. Hayden, “How to unwind in the low field limit,” *Proc Intl Soc Mag Reson Med*, vol. 15, 2007.

[12] A. Einstein, *Investigations on the Theory of The Brownian Movement.* Dover Publications Inc., 1956.

[13] D. S. Lemons and P. Langevin, *An Introduction to Stochastic Processes in Physics.* The Johns Hopkins University Press, 2002.

[14] P. Hagmann, L. Jonasson, P. Maeder, J.-P. Thiran, V. J. Wedeen, and R. Meuli, “Understanding diffusion mr imaging techniques: From scalar diffusion-weighted imaging to diffusion tensor imaging and beyond,” *RadioGraphics*, vol. 26, pp. S206–S224, 2006.

[15] S. Cook, *CUDA Programming, A Developer’s Guide to Parallel Computing with GPUs.* Elsevier, 2013.

[16] *Nvidia GeForce GTX 1080 Whitepaper*, 2016.

[17] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach.* Elsevier.

[18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press.

[19] J. Zhuo and R. P. Gullapalli, “Aapm/rsna physics tutorial for residents,” *RadioGraphics*, vol. 26, no. 1, pp. 275–297, 2006.

[20] L. Axel and D. Morton, “Correction of phase wrapping in magnetic resonance imaging,” *Med. Phys*, vol. 16, no. 2, pp. 284–287, 1989.

A Use of CUDA's Random Number Generator

Random number generation is crucial to the diffusion simulation part of the project. The positions of the particles are randomly placed and moved inside the scanner's sample. CUDA's `curand_uniform()` function was used for the GPU version of the simulator. C's `rand()` function was used for the CPU implementation. Theoretically these two methods should produce the same images. Initially this was not the case as shown in Figure 14.

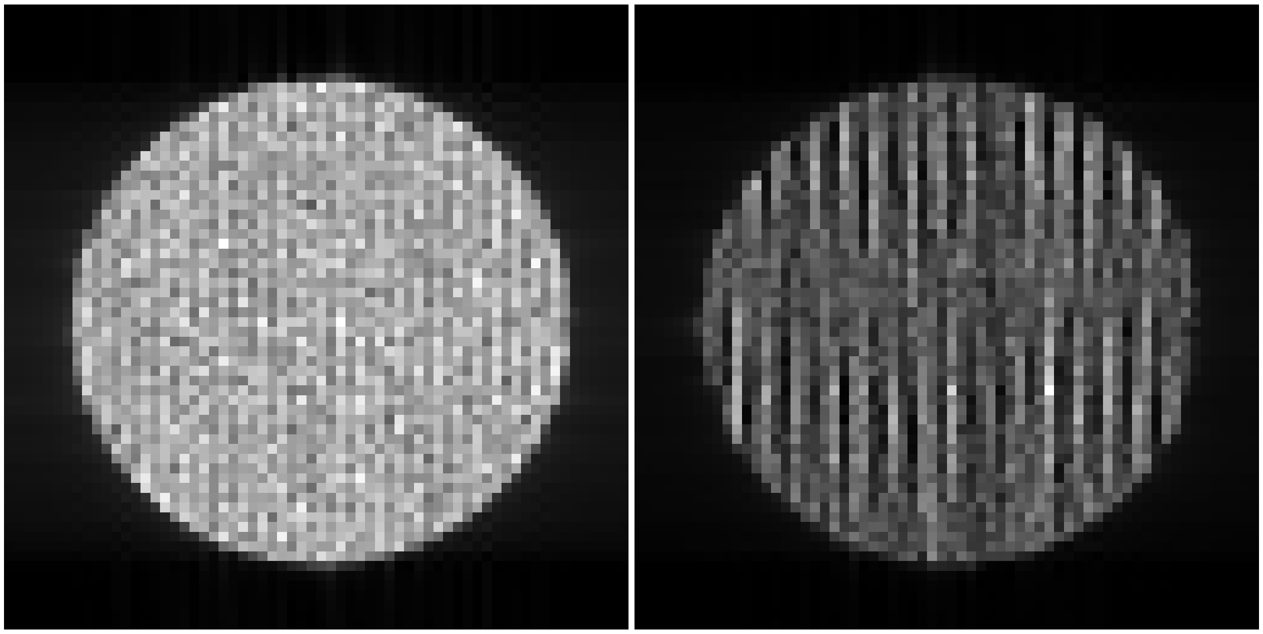
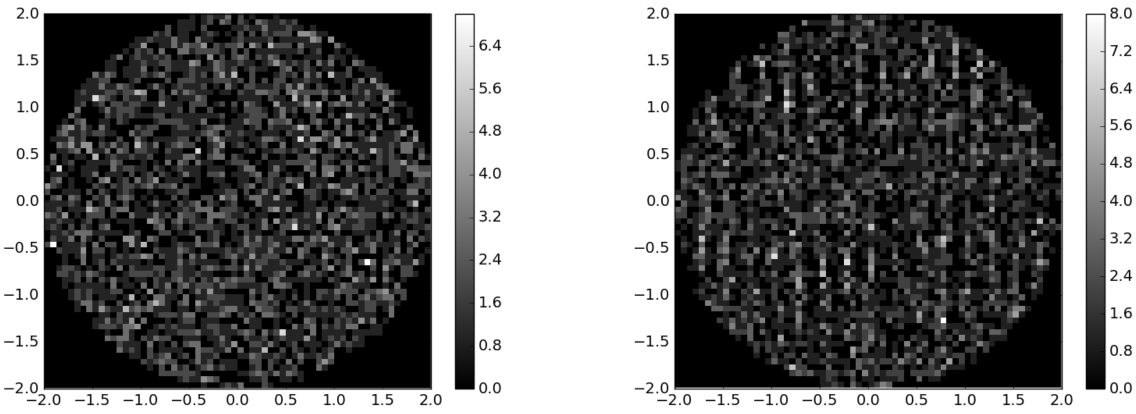

