SIMULATION AND MODELLING OF GRAVITATIONAL MICROLENSING EVENTS USING GRAPHICAL PROCESSING UNITS

A thesis presented in partial
fulfilment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY
IN COMPUTER SCIENCE

at Massey University,
Albany (Auckland), New Zealand.

Cho Hong Ling

2013

Abstract

This thesis presents the results of a study into the use of graphical processing units (GPUs) in the simulation and modelling of gravitational microlensing. Two simulation approaches were investigated: magnification maps and the use of a dynamic engine for directly simulating gravitational microlensing light curves. It was found that the GPUs are able to speed up the generation of magnification maps dramatically. Very high performance in light curve extraction from magnification maps using GPUs is also achieved. Furthermore, the use of texture memory speeds up the extraction of light curves in a further 75% improvement in performance. They provide a speed up of over a 100× faster than CPUs in light curve simulations with finite source effects. The dynamic engine approach use a hybrid computation method with both CPUs and GPUs to simulate light curves for complex microlensing events. It allows us to model microlensing events with orbital motion effects, which are usually done on a cluster computer, on just a desktop computer with GPUs. Modelling strategies and optimization techniques are developed and applied to model different types of microlensing events.

GPU architectures show great promise for tackling the computationally expensive task of numerical modelling of microlensing events. With the modelling strategies developed here, microlensing modelling can be performed on a desktop computer at only a fraction of the cost of a cluster computer. The approach in this thesis provides a very cost-effective solution for the microlensing modelling challenge.

Acknowledgements

I would like to thank my supervisor Ian Bond and my co-supervisor Winston Sweatman for their guidance over the years. This thesis could not have been completed without their continued support and encouragement. I enjoyed the weekly meeting and interesting discussions of my research and life with them.

I would also like to thank everyone at the Institute of Information & Mathematical Sciences for their inspirational seminars, discussions and support.

My family means the world to me and I could not have completed my degree without them. Special thanks to my wife Linh for her selfless support and sacrifices over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The phenomenon of gravitational lensing is a consequence of Albert Einstein's Theory of General Relativity. He predicts the brightness of a distant star will be magnified when a massive object, like a star, passes between the observer and the distant star. The change in brightness of the distant star is caused by the gravity of the massive object distorting spacetime and causing light from the distant star to warp around the massive object and create an effect like an optical magnifying glass. This is such an elegant insight by Einstein. However, Einstein did not expect we would be able to observe this microlensing phenomenon, let alone use this effect to discover distant new worlds.

Gravitational lensing has been observed on a large scale with many observations of images of distant galaxies that have been lensed by foreground galaxies. A paper [1] written by Einstein in 1936 describes the lensing of one star by another. This is what is known as gravitational "microlensing" [2, 3]. The probability was so low that Einstein wrote that there was "no chance of ever observing this phenomenon".

Microlensing has a probability of occurring, at one in a million stars per year and the change in brightness is almost indistinguishable by human eyes. It is almost impossible to discover microlensing events by just observing the sky using our eyes and keeping track of the brightness of millions of stars at once. However the invention of the CCD (charge-coupled device) allows us to do just that. Not only does the CCD allow us to capture millions of stars all at once at any given moment but it also allows us to record the brightness of a star in high precision.

Once we are able to discover a microlensing event using a CCD, we further use microlensing as a tool to discover extra-solar planets. The precise measurement of the brightness of

a star allows us to utilize the potential of microlensing. Since there are deviations to the single lens light curve when the massive object has one or more companions, we are able to re-construct a model that represents the physical star's configuration using a computer that matches with the observed data. It is an elegant way to discover distant new worlds as all we record is the change in brightness of a star. In [4, 5, 6], microlensing is proposed as a planet detection technique. Microlensing even has the sensitivity to discover an Earth mass planet [7] as well as extra-solar planets that have wide orbits. It is also able to detect multi-planetary systems in our galaxy very far away from Earth. At the time of writing, there have been 14 published discoveries of extrasolar planets following the first extra-solar planet discovery by microlensing [8, 9].

Although microlensing is an elegant and powerful tool to detect extra-solar planets in our galaxy, it also presents computational challenges. A typical multi-lens microlensing event is described by a model with more than ten parameters. In order to re-construct a model that represents the physical configuration of the microlensing system, it is necessary to search in multi-dimensional parameter space. Furthermore, the calculation of each microlensing model is very computationally expensive and makes the modelling process even more challenging.

For example, a typical modelling of a binary microlensing event require months in computation time for a full grid search in all parameter space on a desktop computer. The challenge in microlensing modelling is not just to discover the correct model to match with the observed data, but also to prove that there are no other models in the parameter spaces that have a better fit to the data. Moreover, there are degenerate models that are very similar and require extra care when performing microlensing modelling. This is the very reason that makes microlensing modelling such a challenge in both computation and optimization parameter searches.

Microlensing modelling is often performed on a cluster computer because of these computational challenges. Although it is a problem that can be solved by using more computational resources, the problem size presents a significant barrier to people without or with limited access to cluster computing to start microlensing modelling.

The idea of using highly parallel graphics processing units (GPUs), used for displaying 3D graphics, is not new. However, they are very hard to program as all the data must be transformed into pixels and computed by graphics operations. Since the introduction of CUDA (Compute Unified Device Architecture) by NVIDIA in 2006, the scientific community has started to embrace GPUs for their scientific computations as they provide significant

speedup over the central processing units (CPUs) in suitable problems. Since part of the microlensing modelling calculation is repetitive and can be done in parallel, it is very suitable to perform it on a massively parallel processor like the GPU.

We are able to speed up the microlensing modelling computation from months to days just by using a desktop computer with a GPU. Since most computers nowadays have a powerful GPU for displaying 3D graphics, the cost of such a device is very low but provides very high performance for microlensing modelling computations. The cost to performance ratio is higher than any other computation resources in the past and is significantly cheaper than running a cluster computer. The GPU microlensing modelling method presented in this thesis provides a very cost effective solution for the microlensing modelling challenge.

There are generally two approaches to simulating the microlensing effect. One involves the use of "magnification maps". Chapter 4 discusses the application of GPUs to this method. The other approach is to directly generate microlensing amplification profiles. Chapter 5 discusses a hybrid light curve calculation engine using both CPUs and GPUs for more complex microlensing modelling. Chapter 6 presents case studies of different types of microlensing events and applies the modelling methods developed in Chapter 4 and 5. It also presents our modelling strategies and optimization techniques. Chapter 7 concludes the thesis and discusses future forthcoming GPU technologies.

Chapter 2

Gravitational Microlensing and Current Challenges in Microlensing

2.1 Introduction

A gravitational microlensing event occurs when a massive object, the "lens", moves across the line of sight between the observer and a distant "source" star. The brightness of the distant star is magnified and produces a well-recognized symmetric bell-shaped light curve. When the lens star has one or more orbiting planets, additional lensing can occur and produces extra signals in the light curve. We can determine the physical characteristics of the lens system, like the mass fraction and projected separation of the planets, by constructing a model, using a computer, that matches with the observation data. Gravitational microlensing has emerged as a new technique for finding extrasolar planets and a great deal of data has been collected from the MOA (Microlensing Observations in Astrophysics) and other projects.

Since the first robust detection of an extrasolar planet by the microlensing technique [8], this technique has matured as a way of discovering extrasolar planets. To date there are 14 published discoveries of extrasolar planets found by microlensing [9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27], including a super-Earth [25], a super-Earth orbiting

a very low mass star [20] and a multiple planet system. Microlensing events also allow us to produce useful statistics on the abundances of extrasolar planets [28, 29, 30, 31, 32]. Microlensing has also been used to detect free floating planets in our galaxy and was able to produce statistics about their abundance [33]. Microlensing can also be used to study the surface of distant stars that cannot be seen by direct observation techniques [34]. It is also proposed as a tool for searching for black holes [35].

As a result of the computational challenges outlined in Chapter 1, there are a number of microlensing events with possible planetary signals that have not yet been published. This backlog goes back to 2002.

2.2 Simple microlensing

Microlensing occurs when a lens star passes very close to the line of sight of the observer and a distant star. The simplest form of microlensing is a single source star with a single lens star configuration. Figure 2.1 (a) shows the basic geometry of microlensing. The observer O and a point lens L are separated by D_l, and D_s is the distance between the observer and the source, S. The deflection angle is

$$\hat{\alpha_d} = \frac{4GM}{r_E c^2} \tag{2.1}$$

where r_E is the Einstein radius. The location of the source is displaced from the true position S to the image position I by an angle θ_E while $\hat{r}_E$ is the projected Einstein radius. The relationship between θ and the angular separation β is given by $\beta = \theta - \alpha_d$. From small angle approximation, $\hat{\alpha}_d(D_s - D_l) = \alpha_d D_s$, and from the exterior-angle theorem, $\theta_E = \frac{\hat{r}_E}{D_l} - \frac{\hat{r}_E}{D_s}$. Therefore,

$$\beta = \theta - \frac{4GM}{r_E c^2}\frac{D_s - D_l}{D_s} \tag{2.2}$$

For a perfect alignment between the lens and the source ($\beta = 0$), a ring like image called the "Einstein ring" is formed with radius

$$\theta_E = \frac{4GM}{r_E c^2}\frac{D_s - D_l}{D_s} \tag{2.3}$$

Given $\theta_E \equiv r_E/D_l$, the absolute radius of the Einstein ring is

$$r_E^2 = \frac{4GM}{c^2}\frac{(D_s - D_l)D_l}{D_s} \tag{2.4}$$

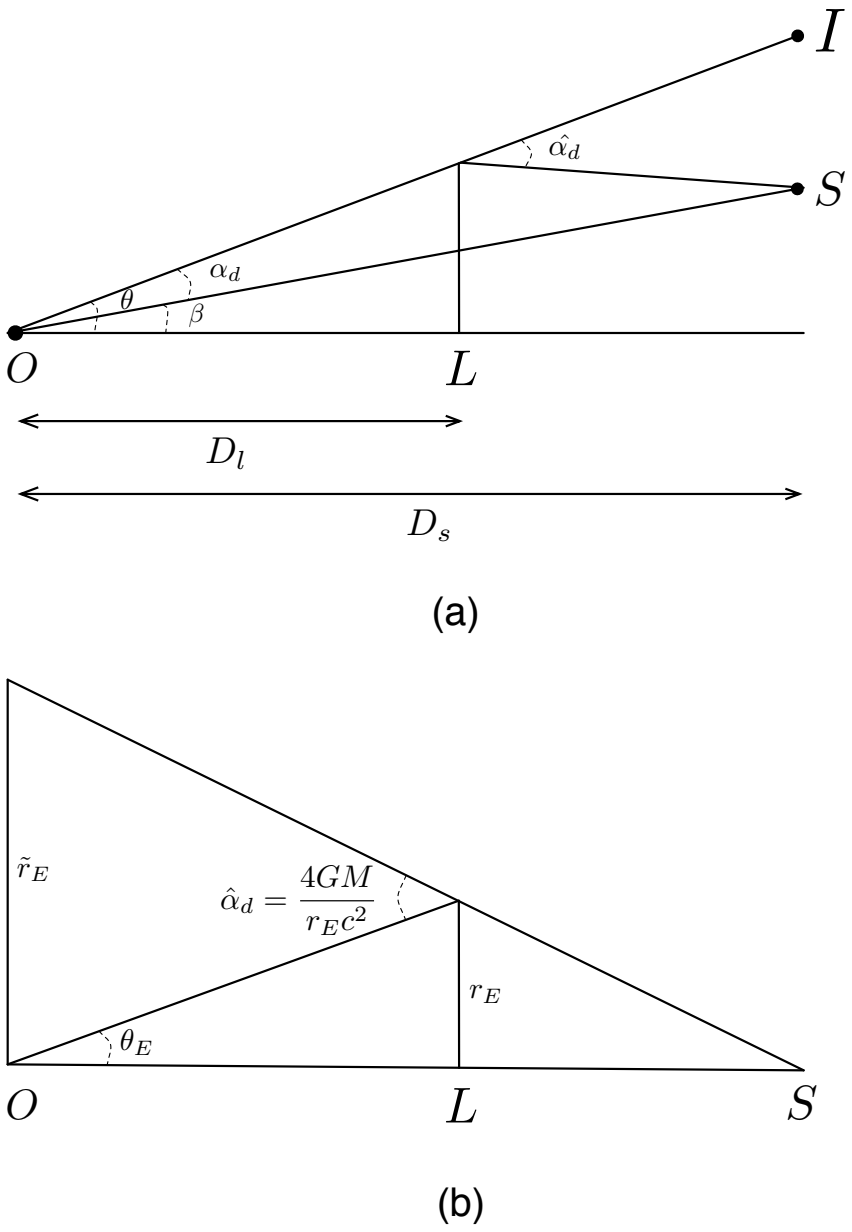

Figure 2.1: Microlensing geometry. The lens L at a distance D_l from the observer O deflects light from the source S and forms an image I with deflection angle $\hat{\alpha}_d$. The relationship between the angle position of the source β and the image θ is $\beta = \theta - \alpha_d$. For a point lens, the deflection angle is $\hat{\alpha_d} = \frac{4GM}{c^2 D_l \theta}$.

Using convenient units of solar masses, $M_\odot$, kiloparsecs, kpc, and Astronomical Units, AU, the Einstein radius can be expressed

$$\frac{r_E}{\mathrm{AU}} = 2.85\left(\frac{M}{M_\odot}\right)^{1/2}\left(\frac{D_l}{D_s}\right)^{1/2}\left(\frac{D_s - D_l}{\mathrm{kpc}}\right)^{1/2} \tag{2.5}$$

Normalizing all angles on the sky by θ_E, we can define $u = \beta/\theta_E$ and $y = \theta/\theta_E$. The lens equation 2.5 reduces to the form

$$u = y - y^{-1} \tag{2.6}$$

A quadratic equation can be written using the above equation: $y^2 - uy - 1 = 0$. In the case of imperfect alignment ($u \neq 0$), there are two images, with positions

$$y_\pm = \pm\frac{1}{2}(\sqrt{u^2+4} \pm u) \tag{2.7}$$

The positive ("major") image is always outside the Einstein ring while the negative ("minor") image is always inside the Einstein ring. The magnification of each image is the ratio of the area of the image to the area of the source. The amplification can be calculated by

$$A_\pm = \left|\frac{y_\pm}{u}\frac{dy_\pm}{du}\right| = \frac{1}{2}\left[\frac{u^2+2}{u\sqrt{u^2+4}} \pm 1\right] \tag{2.8}$$

Thus, the total amplification of a point-source single-lens system is described by

$$A(u) = \frac{u^2+2}{u\sqrt{u^2+4}} \tag{2.9}$$

where u separation of the source and lens in units of the Einstein radius. This radius is that of the Einstein ring when the source, lens and observer are in perfect alignment. Note that, as $u \to \infty$, $A_+ \to 1$ and $A_- \to 0$. Also, for $u \ll 1$, the amplification takes the form $A \simeq 1/u$. The amplification becomes infinite when u is zero. However, this infinity is broken by the finite size of the source star. For uniform rectilinear motion of the source star relative to the lens, their separation as a function of time is given by

$$u(t) = \left[u_0^2 + \left(\frac{t - t_0}{t_E}\right)^2\right]^{1/2} \tag{2.10}$$

where t_0 is the closest impact time, t_E is the time scale to cross the angular Einstein ring radius and u_0 the closest impact distance of the source star [36].

Figure 2.2 shows a sample of the family of theoretical light curve profiles of a point-source single-lens microlensing event with different values of u_0, the closest impact parameter,

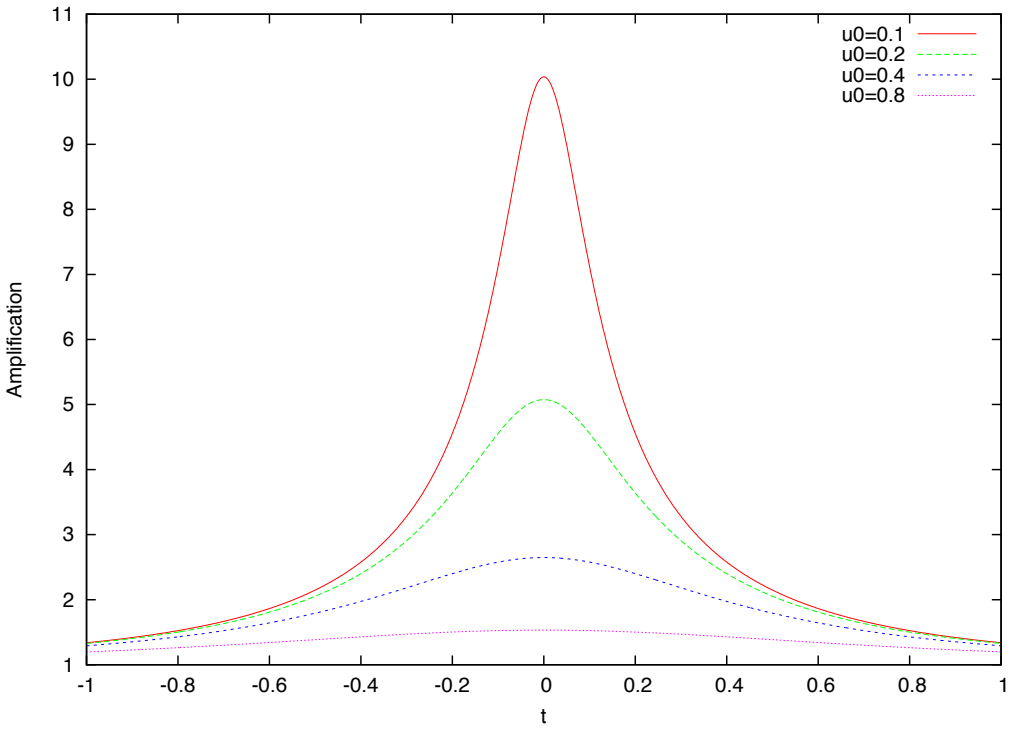

Figure 2.2: Microlensing light curves with single lens point source of $u_0 = 0.1, 0.2, 0.4, 0.8$. t is the normalised time unit.

described by Equation 2.10. This simplest form of microlensing event produces a symmetric bell-shape light curve profile. The shape of the light curve is only affected by u_0 in a point-source single-lens model and it determines the peak amplification of the light curve profile.

For a point source, point lens (PSPL) microlensing event, it is fairly straight forward to use standard non-linear techniques (for example in Numerical Recipes [37]) to find the optimal set of parameters to fit a set of observational data of the event. Although a single-lens microlensing model is relatively easy to model, some microlensing events are far more complicated and often the right model is obtained by the combination of a multitude of other microlensing effects.

The single-lens microlensing light curve is a bell-shape-like light curve and is symmetrical. This classical light curve is easy to spot and determine by the single lens model. Once we depart from a single-lens point-source model, a microlensing event becomes much harder to model as one cannot simply write down an analytic expression for the light curve. For example, Figure 2.3 shows a theoretical light curve for a binary-lens finite-source microlensing

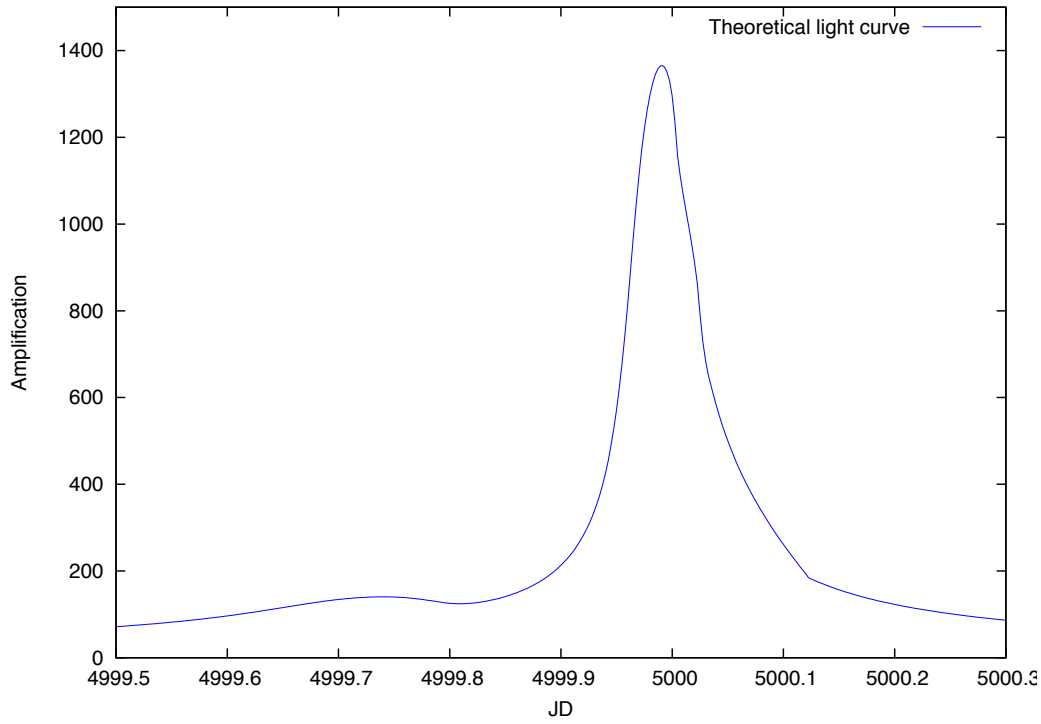

Figure 2.3: Theoretical light curve of a binary-lens finite-source microlensing event.

event. Compared with the single lens model in Figure 2.2, its shape is clearly different from the symmetric bell-shape light-curve profile. The source star in this model moves across the caustic and the source star has a finite size instead of being a point source. The theoretical light curve in Figure 2.3 is computed numerically and is described by several parameters, including the mass fraction and projected separation of the companion, size of the source star, and angle of the source-star trajectory.

A seven-parameter model is the simplest form for a binary-lens finite-source microlensing event. A typical binary microlensing event often requires searching parameter space with more than ten parameters. Moreover, microlensing events become even harder to model when multiple effects are considered, like multiple lenses, finite-source effects, non-rectilinear motion of source stars and non-static multiple-lens systems. The combination of multiple effects makes generation of microlensing models very computationally expensive and also challenge the use of multi-dimensional parameter optimization methods.

2.3 Multiple lenses

The single lens system is straight forward to describe as in Equation 2.10 with relative motion of the source and lens in the line of sight of the observer. We need to define a two-dimensional lens plane and a source plane that is perpendicular to the line of sight between the observer and the source star. The mass fraction, ε, the mass divided by the combined mass of all lenses, is used to define the mass of the lens objects. It is also convenient to describe the lens equation using complex coordinates. The following equation denotes the positions of the source and image positions by ω and z respectively. The multi-lens system can also be described by the lens equation [38, 39].

$$\omega = z - \sum_j \frac{\varepsilon_j}{\overline{z} - \overline{z}_{m,j}} \tag{2.11}$$

where ω and z are the complex positions of the source and image, respectively, ε, is the mass fraction of the jth lens mass, and $z_{m,j}$ are the complex positions of the lens masses.

The lens equation allows us to solve the position of the source ω given that we know the position of the images z. However, we usually only know the position of the source but not the images, therefore it is an inverse problem.

2.3.1 Inverse ray shooting

The amplification of any point on the source plane can be obtain by using the Jacobian in solving the lens equation 2.11. The amplification of each image is given by

$$A = \frac{1}{|J|} \tag{2.12}$$

and the Jacobian determinant of the inverse mapping by the lensing equation from the image plane to the source plane

$$J = \frac{\partial w}{\partial z}\frac{\partial \overline{w}}{\partial \overline{z}} - \frac{\partial w}{\partial \overline{z}}\frac{\partial \overline{w}}{\partial z} = 1 - \left|\frac{\partial w}{\partial \overline{z}}\right|^2 \tag{2.13}$$

where,

$$\frac{\partial w}{\partial z} = \sum_j \frac{\varepsilon_j}{(\overline{z} - \overline{z}_{m,j})^2} \tag{2.14}$$

The position of the images can be solved by inverting the lens equation. For a binary lens, the lens equation becomes a 5th-order complex polynomial equation [39], and the lens

equation becomes a 10th-order complex polynomial equation for a triple lens [40]. Although we are able to compute the amplification by solving the lens equation with its inverted complex polynomial, the amplification across a caustic curve is undetermined as it is infinite. In most microlensing events, we need to take finite-source effects into account which cannot be calculated just by solving the inverted lens equation.

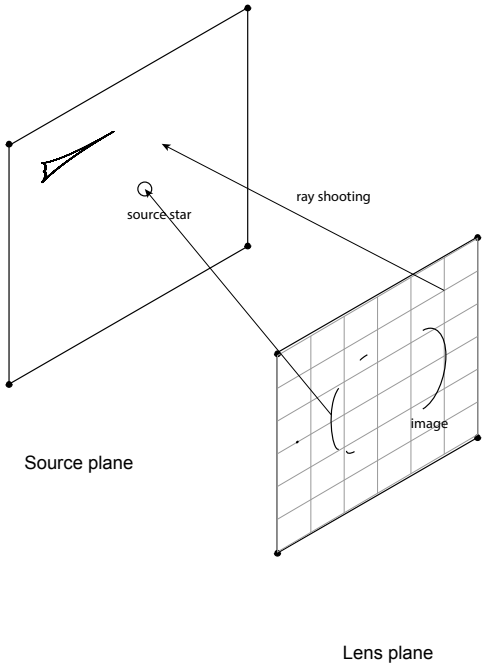

Figure 2.4: Inverse ray shooting.

The inverse ray-shooting technique can be used to avoid the problems above. The amplification of a source can be computed by solving the lens equation numerically. We first set up a lens plane and source plane with the lens plane big enough to cover the area of the image positions that are the direct mapping to the source plane. When lots of rays are shot from the lens plane to the source plane by the lens equation, some rays land on the source star on the source plane. The source star area is acting as a bin and collects rays that land within the source area. When an equal density of rays are shot from the lens plane, the number of rays collected in the source-star bin represents its amplification.

Figure 2.4 shows inverse ray shooting from the lens plane back to the source plane. The circle on the source plane represents the source star and the solid line of the lens plane represents the corresponding image area of the source star with direct mapping by the lens equation. The image area is also affected by the lens configuration and the size of the source

star. Many rays are shot from the lens plane to the source plane. The rays shot from outside the image area will fall on the source plane that is outside of the source star while rays shot within the image area will land within the source star. The image area is usually a long and thin arc-like area with discontinuous sections depending on the source position and the lens configuration. It is inefficient to shoot rays from the whole lens plane. It makes sense to first solve the image area then shoot rays from the image area. But unlike the point source approximation in which the point image locations can be computed by solving the complex polynomial equation, the image area of a finite source star can only be solved numerically by the lens equation which shoots rays form the lens plane to the source plane. Therefore, in order to compute the amplification of a finite source efficiently, we need to shoot rays as well as compute the image area efficiently. We solve this problem using two different approaches which are described in Chapter 4 and Chapter 5.

2.3.2 Critical and caustic curves

For a point-source binary-lens configuration, the amplification become infinite on those source positions where $J = 0$ in Equation 2.12 while the amplification close to those positions is large. From Equation 2.13, the image positions when $J = 0$ are given by,

$$\left| \sum_j \frac{\varepsilon_j}{\left(\overline{z} - \overline{z}_{m,j}\right)^2} \right|^2 = 1 \tag{2.15}$$

Note that the sum of the above equation is equal to unity. We can solve the following equation parametrically to obtain the critical image position,

$$\sum_j \frac{\varepsilon_j}{\left(\overline{z} - \overline{z}_{m,j}\right)^2} = e^{i\phi} \tag{2.16}$$

where $\phi = [0, 2\pi)$.

The set of image positions defined in Equation 2.16 form a closed curve called a critical curve. The caustic curves are created by the direct mapping from a set of image positions (the critical curves) onto the lens plane by the lens equation. In inverse ray shooting, the caustic curves are mapped by shooting rays from the critical curves on the lens plane to the source plane. Rays shot from the critical curve positions land on the caustic image positions. Figure 2.6(a) shows the critical curves (images positions on the lens plane) and their direct mapping (caustic curves) on the source plane.

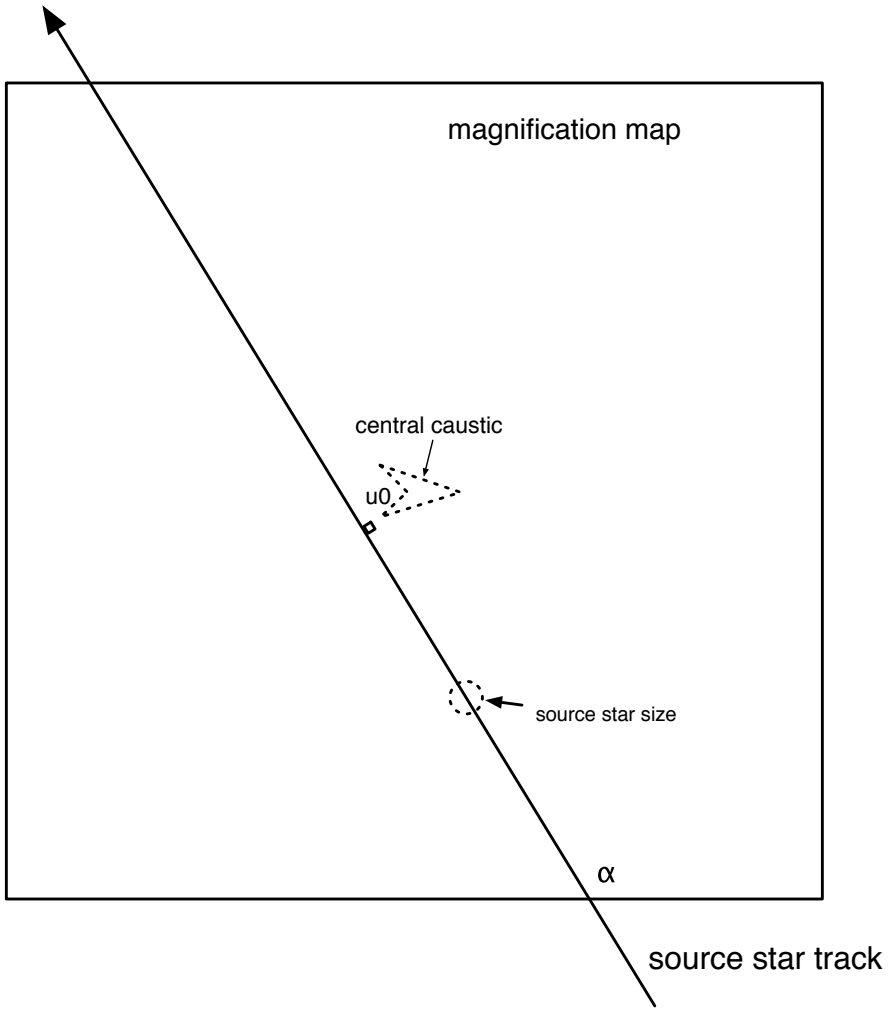

Figure 2.5: Schematic of the source star track and the caustic.

Figure 2.5 shows a schematic of the source star in rectilinear motion approaching a caustic. The impact parameter u_0 defines the closest distance of the source and the centre of the observer-lens line of sight and ϕ is the angle of the source star trajectory.

2.4 Higher-order effects

In almost all microlensing events with a multiple lens configuration, simulation of finite-source effects are required in order to construct a model that will match with the observation data. The size of the source star is described by the parameter, ρ, which is the ratio of angular

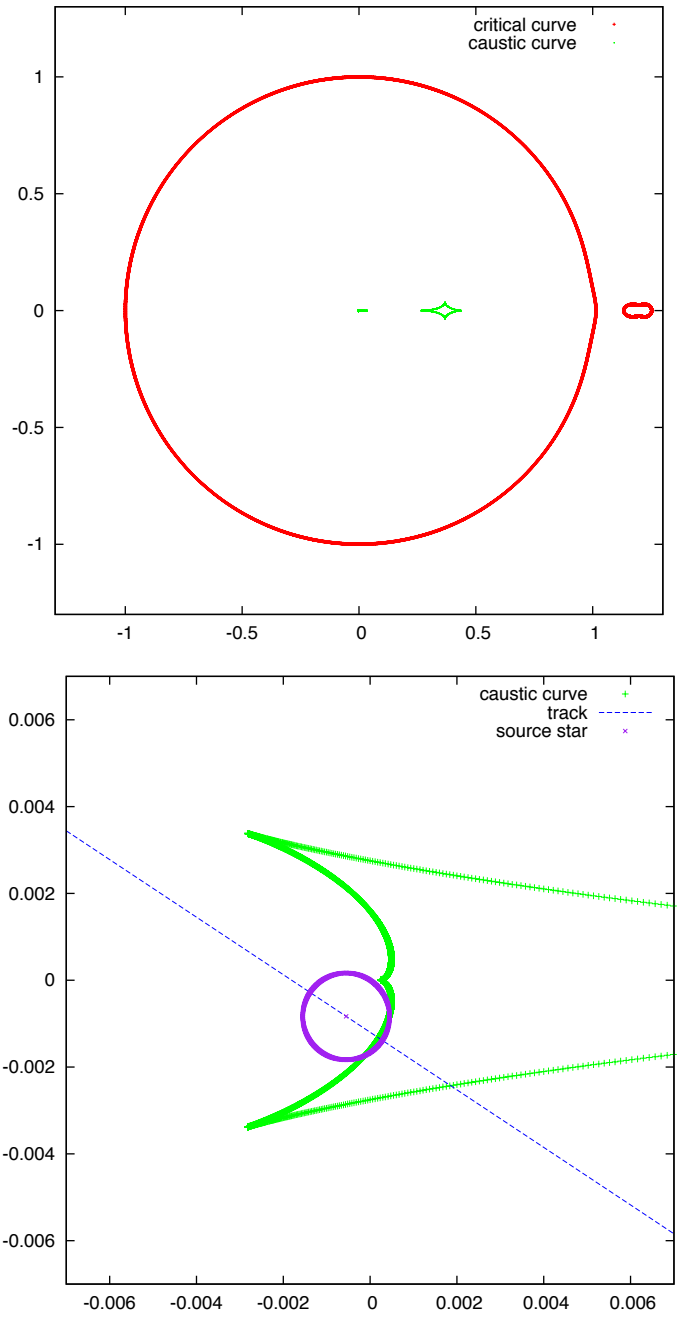

Figure 2.6: Critical and caustic curves for a binary lens with $\varepsilon = 0.001$ and $d = 1.2$. The top figure (a) shows the critical curve (red line) and the caustic curve (green line) of a lens system. The bottom figure (b) shows the caustic curve in more detail. For a point source, the amplification is infinite at the caustic curve.

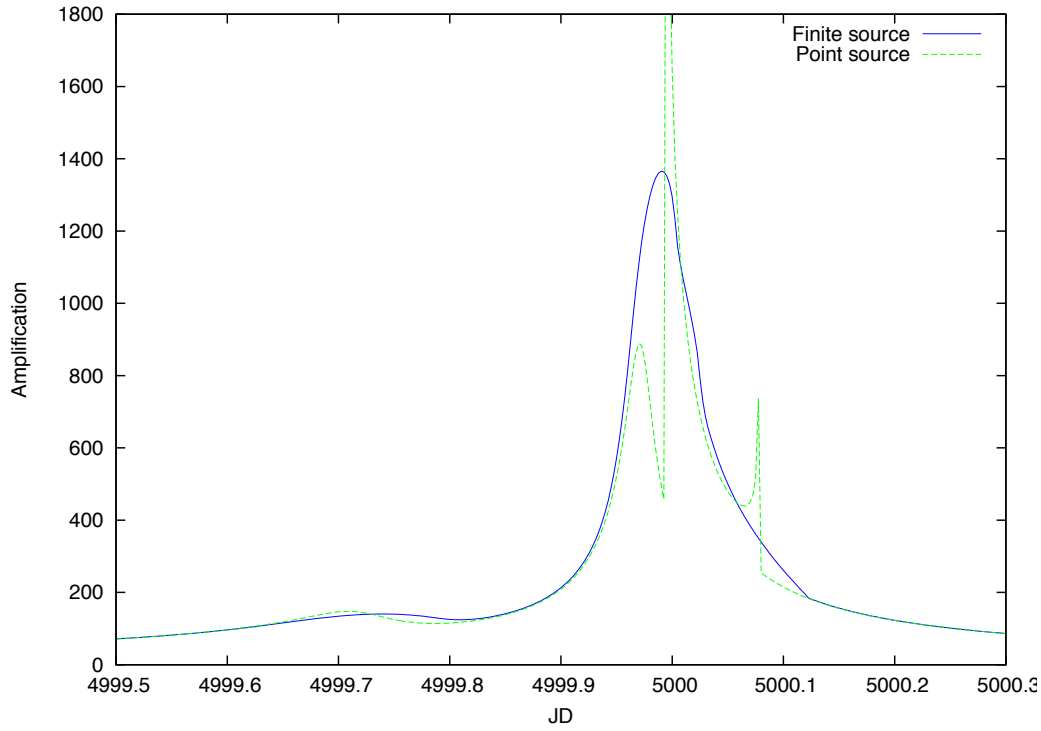

Figure 2.7: Light curve comparing the effects of a finite source and a point source. This figure demonstrates the effect on the theoretical light curve simulation by the finite-source effect. Some features in the point source light curve have changed completely - the spikes in the point source light curve disappearing or being washed out. In this illustration, the finite source effect is pronounced. This figure has the same lensing geometry as in Figure 2.6

size of the source star to angular size of the Einstein ring. Figure 2.7 shows the effect of the finite-source effect on a light curve.

The finite-source effect washes out the features of a light curve and can change the shape of the light curve completely. It often changes the shape of the light curve to a point that even experienced modellers find it hard to distinguish the orientation of caustic entry/exit. It can also hide the planetary signal in some cases as in [41]. The finite-source effect is also computationally expensive to calculate but is often required in multi-lens microlensing modelling. A highly efficient method to compute the finite-source effect using GPU is discussed in Chapter 4.

The other effects include parallax which is caused by the orbital motion of Earth [42]. The motion of the Earth around the Sun causes the alignment of the observer to the source to change slightly and affect the amplification profile. These parallax effects often change

the source-star track from a straight track to a curvy track, which can alter the shape of the light curve in a event with a long crossing time. The inverse effect of parallax is "xarallap", which is due to the orbital motion of the source star. This can have a dramatic effect on the light curve in some cases. A microlensing event is presented in Chapter 6 which shows dramatic effects of xarallap.

There are further effects that are even harder to compute in microlensing modelling like the orbital motion of the lens objects. Orbital motion is very expensive to compute and is often the most challenging effect to model in microlensing modelling. However it is sometimes essential to include these effects in the modelling of events. This makes them very challenging to model. An effective method to compute orbital motion effects using a hybrid method on the CPU and GPU is presented in Chapter 5.

2.5 Computational challenges in microlensing modelling

The previous sections discussed the various effects in microlensing event that are often essential to be included in the modelling process. However, they are also very computationally expensive to compute. A typical microlensing event with finite-source effects can often take months of computing time for a desktop computer to complete. Not to mention the added higher order effects like the orbital motion which extend the computing time to even longer.

For a high workload microlensing modelling case, it takes a 2.93GHz Core i7 CPU 143 days to complete the computation while a desktop computer equipped with a 480GTX GPU takes only 28 hours to complete the same computation which is 122× faster. Furthermore, the latest GPU Titan is around 4-5 times faster than the 480GTX GPU, multiple GPUs can also be added to the desktop computer. The latest GPU allows us to perform microlensing modelling in hours instead of months on a desktop computer.

2.5.1 Multi-dimensional parameter space

A typical microlensing modelling procedure searches in multi-dimensional parameter space to locate the minima of some objective function of these parameters. The surface of this function can be a highly complex topography of hills and valleys. It is often required to break down a part of the parameter space and strategically search for the correct model. The biggest challenge in microlensing modelling is not just searching for the correct model, but

also proving that there are no other better-fitting models existing in the parameter space. A systematic approach is required to discover all the local minima and identify the degenerate models. Chapter 6 discusses in detail the different strategies in microlensing modelling.

2.5.2 Finite-source effect

In a typical microlensing modelling, most of the computation time is spent on computing the finite-source effect. This is one of the most computationally expensive effects but it is also one of the most common effects in microlensing events. It is often essential to include this in modelling for multi-lens events, especially caustic crossing events, because finite source effects can alter the character of light-curve deviations significantly.

One solution for computing finite-source effects effectively is to use magnification maps [43]. A magnification map is essentially the collection of solutions of the lens equation for a particular lens configuration. Therefore, the computed solutions can be reused to generate multiple light curves in a modelling process. However, the magnification map generation is a very computationally expensive task as the lens equation is being calculated millions of time for each magnification map generation.

Chapter 4 discusses magnification map generation and track extraction from the magnification map using GPUs. The GPU solution significantly speeds up the magnification map calculations, and thus solves the computational challenge for finite-source calculation. We are able to speed up the computation more than 100× by using the GPU which also provides a very cost-effective solution for microlensing modelling.

2.5.3 Orbital motion in modelling

The other most challenging task in microlensing modelling is to compute the orbital motion effect. For a multi-lens system, the lens objects are moving around the centre of gravity in orbital motion. Therefore, the lens equation used to compute the amplification for any given point in time is different as the projected separation of lens objects used in the lens equation is changing with time. The magnification map is no longer useful when the orbital motion effect is considered in modelling, thus the finite-source effect computation also cannot be computed effectively.

Chapter 5 discusses a hybrid method using both CPU and GPU to compute the orbital motion effect in microlensing modelling. The computation for CPU includes solving the

complex polynomial of the lens equation and searching for the image area dynamically while the GPU computes the ray shooting using the lens equation in highly parallel fashion. Both the CPU and GPU are used to work on computation that suits its strengths to provide an effective solution for orbital motion calculation.

Chapter 3

GPUs as potential tools for microlensing modelling

3.1 Introduction

Scientific computing usually involves complex and very heavy computation. Some problems, like microlensing modelling, require the processing power of a cluster computer to deliver real-time modelling results. A large cluster computer is expensive and consumes large amounts of resources to run and maintain. For those who do not have the resources to obtain and run a cluster computer, the processing power of a cluster computer is always a dream. However, the modern GPU in the consumer computer that is used for gaming, may provide a solution [44, 45]. The GPU is actually a very powerful processor with the computation power comparable to a cluster computer. The price of a GPU is also only a small fraction of the cost of a cluster computer. There are commercial versions of GPUs that support larger onboard memory and extra features. In this chapter, we discuss the power of the GPU and introduce the architecture and the techniques in programming such a powerful processor.

A cluster computer is usually a shared resource with the exception of those researchers having their own dedicated cluster. Since microlensing modelling can be time critical, as the prediction of light curve behaviour helps to allocate telescope observation resources, it often requires dedicated computing resources.

3.2 Massively parallel processor

A GPU is a massively parallel processor. Unlike a multi-core CPU which is made up by a few full-feature powerful cores, the GPU is made up of a lot of small and simple cores. Each individual core on the GPU is a lot less powerful than the CPU core, but together they have very high data processing throughput by highly utilized data parallelism. While the CPU aims to retrieve data from memory for processing as quickly as possible, the GPU retrieves multiple data from memory and processes them in batches. This creates very high data processing throughput when there are enough data to be retrieved in parallel.

The nature of the GPU is well suited for data-parallel computation applications, but it used to be very hard to access its power. One had to possess extensive and specialized knowledge of the GPU architecture and graphics operations, and it was required to transform the data into pixel arrays and use graphics operations for data processing.

Even before the introduction of modern GPUs, scientists have been trying to use the power of the GPU for years. We know that the GPU is a highly parallel processor capable of computing large amounts of data in parallel. However, it was difficult to program such a chip for scientific computing as all data have to be transformed into grid pixel data and use the image processing operations for parallel computation. The end result also has to be transformed back from the grid pixel data. Moreover, a very low level programming language such as assembly language was usually required to program the GPU. This made the development of a program to work on the GPU very time consuming and required highly specialized skills and knowledge.

Since CUDA (Compute Unified Device Architecture) [46] was introduced in 2006, the scientific computing landscape has changed. CUDA provides a parallel computing platform which allows the user to program the GPU using high-level programming languages, like C and C++ [47]. The scientific community has embraced the GPU as CUDA allows us to program the GPU easily with a high level programming language [48]. This makes programming the GPU even possible by scientists with parallel computer programming skills. The performance of the GPU is also significantly faster than the CPU in suitable problems [49, 50, 51]. Some scientific calculations speed up dramatically by using the GPU. The cost of the GPU is low as it is a consumer device. This make the GPU very attractive for scientific computing. CUDA has been used for all kinds of large-scale computational problems in astrophysics [52, 53, 54, 55], medical [56, 57] and financial computation [58].

The term GPGPU (General Purpose Graphical Processing Units) refers to general purpose GPU computing which uses GPU for computational problems not limited to image-processing problems.

3.3 NVIDIA GPU

The CUDA-enabled NVIDIA GPU is specific designed hardware to provide powerful functionality for general computations on the GPU, it is not just a software implementation. In order to fully utilise and take advantage of the powerful GPU, we need to understand the architecture of the NVIDIA GPU. Some programming models that can provide significant speedup in computations utilise different types of memories for specific tasks [59, 60]. It is important to study the architecture in detail for maximum utilisation [61]. This section discusses the architecture of the NVIDIA GPU in details.

3.3.1 GPU architecture

Figure 3.1 shows the architecture of the Fermi GPU. It consists of 16 Stream Multiprocessors (SM). The GPU is designed to be modular. Each module is made up of smaller units. Figure 3.2 shows the configuration of different modules. Multiple Stream Processor/Core (SP) are grouped together to form an SM, multiple SMs are grouped together to form a Texture/Processor Cluster (TPC) and multiple TPCs form a Streaming Processor Array (SPA). Each SP/core consists of a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). For the GT200 GPU, there are 8 SPs in a SM, 3 SMs in a TPC and there are a total of 10 TPCs on the SPA. Therefore, there are a total of 8SPs×3SMs×10TCPs = 240 cores/SPs on a GT200 GPU while there are 8SPs×2SMs×6TCPs=128 cores on G80 GPU. The architecture of each generation of GPU changed and they usually contain different number of SM and SP. Table 3.1 show the detailed specification of G80, GT200 and Fermi GPU.

Each SP is a single processing core and is a fully pipelined, single-issue, in-order microprocessor complete with two ALUs and a FPU. In G80 and GT200 architecture, each SM is made up of 8 SP and 2 Special Function Units (SFUs). The SFU are used for transcendental operations, like sine, cosine, reciprocal and square root, as well as interpolation in anisotropic texture filtering. There is a instruction cache, a read-only data cache and 16 KB of software managed shared memory in each SM. A TPC consists of SMs, control logic

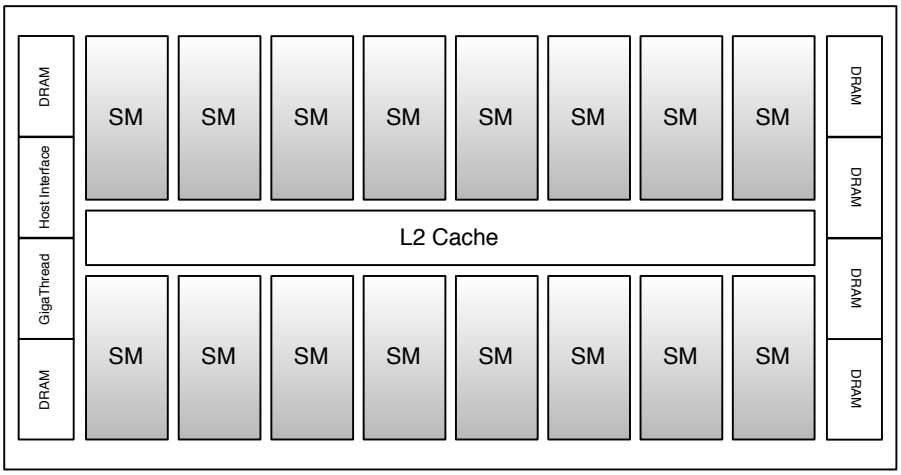

Figure 3.1: Fermi Architecture. This figure show multiple SMs and shared L2 Cache on the Fermi GPU.

and a texture unit. The texture unit includes texture addressing and filtering logic and L1 texture cache.

On the Fermi architecture, there is no TPC, the processing unit is made up of multiple SMs. Each SM has 32 SPs and 4 SFUs. Therefore, there are 32SPs×16SMs=512 cores on the Fermi GPU. Each SM has 64KB of configurable memory on-chip memory. The 64KB configurable memory on Fermi in each SM can be configured as shared memory or L1 cache and can be partitioned as either 16KB shared memory/48KB L1 cache or 48KB shared memory/16KB L1 cache. There are dual warp schedulers and two instruction dispatch units, which allow two warps to be issued and executed concurrently. The term "warp" is explained in the next subsection.

3.3.2 Device Memory Space

CUDA devices use several memory spaces, which have different performance and characteristics. Better performance can be achieved by using the right type of memory in CUDA applications. These memory spaces include global, local, shared, texture, and registers.

Global memory

Global memory is the main and largest memory on the GPU. The on-board device global memory is accessible (read/write) by all running threads. Since the GPU is connected to the

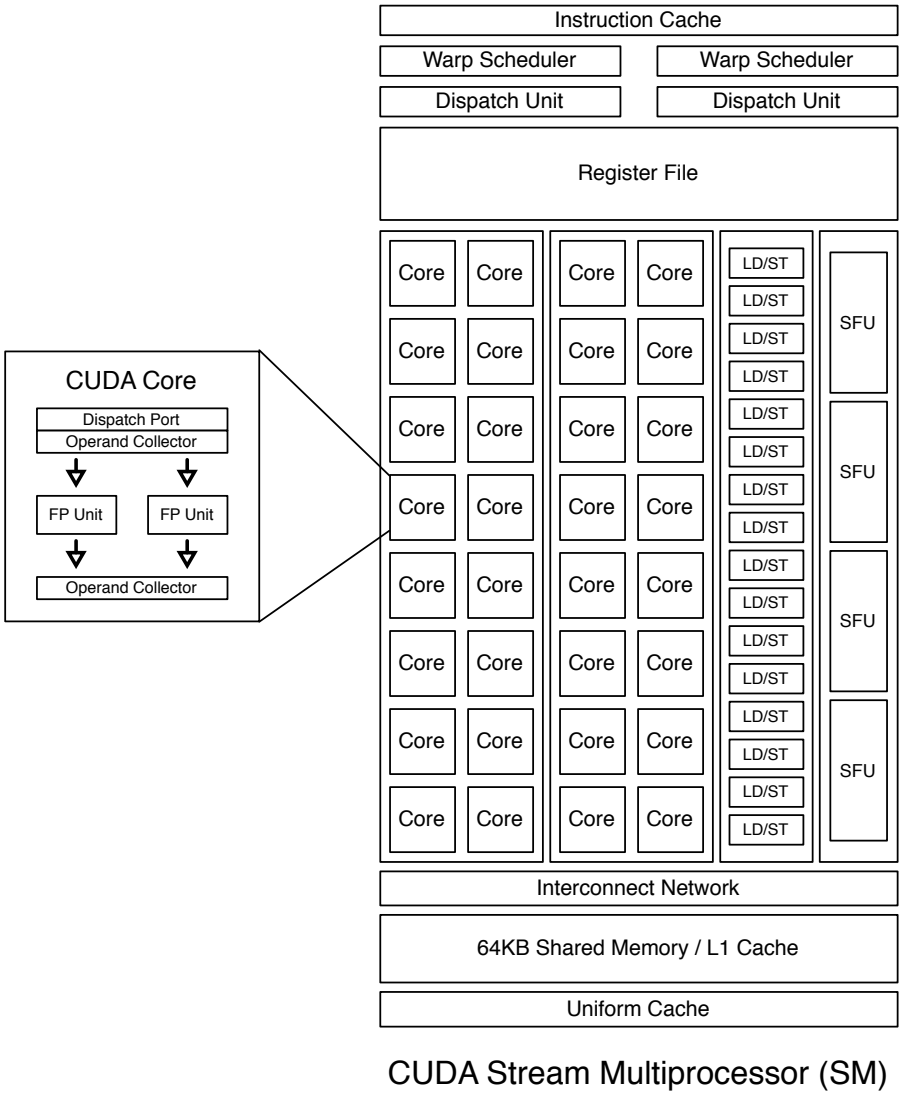

Figure 3.2: Fermi Streaming Multiprocessor.

motherboard through the PCI-E bus, data need to be copied and transferred from the CPU memory to the device global memory before launching the kernel. A kernel is the parallel portion of an application that is executed on the device. When the computation is completed, the result is then transferred back from the device global memory to the CPU memory. Global memory is the slowest type of memory on the GPU. There is a 300-600 clock cycle penalty when accessing the global memory. Coalescing access to global memory is required to achieve good performance in CUDA application. Section 3.3.4 discusses coalescing memory access in detail.

