Engineering Improved Magnetorheological Fluids

By

Benjamin T. Wilson

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Chemical Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

2016

Date of final oral examination: January 26th, 2016

The dissertation is approved by the following members of the Final Oral Committee:

Prof. Daniel J. Klingenberg, Professor, Chemical Engineering

Prof. Michael D. Graham, Professor, Chemical Engineering

Prof. Thatcher W. Root, Professor, Chemical Engineering

Prof. Regina M. Murphy, Chemical Engineering

Prof. Daniel C. Ludois, Assistant Professor, Electrical Engineering

To my family members who could not make it to see today: Harvey, Shirley and Charlene.

And to my close friend David Bertorello who also could not be here.

Acknowledgments

I would first like to thank Prof. Klingenberg for giving me the opportunity to pursue my Ph.D. Under his guidance, I have grown as both a scientist and individual. I certainly would not be at this point without his mentorship. I would also like to thank my funding, the National Science Foundation and General Motors.

I would also like to thank the members of my committee, Prof. Michael Graham, Prof. Thatcher Root, Prof. Regina Murphy, and Prof. Daniel Ludois for taking time out of their busy schedules to serve on my thesis committee.

I need to thank my parents, Richard and Sue, for always providing love, support, and encouragement during my years at both Purdue and the University of Wisconsin. I certainly would not be in this position without them. I would also like to thank my brother, Bill, for the laughs and occasional visit. I need to thank my grandfather, Melvin, for continuing to challenge me intellectually and serving as a role model.

I would also like to thank members of my extended family: Skip and Tami Seaman and their three daughters, Mary, Rebecca, and Sandra. I always look forward to getting to visit with you and appreciate your hospitality whenever I am in town. I am also very thankful for the love, support and encouragement you have provided me throughout my life.

I am also very thankful for the support and encouragement from my cousin Andrew and his wife Stephanie. There is rarely a dull moment around them.

I have been very fortunate to develop some close and long-lasting friendships during both my undergraduate and graduate careers. I appreciate all the time I was able to work with Brian Lowry. I certainly would not have made it to Wisconsin without him. I also want to thank Ashley Baltes and Mary Regier; I am continually learning from the example they set. They have been there for me during good times and bad. I am also indebted to Dr. Peter and Megan Mushenheim. I will always miss my Friday lunches with Peter and have always had fun when Megan is in town. I also would like to thank Tyler Roberts. We have had a lot of fun, and I have always enjoyed our lengthy discussions. I have always enjoyed discussing college football and oddball sports stories with Aaron Fluitt. Aaron's dry sense of humor helped with some of the absurdities and frustrations involved in graduate school. I also appreciate the support and encouragement from Robert Seidel; even if that support and encouragement seemed a little odd most of the time. I am also very grateful to have been able to share an office with Sandy Wang. Sandy made showing up to the office everyday fun. I will miss going to get tea with her.

I have also been lucky to have several good labmates in the Klingenberg group throughout the years. I would first like to thank Joe Samaniuk for not only recruiting me into the group, but also recruiting me to the University of Wisconsin. One of these days I will beat you in fantasy baseball. I would also like to thank my former labmate Jianghui Wang for helping me get started in code development. Life would have been a lot duller without being able to joke around with Josh Duncan. I also appreciated the advice of our former postdoc Anaram Shahravan. I have also enjoyed what work

I have done with Shalaka Burlawar and Sandy Chen.

I have been fortunate to have shared an office with several members of Prof. Graham's group. I have enjoyed joking around with Frank Nguyen. I appreciated the guidance and pep talks from Kushal Sinha in frustrating times, and his sense of humor in the not so frustrating times. I am also very appreciative to all the help in learning how to code from Friedemann Hahn. I am continually learning from Friede and look forward to visiting him in Germany. I also appreciate getting to work with Rafael Henriquez on course work as well as sitting across from him the last two years. I also appreciate the help from Sarit Dutta with LaTex and research discussions.

I also need to thank my friends from the Hugh O'Brien Youth Leadership Association (HOBY). I am continually learning from Holly and Reed Marks, David and Laura Burton Bertorello, Sabrina List, Lena Darnay, Turk White, and Ashley and Curt Shogren. I certainly would not be in this position were it not for their love, support, and encouragement. I am also thankful to have become good friends with Nick and Ali Guzik, Daniel Cannon, Breanna Holder, Sean Kelty, Taylor Bankroff, Curt Brooks, Laura Bryant, Al Carroll, Anne Dare, Brian Fedje, Alana Hahn, Guy and Sandra Harris, Miranda Nehrig, Emily Puffer, Mary Saubert, Haley Klimaszewski, and Mary Miller.

I have been very lucky to have several very good friends from Purdue including Theodore and Katy Kindig Birky, Drew and Lindsay Liebrecht, Lauren Quig, Sean Britt, Seth Chadwick, Jacob Hobson, Ryan McCann, and Lindsay Williams. I also thank Jeries Smirat for introducing me to *Moneyball* as well as all our discussions about baseball.

I would also like to thank the good friends I made at the University of Wisconsin:

Jean Wheasler, Kevin Schulte, Jackie Rand, Rebecca Carlton, Luke Roling, Thomas Schwartz, and Gurdaman Khaira. I also am very appreciative of the help and assistance Andrew Seidl provided when I was learning CUDA. I also thank the women in the front department office who have always been available and willing to answer questions I might have: Christi Balas-Levenson, Kathy Heinzen, and Beth Brandl.

I am was also incredibly lucky to be introduced to an incredibly fun softball team that I have been lucky enough to play with over the last four years. I would like to thank: Mia Hospel (2B), Brian (SS/OF) and Colleen Remer (C), Mark (1B/SS/2B/C/OF) and Tanya Brooks (2B/OF), Kelly Johnson(OF), Patrick Norby(SS/OF), Chris Brady(OF), Gene Zadzilka(P), Michelle Van Schyndel (OF), Kara Kessler (3B), Kyle Christensen (SS), and Keith Foss (1B). The aforementioned Jackie Rand (C/OF) was also a member of the team. My positions were 3B, 1B, and OF. We won the 2015 Fall Championship.

I am also grateful to the close friends from my childhood: Jonathan Guse, Chad Shinneman, Braden Cook, and Jon Kruse.

I am also very thankful for the adults who served as excellent role models for me when I was young including my God parents Jay Fetters and Ame Kersten and my little league coach Jerry Warden.

I would also like to thank the 2015 Chicago Cubs for making the summer of 2015 a summer to remember. That includes, but is not limited to, the Ricketts family, Theo Epstein, Jed Hoyer, Jason McLeod, Crane Kenney, Regina Nicholson, Joe Maddon, Anthony Rizzo, Starlin Castro, Kris Bryant, Jake Arrieta, Jon Lester, Addison Russell, Dexter Fowler, Jorge Soler, Chris Coghlan, Kyle Schwarber, Hector Rondon, Jason Hammel, Kyle Hendricks, Dan Haren, Miguel Montero, David Ross, Tommy

LaStella, Arismendy Alcantara, Jonathan Herrera, Travis Wood, Trevor Cahill, Javier Baez, Justin Grimm, Neil Ramirez, Clayton Richard, Chris Denorfia, Austin Jackson, Jason Motte, Fernando Rodney, Zac Rosscup, Pedro Strop, and Tsuyoshi Wada

Abstract

Particle-level simulations were performed to investigate the rheological properties of magnetorheological suspensions containing a mixture of magnetizable and nonmagnetizable spheres. We demonstrate that nonmagnetizable spheres cause the yield stress to increase in monolayers and three-dimensional simulations, as is observed in three-dimensional experiments. We examine the role of nonmagnetizable spheres in the suspension structure for monolayer and three-dimensional suspensions. Structure measures examined included the fluctuations in volume fraction, the pair distribution functions, and the eigenvalue ratio of the mass moment tensor. Nonmagnetizable spheres cause structural changes to monolayers that differ from those in three-dimensional suspensions. However, all structural changes are small, especially when compared to the structural changes observed in bidsiperse suspensions. Therefore, the small structure changes caused by the addition of nonmagnetizable particles do not appear to cause the increase in yield stress.

Large amplitude oscillatory shear reveals that nonmagnetizable spheres increase the suspension stiffness; the transition to nonlinear rheological properties remains unaffected suggesting that the nonmagnetizable spheres do not alter the stability of the clusters of magnetizable spheres. Snapshots reveal that nonmagnetizable spheres

participate in stress transfer via repulsive-force clusters in a mechanism similar to jamming. The partial stresses, number of repulsive-force clusters, and transient rheological behavior further support that nonmagnetizable spheres directly enhance the stress via repulsive-force clusters. The repulsive-force clusters contain both magnetizable and nonmagnetizable spheres, which likely explains the observation that nonmagnetizable spheres enhance the field-induced stress, even though they are not magnetizable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Magnetorheological (MR) fluids are suspensions of magnetizable particles in a non-magnetizable, viscous, continuous phase. Applying a magnetic field with a flux density on the order of 1 Tesla causes the stress at low deformation rates to increase by orders of magnitude. The field-induced stress increase is both fast and reversible. The magnetic field induces magnetostatic particle interactions which cause the particles to aggregate, changing the suspension from a fluid-like state to a solid-like state, with a magnetic field-dependent yield stress [Ginder (1996); Jolly et al. (1998)]. This dramatic field-induced change in rheological properties is often called the MR effect. The tunable rheological properties make MR suspensions useful in numerous applications, including semiactive shock absorbers, clutches, actuators, servo valves, and precision polishing fluids [Jolly et al. (1998); Carlson and J.L. Sproston (2000); Klingenberg (2001)].

It is desirable to obtain the largest possible difference in rheological properties when the magnetic field is on (the "on-state") and when the magnetic field is off (the

"off-state"). A large difference between off-state and on-state rheological properties allows for a large range of dynamic control, smaller devices and fluid volumes, and therefore reduced costs. Ulicny et al. (2010) showed experimentally that the field-induced yield stress of concentrated MR suspensions can be increased significantly by adding nonmagnetizable particles to the suspension. The yield stress of an MR suspension (at magnetic saturation) with an iron particle volume fraction of 0.30 was increased by 50% by adding glass beads at volume fraction of 0.15. Furthermore, it is possible to increase the field-induced yield stress by replacing a fraction of the magnetizable particles with an equivalent volume of nonmagnetizable particles. Similar magnitudes of yield stress enhancement were observed for a variety of different types of nonmagnetizable particles [Klingenberg and J.C. Ulicny (2011)]. This phenomenon has also been observed in simulations of MR suspensions composed of mixtures of magnetizable and nonmagnetizable spheres [Ulicny et al. (2010); Klingenberg and J.C. Ulicny (2011)]. An understanding of the mechanisms that produce this phenomenon is still lacking.

Previous authors have shown that altering the microstructure of an MR suspension can lead to an enhancement in the yield stress of an MR suspension [Ulicny et al. (2005b); Kittipoomwong et al. (2008)]. In Chapter 3, we examine the role that nonmagnetizable spheres play in altering the structure of MR suspensions. One microstructural change that leads to a stress increase is the transient stress increase, which is attributed to the formation of lamellae, or sheet-like structures, in the plane of shear. Lamellae formation is a microstructural change that is exclusive to three-dimensional suspensions. When a sufficiently large magnetic or electric field is applied to a sheared MR or electrorheological (ER) suspension, respectively, the shear stress

first increases rapidly, and then continues to increase much more slowly [Vieira et al. (2000); Ulicny et al. (2005b)]. The slow transient increase in stress is caused by the formation of lamellar structures [Henley and F.E. Filisko (1999); Tang, X. et al. (2000); Volkova et al. (1999); Vieira et al. (2000); Ulicny et al. (2005b)]. Lamellae formation with transient stress increases has also been observed in particle-level simulations of flowing MR and ER suspensions [Martin (2000); Kittipoomwong (2007)]. We show in Figs. 3.2-3.4 that nonmagnetizable spheres cause the yield stress to increase in monolayer suspensions, contrary to previous studies [Ulicny et al. (2010)]. Since the enhancement occurs in both monolayer and three-dimensional systems, the mechanism for enhancement cannot be attributed to formation of lamellar structures.

Also in Chapter 3, we explore if nonmagnetizable spheres cause the MR suspensions to become more chain-like. Foister (1997) observed that MR suspensions with bimodal particle size distribution possessed a larger field-induced stress than that of monomodal suspensions at the same volume fraction. Similar experimental results were reported by Weiss et al. (2000) and Ulicny et al. (2004). Particle-level simulations of MR suspensions by Kittipoomwong et al. (2005) also produced larger field-induced stresses for bidisperse suspensions than those obtained for monodisperse suspensions at the same volume fraction. Kittipoomwong et al. (2005) probed the structure of the suspensions to determine the mechanisms by which smaller particles cause the bidisperse suspensions to have a larger yield stress than monodisperse suspensions. They measured the microstructure by examining fluctuations in the volume fraction, snapshots, the pair distribution function, and the eigenvalue ratio of the mass moment tensor. Kittipoomwong et al. (2005) showed that bidisperse suspensions formed more chain-like structures; monodisperse suspensions formed more

globular structures. The increase in number of chain-like structures of bidisperse suspensions causes the larger field induced yield stress. We employed these same measures to determine whether nonmagnetizable are altering the microstructure of the suspension, shown in Figs. 3.5-3.17. Nonmagnetizable spheres affect the structure of monolayers and three-dimensional suspensions in different ways. However, all structural changes caused by nonmagnetizable spheres are small, especially when compared to the changes observed in bidisperse suspensions.

In Chapter 3, we show that nonmagnetizable spheres cause only minor changes to the suspension structure. In Chapter 4, we examine the effect of short-range repulsive forces in determining the stress in the suspension. We begin Chapter 4 by examining dynamical measurements of both monolayer and three-dimensional suspensions. Dynamic measurements are common tools for probing the mechanisms of rheological behavior for complex fluids. We use large amplitude oscillatory shear (LAOS) to investigate the mechanisms which cause the yield stress increase for MR fluids that contain nonmagnetizable particles. In Figs. 4.1 - 4.4, we show that nonmagnetizable spheres increase the plateau modulus but do not alter the onset of nonlinearity. This indicates that nonmagnetizable spheres increase the stiffness of the field induced structures but do not alter the stability.

To explore the increased suspension stiffness, we create snapshots that visualize the spheres and the different attractive and repulsive pair-forces in Figs. 4.5-4.7. We show that the nonmagnetizable spheres produce stresses by participating in repulsive-force chains. These force chains are roughly aligned with the compression axis of the simple shear flow, and contain nonmagnetizable as well as magnetizable spheres. The ability of the nonmagnetizable spheres to transmit stress through purely repulsive

forces is similar to that found in jammed, hard-sphere suspensions [Cates et al. (1998), Farr et al. (1997)]. We illustrate the repulsive force chain formation with snapshots of sheared suspensions, characterize the resulting contribution to the shear stress by examining particle stresses and repulsive force statistics in Figs.4.8-4.19, and draw an analogy to previously reported jamming phenomena by considering the stress vs strain behavior in Figs. 4.20-4.21.

Chapter 5 explores the advantages of using parallel computing to simulate MR fluids. In 2007, the graphics card manufacturer NVIDIA began enabling their graphics cards the capability to perform scientific calculations. NVIDIA developed a programming language, based off C, known as Compute Unified Device Architecture (CUDA). Assuming a working knowledge of C, learning the CUDA syntax is straightforward. However, the challenge behind developing CUDA programs is learning how to develop algorithms that run in parallel instead of serial. The purpose of Chapter 5 is to help future students develop a basic understanding of the thought process behind parallel algorithm development.

In Section 5.2, two basic algorithms are described in both serial and parallel: a vector addition and dot product. In Section 5.3, the thought process behind the development of the particle-level simulations in CUDA is explored. The majority of the data generated for this document was done so using parallel algorithms in CUDA. A direct result of running simulations in CUDA is that the systems studied were both bigger and faster than previous studies [Ulicny et al. (2010)]. In Fig. 5.12, the speedup is presented as a function of number of spheres in the simulation. Figure 5.12 shows that the speedup over the serial simulations increases as more spheres are added to the suspension. In Fig. 5.13, the stress is plotted as a function of monolayer area.

Figure (5.13) shows that system size can have an effect on the physical properties of the suspension; therefore, bigger systems should be considered. Section 5.4 considers how hydrodynamic interactions might be introduced to the particle-level simulations presented in Section 5.3.

In Chapter 6, a brief overview of hydrodynamic interactions is presented. Experimental work has shown that coating magnetizable spheres with a nonmagnetizable coating can lower the viscosity of MR fluids when no field is applied [Ulicny et al. (2005a)] When MR fluids are not under the influence of a magnetic field, or the field is low, the fluid is dominated by van der Waals and hydrodynamic interactions. The purpose of Chapter 6 is to serve as a starting point for future students who simulate MR fluids by including hydrodynamic interactions.

Chapter 7 discusses potential avenues of future work in this field. At present, computational limitations have prevented a deeper understanding of MR fluids in the low-field regime. However, with the increased power of parallel computing available from graphics cards, a parallel algorithm which solves for the motion of MR fluids when the magnetic field is low can be implemented.

Chapter 2

Background

2.1 What are Magnetorheological Fluids?

Magnetorheological (MR) fluids consist of magnetizable particles suspended in a viscous continuous phase. Applying a magnetic field causes an MR fluid to undergo rheological changes. The ability to alter fluid properties in real time allows for a broad range of new and exciting devices that offer several advantages over their conventional counterparts. For example, General Motors has developed a shock absorber which uses an MR fluid to allow the driver and passengers to adjust the ride of the car [Corbett and Visnic (2000); Carlson and J.L. Sproston (2000); Klingenberg (2001)]. Another application currently being explored is an artificial leg which uses an MR fluid to better emulate the motion of the human knee. The MR knee offers amputees the ability to regain a range of motion not possible with conventional prosthetics [Flowers (1973); Grimes et al. (1977); James et al. (1990); Carlson and J.L. Sproston (2000); Herr and Wilkenfeld (2003); Johansson et al. (2005)]. Other devices include

MR fluid-based fan clutches which can help reduce fuel consumption [Rabinow (1948); Sakai (1988); Ginder (1996); Klingenberg (2001)]. By better understanding these fluids, it is possible to improve upon the devices currently in existence as well as creating devices not yet in existence.

In most cases, the magnetizable particles are made of a ferromagnetic material, although other magnetic materials do exist. The suspending fluid is usually a hydrocarbon-based liquid. When a magnetic field is applied, the fluid experiences a rapid increase in the apparent viscosity. Also, when the magnetic field is applied, the fluid develops a yield stress [Ginder (1996); Jolly et al. (1998); Ulicny et al. (2005b); Kittipoomwong et al. (2005)]. By controlling the magnetic field it is possible to control the rheological properties of the suspension.

There has been much research devoted to the case when the magnetic field is applied. One observation is that of a critical magnetic field. When the magnetic field is set to a value below the critical magnetic field, the suspension undergoes a rapid initial increase in apparent viscosity, but after the initial rapid increase, the viscosity remains at a constant value. However, when the magnetic field is above the critical magnetic field strength, after the initial jump in apparent viscosity, the suspension continues to undergo a slow, transient increase in the apparent viscosity [Ulicny et al. (2005b); Kittipoomwong et al. (2008)]. One possible source for this increase in apparent viscosity is due to the presence of colloidal forces and formation of lamellae [Ulicny et al. (2005b); Kittipoomwong et al. (2008)].

For commercial applications, it is desirable to obtain a large field-induced change in stresses [Klingenberg (2001)]. Another way of quantifying the change between the on-state and off-state stresses is by defining a turn-up ratio. The turn-up ratio is

the ratio of the shear stress at any particular given magnetic flux density divided by the shear stress when the magnetic flux density is zero (the off-state) [Ulicny et al. (2005a)]. There are two ways to increase the turn-up ratio: increase the on-state yield stress or decrease the off-state apparent viscosity. Using a higher concentration of magnetizable particles is one method for maximizing the on-state shear stress, but it also leads to an increase in price because of the high cost of the carbonyl iron used in most formulations [Lemaire et al. (1995); Klingenberg (2001); Genc and Phulé (2002); Kittipoomwong et al. (2005)]. Furthermore, adding more particles will also lead to an increase in apparent viscosity, which can be problematic in some applications [Klingenberg (2001); Kittipoomwong et al. (2005)]. Therefore, the turn-up ratio decreases as the concentration of magnetic particles suspended in the fluid is increased. The on-state yield stress and the off-state apparent viscosity are thus coupled [Ulicny et al. (2005a)].

2.2 Increasing Yield Stress by Addition of Nonmagnetizable Particles

One way to increase the high field yield stress is by adding nonmagnetizable spheres to the suspension [Ulicny et al. (2010)]. Experiments have shown that adding nonmagnetizable spheres to an MR fluid increases the high-field yield stress [Ulicny et al. (2010), Ulicny et al. (2013)]. Figure 2.1 is a plot of yield stress as a function of iron concentration for several experiments performed by Ulicny et al. (2010). Open circles represent suspensions containing bimodally distributed iron spheres. Open squares represent suspensions containing a mixture of glass spheres and bimodally

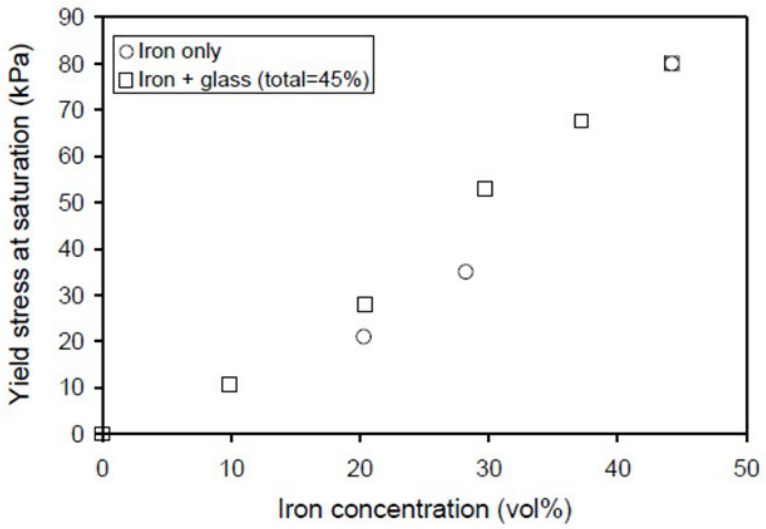

Figure 2.1: Yield stress at magnetic saturation as a function of iron concentration. Open circles represent bimodally distributed suspensions containing only iron spheres. Open squares represent a mixture of glass and bimodally distributed iron particles such that the total volume fraction is $\phi_T = \phi_M + \phi_N = 0.45$ [Ulicny et al. (2010)].

distributed iron spheres such that $\phi_T = \phi_M + \phi_N = 0.45$. For an iron concentration of 30%, adding a 15% concentration of glass creates a suspension with a $\approx 50\%$ increase in yield stress.

Figure 2.2 shows data from eight different experiments [Ulicny et al. (2013)]. In each experiment, the volume fraction of magnetizable particles is fixed at $\phi_M = 0.30$. The volume fraction of nonmagnetizable particles is fixed at $\phi_N = 0.15$. A different nonmagnetizable sphere is considered in each of the eight experiments. Figure 2.2 reveals a $\approx 50\%$ increase in yield stress for all nonmagnetizable particles considered. Therefore, the enhancement is independent of nonmagnetizable particle type.

The experimental results in Figs. 2.1 and 2.2 can be replicated via simulations, shown in Fig. 2.3 [Ulicny et al. (2010)]. In Fig. 2.3, yield stress is plotted as a function of magnetizable sphere volume fraction for three-dimensional suspensions.

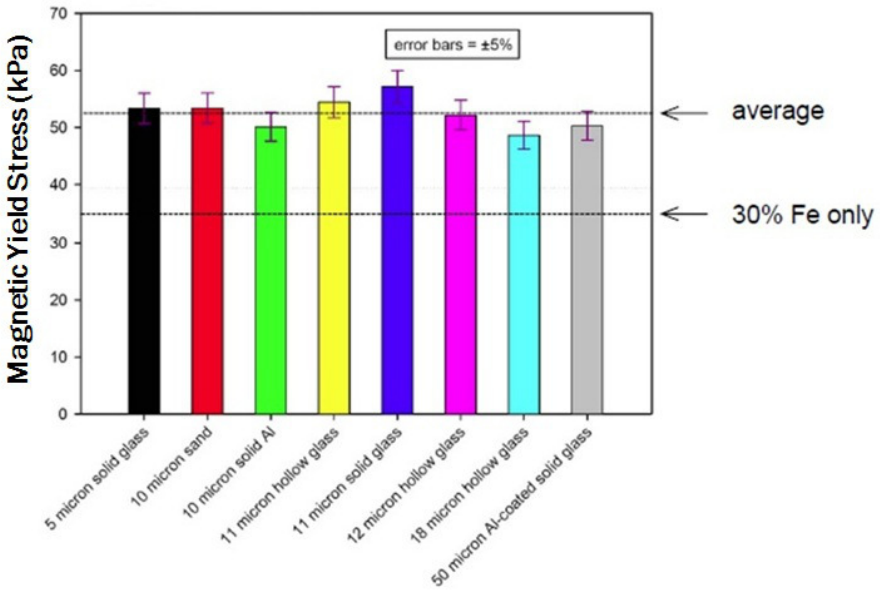

Figure 2.2: Yield stress at magnetic saturation for eight different experiments. In each experiment, a different nonmagnetizable particle was mixed with iron particle. $\phi_M = 0.30$ and $\phi_N = 0.15$ [Ulicny et al. (2013)].

Red squares represent a suspension containing only magnetizable spheres. Green circles represent a suspension containing a mixture of monodisperse magnetizable and nonmagnetizable spheres such that $\phi_T = \phi_M + \phi_N = 0.45$. For the interval $0.25 \leq \phi_M \leq 0.40$, nonmagnetizable spheres cause the suspensions of mixtures to have a larger yield stress than the suspensions containing only magnetizable spheres. By using simulations, systems and situations which cannot be explored through experiment can be pursued and understood. In Chapters 3 and 4, we will explore the underlying mechanism behind the yield stress enhancement due to nonmagnetizable particles. We will show that the nonmagnetizable particles induce a jamming-like phenomenon which causes the yield stress to increase.

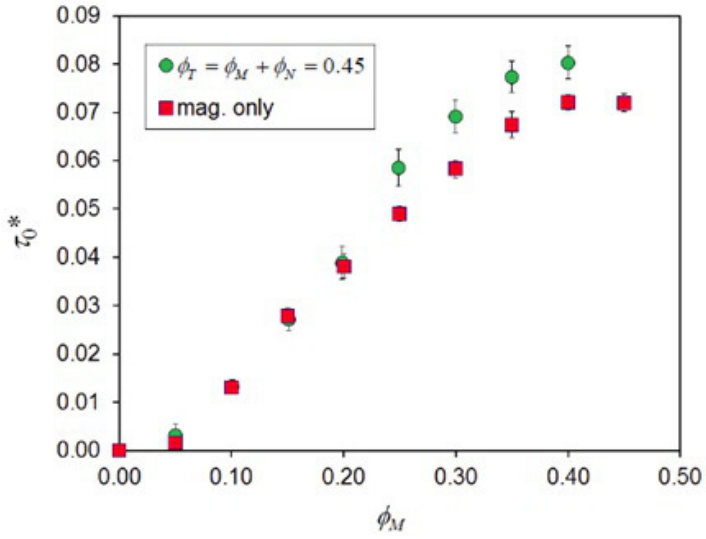

Figure 2.3: Yield stress at magnetic saturation as a function of magnetizable sphere concentration. Open circles represent monodisperse suspensions containing only magnetizable spheres. Open squares represent a mixture of glass and bimodally distributed iron particles such that the total volume fraction is $\phi_T = \phi_M + \phi_N = 0.45$ [Ulicny et al. (2010)].

2.3 Decrease of Off-State Viscosity with Coated Particles

Another proposed method for increasing the turn-up ratio is adding stearate and thiophosphates to the suspension [Ulicny et al. (2005a)]. These treatments become active on the surface of each particle [Klingenberg et al. (2010)]. By changing the surface chemistry of each particle, it appears that there is a drag reduction on the particles in the off-state while leaving the on-state properties unchanged. Figure 2.4 is a plot of stress as a function of shear rate for two types of suspensions. Open circles represent a suspension in which magnetizable particles have not been coated with stearate and thiophosphate. Open squares represent a suspension in which magnetizable particles have been coated with stearate and thiophosphate. The suspension that is untreated

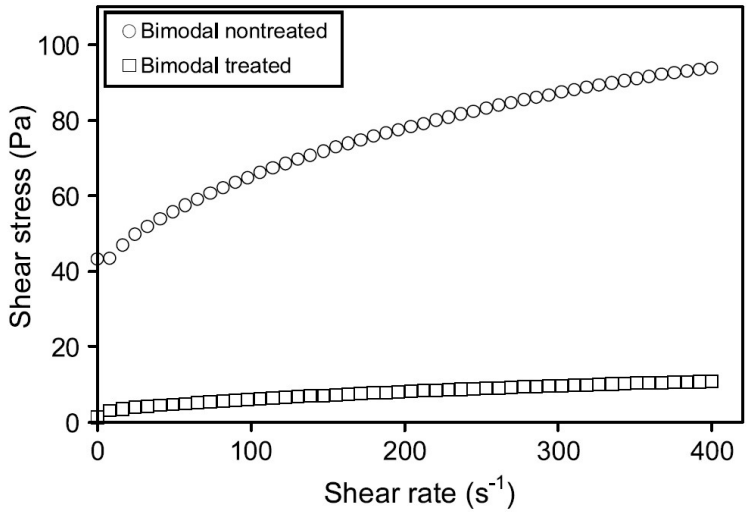

Figure 2.4: Shear stress vs. strain rate for particles treated with the stearate and thiophosphate coating and those without the coating in the off-state [Klingenberg et al. (2010)].

has a much higher shear stress than the suspension that has been treated with the stearate and thiophosphate compound. The coatings cause the interparticle distances to be larger, thereby decreasing the van der Waals attractions. The surface treatment described here has a similar effect on the rheological properties that has been observed by other surface treatments [Fang and H.J. Choi (2008); Aktary et al. (2001); Cho, M.S., S.T. Lim, I.B. Jang, H.J. Choi, M.S. Jhon (2004); Choi et al. (2005)]. This drag reduction leads to improved durability for the fluid. The surface coatings might also reduce the oxidation of the particles [Ulicny et al. (2005a); Ulicny et al. (2007)].

In order to help better understand these micro-scale phenomena, it is important to be able simulate the fluids on the micro-scale. To simulate on the micro scale, it is important to know which forces act on each particle as it moves through the fluid. In the high-field limit, only magnetostatic forces, short-range repulsive forces,

and Stokes' drag are considered to act on the sphere as it moves through the fluid. Stokes' drag is given by $F_{\text{drag}} = 6\pi\mu a$, where μ is the viscosity of the surrounding fluid, and a is the radius of the particle. In the high-field limit, the magnetostatic forces are considered to be much larger than the hydrodynamic interactions. Therefore, the free-draining limit can be used for these simulations. However, in the low-field limit, the magnetostatic forces do not dominate, and the hydrodynamic interactions must be included for an accurate simulation. To better understand what is happening on the macroscopic level in the low-field limit, it is important to understand what is happening in the regime where hydrodynamic interactions become important.

Ball and J.R. Melrose (1997) give a simulation technique for including hydrodynamic interactions in concentrated suspensions. Based on the Stokesian Dynamics (SD) techniques developed by Brady and G. Bossis (1988), Ball and J.R. Melrose (1997) note that in concentrated suspensions, the near-field lubrications interactions dominate the equations of motion. Therefore, far-field hydrodynamic interactions can be neglected. While this provides an important simplification to the traditional SD, the simulations are still very slow. Solving for the motion of the spheres requires solving a system of equations $6N \times 6N$ [Ball and J.R. Melrose (1997)]. Parallel computing offers a potential solution to improving the computational cost of these simulations. More work is needed in this area to better understand MR fluids in the low-field regime.

Chapter 3

Effect of Nonmagnetizable Spheres on the Structure of Magnetorheological Fluids

3.1 Introduction

Magnetorheological (MR) fluids are suspensions of magnetizable particles in a nonmagnetizable, viscous, continuous phase. Applying a magnetic field with a flux density on the order of 1 Tesla causes the stress at low deformation rates to increase by orders of magnitude. The field-induced stress increase is both fast and reversible. The magnetic field induces magnetostatic particle interactions which cause the particles to aggregate, changing the suspension from a fluid-like state to a solid-like state, with a magnetic field-dependent yield stress [Ginder (1996); Jolly et al. (1998)]. This dramatic field-induced change in rheological properties is often called the MR effect. The

tunable rheological properties make MR suspensions useful in numerous applications, including semiactive shock absorbers, clutches, actuators, servo valves, and precision polishing fluids [Jolly et al. (1998); Carlson and J.L. Sproston (2000); Klingenberg (2001)].

It is desirable to obtain the largest possible difference in rheological properties when the magnetic field is on (the "on-state") and when the magnetic field is off (the "off-state"). A large difference between off-state and on-state rheological properties allows for a large range of dynamic control, smaller devices and fluid volumes, and therefore reduced costs. Ulicny et al. (2010) showed experimentally that the field-induced yield stress of concentrated MR suspensions can be increased significantly by adding nonmagnetizable particles to the suspension. The yield stress of an MR suspension (at magnetic saturation) with an iron particle volume fraction of 0.30 was increased by 50% by adding glass beads at volume fraction of 0.15. Furthermore, it is possible to increase the field-induced yield stress by replacing a fraction of the magnetizable particles with an equivalent volume of nonmagnetizable particles. Similar magnitudes of yield stress enhancement were observed for a variety of different types of nonmagnetizable particles [Klingenberg and J.C. Ulicny (2011)]. This phenomenon has also been observed in simulations of MR suspensions composed of mixtures of magnetizable and nonmagnetizable spheres [Ulicny et al. (2010); Klingenberg and J.C. Ulicny (2011)]. An understanding of the mechanisms that produce this phenomenon is still lacking.

Unexpected increases in the field-induced stress of MR suspensions have been reported in experiments and simulations for other situations, which have been attributed to significant changes in the suspension microstructure. One such situation

is the transient stress increase attributed to the formation of lamellae, or sheet-like structures, in the plane of shear. When a sufficiently large magnetic or electric field is applied to a sheared MR or electrorheological (ER) suspension, respectively, the shear stress first increases rapidly, and then continues to increase much more slowly [Vieira et al. (2000); Ulicny et al. (2005b)]. The slow transient increase in stress is caused by the formation of lamellar structures [Henley and F.E. Filisko (1999); Tang, X. et al. (2000); Volkova et al. (1999); Vieira et al. (2000); Ulicny et al. (2005b)]. Lamellae formation with transient stress increases has also been observed in particle-level simulations of flowing MR and ER suspensions [Martin (2000); Kittipoomwong et al. (2008)].

Another unexpected field-induced stress increase was observed by Foister (1997), who reported experiments in which an MR suspension with bimodal particle size distribution possessed a larger field-induced stress than that of a monomodal suspension at the same volume fraction. Similar experimental results were reported by Weiss et al. (2000) and Ulicny et al. (2004). Particle-level simulations of MR suspensions by Kittipoomwong et al. (2005) also produced larger field-induced stresses for bidisperse suspensions than those obtained for monodisperse suspensions at the same volume fraction. Kittipoomwong et al. (2005) probed the structure of the suspensions to determine the mechanisms by which smaller particles cause the bidisperse suspensions to have a larger yield stress than monodisperse suspensions. They examined the volume fraction fluctuations, defined as $\langle\phi^2\rangle - \langle\phi\rangle^2$, where ϕ is the volume fraction, to assess the degree of heterogeneity of the different suspensions. The monodisperse suspensions exhibited the largest volume fraction fluctuations which means that the bidisperse suspensions were more homogeneous. Snapshots revealed that monodis-

perse suspensions tended to contain fewer, more globular clusters, whereas the bidisperse suspensions contained a greater number of more chain-like clusters. The larger number of clusters have less space between them, and thus smaller concentration fluctuations. The presence of more chain-like structures were quantified by calculating the pair distribution function as well as the components of the average mass moment tensor of clusters—both measures revealed a more anisotropic (i.e., more chain-like) structure for the bidisperse suspensions. It is thus apparent that more numerous chain-like structures produce larger stresses than fewer globular clusters [Klingenberg et al. (1991a); Kraynik et al. (1991); Gulley and R.T. Tao (1993); Anderson, R.A. (1994)].

In this article, we examine structure measures similar to those employed by Kittipoomwong et al. (2005) to determine if the dramatic rheological changes caused by the presence of nonmagnetizable spheres can be associated with significant changes in the microstructure, such as those described above. The model and simulation method are presented in the following section. Following the simulation method, new yield stress data are presented for simulations of both three-dimensional and monolayer suspensions. The measures of microstructure examined reveal that nonmagnetizable spheres only cause minor changes to the microstructure. This contrasts with the dramatic changes in microstructure presented by Kittipoomwong et al. (2005) for bidisperse suspensions. This suggests that the mechanisms by which nonmagnetizable spheres significantly influence the rheology of MR suspensions do not require a correspondingly significant change in the microstructure.

3.2 Model

Magnetorheological suspensions are treated as collections of magnetizable and nonmagnetizable spheres (monodisperse, diameter σ, magnetizable spheres with saturation magnetization M_s) immersed in a nonmagnetizable, Newtonian, incompressible, continuous phase (relative permeability $\mu = 1$, viscosity η_c), and subjected to a uniform magnetic field $\boldsymbol{H}_0 = H_0 \boldsymbol{e}_z$ [Klingenberg et al. (1991a); Kittipoomwong et al. (2005)].

The motion of the spheres can be described by Newton's equation of motion. Neglecting the inertia of sphere i gives

$$\boldsymbol{F}_i \left(\{ \boldsymbol{r}_j \} \right) = \boldsymbol{0} \tag{3.1}$$

where $\boldsymbol{F}_i \left(\{ \boldsymbol{r}_j \} \right)$ is the net force on sphere i. The net force has three contributions: the magnetostatic force, the short-range repulsive force, and the hydrodynamic force. The magnetostatic force on sphere i caused by sphere j is given by the point-dipole expression

$$\boldsymbol{F}_{ij}^{\text{mag.}} = F_0 \left(\frac{\sigma}{r_{ij}} \right)^4 \left[\left(3 \cos^2 \theta_{ij} - 1 \right) \boldsymbol{e}_r + \sin 2\theta_{ij} \boldsymbol{e}_\theta \right], \tag{3.2}$$

where r_{ij} is the distance between sphere i and sphere j, and θ_{ij} is the angle between the line-of-centers and the applied magnetic field. The magnitude of the force, F_0, is given by

$$F_0 = \begin{cases} \frac{3\pi}{16} \mu_0 \beta^2 H_0^2 \sigma^2 & \text{linear magnetization} \\ \frac{\pi}{48} \mu_0 \sigma^2 M_s^2 & \text{saturated magnetization} \end{cases}, \tag{3.3}$$

where $\beta = (\mu_p - \mu_c)/(\mu_p + 2\mu_c)$, μ_p is the relative permeability of the particle material,

μ_c is the relative permeability of the continuous phase, and μ_0 is the permeability of free space. To mimic a hard-sphere interaction between spheres i and j, a short-range repulsive force on sphere i caused by sphere j is given by

$$\boldsymbol{F}_{ij}^{\text{rep.}} = -F_0 \exp\left[\kappa\left(\sigma - r_{ij}\right)/\sigma\right] \boldsymbol{e}_r, \tag{3.4}$$

where κ characterizes the range of the repulsive force; $\kappa = 100$ for the results presented here. The spheres also experience a force due to hydrodynamic drag. Following the work of Klingenberg et al. (1991a) and Kittipoomwong et al. (2005), the hydrodynamic drag is treated as Stokes' drag

$$\boldsymbol{F}_i^{\text{hyd.}} = -3\pi\eta_c\sigma\left[\frac{d\boldsymbol{r}_i}{dt} - \boldsymbol{U}^{\infty}\left(\boldsymbol{r}_i\right)\right], \tag{3.5}$$

where $\boldsymbol{U}^{\infty}(\boldsymbol{r}_i)$ is the ambient fluid velocity evaluated at the particle center.

Equation 3.1 can be nondimensionalized using the following length, force, and time scales:

$$L_s = \sigma, \quad F_s = \frac{\pi}{48}\mu_0\sigma^2 M_s^2, \quad t_s = \frac{144\eta_c}{\mu_0 M_s^2}. \tag{3.6}$$

These scales allow Eq. 3.1 to be written

$$\frac{d\boldsymbol{r}_i^*}{dt^*} = \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\text{rep.}} + \boldsymbol{F}_i^{*,\text{wall}} + \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\text{mag.}} + \boldsymbol{U}^{*,\infty}, \tag{3.7}$$

where the asterisks denote dimensionless quantities.

The shear stress in the suspension is calculated by

$$\tau_{xz}^* = -\frac{1}{V^*}\sum_{i=1}^{N} z_i^* F_{x,i}^* \tag{3.8}$$

where $F^*_{x,i}$ is the x component of the total nonhydrodynamic force acting on sphere i.

3.3 Simulation Method

Magnetorheological suspensions were generated by randomly placing N neutrally buoyant spheres in a volume of size $L^*_x \times L^*_y \times L^*_z$. The spheres were bounded by solid surfaces at $z^* = \pm L^*_z/2$ and by periodic boundaries at $x^* = \pm L^*_x/2$ and $y^* = \pm L^*_y/2$.

The total volume fraction of spheres ϕ_T is

$$\phi_T = \phi_M + \phi_N \tag{3.9}$$

where ϕ_M is the volume fraction of magnetizable spheres, and ϕ_N is the volume fraction of nonmagnetizable spheres. Ten different initial configurations were created for each composition studied. The spheres in each configuration were randomly assigned as either magnetizable or nonmagnetizable (subject to the constraint of the specified values of ϕ_M and ϕ_N). Monolayer simulations were generated by placing N spheres in a cell $L^*_x \times L^*_z$ ($y^* = 0$ for all spheres). The total area fraction of spheres is given by $\phi^A_T = \phi^A_M + \phi^A_N$, where ϕ^A_M and ϕ^A_N are the area fractions of the magnetizable and nonmagnetizable spheres, respectively.

Spheres within 0.05σ of a bounding surface were considered stuck and assumed the lateral velocity of the surface; particles sticking to solid surfaces has been observed experimentally [Klingenberg and C.F. Zukoski (1990)]. Since the motion of each sphere in the z direction is still governed by Eq. 3.7, stuck spheres can be removed from the surface, and thus eventually move independently of the solid surface.

The suspensions were sheared by moving the surface located $z^* = +L_z^*/2$ in the positive x direction. The ambient velocity is thus $\boldsymbol{U}^{*,\infty}(\boldsymbol{r}) = \dot{\gamma}^*(z^* + L_z^*/2)\boldsymbol{e}_z$, where $\dot{\gamma}$ is the dimensionless shear rate. Sphere trajectories were determined by numerically integrating Eq. 3.7. Suspensions were sheared to a strain of $\gamma^* = 5.0$ at a strain rate of $\dot{\gamma}^* = 10^{-3}$. The positions of the spheres were saved every strain interval of 0.05. The dynamic yield stress was calculated using the "relaxation" method. Saved configurations were allowed to relax (with $\dot{\gamma}^* = 0$) to equilibrium. The average stress that is calculated with Eq. 3.8 using the relaxed configurations is equated with the dynamic yield stress. The dynamic yield stress is averaged over both configurations and strain interval $1 \leq \gamma \leq 5$. The dynamic yield stress calculated using this method is equivalent to that obtained from simulations at successively smaller shear rates followed by extrapolation to zero shear rate [Klingenberg et al. (1991a)].

The simulation cell size for three-dimensional simulations was $L_x^* = 10, L_y^* = 5, L_z^* = 5$. For the largest volume fraction studied, $\phi_T = 0.45$, the cell contained 215 spheres. The simulation cell size of the monolayer suspensions ($y_i^* = 0$) was $L_x^* = 30, L_z^* = 10$. For the largest area fraction studied, $\phi_T^A = 0.75$, the system contained 287 spheres.

3.4 Results and Discussion

3.4.1 Three-Dimensional Simulations

The dimensionless yield stress for three-dimensional simulations is plotted as a function of ϕ_N for various ϕ_M in Fig. 3.1. For $\phi_M < 0.20$, the nonmagnetizable spheres have no effect on the yield stress for the range of ϕ_N investigated. However, for values

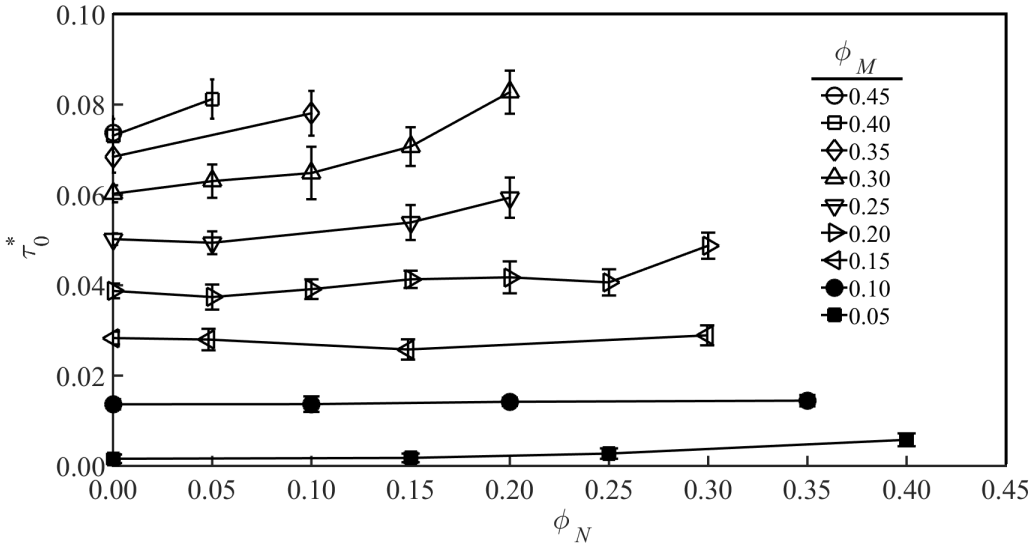

Figure 3.1: Dimensionless yield stress as a function of nonmagnetizable sphere volume fraction for three-dimensional simulations for various magnetizable sphere volume fractions.

of $\phi_M \geq 0.20$, the yield stress increases as ϕ_N is increased. Figure 3.1 illustrates that the yield stress can also be increased by replacing some magnetizable spheres with nonmagnetizable spheres.

3.4.2 Monolayer Simulations

Ulicny et al. (2010) reported simulation results for mixtures of magnetizable and nonmagnetizable spheres confined to monolayers. For a total area fraction of $\phi_T^A = 0.63$, the yield stress was independent of composition for the range of compositions investigated ($0.50 \leq \phi_M^A \leq 0.63$). Their simulations were performed with relatively small systems: $L_x^* = 15$, $L_z^* = 5$, and a total of only 60 spheres.

In contrast, we find that for larger monolayer systems ($L_x^* = 30$, $L_z^* = 10$), the presence of nonmagnetizable spheres produces larger yield stresses, as illustrated in Figs. 3.2 and 3.3, where the dimensionless yield stress is plotted as a function of ϕ_M^A.

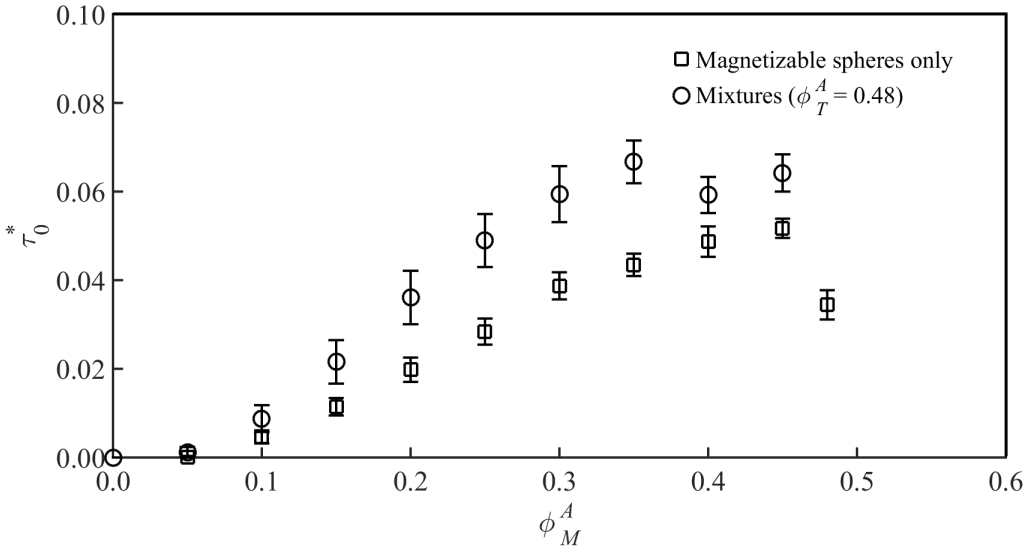

Figure 3.2: Dimensionless yield stress as a function of magnetizable sphere area fraction for monolayers with and without nonmagnetizable spheres. For monolayers with nonmagnetizable spheres, the total area fraction is fixed at $\phi_T^A = 0.48$.

In Fig. 3.2, the open squares represent results for monolayers containing only magnetizable spheres, and the open circles represent results for mixtures of magnetizable and nonmagnetizable spheres with a total area fraction fixed at $\phi_T^A = 0.48$. Figure 3.3 shows similar results, but for mixtures with a total area fraction fixed at $\phi_T^A = 0.75$.

The results in Figs. 3.2 and 3.3 illustrate that the yield stress in monolayer systems is larger for mixtures of magnetizable and nonmagnetizable spheres than it is for systems containing only magnetizable spheres at the same value of ϕ_M^A (≥ 0.10). In Fig. 3.4, the yield stress is plotted as a function of ϕ_N^A for various values of ϕ_M^A. Figures 3.2–3.4 also illustrate that the yield stress can be increased by replacing some magnetizable spheres with nonmagnetizable spheres.

The fact that Ulicny et al. (2010) reported no yield stress enhancement caused by adding nonmagnetizable spheres to monolayers, while we do observe an enhancement, can only be attributed to the sizes of the systems simulated (the same model is used

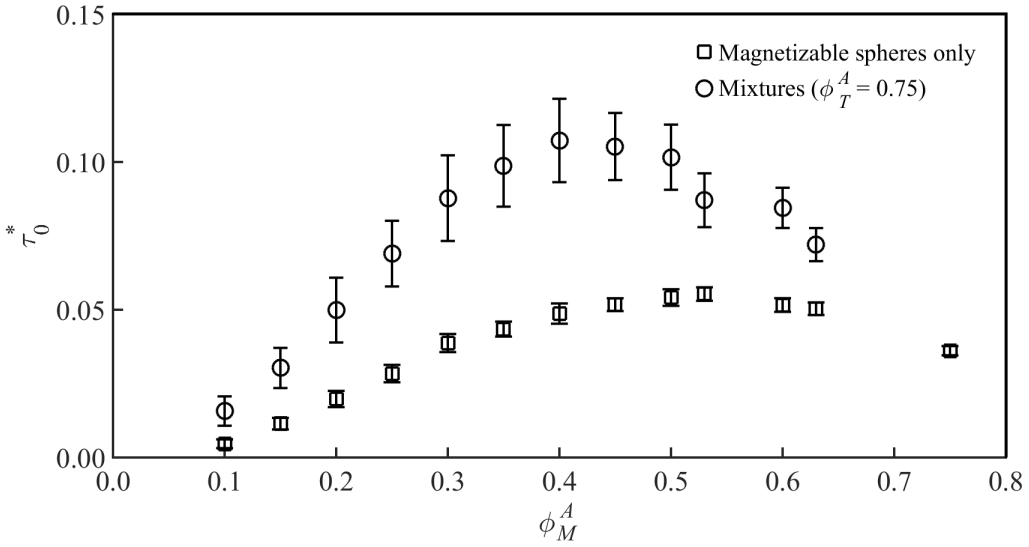

Figure 3.3: Dimensionless yield stress versus magnetizable sphere area fraction for multiple monolayer suspensions for monolayers with and without nonmagnetizable spheres. For monolayers with nonmagnetizable spheres, the total area fraction is fixed at $\phi_T^A = 0.75$.

in both studies)—apparently, the enhancement disappears when the system size is too small. A mechanistic explanation of this phenomenon is currently lacking.

It is now apparent that enhancement of the yield stress caused by nonmagnetizable spheres can be achieved in both three-dimensional and monolayer systems. This implies that the underlying mechanism cannot be a phenomenon only available in three-dimensional systems. Therefore, the mechanism of enhancement cannot be related to the formation of lamellar structures.

