Monte Carlo Diffusion MRI Simulations on the GPU

Trevor Vincent

Undergraduate Honours Thesis

Advisor

Dr. Chris Bidinosti

Committee Members

Dr. Melanie Martin, Dr. Jeff Martin

University of Winnipeg

Winnipeg, MB Canada

May 13, 2013

Abstract

A novel design is presented for Monte Carlo Diffusion MRI (MRI-DMC) simulations with general pulse sequences and general diffusion regions on a multiple NVIDIA-GPU system. This design is shown to be faster than a previous design by Waudby and Christodoulou, with a speedup of almost a factor of two when using double precision and by a factor of 1.25 when using single precision. Furthermore, we also showed that our MRI-DMC simulations on the GPU could obtain a performance increase over a single-threaded CPU by up to 200 times, which is definitely noteworthy. To test the accuracy of the code, the simulation output was compared against the three standard analytical solutions for diffusion in a sphere, cylinder and between parallel planes. The analytical solutions matched the simulation output very closely. Finally, the simulations are used to investigate low-field gradient pulse sequences, where concomitant gradient terms make analytical solutions impossible to obtain. As far as the author is aware, this is the first time investigations into the low-field regime have been done with MRI-DMC simulations.

Acknowledgements

I would like to thank my supervisor Dr. Chris Bidinosti for the idea behind this project and his guidance throughout. I would also wish to thank my Honours Committee members Dr. Melanie Martin and Dr. Jeff Martin for their useful suggestions and feedback.

Contents

List of Figures

1 Outline

Diffusion magnetic resonance imaging (MRI) uses the magnetic moments of particles to indirectly measure their diffusive motion in a sample. In diffusion MRI, the diffusive motion of the particles can be summarized by a single quantity known as the apparent diffusion coefficient (ADC). In a homogeneous sample, the ADC will be constant and rather uninteresting. However, if the sample has structure, in the form of restricting boundaries or compartments, then the ADC changes with time and represents the diffusive motion of the fluid(s) over all compartments of the sample. Furthermore, if the structure changes with time, this will also be reflected in a change in time of the ADC. Thus, studying the effects of tissue structure on the ADC in various ideal cases could help us to use it as a probe of the structure in more complicated samples. In the past, analytical solutions for the ADC in simple restricting systems such as a sphere or cylinder have been derived (see for example [1]). But with the dawn of the computer, numerical methods have also been used to calculate the ADC. The most popular, and perhaps one of the best numerical methods for calculating the ADC is the Monte Carlo (MC) method. With the MC method, we can not only calculate the ADC in arbitrarily complex regions, but we can simulate the effects of arbitrarily complex diffusion MRI magnetic fields. MC numerical simulations thus allow us to dramatically increase the theoretical knowledge base behind identification of tissue structure through ADC.

While the MC numerical method is a robust and accurate method of calculating the ADC, it can be slow. Simulations involving complex diffusion regions can take hours or days when computed on a modern CPU. Fortunately, there has been a recent trend of running numerical simulations using the processing power of the graphics card. The graphics processing unit (GPU), located on the graphics card, contains hundreds of cores which execute instructions in parallel. With this computational power, we could potentially increase performance over a single CPU core by a hundred times, which would minimize the slowness of the MC algorithm. Simulations could potentially be run in seconds instead of hours, allowing us to be more productive and efficient with our investigations involving the ADC.

In this thesis I will develop the theory behind Monte Carlo diffusion MRI (MRI-DMC) simulations and design code to implement the numerical method on the GPU. In the first section of the thesis, entitled “Theoretical Background", I will describe the theory behind diffusion MRI Monte Carlo (MRI-DMC) simulations. The section after the theoretical background, is entitled “The GPU and CUDA" and takes a close look at the NVIDIA hardware and software which now allows scientific computation on the graphics card. The second last section entitled “MRI-DMC on the GPU" will delve into the specifics of a previous design of MRI-DMC simulations on the GPU (WCDMC) and my more flexible, faster, completely different design on the GPU (TDMC). Finally, in the section entitled “Results", I will first test the simulations in analytically solvable systems to see if they are accurate. After this testing, I will use the simulations to investigate low-field diffusion magnetic resonance imaging (MRI), which, as far as I am aware, has not been investigated using MRI-DMC simulations.

All CPU code will be in C/C++ and I will heavily use C++ classes and templates, which allow for incredible generality not obtainable in C. The majority of the code and the derivations will be in the Appendix, so as to not clutter up the text. However, as I introduce topics, I may provide codes or pseudo-codes which illustrate how to implement an idea in C/C++. This allows for a gradual build-up rather than throwing a bunch of code at the reader all at once. For the most part, this code will use only standard library functions in C/C++, with two exceptions. In my code I consistently use a modified version of the Vector3 C++ class created by Ian Millington [2]. This class represents a vector in 3D space and has overloaded operators ($+$, $-$, $*$, $\%$) representing the standard vector operators ($+$, $-$, $\cdot$, $\times$). The class definition is provided in Appendix B.1. The second exception is that I use a C++ typedef called “real" to represent either a single precision number or a double precision number, which can be switched at whim by changing one line of code.

2 Theoretical Background

2.1 Macroscopic Diffusion

Diffusion is a transport process where particles spread into other particles through collisions. The spread of particles could be into themselves. That is, you could have water diffusing into water, for example. This is sometimes called self-diffusion, but in this thesis we will not distinguish between the two. In 1855, Adolf Eugen Fick postulated that the rate of transfer of a diffusing substance through a unit area, denoted $\boldsymbol{J}$ is

$$\boldsymbol{J} = -D\nabla C, \tag{1}$$

where C is the concentration field of the diffusing substance and D is a proportionality constant known as the diffusion coefficient [3]. Qualitatively, Equation (1) states that particles flow from regions of high concentration to regions of low concentration (hence the negative gradient), just as heat flows from high to low temperature regions. Equation 1 is sometimes known as Fick's first law.

The continuity equation describes the evolution of any flowing substance. For conserved substances, the continuity equation takes the form

$$\frac{\partial C}{\partial t} + \nabla \cdot \boldsymbol{J} = 0. \tag{2}$$

Equations 1 and 2 imply

$$\frac{\partial C}{\partial t} = D\nabla^2 C, \tag{3}$$

if D is spatially invariant. Equation 3 is known as the diffusion equation or Fick's 2nd Law.

For an initial condition of $C(\boldsymbol{r},0) = \delta(\boldsymbol{r}-\boldsymbol{r_0})$, the solution to Equation (3) for a boundary value problem is known as the diffusion propagator $P(\boldsymbol{r_0},0|\boldsymbol{r},t)$.[1] Given the propagator $P(\boldsymbol{r_0},0|\boldsymbol{r},t)$ for a certain boundary value problem, the solution to Equation (3) for any general initial distribution $C(\boldsymbol{r_0})$ is

$$C(\boldsymbol{r},t) = \int C(\boldsymbol{r_0})P(\boldsymbol{r_0},0|\boldsymbol{r},t)d^3\boldsymbol{r_0}. \tag{4}$$

One of the most important propagators in diffusion theory is the free space propagator, which is a solution to the diffusion equation when there are no boundary restrictions:

$$P_{free}(\boldsymbol{r_0},0|\boldsymbol{r},t) = (4\pi Dt)^{-d/2}\mathrm{exp}\left[\frac{-(\boldsymbol{r}-\boldsymbol{r_0})^2}{4Dt}\right], \tag{5}$$

where d is the dimension of space ($d = 1, 2, 3, 4...$).

This propagator is both isotropic and Gaussian about the point r_0, with a mean square displacement given by

$$\left\langle(\boldsymbol{r}-\boldsymbol{r_0})^2\right\rangle = 2dDt. \tag{6}$$

Finally, initial condition $C(\boldsymbol{r},0) = \delta(\boldsymbol{r}-\boldsymbol{r_0})$ could represent a single point particle instead of a group of particles. In this case, the propagator represents the probability of the point particle diffusing from $\boldsymbol{r_0}$ to $\boldsymbol{r}$ in time t.

2.2 Microscopic Diffusion

The macroscopic description of diffusion averages over the individual movements of the microscopic particles. For example, if the system is in equilibrium, then J = 0, and Equation

[1]It is also called the heat kernel, or the Green's function.

(2) implies that the concentration field remains fixed. However, experimentally, particles have been shown to move randomly even in thermal equilibrium [4]. A more interesting model of diffusion is the microscopic random walk model, which fills in the gaps left by the macroscopic model. Let us examine the microscopic model.

When a particle is suspended in a fluid medium, the atoms or molecules composing the fluid collide with the particle from different directions in unequal numbers during any given interval of time. The observed random motion of the particle is known as Brownian motion.[2] We can model the Brownian motion of a particle as a random walk.

Consider a Brownian particle moving randomly in some fluid. The net effect of many collisions during a timestep Δt is to displace the particle an amount δ in a random direction. Assume the particle is displaced with equal likelihood in a finite number of directions $2d$ uniformly distributed about space and that the displacements are statistically independent between timesteps. At some later timestep n, the net displacement will be

$$\Delta \boldsymbol{R_{net}} = \sum_{i=1}^{n} \Delta \boldsymbol{R_i}, \tag{7}$$

where $\Delta \boldsymbol{R_i}$ is the displacement of the particle on the i-th step. Now, $\Delta \boldsymbol{R_{net}}$ has the following properties

$$\begin{cases} \text{mean}[\Delta \boldsymbol{R_{net}}] = 0 \\ \text{var}[\Delta \boldsymbol{R_{net}}] = n\delta^2, \end{cases} \tag{8}$$

which follow from the fact that

[2]Robert Brown(1773-1858) observed the diffusion of pollen grains in 1827. While Brown's name is attached to the process, Jan IngenHousz (1730-1799) actually discovered it much earlier.

$$\begin{cases} \text{mean}[\Delta \boldsymbol{R_i}] = 0 \\ \text{var}[\Delta \boldsymbol{R_i}] = \delta^2. \end{cases} \tag{9}$$

Now, the duration of the n-step walk is t = nΔt, therefore

$$\text{var}[\boldsymbol{R_{net}}] = \frac{\delta^2}{\Delta t} t. \tag{10}$$

This random walk model should reproduce the diffusion equation. Indeed, if we want to bring in the macroscopic quantiy D into Equation (10), we will need to find a method of deriving it. To derive it, we need to bring back concentration fields.[3] At time $t + \Delta t$, the concentration at a point $\boldsymbol{r}$ is determined by the motion of all possible other Brownian particles that can reach $\boldsymbol{r}$ in time Δt (there are $2d$ of them). For each of these particles, the probability of them arriving at $\boldsymbol{r}$ is $1/2d$. Thus,

$$c(r, t + \Delta t) = \frac{1}{2d} \sum_{i=1}^{2d} c(r_i, t). \tag{11}$$

Subtracting c(r,t) from both sides and multiplying by a few constants, we get

$$\frac{c(r, t + \Delta t) - c(r, t)}{\Delta t} = \frac{\delta^2}{2d\Delta t} \sum_{i=1}^{2d} \frac{c(r_i, t) - c(r, t)}{\delta^2}. \tag{12}$$

Equation 12 looks like the discretized diffusion equation (see Equation (3)), and in the limit that $\Delta t \to 0$, we can make the identification:

$$D = \lim_{\Delta t \to 0} \frac{\delta^2}{2d\Delta t}. \tag{13}$$

Plugging Equation (13) into (10) gives us:

[3]This derivation roughly follows previous work, see [5].

$$\lim_{\Delta t \to 0} \mathrm{var}[\boldsymbol{R_{net}}] = 2dDt, \tag{14}$$

which matches equation 6 in the last section.

2.3 Diffusion Monte Carlo (DMC)

A Monte Carlo method is any method which uses random number sampling to estimate or determine a quantity. The method was invented by three scientists: Von Neuman, Ulam and Metropolis, who were using it to solve neutron transport problems at Los Alamos in the late 1940s [6]. The method has such broad application that almost every field of science, and subfield of science, uses the estimation technique. While Monte Carlo techniques can and have been used without computers, the method became very popular with the dawn of silicon chip. With the computer, it is possible to generate huge sequences of pseudorandom numbers in seconds. If this sequence of pseudorandom numbers has a very large period, it will be practically indistinguishable from a truly random sequence, at least to the software using it.

To simulate diffusion using Monte Carlo, we first select a random sample of the particles in some region, to describe the initial concentration field. The positions of these selected particles are then updated by generating a random step vector $\hat{\boldsymbol{U}}$, uniformly distributed in space with magnitude $\sqrt{2dD\Delta t}$, where d is the dimension and Δt is the duration of a time step. The stochastic update equation for each particle is thus

$$\boldsymbol{R}(t + \Delta t) = \boldsymbol{r}(t) + \sqrt{2dD\Delta t}\hat{\boldsymbol{U}}, \tag{15}$$

where $\boldsymbol{R}(t + \Delta t)$ is the updated position and $\boldsymbol{r}(t)$ is the old position. Note that we are using a convention where capitilized variables are random (stochastic) and lowercase variables are non-random. This follows the microscopic random walk model. Now, by the Central Limit Theorem,[4] a sum of independent random numbers with some mean and variance approaches a Gaussian random number with the same mean and variance as the sum becomes infinite. Therefore, we may also update the particle positions with the following equation:

$$\boldsymbol{R}(t + \Delta t) = \boldsymbol{r}(t) + \text{Gauss}(0, 2dD\Delta t)\hat{\boldsymbol{U}}, \tag{16}$$

where $\text{Gauss}(0, 2dD\Delta t)$ is a random number picked from the a Gaussian distribution with mean 0 and variance $2dD\Delta t$, and $\hat{\boldsymbol{U}}$ is a random direction vector uniformly distributed in n-dimensional space. This update equation is exactly what we would expect if we picked step vectors from the free diffusion propagator, the step is isotropic about $\boldsymbol{r}(t)$ (due to $\hat{\boldsymbol{U}}$) and Gaussian (due to $\text{Gauss}(0, 2dD\Delta t)$). This latter stochastic equation is known as a Wiener process and is heavily studied in mathematics [7]. Both Equations (15) and (16) can be used to update the particle positions in DMC. Their equivalence in simulations with large number of time steps is guaranteed by the Central Limit Theorem.

Now how do we implement these update equations in code? For simplicity, let us set the number of dimensions to 3, although the update equations are valid in any dimension. The fixed step length update (Equation (15)) is faster to compute than the Gaussian update (Equation (16)) and therefore is usually preferred.[5] To code a fixed step length update you can either generate three numbers with magnitude $\sqrt{2D\Delta t}$ and a random sign (either + or -) or you can pick a uniform point on a sphere of radius $\sqrt{6D\Delta t}$ using 2 uniform deviates (θ and ϕ). The first method, is a lot faster because no transcendental functions need to be computed. The Gaussian step updates are done in exactly the same manner as the fixed-length step updates, except the magnitude of the steps is Gaussian. See Figures 1 and 2 for code implementing these fixed-length update methods.

[4] For a short proof of the Central Limit Theorem, see [7].

[5] A Gaussian deviate is computed by generating at least one uniform deviate, therefore it is always slower.

```
//  RandomSign() = returns either a -1 or a 1
//  with equal probability
//  r = previous position
double stepLength = sqrt(2*D*dt);

//new position:
r += Vector3 ( stepLength*RandomSign(),
               stepLength*RandomSign(),
               stepLength*RandomSign() );
```

Figure 1: Code showing how to implement the fixed-step updates with a RandomSign() method. RandomSign() returns either a -1 or 1 with equal probability.

```
//  UniformDeviate(a,b) = uniform random number between a and b
//  acos = inverse cosine
double stepLength = sqrt(6*D*dt);

//pick random spherical coordinate angles
double phi = 2.0*PI*UniformDeviate(0,1);
double theta = acos(2.0*UniformDeviate(0,1) - 1);

//new position:
r += Vector3 (
        stepLength*sin(theta)*cos(phi),
        stepLength*sin(theta)*sin(phi),
        stepLength*cos(theta)
                        );
```

Figure 2: Code showing how to implement the fixed-step updates with sphere point picking. See Appendix A.6 for a derivation of this method.

The method of updating the steps shown in Figure 2 is known as “sphere point picking". See Appendix A.6 for a derivation of this approach.

2.4 Restricted Diffusion and DMC

It can be easily checked that the diffusion propagator for some general boundary condition can be expanded as

$$P(\boldsymbol{r_0}, t_0|\boldsymbol{r}, t) = \sum_n u_n^*(\boldsymbol{r}) u_n(\boldsymbol{r_0}) e^{D\lambda_n |t-t_0|}, \tag{17}$$

where $\{u_n\}$ are eigenfunctions of the Laplacian operator with eigenvalues $\{\lambda_n\}$. Since the Laplacian operator is Hermitian, the eigenfunctions are orthogonal and the eigenvalues are real. By the same token, the eigenfunctions are also complete:

$$\sum_n u_m^*(r) u_m(r') = \delta(r - r'). \tag{18}$$

Setting t $=$ t_0 in Equation (17) readily produces the initial condition given in Section 2.1: $C(\boldsymbol{r}, 0) = \delta(\boldsymbol{r} - \boldsymbol{r_0})$. Thus the diffusion equation can be solved for any boundary value problem (BVP) by solving the Laplacian eigenfunction equation for the same BVP. If we could find the eigenfunctions, for the BVP we are interested in, we could simulate the trajectories of the Monte Carlo particles by picking step vectors from the probability distribution given in Equation (17). However, the $\{u_n\}$ eigenfunctions are difficult to find for anything but the simplest geometries, so we need some other method.

Most regions are usually unrestricted on some scale, even close to the boundaries.[6] Therefore, we could update a particles position by using Equation (15) if the time step was made small enough such that the local region was unrestricted. But what if one of the updated particle positions crosses a boundary? Clearly we need to amend this step. The amendment will differ depending on what boundary condition we are imposing.

Let us consider first a reflecting, impermeable boundary surrounding a region Ω. In the

[6] A fractal might be a counterexample.

macroscopic model, this boundary condition is given by

$$\boldsymbol{J}(\boldsymbol{r}) \cdot \hat{\boldsymbol{n}} = 0 \qquad (\boldsymbol{r} \in \partial\Omega), \tag{19}$$

where $\partial\Omega$ denotes the boundary. The above equation states that the normal component of the net flux is zero at all points on the boundary. We can achieve this condition in a Monte Carlo simulation by mirror (specular) reflecting the particles off of the boundary. This method of reflection also has a nice interpretation. Compared to the walls of the bounding volume, the particles are massless and therefore should reflect like light. That is, they should specularly reflect off the walls.

The specular reflection operator, which mirror reflects any vector about a normal $\hat{\boldsymbol{n}}$, can be derived (see Appendix A.1) to be

$$\hat{\boldsymbol{R}} = \mathbf{1} - 2\hat{\boldsymbol{n}}\hat{\boldsymbol{n}}, \tag{20}$$

where $\mathbf{1}$ is the identity operator. The specular reflection operator is both coordinate-free and dimension-free (i.e. you could use it for a n-dimensional random walk) The Monte Carlo update equation for a reflecting impermeable enclosing volume is therefore

$$\boldsymbol{X}(t_{n+1}) = \begin{cases} \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W & : \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W \in \Omega \\ \boldsymbol{x}(t_n) + \alpha\boldsymbol{\Delta} W + (1-\alpha)\boldsymbol{R} \cdot \boldsymbol{\Delta} W & : \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W \notin \Omega, \end{cases} \tag{21}$$

where $\boldsymbol{\Delta} W$ is a random step vector generated by either the fixed-length method or the Gaussian method, and α is the relative distance from the initial point of $\boldsymbol{\Delta} W$ to the intersection point. Figure 3 defines these variables clearly.