GPU	G80 (8800GT)	GT200 (GTX280)	Fermi (GTX480)
Year of release	2006	2008	2010
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Number of SMs	16	30	16
Number of SPs per SM	8	8	32
ROPs	16	32	48
Single Precision Floating Point Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Double Precision Floating Point Capability	None	30 FMA ops/clock	256 FMA ops/clock
Shared Memory (per SM)	16 KB	16 KB	Configurable 48KB or 16 KB
Texture Address / Filtering	56 / 56	80 / 80	64 / 64
L1 Cache (per SM)	None	None	Configurable 16 KB or 48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent Kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64bit
Compute Capability	1.0	1.2	2.0

Table 3.1: Architecture specifications showing comparisons between the different generations of NVIDIA GPU boards.

Shared memory

The on-chip shared memory can greatly improve the performance of GPU applications as it is a lot faster than global memory. Threads within the same block are able to communicate and cooperate using the shared memory. A block is a group of threads and this concept is discussed in detail in section 3.3.3. It also allows threads to reuse on-chip data and greatly reduces off-chip traffic. For some CUDA applications, shared memory is the key to achieving

high performance. Shared memory can be defined in the kernel by the *__shared__* keyword.

For existing applications that use shared memory as software managed cache, code can be streamlined to take advantage of the hardware caching system, while still having access to at least 16 KB of shared memory for explicit thread cooperation. Best of all, applications that do not use shared memory automatically benefit from the L1 cache, allowing high performance CUDA programs to be built with minimum time and effort.

Texture memory

The texture memory is a read-only memory and resides in device memory but is cached in texture cache contained on each SM. Data fetching from the texture memory costs one memory read from the device memory on a cache miss, otherwise it is read directly from the texture cache. The texture cache is optimized for 2D spatial locality. Performance can be improved when data are reused within the same warp, where the addresses of the data being accessed are close together in 2D. It also has constant latency. Although a cache hit can reduce global memory access, there is no saving in fetch latency to texture cache. The texture memory does not require coalescing access patterns to achieve good performance when the cache is being used frequently.

Texture memory provides improvement in performance when used properly, but it requires carefully designed algorithms to utilize its benefit. Since Fermi GPU introduced cache memory, it has provided a simpler alternative solution for improving performance of memory access. We show a real example in Chapter 4 on the extraction of light curves from magnification maps.

Constant memory

Constant memory is read-only memory and resides on the global memory space. Therefore, it has high latency, however, it is cached in the constant cache on device. It is a lot faster than the global memory as it is cached. The results are fetched from the cache when a cache hit occurs, otherwise it is fetched from the device memory. Constant memory is limited to 64KB for a whole kernel launch. It can be utilize by using the keyword *__constant__* in front of a memory declaration.

Register

This is the fastest form of memory and is only accessible by the thread. It also only has the lifetime of the thread and each running thread in CUDA is assigned with its own registers. The number of registers used in the kernel limit the number of blocks that can be concurrently executed by a SM.

Local memory

Local memory resides at the device memory space, the same as the global memory, which is subject to high latency. The automatic variables (defined by the keyword *__device__*, *__shared__* and *__constant__*) are usually placed in the register, but there are some cases where they may reside in the local memory. For example, when there are large structures or arrays that are too large to fit into the register memory space or if the kernel uses more register memory than physically available on chip.

Warp

A warp is a group of threads that can be executed concurrently at the same clock cycle. In other words, it is the minimum size of the data processed by single instruction, multiple data (SIMD) on a SM. On the G80 and GT200 architecture, a warp represents 16 threads while on the Fermi architecture a warp is 32 threads. Since each block can only be executed on a SM, the number of threads per block should be a multiple of the warp size to achieve maximum utilization.

L1 and L2 cache

Fermi architecture GPU is the first NVIDIA GPU to introduce on-board cache memory to improve memory access performance. Before NVIDIA introduced the cache system on Fermi, we would only use shared memory as a software cache and it needed to be programmed manually. This usually involves complicated programming as there are issues like shared memory bank conflicts that one needs to be aware of when using shared memory. However, Fermi's cache system allows frequently accessed data to be fetched with very low latency. The cache system is handled by hardware automatically, thus, simplifying the programming model.

There are configurable 16KB or 48KB L1 cache for each SM and 768KB of L2 cache shared by all SMs. Both local and global memory also can utilize the cache memory to improve access performance. There are 64KB of shared/cache memory for each SM. Therefore, memory can be configured in 48KB of L1 cache with 16KB of shared memory or 16KB of L1 cache with 48KB of shared memory. We can configure the arrangement of shared/cache memory using *cudaFuncSetCacheConfig()/cuFuncSetCacheConfig()* instructions. The initial configuration is 16KB of L1 cache with 48KB of shared memory. If no instruction is being used to manually configure the shared/cache memory assignment, then whichever memory configuration was used by any recently executed kernel is used until a kernel launch which requires a different configuration of shared/cache memory.

GPU vs. CPU

For an embarrassingly parallel job where the data can be divided into smaller blocks and processed independently, the CPU and GPU take very different approaches to process the job. The CPU tries to process each block of data as fast as possible and access the next block of data in the shortest time possible. The CPU has sophisticated control logic which optimizes the process by overlapping instructions in the instruction processing pipeline so that multiple instructions in different stages can be executed simultaneously in each clock cycle. It also has a large on-chip cache, which allows previously fetched or future use data to be stored in multiple levels of cache to minimize the data access delay from the main memory.

However on the GPU, multiple data blocks are fetched to multiple cores and then processed simultaneously. The GPU accesses multiple data blocks and executes instructions on different data blocks within the same clock cycle. This is different from the CPU approach. The CPU tries to minimize the processing time for each block of data while the GPU is trying to process multiple data blocks at the same time. Although each core on the GPU is not as powerful as on the CPU, the final throughput may be higher as multiple data sets are processed at the same time. However, this requires good thread management and memory access latency hiding.

3.3.3 CUDA threads organization

In multi-thread CPU programming, we may execute multiple threads which are equal to the number of cores on the CPU to utilise the processing power. However, GPU consists of

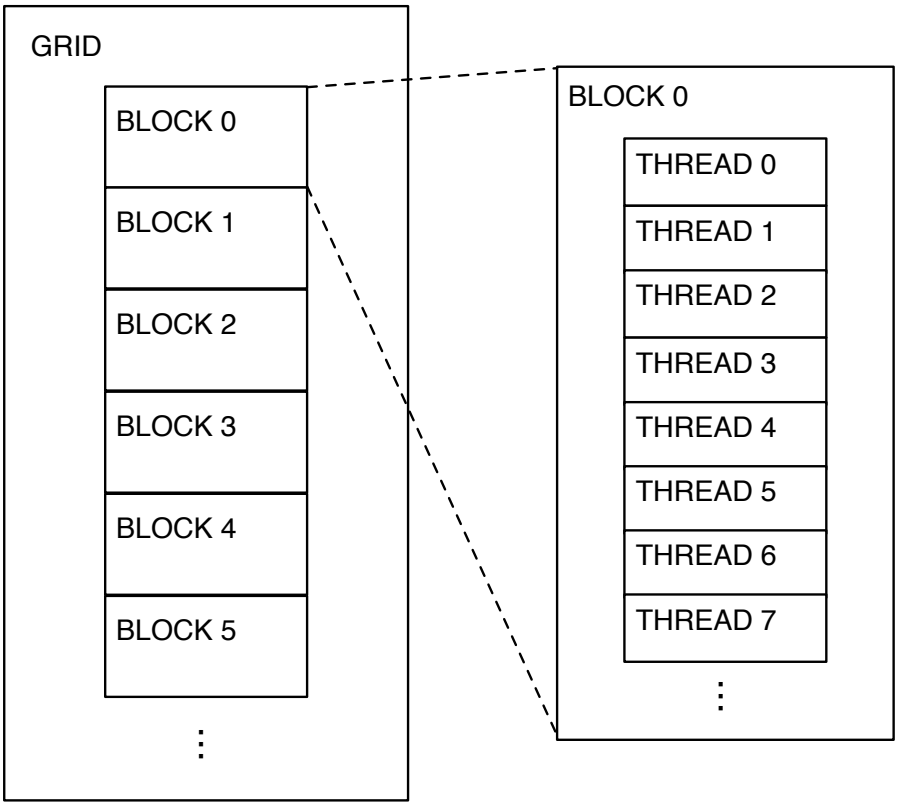

Figure 3.3: Thread organization. Each grid consists of multiple blocks and each block consists of multiple threads.

hundreds of cores and usually requires launching thousands of threads to be fully utilized, compared with a few cores on the CPU running a few threads concurrently. Ideally, all concurrent running threads would be able to communicate with each other quickly in run time, but such a system will be very expensive to build. Therefore, the design of the GPU architecture is a balance between cost and flexibility. Moreover, each generation of GPU architecture will be different. One of the important design goals of CUDA is to allow current code to be run on future GPU hardware. CUDA uses an abstract layer to achieve this goal, providing a logical view of the organization of running threads.

Multiple threads belong to a block, multiple blocks of threads belong to a grid and each GPU can only consist of one grid. Figure 3.3 shows the organization of threads in CUDA. Each block can contain up to 512 threads on the G80 and GT200 series GPU and up to 1024 on the Fermi GPU. Threads in a block can be organized into one-dimensional, two-dimensional and three-dimensional arrays. Blocks of threads in a grid can be organized into either one dimensional or two dimensional structures only.

There are built-in variables for identifing threads and blocks during run time. The variable *blockDim* represents the number of threads in a block, *blockIdx* represents the block index within the grid and *threadIdx* represents the thread index within the block. Those three variables contain integers to represent the index of thread, block and grid. Since threads in a block can be organized into three-dimensional arrays, there are x, y and z integer components in the variable *threadIdx*. The block can be organized into either one- or two-dimensional arrays. There are only x and y components in the variables *blockIdx* and *blockDim*.

For example, consider threads organized into one-dimensional blocks of threads and a one-dimensional grid. Thread id within a block is given by *threadIdx.x* and number of threads in a block is given by *blockDim.x* Therefore, the global thread id can be calculated by

*int globalThreadID = blockIdx.x * blockDim.x + threadIdx.x*

For threads organized into a two-dimensional block of threads and one-dimensional grid, the block and thread index can also be computed by the *blockDim*, *blockIdx* and *threadIdx* variables.

Number of threads in a block is

*int threadsPerBlock = blockDim.x * blockDim.y*

Thread index within a block is

*int threadNumInBlock = threadIdx.x + blockDim.x * threadIdx.y*

Number of blocks in a grid is

*int blockNumInGrid = blockIdx.x + gridDim.x * blockIdx.y*

The global thread id can be calculated by

*int globalThreadID = blockNumInGrid * threadsPerBlock + threadNumInBlock*

3.3.4 Coalescing memory access

The most important factor affecting performance in CUDA applications is coalescing global memory access. Global memory access has a 300-600 cycle penalty so coalescing access is an efficient way to reduce the impact of this latency. When a certain memory access pattern is archived, a half warp (16 words) or a warp (32 words) memory segment can be transferred in one single transaction. For example, for a GPU with warp size 32 reading a memory segment of 128 floats, 32 floats can be read by the GPU in one memory transaction and it only requires 4 memory transactions in total to read 128 floats. It is a 32 times higher

GPU	G80	GT200	Fermi
Maximum number of threads per block	512	512	1024
Maximum number of resident warps per multiprocessor	24	32	48
Maximum number of resident threads per multiprocessor	768	1024	1536
Number of 32-bit registers per multiprocessor	8K	16K	32K
Maximum width for 1D texture reference bound to a CUDA array	8192	8192	65536
Maximum width for 1D texture reference bound to linear memory	2^{27}	2^{27}	2^{27}
Maximum width and height for 2D texture reference bound to a CUDA array	65536 x 32768	65536 x 32768	65536 x 65535
Maximum width and height for 2D texture reference bound to a linear memory	65000 x 65000	65000 x 65000	65000 x 65000

Table 3.2: GPU architecture. Information on threads and warps in different generations of GPU.

bandwidth than reading 128 floats individually. However, a specific memory access pattern is required in order to achieve the maximum memory bandwidth. On different generations of GPU, the rules and restrictions are different. The newer the GPU, the more flexible in the global memory access.

Global memory should be aligned in segments of either 64 bytes (16 words) or 128 bytes (32 words) for coalescing memory access. Figure 3.4 shows three memory access patterns

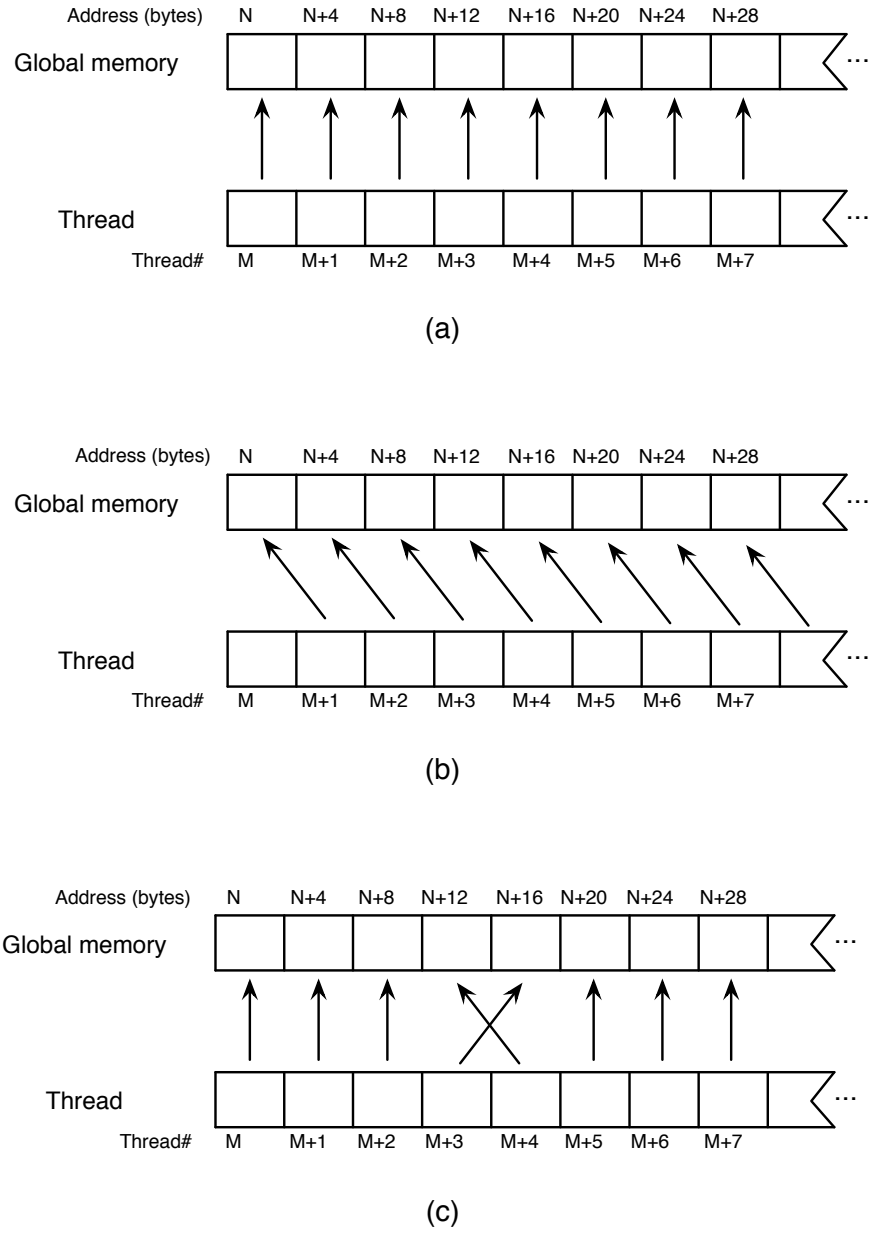

Figure 3.4: GPU Memory Access. Figure (a) shows coalescing global memory access which only takes one memory transaction. Figure (b) and (c) shows non-coalescing global memory access which takes 16 individual memory transactions on devices of compute capability 1.0 and 1.1. However, only one memory transactions is required with compute capability 1.2 or higher, when memory access is within 32/64/128 bytes segment.

which result in different numbers of memory transactions. When each thread within a block sequentially reads one float on an aligned memory segment as shown in Figure 3.4(a), it

only takes one memory transaction to read 16 floats for devices of compute capability 1.x or 32 floats for device of compute capability 2.x. However, Figures 3.4(b) (misaligned starting address) and (c) (non-sequential access) show non-coalescing memory access. This results in 16 individual memory transactions on devices of compute capability 1.0 and 1.1. On devices of compute capability 1.2 or higher, it only takes one memory transaction when memory access is within 32/64/128 bytes segment.

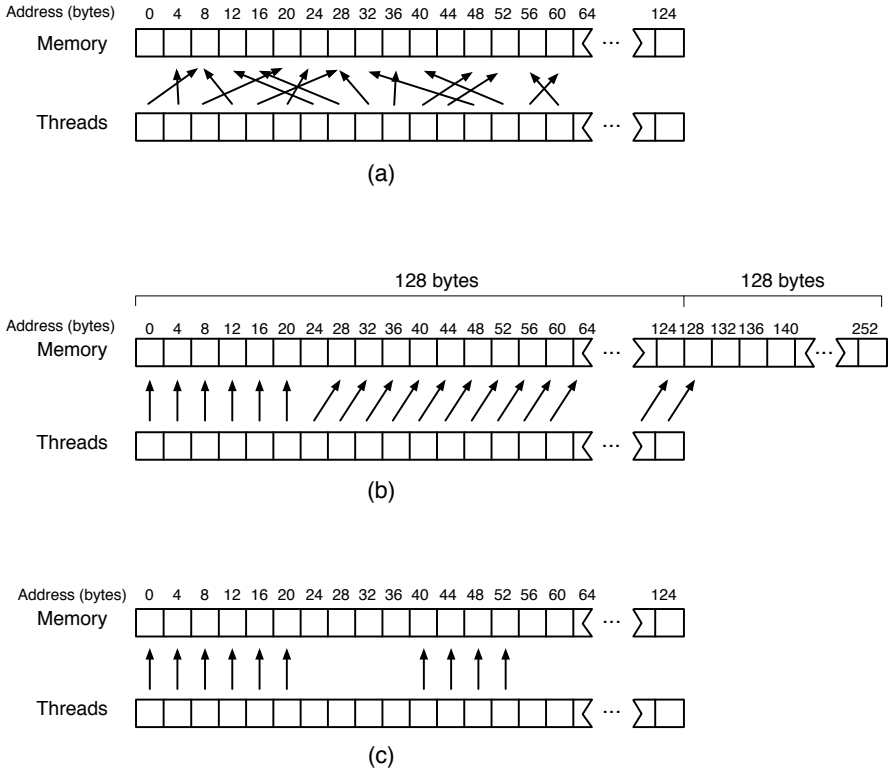

Figure 3.5: GPU Memory Access. Three memory access patterns that are non-coalescing on devices of compute capability 1.0 and 1.1, but are coalescing memory access on devices of compute capability of 1.2 or higher.

The restriction of sequential aligned global memory access may pose a challenge in programming the GPU. However, coalescing global memory access is more flexible on newer hardware. On devices of compute capability of 1.2 or higher, coalescing access can be achieved for any memory access pattern of 32 bytes of 8-bit words, 64 bytes of 16-bit words and 128 bytes of 32-bit and 64-bit words. Figure 3.5 shows three access patterns that are non-coalescing on devices of compute capability 1.0 and 1.1, but are coalescing access on

devices of compute capability of 1.2 or higher. Figure 3.5(a) shows random memory access of a 32-bit float within a 128 bytes memory segment, resulting in one memory transaction on devices of compute capability of 1.2 or higher. When reading floats (32-bit) from a misaligned memory segment that is within a 128 bytes segment, it only takes a single memory transaction on devices of compute capability of 1.2 or higher. However, when the access memory is outside the 128 bytes segment, two memory transactions are issued, as shown in Figure 3.5(b).

When one or more threads within a warp are not participating in memory access (shown in Figure 3.5(c)), 16 serialized transactions are issued on devices of compute capability of 1.1 or lower, but only one transaction on devices of compute capability of 1.2 or higher. However, even though there is only one memory transaction on newer hardware, some memory bandwidth is wasted as all data in the segment are fetched including the memory addresses that are not required. This issue bring an awareness on how memory in CUDA should be organized and accessed.

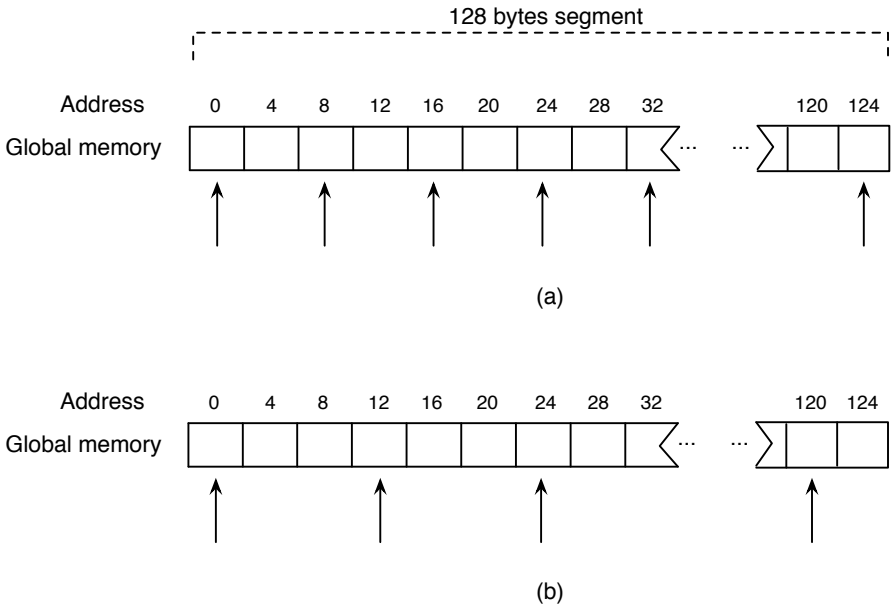

Figure 3.6: GPU Memory Access. Some memory bandwidth is wasted due to not all memory locations being fetched within the 128 bytes memory segment.

In the case that not all required memory locations are being fetched within the 128 bytes memory segment, some memory bandwidth is wasted. Figure 3.6(a) shows only even memory locations are being accessed, therefore half of the memory bandwidth is wasted. Figure 3.6(b) shows only 1/3 of memory locations are being accessed within the 128 bytes

memory segment, thus, 2/3 of memory bandwidth is wasted. This is due to the whole segment of memory being fetched in both cases even though the GPU kernel code does not require the unwanted memory addresses.

The following example shows a similar problem with a C struct having three variables. The struct is allocated a segment of memory on the global memory space using the CUDA memory allocation instruction. In the kernel, a global memory access to the variable x in the struct space is issued. All the variables in the struct are fetched even though the kernel only requires one variable. Therefore, 2/3 of the memory bandwidth is wasted.

```
struct space {
  float x;
  float y;
  float z;
};

space *d_space;
cudaMalloc((void**) &d_space, ... );

// GPU Kernel
__global__ void run_kernel(space *d_space) {
  int gtid = blockIdx.x * blockDim.x + threadIdx.x;
  float x = d_space[gtid].x;
  ...
}
```

Instead of organizing the *x,y,z* arrays in a struct, they can be organized in three arrays and allocated memory space for each array. Alternatively, they can be organized as a struct as in the following example. The memory bandwidth can be maximized in this case as only the *d_space.x* variable is fetched.

```
struct space {
  float *x;
  float *y;
  float *z;
};

space d_space;
```

```
cudaMalloc((void**) &d_space.x, ...);
cudaMalloc((void**) &d_space.y, ...);
cudaMalloc((void**) &d_space.z, ...);

// GPU Kernel
__global__ void run_kernel(space *d_space) {
  int gtid = blockIdx.x * blockDim.x + threadIdx.x;
  float x = d_space.x[gtid];
  ...
}
```

3.3.5 Shared memory bank conflicts

Shared memory is fast on-chip memory which allows shared data between threads within the same block. It is often faster to use shared memory instead of global memory for communication between threads in a block. The latency of accessing the global memory is 100× higher than the shared memory and is almost as fast as the register. The shared memory is often used as a software cache or used as a temporary space for rearranging data to allow coalescing access/write to global memory. Although shared memory is a lot faster than global memory, there are access pattern rules that must be followed in order to achieve the maximum bandwidth.

Shared memory is divided into smaller memory modules called "banks" which can be accessed simultaneously. Each bank is 32 bits in length and organized to carry successive 32-bit words. Each bank can be accessed in a single clock cycle. Therefore, the bandwidth of shared memory is 32 bits per bank per clock cycle. When n threads are successively accessing n banks in shared memory, as shown in Figure 3.7(a), each bank can be accessed simultaneously. Similar to the global memory access pattern, we only consider the access pattern for threads in the same warp. For devices with compute capability 1.x, there are 16 banks for each warp (32 threads). The access is split into two, with the first half of threads accessing the shared memory in one clock cycle and the second half of threads accessing the shared memory in the second clock cycle. For devices with compute capability 2.x, there are 32 banks for each warp, all threads in a warp can access the shared memory in a single clock cycle, assuming there is no shared memory bank conflict. Therefore, the maximum bandwidth of shared memory can be 16 or 32 times the bandwidth of a single bank when there are no bank conflicts.

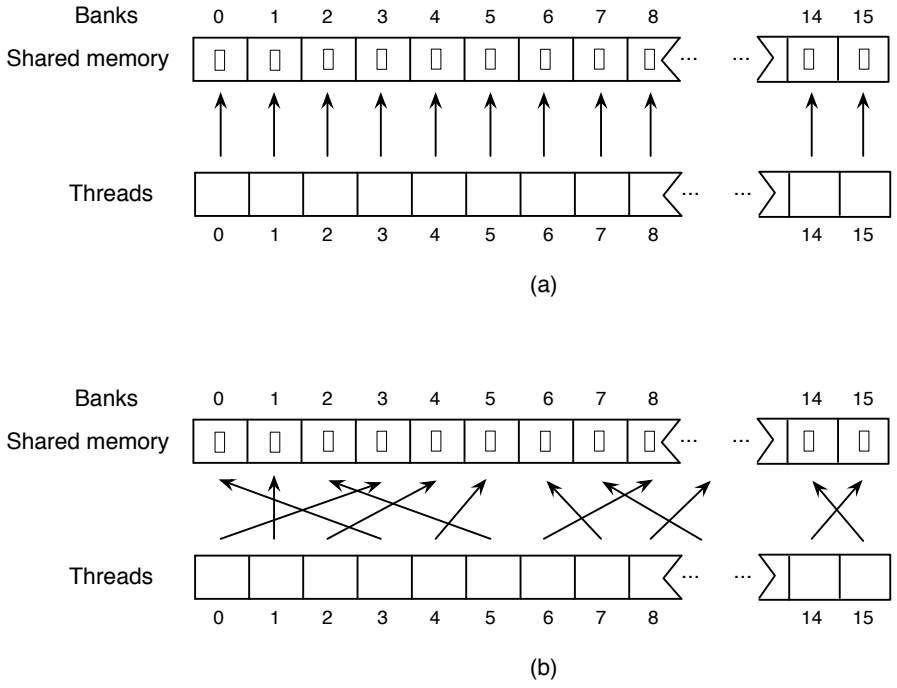

Figure 3.7: GPU Shared Memory Access. Both (a) and (b) are a conflict-free shared memory access, even though there is random access in (b). There is conflict-free access as long as there are no threads accessing *different bytes* of a shared memory bank.

Figure 3.7 shows two scenarios that have no shared memory bank conflict. When each thread within the same warp accesses their own banks, even when they access the shared memory banks randomly, there are no shared memory bank conflicts as long as there are no threads accessing *different bytes of the 32-bit words at the same bank*. Figure 3.8 shows shared memory bank conflicts by multiple threads accessing different bytes at the same banks. When multiple threads access *different* bytes in the same memory bank, the accesses are serialized. The hardware splits the access into as many separate conflict-free accesses as required. Therefore, the effective bandwidth is one half in 2-ways bank conflicts and one fourth in 4-ways bank conflicts.

It is important to realize that when several threads are accessing the *same byte* at the same shared memory bank as in Figure 3.9(a), it is a conflict-free access. There is a conflict access only when multiple threads access *different bytes* at the same shared memory bank. One more exception is when all threads in the warp access the *same byte* at the same shared memory bank as in Figure 3.9(b). This is a broadcast access and is conflict-free access. It

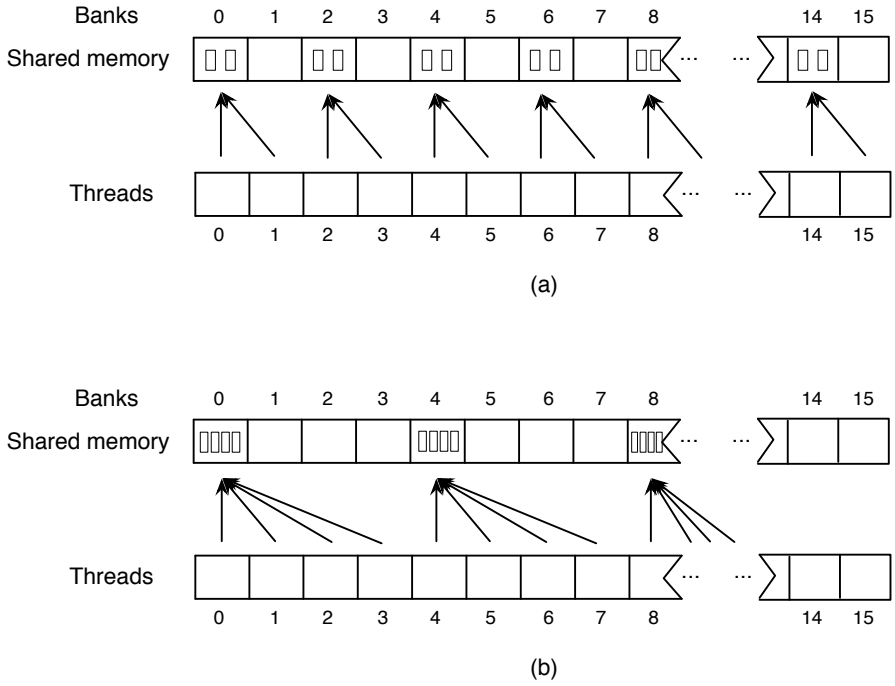

Figure 3.8: GPU Shared Memory Access. Two threads in (a) are accessing *different bytes* of a shared memory bank, resulting in 2-ways bank conflicts. It cut the effective bandwidth in half as the hardware splits the access in two conflict-free accesses. In (b), four threads are accessing *different bytes* of a shared memory bank and there are 4-ways bank conflicts. The effective bandwidth is one fourth of the maximum bandwidth.

only takes one clock cycle for all threads to read the value from that single shared memory segment.

For 64-bits access, there are 2-ways bank conflicts in devices with capability 1.x when each bank is a 64-bits double. However, there are no shared memory bank conflicts in 64-bits access on devices with capability 2.x.

3.4 Floating point accuracy

The double precision floating-point arithmetic is not supported in the G80 architecture. However it is supported on the newer generation of GPU. It is 8 times slower on GT200 and 2 times slower in Fermi compared with single precision floating-point arithmetic, as shown in Table 3.1. Therefore, single precision floating-point arithmetic is used whenever the problem can be solved with acceptable accuracy.

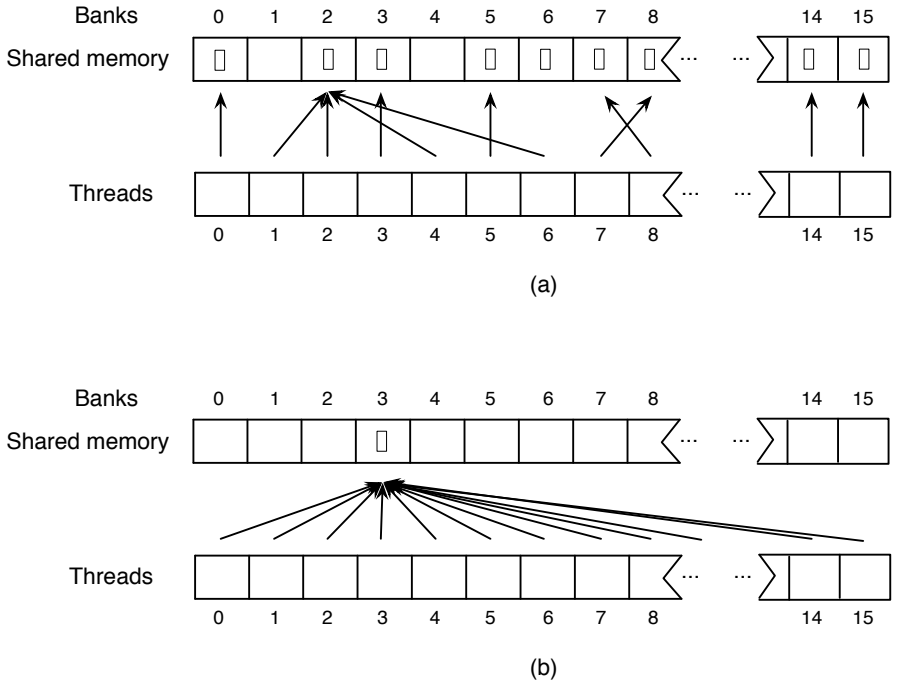

Figure 3.9: GPU Shared Memory Access. Both (a) and (b) are conflict-free access. In (a), since threads 1,2,4 and 6 are accessing the *same byte* of shared memory bank 2, it is a conflict-free access. It only takes one clock cycle for all threads to access the shared memory. All threads in a warp at (b) are accessing the *same byte* of a single shared memory bank, it results in a broadcast access. This is also a conflict-free access and only takes one clock cycle to complete.

CUDA follows the IEEE-754-2008 standard for binary floating-point arithmetic with some exceptions. The computation result may be slightly different when comparing the result between the CPU and the GPU that perform the same computation. This may be due to different round-off methods or computing in different precision. Since single precision floating-point arithmetic is faster on the GPU, most applications are running in single precision when single precision is acceptable while applications on the CPU are mostly running in double precision. Here are some important deviations from the standard that may affect the computed results.

- For single-precision floating-point numbers on devices with compute capability 1.x:
  - Addition and multiplication are combined into a single float multiply-add instruction (FMAD) that truncates the intermediate result of the multiplication.
  - Division is implemented with reciprocal in a non-standard-compliant way.

  - Square root is implemented in reciprocal square root in a non-standard-compliant way.
  - Underflow results are flushed to zero.
  - Round-to-nearest-even and round-toward-zero are the only rounding modes supported for addition and multiplication. Directed rounding toward +/− infinity is not supported.

- For double-precision floating-point numbers on devices with compute capability 1.x:
  - The only supported IEEE rounding mode for reciprocal, division and square root is round-to-nearest-even.

- For conversion of a floating-point value to a integer. If the floating-point value falls outside the range of the integer, the end of the unsupported range is clamped. This is different from the x86 architecture behaviour.

3.5 Performance tuning

For most problems, the performance bottleneck is memory bounded. The main strength of the GPU is the ability to process an array of data in parallel, but it is also a weakness of the GPU as global memory access latency is high. Therefore, coalescing memory access is one of the most important aspects when programming the GPU to improve memory access performance. However for some problems, the performance bottleneck may be instruction bounded. In order to utilize the maximum power of the GPU, we need to beware of several details related to how the GPU handles different instructions.

3.5.1 Control flow

When there is a control flow statement (if, switch, do, for, while) in the kernel, all threads in a warp can execute simultaneously if there is no divergent branch. However, if there is a divergent branch, for example even threads in one direction and odd threads in the other direction, the different execution paths will be serialized. The kernel will first execute the even threads and put the odd threads to sleep and then execute the odd threads and put the even threads to sleep after the even threads have completed the branching. The hardware keeps switching between the two groups of threads until the kernel is completed.

To improve the performance of divergent branching, we can re-organize the threads to avoid diverging within a warp. Since there is performance impact only when threads within the same warp diverge, we can reduce the impact of a diverging branch by making sure the branch granularity is a whole multiple of warp size. The following code fragment will result in branch granularity < warp size.

```
if (threadIdx.x > 8) {...}
else {...}
```

Since the warp size is 32, the first 8 threads are in different paths to the other threads within the warp, so the execution paths are being serialized, thus lowering performance. Below is an example which results in a branch granularity is multiple of warp size.

```
if (threadIdx.x / WARP_SIZE > 2) {...}
else {...}
```

There is no diverging within a warp in this example, all threads in a warp are executed simultaneously. There is no performance impact as long as the diverging has not occurred within a warp. Threads in different warps can have a different execution paths with no performance impact.

Signed vs. unsigned loop counters

A loop counter used in a *for loop* is often declared as an unsigned integer since the counter usually does not have a negative value. However, using a signed integer as a loop counter provides slightly better performance. The signed integer overflow causes undefined results in C language standard. Compilers may often offer better optimization with signed arithmetic than unsigned arithmetic, since the overflow semantics in unsigned variables is well defined and therefore prevents the compiler from using some optimizations like strength reduction.

Divergent iteration

Since CUDA uses highly parallel execution of threads to provide high performance computing, any situations that prevent the parallel execution of threads will have significant

impact on performance, for example, the divergent branching described above. Here is another situation that will cause significant performance impact.

```
__global__ void sum(int *index, float *data, float *result) {
  int gtid = blockIdx.x * blockDim.x + threadIdx.x
  int sum = 0
  for (int i=0; i<index[gtid]; i++) {
    sum += data[i];
  }
  result[i] = sum;
}
```

The above code shows the number of loop iterations for each thread is dependent on the input data. Therefore, loop iterations of each thread in the same warp are different and cause divergent iteration. For example, when the number of loop iterations for most threads in a warp are 10 and there is one thread with 1000 iterations, all threads in the warp are waiting for that single thread to complete the iteration before they can execute the last statement *result[i]=sum;* in the above example. This can post significant performance impact in kernel execution since the hardware cannot execute another warp until the whole warp is completed.

The performance of this situation can be improved by limiting the number of iterations of each thread. The data is rearranged after the first kernel execution and the unfinished jobs continue to be executed in the next kernel launch.

3.5.2 Thread execution

In CUDA, thread management is performed by hardware. Once the kernel is launched, the hardware is in control of the thread execution. It is important to know that the order of thread execution is unspecified as well as the block execution order. It is the developer's job to design an algorithm that works with the above condition since we have no control of thread execution order. The only control a developer has on thread execution is the ability to synchronize threads within the same block. In order to maximize the utilization of the GPU, it is important to understand how CUDA manages thread execution. Here, we look into details on how CUDA organizes threads for execution in this section.

Multiple threads in CUDA are organized into a block and multiple blocks of threads are organized into a grid as discussed in Section 3.3.3. Each block of threads is again organized into a warp (32 threads) during the kernel execution. A block of threads can only execute on the same SM and a warp of threads are executed in parallel on the SM. Each thread is processed by a SP and the number of SPs in a SM is hardware specific. The G80 has 8 SPs while Fermi has 32 SPs in a SM. Multiple blocks are assigned to a SM when the total number of blocks are greater than the number of SMs.

Although the thread execution details are handled by the hardware and the user has no way to control the execution order of threads, we can improve performance of the program by understanding how the GPU handles threads in general. A warp of threads (for example, 32 threads on Fermi) is processed by a SM in parallel. They all access memory and running instructions simultaneously. Therefore, the basic parallel unit is 32 threads on Fermi. There are 16 SMs in total in the Fermi GPU, therefore the total basic parallel unit is $32 \times 16 = 512$ threads. If we launch 512 threads in a kernel and each thread has a loop for processing instructions as well as memory access until the job is done, the GPU will be highly under unitized as there will be very long delays on accessing memory. This will be discussed in detail in the next section.

In order to overcome the high memory access latency, we need to launch more threads than the available GPU cores. When a warp is waiting for long latency operations like global memory access, the GPU selects another warp of threads for execution while waiting for the operations of the first warp. This mechanism avoids much idle time in the GPU when processing long latency operations and the selection of warp does not introduce any delay and is referred to as zero-overhead thread scheduling. This is how the GPU overcomes the delay from the long latency operations and processes large amounts of data efficiently.

Here is an example on the Fermi GPU. There is an array of data consisting of 16384 elements stored in the global memory. Each data element is loaded from the global memory, an arithmetic instruction is performed and the result is written back to the global memory. One may launch 512 threads with block size 32 since there are 16 SPs in Fermi GPU (32 threads for each SP). Each thread iterates 32 times to perform the memory and arithmetic operations. Therefore, threads are being processed in parallel since there are 512 cores in the Fermi GPU. However, the GPU will be highly under utilized as each thread is suffering from the long delay from the global memory read/write operations in each iteration.

Instead of launching 512 threads and letting each thread iterate 32 times, we may launch more blocks of threads and reduce the number of iterations in each thread. There are extra

warps of threads ready to swap in for processing in each SM. For example, we can launch 128 blocks with 32 threads in each block and each thread only iterating 4 times. Therefore, there are 128 x 32 = 4096 threads launched and each SM is assigned 8 blocks (128 blocks / 16 SPs). Since each block has 32 threads, there are an extra 7 warps which are ready to be selected while waiting for the long latency memory operations.

The optimum block size and number of blocks are dependent on the specific hardware as each GPU architecture has a different number of cores and SMs. Although the current code can be run on future GPUs, it may not be optimized to utilize the maximum performance of the new GPU as the optimum block size and number of blocks are different. NVIDIA provides an occupancy calculator with the SDK and one can compute the optimum block size and number of blocks for specific hardware.

3.5.3 Memory latency hiding

When developing a multi-thread program for a multi-core processor, the job is usually divided into multiple smaller jobs equal to the number of processor cores and executed by a few threads in order to fully utilize the CPU cycle. One may think that the same principle can be applied to the GPU. If we divide the job into the number of GPU cores and run a number of threads equal to the GPU cores, it will highly underutilize the GPU. The CPU has a superior cache mechanism, but the GPU only has a manual or simple cache mechanism in comparsion. Data are usually stored within the global memory on the GPU and it has large latency, around 300 - 400 GPU cycle. So global memory latency hiding is essential when programming the GPU.

There are two major techniques to reduce the impact on the global memory latency. The first is to launch large numbers of blocks that are significantly larger than the number of SM, which provide more warp of threads ready to be executed. When a warp of threads is waiting for data to read/write from the global memory, the GPU hardware holds on to the current executing warp and starts to execute instructions in the other warp until the data successfully reads/writes on the first warp. This way the GPU does not waste any time in waiting for the data from the global memory on a single warp. Therefore, launching a large number of blocks allows the GPU to hide the global memory access latency.

The second technique is to use shared memory as a software cache. Beside using a large number of blocks, coalescing memory access also has a crucial role in memory latency hiding.

The performance decreases dramatically when memory access is not done in a coalescing fashion. We can use the shared memory, which can be accessed by all threads within the same block, to rearrange the data and allow coalescing memory access to the global memory.

3.5.4 Arithmetic Instructions

For maximizing the performance of a CUDA program, it is essential to know the throughput of different arithmetic instructions on the GPU. The double precision arithmetic instructions on a device with compute capability 1.x is 8 times slower than single precision arithmetic instructions while it is 2 times slower on devices with compute capability 2.0. The double precision trigonometric functions are particularly costly especially when the argument is large in magnitude. One should use single precision arithmetic instructions for speed when it does not affect the output. Intrinsic functions, that are built-in to CUDA, should be used to replace standard instruction, although it only supported in device code. Integer division and modulo operation are also very expensive and should be avoided. Instead bitwise operations should be used whenever possible.

3.6 Conclusion

The modern GPU is a powerful massively parallel processor with a performance comparable to a cluster computer with only a fraction of the cost. Unlike the old generation GPUs, the modern GPU can be programmed by high level programming languages and are suitable for a wide range of problems. CUDA from NVIDIA provides a platform to program the GPU easily and with the flexibility to solve highly parallel scientific problems such as microlensing modelling.

Although CUDA allows us to program the GPU easily, one has to understand the architecture of the GPU and apply programming principles in order to utilize the maximum performance. This includes the abstract layers in CUDA which enable current code to run on future hardware, the organization of blocks and threads, the optimum number of blocks and threads for each GPU architecture, the memory latency hiding techniques which by including coalescing memory access and keeping the GPU cores busy, utilize different types of available memories on the GPU to maximize performance, choosing the right arithmetic instructions for the problem and minimizing divergent branching in control flow statements.

It is not too hard to increase the performance of a parallel problem a few times faster on the GPU, but it may be difficult and require some effort to increase the performance to ten or hundred times faster. It is essential to understand and utilize the programming principles discussed in this chapter to maximize the performance of the GPU. In the next two chapters, we utilize some of the techniques discussed in this chapter to develop a GPU program to perform microlensing modelling.

Chapter 4

Microlensing Modelling using Magnification Maps

4.1 Introduction

The GPU is a massively parallel processor capable of performing calculations on multiple data simultaneously in a single clock cycle, as described in Chapter 3. It has very high memory bandwidth and arithmetic throughput that is orders of magnitude higher than the CPU. Most of the microlensing modelling calculations are embarrassingly parallel, and are well suited for the parallel nature of the GPU, especially the numerical integration for the finite-source effect that is very demanding in microlensing modelling computation.

Although the performance of the GPU is a lot higher than the CPU, programming the GPU is not without limitations and restrictions. In this chapter we discuss how to use GPUs to perform different aspects of microlensing modelling calculations, the performance and limitations of the GPU, how to achieve the best performance from the GPU by parallel programming techniques, and compare the cost and performance ratio between the CPU and the GPU in these applications.

4.1.1 Magnification map

As we discussed in Chapter 2, the finite-source calculation is the most computationally demanding part of microlensing modeling. One solution to this problem is using the technique

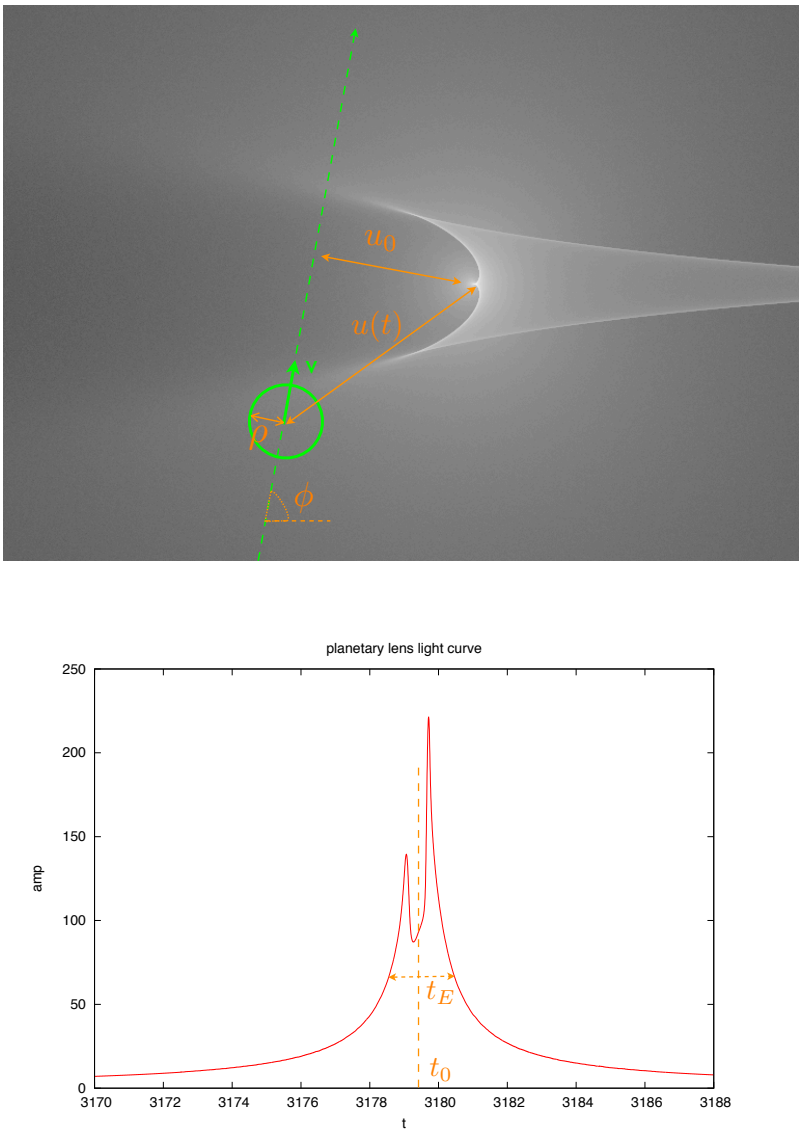

Figure 4.1: The top figure shows a magnification map generated with a binary lens system for a given ε and d. A typical planetary caustic is produced by having a companion with planetary mass fraction. The green dotted line represents the trajectory of the source star and the green solid circle illustrates the source star area. The bottom figure shows a theoretical light curve extracted from the magnification map with the given source star track. The amplification at a given time is calculated by integrating the source star area.

of magnification maps [42]. A magnification map is a 2-dimensional array, each element representing a solution to the lens equation at the line-of-sight to the lens system corre-

sponding to that element. The solutions are obtained by inverse ray shooting. Since the finite-source calculation can be computed by integrating the source star area numerically using inverse ray shooting, multiple finite source calculations are able to computed on a magnification map. In inverse ray shooting, rays are shot in the reverse direction, from the lens plane back to the source, and rays that land within the source area are collected. Each ray is weighted by the limb-darking profile and the amplification can be obtained by the summation of weighted rays.

