HIGH-PERFORMANCE GPU MODELING OF TRIAXIAL GALAXY CLUSTERS

A thesis submitted to the faculty of
San Francisco State University
In partial fulfillment of
The Requirements for
The Degree

Master of Science
In
Computer Science: Computing for Life Science

by

Tyler Chapman

San Francisco, California

May, 2013

CERTIFICATION OF APPROVAL

I certify that I have read *High-Performance GPU Modeling of Triaxial Galaxy Clusters* by Tyler Chapman, and that in my opinion this work meets the criteria for approving a thesis submitted in partial fulllment of the requirements for the degree: Master of Science in Computer Science: Computing for Life Science at San Francisco State University.

Dragutin Petkovic
Professor and Chair, Computer Science

Andisheh Mahdavi
Assistant Professor, Physics and Astronomy

Bill Hsu
Associate Professor, Computer Science

HIGH-PERFORMANCE GPU MODELING OF TRIAXIAL GALAXY CLUSTERS

Tyler Chapman
San Francisco, California
May, 2013

To accurately model the dark matter within a galaxy cluster we present a simulation for non-symmetric, triaxially shaped galaxy clusters. To overcome the computational difficulties associated with this morphology, we use GPU computing, and the CUDA framework to parallelize the calculations. We treat each computation block on the GPU as a simulated pixel from a CCD image of a galaxy cluster, and each thread within a block calculates an energy level for the spectra of the gas. By combining every thread from all of the GPU blocks we get a complete image of a simulated cluster. Using GPUs rather than computing entirely on the CPU we achieve a 100-fold speed increase in generating the spectra of a simulated cluster.

I certify that the Abstract is a correct representation of the content of this thesis.

_____ _____

Chair, Thesis Committee Date

TABLE OF CONTENTS

List of Figures vi

1 Introduction 1

2 Background and Motivation 9

3 Problem Statement 20

3.1 Contributions 20

4 Relevant Work 22

5 Methodology 25

6 Experimental Evaluations 45

6.1 Simulation Viability 45

6.2 Simulation Verification 52

7 Conclusion 61

8 Future Work 62

9 Appendix A: Computing Environment 63

LIST OF FIGURES

Figure		Page
1	Composite image of Abell 2052	3
2	Diagram of an ellipse	6
3	Diagram of an ellipsoid	6
4	Diagram of a triaxial ellipsoid	7
5	Basic flux spectrum	11
6	Temperature, metallicity, and η_ρ profiles	18
7	Cluster representation as a 3-D grid	26
8	Flow diagram of key steps	27
9	Thread diagram by blocks	29
10	Ratio of runtime for CPU-only integration over GPU integration	49
11	Flux plotted against magnitude of energy channel	51
12	Spectra generated with a constant metallicity and η_ρ	54
13	Spherical spectra generated with metallicity, η_ρ, and temperature as a function of l	55
14	Spectra from spherical and triaxial models	56
15	Triaxial spectra generated with metallicity, η_ρ, and temperature as a function of l	57
16	The total flux taken when θ ranges from $10^o - 180^o$ with $A = 10$ and $B = 1$	59
17	The total flux taken when ψ ranges from $10^o - 180^o$ with $A = 10$ and $B = 1$	60

1 Introduction

One of the most fundamental problems in astrophysics is the nature and distribution of dark matter. Dark matter makes up around 23% of the mass-energy density and 85% of the total matter content of the universe, yet it is only observable from its gravitational effects on visible matter. Because astronomical data comes primarily through observations of luminous objects, indirect methods of inferring the dark matter distribution are needed. As the largest gravitationally-bound objects, galaxy clusters contain a representative fraction of the dark matter content of the universe, and are thus essential for studying the properties of dark matter.

Clusters of galaxies are collections of 50 to 1,000 galaxies, and have masses ranging between 10^{44} and 10^{45} kilograms. This corresponds to 10^{14} to 10^{15} solar masses. Solar mass is a conventional unit of mass in astronomy, and will be used throughout this paper. Only about 1% of the mass is visible through optical observations of the galaxies, while about 9% of the mass in the cluster is contained within the inter-galactic gas and the inter-cluster medium (a mixture of 74% hydrogen, 25% helium, and 1% other elements). The density is extremely low at around 1 particle per 1000 cm^3, whereas air on earth is 10^{22} particles per 1000 cm^3. This mass takes the form of 10 million degree gas, and is invisible to the human eye. At these temperatures the gas is in the form of a plasma, emitting strong X-rays that can be studied through the use of X-ray telescopes like NASA's Chandra X-ray Observatory and the European Space Agency's XMM-Newton. Figure 1

shows a composite image of galaxy cluster Abell 2052[1], a galaxy cluster approximately 480 million light years away, and about 1.29 million light years across. This corresponds to about 147 mega parsecs away, and about 395 kilo parsecs across. Parsecs are a more common unit of distance for astronomy, and will be used henceforth. The blue in the image is the X-ray emission from the hot plasma, and the gold is the optical emission from the galaxies themselves. This particular image shows the effects on the plasma caused by a smaller cluster crashing into a much larger cluster. The plasma within the larger cluster was disrupted by the gravitational attraction of the colliding cluster. In the image it is clear just how little of the matter is contained within the galaxies, and how much is swirling around in the dense, hot plasma.

Even though the total mass of plasma is much more than the visible mass, combined they are still only 10% of the total mass of the cluster. The remaining 90% of the mass inside of a cluster is made up of dark matter, which can only be studied indirectly through gravitational interactions. The existence of dark matter was originally hypothesized to account for the discrepancy between calculations of expected rotational velocities for individual stars within a galaxy, and the observed motion. These stellar velocities cannot be explained by the gravitational effect from the total mass contained within the visible stars and planets. A greater amount of unseen matter must be present, appropriately called dark matter, and its effects are detectable on scales larger than a solar system. Similarly, the rotational velocity of individual galaxies at the edge of a cluster are dramatically faster than can be accounted for by only visible matter in the cluster (Zwicky 1937). Without a

[1] http://chandra.harvard.edu/photo/2011/a2052/

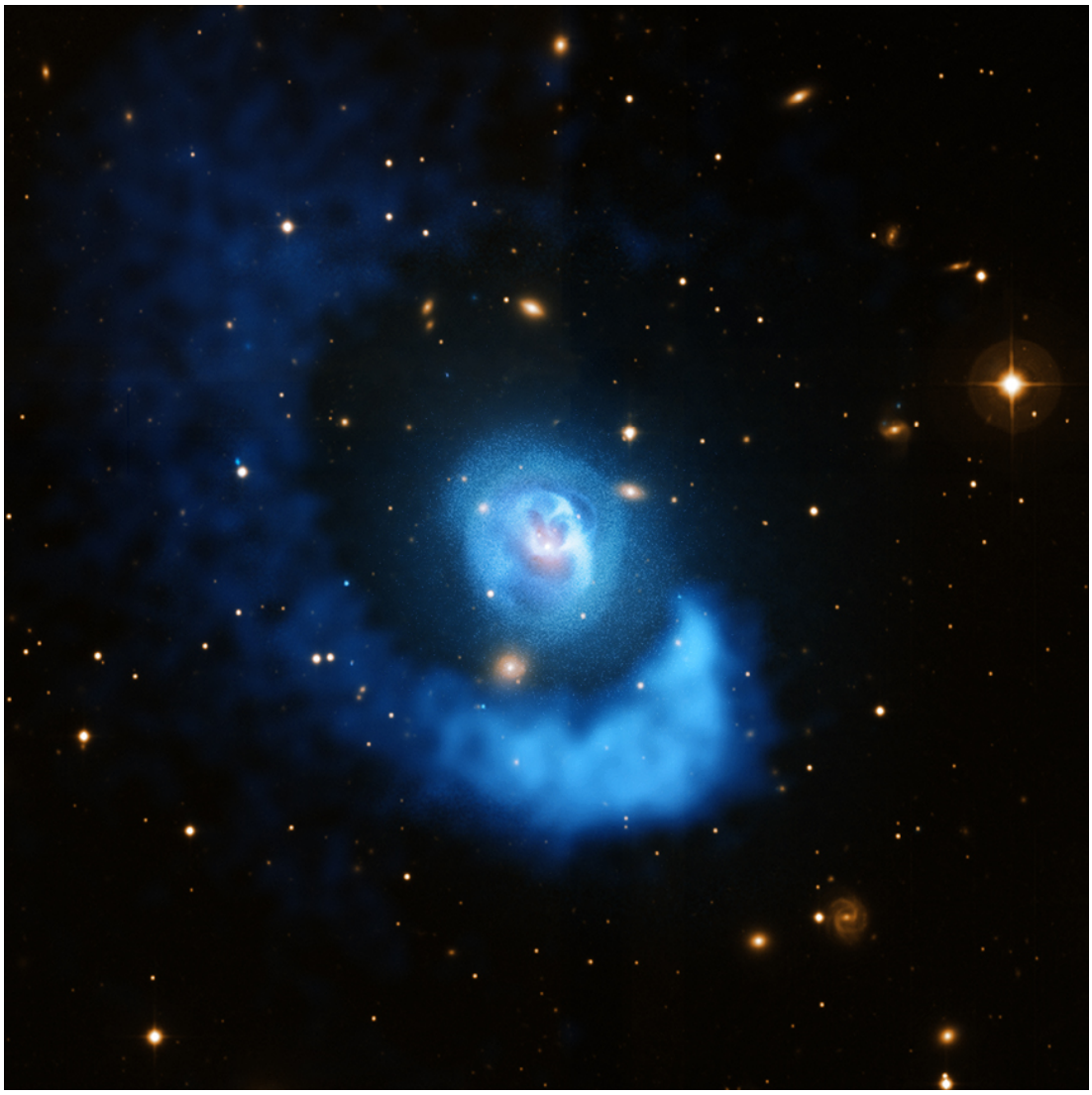

Figure 1: Image of Abell 2052. The X-ray data from Chandra is shown in blue, and the optical data from the Very Large Telescope is shown in gold.

large amount of dark matter the rotational velocity of the edge galaxies would be too great for them to be gravitationally bound to their corresponding cluster, and they would shoot away to other parts of the universe.

While it is difficult to study the properties and behavior of something that is by definition un-seeable, one of the ways to understand the behavior of dark matter within a galaxy cluster is to construct an accurate model of the cluster's gas. Astrophysicists use a combination of gravity and plasma physics to model the spectrum and brightness of the light emitted by the hot, dense gas in the clusters of galaxies. These models can then be used to deduce the dark matter distribution in these gargantuan objects. Initial steps in developing a model take into account the observed data taken from X-ray telescopes that orbit the Earth. From orbit, these telescopes detect the X-rays emitted from the cluster's gas that would otherwise be blocked by the atmosphere of Earth.

The standard approach to modeling cluster atmospheres has been to treat them as spherically symmetric, even though dark matter is thought to be distributed triaxially. A triaxial object is a type of ellipsoid. Ellipsoids, like a football or the Earth, are 3-dimensional closed surfaces that have the property of any cross section being an ellipse or circle. An ellipse is defined as a closed curve consisting of all points whose distances from each of two foci add up to the same value, and the midpoint between the foci is the center. Figure 2 shows the major properties of an ellipse[2]. In Figure 2 the distance to a focus (F) from the center (C) is the eccentricity (e) multiplied by the semi-major axis (a).

[2]http://upload.wikimedia.org/wikipedia/commons/6/65/Ellipse_Properties_of_Directrix_and_String_Construction.svg

e is between zero and one. When e is zero the ellipse becomes a circle, and as e increases the ellipse stretches more and more. Ellipsoids are defined by

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}=1 \tag{1.1}$$

where a, b, and c are the lengths of the semi-principal axes. An ellipsoid that is considered to be triaxial is one that has three distinct semi-principal axes as in Figure 3[3]. With very few exceptions, the standard modeling approach has been to set $a=b=c$ to form a degenerate ellipsoid that is a sphere. Our technique models more closely to observations to allow for each of the semi-principal axes to have a unique positive value. This can create a triaxially shaped galaxy cluster similar to the shape in Figure 4[4].

Assuming spherical symmetry greatly reduces the computational complexity of calculations, but leads to misleading results about the properties of galaxy clusters. Even though for most well-behaved systems the error introduced by assuming spherical symmetry does not affect the total mass determination by more than 15-20%, it can lead to greater errors in determining the distribution of dark matter throughout a given cluster. This distribution is the key quantity that is used to discriminate among various dark matter models. To overcome the computational difficulties of creating a triaxial model we used GPU computing along with NVIDIA's CUDA (Compute Unified Device Architecture)[5] development environment.