3.4.3 Microstructure Changes

Kittipoomwong et al. (2005) observed that the enhancement of the yield stress obtained for mixtures of large and small magnetizable spheres was associated with significant changes in the microstructure. Here we examine the same structural measures

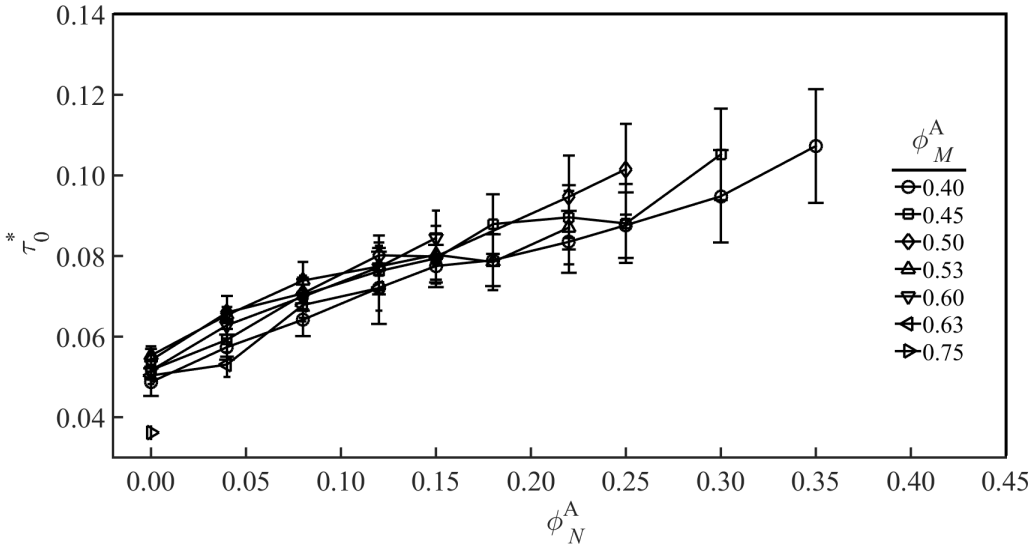

Figure 3.4: Dimensionless yield stress versus the volume fraction of nonmagnetizable spheres for multiple monolayer suspensions.

and the changes produced by the addition of nonmagnetizable spheres.

The fluctuation in the volume fraction of magnetizable spheres,

$$\sigma_M^2 \equiv \left\langle \phi_M^2 \right\rangle - \langle \phi_M \rangle^2 , \tag{3.10}$$

characterizes the degree of heterogeneity in the spatial distribution of magnetizable spheres. To evaluate $\langle \phi_M^2 \rangle$ and $\langle \phi_M \rangle$, the simulation cell was divided into cubes, each of side length L_B (for the results presented here, $L_B = 2.5$). The volume fraction of magnetizable spheres in each cube was calculated by determining the total volume of magnetizable spheres and dividing by the cube volume, L_B^3. The averages $\langle \phi_M^2 \rangle$ and $\langle \phi_M \rangle$ were equated with the averages of ϕ_M^2 and ϕ_M over all cubes. In monolayers, fluctuations in area fraction were considered. The fluctuations σ_M^2 were averaged over initial configurations and the strain interval $1 \leq \gamma \leq 5$.

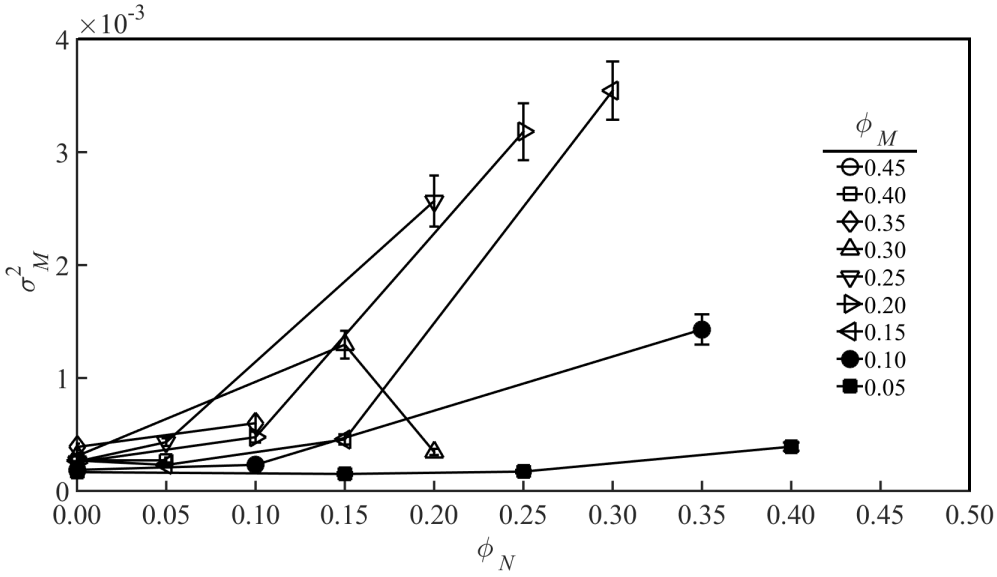

Figure 3.5: Fluctuation in volume fraction of magnetizable spheres as a function of the nonmagnetizable sphere volume fraction for three dimensional systems.

For a well-dispersed system, ϕ_M should be the same in all cubes, which gives $\langle \phi_M^2 \rangle = \langle \phi_M \rangle^2$ and $\sigma_M^2 = 0$. For a heterogeneous suspension, ϕ_M will vary from one cube to another, which gives $\sigma_M^2 > 0$.

The fluctuation in the volume fraction of magnetizable spheres is plotted as a function of ϕ_N for various ϕ_M in Fig. 3.5. For most values of ϕ_M, σ_M^2 increases as ϕ_N is increased. While the increase in σ_M^2 suggests that the distribution of magnetizable spheres becomes more heterogeneous when nonmagnetizable spheres are added to the suspension, the magnitude of the fluctuation is only 10^{-3}. This indicates that the nonmagnetizable spheres have insignificant impact on the homogeneity of three-dimensional systems.

The fluctuation in area fraction of magnetizable spheres in monolayers is plotted as a function of ϕ_N^A in Fig. 3.6 for different values of ϕ_M^A. For $\phi_N^A = 0$, σ_M^2 decreases slightly as ϕ_M^A is increased. As ϕ_N^A is increased, σ_M^2 also decreases. The suspension is

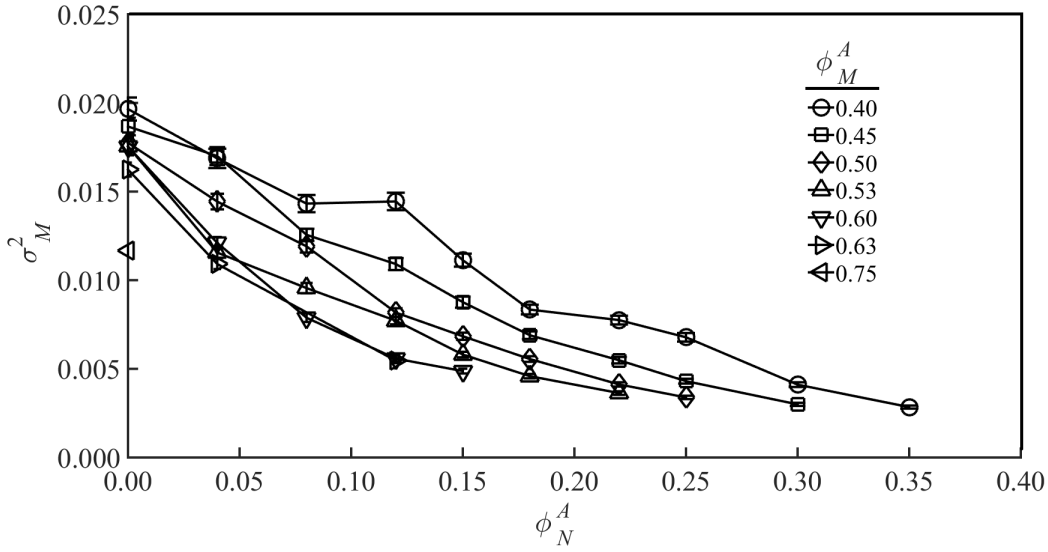

Figure 3.6: Fluctuation in area fraction of magnetizable spheres as a function of the nonmagnetizable sphere area fraction for monolayer systems.

most heterogeneous when nonmagnetizable spheres are absent in the suspension.

For both three-dimensional and monolayer systems, the fluctuation in the concentration of magnetizable spheres is small. Furthermore, although both types of suspensions exhibit an increase in yield stress when nonmagnetizable spheres are added, the impact on the concentration fluctuation is opposite—nonmagnetizable spheres increase σ_M^2 for the three-dimensional systems, and decrease σ_M^2 for the monolayer systems. We therefore conclude that the mechanism of yield stress enhancement is not associated with altering the degree of heterogeneity of the distribution of magnetizable spheres.

Kittipoomwong et al. (2005) discovered that the smaller spheres in bidisperse systems induce the larger spheres to form more chain-like, anisotropic structures than those formed in monodisperse suspensions. Changes in the degree of anisotropy caused by adding the small spheres were evident in snapshots of the simulations, in

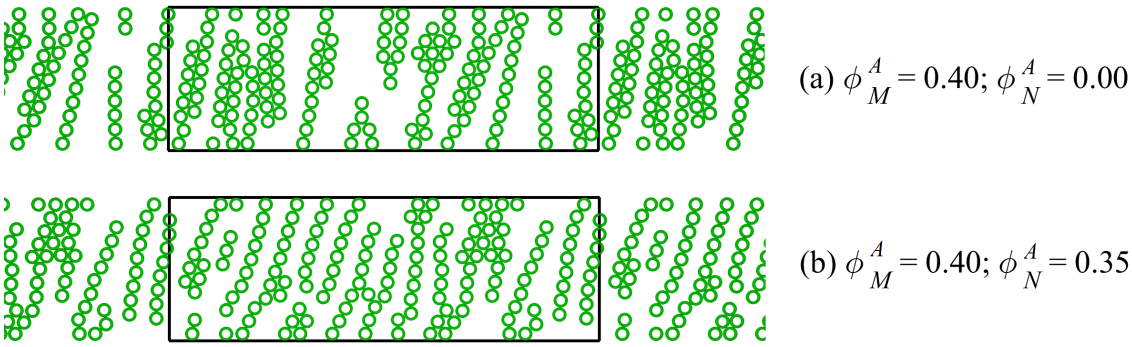

Figure 3.7: (a) Snapshot of a simulation with $\phi_M^A = 0.40$ and $\phi_N^A = 0.00$. Green circles represent magnetizable particles. (b) Snapshot of a simulation with $\phi_M^A = 0.40$ and $\phi_N^A = 0.35$ with the nonmagnetizable particles omitted for clarity.

the pair distribution function, and in the mass moment tensor.

Snapshots of monolayers containing magnetizable and nonmagnetizable spheres are shown in Fig. 3.7. Figure 3.7a depicts a monolayer containing only magnetizable spheres with area fraction $\phi_M^A = 0.40$. Figure 3.7b shows a monolayer mixture with area fractions $\phi_M^A = 0.40, \phi_N^A = 0.35$; the nonmagnetizable spheres have been omitted for clarity. The snapshots show that both systems are anisotropic and very similar. These and other snapshots suggest that the degree of anisotropy is not significantly altered by the addition of the nonmagnetizable spheres.

Next we consider the pair distribution functions. For a mixture of magnetizable and nonmagnetizable spheres, different distribution functions can be defined. For example, $g^{MM}(\boldsymbol{r})$ is the pair distribution function of magnetizable spheres given a magnetizable sphere at the origin.

Pair distribution functions $g^{MM}(\boldsymbol{r})$ in the velocity-velocity gradient (xz) plane are presented in Fig. 3.8 for three-dimensional suspensions with and without nonmagnetizable spheres. Figure 3.8(a) shows $g^{MM}(\boldsymbol{r})$ for a suspension of only magnetizable spheres with $\phi_M = 0.30$. Figure 3.8(b) shows $g^{MM}(\boldsymbol{r})$ for a mixture with $\phi_M = 0.30$

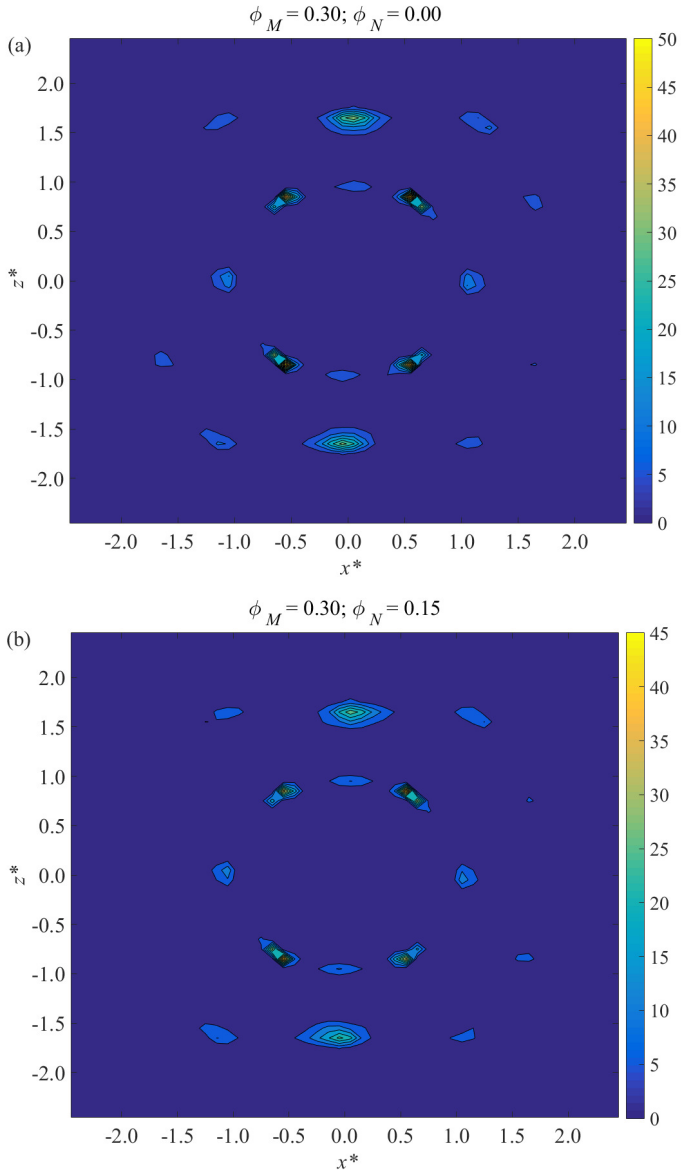

Figure 3.8: (a) The pair distribution function $g^{MM}(\boldsymbol{r})$ for volume fractions $\phi_M = 0.30$ and $\phi_N = 0.00$. (b) The pair distribution function $g^{MM}(\boldsymbol{r})$ for volume fractions $\phi_M = 0.30$ and $\phi_N = 0.15$

and $\phi_N = 0.15$. The two plots are quite similar, indicating that the presence of nonmagnetizable spheres does not qualitatively alter the microstructure of the magnetizable component. This is in stark contrast to the results reported by Kittipoomwong et al. (2005), where the pair distribution of large spheres was qualitatively altered by the addition of small spheres. In that case, the structure became more anisotropic, with new peaks and long-range structure appearing along the z axis.

The differences between Figs. 3.8(a) and (b) are illustrated in Fig. 3.9 where $\Delta g(\boldsymbol{r}) \equiv g^{MM}(\boldsymbol{r};\ \phi_M = 0.30, \phi_N = 0.15) - g^{MM}(\boldsymbol{r}; \phi_M = 0.30, \phi_N = 0)$ in the xz plane is presented. This difference illustrates a negligible change in the pair probability density near $(x^*, z^*) = (0, \pm 1)$, suggesting that the nonmagnetizable spheres do not cause an increase in the chain-like character of the microstructure. The decrease in probability density near the points $(x^*, z^*) = (\pm 0.52, \pm 0.75)$ and $(0, \pm 1.73)$ indicate a decrease in the population of triangular lattice structures (aligned with the flow direction). The magnitude of the decrease in g at the peaks is roughly 20% of the magnitude when only magnetizable spheres are present.

Pair distribution functions for monolayer systems are presented in Figs. 3.10–3.13. Figure 3.10(a) shows $g^{MM}(\boldsymbol{r})$ for a monolayer suspension of only magnetizable spheres with $\phi_M^A = 0.30$. Figure 3.10(b) shows $g^{MM}(\boldsymbol{r})$ for a mixture with $\phi_M^A = 0.30$ and $\phi_N^A = 0.18$. The two plots are similar, and indicate that the microstructure of magnetizable spheres is very chain-like, with high probability densities near $(x^*, z^*) = (0, \pm 1)$ and $(0, \pm 2)$. The addition of the nonmagnetizable spheres enhances these probability densities, and decreases the probability density at all positions away from the z axis.

The difference $\Delta g(\boldsymbol{r}) \equiv g^{MM}\left(\boldsymbol{r}; \phi_M^A = 0.30, \phi_N^A = 0.18\right) - g^{MM}\left(\boldsymbol{r}; \phi_M^A = 0.30, \phi_N^A = 0\right)$

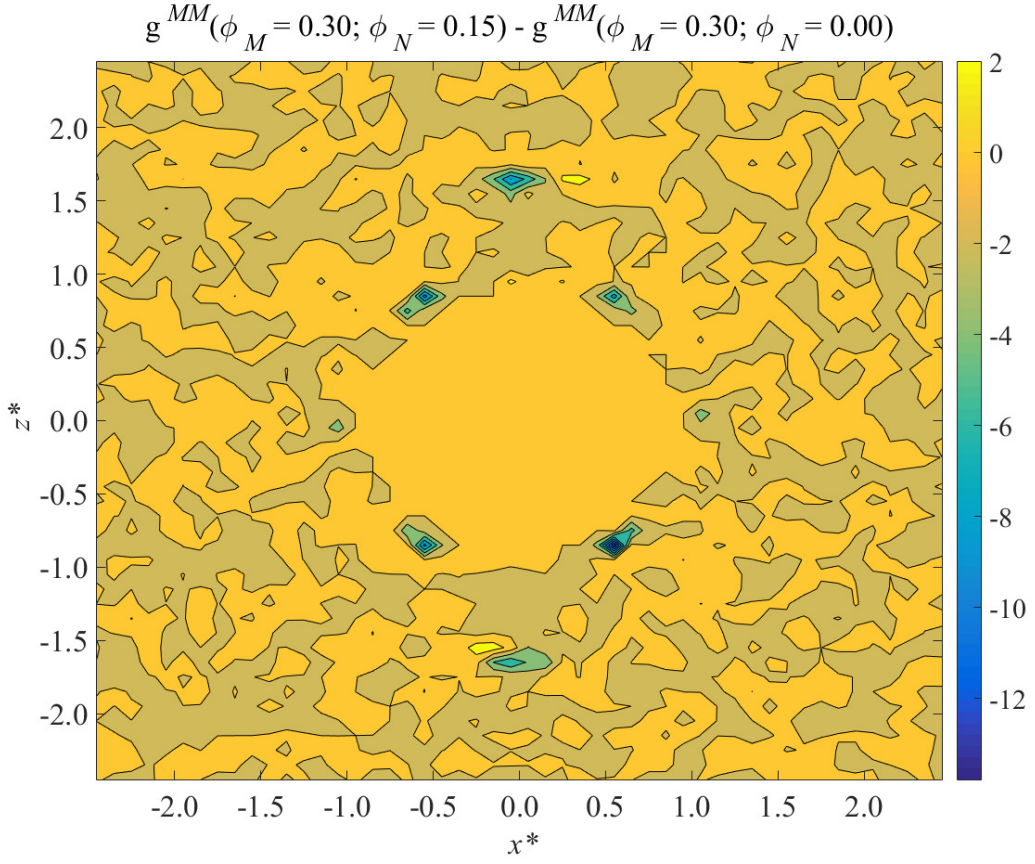

Figure 3.9: The pair distribution function difference $g^{MM}(\boldsymbol{r};\phi_M = 0.30, \phi_N = 0.15) - g^{MM}(\boldsymbol{r};\phi_M = 0.30, \phi_N = 0.00)$.

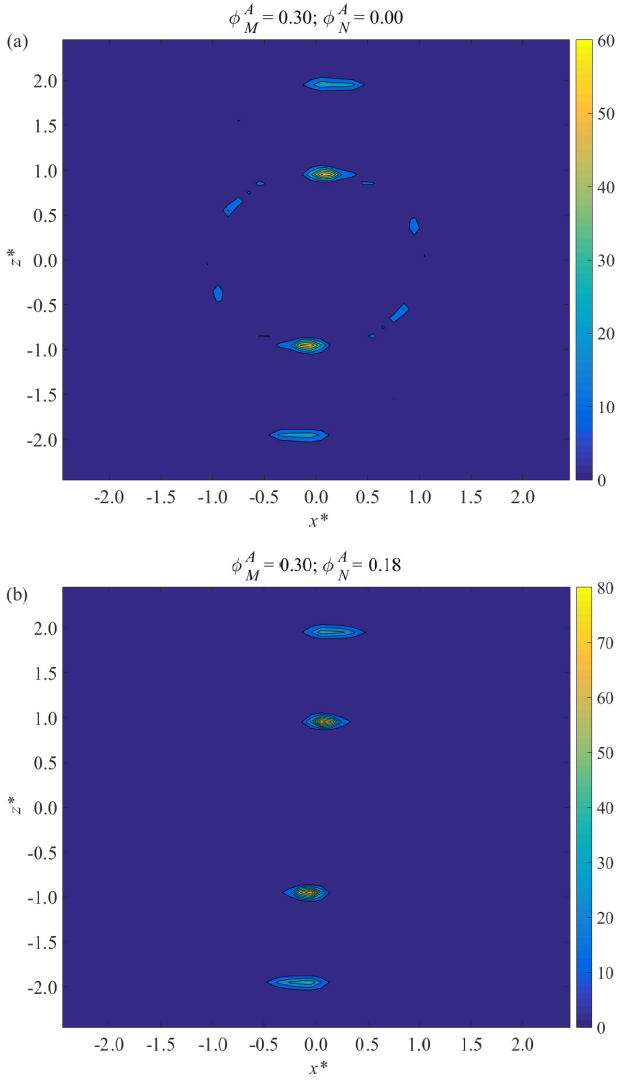

Figure 3.10: (a) The pair distribution function $g^{MM}(\boldsymbol{r})$ for area fractions $\phi_M^A = 0.30$ and $\phi_N^A = 0.00$. (b) The pair distribution function $g^{MM}(\boldsymbol{r})$ for area fractions $\phi_M^A = 0.30$; $\phi_N^A = 0.18$

.

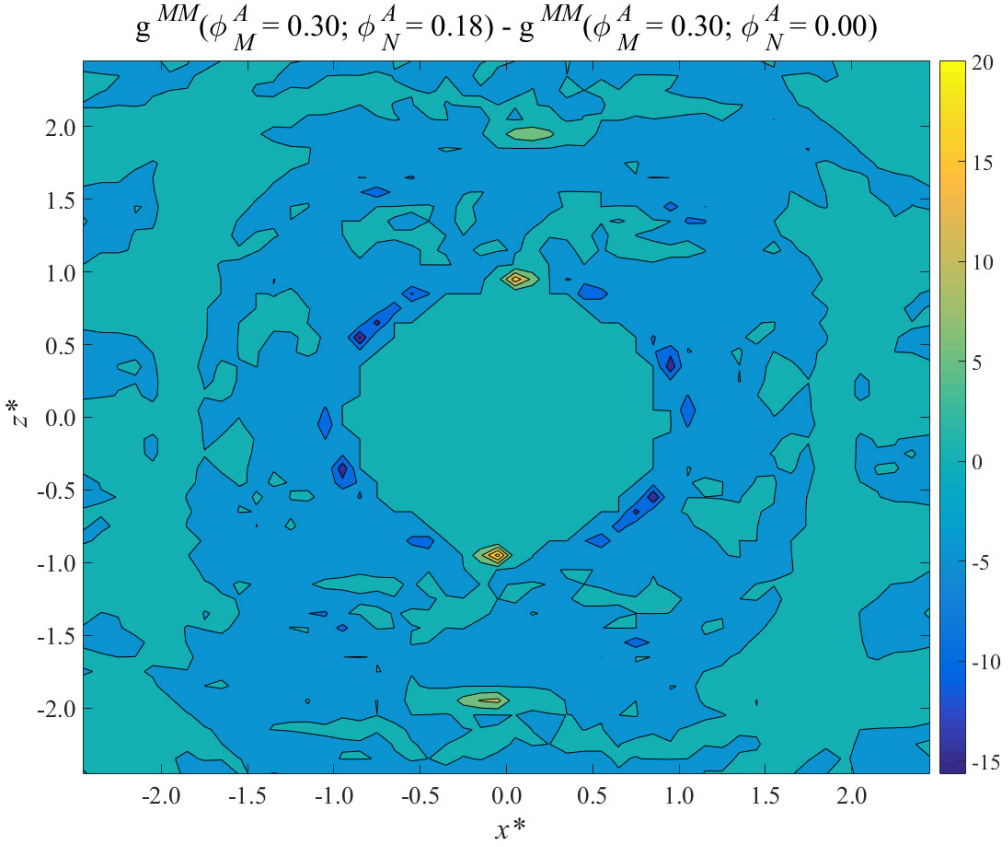

Figure 3.11: The pair distribution function difference $g^{MM}(\boldsymbol{r};\phi_M = 0.30, \phi_N = 0.15) - g^{MM}(\boldsymbol{r};\phi_M = 0.30, \phi_N = 0.00)$.

in the xz plane is presented in Fig. 3.11. This difference, like that demonstrated in Fig. 3.9, illustrates a negligible change in the pair probability density near $(x^*, z^*) = (0, \pm 1)$, suggesting that the nonmagnetizable spheres do not cause the an increase in the chain-like character of the microstructure in monolayers. Unlike the three-dimensional simulation, a decrease in probability density near the points $(x^*, z^*) = (\pm 0.52, \pm 0.75)$ and $(0, \pm 1.73)$ is absent.

Pair distribution functions for more concentrated monolayer systems are presented in Fig. 3.12. Figure 3.12(a) shows $g^{MM}(\boldsymbol{r})$ for a monolayer suspension of only mag-

netizable spheres with $\phi_M^A = 0.40$. Figure 3.12(b) shows $g^{MM}(\boldsymbol{r})$ for a mixture with $\phi_M^A = 0.40$ and $\phi_N^A = 0.35$. The difference $\Delta g(\boldsymbol{r}) \equiv g^{MM}\left(\boldsymbol{r}; \phi_M^A = 0.40, \phi_N^A = 0.35\right) - g^{MM}\left(\boldsymbol{r}; \phi_M^A = 0.40, \phi_N^A = 0\right)$ in the xz plane is presented in Fig. 3.13. For $\phi_M^A = 0.40$ and $\phi_N^A = 0$ (Fig. 3.12(a)), the microstructure exhibits chain-like character, with peaks near $(x^*, z^*) = (0, \pm 1)$ and $(0, \pm 2)$, as well as triangular lattice character, with peaks near $(x^*, z^*) = (\pm 0.52, \pm 0.75)$, $(\pm 1, 0)$, and $(0, \pm 1.73)$. Addition of the nonmagnetizable spheres increases the peak intensities near $(x^*, z^*) = (0, \pm 1)$ and decreases the peak in the triangular lattice positions (Fig. 3.12(b) and 3.13).

For both three-dimensional and monolayer suspensions, it is apparent that the addition of nonmagnetizable spheres consistently reduces the triangular lattice character of the microstructure of the magnetizable sphere component. The chain-like character is not enhanced in the three-dimensional systems, but it is in the monolayer systems—and all systems do exhibit an increase in yield stress upon addition of the nonmagnetizable spheres. Thus, in contrast to the results reported by Kittipoomwong et al. (2005) for bidisperse suspensions, the yield stress enhancement is not associated with an increase in the anisotropic, chain-like character of the microstructure. The enhancement does appear to be associated with a decrease in crystallinity, and thus an increase in the amorphous character. The mechanisms by which a more amorphous microstructure may produce a larger yield stress is not clear.

The anisotropy of the suspensions can also be quantified via the mass moment tensors of clusters within the suspension. To calculate the mass moment tensor, clusters of spheres were first identified. Two spheres were considered to be in direct contact, and thus in the same cluster, if their center-to-center separation was less than 1.05. The algorithm described by Sevick et al. (1988) was used to identify all spheres

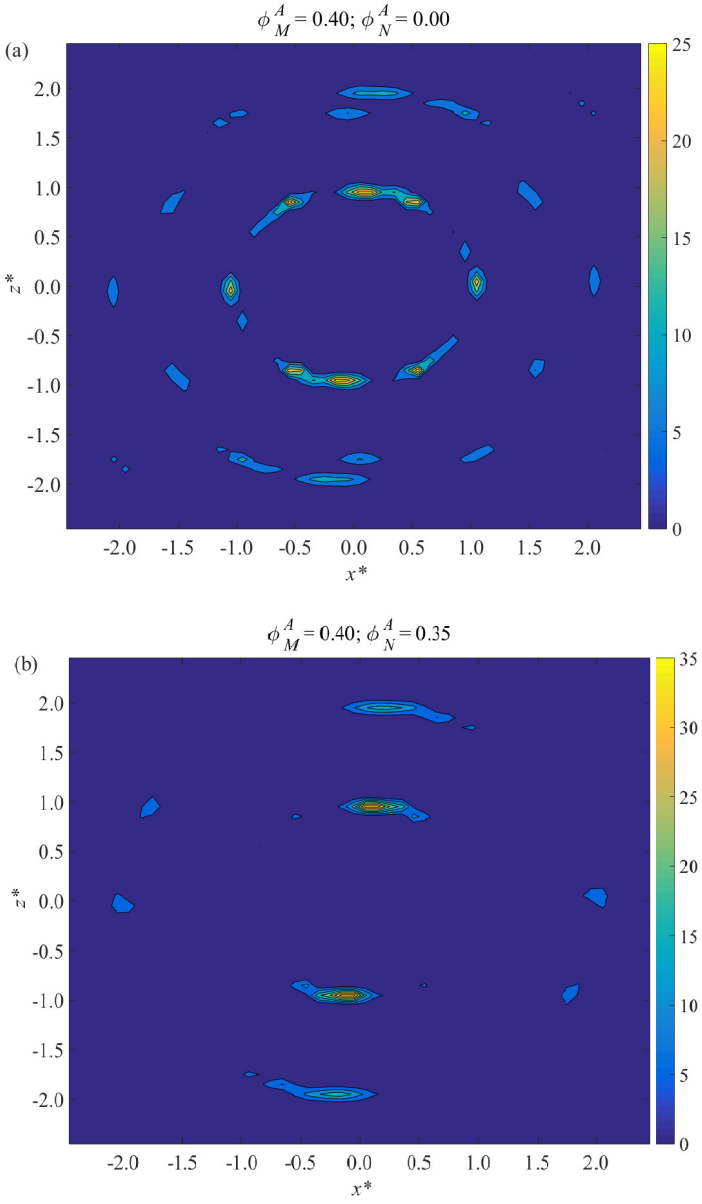

Figure 3.12: (a) The pair distribution function $g^{MM}(\boldsymbol{r})$ for area fractions $\phi_M^A = 0.40$. (b) The pair distribution function $g^{MM}(\boldsymbol{r})$ for area fractions $\phi_M^A = 0.40$ and $\phi_N^A = 0.35$.

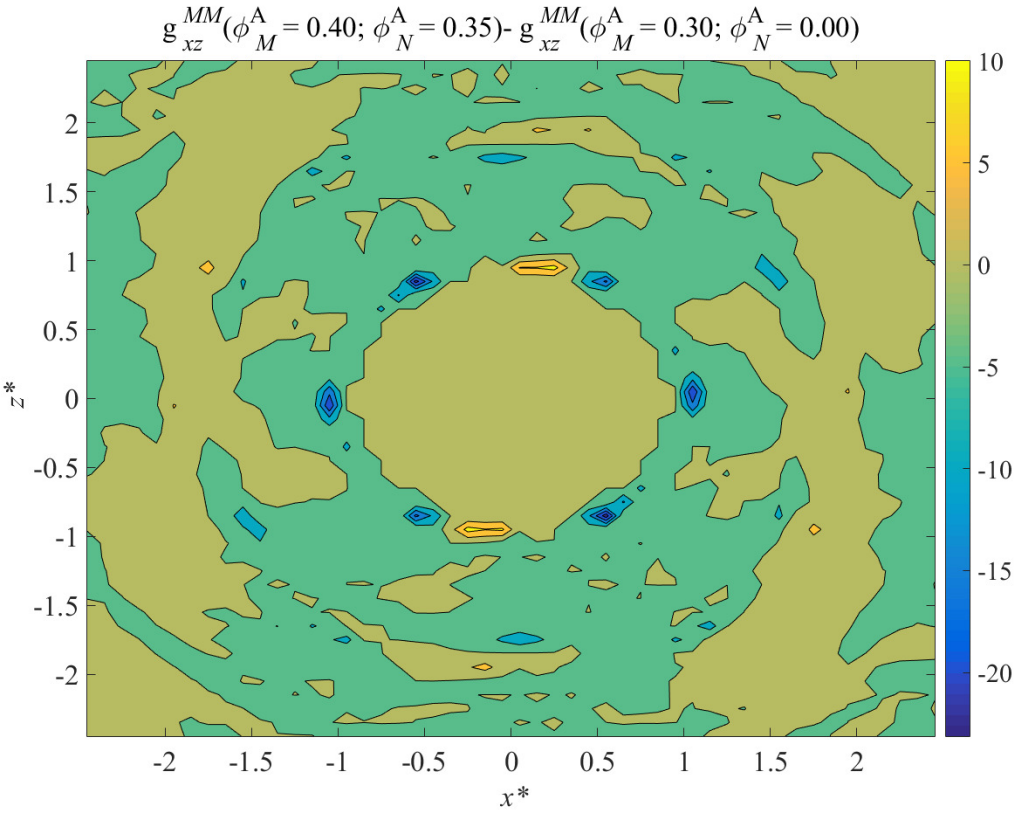

Figure 3.13: The difference $g^{MM}(\boldsymbol{r}; \phi_M^A = 0.40, \phi_N^A = 0.35) - g^{MM}(\boldsymbol{r}; \phi_M^A = 0.40, \phi_N^A = 0.00)$.

within the same cluster. Two types of clusters were identified: those containing only magnetizable spheres, and those containing magnetizable and nonmagnetizable spheres. After identifying the clusters, the mass moment tensor for the k^{th} cluster, which is composed of n_k, spheres is defined by

$$\mathbf{I}_k = \frac{\sum_{i=1}^{n_k} m_i \left(\boldsymbol{x}_i^* - \boldsymbol{x}_c^{*,k}\right)\left(\boldsymbol{x}_i^* - \boldsymbol{x}_c^{*,k}\right)}{\sum_{i=1}^{n_k} m_i}. \tag{3.11}$$

Here, $\boldsymbol{x}_i^*$ is the dimensionless location of the i^{th} sphere in cluster k, $\boldsymbol{x}_c^{*,k} = m_k^{-1}\sum_{i=1}^{n_k} m_i \boldsymbol{x}_i^*$ is the dimensionless center-of-mass of cluster k, and $m_i = \pi\rho\sigma_i^3/6$ is the mass of the i^{th} sphere with density ρ, treated here as a constant.

The anisotropy of a cluster can be quantified by the eigenvalues of the mass moment tensor. In order of decreasing magnitude, these eigenvalues can be labeled as I_1^k, I_2^k, and I_3^k (for monolayers, only I_1^k and I_2^k are needed). The ratio of the eigenvalues is given by

$$I_{\text{ratio}}^k = \frac{I_1^k}{\sqrt{\left(I_2^k\right)^2 + \left(I_3^k\right)^2}}, \tag{3.12}$$

for three dimensional systems, and $I_{\text{ratio}}^k = I_1^k/I_2^k$ for monolayer systems. For a chain of perfectly aligned spheres, $I_{\text{ratio}} \to \infty$.

In order to avoid a single chain of aligned spheres with $I_{\text{ratio}} \gg 1$ skewing the results, the average of the inverse of the eigenvalue ratios was calculated,

$$\left\langle I_{\text{ratio}}^{-1} \right\rangle = \frac{1}{N_C}\sum_{i=1}^{N_C}\left(I_{\text{ratio}}^{-1}\right), \tag{3.13}$$

where N_C is the total number of clusters in the suspension. These values were aver-

aged over both initial configurations and strain interval $1 \leq \gamma \leq 5$. The eigenvalue ratios were calculated for clusters containing both sphere types and clusters containing only magnetizable spheres.

The mass moment tensor eigenvalue ratio of three-dimensional suspensions is plotted as a function of ϕ_N for various ϕ_M in Fig. 3.14. Figure 3.14(a) shows $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ for clusters of magnetizable and nonmagnetizable spheres. Figure 3.14(b) shows $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ for clusters of only magnetizable spheres.

In both Fig. 3.14(a) and (b), $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ decreases as ϕ_M is increased. This indicates that more concentrated suspensions are less anisotropic. As ϕ_N is increased, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ decreases, even when the nonmagnetizable spheres are excluded from clusters. Kittipoomwong et al. (2005) found that for bidisperse systems, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ was orders of magnitude larger than $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ for monodisperse systems, which indicated that bidisperse suspensions are more anisotropic than monodisperse suspensions. In contrast, here we find that increasing ϕ_N causes $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ to decrease, creating a less anisotropic suspension.

Figure 3.15 shows $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ as a function of ϕ_M for three-dimensional suspensions. Open squares represent suspensions containing only magnetizable spheres. Open circles and triangles represent suspensions containing both types of spheres with $\phi_T = 0.45$. For the circles, nonmagnetizable spheres were included in the clusters; for the triangles, nonmagnetizable spheres were excluded from the clusters. At low ϕ_M, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ is smaller for suspensions that contain both types of spheres than for suspensions containing only magnetizable spheres, regardless of whether nonmagnetizable spheres are included or excluded from the clusters. As ϕ_M is increased, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ decreases for all data sets, which indicates that adding magnetizable spheres causes

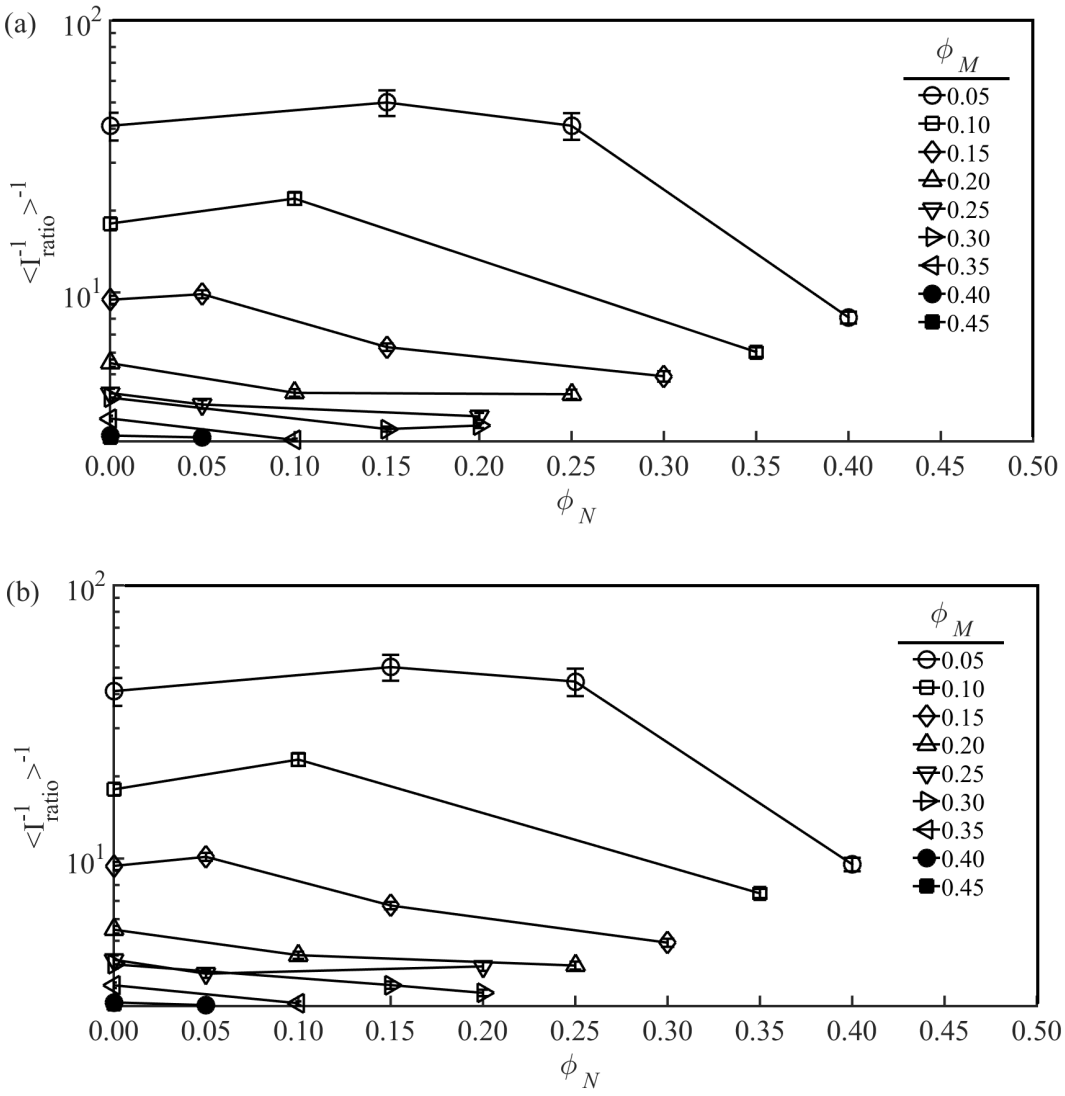

Figure 3.14: The mass moment tensor eigenvalue ratio as a function of ϕ_N for several different values of ϕ_M. (a) Nonmagnetizable particles are included in the clusters. (b) Nonmagnetizable are excluded from the clusters.

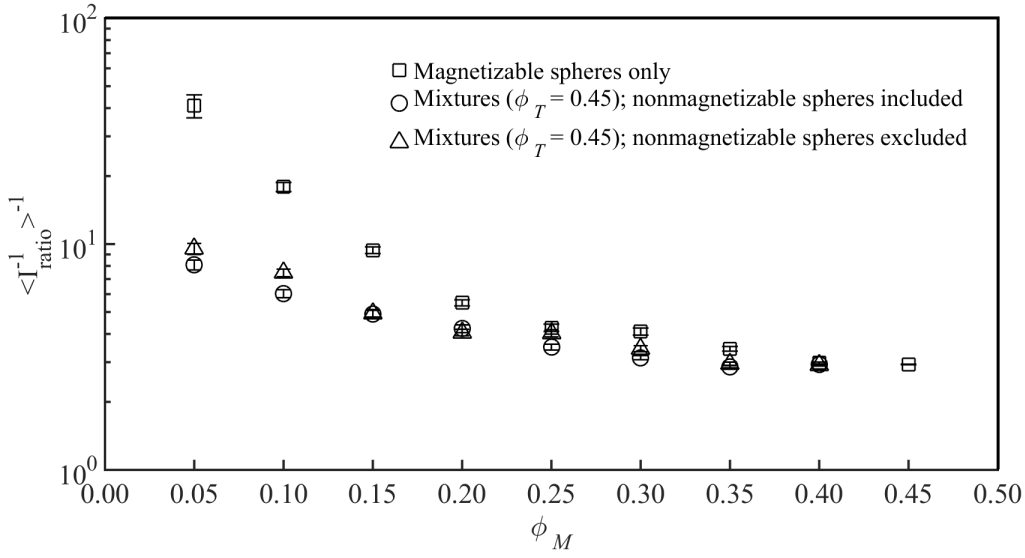

Figure 3.15: $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ as a function of ϕ_M for suspensions containing only magnetizable spheres (squares), and for mixtures with $\phi_T = 0.45$. For the circles, the nonmagnetizable spheres were included in the clusters; for the triangles, nonmagnetizable spheres were excluded.

the suspension to be less anisotropic.

Figure 3.16 shows $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ as a function of ϕ_N^A for various ϕ_M^A for monolayer suspensions. In Fig. 3.16(a), both magnetizable and nonmagnetizable spheres were included in the clusters. In Fig. 3.16(b), the nonmagnetizable spheres were excluded from the clusters.

In Fig.3.16(a), for $\phi_N^A = 0$, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ decreases as ϕ_M^A is increased. As ϕ_N^A is increased, all values of the eigenvalue ratio remain in the range $5 \leq \left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1} \leq 10$. This indicates that adding nonmagnetizable spheres does not cause the suspension to become more anisotropic. However, when the nonmagnetizable spheres are excluded from the clusters (in Fig. 3.16(b)) the eigenvalue ratio is independent of ϕ_N^A. As ϕ_M^A is decreased, the eigenvalue ratio increases, indicating that more chain-like structures appear at lower concentrations of magnetizable spheres. We also note that

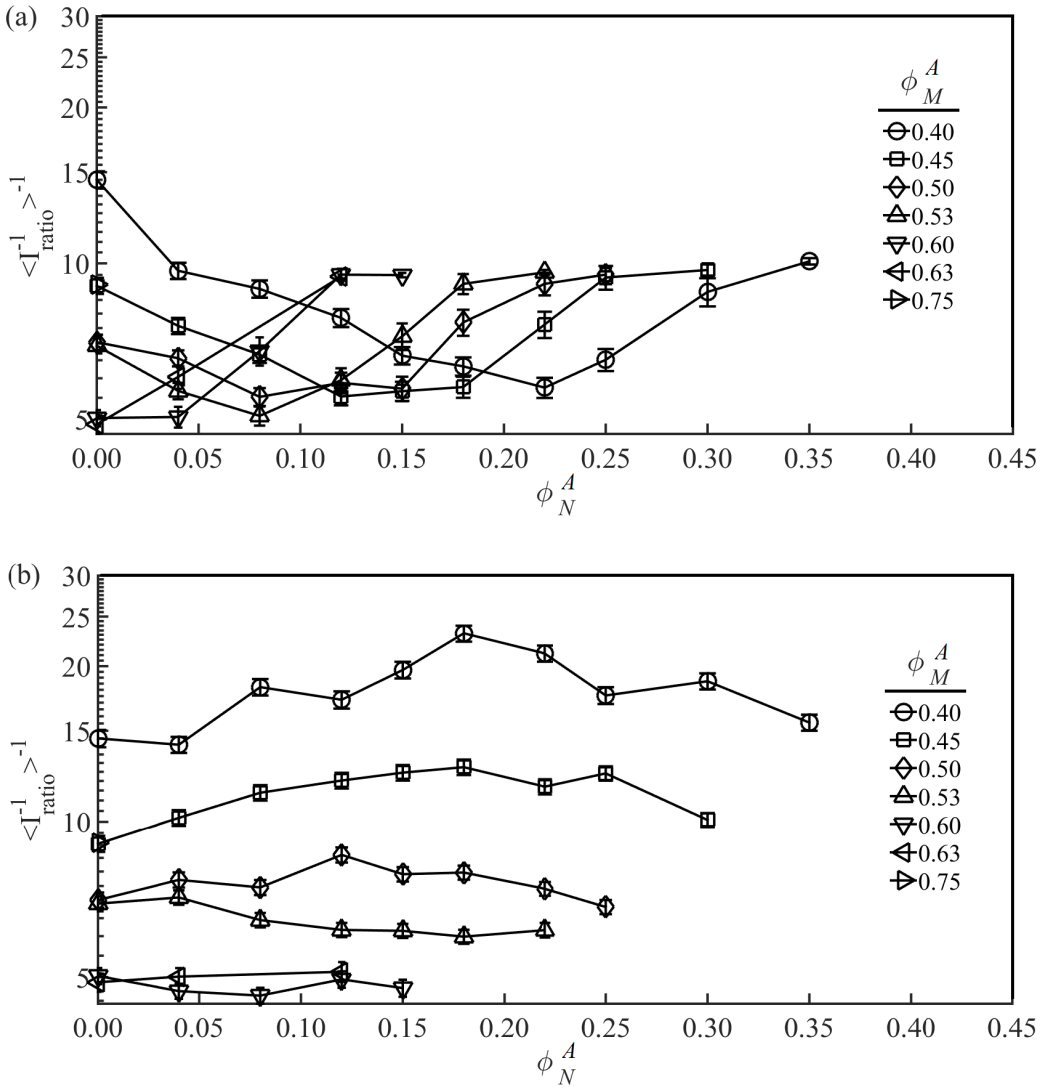

Figure 3.16: The mass moment tensor eigenvalue ratio as a function of ϕ_N for several different values ϕ_M^A(a) Nonmagnetizable particles are included in the clusters. (b) Nonmagnetizable are excluded from the clusters.

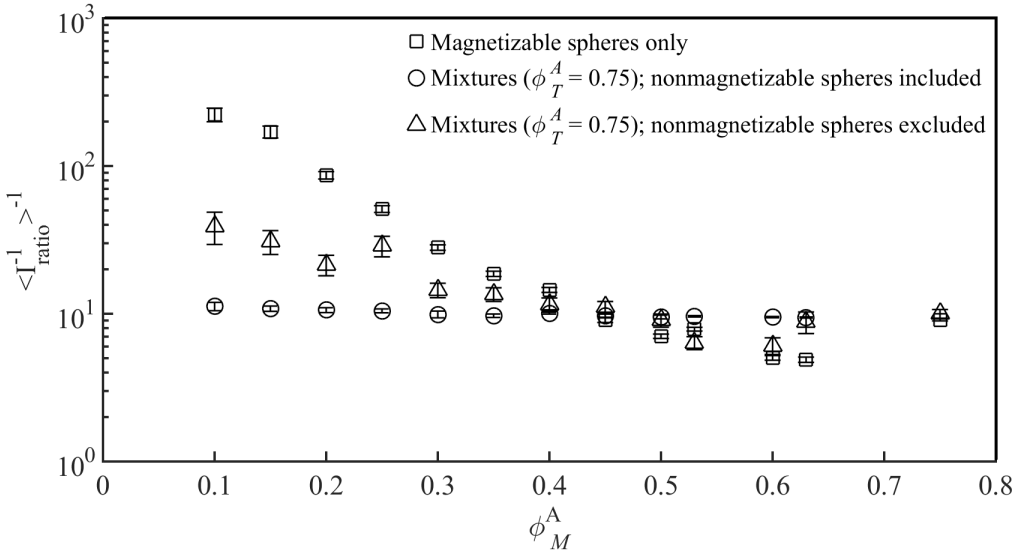

Figure 3.17: $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ as a function of ϕ_M^A for suspensions containing only magnetizable spheres (squares), and for mixtures with $\phi_T^A = 0.75$. For the circles, the nonmagnetizable spheres were included in the clusters; for the triangles, nonmagnetizable spheres were excluded.

the dependence of $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ on ϕ_M^A is much weaker than that reported for bidisperse suspensions [Kittipoomwong et al. (2005)].

The mass moment tensor eigenvalue ratio for monolayer simulations is plotted as a function of ϕ_M^A in Fig. 3.17. Open squares represent suspensions containing only magnetizable spheres. Open circles represent suspensions containing both types of spheres with $\phi_T^A = 0.75$; nonmagnetizable spheres are included in the clusters. Open triangles represent the same suspensions as the open circles, but nonmagnetizable spheres are excluded from the clusters. At low ϕ_M^A, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ is smaller for mixtures than for suspensions of only magnetizable spheres, regardless of whether or not nonmagnetizable spheres are included in the clusters. For suspensions containing only magnetizable spheres, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ decreases as ϕ_M^A is increased. When the nonmagnetizable spheres are included in the clusters, $\left\langle I_{\text{ratio}}^{-1} \right\rangle^{-1}$ remains constant. The data in

Fig. 3.17 suggests that suspensions forms fewer chain-like clusters of spheres when nonmagnetizable particles are added, as well as when ϕ_M^A is increased.

3.5 Conclusions

We have employed a particle-level simulation technique to probe the effect of nonmagnetizable spheres on MR suspensions that contain a mixture of magnetizable and nonmagnetizable spheres. Monolayer simulations exhibit an increase in yield stress when nonmagnetizable spheres are added, which is consistent with experimental results for three-dimensional systems, and in contrast to previously reported results for monolayers [Ulicny et al. (2010)]. We characterized the microstructure of the suspensions by several measures, including volume fraction fluctuations, pair distribution functions, and eigenvalues of the second-order mass moment tensor. We find that nonmagnetizable spheres cause different microstructure changes in monolayer and three-dimensional suspensions. In addition, the microstructure changes are much smaller than those reported for bidisperse suspensions [Kittipoomwong et al. (2005)]. Therefore, microstructure changes caused by the addition of nonmagnetizable spheres do not appear to directly cause the yield stress enhancement.

Chapter 4

Effect of Nonmagnetizable Spheres on the Forces of Magnetorheological Fluids

4.1 Introduction

Magnetorheological (MR) fluids are suspensions of magnetizable particles in a nonmagnetizable, viscous, continuous phase. Application of a magnetic field with a flux density on the order of 1 Tesla causes the stress at low deformation rates to increase by orders of magnitude. The field-induced stress increase is both fast and reversible. The magnetic field induces magnetostatic particle interactions which cause the particles to aggregate, changing the suspension from a fluid-like state to a solid-like state, with a magnetic field-dependent yield stress [Ginder (1996); Jolly et al. (1998)]. This dramatic field-induced change in rheological properties is often called the MR ef-

fect. The tunable rheological properties make MR suspensions useful in numerous applications, including semiactive shock absorbers, clutches, actuators, servo valves, and precision polishing fluids [Jolly et al. (1998); Carlson and J.L. Sproston (2000); Klingenberg (2001)].

It is desirable to obtain the largest possible difference in rheological properties when the magnetic field is on (the "on-state") and when the magnetic field is off (the "off-state"). A large difference between off-state and on-state rheological properties allows for a large range of dynamic control, smaller devices and fluid volumes, and therefore reduced costs. Ulicny et al. (2010) showed experimentally that the field-induced yield stress of concentrated MR suspensions can be increased significantly by adding nonmagnetizable particles to the suspension. The yield stress of an MR suspension (at magnetic saturation) with an iron particle volume fraction of 0.30 was increased by 50% by adding glass beads at a volume fraction of 0.15. Furthermore, it is possible to increase the field-induced yield stress by replacing a fraction of the magnetizable particles with an equivalent volume of nonmagnetizable particles. Similar magnitudes of yield stress enhancement were observed for a variety of different types of nonmagnetizable particles [Klingenberg and J.C. Ulicny (2011)]. This phenomenon has also been observed in simulations of MR suspensions composed of mixtures of magnetizable and nonmagnetizable spheres [Ulicny et al. (2010); Klingenberg and J.C. Ulicny (2011)]. An understanding of the mechanisms that produce this phenomenon is still lacking.