Inverse ray shooting demands lots of computation power as lots of rays must be shot to achieve an accurate result. However, the shot rays can be reused when we divide the parameter space into two sets, $\{\varepsilon, d\}$ and $\{ t_0, t_E, u_0, \phi, \rho\}$. Here ε is the mass fraction, d is the projected separation, t_0 is the time of closest approach, t_E is the Einstein crossing time, u_0 is the distance of closest impact, ϕ is the angle of source star track and ρ is the source star radius. The inverse ray shooting equation (Eq. 2.11) only depends on the physical parameters $\{\varepsilon, d\}$, so we can shoot a large number of rays, accumulate them onto the whole source plane, and then store them to create a magnification map as shown in Figure 4.1. The amplification at a given time can be computed by integrating the source area on the magnification map. We can extract a theoretical light curve by moving the source across the magnification map as shown in Figure 4.1.

The major advantage of using magnification maps is to allow us to reuse the shot rays, as the parameters ε and d can be represented by a single magnification map. Once a magnification map of a given $\{\varepsilon, d\}$ is created, we can extract different light curves from the magnification map that represent the other set of parameters $\{t_0, t_E, u_0, \phi, \rho\}$. The trajectory of the track across the map is defined by the parameters t_0, t_E, u_0 and ϕ. A theoretical light curve can be read from the map together with the parameter ρ. Therefore, searching the best fitting light curve for a given $\{\varepsilon, d\}$ is equivalent of searching the best track across the magnification map. This technique is very powerful and is especially good for grid searches in parameter space ε, d, t_0, t_E, u_0, ϕ, ρ because the finite-source computation can be done relatively quickly by integrating the source area and the shot rays reused to compute a large number of light curves for a given $\{\varepsilon, d\}$.

The disadvantage of using magnification maps is that it is very time consuming to create a large number of magnification maps. We need to compute the inverse ray-shooting equation millions or billions of times for each magnification map. Furthermore, each ray requires two random numbers to randomize the shooting position on the lens plane to avoid artifacts forming on the magnification map. The random number generation alone can take up to

30% to 40% of the total computation time and a good random number generator is needed to produce good quality magnification maps. The number of rays required depends on the required signal to noise ratio. Since the randomised shooting position introduces noise to the magnification map, one way to overcome this numerical instability is to shoot more rays.

There is no set rule-of-thumb in deciding the actual number of rays that one would shoot. The number of required rays depends upon the required signal to noise ratio. Shooting more rays reduces the noise, but takes up more computation time. One tries to achieve a balance between the computation time and the accuracy required and this is often done on a case-by-case basis. Typically millions and sometime billions of rays are required.

On the magnification map, the pixels need to be small enough to resolve the source star extent on the map. The actual size depends upon the extent of the map on the source plane and the sizes of the source star that we may expect to be dealing with. Typical source star size is $\sim 0.001 - 0.005$ Einstein radii for main sequence stars, but there can be as large as $\sim 0.01 - 0.02$ Einstein radii for the rarer microlensing events involving red giant source stars.

Each magnification map also requires storage space ranging from tens of MB to hundreds of MB depending on the resolution of the magnification map. Although we can store the magnification map and reuse them for future modeling, they can use up considerable amounts of storage space and each modelling task may require different resolutions and sizes of the maps. The time it takes to generate the required specification of magnification map for a particular event further increases the modelling time.

Despite the generation of magnification maps being time consuming and requiring large storage space, it is still a very attractive approach in microlensing modelling as it allows us to search through the parameter space $\{ t_0, t_E, u_0, \phi, \rho\}$ for a given $\{\varepsilon, d\}$ quickly. Also the computation can be done in parallel with each processing unit generating magnification maps for a given range in $\{\varepsilon, d\}$ and searching for the best fitting light curve represented by parameters $\{ t_0, t_E, u_0, \phi, \rho\}$. A typical grid search using this technique requires hundreds or even thousands of CPU hours but is very well suited to perform on a large cluster computer with hundreds of nodes as it is an embarrassingly parallel problem. The following section compares the cost between GPU and cluster computer.

4.1.2 From cluster computer to GPU

Although the computation discussed in the previous section is very well suited to perform on a cluster computer, the cost of owning and running such a system is expensive and it

is usually shared between departments. The submitted jobs must wait in a queue until resources are available. This puts constraints on how quickly a microlensing event can be modelled. The MOA and OGLE collaboration discover hundreds or even thousands of microlensing events every year. Since telescope resources are expensive and limited, it is important that we are able to model and predict an anomaly event in real time so that the followup network can focus their resources on important events. This is not possible unless one has a large dedicated cluster computer.

This problem can be solved by using GPUs to help the computation. It is a very cost-effective solution as the computation speed up in microlensing modelling by the GPU is a factor of hundreds but with only $< 2\%$ of the cost of an equally powerful cluster computer. The running cost and space utilized by a computer with GPUs is significantly lower than a large cluster computer.

4.2 Magnification map generation

As discussed in Chapter 2, a microlensing light curve of a point source can be computed by solving the lens equation to obtain amplification as a function of time. The disadvantage of such an approach is that amplification rises to be infinite and is discontinuous when a source crosses the caustic curve. More importantly, point-source approximation is insufficient for modelling planetary events [62]. Evaluation of the finite-source effect is essential, especially for caustic crossing events.

Finite source computation can be done by integrating the disk of the source star with point-source amplification, but this approach presents numerical difficulties as the magnification can rise to be infinite on the caustic curve. We can avoid the singularities by shooting rays (photons) from the lens plane backward to the source plane using the lens equation (Eq. 2.11). Then any rays that are lensed within the source are collected and weighted by the limb-darking profile. The amplification light curve is calculated by determining the amplification at every source position. This technique is referred to as inverse ray shooting.

Although inverse ray shooting can be used to compute the finite-source effect and avoids the problem of discontinuity in amplification, it is relatively computationally expensive. In fact, one can store shot rays and re-use them to evaluate multiple light curves for a given lens configuration. A magnification map can be created by saturating the source plane with rays by shooting rays from the lens plane evenly.

Another benefit of using magnification maps, is that the finite-source calculation can be solved directly by integrating the area of a source position on the magnification map. As discussed in Chapter 2, this approach can avoid the singularities produced by solving the lens equation directly. The calculation of a light curve is very fast even including computation of the finite-source effect, and multiple light curves can be computed from the same magnification map. The challenge is that the generation of a magnification map is computationally expensive. The benefits of the magnification map technique is outweighed by the time required to generate one.

The type of computation for generating the magnification map, which is highly data-parallel and replicative, is very well suited to perform on the GPU. In this section, we first discuss the procedure of generating magnification maps on the CPU, followed by how we can use GPUs to speed up the computation.

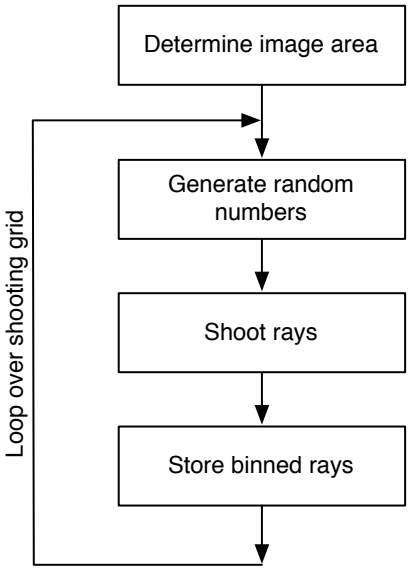

Figure 4.2: General procedure of magnification map generation.

4.2.1 Magnification map generation using CPU

Figure 4.2 shows the general procedure for generating a magnification map. First, we determine the area (image area) on the lens plane for which rays shot from within are lensed on the source plane. Second, random numbers are generated for the ray shooting step. Third,

rays are shot from a random position within the image area on the lens plane backward to the source plane by the lens equation, and all the rays lensed on the source plane are binned and stored.

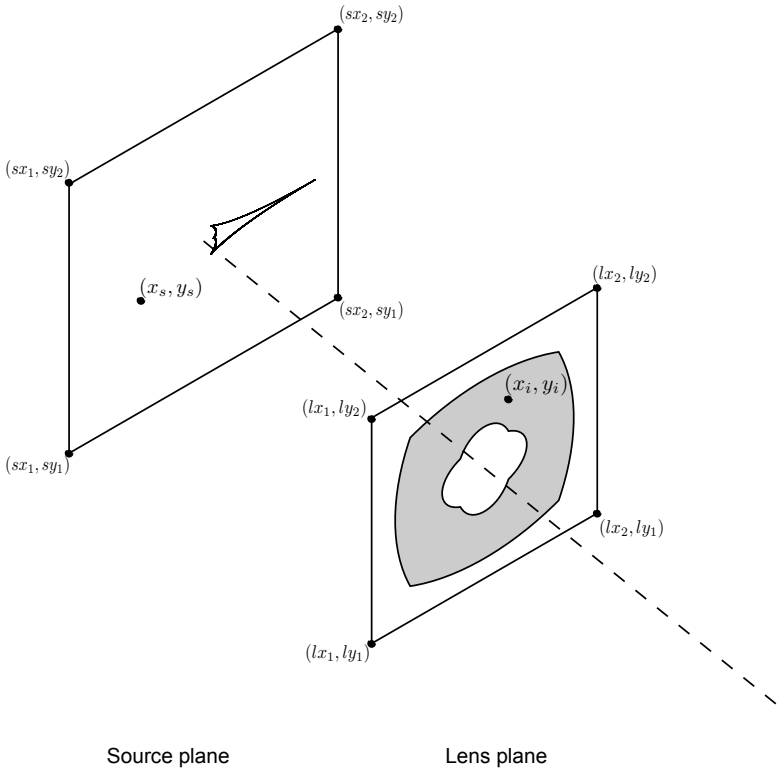

Figure 4.3: Coordinate system used in the source plane and the lens plane. (x_i, y_i) is the image position and (x_s, y_s) is the source position. The image area on the lens plane is coloured in grey.

We first define the dimension of the source plane and lens plane. As shown in Figure 4.3, the size of the source plane is defined by a rectangular box with (sx_1, sy_1) at the bottom left corner and (sx_2, sy_2) at the top right corner, while the lens plane is defined by (lx_1, ly_1) at the bottom left corner and (lx_2, ly_2) at the top right corner, all in the units of the Einstein ring radius.

Solving the image area

The lens equation 2.11 in complex form shown in Chapter 2 can be broken down to two equations representing the real, x, and imaginary, y, components,

$$x_s = x - \sum_{i=1}^{n} \frac{m_i(x - x_i)}{(x - x_i)^2 + (y - y_i)^2} \tag{4.1}$$

$$y_s = y - \sum_{i=1}^{n} \frac{m_i(y - y_i)}{(x - x_i)^2 + (y - y_i)^2} \tag{4.2}$$

where n is the number of lenses, m_i is the mass fraction of the i^{th} lens, (x_i, y_i) are the cartesian coordinates of the i^{th} image position, and (x_s, y_s) are the cartesian coordinates of the source star.

Every image position (x_i, y_i) on the lens plane is mapped to a source position (x_s, y_s) on the source plane by the lens equation. This is not a one-to-one mapping, since a source position can form multiple images on the lens plane and the image positions can be obtained by solving the lens equation. For a binary lens system, we need to solve a fifth-order polynomial and a tenth-order polynomial is required for triple lens systems [63].

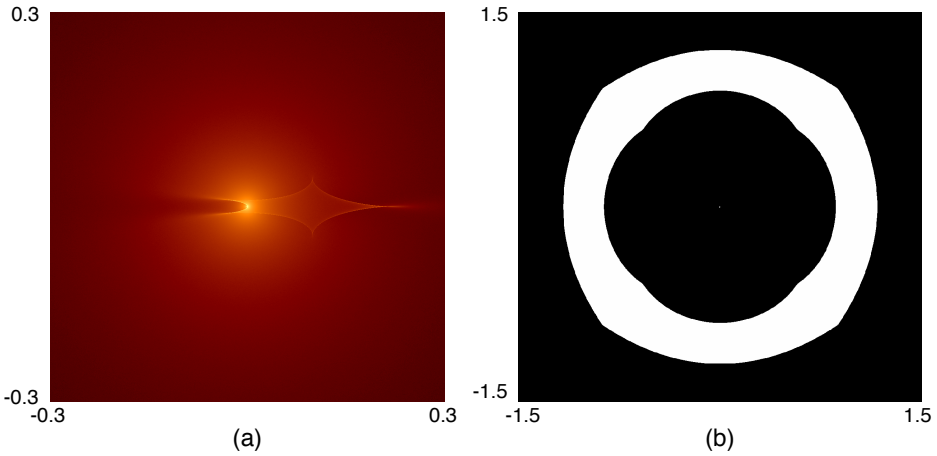

Figure 4.4: (a) is the source plane with boundaries $(sx_1, sy_1) = (-0.3, -0.3)$ and $(sx_2, sy_2) = (0.3, 0.3)$. Rays are shot from the lens plane to form a magnification patten. This shows a resonant caustic with mass fraction $\varepsilon = 0.001$ and projected separation $d = 1.05$. (b) is the corresponding lens plane with size $(lx_1, ly_1) = (-1.5, -1.5)$ and $(lx_2, ly_2) = (1.5, 1.5)$. The distorted ring in white represent the image area. Rays shot by the lens equation within the white region are mapped onto the source plane.

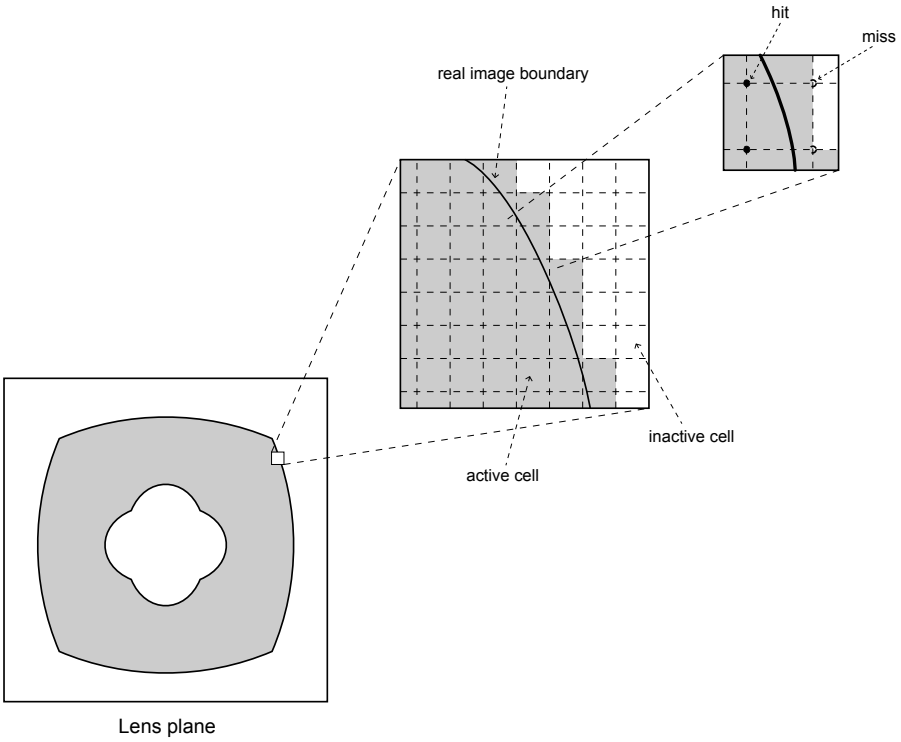

Figure 4.5: Image area and active cells. The area coloured in grey are the image area. Grid cells that coloured in grey are the active cells that form the image area.

It is more efficient to determine the image area by shooting rays from the lens plane inversely to the source plane instead of solving the lens equation directly, since the image area tends to be relatively large as the whole source plane is the target. We utilize the method used in [34] to determine the image boundary on the lens plane. The shape of the image area is usually a thick distorted ring, and its shape and size is changed according to the lens configuration and size of the source plane. Figure 4.4 shows an example of a magnification map and a lens plane with the corresponding image area.

The lens plane is divided into rectangular grids and rays are shot at the four corners of each grid cell. For each grid cell, if any of the rays lensed on the source plane, we label the cell as *active*, as shown in Fig 4.5. The final image area is determined by these *active cells*. The image area can be determined by this method very efficiently.

In the next step, more rays are shot from each active cell and accumulated into pixel bins on the source plane. The higher the resolution of the grid, the more precise the image boundary can be determined. Thus, less time is used on unnecessary computation since rays

shot outside of the real image boundary are going to fall outside the area defined by the source plane and it is not necessary to compute them.

The required lens plane size depends on the size of the source plane. The predefined lens plane dimensions may not contain the whole image area. Therefore, once the whole lens plane has been sampled, we detect whether the image area is being properly covered by the lens plane. If not, we enlarge the lens plane until the image area is contained within the lens plane. The detection of containment of the image area can be done by simply going along the edges of the lens plane. The lens plane needs to be enlarged if any white regions are found at the edge of the lens plane.

Shooting from random positions

Once the image area is determined, rays are shot evenly from the image area on the lens plane to the source plane. One can simply shoot rays with a finer grid from each active cell grid. However, there is problem in shooting rays from a rectangular grid. This grid is transformed by the lens equation into a curvilinear grid. When this transformed grid is superimposed on the rectangular grid of the source plane, unwanted "Moire" patterns can form on the resulting magnification map [64, 65]. This results in a non-smooth amplification of light curves, since the source star integration is affected when the source is positioned over the Moire pattern.

One can increase the signal-to-noise ratio by shooting more rays. However, this increases the computation time and it does not eliminate the Moire pattern. Another approach is shooting rays from a random position within the image area. This can avoid the formation of Moire patterns on the magnification map, but there are two downfalls with this approach. First, the signal-to-noise ratio is lower compared with simply shooting rays from a finer grid, but this can be overcome by shooting more rays to obtain a smooth magnification map. Second, the generation of random numbers takes considerable amounts of computation time. In our sequential version of magnification map generation code, the random number generation alone was observed to take around 35% of the total the computation time. This makes such an approach less practical. But in Section 4.2.2, we discuss an implementation of the parallel version of our reference random number generator that runs on the GPU.

Since tens of millions or even billions of rays are shot for each magnification map, a good random number generator with long period is needed. The Mersenne Twister (MT) [66] is a high performance pseudorandom number generator with extremely long period (up to

$2^{19937} - 1$). Both of our sequential and parallel version of the magnification map generation code are using MT as the random number generator.

Inverse ray shooting

After the image area is determined, a predefined number of rays are shot from random positions within the image area. A practical implementation of such an approach is to shoot rays from a random position within each active grid cell, instead of randomly from the image area. It is easier to count the number of rays per unit area being shot on the lens plane, which is used for scaling the produced light curve to a correct amplification. It also simplifies the calculation of the coordinates for each ray being shot, which saves substantial computation time as many rays are shot. Each ray lensed on the source plane is binned into a pixel. The resulting array of pixels are then stored in a file using the FITS format that is commonly used for storing astronomical imaging data.

Magnification map generation performance on CPU

Figure 4.6 shows performance of magnification map generation on two different CPU processors. The processing time is scaled linearly with workload. The magnification map generation algorithm is fairly simple to implement with multi-threading that utilizes the full power of the multi-core processors. But it is often easier to fire up multiple processes to utilize the multi-threads capability of a multi-core CPU since this problem is embarrassingly parallel. For a magnification map that covers the full Einstein ring radius, around 1 billion rays are required to have good signal-to-noise ratio at the edge of the map. Even for a very fast CPU like the 2.66GHz Core i7 920 and utilizing all four cores, it takes $74.5s \times 5000/3600 = 25.9hrs$ just to generate 5000 magnification maps for a grid search.

The next section discusses the GPU implementation of a magnification map generation algorithm and a performance comparison between the CPU and GPU.

4.2.2 Magnification map generation using GPU

This section discusses the implementation of the GPU version of the magnification map generation algorithm shown in the previous section. Our algorithm utilizes both the CPU and GPU to perform these computations. Moreover, the CPU and GPU are working on different tasks in parallel to achieve the best performance.

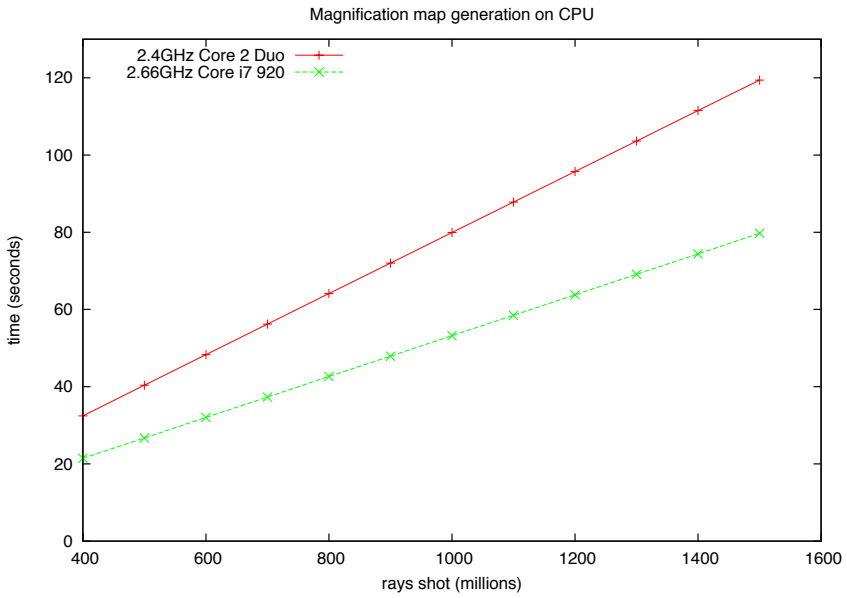

Figure 4.6: CPU performance on magnification map generation using single CPU core. Each experiment is performed six times and the average time is calculated using the last five runs. Note the Core i7 processor turbo boosts its frequency to 2.93GHz when a single thread is running. The uncertainties in each measurement are of the order $\sim$ 10ms. The error bars are smaller than the plot symbols.

The basic procedure of the magnification map generation is similar to the CPU version, but the random number generation and the inverse ray shooting are performed on the GPU. While the image area solving procedure is still performed on the CPU, the active cells are organized in an array of coordinates as input for the GPU kernel. There are two GPU kernels, the random number generator generation kernel and the ray shooting kernel, which will be discussed in detail in the following sections. Figure 4.7 shows the procedure of magnification map generation on a GPU.

Building an active cell array

The image area solving is performed by the CPU instead of the GPU for three main reasons. First, the computation for solving the image area is a memory bounded problem. Even with an 8000 by 8000 pixels high resolution magnification map, there are only $8002 \times 8002 =$

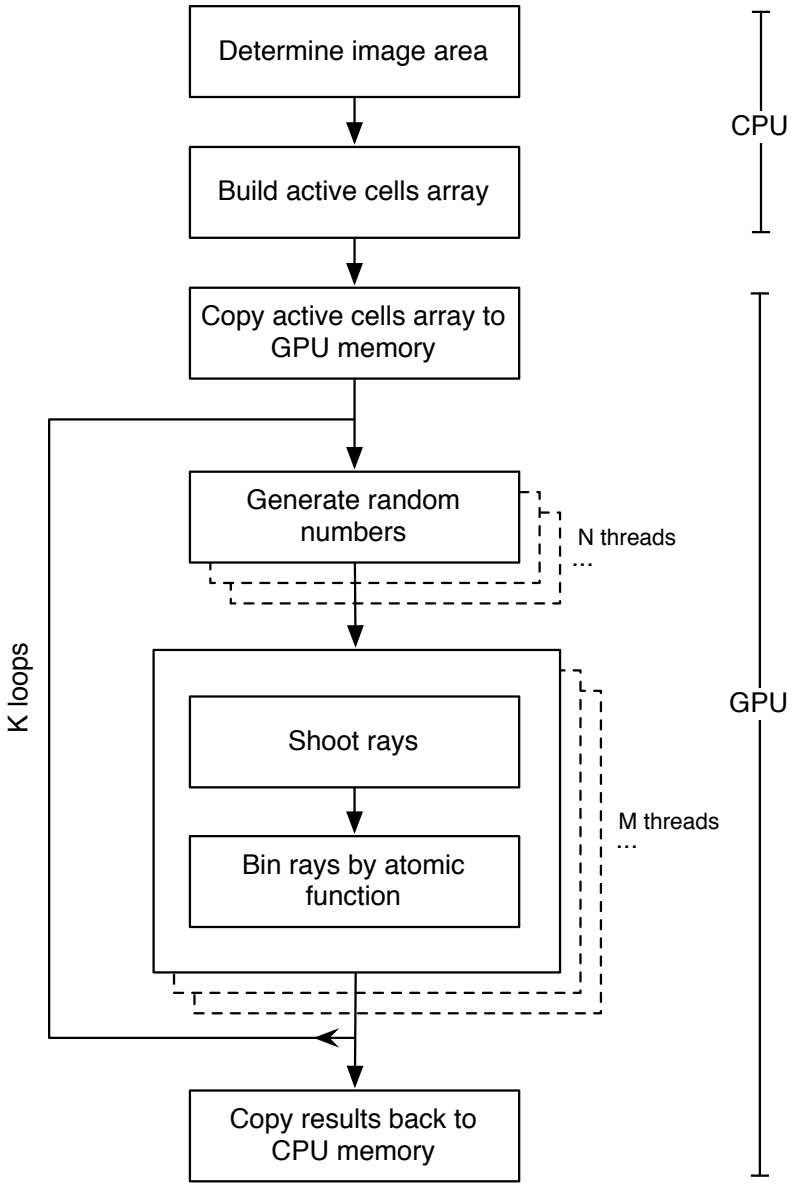

Figure 4.7: General procedure of magnification map generation on GPU. There are two GPU kernel calls, one is for random number generation and the other is for inverse ray shooting. The number of threads N for the random number kernel is dependent on the number of threads M of the ray-shooting kernel. The memory array which holds the random numbers is arranged in a form that allows coalescing memory read by the ray-shooting kernel. The number of K loops depends on the number of rays shot and the memory size for holding the random numbers.

64, 032, 004 rays being shot. It is insignificant compared with shooting a billion rays in the ray shooting procedure in the later stage. For every shot ray, there are four local memory reads and one global memory write operations for a binary lens system. The global memory access latency of the GPU outweighs the benefits of parallel execution of ray shooting.

The second reason is due to the global memory preparation overhead. A large array needs to be first declared on the host (CPU) memory and needs to be initialized to zero as well as on the GPU. After the kernel execution, the array is transferred back from the GPU memory to the CPU host memory. The CPU then loops over the whole array and determine which pixel is belonged to the image area. The array is then re-organized to form an active cells array ready for the GPU ray shooting kernel. The organization of the active cell array needs to be executed on the CPU, it is because to uncoalescing memory operations are involved and are too slow to perform on the GPU. Moreover, the large array need to be first declared and initialized on the CPU memory than transfer back and forth between the CPU and the GPU trough the PCI-E bus. It produces significant overhead in the total image solving computation time. Therefore, the whole image solving procedure is better to be executed solely on the CPU.

Third, the image area solving procedure can be executed in parallel together with the GPU ray shooting kernel by dividing the computation into multiple partitions. Section 4.2.3 discusses in detail how we overlap CPU and GPU computation to achieve better performance.

Random numbers generation by GPU

In the CPU version of the map generation code, it was observed that the generation of random numbers occupies about one third of the computation time. It is a significant part in our computation and it is important to generate random numbers effectively for the ray shooting kernel. The random number generator should generate good quality random number with long period due to millions or billions of rays are being shot in magnification map generation. This subsection discusses a parallel version of the Mersenne Twister random numbers generator execute on the GPU.

We use the parallel Mersenne Twister random numbers generator [67] from the CUDA SDK for generating random numbers. The idea is to have parallel streams of random numbers generated by multiple threads on the GPU. Since Mersenne Twister is a pseudo-random number generator and the sequence of random numbers is determined by the initial

seed, multiple streams of random numbers can be correlated even with a very different seed for each stream. A special offline library, Dynamic Creation of Mersenne Twister parameters (DCMT) [68], is used to compute the initial seeds of each stream. It accepts the 16 bit thread ID as input and produces initial seed values for each stream (thus for each thread), and still maintains good randomness of the final result since it avoids correlated random number streams generated in parallel. The computation of the initial seed from DCMT is time-consuming, it is carried out once and stored in a text file for later use.

Even on the G80 GPU, it takes under a second to produce one billion random numbers. The newer generation GPU is considerably faster compared with the G80. The very high performance and long period sequence of Mersenne Twister is very well suited for our problem. One important note is that the random number remains on the device (GPU) global memory and never transfers back to the host (CPU) memory, this makes significant time saving compared with generating random numbers on the CPU and transfering to the GPU memory. Since there are 2×1 billion $\times$ 16 bits float data that need to be transferred for shooting 1 billion rays, it is impractical to generate random numbers on the CPU.

GPU ray shooting kernel

Once the random numbers are generated, the GPU ray shooting kernel is launched. The input of the GPU ray shooting kernel is not only the random numbers, but also includes the active cells array and the lens system parameters. The random numbers array is ready to be used and is stored in the device global memory. The active cells array is also copied to the device global memory before launching the random number generation GPU kernel. The lens system details are input as parameters with the GPU kernel method call instead of transfering them to the global memory. This is due to the size of the lens system parameters being small and can be stored in the constant memory after passing as parameters. The constant memory is a lot faster than the global memory and is cached. It is perfect for holding small amount of information like the lens system parameters.

Figure 4.8 shows the organization of threads and the memory operations of threads within a warp. Similar to the CPU ray shooting algorithm, a predefined number of rays are shot from each active cell. Each ray is shot from a random position within the active cell. For the GPU ray shooting algorithm, the threads are organized in a way that each thread in a warp performs ray shooting on a different active cell. This organization of threads needed to fullfill two requirements in order to achieve good performance. First, the memory

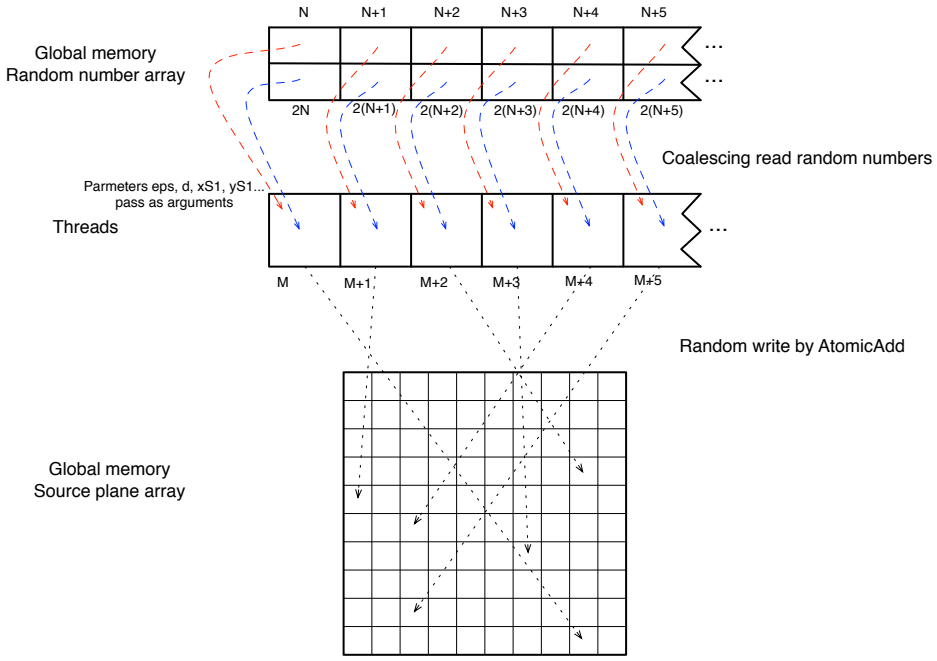

Figure 4.8: Magnification map generation workflow.

read operations for the active cells array and random numbers array needs to be coalesced. Since each thread requires an active cell coordinate as input and every shot ray requires two random numbers, the global memory read operations produces significant delays to the kernel execution. It is important to ensure that the global memory access is in coalescing fashion and large numbers of blocks of thread are launched to hide the memory latency.

Second, each thread should perform ray shooting for a single active cell instead of distributing the workload of a active cell to multiple threads. This way one reduces the number of global memory reads operations as only one active cell coordinate read is required for each thread. More importantly, it reduces the chance of memory write collisions in global memory when binning the rays into pixels. Unlike the CPU ray shooting algorithm, multiple rays are computed at once and are being binned and accumulated on a two-dimensional array. Since the thread execution is managed by hardware on the GPU, one has no control of the thread execution order and multiple threads may write to the same memory location at the same time. This can produce inaccurate results since some rays may be discarded when multiple threads modify the same bin simultaneously.

In order to solve the memory write collision problem, we use the *atomic* instruction for ray binning. This ensures atomicity on each bin. When multiple threads are trying to access the same memory location, the memory write operations are being serialized. This can have

an impact on performance as the memory write operations are executed one after another instead of writing at the same clock cycle in a coalescing fashion. However, when there are no memory write collisions within the same warp, the *atomic* instruction is able to access memory in coalescing fashion.

To ensure the *atomic* instruction causes minimum impact on performance, one can reduce the chances of memory write collisions. This is one of the main reason of the thread organization, for reducing the global memory access being serialized. In our thread organization, each thread within the same warp shoots rays from different active cells and the collision rate of rays being binned to the same pixel is very low. A simulation was carried out to estimate the collision rate in our magnification map procedure. This rate was found to be around 0.06% when shooting 1 billion rays. Therefore, most of the binning is still in coalescing fashion and causes minimal impact to the overall magnification map generation performance. Figure 4.8 shows the random memory write operation by *atomicAdd()* function which adds the suppled value to the existing value at the address specified and stores the result to that address.

Other performance metrics

Beside the coalescing memory access and thread organization discussed in the previous section, there are other aspects to consider in order to achieve even better performance. Since the GPU kernel is shooting billions of rays in generation of a magnification map, any unnecessary instructions within the main loop of the GPU kernel can produce significant overhead. For example, the lens equation can be programmed in general form,

```
for (int n = 0; n < numLens; ++n) {
   dx = xm[n] - xL;
   dy = ym[n] - yL;
   sep2 = dx * dx + dy * dy;

   xS -= epsm[n] * (xL - xm[n]) / sep2;
   yS -= epsm[n] * (yL - ym[n]) / sep2;
}
```

This code has two unnecessary instructions, the comparison instruction (n < numLens) and the increment instruction (++n). Since this piece of code is being executed a billion time when we shoot a billion rays, there are $2 \times 3 \times 1$ billion extra instructions being executed for a binary lens system (with numLens=2, the loop is being executed 3 times). This produces

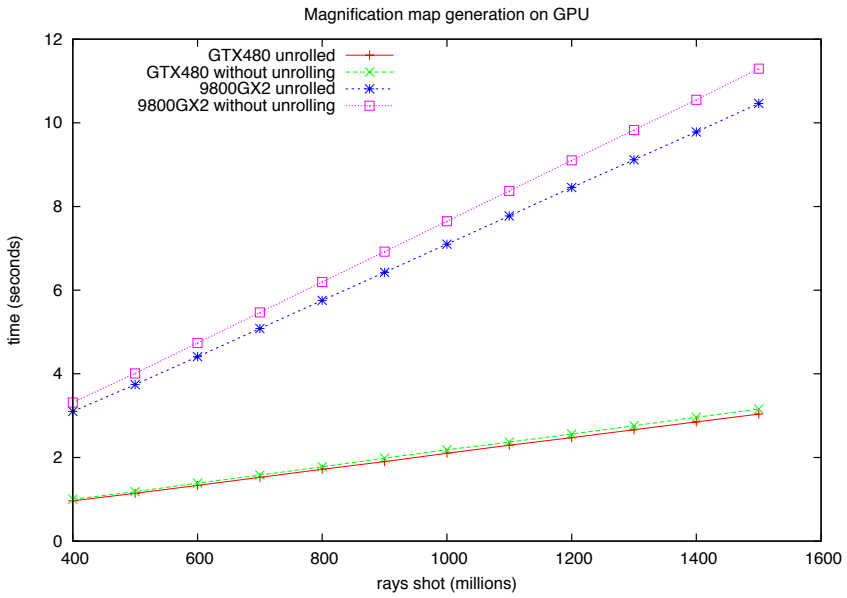

Figure 4.9: GPU performance comparison on magnification map generation between unrolled kernels and kernels without unrolling. The unrolling alone leads to an improvment around 8% of the overall performance for both GPU. The error bars are smaller than the plot symbols.

significant overhead to the overall performance. Therefore, it is good practice to unroll small loops in order to avoid unnecessary instructions being executed within the main kernel loop, like the following code,

```
// unroll the binary lens equation
// 1st lens mass
dx = xm1 - xL;
dy = ym1 - yL;
sep = dx * dx + dy * dy;
xS -= eps1 * (xL - xm1) / sep;
yS -= eps1 * (yL - ym1) / sep;

// 2nd lens mass
dx = xm2 - xL;
dy = ym2 - yL;
sep = dx * dx + dy * dy;
xS -= eps2 * (xL - xm2) / sep;
yS -= eps2 * (yL - ym2) / sep;
```

Figure 4.9 shows the performance improvement by unrolling the lens equation. The unrolling alone improves 8% of the overall performance. The CUDA compiler usually unrolls small loops automatically, but it is not guaranteed to work in all different kinds of loops. The compiler is unable to unroll the lens equation in this case.

4.2.3 Parallel magnification map generation by both CPU and GPU

Solving the image area in parallel

We may speed up the computation for solving the image area by using the GPU. The GPU can compute multiple lens equations simultaneously, thus achieving a higher throughput compared with the CPU. However, the end results have to be transferred from the GPU memory back to the CPU memory though the PCI-E bus. The active cells coordinate is then arranged in an array, since the GPU ray shooting kernel described requires a list of active cell's coordinates as input to achieve maximum performance. The overall performance with this approach is about 50% faster than using the CPU. Actually, the GPU arithmetic throughput can be a lot faster, but the PCI-E bus acts as a bottleneck since it is a lot slower than the memory bus.

One has to determine the balance between precision of the image boundary and number of unnecessary computations later in the ray shooting step. For example, consider a lens plane with dimension (lx_1, ly_1)=(-1.2, -1.2) and (lx_2, ly_2)=(1.2, 1.2), and grid resolution defined by grid scale, $gscale$, the width of a grid cell in Einstein ring radius. With a reasonably dense grid resolution, $gscale = 0.001R_E$, there are $(lx_2 - lx_1)/gscale + 1 = 2201$ grid points on the x-axis and $(ly_2 - ly_1)/gscale + 1 = 2201$ grid points on the y-axis, with a total of $2201 * 2201 = 4844401$ grid points to be evaluated by the lens equation. It takes around 1 second on a 2.4GHz Core 2 Duo CPU using a single core. This is insignificant compared to generating a magnification map with one billion rays shot using the CPU which takes minutes to compute, as the time for searching active cells only accounts for $< 1\%$ of the total computation time. However, the same ray shooting computation on the GPU only takes around 2.6 seconds on a NVIDIA GTX470 graphics card. The time for solving the image area suddenly becomes a significant part in the computation, which accounted for $> 25\%$ in the total map generation time.

When a microlensing event is being modelled, thousands of maps may be generated in order to find the right model. In order to utilize the GPU efficiently when generating

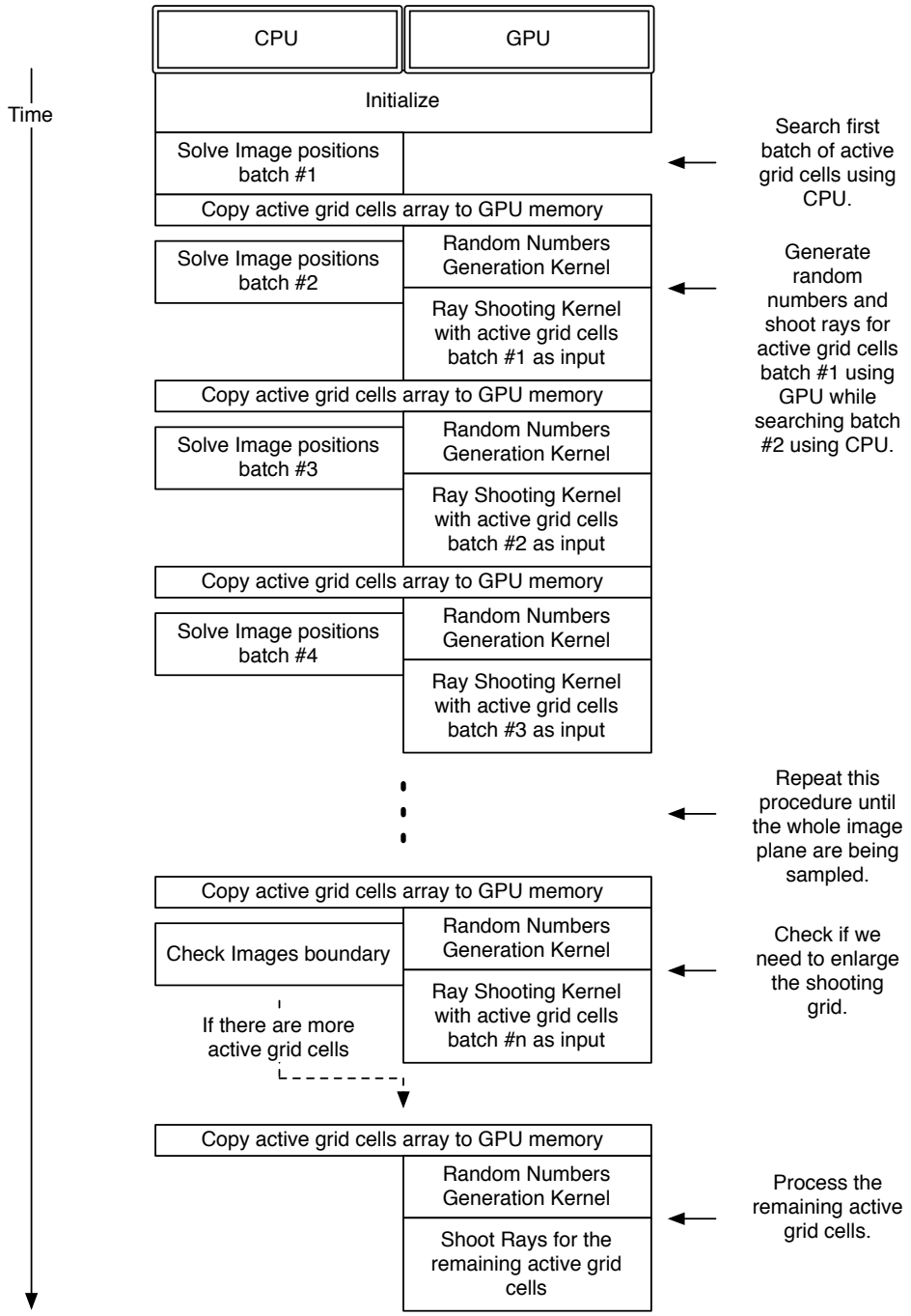

Figure 4.10: Parallel magnification map generation by both CPU and GPU.

multiple maps, the image searching and the ray shooting can be done in parallel. Figure 4.10 shows the procedure in parallel magnification map generation. Since the host gets back in control once the GPU kernel is launched, we can perform tasks on the CPU and GPU simultaneously. We first solve the image positions (active grid cells) of an equally divided

area on the image plane by the CPU, then launch a GPU kernel to perform ray shooting from those active grid cells. The host (CPU) will be able to perform another task once the GPU kernel is launched. We then solve the image positions of the second pre-divided area on the image plane by the CPU and launch the GPU kernel once it is done. The whole procedure is repeated until all the divided area on the image plane is searched and processed.

4.2.4 Performance comparison

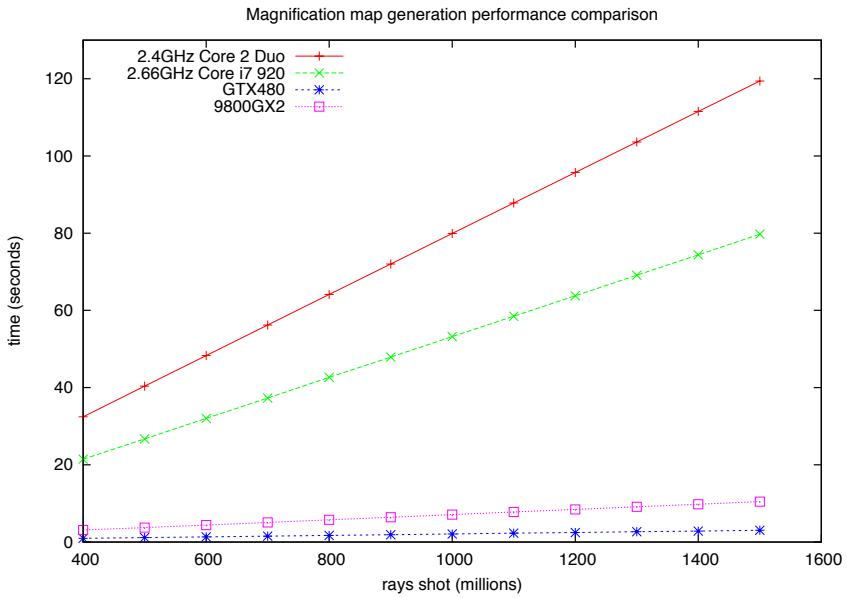

Figure 4.11: Performance comparison between CPU and GPU on magnification map generation. Each experiment is performed six times and the average time is calculated using the last five runs. The difference between each experiment is the number of rays shot. The uncertainties are comparable to the symbol sizes.

Figure 4.11 shows the comparison of magnification map generation performance between CPU and GPU with different generations of hardware. The CPU version of the magnification map generation code is multi-threaded and utilizes all the CPU cores for processing. The GPU still out performs the CPU by a large margin even though all CPU cores are being used.

4.3 Track extraction

The generation of magnification maps is just the first step in computing a theoretical microlensing light curve. Each pixel on the magnification map represents the amplification of the source star for a given lens system. A theoretical light profile can be obtained by reading the value on the magnification map along a path on the magnification map which represents the source star track. The biggest advantage of this approach is that the finite source amplification can be computed directly by integrating the source area on the magnification map. Moreover, multiple light curves with different source tracks and source sizes can be computed by using the same magnification map.

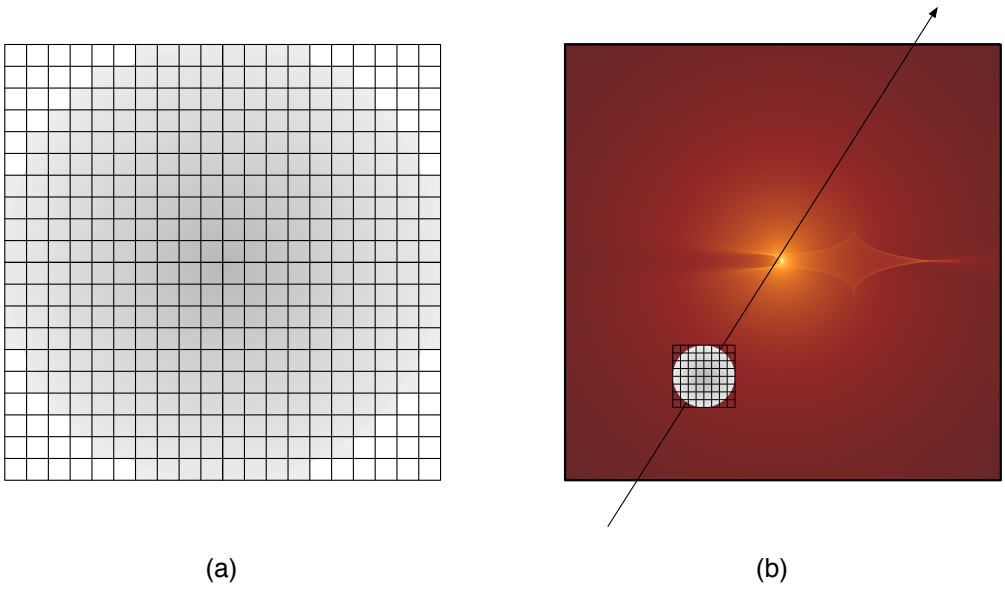

Figure 4.12: Schematic of source star limb darkening profile and track extraction across a magnification map.

As discussed in section 4.1.1, a magnification map for binary microlensing represents the mass fraction ε and projected separation d, which are the most important and interesting physical parameters we would like to obtain by modelling a microlensing event. The path on the magnification map for light curve extraction is the source star track that is described by another set of physical parameters. Therefore, the best fitting theoretical light curve extracted from the magnification map represents the best fitting model of a given ε and d. Hundreds of thousands of light curves are usually computed on each magnification map in order to search for the best fitting model, and thousands of magnification maps are usually required for searching through in each modelling task. This is actually a very

computationally intensive task as it can take months of CPU time on a single computer.

As with magnification map generation, track extraction is also an embarrassingly parallel problem that repeatedly performs the same instructions on multiple data sets. This is very well suited to be computed by the GPU. The GPU can speed up this task significantly by reducing the computation time from weeks to hours on just a desktop computer with a fraction of the cost of a large cluster computer.

In this section, we first discuss the procedure of extracting a theoretical light curve from the magnification map using CPU. Second, we explain the GPU implementation of light curve extraction followed by performance comparison between CPU and GPU. Additionally, we discuss various techniques to optimize the performance of the GPU implementation. Finally, we discuss the trade off between performance and accuracy of using magnification maps to compute a theoretical light curve.

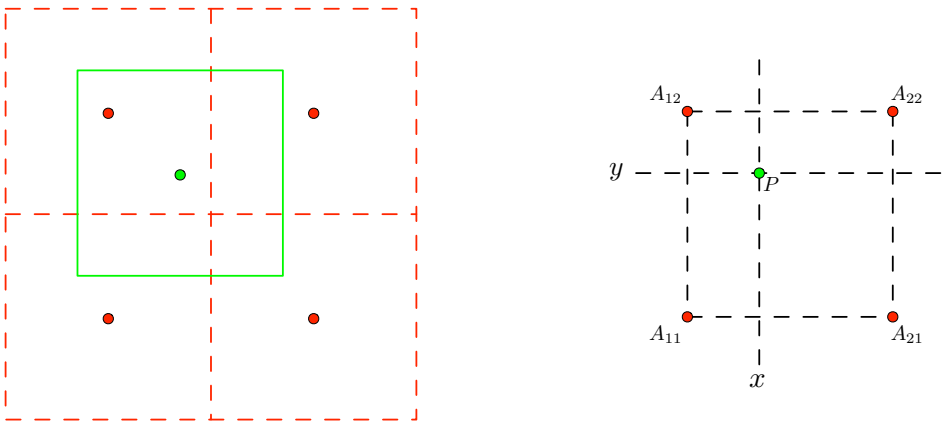

Figure 4.13: Bilinear interpolation. The four red dotted squares are pixels on the magnification map. The green solid square is the target position for amplification read out. The amplification of the target position is calculated by interpolating between the four closest pixels.