(a) Correctly simulated CPU image. (b) Image with effect of GPU simulator bug.

Figure 14: Effect of simulator bug on image quality.

Image (b) shows two strong bands of aliasing along the frequency encoding (horizontal) direction of the image. The initial assumption was that the artifacting was due to a mistake in the frequency encoding gradient. Specifically that the Nyquist sampling theorem was not being followed correctly. However changing sampling rate, FOV, and resolution did not affect the artifact. These parameters are linked to the Nyquist sampling theorem and therefore should have had an effect.

The alternative was that the virtual scanner was working correctly. In this case the reason image (b) contained aliasing was because it was physically present in the sample's particle density distribution. This hypothesis was supported when turning on diffusion reduced the effect of aliasing. Additionally the image from the CPU version of the simulator did not contain this artifact as shown in image (a). To confirm that the initial particle distribution was incorrect a 2D histogram in Figure 15 was taken of particle positions immediately after sample initialization.



(a) CPU (b) GPU

Figure 15: Histograms of initial particle positions on CPU and GPU.

Image (a) shows that the CPU correctly initialized particle positions. Initializing particle positions should be akin to randomly throwing darts at a dartboard. Image (b) shows the GPU's random particle initialization. A similar aliasing pattern is observed indicating a non random particle initialization. It could now be concluded that the artifact in the GPU images was due to improper particle initialization. This was important as it confirmed that the simulator's MRI pipeline algorithms were not the problem.

A GPU specific random initialization error indicated that the `curand_uniform()` function was not working properly. No software random number generator is truly random, but `curand_uniform()` should be adequate for the simulator. However the images (b) from Figures 14 and 15 indicated that it was not.

CUDA's random number generator is initialized for each thread on the device before the main simulator kernel runs. Each thread/particle should have its own sequence of random numbers so that it moves independently of other particles. This is done by using a kernel called `setup_kernel()` as shown in Listing 3.

Listing 3: The flawed random number initialization function initially used in the simulator.

```
__global__ void setup_kernel( curandState * state, int seed )
{
int tid = threadIdx.x + blockIdx.x*SIM_THREADS;
curand_init( seed + tid, 0, 0, &state[tid] );
}
```

Using `curand_init()` each thread was initialized with its own seed. The seed value was incremented for each thread according the thread's id, tid. Typically a different seed dictates a different random sequence of numbers. However CUDA's documentation mentions

> Sequences generated with different seeds usually do not have statistically correlated values, but some choices of seeds may give statistically correlated sequences. Sequences generated with the same seed and different sequence numbers will not have statistically correlated values.[2]

which explains why `curand_uniform()` was not producing random results. However `curand_init()` has a field for sequence number, which typically correlates to tid. The current and more common version of the `setup_kernel()` is given in Listing 4.