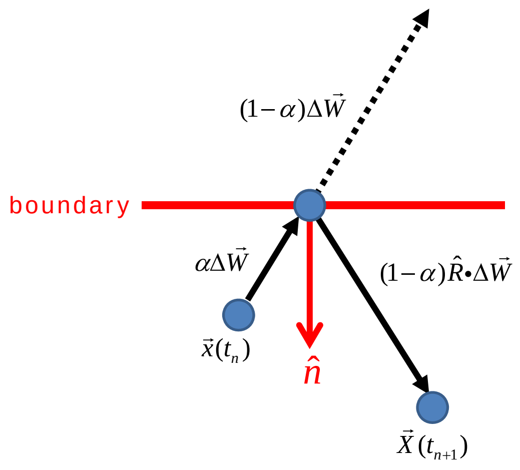


Figure 3: A figure showcasing the different quantities in Equation (21). If the step vector ΔW intersects the boundary, it is first broken up into two parts of size α and $1-\alpha$, where α is the relative distance to the boundary. To get the reflected vector, we compute $(1-\alpha)\hat{\boldsymbol{R}}\cdot\Delta W$.

If the boundary is permeable, a more general boundary condition can be written as

$$\boldsymbol{J}(\boldsymbol{r})\cdot\hat{\boldsymbol{n}} = P(C_+(\boldsymbol{r}) - C_-(\boldsymbol{r})) \quad (\boldsymbol{r}\in\partial\Omega), \tag{22}$$

where P is a constant which describes how permeable the membrane is, and $C_\pm$ are the concentrations on either side of the boundary. This boundary condition is sometimes called Fick's law of membranes [8]. If $P = 0$, Equation (22) reduces back to the reflecting condition of Equation (19). If P $= \infty$, then the boundary is absorbing. See Appendix A.2 for a derivation of Equation (22). The boundary update equation for this condition uses a combination of specular reflection and transmission probabilities; this is derived in Appendix A.3.

Finally, it can be proven in the case of a circle (or a cylinder) that no matter how small the timestep, there is always the possibility of multiple specular reflections within a single timestep (a proof is given in Appendix A.4). Thus, it is important to also take into account this effect. Handling multiple specular reflections is straightforward, we just keep checking for boundary intersections and updating with Equation (21) until no intersections occur. See the pseudo-code below for how you would implement this.

```
while ( boundaryIntersection ( particlePosition ) == true ){
// reflect particle
}
```

Figure 4: Pseudo-code which shows how to implement multiple specular reflection.

2.5 Nuclear Magnetization

In a fluid such as water, the diffusing molecules each have a net magnetic moment due to the spin-1/2 of the Hydrogen nuclei. These magnetic moments can be used to measure the diffusion coefficient. In the current section and the following sections, we will develop the theory of how this is done. Consider a set of N spin-1/2 particles in a magnetic field $\boldsymbol{B}$. The total magnetization M of this group of N particles is given by

$$\boldsymbol{M} = \sum_i \gamma \boldsymbol{S_i}, \tag{23}$$

where $\boldsymbol{S_i}$ is the spin operator of the i-th particle, and γ is a constant known as the gyromagnetic ratio. This equation implies

$$\langle \boldsymbol{M} \rangle = \gamma \langle \boldsymbol{S} \rangle , \tag{24}$$

where $\boldsymbol{S}$ is the total spin operator. Classically, the magnetic field applies a torque on the magnetic moments of the particles. Hence,

$$\frac{d\boldsymbol{S}}{dt} = \boldsymbol{M} \times \boldsymbol{B}. \tag{25}$$

Quantum mechanical averages obey classical equations through Ehrenfest's theorem. Therefore we would expect

$$\frac{d\boldsymbol{M}}{dt} = \gamma \boldsymbol{M} \times \boldsymbol{B}, \tag{26}$$

where we have dropped the $\langle\rangle$ for convenience. If the particles are placed in a strong static magnetic field $B_0\hat{\boldsymbol{z}}$, then after any small pertubative magnetic fields are turned off, the magnetization will return to a thermal equilibrium state aligned with the $B_0\hat{\boldsymbol{z}}$ field. This process is known as relaxation. It can be shown (see reference [9]) that the effects of relaxation in a static $B_0\hat{\boldsymbol{z}}$ field contribute to Equation (26) in the following way:

$$\frac{d\boldsymbol{M}}{dt} = \gamma \boldsymbol{M} \times \boldsymbol{B} - \frac{M_x\hat{\boldsymbol{x}} + M_y\hat{\boldsymbol{y}}}{T_2} - \frac{(M_z - M_0)\hat{\boldsymbol{z}}}{T_1}, \tag{27}$$

where T_1, and T_2 are relaxation time constants for the longitudinal and transverse directions respectively, M_0 is the equilibrium magnetization, and $\boldsymbol{M} = (M_x, M_y, M_z)$. This set of differential equations are known as the Bloch equations. The Bloch equations were extended by Torrey in 1956 to include diffusion [10]:

$$\frac{\partial \boldsymbol{M}}{\partial t} = D\nabla^2 \boldsymbol{M} + \gamma \boldsymbol{M} \times \boldsymbol{B} - \frac{M_x\hat{\boldsymbol{x}} + M_y\hat{\boldsymbol{y}}}{T_2} - \frac{(M_z - M_0)\hat{\boldsymbol{z}}}{T_1}. \tag{28}$$

This equation can now be solved to determine the evolution of the magnetization. However, if we plan on simulating the effects of diffusion using a Monte Carlo method, we can throw out the diffusion term in Equation (28). Why can we do this? Well if we jump into the frame of a group of particles moving together, this group is not diffusing and therefore its magnetization evolves by the Bloch equations. Despite this point, we will need Equation 28 in its full form to derive some useful analytical equations.

In magnetic resonance imaging (MRI), as we will discover in later sections, the transverse component of the magnetization is very important because it produces the signal we detect in a pickup coil. If we define the tranverse magnetization vector and magnetic field as complex numbers then

$$M_{xy} = M_x + iM_y \qquad B_{xy} = B_x + iB_y, \tag{29}$$

and the Bloch-Torrey equations reduce to a set of two equations

$$\begin{cases} \frac{dM_{xy}}{dt} = D\nabla^2 M_{xy} - i\gamma(M_{xy}B_z - M_zB_{xy}) - \frac{M_{xy}}{T_2} \\ \frac{dM_z}{dt} = D\nabla^2 M_z + i\frac{\gamma}{2}(M_{xy}B_{xy}^* - M_{xy}^*B_{xy}) - \frac{M_z - M_0}{T_1}. \end{cases} \tag{30}$$

In high-field Diffusion MRI, the only relevant magnetic field perturbations are in the $\hat{\boldsymbol{z}}$-direction, therefore $B_{xy} = 0$ and the above equations simplify to a set of uncoupled complex differential equations

$$\begin{cases} \frac{dM_{xy}}{dt} = D\nabla^2 M_{xy} - i\gamma(M_{xy}B_z) - \frac{M_{xy}}{T_2} \\ \frac{dM_z}{dt} = D\nabla^2 M_z - \frac{M_z - M_0}{T_1}. \end{cases} \tag{31}$$

Equation Set {32} will be important in the next section, where we discuss how high-field diffusion MRI pulse sequences can measure the diffusion coefficient.

2.6 High-field Diffusion MRI Pulse sequences

In MRI, a set of diffusing spin-1/2 nuclei are exposed to a static $B_0\hat{\boldsymbol{z}}$ field and a succession of magnetic field perturbations, known as pulses. The bulk magnetization produced by the diffusing spins rotates when exposed to magnetic field pulses and thus induces a current in a receiver coil by Faraday's law. If the receiver coil is stationed so that its axis is parellel with the xy plane, the received voltage, or signal S rather, will be proportional to the transverse magnitude of the bulk magnetization. That is, $S \propto |M_{xy}|$. In diffusion MRI, the goal is to sensitize this signal to the diffusive motion of the nuclei in such a way that the the diffusion coefficient can be calculated. Stejskal and Tanner showed, in 1965, that the signal could be sensitized using a high-field pulsed gradient spin echo (PGSE) sequence[11]. A PGSE sequence consists of four pulses which are shown in Figure 5.

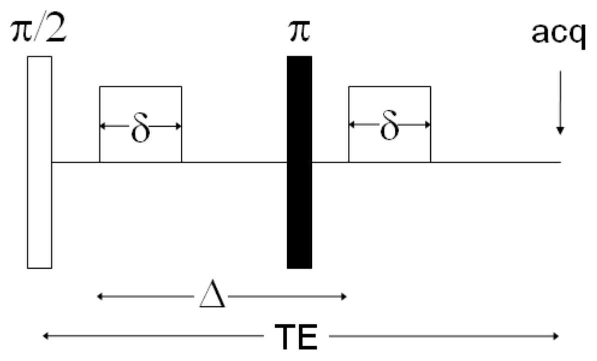


Figure 5: A diagram showing the four pulses in the PGSE sequence. The first pulse, labeled $\frac{\pi}{2}$ is called the 90° RF pulse and flips the bulk magnetization vector $M_0\hat{\boldsymbol{z}}$ to the tranverse plane. The second pulse is known as a gradient field and is applied for a duration δ. We will discuss this type of field later. The third pulse is labeled π and is known as a 180° RF pulse and it rotates the bulk magnetization vector 180 degrees about a direction defined by the 180° RF field. In the absence of inhomogeneities, the net effect of 180° RF pulse is to reverse the sign of the 2nd gradient field, which will be applied for the same duration as the first. Finally the signal is measured at time TE, which is known as the echo time. This signal should be sensitized to the diffusive motion of the particles. This figure is from Reference [8].

The first pulse in the PGSE sequence is known as a 90° radio-frequency (RF) pulse. This pulse rotates the bulk magnetization from its thermal equilibrium position $M_0\hat{\boldsymbol{z}}$ onto the xy-plane. The next three pulses consist of two identical pulses known as gradients, which we will discuss later, and a 180° RF pulse. In a homogeneous $B_0\hat{\boldsymbol{z}}$ field, which we will assume here for simplicity, the only effect of the 180° pulse is to reverse the sign on the last gradient field. Hence, the PGSE sequence effectively consists of only one 90° RF pulse and two gradient pulses of oppositive sign. After the four pulses in the PGSE sequence have been applied, we can measure the signal using a receiver coil. We will show that this signal is sensitized to the diffusive motion of the magnetized particles. To show this, we will need to discuss the most important pulse in the PGSE sequence, which is the gradient pulse. A gradient pulse is a spatially varying field of the form

$$\boldsymbol{B}_{grad} = (\boldsymbol{G}(t) \cdot \boldsymbol{r}(t))\hat{\boldsymbol{z}}, \tag{32}$$

where $\boldsymbol{G}(t)$ is a vector in the direction of the gradient field with a time varying strength and a constant direction. Substituting this field into Equation (31) gives us

$$\frac{dM_{xy}}{dt} = D\nabla^2 M_{xy} - i\gamma(M_{xy}(B_0 + \boldsymbol{G} \cdot \boldsymbol{r})) - \frac{M_{xy}}{T_2}. \tag{33}$$

The solution to this equation, given unrestricted (Gaussian) diffusion is

$$M_{xy} = \exp(-bD)\exp\left(-\frac{t}{T_2}\right)\exp\left(-i\gamma(\int_0^t \boldsymbol{G}(t') \cdot \boldsymbol{r}(t')dt' + B_0 t)\right), \tag{34}$$

where b is a constant known as the diffusion weighting and defined as

$$b = \int_0^t \left(\int_0^{t'} G(t'')dt''\right)^2 dt'. \tag{35}$$

By Equation (34), then,

$$S = S_0 e^{-bD}, \tag{36}$$

where S_0 is a reference signal measured with no diffusion gradients. Thus the signal is sensitized to the diffusive motion of the particles since the exponential decay factor is proportional to D. However, Equation (36) only holds if the diffusion is unrestricted. If the nuclei diffuse within some restricted region, then we might make the “Gaussian phase" approximation (see reference [12]):

$$S \approx S_0 e^{-bADC(t)}. \tag{37}$$

But now the measured “D" value is not the constant intrinsic diffusion coefficient D of the sample, but some other quantity, known as the apparent diffusion coefficient (ADC) which describes the effects of diffusion averaged over all the different compartments of the heterogeneous space. By making this approximation we can measure the effects of restricted diffusion in a sample through the signal, which could ultimately be used to probe tissue structure.

2.7 Simulating Diffusion MRI Pulse experiments (MRI-DMC)

Now we have enough background information to simulate the measurement of a MRI signal in our Monte Carlo diffusion simulations, which will now be abbreviated as MRI-DMC simulations. Let us take a step by step look at the MRI-DMC algorithm.

1. We first select positions for N particles in some diffusion region we are a simulating (i.e. a sphere). The positions for these N particles should be picked to accurately represent the initial concentration field we are interested in. The magnetization vectors for each

of these particles is initialized to some value $\boldsymbol{M}_0$.

2. The positions of these N particles are then updated by Equation (15) or (16).

3. If the updated position of any of the N particles intersects a boundary of the diffusion region, this position must be corrected. The type of correction we apply depends on which boundary condition we are simulating. If our diffusion region has impermeable, reflecting walls, then we correct the position by Equation (21). We must keep correcting the particles position until there are no intersections. Each successive correction will be smaller than the preceding one because the timestep is finite.

4. The magnetization vector of each particle is then updated by numerically solving the Bloch equations (Equation (27)) for some magnetic field $\boldsymbol{B}(\boldsymbol{r}, t)$ which changes in accordance to some diffusion MRI pulse sequence. We will discuss methods of numerically solving the Bloch equations later.

5. Repeat steps 2-4 until the desired number of timesteps is reached.

6. To simulate the measurement of the signal at the end of the simulation, we calculate $S = \left|\sum_i \boldsymbol{M_{xy}^i}\right|$, where $\boldsymbol{M_{xy}^i}$ is the tranverse magnetization vector of the i-th particle. If we want to calculate the ADC, we can now use Equation (37).

We can often simplify the computation of the magnetization (see Step 4 in the above list) by pre-solving the Bloch equations. For MRI-DMC simulations, there are three cases of the Bloch equations which are of interest: 1) T_1- and T_2-relaxation is negligible; 2) high field Diffusion gradient experiments; and 3) general field experiments with T_1- and T_2-relaxation. Let us consider each in turn.

1. If there is no T_1- or T_2-relaxation or the constants are so large that these terms are negligible, the Bloch equations reduce back down to equation 26. Now consider a static field $\boldsymbol{B} = B_0\hat{z}$. The solution to Equation (26) for this magnetic field is

$$\boldsymbol{M} = M_{0_{xy}} cos(\gamma B_0 t)\hat{\boldsymbol{x}} - M_{0_{xy}} sin(\gamma B_0 t)\hat{\boldsymbol{y}} + M_{0_z}\hat{\boldsymbol{z}}. \tag{38}$$

Thus, the magnetization rotates around the magnetic field unit vector $\hat{B} = \hat{\boldsymbol{z}}$. The angle of rotation evolves by the equation

$$\phi = -\gamma|\boldsymbol{B}|t. \tag{39}$$

Now, lets consider a static B-field which has an arbitrary direction vector. If we rotate our coordinate system so that it lies along the z-axis, then Equations (38) and (39) hold. Thus, in the unrotated coordinate system, the magnetization rotates around the $\hat{\boldsymbol{B}}$ unit vector with an angle of rotation which evolves by Equation (39). Now, if the field is time dependent, we can always break up time into small enough chunks such that the magnetic field is approximately static. Thus, to update the magnetization in a time-varying field, we break up time in chunks of size dt and then rotate the magnetization around the magnetic field unit vector $\hat{\boldsymbol{B}}$ by an angle $-\gamma|\boldsymbol{B}|dt$.

2. The second case is of great importance. As far as this author is aware, all available MRI-DMC simulation codes to date use the high-field approximation. For this approximation, one assumes that the static $B_0\hat{\boldsymbol{z}}$ is so big that gradient terms in the tranverse plane are negligible and take the form of Equation (32). We can thus use Equation (34) directly without the diffusion term, to simulate the effects of the pulses. A phase value is held in memory for each particle and is updated by

$$\phi(t_{n+1}) = \phi(t_n) - \gamma\boldsymbol{G}(t) \cdot \boldsymbol{r}(t)dt. \tag{40}$$

The phase factor involving B_0 in Equation (34) can be ignored because it adds the same phase to all particles and therefore does not cause any phase incoherence.

Since the transverse We also hold a variable in memory which tracks the time spent in different T_2 regions (divided by the corresponding T_2 constant), in case the geometry consists of multiple compartments with different T_2 parameters. At the end of the simulation, the final signal is computed as

$$S = \left| \sum_n e^{i\phi_n} e^{-\alpha_n} \right|, \tag{41}$$

where $\alpha_n = \sum \frac{\delta t_i}{T_{2_i}}$, is the sum of times spent in different T_2 regions divided by the corresponding T_2 value.

3. The third case is completely general and could arise if you want to simulate the effects of relaxation in a low-field regime or pulse sequence which involves more than just gradient fields. In this case, it is difficult to simplify matters and a Runge-Kutta integrator (or some other numerical solving technique) must be used to solve the Bloch equations (see Equation (27)). The simplest Runge-Kutta integrator is the Euler integrator. To find the magnetization vector of a particle on the $(n+1)$-step using Euler integration, we compute

$$\boldsymbol{M_{n+1}} = \left(\gamma \boldsymbol{M_n} \times \boldsymbol{B_n} + \boldsymbol{T_n}\right) \Delta t + \boldsymbol{M_n}, \tag{42}$$

where $\boldsymbol{M_n}$ and $\boldsymbol{B_n}$ are the magnetiation vector and magnetic field vector on the n-th timestep, Δt is the timestep, and $\boldsymbol{T_n}$ is the relaxation vector $\boldsymbol{T_n} = \left(\frac{-M_x}{T_2}, \frac{-M_y}{T_2}, \frac{M_0 - M_z}{T_2}\right)$ on the n-th timestep. Higher order Runge-Kutta integrators such as the 5th-order Dormand-Prince method are usually preferred over Euler integration.

3 The GPU and CUDA

3.1 Introduction

Modern single-core CPUs have hit a clock rate limit of 4 GHz. No one has been able to manufacture a faster single-core chip because of the enormous power consumption and heat generation involved in its operation. Instead, processor manufacturers such as Intel and AMD are producing chips with multiple processing cores (with reduced clock rates). Overall, this multi-core scheme produces a higher amount of FLOPS (floating point operations per second) if the software has been written to work with all available cores [13].

Modern computers also have graphics cards which interface between the CPU and the monitor to produce the colored display we see. The graphics card has a graphics processing unit (GPU) which calculates an enormous amount of pixel colors and pixel positions concurrently. Traditionally, the GPU has not been fully utilized to do numerical calculations[14]. At the very least, attempts to do so usually did not bring enough performance gain to compensate for the vast amount of time writing code with obscure languages such as OpenGL or DirectX[14].

In 2007, NVIDIA released graphics cards which could support NVIDIA software to do general purpose calculations with the GPU. They called this hardware-software model CUDA (Compute Unified Device Architecture).[7] Since 2007, NVIDIA has gone through three generations of CUDA graphics cards: Tesla (2007), Fermi (2010) and Kepler (2012).[8] The major trend between these three generations is an increasing core count with a decreasing core clock rate.

[7]Note that sometimes NVIDIA refers to just the software as CUDA, but techinically the name applies to both the hardware and software

[8]There are also subgenerations, albeit these are not named. Instead NVIDIA labels cards with a number they call the “compute capability". For example, a card with compute capability 3.5, is from the third generation (Kepler), hence the 3, but it has a few more features than a 3.0 Kepler card, which is of a different subgeneration.

3.2 CUDA Hardware Model

NVIDIA has three types of graphics cards available on the market: Geforce (for personal use), Quadro (for visualization) and Tesla (dedicated CUDA computation). All three card types can be used for CUDA computation (if they were made after 2007) and can be from any of the three generations.[9] An NVIDIA Graphics card will usually have anywhere between 2 to 16 multiprocessors (MP). Each MP has 8, 32, and 192 cores on Tesla, Fermi and Kepler generations respectively. Each MP also has a small amount of memory (on the order of kilobytes) in several different storage types, which have varying features and overhead. The MPs also have a few special function units (SFUs) which calculate transcendental functions and a few load/store units which calculate the memory addresses for retrieving and storing data. A schematic diagram of a Fermi generation MP design is shown in Figure 20 in Appendix C.