4.3.1 Track extraction using CPU

For uniform rectilinear motion of the source star, the path that it takes across the magnification map is defined by the parameters t_0, t_E, u_0 and ϕ. The position of the source on the

magnification map in Einstein units at time t can be calculated by

$$x = \frac{(t - t_0)}{t_E} \cos(\phi) - u_0 * \sin(\phi) \tag{4.3}$$

$$y = \frac{(t - t_0)}{t_E} \sin(\phi) - u_0 * \cos(\phi) \tag{4.4}$$

Therefore we can compute a list of source positions as functions of time with parameters t_0, t_E, u_0 and ϕ. The finite-source calculation is done by intergrating the source area which is defined by the Einstein source size radius ρ. Figure 4.1 shows the geometry of the source-star track in relation with the related physical parameters.

We first create a source star model with the same resolution as the magnification map. Each pixel on the source model is stored with scale factor computed by the limb-darkening profile as shown in Figure 4.12. The source star model acts as a mask as we compute the amplification on each source model pixel position. We compute the amplification by interpolating between the closest four pixels on the magnification map and weighting by the scaling factor on the source star model. The final finite-source amplification is calculated by summing the normalized values of the weighted amplifications of the source star model.

Bilinear interpolation

Since the shot rays are binned into a pixel grid, each source position required for amplification reading will be surrounded by four pixels on the magnification map. Therefore, we can compute the amplification by interpolating between the four pixels. Bilinear interpolation is an extension of linear interpolation which works on a two-dimensional grid. The amplification of any position on the magnification map is obtained by interpolating between the four closest pixel values.

Figure 4.13 shown an example of bilinear interpolation on the magnification map. The green solid square is the target position for amplification calculation, which is located at (x, y) between the four closest pixels represented by the red dotted square. The amplification at (x, y) can be calculated by Equation 4.5 when a unit square is used for the coordinate system.

$$A_{xy} = A_{11}(1 - x)(1 - y) + A_{21}x(1 - y) + A_{12}(1 - x)y + A_{22}xy \tag{4.5}$$

where A_{xy} is the amplification at (x, y) calculated using interpolation between the four closest known amplification values A_{11}, A_{21}, A_{12} and A_{22} shown in Figure 4.13.

Limb-darkening profile

A limb-darkening model is first computed to created a pixelated source star model which acts as a mask for weighting each computed amplification. A limb darkening profile of the following form is used

$$S = 1 - \lambda_1(1 - \cos(\alpha)) - \lambda_2(1 - \sqrt{(\cos(\alpha))}) \tag{4.6}$$

where α is the angle between the line of sight and the emitted radiation. The parameters λ_1 and λ_2 are two coefficients describing the model. These can be fixed in parameter searching using a standard limb-darkening model appropriate to the colour of the source star [69] or we may allow them to be free parameters.

4.3.2 Track extraction using GPU

In the modelling process, computing the theoretical light curves from the magnification map is the most computationally expensive task. Although we are able to extract a light curve from the magnification map in just milliseconds, tens of thousands of light curves are usually computed from a magnification map and thousands of magnification maps are used in a grid search. Therefore, tens of millions of theoretical light curves are computed in a typical modelling process and it takes weeks or even months of CPU time.

The track extraction task is an embarrassingly parallel problem, like magnification map generation discussed in section 4.2.2. Each source integration use the same algorithm, thus, the same instructions are applied on multiple datasets. It is a problem that is well suited to implement on the GPU. As with the magnification map generation problem, we cannot simply use the CPU algorithm and implement it on the GPU. We need to solve the problem from the GPU perspective in order to achieve good performance.

This section shows four implementations of track extraction using GPU. We first discuss the easiest implementation but also the slowest, followed by three more implementations using different optimization techniques to improve performance. We also discuss the effectiveness of such optimization techniques on different generations of hardware.

Kernel 1 - global memory + CPU sum

There are two ways to parallelize the track extraction task. First is to compute each source integration in parallel. Second is to compute each bilinear interpolation in parallel. We

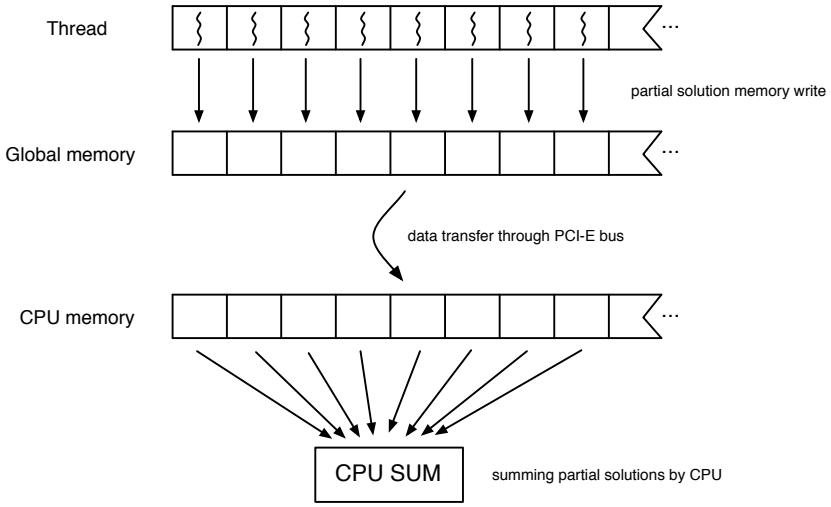

Figure 4.14: In Kernel 1 and 2, each thread writes the partial solution to GPU global memory. The partial solution is transferred back to the CPU memory. The final solution is computed by using the CPU by summing partial results from each block.

can achieve both by organizing each block to compute a source integration, which means each block of threads shares the work on bilinear interpolation for each source integration. Therefore, multiple bilinear interpolations are computed within a warp and multiple source integrations are computed by multiple MPs. Note that the number of blocks launched in each kernel call depends on the number of source integrations. The advantage of this approach is that it is simple to implement and easy to organize the threads. The disadvantage is that each kernel call may not achieve the maximum occupancy. There will be more in depth discussion in Section 4.3.3.

The magnification map is first loaded into the global memory on the GPU. We create a two-dimensional block of threads in each block to compute the source integration. Since the source model is a two-dimensional grid, the two-dimensional block of threads is able to equally distribute the jobs. The computed amplifications of each grid point on the source model are weighted by the limb-darkening model and stored in a temporary array in global memory. These partial solutions are transferred back to the CPU memory at the end of the kernel call. The final amplification is calculated by summing the partial solutions of each block. Figure 4.14 shows this procedure.

In order to achieve good performance, the magnification map should stay in the device global memory for multiple track extractions. The time required to copy the magnification map to the GPU global memory is longer than the computation time of a track extraction.

Therefore, the performance benefit of track extraction will be outweighed by the time to transfer the magnification map from the CPU memory to the GPU memory through the PCI-E bus. Since the host (CPU) code is executed after each GPU kernel call, a global device memory pointer needs to be maintained in order to locate the magnification map stored in the device global memory. In order to access the global device memory pointer from the kernel, we need to make a copy of the pointer to the device memory. Since the usual device pointer like *d_mapimg* is stored on the host side, the kernel does not recognize its existence, therefore we need to make a copy of the pointer on the device side by using the following commands.

```
float *d_mapimg; // array pointer for magmap image for device
__device__ float *dd_mapimg; // array pointer for magmap image for device

cudaMemcpyToSymbol("dd_mapimg", &d_mapimg, sizeof(dd_mapimg), 0, cudaMemcpyHostToDevice);
```

Kernel 2 - texture memory + CPU sum

The main bottleneck in kernel 1 is memory read operations, as each bilinear interpolation requires four global memory reads. Since reading from global memory has a 400-600 cycle penalty, this is a substantial restraint on performance. Although we can hide the memory latency by using a large number of blocks, the number of blocks is dependent on the number of source amplification calculations. Every track extraction may have different numbers of source amplification calculations, so this is not a reliable method to maximize performance in this situation. One can use texture memory instead of global memory for holding the magnification map to take advantage of the caching capability of texture memory.

As we discussed in Chapter 3, texture memory is a read-only memory which does not require coalescing access and is cached in texture cache. The result can be directly read from the texture cache by only one memory read when there is a cache hit, otherwise one memory read from the device memory when a cache misses. The texture cache is designed to be accessed efficiently in two-dimensional spatial locality. Therefore threads within the same warp reading memory address close together in two-dimension can maximize performance gain by texture cache. This makes texture memory the ideal memory type for storing the magnification map.

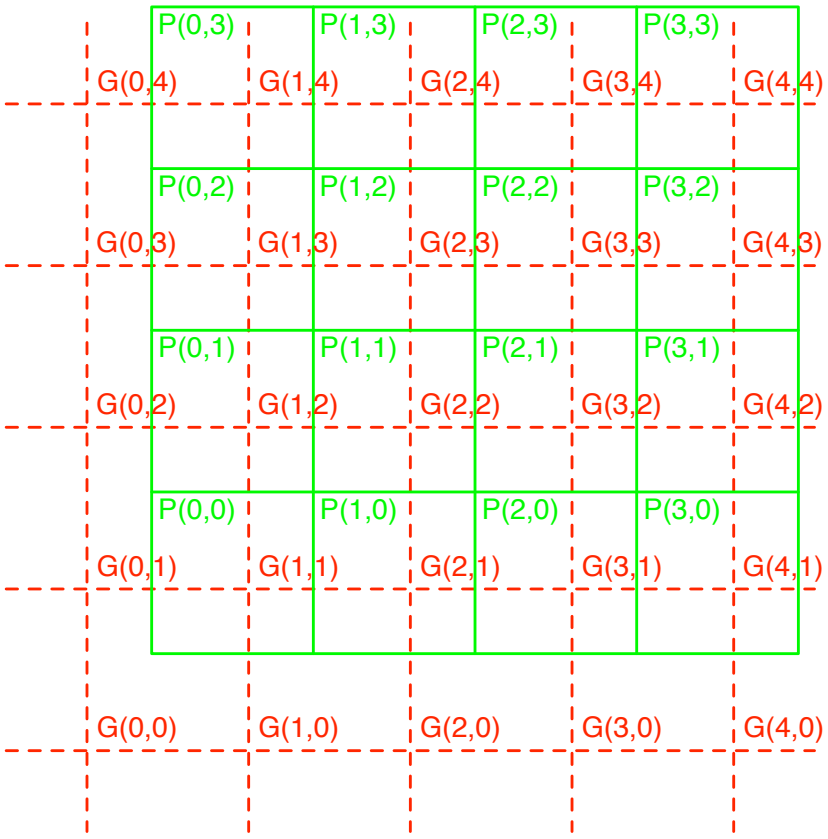

Figure 4.15: This figure shows threads within a block taking advantage of locality to utilize caching capability of texture memory. The red grid G represents the magnification map stored in texture memory. The green grid P represents the source star model. Each thread computes amplification of the P grid by interpolation between the four G grids. For example, the thread to compute amplification for P(0,0) performs memory read from grids G(0,0), G(0,1), G(1,0) and G(1,1). Each thread first performs a memory read on the bottom left G grid, then the top left grid, followed by bottom right grid and read the top right grid last. For each thread, the value of grid G required by the second memory read is already read by other threads from the first memory read, there is a cache hit and the data can be attained through the cache without the global memory read penalty. The same operation happens to the third and fourth memory read.

In our kernel, each block of threads is organized into a two-dimensional array and reading memory address close to each other. Each bilinear interpolation has four memory reads. We can take advantage of locality when amplification calculations are adjacent to threads in a block. The following code shows the thread organization in this kernel.

```
const int tx_id = threadIdx.x;
const int ty_id = threadIdx.y;
const int b_xdim = blockDim.x;
const int b_ydim = blockDim.y;

// loop from bottom left hand coner
for (i=tx_id; i<starimg_npix; i+=b_xdim) {
  for (j=ty_id; j<starimg_npix; j+=b_ydim) {
    x = ...;
    y = ...;
    f = bilinear_interpolation(x, y);
    ...
  }
}

float bilinear_interpolation(int x, int y) {
  f_00 = read_G00();
  f_01 = read_G01();
  f_10 = read_G10();
  f_11 = read_G11();

  f = bi_linear(f_00, f_01, f_10, f_11);

  return f;
}
```

The threads within a block are organized in the way that the threads in each warp access the adjacent memory block in 2D. The value of the first memory read for bilinear interpolation of each thread is stored in the texture cache. The second memory read of each interpolation will have a cache hit since the data is loaded into the cache from the first memory read. The same happens to the third and fourth memory reads. Therefore, three out of four memory reads will have a cache hit within a warp. Figure 4.15 shows blocks of threads taking advantage of spatial locality in texture memory caching.

Since three out of four memory reads is from the texture cache, we significantly minimize the penalty of device memory access. Although it does not reduce the fetch latency, launching large numbers of blocks in kernel calls help in hiding the memory latency. Figure 4.17 shows the performance gain from Kernel 2 compared with Kernel 1. We first compare the performance between Kernel 1 and Kernel 2 of the 9800GX2 GPU. There is a significant

improvement in performance from Kernel 2 with an average 75% gain in performance from using texture memory instead of global memory to store the magnification map. This is expected since we save three out of four memory accesses to global memory and the memory access is the bottleneck of this kernel. However, there is only an average 13.5% performance gain in GTX480 GPU. This shows the capability of the new caching system in Fermi architecture. There is cache memory for global memory in Fermi, similar to the CPU's cache memory hierarchy. We will discuss these issues in detail in Section 4.3.3.

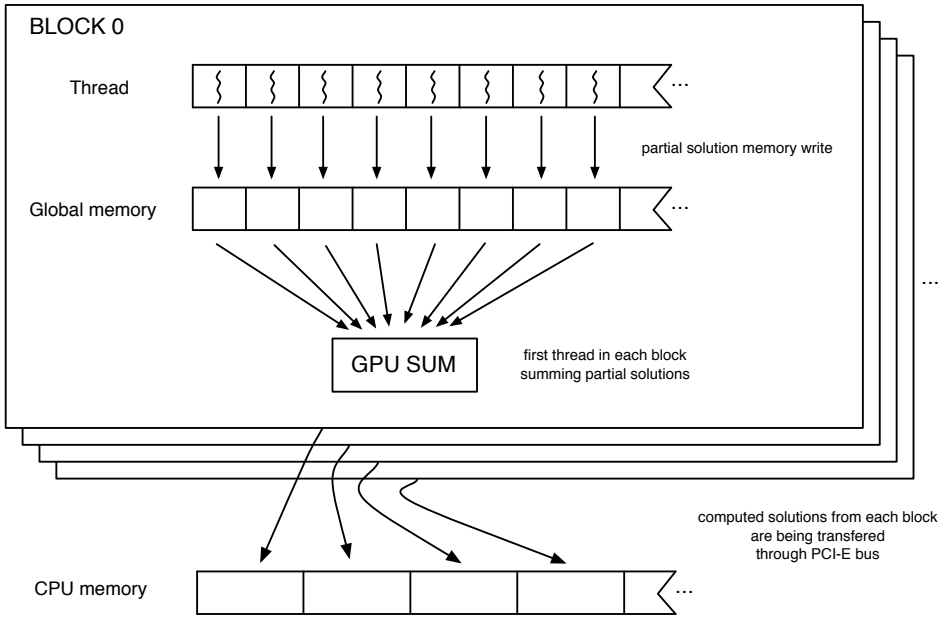

Figure 4.16: In Kernel 2, the partial solutions are being calculated by the first thread of each block and only the final solution is being transfered back to the CPU memory. The final solution is computed by the GPU before transfer back to the CPU memory.

Kernel 3 - texture memory + GPU sum

For each magnification map, we extract tens of thousands of light curves. In Kernel 2, the partial solution of a light curve is transfered back to the CPU memory through the PCI-E bus and the final solution is computed by the CPU. The PCI-E bus can be another bottleneck since it has a lot lower bandwidth than GPU memory. We should minimize the amount of data transfer between the host and the device to achieve the best overall performance.

In order to minimize the impact of the data transfer between the CPU memory and GPU memory, we compute the final solution using the GPU instead of the CPU. However,

this creates a problem as the final amplification is the summation of all partial results from threads within a block. Threads within a block can only communicate by either shared memory or global memory. Usually the shared memory is the first consideration of choice, but it has very limited size. Even the new Fermi architecture GPU has only 48KB shared memory per SM. Assuming we launch a kernel with 3000 source integrations, we require memory size = numSource * dimBlock.x * dimBlock.y * sizeof(float) = 3000 * 8 * 8 * 4 = 768KB in total. This means only 768 / 48 = 16 blocks can be launched for each kernel call, which will under utilize the GPU and give very low performance. Therefore, we cannot use shared memory in this kernel.

Another way is to use the global memory for temporary storage. We create a temporary array in global memory to store the partial solutions from each thread. The first thread in each block reads the partial solutions produced by other threads in the same block and performs summation operations to get the final answer, as shown in Figure 4.16. Although only the first thread of each block is doing the computation, the data being transfered from the GPU memory to the CPU memory is smaller. It is still a lot faster than transfering a large block of partial results back to the CPU and computing the final result. We save time on the PCI-E bus memory transfer.

Note that we need to synchronize threads within a block before we can perform the summation operation. We can use the call *__syncthreads()* to place a barrier within a thread. All threads within that thread block will wait until all threads reach the same instruction. This ensures the integrity of the final result. The following code shows the synchronize barrier and uses the first thread in each block to perform the summation.

```
const int tx_id = threadIdx.x;
const int ty_id = threadIdx.y;
const int b_xdim = blockDim.x;
const int b_ydim = blockDim.y;
const int bid = blockIdx.x;
const int threadNum = blockDim.x * blockDim.y;

...

int temp_pos = tx_id + ty_id * b_xdim + (threadNum * bid);
d_temp[temp_pos] = sum;

__syncthreads();
```

```
float total = 0;
if ((tx_id + ty_id) == 0) {
  for (i=0; i<threadNum; i++) {
    index = i + (threadNum * bid);
    total += d_temp[index];
  }
  d_amp[bid] = total;
}
```

There is on average a 17% gain in performance from Kernel 2 to Kernel 3 on 9800GX2 GPU and there is on average a 11% gain in performance in GTX480 GPU, which is shown in Figure 4.17. The performance gain depends on the performance of the CPU. The 9800GX2 GPU is paired with a 3.4GHz Core Duo CPU and the GTX480 GPU is pair with a 2.66GHz Core i7 CPU. We cannot compare their results as the performance of the CPUs is different.

Kernel 4 - texture memory + GPU sum + texture memory source star model

There is another memory access task in the track extraction kernel that can be optimized. The source star model is accessed multiple times for each kernel call depending on the number of source integrations. Since the source star model acts as a mask and is used to weigh the amplification with a limb-darkening profile, the same piece of memory is being accessed repeatedly by each block. In kernel 1, 2 and 3, global memory is being used to store the source star model. We can use the same technique to optimize this memory access as in kernel 2 by storing the source star model in texture memory and using its caching capability.

Figure 4.17 shows there is on average a 31.3% performance gain in 9800GX2 GPU. This shows we have a cache hit around one third of the time. This is again a significant performance improvement by using texture memory. However, the GTX480 GPU does not have any performance gain. There is even an average of 0.8% performance loss by using the texture memory.

The lack of performance improvement over the texture memory in GTX480 GPU is because of the caching capability in Fermi architecture. We can see considerable performance improvement in the G80 series GPU, but not the new Fermi architecture GPU. Since the Fermi architecture incorporates small amounts of L1 and L2 cache, the source star model is

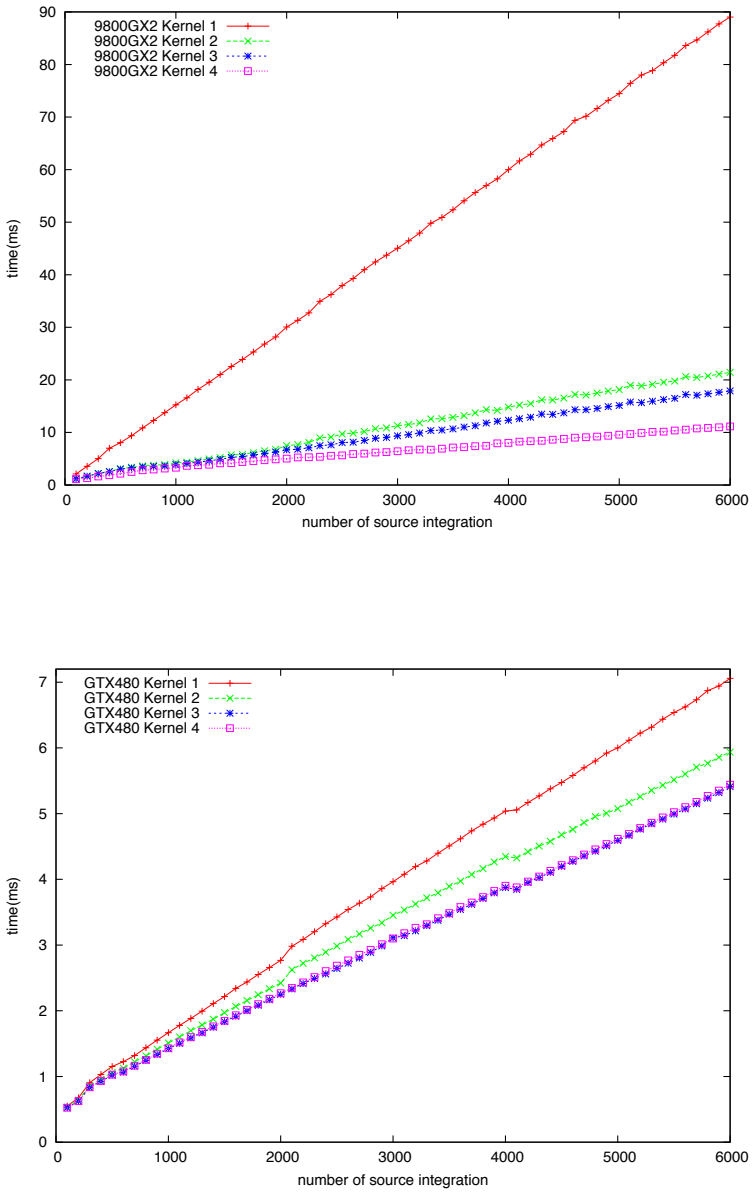

Figure 4.17: Top figure (a) shows the performance on track extraction in four different kernels on G80 series GPUs and and the bottom figure (b) shows performance on Fermi GPUs.

being cached even if we only use global memory. Moreover, the internal cache is even faster than the texture cache. This is one example to show Fermi's new cache memory capability. We will discuss this aspect in detail in the next section.

4.3.3 Performance comparison and discussion

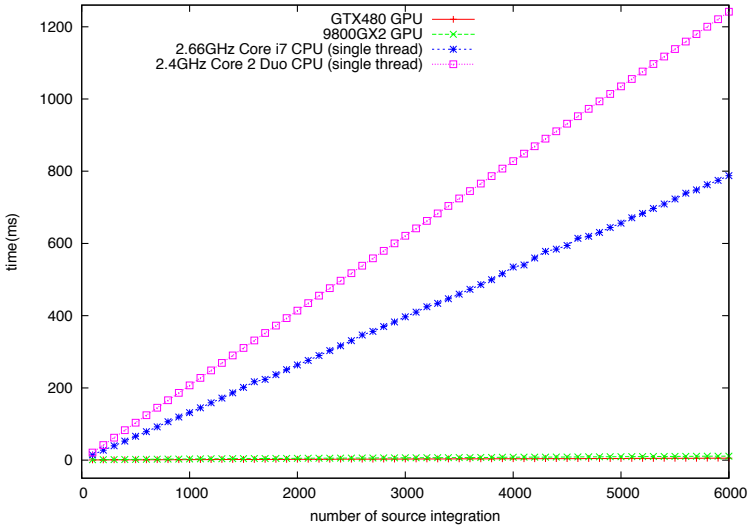

Figure 4.18: CPU vs. GPU on track extraction.

The optimization techniques discussed in the previous section were aimed to compute theoretical light curves from the magnification map as quickly as possible, since it is very computationally expensive to perform them on the CPU. Figure 4.18 displays the performance benefits from the GPU compared with the CPU. For example, with 3000 source integrations on the magnification map, the 2.66GHz Core i7 CPU with turbo boost to 2.93GHz takes 397 milliseconds while the 480GTX GPU takes only 3.099 milliseconds. The GPU is 128 times faster than the CPU. In a grid search, the number of light curves computed for each magnification map is dependent on the event being modelled as well as optimization methods and strategies.

There is a significant speedup using the GPU over the CPU. But when modelling a microlensing event, the speedup is less than the above figures as there are other extra computations needed to be done by the CPU, like computing chi-square for each light

curve as well as determining the next parameter set by the optimization methods. The above comparison is based on a single core of CPU performance with non-vectorised code compiled by GCC with the -O3 optimisation flag. If vectorized libraries such as Intel IPP (Integrated Performance Primitives) [70] were used, there is a potential speed up. The following sub-sections discuss in detail the optimization techniques in kernels 1 to 4.

A practical performance comparison example, a high workload microlensing modelling case, it takes a 2.93GHz Core i7 CPU 69 minutes to search through a magnification map using a single CPU core (including the magnification map generation time and light curve extraction for chi-square computation, 3000 amplifications points are being computed on a single light curve and around 10,000 light curves are being computed on each magnification map). For a parameter grid with 3000 maps, it will take 143 days to complete the computation. While a 480GTX GPU will only take 34 seconds to search through a magnification map, therefore the GPU only takes a total of 28 hours to complete the same computation which is 122× faster than the CPU.

Coalescing memory access and hiding memory latency

The basic and most effective way to achieve good performance using the GPU is to ensure coalescing memory access and to launch large numbers of threads to hide the memory latency. In kernel 1, all threads within a warp access memory location sequentially with a word apart. Therefore, only one memory read cycle is required for all threads within a warp. On Fermi GPU, the warp size is 32, thus 32 words are read from the global memory in one memory read cycle. If the memory access pattern is non-coalesced, 32 separate memory reads are required for all the threads within a warp. Therefore, coalescing memory access is 32 times faster than non-coalescing memory access in this case.

For the track extraction in kernel 1, threads in each block are organized as a two-dimensional 8 by 8 block of threads. The number of blocks is dependent on the number of source integrations required. For 3000 source integration calculations, $8 \times 8 \times 3000 = 192000$ threads are launched in a kernel call. The GPU can hide the memory latency by launching large numbers of threads. When a block is waiting for memory read (with 300-400 cycles delay), other blocks in the same MP are executed while waiting for the memory access. Therefore, launching large numbers of blocks in a kernel call will help to minimize the memory access latency. Since the GPU can manage a large number of threads concurrently, this is often a very useful strategy to improve performance.

Spatial locality and texture cache

The optimization strategy in kernel 2 utilizes texture memory instead of global memory. The bottleneck in the track extraction kernel is memory reading from the magnification map. Since each amplification calculation requires four memory reads for the bilinear interpolation calculation, a source model with resolution 32 by 32 and a light curve with 3000 source integrations needs $32 \times 32 \times 4 \times 3000 = 12288000$ memory read operations. Even though coalescing memory reads and launches a large number of blocks help to minimize the problem, the large number of memory reads still causes a bottleneck in kernel 1. More importantly, at least 3/4 of memory reads are overlapped as threads within the same block require the same value to perform the bilinear interpolation. Reusing the read values from the magnification map is the key to further improve performance.

Since the global memory is not cached (for hardware before Fermi), the read values cannot be reused. We can improve performance by using texture memory instead of global memory, as texture memory is cached and does not require the memory address to be accessed in coalescing fashion. The bilinear interpolation computation largely operates on data that are used by other threads. Therefore, there should be a significant improvement in performance by using texture memory. There is around a 75% improvement in performance by using texture memory instead of global memory for storing the magnification map in 9800GX2 GPU, as shown in Figure 4.17. The amount of performance improvement matches the logic of the algorithm, as three out of four values read from the magnification map of each thread are used by other threads within the same block. In order to take advantage of the texture cache, the required memory values of the second, third and fourth memory read operations are already read by the adjacent threads in the previous instruction. This ensures threads within the same warp have good spatial locality for the texture cache. Moreover, a warp of threads can only be collected by threads in the same block and texture cache can only be shared by threads within the same warp. Therefore, we optimize the use of texture cache by the way we organize the threads.

One may need to beware of the limitation on the texture memory size. The maximum width of 1D texture reference is only 8192 when using a CUDA array, the maximum width of 1D texture reference is 2^{27} when using linear memory on the G80 and GT200 architecture. The Fermi architecture supports 1D texture width up to 65536 when using a CUDA array as shown in Table 3.2.

L1 and L2 cache in Fermi

However, compared with a 75% improvement in the 9800GX2 GPU, the Fermi architecture GTX480 GPU has only around a 13.5% improvement in performance when using texture memory. The Fermi architecture has up to 48KB of L1 per SM and total of 768KB of L2 cache shared by all SMs. It is the L1 cache that makes the difference in the amount of performance improvement as the global memory can be cached in Fermi. Unlike the shared memory, the L1 and L2 cache is hardware managed. There is no way to control the behaviour of the L1 and L2 cache. Since there is only a maximum 48KB of L1 cache per SM, which is shared with all the blocks executed by the same SM, there are still improvements in performance by using the texture memory, since the texture cache is operated at warp level and our algorithm is optimized to take advantage of the spatial locality in texture memory caching. However, there are some cases where the L1 cache performs even better than the texture cache.

In kernel 4, the source star model is bound to the texture cache instead of using the global memory. Since each block uses the same source star model, there should be an improvement in performance by using texture memory. As shown in Figure 4.17, there is on average a 31.3% performance improvement in 9800GX2 GPU. However, there is a 0.8% performance drop in the GTX480 GPU. Since the size of the source star model is small, the 48KB of L1 cache per SM is enough to store the entire source star model. Therefore, the source star model in the global memory is being cached and reused by threads executed in the same SM. The overhead of binding the texture memory causes the slight performance drop. Using the texture cache for the source star model from the texture memory is even slower compared with automatic global memory caching. This shows the capability of the Fermi cache system and it makes some of the memory optimization techniques, that work well on the older architecture, ineffective.

The efficiency of the cache system depends on the number of data being reused by threads within the same warp as well as the size of the data. Since there are only a maximum 48KB of L1 cache per SM, there may not be enough memory to store all the data being reused. Therefore, there are still performance improvements by using texture memory on the GTX480 GPU in kernel 2. On the Fermi GPU, even though the caching is managed by hardware, it may not perform optimally for a particular algorithm. We found that using texture memory programmatically is more efficient in our case. When the data being reused is small and is being used by other threads in the same warp, the cache system on Fermi is really effective as we shown in kernel 4.

Reduce data transfer between Host and Device

The optimization strategy used in kernel 3 is to reduce data transfer between host and device by performing computations on the GPU as much as possible. In kernel 2, the computed amplifications are stored and accumulated at memory space in the register. The partial solution from each thread is written to the global memory and transfered back to the host memory. The CPU computes the final solution by summing the partial solutions from the same block. Although the computation of the final solution cannot be parallelized, it is still worth computing using the GPU instead of the CPU. The first thread in a block reads the partial solutions stored in the global memory computed by other threads within the same block and computes the final solution by summing the partial solutions. The final solutions are the amplification of the source integrations and are written to the device memory and transfered back to the host memory. Even though the memory read operation of the first thread is non-coalescing, it is still better to compute the final solution using the GPU in this case. For a 32 by 32 pixels source star model with 3000 source integrations, the data size is reduced by a factor

$$\frac{32 \times 32 \times 3000 \times sizeof(float)}{3000 \times sizeof(float)} = 1024\times$$

By computing the final solution using the GPU, we not only reduce the data size being transfered from the device memory back to the host memory, we also reduce CPU computation time. Note that a parallel algorithm for this reduction was attempted, but the performance was worse due to the extra overhead on issuing a new kernel call.

Workload distribution

The amount of computation required by a light curve is dependent on the number of source integrations and the size of the source star model. Since the source star model has the same pixel density as the magnification map, the resolution of the source model is dependent on both the source star size and the resolution of the magnification map. For a magnification map with both width and height of 1.0 in Einstein ring radii, and resolution of 4000 by 4000, the pixel width is $\frac{1.0}{4000} = 0.00025$. For a source radius 0.001, the width and height of the source star model is $0.001 \times 2/0.00025 = 8$. Therefore, this produces a source star model with resolution of 8 by 8 pixels.

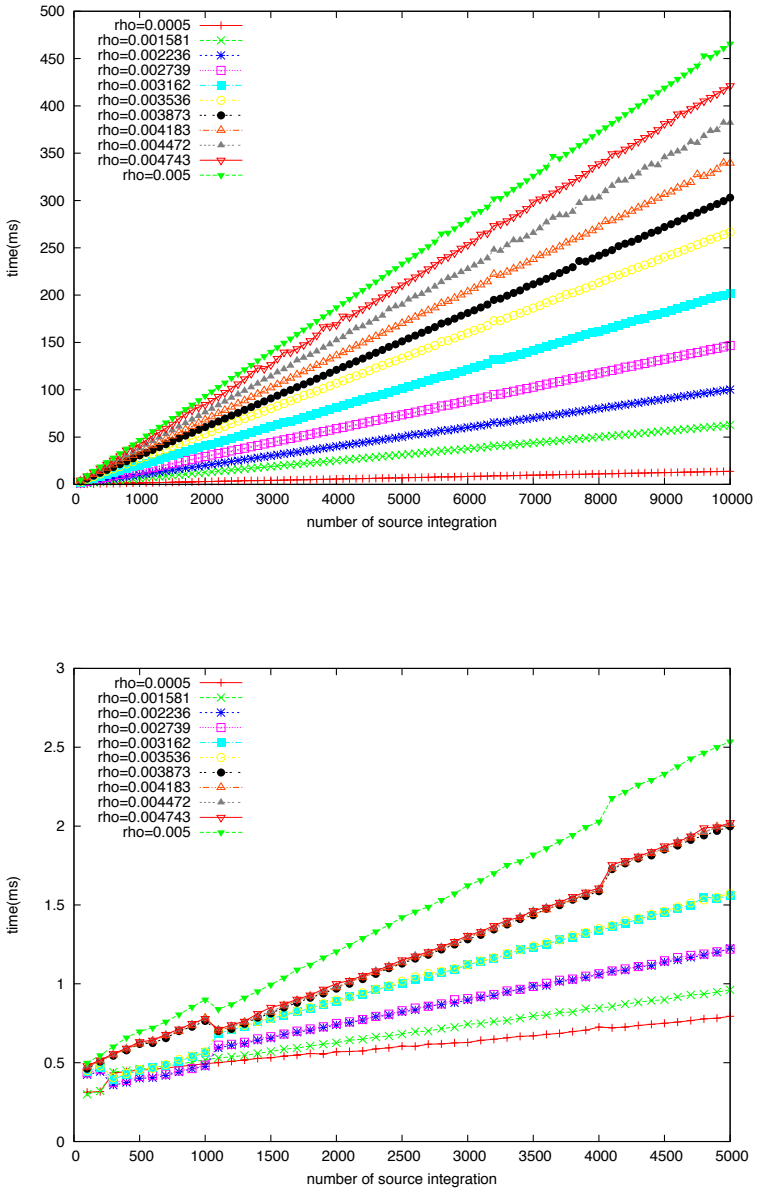

Figure 4.19: Top figure shows track extraction performance on a 2.66GHz Core i7 CPU turbo boosted to 2.93GHz. Bottom figure shows track extraction performance on a GTX480 GPU.

On the CPU, the computation workload gets heavier when the source star gets bigger. The computation time should be in a linear relationship to the number of source star integrations and the size of the source star model. Figure 4.19 (top) shows the computation time scales linearly with the number of source star integrations, and the computation time is double when the size of the source star model is double.

On the GPU, however, Figure 4.19 (bottom) shows the computation time does not increase linearly with the source star size. While the computation time of $\rho = 0.002236$ (with 18 by 18 pixels source star model) is double that of $\rho = 0.001581$ (with 12 by 12 pixels source star model), the computation time of $\rho = 0.002739$ (with 22 by 22 pixels source star model) is the same as $\rho = 0.002236$. Similar behaviours happen to computation with source star radius {0.003162, 0.003536}, {0.003873, 0.004183, 0.004472, 0.004743}.

The reason behind the non-linear increase in computation time with the source star model size is due to the number of loops performed on each thread. For a source radius 0.001 and source star model resolution of 8 by 8, every thread in a block processes one amplification calculation when a kernel is launched with an 8 by 8 block of threads. Since we launch a kernel with an 8 by 8 block of threads, when $\rho = 0.001581$ (with a 12 by 12 pixels source star model), each thread in a block needs to perform 2 loops to complete the computation. Even though some threads within the block do not receive jobs from the second loop, they still need to wait until the other threads that received job finish. Therefore, although a kernel call with $\rho = 0.002739$ (with a 22 by 22 pixels source star model) has double the workload of the kernel call with $\rho = 0.002236$ (with a 18 by 18 pixels source star model), they both require 3 loops to complete all the distributed jobs. So they take about the same computation time. The same principle can be applied to other different source star radius kernel calls. The source star integration time mainly depends on the number of loops in a thread required to complete all the jobs.

Occupancy

Figure 4.19 (bottom) shows a drop in performance for some kernel calls with ρ={0.003873, 0.004183, 0.004472, 0.004743, 0.005} at number of source integrations 1100 while displaying a gain in performance at 4100 source integrations. The kernel calls with ρ={0.002236, 0.002739, 0.003162, 0.003536} shows a drop in performance at 300 source integrations while gain in performance at 1100 source integrations. This fluctuation in performance is due to differences in occupancy when the number of blocks launched for a kernel call changes.

The occupancy represents the utilization of the GPU and depends on the combination of size of the kernel, size of block, number of blocks launched and number of MPs on the GPU. The size of the kernel is calculated by the amount of memory used, which includes the number of registers, and the sizes of shared memory and constant memory. Since each thread is distributed with its own register and shared memory space, the larger the kernel, the smaller the number of threads that can be managed by a MP. The number of blocks each MP is able to manage depends on the number of registers and shared memory used by the kernel. Therefore, the kernel size is kept small by reusing variables.

4.4 Conclusion

In this chapter, we have investigated the use of a GPU to perform microlensing modelling using magnification maps which include magnification map generation and track extraction for searching for the right model. The GPU outperforms the CPU by a large margin, more than a 100× increase in performance. This is comparable to the computation power of a small cluster computer. The very high performance of the GPU on microlensing modelling also allow us to model microlensing anomaly events in real time by using only a desktop computer with one or multiple GPUs.

We have discussed how to utilize the GPU to generate magnification maps as well as extracting light curves from the magnification map effectively. The GPU is ideal for repetitive tasks like ray shooting and integration. There are significant improvements in performance in both tasks by using the GPU. We have shown that a desktop computer with one or multiple GPU has enough power to out perform a cluster with 50 nodes or more.

In order to achieve more than 100× performance improvement over the CPU, we have to utilize techniques discussed in Chapter 3. This includes having coalescing global memory access, hiding memory latency, using texture cache to reduce global memory access, performing computation simultaneously on both CPU and GPU to reduce ideal time and reducing divergent branches in control flow statements. We also discussed the advantage of the caching ability on the Fermi GPU, reducing the data transfer between host and device as well as how to utilize the GPU even further by giving the right amount of workload.

The key aspect of using the power of the GPU efficiently is hiding the long latency operations as well as utilizing the hardware efficiently. This can be done by launching a large number of blocks so that each SM has enough warps ready to be selected while waiting for

the delay in long latency operations from the other warp. The selection of warp for execution does not introduce extra ideal time and is known as *zero-overhead thread scheduling*. Therefore, its operation is very effective in hiding long latency memory operations. The block size should also be multiple of warp size to effectively utilize all the SPs (cores) in each SM.

We have shown that the cache system on the Fermi GPU is very effective and sometimes performs even better than using texture memory. The cache system performs really well when the data being reused are small and are being reused in the warp level. However, it still is not as effective as when the data being reused is large. In most cases, the cache system on Fermi provide free performance improvement without using special software caching techniques. The cache system can be utilized effectively when the algorithm is being performed with an awareness of the caching limitation.

If there are divergent branches within a warp, the processing of all threads within that warp will be serialized. Thus, reducing the performance significantly when large numbers of warps have divergent branches. This is one limitation of the current GPU as most of the hardware resources are dedicated to floating point operations instead of managing divergent branching. But this is also the strength of the GPU as the floating point operations performance on the GPU highly outperforms the CPU. One should beware of this limitation and avoid large amounts of warps having divergent branches.

The performance provided by the GPU on microlensing modelling is therefore very promising for planet searches because we can perform modelling of anomalous events in real time with inexpensive hardware. The ability to perform real time modelling on a desktop computer, and not only be limited to powerful cluster computers, can increase the chance of planetary microlensing discovery as more real time information can be used for decision making in the distribution of telescope resources, and allow us to focus on important anomaly events.

Chapter 5

Dynamic Light Curve Engine for Complex Microlensing Modelling

5.1 Introduction

In the magnification map technique described in the previous chapter, we can generate a magnification map for a given set of physical lens parameters. We can quickly generate a large number of light curves for any given sets of parameters describing the source star profile and source star trajectories across the map using our GPU track extraction method. This modelling technique is great for grid searches to find initial models quickly. It works very well on a static lens system, but is not suitable if we wish to include dynamical effects such as orbital motion of the lenses in the system. Although the orbital motion effect does not change the mass fraction of the lens, it does change the lens objects positions. Therefore, a new magnification map is required for each amplification calculation.

In this chapter, we consider the problem of computing an amplification directly from the lens system and source star parameters. This involves solving the lens equation for the location of the images of the source star, and shooting rays from these positions. This procedure is carried out for each amplification calculation in the simulation, we called this light curve generation engine the "dynamic engine" for simulating amplification profiles. We generate

these profiles "on the fly" without the intermediate step of producing a two-dimensional magnification map. One can also think of this as "image-centred ray shooting"[71].

We are going to discuss implementation of a hybrid algorithm using both CPUs and GPUs to perform microlensing light curve computation for orbital motion. There are two major tasks with our light curve computation algorithm, image solving and ray shooting. We first discuss an efficient image solving algorithm that runs on the CPU. Then we show how to use the GPU for inverse ray shooting and limb-darkening computations. We will also compare the performance of different computation models. Finally, we discuss the optimization method used to model orbital motion effects.

5.2 Dynamic light curve engine

There are two major challenges in microlensing light curve calculations. First, the finite source calculation which usually consumes the most computation time due to the integration over the source star. We speed up the finite source calculation significantly by using magnification map techniques with GPUs as discussed in chapter 4. The magnification map techniques reuse the shot rays for multiple amplification calculations, and the ray shooting and integration can be done very quickly using the GPU. However, when there are other high-order effects like orbital motion involved in the calculation, magnification map techniques are a less attractive solution. This is because orbital motion affects the position of the lens star and its companion. Hence, the separation between the lens star and the companion, d, will change over the Einstein crossing time. Therefore, we cannot reuse the shot rays like in the magnification map technique as each amplification calculation requires a different value of d to be used in the lens equation.

The computation of the orbital motion effect together with finite source effects is usually the slowest and the most challenging task in microlensing modelling. In order to illustrate the challenge, we use the magnification map technique as an example. As discussed in the previous chapter, a magnification map essentially represents the parameter space of the mass fraction ε and projected separation d. A magnification map can be created very quickly using the GPU in seconds and also millions of light curves can be extracted using the GPU. Each light curve is made up by multiple amplification points in time and also represents the other set of parameter spaces like time of impact t_0, Einstein crossing time t_E, closest impact distance u_0, angle of track ϕ and the source star radius ρ. We are able to search

through the parameter spaces of t_0, t_E, u_0 and ϕ quickly by using one magnification map which represents ε and d.

To use magnification map techniques to compute light curves with orbital motion and finite source effects, each amplification calculation near the caustic will require a new magnification map since the result from the lens equation cannot be reused. This means we no longer are able to search through the parameter spaces t_0, t_E, u_0 and ϕ by using only one magnification map. Actually, we cannot even compute a single light curve by only using one magnification map as d is changing over time so each amplification point on the light curve requires different value of d to be used in the lens equation. Therefore, for a light curve made up of 1000 amplification points, twenty thousand light curves are computed (representing t_0, t_E, u_0 and ϕ) for each set of ε and d. There are 3000 discrete sets of ε and d that need to be searched in the modelling process and we need to generate $1000 \times 20000 \times 3000$ more magnification maps to complete the calculation compared with no orbital motion effect. Although each magnification map can be generated quicker as less rays need to be shot for each amplification point, it still takes months or years to compute. This is the very reason why orbital motion modelling is slow and challenging.

Various techniques [72, 73] are being used to speed up this computation, but the trade off in gains in speed is usually lower accuracry. In some microlensing events, like high magnification and caustic crossing events with orbital motion, high accuracy light curve calculations are required for successful modelling. These types of microlensing events are usually the most challenging to solve.

We have developed an efficient light curve calculation methods with high accuracy and high performance to model challenging microlensing events. It is a hybrid method involving both CPU and GPU to perform different calculations as shown in Section 5.1. The general idea is similar to the magnification map techniques which involve ray shooting to accumulate rays on the source plane. The difference with the magnification map techniques is that we are shooting rays to the much smaller source star instead of the entire source plane. For each amplification calculation, we first find out the image area on the lens plane that is the direct mapping to the source star area on the source plane, then shoot rays from the image area and collect rays that fall into the source. A more efficient image area solving method is also required as it is too slow to compute the image area for each amplification calculation by using the method used in the magnification map generation techniques.

The dynamic light curve engine consists of two major computational challenges - solving for the image area and inverse ray shooting. An efficient image area solving algorithm is

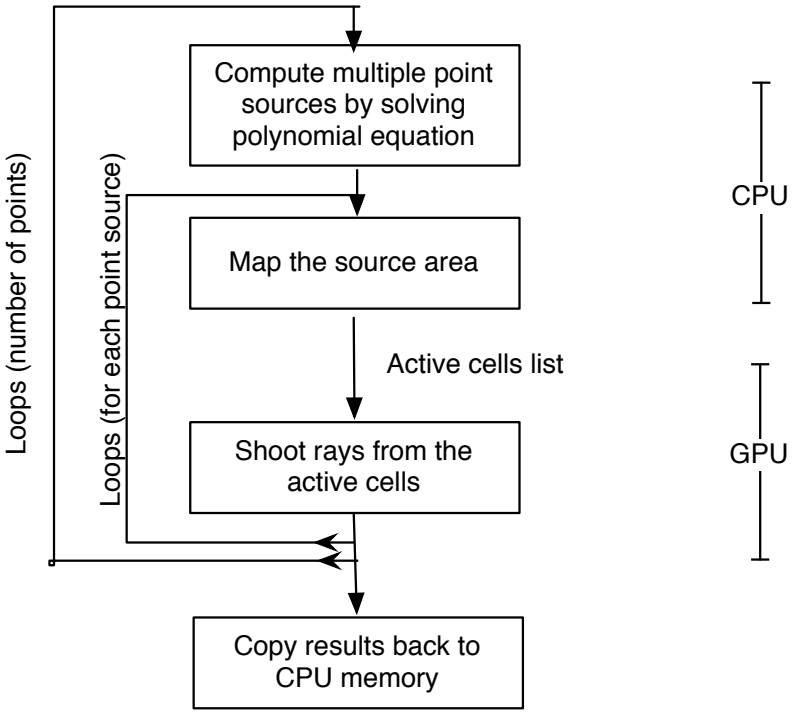

Figure 5.1: Dynamic light curve engine procedure

developed that runs on the CPU and the fast inverse ray shooting is performed on the GPU to provide a high performance light curve calculation engine. Before the calculation starts, the light curve is divided into multiple sections and different amplification calculation methods are used. While our hybrid amplification calculation method is used to compute part of the light curve that requires high accuracy and is computationally expensive, some parts of the light curve can be computed by other approximation methods to provide fast calculations with acceptable accuracy. The whole light curve is scaled accordingly at the end of the calculation.

5.2.1 Light curve partitioning

Since the projected separation used in the lens equation changes over time, each amplification point needs to be computed independently and the computed image area and rays shot from other amplification points cannot be reused. Therefore, each light curve is very expensive to compute. In order to speed up the computation, some parts of the light curve can be computed using faster methods like point-source approximation with acceptable accuracy.

Before the light curve calculation takes place, we need to separate the light curve into different sections that require different calculation methods.

Figure 5.4 shows light curves that are calculated by two different methods - the dynamic light curve calculation method and the point source approximation method. This microlensing light curve is made up by a binary lens and a finite source, the source star crosses the central caustic as shown in Figure 5.2. The point source approximation method computes a light curve with a point source instead of a finite source. Compared with the dynamic light curve calculation method which takes finite source effects into account, part of the light curve features are being washed out in the point source approximation method.

The point source binary lens light curve can be computed very quickly using the point source approximation, but there are two major problems so that this method alone is not sufficient in microlensing modelling. First, the source amplification is infinite over the caustic curve resulting in a discontinuous light curve. Second, the finite-source effect changes the shape or features of the light curve when the source star travels across the caustic.

When the source star is far away enough from the caustic, the point source approximation method is sufficient for amplification calculations as the finite-source effect mainly affects parts of the light curve for which the source star is close to or travels across the caustic. Figure 5.4 shows part of the light curve generated by the point source approximation method overlapping the dynamic light curve calculation method. So the point source approximation can be used when the source star is not too close to the caustic. In [71], the procedure switches to a point source model when the source is 6 source star radii away from the caustic. This threshold was found to be acceptable here.