In Chapter 3 it was shown that, unlike other systems in which shear stresses increased, the increase in shear stress resulting from the addition of nonmagnetizable particles is not associated with a significant change in the microstructure. Further-

more, subtle changes in measures of the microstructure differ qualitatively in monolayer and 3D systems. These observations suggest that the increase in stress observed by adding nonmagnetizable particles is not caused by a change in microstructure.

Dynamic measurements are common tools for probing the mechanisms of rheological behavior for complex fluids. We use large amplitude oscillatory shear (LAOS) to investigate the mechanisms that cause the yield stress increase for MR fluids that contain nonmagnetizable particles. Suspensions were sheared in the limit of zero shear rate using a relaxation method Klingenberg et al. (1991b). Configurations saved during shear were subjected to LAOS strain sweeps. The presence of nonmagnetizable spheres causes the shear modulus in the linear regime to increase, without significantly altering the critical strain that marks the transition to nonlinear deformation. This suggests that the nonmagnetizable spheres enhance stress transfer by increasing the stiffness of the field-induced structures, as opposed to stabilizing the structures.

We show that the nonmagnetizable spheres produce stresses by participating in repulsive force chains. These force chains are roughly aligned with the compression axis of the simple shear flow, and contain nonmagnetizable as well as magnetizable spheres. The ability of the nonmagnetizable spheres to transmit stress through purely repulsive forces is similar to that found in jammed, hard-sphere suspensions [Farr et al. (1997); Cates et al. (1998)]. We illustrate the repulsive force chain formation with snapshots of sheared suspensions, draw analogy to previously reported jamming phenomena by considering the stress versus strain behavior, and characterize the resulting contribution to the shear stress by examining particle stresses and repulsive force statistics.

4.2 Model

Magnetorheological suspensions are treated as collections of magnetizable and nonmagnetizable spheres (monodisperse, diameter σ, magnetizable spheres with saturation magnetization M_s) immersed in a nonmagnetizable, Newtonian, incompressible, continuous phase (relative permeability $\mu = 1$, viscosity η_c), and subjected to a uniform magnetic field $\boldsymbol{H}_0 = H_0 \boldsymbol{e}_z$ [Klingenberg et al. (1991a); Kittipoomwong et al. (2005)].

The motion of the spheres can be described by Newton's equation of motion. By neglecting the inertia of sphere i, the equation of motion for sphere i can be written

$$\boldsymbol{F}_i\left(\{\boldsymbol{r}_j\}\right) = \boldsymbol{0} \tag{4.1}$$

where $\boldsymbol{F}_i\left(\{\boldsymbol{r}_j\}\right)$ is the net force on sphere i. The net force has three contributions: the magnetostatic force, the short-range repulsive force, and the hydrodynamic force. The magnetostatic force on sphere i caused by sphere j is given by the point-dipole expression

$$\boldsymbol{F}_{ij}^{\text{mag.}} = F_0 \left(\frac{\sigma}{r_{ij}}\right)^4 \left[\left(3\cos^2\theta_{ij} - 1\right)\boldsymbol{e}_r + \sin 2\theta_{ij}\boldsymbol{e}_\theta\right], \tag{4.2}$$

where r_{ij} is the distance between sphere i and sphere j, and θ_{ij} is the angle between the line-of-centers and the applied magnetic field. The magnitude of the force, F_0, is given by

$$F_0 = \begin{cases} \frac{3\pi}{16}\mu_0\beta^2 H_0^2\sigma^2 & \text{linear magnetization} \\ \frac{\pi}{48}\mu_0\sigma^2 M_s^2 & \text{saturated magnetization} \end{cases}, \tag{4.3}$$

where $\beta = (\mu_p - \mu_c)/(\mu_p + 2\mu_c)$, μ_p is the relative permeability of the particle material,

μ_c is the relative permeability of the continuous phase, and μ_0 is the permeability of free space. To mimic a hard-sphere interaction between spheres i and j, a short-range repulsive force on sphere i caused by sphere j is given by

$$\boldsymbol{F}_{ij}^{\text{rep}} = -F_0 \exp\left[\kappa\left(\sigma - r_{ij}\right)/\sigma\right] \boldsymbol{e}_r, \tag{4.4}$$

where κ characterizes the range of the repulsive force ($\kappa = 100$ in this study). The spheres also experience a force due to hydrodynamic drag. Following the work of Klingenberg et al. (1991a) and Kittipoomwong et al. (2005), the hydrodynamic drag is treated as Stokes' drag

$$\boldsymbol{F}_i^{\text{hyd}} = -3\pi\eta_c\sigma\left[\frac{d\boldsymbol{r}_i}{dt} - \boldsymbol{U}^{\infty}\left(\boldsymbol{r}_i, t\right)\right], \tag{4.5}$$

where $\boldsymbol{U}^{\infty}(\boldsymbol{r}_i, t)$ is the ambient fluid velocity evaluated at the particle center.

Equation 4.1 can be nondimensionalized using the following length, force, and time scales:

$$L_s = \sigma, \quad F_s = \frac{\pi}{48}\mu_0\sigma^2 M_s^2, \quad t_s = \frac{144\eta_c}{\mu_0 M_s^2}. \tag{4.6}$$

These scales allow Eqn. 4.1 to be written

$$\frac{d\boldsymbol{r}_i^*}{dt^*} = \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\text{rep}} + \boldsymbol{F}_i^{*,\text{wall}} + \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\text{mag}} + \boldsymbol{U}^{*,\infty}\left(\boldsymbol{r}_i, t\right), \tag{4.7}$$

where the asterisks denote dimensionless quantities.

The shear stress in the suspension is

$$\tau_{xz}^* = -\frac{1}{V^*}\sum_{i=1}^{N} z_i^* F_{x,i}^* \tag{4.8}$$

where $F^*_{x,i}$ is the x component of the total nonhydrodynamic force acting on sphere i.

4.3 Simulation Methods

Magnetorheological suspensions were generated by randomly placing N neutrally buoyant spheres in a volume of size $L^*_x \times L^*_y \times L^*_z$. The spheres were bounded by solid surfaces at $z^* = \pm L^*_z/2$ and by periodic boundaries at $x^* = \pm L^*_x/2$ and $y^* = \pm L^*_y/2$. Interparticle forces are evaluated within a cutoff radius of $r^* = 2.5$.

The total volume fraction of spheres ϕ_T is

$$\phi_T = \phi_M + \phi_N \tag{4.9}$$

where ϕ_M is the volume fraction of magnetizable spheres, and ϕ_N is the volume fraction of nonmagnetizable spheres. Ten different initial configurations were created for each composition studied. The spheres in each configuration were randomly assigned as either magnetizable or nonmagnetizable (subject to the constraint of the specified values of ϕ_M and ϕ_N). Monolayer simulations were generated by placing N spheres in a cell $L^*_x \times L^*_z$ ($y^* = 0$ for all spheres). The total area fraction of spheres is given by $\phi^A_T = \phi^A_M + \phi^A_N$, where ϕ^A_M and ϕ^A_N are the area fractions of the magnetizable and nonmagnetizable spheres, respectively.

Spheres within 0.05σ of a bounding surface were considered stuck and assumed the lateral velocity of the surface; particles sticking to solid surfaces has been observed experimentally [Klingenberg and C.F. Zukoski (1990)]. Since the motion of each sphere in the z direction is still governed by Eq. 4.7, stuck spheres can be removed

from the surface, and thus eventually move independently of the solid surface.

The suspensions were sheared by moving the surface located $z^* = +L_z^*/2$ in the positive x direction. The ambient velocity is thus $\boldsymbol{U}^{*,\infty}(\boldsymbol{r}) = \dot{\gamma}^*(z^* + L_z^*/2)\boldsymbol{e}_z$, where $\dot{\gamma}$ is the dimensionless shear rate. Sphere trajectories were determined by numerically integrating Eq. 3.7. Suspensions were sheared to a strain of $\gamma^* = 5.0$ at a strain rate of $\dot{\gamma}^* = 10^{-3}$. The positions of the spheres were saved every strain interval of 0.05. The dynamic yield stress was calculated using the "relaxation" method. Saved configurations were allowed to relax (with $\dot{\gamma}^* = 0$) to equilibrium. The average stress that is calculated with Eq. 3.8 using the relaxed configurations is equated with the dynamic yield stress. The dynamic yield stress is averaged over both configurations and strain interval $1 \leq \gamma \leq 5$. The dynamic yield stress calculated using this method is equivalent to that obtained from simulations at successively smaller shear rates followed by extrapolation to zero shear rate [Klingenberg et al. (1991a)].

The relaxed configurations were sheared by oscillating the surface located $z^* = +L_z^*/2$ in the x direction. The ambient velocity is thus

$$\boldsymbol{U}^{\infty}(\boldsymbol{r}, t) = \omega^* \gamma_0 \left(z_i^* + L_z^*/2\right) \cos(\omega^* \mathrm{t}^*) \mathrm{e}_\mathrm{x} \tag{4.10}$$

where ω^* is the dimensionless oscillation frequency ($\omega^* = 0.01$ in this study), γ_0 is the oscillation amplitude, and t^* is the dimensionless time. The equations of motion were then solved by the simulation method outlined by Klingenberg et al. (1989). Suspensions were oscillated for eight periods. The viscoelastic properties were calculated by Fourier transforming the last five periods of $\tau_{xz}^*(t^*)$. Storage moduli were averaged over initial configurations and the strain interval $1 \leq \gamma \leq 5$.

The simulation cell size for three-dimensional simulations was $L_x^* = 10, L_y^* = 5, L_z^* = 5$. For the largest volume fraction studied, $\phi_T = 0.45$, the cell contained 215 spheres. The simulation cell size of the monolayer suspensions ($y_i^* = 0$) was $L_x^* = 30, L_z^* = 10$. For the largest area fraction studied, $\phi_T^A = 0.75$, the system contained 287 spheres.

4.4 Discussion

In Fig. 4.1, $G_1'^*$ is plotted as a function of γ_0 for LAOS simulations of monolayer suspensions. Open squares represent results for mixtures of magnetizable and non-magnetizable spheres with area fractions $\phi_M^A = 0.45$ and $\phi_N^A = 0.30$. Open circles represent results for suspensions of only magnetizable spheres with area fractions $\phi_M^A = 0.45$ ($\phi_N^A = 0$). At low strain amplitudes, the storage modulus for mixtures is larger than that for suspensions in which $\phi_N^A = 0$. The larger plateau modulus for mixtures indicates that nonmagnetizable spheres participate in stress transfer via repulsive forces.

Figure 4.1 illustrates that both types of suspensions transition from linear to nonlinear viscoelastic behavior at similar strain amplitudes. To further examine the transition to nonlinear deformation, we calculated the ratio of the magnitude of the third harmonic, $|\tilde{\tau}_3|$, to first harmonic, $|\tilde{\tau}_1|$. Figure 4.2 shows $|\tilde{\tau}_3| \,/\, |\tilde{\tau}_1|$ as a function of γ_0 for monolayers with and without nonmagnetizable spheres. Open squares represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0.30$. Open circles represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0$. The dependence of $|\tilde{\tau}_3^*| \,/\, |\tilde{\tau}_1^*|$ on strain amplitude is nearly identical for the two systems, with the ratio increasing with increasing strain

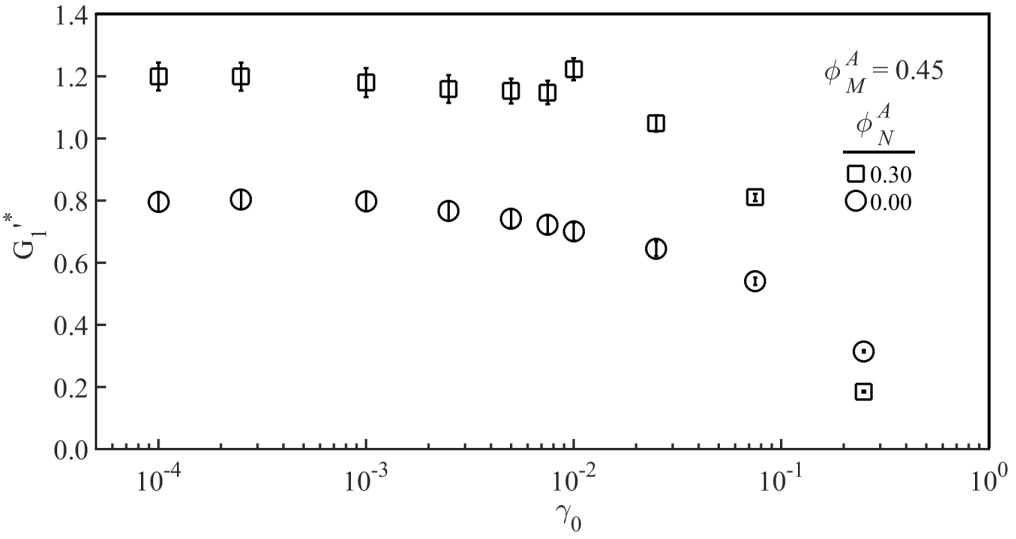

Figure 4.1: Storage modulus as a function of strain amplitude for monolayer suspensions. Open squares represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0.30$. Open circles represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0$.

amplitude. Thus the onset of nonlinear deformation is unaffected by the presence of nonmagnetizable spheres.

The storage modulus and onset of nonlinear deformation were also determined for three-dimensional suspensions. Figure 4.3 presents G_1' as a function of γ_0 for three-dimensional suspensions with different compositions. Open squares represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0.15$. Open circles represent suspensions of volume fraction $\phi_M = 0.30$ and $\phi_N = 0$. Just as for the monolayers in Fig.4.1, the plateau modulus of the mixture is larger than the plateau modulus for the suspension containing only magnetizable spheres.

The ratio $|\tilde{\tau}_3^*| \,/\, |\tilde{\tau}_1^*|$ is plotted as a function of γ_0 for three-dimensional suspensions in Fig. 4.4. Open squares represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0.15$. Open circles represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0$. Just as observed for monolayers, the dependence of $|\tilde{\tau}_3^*| \,/\, |\tilde{\tau}_1^*|$ on strain amplitude is nearly identical for

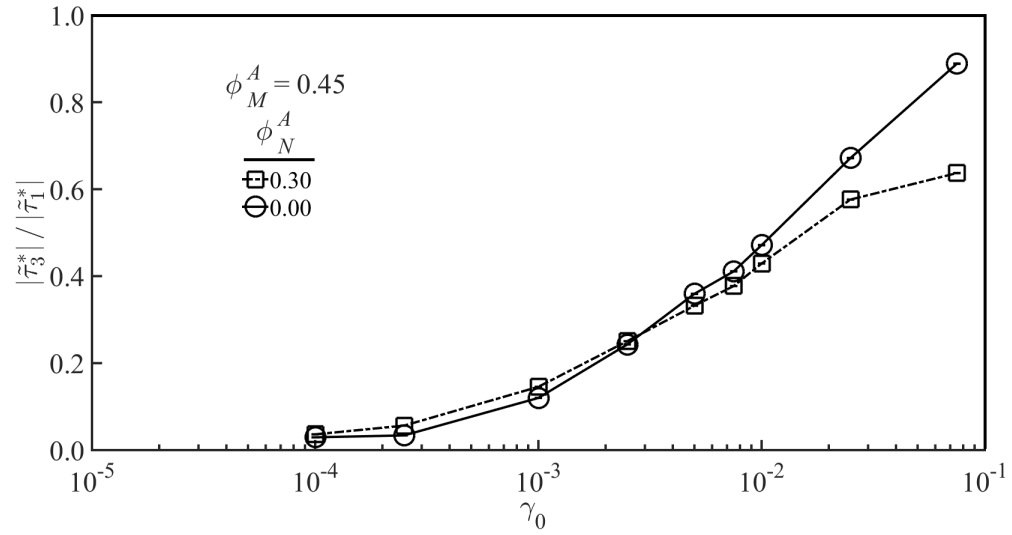

Figure 4.2: $|\tilde{\tau}_3| / |\tilde{\tau}_1|$ as a function of γ_0 for monolayer suspensions. Open squares represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0.30$. Open circles represent suspensions with $\phi_M^A = 0.45$ and $\phi_N^A = 0$.

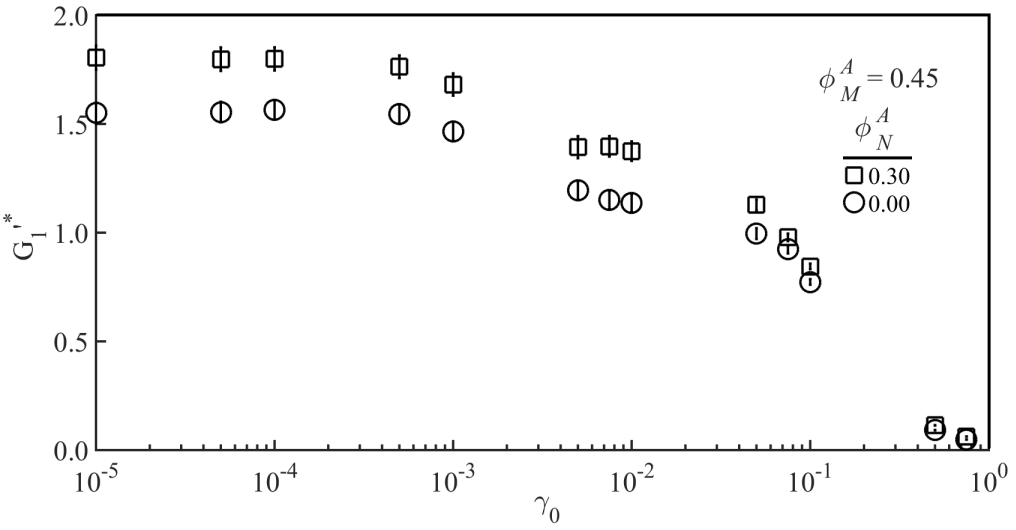

Figure 4.3: Storage modulus as a function of strain amplitude for three-dimensional suspensions. Open squares represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0.15$. Open circles represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0$.

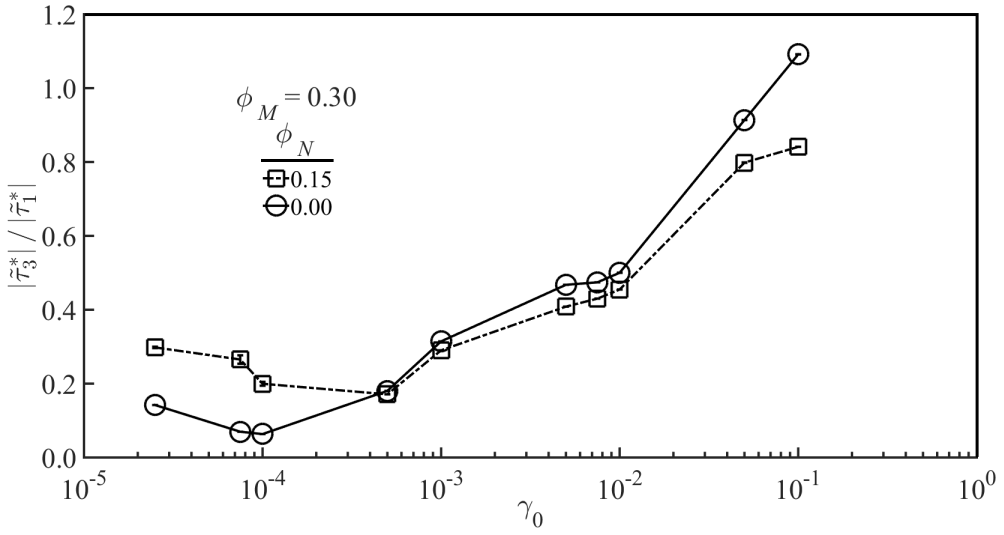

Figure 4.4: $|\tilde{\tau}_3| / |\tilde{\tau}_1|$ as a function of γ_0 for three-dimensional suspensions. Open squares represent suspensions with $\phi_M^A = 0.30$ and $\phi_N^A = 0.15$. Open circles represent suspensions with $\phi_M = 0.30$ and $\phi_N = 0$.

the two systems, with the ratio increasing with increasing strain amplitude. Thus the onset of nonlinear deformation in three-dimensional suspensions is unaffected by the presence of nonmagnetizable spheres.

Parthasarathy and D.J. Klingenberg (1995a,b) showed that the onset of nonlinear deformation in electrorheological fluids at low frequencies (such as that employed in the present study) is caused by the slight rearrangement of unstable structures. In shear flow (continuous or oscillatory), structures are sheared into unstable configurations, which then rearrange, producing nonlinear stress-strain behavior. Investigation of model structures illustrated that the critical strain marking the transition to nonlinear behavior can vary significantly from one structure to another.

The model employed by Parthasarathy and D.J. Klingenberg (1995a,b) is the electrostatic analog of the magnetostatic model employed here, and thus their results apply. Their observations, along with the similarity of two data sets in Figs. 4.2 and

4.4, suggest that the presence of nonmagnetizable spheres does not alter the stability of the structures. As a result, the data presented in Figs. 4.1 and 4.3 suggest that the nonmagnetizable spheres act to increase the suspension stress by directly increasing the suspension stiffness, as opposed to increasing the stability of the structures (i.e., as opposed to allowing the structures to be sheared to a larger strain before rearranging).

To understand how the nonmagnetizable spheres directly increase the suspension stiffness and shear stress, consider the snapshots of simulated monolayers in Figs. 4.5–4.7. The magnetizable spheres are represented by the green circles, and the nonmagnetizable spheres are represented by the red circles. Also shown in these figures are lines connecting spheres (within the cut-off radius) that represent the sign and magnitude of the net pair interaction force acting along the line-of-centers, $\boldsymbol{F}_{ij}^{*,\mathrm{net}} \cdot \boldsymbol{r}_{ij}^{*}$, where $\boldsymbol{F}_{ij}^{*,\mathrm{net}} = \boldsymbol{F}_{ij}^{*,\mathrm{mag.}} + \boldsymbol{F}_{ij}^{*,\mathrm{rep.}}$. If $\boldsymbol{F}_{ij}^{*,\mathrm{net}} \cdot \boldsymbol{r}_{ij}^{*} > 0$, the net force is attractive and the line connecting the spheres is yellow; if $\boldsymbol{F}_{ij}^{*,\mathrm{net}} \cdot \boldsymbol{r}_{ij}^{*} < 0$, the net force is repulsive and the line connecting the spheres is black. The line thickness represents the magnitude of the interaction force, with thicker lines representing larger magnitude forces. Because the repulsive force magnitudes can be much larger than the attractive force magnitudes, the dimensionless line thickness is prescribed by the monotonic, but nonlinear function $(1/2)\tanh(|\boldsymbol{F}_{ij}^{*,\mathrm{net}}|/4)$.

Snapshots of sheared monolayers with ϕ_M^A fixed at 0.45 and various values of ϕ_N^A ($0 \leq \phi_N^A \leq 0.30$) are presented in Fig.4.5. For $\phi_N^A = 0$ (Fig.4.5(a)), the magnetizable spheres form column-like structures, as expected. Single-sphere-width clusters tend to be strained and tilted in the flow direction, with shear forces transmitted between the shearing surfaces via attractive magnetostatic forces; this behavior is illustrated in Fig. 4.5(a) by the yellow lines connecting the spheres within the strained, single-

sphere width clusters. Also apparent in Fig.4.5(a) are black lines that connect some spheres within the larger clusters, which indicates that repulsive forces also play a role in stress transfer.

As the concentration of nonmagnetizable spheres is increased (Figs. 4.5(b)–(f)), the number of black lines—and thus the prominence of repulsive forces—increases. Of most importance are the repulsive forces that act along the compression axis of the shear flow, which contribute to the stress resisting the deformation. The gray circles in Figs. 4.5(b)–(f) illustrate clusters of spheres in which the repulsive forces act along the compression axis throughout the cluster. Some of these "repulsive-force clusters" percolate (extend from one shearing surface to the other), particularly at larger values of ϕ_N^A.

The shear-induced formation of repulsive-force clusters is similar to the force chains in jammed, hard-sphere systems that form along the compression axis of the shear flow [Farr et al. (1997); Cates et al. (1998)]. In the present case, in most if not all of the repulsive-force clusters, the force chains include both magnetizable and nonmagnetizable spheres. In addition, the magnetizable spheres that participate in the black repulsive-force chains often simultaneously participate in the yellow attractive-force chains. This likely explains the observation (experimental and simulation) that the nonmagnetizable spheres enhance the field-induced stress, even though these spheres are not magnetizable—the nonmagnetizable spheres act within a field-induced structure of magnetizable spheres, and thus both structures disappear when the field is removed.

Repulsive-force clusters can also appear in sheared monolayers of only magnetizable spheres at sufficiently large concentration, as illustrated in Fig. 4.6. At low

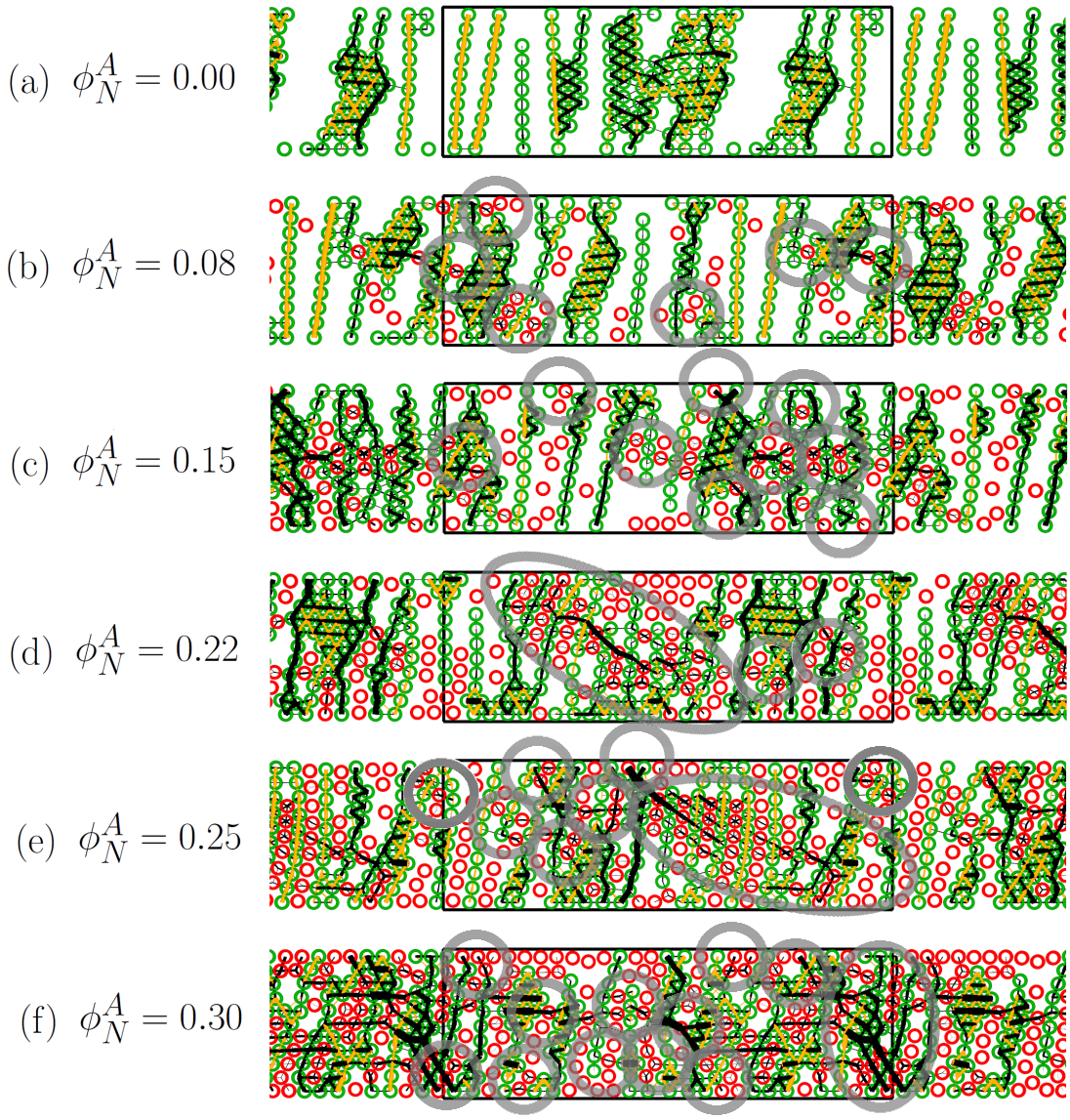

Figure 4.5: Snapshots of sheared monolayer suspensions with $\phi_M^A = 0.45$ and various values of ϕ_N^A (a) $\phi_N^A = 0.00$; (b) $\phi_N^A = 0.08$; (c) $\phi_N^A = 0.15$; (d) $\phi_N^A = 0.22$; (e) $\phi_N^A = 0.25$; (f) $\phi_N^A = 0.30$.

concentrations (Figs. 4.6(a) and (b)), percolating single-sphere width chains transmit stress via attractive forces. As the concentration is increased, repulsive forces become more prominent. At the highest concentrations (Figs. 4.6(e) and (f)), the repulsive forces can act within anisometric clusters oriented along the compression axis of the shear flow.

Snapshots of a monolayer mixture of magnetizable and nonmagnetizable spheres ($\phi_M^A = 0.45$, $\phi_N^A = 0.15$) at different strains are presented in Fig. 4.7 along with a plot of the shear stress as a function of shear strain. The shear stresses that correspond to the snapshots are represented by labeled points along the curve; the chosen snapshots correspond to local shear stress maxima. For all snapshots shown, repulsive-force clusters are apparent (illustrated with gray circles), and each contains at least one percolating cluster roughly oriented along the compression axis. Again, these clusters contain both magnetizable and nonmagnetizable spheres.

To quantify the contribution of the nonmagnetizable spheres to the shear stress, we employ partial stresses [Ahn and D.J. Klingenberg (1994)]. Equation 4.8 for the dimensionless shear stress can be separated into two summations, one over each type of sphere,

$$\tau_{xz}^* = -\frac{1}{V^*}\sum_{i=1}^{N_M} z_i^* F_{x,i}^* - \frac{1}{V^*}\sum_{i=1}^{N_N} z_i^* F_{x,i}^* \tag{4.11}$$

$$= \tau_{xz}^{*,M} + \tau_{xz}^{*,N} \tag{4.12}$$

where N_M and N_N are the number of magnetizable and nonmagnetizable spheres, respectively. The first summation above is the partial stress of the magnetizable spheres, and the second summation is the partial stress of the nonmagnetizable spheres. Note

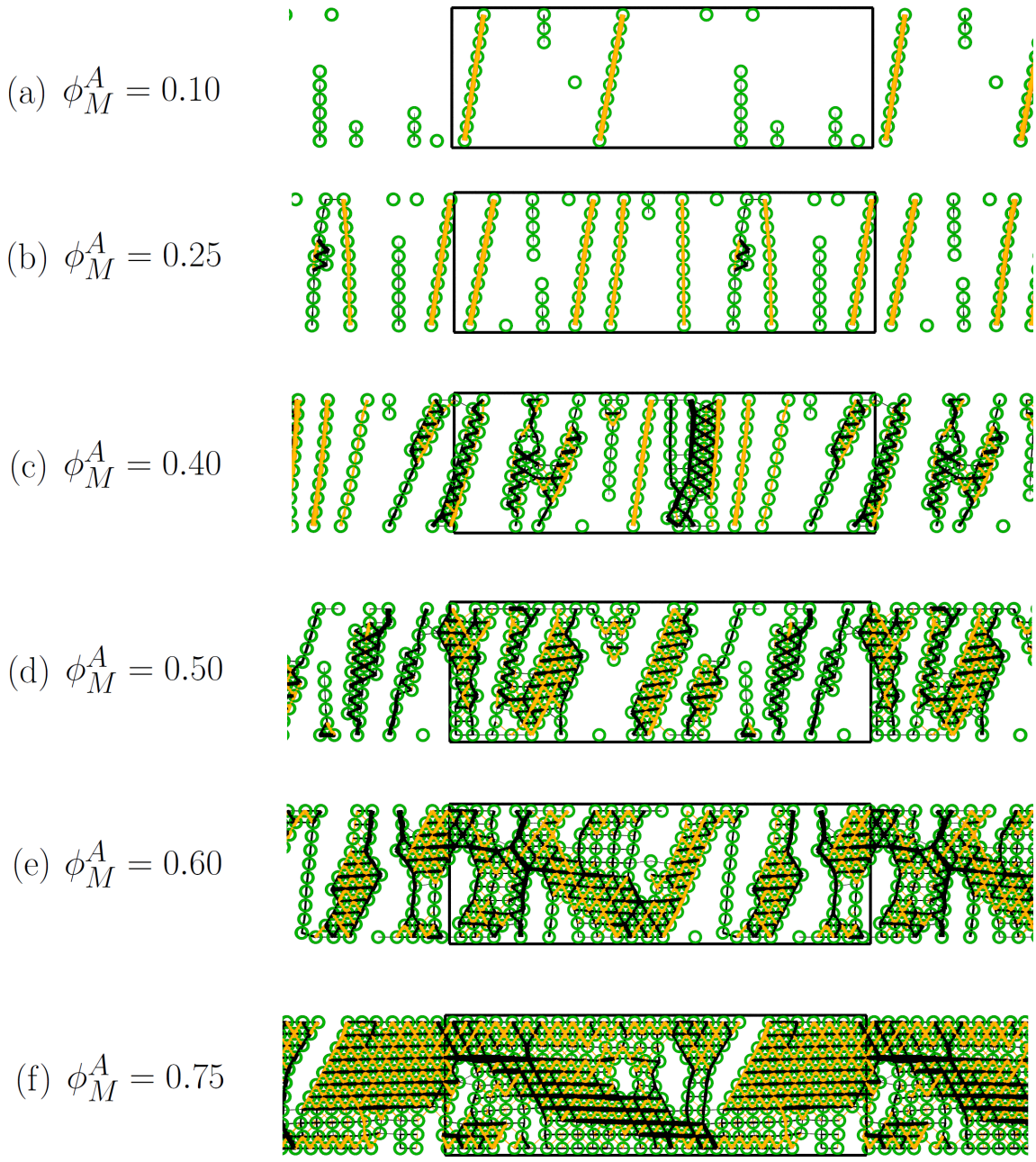

Figure 4.6: Shapshots of sheared monolayer suspensions with $\phi_N^A = 0.00$ and various values of ϕ_M^A (a) $\phi_M^A = 0.10$; (b) $\phi_M^A = 0.25$; (c) $\phi_M^A = 0.40$; (d) $\phi_M^A = 0.50$; (e) $\phi_M^A = 0.60$; (f) $\phi_M^A = 0.75$.

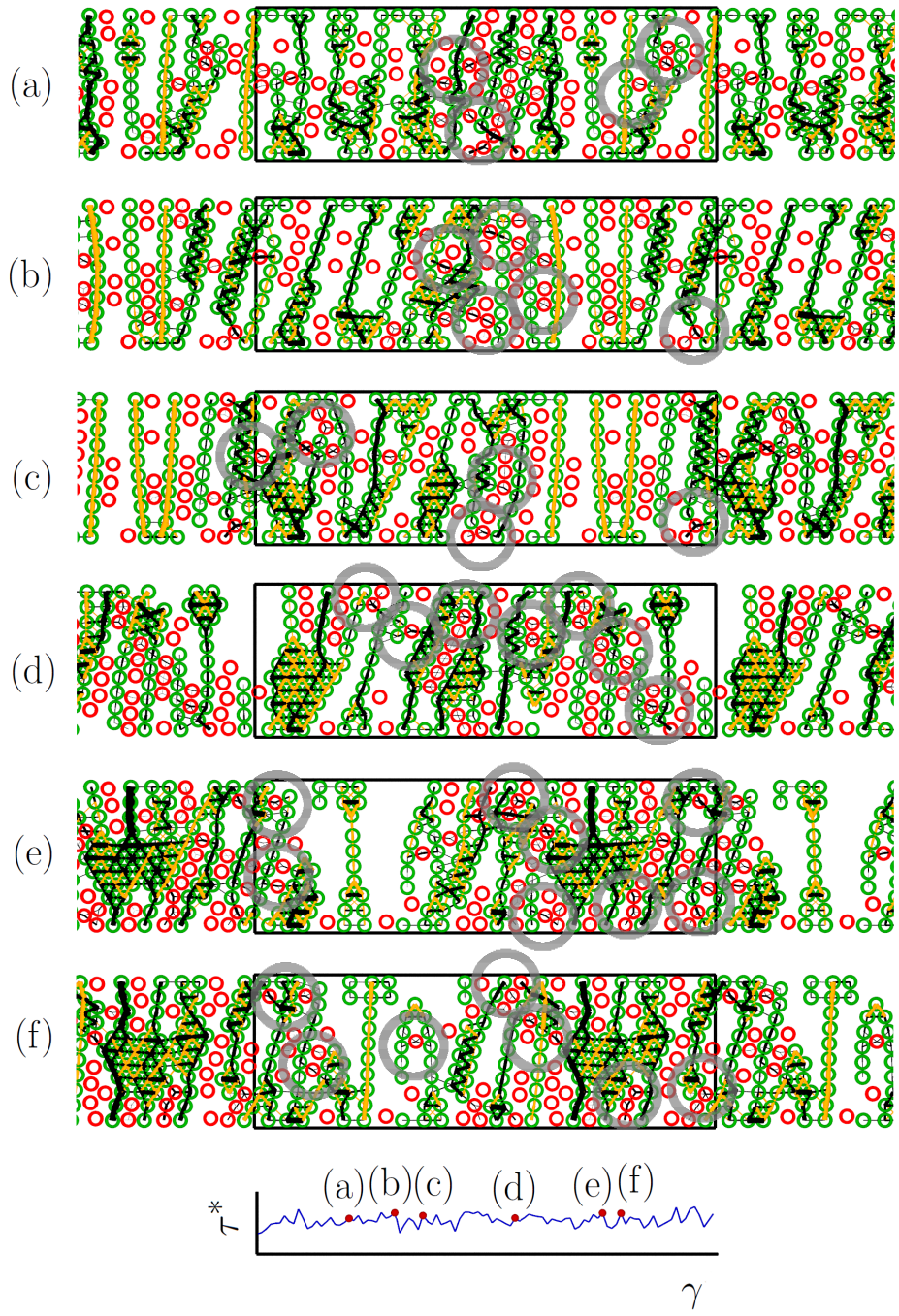

Figure 4.7: Sequence of snapshots for a monolayer suspension at various shear strains ($\phi_M^A = 0.45$ and $\phi_N^A = 0.15$).

that $\tau_{xz}^{*,N}$ can only consist of contributions from nonmagnetizable spheres interacting via short-range repulsive forces with either magnetizable or nonmagnetizable spheres, whereas $\tau_{xz}^{*,M}$ consists of contributions from magnetizable spheres interacting through short-range repulsive forces with either type of sphere, as well as via magnetostatic interactions with other magnetizable spheres. For all results presented here, the stresses from relaxed configurations will be presented (i.e., the partial stresses are the respective contributions to the yield stress), averaged over initial configurations and the strain interval $1 \leq \gamma \leq 5$.

The partial stresses are plotted along with the total stresses in Figs. 4.8-4.10 for monolayer and three-dimensional simulations. In Fig. 4.8, the partial and total yield stresses are plotted as a function of ϕ_N^A for $\phi_M^A = 0.45$; these are the same conditions employed in Fig. 4.5. Open squares represent the total shear stress, open circles represent $\tau_{xz}^{*,M}$, and open triangles represent $\tau_{xz}^{*,N}$. For all $\phi_N^A > 0$, the partial stress $\tau_{xz}^{*,N}$ is positive, indicating that the nonmagnetizable spheres directly contribute to the stress. The partial stress $\tau_{xz}^{*,M}$ also increases as ϕ_N^A is increased, indicating that the nonmagnetizable spheres also indirectly contribute to the stress in monolayer systems. The fact that both $\tau_{xz}^{*,N}$ and $\tau_{xz}^{*,M}$ increase as ϕ_N^A is increased is not surprising because, as illustrated in Fig. 4.5, both types of spheres participate in the repulsive-force clusters.

In Fig. 4.9, the partial and total stresses for three-dimensional systems are plotted as a function of ϕ_N for $\phi_M = 0.30$. In this case, $\tau_{xz}^{*,M}$ is not as strongly affected by the addition of nonmagnetizable spheres as in the case of monolayer systems, but $\tau_{xz}^{*,N}$ is still greater than 0, and increases monotonically with ϕ_N.

In Fig. 4.10, the partial and total stresses for three-dimensional systems are

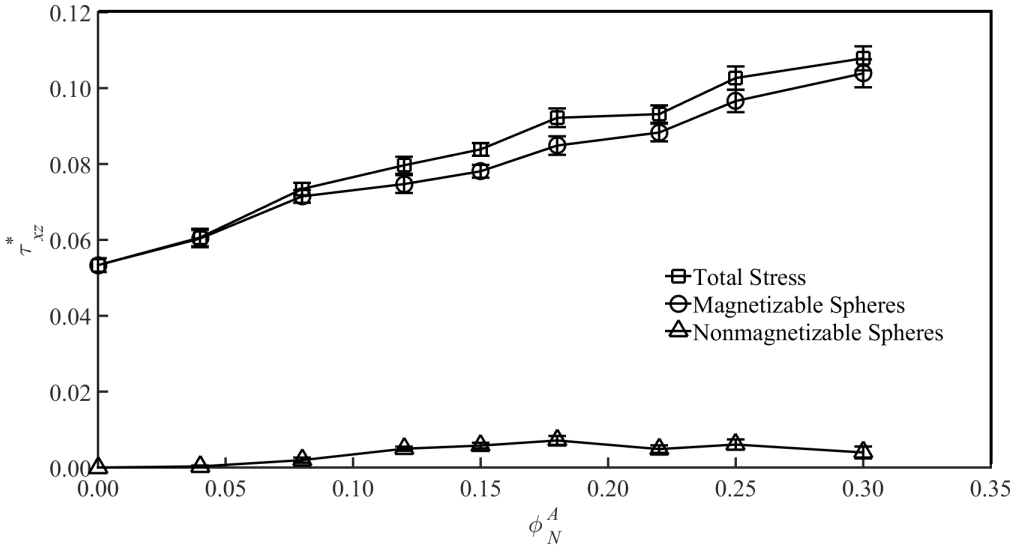

Figure 4.8: Total yield and partial stresses as a function of ϕ_N^A for $\phi_M^A = 0.45$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetizable spheres. Open triangles represent the partial stress associated with nonmagnetizable spheres.

plotted as a function of ϕ_M for ϕ_T fixed at 0.45. As before, $\tau_{xz}^{*,N} > 0$. The total stress and $\tau_{xz}^{*,M}$ pass through a maximum at $\phi_M \approx 0.40$, whereas $\tau_{xz}^{*,N}$ passes through a maximum at $\phi_M \approx 0.30$. These results also suggest that nonmagnetizable spheres directly increase the stress (via $\tau_{xz}^{*,N}$), as well as indirectly affect the stress (by altering $\tau_{xz}^{*,M}$).

Partial stresses may also be defined in terms of the types of forces as opposed to the types of spheres. The nonhydrodynamic force on each sphere consists of magnetostatic (Eq. 4.2) and short-range repulsive (Eq. 4.4) forces. The summation for the stress (Eq. 4.8) can thus be separated into summations over each type of force,

$$\tau_{xz}^* = -\frac{1}{V^*}\sum_{i=1}^{N} z_i^* F_{x,i}^{*,\text{mag.}} - \frac{1}{V^*}\sum_{i=1}^{N} z_i^* F_{x,i}^{*,\text{rep.}} \tag{4.13}$$

$$= \tau_{xz}^{*,\text{mag.}} + \tau_{xz}^{*,\text{rep.}} \tag{4.14}$$

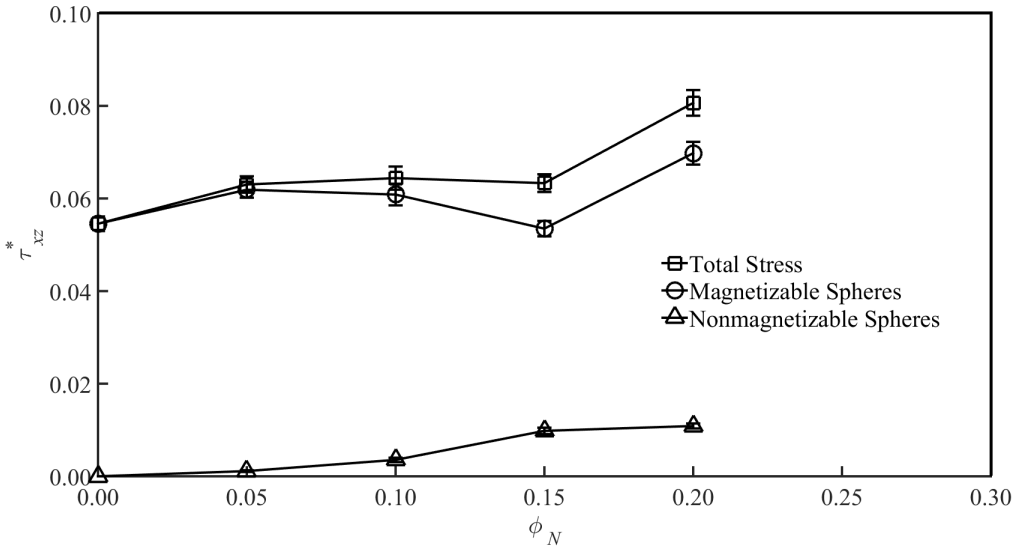

Figure 4.9: Total yield and partial stresses as a function of ϕ_N for $\phi_M = 0.30$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetizable spheres. Open triangles represent the partial stress associated with nonmagnetizable spheres.

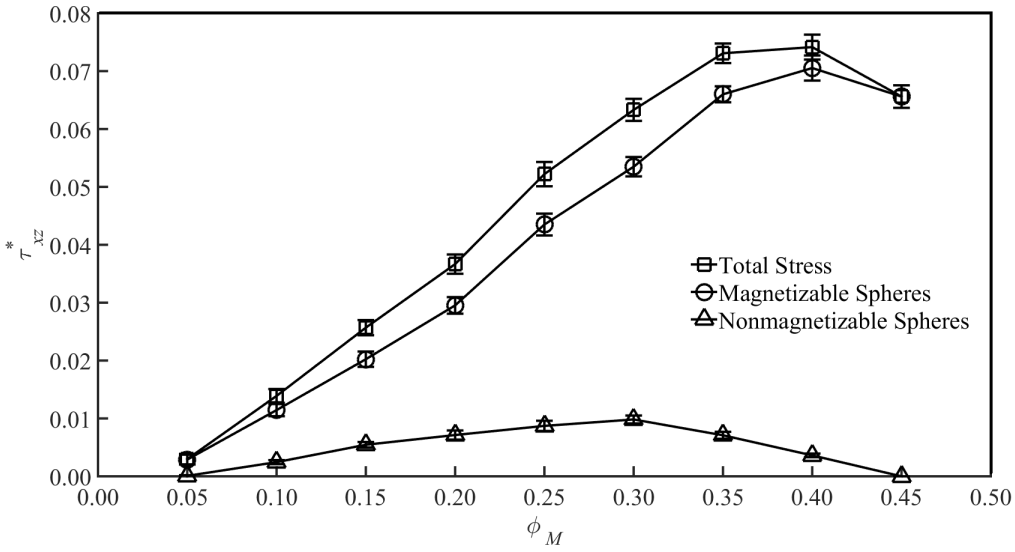

Figure 4.10: Total yield and partial stresses as a function of ϕ_M for a fixed $\phi_T = 0.45$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetizable spheres. Open triangles represent the partial stress associated with nonmagnetizable spheres.

where $F_{x,i}^{*,\mathrm{mag.}}$ is the sum of all the pair magnetostatic forces acting on sphere i, $F_{x,i}^{*,\mathrm{rep.}}$ is the sum of all the pair short-range repulsive forces acting on sphere i, $\tau_{xz}^{*,\mathrm{mag.}}$ is the partial stress caused by magnetostatic forces, and $\tau_{xz}^{*,\mathrm{rep.}}$ is the partial stress caused by short-range repulsive forces. Note that only magnetizable spheres can contribute directly to $\tau_{xz}^{*,\mathrm{mag.}}$, whereas both magnetizable and nonmagnetizable spheres can directly contribute to $\tau_{xz}^{*,\mathrm{rep.}}$.

The partial stresses $\tau_{xz}^{*,\mathrm{mag.}}$ and $\tau_{xz}^{*,\mathrm{rep.}}$ are plotted along with the total stress as a function of ϕ_N^A in Fig. 4.11 for monolayer simulations with $\phi_M^A = 0.40$. The magnetostatic force contribution $\tau_{xz}^{*,\mathrm{mag.}}$ increases monotonically with ϕ_N^A, but $\tau_{xz}^{*,\mathrm{rep.}} < 0$, and decreases monotonically with ϕ_N^A. Because $\tau_{xz}^{*,\mathrm{rep.}}$ is equal to $\tau_{xz}^{*,N}$ (> 0) plus the repulsive force contribution from the magnetizable spheres, this implies that the repulsive force contribution from the magnetizable spheres is negative for monolayers for these compositions. Thus, although the magnetizable spheres participate in the repulsive force clusters, it is their magnetostatic contribution to the stress that is enhanced, while their repulsive force contribution detracts from the total stress.

The partial stresses $\tau_{xz}^{*,\mathrm{mag.}}$ and $\tau_{xz}^{*,\mathrm{rep.}}$ are plotted along with the total stress as a function of ϕ_M^A in Fig. 4.12 for monolayer simulations with $\phi_N^A = 0.00$. For $\phi_M^A \geq 0.50$, the total stress is nearly constant, while $\tau_{xz}^{*,\mathrm{mag.}}$ decreases and $\tau_{xz}^{*,\mathrm{rep.}}$ increases with increasing ϕ_M^A (decreasing ϕ_N^A). Comparison of these results with those in Fig. 4.11 reveals that nonmagnetizable spheres cause $\tau_{xz}^{*,\mathrm{rep.}}$ to decrease, which in turn increases $\tau_{xz}^{*,\mathrm{mag.}}$, further supporting the notion that nonmagnetizable spheres enhance the magnetostatic contribution.

The partial stresses $\tau_{xz}^{*,\mathrm{mag.}}$ and $\tau_{xz}^{*,\mathrm{rep.}}$ are plotted along with the total stress as a function of ϕ_N for three-dimensional simulations with $\phi_M = 0.30$ in Fig. 4.13. In

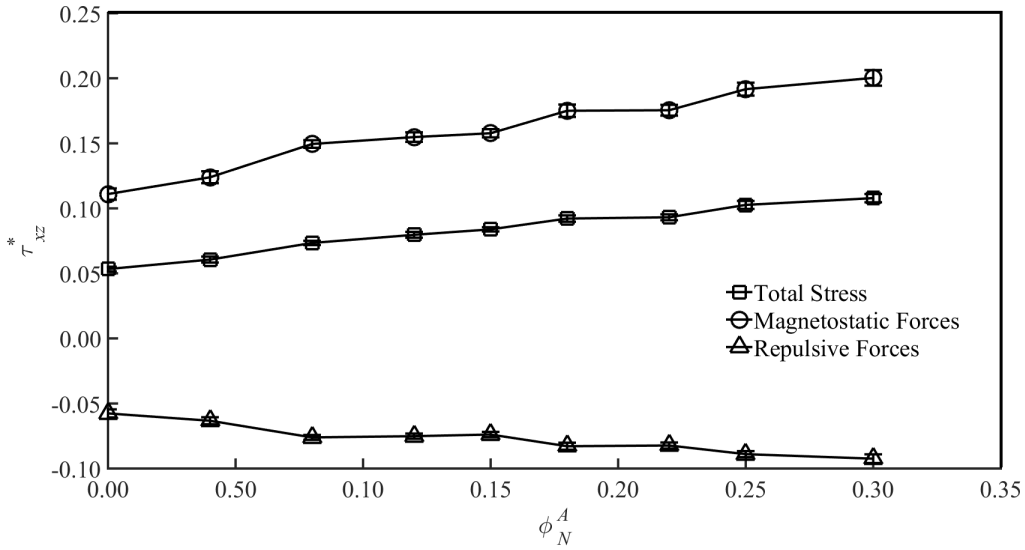

Figure 4.11: Total yield and partial stresses as a function of ϕ_N^A for $\phi_M^A = 0.45$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetostatic forces. Open triangles represent the partial stress associated with repulsive forces.

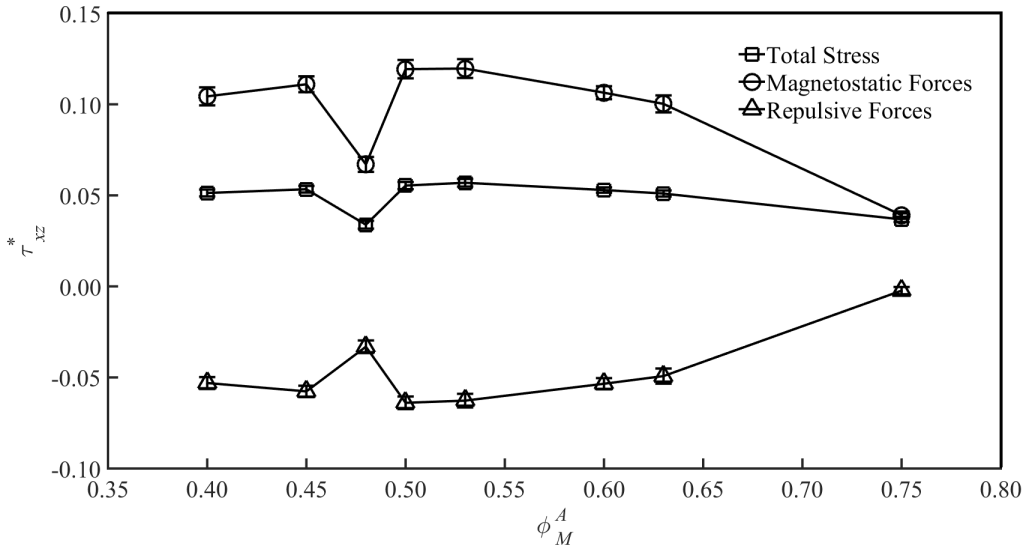

Figure 4.12: Total yield and partial stresses as a function of ϕ_M^A for $\phi_N^A = 0$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetostatic forces. Open triangles represent the partial stress associated with repulsive forces.