The main memory storage unit on the graphics card is Dynamic Random Access Memory (DRAM). DRAM has storage on the order of gigabytes. DRAM is off-chip and far away from the GPU. Unfortunately, it takes the GPU a long time to access DRAM. For example, the Tesla K20X (Kepler) card has a peak performance of 3.59 TFLOPS but a DRAM bandwidth of 208GB/s. Since each floating point number takes up 4 bytes of memory, the card, if bounded by the memory access rate, could only have 48 GFLOPs, which is 75 times less than its theoretical maximum performance. Now, most people want to do calculations on large amounts of data, which will most certainly be held in DRAM. How does NVIDIA get around this memory bandwidth problem? By oversaturating the GPU with tasks to do while its waiting for data.

Let us have a closer look. A single independent task is known as a **thread**. In MRI-DMC, the calculation of a single particles position and magnetization trajectory would represent a single independent task or thread. The CUDA model essentially allows a practically

[9]For example, there are Tesla cards from the Tesla, Fermi and Kepler generations.

unlimited number of these so-called threads to be stored for execution on the device. The large number of threads is partitioned into what are called **blocks**, these are groups of threads which have a programmer defined size.[10] At any time, up to eight of these blocks are sent to each of the MPs. The MP then partitions these eight blocks into groups of 32 threads known as **warps**. The warp size is hardware defined, and cannot be changed by the programmer. A warp of threads is then computed in parallel on the MP. If this warp requires data from DRAM, instead of waiting for the data to arrive to the MP, the MP will request the memory transaction and then switch to another warp which hasn't been handled. This instant switching between warps allows the MP to always keep busy and overcome the DRAM bandwidth bottleneck, thus obtaining the peak performance of the card.

3.3 CUDA Software Model

The CUDA software development kit (SDK) comes with a compiler which supports a C/C++ language that has been extended with some bells and whistles to communicate with their graphics cards. Languages other than C/C++ are supported as well, but they will not be discussed in this thesis. Futhermore, we will be describing and using only the **runtime** CUDA language extension of C/C++. Other CUDA language extensions exist which offer higher or lower levels of abstraction, such as the lower level driver extension, the assembly language PTX (Parallel Thread eXecution), and the higher level Thrust library, but these are less popular than the runtime extension and are outside the scope of this work. I should mention that all available APIs can be used inside the same code text file and compiled together. Finally, in all CUDA documentation the CPU is known as the **host** and the GPU as the **device**[15]. I will keep this convention from now on.

In the runtime language extension, functions that are called by the host and executed on the device are labeled with __global__ tags and called **kernels**. Functions which are called

[10]Despite the allowed variability in block size, there is always an optimal width for the blocks which produces the maximal program speed.

by a kernel and executed on the device are labeled with __device__ tags and are called **device functions**. Finally, if the function can be called by the host and the device , then it should have both __host__ and __device__ tags.

In the runtime API, memory is allocated on the device using the function cudaMalloc() and freed on the device using the function cudaFree() . Data are copied to the device using the function cudaMemcpy() and data are transferred back to the host from the device using the same function. Finally the kernel, the main GPU computation function, is called from the host with yourKernelName<<<blocks,threads>>>. All of this section is summarized in the simple code file listed in Appendix B.2. This code file adds two arrays on the device and sends the end result back to the host. The equivalent CPU code is also listed in Appendix B.2 so that you can see how close CUDA C/C++ is to regular C/C++.

3.4 Wrapping the Runtime

If the code you are trying to write is more complicated than adding two vectors, then the runtime function calls are annoying to write repetitively. Futhermore, as you call more and more CUDA Runtime functions and CUDA kernels, the code becomes messy and unreadable. To get around this, libraries such as Thrust wrap the runtime in a higher level, more readable framework. The problem with the Thrust design is that it does not support CUDA streams currently. A CUDA stream is a queue of commands, consisting of memory copies from the CPU to the GPU or GPU kernel calculations, that execute synchronously. By default, all CUDA commands are queued into only one stream: stream 0. If you have more than one stream, then commands between streams are asynchronous and can potentially be executed on the card simultaneously. This stream system is very beneficial if you are loading large sets of data to the card between kernel calls. Loading data from the CPU to the GPU can waste lots of time and it would be nice to load the data while calculations are being performed. This is possible with multiple CUDA streams. Furthermore, unless you have the same number of CPU cores as GPU cards, you cannot run CUDA code on multiple GPUs without streams, at least not in a manner beneficial for performance. Finally, the Thrust framework is very general and can slow down performance because their code was written to handle almost any possible end-user scenario.

Thus, the Thrust libraries cannot be used if we want to use CUDAstreams or multiple GPU systems. Instead I have implemented my own wrapper framework which supports streams and multiple GPUs. There are four wrapper classes in my library: cudaVector, cudaScalar, pinnedVector, and pinnedScalar. The cudaVector class wraps a pointer to an array in device memory . cudaMalloc() is called in the constructor (the initialization function for a class) of the cudaVector class to allocate the memory of the array and cudaFree() is called in the destructor (the function that frees the memory defined by the class) to free the memory of the device array. All cudaVector member functions support streams and multiple devices. cudaScalar is the same as cudaVector, but for arrays with a size of one – a single variable. This was done to limit the use of redundant higher structures when we are holding only a single value. Appendix B.2 showcases the use of my wrapper framework to add two vectors. The code is much simpler and more readable than the corresponding CPU AND runtime GPU codes. Finally, to use CUDA streams, you must store all the data you want transferred from the CPU to the GPU in **pinned memory**. Pinned memory is a type of CPU memory which is guaranteed not to be moved. The GPU needs this guarantee in order to execute memory transactions while simultaneously computing or doing other memory transactions. Hence, I have created a pinnedVector class which wraps a pointer to pinned CPU memory. Thrust has no support for pinned memory, which is one of the reasons it can't support streams.

4 MRI-DMC on the GPU

4.1 Waudby-Christodoulo GPU DMC Algorithm (WCDMC)

In 2011, Christopher Waudby and John Christodoulou proposed a design for high-field MRI-DMC simulations which utilizes the CUDA library to interface with NVIDIA GPUS [16], hereafter referred to as WCDMC. This was the first time that a high-field MRI-DMC algorithm had been brought to the GPU. Despite this achievement, there are many aspects of their design which we could improve on. The following list outlines some of the major limitations of the WCDMC design.

1. In the WCDMC design, the magnetic field applied to the diffusing particles must consist of only a static $B_0\hat{\boldsymbol{z}}$ field and a gradient field. These are the standard fields used in high-field Diffusion MRI sequences, but there is no need to limit the simulations to such a narrow scope.

2. In the WCDMC design, only gradient measurements with the same time profile can be executed on the GPU at a single instance. If multiple gradient measurements with different time profiles are required, the data have to be transferred to the GPU multiple times. This violates a golden rule expounded in the CUDA programmers guide and several books on CUDA [17]: "Get the data on the GPU and keep it there." Transferring data from the CPU to the GPU multiple times is expensive and should be avoided.

3. WCDMC does not have a framework which can support complex multi-compartment regions. In fact, WCDMC can only support single pore systems and even at that it is limited. If you want to add a new pore type in WCDMC, you have to copy and paste the code for an old pore type and slightly modify that file to desribe the new features of the pore. This is definitely not an elegant system. That being said, it was probably never the author‘s attention to make such a framework.

4. WCDMC does not support double precision. Waudby and Christodoulou were using a 1st generation Geforce card with very limited double precision capability. So supporting double precision was definitely not a priority.

5. WCDMC does not support multiple GPU systems or multiple streams. Stream support would have added a great boost to their design since they are loading big gradient profile arrays on to the device.

6. WCDMC does not support specular reflection. They claim that this boundary condition is not suited for the GPU hardware. Instead, they use a different boundary condition called "simple rejection" which is much more suited to the GPU hardware design (more on this later).

7. WCDMC initializes the particles uniformly about the single pore on the CPU. Waudby and Christodoulou did not mention why they did this, but I assume it was because they thought the initialization methods would not be suited for the GPU. However, it turns out that this is probably not true.

These are the seven major limitations of WCDMC. Despite these issues, the WCDMC template proposed a few solutions for problems one encounters when attempting to implement a MRI-DMC algorithm on the GPU. These solutions are of interest and will be discussed as well in the next sections.

4.2 The Trevor DMC Algorithm (TDMC)

A diffusion MRI experiment can be broken up into three independent parts: 1) the MRI apparatus, 2) a sample or geometry, and 3) the diffusing particles in that sample being traced. This threeway split gives us a guideline on how we might design code to simulate diffusion MRI. To simulate the applied fields, we will have four independent C++ classes. The first class will be a C++ functor, which returns the total magnetic field at any point in time and space. With functors, we can simulate any pulse sequence, we are no longer limited to the high-field gradient regime. For a clear example of a functor, see the high-field cosine gradient functor in appendix B.5. The next MRI class in my design is a container called Acquisition and it holds all of the field functors which have the same duration (time profile). The second last class is another container called AcquistionStream which holds all of the Acquisition class objects, thus allowing for storage of pulse sequence measurements with different time profiles. Finally, we have the Integrator class, which integrates the Bloch equations. This class, like the field functors, is user-specified and can represent any Runge-Kutta integrator, i.e. the traditional Runge Kutta 4th-order, the adaptive Dormand-Prince Runge Kutta, or even just a function that rotates the magnetization (which can be used when there is no relaxation). It has been shown that Dormand-Prince and low-storage Runge Kutta integrators can solve vector-valued differential equations up to 100 times faster than the CPU [18]. Since we are solving the Bloch equations for millions of particle threads, we are effectively solving a vector-valued differential equation and should receive similar speed ups.

To model the geometry of the sample we will have two C++ classes called Lattice and Basis. The Basis class will represent the objects you want to put on the tissue lattice, anything from a simple cylinder to a polygon mesh. The Lattice class will contain functions which enforce periodic boundary conditions and calculate properties pertaining to the bulk geometry. The Lattice class is redundant when all that needs to be modelled is a single pore structure (such as diffusion in a cylinder). In this case, we will just send the Basis class to the GPU and not the Lattice class. A typical basis-lattice geometry is shown in Figure 7. The class structure is summarized in Figure 6.

After the set of aquisitions have been defined and stored in an AcquisitionStream object, and the user has defined the geometry by a basis and lattice, we can then begin the Monte Carlo simulation. The AcquisitionStream class will contain a function which partitions the number of Acquisition objects to each available GPU. Figure 8 shows this partitioning scheme. These Acquisition objects will then be partitioned into two different CUDA streams along with the Basis, Lattice and any other data needed as shown in Figure 9. These queues will send data to and from the main computation kernel which updates the positions of the particles and calculates their magnetization vectors. The code for this function is shown in Appendix B.8.

This multi-class design is possible because CUDA allows classes (and templates) to be loaded and used on GPUs since version 3.0 of their software development kit. On the opposite end of the spectrum, WCDMC does not even use a single class, their design is based heavily on the C programming style where classes are not used.

Finally, since WCDMC only involves high-field MRI gradient sequences, it would be unfair if we were to compare the speed of TDMC, a completely general framework to the more specific WCDMC design. Thus I have split up the Acquisition class family into two sets: phaseAcquisition, which deals with high-field phase measurments and magAcquisition, which is the set of classes for solving the Bloch equations in the general case. The phaseAcquisition calculations can be entirely handled by the more general magAcquisition classes, but the magAcquisition classes will be slightly slower, a sacrifice well worth the generality. This being said, the phaseAcquisition and the magAcquisition classes are almost identical, so in Appendix B I may provide examples of just one to eliminate clutter. Different computation kernels were also made for the high-field Diffusion measurments, because for these you only need to update the phase using equation 40.

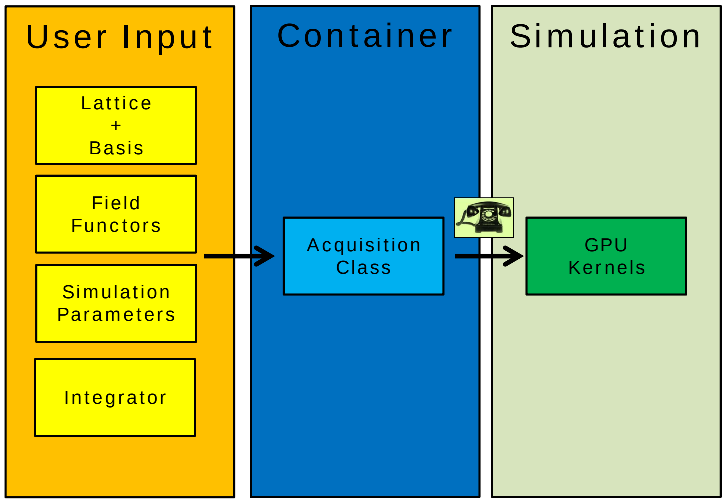


Figure 6: The C++ class structure of TDMC. The user input is stored in Acquisition classes, which then call the GPU kernels.

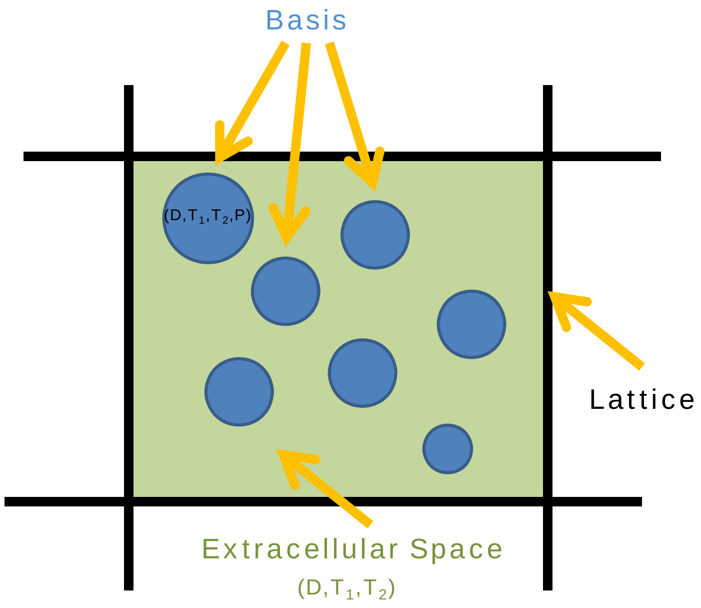


Figure 7: A typical geometry used in MRI-DMC simulations consists of a lattice and basis. Each basis object has four intrinsic properties: T_1, T_2, D and P. The empty lattice space can be used as surrounding fluid with its own parameters T_1, T_2 and D. If the basis represents cells, then this surrounding fluid is usually known as the extracellular space.

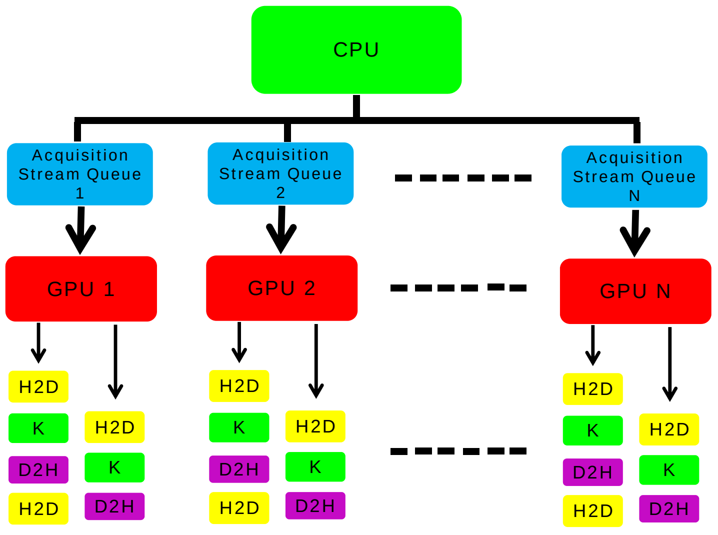

CPU
Acquisition Stream Queue 1
Acquisition Stream Queue 2
Acquisition Stream Queue N
GPU 1
GPU 2
GPU N
H2D
K
D2H
H2D
H2D
K
D2H

Figure 8: A single CPU thread gives a few of the Acquisition classes to each available GPU. These are then partitioned again onto two CUDA streams, which are executed simultaneously. H2D = Host to Device transfer. D2H = Device to Host transfer. K = kernel.

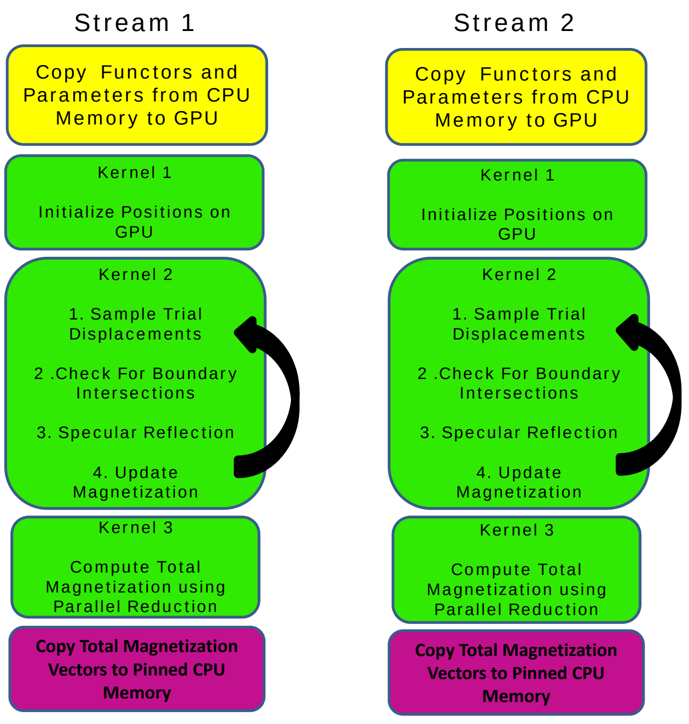

Stream 1
Stream 2
Copy Functors and Parameters from CPU Memory to GPU
Kernel 1
Initialize Positions on GPU
Kernel 2
1. Sample Trial Displacements
2 .Check For Boundary Intersections
3. Specular Reflection
4. Update Magnetization
Kernel 3
Compute Total Magnetization using Parallel Reduction
Copy Total Magnetization Vectors to Pinned CPU Memory

Figure 9: Each CUDA stream runs a queue of commands shown here. First the data are put on the GPU, then calculations are made, then the data are transferred back to the CPU.

4.3 Parallel Problem: Thread Divergence

The instruction scheduler on a NVIDIA graphics card MP is designed to give only one instruction at a time to its group of cores . These cores execute that same instruction in lockstep on different data. This is called a “Single Instruction Multiple Data" (SIMD) design. But what happens if these cores try to execute something like an if-else statement? Consider the following pseudo-code.

```
if ( data dependent condition ){
        // do something
}
else{
        // do something else
}
```

Figure 10: Pseudo-code showing a data dependent if-else statement. This if-branch and the else-branch represent two different instruction paths.

This if-else statement represents two distinct paths of instructions. Let us say the data for just one of the 32 cores on a Fermi MP satisfies the data dependent condition. The MP will execute the if-branch for that one core and hold the other 31 cores idle. After the if-branch has been executed, it will then move on to the else-branch. It will execute the else-branch for the remaining 31 cores, holding the other core idle. This path divergence represents a major bottleneck and could decrease the computation speed dramatically. On the CPU, a major chunk of the chip is devoted to “control flow" prediction elements which predict the instruction paths a thread might go into, eliminating the problem of divergence. NVIDIA decided to maximize the number of cores on the GPU chip and limit everything else, so their GPU chips have almost no control flow prediction.

Why is this a problem for us? Well, Waudby and Christodoulou claimed that the multiple specular reflection boundary condition would cause major divergence. Indeed, if we examine code listing **??** , we notice a data dependent while loop! This could cause a major slowdown.[11] Waudby and Christodoulou proposed a solution for this problem. Instead of multiple specular reflection, they use a different boundary condition known as simple rejection. The update equation for simple rejection is

$$\boldsymbol{X}(t_{n+1}) = \begin{cases} \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W & : \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W \in \Omega \\ \boldsymbol{x}(t_n) & : \boldsymbol{x}(t_n) + \boldsymbol{\Delta} W \notin \Omega. \end{cases} \tag{43}$$

The code for this update equation consists of a small data dependent if-statement which will lead to no divergence. How could this be? Well if the data dependent if-statement is small (under 7 lines of code in each branch approximately) the NVIDIA compiler will optimize it out, by forcing the cores through all branches even if the calculation is useless.

For a single pore system (the only system WCDMC can handle), the divergence due to multiple specular reflection isn't much of a problem. At small timesteps, multiple reflections are less likely to occur. Even in the case of a very small pore with a large timestep, where you would expect maximal divergence, there will be very little. Since in that case, almost every thread will reflect off the pore walls and hence take a similar instruction path. To test the levels of divergence in a single pore structure, I ran two tests on the CPU and the GPU with large ($\frac{1}{10}$ of the cylinder radius) and small ($\frac{1}{100}$ of the cylinder radius) step lengths. These simulations only updated the particle position, not their magnetization. Tests were run on a HP Z240 Workstation whose CPU and GPU specifics are given in the next section. Figure 11 shows the results. The ratio of computation time between using simple rejection and specular reflection on the CPU is roughly the same as that of the GPU, even when the step length is very big and multiple intersections are likely to occur. We would expect this result only if the divergence was negligible.

[11]Data dependent while loops and for loops also cause divergence. If the loop or if statement is not data dependent then there is no path divergence and hence no bottleneck.

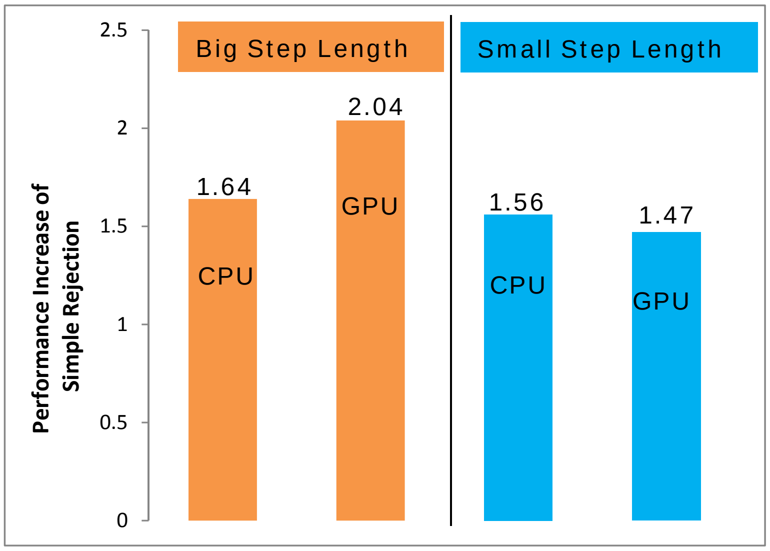


Figure 11: A bar graph showing the performance increase of simple rejection updates over specular reflection updates on the CPU and GPU for two different step lengths. When the step length is big (in this case $\frac{1}{10}$ of the pore size), we expect many collisions and maximum divergence. But this graph shows that the ratio on the GPU (2.04) is roughly the same as on the CPU (1.64), therefore divergence doesn't seem to be having a such a great impact on the performance as Waudby and Christodoulou claimed. In the case of small step lengths, we expect negligible divergence. In this case the ratio on the GPU (1.47) is very close to the CPU ratio (1.56), which is what we would expect if divergence was negligible.

So, at least for single pore systems, divergence is not an issue. For complex tissue models, the divergence due to multiple reflections would probably be more apparent. While simple rejection updates are a lot faster than specular reflection updates, they obviously do not satisfy the reflecting boundary condition (19). However, Szymczak and Ladd [19] found that simple rejection is better than some of the other alternatives to specular reflection. They found that simple rejection does preserve an initial uniform concentration of particles in one dimension, but it fails to impose the reflecting boundary condition correctly for a non-uniform initial distribution of particles in one dimension. In both cases, only specular reflection preserves the boundary condition. Thus, to maximize options for speed and accuracy, both boundary intersection update schemes should be present in any DMC design. The simple rejection scheme can then be used to speed up the simulations if it first validated by the specular reflection output. Lastly, attempts were made by the author to eliminate the divergence in the multiple reflection update method. See Appendix B.9 for a novel method that uses “warp voting" to gain a speed up of 1% to 2% over the normal multiple reflection update method.

4.4 Benchmarking

The entire point of writing MRI-DMC algorithms for the GPU is to speed up simulation time over the CPU. Hence, it is important to see how much faster TDMC is on the GPU versus a single-threaded CPU. It is also interesting to compare the speed of TDMC to WCDMC. Waudby and Christodoulou provide their code freely [20]. However, a few issues arise if we want to compare their raw code to TDMC. These are the following:

1. WCDMC only supports floating point. So this eliminates any double precision comparisions.
2. WCDMC uses a less general CPU interface and this might bias a GPU timing comparison.
3. WCDMC has an option to generate random gradient directions for each spin. While this option can be turned off in their code, the generation of random matrices is still present in the code and will slow it down compared to my code, which does not currently have this option.

To get around these problems I have implemented the WCDMC algorithm into my CPU framework. The only changes to their algorithm are:

1. No Kahan summation. Kahan summation in WCDMC is used to decrease the rounding error in floating point summation. Kahan summation keeps track of a compensation variable, which the algorithm later adds to minimize the rounding error. If we replace the Kahan summation with normal summation, the WCDMC algorithm will only run faster.

2. The random gradient matrix generation was taken out of their code. This would only slow it down anyway.

By reimplementing their code into my CPU framework, we will no longer be comparing my more general CPU interface to their interface, but just the CPU-GPU interaction and the GPU code. Furthermore, we can make double precision comparisons as well as single precision comparisons.

The CPU code used for testing follows the TDMC GPU algorithm exactly. The only difference between the CPU and GPU algorithm is the random number generator used. On the CPU side we use the C library built-in rand() function. While this function is generally considered not trustworthy due to its small period, it is fast and we want to give the CPU every possible advantage in terms of speed. On the GPU side we use the CUDA Curand GPU library, which implements a large number of different random number generators rigorously tested by the TestU01 generator testing library [21]. By default, the Curand library uses the XORWOW random number generator, which is fast and has a period of 2^{190}. We will use the XORWOW generator in all tests using the GPU. WCDMC uses their own builtin XORSHIFT generator, which will most likely be slower than the NVIDIA implemented Curand library.

For the timing comparison, I used a high-field cosine gradient sequence to generate 100 signals and a sphere as a single-pore diffusion region. Since the WCDMC algorithm uses a simple rejection boundary condition, I used that boundary update method in my code as well for the test. The test was run on a HP Z240 workstation containing a Intelő Xeonő

Processor E5-1650 6-core 3.20GHz CPU. The HP Z240 workstation contained two graphics cards, a Tesla C2075 (Fermi 2.0) graphics card for dedicated CUDA computation and a Quadro 600 (Fermi 2.1) graphics card handling the display. For the CPU computation only a single core was used.

The results of this single test are shown in figures 12 and 13.

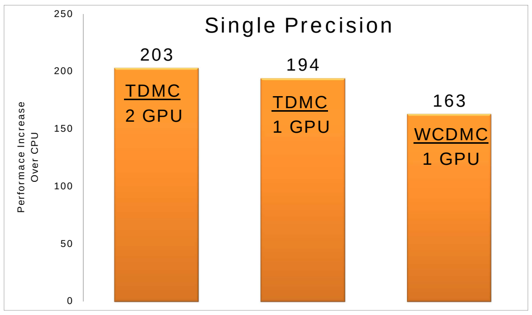


Figure 12: The results illustrated in this figure show the performance of my algorithm on 2 GPUs (Tesla C2075,Quadro 600) and 1 GPU (Tesla C2075) when computation is done with single precision. The numerical values represent the performance increase factor over the CPU. We can see that TDMC is roughly 1.25 times faster than WCDMC in the single precision case.

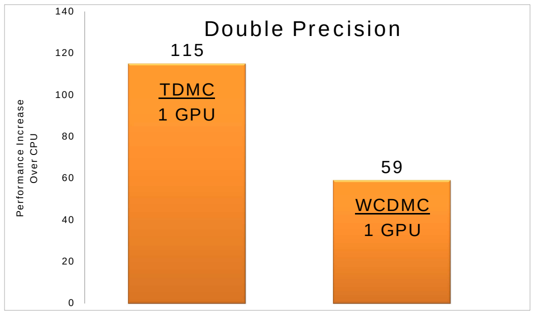


Figure 13: The results illustrated in this figure show the performance of my algorithm on 1 GPU (Tesla C2075) when computation is done with double precision. The numerical values represent the performance increase factor over the CPU. We can see that TDMC is roughly 2 times faster than WCDMC in this case. For these tests, using the extra Quadro 600 card produced only a 1% to 2% speed up.

We first notice that the WCDMC is only tested on 1 GPU, this is because in its raw form, the design only allowed for computation on one GPU, while TDMC utilizes any number of GPUs. The second thing we notice is that we are seeing performance increases of 100-200 times the CPU speed, this definitely shows why these algorithms should be ported to the GPU. We also notice that TDMC is faster by a factor of 1.25 in the case of single precision and a whopping factor of 2 in the case of double precision. Lastly, the sphere point picking update method was used in TDMC. WCDMC uses Gaussian updates. So we could have seen an even greater performance speed up over WCDMC and the CPU by using the RandomSign method of step updating.

5 Results

5.1 Testing TDMC against Analytical Solutions

Analytical calculations of the NMR signal for diffusion in a sphere, cylinder and parallel planes have been done in the literature [1]. Other systems have also been investigated, but these are the three standard analytical examples [22]. For each of the three cases, the eigenfunctions of the Laplacian operator can be found exactly and the diffusion propagator determined (see [5] for example). The more difficult part is deriving what the measured signal will be for a MRI sequence applied to a diffusing system described by the propagator. One of the most powerful approaches was introduced by Stepisnik in 1981 [23]. Stepisnik used a density matrix calculation to derive the signal attenuation for a high-field PGSE sequence with no asumptions about the geometry or the gradient shape. The derivation in Appendix A.5 reveals that the signal attenuation at the signal measurement time (2τ) for a high-field PGSE sequence with arbitrary gradient shape is:

$$\begin{aligned} S(2\tau) &= exp(-\beta(2\tau)) \\ &= exp\left\{-\frac{\gamma^2}{2}\int_0^{2\tau} dt_1 \int_0^{2\tau} dt_2 \boldsymbol{g}(t_1)\cdot\langle \boldsymbol{r}(t_1)\boldsymbol{r}(t_2)\rangle\cdot\boldsymbol{g}(t_2)\right\}, \end{aligned} \tag{44}$$

where $\boldsymbol{g}(t)$ is the gradient field, $\boldsymbol{r}(t)$ is the position vector of a diffusing spin and $\langle\rangle$ denotes the ensemble average.

Substituting in a general diffusion propagator $P(\boldsymbol{r}_1, t_1|\boldsymbol{r}_2, t_2)$ we get

$$\beta(2\tau) = \frac{\gamma^2}{2}\int_0^{2\tau} dt_1 \int_0^{2\tau} \int_V d\boldsymbol{r}_1 \int_V d\boldsymbol{r}_2 \rho(\boldsymbol{r}_1, t_1) P(\boldsymbol{r}_1, t_1|\boldsymbol{r}_2, t_2) \boldsymbol{r}_1 \cdot \boldsymbol{g}(t_1) \boldsymbol{r}_2 \cdot \boldsymbol{g}(t_2). \quad (45)$$

If we assume a uniform initial density ρ of the diffusing spins, then the above equation splits nicely into a spatial part and a temporal part when we insert Equation 17:

$$\beta(2\tau) = \frac{\gamma^2}{2}\sum_n B_n \int_0^{2\tau} dt_1 \int_0^{2\tau} dt_2 \exp(-\lambda_n D|t_2 - t_1|) g(t_1) g(t_2), \quad (46)$$

where B_n is the spatial part given by

$$B_n = \int_V d\boldsymbol{r}_1 \int_V d\boldsymbol{r}_2 (\hat{\boldsymbol{g}} \cdot \boldsymbol{r}_1)(\hat{\boldsymbol{g}} \cdot \boldsymbol{r}_2) u_n(\boldsymbol{r}_1) u_n(\boldsymbol{r}_2). \quad (47)$$

Now, all we have to do is calculate this integral for some gradient shape g(t) that we are interested in. If we can calculate this integral for a oscillating cosine gradient without any restrictions on the frequency ω, we can get two solutions for the price of one, because a rectangular pulse sequence is just a cosine sequence with $\omega = 0$.

For a cosine gradient sequence defined by:

$$\boldsymbol{g}(t) = \begin{cases} Gcos(\omega t) : 0 < t < \sigma \\ -Gcos(\omega(t-\tau)) : \tau < t < \sigma + \tau \\ 0 : \text{otherwise} \end{cases} \quad (48)$$

the attenuation factor $\beta(2\tau)$ in Equation (46) was calculated by Xu *et al* [24] for a gradient in the $\hat{\boldsymbol{x}}$ direction to be

$$\beta(2\tau) = 2(\gamma g)^2 \sum_n \frac{B_n \lambda_n^2}{(\lambda_n^2 + \omega^2)^2} \left\{ \frac{\lambda_n^2 + \omega^2}{\lambda_n} \left[\frac{\sigma}{2} + \frac{\sin(2\omega\sigma)}{4\omega} \right] - 1 + \exp(-\lambda_n \sigma) \right. \quad (49)$$
$$\left. + \exp(-\lambda_n \tau)(1 - \cosh(\lambda_n \sigma)) \right\}.$$

Using Equation (37), the ADC is then,

$$ADC = \frac{\beta(2\tau)}{b}. \tag{50}$$

The $\{B_n\}$ and $\{\lambda_n\}$ values for a cylinder, sphere and parallel planes are listed in [25]. These analytical solutions were compared to the Monte Carlo simulations for four different diameters with cosine gradient frequencies between 0 and 10 khz. Each simulation used 114688 particles and 3000 timesteps. The results are shown in Figure 14.

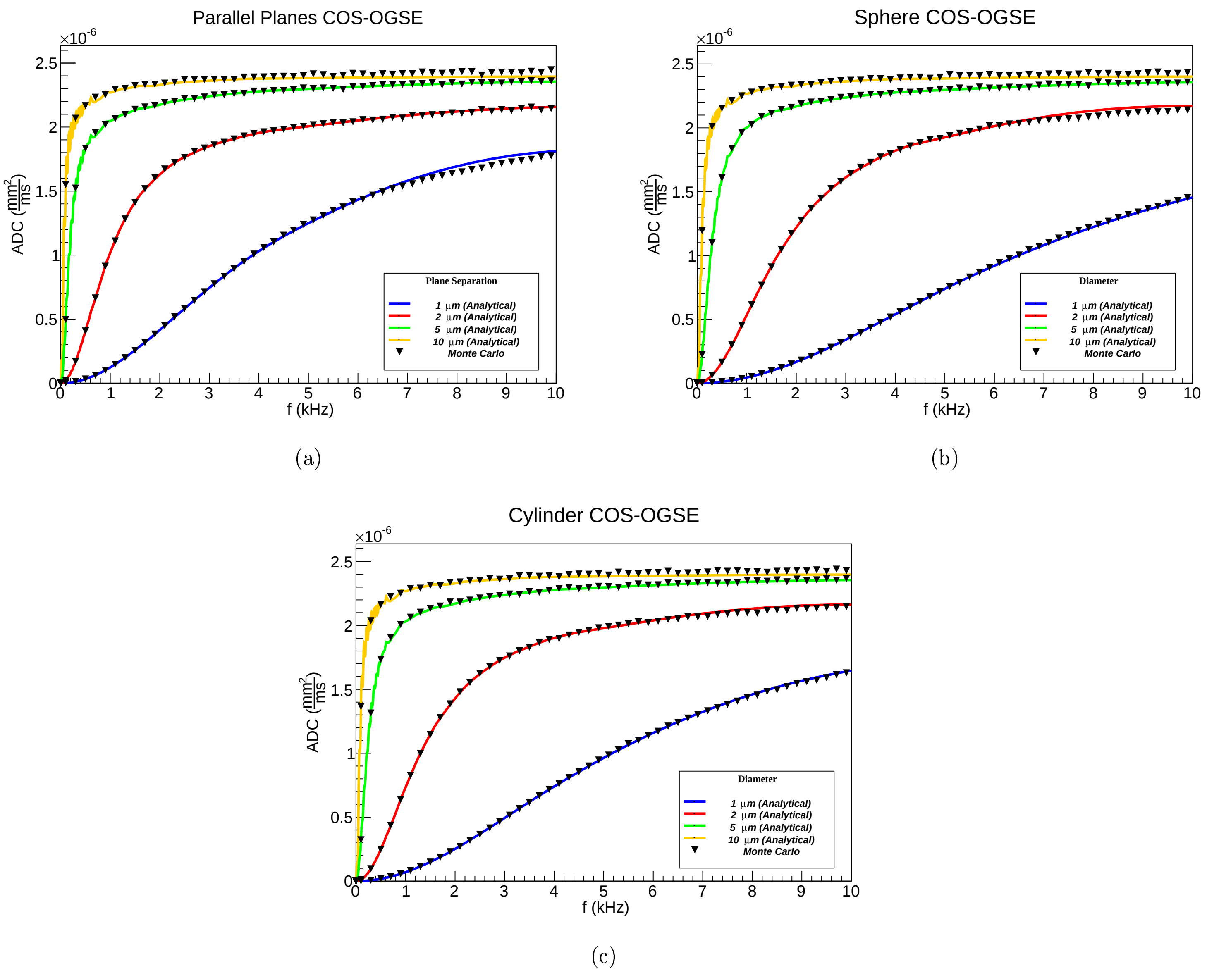


Figure 14: The TDMC and Analytical output for a cosine gradient sequence with diffusion between parallel planes (a), in a sphere (b), and in a cylinder (c).

The results displayed in Figure 14 show a close match between the TDMC results and the analytical equations.

5.2 Investigations with TDMC: Low-field MRI Pulse Sequences

The TDMC design allows any general pulse sequence to be modelled, not just the standard high-field pulse sequences which have already been investigated, see [26] for example. Consider a high-field gradient in the y-direction

$$B_{grad} = Gy\hat{\boldsymbol{z}}, \tag{51}$$

This trivially satisfies the Maxwell equation $\nabla \cdot \boldsymbol{B} = 0$ but not $\nabla \times \boldsymbol{B} = 0$. Thus this field is not actually possible to obtain in reality. It so happens that it is valid to a good approximation when the $B_0\hat{\boldsymbol{z}}$ field is very strong, the so-called high-field limit. To see this, consider a gradient field in the y-direction which satisfies Maxwell's equations:

$$\boldsymbol{B} = yG\hat{\boldsymbol{z}} + zG\hat{\boldsymbol{y}}. \tag{52}$$

The $\hat{\boldsymbol{y}}$ term is known as a concomitant gradient field [27]. The magnitude of the total field is

$$|\boldsymbol{B}| = \sqrt{(B_0 + yG)^2 + z^2G^2}, \tag{53}$$

and the spins rotate at an angular frequency

$$\omega = \gamma|\boldsymbol{B}| = \gamma\left[B_0 + Gy + \frac{G^2z^2}{2B_0} + O\left(\frac{G^3}{B_0^2}\right)\right]. \tag{54}$$

In the high-field limit, $B_0 \gg |zG|, |yG|$, Equation (54) reduces to $\omega = \gamma(B_0 + yG)$, consistent with Equation (32). Without the high-field approximation, analytical derivations of the MRI signal are impossible for gradient sequences. Therefore, we must derive the MRI signal numerically. Unlike previous designs of MRI-DMC simulations [28, 26, 16], only TDMC can support fully general pulses such as low-field concomitant gradients. To apply a low-field pulse sequence in TDMC, you just create a functor representing the sequence. See Appendix B.5 for an example of a low-field gradient echo sequence in the y-direction. We can use this functor to study some interesting low-field cases.

Consider a bunch of molecules in a cylindrical container. What happens if we set D=0, i.e. they don't move?. In this case we would expect no signal attenuation in the high-field limit where $B \gg |Gr|$ (Set $D = 0$ in Equation (36)). However, in the low-field case, we can see from the simulation results in Figure 15 that the concomitant gradient spoils this nice result when B_0 is small. This is because of the G^2 term in Equation (54) which prevents a complete rephasing when the gradient switches signs in the pulse sequence (G^2 stays the same even if $G \rightarrow -G$).