Caustic curve approximation

In order to compute the distance of the source star to the caustic, the caustic curve, shown in Figure 5.2, needs to be computed. The binary lens caustic curve can be computed by just solving the 4th-order complex polynomial equation from the lens equation to obtain the critical curve. The caustic curve can then be computed by shooting rays from the location of the critical curve. The resolution depends on the number of caustic curve points being computed. For most caustics, it was found at least 72 points are needed to resolve the complete caustic, otherwise the shape of caustics cannot be resolved in enough detail. When the closest distance between the source star and the caustic is greater than 6 source radii, the point source approximation method can be used. Although the caustic curve can

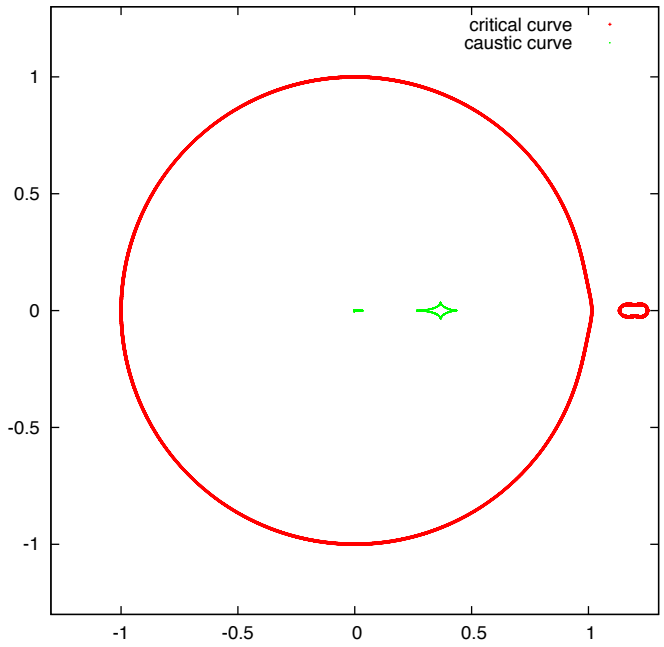

Figure 5.2: Critical curve and caustic curve. For binary lensing with planetary mass fraction 0.001 and projected separation of 1.2 Einstein radii between the components.

be computed relatively quickly, it takes a considerable amount of time when the the closest distance between the source and the caustic is computed for every amplification calculation. The time used to partition the light curve may outweigh the benefit. Therefore, a faster source to caustic distance calculation method is essential.

Caustics come in different sizes and shapes depending on the mass fraction, ε, and projected separation, d, used in the lens equation. For a binary lens, there can be either 1, 2 or 3 separated caustics depending on d. For each caustic, we search for the maximum and minimum values on both the x and y axes. All four values together can form a bounding box that is just enough to cover the caustic. Each caustic box is then enlarged to 6 source radii in each direction. Therefore, it is a lot quicker to partition the light curve as we are looking for the distance between the source and the caustic box instead of computing the distance to all 72 points on the caustic curve to find the closest distance.

For amplification points that lie outside the Einstein ring, point-source, point-lens amplification calculations are being used. This is because the caustic created by a multiple-lens system does not affect the amplification outside the Einstein ring. For calculations outside and far away from the caustic (more than 6 source radii), point-source approximation provides acceptable accuracy. For amplification calculations very close to or within the caustic,

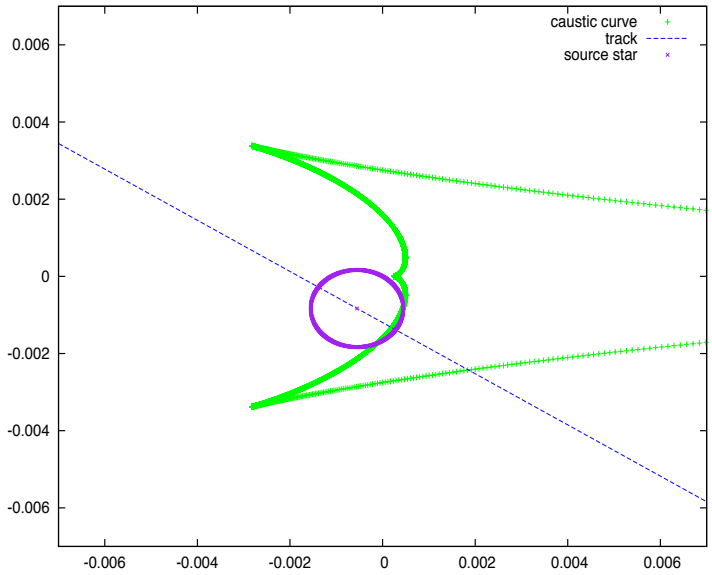

Figure 5.3: A source star and its trajectory as it crosses the central caustic. The lensing geometry is the same as in Figure 5.2.

the dynamic ray shooting methods are used.

Our finite-source amplification computation procedure involves three steps, the first and second steps are computed by the CPU and the last step is computed by the GPU. First, the image positions of multiple point source locations are found by solving the 5th-order complex polynomial equation. Second is the search for the image area which maps to the source area. The image area is computed by discovering neighbouring areas, which also map to the source, starting from the resolved image positions from the first step. The image plane is divided into grid cells. For those cells that are within the image area, we label them as active cells. Lastly, rays are shot from each active cell using GPU.

5.2.2 Efficient image solving algorithm

When computing light curve amplification with orbital motion effects, each amplification point on the light curve needs to be solved independently due to the projected separation d used in the lens equation changing over time and being different for each amplification point. This requires a lot more image solving calculations compared with only once in the magnification map techniques where the same image area is reused for thousands of light curves' computation. Therefore, an efficient image solving algorithm is crucial for high performance in dynamic light curve generation.

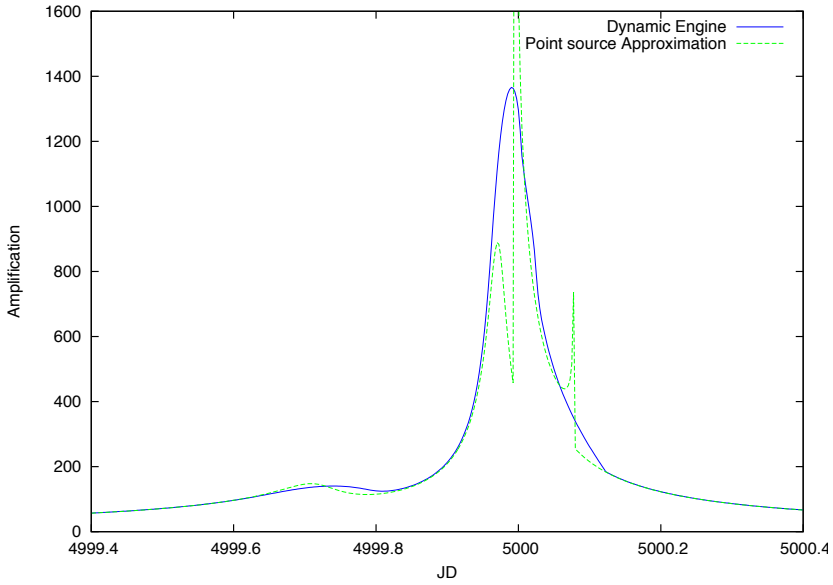

Figure 5.4: The light curve corresponding to the source star track shown in Figure 5.3. The solid blue line is generated by the dynamic engine with a finite source. The dotted green line is generated by the point source approximation.

The image area shown in Figure 5.5 is the direct mapping of the source star area using the lens equation. Rays shot from the image area on the lens plane will land inside the source star on the source plane. For each amplification point calculation, an equal density of rays will be shot from the image area on the lens plane. The number of collected rays inside the source star is the relative amplification of the source star at a particular time. At the end, all the amplification points on the light curve will be scaled accordingly. The image area of a finite source can only be solved numerically, as rays can only be shot from the lens plane to the source plane. Therefore, for every amplification calculation, the image area needs to be resolved before ray shooting starts.

The grid point image area solving methods [34] used in generating the magnification map in Chapter 4 are too slow to be used in the dynamic light curve generation. Although it only takes around 100ms for the magnification map generation procedure, it takes considerably longer in the dynamic light curve generation process. This is because the grid point resolution for searching the image area in dynamic light curve generation needs to be a lot higher to achieve good accuracy. Since the targeted mapping area on the source plane is a significantly smaller area (area of the source star) compared with the area of the whole magnification map, a much higher resolution grid is required to resolve the very thin arc-like

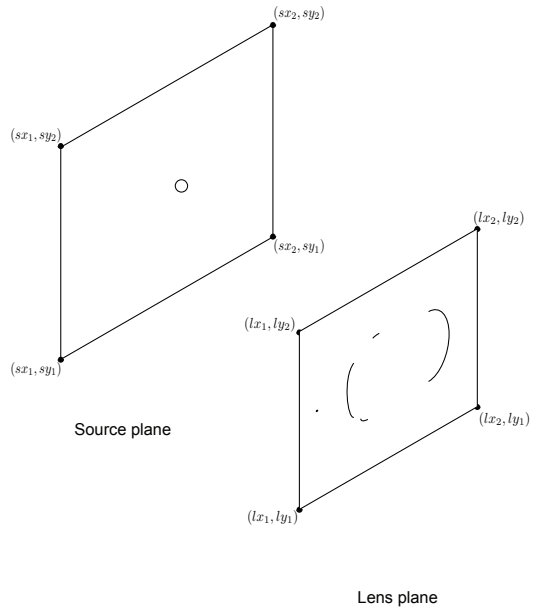

Figure 5.5: The image area (arc-like dark line on the lens plane) shown in the above figure is mapped from the source star (circle on the source plane). Rays that are shot from the image area by inverse ray shooting will land on the source star.

image area when the source is close to the caustic. If the grid resolution is too low, we may not be able to resolve most of the image area. The time taken to resolve the image area in the dynamic light curve engine will be minutes instead of milliseconds for each light curve. It becomes an impractical task when millions of light curves are being generated.

Since the CPU is used for solving the image area while the GPU uses the results from the CPU for ray shooting, the key to achieve high performance is an efficient image solving algorithm as well as lower overhead in preparation of data for the GPU. The image area solving algorithm should produce results that are ready for the GPU to consume rather than rearrange the data for every GPU computation. Therefore, our algorithm is designed to be both efficient and reduce the data rearrangement overhead. The results from the image solving algorithm are ready to be used in the next procedure, ray shooting by the GPU.

Solving image positions.

The image of a finite source is an area on the lens plane. The shape and number of disconnected image areas depends on the source star location relative to the caustic. We first consider a point source binary lens. There are either 3 or 5 point images depending on the

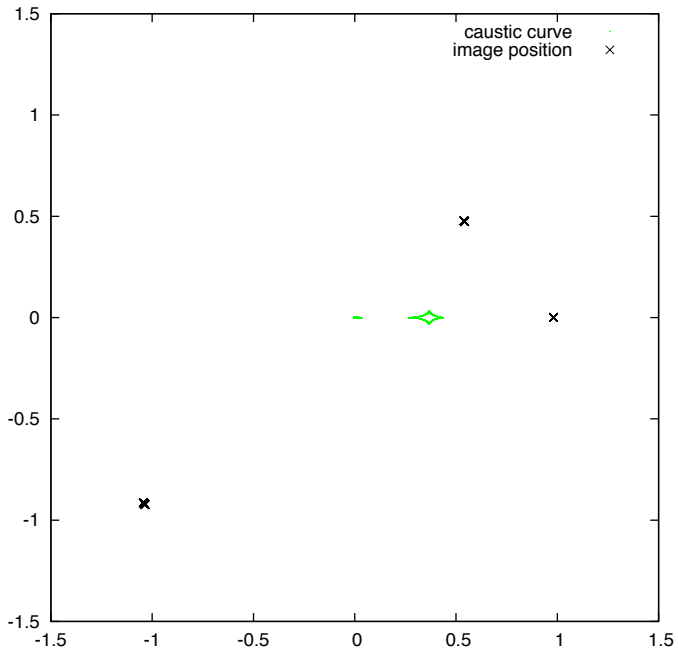

Figure 5.6: Image locations for a binary lens. The black crosses are the image positions obtained by solving the inverted binary lens equation. The green lines trace the corresponding caustic curve of such a lens system.

source position. The image positions of a binary point lens with point source can be obtained by solving the inverted binary lens equation, which is a 5th order complex polynomial equation. The polynomial roots can be solved by efficient numerical methods [37]. However, when the finite source effect is involved in the calculation, the images are no longer point locations but disjointed areas on the lens plane instead. The image area depends on the source star location on the source plane and the image becomes an arc-like ring as the source approaches the centre of the Einstein ring. Figure 5.6 shows the resolved image position of a point source with binary lens.

Given the source position located near the caustic, Figure 5.8 shows the image positions of the centre of the source and image area of the finite source. The image area is where rays are being shot and collected since the image area on the image plane is mapped to the source on the source plane. The principal idea is to first solve the point image positions of the centre of the source, then discover the neighbouring image positions that are also mapped to the source by the lens equation. Since the number of images depends on the source location, 3 images when the source is outside the caustic and 5 images when the source is inside the caustic, solving the image positions of the centre of the source is not

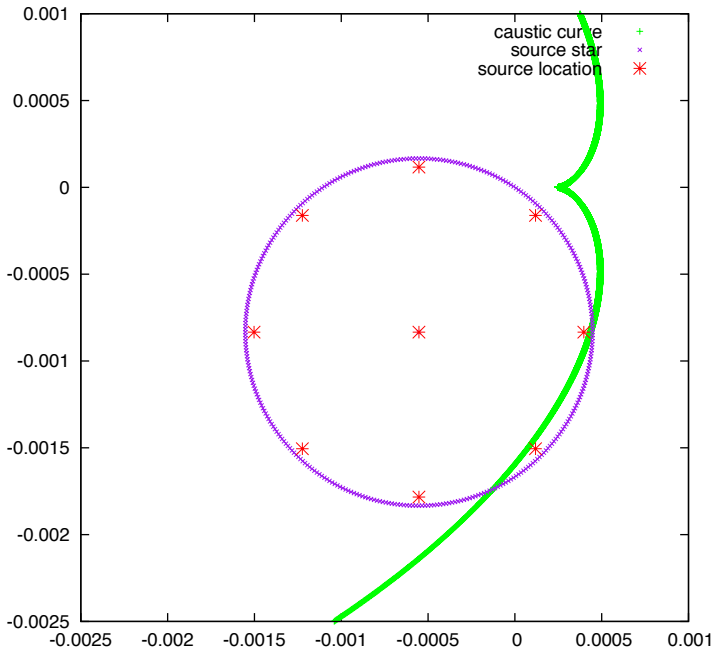

Figure 5.7: Source star lies at the edge of the caustic. In this case, it is not enough to only solve the image positions of the point-source location at the centre of the source star since a part of the source star lies within the caustic. The image positions are solved for the positions on the source star indicated by the asterisks on the figure.

enough when the source is located partially within the caustic. In our procedure, we are not just solving the image positions of the centre of the source, but also eight other locations around the edge of the source as shown in Figure 5.7. There are in total nine point-source locations to be resolved before we start finding the image area.

Searching image area

The image area can only be found numerically as it is a one-directional mapping by the lens equation. As discussed in Chapter 4, the image area of a magnification map can be resolved by using the grid points method. Since we are searching the image area of a much smaller area on the source plane (the area of the source star), the resolution of the grid on the lens plane needs to be much higher. Moreover, the performance becomes even slower when the image area of the source needs to be computed for each amplification calculation.

Instead of searching the image area by shooting rays on the whole lens plane, we start searching the image area for the point image positions that we computed from a point source

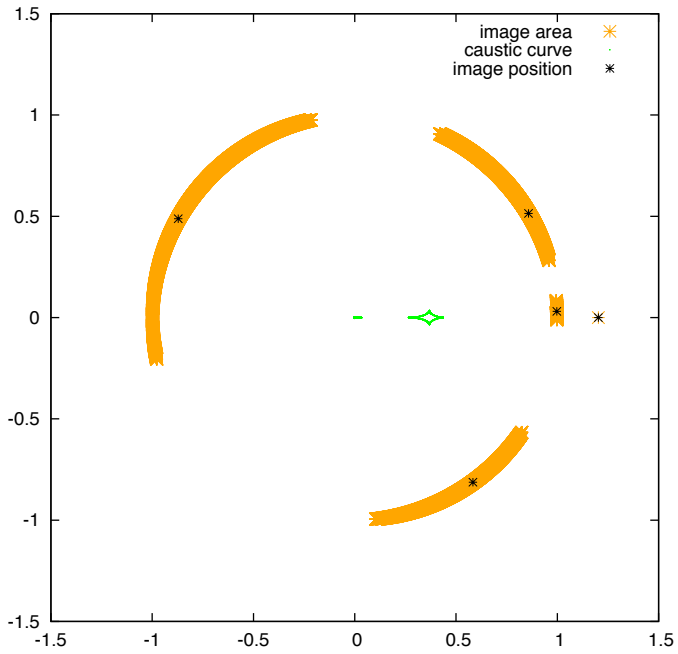

Figure 5.8: The black crosses are the point image positions obtained by solving the nine point-source locations shown in Figure 5.7. The area coloured in orange is the image area searched by the image area solving algorithm from the point image positions (black crosses). The image area is also the direct mapping of the source star from the lens equation.

as discussed the previous section. Once the image positions of the nine point-source locations are resolved, we start the second stage of our image solving algorithm. We divide the image plane into grid cells and the dimension depends on the source star size. The average image width is $\pi/4 \times \rho$, so the smaller the source star, the higher the grid resolution needed. We define the grid scale$= \pi/4 \times \rho \times 0.7$, which is 70% of the mean image width to ensure the resolution of the image grid is high enough to resolve the image area. Grid cells that contain the resolved point-like images are labeled as active cells.

Figure 5.9 shows a lens plane with five separated image areas. The lens plane is covered by a grid with sufficient resolution to resolve the image areas. The cross in the figure is the computed point image and cells that contain point images are labeled as active cells. The search of the image area is started from the initially labeled active cells using the grid point test method. The surrounding cells around the active cells will be tested and labeled as active if within the image area. The procedure is repeated until reaching the boundary of the image area. The image-solving algorithm can be described with the following recursive method.

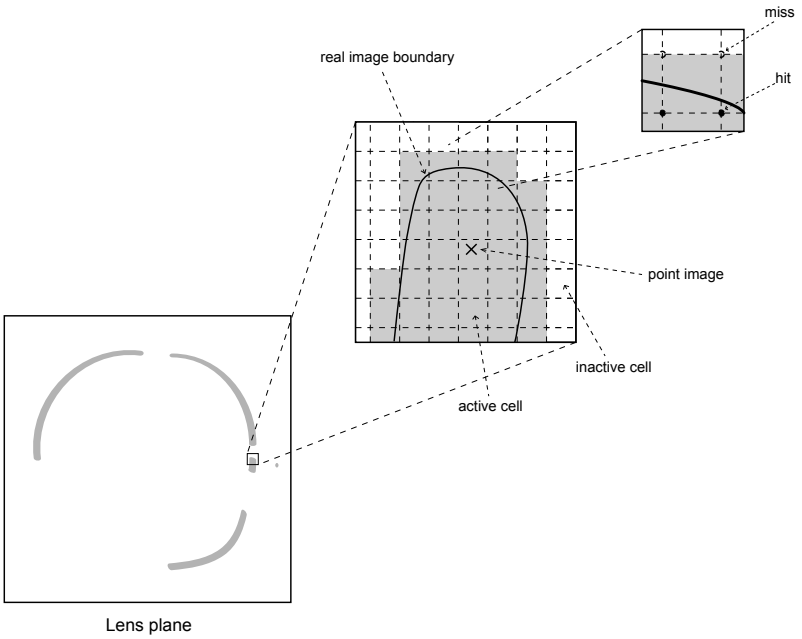

Figure 5.9: The above figure shows five disjointed image areas on the lens plane that map to a source star on the source plane. The lens plane is divided into a grid and the point images resolved from the point sources are marked as crosses. The cell that contains the point image is labeled as active. The image area searching algorithm searches the neighbouring cells of the active cells and checks if they are active.

```
void checkCels(cell) {
 if cell is active
   label as active
 for each neighbouring cell
    if cell is NOT active
      checkCell()
    else
      die
}
```

First, the cells that contain the point images will be labeled as active. The image area can be resolved by running this recursive method on each active cell that is labeled in the previous step. In each method call, the image area grows from the starting active cell in all directions until, at the boundary, non-active cells are reached. This recursive method

is simple to code and can be run in parallel as long as there is a mechanism for checking and labeling active cells that can be used by all running threads. Although it is convenient to implement a recursive version of our algorithm, the drawback of the recursive method is that it is slower in performance compared with the iterative version. It may also cause stack overflow as the number of active cells is unpredictable and depends on the grid resolution and lens configuration. More importantly, the output from the recursive algorithm needs to be rearranged before fetching to the GPU memory for processing. In order to utilize the GPU efficiently, the GPU code takes a list of active cells (coordinates of the grid cells) as input for maximizing data parallelism, similar to the magnification map engine discussed in Chapter 4. Therefore, extra overhead is required to create a list of active cells by looping every grid cell on the lens plane for the GPU.

5.2.3 Efficient image area searching method

To address the drawbacks and performance issues of the recursive version of our algorithm, we developed an iterative version which is more efficient and can be better integrated into our GPU ray shooting code. It is designed with the GPU procedure in mind, as it can produce the active cells list that is required by the GPU as an input at no extra cost.

Custom queue data structure

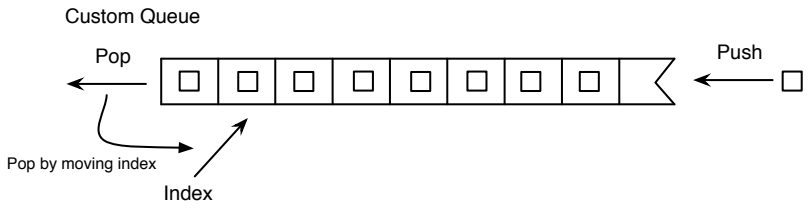

Figure 5.10: Custom queue data structure for storing the active cells array. These are ready to be used by the GPU without rearrangement.

We designed a custom queue data structure for use as the core model for our data processing procedure as well as the storage of the results of our algorithm. At the end, the results in the queue data structure are ready to be copied directly to the GPU memory without any extra processing. Figure 5.10 shows the custom queue data structure used in our algorithm. The queue data structure is made up by a flexible array that will dynamically

enlarge itself when needed. Each memory space in the array is pointed to a C struct which contains two variables for storing coordinate information. The coordinates of active cells are stored by the two variables when an active cell is being pushed into the queue. When data is being popped from the queue, an index is moved and pointed to the next memory space instead of deleting the content of the popped data. Therefore, all the pushed active cells' coordinates will be stored in the array and are ready to be used by the GPU without extra rearrangement that cause overhead.

Active cells labeling and overlap checking

Another central component of our image area searching algorithm is the mechanism for checking already labeled active cells to avoid over counting. Since each active cell should only be used once for ray shooting, it is important to avoid duplicated active cells being fetched to the GPU. An efficient method is crucial to give high performance for our algorithm as every grid cell needs to be checked before being labeled as active and pushed into the queue. Moreover, a reliable and robust mechanism is essential for the image area searching algorithm to work probably. We have designed two mechanisms for this purpose.

The first mechanism uses a large piece of memory space for labeling active cells. Memory space for an array of 8-bits *char* is allocated before the image searching procedure starts. Each grid cell is represented by a memory space and marked as 1 if it is an active cell, otherwise 0. The size of the array depends on the resolution and size of the grid. The resolution depends on the source star size and the size of the grid is predefined. The array usually takes hundreds of megabytes to several gigabytes of memory space.

This mechanism provides very fast active cell checking and labeling to avoid over counting. The trade off is a large piece of memory space is required as it trades off memory space for computing speed. It may seem highly inefficient to allocate such a large piece of memory space. However, on some platforms/distubutions, like BSD Unix, Ubuntu and Max OSX, the method call *calloc()* allocates memory space dynamically and only allocates extra space when needed. Therefore the usage of memory space only grows as the algorithm utilizes more memory space when more active cells are being discovered.

Image area searching algorithm

Once the image locations of the nine point-source locations are resolved, the coordinates of the grid cells that contain the images are pushed into the custom queue data structure.

Those grid cells are also labeled as active cells using the labeling mechanism. The queue data structure serves a dual purpose, tracking which active cells are being processed and acting as a storage space for active cells' information that will be consumed by the GPU ray shooting code.

The following code shows the active cells searching algorithm. Once the procedure is started, the main iterative loop continues working until the queue is empty. First, an active cell is popped from the queue. The neighbouring cells of the first active cells are being checked using the grid point method. If the cell being processing is within the image area and is not active, that cell is pushed to the queue and is labeled active. The neighbouring cells are tested in the order of left to right, top to bottom. Once the procedure is done, the second active cell is popped from the queue and being processed with the same procedure. The whole procedure continues until the queue is empty.

```
while (queueIsNotEmpty) {
  xL, yL = popQueue()
  for each neighbouring cell {
    if (checkIfCellisActive(thisCell)) {
      pushQueue(axL, ayL)
      labelThisCellActive(thisCell)
    }
  }
}
```

In Figure 5.11, the lens plane is divided in a grid with resolution according to the source star size. The dark solid line represents the boundary of the real image area. The cross marks are the resolved point images from the point sources within the source star. The cells that contain the images are marked as active (grey in colour) and pushed into the queue. The top figure shows content of the queue before the main loop starts. The coordinates of the first labeled active cells are pushed into the queue and act as initial active cells in searching the image area. In loop 1, cell A1 is popped and its surrounding cells are tested using the grid point method. Since cells B1 to B8 are also within the image area boundary, they are being labeled as active and are pushed into the queue. The whole procedure is repeated until the queue is empty, thus the whole image area is covered by active cells. Note that only the index is moved when a cell was popped from the queue. Therefore, all the active cells can be stored inside the queue and are ready for copying into the GPU memory for ray shooting.

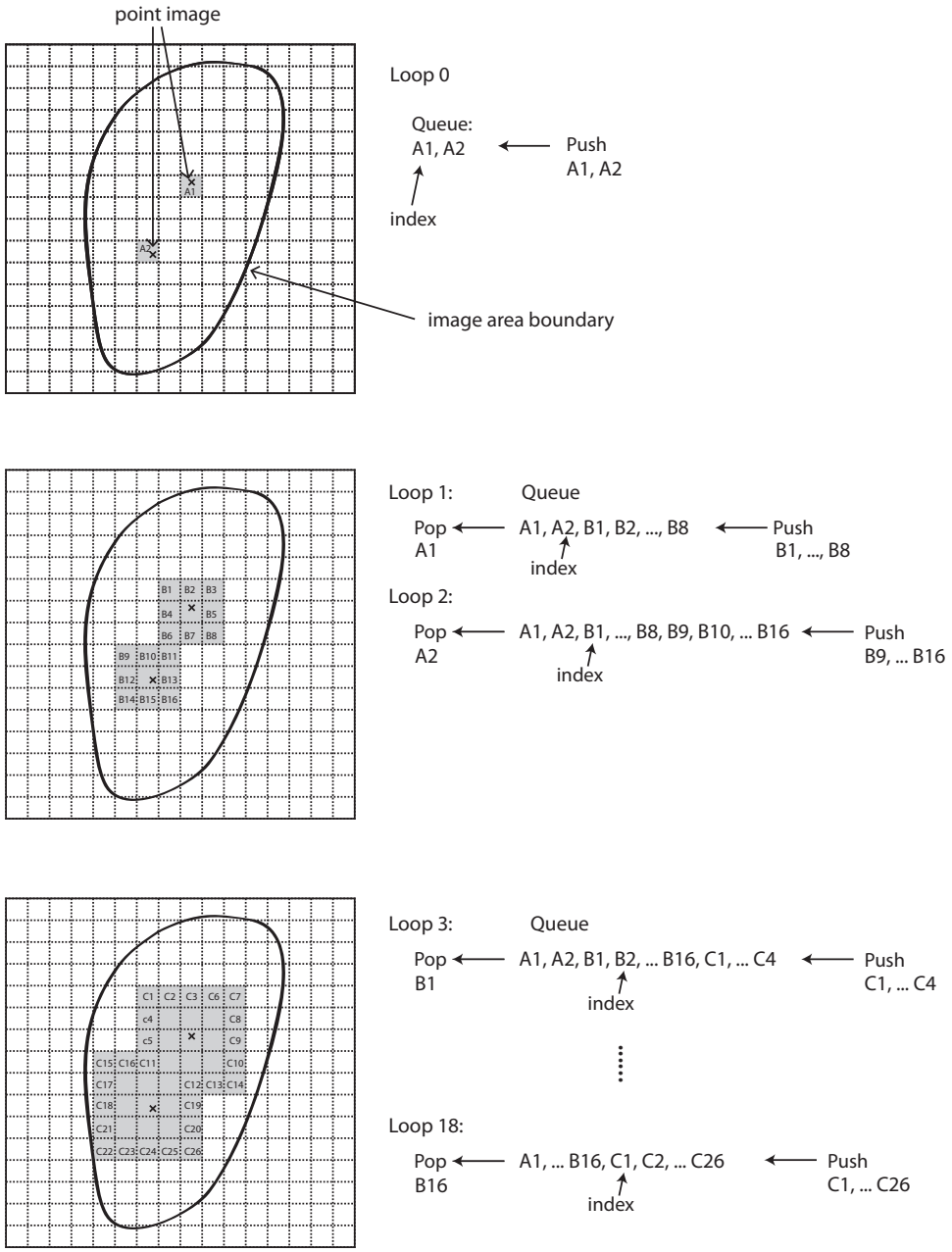

Figure 5.11: The image area searching algorithm at work at different stages. The point images are marked as black crosses and labeled active cells are in grey. The dark solid line is the real image area boundary.

The iterative version of our algorithm works similarly to the recursive version. The difference is that a custom queue is used to manage the active cells instead of a stack of method calls in the recursive procedure. This not only allows us to have total control in memory usage and performance tuning, but also provides an active cells list that is ready for the GPU to process with no extra data rearrangement. It is achieved by our custom design queue data structure which acts as an active cell manager as well as an active cells storage structure. The custom queue does not delete the contents of the queue when popping data, instead, it uses an index as pointer to keep track of the head of the queue. The content in the queue is ready to transfer to the GPU once the image area searching algorithm is completed as it is essentially an array with active cells' coordinates.

Our image area searching algorithm is designed to be very efficient in searching the image area. It only expends its search to the place where the image area is extended from the initial active cell. This is especially important when computing light curves of high magnification events as the image area is a very long thin arc when the source star is located close to the centre of the Einstein ring. Our image area searching algorithm does not spend extra computation on areas that are not mapped to the source.

5.2.4 GPU inverse ray shooting

Once the searching of the image area located at the lens plane is completed, rays are shot from the image area and collected on the source plane. Most of the rays will fall into the source star area as the image area is the mapping of the source by the lens equation. If the ray falls into the source star area, it will be weighted by the limb darkening equation and added to the source star bin. The ray shooting is computationally expensive as lots of rays need to be shot to achieve good accuracy for the amplification calculation. In this section, we describe how to use the GPU to speedup the ray shooting process in the dynamic light curve engine.

In the previous step, the image area is discovered and divided into a small grid and the coordinates of each active cell are stored in an array (the custom queue data structure). Similarly to the inverse ray shooting in magnification map generation, rays are shot from each active cell using the lens equation,

$$\omega = z - \sum_j \frac{\epsilon_j}{\overline{z} - \overline{z}_{m,j}}$$

However, for each active cell, a pre-defined number of rays are shot in a grid pattern instead of from random positions within the cell. Rays that land within the source area will be weighted by the limb darkening equation

$$S = 1 - \lambda_1(1 - \cos(\alpha)) - \lambda_2(1 - \sqrt{(\cos(\alpha))})$$

and the result will be added to the source bin instead of binning into pixels like in the magnification map generation.

The custom queue data structure stores the coordinates of the active cells ready to be copied to the GPU memory. The active cells' coordinates are copied from the CPU memory to the GPU global memory once the image area solving algorithm is completed. The ray shooting algorithm is similar to the one used in generating the magnification maps. But there are far less active cells compared with magnification map generation as the total image area is significantly smaller. Therefore, the ray shooting job needs to be distributed to as many threads as possible to utilize the GPU efficiently.

5.2.5 GPU ray shooting kernel

Since we are computing the amplification of a point in time, there is only one single bin to collect rays. All rays shot from the active cells are collected in one bin. In order to avoid memory race conditions, rays shot by each thread need to be stored in the local memory and added together at the end. There is the trade off between level of parallelism versus time spent on summing all the rays. At the extreme end, if all ray shooting jobs are distributed to achieve maximum parallelism, each thread only shoots a single ray, weights by the limb darkening and the weighted result is written back to the global memory in this case. There is extra processing overhead as the results in the global memory need to be added together. At the other extreme, if all rays are shot by only a few threads, the final result can be added together quickly, but the GPU is being highly under utilized. Therefore, a balance needs to be made to achieve maximum performance from the GPU.

The grid resolution depends on the source star size as discussed in section 5.2.2. The size of the grid is also changed according to the source star size. In order to shoot rays in equal density, we define the number of rays shot from each active cell as the *ray density in Einstein units*. This allows us to shoot rays in equal density from the image area independent of the size and resolution of the grid.

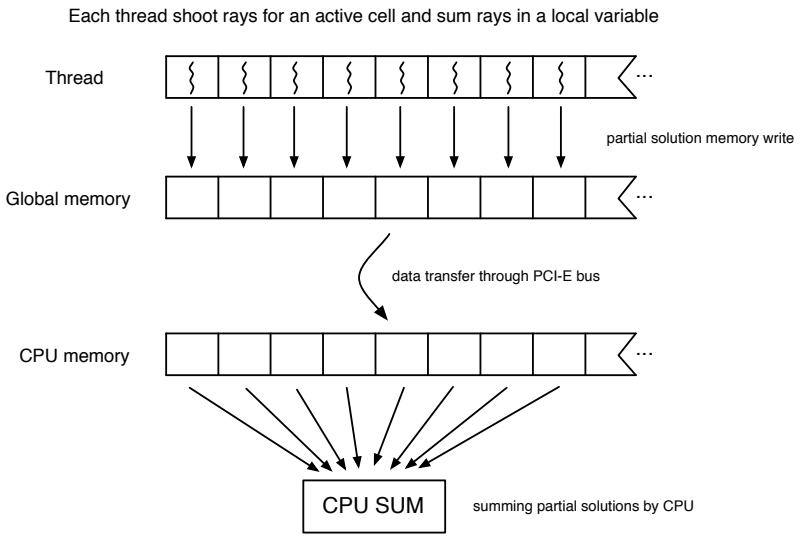

Figure 5.12: Ray Shooting by GPU

The array of active cells list is copied to the GPU global memory before launching the GPU kernel. Each active cell's coordinate is read by a thread and each thread shoots multiple rays according to the defined ray density. Each thread stores and accumulates the rays in its own local variable (register). The partial amplification from each thread will be copied to the CPU memory and be added together at the end. The advantage of this thread organization is that the coordinates of the active cells are read in coalescing fashion. Since each thread executes the same operations and has the same amount of workload, the writing operations to the global memory at the end is also in coalescing fashion.

5.3 Dynamic light curve engine models

In the previous section, we described in detail the procedure of computing a light curve using the dynamic light curve engine. A light curve is made up by multiple source star amplifications in time. The light curve can be partitioned into different sections and each section can be computed using the appropriate methods to achieve maximum performance. For an amplification calculation using the hybrid CPU and GPU method, the image areas that are mapped to the source star need to be first resolved, then rays are shot from the image areas and are collected in the source star bin.

At this point, the inverse ray shooting by the GPU is the only part using parallel computation in the dynamic light curve engine. There is room to increase the parallelism in

computation since most parts of the light curve calculation are an embarrassingly parallel problem. For every light curve calculation, multiple amplifications need to be computed which means multiple image areas need to be solved. The image area searching computation also can be parallelized as well as ray shooting. The optimum and most efficient way to parallelize the light curve computation problem is the key to achieve the best performance. We are going to investigate different parallel computation models for the light curve calculation.

5.3.1 Light Curve Engine 1 - Sequential CPU and GPU computation

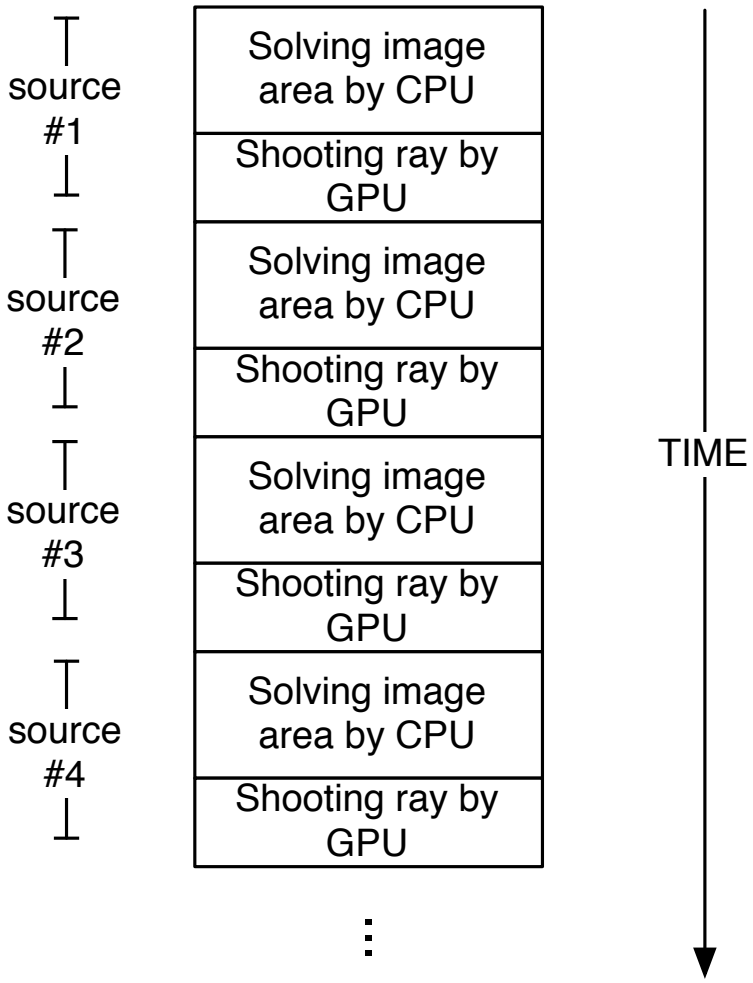

Figure 5.13: Light Curve Engine 1

In light curve engine 1, the CPU and GPU work sequentially. For each amplification calculation, the CPU first computes the image area. Then the image area in the form of an active cells list is transferred to the GPU for ray shooting. The GPU wait until the CPU's computation is completed to get the result from the CPU. The CPU also wait for the GPU to complete ray shooting before starting to compute the image area for the next source amplification calculation.

This model is simple and easy to implement but both the CPU and GPU have lots of idle time as they need to wait for each other to complete their computation. Figure 5.13 shows how the CPU and GPU work together in sequential order for image area searching and ray shooting, respectively.

CPU versus GPU ray shooting

Figure 5.14 shows the performance comparison between light curve generation performance for the CPU-only methods (CPU compute both image area and ray shooting) and the hybrid CPU and GPU method (image area computed by CPU and ray shooting by GPU).

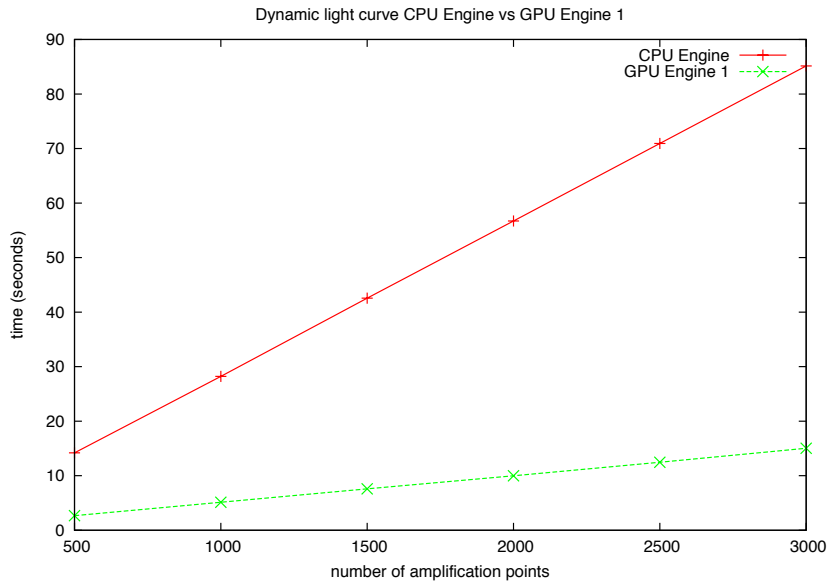

Figure 5.14: Light curve engine performance CPU vs. GPU Engine 1

The GPU rays shooting performance has an improvement over the CPU ranging from 23× (500 amplification points calculation) to 34× (3000 amplification points calculation). The computation was done on an Intel Core i7 3.0 GHz CPU with an NVIDIA GTX480 GPU.

However, when combining the performance of both image area solving and ray shooting computation, the performance improvement is around 5× compared with a CPU-only light curve generation engine. The modest performance improvement over the CPU-only light curve engine is due to the majority of computation time now being spent on image area solving done by the CPU.

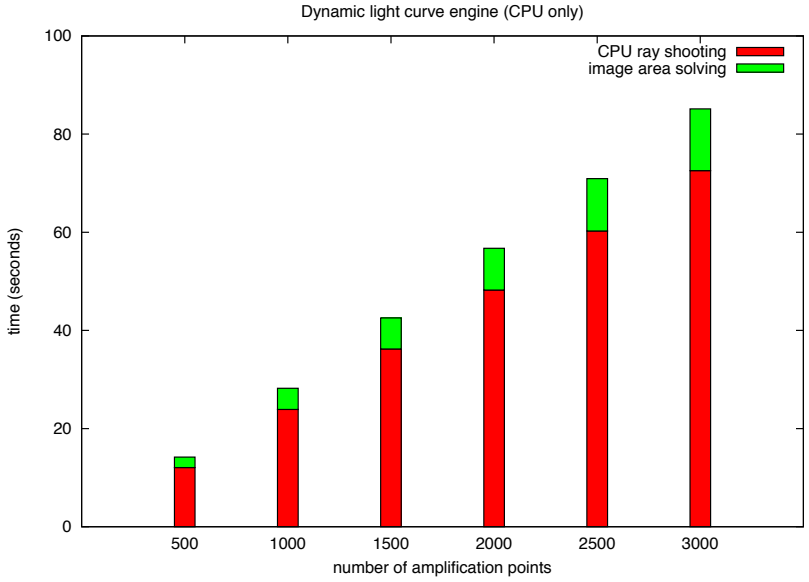

Figure 5.15: CPU-only light curve engine performance. The CPU is an Intel Core i7 3.0GHz CPU running in turbo mode.

Figure 5.15 shows computation time of image area solving and ray shooting by the CPU-only light curve generation engine. In a light curve calculation, 85% of the computation time is spent on ray shooting while the image area solving algorithm uses only 15% of total computation time. Therefore, speeding up the ray shooting by the GPU is going to improve the light curve engine performance significantly.

In dynamic light curve engine 1, we significantly improve the performance by performing ray shooting on the GPU. Figure 5.16 shows the ray shooting time using the GPU only represents 14% to 19% of total light curve computation time. Table 5.1 shows the performance improvement in GPU ray shooting over the CPU version. It was performed on an Intel Core i7 3.0GHz CPU in turbo boost mode (single thread) and an NVIDIA GTX480 GPU. Every performance test was run 10 times and the average is recorded with an initial run being discarded.

The speed up by the GPU shown in Table 5.1 is not linear, in fact, more work results in

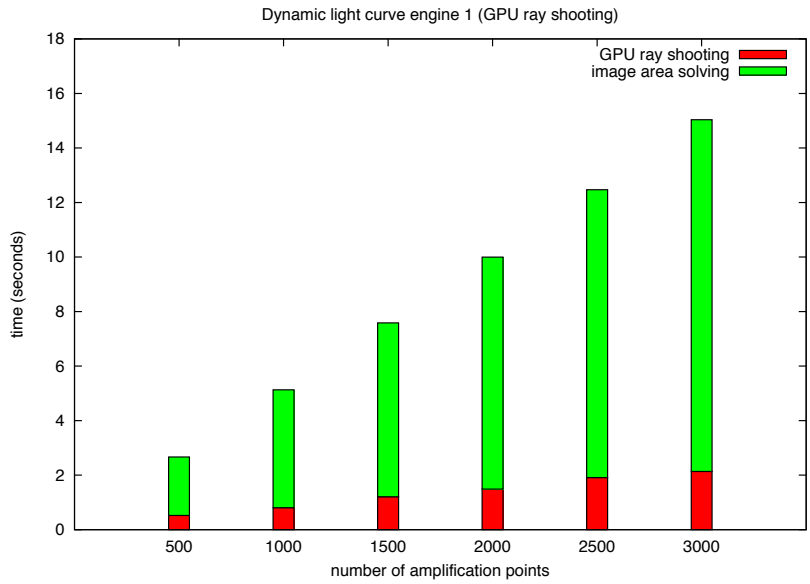

Figure 5.16: Light curve GPU Engine 1 performance. CPU is an Intel Core i7 3.0GHz CPU running in turbo mode and GPU is an NVIDIA GTX480.

Number of amplification points	CPU ray shooting	GPU ray shooting	Speedup
500	12.05s	0.52s	23.17×
1000	23.90s	0.81s	29.51×
1500	36.21s	1.21s	29.93×
2000	48.24s	1.49s	32.38×
2500	60.27s	1.91s	31.56×
3000	72.56s	2.13s	34.07×

Table 5.1: GPU rays shooting performance. CPU is an Intel Core i7 3.0GHz CPU running in turbo mode and GPU is an NVIDIA GTX480.

better speedup performance. This is because workload to GPU overhead ratio is increased when more amplification points are being computed. The workload of every amplification point calculation is different, the higher the amplification, the larger the image area, thus more rays are shot. The workload to overhead ratio increases with more amplification point calculations because more high amplification points are being computed. The higher amplification computations mean that more rays are shot, and better utilization of the GPU is achieved.

5.3.2 Light Curve Engine 2 - Overlapping CPU and GPU computation

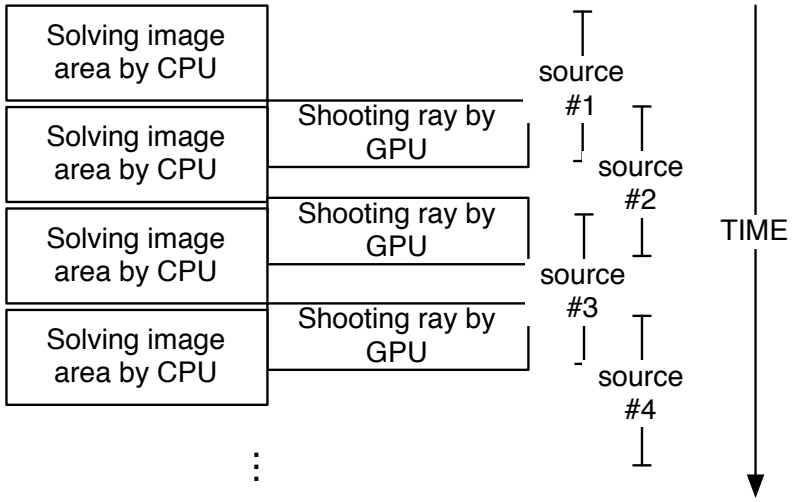

Figure 5.17: Light Curve Engine 2

In light curve engine 1, the GPU shoots rays using the active cells' coordinates computed by the image solving algorithm using the CPU. The procedure is repeated for every source location. The GPU is idle while waiting for the CPU to compute the image area and the CPU is also waiting for the GPU to complete the ray shooting. Since the GPU passes the control back to the host (CPU) once the kernel is launched, we can take advantage of this feature to reduce the idle time of both CPU and GPU.

Light curve engine 2 uses both CPU and GPU for computation simultaneously by overlapping their computation. After the image area of the first source location is computed, the CPU starts to compute the image area of the next source location while the GPU is shooting rays for the previous source location. The CPU passes the active cell coordinates of the second source location to the GPU once the image solving is completed or the GPU is ready to receive jobs again. Figure 5.19 shows the comparison between sequential CPU to GPU computation versus parallel CPU and GPU computation.

This approach reduces the idle cycles for both the CPU and the GPU and achieves higher parallelism in computation. The improvement in performance over the light curve engine 1 depends on the workload of the ray shooting kernel. Figure 5.18 shows the performance improvement of light curve engine 2 over light curve engine 1. The performance improvement

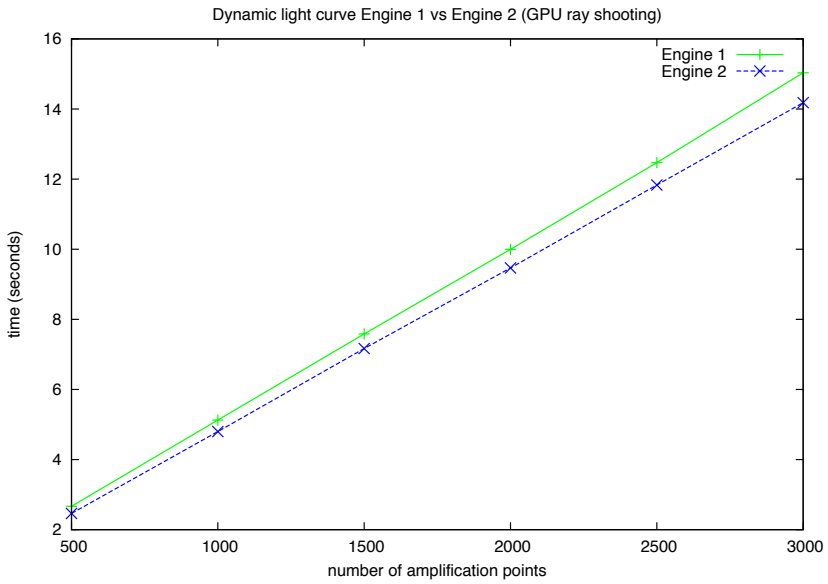

Figure 5.18: Light curve engine performance GPU Engine 1 vs. GPU Engine 2. CPU is an Intel Core i7 3.0GHz CPU running in turbo mode and GPU is an NVIDIA GTX480. Each experiment is performed six times and the average time is calculated using the last five runs. The uncertainties in each measurement are of the order $\sim$ 10ms. The error bars are smaller than the plot symbols.

is around 6% to 9% in total light curve generation computation time. In the previous section, we showed that the GPU ray shooting only accounted for 14% to 19% of the total light curve computation time. Therefore, there is actually a 50% performance improvement in the total GPU computation time. The GPU in light curve engine 2 is being utilized more efficiently with much less idle time.

5.3.3 Light Curve Engine 3 - Multi-threading image area search + GPU ray shooting kernel 1

Although light curve engine 2 achieves better performance over light curve engine 1 by overlapping the CPU and GPU computation, the image area searching computation by the CPU is still a significant part in the overall computation. In light curve engine 2, the idle time of the CPU and GPU is reduced to achieve better parallelism, but the image area searching computation only utilizes a single CPU core. We can achieve even better parallelism and improve light curve generation performance by utilizing multiple CPU cores for computation in solving the image area.