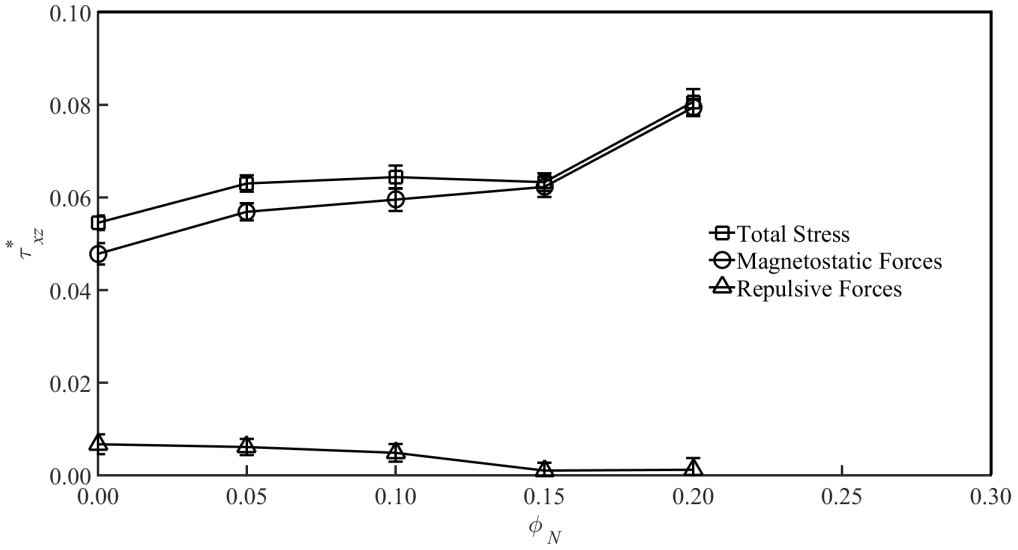

Figure 4.13: Total yield and partial stresses as a function of ϕ_N for $\phi_M = 0.30$. Open squares represent the total yield stress. Open circles represent the partial stress associated with magnetostatic forces. Open triangles represent the partial stress associated with repulsive forces.

contrast to the monolayer systems, $\tau^{*,\text{rep.}}_{xz} > 0$, but still decreases as ϕ_N is increased. However, since $\tau^{*,N}_{xz} > 0$ and increases with ϕ_N (Fig. 4.9), the contribution to the total stress from the repulsive forces on magnetizable spheres is still negative and decreases with increasing ϕ_N. So, as with the monolayer systems, addition of nonmagnetizable spheres enhances the magnetostatic force contribution to $\tau^{*,M}_{xz}$.

The repulsive-force clusters can be identified by modifying the cluster detection algorithm devised by Sevick et al. (1988). Specifically, we define two spheres to be directly connected within a repulsive-force cluster if $\boldsymbol{F}^{*,\text{net}}_{ij} \cdot \boldsymbol{r}^*_{ij} < 0$ and $\left|\boldsymbol{F}^{*,\text{net}}_{ij}\right| > 1.5$, which corresponds to sphere pairs that overlap to a center-to-center separation of $\boldsymbol{r}^*_{ij} < 0.99$; all the spheres within the same cluster can then be determined as described by Sevick et al. (1988). The number of repulsive-force clusters for a given composition were calculated for each relaxed configuration, and then averaged over the strain interval $1 \leq \gamma \leq 5$, and over the 10 different initial configurations.

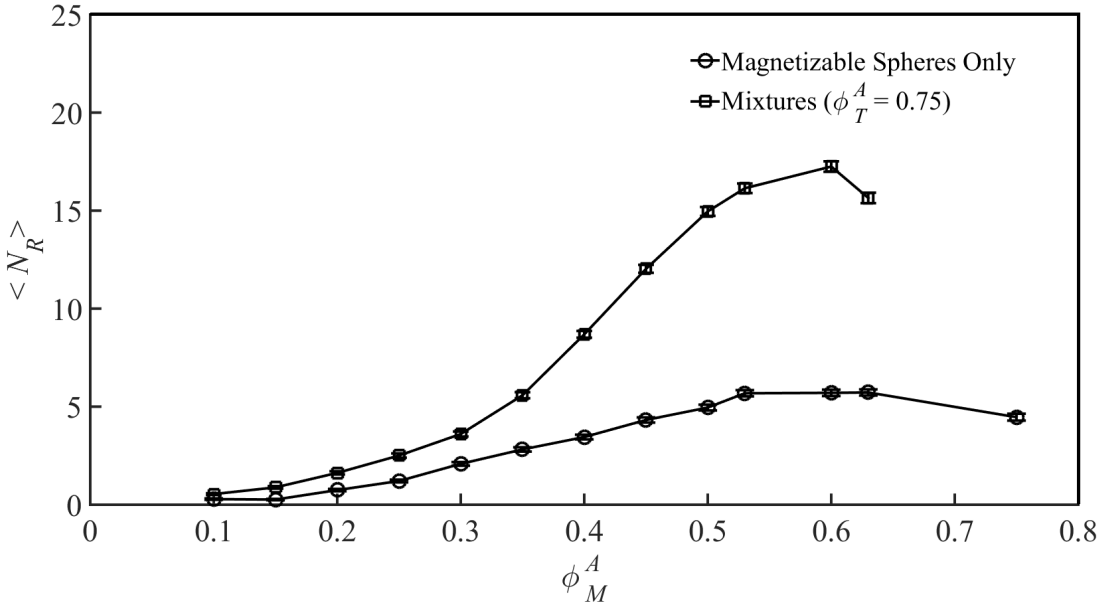

Figure 4.14: Average number of repulsive-force clusters as a function of magnetizable sphere area fraction. Open circles represent suspensions containing only magnetizable spheres. Open squares represent suspensions containing a mixture of spheres with the total area fraction fixed at $\phi_T^A = 0.75$.

The average number of repulsive-force clusters, $\langle N_R \rangle$, is plotted as a function of ϕ_M^A in Fig. 4.14 for monolayer simulations of only magnetizable spheres, and for monolayer simulations of mixtures of magnetizable and nonmagnetizable spheres with $\phi_T^A = 0.75$. For a given value of ϕ_M^A, the system with nonmagnetizable spheres contains more repulsive-force clusters than the system containing only magnetizable spheres. The number of clusters passes through a maximum at large ϕ_M^A, presumably because the magnetostatic forces cause the magnetizable spheres to form larger, less fibrous clusters at large ϕ_M^A [Klingenberg et al. (1991b)].

The average number of repulsive-force clusters is plotted as a function of ϕ_N^A for various values of ϕ_M^A in Fig. 4.15 for monolayer simulations. For fixed ϕ_M^A, the number of clusters increases monotonically with ϕ_N^A. This again is consistent with the snapshots in Fig. 4.5, where the number of repulsive-force clusters appears to increase

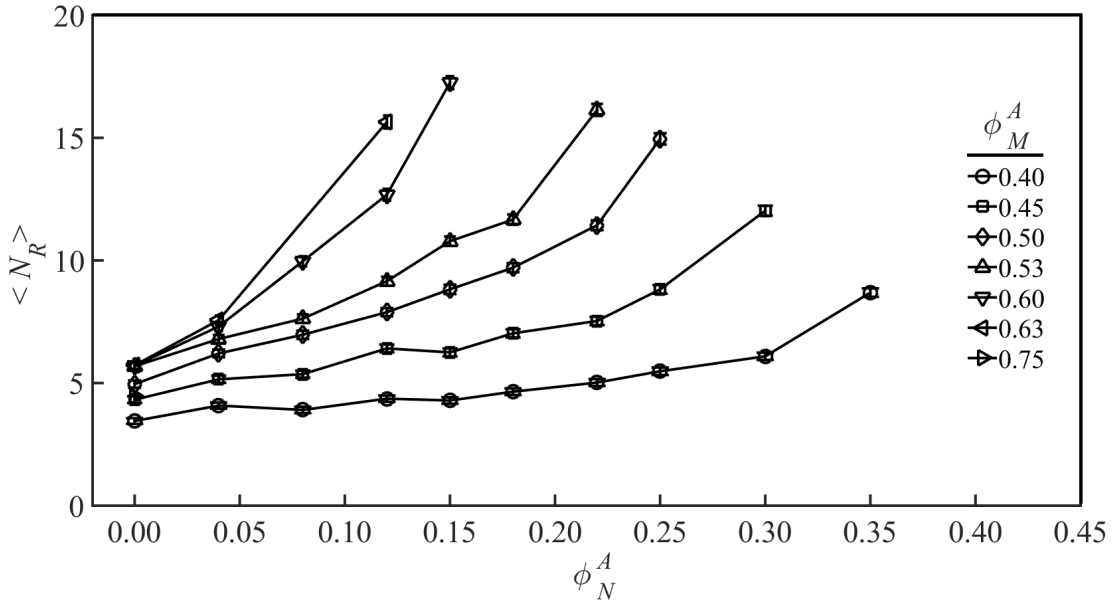

Figure 4.15: Average number of repulsive-force clusters as a function of nonmagnetizable sphere area fraction for various magnetizable sphere area fractions.

with ϕ_N^A. In contrast with the results in Fig. 4.14, the number of clusters does not pass through a maximum when plotted as a function of ϕ_N^A (for the range of ϕ_N^A considered). The ϕ_N^A-dependence of the yield stress follows the same monotonic behavior, consistent with the notion of the shear stress being increased through repulsive-force clusters similar to that in jammed, hard-sphere suspensions as shown in Fig. 3.1 of Chapter 3.

Results for three-dimensional simulations are qualitatively similar to those for the monolayer systems. The average number of repulsive-force clusters is plotted as a function of ϕ_M in Fig. 4.16 for three-dimensional simulations suspensions of only magnetizable spheres, and for simulations of mixtures of magnetizable and nonmagnetizable spheres with $\phi_T = 0.45$. The mixtures contain more repulsive-force clusters than suspensions of only magnetizable spheres, which is consistent with the

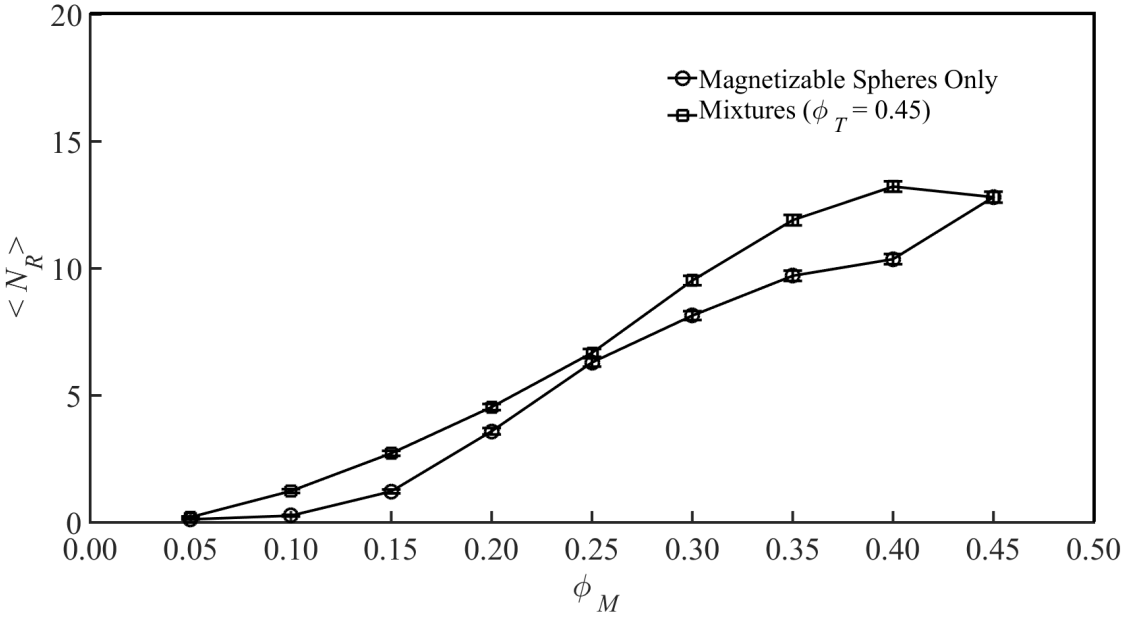

Figure 4.16: Average number of repulsive-force clusters as a function of magnetizable sphere area fraction. Open circles represent suspensions containing only magnetizable spheres. Open squares represent suspensions containing a mixture of spheres with the total volume fraction fixed at $\phi_T^A = 0.45$.

notion that nonmagnetizable enhance the stress through repulsive-force clusters in three-dimensional systems as well.

In Fig. 4.17, the average number of repulsive-force clusters is plotted as a function of ϕ_N for various values for ϕ_M for three-dimensional systems. For small ϕ_M, the number of repulsive-force clusters is insensitive to ϕ_N. The number increases with ϕ_N for large ϕ_M ($\gtrsim 0.20$), for sufficiently large ϕ_N. The shapes of the curves in Fig. 4.17 are strikingly similar to those for the stress as a function of ϕ_N [Fig. 3.1 in Chapter 3] providing further evidence of the role of repulsive-force clusters in the stress enhancement.

Cluster size distributions are illustrated in Figs. 4.18 and 4.19. Figure 4.18 is a plot of number of clusters that contain N spheres (magnetizable plus nonmagnetizable) as a function of N for monolayers with fixed magnetizable sphere area fraction $\phi_M^A = 0.45$

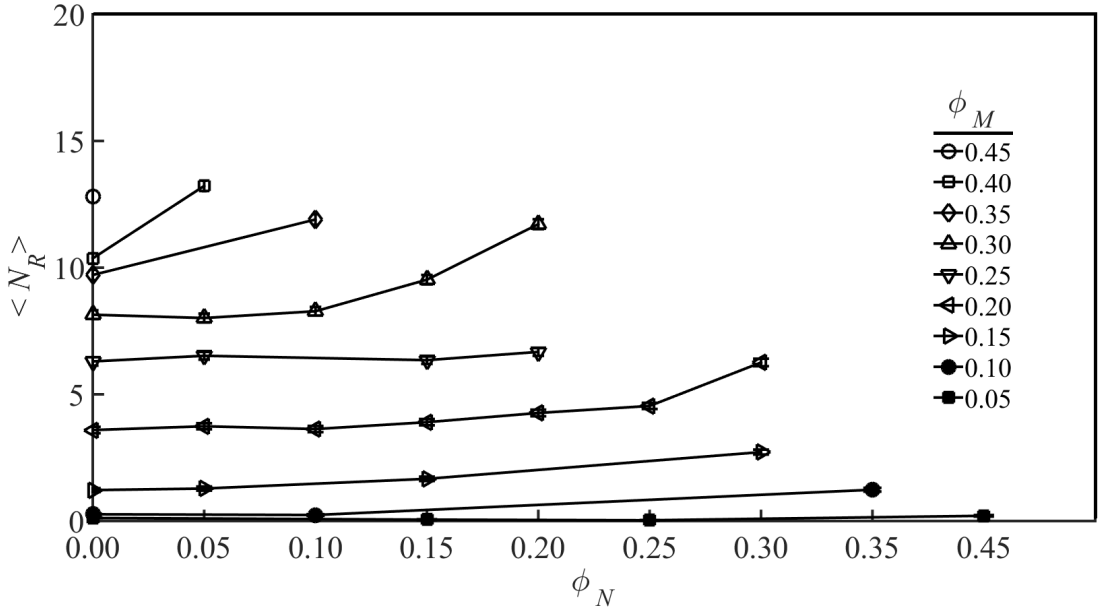

Figure 4.17: Average number of repulsive-force clusters as a function of nonmagnetizable sphere volume fraction for various magnetizable sphere volume fractions.

and various values of ϕ_N^A. For most suspensions, larger ϕ_N^A translates to a larger number of clusters for all cluster sizes, with greater changes for larger values of N.

Figure 4.19 is a plot of number of clusters that contain N spheres (magnetizable plus nonmagnetizable) as a function of N for three-dimensional suspensions with volume fraction $\phi_M = 0.30$ and various values of ϕ_N. Similar to the results presented in Fig. 4.18, larger ϕ_N leads to more repulsive clusters regardless of the cluster size. Both Fig. 4.18 and 4.19 further indicate that adding nonmagnetizable spheres increases the number of repulsive clusters, with greater changes for larger values of N.

The transient rheological behavior of MR suspensions composed of mixtures of magnetizable and nonmagnetizable spheres also shows behavior consistent with jammed systems. Hard-sphere dispersions that jam exhibit a strain-dependent shear stress

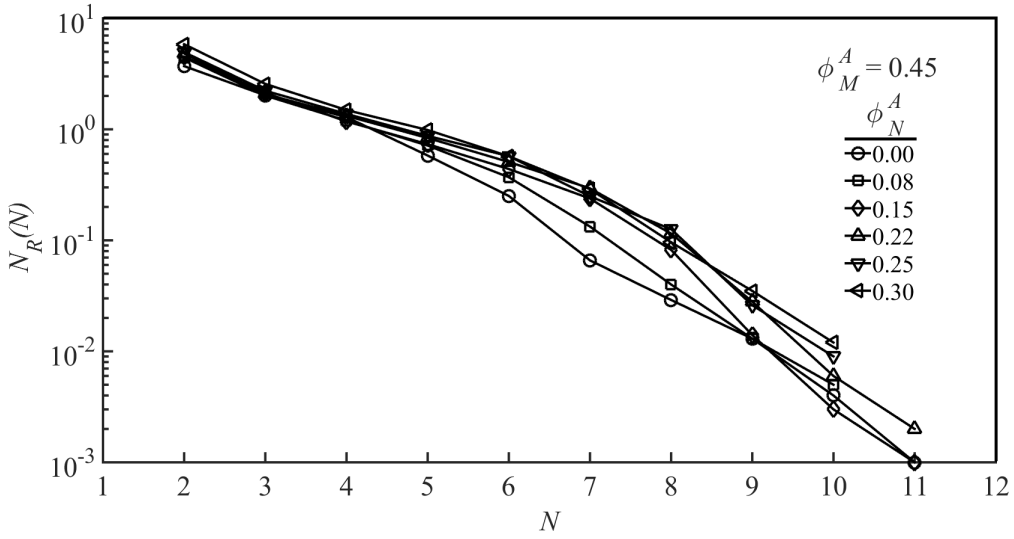

Figure 4.18: Number of repulsive-force clusters that contain N spheres as a function of N for $\phi_M^A = 0.45$ and various values of ϕ_N^A.

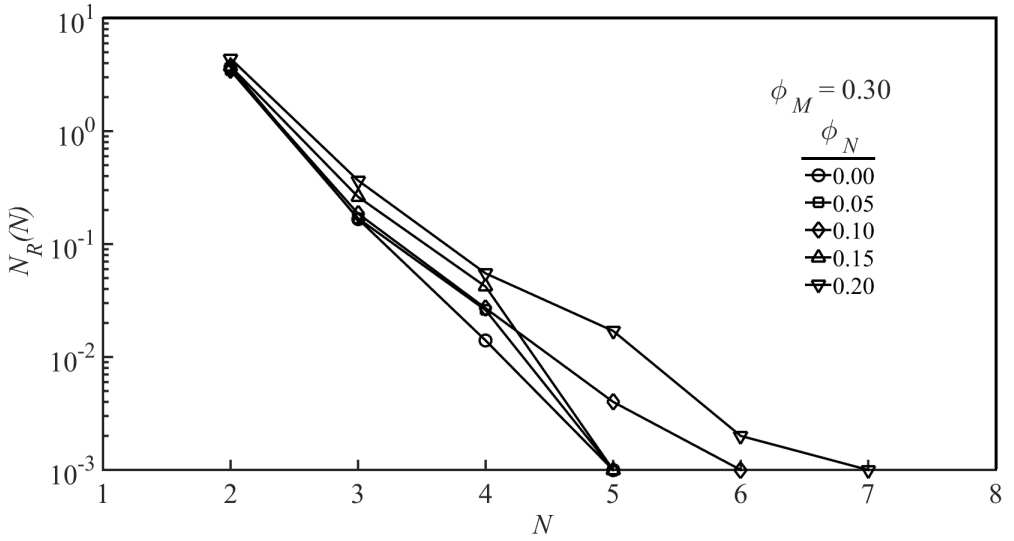

Figure 4.19: Number of repulsive-force clusters that contain N spheres as a function of N for $\phi_M = 0.3$ and various values of ϕ_N.

that increases as the system approaches the jammed state, which occurs at a nonzero, finite shear strain less than 1 [Farr et al. (1997)]. The jammed state occurs because of the shear-induced formation of large particle clusters.

Figure 4.20 contains plots of the shear stress as a function of shear strain for monolayer systems with various values of ϕ_M^A and ϕ_N^A. These stresses were determined from the relaxed configurations, and thus Fig. 4.20 represents quasistatic results at effectively zero shear rate. Each data point represents the stress at a specific strain, averaged over 10 different initial conditions. For $\phi_N^A = 0$, the stress increases gradually to a steady-state value by a shear strain of roughly 1 (some systems appear to first exhibit a slight stress overshoot). The increase in shear stress with shear strain for suspensions of only magnetizable spheres has been attributed to the deformation of field-induced structures [Klingenberg et al. (1991a)]. As ϕ_N^A is increased, the strain-dependent shear stress behaves similarly, exhibiting a transient increase to a steady-state value for shear strains of roughly 1. The steady-state stresses increase with ϕ_N^A, exhibiting the well-established stress enhancement by the addition of nonmagnetizable spheres. This behavior is similar to that exhibited by jammed, hard-sphere dispersions in that the stress is shear-induced (it is not apparent at $\gamma = 0$), and occurs over a finite strain of order 1. In this sense, the systems with only magnetizable spheres ($\phi_N^A = 0$) also exhibit behavior consistent with jammed systems; however neither MR system exhibits divergent stresses. That the stress in mixtures should increase at least as slowly as the systems with only magnetizable spheres is perhaps expected, because the field-induced structures must deform before any field-induced stresses appear, and the enhancement caused by the nonmagnetizable spheres is a field-induced stress.

Similar behavior is observed for three-dimensional systems, as illustrated in Fig.

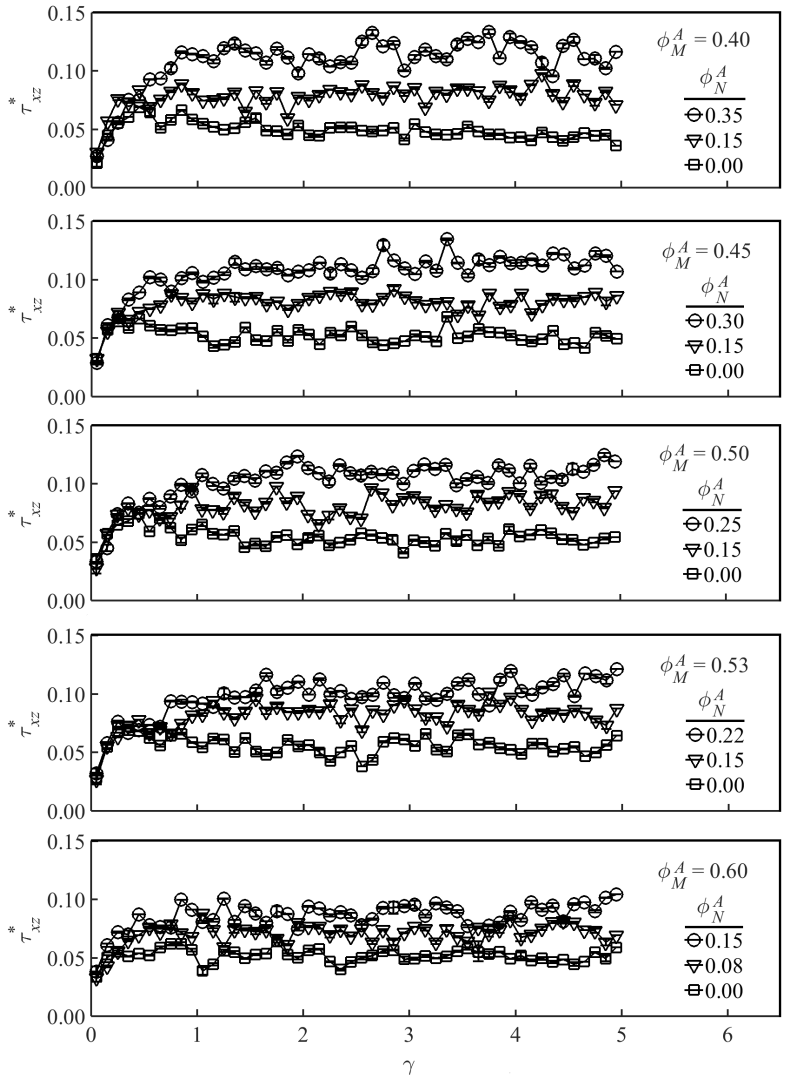

Figure 4.20: Stress as a function of strain for various values of ϕ_N^A and ϕ_M^A for monolayer suspensions.

4.21 where the quasistatic shear stress is plotted as a function of shear strain for various values of ϕ_M and ϕ_N. The enhancements in stress caused by the nonmagnetizable spheres is smaller for these three-dimensional systems than those depicted in Fig. 4.20. The stresses also appear to reach steady-state at somewhat smaller strains than those observed for monolayer system, but nonetheless the behavior is similar to that observed in jammed, hard-sphere dispersions.

4.5 Conclusion

This study was an attempt to understand the mechanism(s) by which nonmagnetizable spheres enhance the field-induced shear stress in MR suspensions. Previous work [Chapter 3] illustrated that the nonmagnetizable spheres do not produce a significant change in the microstructure of the magnetizable spheres, and in fact, produce small but different changes in the structure of monolayer and three-dimensional systems.

Large amplitude oscillatory shear simulations show that the nonmagnetizable spheres increase the suspension stiffness, but do not significantly alter the transition to nonlinear rheological behavior. These results suggest that the nonmagnetizable spheres directly participate in the stress transfer, as opposed to altering the stability of clusters of magnetizable spheres.

Snapshots of sheared monolayers reveal that the nonmagnetizable spheres participate in repulsive-force clusters with force chains roughly oriented along the compression axis of the shear flow. This behavior is similar to that observed in jammed, hard-sphere dispersions, where the shear induced repulsive-force chains orient along the compression axis. Examination of partial stresses, repulsive-force cluster num-

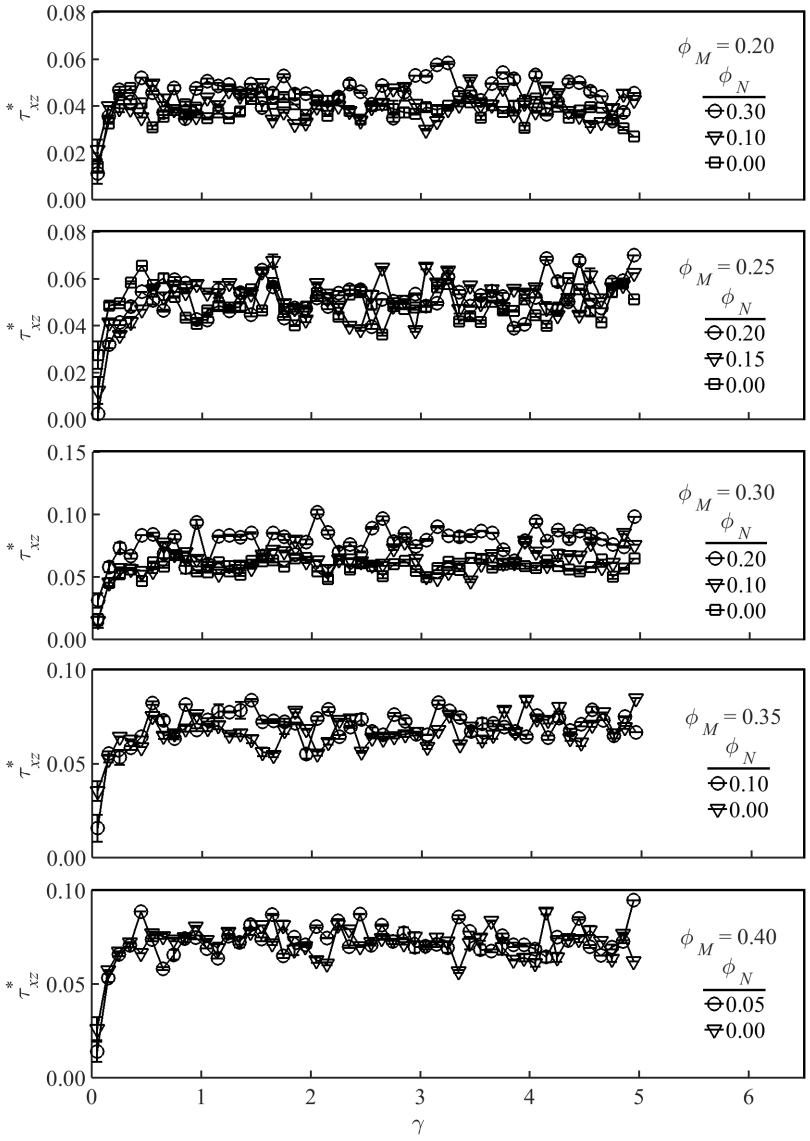

Figure 4.21: Stress as a function of strain for various values of ϕ_N and ϕ_M for three-dimensional suspensions.

bers, and transient rheological behavior all support the notion that nonmagnetizable spheres directly enhance the stress via repulsive-force clusters. The repulsive-force clusters contain both magnetizable and nonmagnetizable spheres, which likely explains the observation that the nonmagnetizable spheres enhance the field-induced stress, even though they are not magnetizable. The participation of the magnetizable in these clusters also tends to increase the magnetostatic contribution of magnetic sphere contribution to the total stress.

Chapter 5

Overview of Parallel Computing in CUDA

5.1 Introduction

Prior to 2011, all magnetorheological (MR) suspensions simulated by the Klingenberg group were performed using sequential algorithms written in FORTRAN. Beginning in 2011, we developed algorithms in parallel which could simulate MR suspensions. We developed parallel algorithms to simulate MR suspensions using the Compute Unified Device Architecture (CUDA) platform developed by graphics card manufacturer NVIDIA. In 2007, NVIDIA began enabling their graphics cards to perform scientific computing. To make parallel computing more accessible, NVIDIA developed the programming language CUDA to be used exclusively on their graphics cards for general purpose computing. CUDA is a language, based on C, that has extensions which enable the user to perform scientific calculations on an NVIDIA graphics card. A

graphics card, also referred to as a graphics processing unit (GPU) uses hundreds of arithmetic logic units (ALUs) to power a display, making it essentially a processor with hundreds of cores [Garland et al. (2008)]. Therefore, the graphics card is a natural choice for performing massively parallel calculations [Taufer et al. (2010)].

Many problems existed in the early versions of CUDA. Codes were not very portable. All CUDA capable GPUs released prior to compute capability 2.0 were unable to print to screen. Few libraries existed which could take advantage of the parallel architecture. Simulations lacked reproducibility [Taufer et al. (2010)]. However, with the release of GPU compute capability 2.0 in 2011, many of these issues began to be corrected. For instance, NVIDIA enabled the GPUs to be able to print to screen, which served to make debugging easier. With the release of CUDA 4.0, NVIDIA began including the *Thrust* library, which contains common algorithms optimized to run in parallel on NVIDIA GPUs. With each new version of CUDA, NVIDIA includes more libraries and functionality. As of December 2015, the most advanced version of CUDA is CUDA 7.5, which includes libraries that perform Fast Fourier Transforms (FFT) and LU decomposition solvers, among others.

CUDA is easiest to learn when the user has a good understanding of C. From there, learning the CUDA syntax is relatively straightforward. For instance, to dynamically allocate a block of memory on the CPU in C, the command `malloc()` is commonly used. To dynamically allocate a block of memory on the GPU in CUDA, the command `cudaMalloc()` can be used. Allocating memory is only one of many examples in which NVIDIA mirrors a CUDA command off of a traditional C command.

The main difficulties incurred when developing in CUDA occur when developing algorithms that can exploit the parallel architecture of the graphics card. Therefore,

the purpose of this chapter is to assist future students to develop a basic understanding of parallel computing in CUDA. For a more thorough introduction to CUDA, please consult Sanders and E. Kandrot (2010) and NVIDIA Corporation (2015).

5.2 CUDA: Simple Algorithms

To be able to use CUDA, the user first needs access to a CUDA capable GPU. A local GPU is the easiest way to create and debug CUDA codes. Installing the latest version of CUDA is also very useful because it allows the user to take advantage of the latest functionality; common algorithms such as a parallel FFTs do not need to be developed by the user.

In C, a function can be used to enclose a specific computation such that it can be implemented easily [Kernighan et al. (1988)]. A kernel is the CUDA equivalent to the function in C. However, there are some key differences between a function in C and a kernel in CUDA. To understand these differences, Figs. 5.1 and 5.2 show sample code for two programs for vector addition: one in C and one in CUDA. Figure 5.1 shows two vectors added using a sequential algorithm written in C. Figure 5.2 shows two vectors added using a parallel algorithm written in CUDA.

The code in Fig. 5.1 shows how a typical vector addition algorithm might look. The function is declared before the main body of the code. Inside the main function, arrays `a`, `b`, and `c` are first declared as pointers. A block of memory associated with each pointer is then allocated using the command `malloc()`; these blocks of memory are used to store the elements of the arrays `a`, `b`, and `c`. Each element of arrays `a` and `b` is assigned a value before calling the function `vector_add()`. In `vector_add()`,

each element of array `c[i]` is assigned the value of the sum of `a[i]` and `b[i]`. Each array is then deallocated using the command `free()`.

The vector addition algorithm in C is very straight forward and, for small values of n, is very fast. However, this algorithm scales as $O(n)$, so as n gets large, the algorithm will slow down. For simple algorithms in which `vector_add()` is only executed once, $O(n)$ scaling is of only minor concern. However, if `vector_add()` is repeated multiple of times, as is often the case in a simulation, smaller scaling becomes imperative to reduce simulation time.

Taking advantage of the architecture of the graphics card allows vector addition to be reduced from $O(n)$ to $O(1)$. Vector addition is a highly parallelizable operation. Each element in an array is independent of all other elements in the same array (i.e. `a[1]` does not depend on the value `a[n]`). As a result, vector addition can be performed in CUDA in a single step, shown in Fig. 5.2. The CUDA code in Fig. 5.2 appears much more complicated than the C code in Fig. 5.1. However, both codes perform the same vector addition.

Figure 5.3 is a flowchart included to assist in conceptualizing vector addition in parallel. In Fig. 5.3, each array is represented by a rectangle with four elements. Each arrow represents a thread accessing an element of the array. Since there are only 4 elements in each array in the present example, only four threads are needed to complete the vector addition.

CUDA assigns each thread its own unique identification number. The identification number can be used by the developer to instruct the thread which element of an array to access. The thread is then performing the necessary calculation. For instance, in Fig. 5.3, the thread with identification number 3 will access the memory

```c
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

void vector_add(int */*a*/, int */*b*/, int */*c*/, int /*n*/);

int main(void)
{
    int *a = NULL, *b = NULL, *c = NULL;
    int  i       ,  n = 4;

    a = (int *)malloc(n * sizeof(int));
    b = (int *)malloc(n * sizeof(int));
    c = (int *)malloc(n * sizeof(int));

    for (i = 0; i < n; i++)
    {
        a[i] = i;
        b[i] = i * i;

    }

    vector_add(a, b, c, n);
    free(a);
    free(b);
    free(c);
}

void vector_add(int *a, int *b, int *c, int n)
{
    int i;

    for (i = 0; i < n; i++)
    {
        c[i] = a[i] + b[i];
    }
}
```

Figure 5.1: Vector addition performed in C by a serial algorithm.

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>

#define BLOCK_SIZE 1024

__global__ void vector_add_cuda(int *a, int *b, int *c, int n)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;

    if (tid < n)
    {
        c[tid] = a[tid] + b[tid];
    }

}

int main(void)
{
    int *a_h = NULL, *b_h = NULL, *c_h = NULL;
    int *a_d = NULL, *b_d = NULL, *c_d = NULL;
    int  n = 4;

    a_h = (int *)malloc(n * sizeof(int));
    b_h = (int *)malloc(n * sizeof(int));
    c_h = (int *)malloc(n * sizeof(int));

    cudaMalloc((void**)&a_d, n * sizeof(int));
    cudaMalloc((void**)&b_d, n * sizeof(int));
    cudaMalloc((void**)&c_d, n * sizeof(int));

    for (i = 0; i < n; i++)
    {
        a_h[i] = i;
        b_h[i] = i * i;

    }

    cudaMemcpy(a_d, a_h, n*sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, b_h, n*sizeof(int), cudaMemcpyHostToDevice);

    vector_add_cuda<<<1, n>>>(a_d, b_d, c_d, n);

    cudaMemcpy(c_h, c_d, n*sizeof(int), cudaMemcpyDeviceToHost);

    free(a_h);
    free(b_h);
    free(c_h);


    cudaFree(a_d);
    cudaFree(b_d);
    cudaFree(c_d);
}
```

Figure 5.2: Vector addition performed in CUDA using a parallel algorithm.

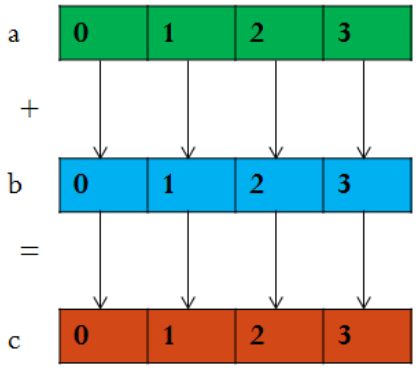

Figure 5.3: Depiction of vector addition performed in parallel. Each thread access and operates on array elements according to the thread identification number.

locations associated with `a[3]` and `b[3]`, add them together, then store the result in the memory location associated with `c[3]`.

An immediately noticeable difference between the C code in Fig. 5.1 and the CUDA code in Fig. 5.2 is that the kernel, `vector_add_cuda()`, is both declared and written before the main function. Furthermore, the CUDA main function has twice as many allocated arrays the main function in C. In Fig. 5.2, for each vector of interest, an array is created both on the host (the CPU), designated by `_h`, and on the device (the GPU), designated by `_d`. Since the arrays `a_h`, `b_h`, and `c_h` reside on the host, they are declared using `malloc()` just as the arrays `a`, `b`, and `c` were declared for the C code in Fig. 5.1. To perform calculations on the device, arrays must also be allocated in the memory of the graphics card. The command `cudaMalloc` is used to allocate vectors `a_d`, `b_d`, and `c_d`. To better understand the syntax of `cudaMalloc`, please see Sanders and E. Kandrot (2010) and NVIDIA Corporation (2015).

The arrays allocated on the GPU contain no initial values. The values stored in each element must either be modified on the graphics card or by copying values from existing arrays on the host. In Fig. 5.2, the values of `a_h` and `b_h` are copied to `a_d`

and `b_d` using the function `cudaMemcpy()`. In the syntax for `cudaMemcpy()`, the first variable is the target location to send the data. The second variable is the current location of the data to be sent. The third variable is the size of the amount of data to be sent; in this case, all elements of the vector are to be sent. The final element of `cudaMemcpy()` tells the compiler which direction the data is being transferred; in this case, data is being transferred from the host to the device.

Calling a kernel in CUDA is similar to calling a function in C; however, a kernel call has key differences. The most noticeable difference is the angle brackets which surround `«<1, n»>`. These angle brackets are used to allocate the threads and blocks necessary to complete the desired calculations. Equation 5.1 illustrates the syntax of declaring threads and blocks.

```
«< number of blocks, number of threads per block »>        (5.1)
```

First, the number of blocks needed to perform the desired calculations is specified. Then the number of threads per block is specified. In Fig. 5.2, `n` indicates that there are n threads per block, and the 1 indicates that only one block is allocated. As of December 2015, the maximum number of blocks available is $65,535$. The maximum number of threads allowed per block is 1024 [NVIDIA Corporation (2015)]. Since $n = 4$ in Fig. 5.2, only one block is necessary.

In C, variables are passed by value to a function via parenthesis, shown in Fig. 5.1. In much the same way, variables are passed by value in CUDA to the kernel via parenthesis. In kernels, the device can only perform calculations with variables stored in its memory. As mentioned previously, arrays must either be calculated on

the device or copied from the host. However, scalars can be passed without performing a `cudaMemcpy` to the device. In Fig. 5.2, `a_d`, `b_d`, and `c_d` exist on the card; the scalar `n` exists on the host but is passed to the kernel through the parenthesis in `vector_add_cuda()`.

Once the kernel is called in the main function, the vector addition begins. In Fig. 5.2, the variables `threadIdx.x`, `blockDim.x`, and `blockIdx.x` are all variables which are native to CUDA; they are not specified by the user. The variable `blockIdx.x` indicates which block on the graphics card a particular calculation is to be performed. The variable `threadIdx.x` identifies which thread on `blockIdx.x` will perform the desired calculation. The largest `threadIdx.x` available on a block depends on the number of threads per block declared in the kernel call. In Fig. 5.2, the largest `threadIdx.x` is 3, since indexing begins at 0. The variable `blockDim.x` is also specific to CUDA and is the number of threads in a block, which is specified in the kernel call. In Fig. 5.2, `blockDim.x` = 4. Since only one block is called in Fig. 5.2, `blockIdx.x` = 0. Therefore, the variable `tid` can only be 0, 1, 2, or 3. While not immediately obvious, `tid` is declared in case an array contains more elements than there are threads on a block; this point will be clarified in the following paragraphs. Also, typing `tid` is much shorter and faster than continually typing `threadIdx.x` and thus less prone to a syntax error.

In the simple vector addition shown in Figs. 5.1 and 5.2, allocating the proper number of threads is very simple since there are only four elements in each array. However, the maximum number of threads allowed on a block is 1024 [NVIDIA Corporation (2015)]. Therefore, if an array contains greater than 1024 elements, multiple blocks will be required. Sanders and E. Kandrot (2010) demonstrate a simple way

to allocate enough blocks to ensure that the proper number of threads are allocated to perform the calculations required for the particular code. Sanders and E. Kandrot (2010) first recognize that if the number of elements in an array is evenly divisible by the number of threads per block, the correct number of blocks are launched. For instance, if there are 1024 elements in an array, and 512 threads per block, by integer math $1024/512 = 2$ blocks will be launched. However, if the number of elements is not evenly divisible by the number of threads per block, too few blocks will be allocated. If instead the array of interest has 1022 elements, by integer math $1022/512 = 1$ block would be launched even though two blocks are needed. To correct for this problem, Sanders and E. Kandrot (2010) also add the number of threads per block to the number of elements before dividing by the number of threads per block. Therefore, for the example of an array with 1022 elements, adding the number of threads per block to the number of elements in the array before dividing would give $(1022 + 512)/512 = 2$ blocks launched.

On each block, thread indexing using the built in variable `threadIdx.x` begins at zero. Therefore, three different blocks are launched, and there will be three threads with `threadIdx.x`= 0. As a result, simply using `threadIdx.x` as the index would cause array element `b_d[threadIdx.x]` to be accessed by three different threads. To avoid this problem, Sanders and E. Kandrot (2010) define a unique identification number for each thread according to which block on which it is located. Sanders and E. Kandrot (2010) use the equation

```
tid = threadIdx.x + blockDim.x * blockIdx.x;          (5.2)
```

```
for(i = 0; i < n; i++)
{
    d += a[i] * b[i];
}
```

Figure 5.4: Multiplication of elements in serial.

to declare an individual thread identification number for all threads. By using the method outlined by Sanders and E. Kandrot (2010) for allocating blocks, when the number of elements in the array is not evenly divisible by the number of threads per block, more threads will be launched than elements in the array. To prevent a thread from indexing beyond the length of the array, Sanders and E. Kandrot (2010) include the if statement `if(tid < n){}`, shown in Fig. 5.2.

When finishing both C and CUDA codes, releasing arrays from memory helps to prevent values from the just completed code from corrupting future executables. In C, memory can be freed using the command `free()`. Similarly, memory can be freed in CUDA using `cudaFree()`.

Parallel vector addition is an easy algorithm to conceptually understand. Another common, but equally important, operation is the dot product. In serial, the dot product is very simple. A sample of a sequential dot product algorithm is shown in Fig. 5.4. Here, the product of each `a[i]` and `b[i]` is summed over n elements. The sum, represented as `d` in Fig. 5.4, is the dot product. Just like vector addition in serial, this is an $O(n)$ procedure.

The dot product calculation in parallel is less straight forward. Parallel algorithms are most effective when the maximum number of threads are operating on the data set [Sanders and E. Kandrot (2010)]. Therefore, multiplying them in parallel then

summing the dot product in serial is counter-productive. The summation to obtain the dot product must be performed in parallel. One way to perform a parallel dot product is to first multiply the individual elements in parallel; a procedure similar to the vector addition example except multiplication replaces addition. Then, a parallel sums reduction must be performed in parallel. Reduction is a general term given to a process that takes an input array and performs computations which results in a smaller array [Sanders and E. Kandrot (2010)]. The following reduction algorithm assumes there are 2^n elements in the array.

One way to perform an array reduction in parallel is to first launch $n/2$ threads. Each thread will then add two array elements together: the value of the array element associated with the thread identification number and an array value a specified stride length away. The stride length is divided by two and each thread again adds the value associated with its array element as well as the array element the new stride length away. The process is repeated until the stride length equals zero. A flowchart of this version of a parallel reduction is presented in Fig. 5.5

Figure 5.5 presents a concept of this parallel reduction. The shaded regions represent array elements that are being accessed on the particular step. As the reduction progresses, the stride decreases, causing fewer array elements to be added. Finally, the last two elements are added to provide the dot product stored in the thread zero. The parallel reduction algorithm written in CUDA is presented in Fig. 5.6.

The computation time of the parallel reduction code presented in Fig. 5.6 scales as $\log_2(n)$. The parallel reduction requires fewer iterations than the sequential version. However, as mentioned, the algorithm presented only works for arrays that contain 2^n elements. A parallel reduction for an arbitrary number of elements is beyond

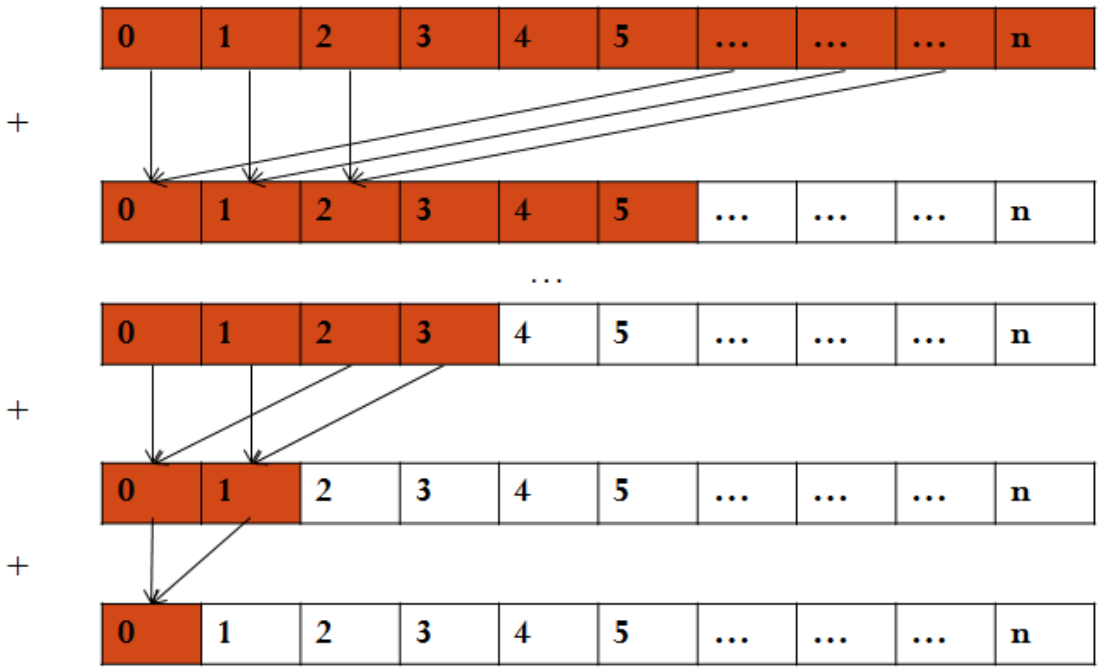

Figure 5.5: Parallel reduction for dot product.