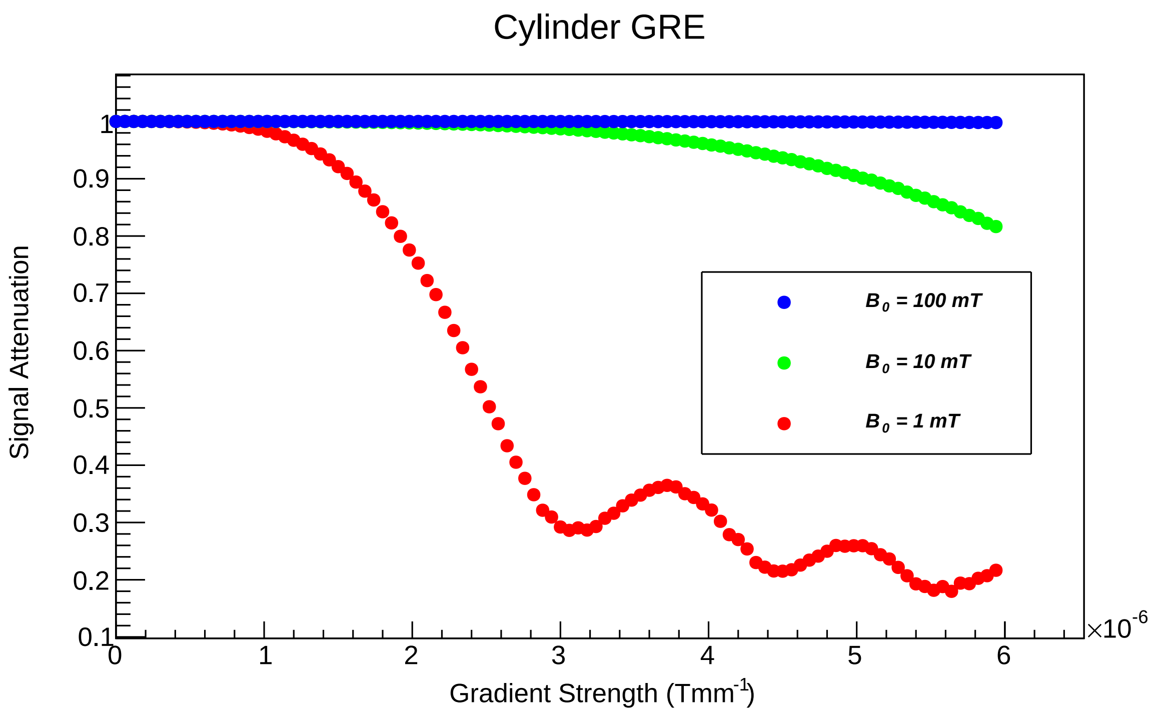


Figure 15: Diffusion in a large cylinder with $D = 0$. The signal attenuation was measured using a low-field rectangular gradient echo sequence. There is no signal attenuation in the high-field limit (B_0 large). The signal attenuation when B_0 is small compared to G is due to incomplete rephasing because of the G^2 term in Equation (54).

Now let's consider molecules diffusing in a sphere of cellular size (radius $= 2.5\ \mu m$) with $D = 2.5\frac{\mu m^2}{ms}$. In this case we expect as B_0 increases in magnitude that we will regain the high-field analytical ADC for diffusion in a sphere. This is exactly what the TDMC simulation results show in figure 16. Furthermore, this graph can be split into two regions, unrestricted diffusion and restricted diffusion. The unrestricted regime occurs when the diffusing particles have not, on average, reached the boundary of the sphere. This means that the gradient duration δ is less than $\frac{r^2}{6D} \approx 4.2ms$, where r is the radius of the sphere. The restricted diffusion regime occurs when δ is greater than $\frac{r^2}{6D} \approx 4.2ms$. The demarcation line between these two regions is shown in yellow in Figure 16.

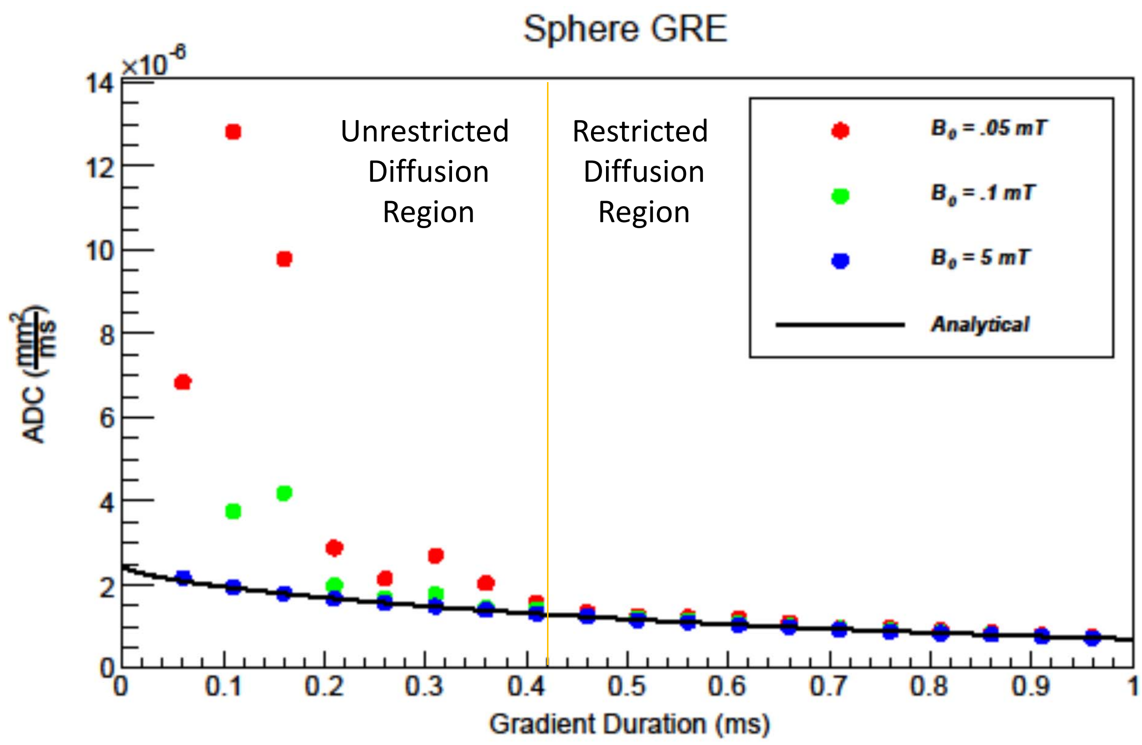


Figure 16: Diffusion in a sphere of cellular size with $D = 2.5\frac{\mu m^2}{ms}$. The ADC was measured using a low-field rectangular gradient echo sequence. In the high-field limit, the TDMC output matches the analytical high-field solution as expected. This graph can be separated into two different regions, unrestricted and restricted diffusion. The demarcation line (yellow) is at a gradient duration $\frac{r^2}{6D} \approx 4.2ms$.

We used the analytical high-field signal attenuation equation derived by Neuman [1] in Figure 16. However I found that the Stepisnik equation produces the exact same analytical curve, so it doesn't matter which analytical framework we use here.

6 Conclusion and Future Work

A novel design was presented in this thesis for Monte Carlo Diffusion MRI (MRI-DMC) simulations with general pulse sequences and general diffusion regions on a multiple NVIDIA-GPU system. This design was shown to be more flexible and faster than a previous design by Waudby and Christodoulou, with a speedup of almost a factor of two when using double precision and by a factor of 1.25 when using single precision. Furthermore, we also showed that MRI-DMC simulations on the GPU could obtain a performance increase over a single-threaded CPU by up to 200 times, which is definitely noteworthy.

To test the accuracy of the code, the simulation output was compared against the three standard analytical solutions for a sphere,cylinder and parallel planes. The analytical solutions matched the simulation output very closely. The problem of thread divergence on the GPU was discussed and determined to be not a serious problem for single pore MRI-DMC simulations. In simulations with more complicated geometries, we would expect that the divergence would affect performance in a more substantial way. However, the GPU code would probably still be faster than the CPU code even in this case. Finally, the simulations were used to study low-field gradient pulse sequences, where concomitant gradient terms make analytical solutions impossible to obtain. As far as the author is aware, this is the first time investigations into the low-field regime have been done with MRI-DMC simulations.

There are always ways in which we could have made the design better. One obvious point of improvement is the use of the multiple CPU cores available on most modern computers. Since we were working on a 2-GPU computer, we would see almost no benefit in using the 6 available cores on the CPU. But with (n>2)-GPU computers, it would be faster to handle each GPU on a separate CPU thread. Furthermore, no MRI-DMC on multiple-core CPUs have been performed and it would be very interesting to see how they compare to GPU codes. Another point of improvement would be with the double precision code. Clearly GPUs are better at floating point calculations, since they have been designed only to use this precision in the past. However, with the dawn of scientific GPU computing, the idea of mixed-precision code is becoming popular. It would be interesting to test a well-designed mixed precision MRI-DMC code against the floating point Kahan summation algorithm Waudby and Christodoulou used. While there are many more points of improvement, one last note should be made of testing in general diffusion regions. TDMC has the capability of simulating diffusion in very complex multi-compartment systems with differing T_1,T_2,D parameters, but none of this was rigorously tested. In future work, the simulation output for these more complicated regions could be compared to the available asymptotic analytical solutions. Lastly, there is no limit to the number of different investigations we could do in the low-field regime. Before this thesis, this regime had not been investigated with MRI-DMC simulations. This is definitely a major point for future work.

7 References

[1] C. Neuman, "Spin echo of spins diffusing in a bounded medium," The Journal of Chemical Physics, vol. 60, p. 4508, 1974.

[2] I. Millington, Game Physics Engine Development: How to Build a Robust Commercial-Grade Physics Engine for Your Game.. Morgan Kaufmann, Morgan Kaufmann Publishers/Elsevier, 2010.

[3] J. Crank, The Mathematics Diffusion. Oxford Science Publications, [Eng] Clarendon Press, 1979.

[4] H. Johansen-Berg and T. Behrens, Diffusion MRI: From quantitative measurement to in-vivo neuroanatomy. Elsevier Science, 2009.

[5] D. S. Grebenkov, "Laplacian eigenfunctions in nmr. i. a numerical tool," Concepts in Magnetic Resonance Part A, vol. 32, no. 4, pp. 277–301, 2008.

[6] R. Shonkwiler and F. Mendivil, Explorations in Monte Carlo Methods. Undergraduate Texts in Mathematics, Springer, 2009.

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

[8] D. S. Tuch, Diffusion MRI of complex tissue structure. PhD thesis, Citeseer, 2002.

[9] D. Ter Haar, "Simple derivation of the bloch equation," American Journal of Physics, vol. 34, p. 1164, 1966.

[10] H. C. Torrey, "Bloch equations with diffusion terms," Physical Review, vol. 104, no. 3, p. 563, 1956.

[11] E. Stejskal and J. Tanner, "Spin diffusion measurements: spin echoes in the presence of a time-dependent field gradient," The journal of chemical physics, vol. 42, no. 1, p. 288,

1965.

[12] P. Professor and C. Director of MRI Cardiff University Brain Research Imaging Centre Derek K. Jones, Diffusion MRI : Theory, Methods, and Applications: Theory, Methods, and Applications. Oxford University Press, USA, 2010.

[13] S. Cook, CUDA Programming: A Developer's Guide to Parallel Computing with GPUs. Applications of GPU Computing Series, Elsevier Science, 2012.

[14] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Pearson Education, 2010.

[15] NVIDIA, NVIDIA CUDA Programming Guide 4.2. 2012.

[16] 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.

[17] R. Farber, CUDA Application Design and Development. Applications of GPU computing series, Elsevier Science, 2011.

[18] L. Murray, "GPU acceleration of runge-kutta integrators," Parallel and Distributed Systems, IEEE Transactions on, vol. 23, no. 1, pp. 94–101, 2012.

[19] P. Szymczak and A. Ladd, "Boundary conditions for stochastic solutions of the convection-diffusion equation," Physical review E, vol. 68, no. 3, p. 036704, 2003.

[20] "WCDMC CUDA code." `http://www.smb.ucl.ac.uk/christodoulou/software/`.

[21] P. L'Ecuyer and R. Simard, "Testu01: A software library in ansi c for empirical testing of random number generators," 2007.

[22] R. Thambynayagam, The Diffusion Handbook: Applied Solutions for Engineers. McGraw-Hill Education, 2011.

[23] J. Stepišnik, "Analysis of nmr self-diffusion measurements by a density matrix calculation," Physica B+ C, vol. 104, no. 3, pp. 350–364, 1981.

[24] J. Xu, M. D. Does, and J. C. Gore, "Quantitative characterization of tissue microstructure with temporal diffusion spectroscopy," Journal of Magnetic Resonance, vol. 200, no. 2, pp. 189–197, 2009.

[25] J. C. Gore, J. Xu, D. C. Colvin, T. E. Yankeelov, E. C. Parsons, and M. D. Does, "Characterization of tissue structure at varying length scales using temporal diffusion spectroscopy," NMR in Biomedicine, vol. 23, no. 7, pp. 745–756, 2010.

[26] M. G. Hall and D. C. Alexander, "Convergence and parameter choice for monte-carlo simulations of diffusion mri," Medical Imaging, IEEE Transactions on, vol. 28, no. 9, pp. 1354–1364, 2009.

[27] D. G. Norris and J. Hutchison, "Concomitant magnetic field gradients and their effects on imaging at low magnetic field strengths," Magnetic resonance imaging, vol. 8, no. 1, pp. 33–37, 1990.

[28] B. A. Landman, J. A. Farrell, S. A. Smith, D. S. Reich, P. A. Calabresi, and P. van Zijl, "Complex geometric models of diffusion and relaxation in healthy and damaged white matter," NMR in biomedicine, vol. 23, no. 2, pp. 152–162, 2010.

[29] F. Bloch, W. Hansen, and M. Packard, "Nuclear induction," Physical review, vol. 70, no. 7-8, pp. 460–474, 1946.

[30] D. Kirk and W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Applications of GPU Computing Series, Elsevier Science, 2010.

[31] M. Friedman, Principles and models of biological transport. Springer-Verlag New York, 2008.

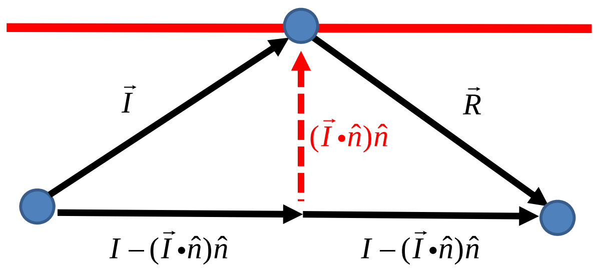
[32] A. Szafer, J. Zhong, and J. C. Gore, "Theoretical model for water diffusion in tissues," Magnetic Resonance in Medicine, vol. 33, no. 5, pp. 697–712, 1995.

[33] G. Rule and T. Hitchens, Fundamentals of Protein NMR Spectroscopy. Focus on Structural Biology, Springer, 2006.

A Derivations

A.1 Specular Reflection Operator

Consider the following diagram of a particle being specularly reflected off of a boundary with a normal $\hat{\boldsymbol{n}}$, incident vector $\boldsymbol{I}$ and reflected vector $\boldsymbol{R}$:



$\vec{I}$
$\vec{R}$
$(\vec{I}\bullet\hat{n})\hat{n}$
$I-(\vec{I}\bullet\hat{n})\hat{n}$
$I-(\vec{I}\bullet\hat{n})\hat{n}$

Figure 17: This figure shows the different quantities used in the specular reflection operator derivation. A single particle (blue) approaches the boundary with an a normal $\hat{\boldsymbol{n}}$ and a incident vector $\hat{\boldsymbol{I}}$

From this diagram, we can see that the reflected vector is

$$\boldsymbol{R} = \boldsymbol{I} - 2(\boldsymbol{I}\cdot\hat{\boldsymbol{n}})\hat{\boldsymbol{n}}. \tag{55}$$

We can write this as

$$\boldsymbol{R} = (\boldsymbol{1} - 2\hat{\boldsymbol{n}}\hat{\boldsymbol{n}})\cdot\boldsymbol{I}. \tag{56}$$

Thus, we define the specular reflection operator as

$$\hat{\boldsymbol{R}} = \boldsymbol{1} - 2\hat{\boldsymbol{n}}\hat{\boldsymbol{n}}. \tag{57}$$

A.2 Fick's Law of Membranes

The boundary condition for a permeable boundary is

$$J(\boldsymbol{r}) \cdot \hat{n} = P(C_+(\boldsymbol{r}) - C_-(\boldsymbol{r})), \tag{58}$$

where P is the permeability and C_+ and C_- are the concentrations on either side of the boundary. This condition is often called "Fick's Law of Membranes", let's derive it. The following derivation is similar to the one given in [31].

Consider a thin boundary of finite thickness t_m between two regions. According to Fick's first law of diffusion, the net diffusion current in the region of the membrane is

$$J = -D_m \frac{dC}{dx}. \tag{59}$$

Equation 59 implies

$$Jdx = -D_m dC. \tag{60}$$

If the boundary is very thin, then J is roughly constant across the boundary. In this case, we can integrate Equation (60) across the boundary:

$$J = \frac{D_m(C_+ - C_-)}{t_m}, \tag{61}$$

where, C_+ and C_- are the concentrations on either side of the boundary.

In the limit that both D_m and t_m go to zero, our boundary becomes infinitesimal and the ratio $\frac{D_m}{t_m}$ is defined as the permeability P. With this definition, we get the boundary condition

$$J = P(C_+ - C_-). \tag{62}$$

While this derivation proceeded using the one-dimensional form of J, it is not hard to see that the generalization in 3 dimensions would be

$$\boldsymbol{J} \cdot \hat{\boldsymbol{n}} = P(C_+ - C_-). \tag{63}$$

A.3 Permeability and Transmission Probabilities

The standard method used to simulate boundary condition 63 is by transmission probabilities. If a walker intersects a boundary during a timestep, there is a certain probability that it will be transmitted. We can derive this probability as a function of the walker's velocity and the surface permeabiliity.

Consider an ensemble of walkers restricted by some semi-permeable ($P \neq 0$) barrier. The diffusion current density J, colliding with one side of the boundary is given by the standard kinetics relation [32]

$$\boldsymbol{J} \cdot \hat{\boldsymbol{n}} = \frac{1}{4} v C(x, y, z), \tag{64}$$

where C is the concentration near a point (x,y,z) on the boundary and v is the particle velocity. To derive 64, consider a bunch of diffusing particles enclosed by some boundary surface. In a single timestep, these particles travel a distance $\lambda = v\delta t$, where v is their speed. Let us focus our attention on a collection of particles near a small surface element dA of the boundary. Let us say their perpendicular distance to the surface element is between x and x+dx. Thus there are $CdxdA$ particles in this collection, where C is the concentration close to the area element. For each of these particles, they will hit the boundary if their trajectory is within a cone of diagonal length λ. The solid angle subtended by a cone is well known and is $2\pi\left(1 - \frac{x}{\lambda}\right)$ for cones of height x and diagonal length λ. The fraction of particles in this collection that hits the surface is thus

$$f = \frac{2\pi\left(1 - \frac{x}{\lambda}\right)}{4\pi}. \tag{65}$$

The total number of particles in all collections that can hit this element is therefore

$$\begin{aligned} N &= \int_0^{\lambda} CdxdA\frac{1}{2}\left(1-\frac{x}{\lambda}\right) \\ &= \frac{CdA\lambda}{4} \\ &= \frac{CdAv\delta t}{4}. \end{aligned} \tag{66}$$

In the limit that δt goes to dt, we can also calculate the number of particles that hit this surface element by

$$N = \boldsymbol{J} \cdot \hat{\boldsymbol{n}} dAdt. \tag{67}$$

Comparing equations 66 and 67 we get equation 64.