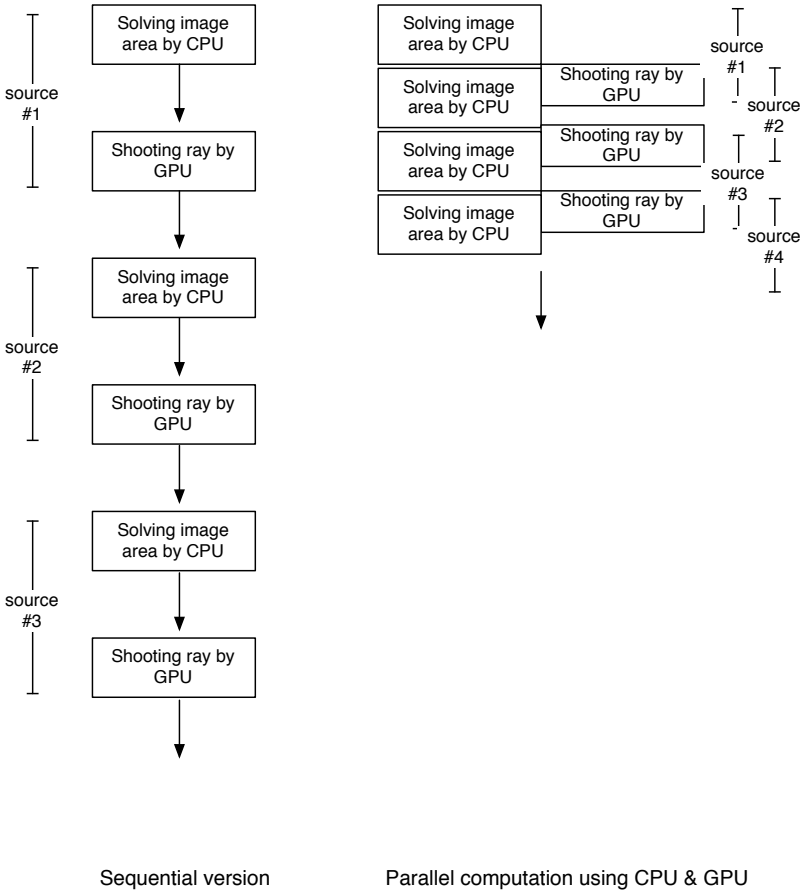

Figure 5.19: Sequential vs. Parallel computation using CPU and GPU

The nature of the image area searching algorithm allows us to easily transform it into a multi-threading algorithm. Each active cell popped from the queue can be processed independently and distributed to multiple threads. But a memory-locking mechanism has to be in place for the active cells checking grid as it is shared with multiple threads. This may cause significant delay and only obtain moderate performance improvement.

A better way to utilize multiple CPU cores is to distribute the multiple image area solving problems amongst these cores. Since there are usually hundreds or thousands of amplification calculations in light curve generation, it is natural to distribute workload at a higher level instead of at the image area solving algorithm level. This allows multiple CPU cores to be more efficiently utilized as it avoids significant shared memory locking delay and thread creation overhead.

Figure 5.20 shows multiple CPU cores used for solving the image area. Each thread is

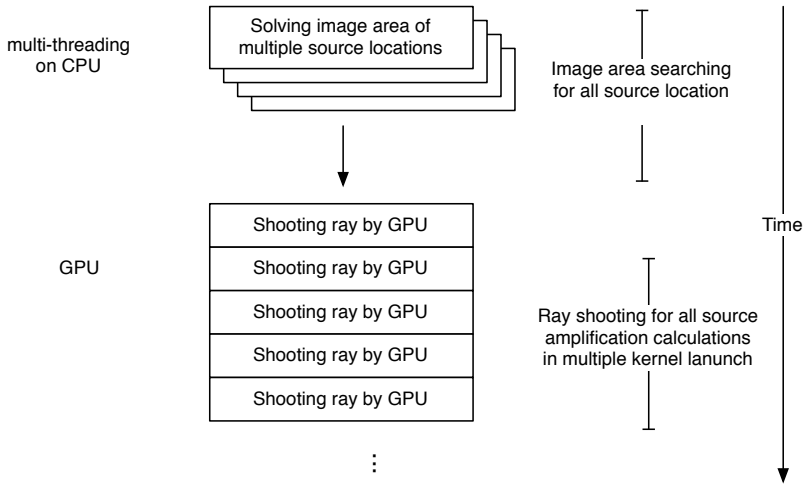

Figure 5.20: Light curve engine 3. Solving image area using multiple CPU threads and performing ray shooting using GPU ray shooting kernel 1.

assigned multiple jobs. The result of each job is stored in the CPU memory, then the GPU performs ray shooting sequentially for each source amplification calculation once all the image area solving computation is completed by the CPU. Since workload of each amplification calculation is different, a good load balancing is important to achieve good performance. We assign amplification calculation jobs to each thread sequentially from the beginning of the light curve and repeat the procedure until all jobs are assigned.

The performance test was run on the Quad core Intel Core i7 3.0GHz. Since it is capable of hyper threading, we start 8 threads for each logical core to achieve optimum performance. Figure 5.21 shows the performance of light curve engine 3 compared with light curve engines 1 and 2. There is significant performance improvement ranging from 42% to 48% in light curve engine 3. The multi-thread image area solving is about twice as fast compared with the single thread version. As it is a quad cores CPU, the ideal performance improvement should be four times. However, the image area solving performance only has about two times improvement in performance. This is because the CPU is running in turbo mode and increases its running frequency from 3.0GHz to 3.6GHz when running a single thread application, resulting in a faster single thread application performance. There are also overheads in workload distribution and result organization.

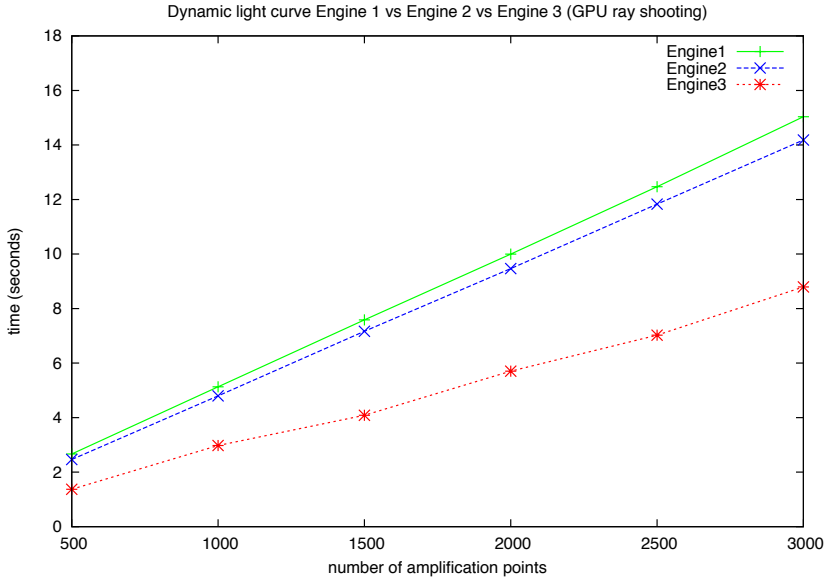

Figure 5.21: Light curve engine performance GPU Engine 1 vs. GPU Engine 2 vs. GPU Engine 3. CPU is a Quad core Intel Core i7 3.0GHz CPU and GPU is an NVIDIA GTX480. The error bars are smaller than the plot symbols.

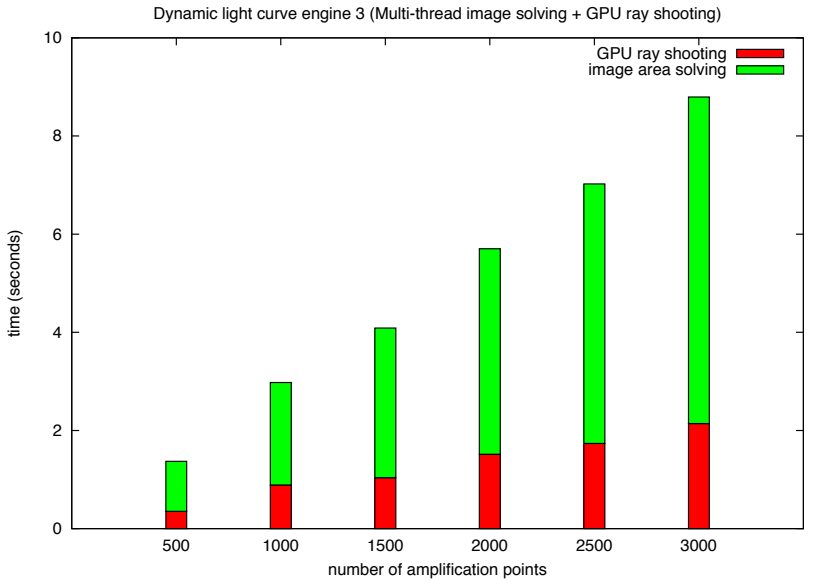

Figure 5.22: Light curve GPU Engine 2 performance. CPU is an Intel Core i7 3.0GHz CPU running in turbo mode and GPU is an NVIDIA GTX480. The error bars are smaller than the plot symbols.

Figure 5.22 shows the overall image area solving computation time consists of around 75% in total light curve generation time, compared with around 85% in light curve engine 1.

5.3.4 Light Curve Engine 4 - Multi-threading image area search + GPU ray shooting kernel 2

Although the light curve engine 3 discussed in the previous section utilizes multiple CPU cores by distributing image area solving computation to multiple threads, the GPU is actually under utilized due to each amplification computation having different workloads for the GPU to process. The total image area increases as the source star approaches the centre of the Einstein ring, as do the number of active cells being discovered, thus more workload is produced for the GPU. On the other hand, ray shooting workload for the GPU is very low with amplification calculation of a source star farther away from the centre of the Einstein ring. This can cause the GPU to be under utilized.

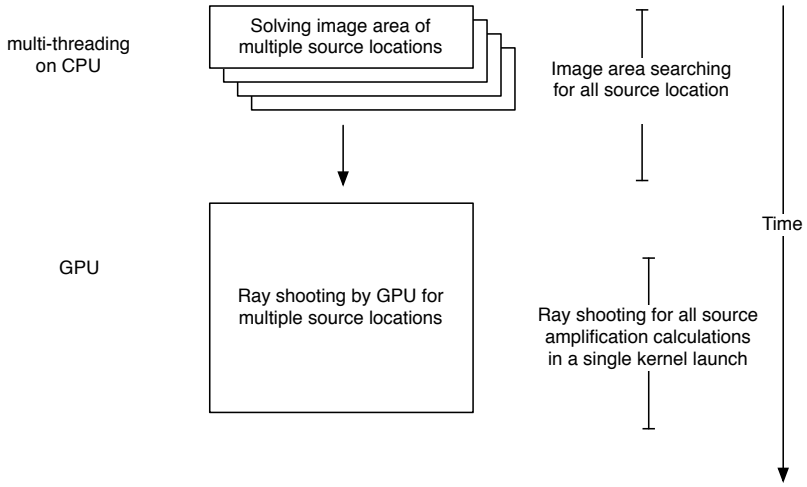

Figure 5.23: Light curve engine 4

In order to maximize the utilization of the GPU, we perform the ray shooting of multiple amplification calculations in a single GPU kernel launch. In light curve engine 4, image area (in the form of active cells list) of all the amplification calculations is transferred to the GPU global memory in a single memory copy operation. A single kernel call is used to perform all the ray shooting calculations for every source amplification calculation. Each GPU thread reads an active cell coordinate from the global memory and performs ray shooting according

to the ray density and stores the partial result in its own register, as in the ray shooting kernel 1. At the end, the partial result of all amplification calculations are copied back to the CPU memory. The partial result adding of each amplification calculation is performed on the CPU.

As Figure 5.23 shows, the image areas of all amplification calculations are first solved by multiple CPU cores, then the GPU performs ray shooting for all amplification calculations in a single kernel launch.

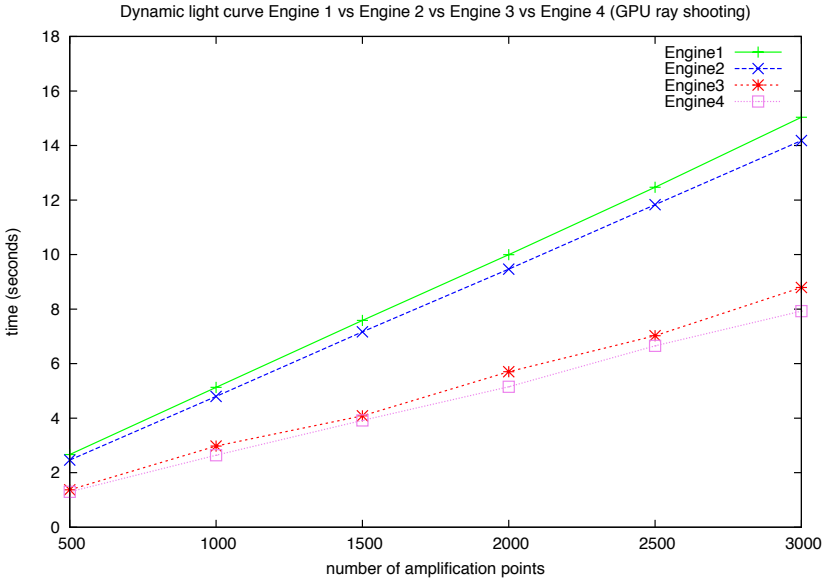

Figure 5.24: Light curve engine performance GPU Engine 1 versus GPU Engine 2 versus GPU Engine 3 versus GPU Engine 4. The error bars are smaller than the plot symbols.

Figure 5.24 shows a performance comparison between light curve engines 1 to 4. The overall performance has 5% to 10% improvement while the GPU ray shooting alone has 12% (500 amplification points calculation) to 27% (3000 amplification points calculation) improvement. The combined ray shooting of all amplification point calculations in a single kernel launch has reduced the GPU initialization overhead and improved utilization of the GPU.

Light curve engine 4 is around twice as fast as the light curve engine 1 on a quad Intel Core i7 3.0GHz CPU running in turbo mode and an NVIDIA GTX480 GPU. The speedup factor depends on the configuration of the machine. Since the CPU is running at higher frequency in single thread application (light curve engine 1), the multi-thread image area solving algorithm (light curve engine 3 and 4) is not linearly sped up according to the number

of CPU cores.

5.3.5 Light Curve Engine 5 - Multi-threading plus overlapping computation between CPU and GPU

The performance of the light curve engine has incremental improvement from light curve engines 1 to 4 by achieving better parallelism and better utilization of both CPU and GPU. It may even be possible to achieve better utilization by using overlapping of the CPU and GPU computation as in light curve engine 2 and apply it to light curve engine 4. Instead of waiting for the CPU to resolve the image area of all amplification calculations, we can sub-divide the job and transfer the active cells list of first batch amplification calculations for the GPU. While the GPU is performing ray shooting of the first batch of jobs, the CPU can start to compute the image area of the second batch of amplification calculations. The whole process is repeated until all jobs are completed. This gives maximum utilization of both CPU and GPU.

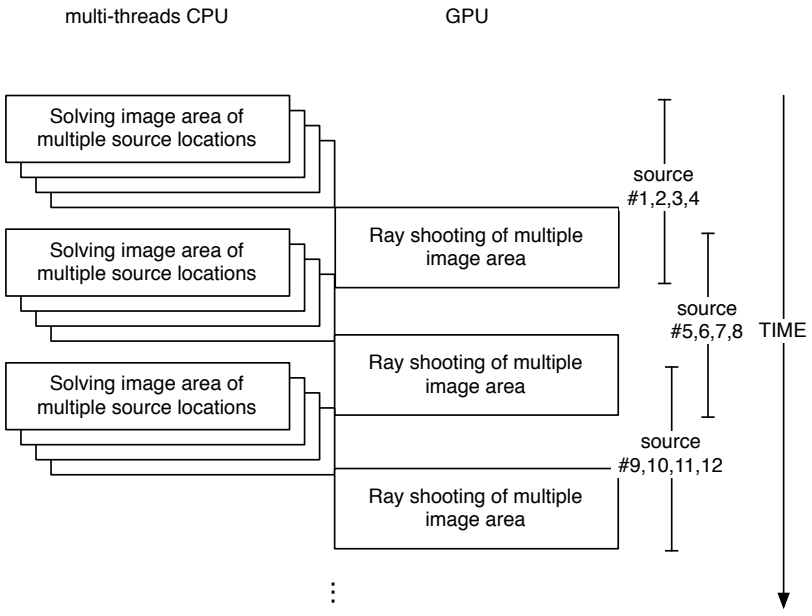

Figure 5.25: Light Curve Engine 5

Figure 5.25 shows the overlapping computation of multiple CPU cores and GPU. The image area of the first batch of source amplification is copied to the GPU memory once it is done and the GPU kernel is called. The CPU then takes back control and starts the image area solving for the second batch of source amplification calculation.

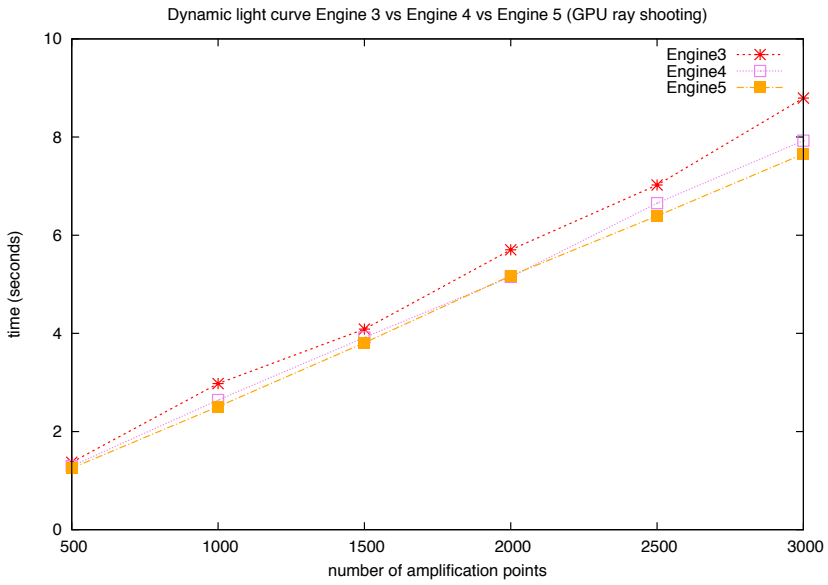

Figure 5.26: Light curve engine performance GPU Engine 3 vs. GPU Engine 4 vs. GPU Engine 5. The error bars are smaller than the plot symbols.

There are only slight performance improvement of the Light Curve Engine 5 over the Light Curve Engine 4. Although we use the same technique to overlap the CPU and GPU computation as in Light Curve Engine 2, the overhead in allocation and division of jobs, as well as threads creation, is too big to see a significant benefit. The ideal number of jobs in each batch is different for each light curve computation and depends on the number of amplification point calculations as well as the size of the computation task for each amplification calculation. The Dynamic Light Curve Engine is best suited for high amplification caustic crossing events with orbital motion effect which are usually very expensive to compute just with the CPU.

5.4 Conclusion

In this chapter, we have discussed a hybrid method using both CPU and GPU to generate microlensing light curves with the orbital motion effect for microlensing modelling. Different parts of computations are assigned to the CPU and GPU to take advantage of their strength. We used the CPU to solve the complex polynomial equation and dynamically solve the image area while the GPU is used to perform inverse ray shooting in highly parallel fashion. We have also discussed how to further improve performance by arranging and overlapping the

computations of CPU and GPU.

Although our hybrid method provides a solution for light curve computation with the orbital motion effect, a suitable optimization technique is needed in order to successfully obtain the correct model for a microlensing event. We discuss different optimization strategies used in microlensing modelling in Chapter 6 as well as showcase some events that are modelled using our strategies.

Chapter 6

Development of Modelling Strategies and Microlensing Event Case Studies

6.1 Introduction

The light curve computation methods discussed in Chapter 4 and Chapter 5 aim to model microlensing events by generating a theoretical light curve and matching the observed data. In microlensing event modelling, it is generally required to search in multi-dimensional parameter space to successfully model an microlensing event. The magnification map generation and track extraction technique in Chapter 4 allows us to search through a part of the parameter space efficiently while the dynamic light curve engine in Chapter 5 provides us with the ability to search the full parameter space accurately and effectively. A microlensing event can be modelled successfully by combining both light curve generation methods and utilizing different modelling strategies.

In this chapter, we first discuss how we use the magnification map technique and dynamic light curve engine to model microlensing events. We also show how we use different modelling strategies and optimization techniques to acquire the potential model. Later in the chapter, we display some successfully modelled microlensing events as well as exclusion zone computation.

6.2 Modelling Strategies

Although some complex microlensing events require a search through 10+ parameter space, many anomalous microlensing events are binary lenses with finite source effects and can be described by seven parameters. This is ideal for the magnification map technique to be used in search for those seven parameters. For more complex events, the dynamic light curve engine can be used to search in even higher-dimensional parameter space with more accurate models. Furthermore, since the initial grid search using the magnification map technique divides the mass fraction ε and projected separation d into discrete values, the dynamic light curve engine can be used to obtain a more accurate model from the result of the grid search as it optimise all parameters. Both techniques have their own advantages and they complement each other.

6.2.1 Microlensing modelling using magnification map technique

A binary lens microlensing event with finite source can be modelled with seven parameters $\{\varepsilon,\ d,\ t_0,\ t_E,\ u_0,\ \phi,\ \rho\}$. We can split them into two sets of parameter space, $\{\varepsilon,\ d\}$ and $\{\ t_0,\ t_E,\ u_0,\ \phi,\ \rho\}$. Since a magnification map essentially represents parameters $\{\varepsilon,\ d\}$, and a track extracted from the magnification map is described by the parameters $\{\ t_0,\ t_E,\ u_0,\ \phi,\ \rho\}$, the magnification map technique is a very effective method to search through these seven parameters systematically. It is computationally expensive to generate a magnification map, but we have solved the computation challenge by the GPU in Chapter 4.

In general, a microlensing event will be observed by a number of different telescopes through various filters or "passbands". Each passband from each telescope will comprise a set of flux measurements and their corresponding uncertainties. The microlensing modelling procedure on a given event requires minimizing an objective function defined as follows:

$$\chi^2 = \sum_{i=1}^{N_p} \sum_{j=1}^{N_i} \left(\frac{F_{ij} - F_{base,i} A(t,f)}{\sigma_{ij}} \right)^2 \tag{6.1}$$

where F_{ij}, σ_{ij} are the j^{th} flux and uncertainty measurements for the i^{th} passband, $F_{base,i}$ is the baseline flux for passband i, $A(t,f)$ is the theoretical amplification profile for the microlensing model as a function of time and the set of parameters, f.

Figure 6.1 shows the strategy for searching the seven parameters $\{\varepsilon,\ d,\ t_0,\ t_E,\ u_0,\ \phi,\ \rho\}$ by using magnification map generation and track extraction combined with the downhill

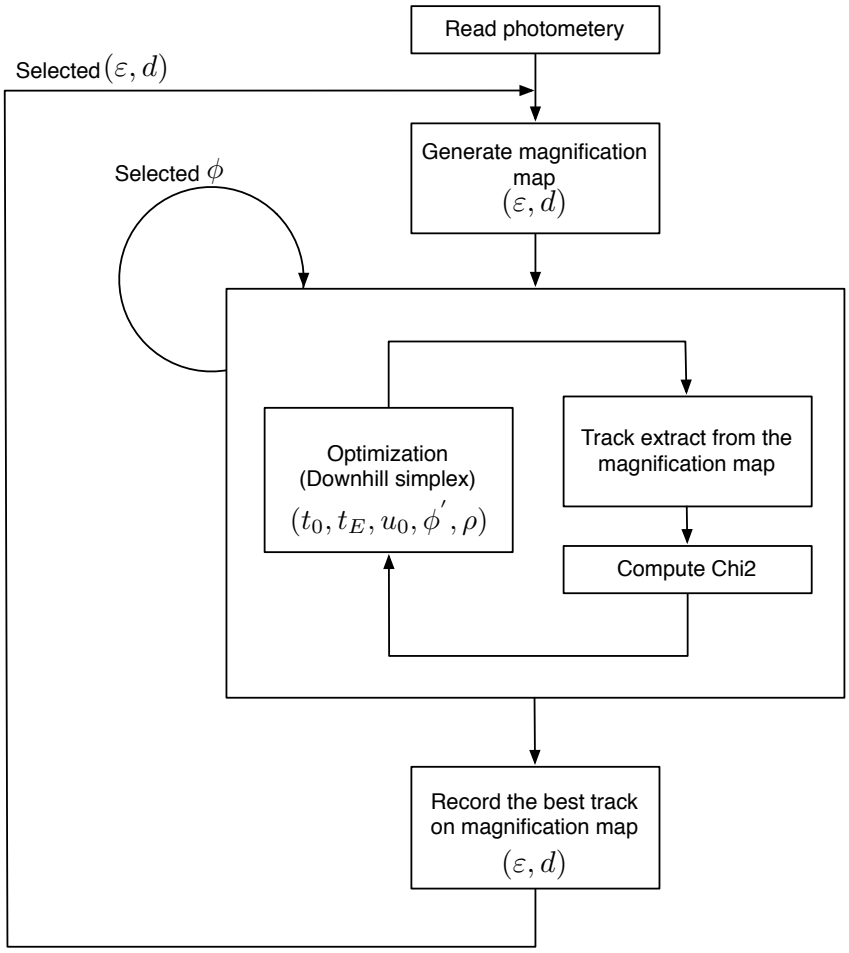

Figure 6.1: Work flow for the grid search using the magnification map technique.

simplex optimization technique. The χ^2 surface of the seven parameters usually has several local minima. It is essential to find and examine all of the potential minima to determine which is the global minimum. Since it is not practical to search the seven parameters space by brute force, even using the GPU, we need to strategically divide the parameter space and perform a systematic search to ensure no local minima are missed.

Hybrid optimization and brute force searching

We use a combination of optimization and brute force techniques for searching for the best-fitting light curve. The parameter space of $\{\varepsilon, d\}$ is divided into a 2D grid space and the best fit parameters of $\{ t_0, t_E, u_0, \phi, \rho\}$ will be found at each grid point. A magnification map is generated for each $\{\varepsilon, d\}$ combination and is used to search for the best fitted theoretical

light curve that matches with the observed data described by the parameters t_0, t_E, u_0, ϕ, and ρ.

On each magnification map, we are looking for the best source star track that produces the best-fitting light curve. Since the χ^2 surface is very rough with steep valleys, a further break down in parameter space is needed. One parameter affecting the shape and characteristics of the light curve significantly is ϕ, which describes the angle of the source star track. When the source star is close to or across a caustic, change in ϕ produces dramatically different light curves. Therefore, a break down of ϕ in discrete values to perform a systematic search is essential.

The parameters t_0, t_E and u_0 can be estimated by fitting a single lens light curve model to the observed data. Since the single lens light curve model can be computed very quickly and only parameters t_0, t_E and u_0 need to be fitted, an approximate single lens model can usually be found by simple optimization methods. The observed data with significant deviation from the single lens model is usually removed when fitting with a single lens theoretical light curve. The estimated value of t_0, t_E and u_0 will be used as the initial parameter for the optimization algorithm.

The parameter ρ represents the source star radius and affects the size of the source star area for integration on the magnification map. The size of the source star also has a significant effect (finite-source effect) on the light curve characteristics. When the source star track is close to or across a caustic, it usually produces significant deviations in the single lens model. But the size of the source star also has a role in the shape of the deviation. The larger the source star, the more washed out the features appears. The finite-source effect is expensive to compute, but we have sped up the finite-source computation significantly using the GPU. Therefore, this allows us to use optimization techniques to search for the value of ρ. The size of the source star can be estimated by collected statistics of the star sizes in our galaxy. We start searching ρ by choosing the statistically estimated value as the initial parameter for the optimization algorithm.

Downhill simplex optimization

As discussed in the previous section, the χ^2 surface is very rough with steep valleys. The aim of the strategy we use break down the seven parameter space is to discover all the local minima on the χ^2 surface. Each parameter in $\{\varepsilon, d, t_0, t_E, u_0, \phi, \rho\}$ is sliced strategically to create an entry point (initial parameter) to the local minima on the χ^2 surface.

In order to search the local minima on the χ^2 surface effectively from the initial parameters of each slice of the seven parameter space, we need to use an optimization method that is able to discover the local minima effectively but also be able to terminate itself quickly. The ability to converge quickly is important for our hybrid optimization and brute force searching technique to work effectively.

The downhill simplex optimization method [37], also called Nelder-Mead method is used in our searching technique. It is a multidimensional optimization method that does not require knowledge of partial derivatives. It uses the properties of a geometric object called a simplex, which has N+1 vertices in N dimensional space, to discover the minimum of a function by reflection, expansion and contraction of the vertices of simplex. This optimization method can converge to the minimum quickly if the function has a clear direction to the minimum. It is an effective optimization method for our parameter searching strategy as the parameter space is sliced into pieces that contain possible local minima.

Searching the best source star track strategies

Microlensing light curves come in all sorts of shapes depending on the lens system as well as the size and trajectory of the source star. In most cases, we are able to distinguish/guess the type of microlensing event from the observed data by experience. Some events may consist of distinct features like the binary caustic crossing shown in Figure 6.11. Others may have strong finite-source effect as shown in Figure 6.17. Since there are may possible combination of the trajectory of the source star and the lens system, some events may not be possible to classify right away.

Each type of event discussed in the previous paragraph requires a different strategy to be modelled effectively. We have developed an optimization strategy using the downhill simplex method for each type of event. The following sections discuss using combination of optimization strategies to model different type of events.

Optimization strategy 1

For binary events with distinct binary caustic approach/crossing features with little finite-source effect, strategy 1 shown in Figure 6.2 is used. We divide ϕ in the range $0 - \pi$ into 72 steps and the angle in each step is represented by $\phi^{'}$. In each step, $\phi^{'}$ along with estimated values of t_0, t_E, u_0 and ρ are used as initial parameters for the simplex optimization

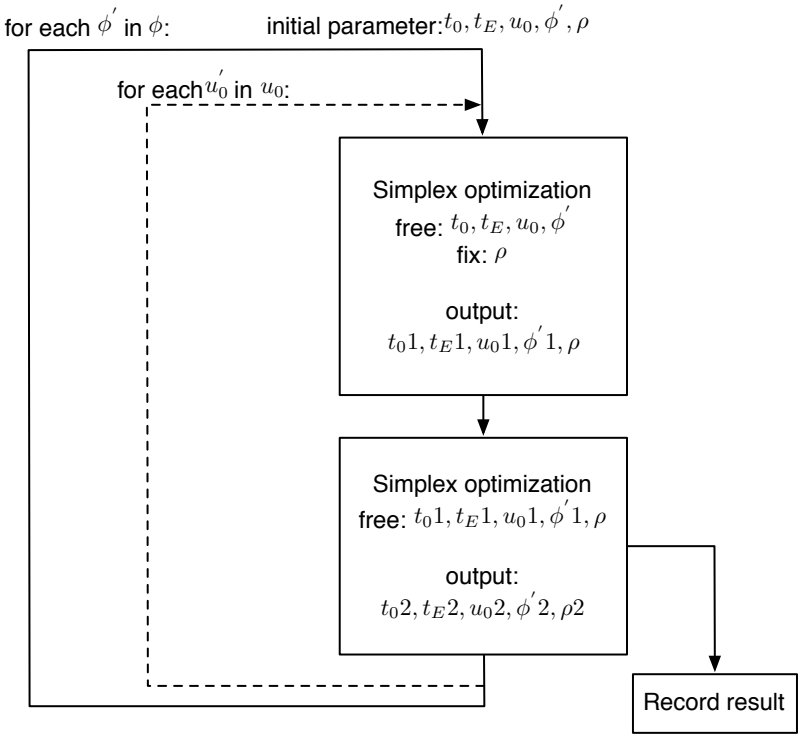

Figure 6.2: Optimization strategy 1 work flow

method. The simplex optimization is run twice. In the first run, only $\{t_0, t_E, u_0, \phi^{'}\}$ are free parameters and ρ is fixed to 0.001. In the second run, the output of the optimized parameter $\{t_0, t_E, u_0, \phi^{'}\}$ are used as the initial parameters along with ρ for the second optimization run. All parameters are set free and searched by the simplex optimization method.

The reason to first fix ρ and only search for $\{t_0, t_E, u_0, \phi^{'}\}$ is because $\phi^{'}$ is a sensitive parameter in binary passage/crossing events. The shape of the light curve will change significantly with slight changes of $\phi^{'}$ while ρ does not cause such a significant effect to the χ^2 surface compared with $\phi^{'}$. Since the downhill simplex optimization method become less effective when higher-dimensional space is being searched, it is important to first focus on searching the parameter with significant effect on the overall light curve. Once the first simplex optimization converges on a model to match the general shape of the light curve, the second simplex optimization run can fine tune all the parameters and search the parameter ρ.

Whether or not the inner loop is executed, depends upon the particular microlensing case. For some binary caustic crossing events, it is difficult to estimate the correct u_0

parameter since the caustic crossing feature dominates the light curve (which is hard to fit by a single lens model). Changes in u_0 in caustic crossing events have significant effect on the light curve since they change how the source star crosses the caustic. Since the simplex optimization method may not be able to jump between the steep valleys in u_0 χ^2 space, we choose several values of u_0 as initial parameter and repeat the above optimization run for each value.

Optimization strategy 2

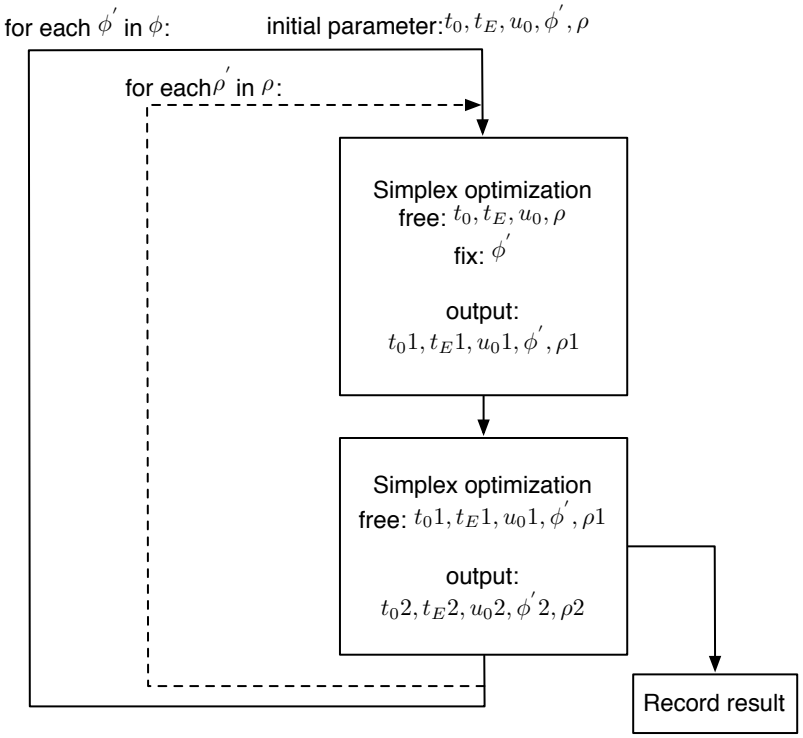

Figure 6.3: Optimization strategy 2 work flow

For an event that is dominated by a heavy finite-source effect with caustic approach/crossing, strategy 2 shown in Figure 6.3 is used. Like in strategy 1, we first try to converge the model to match the overall shape of the light curve. Parameter ϕ is again divided into 72 steps and the parameters are again searched by two passes with the simplex optimization method. The parameter $\phi^{'}$ is a fixed parameter instead of ρ as in strategy 1 for the first optimization run. The parameters $\{t_0, t_E, u_0, \rho\}$ are set to be free parameters in the first run and the optimized result is used as the initial parameters for the second simplex optimization run with $\phi^{'}$ also being a free parameter. All parameters are free in the second simplex optimization.

Since the microlensing event is dominated by the finite-source effect, ρ is an important parameter that has a significant effect on the shape of the light curve, while ϕ is less sensitive in this type of microlensing event. Therefore we first search in parameter space {t_0, t_E, u_0, ρ} and let $\phi^{'}$ be fixed. The aim of the first optimization run is to search for the right value of ρ. Once the model converges to match the overall shape of the light curve, the second optimization run is used to search for the best-fitting model in parameter space {t_0, t_E, u_0, ρ, $\phi^{'}$}.

For this type of microlensing event, multiple models that also match with the overall shape of the light curve may be found with different values of ρ when u_0 is not being well estimated. In this case, we choose several value of ρ as initial parameters and repeat the two-step optimization run to make sure all the local minima are discovered.

Optimization strategy 3

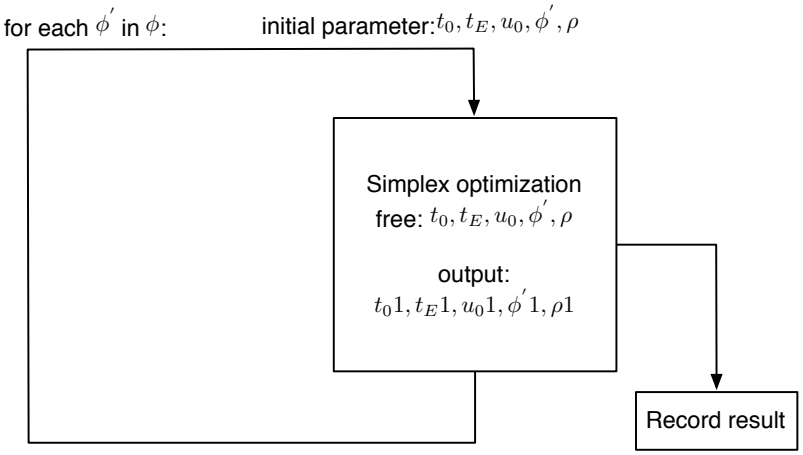

Figure 6.4: Optimization strategy 3 work flow

For a classic planetary event, a small deviation usually happens at the shoulder of a single lens light curve. The deviation is usually caused by a caustic approach/crossing. Since the planetary microlensing event has a very small caustic compared with a binary event, the deviation is usually small and with short duration. However, this type of event is usually straightforward to model. Figure 6.4 shows the work flow of strategy 3 for this type of event.

Since most parts of the light curve can be modelled accurately by the single lens light curve, the parameters t_0, t_E and u_0 are well estimated. The trajectory of the source star,

described by ϕ, can be estimated by the time which the deviation occurs on the light curve. The height of the deviation provides an estimation of ε and d can also be estimated assuming the source star approaches/crosses the secondary caustic. The size of the source star can also be estimated by the sharpness of the deviation. In general, the sharper the deviation, the smaller the source star compared with the caustic size. The caustic size is mainly affected by the mass fraction ε.

An estimation of the seven parameters $\{\varepsilon, d, t_0, t_E, u_0, \rho, \phi^{'}\}$ can be worked out prior to starting the search in the parameter space. Therefore, we can limit the range when searching some of the parameters. We may only generate magnification maps with combinations of ε and d around the estimated value. For each magnification map, ϕ may only be divided into several steps in a range close to the estimated value, instead of searching the entire range. Only one simplex optimization run is enough since most of the parameters are well estimated. All parameters in parameter space $\{t_0, t_E, u_0, \rho, \phi^{'}\}$ are set free for searching for the optimized value.

Optimization strategy 4

In some cases, we may need to perform a blind search in all parameter space when parameters are not well constrained. The magnification map technique speedup by the GPU together with our optimization strategies can perform such searches very effectively. In strategies 1 to 3, we strategically break down the parameter space to give important parameters higher priority in the optimization search to increase the effectiveness in the ability to discover local minima on the χ^2 surface. Similar strategies can be used in a blind search in all parameter space.

To perform a blind search, we select a set of values as the initial parameters in the parameter space $\{t_0, t_E, u_0, \rho\}$. The combination of those parameters will act as the initial parameters to the downhill simplex optimization method. If each parameter has 4 selected values of initial parameter, there will be 4 x 4 = 16 optimization runs from parameter space $\{t_0, t_E, u_0, \rho\}$. So if ϕ is divided into 72 steps, there will be 4 x 4 x 72 = 1152 optimization runs on a magnification map which represents a set of $\{\varepsilon, d\}$.

6.2.2 Systematic grid search

In the previous section, we discussed different strategies in searching the parameter space $\{t_0, t_E, u_0, \rho, \phi^{'}\}$. Although the magnification map technique allows us to search in parameter

space $\{t_0, t_E, u_0, \rho, \phi^{'}\}$ very quickly, a different magnification map has to be generated in order to search in each parameter space $\{\varepsilon, d\}$. A search on the map is essentially searching for the best-fitting parameters of $\{t_0, t_E, u_0, \rho, \phi^{'}\}$.

We have a strategy in place to search for the best-fitting model on a magnification map which represents parameters ε and d. A grid search in parameter space $\{\varepsilon, d\}$ will allow us to find the best-fitting model in all parameter space and also discover all the local minima. This is one of the challenges in microlensing modelling as we not only need to find the correct model, but also need to rule out any other possible models.

Fixed grid search

One of the common grid search strategy is to have a fixed grid of parameters $\{\varepsilon, d\}$. This allow us systematically search for the best-fitting model as well as all the minima on the χ^2 surface in parameter space $\{\varepsilon, d\}$. Figure 6.16 shows an example of a fixed grid χ^2 surface. The fixed grid search is usually straight forward to execute, but the step size of both ε and d can affect the result significantly. A step size that is too big may miss out some local minima while a step size that is too small may take too long to complete the grid search. The optimum step size in ε and d depends on the microlensing event. For example, a binary caustic crossing microlensing event with a sharp spike may require smaller step size in ε as small changes in ε have a big effect on the size of the binary caustic, while a planetary microlensing event that crossing the secondary caustic may require smaller step sizes in d as the position of the secondary caustic is very sensitive to changes in d. There are many more scenarios that require different optimum step sizes in ε and d, it usually requires experience or trial and error to determine the optimum step size.

Random non-fixed grid search

Since it is difficult to determine the best step size of ε and d in a grid search for a particular microlensing event, a better approach is to perform a non-fixed grid search with randomly selected grid points in parameter space $\{\varepsilon, d\}$. For each magnification map generation, a random value of ε and d is selected within the determined range. This allow grid points in parameter space $\{\varepsilon, d\}$ to be evenly sampled over time. This approach avoids the problem in selecting the optimum step size in ε and d but still is able to discover all the minima as long as the random selected ε and d are evenly sampled and with a large enough iteration.

Once an initial model is discovered, the parameter is dispatched to the dynamic light curve engine to search for a more accurate model. This procedure is repeated once another initial model is discovered. This approach also allows us to discover the best possible fitting model faster as we do not need to wait until all the grid points are searched to obtain the complete χ^2 surface as in the fixed grid search.

Stochastic grid search for Real time modelling

For some microlensing events, modelling time is critical as we model the newly discovered microlensing event as it develops. Telescope observation resources are limited and we can only focus on some important events like planetary events. Therefore, predicting a microlensing event development by real time modelling is important to help distribute observation resources.

Figures 6.7 illustrate that there is usually a valley on the χ^2 surface in parameter space $\{\varepsilon, d\}$. We can speed up the discovery rate of minima by altering the probability distribution for selecting the random position in parameter space $\{\varepsilon, d\}$. We first start with the random non-fixed grid search until a certain number of grid points are sampled. Then we increase the probability of surrounding locations of the sampled grid points with lower χ^2 value. Therefore, the locations close to the valley on the $\{\varepsilon, d\}$ χ^2 surface will be sampled more frequently, thus increasing the effectiveness in discovering all the possible models.

For each possible model discovered by the stochastic grid search method, we fetch the model to the dynamic light curve engine to find the best-fitting model in the whole parameter space.

6.2.3 Complex modelling using the dynamic light curve engine

The magnification map technique allows us to effectively search in parameter space $\{\varepsilon, d\}$. But since each search aims to optimize other parameters on a magnification map, ε and d are not optimized. The best-fitting model need to be optimized in all parameters. The dynamic light curve engine discussed in Chapter 5 not just allows us to search for the optimum model in the seven parameter space $\{\varepsilon, d, t_0, t_E, u_0, \rho, \phi\}$, but also allows more complex microlensing modelling like searching models with orbital motion.

Unlike the magnification map technique, each amplification point is computed independently in the dynamic light curve engine. This allows us to optimize all the parameters

when fitting a microlensing model. The downhill simplex optimization method is less effective when searching high-dimensional parameter space and the best-fitting model with 2nd order effects may be described by 15+ parameters. Therefore, an optimization method that is able to search in such a high-dimensional parameter space is needed.

Markov chain Monte Carlo (MCMC) optimization

In order to effectively search in the high-order-dimensional parameter space, we utilize a modified version of the MCMC algorithm from [71]. We first fetch the initial starting parameters and the step size of each parameter, which will be used to determine the probability distribution in sampling value, to the MCMC algorithm. Each parameter space will be sampled from the standard normal distribution constructed by the step size of each input parameter in the first 20 chains. After the first 20 chains, the MCMC algorithm samples values from a correlation kernel constructed by learning the previous sampled results. This allows moving towards the minima quickly. Note that we only use the previously accepted sample values to construct the correlation kernel.

The correlation kernel should allow χ^2 to improve quickly, for an optimization search using the local minima from the grid search. We set the temperature in the MCMC algorithm to be high in order to discover the minima quickly. However it may get stuck in a local minima. Therefore we need to have a mechanism in the algorithm to avoid being stuck in a single minima.

We implemented two mechanisms to avoid being stuck in a single minima. The first one is to perform a big jump when more than 25 consecutive chains have been rejected. Once the algorithm needs to perform a big jump, it samples parameters bigger than 4 sigma in distribution from the correlation kernel. This allows a relatively big jump from the previous sampling area. The second mechanism is to restart the construction of the correlation kernel once 10× *dimension number* chains have been sampled. The algorithm performs a 4 sigma big jump like the first mechanism, rebuilds the correlation kernel, forgets all the previous chains and starts sampling 20 chains with the standard normal distribution defined by the step size. This allows a fresh start to the MCMC algorithm and not being stuck by the old correlation kernel which may not give the optimum distribution sampling.

6.3 Event showcase

In this section we show several selected microlensing events that are modelled by our modelling techniques and strategies. They consist of different types of microlensing events ranging from binary stars to planetary events and each with unique challenges in modelling. We also utilize our GPU code to perform exclusion region computation on multiple events.

6.3.1 Binary microlensing event - MOA2002-BLG-042

This is a high magnification microlensing event discovered by MOA in 2002. High magnification events are sensitive to small planets and the unusual shape of the observed light curve indicated it may be a special event. Although a considerable amount of effort had been put into modelling this event, no compelling model had been successfully discovered. We used our modelling strategy and computing technique to successfully model this event.

The challenge of modelling this microlensing event is that the nature of the lens system is unclear. The observed data show that it is possibly a caustic crossing event. The sharp rise in brightness indicates a caustic entry. But the unusual shape of the observed light curve does not indicate whether it is a binary lens or a planetary system. It also indicates a finite-source effect is present in the MOA data, which shows a round peak instead of a spike-like peak. Furthermore, this event is not fit very well by a single lens model which gives little information on parameters t_0, t_E and u_0.

Parameter estimation

Since all parameters cannot be well estimated, a blind search (strategy 4) may be used to search for the correct model. But a blind search takes considerably longer, and we may first try to best guess some parameters and only blind search for the parameters that cannot be constrained. Although we cannot estimate t_0 and t_E accurately due to the unusual shape of the observed data, a best guess of those parameters provides us with a starting point. The value of t_0 should be around the peak of the observed data and t_E also can be estimated by measuring the event duration. We choose t_0 to be 2488.0 and t_E to be 40.0 as initial parameters.

For u_0, we do not have enough information to have a good estimation, so we selected a range of values to be initial parameters in our grid search. The range we used for u_0 as

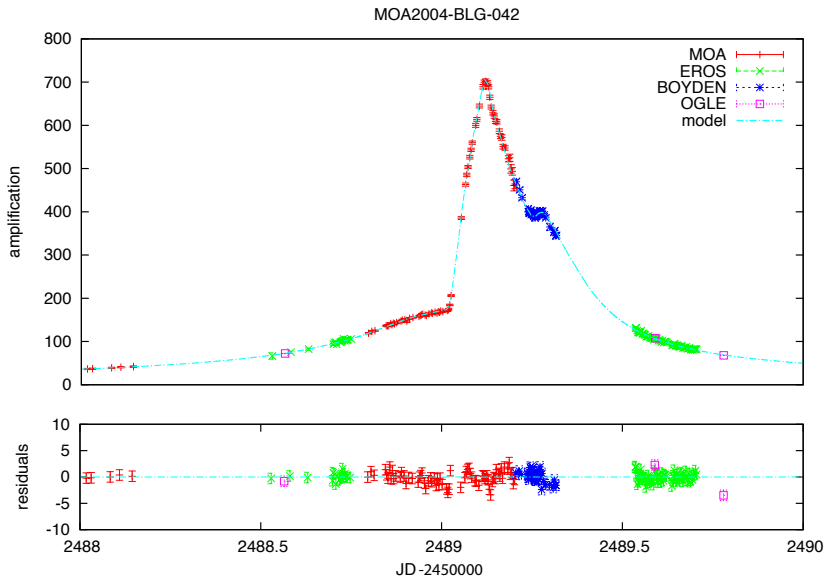

Figure 6.5: Light curve from 4 telescopes of the microlensing event MOA2002-BLG-042, together with the best-fitting binary-lens model. In the horizontal axis, JD is the number of Julian Days since JD2450000. The parameters of the model are $\varepsilon = 0.6526, d = 11.95, t_0 = 2487.94, t_E = 41.82, u_0 = 0.006294, \phi = 6.114, \rho = 0.001163$

initial parameter is dependent on the range of ε that we are searching. Since the caustic size depends on ε and a planetary caustic is much smaller than a binary caustic. We already know that it is a caustic crossing event, therefore we can constrain the search range of u_0 according to the size of the caustic.

Although there is finite-source effect observed at the peak, it does not dominate the effect of the caustic crossing. Therefore we choose the initial parameter of ρ as the typical value 0.001 for initial parameter.

Grid search

Before the grid search starts, we made an estimation on the parameters t_0, t_E and ρ and a range of initial values of u_0 as initial parameters for the grid search. We use strategy 1 with an extra multiple u_0 search to perform the grid search. Since we had chosen different ranges of initial value of u_0 according to ε, we first perform a planetary mass fraction range search in ε with d ranging from 0.01 to 3.0. The choice of range in d is due to the probability of having a planet in the above range is the highest.

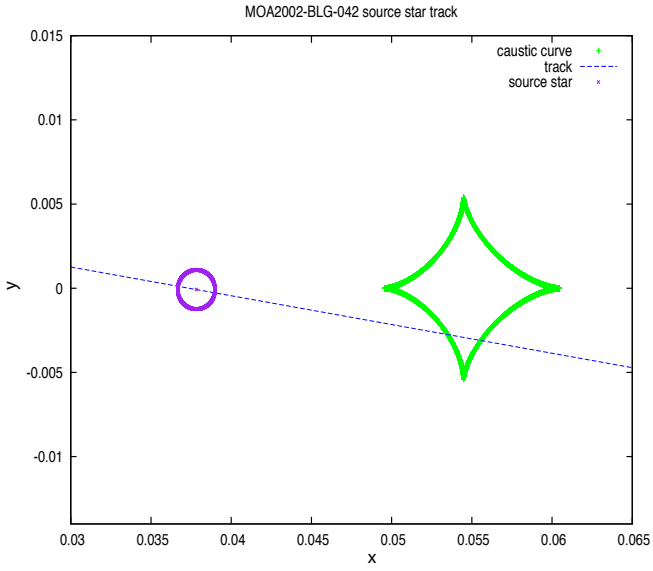

Figure 6.6: Diagram representing the geometry of the best-fitting model for MOA2002-BLG-042. The dotted line shows the trajectory of the source star as it approaches the caustic. The axes represent the position on the source plane in units of the Einstein radius.