```
__global__ void reduction(float *g_data, int n)
{
    int stride = 512;
    int tid;
    int sum = 0;

    tid = threadIdx.x + blockIdx.x*blockDim.x;

    while(stride!=0)
    {
        g_data[tid] = g_data[tid]+g_data[tid+stride];
        __syncthreads();
        stride = stride/2;
    }

}
```

Figure 5.6: Sample dot product algorithm written in serial.

the scope of this appendix; however, a description can be found in Sanders and E. Kandrot (2010).

5.3 CUDA: Particle Level Simulations

Particle level computer simulations can be used to better understand the rheology of MR suspensions. Magnetorheological suspensions can be modeled as a collection of magnetizable and nonmagnetizable spheres (monodisperse, diameter σ, magnetizable spheres with saturation magnetization M_s) immersed in a nonmagnetizable, Newtonian, incompressible, continuous phase (relative permeability $\mu = 1$, viscosity η_c), and subjected to a uniform magnetic field $\boldsymbol{H}_0 = H_0 \boldsymbol{e}_z$ [Klingenberg et al. (1991a); Kittipoomwong et al. (2005)].

The motion of the spheres can be described by Newton's equation of motion. By neglecting the inertia of sphere i, the equation of motion for sphere i can be written

$$\boldsymbol{F}_i\left(\{\boldsymbol{r}_j\}\right) = \boldsymbol{0} \tag{5.3}$$

where $\boldsymbol{F}_i\left(\{\boldsymbol{r}_j\}\right)$ is the net force on sphere i. The net force has three contributions: the magnetostatic force, the short-range repulsive force, and the hydrodynamic force. The magnetostatic force on sphere i caused by sphere j is given by the point-dipole expression

$$\boldsymbol{F}_{ij}^{\mathrm{mag}} = F_0 \left(\frac{\sigma}{r_{ij}}\right)^4 \left[\left(3\cos^2\theta_{ij} - 1\right)\boldsymbol{e}_r + \sin 2\theta_{ij}\boldsymbol{e}_\theta\right], \tag{5.4}$$

where r_{ij} is the distance between sphere i and sphere j, and θ_{ij} is the angle between the line-of-centers and the applied magnetic field. The magnitude of the force, F_0, is

given by

$$F_0 = \begin{cases} \frac{3\pi}{16}\mu_0\beta^2 H_0^2\sigma^2 & \text{linear magnetization} \\ \frac{\pi}{48}\mu_0\sigma^2 M_s^2 & \text{saturated magnetization} \end{cases}, \tag{5.5}$$

where $\beta = (\mu_p - \mu_c)/(\mu_p + 2\mu_c)$, μ_p is the relative permeability of the particle material, μ_c is the relative permeability of the continuous phase, and μ_0 is the permeability of free space. To mimic a hard-sphere interaction between spheres i and j, a short-range repulsive force on sphere i caused by sphere j is given by

$$\boldsymbol{F}_{ij}^{\text{rep}} = -F_0 \exp\left[\kappa\left(\sigma - r_{ij}\right)/\sigma\right]\boldsymbol{e}_r, \tag{5.6}$$

where κ characterizes the range of the repulsive force ($\kappa = 100$ in this study). The spheres also experience a force due to hydrodynamic drag. Following the work of Klingenberg et al. (1991a) and Kittipoomwong et al. (2005), the hydrodynamic drag is treated as Stokes' drag

$$\boldsymbol{F}_i^{\text{hyd}} = -3\pi\eta_c\sigma\left[\frac{d\boldsymbol{r}_i}{dt} - \boldsymbol{U}^{\infty}\left(\boldsymbol{r}_i, t\right)\right], \tag{5.7}$$

where $\boldsymbol{U}^{\infty}(\boldsymbol{r}_i, t)$ is the ambient fluid velocity evaluated at the particle center. The ambient fluid velocity is given by $\boldsymbol{U}^{\infty}(\boldsymbol{r}) = \dot{\gamma}(z_i^* + L_z^*/2)\boldsymbol{e_z}$ where $\dot{\gamma}$ is the strain rate.

Equation 5.3 can be nondimensionalized using the following length, force, and time scales:

$$L_s = \sigma, \quad F_s = \frac{\pi}{48}\mu_0\sigma^2 M_s^2, \quad t_s = \frac{144\eta_c}{\mu_0 M_s^2}. \tag{5.8}$$

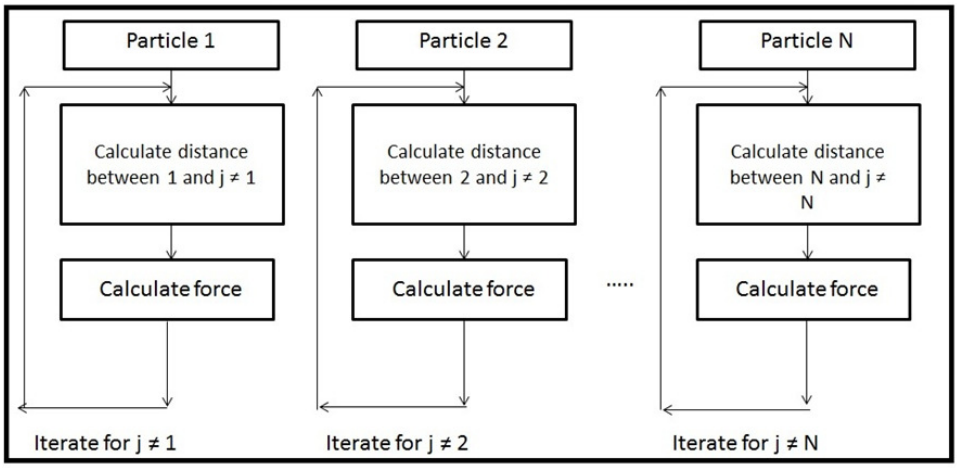

Figure 5.7: Flowchart of a particle-level simulation performed sequentially

These scales allow Eqn.5.3 to be written

$$\frac{d\boldsymbol{r}_i^*}{dt^*} = \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\mathrm{rep}} + \boldsymbol{F}_i^{*,\mathrm{wall}} + \sum_{j\neq i}^{N} \boldsymbol{F}_{ij}^{*,\mathrm{mag}} + \boldsymbol{u}^{*,\infty}, \tag{5.9}$$

where the asterisks denote dimensionless quantities.

From Eqs. 5.4, 5.6, and 5.9, the motion of a single particle, i, is dependent on the position of all other particles j. As a result, Eq. 5.9 must be solved numerically. One way to determine the interparticle interactions is to implement a sequential algorithm to calculate the net force on each sphere. A sequential algorithm contains both an outer loop and an inner loop. The outer loop is used to iterate over each particle i. The inner loop is then used to calculate the interaction between sphere i and all other spheres $j \neq i$. A flowchart of a sequential algorithm is presented in Fig. 5.7.

The algorithm presented in Fig. 5.7 is computationally expensive, $O(N^2)$. The algorithm can be improved by only considering the nearest neighbors of each sphere by implementing a neighbor list. A neighbor list decreases the computational cost of

a simulation by allowing the program to iterate only over spheres j which interact, or have the potential to interact, with sphere i [Allen and D.J. Tildesley (1989)]. A neighbor list allows the computation time of the simulation to decrease from $O(N^2)$ to $O(N^2/L^3)$, where L is the dimension of the simulation cell [Allen and D.J. Tildesley (1989)].

Even with a neighbor list, computation time for a particle level simulation remains very expensive. However, particle level simulations are highly parallelizable [Taufer et al. (2010)]. The position of each sphere i is independent of the position of each sphere j. A parallel algorithm can drastically reduce the computation time of required of MR simulations. A simple way to parallelize the simulation is to simply launch N threads and then have each thread iterate over $N-1$ spheres to calculate all $\boldsymbol{F}_{ij}$. The resulting force calculation would require $O(N)$ steps, an improvement over the sequential algorithm. Pseudocode for an $O(N)$ CUDA force calculation is shown in Fig. 5.8.

However, $O(N)$ steps is still computationally expensive. One way to reduce the number of iterations is to create a neighbor list in parallel. A neighbor list can be created by following a similar procedure to the collision detection algorithm outlined by Mazhar et al. (2011). Once the neighbor list is created, calculating the interparticle forces is straight forward and requires $O(1)$ calculations. A flowchart of the force calculation from the neighbor list is shown in Fig. 5.9.

In the neighbor list presented in this thesis, the neighbor list consists of two arrays: a `particle` array and a `neighbor` array. The `particle` array is treated as sphere i; the `neighbor` array is treated as sphere j. The `particle` array repeats the sphere identification number for the total number of possible interactions a sphere might

```
int tid = threadIdx.x + blockDim.x*blockIdx.x;

if(tid < n)
{
    for(i = 0; i < n; i++)
    {
        if(i == tid)
            continue;

        xij = x[tid] - x[i];
        yij = y[tid] - y[i];
        zij = z[tid] - z[i];

        r2 = xij*xij + yij*yij + zij*zij;

        if(r2 < rc*rc)
        {
            fx[tid] = ;//x force
            fy[tid] = ;//y force
            fz[tid] = ;//z force
        }
    }
}
```

Figure 5.8: Psuedocode for an $O(N)$ CUDA force calculation.

have. For instance, if sphere 0 has four interactions, the first four entries in `particle` will be the number 0, as illustrated in Fig. 5.9. The `neighbor` array contains all the spheres that each sphere in the `particle` array interacts. An important feature of the version of neighbor list used in these simulations is that it does not utilize the fact that the interaction $i - j$ is the equal and opposite of the interaction $j - i$; the interactions are treated separately. Since the length of the `particle` and `neighbor` arrays is not known a priori, the arrays are sized $A \times N$, where A is a factor based on the maximum number of spheres inside the neighbor list cutoff radius. A future improvement for these simulations should include reducing the memory requirement by taking advantage of the fact that interactions $i - j$ are equal and opposite to $j - i$. Reducing the memory requirement for simulations will allow for suspensions with more spheres to be studied.

To calculate the interparticle forces, $A \times N$ threads must first be launched. Each

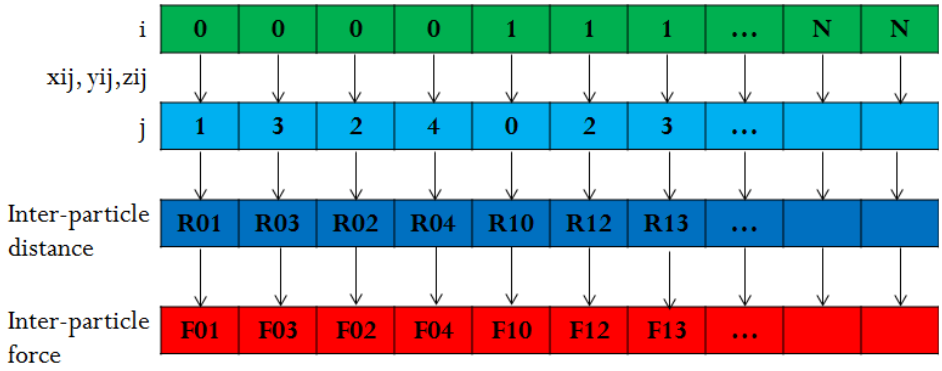

Figure 5.9: Flowchart of an interparticle force calculation in parallel.

thread then accesses `particle[tid]` and `neighbor[tid]` to obtain the interacting particles for which the respective thread will calculate $\boldsymbol{F}_{ij}$. Each $\boldsymbol{F}_{ij}$ is stored in an array that is also $A \times N$ in length. The calculation of $\boldsymbol{F}_{ij}$ is depicted in Fig. 5.9.

Calculating each $\boldsymbol{F}_{ij}$ in parallel is straight forward. However, to update the particle position, the total force acting on each sphere must be calculated. To calculate the total force on each sphere, a parallel reduction can be performed on each sphere. The `particle` array serves as a key to identify which elements of the array containing $\boldsymbol{F}_{ij}$ are associated with each sphere.

The parallel reduction by key begins with $A \times N$ threads launched. However, after the initial kernel launch, the number of threads accessing the $\boldsymbol{F}_{ij}$ data is then limited to N threads. Since the `particle` array contains multiple entries of each sphere number, only the threads associated with the first entry of a particular sphere are used when calculating the total force for the respective sphere. Identifying which thread calculates the total force on each sphere is depicted in Fig. 5.10. In Fig. 5.10, since sphere 0 ends after the third element of the `particle` array, thread `tid`= 0 calculates the total force on sphere 0 and `tid`= 4 calculates the total force on sphere

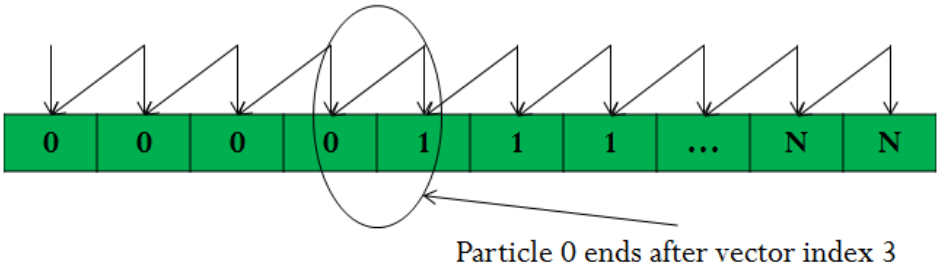

Figure 5.10: An example of a `particle` array. The `partcle` array is used as the key for the parallel reduction by key which calculates the total force.

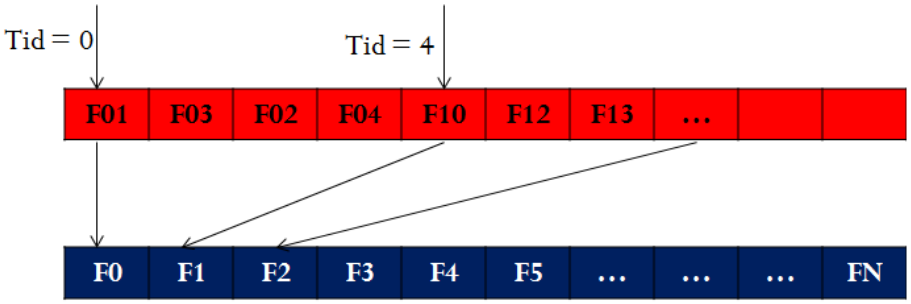

Figure 5.11: Flowchart of a parallel reduction by key.

1.

Once the first entry of a particular sphere is identified in the `particle` array, the thread associated with that entry then sums all $\boldsymbol{F}_{ij}$ associated with that sphere. The maximum number of spheres that an individual sphere can interact with is determined by the neighbor list cutoff radius. Therefore, each of the N threads will perform no more than $r^3_{\text{list}}/r^3_{\text{sphere}}$ iterations. Therefore, by creating and using a neighbor list, we were able to reduce the the scaling of the computations in iterations from $O(N)$ to $O(r^3_{\text{list}}/r^3_{\text{sphere}})$. A flowchart of the parallel reduction by key to calculate the total force on each particle is shown in Fig. 5.11. Once the force is totaled for each sphere, a simple vector addition can be used to update the positions of the spheres.

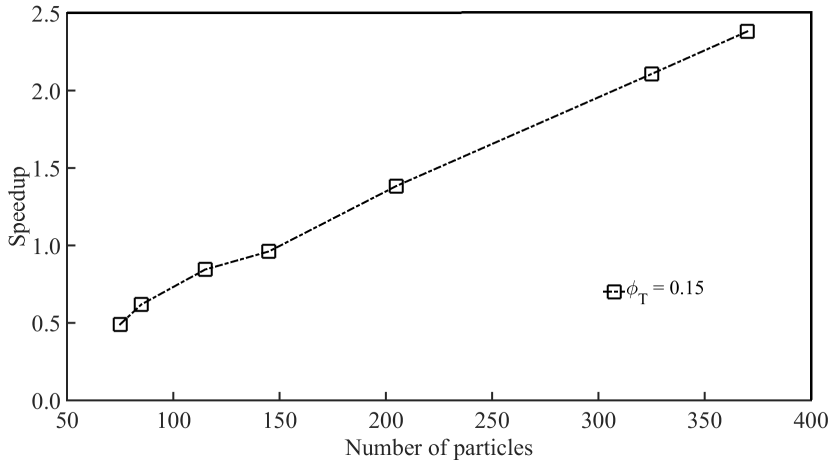

Figure 5.12: Speedup as a function of number of spheres. Suspension at fixed $\phi_M = 0.15$.

The purpose of converting algorithms from sequential to parallel is to reduce computation time. The speedup of simulations performed in CUDA over serial simulations performed in C is plotted as a function of number of spheres in Fig. 5.12. The simulations in Fig. 5.12 were performed in three dimensions and contained an MR fluid of total volume fraction $\phi_T = 0.15$. In Fig. 5.12, the parallel simulations are slower for low numbers of spheres. As the number of spheres in the suspension is increased, the speedup also increases. The break-even point at which parallel simulations perform at the same rate as sequential simulations is 150 spheres. When the suspension contains more than 150 spheres, the parallel algorithm is faster, with the speedup increasing as more particles are included in the simulation.

For simulations containing fewer than 150 spheres, the parallel simulations actually perform slower. Parallel simulations performed in CUDA require data to be transferred to the card via the PCI express bus of the motherboard, which is a slow process. Therefore, parallel simulations work best when large amounts of data are sent to the card at once. In addition, computing in CUDA is most effective when

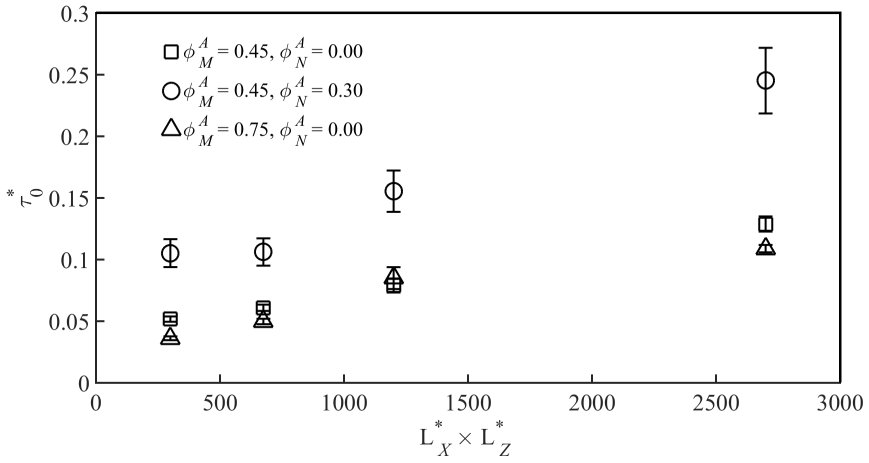

Figure 5.13: Stress as a function of monolayer area for a fixed system aspect ratio $L_x^*/L_z^* = 3$. Open squares are $\phi_M^A = 0.45$. Open circles are $\phi_M^A = 0.45$ and $\phi_N^A = 0.30$. Open triangles are $\phi_M^A = 0.75$.

as many threads as possible operate on the data at the same time [Sanders and E. Kandrot (2010)]. Both of these factors contribute to parallel simulations being less effective than sequential at low sphere numbers.

A byproduct of the observed speedup with CUDA is that systems containing more spheres can be studied. Previous sequential monolayer systems containing both magnetizable and nonmagnetizable spheres contained a maximum of 70 spheres [Ulicny et al. (2010)]. The monolayer systems presented in previous chapters contained a maximum of 287 spheres. As a result of the larger system, the yield stress enhancement caused by the addition of nonmagnetizable spheres was observed, unlike in the work by Ulicny et al. (2010) which did not contain enough spheres. Another interesting result is the effect of system size on the yield stress in the suspension, shown in Fig. 5.13.

In Fig. 5.13, stress is plotted as a function of monolayer area. For each concentration considered, when the dimensionless area is increased, which in turn increases

the number of spheres for respective concentration, the stress increases. The largest system in Fig. 5.13 contains 2500 spheres, more than 35× as many spheres as previous work [Ulicny et al. (2010)]. Furthermore, since the scaling is independent of number of spheres, the time of the simulation remains relatively constant. The effect of system size on the yield stress should be investigated more in the future.

5.4 CUDA: Creating the Resistance Matrix

A common problem in rheology involves understanding the effect of hydrodynamic interactions on the motion of particles. One method for simulating a suspension that includes hydrodynamic interactions is outlined by Ball and J.R. Melrose (1997). Their method is useful for suspensions at high volume fraction, ϕ, which are dominated by lubrication terms. At large ϕ, the lubrication and torque terms can be decoupled. The equation of motion can then be simplified to,

$$\boldsymbol{F}^{\mathrm{NH}} = \mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}}(\boldsymbol{U} - \boldsymbol{U}^{\infty}) \tag{5.10}$$

where $\boldsymbol{F}^{\mathrm{NH}}$ is the nonhydrodynamic force between the spheres, $\mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}}$ is the two body resistance matrix which only considers lubrication interactions, $\boldsymbol{U}$ is the sphere velocity, and $\boldsymbol{U}^{\infty}$ is the ambient fluid velocity. Here, $\boldsymbol{F}^{\mathrm{NH}}$ is the magnetostatic force. The two body resistance matrix, $\mathcal{R}_{2B}^{\mathrm{lub}}$, is given by,

$$\mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}} = R_0\boldsymbol{\delta} + R_{\mathrm{lub}}(h_{ij})\boldsymbol{dd} \tag{5.11}$$

where $R_0 = 3\phi\eta_c\sigma$, $R_{\text{lub}}(h_{ij}) = 3\pi\eta_c\sigma(\sigma/8h_{ij})$, and $\boldsymbol{d}$ is the unit vector along the line of centers [Kim and S.J. Karrila (2013)]. For more detail concerning these simplifications, see Chapter 6.

Equations 5.10 and 5.11 form a $3N \times 3N$ system of equations for sphere motion, where N is the number of spheres in the suspension. Creating and solving a large system of equations by a sequential algorithm is computationally intensive. To avoid large calculations, previous authors only studied suspensions at zero shear, thereby, neglecting hydrodynamic interactions [Parthasarathy and Klingenberg (1996), Kittipoomwong et al. (2005), Ulicny et al. (2010)]. However, solving large systems of equations is a highly parallelizable operation. Therefore, the equations of motion for a suspension can be solved by taking advantage of the computing power of the graphics card.

With the release of CUDA 7 in 2015, NVIDIA began including the library `cusolver`. This library includes algorithms that solve large systems of equations efficiently in parallel. A direct result of the parallel solvers is that Eq. 5.10 can be solved for the particle velocities significantly faster and with less effort from the user.

While `cusolver` makes it easy to solve a system of equations in parallel, the equations must still be created in parallel to take full advantage of the graphics card. A diagram of the resistance matrix as a two-dimensional array is given in Fig. 5.14. The numbers to the left in black represent sphere i. The numbers along the top in blue represent sphere j. The diagonal submatrices represent the self-interaction hydrodynamic term. The submatrices that are off the diagonal represent the hydrodynamic interaction between sphere i and sphere j. The colors of the submatrices only serve to differentiate between the submatrices. The numbers inside

	0			1			2		
0	0,0	0,1	0,2	0,0	0,1	0,2	0,0	0,1	0,2
	1,0	1,1	1,2	1,0	1,1	1,2	1,0	1,1	1,2
	2,0	2,1	2,2	2,0	2,1	2,2	2,0	2,1	2,2
1	0,0	0,1	0,2	0,0	0,1	0,2	0,0	0,1	0,2
	1,0	1,1	1,2	1,0	1,1	1,2	1,0	1,1	1,2
	2,0	2,1	2,2	2,0	2,1	2,2	2,0	2,1	2,2
2	0,0	0,1	0,2	0,0	0,1	0,2	0,0	0,1	0,2
	1,0	1,1	1,2	1,0	1,1	1,2	1,0	1,1	1,2
	2,0	2,1	2,2	2,0	2,1	2,2	2,0	2,1	2,2

Figure 5.14: An example of the indexing for $\mathcal{R}_{2B}^{\text{lub}}$ for three spheres.

the submatrices indicate the cartesian directions: 0 represents the x direction, 1 represents the y direction, and 2 represents the z direction.

In a sequential algorithm, the resistance matrix can be created within the force calculation loops. However, creating the resistance matrix in CUDA is much less intuitive. Unlike C, early versions of CUDA did not allow two-dimensional arrays. Array allocation and construction was simpler by treating the matrix as a long one-dimensional array. In newer versions of CUDA, such as CUDA 7.5, two-dimensional array allocation and access is similar to the capabilities of C [NVIDIA Corporation (2015)]. Therefore, both types of arrays will be considered in Subsections (5.4.1) and (5.4.2).

5.4.1 Resistance Matrix in a One-Dimensional Array

In Fig. 5.15, the resistance matrix is treated as a large one-dimensional array; only the numbers within the submatrices have changed. Just as in Fig. 5.14, the numbers

		0			1			2	
0	0	9	18	27	36	45	54	63	72
	1	10	19	28	37	46	55	64	73
	2	11	20	29	38	47	56	65	74
1	3	12	21	30	39	48	57	66	75
	4	13	22	31	40	49	58	67	76
	5	14	23	32	41	50	59	68	77
2	6	15	24	33	42	51	60	69	78
	7	16	25	34	43	52	61	70	79
	8	17	26	35	44	53	62	71	80

Figure 5.15: Representation of indexing for a one-dimensional resistance matrix.

along the left side represent sphere i, and the numbers along the top represent sphere j. Instead of representing the directions 0, 1, or 2 like in Fig. 5.14, the numbers in the submatrices represent which element in a one-dimensional array would correspond to each element in Fig. 5.14. For instance, the yz component of the $i = 1, j = 2$ sphere interaction would correspond to element 76 in a one-dimensional array.

The first step in determining how to access the elements of the one-dimensional resistance matrix is to identify which variables should be used to access the array. The index for sphere i, and the index for sphere j could be considered. The number of spheres, n, will change the size of the array, and, therefore, it should also be considered. Since $\boldsymbol{d}$ exists in three dimensions, the algorithm contains two loops which iterate over the x, y, and z directions to obtain $\boldsymbol{dd}$. Therefore, the indices associated with the two directional loops could be useful. The hydrodynamic interactions occur in three dimensions, so 3 is also useful to separate threads for x, y, and z. Once the necessary variables are determined, they must be arranged such that all elements of the resistance array are accessed. Equation 5.12 shows a way to arrange the variables

such that all elements of the resistance matrix are accessed.

```
Resistance_matrix[3 * n*(jloop + 3 * j) + iloop + 3 * i] = dd / (8 * h);
```
(5.12)

In Eq. 5.12, `iloop` and `jloop` iterate over the three dimensions occupied by the center-to-center vector $\boldsymbol{d}$. To understand this indexing, consider the xz component of the $i = 0, j = 1$ interaction in a three sphere system. We want to know the value of the xz element of the resistance matrix. For the x direction, `iloop` $= 0$ and for the z direction, `jloop` $= 2$. Following the indexing formula in Eq. 5.12, the array index for the xz element of the $i = 0, j = 1$ sphere interaction is 45. Comparing Figs. 5.14 and 5.15, index 45 does indeed correspond to the xz element of the $i = 0, j = 1$ interaction. Take note that `tid` ended up not being needed, a result of using trial and error to determine the indexing.

5.4.2 Resistance Matrix in a Two-Dimensional Array

In newer versions of CUDA, two-dimensional arrays can be created. One way to access the elements of a two-dimensional array is to take advantage of two-dimensional block and thread indexing. From Fig. 5.2, the syntax for the block and thread identification numbers is `blockIdx.x` and `threadIdx.x`. The ending of both the block and thread identification syntax is `.x`. The `.x` signifies that the indexing is in the x direction of the GPU. While the x direction is the most common direction to index blocks and threads, the user can also access a y and z direction. In the kernel, the y and z directions can be accessed by replacing `.x` with `.y` or `.z`, respectively.

Blocks can be launched in two dimensions from the kernel call [Sanders and E.

Kandrot (2010)]. To launch blocks in two dimensions, the kernel call requires a variable of type `dim3`. The variable type `dim3` is specific to CUDA; it creates a three-dimensional variable of integers. The variable `dim3` is declared via the syntax,

```
dim3(n_x, n_y, n_z)
```
(5.13)

where `n_x` is the number of threads in the x direction on the device, `n_y` is the number of threads in the y direction on the device, and `n_z` is the number of threads in the z direction on the device. If columns are treated as the x thread direction, and the rows are treated as the y thread direction, the indexing of the grand resistance matrix can be expressed,

```
Resistance_matrix[3*blockIdx.y + ty][3*blockIdx.x + tx].
```
(5.14)

In the kernel call, n blocks would be launched in both the x and y directions. From there, three threads could be launched in the x and y directions. This method would allow all elements of the resistance matrix to be calculated at the same time. Therefore, the resistance matrix could be formed in one step. With the use of `cusolver`, solving the equation of motion for each sphere including hydrodynamic interactions becomes computationally feasible.

5.5 Conclusion

The purpose of this chapter was to give future students a starting point for developing parallel algorithms in CUDA. As of December 2015, finding a CUDA capable GPU

is very easy; any NVIDIA graphics card will work. CUDA can then be installed for free from the NVIDIA website. Once CUDA is installed, parallel algorithms can be implemented without too much effort from the user. With each new version of CUDA, more built in functions and functionality are added by NVIDIA.

Particle-level simulations can be performed by algorithms developed using CUDA. Simulations performed in parallel increase speedup as particles are added to the suspension. These faster simulations allowed larger systems to be studied. Future students should investigate incorporating hydrodynamic interactions into particle-level simulations. Advances in CUDA by NVIDIA make solving for particle motion with hydrodynamic interactions approachable.

Chapter 6

Overview of Hydrodynamic Interactions

6.1 Introduction

Magnetorheological (MR) fluids consist of magnetizable particles suspended in a viscous, continuous phase. These suspensions exhibit fast and reversible changes to the stress in the fluid caused by manipulating a magnetic field. The magnetic field induces magnetostatic particle interactions which cause the particles to aggregate, changing the suspension from a fluid-like state to a solid-like state, with a magnetic field-dependent yield stress [Ginder (1996), Jolly et al. (1998)]. This is known as the "MR effect". To take full advantage of the MR effect, increasing the dynamic range of control can be done in two ways: increasing the stress in the fluid at high-fields and by decreasing the stress in the off-state [Foister (1997), Ulicny et al. (2005b)]. In the off-state, colloidal forces and hydrodynamic interactions become more significant;

the field-induced forces are not present in the system. Experiments have shown that fluids in the off-state regime exhibit a yield stress [Ulicny et al. (2005b)]. For MR fluids in the off-state, experiments have shown that coating the magnetizable particles with a nonmagnetizable material can reduce the yield stress [Ulicny et al. (2005a)]. Developing a better understanding of MR fluids in the off-state can lead to designing fluids with desirable off-state properties as well as high-field properties.

In the low-field regime, colloidal forces and hydrodynamic interactions are on the same order of magnitude as the magnetic forces. To quantify the relationship between magnetic field forces and hydrodynamic forces, the dimensionless Mason number can be defined. The Mason number is given by Klingenberg et al. (2007),

$$\mathrm{Mn}_{\mathrm{hydro}} = \frac{\text{hydrodynamic forces}}{\text{magnetic forces}} = \frac{9}{2}\frac{\eta_{\mathrm{c}}\dot{\gamma}\phi^2}{\mu_0\mu_{\mathrm{c}}\langle \mathrm{M}\rangle^2}. \tag{6.1}$$

Ulicny et al. (2005a) experimentally determined stress data for an MR fluid in the high Mn regime, presented in Chapter 2, Fig. 2.4. Figure 2.4 shows the shear stress plotted as a function of shear rate for two different systems: one in which the particles are not coated with a nonmagnetizable material, and one in which the particles are coated with a nonmagnetizable material. The open circles represent a suspension in which the magnetizable particles are not coated with a nonmagnetizable material. The open squares represent a suspension in which the magnetizable particles are coated with a nonmagnetizable material. The suspension in which the magnetizable particles are not coated with a nonmagnetizable material exhibits a yield stress of $\approx$40 Pa. However, the suspension in which the magnetizable particles are coated with a nonmagnetizable material exhibits a significantly reduced yield stress. The

curve is much closer to a constant viscosity than the suspension that is not coated.

To better understand the decrease in low-field yield stress and viscosity, simulations must include field forces, van der Waals forces, and hydrodynamic forces. The following section contains a brief overview of the model used for the MR suspensions with emphasis on hydrodynamic interactions. From there, a summary of current simulation techniques is presented.

6.2 Model

Magnetorheological suspensions are treated as a collection of magnetizable and non-magnetizable spheres (monodisperse, diameter σ, permeability μ_p, magnetizable spheres with a saturation magnetization M_s) immersed in a nonmagnetizable, Newtonian, incompressible, continuous phase (relative permeability $\mu = 1$, viscosity η_c), and subjected to a uniform magnetic field $\boldsymbol{H}_0 = H_0\boldsymbol{h}$ ($\boldsymbol{h}$ is the unit vector in the direction of the applied field).

The motion of the spheres is described by Newton's equation of motion. Neglecting the acceleration of sphere i gives

$$\boldsymbol{F}_i\left(\{\boldsymbol{r}_j\}\right) = 0. \tag{6.2}$$

where $\boldsymbol{F}_i$ is the total force acting on sphere i. Four forces can be considered in these systems: the magnetostatic force, the van der Waals force, the short range repulsive force, and the hydrodynamic force. Applying a magnetic field enables the magnetostatic force on sphere i caused by sphere j to be given given by the point

dipole expression

$$\boldsymbol{F}_{ij}^{\text{mag}} = F_0 \left(\frac{\sigma}{r_{ij}}\right)^4 \left[\left(3\cos^2\theta_{ij} - 1\right)\boldsymbol{e}_r + \sin 2\theta_{ij}\boldsymbol{e}_\theta\right]. \tag{6.3}$$

where the magnitude of the force, F_0 is given

$$F_0 = \begin{cases} \frac{3\pi}{16}\mu_0\beta^2 H_0^2\sigma^6 & \text{linear magnetization} \\ \frac{\pi}{48}\mu_0\sigma^2 M_s^2 & \text{saturated magnetization} \end{cases}, \tag{6.4}$$

where $\mu_0 = 4\pi \times 10^{-7}\text{N/A}^2$, and $\beta = (\mu_p - 1)/(\mu_p + 2)$. For low magnetic fields, the magnitude of the force is given for linear magnetization. For high magnetic fields, magnitude of the force is given for saturation magnetization. Since most applications require fluids in the high field regime, the magnitude of the force for magnetic saturation is usually chosen.

The spheres always experience a van der Waals attraction. When the field is close to saturation, these van der Waals attractions are dwarfed by the field forces, allowing them to be neglected. However, when the field is low or off, the van der Waals attractions cannot be neglected. The van der Waals attractions can be expressed [Israelachvili (2011)]

$$\boldsymbol{F}^{\text{vdw}} = \begin{cases} \frac{A}{24}\frac{\sigma}{h_{ij}^2}\boldsymbol{e}_r & \text{for} \quad h_{ij} > h_{\min} \\ \frac{A}{24}\frac{\sigma}{h_{\min}^2}\boldsymbol{e}_r & \text{for} \quad h_{ij} \leq h_{\min} \end{cases} \tag{6.5}$$

where h_{ij} is the gap distance between spheres i and j, and A is the Hamaker coefficient, which is a material property.

Much work has been performed at vanishing shear rates (low Mn) [Klingenberg

et al. (1991a), Kittipoomwong et al. (2005)]. At vanishing shear rates, the hydrodynamic forces are expected to have little impact on the final structure of the fluid. As a result, the hydrodynamic force can be expressed using the free-draining approximation, given by Stokes' drag $\boldsymbol{F}^{\mathrm{H}} = -3\pi\mu\sigma\left(\boldsymbol{U} - \boldsymbol{U}^{\infty}\right)$, where $\boldsymbol{U}$ is the translational velocity of the sphere and $\boldsymbol{U}^{\infty}$ is the ambient velocity of the fluid. However, at nonzero shear rates, the free-draining approximation is not enough to model the physics of the suspension. The force exerted on the fluid due to sphere motion decays very slowly. Therefore, at nonzero shear, the motion of one sphere interacts with the motion of the other spheres via the hydrodynamic forces imparted on the viscous fluid by the spheres [Russel et al. (1992)].

The interaction of the fluid with the spheres can be described using three quantities: the hydrodynamic force $\boldsymbol{F}^{H}$, the torque $\boldsymbol{T}$, and the stresslet $\underline{\underline{\boldsymbol{S}}}$ [Kim and Karrila (1991)]. These three quantities can be related to the particle motion in two different ways: the resistance problem and the mobility problem. In the resistance problem, the motion of the spheres is specified as the boundary condition. Following Kim and Karrila (1991), the disturbance velocity, $\boldsymbol{v}^{D}\left(\boldsymbol{x}\right)$, on the surface of the particle at position $\boldsymbol{x}$ is given by

$$\boldsymbol{v}^{D}\left(\boldsymbol{x}\right) = \boldsymbol{U} - \boldsymbol{U}^{\infty} + \left(\boldsymbol{\omega} - \boldsymbol{\Omega}^{\infty}\right) \times \boldsymbol{x} - \underline{\underline{\boldsymbol{E}}}^{\infty} \cdot \boldsymbol{x}, \tag{6.6}$$

where $\boldsymbol{\omega}$ is the rotational velocity of the particle, $\boldsymbol{\Omega}^{\infty}$ is the ambient rotational velocity of the fluid, and $\underline{\underline{\boldsymbol{E}}}^{\infty}$ is the rate of strain tensor of the fluid. Due to the linearity of Stokes' equations, the resistance problem can be expressed as a system of linear

equations, given by

$$\begin{pmatrix} \boldsymbol{F}^H \\ \boldsymbol{T}^H \\ \underline{\underline{\boldsymbol{E}}}^\infty \end{pmatrix} = -\mathcal{R} \cdot \begin{pmatrix} \boldsymbol{U} - \boldsymbol{U}^\infty \\ \boldsymbol{\Omega} - \boldsymbol{\Omega}^\infty \\ \underline{\underline{\boldsymbol{S}}} \end{pmatrix} \tag{6.7}$$

where $\mathcal{R}$ is the grand resistance matrix.

The generalized mobility formulation involves specifying the hydrodynamic force, $\boldsymbol{F}^H$, hydrodynamic torque, $\boldsymbol{T}^H$, and rate of strain tensor $\underline{\underline{\boldsymbol{E}}}^\infty$ as boundary the boundary conditions used to calculate the particles' translational velocity, angular velocity, and stresslet Kim and S.J. Karrila (2013). The generalized mobility relation is written

$$\begin{pmatrix} \boldsymbol{U} - \boldsymbol{U}^\infty \\ \boldsymbol{\Omega} - \boldsymbol{\Omega}^\infty \\ \underline{\underline{\boldsymbol{S}}} \end{pmatrix} = -\mathcal{M} \cdot \begin{pmatrix} \boldsymbol{F}^H \\ \boldsymbol{T}^H \\ \underline{\underline{\boldsymbol{E}}}^\infty \end{pmatrix} \tag{6.8}$$

where $\boldsymbol{\Omega}^\infty$ is the ambient angular velocity and $\mathcal{M}$ is the grand mobility matrix. The resistance matrix is related to the mobility matrix by $\mathcal{R} = \mathcal{M}^{-1}$. The complete expressions for the grand resistance and grand mobility matrices can be found in Kim and Karrila (1991).

6.3 Simulation Methods

6.3.1 Calculating Hydrodynamic Interactions

To address the computational challenges associated with simulating suspensions, Brady and Bossis developed the algorithm Stokesian dynamics [Brady and G. Bossis (1988)]. Stokesian dynamics is a molecular dynamics-like simulation technique which

leverages both the mobility and resistance formulations to simulate the rheological behavior of suspensions.

Both the mobility formulation and the resistance formulation have advantages and disadvantages. The mobility formulation conveniently preserves the far-field hydrodynamic interactions. Also, the velocity can be solved without inverting a matrix. Matrix inversion is computationally expensive, requiring $O(N^3)$ steps. However, the mobility formulation does not preserve the near-field lubrication interactions. The inability of the mobility formulation to preserve the lubrication forces leads to the spheres overlapping [Bossis and J.F. Brady (1984)]. Unlike the mobility formulation, the resistance formulation preserves the lubrications interactions and also accurately represents the underlying physics of the problem. However, to obtain sphere velocities in the resistance formulation, Eq. (6.7) must be solved for velocities. This calculation is computationally expensive.

The far-field is most conveniently expressed in the mobility formulation. The far-field mobility matrix, denoted $\mathcal{M}^{\infty}$, is constructed and then inverted to give an approximation for the far-field resistance matrix. Since M^{∞} is sparse, the inversion can be done efficiently. The resistance formulation is needed to preserve the near-field lubrication forces. Lubrication forces occur between two closely-spaced bodies, which allows them to be treated as pairwise additive. The lubrication forces can be represented by the two-body resistance matrix, $\mathcal{R}_{\mathrm{2B}}$. In addition to the lubrication forces, the two-body resistance matrix also includes far-field two-body interactions which are already accounted for in $(\mathcal{M}^{\infty})^{-1}$; therefore, the two body far-field interactions must be subtracted off, and are expressed as $\mathcal{R}^{\infty}_{\mathrm{2B}}$. The grand resistance matrix can be

expressed

$$\mathcal{R} = (\mathcal{M}^{\infty}) + \mathcal{R}_{2\mathrm{B}} - \mathcal{R}_{2\mathrm{B}}^{\infty}. \tag{6.9}$$

Equation (6.9) can be used in Eq. (6.7) to give Newton's equation of motion for a collection of spheres interacting through hydrodynamic interactions. The sphere velocities can be found by solving Eq. (6.7) and the sphere positions updated. The bulk properties of the suspension can then be calculated using an ensemble average of the sphere positions [Batchelor, G.K. (1970)].

Sierou and Brady implemented an improved version of Stokesian dynamics which reduced the number of iterations from $O(N^3)$ for traditional Stokesian dynamics to $O(N\ln(N))$. To accomplish the reduction in iterations, the far-field interactions were calculated using Ewald summations [Sierou and J.F. Brady (2001)]. Ewald summations reduce the number of iterations by replacing calculations that converge very slowly with calculations that converge rapidly [Sierou and J.F. Brady (2001), Frenkel and Smit (1987)]. Despite this improvement, Stokesian dynamics is still very computationally expensive. Ball and J.R. Melrose (1997) showed that for highly concentrated suspensions ($\phi_T > 0.40$), the near-field lubrication forces dominate the resistance matrix calculation. Therefore, to leading order, only the near-field lubrication interactions need be included in determining the resistance matrix. Furthermore, translational and rotational motion become decoupled enabling the resistance matrix to be expressed

$$\mathcal{R}_{2\mathrm{B}}^{\mathrm{lub}} = R_0\boldsymbol{\delta} + R_{\mathrm{lub}}(h_{ij})\boldsymbol{dd} \tag{6.10}$$

where $R_0 = 3\pi\eta_c\sigma$, $R_{\mathrm{lub}}(h_{ij}) = 3\pi\eta_c\sigma(\sigma/8h_{ij})$, and $\boldsymbol{d}$ is the unit vector along the line of centers [Kim and Karrila (1991)]. Equation (6.10) can be substituted into Eq.

(6.7) to give

$$F^{\mathrm{H}} = -\mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}}(\boldsymbol{U} - \boldsymbol{U}^{\infty}). \tag{6.11}$$

Since the total force on each particle is zero, the equation of motion can be expressed

$$F^{\mathrm{NH}} = \mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}}(\boldsymbol{U} - \boldsymbol{U}^{\infty}). \tag{6.12}$$

where $\boldsymbol{F}^{NH}$ represents the total nonhydrodynamic force.

6.3.2 System Parameters

The MR suspensions consist of N neutrally buoyant spheres in a volume $L_x \times L_y \times L_z$. The spheres are bounded at $\pm L_z^*/2$ by a solid surface and periodic boundaries at $\pm L_x^*/2$ and $\pm L_y^*/2$. The spheres are given random initial positions.

Spheres within 0.05σ of a bounding surface are considered stuck and assume the lateral velocity of the surface. Spheres have been experimentally observed to stick to the bounding surface [Klingenberg and C.F. Zukoski (1990)]. Furthermore, since the motion of each sphere in the z direction is governed by Eq. (6.12), stuck spheres can be removed from the surface.

Using the initial positions, the nonhydrodynamic force on each sphere and the resistance matrix can be calculated. This creates a $3N \times 3N$ system of equations that must be solved numerically to obtain velocity.

6.3.3 Numerical Methods

Solving a large system of equations is computationally intensive. Exact procedures, such as Gauss-Jordan elimination, require $O(N^3)$ iterations [Press et al. (1986)].

Basic iterative procedures, such as Gauss-Seidel, require $O\left(N\ln N\right)$ iterations [Press et al. (1986)]. As a result, all numerical schemes limit the size of the system.

The resistance matrix is symmetric positive definite. Furthermore, since the suspensions of interest are at high concentration and lubrications forces dominate the hydrodynamics, $\mathcal{R}_{\mathrm{2B}}^{\mathrm{lub}}$ is sparse. These two features allow for more advanced iterative schemes to be used such as GMRES or Conjugate Gradients. Convergence of these algorithms is dependent on many factors discussed in more detail elsewhere [Trefethen, L.N. and D. Bau III (1997)].

Previously, solving large systems of equations has also faced hardware limitations. More specifically, calculations could only be made sequentially, regardless if they were independent. Most calculations involving matrix operations are independent and thus highly parallelizable. The increased availability of multiple core processors enables some of these matrix manipulations to be performed at the same time. However, to date, the largest number of cores available for a multiple core processor is eight [Intel (2015)]. Therefore, a matrix of $3N \times 3N$ is still limited to only eight calculations per computer clock cycle.

Another innovation in algorithm parallelization is the ability to perform calculations by leveraging the computing capability of the computer's graphics card. In 2007, graphics card company NVIDIA developed a new architecture for their graphics cards which allowed the user to create highly parallelizable algorithms which could be performed on the graphics card [Sanders and E. Kandrot (2010)]. To go with this new architecture, NVIDIA developed a coding language known as Compute Unified Device Architecture (CUDA). CUDA is a language based on C and has extensions enabling the graphics card to be used for scientific computing [Sanders and E. Kandrot

(2010)]. Since CUDA is based on C, it makes it more accessible than other parallel computing languages (OpenGL, OpenCL, etc...). Furthermore, the high demand for enhanced video game graphics has driven graphics card companies, like NVIDIA, to produce more advanced graphics cards that are also more affordable. The upshot is that more powerful computation is available at a lower cost to scientists [Sanders and E. Kandrot (2010)].

Since many linear algebra operations are highly parallelizable, NVIDIA has created libraries in CUDA that perform many of the standard linear algebra operations. The algorithms for these linear algebra operations are optimized to leverage the parallel capabilities of the graphics card. In addition, with the release of CUDA 7.0 in the spring of 2015, the library cuSOLVER became available. cuSOLVER contains parallel algorithms designed to solve large systems of equations in parallel. CUDA 7.0 is the first release of CUDA with built in solvers, so future releases of CUDA will most likely see inclusion of additional and more advanced solvers.

Chapter 7

Conclusions and Future Work

The purpose of this study was to understand and describe the mechanism(s) by which nonmagnetizable spheres enhance the field-induced shear stress in MR suspensions. We have employed a particle-level simulation technique to probe the effect of nonmagnetizable spheres on MR suspensions that contain a mixture of magnetizable and nonmagnetizable spheres. Both monolayer and three-dimensional suspensions exhibit a yield stress enhancement when nonmagnetizable spheres are added to the suspension. Previously, monolayers were unable to show a yield stress enhancement for suspensions containing both sphere types [Ulicny et al. (2010)]. We characterized the microstructure of the suspensions by several measures, including volume fraction fluctuations, pair distribution functions, and eigenvalues of the second-order mass moment tensor. Nonmagnetizable spheres cause monolayers to become more anisotropic. However, in three dimensions, nonmagnetizable spheres make the suspensions less anisotropic. Even though nonmagnetizable spheres cause different changes to the microstructure in monolayers and three-dimensional suspensions, the changes are very

small (Chapter 3). The microstructure changes observed for suspensions containing magnetizable and nonmagnetzable spheres are much smaller than the microstructure changes reported for bidisperse suspensions [Kittipoomwong et al. (2005)]. Therefore, microstructure changes caused by the addition of nonmagnetizable spheres do not directly cause the yield stress enhancement.

Large amplitude oscillatory shear (LAOS) simulations show that the nonmagnetizable spheres increase the suspension stiffness. However, nonmagnetizable spheres do not alter the transition to nonlinear rheological behavior. These results suggest that the nonmagnetizable spheres directly participate in the stress transfer. Conversely, nonmagnetizable spheres do not alter the stability of the magnetizable sphere clusters.

Snapshots of sheared monolayers reveal that the nonmagnetizable spheres participate in stress transfer by forming repulsive-force clusters that are oriented along the compression axis of the shear flow, similar to jamming. In hard-sphere dispersions, jamming occurs when shear-induced repulsive-force clusters form along the compression axis [Cates et al. (1998)]. Examination of partial stresses, the number of repulsive-force clusters, and transient rheological behavior support that nonmagnetizable spheres directly enhance the stress via repulsive-force clusters. The repulsive-force clusters contain both magnetizable and nonmagnetizable spheres; this explains why nonmagnetizable spheres enhance the yield stress even though they are unaffected by the magnetic field. The participation of the nonmagnetizable spheres in these repulsive-force clusters also tends to enhance the magnetostatic contribution of magnetic sphere contribution to the total stress, shown in the partial stresses calculated by force type.

The majority of the data presented has been for suspensions that are small in size

(< 300 spheres). Previous studies were limited to even smaller numbers of spheres due to the computational limitations of the sequential FORTRAN algorithms [Kittipoomwong et al. (2005), Kittipoomwong et al. (2008), Ulicny et al. (2010)]. In Chapter 6, we demonstrated that with the parallel simulations in CUDA can perform bigger simulations in a fraction of the time of the FORTRAN algorithms. Furthermore, in Fig. 5.13, the stress in the suspension depends on simulation size; larger suspensions lead to a larger yield stress.

To study larger suspensions, a new algorithm for creating initial configurations should be developed and implemented. The current method for creating random initial configurations is similar to Monte Carlo (MC) methods. The MC-like algorithms used to create initial configurations work by first randomly selecting a sphere. Then, the randomly selected sphere is moved in an arbitrary direction. The arbitrary move is then either accepted or rejected based on a criteria specified by the user. When creating initial configurations, the only criteria is that spheres do not overlap. This algorithm is currently performed sequentially and is difficult to parallelize. Furthermore, for concentrated suspensions, very few moves are accepted because most arbitrary moves will cause the random sphere to overlap with another sphere. Therefore, the initial configurations do not become very randomized. Also, suspensions containing a large number of spheres, this MC-like procedure is very slow. One possible method for creating initial configurations could involve Molecular Dynamics (MD) type algorithm. The spheres would interact via a Lennard-Jones potential [Frenkel and Smit (1987)]. Since the positions of each sphere are independent, just as simulations of MR suspensions, an MD code is highly parallelizable [Taufer et al. (2010)]. Molecular dynamics algorithms do not rely on randomly selecting a single individual

sphere at a time. Also, due to the repulsive interactions built into the Lennard-Jones potential, the spheres would move without overlapping; in the MC type of configuration creator, if a move causes the spheres to overlap, the sphere movement would be rejected and the spheres would not move.

Furthermore, the CUDA algorithms need to be upgraded to improve their memory usage. The forces in the suspension are symmetrical; the force of sphere i on sphere j is equal and opposite to the force of sphere j on sphere i. The current algorithm does not take this symmetry into account (Chapter 5). As a result, twice as much memory is used than is required for the simulation. For simulations in double precision that contain $\lessapprox$ 2500 spheres, the 2 GB memory on the current graphics cards contain enough memory to perform the simulations. However, bigger simulations require more memory than is available on all but the highest end NVIDIA cards [NVIDIA Corporation (2015)]. Therefore, the algorithms should be upgraded to take advantage of the symmetry.

Another area of interest would be the suspensions at low magnetic fields, mentioned in Chapter 2. In Fig. 2.4, suspensions coated with thiophosphate and stearate have a lower off-state viscosity than suspensions which are not coated [Klingenberg et al. (2010)]. To consider suspensions in the low-field limit, hydrodynamic interactions must be included. Previous studies only examined the high-field limit [Klingenberg et al. (1991a), Parthasarathy (1998), Kittipoomwong et al. (2005)]. In the high-field limit, the magnetostatic interactions are considered to be much greater than the hydrodynamic interactions. Therefore, the hydrodynamic interactions can be excluded when considering the high-field limit [Klingenberg et al. (1991a), Kittipoomwong et al. (2005)]. For more information concerning the simulation methods

in the high-field limit, see Chapters 3 and 4. One method for including hydrodynamic interactions is outlined by Sierou and J.F. Brady (2001) and is known as Accelerated Stokesian Dynamics (ASD). Accelerated Stokesian Dynamics is a method which uses Fast Fourier Transforms (FFT) to speed up the traditional Stokesian Dynamics (SD) algorithms originally outlined by Brady and G. Bossis (1988).

However, even though ASD provides a significant speedup over the traditional SD, for concentrated suspensions, ASD is still slow. One reason for the large computational costs is due to the inclusion of far-field hydrodynamic interactions. Ball and J.R. Melrose (1997) demonstrated that, in concentrated suspensions, the far-field hydrodynamic interactions were negligible compared to the near-field lubrication interactions. In Chapter 5, we demonstrated that parallel computing in CUDA can dramatically speedup simulations. To study suspensions in the low-field limit, the method developed by Ball and J.R. Melrose (1997) should be employed in parallel. Preliminary results from sequential code implemented in C are presented in Fig. 7.1.

In Fig. 7.1, apparent viscosity is plotted as a function of the Mason number divided by the volume fraction, ϕ. Diamonds represent three-dimensional results generated using a sequential simulation implemented in C. Squares represent data from simulations performed by Bonnecaze and J.F. Brady (1992) in two dimensions. In both systems, the same trend is observed; as Mn/ϕ is increased, η/η_∞ decreases to a plateau. The numerical similarities are purely coincidence.

Figure 7.1 demonstrates that the Ball and J.R. Melrose (1997) method can produce qualitatively similar results to SD. However, the systems studied in Fig. 7.1 are very small ($<$ 100 spheres). Therefore, a parallel algorithm should be used to include

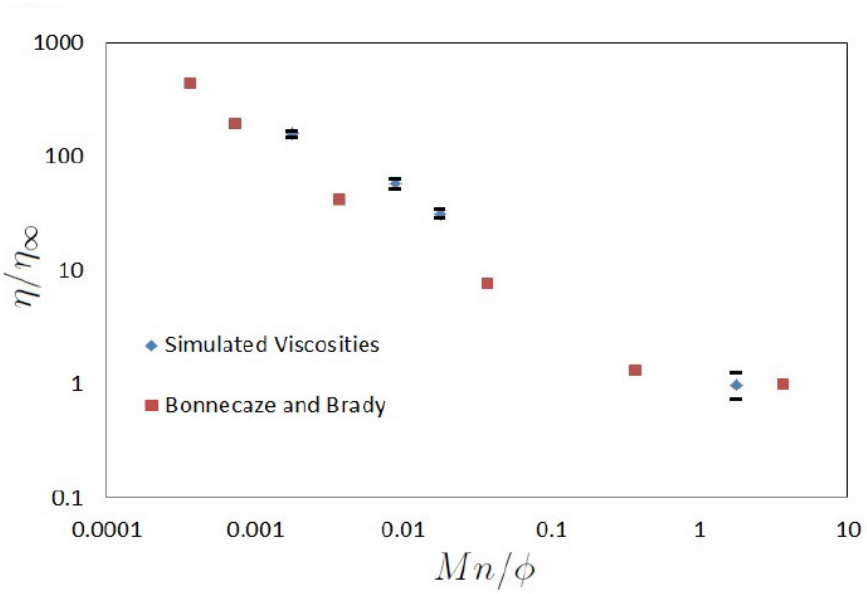

Figure 7.1: Apparent viscosity plotted as a function of the ratio of Mason number to volume fraction. Diamonds represents three-dimensional simulated using the algorithm outlined by Ball and J.R. Melrose (1997). Squares represent data reported by Bonnecaze and J.F. Brady (1992)

hydrodynamic interactions.

The effect of nonmagnetizable spheres in the low-field limit is also unknown and should be explored. Nonmagnetizable spheres will experience both short-range repulsive forces as well as lubrication interactions. Understanding nonmagnetizable sphere involvement at low magnetic fields could lead to improved MR fluids. Experiments could then be employed to determine if the simulations are indeed representative of what happens in the real system.

Appendix A

Viscoelastic Property Derivation

Viscoelasticity can be represented using a Fourier Series, expressed by,

$$\sigma(t) = \sum_{n:\text{odd}} \alpha_n \cos(n\omega t) + \beta_n \sin(n\omega t) \tag{A.1}$$

where α_n and β_n are constants. The storage modulus, G_1', and the loss modulus, G_1'', can both be calculated from α_n and β_n, using a similar procedure found in Deen (1998). The relationship between the moduli and the Fourier constants is given in Ewoldt (2013). To calculate α_n, first multiply both sides of Eq. A.1 $\cos(m\omega t)$ to give,

$$\sigma(t)\cos(m\omega t) = \sum_{n:\text{odd}} \alpha_n \cos(n\omega t)\cos(m\omega t) + \beta_n \sin(n\omega t)\cos(m\omega t)\,. \tag{A.2}$$

Both sides of Eq. A.2 can be integrated over an integer number of periods, N_p. Since sin and cos are orthogonal function, $\int_{N_p} \sin(n\omega t)\cos(m\omega t)\,dt = 0$ for all m and n. Also, for $m \neq n$, $\int_{N_p} \cos(n\omega t)\cos(m\omega t)\,dt = 0$, which allows Eq. A.2 to be

simplified to,

$$\int_{N_p} \sigma(t) \cos(n\omega t)\ dt = \int_{N_p} \alpha_n \cos^2(n\omega t)\ dt. \tag{A.3}$$

Since α_n is a constant, it can be pulled outside the integral to give,

$$\alpha_n = \frac{\int_{N_p} \sigma(t) \cos(n\omega t)\ dt}{\int_{N_p} \alpha_n \cos^2(n\omega t)\ dt}. \tag{A.4}$$

The other viscoelastic constant, β_n, can be calculated by following a similar procedure, only $\sin(m\omega t)$ is applied instead.

From Ewoldt (2013), the storage and loss moduli can be defined by

$$\begin{aligned} G'_n &= \tfrac{1}{\gamma_0}\left(\alpha_n \sin(n\delta^*) + \beta_n \cos(n\delta^*)\right) \\ G''_n &= \tfrac{1}{\gamma_0}\left(\alpha_n \cos(n\delta^*) - \beta_n \sin(n\delta^*)\right) \end{aligned} \quad n : \text{odd}, \tag{A.5}$$

where γ_0 is the strain amplitude of the simulation and δ^* is the phase shift. For the simulations performed, the phase shift was absent, $\delta^* = 0$ to give

$$\begin{aligned} G'_n &= \tfrac{1}{\gamma_0}\beta_n \\ G''_n &= \tfrac{1}{\gamma_0}\alpha_n. \end{aligned} \tag{A.6}$$

Appendix B

Mix_Strain.cu

This appendix contains the code *Mix_Strain.cu*. This code is used to strain the suspensions. Every `n_print` configuration is saved. Each configuration is then relaxed using the code *Mix_Relax.cu*.

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#include <thrust/scan.h>
#include <thrust/sort.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_free.h>
#include <thrust/device_malloc.h>

#define BLOCK_SIZE 512

//This kernel zeroes all the arrays needed for the neighbor list.
__global__ void keyzero(int *key, int *neighbor, int *particle, int n, int al)
{
//This creates a unique thread index for each thread needed to zero out
//the arrays.
   int tid = threadIdx.x + blockDim.x*blockIdx.x;

//The if statement allows only a number of threads equal to the number of
//elements in the array.
   if(tid < n)
   {
      key[tid] = 0;
   }

//The "particle" and "neighbor" arrays are padded to account for the fact
//that the total number of interactions is not known a priori. Since
//the arrays are padded, they are initialized to -1 because there is no
//-1 sphere. Since the arrays are padded, there will be more array
//elements than needed threads.
   if(tid < al*n)
   {
      neighbor[tid] = -1;
      particle[tid] = -1;
   }

}

//This kernel calculates the total number of interactions for each sphere.
__global__ void check(double *x, double *y, double *z, int *key,
                      int *neighbor, int *particle, double rl2, int n,
                      double Lx, double Ly, double Lz)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   double xij, yij, zij, r2, xmag, ymag;//, zmag;
   int i;



   if(tid < n)
   {
      for(i = 0; i < n; i++)
      {
         if(tid != i)
         {
            xij = x[i] - x[tid];
            yij = y[i] - y[tid];
            zij = z[i] - z[tid];

            xmag = fabs(xij);
            ymag = fabs(yij);
            //zmag = fabs(zij);

            if(xmag > .5*Lx)
               xij = xij*(1 - Lx/xmag);
            if(ymag > .5*Ly)
               yij = yij*(1 - Ly/ymag);

            r2 = xij*xij + yij*yij + zij*zij;


//The atomicAdd command is used to add the total number of interactions
//per sphere. The atomicAdd command prevents alternate threads from
//interfering with adding key[i].
            if(r2 < rl2)
            {
               atomicAdd( &(key[i]), 1);
            }
         }
      }
   }
}

//This kernel builds the particle vector. Note that this algorithm
//does not take into account that Fij = -Fji. Therefore, the
```

```
//particle vector will resemble something like: particle =[0; 0; 0; 0;
//0; 1; 1; 1; 1; 2; 2; 2;...-1; -1; -1] depending on how many
//interactions each particle has. Both vectors "particle" and
//"neighbor" are buffered by al*n. al stands for "array length".
___global___ void setup(double *x, double *y, double *z, int *key,
            int *particle, int n)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, beg, end;

   if(tid < n)
   {
        //end and beg combine to tell the thread how many
           //interactions each particle has
       end = key[tid];

       if(tid == 0)
          beg = 0;
       else
          beg = key[tid - 1];

       for(i = beg; i < end; i++)
          particle[i] = tid;
   }
}

//This kernel populates the neighbor list vector with which particles are
//interacting with the particle in the vector "particle".
___global___ void populate(double *x, double *y, double *z, int *neighbor,
                  int *particle, int n, double rl2, double Lx, double Ly,
                  double Lz, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int start, tpart, jspher, tcount;
   double xij, yij, zij, r2, xmag, ymag;

   if(tid < al*n)
   {

       if(particle[tid] != -1)
       {
         //tcount counts the number of iterations a thread makes when populating
          //the neighbor array
          tcount = 0;

       if((particle[tid] != particle[tid - 1] | tid == 0))
       {
          //This tells the thread the particle whose neighbors it is trying
          //to find.
          start = particle[tid];
          tpart = start;

       //Loop over all particles, however, once the number of neighbors a
          //particle has is hit, it exits the loop.
          for(jspher = 0; jspher < n; jspher++)
          {

              xij = x[jspher] - x[tpart];
              yij = y[jspher] - y[tpart];
              zij = z[jspher] - z[tpart];

              xmag = fabs(xij);
              ymag = fabs(yij);
              //zmag = fabs(zij);

              if(xmag > .5*Lx)
                 xij = xij*(1 - Lx/xmag);
              if(ymag > .5*Ly)
                 yij = yij*(1 - Ly/ymag);

              r2 = xij*xij + yij*yij + zij*zij;

              if(r2 < rl2)
              {
                 if(tpart != jspher)
                 {
                    neighbor[tid + tcount] = jspher;
                    tcount++;
                          //determine if the thread is still looking at
                          //the same particle if not, break the loop
                          //because the all neighbors have been
                          //accounted for.
                    tpart = particle[tid + tcount];
                    if(tpart != start)
                       break;
                 }
              }
          }
       }
   }
}
```

```
    }
}

//Set all the forces to zero.
___global___ void init_force(double *fx, double *fy, double *fz, int n,
                double *fxtot, double *fytot, double *fztot, int al,
                    double *tau, int *particle, int *neighbor)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;

    if(tid<n)
    {
        fxtot[tid] = 0;
        fytot[tid] = 0;
        fztot[tid] = 0;

        tau[0] = 0;
    }

    if(tid < al*n)
    {
        fx[tid] = 0;
        fy[tid] = 0;
        fz[tid] = 0;
    }
}

//This kernel calculates each nonhydrodynamic force Fij. It does NOT
//calculate the total force on each particle. This is done in a
//later kernel.
___global___ void force_calc(double *x, double *y, double *z, double *fx,
                double *fy, double *fz, int n, double rc2, double Lx,
                    double Ly, double Lz, int *neighbor, int *particle,
                    int *mag_key, double rc, double *fxtot, double *fytot,
                    double *fztot, int al)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    int i, j;
    double xij, yij, zij, xmag, ymag, zmag, r2, r;
     double zpp, ri2, ri, ri4;
    double wh, iterm, jterm, ijterm;
    double C, C2, r4, rep;
    double fxi, fyi, fzi;

if(tid < al*n)
{
    if(particle[tid] != -1)
    {
      //This part of the code is very similar to what is done in the
      //fortran codes. The only difference is that there are no
      //loops because i and j are calculated on al*n different threads.
       i = particle[tid];
       j = neighbor[tid];

       iterm = (double)mag_key[i];
       jterm = (double)mag_key[j];
       ijterm = iterm*jterm;

       zmag = fabs(z[i]);
       wh = (Lz/2.00) - zmag;

       xij = x[j] - x[i];
       yij = y[j] - y[i];
       zij = z[j] - z[i];

       xmag = fabs(xij);
       ymag = fabs(yij);

       if(xmag > (0.5 * Lx))
          xij = xij * (1 - Lx / xmag);
       if(ymag > (0.5 * Ly))
          yij = yij * (1 - Ly / ymag);

       r2 = xij*xij + yij*yij + zij*zij;


       if(r2 < rc*rc)
       {
          r = sqrt(r2);
          C = zij/r;
          C2 = 5.0*C*C;
          r4 = r*r*r*r;
          //r4 = r2*r2;

          rep = exp((1.0 - r)/0.01);

          fx[tid] = ((ijterm * (C2 - 1.0) / (r4)) - rep) * (xij / r);
          fy[tid] = ((ijterm * (C2 - 1.0) / (r4)) - rep) * (yij / r);
          fz[tid] = ((ijterm * (C2 - 3.0) / (r4)) - rep) * C;
```

```
        }

        if(wh < rc)
        {
            zpp = (z[i] / zmag) * Lz - z[j] - z[i];
            ri2 = zpp * zpp + xij * xij + yij * yij;


            if(ri2 < rc*rc)
            {
                ri = sqrt(ri2);
                C = zpp/ri;
                C2 = 5.0*C*C;
                ri4 = ri*ri*ri*ri;
                //ri4 = ri2*ri2;

                fxi = (xij / ri) * ijterm * (C2 - 1.0) / ri4;
                fyi = (yij / ri) * ijterm * (C2 - 1.0) / ri4;
                fzi = C          * ijterm * (C2 - 3.0) / ri4;

//              fxi = (xij/ri)*(C2 - 1.0)/ri4;
//              fyi = (yij/ri)*(C2 - 1.0)/ri4;
//              fzi = C*(C2 - 3.0)/ri4;

                fx[tid] += fxi;
                fy[tid] += fyi;
                fz[tid] += fzi;

            }
        }
    }
  }
}

//This totals the nonhydrodynamic force on each particle. This is a
//reduction by key.
__global__ void force_total(double *z, double *fx, double *fy,
                double *fz, double *fxtot, double *fytot,
                double *fztot, int *particle, int *mag_key,
                double Lz, double rc, int n, int *neighbor, int al)
{
  int     tid = threadIdx.x + blockDim.x*blockIdx.x;
  int     i, check, redkey;
  double sumx, sumy, sumz;

  if(tid < al*n)
  {

    if(particle[tid] != -1)
    {
      //"check" identifies which particle the thread is considering
        check = particle[tid];

      //if tid looks at particle 0 OR tid is not looking at the
        //same particle as the thread behind it. Therefore, only n
        //threads are being utilized at once to total the force on
        //each particle.
      if(tid == 0 | check != particle[tid - 1])
      {
        sumx = fx[tid];
        sumy = fy[tid];
        sumz = fz[tid];
        i = tid;

       //This makes sure that the thread is only considering one
       //particle. Once it reaches a different particle, stop
       //totaling the force.

        while(check == particle[i+1])
        {
          sumx += fx[i+1];
          sumy += fy[i+1];
          sumz += fz[i+1];
          i++;
          check = particle[i];
        }

        redkey = particle[tid];

        fxtot[redkey] = sumx;
        fytot[redkey] = sumy;
        fztot[redkey] = sumz;
      }
    }
  }
}

//This calculates the force from the wall on the particle. This
```

```
//is pretty similar to the fortran code only there is no iteration over n.
___global___ void force_wall(double *z, double Lz, double *fztot, int n,
                double rc, int *mag_key, double *fxtot, double *fytot,
                    double *tau)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   double wh, wh4, iterm, zmag, term1, term2, fzw;

   if(tid < n)
   {
      iterm = (double)mag_key[tid];
      zmag = fabs(z[tid]);
      wh = (Lz/2.00) - zmag;

      if(wh < (0.5 * rc))
      {
         wh4 = wh*wh*wh*wh;
         term1 = iterm * (1.0 / wh4) / 8.0;
         term2 = exp((0.5 - wh) / 0.01);

         fzw = term1 - term2;

         fztot[tid] += fzw * (z[tid] / zmag);

      }
   }
}

//This updates the position of each particle. This is essentially the same as
//the fortran code only in parallel.
___global___ void update_pos(double *x, double *y, double *z, double *fxtot,
        double *fytot, double *fztot, double Lx, double Ly, double Lz,
          int n, double gd, double dt, double *dabsx, double *dabsy,
          double *dabsz)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   double xmag, ymag, zmag, dx, dy, dz;


   if(tid < n)
   {

      zmag = fabs(z[tid]);

      dz = fztot[tid] * dt;

      if(zmag >= ((0.5 * Lz) - .5 - .05))
      {
         dy = 0;
         if(z[tid] < 0)
            dx = 0;
         else
            dx = gd * Lz * dt;
      }
      else
      {
         dx = (fxtot[tid] + gd * (z[tid] + 0.5 * Lz) ) *dt;
         dy = (fytot[tid]) * dt;
      }

      x[tid] += dx;
      y[tid] += dy;
      z[tid] += dz;

      dabsx[tid] += dx;
      dabsy[tid] += dy;
      dabsz[tid] += dz;

      xmag = fabs(x[tid]);
      ymag = fabs(y[tid]);

      if(xmag > (0.5 * Lx))
         x[tid] *= (1 - Lx/xmag);
      if(ymag > (0.5 * Ly))
         y[tid] *= (1 - Ly/ymag);

   }
}

//Calculates the stress in the fluid.
___global___ void tau_calc(double *z, double *fxtot, double *tau, int n,
             double Lz)
{
   int tid = threadIdx.x + blockIdx.x*blockDim.x;
   int i;

   if(tid == 0)
   {
      *tau = 0;
      for(i = 0; i < n; i++)
```

```
    {
        if((z[i]) > ((Lz / 2.0) - 0.5 - 0.01))
        {
            *tau += fxtot[i];
            //*tau += z[i]*(fxtot[i]);
        }
    }
  }
}



void nlist( double *, double *, double *, int *, int *, int *, double,
            int, double, double, double, int, thrust::device_ptr<int>);

int main(void)
{
  int     *key = NULL  , *particle = NULL, *neighbor = NULL;
  int     *mag_key = NULL, *mag_keyd = NULL, *neighbor_h = NULL;
  int     *particle_h = NULL, n, kstart, nsteps,  nprint,  lsteps;
  int      i, k, index;
  double *xd = NULL, *yd = NULL, *zd = NULL;
  double *x  = NULL, *y = NULL, *z = NULL;
  double *fx = NULL, *fy = NULL, *fz = NULL;
  double *fxtot = NULL, *fytot = NULL, *fztot = NULL;
  double *fxpar_h = NULL, *fypar_h = NULL, *fzpar_h = NULL;
  double  Lx, Ly, Lz,  dt, rc , rch,  corrfac;
  double  gd, rl, sphere, rc2, rl2;
  double *tau_h = NULL, *tau_d = NULL;
  double  ctime_tot = 0 , ctime_avg, timetot;
  double  omega, gamma0, time, gamma, g, dxwall;
  int     timei;
  int     cuda_count = 0;
  int     al, file_select ;
  float   elapsedTime;
  double *dabsx_h = NULL, *dabsy_h = NULL,
         *dabsz_h = NULL;
  double *dabsx_d = NULL, *dabsy_d = NULL,
         *dabsz_d = NULL;
  FILE   *pos_input, *par_input, *pos_output,
          *time_out, *tau_out, *dabs_out;
  FILE   *force_out;
  char    parameters[] = "parameters.txt";
  char    tt [80], p_out[100], f_out[100], p_in[100],
          d_out[100], t_out[100];
```

```
clock_t t0, t1;

file_select  = 0;


sprintf(p_out,"position_out%d.txt",file_select);
pos_output = fopen(p_out, "w");

sprintf(p_in,"position%05d_mono.txt",file_select);
pos_input = fopen(p_in, "r");

sprintf(d_out,"dabs_out%d.txt",file_select);
dabs_out = fopen(d_out, "w");

sprintf(f_out,"force_pair_out%d.txt",file_select);
force_out = fopen(f_out, "w");

sprintf(t_out,"tau_out%d.txt",file_select);
tau_out = fopen(t_out, "w");



par_input = fopen(parameters, "r");
fscanf(par_input,"%lf",&Lx);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&Ly);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&Lz);fgets(tt,80,par_input);
fscanf(par_input,"%d",&n);fgets(tt,80,par_input);
fscanf(par_input,"%d",&kstart);fgets(tt,80,par_input);
fscanf(par_input,"%d",&nsteps);fgets(tt,80,par_input);
fscanf(par_input,"%d",&nprint);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&dt);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&rc);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&rch);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&corrfac);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&gd);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&omega);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&gamma0);fgets(tt,80,par_input);
fscanf(par_input,"%lf",&rl); fgets(tt,80,par_input);
fscanf(par_input,"%d",&lsteps);fgets(tt,80,par_input);
fscanf(par_input,"%d",&al);fgets(tt,80,par_input);
fscanf(par_input,"%d",&timei);fgets(tt,80,par_input);
fclose(par_input);

rc2 = rc*rc;
```

```
rl2 = rl*rl;



x           = (double *)malloc(n *       sizeof(double));
y           = (double *)malloc(n *       sizeof(double));
z           = (double *)malloc(n *       sizeof(double));

dabsx_h     = (double *)malloc(n *       sizeof(double));
dabsy_h     = (double *)malloc(n *       sizeof(double));
dabsz_h     = (double *)malloc(n *       sizeof(double));

fxpar_h     = (double *)malloc(al * n  * sizeof(double));
fypar_h     = (double *)malloc(al * n  * sizeof(double));
fzpar_h     = (double *)malloc(al * n  * sizeof(double));

neighbor_h = (int    *)malloc(al * n  * sizeof(int));
particle_h = (int    *)malloc(al * n  * sizeof(int));

tau_h       = (double *)malloc(sizeof(double));
cudaMalloc((void**)&tau_d, sizeof(double));
*tau_h = 0;

mag_key = (int*)malloc(n * sizeof(int));

cudaMalloc((void**)&xd, n * sizeof(double));
cudaMalloc((void**)&yd, n * sizeof(double));
cudaMalloc((void**)&zd, n * sizeof(double));

cudaMalloc((void**)&dabsx_d, n * sizeof(double));
cudaMalloc((void**)&dabsy_d, n * sizeof(double));
cudaMalloc((void**)&dabsz_d, n * sizeof(double));
/*thrust::device_vector<double> xdt(n);
thrust::device_vector<double> ydt(n);
thrust::device_vector<double> zdt(n);

double *xd = thrust::raw_pointer_cast(&xdt[0]);
double *yd = thrust::raw_pointer_cast(&ydt[0]);
double *zd = thrust::raw_pointer_cast(&zdt[0]);

thrust::device_vector<double> fxtott(n);
thrust::device_vector<double> fytott(n);
thrust::device_vector<double> fztott(n);

double *fxtot = thrust::raw_pointer_cast(&fxtott[0]);
double *fytot = thrust::raw_pointer_cast(&fytott[0]);
double *fztot = thrust::raw_pointer_cast(&fztott[0]);*/

cudaMalloc((void**)&fxtot  , al * n * sizeof(double));
cudaMalloc((void**)&fytot  , al * n * sizeof(double));
cudaMalloc((void**)&fztot  , al * n * sizeof(double));

/*thrust::device_vector<double> fxt(m*n);
thrust::device_vector<double> fyt(m*n);
thrust::device_vector<double> fzt(m*n);

double *fx = thrust::raw_pointer_cast(&fxt[0]);
double *fy = thrust::raw_pointer_cast(&fyt[0]);
double *fz = thrust::raw_pointer_cast(&fzt[0]);*/

cudaMalloc((void**)&fx     , al * n * sizeof(double));
cudaMalloc((void**)&fy     , al * n * sizeof(double));
cudaMalloc((void**)&fz     , al * n * sizeof(double));

//thrust::device

cudaMalloc((void**)&key    , n  *     sizeof(int));
cudaMalloc((void**)&particle, al * n * sizeof(int));
cudaMalloc((void**)&neighbor, al * n * sizeof(int));
cudaMalloc((void**)&mag_keyd, n *    sizeof(int));

//thrust::device_ptr<int> key_thrust(key);

//pos_input = fopen(position, "r");
fgets(tt,80,pos_input);
   for( i = 0; i < n; i++)
   {
      fscanf(pos_input, "%lf", &sphere);
      fscanf(pos_input, "%lf", &(x[i]));
      fscanf(pos_input, "%lf", &(y[i]));
      fscanf(pos_input, "%lf", &(z[i]));
      fscanf(pos_input, "%d", &(mag_key[i]));
      dabsx_h[i] = 0;
      dabsy_h[i] = 0;
      dabsz_h[i] = 0;
      fxpar_h[i] = 0;
      fypar_h[i] = 0;
      fzpar_h[i] = 0;
```

```
    }
 fclose(pos_input);

 cudaMemcpy(xd, x, n*sizeof(double), cudaMemcpyHostToDevice);
 cudaMemcpy(yd, y, n*sizeof(double), cudaMemcpyHostToDevice);
 cudaMemcpy(zd, z, n*sizeof(double), cudaMemcpyHostToDevice);
 cudaMemcpy(mag_keyd, mag_key, n*sizeof(int),
                cudaMemcpyHostToDevice);

 cudaMemcpy(dabsx_d, dabsx_h, n*sizeof(double),
                cudaMemcpyHostToDevice);
 cudaMemcpy(dabsy_d, dabsy_h, n*sizeof(double),
                cudaMemcpyHostToDevice);
 cudaMemcpy(dabsz_d, dabsy_h, n*sizeof(double),
                cudaMemcpyHostToDevice);


 thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);

 nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz,
         al, key_thrust);

 cudaEvent_t startEvent, stopEvent;
 cudaEventCreate(&startEvent);
 cudaEventCreate(&stopEvent);



 //thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
 /*init_force<<<(BLOCK_SIZE + al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
  (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
  cudaThreadSynchronize();
 force_calc <<< (BLOCK_SIZE + al*n)/BLOCK_SIZE , BLOCK_SIZE >>>
  (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor, particle,
  mag_keyd, rc, fxtot, fytot, fztot, al);
 cudaThreadSynchronize();
 force_total<<< (BLOCK_SIZE + al*n)/BLOCK_SIZE , BLOCK_SIZE >>>
  (zd, fx, fy, fz, fxtot, fytot, fztot, particle, mag_keyd, Lz,
  rc, n, neighbor, al);
 cudaThreadSynchronize();
 force_wall<<< (BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE >>>
  (zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);
 cudaThreadSynchronize();*/

 //if(timei == 0)
 //{
 // time = dt*timei;
 // gamma = gd * time;
 // tau_calc<<<(BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE>>>
  //                (zd, fxtot, tau_d, n, Lz);
 // cudaMemcpy(tau_h, tau_d, sizeof(double), cudaMemcpyDeviceToHost);

 // *tau_h = (-*tau_h);//(Lx*Ly*(Lz-1.0));
    //g        = *tau_h;
 // g = *tau_h;//gamma0;
 //  fprintf(tau_out,"%lf  %16.12lf    %16.12lf\n", time, gamma, g);
 //}



 t0 = clock();

 for(k = kstart; k < (nsteps + 1); k++)
 {
    time = dt * (double)k;
    //timeold = dt*(double)(k-1);
    //gamma = gamma0*sin(omega*time);
    //gammaold = gamma0*sin(omega*timeold);
    //dgamma = gamma - gammaold;
    //gd = dgamma/dt;
    //dxwall = dgamma*Lz;

    if((k%lsteps) == 0)
        nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx,
            Ly, Lz, al, key_thrust);

    init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
      (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d, particle,
      neighbor);
    cudaThreadSynchronize();


    dxwall = gd * Lz * time;

    //if((k%lsteps) == 0)
    // nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx,
      //   Ly, Lz, al, key_thrust);

    cudaThreadSynchronize();
```

```
cudaEventRecord(startEvent, 0);

force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
  (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor,
   particle, mag_keyd, rc, fxtot, fytot, fztot, al);

cudaThreadSynchronize();

force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
  (zd, fx, fy, fz, fxtot, fytot, fztot, particle, mag_keyd,
  Lz, rc, n, neighbor, al);

cudaThreadSynchronize();

force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
  (zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);

cudaThreadSynchronize();

update_pos<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
  (xd, yd, zd, fxtot, fytot, fztot, Lx, Ly, Lz, n, gd,
  dt, dabsx_d, dabsy_d, dabsz_d);

cudaThreadSynchronize();

//init_force<<<(BLOCK_SIZE + al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
  //(fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
//force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
  //(xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor,
  //particle, mag_keyd, rc, fxtot, fytot, fztot, al);
//force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
  //(zd, fx, fy, fz, fxtot, fytot, fztot, particle, mag_keyd,
  //Lz, rc, n, neighbor, al);
  //force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
  //(zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);



cudaEventRecord(stopEvent,0);
cudaEventSynchronize(stopEvent);

cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);

ctime_tot += elapsedTime;
cuda_count++;
```

```
if(k % nprint == 0)
{
    printf ("%d\n", k);
    fprintf (pos_output, "         %d \\\\ \n \\\\ \n", k);
    fprintf (dabs_out , "\\\\ \n %d \\\\ \n \\\\ \n", k);
    fprintf (force_out , "\\\\ \n %d \\\\ \n \\\\ \n", k);

//tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
      //(zd, fxtot, tau_d, n, Lz);
   //cudaThreadSynchronize();


    cudaMemcpy(tau_h, tau_d, sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(x, xd, n *sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(y, yd, n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(z, zd, n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(fxpar_h, fx, al * n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(fypar_h, fy, al * n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(fzpar_h, fz, al * n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(dabsx_h, dabsx_d, n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsy_h, dabsy_d, n * sizeof(double),
       cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsz_h, dabsz_d, n * sizeof(double),
       cudaMemcpyDeviceToHost);
```

```
cudaThreadSynchronize();

cudaMemcpy(particle_h, particle, al * n * sizeof(int),
    cudaMemcpyDeviceToHost);
cudaThreadSynchronize();

cudaMemcpy(neighbor_h, neighbor, al * n * sizeof(int),
    cudaMemcpyDeviceToHost);
cudaThreadSynchronize();

//gtot = k*gd*dt;
*tau_h = 0;

tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
    (zd, fxtot, tau_d, n, Lz);
cudaMemcpy(tau_h,tau_d,sizeof(double),cudaMemcpyDeviceToHost);

*tau_h = (-*tau_h);// ((Lx*Ly)*(Lz-1.0));
//printf("%lf\n", *tau_h);
//g       = *tau_h;
g       = *tau_h;//gamma0;
gamma = gd * k * dt;

fprintf(tau_out,"% lf & % lf & % lf \\\\ \n",time,gamma,g);

for(index = 0; index < al * n; index++)
{
    if(index < n)
    {
        fprintf(pos_output,"% d &", index);
        fprintf(pos_output,"% lf &", x[index]);
        fprintf(pos_output,"% lf &", y[index]);
        fprintf(pos_output,"% lf &", z[index]);
        fprintf(pos_output,"% d \\\\ \n", mag_key[index]);

        fprintf(dabs_out, "% 17.16lf & ", dabsx_h[index]);
        fprintf(dabs_out, "% 17.16lf & ", dabsy_h[index]);
        fprintf(dabs_out, "% 17.16lf \\\\ \n", dabsz_h[index]);
    }

    fprintf(force_out,"% d & ",particle_h[index]);
    fprintf(force_out,"% d & ", neighbor_h[index]);
    fprintf(force_out,"% 17.16lf & ", fxpar_h[index]);
    fprintf(force_out,"% 17.16lf & ", fypar_h[index]);
    fprintf(force_out,"% 17.16lf \\\\ \n", fzpar_h[index]);

        if(particle_h[index] == -1)
            break;


    }//End of index loop for printing
  }//End of print if statement
}

t1 = clock();
timetot = ((double)(t1-t0)/(double)CLOCKS_PER_SEC)/60;

time_out = fopen("time.txt", "w");


ctime_avg = ctime_tot/((double)cuda_count);

fprintf(time_out, "Total simulation time = %lf minutes \n", timetot);
fprintf(time_out, "The time needed to perform the calculations is %lf
 ms\n", ctime_avg);


thrust::device_free(key_thrust);

fclose    (time_out);
fclose    (pos_output);
fclose    (dabs_out);
fclose    (force_out);
cudaFree  (dabsx_d);
cudaFree  (dabsy_d);
cudaFree  (dabsz_d);
free      (dabsx_h);
free      (dabsy_h);
free      (dabsz_h);
fclose    (tau_out);
cudaFree  (xd);
cudaFree  (yd);
cudaFree  (zd);
cudaFree  (mag_keyd);
cudaFree  (neighbor);
cudaFree  (particle);
cudaFree  (key);
cudaFree  (fx);
cudaFree  (fy);
cudaFree  (fz);
```

```
    cudaFree (fxtot);
    cudaFree (fytot);
    cudaFree (fztot);
    free      (x);
    free      (y);
    free      (z);
    free      (mag_key);

}


void  nlist (double *xd, double *yd, double *zd, int  *neighbor, int  *particle,
   int  *key, double rl2, int  n, double Lx, double Ly, double Lz, int  al,
   thrust::device_ptr<int> key_thrust)
{
    keyzero<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
     (key, neighbor, particle, n, al);

    cudaThreadSynchronize();

    check<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
     (xd, yd, zd, key, neighbor, particle, rl2, n, Lx, Ly, Lz);

    cudaThreadSynchronize();

    //thrust::device_ptr<int> key_thrust(key);
    //thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
    key_thrust = thrust::device_pointer_cast(key);

    thrust::inclusive_scan(key_thrust, key_thrust + n, key_thrust);
    //thrust::device_free(key_thrust);

    cudaThreadSynchronize();

    setup<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
     (xd, yd, zd, key, particle, n);

    cudaThreadSynchronize();

    populate<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
     (xd, yd, zd, neighbor, particle, n, rl2, Lx, Ly, Lz, al);

    cudaThreadSynchronize();

// thrust::device_free(key_thrust);

}
```