The flux which gets through the - side of the boundary is therefore

$$\boldsymbol{J}_{-} \cdot \hat{\boldsymbol{n}} = \frac{1}{4} v C_{-}(\text{x,y,z}) p_{-}, \tag{68}$$

where p_{-} is the probability of a single particle transmitting through. We can write a similar equation for $\boldsymbol{J}_{+}$, incident on the boundary from the other side. Putting it together, we derive

$$\frac{1}{4}C_{+}v_{+}p_{+} - \frac{1}{4}C_{-}v_{-}p_{-} = P(C_{+} - C_{-}). \tag{69}$$

The only way Equation (69) could hold is if

$$\begin{aligned} P &= \frac{1}{4}p_{+}v_{+} \\ &= \frac{1}{4}p_{-}v_{-}. \end{aligned} \tag{70}$$

The update equation for a particle that is initially on the - side of the boundary would therefore be

$$X(t_{n+1}) = \begin{cases} x(t_n) + \Delta W & : x(t_n) + \Delta W \in \Omega \\ x(t_n) + \alpha \Delta W + (1-\alpha) R \cdot \Delta W & : x(t_n) + \Delta W \notin \Omega \text{ and } U(0,1) > \frac{4P}{v} \\ x(t_n) + \alpha \Delta W + (1-\alpha) \frac{v_+}{v_-} \Delta W & : \text{otherwise}, \end{cases} \tag{71}$$

where U(0,1) is a uniform deviate between 0 and 1, and $v_\pm$ are the speeds on either side of the boundary. Thus, in a simulation, if a walker intersected a semi-permeable boundary in a single timestep, we would update its position with the following pseudo-code:

Listing 1: Permeability Update Equation Pseudo-Code

```
/////////////////////////////////////////////////////////////////
//  UniformDeviate(0,1) = uniform random number between 0 and 1
//  P = boundary permeability
//  v = current velocity of particle
/////////////////////////////////////////////////////////////////

//The while loop is needed in order to catch multiple reflections
//or multiple transmissions.

while ( boundaryIntersection( particlePosition ) == true){
if ( UniformDeviate(0,1) > 4*P/v ) {
// reflect particle
}

else {
//transmit particle
}

}
```

A.4 Multiple Reflection in a Circle

We will prove that no matter how small you make the timestep, there is always a possibility that a particle could reflect multiple times off of a circular boundary within that single timestep. Consider a circle of radius r_c and let the origin of our coordinate system coincide with the center of the circle. Now consider a particle inside the circle that just interesected the boundary with an incident vector $\boldsymbol{I}$. Let the intersection point lie a distance α from the initial point of the incident vector and suppose the step length has magnitude λ.

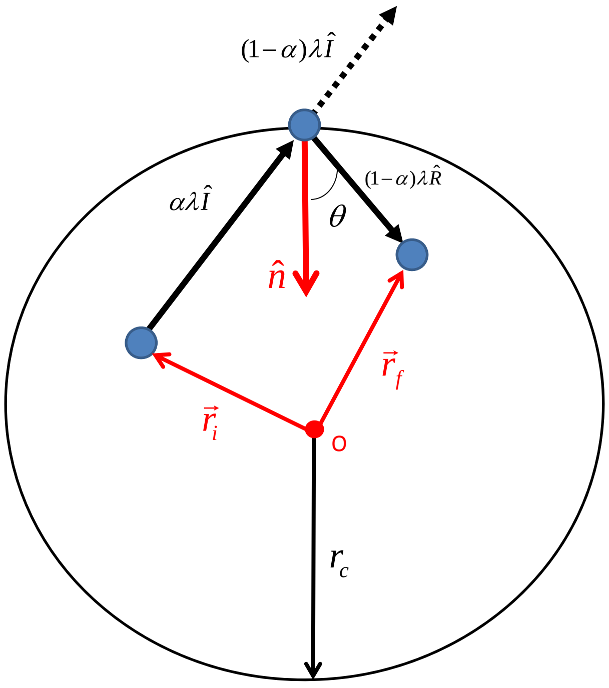


Figure 18: Specular reflection of a particle.

From appendix A.1, we know that the reflected vector will have a unit vector

$$\hat{\boldsymbol{R}} = \hat{\boldsymbol{I}} - 2(\hat{\boldsymbol{n}} \cdot \hat{\boldsymbol{I}}), \tag{72}$$

and the initial and final vectors are

$$\boldsymbol{r_f} = -r_c\hat{\boldsymbol{n}} + (1-\alpha)\lambda\hat{\boldsymbol{R}} \qquad \text{and} \qquad \boldsymbol{r_i} = -r_c\hat{\boldsymbol{n}} - \alpha\lambda\hat{\boldsymbol{I}}.$$

So far, the particle has only reflected once. But what is the condition for a second reflection? That would be

$$\boldsymbol{r_f}^2 > r_c^2 \qquad \text{and} \qquad \boldsymbol{r_i}^2 < r_c^2.$$

These conditions lead to the following constraints

$$(1-\alpha)\frac{\lambda}{r_c} > \cos(\theta) \qquad \text{and} \qquad 1 < \frac{\cos(\theta) r_c}{\alpha\lambda}.$$

where $\cos(\theta) = \hat{\boldsymbol{R}} \cdot \hat{\boldsymbol{n}}$ and $\alpha \neq 0, 1$. Now, we want to investigate what happens when the step length is arbitrarily small. So fix some ratio $\frac{\lambda}{r_c}$ and make it as close to zero as you want. I claim that I can find an angle θ and a α such that there will be a second reflection. Let me choose: $\alpha = \frac{1}{4}$ and $\cos(\theta) = (1-2\alpha)\frac{\lambda}{r_c}$. These choices satisfy both constraints. Thus, no matter how large you make the circle or how small you make the step length (or both), I can always find a θ and a α where a second reflection will occur. QED.

A.5 Derivation of Stepisnik Signal Attenuation Equation

The following proof is a less rigorous version of the derivation of Equation (44) by Stepisnik in [23]. Some of the details he left out are filled in. So let's begin.

In a standard MRI experiment, a series of magnetic field pulses are applied to a tissue sample containing nuclear spins. The nuclear spins start precessing and a signal is produced in a pickup coil. The received MRI signal is the transverse plane and is of the form

$$S = \left|\langle I_x \rangle + i \langle I_y \rangle\right|, \tag{73}$$

where I_x and I_y are the usual nuclear spin operators.

With this in mind, consider a single nuclear spin-1/2 particle with a wavefunction

$$\psi(t) = \sum c_i(t) \left| u_i \right\rangle, \tag{74}$$

where $\{\left| u_i \right\rangle\}$ are a complete set of states.

A measurement of the x-component of the tranverse signal would give

$$\langle I_x \rangle = \sum_{ij} c_j{}^*(t) c_i(t) \left\langle u_j \right| I_x \left| u_i \right\rangle. \tag{75}$$

We can define the operator ρ, known as the density matrix, by

$$\rho_{ij} = c_j{}^*(t) c_i(t). \tag{76}$$

The density matrix for a single spin has the following nice property, which is easily proven:

$$\langle I_x \rangle = trace[I_x \rho]. \tag{77}$$

If there is more than one particle, it can be shown that the ensemble average is [33]

$$\overline{\langle I_x \rangle} = trace[\overline{\rho} I_x]. \tag{78}$$

Thus, we can describe the ensemble system with the average density matrix operator. Hereafter, the overbar denoting the ensemble average will be dropped. Since the density matrix is an operator, it can be rotated by

$$R \rho R^\dagger, \tag{79}$$

where R is a rotation operator. In high-field MRI, in a frame rotating at the larmor frequency, a pulse can be represented as a rotation operator. In a gradient echo sequence, the evolution of the density matrix would thus be

$$\rho(t) = R_G R_{90} \rho(0) R_{90}^\dagger R_G^\dagger, \tag{80}$$

where

$$R_{90} = exp\left(i\frac{\pi}{2}I_x\right), \text{ and} \tag{81}$$

$$R_G = exp\left(i\gamma\sum_n\int_0^t \boldsymbol{G}(t')\cdot\boldsymbol{r}_n(t')dt' I_z^n\right). \tag{82}$$

Here $\boldsymbol{r}_n(t)$ is the position of the nth particle and $\boldsymbol{G}(t')$ is the gradient strength.

Now, the x component of the complex signal is

$$\langle I_x\rangle = \langle Trace[I_x\rho(t)]\rangle, \tag{83}$$

where the trace runs over the nuclear spin coordinates and the $\langle\rangle$ is the ensemble average of a bunch of identically prepared systems. Substituting in Equation (80), we get

$$\langle I_x\rangle = \left\langle Trace[I_x R_G R_{90}\rho(0)R_{90}^\dagger R_G^\dagger]\right\rangle. \tag{84}$$

Now, at t = 0, we expect a Boltzmann distribution of spins in the $\frac{-\omega_0\hbar}{2}$ and $\frac{\omega_0\hbar}{2}$ spin up and spin down states. That is,

$$\begin{aligned}\rho(0) &= \frac{e^{-\beta H}}{Z} \\ &\approx \frac{1}{Z}(1-\beta\omega_0 I_z),\end{aligned} \tag{85}$$

in the high temperature limit. Equation 85 into 84 gives us

$$\begin{aligned}
\langle I_x \rangle &= -\frac{\beta}{Z}\omega_0 Trace\left[\left\langle I_x R_G R_{90} I_z R_{90}^\dagger R_G^\dagger \right\rangle\right] \\
&= -\frac{\beta}{Z}\omega_0 Trace\left[\left\langle I_x R_G I_y R_G^\dagger \right\rangle\right] \\
&= -\frac{\beta}{Z}\omega_0 Trace\left[\left\langle I_x \sum_n \left\{cos(\gamma\phi_n) I_y^n + sin(\gamma\phi_n) I_x^n\right\} \right\rangle\right] \\
&= -\frac{\beta}{Z}\omega_0 \sum_n Trace\left[\left\langle sin(\gamma\phi_n)(I_x^n)^2 \right\rangle\right] \\
&= -\frac{\beta}{2Z}\omega_0 \sum_n Trace\left[\left\langle sin(\gamma\phi_n)\left((I_x^n)^2 + I_z^n + (I_y^n)^2\right) \right\rangle\right] \\
&= -\frac{\beta}{2Z}\omega_0 \sum_n Trace\left[\left\langle I_+^n I_-^n sin(\gamma\phi_n) \right\rangle\right],
\end{aligned} \tag{86}$$

where

$$\phi_n(t) = \int_0^t \boldsymbol{G}(t') \cdot \boldsymbol{r}_n(t') dt'. \tag{87}$$

Similarly,

$$\langle I_y \rangle = -\frac{1}{2Z}\beta\omega_0 \sum_n Trace\left[I_+^n I_-^n \left\langle cos(\gamma\phi_n(t)) \right\rangle\right]. \tag{88}$$

Therefore, the complex signal is

$$S_c = -i\frac{1}{2}\beta\omega_0 \sum_n Trace\left[I_+^n I_-^n \left\langle exp(-i\gamma\phi_n(t)) \right\rangle\right]. \tag{89}$$

Now, by the cumulant expansion theorem states[12] that

$$ln \left\langle exp(-i\gamma\phi_i(t)) \right\rangle = \sum_1^\infty \frac{(-i\gamma)^n}{n} \left\langle \phi_i^n \right\rangle_c, \tag{90}$$

where $\langle \phi_i^n \rangle_c$ is the nth order cumulant of ϕ_i.[12]. Therefore

[12]For a definition of a n-th order cumulant see [12] The first order cumulant is the mean and the second order cumulant is the variance. For this derivation, that is all that needs to be known.

$$\langle exp(-i\gamma\phi_i(t))\rangle = exp\left[\sum_1^\infty \frac{(-i\gamma)^n}{n}\langle\phi_i^n\rangle_c\right]. \tag{91}$$

Now the first and second order cumulants (the mean and variance) reduce down to the normal moments when $\langle r_i(t)\rangle = 0$, which is the case for stagnant liquids. If we assume a stagnant liquid, then the first order cumulant drops out and

$$S \approx \left|-i\frac{1}{2}\beta\hbar\omega_0\sum_n Trace\left[I_+^n I_-^n exp(-\beta)\right]\right|, \tag{92}$$

where

$$\beta = \frac{1}{2}\gamma^2\int_0^t\int_0^t dt_1dt_2\left\langle \boldsymbol{G}(t_1)\cdot\boldsymbol{r}(t_1)\boldsymbol{G}(t_2)\cdot\boldsymbol{r}(t_2)\right\rangle. \tag{93}$$

We have dropped the subscript n on $\boldsymbol{r}$ in β because the 2nd order moment will be the same for each particle. We have also dropped the subscript c on the $\langle\rangle$, because the 2nd order cumulant reduced down to the 2nd order moment due to the condition $\langle r_i(t)\rangle = 0$.

In gradient echo sequences, the integral of the gradient over time is always zero. Therefore, at the echo time, the average of the phase should be approximately zero and hence Equation (92) becomes a very good approximation, where the echo attenuation factor is given by Equation (93). QED.

A.6 Picking Points Uniformly on a Sphere

We want to pick points uniformly on a sphere and it would seem at first to be very easy. Why not just pick uniform deviates from $\phi \in [0, 2\pi]$ and $\theta \in [0, \pi]$? When dealing with spherical coordinates, we must recall that the area element size is a function of θ, namely, $sin(\theta)$. It thus becomes vanishly small as θ goes to 0. This means that if we uniformly pick $\theta \in [0, \pi]$ and transform back to cartesian coordinates, we will find that most of the points land near the poles (since more area elements are near the poles and less elements are at the equator, where they are gigantic.). We can, however, uniformly pick $\phi \in [0, 2\pi]$ because the area element doesn't change size as we change ϕ. For θ what we need to find is a bijective mapping between it and some uniform variable on $[0, 1]$ and then invert that mapping. Fortunately this is easy, consider the following

$$\begin{aligned} U(0,1) &= \frac{1}{2}\int_{\theta}^{\pi} \sin(\theta')d\theta' \\ &= \frac{1}{2}(1+\cos(\theta)), \end{aligned} \tag{94}$$

where U(0,1) is a uniform deviate between 0 and 1 and $\theta \in [0, \pi]$.

Inverting this equation we get

$$\theta = \operatorname{acos}(2U(0,1) - 1). \tag{95}$$

Thus to pick points (θ, ϕ) uniformly on a sphere we use

$$(\theta, \phi) = (\operatorname{acos}(2\boldsymbol{U}(0,1) - 1), 2\pi \boldsymbol{V}(0,1)), \tag{96}$$

where U and V are uniform random numbers between 0 and 1.

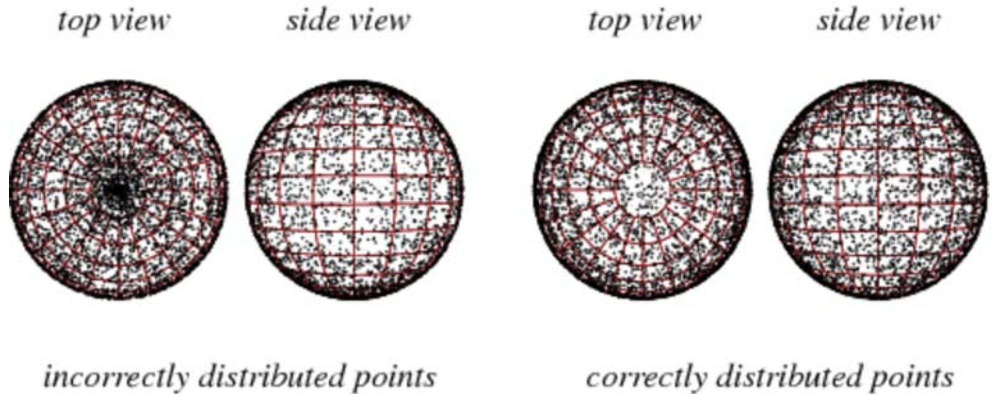


Figure 19: This is a figure showing points picked on sphere incorrectly and correctly. We notice in the incorrect case, that the points are mostly near the poles. This figure is from: `http://mathworld.wolfram.com/SpherePointPicking.html`

B Code

In this section I will present snippets of code from the TDMC library. The entire library consists of 100+ pages of code and cannot be displayed in its full form within this thesis.

B.1 Vector3 Class

This class is a modified (for use with CUDA) version of one appearing in [2].

Listing 2: Vector3 Class

```
class Vector3{
  public:
  /* Holds the value along the x axis. */
  real x;
  /* Holds the value along the y axis */
  real y;
  /* Holds the value along the z axis */
  real z;
  /* The default constructor creates a zero vector */
  __device__ __host__ Vector3() : x(0), y(0), z(0) {}
  /* The explicit constructor creates a vector with the
  given components    */
  __device__ __host__ Vector3(const real x, const real y, const real z) : x(x), y(y), z(z) {}
  // /* Flips all the components of the vector */
  __device__ __host__ void invert()
  {
  x = -x;
  y = -y;
  z = -z;
  }
  /* Gets the magnitude of this vector. */
  __device__ __host__ real magnitude() const
  {
  return sqrt(x*x + y*y + z*z);
  }
  /* Gets the squared magnitude of this vector. In some
```

```
cases we do not need the exact magnitude; for example, when
we need to compare two magnitudes to see which is greater, it
is faster the compare the squares of the magnitudes. This is
b/c we do not need to call the sqrt function.*/
__device__ __host__ real squareMagnitude() const
{
return x*x+y*y+z*z;
}
/* Turns a non-zero vector into a vector of unit length. */
__device__ __host__ void normalize()
{
real l = magnitude();
x /= l;
y /= l;
z /= l;
}
/* Multiplis this vector by the given scalar. */
__device__ __host__ void operator *= (const real value)
{
x *= value;
y *= value;
z *= value;
}
/* Returns a copy of this vector scaled to the given value */
__device__ __host__ Vector3 operator*(const real value) const
{
return Vector3(x*value,y*value,z*value);
}
__device__ __host__ Vector3 operator/(const real value) const
{
return Vector3(x/value,y/value,z/value);
}
/* Adds the given vector to this. */
__device__ __host__ void operator += (const Vector3& v)
{
x += v.x;
y += v.y;
z += v.z;
}
/* Returns the value of the given vector added to this. */
__device__ __host__ Vector3 operator +(const Vector3& v) const
```

```
{
return Vector3(x+v.x,y+v.y,z+v.z);
}
__device__ __host__ bool operator <(const Vector3 & v) const
{
return ( ( x < v.x ) && ( y < v.y ) && (z < v.z) );
}
__device__ __host__ bool operator >(const Vector3 & v) const
{
return ( ( x > v.x ) && ( y > v.y ) && (z > v.z) );
}
/* Subtracts the given vector from this. */
__device__ __host__ void operator-=(const Vector3& v)
{
x -= v.x;
y -= v.y;
z -= v.z;
}
/* Returns the value of the given vector subtracted from this. */
__device__ __host__ Vector3 operator-(const Vector3& v) const
{
return Vector3(x-v.x, y-v.y, z-v.z);
}
/* Adds the given vector to this, scaled by the given amount.
We could use the preceding methods to accomplish this, but it is
useful to have a separate method.*/
__device__ __host__ void addScaledVector(const Vector3& vector, real scale)
{
x += vector.x * scale;
y += vector.y * scale;
z += vector.z * scale;
}
/* Calculates and returns a component-wise product of this
vector with the given vector.*/
__device__ __host__ Vector3 componentProduct(const Vector3 &vector) const
{
return Vector3(x* vector.x, y*vector.y, z* vector.z);
}
/* Performs a component-wise product with the given vector
and sets this vector to its result*/
__device__ __host__ void componentProductUpdate(const Vector3 &vector)
```

```
{
x *= vector.x;
y *= vector.y;
z *= vector.z;
}
/* Calculates and returns the scalar product of this vector
with the given vector*/
__device__ __host__ real scalarProduct(const Vector3 &vector) const
{
return x*vector.x + y*vector.y + z*vector.z;
}
/* Calculates and returns the scalar product of this vector
with the given vector. */
__device__ __host__ real operator *(const Vector3 &vector) const
{
return x*vector.x + y*vector.y + z*vector.z;
}
/* Calculates and returns the vector product of this vector
with the given vector. */
__device__ __host__ Vector3 vectorProduct(const Vector3 &vector) const
{
return Vector3 (y*vector.z-z*vector.y,
z*vector.x-x*vector.z,
x*vector.y-y*vector.x);
}
/* Updates this vector to be the vector product of its
current value and the given vector*/
__device__ __host__ void operator %=(const Vector3 &vector)
{
*this = vectorProduct(vector);
}
/* Calculates and returns the vector product of this
vector with the given vector. */
__device__ __host__ Vector3 operator%(const Vector3 &vector) const
{
return Vector3(y*vector.z-z*vector.y, z*vector.x-x*vector.z, x*vector.y -y*vector.x);
}
__host__ friend std::ostream& operator<< (std::ostream &out, Vector3 & v){
out << v.x << " " << v.y << " " << v.z ;
return out;
}
```

```
};
```

B.2 Example of CUDA code

Consider the problem of adding two arrays (vectors) of size 1024. This size represents the number of threads we need on the GPU. Let us see how we add these two arrays using standard C on the CPU:

Listing 3: Adding two vectors with standard C

```
#include <iostream>
#define N 1024

void add( int *a, int *b, int *c ) {
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}

int main( void ) {
  int* a, b, c;

  //alocate space on the host
  a = (int *)malloc( N * sizeof(int) ) ;
  b = (int *)malloc( N * sizeof(int) ) ;
  c = (int *)malloc( N * sizeof(int) ) ;

  for (int i=0; i<N; i++) {a[i] = -i; b[i] = i * i;}

  //compute sum on host
  add( a, b, c );

  // display the results
  for (int i=0; i<N; i++) {std::cout << c[i] << std::endl; }

  // free the memory allocated on the host
  free( a ) ; free( b ); free( c ) ;
}
```

Now, let's add the two vectors using CUDA C.