[3] http://upload.wikimedia.org/wikipedia/commons/5/50/Ellipsoid_tri-axial_abc.svg
[4] http://upload.wikimedia.org/wikipedia/commons/6/6d/Triaxial_Ellipsoid.jpg
[5] www.nvidia.com/object/cuda_home_new.html

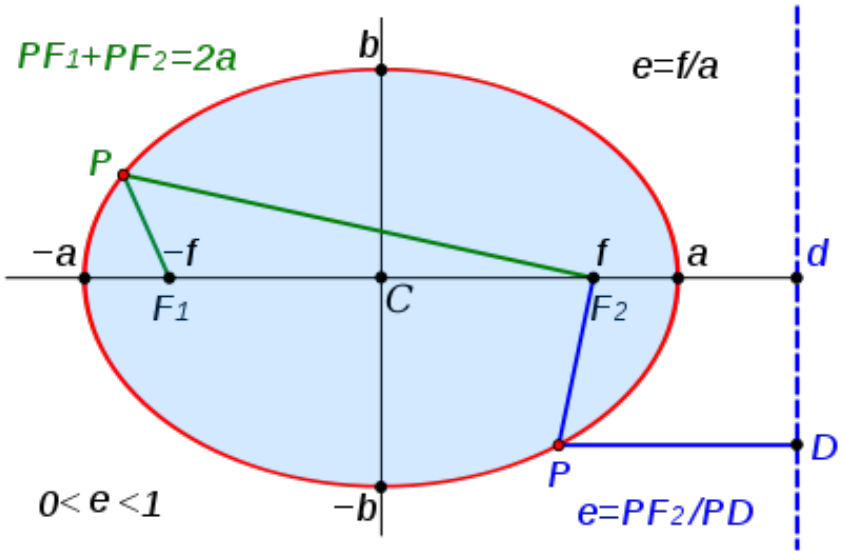

Figure 2: Diagram of the basic properties of an ellipse.

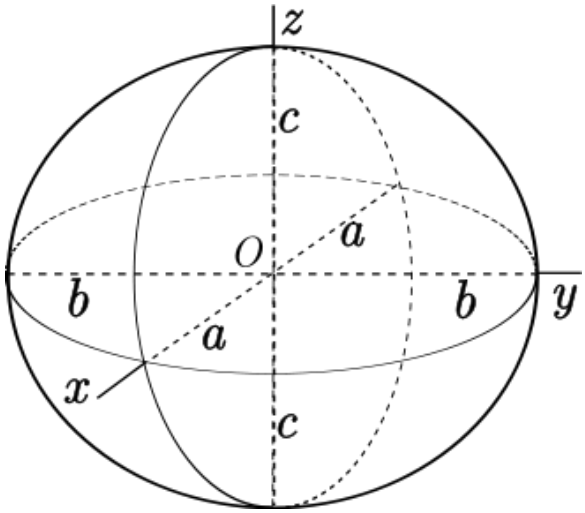

Figure 3: Diagram of an ellipsoid. The semi-principal axes are labelled a, b, and c.

Figure 4: Diagram of a triaxial ellipsoid. The semi-principal axes have the property that $c > b > a$ to give the exaggerated shape.

GPU computing utilizes GPUs (graphics processing units) to perform a tremendous amount of relatively simple computations that would normally be done on the CPU. The work done on the GPU runs dramatically faster, because the computation is broken up into millions of parallelizable threads. This is made possible by the CUDA architecture that was designed by NVIDIA for use on their Fermi Tesla GPUs. The help of GPU computing, and the CUDA architecture, makes it possible to create a triaxial model of the dark matter distribution within a galaxy cluster.

This project can have a large impact on the astronomy community. Not only is the data significant for helping astronomers pin down the structure of dark matter within galaxy clusters, it also demonstrates the benefit of using GPU computing for other complex astronomy problems. Time on supercomputers is limited, but with some programming inge-

nuity many computational astronomy problems can be solved in this manner.

2 Background and Motivation

To create a model of a galaxy cluster an accurate prediction of the spectra must be compared with the spectra taken from actual observations. The characteristics of the X-rays emitted from a cluster of galaxies depend on the density, temperature, and composition of the hot gas. When those X-rays are observed, two of the properties that can be analyzed from the light are "color" and brightness. Just as we see colors from visible light here on Earth, there are also X-ray "colors". Visible light is a section of the electromagnetic spectrum with a frequency ranging from around 400-790 Terahertz (1 THz = 10^{12} Hertz), and the color we see depends on the frequency of the light reaching our eyes. Blue light has a frequency range of 606-668 THz, and red light has a range of 400-484 THz. The higher the energy of light radiation, the higher the frequency, and the bluer the light is perceived. The X-ray spectrum ranges from 30 to 30,000 Petahertz (10^{16} Hertz), and it can be broken up into "colors" in the same manner as visible light. The temperature of the gas can be determined from the "color" of the observed light. The hotter the gas, the higher the frequency of the emitted X-rays, and the "bluer" the radiation becomes. This is observed as a shift to the higher end of the spectrum.

The density of the gas can be determined from the brightness of the observed light. The denser the gas, the greater collision probability of the individual electrons and protons, and the creation of more emitted radiation. The more radiation emitted from the cluster, the brighter the light will be when observed by the telescope. When integrated through the cluster the brightness and "color" of the light are what make up the shape and magnitude

of the continuum of an observed spectrum. This can be seen as the blue line in Figure 5. In this sense the graphed continuum is a plot of the temperature and density of the hot gas in the galaxy cluster.

A third property observed from studying the emitted light is the peaks (or lines) within the spectrum. This is caused by the amount and type of "metals" (elements heavier than helium) the gas contains, and are pointed out in Figure 5 by the red arrows. These features correspond to specific elements in the gas emitting radiation when they interact within the gas. The higher the metallicity of the gas the more features will be observed in the spectrum.

All of the above effects are encapsulated within the cooling function. The cooling function Λ_υ quantifies the temperature and metallicity of the gas in the cluster. The spectrum will be "bluer" if the temperature T is higher, and contain more features if the metallicity Z is higher. Since both metallicity and temperature of the gas are dependent on the position within the cluster T, Z and Λ_υ must be integrated with respect to r to achieve a spectrum similar the one shown in Figure 5.

Mahdavi et al. (2007) gives the process for creating a spherically symmetric cluster model. The spectrum within an annulus with inner and outer radius R_1 and R_2 can be calculated from:

$$L_\upsilon = \int_{R_1}^{R_2} 2\pi R dR \int_{R}^{r_{max}} n_e n_H \Lambda_\upsilon[T(r), Z(r)] \frac{2r}{\sqrt{r^2 - R^2}}\, dr \tag{2.1}$$

r is the unprojected radius, R is the projected radius, r_{max} is the termination radius of the X-

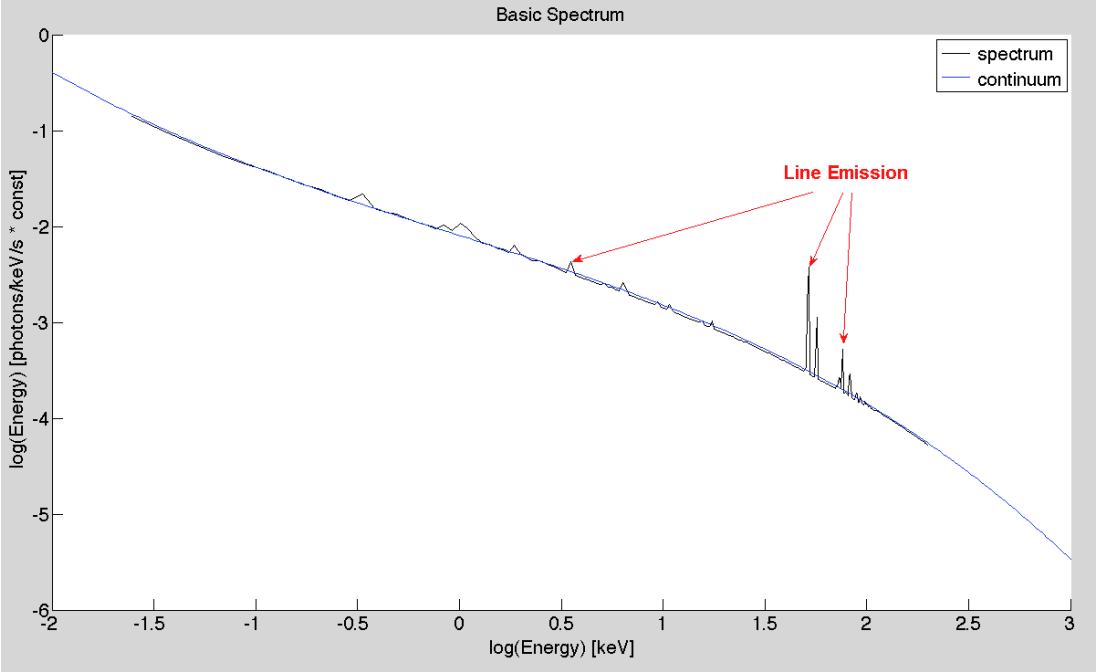

Figure 5: A basic spectrum of the flux integrated through the cluster. The metallicity lines are pointed out with red arrows, and the continuum is shown as a blue line.

ray emitting gas, and Λ_υ is the frequency-dependent cooling function which is a function of temperature T and metallicity Z. n_e is the electron number density which is the amount of electrons within a volume of space. n_H is the ratio of hydrogen gas density to proton mass. $n_e n_H$ is sometimes referred to as emission member density defined from now on as η_ρ. The termination radius can be thought of as the outer boundary of the cluster, which is commonly placed at the point where the effects of surrounding intergalactic medium become important.

A model using Equation 2.1 is solvable with a relatively small amount of computational power. Unfortunately, there is an extensive variety of evidence that galaxy clusters are not spherically symmetric. Specifically, X-ray surface brightness maps (Fabricant et al. 1984; Buote and Canizares 1996; Kawahara 2010), Sunyaev Zeldovich pressure maps (Sayers et al. 2011a), and both strong and weak gravitational lensing (Soucail et al. 1987; Evans and Bridle 2009). To model a cluster that is non-symmetric the spectrum is calculated as:

$$L_\upsilon = \int_{l_a}^{l_b} \eta_\rho(l)\Lambda_\upsilon[T(l),Z(l)]dl \tag{2.2}$$

Where all of the previous functions are parameterized with l defined as:

$$l^2 = x^2 + \frac{y^2}{A^2} + \frac{z^2}{B^2} \tag{2.3}$$

In triaxial coordinates l is equivalent to radius in spherical coordinates. For a surface with constant radius (a sphere) T, Z, and η_ρ are constant, and for a triaxial object they are

constant with respect to constant l. If $A = B$, then spherical symmetry is intact. The values of A and B determine the triaxial shape, and give the necessary freedom to model cluster morphology.

Although this equation appears simpler, the complexity is hidden within the parameterization. The most computationally demanding step for a non-spherical cluster is simultaneously solving for the spectrum at each spot in two-dimensional space and integrating through the third dimension of the cluster. The computational difficulty of the problem can be illustrated by demonstrating the derivation of a simplified temperature profile. The first step is to assume that the gas within the cluster is in hydrostatic equilibrium with the gravitational potential. This means that the thermal pressure from the gas is balanced by the gravitational effects within the cluster, so that the gas is neither collapsing or dissipating away from the cluster. Clusters that are undergoing mergers with other clusters will not satisfy this property, so the model is specific to what are called relaxed clusters. Starting from the triaxial formulation of the Hydrostatic Equilibrium Equation gives:

$$\frac{\partial P}{\partial x}\hat{x} + \frac{\partial P}{\partial y}\hat{y} + \frac{\partial P}{\partial z}\hat{z} = -\rho_g\left(\frac{\partial \varphi}{\partial x}\hat{x} + \frac{\partial \varphi}{\partial y}\hat{y} + \frac{\partial \varphi}{\partial z}\hat{z}\right) \tag{2.4}$$

where the left side of the equation is the force from gas pressure, and the right side is competing force from the gravitational potential. In order to solve for a triaxial system it can be parameterized with l and rewritten as:

$$\frac{\partial P}{\partial l} = -\rho_g \frac{\partial \varphi}{\partial l} \tag{2.5}$$

In general the cluster will not be viewed along a line of sight that corresponds to one of the ellipsoid axes. The cluster will be rotated in a random direction, and this rotation can be best described by a coordinate transformation matrix. The simulated cluster will be shifted into a rotated frame $\mathbf{x}'$ using:

$$\mathbf{x}' = \mathbf{A}\mathbf{x}$$

with the rotation matrix $\mathbf{A}$ defined as:

$$\mathbf{A} = \begin{bmatrix} \cos\psi cos\phi - cos\theta sin\phi sin\psi & \cos\psi sin\phi + cos\theta cos\phi sin\psi & sin\psi sin\theta \\ -sin\psi cos\phi - cos\theta sin\phi cos\psi & -sin\psi sin\phi + cos\theta cos\phi cos\psi & \cos\psi sin\theta \\ sin\theta sin\phi & -sin\theta cos\phi & cos\theta \end{bmatrix} \quad (2.6)$$

The values of ψ, ϕ, and θ determine the orientation of the cluster, and are required to compare with actual X-Ray observations of clusters.

To begin validating the simulation we start with a simple density distribution in hydrostatic equilibrium with a broken power law density profile. A general power law density profile has the form $\rho(l) \propto l^{-\alpha}$ where $\alpha > 1$. In contrast, the broken power law has different asymptotic values of the log slope of α depending on the value of the distance r. The threshold for the transition between the inner slope and the outer slope is denoted as r_0, the inner radius of the cluster. Even when relaxed, realistic clusters have a more complicated structure, but the broken power law formula is sufficiently accurate as to present a realistic test scenario for the simulation. To get a simplified profile the Hernquist model of density

distribution (Hernquist 1990) can be used. The Hernquist profile is a fairly accurate representation of the dark matter distribution predicted by supercomputer simulations of the evolution of structure in the universe. This gives a potential of:

$$\varphi = \frac{-GM}{r_0 + r} \Longrightarrow \frac{-GM}{r_0 + l}$$

G is the gravitational constant, and M is the total mass. Plugging the potential back into the parameterized version of the hydrostatic equilibrium equation, taking the derivative, and solving for pressure gives:

$$P = \int \frac{\rho_g GM}{(r_0 + l)^2} dl$$

With this pressure profile, and the ideal gas law a realistic temperature profile can be derived. Starting with the statistical mechanics version of the ideal gas law as:

$$PV = NkT$$

Where k is Boltzmann's constant, and N is the number of particles in the gas. N can be further defined as:

$$N = \frac{m}{\mu m_p}$$

Where μ is the average particle mass, m_p is the mass of a proton, and m is the total mass.