A sample parameter input file for the code *mix_strain.cu* would look like:

```
30.0     Lx
10.0     Ly
10.0     Lz
20       Number of spheres
1        Start
100      Number of steps
1        Print Statements
0.0001   dt
2.5      Cutoff radius
0.125    Hydrodynamic Cutoff Radius
1E-4     h correction factor
0.001    Dimensionless shear
.01      Frequency, omega
0.00001  Strain amplitude, gamma0
2.7      Cutoff radius for neighbor list
100      Number of steps to calculate neighbor list
70       Neighbor list length
0        Initial time
```

A sample position input file for the code *mix_strain.cu* would look like:

```
 Sphere ID   X   Y    Z    Mag ID
     0      -1   0  -4.5     1
     1      -1   0  -3.5     1
     2      -1   0  -2.5     1
     3      -1   0  -1.5     1
     4      -1   0  -0.5     1
     5      -1   0   0.5     1
     6      -1   0   1.5     1
     7      -1   0   2.5     1
     8      -1   0   3.5     1
     9      -1   0   4.5     1
    10       1   0  -4.5     1
    11       1   0  -3.5     1
    12       1   0  -2.5     1
    13       1   0  -1.5     1
    14       1   0  -0.5     1
    15       1   0   0.5     1
    16       1   0   1.5     1
    17       1   0   2.5     0
    18       1   0   3.5     0
    19       1   0   4.5     0
```

This is an example of the output position file *position_out_relaxed0.txt* for the preceding input files.

```
500000

  0     -1.000000  0.000000  -4.497543  1
  1     -0.992883  0.000000  -3.495495  1
  2     -0.970887  0.000000  -2.493811  1
  3     -0.935373  0.000000  -1.492574  1
  4     -0.888962  0.000000  -0.492092  1
  5     -0.829722  0.000000   0.507747  1
  6     -0.757412  0.000000   1.506801  1
  7     -0.672542  0.000000   2.504633  1
  8     -0.587958  0.000000   3.502146  1
  9     -0.500000  0.000000   4.499376  1
 10      1.000000  0.000000  -4.509048  1
 11      1.139411  0.000000  -3.527136  1
 12      1.244195  0.000000  -2.541088  1
```

```
  13     1.327551   0.000000  -1.553091  1
  14     1.389497   0.000000  -0.563581  1
  15     1.433681   0.000000   0.426954  1
  16     1.456652   0.000000   1.418793  1
  17     1.090460   0.000000   2.410850  0
  18     1.522329   0.000000   3.380600  0
  19     1.479899   0.000000   4.440285  0
1000000

   0    -1.000000   0.000000  -4.493517  1
   1    -0.940879   0.000000  -3.489616  1
   2    -0.868774   0.000000  -2.486688  1
   3    -0.782711   0.000000  -1.484827  1
   4    -0.682862   0.000000  -0.484125  1
   5    -0.568770   0.000000   0.515271  1
   6    -0.439356   0.000000   1.513143  1
   7    -0.292544   0.000000   2.508237  1
   8    -0.148993   0.000000   3.502886  1
   9     0.000000   0.000000   4.496770  1
  10     1.000000   0.000000  -4.509141  1
  11     1.160114   0.000000  -3.530322  1
  12     1.283060   0.000000  -2.546322  1
  13     1.383013   0.000000  -1.559821  1
  14     1.459514   0.000000  -0.571287  1
  15     1.512157   0.000000   0.418778  1
  16     1.539779   0.000000   1.410461  1
  17     1.396964   0.000000   2.465428  0
  18     1.977491   0.000000   3.364490  0
  19     1.948857   0.000000   4.432653  0
1500000

   0    -1.000000   0.000000  -4.486657  1
   1    -0.889789   0.000000  -3.481195  1
   2    -0.767936   0.000000  -2.477148  1
   3    -0.631116   0.000000  -1.474658  1
   4    -0.478754   0.000000  -0.473752  1
   5    -0.309211   0.000000   0.525540  1
   6    -0.119696   0.000000   1.523417  1
   7     0.099896   0.000000   2.514415  1
   8     0.302386   0.000000   3.504194  1
```

```
   9      0.500000   0.000000   4.494659   1
  10      1.000000   0.000000  -4.509365   1
  11      1.171383   0.000000  -3.532592   1
  12      1.302616   0.000000  -2.549802   1
  13      1.407907   0.000000  -1.563989   1
  14      1.485550   0.000000  -0.575663   1
  15      1.534112   0.000000   0.414524   1
  16      1.550244   0.000000   1.406435   1
  17      1.755203   0.000000   2.481698   0
  18      2.409698   0.000000   3.348657   0
  19      2.419851   0.000000   4.425657   0
2000000

   0     -1.000000   0.000000  -4.508712   1
   1     -1.100209   0.000000  -3.521924   1
   2     -1.170745   0.000000  -2.532687   1
   3     -1.222197   0.000000  -1.542342   1
   4     -1.257551   0.000000  -0.551422   1
   5     -1.280571   0.000000   0.439853   1
   6     -1.291939   0.000000   1.431924   1
   7      1.048538   0.000000   2.500505   1
   8      1.024719   0.000000   3.500262   1
   9      1.000000   0.000000   4.499998   1
  10      1.000000   0.000000  -4.499319   1
  11      1.045932   0.000000  -3.499687   1
  12      1.074398   0.000000  -2.499593   1
  13      1.089145   0.000000  -1.499314   1
  14      1.092996   0.000000  -0.499232   1
  15      1.087440   0.000000   0.500816   1
  16      1.072754   0.000000   1.500757   1
  17      2.172676   0.000000   2.471378   0
  18      2.833465   0.000000   3.340518   0
  19      2.891533   0.000000   4.420849   0
2500000

   0     -1.000000   0.000000  -4.508655   1
   1     -1.072567   0.000000  -3.519521   1
   2     -1.117634   0.000000  -2.528848   1
   3     -1.146198   0.000000  -1.537705   1
   4     -1.162148   0.000000  -0.546288   1
```

```
   5     -1.166677  0.000000   0.445249  1
   6     -1.165101  0.000000   1.437393  1
   7      1.415889  0.000000   2.498351  1
   8      1.457150  0.000000   3.498642  1
   9      1.500000  0.000000   4.498872  1
  10      1.000000  0.000000  -4.498289  1
  11      1.093327  0.000000  -3.500507  1
  12      1.167894  0.000000  -2.501450  1
  13      1.231605  0.000000  -1.502101  1
  14      1.286302  0.000000  -0.502322  1
  15      1.332771  0.000000   0.497789  1
  16      1.374399  0.000000   1.498068  1
  17      2.545140  0.000000   2.469792  0
  18      3.250777  0.000000   3.332446  0
  19      3.363183  0.000000   4.416009  0
3000000

   0     -1.000000  0.000000  -4.508510  1
   1     -1.042461  0.000000  -3.517577  1
   2     -1.062509  0.000000  -2.526110  1
   3     -1.067028  0.000000  -1.534573  1
   4     -1.060226  0.000000  -0.543044  1
   5     -1.055822  0.000000   0.448498  1
   6     -1.053368  0.000000   1.440641  1
   7      1.788251  0.000000   2.496687  1
   8      1.892190  0.000000   3.496307  1
   9      2.000000  0.000000   4.495551  1
  10      1.000000  0.000000  -4.495344  1
  11      1.140169  0.000000  -3.499058  1
  12      1.260850  0.000000  -2.500803  1
  13      1.372497  0.000000  -1.501791  1
  14      1.476268  0.000000  -0.502153  1
  15      1.580264  0.000000   0.497460  1
  16      1.684242  0.000000   1.497075  1
  17      2.918587  0.000000   2.469632  0
  18      3.666817  0.000000   3.327941  0
  19      3.834459  0.000000   4.411613  0
3500000

   0     -1.000000  0.000000  -4.508497  1
```

```
   1     -1.026226  0.000000  -3.517023  1
   2     -1.032180  0.000000  -2.525378  1
   3     -1.023262  0.000000  -1.533865  1
   4     -1.016711  0.000000  -0.542334  1
   5     -1.011878  0.000000   0.449206  1
   6     -1.009233  0.000000   1.441349  1
   7      2.170525  0.000000   2.495312  1
   8      2.332443  0.000000   3.493849  1
   9      2.500000  0.000000   4.491575  1
  10      1.000000  0.000000  -4.491320  1
  11      1.190740  0.000000  -3.495295  1
  12      1.360790  0.000000  -2.497307  1
  13      1.522579  0.000000  -1.498753  1
  14      1.684567  0.000000  -0.500239  1
  15      1.846529  0.000000   0.498279  1
  16      2.008504  0.000000   1.496798  1
  17      3.292225  0.000000   2.469616  0
  18      4.082778  0.000000   3.325770  0
  19      4.305328  0.000000   4.407796  0
4000000

   0     -1.000000  0.000000  -4.508473  1
   1     -0.996346  0.000000  -3.516779  1
   2     -0.990177  0.000000  -2.525246  1
   3     -0.984211  0.000000  -1.533712  1
   4     -0.978909  0.000000  -0.542174  1
   5     -0.974854  0.000000   0.449370  1
   6     -0.972567  0.000000   1.441513  1
   7      2.612427  0.000000   2.552299  1
   8      2.801243  0.000000   3.530052  1
   9      3.000000  0.000000   4.505877  1
  10      1.000000  0.000000  -4.505153  1
  11      1.495689  0.000000  -3.317376  1
  12      1.678448  0.000000  -2.338175  1
  13      1.863845  0.000000  -1.359847  1
  14      2.049844  0.000000  -0.381616  1
  15      2.236633  0.000000   0.596482  1
  16      2.424192  0.000000   1.574448  1
  17      3.675654  0.000000   2.468872  0
  18      4.498838  0.000000   3.324875  0
```

```
  19     4.775836   0.000000   4.404494   0
4500000

   0    -1.000000   0.000000  -4.508465   1
   1    -0.989495   0.000000  -3.516812   1
   2    -0.980004   0.000000  -2.525304   1
   3    -0.971543   0.000000  -1.533786   1
   4    -0.964559   0.000000  -0.542258   1
   5    -0.959473   0.000000   0.449281   1
   6    -0.956696   0.000000   1.441424   1
   7     3.292838   0.000000   2.536717   1
   8     3.391548   0.000000   3.523372   1
   9     3.500000   0.000000   4.509023   1
  10     1.000000   0.000000  -4.507717   1
  11     2.950799   0.000000  -3.401270   1
  12     2.972278   0.000000  -2.409519   1
  13     3.010193   0.000000  -1.418832   1
  14     3.063295   0.000000  -0.428818   1
  15     3.129667   0.000000   0.560426   1
  16     3.207149   0.000000   1.548895   1
  17     4.330426   0.000000   2.440222   0
  18     4.916969   0.000000   3.327439   0
  19     5.246073   0.000000   4.401703   0
5000000

   0    -1.000000   0.000000  -4.508466   1
   1    -0.989495   0.000000  -3.516812   1
   2    -0.979998   0.000000  -2.525304   1
   3    -0.971533   0.000000  -1.533787   1
   4    -0.964545   0.000000  -0.542259   1
   5    -0.959457   0.000000   0.449281   1
   6    -0.956679   0.000000   1.441423   1
   7     3.959333   0.000000   2.526112   1
   8     3.978690   0.000000   3.517481   1
   9     4.000000   0.000000   4.508802   1
  10     1.000000   0.000000  -4.507541   1
  11     3.892375   0.000000  -3.423377   1
  12     3.896336   0.000000  -2.431239   1
  13     3.903772   0.000000  -1.439713   1
  14     3.914256   0.000000  -0.448215   1
```

```
15      3.927332   0.000000   0.543255   1
16      3.942545   0.000000   1.534695   1
17      5.002475   0.000000   2.350244   0
18      5.357748   0.000000   3.371630   0
19      5.717441   0.000000   4.402841
```

The neighbor list function at the end of the code is based off the collision detection developed by Mazhar et al. (2011)

Appendix C

Mix_Relax.cu

This appendix contains the code *Mix_Relax.cu*. This code is used to relax the positions saved from *Mix_Strain.cu*. Every `n_print` configuration is saved.

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#include <thrust/scan.h>
#include <thrust/sort.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_free.h>
#include <thrust/device_malloc.h>

#define BLOCK_SIZE 512

__global__ void keyzero(int *key, int *neighbor, int *particle, int n, int al)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;

  if(tid < n)
  {
    key[tid] = 0;
  }

  if(tid < al*n)
  {
    neighbor[tid] = -1;
    particle[tid] = -1;
  }

}

__global__ void check(double *x, double *y, double *z, int *key,
                      int *neighbor, int *particle, double rl2, int n,
                      double Lx, double Ly, double Lz)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  double xij, yij, zij, r2, xmag, ymag;//, zmag;
  int i;



  if(tid < n)
  {
    for(i = 0; i < n; i++)
    {
      if(tid != i)
      {
        xij = x[i] - x[tid];
        yij = y[i] - y[tid];
        zij = z[i] - z[tid];

        xmag = fabs(xij);
        ymag = fabs(yij);
        //zmag = fabs(zij);

        if(xmag > .5*Lx)
          xij = xij*(1 - Lx/xmag);
        if(ymag > .5*Ly)
          yij = yij*(1 - Ly/ymag);

        r2 = xij*xij + yij*yij + zij*zij;

        if(r2 < rl2)
        {
          atomicAdd( &(key[i]), 1);
        }
      }
    }
  }
}

__global__ void setup(double *x, double *y, double *z, int *key,
                      int *particle, int n)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  int i, beg, end;

  if(tid < n)
  {
    end = key[tid];

    if(tid == 0)
      beg = 0;
    else
      beg = key[tid - 1];

    for(i = beg; i < end; i++)
      particle[i] = tid;
  }
```

```
}

__global__ void populate(double *x, double *y, double *z, int *neighbor,
                         int *particle, int n, double rl2, double Lx, double Ly,
                         double Lz, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int start, tpart, jspher, tcount;
   double xij, yij, zij, r2, xmag, ymag;

   if(tid < al*n)
   {

      if(particle[tid] != -1)
      {
         tcount = 0;

         if((particle[tid] != particle[tid - 1] | tid == 0))
         {
            start = particle[tid];
            tpart = start;

            for(jspher = 0; jspher < n; jspher++)
            {

               xij = x[jspher] - x[tpart];
               yij = y[jspher] - y[tpart];
               zij = z[jspher] - z[tpart];

               xmag = fabs(xij);
               ymag = fabs(yij);
               //zmag = fabs(zij);

               if(xmag > .5*Lx)
                  xij = xij*(1 - Lx/xmag);
               if(ymag > .5*Ly)
                  yij = yij*(1 - Ly/ymag);

               r2 = xij*xij + yij*yij + zij*zij;

               if(r2 < rl2)
               {
                  if(tpart != jspher)
                  {
                     neighbor[tid + tcount] = jspher;
                     tcount++;
                     tpart = particle[tid + tcount];
                     if(tpart != start)
                        break;
                  }
               }
            }
         }
      }
   }
}

__global__ void init_force(double *fx, double *fy, double *fz, int n,
                           double *fxtot, double *fytot, double *fztot, int al,
                           double *tau, int *particle, int *neighbor)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;

   if(tid<n)
   {
      fxtot[tid] = 0;
      fytot[tid] = 0;
      fztot[tid] = 0;

      tau[0] = 0;
   }

   if(tid < al*n)
   {
      fx[tid] = 0;
      fy[tid] = 0;
      fz[tid] = 0;
   }
}

__global__ void force_calc(double *x, double *y, double *z,
                           double *fx, double *fy, double *fz, int n,
                           double rc2, double Lx, double Ly, double Lz,
                           int *neighbor, int *particle, int *mag_key,
                           double rc, double *fxtot, double *fytot,
                           double *fztot, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, j;
   double xij, yij, zij, xmag, ymag, zmag, r2, r, zpp, ri2, ri, ri4;
```

```
double wh, iterm, jterm, ijterm;
double C, C2, r4, rep;
double fxi, fyi, fzi;


if(tid < al*n)
{
    if(particle[tid] != -1)
    {
        i = particle[tid];
        j = neighbor[tid];

        iterm = (double)mag_key[i];
        jterm = (double)mag_key[j];
        ijterm = iterm*jterm;

        zmag = fabs(z[i]);
        wh = (Lz/2.00) - zmag;

        xij = x[j] - x[i];
        yij = y[j] - y[i];
        zij = z[j] - z[i];

        xmag = fabs(xij);
        ymag = fabs(yij);

        if(xmag > (0.5 * Lx))
            xij = xij * (1 - Lx / xmag);
        if(ymag > (0.5 * Ly))
            yij = yij * (1 - Ly / ymag);

        r2 = xij*xij + yij*yij + zij*zij;


        if(r2 < rc*rc)
        {
            r = sqrt(r2);
            C = zij/r;
            C2 = 5.0*C*C;
            r4 = r*r*r*r;
            //r4 = r2*r2;

            rep = exp((1.0 - r)/0.01);

            fx[tid]=((ijterm*(C2-1.0)/r4)-rep)*(xij/r);
            fy[tid]=((ijterm*(C2-1.0)/r4)-rep)*(yij/r);
            fz[tid]=((ijterm*(C2-3.0)/r4)-rep)*C;

        }

        if(wh < rc)
        {
            zpp = (z[i] / zmag) * Lz - z[j] - z[i];
            ri2 = zpp * zpp + xij * xij + yij * yij;


            if(ri2 < rc*rc)
            {
                ri = sqrt(ri2);
                C = zpp/ri;
                C2 = 5.0*C*C;
                ri4 = ri*ri*ri*ri;
                //ri4 = ri2*ri2;

                fxi=(xij/ri)*ijterm*(C2-1.0)/ri4;
                fyi=(yij/ri)*ijterm*(C2 - 1.0)/ri4;
                fzi=C*ijterm*(C2-3.0)/ri4;

//              fxi = (xij/ri)*(C2 - 1.0)/ri4;
//              fyi = (yij/ri)*(C2 - 1.0)/ri4;
//              fzi = C*(C2 - 3.0)/ri4;

                fx[tid] += fxi;
                fy[tid] += fyi;
                fz[tid] += fzi;

            }
        }
    }
}


__global__ void force_total(double *z, double *fx, double *fy, double *fz,
            double *fxtot, double *fytot, double *fztot, int *particle,
            int *mag_key, double Lz, double rc, int n, int *neighbor,
            int al)
{
    int     tid = threadIdx.x + blockDim.x*blockIdx.x;
    int     i, check, redkey;
```

```
    double sumx, sumy, sumz;


    if(tid < al*n)
    {

        if(particle[tid] != -1)
        {
            check = particle[tid];

            if(tid == 0 | check != particle[tid - 1])
            {
                sumx = fx[tid];
                sumy = fy[tid];
                sumz = fz[tid];
                i = tid;

                while(check == particle[i+1])
                {
                    sumx += fx[i+1];
                    sumy += fy[i+1];
                    sumz += fz[i+1];
                    i++;
                    check = particle[i];
                }

                redkey = particle[tid];

                fxtot[redkey] = sumx;
                fytot[redkey] = sumy;
                fztot[redkey] = sumz;
            }
        }
    }
}

__global__ void force_wall(double *z, double Lz, double *fztot, int n,
                           double rc, int *mag_key, double *fxtot, double *fytot,
                           double *tau)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    double wh, wh4, iterm, zmag, term1, term2, fzw;

    if(tid < n)
    {
        iterm = (double)mag_key[tid];
        zmag = fabs(z[tid]);
        wh = (Lz/2.00) - zmag;

        if(wh < (0.5 * rc))
        {
            wh4 = wh*wh*wh*wh;
            term1 = iterm * (1.0 / wh4) / 8.0;
            term2 = exp((0.5 - wh) / 0.01);

            fzw = term1 - term2;

            fztot[tid] += fzw * (z[tid] / zmag);

        }
    }
}

__global__ void update_pos(double *x, double *y, double *z, double *fxtot,
                           double *fytot, double *fztot, double Lx, double Ly, double Lz,
                           int n, double gd, double dt, double *dabsx, double *dabsy,
                           double *dabsz)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    double xmag, ymag, zmag, dx, dy, dz;


    if(tid < n)
    {

        zmag = fabs(z[tid]);

        dz = fztot[tid] * dt;

        if(zmag >= ((0.5 * Lz) - .5 - .05))
        {
            dy = 0;
            if(z[tid] < 0)
                dx = 0;
            else
                dx = gd * Lz * dt;
        }
        else
        {
            dx = (fxtot[tid] + gd * (z[tid] + 0.5 * Lz) ) *dt;
```

```
        dy = (fytot[tid]) * dt;
      }

      x[tid] += dx;
      y[tid] += dy;
      z[tid] += dz;

      dabsx[tid] += dx;
      dabsy[tid] += dy;
      dabsz[tid] += dz;

      xmag = fabs(x[tid]);
      ymag = fabs(y[tid]);

      if(xmag > (0.5 * Lx))
        x[tid] *= (1 - Lx/xmag);
      if(ymag > (0.5 * Ly))
        y[tid] *= (1 - Ly/ymag);

  }
}

___global___ void tau_calc(double *z, double *fxtot, double *tau, int n,
                  double Lz)
{
  int tid = threadIdx.x + blockIdx.x*blockDim.x;
  int i;

  if(tid == 0)
  {
    *tau = 0;
    for(i = 0; i < n; i++)
    {
      if((z[i]) > ((Lz / 2.0) - 0.5 - 0.01))
      {
        *tau += fxtot[i];
        //*tau += z[i]*(fxtot[i]);
      }
    }
  }
}

void nlist( double *, double *, double *, int *, int *, int *, double, int,
double, double, double, int, thrust::device_ptr<int>);

int main(void)
{
  int    *key = NULL , *particle    = NULL, *neighbor = NULL;
  int    *mag_key = NULL, *mag_keyd = NULL;
  int                     *particle_h = NULL, *neighbor_h = NULL;
  int     n    , kstart          , nsteps          , nprint;
  int     lsteps          , i, k, index, n_tpt, tdx;
  double *xd     = NULL, *yd       = NULL, *zd       = NULL;
  double *x      = NULL, *y        = NULL, *z        = NULL;
  double *fx     = NULL, *fy       = NULL, *fz       = NULL;
  double *fxpar_h = NULL, *fypar_h = NULL, *fzpar_h = NULL;
  double *fxtot  = NULL, *fytot    = NULL, *fztot    = NULL;
  double  Lx, Ly, Lz, dt, rc, rch, corrfac, gd, rl, sphere, rc2, rl2;
  double *tau_h  = NULL, *tau_d = NULL;
  double  ctime_tot = 0 , ctime_avg         , timetot;
  double  omega         , gamma0            , time    , gamma, g;
  int     file_select ;
  int     timei;
  int     cuda_count = 0;
  int     al;
  float   elapsedTime;
  double *dabsx_h = NULL, *dabsy_h = NULL, *dabsz_h = NULL;
  double *dabsx_d = NULL, *dabsy_d = NULL, *dabsz_d = NULL;
  char    parameters[] = "parameters.txt";
  //char    position []    = "position_out0.xyz";
  char    tt [80], p_out[100], f_out[100], p_in[100], d_out[100], t_out[100];
  FILE  *pos_input   , *par_input, *pos_output, *time_out, *tau_out;
  FILE  *dabs_out, *force_out;
  clock_t t0, t1;

  file_select  = 12;

  sprintf(p_out, "position_out_relaxed%d.txt" , file_select);
  pos_output = fopen(p_out, "w");

  sprintf(p_in, "position_out%d.txt"  ,  file_select );
  pos_input = fopen(p_in, "r");

  sprintf(d_out, "dabs_out_relaxed%d.txt"   ,  file_select );
  dabs_out = fopen(d_out, "w");

  sprintf(t_out, "tau_out_relaxed%d.txt"     ,  file_select );
  tau_out = fopen(t_out, "w");

  sprintf(f_out, "force_pair_out_relaxed%d.txt", file_select);
```

```
force_out = fopen(f_out, "w");


par_input = fopen(parameters, "r");
fscanf(par_input, "%lf", &Lx);          fgets(tt,80,par_input);
fscanf(par_input, "%lf", &Ly);          fgets(tt,80,par_input);
fscanf(par_input, "%lf", &Lz);          fgets(tt,80,par_input);
fscanf(par_input, "%d" , &n);           fgets(tt,80,par_input);
fscanf(par_input, "%d" , &kstart);      fgets(tt,80,par_input);
fscanf(par_input, "%d" , &nsteps);      fgets(tt,80,par_input);
fscanf(par_input, "%d" , &nprint);      fgets(tt,80,par_input);
fscanf(par_input, "%d" , &n_tpt);       fgets(tt,80,par_input);
fscanf(par_input, "%lf", &dt);          fgets(tt,80,par_input);
fscanf(par_input, "%lf", &rc);          fgets(tt,80,par_input);
fscanf(par_input, "%lf", &rch);         fgets(tt, 80, par_input);
fscanf(par_input, "%lf", &corrfac);     fgets(tt, 80, par_input);
fscanf(par_input, "%lf", &gd);          fgets(tt,80,par_input);
fscanf(par_input, "%lf", &omega);       fgets(tt,80,par_input);
fscanf(par_input, "%lf", &gamma0);      fgets(tt,80,par_input);
fscanf(par_input, "%lf", &rl);          fgets(tt,80,par_input);
fscanf(par_input, "%d" , &lsteps);      fgets(tt,80,par_input);
fscanf(par_input, "%d" , &al);          fgets(tt,80,par_input);
fscanf(par_input, "%d" , &timei);       fgets(tt,80,par_input);
fclose(par_input);

rc2 = rc*rc;
rl2 = rl*rl;



x          = (double *)malloc(     n * sizeof(double));
y          = (double *)malloc(     n * sizeof(double));
z          = (double *)malloc(     n * sizeof(double));

dabsx_h    = (double *)malloc(     n * sizeof(double));
dabsy_h    = (double *)malloc(     n * sizeof(double));
dabsz_h    = (double *)malloc(     n * sizeof(double));

fxpar_h    = (double *)malloc(al * n * sizeof(double));
fypar_h    = (double *)malloc(al * n * sizeof(double));
fzpar_h    = (double *)malloc(al * n * sizeof(double));

particle_h = (int    *)malloc(al * n * sizeof(int));
neighbor_h = (int    *)malloc(al * n * sizeof(int));
```