Listing 4: Adding two vectors with CUDA C on the GPU

```
#define N 1024
#include <iostream>

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    // this thread handles the data at its thread id
    c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_b, N * sizeof(int) ) ;
    cudaMalloc( (void**)&dev_c, N * sizeof(int) ) ;

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {a[i] = -i; b[i] = i * i;}

    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    //Compute the N threads using 1 Block
    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy( c, dev_c, N * sizeof(int),cudaMemcpyDeviceToHost ) ;

    // display the results
    for (int i=0; i<N; i++) {std::cout << c[i] << std::endl; }

    // free the memory allocated on the GPU
    cudaFree( dev_a ) ;
    cudaFree( dev_b ) ;
    cudaFree( dev_c ) ;

    return 0;
}
```

Now lets add these two vectors using my wrapping library.

Listing 5: Adding two vectors with TDMC Wrapping library

```
#define N 1024
#include <iostream>
#include "cudaVector.cuh"

int main( void ) {
  cudaVector<int> a(N);
  cudaVector<int> b(N);
  cudaVector<int> c(N);

  for (int i=0; i<N; i++) {a[i] = -i; b[i] = i * i;}

  //copy to GPU
  a.copyToDevice();
  b.copyToDevice();

  //add a and b on the GPU

  c.equalToSumOf(a,b);

  //copy result back to the CPU

  c.copyFromDevice();
  for (int i=0; i<N; i++) {std::cout << c[i] << std::endl; }
    return 0;
}
```

The above code is much more readable and simpler.

B.3 Under the Microscope 1: The Computation Kernel

For simplicity, we will examine the single pore kernel, without the Lattice class but with otherwise maximum generality. The full kernel looks like:

Listing 6: The Computation Kernel

```
template <
     class FieldType,
     class Basis,
     class Integrator,
     class StepCreator,
     class StepAmender
     >

__global__ void updateWalkersMag(
          const SimuParams *par,
          const Basis* basis,
          const FieldType * B,
          curandState* globalState,
          real* Mx,
          real* My,
          real* Mz
         )
{

  Integrator integrator;
  StepCreator stepCreator;
  StepAmender stepAmender;

  const unsigned int tid = threadIdx.x + blockIdx.x*blockDim.x;
  curandState localState = globalState[tid];
  real speed = sqrt(6.0*basis->getD()/par->timestep);
  Vector3 r = basis[0].unifRand(localState);

  for (int i = 0; i < par->measurements; i++){

    Mx[tid + i*par->number_of_particles] = par->mx_initial;
    My[tid + i*par->number_of_particles] = par->my_initial;
    Mz[tid + i*par->number_of_particles] = par->mz_initial;

  integrator.integrate (
          &Mx[tid + i*par->number_of_particles],
            &My[tid + i*par->number_of_particles],
            &Mz[tid + i*par->number_of_particles],
          basis,
          B[i],
            r,
          0,
            par->timestep
        )

  }

  for (int i = 1; i < par->steps; i++){

    Vector3 ri = r;
  stepCreator.update(r,v);
  stepAmender.correct(ri,r,speed,basis,par->timestep);

    for (int j = 0; j < par->measurements; j++){
  integrator.integrate (
          &Mx[tid + j*par->number_of_particles],
            &My[tid + j*par->number_of_particles],
            &Mz[tid + j*par->number_of_particles],
          basis,
          B[j],
            r,
          i,
            par->timestep
        )
    }

  }
}
```

The data sent to the GPU with the “const" (Basis, FieldType, SimuParams, etc) prefix is stored in the L1-Cache on the MP, which is a on-chip memory of user-specifiable size (up to 48K). All other data is held in DRAM. In the WCDMC design, large arrays representing the high-field gradient profile are stored in DRAM, which is slow. With functors we no longer need to store large arrays in slow DRAM memory (we store it in the L1-Cache) and therefore a substantial speedup should be seen.

B.4 Under the Microscope 2: The CPU Interface

Due to the generality of the design, the CPU interface of TDMC is quite easy to use. Let us see how you would set up a high-field cosine gradient echo sequence and a hexagonal lattice full of cylinders.

Listing 7: The CPU Interface

```
int main(){

  int number_of_particles = 114688; //needs to be a multiple of two
  real timestep = .001;

  int threads = 128;
  int blocks = number_of_particles/threads;

  phaseAcquisitionStream<CosGFunc> pas(number_of_particles);

  int NOI = 100;
  int NOM = 2;

  //set up cosine gradient echo. We will calculate 100 ADCs (see NOI) using 2 signal ↩
      measurements (see NOM)
  for (int j = 0; j < NOI; j++){

    real gradient_duration= 10;
    real gradient_spacing  = 2.0;
    real echo_time = 2.0*gradient_duration + gradient_spacing ;
    int number_of_timesteps = (int) (echo_time/timestep);
    phaseAcquisition<CosGFunc> pa(NOM,number_of_timesteps,number_of_particles,j*time(NULL));
    for (int i = 0; i < NOM; i++) {

      int N = 1+j;
      real G = i*0.0000025*N;

      CosGFunc cosGRAD(G, gradient_duration,gradient_spacing, N, Vector3(1.0,0.0,0.0));
      pa.addMeasurement(cosGRAD);

    }
    pas.addAcquisition(pa);
  }

  for (int r = 0; r < 4; r++){

  real radius = .0025;
  real D_extra = 2.5E-6;
  real D_intra = 1.0E-6;
  real T2_i = 200;
  real T2_e = 200;

  //intracellular fraction
  real f = .6 + r*.1;

  //distance between cylinders on hexagonal array
  real d = sqrt( 2.0*PI*radius*radius/( sqrt(3.0)*f ) );

  //four cylinders make up the basis unit cell for a hexagonal array
  Cylinder_XY cyls[4];

  //the lattice
  Lattice lattice(d, 2.0*d*0.86602540378443864676372317075294, d, T2_e, 0.0, D_extra,4);

  cyls[0] = Cylinder_XY(d/2.0, 0.0,  radius,  T2_i,0.0, D_intra, 1,0.0);
  cyls[1] = Cylinder_XY(0.0, d*0.86602540378443864676372317075294,  radius,  T2_i,0.0, D_intra,↩
       2,0.0);
  cyls[2] = Cylinder_XY(d/2.0, 2.0*d*0.86602540378443864676372317075294,  radius,  T2_i,0.0, ↩
      D_intra, 3,0.0);
  cyls[3] = Cylinder_XY(d, d*0.86602540378443864676372317075294,  radius,  T2_i,0.0, D_intra, ↩
      4,0.0);

  //This is where you specify how many GPUs you have and how the work should be partitioned ↩
```

```
    onto them.
  //In this case we want only one GPU to do all acquisitions (NOI).
  std::vector<int> plan(3); plan[0] = 0; plan[1] = NOI;  plan[2] = NOI;
  std::vector<int> numOfSMPerDevice(1); numOfSMPerDevice[0] = 14; numOfSMPerDevice[1] = 2;

  //number of back to back trials
  int repeats = 3;

  //Taking the mean of a bunch of back to back simulations is very easy! Just wrap everything ↩
    in a for loop
  //and the interface will take care of the rest.
  for (int i = 0; i < repeats; i++){
    pas.runAcquisitionStreamLattice(cyls, lattice,  timestep, blocks, threads, 1, plan, ↩
        numOfSMPerDevice);
    pas.calcEveryADC();
    pas.changeSeeds(time(NULL)*(i+1));
  }

  for (int i = 0; i < NOI; i++){
    std::cout << setprecision(20);
    std::cout << pas.getAcquisition(i).getGradientFunctors()[0].getFreq() << " " ;
    std::cout << pas.getAcquisition(i).getADC() << std::endl;
  }
  }
}
```

B.5 Under the Microscope 3: The Basis Class

The Basis class is actually a placeholder name for any user-specified object to be placed on a lattice or used as a single pore constraining boundary. In this section I will briefly examine a simple 3d example, the sphere. All of the basis objects must share a few of same variables and functions in order for them to work in the kernel, I will mention which these are as I introduce them. Finally, the basis class design allows for almost any imaginable object or set of objects to be described.

So let‘s begin creating a Sphere basis class:

Listing 8: The Sphere Basis Class

```
class Sphere {

private:

  real centerX;
  real centerY;
  real centerZ;
  real radius;
  real D;
  real T2;
  real T1;
  real permeability
  int region;

//...
```

With the exception of the radius and the center variables, all of the other private variables must be in each basis object. The only exception is if the object is a subcomponent of some overall basis object (i.e. a single triangle in a triangular mesh). Since the L1 Cache can have up to 48K of space per SM, it will easily fit the 68 bytes (36 bytes for single precision) allocated for a single double precision sphere object. The constructor for this class is non-interesting since it just initializes each of the private members. Therefore, we move on to the first important function of the Sphere class:

Listing 9: The Sphere Basis Class

```
__device__ Vector3 Sphere::unifRand(curandState localState) const{
Vector3 r;
//This do-while loop generates random points within a cube of size diameter x diameter x ↩
    diameter
//until it finds one that lies in the sphere.
//the function "inside(r)" returns true if r is in the sphere

do {
r = Vector3( radius*(2.0*curand_uniform(&localState) - 1),
      radius*(2.0*curand_uniform(&localState) - 1),
        radius*(2.0*curand_uniform(&localState) - 1));

} while ( !inside(r) );

return r;
}

//...
```

If the sphere will be used as a single pore without a lattice, then we will need to uniformly distribute the particles initially within the sphere. Interestingly, this can ONLY be done using rejection sampling, where we generate points within an bounding cubic volume until one lands within the sphere. Other methods generate points that are too close to the boundary of the sphere (below the finite precision of the computer), which is always a problematic region in finite precision computation.

With a Lattice, this last function would be rarely used since most of time we would want the particles uniformly distributed about the lattice, not a single basis object. However, any basis object which will be used as a single pore system must have this function. The next important function, is one which determines whether a point is inside the basis. For the Sphere class, this would be:

Listing 10: The Sphere Basis Class

```
  __device__ __host__ bool inside(const Vector3 & r) const{
    return ((r.x - centerX)*(r.x - centerX) +
      (r.y - centerY)*(r.y - centerY) +
      (r.z - centerZ)*(r.z - centerZ) < radius*radius);
  }
//...
```

This is a very useful function. For example, if you have a Lattice with many basis objects you will need to know where a particle is so you can return the right D, T_1, T_2, P etc and for this you can just determine whether the particle is inside the basis. What if this Sphere had another sphere in it which had different properties? In this case, these ”subcomponents" should be joined into a single basis object easily.

The last important function is the intersect function, this determines whether the basis object has been intersected. This function will vary greatly between different basis objects.

Listing 11: The Sphere Basis Class

```
/////////////////////////////////////////////////////
// This function determines the intersection point
// of a particles trajectory with the Sphere object
// and returns true if there has been an intersection.
//
// ri = initial position of the particle
// rf = final position of the particle
// v = intersection parameter (between 0 and 1)
////////////////////////////////////////////////////

  __device__ __host__ bool intersect(const Vector3 & ri, const Vector3 & rf, real & v) const{

  Vector3 dr = rf - ri;
    real step_mag = dr.magnitude();

  //components of the quadratic ax^2 + bx + c
  real a = dr.x*dr.x + dr.y*dr.y + dr.z*dr.z;
  real b = 2.0*ri.x*dr.x - 2.0*dr.x*centerX + 2.0*ri.y*dr.y - 2.0*dr.y*centerY + 2.0*ri.z*dr.z ↩
      - 2.0*dr.z*centerZ;
  real c = ri.x*ri.x + ri.y*ri.y +ri.z*ri.z -2*ri.x*centerX -2*ri.y*centerY -2*ri.z*centerZ  + ↩
      centerX*centerX + centerY*centerY + centerZ*centerZ - radius*radius;

  //solving the quadratic equation ax^2 + bx + c = 0 for the two roots
  real q = -.5*(b + sgn(b)*sqrt(b*b - 4*a*c));
    real root1 = q/a;
    real root2 = c/q;

  //the root of the quadratic equation must be:
  // 1) between 0 and 1,
  // 2) must not be complex,
  // 3) and must be not be the initial point (i.e. the particle started on the boundary)
  // Condition 3) occurs whenever there is a multiple reflection

    bool s1 = (root1 > 0.0 && root1 < 1.0 && b*b>4*a*c && !doub_equal(root1*step_mag,0.0));
    bool s2 = (root2 > 0.0 && root2 < 1.0 && b*b>4*a*c && !doub_equal(root2*step_mag,0.0));
    bool s3 = (fabs(root1) < fabs(root2));

  //if both roots are valid intersections, we set v to the closest intersection
  //otherwise we set v to the only valid intersection
  //or nothing if no intersection exists
    if ( (s1 && s2 && s3) || (s1 && !s2)){ v = root1; return true;}
  else if ((s1 && s2 && !s3) || (s2 && !s1)){ v = root2; return true;}
  else {return false;}
  }
//...
```

So what is going on here? Well, a particles position can be represented by the parametric equation:

$$\boldsymbol{r} = v(\boldsymbol{r_f} - \boldsymbol{r_i}) + \boldsymbol{r_i} \tag{97}$$

and we are solving for the parameter v such that r satisfies:

$$(\boldsymbol{r} - \boldsymbol{r_c})^2 = (radius)^2 \tag{98}$$

The rest of the sphere class consists of “getter" functions which return the parameters (T_1, T_2, D, P) is a particle is found to be within the basis object.

B.6 Under the Microscope 3: The Lattice Class

If your geometry consists of multiple basis objects and you want periodic boundary conditions then you will need a Lattice. A Lattice in TDMC imposes periodic boundary conditions and returns properties of the bulk geometry (i.e. all of the basis objects). If you do not want periodic boundary conditions, but you want multiple basis objects, what you can do is wrap the basis objects in another basis object, for example you could wrap a bunch of cylinders into a bigger impermeable cube. Usually periodicity is assumed in biological tissue, so lattices are quite useful. Lastly, the lattice space which the basis objects sit on also has parameters (T_1, T_2, D) (no permeability because the lattice doesn't have boundaries). In biological tissues, often there is a extracellular space which can be easily represented in this framework as the lattice space. So let's examine the Lattice class:

Listing 12: The Lattice Class

```
class Lattice {

private:

  real a, b, c; //lattice parameters
  real T2_lattice;
  real T1_lattice;
  real D_lattice;
  int basisSize; //number of basis objects

public:

  Lattice(){}
  Lattice(real _a, real _b, real _c, real _T2, real _T1, real _D, int _basisSize){
    T2_lattice = _T2;
  T1_lattice = _T1;
  D_lattice = _D;
  a=_a; b=_b; c=_c;
  basisSize = _basisSize;
  }

  //initialize uniformly about the lattice space on the GPU
  __device__ void initializeUniformly(Vector3 & r, curandState & localState) const
  {

  r.x = curand_uniform(&localState)*a;
  r.y = curand_uniform(&localState)*b;
  r.z = curand_uniform(&localState)*c;

  }

  //initialize uniformly about the lattice space on the CPU
    __host__ void initializeUniformlyCPU(Vector3 & r) const
  {

    r.x = unifRandCPP()*a;
    r.y = unifRandCPP()*b;
    r.z = unifRandCPP()*c;

  }

  //initialize particles in a certain basis object
  template <class Basis>
    __device__ void initializeInRegion( Basis * basis,
             curandState & localState,
             Vector3 & r,
           int region
           ) const

    {

    do {
      r.x = curand_uniform(&localState)*a;
      r.y = curand_uniform(&localState)*b;
      r.z = curand_uniform(&localState)*c;
    } while ( getRegion(basis,r) != region);

    }

  //if the particle is outside the lattice unit cell,
  //take the modulo division of its position with the basis size to
  //move it back inside the unit cell
  __device__ __host__ void correctBoundary(Vector3 & r) const
  {
    if (!inLatticeCell(r)){
      lmod(r);
    }
  }
```

```
//lattice modulo division - very useful when a particle moves outside
//the lattice unit cell
__device__ __host__ void lmod(Vector3 & r) const {

  if( r.x > a) { r.x = fmod(r.x,a); }
  if( r.x < 0.0) { r.x = fmod(r.x,a) + a; }

  if( r.y > b) { r.y = fmod(r.y,b); }
  if( r.y < 0.0) { r.y = fmod(r.y,b) + b; }

  if( r.z > c) { r.z = fmod(r.z,c); }
  if( r.z < 0.0) { r.z = fmod(r.z,c) + c; }

}

//return the T2 value of the object the particle is currently in
//if the particle isn't in any object, return the T2 value of the lattice
template <class Basis>
__device__ __host__ double getT2(Basis * basis, Vector3 & r) const{

  double T2;
Vector3 temp = r;
lmod(temp);
  for (int i = 0; i < basisSize; i++){
    if( T2 = basis[i].getT2(temp), T2 > 0){
  return T2;
    }
  }
  return T2_lattice;

}

//return the T1 value of the object the particle is currently in
//if the particle isn't in any object, return the T1 value of the lattice
 template <class Basis>
__device__ __host__ double getT1(Basis * basis, Vector3 & r) const{

  double T1;
Vector3 temp = r;
lmod(temp);
```

```
    for (int i = 0; i < basisSize; i++){
      if( T1 = basis[i].getT1(temp), T1 > 0){
    return T1;
      }
    }
    return T1_lattice;

  }

  //return the D value of the object the particle is currently in
  //if the particle isn't in any object, return the D value of the lattice
 template <class Basis>
  __device__ __host__ double getD(const Basis * basis, const Vector3 & r) const{

    double D;
  Vector3 temp = r;
  lmod(temp);
    for (int i = 0; i < basisSize; i++){
      if( D = basis[i].getD(temp), D > 0){
  return D;
      }
    }

    return D_lattice;

  }

  //If you have multiple objects, it is good to label them with numbers. Think
  //about the child "paint by number" coloring books to get the idea.
  //return the region value of the object the particle is currently in
  //if the particle isn't in any object, return the region value of the lattice (region = 0)
   template <class Basis>
  __device__ __host__ int inRegion(const Basis * basis, const Vector3 & r) const{

    int region;
  Vector3 temp = r;
  lmod(temp);
    for (int i = 0; i < basisSize; i++){
      if ( region = basis[i].getRegion(temp), basis[i].inside(temp)){
    return region;
      }
    }

    return 0;
  }

  //determines whether a particle is in the unit cell of the lattice
  __device__ __host__ bool inLatticeCell (Vector3 & r)const{
    if ( r.x > a || r.x < 0.0 ||
        r.y > b || r.y < 0.0 ||
             r.z > c || r.z < 0.0 )
      {
    return false;
      }
  return true;
  }

  //checks for intersections with ALL basis objects
  //before an intersection with a basis object is checked, the particles
  //position must be modulo divided by the basis size, otherwise you will
  //get incorrect results. This function should be used if the basis objets
  //have permeability.
   template <class Basis>
  __device__ __host__ bool intersectionCheckPermeable (const Basis *basis,
               const Vector3 & ri,
               const Vector3 & rf,
               real & v,
             Vector3 & n,
             real & permeability)const

  {

    //displacement vector

    Vector3 dr = rf - ri;

    //modded line 1

    Vector3 mod1_f = rf; lmod(mod1_f);
    Vector3 mod1_i = mod1_f - dr;

    //modded line 2
    Vector3 mod2_i = ri; lmod(mod2_i);
    Vector3 mod2_f = mod2_i + dr;

    real vTemp = 0.0;
    real vBestEst = 10.0;

    for (int i = 0; i < basisSize; i++){

      if(basis[i].intersect(mod1_i, mod1_f, vTemp) && vTemp < vBestEst){
    vBestEst = vTemp;
    Vector3 intPoint = dr*vBestEst + mod1_i;
    n = basis[i].getNormal( intPoint );
    permeability = basis[i].getPermeability();
      }

      if(basis[i].intersect(mod2_i, mod2_f, vTemp) && vTemp < vBestEst){
    vBestEst = vTemp;
    Vector3 intPoint = dr*vBestEst + mod2_i;
    n = basis[i].getNormal( intPoint );
    permeability = basis[i].getPermeability();
      }

    }

  //return the closest intersection (vBestEst)
  v = vBestEst;
    return (vBestEst < 5.0);

  }

  //checks for intersections with ALL basis objects
  //before an intersection with a basis object is checked, the particles
  //position must be modulo divided by the basis size, otherwise you will
  //get incorrect results. This function should be used if the basis objects
  //have NO permeability. Note that technically this function isn't needed
  //because we could just use the function above with P = 0. However, it is
  //slightly faster in the case of P = 0, so it might be preferable in this case.

   template <class Basis>
  __device__ __host__ bool intersectionCheckImpermeable(const Basis *basis,
               const Vector3 & ri,
               const Vector3 & rf,
               real & v,
             Vector3 & n)const

  {

    //displacement vector

    Vector3 dr = rf - ri;

    //modded line 1

    Vector3 mod1_f = rf; lmod(mod1_f);
    Vector3 mod1_i = mod1_f - dr;

    //modded line 2
    Vector3 mod2_i = ri; lmod(mod2_i);
    Vector3 mod2_f = mod2_i + dr;

    real vTemp = 0.0;
    real vBestEst = 10.0;

    for (int i = 0; i < basisSize; i++){

      if(basis[i].intersect(mod1_i, mod1_f, vTemp) && vTemp < vBestEst){
    vBestEst = vTemp;
    Vector3 intPoint = dr*vBestEst + mod1_i;
    n = basis[i].getNormal( intPoint );
      }

      if(basis[i].intersect(mod2_i, mod2_f, vTemp) && vTemp < vBestEst){
    vBestEst = vTemp;
    Vector3 intPoint = dr*vBestEst + mod2_i;
    n = basis[i].getNormal( intPoint );
      }

    }

  v = vBestEst;
    return (vBestEst < 5.0);
  }

  //return the number of basis objects
  __device__ __host__ int getBasisSize(){
  return basisSize;
  }

};
```