Using the definition of density ($\rho_g = \frac{m}{V}$), plugging in N, and solving for temperature gives:

$$T = \frac{\mu m_p}{k\rho_g} P$$

Using the pressure profile from above gives the desired temperature profile:

$$T = \frac{\mu m_p}{k\rho_g} \int_l^\infty \frac{\rho_g GM}{(r_0 + l)} dl$$

At this point we have picked a well-motivated shape for the overall mass distribution. The next step is to choose a suitable form for the gas distribution, since it is not coupled with the mass distribution within the hydrostatic equilibrium equation. While the details of the gas distribution differ from cluster to cluster, a reasonable form for our purposes is the following observationally determined average profile:

$$\rho_g \propto \frac{1}{(r_0 + l)^3}$$

yields the manageable temperature profile:

$$T(l) = 6.7 * \frac{1}{(r_0 + l)} \, keV \tag{2.7}$$

where *keV* is the unit of kilo-electron volts. In astrophysics keV is the unit used for temperature when dealing with high energy photons even though it is technically a unit of energy. When divided by the Boltzmann constant it converts to the common temperature

unit of Kelvin. The η_ρ profile is approximated in a similar manner to be:

$$\eta_\rho(l) = \frac{1}{(r_0 + l)^6}\, cm^{-6} \tag{2.8}$$

The simplified metallicity profile is slowly decreasing from 1 to 0 as:

$$Z(l) = \frac{lMax - l}{lMax} \tag{2.9}$$

These three profiles can be seen in Figure 7. They vary with l allowing for solutions to Equation 2.2, so the spectra throughout the entire cluster can be calculated. This can be done whether or not the simulated cluster is spherical.

On a standard CPU the simulation cannot be calculated in a reasonable amount of time. From testing, a moderately sized simulation running approximately 100 million integrations will take around 45 minutes. In actual "production" runs, hundreds of thousands of different cluster models must be evaluated before confirming a data match via likelihood analysis. This makes a run time of 45 minutes is unrealistically long. Fortunately, the calculations can be broken up by position on a grid, which is modeled after the charge-coupled device (CCD) of a telescope. They can also be broken up by energy range. The calculation for each position within each range is embarrassingly parallel, so it is possible to overcome the computational difficulties of creating a triaxial model. There are many ways to solve embarrassingly parallel problems, but the huge number of threads we require fits in nicely as a GPU computing application.

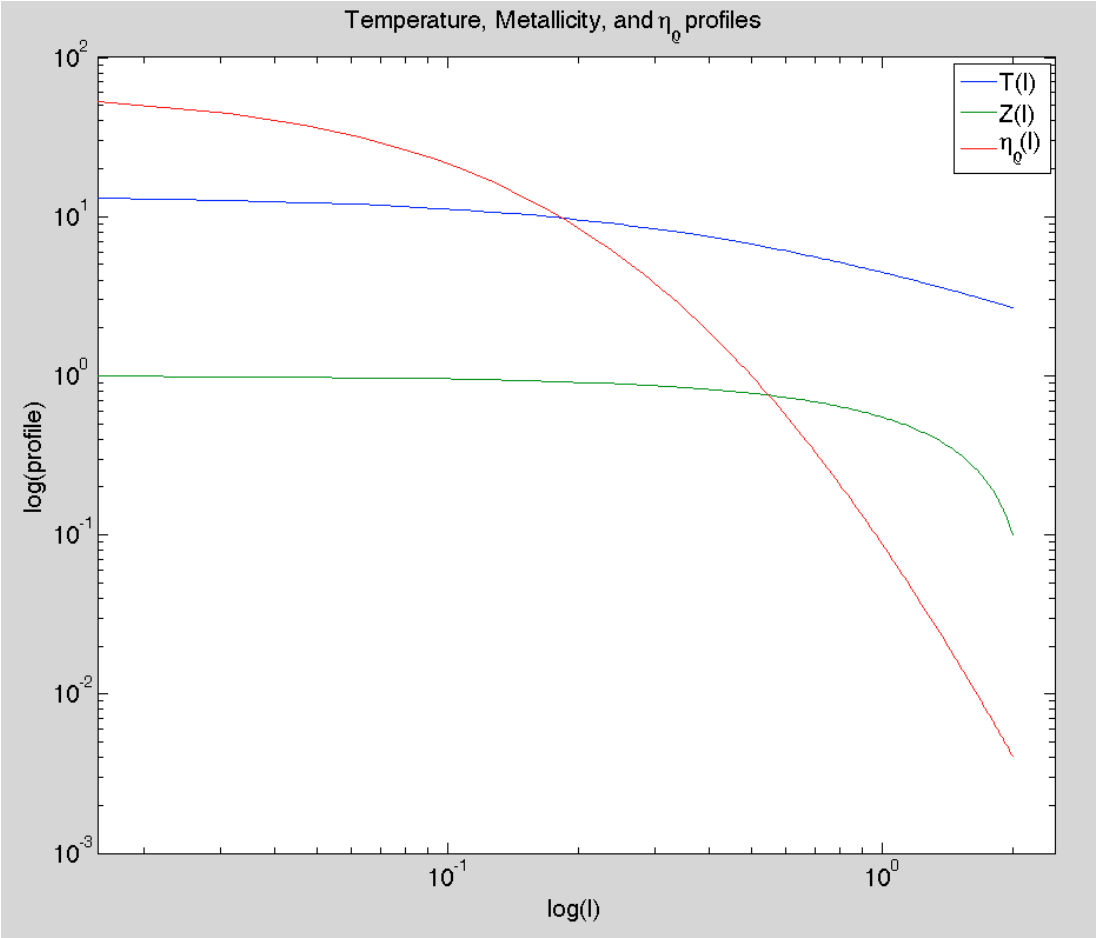

Figure 6: Temperature, metallicity, and η_ρ profiles used for the basic testing of the simulation. They are all a function of l, so they can accommodate both spherical and triaxial shapes of the simulated cluster.

The model for GPU computing runs the CPU and GPU together in a co-processing computing model. The sequential part of the application runs on the CPU, and the computationally intensive part is handled by the GPU. In this programming model, the application is modified to take the most computationally intensive sections, and map them to the GPU. Mapping a function to the GPU involves both rewriting the function to expose the parallelism, and moving the data back and forth from the GPU. Once this has been done, millions of threads can be run simultaneously. The GPU hardware manages the threads and does thread scheduling. With this programming infrastructure in place, GPU computing can offer very advantageous computational benefits. GPUs are designed to perform a tremendous amount of relatively simple computations that would normally be done on the CPU, so the primary problem is to properly parallelize the integration required to solve the Hydrostatic Equilibrium Equation.

3 Problem Statement

The primary problem addressed by this work is the creation of a C++ code using CUDA API to simulate a galaxy cluster that has a triaxial shape. The code integrates Equation 2.2 through the cluster's depth at all points within a grid. When this is completed it represents a CCD image of a galaxy cluster taken from an X-Ray telescope. For the simulation to be useful it will need to be run hundreds of thousands of times, so the individual runtime must be extremely fast. To achieve the desired runtime the code is properly parallelized so that it runs the integration on an NVIDIA Fermi Tesla GPU.

3.1 Contributions

Our contribution to this problem took many steps. The first was the confirmation that CUDA was properly set up on the cluster of computers to be used for the simulation. Then the specific project setup was organized, and test programs were run to verify that GPU computing is a viable way to run a simulation of this complexity. The most difficult part of the simulation was the integration, so a C++ program was written to utilize the extended trapezoidal rule for numerically solving an integral. This program was then rewritten to run in parallel with each thread being an energy level within a point on a 2-D grid. Equation 2.2 requires lookups for temperature, metallicity, ρ_g, and the cooling function, so we designed a way to parameterize the 3-D profiles to be a function of one parameter. This parameter is a function of both the desired shape and position of the cluster, so a rotation function and a shape function were written to run on the GPU. We also created an

efficient way to transform the profiles into 1-D arrays so they can be properly transferred to the GPU after the initial setup. During integration the 1-D arrays are looked up at points that are in between their preset values, so we wrote a 1-D interpolation function and a 2-D interpolation function. These functions will give an estimate of the value the profile would have at that point. A function was also written to transform the results back into a 3-D array that contains the spectra of the simulated cluster. From that 3-D array functions were written to create output files of the spectra at specified points, and the total flux of the cluster at each point on the simulated CCD grid. These were used by code that was written for MATLAB to create the images used in this work, and to verify the steps that were taken to create the code. The code must be highly flexible to be properly utilized as a simulation so a struct was created to handle all the constants the code requires to run the simulation.

4 Relevant Work

Numerical modeling is an essential tool for astronomers. It allows for the comparison of theory to actual observed data, but on a dramatically more detailed scale. By modeling structures like galaxy clusters the astronomer can test competing theories to compare with the observed structures seen through telescopes. There is no well defined method for an astronomer to create a model. When a high school Physics student graphs a basic equation of motion like $x = x_0 + vt$ they are creating a model of the position of a object at a certain time after the object was located at x_0. The astronomer's task is a more complicated version of what the student is doing. Depending on what the astronomer wants to study they will create a model from the known equations and theories that apply to the subject. They are not bound by a type of software, or a specific algorithm. The goal is to match theory to actual observations. The raw observations are in the form of a picture, or a spectrum similar to the one in Figure 5. If the theory does not match observation the difference must be explained by a new theory of the subject being studied and the model must be modified.

The advance of GPUs used for scientific modeling has caused something of a revolution in parallel programming, and allowed for advances in a number of scientific disciplines. In astronomy alone GPUs have been used for N-body simulations (Tanikawa et al. 2012), radio astronomy measurements (Clark et al. 2011), adaptive mesh refinement hydrodynamics (Schive et al. 2010), pulsar dedispersion (van Straten & Bailes 2010), and even solving Kepler's equations (Ford 2008).

Thompson et al. (2010) used CUDA and an NVIDIA S1070 Tesla unit to show that a

billion-lens microlensing simulation can be run on order of a day. They compared their GPU version of the simulation with one that ran entirely on one thread of a high end CPU (Intel Core 2 Quad Q6600). The Tesla card comprises of 112 stream processors, each running at 1.6 GHz, with a peak performance of 336 Gflop/s, while the CPU was only capable of 76.8 Gflop/s. NVIDIA is one of the two major graphics processor manufacturers, and they introduced GPGPU (General Purpose computation on Graphics Processing Units) software API in 2006. CUDA extends C and C++ programming languages, and greatly simplify GPGPU programming by extracting the GPU code from the rest of the program. CUDA allows the programmer to define functions within the standard code that are then executed in parallel on any connected GPUs. Thompson et al. (2010) showed a 100-fold increase in performance when using the GPU code on only one GPU, and even more dramatic speedup when utilizing a four core GPU. This is significant for microlensing, because the brute force method of taking into account for the macro-model shear was previously written off as impractical. It would have taken months or years of computation time, but with a S1070 Tesla it can be computed in about 7 hours.

The dramatic results shown by Thompson et al. (2010) are not confined to microlensing. Bard et al. (2012) showed a 100-300x speedup (depending on amount of data) in calculating the two-point angular correlation function and the aperture mass statistic when using CUDA and a Tesla M2070 GPU card. Their motivation comes from the massive scale of modern telescopic surveys, and that the calculation of many cosmological quantities are of complexity of $O(n^2)$. This complexity can be broken down to $O(n_{pts} * n_{gals})$ where n_{pts} is the number of points in space that are being evaluated, and n_{gals} is the number

of galaxies within the observation. The benefit of computing on a GPU is that each point in space becomes a thread that is run in parallel and the complexity is effectively reduced to $O(n_{gals})$. There are increases in time due to the amount of memory transferred between the CPU and GPU, but this not on the same scale as the speedup from the parallelization. The methodology taken in Bard et al. (2012) can be compared to the methodology used in modeling galaxy clusters. To model a galaxy cluster a calculation must be made for each pixel at each energy level, and each calculation involves a number of evaluations between the integration limits. If done without parallelization the complexity would be $O(n_{pix} * n_{ebins} * n_{evals})$, but by using the GPU one thread can be used for one energy bin on one pixel. This simplifies the complexity to $O(n_{evals})$, to give the speedup shown below.

5 Methodology

When astronomers observe a galaxy cluster with an X-ray telescope the data they receive is similar to a digital image, in the form of a 2-dimensional grid of pixels. Just like all pictures of 3-dimensional objects, when the picture is taken it is recorded as a projection onto a 2-D plane. The measurements recorded by each pixel on the 2-D grid are the result of the added light through the line of sight of the observer. This is visualized within Figure 7. Galaxies are shown in the figure, even though the emission comes from the gas contained within the cluster. The recorded energies on the pixel can be displayed similarly to the spectrum in Figure 5. The properties of the cluster can then be analyzed from the information contained within the spectrum. A spectrum can be thought of as a one-dimensional array of luminosity at increasing energy channels. In essence the data received from the telescope is a 2-D map of a 3-D object, and at each point in the map there is a 1-D array of the spectrum. The final result of the simulation is to have the same 2-D grid with a spectrum for each pixel. This is achieved by properly solving Equation 5.1 at all points throughout the grid.

$$L_\upsilon = \int_{l_a}^{l_b} \eta_\rho(l)\Lambda_\upsilon[T(l),Z(l)]dl \tag{5.1}$$

Since Equation 5.1 is dependent on temperature, metallicity, and η_ρ these values must be calculated before inputting them into the equation for the spectra.

The integration in Equation 5.1 is done on the GPU while lookup tables are created by

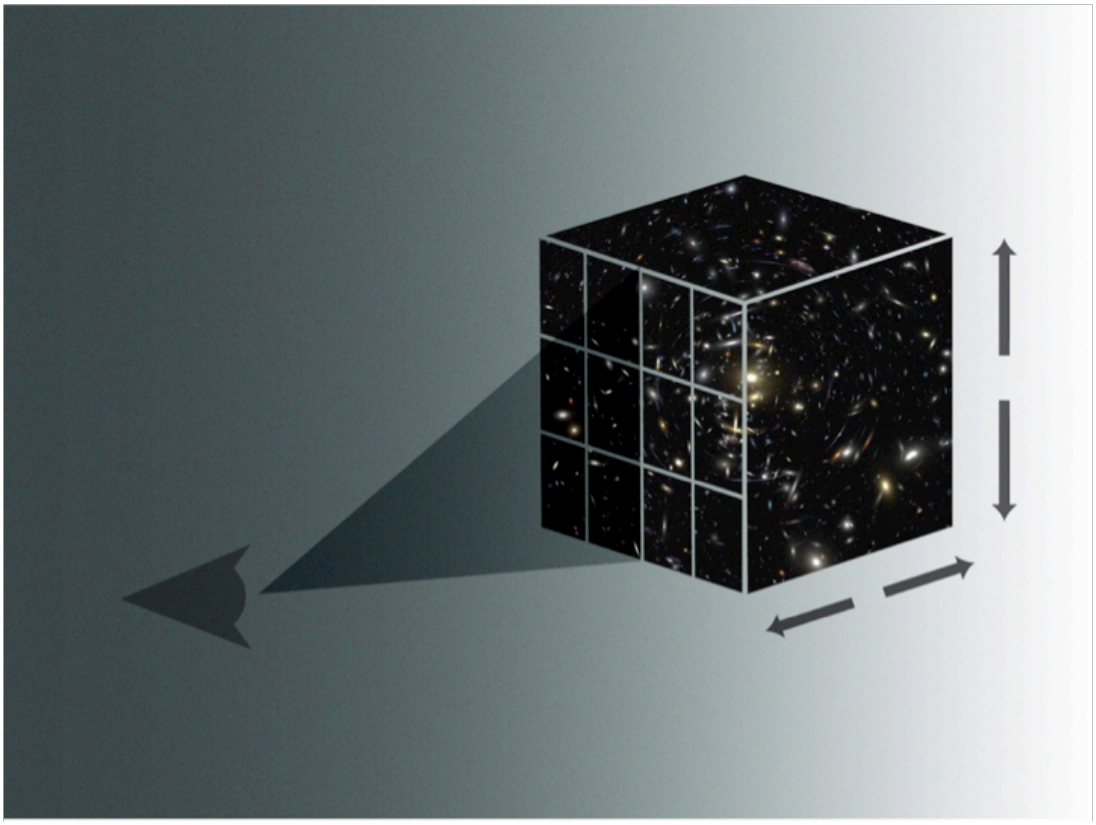

Figure 7: The spatial orientation of the cluster is broken up into a 3-D grid. The projected plane seen by the observer is represented by the grid of threads run on the GPU. Each thread can be thought to integrate through the depth of the box. Only galaxies are visible in this figure even though the simulation is calculating for the properties of the gas within the cluster.

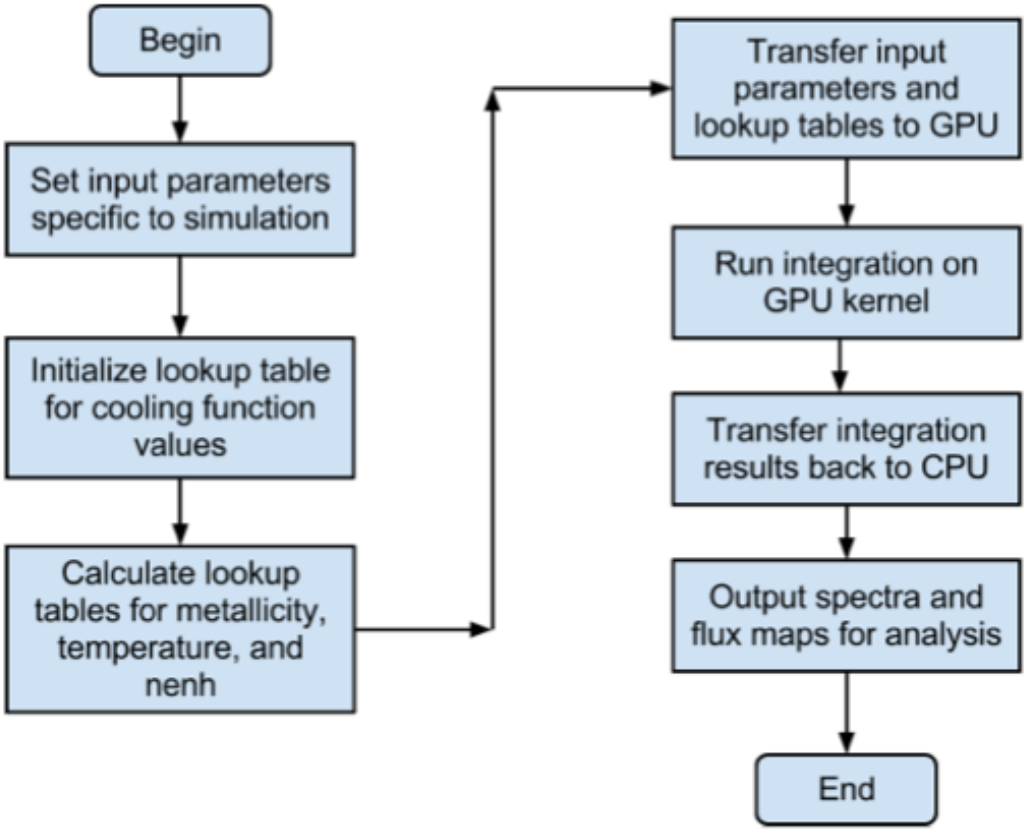

Figure 8: Flow diagram outlining the key steps of the simulation.

CPU before the call to the GPU is made. For the GPU to be able to calculate the spectra it needs to be fed three-dimensional arrays that describe the temperature, metallicity, and η_ρ profiles. The values that populate these profiles are dependent on the spatial coordinates and orientation of the simulated cluster. The values of these arrays are generated by the CPU before the GPU kernel is called. Each pixel of a simulated image is represented as a computational block by the GPU, and each energy level in the spectrum is represented by a thread running on the GPU for each block. The breakup of blocks and threads is visualized in Figure 9. Each thread can be thought of as integrating along the line of sight, by the way it solves the integral in Equation 5.1. When the GPU kernel is called to make the calculations, it divides the work into the pre-described blocks (pixels) and threads (energy channels). When all the threads have completed their calculations, the solutions are collected and sent back to the CPU. The number of values sent back to the CPU will be the number of blocks times the number of threads. This corresponds to the number of pixels times the number of energy channels. For an accurate simulation there is at least 4 million threads running in parallel.

The basic flow of the simulation is shown in Figure 9, and a more detailed description of the algorithmic steps and their motivations are as follows:

1. Before the simulation has been started a lookup table of all possible cooling function ($\Lambda_\upsilon[T(l), Z(l)]$) values is created. Instead of the cooling function being continuous, it is converted to a table that takes discrete inputs of temperature and metallicity. The size of the lookup table is based on the resolution required by the simulation.

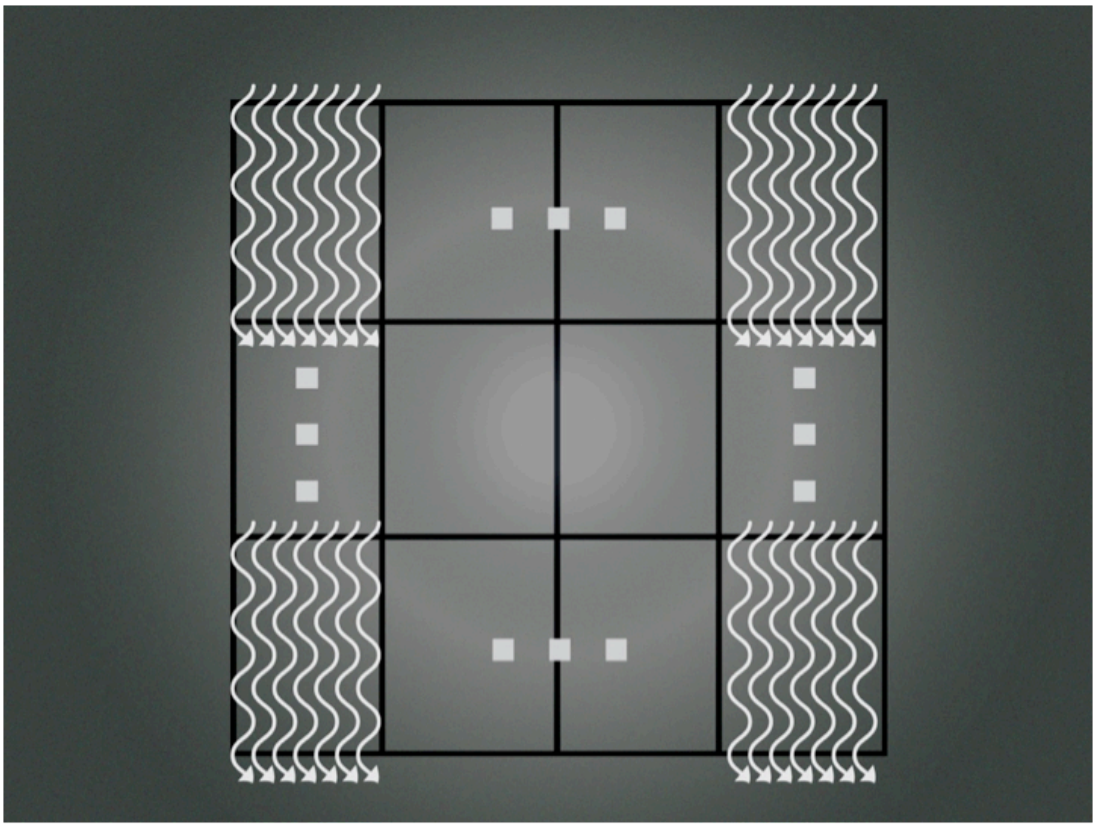

Figure 9: Each block of the grid is characterized by its X and Y coordinate. Within each block of the grid there is a calculation thread for each energy channel. Each thread integrates through the Z-axis of the cluster. All threads run simultaneously, and can share memory within their block.

The lookup table is converted into a 1-D array before it is transferred to the GPU.

2. Using assumed and highly flexible functional forms for the triaxial potential and metallicity distributions, the temperature, η_ρ, and metallicity are calculated at discrete points within the 3-D structure of the cluster. The number and position of discrete points are determined by the range of values of *l*, which is an input parameter of the simulation. The larger the range, the higher the resolution of the simulation, and the larger the arrays will be for the three distributions. The values for *l* are initialized in ellArrInit(), where nEll is the number of values of *l*, and ellMax is the maximum value of *l*.

```
//initialize array for l
double* ellArrInit(int nEll, double ellMax){
    int x;
    double eMin = 0.1;
    double eMax = ellMax;
    double binWidth = (eMax-eMin)/nEll;
        double *ellArr = new double[nEll];
        ellArr[0] = binWidth;
        for (x=1; x<nEll; x++) {
            ellArr[x] = ellArr[x-1]+binWidth;
        }
    return ellArr;
}
```

3. The 3-D temperature (T), metallicity (Z), and η_ρ profiles are initialized as a function of l. This allows them to be parameterized into 1-D arrays for easy transfer to the GPU. For preliminary testing the metallicity profile is defined by Equation 2.9, and is initialized by metalArrInit(). This is a simple decreasing function ranging from 1 to 0.1.

```
//initialize metallicity array
double *metalArrInit(int ellMax){
    double *metalArr = new double[nEll];
    double offset;
    for(int l=0; l<=ellMax; k++){
        offset = l*0.9;
        metalArr[l] = log10((ellMax-offset)/ellMax);
    }
    return metalArr;
}
```

The η_ρ profile is defined by Equation 2.8, and is initialized by emmArrInit(). For emmArrInit(), nEll is the number of values of l, and ellMax is the maximum value of l. rNot is a constant determined by the ratio of pixels to mega-parsecs, multiplied by the diameter of the simulated cluster.

```
//initialize nenh array
double *emmArrInit(int nEll, double ellMax, double rNot){
```

```
    double *emmArr = new double[nEll];
    double *ellArr = new double[nEll];
    ellArr = ellArrInit(nEll, ellMax);
    for(int l=0; l<nEll; l++){
        emmArr[l] = log10(pow(rNot+ellArr[l],-6));
    }
    return emmArr;
}
```

The temperature profile is defined by Equation 2.7, and is initialized by tempArrInit().

```
//initialize temperature array
double* tempArrInit(int nEll, double ellMax, double rNot){
    double *tempArr = new double[nEll];
    double *ellArr = new double[nEll];
    ellArr = ellArrInit(nEll, ellMax);
    for(int l=0; l<nEll; l++){
        tempArr[l]=log10(6.7*pow(rNot+ellArr[l],-1.0));
    }
    return tempArr;
}
```

4. Space is allocated on the GPU for the arrays, the cooling function lookup table, the rotation matrix, and the integral solution table. This is done using the CUDA

API call for Malloc. The memory allocation for the metallicity array is done with cudaMalloc().

```
size_t sizeArr = theConst.n_ell*sizeof(double);
cudaMalloc((void**)&metalArr_d, sizeArr);
```

The lookup arrays are then transferred into the GPU. This is done using the CUDA API call for Memcpy. The memory transfer for the metallicity array is done with cudaMemcpy().

```
cudaMemcpy(metalArr_d, metalArr_h, sizeArr, cudaMemcpyHostToDevice);
```

Where metalArr_d is the metallicity array on the device (GPU), metalArr_h is the metallicity array on the host (CPU), and sizeArr is the size of both arrays determined by the number of l values (theConst.n_ell). This procedure is repeated for all arrays transferred to the GPU. Once this step is complete there are identical matrices on both the device and the host.

5. The GPU kernel is called to calculate the integral in Equation 5.1. Within the GPU the space is divided into a grid of blocks where each block on the grid represents a pixel in the final 2-D image. Each pixel has an X and Y coordinate, that is used to look up values required to solve Equation 5.1. Within each pixel there are a number of threads, each representing a single photon energy channel. The number of threads is dependent on the resolution of the spectra required by the simulation. To call the GPU function integrate() the CUDA API is accessed in the following way:

```
integrate<<<theConst.nPixX*theConst.nPixY, theConst.numEnergyChannels>>>
(rebinArr_d, tempArr_d, metalArr_d, emmArr_d, integral_d, tempGrid_d,
metalGrid_d, rotMat_d, theConst, debugging)
```

Instead of a normal function in C/C++ the number of blocks are stated before the call as theConst.nPixX*theConst.nPixY. The number of threads per block is stated as theConst.numEnergyChannels. These two variables are recognized by the $<<<$ and $>>>$ in the function call. The variable rebinArr_d is the cooling function lookup table, tempArr_d is the temperature array, metalArr_d is the metallicity array, emmArr_d is the η_ρ array, and integral_d is the array for the results of the integration. The variables for tempGrid_d and metalGrid_d are arrays used for the bilinear interpolation. Variable rotMat_d is the rotation matrix used to transform l into the rotated frame of the cluster. The identifier _d is the convention for denoting that the data structure resides on the GPU. Variable theConst contains all of the input parameters that serve as constants, and variable debugging is a boolean flag used for debugging errors on the GPU.

6. Each thread solves the integral in Equation 5.1 with respect to l by using the extended trapezoidal rule:

$$\int_a^b f(x)\,dx = h[\frac{1}{2}f_1 + f_2 + f_3 + ... + f_{N-1} + \frac{1}{2}f_N] + O\left(\frac{(b-a)^3 f''}{N^2}\right)$$

Where h is the step width defined as $h = \frac{b-a}{N}$, and N is the number of terms being

evaluated for function f. The final term is the error term that is ignored, because it is insignificantly small. $f(x)$ is the function is being integrated. For the simulation it is the function to calculate the spectrum defined as:

$$f(l) = \eta_\rho(l)\Lambda_\upsilon[T(l), Z(l)] \tag{5.2}$$

The function first pulls lookup values from both the temperature profile ($T(l)$) and the metallicity profile ($Z(l)$). These lookups are used to evaluate the cooling function (Λ_υ). The cooling function value is then multiplied by a lookup value from the η_ρ array. This is done multiple times at discrete points between the integration limits. Implementing the extended trapezoidal rule can be done iteratively until an accurate result is found.

```
Calculate the trapezoidal rule for only limits of integration a and b
Set results of trapezoidal rule to next_value
Set actual_error as next_value
Initialize step to 1
While actual_error is greater than 0.1
    Set last_value to next_value
    Multiply step by 2
    Calculate trapezoidal rule for step number of values
    Set results of trapezoidal rule to next_value
    Set actual_error to the absolute value of (next_value-last_value)
Return last_value
```

This process is implemented in the CUDA function integrate(). The first step in integrate() is to initialize all the variables needed.

```
__global__ void integrate(double* rebinArr, double* tempArr,
double* metalArr, double* emmArr, double* integral, double* tempGrid,
double* metalGrid, double* rotMat, constants theConst, bool debugging){
  //threadIdx.x is the channel/energy-bin
int i=blockDim.x * blockIdx.x + threadIdx.x;
int j= blockIdx.x;
int x, y;
//integrate from depth/2 to -depth/2
x = j/theConst.nPixX-(theConst.nPixX/2);
y = j
//convert from pixels to mega-parsecs
double pixToMpc = 0.01;
double xMpc = pixToMpc*x;
double yMpc = pixToMpc*y;
int energyBin = threadIdx.x;
int n;
double a, b;
double step=1.0;
double tFunct, h, actErr, last, nextVal;
double T, Z, rebinA, rebinB;
double prevStep[200];
```

```
double ell;
//a and b are the limits of integration
b = theConst.nz/2;
a = -theConst.nz/2;
h = b-a;
actErr = 1.0;
last = 1.E30;
nextVal = 0.0;
n = 1;
```

The next step is to solve for $\frac{1}{2}f_1$ and $\frac{1}{2}f_N$ in the trapezoidal rule. This entails solving Equation 5.2 at the limits of integration a and b.

```
//solve for the limit at a
ell = getEll(xMpc, yMpc, a, rotMat, theConst.a_ell, theConst.b_ell);
T = linearInterpEll(tempArr, ell, theConst);
Z = linearInterpEll(metalArr, ell, theConst);
rebinA = bilinInterpVal(rebinCool, T, Z, energyBin, tempGrid,
metalGrid, theConst);
tFunct = 0.5*__powf(__powf(10,linearInterpEll(emmArr, ell, theConst)),
 2.0)*rebinA;
//solve for the limit at b
ell = getEll(xMpc, yMpc, b, rotMat, theConst.a_ell, theConst.b_ell);
T = linearInterpEll(tempArr, ell, theConst);
```

```
Z = linearInterpEll(metalArr, ell, theConst);
rebinB = bilinInterpVal(rebinCool, T, Z, energyBin, tempGrid,
 metalGrid, theConst);
//sum to get first step of integration
tFunct += 0.5*__powf(__powf(10,linearInterpEll(emmArr, ell,
 theConst)), 2.0)*rebinB;
```

Once the limits have been determined the next step is to go through the iterative process of solving the trapezoidal rule.

```
//iterate until convergence of integral
prevStep[0] = tFunct;
//alpha is the threshold value
double alpha = 0.1;
while (actErr>=alpha) {
    step=step*2.0;
    h = double(b-a)/step;
    nextVal = 0.0;
    for (int l=1; l<step; l=l+2) {
        ell = getEll(xMpc, yMpc, l*h+a, rotMat, theConst.a_ell,
         theConst.b_ell);
        T = linearInterpEll(tempArr, ell, theConst);
        Z = linearInterpEll(metalArr, ell, theConst);
        nextVal+=bilinInterpVal(rebinCool, T, Z, energyBin, tempGrid,
```

```
        metalGrid, theConst)*__powf(__powf
        (10,linearInterpEll(emmArr, ell, theConst)), 2.0);
    }
    nextVal+=prevStep[n-1];
    prevStep[n]=nextVal;
    nextVal=h*(nextVal);
    actErr=fabs(last-nextVal);
    last=nextVal;
    n++;
    }
//place integrations into the 1D array by thread number
integral[i] = last;
}
```

With the solution in place the results are sent back to the CPU in the integral array. The individual solutions are indexed by the thread number i. Each evaluation point of Equation 5.2 is done in a multistep process.

(a) X, Y, and Z are modified based on the rotation of the desired cluster. This is done by using the predetermined values of ψ, ϕ, and θ as seen in Equation 2.3. X and Y come from the specific block (pixel), and Z is the point being evaluated for the trapezoidal rule.

(b) The value of l is determined using Equation 2.6 with X, Y, Z, A, and B. A and B are specified by the input parameters. This is returned from the following

function:

```
__device__ double getEll(double x, double y, double z,
 double* rotMat, double a, double b){
    double xPrime, yPrime, zPrime;
    xPrime = rotMat[0]*x + rotMat[1]*y + rotMat[2]*z;
    yPrime = rotMat[3]*x + rotMat[4]*y + rotMat[5]*z;
    zPrime = rotMat[6]*x + rotMat[7]*y + rotMat[8]*z;
    return __powf(xPrime*xPrime + yPrime*yPrime/(a*a)
     + zPrime*zPrime/(b*b),0.5);
}
```

(c) The lookup values for T, Z, and η_ρ are calculated from l in the previous step. Since the T, Z, and η_ρ arrays are not continuous functions, if l lies between two lookup points of the array a linear interpolation is performed on the array in question.

(d) The cooling function array is a 1-D array within the GPU, but is a converted 2-D lookup table since it is a function of both T and Z. Since it is actually in two dimensions a bilinear interpolation on the cooling function array is performed to return the lookup value from the input values of T and Z found in the previous step. This complex interpolation is returned from the following functions:

```
__device__ double bilinInterpVal(double* interpMat, double y,
 double z, int energyBin, double* tempGrid, double* metalGrid,
```

```
 constants theConst){
int y1, y2, z1, z2;
double temp1, temp2, R1, R2, interpValue;
double ten11, ten12, ten21, ten22;
double y1val, y2val, z1val, z2val;
int nx = theConst.binCenterSize;
int ny = theConst.tGridSize;
int nz = theConst.mGridSize;

y1 = getLowerIndex(tempGrid, __powf(10,y), theConst.tGridSize);
y2 = y1+1;
z1 = getLowerIndex(metalGrid, __powf(10,z), theConst.mGridSize);
z2 = z1+1;

y1val = tempGrid[y1];
y2val = tempGrid[y2];
z1val = metalGrid[z1];
z2val = metalGrid[z2];
y = __powf(10,y);
z = __powf(10,z);
temp1 = (y2val-y)/(y2val-y1val);
temp2 = (y-y1val)/(y2val-y1val);
ten11 = __powf(10,tenRetrieve(interpMat, nx, ny, nz,
```

```
 energyBin, y1, z1));
ten21 = __powf(10,tenRetrieve(interpMat, nx, ny, nz,
 energyBin, y2, z1));
ten12 = __powf(10,tenRetrieve(interpMat, nx, ny, nz,
 energyBin, y1, z2));
ten22 = __powf(10,tenRetrieve(interpMat, nx, ny, nz,
 energyBin, y2, z2));
R1 = temp1*ten11 + temp2*ten21;
R2 = temp1*ten12 + temp2*ten22;
interpValue = ((z2val-z)/(z2val-z1val))*R1 +
((z-z1val)/(z2val-z1val))*R2;
return interpValue;
}

//helper function to get the lower index when a point lies
//between two lookups of an array
__device__ int getLowerIndex(double* axisArr, double value,
 int arrLength){
    for(int i=0; i<arrLength; i++){
        if (value<axisArr[i]) {
            return i-1;
        }
    }
```

```
    return arrLength-2;
}
//function to pull a value from the tensors when they have
//been converted into 1D arrays
__device__ double tenRetrieve(double* oneDArray, int nx,
int ny, int nz, int x, int y, int z){
    int index = x*ny*nz + y*nz + z;
    return oneDArray[index];
}
```

(e) The cooling function interpolation value is multiplied by the lookup returned by the η_ρ array to get the result for the point being evaluated.

7. When all of the threads have calculated their respective integrals send the array of spectra back to the CPU.

8. The 1-D resulting array is transformed back into a three dimensional tensor that is a function of X, Y, and the energy channel.

The model spectra are then complete, and can be used to compare with observational data. This can be done by evaluating the values at all the energy levels at one XY-coordinate to get one spectrum similar to the one seen in Figure 5. If all the energy levels at each coordinate are summed and all the coordinates are mapped this will give a flux map of the entire galaxy cluster.

This algorithm has the time complexity of $O(n_{pix} * n_{ebins} * n_{evals})$ where n_{pix} is the number of pixels on the CCD being modeled, n_{ebins} is the number of photon energy channels, and n_{evals} is the number of lookups need to be made before the integration converges. In practice n_{evals} ranges from 5 to 66 evaluations for a high accuracy. By a large margin it is most frequently equal to 7. Each thread of computation is for one energy channel on each pixel, and since they execute in parallel the complexity reduces to $O(n_{evals})$ per thread.