```
tau_h = (double*)malloc(sizeof(double));
cudaMalloc((void**)&tau_d, sizeof(double));
*tau_h = 0;

//neighborh = (int*)malloc(al*n*sizeof(int));
//particleh = (int*)malloc(al*n*sizeof(int));

mag_key = (int*)malloc(n*sizeof(int));

cudaMalloc((void**)&xd, n*sizeof(double));
cudaMalloc((void**)&yd, n*sizeof(double));
cudaMalloc((void**)&zd, n*sizeof(double));

cudaMalloc((void**)&dabsx_d, n*sizeof(double));
cudaMalloc((void**)&dabsy_d, n*sizeof(double));
cudaMalloc((void**)&dabsz_d, n*sizeof(double));


cudaMalloc((void**)&fxtot, n*sizeof(double));
cudaMalloc((void**)&fytot, n*sizeof(double));
cudaMalloc((void**)&fztot, n*sizeof(double));


cudaMalloc((void**)&fx, al*n*sizeof(double));
cudaMalloc((void**)&fy, al*n*sizeof(double));
cudaMalloc((void**)&fz, al*n*sizeof(double));


cudaMalloc((void**)&key, n*sizeof(int));
cudaMalloc((void**)&particle, al*n*sizeof(int));
cudaMalloc((void**)&neighbor, al*n*sizeof(int));
cudaMalloc((void**)&mag_keyd, n*sizeof(int));

thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);

for(tdx = 0; tdx < n_tpt; tdx++)
{

    fgets(tt,80,pos_input);
      for( i = 0; i<n; i++)
      {
          fscanf(pos_input, "%lf", &sphere);
          fscanf(pos_input, "%lf", &(x[i]));
          fscanf(pos_input, "%lf", &(y[i]));
          fscanf(pos_input, "%lf", &(z[i]));
```

```
        fscanf(pos_input, "%d", &(mag_key[i]));
        dabsx_h[i] = 0;
        dabsy_h[i] = 0;
        dabsz_h[i] = 0;


    }
//fclose(pos_input);

cudaMemcpy(xd, x, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(yd, y, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(zd, z, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(mag_keyd, mag_key, n*sizeof(int),
                cudaMemcpyHostToDevice);

cudaMemcpy(dabsx_d, dabsx_h, n*sizeof(double),
                cudaMemcpyHostToDevice);
cudaMemcpy(dabsy_d, dabsy_h, n*sizeof(double),
                cudaMemcpyHostToDevice);
cudaMemcpy(dabsz_d, dabsy_h, n*sizeof(double),
                cudaMemcpyHostToDevice);

nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz,
        al, key_thrust);

cudaEvent_t startEvent, stopEvent;
cudaEventCreate(&startEvent);
cudaEventCreate(&stopEvent);



//thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
/*init_force<<<(BLOCK_SIZE + al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
  (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
cudaThreadSynchronize();
force_calc <<< (BLOCK_SIZE + al*n)/BLOCK_SIZE , BLOCK_SIZE >>>
  (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor, particle,
  mag_keyd, rc, fxtot, fytot, fztot, al);
cudaThreadSynchronize();
force_total<<< (BLOCK_SIZE + al*n)/BLOCK_SIZE , BLOCK_SIZE >>>
  (zd, fx, fy, fz, fxtot, fytot, fztot, particle, mag_keyd, Lz,
  rc, n, neighbor, al);
cudaThreadSynchronize();
force_wall<<< (BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE >>>
  (zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);
cudaThreadSynchronize();*/

//if(timei == 0)
//{
// time = dt*timei;
// gamma = gd * time;
// tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
  //              (zd, fxtot, tau_d, n, Lz);
// cudaMemcpy(tau_h,tau_d,sizeof(double),cudaMemcpyDeviceToHost);

// *tau_h = (-*tau_h);//(Lx*Ly*(Lz-1.0));
   //g       = *tau_h;
// g = *tau_h;//gamma0;
//  fprintf (tau_out,"%lf %16.12lf %16.12lf\n", time, gamma, g);
//}



t0 = clock();

for(k = kstart; k < (nsteps + 1); k++)
{
   time = dt * (double)k;
   //timeold = dt*(double)(k-1);
   //gamma = gamma0*sin(omega*time);
   //gammaold = gamma0*sin(omega*timeold);
   //dgamma = gamma - gammaold;
   //gd = dgamma/dt;
   //dxwall = dgamma*Lz;

   if((k%lsteps) == 0)
      nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz,
          al, key_thrust);

   init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
      (fx,fy,fz,n,fxtot,fytot,fztot,al,tau_d,particle,neighbor);
   cudaThreadSynchronize();


   //dxwall = gd * Lz * time;

   //if((k%lsteps) == 0)
   // nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz, al,
      //key_thrust);
```

```
cudaThreadSynchronize();

cudaEventRecord(startEvent, 0);

force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE >>>
            (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz,
            neighbor, particle, mag_keyd, rc, fxtot, fytot,
            fztot, al);

cudaThreadSynchronize();

force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
                  (zd, fx, fy, fz, fxtot, fytot, fztot,
                  particle, mag_keyd, Lz, rc, n, neighbor,
                  al);

cudaThreadSynchronize();

force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                 (zd, Lz, fztot, n, rc, mag_keyd, fxtot,
                 fytot, tau_d);

cudaThreadSynchronize();

update_pos<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                 (xd, yd, zd, fxtot, fytot, fztot, Lx,
                 Ly, Lz, n, gd, dt, dabsx_d, dabsy_d,
               dabsz_d);

cudaThreadSynchronize();

/*
init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
   (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE >>>
   (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor,
    particle, mag_keyd, rc, fxtot, fytot, fztot, al);
force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE >>>
   (zd, fx, fy, fz, fxtot, fytot, fztot, particle, mag_keyd,
   Lz, rc, n, neighbor, al);
force_wall<<< (BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE >>>
   (zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);
*/
```

```
cudaEventRecord(stopEvent,0);
cudaEventSynchronize(stopEvent);

cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);

ctime_tot += elapsedTime;
cuda_count++;


if(k % nprint == 0)
{


    printf("%d    %d\n", k, tdx);
    fprintf(pos_output, " %d\n\n", k);
    fprintf(dabs_out , "\n%d \n\n", k);
    fprintf(force_out , "\n %d \n\n", k);

    //tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
        //(zd, fxtot, tau_d, n, Lz);
    //cudaThreadSynchronize();


    cudaMemcpy(tau_h,tau_d,sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(x,xd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(y,yd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(z,zd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(dabsx_h,dabsx_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsy_h,dabsy_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsz_h,dabsz_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(fxpar_h,fx,al*n*sizeof(double),
                    cudaMemcpyDeviceToHost);
```

```
cudaThreadSynchronize();

    cudaMemcpy(fypar_h,fy,al*n*sizeof(double),
               cudaMemcpyDeviceToHost);
cudaThreadSynchronize();
cudaMemcpy(fzpar_h,fz,al*n*sizeof(double),
               cudaMemcpyDeviceToHost);
cudaThreadSynchronize();

cudaMemcpy(particle_h,particle,al*n*sizeof(int),
               cudaMemcpyDeviceToHost);
cudaThreadSynchronize();

cudaMemcpy(neighbor_h, neighbor,al*n*sizeof(int),
               cudaMemcpyDeviceToHost);
cudaThreadSynchronize();

//gtot = k*gd*dt;
*tau_h = 0;

tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
    (zd, fxtot, tau_d, n, Lz);
cudaMemcpy(tau_h, tau_d, sizeof(double),
               cudaMemcpyDeviceToHost);

*tau_h = (-*tau_h);// ((Lx*Ly)*(Lz-1.0));
g      = *tau_h;//gamma0;
gamma = gd * k * dt;

fprintf(tau_out,"% lf          % 16.12lf          % 16.12lf\n", time, gamma, g);

for(index = 0; index < (al*n); index++)
{
   if(index < n)
   {
      fprintf(pos_output, "% d", index);
      fprintf(pos_output, "% lf", x[index]);
      fprintf(pos_output, "% lf", y[index]);
      fprintf(pos_output, "% lf", z[index]);
      fprintf(pos_output, "% d \n", mag_key[index]);

      fprintf(dabs_out, "% 17.16lf ", dabsx_h[index]);
      fprintf(dabs_out, "% 17.16lf ", dabsy_h[index]);
      fprintf(dabs_out, "% 17.16lf \n", dabsz_h[index]);
   }
```

```
      fprintf(force_out , "% d"     , particle_h[index]);
      fprintf(force_out , "% d"     , neighbor_h[index]);
      fprintf(force_out , "% 17.16lf", fxpar_h[index]);
      fprintf(force_out , "% 17.16lf", fypar_h[index]);
      fprintf(force_out , "% 17.16lf\n", fzpar_h[index]);

      if(particle_h[index] == -1)
        break;


    } //End of index for printing configurations
  } //End of print statement if statement
 }
 fgets(tt,80,pos_input);
 fgets(tt,80,pos_input);
}

t1 = clock();
timetot = ((double)(t1-t0)/(double)CLOCKS_PER_SEC)/60;
time_out = fopen("time.txt", "w");


ctime_avg = ctime_tot/((double)cuda_count);

fprintf(time_out, "Total simulation time = %lf minutes \n", timetot);
fprintf(time_out, "The time needed to perform the calculations is %lf
      ms\n", ctime_avg);
fclose(time_out);

// thrust::device_free(key_thrust);

fclose (pos_output);
fclose (pos_input);
fclose (dabs_out);
fclose (force_out);
fclose (tau_out);

cudaFree(dabsx_d);
cudaFree(dabsy_d);
cudaFree(dabsz_d);
cudaFree(xd);
cudaFree(yd);
```

```
    cudaFree(zd          );
    cudaFree(mag_keyd );
    cudaFree(neighbor   );
    cudaFree(particle    );
    cudaFree(key         );
    cudaFree(fx          );
    cudaFree(fy          );
    cudaFree(fz          );
    cudaFree(fxtot       );
    cudaFree(fytot       );
    cudaFree(fztot       );

    free     (dabsx_h    );
    free     (dabsy_h    );
    free     (dabsz_h    );
    free     (fxpar_h    );
    free     (fypar_h    );
    free     (fzpar_h    );
    free     (x          );
    free     (y          );
    free     (z          );
    free     (mag_key    );
    free     (neighbor_h );
    free     (particle_h );

}


void  nlist (double *xd, double *yd, double *zd, int  *neighbor,
             int  * particle , int  *key, double rl2 , int  n,
             double Lx, double Ly, double Lz, int  al ,
             thrust :: device_ptr<int> key_thrust)
{
```

```
    keyzero<<<(BLOCK_SIZE + al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
                (key, neighbor, particle , n, al );

    cudaThreadSynchronize();

    check<<<(BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE>>>
              (xd, yd, zd, key, neighbor, particle , rl2 , n, Lx, Ly, Lz);

    cudaThreadSynchronize();
    //thrust :: device_ptr<int> key_thrust(key);
    //thrust :: device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
    key_thrust = thrust::device_pointer_cast(key);

    thrust :: inclusive_scan(key_thrust, key_thrust + n, key_thrust);
    //thrust :: device_free(key_thrust);

    cudaThreadSynchronize();

    setup<<<(BLOCK_SIZE + n)/BLOCK_SIZE, BLOCK_SIZE>>>
              (xd, yd, zd, key, particle , n);

    cudaThreadSynchronize();

    populate<<<(BLOCK_SIZE + al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
                (xd, yd, zd, neighbor, particle , n, rl2 , Lx, Ly, Lz, al );

    cudaThreadSynchronize();

// thrust :: device_free(key_thrust);

}
```

A sample parameter file for *mix_relax.cu* is given in Table C.1. The position input is simply the file *position_out0.txt* from the code *mix_strain.cu*. Even though the system given is a monolayer, $L_y = 10$ so that the spheres do not interact with mirror images in the y direction.

30.0	Lx
10.0	Ly
10.0	Lz
20	Number of spheres
1	Start
5000000	Number of steps
5000000	Print Statements
100	Number of Time Points
0.0001	dt
2.5	Cutoff radius
0.125	Hydrodynamic Cutoff Radius
1E-4	h correction factor
0.0000	Dimensionless shear
.01	Frequency, omega
0.00001	Strain amplitude, gamma0
2.7	Cutoff radius for neighbor list
100	Number of steps to calculate neighbor list
70	Neighbor list length
0	Initial time

Table C.1: File *parameters.txt* for the code *mix_relax.cu*

The position output file, *position_out_relaxed0.txt*, for *mix_relax.cu* is given in Table C.2.

Table C.2: File *position_out_relaxed0.txt* for the code *mix_relax.cu*

```
5000000

0        -1.000000  0.000000  -4.497730  1
1        -0.995068  0.000000  -3.495829  1
2        -0.974937  0.000000  -2.494263  1
3        -0.941111  0.000000  -1.493125  1
4        -0.894182  0.000000  -0.492522  1
```

```
5         -0.834284  0.000000  0.507425   1
6         -0.761061  0.000000  1.506569   1
7         -0.674946  0.000000  2.504444   1
8         -0.589151  0.000000  3.501990   1
9         -0.500000  0.000000  4.499250   1
10        1.000000   0.000000  -4.508943  1
11        1.135324   0.000000  -3.526375  1
12        1.237025   0.000000  -2.539913  1
13        1.317985   0.000000  -1.551620  1
14        1.378678   0.000000  -0.561931  1
15        1.419593   0.000000  0.428747   1
16        1.440610   0.000000  1.420639   1
17        0.978175   0.000000  2.417377   0
18        1.599448   0.000000  3.322618   0
19        1.466186   0.000000  4.408345   0
5000000

0         -1.000000  0.000000  -4.493486  1
1         -0.942081  0.000000  -3.489505  1
2         -0.870916  0.000000  -2.486493  1
3         -0.785525  0.000000  -1.484551  1
4         -0.686022  0.000000  -0.483784  1
5         -0.571897  0.000000  0.515650   1
6         -0.442032  0.000000  1.513520   1
7         -0.294037  0.000000  2.508487   1
8         -0.149717  0.000000  3.503023   1
9         0.000000   0.000000  4.496795   1
10        1.000000   0.000000  -4.509144  1
11        1.153968   0.000000  -3.529391  1
12        1.271557   0.000000  -2.544774  1
13        1.366701   0.000000  -1.557825  1
14        1.439180   0.000000  -0.569006  1
15        1.488822   0.000000  0.421207   1
16        1.514721   0.000000  1.412937   1
17        1.320978   0.000000  2.493480   0
18        2.045512   0.000000  3.318398   0
19        1.941178   0.000000  4.407846   0
5000000

0         -1.000000  0.000000  -4.486698  1
```

```
1         -0.892962  0.000000  -3.481054  1
2         -0.773751  0.000000  -2.476793  1
3         -0.638943  0.000000  -1.474079  1
4         -0.487730  0.000000  -0.472960  1
5         -0.318199  0.000000  0.526526   1
6         -0.127255  0.000000  1.524695   1
7         0.098000   0.000000  2.515144   1
8         0.301854   0.000000  3.504501   1
9         0.500000   0.000000  4.494756   1
10        1.000000   0.000000  -4.509368  1
11        1.156332   0.000000  -3.530196  1
12        1.274395   0.000000  -2.545827  1
13        1.367834   0.000000  -1.558884  1
14        1.435516   0.000000  -0.569865  1
15        1.476492   0.000000  0.420649   1
16        1.487809   0.000000  1.412630   1
17        1.731220   0.000000  2.480517   0
18        2.443608   0.000000  3.313232   0
19        2.419528   0.000000  4.407313   0
5000000

0         -1.000000  0.000000  -4.508727  1
1         -1.106341  0.000000  -3.522564  1
2         -1.182179  0.000000  -2.533713  1
3         -1.238281  0.000000  -1.543620  1
4         -1.277815  0.000000  -0.552849  1
5         -1.303921  0.000000  0.438354   1
6         -1.317014  0.000000  1.430406   1
7         1.046665   0.000000  2.500532   1
8         1.023798   0.000000  3.500287   1
9         1.000000   0.000000  4.500021   1
10        1.000000   0.000000  -4.499302  1
11        1.044475   0.000000  -3.499601  1
12        1.071772   0.000000  -2.499474  1
13        1.085915   0.000000  -1.499218  1
14        1.089496   0.000000  -0.499162  1
15        1.084068   0.000000  0.500862   1
16        1.069923   0.000000  1.500786   1
17        2.158555   0.000000  2.447872   0
18        2.849065   0.000000  3.308027   0
```

```
19        2.894211   0.000000  4.404733   0
5000000

0         -1.000000  0.000000  -4.508652  1
1         -1.074744  0.000000  -3.519670  1
2         -1.121519  0.000000  -2.529069  1
3         -1.151350  0.000000  -1.537959  1
4         -1.168015  0.000000  -0.546550  1
5         -1.172496  0.000000  0.444989   1
6         -1.172215  0.000000  1.437135   1
7         1.412643   0.000000  2.498505   1
8         1.455488   0.000000  3.498728   1
9         1.500000   0.000000  4.498884   1
10        1.000000   0.000000  -4.498300  1
11        1.091572   0.000000  -3.500394  1
12        1.164661   0.000000  -2.501261  1
13        1.227165   0.000000  -1.501862  1
14        1.280993   0.000000  -0.502058  1
15        1.326942   0.000000  0.498059   1
16        1.369732   0.000000  1.498284   1
17        2.539271   0.000000  2.461389   0
18        3.254235   0.000000  3.306840   0
19        3.366718   0.000000  4.401837   0
5000000

0         -1.000000  0.000000  -4.508515  1
1         -1.050577  0.000000  -3.517948  1
2         -1.077977  0.000000  -2.526649  1
3         -1.089372  0.000000  -1.535165  1
4         -1.088657  0.000000  -0.543614  1
5         -1.088702  0.000000  0.447938   1
6         -1.088699  0.000000  1.440085   1
7         1.787975   0.000000  2.496721   1
8         1.892099   0.000000  3.496321   1
9         2.000000   0.000000  4.495553   1
10        1.000000   0.000000  -4.495325  1
11        1.139439   0.000000  -3.498953  1
12        1.259679   0.000000  -2.500665  1
13        1.371212   0.000000  -1.501651  1
14        1.475213   0.000000  -0.502038  1
```

```
15        1.579471   0.000000  0.497548   1
16        1.683721   0.000000  1.497135   1
17        2.918676   0.000000  2.468562   0
18        3.664386   0.000000  3.307632   0
19        3.837813   0.000000  4.398828   0
5000000

0         -1.000000  0.000000  -4.508502  1
1         -1.035614  0.000000  -3.517312  1
2         -1.050095  0.000000  -2.525754  1
3         -1.049187  0.000000  -1.534203  1
4         -1.049244  0.000000  -0.542651  1
5         -1.049240  0.000000  0.448901   1
6         -1.049241  0.000000  1.441047   1
7         2.170538   0.000000  2.495339   1
8         2.332503   0.000000  3.493860   1
9         2.500000   0.000000  4.491582   1
10        1.000000   0.000000  -4.491271  1
11        1.190245   0.000000  -3.495170  1
12        1.360088   0.000000  -2.497171  1
13        1.521991   0.000000  -1.498633  1
14        1.684132   0.000000  -0.500141  1
15        1.846265   0.000000  0.498353   1
16        2.008399   0.000000  1.496846   1
17        3.294427   0.000000  2.469512   0
18        4.080027   0.000000  3.312381   0
19        4.308128   0.000000  4.396403   0
5000000

0         -1.000000  0.000000  -4.508470  1
1         -1.000320  0.000000  -3.516766  1
2         -1.000300  0.000000  -2.525214  1
3         -1.000301  0.000000  -1.533663  1
4         -1.000301  0.000000  -0.542112  1
5         -1.000301  0.000000  0.449440   1
6         -1.000301  0.000000  1.441586   1
7         2.618648   0.000000  2.551489   1
8         2.804570   0.000000  3.529641   1
9         3.000000   0.000000  4.505974   1
10        1.000000   0.000000  -4.505218  1
```

```
11        1.502430   0.000000  -3.317697  1
12        1.686799   0.000000  -2.338854  1
13        1.873250   0.000000  -1.360800  1
14        2.059591   0.000000  -0.382726  1
15        2.245937   0.000000  0.595347   1
16        2.432282   0.000000  1.573420   1
17        3.723429   0.000000  2.465083   0
18        4.497049   0.000000  3.317852   0
19        4.777669   0.000000  4.393429   0
5000000

0         -1.000000  0.000000  -4.508469  1
1         -1.000255  0.000000  -3.516765  1
2         -1.000239  0.000000  -2.525214  1
3         -1.000240  0.000000  -1.533662  1
4         -1.000240  0.000000  -0.542111  1
5         -1.000240  0.000000  0.449441   1
6         -1.000240  0.000000  1.441587   1
7         3.499976   0.000000  2.525707   1
8         3.499988   0.000000  3.517259   1
9         3.500000   0.000000  4.508799   1
10        1.000000   0.000000  -4.507538  1
11        3.499976   0.000000  -3.424196  1
12        3.499976   0.000000  -2.432050  1
13        3.499976   0.000000  -1.440498  1
14        3.499976   0.000000  -0.448947  1
15        3.499976   0.000000  0.542605   1
16        3.499976   0.000000  1.534156   1
17        4.584993   0.000000  2.315758   0
18        4.938369   0.000000  3.358168   0
19        5.263080   0.000000  4.405308   0
5000000

0         -1.000000  0.000000  -4.508469  1
1         -1.000255  0.000000  -3.516765  1
2         -1.000239  0.000000  -2.525214  1
3         -1.000240  0.000000  -1.533662  1
4         -1.000240  0.000000  -0.542111  1
5         -1.000240  0.000000  0.449441   1
6         -1.000240  0.000000  1.441587   1
```

7	3.999980	0.000000	2.525707	1
8	3.999990	0.000000	3.517259	1
9	4.000000	0.000000	4.508799	1
10	1.000000	0.000000	-4.507538	1
11	3.999980	0.000000	-3.424196	1
12	3.999980	0.000000	-2.432050	1
13	3.999980	0.000000	-1.440498	1
14	3.999980	0.000000	-0.448947	1
15	3.999980	0.000000	0.542605	1
16	3.999980	0.000000	1.534156	1
17	5.083646	0.000000	2.299502	0
18	5.354356	0.000000	3.369432	0
19	5.730386	0.000000	4.402375	0

Appendix D

Mix_LAOS.cu

This appendix contains the code *Mix_LAOS.cu*. This code is used to strain the suspensions. Every `n_print` configuration is saved.

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#include <thrust/scan.h>
#include <thrust/sort.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_free.h>
#include <thrust/device_malloc.h>

#define BLOCK_SIZE 512

//Zero out the vectors associated with the neighbor list
__global__ void keyzero(int *key, int *neighbor, int *particle, int n, int al)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;

    if(tid < n)
    {
        key[tid] = 0;
    }

    if(tid < al*n)
    {
        // Initialize the neighbor and particle vectors to -1 because there is
        //no -1 particle
        neighbor[tid] = -1;
        particle[tid] = -1;
    }
}

//This kernel finds out how many nearest neighbors a particular sphere has.
__global__ void check(double *x, double *y, double *z, int *key,
                      int *neighbor, int *particle, double rl2, int n, double Lx,
                      double Ly, double Lz)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    int i;
    double xij, yij, zij, r2, xmag, ymag; //, zmag;

    if(tid < n)
    {
        for(i = 0; i < n; i++)
        {
            if(tid != i)
            {
                xij = x[i] - x[tid];
                yij = y[i] - y[tid];
                zij = z[i] - z[tid];

                xmag = fabs(xij);
                ymag = fabs(yij);
                //zmag = fabs(zij);

                if(xmag > .5*Lx)
                    xij = xij*(1 - Lx/xmag);
                if(ymag > .5*Ly)
                    yij = yij*(1 - Ly/ymag);

                r2 = xij*xij + yij*yij + zij*zij;


                if(r2 < rl2)
                {
                    //Atomic add will cause it so that individual threads all
                    //have the same value of key[i]. Otherwise, each tid would
                        //access and save a different value of key[i].
                    atomicAdd( &(key[i]), 1);
                    //particle[i] = tid;
                }
            }
        }
    }
}

//This occurs after an inclusive parallel reduction scan by the thrust library.
//The inclusive reduction scan changes the key vector such that it identifies
//which indices belong to which sphere. That is, if sphere 0 has 10
//interactions, the key array will look like key = [0, 10, ...]. For
//more information on the inclusive scan, see the thrust literature.

//This creates the "particle" list part of the neighbor list. It does not
//take advantage of the fact that rij = -rji. Therefore, if thread 0 interacts
//with 10 spheres, there will be ten entries for thread 0.
__global__ void setup(double *x, double *y, double *z, int *key,
                      int *particle, int n)
{
```

```
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, beg, end;

   if(tid < n)
   {
      //tid represents the sphere number here
      //therefore, the last entry for sphere tid
      //will be key[tid-1], however, since the loop is
      //i < end and not i <= end, key[tid] is treated as
      //the last entry of the sphere.  Continuing with the above example,
      //the last entry of sphere 0 will occur at particle [9].  The particle vector
      //will then look like:
      //particle [0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  ....]
      end = key[tid];

      if(tid == 0)
         beg = 0;
      else
         beg = key[tid - 1];

      for(i = beg; i < end; i++)
          particle[i] = tid;
   }
}

//This populates the neighbor list with the spheres associated with the particles
//in the vector "particle".
__global__ void populate(double *x, double *y, double *z, int *neighbor, int *particle,
                         int n, double rl2, double Lx, double Ly, double Lz, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int start, tpart, jspher, tcount;
   double xij, yij, zij, r2, xmag, ymag;

   if(tid < al*n)
   {

      if(particle[tid] != -1)
      {
         tcount = 0;
         //This identifies the first sphere in each segment of spheres.
         //For example, if sphere 0 has 10 spheres, the block of 0s
            //will start at 0 and end at 9.  Threads 1-9 do not pass
            //through the if statement since particle[tid] ==
            //particle[tid - 1].  However, tid = 10 will pass through
            //because it is the first entry of sphere 1.
      if((particle[tid] != particle[tid - 1] | tid == 0))
      {
         start = particle[tid];
         tpart = start;

         //This will cycle through all the spheres until the total number
             //of neighbors for each sphere in the particle vector
             //are found.
         for(jspher = 0; jspher < n; jspher++)
         {

            xij = x[jspher] - x[tpart];
            yij = y[jspher] - y[tpart];
            zij = z[jspher] - z[tpart];

            xmag = fabs(xij);
            ymag = fabs(yij);
            //zmag = fabs(zij);

            if(xmag > .5*Lx)
               xij = xij*(1 - Lx/xmag);
            if(ymag > .5*Ly)
               yij = yij*(1 - Ly/ymag);

            r2 = xij*xij + yij*yij + zij*zij;

            if(r2 < rl2)
            {
               if(tpart != jspher)
               {
                  neighbor[tid + tcount] = jspher;
                  tcount++;
                  tpart = particle[tid + tcount];
                  //If the the next element in the particle array is
                        //not equal  to the particle[tid], then you have
                        //found all the neighbors of particle[tid]
                  if(tpart != start)
                     break;
               }
            }
         }
      }
   }
}
```

```
}

// Initializes all the forces to zero.
___global___ void init_force(double *fx, double *fy, double *fz, int n, double
*fxtot, double *fytot, double *fztot, int al, double *tau)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;

   if(tid<n)
   {
      fxtot[tid] = 0;
      fytot[tid] = 0;
      fztot[tid] = 0;

      tau[0] = 0;
   }

   if(tid < al*n)
   {
      fx[tid] = 0;
      fy[tid] = 0;
      fz[tid] = 0;
   }
}

//Calculates the interparticle forces in parallel.
___global___ void force_calc(double *x, double *y, double *z, double *fx, double
*fy, double *fz, int n, double rc2, double Lx, double Ly, double Lz, int
*neighbor, int *particle, int *mag_key, double rc, double *fxtot, double *fytot,
double *fztot, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, j;
   double xij, yij, zij, xmag, ymag, zmag, r2, r, zpp, ri2, ri, ri4;
   double wh, iterm, jterm, ijterm;
   double C, C2, r4, rep;
   double fxi, fyi, fzi;


   if(tid < al*n)
   {

      if(particle[tid] != -1)
      {
         //Since this algorithm does not take into account rij = -rji, this is
            //just a straight calculation.
         i = particle[tid];
         j = neighbor[tid];

         iterm = (double)mag_key[i];
         jterm = (double)mag_key[j];
         ijterm = iterm*jterm;

         zmag = fabs(z[i]);
         wh = (Lz/2.00) - zmag;

         xij = x[j] - x[i];
         yij = y[j] - y[i];
         zij = z[j] - z[i];

         xmag = fabs(xij);
         ymag = fabs(yij);

         if(xmag > (0.5*Lx))
            xij = xij*(1 - Lx/xmag);
         if(ymag > (0.5*Ly))
            yij = yij*(1 - Ly/ymag);

         r2 = xij*xij + yij*yij + zij*zij;

         if(r2 < rc2)
         {

         // iterm = (double)mag_key[i];
         // jterm = (double)mag_key[j];
         // ijterm = iterm*jterm;

            r = sqrt(r2);
            C = zij/r;
            C2 = 5.0*C*C;
            r4 = r*r*r*r;

            rep = exp((1.0 - r)/0.01);

            fx[tid] = ((ijterm*(C2 - 1.0)/(r4)) - rep)*(xij/r);
            fy[tid] = ((ijterm*(C2 - 1.0)/(r4)) - rep)*(yij/r);
            fz[tid] = ((ijterm*(C2 - 3.0)/(r4)) - rep)*C;
         }

         if(wh < rc)
```

```
        {
            zpp = (z[i]/zmag)*Lz - z[j] - z[i];
            ri2  = zpp*zpp + xij*xij + yij*yij;


            if(ri2 < rc2)
            {
                ri  = sqrt(ri2);
                C = zpp/ri;
                C2 = 5.0*C*C;
                ri4  = ri*ri*ri*ri;

                fxi  = (xij/ri)*ijterm*(C2 - 1.0)/ri4;
                fyi  = (yij/ri)*ijterm*(C2 - 1.0)/ri4;
                fzi  = C*ijterm*(C2 - 3.0)/ri4;

//              fxi  = (xij/ri)*(C2 - 1.0)/ri4;
//              fyi  = (yij/ri)*(C2 - 1.0)/ri4;
//              fzi  = C*(C2 - 3.0)/ri4;

                fx[tid]  += fxi;
                fy[tid]  += fyi;
                fz[tid]  += fzi;
            }
        }

    }
  }
}


__global__ void force_total(double *z, double *fx, double *fy,
                double *fz, double *fxtot, double  *fytot,
                double *fztot, int *particle, int *mag_key, double Lz,
                double rc, int n, int *neighbor, int al)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  int    i   , check, redkey;
  double sumx, sumy , sumz ;

  if(tid < al*n)
  {
    if( particle[tid] != -1)
    {
      check = particle[tid];

      if(tid == 0 | check != particle[tid - 1])
      {
        sumx = fx[tid];
        sumy = fy[tid];
        sumz = fz[tid];
        i = tid;

        while(check == particle[i+1])
        {
          sumx += fx[i+1];
          sumy += fy[i+1];
          sumz += fz[i+1];
          i++;
          check = particle[i];
        }

        redkey = particle[tid];

        fxtot[redkey] = sumx;
        fytot[redkey] = sumy;
        fztot[redkey] = sumz;
      }
    }
  }

}

__global__ void force_wall(double *z, double Lz, double *fztot, int n,
                double rc, int *mag_key, double *fxtot, double *fytot,
                double *tau)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  double wh, wh4, iterm, zmag, term1, term2, fzw;


  if(tid < n)
  {
    iterm = (double)mag_key[tid];
    zmag = fabs(z[tid]);
    wh = (Lz/2.00) - zmag;

//    if((z[tid] + 0.5) > (0.5*Lz - 0.05))
//    {
//      fx = (float)fxtot[tid];
```

```
//          atomicAdd( &(tau[0]), -fx);
//          printf("hello\n");
//
//      }


        if(wh < (.5*rc))
        {
            wh4 = wh*wh*wh*wh;
            term1 = iterm*(1.0/wh4)/8.0;
            term2 = exp((0.5-wh)/0.01);

            fzw = term1 - term2;
            fztot[tid] += fzw*(z[tid]/zmag);
        }
    }
}

___global___ void update_pos(double *x, double *y, double *z,
                    double *fxtot, double *fytot, double *fztot,
                    double Lx, double Ly, double Lz, int n,
                    double gd, double dt)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    double xmag, ymag, zmag, dx, dy, dz;



    if(tid < n)
    {
        //xmag = fabs(x[tid]);
        //ymag = fabs(y[tid]);
        zmag = fabs(z[tid]);


        dz = fztot[tid]*dt;

        if(zmag >= (0.5*Lz - .5 - .05))
        {
            dy = 0;
            if(z[tid] < 0)
                dx = 0;
            else
                dx = gd*Lz*dt;
        }
        else
        {
            dx = (fxtot[tid] + gd*(z[tid] + 0.5*Lz))*dt;
            dy = (fytot[tid])*dt;
        }

        x[tid] += dx;
        y[tid] += dy;
        z[tid] += dz;

        xmag = fabs(x[tid]);
        ymag = fabs(y[tid]);

        if(xmag > .5*Lx)
            x[tid] *= (1 - Lx/xmag);
        if(ymag > .5*Ly)
            y[tid] *= (1 - Ly/ymag);
    }
}

___global___ void tau_calc(double *z, double *fxtot, double *tau, int n,
                           double Lz)
{
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    int i;

    if(tid == 0)
    {
        *tau = 0;
        for(i = 0; i<n; i++)
        {
            //if((z[i] + 0.5) > (0.5*Lz - 0.05))
                *tau += z[i]*(fxtot[i]);
        }
    }
}



void nlist( double *, double *, double *, int *, int *, int *, double, int,
            double, double, double, int, thrust::device_ptr<int>);



int main(void)
```

```
{
  int      *key    = NULL , *particle = NULL, *neighbor = NULL;
   int      *mag_key = NULL, int *mag_keyd = NULL;
  int       n, kstart, nsteps, nprint;
   int       lsteps , i , k, index ,  lll ,  al,  num_pos ;
   int       file_select ;
  double  Lx, Ly, Lz, dt;
   double  rc , rch, corrfac , gd, rl  , sphere, rc2  , rl2 ;
  double  omega, gamma0, time,  timeold         ;
   double  gamma           , gammaold, dgamma , g;
  double *xd     = NULL , *yd       = NULL, *zd       = NULL;
  double *x      = NULL , *y        = NULL, *z        = NULL;
  double *fx     = NULL , *fy       = NULL, *fz       = NULL;
  double *fxtot = NULL, *fytot     = NULL, *fztot    = NULL;
  double *tau_h = NULL, *tau_d = NULL;
  double  ctime_tot = 0, ctime_avg, timetot;
  int       timei , n_config;
  int       cuda_count = 0;
  float     elapsedTime;
  char      parameters[] = "parameters.txt";
  FILE   *pos_input, *par_input , *pos_output, *time_out , *tau_out ;
  char      tt [80]     ,   p_out[100],  p_in[100],    t_out[100];
  //double *dabsx_h = NULL, *dabsy_h = NULL, *dabsz_h = NULL;
  //double *dabsx_d = NULL, *dabsy_d = NULL, *dabsz_d = NULL;

   file_select  = 0;


   sprintf(p_out, "position_out_LAOS%05d.txt" , file_select);
  pos_output = fopen(p_out, "w");

   sprintf(p_in, "position_out_relaxed%d.txt"          ,  file_select );
  pos_input = fopen(p_in, "r");

   sprintf(t_out, "tau_out_LAOS_%05d.txt"   ,  file_select );
  tau_out = fopen(t_out, "w");


  clock_t t0, t1;

  par_input = fopen(parameters, "r");
   fscanf(par_input, "%lf",  &Lx);         fgets(tt ,80, par_input);
   fscanf(par_input, "%lf",  &Ly);         fgets(tt ,80, par_input);
   fscanf(par_input, "%lf",  &Lz);         fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &n);           fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &kstart);      fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &nsteps);      fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &nprint);      fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &dt);          fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &rc);          fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &rch);         fgets(tt,  80,  par_input);
   fscanf(par_input, "%lf", &corrfac);     fgets(tt,  80,  par_input);
   fscanf(par_input, "%lf", &gd);          fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &omega);       fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &gamma0); fgets(tt ,80, par_input);
   fscanf(par_input, "%lf", &rl);          fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &lsteps);      fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &al);          fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &n_config); fgets(tt ,80, par_input);
   fscanf(par_input, "%d ", &num_pos); fgets(tt,80,par_input);
   fscanf(par_input, "%d ", &timei);       fgets(tt ,80, par_input);
   fclose (par_input);

  rc2  = rc*rc;
  rl2  = rl*rl;



  x = (double*)malloc(n*sizeof(double));
  y = (double*)malloc(n*sizeof(double));
  z = (double*)malloc(n*sizeof(double));


  tau_h = (double*)malloc(sizeof(double));
  cudaMalloc((void**)&tau_d, sizeof(double));
  *tau_h = 0;

  mag_key = (int*)malloc(n*sizeof(int));

  cudaMalloc((void**)&xd, n*sizeof(double));
  cudaMalloc((void**)&yd, n*sizeof(double));
  cudaMalloc((void**)&zd, n*sizeof(double));

  /*thrust::device_vector<double> xdt(n);
  thrust::device_vector<double> ydt(n);
  thrust::device_vector<double> zdt(n);

  double *xd = thrust::raw_pointer_cast(&xdt[0]);
  double *yd = thrust::raw_pointer_cast(&ydt[0]);
  double *zd = thrust::raw_pointer_cast(&zdt[0]);
```

```
    thrust::device_vector<double> fxtott(n);
    thrust::device_vector<double> fytott(n);
    thrust::device_vector<double> fztott(n);

    double *fxtot = thrust::raw_pointer_cast(&fxtott[0]);
    double *fytot = thrust::raw_pointer_cast(&fytott[0]);
    double *fztot = thrust::raw_pointer_cast(&fztott[0]);*/

    cudaMalloc((void**)&fxtot, n*sizeof(double));
    cudaMalloc((void**)&fytot, n*sizeof(double));
    cudaMalloc((void**)&fztot, n*sizeof(double));

    /*thrust::device_vector<double> fxt(m*n);
    thrust::device_vector<double> fyt(m*n);
    thrust::device_vector<double> fzt(m*n);

    double *fx = thrust::raw_pointer_cast(&fxt[0]);
    double *fy = thrust::raw_pointer_cast(&fyt[0]);
    double *fz = thrust::raw_pointer_cast(&fzt[0]);*/

    cudaMalloc((void**)&fx, al*n*sizeof(double));
    cudaMalloc((void**)&fy, al*n*sizeof(double));
    cudaMalloc((void**)&fz, al*n*sizeof(double));

    //thrust::device

    cudaMalloc((void**)&key     ,         n * sizeof(int));
    cudaMalloc((void**)&particle, al * n * sizeof(int));
    cudaMalloc((void**)&neighbor, al * n * sizeof(int));
    cudaMalloc((void**)&mag_keyd,    n * sizeof(int));
    thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);

    //thrust::device_ptr<int> key_thrust(key);

// pos_input = fopen(position, "r");

// pos_output = fopen("position_out1.xyz", "w");
// tau_out = fopen("tau_out0.txt", "w");

    for(lll = 0; lll < (n_config); lll++)
    {

        //if(lll == 0)
        fgets(tt,80,pos_input);
```

```
//fgets(tt,80,pos_input);

    for(i = 0; i<n; i++)
    {
        fscanf(pos_input, "%lf", &sphere);
        fscanf(pos_input, "%lf", &(x[i]));
        fscanf(pos_input, "%lf", &(y[i]));
        fscanf(pos_input, "%lf", &(z[i]));
        fscanf(pos_input, "%d ", &(mag_key[i]));

        //if(i != (n - 1))
        // fgets(tt, 80, pos_input);
        //dxabs[i] = 0;
        //dyabs[i] = 0;
        //dzabs[i] = 0;
    }

cudaMemcpy(xd, x, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(yd, y, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(zd, z, n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(mag_keyd, mag_key, n*sizeof(int),
           cudaMemcpyHostToDevice);

//thrust::device_ptr<int>key_thrust = thrust::device_malloc<int>(n);

nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz,
      al, key_thrust);

//pos_output = fopen("position_out.xyz", "w");
//pos_out2 = fopen("pos_out2.txt", "w");
//tau_out = fopen("tau_out.txt", "w");

cudaEvent_t startEvent, stopEvent;
cudaEventCreate(&startEvent);
cudaEventCreate(&stopEvent);



//thrust::device_ptr<int>key_thrust=thrust::device_malloc<int>(n);
init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
    (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
    (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor,
        particle, mag_keyd, rc, fxtot, fytot, fztot, al);
```

```
  force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
                    (zd, fx, fy, fz, fxtot, fytot, fztot, particle,
                     mag_keyd, Lz, rc, n, neighbor, al);
force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
          (zd, Lz, fztot, n, rc, mag_keyd, fxtot, fytot, tau_d);

if(timei == 0)
{
   //printf("hello\n");
   time = dt*timei;
   gamma = gamma0*sin(omega*time);
   tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                    (zd, fxtot, tau_d, n, Lz);
   cudaMemcpy(tau_h, tau_d, sizeof(double), cudaMemcpyDeviceToHost);

   *tau_h = (-*tau_h)/(Lx*Ly*(Lz-1.0));
   g = *tau_h/gamma0;
    fprintf(tau_out," % lf          % lf          % 16.12lf\n", time, gamma, g);
}



t0 = clock();

for(k = kstart; k < (nsteps + 1); k++)
{
   time = dt*(double)k;
   timeold = dt*(double)(k-1);
   gamma = gamma0*sin(omega*time);
   gammaold = gamma0*sin(omega*timeold);
   dgamma = gamma - gammaold;
   gd = dgamma/dt;
   //init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE, BLOCK_SIZE>>>
      //(fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
   //cudaThreadSynchronize();
   if((k%lsteps) == 0)
      nlist(xd, yd, zd, neighbor, particle, key, rl2, n,
                Lx, Ly, Lz, al, key_thrust);

   cudaThreadSynchronize();

   cudaEventRecord(startEvent, 0);

   //force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE >>>
      //(xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz, neighbor,
      //particle, mag_keyd, rc, fxtot, fytot, fztot, al);

//cudaThreadSynchronize();

//force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
   //BLOCK_SIZE>>>(zd, fx, fy, fz, fxtot, fytot,
   //fztot, particle, mag_keyd, Lz, rc, n, neighbor, al);

//cudaThreadSynchronize();

//force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,
   //BLOCK_SIZE>>>(zd, Lz, fztot, n, rc, mag_keyd,
   //fxtot, fytot, tau_d);

cudaThreadSynchronize();

update_pos<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                (xd, yd, zd, fxtot, fytot, fztot, Lx,
                 Ly, Lz, n, gd, dt);

cudaThreadSynchronize();

init_force<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
              (fx, fy, fz, n, fxtot, fytot, fztot, al, tau_d);
force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
                (xd, yd, zd, fx, fy, fz, n, rc2, Lx, Ly, Lz,
                 neighbor, particle, mag_keyd, rc, fxtot,
                 fytot, fztot, al);
force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
                (zd, fx, fy, fz, fxtot, fytot, fztot,
                  particle, mag_keyd, Lz, rc, n, neighbor, al);
force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                (zd, Lz, fztot, n, rc, mag_keyd, fxtot,
                 fytot, tau_d);

cudaEventRecord(stopEvent,0);
cudaEventSynchronize(stopEvent);

cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);

ctime_tot += elapsedTime;
cuda_count++;


if(k % nprint == 0)
```

```
{
    printf("%d    %d\n", k, lll );

    cudaMemcpy(tau_h, tau_d, sizeof(double), cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    *tau_h = 0;

    tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                    (zd, fxtot, tau_d, n, Lz);
    cudaMemcpy(tau_h, tau_d, sizeof(double), cudaMemcpyDeviceToHost);

    *tau_h = (-*tau_h)/(Lx*Ly*(Lz-1.0));
    g = *tau_h/gamma0;

    fprintf(tau_out," %lf          %lf          %16.12lf\n", time, gamma, g);

    if( lll  > (n_config - num_pos))
    {
        fprintf(pos_output, " %d\n\n", k);

          //tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,
             //BLOCK_SIZE>>>(zd, fxtot, tau_d, n, Lz);
       //cudaThreadSynchronize();

       cudaMemcpy(x,xd,n*sizeof(double),cudaMemcpyDeviceToHost);
       cudaThreadSynchronize();
       cudaMemcpy(y,yd,n*sizeof(double),cudaMemcpyDeviceToHost);
       cudaThreadSynchronize();
       cudaMemcpy(z,zd,n*sizeof(double),cudaMemcpyDeviceToHost);
       cudaThreadSynchronize();

    //gtot = k*gd*dt;
    //*tau_h = 0;

    //tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,
        //BLOCK_SIZE>>>(zd, fxtot, tau_d, n, Lz);
    //cudaMemcpy(tau_h, tau_d, sizeof(double),
        //cudaMemcpyDeviceToHost);

    //*tau_h = (-*tau_h)/(Lx*(Lz-1.0));
    //g = *tau_h/gamma0;

    // fprintf(tau_out," %lf          %lf          %lf\n", time, gamma, g);

        for(index = 0; index < n; index++)
        {
            fprintf(pos_output, "      %d     ", index);
            fprintf(pos_output, "      %lf       ", x[index]);
            fprintf(pos_output, "      %lf    ", y[index]);
            fprintf(pos_output, "      %lf       ", z[index]);
            fprintf(pos_output, "      %d      \n", mag_key[index]);
        }

        //fgets(tt,80,pos_input);
        //fgets(tt,80,pos_input);
        //fgets(tt,80,pos_input);
    }
}
    }
    fgets(tt,80,pos_input);
    //fgets(tt,80,pos_input);
    //fgets(tt,80,pos_input);
    fprintf(tau_out, "\n");
}

fclose(pos_input);
t1 = clock();
timetot = ((double)(t1-t0)/(double)CLOCKS_PER_SEC)/60;

time_out = fopen("time.txt", "w");


ctime_avg = ctime_tot/((double)cuda_count);

fprintf(time_out, "Total simulation time = %lf  minutes \n", timetot);
fprintf(time_out, "The time needed to perform the calculations is %lf
        ms\n", ctime_avg);

fclose(time_out);

thrust::device_free(key_thrust);

fclose   (pos_output);
fclose   (tau_out);
cudaFree(xd);
cudaFree(yd);
cudaFree(zd);
cudaFree(mag_keyd);
cudaFree(neighbor);
```

```
    cudaFree(particle);
    cudaFree(key);
    cudaFree(fx);
    cudaFree(fy);
    cudaFree(fz);
    cudaFree(fxtot);
    cudaFree(fytot);
    cudaFree(fztot);
    free      (x);
    free      (y);
    free      (z);
    free      (mag_key);

}


void  nlist(double *xd, double *yd, double *zd, int  *neighbor,
            int  *particle, int  *key, double rl2, int  n,
            double Lx, double Ly, double Lz, int  al,
thrust::device_ptr<int> key_thrust)
{

    keyzero<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
              (key, neighbor, particle, n, al);

    cudaThreadSynchronize();

    check<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
            (xd, yd, zd, key, neighbor, particle, rl2, n, Lx, Ly, Lz);

    cudaThreadSynchronize();
    //thrust::device_ptr<int> key_thrust(key);
    //thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
    key_thrust = thrust::device_pointer_cast(key);

    thrust::inclusive_scan(key_thrust, key_thrust + n, key_thrust);
    //thrust::device_free(key_thrust);

    cudaThreadSynchronize();

    setup<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
            (xd, yd, zd, key, particle, n);

    cudaThreadSynchronize();

    populate<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,BLOCK_SIZE>>>
              (xd, yd, zd, neighbor, particle, n, rl2, Lx, Ly, Lz, al);

    cudaThreadSynchronize();

// thrust::device_free(key_thrust);

}
```

A sample parameter input file for the code *mix_strain.cu* would look like:

```
30.0    Lx
10.0    Ly
10.0    Lz
20      Number of spheres
1       Start
5500    Number of steps
80      Print Statements
0.001   dt
2.5     Cutoff radius
0.125   Hydrodynamic Cutoff Radius
1E-4    h correction factor
0.0001  Dimensionless shear
0.01    Frequency, omega
0.0001  Strain amplitude, gamma0
2.7     Cutoff radius for neighbor list
100     Number of steps to calculate neighbor list
100     Neighbor list length
100     Number of Configurations
5       Number of positions to be printed
0       Initial time
```

Appendix E

Mix_Hydro.cu

This appendix contains the code *Mix_Hydro.cu*. This code is used to strain the suspensions. Hydrodynamic interactions are included. Every `n_print` configuration is saved.

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#include <time.h>
#include <thrust/scan.h>
#include <thrust/sort.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_free.h>
#include <thrust/device_malloc.h>
#include <cusolversp.h>
#include <cusparse.h>
#include <cublas_v2.h>

#define BLOCK_SIZE 512


// This kernel zeroes the vectors that make up the neighborlist.
__global__ void keyzero(int *key, int *neighbor, int *particle, int n, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;

   if (tid < n)
   {
      key[tid] = 0;
   }

   if (tid < al*n)
   {
      //In the force kernel, particle will be "i" and neighbor will be "j".
      //They are set to -1 because there is no particle "-1".
      neighbor[tid] = -1;
      particle[tid] = -1;
   }


}

//This kernel determines how many neighbors each particle has.
__global__ void check(double *x, double *y, double *z, int *key,
                      int *neighbor, int *particle, double rl2, int n,
                      double Lx, double Ly, double Lz)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   double xij, yij, zij, r2, xmag, ymag; //zmag;
   int i;



   if (tid < n)
   {
      //Each thread loops over every particle to determine which particles are
        //interacting.
      for (i = 0; i < n; i++)
      {
         if (tid != i)
         {
            xij = x[i] - x[tid];
            yij = y[i] - y[tid];
            zij = z[i] - z[tid];

            xmag = fabs(xij);
            ymag = fabs(yij);
            //zmag = fabs(zij);

            if (xmag > .5*Lx)
               xij = xij*(1 - Lx / xmag);
            if (ymag > .5*Ly)
               yij = yij*(1 - Ly / ymag);

            r2 = xij*xij + yij*yij + zij*zij;

            if (r2 < rl2)
            {
               //atomicAdd is used because it will add to an individual
                     //memory location. See Cuda by Example by Sanders and
                     //Kandrot or similar resource.
               atomicAdd(&(key[i]), 1);
            }
         }
      }
   }
}


//This kernel builds the particle vector. Note that this algorithm
//does not take into account that Fij = -Fji. Therefore, the particle
//vector will resemble something like: particle =[0; 0; 0; 0; 0; 1;
```

```
//1; 1; 1; 2; 2; 2;...-1; -1; -1] depending on how many interactions
//each particle has. Both vectors "particle" and "neighbor" are
//buffered by al*n. al stands for "array length".