The search in the planetary mass fraction range did not discover a model match with the observed data. We performed our search in the binary star mass fraction range with d ranging from 0.1 to 4.0 for a close binary star model search. Unfortunately, we also did not discover a matching model in the close binary range. Therefore, the next logical direction is to search for a wide binary star model. But a search in d in wide binary star range takes significantly more computation resources.

We had performed our grid search using the magnification map technique speedup by the GPU. It only takes several hours to perform the above search by a desktop computer equipped with GPU. This amount of computation used to be performed on a cluster computer. Therefore, searching the range in d in the common planetary model and the close to common binary star model used to take a considerable amount of resources, not to mention searching model beyond the above range. With our GPU code, we are able to perform an extended range grid search in a reasonable time on a desktop computer.

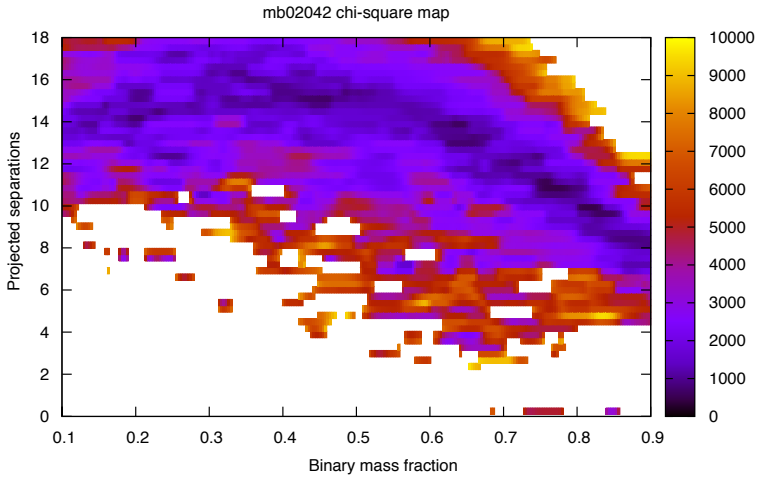

Figure 6.7: MOA2002-BLG-042 χ^2 map resulting from a grid search of binary models over the binary mass fraction, ε, and the projected separation, d. This was generated using optimisation 1 described in Section 6.2.

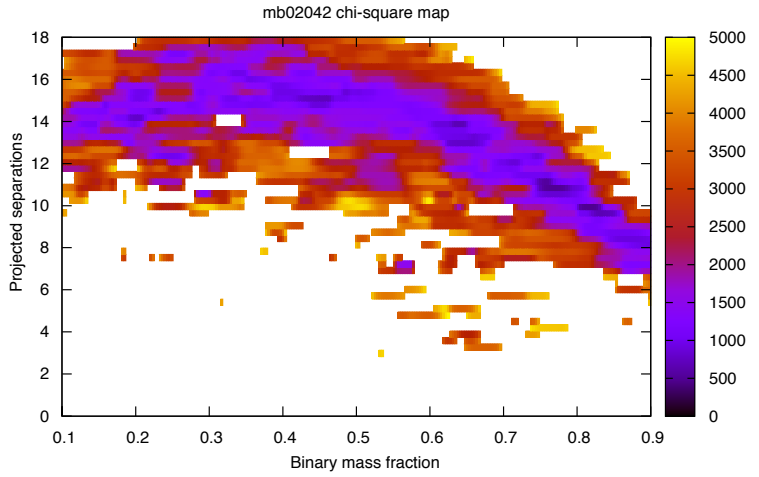

Figure 6.8: MOA2002-BLG-042 χ^2 map as in Figure 6.7. The stretch in χ^2 ranges from 0 to 5000.

Extended range grid search

We have performed an extended range grid search in d with range from 0.1 to 18 using our GPU code. Figure 6.7 shows the χ^2 map of the grid search result. Note that there is a

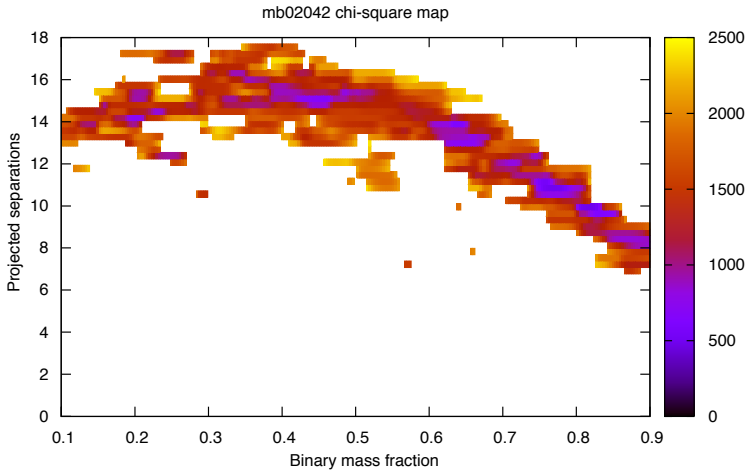

Figure 6.9: MOA2002-BLG-042 χ^2 map as in Figure 6.7. The stretch in χ^2 ranges from 0 to 2500.

valley in the ε, d parameter space where most minima are located. Figure 6.8 cuts the χ^2 range up to 5000 and further displays the location of the minima. Figure 6.9 restricted the range of χ^2 up to 2500 and several minima in the ε, d parameter space can be seen. Each of those minima is put to future investigation in order to find the best-fitting model.

Since the grid search method only searches the best fitted model based on the ε, d set, future modelling with a different technique is required to find the true minimum. The parameters of each minima are used as initial parameters for modelling using the dynamic light curve engine with the MCMC optimization method. Therefore, all parameters will be optimized. The MCMC algorithm is tuned to have a smaller jumping factor so it does not try to cross over to the other minima but instead investigates the local minima.

There are several minima that produce similar models. Figure 6.5 shows the best-fitting model after all the minima are investigated and Figure 6.6 shows the source star track of the best-fitting model. It is a wide binary model with mass fraction $\varepsilon = 0.6526$ and projected separation $d = 11.95$. It is a challenging microlensing event to be modelled due to the large projected separation, which means a very wide range of parameter space in ε and d needed to be searched. An extensive search like this usually takes months to complete. It also takes weeks even on a cluster computer with hundreds of nodes. But with our magnification map technique and dynamic light curve engine speed up by the GPU, it only takes days on a

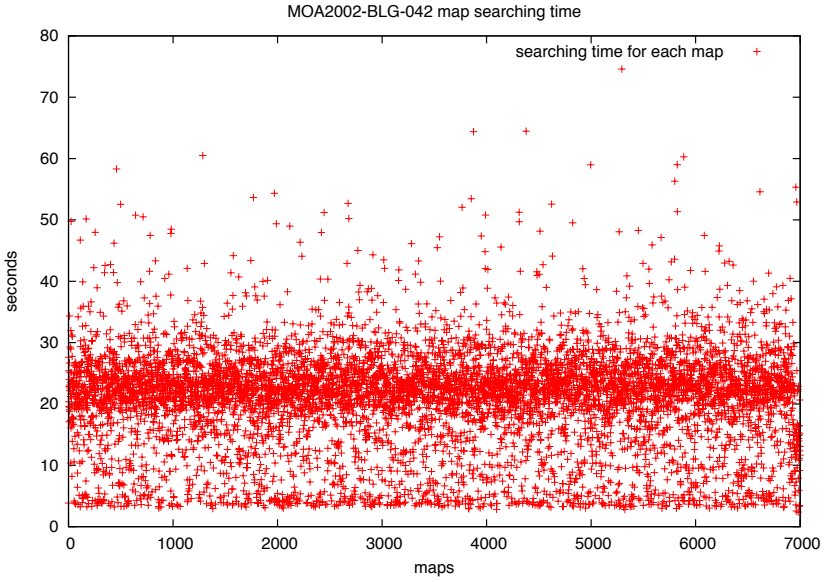

Figure 6.10: Searching time on each magnification map in MOA2002-BLG-042 modelling

desktop computer.

Figure 6.10 shows the searching time on each magnification map of the extended range grid search. There are in total 7000 maps being generated and searched using the discussed strategy. It only takes on average 22.5 seconds for each map. The total computation time is around 45 hours and it is performed on an old generation GPU, the G80 architecture. It is estimated the Fermi architecture GPU only takes around 15 hours with the equivalent task. The estimated computation time for such a task on a quad core CPU is around 50 days.

Figure 6.10 also shows the majority of magnification map searching time is around 22.5 seconds. This shows the simplex optimization method is well suited for such a task. It converges to local minima quickly but also stop quickly when no solution can be found. It is a stable and predictable optimization method.

6.3.2 Binary microlensing event with parallex - MOA2008-BLG-199

This microlensing event was discovered in 2008 and shows typical binary caustic crossing features in the observed data. It should be a straight forward event to model using our grid search method. However, we discovered there is also a parallax effect in this microlensing event when we did the modelling. The parallax effect can also be modelled very effectively

using our magnification map technique since parallax only affects the source star track and not the mass fraction and projected separation. Therefore we can perform a parallax search on the magnification map using our track extraction GPU engine.

The parallax effect is described by two parameters, π_E and π_N that give the parallax, in units of Einstein radius, in the East and North directions [42]. The two parameters describe the effect of parallax in two different directions respectively. There can be degenerate models, so a grid search in the parameter space $\{\pi_E, \pi_N\}$ is often required. If we starting searching the parameter space $\{\pi_E, \pi_N\}$ together with parameter space $\{t_0, t_E, u_0, \rho, \phi\}$, the chance of successfully discovering the correct model is very low. Since the χ^2 surface is very rough with a steep valley in parameter space $\{\pi_E, \pi_N\}$, it is very hard for any optimization method to search all parameters at once from their initial parameters. We develop a strategy in searching $\{\pi_E, \pi_N\}$ on the magnification map while performing a grid search in ε, d.

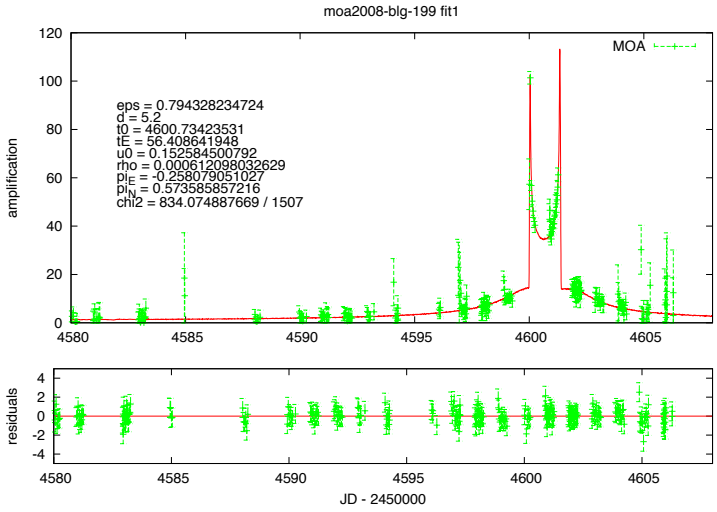

Figure 6.11: Light curve, obtained by MOA, of event MOA2008-BLG-199, together with the best-fitting binary lensing plus parallax model.

Since the observed data shows distinctive binary star light curve features, we start searching in the binary range mass fraction in ε. We set the range of d in our grid search from 0.1 to 8.0. We utilize strategy 1 for searching this microlensing event. For the initial parameters, we can only estimate t_0 and t_E roughly from the observed light curve with $t_0 = 4601.0$ and $t_E = 50.0$. Since there is no obvious finite-source effect, we set the initial parameter of ρ to

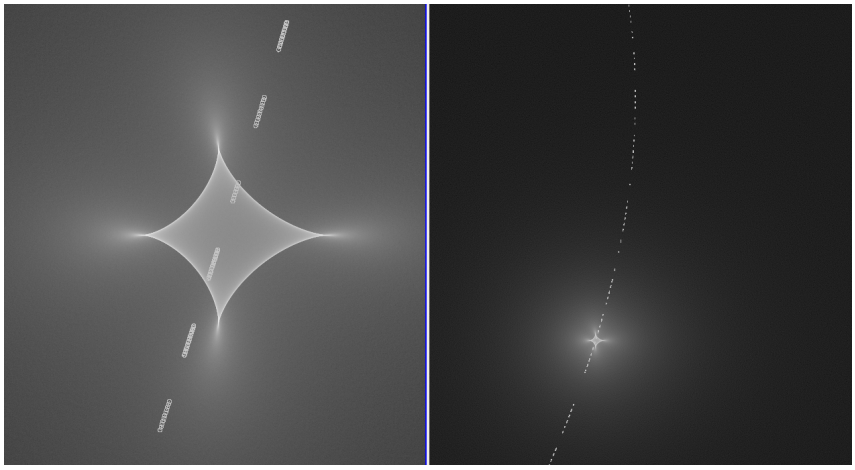

Figure 6.12: Magnification map image and source star track for the best-fitting parallax model of event MOA2008-BLG-199. The left image covers the range (0.1 - 0.6, -0.3 - 0.3) Einstein units on the source plane, and the right image covers the extent (-0.5 - 1.5, -0.75 - 1.75). Note here the curved trajectory of the source star.

0.001. We cannot make a good estimation of u_0, so we try initial values of u_0 from the list [0.001, 0.005, 0.01]. This should give the downhill simplex optimization method searching at different parts of the caustic.

For parameter space $\{\pi_E, \pi_N\}$, it is very inefficient if the optimization method searches all the parameters at once. It also highly likely that the simplex optimization method cannot discover the correct model since it does not often pass a very steep valley on the χ^2 surface when searching high-dimensional parameter space. A better approach is to first search the best-fitting parameters in parameter space $\{t_0, t_E, u_0, \rho\}$, then to use the result as initial parameters and perform a grid search in parameter space $\{\pi_E, \pi_N\}$.

Figure 6.13 shows the optimization strategy in modelling this microlensing event. The basic strategy is similar to strategy 1, we first divide ϕ into 72 steps and do a optimization run with $\phi^{'}$ and with each selected u_0 in the range. Note that all parameters in parameter space $\{t_0, t_E, u_0, \rho\ \phi^{'}\}$ are set to be free. We use the result from the best-fitting model as the new set of initial parameters and perform optimization runs with selected $\{\pi_E, \pi_N\}$. We divide π_E and π_N into 8 steps and the combinations of selected π_E and π_N act as initial parameter for each optimization run. Therefore, there is 3 + 8 x 8 = 67 optimization runs for each $\phi^{'}$ and there is a total of $72 \times (3 + 8 \times 8) = 4824$ optimization runs on each magnification map.

Figure 6.11 shows the best-fitting model of microlensing event MOA2008-BLG-199 and

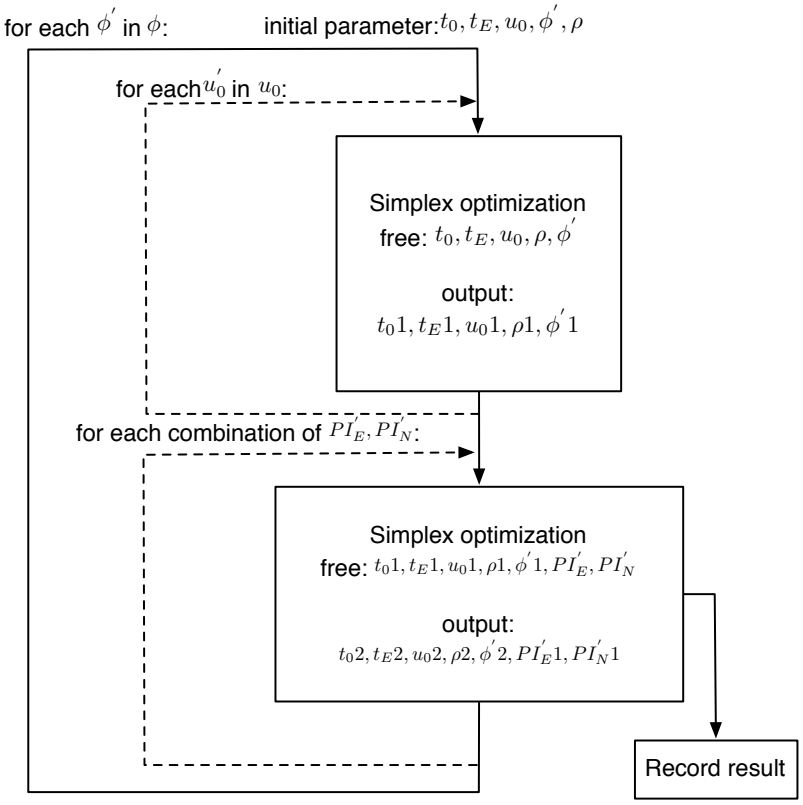

Figure 6.13: MOA2008-BLG-199 optimization flowchat

Figure 6.12 shows the source star track on the magnification map. The source star crosses the binary caustic with a curved track. Note that the curvature is not significance when the source star crosses the caustic. The parallax does not cause significant effect until the source is further away from the centre of the Einstein ring. That is why we can first search for the best-fitting model in parameter space $\{t_0,\ t_E,\ u_0,\ \rho\}$, then perform a grid search in parameter space $\{\pi_E,\ \pi_N\}$. In most cases, the parallax effect does not affect the fitting of parameters t_0, t_E, u_0, ρ and ϕ. Therefore, our optimization strategy is designed to take advantage of this feature and give an effective method in fitting the parameters π_E and π_N.

The χ^2 map of π_E vs π_N is shown in Figure 6.14. This shows a grid search in parameter space $\{\pi_E,\ \pi_N\}$ is essential as there are several local minima in π_E and π_N combinations at different places on the χ^2 surface.

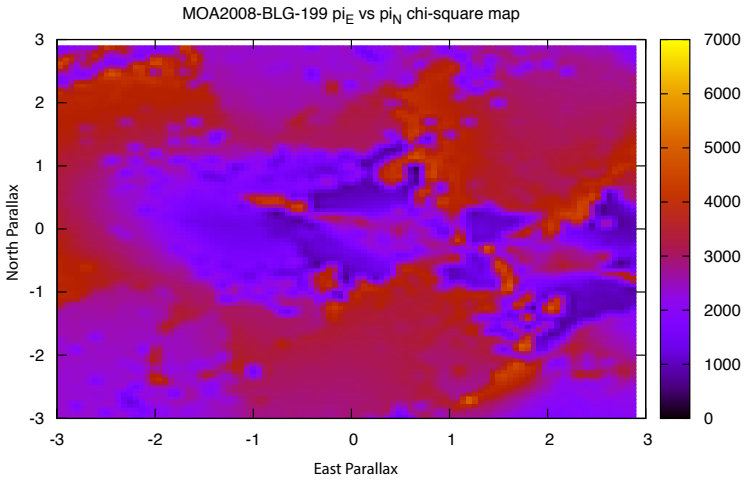

Figure 6.14: MOA2008-BLG-199 χ^2 map resulting from a grid search over the two parallax parameters π_E and π_N

6.3.3 Planetary microlensing event with a strong finite-source effect - MOA2007-BLG-400

This event was discovered in 2007 and is the first microlensing event that has a planetary signal hidden under the finite-source effect. The finite-source effect is a dominant feature of the observed light curve which looks very much like a single lens microlensing event with heavy finite source effect. But when fitted by a single lens model, a significant deviation is observed from the residuals. It was soon discovered that the source star is actually crossing a caustic. Since the source star size is even bigger than the crossed caustic, the deviated signal has been washed out. Therefore, the observed light curve looks just like a single lens event with finite-source effect.

Single lens model

We first try to fit this event by a single lens model using the magnification map technique. Since the finite-source effect can be computed effectively on a single lens magnification map, the high performance GPU track extraction engine is the ideal method to model this event. The peak of the light curve is well covered by the CMB data and the base line is well covered by the MOA data. Parameters t_0, t_E and u_0 can be well estimated. However, parameter

ρ is uncertain and multiple values of ρ are used as initial parameter in model searching, as described in Strategy 1. As the magnification map is only generated by a single lens, the parameter ϕ become irrelevant. Therefore, we fix the parameter ϕ to zero and use a range of values of parameter ρ as initial parameters together with the estimated t_0, t_E and u_0.

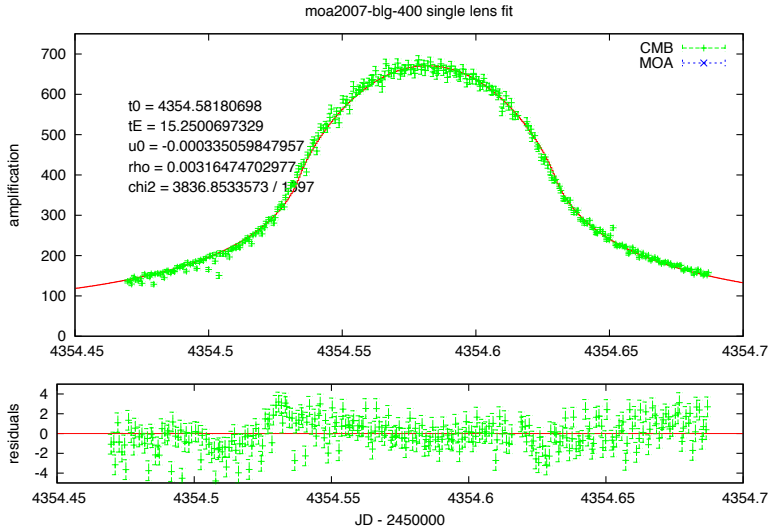

Figure 6.15: Light curve obtained by MOA and CTIO of the event MOA2007-BLG-400 together with the best-fitting finite-source single lens model. Deviations from this model are apparent in the residuals on both shoulders of the light curve.

Since the source star covered the entire caustic, there is no immediately obvious deviation from the observed data as the deviation feature as been heavily washed out. But by fitting the observed data by a single lens model, there is obvious deviation in the residuals, which are shown in Figure 6.15. Although one may argue that this may be caused by systematic error in the observed data, the double peak deviations from the residuals shows that the signal is actually real. This is because the double peak deviations in opposite directions is the evidence of the caustic entry and exit by the source star. Also, one reason that the signal had not been completely washed out by the finite source effect is due to it being a high amplification event, which is highly sensitive to planetary signals.

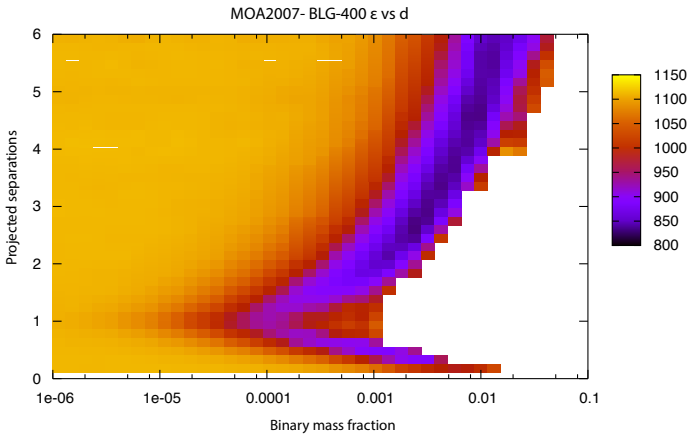

Figure 6.16: MOA2007-BLG-400 χ^2 map resulting from a grid search over the binary mass fraction and projected separations.

Grid search in ε and d

A grid search is performed in parameter space $\{\varepsilon, d\}$. Since parameters t_0, t_E, u_0 and ρ are well estimated by the single lens model fitting, the unknown parameters are ε, d and ϕ. We divide ϕ into 72 steps as initial parameters in the optimization search. For this event, we used a fixed grid search in parameter space $\{\varepsilon, d\}$ as we want to show the χ^2 surface in parameter space $\{\varepsilon, d\}$. Since the caustic has to been smaller than the source star, we constrain our search to the planetary range mass fraction.

The search is performed with ε in the range from 10^{-6} to 0.1 and d in range from 0.1 to 6. Figure 6.16 shows the χ^2 surface of a fixed grid search in parameter space $\{\varepsilon, d\}$. There is a clear valley in the χ^2 surface which shows the relationship of ε and d. All minima in the valley are examined and the best-fitting model is shown in Figure 6.17.

Planetary model

The best fitted planetary model has mass fraction of 0.0025 and projected separation of 2.7. The best-fitting single lens model yields $\chi^2 = 3836.85$ with degrees of freedom $\nu = 1596$.

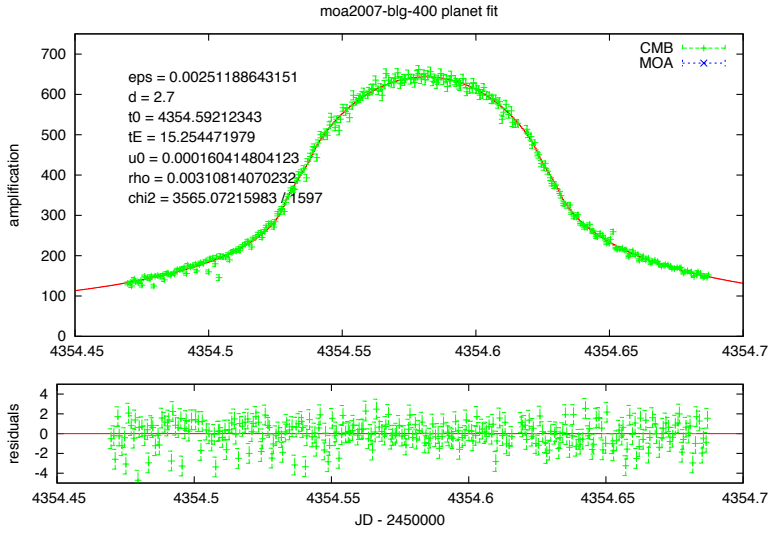

Figure 6.17: Light curve of MOA2007-BLG-400 as in Figure 6.15. Here the best-fitting binary-lens model is shown.

The best-fitting planetary model yields $\chi^2 = 3565.07$ with $\nu = 1593$. The planetary model is significant at the level of $\Delta\chi^2 = 271.78$. The best-fitting parameters of t_0, t_E, u_0 and ρ remain very close to the values in the single lens fitted model.

The source star size of the single lens model shown in Figure 6.17 may not be a typically large source star, but it is relatively large compared with the caustic as shown in Figure 6.18. Therefore, it has washed out the feature of a caustic crossing and does not make a significant change to the light curve. But there is a clear deviation signal in the residuals and the χ^2 minima valley shown in Figure 6.16 provides an evidence of the existence of a minima.

6.3.4 Complex modelling event - MOA2004-BLG-33

This is a high magnification event discovered in 2004 and is one of the most challenging microlensing events to model. Significant efforts have been put into modelling this event by different research groups over the years and no solution had been found. Different lens system configurations have been attempted including binary lens, triple lens and even quadruple lens systems. But at the end, we find a relatively simple solution for this event and successfully discover a model for this high magnification event.

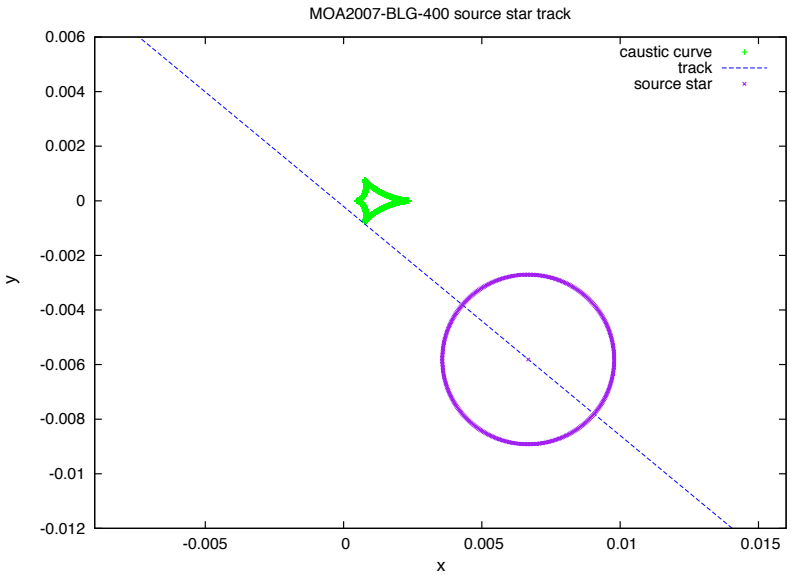

Figure 6.18: Diagram representing the geometry of the best-fitting model for MOA2007-BLG-400. The dotted line shows the trajectory of the source star as it approaches the caustic. The axes represent the position on the source plane in units of the Einstein radius.

This is a high magnification event with a light curve that significantly deviates from the single lens model. From experience, it looks like a planetary microlensing event with a complex lens system. This event also has a finite-source effect observed from the data at the peak and the caustic exit does not look like it is caused by a binary lens caustic. Moreover, we also performed a grid search in the binary mass fraction range as well as the planetary mass fraction range. Unfortunately, there is no matching model in both ranges. We had extended the search to include wide binaries as well, but still no discovery.

The next step we took is to perform a search with a triple lens system. Unlike the binary lens system which consists of 2 stars or a star with a planet, a triple lens system can be 1 star with 2 planets or 2 stars with 1 planet or 3 stars. The combinations of triple lens system configurations are a lot more than a binary lens system. Therefore, the grid search of a triple lens system is significantly more computationally intensive than a grid search for a binary lens system. The magnification map modelling technique using GPU discussed in Chapter 4 is ideal for such a computationally intensive task.

We randomly selected triple lens system configurations using the random non-fixed grid search method discussed in Section 6.2.2. Although an extensive search has been done, there are still no correct models found with a triple lens system. However, we discovered that there

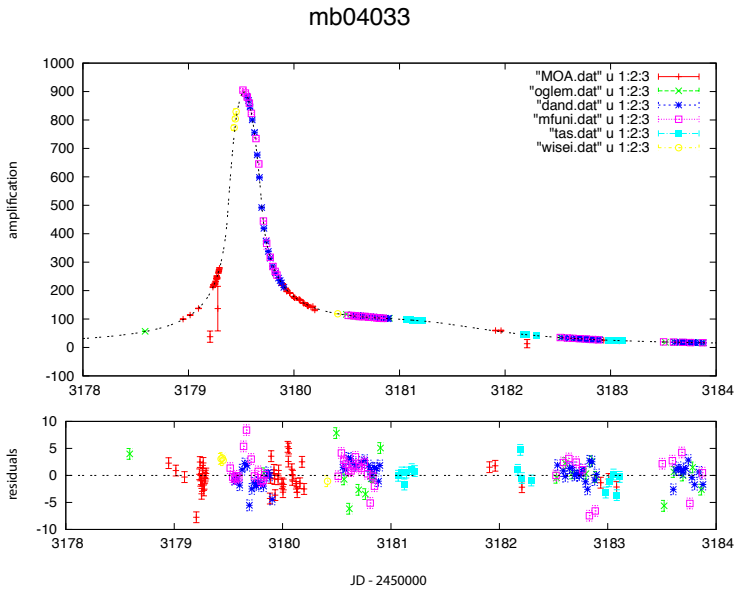

Figure 6.19: Light curve from 6 telescopes of the microlensing event MOA2004-BLG-33, together with the best-fitting xarallap model.

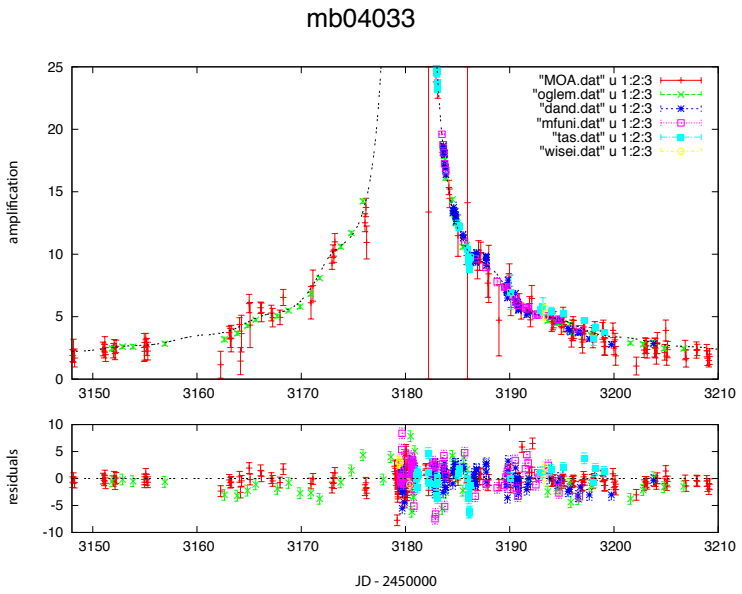

Figure 6.20: Detailed view of xarallap fitting model of MOA2004-BLG-33 as in Figure 6.19.

is a good fitting model with unphysical parallax parameters and the model showed short periodic signals in the data. Although this model is unphysical, it gave us a direction for searching for the correct model.

The short periodic signal can be caused by the "xarallap" effect - orbital motion of a binary *source* star. Since there are finite source effects in the observed data, we used a single lens magnification map and the GPU track extraction method for searching the correct model. Although we can estimate the period of the binary source stars, the other parameters are not very well constrained. The GPU track extraction method is able to perform light curve generation in a very high performance. We perform a 100,000 chain MCMC optimization to make sure that we have covered all the local minima. Figures 6.19 and 6.20 show the best-fitting model. The xarallap effect alone is able to produce an amplification profile that is similar to a planetary profile.

A grid search in triple or more lenses system is very computationally expensive even with the GPU magnification map grid search approach. At this stage, we only randomly select parameters from the lens system in searching for a best-fitting model. A systematic approach may be developed in future work which is discussed in Chapter 7.

6.4 Exclusion Region mapping

The magnification map modelling technique discussed in Chapter 4 is not only very efficient for performing a grid search as in microlensing modelling, it is also highly suitable for generating exclusion regions for microlensing events. The computation involved in exclusion region generation is very similar to using the magnification map technique for grid searches as a similar strategy can be used. The exclusion region represents the sensitivity of a microlensing event to various masses of planetary companion.

In order to generate an exclusion region map, we first fit a single lens model to an event. Then we place planets with different masses around the lens star at various distances and angles that are represented by ε, d and ϕ. We search for the best-fitting model with each combination of $\{\varepsilon,\ d,\ \phi\}$. If the difference in χ^2 is greater than 60 compared with the single lens model, this model with $\{\varepsilon,\ d,\ \phi\}$ is excluded which means we are able to detect the deviation signal if there is a planet with mass fraction ε, projected separation d and positioned at angle ϕ.

6.4.1 MOA2007-BLG-397

This is a high magnification event discovered in 2007 with good data coverage of the peak by MOA. A single lens model with finite source effect fit is able to describe this event and there is no extra deviation observed within the data covered light curve. Figure 6.21 shows the data obtained by MOA. There is a very good coverage over the peak of this event and gaps at the shoulder of the light curve due to New Zealand day time.

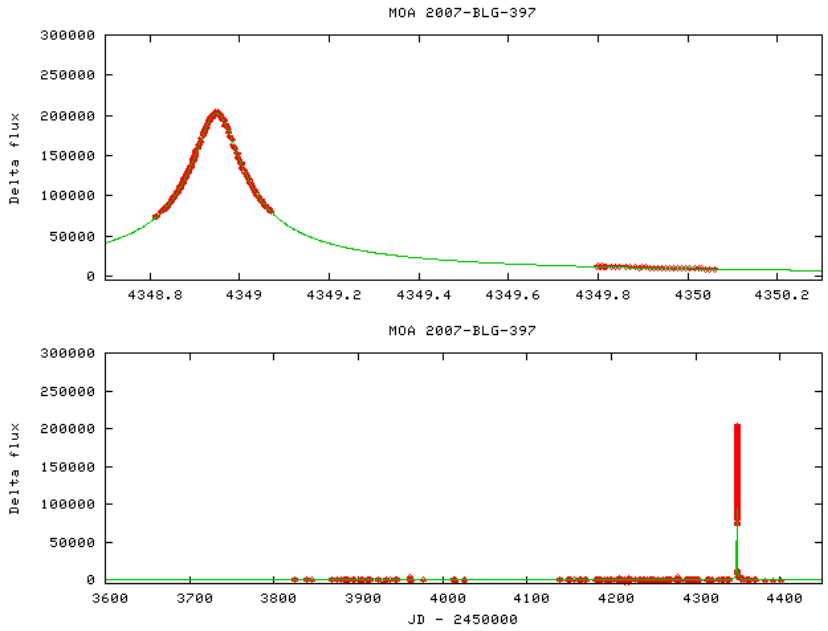

Figure 6.21: Light curve obtained by MOA of the event MOA2007-BLG-397 together with the best-fitting point-source single lens model.

Figure 6.22 shows the exclusion region of this event. The green region represents the exclusion region with $\varepsilon = 10^{-5}$ (Earth-mass planet), the red region represents the exclusion region with $\varepsilon = 10^{-4}$ (Neptune-mass planet) and the blue region represents the exclusion region with $\varepsilon = 0.003$ (Jupiter-mass planet). The exclusion region map shows the planetary detection sensitivity of this event. Therefore if there is a planet around the lens star with the above masses within the coloured region, we are able to detect the deviation signal.

The blue region which represents the sensitivity in detecting a Jupiter-mass planet in Figure 6.22 is much larger than the red and green region which represents the detectability of smaller planets. The blue region extents to projected separation 4.0 in Einstein units. Therefore, there is a very high chance that this event is able to detect a planet with Jupiter mass if it exists. The shape of the blue region is affected by the coverage of the observed

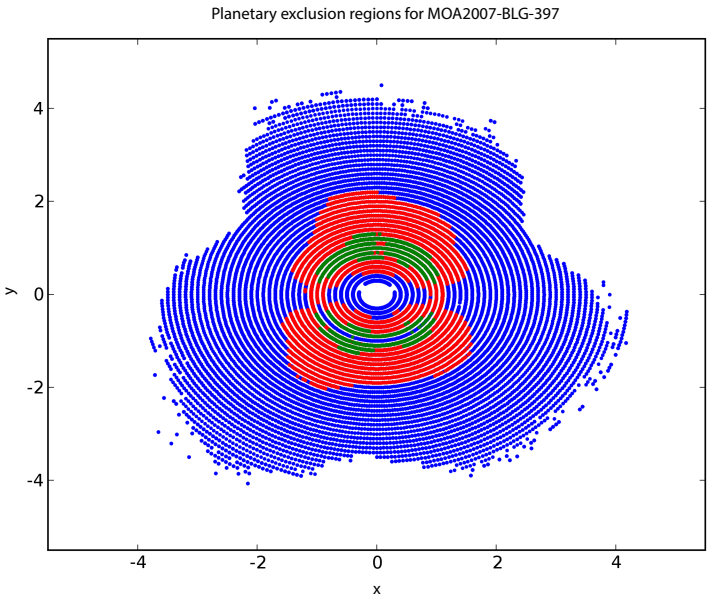

Figure 6.22: Planetary exclusion regions for MOA2007-BLG-397. A dot at a given x and y position projected on the source plane, means that a planet at that position is excluded at a confidence level of $\Delta\chi^2 = 60$. The green dots correspond to a planet: star mass fraction of $\varepsilon = 10^{-5}$, red correspond to $\varepsilon = 10^{-4}$, and blue corresponds to $\varepsilon = 0.003$. Both axes are in Einstein units.

data. Since there are gaps within the light curve, the planetary signals of certain caustic configurations at far distance may fall into the gap of the data.

The red and green region in Figure 6.22 represents the sensitivity in detecting Neptune-mass and Earth-mass planets respectively. They are much smaller compared with the blue region, especially the green region for detectability of a Earth-mass planet. Since the caustic of low-mass planets is very small compared with large-mass planets, the sensitivity in detecting the low-mass planets is also lower. A high magnification event is often required to detect low mass planets as the deviation signal is often more significant in high magnification events. Similarly to the blue region, the shape of the red and green region is affected by the coverage of the observed data. The gap between the green region (representing the sensitivity in detecting an Earth-mass planet) shows the planetary deviation signals fall into the data gap when the caustic is angled at around 90 and 270 degrees.

6.4.2 MOA2007-BLG-312

This is also a high magnification event discovered in 2007 and with good coverage at the peak of the light curve by MOA. The observed data is fitted well by a single lens model. Figure 6.23 shows data obtained by MOA which has a good coverage over the peak of the event and is fitted nicely by a single lens model. But there are also gaps at the shoulder of the light curve which may affect the detectability of some planetary systems with certain masses, projected separation and angle.

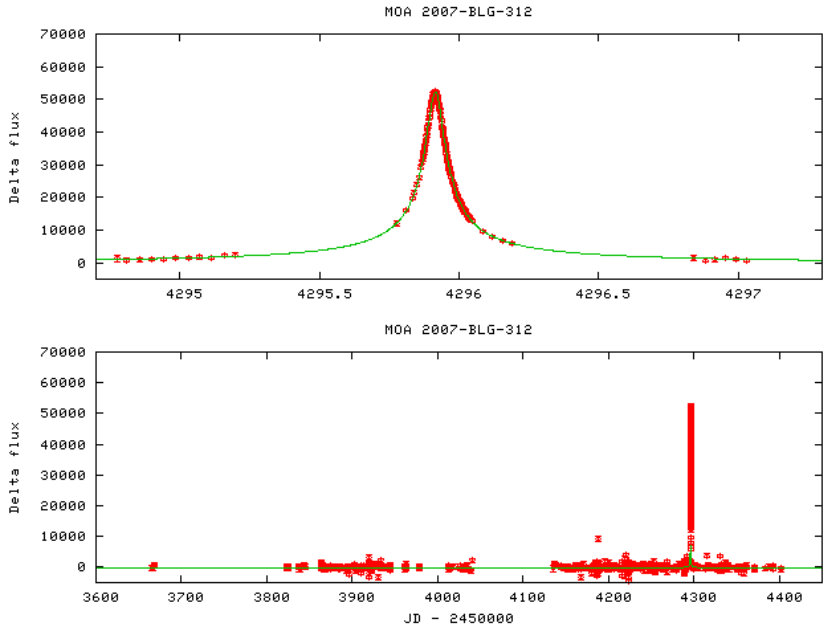

Figure 6.23: Light curve obtained by MOA of the event MOA2007-BLG-312 together with the best-fitting point-source single lens model.

Similarly to the previous event, we generate an exclusion region of this event by the magnification map GPU engine. We place a planet with a certain mass fraction around the lens star at various distances. If the χ^2 of the best fitted model with $\{\varepsilon,\ d,\ \phi\}$ is larger than 60 compared with the single lens model, the planetary system with mass fraction ε, projected separation d and angle ϕ is detectable by this microlensing event.

Figure 6.24 shows the exclusion region of the microlensing event MOA2007-BLG-312 of three different mass fractions. The blue region represents the sensitivity of a Jupiter-mass planet with mass fraction $\varepsilon = 0.003$. The detectable zone is large and extents to 3-4 Einstein units in projected separation. Therefore, if there is a Jupiter-size planet, it is highly likely we are able to make a detection. Compared with the blue region in the previous discussed

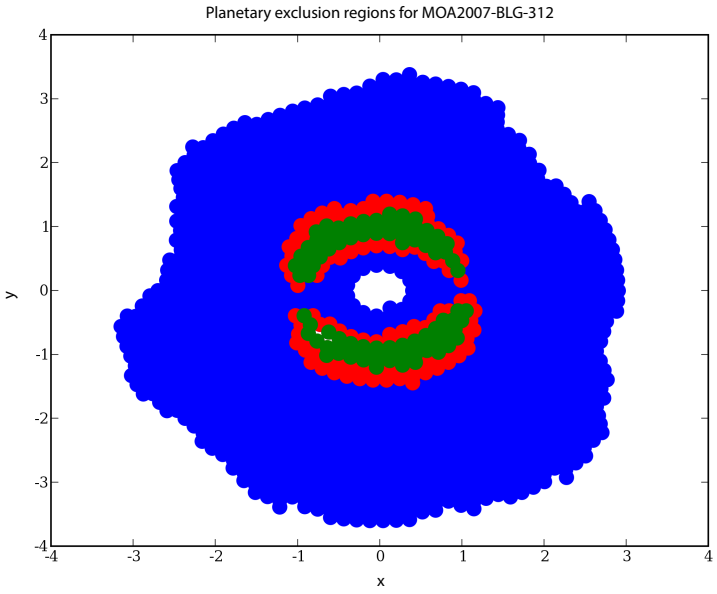

Figure 6.24: Planetary exclusion regions for MOA2007-BLG-312. A dot at a given x and y position projected on the source plane, means that a planet at that position is excluded at a confidence level of $\Delta\chi^2 = 60$. The green dots correspond to a planet: star mass fraction of $\varepsilon = 10^{-5}$, red correspond to $\varepsilon = 10^{-4}$, and blue corresponds to $\varepsilon = 0.003$. Both axes are in Einstein units.

event MOA2007-BLG-397 in Figure 6.22, this event's blue region is in a more rounded shape due to it having better data coverage over the peak. The shape of the exclusion region is dependent on the data coverage of a microlensing event. The missing part of the blue region on the top left-hand side in Figure 6.24 corresponded to the data gap on the left shoulder of the light curve in Figure 6.23.

The red region and green region represent the sensitivity of Neptune-mass planets with $\varepsilon = 10^{-4}$ and Earth-mass planets with $\varepsilon = 10^{-5}$ respectively. The detectable zone of the red and green region in Figure 6.24 is much smaller than the blue region due to the deviation signal of a small planet being much smaller. The gap in the red and green region on the x-axis is due to the planetary signal occurring in the gap of the observed data at the shoulder of the light curve which is shown in Figure 6.23. The caustic of the planetary system with low mass fraction is very small compared with the planetary system with large mass fraction. Moreover, the caustic gets even smaller when the projected separation d is larger than 1 due to the separation of the primary caustic and the secondary caustic. Therefore, the planet

detectable zone of red and green (small planets) is much smaller than the blue region which represents the detectable zone of a big planet.

6.5 Conclusion

In this chapter, we have discussed using the magnification map technique and the dynamic light curve engine described in Chapter 4 and Chapter 5 respectively to model a range of microlensing events consisting of different challenges. Both the magnification map generation/track extraction and dynamic light curve engine as performed on the NVIDIA GPU provide significant speed up over only using the CPU. The speed up in performance shortens the computation time from months to hours. It does not only provide a very cost effective solution for microlensing modelling, but also allows us to perform wider range and more complex model searches.

Although the GPU based modelling method provide significant speed up advantage when perform microlensing modelling, a modelling strategy is required as it is not feasible to perform a brute force search in all parameter space. The first section in this chapter discussed a combination of optimization technique and strategies on different types of microlensing modelling challenges. The events showcase in this chapter display the flexibility and power of our modelling technique and strategies. A large part of our success in modelling strategies is due to the speed up provided by our GPU-based modelling engine. Not only can we perform our search faster, but it also allows us to combine different optimization strategies to cover wider parameter space and still maintain high performance.

The last chapter discusses how the GPU changes the way we do microlensing modelling as well as the future in high performance microlensing modelling.

Chapter 7

Conclusion and Discussion

Microlensing is an elegant and very powerful tool for discovering extra-solar planets in our galaxy. It has the sensitivity to discover Earth-mass planets and also planets that have very wide orbits. This is a significant advantage over other exo-planet searching techniques. But it also presents a challenge as microlensing modelling is very computationally expensive and is usually performed on a cluster computer.

7.1 GPU solution for microlensing modelling

We have developed a GPU solution for the microlensing modelling challenge that provides a very cost-effective solution to this problem. The GPU-based solution is able to significantly speed up the microlensing modelling process from months to only days. We are even able to perform a microlensing modelling in just hours. The calculation which used to be performed on a cluster now can be computed on just a desktop computer with GPUs. The ownership and running cost of a desktop computer with GPUs is significantly lower than a cluster computer.

7.1.1 Magnification map generation and light curve extraction

One of the major challenges in microlensing modelling is the calculation of the finite-source effect. The magnification map technique provides an effective way to compute the finite-source effect as multiple light curves with finite sources can be generated by using the same

shot rays collected on the magnification map. However, the magnification map generation is computationally expensive as millions of rays are computed by the lens equations for each map.

We have provided a GPU solution to the magnification map generation as well as light curve extraction from the magnification map in Chapter 4. We are able to speed up the generation of magnification maps from minutes to seconds and also significantly speed up the light curve extraction from the magnification map using the GPU. This allows us to perform a grid search very effectively for a microlensing event in multi-dimensional parameter space. We are able to discover all the minima on the chi-square surface in just hours. This process used to take months and is now only hours by using our methods.

7.1.2 Dynamic light curve engine for complex microlensing events

Another major challenge in microlensing modelling is the computation of orbital motion together with finite source calculation. This presents a major computational challenge as every amplification calculation needs to be computed individually and no shot rays can be reused. The reuse of computed rays is the reason that we are able to compute the finite-source effect effectively.

We have developed a hybrid method using both CPU and GPU for microlensing events with orbital motion and finite-source calculation. Different parts of microlensing modelling calculations are assigned to CPU and GPU according to their strength. We use the CPU to solve the complex polynomial of the lens equation as well as dynamically searching the image area. The GPU is used to perform inverse ray shooting in highly parallel fashion. We also further increase parallelism by performing the computation on multiple CPU cores as well as overlapping the computation of CPU and GPU.

7.1.3 Modelling strategies

The challenge in microlensing modelling is not only searching for the right model that matches with the observed data, but also proving there are no other models in the multi-dimensional parameter space with a better fit. We have discussed different modelling strategies for different type of microlensing events in Chapter 6. We also showcase the modelling of different types of microlensing events using the described strategies.

Chapter 6 provides examples in modelling different types of microlensing events and also demonstrates the effectiveness of our modelling strategies together with our developed GPU based modelling code. The type of events showcased in Chapter 6 include binary events to planetary events, from real time modelling to complex microlensing modelling.

The reasons for the effectiveness of our modelling strategies is due to the high performance provided by the GPU modelling code. The improvement in performance over just using the CPU for computation provides an invaluable flexibility for us to incorporate different types of modelling strategies in our modelling procedure. It also allows us to test new ideas easier as the high performance in light curve generation by our GPU-based code provides faster feedback, thus encouraging us to experiment with different ideas and develop new modelling strategies more effectively.

7.2 Future work

The demand on microlensing modelling is going to get stronger as more telescopes dedicated for microlensing observation are being built. There are great opportunities to extend this research for more contributions to the field. This section discuss some ideas for future work.