6 Experimental Evaluations

6.1 Simulation Viability

The motivation of this project is to create a simulation that is both accurate and efficient, so each of the key elements was rigorously tested on its own. The most important section of the code is the actual integration, so it was first written as a general 1-D integration with no use of the GPU. The numerical integration was first implemented using the trapezoidal rule, and at each step the values between the integration limits were re-evaluated. This can be seen in the following code section:

```
//function to integrate a continuous function defined as f(x)
void cpuTrapzd(int* list, float* funct, int lSize, const float err){
    float h, tFunct;
    float nextVal, actErr;
    int a, b, step=1;
    float last;
    int i, k;
    for (i=0; i < lSize; i++){
        a=list[i];
        b=list[i+lSize];
        h=(b-a);
        actErr=1.0;
        nextVal=0.0;
```

```
        last=1.E30;
        tFunct=0.5*(f(a)+f(b)); //first step with only end values
        while (actErr>=err) { //will break when difference of
        //successive calculations less than err
            h= (float)(b-a)/step;
            nextVal=0;
            for (k=1; k<step; k++) {
                nextVal+=f(k*h+a); //iteration with equal hashes
            }
            nextVal=h*(tFunct+nextVal);
            actErr=fabs(last-nextVal);
            last=nextVal;
            step++;
        }
        funct[i]=last;
    }
}
```

Once the accuracy of this approach was verified, it was optimized by storing previous iterations, and doubling the number of evaluations at each step. This sped up the basic integration by two fold. After optimizing the routine, the CPU version was modified to run on the GPU. This was a straightforward change, as each integration was identified by the thread number collected within the GPU rather than having a “for loop” handle all the integrations in series. Using thread number as an identifier allows for easy lookup once

the integration has been completed, and the values are sent back to the CPU for evaluation. The modified GPU version can be seen in the following code section:

```
//calling the gpuTrapzd function
gpuTrapzd<<<blocksPerGrid, threadsPerBlock>>>(d_inputList,
 d_test1, nIntegrals, accuracy);


//GPU function to integrate f(x) but f(x) is preset in this version due
// to limitations on transferring functions to the GPU
// here f(x) = (float(__sinf(__powf(x, 1.0/3.0)))
__global__ void gpuTrapzd(int* list, float* funct,
int nIntegrals, const float err){
    float h, tFunct;
    float nextVal, actErr;
    int a, b, step=1;
    int i=blockDim.x * blockIdx.x + threadIdx.x;
    float last;
    double prevStep[200];
    if (i<nIntegrals) {  //section for f(x)
        a=list[i];
        b=list[i+nIntegrals];
        h=(b-a);
        actErr=1.0;
        last=1.E30;
```

```
        nextVal=0.0;
        tFunct=0.5*(float(__sinf(__powf(a, 1.0/3.0)))+
        float(__sinf(__powf(b, 1.0/3.0))));
        prevStep[0]=tFunct;
        while (actErr>=err) { //will break when difference of
         //successive calculations less than err
            step=step*2;
            h= float(b-a)/step;
            nextVal=0.0;
            for (int k=1; k<step; k=k+2) {
                nextVal+=float(__sinf(__powf(k*h+a, 1.0/3.0)));
            }
            nextVal+=prevStep[n-1];
            prevStep[n]=nextVal;
            nextVal=h*(nextVal);
            actErr=fabs(last-nextVal);
            last=nextVal;
        }
        funct[i]=float(last);
    }
}
```

In Figure 10 the ratio of runtimes from CPU and GPU versions are shown. The 1-D integration was tested with $\int_{x_a}^{x_b} sin(x^{1/3})\,dx$, but any integratable function could have been

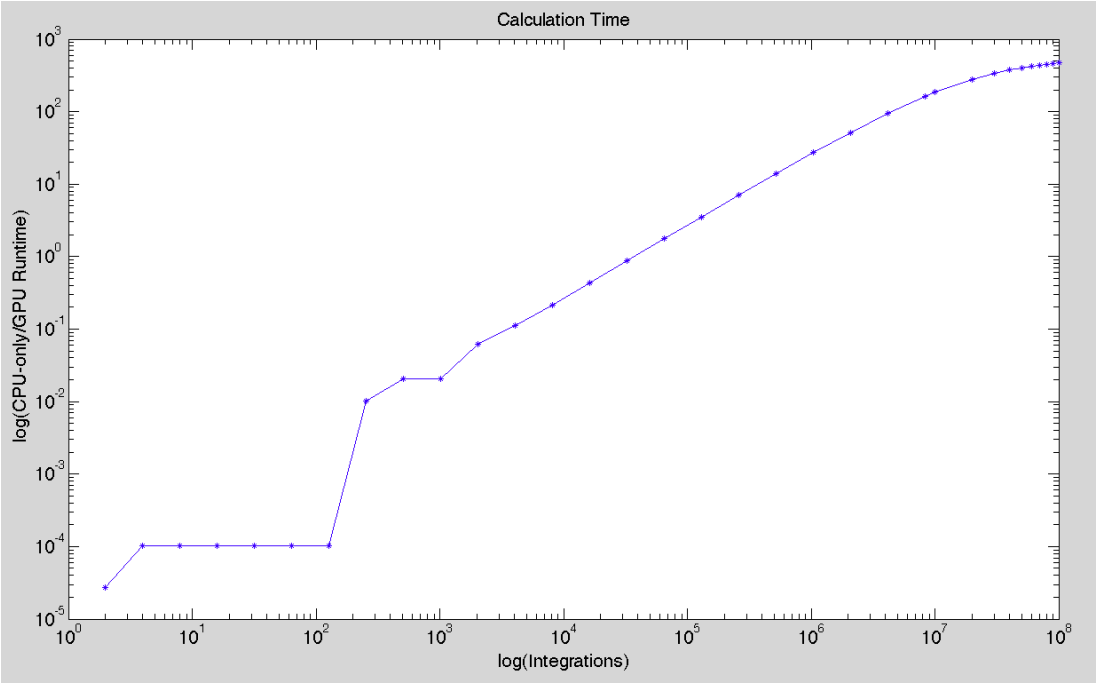

Figure 10: Ratio of runtime for CPU-only integration over GPU integration. This is plotted against the number of integrations performed. The GPU vastly outperforms the CPU-only approach by remaining relatively constant as the number of integrations increases. CPU-only version takes approximately 44 minutes with 100 million calculations, while the GPU method still only takes around 5.6 seconds.

used. The limits of integration were randomly generated. At around 4 million integrations, the GPU version approaches a 100-fold increase in speed. This is significant in that the smaller versions of the simulation will run on around 4 million threads.

Once the 1-D version was verified it was converted into a 3-D version that would fit the needs of the theoretical simulation. Dummy functions were set up to initialize the 3-D temperature, metallicity, and η_ρ profiles. After initial testing, it was discovered that the

way the data structures are transferred into the GPU would cause the 3-D arrays to lose data. To overcome this obstacle the profiles were parameterized into 1-D arrays, and then evaluated after they are sent to the GPU. This way they are treated as a lookup table that has been populated beforehand. They were parameterized by l so the lookup values can be generated with any equations similar to equation 2.7. When the value of l is looked up within the GPU, first the orientation of the cluster must be taken into account. The x, y, and z coordinates are multiplied by the rotation matrix $\mathbf{A}$ defined in equation 2.6. The rotated x, y, and z coordinates along with the chosen values of A and B are used with equation 2.3, to get the specific l at the desired point and shape. The choice to conveniently do the rotation at the same time as the triaxial modifications was not apparent in the initial planning of the simulation. It was tested to either be generated inside or outside the GPU, and it was determined that the lookup tables would be dramatically larger if all possible rotations were calculated outside, and inside they were merely looked up. Since memory transfer is assumed to be the time bottleneck of a GPU simulation, much larger lookup tables means much more transferred data, and would result in a dramatically longer run time. It is also much simpler to do the rotation and spatial modifications whenever the parameterized value of l is needed. Refer to the Methodology section for the version of these lookups and the final GPU integration function.

After the different pieces of GPU code were placed together, the entire simulation was clocked for each of the different segments of calculation and memory transfer. This was done to verify the assumptions about location of various bottlenecks in calculation time. It was found that the largest two computational sections with respect to time are the memory

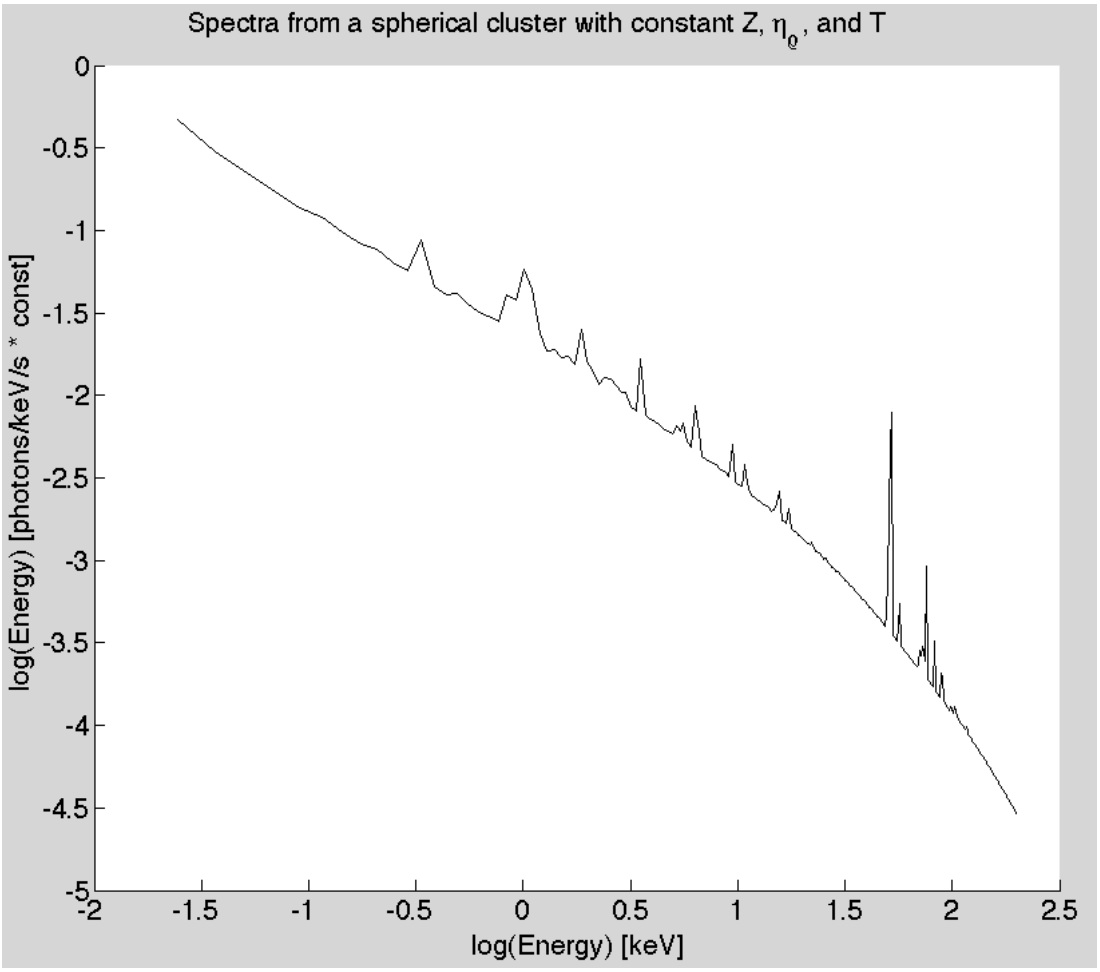

Figure 11: Flux plotted against magnitude of energy channel to give the spectrum of the cluster at an XY-coordinate. Generated with a constant metallicity of 1.0, temperature of 3 keV, η_ρ of 1.0, and $A = B = 1.0$. Since the profiles are constant with respect to l the spectrum is the same at all coordinates.

transfers between the CPU and GPU, and the initialization of the profile matrices. For a basic 128x128 XY-grid with 256 energy bins the generating of the temperature, metallicity, and η_ρ profiles takes about 2 seconds, the memory transfer back and forth to the GPU about 5 seconds, and the actual integration takes about 1 second. This supports the preconceived notions about GPU computing, and was the motivation for making this simulation only call the GPU once. The simulation is optimized by having only one memory transfer, and maximizing calculations within the GPU before sending results back to the CPU.

The codebase was then reconfigured to take a C struct with all of the input parameters needed to run the simulation. This struct contains the number of X and Y values, the depth of the cluster in Z, and the number of energy bins for each spectrum. It also contains the values for A, B, ϕ, θ, and ψ. The arrays it will bring into the simulation are the binned cooling function, as well as the temperature, metallicity, and η_ρ profiles. The arrays can also be generated within the simulation, but for simplicity they will be supplied by the external C struct. The C struct serves as a container for all of the constants required by simulation, so it can be easily modified to take into account further theoretical modifications. Once the values are set at startup they are referenced throughout the code.

6.2 Simulation Verification

To ensure the returning spectra were accurate, each step of the code was verified before the next step considered. The verification was done through comparison with previous models, and plots of equations of hot gas spectra. The first major check was that the

spectra generated for each point in the grid has the expected profile at each energy channel. This was first verified on a spherical cluster, and the spectra were generated with constant profiles for temperature, metallicity, and η_ρ. This results in the same projected spectrum at any coordinate within the grid. The spectra generated from this test are seen in Figure 11. This is what would be expected from a spherical gas cloud where temperature, metallicity, and η_ρ are not dependent on distance from the center of cloud.

Once this had been verified it was checked again with more complicated metallicity and temperature profiles that varied depending on position within the cluster. Figure 12 shows generated spectra of a spherical cluster with a constant metallicity of 0.5, constant η_ρ of 3.0, and a temperature profile given by equation 2.7. Each spectrum is from a different region of the cluster and they fit in well with the expected spectra for this theoretical cluster. Figure 13 shows how the spectra look when temperature, metallicity, and η_ρ are all a function of l.

After the generated spherical spectra were verified, the values of A and B were changed from 1.0 to create an elongated cluster instead a spherical one, and the previous tests were run again. The comparison between triaxial and spherical can be seen in Figure 14. Each spectrum is taken at the same X and Y coordinates, but only the green line is from a spherical cluster and the other two are elongated in opposite directions. Since only the temperature profile is a function of l this figure resembles Figure 12. Rather than taking measurements from different locations in the cluster like Figure 12, three different clusters were generated, and each spectrum was taken at the same location.

After the results of these tests were verified, the rotational parameters were tested. To

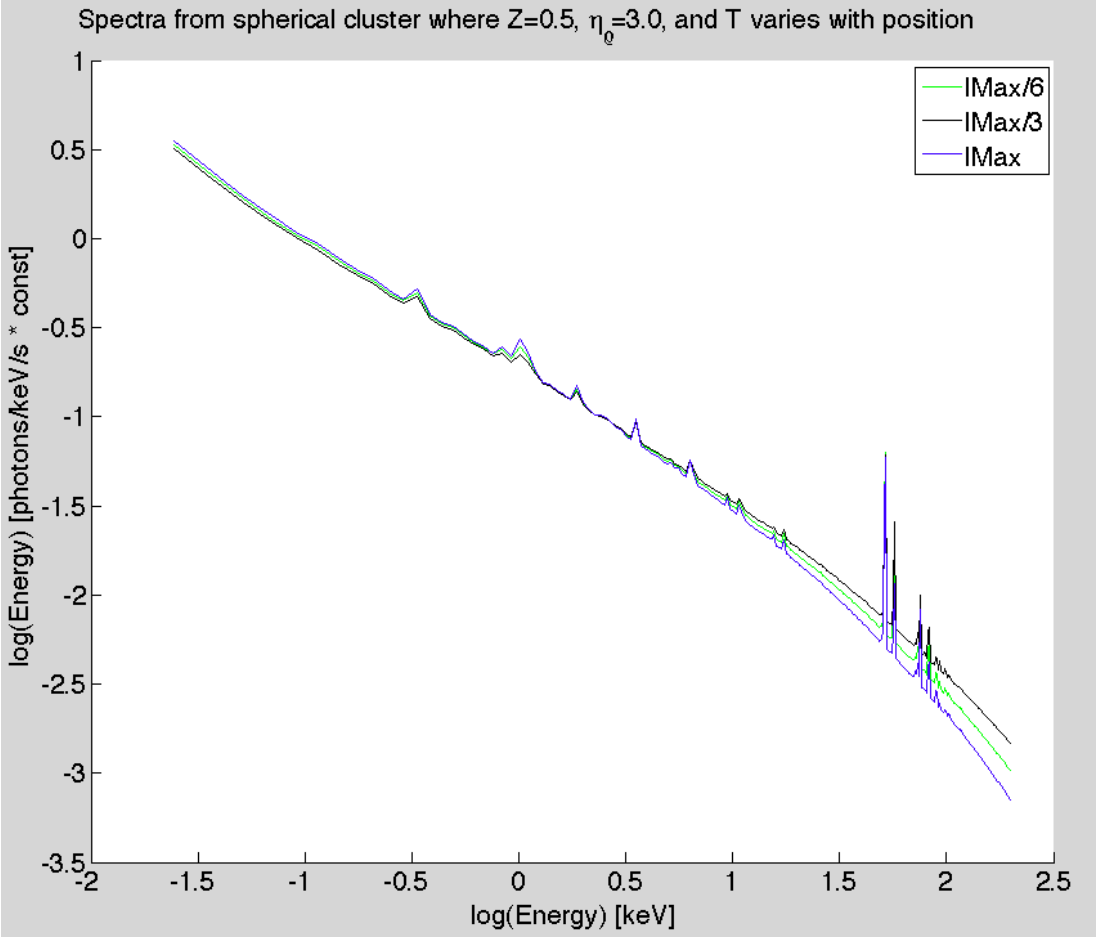

Figure 12: Spectra generated with a constant metallicity ($Z = 0.5$) and $\eta_\rho = 3.0$. This is also a spherical cluster where $A = B = 1.0$. Temperature is a function of l, so different locations within the cluster will have different calculated spectra. The blue spectrum is at the edge of the cluster where $l = lMax$, the black one is at $l = lMax/3$, and the green spectrum is at $l = lMax/6$.

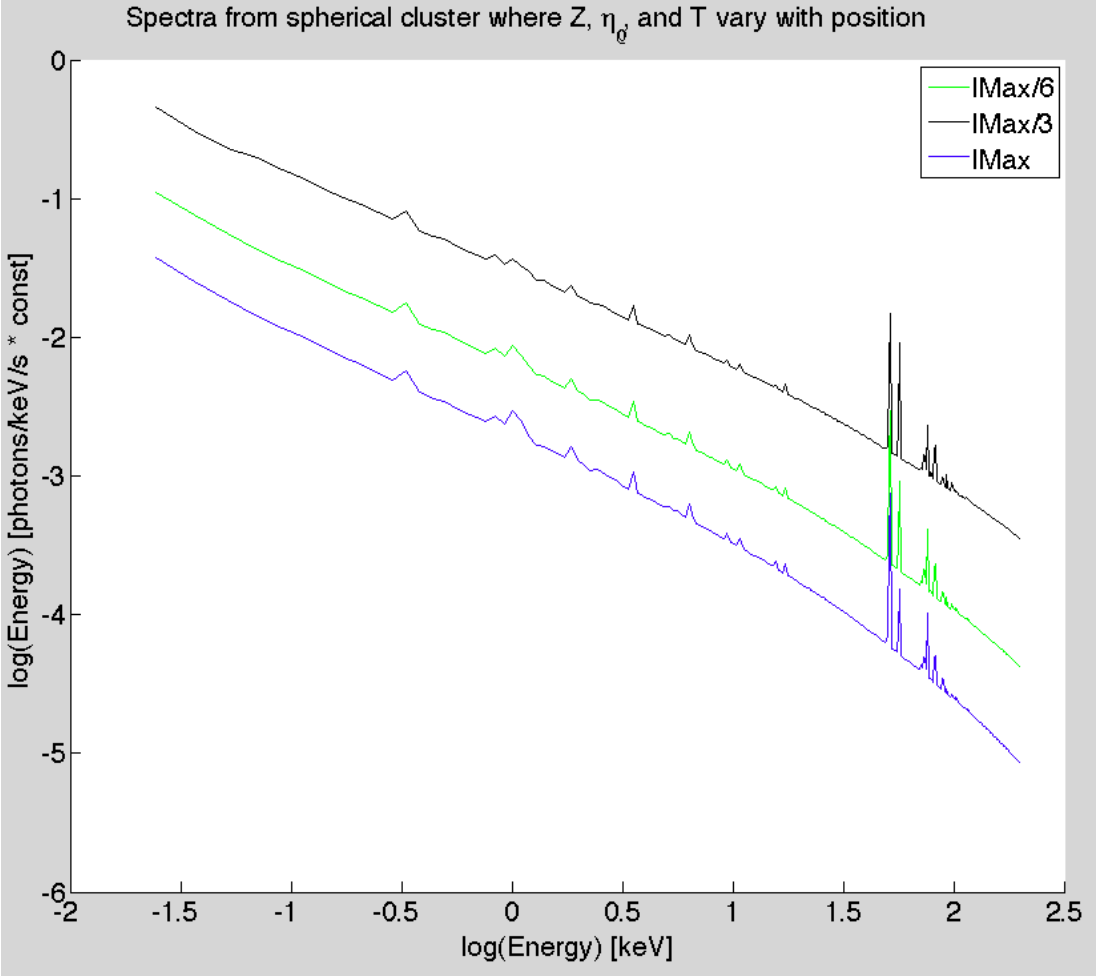

Figure 13: Spectra generated with metallicity, η_ρ, and temperature as a function of l, so they are all changing with respect to position in the cluster being graphed. This is spherical cluster with $A = B = 1.0$. The blue spectrum is at the edge of the cluster where $l = lMax$, the black one is at $l = lMax/3$, and the green spectrum is at $l = lMax/6$.

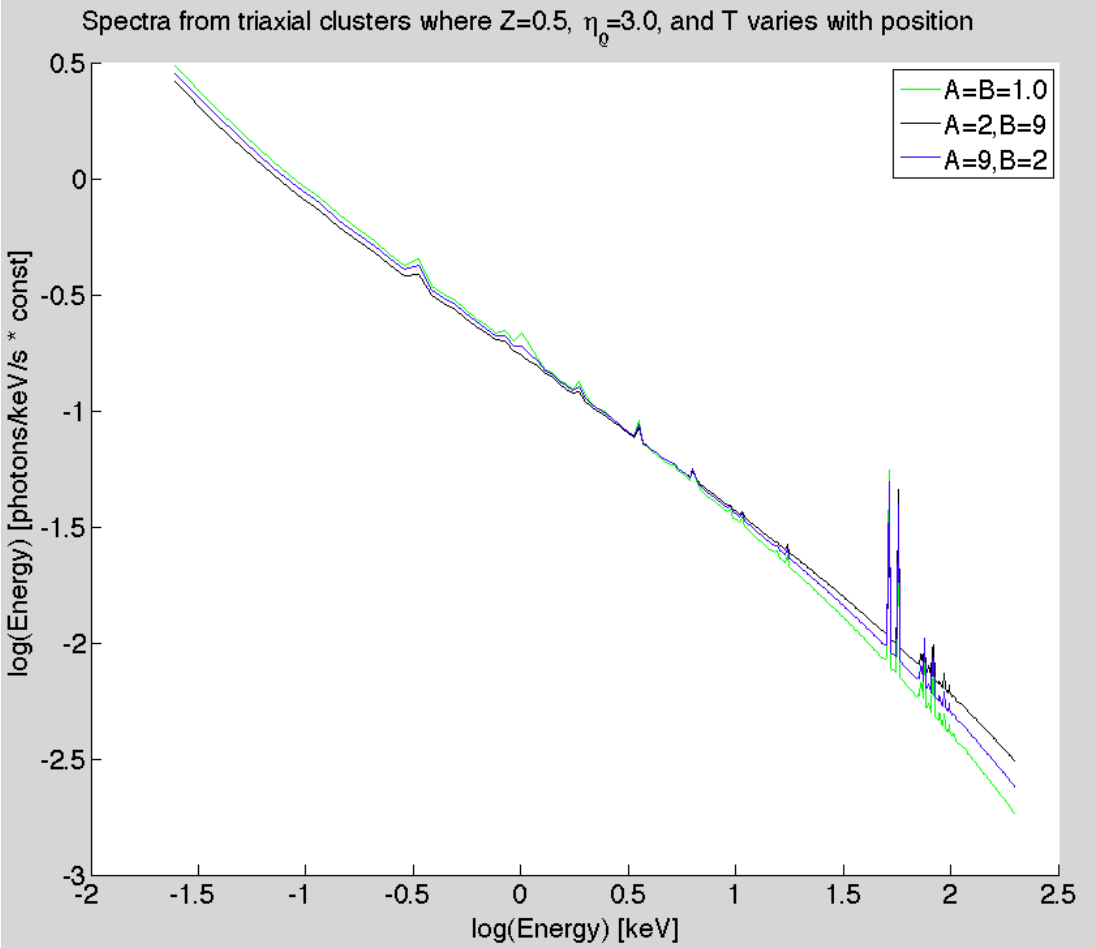

Figure 14: Each spectrum is calculated at the point where $X = Y = 64$ which is about half of the distance from the center of the cluster. The green spectrum is from a spherical cluster with $A = B = 1.0$. Black is from a triaxial cluster with $A = 2$ and $B = 9$, and blue is from a cluster with $A = 9$ and $B = 2$.