Listing 4: The flawed random number initialization function initially used in the simulator.

```
__global__ void setup_kernel( curandState * state, int seed )
{
int tid = threadIdx.x + blockIdx.x*SIM_THREADS;
curand_init( seed, tid, 0, &state[tid] );
}
```

This version guarantees that each thread has its own random sequence of numbers. This resolves the artifacting shown in image (b) from Figure 14. Now images from the GPU and CPU are identical.

[2]http://docs.nvidia.com/cuda/curand/index.html

B Code

Select sections of simulator code that were discussed throughout the thesis are documented. These are areas of code that required extensive optimization, careful thought, or would be regularly interacted with by an end user.

B.1 Demo Scan

An example of a basic scan of a single cylindrical phantom is shown in Listing 5.

Listing 5: A demo scan of a single cylindrical phantom.

```
//Main simulator library.
#include "master_def.h"

//Specific coil, sequence... for this simulation.
#include <iostream>
#include "sequence/GRE.cuh"
#include "coil/coil_ideal.cuh"
#include "scanner/scanner.cuh"
#include "primitives/CylinderXY.cuh"
#include "params/simuParams.cuh"
#include "util/recorder.h"
#include "util/vector3.cuh"

int main(){
//Simulation properties.
SimuParams test_params(
        102400, //Number of particles.
        102400, //Number of particles per stream.
        2.5,    //Sequence repeat time.
        0.5,    //Sequence echo time.
        0.001,  //Simulation timestep.
        0,      //Number of particles to track continual,
                //individual magnetization.
        Vector3(0, 0, 1),       //Initial magnetization vector.
        Vector3(0, 0, 0.001),   //Main B0 field direction / strength.
        64,     //(vertical) resolution.
        64,     //(horizontal) resolution.
        5,      //(vertical) FOV.
        5       //(horizontal) FOV.
);

Coil_Ideal test_coil;
GRE test_sequence(&test_params);
test_sequence.save_pulse_diagram();
Scanner test_scanner(test_sequence, test_coil, test_params);
Cylinder_XY test_primitive(Vector3(0.0,0.0,0.0), 2.0, 2.0,
        2.0, 4.0, 0, 0, 0, 10240);
test_scanner.add_primitive(test_primitive);

test_scanner.scan();

//Post simulation commands.
test_scanner.acqs[0]->save_signal("signal");
test_scanner.acqs[0]->save_tracked("test_track");

return 0;
}
```

B.2 GRE Sequence Class

One of the included sequences in the simulator is the GRE sequence shown in Listing 6. Users can add their own sequences using this sequence as a template. Many required functions are implemented by the sequence superclass.

Listing 6: A GRE sequence implementation.

```
#include "GRE.cuh"
#include "pulses.cuh"

__global__ void GRE_GPU(Sequence** obj_ptr, SimuParams* par,
        int phase_enc_offset = 0, int _local_res_x = 0);

__device__ __host__ GRE::GRE():Sequence(){}

__device__ __host__ GRE::GRE(SimuParams* par, int _phase_enc_offset,
        int _local_res_x) : Sequence(1,5, phase_enc_offset, _local_res_x, par)
        {
        //Calculate the available bandwidth.
        BW = 1/(10*par->timestep)*(1.0/2.0);
        //Calculate the strength of the read gradient.
        Gf = 4.0*PI*BW*(1.0/par->FOVy)*(1.0/GAMMA);
        //Length of ramp time for trapezoidal pulses.
        ramp_time = 0.005;
        //Used when running each phase encoding step as a separate scan.
        phase_enc_offset = _phase_enc_offset;
        //How long the read gradient is on.
        T_read_duration = par->res_y/(2*BW);
        //Set desired duration of phase encoding gradient.
        T_phase_dur = 0.5;
        //Calculate the total number of readsteps for the sequence.
        readSteps = local_res_x * par->res_y;
        //Number of phase steps required.
        phase_steps = par->res_x - 1.0;
        //Total number of timesteps in simulation.
        steps = (TR/par->timestep)*local_res_x;
        //Maximum phase encoding gradient.
        G_max = (1.0 / (par->FOVx)) * (PI / GAMMA) *
                (par->res_x -1.0)* (1.0 / (T_phase_dur));
        //Avoid division by zero if only one phase encode gradient.
        if (phase_steps == 0){
        delta_G = 0;
        }
        else {
        //Amount phase encoding strength changes per TR.
        delta_G = (2.0 * G_max) / (par->res_x);
        }
        //How many time-steps for each read-step.
        readFactor = (int) ((T_read_duration-2*ramp_time)
                / (par->timestep*(par->res_y - 1)));

        //Define all the pulses for the sequence.
        pulse[0] = new RFflip(0.0, 0.1, PI/4.0, par->B0);
        pulse[1] = new PulseTrap(pulse[0]->end, T_phase_dur, G_max,
                Vector3(0,0,1), Vector3(0,1,0), ramp_time, delta_G);
        pulse[2] = new PulseTrap(pulse[1]->end, T_read_duration/2.0, -Gf,
                Vector3(0,0,1), Vector3(1,0,0), ramp_time, 0);
        pulse[3] = new PulseTrap(pulse[2]->end, T_read_duration, Gf,
                Vector3(0,0,1), Vector3(1,0,0), ramp_time, 0);
        pulse[4] = new PulseTrap(pulse[3]->end, T_phase_dur, -G_max,
                Vector3(0,0,1), Vector3(0,1,0), ramp_time, -delta_G);

        //Specific behaviour if class isn't on GPU.
        #ifndef __CUDA_ARCH__
                printf("BW: %f\n", BW);
                printf("read-factor float %f\n", (T_read_duration
                        / (par->timestep*(par->res_y - 1))));
                printf("Max phase encoding step: %f\n", G_max);
                printf("Max freq encoding step: %f\n", Gf);
                printf("Read factor: %d\n", readFactor);
                //Allocate class memory for GPU.
                safe_cuda(cudaMalloc(&dev_ptr, sizeof(GRE**)));
                //Build identical class for device and store reference.
                GRE_GPU<<<1, 1>>>(dev_ptr, par->devPointer,
                        _phase_enc_offset, _local_res_x);
                //If using pre-allocated gradients.
                #ifdef ALLOC_G
                        make_tensors();
                #endif
        #endif
}

//Special constructor for preparing each phase encoding step to run in its
//own specific CUDA stream.
__device__ __host__ GRE::GRE(bool parallel, SimuParams* par)
        : Sequence(par->res_x,5, 0, 0, par), parallel(parallel)
        {
        printf("Creating sub-sequences.\n");
        sub_seq = new GRE[par->res_x];
        for (int i = 0; i < par->res_x; i++){
                sub_seq[i] = GRE(par, i, 1);
        }
}

//Function that correlates timestep to k-space position.
__device__ __host__ Vector3 GRE::getK(int readStep) const{
        int k_y = readStep % par->res_y;
        int k_x = readStep / par->res_y;

        return Vector3(k_x, k_y, 0);
}

//Returns the starting position in k-space.
__device__ __host__ int GRE::get_k_start() const{
        return phase_enc_offset*par->res_y;
}

//Returns the end position in k-space.
__device__ __host__ int GRE::get_k_end() const{
        return (local_res_x+phase_enc_offset) * par->res_y;
}

//Returns individual phase encoding steps if applicable.
__host__ const Sequence* GRE::getSubSequences(int i) const{
        if (parallel)
                return &sub_seq[i];
        else
                return this;
}

//Start time to start reading signal.
__device__ __host__ real GRE::getReadStart(real time) const{
        int seg = (int)(time / TR);
        return seg*TR + pulse[3]->start;
}

//End time to end reading signal.
__device__ __host__ real GRE::getReadFinish(real time) const{
        int seg = (int)(time / TR);
        return seg*TR + pulse[3]->end;
}

//Sequence specific kernel to initialize class on device.
__global__ void GRE_GPU(Sequence** obj_ptr, SimuParams* par,
        int _phase_enc_offset, int _local_res_x)
        {
        *obj_ptr = new GRE(par, _phase_enc_offset, _local_res_x);
}
```

C Hardware

Table 1 lists the hardware used to develop and run the simulator.

Table 1: The hardware specifications of the system used to test the simulator.

System	Host (CPU)	Device (GPU)
Name	AMD Phenom II X4 955	Geforce GTX 1070
Clock	4.0 GHz (Overclocked from 3.2 GHz)	1582MHz - 1771MHz
Memory	DDR3 8 GB	GDDR5 8 GB