__
global__ void setup(double *x, double *y, double *z, int *key, int *particle, int n)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, beg, end;

   if (tid < n)
   {
      //end and beg combine to tell the thread how many interactions each
        //particle has
      end = key[tid];

      if (tid == 0)
         beg = 0;
      else
         beg = key[tid - 1];

      for (i = beg; i < end; i++)
          particle[i] = tid;
   }
}

//This kernel populates the neighbor list vector with which particles are
//interacting with the particle in the vector "particle".
__global__ void populate(double *x, double *y, double *z, int *neighbor, int
*particle, int n, double rl2, double Lx, double Ly, double Lz, int al)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int start, tpart, jspher, tcount;
   double xij, yij, zij, r2, xmag, ymag;

   //If the thread id is inside the the size of the particle and neighbor vector
   if (tid < al*n)
   {
      //no particle "-1"
      if (particle[tid] != -1)
      {
         //tcount counts the number of iterations a thread makes when populating
         //the neighbor vector
         tcount = 0;

         if ((particle[tid] != particle[tid - 1] | tid == 0))
{
   //This tells the thread the particle whose neighbors it
       //is trying to find.
   start = particle[tid];
   tpart = start;

   //Loop over all particles, however, once the number of neighbors
       //a particle has is hit,
   //it exits.
   for (jspher = 0; jspher < n; jspher++)
   {

      xij = x[jspher] - x[tpart];
      yij = y[jspher] - y[tpart];
      zij = z[jspher] - z[tpart];

      xmag = fabs(xij);
      ymag = fabs(yij);
      //zmag = fabs(zij);

      if (xmag > .5*Lx)
         xij = xij*(1 - Lx / xmag);
      if (ymag > .5*Ly)
         yij = yij*(1 - Ly / ymag);

      r2 = xij*xij + yij*yij + zij*zij;

      if (r2 < rl2)
      {
         if (tpart != jspher)
         {
            neighbor[tid + tcount] = jspher;
            tcount++;
            //determine if the thread is still looking at the
                  //same particle if not, break the loop because
                  //the all neighbors have been accounted for.
            tpart = particle[tid + tcount];
            if (tpart != start)
               break;
         }
      }
   }
}
}
```

```
    }
}

//Set all the forces to zero.
___global___ void init_force(double *fx, double *fy, double *fz, int n,
                             double *fxtot, double *fytot, double *fztot,
                             int al, double *tau, double *Rmat, int *row_dx,
                             int *row_el, int *col_dx, double *val_r,
                             double *velocity_out, double *vx, double *vy,
                             double *vz, double *force_tot)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;

    if (tid<n)
    {
        fxtot     [tid] = 0;
        fytot     [tid] = 0;
        fztot     [tid] = 0;

        vx        [tid] = 0;
        vy        [tid] = 0;
        vz        [tid] = 0;

        force_tot[tid] = 0;

        tau       [0]   = 0;
    }

    if (tid < al*n)
    {
        fx[tid] = 0;
        fy[tid] = 0;
        fz[tid] = 0;
    }

    if (tid < 3 * n * 3 * n)
        Rmat[tid] = 0;

    if (tid < (3 * n + 1))
    {
        row_dx[tid] = 0;
    }

    if (tid < (3 * n * al))
    {
        col_dx[tid] = 0;
        val_r [tid] = 0;

    }

    if (tid < 3 * n)
        velocity_out[tid] = 0;
        row_el[tid] = 0;
}

//This kernel calculates each nonhydrodynamic force Fij. It does NOT
//calculate the total force on each particle. This is done in a later
//kernel.
___global___ void force_calc(double *x, double *y, double *z, double *fx,
                             double *fy, double *fz, int n, double rc2,
                             double Lx, double Ly, double Lz, int *neighbor,
                             int *particle, int *mag_key, double rc,
                             double *fxtot, double *fytot, double *fztot,
                             int al)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    int i, j, index;
    double xij, yij, zij, xmag, ymag, zmag, r2, r, zpp, ri2, ri, ri4;
    double wh, iterm, jterm, ijterm;
    double C, C2, r4, rep, h;
    double fxi, fyi, fzi;



    if (tid < al*n)
    {


        if (particle[tid] != -1)
        {
            //This part of the code is very similar to what is done in
              //the fortran codes. The only difference is that there
              //are no loops because i and j are calculated on al*n
              //different threads.
            i = particle[tid];
            j = neighbor[tid];

            iterm = (double)mag_key[i];
            jterm = (double)mag_key[j];
            ijterm = iterm*jterm;
```

```
zmag = fabs(z[i]);
wh = 0.5 * Lz - zmag;

xij  = x[j]  - x[i];
yij  = y[j]  - y[i];
zij  = z[j]  - z[i];

xmag = fabs(xij);
ymag = fabs(yij);

if (xmag > (0.5 * Lx))
    xij  = xij * (1 - Lx / xmag);
if (ymag > (0.5 * Ly))
    yij  = yij * (1 - Ly / ymag);

r2 = xij*xij + yij*yij + zij*zij;


if (r2 < rc*rc)
{

    r  = sqrt(r2);
    C  = zij / r;
    C2 = 5.0*C*C;
    //r4 = r*r*r*r;
    r4 = r2*r2;
    h  = r - 1.0;

    if (h < 0.0001)
       r += 0.0001;

    rep = exp((1.0 - r) / 0.01);// / (1.0 - exp((1.0 - r) / .01));


    fx[tid] = ((ijterm * (C2 - 1.0) / (r4)) - rep) * (xij / r);
    fy[tid] = ((ijterm * (C2 - 1.0) / (r4)) - rep) * (yij / r);
    fz[tid] = ((ijterm * (C2 - 3.0) / (r4)) - rep) * C;

    // This provides an extra resistance force to keep the gaps
        //between the particles from becoming unphysically close.
It is
        //from Melrose's paper on ER Fluids.

        if (h < 0.01)
        {
            fx[tid] -= (xij / r)*(1.0 - h / .01);
            fy[tid] -= (yij / r)*(1.0 - h / .01);
            fz[tid] -= (zij / r)*(1.0 - h / .01);
        }
    }


    //if (tid == 0)
      //printf("tid = %d    i = %d  j = %d    fz[%d] = %lf\n",
          //tid, i, j, fz[tid]);



    if (wh < rc)
    {
        zpp = (z[i] / zmag) * Lz - z[j] - z[i];
        ri2 = zpp * zpp + xij * xij + yij * yij;

        if (ri2 < rc*rc)
        {
            ri = sqrt(ri2);
            C = zpp / ri;
            C2 = 5.0*C*C;
            //ri4 = ri*ri*ri*ri;
            ri4 = ri2*ri2;

            fxi = (xij / ri) * ijterm * (C2 - 1.0) / ri4;
            fyi = (yij / ri) * ijterm * (C2 - 1.0) / ri4;
            fzi = C          * ijterm * (C2 - 3.0) / ri4;


            fx[tid] += fxi;
            fy[tid] += fyi;
            fz[tid] += fzi;

        }
    }
  }
 }
}

//This totals the nonhydrodynamic force on each particle.
__global__ void force_total(double *z, double *fx, double *fy, double *fz,
```

```
                        double *fxtot, double *fytot, double *fztot,
                        int *particle, int *mag_key, double Lz,
                        double rc, int n, int *neighbor, int al)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  int i;
  int check, redkey;
  double sumx, sumy, sumz;


  if (tid < al*n)
  {

    if (particle[tid] != -1)
    {
      //"check" identifies which particle the thread is considering
      check = particle[tid];

      //if tid looks at particle 0 OR tid is not looking at
        //the same particle as the thread behind it. Therefore,
        //only n threads are being utilized at once to total the
        //force on each particle.
      if (tid == 0 | check != particle[tid - 1])
      {
        sumx = fx[tid];
        sumy = fy[tid];
        sumz = fz[tid];
        i = tid;

        //This makes sure that the thread is only considering one particle.
        //Once it reaches a different particle, stop totaling the force.
        while (check == particle[i + 1])
        {
          sumx += fx[i + 1];
          sumy += fy[i + 1];
          sumz += fz[i + 1];
          i++;
          check = particle[i];
        }

        redkey = particle[tid];

        fxtot[redkey] = sumx;
        fytot[redkey] = sumy;
        fztot[redkey] = sumz;
      }
    }
  }
}

//This calculates the force from the wall on the particle. This is pretty
//similar to the fortran code only there is no iteration over n.
__global__ void force_wall(double *z, double Lz, double *fztot, int n, float
rc,
int *mag_key, double *fxtot, double *fytot, double *tau)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  double wh, wh4, iterm, zmag, term1, term2, fzw;

  if (tid < n)
  {
    iterm = (double)mag_key[tid];
    zmag = fabs(z[tid]);
    wh = (Lz / 2.00) - zmag;


    if (wh < (0.5 * rc))
    {
      wh4 = wh*wh*wh*wh;
      term1 = iterm * (1.0 / wh4) / 8.0;
      term2 = exp((0.5 - wh) / 0.01);

      fzw = term1 - term2;

      fztot[tid] += fzw * (z[tid] / zmag);

    }
  }
}

//This puts the three force vectors fxtot, fytot, fztot into one single vector.
//This allows for it the vector to be used when solve F = R*U.
//force_tot is 3Nx1.
__global__ void force_combine(double *force_tot, double *fxtot,
                              double *fytot, double *fztot,
                              int n, int *row_dx, int *nz_d,
                              double *val_r)
{
  int tid = threadIdx.x + blockDim.x*blockIdx.x;
  int part_num, i;
```

```
   if (tid < 3 * n)
   {
      if (tid == 0)
         nz_d[0] = row_dx[3 * n];

   // if (tid == 0)
      //{
         //for (i = 0; i < 3 * n; i++)
         //printf("val_r[%d] = %lf\n", tid, val_r[tid]);
//    }

      part_num = tid / 3;

      if (tid % 3 == 0)
         force_tot[tid] = fxtot[part_num];

      if (tid % 3 == 1)
         force_tot[tid] = fytot[part_num];

      if (tid % 3 == 2)
         force_tot[tid] = fztot[part_num];

   } // End if tid < 3n


} // End force_combine

//This builds the resistance matrix.
__global__ void hydro_2(int *particle, int *neighbor, double *x, double *y,
double *z, double *Rmat, double *d, int al, int n, double rc2, double corr_fac,
double Lx, double Ly, double Lz)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i, j, iloop, jloop;
   double r, r2, xij, yij, zij, h, dd, dx, dy, dz, xmag, ymag;

   if (tid < (al * n))
   {

      if (particle[tid] != -1)
      {
         i = particle[tid];
         j = neighbor[tid];

         xij = x[j] - x[i];
         yij = y[j] - y[i];
         zij = z[j] - z[i];

         xmag = fabs(xij);
         ymag = fabs(yij);

         if (xmag > .5*Lx)
            xij = xij*(1.0 - Lx / xmag);
         if (ymag > .5*Ly)
            yij = yij*(1.0 - Ly / ymag);

         r2 = xij * xij + yij * yij + zij * zij;


         if (r2 < rc2)
         {
            r = sqrt(r2);
            h = r - 1.0;

            if (h < corr_fac)
            {
               r += corr_fac;
               h = corr_fac;
            } // End if h < corr_fac

            //d[0] = xij / r;
            //d[1] = yij / r;
            //d[2] = zij / r;

            dx = xij / r;
            dy = yij / r;
            dz = zij / r;


            if (h < 0.125)
            {
               for (iloop = 0; iloop < 3; iloop++)
               {
                  for (jloop = 0; jloop < 3; jloop++)
                  {
                     //dd = d[iloop] * d[jloop];

                     //This could probably be done more efficiently, but
                           //it was easier to debug as written.
```

```
if (iloop == 0 && jloop == 0)
{
   dd = dx * dx;
}

if (iloop == 0 && jloop == 1)
{
   dd = dx * dy;
}

if (iloop == 0 && jloop == 2)
{
   dd = dx * dz;
}

if (iloop == 1 && jloop == 0)
{
   dd = dy * dx;
}

if (iloop == 1 && jloop == 1)
{
   dd = dy * dy;
}

if (iloop == 1 && jloop == 2)
{
   dd = dy * dz;
}

if (iloop == 2 && jloop == 0)
{
   dd = dz * dx;
}

if (iloop == 2 && jloop == 1)
{
   dd = dz * dy;
}

if (iloop == 2 && jloop == 2)
{
   dd = dz * dz;
}
```

```
                        //The indexing in this is really tricky. I had to take
                              //a few days to come up with this. CUDA doesn't
                              //allow multiple dimension arrays, so Rmat
                              //3N*3Nx1. The best way to understand this
                              //is to put it into Excel and work through
                              //it by hand.
                        Rmat[3 * n*(jloop+3*j)+iloop+3*i]=dd/(8*h);


                     }// End jloop
                  } // End iloop
               } // End if h < 0.125
            } // End r2 if
         }//End if particle[tid] !=-1
      }//End if tid < (al * n)
   } // End hydro_2


   //This calculates the diagonal component of Rmat.
   __global__ void diag(double *Rmat, int n)
   {
      int tx = threadIdx.x;
      int ty = threadIdx.y;
      int bx = blockIdx.x;
      int i;
      int index;
      double sum = 0;

      //This takes advantage of CUDA's ability to index in
      //multiple dimensions. tx is the thread in the "x"
      //direction of the matrix and ty is the thread in the
      //"y" direction. This looks at a specific part of each
      //submatrice in Rij.
      if ((tx < 3) && (ty < 3))
      {
         //Loop over the n submatrice in Rij.
         for (i = 0; i < n; i++)
         {
            sum += Rmat[3 * n*(ty + 3 * i) + 3 * bx + tx];
         }

         if (tx == ty)
         {
            index = 3 * n*(ty + 3 * bx) + 3 * bx + tx;
            Rmat[3 * n*(ty + 3 * bx) + 3 * bx + tx] = 1.000 + sum;
```

```
      }
      else
      {
         Rmat[3 * n*(ty + 3 * bx) + 3 * bx + tx] = sum;
      }


   }
}

//The next three kernels put F, U, and Rmat into CSR notation
//so that it is compatible with cuSolve
___global___ void row_el_calc(double *Rmat, double Rmat_tol,
                              int *row_el, int n)
{
   int i, sum, stride;
   int tid = threadIdx.x + blockDim.x*blockIdx.x;

   if (tid < 3 * n)
   {
      stride = 3 * n;
      sum = 0;

      for (i = 0; i < (3 * n); i++)
      {
         if (fabs(Rmat[tid + i * stride]) > Rmat_tol)
            sum++;

      } // End i loop

      row_el[tid] = sum;

   } // End if tid < 3n


} // End row_el_calc

//The purpose of the kernel is to put the resistance matrix
//into compressed sparse row notation. See the cuSPARSE
//programming guide for further explanation
___global___ void csr_create(double *val_r, double *Rmat,
                             double Rmat_tol, int *col_dx,
                             int *row_dx, int *row_el, int n)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int idx, num_col_el, column_idx, end, stride, i;

   if (tid < 3 * n)
   {
      row_dx[tid + 1] = row_el[tid];

      stride     = 3 * n;
      //column_idx   = row_dx[tid];
      end        = row_dx[tid + 1];
      num_col_el = row_dx[tid + 1] - row_dx[tid];

      if (tid == 0)
         column_idx = 0;
      else
         column_idx = row_el[tid - 1];


      for (idx = 0; idx < 3 * n; idx++)
      {

         if (fabs(Rmat[tid + idx * stride]) > Rmat_tol)
         {
            col_dx[column_idx] = idx;
            val_r [column_idx] = Rmat[tid + idx * stride];
            //col_i++;


            column_idx++;


         } // End Rmat if

         if (column_idx == end)
            break;
      }// End idx for loop
   } //End if tid < 3n
}//End csr_create

//This splits up the velocity so that the position can be updated easily.
___global___ void split(double *velocity_out, double *vx, double *vy,
                        double *vz, int n)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int part_num;
```

```
    if (tid < (3 * n))
    {
        part_num = tid / 3;

        if (tid % 3 == 0)
            vx[part_num] = velocity_out[tid];
        if (tid % 3 == 1)
            vy[part_num] = velocity_out[tid];
        if (tid % 3 == 2)
            vz[part_num] = velocity_out[tid];

    } // End if tid < 3n

}//End split

//This updates the position of each particle. This is essentially the
//same as the fortran code.
__global__ void update_pos(double *x, double *y, double *z, double *vx,
                           double *vy, double *vz, double Lx, double Ly,
                           double Lz, int n, double gd, double dt,
                           double *dabsx, double *dabsy, double *dabsz)
{
    int tid = threadIdx.x + blockDim.x*blockIdx.x;
    double xmag, ymag, zmag, dx, dy, dz;


    if (tid < n)
    {
        //xmag = fabs(x[tid]);
        //ymag = fabs(y[tid]);

        zmag = fabs(z[tid]);


        dz = vz[tid] * dt;

        if (zmag >= ((0.5 * Lz) - .50 - .05))
        {
            dy = 0;
            if (z[tid] < 0)
                dx = 0;
            else
                dx = gd * Lz * dt;
        }
        else
        {
            dx = (vx[tid] + gd * (z[tid] + 0.5 * Lz)) *dt;
            dy = (vy[tid]) * dt;
        }

        x[tid] += dx;
        y[tid] += dy;
        z[tid] += dz;

        dabsx[tid] += dx;
        dabsy[tid] += dy;
        dabsz[tid] += dz;

        xmag = fabs(x[tid]);
        ymag = fabs(y[tid]);

        if (xmag > (0.5 * Lx))
            x[tid] *= (1 - Lx / xmag);
        if (ymag > (0.5 * Ly))
            y[tid] *= (1 - Ly / ymag);


    }
}

//Calculates the stress in the fluid.
__global__ void tau_calc(double *z, double *fxtot, double *tau, int n,
                         double Lz)
{
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    int i;

    if (tid == 0)
    {
        *tau = 0;
        for (i = 0; i < n; i++)
        {
            if ((z[i]) >((Lz / 2.0) - 0.5 - 0.01))
            {
                *tau += fxtot[i];
                //*tau += z[i]*(fxtot[i]);
            }
        }
    }
}
```

```
//Used for debugging.
___global___ void printer(double *Rmat, int *row_dx, int *col_dx,
                          double *val_r, double *force_tot, int n,
                          double *velocity_out)
{
   int tid = threadIdx.x + blockDim.x*blockIdx.x;
   int i;

   if (tid == 0)
   {
      //for (i = 0; i < (3 * 3 * n * n); i++)
         //printf("Rmat[%d] = %lf\n", i, Rmat[i]);

//     for (i = 0; i < (3 * n + 1); i++)
   //     printf("row_dx[%d] = %d\n", i, row_dx[i]);

      //for (i = 0; i < 90; i++)
         //printf("col_dx[%d] = %d\n", i, col_dx[i]);

      //for (i = 0; i < 207; i++)
         //printf("val_r[%d] = %lf\n", i, val_r[i]);

      //for (i = 0; i < (3 * n); i++)
         //printf("force_tot[%d] = %lf\n", i, force_tot[i]);

      //for (i = 0; i < (3 * n); i++)
         //printf("velocity[%d] = %lf\n", i, velocity_out[i]);

   }

}


void nlist(double * /*xd*/, double * /*yd*/, double * /*zd*/,
int */*neighbor*/, int */*particle*/, int */*key*/,
double /*rl2*/, int /*n*/, double /*Lx*/, double /*Ly*/,
double /*Lz*/, int /*al*/, thrust::device_ptr<int> /*key_thrust*/);

void hydro_create(double * /*xd*/, double * /*yd*/, double * /*zd*/,
int */*neighbor*/, int */*particle*/, double * /*Rmat*/,
double /*Rmat_tol*/, int */*row_el*/, int */*row_dx*/, int */*col_dx*/,
double * /*val_r*/, double * /*d*/, int /*al*/, int /*n*/,
double /*rc2*/, double /*corr_fac*/, double /*Lx*/, double /*Ly*/,
double /*Lz*/, dim3 /*dimBlock*/, thrust::device_ptr<int>
/*row_thrust*/);



int main(void)
{
   int *key=NULL, *particle=NULL, *neighbor=NULL, *mag_key=NULL;
   int *mag_keyd = NULL, *neighbor_h = NULL, *particle_h = NULL;
    int     *sing      = NULL, *nz = NULL, *nz_d = NULL;
   int     *row_el = NULL, *row_dx = NULL, *col_dx = NULL;
   int      n, kstart, nsteps, nprint, lsteps, i, k, index, nz_s, m, iii, jjj;
   double *xd         = NULL, *yd         = NULL, *zd      = NULL;
   double *x          = NULL, *y          = NULL, *z       = NULL;
   double *fx         = NULL, *fy         = NULL, *fz      = NULL;
   double *fxtot      = NULL, *fytot      = NULL, *fztot   = NULL;
   double *fxpar_h    = NULL, *fypar_h    = NULL, *fzpar_h = NULL;
   double *Rmat       = NULL, *d          = NULL, *val_r   = NULL;
   double *vx         = NULL, *vy         = NULL, *vz      = NULL;
   double *force_tot = NULL, *velocity_out = NULL;
   double  Lx, Ly, Lz, dt, rc, rch, corr_fac, Rmat_tol;
   double  gd, rl, sphere, rc2, rl2;
   double *tau_h = NULL, *tau_d = NULL;
   double *rmat_h = NULL;
   double  ctime_tot = 0, ctime_avg, timetot;
   double  omega, gamma0, time, gamma, g, dxwall;
   int     timei;
   int     cuda_count = 0;
   int     al, file_select;
   float   elapsedTime;
   double *dabsx_h = NULL, *dabsy_h = NULL, *dabsz_h = NULL;
   double *dabsx_d = NULL, *dabsy_d = NULL, *dabsz_d = NULL;
   FILE  *pos_input,*par_input,*pos_output,*time_out,*tau_out,*dabs_out;
    FILE  *force_out, *r_mat;
   char    parameters[] = "parameters.txt";
   char    tt[80], p_out[100], f_out[100], p_in[100], d_out[100],
            t_out[100], r_out[100];
   clock_t t0, t1;
   dim3 dimBlock(3, 3);


    file_select = 0;



// sprintf(p_out, "position_out%d.txt", file_select);
```

```
// pos_output = fopen(p_out, "w");


   //Read in the input files and declare the output files
   sprintf(p_in, "position%05d_mono.txt", file_select);
   pos_input = fopen(p_in, "r");

   sprintf(d_out, "dabs_out%d.txt", file_select);
   dabs_out = fopen(d_out, "w");

   sprintf(f_out, "force_pair_out%d.txt", file_select);
   force_out = fopen(f_out, "w");

   sprintf(t_out, "tau_out%d.txt", file_select);
   tau_out = fopen(t_out, "w");

   //Read in the parameters. Parameter descriptions in the file "parameters".
   par_input = fopen(parameters, "r");
   fscanf(par_input, "%lf", &Lx);       fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &Ly);       fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &Lz);       fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &n);         fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &kstart);    fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &nsteps);    fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &nprint);    fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &dt);       fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &rc);       fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &rch);      fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &corr_fac); fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &Rmat_tol); fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &gd);       fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &omega);    fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &gamma0);   fgets(tt, 80, par_input);
   fscanf(par_input, "%lf", &rl);       fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &lsteps);    fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &al);        fgets(tt, 80, par_input);
   fscanf(par_input, "%d", &timei);     fgets(tt, 80, par_input);
   fclose(par_input);

   //Create rc2 and rl2 so r*r does not need to be performed at
    //every iteration
   rc2 = rc*rc;
   rl2 = rl*rl;

   m   = 3 * n;


   //Allocate vectors on the host
   x       = (double *)malloc(n *       sizeof(double));
   y       = (double *)malloc(n *       sizeof(double));
   z       = (double *)malloc(n *       sizeof(double));

   rmat_h = (double *)malloc(3 * n * 3 * n  * sizeof(double));

   dabsx_h = (double *)malloc(n *       sizeof(double));
   dabsy_h = (double *)malloc(n *       sizeof(double));
   dabsz_h = (double *)malloc(n *       sizeof(double));

   fxpar_h = (double *)malloc(al * n *  sizeof(double));
   fypar_h = (double *)malloc(al * n *  sizeof(double));
   fzpar_h = (double *)malloc(al * n *  sizeof(double));

   neighbor_h = (int    *)malloc(al * n  * sizeof(int));
   particle_h = (int    *)malloc(al * n  * sizeof(int));

   sing       = (int    *)malloc(          sizeof(int));
   nz         = (int    *)malloc(          sizeof(int));

   tau_h = (double *)malloc(sizeof(double));
   cudaMalloc((void**)&tau_d, sizeof(double));
   *tau_h = 0;

   mag_key = (int*)malloc(n * sizeof(int));


   //Allocate vectors on the graphics card
   cudaMalloc((void**)&xd     ,  n  *          sizeof(double));
   cudaMalloc((void**)&yd     ,  n  *          sizeof(double));
   cudaMalloc((void**)&zd     ,  n  *          sizeof(double));

   cudaMalloc((void**)&dabsx_d , n *           sizeof(double));
   cudaMalloc((void**)&dabsy_d , n *           sizeof(double));
   cudaMalloc((void**)&dabsz_d , n *           sizeof(double));

   cudaMalloc((void**)&fxtot  ,  al * n *      sizeof(double));
   cudaMalloc((void**)&fytot  ,  al * n *      sizeof(double));
   cudaMalloc((void**)&fztot  ,  al * n *      sizeof(double));


   cudaMalloc((void**)&fx     ,  al * n *      sizeof(double));
   cudaMalloc((void**)&fy     ,  al * n *      sizeof(double));
```

```
cudaMalloc((void**)&fz      ,   al * n *          sizeof(double));

cudaMalloc((void**)&vx      ,        n *          sizeof(double));
cudaMalloc((void**)&vy      ,        n *          sizeof(double));
cudaMalloc((void**)&vz      ,        n *          sizeof(double));


cudaMalloc((void**)&key     ,   n  *              sizeof(    int));
cudaMalloc((void**)&particle,  al * n *          sizeof(    int));
cudaMalloc((void**)&neighbor, al * n *           sizeof(    int));
cudaMalloc((void**)&mag_keyd, n *                sizeof(    int));

cudaMalloc((void **)&Rmat       ,  3  * n * 3 * n * sizeof(double));
cudaMalloc((void **)&row_el     ,  3  * n *         sizeof(    int));
cudaMalloc((void **)&d          ,  3  *             sizeof(double));
cudaMalloc((void **)&row_dx     , (3 * n + 1)   * sizeof(    int));
cudaMalloc((void **)&col_dx     , (3 * n * al)  * sizeof(    int));
cudaMalloc((void **)&val_r      , (3 * n * al)  * sizeof(double));
cudaMalloc((void **)&force_tot  ,  3 * n *          sizeof(double));
cudaMalloc((void **)&velocity_out, 3 * n *          sizeof(double));
cudaMalloc((void **)&nz_d       ,                   sizeof(int));


//This establishes the variables needed to run cuSolver
cusolverSpHandle_t cusolverhandle = 0;
cusolverStatus_t cusolverStatus;
cudaStream_t solver_stream = 0;

cusparseMatDescr_t descrip_A = 0;
cusparseHandle_t cusparseHandle = 0;
cusparseStatus_t cusparseStatus;
cusparseStatus = cusparseCreateMatDescr(&descrip_A);
cusparseSetMatType(descrip_A,
                   CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase(descrip_A,
                   CUSPARSE_INDEX_BASE_ZERO);

cusolverStatus = cusolverSpCreate(&cusolverhandle);
cusolverSpSetStream(cusolverhandle, solver_stream);



//Read in the initial positions of the suspension
//Also establish the the initial absolute positions at zero
fgets(tt, 80, pos_input);

for (i = 0; i < n; i++)
{
    fscanf(pos_input, "%lf", &sphere);
    fscanf(pos_input, "%lf", &(x[i]));
    fscanf(pos_input, "%lf", &(y[i]));
    fscanf(pos_input, "%lf", &(z[i]));
    fscanf(pos_input, "%d" , &(mag_key[i]));
    dabsx_h[i] = 0;
    dabsy_h[i] = 0;
    dabsz_h[i] = 0;
    fxpar_h[i] = 0;
    fypar_h[i] = 0;
    fzpar_h[i] = 0;


}
fclose(pos_input);

//Copy the position and mag_key vectors to the device from the host
cudaMemcpy(xd , x , n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(yd , y , n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(zd , z , n*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(mag_keyd,mag_key,n*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(dabsx_d,dabsx_h,n*sizeof(double),
           cudaMemcpyHostToDevice);
cudaMemcpy(dabsy_d,dabsy_h,n*sizeof(double),
           cudaMemcpyHostToDevice);
cudaMemcpy(dabsz_d,dabsy_h,n*sizeof(double),
           cudaMemcpyHostToDevice);

// Initiate the thrust vectors
thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>( n);
thrust::device_ptr<int> row_thrust = thrust::device_malloc<int>(3 * n);

//Call the neighbor list
nlist(xd, yd, zd, neighbor, particle, key, rl2, n, Lx, Ly, Lz, al,
      key_thrust);

//Create the timing events associated with CUDA
cudaEvent_t startEvent, stopEvent;
cudaEventCreate(&startEvent);
cudaEventCreate(&stopEvent);

t0 = clock();
```

```
for (k = kstart; k < (nsteps + 1); k++)
{
    time = dt * (double)k;
    //timeold = dt*(double)(k-1);
    //gamma = gamma0*sin(omega*time);
    //gammaold = gamma0*sin(omega*timeold);
    //dgamma = gamma - gammaold;
    //gd = dgamma/dt;
    //dxwall = dgamma*Lz;

    //Calls each kernel and neighborlist

    if ((k%lsteps) == 0)
        nlist (xd, yd, zd, neighbor, particle , key, rl2 , n, Lx, Ly, Lz,
                al , key_thrust);

    init_force<<<(BLOCK_SIZE+3*n*3*n)/BLOCK_SIZE,
                    BLOCK_SIZE>>>(fx, fy, fz, n,
                    fxtot , fytot , fztot , al , tau_d,
                    Rmat, row_dx, row_el, row_dx, val_r,
                    velocity_out, vx, vy, vz, force_tot);
    cudaThreadSynchronize();


    dxwall = gd * Lz * time;
        cudaThreadSynchronize();
    //cudaEventRecord(startEvent, 0);

    force_calc<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
                    BLOCK_SIZE >> >(xd, yd, zd, fx,
                    fy , fz , n, rc2 , Lx, Ly, Lz,
                    neighbor, particle , mag_keyd, rc,
                    fxtot , fytot , fztot , al );
        cudaThreadSynchronize();
    force_total<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
                     BLOCK_SIZE>>>(zd, fx, fy, fz,
                     fxtot , fytot , fztot , particle ,
                     mag_keyd, Lz, rc, n, neighbor, al );
        cudaThreadSynchronize();
    force_wall<<<(BLOCK_SIZE+n)/BLOCK_SIZE,
                    BLOCK_SIZE >> >(zd, Lz, fztot, n, rc,
                    mag_keyd, fxtot, fytot, tau_d);
        cudaThreadSynchronize();
    //printer<<<(BLOCK_SIZE+3*n*3*n)/BLOCK_SIZE,
      //BLOCK_SIZE>>>(Rmat, row_dx, col_dx, val_r,
      //force_tot, n, velocity_out);

    hydro_create(xd, yd, zd, neighbor, particle , Rmat, Rmat_tol, row_el,
      row_dx, col_dx, val_r, d, al, n, rc2, corr_fac, Lx, Ly, Lz, dimBlock,
      row_thrust);
       cudaThreadSynchronize();

    force_combine<<<(BLOCK_SIZE+3*n)/BLOCK_SIZE,
                          BLOCK_SIZE>>>(force_tot, fxtot,
                          fytot , fztot , n, row_dx, nz_d, val_r);


    //printer << <(BLOCK_SIZE + 3 * n * 3 * n) / BLOCK_SIZE,
      //BLOCK_SIZE >> >(Rmat, row_dx, col_dx, val_r,
      //force_tot, n, velocity_out);
    cudaThreadSynchronize();

    cudaMemcpy(nz, nz_d, sizeof(int), cudaMemcpyDeviceToHost);

    nz_s = *nz;

    //Call the solver from CUDA. See cuSOLVER literature for more details
    cusolverSpDcsrlsvchol(cusolverhandle,m,nz_s,descrip_A,val_r,row_dx,
      col_dx,force_tot,0.000001,0,velocity_out,sing );

    //printer << <(BLOCK_SIZE + 3 * n * 3 * n) / BLOCK_SIZE,
      //BLOCK_SIZE >> >(Rmat, row_dx, col_dx, val_r, force_tot,
      //n, velocity_out);
    cudaThreadSynchronize();

    split <<<(BLOCK_SIZE+3*n)/BLOCK_SIZE,BLOCK_SIZE>>>
      (velocity_out, vx, vy, vz, n);
    cudaThreadSynchronize();

    update_pos<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                    (xd, yd, zd, vx, vy, vz, Lx, Ly, Lz,
                     n, gd, dt, dabsx_d, dabsy_d, dabsz_d);
    cudaThreadSynchronize();

    //printf("k = % d\n", k);

//     cudaEventRecord(stopEvent, 0);
   // cudaEventSynchronize(stopEvent);
```

```
//cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);

//ctime_tot += elapsedTime;
//cuda_count++;

//Prints out the results at each desired time step.
if (k % nprint == 0)
{
    printf("%d\n", k);

    sprintf(p_out, "position_out%05d.txt", (k / nprint));
    pos_output = fopen(p_out, "w");

    fprintf(pos_output, "   %d \n\n", k);
    fprintf(dabs_out, "\n %d \n\n", k);
    fprintf(force_out, "\n %d \n\n", k);

    //Copy data to the host from the device.
    cudaMemcpy(x,xd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(y,yd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(z,zd,n*sizeof(double),cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(fxpar_h,fx,al*n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(fypar_h,fy,al*n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(fzpar_h,fz,al*n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

    cudaMemcpy(dabsx_h,dabsx_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsy_h,dabsy_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();
    cudaMemcpy(dabsz_h,dabsz_d,n*sizeof(double),
                    cudaMemcpyDeviceToHost);
    cudaThreadSynchronize();

        cudaMemcpy(particle_h,particle,al*n*sizeof(int),
                        cudaMemcpyDeviceToHost);
        cudaThreadSynchronize();
        cudaMemcpy(neighbor_h, neighbor, al * n * sizeof(int),
                        cudaMemcpyDeviceToHost);
        cudaThreadSynchronize();

//      cudaMemcpy(rmat_h, Rmat, 3 * n * 3 * n * sizeof(double),
                        cudaMemcpyDeviceToHost);
    //  cudaThreadSynchronize();


    // sprintf(r_out, "RESISTANCE_MATRIX%05d_mono.txt", k);
        //r_mat = fopen(r_out, "w");

    //  for (iii = 0; iii < (3 * n); iii++)
    // {
        // for (jjj = 0; jjj < (3 * n); jjj++)
            //{
                // fprintf(r_mat, "% lf, ", rmat_h[iii + 3 * n*jjj]);
    //      }
        //  fprintf(r_mat,"\n");
    //  }
        //fclose(r_mat);



        *tau_h = 0;

        tau_calc<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
                        (zd, fxtot, tau_d, n, Lz);
        cudaMemcpy(tau_h, tau_d, sizeof(double),
                        cudaMemcpyDeviceToHost);

        *tau_h = (-*tau_h);// ((Lx*Ly)*(Lz-1.0));

        g = *tau_h;//gamma0;
        gamma = gd * k * dt;

        fprintf(tau_out, "% lf          % lf          % lf\n", time, gamma, g);

        //Print out the interparticle forces. This file can then be used
            //in the configuration creator to create visualizations.
        for (index = 0; index < al * n; index++)
```

```
        {
            if (index < n)
            {
                fprintf(pos_output, "        % d              ", index);
                fprintf(pos_output, "        % lf               ", x[index]);
                fprintf(pos_output, "        % lf               ", y[index]);
                fprintf(pos_output, "        % lf               ", z[index]);
                fprintf(pos_output, "        % d              \n", mag_key[index]);

                fprintf(dabs_out, "        % 17.16lf          ", dabsx_h[index]);
                fprintf(dabs_out, "        % 17.16lf          ", dabsy_h[index]);
                fprintf(dabs_out, "        % 17.16lf          \n", dabsz_h[index]);
            }

            fprintf(force_out, "        % d              ", particle_h[index]);
            fprintf(force_out, "        % d              ", neighbor_h[index]);
            fprintf(force_out, "        % 17.16lf          ", fxpar_h[index]);
            fprintf(force_out, "        % 17.16lf          ", fypar_h[index]);
            fprintf(force_out, "        % 17.16lf          \n", fzpar_h[index]);

            if (particle_h[index] == -1)
               break;


        }//End of index loop for printing

        fclose(pos_output);
    }//End of print if statement
} //End of integration loop

t1 = clock();
timetot = ((double)(t1 - t0) / (double)CLOCKS_PER_SEC) / 60;

time_out = fopen("time.txt", "w");


//ctime_avg = ctime_tot / ((double)cuda_count);

fprintf(time_out, "Total simulation time = %lf minutes \n", timetot);
//fprintf(time_out, "The time needed to perform the calculations is %lf
 //ms\n", ctime_avg);


thrust::device_free(key_thrust);
thrust::device_free(row_thrust);
```

```
fclose(tau_out);
fclose(time_out);
//fclose(pos_output);
fclose(dabs_out);
fclose(force_out);
cudaFree(dabsx_d);
cudaFree(dabsy_d);
cudaFree(dabsz_d);
free(dabsx_h);
free(dabsy_h);
free(dabsz_h);
fclose(tau_out);
cudaFree(xd);
cudaFree(yd);
cudaFree(zd);
cudaFree(mag_keyd);
cudaFree(neighbor);
cudaFree(particle);
cudaFree(key);
cudaFree(fx);
cudaFree(fy);
cudaFree(fz);
cudaFree(fxtot);
cudaFree(fytot);
cudaFree(fztot);
cudaFree(Rmat);
cudaFree(d);
cudaFree(row_el);
cudaFree(row_dx);
cudaFree(col_dx);
cudaFree(val_r);
cudaFree(vx);
cudaFree(vy);
cudaFree(vz);
cudaFree(velocity_out);
cudaFree(nz_d);
cudaFree(sing);
cudaFree(force_tot);
free(nz);
free(x);
free(y);
free(z);
free(mag_key);
free(neighbor_h);
```

```
    free(particle_h);
    free(fxpar_h);
    free(fypar_h);
    free(fzpar_h);
    free(tau_h);
    cudaFree(tau_d);

}

//Function which builds the neighbor list. There are kernel calls in this
//function to make coding the neighbor list more concise.
void nlist(double *xd, double *yd, double *zd, int *neighbor,
           int *particle, int *key, double rl2, int n,
           double Lx, double Ly, double Lz, int al,
           thrust::device_ptr<int> key_thrust)
{
    keyzero<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
              BLOCK_SIZE>>>(key,neighbor, particle, n, al);

    cudaThreadSynchronize();

    check<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
          (xd, yd, zd, key, neighbor, particle,
           rl2, n, Lx, Ly, Lz);

    cudaThreadSynchronize();
    //thrust::device_ptr<int> key_thrust(key);
    //thrust::device_ptr<int> key_thrust = thrust::device_malloc<int>(n);
    key_thrust = thrust::device_pointer_cast(key);

    thrust::inclusive_scan(key_thrust, key_thrust + n, key_thrust);
    //thrust::device_free(key_thrust);

    cudaThreadSynchronize();

    setup<<<(BLOCK_SIZE+n)/BLOCK_SIZE,BLOCK_SIZE>>>
          (xd, yd, zd, key, particle, n);

    cudaThreadSynchronize();

    populate<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
               BLOCK_SIZE>>>
               (xd, yd, zd, neighbor, particle,
                n, rl2, Lx, Ly, Lz, al);

    cudaThreadSynchronize();

    // thrust::device_free(key_thrust);

}

//This builds the resistance matrix and puts it into CSR notation.
void hydro_create(double *xd, double *yd, double *zd, int *neighbor, int
*particle, double *Rmat, double Rmat_tol, int *row_el, int *row_dx,
 int *col_dx, double *val_r, double *d, int al, int n, double rc2,
 double corr_fac, double Lx, double Ly, double Lz, dim3 dimBlock,
 thrust::device_ptr<int> row_thrust)
{
    //Builds the resistance and puts it in CSR notation.
    hydro_2<<<(BLOCK_SIZE+al*n)/BLOCK_SIZE,
              BLOCK_SIZE>>>
              (particle, neighbor, xd, yd, zd, Rmat,
               d, al, n, rc2, corr_fac, Lx, Ly, Lz);
       cudaThreadSynchronize();

    diag << < n, dimBlock >> >(Rmat, n);
       cudaThreadSynchronize();
    row_el_calc<<<(BLOCK_SIZE+3*n)/BLOCK_SIZE,BLOCK_SIZE>>>
                (Rmat, Rmat_tol, row_el, n);
    cudaThreadSynchronize();

    row_thrust = thrust::device_pointer_cast(row_el);
    thrust::inclusive_scan(row_thrust, row_thrust + 3 * n, row_thrust);
    cudaThreadSynchronize();

    csr_create<<<(BLOCK_SIZE+3*n)/BLOCK_SIZE,BLOCK_SIZE>>>
              (val_r, Rmat, Rmat_tol, col_dx, row_dx,
               row_el, n);
    cudaThreadSynchronize();
    //printer<<<(BLOCK_SIZE+3*n*3*n)/BLOCK_SIZE,
     //BLOCK_SIZE>>>(Rmat, row_dx, col_dx, val_r, n);


} // End hydro_create
```

A sample parameter input file for the code *mix_hydro.cu* would look like:

```
10.0      Lx
5.0       Ly
5.0       Lz
150       Number of spheres
1         Start
50000000  Number of steps
100000    Print Statements
0.00001   dt
2.5       Cutoff radius
0.125     Hydrodynamic Cutoff Radius
0.0001    h correction factor
0.00001   Resistance Matrix Tolerance
0.01      Dimensionless shear
.01       Frequency, omega
0.00001   Strain amplitude, gamma0
2.7       Cutoff radius for neighbor list
100       Number of steps to calculate neighbor list
70        Neighbor list length
0         Initial time
```

Bibliography

[1] Ahn, K. and D.J. Klingenberg (1994). Relaxation of polydisperse electrorheological suspensions. *J. Rheol.*, 38(3):713–741.

[2] Aktary, M., M.T. McDermott, and J. Torkelson (2001). Morphological evolution of films formed from thermooxidative decomposition of zddp. *Wear*, 247(2):172–179.

[3] Allen, M. and D.J. Tildesley (1989). *Computer simulation of liquids*. Oxford university press.

[4] Anderson, R.A. (1994). Electrostatic forces in an ideal spherical-particle electrorheological fluid. *Langmuir*, 10(9):2917–2928.

[5] Ball, R. and J.R. Melrose (1997). A simulation technique for many spheres in quasi-static motion under frame-invariant pair drag and brownian forces. *Phys. A*, 247(1):444–472.

[6] Batchelor, G.K. (1970). The stress system in a suspension of force-free particles. *J. Fluid Mech.*, 41(03):545–570.

[7] Bonnecaze, R. and J.F. Brady (1992). Dynamic simulation of an electrorheological fluid. *J. Chem. Phys.*, 96(3):2183–2202.

[8] Bossis, G. and J.F. Brady (1984). Dynamic simulation of sheared suspensions. I. General method. *J. Chem. Phys.*, 80(10):5141–5154.

[9] Brady, J. and G. Bossis (1988). Stokesian dynamics. *Annu. Rev. Fluid Mech.*, 20:111–157.

[10] Carlson, J. and J.L. Sproston (2000). Controllable fluids in 2000-status of ER and MR fluid technology. In *Actuator 2000-7th International Conference on New Actuators*, pages 126–130.

[11] Cates, M., J.P. Wittmer, J.P. Bouchaud, and P.H. Claudin (1998). Jamming, force chains, and fragile matter. *Phys. Rev. Lett.*, 81(9):1841–1844.

[12] Cho, M.S., S.T. Lim, I.B. Jang, H.J. Choi, M.S. Jhon (2004). Encapsulation of spherical iron-particle with pmma and its magnetorheological particles. *IEEE Trans. Magn.*, 40(4):3036–3038.

[13] Choi, H., I.B. Jang, J.Y. Lee, A. Pich, S. Bhattacharya, and H-J Adler (2005). Magnetorheology of synthesized core-shell structured nanoparticle. *IEEE Trans. Magn.*, 41(10):3448–3450.

[14] Corbett, B. and Visnic, W. (2000). 'riding the magnetic wave. *Ward's Auto World*, 36:49.

[15] Deen, W. M. (1998). *Analysis of transport phenomena (topics in chemical engineering)*, volume 3. Oxford University Press, New York.

[16] Ewoldt, R. H. (2013). Defining nonlinear rheological material functions for oscillatory shear. *Journal of Rheology (1978-present)*, 57(1):177–195.

[17] Fang, F. and H.J. Choi (2008). Noncovalent self-assembly of carbon nanotube wrapped carbonyl iron particles and their magnetorheology. *J. Appl. Phys.*, 103(7):07A301.

[18] Farr, R., J.R. Melrose, and R.C. Ball (1997). Kinetic theory of jamming in hard-sphere startup flows. *Phys. Rev. E*, 55(6):7203–7211.

[19] Flowers, W. C. (1973). *A man-interactive simulator system for above-knee prosthetics studies.* PhD thesis, Massachusetts Institute of Technology.

[20] Foister, R. T. (1997). Microspheres dispersed in liquid, increase in flow resistance. US Patent 5,667,715.

[21] Frenkel, D. and Smit, B. (1987). *Understanding Molecular Simulations*, volume 1 of *Computational Science Series*. Academic Press, 2002 edition.

[22] Garland, M., S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov (2008). Parallel computing experiences with CUDA. *IEEE Micro*, (4):13–27.

[23] Genc, S. and Phulé, P. P. (2002). Rheological properties of magnetorheological fluids. *Smart Materials and Structures*, 11(1):140.

[24] Ginder, J. (1996). Rheology controlled by magnetic fields. *Encycl. Appl. Phys.*, 16:487–503.

[25] Grimes, D., Flowers, W., and Donath, M. (1977). Feasibility of an active control scheme for above knee prostheses. *Journal of Biomechanical Engineering*, 99(4):215–221.

[26] Gulley, G. and R.T. Tao (1993). Static shear stress of electrorheological fluids. *Phys. Rev. E*, 48(4):2744.

[27] Henley, S. and F.E. Filisko (1999). Flow profiles of electrorheological suspensions: An alternative model for ER activity. *J. Rheol.*, 43(5):1323–1336.

[28] Herr, H. and Wilkenfeld, A. (2003). User-adaptive control of a magnetorheological prosthetic knee. *Industrial Robot: An International Journal*, 30(1):42–55.

[29] Intel, C. (2015). Intel®high end desktop processors.

[30] Israelachvili, J. N. (2011). *Intermolecular and surface forces: revised third edition*. Academic press.

[31] James, K., Stein, R., Rolf, R., and Tepavac, D. (1990). Active suspension above-knee prosthesis. In *Proceedings of the 6th International Conference on Biomedical Engineering*, pages 6–8.

[32] Johansson, J., D.M. Sherrill, P.O. Riley, P. Bonato, and H. Herr (2005). A clinical comparison of variable-damping and mechanically passive prosthetic knee devices. *American Journal of Physical Medicine & Rehabilitation*, 84(8):563–575.

[33] Jolly, M., J.W. Bender, and J.D. Carlson (1998). Properties and applications of commercial magnetorheological fluids. In *5th Annual International Symposium on Smart Structures and Materials*, pages 262–275. International Society for Optics and Photonics.

[34] Kernighan, B., D.M. Ritchie, and P. Ejeklint (1988). *The C programming language*, volume 2. prentice-Hall Englewood Cliffs.

[35] Kim, S. and Karrila, S. (1991). Microhydrodynamics: Principles and applications.

[36] Kim, S. and S.J. Karrila (2013). *Microhydrodynamics: principles and selected applications*. Courier Corporation.

[37] Kittipoomwong, D., D.J. Klingenberg, and J.C. Ulicny (2005). Dynamic yield stress enhancement in bidisperse magnetorheological fluids. *Journal of Rheology (1978-present)*, 49(6):1521–1538.

[38] Kittipoomwong, D., D.J. Klingenberg, Y.M. Shkel, J.F. Morris, and J.C. Ulicny (2008). Transient behavior of electrorheological fluids in shear flow. *J. Rheol.*, 52(1):225–241.

[39] Kittipoomwong, P. (2007). *Impact of microstructure evolution on the rheology of electro-and magnetorheological suspensions*. PhD thesis.

[40] Klingenberg, D. (2001). Magnetorheology: Applications and challenges. *AIChE J.*, 47(2):246–249.

[41] Klingenberg, D. and C.F. Zukoski (1990). Studies on the steady-shear behavior of electrorheological suspensions. *Langmuir*, 6(1):15–24.

[42] Klingenberg, D., C.H. Olk, M.A. Golden, and J.C. Ulicny (2010). Effects of nonmagnetic interparticle forces on magnetorheological fluids. *J. Phys.: Condens. Matter*, 22(32):324101.

[43] Klingenberg, D., F. Van Swol, and C.F. Zukoski (1989). Dynamic simulation of electrorheological suspensions. *J. Chem. Phys.*, 91(12):7888–7895.

[44] Klingenberg, D., F. Van Swol, and C.F. Zukoski (1991a). The small shear rate response of electrorheological suspensions. I. Simulation in the point–dipole limit. *J. Chem. Phys.*, 94(9):6160–6169.

[45] Klingenberg, D., F. Van Swol, and C.F. Zukoski (1991b). The small shear rate response of electrorheological suspensions. II. Extension beyond the point–dipole limit. *J. Chem. Phys.*, 94(9):6170–6178.

[46] Klingenberg, D. and J.C. Ulicny (2011). Enhancing magnetorheology. *Int. J. Mod. Phys. B*, 25(07):911–917.

[47] Klingenberg, D., J.C. Ulicny, and M.A. Golden (2007). Mason numbers for magnetorheology. *J. Rheol.*, 51(5):883–893.

[48] Kraynik, A., R.T. Bonnecaze, and J.F. Brady (1991). Electrically induced stresses in ER fluids: The role of particle chain structure. *Electrorheological Fluids, Mechanisms, Properties, Structure, Technology, and Applications*, pages 15–16.

[49] Lemaire, E., Meunier, A., Bossis, G., Liu, J., Felt, D., Bashtovoi, P., and Matoussevitch, N. (1995). Influence of the particle size on the rheology of magnetorheological fluids. *Journal of Rheology (1978-present)*, 39(5):1011–1020.

[50] Martin, J. E. (2000). Thermal chain model of electrorheology and magnetorheology. *Phys. Rev. E*, 63(1):011406.

[51] Mazhar, H., T. Heyn, and D. Negrut (2011). A scalable parallel method for large collision detection problems. *Multibody System Dynamics*, 26(1):37–55.

[52] NVIDIA Corporation (2015). *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation.

[53] Parthasarathy, M. (1998). *Dynamic rheology of electrorheological suspensions*. PhD thesis, University of Wisconsin-Madison.

[54] Parthasarathy, M. and D.J. Klingenberg (1995a). A microstructural investigation of the nonlinear response of electrorheological suspensions i. start-up of steady shear flow. *Rheologica acta*, 34(5):417–429.

[55] Parthasarathy, M. and D.J. Klingenberg (1995b). A microstructural investigation of the nonlinear response of electrorheological suspensions ii. oscillatory shear flow. *Rheol. Acta*, 34.

[56] Parthasarathy, M. and Klingenberg, D. J. (1996). Electrorheology: mechanisms and models. *Materials Science and Engineering: R: Reports*, 17(2):57–103.

[57] Press, W., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling (1986). *Numerical Recipes in Fortran 77: the art of scientific computing, ch. 13.10*, volume 1992.

[58] Rabinow, J. (1948). The magnetic fluid clutch. *American Institute of Electrical Engineers, Transactions of the*, 67(2):1308–1315.

[59] Russel, W., D.A. Saville, and W.R. Schowalter (1992). *Colloidal dispersions*. Cambridge university press.

[60] Sakai, Y. (1988). The "ecvt" electro continuously variable transmission. Technical report, SAE Technical Paper.

[61] Sanders, J. and E. Kandrot (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.

[62] Sevick, E., P.A. Monson, and J.M. Ottino (1988). Monte carlo calculations of cluster statistics in continuum models of composite morphology. *J. Chem. Phys.*, 88(2):1198–1206.

[63] Sierou, A. and J.F. Brady (2001). Accelerated stokesian dynamics simulations. *Journal of Fluid Mechanics*, 448:115–146.

[64] Tang, X., X. Zhang, R. Tao, and Y. Rong (2000). Structure-enhanced yield stress of magnetorheological fluids. *J. Appl. Phys.*, 87(5):2634–2638.

[65] Taufer, M., O. Padron, P. Saponaro, and S. Patel (2010). Improving numerical reproducibility and stability in large-scale numerical simulations on gpus. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–9. IEEE.

[66] Trefethen, L.N. and D. Bau III (1997). *Numerical linear algebra*, volume 50. Siam.

[67] Ulicny, J., A.L. Smith, M.A. Golden, B.L. McDermott, and T.J. Chapaton (2004). Magnetorheological fluids with an additive package. US Patent 6,824,701.

[68] Ulicny, J., A.L. Smith, M.A. Golden, and C.A. Hayden (2005a). Magnetorheological fluids with stearate and thiophosphate additives. US Patent 6,881,353.

[69] Ulicny, J., D.J. Klingenberg, A.L. Smith, and M.A. Golden (2007). Magnetorheological fluid device. US Patent 7,306,083.

[70] Ulicny, J., K.S. Snavely, M.A. Golden, and D.J. Klingenberg (2010). Enhancing magnetorheology with nonmagnetizable particles. *Applied Physics Letters*, 96(23):231903.

[71] Ulicny, J., M.A. Golden, C.S. Namuduri, and D.J. Klingenberg (2005b). Transient response of magnetorheological fluids: Shear flow between concentric cylinders. *J. Rheol.*, 49(1):87–104.

[72] Ulicny, J., M.A. Golden, K.S. Snavely, and P. Marur (2013). Magnetorheological compositions including nonmagnetic material. US Patent 8,361,341.

[73] Vieira, S., L.B.P. Neto, and A.C.F. Arruda (2000). Transient behavior of an electrorheological fluid in shear flow mode. *J. Rheol.*, 44(5):1139–1149.

[74] Volkova, O., S. Cutillas, and G. Bossis (1999). Shear banded flows and nematic-to-isotropic transition in er and mr fluids. *Phys. Rev. Lett.*, 82(1):233.

[75] Weiss, K., J.D. Carlson, and D.A. Nixon (2000). Magnetic field responsive fluids containing a field polarizable particle component and a liquid carrier component, used within the working gap of dampers, shock absorbers, clutches, brakes and valves to provide varying damping force. US Patent 6,027,664.