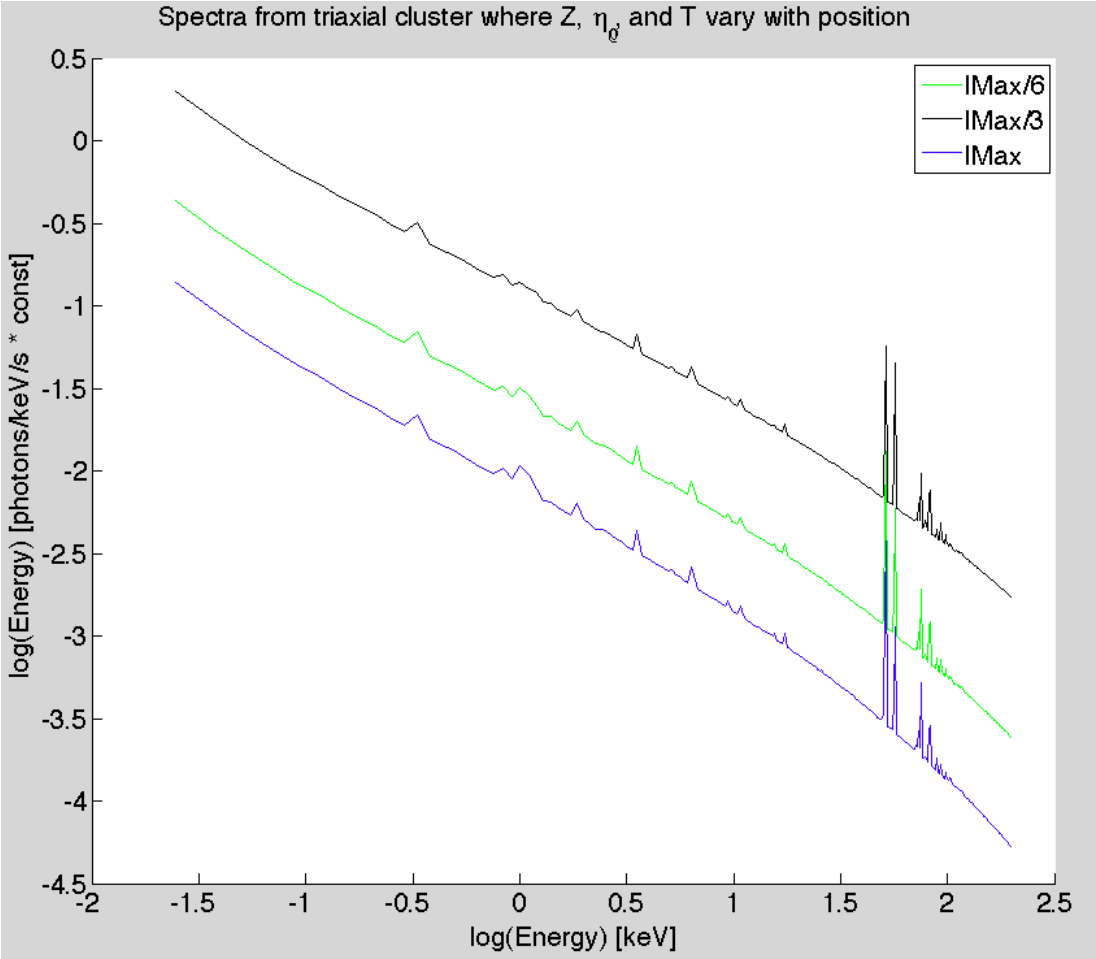

Figure 15: Spectra generated with metallicity, η_ρ, and temperature as a function of l, so they are all changing with respect to position in the cluster being graphed. This is a triaxial cluster with $A = 2$ and $B = 9.0$. The blue spectrum is at the edge of the cluster where $l = lMax$, the black one is at $l = lMax/3$, and the green spectrum is at $l = lMax/6$.

evaluate the orientation of the cluster it is easiest to sum up the flux at each coordinate in the cluster's grid. This results in a heat map image of what the simulated cluster would look like if observed by an orbiting X-ray telescope. The simulation was run with a range of values for ϕ, θ, and ψ inputted into the rotation matrix $\mathbf{A}$ defined in Equation 2.6. Figure 16 shows how the cluster rotates when θ is increased. This is a rotation around the X-axis, and the plots correctly show how this cluster would be oriented through the range of θ values. Figure 17 displays the cluster orientation when ψ is changed with $\phi = 0^o$ and $\theta = 90^o$. This map was generated with $A = 10$ and $B = 1$ to create an exaggerated shape that resembles a football. This shape is then rotated around ψ to orient the cluster along the X-axis, and then back to the Z-axis. These results are exactly as expected. Once the simulation passed all of these tests it was determined that it can accurately simulate any cluster that has valid input parameters.

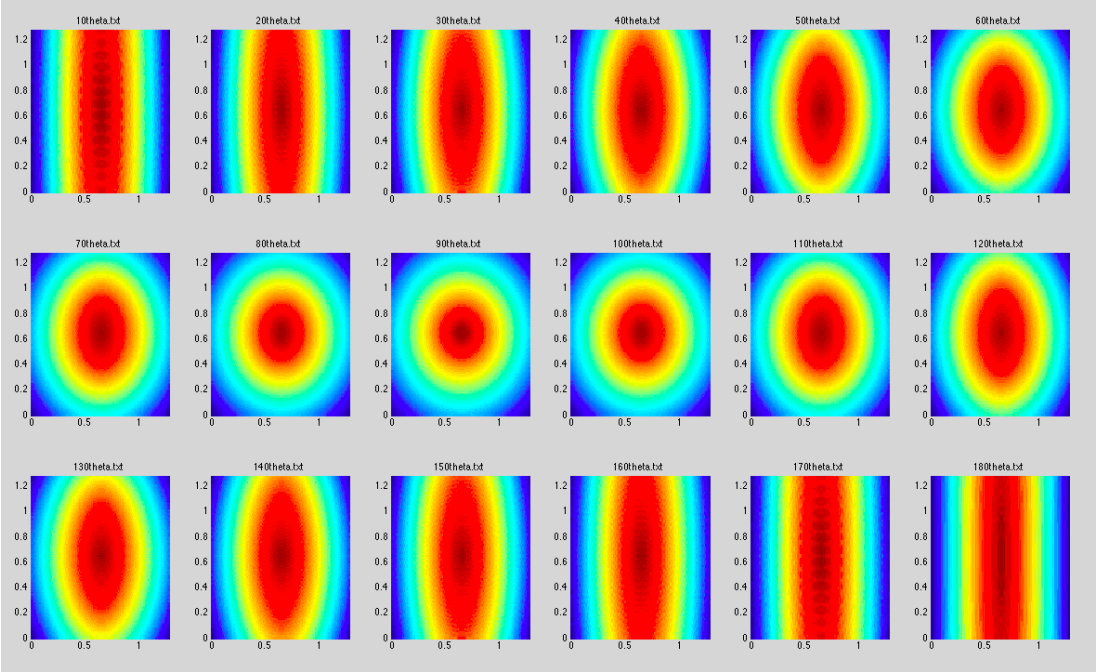

Figure 16: The total flux taken when θ ranges from $10^o - 180^o$ with $A = 10$ and $B = 1$. These values of A and B create a cluster that resembles a football. In the first plot the longest component is along the Y-axis. As θ increases the cluster rotates around the X-axis to be aligned on the Z-axis and then it returns to alignment along the Y-axis.

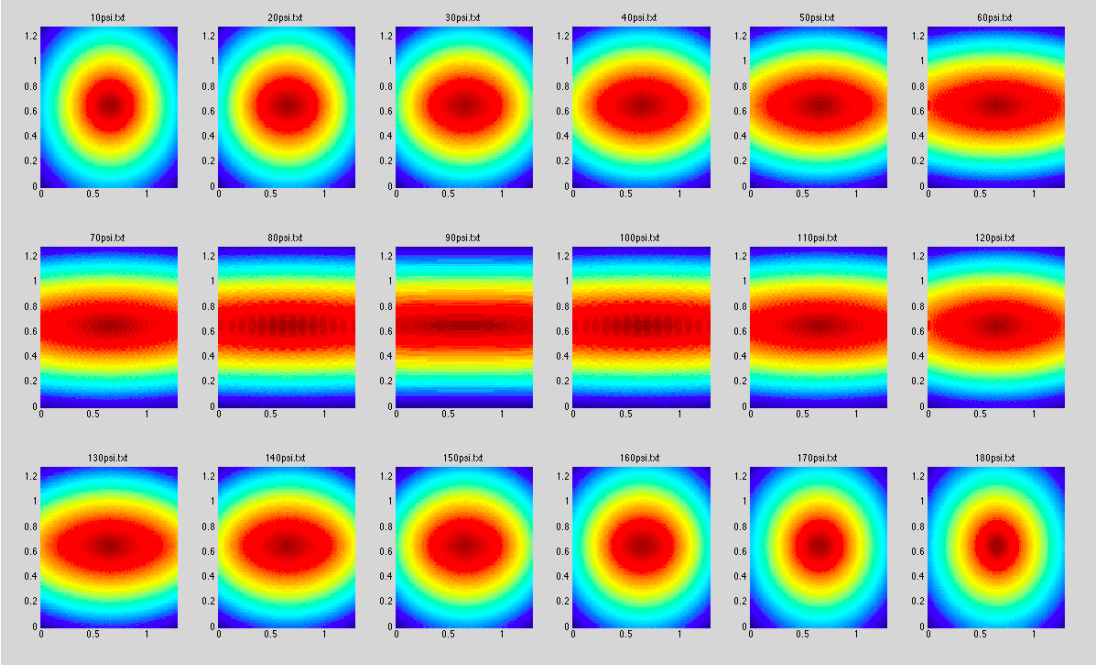

Figure 17: The total flux taken when ψ ranges from $10^o - 180^o$ with $A = 10$ and $B = 1$. These values of A and B create a cluster that resembles a football. As ψ increases the cluster rotates from head on to completely sideways, and back to head on.

7 Conclusion

This project has shown the viability of GPU computing to run a complex astronomical simulation. It has been clearly shown that there is a dramatic speedup when using massive multi-threading to simulate galaxy cluster spectra. The basic version of the simulation achieves a speedup of two orders of magnitude over the simulation run in series. For simulations with a larger XY-grid or more precise energy values, the difference in speed will only increase. Using the benefits of GPU computing, simulating triaxial shaped galaxy clusters can be completed in a reasonable amount of time. This allows for a more accurate representation of galaxy clusters that is needed for the study of dark matter contained within these clusters. With the tools provided by more complex simulations the astronomy community can get closer to understanding the unusual properties of dark matter.

8 Future Work

Future work will first involve extensive verification on the generated spectra with the shape parameters being set as $A = B = 1$, and different temperature and metallicity profiles. These spectra will have to agree with spectra generated from previous spherical simulations. Once these are verified to be correct the different profiles will be tested with various values for A and B. After all the verification has been completed the method will modified to include post-processing of the model spectra. This will take into account variations with instrument-specific spectral and spatial point-spread-functions. The method will also be modified to produce spectra that are closer to actual raw observations. This includes coadding the spectra from multiple points on the CCD to match the observers actual, signal-to-noise-based spatial binning.

9 Appendix A: Computing Environment

The GPU implementation was run on 2 Tesla C2050 chips. They each have 2818 MB of global memory, and 49252 bytes of shared memory. They both have a clock speed of 1.15 GHz. Tests on the CPU were run on an Intel Xeon E5520 processor with a 2.27 GHz clock speed. The code was compiled using CUDA version 3.10 with the gcc version 4.1.2. The operating system was Red Hat version 4.1.2-46. For the timing comparisons the CPU code was written in C/C++, and compiled using the gcc compiler.

References

1. Zwicky, F., On the Masses of Nebulae and of Clusters of Nebulae. 1937, ApJ, 86, 217

2. Mahdavi, A., Hoekstra, H., Babul, A., Sievers, J., Myers, S. T., Henry, J. P., Joint Analysis of Cluster Observations. I. Mass Profile of Abell 478 from Combined X-Ray, Sunyaev-Zel'dovich, and Weak-Lensing Data. 2007, ApJ, 664, 162

3. Fabricant, D., Rybicki, G., Gorenstein, P., X-ray measurements of the nonspherical mass distribution in the cluster of galaxies A2256. 1984, ApJ, 286, 186

4. Buote, D.A., Canizares, C.R., X-Ray Constraints on the Intrinsic Shapes and Baryon Fractions of Five Abell Clusters. 1996, ApJ, 457, 565

5. Kawahara,H., The Axis Ratio Distribution of X-ray Clusters Observed by XMM-Newton. 2010, ApJ, 719, 1926

6. Sayers, J., Golwala, S.R., Ameglio, S., Pierpaoli, E., Cluster Morphologies and Modelindependent Y SZ Estimates from Bolocam Sunyaev-Zeldovich Images. 2011a, ApJ, 728, 39

7. Soucail, G., Fort, B., Mellier, Y., Picat, J. P., A Blue Ring-Like Structure, in the Center of the A370 Cluster of Galaxies. 1987, AandA, 172, 14

8. Evans,A. K. D., Bridle, S., A Detection of Dark Matter Halo Ellipticity using Galaxy Cluster Lensing in the SDSS. 2009, ApJ, 695, 1446

9. Hernquist, L., An analytical model for spherical galaxies and bulges. 1990, ApJ, 365, 359

10. Tanikawa, A., Kohji, Y., Keigo N., Takashi O., Phantom-GRAPE: numerical software library to accelerate collisionless N-body simulation with SIMD instruction set on x86 architecture. 2012, ArXiv e-prints

11. Clark, M., La Plante, P., Greenhill, L., Accelerating Radio Astronomy Cross-Correlation with Graphics Processing Units. 2011, ArXiv e-prints

12. Schive, H., Tsai, Y., Chiueh, T., Analysing Astronomy Algorithms for GPUs and Beyond. 2010, ApJ. Supp., 186, 457

13. van Straten, W., Bailes, M., DSPSR: Digital Signal Processing Software for Pulsar Astronomy. 2010, ArXiv e-prints

14. Ford, E.B., Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units: Application to Analysis of Doppler Exoplanet Searches. 2008, NewA, 14, 406

15. Thompson, A. C., Fluke, C. J., Barnes D. G., Barsdell B. R., Teraflop per second gravitational lensing ray-shooting using graphics processing units. 2010, NewA, 15, 16

16. Bard, D., Bellis, M., Allen, M. T., Yepremyan H., Kra-tochvil J. M., Cosmological Calculations on the GPU. 2012, ArXiv e-prints