B.7 Under the Microscope 4: Field Functors

This section will provide some examples of field functors. It should be noted that these examples don't even scratch the surface of what you can do with field functors. Almost any pulse sequence can be represented. Consider first a very simple field functor, that of a high-field rectangular pulse GRE sequence.

Listing 13: High Field Rectangular Pulse GRE

```
class RECT_GRE{
private:
  real B0;
  Vector3 G;
  real gradientDuration;
  real gradientSpacing;
public:
  __device__ __host__ RECT_GRE(){
    B0 = 0.0;
    G = Vector3(0.0,0.0,0.0);
    gradientDuration = 0.0;
    gradientSpacing = 0.0;
  }
  __device__ __host__ RECT_GRE(real _B0, Vector3 _G, real _gradientDuration, real ↩
      _gradientSpacing){
    B0 = _B0;
    G = _G;
    gradientDuration = _gradientDuration;
    gradientSpacing = _gradientSpacing;
  }
  __device__ __host__ Vector3 operator() (Vector3 r, real time) const {
    if ( time < gradientDuration) {
      return Vector3(0.0, 0.0, B0 + G*r);
    }
    else if ( time >= gradientSpacing + gradientDuration && time < 2.0*gradientDuration + ↩
        gradientSpacing) {
      return Vector3(0.0, 0.0, B0 - G*r);
    }
    else {
      return Vector3(0.0, 0.0, B0);
    }
  }
};
```

For a second example, consider the above sequence but with a gradient in the y-direction with the corresponding gradient term.

Listing 14: Low Field Rectangular Pulse GRE

```
class QSC_GRE{
private:
  real B0;
  real G;
  real gradientDuration;
  real gradientSpacing;
public:
  __device__ __host__ QSC_GRE(){
    B0 = 0.0;
    G = 0.0;
    gradientDuration = 0.0;
    gradientSpacing = 0.0;
  }
  __device__ __host__ QSC_GRE(real _B0, real _G, real _gradientDuration, real _gradientSpacing)↩
      {
    B0 = _B0;
    G = _G;
    gradientDuration = _gradientDuration;
    gradientSpacing = _gradientSpacing;
  }
  __device__ __host__ Vector3 operator() (Vector3 r, real time) const {
    if ( time < gradientDuration) {
      return Vector3( 0.0, G*r.z, B0 + G*r.y );
    }
    else if ( time >= gradientSpacing + gradientDuration && time < 2.0*gradientDuration + ↩
        gradientSpacing) {
      return Vector3( 0.0, -G*r.z, B0 - G*r.y );
    }
    else {
      return Vector3(0.0, 0.0, B0);
    }
  }
  __host__ real getB(){
  return G*G*gradientDuration*gradientDuration*GAMMA*GAMMA*( (2.0/3.0)*gradientDuration + ↩
      gradientSpacing);
  }
  __host__ real getDiffusionTime(){
  return gradientSpacing + gradientDuration;
  }
};
```

B.8 Under the Microscope 5: The AcquisitionStream Class

The AcquisitionStream class is a container for all the user input and it is given the important job of calling the computation kernel from the CPU. The function which calls the kernel is perhaps the most complicated function in the entire design. It not only partitions the work onto multiple GPUs, but then subpartitions this work into two streams. The design is very similar to the one outlined for Multiple GPUs with two buffers in [13]. Let us take a look at this function in the magAcquisitionStream class for a single pore simulation (no Lattice)

Listing 15: The GPU calling function in the AcquisitionStream Class

```
template <class Basis>
  void magAcquisitionStream::runAcquisitionStream(Basis & basis,
              Vector3 & initialM,
              real timestep,
              int blocks,
              int threads ,
              int numOfDevices,
              std::vector<int> & plan,
              std::vector<int> & numOfSMPerDevice){

  //these vectors will exist in Pinned CPU memory
    vector< pinnedVector<FieldType> > host_fields(acqSet.size());
  vector< pinnedScalar<SimuParams > > host_pars(acqSet.size());
    vector< pinnedVector<real> > host_mx(acqSet.size());
  vector< pinnedVector<real> > host_my(acqSet.size());
  vector< pinnedVector<real> > host_mz(acqSet.size());

  //allocate pinned memory
    for (int i = 0; i < acqSet.size(); i++){

    host_fields[i].alloc(meas_max);
    host_mx[i].alloc(meas_max);
    host_my[i].alloc(meas_max);
    host_mz[i].alloc(meas_max);

    }

  //these vectors will contain data to be stored or retrieved on the GPU. We will need a sub-↩
      vector
  //for each GPU and a sub-sub-vector for each of the two streams!
    std::vector< std::vector< cudaVector< FieldType > > > dev_fields(numOfDevices,std::vector< ↩
        cudaVector< FieldType > >(2) );
  std::vector< std::vector< cudaScalar<SimuParams> > > dev_pars(numOfDevices, std::vector< ↩
      cudaScalar< SimuParams > >(2) );
  std::vector< std::vector< cudaVector<real> > >dev_mx(numOfDevices,std::vector< cudaVector<↩
      real> >(2));
  std::vector< std::vector< cudaVector<real> > >dev_my(numOfDevices,std::vector< cudaVector<↩
      real> >(2));
  std::vector< std::vector< cudaVector<real> > >dev_mz(numOfDevices,std::vector< cudaVector<↩
      real> >(2));
```

```
std::vector< std::vector< cudaVector<real> > >dev_total_mx(numOfDevices,std::vector< ↩
    cudaVector<real> >(2));
std::vector< std::vector< cudaVector<real> > >dev_total_my(numOfDevices,std::vector< ↩
    cudaVector<real> >(2));
std::vector< std::vector< cudaVector<real> > >dev_total_mz(numOfDevices,std::vector< ↩
    cudaVector<real> >(2));
std::vector< std::vector< cudaVector<curandState> > > devStates(numOfDevices,std::vector< ↩
    cudaVector<curandState> >(2));
std::vector< std::vector< cudaScalar<Basis> > > dev_basis(numOfDevices, std::vector< ↩
    cudaScalar< Basis > >(2) );
std::vector < std::vector<cudaStream_t> >streams(numOfDevices, std::vector<cudaStream_t>(2));

//allocate memory for the above vectors on each of the devices
for (int i = 0; i < numOfDevices; i++){
  safe_cuda(cudaSetDevice(i));

  for (int j = 0; j < 2; j++){

  dev_fields[i][j].setDevice(i);
  dev_mx[i][j].setDevice(i);
  dev_my[i][j].setDevice(i);
  dev_mz[i][j].setDevice(i);
  dev_total_mx[i][j].setDevice(i);
  dev_total_my[i][j].setDevice(i);
  dev_total_mz[i][j].setDevice(i);
  devStates[i][j].setDevice(i);
  dev_pars[i][j].setDevice(i);
  dev_basis[i][j].setDevice(i);

  dev_fields[i][j].malloc(meas_max);
  dev_mx[i][j].malloc(meas_max*number_of_particles);
  dev_my[i][j].malloc(meas_max*number_of_particles);
  dev_mz[i][j].malloc(meas_max*number_of_particles);
  dev_total_mx[i][j].malloc(meas_max);
  dev_total_my[i][j].malloc(meas_max);
  dev_total_mz[i][j].malloc(meas_max);
  devStates[i][j].malloc(number_of_particles);
  dev_pars[i][j].malloc();
  dev_basis[i][j] = basis;
  dev_basis[i][j].copyToDevice();
  cudaStreamCreate( &streams[i][j] );
  }
}

//run through the number of GPUs on the system
for (int devNum = 0; devNum < numOfDevices; devNum++){

safe_cuda(cudaSetDevice(devNum));
int numOfSM = numOfSMPerDevice[devNum];

for (int i = plan[devNum]; i < plan[devNum+1]; i=i+2){

//variables set for TWO streams
int steps = acqSet[i].getNumOfSteps();
int steps1 = acqSet[i+1].getNumOfSteps();
int measurements = acqSet[i].getNumOfMeas();
int measurements1 = acqSet[i+1].getNumOfMeas();

//copying to Pinned memory
host_fields[i] = acqSet[i].getFieldFunctors();
host_fields[i+1] = acqSet[i+1].getFieldFunctors();
host_pars[i] = SimuParams(number_of_particles,
               steps,
               measurements,
               timestep,
               acqSet[i].getSeed(),
               initialM.x,
               initialM.y,
               initialM.z );



host_pars[i+1] = SimuParams(number_of_particles,
               steps1,
               measurements1,
               timestep,
               acqSet[i+1].getSeed(),
               initialM.x,
               initialM.y,
               initialM.z );
```

```
host_fields[i].copyToDevice(dev_fields[devNum][0], streams[devNum][0]);
host_fields[i+1].copyToDevice(dev_fields[devNum][1], streams[devNum][1]);

host_pars[i].copyToDevice(dev_pars[devNum][0], streams[devNum][0]);
host_pars[i+1].copyToDevice(dev_pars[devNum][1], streams[devNum][1]);

//initialize the seeds for each particle on the GPU
setup_kernel <<<blocks, threads, 0, streams[devNum][0]>>> (devStates[devNum][0].getPointer(),↩
     dev_pars[devNum][0].getPointer());
setup_kernel <<<blocks, threads, 0, streams[devNum][1]>>> (devStates[devNum][1].getPointer(),↩
     dev_pars[devNum][1].getPointer());

//run the simulation
updateWalkersMag<<<blocks, threads, 0, streams[devNum][0]>>> (dev_pars[devNum][0].getPointer↩
    (),dev_basis[devNum][0].getPointer(), dev_fields[devNum][0].getPointer(), devStates[↩
    devNum][0].getPointer(), dev_mx[devNum][0].getPointer(), dev_my[devNum][0].getPointer(), ↩
    dev_mz[devNum][0].getPointer() ) ;
updateWalkersMag<<<blocks, threads, 0, streams[devNum][1]>>> (dev_pars[devNum][1].getPointer↩
    (),dev_basis[devNum][1].getPointer(), dev_fields[devNum][1].getPointer(), devStates[↩
    devNum][1].getPointer(), dev_mx[devNum][1].getPointer(), dev_my[devNum][1].getPointer(), ↩
    dev_mz[devNum][1].getPointer() );

//Compute the total magnetization by parallel reduction
dev_mz[devNum][0].sum(dev_total_mz[devNum][0], 768, numOfSM, measurements, ↩
    number_of_particles, streams[devNum][0]);
dev_mz[devNum][1].sum(dev_total_mz[devNum][1], 768, numOfSM, measurements1, ↩
    number_of_particles, streams[devNum][1]);
dev_my[devNum][0].sum(dev_total_my[devNum][0], 768, numOfSM, measurements, ↩
    number_of_particles, streams[devNum][0]);
dev_my[devNum][1].sum(dev_total_my[devNum][1], 768, numOfSM, measurements1, ↩
    number_of_particles, streams[devNum][1]);
dev_mx[devNum][0].sum(dev_total_mx[devNum][0], 768, numOfSM, measurements, ↩
    number_of_particles, streams[devNum][0]);
dev_mx[devNum][1].sum(dev_total_mx[devNum][1], 768, numOfSM, measurements1, ↩
    number_of_particles, streams[devNum][1]);

//copy the magnetization back to the CPU
host_mz[i].copyFromDevice(dev_total_mz[devNum][0], streams[devNum][0]);
host_mz[i+1].copyFromDevice(dev_total_mz[devNum][1], streams[devNum][1]);
host_my[i].copyFromDevice(dev_total_my[devNum][0], streams[devNum][0]);
host_my[i+1].copyFromDevice(dev_total_my[devNum][1], streams[devNum][1]);
```

```
host_mx[i].copyFromDevice(dev_total_mx[devNum][0], streams[devNum][0]);
host_mx[i+1].copyFromDevice(dev_total_mx[devNum][1], streams[devNum][1]);


  }
}

//free the two streams for each of the devices
for (int devNum = 0; devNum < numOfDevices; devNum++){
  safe_cuda(cudaSetDevice(devNum));
  cudaStreamSynchronize( streams[devNum][0] );
  cudaStreamSynchronize( streams[devNum][1] );
  cudaStreamDestroy( streams[devNum][0] );
  cudaStreamDestroy( streams[devNum][1] );

}

  addMagnetizationSet(host_mx, host_my, host_mz);
}
```

B.9 Under the Microscope 6: Multiple reflection and Warp Voting

Consider the multiple specular reflection update code:

Listing 16: The specular reflection update method (no warp voting).

```
template <class Basis>
__device__ void boundaryNormal(Vector3 & ri, Vector3 & r, const real currentSpeed,  const Basis↩
    * basis, const real timestep){

  real v = 0.0;
  real accumtime = 0.0;

  while (basis->intersect(ri, r, &v)){

    //determine incident vector
    r = (r-ri)*v + ri;
    Vector3 I = (r-ri);
    real Imag = I.magnitude();

    //calculate accumulated time
    accumtime += Imag/currentSpeed;

    //determine normal for reflection calculation
    Vector3 n = basis->getNormal(r);

    //set last position to boundary position before rflection
    ri = r;

    //reflect, reflection travel time is the remaining time (timestep - accumtime)
    r += ( I - n*2.0*(n*I) )*(currentSpeed/Imag)*( timestep - accumtime );
  }
}
```

We can try and optimize this like the CUDA compiler. We can force all of the threads in a warp through the same instruction path by using “warp voting". Consider the code:

Listing 17: The specular reflection update method with warp voting.

```
template <class Basis>
__device__ void boundaryWarpVoting(Vector3 & ri, Vector3 & r, const real currentSpeed,  const ↩
    Basis* basis, const real timestep){

  real v = 0.0;
  real accumtime = 0.0;
  real alive;

  //the function __any(alive) returns true if any of the threads in a warp have alive = 1
  //thus one divergent thread forces the entire warp to continue on down the same instruction ↩
      path
  while (alive = basis->intersect(ri, r, &v), __any(alive) ) {

    r = ( ( r-ri)*v + ri )*(alive) + r*(!alive);

    //determine incident vector
    const Vector3 I = (r-ri);
    const real Imag = I.magnitude();

    //calculate accumulated time
    accumtime += Imag/currentSpeed;

    //determine normal for reflection calculation
    const Vector3 n = basis->getNormal(r);

    //set last position to boundary position before rflection
    ri = r*(alive) + ri*(!alive);

    //reflect, reflection travel time is the remaining time (timestep - accumtime)
    r += ( I - n*2.0*(n*I) )*(currentSpeed/Imag)*( timestep - accumtime ) * (alive);
  }
}
```

The __any(alive) CUDA function returns true if any of the threads in a warp have alive = 1. This means that if there is even a single divergent thread, the entire warp gets forced through the same instruction path. Surprisingly this warp voting actually increases the speed of the multiple reflection update on the GPU by 1% to 2%. It is very small because the time saved removing thread divergence is almost balanced by the time lost computing extra “useless" instructions.

C Other Figures

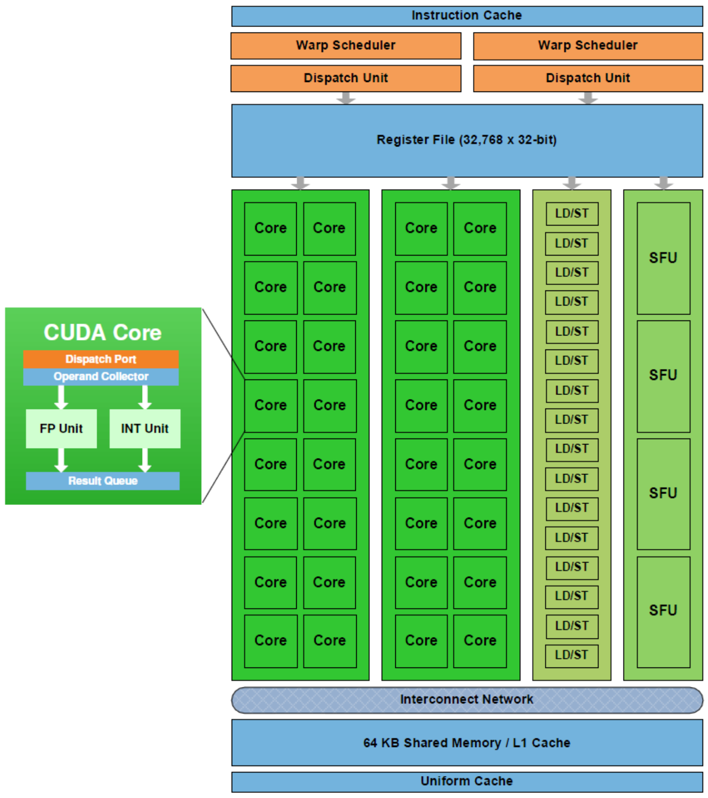


Figure 20: Schematic Diagram of a Fermi Generation MP. Image taken from `http://www.nvidia.ca/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`