7.2.1 Real-time automatic microlensing modelling system

With the GPU-based microlensing modelling computation methods and modelling strategies developed in this thesis, it is natural to extend our work to build a fully or semi automated modelling system. The automated system will download the latest observation data and search for the current best model for multiple microlensing events automatically. Since it is critical to make predictions for some events for allocating appropriate telescope observation resources, a real-time automated modelling system will be very beneficial for the microlensing community.

We can develop a semi or fully automated modelling system using the modelling strategies discussed in Chapter 6 and the GPU-based computation methods discussed in Chapter 4 and 5. An automated real-time modelling system is able to search and update the best model as more observation data are collected. It should be possible to dynamically switch between modelling jobs depending on the urgency of the event. Therefore, a database for storing modelling results is needed. The database should be in some form of standard and be

machine readable, this allows the global computing resources for microlensing modelling to be shared effectively. In microlensing modelling, there are always more jobs than computing resources available, thus a standard machine readable way for sharing modelling results is highly beneficial for the microlensing community as well.

7.2.2 GPU mini cluster

In the past, an automated microlensing modelling system has been very expensive to build as a large cluster computer is required to have enough processing power for the task. It costs hundred thousands of dollars, not to mention the cost of running such a large cluster, cooling, maintenance, power consumption, etc. But a equally powerful GPU based mini cluster only costs a few thousands dollars. It can also fit inside any office room easily with minimal cooling and such a system is very efficient in terms of power to performance ratio. Moreover, a cluster computer is usually a shared resource as it is very expensive to build and own, while a GPU mini cluster can be one's dedicated computing resource as it is relatively cheap to build. A dedicated computing resource is important for microlensing modelling as some tasks are time critical.

7.2.3 Multi planetary system modelling

There will be more telescopes designed and delicated to microlensing observation in the next few years. This allows us to have better coverage of microlensing events and also have better quality and higher precision observation data. We may observe more multi planetary microlensing events, for example, triple lens or quadruple lens systems. As we discussed in Chapter 6, a multi planetary system takes significantly more computing resources to model as the parameter space is much larger. To build a computing facility consisting of only CPUs for such a modelling purpose is very expensive, but is much more manageable for a GPU cluster. The cost to scale the computing resources of a GPU-based system is relativity small as thousands of dollars spent on GPU systems has the equivalent computing power of a cluster costing hundred of thousands of dollars.

A more systematic approach for searching for a three or more lenses system is also required to reduce the time in searching the parameter space of the lens system. One may consider putting physical knowledge of the likelihood of different multi-lens configuration into the modelling system to avoid searching in unphysical parameter space and thus reduce the

computation time. This may require developing a system to select parameters statistically using the existing knowledge of multi-lens system.

7.2.4 Parallel computing in the near future for microlensing modelling

Since the computing hardware is moving from single CPU core to multiple CPU cores as well as the popular uses of GPUs, parallel computing is essential in scientific computing in the near future. Most part of the microlensing modelling calculations are suitable to compute in parallel. The development cost for parallel programming code may be higher than sequential code, but the benefit is significant and outweighs the cost. Furthermore, the cost of parallel code development should become cheaper over time as more smart tools and education are put into parallel programming.

The development in parallel computing hardwares and software tools are just started and they may change very quickly in the near future. NVIDIA CUDA is designed to be backwards compatible, so older code is still able to run on the latest hardwares with minimal modifications. But older code usually cannot utilize the full power of the latest hardwares as new instructions and architecture are introduced. One can only achieve the maximum performance by using the latest instructions and designing the program to fit well with the architecture. There are other frameworks like OpenMP aimed to generalize the parallel programming language and able to work on any parallel computing capable hardwares, but they still cannot utilize the full power of a specific hardware at this stage.

There are also more parallel computing hardwares being introduced. One of them is Intel's Xeon Phi [74] which will be on the market in 2013. It is not yet released to the market at the point of this thesis being written. From the early announced specifications, it consists of 60 processors with vector engines capable of running 240 threads. Unlike GPU, each core has it's own cache system and is able to perform branching without affecting the performance of other cores. This is very different from GPU as divergent branching degrade GPU performance significantly. The hybrid methods used in the dynamic light curve engine discussed in Chapter 5 which use both CPU and GPU to perform different parts of the modelling are developed is to avoid divergent branching in the GPU computation. Therefore, the Xeon Phi processor should be highly suitable for the dynamic light curve engine while the magnification map technique that requires raw processing power is highly suitable to perform on the GPU.

Beside the introduction of Intel Xeon Phi, NVIDIA also introduced the new Kepler GK110 architecture GPU [75]. It provides over 1TFlop of double precision throughput and is able to deliver up to 3× the performance per watt of Fermi. It also has some new important features like "Dynamic Parallelism" which allow a GPU thread to create new threads during run time without involving the CPU. Dynamic Parallelism allows a wider range of algorithms to be able to run on the GPU and it is a very useful feature for implementing the dynamic light curve engine for running entirely on the GPU. Other feature like "Hyper-Q" allows multiple CPU cores to launch a kernel on a single GPU simultaneously. This can greatly improve the utilization of the GPU for some problems. Both features together may allow multiple dynamic light curve engines to run simultaneously in high performance.

The development trend of the new generation of modern processors seems to be the mixture of both conventional CPU and GPU. While the CPU is adding more "GPU like" features like vector engines for parallel processing, GPU is also become more flexible to program like the CPU. The development in parallel computing is getting more rapid in recent years and this provides an exciting future for the microlensing modelling computing challenge.

Bibliography

[1] A. Einstein, "Lens-Like Action of a Star by the Deviation of Light in the Gravitational Field," *Science*, vol. 84, pp. 506–507, Dec. 1936.

[2] S. Liebes, "Gravitational Lenses," *Physical Review*, vol. 133, pp. 835–844, Feb. 1964.

[3] B. Paczynski, "Gravitational microlensing by the galactic halo," *The Astrophysical Journal*, vol. 304, pp. 1–5, May 1986.

[4] S. Mao and B. Paczynski, "Gravitational microlensing by double stars and planetary systems," *The Astrophysical Journall*, vol. 374, pp. L37–L40, Jun. 1991.

[5] A. Gould and A. Loeb, "Discovering planetary systems through gravitational microlenses," *The Astrophysical Journal*, vol. 396, pp. 104–114, Sep. 1992.

[6] A. D. Bolatto and E. E. Falco, "The detectability of planetary companions of compact Galactic objects from their effects on microlensed light curves of distant stars," *The Astrophysical Journal*, vol. 436, pp. 112–116, Nov. 1994.

[7] D. P. Bennett and S. H. Rhie, "Detecting Earth-Mass Planets with Gravitational Microlensing," *The Astrophysical Journal*, vol. 472, p. 660, Nov. 1996.

[8] I. A. Bond, "The first extrasolar planet detected via gravitational microlensing," *New Astronomy Reviews*, vol. 56, pp. 25–32, Jan. 2012.

[9] I. A. Bond, A. Udalski, M. Jaroszyński, N. J. Rattenbury, B. Paczyński, I. Soszyński, L. Wyrzykowski, M. K. Szymański, M. Kubiak, O. Szewczyk, K. Żebruń, G. Pietrzyński, F. Abe, D. P. Bennett, S. Eguchi, Y. Furuta, J. B. Hearnshaw, K. Kamiya, P. M. Kilmartin, Y. Kurata, K. Masuda, Y. Matsubara, Y. Muraki, S. Noda, K. Okajima, T. Sako, T. Sekiguchi, D. J. Sullivan, T. Sumi, P. J. Tristram, T. Yanagisawa, P. C. M. Yock, and OGLE Collaboration, "OGLE 2003-BLG-235/MOA 2003-BLG-53: A Planetary Microlensing Event," *The Astrophysical Journal Letters*, vol. 606, pp. L155–L158, May 2004.

[10] C. Han, A. Udalski, J.-Y. Choi, J. C. Yee, A. Gould, G. Christie, T.-G. Tan, M. K. Szymański, M. Kubiak, I. Soszyński, G. Pietrzyński, R. Poleski, K. Ulaczyk, P. Pietrukowicz, S. Kozłowski, J. Skowron, Ł. Wyrzykowski, OGLE Collaboration, L. A. Almeida, V. Batista, D. L. Depoy, S. Dong, J. Drummond, B. S. Gaudi, K.-H. Hwang, F. Jablonski, Y.-K. Jung, C.-U. Lee, J.-R. Koo, J. McCormick, L. A. G. Monard, T. Natusch, H. Ngan, H. Park, R. W. Pogge, I. Porritt, I.-G. Shin, and μFUN Collaboration, "The Second Multiple-planet System Discovered by Microlensing: OGLE-2012-BLG-0026Lb, c–A Pair of Jovian Planets beyond the Snow Line," *The Astrophysical Journal Letters*, vol. 762, p. L28, Jan. 2013.

[11] D. P. Bennett, T. Sumi, I. A. Bond, K. Kamiya, F. Abe, C. S. Botzler, A. Fukui, K. Furusawa, Y. Itow, A. V. Korpela, P. M. Kilmartin, C. H. Ling, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, K. Ohnishi, N. J. Rattenbury, T. Saito, D. J. Sullivan, D. Suzuki, W. L. Sweatman, P. J. Tristram, K. Wada, P. C. M. Yock, and The MOA Collaboration, "Planetary and Other Short Binary Microlensing Events from the MOA Short-event Analysis," *The Astrophysical Journal*, vol. 757, p. 119, Oct. 2012.

[12] E. Bachelet, I.-G. Shin, C. Han, P. Fouqué, A. Gould, J. W. Menzies, J.-P. Beaulieu, D. P. Bennett, I. A. Bond, S. Dong, D. Heyrovský, J.-B. Marquette, J. Marshall, J. Skowron, R. A. Street, T. Sumi, A. Udalski, L. Abe, K. Agabi, M. D. Albrow, W. Allen, E. Bertin, M. Bos, D. M. Bramich, J. Chavez, G. W. Christie, A. A. Cole, N. Crouzet, S. Dieters, M. Dominik, J. Drummond, J. Greenhill, T. Guillot, C. B. Henderson, F. V. Hessman, K. Horne, M. Hundertmark, J. A. Johnson, U. G. Jørgensen, R. Kandori, C. Liebig, D. Mékarnia, J. McCormick, D. Moorhouse, T. Nagayama, D. Nataf, T. Natusch, S. Nishiyama, J.-P. Rivet, K. C. Sahu, Y. Shvartzvald, G. Thornley, A. R. Tomczak, Y. Tsapras, J. C. Yee, V. Batista, C. S. Bennett, S. Brillant, J. A. R. Caldwell, A. Cassan, E. Corrales, C. Coutures, D. Dominis Prester, J. Donatowicz, D. Kubas, R. Martin, A. Williams, M. Zub, The PLANET Collaboration, L. A. de Almeida, D. L. DePoy, B. S. Gaudi, L.-W. Hung, F. Jablonski, S. Kaspi, N. Klein, C.-U. Lee, Y. Lee, J.-R. Koo, D. Maoz, J. A. Muñoz, R. W. Pogge, D. Polishook, A. Shporer, T. F. μCollaboration, F. Abe, C. S. Botzler, P. Chote, M. Freeman, A. Fukui, K. Furusawa, P. Harris, Y. Itow, S. Kobara, C. H. Ling, K. Masuda, Y. Matsubara, N. Miyake, K. Ohmori, K. Ohnishi, N. J. Rattenbury, T. Saito, D. J. Sullivan, D. Suzuki, W. L. Sweatman, P. J. Tristram, K. Wada, P. C. M. Yock, The MOA Collaboration, M. K. Szymański, I. Soszyński, M. Kubiak, R. Poleski, K. Ulaczyk, G. Pietrzyński, Ł. Wyrzykowski, The OGLE Collaboration, N. Kains, C. Snodgrass, I. A. Steele, The RoboNet Collaboration, K. A. Alsubai, V. Bozza, P. Browne, M. J. Burgdorf, S. Calchi Novati, P. Dodds, S. Dreizler, F. Finet, T. Gerner, S. Hardis, K. Harpsøe, T. C. Hinse, E. Kerins, L. Mancini, M. Mathiasen, M. T. Penny, S. Proft, S. Rahvar, D. Ricci, G. Scarpetta, S. Schäfer, F. Schönebeck, J. Southworth, J. Surdej, J. Wambsganss, and T. MiNDSTEp Consortium, "MOA 2010-BLG-477Lb: Constraining the Mass of a Microlensing Planet from Microlensing Parallax, Orbital Motion, and Detection of Blended Light," *The Astrophysical Journal*, vol. 754, p. 73, Jul. 2012.

[13] Y. Muraki, C. Han, D. P. Bennett, D. Suzuki, L. A. G. Monard, R. Street, U. G. Jorgensen, P. Kundurthy, J. Skowron, A. C. Becker, M. D. Albrow, P. Fouqué, D. Heyrovský, R. K. Barry, J.-P. Beaulieu, D. D. Wellnitz, I. A. Bond, T. Sumi, S. Dong, B. S. Gaudi, D. M. Bramich, M. Dominik, F. Abe, C. S. Botzler, M. Freeman, A. Fukui, K. Furusawa, F. Hayashi, J. B. Hearnshaw, S. Hosaka, Y. Itow, K. Kamiya, A. V. Korpela, P. M. Kilmartin, W. Lin, C. H. Ling, S. Makita, K. Masuda, Y. Matsubara, N. Miyake, K. Nishimoto, K. Ohnishi, Y. C. Perrott, N. J. Rattenbury, T. Saito, L. Skuljan, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, K. Wada, P. C. M. Yock, MOA Collaboration, G. W. Christie, D. L. DePoy, E. Gorbikov, A. Gould, S. Kaspi, C.-U. Lee, F. Mallia, D. Maoz, J. McCormick, D. Moorhouse, T. Natusch, B.-G. Park, R. W. Pogge, D. Polishook, A. Shporer, G. Thornley, J. C. Yee, μFUN Collaboration, A. Allan, P. Browne, K. Horne, N. Kains, C. Snodgrass, I. Steele, Y. Tsapras, RoboNet Collaboration, V. Batista, C. S. Bennett, S. Brillant, J. A. R. Caldwell, A. Cassan, A. Cole, R. Corrales, C. Coutures, S. Dieters, D. Dominis Prester, J. Donatowicz, J. Greenhill, D. Kubas, J.-B. Marquette, R. Martin, J. Menzies, K. C. Sahu, I. Waldman, A. Williams, M. Zub, PLANET Collaboration, H. Bourhrous, Y. Matsuoka, T. Nagayama, N. Oi, Z. Randriamanakoto, IRSF Observers, V. Bozza, M. J. Burgdorf, S. Calchi Novati, S. Dreizler, F. Finet, M. Glitrup, K. Harpsøe, T. C. Hinse, M. Hundertmark, C. Liebig, G. Maier, L. Mancini, M. Mathiasen, S. Rahvar, D. Ricci, G. Scar-

petta, J. Skottfelt, J. Surdej, J. Southworth, J. Wambsganss, F. Zimmer, MiNDSTEp Consortium, A. Udalski, R. Poleski, Ł. Wyrzykowski, K. Ulaczyk, M. K. Szymański, M. Kubiak, G. Pietrzyński, I. Soszyński, and OGLE Collaboration, "Discovery and Mass Measurements of a Cold, 10 Earth Mass Planet and Its Host Star," *The Astrophysical Journal*, vol. 741, p. 22, Nov. 2011.

[14] V. Batista, A. Gould, S. Dieters, S. Dong, I. Bond, J. P. Beaulieu, D. Maoz, B. Monard, G. W. Christie, J. McCormick, M. D. Albrow, K. Horne, Y. Tsapras, M. J. Burgdorf, S. Calchi Novati, J. Skottfelt, J. Caldwell, S. Kozłowski, D. Kubas, B. S. Gaudi, C. Han, D. P. Bennett, J. An, MOA Collaboration, F. Abe, C. S. Botzler, D. Douchin, M. Freeman, A. Fukui, K. Furusawa, J. B. Hearnshaw, S. Hosaka, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, B. S. Makita, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, M. Nagaya, K. Nishimoto, K. Ohnishi, T. Okumura, Y. C. Perrott, N. Rattenbury, T. Saito, D. J. Sullivan, T. Sumi, W. L. Sweatman, P. J. Tristram, E. von Seggern, P. C. M. Yock, PLANET Collaboration, S. Brillant, J. J. Calitz, A. Cassan, A. Cole, K. Cook, C. Coutures, D. Dominis Prester, J. Donatowicz, J. Greenhill, M. Hoffman, F. Jablonski, S. R. Kane, N. Kains, J.-B. Marquette, R. Martin, E. Martioli, P. Meintjes, J. Menzies, E. Pedretti, K. Pollard, K. C. Sahu, C. Vinter, J. Wambsganss, R. Watson, A. Williams, M. Zub, FUN Collaboration, W. Allen, G. Bolt, M. Bos, D. L. DePoy, J. Drummond, J. D. Eastman, A. Gal-Yam, E. Gorbikov, D. Higgins, J. Janczak, S. Kaspi, C.-U. Lee, F. Mallia, A. Maury, L. A. G. Monard, D. Moorhouse, N. Morgan, T. Natusch, E. O. Ofek, B.-G. Park, R. W. Pogge, D. Polishook, R. Santallo, A. Shporer, O. Spector, G. Thornley, J. C. Yee, MiNDSTEp Consortium, V. Bozza, P. Browne, M. Dominik, S. Dreizler, F. Finet, M. Glitrup, F. Grundahl, K. Harpsøe, F. V. Hessman, T. C. Hinse, M. Hundertmark, U. G. Jørgensen, C. Liebig, G. Maier, L. Mancini, M. Mathiasen, S. Rahvar, D. Ricci, G. Scarpetta, J. Southworth, J. Surdej, F. Zimmer, RoboNet Collaboration, A. Allan, D. M. Bramich, C. Snodgrass, I. A. Steele, and R. A. Street, "MOA-2009-BLG-387Lb: a massive planet orbiting an M dwarf," *Astronomy and Astrophysics*, vol. 529, p. A102, May 2011.

[15] N. Miyake, T. Sumi, S. Dong, R. Street, L. Mancini, A. Gould, D. P. Bennett, Y. Tsapras, J. C. Yee, M. D. Albrow, I. A. Bond, P. Fouqué, P. Browne, C. Han, C. Snodgrass, F. Finet, K. Furusawa, K. Harpsøe, W. Allen, M. Hundertmark, M. Freeman, D. Suzuki, F. Abe, C. S. Botzler, D. Douchin, A. Fukui, F. Hayashi, J. B. Hearnshaw, S. Hosaka, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, S. Makita, K. Masuda, Y. Matsubara, Y. Muraki, T. Nagayama, K. Nishimoto, K. Ohnishi, Y. C. Perrott, N. Rattenbury, T. Saito, L. Skuljan, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, K. Wada, P. C. M. Yock, MOA Collaboration, G. Bolt, M. Bos, G. W. Christie, D. L. DePoy, J. Drummond, A. Gal-Yam, B. S. Gaudi, E. Gorbikov, D. Higgins, K.-H. Hwang, J. Janczak, S. Kaspi, C.-U. Lee, J.-R. Koo, S. Kozłowski, Y. Lee, F. Mallia, A. Maury, D. Maoz, J. McCormick, L. A. G. Monard, D. Moorhouse, J. A. Muñoz, T. Natusch, E. O. Ofek, R. W. Pogge, D. Polishook, R. Santallo, A. Shporer, O. Spector, G. Thornley, μFUN Collaboration, A. Allan, D. M. Bramich, K. Horne, N. Kains, I. Steele, RoboNet Collaboration, V. Bozza, M. J. Burgdorf, S. Calchi Novati, M. Dominik, S. Dreizler, M. Glitrup, F. V. Hessman, T. C. Hinse, U. G. Jørgensen, C. Liebig, G. Maier, M. Mathiasen, S. Rahvar, D. Ricci, G. Scarpetta, J. Skottfelt, J. Southworth, J. Surdej, J. Wambsganss, F. Zimmer, MiNDSTEp Consortium, V. Batista, J. P. Beaulieu, S. Brillant, A. Cassan, A. Cole, E. Corrales, C. Coutures, S. Dieters, J. Greenhill, D. Kubas, J. Menzies, and PLANET Collaboration, "A Sub-Saturn Mass Planet, MOA-2009-BLG-319Lb," *The Astrophysical Journal*, vol. 728, p. 120, Feb. 2011.

[16] J. Janczak, A. Fukui, S. Dong, L. A. G. Monard, S. Kozłowski, A. Gould, J. P. Beaulieu, D. Kubas, J. B. Marquette, T. Sumi, I. A. Bond, D. P. Bennett, F. Abe, K. Furusawa, J. B. Hearnshaw, S. Hosaka, Y. Itow, K. Kamiya, A. V. Korpela, P. M. Kilmartin, W. Lin, C. H. Ling, S. Makita, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, M. Nagaya, T. Nagayama, K. Nishimoto, K. Ohnishi, Y. C. Perrott, N. J. Rattenbury, T. Sako, T. Saito, L. Skuljan, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, P. C. M. Yock, MOA Collaboration, J. H. An, G. W. Christie, S.-J. Chung, D. L. DePoy, B. S. Gaudi, C. Han, C.-U. Lee, F. Mallia, T. Natusch, B.-G. Park, R. W. Pogge, μFUN Collaboration, T. Anguita, S. Calchi Novati, M. Dominik, U. G. Jørgensen, G. Masi, M. Mathiasen, MiNDSTEp Collaboration, V. Batista, S. Brillant, A. Cassan, A. Cole, E. Corrales, C. Coutures, S. Dieters, P. Fouqué, J. Greenhill, and PLANET Collaboration, "Sub-Saturn Planet MOA-2008-BLG-310Lb: Likely to be in the Galactic Bulge," *The Astrophysical Journal*, vol. 711, pp. 731–743, Mar. 2010.

[17] T. Sumi, D. P. Bennett, I. A. Bond, A. Udalski, V. Batista, M. Dominik, P. Fouqué, D. Kubas, A. Gould, B. Macintosh, K. Cook, S. Dong, L. Skuljan, A. Cassan, F. Abe, C. S. Botzler, A. Fukui, K. Furusawa, J. B. Hearnshaw, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, M. Nagaya, T. Nagayama, K. Ohnishi, T. Okumura, Y. C. Perrott, N. Rattenbury, T. Saito, T. Sako, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, P. C. M. Yock, MOA Collaboration, J. P. Beaulieu, A. Cole, C. Coutures, M. F. Duran, J. Greenhill, F. Jablonski, U. Marboeuf, E. Martioli, E. Pedretti, O. Pejcha, P. Rojo, M. D. Albrow, S. Brillant, M. Bode, D. M. Bramich, M. J. Burgdorf, J. A. R. Caldwell, H. Calitz, E. Corrales, S. Dieters, D. Dominis Prester, J. Donatowicz, K. Hill, M. Hoffman, K. Horne, U. G. Jørgensen, N. Kains, S. Kane, J. B. Marquette, R. Martin, P. Meintjes, J. Menzies, K. R. Pollard, K. C. Sahu, C. Snodgrass, I. Steele, R. Street, Y. Tsapras, J. Wambsganss, A. Williams, M. Zub, PLANET Collaboration, M. K. Szymański, M. Kubiak, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, OGLE Collaboration, W. Allen, G. W. Christie, D. L. DePoy, B. S. Gaudi, C. Han, J. Janczak, C.-U. Lee, J. McCormick, F. Mallia, B. Monard, T. Natusch, B.-G. Park, R. W. Pogge, R. Santallo, and μFUN Collaboration, "A Cold Neptune-Mass Planet OGLE-2007-BLG-368Lb: Cold Neptunes Are Common," *The Astrophysical Journal*, vol. 710, pp. 1641–1653, Feb. 2010.

[18] S. Dong, I. A. Bond, A. Gould, S. Kozłowski, N. Miyake, B. S. Gaudi, D. P. Bennett, F. Abe, A. C. Gilmore, A. Fukui, K. Furusawa, J. B. Hearnshaw, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, K. Masuda, Y. Matsubara, Y. Muraki, M. Nagaya, K. Ohnishi, T. Okumura, Y. C. Perrott, N. Rattenbury, T. Saito, T. Sako, S. Sato, L. Skuljan, D. J. Sullivan, T. Sumi, W. Sweatman, P. J. Tristram, P. C. M. Yock, MOA Collaboration, G. Bolt, G. W. Christie, D. L. DePoy, C. Han, J. Janczak, C.-U. Lee, F. Mallia, J. McCormick, B. Monard, A. Maury, T. Natusch, B.-G. Park, R. W. Pogge, R. Santallo, K. Z. Stanek, μFUN Collaboration, A. Udalski, M. Kubiak, M. K. Szymański, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, and OGLE Collaboration, "Microlensing Event MOA-2007-BLG-400: Exhuming the Buried Signature of a Cool, Jovian-Mass Planet," *The Astrophysical Journal*, vol. 698, pp. 1826–1837, Jun. 2009.

[19] D. Kubas, J. P. Beaulieu, D. P. Bennett, A. Cassan, A. Cole, J. Lunine, J. B. Marquette, S. Dong, A. Gould, T. Sumi, V. Batista, P. Fouqué, S. Brillant, S. Dieters, C. Coutures, J. Greenhill, I. Bond, T. Nagayama, A. Udalski, E. Pompei, D. E. A. Nürnberger, and J. B. Le Bouquin, "A frozen super-Earth orbiting a star at the bottom of the main sequence," *Astronomy and Astrophysics*, vol. 540, p. A78, Apr. 2012.

[20] D. P. Bennett, I. A. Bond, A. Udalski, T. Sumi, F. Abe, A. Fukui, K. Furusawa, J. B. Hearnshaw, S. Holderness, Y. Itow, K. Kamiya, A. V. Korpela, P. M. Kilmartin, W. Lin, C. H. Ling, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, M. Nagaya, T. Okumura, K. Ohnishi, Y. C. Perrott, N. J. Rattenbury, T. Sako, T. Saito, S. Sato, L. Skuljan, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, P. C. M. Yock, M. Kubiak, M. K. Szymański, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, V. Batista, J. P. Beaulieu, S. Brillant, A. Cassan, P. Fouqué, P. Kervella, D. Kubas, and J. B. Marquette, "A Low-Mass Planet with a Possible Sub-Stellar-Mass Host in Microlensing Event MOA-2007-BLG-192," *The Astrophysical Journal*, vol. 684, no. 1, pp. 663–683, Sep. 2008.

[21] D. P. Bennett, S. H. Rhie, S. Nikolaev, B. S. Gaudi, A. Udalski, A. Gould, G. W. Christie, D. Maoz, S. Dong, J. McCormick, M. K. Szymański, P. J. Tristram, B. Macintosh, K. H. Cook, M. Kubiak, G. Pietrzyński, I. Soszyński, O. Szewczyk, K. Ulaczyk, Ł. Wyrzykowski, OGLE Collaboration, D. L. DePoy, C. Han, S. Kaspi, C.-U. Lee, F. Mallia, T. Natusch, B.-G. Park, R. W. Pogge, D. Polishook, μFUN Collaboration, F. Abe, I. A. Bond, C. S. Botzler, A. Fukui, J. B. Hearnshaw, Y. Itow, K. Kamiya, A. V. Korpela, P. M. Kilmartin, W. Lin, J. Ling, K. Masuda, Y. Matsubara, M. Motomura, Y. Muraki, S. Nakamura, T. Okumura, K. Ohnishi, Y. C. Perrott, N. J. Rattenbury, T. Sako, T. Saito, S. Sato, L. Skuljan, D. J. Sullivan, T. Sumi, W. L. Sweatman, P. C. M. Yock, MOA Collaboration, M. Albrow, A. Allan, J.-P. Beaulieu, D. M. Bramich, M. J. Burgdorf, C. Coutures, M. Dominik, S. Dieters, P. Fouqué, J. Greenhill, K. Horne, C. Snodgrass, I. Steele, Y. Tsapras, F. t. PLANET, RoboNet Collaborations, B. Chaboyer, A. Crocker, and S. Frank, "Masses and Orbital Constraints for the OGLE-2006-BLG-109Lb,c Jupiter/Saturn Analog Planetary System," *The Astrophysical Journal*, vol. 713, pp. 837–855, Apr. 2010.

[22] B. S. Gaudi, D. P. Bennett, A. Udalski, A. Gould, G. W. Christie, D. Maoz, S. Dong, J. McCormick, M. K. Szymański, P. J. Tristram, S. Nikolaev, B. Paczyński, M. Kubiak, G. Pietrzyński, I. Soszyński, O. Szewczyk, K. Ulaczyk, Ł. Wyrzykowski, OGLE Collaboration, D. L. DePoy, C. Han, S. Kaspi, C.-U. Lee, F. Mallia, T. Natusch, R. W. Pogge, B.-G. Park, μ-Fun Collabortion, F. Abe, I. A. Bond, C. S. Botzler, A. Fukui, J. B. Hearnshaw, Y. Itow, K. Kamiya, A. V. Korpela, P. M. Kilmartin, W. Lin, K. Masuda, Y. Matsubara, M. Motomura, Y. Muraki, S. Nakamura, T. Okumura, K. Ohnishi, N. J. Rattenbury, T. Sako, T. Saito, S. Sato, L. Skuljan, D. J. Sullivan, T. Sumi, W. L. Sweatman, P. C. M. Yock, MOA Collaboration, M. D. Albrow, A. Allan, J.-P. Beaulieu, M. J. Burgdorf, K. H. Cook, C. Coutures, M. Dominik, S. Dieters, P. Fouqué, J. Greenhill, K. Horne, I. Steele, Y. Tsapras, Planet Collaboration, RoboNet Collaborations, B. Chaboyer, A. Crocker, S. Frank, and B. Macintosh, "Discovery of a Jupiter/Saturn Analog with Gravitational Microlensing," *Science*, vol. 319, pp. 927–, 2008.

[23] S. Dong, A. Gould, A. Udalski, J. Anderson, G. W. Christie, B. S. Gaudi, OGLE Collaboration, M. Jaroszyński, M. Kubiak, M. K. Szymański, G. Pietrzyński, I. Soszyński, O. Szewczyk, K. Ulaczyk, Ł. Wyrzykowski, μFUN Collaboration, D. L. DePoy, D. B. Fox, A. Gal-Yam, C. Han, S. Lépine, J. McCormick, E. Ofek, B.-G. Park, R. W. Pogge, MOA Collaboration, F. Abe, D. P. Bennett, I. A. Bond, T. R. Britton, A. C. Gilmore, J. B. Hearnshaw, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, K. Masuda, Y. Matsubara, M. Motomura, Y. Muraki, S. Nakamura, K. Ohnishi, C. Okada, N. Rattenbury, T. Saito, T. Sako, M. Sasaki, D. Sullivan, T. Sumi, P. J. Tristram, T. Yanagisawa, P. C. M. Yock, T. Yoshoika, PLANET/RoboNet Collaborations, M. D. Albrow, J. P. Beaulieu, S. Brillant, H. Calitz, A. Cassan, K. H. Cook, C. Coutures, S. Dieters, D. D. Prester, J. Donatowicz, P. Fouqué, J. Greenhill, K. Hill, M. Hoffman, K. Horne, U. G. Jørgensen, S. Kane, D. Kubas, J. B. Marquette, R. Martin, P. Meintjes, J. Menzies,

K. R. Pollard, K. C. Sahu, C. Vinter, J. Wambsganss, A. Williams, M. Bode, D. M. Bramich, M. Burgdorf, C. Snodgrass, I. Steele, V. Doublier, and C. Foellmi, "OGLE-2005-BLG-071Lb, the Most Massive M Dwarf Planetary Companion?" *The Astrophysical Journal*, vol. 695, pp. 970–987, Apr. 2009.

[24] A. Gould, A. Udalski, D. An, D. P. Bennett, A.-Y. Zhou, S. Dong, N. J. Rattenbury, B. S. Gaudi, P. C. M. Yock, I. A. Bond, G. W. Christie, K. Horne, J. Anderson, K. Z. Stanek, D. L. DePoy, C. Han, J. McCormick, B.-G. Park, R. W. Pogge, S. D. Poindexter, I. Soszyński, M. K. Szymański, M. Kubiak, G. Pietrzyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, B. Paczyński, D. M. Bramich, C. Snodgrass, I. A. Steele, M. J. Burgdorf, M. F. Bode, C. S. Botzler, S. Mao, and S. C. Swaving, "Microlens OGLE-2005-BLG-169 Implies That Cool Neptune-like Planets Are Common," *The Astrophysical Journall*, vol. 644, pp. L37–L40, Jun. 2006.

[25] J.-P. Beaulieu, D. P. Bennett, P. Fouqué, A. Williams, M. Dominik, U. G. Jørgensen, D. Kubas, A. Cassan, C. Coutures, J. Greenhill, K. Hill, J. Menzies, P. D. Sackett, M. Albrow, S. Brillant, J. A. R. Caldwell, J. J. Calitz, K. H. Cook, E. Corrales, M. Desort, S. Dieters, D. Dominis, J. Donatowicz, M. Hoffman, S. Kane, J.-B. Marquette, R. Martin, P. Meintjes, K. Pollard, K. Sahu, C. Vinter, J. Wambsganss, K. Woller, K. Horne, I. Steele, D. M. Bramich, M. Burgdorf, C. Snodgrass, M. Bode, A. Udalski, M. K. Szymański, M. Kubiak, T. Więckowski, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, B. Paczyński, F. Abe, I. A. Bond, T. R. Britton, A. C. Gilmore, J. B. Hearnshaw, Y. Itow, K. Kamiya, P. M. Kilmartin, A. V. Korpela, K. Masuda, Y. Matsubara, M. Motomura, Y. Muraki, S. Nakamura, C. Okada, K. Ohnishi, N. J. Rattenbury, T. Sako, S. Sato, M. Sasaki, T. Sekiguchi, D. J. Sullivan, P. J. Tristram, P. C. M. Yock, and T. Yoshioka, "Discovery of a cool planet of 5.5 Earth masses through gravitational microlensing," *Nature*, vol. 439, pp. 437–440, Jan. 2006.

[26] A. Udalski, M. Jaroszyński, B. Paczyński, M. Kubiak, M. K. Szymański, I. Soszyński, G. Pietrzyński, K. Ulaczyk, O. Szewczyk, Ł. Wyrzykowski, G. W. Christie, D. L. DePoy, S. Dong, A. Gal-Yam, B. S. Gaudi, A. Gould, C. Han, S. Lépine, J. McCormick, B.-G. Park, R. W. Pogge, D. P. Bennett, I. A. Bond, Y. Muraki, P. J. Tristram, P. C. M. Yock, J.-P. Beaulieu, D. M. Bramich, S. W. Dieters, J. Greenhill, K. Hill, K. Horne, and D. Kubas, "A Jovian-Mass Planet in Microlensing Event OGLE-2005-BLG-071," *The Astrophysical Journall*, vol. 628, pp. L109–L112, Aug. 2005.

[27] D. P. Bennett, J. Anderson, I. A. Bond, A. Udalski, and A. Gould, "Identification of the OGLE-2003-BLG-235/MOA-2003-BLG-53 Planetary Host Star," *The Astrophysical Journall*, vol. 647, pp. L171–L174, Aug. 2006.

[28] A. Gould, A. Udalski, D. An, D. P. Bennett, A.-Y. Zhou, S. Dong, N. J. Rattenbury, B. S. Gaudi, P. C. M. Yock, I. A. Bond, G. W. Christie, K. Horne, J. Anderson, K. Z. Stanek, D. L. DePoy, C. Han, J. McCormick, B.-G. Park, R. W. Pogge, S. D. Poindexter, I. Soszyński, M. K. Szymański, M. Kubiak, G. Pietrzyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, B. Paczyński, D. M. Bramich, C. Snodgrass, I. A. Steele, M. J. Burgdorf, M. F. Bode, C. S. Botzler, S. Mao, and S. C. Swaving, "Microlens OGLE-2005-BLG-169 Implies That Cool Neptune-like Planets Are Common," *The Astrophysical Journal*, vol. 644, p. L37, Jun. 2006.

[29] B. S. Gaudi, J. Beaulieu, D. P. Bennett, I. A. Bond, S. Dong, A. Gould, C. Han, B.-G. Park, and S. Takahiro, "The Demographics of Extrasolar Planets Beyond the Snow Line with Ground-based Microlensing Surveys," *Astro2010: The Astronomy and Astrophysics Decadal Survey*, vol. 2010, p. 85, 2009.

[30] A. Gould, S. Dong, B. S. Gaudi, A. Udalski, I. A. Bond, J. Greenhill, R. A. Street, M. Dominik, T. Sumi, M. K. Szymański, C. Han, W. Allen, G. Bolt, M. Bos, G. W. Christie, D. L. DePoy, J. Drummond, J. D. Eastman, A. Gal-Yam, D. Higgins, J. Janczak, S. Kaspi, S. Kozłowski, C.-U. Lee, F. Mallia, A. Maury, D. Maoz, J. McCormick, L. A. G. Monard, D. Moorhouse, N. Morgan, T. Natusch, E. O. Ofek, B.-G. Park, R. W. Pogge, D. Polishook, R. Santallo, A. Shporer, O. Spector, G. Thornley, J. C. Yee, μFUN Collaboration, M. Kubiak, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, R. Poleski, OGLE Collaboration, F. Abe, D. P. Bennett, C. S. Botzler, D. Douchin, M. Freeman, A. Fukui, K. Furusawa, J. B. Hearnshaw, S. Hosaka, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, S. Makita, K. Masuda, Y. Matsubara, N. Miyake, Y. Muraki, M. Nagaya, K. Nishimoto, K. Ohnishi, T. Okumura, Y. C. Perrott, L. Philpott, N. Rattenbury, T. Saito, T. Sako, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, E. von Seggern, P. C. M. Yock, MOA Collaboration, M. Albrow, V. Batista, J. P. Beaulieu, S. Brillant, J. Caldwell, J. J. Calitz, A. Cassan, A. Cole, K. Cook, C. Coutures, S. Dieters, D. Dominis Prester, J. Donatowicz, P. Fouqué, K. Hill, M. Hoffman, F. Jablonski, S. R. Kane, N. Kains, D. Kubas, J.-B. Marquette, R. Martin, E. Martioli, P. Meintjes, J. Menzies, E. Pedretti, K. Pollard, K. C. Sahu, C. Vinter, J. Wambsganss, R. Watson, A. Williams, M. Zub, PLANET Collaboration, A. Allan, M. F. Bode, D. M. Bramich, M. J. Burgdorf, N. Clay, S. Fraser, E. Hawkins, K. Horne, E. Kerins, T. A. Lister, C. Mottram, E. S. Saunders, C. Snodgrass, I. A. Steele, Y. Tsapras, RoboNet Collaboration, U. G. Jørgensen, T. Anguita, V. Bozza, S. Calchi Novati, K. Harpsøe, T. C. Hinse, M. Hundertmark, P. Kjærgaard, C. Liebig, L. Mancini, G. Masi, M. Mathiasen, S. Rahvar, D. Ricci, G. Scarpetta, J. Southworth, J. Surdej, C. C. Thöne, and MiNDSTEp Consortium, "Frequency of Solar-like Systems and of Ice and Gas Giants Beyond the Snow Line from High-magnification Microlensing Events in 2005-2008," *The Astrophysical Journal*, vol. 720, pp. 1073–1089, Sep. 2010.

[31] A. Cassan, D. Kubas, J.-P. Beaulieu, M. Dominik, K. Horne, J. Greenhill, J. Wambsganss, J. Menzies, A. Williams, U. G. Jørgensen, A. Udalski, D. P. Bennett, M. D. Albrow, V. Batista, S. Brillant, J. A. R. Caldwell, A. Cole, C. Coutures, K. H. Cook, S. Dieters, D. D. Prester, J. Donatowicz, P. Fouqué, K. Hill, N. Kains, S. Kane, J.-B. Marquette, R. Martin, K. R. Pollard, K. C. Sahu, C. Vinter, D. Warren, B. Watson, M. Zub, T. Sumi, M. K. Szymański, M. Kubiak, R. Poleski, I. Soszynski, K. Ulaczyk, G. Pietrzyński, and Ł. Wyrzykowski, "One or more bound planets per Milky Way star from microlensing observations," *Nature*, vol. 481, pp. 167–169, Jan. 2012.

[32] M. D. Albrow, J. An, J.-P. Beaulieu, J. A. R. Caldwell, D. L. DePoy, M. Dominik, B. S. Gaudi, A. Gould, J. Greenhill, K. Hill, S. Kane, R. Martin, J. Menzies, R. M. Naber, J.-W. Pel, R. W. Pogge, K. R. Pollard, P. D. Sackett, K. C. Sahu, P. Vermaak, P. M. Vreeswijk, R. Watson, and A. Williams, "Limits on the Abundance of Galactic Planets From 5 Years of PLANET Observations," *The Astrophysical Journal Letters*, vol. 556, pp. L113–L116, Aug. 2001.

[33] T. Sumi, K. Kamiya, D. P. Bennett, I. A. Bond, F. Abe, C. S. Botzler, A. Fukui, K. Furusawa, J. B. Hearnshaw, Y. Itow, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, K. Masuda, Y. Matsubara, N. Miyake, M. Motomura, Y. Muraki, M. Nagaya, S. Nakamura, K. Ohnishi, T. Okumura, Y. C. Perrott, N. Rattenbury, T. Saito, T. Sako, D. J. Sullivan, W. L. Sweatman, P. J. Tristram, A. Udalski, M. K. Szymański, M. Kubiak, G. Pietrzyński, R. Poleski, I. Soszyński, Ł. Wyrzykowski, K. Ulaczyk, and Microlensing Observations in Astrophysics (MOA) Collaboration, "Unbound or distant planetary mass population detected by gravitational microlensing," *Nature*, vol. 473, pp. 349–352, May 2011.

[34] N. J. Rattenbury, I. A. Bond, J. Skuljan, and P. C. M. Yock, "Planetary microlensing at high magnification," *Monthly Notices of the Royal Astronomical Society*, vol. 335, pp. 159–169, Sep. 2002.

[35] B. Paczynski, "Gravitational Microlensing: Black Holes, Planets; OGLE, VLTI, HST and Space Probes," *ArXiv Astrophysics e-prints, astro-ph/0306564*, Jun. 2003.

[36] B. S. Gaudi, *Microlensing by Exoplanets*, 2011, pp. 79–110.

[37] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, 3rd ed. New York, NY, USA: Cambridge University Press, 2007.

[38] H. Witt, "Investigation of high amplification events in light curves of gravitationally lensed quasars," *Astronomy and Astrophysics*, vol. 236, pp. 311–322, 1990.

[39] S. H. Rhie, D. P. Bennett, M. Clampin, K. H. Cook, A. J. Drake, A. Gould, K. Horne, S. D. Horner, D. C. Jewitt, G. I. Langston, T. R. Lauer, A. Lumsdaine, D. Minniti, S. J. Peale, M. Shao, R. L. Stevenson, D. Tenerelli, D. Tytler, and N. J. Woolf, "The Galactic Exoplanet Survey Telescope (GEST): A Search for Extra-Solar Planets via Gravitational Microlensing and Transits," *American Astronomical Society*, vol. 32, p. 1053, Oct. 2000.

[40] S. H. Rhie, "How Cumbersome is a Tenth Order Polynomial?: The Case of Gravitational Triple Lens Equation," *ArXiv Astrophysics e-prints, astro-ph/0202294*, Feb. 2002.

[41] S. Dong, I. A. Bond, A. Gould, S. Kozłowski, N. Miyake, B. S. Gaudi, D. P. Bennett, F. Abe, A. C. Gilmore, A. Fukui, K. Furusawa, J. B. Hearnshaw, Y. Itow, K. Kamiya, P. M. Kilmartin, A. Korpela, W. Lin, C. H. Ling, K. Masuda, Y. Matsubara, Y. Muraki, M. Nagaya, K. Ohnishi, T. Okumura, Y. C. Perrott, N. Rattenbury, T. Saito, T. Sako, S. Sato, L. Skuljan, D. J. Sullivan, T. Sumi, W. Sweatman, P. J. Tristram, P. C. M. Yock, T. M. Collaboration, G. Bolt, G. W. Christie, D. L. DePoy, C. Han, J. Janczak, C.-U. Lee, F. Mallia, J. McCormick, B. Monard, A. Maury, T. Natusch, B.-G. Park, R. W. Pogge, R. Santallo, K. Z. Stanek, T. . Collaboration, A. Udalski, M. Kubiak, M. K. Szymański, G. Pietrzyński, I. Soszyński, O. Szewczyk, Ł. Wyrzykowski, K. Ulaczyk, and T. O. Collaboration, "Microlensing Event MOA-2007-BLG-400: Exhuming the Buried Signature of a Cool, Jovian-Mass Planet," *The Astrophysical Journal*, vol. 698, p. 1826, Jun. 2009.

[42] A. Gould, "Resolution of the MACHO-LMC-5 Puzzle: The Jerk-Parallax Microlens Degeneracy," *The Astrophysical Journal*, vol. 606, p. 319, May 2004.

[43] S. Dong, D. L. DePoy, B. S. Gaudi, A. Gould, C. Han, B.-G. Park, R. W. Pogge, A. Udalski, O. Szewczyk, M. Kubiak, M. K. Szymański, G. Pietrzyński, I. Soszyński, Ł. Wyrzykowski, and K. Żebruń, "Planetary Detection Efficiency of the Magnification 3000 Microlensing Event OGLE-2004-BLG-343," *The Astrophysical Journal*, vol. 642, p. 842, May 2006.

[44] H. M. Fatahalian K, "A Closer Look at GPUs." *Communications of the ACM*, vol. 51(10), pp. 50–57, Oct. 2008.

[45] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007. [Online]. Available: http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x

[46] NVIDIA, "NVIDIA CUDA," http://www.nvidia.com/object/cuda_home_new.html, [Accessed: 1-12-2012].

[47] P. Messmer, P. J. Mullowney, and B. E. Granger, "GPULib: GPU Computing in High-Level Languages." *Computing in Science and Engineering*, vol. 10, no. 5, pp. 70–73, 2008. [Online]. Available: http://dblp.uni-trier.de/db/journals/cse/cse10.html#MessmerMG08

[48] A. Leist, D. P. Playne, and K. A. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 18, pp. 2400–2437, 25 December 2009, CSTN-065.

[49] M. G. B. Johnson, D. P. Playne, and K. A. Hawick, "Data-Parallelism and GPUs for Lattice Gas Fluid Simulations," in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*. Las Vegas, USA: CSREA, 12-15 July 2010, pp. 210–216, pDP4521.

[50] K. A. Hawick and D. P. Playne, "Numerical Simulation of the Complex Ginzburg-Landau Equation on GPUs with CUDA," in *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, no. CSTN-070. Innsbruck, Austria: IASTED, 15-17 February 2011, pp. 39–45.

[51] A. G. Mark J Stock, "Toward efficient GPU-accelerated N-body simulations," *AIAA paper*, vol. 608, pp. 7–10, 2008.

[52] N. F. Bate, C. J. Fluke, B. R. Barsdell, H. Garsden, and G. F. Lewis, "Computational advances in gravitational microlensing: A comparison of CPU, GPU, and parallel, large data codes," *New Astronomy*, vol. 15, pp. 726–734, Nov. 2010.

[53] C. J. Fluke, D. G. Barnes, B. R. Barsdell, and A. H. Hassan, "Astrophysical Supercomputing with GPUs: Critical Decisions for Early Adopters," *pasa*, vol. 28, pp. 15–27, Jan. 2011.

[54] A. C. Thompson, C. J. Fluke, D. G. Barnes, and B. R. Barsdell, "Teraflop per second gravitational lensing ray-shooting using graphics processing units," *New Astronomy*, vol. 15, pp. 16–23, Jan. 2010.

[55] G. Kavanagh, M. Lewis, and B. L. Massingill, "GPGPU Planetary Simulations with CUDA." in *CSC*, H. R. Arabnia, Ed. CSREA Press, 2008, pp. 180–185. [Online]. Available: http://dblp.uni-trier.de/db/conf/csc/csc2008.html#KavanaghLM08

[56] K. M. S. Ha, J. Pi, "GPU-Accelerated First-Order Scattering Simulation for X-ray CT Image Reconstruction," *Proc. 2nd International Conference on Image Formation in X-ray Computed Tomography*, July 2012.

[57] K. M. W. Xu, S. Ha, "Database-Assisted Low-Dose CT Image Restoration," *Medical Physics*, vol. 3(40):031109, 2013.

[58] NVIDIA, "NVIDIA CUDA Computational Finance," http://www.nvidia.com/object/computational_finance.html., [Accessed: 05-10-2013].

[59] D. Playne and K. Hawick, "Data Parallel Three-Dimensional Cahn-Hilliard Field Equation Simulation on GPUs with CUDA," in *Proc. 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'09)*. Las vegas, USA: WorldComp, 13-16 July 2009, pp. 104–110.

[60] D. P. Playne, M. G. B. Johnson, and K. A. Hawick, "Benchmarking GPU Devices with N-Body Simulations," in *Proc. 2009 International Conference on Computer Design (CDES 09) July, Las Vegas, USA.* Las Vegas, USA: WorldComp, 13-16 July 2009, pp. 150–156.

[61] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[62] A. Gould, "The Hollywood Strategy for Microlensing Detection of Planets." in *Variables Stars and the Astrophysical Returns of the Microlensing Surveys*, R. Ferlet, J.-P. Maillard, and B. Raban, Eds., 1997, p. 125.

[63] H. J. Witt and S. Mao, "On the Minimum Magnification Between Caustic Crossings for Microlensing by Binary and Multiple Stars," *Astrophysical Journal Letters*, vol. 447, p. L105, 1995.

[64] P. Yock, "Controlling the Moire Effect in Gravitational Microlensing," Technical Report, Aug. 2007.

[65] L. Philpott, "Creating theoretical microlensing light curves from magnification maps," Research Report, June 2005.

[66] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS) - Special issue on uniform random number generation*, 1998.

[67] V. Podlozhnyuk, "Parallel Mersenne Twister," *NVIDIA Corporation*, 2008.

[68] T. N. Makoto Matsumoto, "Dynamic Creation of Pseudorandom Number Generators," *Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer*, pp. 56–69, 2000.

[69] B. S. Gaudi, H.-Y. Chang, and C. Han, "Probing Structures of Distant Extrasolar Planets with Microlensing," *The Astrophysical Journal*, vol. 586, pp. 527–539, Mar. 2003.

[70] Intel, "Intel Integrated Performance Primitives," http://software.intel.com/en-us/intel-ipp, [Accessed: 20-11-2013].

[71] D. P. Bennett, "An Efficient Method for Modeling High-magnification Planetary Microlensing Events," *The Astrophysical Journal*, vol. 716, pp. 1408–1422, Jun. 2010.

[72] A. Gould, "Hexadecapole Approximation in Planetary Microlensing," *The Astrophysical Journal*, vol. 681, p. 1593, Jul. 2008.

[73] M. Dominik, "Adaptive contouring - an efficient way to calculate microlensing light curves of extended sources," *Monthly Notices of the Royal Astronomical Society*, vol. 377, p. 1679, Jun. 2007.

[74] Intel, "Intel xeon phi," http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html, [Accessed: 10-12-2012].

[75] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," Tech. Rep., Accessed: 1-12-2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf